Amortized Complexity

Advanced Algorithms and Data Structures - Lecture 7

Venanzio Capretta

Monday 11 November 2019

School of Computer Science, University of Nottingham

Amortized Analysis

Some data structures have operations with high worst-case complexity,
but when we do a sequence of operations,
the average cost is small:
one costly operation can be compensated by many cheap ones

Amortized Analysis assigns to each operation

- An amortized cost that
- may be smaller than actual cost
- but takes into account a way of averaging computation steps over several operations.

Amortized Cost

Amortized cost must be defined so that the total amortized cost of a sequence of operations is larger or equal to the actual cost:
We perform a sequence of operations on the data:

$$f_1, f_2, f_3, \cdots, f_m$$

Each operation f_i has an actual cost t_i
We assign to it an amortized cost a_i
We must guarantee that

$$\sum_{i=1}^{m} a_i \geq \sum_{i=1}^{m} t_i$$

So the amortized complexity is an overestimation of the actual complexity

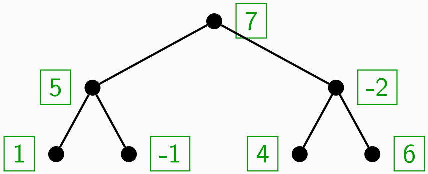
Accounting method

There are two main methods of amortized analysis:
the accounting method and the potential method

THE ACCOUNTING METHOD (also called *banker's method*)

We imagine that every location in the data structure has a store where we can save *credits*, virtual time steps that can be used at a different time

For example, a tree structure will store credits in each node:



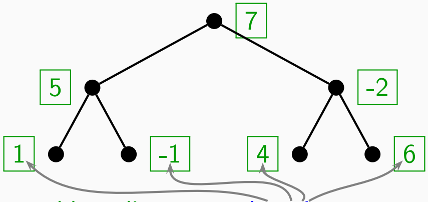
Accounting method

There are two main methods of amortized analysis:
the accounting method and the potential method

THE ACCOUNTING METHOD (also called *banker's method*)

We imagine that every location in the data structure has a store
where we can save *credits*, virtual time steps
that can be used at a different time

For example, a tree structure will store credits in each node:



Every operation can add credits c_i to a location
or use credits $\overline{c_i}$ from a location

Credit Accounts

Amortized cost of operation f_i:

$$a_i = t_i + c_i - \overline{c_i}$$

where

- t_i is the actual time cost of f_i
- c_i is the number of credits allocated by operation f_i
- $\overline{c_i}$ is the number of credits spent by operation f_i

For the total amortized cost to be an overestimate of actual cost

$$\sum_{i=1}^{m} a_i \geq \sum_{i=1}^{m} t_i$$

The overall credit must always be positive (never in debt):

$$\sum_{i=1}^{m} c_i \geq \sum_{i=1}^{m} \overline{c_i}$$

Potential Method

Also called *physicist's method*

We associate a potential function to a data structure D

$$\phi : D \to \mathbb{R}_{\geq 0}$$

Intuitively, the potential gives us some *complexity for free*:
We can compensate for a costly operation by using some of the potential

Cheap operations may increase the potential, so it can be used later

Usually we define ϕ so that the initial (empty) data structure has potential zero

Variation of Potential

Amortized cost of an operation f_i:

$$a_i = t_i + \phi(d_i) - \phi(d_{i-1})$$

- t_i is the actual cost
- $d_{i-1} \in D$ is the state of the data structure before operation f_i
- $d_i \in D$ is the state of the data structure after operation f_i
- So $\phi(d_i) - \phi(d_{i-1})$ is the change of potential

The actual cost is $t_i = a_i + \phi(d_{i-1}) - \phi(d_i)$
(The amortized cost minus the change of potential
ie, me must spend actual time to charge the potential)

Sequence of Operations

If we perform several operation in sequence,
starting with the data structure in state d_0

$$d_0 \xmapsto{f_1} d_1 \xmapsto{f_2} d_2 \xmapsto{f_3} \cdots \xmapsto{f_m} d_m$$

Sequence of Operations

If we perform several operation in sequence,
starting with the data structure in state d_0

$$d_0 \xmapsto{f_1} d_1 \xmapsto{f_2} d_2 \xmapsto{f_3} \cdots \xmapsto{f_m} d_m$$

The total actual cost is:

$$\sum_{i=1}^{m} t_i$$

Sequence of Operations

If we perform several operation in sequence,
starting with the data structure in state d_0

$$d_0 \xmapsto{f_1} d_1 \xmapsto{f_2} d_2 \xmapsto{f_3} \cdots \xmapsto{f_m} d_m$$

The total actual cost is:

$$\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} (a_i + \phi(d_{i-1}) - \phi(d_i))$$

Sequence of Operations

If we perform several operation in sequence,
starting with the data structure in state d_0

$$d_0 \xmapsto{f_1} d_1 \xmapsto{f_2} d_2 \xmapsto{f_3} \cdots \xmapsto{f_m} d_m$$

The total actual cost is:

$$\begin{aligned}
\sum_{i=1}^{m} t_i &= \sum_{i=1}^{m} (a_i + \phi(d_{i-1}) - \phi(d_i)) \\
&= \sum_{i=1}^{m} a_i + \sum_{i=1}^{m} (\phi(d_{i-1}) - \phi(d_i)) \quad \text{(telescoping summation)}
\end{aligned}$$

Sequence of Operations

If we perform several operation in sequence,
starting with the data structure in state d_0

$$d_0 \xmapsto{f_1} d_1 \xmapsto{f_2} d_2 \xmapsto{f_3} \cdots \xmapsto{f_m} d_m$$

The total actual cost is:

$$\begin{aligned}
\sum_{i=1}^{m} t_i &= \sum_{i=1}^{m} (a_i + \phi(d_{i-1}) - \phi(d_i)) \\
&= \sum_{i=1}^{m} a_i + \sum_{i=1}^{m} (\phi(d_{i-1}) - \phi(d_i)) \quad \text{(telescoping summation)} \\
&= \sum_{i=1}^{m} a_i + \phi(d_0) - \phi(d_m)
\end{aligned}$$

Sequence of Operations

If we perform several operation in sequence,
starting with the data structure in state d_0

$$d_0 \xmapsto{f_1} d_1 \xmapsto{f_2} d_2 \xmapsto{f_3} \cdots \xmapsto{f_m} d_m$$

The total actual cost is:

$$\begin{aligned}
\sum_{i=1}^{m} t_i &= \sum_{i=1}^{m} (a_i + \phi(d_{i-1}) - \phi(d_i)) \\
&= \sum_{i=1}^{m} a_i + \sum_{i=1}^{m} (\phi(d_{i-1}) - \phi(d_i)) \quad \text{(telescoping summation)} \\
&= \sum_{i=1}^{m} a_i + \phi(d_0) - \phi(d_m)
\end{aligned}$$

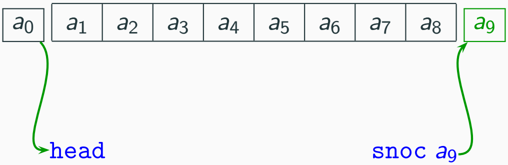
If the initial potential is zero, $\phi(d_0) = 0$, then $\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} a_i - \phi(d_m)$
and the actual cost is smaller than the amortized cost: $\sum_{i=1}^{m} a_i \geq \sum_{i=1}^{m} t_i$

FIFO Queues

A simple example of use of amortized analysis is the data structure of First In First Our (FIFO) Queues

Lists of elements of some type A with operations:

- Insert a new element at the end (snoc)
- Get an element from the front (head)



(The word `snoc` is the inverse of `cons`, which is the usual operation to add an element in front of a list)

Queue Type

The data type `Queue` is required to have the following methods:

Queue Type

The data type `Queue` is required to have the following methods:

- `empty :: Queue`
  The queue with not elements

Queue Type

The data type `Queue` is required to have the following methods:

- `empty :: Queue`
  The queue with not elements
- `isEmpty :: Queue -> Bool`
  Test for emptiness

Queue Type

The data type `Queue` is required to have the following methods:

- `empty :: Queue`
  The queue with not elements
- `isEmpty :: Queue -> Bool`
  Test for emptiness
- `snoc :: Queue -> A -> Queue`
  Adds an element at the end of the queue

Queue Type

The data type `Queue` is required to have the following methods:

- `empty :: Queue`
  The queue with not elements
- `isEmpty :: Queue -> Bool`
  Test for emptiness
- `snoc :: Queue -> A -> Queue`
  Adds an element at the end of the queue
- `head :: Queue -> A`
  Read the first element of the queue

Queue Type

The data type `Queue` is required to have the following methods:

- `empty :: Queue`
  The queue with not elements
- `isEmpty :: Queue -> Bool`
  Test for emptiness
- `snoc :: Queue -> A -> Queue`
  Adds an element at the end of the queue
- `head :: Queue -> A`
  Read the first element of the queue
- `extract :: Queue -> (A,Queue)`
  Remove the first element of the queue,
  return it together with the tail

Imperative Implementation

In imperative programming we can realize queues as doubly-linked lists:

This allows us to perform all operations in constant time

Imperative Implementation

In imperative programming we can realize queues as doubly-linked lists:

This allows us to perform all operations in constant time

In functional programming
(or imperative programming if we want to save on pointers),
we can achieve constant amortized cost
by representing a queue as a pair of lists

$$([a_0, a_1], [a_3, a_2])$$

(the second part is inverted)

Functional Implementation

`Queue = ([A], [A])`

A queue is split in two parts:

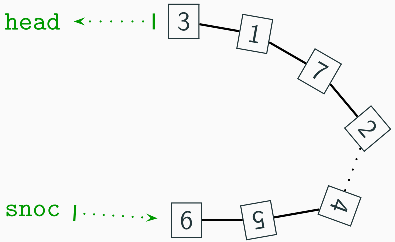
- A front portion and
- A rear portion, which is reversed

The queue

3	1	7	2	4	5	9

can be represented as $([3, 1, 7, 2], [9, 5, 4])$

Imagine that the queue is *bent* to present both ends to the user:



Different Representations

The representation is not unique

The same queue has alternative representations:

$$([3, 1, 7], [9, 5, 4, 2]) \qquad ([3, 1, 7, 2, 4, 5], [9]) \qquad \text{etc.}$$

Different Representations

The representation is not unique

The same queue has alternative representations:

$$([3, 1, 7], [9, 5, 4, 2]) \qquad ([3, 1, 7, 2, 4, 5], [9]) \qquad \text{etc.}$$

It is possible that one of the two portions is empty:

$$([3, 1, 7, 2, 4, 5, 9], []) \qquad ([], [9, 5, 4, 2, 7, 1, 3])$$

Different Representations

The representation is not unique

The same queue has alternative representations:

$$([3,1,7],[9,5,4,2]) \qquad ([3,1,7,2,4,5],[9]) \qquad \text{etc.}$$

It is possible that one of the two portions is empty:

$$([3,1,7,2,4,5,9],[]) \qquad ([],[9,5,4,2,7,1,3])$$

All operations can be executed in constant time

Except `head` and `extract` when the front list is empty

In that case, we must first reverse the rear list and then extract:

$$\begin{array}{cl} ([],[9,5,4,2,7,1,3]) & \\ \downarrow & \text{reverse the rear} \quad O(n) \\ ([3,1,7,2,4,5,9],[]) & \\ \downarrow & \texttt{extract} \\ (3,([1,7,2,4,5,9],[])) & \end{array}$$

Implementation of Insertion and Extraction

```
snoc (f,r) x = (f, x:r)
```

Add the new element at the front of the rear list

Remember that the rear list is inverted

Implementation of Insertion and Extraction

```
snoc (f,r) x = (f, x:r)
```

Add the new element at the front of the rear list

Remember that the rear list is inverted

```
extract (x:f, r) = (x, (f,r))
extract ([],[])  = error "Empty Queue"
extract ([],r)   = extract (reverse r, [])
```

In the last case we can assume that the rear list is not empty, so the recursive call to `extract` will hit the first case

Implementation of Insertion and Extraction

```
snoc (f,r) x = (f, x:r)
```

Add the new element at the front of the rear list
Remember that the rear list is inverted

```
extract (x:f, r) = (x, (f,r))
extract ([],[])  = error "Empty Queue"
extract ([],r)   = extract (reverse r, [])
```

In the last case we can assume that the rear list is not empty, so the recursive call to `extract` will hit the first case

The last case of `extract` has cost $O(n)$ because we must reverse the rear list. But after that, we can extract the next n elements in constant time.

We can show that all operations have constant amortized cost

Potential for Queues

The potential function for queues is the length of rear list:

$$\phi(\texttt{f}, \texttt{r}) = \texttt{length}\ \texttt{r}$$

Potential for Queues

The potential function for queues is the length of rear list:

$$\phi(\texttt{f}, \texttt{r}) = \texttt{length}\,\texttt{r}$$

Analysis of the amortized cost of extract:

- First case: `extract (s:f,r)`

Potential for Queues

The potential function for queues is the length of rear list:

$$\phi\,(\mathtt{f}, \mathtt{r}) = \mathtt{length}\;\mathtt{r}$$

Analysis of the amortized cost of extract:

- First case: `extract (s:f,r)`

$$\begin{array}{rcccccc} a & = & t & + & \phi\,(\mathtt{f}, \mathtt{r}) & - & \phi\,(\mathtt{x} : \mathtt{f}, \mathtt{r}) \\ & & \| & & \| & & \| \\ & & 1 & & \mathtt{length}\;\mathtt{r} & & \mathtt{length}\;\mathtt{r} \\ & & \uparrow & & & & \\ & & \text{just one step to get the head} & & & & \\ & = & 1 & & & & \end{array}$$

Amortized Analysis of `extract`

- Third case: `extract ([],r)`

Amortized Analysis of `extract`

- Third case: `extract ([],r)`

$$
\begin{array}{rclcccccc}
a & = & t & + & \phi(\texttt{tail}(\texttt{reverse}\,\texttt{r}), []) & - & \phi([], \texttt{r}) \\
& & \| & & \| & & \| \\
& & \texttt{length}\,\texttt{r} + 1 & & 0 & & \texttt{length}\,\texttt{r} \\
& & \uparrow & & & & \\
& & \text{invert } \texttt{r} \text{ and take the head} & & & & \\
& = & 1 & & & &
\end{array}
$$

Amortized Analysis of `extract`

- Third case: `extract ([],r)`

$$
\begin{array}{rcl}
a & = & \underset{\substack{\| \\ \texttt{length}\,\texttt{r}+1 \\ \uparrow \\ \text{invert } \texttt{r} \text{ and take the head}}}{t} \quad + \quad \underset{\substack{\| \\ 0}}{\phi(\texttt{tail}(\texttt{reverse}\,\texttt{r}),[])} \quad - \quad \underset{\substack{\| \\ \texttt{length}\,\texttt{r}}}{\phi([],\texttt{r})} \\
 & = & 1
\end{array}
$$

The amortized cost is 1 in all cases of `extract`

Amortized Analysis of `extract`

- Third case: `extract ([],r)`

$$
\begin{array}{rcl}
a & = & \underbrace{t}_{\substack{\texttt{length}\,\texttt{r} + 1 \\ \uparrow \\ \text{invert } \texttt{r} \text{ and take the head}}} + \underbrace{\phi(\texttt{tail}(\texttt{reverse}\,\texttt{r}), [])}_{0} - \underbrace{\phi([], \texttt{r})}_{\texttt{length}\,\texttt{r}} \\
 & = & 1
\end{array}
$$

The amortized cost is 1 in all cases of `extract`

Amortized cost of `snoc`:
It adds one element to the rear list:

- One actual step of computation
- The potential increases by one

So the amortized cost is 2.

Amortized Analysis of `extract`

- Third case: `extract ([],r)`

$$
\begin{array}{rclllll}
a & = & t & + & \phi(\texttt{tail}(\texttt{reverse}\,\texttt{r}),[]) & - & \phi([],\texttt{r}) \\
 & & \| & & \| & & \| \\
 & & \texttt{length}\,\texttt{r}+1 & & 0 & & \texttt{length}\,\texttt{r} \\
 & & \uparrow & & & & \\
 & & \text{invert } \texttt{r} \text{ and take the head} & & & & \\
 & = & 1 & & & &
\end{array}
$$

The amortized cost is 1 in all cases of `extract`

Amortized cost of `snoc`:
It adds one element to the rear list:

- One actual step of computation
- The potential increases by one

So the amortized cost is 2.

All operations have $O(1)$ amortized cost