Lecture V
AMORTIZATION (Excerpt)

Many algorithms amount to a sequence of operations on a data structure. For instance, the well-known **heapsort algorithm** is a sequence of `insert`'s into an initially empty priority queue, followed by a sequence of `deleteMin`'s from the queue until it is empty. Thus if c_i is the cost of the ith operation, the algorithm's running time is $\sum_{i=1}^{2n} c_i$, since there are $2n$ operations for sorting n elements. We normally ensure that *each* operation is efficient, say $c_i = O(\log n)$, leading to the conclusion that the overall algorithm is $O(n \log n)$. The idea of **amortization** exploits the fact that we may be able to obtain the same bound $\sum_{i=1}^{2n} c_i = O(n \log n)$ without ensuring that each c_i is logarithmic. We then say that the **amortized cost** of each operation is logarithmic. Thus "amortized complexity" is a kind of average complexity although it has nothing to do with probability. Tarjan [3] gives the first systematic account of this topic.

Why amortize? Even in problems where we could have ensured *each* operation is logarithmic time, it may be advantageous to achieve only logarithmic behavior in the amortized sense. This is because the extra flexibility of amortized bounds may lead to simpler or more practical algorithms. In fact, many "amortized" data structures seem to be quite implementable. To give a concrete example, consider any balance binary search tree scheme. The algorithms for such trees must perform considerable book-keeping to maintain its balanced shape. In contrast, we will see an amortization scheme for binary search tree which is considerably simpler and "lax" about balancing. The operative word in such amortized data structures is[1] laziness: try to defer the book-keeping work to the future if it can be helped. This will be clearer when we discuss splay trees below.

This lecture is in 3 parts: we begin by introducing the **potential function framework** for doing amortization analysis. Then we introduce two data structures, **splay trees** and **Fibonacci heaps**, which can be analyzed using this framework. We give a non-trivial application of each data structure: splay trees are used to maintain the convex hull of a set of points in the plane, and Fibonacci heaps are used for implement Prim's algorithm for minimum spanning trees.

§1. The Potential Framework

We formulate an approach to amortized analysis using the concept of "potential functions". Borrowing a concept from Physics, we imagine data structures as storing "potential energy" that can be released to do useful work. First, we view a data structure as a binary search tree as a persistent object that has a state which can be changed by operations (e.g., insert, delete, etc). The *characteristic property* of potential functions is that they are a function of the *current* state of the data structure alone, independent of the history of how the data structure was derived.

A "Counter Example". We begin with a simple example. Suppose that we have a binary counter C that is represented by a linked list of 0's and 1's. The only operation on C is to increment its value. For instance, if $C = (011011)$ then after incrementing, $C = (011100)$. This linked list representation determines our **cost model**: the cost to increment C is defined to be the length of the suffix of C of the form 01^*. (We may assume that C begins with a 0-bit in its binary representation, so a suffix of this form always exists.) Thus in our example, the cost is 3 since C has the suffix 011. Let us now analyse the cost of a sequence of n increments, starting from an initial counter value of 0. In the worst case, an increment operation costs $O(\lg n)$. Therefore a worst-case analysis would conclude that the total cost is $O(n \lg n)$.

[1] In algorithmics, it appears that we like to turn conventional vices (greediness, laziness, gambling with chance, etc) into virtues.

We can do better by using amortized analysis: We also associate with C a **potential** $\Phi = \Phi(C)$ that is equal to the number of 1's in its list representation. Informally, we will "store" in C exactly Φ units of work. To analyze the increment operation, we consider two cases. (I) Suppose the least significant bit of C is 0. Then the increment operation just changes this bit to 1. We can **charge** this operation 2 units – one unit to do the work and one unit to pay for the increase in potential. (II) Suppose an increment operation changes a suffix $\underbrace{0111\cdots11}_{k}$ of length $k \geq 2$ into $\underbrace{1000\cdots00}_{k}$: the cost incurred is $\Theta(k)$. Notice that the potential Φ decreases by $k-2$. This decrease "releases" $k-2$ units of work that can pay for $\Theta(k-2)$ of the cost incurred. So we only need to charge this operation 2 units. Thus, in both cases (I) and (II), we only charge 2 units of work for an operation, and so the total charges over n operations is only $2n$. We conclude that the amortized cost of incrementing C is $O(1)$.

Abstract Formulation. We present now one abstract formulation of amortization analysis. It is assumed that we are analyzing the cost of a sequence

$$p_1, p_2, \ldots, p_n$$

of **requests** on a data structure D. The term "request" is meant to cover two types of operations: "updates" that modify D and "queries" that need not[2] modify D. The data structure is dynamically changing: at any moment, the data structure is in some **state**, and each request transforms the current state of D. Let D_i be the state of the data structure after request p_i, with D_0 the initial state.

Each p_i has a non-negative **cost**, denoted $\textsc{Cost}(p_i)$. This cost depends on the complexity model which is part of the problem specification. To carry out an amortization argument, we must specify a **charging scheme** and a **potential function**. Unlike the cost function, the charging scheme and potential function are not inherent to the complexity model, and requires some amount of ingenuity.

A **charging scheme** is just any systematic way to associate a non-negative number $\textsc{Charge}(p_i)$ to each operation p_i. Informally, we "levy" a **charge** of $\textsc{Charge}(p_i)$ on the operation. We emphasize that this levy need not have any obvious relationship to the cost of p_i. The **credit** of this operation is defined to be the "excess charge",

$$\textsc{Credit}(p_i) := \textsc{Charge}(p_i) - \textsc{Cost}(p_i). \tag{1}$$

In view of this equation, specifying a charging scheme is equivalent to specifying a credit scheme. The credit of an operation can be a negative number (in which case it is really a "debit").

A **potential function** is a non-negative real function Φ on the set of possible states of D satisfying

$$\Phi(D_0) = 0.$$

We call $\Phi(D_i)$ the **potential** of state D_i. The amortization analysis amounts to verifying the following inequality at every step:

$$\textsc{Credit}(p_i) \geq \Phi(D_i) - \Phi(D_{i-1}). \tag{2}$$

We call this the **credit-potential invariant**. We denote the *increase in potential* by

$$\Delta\Phi_i := \Phi(D_i) - \Phi(D_{i-1}).$$

Thus equation (2) can be written: $\textsc{Credit}(p_i) \geq \Delta\Phi_i$.

The idea is that credit is stored as "potential" in the data structure.[3] Conceptually, the potential function and the charging scheme are independently defined, so the truth of the invariant ((2)) is not a

[2] Nevertheless, it may turn out to be advantageous to modify D into some other data structure (equivalent to D, of course). Thus the state of D can change even in case of a query.

[3] Admittedly, we are mixing financial and physical metaphors. The credit or debit ought to be put into a "bank account" and so Φ could be called the "current balance".

foregone conclusion. Moreover, the invariant may sometimes be a strict inequality – this means that some credit is discarded (the analysis is not tight in this case).

If the credit-potential invariant is verified, we can call the charge for an operation its **amortized cost**. This is justified by the following derivation:

$$\begin{array}{rcll} \sum_{i=1}^n \textsc{Cost}(p_i) & = & \sum_{i=1}^n (\textsc{Charge}(p_i) - \textsc{Credit}(p_i)) & \text{(by the definition of credit)} \\ & \leq & \sum_{i=1}^n \textsc{Charge}(p_i) - \sum_{i=1}^n \Delta\Phi_i & \text{(by the credit-potential invariant)} \\ & = & \sum_{i=1}^n \textsc{Charge}(p_i) - (\Phi(D_n) - \Phi(D_0)) & \text{(telescopy)} \\ & \leq & \sum_{i=1}^n \textsc{Charge}(p_i) & \text{(since } \Phi(D_n) - \Phi(D_0) \geq 0\text{).} \end{array}$$

The distinction between "charge" and "amortized cost" should be clearly understood: the former is a definition and the latter is an assertion. A charge can only be called an amortized cost if the overall scheme satisfies the credit-potential invariant.

Summary. In an amortization analysis, we are given a sequence of operations on a data structure, together with a cost model (often implicit). We must invent a charging scheme and a potential function. After verifying that the credit-potential invariant holds for each operation, we may conclude that the charge is an amortized cost.

Exercises

Exercise 1.1: We generalize the example of incrementing binary counters. Suppose we have a collection of binary counters, all initialized to 0. We want to perform a sequence of operations, each of the type

$$\texttt{inc}(C), \quad \texttt{double}(C), \quad \texttt{add}(C, C')$$

where C, C' are names of counters. The operation $\texttt{inc}(C)$ increments the counter C by 1; $\texttt{double}(C)$ doubles the counter C; finally, $\texttt{add}(C, C')$ adds the contents of C' to C while simultaneously set the counter C' to zero. Show that this problem has amortized constant cost per operation.

To be precise, we need to define the cost model. The length of a counter is the number of bits used to store its current value (so the length can change). The cost to double a counter C is just 1 (you only need to prepend a single bit to C). The cost of $\texttt{add}(C, C')$ is the number of bits that the standard algorithm needs to look at (and possibly change) when when adding C and C'. E.g., if $C = 11,1001,1101$ and $C' = 110$, then $C + C' = 11,1010,0011$ and the cost is 9. This is because the algorithm only has to look at 6 bits of C and 3 bits of C'. Note that the first 4 bits of C is not looked at (you can think of them being simply "copied" to the output, although this happens by just not doing anything). After this operation, C has the value $11,1010,0011$ and C' has the value 0.
HINT: The potential of a counter C should take into account the number of 1's as well as the bit-length of the counter.

REMARK: in our cost model, $\texttt{add}(C, C')$ *and* $\texttt{add}(C', C)$ *have the same cost.* How to implement this so that our cost model is realistic is left for the next exercise.

♢

Exercise 1.2: In the previous counter problem, we define a cost model for $\texttt{add}(C, C')$ that depends only on the bit patterns in C and C'. In particular, the cost of $\texttt{add}(C, C')$ and $\texttt{add}(C', C)$ are the same. How can you implement this algorithm so that the cost model is realistic?

HINT: To understand the issues, suppose $C = 11,1010,0011$ and $C' = 11$ as in the previous problem. Instead of $\texttt{add}(C, C')$, suppose we want to implement $\texttt{add}(C', C)$. How can you implement this so that

the cost of 9 is still realistic? If you simply "add C to C'" in the obvious way, the real cost would be the sum of the lengths of C and C', namely $2 + 10 = 12$. One possibility is to first "add C' to C, then rename these counters". But to implement it this way, you need detect which counter is longer, and to always add the shorter counter to the longer. Another way is to copy the initial results of the addition to an intermediate counter before committing yourself as to which counter will be zero'd out.

♢

Exercise 1.3: Joe Smart says: it stands to reason that if we can increment counters for an amortized cost of $O(1)$, we should be able to also support the operation of "decrementing a counter" in addition to those in the previous exercise. Someone pointed out that the potential functions that have been used so far does not bear out this conjecture of Smart. Smart retorts: of course, the failure of any particular potential function is no proof that my suggestion is incorrect.
(a) Can you please give Joe Smart a more convincing argument?
(b) In what way is the intuition of Joe Smart about the symmetry of decrement and increments correct? Formalize this by a result about amortized cost. ♢

Exercise 1.4: Generalize the previous exercise by assuming that the counters need not be initially zero, but may contain powers of 2. ♢

End Exercises

§2. Splay Trees

The **splay tree data structure** of Sleator and Tarjan [2] is a practical approach to implementing all operations listed in §III.2. A key motivation for splay trees is a simple heuristic called the **move-to-front heuristic** – it basically says that if we want to repeatedly access items in a list, then it is a good idea to move any accessed item to the front of the list, to facilitate future accesses to this item. Of course, there is no guarantee that we would want to access this item again in the future. But even if we never again access this item, we have not lost much because the cost of moving the item has already been paid for (using the appropriate accounting method). Amortization (and probabilistic) analysis can be used to prove that this heuristic is a good idea. This material is in appendix A.

The analogue of the move-to-front heuristic for maintaining binary search trees should be clear: after we access (`lookUp`) an item in a tree, we move it to the root. We will call the operation of moving an item to the root **splaying**. More precisely,

$$\texttt{splay}(\textit{Key } K, \textit{Tree } T) \tag{3}$$

re-structures the binary search tree T so that the root of T now contains K (if K occurs in T) or else, the root contains *either* the successor *or* predecessor of K in T. We are indifferent as to whether it is the successor or predecessor. In particular, if K is smaller than any key in T, the root of T after $\texttt{splay}(K, T)$ will contain the smallest key in T. A similar remark applies if K is larger than any key in T. We will exploit these properties of splaying below.

Whenever we use the operation (3) above, the following assumption on T will hold:

T is non-empty and all the keys in T are distinct.

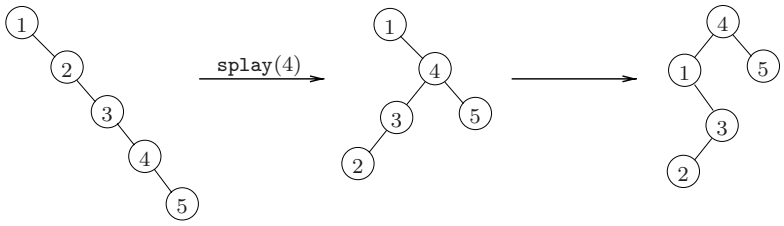

Figure 1: Splaying key 4 (an intermediate step shown).

Since T is non-empty, any key K will have a successor or predecessor (perhaps not both) in T and $\texttt{splay}(K, T)$ can always be well-defined. See figure 1 for examples of splaying.

Before describing the splay algorithm, we show how it will be used.

Reduction to Splaying. We now implement the fully mergeable dictionary ADT (§III.2). The implementation is quite simple: every ADT operation is reducible to one or two splaying operations.

- `lookUp`(*Key* K, *Tree* T): examine the root of $\texttt{splay}(K, T)$. It is important to realize that we intentionally modify the tree T by splaying it. This is necessary for our analysis.
- `insert`(*Item* X, *Tree* T): examine the key K' at the root of $\texttt{splay}(X.\texttt{Key}, T)$. If $K' = X.\texttt{Key}$, we declare an error (recall that keys are distinct in T). If $K' > X.\texttt{Key}$, we can install a new root containing X, and K' becomes the right child of X as in figure 2. The case $K' < X.\texttt{Key}$ is symmetrical. In either case, the new root has key equal to $X.\texttt{Key}$.
- `merge`(*Tree* T_1, T_2) $\rightarrow T$: recall that all the keys in T_1 must be less than any key in T_2. First let $T \leftarrow \texttt{splay}(+\infty, T_1)$. Here $+\infty$ denotes an artificial key larger than any real key in T_1. So the root of T has no right child. We then make T_2 the right subtree of T.
- `delete`(*Key* K, *Tree* T): if $\texttt{splay}(K, T)$ does not contain K in its root, there is nothing to do. Otherwise, delete the root and merge the left and right subtrees.
- `split`(*Key* K, *Tree* T) $\rightarrow T'$: perform $\texttt{splay}(K, T)$ so that the root of T now contains the successor or predecessor of K in T. Split off the right subtree of T, perhaps including the root of T, into a new tree T'.

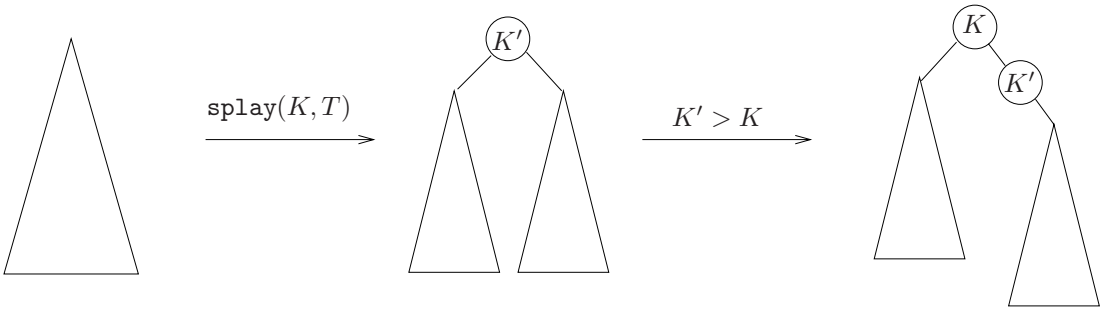

Figure 2: Inserting an item with key K: case $K' > K$.

Reduction to SplayStep. Splaying T at key K is easily accomplished in two stages:

- Perform the usual binary tree search for K. Say we terminate at a node u that contains K in case T contains such a node. Otherwise, u contains the successor or predecessor of K in T.
- Now repeatedly call the subroutine

  `splayStep`(u)

  until u becomes the root of T. Termination is guaranteed because `splayStep`(u) always reduce the depth of u.

It remains to explain the SplayStep subroutine. We need a terminology: A grandchild u of a node v is called a **extreme left grandchild** if u is the left child of the left child of v. Similarly for **extreme right grandchild**.

`splayStep`(*Node* u):
There are three cases.
Base Case. If u.`Parent` is the root,
then we simply `rotate`(u) (see figure 4).
Case I. Else, if u is an extreme (left or right) grandchild,
perform two rotations: `rotate`(u.`Parent`), followed by `rotate`(u). See figure 3.
Case II. Else, we perform `rotate`(u) twice (see figure 3).

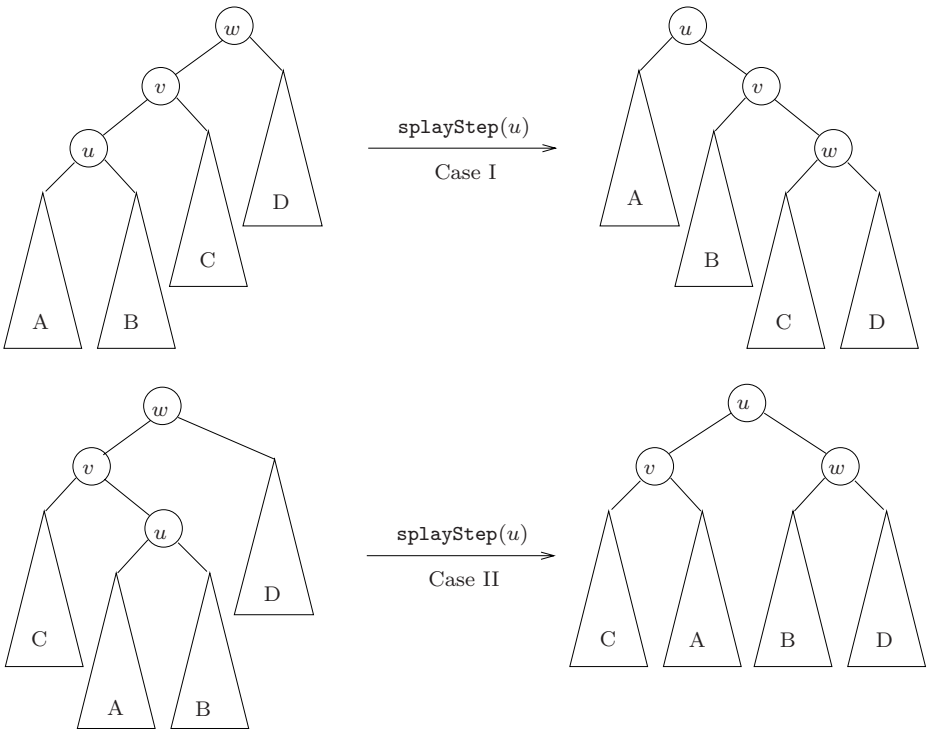

Figure 3: SplayStep at u: Cases I and II.

In figure 1, we see two applications of `splayStep`(4). Following Sleator and Tarjan, we may also call the three cases zig (base case), zig-zig (case I) and zig-zag (case II). It is easy to see that the depth of u decreases by 1 in a zig, and decreases by 2 otherwise. Hence, if the depth of u is h, the splay operation will halt in about $h/2$ `splayStep`'s. Recall in §III.6, we call the zig-zag a "double rotation".

Before moving to the analysis of this data structure, briefly consider the possible behavior of this data structure. Notice that the search trees are by no means required to be balanced. Imagine a sequence of insertions to an empty tree: if the key of each successive insertion is larger than the previous one, we would end up with a linear structure.

Top Down Splaying. We now introduce a variation of splaying. The Sleator-Tarjan splay algorithms requires two passes over the splay path. Suppose we wish to have a one-pass algorithm. The basic idea is this: *for each node that we "visit" in our search path, we will make it the root before we "visit" it.* Initially, we begin at the root so the basic idea is satisfied. The next node (if any) we visit is either the left or right child of the root. So our basic idea amounts to just performing a left or right rotation at the root, and now we continue recursively. This idea has a pitfall (Exercise). The correct solution is as follows. Let the top-down splaying procedure be denoted `topSplay`($KeyK, Nodeu$). We have 4 possible states of our algorithm:

- State 0: Both u_L and u_R have not been visited.
- State 1: u_L but not u_R has been visited.
- State 2: u_R but not u_L has been visited.
- State 3: Both u_L and u_R have been visited.

Here is the transition rule for the states.

State 0: Initially, we are in state 0. If u.`Key` $> K$, then rotate u.`Left` and we move into state 1; if u.`Key` $< K$, then we rotate u.`Right` and we move into state 2.

State 1: If u.`Key` $> K$ then we next move into state 3 and perform the actions

$$v \leftarrow u.\texttt{Left.Right}; rotate(v); rotate(v); \texttt{topSplay}(K, v).$$

Otherwise, u.`Key` $< K$ and remain in state 1 and perform the actions

$$v \leftarrow u.\texttt{Right}; \texttt{rotate}(v); \texttt{topSplay}(K, v).$$

State 2: This is symmetrical to State 1.

State 3: Once we are in state 3, we remain in state 3. If u.`Key` $> K$ then $v \leftarrow u$.`Left.Right` else $v \leftarrow u$.`Right.Left`. In any case, perform the actions

$$rotate(v); rotate(v); \texttt{topSplay}(K, v).$$

An alternative description is to perform cases I, II or III in direct analogy to SplayStep.

What, really, is a Splay Tree? We never quite "characterize splay trees", other than require that they be binary search trees! Let us, for the sake of the exercises below, define a binary search tree to be a **splay tree** if it arises from a sequence of splay tree operations (insertions, deletions, lookups, merges and splits), starting from initially empty trees.

Exercises

Exercise 2.1: Perform the following splay tree operations, starting from an initially empty tree.

$$Ins(3,2,1,6,5,4,9,8,7), LookUp(3), Del(7), Ins(12,15,14,13), Split(8).$$

Show the splay tree after each step. ♢

Exercise 2.2: Show that the worst case time for any of the splay tree operations is $\Omega(n)$. ♢

Exercise 2.3: To splay a tree at a key K, our algorithm begins by doing the conventional `lookUp` of K. If K is not in the tree, and u is the last node reached, then clearly u has at most one child. Prove that u contains the successor or predecessor of K. ♢

Exercise 2.4: Let T be a binary search tree in which every non-leaf has one child. Thus T has a linear structure with a unique leaf.
(a) What is the effect of `lookUp` on the key at the leaf of T?
(b) What is the minimum number of `lookUp`'s to make T balanced? ♢

Exercise 2.5: A variant of the insertion algorithm is to make the inserted node to be equal to either the left or right child of the root. What are the relative advantages/disadvantages of this over what we specified in the text? ♢

Exercise 2.6: (Top Down Splaying)
(a) Explain the "pitfall" mentioned for the obvious implementation of the top-down splaying algorithm.
(b) Give an efficient implementation of `topSplay` as described above. Efficiency here means trying to reduce the number of pointer manipulations, and this may entail combining the pointer manipulations of several rotations.
(c) Do an empirical study of Top Down Splaying, comparing its performance to standard Splaying. ♢

Exercise 2.7: The following question seems to be unexplored. Is every binary search tree a splay tree (defined to mean one that arises from a sequence of splay tree operations, starting from empty trees). A similar question is: can every binary search tree be obtained by repeated splay-tree insertions, starting from an initially empty binary tree? ♢

End Exercises

§3. Splay Analysis

Our main goal next is to prove:

(*) *The amortized cost of each splay operation is $O(\log n)$ assuming at most n items in a tree.*

Let $\textsc{Size}(u)$ denote as usual the number of nodes in the subtree rooted at u, and define its **potential** to be

$$\Phi(u) = \lfloor \lg \textsc{Size}(u) \rfloor .$$

Initially, the data structure has no items and has zero potential. If $S = \{u_1, u_2, \ldots\}$ is a set of nodes, we may write $\Phi(S)$ or $\Phi(u_1, u_2, \ldots)$ for the sum $\sum_{u \in S} \Phi(u)$. If S is the set of nodes in a splay tree T or in the entire data structure then $\Phi(S)$ is called the potential of (respectively) T or the entire data structure.

Lemma 1 (key) *Let Φ be the potential function before we apply* `splayStep`*(u), and let Φ' be the potential after. The credit-potential invariant is preserved if we charge the SplayStep*

$$3(\Phi'(u) - \Phi(u)) \tag{4}$$

units of work in cases I and II. In the base case, we charge one extra unit, in addition to the charge (4).

The main goal (*) follows easily from this key lemma. To see this, suppose that splaying at u reduces to a sequence of k SplaySteps at u and let $\Phi_i(u)$ be the potential of u after the ith SplayStep. The total charges to this sequence of SplaySteps is

$$1 + \sum_{i=1}^{k} 3[\Phi_i(u) - \Phi_{i-1}(u)] = 1 + 3[\Phi_k(u) - \Phi_0(u)]$$

by telescopy. Note that the "1" comes from the fact that the last SplayStep may belong to the base case. Clearly this total charge is at most $1 + 3 \lg n$. To finish off the argument, we must account for the cost of looking up u. But it easy to see that this cost is proportional to k and so it can be covered by charging one extra unit to every SplayStep. This only affects the constant factor in our charging scheme. This concludes the proof of the main goal.

We now address address the Key Lemma. The following is a useful remark about rotations:

Lemma 2 *Let Φ be the potential function before a rotation at u and Φ' the potential function after. Then the increase in potential of the overall data structure is at most*

$$\Phi'(u) - \Phi(u).$$

The expression $\Phi'(u) - \Phi(u)$ is always non-negative.

Proof. We refer to figure 4. The increase in potential is

$$\begin{array}{rcll} \Delta\Phi & = & \Phi'(u,v) - \Phi(u,v) & \\ & = & \Phi'(v) - \Phi(u) & (\text{as } \Phi'(u) = \Phi(v)) \\ & \leq & \Phi'(u) - \Phi(u) & (\text{as } \Phi'(u) \geq \Phi'(v)). \end{array}$$

It is obvious that $\Phi'(u) \geq \Phi(u)$. **Q.E.D.**

Proof of Key Lemma. The base case is almost immediate from lemma 2: the increase in potential is at most $\Phi'(u) - \Phi(u)$. This is at most $3(\Phi'(u) - \Phi(u))$ since $\Phi'(u) - \Phi(u)$ is non-negative. The charge of $1 + 3(\Phi'(u) - \Phi(u))$ can therefore pay for the cost of this rotation and any increase in potential.

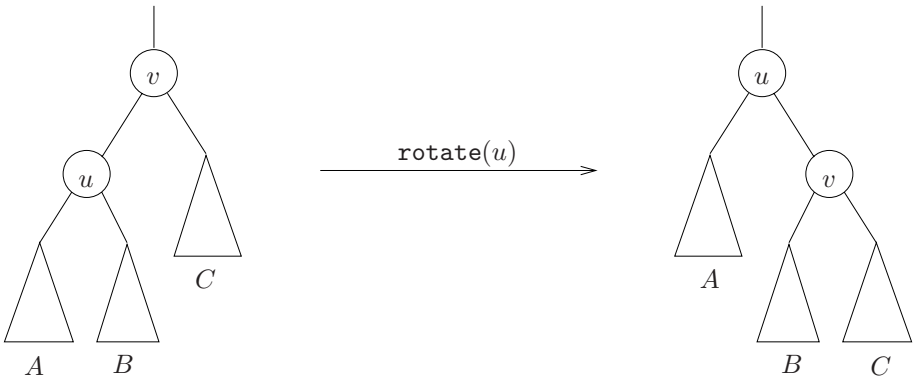

Figure 4: Rotation at u.

Refer to figure 3 for the remaining two cases. Let the sizes of the subtrees A, B, C, D be a, b, c, d, respectively.

Consider case I. The increase in potential is

$$\begin{array}{rcll} \Delta\Phi & = & \Phi'(u,v,w) - \Phi(u,v,w) & \\ & = & \Phi'(v,w) - \Phi(u,v) & (\text{as } \Phi'(u) = \Phi(w)) \\ & \leq & 2(\Phi'(u) - \Phi(u)) & (\text{as } 2\Phi'(u) \geq \Phi'(v,w), \quad 2\Phi(u) \leq \Phi(u,v)). \end{array}$$

Since $\Phi'(u) \geq \Phi(u)$, we have two possibilities: (a) If $\Phi'(u) > \Phi(u)$, then the charge of $3(\Phi'(u) - \Phi(u))$ can pay for the increased potential *and* the cost of this splay step. (b) Next suppose $\Phi'(u) = \Phi(u)$. By assumption, $\Phi'(u) = \lfloor \lg(3 + a + b + c + d) \rfloor$ and $\Phi(u) = \lfloor \lg(1 + a + b) \rfloor$ are equal. Thus $1 + a + b > 2 + c + d$, and so $3 + a + b + c + d > 2(2 + c + d)$ and

$$\Phi'(w) = \lfloor \lg(1 + c + d) \rfloor < \lfloor \lg(3 + a + b + c + d) \rfloor = \Phi(u).$$

Also,

$$\Phi'(v) \leq \Phi'(u) = \Phi(u) \leq \Phi(v).$$

Combining these two inequalities, we conclude that

$$\Phi'(w, v) < \Phi(u, v).$$

Hence $\Delta\Phi = \Phi'(w, v) - \Phi(u, v) < 0$. Since potentials are integer-valued, this means that $\Delta\Phi \leq -1$. Thus the change in potential releases at least one unit of work to pay for the cost of the splay step. Note that in this case, we charge nothing since $3(\Phi'(u) - \Phi(u)) = 0$. Thus the credit-potential invariant holds.

Consider case II. The increase in potential is again $\Delta\Phi = \Phi'(v, w) - \Phi(u, v)$. Since $\Phi'(v) \leq \Phi(v)$ and $\Phi'(w) \leq \Phi'(u)$, we get

$$\Delta\Phi \leq \Phi'(u) - \Phi(u).$$

If $\Phi'(u) - \Phi(u) > 0$, then our charge of $3(\Phi'(u) - \Phi(u))$ can pay for the increase in potential and the cost of this splay step. Hence we may assume otherwise and let $t = \Phi'(u) = \Phi(u)$. In this case, our charge is $3(\Phi'(u) - \Phi(u)) = 0$, and for the credit potential invariant to hold, it suffices to show

$$\Delta\Phi < 0.$$

It is easy to see that $\Phi(v) = t$, and so $\Phi(u, v) = 2t$. Clearly, $\Phi'(v, w) \leq 2\Phi'(u) = 2t$. If $\Phi'(v, w) < 2t$, then $\Delta\Phi = \Phi'(v, w) - \Phi(u, v) < 0$ as desired. So it remains to show that $\Phi'(v, w) = 2t$ is impossible. For, if $\Phi'(v, w) = 2t$ then $\Phi'(v) = \Phi'(w) = t$ (since $\Phi'(v), \Phi'(w)$ are both no larger than t). But then

$$\Phi'(u) = \left\lfloor \lg(\textsc{Size}'(v) + \textsc{Size}'(w) + 1) \right\rfloor \geq \left\lfloor \lg(2^t + 2^t + 1) \right\rfloor \geq t + 1,$$

a contradiction. This proves the Key Lemma.

We conclude with the main result on splay trees.

Theorem 3 *A sequence of m splay tree requests (*`lookUp`*,* `insert`*,* `merge`*,* `delete`*,* `split`*) involving a total of n items takes $O(m \log n)$ time to process. As usual, we assume that the potential of the data structure is initially 0.*

Proof. This follows almost immediately from (*) since each request can be reduced to a constant number of splay operations plus $O(1)$ extra work. We need to attend to one detail in Insertion. Here, we introduce a new node with potential at most $\lg n$. This increase of potential must be charged but clearly this additional does not change our overall cost. Similarly for Merge and Deletion. **Q.E.D.**

Application: Splaysort Clearly we can obtain a sorting algorithm by repeated insertion into a splay tree. Such an algorithm has been implemented [1]. Splaysort has the ability to take advantage of "presortedness" in the input sequence and hence may run faster than Quicksort for some inputs. One way to quantify presortedness is to count the number of pairwise inversions in the input sequence.

Exercises

Exercise 3.1: Where in the proof is the constant "3" actually needed in our charge of $3(\Phi'(u) - \Phi(u))$? ♢

Exercise 3.2: Adapt the proof of the Key Lemma to justify the following variation of SplayStep:

> VarSplayStep(u):
> (Base Case) if u is a child or grandchild of the root,
> then rotate once or twice at u until it becomes the root.
> (General Case) else rotate at u.`Parent`, followed by two rotations at u.

♢

Exercise 3.3:
(i) Is it true that splays always decrease the height of a tree? The average height of a tree? (Define the average height to be the average depth of the leaves.)
(ii) What is the effect of splay on the last node of a binary tree that has a linear structure, *i.e.*, in which every internal node has only one child? HINT: First consider two simple cases, where all non-roots is a left child and where each non-root is alternately a left child and a right child. ♢

Exercise 3.4: Assume that node u has a great-grandparent. Give a simple description of the effect of the following sequence of three rotations: `rotate`(u.`Parent.Parent`); `rotate`(u.`Parent`); `rotate`(u). ♢

Exercise 3.5: For any node u,

$$\Phi(u_L) = \Phi(u_R) \Rightarrow \Phi(u) = \Phi(u_L) + 1$$

where u_L, u_R are the left and right child of u. ♢

Exercise 3.6: Modify our splay trees to maintain (in addition to the usual children and parent pointers) pointers to the successor and predecessor of each node. Show that this can be done without affecting the asympotic complexity of all the operations (`lookUp`, `insert`, `delete`, `merge`, `split`) of splay trees. ♢

Exercise 3.7: We consider some possible simplifications of the `splayStep`.
(A) One-rotation version: Let `splayStep`(u) simply amount to `rotate`(u).
(B) Two-rotation version:

> SplayStep(u):
> (Base Case) if u.`Parent` is the root, `rotate`(u).
> (General Case) else do `rotate`(u.`Parent`), followed by `rotate`(u).

For both (A) and (B):
(i) Indicate how the proposed SplayStep algorithm differs from the original.
(ii) Give a general counter example showing that this variation does not permit a result similar to the Key Lemma. ♢

Exercise 3.8: Modify the above algorithms so that we allow the search trees to have identical keys. Make reasonable conventions about semantics, such as what it means to lookup a key. ♢

Exercise 3.9: Can we use the simpler potential function $\Phi(u) = \lg \text{Size}(u)$ in our splay analysis? ♢

End Exercises

References

[1] A. Moffat, G. Eddy, and O. Petersson. Splaysort: Fast, versatile, practical. *Software - Practice and Experience*, 126(7):781–797, 1996.

[2] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. of the ACM*, 32:652–686, 1985.

[3] R. E. Tarjan. Amortized computational complexity. *SIAM J. on Algebraic and Discrete Methods*, 6:306–318, 1985.