Lecture VI
AMORTIZATION

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. In worst case analysis, we 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 [9] 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 are relatively easy to implement. 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, when we study splay trees, we will see an amortization scheme for binary search trees 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, when it might be more convenient to do this work. This remark will be illustrated by splay trees below.

This chapter 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 such 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^* (i.e., 0 followed by zero or more occurrences of 1). We

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

may assume that C always begins with a 0-bit in its binary representation, so a suffix of the form $01*$ always exists. Thus in our example, the cost is 3 since C has the suffix 011.

The problem we consider is to bound the cost of a sequence of n increments, starting from an initial counter value of 0. For instance, for $n = 5$, our counter C goes through the following sequence of states:

$$(0), \quad (01), \quad (010), \quad (011), \quad (0100), \quad (0101).$$

In the worst case, an increment operation costs $\Theta(\lg n)$. Therefore a worst-case analysis would conclude that the total cost is $O(n \lg n)$. We can do better by using amortized analysis: let us associate with C a **potential** $\Phi = \Phi(C)$ that is equal to the number of 1's in its list representation. For instance, $\Phi(011011) = 4$. Informally, we will "store" $\Phi(C)$ units of work in C. 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 k. Notice that the potential Φ decreases by $k-2$. This decrease "releases" $k-2$ units of work that can pay for $k-2$ units of the incurred cost. Hence we only need to charge this operation 2 units.

Thus, in both cases (I) and (II), we 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 is $O(1)$ per increment operation.

Abstract Formulation. We present now a formulation of amortization analysis that captures the essence of the above Counter Example. 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. We view a data structure D as comprising two parts: its **contents** and its **state**, with the state representing some organization of the contents. The term "request" is meant to cover two types of operations: **updates** that modify the contents of D and **queries** that are a function of the contents and that does not modify the contents. A query is not required to modify the state of D, but it may be advantageous to do so.

For example, D may be a binary tree storing a set of keys; the contents of D are these keys and the state of D is the binary tree itself. When looking up a key in D, we need not change the state of D. Nevertheless, we might wish to do some rotations to bring the searched for node nearer to the root. This is the "move-to-front" heuristic which we will discuss later.

The data structure D is dynamically changing: at any moment, it 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 part of the problem specification. They are artifacts of our analysis and may require some amount of ingenuity to be formulated.

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* of the ith step 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.[2] Since the potential function and the charging scheme are defined independently of each other, the truth of the invariant (2) is not a foregone conclusion. It must be verified for each 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:

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

When invariant (2) is a strict inequality, it means that some credit is discarded and the analysis is not tight in this case. For our "counter" example, the invariant is tight in every case! This means that our preceding derivation is an equality at each step until the very last step (when we assume $\Phi(D_n) - \Phi(D_0) \geq 0$). Thus we have the exact cost of incrementing a counter from 0 to n is **exactly** equal to

$$\sum_{i=1}^n c_i = 2n - \Phi(D_n)$$

where $\Phi(D_n)$ is the number of 1's in the binary representation of n.

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.

[2] 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".

So what is Amortization? In the amortization framework, we are given a sequence of n requests on a data structure. We are also given a cost model (this may be implicit) which tells us the true cost c_i for the ith operation. We want to upper bound the total cost $\sum_{i=1}^{n} c_i$. In an amortization analysis, we hope to achieve a bound that is tighter than what can be achieved by replacing each c_i by the worst case cost. This requires the ability to take advantage of the fact that the cost of each type of request is variable, but its variability can somehow be smoothened out by sharing cost across different operations. This sharing of costs can be done is various ways. In the potential framework, we are required to 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.

The potential function can be generalized in several ways: it need not be defined just for the data structure, but could be defined for any suitable abstract feature. Thus, we might have one potential function Φ_j for the jth feature ($j = 1, 2, \ldots$). The charge for an operation could be split up in several ways, and applied to each of the potential functions Φ_j.

We illustrate this by giving an alternative argument for the amortized cost of incrementing binary counters: let us set up a "charge account" at each bit position of the binary counter: let A_i be the account at the ith smallest position (the least significant position is $i = 0$, the next most significant position is $i = 1$, etc). Each unit of work changes the value of a particular bit of the counter; if the ith bit is changed, we charge the account A_i. Note that A_0 is n times. The account A_1 is charged $\leq n/2$ times, and in general, the account A_i is charged $\leq n/2^i$ times. Hence the overall charges is at most $\leq n(1 + \frac{1}{2} + \frac{1}{4} + \cdots) \leq 2n$. Hence the amortize cost per increment is ≤ 2.

Note that this charging scheme is actually simpler than the potential method, since we charge each operation the *exact* cost of the operation! We will return to these ideas in a later chapter on the Union Find data structure.

Exercises

Exercise 1.1: Our model and analysis of counters can yield the exact cost to increment from any initial counter value to any final counter value. Show that the exact number of work units to increment a counter from 68 to 125 is 190. ♢

Exercise 1.2: A simple example of amortized analysis is the cost of operating a special kind of pushdown stack. Our stack S supports the following two operations: S.`push`(K) simply add the key K to the top of the current stack. But S.`pop`(K) will keep popping the stack as long at the current top of stack has a key smaller than K (the bottom of the stack is assume to have the key value ∞). The cost for push operation is 1 and the cost for popping $m \geq 0$ items is $m + 1$.

(a) Use our potential framework to give an amortized analysis for a sequence of such push/pop operations, starting from an initially empty stack.

(b) How tight is your analysis? E.g., can it give the *exact cost*, as in our Counter Example?

REMARK: Such a stack is used, for instance, in implementing Graham's algorithm for the convex hull of a set of planar points (see Section 4 on convex hull in this chapter). ♢

Exercise 1.3: Let us 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.

We must define the cost model. The length of a counter is the number of bits used to store its value. 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 modify) 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 4 high-order bits of C are not looked at (think of them as simply being "copied" to the output). 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.

♢

Exercise 1.4: 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 the addition algorithm so that the cost model is justified? HINT: recall that counters are linked lists, and you must describe your algorithm in terms of list manipulation. ♢

Exercise 1.5: Generalize the previous exercise by assuming that the counters need not be initially zero, but may contain powers of 2. ♢

Exercise 1.6: Joe Smart reasons that if we can increment counters for an amortized cost of $\mathcal{O}(1)$, we should be able to also support the operation of "decrementing a counter", in addition to those in the previous exercise. This should have an amortized cost of $\mathcal{O}(1)$, of course.
(a) Can you please give Joe a convincing argument as to why he is wrong?
(b) Joe's intuition about the symmetry of decrement and increment is correct if we change our complexity model. Vindicate Joe by a showing a model where we can increment, decrement, add and double in $O(1)$ per operation. HINT: we allow our counters to store negative numbers. We also need a more general representation of counters.
(c) In your solution to (b), let us add another operation, testing if a counter value is 0. What is the amortized complexity of this operation? ♢

Exercise 1.7: Suppose we want to generate a lexicographic listing of all n-permutations (See Chapter 5 on generating permutations). Give an amortized analysis of this process. ♢

End Exercises

§2. Splay Trees

The **splay tree data structure** of Sleator and Tarjan [8] is a practical approach to implementing all operations listed in §III.2. Splay trees are just ordinary binary search trees – they are only distinguished by the algorithms used to implement standard binary tree operations. These operations invariably contain a splaying operation. The splaying operation, applied to an arbitrary node of the tree, will bring this node to the root position.

Splaying may be traced to an idea called the **move-to-front heuristic**: suppose we want to repeatedly access items in a list, and the cost of accessing the item is proportional to its distance from the front of the list. The heuristic says that it is a good idea to move an accessed item to the front of the list. Intuitively, this move will 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. See Appendix A.

The analogue of the move-to-front heuristic for maintaining binary search trees is this: after we access (i.e., `lookUp`) a key K in a tree T, we must move the node containing K to the root. What if K is not in T? In this case, we move the successor or predecessor of K to move to the root. Recall that the **successor** of K in T is the smallest key K' in T such that $K \leq K'$; the **predecessor** of K in T is the largest key K' in T such that $K' \leq K$. Thus K does not have a successor (resp., predecessor) in T if K is larger (resp., smaller) than any key in T. Also, the successor and predecessor coincide with K iff K is in T. We characterize the **splay** operation as follows:

$$\texttt{splay}(\textit{Key } K, \textit{Tree } T) \to T' \tag{3}$$

re-structures the binary search tree T into an equivalent binary search tree T' so that the key K' at the root of T' is equal to *either* the successor *or* predecessor of K in T. We are indifferent as to whether K' is the successor or predecessor. In particular, if K is smaller than any key in T, then K' is the smallest key in T. A similar remark applies if K is larger than any key in T. Note that any key K will have a successor or predecessor in T, provided T is non-empty. See Figure 1 for examples of splaying.

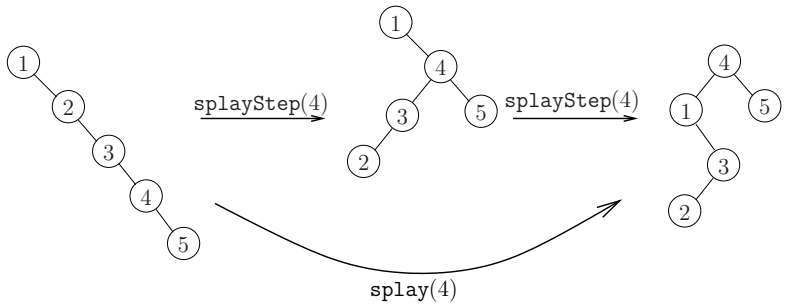

Figure 1: Splaying key 4 (with intermediate step)

If u is any node of T, we say "$\texttt{splay}(u, T)$" to refer to the operation $\texttt{splay}(u.Key, T)$. 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): first perform $\texttt{splay}(K, T)$. Then examine the root of the resulting tree to see if K is at the root. `lookUp` is a success iff K is at the root. It is important to

realize that we deliberately modify the tree T by splaying it. It seems that splaying that has no logical necessity.

- `insert`(*Item* X, *Tree* T): perform the standard binary search tree insertion of X into T. Note that this insertion may not be successful (e.g., when X.`key` is already in T). In any case, we end up with a node u which contains X.`key`, or its successor or predecessor. We splay this node. See Exercise for a variant algorithm.

- `merge`(*Tree* T_1, T_2) $\to T$: recall that all the keys in T_1 must be less than any key in T_2. First let $T \leftarrow$ `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): first perform `splay`(K, T). If the root of the resulting tree does not contain K, there is nothing to delete. Otherwise, delete the root and merge the left and right subtrees, as described in the previous bullet.

- `deleteMin`(*Tree* T): we perform $T' \leftarrow$ `splay`$(-\infty, T)$ and return the right subtree of T'.

- `split`(*Key* K, *Tree* T) $\to T'$: perform `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'.

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, let u be the last node that we visit before the binary tree search algorithm attempts to follow a null pointer. This node 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 which is applied to a non-root u. We need a terminology: A grandchild u of a node v is called a **outer left grandchild** if u is the left child of the left child of v. Similarly for **outer right grandchild**. So an **outer grandchild** is either an outer left or outer right grandchild. If a node has a grandparent and is not an outer grandchild, then it is a **inner grandchild**.

`splayStep`(*Node* u):
There are three cases.
Base Case. If u.`parent` is the root,
then we simply `rotate`(u) (see Figure 7).
Case I. Else, if u is an outer grandchild,
perform two rotations: `rotate`(u.`parent`), followed by `rotate`(u). See Figure 2.
Case II. Else, u is an inner grandchild and we perform a double rotation (`rotate`(u) twice). See Figure 2.

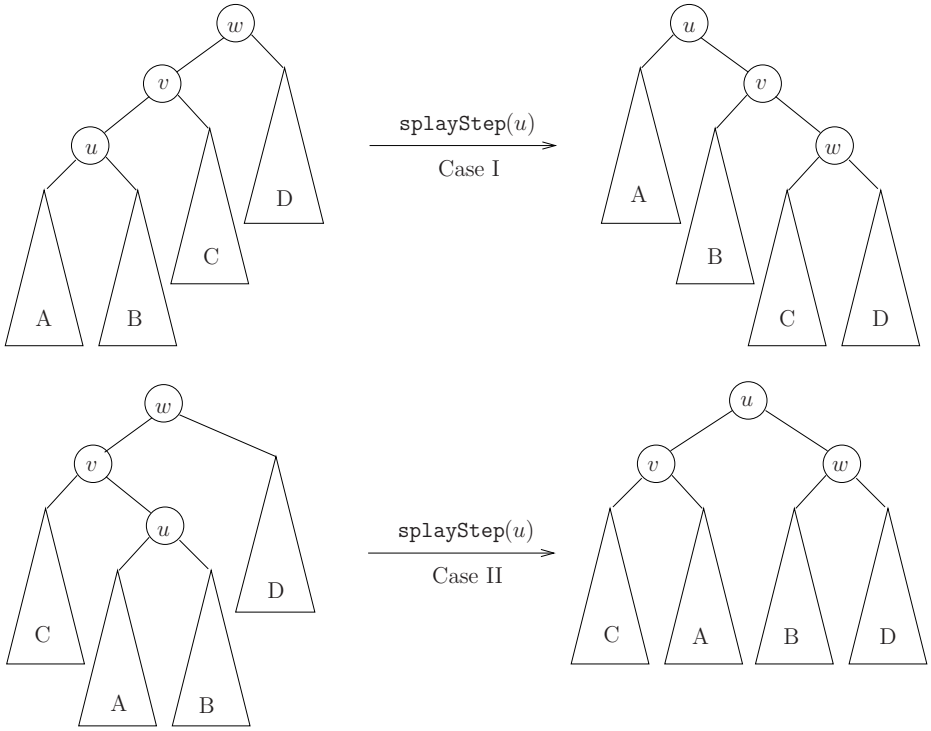

Figure 2: SplayStep at u: Cases I and II.

In Figure 1, we see two applications of `splayStep`(4). Sleator and Tarjan calls the three cases of SplayStep the zig (base case), zig-zig (case I) and zig-zag (case II) cases. 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".

We illustrate the fact that `splay`(K, T) may return the successor or predecessor: let T_0 be the splay tree in Figure 3. If we call `splay`$(6, T_0)$, the result will be T_1 in the figure, where $u.Key = 7$. But if we call `splay`$(6, T_1)$, the result will be the tree T_2 in the figure, where $u.Key = 5$. What if you call `splay`$(6, T_2)$?

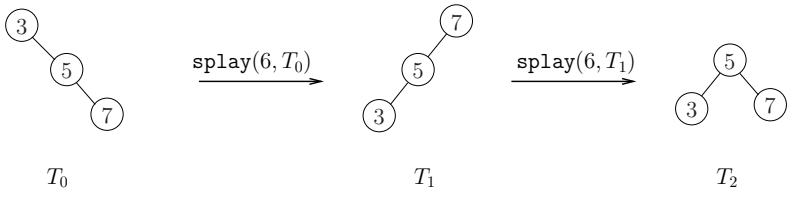

Figure 3: Splaying may return successor or predecessor

Before moving to the analysis of splay trees, 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 introduce a variation of splaying. The above splay algorithms require two passes over the splay path. Suppose we wish to have a one-pass algorithm, denoted `topSplay`(*Key* K, *Node* u).

The basic idea is to avoid the 2-phase approach of (1) going down the tree to look for K, and then (2) to bring the node which contains the successor or predecessor of K up to the root. Instead, we combine the 2 phases by bring each node that we wish to visit to the root *before* we "visit" it. This is called "top-down splaying".

Before giving the correct solution, it is instructive let us illustrate with some false starts: suppose we are looking for the key K and u is the node at the root. If $u.Key = K$, we are done. If not, we must next visit the left or right child (u_L or u_R) of u. But instead of going down the tree these, we simply rotate u_L or u_R to the root! So our basic idea amounts to a repeated left- or right-rotation at the root. Unfortunately, this simple approach has a pitfall (Exercise). To fix this, let us store some state information: we have 4 possible states in our algorithm. Again, u will be the root node and u_L, u_R denote the left and right child of u (these children may be null).

- State 0: Both u_L and u_R have not been visited.
- State 1: u_L, but not u_R, has been visited. Moreover, u_L.`key` $< K$.
- State 2: u_R, but not u_L, has been visited. Moreover, u_R.`key` $> K$.
- State 3: Both u_L and u_R have been visited. Moreover, u_K.`key` $< K < u_R$.`key`.

We have a global variable `State` that is initialized to 0. Here are the possible state transitions. From state 0, we must enter either state 1 or state 2. From states 1 or 2, we can either remain in the same state or enter state 3. Once state 3 is entered, we remain in state 3. Unfortunately this `topSplay` algorithm does not have amortized logarithmic behavior (Exercise).

Here then, is the solution: we maintain 3 splay trees, L, C, R, corresponding to the Left-, Center- and Right-trees. Initially Left- and Right-trees are empty, and the Center-tree is the input tree. The keys in L are less than the keys in C, which are in turn less than the keys in R. Inductively, assume the key K that we are looking for is in C. There are three cases: suppose u is the root.

- CASE (0): This is the base case. The key K we are looking for is equal to the key at u. We attach the left and right children of u at the rightmost tip and leftmost tip of L and R, resp. We then make u the root of a tree with left and right children L and R. See Figure 4. In case K is equal to the key at a child of u, we just rotate that child before applying the preceding transformation.
- CASE (I): Suppose the key K is found in an outer grandchild w of u. By symmetry, assume that w is the left child of v, where v is the left child of u. In this case, we transfer the nodes u, v and the right subtrees of u and v to the left-tip of R, as shown in Figure 4.
- CASE (II): Suppose key K is found in an inner grandchild w of u. By symmetry, assume w is right child of v, where v is the left child of u. In this case, we transfer the node u and its right subtree to the left-tip of R, and the node v and its left-subtree to the right-tip of L, as shown in Figure 4.

The correctness of this procedure is left as an easy exercise.

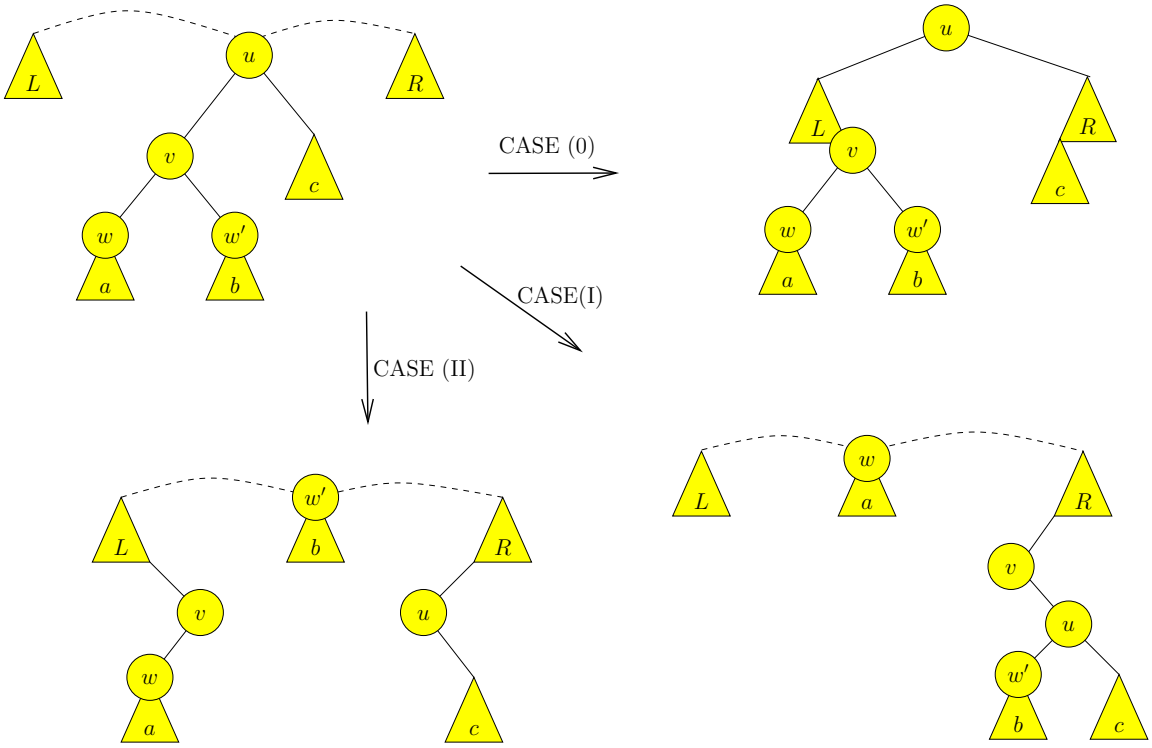

Figure 4: Top Splay: Case I and II

Exercises

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

$\texttt{insert}(3,2,1,6,5,4,9,8,7), \texttt{lookUp}(3), \texttt{delete}(7), \texttt{insert}(12,15,14,13), \texttt{split}(8)$.

Show the splay tree after each step. ♢

Exercise 2.2: Show the result of $\texttt{merge}(T_1, T_2)$ where T_1, T_2 are the splay trees shown in Figure 5. ♢

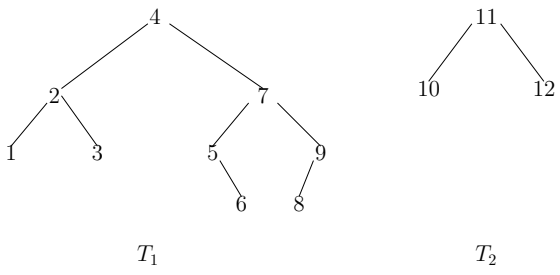

Figure 5: Splay trees T_1, T_2

Exercise 2.3: Consider the insertion of the following sequence of keys into an initially empty tree: $1, -1, 2, -2, 3, -3, \ldots, n, -n$. Let T_n be the final splay tree.

(a) Show T_n for $n = 1, 2, 3$.
(b) State and prove a conjecture about the shape of T_n. ♢

Exercise 2.4: Consider the insertion of the following sequence of keys into an initially empty tree: $3, 2, 1; 6, 5, 4; 9, 8, 7; \ldots; 3n, 3n-1, 3n-2$, where we have written the keys in groups of three to indicate the pattern. Let T_n be the final splay tree.
(a) Show T_n for $n = 1, 2, 3, 4$.
(b) State and prove a conjecture about the shape of T_n. ♢

Exercise 2.5: (Open ended) Prove that if we have any "regular" sequence of insertions, the result is a "regular" splay tree. Part of this problem is to capture various notions of regularity. Let us capture the previous two exercises: let $G(n) = (g_1(n), g_2(n), \ldots, g_k(n))$ where k is a constant and $g_i(n)$ is a integer polynomial in n. E.g. $G(n) = (n, -n)$ and $G(n) = (3n, 3n-1, 3n-2)$ captures the regularity of the previous two exercises. Assume that the sequences $G(n)$ and $G(m)$ have different members for $m \neq n$. Then we want to show that the insertion of the sequence $G(1); G(2); G(3); \ldots; G(n)$ yields a splay tree T_n that is "regular", but in what sense? ♢

Exercise 2.6: 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 2.7: In our operations on splay trees, we usually begin by performing a splay. This is not the case in our insertion algorithm. But consider the following variant for insertion an item X into T:
1. Perform `splay`$(X.\texttt{key}, T)$ to give us an equivalent tree T'.
2. Now examine the key K' at root of T': if $K' = X.\texttt{key}$, we declare an error (recall that keys must be distinct).
3. If $K' > X.\texttt{key}$, we can install a new root containing X, and K' becomes the right child of X; the case $K' < X.\texttt{key}$ is symmetrical. In either case, the new root has key equal to $X.\texttt{key}$. See Figure 6.
(a) Prove that the amortize complexity this insertion algorithm remains $O(\log n)$.
(b) Compare the amortized complexity of this method with the one in the text. Up to constant factors, there is no difference, of course. But which has a better constant factor? ♢

Exercise 2.8: (Top Down Splaying)
(a) What is the "pitfall" mentioned for the initial implementation of the top-down splaying algorithm.
(b) Show that the amortized complexity of the second attempt cannot be $O(\log n)$. To be specific, here is the code:

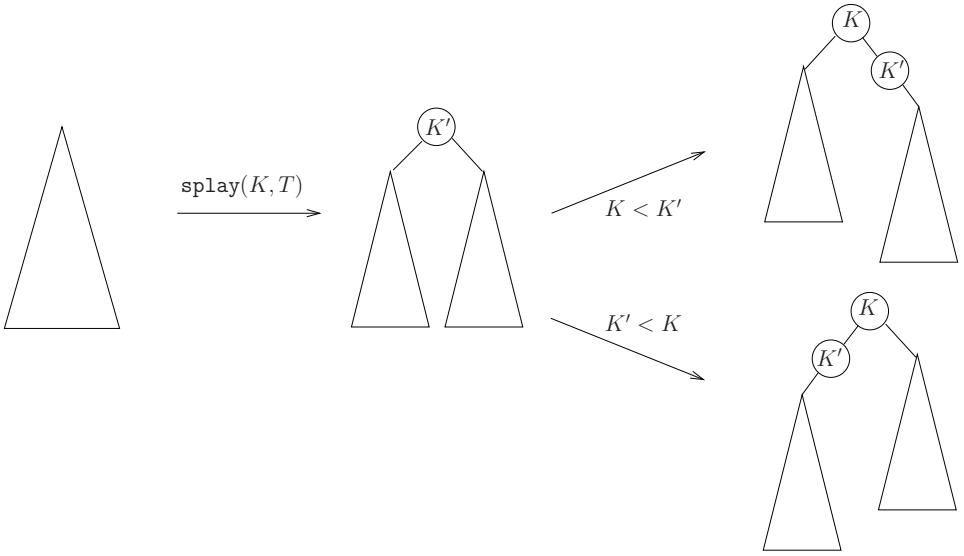

Figure 6: Alternative method to insert a key K.

```
topSplay(K, u)
    Input: K is a key and u a node; we are looking for K in the subtree T_u rooted at u
    Output: We return "found" or "not found". In any case, as side effect, in place of u
            will be a node containing the predecessor or successor of K in the tree T_u.
1.  if (u.key = K) return("found").
2.  case(State)
        State=0:
            if (u.key > K)
                rotate(u.left);
                State ← 2.
            else
                rotate(u.right);
                State ← 1.
        State=1:
            if (u.key > K)
                v ← u.left.right;
                if (v = nil) return("not found").
                rotate^2(v);
                State ← 3;
            else
                v ← u.right;
                if (v = nil) return("not found").
                rotate(v);    ◁ Remain in State 1.
        State=2:
            ···    ◁ Omitted: symmetrical to State 1.
        State=3:
            if (u.key > K)
                v ← u.left.right
            else
                v ← u.right.left.
            if (v = nil) return("not found").
            rotate^2(v)
      ◁ End Case Statement
3.  topSplay(K, v).    ◁ Tail recursion
```

(c) (Open) Does Top Down Splaying have amortized logarithmic complexity? ♢

Exercise 2.9: A **splay tree** is a binary search tree that arises from a sequence of splay tree operations, starting from empty trees.
(a) Is every binary search tree a splay tree?
(b) Let T and T' be equivalent binary search trees (i.e., they store the same set of keys). Can we transform T to T' by repeated splays? ♢

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.*

Before proving this result, let us show that it is a "true" amortization bound. In other words, the worst case bound is not logarithmic. In fact, the worst case bound is linear. To see this, consider the repeated insertion of the keys $1, 2, 3, \ldots, n$ into an initially empty tree. In the ith insertion, we insert key i. We claim that this results in a tree that is just a linear list: if a node contains the key $i \geq 2$, then its left child contain the key $i - 1$. Thus, there is a unique leaf containing the key 1 and the root contains the key n. If we next insert key $n + 1$ into this tree, it will be the right child of the root, but upon splaying, the newly inserted key $n + 1$ will become the root. Thus our induction is preserved. Finally, if we perform a lookup on key 1 in this tree, we must expend $\Theta(n)$ units of work.

To start the amortized analysis, we must devise a potential function: let $\textsc{Size}(u)$ denote, as usual, the number of nodes in the subtree rooted at u. Define its **potential** to be

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

Note that $\textsc{Size}(u) = 1$ iff u is a leaf. Thus $\Phi(u) = 0$ iff u is a leaf. Initially, the data structure has no items and has zero potential. If $S = \{u_1, u_2, \ldots u_k\}$ is a set of nodes, we may write $\Phi(S)$ or $\Phi(u_1, u_2, \ldots u_k)$ 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 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.

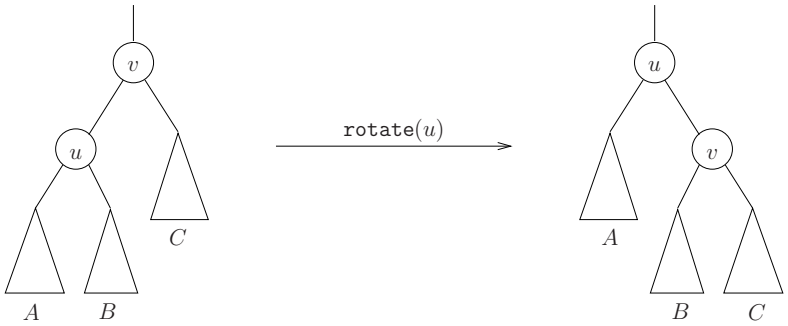

Figure 7: Rotation at u.

Proof. We refer to Figure 7. 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.

Refer to Figure 2 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. Here, $\textsc{Size}'$ denotes the size after the splay step operation. This proves the Key Lemma.

Lemma 3. *Let T' be a binary tree with $n+1$ items, and T is obtained from T' by deleting some leaf x. Then $\Phi(T') - \Phi(T) \leq \lg n$.*

Proof. Let $(u_0, u_1, \ldots, u_m)$ denote the path from the root of T' to $x = u_m$. The potential of each u_i is $\lfloor \lg(n_i + 1) \rfloor$ where $n > n_0 > n_1 > \cdots > n_m = 0$. Furthermore,

$$\Phi(T') - \Phi(T) = \sum_{i=0}^{m-1} \lfloor \lg(n_i+1) \rfloor - \lfloor \lg(n_i) \rfloor .$$

But we observe that $\lfloor \lg(n_i+1) \rfloor - \lfloor \lg(n_i) \rfloor = 1$ iff $n_i + 1$ is a power of 2. There are at most $\lg n$ values of i for which this happens. Hence

$$\Phi(T') - \Phi(T) \leq \lg n.$$

Q.E.D.

We conclude with the main result on splay trees.

Theorem 4. *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. The splay operations are charged $O(\lg n)$ units, and the extra work is charged $O(1)$ units. But there are two important detail that must not be overlooked: sometimes, the extra $O(1)$ work increases the potential by a non-constant amount, and this increase must be properly charged. This situation happens in two situations.

(A) When inserting a new key: the key will become a leaf of the tree, followed by splaying at this leaf. While the splaying cost is accounted for, the act of creating this leaf may also increase the potential of every node along the path to the leaf. By the previous lemma, this increase is at most $\lg n$.

(B) When merging two trees T_1, T_2. In this case, we first perform `splay`$(+\infty, T_1)$. If u is the root of the resulting tree, then u has no right child and we simply attach T_2 as the right subtree of u. This "attachment" will increase the potential of u by less than $1 + \lg(1 + |T_2|/|T_1|) < 1 + \lg n$. Thus we just have to charge this operation an extra $1 + \lg n$ units. Note that deletion is also reduced to merging, and so its charge must be appropriately increased. In any case, all charges remain $O(\lg n)$, as we claimed. **Q.E.D.**

Note that the above argument does not immediately apply to `topSplay`: this is treated in the Exercises. Sherk [7] has generalized splaying to k-ary search trees. In such trees, each node stores an ordered sequence of $t - 1$ keys and t pointers to children where $2 \leq t \leq k$. This is similar to B-trees.

Application: Splaysort Clearly we can obtain a sorting algorithm by repeated insertion into a splay tree. Such an algorithm has been implemented [5]. Splaysort has the ability to take advantage of "presortedness" in the input sequence. One way to quantify presortedness is to count the number of pairwise inversions in the input sequence. E.g., if the input is already in sorted order, we would like to (automatically) achieve an $O(n)$ running time. Algorithms that can take advantage of presortedness may run faster than, say, Quicksort for such inputs. Quicksort, by its very nature, deliberately destroy any structure such as presortedness in its input.

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: Let us define the potential of node u to be $\Phi(u) = \lg(\text{Size}(u))$, instead of $\Phi(u) = \lfloor \lg(\text{Size}(u)) \rfloor$.

(a) How does this modification affect the validity of our Key Lemma about how to charge `splayStep`? In our original proof, we had 2 cases: either $\Phi'(u) - \Phi(u)$ is 0 or positive. But now, $\Phi'(u) - \Phi(u)$ is always positive. Thus it appears that we have eliminated one case in the original proof. What is wrong with this suggestion?
(b) Cosider Case I in the proof of the Key Lemma. Show that if $\Phi'(u) - \Phi(u) \leq \lg(6/5)$ then $\Delta\Phi = \Phi'(w, v) - \Phi(u, v) \leq -\lg(6/5)$. HINT: the hypothesis implies $a + b \geq 9 + 5c + 5d$.
(c) Do the same for Case II. ♢

Exercise 3.4: We continue with the development of the previous question; in particular, we still define $\Phi(u)$ to be $\lg \textsc{Size}(u)$.
(i) Let T be a splay tree on n nodes, and let T' be the result of inserting a new key into T using the standard insertion algorithm. So, the new key appears as a leaf u in T'. Prove that $\Phi(T') - \Phi(T) = O(\lg n)$.
(ii) Prove the Key Lemma under this new definition of potential: *i.e.*, it suffices to charge $3(\Phi'(u) - \Phi(u))$ units for Case I and II of `splayStep`. HINT: Although $\Phi'(u) - \Phi(u)$ is always positive, we need to ensure that $\Phi'(u) - \Phi(u) \geq \alpha$ for some fixed positive constant α. So you must prove that in case $\Phi'(u) - \Phi(u) < \alpha$, the increase in potential, $\Delta\Phi$, is actually a negative value less than $-\alpha$. You may choose $\alpha = \lg(5/4)$.
(iii) Conclude with the alternative proof of the main theorem on splay trees (Theorem 4). ♢

Exercise 3.5:
(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 3.6: 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 3.7: Does our Key Lemma hold if we define $\Phi(u) = \lceil \lg \textsc{Size}(u) \rceil$? ♢

Exercise 3.8: 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 3.9: 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 3.10: 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 3.11: 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 3.12: Consider the `topSplay` algorithm:
(i) Characterize the situations where it gives the same result as the 2-pass `splay` algorithm.
(ii) OPEN: Does it have amortized cost of $O(\log n)$? ♢

End Exercises

§4. Application to Convex Hulls

The following application is interesting because it illustrates the idea of an **implicit binary search tree**. The usual notion of keys is inapplicable. But by using information distributed at a node u and its children u_L and u_R, we are able to perform tests to make decision which simulates searching in a binary search tree.

Given a set X of $n \geq 1$ points in the plane, its **convex hull** $CH(X)$ is the smallest convex subset of the plane that contains X. As $CH(X)$ is a convex polygon, we may represent it as a sequence

$$H = (v_1, v_2, \ldots, v_m), \quad 1 \leq m \leq n$$

where $v_i \in X$ and these v_i's appear as consecutive vertices of the polygon $CH(X)$. We assume that H is a strictly convex sequence, i.e., each 3 consecutive vertices v_{i-1}, v_i, v_{i+1} are not collinear. We shall use H and $CH(X)$ interchangeably. We want to dynamically maintain H subject to two types of requests:

$$\texttt{tangent}(p, H) \quad \text{and} \quad \texttt{insert}(p, H)$$

where p is a new point. If p is outside H, `tangent`(p, H) will return a pair (q, r) of distinct points on H such that the lines $\overline{pq}$ and $\overline{pr}$ are both tangential to H.

We call q and r the **tangent points** of H from p. If p is inside the current hull, we return "↑" since H has no tangent points from p. E.g., in Figure 8(a), v_3, v_5 are the tangent points from p. The request `insert`(p, H) means we want to update H to represent $CH(X \cup \{p\})$. Note that if p is inside the current hull, H is unchanged.

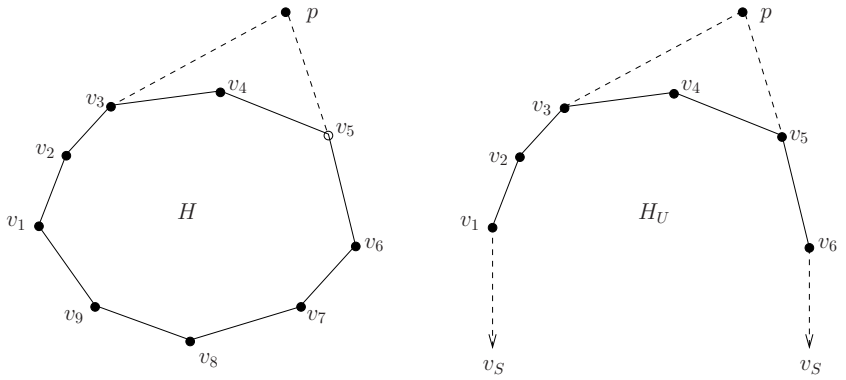

Figure 8: (a) $H = (v_1, \ldots, v_9)$, (b) $H_U = (v_1, \ldots, v_6)$.

Reduction to Half-Hulls. We may assume that v_1 and v_ℓ $(1 \leq \ell \leq m)$ has the smallest and largest x-coordinates among the v_i's. For simplicity, assume that any two consecutive vertices, v_i and v_{i+1}, have distinct x-coordinates so that v_1 and v_ℓ are unique. Then we can break H into two convex chains,

$$H_U = (v_1, v_2, \ldots, v_\ell), \qquad H_L = (v_1, v_m, v_{m-1}, \ldots, v_{\ell+1}, v_\ell).$$

Note that H_U and H_L share precisely their common endpoints. Assuming that H_U lies above the segment $v_1 v_\ell$, we call H_U the **upper hull** and H_L the **lower hull** of H. Let $v_S = (0, -\infty)$ and $v_N = (0, +\infty)$ be points[3] at infinity (the South and North Poles, respectively). Note that H_U is essentially the convex hull of $X \cup \{v_S\}$: indeed,

$$CH((X \cup \{v_S\} = (v_1, v_2, \ldots, v_\ell, v_S).$$

Similarly, H_L is essentially the convex hull of $X \cup \{v_N\}$. Collectively, H_U and H_L are the two **half-hulls** of X. We implement $\texttt{insert}(p, H)$ by reducing it to insertion into the upper and lower hulls:

$$\texttt{insert}(p, H_U); \texttt{insert}(p, H_L).$$

By symmetry, we may focus on upper hulls. Clearly p is inside H iff p is inside both H_U and H_L. Now suppose p is not inside $H_U = (v_1, \ldots, v_\ell)$. Then $\texttt{tangent}(p, H_U)$ returns the pair (q, r) of tangent points of H_U from p where q lies to the left of r. For instance, $\texttt{tangent}(p, H_U)$ returns v_3, v_5 in Figure 8. There are two special cases. If p is left of v_1, then $q = v_S$; if p is right of v_ℓ then $r = v_S$. The details of how to reduce $\texttt{tangent}(p, H)$ to half-hulls is left to an exercise.

Reduction to Fully Mergeable Dictionary Operations. We are now going to store the upper hull H_U in a binary search tree T using the x-coordinates of vertices as keys. Suppose the sequence of points in H_U is $(v_1, \ldots, v_\ell)$ sorted so that

$$v_1 <_x v_2 <_x \cdots <_x v_\ell, \tag{5}$$

where, in general, we write

$$p <_x q \tag{6}$$

for any points p and q such that $p.x < q.y$; similarly, we may write $p \leq_x q$, $p =_x q$, $p <_y q$, etc.

[3] We could define v_S and v_N to be $(r, -\infty)$ and $(r', +\infty)$, respectively, for any $r, r' < \infty$.

To facilitate the insertion of new points, we must be able to split and merge our binary search trees. To see this, suppose we insert a new point p into H_U, and v_{i-1} and v_{j+1} are the tangent points returned by $\texttt{tangent}(p, H_U)$. Then we need to replace a subsequence $(v_i, v_{i+1}, \ldots, v_j)$ $(1 \leq i \leq j \leq \ell)$ by the point p. This can be done efficiently if T is a splay tree and implements the fully mergeable dictionary ADT (see §III.2). By calling the split operation twice, we can split the upper hull into three upper hulls, $T_1 : (v_1, \ldots, v_{i-1})$, $T_2 : (v_i, \ldots, v_j)$ and $T_3 : (v_{j+1}, \ldots, v_\ell)$. Finally, we obtain the new upper hull by forming the tree rooted at p with T_1 and T_3 as left and right subtrees.

We may assume that nodes in T have successor and predecessor pointers. We next show how to implement the requests

$$\texttt{insert}(p, T) \quad \text{and} \quad \texttt{tangent}(p, T)$$

where the binary tree T represents an upper hull. We will give two methods for find the tangent points (q, r) from a query point $p = (p.x, p.y)$.

Method One for Tangent Point: Walking Method. The first method is perhaps easier to understand. We first do a lookup on the key $p.x$. Suppose as a result of this lookup, we determine the two consecutive hull vertices v_k, v_{k+1} such that

$$v_k <_x p \leq_x v_{k+1} \qquad (1 \leq k < \ell). \tag{7}$$

Thus $v_{k+1}.x$ is the the successor of $p.x$ in T. Recall that lookup will find either the successor or predecessor of $p.x$ in T. From this, we can easily determine the points v_k and v_{k+1}, using the successor and predecessor pointers in T.

We can now decide if p is strictly outside the upper hull or not — this amounts to whether p is above the line $\overline{v_k, v_{k+1}}$ or not. If not strictly outside, we return $\uparrow$. Otherwise, we want to return the pair (q, r) of tangent points from p. Consider how we locate q (locating r is similar). We know that $q = v_{i_0}$ for some $i_0 \leq k$. To find i_0, we use the predecessor points to "walk" along the upper hull from v_k, v_{k-1}, v_{k-2}, etc. Moreover, for each $i \leq k$, we can decide whether $i = i_0$, $i < i_0$ or $i > i_0$ according to the following cases:

CASE (i) v_{i-1} and v_{i+1} lie below $\overline{v_i p}$.
Then $i_0 = i$. Here, if $i = 1$ then v_0 is the south pole (which lies below any line).

CASE (ii) v_{i-1}, but not v_{i+1}, lies below $\overline{v_i p}$.
Then $i_0 > i$.

CASE (iii) v_{i+1}, but not v_{i-1}, lies below $\overline{v_i p}$.
Then $i_0 < i$.

We have ignored degeneracies in these three cases. This will be treated in the Exercises.

Method Two: Implicit Binary Search. The previous method for finding tangent points (q, r) is worst case linear time. This is not a problem in case the subchain from q to r is immediately deleted; this is the case if we were inserting the point p. But we now give another method that ensures that finding tangent points has $O(\lg n)$ amortized complexity.

The method uses the 3-way decision (i)-(iii) above to perform an "implicit" binary search for the point $q = v_{i_0}$. But there is another case to contend with:

(CASE (o) p lies inside the upper hull.
In this case i_0 is undefined but we know that $(p, v_{i-1}, v_i, v_{i+1})$ forms a convex quadrilateral for any i. Here, when $i = 1$ or $i = \ell$, the point v_0 or $v_{\ell+1}$ is just the south pole v_S.

FindLeftTangentPoint(p, T):

1. Initialize u to the root of T.
   Let v_i $(1 \le i < \ell)$ be the upper hull vertex stored at u.
   If CASE (o) holds, then return($\uparrow$). ◁ *p is inside the upper hull*
2. Repeat:
   If $v_i \ge_x p$, set $u \leftarrow u$.leftChild.
   Else, we have the three possibilities described above:
      If CASE (i) holds, return(v_i). ◁ $i = i_0$
      If CASE (ii) holds, set $u \leftarrow u$.rightChild. ◁ $i > i_0$
      If CASE (iii) holds, set $u \leftarrow u$.leftChild. ◁ $i < i_0$
   Update i to the index of the vertex in u.

Why is this binary search "implicit"? Looking for v_{i_0} is the same as a lookup on the key value $K = v_{i_0}.x$. But K is not explicitly known. Instead, using the points v_i, v_{i-1}, v_{i+1} and p, we can decide whether K is larger than, equal to, or less than the key of the current node u.

In the above subroutines, we used "geometric primitives" such as checking whether a point is above or below a line. Such primitives must ultimately be reduced to numerical computation and comparisons. In fact, all the geometric primitives can essentially be reduced to a single primitive, the **LeftTurn Predicate**. Given any three points p, q, r, define

$$\texttt{LeftTurn}(p,q,r) = \begin{cases} 0 & \text{if the points } p,q,r \text{ lies on a line} \\ +1 & \text{if the path } (p,q,r) \text{ make a left turn at } q \\ -1 & \text{if the path } (p,q,r) \text{ make a right turn at } q \end{cases} \tag{8}$$

In an Exercise below, you will see how this is easily implemented as the sign of a certain 3×3 determinant. We should observe a peculiarity – we call this a "predicate" even though this is a 3-valued function. In logic, predicates are usually 2-valued (true or false). This is a general phenomenon in geometry, and we might call such 3-valued predicates a **geometric predicate** as opposed to the standard **logical predicate**.

Let us say that the input set X of points is **degenerate** if there exists three distinct points $p, q, r \in X$ such that $\texttt{LeftTurn}(p, q, r) = 0$; otherwise, X is **nondegenerate**. We will assume X is nondegenerate in the following. The main reason for this is pedagogical: the non-degenerate cases are easier to understand. We also note that there are general techniques in computational geometry for handling degeneracies, but that is beyond our scope. See Exercises.

We present the algorithm:

```
FindLeftTangentPoint(p, T):
    INPUT: p is outside the convex hull H represented by T
    OUTPUT: The LeftTangent of p to H.
    1. Initialize u to the root of T.
    2. Repeat:
        Let u_0 = u.succ and u_1 = u.pred.   ◁ These may be NULL .
        If (p <_x u)
            If (u_0 = NULL), Return(v_S).   ◁ South Pole
            u ← u.leftChild and break.
        if ((u_0 ≠ NULL) and LeftTurn(u_0, u, p) = 1))
            u ← u.leftChild and break.
        if ((u_1 ≠ NULL) and LeftTurn(u, u_1, p) = −1))
            u ← u.rightChild and break.
        Return(u).   ◁ This is correct, even if u_0 or u_1 are NULL.
```

Next consider the implementation of `insert`(p, T). We first perform `tangent`(p, T) and assume the non-trivial case where a pair (q, r) of tangent points are returned. Then we need to delete from T those vertices v_i that lies strictly between q and r, and replace them by the point p. This is easily accomplished using the operations of `split` and `merge` on splay trees. This is left for an exercise.

We conclude with the following. Let D be our data structure for the convex hull H (so D is basically a pair of splay trees).

Theorem 5.
(i) Using the data structure D to represent the convex hull H of a set of points, we can support `insert`(p, D) *and* `tangent`(p, D) *requests with an amortized cost of $O(\log |H|)$ time per request.*
(ii) From D, we can produce the cyclic order of points in H in time $O(|H|)$. In particular, this gives an $O(n \log n)$ algorithm for computing the convex hull of a set of n points.

REMARKS:
(a) There are over a dozen known algorithms for convex hulls in the literature (see Exercise for some). The complexity is usually expressed as a function of n, the number of input points. But an interesting concept of "output sensitivity" is to measure the complexity in terms of n and h, where h is the size of the final convex hull. The sharpest result of this kind is from Kirkpatrick and Seidel, achieving the bound $O(n \log h)$.
(b) Our data structure D for representing convex hulls is only semi-dynamic because we do not support the deletion of points. If we want to allow deletion of points, then points that are inside the current convex hull must be represented in the data structure. Overmars and van Leeuwen designed a data structure for a fully dynamic convex hull that uses $O(\log^2 n)$ time for insertion and deletion.

Exercises

Exercise 4.1: What is "convex hull" in 1-dimension? ♢

Exercise 4.2: Let $a, b, c \in \mathbb{R}^2$.
(i) Show that if

$$M = \begin{bmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{bmatrix}$$

then $\det M$ is twice the signed area of the triangle $\Delta(a,b,c)$. Thus, a,b,c are collinear or coincident iff $\det M = 0$. Also, show that $\det M > 0$ iff a,b,c list the vertices counter-clockwise about the triangle.
(ii) What is the relation between $\texttt{sign}(\det M))$ and $\texttt{LeftTurn}(a,b,c)$?
(iii) Let R be the smallest axes-parallel triangle that contains $\Delta(a,b,c)$. Then at least one of the vertices of $\Delta(a,b,c)$ must be at a corner of R. Without loss of generality, let a be the south-west corner of R, b touches the right vertical edge of R and c touches the top horizontal edge of R. Let r be the rectangle with one corner at a and whose opposite corner is (c_x, b_y). Show by a direct geometric argument that the area of $\Delta(a,b,c)$ is equal to $(|R|-|r|)/2$ where $|R|, |r|$ are the areas of the rectangles R, r (respectively). Hence $|R|-|r|$ is the area of the "L" shape $R \setminus r$.
(iv) Verify that the result of (iii) also follows from (i).
(vi) The determinant of M is equal to the determinant of a related 2×2 matrix. ♢

Exercise 4.3: Prove the correctness of the FindLeftTangent(p,T) algorithm. ♢

Exercise 4.4: Treatment of Degeneracy. Recall our definition of degeneracy in the previous question.
(i) First define carefully what we mean by the convex hull (and upper hull) of X in case of degeneracies. You have two choices for this. Also define what we mean by "left-tangent" of p in case p lines on a line through two consecutive vertices of the convex hull.
(ii) Modify the FindLeftTangent(p,T) algorithm so that it works correctly for all inputs, degenerate or not. Actually, you need to describe two versions, depending on which way a degenerate convex hull is defined, etc. ♢

Exercise 4.5: Let us attend to several details in the convex hull algorithm.
(i) Show how to the non-degeneracy assumptions in the text: recall that we assume consecutive vertices on the convex hull have distinct x-coordinates, and input set X is nondegenerate.
(ii) Implement the operation $\texttt{tangent}(p,H)$ in terms of $\texttt{tangent}(p,H_U)$ and $\texttt{tangent}(p,H_L)$.
(iii) Implement the operation $\texttt{insert}(p,H_U)$.
(iv) When we inserted a new point p, we split the original tree into T_1, T_2, T_3 and then form a new splay tree rooted at p with left and right subtrees T_1, T_3. The cost of forming the new tree is $O(1)$. What is the amortized cost of this operation? ♢

Exercise 4.6: Suppose that we do not need to implement the $\texttt{tangent}(\cdot,\cdot)$ query. This would be the case if we are only interested in constructing the convex hull. Show that we can achieve the same $O(\log n)$ per $\texttt{insert}(\cdot,\cdot)$ by a simpler algorithm. Specifically, we can avoid the implicit binary search procedure. ♢

Exercise 4.7: One of the simplest algorithms for convex hull is the so-called Gift-Wrapping algorithm. Start with v_1 the leftmost point of the convex hull. Now try to find v_2, v_3, etc in this order. Show that you can find the next point in $O(n)$ time. Analyze the complexity of this algorithm as a function of n and h, where $1 \leq h \leq n$ is the number of vertices on the convex hull. How does this algorithm compare to $O(n \log n)$ algorithm? ♢

Exercise 4.8: Modified Graham's algorithm for upper hulls. Let $S_n = (v_1, \ldots, v_n)$ be an input sequence of points in the plane. Assume that the points are sorted by x-coordinates and satisfy $v_1 <_x v_2 <_x \cdots <_x v_n$. (Recall "$a <_x b$" means that $a.x < b.x$.) Our goal is to

compute the upper hull of S_n. In stage i ($i = 1, \ldots, n$), we have processed the sequence S_i comprising the first i points in S_n. Let H_i be the upper hull of S_i. The vertices of H_i are stored in a push-down stack data structure, D. Initially, D contain just the point v_1.
(a) Describe a subroutine $Update(v_{i+1})$ which modifies D so that it next represents the upper hull H_{i+1} upon the addition of the new point v_{i+1}. HINT: Assume D contains the sequence of points $(u_1, \ldots, u_h)$ where $h \geq 1$ and u_1 is at the top of stack, with $u_1 >_x u_2 >_x \cdots >_x u_h$. For any point p, let $LT(p)$ denote the predicate `LeftTurn`(p, u_1, u_2). If $h = 1$, $LT(p)$ is defined to be `true`. Implement $Update(v_{i+1})$ using the predicate $LT(p)$ and the (ordinary) operations of push and pop of D.
(b) Using part (a), describe an algorithm for computing the convex hull of a set of n points. Analyze the complexity of your algorithm.

REMARK: The amortized analysis of $S.Update(p)$ was essentially described in an Exercise (Section 1, this Chapter). Graham's original idea is to sort the vertices by their angular angle about some point p_0 in the interior of the convex hull. We must implement this with care, so as to avoid the actual computation of angles (such computation would be inexact and have robustness problems). ♢

Exercise 4.9: The divide-and-conquer for convex hull is from Shamos: divide the set into two sets X_L, X_R, each of size about $n/2$ and the two sets are separated by some vertical line L. Recursively compute their convex hulls H_L, H_R. What kind of operation(s) will allow you to compute $CH(X)$ from H_L and H_R? Show that these operations can be implemented in $O(n)$ time. ♢

End Exercises

§5. Fibonacci Heaps

The **Fibonacci heap data structure** invented by Fredman and Tarjan (1987) gives an efficient implementation of the mergeable queues abstract data type (ADT), which we now explain.

The mergeable queues ADT. The mergeable queues ADT involves domains of three types: *Key*, *Item* and *(mergeable)* `Queue`. As usual, each item stores a key and each queue stores a collection of items. The ADT represents a collection of queues, supporting these operations:

`makeQueue`()$\rightarrow Q$	returns an empty queue Q
`insert`(*Item* x, `Queue` Q)	
`union`(`Queue` Q_1, Q_2)	
`deleteMin`(`Queue` Q)$\rightarrow$ *Item* x	x is minimum item in Q, which is now deleted
`decreaseKey`(*Item* x, *Key* k, `Queue` Q).	

Mergeable queues are clearly extensions of priority queues (§III.2). The above operations are almost self-explanatory. In the union of Q_1, Q_2, the items in Q_2 are first moved into queue Q_1, then queue Q_2 is destroyed. Thus, the number of queues can increase or decrease over the lifetime of the data structure. The operation `deleteMin`(Q) returns a minimum item in Q, and this item is deleted from Q. This operation is unspecified in case Q is empty. In `decreaseKey`(x, k, Q), we make k the new key of x in Q. But this operation assumes k is smaller than the current key of x

– otherwise, we may define it to be either an error or a null-operation (we will leave this decision unspecified).

Union is sometimes known as the **meld** operation. There may be useful operations that should be provided in practice but omitted above for the sake of economy: deleting an item, making a singleton queue, getting the minimum item without deleting it. These can be defined as follows:

`delete`(*Item* x, `Queue` Q)	$\equiv$	`decreaseKey`(x,$-\infty$,Q); `deleteMin`(Q).
`makeQueue`(*Item* x)$\rightarrow Q$	$\equiv$	`makeQueue`()$\rightarrow Q$; `insert`(x,Q).
`min`(`Queue` Q)$\rightarrow x$	$\equiv$	`deleteMin`(Q)$\rightarrow x$; `insert`(x,Q).

The Fibonacci heap data structure. Each mergeable queue is implemented by a Fibonacci heap. A Fibonacci heap H is a collection of trees $T_1, \ldots, T_m$ with these properties:

- Each tree T_i satisfies the min-heap property. In particular, the root of T_i has the minimum item in T_i.
- The roots of these trees are kept in a doubly-linked list, called the **root-list** of H.
- There are two fields H.`min`, $H.n$ associated with H. The field H.`min` points to the node with a minimum key, and $H.n$ is the number of items in H.
- For each node x in a tree T_i, we have four pointers that point to (i) the parent of x, (ii) one of its children, and (iii) two of its siblings. The sibling pointers are arranged so that all the children of x appears in a circular doubly-linked list called the **child-list** of x. If y is a child of x, the **sibling-list** of y is the child-list of x. Also, we keep track of x.`degree` (the number of children of x) and x.`mark` (a Boolean value to be explained).

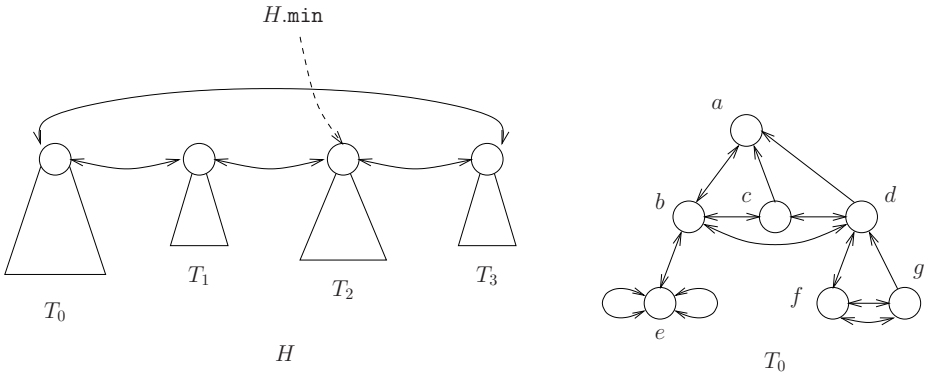

Figure 9: A Fibonacci heap $H = (T_0, \ldots, T_3)$: T_0 in detail

This is illustrated in Figure 9. One of the trees T_0 is shown in detail: the root a of T_0 has 3 children b, c and d and they each point to a; on the other hand, a points only to b. There are two non-trivial sibling lists: (b, c, d) and (f, g).

Linking, cutting and marking. We describe some elementary operations used in maintaining Fibonacci heaps.

(a) If x, y are two roots such that the item in x is not less than the item in y then we can **link** x **and** y; this simply makes y the parent of x. The appropriate fields and structures are then updated: x is deleted from the root-list, and then inserted into the child-list of y, the degree of y incremented, etc. This operation costs $O(1)$.

(b) The converse to linking is **cutting**. If x is a non-root in a Fibonacci heap H then we can perform $Cut(x,H)$: this basically removes x from the child-list of its parent and inserts x into the root-list of H. The appropriate data variables are updated. E.g., the degree of the parent of x is decremented. Again, this operation costs $O(1)$.

(c) We say x is **marked** if $x.\texttt{mark} = true$, and **unmarked** otherwise. Initially, x is unmarked. Our rules will ensure that a root is always unmarked. We mark x if x is not a root and x loses a child (*i.e.*, a child of x is cut); we unmark x when x itself is cut (and put in the root-list). Moreover, we ensure that a marked x does not lose another child before x itself is cut (thereby reverting to unmarked status).

To do amortized analysis, we define a potential function. The **potential** of a Fibonacci heap H is defined as

$$\Phi(H) := t(H) + 2 \cdot m(H)$$

where $t(H)$ is the number of trees in H and $m(H)$ is the number marked items in H. The potential of a collection of Fibonacci heaps is just the sum of the potentials of the individual heaps.

One more definition: let $D(n)$ denote the maximum degree of a node in a Fibonacci heap with n items. We will show later that $D(n) \leq 2 \lg n$.

Remark: The reader may observe how "low-tech" this data structure appears – along with the humble array structure, linked-lists is among the simplest data structures. Yet we intend to achieve the best known overall performance for mergeable queues with Fibonacci heaps. This should be viewed as a testimony to the power of amortization.

§6. Fibonacci Heap Algorithms

We now implement the mergeable queue operations. Our goal is to achieve an amortized cost of $O(1)$ for each operation except for `deleteMin`, which will have logarithmic amortized cost.

Recall that for each operation p, we have a **cost** $\textsc{Cost}(p)$ which will be mostly self-evident in the following description. We must define a **charge** $\textsc{Charge}(p)$. The **credit** is thereby determined: $\textsc{Credit}(p) = \textsc{Charge}(p) - \textsc{Cost}(p)$. This charging scheme will achieve the stated goal in the previous paragraph: $\Theta(1)$ charges for all the non-deletion operations, and $\Theta(\log n)$ for the two deletion operations. Finally, we verify the credit-potential invariant equation (2) for each operation.

`makeQueue`(): we create an empty root-list. The cost is 1, the charge is 1, so credit is 0, and finally $\Delta\Phi = 0$. The credit-potential invariant holds trivially.

> The cost and $\Delta\Phi$ is automatic at this point (our earlier decisions have determined this). Although we said that the "charge" is part of our creative design, at this point, we really have little choice if we wish to satisfy the credit-potential invariant. We might as well define charge to be (at least) the cost plus $\Delta\Phi$.

`insert`(H, x): we create a new tree T containing only x and insert T into the root-list of H. Update H.`min`, etc. Let us check the credit-potential invariant:

$$\textsc{Cost} \leq 1, \quad \textsc{Charge} = 2, \quad \textsc{Credit} \geq 1, \quad \Delta\Phi = 1.$$

`union`(H_1, H_2): concatenate the two root-lists and call it H_1. Update $\min[H_1]$, etc. Checking the credit-potential invariant:

$$\textsc{Cost} \leq 1, \quad \textsc{Charge} = 1, \quad \textsc{Credit} \geq 0, \quad \Delta\Phi = 0.$$

`deleteMin`(H): we remove H.`min` from the root-list, and the child-list of H.`min` can now be regarded as the root-list of another Fibonacci heap. These two circular lists can be concatenated in constant time into a new root-list for H. If t_0 is the old value of $t(H)$, the new value of $t(H)$ is at most $t_0 + D(n)$. Next we need to find the new value of H.`min`. Unfortunately, we do not know the new minimum item of H. There is no choice but to scan the new root-list of H. While scanning, we might as well[4] spend some extra effort to save future work. This is a process called **consolidation** which is explained next.

Consolidation. In this process, we are given a root-list of length L ($L \leq t_0 + D(n)$ above). We must visit every member in the root-list, and at the same time do repeated linkings *until there is at most one root of each degree*. We want to do this in $O(L)$ time. By assumption, each root has degree at most $D(n)$.

The basic method is that, for each root x, we try to find another root y of the same degree and link the two. So we create a 'new' root of degree $k+1$ from two roots of degree k. If we detect another root of degree $k+1$, we link these two to create another 'new' root of degree $k+2$, and so on. The way that we detect the presence of another root of the same degree is by indexing into an array $A[1..D(n)]$ of pointers. Initialize all entries of the array to nil. Then we scan each item x in the root-list. If $k = x$.`degree` then we try to "insert" x into $A[k]$. This means making $A[k]$ point to x. But we only do this if $A[x.\texttt{degree}] = \textsf{nil}$; in case $A[x.\texttt{degree}] \neq \textsf{nil}$, then it points to some y. In this case, link x to y or vice-versa. If x is linked to y, the latter now has degree $k+1$ and we try to "insert" y into $A[k+1]$, and so on. So each failed insertion leads to a linking, and there are at most L linking operations. Since each linking removes one root, there are at most L linkings in all. (This may not be obvious if we see this the wrong way!) Thus the total cost of consolidation is $O(L)$.

Returning to `deleteMin`, let us check its credit-potential invariant.

$$\begin{aligned} &\textsc{Cost} \leq 1 + t_0 + D(n), \qquad \textsc{Charge} = 2 + 2D(n), \\ &\textsc{Credit} \geq 1 + D(n) - t_0, \\ &\Delta\Phi \leq 1 + D(n) - t_0. \end{aligned}$$

We need to explain our bound for $\Delta\Phi$. Let t_0, m_0 refer to the values of $t(H)$ and $m(H)$ before this `deleteMin` operation. If Φ_0, Φ_1 are (respectively) the potentials before and after this operation, then $\Phi_0 = t_0 + 2m_0$ and $\Phi_1 \leq 1 + D(n) + 2m_0$. To see this bound on Φ_1, note that no node can have degree more than $D(n)$ (by definition of $D(n)$) and hence there are at most $1 + D(n)$ trees after consolidation. Moreover, there are at most m_0 marked after consolidation. Then $\Delta\Phi = \Phi_1 - \Phi_0 \leq 1 + D(n) - t_0$, as desired.

`decreaseKey`(x, k, H): this is the remaining operation and we will exploit the marking of items in a crucial way. First, we decrease the key of x to k (first checking that $k \leq x$.`key`). If x is a root,

[4] OK, we may be lazy but not stupid.

we are done. Otherwise, let y be the parent of x. If $k \leq y.\texttt{key}$, we are done. Otherwise, we cut x. Since x is now in the root list, we need to update $H.\texttt{min}$, etc. If x was marked, it is now unmarked. It remains to treat y: If y is a root, we are done. Otherwise, if y was unmarked, we mark y and we are done. If y was marked (*i.e.*, has previously lost a child), then we now "recursively cut" y, using the following code fragment:

```
RecursiveCut(y, H):
    if (y.mark = false and y ≠ root) then y.mark := true;
    if y ≠ root then
            Cut(y, H);
            RecursiveCut(y.parent, H).
```

Note that if $c \geq 1$ is the number of cuts, then $t(H)$ is increased by c, but $m(H)$ is decreased by $c-1$ or c (the latter iff x was marked). This implies $\Delta\Phi \leq c - 2(c-1) = 2 - c$. If

$$\textsc{Cost} \leq c, \quad \textsc{Charge} = 2, \quad \textsc{Credit} \geq 2 - c,$$

then the credit-potential invariant is verified.

SUMMARY: we have achieved our goal of charging $O(1)$ units to every operation except for `deleteMin` which is charged $O(1) + D(n)$. We next turn to bounding $D(n)$. We remark on an unusual feature in our marking scheme: in general, each node y can suffer the loss of at most one child before y itself is made a root. But if y is already a root, we allow it to lose an unlimited number of children.

§7. Degree Bound

Our goal is to show that $D(n) = O(\log n)$.

Recall the ith Fibonacci number $i = 0, 1, 2, \ldots$ is defined by $F_i = i$ if $i = 0, 1$ and $F_i = F_{i-1} + F_{i-2}$ for $i \geq 2$. Thus the sequence of Fibonacci numbers starts out as

$$0, 1, 1, 2, 3, 5, 8, \ldots.$$

We will use two simple facts:

(a) $F_i = 1 + \sum_{j=1}^{i-2} F_j$ for $i \geq 2$.

(b) $F_{j+2} \geq \phi^j$ for $j \geq 0$, where $\phi = (1 + \sqrt{5})/2 > 1.618$.

Fact (a) follows easily by induction, or better still, by "unrolling" the recurrence for F_i. For fact (b), we observe that ϕ is a solution to the equation $x^2 - x - 1 = 0$ so $\phi^2 = 1 + \phi$. Clearly $F_2 = 1 \geq \phi^0$ and $F_3 = 2 \geq \phi^1$. Inductively,

$$F_{j+2} = F_{j+1} + F_j \geq \phi^{j-1} + \phi^{j-2} = \phi^{j-2}(\phi + 1) = \phi^j.$$

Let x be a node in a Fibonacci heap with n items, and let

$$y_1, y_2, \ldots, y_d \tag{9}$$

be the children of x, given in the order in which they are linked to x. So $x.\texttt{degree} = d$ and y_1 is the earliest child (among $y_1, \ldots, y_d$) to be linked to x.

Lemma 6.

$$y_i.\texttt{degree} \geq \begin{cases} 0 & \text{if } i = 1 \\ i-2 & \text{if } i \geq 2 \end{cases}$$

Proof. This is clearly true for $i = 1$. For $i \geq 2$, note that when y_i was linked to x, the degree of x is at least $i-1$ (since at least $y_1, \ldots, y_{i-1}$ are children of x at the moment of linking). Hence, the degree of y_i at that moment is at least $i-1$. This is because we only do linking during consolidation, and we link two roots only when they have the same degree. But we allow y_i to lose at most one child before cutting y_i. Since y_i is not (yet) cut from x, the degree of y_i is at least $i-2$. **Q.E.D.**

Lemma 7. *Let* $\textsc{Size}(x)$ *denote the number of nodes in the subtree rooted at* x *and* d *the degree of* x*. Then*

$$\textsc{Size}(x) \geq F_{2+d}, \quad d \geq 0.$$

Proof. This is seen by induction on $\textsc{Size}(x)$. The result is true when $\textsc{Size}(x) = 1, 2$ since in these cases $d = 0, 1$, respectively. If $\textsc{Size}(x) \geq 3$, let $y_1, \ldots, y_d$ be the children of x as in (9). Then

$$\begin{aligned} \textsc{Size}(x) &= 1 + \sum_{i=1}^{d} \textsc{Size}(y_i) \\ &\geq 1 + \sum_{i=1}^{d} F_{y_i.\texttt{degree}+2} \text{ (by induction)} \\ &\geq 2 + \sum_{i=2}^{d} F_i \text{ (by last lemma)} \\ &= 1 + \sum_{i=1}^{d} F_i = F_{d+2}. \end{aligned}$$

Q.E.D.

It follows that if x has degree d, then

$$n \geq \textsc{Size}(x) \geq F_{d+2} \geq \phi^d.$$

Taking logarithms, we immediately obtain:

Lemma 8.

$$D(n) \leq \log_\phi(n).$$

This completes our analysis of Fibonacci heaps. It is now clear why the name "Fibonacci" arises.

Background: Prior to Fibonacci heaps, the binomial heaps from Vuillemin (1978) were considered the best data structure for the mergeable queue ADT. The exercises below explores some basic properties of binomial heaps. There is some interest to improve the amortized complexity of Fibonnaci heaps to worst case complexity bounds. This was finally achieved by Brodal [**?**] in 2005.

Exercises

Exercise 7.1: Suppose that instead of cutting a node just as it is about to lose a second child, we cut a node just as it is about to lose a third child. Carry out the analysis as before. Discuss the pros and cons of this variant Fibonacci heap. ♢

Exercise 7.2:
(a) Determine $\widehat{\phi}$, the other root of the equation $x^2 - x - 1 = 0$. Numerically compute $\widehat{\phi}$ to 3 decimal places.
(b) Determine F_i exactly in terms of ϕ and $\widehat{\phi}$ HINT: $F_i = A\phi^i + B\widehat{phi}^i$ for constants A, B.
(b) What is the influence of the $\widehat{\phi}$-term on the relative magnitude of F_i? ♢

Exercise 7.3: A binomial tree is a tree whose shape is taken from $\{B_0, B_1, B_2, \ldots\}$ of trees, inductively defined as follows: the singleton node is a binomial tree denoted B_0. If x, y are roots of two binomial B_i are
... ♢

End Exercises

§8. Pointer Model of Computation

There is an esthetically displeasing feature in our consolidation algorithm, namely, its use of array indexing does not seem to conform to the style used in the other operations. Intuitively the reason is that, unlike the other operations, indexing does not fit within the "pointer model" of computation. In this section, we will give the pointer-based solution to consolidation. We will also look at an elegant formalization of the pointer model. This model can be used for a formal theory of computability and complexity. It has many advantages over the standard Turing machines model.

Pointer based consolidation. We outline a purely pointer-based method for consolidation. We rely on the reader's understanding of pointers as found in conventional programming languages such as `C` or `C++`.

Assume that if $k \leq D(n)$ is the maximum degree of any node (past or present) in the Fibonacci heap, we have a doubly-linked list of nodes

$$(R_0, R_1, \ldots, R_k).$$

We call this the "degree register" because every node in the heap of degree i will have a pointer to R_i. Here k is the largest degree of a node that has been seen so far. Note that when we link x to y then the degree of y increments by one and when we cut x, then the parent of x decrements by one, and these are the only possibilities. If item x has its degree changed from i to $i \pm 1$ then we can re-register x by pointing it to $R_{i\pm1}$ in constant time. Occasionally, we have to extend the length of the register by appending a new node R_{k+1} to the doubly-linked list (when some node attains a degree $k + 1$ that is larger than any seen so far). It is thus easy to maintain this degree register.

Now suppose we must consolidate a root list J. By going through the items in J, we can create (with the help of the degree register) a list of lists

$$(L_0, L_1, \ldots, L_k)$$

where list L_i comprises the roots of degree i in J. This takes $O(D(n) + t)$ operations if J has t elements. It is now easy to consolidate the lists $L_0, \ldots, L_k$ into one list in which no two trees have the same degree, using $O(t)$ time. The cost of this procedure is $O(D(n) + t)$, as in the solution that uses array indexing.

But we can take this idea further: we can reinterprete the circular list H as a degree register. Whenever we register a root of degree k, we check if there is already another such root. If so, we simply link them together and recursively register the new root of degree $k + 1$. The worst case cost of registration in $\Theta(\lg n)$, but the amortized cost is only $O(1)$ using the Counter Example analysis.

We get several benefits in this approach: (i) The space for the register H is $D(n) = O(\log n)$. (ii) The operation `deleteMin`(H) amounts to a simple search through this register; hence its worst case time is no longer $O(n)$ but $O(\log n)$. (iii) We have eliminated the explicit consolidation process.

The Pointer Computational Model. We now give a formal model of the pointer model. A **pointer program** Π consists of a finite sequence of instructions that operate on an implicit potentially infinite digraph G. All program variables in Π are of type POINTER, but we also manipulate integer values via these pointers. Each pointer points to some node in G. Each node N in G has four components:

(integer-value, 0-pointer, 1-pointer, 2-pointer).

These are accessed as P.`Val`, P.0, P.1 and P.2 where P is any pointer variable that points to N. There is a special node $N_0 \in G$ and this is pointed to by the nil pointer. By definition. nil.`Val` $= 0$ and nil.i = nil for $i = 0, 1, 2$. Note that with 3 pointers, it is easy to model binary trees.

Pointer Expressions. In general, we can specify a node by a **pointer expression**, ⟨`pointer-expr`⟩, which is either the constant nil, the NEW() operator, or has the form $P.w$ where P is a pointer variable and $w \in \{0, 1, 2\}^*$. The string w is also called a **path**. Examples of pointer expressions:

$$\text{nil}, \quad \text{NEW}(), \quad P, \quad P.0, \quad Q.1, \quad P.2, \quad Q.1202, \quad P.2120120$$

where P, Q are pointer variables. The NEW() operator (with no arguments) returns a returns a pointer to a "spanking new node" N where $N.0 = N.1 = N.2 =$ nil and N.`Val` $= 1$. The only way to access a node or its components is via such pointer expressions.

The integer values stored in nodes are unbounded and one can perform the four arithmetic operations; compare two integers; and assign to an integer variable from any integer expression (see below).

We can compare two pointers for equality or inequality, and can assign to a pointer variable from another pointer variable or the constant nil or the function NEW(). Assignment to a nil pointer has no effect. Note that we are not allowed to do pointer arithmetic or to compare them for the "less than" relation.

The assigment of pointers can be explained with an example:

$$P.0121 \leftarrow Q.20002$$

If N is the node referenced by $P.012$ and N' is the node referenced by $Q.20002$, then we are setting $N.1$ to point to N'. If N is the nil node, then this assignment has no effect.

Naturally, we use the result of a comparison to decide whether or not to branch to a labelled instruction. Assume some convention for input and output. For instance, we may have two special pointers P_{in} and P_{out} that point (respectively) to the input and output of the program.

To summarize: a pointer program is a sequence of instructions (with an optional label) of the following types.

- Value Assignment: ⟨pointer-expr.Val⟩ ← ⟨integer-expr⟩;
- Pointer Assignment: ⟨path-expr⟩ ← ⟨pointer-expr⟩;
- Pointer Comparison: if ⟨pointer-expr⟩ = ⟨pointer-expr⟩ then goto ⟨label⟩;
- Value Comparison: if ⟨integer-expr⟩ ≥ 0 then goto ⟨label⟩;
- Halt

Integer expressions denote integer values. For instance

$$(74 * P.000) - (Q.21 + P)$$

where P, Q are pointer variables. Here, $P.000, Q.21, P$ denotes the values stored at the corresponding nodes. Thus, an integer expression ⟨integer-expr⟩ is either

- Base Case: any literal integer constant (e.g., $0, 1, 74, -199$), a ⟨pointer-expr⟩ (e.g., $P.012, Q$, nil); or
- Recursively:
$$(⟨\text{integer-expr}⟩⟨op⟩⟨\text{integer-expr}⟩)$$

where ⟨op⟩ is one of the four arithmetic operations. Recall that nil.Val $= 0$. Some details about the semantics of the model may be left unspecified for now. For instance, if we divide by 0, the program may be assumed to halt instantly.

For a simple complexity model, we may assume each of the above operations take unit time regardless of the pointers or the size of the integers involved. Likewise, the space usage can be simplified to just counting the number of nodes used.

One could embellish it with higher level constructs such as while-loops. Or, we could impoverish it by restricting the integer values to Boolean values (to obtain a better accounting of the bit-complexity of such programs). In general, we could have pointer models in which the value of a node P.Val comes from any domain. For instance, to model computation over a ring R, we let P.Val be an element of R. We might wish to have an inverse to New(), to delete a node.

List reversal example. Consider a pointer program to reverse a singly-linked list of numbers (we only use 0-pointer of each node to point to the next node). Our program uses the pointer variables P, Q, R and we write $P \leftarrow Q \leftarrow R$ to mean the sequential assignments "$P \leftarrow Q$; $Q \leftarrow R$;".

```
ReverseList:
Input: P_in, pointer to a linked list.
Output: P_out, pointer to the reversal of P_in.
      P ← nil; Q ← P_in;
      if Q = nil then goto E;
           R ← Q.0 ← P;
L:    if R = nil then goto E;
T:         P ← Q ← R ← Q.0 ← P;
      goto L;
E:    P_out ← Q.
```

This program is easy to grasp once the invariant preceding Line T is understood (see Figure 10 and Exercise).

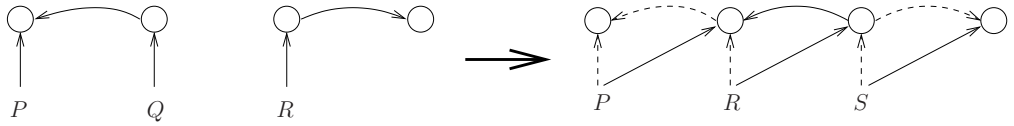

Figure 10: List Reversal Algorithm: the transformation at Line T.

Remark: This model may be more convenient than Turing machines to use as a common basis for discussing complexity theory issues. The main reservation comes from our unit cost for unbounded integers operations. In that case we can either require that all integers be bounded, or else charge a suitable cost $M(n)$ for multiplying n-bit integers, etc, reflecting the Turing machine cost. Of course, the use of pointers is still non-elementary from the viewpoint of Turing machines, but this is precisely the convenience we gain.

Exercises

Exercise 8.1: State the invariant before line T in the pointer reversal program; then proving the program correct. ♢

Exercise 8.2: Write the pointer program for the consolidation. ♢

Exercise 8.3: Implement in detail all the Fibonacci heap algorithms using our pointer model, ♢

Exercise 8.4: Write a sorting program and a matrix multiplication program in this model. What is the time complexity of your algorithms? ♢

End Exercises

§9. Application to Minimum Spanning Tree

The original application of Fibonacci heaps is in computing minimum spanning trees (MST). In Lecture IV §4, we considered Prim's algorithm for MST. The input for MST is a connected bigraph $G = (V, E; C)$ with cost function $C : E \to \mathbb{R}$.

Although our goal is to compute a minimum spanning tree, let us simplify our task by computing only the **cost** of a minimum spanning tree. This is consistent with a general point of pedagogy: for many computational problems that seek to compute a data structure $D = D^*$ which minimizes an associated cost function $f(D)$, it is easier to develop the algorithmic ideas for computing $f(D^*)$ than for computing D^*. Invariably, we could easily transform the algorithm for the minimum value $f(D^*)$ into an algorithm that produces the optimal structure D^*.

Prim-safe sets. It is easy to see that if U is a singleton then U is Prim-safe. Suppose U is Prim-safe and we ask how U might be extended to a larger Prim-safe set. Let us maintain the following information about U:
i) $\texttt{mst}[U]$, denoting the cost of the minimum spanning tree of $G|U$.
ii) For each $v \in V - U$, the least cost $\texttt{lc}_U[v]$ of an edge connecting v to U:

$$\texttt{lc}_U[v] := \min\{C(v, u) : (v, u) \in E, u \in U\}.$$

We usually omit the subscript U and just write "$\texttt{lc}[v]$" without confusion.

In order to find a node $u^* \in V - U$ with the minimum $\texttt{lc}$-value, we will maintain $V - U$ as a **single**[5] mergeable queue Q in which the least cost $\texttt{lc}[u]$ serves as the key of the node $u \in V - U$. Hence extending the Prim-safe set U by a node u^* amounts to a `deleteMin` from the mergeable queue. After the deletion, we must update the information $\texttt{mst}[U]$ and $\texttt{lc}[v]$ for each $v \in V - U$. But we do not really need to consider every $v \in V - U$: we only need to update $\texttt{lc}[v]$ for those v that are adjacent to u^*. The following code fragment captures our intent.

```
Update(u*, U):
1.  U ← U ∪ {u*}.        {This step need not be performed}
2.  mst[U] ← mst[U] + lc[u*].
3.  for v adjacent to u* and v ∉ U, do
        if lc[v] > C[v, u*] then
            lc[v] ← C[v, u*]}.
            DecreaseKey(v, lc[v], Q).
```

We need not explicitly carry out step 1 because U is implicitly maintained as the complement of the items in Q. We now present the MST Cost version of Prim's algorithm.

[5] So we are not using the full power of the mergeable queue ADT which can maintain several mergeable queues. In particular, we never perform the union operation in this application.

MST Cost Algorithm:
Input: $G = (V, E; C)$, a connected costed bigraph.
Output: the cost of an MST of G.
Initialize:
1. $U \leftarrow \{v_0\}$; $\texttt{mst}[U] \leftarrow 0$;
2. for $v \in V - U$, do $\texttt{lc}[v] \leftarrow C(v, v_0)$;
3. Set up $V - U$ as a single mergeable queue Q:
   $Q \leftarrow MakeQueue()$;
   Insert each element of $V - U$ into Q.
Loop:
4. while $Q \neq \emptyset$, do
   $u^* \leftarrow \texttt{deleteMin}(Q)$;
   Update(u^*, U).
5. return($\texttt{mst}[U]$).

We do not need to maintain U explicitly, although it seems clearer to put this into our pseudo-code above. In practice, the updating of U can be replaced by a step to add edges to the current MST.

Analysis. The correctness of this algorithm is immediate from the preceding discussion. To bound its complexity, let $n := |V|$ and $m := |E|$. Assume that the mergeable queue is implemented by a Fibonacci heap. In the Update subroutine, updating the value of $\texttt{lc}[v]$ becomes a DecreaseKey operation. Each operation in Update can be charged to an edge or a vertex. As each edge or vertex is charged at most once, and since the amortized cost of each operation is $O(1)$, the cost of all the updates is $O(m+n)$. The initialization takes $O(n)$ time. In the main procedure, we make $n-1$ passes through the whileloop. So we perform $n-1$ `deleteMin` operations, and as the amortized cost is $O(\log n)$ per operation, this has total cost $O(n \log n)$. We have proven:

Theorem 9. *The cost of a minimum spanning tree of a graph $(V, E; C)$ can be found in $O(|V| \log |V| + |E|)$ operations.*

Final Remarks. The amortization idea is closely related to two other topics. One is "self-organizing data structures". Originally, this kind of analysis is undertaken by assuming the input has certain probability distribution. McCabe (1965) is the first to discuss the idea of move-to-front rule. See "An account of self-organizing systems", W.J. Hendricks, *SIAM J.Comp.*, 5:4(1976); also "Heuristics that dynamically organizes data structures", James R. Bitner, *SIAM J.Comp.*, 8:1(1979)82-100. But starting from the work of Sleator and Tarjan, the competitive analysis approach has become dominant. Albers and Westbrook gives a survey in [2]. Indeed, competitive analysis is the connection to the other major topic, "online algorithms". Albers gives a survey [1].

Exercises

Students should be able to demonstrate understanding of Prim's algorithm by doing hand simulations. The first exercise illustrates a simple tabular form for hand simulation.

Exercise 9.1: Hand simulate Prim's algorithm on the following graph (Figure 11) beginning with v_1:

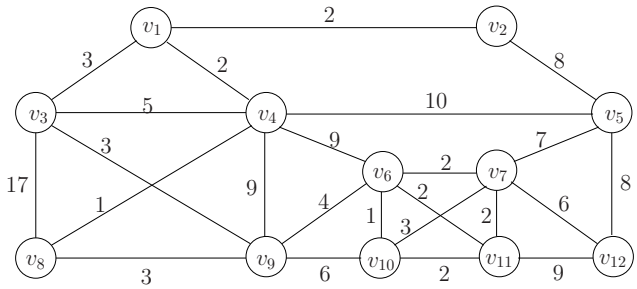

Figure 11: Graph of a House

It amounts to filling in the following table, row by row. We have filled in the first two rows already.

i	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	$mst[U]$	New Edge
1	$\underline{2}$	3	2	∞	∞	∞	∞	∞	∞	∞	∞	2	(v_1, v_2)
2	$*$	"	$\underline{"}$	8	"	"	"	"	"	"	"	4	(v_1, v_4)

Note that the minimum cost in each row is underscored, indicating the item to be removed from the priority queue. ♢

Exercise 9.2: Let G_n be the graph with vertices $\{1, 2, \ldots, n\}$ and for $1 \leq i < j \leq n$, we have an edge (i, j) iff i divides j. For instance, $(1, j)$ is an edge for all $1 < j \leq n$. The **cost** of the edge (i, j) is $j - i$.
(a) Hand simulate (as in the previous exercise) Prim's algorithm on G_{10}. Show the final MST and its cost.

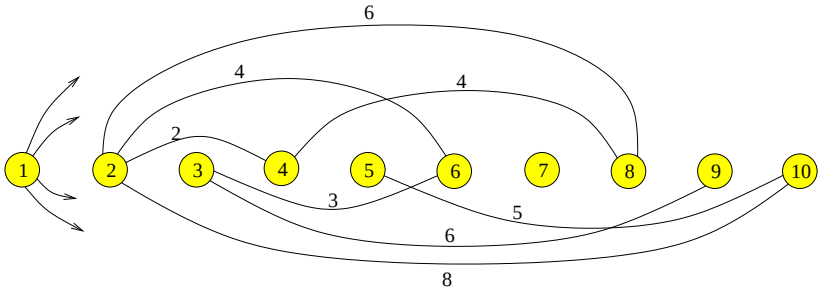

Figure 12: G_{10}: edges from node 1 are omitted for clarity.

(b) What can you say about the MST of G_n? Is it unique? What is the asymptotic cost of the MST? ♢

Exercise 9.3: Modify the above algorithm to compute a minimum spanning tree. ♢

Exercise 9.4: Modify the above algorithm to compute a minimum spanning forest in case the input graph is not connected. ♢

Exercise 9.5: Let $G = (V, E; \mu)$ be an edge-costed bigraph and $S \subseteq E$, $U \subseteq V$. Let $V(S) = \{v \in V : \exists u, (u, v) \in S\}$ denote the *vertices of* S, and $G|U := (U, E'; \mu)$ where $E' = E \cap \binom{U}{2}$

denote the *restriction of G to U*. We define S to be *prim-safe* if S is an MST of $G|V(S)$ and S can be extended into an MST of G. We define U to be *prim-safe* if U is singleton or there exists a prim-safe set S of edges such that $U = V(S)$. Show or give a counter-example:
(a) S is a tree of $G|V(S)$ and can be extended into an MST of G implies S is prim safe.
(b) U is prim-safe implies every MST of $G|U$ is prim-safe. ♢

End Exercises

§A. APPENDIX: List Update Problem

The splay tree idea originates in the "move-to-front rule" heuristic for following **list update problem**: let L be a doubly-linked list of **items** where each item has a unique key. For simplicity, we usually write L as a sequence of keys. This list supports the **access request**. Each access request r is specified by a key (also denoted r), and we satisfy this request by returning a pointer to the item in L with key r. (We assume such an item always exist.) We are interested in a special class of algorithms: such an algorithm α, on an input L and r, searches sequentially in L for the key r by starting at the head of the list. Upon finding the item with key r, α is allowed to move the item to some position nearer the head of the list (the relative ordering of the other items is unchanged). Here are three alternative rules which specify the new position of an updated item:

- (R_0) The **lazy rule** never modifies the list L.
- (R_1) The **move-to-front** rule always make updated item the new head of the list L.
- (R_2) The **transpose rule** just moves the updated item one position closer to the head of the list.

Let α_i denote the list update algorithm based on Rule R_i ($i = 0, 1, 2$). For instance, α_1 is the "move-to-front algorithm". For any algorithm α, let $COST_\alpha(r, L)$ denote the cost of an update request r on a list L using α. For $i = 0, 1, 2$, we write $COST_i(r, L)$ instead of $COST_{\alpha_i}(r, L)$. We may define $COST_i(r, L)$ to be $1 + j$ where j is the position of the accessed item in L. If α is an update algorithm, then $\alpha(L, r)$ denotes the updated list upon applying α to L, r. We extend this notation to a sequence $U = \langle r_1, r_2, \ldots, r_n\rangle$ of requests, by defining

$$\alpha(L, U) := \alpha(\alpha(L, \langle r_1, \ldots, r_{n-1}\rangle), r_n).$$

Similarly, $COST_\alpha(L, U)$ or $COST_i(L, U)$ denotes the sum of the individual update costs.

Example: Let $L = \langle a, b, c, d, e\rangle$ be a list and c an update request. Then $\alpha_0(L, c) = L$, $\alpha_1(L, c) = \langle c, a, b, d, e\rangle$ and $\alpha_2(L, c) = \langle a, c, b, d, e\rangle$. Also $COST_i(L, c) = 4$ for all $i = 0, 1, 2$.

Probabilistic Model. We analyze the cost of a sequence of updates under the lazy rule and the move-to-front rule. We first analyze a probabilistic model where the probability of updating a key k_i is p_i, for $i = 1, \ldots, m$. The lazy rule is easy to analyze: if the list is $L = \langle k_1, \ldots, k_m\rangle$ then the expected cost of a single access request is

$$C(p_1, \ldots, p_m) = \sum_{i=1}^{m} i \cdot p_i.$$

It is easy to see that this cost is minimized if the list L is rearranged so that $p_1 \geq p_2 \geq \cdots \geq p_m$; let C^* denote this minimized value of $C(p_1, \ldots, p_m)$.

What about the move-to-front rule? Let $p(i, j)$ be the probability that k_i is in front of k_j in list L. This is the probability that, if we look at the last time an update involved k_i or k_j, the operation involves k_i. Clearly

$$p(i, j) = \frac{p_i}{p_i + p_j}.$$

The expected cost to update k_i is

$$1 + \sum_{j=1, j\neq i}^{m} p(j, i).$$

The expected cost of an arbitrary update is

$$\begin{aligned}
\widehat{C} &:= \sum_{i=1}^{m} p_i \left[1 + \sum_{j=1, j\neq i}^{m} p(i,j) \right] \\
&= 1 + \sum_{i=1}^{m} \sum_{j\neq i}^{m} p_i \cdot p(i,j) \\
&= 1 + 2 \sum_{1\leq j<i\leq m} \frac{p_i p_j}{p_i + p_j} \\
&= 1 + 2 \sum_{i=1}^{m} p_i \sum_{j=1}^{i-1} \cdot p(j,i) \\
&\leq 1 + 2 \sum_{i=1}^{m} p_i \cdot (i-1) \\
&= 2C^* - 1.
\end{aligned}$$

This proves

$$\widehat{C} < 2C^*. \tag{10}$$

Amortization Model. Let us now consider the amortized cost of a sequence of updates

$$U = (r_1, r_2, \ldots, r_n) \tag{11}$$

on an initial list L_0 with m items. Clearly the worst case cost per update is $O(m)$. So, updates over the sequence U costs $O(mn)$. This worst case bound cannot be improved if we use the lazy rule. The best case for the lazy rule is $O(1)$ per update, or $O(n)$ overall.

What about the move-to-front rule? In analogy to equation (10), we show that it is never incur more than twice the cost of any update algorithm. In particular, it is never more than twice cost of an optimal offline update algorithm α_*: if the cost of α_* is denoted $COST_*$, we prove

$$COST_1(L,U) \leq 2 \cdot COST_*(L,U). \tag{12}$$

We use an amortization argument based on potential functions. A pair (k,k') of keys is an **inversion** in a pair (L,L') of lists if k occurs before k' in L but k occurs after k' in L'. Fix a list L_0 and for any list L, define its **potential** $\Phi(L)$ to be the number of inversions in (L_0, L).

Consider the jth request ($j = 1, \ldots, n$). Let L_j (resp. L_j^*) be the list produced by the move-to-front (resp. optimal) algorithm after the jth request. Write Φ_j for $\Phi(L_j)$. Let c_j and c_j^* denote the cost of serving the jth request under two algorithms (respectively). Let x_j be the item accessed in the jth request and k_j is the number of items that are in front of x_j in both lists L_j and L_j^*. Let ℓ_j be the number of items that are in front of x_j in L_j but behind x_j in L_j^*. Hence

$$c_j = k_j + \ell_j + 1, \quad c_j^* \geq k_j + 1.$$

On the other hand, the number of inversions destroyed is ℓ_j and the number of inversions created is at most k_j. It follows

$$\Phi_j - \Phi_{j-1} \leq k_j - \ell_j.$$

Combining these two remarks,

$$\begin{aligned}
c_j + \Phi_j - \Phi_{j-1} &\leq 2k_j + 1 \\
&\leq 2c_j^* - 1.
\end{aligned}$$

Summing up over all $j = 1, \ldots, n$, we obtain

$$\begin{aligned} COST_1(L_0, U) &= \left(\sum_{j=1}^{n} c_j\right) + \Phi_n - \Phi_0 \\ &\leq \sum_{j=1}^{n}(2c_j^* - 1), \qquad (\text{since } \Phi_n \geq 0, \Phi_0 = 0) \\ &= 2COST_*(L_0, U) - n. \end{aligned}$$

Competitive Algorithms. Let $\beta(k)$ be a function of k. We say an algorithm α is $\beta(k)$-**competitive** if there is some constant a, for all input lists L of length k and for all sequences U of requests

$$COST_\alpha(L, U) \leq \beta(k) \cdot COST_*(L, U).$$

Here $COST_*$ is the cost incurred by the optimal offline algorithm.

We have just shown that the Move-to-Front algorithm is 2-competitive. This idea of competiveness from Sleator and Tarjan is an extremely powerful one as it opens up the possibility of measuring the performance of online algorithms (such as the move-to-front algorithm) without any probabilistic assumption on the input requests.

Remark. An application of the list update problem is data-compression (Exercise). Chung, Hajela and Seymour [3] determine that cost of the move-to-front rule over the cost of an optimal static ordering of the list (relative to some probability of accessing each item) is $\pi/2$. See also Lewis and Denenberg [4] and Purdom and Brown [6].

Exercises

Exercise A.1: We extend the list update problem above in several ways:

(a) One way is to allow other kinds of requests. Suppose we allow insertions and deletions of items. Assume the following algorithm for insertion: we put the new item at the end of the list and perform an access to it. Here is the deletion algorithm: we access the item and then delete it. Show that the above analyses extend to a sequence of access, insert and delete requests.

(b) Extend the list update analysis to the case where the requested key k may not appear in the list.

(c) A different kind of extension is to increase the class of algorithms we analyze: after accessing an item, we allow the algorithm to to transpose any number of pairs of adjacent items, where each transposition has unit cost. Again, extend our analyses above. ♢

Exercise A.2: The above update rules R_i $(i = 0, 1, 2)$ are memoryless. The following two rules require memory.

- (R_3) The **frequency rule** maintains the list so that the more frequently accessed items occur before the less frequently accessed items. This algorithm, of course, requires that we keep a counter with each item.
- (R_4) The **timestamp rule** (Albers, 1995) says that we move the requested item x in front of the first item y in the list that precedes x and that has been requested at most

once since the last request to x. If there is no such y or if x has not been requested so far, do not move x.

(a) Show that R_3 is not c-competitive for any constant c.
(b) Show that R_4 is 2-competitive. ♢

Exercise A.3: (Bentley, Sleator, Tarjan, Wei) Consider the following data compression scheme based on any list updating algorithm. We encode an input sequence S of symbols by each symbol's position in a list L. The trick is that L is dynamic: we update L by accessing each of the symbols to be encoded. We now have a string of integers. To finally obtain a binary string as our output, we encode this string of integers by using a prefix code for each integer. In the following, assume that we use the move-to-front rule for list update. Furthermore, we use the prefix code of Elias in Exercise IV.1.1.6 that requires only

$$f(n) = 1 + \lfloor \lg n \rfloor + 2 \lfloor \lg(1 + \lg n) \rfloor$$

bits to encode an integer n.
(a) Assume the symbols are a, b, c, d, e and the initial list is $L = (a, b, c, d, e)$. Give the integer sequence corresponding to the string $S = abaabcdabaabecbaadae$. Also give the final binary string corresponding to this integer sequence.
(b) Show that if symbol x_i occurs $m_i \geq 0$ times in S then these m_i occurrences can be encoded using a total of

$$m_i f(m/m_i)$$

bits where $|S| = m$. HINT: If the positions of x_i in S are $1 \leq p_1 < p_1 < \cdots < p_{m_i} \leq m$ then the jth occurrence of x_i needs at most $f(p_j - p_{j-1})$. Then use Jensen's inequality for the concave function $f(n)$.
(c) If there are n distinct symbols $x_1, \ldots, x_n$ in S, define

$$A(S) := \sum_{i=1}^{n} \frac{m_i}{m} f\left(\frac{m}{m_i}\right).$$

Thus $A(S)$ bounds the average number of bits per symbol used by our compression scheme. Show that

$$A(S) \leq 1 + H(S) + 2 \lg(1 + H(S))$$

where

$$H(S) := \sum_{i=1}^{n} \frac{m_i}{m} \lg\left(\frac{m}{m_i}\right).$$

NOTE: $H(S)$ is the "empirical entropy" of S. It corresponds to the average number of bits per symbol achieved by the Huffman code for S. In other words, this online compression scheme achieves close to the compression of the offline Huffman coding algorithm. ♢

End Exercises

References

[1] S. Albers. Competitive online algorithms. BRICS Lecture Series LS-96-2, BRICS, Department of Computer Science, University of Aarhus, September 1996.

[2] S. Albers and J. Westbrook. A survey of self-organizing data structures. Research Report MPI-I-96-1-026, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, October 1996.

[3] F. R. K. Chung, D. J. Hajela, and P. D. Seymour. Self-organizing sequential search and hilbert's inequalities. *ACM Symp. on Theory of Computing*, 7, 1985. Providence, Rhode Island.

[4] H. R. Lewis and L. Denenberg. *Data Structures and their Algorithms*. Harper Collins Publishers, New York, 1991.

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

[6] J. Paul Walton Purdom and C. A. Brown. *The Analysis of Algorithms*. Holt, Rinehart and Winston, New York, 1985.

[7] M. Sherk. Self-adjusting k-ary search trees. In *Lecture Notes in Computer Science*, volume 382, pages 373–380, 1989. Proc. **Workshop on Algorithms and Data Structures**, Aug. 17-19, 1989, Carleton University, Ottawa, Canada.

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

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