Amortized Analysis

The time complexity of an algorithm is an important indication of the efficiency of the algorithm. Often the worst case analysis is used. This, however, is sometimes too pessimistic for practical applications of an algorithm. The average case analysis is more appropriate, but it is often very difficult to calculate. The same is true for the actual time complexity. The *amortized analysis* is an upper bound for the actual complexity and we study it in this task.

Let us assume that we have a data structure D and a collection of operations manipulating D. After every operation the data structure changes its state. Let the initial state be S_0 and after the ith operation the state is S_i. Suppose that we call the operations n times, starting from the state S_0. The amortized time analysis of this sequence of operations starts by associating with each susceptible a state S_i a real number $\Phi(S_i)$, called the *potential of* S_i, chosen so that the more susceptible a state is, the higher its potential. For example, it would be reasonable to let the potential of a stack be the number of entries it contains, because a large stack is susceptible to expensive pops, and a small one is not.

Consider now a sequence of n calls to the data structure D. Let t_i be the actual worst case complexity of the ith call, as usual. Define the amortized complexity a_i of the ith call by

$$a_i = t_i + \Phi(S_i) - \Phi(S_{i-1}),$$

where S_{i-1} is the state of D just before the ith call begins, and S_i is the state of D just after it has finished. Now,

$$\begin{aligned}
\sum_{i=1}^{n} t_i &= \sum_{i=1}^{n} [a_i - \Phi(S_i) + \Phi(S_{i-1})] \\
&= \sum_{i=1}^{n} a_i - \Phi(S_n) + \Phi(S_0) \\
&\leq \sum_{i=1}^{n} a_i
\end{aligned}$$

assuming that $\Phi(S)$ has been chosen so that $\Phi(S_n) \geq \Phi(S_0)$, which is usually the case since S_0 is the initial state and so is likely to have minimal susceptibility to expensive operations. The total amortized complexity is an upper bound on the total actual complexity, and the potentials cancel out.

As an example, consider the traversal of a binary tree in inorder. If we traverse the binary tree in Figure 1 in inorder, then the nodes are visited in order A, B, C, D, E, F, G, H.

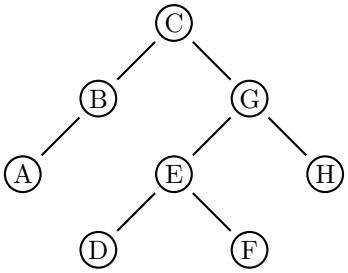

Figure 1: Binary Tree

We consider two operations, `inorder_first` and `inorder_next`. The former operation traverses to the first node of the binary tree in inorder. If the binary tree of the picture is denoted by T, then, when applied to T, `inorder_first` returns the node A. The latter operation proceeds to the next node in inorder. We define the state of the binary tree data structure after the operations `inorder_first` and `inorder_next` to be the original binary tree with one node distinguished. The distinguished node is the node where the search has proceeded to. In the beginning the distinguished node is the root node. For example, after `inorder_first` the distinguished node is A. After this, `inorder_next` returns B which becomes a new distinguished node. Similarly, if the distinguished node is F, then `inorder_next` returns G which becomes a distinguished node.

If there are n nodes in a binary tree, then a single `inorder_next` operation can take $n-1$ steps in the worst case. Based on this, the traversal of a binary tree in inorder takes $\mathcal{O}(n^2)$ steps in the worst case. Obviously this is too pessimistic an estimation and we show next that the amortized analysis gives a better estimation.

The first task is to construct a potential function. We define the potential function to be the *rank* of the distinguished node. The rank is defined as follows. The rank of the root is 0. If the rank of a node is r, then the rank of its left child is $r-1$ and the rank of its right child is $r+1$. For example, the ranks of the tree in Figure 1 are: $r(C) = 0$, $r(B) = -1$, $r(A) = -2$, $r(G) = 1$, $r(E) = 0$, $r(D) = -1$, $r(F) = 1$, $r(H) = 2$. Typically, choosing a potential function is a matter of experience and trial and error.

The rank seems to be the right candidate for a potential function. The more deep on the right the distinguished node is, the more steps, potentially, the `inorder_next` operation must do in order to reach the next node. On the other hand, if the distinguished node is deep on the left, then only one step is needed to reach the next node. So the deeper the search is on the left, the smaller the potential function is. Conversely, the deeper the search is on the right, the larger the potential function is.

Next task is to calculate the values $a_i = t_i + \Phi(S_i) - \Phi(S_{i-1})$. A useful trick for simplifying amortized analyses is to break the operations into small stages and analyse each stage. Consider the ith operation. It has actual complexity t_i, amor-

tized complexity a_i, and it takes the data structure from state S_{i-1} to state S_i. Now suppose the data structure passes through some intermediate states during the operation; call them $S'_1, S'_2, ..., S'_{k-1}$, and let $S_{i-1} = S'_0$ and $Si = S'_k$. For example, the intermediate states with the binary tree operation `inorder_next` are the nodes the search traverses when proceeding to the next state in inorder. Let t'_j be the actual cost incurred in moving from S'_{j-1} to S'_j, for $1 \leq j \leq k$. Clearly, the actual complexity of the whole operation is the sum of these numbers:

$$t_i = \sum_{j=1}^{k} t'_j.$$

The amortized complexity of each stage may be defined in the usual way, with the formula $a'_j = t'_j + \Phi(S'_j) - \Phi(S'_{j-1})$, and this leads to

$$\begin{aligned}
\sum_{j=1}^{k} a'_j &= \sum_{j=1}^{k} [t'_j + \Phi(S'_j) - \Phi(S'_{j-1})] \\
&= \sum_{j=1}^{k} t'_j + \Phi(S'_k) - \Phi(S'_0) \\
&= t_i + \Phi(S_i) - \Phi(S_{i-1}) \\
&= a_i
\end{aligned}$$

The amortized complexity of the whole operation is the sum of the amortized complexities of its stages. We apply this approach to our binary search tree example.

Consider the operation `inorder_next`. We have to check two different situations in Figure 2, where it is necessary to take more than one step in order to reach the next node in inorder.

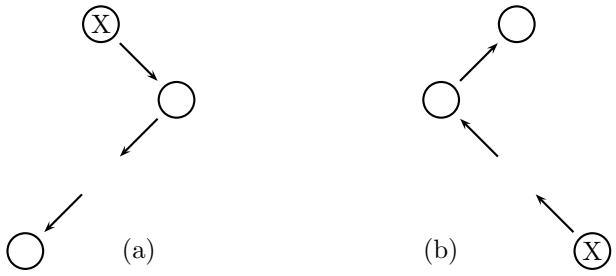

Figure 2: Two cases of inorder-next

In the case (a), there is one step to the right and one or more steps to the left. Suppose a_i represents the value of the amortized complexity at the node X. When proceeding one step to the right, $a_{i+1} = t_{i+1} + \Phi(S_{i+1}) - \Phi(S_i) = 1 + r(X) + 1 - r(X) = 2$, where r is the rank of a node. And when proceeding to the left, $a_{i+2} = 1 + r(X) + 1 - 1 - (r(X) + 1) = 0$. Continuing in this way we see that in the case (a), starting from X, the next value in inorder of a is 2. Calculating in the same way also the case (b) produces the value 2 for the next a. The value in the root, a_0, is defined to be 0. Also, the final value of a is 0. Thus the amortized complexity of $n-1$ calls to `inorder_next` is $2(n-1)$, considerably less than n^2.

Task 1

Test the above result experimentally. To do so, make a program that generates random binary trees of arbitrary sizes (the sizes should be at least between 50 and 100) and a program that traverses a binary tree in inorder. The traversal program should count the amount of steps it takes during the traversal. Make a table (number of nodes in a tree, number of steps in the traversal) using at least 30-50 different trees. Compare the experimental results to the estimation of the amortized analysis. (10 points)

Remark. The generation of random binary trees can be done in many different ways. One possibility is to generate a sequence of random integers and then adding them one by one to an originally empty search tree. The other possibility is to start to generate a binary tree and deciding randomly for every node if it has none, left, right or both left and right decendants. Or you can also generate reasonably random binary trees manually, but then you cannot get 100% of the points.

When implementing binary trees, you should use linked structures, but you can decide yourself, what kind of links you need. For example, there should be links to the left and right child of a node, but you can decide, if you need a link from a child to its parent.

Task 2

Consider the data structure *Stack* with operations $push(x)$ and $pop(k)$. The former pushes an item x onto the top of the stack and the latter pops k entries from the stack at once. The implementation is a linked list of entries.

Consider further the worst case analysis of this data structure implementation. The complexity of $push(x)$ is clearly $\mathcal{O}(1)$, but $pop(k)$ requires k reference operations, and so, since k could be as large as the size of the stack, its worst-case complexity is $\mathcal{O}(n)$, where n is the maximal size of the stack. The total complexity of a sequence of m_1 calls to $push(x)$ and m_2 calls to $pop(k)$ will be $\mathcal{O}(m_1+m_2n)$. This is too pessimistic an estimation, because it is not possible to pop more elements than there are in the stack. *Make the amortized analysis of m_1 calls to $push(x)$ and m_2 calls to $pop(k)$, arbitrary intermixed in order to get a better complexity estimation.* (6 points)