Amortized Analysis

Victor Adamchik

Amortized analysis gives the average performance (over time) of each operation in the worst case. In a sequence of operations the worst case does not occur often in each operation - some operations may be cheap, some may be expensive Therefore, a traditional worst-case per operation analysis can give overly pessimistic bound. For example, in a dynamic array only some inserts take a linear time, though others - a constant time. When different inserts take different times, how can we accurately calculate the total time? The amortized approach is going to assign an "artificial cost" to each operation in the sequence, called the amortized cost of an operation. It requires that the total real cost of the sequence should be bounded by the total of the amortized costs of all the operations. Note, there is sometimes flexibility in the assignment of amortized costs.

Three methods are used in amortized analysis

1. Aggregate Method (or brute force)
2. Accounting Method (or the banker's method)
3. Potential Method (or the physicist's method)

- **Table Resizing**

In data structures like dynamic array, heap, stack, hashtable (to name a few) we used the idea of resizing the underlying array, when the current array was either full or reached a chosen threshold. During resizing we allocate an array twice as large and move the data over to the new array. So, if an array is full, the cost of insertion is linear; if an array is not full, insertion takes a constant time. To derive an amortized cost we start with an array of size 1 and perform *n* insertions.

insert	old size	new size	copy
1	1	–	–
2	1	2	1
3	2	4	2
4	4	–	–
5	4	8	4
6	8	–	–
7	8	–	–
8	8	–	–
9	8	16	8

The table shows that 9 inserts require 1 + 2 + 4 + 8 copy operations. In general $2^n + 1$ inserts require

$$1 + 2 + 2^2 + 2^3 + ... + 2^n = 2^{n+1} - 1$$

copies. Thus, the total cost (copying and insertion) is given by

$$(2^{n+1} - 1) + (2^n + 1) = 3\ 2^n$$

Then the average cost of $2^n + 1$ inserts is

$$\lim_{n\to\infty} \frac{3\ 2^n}{2^n + 1} = 3$$

and we can say that the amortized cost of insertion is constant. Such method of analysis is called an aggregate method. In this method, we show that a sequence of n operations (in our example, insertions) takes $T(n)$ time, and then we say that the amortized cost per insertion is $T(n) / n$.

Second analysis: the accounting method We will assign a dollar token to each operation. If we assign only one token for insertion we will go bust - there will be no tokens to pay for future copying. Assigning two tokens seems enough, but this will work only for one insert and one copy. Since an array can be resized several times, we have to make sure we have enough tokens for all resizings. Therefore, we assign three tokens for each insert. To understand this, imagine an array which is currently full. First half of this array has no any tokens available (we paid $1 for insert and $2 for previous doublings). So, we have no tokens to pay for the next resizing. Fortunately, the second half has enough tokens (actually $2) to bailout those guys

0 0 $2 $2 before doubling

0 0 0 0 – – – – after doubling

We conclude that if we allocate $3 for each operation, we'll never run out of money. Thus the amortized cost of an operation is 3. It is very important in this analysis to make sure that the bank account never go negative!

- **FIFO Queue with two stacks**

We can implement a FIFO queue using two stacks as follows:

- enqueue(x): push x onto stack1.
- dequeue(): if stack2 is not empty, pop it.
  otherwise, we POP the entire contents of stack1 and PUSH it into stack2. Next simply POP from stack2 and return the result.

Clearly, enqueue() has $O(1)$ the worst-case runtime complexity, and dequeue() - $O(n)$. We will show the amortized cost of enqueue is 3 and dequeue is 1.

The aggregate method. If an element is processed, it is pushed at most twice and popped at most twice. However, if an element is not dequeued, it's only pushed twice and popped once. So the amortized cost is 3 for each enqueue. The amortized cost of the dequeue is 1.

The accounting method. When we enqueue, we assign three tokens to it. We use one to push the element onto stack1, and the other two are for future use - one to pop it from stack1 and the second to push it onto stack2.

- **Potential method**

This is similar to the accounting method, except that we consider the bank account as a potential energy. Consider a sequence of n operations. Each operation changes the current state of data structure. Let us denote states by $s_0,\ s_1,\ ...,\ s_n$. Next, we define a potential function

$$\Phi : s_k \to \mathbb{R}$$

such that

$$\Phi(s_0) = 0$$

$$\Phi(s_k) \geq 0,\ k > 0$$

Let the cost of $k-$th operation is c_k. Then, the amortized cost m_k is the actual cost plus a change in the potential $\Delta\Phi(s_k)$

$$m_k = c_k + \Delta\Phi(s_k) = c_k + \Phi(s_k) - \Phi(s_{k-1})$$

Note that a change in the potential could be either positive (stores potential for future use) or negative (pays for $k-$th operation). The amortized cost is the upper bound for the true cost. Indeed, sum up the previous formula wrt k :

$$\sum_{k=1}^{n} m_k = \sum_{k=1}^{n} c_k + \sum_{k=1}^{n} \Delta\Phi(s_k) = \sum_{k=1}^{n} c_k + \Phi(s_n) - \Phi(s_0) = \sum_{k=1}^{n} c_k + \Phi(s_n) \geq \sum_{k=1}^{n} c_k$$

The key to amortized analysis is to define the right potential function. As with the accounting approach, the potential function must save enough for future expenses. On the other hand, it cannot save much more, because this will cause the amortized cost to be too high.

Let us do this analysis on dynamic array. We define the potential function in the following way

$$\Phi(s_k) = 2k - \text{current\_array\_size} \geq 0$$

To get a feeling of this function, choose *current_array_size* to be 8. Then

$$\Phi(s_4) = 0,\ \Phi(s_5) = 2,\ \Phi(s_6) = 4,\ \Phi(s_7) = 6,\ \Phi(s_8) = 8$$

Consider two cases.

Case 1: $k \leq$ current_array _size. In this case $\Delta\Phi(s_k)$ changes by 2. The actual cost $c_k = 1$ (one insertion). So the amortized cost is

$$m_k = c_k + \Delta\Phi(s_k) = 1 + \Delta\Phi(s_k) = 1 + 2 = 3$$

Case 2: $k = 1 +$ current_array _size. In this case our table is resized. The actual cost is one insertion plus copy:

$$c_k = 1 + \text{current\_array\_size}$$

The change in potential is

$$\Delta\Phi(s_k) = \Phi(s_k) - \Phi(s_{k-1}) = 2 - \text{current\_array\_size}$$

since

$$\Phi(s_k) = 2k - 2 * \text{current\_array\_size}$$

$$\Phi(s_{k-1}) = 2k - 2 - \text{current\_array\_size}$$

So the amortized cost is

$$m_k = c_k + \Delta\Phi(s_k) = 3$$

- **FIFO Queue with two stacks**

The potential method. Let the potential Φ be defined by

$$\Phi(s_k) = 2 * \text{size\_stack1}$$

Enqueue:

$$m_k = c_k + \Delta\Phi(s_k) = 1 + \Delta\Phi(s_k) = 1 + 2 = 3$$

Dequeue:

Case1: if stack2 is not empty

$$m_k = c_k + \Delta\Phi(s_k) = 1 + \Delta\Phi(s_k) = 1 + 0 = 1$$

Case 2: otherwise, we POP the entire contents of stack1 and PUSH it into stack2. Next simply POP from stack2 and return

$$m_k =$$
$$(s_k) = 2 * \text{size\_stack1} + 1 - 2\,\text{size\_stack1} = 1$$

- **Binary counter**

Given binary numbers from 0 to n. If the cost of incrementing a binary number is the number of bits flipped, what is the amortized cost per increment? Clearly, the worst-case cost per increment is $O(\log n)$ – all bits are flipped.

```
...0000
...0001
...0010
...0011
...0100
...0101
```

The aggregate method. Lets us start with the least significant bit. Each time we increment a number, that bit is changed. Thus, the number of times this bit changes is n. Look at the next significant bit. it changes $n/2$ times, the next one - $n/4$, and so on. Thus the total cost is

$$n + \frac{n}{2} + \frac{n}{4} + ... = n \sum_{k=0}^{\log_2 n} \frac{1}{2^k} = n\left(2 - \frac{1}{n}\right)$$

It follows that the amortized cost per increment is 2.

The potential method. Let the potential Φ be the number of 1's in the current number. Consider the $k-$th increment that changes the number from $k-1$ to k. Let the number of bits that change from 1 to 0 is j. Note, that though different increments can have different numbers of $1 \to 0$ flips, each increment has exactly one $0 \to 1$ flip. The change in potential is $1 - j$. The actual cost is one increment is $1 + j$. Thus,

$$m_k = c_k + \Delta\Phi(s_k) = 1 + j + (1 - j) = 2$$

■ Binomial Heaps

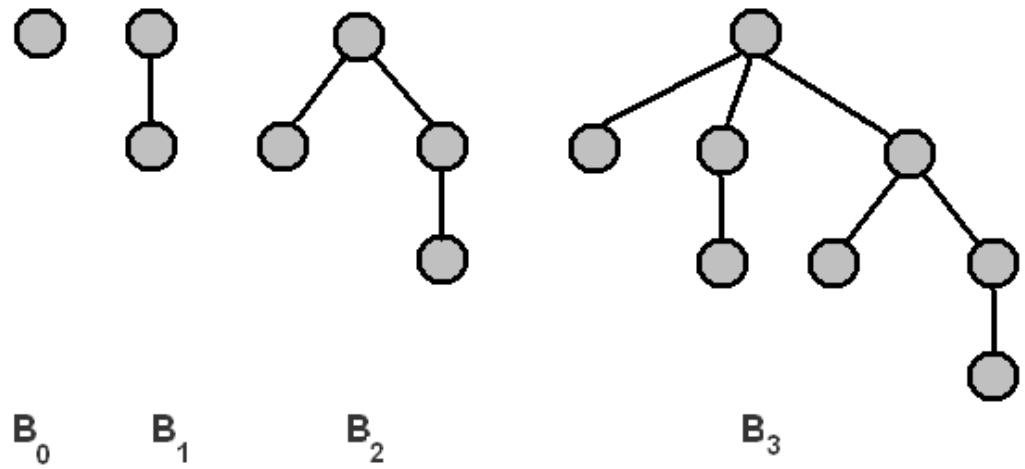
We describe a different kind of heap that has a slight improvement over the binary heap. This data structure was introduced by Vuillemin in 1978.

	findMin	deleteMin	insert	decreaseKey	merge
binary heap	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
binomial heap	$O(1)$	$O(\log n)$	$O(\log n)$	–	$O(\log n)$

First we define a binomial tree.

The binomial tree of rank k, B_k, is defined recursively as follows

1. B_0 is a single node
2. B_k is formed by joining two B_{k-1} trees



Simple properties of binomial trees:

1. B_k has 2^k nodes
2. The root of B_k has k children
3. These children are again binomial trees of ranks $0,\ 1,\ \ldots,\ k-1$
4. The number of nodes at level d from the root in B_k is $\binom{k}{d}$, where $0 \le d \le k$.

The last property readily follows from the binomial theorem:

$$2^n = (1+1)^n = \sum_{j=0}^{n} \binom{n}{j}$$

In order to store n nodes it is NOT sufficient to have just one binomial tree, but a set of them.

Since each B_k has 2^k nodes and any number can be written in binary, we need at most $\log n$ (**why -?**) binary trees to store n nodes

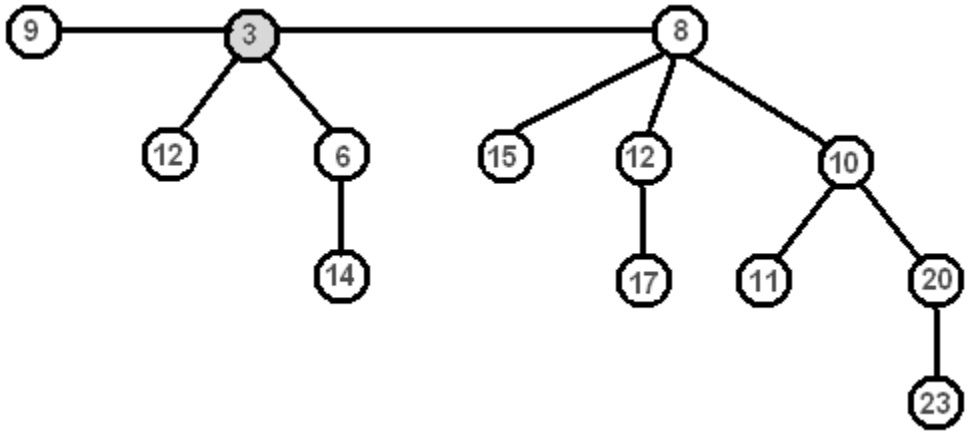
$6_{10} = 110_2 \quad \rightarrow$ use $B_2,\ B_1$

$11_{10} = 1011_2 \quad \rightarrow$ use $B_3,\ B_1,\ B_0$

$18_{10} = 10010_2 \quad \rightarrow$ use $B_4,\ B_1$

Definition.

A binomial heap is a collection of binomial trees in increasing order of size where each tree has a heap property. There is at most one binomial tree of any given rank.



To complete the definition we organize all roots of binomial trees in a doubly linked list. The length of that list is $O(\log n)$! Finally, we add an additional reference to the minimum element in the heap.

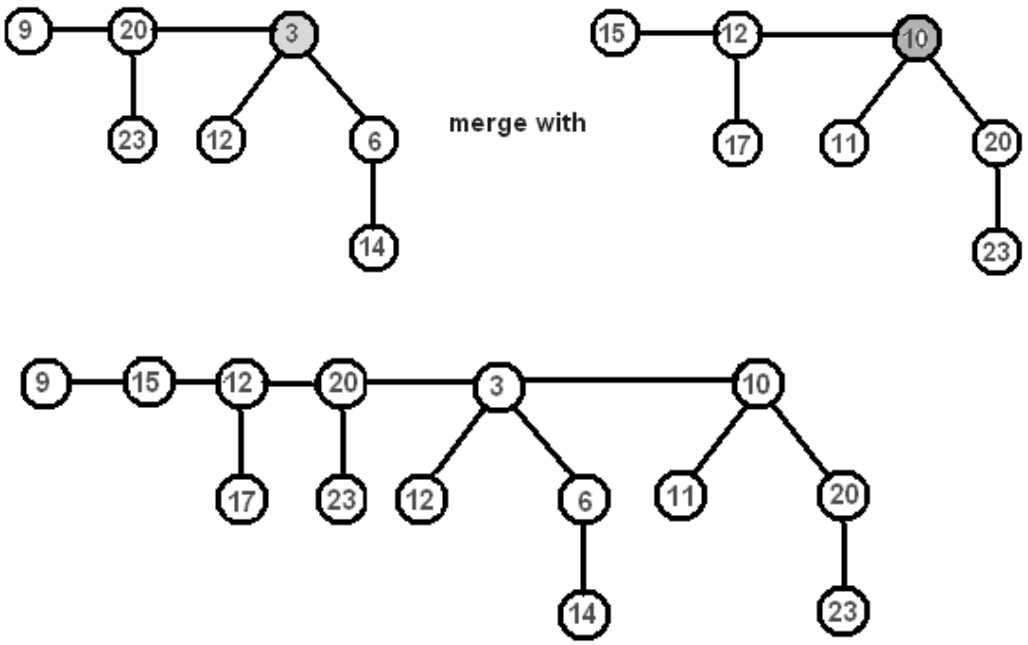
- **Finding the minimum**

  Clearly, this requires returning an appropriate reference, so it takes $O(1)$.

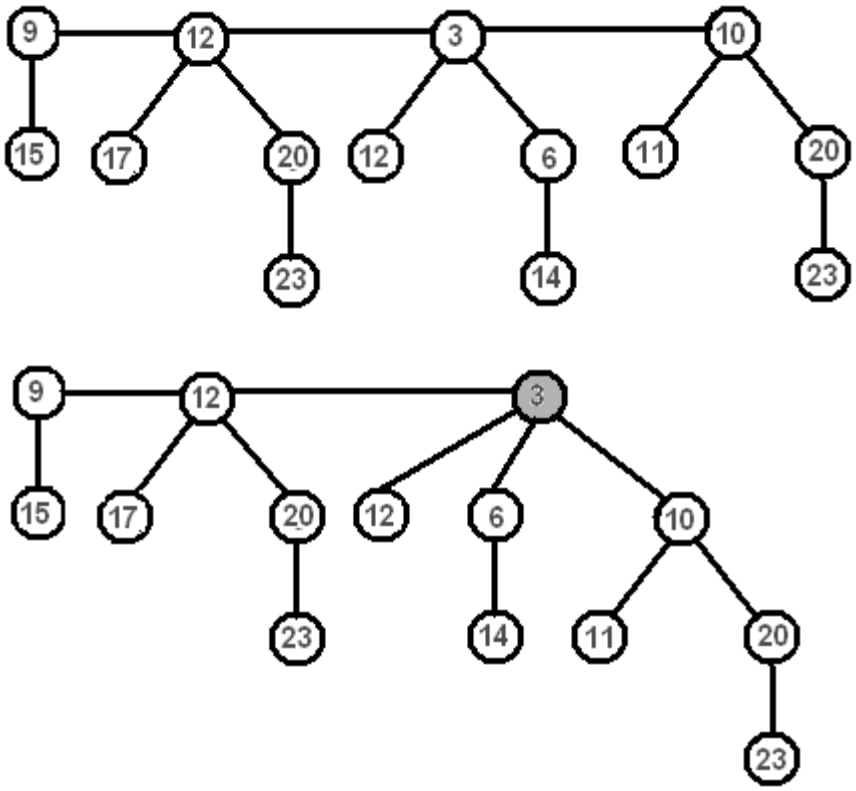
- **Merging two heaps**

  Merging heaps is based on recursive merging binomial trees of the same rank.

  The procedure has two phases. The first phase merges the roots of binomial heaps H_1 and H_2 into one linked list H that is sorted by degree in monotonically increasing order.



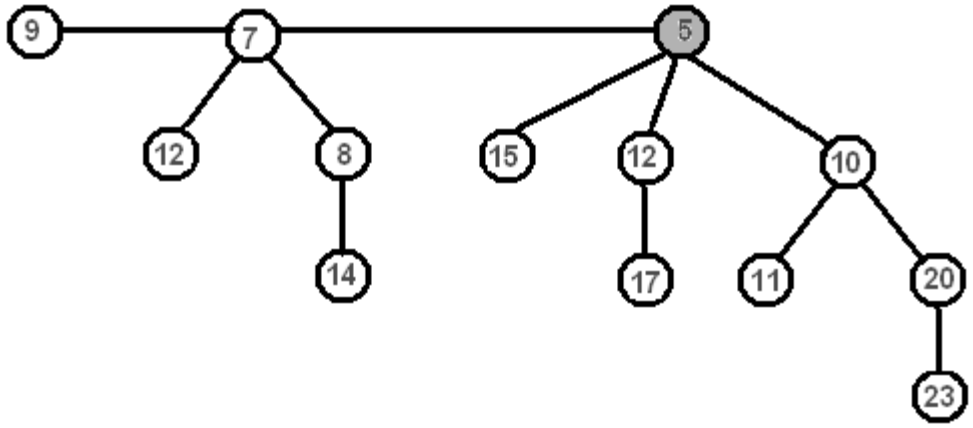
The second phase links roots of equal rank binomial trees until at most one root remains of each rank.



Question. What is the complexity of merging?

- **Deleting the Minimum**

  Given the binomial tree



The procedure of deleting the minimum has the following three steps

1. Remove the tree, say B, that has the min element, creating a new heap H_1
2. Remove the root of B and join the binomial trees (the subtrees of this root) into a heap H_2
3. Merge H_1 with H_2

It takes $O(\log n)$ in the worst case.

- **Insertion**

  Essentualy this is a union of two binomial heaps, therefore it takes $O(\log n)$ in the worst case.

- **Amortized cost of insertion**

  Each insert operation creates a single binomial tree, this costs us $1, in addition we have to store another $1 in the bank. This saving will be used for merging binomial trees. Thus, in the heap, each binomial tree has $1 credit. When we merge two trees, we will pay with the credit associated with the root of the subordinate tree. The insert operation might require a cascade of merging, but since each tree has a $1 credit, we have enough money to pay for this job.

  We define the potential function in the following way

  $$\Phi(s_k) = \text{number\_of \_trees} \geq 0$$

  The amortized cost m_k is the actual cost plus a change in the potential $\Delta\Phi(s_k)$

  $$m_k = c_k + \Delta\Phi(s_k)$$

  The actual cost of the insert is the number of trees merged plus 1. A change in potential - the number of trees merged minus 1. Thus

  $$m_k = 2$$