Amortized Analysis

Christian Wulff-Nilsen

Sixth lecture

Algorithms and Data Structures

DIKU

February 22, 2023

Overview for today

- Introduction
- Aggregate analysis
- The accounting method
- The potential method
- Applications:
  - Stacks with the `MULTIPOP` operation
  - Binary counters
  - Dynamic tables

Introduction

- So far in the course, we have looked at the worst-case running time/space of an algorithm
- Now, we consider instead the cost of maintaining a data structure under a sequence of operations
- For a stack, this would be a sequence of PUSH and POP operations
- Instead of bounding the worst-case time of a single operation, we wish to bound the *total* worst-case time of *all* operations
- This also allows us to bound the average time of an operation: simply divide the bound on the total cost by the number of operations
- We will study three methods to obtain such bounds on various types of data structures:
  - Aggregate analysis
  - The accounting method
  - The potential method

A stack with the MULTIPOP operation

- Consider a stack data structure with four operations:
  - PUSH(S, x): pushes element x onto stack S
  - POP(S): pops the top element from stack S (we assume S is non-empty just prior to the pop)
  - MULTIPOP(S, k): pops the top $\min\{s, k\}$ elements from stack S where s is the number of elements on the stack just prior to the operation (we assume $\min\{s, k\} > 0$)
  - STACK-EMPTY(S): outputs true if S is empty and false otherwise
- Consider an implementation where each call to PUSH, POP, and STACK-EMPTY takes $\Theta(1)$ worst-case time
- We implement MULTIPOP with $\min\{s, k\}$ successive POP operations, with calls to STACK-EMPTY to check if the stack becomes empty
- MULTIPOP runs in $\Theta(\min\{s, k\})$ worst-case time
- Technical detail: we require $\min\{s, k\} > 0$ as otherwise, $\Theta(\min\{s, k\}) = \Theta(0)$ is not a valid time bound; MULTIPOP spends $\Theta(1)$ worst-case time even when popping no elements

Worst-case running time of our stack data structure

- Consider n operations on an initially empty stack
- What is the worst-case *total* time of these operations?
- Worst-case time for a single operation is $O(n)$ (a single `MULTIPOP` operation can take up to $\Theta(n)$ worst-case time)
- Since there are n operations, we get a total worst-case time over all n operations of $O(n^2)$
- The average time per operation is thus $O(n)$
- Our analysis is correct but the $O(n)$ average bound is very weak
- Using amortized analysis, we can improve it to $O(1)$

Aggregate analysis applied to the stack example

- In aggregate analysis, we calculate an upper bound $T(n)$ on the total worst-case time of n operations and then calculate an upper bound on the average cost, or amortized cost, as $T(n)/n$
- Aggregate analysis is typically more refined than on the previous slide where we simply used the upper bound $O(n)$ for every operation
- Aggregate analysis for the stack example:
  - It suffices to bound the total worst-case time spent on `PUSH` and `POP` operations (Why?)
  - There are at most n `PUSH` operations in total
  - The number of `POP` operations (including those applied as part of `MULTIPOP`) cannot be larger than the number of `PUSH` operations
  - Hence, total worst-case time for all n operations is $O(n)$
  - The amortized cost per operation is $O(n)/n = O(1)$

Binary counter

- We now consider implementing a binary k-bit counter which starts at 0 and counts upwards with the operation $\texttt{INCREMENT}$
- The counter resets to 0 after 2^k increments (overflow)
- The counter is stored in an array $A[0 \ldots k-1]$ where $A[0]$ is the least and $A[k-1]$ is the most significant bit

`INCREMENT` example

- Example of repeated application of `INCREMENT` with $k = 4$:

$A[3]$	$A[2]$	$A[1]$	$A[0]$
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Implementation of `INCREMENT`

- `INCREMENT` is implemented as follows:

`INCREMENT`(A)

$i = 0$
while $i < A.length$ and $A[i] == 1$
    $A[i] = 0$
    $i = i + 1$
if $i < A.length$
    $A[i] = 1$

$A[3]$	$A[2]$	$A[1]$	$A[0]$
0	1	1	1
1	0	0	0

A simple running time analysis

- Consider n $\texttt{INCREMENT}(A)$ operations
- The worst-case time for a call to $\texttt{INCREMENT}(A)$ is proportional to the number of bits that this operation flips which is $O(k)$
- Average time bound: $O(k)$
- We now strengthen this bound with aggregate analysis

Aggregate analysis applied to the binary counter example

- $A[0]$ flips for every call to $\texttt{INCREMENT}(A)$
- However, $A[1]$ only flips for every second call
- In general, $A[i]$ only flips for every 2^ith call, for $i = 0, \ldots, k-1$
- It follows that $A[i]$ only flips $\lfloor n/2^i \rfloor$ times in total
- The total number of flips over all n operations is thus

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor \leq \sum_{i=0}^{k-1} \frac{n}{2^i} = n \sum_{i=0}^{k-1} \frac{1}{2^i} < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

- The total worst-case time for all n operations is thus $O(n)$
- The amortized cost per operation is $O(1)$

The accounting method

- Consider a sequence of n operations to some data structure (for instance the stack or the binary counter)
- Let the time cost of the ith operation be c_i, $i = 1, \ldots, n$
- In the accounting method, we assign artificial costs $\hat{c}_1, \ldots, \hat{c}_n$ to the n operations
- $\hat{c}_i$ is called the *amortized cost* of the ith operation
- Unlike aggregate analysis, we now allow different amortized costs to each of the n operations

The accounting method: requirement on amortized costs

- If $\hat{c}_i > c_i$, we overcharge the operation by the amount $\hat{c}_i - c_i$ which is stored as credit in specific objects of the data structure for later
- If $\hat{c}_i < c_i$, we undercharge the operation by the amount $c_i - \hat{c}_i$ and we pay for this difference with credit stored from previous operations
- Important requirement: for *any* sequence of n operations,

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

- Goal: obtain upper bound on $\sum_{i=1}^{n} \hat{c}_i$; by the above inequality, this will give an upper bound on $\sum_{i=1}^{n} c_i$
- The inequality says that, after having paid the actual cost $\sum_{i=1}^{n} c_i$ for the n operations, the remaining $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i$ credit stored in the data structure must be non-negative

The accounting method applied to the stack example

- Actual costs of stack operations (after scaling by a constant):
  - `PUSH`: 1
  - `POP`: 1
  - `MULTIPOP`: $\min\{s, k\}$
- We choose the following amortized costs:
  - `PUSH`: 2
  - `POP`: 0
  - `MULTIPOP`: 0
- The `PUSH` operation is overcharged by 1
- 1 of the 2 credits pays the actual cost of the `PUSH` operation and the remaining credit is left on the element that was pushed (imagine a coin left on the stack element)
- The `POP` operation is undercharged by 1 and its actual cost is paid for by the credit associated with the popped element
- The amount of credit associated with the stack is never negative

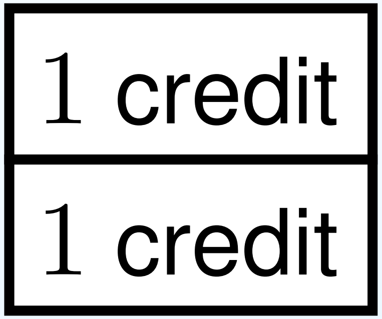
The accounting method applied to the stack example

- Example:

$\boxed{1 \text{ credit}}$

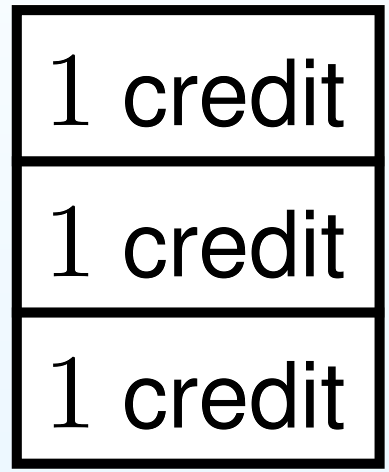
The accounting method applied to the stack example

- Example:



The accounting method applied to the stack example

- Example:



The accounting method applied to the stack example

- Example:

1 credit
1 credit

The accounting method applied to the stack example

- For any sequence of n operations, this gives

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i \leq 2n$$

- The average cost per operation is thus $O(1)$, which is what we obtained earlier using the aggregate method
- Similarly, we can use the accounting method for the binary counter example
- Note that the data structure does not keep track of credits; we only use them in the analysis

The potential method

- Similar to the accounting method except that credit is not stored with specific objects of the data structure (such as elements of the stack)
- Instead, credit is stored in a single place – "the bank"
- The current amount of credits in the bank is expressed by a *potential function* Φ
- Consider n operations to a data structure where D_0 is the data structure before the first operation and D_i is the data structure just after the ith operation, for $i = 1, \ldots, n$
- For $i = 0, \ldots, n$, we denote by $\Phi(D_i)$ the credit stored with the current data structure D_i

Amortized costs with the potential method

- For $i = 1, \ldots, n$, if c_i is the actual cost of the ith operation, we define the amortized cost $\hat{c}_i$ as

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

- If $\Phi(D_i) - \Phi(D_{i-1}) > 0$, we overcharge the ith operation and put $\Phi(D_i) - \Phi(D_{i-1})$ credit into the bank
- If $\Phi(D_i) - \Phi(D_{i-1}) < 0$, we undercharge the ith operation and withdraw $\Phi(D_{i-1}) - \Phi(D_i)$ credit from the bank to help pay for the ith operation

Summing up amortized costs

- Recall from the accounting method the requirement that for any sequence of n operations,

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

- We require the same for the potential method
- By a telescoping sums argument,

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \left(\sum_{i=1}^{n} c_i \right) + \Phi(D_n) - \Phi(D_0)$$

Amortized cost as upper bound on actual cost

- By requiring that $\Phi(D_n) \geq \Phi(D_0)$, we get the desired inequality:

$$\sum_{i=1}^{n} \hat{c}_i = \left(\sum_{i=1}^{n} c_i\right) + \Phi(D_n) - \Phi(D_0) \geq \sum_{i=1}^{n} c_i$$

- Since we might not know n, we require $\Phi(D_i) \geq \Phi(D_0)$ for all $i \geq 0$
- In this case, we say that Φ is *valid*
- Typically, we pick Φ such that $\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all $i \geq 0$
- This clearly ensures that Φ is valid
- In words, the amount of credit in the bank can never be negative

The potential method applied to the stack example

- Let D_0 be the initial stack and let D_i be the stack just after the ith operation
- We choose $\Phi(D_i)$ to be the number of elements on the stack D_i, for $i = 0, \ldots, n$
- Φ satisfies the requirements on the previous slide since $\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i
- We thus have $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$
- We now upper bound $\sum_{i=1}^{n} \hat{c}_i$ to get an upper bound on $\sum_{i=1}^{n} c_i$

Upper bounding $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

- Let $i \in \{1, \ldots, n\}$ be given and consider the ith operation
- Recall that $\Phi(D_i)$ is the number of elements on stack D_i
- If the ith operation is `PUSH`:
  - $\Phi(D_i) - \Phi(D_{i-1}) = 1$
  - Hence, $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$
- If it is `POP`:
  - $\Phi(D_i) - \Phi(D_{i-1}) = -1$
  - Hence, $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 - 1 = 0$
- If it is `MULTIPOP`:
  - Let $k' > 0$ be the number of elements popped
  - $\Phi(D_i) - \Phi(D_{i-1}) = -k'$
  - Hence, $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k' - k' = 0$
- In all cases: $\hat{c}_i \leq 2$

Upper bounding $\sum_{i=1}^{n} \hat{c}_i$ and hence $\sum_{i=1}^{n} c_i$

- We have shown that for each $i \in \{1, \ldots, n\}$, $\hat{c}_i \leq 2$
- Hence, $\sum_{i=1}^{n} \hat{c}_i \leq 2n$
- This also upper bounds the total actual cost of the n operations:

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i \leq 2n$$

- Hence, the average time spent per operation is $O(n)/n = O(1)$

Dynamic tables

- Consider some abstract data structure T which we refer to as a *table*
- It supports the insertion of an element with the operation `INSERT` and the deletion of an element with the operation `DELETE`
- We do not focus on the details of how T supports these operations; we only require that:
  - T is initially empty,
  - the memory used by T is allocated as an array of slots,
  - each operation is supported in $O(1)$ time, and
  - allocating/freeing space for an array of size k takes $O(k)$ time

k - ele

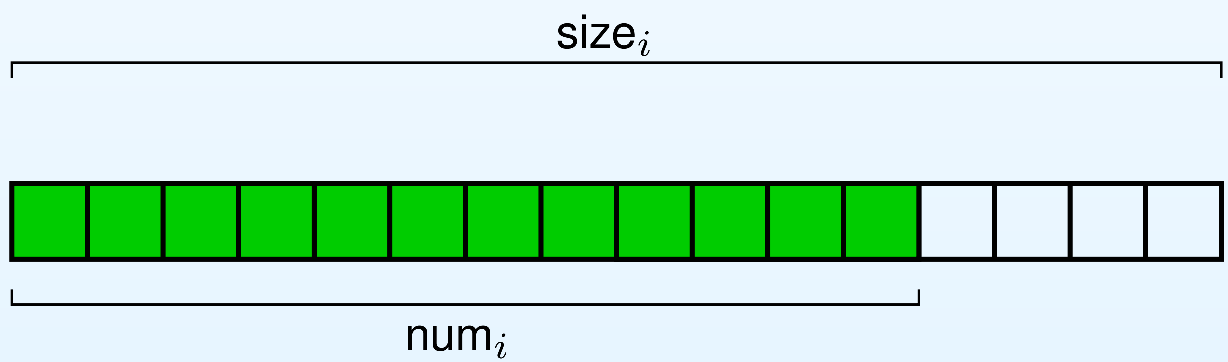
- For simplicity of illustration, we assume in the following that T is a stack with `INSERT` corresponding to `PUSH` and `DELETE` corresponding to `POP`
- All of our observations immediately generalize to other data structures such as heaps and hash tables

Dynamic tables: Insertions only

- Assume for now that T only supports `INSERT` operations
- Goal: T should be able to dynamically allocate a new larger array once the old array is too small to contain all elements of T
- Using the potential method, we show how to do this using only $O(1)$ average time per insertion

Notation in the following

- num_i: number of elements in T just after the ith operation
- size_i: size of array associated with T just after the ith operation
- $\alpha_i = \text{num}_i/\text{size}_i$: the *load factor* α_i of T just after the ith operation
- If $\text{size}_i = 0$, define $\alpha_i = 1$
- α_i indicates how big a fraction of the array is filled with elements
- Example (non-empty entries of T are green):



$$\alpha_i = 3/4$$

Table expansion

- Initially, T has an empty array
- Just prior to inserting the ith element, if num$_{i-1}$ $=$ size$_{i-1}$ (equivalently, if $\alpha_{i-1} = 1$) then T is expanded:
  - A new array twice as big is allocated, i.e., size$_i$ $= 2$size$_{i-1}$
  - The elements from the old array are copied to the new array
  - The old array is deallocated

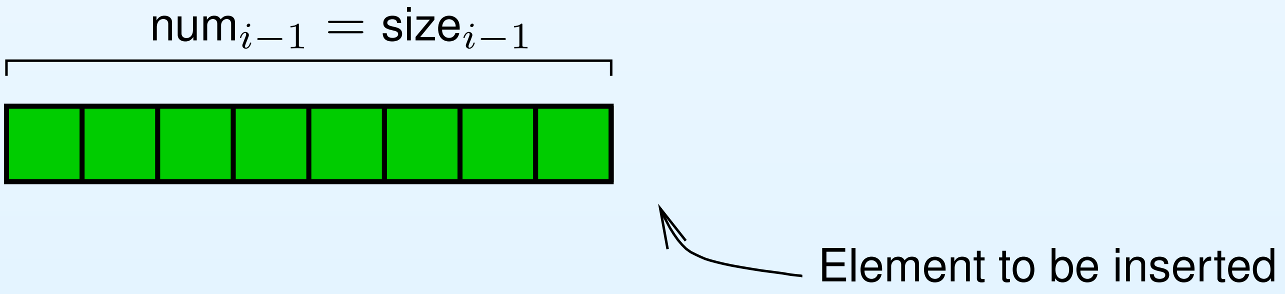
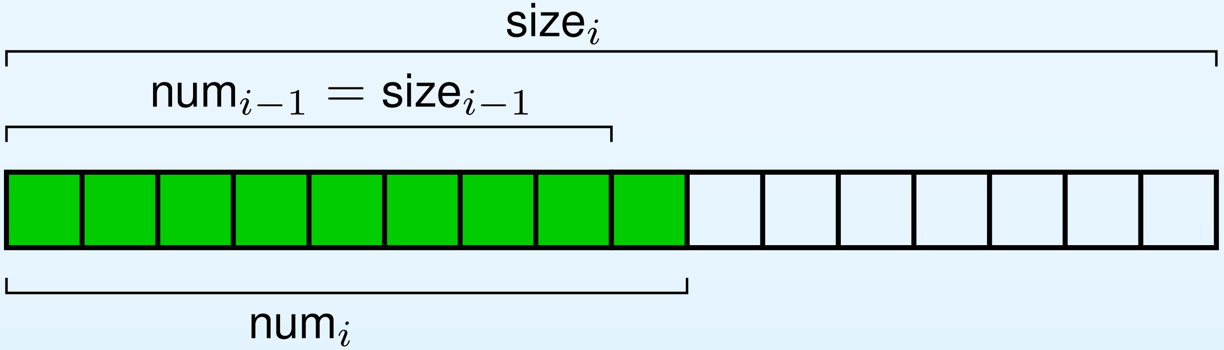


Table expansion

- Initially, T has an empty array
- Just prior to inserting the ith element, if $\text{num}_{i-1} = \text{size}_{i-1}$ (equivalently, if $\alpha_{i-1} = 1$) then T is expanded:
  - A new array twice as big is allocated, i.e., $\text{size}_i = 2\text{size}_{i-1}$
  - The elements from the old array are copied to the new array
  - The old array is deallocated



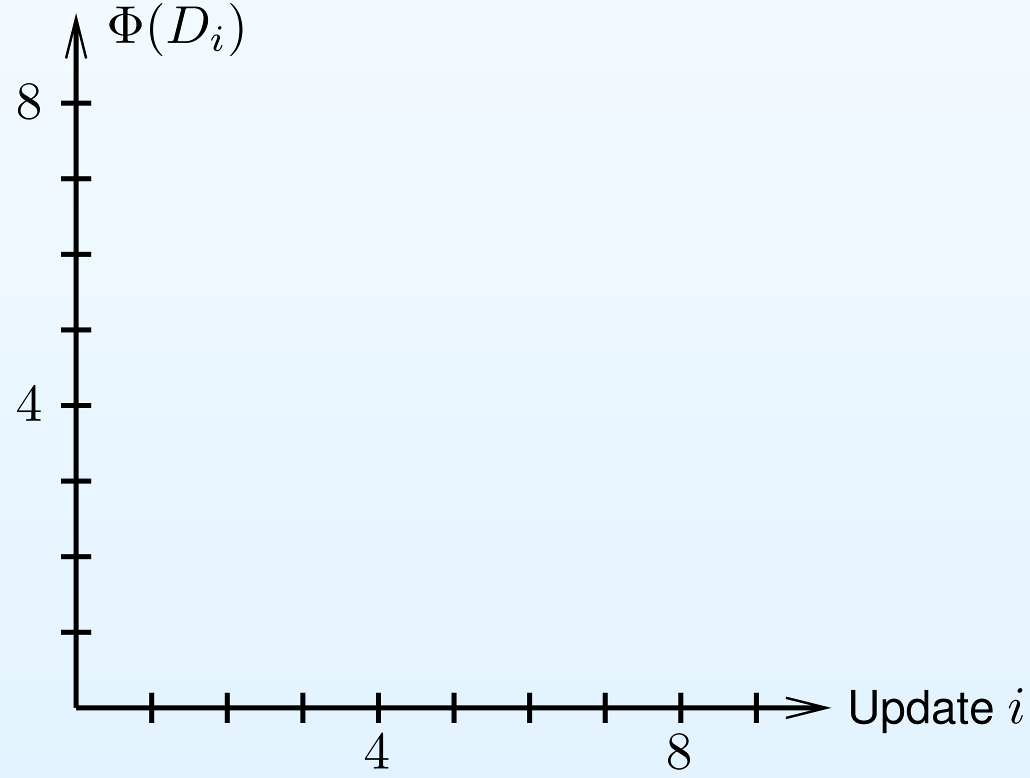
The potential method applied to the dynamic table example

$+$

- Table expansion plus the insertion of the new element takes worst-case time $O(\text{num}_i)$ $5 + 1 = 6$
- Hence, if the ith operation requires a table expansion, we can set $c_i = \text{num}_i$ (after scaling by a constant factor)
- If no table expansion is required, $c_i = 1$
- We will now show how to bound the total cost $\sum_{i=1}^{n} c_i$ of a sequence of n operations using the potential method

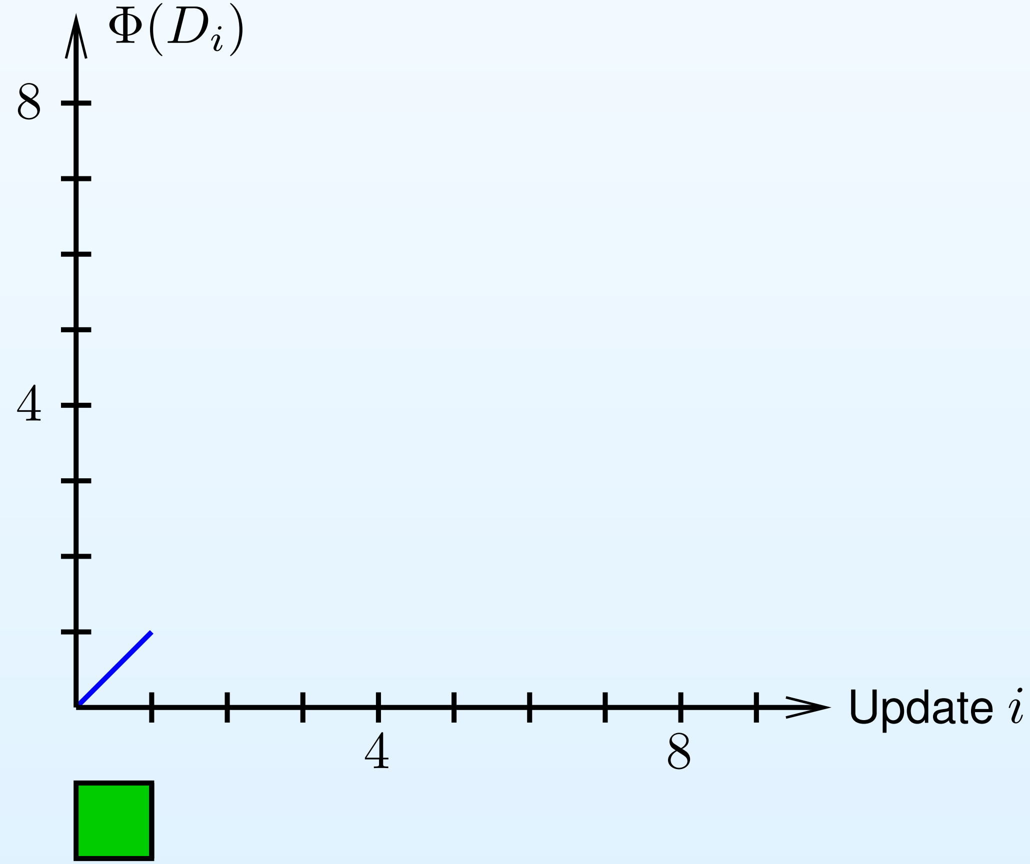
The potential method applied to the dynamic table example

- Let D_0 be the initial empty table T and for $i = 1, \ldots, n$, let D_i be T just after the ith update
- We choose the potential function $\Phi(D_i) = 2\text{num}_i - \text{size}_i$
- Potential function as a function of i:



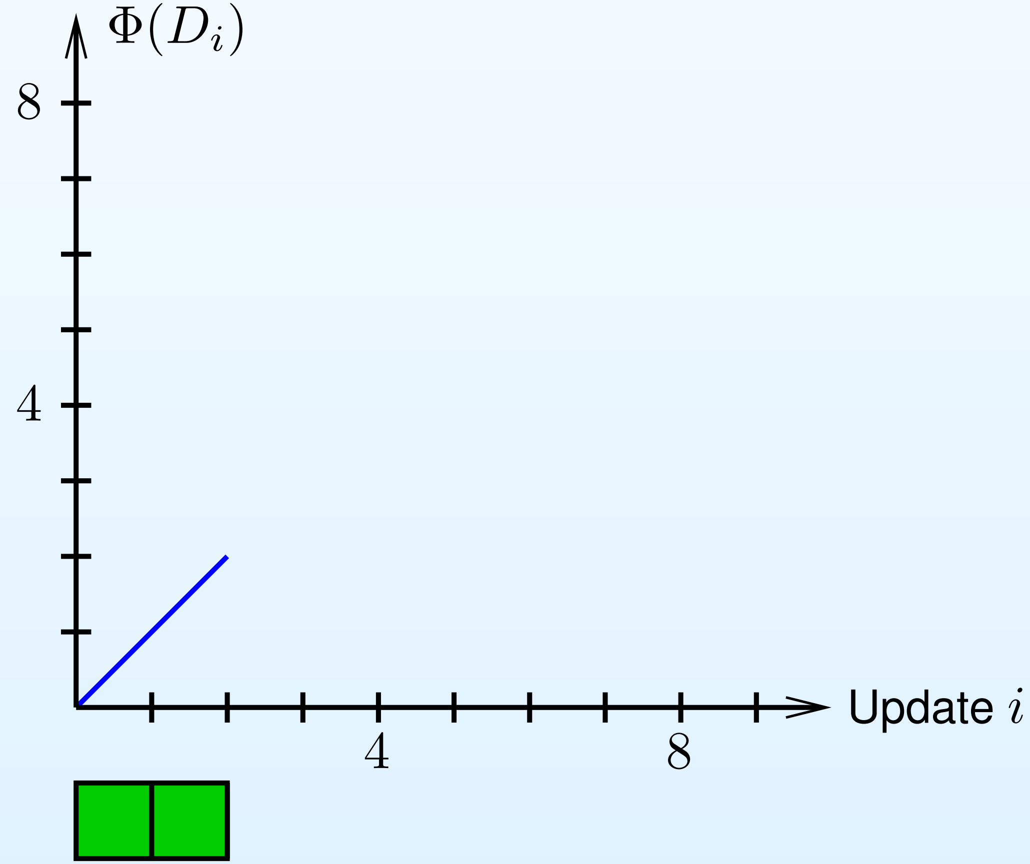
The potential method applied to the dynamic table example

- Let D_0 be the initial empty table T and for $i = 1, \ldots, n$, let D_i be T just after the ith update
- We choose the potential function $\Phi(D_i) = 2\text{num}_i - \text{size}_i$
- Potential function as a function of i:



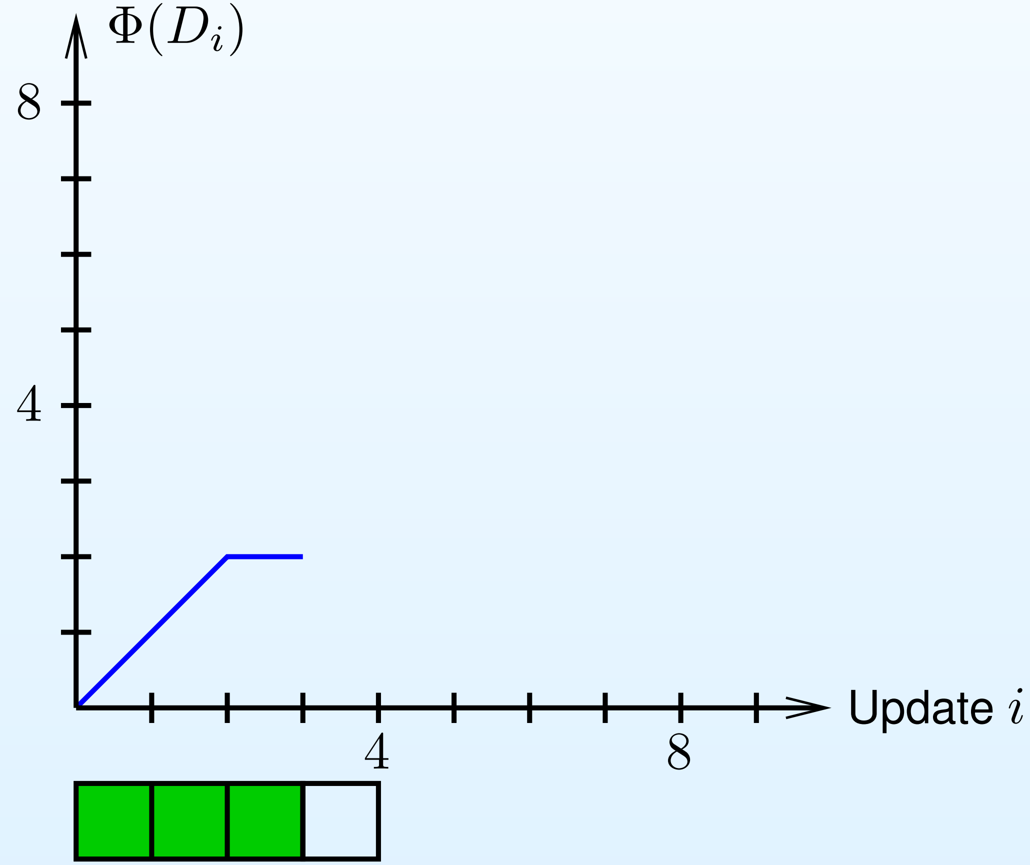
The potential method applied to the dynamic table example

- Let D_0 be the initial empty table T and for $i = 1, \ldots, n$, let D_i be T just after the ith update
- We choose the potential function $\Phi(D_i) = 2\text{num}_i - \text{size}_i$
- Potential function as a function of i:



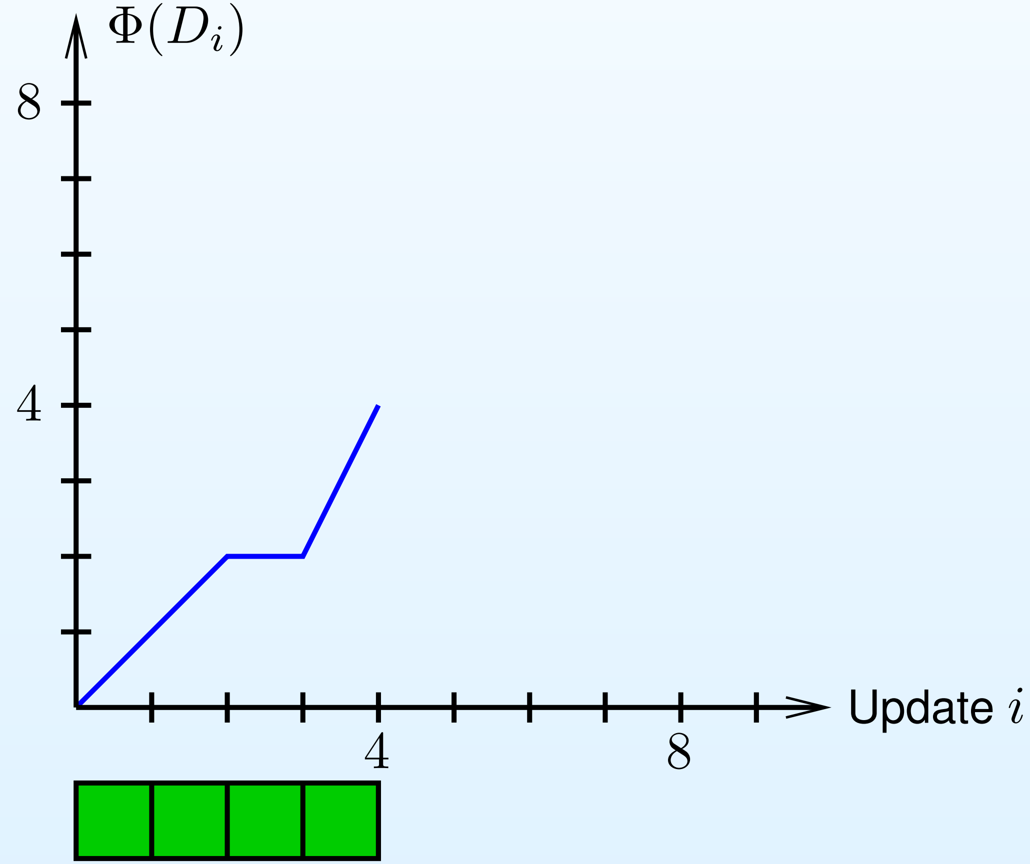
The potential method applied to the dynamic table example

- Let D_0 be the initial empty table T and for $i = 1, \ldots, n$, let D_i be T just after the ith update
- We choose the potential function $\Phi(D_i) = 2\text{num}_i - \text{size}_i$
- Potential function as a function of i:



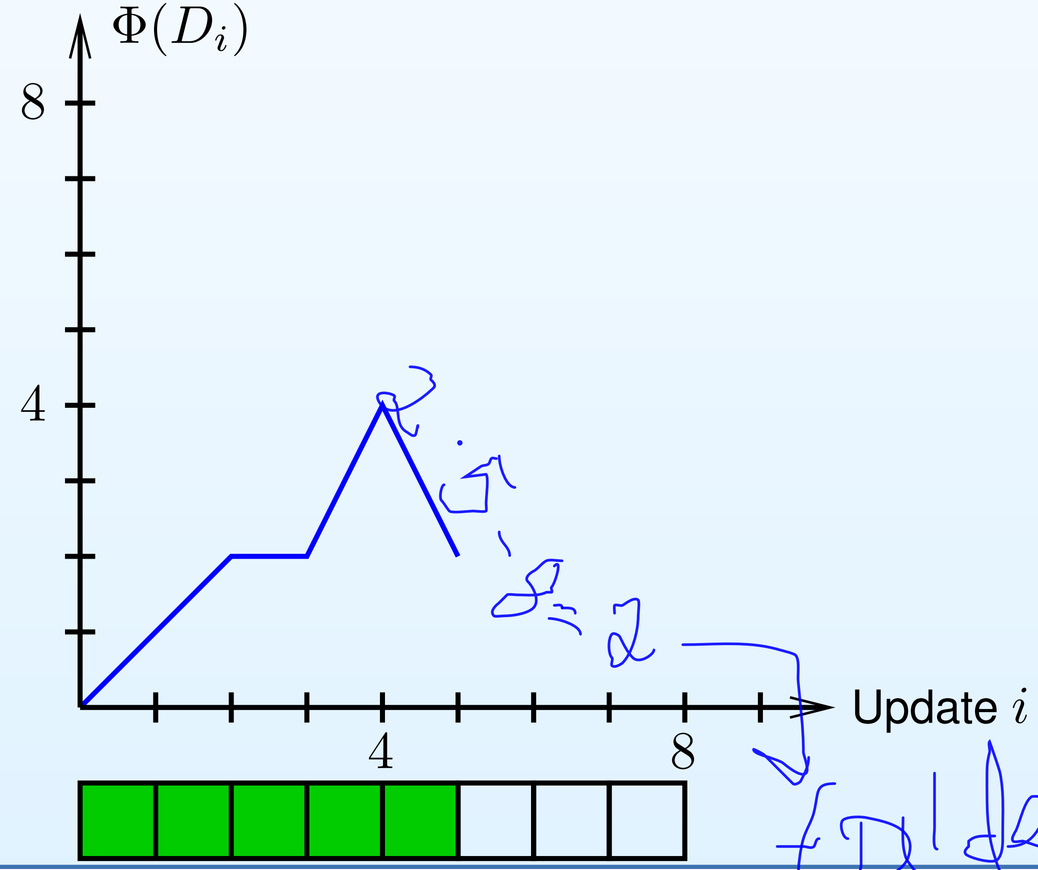
The potential method applied to the dynamic table example

- Let D_0 be the initial empty table T and for $i = 1, \ldots, n$, let D_i be T just after the ith update
- We choose the potential function $\Phi(D_i) = 2\text{num}_i - \text{size}_i$
- Potential function as a function of i:



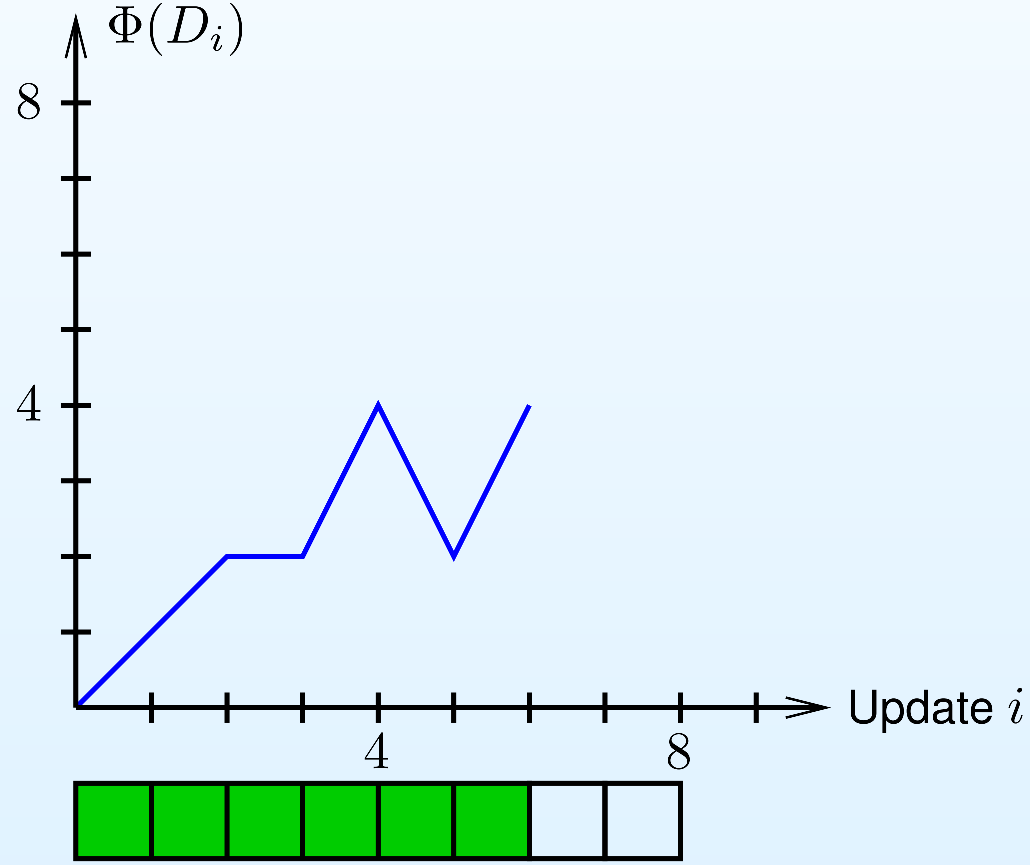
The potential method applied to the dynamic table example

- Let D_0 be the initial empty table T and for $i = 1, \ldots, n$, let D_i be T just after the ith update
- We choose the potential function $\Phi(D_i) = 2\text{num}_i - \text{size}_i$
- Potential function as a function of i:



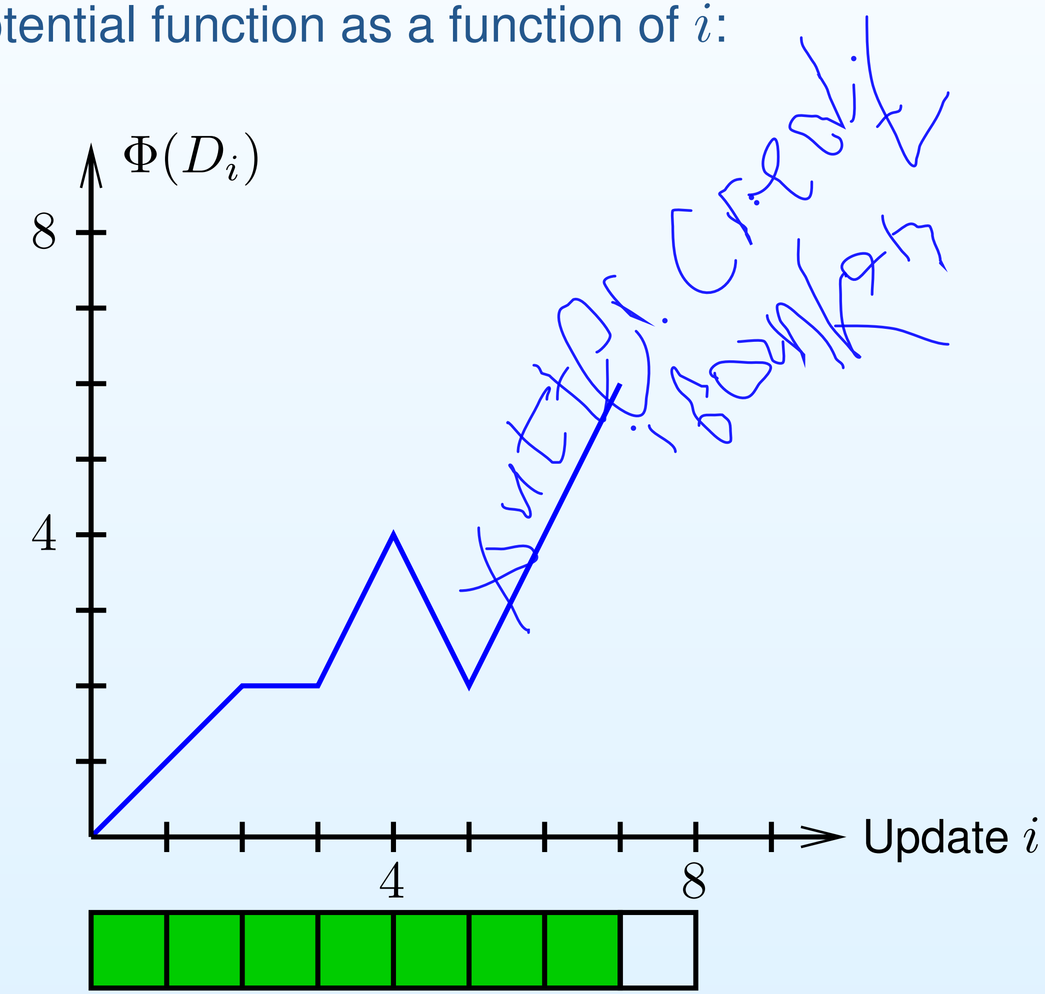
The potential method applied to the dynamic table example

- Let D_0 be the initial empty table T and for $i = 1, \ldots, n$, let D_i be T just after the ith update
- We choose the potential function $\Phi(D_i) = 2\text{num}_i - \text{size}_i$
- Potential function as a function of i:



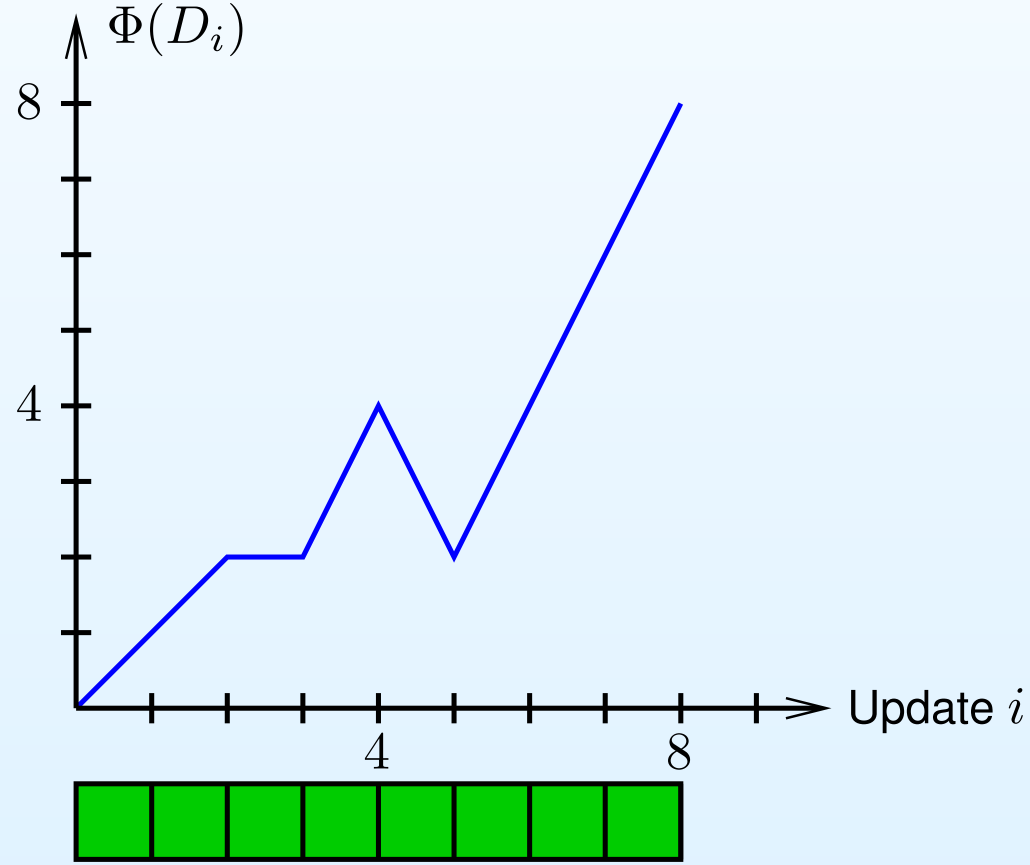
The potential method applied to the dynamic table example

- Let D_0 be the initial empty table T and for $i = 1, \ldots, n$, let D_i be T just after the ith update
- We choose the potential function $\Phi(D_i) = 2\text{num}_i - \text{size}_i$
- Potential function as a function of i:



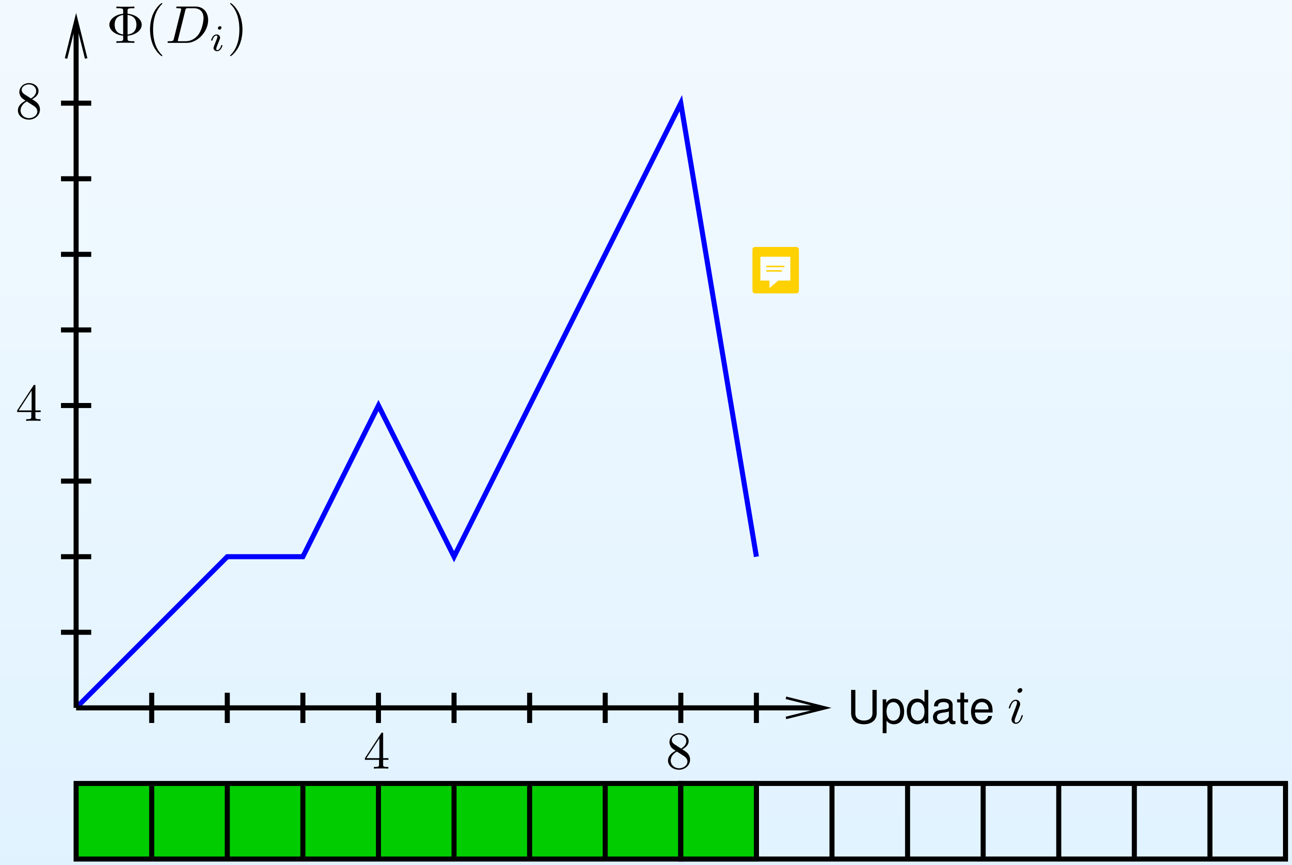
The potential method applied to the dynamic table example

- Let D_0 be the initial empty table T and for $i = 1, \ldots, n$, let D_i be T just after the ith update
- We choose the potential function $\Phi(D_i) = 2\text{num}_i - \text{size}_i$
- Potential function as a function of i:



The potential method applied to the dynamic table example

- Let D_0 be the initial empty table T and for $i = 1, \ldots, n$, let D_i be T just after the ith update
- We choose the potential function $\Phi(D_i) = 2\text{num}_i - \text{size}_i$
- Potential function as a function of i:



The potential function is valid

- We show that Φ is valid by proving that $\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i
- $\Phi(D_0) = 0$ is clear since T is initially empty
- Since T is always at least half full, $\Phi(D_i) \geq 0$ for all i
- It follows that Φ is valid and hence $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$
- We can thus upper bound the total actual cost $\sum_{i=1}^{n} c_i$ by upper bounding $\sum_{i=1}^{n} \hat{c}_i$

The potential method applied to the dynamic table example

- Recall that $\Phi(D_i) = 2\text{num}_i - \text{size}_i$ for $i = 0, \ldots, n$
- Consider the ith insertion operation, $i \in \{1, \ldots, n\}$
- If no table expansion occurs, the amortized cost $\hat{c}_i$ is

$$
\begin{aligned}
\hat{c}_i &= c_i + \overbrace{\Phi(D_i) - \Phi(D_{i-1})}^{2} \\
&= 1 + (2\text{num}_i - \text{size}_i) - (2\text{num}_{i-1} - \text{size}_{i-1}) \\
&= 1 + (\cancel{2\text{num}_i} - \cancel{\text{size}_i}) - (2(\cancel{\text{num}_i} - 1) - \cancel{\text{size}_i}) \\
&= 3
\end{aligned}
$$

$1 + 2$

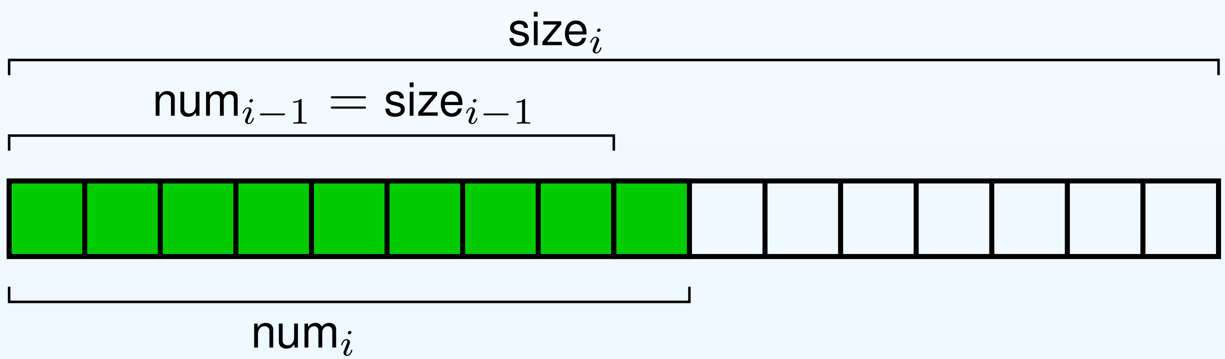
The potential method applied to the dynamic table example

- Recall that $\Phi(D_i) = 2\text{num}_i - \text{size}_i$ for $i = 0, \ldots, n$
- Consider the ith insertion operation, $i \in \{1, \ldots, n\}$
- If table expansion does occur and $i = 1$, the amortized cost $\hat{c}_i$ is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 2$$

The potential method applied to the dynamic table example

- Recall that $\Phi(D_i) = 2\text{num}_i - \text{size}_i$ for $i = 0, \ldots, n$
- Consider the ith insertion operation, $i \in \{1, \ldots, n\}$
- Now assume that table expansion occurs and $i > 1$:



- The amortized cost $\hat{c}_i$ is

$$
\begin{aligned}
\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&= \text{num}_i + (2\text{num}_i - \text{size}_i) - (2\text{num}_{i-1} - \text{size}_{i-1}) \\
&= \text{num}_i + (2\text{num}_i - 2(\text{num}_i - 1)) - (2(\text{num}_i - 1) - (\text{num}_i - 1)) \\
&= 3
\end{aligned}
$$

Table contraction

- So far, we only considered the operation `INSERT`
- Now, we also include the operation `DELETE`
- To save memory, we want the array to contract to a smaller array whenever the load factor $\alpha_i = \text{num}_i / \text{size}_i$ becomes sufficiently small
- Suppose we contract as soon as $\alpha_i < 1/2$
- What is the problem?

Table contraction – the bad case

- Let n be an exact power of 2
- Suppose the first $n/2+1$ operations are of the type `INSERT`
- Suppose the last $n/2-1$ operations form the sequence:

$$\texttt{DELETE}, \texttt{DELETE}, \texttt{INSERT}, \texttt{INSERT},$$
$$\texttt{DELETE}, \texttt{DELETE}, \texttt{INSERT}, \texttt{INSERT}, \ldots$$

- The first two of these causes a contraction, the next two an expansion, the next two a contraction, and so on:

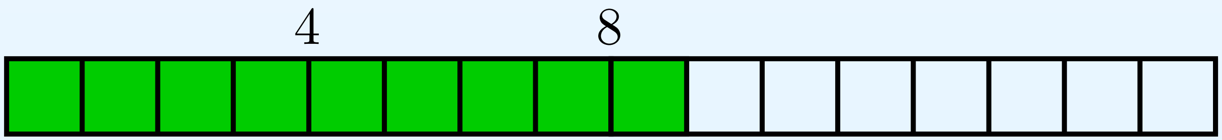


Table contraction – the bad case

- Let n be an exact power of 2
- Suppose the first $n/2+1$ operations are of the type $\texttt{INSERT}$
- Suppose the last $n/2-1$ operations form the sequence:

$$\texttt{DELETE}, \texttt{DELETE}, \texttt{INSERT}, \texttt{INSERT},$$
$$\texttt{DELETE}, \texttt{DELETE}, \texttt{INSERT}, \texttt{INSERT}, \ldots$$

- The first two of these causes a contraction, the next two an expansion, the next two a contraction, and so on:

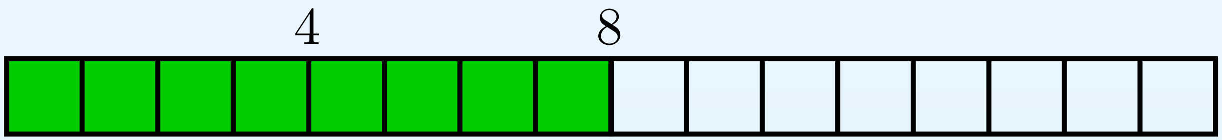


Table contraction – the bad case

- Let n be an exact power of 2
- Suppose the first $n/2+1$ operations are of the type $\mathtt{INSERT}$
- Suppose the last $n/2-1$ operations form the sequence:

$$\mathtt{DELETE}, \mathtt{DELETE}, \mathtt{INSERT}, \mathtt{INSERT},$$
$$\mathtt{DELETE}, \mathtt{DELETE}, \mathtt{INSERT}, \mathtt{INSERT}, \ldots$$

- The first two of these causes a contraction, the next two an expansion, the next two a contraction, and so on:

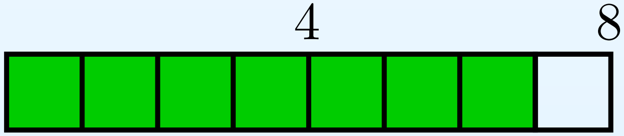


Table contraction – the bad case

- Let n be an exact power of 2
- Suppose the first $n/2+1$ operations are of the type $\texttt{INSERT}$
- Suppose the last $n/2-1$ operations form the sequence:

$$\texttt{DELETE}, \texttt{DELETE}, \texttt{INSERT}, \texttt{INSERT},$$
$$\texttt{DELETE}, \texttt{DELETE}, \texttt{INSERT}, \texttt{INSERT}, \ldots$$

- The first two of these causes a contraction, the next two an expansion, the next two a contraction, and so on:

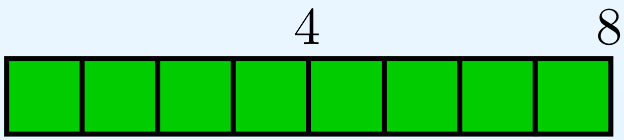


Table contraction – the bad case

- Let n be an exact power of 2
- Suppose the first $n/2+1$ operations are of the type $\texttt{INSERT}$
- Suppose the last $n/2-1$ operations form the sequence:

$$\texttt{DELETE}, \texttt{DELETE}, \texttt{INSERT}, \texttt{INSERT},$$
$$\texttt{DELETE}, \texttt{DELETE}, \texttt{INSERT}, \texttt{INSERT}, \ldots$$

- The first two of these causes a contraction, the next two an expansion, the next two a contraction, and so on:

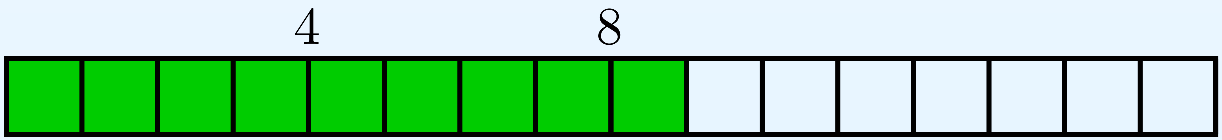


Table contraction – the bad case

- Let n be an exact power of 2
- Suppose the first $n/2+1$ operations are of the type `INSERT`
- Suppose the last $n/2-1$ operations form the sequence:

$$\texttt{DELETE}, \texttt{DELETE}, \texttt{INSERT}, \texttt{INSERT},$$
$$\texttt{DELETE}, \texttt{DELETE}, \texttt{INSERT}, \texttt{INSERT}, \ldots$$

- The first two of these causes a contraction, the next two an expansion, the next two a contraction, and so on:

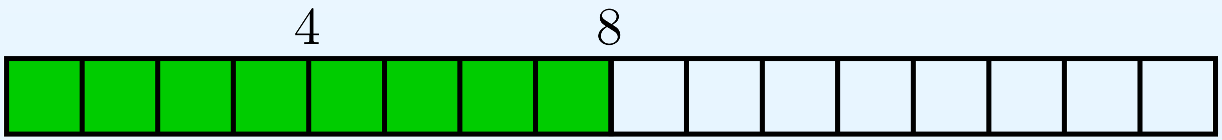


Table contraction – the bad case

- Let n be an exact power of 2
- Suppose the first $n/2 + 1$ operations are of the type INSERT
- Suppose the last $n/2 - 1$ operations form the sequence:

$$\texttt{DELETE}, \texttt{DELETE}, \texttt{INSERT}, \texttt{INSERT},$$
$$\texttt{DELETE}, \texttt{DELETE}, \texttt{INSERT}, \texttt{INSERT}, \ldots$$

- The first two of these causes a contraction, the next two an expansion, the next two a contraction, and so on:

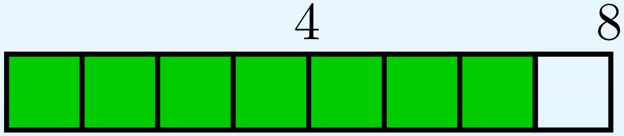
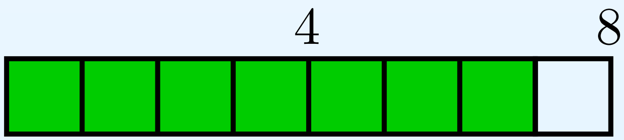


Table contraction – the bad case

- Let n be an exact power of 2
- Suppose the first $n/2+1$ operations are of the type $\texttt{INSERT}$
- Suppose the last $n/2-1$ operations form the sequence:

$$\texttt{DELETE}, \texttt{DELETE}, \texttt{INSERT}, \texttt{INSERT},$$
$$\texttt{DELETE}, \texttt{DELETE}, \texttt{INSERT}, \texttt{INSERT}, \ldots$$

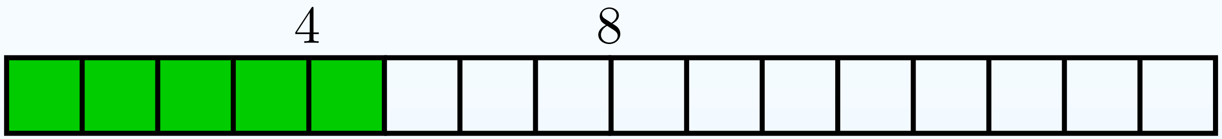
- The first two of these causes a contraction, the next two an expansion, the next two a contraction, and so on:



- This is slow, even on average

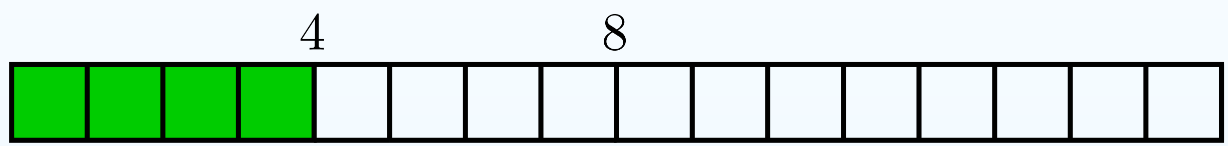
Modifying the data structure

- We modify our approach by contracting when a `DELETE` operation causes the load factor α_i to become less than $1/4$



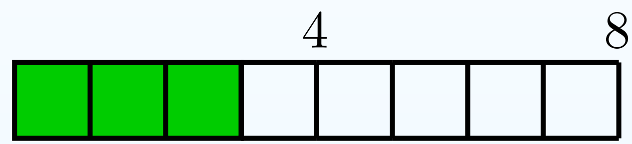
Modifying the data structure

- We modify our approach by contracting when a `DELETE` operation causes the load factor α_i to become less than $1/4$



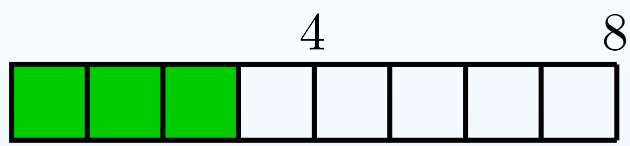
Modifying the data structure

- We modify our approach by contracting when a DELETE operation causes the load factor α_i to become less than $1/4$



Modifying the data structure

- We modify our approach by contracting when a `DELETE` operation causes the load factor α_i to become less than $1/4$



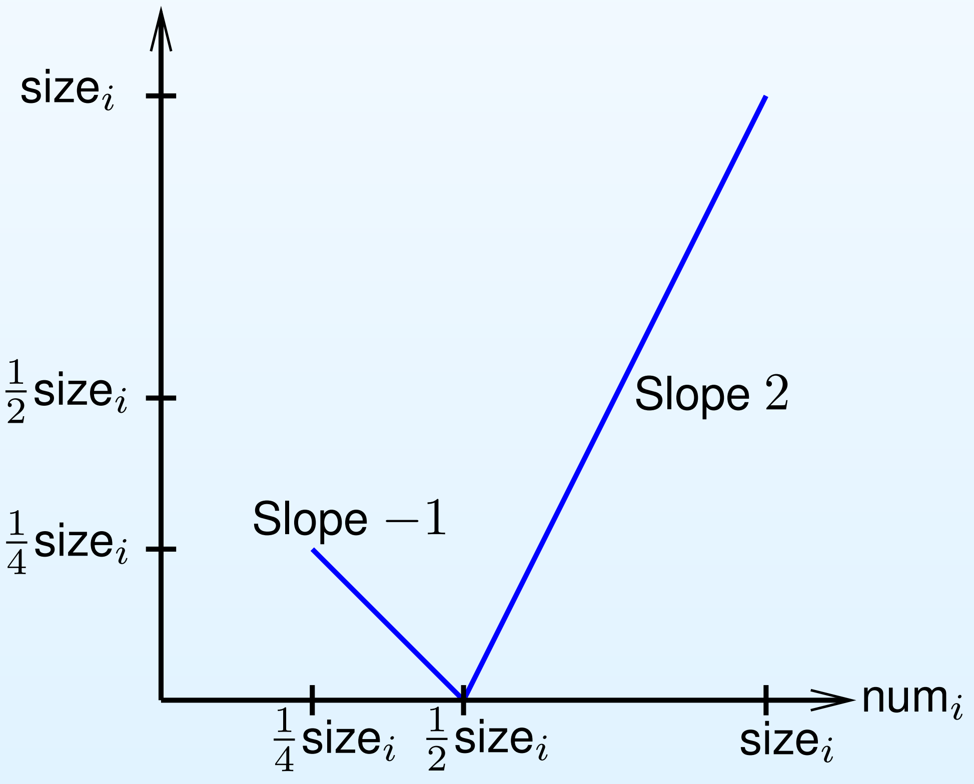
- We expand in the same way as before, i.e., when an `INSERT` operation is applied to a full table T
- The modification avoids the problem on the previous slide
- Using the potential method, we show that the average time per operation is $O(1)$ Wow!

The new potential function supporting `INSERT` and `DELETE`

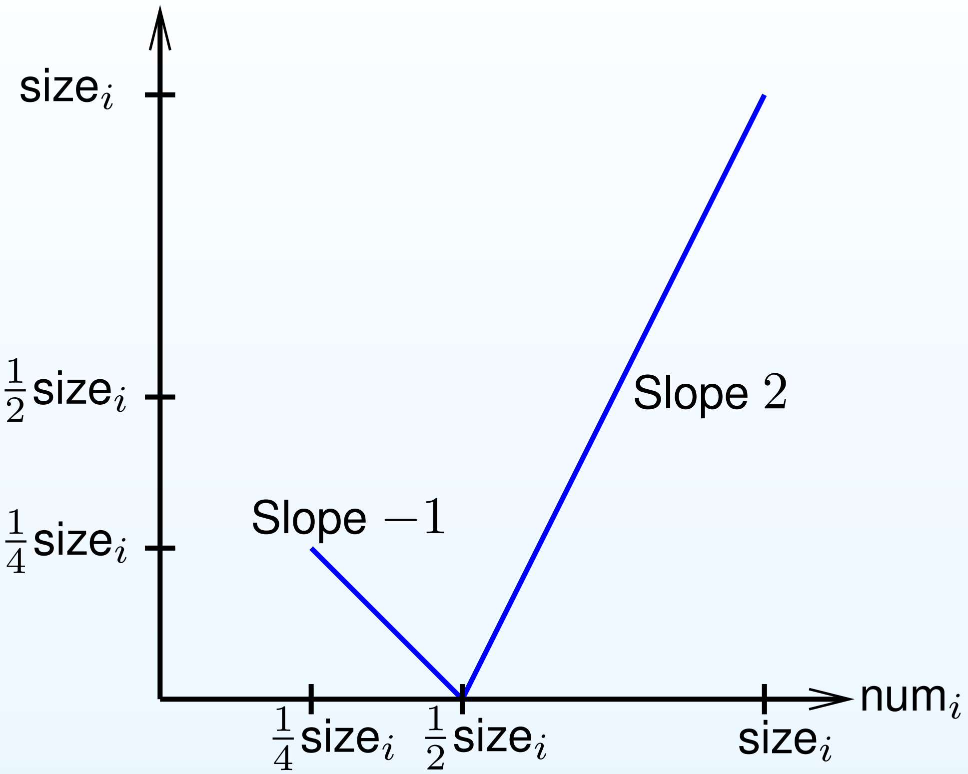
- We pick the following potential function:

$$\Phi(D_i) = \begin{cases} 2\text{num}_i - \text{size}_i & \text{if } \alpha_i \geq 1/2 \\ \text{size}_i/2 - \text{num}_i & \text{if } \alpha_i < 1/2 \end{cases}$$

- Plotting the right-hand size as a function of num_i in the range $[\frac{1}{4}\text{size}_i, \text{size}_i]$ (in this range, size_i stays the same):



High-level proof that amortized cost is $\Theta(1)$



- Just before a table expansion/contraction, Φ has value num_i
- Just afterwards, the table is roughly half full and so the value of Φ drops to roughly 0
- This drop in potential pays for the expansion/contraction
- When no expansion/contraction occurs, the amortized cost is at most 3 since the slope of Φ has absolute value at most 2
- We now give a detailed proof of this

The potential function is valid

- Φ is a valid potential function, i.e., $\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i (see plot on previous slide)
- Hence,

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

- We will show that $\hat{c}_i \leq 3$ for $i = 1, \ldots, n$ so that

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i \leq 3n = O(n)$$

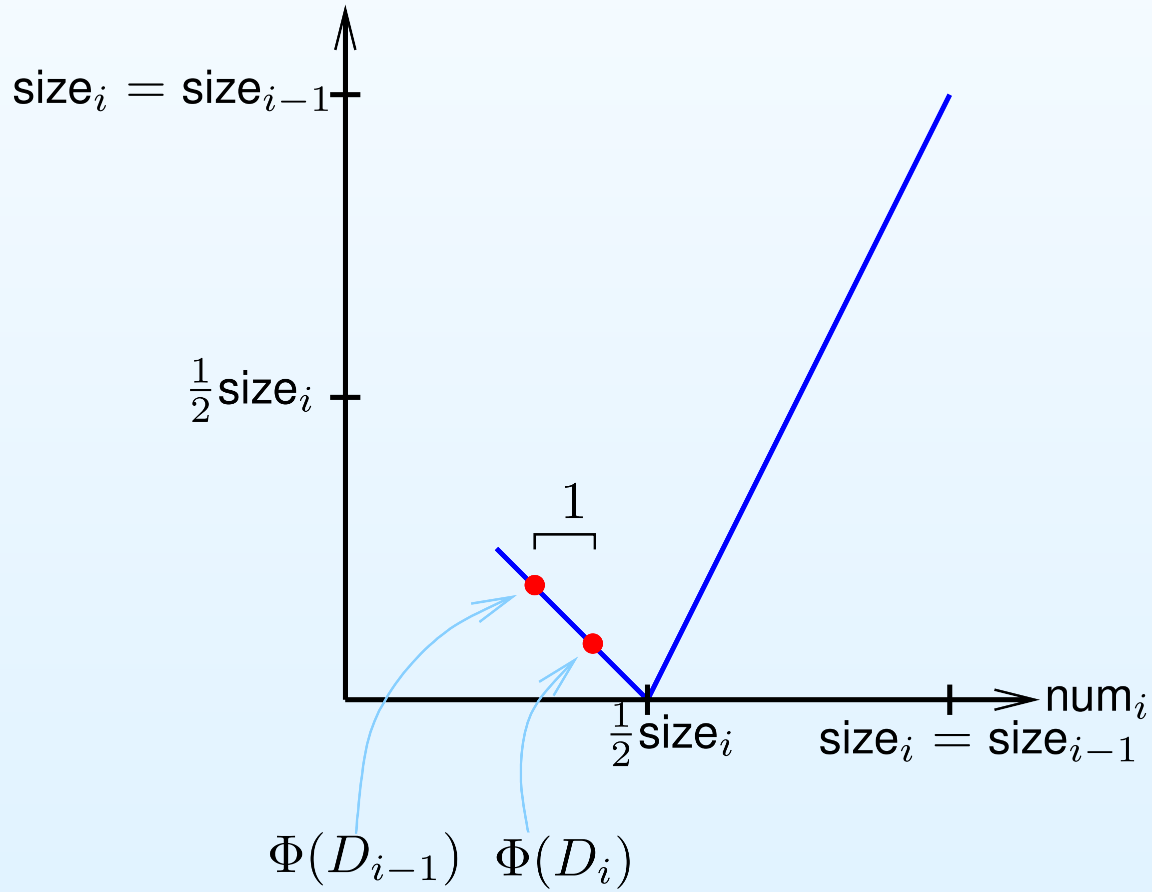
- This will show that the amortized cost of each update is $O(n)/n = O(1)$

Amortized cost of `INSERT`

- Consider the ith operation and assume that it is `INSERT`
- Assume first $\alpha_{i-1} \geq 1/2$ and $\alpha_i \geq 1/2$
- Then $\Phi(D_{i-1}) = 2\text{num}_{i-1} - \text{size}_{i-1}$ and $\Phi(D_i) = 2\text{num}_i - \text{size}_i$
- As shown earlier (when we only allowed insertions), $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq 3$

Amortized cost of `INSERT`

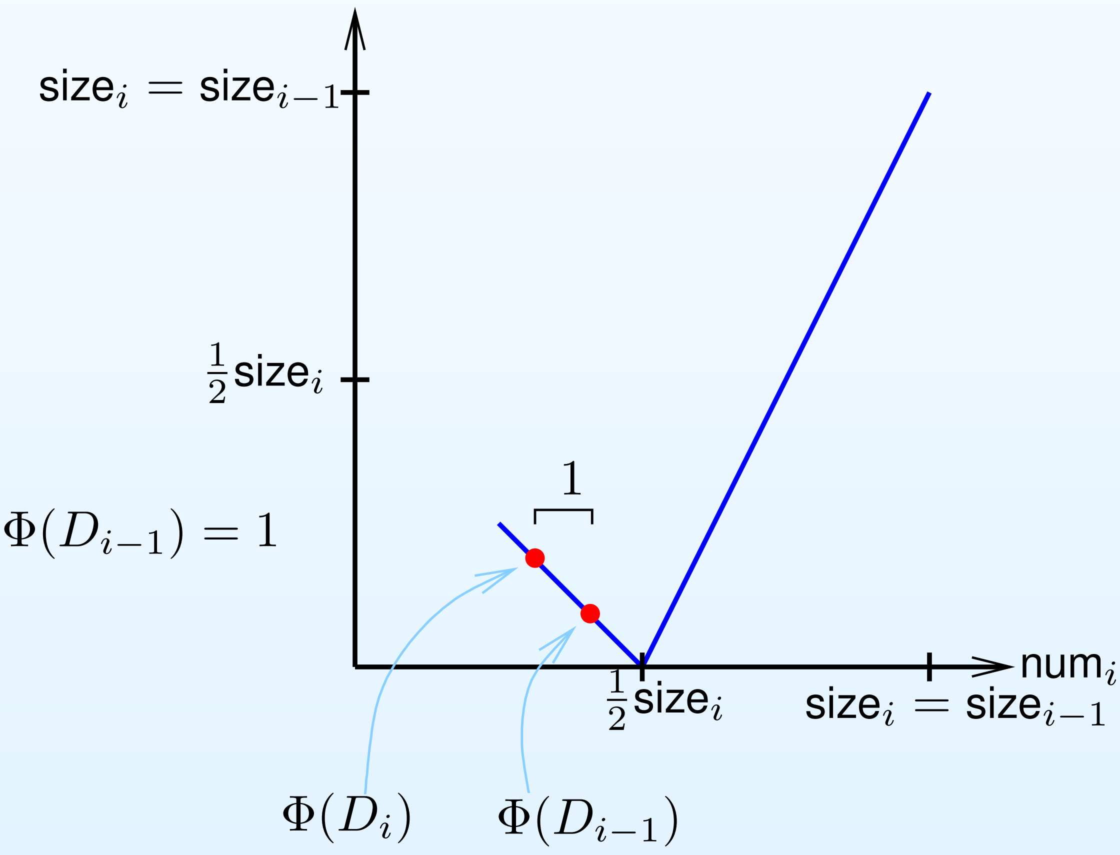
- Now assume that not both $\alpha_{i-1} \geq 1/2$ and $\alpha_i \geq 1/2$
- Then $\alpha_{i-1} < 1/2$ and $\alpha_i \leq 1/2$ so table expansion cannot occur
- We have $\Phi(D_i) - \Phi(D_{i-1}) = -1$ so $\hat{c}_i = 1 - 1 = 0$:



Amortized cost of `DELETE`

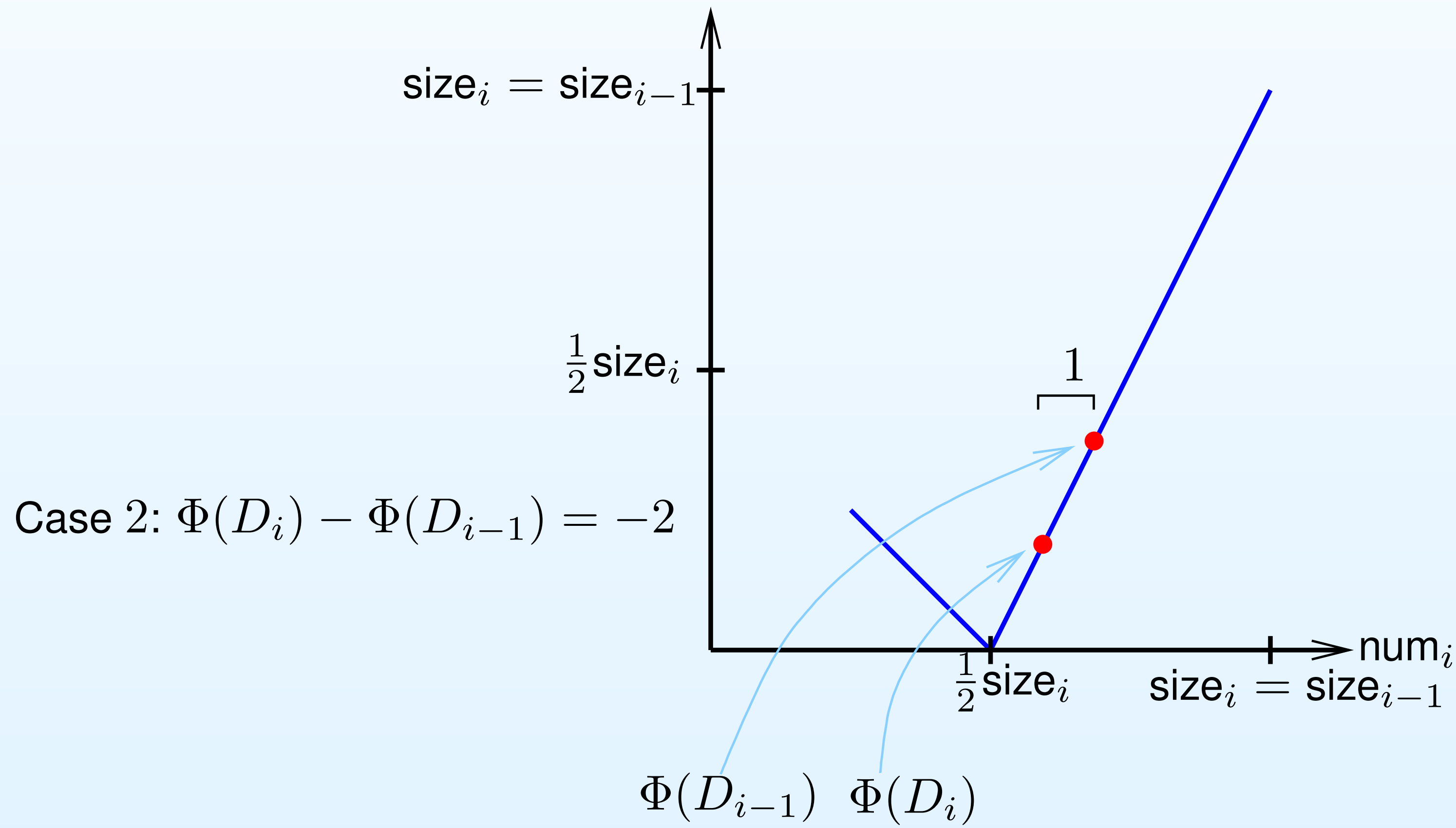
- Next, assume that the ith operation is `DELETE`
- If no table contraction occurs, we have $c_i = 1$
- Thus, $\hat{c}_i \leq 2$ since $\Phi(D_i) - \Phi(D_{i-1}) \leq 1$:

Case 1: $\Phi(D_i) - \Phi(D_{i-1}) = 1$



Amortized cost of `DELETE`

- Next, assume that the ith operation is `DELETE`
- If no table contraction occurs, we have $c_i = 1$
- Thus, $\hat{c}_i \leq 2$ since $\Phi(D_i) - \Phi(D_{i-1}) \leq 1$:



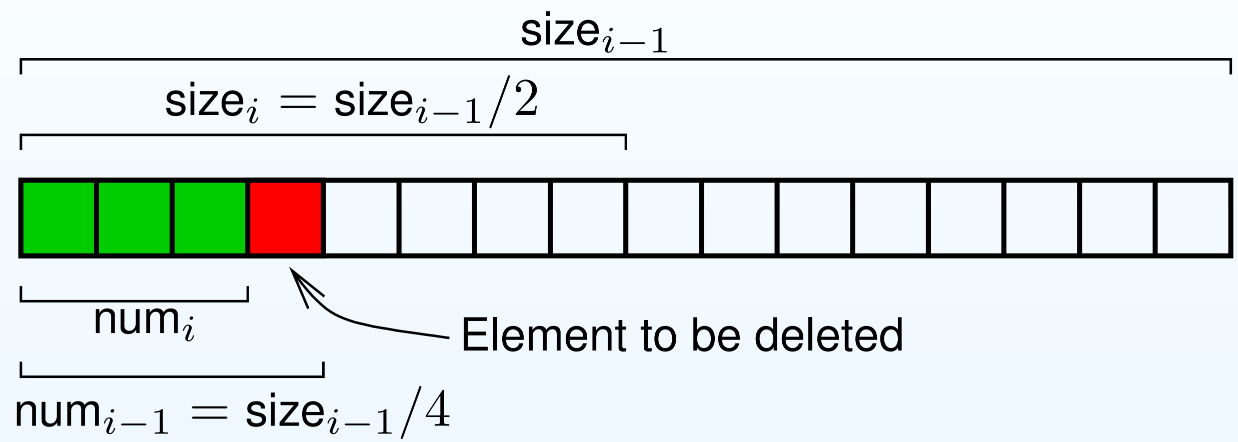
Amortized cost of DELETE

- Now, assume that table contraction occurs
- In this case, one item is deleted and num_i items are moved to the contracted array
- Thus, the actual cost is $c_i = \text{num}_i + 1$
- If $\text{num}_i = 0$ then since $\text{size}_{i-1} \in \{1, 2, 4\}$, we have $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 0 - \Phi(D_{i-1}) \leq 1$
- Now, assume $\text{num}_i > 0$
- Then $\alpha_{i-1} = 1/4 < 1/2$ and $\alpha_i < 1/2$ so we have $\Phi(D_i) = \text{size}_i/2 - \text{num}_i$ and $\Phi(D_{i-1}) = \text{size}_{i-1}/2 - \text{num}_{i-1}$
- The amortized cost is then

$$
\begin{aligned}
\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&= (\text{num}_i + 1) + (\text{size}_i/2 - \text{num}_i) - (\text{size}_{i-1}/2 - \text{num}_{i-1})
\end{aligned}
$$

Amortized cost of DELETE

- Illustration of table contraction:



- We express everything in terms of num$_i$:

$$
\begin{aligned}
\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&= \text{num}_i + 1 + (\text{size}_i/2 - \text{num}_i) - (\text{size}_{i-1}/2 - \text{num}_{i-1}) \\
&= \text{num}_i + 1 + ((\text{num}_i + 1) - \text{num}_i) \\
&\quad - (2(\text{num}_i + 1) - (\text{num}_i + 1)) \\
&= 1
\end{aligned}
$$

Bounding the average cost for dynamic table

- We have shown that $\hat{c}_i \leq 3$ for each operation i
- We can thus bound the total actual cost by

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i \leq 3n$$

- The average running time per operation is thus $O(n)/n = O(1)$

Plan for the lecture on February 27

- Fibonacci Heaps
- We will apply the potential method to analyze the performance of this data structure