W09: Amortisation

CS3230 AY21/22 Sem 2

Table of Contents

Click on the link to jump to the relevant sections!

Amortisation Introduction

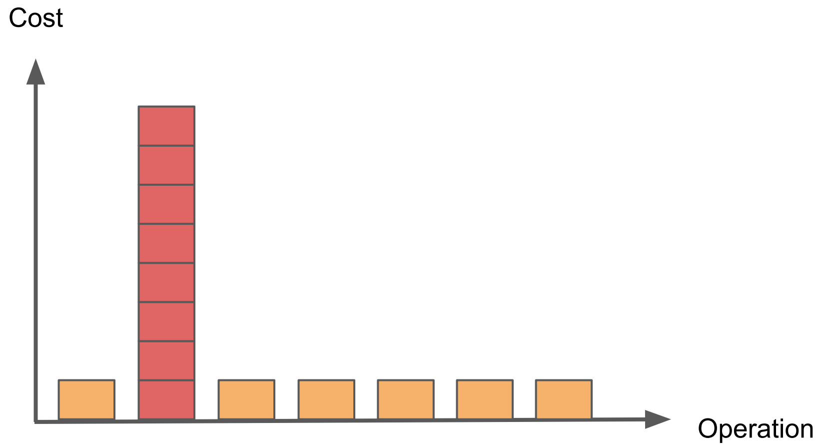
Motivation

- In analysing a sequence of operations on data structure, the easy way out is for you to always take the **worst case running time** of **each** operation

Motivation

- In analysing a sequence of operations on data structure, the easy way out is for you to always take the **worst case running time** of **each** operation

- But! You might be **overcounting**! What if some operations actually take **way less** cost?



Amortisation

Common techniques:

- *Aggregate* method
- *Accounting* method (Banker)
- *Potential* method (Physicist)

Amortisation

Common techniques:

- *Aggregate* method
- *Accounting* method (Banker)
- *Potential* method (Physicist)

Goal of Amortisation:
To give a guarantee of average performance of each operation **over all operations** so far

Amortisation

Common techniques:

- *Aggregate* method
- *Accounting* method (Banker)
- *Potential* method (Physicist)

Note:

- Average-Case analysis is averaging over the random inputs
- Amortisation here has **no randomness**, it averages over total cost of all operations

Goal of Amortisation:

To give a guarantee of average performance of each operation **over all operations** so far

Amortisation

Common techniques:

- *Aggregate* method (not discussed in this tut. Not as flexible as the other two)
- *Accounting* method (Banker)
- *Potential* method (Physicist)

Note:

- Average-Case analysis is averaging over the random inputs
- Amortisation here has **no randomness**, it averages over total cost of all operations

Goal of Amortisation:

To give a guarantee of average performance of each operation **over all operations** so far

Accounting Method

Accounting Method

Analogy -- Living Expenses:

Accounting Method

Analogy -- Living Expenses:

- Let’s say you have daily allowance - $15

Accounting Method

Analogy -- Living Expenses:

- Let’s say you have daily allowance - $15
- Every day, you need to spend a bit -- maybe for food, or other needs
  - But you **set aside some money every day too** - $5

I am fully aware rent is not this cheap. Numbers are for illustration purposes only

Accounting Method

Analogy -- Living Expenses:

- Let's say you have daily allowance - $15
- Every day, you need to spend a bit -- maybe for food, or other needs
  - But you **set aside some money every day too** - $5
- At the end of the month: pay rent for $120!
  - Expensive -- it's more than your daily allowance :<

I am fully aware rent is not this cheap. Numbers are for illustration purposes only

Accounting Method

Analogy -- Living Expenses:

- Let's say you have daily allowance - $15
- Every day, you need to spend a bit -- maybe for food, or other needs
  - But you **set aside some money every day too** - $5
- At the end of the month: pay rent for $120!
  - Expensive -- it's more than your daily allowance :<
  - But! You set aside some money every day. Pay the rent with that!

I am fully aware rent is not this cheap. Numbers are for illustration purposes only

Accounting Method

Analogy -- Living Expenses:

- Let's say you have daily allowance - $15
- Every day, you need to spend a bit -- maybe for food, or other needs
  - But you **set aside some money every day too** - $5
- At the end of the month: pay rent for $120!
  - Expensive -- it's more than your daily allowance :<
  - But! You set aside some money every day. Pay the rent with that!
  - Now rent is "cheap" -- because your savings paid for it and it doesn't take anything from today's daily allowance

Accounting Method

- Idea: Impose an **extra charge** on **inexpensive operations**, and use it to pay for a later expensive operation

Accounting Method

- Idea: Impose an **extra charge** on **inexpensive operations**, and use it to pay for a later expensive operation
- Charge *i*-th operation a fictitious **amortised cost** $c(i)$, where \$1 = 1 unit of time [analogy: your daily allowance]

Accounting Method

- Idea: Impose an **extra charge** on **inexpensive operations**, and use it to pay for a later expensive operation
- Charge *i*-th operation a fictitious **amortised cost** $c(i)$, where $1 = 1 unit of time [analogy: your daily allowance]
- The amortised cost will be used for that operation in **two** ways:
  - Immediately use a portion to pay for the **true (actual) cost** $t(i)$ [analogy: daily food]
  - Defer the cost by storing it in a **bank** [analogy: money you set aside]

Accounting Method

- Idea: Impose an **extra charge** on **inexpensive operations**, and use it to pay for a later expensive operation
- Charge *i*-th operation a fictitious **amortised cost** $c(i)$, where \$1 = 1 unit of time [analogy: your daily allowance]
- The amortised cost will be used for that operation in **two** ways:
  - Immediately use a portion to pay for the **true (actual) cost** $t(i)$ [analogy: daily food]
  - Defer the cost by storing it in a **bank** [analogy: money you set aside]
- When the expensive operation comes, use money from the bank instead! [analogy: paying rent with savings]

Living Expenses

Scenario: You just started life. Bank contains no money

	What you have to pay for today	Daily Allowance	How much you save
Day	*Actual Cost*	*Amortised Cost*	*To the bank*

Living Expenses

Day 1: You bought Chicken Rice for $10. And you saved $5

	What you have to pay for today	Daily Allowance	How much you save
Day	*Actual Cost*	*Amortised Cost*	*To the bank*
1	*$10*	*$15*	*$5*

Living Expenses

Day 2: You try another food that also costs $10.
Also saved $5

Day	Actual Cost (What you have to pay for today)	Amortised Cost (Daily Allowance)	To the bank (How much you save)
1	$10	$15	$5
2	$10	$15	$5

Living Expenses

Day 3: Not hungry, but bought groceries for $5

	What you have to pay for today	Daily Allowance	How much you save
Day	*Actual Cost*	*Amortised Cost*	*To the bank*
1	*$10*	*$15*	*$5*
2	*$10*	*$15*	*$5*
3	*$5*	*$15*	*$10*

Living Expenses

Day 30: After a sequence of operation, let’s say you have $150 in the bank!

	What you have to pay for today	Daily Allowance	How much you save
Day	*Actual Cost*	*Amortised Cost*	*To the bank*
1	*$10*	*$15*	*$5*
2	*$10*	*$15*	*$5*
3	*$5*	*$15*	*$10*
...	...	...	...
30	*$10*	*$15*	*$5*

Living Expenses

Day 31: Rent payment of $120 comes! You still need to buy food for $10

	What you have to pay for today	Daily Allowance	How much you save
Day	*Actual Cost*	*Amortised Cost*	*To the bank*
1	*$10*	*$15*	*$5*
2	*$10*	*$15*	*$5*
3	*$5*	*$15*	*$10*
...	...	...	...
30	*$10*	*$15*	*$5*
31	*$120 + $10*	*$15*	

Living Expenses

Use the saved-up money first

	What you have to pay for today	Daily Allowance	How much you save
Day	*Actual Cost*	*Amortised Cost*	*To the bank*
1	*$10*	*$15*	*$5*
2	*$10*	*$15*	*$5*
3	*$5*	*$15*	*$10*
...	...	...	...
30	*$10*	*$15*	*$5*
31	*$120 + $10*	*$15*	

Living Expenses

The rent is "cheap" (compared to daily allowance) -- the thrifty you in the past paid for it! So focus on today's expenses

	What you have to pay for today	Daily Allowance	How much you save
Day	*Actual Cost*	*Amortised Cost*	*To the bank*
1	*$10*	*$15*	*$5*
2	*$10*	*$15*	*$5*
3	*$5*	*$15*	*$10*
...	...	...	...
30	*$10*	*$15*	*$5*
31	*$120 + $10*	*$15*	

Living Expenses

Only costs $10 like usual, but you are ready to pay $15 -- $5 to the bank!

	What you have to pay for today	Daily Allowance	How much you save
Day	*Actual Cost*	*Amortised Cost*	*To the bank*
1	*$10*	*$15*	*$5*
2	*$10*	*$15*	*$5*
3	*$5*	*$15*	*$10*
...	...	...	...
30	*$10*	*$15*	*$5*
31	*$120 + $10*	*$15*	*$5*

Living Expenses

Computing Total Costs: It's okay to have excess money in the bank!

	What you have to pay for today	Daily Allowance	How much you save
Day	*Actual Cost*	*Amortised Cost*	*To the bank*
1	*$10*	*$15*	*$5*
2	*$10*	*$15*	*$5*
3	*$5*	*$15*	*$10*
...	...	...	...
30	*$10*	*$15*	*$5*
31	*$120 + $10*	*$15*	*$5*
TOTAL	*$430*	*$465*	

Suppose 31 days = $310
and rent = $120

Living Expenses

Computing Total Costs: It’s okay to have excess money in the bank!

	What you have to pay for today	Daily Allowance	How much you save
Day	*Actual Cost*	*Amortised Cost*	*To the bank*
1	*$10*	*$15*	*$5*
2	*$10*	*$15*	*$5*
3	*$5*	*$15*	*$10*
...	...	...	...
30	*$10*	*$15*	*$5*
31	*$120 + $10*	*$15*	*$5*
TOTAL	*$430*	*$465*	

Total Actual Cost ≤ Total Amortised Cost

Total Actual Cost ≤ 465 units

As long as bank never goes negative

Living Expenses

Computing Total Costs: It’s okay to have excess money in the bank!

$$\sum_{i=1}^{n} t(i) \le \sum_{i=1}^{n} c(i)$$

Total Actual Cost ≤ Total Amortised Cost

	What you have to pay for today	Daily Allowance	How much you save
Day	*Actual Cost*	*Amortised Cost*	*To the bank*
1	*$10*	*$15*	*$5*
2	*$10*	*$15*	*$5*
3	*$5*	*$15*	*$10*
...	...	...	...
30	*$10*	*$15*	*$5*
31	*$120 + $10*	*$15*	*$5*
TOTAL	*$430*	*$465*	

Total Actual Cost ≤ Total Amortised Cost
Total Actual Cost ≤ 465 units
As long as bank never goes negative

Living Expenses

Computing Total Costs: It’s okay to have excess money in the bank!

$$\sum_{i=1}^{n} t(i) \leq \sum_{i=1}^{n} c(i)$$

Total Actual Cost ≤ Total Amortised Cost
If Total Amortised Cost = O(n),
Then Total Actual Cost = O(n)

	What you have to pay for today	Daily Allowance	How much you save
Day	*Actual Cost*	*Amortised Cost*	*To the bank*
1	*$10*	*$15*	*$5*
2	*$10*	*$15*	*$5*
3	*$5*	*$15*	*$10*
...	...	...	...
30	*$10*	*$15*	*$5*
31	*$120 + $10*	*$15*	*$5*
TOTAL	*$430*	*$465*	

Total Actual Cost ≤ Total Amortised Cost
Total Actual Cost ≤ 465 units
As long as bank never goes negative

Question 1: T/F on Amortisation

Question 1

Which of the following statements is **false**?

- The amortized cost for insert in dynamic tables is $\Theta(1)$.
- In the accounting method, the amortized cost $\hat{c}_i$ is always greater than the actual cost c_i of an operation.
- $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \geq 0$ where $\hat{c}_i$ and c_i are the amortized and actual costs of the i-th operation respectively.

Question 1 (Ans)

Option A: True

Charge \$3:

- \$1 for the insert
- \$2 to the bank

- The amortized cost for insert in dynamic tables is $\Theta(1)$.
- In the accounting method, the amortized cost $\hat{c}_i$ is always greater than the actual cost c_i of an operation.
- $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \geq 0$ where $\hat{c}_i$ and c_i are the amortized and actual costs of the i-th operation respectively.

Question 1 (Ans)

- The amortized cost for insert in dynamic tables is $\Theta(1)$.
- In the accounting method, the amortized cost $\hat{c}_i$ is always greater than the actual cost c_i of an operation.
- $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \geq 0$ where $\hat{c}_i$ and c_i are the amortized and actual costs of the i-th operation respectively.

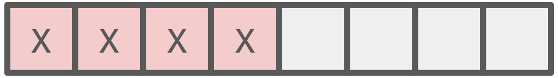
Option A: True

Charge $3:

- $1 for the insert
- $2 to the bank

Insert Amortised

```
Let’s say we *just* doubled the table
```



Note: Ignore costs of allocation and deleting elements. Only count cost of inserting and transferring (demo purposes)

Question 1 (Ans)

- The amortized cost for insert in dynamic tables is $\Theta(1)$.
- In the accounting method, the amortized cost $\hat{c}_i$ is always greater than the actual cost c_i of an operation.
- $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \geq 0$ where $\hat{c}_i$ and c_i are the amortized and actual costs of the i-th operation respectively.

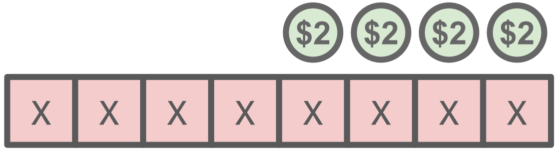
Option A: True

Charge $3:

- $1 for the insert
- $2 to the bank

Insert Amortised

Then let’s say we insert 4 more times. Each one of them stores $2 to the bank



Note: Ignore costs of allocation and deleting elements. Only count cost of inserting and transferring (demo purposes)

- The amortized cost for insert in dynamic tables is $\Theta(1)$.
- In the accounting method, the amortized cost $\hat{c}_i$ is always greater than the actual cost c_i of an operation.
- $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \geq 0$ where $\hat{c}_i$ and c_i are the amortized and actual costs of the i-th operation respectively.

Question 1 (Ans)

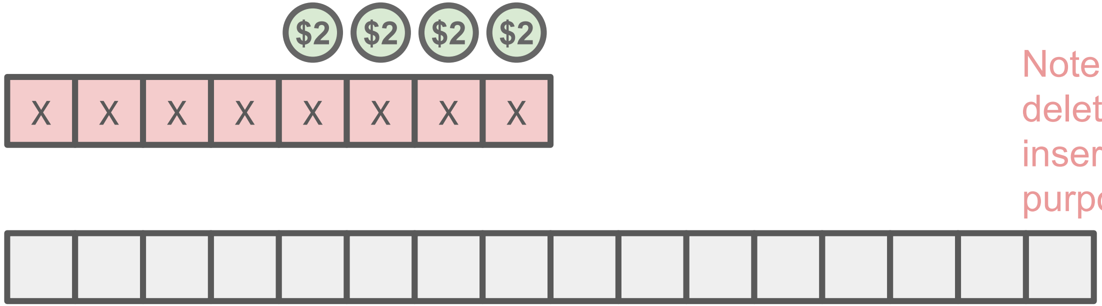
Option A: True

Charge $3:

- $1 for the insert
- $2 to the bank

Insert Amortised

```
When we insert one more time, need to double!
```



Note: Ignore costs of allocation and deleting elements. Only count cost of inserting and transferring (demo purposes)

Question 1 (Ans)

- The amortized cost for insert in dynamic tables is $\Theta(1)$.
- In the accounting method, the amortized cost $\hat{c}_i$ is always greater than the actual cost c_i of an operation.
- $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \geq 0$ where $\hat{c}_i$ and c_i are the amortized and actual costs of the i-th operation respectively.

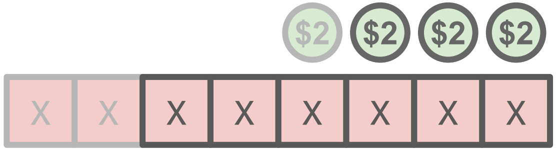
Option A: True

Charge $3:

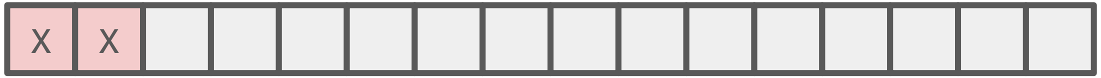
- $1 for the insert
- $2 to the bank

Insert Amortised

Cost of transferring -- paid by the bank. An element pays for two transfers!



Note: Ignore costs of allocation and deleting elements. Only count cost of inserting and transferring (demo purposes)



- The amortized cost for insert in dynamic tables is $\Theta(1)$.
- In the accounting method, the amortized cost $\hat{c}_i$ is always greater than the actual cost c_i of an operation.
- $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \geq 0$ where $\hat{c}_i$ and c_i are the amortized and actual costs of the i-th operation respectively.

Question 1 (Ans)

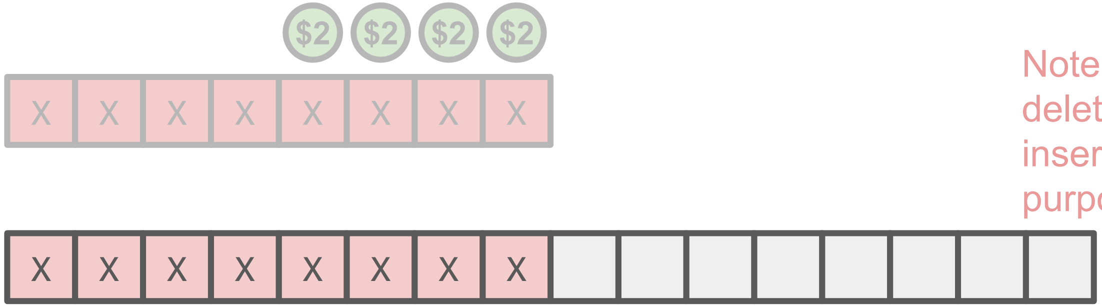
Option A: True

Charge $3:

- $1 for the insert
- $2 to the bank

Insert Amortised

Cost of transferring -- paid by the bank



Note: Ignore costs of allocation and deleting elements. Only count cost of inserting and transferring (demo purposes)

Question 1 (Ans)

- The amortized cost for insert in dynamic tables is $\Theta(1)$.
- In the accounting method, the amortized cost $\hat{c}_i$ is always greater than the actual cost c_i of an operation.
- $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \geq 0$ where $\hat{c}_i$ and c_i are the amortized and actual costs of the i-th operation respectively.

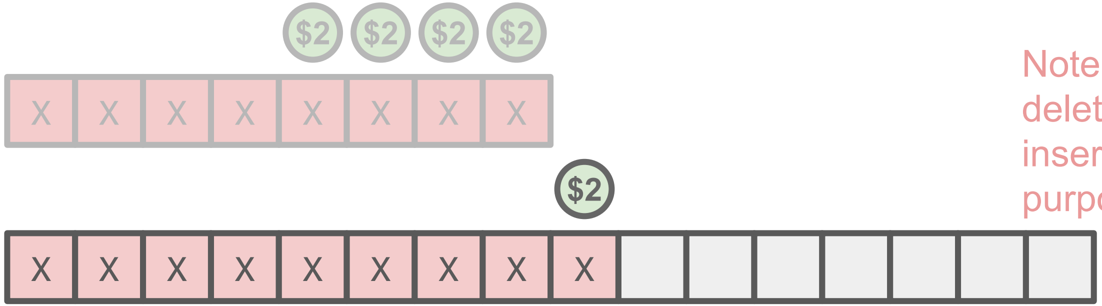
Option A: True

Charge $3:

- $1 for the insert
- $2 to the bank

Insert Amortised

`Do the actual insert!`



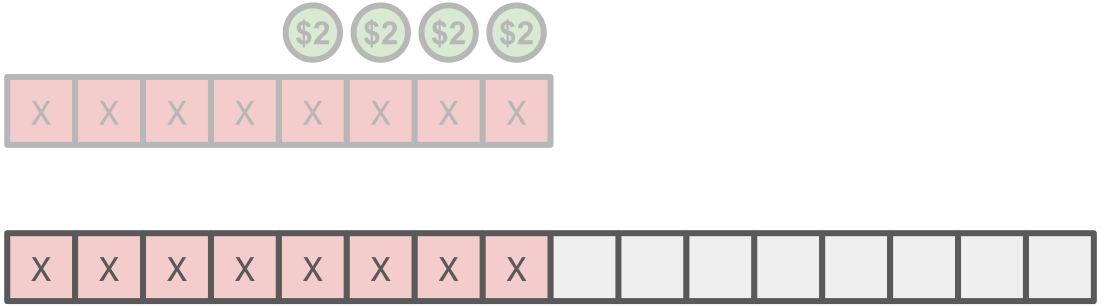
Note: Ignore costs of allocation and deleting elements. Only count cost of inserting and transferring (demo purposes)

Question 1 (Ans)

- The amortized cost for insert in dynamic tables is $\Theta(1)$.
- In the accounting method, the amortized cost $\hat{c}_i$ is always greater than the actual cost c_i of an operation.
- $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \geq 0$ where $\hat{c}_i$ and c_i are the amortized and actual costs of the i-th operation respectively.

Option B: False

Recall insertion example -- our amortised cost for the operation is $3, but the **actual cost is more** than $3! (Probably something like 9 here. Copy 8 elements and insert the actual element)



Question 1 (Ans)

- The amortized cost for insert in dynamic tables is $\Theta(1)$.
- In the accounting method, the amortized cost $\hat{c}_i$ is always greater than the actual cost c_i of an operation.
- $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \geq 0$ where $\hat{c}_i$ and c_i are the amortized and actual costs of the i-th operation respectively.

Option C: True

Total amortised cost must be larger than sum of actual costs

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i$$

$$0 \leq \sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i$$

Question 2:
Fancier Queue (Accounting Method)

Question 2

- Consider a data structure that is based on a queue with these operations:
  - `enqueue(a)` - Add element `a` into the queue
  - `dequeue()` - Dequeue a single element from the queue
  - `delete(k)` - Dequeue `k` elements from the queue
  - `add(A)` - enqueue all elements in `A`

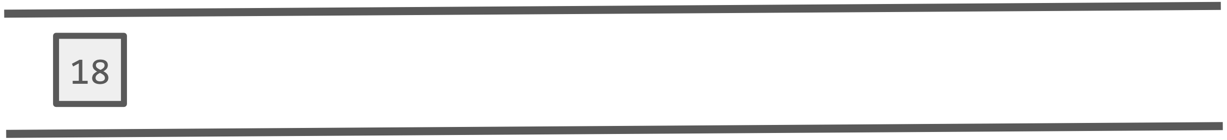
Question 2

```
Start off with an empty queue
```

- `enqueue(a)` - Add element `a` into the queue
- `dequeue()` - Dequeue a single element from the queue
- `delete(k)` - Dequeue `k` elements from the queue
- `add(A)` - enqueue all elements in `A`

Question 2

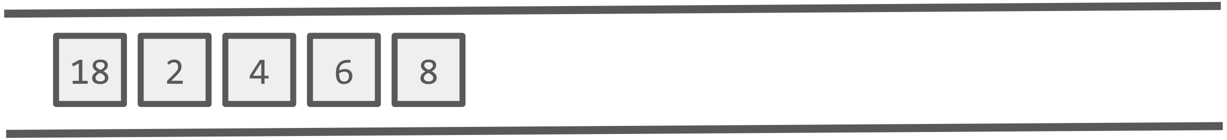
`enqueue(18)`



- `enqueue(a)` - Add element `a` into the queue
- `dequeue()` - Dequeue a single element from the queue
- `delete(k)` - Dequeue `k` elements from the queue
- `add(A)` - enqueue all elements in `A`

Question 2

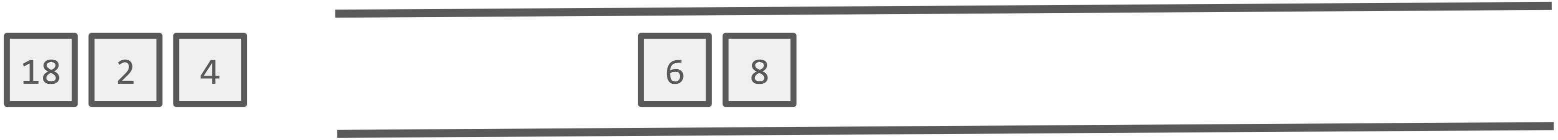
`add([2, 4, 6, 8])`



- `enqueue(a)` - Add element `a` into the queue
- `dequeue()` - Dequeue a single element from the queue
- `delete(k)` - Dequeue `k` elements from the queue
- `add(A)` - enqueue all elements in `A`

Question 2

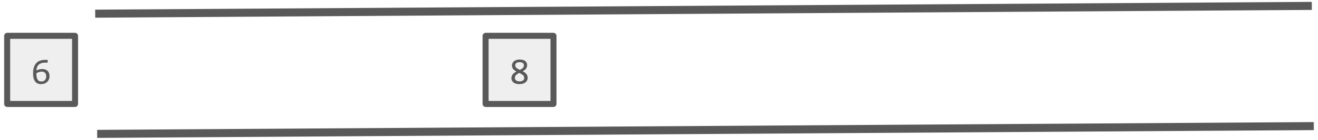
`delete(3)`



- `enqueue(a)` - Add element `a` into the queue
- `dequeue()` - Dequeue a single element from the queue
- `delete(k)` - Dequeue `k` elements from the queue
- `add(A)` - enqueue all elements in `A`

Question 2

`dequeue()`



- `enqueue(a)` - Add element `a` into the queue
- `dequeue()` - Dequeue a single element from the queue
- `delete(k)` - Dequeue `k` elements from the queue
- `add(A)` - enqueue all elements in `A`

Question 2

- Consider a data structure that is based on a queue with these operations:
  - `enqueue(a)` - Add element `a` into the queue
  - `dequeue()` - Dequeue a single element from the queue
  - `delete(k)` - Dequeue `k` elements from the queue
  - `add(A)` - enqueue all elements in `A`
- **Claim:**
  - `enqueue(a), dequeue(), delete(k)` runs in amortised *O(1)* time
  - `add(A)` runs in amortised *O(|A|)* time

Question 2

- Consider a data structure that is based on a queue with these operations:
  - `enqueue(a)` - Add element `a` into the queue
  - `dequeue()` - Dequeue a single element from the queue
  - `delete(k)` - Dequeue `k` elements from the queue
  - `add(A)` - enqueue all elements in `A`
- **Claim:**
  - `enqueue(a), dequeue(), delete(k)` runs in amortised *O(1)* time
  - `add(A)` runs in amortised *O(|A|)* time
- **Using accounting method**, prove these time complexities
  - To use accounting method, you need to state the charge (this is the amortised cost)

Question 2

- Consider a data structure that is based on a queue with four operations:
  - ENQUEUE(a): Add the element a into the queue
  - DEQUEUE(): Dequeue a single element from the queue
  - DELETE(k): Dequeue k elements from the queue
  - ADD(A): Enqueue all elements in A
- **Claim**: ENQUEUE, DEQUEUE and DELETE run in amortized O(1) time while ADD runs in amortized O(|A|) time.
- Using accounting method, can you show that these time complexities are correct?
- (Please state the charge for each operation.)

Question 2 Idea

- Consider a data structure that is based on a queue with these operations:
  - `enqueue(a)` - Add element `a` into the queue
  - `dequeue()` - Dequeue a single element from the queue
  - `delete(k)` - Dequeue `k` elements from the queue
  - `add(A)` - enqueue all elements in `A`
- **Claim:**
  - `enqueue(a), dequeue(), delete(k)` runs in amortised $O(1)$ time
  - `add(A)` runs in amortised $O(|A|)$ time

- Naive analysis of delete is $O(k)$ time -- but we now want amortised $O(1)$ time!

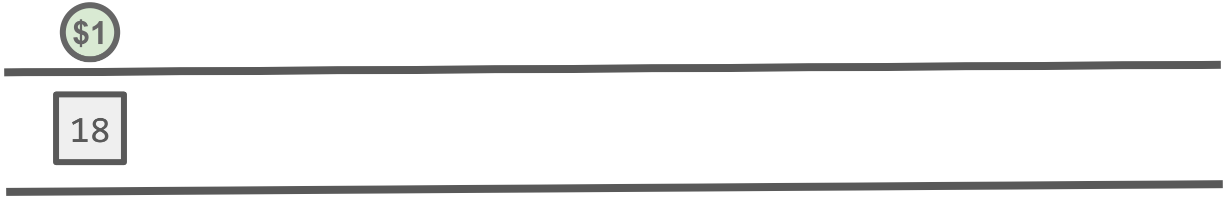
Question 2 Idea

- Consider a data structure that is based on a queue with these operations:
  - `enqueue(a)` - Add element `a` into the queue
  - `dequeue()` - Dequeue a single element from the queue
  - `delete(k)` - Dequeue `k` elements from the queue
  - `add(A)` - enqueue all elements in `A`
- **Claim:**
  - `enqueue(a), dequeue(), delete(k)` runs in amortised $O(1)$ time
  - `add(A)` runs in amortised $O(|A|)$ time

- Naive analysis of delete is $O(k)$ time -- but we now want amortised $O(1)$ time!
- Pre-charge other operations to “pay in advance” for deletion

Question 2

`enqueue(18)`

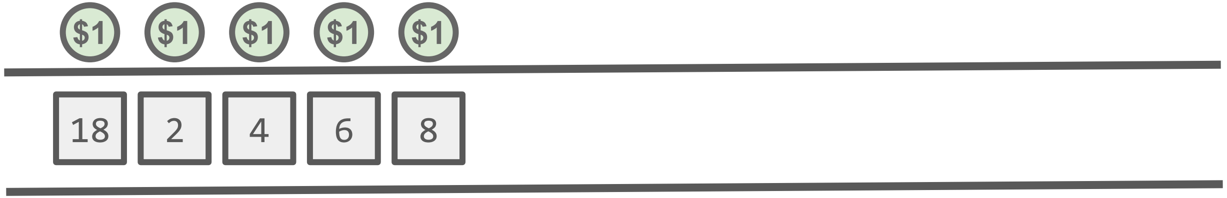


- `enqueue(a)` - Charge $2:
  - $1 for the actual insertion
  - $1 in the bank, for future dequeue

- `enqueue(a)` - Add element `a` into the queue
- `dequeue()` - Dequeue a single element from the queue
- `delete(k)` - Dequeue `k` elements from the queue
- `add(A)` - enqueue all elements in `A`

Question 2

`add([2, 4, 6, 8])`

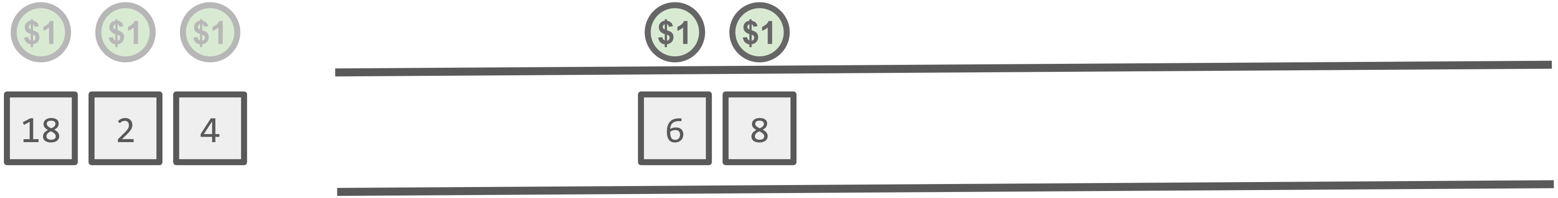


- `add(a)` - Charge $2|A|:
  - Consists of |A| enqueues. Each charged $2
    - $1 for the actual insertion
    - $1 in the bank, for future dequeue

- `enqueue(a)` - Add element `a` into the queue
- `dequeue()` - Dequeue a single element from the queue
- `delete(k)` - Dequeue `k` elements from the queue
- `add(A)` - enqueue all elements in `A`

Question 2

`delete(3)`

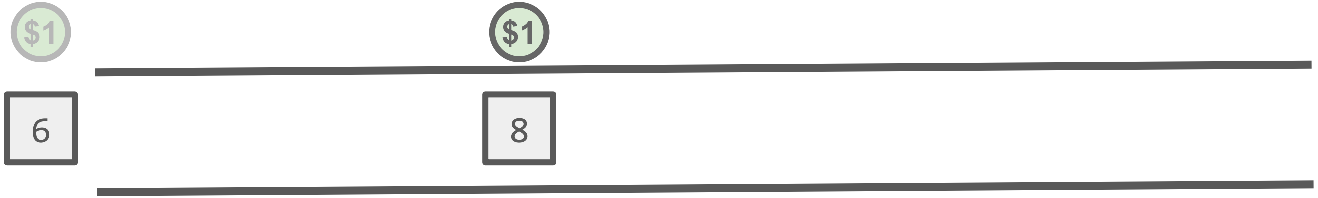


- `delete(k)` - Charge **$0**:
  - The actual cost is **paid for already**
  - k elements deleted with $k from the bank

- `enqueue(a)` - Add element `a` into the queue
- `dequeue()` - Dequeue a single element from the queue
- `delete(k)` - Dequeue `k` elements from the queue
- `add(A)` - enqueue all elements in `A`

Question 2

`dequeue()`



- `dequeue()` - Charge **$0**:
  - The actual cost is **paid for already**
  - Delete using $1 from bank

- `enqueue(a)` - Add element `a` into the queue
- `dequeue()` - Dequeue a single element from the queue
- `delete(k)` - Dequeue `k` elements from the queue
- `add(A)` - enqueue all elements in `A`

One final thing to argue

- **Does the bank ever go negative?**

One final thing to argue

- **Does the bank ever go negative?**

- After each insertion of element, $1 is associated to each element in the queue
- Hence when we dequeue that element, we can use the $1 in the bank associated to that element. i.e. there is always enough money in the bank

Q2 Summary

Operation	*Actual Cost*	*Amortised Cost*	*Effect on bank*
`enqueue(a)`	*$1*	*$2*	*$1*
`add(A)`	*$|A|*	*$2|A|*	*$|A|*
`dequeue()`	*$1*	*$0*	*-$1*
`delete(k)`	*$k*	*$0*	*-$k*

- **Claim:**
  - `enqueue(a), dequeue(), delete(k)` runs in amortised *O(1)* time
  - `add(A)` runs in amortised *O(|A|)* time

Q2 Summary

Tip: It's okay if you decide to "overcharge". e.g. Say amortised cost is \$5. Don't need to be "tight"! Sometimes overcharging makes it easier to argue things

Operation	*Actual Cost*	*Amortised Cost*	*Effect on bank*
`enqueue(a)`	*\$1*	*\$2*	*\$1*
`add(A)`	*\$|A|*	*\$2|A|*	*\$|A|*
`dequeue()`	*\$1*	*\$0*	*-\$1*
`delete(k)`	*\$k*	*\$0*	*-\$k*

- **Claim:**
  - `enqueue(a), dequeue(), delete(k)` runs in amortised $O(1)$ time
  - `add(A)` runs in amortised $O(|A|)$ time

Potential Method

Potential Method

- Instead of **paying in advance** by “storing money” in the bank (accounting method), the potential method pays in advance by “building up potential”

Potential Method

- Instead of **paying in advance** by "storing money" in the bank (accounting method), the potential method pays in advance by "building up potential"
- ϕ is the **Potential function** associated with the algorithm / data structure
    - This is something you have to cleverly define yourself!

Potential Method

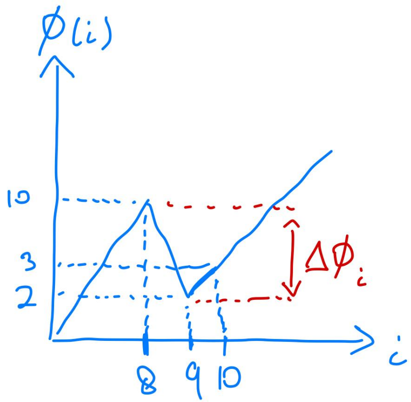
- Instead of **paying in advance** by “storing money” in the bank (accounting method), the potential method pays in advance by “building up potential”
- ϕ is the **Potential function** associated with the algorithm / data structure
  - This is something you have to cleverly define yourself!
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**

Potential Method

- Instead of **paying in advance** by "storing money" in the bank (accounting method), the potential method pays in advance by "building up potential"
- ϕ is the **Potential function** associated with the algorithm / data structure
  - This is something you have to cleverly define yourself!
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**
- $\Delta\phi_i = \phi(i) - \phi(i-1)$ = potential **difference** going from $(i-1)^{th}$ to i^{th} operation

Potential Method

- Instead of **paying in advance** by "storing money" in the bank (accounting method), the potential method pays in advance by "building up potential"
- ϕ is the **Potential function** associated with the algorithm / data structure
    - This is something you have to cleverly define yourself!
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**
- $\Delta\phi_i = \phi(i) - \phi(i-1)$ = potential **difference** going from $(i-1)^{th}$ to i^{th} operation



Not to scale

$\phi(8) = 10$, $\phi(9) = 2$, $\phi(10) = 3$

$\Delta\phi_9 = 2 - 10 = -8$ (Change in potential after 9^{th} op)

$\Delta\phi_{10} = 3 - 2 = 1$ (Change in potential after 10^{th} op)

Potential Method Analogy (Source)

- Water tank:

Potential Method Analogy (Source)

- Water tank:
  - When you open the faucet you have nice **water pressure**. [high potential]
  - When empty, no pressure. [low potential]
  - You need to slowly refill to build up pressure [building up potential energy]

Potential Method Analogy (Source)

- Water tank:
  - When you open the faucet you have nice **water pressure**. [high potential]
  - When empty, no pressure. [low potential]
  - You need to slowly refill to build up pressure [building up potential energy]
- Potential Function: The water in the water tank itself
  - As you fill more water, the potential in the water tank increases
  - If you draw water, there will be some nice water pressure for you to get the water, but the potential decreases over time until there is no more

Potential Method Analogy (Source)

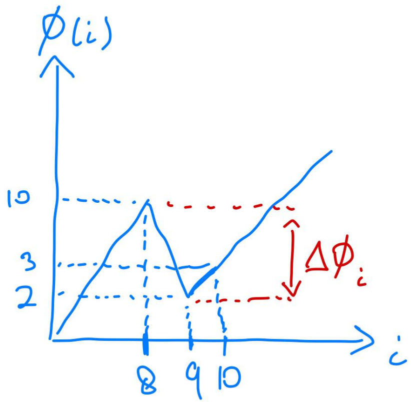
- Water tank:
  - When you open the faucet you have nice **water pressure**. [high potential]
  - When empty, no pressure. [low potential]
  - You need to slowly refill to build up pressure [building up potential energy]
- Potential Function: The water in the water tank itself
  - As you fill more water, the potential in the water tank increases
  - If you draw water, there will be some nice water pressure for you to get the water, but the potential decreases over time until there is no more

- Drawing a lot of water is a “cheap” operation, because of the water pressure (the potential energy) pushing it out for you [Similar to accounting method. Where your ‘stored up energy’ will pay for the ‘expensive operation’]

Potential Method

- Instead of **paying in advance** by "storing money" in the bank (accounting method), the potential method pays in advance by "building up potential"
- ϕ is the **Potential function** associated with the algorithm / data structure
  - This is something you have to cleverly define yourself!
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**
- $\Delta\phi_i = \phi(i) - \phi(i-1)$ = potential **difference** going from $(i-1)^{th}$ to i^{th} operation

Sorry, not to scale :p



$\phi(8) = 10$, $\phi(9) = 2$, $\phi(10) = 3$

$\Delta\phi_9 = 2 - 10 = -8$ (Change in potential after 9th op)

$\Delta\phi_{10} = 3 - 2 = 1$ (Change in potential after 10th op)

Potential Method

- ϕ is the **Potential function** associated with the algorithm / data structure
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**
- $\Delta\phi_i = \phi(i) - \phi(i\text{-}1)$ = potential **difference** going from $(i\text{-}1)^{th}$ to i^{th} operation

Game plan to show why potential method works:

1. Define an amortised cost $c(i)$ that is the actual cost $t(i)$ but adjusted
2. Show that SUM of amortised cost ≥ SUM of actual cost
3. Conclude that SUM of amortised cost is $O(f(n))$.
   So we know that SUM of actual cost is $O(f(n))$

Potential Method

- ϕ is the **Potential function** associated with the algorithm / data structure
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**
- $\Delta\phi_i = \phi(i) - \phi(i-1)$ = potential **difference** going from $(i-1)^{th}$ to i^{th} operation

1. Define an amortised cost $c(i)$ that is the actual cost $t(i)$ but adjusted
2. Show that SUM of amortised cost ≥ SUM of actual cost
3. Conclude that SUM of amortised cost is $O(f(n))$.
   So we know that SUM of actual cost is $O(f(n))$

In Potential Method, definition of $c(i) = t(i) + \phi(i) - \phi(i-1) = t(i) + \Delta\phi_i$

- t(i) is small, then we want $\Delta\phi_i$ to be positive and small. So $c(i)$ is still small
- t(i) is **large**, then we want $\Delta\phi_i$ to be **negative** and large! So $c(i)$ is **also small**

Potential Method

- ϕ is the **Potential function** associated with the algorithm / data structure
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**
- $\Delta\phi_i = \phi(i) - \phi(i\text{-}1)$ = potential **difference** going from $(i\text{-}1)^{th}$ to i^{th} operation

1. Define an amortised cost $c(i)$ that is the actual cost $t(i)$ but adjusted
2. Show that SUM of amortised cost ≥ SUM of actual cost
3. Conclude that SUM of amortised cost is $O(f(n))$.
   So we know that SUM of actual cost is $O(f(n))$

Additionally:

- $\phi(0) = 0$
- $\phi(i) \geq 0$

In Potential Method, definition of $c(i) = t(i) + \phi(i) - \phi(i\text{-}1) = t(i) + \Delta\phi_i$

Here, we will argue the case for $\phi(0) = 0$ only

Goal: Show that SUM of amortised cost ≥ SUM of actual cost

Potential Method

- ϕ is the **Potential function** associated with the algorithm / data structure
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**
- $\Delta\phi_i = \phi(i) - \phi(i-1)$ = potential **difference** going from $(i-1)^{th}$ to i^{th} operation
- Amortised cost $c(i)$ and true cost $t(i)$
- In Potential Method, definition of $c(i) = t(i) + \phi(i) - \phi(i-1) = t(i) + \Delta\phi_i$

Additionally:

- $\phi(0) = 0$
- $\phi(i) \geq 0$

$$\sum_{i=1}^{n} c(i) = \sum_{i=1}^{n} \left(t(i) + \phi(i) - \phi(i-1) \right)$$

Potential Method

By our definition of c(i)

Goal: Show that SUM of amortised cost ≥ SUM of actual cost

Potential Method

- ϕ is the **Potential function** associated with the algorithm / data structure
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**
- $\Delta\phi_i = \phi(i) - \phi(i-1)$ = potential **difference** going from $(i-1)^{th}$ to i^{th} operation
- Amortised cost $c(i)$ and true cost $t(i)$
- In Potential Method, definition of $c(i) = t(i) + \phi(i) - \phi(i-1) = t(i) + \Delta\phi_i$

Additionally:

- $\phi(0) = 0$
- $\phi(i) \geq 0$

$$\sum_{i=1}^{n} c(i) = \sum_{i=1}^{n} \left(t(i) + \phi(i) - \phi(i-1)\right)$$

$$\begin{aligned} = &\ t(n) + \phi(n) - \phi(n-1) \\ &+ t(n-1) + \phi(n-1) - \phi(n-2) \\ &\ \vdots \\ &+ t(2) + \phi(2) - \phi(1) \\ &+ t(1) + \phi(1) - \phi(0) \end{aligned}$$

Potential Method

Expanding the sums!

Goal: Show that SUM of amortised cost ≥ SUM of actual cost

Potential Method

- ϕ is the **Potential function** associated with the algorithm / data structure
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**
- $\Delta\phi_i = \phi(i) - \phi(i-1)$ = potential **difference** going from $(i-1)^{th}$ to i^{th} operation
- Amortised cost $c(i)$ and true cost $t(i)$
- In Potential Method, definition of $c(i) = t(i) + \phi(i) - \phi(i-1) = t(i) + \Delta\phi_i$

Additionally:

- $\phi(0) = 0$
- $\phi(i) \geq 0$

$$\sum_{i=1}^{n} c(i) = \sum_{i=1}^{n} \left(t(i) + \phi(i) - \phi(i-1)\right)$$

$$\begin{aligned} &= t(n) + \phi(n) - \cancel{\phi(n-1)} \\ &\quad + t(n-1) + \cancel{\phi(n-1)} - \cancel{\phi(n-2)} \\ &\quad \vdots \\ &\quad + t(2) + \cancel{\phi(2)} - \cancel{\phi(1)} \\ &\quad + t(1) + \cancel{\phi(1)} - \phi(0) \end{aligned}$$

Potential Method

These terms cancel! (This is called 'Telescoping')

Goal: Show that SUM of amortised cost ≥ SUM of actual cost

Potential Method

- ϕ is the **Potential function** associated with the algorithm / data structure
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**
- $\Delta\phi_i = \phi(i) - \phi(i-1)$ = potential **difference** going from $(i-1)^{th}$ to i^{th} operation
- Amortised cost $c(i)$ and true cost $t(i)$
- In Potential Method, definition of $c(i) = t(i) + \phi(i) - \phi(i-1) = t(i) + \Delta\phi_i$

Additionally:

- $\phi(0) = 0$
- $\phi(i) \geq 0$

$$\sum_{i=1}^{n} c(i) = \sum_{i=1}^{n} \left(t(i) + \phi(i) - \phi(i-1)\right)$$

$$\begin{aligned} &= t(n) + \phi(n) - \cancel{\phi(n-1)} \\ &\quad + t(n-1) + \cancel{\phi(n-1)} - \cancel{\phi(n-2)} \\ &\quad \vdots \\ &\quad + t(2) + \cancel{\phi(2)} - \cancel{\phi(1)} \\ &\quad + t(1) + \cancel{\phi(1)} - \phi(0) \end{aligned}$$

$$= \left(\sum_{i=1}^{n} t(i)\right) + \phi(n) - \phi(0)$$

Potential Method

Sum up over t(i), and the remaining not-cancelled $\phi(n)$ and $\phi(0)$

Goal: Show that SUM of amortised cost ≥ SUM of actual cost

Potential Method

- ϕ is the **Potential function** associated with the algorithm / data structure
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**
- $\Delta\phi_i = \phi(i) - \phi(i-1)$ = potential **difference** going from $(i-1)^{th}$ to i^{th} operation
- Amortised cost $c(i)$ and true cost $t(i)$
- In Potential Method, definition of $c(i) = t(i) + \phi(i) - \phi(i-1) = t(i) + \Delta\phi_i$

Additionally:

- $\phi(0) = 0$
- $\phi(i) \geq 0$

$$\sum_{i=1}^{n} c(i) = \left(\sum_{i=1}^{n} t(i)\right) + \phi(n) - \phi(0)$$

Potential Method

Copying over from prev slide

Goal: Show that SUM of amortised cost ≥ SUM of actual cost

Potential Method

- ϕ is the **Potential function** associated with the algorithm / data structure
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**
- $\Delta\phi_i = \phi(i) - \phi(i-1)$ = potential **difference** going from $(i-1)^{th}$ to i^{th} operation
- Amortised cost $c(i)$ and true cost $t(i)$
- In Potential Method, definition of $c(i) = t(i) + \phi(i) - \phi(i-1) = t(i) + \Delta\phi_i$

Additionally:

- $\phi(0) = 0$
- $\phi(i) \geq 0$

$$\sum_{i=1}^{n} c(i) = \left(\sum_{i=1}^{n} t(i)\right) + \phi(n) - \phi(0)$$

$$= \sum_{i=1}^{n} t(i) + \phi(n) \qquad [\phi(0) = 0]$$

Potential Method

Restriction on $\phi(0) = 0$

Goal: Show that SUM of amortised cost ≥ SUM of actual cost

Potential Method

- ϕ is the **Potential function** associated with the algorithm / data structure
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**
- $\Delta\phi_i = \phi(i) - \phi(i-1)$ = potential **difference** going from $(i-1)^{th}$ to i^{th} operation
- Amortised cost $c(i)$ and true cost $t(i)$
- In Potential Method, definition of $c(i) = t(i) + \phi(i) - \phi(i-1) = t(i) + \Delta\phi_i$

Additionally:

- $\phi(0) = 0$
- $\phi(i) \geq 0$

$$\sum_{i=1}^{n} c(i) = \left(\sum_{i=1}^{n} t(i)\right) + \phi(n) - \phi(0)$$

$$= \sum_{i=1}^{n} t(i) + \phi(n) \qquad [\phi(0) = 0]$$

$$\geq \sum_{i=1}^{n} t(i) \qquad [\phi(i) \geq 0]$$

Potential Method

Restriction on $\phi(i) \geq 0$

Potential Method

- ϕ is the **Potential function** associated with the algorithm / data structure
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**
- $\Delta\phi_i = \phi(i) - \phi(i-1)$ = potential **difference** going from $(i-1)^{th}$ to i^{th} operation
- Amortised cost $c(i)$ and true cost $t(i)$
- In Potential Method, definition of $c(i) = t(i) + \phi(i) - \phi(i-1) = t(i) + \Delta\phi_i$

Additionally:

- $\phi(0) = 0$
- $\phi(i) \geq 0$

$$\sum_{i=1}^{n} c(i) = \left(\sum_{i=1}^{n} t(i)\right) + \phi(n) - \phi(0)$$

$$= \sum_{i=1}^{n} t(i) + \phi(n) \qquad [\phi(0) = 0]$$

$$\geq \sum_{i=1}^{n} t(i) \qquad [\phi(i) \geq 0]$$

Goal: Show that SUM of amortised cost ≥ SUM of actual cost

Shown!

Potential Method

Restriction on $\phi(i) \geq 0$

Potential Method

- ϕ is the **Potential function** associated with the algorithm / data structure
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**
- $\Delta\phi_i = \phi(i) - \phi(i-1)$ = potential **difference** going from $(i-1)^{th}$ to i^{th} operation
- Amortised cost $c(i)$ and true cost $t(i)$
- In Potential Method, definition of $c(i) = t(i) + \phi(i) - \phi(i-1) = t(i) + \Delta\phi_i$

Additionally:

- $\phi(0) = 0$
- $\phi(i) \geq 0$

1. Define an amortised cost $c(i)$ that is the actual cost $t(i)$ but adjusted
2. Show that SUM of amortised cost ≥ SUM of actual cost
3. Conclude that SUM of amortised cost is $O(f(n))$.
   So we know that SUM of actual cost is $O(f(n))$

Since $\sum_{i=1}^{n} c(i) \geq \sum_{i=1}^{n} t(i)$,

If $\sum_{i=1}^{n} c(i) = O(f(n))$

then $\sum_{i=1}^{n} t(i) = O(f(n))$

Potential Method

- ϕ is the **Potential function** associated with the algorithm / data structure
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**
- $\Delta\phi_i = \phi(i) - \phi(i-1)$ = potential **difference** going from $(i-1)^{th}$ to i^{th} operation
- Amortised cost $c(i)$ and true cost $t(i)$
- In Potential Method, definition of $c(i) = t(i) + \phi(i) - \phi(i-1) = t(i) + \Delta\phi_i$

Additionally:

- $\phi(0) = 0$
- $\phi(i) \geq 0$

1. Define an amortised cost $c(i)$ that is the actual cost $t(i)$ but adjusted
2. Show that SUM of amortised cost ≥ SUM of actual cost
3. Conclude that SUM of amortised cost is $O(f(n))$.
   So we know that SUM of actual cost is $O(f(n))$

Since $\sum_{i=1}^{n} c(i) \geq \sum_{i=1}^{n} t(i)$,

If $\sum_{i=1}^{n} c(i) = O(f(n))$

then $\sum_{i=1}^{n} t(i) = O(f(n))$

It means that all we need to bound is just the **total amortised cost**

And we bound the **total actual cost** *for free!*

Potential Method - Recipe

- Try to select a suitable $\boldsymbol{\phi}$, so that for the costly operation, $\Delta\boldsymbol{\phi}_i$ is negative to such an extent that it *nullifies* or *reduces* the effect of actual cost.

Potential Method - Recipe

- Try to select a suitable $\boldsymbol{\phi}$, so that for the costly operation, $\Delta\boldsymbol{\phi}_i$ is negative to such an extent that it *nullifies* or *reduces* the effect of actual cost.
- **Question**: How to find such a suitable potential function $\boldsymbol{\phi}$?

Try to view carefully the costly operation and see if there is some quantity that is "**decreasing**" during the operation.

Question 3:
Fancier Queue (Potential Method)

Question 3

- Consider a data structure that is based on a queue with these operations:
  - `enqueue(a)` - Add element `a` into the queue
  - `dequeue()` - Dequeue a single element from the queue
  - `delete(k)` - Dequeue `k` elements from the queue
  - `add(A)` - enqueue all elements in `A`
- **Claim:**
  - `enqueue(a), dequeue(), delete(k)` runs in amortised *O(1)* time
  - `add(A)` runs in amortised *O(|A|)* time
- **Using potential method**, prove these time complexities
  - To use potential method, state the potential function

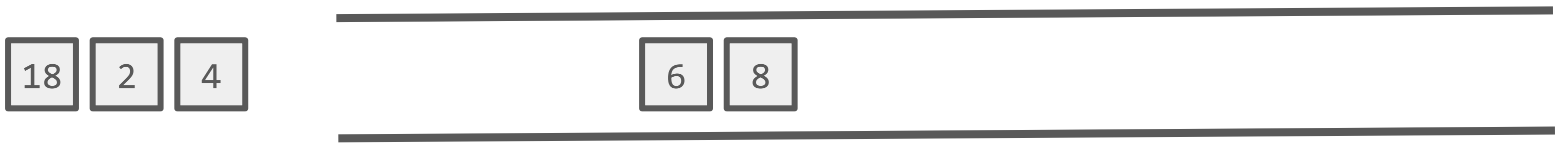
Question 3

- Consider a data structure that is based on a queue with four operations:
  - ENQUEUE(a): Add the element a into the queue
  - DEQUEUE(): Dequeue a single element from the queue
  - DELETE(k): Dequeue k elements from the queue
  - ADD(A): Enqueue all elements in A
- **Claim**: ENQUEUE, DEQUEUE and DELETE run in amortized $O(1)$ time while ADD runs in amortized $O(|A|)$ time.
- Using Potential method, can you show that these time complexities are correct?
- (Please state your potential function.)

Question 3 Idea

- **Claim:**
  - `enqueue(a), dequeue(), delete(k)` runs in amortised *O(1)* time
  - `add(A)` runs in amortised *O(|A|)* time

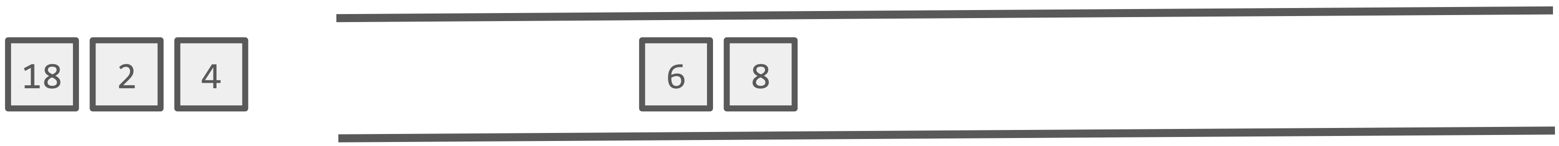
- The delete is our **costly** operation that we want to "nullify" - what's decreasing during deletion?



Question 3 Idea

- **Claim:**
  - `enqueue(a), dequeue(), delete(k)` runs in amortised *O(1)* time
  - `add(A)` runs in amortised *O(|A|)* time

- The delete is our **costly** operation that we want to “nullify” - what’s decreasing during deletion?
- **The number of elements in the queue!**



Question 3 Solution

- ϕ is the **Potential function** associated with the algorithm / data structure
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**
- $\Delta\phi_i = \phi(i) - \phi(i-1)$ = potential **difference** going from $(i-1)^{th}$ to i^{th} operation
- Amortised cost $c(i)$ and true cost $t(i)$
- In Potential Method, definition of $c(i) = t(i) + \phi(i) - \phi(i-1) = t(i) + \Delta\phi_i$

Additionally:

- $\phi(0) = 0$
- $\phi(i) \geq 0$

$\phi(i)$ = number of elements in the queue after i^{th} operation

Question 3 Solution

- ϕ is the **Potential function** associated with the algorithm / data structure
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**
- $\Delta\phi_i = \phi(i) - \phi(i-1)$ = potential **difference** going from $(i-1)^{th}$ to i^{th} operation
- Amortised cost $c(i)$ and true cost $t(i)$
- In Potential Method, definition of $c(i) = t(i) + \phi(i) - \phi(i-1) = t(i) + \Delta\phi_i$

Additionally:

- $\phi(0) = 0$
- $\phi(i) \geq 0$

$\phi(i)$ = number of elements in the queue after i^{th} operation

For `enqueue(a)`:

- Actual cost $t(i) = 1$

Question 3 Solution

- ϕ is the **Potential function** associated with the algorithm / data structure
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**
- $\Delta\phi_i = \phi(i) - \phi(i-1)$ = potential **difference** going from $(i-1)^{th}$ to i^{th} operation
- Amortised cost $c(i)$ and true cost $t(i)$
- In Potential Method, definition of $c(i) = t(i) + \phi(i) - \phi(i-1) = t(i) + \Delta\phi_i$

Additionally:

- $\phi(0) = 0$
- $\phi(i) \geq 0$

$\phi(i)$ = number of elements in the queue after i^{th} operation

For `enqueue(a)`:

- Actual cost $t(i) = 1$
- **One new element**: $\phi(i) - \phi(i - 1) = \Delta\phi_i = 1$

Question 3 Solution

- ϕ is the **Potential function** associated with the algorithm / data structure
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**
- $\Delta\phi_i = \phi(i) - \phi(i-1)$ = potential **difference** going from $(i-1)^{th}$ to i^{th} operation
- Amortised cost $c(i)$ and true cost $t(i)$
- In Potential Method, definition of $c(i) = t(i) + \phi(i) - \phi(i-1) = t(i) + \Delta\phi_i$

Additionally:

- $\phi(0) = 0$
- $\phi(i) \geq 0$

$\phi(i)$ = number of elements in the queue after i^{th} operation

For `enqueue(a)`:

- Actual cost $t(i) = 1$
- **One new element**: $\phi(i) - \phi(i - 1) = \Delta\phi_i = 1$
- Amortised Cost $c(i) = t(i) + \Delta\phi_i = 1 + 1 = 2$

Question 3 Solution

- ϕ is the **Potential function** associated with the algorithm / data structure
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**
- $\Delta\phi_i = \phi(i) - \phi(i-1)$ = potential **difference** going from $(i-1)^{th}$ to i^{th} operation
- Amortised cost $c(i)$ and true cost $t(i)$
- In Potential Method, definition of $c(i) = t(i) + \phi(i) - \phi(i-1) = t(i) + \Delta\phi_i$

Additionally:

- $\phi(0) = 0$
- $\phi(i) \geq 0$

$\phi(i)$ = number of elements in the queue after i^{th} operation

For `dequeue()`:

- Actual cost $t(i) = 1$
- **One LESS element**: $\phi(i) - \phi(i - 1) = \Delta\phi_i = -1$
- Amortised Cost $c(i) = t(i) + \Delta\phi_i = 1 + (-1) = 0$

Question 3 Solution

- ϕ is the **Potential function** associated with the algorithm / data structure
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**
- $\Delta\phi_i = \phi(i) - \phi(i\text{-}1)$ = potential **difference** going from $(i\text{-}1)^{th}$ to i^{th} operation
- Amortised cost $c(i)$ and true cost $t(i)$
- In Potential Method, definition of $c(i) = t(i) + \phi(i) - \phi(i\text{-}1) = t(i) + \Delta\phi_i$

Additionally:

- $\phi(0) = 0$
- $\phi(i) \geq 0$

$\phi(i)$ = number of elements in the queue after i^{th} operation

For `delete(k)`:

- Actual cost $t(i) = k$
- **k LESS element**: $\phi(i) - \phi(i - 1) = \Delta\phi_i = -k$
- Amortised Cost $c(i) = t(i) + \Delta\phi_i = k + (-k) = 0$

Question 3 Solution

- ϕ is the **Potential function** associated with the algorithm / data structure
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**
- $\Delta\phi_i = \phi(i) - \phi(i\text{-}1)$ = potential **difference** going from $(i\text{-}1)^{th}$ to i^{th} operation
- Amortised cost $c(i)$ and true cost $t(i)$
- In Potential Method, definition of $c(i) = t(i) + \phi(i) - \phi(i\text{-}1) = t(i) + \Delta\phi_i$

Additionally:

- $\phi(0) = 0$
- $\phi(i) \geq 0$

$\phi(i)$ = number of elements in the queue after i^{th} operation

For `add(A)`:

- Actual cost $t(i) = |A|$
- **|A| new element**: $\phi(i) - \phi(i - 1) = \Delta\phi_i = |A|$
- Amortised Cost $c(i) = t(i) + \Delta\phi_i = |A| + |A| = 2|A|$

Potential Method -- to fulfill

Additionally:

- $\phi(0) = 0$
- $\phi(i) \geq 0$

$\phi(i)$ = *number of elements in the queue after* i^{th} *operation*

Potential Method -- to fulfill

Additionally:

- $\phi(0) = 0$
- $\phi(i) \geq 0$

$\phi(i)$ *= number of elements in the queue after* i^{th} *operation*

In the beginning, we have empty queue → $\phi(0) = 0$

Never do we have negative elements → $\phi(i) \geq 0$

Question 4: Dynamic Table (Delete-only)

Question 4

```
def DynamicTableDeleteOnly(T):
n = number of elements in T
delete last element x from T
n -= 1
if n == 0:
  free(T)
else:
  if n == size(T)/2:
    T' = create_table(n/2)
    copy all elements from T to T'
    free(T)
    T = T'
```

The idea:

- Start with a full table
- Once number of elements is half the elements it can hold, halve it

Note: we do not call any other operations on `T`

Show that amortised cost of `DynamicTableDeleteOnly` is O(1)

Question 4

Delete x from T;

$n \leftarrow n - 1$;

If ($n = 0$)

  free(T);

Else

  If(n = size(T)/2)

  { $T' \leftarrow$ createTable($n/2$);

   copy(T,T');

   free(T);

   $T \leftarrow T'$

  }

Note, T is the dynamic table that supports only deletions.

Using Potential method show that the amortized cost of each Deletion operation is O(1).

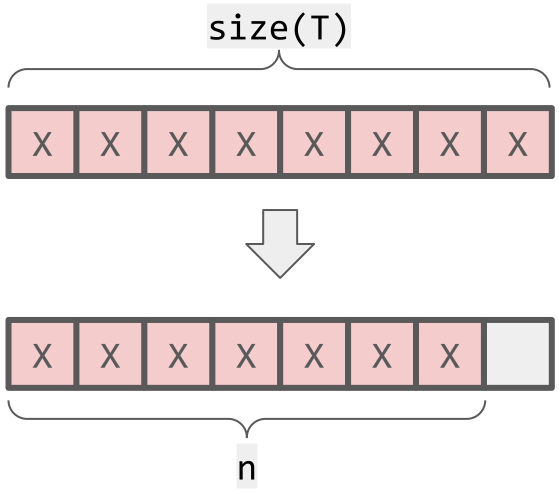
(State your potential function.)

Question 4 Idea

There are two possibilities when deleting:

Case 1: Table doesn't shrink (just delete)

This is cheap!

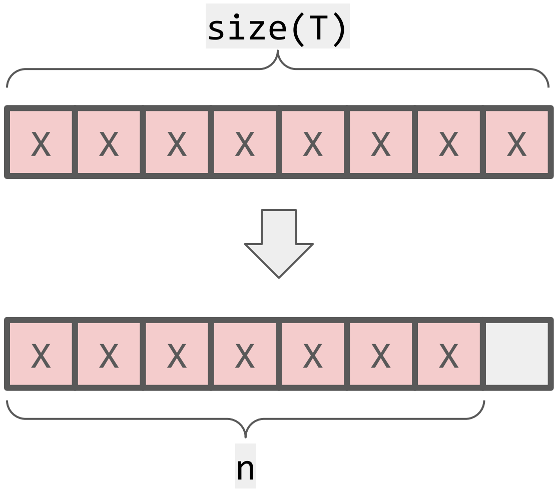


Question 4 Idea

There are two possibilities when deleting:

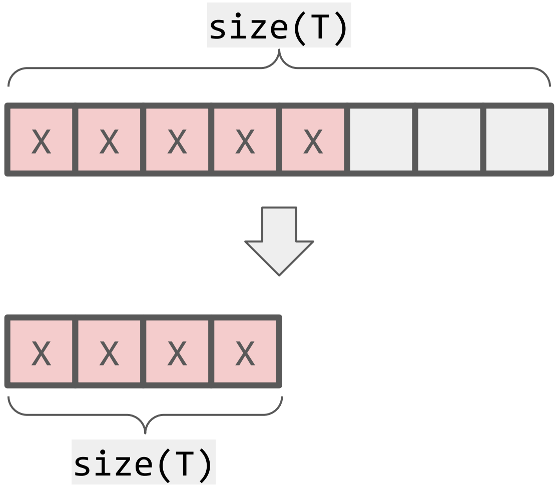
Case 1: Table doesn't shrink (just delete)

This is cheap!



Case 2: Table **shrinks**

This operation is expensive!



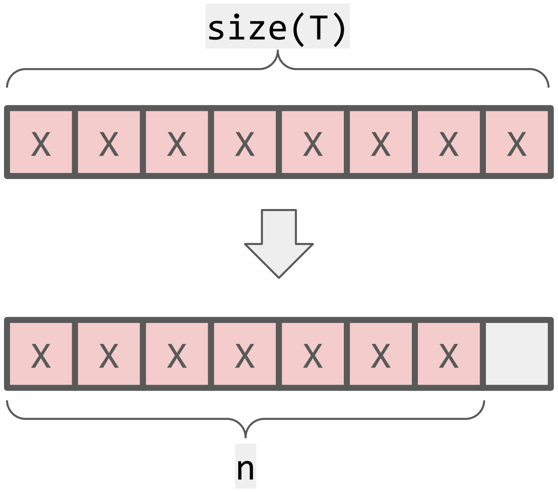
Look for a “decreasing” quantity during expensive operation.
Potential function?

Question 4 Idea

There are two possibilities when deleting:

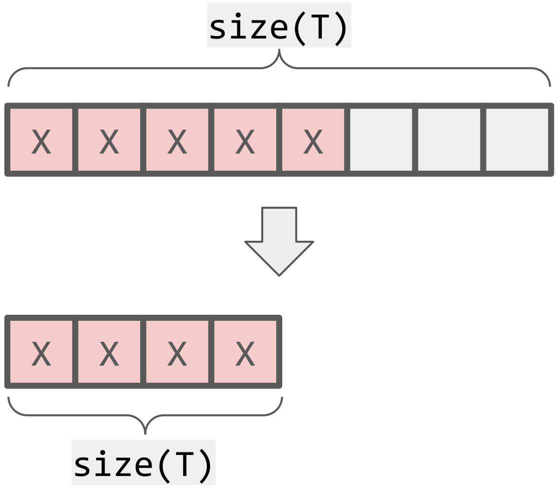
Case 1: Table doesn’t shrink (just delete)

This is cheap!



Case 2: Table **shrinks**

This operation is expensive!



Question 4 Idea

Look for a "decreasing" quantity during expensive operation.
Potential function? `size(T) - n`
The number of empty slots!

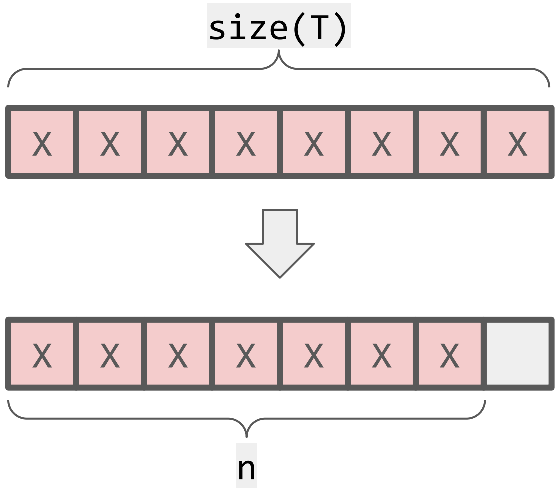
There are two possibilities when deleting:

Additionally:

- $\phi(0) = 0$
- $\phi(i) \geq 0$

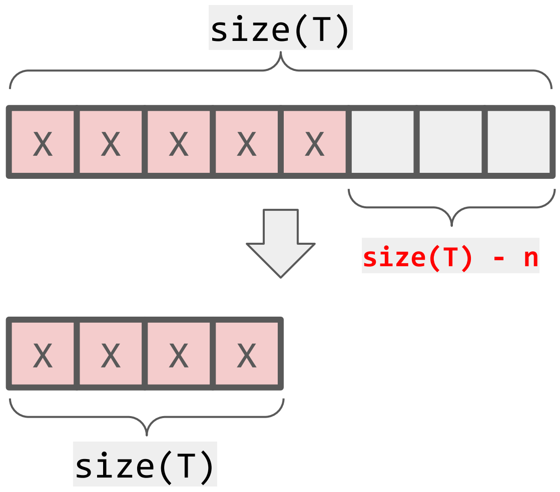
Case 1: Table doesn't shrink (just delete)

This is cheap!



Case 2: Table **shrinks**

This operation is expensive!



Question 4 Idea

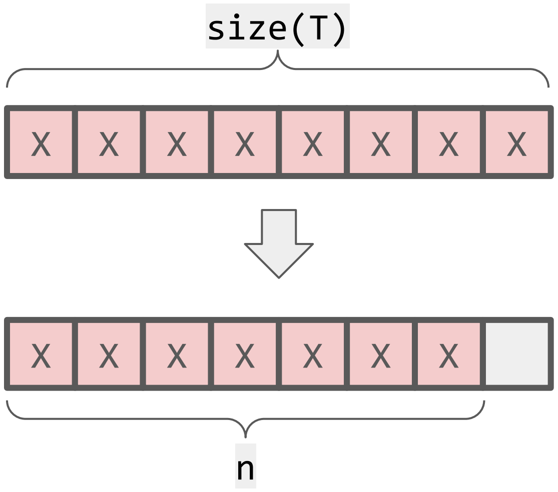
Look for a “decreasing” quantity during expensive operation.
Potential function? `size(T) - n`
The number of empty slots!
Verify that it satisfies these properties

Additionally:
- $\phi(0) = 0$
- $\phi(i) \geq 0$

There are two possibilities when deleting:

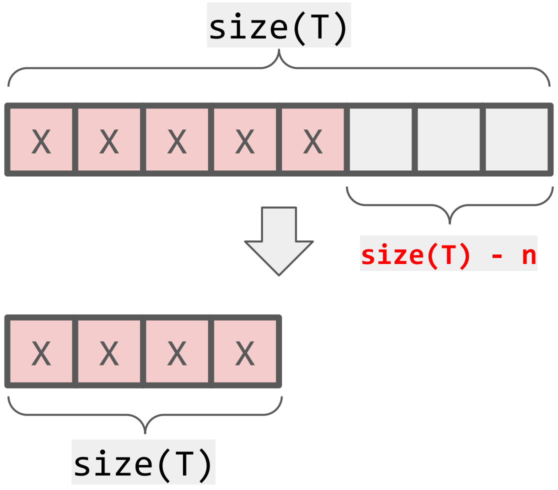
Case 1: Table doesn’t shrink (just delete)

This is cheap!



Case 2: Table **shrinks**

This operation is expensive!



Question 4 Soln

- ϕ is the **Potential function** associated with the algorithm / data structure
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**
- $\Delta\phi_i = \phi(i) - \phi(i-1)$ = potential **difference** going from $(i-1)^{th}$ to i^{th} operation
- Amortised cost $c(i)$ and true cost $t(i)$
- In Potential Method, definition of $c(i) = t(i) + \phi(i) - \phi(i-1) = t(i) + \Delta\phi_i$

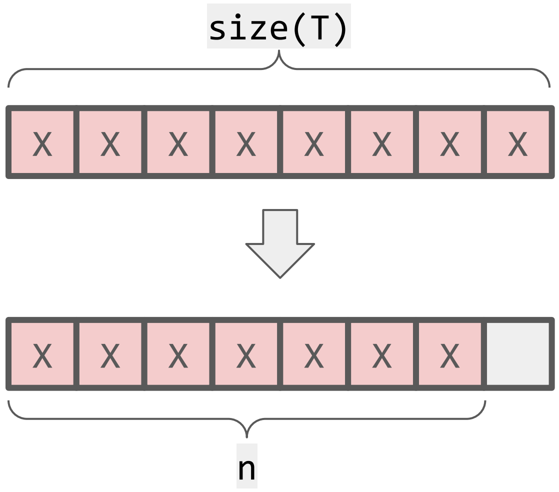
Additionally:

- $\phi(0) = 0$
- $\phi(i) \geq 0$

$\phi(i)$: `size(T) - n`

Case 1: Table doesn't shrink (just delete)

This is cheap!



Actual Cost	$\Delta\phi_i$	*Amortised Cost*

Case 1

Remember:

amortised = actual + potential difference

Question 4 Soln

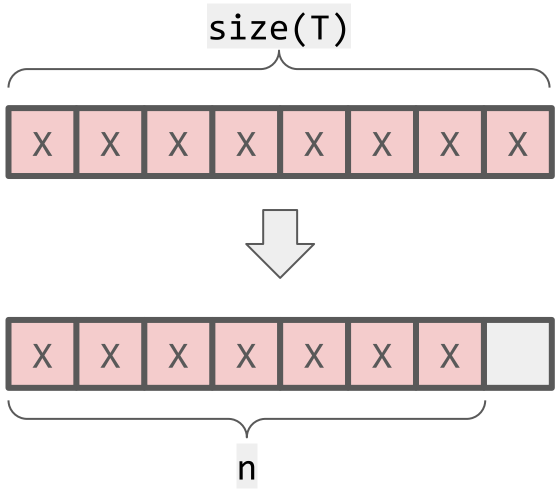
- ϕ is the **Potential function** associated with the algorithm / data structure
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**
- $\Delta\phi_i = \phi(i) - \phi(i-1)$ = potential **difference** going from $(i-1)^{th}$ to i^{th} operation
- Amortised cost $c(i)$ and true cost $t(i)$
- In Potential Method, definition of $c(i) = t(i) + \phi(i) - \phi(i-1) = t(i) + \Delta\phi_i$

Additionally:
- $\phi(0) = 0$
- $\phi(i) \geq 0$

$\phi(i)$: `size(T) - n`

Case 1: Table doesn't shrink (just delete)

This is cheap!



Actual Cost	$\Delta\phi_i$	*Amortised Cost*
1		

Case 1

```
Actual Cost = 1 because we just delete
that element
```

Question 4 Soln

- ϕ is the **Potential function** associated with the algorithm / data structure
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**
- $\Delta\phi_i = \phi(i) - \phi(i-1)$ = potential **difference** going from $(i-1)^{th}$ to i^{th} operation
- Amortised cost $c(i)$ and true cost $t(i)$
- In Potential Method, definition of $c(i) = t(i) + \phi(i) - \phi(i-1) = t(i) + \Delta\phi_i$

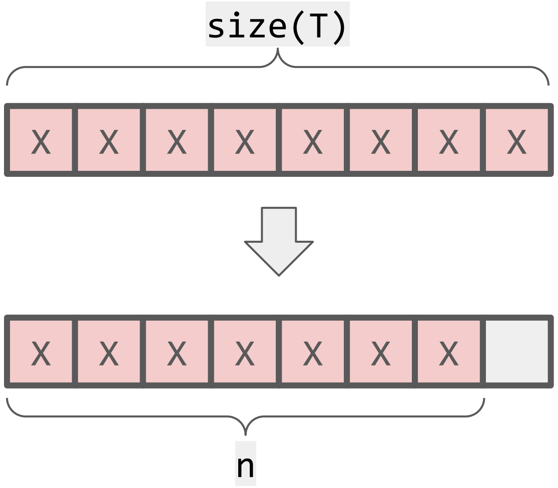
Additionally:

- $\phi(0) = 0$
- $\phi(i) \geq 0$

$\phi(i)$: `size(T) - n`

Case 1: Table doesn't shrink (just delete)

This is cheap!



Actual Cost	$\Delta\phi_i$	*Amortised Cost*
1	1	

Case 1

```
Potential difference is 1 - 0 = 1
Recall: potential is empty slots in table
```

Question 4 Soln

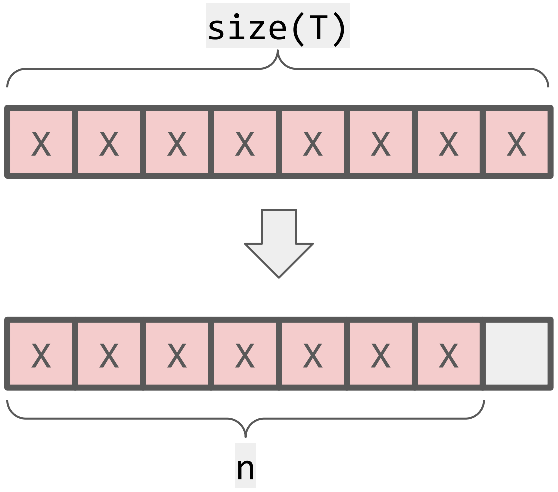
- ϕ is the **Potential function** associated with the algorithm / data structure
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**
- $\Delta\phi_i = \phi(i) - \phi(i-1)$ = potential **difference** going from $(i-1)^{th}$ to i^{th} operation
- Amortised cost $c(i)$ and true cost $t(i)$
- In Potential Method, definition of $c(i) = t(i) + \phi(i) - \phi(i-1) = t(i) + \Delta\phi_i$

Additionally:
- $\phi(0) = 0$
- $\phi(i) \geq 0$

$\phi(i)$: `size(T) - n`

Case 1: Table doesn't shrink (just delete)

This is cheap!



Actual Cost	$\Delta\phi_i$	*Amortised Cost*
1	1	2

Case 1

`Amortised Cost is 1 + 1 = 2`

Question 4 Soln

- ϕ is the **Potential function** associated with the algorithm / data structure
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**
- $\Delta\phi_i = \phi(i) - \phi(i-1)$ = potential **difference** going from $(i-1)^{th}$ to i^{th} operation
- Amortised cost $c(i)$ and true cost $t(i)$
- In Potential Method, definition of $c(i) = t(i) + \phi(i) - \phi(i-1) = t(i) + \Delta\phi_i$

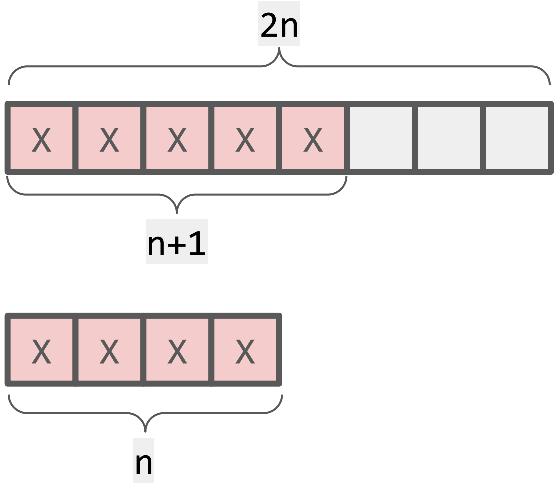
Additionally:
- $\phi(0) = 0$
- $\phi(i) \geq 0$

$\phi(i)$: `size(T) - n`

Actual Cost	$\Delta\phi_i$	*Amortised Cost*

Case 2: Table **shrinks**

This operation is expensive!



Case 2

For simplicity, I will base number of elements to the final number of elements

Question 4 Soln

- ϕ is the **Potential function** associated with the algorithm / data structure
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**
- $\Delta\phi_i = \phi(i) - \phi(i\text{-}1)$ = potential **difference** going from $(i\text{-}1)^{th}$ to i^{th} operation
- Amortised cost $c(i)$ and true cost $t(i)$
- In Potential Method, definition of $c(i) = t(i) + \phi(i) - \phi(i\text{-}1) = t(i) + \Delta\phi_i$

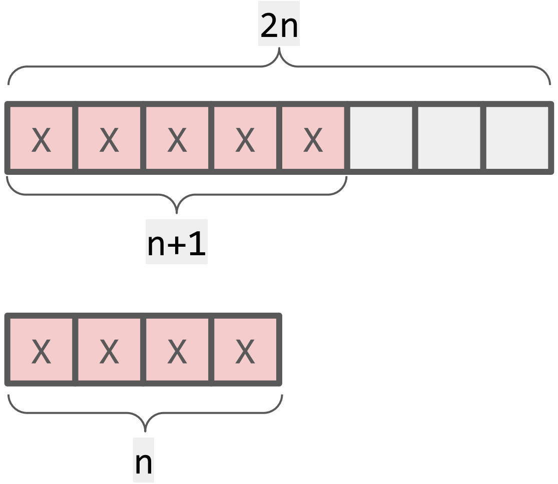
Additionally:
- $\phi(0) = 0$
- $\phi(i) \geq 0$

$\phi(i)$: `size(T) - n`

Actual Cost	$\Delta\phi_i$	*Amortised Cost*
$n+1$		

Case 2: Table **shrinks**

This operation is expensive!



Case 2

Actual cost is n+1, because we delete one of the elements, and copy the rest

Question 4 Soln

- ϕ is the **Potential function** associated with the algorithm / data structure
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**
- $\Delta\phi_i = \phi(i) - \phi(i-1)$ = potential **difference** going from $(i-1)^{th}$ to i^{th} operation
- Amortised cost $c(i)$ and true cost $t(i)$
- In Potential Method, definition of $c(i) = t(i) + \phi(i) - \phi(i-1) = t(i) + \Delta\phi_i$

Additionally:

- $\phi(0) = 0$
- $\phi(i) \geq 0$

$\phi(i)$: `size(T) - n`

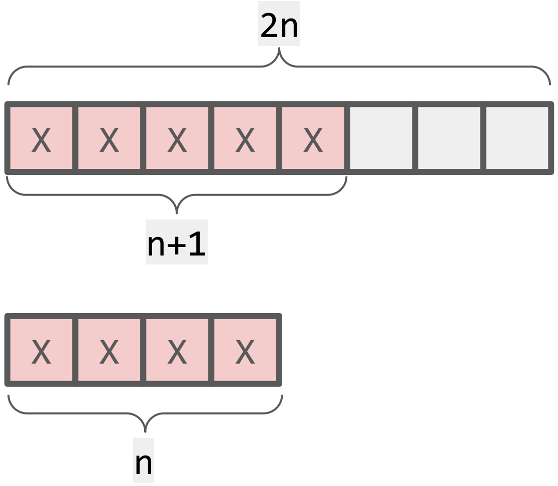
Actual Cost	$\Delta\phi_i$	*Amortised Cost*
$n+1$	$1-n$	

Case 2

```
Potential After: 0
Potential Before: (2n) - (n+1) = n-1
Potential Difference = (0) - (n-1) = 1-n
```

Case 2: Table **shrinks**

This operation is expensive!



Question 4 Soln

- ϕ is the **Potential function** associated with the algorithm / data structure
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**
- $\Delta\phi_i = \phi(i) - \phi(i-1)$ = potential **difference** going from $(i-1)^{th}$ to i^{th} operation
- Amortised cost $c(i)$ and true cost $t(i)$
- In Potential Method, definition of $c(i) = t(i) + \phi(i) - \phi(i-1) = t(i) + \Delta\phi_i$

Additionally:

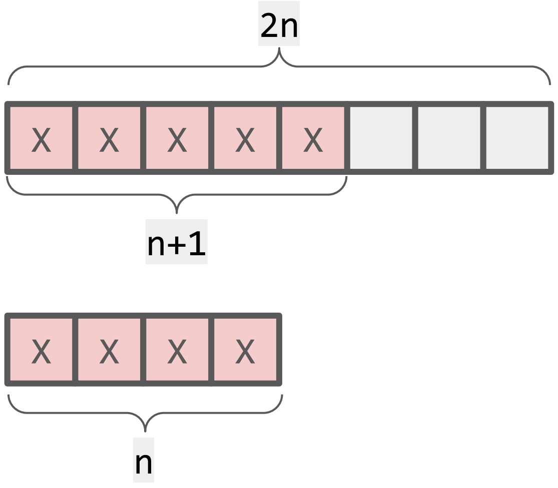
- $\phi(0) = 0$
- $\phi(i) \geq 0$

$\phi(i)$: `size(T) - n`

Actual Cost	$\Delta\phi_i$	*Amortised Cost*
$n+1$	$1-n$	2

Case 2: Table **shrinks**

This operation is expensive!



Case 2

`Amortised Cost: (n+1) + (1-n) = 2`

Q4 Summary

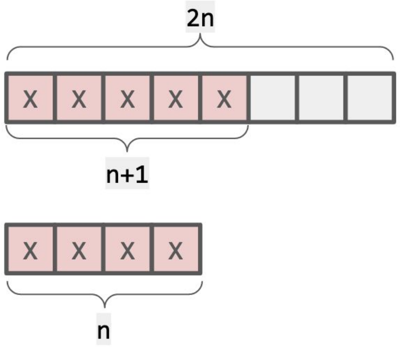
- ϕ is the **Potential function** associated with the algorithm / data structure
- $\phi(i)$ is the **Potential at the end of the i^{th} operation**
- $\Delta\phi_i = \phi(i) - \phi(i-1)$ = potential **difference** going from $(i-1)^{th}$ to i^{th} operation
- Amortised cost $c(i)$ and true cost $t(i)$
- In Potential Method, definition of $c(i) = t(i) + \phi(i) - \phi(i-1) = t(i) + \Delta\phi_i$

Additionally:

- $\phi(0) = 0$
- $\phi(i) \geq 0$

$\phi(i)$: `size(T) - n`

Cases	Actual Cost	$\Delta\phi_i$	Amortised Cost
Case 1: No shrink	*1*	*1*	*2*
Case 2: Shrinking	*n+1*	*1-n*	*2*



Thus, the amortised cost of this deletion operation is O(1)