8 Amortized Analysis

Amortization is an analysis technique that can influence the design of algorithms in a profound way. Later in this course, we will encounter data structures that owe their very existence to the insight gained in performance due to amortized analysis.

Binary counting. We illustrate the idea of amortization by analyzing the cost of counting in binary. Think of an integer as a linear array of bits, $n = \sum_{i \geq 0} A[i] \cdot 2^i$. The following loop keeps incrementing the integer stored in A.

```
loop $i = 0$;
  while $A[i] = 1$ do $A[i] = 0$; $i$++ endwhile;
  $A[i] = 1$.
forever.
```

We define the *cost* of counting as the total number of bit changes that are needed to increment the number one by one. What is the cost to count from 0 to n? Figure 28 shows that counting from 0 to 15 requires 26 bit changes. Since n takes only $1 + \lfloor \log_2 n \rfloor$ bits or positions in A,

5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$\boxed{0}$
3	0	0	0	0	0	0	0	$\boxed{0}$	1	1	1	1	1	1	1	$\boxed{1}$
2	0	0	0	$\boxed{0}$	1	1	1	$\boxed{1}$	0	0	0	$\boxed{0}$	1	1	1	$\boxed{1}$
1	0	$\boxed{0}$	1	$\boxed{1}$	0	$\boxed{0}$	1	$\boxed{1}$	0	$\boxed{0}$	1	$\boxed{1}$	0	$\boxed{0}$	1	$\boxed{1}$
0	$\boxed{0}$	$\boxed{1}$	$\boxed{0}$	$\boxed{1}$	$\boxed{0}$	$\boxed{1}$	$\boxed{0}$	$\boxed{1}$	$\boxed{0}$	$\boxed{1}$	$\boxed{0}$	$\boxed{1}$	$\boxed{0}$	$\boxed{1}$	$\boxed{0}$	$\boxed{1}$

Figure 28: The numbers are written vertically from top to bottom. The boxed bits change when the number is incremented.

a single increment does at most $2 + \log_2 n$ steps. This implies that the cost of counting from 0 to n is at most $n \log_2 n + 2n$. Even though the upper bound of $2 + \log_2 n$ is almost tight for the worst single step, we can show that the total cost is much less than n times that. We do this with two slightly different amortization methods referred to as aggregation and accounting.

Aggregation. The aggregation method takes a global view of the problem. The pattern in Figure 28 suggests we define b_i equal to the number of 1s and t_i equal to the number of trailing 1s in the binary notation of i. Every other number has no trailing 1, every other number of the remaining ones has one trailing 1, etc. Assuming $n = 2^k - 1$, we therefore have exactly $j - 1$ trailing 1s for $2^{k-j} = (n+1)/2^j$ integers between 0 and $n - 1$. The total number of bit changes is therefore

$$T(n) \;=\; \sum_{i=0}^{n-1} (t_i + 1) \;=\; (n+1) \cdot \sum_{j=1}^{k} \frac{j}{2^j}.$$

We use index transformation to show that the sum on the right is less than 2:

$$\begin{aligned} \sum_{j \geq 1} \frac{j}{2^j} &= \sum_{j \geq 1} \frac{j-1}{2^{j-1}} \\ &= 2 \cdot \sum_{j \geq 1} \frac{j}{2^j} - \sum_{j \geq 1} \frac{1}{2^{j-1}} \\ &= 2. \end{aligned}$$

Hence the cost is $T(n) < 2(n+1)$. The *amortized cost* per operation is $\frac{T(n)}{n}$, which is about 2.

Accounting. The idea of the accounting method is to charge each operation what we think its amortized cost is. If the amortized cost exceeds the actual cost, then the surplus remains as a credit associated with the data structure. If the amortized cost is less than the actual cost, the accumulated credit is used to pay for the cost overflow. Define the amortized cost of a bit change $0 \rightarrow 1$ as \$2 and that of $1 \rightarrow 0$ as \$0. When we change 0 to 1 we pay \$1 for the actual expense and \$1 stays with the bit, which is now 1. This \$1 pays for the (later) cost of changing the 1 to 0. Each increment has amortized cost \$2, and together with the money in the system, this is enough to pay for all the bit changes. The cost is therefore at most $2n$.

We see how a little trick, like making the $0 \rightarrow 1$ changes pay for the $1 \rightarrow 0$ changes, leads to a very simple analysis that is even more accurate than the one obtained by aggregation.

Potential functions. We can further formalize the amortized analysis by using a potential function. The idea is similar to accounting, except there is no explicit credit saved anywhere. The accumulated credit is an expression of the well-being or potential of the data structure. Let c_i be the actual cost of the i-th operation and D_i the data structure after the i-th operation. Let $\Phi_i = \Phi(D_i)$ be the potential of D_i, which is some numerical value depending on the concrete application. Then we define $a_i = c_i + \Phi_i - \Phi_{i-1}$ as the *amortized cost* of the i-th operation. The sum of amortized costs of n operations is

$$\begin{aligned}\sum_{i=1}^{n} a_i &= \sum_{i=1}^{n} (c_i + \Phi_i - \Phi_{i-1}) \\ &= \sum_{i=1}^{n} c_i + \Phi_n - \Phi_0.\end{aligned}$$

We aim at choosing the potential such that $\Phi_0 = 0$ and $\Phi_n \geq 0$ because then we get $\sum a_i \geq \sum c_i$. In words, the sum of amortized costs covers the sum of actual costs. To apply the method to binary counting we define the potential equal to the number of 1s in the binary notation, $\Phi_i = b_i$. It follows that

$$\begin{aligned}\Phi_i - \Phi_{i-1} &= b_i - b_{i-1} \\ &= (b_{i-1} - t_{i-1} + 1) - b_{i-1} \\ &= 1 - t_{i-1}.\end{aligned}$$

The actual cost of the i-th operation is $c_i = 1 + t_{i-1}$, and the amortized cost is $a_i = c_i + \Phi_i - \Phi_{i-1} = 2$. We have $\Phi_0 = 0$ and $\Phi_n \geq 0$ as desired, and therefore $\sum c_i \leq \sum a_i = 2n$, which is consistent with the analysis of binary counting with the aggregation and the accounting methods.

2-3-4 trees. As a more complicated application of amortization we consider 2-3-4 trees and the cost of restructuring them under insertions and deletions. We have seen 2-3-4 trees earlier when we talked about red-black trees. A set of keys is stored in sorted order in the internal nodes of a 2-3-4 tree, which is characterized by the following rules:

(1) each internal node has $2 \leq d \leq 4$ children and stores $d - 1$ keys;

(2) all leaves have the same depth.

As for binary trees, being sorted means that the left-to-right order of the keys is sorted. The only meaningful definition of this ordering is the ordered sequence of the first subtree followed by the first key stored in the root followed by the ordered sequence of the second subtree followed by the second key, etc.

To insert a new key, we attach a new leaf and add the key to the parent ν of that leaf. All is fine unless ν overflows because it now has five children. If it does, we repair the violation of Rule (1) by climbing the tree one node at a time. We call an internal node *non-saturated* if it has fewer than four children.

`Case 1.` ν has five children and a non-saturated sibling to its left or right. Move one child from ν to that sibling, as in Figure 29.

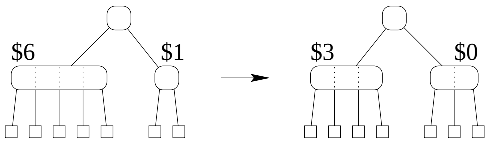

Figure 29: The overflowing node gives one child to a non-saturated sibling.

`Case 2.` ν has five children and no non-saturated sibling. Split ν into two nodes and recurse for the parent of ν, as in Figure 30. If ν has no parent then create a new root whose only children are the two nodes obtained from ν.

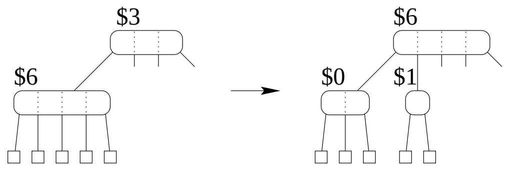

Figure 30: The overflowing node is split into two and the parent is treated recursively.

Deleting a key is done is a similar fashion, although there we have to battle with nodes ν that have too few children rather than too many. Let ν have only one child. We repair Rule (1) by adopting a child from a sibling or by merging ν with a sibling. In the latter case the parent of ν looses a child and needs to be visited recursively. The two operations are illustrated in Figures 31 and 32.

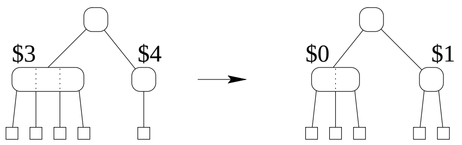

Figure 31: The underflowing node receives one child from a sibling.

Amortized analysis. The worst case for inserting a new key occurs when all internal nodes are saturated. The insertion then triggers logarithmically many splits. Symmetrically, the worst case for a deletion occurs when all

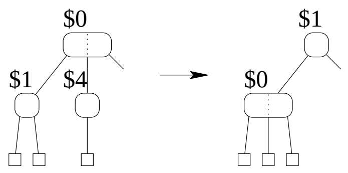

Figure 32: The underflowing node is merged with a sibling and the parent is treated recursively.

internal nodes have only two children. The deletion then triggers logarithmically many mergers. Nevertheless, we can show that in the amortized sense there are at most a constant number of split and merge operations per insertion and deletion.

We use the accounting method and store money in the internal nodes. The best internal nodes have three children because then they are flexible in both directions. They require no money, but all other nodes are given a positive amount to pay for future expenses caused by split and merge operations. Specifically, we store \$4, \$1, \$0, \$3, \$6 in each internal node with 1, 2, 3, 4, 5 children. As illustrated in Figures 29 and 31, an adoption moves money only from ν to its sibling. The operation keeps the total amount the same or decreases it, which is even better. As shown in Figure 30, a split frees up \$5 from ν and spends at most \$3 on the parent. The extra \$2 pay for the split operation. Similarly, a merger frees \$5 from the two affected nodes and spends at most \$3 on the parent. This is illustrated in Figure 32. An insertion makes an initial investment of at most \$3 to pay for creating a new leaf. Similarly, a deletion makes an initial investment of at most \$3 for destroying a leaf. If we charge $\$2$ for each split and each merge operation, the money in the system suffices to cover the expenses. This implies that for n insertions and deletions we get a total of at most $\frac{3n}{2}$ split and merge operations. In other words, the amortized number of split and merge operations is at most $\frac{3}{2}$.

Recall that there is a one-to-one correspondence between 2-3-4 tree and red-black trees. We can thus translate the above update procedure and get an algorithm for red-black trees with an amortized constant restructuring cost per insertion and deletion. We already proved that for red-black trees the number of rotations per insertion and deletion is at most a constant. The above argument implies that also the number of promotions and demotions is at most a constant, although in the amortized and not in the worst-case sense as for the rotations.