July 23, 2014

# Amortized Analysis

The concept of amortized runtime analysis is usually first encountered when studying the efficiency of the resizing operation for an array or vector, such as in the `Vector::PushBack` operation.

When the vector capacity is greater than it's official size, a call to `PushBack(t)` is implemented by just incrementing the size variable and copying the new element into the revealed slot:

```
PushBack(T t)
{
  data_[size_] = t;
  ++size_;
}
```

This clearly has constant runtime $\Theta(1)$. But occasionally there is no unused capacity for the vector object, and a resizing operation must be performed:

```
PushBack(T t)
{
  if (size_ == capacity_)
    Resize();
  data_[size_] = t;
  ++size_;
}
Resize()
{
  newCapacity = 2 * capacity_;
  newData = new T [newCapacity];
  for (size_t i = 0; i < capacity_; ++i) newData[i] = data_[i];
  delete [] data_;
  data_ = newData;
  capacity_ = newCapacity;
}
```

which clearly has runtime $\Theta(n)$ ($n =$ `size_`) and a slow one at that due to the memory management calls and the implicit $2n$ calls to the `T` constructor and $n$ calls to the T destructor. Thus the worst case runtime of PushBack is $\Theta(n)$. But, given that most

calls to PushBack have runtime cost proportional to 1 and only the occasional call has cost proportional to $n$, what is the average cost over $m$ PushBack operations?

A similar situation arises in tree iterators, which sometimes need to traverse large portions of the data structure to get to the "next" element. In worst cases, this can be the entire tree, meaning that a particular operation might be as costly as $\Omega(n)$. What is the average cost of a sequence of $m$ tree::iterator operations? Or, as it is often stated, what is the runtime cost of an entire tree traversal?

And hash table Iterators sometimes need to skip over an indeterminate number of empty buckets to find the next non-empty bucket. Again, in certain cases this could be almost all of the buckets, meaning that operation is $\Omega(b+n)$ worst case, where $b =$ number of buckets and $n$ the size of the table. What is the cost of an entire traversal of a Hash Table?

The answers to questions like these are expressed in terms of the "amortized" cost. The high cost of the resizing, skipping, or searching operations is averaged out over all of the other less expensive calls.

THEOREM 1. The runtime of $m$ calls to Vector::PushBack is $\Theta(m)$, provided $m >= n$.

Observing that $\Theta(m)/m = \Theta(1)$, we get:

COROLLARY. The amortized runtime cost of Vector::PushBack is constant.

Recall that a *standard traversal* of a container `c` of type `C` is defined to be the loop

```
for ( C::Iterator i = c.Begin() ; i != c.End() ; ++i )
{
  // some action
}
```

THEOREM 2. The runtime of a standard traversal of a tree with $n$ nodes is $\Theta(n)$.

COROLLARY. The amortized runtime cost of a Tree::Iterator operation is constant.

Assuming HashTable uses chaining to resolve collisions, as in the "vector of lists" implementation, with $b$ buckets and $n$ elements, we have:

THEOREM 3. The runtime of a standard traversal of a HashTable is $\Theta(b+n)$.

COROLLARY. The amortized runtime cost of a HashTable::Iterator operation is $\Theta(1+b/n)$.

Note that $\Theta(1+b/n) = \Theta(1)$ if we have the usual structural assumption that $b = \Theta(n)$.

(An aggregate analysis proof of Theorem 1 is in the lecture notes on vectors, and a similar proof of Theorem 2, for binary trees, is in the lecture notes on trees. Theorem 3 is covered in one of the Exercises at the end of this chapter.)

## 1 Techniques for Amortized Analysis

We look at three techniques for amortized analysis. The general goal is to analyze the total cost of a sequence of $m$ operations. The amortized cost is defined to be the average cost of the sequence of operations, per operation:

$$\text{Amortized cost per operation} = \frac{\text{cost of } m \text{ operations}}{m}$$

Three general approaches to amortized analysis are listed below:

### 1.1 Aggregate Analysis

In aggregate analysis, the total cost of a sequence of $m$ operations is estimated and simplified.

### 1.2 Cost Accounting

In the cost accounting method we assign an "amortized" cost along with the intrinsicly defined actual cost:

$$\begin{aligned}
c_i &= \text{actual cost of the ith operation} \\
a_i &= \text{amortized cost of the ith operation} \\
a_i - c_i &= \text{credit amount for the ith operation} \\
c = \sum_{i=0}^{m} c_i &= \text{total cost} \\
a = \sum_{i=0}^{m} a_i &= \text{total amortized cost}
\end{aligned}$$

and subject to the constraint that the total credit is non-negative:

$$a - c = \sum_{i=0}^{m} a_i - \sum_{i=0}^{m} c_i \geq 0$$

Note that given the above, the total cost is bounded by the total amortized cost:

$$\text{total cost} = c \leq a$$

and, hence, cost $\leq O(a)$.

While the cost $c_i$ is inherent to the algorithm, the amortized cost $a_i$ is something we can define for convenience of proof, subject only to the constraint that $c \leq a$.

### 1.3* Potential Functions

For the potential function method it is helpful to think of the evolving data structure as a sequence $D_0, D_1, D_2, \ldots$ as transformed by the sequence of operations, beginning with $D_0$ before the first operation. Define

$$c_i = \text{actual cost of the operation transforming } D_{i-1} \text{ to } D_i$$

We need to define a *potential function* $\Phi$ that maps the instances of the data structure to the real numbers such that:

$$\Phi(D_i) >= \Phi(D_0) \quad \text{for each } i$$

Then the amortized cost can be defined as

$$a_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

Note that

$$\begin{aligned}
\sum_{i=1}^{m} a_i &= \sum_{i=1}^{m} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\
&= \sum_{i=1}^{m} c_i + \sum_{i=1}^{m} (\Phi(D_i) - \Phi(D_{i-1})) \\
&= \sum_{i=1}^{m} c_i + \Phi(D_m) - \Phi(D_0)
\end{aligned}$$

That is, the total amortized cost is the total cost plus the difference in beginning and ending potential. Since $\Phi(D_m) >= \Phi(D_0)$, the total amortized cost is an upper bound on the total cost.

*More advanced topic.

Thus in the potential function method we are free to define a potential function $\Phi$, subject only to the constraint that the potential $\Phi(D_i)$ is not smaller than the beginning potential $\Phi(D_0)$, and conclude that the runtime of a sequence of $m$ operations is bounded by the total amortized cost.

## 2 Illustratative Problems

We illustrate the three amortized analysis methods each with two problems:

### 2.1 MultiStack

A MultiStack is an ordinary stack contrived with an additional operation `MultiPop(i)` that pops the stack $i$ times (or $n$ = size times, if $i > n$):

```
MultiPop(i)
{
  while (i > 0 && !s.Empty())
  {
    s.Pop();
    --i;
  }
}
```

### 2.2 Binary counter increment

Incrementing a binary counter is an interesting example to analyze, because (1) it is an inherent part of most loops and (2) calls to increment have data-dependent cost that can range from 1 to $k$, where $k$ is the number of bits. Here is C++ pseudocode for incrementing a counter at the bit level:

```
BinaryCounter::operator ++()
{
  i = 1; // location of bit
  while ( (not out of bits) && (bit i is 1) )
  {
    unset bit i;
    shift to next bit;
  }
  if (bit i is 0)
    set bit i;
}
```

The operator++ algorithm can be analyzed in terms of the three primitive operations `unset`, `set` and `shift`. Note that these three operations have hardware support (see Appendix: Code) and run in constant hardware cost. Note also that, in the algorithm above, `shift` is paired with `unset` in every case, so we can estimate runtime cost by counting only `set` and `unset` operations.

An instrumented implementation for this algorithm is available for experimentation in `[LIB]/area51/binarycounter.x`.

## 3 Aggregate Analysis

The two algorithms developed in the previous section serve to illustrate the use of the three methods for amortized analysis. We begin with aggregate analysis.

### 3.1 Aggregate Analysis of MultiStack

The worst case runtime [WCRT(1)] for one MultiPop operation is s.Size(). It follows that for $m$ MultiStack ops, starting with an empty stack, the worst case runtime is

$$\begin{aligned}\text{WCRT}(m) &<= (\text{number of ops}) \times (\text{worst case cost of any op}) \\ &= m \times m = m^2\end{aligned}$$

because the size can be $m$. This is a correct bound, but it is not tight. We can do better!

Observe that Pop (including those in MultiPop) can be called at most $m$ times, because Push can be called at most $m$ times, so the cost of all MultiPop operations in a set of m operations is $\leq m$.[1] And the cost of all the other operations is the number of them (since each is constant cost) and hence is $\leq m$. Therefore the total cost of $m$ MultiStack operations is $\leq m + m$, and we have:

$$\text{WCRT}(m) <= O(m)$$

Therefore the cost of a single multistack operation, amortized over $m$ operations, is $O(m)/m = O(1) = \Theta(1)$.

### 3.2 Aggregate Analysis of binary counter increment

What is the cost of operator++ ? We have already observed that `set` and `unset` have constant runtime. Therefore the runtime cost of operator ++ is the number of `set` operations plus the number of `unset` operations. (We can ignore the cost of `shift` since it is tied with `unset`.) Either of these two operations we call a "bit flip".

[1] Here is where we "aggregate" the number of Pop operations.

First observe that the maximum length of the loop in operator++ is $k$ = the number of bits in the counter. Therefore

$$\text{WCRT} = \text{length of loop} = \text{number of bits in counter} = k$$

This yields an estimate of $O(km)$ for a sequence of $m$ increments. But we can do better. Counting bits from least to most significant, starting with 0, observe:

bit 0 flips each time ++ is called (either set or unset)

bit 1 flips every other time ++ is called

bit 2 flips every $4^{th}$ time

...

bit i flips one out of every $2^i$ times operator ++ is called

...

In a sequence of $m$ increments, bit $i$ is flipped $m/(2^i)$ times (assuming $m < k$) so the total number of flips is

$$\begin{aligned}\sum_{i=0}^{k} \frac{m}{2^i} &= m \sum_{i=0}^{k} \frac{1}{2^i} \\ &\leq m \sum_{i \geq 0} \frac{1}{2^i} \\ &= m \times 2 \\ &= 2m\end{aligned}$$

Therefore the cost of all $m$ calls to operator++ is $\leq O(m)$ and the amortized cost of one call is $\leq O(m)/m = \Theta(1)$.[2]

## 4 Cost Accounting Analysis

Again we treat each of the two example algorithms.

### 4.1 Cost Accounting Analysis of MultiStack

Define the following cost `c` and amortized cost `a` for each of the three MultiStack operations:

```
op           c             a
--           --            --
Push         1             2 // leave extra 1 credit with item
Pop          1             0
MultiPop(i)  min(i,size)   0
```

[2]This argument is very similar to that in the lecture notes for analysis of Vector::PushBack.

As we Push items on the stack, leave the 1 credit with the pushed item. Then every item has enough credit to pay for the Pop operation that removes it from the stack (including any Pop that is part of MultiPop). Therefore the amortized cost of Push pays for all of the Pop and MultiPop operations:

$$\text{Total cost} \leq \text{total amortized cost} = 2m$$

and amortized cost per operation is $O(2m)/m = \Theta(1)$. Note that our insightful definition of the amortized costs eliminated having to deal with the summations, making the analysis seem "easier". The logic of depositing the credit with the pushed item makes the proof that (cost of any sequence of operations) $\leq$ (total amortized cost of that sequence) more or less self-evident.

### 4.2 Cost Accounting Analysis of Binary Increment

Observe that a bit must be set before it can be unset. So we may as well pay for the subsequent unset operation at the time the bit is set:

```
op        c         a
--        --        --
set       1         2   // leave credit with bit being set
unset     1         0
```

Thus in the loop defining operator++, all of the unset operations are already paid for by the previous set operation that set the bit being unset. But set occurs only once in a call to operator++, outside the loop. Therefore

$$a = \sum_i a_i \geq \sum c_i = c$$

(the necessary condition for accounting analysis) and

$$a = \sum_i a_i = 2m$$

It follows that the runtime cost of $m$ increment operations is $O(m)$ and the amortized cost of one increment is $\Theta(1)$.

## 5* Potential Function Analysis

We continue as above with analysis of the two example algorithms.

### 5.1 Potential Function Analysis of MultiStack

We are given a sequence of $m$ MultiStack operations beginning with an empty stack $S_0$ and transforming to stacks $S_1, S_2, ..., S_m$. Define the *potential* of a stack $S$ as the number of elements in the stack:

$$\Phi(S) = S.\text{size}$$

Then $\Phi(S_0) = 0$ because $S_0$ is empty, and $\Phi(S_i) >= 0$ because no stack has negative size, so the potential requirements are met.

Calculate the amortized cost by operation:

$$\begin{aligned}
\text{Push:} \quad & a_i = c_i + \Phi(S_i) - \Phi(S_{i-1}) = 1 + S_i.\text{size} - S_{i-1}.\text{size} = 1 + 1 = 2 \\
\text{Pop:} \quad & a_i = c_i + \Phi(S_i) - \Phi(S_{i-1}) = 1 + S_i.\text{size} - S_{i-1}.\text{size} = 1 - 1 = 0 \\
\text{MultiPop(k):} \quad & a_i = c_i + \Phi(S_i) - \Phi(S_{i-1}) = k' + S_i.\text{size} - S_{i-1}.\text{size} = k' - k' = 0
\end{aligned}$$

where $k' = \min(k, S_{i-1}.\text{size})$. Thus (the total cost of $m$ MultiStack operations) $\leq$ (the total amortized cost) $\leq 2m$, and the amortized cost per operation is $\Theta(1)$.

### 5.2 Potential Function Analysis of BinaryIncrement

We are given a sequence of $m$ Increment operations on a binary counter $C_0$ initialized to 0 and transforming to counters $C_1, C_2, ..., C_m$. Define the potential of a counter as the number of 1 bits in its binary representation:

$$\Phi(C) = \text{number of 1 bits in } C.\text{word}$$

Clearly $\Phi(C) >= 0$ for any counter $C$, and $\Phi(C_0) = 0$, so the potential conditions are met.

For convenience, denote $\Phi(C_i)$ by $b_i$ = number of 1 bits in $C_i$.word. Suppose that the $i^{th}$ application of operator++ unsets $t_i$ bits before setting a bit. (The loop body executes $t_i$ times.) Examine these values by cases:

*More advanced topic.

Case: $b_i = 0$
Then the increment getting us to $C_i$ unset all bits and did no set:

$$b_{i-1} = t_i = k$$
$$c_i = t_i$$

Case: $b_i \neq 0$

$$b_i = b_{i-1} - t_i + 1$$
$$c_i = t_i + 1$$

Therefore in all cases

$$b_i <= b_{i-1} - t_i + 1 \quad \text{and}$$
$$b_i - b_{i-1} <= -t_i + 1$$

Calculating the amortized cost of operation i, we get:

$$\begin{aligned} a_i &= c_i + \Phi(C_i) - \Phi(C_{i-1}) \\ &\leq (t_i + 1) + b_i - b_{i-1} \\ &\leq t_i + 1 - t_i + 1 \\ &= 2 \end{aligned}$$

and therefore the cost of $m$ increments is $O(m)$ and the amortized cost of one increment is constant.

## Exercises

1. Construct a proof of Theorem 1 using the Cost Accounting method.
2. Construct a proof of Theorem 3 using the Potential Function method.

## Appendix: Code

```
// binary counter class
template < typename N  = size_t >
class BinaryCounter
{
public:
  BinaryCounter(N begin = 0) : word_ (begin)
  {}
  BinaryCounter& operator ++()
  {
    N i = 0x01;      // mask determining location of bit
    while ((i != 0) && ((word_ & i) != 0))
    {
      word_ &= ~i;   // unset bit i
      i = i << 1;    // shift to next bit
    }
    if (i != 0)      // word_ & i == 0
    {
      word_ |= i;    // set bit i
    }
   return *this;
  }

  N Value () const { return word_; }

private:
  N word_; //  bits = 8 * sizeof(N)
};

// example binary counter client program
int main(int argc, char* argv[])
{
  size_t start = 0, stop = 1025;
  if (argc > 1)
    start = atoi(argv[1]);
  if (argc > 2)
    stop = atoi(argv[2]);

  for (BinaryCounter<> bc(start); bc.Value() < stop; ++bc)
    std::cout << ''Counter value: '' << bc.Value() << '\n';
}
```