Chapter 17

Appendix: Splay trees and link-cut trees

17.1 Binary Search Trees

A binary search tree (BST) is a rooted binary tree in which each nonroot node x is designated as either its parent's *left* child or its parent's *right child*. (Each node has at most one left child and at most one right child.)

Such a tree T defines a total order on its vertices, called *BST order* (or *symmetric order*). Intuitively, the order is left-to-right. Formally, we inductively define BST order as follows. The BST sequence for an empty tree is the empty sequence. For a nonempty tree T, the BST sequence consists of the BST sequence of the left subtree of T (if there is such a subtree) followed by the root of T followed by the BST sequence of the right subtree (if there is such a subtree).

BSTs are often presented as data structures for representing dictionaries, but they are also useful for representing sequences and, more specifically, orderings. To represent an ordering σ of some finite set S, we use a BST whose vertices are the elements of S; the BST order gives the ordering σ.

The non-uniqueness of the representation turns out to be very useful. This is a recurring theme in computer science.

A BST can be implemented using three mappings from S to S: parent$(\cdot)$, left$(\cdot)$, and right$(\cdot)$.

Often it is useful to represent an assignment $w(\cdot)$ of weights to the elements of S. The next section introduces a useful way to represent such an assignment.

17.1.1 Delta Representation of weights

Rather than store explicit values at each node representing the item's weight, we can use an implicit representation. We represent the weight of node u by storing the difference $\Delta w(u)$ between the node's weight and that of its parent.

If the node is the root then $\Delta w(u)$ is assigned the weight of u itself. Thus the following invariant holds:

Delta invariant: For each node u, $w(u) = \sum_v \Delta w(v)$, where the sum is over the ancestors of u.

Refer to Figure 17.1 for an example.

This representation allows us to efficiently perform an operation that adds to the weights of many vertices at a time. To add β to the weights of all the descendants of a node v, add β to $\Delta w(v)$.

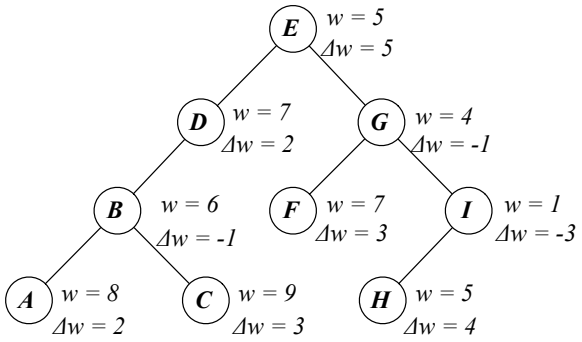

Figure 17.1: A BST with node-weights. The weight of each node is indicated by w. The difference between w for a node and w for its parent is indicated by Δw. Note that, for each node, w for that node can be computed by summing Δw over all the ancestors of that node.

We use the term *decoration* to refer to additional field associated with each node.

17.1.2 Supporting searches for small weight

We can augment the BST to support queries such as this:

> SearchLeftmost(λ)**:** What is the leftmost element x in σ whose weight $w(x)$ is no more than λ?

To support such queries, we define $minw(\cdot)$ so that, for each node v, $minw(v)$ is the minimum weight assigned to a descendant of v.

Note that, for a node v with children $v_1, \ldots, v_k$ ($k \leq 2$),

$$minw(v) = \min\{w(v), \min_i minw(v_i)\} \tag{17.1}$$

This equation will be useful in updating $minw(v)$ when v's children change.

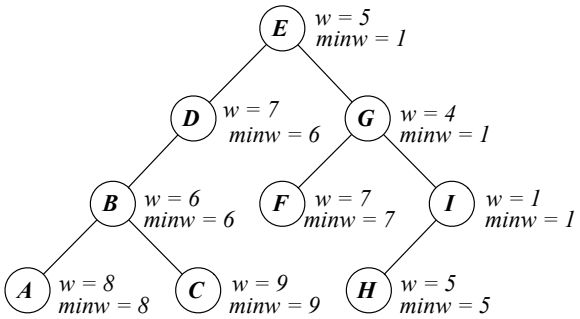

Figure 17.2: The weight of each node is indicated by w. The minimum weight of the subtree rooted at a node is indicated by *minw*.

Problem 17.1.1. *Write pseudocode in terms of* minw$(\cdot)$ *for* SearchLeftmost(λ). *The time required should be proportional to the depth of the tree.*

17.1.3 Delta Representation of min-weights

The implicit representation of $minw(\cdot)$ builds on the implicit representation of $w(\cdot)$. We label the vertices using $\Delta minw(\cdot)$ so that the following invariant is satisfied:

> **Delta min invariant:** For each node u, $minw(u) = \Delta minw(u) + w(u)$

Refer to Figure 17.3.

When we add β to the weights of descendants of u by adding β to $w(u)$, the *Delta min* invariant is automatically maintained.

Altering the structure of a BST in such a way as to change the descendants of u changes $minw(u)$, so $\Delta minw(u)$ must be updated. Now we derive the rule for the update. Let $\hat{w}$ be the weight of the parent of u, or zero if u has no parent. Let $u_1, \ldots, u_k$ be the children of u.

$$\begin{aligned}
minw(u) &= \min\{w(u), minw(u_1), \ldots, minw(u_k)\} \\
\Delta minw(u) &= minw(u) - w(u) \\
&= \min\{w(u) - w(u), minw(u_1) - w(u), \ldots, minw(u_k) - w(u)\} \\
&= \min\{0, (\Delta minw(u_1) + w(u_1)) - w(u), \ldots, (\Delta minw(u_k) + w(u_k)) - w(u)\} \\
&= \min\{0, \Delta minw(u_1) + \Delta w(u_1), \ldots, \Delta minw(u_k) + \Delta w(u_k)\} \qquad (17.2)
\end{aligned}$$

17.1.4 Delta representation of left-right

Suppose that, in our representation of a sequence σ using a BST, we wish to quickly reverse the consecutive subsequence induced by the descendants of a

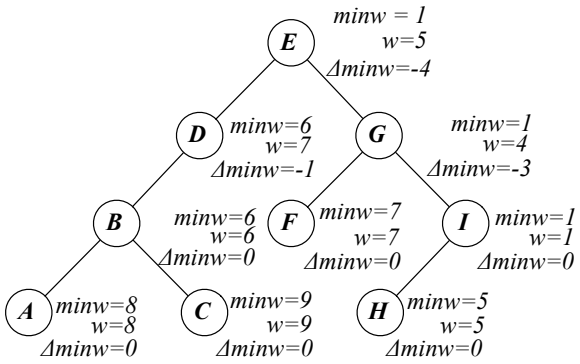

Figure 17.3: The minimum weight of the subtree rooted at a node is indicated by *minw*. The weight of each node is indicated by w. The difference bewteeen *minw* and w for a node is indicated by Δ*minw*.

node. For example, the BST in Figures 17.2 and 17.1 represents the sequence $ABCDEFGHI$. We wish to modify this sequence by reversing the subsequence $FGHI$, obtaining $ABCDEIHGF$.

To support such modifications, we use a delta representation of left and right. We associate with each node a two-element array, indexed by 0 and 1. A binary value *flipped* associated with each node signifies which element of the array gives the left child of that node. Instead of explicitly representing *flipped*(v) at each node, we explicitly represent Δ*flipped*(v), which is defined as *flipped*(v) − *flipped*(parent(v)) where the subtraction is mod 2. Therefore for each node v the value of *flipped*(v) is the sum of Δ*flipped*(x) over all ancestors x of v, where the sum is mod 2.

To reverse the order among the descendants of a node v, we add one (mod 2) to Δ*flipped*(v).

17.1.5 Rotation: An order-preserving structural change to a binary search tree

A binary search tree can be structurally changed by rotating a node up in the tree. For example, for any node x in a BST there is a series of rotations that results in x being the root. See Figure 17.4. Rotation preserves the BST order. We will later describe a rule for carrying out rotations that enables us to get good time bounds for binary-tree operations.

17.1.6 Updating Delta representations in a rotation

Since Delta representations depend on the structure of a BST, these must be updated when a rotation takes place.

Refer to Figure 17.5. As shown there, let b be the root of the subtree labeled B. Let $\hat{w}$ be the weight of the parent of y (or zero, if y has no parent).

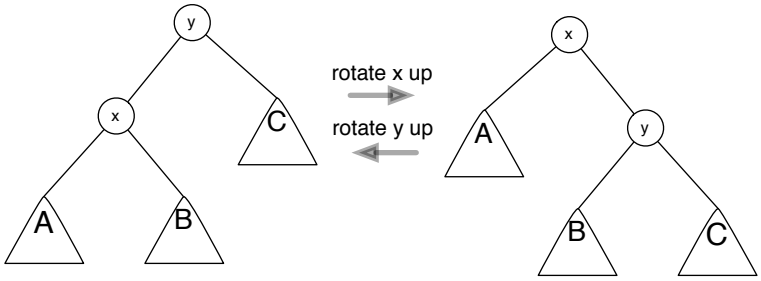

Figure 17.4: Starting with the tree on the left, rotating x up yields the tree on the right. Starting with the tree on the right, rotating y up yields the tree on the left. The triangles represent subtrees that are not changed by the rotation.

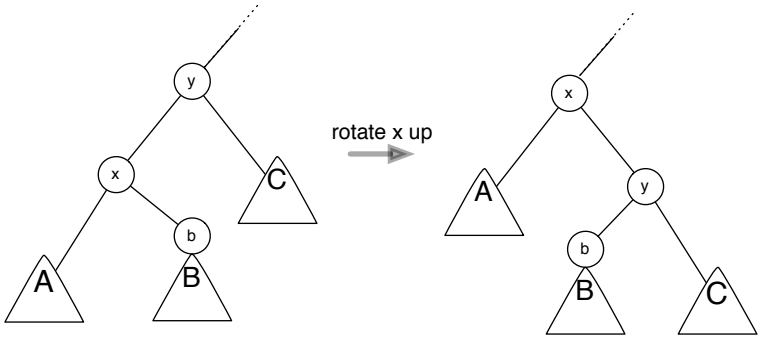

Figure 17.5: This figure helps us derive the rule for updating Delta representations in a rotation.

We have the following equations.

$$\begin{aligned} w(y) &= \Delta w(y) + \hat{w} \\ w(x) &= \Delta w(x) + \Delta w(y) + \hat{w} \\ w(b) &= \Delta w(b) + w(x) \end{aligned}$$

Now consider the tree resulting from rotating x, which appears on the right in Figure 17.4. Note that the parent of x in this tree is the parent of y in the tree on the left. that the weight of the Using $\Delta w'(\cdot)$ to denote the values of the Δw decorations in this tree, we have the following equations.

$$\begin{aligned} w(x) &= \Delta w'(x) + \hat{w} \\ w(y) &= \Delta w'(y) + w(x) \\ w(b) &= \Delta w'(b) + w(y) \end{aligned}$$

We can use the equations to derive rules for updating the $\Delta w(\cdot)$ decorations:

$$\begin{aligned}
\Delta w'(x) &= w(x) - \hat{w} \\
&= \Delta w(x) + \Delta w(y) \\
\Delta w'(y) &= w(y) - w(x) \\
&= -\Delta w(x) \\
\Delta w'(b) &= w(b) - w(y) \\
&= \Delta w(b) + \Delta w(x)
\end{aligned}$$

leading to the update

$$\Delta w'(x), \Delta w'(y), \Delta w'(b) = \Delta w(x) + \Delta w(y), -\Delta w(x), \Delta w(b) + \Delta w(x)$$

The same technique can be used for preserving *flipped*$(\cdot)$.

17.1.7 Updating $\Delta \min$ representations in a rotation

The rotation changes the descendants of x and y. Therefore, after the updates to $\Delta w(\cdot)$ have been made, if $\Delta \min(\cdot)$ is being maintained then $\Delta minw(x)$ and $\Delta minw(y)$ must be updated using Equation 17.2.

17.2 Splay trees

In moving x to the root, there is some freedom in selecting the rotations. The *splay tree* data structure [Sleator and Tarjan, 1985] specifies some rules for selecting these rotations so as to ensure that the time required is small. Moving a node x to the root following these rules is called *splaying x to the root*. We will prove a theorem that implies a bound on the total number of rotations required for many such splayings. Each rotation can be done in constant time, so we will obtain a time bound.

We define three operations performed on a node x in a BST, called *splay operations*. Each splay operation consists of one or two rotations, and each operation moves the node x closer to the root. Which operation should be applied depends on the relationship between x and its parent and grandparent (if they exist). are

- If x has no grandparent and is the left child of its parent, we perform a *zig* operation, which simply rotates x up. (See Figure 17.6.)
- Suppose x has a grandparent. If x is the left child of its parent and its parent is the left child of its grandparent, or if x is the right child of its parent and its parent is the right child of its grandparent, we perform a *zig-zig* operation: first rotate up the parent of x and then rotate up x. (See Figure 17.7.)

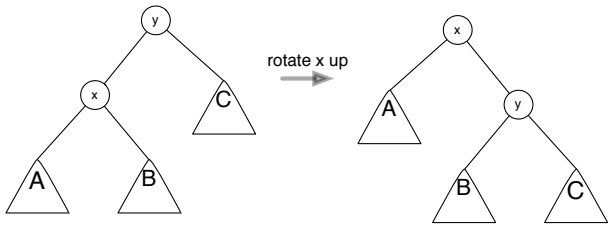

Figure 17.6: The zig step

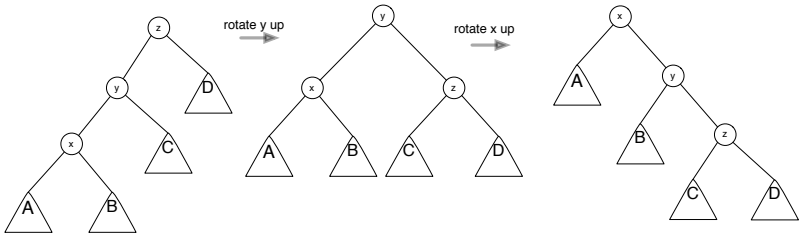

Figure 17.7: The zig-zig step.

- If x is the left child of its parent and its parent is the right child of its grandparent, or vice versa, we perform a *zig-zag* operation: first rotate up x and then again rotate up x. (See Figure 17.8.)

Splaying a node x to the root consists of repeatedly applying the applicable splay operation until x is the root.

17.2.1 Potential Functions for Amortized Analysis

The analysis of splay trees will show that each operation takes amortized $O(\log n)$ time: over a sequence of many operations on the tree, the average time per operation is $O(\log n)$. Each individual operation can take $O(n)$ time in the worst case.

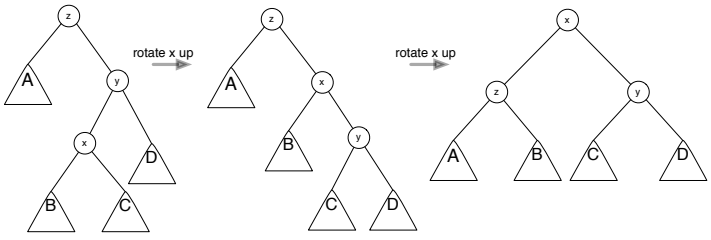

Figure 17.8: The zig-zag step

To perform this analysis we will associate a potential Φ with the state of the current tree. Typically, Φ is chosen in a way that it is bounded initially (by an amount that will be useful in the analysis of the running time) and always remains nonnegative.

The actual cost of an operation is defined so as to be a constant times the actual time required to perform the operation. The virtual cost is the actual cost plus the change in potential: $\Delta\Phi = \Phi(\text{after}) - \Phi(\text{before})$. Summing the virtual costs over a (long) sequence of operations to the tree gives:

$$\begin{aligned}\sum \text{virtual costs} &= \sum \text{actual cost} + \sum \Delta\Phi \\ &= \sum \text{actual costs} + \Phi(\text{finally}) - \Phi(\text{initially})\end{aligned}$$

where the last step is obtained by telescoping the sum $\sum \Delta\Phi$. Therefore the total actual cost is bounded as

$$\begin{aligned}\sum \text{actual cost} &= \sum \text{virtual costs} + \Phi(\text{initially}) - \Phi(\text{finally}) \\ &\leq \sum \text{virtual costs} + \Phi(\text{initially})\end{aligned}$$

using the fact that Φ(finally) is nonnegative.

17.2.2 Analysis of splay trees

A function $f : [\alpha, \beta] \longrightarrow \mathbb{R}$ is *monotone nondecreasing* if $f(x) \leq f(y)$ for all $x \leq y$.

Fact 17.2.1. *If the first derivative of f is nonnegative then the function is monotone nondecreasing.*

A continuous function $f : [\alpha, \beta] \longrightarrow \mathbb{R}$ is *concave* if $\frac{f(x)+f(y)}{2} \leq f(\frac{x+y}{2})$ for all $x, y \in [\alpha, \beta]$ (i.e. if the average of the images of x and y under f is at most the image of the average of x and y).

Fact 17.2.2. *If the second derivative of a function is nonpositive over some interval then the function is concave over that interval.*

For example, a little calculus shows that $\log x$ and $\sqrt{x}$ are monotone nondecreasing and concave.

Lemma 17.2.3. $\max\{\log u + \log v \,:\, u, v \geq 0,\, u + v \leq 1\} \leq -2$

Proof. Applying the definition of concavity with $x = 2u, y = 2v$, we obtain

$$\frac{\log(2u) + \log(2v)}{2} \leq \log(u + v)$$

Applying the definition of monotonicity with $x = u + v$ and $y = 1$, we obtain

$$\log(u + v) \leq \log(1)$$

These two inequalities yield

$$2 + \log u + \log v \leq 0$$

□

Define the *rank* of a node in a BST to be the base-2 logarithm of the number of descendants of that node. We denote the rank of x in T by $r(x, T)$. Define the *potential* of a BST T to be the sum of the ranks of its vertices. We denote the potential of T by $\Phi(T)$.

Define the *actual cost* of a splay operation to be the number of rotations it performs, i.e. 1 for a zig operation and 2 for a zig-zig operation and a zig-zag operation. Define the *virtual cost* of a splay operation to be the actual cost plus the resulting change in the potential in the BST.

Lemma 17.2.4. *Let T be a BST, and let x be a nonroot node. Let T' be the BST resulting from performing a single splay operation on x.*

- *If the operation was a zig, the virtual cost of the operation is at most $1 + 3(r(x, T') - r(x, T))$.*
- *Otherwise, the virtual cost is at most $3(r(x, T') - r(x, T))$.*

Proof. For brevity, we denote $r(x, T)$ by $r(x)$ and $r(x, T')$ by $r'(x)$. We denote the parent of x by y. If x has a grandparent, we denote it by z.

zig: The virtual cost is given by:

$$\begin{aligned} 1 + \Phi(T') - \Phi(T) &= 1 + r'(x) - r(x) + r'(y) - r(y) \\ &\leq 1 + r'(x) - r(x) && \text{since } r'(y) < r(y) \\ &\leq 1 + 3(r'(x) - r(x)) && \text{since } r'(x) > r(x) \end{aligned}$$

zig-zig: The virtual cost is given by:

$$\begin{aligned} 2 + \Phi(T') - \Phi(T) &= 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \\ &= 2 + r'(y) + r'(z) - r(x) - r(y) \text{ since } r'(x) = r(z) \\ &\leq 2 + r'(x) + r'(z) - r(x) - r(x) \text{ since } r'(y) \leq r'(x) \text{ and } r(y) \geq r(x) \end{aligned}$$

The following inequalities are equivalent:

$$\begin{aligned} 2 + r'(x) + r'(z) - r(x) - r(x) &\leq 3(r'(x) - r(x)) \\ \iff 2 &\leq 2r'(x) - r'(z) - r(x) \\ \iff -2 &\geq -2r'(x) + r'(z) + r(x) \\ \iff -2 &\geq r(x) - r'(x) + r'(z) - r'(x) \end{aligned}$$

For a node v, let $D(v)$ be the set of descendants of v in T, and let $D'(v)$ be the set of descendants in T'. By the definition of rank, the last inequality is equivalent to

$$-2 \geq \log \frac{|D(x)|}{|D'(x)|} + \log \frac{|D'(z)|}{|D'(x)|}$$

Since $D(x) \cup D'(z) \subseteq D'(x)$ and $D(x) \cap D'(z) = \emptyset$, $\frac{|D(x)|}{|D'(x)|} + \frac{|D'(z)|}{|D'(x)|} \leq 1$. Lemma 17.2.3 therefore implies that $\log\left(\frac{d(x)}{d'(x)}\right) + \log\left(\frac{d'(z)}{d'(x)}\right) \leq -2$.

zig-zag: The virtual cost is given by:

$$\begin{aligned} 2 + \Phi(T') - \Phi(T) &= 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \\ &= 2 + r'(y) + r'(z) - r(x) - r(y) \\ &\quad \text{since } r'(x) = r(z) \\ &\leq 2 + r'(y) + r'(z) - 2r(x) \\ &\quad \text{since } r(y) > r(x) \end{aligned}$$

$$\begin{aligned} & 2 + r'(y) + r'(z) - 2r(x) \leq 2(r'(x) - r(x)) \\ \iff & 2 \leq 2r'(x) - r'(y) - r'(z) \\ \iff & -2 \geq r'(y) - r'(x) + r'(z) - r'(x) \\ \iff & -2 \geq \log\tfrac{d(x)}{d'(x)} + \log\tfrac{d'(z)}{d'(x)} \end{aligned}$$

The last line is true by the same argument used for the zig-zig case. □

Corollary 17.2.5. *Let T be a BST, and let x be a nonroot node. Let T' be the BST resulting from splaying x to the root. The total virtual cost at most $3(r(x,T') - r(x,T)) + 1$.*

Proof. Suppose k splay steps are required to splay x to the root. If there is any zig step, it is the last step. For $i = 0, 1, \ldots, k$, let r_i be the rank of x after i splay steps. For $i < k$, the virtual cost of the i^{th} splay step is at most $3(r_i - r_{i-1})$. For $i = k$, the virtual cost is at most $3(r_i - r_{i-1}) + 1$. Using telescoping sums, we infer that the total is $3(r_k - r_0) + 1$. □

This can already be used to show a bound of $O(n \log n)$ on the actual cost of n splay operations on trees over n elements. However, in the data structure described in the next section, we use the specific form of the bound stated in Corollary 17.2.5.

17.2.3 Using splay trees to represent sequences and trees

Problem 17.2.6. *Using splay trees with Delta representations of $w(\cdot)$ and* minw$(\cdot)$*, show how to represent a sequence σ of n items with associated weights such that the following operations are supported in $O(\log n)$ time.*

- MinRange(u, v)*: find the item in the substring $\sigma[u, v]$ whose weight is minimum.*
- AddToRange(u, v, β)*: add β to the weight of all items in the substring $\sigma[u, v]$.*

Problem 17.2.7. *Describe a data structure for representing a sequence σ over S and an assignment of weights to the elements of S. The following operations should be supported in amortized $O(\log n)$ time per operation, where $n = |S|$:*

- *Given elements x and y, and a number β, add β to the elements of the substring $\sigma[x, y]$.*
- *Given elements x and y, reverse the substring $\sigma[x, y]$.*
- *Given number λ, find the leftmost element in σ whose weight is at most λ.*

Problem 17.2.8. *In this problem, we discuss a data structure that supports some interesting tree operations (but is simpler than the data structure we describe in the next section).*

Corresponding to each edge of a graph, there are two darts, one oriented in each direction. (Darts will come up often in the remainder of the book.)

Given any tree (indeed, any connected graph), there is a closed path that uses every dart exactly once. (There might be many such paths.) We represent a tree by the sequence σ of darts, and we represent the sequence σ by a splay tree whose vertices are the darts.

We represent a rooted forest by a set of such splay trees, one for each tree. Give pseudocode for each of the following operations, and then show that the amortized time per operation is $O(\log n)$ where n is the maximum number of edges.

- REMOVE(e)*: remove an edge e from the forest.*
- ADD(e, d_1, d_2)*: add an edge e to the forest between the tail of dart d_1 and the tail of dart d_2.*
- ANCESTOR(e_1, e_2)*: return* true *if the e_1-to-root path in the tree containing e_1 contains e_2.*

Problem 17.2.9. *Augment the data structure of Problem 17.2.8 to represent a labeling $d(\cdot)$ of the vertices by numbers so as to support the following operations (in addition to those specified in Problem 17.2.8):*

- GETVALUE(e)*: return the label of the child endpoint of e.*
- ADD(e, β)*: add β to the label of every node in the subtree rooted at the child endpoint of e.*

17.3 Representation of link-cut trees

As we have seen, splay trees can be used for representing sequences. A sequence corresponds in graph theory to a path. We need to represent rooted forests more generally. We use the *link-cut* trees of [Sleator and Tarjan, 1983,

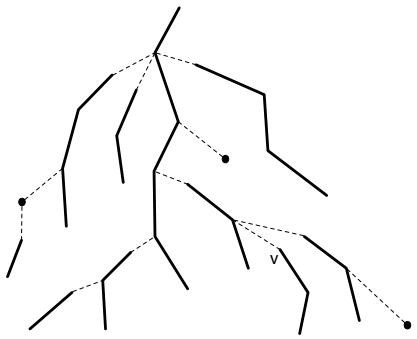

Figure 17.9: A rooted tree divided into solid paths.

Sleator and Tarjan, 1985]. A rooted tree is represented as a collection of paths joined together.

To choose a representation, we will designate each arc of the tree as either dashed or solid. The maximal paths consisting of solid arcs are called *solid* paths. We will maintain the property that each node has at most one solid child-arc. An example is illustrated in Figure 17.9.

We will make use of the operation of *exposing* a node u:

> Ensure that the solid path containing v contains all u's ancestors by converting dashed arcs along the path to solid and solid arcs incident to the path to dashed.

The operation is illustrated in Figure 17.10. Note that exposing u might leave u with a solid child edge.

In preparation for the analysis of the *expose* operation, we define an arc from u to its parent v to be *heavy* if $d(u) > \frac{1}{2}d(v)$ and *light* otherwise.

Lemma 17.3.1. *Every node v has at most one heavy child arc and at most $\lfloor \log n \rfloor$ light ancestor arcs.*

17.3.1 High-level analysis of the *expose* operation

Splicing a dashed arc uv means converting uv to a solid arc and converting the other solid arc entering v (if there is one) to a dashed arc. See Figure 17.11 for an illustration.

The operation of exposing a node u can be performed by performing a series of splices and, if u has a solid child arc, converting that arc to a dashed arc. Define the *actual cost* of an *expose* operation to be the number of splices.

Now we analyze the number of splices due to exposing u. Each splice converts

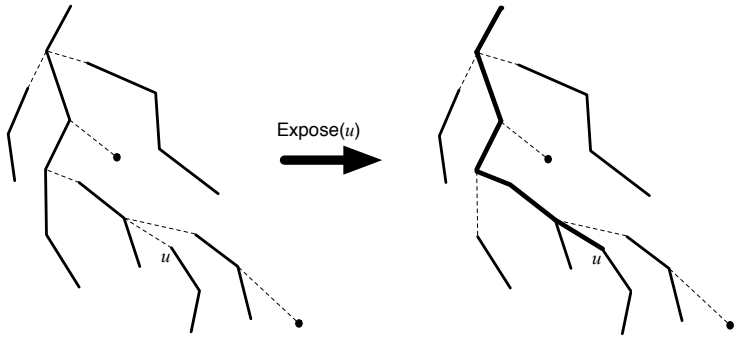

Figure 17.10: An example of exposing a node u. The effect is that the solid path containing u contains all of u's ancestors. In this figure, the heavy solid path comprises u's ancestors.

a dashed arc to solid.

number of splices

$$\begin{aligned} &= |\{\text{arcs converted from dashed to solid}\}| \\ &= |\{\text{light dashed arcs converted to solid}\}| + |\{\text{heavy dashed arcs converted to solid}\}| \end{aligned} \tag{17.3}$$

By Lemma 17.3.1, at most $\log n$ splices convert a light dashed arc to solid.

Let F denote a rooted forest in which each arc is designated either solid or dashed. Now we define the potential function, which we call Φ_A. Define $\Phi_A(F) = n - 1 - |\{\text{heavy solid arcs}\}|$.

Lemma 17.3.2. *When a node u is exposed, the number of splices plus the increase in Φ_1 is at most $1 + 2\log n$.*

Proof.

$$\begin{aligned} \text{virtual cost} &= \text{actual cost} + \text{increase in } \Phi_1 \\ &= \text{number of splices} + |\{\text{heavy solid arcs converted to dashed}\}| \\ &\quad -|\{\text{heavy dashed arcs converted to solid}\}| \\ &\leq \text{number of splices} + |\{\text{light dashed arcs converted to solid}\}| \\ &\quad -|\{\text{heavy dashed arcs converted to solid}\}| \\ &= 2 \cdot |\{\text{light dashed arcs converted to solid}\}| \text{ by (17.3)} \\ &\leq 2\log n \text{ by Lemma 17.3.1} \end{aligned}$$

□

Now we analyze the actual costs. Using the fact that the initial value of the potential function is at most $n - 1$, we obtain

Corollary 17.3.3. *For trees comprising n vertices, for at least $n - 1$ expose operations, the average number of splices per operation is at most $1 + 2 \log n$.*

17.3.2 Representation of trees

We will now describe how the data structure represents a forest of rooted trees, each decomposed into solid paths. We will use the term *abstract tree* to signify one of the trees represented by the data structure. We will use the term *concrete tree* to signify a tree that the data structure uses. When we refer to a node's parent/children/descendants/ancestors in the concrete tree, we will use the modifier *concrete*, as in *concrete parent* or *concrete descendants*.

The precise representations depends on which operations must be supported. Two categories of operations are *ancestor search* and *descendant search*. In the former, one searches the ancestors of a given node v, and in the latter one searches the descendants. Another operation, *eversion*, allows one to change the root.

The simplest implementation is the one supporting neither descendant search nor eversion, so we will describe that one first. Supporting descendant search involves introducing a new kind of node and an additional pointer per node. In either case, eversion is supported using a Delta representation of left-right as described in Section 17.1.4.

17.3.3 Link-cut trees that do not support descendant search

In the implementation of link-cut trees that do not support descendant search or eversion, the abstract tree is represented by a collection of splay trees that are linked together, as shown in Figure 17.12. Each node v has three pointer fields: parent(v), left(v), and right(v). If eversion is to be supported, each node v has a pointer field parent(v) and a two-element array children[$\cdot$] of pointers, and a Delta representation Δ*flipped*($\cdot$) of left-right.

Each solid path P in the abstract tree is represented in the concrete tree by a splay tree B over the vertices of P, called a *solid tree*, whose BST order coincides with the rootward order of vertices in P. For each node v in B, left(v) points to v's left child in B (or has the value *null* if v has no left child), and similarly for right(v). If v has a parent in B, then parent(v) points to the parent. For a node v in B, the children of v in B are called the *solid children* of v.

The *solid descendants* of v are those concrete descendants of v that are in the same solid tree. The *non-solid descendants* are those concrete descendants of v that are not in the same solid tree.

Next we specify the value of parent(v) in the case when v is the root of its solid tree B. There are two cases. Let x be the rootmost end of P. In the abstract tree, either x has a parent y or x is the root. In the former case, parent(v) is y. In the latter case, parent(x) is *null*.

Consider the former case. Note that y belongs to its own solid path in the abstract tree, and *left(y)* and *right(y)* are determined by y's position in the corresponding solid tree. Therefore x is neither *left(y)* nor *right(y)* even though parent$(x) = y$. We say in this case that x is a ~~*bastard*~~ *unacknowledged* child. The node y might have many unacknowledged children.

Note also that, although x is an (unacknowledged) child of y in the representing tree, x is not necessarily the child of y in the abstract tree.

To summarize, each solid path in the abstract tree is represented in the concrete tree by a solid tree, and each dashed arc xy is represented by an arc to y from the root of the solid tree containing x.

Question 17.3.4. *Write a procedure* IsSolidRoot(v) *that returns* true *if v is the root of its solid tree.*

17.3.4 Implementing the *expose* operation for trees not supporting descendant search

We give a procedure Expose that implements the *expose* operation. Since the implementation depends only in some details on whether the link-cut trees are to support descendant search, we will use a subroutine Expose_Step that encapsulates the differences. We will give one implementation for Expose_Step now, one that works for link-cut trees not supporting descendant search. Later we will give another implementation for Expose_Step, one that works for link-cut trees supporting descendant search. The code for Expose will not change.

We will use the same strategy to encapsulate code related to maintaining decoration invariants. The procedure RotateUp(u) rotates u up. It is used in splaying, and is also called directly in Expose. When a rotation takes place, decorations must be updated to preserve invariants. We address this issue when discussing decorations. For now, just assume that RotateUp(u) rotates u up as in Section 17.1.5.

The code for Expose_Step includes calls to two subroutines, SolidToDashed and DashedToSolid:

- SolidToDashed(v) is called when v is the left child of the root, and is about to become an unacknowledged child.
- DashedToSolid(u, v) is called when u is an unacknowledged child of v, and is about to become its left child.

These subroutines are "hooks". For now, you can assume that they do nothing. Later we will give implementations that preserve invariants on decorations.

```
def Expose(u):
1 splay u to the root of its solid tree
2 while u is not the root of the concrete tree,
3       # now u is at the root of its solid tree.
4       Expose_Step(u)# move u's parent to root of its solid tree
```

```
5    RotateUp(u)
```

```
def Expose_Step(u):
    pre: u is root of its solid tree, u is not root of its concrete tree
    post: u's parent is root of its solid tree, and u is its left child
    splay parent(u) to root of its solid tree
    v := left(parent(u)) # v might be null
    if u' ≠ null, SolidToDashed(v) # v is about to become an unacknowledged child of parent(u)
    DashedToSolid(u, parent(u)) # u is about to become a solid child of parent(u)
    left(parent(u)) := u
```

17.3.5 Analysis of Expose(u) for trees not supporting descendant search

We combine the technique from splay-tree analysis with the analysis in Section 17.3.1 of the number of splices required when a node u is exposed. Recall that $\Phi_A = n - 1 - |\{\text{heavy solid arcs}\}|$ is the potential function defined in Section 17.3.1. Note that Φ_A depends only on the abstract tree, which is why we use the subscript A. Define $\Phi_C = \sum_v r(v)$ where $r(v)$ is the base-2 logarithm of the number of descendants of v in its concrete tree. We use the subscript C to reflect the fact that Φ_C depends on the concrete tree. Define the overall potential function to be $\Phi = 2\Phi_A + \Phi_C$. The time required by a call to Expose is bounded by a constant times the number of rotations peformed. We therefore define the actual cost of Expose(u) to be the number of rotations performed. As usual, the virtual cost is the actual cost plus the increase in the potential function.

Lemma 17.3.5. *The virtual cost with respect to Φ of* Expose(u) *is at most* $3 + 8 \log n$.

Proof. Consider a call to Expose(u), and suppose it involves k iterations. The call does

- $k+1$ splayings, one in Step 1 and one in each invocation of Expose_Step,
- k splicings, each done in the last step of Expose_Step, and
- k additional rotations in Step 5 of Expose.

The splayings and the additional rotations do not change the abstract tree or its decomposition into heavy paths, and so do not affect Φ_A. The splicings do not change the number of descendants of any node, and so do not affect Φ_C.

Let $u{=}v_0, v_1, v_2, \ldots, v_k$ be the vertices splayed to the roots of their solid trees. For each v_i, let $r(v_i)$ be the rank of v_i before it is splayed, and let $r'(v_i)$ be its rank after the splaying.

By Corollary 17.2.5, the virtual cost with respect to Φ_C of all the splayings is at most

$$\sum_{i=0}^{k} 1 + 3(r'(v_i) - r(v_i)) \tag{17.4}$$

For $i = 0, 1 \ldots, k-1$, just after the splaying of v_i, the number of descendants of v_i is less than the number of descendants of v_{i+1}. The number of descendants of v_{i+1} does not subsequently change until just before v_{i+1} is splayed, which shows $r'(v_i) < r(v_{i+1})$. Therefore the value of (17.4) is bounded by $k + 1 + 3(r'(v_k) - r(v_0))$, which in turn is bounded by $k + 1 + 3\log n$.

Consider the rotations in Step 5 of EXPOSE. Let $r_i(u)$ be the rank of u after i iterations of Step 5. As in the analysis of *zig*, the virtual cost with respect to Φ_C of the i^{th} rotation is at most $1 + r_i(u) - r_{i-1}(u)$. Hence the total virtual cost with respect to Φ_C of these rotations is at most $k + r_k(u) - r_0(u)$, which is bounded by $k + \log n$.

Thus the total virtual cost with respect to Φ_C of EXPOSE(u) is at most $2k + 1 + 4\log n$. By Lemma 17.3.2, $k +$ increase in $\Phi_A \leq 1 + 2\log n$. Therefore the total virtual cost with respect to $\Phi = 2\Phi_A + \Phi_C$ is at most $2(1 + 2\log n) + (1 + 4\log n)$, which in turn is at most $3 + 8\log n$. □

In the next section, we describe two additional operations and we show that each has virtual cost $O(\log n)$ as well.

Since Φ is always nonnegative and never exceeds $2n + n\log n$, the amortized actual cost per call of m calls to EXPOSE is at most $3+8\log n+(2n+n\log n)/m$. In particular, if $m \geq n$ then the amortized actual cost per call is $O(\log n)$.

The following problem shows that, with care, one can ensure $O(\log n)$ actual cost per operation amortized over even fewer operations.

Problem 17.3.6. *Show that there is a constant d such that, for any n-node rooted binary tree T, there is a corresponding concrete tree for which $\Phi_C \leq dn$.*

17.4 Link-cut trees that support descendant search

This section needs more rewriting–it is a bit repetitive.

In order to support descendant search, a node's unacknowledged children must be partially acknowledged so that the tree search can descend from parent to an unacknowledged child. Since a prolific node might have many unacknowledged children (a common condition among celebrity vertices), they are organized using a splay tree of auxiliary vertices. We refer to these splay trees as *dotted trees*, and to the vertices comprising them as *dotted vertices*. We refer to the original vertices of the concrete tree, the ones representing vertices of the abstract tree, as *solid vertices*.

Each node v in the concrete tree, whether a solid node or a dotted node, has one additional pointer, middle(v). If v is a solid node, middle(v) either points to a dotted node or is null. If v is a dotted node, middle(v) points to a solid node.

As before, the solid vertices form splay trees, one per solid path of the abstract tree, and these splay trees are called *solid trees*. The dotted vertices also form trees, called *dotted trees*. The pointers parent($\cdot$), left($\cdot$), and right($\cdot$) are used to represent these trees in the usual way. If v is the root of a dotted tree, however, parent(v) points to a solid node y. If v is the root of a solid tree, parent(v) points to a dotted node (unless v is the root of the abstract tree, in which case parent(v) is null.

The dashed arcs in the abstract tree correspond to dotted vertices in the concrete tree. Consider a dashed arc uv in the abstract tree. The child u belongs to some solid path P. In the concrete tree, just as before, the solid path P is represented by a solid tree, a tree of solid vertices. The concrete parent of the root of that solid tree is not u (as before) but is instead a dotted node. That dotted node belongs to a tree of dotted vertices (a *dotted tree*), and the parent of the root of that dotted tree is v.

Furthermore, middle(v) points to the root of this dotted tree, and each dotted node points to the root of the corresponding solid tree, so the structure can be traversed top-down.

Problem 17.4.1. *Write pseudocode for* EXPOSE_STEP(u) *for trees with solid and dotted vertices. Next, analyze this variant:*

- *Write the potential function.*
- *Show that the virtual cost of* EXPOSE *is* $O(\log n)$.

Problem 17.4.2. *Extend the result in Problem 17.3.6 to handle the concrete representation that supports descendant search. That is, show that there is a constant d such that, for any n-node rooted binary tree T, there is a corresponding concrete tree consisting of solid and dotted vertices for which the value of the potential function is at most dn.*

17.5 Topological updates in link-cut trees

Now we describe two operations that modify the structure of a tree:

- CUT(u): given a node u that is not an abstract root, remove its parent edge from the forest, making u a root.
- LINK(u, v): given two vertices u, v in different trees such that u is the root of its tree, add the arc uv, making u a child of v.

We can implement these operations using EXPOSE(v). Again we include "hooks" in the code, LOSEPARENT, LOSERIGHT, and GAINPARENT, that we will later use to preserve invariants on decorations:

- LOSEPARENT(v) is called when v is the right child of the concrete root, and the arc from v to its parent is about to be severed.

- LoseRight(u) is called when u is the concrete root and is about to lose its right child.
- GainParent(v, u) is called when v and u are concrete roots and v is about to be made the right child of u.

```
def Cut(u):
    pre: u is not the root of its abstract tree
    Expose(u)
    # u is root of its concrete tree, and it has a right child
    LoseParent(right(u)) # right(u) is about to become root of its concrete tree
    LoseRight(u) # u is about to lose a child
    right(u), parent(right(u)) := null, null
```

```
def Link(u, v)
    pre: u is the root of its abstract tree.
    post: u is the child of v in their abstract tree.
    Expose(u)
    # u has no right child
    Expose(v)
    #v is a concrete root
    GainParent(v, u) # v is about to become the right child of u
    parent(v), right(u) := u, v
```

17.5.1 Analysis of link and cut operations

We consider the cost of the link and cut operations. In each operation, the Expose operations have virtual cost $O(\log n)$. Cut removes a solid arc. This causes at most an increase of 1 in Φ_A and can only decrease Φ_C. Thus the Cut operation has virtual cost $O(\log n)$.

In Link, after the Expose operations, u is the root of its concrete tree. Therefore making v the right child of u increases the rank of v but of no other node. This causes at most an increase of $\log n$ in Φ_C and can only decrease Φ_A. Thus the Link operation has virtual cost $O(\log n)$.

17.6 Weight updates for link-cut trees

We now describe how to represent an assignment of weights to vertices so as to facilitate quickly updating the weights of many vertices at a time.

We consider two operations for such bulk updates:

- AddToDescendants(u, α), which adds α to the weights of all the descendants of u.

- AddToAncestors(u, α), which adds α to the weights of all the ancestors of u.

For each of the above operations, we show how to represent the weights so as to support the operation. In each case, we avoid explicitly representing the weights; Instead, we use the Delta representation of weights described in Section 17.1.1. Each node v stores a *weight increment* $\Delta w(v)$ such that the weight of a node equals the sum of the weight increments of some of its ancestors. The exact invariant satisfied by $\Delta w(\cdot)$ depends on which bulk update is to be supported.

17.6.1 Supporting AddToDescendants

To support AddToDescendants, we maintain the following invariant:

> The weight of a node u is the sum $\sum_v \Delta w(v)$ over all ancestors v of u in the concrete tree.

Under this invariant, AddToDescendants is implemented as follows:

```
def AddToDescendants(u, α):
    Expose(u)
    # The concrete descendants of u that are not concrete descendants
      of u's right child are u's abstract descendants.
    Δw(u) += α
    if u has a right child, Δw(right(u)) −= α
```

To preserve the invariant, the rotation procedure RotateUp should be modified as in Section 17.1.6. The "hook" subroutines SolidToDashed and DashedToSolid of Section 17.3.4 don't need to do anything since the invariant does not distinguish between acknowledged and unacknowledged children.

To preserve the invariant during link and cut operations, we define GainParent and LoseParent as follows:

```
def GainParent(v, u)
    pre: v and u are roots of their concrete trees
         v is about to be made the right child of u
    Δw(v)− = Δw(u)
```

```
def LoseParent(v):
    # v is the right child of the concrete root, is about to be severed from parent
    Δw(v)+ = Δw(parent(v))
```

17.6.2 Supporting AddToAncestors

To support AddToAncestors, we use a similar invariant:

> The weight of a node u is the sum $\sum_v \Delta w(v)$ over all ancestors v of u in u's solid tree.

Unlike the invariant for AddToDescendants, here the sum is restricted to solid ancestors.

Under this invariant, AddToAncestors is implemented as follows:

```
def AddToAncestors(u, α):
    Expose(u)
    # The concrete descendants of u that are not concrete descendants
    # of u's left child are u's abstract ancestors.
    Δw(u) += α
    if u has a left child, Δw(left(u)) -= α
```

The step Expose(u) ensures that the solid path containing u contains all of u's ancestors in the abstract tree, so these ancestors of u are exactly the descendants of u in its solid tree that are not descendants of its left child (if it has one). Adding α to $\Delta w(u)$ increases by α the weight of all descendants of u in its solid tree, and subtracting α from $\Delta w(\text{left}(u))$ compensates so as to not increase the weights of solid descendants that are not abstract ancestors.

If we had used the same invariant as we used for AddToDescendants, adding α to $\Delta w(u)$ in AddToAncestors(u, α) would have had the effect of adding α to the weights of descendants of proper ancestors of u. This is why we needed to use a different invariant.

To preserve the invariant, we define RotateUp, GainParent, and LoseParent exactly as in Section 17.6.1. Since the invariant distinguishes between solid children and unacknowledged children, we must take care to preserve the invariant when changing edges from solid to dashed and vice versa. We therefore define procedures for the hooks SolidToDashed and DashedToSolid:

```
def SolidToDashed(u):
    pre: u is solid child of root of concrete tree, about to become dashed child
    Δw(u) += Δw(parent(u))
```

and

```
def DashedToSolid(u, v):
    pre: u is dashed child of v, about to become solid child
         v is root of concrete tree
    Δw(u) -= Δw(v)
```

17.6.3 Getting the weight of a node

The following procedure returns the weight of a given node v:

```
def GetWeight(v):
    Expose(v)
    return Δw(v)
```

The correctness of this procedure uses the fact that, once v is the root of its concrete tree, $\Delta w(v)$ is its weight.

17.7 Weight searches in link-cut trees

We now discuss how to provide support for searching a tree for a low-weight node. One form of search is to find a node with minimum weight among a set of vertices (descendants or ancestors). We break ties by choosing the *leafmost* or *rootmost* among those vertices of minimum weight. To indicate which tie-breaking rule to use, we use a parameter *dir*, which has value either L (for *leafmost*) or R (for *rootmost*).

- AncestorFindMin(u, *dir*): given a node u and a direction *dir*, find the *dir*most minimum-weight ancestor of u in the conceptual tree.
- DescendantFindMin(u, *dir*): given a node u and a direction *dir*, find the *dir*most minimum-weight descendant of u in the conceptual tree.

Make clear that dir=leafmost does not mean you have to prefer deeper to less deep among incomparable vertices.

We will see that each of the above search operations can be implemented using one of the following two operations, which are useful in their own right: finding a node having weight at most a given threshold:

- AncestorFindWeight(u, α, *dir*): given a node u, a number α, and a direction *dir*, return the *dir*most ancestor of u having weight at most α, or *null* if there is no such ancestor.
- DescendantFindWeight(u, α, *dir*): given a node u, a number α, and a direction *dir*, return the *dir*most descendant of u having weight at most α, or *null* if there is no such descendant.

The key to supporting searches is maintaining a representation of $minw(\cdot)$, defined for BSTs in Section 17.1.2. Here we define $minw(u)$ to be the minimum weight among *solid* descendants of u.

A Delta representation of $minw(\cdot)$ is used, as described in Section 17.1.3. The invariant is

> **Delta min invariant:** For each node u, $minw(u) = \Delta minw(u) + w(u)$

Under what circumstances must we update the $\Delta minw(\cdot)$ label of a node? We first note that no updates are needed if the structure of the concrete tree does not change:

- The invariant makes clear that if an operation does not change the structure of the concrete tree, and does not change $minw(u)$ or $w(u)$, the operation need not change $\Delta minw(u)$.

- In bulk updates, AddToDescendants(u, α) or AddToAncestors(u, α), α is added to the weights of either all the descendants of u (in case of AddtoDescendants) or the solid descendants of u (in case of AddToAncestors). In each case, doing so also adds α to the $minw(\cdot)$ values for the same set of vertices. Therefore no change needs to be made to $\Delta minw(v)$ for any node v.

The procedures that change the structure of the tree are RotateUp, Link, and Cut. In order to maintain $\Delta minw(\cdot)$, we need to add a call to a hook, ChildChange$(\cdot)$, at the end of these three procedures. RotateUp(u) changes the children of two vertices, u and its former parent, so ChildChange must be called twice—first for u's former parent (now u's child) and second for u.

Let u be a node, and let its solid children be $u_1, \ldots, u_k$. ChildChange(u) must update $\Delta minw(u)$ based on the values of $u_1, \ldots, u_k$ as given in Equation 17.2.

17.8 Supporting ancestor searches

17.8.1 FindSolid

For ancestor search, we use an auxiliary procedure SolidFind(u, α, dir) that takes as input a node u, a number α such that

$$\alpha \geq \Delta minw(u) \tag{17.5}$$

and a direction *dir* (L or R). In this case, L signifies *left* and R signifies *right*. The procedure returns the *dir*most solid descendant v of u such that $w(v) \leq \alpha + w(u)$.

By the Delta min invariant, the precondition 17.5 is equivalent to the condition

$$\alpha + w(u) \geq minw(u) \tag{17.6}$$

```
def SolidFind(u, α, dir):
    pre: α ≥ Δminw(u)
    post: Returns the dirmost solid descendant v such that w(x) ≤ α + w(u).
1   u1, u2 := left(u), right(u) if dir == left else right(u), left(u)
2   if u1 ≠ null and α − Δw(u1) ≥ Δminw(u1),
```

```
3    return SolidFind(u_1, α − Δw(u_1), dir)
4   if α ≥ 0
5    splay u to the root of its solid tree
6    return u
7   return SolidFind(u_2, α − Δw(u_2), dir)
```

First we prove its correctness. By 17.6 (equivalent to the precondition) and the definition of $\min(u)$, there exists a solid descendant v of u such that $w(v) \leq \alpha + w(u)$. As in Step 1, define

$$u_1, u_2 := \text{left}(u), \text{right}(u) \text{ if } dir == \text{left else right}(u), \text{left}(u)$$

One of the following cases must hold:

1. u_1 has a solid descendant v of weight at most $\alpha + w(u)$,
2. the weight of u itself is at most $\alpha + w(u)$, or
3. u_2 has a solid descendant v of weight at most $\alpha + w(u)$.

Case 1 holds if $u_1 \neq null$ and $minw(u_1) \leq \alpha + w(u)$. By the Delta min invariant, the latter inequality holds if $\Delta minw(u_1) + w(u_1) \leq alpha + w(u)$, which holds if $\Delta minw(u_1) \leq \alpha - \Delta w(u_1)$. The condition in Step 2 therefore tests whether Case 1 holds. Moreover, if that condition holds then the precondition for the recursive call in Step 3 is satisfied.

Case 2 holds if $w(u) \leq \alpha + w(u)$, so the condition in Step 4 tests whether this case holds.

Finally, if neither Case 1 nor Case 2 holds then Case 3 must hold, so $minw(u_2) \leq \alpha + w(u)$, so $\Delta minw(u_2) + w(u_2) \leq \alpha + w(u)$, so $\Delta minw(u_2) \leq \alpha - \Delta w(u_2)$. Therefore in this case the precondition for the recursive call in Step 7 is satisfied.

Note that, in case multiple descendants v satisfy the inequality, the *dir*most descendant is returned. The correctness of the procedure therefore follows by induction on the depth of recursion.

Now we consider the time required by the procedure. (Note that the procedure is tail-recursive, so can easily be implemented iteratively.) The depth of recursion equals the depth in in the solid tree of the node returned. In Step 5, the node returned is splayed to the root of its solid tree. We define the actual cost of the operation to be the actual cost of this splaying. It follows that the true running time is at most a constant times the actual cost. We know that the virtual cost of the splaying (and therefore of the whole procedure) is $O(\log n)$.

17.8.2 AncestorFindWeight and AncestorFindMin

Now we write AncestorFindWeight(u, α, dir). Recall that the goal is to return u's *dir*most abstract ancestor whose weight is at most α. Say a node is a *candidate* if it is an abstract ancestor whose weight is at most α.

First the procedure exposes u so that it is the concrete root. Its ancestors consist of it and the descendants in its right subtree. The procedure next determines whether u itself is a candidate by checking whether its weight is at most α. Since u is the concrete root, its weight is $\Delta w(u)$.

Next the procedure checks whether the right subtree contains a candidate. It does this by checking whether there are any vertices in the right subtree (right$(u) \neq$ *null*) and, if so, whether $minw(\text{right}(u)) \leq \alpha$. By the Delta min invariant, $minw(\text{right}(u)$ equals $\Delta minw(\text{right}(u)) + w(\text{right}(u))$. By the invariant for $\Delta w(\cdot)$, $w(\text{right}(u)) = \Delta w(\text{right}(u)) + \Delta w(u)$, so the procedure checks whether $\Delta minw(\text{right}(u) + \Delta w(\text{right}(u)) + \Delta w(u) \leq \alpha$.

If u is a candidate and either we seek the leftmost candidate or there are no candidates to the right, the procedure returns u. Otherwise, the procedure uses SolidFind to find the *dir*most candidate in the right subtree (unless the right subtree has not candidate, in which case the procedure returns *null*).

```
def AncestorFindWeight(u, α, dir):
    Expose(u)
    u_is_candidate := Δw(u) ≤ α
    candidate_on_right := right(u) ≠ null and Δminw(right(u)) + Δw(right(u)) + Δw(u) ≤ α
    if u_is_candidate and (dir==L or not candidate_on_right),
        return u
    if not candidate_on_right, return null
    return SolidFind(right(u), α − Δw(right(u)) − Δw(u), dir)
```

Problem 17.8.1. *Write the procedure* AncestorFindMin *in terms of* SolidFind. *Argue that your procedure is correct.*

17.8.3 Supporting descendant searches

To support descendant searches, we use dotted vertices as described in Section 17.4. Since the dotted vertices do not have weights, they do not need $\Delta w(\cdot)$ decorations. Since they do not belong to solid trees, they do not need $\Delta minw(\cdot)$ decorations. However, to facilitate searching among them, we do need a decoration. For each dotted node x, we define $\widehat{minw}(x)$ to be the minimum weight among all concrete descendants of x. For each solid node u, we define $\widehat{minw}(u)$ to be the minimum weight among all concrete descendants of u that are not in the same solid tree as u. In either case, if there are no eligible descendants, the value is ∞.

Whether we use an explicit or Delta representation of $\widehat{minw}(\cdot)$ depends on whether we need to support ancestor updates or descendant updates.

Representing $\widehat{minw}(\cdot)$ for descendant searches with ancestor updates

To support ancestor updates, $\widehat{minw}(\cdot)$ is represented explicitly as a decoration on each of the vertices, both solid and dotted. We describe how it can be updated when a node's children change.

For a node pointer p, define

$$\widehat{minw}^*(p) = \begin{cases} \infty & \text{if } p = null \\ \widehat{minw}(p) & \text{otherwise} \end{cases}$$

For a solid node u, we update $\widehat{minw}(u)$ by

$$\widehat{minw}(u) := \min\{\widehat{minw}^*(\text{left}(u)), \widehat{minw}^*(\text{right}(u)), \widehat{minw}^*(\text{middle}(u))\}$$

For a dotted node x, we update $\widehat{minw}(x)$ by

$$\widehat{minw}(x) := \min\{\widehat{minw}^*(\text{left}(x)), \widehat{minw}^*(\text{right}(x)), \widehat{minw}(\text{middle}(x)), \Delta minw(\text{middle}(x))\}$$

These updates should be incorporated into ChildChange$(\cdot)$ for solid and dotted vertices.

Clarify interpretation of L/R for trees with middle children.

Problem 17.8.2. *Write a procedure* Find*, analogous to* SolidFind*, that searches among all the descendants of u, not just the solid descendants, for the* dir*most descendant having weight at most a given amount. You can assume that $\widehat{\text{minw}}(\cdot)$ is represented explicitly. Your search procedure will need to use both $\widehat{\text{minw}}(\cdot)$ and $\Delta\min(\cdot)$.*

Clarify that Find can assume there is such a descendant. Specify the given amount to be given explicitly or in terms of w(u) as in FindSolid?

Problem 17.8.3. *Write* DescendantFindMin *in terms of* Find*. You can assume that $\widehat{\text{minw}}(\cdot)$ is represented explicitly.*

Maybe should be two mutually recursive procedures, one for solid vertices and one for dashed.

Representing $\widehat{minw}(\cdot)$ for descendant searches with descendant updates

To support descendant updates, we use a Delta representation of $\widehat{minw}(\cdot)$. Each solid node u has a decoration $\Delta\widehat{minw}(u)$ such that

> **Delta $\widehat{\min}$ invariant for solid vertices:** $\widehat{minw}(u) = \Delta\widehat{minw}(u) + w(u)$

Each dotted node x has a decoration $\Delta\widehat{minw}(x)$ such that

> **Delta $\widehat{\min}$ invariant for dotted vertices:** $\widehat{minw}(x) = \Delta\widehat{minw}(x) + w(u)$ where u is x's closest solid ancestor

Need to discuss how to represent dart-weights with auxiliary vertices, including evert.

MORE DETAILS ARE NEEDED HERE

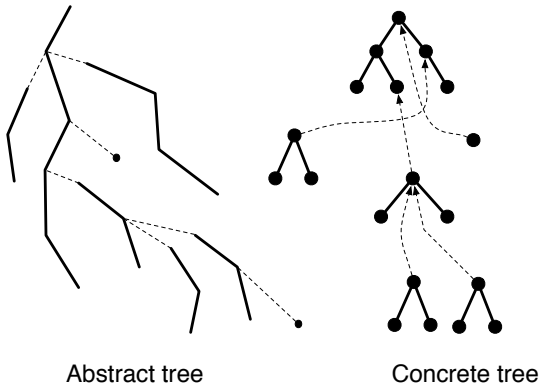

Figure 17.12: On the left is a diagram of an abstract tree. On the right is the corresponding concrete tree.

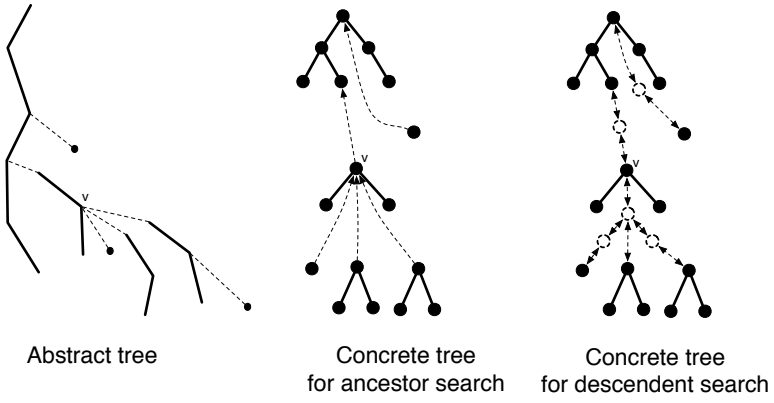

Figure 17.13: This figure shows two concrete trees corresponding to the same abstract tree. The middle diagram shows a concrete tree with the structure described in Section 17.3.3. The right diagram shows a concrete tree that supports descendant search. Each solid node has a *middle* pointer that either is null or points to a dotted node, the root of a BST of dotted vertices. Each dotted node has a middle pointer that points to a solid node. The dotted lines between the vertices have arrowheads on both ends because these lines represent pointers in both directions.Consider the node v of the abstract tree. It has one child in its solid path and three children not in its solid path. In the concrete tree in the middle diagram, v has three unacknowledged children, which are the roots of solid trees representing solid paths. In the concrete tree in the right diagram, v has one middle child, a dotted node, the root of a BST of dashed vertices, each of which in turn has as its middle node the root of a solid tree.