" ...self-amortizing canals... "

— Mr. Banks in *Mary Poppins* (the movie)

$$1 + 2 + 3 + 4 + \cdots = -\tfrac{1}{12}$$

$$\sqrt{1 + 2\sqrt{1 + 3\sqrt{1 + 4\sqrt{\cdots}}}} = 3$$

— Ramanujan (1887–1920)

Lecture VI
AMORTIZATION

Amortization is the algorithmic idea of distributing computational cost over a period of time. The terminology comes from home mortgages: most Americans pay for a home by taking out a long-term[1] loan called a home mortgage. This loan is repaid in monthly installments, over the period of the loan. In an amortized analysis of an algorithm, we likewise spread the cost of an operation over the entire run of the algorithm. Suppose each run of the algorithm amounts to a sequence of operations on a data structure. For instance, to sort n items, the well-known **heapsort algorithm** (¶III.7) is a sequence of n `insert`'s into an initially empty priority queue, followed by a sequence of n `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 $2n$ priority queue operations in all. 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 $O(n \log n)$. But an **amortization argument** might 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. In this case, we will 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 [13] gave the first systematic account of this topic.

¶1. Why amortize? For the heapsort problem above, we could have ensured that *each* operation is logarithmic time. Nevertheless, we may find it advantageous to consider data structures that achieve logarithmic behavior only in the weaker amortized sense. The extra flexibility afforded by using amortized bounds often lead to simpler or more practical algorithms. Indeed many "amortized" data structures are relatively easy to implement. As a concrete example, consider any balanced binary search tree scheme. The algorithms for such trees must constantly perform considerable book-keeping to maintain its balanced shape. In particular, it must maintain some balance information at each node. In contrast, the splay trees in this chapter provide an amortization scheme for binary search trees that is considerably simpler, needing no extra storage at each node, and is quite lax about balancing. The operative word in amortization is "laziness" – try to defer the book-keeping work to the future, when it might be more convenient to do this work.

In algorithmics, we like to turn conventional vices (greediness, laziness, gambling with chance, etc) into virtues

[1] Long-term means something like 15 to 30 years. Your home is said to be mortgaged, serving as collateral for the loan. This mortgaging analogy fails in some detail. We will see that amortization is closer to a banking model where you maintain an balance account that must not go negative.

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 for maintaining the convex hull of a set of points in the plane, and Fibonacci heaps for implementing 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, i.e., it exists in time and has a history. At each moment it has a state which can be changed by operations such as insert, delete, rotation, etc. The *characteristic property* of potential functions is that they are a function of the *current* state of the data structure alone; it is independent of the history of the data structure. Invariably the potential function depends only on the shape of the underlying binary tree. For instance, consider two binary search trees T_1 and T_2 of the same shape. The first is obtained by inserting the keys $1, 2, 3$ (in this order) into an empty tree; the second is obtained by inserting the keys $3, 1, 2, 4$, followed by deletion of 4. The potential of T_1 and T_2 will be the same, despite their different history.

not your usual counter example...

¶2. A "Counter Example". Let C be a binary counter and the only operation on C is to increment its value. Starting with a counter value of 0, our counter C goes through the following sequence of states as we repeatedly increment it:

$$(0) \to (01) \to (010) \to (011) \to (0100) \to (0101) \to \dots$$

Here, we use the convention of prepending a 0 bit to the standard binary notation for an integer.

Our problem is to bound the total cost of a sequence of n increments, starting from an initial counter value of 0. If the ith increment costs c_i, then the total cost is $\sum_{i=1}^{n} c_i$. It is easy to see that $c_i = O(\lg n)$ since the length of the binary numbers are $O(\lg n)$. This is also a lower bound on the worst case c_i since the increment $(0\underbrace{11\cdots1}_{k}) \to (01\underbrace{00\cdots0}_{k})$ costs $\Theta(k)$, and it is easily checked that k can reach $\lg n$. Using a worst-case analysis, we could only conclude that the total cost $\sum_{i=1}^{n} c_i$ is $O(n \lg n)$.

We now give a better bound using amortized analysis. Assume C is represented by a (sufficiently long) binary array, or[2] alternatively, a linked list of 0 and 1's. This representation determines our **cost model**: *the cost of an increment operation is the number of bits of C that we need to flip.* Note the number of bits to flip is the length of the suffix of C of the form 01^*, i.e., 0 followed by zero or more occurrences of 1. By convention, C always begins with a 0-bit. So a suffix of the form 01^* exists. This cost is at least 1.

For instance, if the value of C is 27, then $C = (011011)$. After incrementing, we get $C = (011100)$. The cost of this increment is 3 since C has the suffix 011 before (resp., 100 after) the increment. Note that we also need to prepend a 0 to C when our increment operation flips the leftmost 0 to a 1: E.g., when we increment $C = (0111)$, the result is $C = (1000)$. Then

[2] The linked list interpretation becomes necessary if we want to do other operations with multiple counters efficiently. See Exercises.

we need to prepend a 0 bit to get $C = (01000)$. The prepending cost is $O(1)$ and it can[3] be absorbed into the overall (positive) cost to flip bits. Therefore we simply say that the cost for incrementing $C = (0111)$ is 4.

To begin our amortized analysis, we associate with C a **potential** $\Phi = \Phi(C)$ that is equal to the number of 1's in its list representation. In our preceding example, $\Phi(011011) = 4$ and $\Phi(011100) = 3$; so the potential of C decreased by 1 after this particular increment. Informally, we will "store" $\Phi(C)$ units of work (or energy) 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. Note that the potential increases by 1 by this operation. 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 to make up the difference.

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. What justifies this conclusion?

¶3. Abstract Formulation. We now formulate an 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 **structure**, where the structure represents some organization of the contents. The (contents, structure) pair is called the **state** of the data structure. The term "request" is meant to cover two types of operations: **updates** that modify the contents of D, and **queries** that computes a function of the contents but do not modify the contents. Thus there is no logical necessity to modify the structure of D in a query. *Nevertheless, it may be advantageous to modify the structure during queries.*

For example, D may be a binary search tree storing a set of keys; the contents of D are these keys and the structure of D is the shape of binary tree itself. Inserting into D is an update request, and looking up a key in D is a query request. In Lookups, it is clear that the D need not change. Nevertheless, to ensure a favorable complexity over a sequence of such operations, we may perform some rotations. We saw this with AVL trees in Chapter III, but those rotations were quite sophisticated (being aimed at worst-case logarithmic complexity). But a truly amortization illustration of this idea is to simply rotate the searched-for node so that its depth is reduced! This is called the "move-to-front" heuristic.

The data structure D is dynamically changing: 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.

[3] Alternatively, we can separately account for the prepending cost: it is at $\Theta(\log n)$ since we do this operation $\lg n$ times.

Each p_i has a positive **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 invent a **charging scheme** and a **potential function**. Unlike the complexity model, the charging scheme and potential function are not part of the problem specification. They are artifacts of our analysis and their invention usually requires some ingenuity.

A **charging scheme** is just any systematic way to associate a real number $\textsc{Charge}(p_i)$ to each operation p_i. E.g., for our Counter Example, we define $\textsc{Charge}(p_i) := 2$ for each p_i. Informally, we "levy" a **charge** of $\textsc{Charge}(p_i)$ on the operation. We emphasize that this levy need not bear any obvious relationship to $\textsc{Cost}(p_i)$. The **credit** of this operation is defined to be the "excess charge",

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

E.g., suppose I charge an operation 5 units when its cost is 3, the credit is 2. In another scheme, I might charge the same operation 2, in which case the credit is -1. In view of (1), 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 function Φ that assigns a non-negative real number to every possible possible state of D. Moreover, we require that

$$\Phi(D_0) = 0$$

where D_0 is the initial state of D. Call $\Phi(D_i)$ the **potential** of state D_i. Let the *increase in potential* of the ith step be denoted by

$$\Delta\Phi_i := \Phi(D_i) - \Phi(D_{i-1}). \tag{2}$$

OK, we are mixing financial and physical metaphors here. Since mortgages is a financial term, perhaps the credit/debit ought to be put into a "bank account" and Φ is the "current balance".

The amortization analysis amounts to verifying the following inequality at every step:

$$\textsc{Credit}(p_i) \geq \Delta\Phi_i. \tag{3}$$

Call (3) the **credit-potential invariant**.

The idea is that credit is stored as "potential" in the data structure. Since the potential function and the charging scheme are defined independently of each other, the truth of the invariant (3) 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 simple result:

Theorem 1 (Justification of Amortized Cost)
Assuming the credit-potential invariant (3), the total cost is upper bounded by the total charge:

$$\sum_{i=1}^{n} \textsc{Cost}(p_i) \leq \sum_{i=1}^{n} \textsc{Charge}(p_i).$$

Proof.

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

Q.E.D.

When invariant (3) 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 $\Phi(D_n) \geq 0$). We conclude:

Lemma 2 *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*.*

E.g., for our Counter Example, the cost to count to 25, i.e., to reach the counter value $C = (011001)$, is exactly $2(25) - \Phi(C) = 50 - 3 = 47$. You should find this result remarkable.

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. *Get this!*

¶4. Again, 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 taking advantage of the fact that each request has variable cost, but their variability can somehow be smoothed out by sharing cost over a sequence of requests. This sharing of costs can be done in 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 corresponds to $i = 0$, the next most significant position to $i = 1$, 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 total charge is at most

$$n(1 + \frac{1}{2} + \frac{1}{4} + \cdots) \leq 2n.$$

Hence the amortized cost per increment is ≤ 2.

Note that this charging scheme is actually simpler than the potential method, since we charge each operation the *actual* cost of the operation! In other words, the credit of each

operation is 0. Nevertheless, it does not lead to the the exact cost (Lemma 2) of the potential method. 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 number of work units to increment a counter from 68 to 125 is (exactly!) 110. ♢

Exercise 1.2: Recall that our model of cost for incrementing a binary counter.
(a) What is the *exact* number of work units to count from 0 to 9?
(b) What is the *exact* number of work units to count from 0 to n?
(c) What is the *exact* number of work units to count from m to n ($m \leq n$)? ♢

Exercise 1.3: We consider the incrementing of k-ary counters. The cost model is this: changing a k-ary digit costs 1 unit.
(a) Give an amortized analysis for incrementing a 3-ary counter, concluding that the amortized cost is $O(1)$ per operation. Can you make the analysis tight?
(b) Generalize to the incrementing of a k-ary counter for any $k \geq 2$. Your amortized cost should take k into account. ♢

Exercise 1.4: In a ternary counter, we have 'trits' $(0, 1, 2)$ instead of 'bits'. Changing each trit costs 1 unit. What is the exact cost when I increment my ternary counter from 0 to 83? HINT: partial credit if you only get an *upper bound* on the cost. ♢

Exercise 1.5: 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: $S.\texttt{push}(K)$ simply add the key K to the top of the current stack. But $S.\texttt{pop}(K)$ 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 1.6: 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 "linked" to the output. Here is where the linked list representation of counters is exploited. After this operation, C has the value $11,1010,0011$ and C' has the value 0.

You couldn't do this with arrays!

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 1.7: 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 1.8: Generalize the previous exercise by assuming that the counters need not be initially zero, but may contain powers of 2. ♢

Exercise 1.9: 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 1.10: 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 [12] is a practical approach to implementing all ADT operations listed in §III.2. Splay trees are just ordinary binary search trees – there are no structural requirements (cf. AVL trees) on splay tree. For instance, it could be a tree consisting of a single path (effectively a list), as shown in the leftmost tree in Figure 1. What distinguishes them are the special splay-tree algorithms used to implement standard binary tree operations such as lookup, insert and delete. These algorithms invariably contain a

special **splay** operation. Like rotations in Chapter III, its role is purely to restructure a binary tree.

The splay 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 various 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 *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 by moving the item to the front: the cost of this (preemptive) move has already been paid for (by "charging" the cost to the cost of looking up the item). Thus, this move-to-front heuristic can be justified by amortization arguments; alternatively, one can also use probabilistic analysis (see §10).

The analogue of the move-to-front heuristic in maintaining binary search trees is this: after accessing a key K in a tree T, we must move the node containing K to the root. This can be done by rotations, of course. What if we looked up K, and it is not in T? It is still essential to perform this heuristic (otherwise, it is clearly impossible to achieve a sublinear-time amortized bound). In this case, we move the successor or predecessor of K 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. Let the tree T' be the result of splaying T at a key K:

$$\texttt{splay}(\textit{Key } K, \textit{Tree } T) \to T' \tag{4}$$

Then T and T' are equivalent binary search trees such that the key K' at the root of T' is *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. If T is non-empty, then any key K must have either a successor or predecessor in T, and thus the splay operation is well-defined on non-empty binary search trees. For example, starting from the BST in Figure 1(a), splaying the key $K = 4$ yields the BST in Figure 1(c).

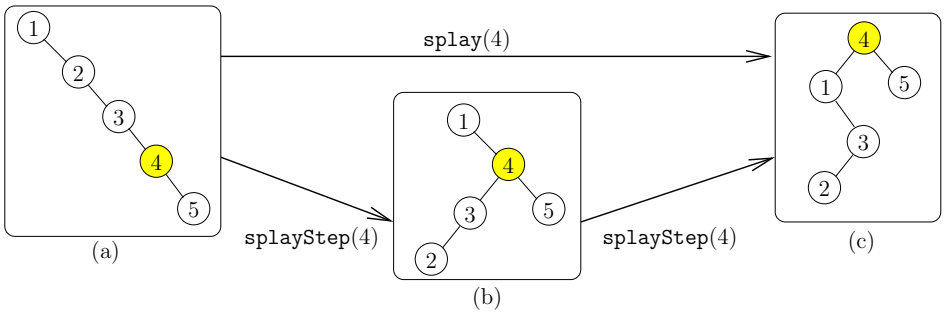


Figure 1: Splaying key 4 (with intermediate step)

So far, we have described $\texttt{splay}(K, T)$ where K is a key. We describe a variant of `splay` where K is replaced by a node u in T: $\texttt{splay}(u, T)$. The operation of $\texttt{splay}(u, T)$ is to bring the node u to the root of T by repeated rotation; it returns the new tree T'. Then $\texttt{splay}(K, T)$ is reduced to its variant as follows:

Two variants: $\texttt{splay}(K, T)$ *versus* $\texttt{splay}(u, T)$

```
splay(K,T):
    u ← lookUp(K, T)
    Return splay(u, T)
```

Recall that `lookUp`(K, T) to find a node u in which u.`key` is the successor or predecessor of K in T. Before describing how `splay`(u, T) is implemented, we show how the splay operation is used.

¶5. Reduction of ADT operations to Splaying. We now implement the operations of the fully mergeable dictionary ADT (§III.2). The implementation is quite simple: each ADT operation is reducible to one or two splaying operations plus some easy $O(1)$ operations.

- `lookUp`(*Key* K, *Tree* T): first perform `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. Note that the tree T has been modified by splaying. Compared to the `lookUp` algorithms in Chapter III, this behavior is new!

- `insert`(*Item* X, *Tree* T): perform the standard binary search tree insertion of X into T. Note that this insertion would not be successful if X.`key` is already in T. In any case, the insertion process yields a node u whose key is equal to X.`key` (this key may either be newly inserted or was already there). Now splay this node u using our second variant of splaying, `splay`(u, T).

  Version 2: we could also first `splay`$(X$.`key`$, T)$. Let K be the key at the root after this splay. If $K = X$.`key`, then we cannot insert X. Otherwise, insert X as the right or left-child of the root, depending on whether $K < X$.`key` or $K > X$.`key`. See Exercises.

- `merge`(*Tree* $T_1, T_2) \to T$: recall that each key 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. This is illustrated in Figure 2.

  Version 2: we can first do the standard deletion algorithm. This deletion identifies a node u whose child has been "cut" (removed). We then do `splay`(u, T).

- `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'.

It is interesting to compare the above ADT implementations with our treatment of Binary Search Trees in Chapter 3: there, we reduce all ADT operations to a single operation, rotation. Now, all operations are reduced to splaying. We shall see that splaying itself is reduced to rotations.

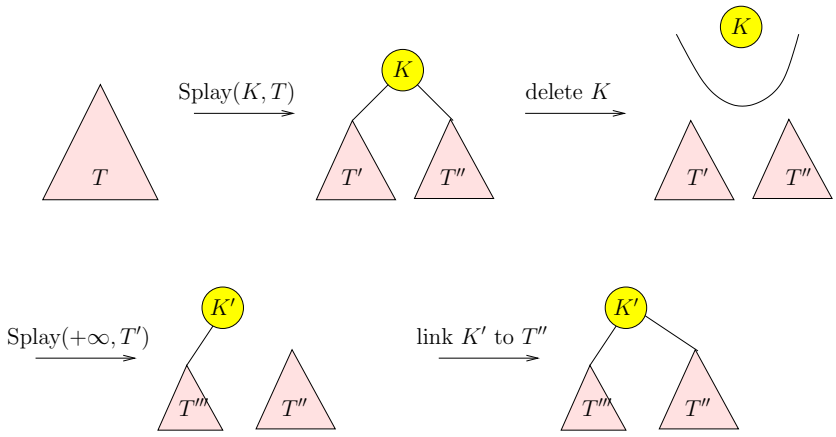


Figure 2: Steps in `delete`(K,T)

¶6. Reduction to SplayStep. Let u be a node in a splay tree T. The operation `splay`(u,T) amounts to repeated application of a single operation

`splayStep`(u)

until u becomes the root of T. Termination is guaranteed if we insist that `splayStep`(u) always reduce the depth of any non-root u. There is an obvious interpretation of `splayStep`(u): it is simply "`rotate`(u)". Call this "naive splaying". Naive splaying does ensure the correctness of our splay algorithms in ¶5. The problem is that it is insufficient for achieving our amortization goal, which is to show that *each of the operations in ¶5 has an amortized cost of* $O(\log n)$.

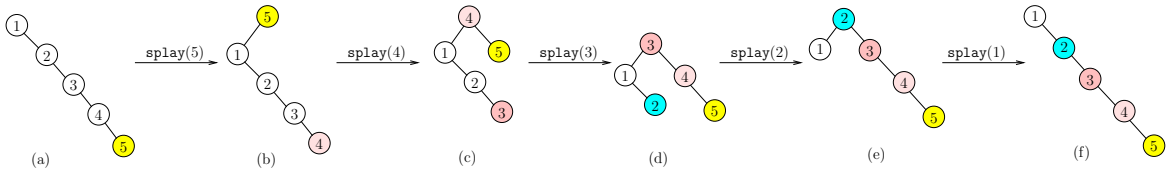


Figure 3: Naive splaying on T_5

The trouble with naive splaying. Consider the binary tree T_n that contains the keys $1,2,\ldots,n$ and whose shape is just a right-path. The case T_5 is illustrated in Figure 3(a). It is easy to verify that we can get T_n by inserting the keys in the following order:

$$n, n-1, n-2, \ldots, 2, 1.$$

The insertion algorithm follows the prescription in ¶5: first do standard insertion and then splay the inserted node.
Next, let us perform a sequence of Lookups on T_n: first do `lookUp`(n,T_n). Using our splay-based lookup algorithm, we first `splay`(n,T_n), which produces a tree rooted at n with a left child that is T_{n-1}. This is illustrated in Figure 3(b). We can repeat this process by performing lookups on $n-1$, $n-2,\ldots,1$. As seen in Figure 3(f), the final result is again T_n. The cost for this sequence of operations is Θ-order of $\sum_{i=2}^{n} i = \Theta(n^2)$. Therefore it is impossible to achieve an amortized cost of $O(\log n)$ on each operation in this Lookup sequence.

The preceding example of naive splaying shows that the correct implementation of `splayStep` is quite subtle. We need a terminology: A grandchild u of a node v is called an **outer grandchild** if u is the left child of the left child of v, or u is the right child of the right child of v. If a node has a grandparent and is not an outer grandchild, then it is an **inner grandchild**.

`splayStep`(*Node* u):
There are three cases.
Base Case. If u has no grandparent,
then `rotate`(u) (see Figure 9).
Case I. Else, if u is an outer grandchild,
do `rotate`(u.`parent`), followed by `rotate`(u). See Figure 4.
Case II. Else, u is an inner grandchild and
we do a double rotation (`rotate`(u) twice). See Figure 4.

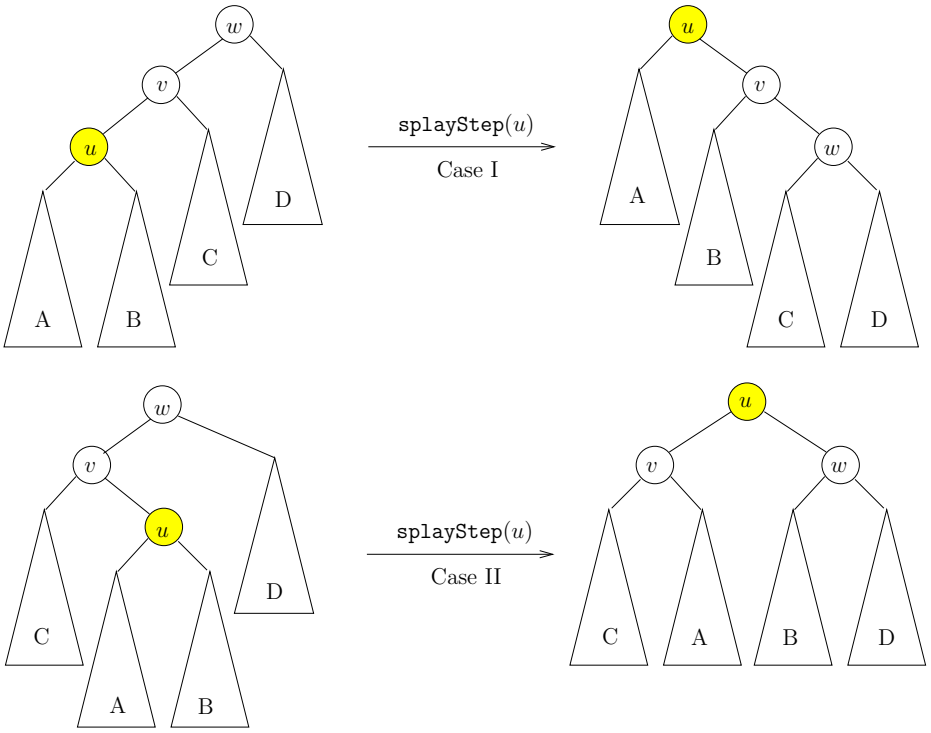


Figure 4: SplayStep at u: Cases I and II.

Note that the Base Case occurs when u or u.`parent` is the root. If u is the root, `rotate`(u) is a no-op; otherwise, `rotate`(u) brings u to the root. 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 $\lceil h/2 \rceil$ `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 5. 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`$(6, T_2)$?

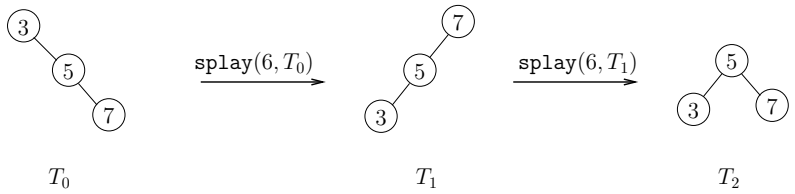


Figure 5: Splaying may return successor or predecessor

¶7. 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 successor or predecessor of K up to the root. Instead, we combine the 2 phases by bringing 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 to illustrate 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, we simply rotate u_L or u_R to the root! For instance, if $u.Key > K$, then we must search for K in the subtree rooted at u_L, and so we do the rotation `rotate`(u_L) to bring u_L to the root. So our initial idea amounts to a repeated left- or right-rotation at the root. Unfortunately, this simple approach has a pitfall. 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). *Find the pitfall!*

- 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_L.`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. Refer to Figure 6(a). 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

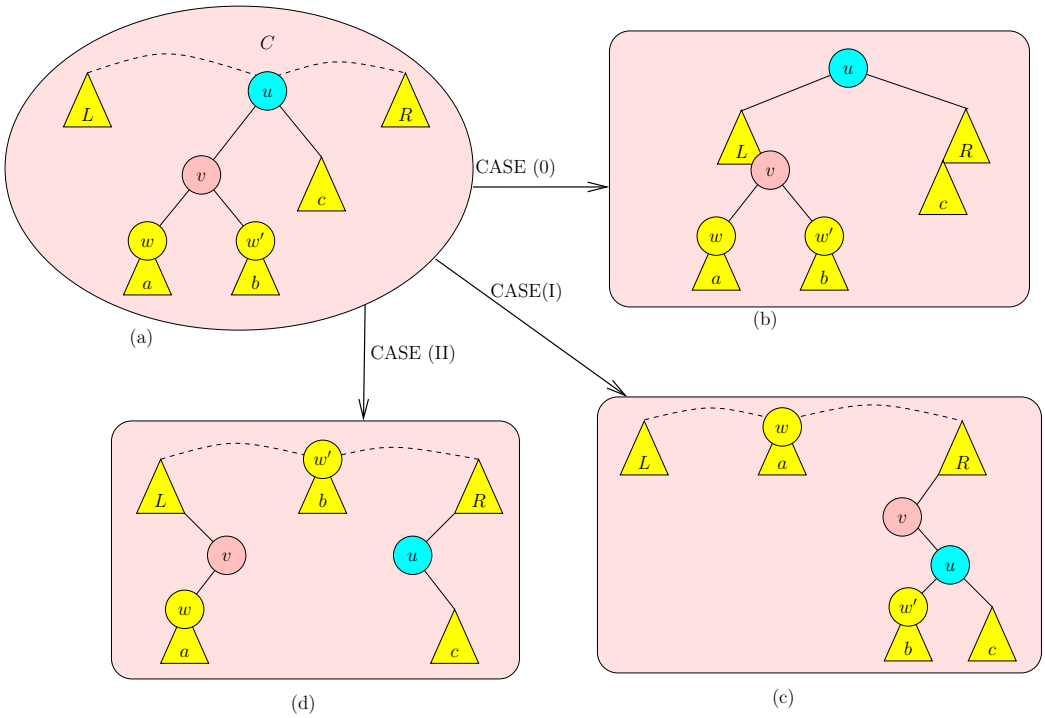


Figure 6: Top Splay: Cases 0, I and II

R, resp. We then make u the root of a tree with left and right children L and R. See Figure 6(b). 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 the subtree rooted at 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 6(c).

- CASE (II): Suppose key K is found in the subtree rooted at 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 6(d).

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

Exercise 2.1: Comment on the following assertions:
(a) Every splay tree is a binary search tree.
(b) Every binary search tree is a splay tree. ♢

Exercise 2.2: Let T be a binary tree with nodes $u_1, \ldots, u_5$ where u_1 is root, u_{i+1} is the child of u_i. However, u_{i+1} is a left-child if i is odd, and a right-child if i is even.
(a) Attach the five keys $1, 2, 3, 4, 5$ to the nodes in T so that the result is a splay tree. Draw the resulting tree, denoted T_a.

(b) Splay 3 in T_a and show the resulting tree, denoted T_b.
(c) Insert 6 in T_a and show the resulting T_c.
(d) Insert 3.5 in T_a and show the resulting T_d.
(e) Delete 3 in T_a and show the resulting T_e.
(f) Delete 2 in T_a and show the resulting T_f. ♢

Exercise 2.3: In this question, we ask which BSTs can arise from splay operations. Define an **IL splay tree** to be one that can be obtained by a sequence of insertions (I) and lookups (L) into an initially empty splay tree. Note that we omit deletes (D). If we include deletes, we have an **ILD splay tree**. Remember that it is the **shape** of a BST that matters, and there is no need to indicate explicit keys in your BSTs. Furthermore, *assume that insertion and deletion is implemented by first doing the standard BST insertion/deletion, followed by a splay.* This assumption will facilitate our enumeration of ILD shapes: For instance, given any shape of size n, it is easy to see all the shapes of size $n+1$ that can result from an insertion: just add a leaf, and splay from that leaf. There are $n+1$ ways to insert this leaf.
(a) Show that not every BST on 3 nodes is an IL splay tree.
(b) Show that every BST on 3 nodes is an ILD splay tree.
(c) Show that any BST on 4 nodes is an IL splay tree.
(d) Show that any BST on 5 nodes is an IL splay tree.
HINT: Furthermore, it is sufficient to enumerate shapes up to left-right symmetry (e.g., for BSTs with 3 nodes, there are only distinct 3 shapes, up to left-right symmetry). Moreover, when we insert into a given
(e) Prove or disprove: every BST on $n > 5$ nodes is an IL splay tree. ♢

Exercise 2.4: In the text, we splay the sequence of keys $n, n-1, n-2, \ldots, 1$ in the tree T_n to show that for naive splaying is not amortized $O(\log n)$. Please re-do this example using regular splaying. First illustrates what happens for $n = 5$ and $n = 6$. What is the end result after doing the sequence of lookups $n, n-1, \ldots, 2, 1$ on T_n. ♢

Exercise 2.5: 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 2.6: Show the result of $\texttt{merge}(T_1, T_2)$ where T_1, T_2 are the splay trees shown in Figure 7. ♢

Exercise 2.7: 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_k be the splay tree after inserting k (for $k = 1, -1, 2, -2$, etc).
(i) Show T_n and T_{-n} for $n = 1, 2, 3$.
(ii) State and prove a conjecture about the shape of T_n. ♢

Exercise 2.8: 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 splay tree after the nth insertion (so T_0 is

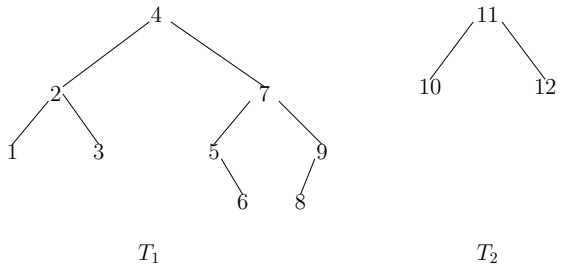


Figure 7: Splay trees T_1, T_2

the empty tree).
(a) Show T_{3n} for $n = 1, 2, 3, 4$.
(b) State and prove a conjecture about the shape of T_n.
(a', b') Repeat (a) and (b), but use the following sequence of input keys

$$2, 3, 1; 5, 6, 4; 8, 9, 7; \ldots; 3n-1, 3n, 3n-2.$$

♢

Exercise 2.9: Consider the insertion of the following sequence of keys into an initially empty tree: $2, 3, 1; 5, 6, 4; 8, 9, 7; \ldots; 3n-1, 3n, 3n-2$, where we have written the keys in groups of three to indicate the pattern. Let T_n be the splay tree after the nth insertion (so T_0 is the empty tree).
(a) Show T_{3n} for $n = 1, 2, 3, 4$.
(b) State and prove a conjecture about the shape of T_n. ♢

Exercise 2.10: (Open ended) Prove that if we have any "regular" sequence of insertions, as in the previous exercises, 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 2.11: Fix a key K and a splay tree T_0. Let $T_{i+1} \leftarrow \texttt{splay}(K, T_i)$ for $i = 0, 1, \ldots$.
(a) Under what conditions will the T_i's stabilize (become a constant)? How many splays will bring on the stable state?
(b) Under what conditions will the T_i's not stabilize? How many splays will bring on this condition? ♢

Exercise 2.12: 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 2.13: In our operations on splay trees, we usually begin by performing a splay. This was not the case with our insertion algorithm in the text. But consider the following

variant insertion algorithm. To insert 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 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 8.
(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 2.14: As in the previous Exercise, carry out the details of the variant deletion algorithm noted in the text. ♢

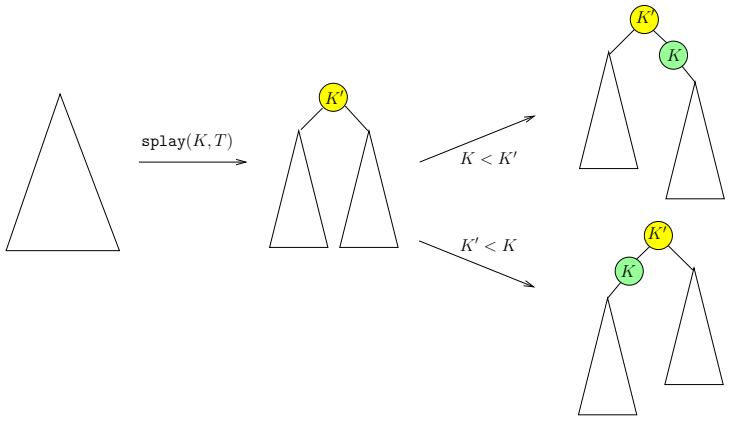


Figure 8: Alternative method to insert a key K.

Exercise 2.15: (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:

```
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
```

♢

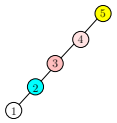
End Exercises

§3. Splay Analysis

Our main goal is to prove:

Theorem 3 (Splay Theorem) *The amortized cost of each splay operation is $O(\log n)$ assuming at most n items in the tree and we started out with an empty tree.*

Before proving this result, let us show that it is a "true" amortization bound. More precisely, we show that the worst case bound is linear, not logarithmic. To see this, consider the repeated



Left-path L_n

insertion of the keys $1, 2, 3, \ldots, n$ (in this order) into an initially empty tree. It is easy to see by induction that the result is a left-path L_n of length $n-1$. Finally, if we perform a lookup on key 1 in this tree, we must expend $\Theta(n)$ units of work.

¶8. Proof of Splay Theorem. 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. 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. By definition, a tree with 0 or 1 node has no potential, $\Phi(T) = 0$.

Lemma 4 (Key Lemma) *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*

$$3(\Phi'(u) - \Phi(u)) \tag{5}$$

units of work in cases I and II. In the base case, we charge one extra unit, in addition to the charge (5).

Theorem 3 follows easily from the 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 at most

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

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 is proportional to the overall cost of splaying. This only increases the constant factor in our charging scheme. This concludes the proof of the main goal.

¶9. Proof of Key Lemma. The following is a useful remark about rotations:

Lemma 5 *Let Φ be the potential function before a rotation at u and Φ' the potential function after. Then the increase in potential of the overall data structure is at most*

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

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

Proof. We refer to Figure 9. The increase in potential is

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

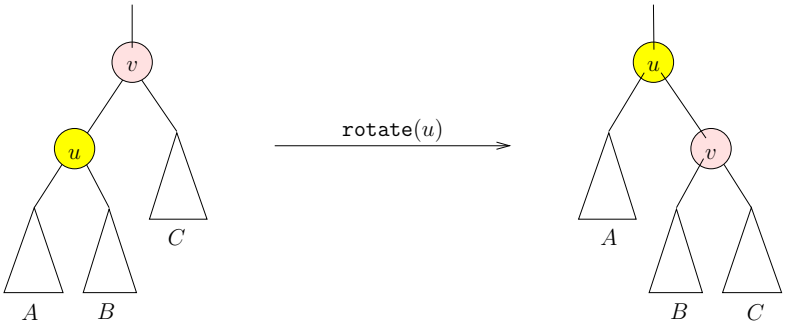


Figure 9: Rotation at u.

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

Proof of Key Lemma. The Base Case is almost immediate from lemma 5: 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 4 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

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

a contradiction. Here, $\textsc{Size}'$ denotes the size after the splay step operation. This proves the Key Lemma.

All our splay tree operations are ultimately reduced to splay operation, and some simple primitive operations. The Key Lemma bounds the potential change in the splay operation. But some primitive operations change the potential of our splay trees via the deletion or insertion of a leaf: our next lemma bounds such potential changes.

Lemma 6 *Let T^+ be a binary tree with $n + 1$ items, and T is obtained from T^+ by deleting some leaf x. Alternatively, T^+ can be viewed as the result of inserting a new leaf x into T. 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 $x = u_m$. Let n_i be the size of the subtree at u_i in T, and so $n > n_0 > n_1 > \cdots > n_m = 0$. Therefore the potential of u_i in T^+ is $\lfloor \lg(n_i + 1) \rfloor$, and the increase in potential in going from T to T^+ is

$$\Phi(T^+) - \Phi(T) = \sum_{i=0}^{m-1} \lfloor \lg(n_i + 1) \rfloor - \lfloor \lg(n_i) \rfloor .$$

Observe that $\lfloor \lg(n_i + 1) \rfloor - \lfloor \lg(n_i) \rfloor$ is either 0 or 1. Moreover, it is 1 iff $n_i + 1$ is a power of 2. There are at most $\lg n$ values of i for which it is equal to 1. Hence

$$\Phi(T^+) - \Phi(T) \leq \lg n.$$

Q.E.D.

¶10. Amortized Cost of Splay Implementation of ADT Operations.

We conclude with an amortized cost statement for splay trees.

Theorem 7 *Starting from an initially empty splay tree, any sequence of m requests of the types*

`lookUp`, `insert`, *`merge`*, `delete`, `deleteMin`, *`split`*,

and involving a total of n items, has total time complexity of $O(m \log n)$. Thus, the amortized cost is $O(\log n)$ per request.

Proof. This follows almost immediately from Theorem 3 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 two important details 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 Lemma 6, this increase is at most $\lg n$.

(B) When merging two trees T_1, T_2. In this case, we first perform $\texttt{splay}(+\infty, T_1)$. 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 at most $1+\lg(|T_1|+|T_2|)-\lg(|T_1|) = 1+\lg(1+|T_2|/|T_1|) < 1+\lg n$. 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 [11] 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.

¶11. Application: Splaysort Clearly we can obtain a sorting algorithm by repeated insertions into a splay tree, followed by repeated deleteMins — this is analogous to the heapsort algorithm (¶III.7). This algorithm is known as **splaysort**. It is not only theoretically optimal with $O(n \log n)$ complexity, but has been shown to be quite practical [7]. Splaysort has the ability to take advantage of "presortedness" in the input sequence. For instance, running splaysort on the input sequence $x_1 > x_2 > \cdots > x_n$ will take only $O(n)$ time in the worst case. 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. However, if $x_1 < x_2 < \cdots < x_n$ is our input, the advantage of presortedness is lost for our version of splaysort. The Exercises discuss some ways to overcome this.

Quicksort (Lectures II and VIII) is regarded as one of the fastest sorting algorithms in practice. But algorithms like splaysort may run faster than Quicksort for "well presorted" inputs. Quicksort, by its very nature, deliberately destroy any property 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 $\Phi(u) = \lfloor \lg(\text{Size}(u)) \rfloor$. In other words, we avoid the floor function to make the potential function continuous.
(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) Consider 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: In the previous question, we computed a constant $K > 0$ such that can pay for our amortization. Optimize this constant. ♢

End Exercises

§4. Application to Dynamic String Encoding

We now present an application of splay trees to the dynamic Huffman code problem in §V.4. We want to design a protocol to transmit any string $X = x_1 x_2 \cdots x_m$ over a set Σ of symbols. The transmission protocol is restricted to making only one pass over the string X, and as each symbol $x_i \in X$ is scanned, the transmission protocol emits a binary sequence. The protocol has state information that depends on sequence of symbols scanned so far, and its binary sequence emitted for each symbol depends on this state. Thus the emitted sequence for x_i $(i = 1, \ldots, m)$ may be denoted by "$e(x_1 x_2 \cdots x_i)$", not "$e(x_i)$". Writing $X_i = x_1 x_2 \cdots x_i$ for the prefix of length i, the output of this protocol on input X is the binary string

$$E(X) := e(X_1)e(X_2)\cdots e(X_m).$$

The receiver protocol, on receiving $e(X_i)$ should be able to reconstruct x_i for each i. What does it mean to "reconstruct the symbol x_i"? Computationally, we must ultimately reduce each symbol in Σ to a string over a fixed alphabet, say $\{0, 1\}$. A simple solution is to identify Σ with some subset of $\{0, 1\}^N$ (bit strings of length N). In reality, Σ is some abstract set with many conventional properties that we would like to exploit in our discussions. For this reason, it is useful not to identify Σ with any set of bit strings, but introduce an independent injective function

$$\text{A}SC : \Sigma \to \{0, 1\}^N \tag{6}$$

called the **standard encoding** of Σ. Any set Σ with such an encoding is called a **character set**. Each element $x \in \Sigma$ is called a **character** and $\text{ASC}(x)$ is the **standard code** of x. In discourse, we treat x and $\text{ASC}(x)$ interchangeably. Since there is a natural sorting order on $\{0, 1\}^N$, this induces the standard sorting order on the character set Σ. In our protocols, the transmitter and receiver know this standard encoding, and this sorting order.

The ASCII Character Set. Our prime example of Σ is the (extended) **ASCII set**. The standard encoding (6) here will be denoted A$SCII$, and N is equal to 8. Thus, ASCII(`x`) is the ASCII code for x. It can be written as a 8-bit string, or as a sequence of 2 hexadecimal digits. Generally, the latter is preferred. Our hexadecimal strings are prefixed by '0x' followed by hexadecimal digits: $0, 1, \ldots, 9, A, B, \ldots, F$. Thus, the ASCII code for the character `A` is $(01000001)_2$ in binary or $0x41$ in hexadecimal notation. Again, ASCII(`Z`) $= 0x5A$. See §V.5 for the full table of ASCII codes. Note that the character NUL with the ASCII code $0x00$ is the first ASCII character in the standard sorting order. But in our examples, we use $*$ as the first character in Σ (following conventions in Huffman coding).

A naive protocol is to just emit the N bits ASC(x) for each character $x \in \Sigma$. But the goal of dynamic string encoding is to design a protocol that can dynamically change the encoding as the transmission progresses. For instance, we would like a protocol that could transmit $X =$ `aaaaa` much more efficiently than $X =$ `abcde`. We now present a simple online protocol based on splay trees. The online feature is important when X represents a very large file. Using the splay tree protocol, we can transmit $X = \underbrace{\texttt{aaa}\cdots\texttt{aa}}_{n}$, we only transmit ASC($a$)$\underbrace{\texttt{111}\cdots\texttt{11}}_{n-1}$, using only $7 + n$ bits instead of $8n$ bits. As $n \to \infty$, this transmission rate is clearly optimal, and amounts to a factor of 8 data compression rate.

¶12. External Splay Trees. We cannot use our usual splay trees which are based on ordinary BSTs. External BSTs turns out to be essential: recall (¶III.32) that items are stored only in the the leaves of external BSTs. These leaves are called external nodes. Except for the minimum external node, there is a 1-1 correspondence that **partners** each external node to its predecessor which is an internal node that holds the same key as its partner. We have already discussed the insertion and deletion algorithms for external BST. Now, we must add the splay feature to these algorithms: after each lookup, insertion or deletion, we must splay the parent of the external node where lookup/insertion/deletion occurred. The resulting data structure is now called **external splay trees**. Observe that we only splay internal nodes, never an external node. We remark that *rotation can be freely performed on the internal nodes of an external BST*. This is because the result remains an external BST because the predecessor and successor relation of nodes is preserved.

Why we splay external nodes?

We use an external splay tree T to represent store a dynamic subset S of Σ. Typically, S is a much smaller set than Σ. We view T as a prefix-free code for S: the path from the root of T to a character $x \in S$ can be read as a binary string $e(x)$ where 0 indicates a left child and 1 a right child. Recall that Σ has a sorting order, and we will exploit this sorting order in order to interpret T as a BST. Given a particular character $x \in \Sigma$, we can use x as a key for LookUp. If x is not found, we can insert it into T. In reality, each x is represented by its standard encoding ASC(x) in T.

As in dynamic Huffman codes, we require T to have a leaf representing a special character $*$ that is not a character in S. So the smallest possible external splay tree has one node (which is both root and leaf) storing $*$. Call this the **$*$-tree**. It is convenient to assume that $*$ is least character in Σ. Strictly speaking, it is not even necessary to assume $*$ is a character in Σ at all.

¶13. The Protocol We have two parties, the **transmitter** and the **receiver**. The transmitter has a string $X \in \Sigma^*$ which is to be transmitted as a bit stream $E(X) \in \{0, 1\}^*$. Both

parties know the standard encoding ASC (see (6)).

- Each party maintains an external splay tree: T for the transmitter and R for the receiver. The idea is for both parties to keep T and R synchronized. Initially, T and R are both the trivial $*$-tree. Let $X = x_1 x_2 \ldots x_m$. After the transmitter has transmitted the first i characters of X, the splay tree T will have $1 + |\{x_1, x_2, \ldots, x_i\}|$ external nodes, each storing a character in the set $\{x_1, \ldots, x_i\}$, including the special character $*$.
- TRANSMITTER PROTOCOL: To transmit the character $i+1$st character, the transmitter performs a LookUp in T using the key x_{i+1}. As noted above, in reality, the string $\text{ASC}(x_{i+1})$ is probably used for x_{i+1}. The LookUp will end in some external node b. We have two cases: the easy case if when $b.\texttt{key} = x_{i+1}$, then simply emit the string $e(X_{i+1})$ which encodes the path from the root to b. After this, we must splay the parent of b.

  The hard case is when $b.\texttt{key} \neq x_{i+1}$. We first emit the string $E(*)$. Note that $E(*)$ is just a string of 0's that can be emitted as follows: begin at the root of T. As long as the current node is not a leaf, emit a 0 and go to the left child. When we reach the leftmost leaf containing $*$, we must splay the parent of the $*$-node, and also emit $\text{ASC}(x_{i+1})$. Then we insert x_{i+1} into the tree and splay b (which is be the parent of x_{i+1}). Let us explicitly describe this insertion process: there is no need to do a LookUp, since we know x_{i+1} will be inserted as a child of b. So we simply turn b into an internal node by creating two children for b. These two children will contain x_{i+1} and also the character that was originally at b. The `left` and `right` links at these two children must also be updated to reflect an external splay tree. Finally, we must splay the node b.
- RECEIVER PROTOCOL: In each round, the receiver is awaiting a bit stream that represents a path from the root of R to an external node. For the first round, this path is the empty string, and we expect the first N bits to be $\text{ASC}(x_1)$. For the subsequent rounds, we will receive a non-trivial path which leads to a leaf b in R. There are two cases: in the easy case, b is a non-$*$ node. The protocol will splay the parent of b and output $b.\texttt{key}$. This finishes the round. In the hard case, b is the $*$-node. We must first splay the parent of the $*$-node. The next character x is new, and will arrive in the next N bits as $\text{ASC}(x)$. The protocol then inserts x (i.e., $\text{ASC}(x)$) into R, and splay the parent of the newly inserted external node.

The above protocol ignores additional "handshaking" conventions between the transmitter and receiver: both sides must agree on the start of the transmission, and have some convention to indicate the end of transmission.

Let us see what this protocol will transmit in case $X = \underbrace{\texttt{aaa}\cdots\texttt{aa}}_{n}$. At the beginning, a LookUp on the first character a in X will fail, and the path to the $*$-node is the empty string. So the transmitter just transmits $\text{ASC}(\texttt{a})$. But each subsequent occurrence of $\text{ASC}(\texttt{a})$ in X will cause a single binary bit 1 to be transmitted. This confirms our earlier remark that $E(\underbrace{\texttt{aaa}\cdots\texttt{aa}}_{n}) = \text{ASC}(\texttt{a})\,\underbrace{11\cdots11}_{n-1}$.

Let us show the sequence of trees that results from insertion of consecutive characters from our famous 12-character string: `hello␣world!` Here is[4] the sorting order we use in this example:

$$\texttt{*} < \texttt{␣} < \texttt{!} < \texttt{a} < \texttt{b} < \texttt{c} < \cdots < \texttt{x} < \texttt{y} < \texttt{z}.$$

[4]This is almost the sorting order if these characters are encoded in ASCII, except that you must interpret the character "`*`" as the NUL character with ASCII code $0x00$. The actual ASCII codes are $\text{ASCII}(*) = 0x2A$, $\text{ASCII}(\texttt{␣}) = 0x20$, $\text{ASCII}(\texttt{!}) = 0x21$, $\text{ASCII}(\texttt{a}) = 0x61$, $\text{ASCII}(\texttt{b}) = 0x62, \ldots$ and $\text{ASCII}(\texttt{z}) = 0x71$.

Initially, we have the $*$-tree.

¶14. Analysis. To bound the length of the transmitted string $E(X)$, we exploit the key result about splay trees, *viz.*, each splay operation has amortized logarithmic time.

Lemma 8 *If $X = x_1 x_2 \cdots x_m$ has n distinct characters, then the total cost of transmitting X is $O((m+n)\log n)$. Hence the length of $E(X)$ is $O((m+n)\log n)$.*

Proof. To transmit X, we perform m LookUps; in n of these LookUps, we must also insert a new leaf and splay the $*$-node. This results in $m + 2n$ splays, each with an amortized cost of $O(\log n)$. Also, when we attach a new leaf to a splay tree, we increase the potential of the tree by $O(\log n)$ (Lemma 6). This yields the $O((m+n)\log n)$ total cost. Clearly, the length of $E(X)$ cannot exceed the cost to compute $E(X)$. **Q.E.D.**

Exercises

Exercise 4.1: What will be transmitted by our splay tree algorithm for the string $a^n b^n c^n$ where a, b, c are distinct characters? ♢

Exercise 4.2: Consider the transmission of the string "hello world!" using the Dynamic Huffman tree algorithm and the splay tree algorithm.
(a) First describe the bit string that is transmitted using the Dynamic Huffman tree algorithm. Do not just give the bit string, but summarize the processes that goes into the production of each substring.
(b) Do the same, but using the Splay tree approach.
(c) Draw some conclusion about the relative merits of the two methods. ♢

Exercise 4.3: Our protocol above assumes a setting where both parties knows when the transmission has begun, and when it has stopped. Design a hand-shaking protocol that takes care of this detail. Make any reasonable assumptions you need. ♢

Exercise 4.4: We want to transmit an ASCII string X using our external splay tree protocol. Here $|X| = 3n$. As $n \to \infty$, how how many bits per characters are transmit by our protocol in the following scenarios:
(a) $X = \text{aabaab} \cdots \text{aab} = (\text{aab})^n$
(b) $X = X_1 X_2 \cdots X_n$ where each $X_i \in \{aaa, bbb\}$?
(c) $X = X_1 X_2 \cdots X_n$ where each $X_i \in \{aaa, bbb, aba, bab\}$? ♢

Exercise 4.5: (a) Show the result of inserting the following characters, `h, e, l, o, w, r, d` (in this order) into an external splay tree that initially contains the character `*`. Assume the characters have the usual alphabetic sorting order but `*` is smaller than any other character. REMARK: it suffices to show the external splay tree after each insertion.
(b) Show the tree after deleting `w` from the final tree in part(a). ♢

Exercise 4.6: Please use our external splay tree to transmit the string `Hello World!` using the standard sorting order of the ASCII characters. How many bits are transmitted? ♢

Exercise 4.7:
(a) Program the above transmission and receiving protocols based on external splay trees in your favorite programming language.
(b) Do the same for the Dynamic Huffman tree protocol (§V.4).
(c) Compare the performance of the splay tree protocol with that of the Dynamic Huffman Tree for Lincoln's speech at Gettysburg (Exercise, §V.4). ♢

Exercise 4.8: Here is an ASCII string $X = b^m a^n c^p$ where m, n, p are positive integers. So $|X| = m + n + p$. The ASCII ordering on the characters is $a < b < c$ (of course). Recall our protocol for transmitting strings using External Splay trees.
(a) If $m = 10, n = 2, p = 20$, what is the length of $E(X)$? The answer is a single number, but explain how you get this number.
(b) TRUE or FALSE: there exist constants $\alpha, \beta, \gamma, \delta$ such that the length of $E(X)$ is equal to

$$|E(X)| = \alpha m + \beta n + \gamma p + \delta$$

♢

Exercise 4.9: We want a dynamic transmission protocol based on external splay trees. Suppose the string X to be transmitted is 'bilingual'. That is, X is a string over two character sets Σ_A and Σ_B, with standard encodings $\mathrm{ASC}_\mu : \Sigma_\mu \to \{0,1\}^{N_i}$ ($\mu = A$ or $\mu = B$) for each character set.
(a) Give a solution to this problem using external splay trees.
(b) Suppose there are n_μ ($\mu = A, B$) characters from Σ_μ in X (so $|X| = n_A + n_B$). Also the number of distinct characters among the n_μ characters is s_μ (so $s_\mu \leq n_\mu$). Show that the transmitted string has length $O(s_A \log n_A + s_B \log n_B)$ ♢

Exercise 4.10: What is the "worst case" behavior of our external splay tree protocol? One notion of worst case is when $E(X_i)$ has n bits when the tree T has $n+1$ external nodes. Can you force this to happen arbitrarily often? ♢

Exercise 4.11: In our protocol, we do not use deletion in the external splay tree T. To introduce deletion, we introduce an upper bound B on the number of external nodes in T. When we need to insert a new character c' when T already has B external nodes, we need to pick a character c in T to delete. For this purpose, we maintain the "last access time" $LAT(c)$ for each c in T. E.g., if $x_i = c$ then we set $LAT(c) = i$ after we transmit x_i. Then, before we insert c', we first delete the character c whose $LAT(c)$ is minimum. ♢

End Exercises

§5. Application to Convex Hulls

The following application is interesting because it illustrates the idea of an **implicit key BST** (¶III.32) In a standard BST, we store a key $u.Key$ at each BST node u. When Lookup, we are given a key k and we can use the binary search property of the BST to compare k with $u.Key$ and then proceed to the left or right subtree. In our "implicit" scenario, to "Lookup" some object q at a node u, we must compute some predicate "$P(q, u.\texttt{left}.Key, u.\texttt{right}.Key)$" such that if the predicate $P(q, u.\texttt{left}.Key, u.\texttt{right}.Key)$ returns true then we continue the search on $u.\texttt{left}$, otherwise continue to $u.\texttt{right}$. Thus, although $u.Key$ is

¶15. Analysis. 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 the boundary of $CH(X)$ is a convex polygon, we may represent it as a sequence

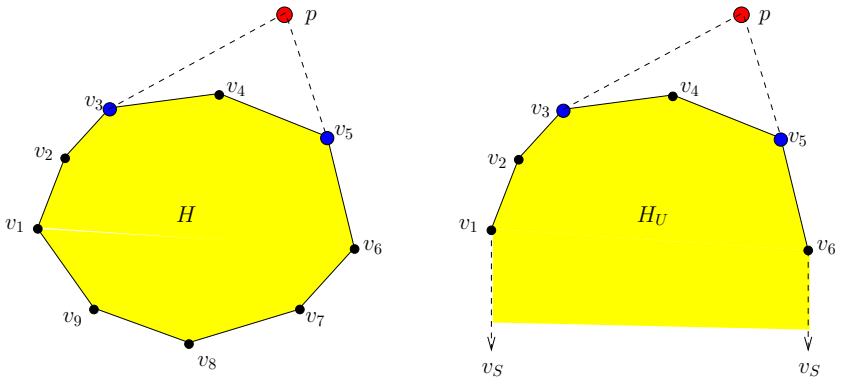
$$H = (v_1, v_2, \ldots, v_m), \quad 1 \leq m \leq n$$

where $v_i \in X$ and these v_i's appear as consecutive vertices in a clockwise traversal of the polygon $CH(X)$. We assume that H is a strictly convex sequence, i.e., no 3 consecutive vertices v_{i-1}, v_i, v_{i+1} are 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 an arbitrary point.

- $\texttt{insert}(p, H)$: this update H to represent $CH(X \cup \{p\})$. Note that if p is inside the current hull, $CH(p, H)$ is a NO-OP.
- $\texttt{tangent}(p, H)$: this requests has two cases. **CASE 1**, p is inside the current hull. Then we return "↑". **CASE 2**, p is outside H. Then return a pair (q, r) of distinct points on H such that the lines $\overline{pq}$ and $\overline{pr}$ are both tangential to H. Call q and r the **tangent points** of H from p. We will specify an ordering on q, r later so that q (resp., r) is the left (resp., right) tangent points. in Figure 10(a), $\texttt{tangent}(p, H)$ returns (v_3, v_5).



(a) (b)

Figure 10: (a) $H = (v_1, \ldots, v_9)$, (b) $H_U = (v_1, \ldots, v_6)$.

¶16. Reduction to Half-Hulls. We may assume that v_1 and v_ℓ $(1 \leq \ell \leq m)$ have (resp.) 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,

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

So 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[5] at infinity (the South and North Poles, respectively). In a certain sense, H_U is the convex hull of $X \cup \{v_S\}$:

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

This can be made precise by a limiting argument. Similarly, H_L is 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 10. 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.

¶17. 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

$$v_1 <_x v_2 <_x \cdots <_x v_\ell, \tag{7}$$

where, in general, we write

$$p <_x q \tag{8}$$

for 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.

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. Let $\texttt{tangent}(p, H_U) = (q, r)$ where $q <_x r$; we call q the **left tangent point** and r the **right tangent point**. Wlog, let $q = v_{i-1}$ and $r = v_{j+1}$); note that $i \leq j + 1$. So 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. Of course, if $i = j + 1$, the subsequence is empty. This replacement can be done efficiently if T is a splay tree that implements the operations of the fully mergeable dictionary ADT (¶5). 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[6] 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. Note that both $\texttt{insert}(p, T)$ and $\texttt{tangent}(p, T)$ must take a common first step to decide if p is outside the upper hull of T or not. If not, then both algorithms return $\uparrow$. So our first task is:

[5] We could define v_S and v_N to be $(r, -\infty)$ and $(r, +\infty)$, respectively, for any finite value r.

[6] Observe that this operation resembles a merge but is not the same as a standard merge. Nevertheless, an amortized cost of $O(\log n)$ can be achieved using our usual arguments.

¶18. Deciding if a point p in outside a upper hull. This calls for an explicit binary search. Our first task is to determine the two consecutive hull vertices v_k, v_{k+1} such that

$$v_k <_x p \leq_x v_{k+1} \qquad (0 \leq k \leq \ell). \tag{9}$$

To do this, we do a lookup on the key $p.x$. Recall that lookup will return u that is either the successor or predecessor of $p.x$ in T. From u, and using the successor and predecessor pointers in T, we can easily determine the k that satisfies (9).

By convention, if $k = 0$ then v_k is undefined, and if $k = \ell$ then v_{k+1} is undefined. Moreover either $k = 0$ or $k = \ell$ implies that p is outside the upper hull. Hence assume that $0 < k < \ell$. Now we conclude that p is outside the upper hull iff p lies above the line $\overline{v_k, v_{k+1}}$.

We now address the problem of computing `tangent`(p, T), given that p lies outside the upper hull of T. We next give two methods for doing this.

¶19. Method One for Tangent Point: Walking Method. The first method is more intuitive. To implement `tangent`(p, T), we have to find the left and right tangent points separately. By symmetry, let us only discuss the left tangent point.

Using the explicit binary search in ¶18, we can decide if p is outside the upper hull. If not, we can return $\uparrow$ since `tangent`$(p, T) = \uparrow$. Otherwise, this procedure gives us the k such that (9) holds. If $k = 0$, we can return $\uparrow$ since the left-tangent is undefined. Otherwise, we continue to search for the left tangent point q. We know that $q = v_{i_0}$ for some $i_0 \leq k$. To find i_0, we use the predecessor pointers to "walk" along the upper hull, starting from v_k to v_{k-1}, v_{k-2}, etc. In general, for any index i with the property that $v_i <_x p$, we can decide whether $i_0 = i$, $i_0 < i$ or $i_0 > i$ according to the following cases:

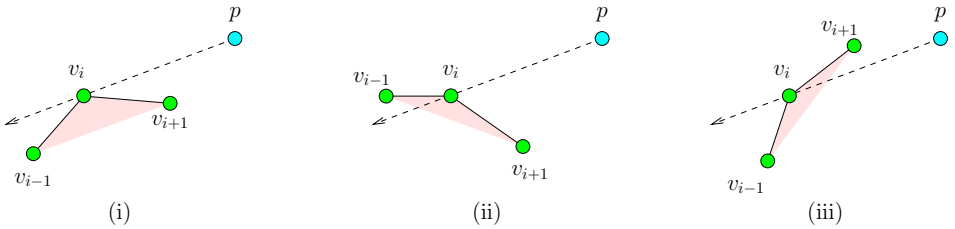


Figure 11: LeftTangent Search from p: (i) $i_0 = i$, (ii) $i_0 < i$, (iii) $i_0 > i$.

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 $i_0 > i$.

These three cases are illustrated in Figure 11. Of course, in our "walk" from v_k to v_{k-1}, v_{k-2}, etc, we will never encounter case (iii). We have ignored degeneracies in these three cases. This will be treated in the Exercises.

¶20. Method Two: Implicit Binary Search. The previous method for finding tangent points (q, r) takes linear time in the worst case. But if we were inserting the point p into the upper hull, then the elements lying strictly between q and r would be deleted. It is then easy to see that, in an amortized sense, this linear search has cost $O(1)$. But if we view the convex hull as a generic structure that supports arbitrary tangent queries, then this linear cost cannot be amortized into a sublinear cost. Our second method is able to overcome this problem and achieve an amortized cost of $O(\lg n)$.

Again, assume we want to find the left-tangent q (if it exists) of a query point p. As before, we call the explicit binary search in ¶18 to determine the k satisfying (9); we may assume that $k > 0$ and p is outside the upper hull (otherwise q does not exist). We now search for q *using another binary search*, not by walking along the upper hull. This search begins at the root of the splay tree T that stores the upper hull, and uses the 3-way decision (Cases (i)-(iii) above) to search for the left tangent point $q = v_{i_0}$. This is an "implicit" binary search because the 3-way decisions are not based on any explicitly stored keys. Indeed, the decisions depend not just on the stored data v_{i-1}, v_i, v_{i+1} but *also* the query point p. We are searching for the index i_0, but we use the data v_{i-1}, v_i, v_{i+1}, p to decide if $i = i_0$, or $i < i_0$ or $i > i_0$.

FindLeftTangentPoint(p, T):
▷ *Explicit Binary Search*
1. Perform an explicit binary search using the key $p.x$.
   This locates the k such that (9) holds.
   We can then determine if p lies inside the upper hull.
   If so, Return($\uparrow$)
▷ *Implicit Binary Search*
   Initialize u to the root of T; let v_i be the vertex at u.
2. Repeat:
   If $v_i \geq_x p$, set $u \leftarrow u$.`leftChild`.
   Else, we have three possibilities given in ¶19:
      If CASE (i) holds, Return(v_i). ◁ *Left tangent found:* $i_0 = i$
      If CASE (ii) holds, set $u \leftarrow u$.`rightChild`. ◁ *Go right:* $i > i_0$
      If CASE (iii) holds, set $u \leftarrow u$.`leftChild`. ◁ *Go left:* $i < i_0$
   Update i to the index of the vertex in u.

¶21. Geometric Primitives. 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[7] **LeftTurn Predicate**. Given any three points p, q, r, define

$$\texttt{LeftTurn}(p, q, r) = \begin{cases} 0 & \text{if the points } p, q, r \text{ are collinear} \\ +1 & \text{if the path } (p, q, r) \text{ make a left turn at } q \\ -1 & \text{if the path } (p, q, r) \text{ make a right turn at } q \end{cases} \tag{10}$$

In an Exercise below, you will see how this is easily implemented as the sign of a certain 3×3 determinant. Note that we call this a "predicate" even though this is a 3-valued function. In logic, predicates are usually 2-valued (i.e., true or false). This is a general phenomenon

[7] Also known as orientation predicate.

in geometry: we might call 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 it is beyond our scope.

We now make the inner loop of the implicit search in the FindLeftTangent algorithm explicit:

```
2. Repeat:
2.1     Let u_0 = u.pred and u_1 = u.succ.      ◁ These may be NULL .
2.2     If (p <_x u)
            If (u_0 = NULL), Return(v_S).       ◁ South Pole
            else u ← u.leftChild
2.3     elif ((u_0 ≠ NULL) and LeftTurn(u_0, u, p) = 1))
            u ← u.leftChild
2.4     elif ((u_1 ≠ NULL) and LeftTurn(u, u_1, p) = −1))
            u ← u.rightChild
2.5     else Return(u).     ◁ This is correct, even if u_0 or u_1 are NULL.
```

To see why the return statement in line 2.5 is correct, first assume u_0 and u_1 are non-null. Then line 2.2 has ensured that $p <_x u$, line 2.3 has verified that (u_0, u, p) is a right-turn and line 2.4 has verified (u, u_1, p) is a left-turn. These verifications depend on non-degeneracy assumptions. The reader should next verify correctness in case u_0 or u_1 are null.

¶22. Inserting a point into an upper hull. It is now easy to implementation $\texttt{insert}(p, T)$: we first check, using the above routine to determine if p is outside the upper hull of T. If so, we may return as $\texttt{insert}(p, T)$ is a no-op. We first perform $\texttt{tangent}(p, T)$ 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 accomplished using the operations of `split` and `merge` on splay trees as described earlier.

We may conclude with the following. Let D be our data structure for the convex hull H. So D is a pair of splay trees representing the upper and lower hulls of H.

Theorem 9
(i) Using the data structure D to represent the convex hull H of a set of points, we can support $\texttt{insert}(p, D)$ *and* $\texttt{tangent}(p, D)$ *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.*

This theorem gave us an $O(n \log n)$ algorithm for computing the convex of a planar point set. Under reasonable models of computation, it can be shown that $O(n \log n)$ is optimal. 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. Note h is a measure of the output size, satisfying $1 \leq h \leq n$. For instance, there is a gift-wrap algorithm for convex hull that takes time $O(hn)$. So gift-wrap is faster than $O(n \log n)$ algorithm when $h = o(\log n)$. But Kirkpatrick and Seidel [5] gave an output-sensitive algorithm whose complexity is $O(n \log h)$.

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 [8] designed a data structure for a fully dynamic convex hull that uses $O(\log^2 n)$ time for insertion and deletion.

There are many applications and generalizations of the convex hull problem. An obvious extension is to ask for the convex hull of a set of points in $\mathbb{R}^d$ ($d \geq 3$). Or we can replace "points" by balls or other geometric objects in $\mathbb{R}^d$. We encourage the student to explore further.

Exercises

Exercise 5.1: What is "convex hull" in 1-dimension? What is the upper and lower hull here? ♢

Exercise 5.2: You need some basic facts about matrices and determinants to do this problem, and to use some physical intuition. Let $a, b, c \in \mathbb{R}^2$ and consider the following 3×3 matrix,

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

(i) Show that $\det M = 0$ iff a, b, c are collinear (i.e., all 3 points lies on some line). HINT: Wlog, assume $a \neq b$. Express c in terms of a and b if c lies on the line through a, b?
(ii) Show that $\det M$ is invariant if you translate all the points, i.e., (a, b, c) is replaced by $(a + t, b + t, c + t)$ for any $t \in \mathbb{R}^2$.
(iii) Show that, up to sign, $\det M$ is the twice the area of the triangle (a, b, c). HINT: from part (ii), you may assume $a = (0, 0)$ is the origin. First do the case where b, c both lie in the first quadrant, and then extend the argument, and do some simple calculations!
(iv) Show that $\det M > 0$ ($\det M < 0$) iff a, b, c list the vertices counter-clockwise (clockwise) about any point in the interior of the triangle (a, b, c).
(v) What is the relation between $\texttt{sign}(\det M))$ and $\texttt{LeftTurn}(a, b, c)$? ♢

Exercise 5.3: Prove the correctness of the FindLeftTangent(p, T) algorithm. ♢

Exercise 5.4: We want to compute the upper hull of an input sequence of points $X = (p_1, \ldots, p_n)$, where we assume that $p_1 <_x p_2 <_x \cdots <_x p_n$. We want to use an array-based approach where an upper hull $U = (v_0, \ldots, v_k)$ with $k + 1$ vertices is represented by an array $U[0..k]$, where $U[i] = v_i$. Let build U incrementally using METHOD ONE in the text, by successive insertion of p_i into the upper hull of $(p_1, \ldots, p_{i-1})$. Since the length of U can change, we must also maintain a variable to represent k. Assume the set of input points is non-degenerate (no 3 points are collinear).
(a) Suppose U currently represents the upper hull of $(p_1, \ldots, p_i)$. Describe in detail how to carry out the operation of finding LeftTangent of the next point p_{i+1} in U.
(b) Describe how to implement the insertion of a new point p_{i+1} into U.
(c) Carry out a complexity analysis of your algorithm.

(d) Show how to modify your algorithms if the input can be degenerate. You must discuss how to implement the changes using the `LeftTurn`(p, q, r) predicate. NOTE: What is the correct output for U in the degenerate case? For this problem, ASSUME that an input point p is in U if it does NOT lie in the interior of the convex hull. ♢

Exercise 5.5: 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 lies 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 5.6: 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 `tangent`(p, H) in terms of `tangent`(p, H_U) and `tangent`(p, H_L).

(iii) Implement the operation `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 5.7: 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 5.8: 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 5.9: 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

§6. Fibonacci Heap Data Structure

¶23. The mergeable queues ADT. The **Fibonacci heap data structure** invented by Fredman and Tarjan (1987) gives an efficient implementation of the mergeable queues abstract data type (ADT).

The mergeable queues data structure stores several sets of items, and each set is stored in its own priority queue. As usual, the with priority given by the keys in item. The supported operations are as follows: assume Q, Q_i are queues, x is an item and k a key.

1.	`makeQueue`()$\to Q$	returns an empty queue Q
2.	`insert`(x, Q)	insert x into Q
3.	`deleteMin`(Q)$\to x$	the minimum item x in Q is deleted
4.	`decreaseKey`(x, k, Q)	the key in x is replaced by a smaller key k
5.	`union`(Q_1, Q_2)	form the union of the two queues

The first three operations of mergeable queues are just those of the priority queue ADT (§III.2). The two new operations are the ability to decrease the key of an item, and the ability to merge two queues. The full power of mergeable queues are not needed in applications with only one mergeable queue (so the `union` operation is irrelevant). See for example, the algorithms of Prim and Dijkstra in the next section.

We briefly add some further clarifications on these operations. 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. Union is sometimes known as the **meld** operation. 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 undefined if Q is empty. The operation `decreaseKey`(x, k, Q) is undefined if the key k is greater than the key in x.

There are additional useful operations that should probably be provided in practice, but is omitted in the above list for the sake of economy: deleting an item, making a singleton queue, and 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)$\to Q$	$\equiv$	`makeQueue`()$\to Q$; `insert`(x,Q).
`min`(*Queue* Q)$\to x$	$\equiv$	`deleteMin`(Q)$\to x$; `insert`(x,Q).

¶24. The Fibonacci heap data structure. We now consider the implementation details. Each mergeable queue Q is implemented by a **Fibonacci heap** H, where H is a collection of trees $T_1, \ldots, T_m$ with these properties:

- Each tree T_i satisfies the **min-heap property**: this means that each node has a key that is less than or equal to the key of its parent. 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 $H.\texttt{min}$, $H.n$ associated with H. The field $H.\texttt{min}$ 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 $x.\texttt{degree}$ (the number of children of x) and $x.\texttt{mark}$ (a Boolean value to be explained).

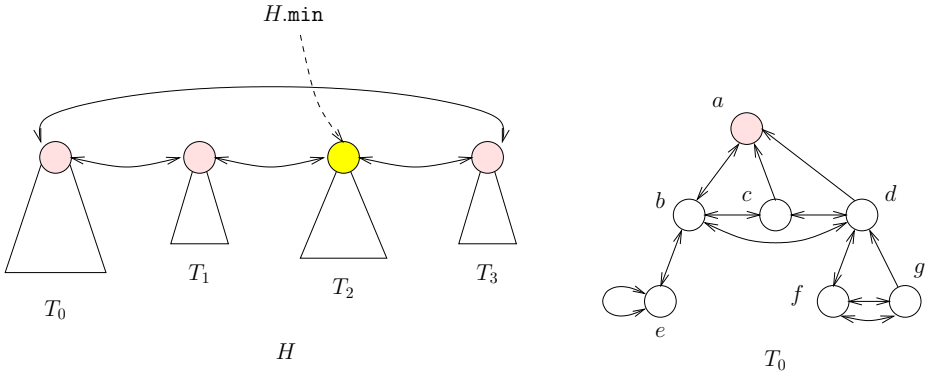


Figure 12: A Fibonacci heap $H = (T_0, \ldots, T_3)$: T_0 in detail

This is illustrated in Figure 12. 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).

¶25. 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 $x.\texttt{mark} = \texttt{true}$, and **unmarked** otherwise. Initially, x is unmarked. Our rules ensure that a root is always unmarked. We mark x if x is not a root and x loses a child for the first time. If x loses a child for the second time, it is cut and put in the root-list as a root (and therefore becomes unmarked again).

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

$$\Phi(H) := t(H) + 2 \cdot m(H) \tag{11}$$

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.

§7. 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 `deleteMin` operation. Finally, we verify the credit-potential invariant equation (3) 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.

`insert`(H, x): we create a new tree T containing only x and insert T into the root-list of H. Update $H.\texttt{min}$ and $H.n$ in the obvious way. Let us check the credit-potential invariant:

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

`union`(H_1, H_2): concatenate the two root-lists and call it H_1. It is easy to update $H_1.\texttt{min}$ and $H_1.n$. Checking the credit-potential invariant:

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

`deleteMin`(H): we remove $H.\texttt{min}$ from the root-list, and the child-list of $H.\texttt{min}$ 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[8] spend some extra effort to save future work. This is a process called **consolidation** which is explained next.

¶26. 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 to 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 make $A[k]$ point to x (we say x is "inserted" into the array A when this happens). But we can only insert x if $A[k] =$ `nil`; otherwise, suppose $A[k]$ points to some y. We then link x to y if x.`key` $\geq y$.`key`, and otherwise link y to x. Wlog, assume we linked x to y. Now, y has degree $k+1$ and we recursively try to "insert y", i.e., make $A[k+1]$ point to y, etc. 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 nodes after consolidation. Therefore $\Delta\Phi = \Phi_1 - \Phi_0 \leq 1 + D(n) - t_0$, as desired.

`decreaseKey`(x, k, H): this is the remaining operation and we will exploit the marking of items in a crucial way. The idea of the `decreaseKey` algorithm is basically to maintain heap property, namely that the parent should not have larger keys than its children. But by decreasing the key of x to k, we may cause a violation of this heap property. To remove the violation, we can cut the link of x to its parents. With this in mind, consider the details of this operation: First, we decrease the key of x to k (first checking that $k \leq x$.`key`). If x is a root, we are done. Otherwise, let y be the parent of x. If $k \geq y$.`key`, we are done. Otherwise, we cut x. Since x is now in the root list, we need to update H.`min`, etc. If x was marked, it is now unmarked. We must now treat y: if y is a root, we are done. Otherwise, if y was unmarked, we mark y and we are done. The last case is when y is marked. This is the most interesting case. But before treating it, we note that the other cases so far all have constant cost, and have constant increase in potential. Thus, they can be directly charged.

To treat the case where y is marked, we recall that this means y had previously lost a child,

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

and since it just lost a second child, viz. x, our rules require y to be cut. In fact, the cut may cascade into more cuts and the cost may be unbounded. Here is code fragment to "recursively cut" y:

```
RecursiveCut(y, H):
    If (y.mark = false and y ≠ root)
        y.mark ← true;
    elif (y ≠ root)      ◁ So y is marked
        y.mark ← false and 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). The worst case is when $m(H)$ decreases by $c-1$, giving us $\Delta\Phi \leq c - 2(c-1) = 2 - c$. By charging 2 units for this operation,

$$\textsc{Cost} = 1 + c, \quad \textsc{Charge} = 3, \quad \textsc{Credit} = 2 - c, \quad \Delta\Phi \leq 2 - c$$

and so the credit-potential invariant is verified.

In summary, we have achieved our goal of charging $O(1)$ units to every operation except for `deleteMin` which is charged $2 + 2D(n)$. We next turn to bounding $D(n)$. We remark on an unusual feature in our marking scheme: each non-root y can suffer the loss of at most two children before y itself is cut and made a root. But if y is a root, it may lose an unlimited number of children.

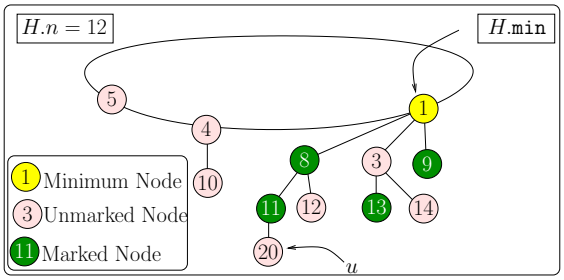


Figure 13: A Fibonacci heap

¶27. Example. Consider the Fibonacci heap in Figure 13, with keys written inside each node. Four of the nodes (keys $8, 9, 11, 13$) are "marked". We do not show sibling pointers to avoid clutter (e.g., $8, 3, 9$ are siblings).

- Consider the operation `decreaseKey`$(u, 6)$: the results in the data structure in Figure 14(a): we have to cut the node u, and this leads to a recursive cut of the marked keys 11 and 8. Thus, we add three new roots to the root-list: node u (6, formerly 20), node 11 and node 8.
- Consider the further operation of `deleteMin`(). The key 1 is deleted, but its children 3 and 9 are now added to the root-list. We now do consolidation of the list of roots

$$5, 4, 6, 11, 8, 3, 9.$$

It is not hard to see that we end up with the two roots 3 and 8. Moreover, $H.\texttt{min} \leftarrow 3$.

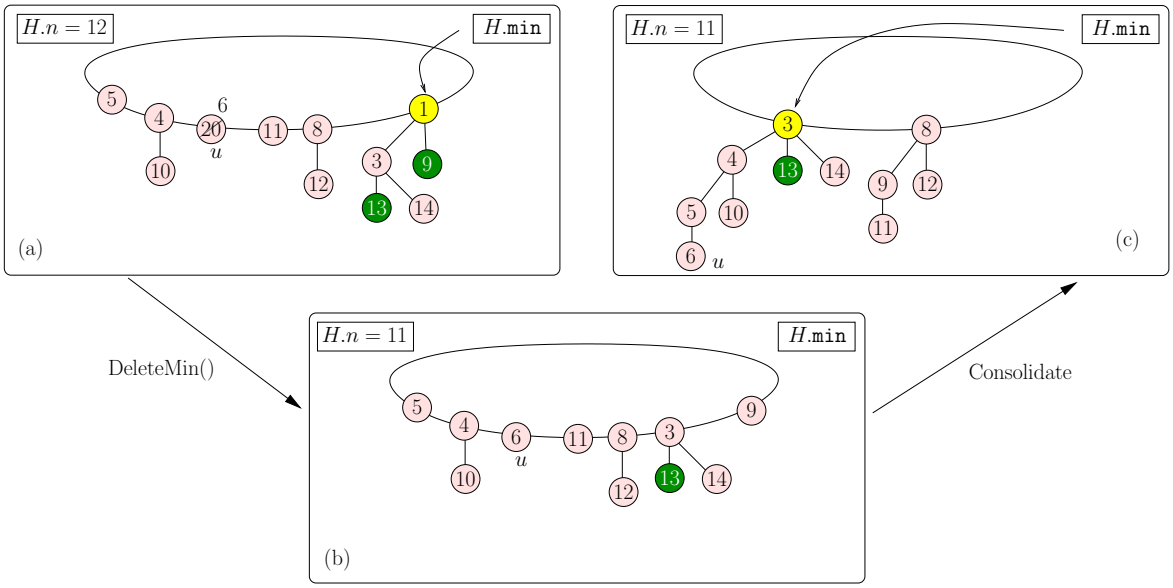


Figure 14: (a) After $\texttt{decreaseKey}(u, 6)$ (c) After $\texttt{deleteMin}()$

¶28. Degree Bound Let us now 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:

Lemma 10
(a) $F_i = 1 + \sum_{j=1}^{i-2} F_j$ *for* $i \geq 1$.
(b) $F_{j+2} \geq \phi^j$ *for* $j \geq 0$, *where* $\phi = (1 + \sqrt{5})/2 > 1.618$.

Proof. Part(a) follows easily by induction, or better still, by "unrolling" the recurrence for F_i. For part(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,

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

Q.E.D.

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

$$y_1, y_2, \ldots, y_d \tag{12}$$

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 11

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

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 12 *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 d. The result is true when $d = 0$ or $d = 1$. If $d \geq 2$, let $y_1, \ldots, y_d$ be the children of x as in (12). Then

$$\begin{array}{rcll} \textsc{Size}(x) & = & 1 + \sum_{i=1}^{d} \textsc{Size}(y_i) & \\ & \geq & 1 + \sum_{i=1}^{d} F_{2+y_i.\texttt{degree}} & \text{(by induction)} \\ & \geq & 2 + \sum_{i=2}^{d} F_i & \text{(by last lemma, and since } d \geq 2) \\ & = & 1 + \sum_{i=1}^{d} F_i = F_{d+2}. & \text{(by Lemma 10(a))} \end{array}$$

Q.E.D.

It follows that if x has degree d, then

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

where the last inequality is from Lemma 10(b). Taking logarithms, we immediately obtain:

Lemma 13

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

So that is where the name "Fibonacci" arises!

This completes our analysis of Fibonacci heaps.

Historical remarks. Prior to Fibonacci heaps, **binomial heaps** from Vuillemin (1978) were the best data structure for the mergeable queue ADT. Indeed, Fibonacci heaps has vestiges of binomial heaps as can be seen when we do consolidation. There is some interest to improve the amortized complexity of Fibonacci heaps to worst case complexity bounds. This was achieved by Brodal in 1996.

Exercises

Exercise 7.1: Suppose you insert the following sequence of items (represented by their keys) into an initially empty Fibonacci heap: $2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31$. Then you do a deleteMin. Next, you insert the following sequence of items into another Fibonacci heap: $37, 41, 43, 47, 53, 57, 59$. Then you do a deleteMin. (a) Draw the resulting data structure. (b) ♢

Exercise 7.2: On marking items.
(a) Which of the mergeable queue operations calls for marking?
(b) What is the rationale for marking items? ♢

Exercise 7.3: "Second chance Fibonacci heap." Suppose that instead of cutting a node just after it loses a second child, we only cut a node just after it loses a third child. Carry out the analysis as before. ♢

Exercise 7.4: Suppose $x > 1$ and $x^3 = x^2 + 1$, and ϕ is the Golden ratio. Prove that $x^2 > \phi > x$. ♢

Exercise 7.5: Generalize the above Exercises to the c-chance Fibonacci heap, for any integer $c > 1$. The constants ϕ for $c = 1, 2, 3, 7$ are plotted in Figure 15. ♢

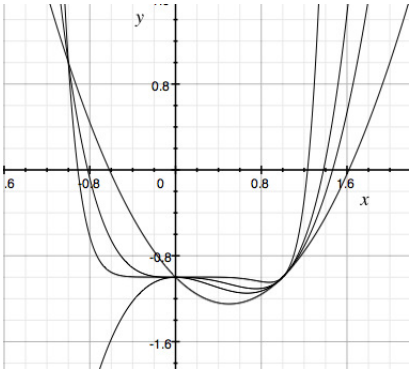


Figure 15: Constants for $c = 1, 2, 3, 7$

Exercise 7.6:
(a) Determine $\widehat{\phi}$, the other root of the equation $x^2 - x - 1 = 0$. Numerically compute $\widehat{\phi}$ to 5 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? ♢

End Exercises

§8. Pointer Model of Computation

There is an aesthetically displeasing feature in our consolidation algorithm: 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 also provide an elegant formalization of the pointer model, going back to Schönhage. This model can be used for a formal theory of computability and complexity. It has many advantages over the standard Turing machines model.

¶29. 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++`.

For each Fibonacci heap H, suppose we maintain a doubly-linked list of nodes

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

where $k \leq D(n)$ is the maximum degree of any node seen so far in H. We call this list the **degree register** because each node in H of degree i is required to have a pointer to node R_i (i.e., that node is "registered" at 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 degree of x.`parent` 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 $\mathcal{O}(D(n) + t)$ 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 $\mathcal{O}(t)$ time. The cost of this procedure is $\mathcal{O}(D(n) + t)$, as in the solution that uses array indexing.

But we can take this idea further: we can reinterpret the root list of H as a sublist of the degree register, and we now require that the degrees of roots in the root-list must be distinct. This ensure that the length of the root list is $\leq D(n)$, not n in the worst case. Whenever we add a new root of degree i to the root-list, we must check if there is already a root of degree i. If so, we link these two roots, and recursively register the result as a root of degree $i + 1$. The worst case cutting a node is thus $\Theta(\lg n)$ because of this recursion. Nevertheless, the amortized cost is $\mathcal{O}(1)$, using the same ideas as the Counter Example analysis (see §1). We get several benefits in this approach:

- We have described a purely pointer-based implementation of Fibonacci heaps.
- We have eliminated the explicit consolidation process.
- The operation `deleteMin`(H) amounts to a simple search through the degree register; hence its worst case time is no longer $\mathcal{O}(n)$ but $\mathcal{O}(\log n)$.

But there is a down side: when we merge two heaps, the cost is no longer $\mathcal{O}(1)$ but $\mathcal{O}(\log n)$ because of the need to merge their degree registers. But many applications of Fibonacci heaps (e.g., in the implementation of Prim's Algorithm for MST) do not need to merge heaps. In this case, as a degree register is no burden.

¶30. 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.

¶31. 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:

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

where P, Q are pointer variables. The New() operator (with no arguments) returns a 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 assignment 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 labeled 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:

$$(\langle \texttt{integer-expr} \rangle \langle op \rangle \langle \texttt{integer-expr} \rangle)$$

where $\langle op \rangle$ 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.

¶32. 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 16 and Exercise).

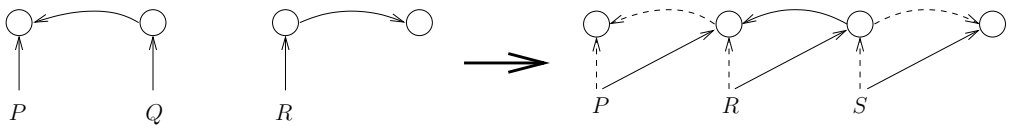


Figure 16: List Reversal Algorithm: the transformation at Line T.

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 Algorithms of Prim and Dijkstra

An original application of Fibonacci heaps is in computing minimum spanning trees (MST). We saw in §V.4 that there are several known algorithms for MST. Here, we are interested in efficient implementation of Prim's algorithm. Basically, we will reduce Prim's algorithm is a sequence of requests on a mergeable queue ADT. It turns out that Prim's algorithm has the same algorithmic framework as another famous algorithm, namely, Dijkstra's shortest path[9] algorithm. Hence we shall give them a unified treatment in this section.

Sorting Student: hey, I have seen this before!

The input for MST is a connected bigraph $G = (V, E; C)$ with edge cost function $C : E \to \mathbb{R}$. The fact that G is connected is not essential, but simplifies our discussion.

The shortest path problem solved by Dijkstra is for an input digraph $G = (V, E; C, s)$ where $C : E \to \mathbb{R}_{\geq 0}$ and $s \in V$. This is slightly different from the input for MST: we now treat a

[9] The general treatment of min-cost paths may be found in Chapter XIV.

digraph instead of a bigraph, and edge costs must be non-negative. Also, we are given a vertex $s \in V$ called the **source**. The goal is to compute the min-cost paths from s to each $v \in V$, where the cost of a path $v_0 - v_1 - \cdots - v_k$ is just $\sum_{i=1}^{k} C[v_{i-1}, v_i]$. The fact that edge costs for non-negative is critical for Dijkstra's algorithm, but not essential for MST. Basically, if we add a fixed large positive value to every edge cost of the MST, we get an equivalent MST instance with positive costs. Hence, we shall assume all costs are non-negative in this section.

Edges have non-negative costs

¶33. The Generic Framework. Both algorithms can be viewed as growing a set $S \subseteq V$ of vertices, adding one vertex to S at a time. The algorithms stop when $S = V$. We will initialize $S \leftarrow \{v_0\}$ where v_0 is arbitrary for MST but v_0 is the source for the shortest path instance. Define $U := V \setminus S$ as the complement of S. To choose the next vertex u from U to be added to S, we put the vertices of U into a min-priority queue Q. To keep track of the membership in U, let $\texttt{inU}[v \in V]$ be a global Boolean array where $\texttt{inU}[v] = \texttt{true}$ iff $v \in U$. Our choice for the next vertex u is just the minimum vertex in Q. The priority of each $v \in V$ will be defined by a global **min-cost array**

$$\texttt{minC}[v \in V]$$

that associates a cost $\texttt{minC}[v]$ with each $v \in V$. As discussed above, costs are non-negative: $\texttt{minC}[v] \geq 0$. The interpretation of $\texttt{minC}[v]$ will vary for the two applications. We can initialize $\texttt{minC}[v]$ to the cost of the edge $v_0 - v$ for both MST and Dijkstra.

S as the source set, and U as the unknown set.

Suppose we just added vertex u to S. We must now update the array $\texttt{minC}$ from u. Let us postulate a subroutine Update(u) for this task. Two properties of Update(u) are important to note: (1) We only need to update $\texttt{minC}[v]$ for those vertices v that are adjacent to u. (2) Any updated value $\texttt{minC}[v]$ must be a strict decrease in value. Since $\texttt{minC}[v]$ is used as the priority of v in Q, it means that we need to call $\texttt{decreaseKey}(v, \texttt{minC}[v], Q)$.

```
Generic Algorithm:
    ▷ INITIALIZATION
1.      Initialize the arrays inU, minC
    ▷ MAIN LOOP
2.      While Q ≠ ∅ do
3.          u* ← Q.deleteMin()
4.          Update(u*)
5.      Postprocess(minC)
```

After the main loop, Line 5 will construct the necessary output using the data in the min-cost array $\texttt{minC}$. Usually, Prim's algorithm must output a min-cost spanning tree, and Dijkstra's algorithm must output a shortest path from s to each $v \in V$. But for the simplified versions of Prim and Dijkstra, we reduce our tasks to computing only the respective minimum costs: min-cost of a spanning tree, and min-cost of the paths $s - \cdots - v$. It will be easy to convert our min-cost algorithms into algorithms that actually construct some min-cost objects (Exercise). For this simplified version, the post-processing of $\texttt{minC}$ is trivial in Dijkstra: we just output the array $\texttt{minC}$ because $\texttt{minC}[v]$ is just the min-cost of the path from from s to v. For Prim, postprocessing amounts to computing $\sum_{v \in V} \texttt{minC}[v]$.

¶34. Updates. Let us now develop the details specific to Prim's Algorithm. This amounts to filling the details for Update(u). At the end, we will mention changes to achieve the same

for Dijkstra's algorithm.

Recall that the notion of a set $S \subseteq V$ being "Prim-good" (§V.6). If S is a singleton then S is Prim-good. Suppose S is Prim-good and we ask how S might be extended to a larger Prim-good set. There is an ordering of the vertices in S such that $S = \{v_0, v_1, \ldots, v_t\}$ ($t = |S| + 1$) where v_i is added to S in the ith iteration. Write $S_i = \{v_0, \ldots, v_i\}$. Since S_i is Prim-good, there is a set E_i of edges that forms the MST for $G|S_i$. Let e_i be the edge such that $E_i = E_{i-1} \cup \{e_i\}$. In particular e_i has the form v_j–v_i for some $j < i$. We do not plan to explicitly maintain any of these information, but this implicit information will be necessary for understanding the correctness. Let us maintain the following invariant relative to the set S:

- For each $v_i \in S = \{v_0, \ldots, v_t\}$, let $\texttt{minC}[v_i]$ be the cost of the edge e_i.
- For each $u \in U$, let

$$\texttt{minC}[w] := \min\{C[v, w] : (u\text{–}v) \in E, v \in S\}.$$

In order to find a node $u^* \in V - S$ with the minimum `minC`-value, we will maintain $V - S$ as a **single**[10] mergeable queue Q in which the least cost $\texttt{minC}[u]$ serves as the key of the node $u \in V - S$. Hence extending the Prim-good set S by a node u^* amounts to a `deleteMin` from the mergeable queue. After the deletion, we must update the information $\texttt{mst}[S]$ and $\texttt{minC}[v]$ for each $v \in V - S$. But we do not really need to consider every $v \in V - S$: we only need to update $\texttt{minC}[v]$ for those v that are adjacent to u^*. The following code fragment captures our intent.

```
UPDATE(u, S):
1.   inU[u] ← false.
2.   for w adjacent to u,
2.1      If inU[w] and (minC[w] > C[u,w]) then
2.2          minC[w] ← C[u,w] .
2.3          DecreaseKey(w, minC[w], Q).
```

For Dijkstra's Algorithm, we just change Steps 2.1 and 2.2 to:

```
2.2          minC[w] ← minC[u] + C[u,w]}.
2.1      If inU[w] and (minC[u] + C[u,w]) then
2.2          minC[w] ← minC[u] + C[u,w] .
```

The correctness of Dijkstra's Algorithm will be proved in §XIV.3.

¶35. Complexity Analysis. The complexity parameters are $n := |V|$ and $m := |E|$. We assume the mergeable queue is implemented by a Fibonacci heap. The initialization takes $O(n)$ time. In the main while-loop, we perform $n - 1$ iterations to add vertices to S. Each iteration calls `deleteMin` once, for a total cost of $O(n \log n)$. We account for the calls to UPDATE(u)

[10] 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.

separately: when calling Update(u) subroutine, we call the DecreaseKey operation for each edge in u's adjacency list. This gives a total of m such calls in all the Update calls. This has total cost $O(m)$. The other costs of Update(u) is $O(1)$ and can can be charged to u, for an overall cost of $O(n)$. Thus the total cost to do the updates is $O(m+n)$. 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 $O(n \log n)$. The postprocessing work is $O(n)$. We have proven:

Theorem 14 *The complexity of Prim's and Dijkstra's algorithm on a graph with n vertices and m edges is in $O(m + n \log n)$.*

Exercises

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

Exercise 9.1: Use the Tabular Method to hand simulate Prim's algorithm on the following graph (Figure 17) beginning with v_1:

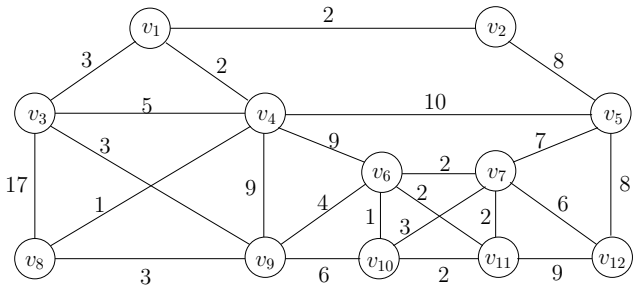


Figure 17: Graph of a House

The Tabular Method amounts to filling in the following table, row by row. We have filled in the first three 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}	$mst[S]$	New Edge
1	$\underline{2}$	3	2	∞	∞	∞	∞	∞	∞	∞	∞	2	(v_1, v_2)
2	*	"	$\underline{"}$	8	"	"	"	"	"	"	"	4	(v_1, v_4)
3	*	$\underline{"}$	*	"	9	"	1	9	"	"	"	7	(v_1, v_3)

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

Exercise 9.2: Repeat the simulation of the previous exercise, but this time, we implement Prim's Algorithm using Fibonacci heaps. and run the algorithm on the Graph in Figure 14 (Lect.VI). Show the state of your Fibonacci heap in each stage of the algorithm. ♢

Exercise 9.3: 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 using the Tabular Method (see the first Exercise above). Show the final MST and its cost.

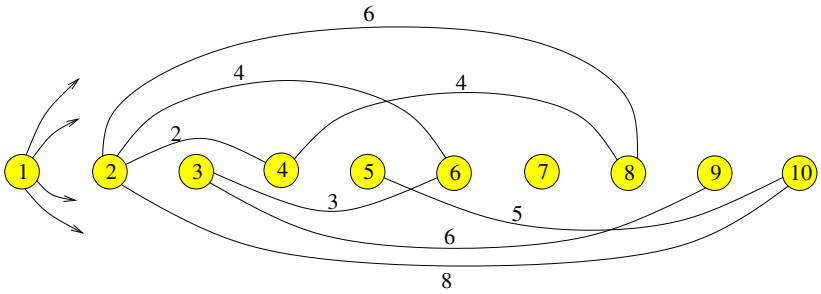


Figure 18: G_{10}: edges from node 1 are omitted for clarity.

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

Exercise 9.4: Modify the above algorithm for computing the minimum cost of a MST into one that constructs the MST. ♢

Exercise 9.5: Modify the above algorithm to compute a minimum spanning forest in case the input graph is not connected. ♢

Exercise 9.6: Let $G = (V, E; \mu)$ be an edge-costed bigraph and $T \subseteq E$, $S \subseteq V$. Let $V(T) = \{v \in V : \exists u, (u, v) \in T\}$ denote the *vertices of* T, and $G|S := (S, E'; \mu)$ where $E' = E \cap \binom{S}{2}$ denote the *restriction of* G *to* S. We define T to be *prim-good* if T is an MST of $G|V(T)$ and T can be extended into an MST of G. We define S to be *prim-good* if S is singleton or there exists a prim-good set T of edges such that $S = V(T)$. Show or give a counter-example:
(a) T is a tree of $G|V(T)$ and can be extended into an MST of G implies T is prim good.
(b) S is prim-good implies every MST of $G|S$ is prim-good. ♢

End Exercises

§10. 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.

¶36. 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 $i = 0, 1, 2$.

¶37. 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

$$\widehat{C} < 2C^*. \tag{13}$$

Thus, the move-to-front rule achieves close to optimal bound without having to know in advance the frequency distribution of requests, or to sort the lists.

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

$$U = (r_1, r_2, \ldots, r_n) \tag{14}$$

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 (13), 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_*. Note that α_*, being offline, can determine the best position to move each element after it has been accessed *based on the entire sequence U in (14)*. If the cost of α_* is denoted $COST_*$, we prove

$$COST_1(L, U) \leq 2 \cdot COST_*(L, U). \tag{15}$$

We introduce the following potential function on lists. 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'. We will compare the list L produced by our move-to-front algorithm to the list L^* obtained from the optimal algorithm: the **potential** $\Phi(L)$ of L is defined to be the number of inversions in (L, L^*). For instance, $\Phi(L) = 5$ in Figure 19 as there is one inversion involving a, two inversions involving b (not counting that with a), two inversions involving c (not counting those with a or b) and 0 inversion involving d (not counting those with a, b, c).

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

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

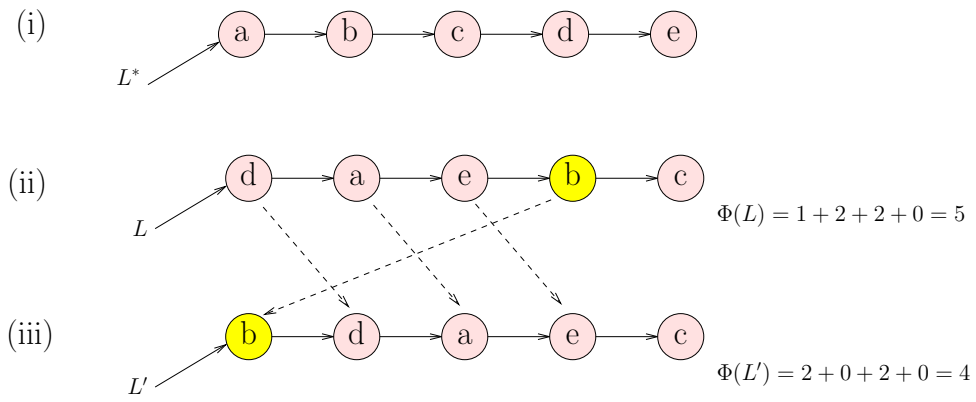


Figure 19: How potential changes under update: $L' = \alpha_1(L, b)$

CLAIM: *The number of inversions destroyed is ℓ_j and the number of inversions created is at most k_j.*

In illustration, consider the list L' in Figure 19(iii), produced by the move-to-front algorithm after accessing b in L. Thus

$$x_j = b, \quad k_j = 1, \quad \ell_j = 2, \quad c_j = 4, \quad c_j^* = 2.$$

Note that ℓ_j counts the elements d and e. They represent the inversions $\{b, e\}$ and $\{b, d\}$ in L. Both inversion were **destroyed** when b moved to the front in L'. Likewise, k_j counts the element a; it represents the new inversion $\{a, b\}$ that was **created** in L' when b moved to the front. Next, the optimal algorithm α^* is allowed to update L' by moving b closer to the front of its list. Each element that b moves past when updating L' will reduce the number of created inversions by the move-to-front algorithm. It should now be clear that the above CLAIM is true.

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

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

¶39. 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 competitiveness 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.

¶40. Final Remarks. Amortization 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]. 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 [6] and Purdom and Brown [9].

Exercises

Exercise 10.1: 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 10.2: 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 10.3: (Bentley, Sleator, Tarjan, Wei) Consider the following data compression scheme based on any list updating algorithm. We encode an input sequence S of characters by each character's position in a list L. The trick is that L is dynamic: we update

L by accessing each of the characters 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 V.4.26 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 characters 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 character 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 characters $x_1, \ldots, x_n$ 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 character 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 character 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

§11. Amortized Subdivision Trees

Quadtrees are 2-dimensional generalizations of binary trees. For this purpose, it is useful to re-imagine binary trees as having nodes that are real intervals. Let T be a binary tree whose root is an interval I_0. In general, each node of T is an interval $I = [a, b]$ and the children of I are the subintervals $[a, (a + b)/2]$ and $[(a + b)/2, b]$. Thus the leaves of T is therefore collection of intervals that forms a subdivision of I_0.

In 2-dimensions, a quadtree has nodes that are boxes, where a box B is a Cartesian product of two intervals, $B = I \times J$. See Figure 20(b) for a quadtree. Each box B can be subdivided into four congruent subboxes called the **children** of B. A **quadtree** is a tree whose nodes are boxes, and each internal node B has four children which are congruent subboxes as described. The set of leaves of the quadtree forms a **subdivision** of the root box. Figure 20(a) shows the subdivision of the root box of the quadtree in Figure 20(b). One application of quadtrees is in robot motion planning, illustrated in Figure 21.

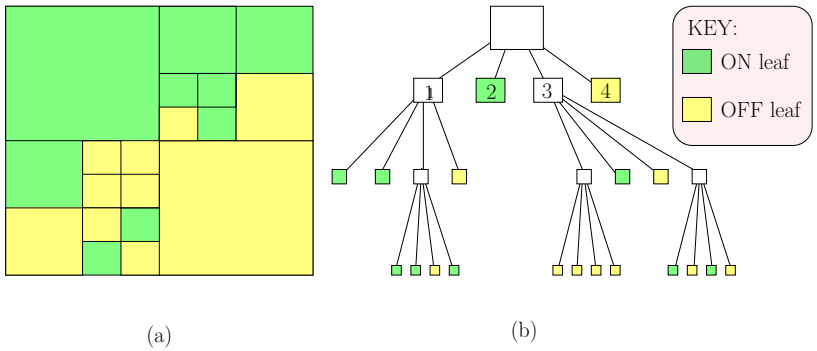


Figure 20: (a) Subdivision of a box and (b) associated quadtree

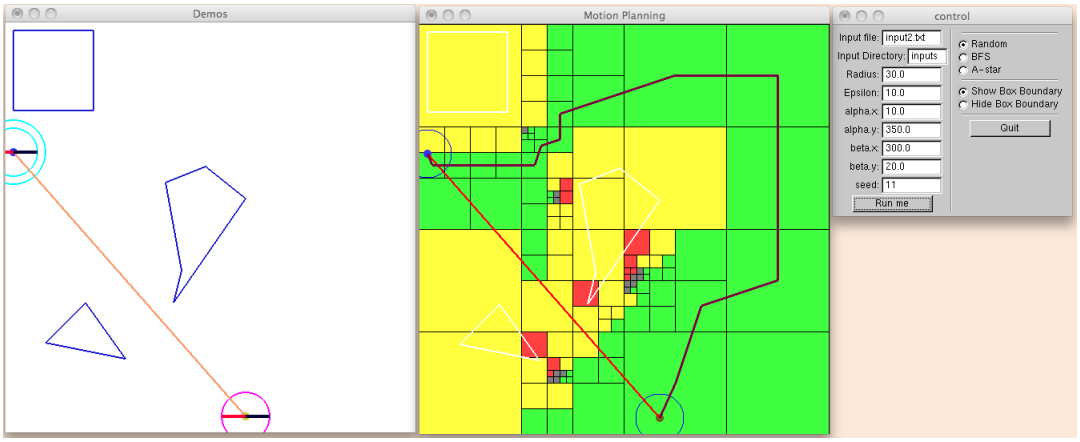


Figure 21: (a) start/goal position of disc robot amidst obstacles, (b) Subdivision Search for Path

The boundary of a box can be divided four closed line segments called **sides**. Borrowing a cartographic term, we label the sides after the four (cardinal) compass directions: N, S, E, W. Two sides meet at a **corner**, which can be label with the four ordinal compass directions: NE, NW, SW, SE. Two boxes B, B' are **k-adjacent** if $B \cap B'$ is a k-dimensional set. Note these possibilities:

- 2-adjacent: this means the interiors of B and B' intersect. If B, B' are nodes of a quadtree, this implies a containment relationship, $B \subseteq B'$ or $B' \subseteq B$.
- 1-adjacent: this means $B \cap B'$ is a line segment of positive length. In a quadtree, $B \cap B'$ must be the side of either B or B' (or both).
- 0-adjacent: $B \cap B'$ is a common corner of B and B'.
- (-1)-adjacent: B and B' are disjoint.

For most applications, the 1-adjacency relationship between two boxes are the most important. We will simply say "adjacent" to mean 1-adjacency. Say B' is **adjacent in direction** $D \in \{N, S, E, W\}$ if $B' \cap B$ is contained in the D-side of B. The **D-neighbors** of B are those leaf boxes which are adjacent in the direction D. In general, B will have at least four neighbors in

each of the compass directions, unless B shares the boundary of the root B_0. The four directions have some underlying structure: the **opposite** of any direction D is defined as follows: N and S are opposites of each other, as are E and W. Two directions that are not opposites of each other are said to be **orthogonal**.

¶41. Semi-Dynamic Quadtrees. So far, we have viewed a quadtree as a static data structure in which we can make neighbor queries. But most applications require a more dynamic data structure, in particular one that can grow by "splitting" at any leaf u. To split at u, we just add its four children to the tree. Now u is no longer a leaf, but these four children are new leaves. Let n be the size of the tree, n_i the number of internal nodes, and n_ℓ the number of leaves. This split operation shows that n, n_i, n_ℓ increases by 4, 1 and 3, respectively. Initially, $(n, n_i, n_\ell) = (1, 0, 1)$. Thus after k splits from the initial root box, $(n, n_i, n_\ell) = (1+4k, k, 1+3k)$. E.g., Figure 20 shows a quadtree with $k = 6$. Thus $(n, n_i, n_\ell) = (25, 6, 19)$. In particular, the ratio of size to number of leaves is $(1+4k)/(1+3k)$ which approaches $4/3$ as k grows. Note that there are some applications where we only need the leaves of the quadtree i.e., the underlying subdivision of the root B_0. This analysis shows that by maintaining the entire quadtree instead of just the leaves, we incur an overhead of 33.3% in space usage. In the following, we shall assume that it is not an issue.

Partial Quadtrees. There are applications where may not need even the full set of leaves. For example, in motion planning, boxes represent a set of robot positions. Some robot positions are forbidden, and we classify boxes as "OFF" if all the positions in that box is forbidden, otherwise it is "ON". If we are interesting in path finding, we may[a] ignore the "OFF" nodes. This is illustrated in Figure 20 where the green boxes are ON, and yellow boxes are OFF.
More generally, let us say that we are given a box predicate $ON(B)$ that evaluates to true iff the box B is of interest to the application. We say B is either "ON" or "OFF" accordingly. Moreover, the predicate is hereditary in the sense that if B is OFF, then any subbox $B' \subseteq B$ is also OFF. In this case, we can maintain a **partial quadtree** where we only retain those leaves that are ON; now each internal node between 1 and 4 children. Such a tree might have a path in which all the nodes have only one child. There are techniques for compressing the representation in this case.

[a] This is not entirely true; if we wish to detect non-existence of paths in finite time, we may need to maintain the OFF boxes to detect such conditions.

Let us consider the problem of maintain a binary tree, subject to splitting a leaf, and neighbor queries. In order to answer neighbor queries, we need ma

Exercises

Exercise 11.1: Suppose for each box u in our quadtree, we maintain four **adjacency pointers**, $u.D$ ($D \in \{N, S, E, W\}$) which point to a box adjacent in direction D. The box $u.D$ is uniquely defined if we require it to have the largest depth that is at most $depth(u)$. Of course, if u has no adjacent boxes in direction D, we let $u.D =$ `nil`. Show that we can find all the neighbors of any box u in a quadtree in amortized $O(1)$ time per neighbor. ♢

End Exercises

§12. Continuous Amortization

We consider the highly classical problem of root finding: given a real function $f : \mathbb{R} \to \mathbb{R}$, to compute all the real roots of f. The **roots** or **zeros** of f are those $\alpha \in \mathbb{R}$ such that $f(\alpha) = 0$. Equivalently, we want to solve the equation

$$f(x) = 0.$$

There are many variations of this problem: if $f(x) = ax^2 + bx + c$, then we know from high school the quadratic formula for the roots: $\alpha = (-b \pm \sqrt{b^2 - 4ac})/2a$. But Abel tells us that when f is a polynomial of degree more than 5, we cannot expect to find analogous formula in terms of radicals. Hence, we seek numerical approximations. Here are some results from the computer algebra system `Maple`: if

$$f := 2\,x^5 - 11\,x^4 - 7\,x^3 + 12\,x^2 - 4\,x$$

then calling `fsolve`(f) will produce the result:

MAPLE CODE HERE (THIS HAS BEEN COMMENTED OUT TEMPORARILY – Feb 2017)

So `Maple` gives us three approximate roots (including the root $\alpha = 0$ which we could already expect just by looking at f). The above output shows 10 digits of significant digits, by default. But we could have ask `Maple` for any number of significant digits. Here is another example from `Maple`: if

$$g := x^{20} - (20\,x - 1)^2$$

then `fsolve`(g) produces four real roots:

MAPLE CODE HERE (THIS HAS BEEN COMMENTED OUT TEMPORARILY – Feb 2017)

$$-1.400398846,\ 0.05000000000,\ 0.05000000000,\ 1.389281264$$

This example show two roots that are numerically identical up to 10 significant digits, both roughly equal to 0.05. Are they identical roots or is their difference less than 10^{-10}? Of course, if there are not identical, we can keep asking `Maple` for more and more significant digits until the difference is revealed. But if they are identical, this process will not terminate. In algebraic computation, we therefore prefer to approximate α by an **isolation interval**, namely an interval $[\underline{a}, \overline{a}]$ that contains exactly one root of $f(x)$ and that root is α. So we pose the **real root isolation problem** as follows: *given a real function f and an interval of interest I_0, to compute an isolating interval for each real root of f in I_0.*

Several remarks: We may choose I_0 to be the real line in which case we are interested in all the real roots of f. If we further specialize the function f to be polynomials, and the coefficients of f specialized to integers, we will call it the **benchmark problem**. This critical case has occupied theoretical Computer Science since Schönhage's 1982 landmark unpublished paper [10] in which he established that the benchmark problem has bit complexity $\widetilde{O}(d^3L)$ where d is the degree of f and its coefficients have magnitude $< 2^L$. When the coefficients of f are real numbers for which you can ask for arbitrarily good approximations (this is also known as the bit-stream model) is important for many applications in computer algebra (the coefficients

here are actually real algebraic numbers). Of course we can also isolate the complex roots of f. Another important generalization is to look at multivariate functions. Suppose we have m polynomials $f_1, \ldots, f_m \in \mathbb{Z}[X_1, \ldots, X_n]$ in n variables. We want to solve the simultaneous equation $0 = f_1 = f_2 = \cdots = f_m$. The two main cases here are (i) $m = n$ and there are finitely many zeroes, (ii) $m = 1$ and the zero set is a $n-1$ dimensional surface. We must interpret (ii) correctly: since the zero set is infinite, we can only approximate the surface. The analogue of root isolation is to compute an approximate surface with the correct topology (technically, the approximate surface should be isotopic to the zero set).

¶42. Generic Subdivision Algorithm. We shall focus on a very simple class of algorithms for root isolation. Suppose we have a interval predicate C_* such that for any interval I, we have $C_*(I) \in \mathbb{N}$ with the properties

- $C_*(I) = 0$ implies I contains no zeros of f.
- $C_*(I) = 1$ implies I contains one zeros of f.

Consider the following algorithm that uses two queue data structures, Q and P:

```
Generic Subdivision Algorithm:
Input: f and I_0
Output: A set P of intervals
    Q ← {I_0}, P ← ∅
    WHILE Q ≠ ∅
        I ← Q.remove()
        If (C_*(I) < 2)
            P.append(I)
        Else
            Let I = [a, b] and m = (a + b)/2
            Q.append([a, m]), Q.append([m, b])
```

This algorithm creates a **subdivision tree** rooted at I_0 such that each internal node is an interval $[a, b]$ with two children $[a, (a+b)/2]$ and $[(a+b)/2, b]$. At any moment, the set of the intervals in $P \cup Q$ forms a **subdivision** of I_0, i.e., $I_0 = \cup I : I \in P \cup Q$ and the interiors of two intervals in $P \cup Q$ are disjoint.

Upon termination, the intervals in P forms a partition of I_0. Every root α of f in I_0 will be represented by some isolating interval in P. But notice that if two intervals in P shares an endpoint m that happens to be a root, then we must choose only one of these two intervals for a proper output for the root isolation problem. But by a post processing of P, we can easily resolve this issue. Another important remark to make is that we assume that it is possible to determine the sign of f at the endpoint of any subdivision interval. This assumption is quite nontrivial. Assuming that I_0 initially has endpoints that are dyadic numbers, we are assuming that the exact sign of f at dyadic numbers are possible. The Exercise explores alternatives that avoid this assumption.

The main question is whether this algorithm terminates. Suppose C_* is convergent in the following sense: for $p \in \mathbb{R}$, let $C_*(p) = 0$ if $f(p) \neq 0$ and $C_*(p) = 1$ if $f(p) = 0$. for any sequence of intervals $I_1, I_2, \ldots$ that converges to a point p, we have $C_*(I_i) \to C_*(p)$ (this means that $C_*(I_i) = C_*(p)$ for all i large enough. We have the following easy lemma:

Lemma 15 *If C_* is convergent, then the Generic Subdivision Algorithm always halts. Upon halting, P contains (among other things) an isolating interval for each root of f in I_0.*

¶43. Sturm Method. Suppose f is a polynomial. The theory of Sturm sequences can provide a predicate $C_*(I)$ that counts exactly the number of real roots of f inside I_0. The predicate is clearly convergent.

Give an Example.

The subdivision tree produced in this case is the minimal subdivision tree possible. It can be shown that for the benchmark problem, the tree size is $O(d(L + \log d))$. Unfortunately, in practice, this methods based on Sturm sequences is not competitive.

¶44. Descartes Method. Suppose $f = \sum_{i=0}^{d} a_i X^i$ is a polynomial. The **sign variation** of f is the number of sign changes in the sequence $(a_0, a_1, \ldots, a_d)$, where (a_i, a_j) counts as a sign change if $i < j$, $a_i a_j < 0$ and $0 = a_{i+1} = \cdots = a_{j-1}$. Denote this number by $Var(f)$.

Lemma 16 ([) *escartes' Rule of Sign/descartes Suppose f has k positive real roots. Then $Var(f) - k$ is a non-negative even number.*

Thus, $Var(f)$ is an upper bound on the number of real positive roots of f. What if we are interested in roots of f in some interval $I = [a, b]$, not just the positive real roots of f? Well, we can apply a Mobius transformation: let

$$f_I(X) := \frac{1}{(X+1)^d} f\left(\frac{aX + b}{X + 1}\right).$$

Then the set of positive real root of f_I is mapped bijectively to the real roots of f in I. We can therefore apply Descartes rule of sign to f_I to estimate the number of roots of f in I.

Example.

We now instantiate the predicate $C_*(I)$ in the Generic Subdivision Algorithm by the **Descartes predicate**:

$$C_*(I) := Var(f_I).$$

By Lemma **??**, if $C_*(I) = 0$, there are no roots of f in I, and if $C_*(I) = 1$, there is exactly one real root of f in I. We can further prove:

Lemma 17 *The Descartes predicate, $C_*(I) = Var(f_I)$ is convergent.*

Currently, the Descartes method for root isolation is one of the most efficient exact algorithms. Moreover, one can show [4] that the tree size for the Descartes method in the benchmark problem is also $O(d(L + \log d))$.

¶45. Bolzano Method. The following result is most intuitive: *if f is a continuous real function, and $a \neq b$ with $f(a)f(b) < 0$, then there exists some c between a and b such that $f(c) = 0$.*

This statement goes back to Bolzano (1817) which is a form of the Intermediate Value Theorem. We propose to develop predicate $C_*(I)$ based on this simple principle. The advantage is that f can be quite general functions. The basis for such methods is interval arithmetic: if $f : \mathbb{R} \to \mathbb{R}$ is any function, then its set theoretic extension (still denoted by f) is defined by $f(S) := \{f(x) : x \in S\}$ for any $S \subseteq \mathbb{R}$. Now, a function of the form $\Box f : \Box\mathbb{R} \to \Box\mathbb{R}$ is called an **interval version** of f if for all $I \in \Box\mathbb{R}$, $f(I) \subseteq \Box f(I)$. If for all interval sequences $I_1, I_2, \ldots$ that converges to a point $p \in \mathbb{R}$, we have $\Box f(I_i) \to f(p)$ then we say $\Box f$ is **convergent**. A **box function** for f is any interval version of f that is convergent. We now define

$$C_*(I) = \left.\begin{array}{ll} 0 & \text{if } 0 \notin \Box f(I), \\ 1 & \text{if } 0 \notin \Box f'(I), \text{and } f(a)f(b) < 0 \text{ where } I = [a, b], \\ 2 & \mathsf{else}. \end{array}\right\} \tag{16}$$

Here $\Box f$ and $\Box f'$ are box functions for f. Using interval arithmetic, it is easy to construct such box functions for polynomials.

By the EVAL algorithm, we mean the Generic algorithm instantiated by (16). Our goal is to sketch out a proof that EVAL also produces an optimal tree size of $O(d(L + \log d))$ for the benchmark problem. But we need to be careful: the result cannot be true for arbitrary box functions. It is true for the "idea

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. Theory of Comput.*, 7, 1985. Providence, Rhode Island.

[4] A. Eigenwillig, V. Sharma, and C. Yap. Almost tight complexity bounds for the Descartes method. In *31st Int'l Symp. Symbolic and Alge. Comp. (ISSAC'06)*, pages 71–78, 2006. Genova, Italy. Jul 9-12, 2006.

[5] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM J. Comput.*, 15:287–299, 1986.

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

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

[8] M. H. Overmars and J. van Leeuwen. Dynamic multi-dimensional data structures based on quad- and k-d trees. *Acta Inform.*, 17:267–285, 1982.

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

[10] A. Schönhage. The fundamental theorem of algebra in terms of computational complexity, 1982. Manuscript, Department of Mathematics, University of Tübingen. Updated in 2004 with typo corrections and an appendix of related subsequent papers.

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

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

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