Chapter 18

Appendix: Splay trees and link-cut trees

In this Chapter we provide data structures for representing weighted sequences and weighted trees. These data structures are used in Chapters 7 and 10. In fact, the set of operations supported by the data structures we describe is more general than those required by the algorithms in this book.

We start by stating a theorem on representing disjoint sequences.

Theorem 18.0.1. *There is a data structure that, for a finite n-element set Ω, represents a set of disjoint sequences consisting of the elements of Ω and an assignment of weights to these elements. The data structure has size $O(n)$ and supports each of the following operations in $O(\log n)$ amortized time per operation:*

- First(x)*: return the first element of the sequence containing x.*
- Last(x)*: return the last element of the sequence containing x.*
- Split(x)*: split the sequence σ containing x into two sequences $\sigma[\cdot, x)$ and $\sigma[x, \cdot]$.*
- Concatenate(x, y)*: assuming x and y belong to distinct sequences σ_1 and σ_2, replace those sequences with the sequence $\sigma_1\sigma_2$.*
- GetWeight(x)*: return the weight of element x.*
- AddToRange(x, y, β)*: assuming x and y belong to a sequence σ, add β to the weights of all elements of the substring $\sigma[x, y]$.*
- Reverse(x, y)*: assuming x and y belong to a sequence σ, modify σ by replacing the substring $\sigma[x, y]$ with its reverse.*
- MinRange(x, y)*: assuming x and y belong to a sequence σ, return the leftmost element with minimum weight in the substring $\sigma[x, y]$.*

- FINDWEIGHT(x, y, λ): *return the leftmost element with weight at most* λ *in the substring* $\sigma[x, y]$.

The proof of this theorem appears in Sections 18.1 and 18.2.

Next we state a theorem on representing disjoint rooted trees.

Theorem 18.0.2. *There is a data structure, that, for a finite* n*-element set* Ω*, represents a collection of disjoint rooted trees whose nodes are the elements of* Ω*, and an assignment of weights to these elements. The data structure has size* $O(n)$ *and supports each of the following operations in* $O(\log n)$ *amortized time per operation:*

- ROOT(u): *given a node* u*, return the root of the tree containing* u.
- CUT(u): *given a node* u *that is not a root in* F*, remove its parent edge, making* u *the root of its subtree.*
- LINK(u, v): *given two vertices* u, v *in different trees of* F *such that* u *is the root of its tree, add the edge* uv*, making* u *a child of* v.
- EVERT(u): *make* u *the root of the tree containing* u.
- GETWEIGHT(u): *return the weight of node* u.
- ADDTODESCENDANTS(u, α): *add* α *to the weights of all the descendants of* u.
- ADDTOANCESTORS(u, α): *add* α *to the weights of all the ancestors of* u.
- ANCESTORFINDMIN(u, dir): *given a node* u *and a direction* dir$\in\{$*root, leaf*$\}$*, find the* dir*most minimum-weight ancestor of* u.
- DESCENDANTFINDMIN(u, dir): *given a node* u *and a direction* dir$\in\{$*root, leaf*$\}$*, find the* dir*most minimum-weight descendant of* u.
- ANCESTORFINDWEIGHT(u, α, dir): *given a node* u*, a number* α*, and a direction* dir$\in\{$*root, leaf*$\}$*, 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$\in\{$*root, leaf*$\}$*, return the* dir*most descendant of* u *having weight at most* α*, or* null *if there is no such descendant.*

The proof of the theorem starts in Section 18.3.

Theorem 18.0.2 can be used as a black box to support edge weights $w : E \to \mathbb{R}$. This can be done by adding an artificial node v_e in the middle of each edge e, and storing the weight $w(e)$ of e in the artificial node v_e (the weight of the non-artificial nodes is set to a sufficiently large value so they do not affect any of the operations).

For some applications such as MSSP and single-source single-sink maximum flow, one needs a variant of Theorem 18.0.2 that supports dart weights. In

this variant, operations/queries on ancestors/descendants additionally specify whether they relate to the rootward darts or to the leafward darts of the ancestor/descendant edges. Without EVERT, this can be easily supported by storing two weights for each edge, one for each dart. Supporting this variant with EVERT(u) is less trivial, because eversion swaps the leafward and rootward darts along the path from the new root u to the old root. We shall explain how to implement this variant in a non black-box manner after proving theorem 18.0.2.

Problem 18.1. *Show how to use Lemma 18.0.2 to support in* $\log n$ *time the operation* $\mathrm{LCA}(u, v)$*, which returns the lowest common ancestor of nodes* u *and* v *that belong to the same tree in* F*.*

18.1 Binary Search Trees

The proof of Theorem 18.0.1 is based on *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* and *inorder*). 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. To represent a sequence σ over some finite set S, we use a BST whose vertices are the elements of S; the BST order gives the sequence 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.

18.1.1 Delta Representation of weights

Rather than store explicit values at each node representing the item's weight, the data structure uses an implicit representation. It represents 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 18.1 for an example.

This representation allows the data structure 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 $\Delta w[v]$.

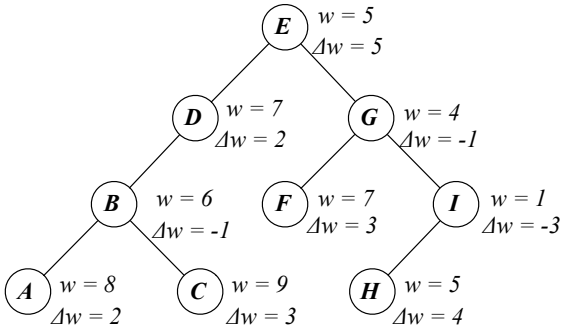

Figure 18.1: A BST with node-weights. The weight of each node is indicated by w. The difference between w for a node and w for its parent is indicated by Δw. Note that, for each node, w for that node can be computed by summing Δw over all the ancestors of that node.

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

18.1.2 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 $v_1, \ldots, v_k$ ($k \leq 2$),

$$minw(v) = \min\{w(v), \min_i minw(v_i)\} \tag{18.1}$$

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

Problem 18.2. *Write pseudocode in terms of* minw($\cdot$) *for* SearchLeftmost(λ). *The time required should be proportional to the depth of the tree.*

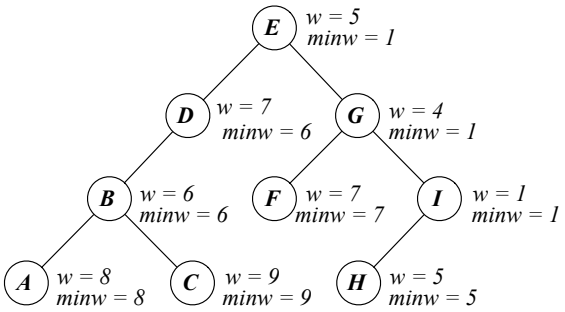

Figure 18.2: The weight of each node is indicated by w. The minimum weight of the subtree rooted at a node is indicated by *minw*.

18.1.3 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 18.3.

When we add β to the weights of descendants of u by adding β to $\Delta w[u]$, 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.

$$\begin{aligned}
minw(u) &= \min\{w(u), minw(u_1), \ldots, minw(u_k)\} \\
\Delta minw(u) &= minw(u) - w(u) \\
&= \min\{w(u) - w(u), minw(u_1) - w(u), \ldots, minw(u_k) - w(u)\} \\
&= \min\{0, (\Delta minw[u_1] + w(u_1)) - w(u), \ldots, (\Delta minw[u_k] + w(u_k)) - w(u)\} \\
&= \min\{0, \Delta minw[u_1] + \Delta w[u_1], \ldots, \Delta minw[u_k] + \Delta w[u_k]\} \qquad (18.2)
\end{aligned}$$

18.1.4 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 node. For example, the BST in Figures 18.2 and 18.1 represents the sequence $ABCDEFGHI$. We wish to modify this sequence by reversing the subsequence $FGHI$, obtaining $ABCDEIHGF$.

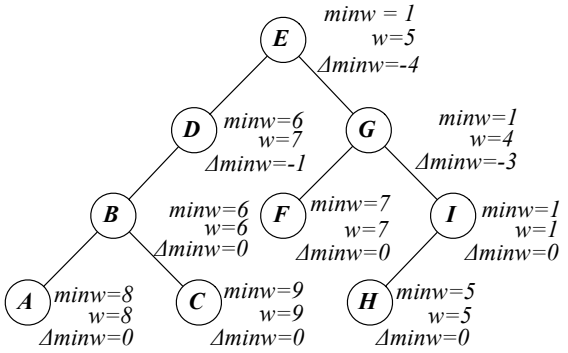

Figure 18.3: The minimum weight of the subtree rooted at a node is indicated by *minw*. The weight of each node is indicated by w. The difference between *minw* and w for a node is indicated by $\Delta minw$.

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*$[v]$.

18.1.5 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 18.4. 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.

18.1.6 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 18.5. 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).

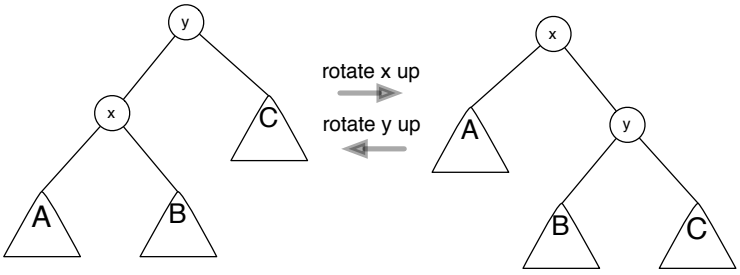

Figure 18.4: Starting with the tree on the left, rotating x up yields the tree on the right. Starting with the tree on the right, rotating y up yields the tree on the left. The triangles represent subtrees that are not changed by the rotation.

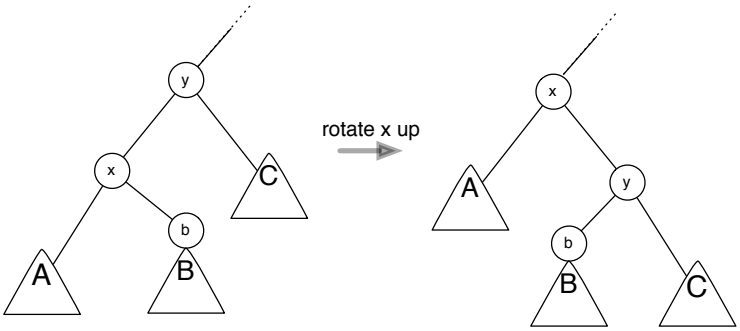

Figure 18.5: This figure helps us derive the rule for updating Delta representations in a rotation.

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 18.4. Note that the parent of x in this tree is the parent of y in the tree on the left. 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*$(\cdot)$.

18.1.7 Updating Δ*minw* 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 $\Delta minw[\cdot]$ is being maintained then $\Delta minw[x]$ and $\Delta minw[y]$ must be updated using Equation 18.2.

18.2 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).

- 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 18.6.)

- 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 18.7.)

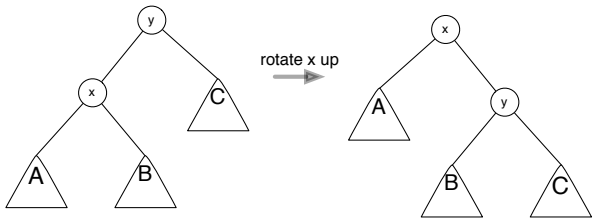

Figure 18.6: The zig step

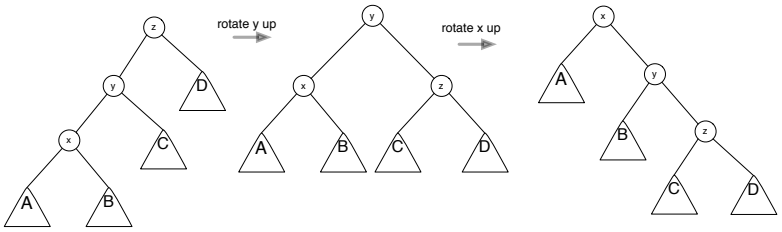

Figure 18.7: The zig-zig step.

- 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 18.8.)

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

18.2.1 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.

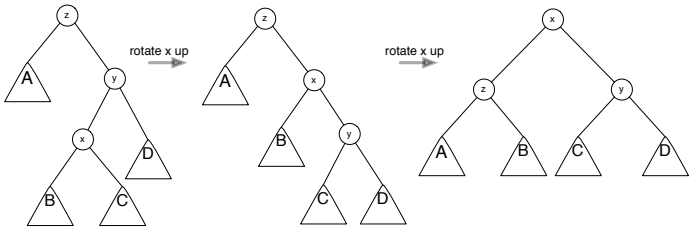

Figure 18.8: The zig-zag step

To perform this analysis we will associate a potential Φ with the state of the current tree. As is traditional in analyses of this kind, 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:

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

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

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

using the fact that $\Phi(\text{finally})$ is nonnegative.

18.2.2 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 18.2.1. *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 18.2.2. *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 18.2.3. $\max\{\log u + \log v \,:\, u, v \geq 0,\, u + v \leq 1\} \leq -2$

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 18.2.4. *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 18.2.3 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:

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

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

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

Corollary 18.2.5. *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 18.2.5.

The above discussion of splay trees and Delta-representation of $w(\cdot)$ and of *minw* can be easily used to prove Theorem 18.0.1.

Problem 18.3. *Prove Theorem 18.0.1 using splay trees with Δ representations of $w(\cdot)$ and* minw$(\cdot)$.

Problem 18.4. *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 18.5. *Augment the data structure of Problem 18.4 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 18.4):*

- 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.*

18.3 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 a data structure called *link-cut* trees. A rooted tree is represented as a collection of paths joined together.

To choose a representation, we will designate each edge of the tree as either dashed or solid. The maximal paths consisting of solid edges are called *solid* paths. The algorithm will maintain the property that each node has at most one solid child edge. An example is illustrated in Figure 18.9.

The algorithm makes use of the operation of *exposing* a node u:

> Ensure that the solid path containing u consists of all u's ancestors by converting dashed edges along the path to solid and solid edges incident to the path to dashed.

The operation is illustrated in Figure 18.10.

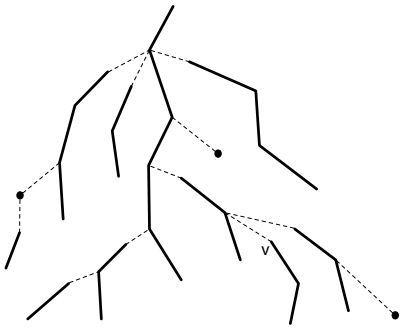

Figure 18.9: A rooted tree divided into solid paths.

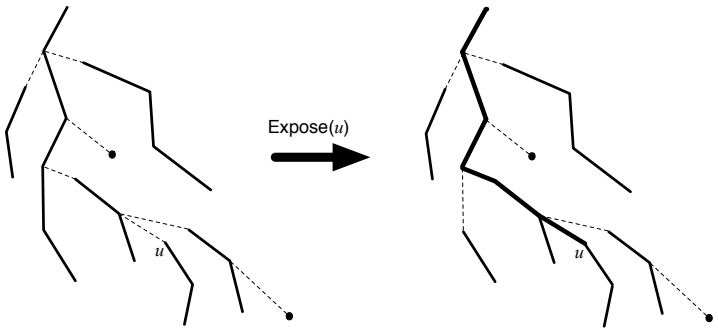

Figure 18.10: An example of exposing a node u. The effect is that the solid path containing u contains all of u's ancestors. In this figure, the heavy solid path comprises u's ancestors.

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

Lemma 18.3.1. *Every node v has at most one heavy child edge and at most $\lfloor \log n \rfloor$ light ancestor edges.*

18.3.1 High-level analysis of the *expose* operation

Splicing a dashed edge uv means converting uv to a solid edge and converting the other solid child edge of v (if there is one) to a dashed edge. See Figure 18.11 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 edge, converting that edge to a dashed edge. Define the *actual cost* of an *expose* operation to be the number of splices.

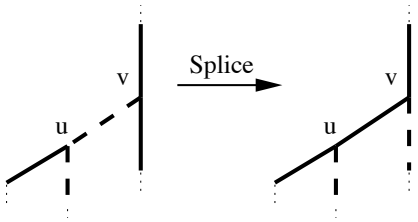

Figure 18.11: Splicing the edge uv.

Now we analyze the number of splices due to exposing u. Each splice converts a dashed edge to solid.

number of splices

$$\begin{aligned} &= |\{\text{edges converted from dashed to solid}\}| \\ &= |\{\text{light dashed edges converted to solid}\}| + |\{\text{heavy dashed edges converted to solid}\}| \end{aligned} \tag{18.3}$$

By Lemma 18.3.1, at most $\log n$ splices convert a light dashed edge to solid.

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

Lemma 18.3.2. *When a node u is exposed, the number of splices plus the increase in Φ_A is at most $1 + 2\log n$.*

Proof.

$$\begin{aligned}
\text{virtual cost} &= \text{actual cost} + \text{increase in } \Phi_A \\
&= \text{number of splices} + |\{\text{heavy solid edges converted to dashed}\}| \\
&\quad -|\{\text{heavy dashed edges converted to solid}\}| \\
&\leq \text{number of splices} + |\{\text{light dashed edges converted to solid}\}| \\
&\quad -|\{\text{heavy dashed edges converted to solid}\}| \\
&= 2 \cdot |\{\text{light dashed edges converted to solid}\}| \text{ by (18.3)} \\
&\leq 2 \log n \text{ by Lemma 18.3.1}
\end{aligned}$$

□

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

18.3.2 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 18.1.4.

18.3.3 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 18.12. 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[·] of pointers, and a Delta representation Δ*flipped*(·) of left-right.

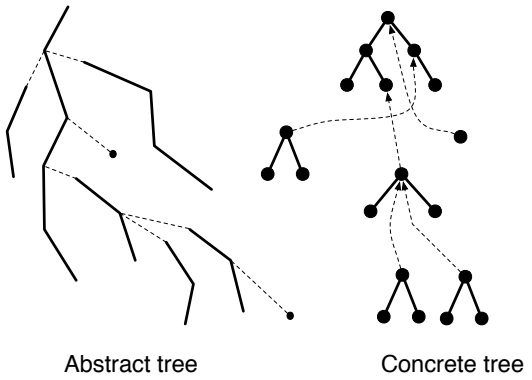

Figure 18.12: On the left is a diagram of an abstract tree. On the right is the corresponding concrete tree.

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(v) 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 concrete 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 edge xy is represented by an edge to y from the root of the solid tree containing x.

Question 18.3.4. *Write a procedure* IsSolidRoot(v) *that returns* true *if v is the root of its solid tree.*

18.3.4 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 an 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 18.1.5.

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 of its solid tree, 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
    z := parent(u)
    splay z to root of its solid tree
    v := left(z) # v might be null
    if v ≠ null, SOLIDTODASHED(v) # v is about to become an unacknowledged child of z
    DASHEDTOSOLID(u, z) # u is about to become a solid child of z
    left(z) := u
```

18.3.5 Analysis of EXPOSE(u) for trees not supporting descendant search

We combine the technique from splay-tree analysis with the analysis in Section 18.3.1 of the number of splices required when a node u is exposed. Recall that $\Phi_A = n - 1 - |\{\text{heavy solid edges}\}|$ is the potential function defined in Section 18.3.1. 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 18.3.5. *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 18.2.5, the virtual cost with respect to Φ_C of all the splayings is at most

$$\sum_{i=0}^{k} 1 + 3(r'(v_i) - r(v_i)) \tag{18.4}$$

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 (18.4) 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 18.3.2, $k +$ 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 18.6. *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 $\Phi_C \leq dn$.*

18.4 Link-cut trees that support descendant search

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 edges in the abstract tree correspond to dotted vertices in the concrete tree. Consider a dashed edge 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 v (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

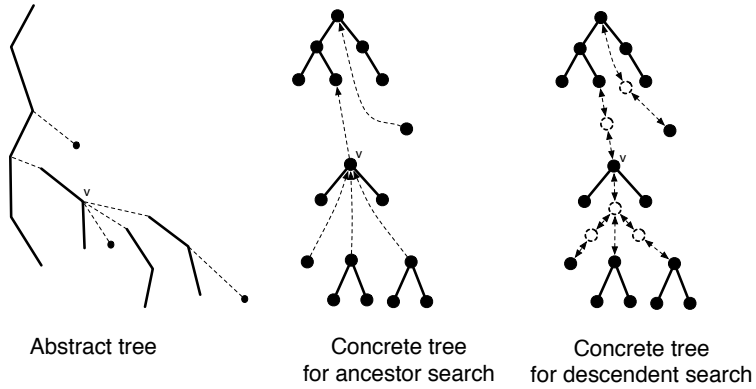

Figure 18.13: This figure shows two concrete trees corresponding to the same abstract tree. The middle diagram shows a concrete tree with the structure described in Section 18.3.3. The right diagram shows a concrete tree that supports descendant search. Each solid node has a *middle* pointer that either 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.

be traversed top-down. Example illustrating a conceptual tree and its concrete tree representatiopns are given in Figures 18.13 and 18.4.

The pseudocode for EXPOSE_STEP is now as follows:

```
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
    x := parent(u) # x is a dotted node
    splay x to root of its dotted tree
    z := parent(x) # v is a solid node
    splay z to root of its solid tree
    v := left(z) # v might be null
    if v ≠ null, SOLIDTODASHED(v)
    DASHEDTOSOLID(u, z)
    parent(u), left(z), parent(v), middle(x) = z, u, x, v
```

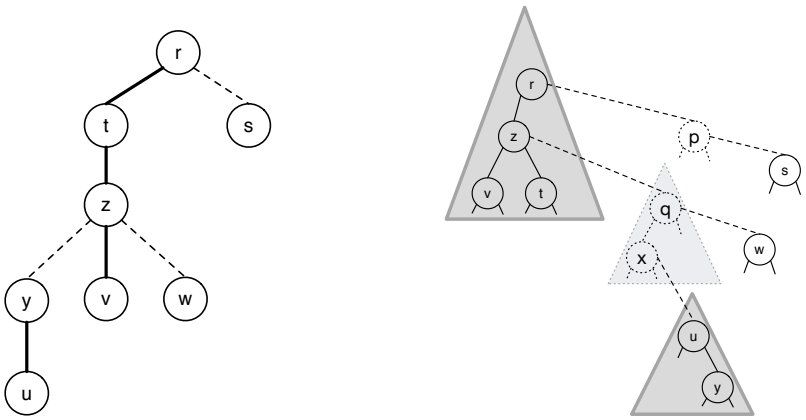

Figure 18.14: A conceptual tree is shown on the left. Solid and dashed edges are indicated. The corresponding concrete tree is shown on the right. Each solid tree is indicated by a darker gray triangle. A dotted tree that is not a singleton is indicated by a lighter grey triangle.

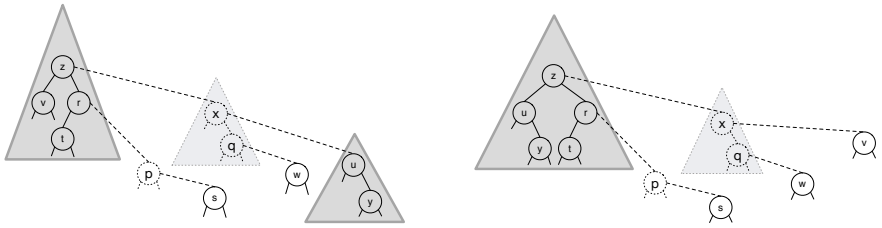

Figure 18.15: Illustration of ExposeStep(u) for the tree in Figure 18.4. The concrete tree after splaying x and z to the root of their trees is shown on the left. The concrete tree after the splice exchanging v and u is shown on the right.

Problem 18.7. *Analyze the variant of* Expose_Step(u) *that supports descendant search.*

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

Problem 18.8. *Extend the result in Problem 18.6 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.*

18.5 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 edge 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 edge 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
```

18.5.1 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 edge. 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 $O(\log n)$.

18.5.2 Evert

Everting node u means making u the root of the conceptual tree. This is achieved by making the path from the current root of the conceptual tree to u solid, and then reversing this path. It is easy to implement EVERT(u) using EXPOSE and the Delta representation of the binary value *flipped* keeping track of the left-right order of children as described in Section 18.1.4.

```
def EVERT(u):
    EXPOSE(u)
    Δflipped(u) = (Δflipped(u) + 1)  mod 2
```

Problem 18.9. *Show that the virtual cost of* EVERT *is* $O(\log n)$.

18.6 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.

We express the weight $w(v)$ of each vertex as the sum of two quantities, the contribution $w_d(v)$ of calls to ADDTODESCENDANTS that affect the weight of v, and the contribution $w_a(v)$ of calls to ADDTOANCESTORS that affect the weight of v. 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 quantities $w_d(\cdot)$ and $w_a(\cdot)$; Instead, we use the Delta representation of weights described in Section 18.1.1. Each node v stores *weight increments* $\Delta w_d[v]$ and $\Delta w_a[v]$, such that the weights $w_d(v), w_a(v)$ of a node v equal the sum of the corresponding weight increments of some of its ancestors. The exact invariants satisfied by Δw_d and Δw_a are slightly different, because the operations they support are different. For convenience of notation we define $\Delta w(v) = \Delta w_d[v] + \Delta w_a[v]$.

18.6.1 Supporting ADDTODESCENDANTS

To support ADDTODESCENDANTS, we maintain the following invariant:

> For each node u, the quantity $w_d(u)$ is the sum $\sum_v \Delta w_d[v]$ 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_d[u] += α
    if u has a right child, Δw_d[right(u)] −= α
```

To preserve the invariant, the rotation procedure RotateUp should be modified as in Section 18.1.6. The "hook" subroutines SolidToDashed and DashedToSolid of Section 18.3.4 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_d[v] -= Δw_d[u]
```

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

18.6.2 Supporting AddToAncestors

To support AddToAncestors, we use a similar invariant:

> For each node u, the quantity $w_a(u)$ is the sum $\sum_v \Delta w_a[v]$ 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_a[u] += α
    if u has a left child, Δw_a[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 $\Delta w_a(u)$ increases by α the weight of all descendants of u in its solid tree, and subtracting α from $\Delta w_a[\text{left}(u)]$ 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 $\Delta w_a[u]$ 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 18.6.1. 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 a solid tree, about to become dashed child
    Δw_a[u] += Δw_a[parent(u)]
```

and

```
def DashedToSolid(u, v):
    pre: u is dashed child of v, about to become solid child
         v is root of a solid tree
    Δw_a[u] -= Δw_a[v]
```

18.6.3 Getting the weight of a node

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

```
def GetWeight(v):
    Expose(v)
    return Δw_d[v] + Δw_a[v]
```

The correctness of this procedure uses the fact that, once v is the root of its concrete tree, $\Delta w_a[v]$ is $w_a(v)$ and $\Delta w_d[v]$ is $w_d(v)$.

18.7 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.

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:

- 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 18.1.2. Here we define $minw(u)$ to be the minimum weight $w(v) = w_d(v) + w_a(v)$ among *solid* descendants v of u.

A Delta representation of $minw(\cdot)$ is used, as described in Section 18.1.3. 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 consider the procedure RotateUp. It changes the children of some vertices, and hence the solid descendants of some vertices. In order to maintain $\Delta minw[\cdot]$, we need to add a call to a hook, ChildChange($\cdot$), whenever the children of a node change. Let v be a node, and let its solid children be $v_1, \ldots, v_k$. ChildChange(v) updates $\Delta minw[v]$ based on the values of $v_1, \ldots, v_k$ as given in Equation 18.2. 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. We note that u and its former parent are the only vertices whose descendants change by the rotation.

We next consider the procedure Expose. It mainly involves calls to RotateUp, either directly or by splaying. In addition, it makes calls to ExposeStep(u). Consider such a call. Let v be the parent of u. After v is splayed to the root of its solid tree, the solid left edge of v is exchanged with the dashed parent edge of u. As a result, $minw(v)$ may change. To maintain the invariant, the hook DashedToSolid(u, v) needs to also call ChildChange(v). Note that since v is the root of its solid tree, it is the only vertex whose solid descendants change when ExposeStep(u) is called.

The changes required to Link and cut are similar; ChildChange should be called after u loses its right child in Cut, and after u gains v as a right child in Link.

Finally, consider the bulk update operations AddToDescendants(u, α) or AddToAncestors(u, α). Both procedures make a single call to Expose(u),

making u the root of the concrete tree. Then, α is added to the weight of u and to the weights of either all the descendants of u's left child (in case of AddToDescendants) or to the solid descendants of u's right child (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 strict descendant v of u. We update $\Delta minw[u]$ based on the $\Delta minw$ values of its children as given in Equation 18.2.

18.7.1 Supporting ancestor searches

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

$$\alpha \geq \Delta minw[u] \tag{18.5}$$

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

By the Delta min invariant, the precondition 18.5 is equivalent to the condition

$$\alpha + w(u) \geq minw(u) \tag{18.6}$$

```
def SolidFind(u, α, dir):
    pre: α ≥ Δminw[u]
    post: Returns the dirmost solid descendant v such that w(v) ≤ 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(u1, α − Δw[u1], dir)
4   if α ≥ 0
5     splay u to the root of its solid tree
6     return u
7   return SolidFind(u2, α − Δw[u2], dir)
```

First we prove its correctness. By 18.6 (equivalent to the precondition) and the definition of $minw(u)$, 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 $u_1 \neq \mathit{null}$ and $\mathit{minw}(u_1) \leq \alpha + w(u)$. By the Delta min invariant, the latter inequality holds if $\Delta\mathit{minw}(u_1) + w(u_1) \leq \alpha + w(u)$, which holds if $\Delta\mathit{minw}[u_1] \leq \alpha + w(u) - w(u_1) = \alpha - \Delta w[u_1]$. 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 $\mathit{minw}(u_2) \leq \alpha + w(u)$, so $\Delta\mathit{minw}[u_2] + w(u_2) \leq \alpha + w(u)$, so $\Delta\mathit{minw}[u_2] \leq \alpha - \Delta w[u_2]$. 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 the 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 $O(\log n)$.

AncestorFindWeight and AncestorFindMin

Now we write AncestorFindWeight$(u, \alpha, \mathit{dir})$. 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 solid 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 ($\mathrm{right}(u) \neq \mathit{null}$) and, if so, whether $\mathit{minw}(\mathrm{right}(u)) \leq \alpha$. By the Delta min invariant, $\mathit{minw}(\mathrm{right}(u))$ equals $\Delta\mathit{minw}[\mathrm{right}(u)] + w(\mathrm{right}(u))$. By the invariant for $\Delta w[\cdot]$, $w(\mathrm{right}(u)) = \Delta w[\mathrm{right}(u)] + \Delta w[u]$, so the procedure checks whether $\Delta\mathit{minw}[\mathrm{right}(u)] + \Delta w[\mathrm{right}(u)] + \Delta w[u] \leq \alpha$.

If u is a candidate and either we seek the leafmost 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 no candidate, in which case the procedure returns *null*).

```
def AncestorFindWeight(u, α, dir):
    Expose(u)
```

u_is_candidate := $\Delta w[u] \leq \alpha$
candidate_on_right := (right(u) $\neq$ *null*) and
($\Delta minw$[right(u)] + Δw[right(u)] + $\Delta w[u] \leq \alpha$)
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), $\alpha - \Delta w$[right(u)] $- \Delta w[u]$, *dir*)

Problem 18.10. *Write the procedure* AncestorFindMin *in terms of* SolidFind. *Argue that your procedure is correct.*

18.7.2 Supporting descendant searches

To support descendant searches, we use dotted vertices as described in Section 18.4. 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 ∞.

We represent $\widehat{minw}$ using Delta representation, using a similar invariant to the one of Δw_d. Each solid node u has a decoration $\Delta\widehat{minw}[u]$ such that

> The weight of $\widehat{minw}(u)$ of a node u is the sum $\sum_v \Delta\widehat{minw}[v]$ over all ancestors v of u in the concrete tree.

The decorations $\Delta\widehat{minw}[u]$ need to be updated to maintain these invariants. To support this we modify Expose(u) to first collect the values of $w(v)$ and $\widehat{minw}(v)$ for all nodes v along the path from the root of the concrete tree to u. Since Expose(u) makes u the root of the concrete tree using rotations, traversing the path from the root to u does not change the asymptotic running time of Expose.

Calls to RotateUp only occur through calls to Expose. The only nodes whose descendants change when Rotatup(u) is called are the former parent of u and u itself. Hence, the only nodes whose $\widehat{minw}$ value might change are the former parent of u and u, and the only nodes whose $\Delta\widehat{minw}$ value might change are these two nodes and their children. Note that only a constant number of nodes are affected, and that all affected nodes are adjacent to the rotating nodes. We first compute $\widehat{minw}(v)$ for each such affected node v as the minimum over the $\widehat{minw}$ values of its children. We can then update the values of $\Delta\widehat{minw}$ for each affected node v by calculating the difference $\widehat{minw}(\text{parent}(v)) - \widehat{minw}(v)$, starting from the rootmost affected node.

Calls to Expose also change some solid and dashed edges via ExposeStep(u). Let x, z and v be as defined in the pseudocode of ExposeStep(u). The nodes u

and v swap roles during ExposeStep(u); The node u becomes the left acknowledged child of z, and v becomes an unacknowledged child of z by becoming the middle child of the dotted node x. See Figure 18.16. The only nodes whose $\widehat{minw}$ values changes in this process are z and x. The values $\widehat{minw}(z), \widehat{minw}(x)$ can be calculated from the $\widehat{minw}$ values of their children, which were collected at the beginning of the Expose operation. The $\Delta\widehat{minw}$ of x, z and their children can then be updated accordingly to maintain the invariant.

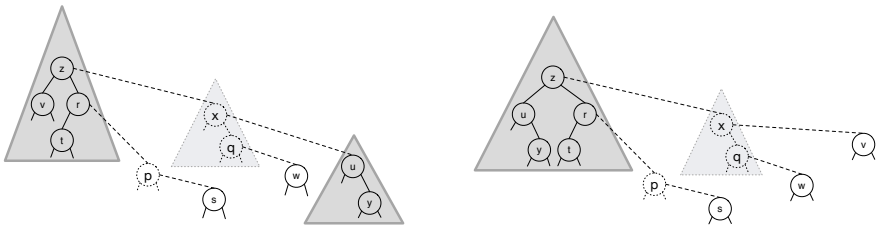

Figure 18.16: Illustration of ExposeStep(u) for the tree in Figure 18.4. The concrete tree after splaying x and z to the root of their trees is shown on the left. The concrete tree after the splice exchanging v and u is shown on the right.

We next consider maintaining the invariant on $\Delta\widehat{minw}$ during calls to Cut(u) and Link(u, v). After the calling Expose, the only changes concern the root of the concrete tree and its children, so the invariant can be updated explicitly by calculating the $\widehat{minw}$ values of the affected nodes and then their new $\Delta\widehat{minw}$ values.

Calling AddToDescendants(u, α) first makes u the root of the concrete tree by calling Expose(u), and then adds α to u, and to all the descendant of u's left child v. This increases $\widehat{minw}(w)$ by α for every descendant w of v, and possibly also changes $\widehat{minw}(u)$. We first increase $\Delta\widehat{minw}[v]$ by α. This maintains the invariant on $\Delta\widehat{minw}$ for v and its descendants. We can then then calculate $\widehat{minw}(u)$ from the $\widehat{minw}$ value of its children, and restore the invariant for $\Delta\widehat{minw}[u]$ as well.

Calling AddToDescendants(u, α) first makes u the root of the concrete tree by calling Expose(u), and then adds α to u, and to all the concrete descendant of u's right child v. This does not change $\widehat{minw}$ for any node, so the invariant on $\Delta\widehat{minw}$ is trivially maintained.

Problem 18.11. *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 $\Delta\min[\cdot]$.*

Problem 18.12. *Write* DescendantFindMin *in terms of* Find. *You can assume that* $\widehat{\text{minw}}(\cdot)$ *is represented explicitly.*

18.7.3 Representing trees with dart weights

For some applications such as MSSP (Chapter 7) and single-source single-sink maximum flow (Chapter 10), one needs a variant of Theorem 18.0.2 that supports dart weights. That is, we maintain a directed tree of arcs, and each of the two darts of each arc of the tree has its own weight. Operations and queries on ancestors or descendants in the tree additionally specify whether they relate to the rootward darts or to the leafward darts of the ancestor/descendant edges. We had already mentioned at the beginning of this chapter that edge weights can be represented by adding an artificial vertex v_e in the middle of each edge e, and storing the weight of edge e in the weight of v_e (the weight of the non-artificial nodes is set to a sufficiently large value). Maintaining dart weights without *Evert*, can be easily supported by storing two weights for each arc, one for each dart (each with its own Delta representations for ancestor and descendant updates). Supporting dart weights with Evert(u) is a bit less trivial, because eversion swaps the leafward and rootward darts along the path from the new root u to the old root. To support dart weights with evert, each arc e has two weights $w_0(e)$ and $w_1(e)$ (each of these weights is represented by its own Delta representations for ancestor and descendant updates). Recall that to implement Evert we have used the *flipped* decoration to keep track of the left-right ordering within each solid tree, and that when u is everted, toggling *flipped*(u) has the effect of reversing the path from the former root of the conceptual tree to u. We use *flipped* to also determine which of the two weights is associated with the leafward dart and which with the rootward dart. That is, the weight of the leafward dart of e is represented by $w_{\textit{flipped}(v_e)}(v_e)$, and the weight of the rootward dart of e is represented by $w_{\textit{flipped}(v_e)+1 \pmod 2}(v_e)$. This way, toggling *flipped*(u) both reverses the path from u to the former root and swaps the rootward and leafward darts along that path.

18.8 Chapter Notes