12

Splay Trees

Sanjeev Saxena
Indian Institute of Technology, Kanpur

12.1 Introduction

In this chapter we discuss following topics:

1. Introduction to splay trees and their applications
2. Splay Trees–description, analysis, algorithms and optimality of splay trees.
3. Linking and Cutting Trees
4. Case Study: Application to Network Flows
5. Variants of Splay Trees.

There are various data structures like AVL-trees, red-black trees, 2-3-trees (Chapter 10) which support operations like insert, delete (including deleting the minimum item), search (or membership) in $O(\log n)$ time (for each operation). Splay trees, introduced by Sleator and Tarjan [13, 15] support all these operations in $O(\log n)$ amortized time, which roughly means that starting from an empty tree, a sequence of m of these operations will take $O(m \log n)$ time (deterministic), an individual operation may take either more time or less time (see Theorem 12.1). We discuss some applications in the rest of this section.

Assume that we are searching for an item in a "large" sorted file, and if the item is in the kth position, then we can search the item in $O(\log k)$ time by exponential and binary search. Similarly, finger search trees (Chapter 11) can be used to search any item at distance f from a finger in $O(\log f)$ time. Splay trees can search (again in amortized sense) an item

from any finger (which need not even be specified) in $O(\log f)$ time, where f is the distance from the finger (see Section 12.4.2). Since the finger is not required to be specified, the time taken will be minimum over all possible fingers (time, again in amortized sense).

If we know the frequency or probability of access of each item, then we can construct an optimum binary search tree (Chapter 14) for these items; total time for all access will be the smallest for optimal binary search trees. If we do not know the probability (or access frequency), and if we use splay trees, even then the total time taken for all accesses will still be the same as that for a binary search tree, up to a multiplicative constant (see Section 12.4.1).

In addition, splay trees can be used almost as a "black box" in linking and cutting trees (see Section 12.5). Here we need the ability to add (or subtract) a number to key values of all ancestors of a node x.

Moreover, in practice, the re-balancing operations (rotations) are very much simpler than those in height balanced trees. Hence, in practice, we can also use splay trees as an alternative to height balanced trees (like AVL-trees, red-black trees, 2-3-trees), if we are interested only in the total time. However, some experimental studies [3] suggest, that for random data, splay trees outperform balanced binary trees only for highly skewed data; and for applications like "vocabulary accumulation" of English text [16], even standard binary search trees, which do not have good worst case performance, outperform both balanced binary trees (AVL trees) and splay trees. In any case, the constant factor and the algorithms are not simpler than those for the usual heap, hence it will not be practical to use splay trees for sorting (say as in heap sort), even though the resulting algorithm will take $O(n \log n)$ time for sorting, unless the data has some degree of pre-sortedness, in which case splay sort is a practical alternative [10]. Splay trees however, can not be used in real time applications.

Splay trees can also be used for data compression. As splay trees are binary search trees, they can be used directly [4] with guaranteed worst case performance. They are also used in data compression with some modifications [9]. Routines for data compression can be shown to run in time proportional to the entropy of input sequence [7] for usual splay trees and their variants.

12.2 Splay Trees

Let us assume that for each node x, we store a real number key(x).

In any binary search tree left subtree of any node x contains items having "key" values less than the value of key(x) and right subtree of the node x contains items with "key" values larger than the value of key(x).

In splay trees, we first search the query item, say x as in the usual binary search trees— compare the query item with the value in the root, if smaller then recursively search in the left subtree else if larger then, recursively search in the right subtree, and if it is equal then we are done. Then, informally speaking, we look at every disjoint pair of consecutive ancestors of x, say y =parent(x) and z =parent(y), and perform certain pair of rotations. As a result of these rotations, x comes in place of z.

In case x has an odd number of proper ancestors, then the ancestor of x (which is child of the root), will also have to be dealt separately, in terminal case— we rotate the edge between x and the root. This step is called *zig* step (see Figure 12.1).

If x and y are both left or are both right children of their respective parents, then we first rotate the edge between y and its parent z and then the edge between x and its parent y. This step is called *zig-zig* step (see Figure 12.2).

If x is a left (respectively right) child and y is a right (respectively left) child, then we

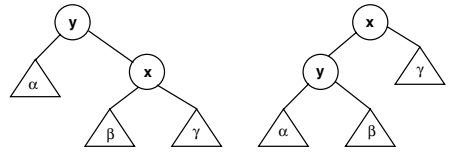

FIGURE 12.1: parent(x) is the root— edge xy is rotated (Zig case).

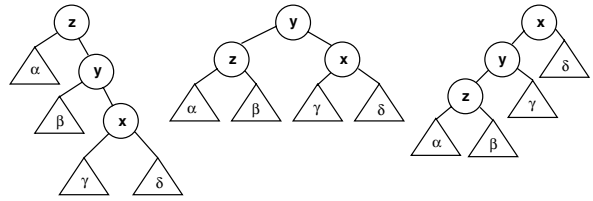

FIGURE 12.2: x and parent(x) are both right children (Zig-Zig case) —first edge yz is rotated then edge xy.

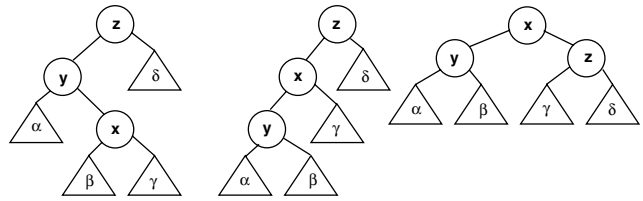

FIGURE 12.3: x is a right child while parent(x) is a left child (Zig-Zag case)— first edge xy is rotated then edge xz.

first rotate the edge between x and y and then between x and z, this step is called *zig-zag* step (see Figure 12.3).

These rotations (together) not only make x the new root, but also, roughly speaking halve the depth (length of path to root) of all ancestors of x in the tree. If the node x is at depth "d", splay(x) will take $O(d)$ time, i.e., time proportional to access the item in node x.

Formally, **splay**(x) is a sequence of rotations which are performed (as follows) until x becomes a root:

- If parent(x) is root, then we carry out usual rotation, see Figure 12.1.
- If x and parent(x) are both left (or are both right) children of their parents, then we first rotate at y =parent(x) (i.e., the edge between y and its parent) and then rotate at x, see Figure 12.2.
- If x is left (or respectively right) child but parent(x) is right (respectively left) child of its parent, then first rotate at x and then again rotate at x, see Figure 12.3.

12.3 Analysis

We will next like to look at the "amortized" time taken by splay operations. Amortized time is the average time taken over a worst case sequence of operations.

For the purpose of analysis, we give a positive weight $w(x)$ to (any) item x in the tree. The weight function can be chosen completely arbitrarily (as long it is strictly positive). For analysis of splay trees we need some definitions (or nomenclature) and have to fix some parameters.

Weight of item x: For each item x, an arbitrary positive weight $w(x)$ is associated (see Section 12.4 for some examples of function $w(x)$).

Size of node x: Size(x) is the sum of the individual weights of all items in the subtree rooted at the node x.

Rank of node x: Rank of a node x is $\log_2(\text{size}(x))$.

Potential of a tree: Let α be some positive constant (we will discuss choice of α later), then the potential of a tree T is taken to be
$\alpha(\text{Sum of rank}(x) \text{ for all nodes } x \in T) = \alpha \sum_{x \in T} \text{rank}(x)$.

Amortized Time: As always,
Amortized time = Actual Time + New Potential − Old Potential.

Running Time of Splaying: Let β be some positive constant, choice of β is also discussed later but $\beta \leq \alpha$, then the running time for splaying is
$\beta \times$Number of rotations.
If there are no rotations, then we charge one unit for splaying.

We also need a simple result from algebra. Observe that $4xy = (x+y)^2 - (x-y)^2$. Now if $x + y \leq 1$, then $4xy \leq 1 - (x-y)^2 \leq 1$ or taking logarithms[1], $\log x + \log y \leq -2$. Note that the maximum value occurs when $x = y = \frac{1}{2}$.

FACT 12.1 *[Result from Algebra] If $x + y \leq 1$ then $\log x + \log y \leq -2$. The maximum value occurs when $x = y = \frac{1}{2}$.*

LEMMA 12.1 [Access Lemma] The amortized time to splay a tree (with root "t") at a node "x" is at most

$$3\alpha(\text{rank}(t) - \text{rank}(x)) + \beta = O\left(\log\left(\frac{\text{Size}(t)}{\text{Size}(x)}\right)\right)$$

Proof We will calculate the change in potential, and hence the amortized time taken in each of the three cases.

Let $s(\ )$ denote the sizes before rotation(s) and $s'(\ )$ be the sizes after rotation(s). Let $r(\ )$ denote the ranks before rotation(s) and $r'(\ )$ be the ranks after rotation(s).

Case 1– x and parent(x) are both left (or both right) children

[1] All logarithms in this chapter are to base two.

Please refer to Figure 12.2. Here, $s(x) + s'(z) \leq s'(x)$, or $\frac{s(x)}{s'(x)} + \frac{s'(z)}{s'(x)} \leq 1$. Thus, by Fact 12.1,

$$-2 \geq \log \frac{s(x)}{s'(x)} + \log \frac{s'(z)}{s'(x)} = r(x) + r'(z) - 2r'(x),$$

or

$$r'(z) \leq 2r'(x) - r(x) - 2.$$

Observe that two rotations are performed and only the ranks of x, y and z are changed. Further, as $r'(x) = r(z)$, the Amortized Time is
$= 2\beta + \alpha((r'(x) + r'(y) + r'(z)) - (r(x) + r(y) + r(z)))$
$= 2\beta + \alpha((r'(y) + r'(z)) - (r(x) + r(y)))$
$\leq 2\beta + \alpha((r'(y) + r'(z)) - 2r(x))$, (as $r(y) \geq r(x)$).
As $r'(x) \geq r'(y)$, amortized time
$\leq 2\beta + \alpha((r'(x) + r'(z)) - 2r(x))$
$\leq 2\beta + \alpha((r'(x) + \{2r'(x) - r(x) - 2\} - 2r(x)))$
$\leq 3\alpha(r'(x) - r(x)) - 2\alpha + 2\beta$
$\leq 3\alpha(r'(x) - r(x))$ (as $\alpha \geq \beta$).

Case 2– x is a left child, parent(x) is a right child

Please refer to Figure 12.3. $s'(y) + s'(z) \leq s'(x)$, or $\frac{s'(y)}{s'(x)} + \frac{s'(z)}{s'(x)} \leq 1$. Thus, by Fact 12.1,
$-2 \geq \log \frac{s'(y)}{s'(x)} + \log \frac{s'(z)}{s'(x)} = r'(y) + r'(z) - 2r'(x)$, or,
$r'(y) + r'(z) \leq 2r'(x) - 2$.
Now Amortized Time$= 2\beta + \alpha((r'(x) + r'(y) + r'(z)) - (r(x) + r(y) + r(z)))$. But, as $r'(x) = r(z)$, Amortized time $= 2\beta + \alpha((r'(y) + r'(z)) - (r(x) + r(y)))$. Using $r(y) \geq r(x)$, Amortized time
$\leq 2\beta + \alpha((r'(y) + r'(z)) - 2r(x))$
$\leq 2\alpha(r'(x) - r(x)) - 2\alpha + 2\beta$
$\leq 3\alpha(r'(x) - r(x)) - 2(\alpha - \beta) \leq 3\alpha(r'(x) - r(x))$

Case 3– parent(x) is a root

Please refer to Figure 12.1. There is only one rotation, Amortized Time
$= \beta + \alpha((r'(x) + r'(y)) - (r(x) + r(y)))$.
But as, $r'(x) = r(y)$, Amortized time is
$\beta + \alpha(r'(y) - r(x))$
$\leq \beta + \alpha(r'(x) - r(x))$
$\leq \beta + 3\alpha(r'(x) - r(x))$.
As case 3, occurs only once, and other terms vanish by telescopic cancellation, the lemma follows.

THEOREM 12.1 *Time for m accesses on a tree having at most n nodes is $O\big((m + n)\log n\big)$*

Proof Let the weight of each node x be fixed as $1/n$. As there are n nodes in the entire tree, the total weight of all nodes in the tree is 1.

If t is the root of the tree then, size$(t) = 1$ and as each node x has at least one node (x itself) present in the subtree rooted at x (when x is a leaf, exactly one node will be present), for any node x, size$(x) \geq (1/n)$. Thus, we have following bounds for the ranks— $r(t) \leq 0$ and $r(x) \geq -\log n$.

Or, from Lemma 12.1, amortized time per splay is at most $1 + 3 \log n$. As maximum possible value of the potential is $n \log n$, maximum possible potential drop is also $O(n \log n)$, the theorem follows.

We will generalize the result of Theorem 12.1 in Section 12.4, where we will be choosing some other weight functions, to discuss other optimality properties of Splay trees.

12.3.1 Access and Update Operations

We are interested in performing following operations:

1. Access(x)— x is a key value which is to be searched.
2. Insert(x)— a node with key value x is to be inserted, if a node with this key value is not already present.
3. Delete(x)— node containing key value x is to be deleted.
4. Join(t_1, t_2)— t_1 and t_2 are two trees. We assume that all items in tree t_1 have smaller key values than the key value of any item in the tree t_2. The two trees are to be combined or joined into a single tree as a result, the original trees t_1 and t_2 get "destroyed".
5. Split(x, t)— the tree t is split into two trees (say) t_1 and t_2 (the original tree is "lost"). The tree t_1 should contain all nodes having key values less than (or equal to) x and tree t_2 should contain all nodes having key values strictly larger than x.

We next discuss implementation of these operations, using a single primitive operation— splay. We will show that each of these operations, for splay trees can be implemented using $O(1)$ time and with one or two "splay" operations.

Access(x, t) Search the tree t for key value x, using the routines for searching in a "binary search tree" and splay at the last node— the node containing value x, in case the search is successful, or the parent of "failure" node in case the search is unsuccessful.

Join(t_1, t_2) Here we assume that all items in splay tree t_1 have key values which are smaller than key values of items in splay tree t_2, and we are required to combine these two splay trees into a single splay tree.
Access largest item in t_1, formally, by searching for "$+\infty$", i.e., a call to Access($+\infty, t_1$). As a result the node containing the largest item (say r) will become the root of the tree t_1. Clearly, now the root r of the splay tree t_1 will not have any right child. Make the root of the splay tree t_2 the right child of r, the root of t_1, as a result, t_2 will become the right sub-tree of the root r and r will be the root of the resulting tree.

Split(x, t) We are required to split the tree t into two trees, t_1 containing all items with key values less than (or equal to) x and t_2, containing items with key values greater than x.
If we carry out Access(x, t), and if a node with key value x is present, then the node containing the value x will become the root. We then remove the link from node containing the value x to its right child (say node containing value y); the resulting tree with root, containing the value x, will be t_1, and the tree with root, containing the value y, will be the required tree t_2.
And if the item with key value x is not present, then the search will end at a node

(say) containing key value z. Again, as a result of splay, the node with value z will become the root. If $z > x$, then t_1 will be the left subtree of the root and the tree t_2 will be obtained by removing the edge between the root and its left child.

Otherwise, $z < x$, and t_2 will be the right subtree of the root and t_1 will be the resulting tree obtained by removing the edge between the root and its right child.

Insert(x, t) We are required to insert a new node with key value x in a splay tree t. We can implement insert by searching for x, the key value of the item to be inserted in tree t using the usual routine for searching in a binary search tree. If the item containing the value x is already present, then we splay at the node containing x and return. Otherwise, assume that we reach a leaf (say) containing key y, $y \neq x$. Then if $x < y$, then add the new node containing value x as a left child of node containing value y, and if $x > y$, then the new node containing the value x is made the right child of the node containing the value y, in either case we splay at the new node (containing the value x) and return.

Delete(x, t) We are required to delete the node containing the key value x from the splay tree t. We first access the node containing the key value x in the tree t— Access(x, t). If there is a node in the tree containing the key value x, then that node becomes the root, otherwise, after the access the root will be containing a value different from x and we return(-1)— value not found. If the root contains value x, then let t_1 be the left subtree and t_2 be the right subtree of the root. Clearly, all items in t_1 will have key values less than x and all items in t_2 will have key values greater than x. We delete the links from roots of t_1 and t_2 to their parents (the root of t, the node containing the value x). Then, we join these two subtrees — Join(t_1, t_2) and return.

Observe that in both "Access" and "Insert", after searching, a splay is carried out. Clearly, the time for splay will dominate the time for searching. Moreover, except for splay, everything else in "Insert" can be easily done in $O(1)$ time. Hence the time taken for "Access" and "Insert" will be of the same order as the time for a splay. Again, in "Join", "Split" and "Delete", the time for "Access" will dominate, and everything else in these operations can again be done in $O(1)$ time, hence "Join", "Split" and "Delete" can also be implemented in same order of time as for an "Access" operation, which we just saw is, in turn, of same order as the time for a splay. Thus, each of above operations will take same order of time as for a splay. Hence, from Theorem 12.1, we have

THEOREM 12.2 *Time for m update or access operations on a tree having at most n nodes is $O\big((m+n)\log n\big)$.*

Observe that, at least in amortized sense, the time taken for first m operations on a tree which never has more than n nodes is the same as the time taken for balanced binary search trees like AVL trees, 2-3 trees, etc.

12.4 Optimality of Splay Trees

If $w(i)$ the weight of node i is independent of the number of descendants of node i, then the maximum value of size(i) will be $W = \sum w(i)$ and minimum value of size(i) will be $w(i)$.

As size of the root t, will be W, and hence rank $\log W$, so by Lemma 12.1, the amortized

time to splay at a node "x" will be $O\left(\log\left(\frac{\text{Size}(t)}{\text{Size}(x)}\right)\right) = O\left(\log\left(\frac{W}{Size(x)}\right)\right) = O\left(\log\frac{W}{w(x)}\right)$.

Also observe that the maximum possible change in the rank (for just node i) will be $\log W - \log w(i) = \log(W/w(i))$ or the total maximum change in all ranks (the potential of the tree, with $\alpha = 1$) will be bounded by $\sum \log(W/w(i))$.

Note that, as $\sum \frac{w(i)}{W} = 1$, $\sum \left|\log\frac{W}{w(i)}\right| \leq n \log n$ (the maximum occurs when all $\left(\frac{w(i)}{W}\right)$s are equal to $1/n$), hence maximum change in potential is always bounded by $O(n \log n)$.

As a special case, in Theorem 12.1, we had fixed $w(i) = 1/n$ and as a result, the amortized time per operation is bounded by $O(\log n)$, or time for m operations become $O((m + n)\log n)$. We next fix $w(i)$'s in some other cases.

12.4.1 Static Optimality

On any sequence of accesses, a splay tree is as efficient as the optimum binary search tree, up to a constant multiplicative factor. This can be very easily shown.

Let $q(i)$ be the number of times the ith node is accessed, we assume that each item is accessed at least once, or $q(i) \geq 1$. Let $m = \sum q(i)$ be the total number of times we access any item in the splay tree. Assign a weight of $q(i)/m$ to item i. We call $q(i)/m$ the *access frequency* of the ith item. Observe that the total (or maximum) weight is 1 and hence the rank of the root $r(t) = 0$.

Thus

$$r(t) - r(x) = 0 - r(x) = -\log\Big(\sum_{i \in T_x} \frac{q(i)}{m}\Big) \leq -\log\Big(\frac{q(x)}{m}\Big).$$

Hence, from Lemma 12.1, with $\alpha = \beta = 1$, the amortized time per splay (say at node "x") is at most
$3\alpha(r(t) - r(x)) + \beta$
$= 1 + 3(-\log(q(x)/m))$
$= 1 + 3\log(m/q(x))$.

As ith item is accessed $q(i)$ times, amortized total time for all accesses of the ith item is $O\big(q(i) + q(i)\log(\frac{m}{q(i)})\big)$, hence total amortized time will be $O\big(m + \sum q(i)\log(\frac{m}{q(i)})\big)$. Moreover as the maximum value of potential of the tree is $\sum \max\{r(x)\} \leq \sum \log(\frac{m}{q(i)}) = O\big(\sum \log(\frac{m}{q(i)})\big)$, the total time will be $O\big(m + \sum q(i)\log(\frac{m}{q(i)})\big)$.

THEOREM 12.3 *Time for m update or access operations on an n-node tree is $O\big(m + \sum q(i)\log(\frac{m}{q(i)})\big)$, where $q(i)$ is the total number of times item i is accessed, here $m = \sum q(i)$.*

REMARK 12.1 The total time, for this analysis is the same as that for the (static) optimal binary search tree.

12.4.2 Static Finger Theorem

We first need a result from mathematics. Observe that, in the interval $k - 1 \leq x \leq k$, $\frac{1}{x} \geq \frac{1}{k}$ or $\frac{1}{x^2} \geq \frac{1}{k^2}$. Hence, in this interval, we have, $\frac{1}{k^2} \leq \int_{k-1}^{k} \frac{dx}{x^2}$ summing from $k = 2$ to n, $\sum_2^n \frac{1}{k^2} \leq \int_1^n \frac{dx}{x^2} = 1 - \frac{1}{n}$ or $\sum_{k=1}^n \frac{1}{k^2} < 2$.

If f is an integer between 0 and n, then we assign a weight of $1/(|i - f| + 1)^2$ to item i. Then $W \leq 2\sum_{k=1}^{\infty} \frac{1}{k^2} < 4 = O(1)$. Consider a particular access pattern (i.e. a snapshot or history or a run). Let the sequence of accessed items be $i_1, \cdots, i_m$, some i_j's may occur

more than once. Then, by the discussion at the beginning of this section, amortized time for the jth access is $O(\log(|i_j - f| + 1)$. Or the total amortized time for all access will be $O(m + \sum_{j=1}^{m} \log(|i_j - f| + 1))$. As weight of any item is at least $1/n^2$, the maximum value of potential is $n \log n$. Thus, total time is at most $O(n \log n + m + \sum_{j=1}^{m} \log(|i_j - f| + 1))$.

REMARK 12.2 f can be chosen as any fixed item (finger). Thus, this out-performs finger-search trees, if any fixed point is used as a finger; but here the finger need not be specified.

12.4.3 Working Set Theorem

Splay trees also have the working set property, i.e., if only t different items are being repeatedly accessed, then the time for access is actually $O(\log t)$ instead of $O(\log n)$. In fact, if t_j different items were accessed since the last access of i_jth item, then the amortized time for access of i_jth item is only $O(\log(t_j + 1))$.

This time, we number the accesses from 1 to m in the order in which they occur. Assign weights of $1, 1/4, 1/9, \cdots, 1/n^2$ to items in the order of the first access. Item accessed earliest gets the largest weight and those never accessed get the smallest weight. Total weight $W = \sum(1/k^2) < 2 = O(1)$.

It is useful to think of item having weight $1/k^2$ as being in the kth position in a (some abstract) queue. After an item is accessed, we will be putting it in front of the queue, i.e., making its weight 1 and "pushing back" items which were originally ahead of it, i.e., the weights of items having old weight $1/s^2$ (i.e., items in sth place in the queue) will have a new weight of $1/(s+1)^2$ (i.e., they are now in place $s+1$ instead of place s). The position in the queue, will actually be the position in the "move to front" heuristic.

Less informally, we will be changing the weights of items after each access. If the weight of item i_j during access j is $1/k^2$, then after access j, assign a weight 1 to item i_j. And an item having weight $1/s^2$, $s < k$ gets weight changed to $1/(s+1)^2$.

Effectively, item i_j has been placed at the head of queue (weight becomes $1/1^2$); and weights have been permuted. The value of W, the sum of all weights remains unchanged.

If t_j items were accessed after last access of item i_j, then the weight of item i_j would have been $1/t_j^2$, or the amortized time for jth access is $O(\log(t_j + 1))$.

After the access, as a result of splay, the i_jth item becomes the root, thus the new size of i_jth item is the sum of all weights W— this remains unchanged even after changing weights. As weights of all other items, either remain the same or decrease (from $1/s^2$ to $1/(s+1)^2$), size of all other items also decreases or remains unchanged due to permutation of weights. In other words, as a result of weight reassignment, size of non-root nodes can decrease and size of the root remains unchanged. Thus, weight reassignment can only decrease the potential, or amortized time for weight reassignment is either zero or negative.

Hence, by discussions at the beginning of this section, total time for m accesses on a tree of size at most n is $O(n \log n + \sum \log(t_j + 1))$ where t_j is the number of different items which were accessed since the last access of i_jth item (or from start, if this is the first access).

12.4.4 Other Properties and Conjectures

Splay trees are conjectured [13] to obey "Dynamic Optimality Conjecture" which roughly states that cost for any access pattern for splay trees is of the same order as that of the best possible algorithm. Thus, in amortized sense, the splay trees are the best possible dynamic binary search trees up to a constant multiplicative factor. This conjecture is still open.

However, dynamic finger conjecture for splay trees which says that access which are close to previous access are fast has been proved by Cole[5]. Dynamic finger theorem states that the amortized cost of an access at a distance d from the preceding access is $O(\log(d+1))$; there is however $O(n)$ initialization cost. The accesses include searches, insertions and deletions (but the algorithm for deletions is different)[5].

Splay trees also obey several other optimality properties (see e.g. [8]).

12.5 Linking and Cutting Trees

Tarjan [15] and Sleator and Tarjan [13] have shown that splay trees can be used to implement linking and cutting trees.

We are given a collection of rooted trees. Each node will store a value, which can be any real number. These trees can "grow" by combining with another tree *link* and can shrink by losing an edge *cut*. Less informally, the trees are "dynamic" and grow or shrink by following operations (we assume that we are dealing with a forest of rooted trees).

link If x is root of a tree, and y is any node, not in the tree rooted at x, then make y the parent of x.

cut Cut or remove the edge between a non-root node x and its parent.

Let us assume that we want to perform operations like

- Add (or subtract) a value to all ancestors of a node.
- Find the minimum value stored at ancestors of a query node x.

More formally, following operations are to be supported:

find_cost(v)**:** return the value stored in the node v.

find_root(v)**:** return the root of the tree containing the node v.

find_min(v)**:** return the node having the minimum value, on the path from v till find_root(v), the root of the tree containing v. In case of ties, choose the node closest to the root.

add_cost(v, δ)**:** Add a real number δ to the value stored in every node on the path from v to the root (i.e., till find_root(v)).

find_size(v) find the number of nodes in the tree containing the node v.

link(v, w) Here v is a root of a tree. Make the tree rooted at v a child of node w. This operation does nothing if both vertices v and w are in the same tree, or v is not a root.

cut(v) Delete the edge from node v to its parent, thus making v a root. This operation does nothing if v is a root.

12.5.1 Data Structure

For the given forest, we make some of the given edges "dashed" and the rest of them are kept solid. Each non-leaf node will have only one "solid" edge to one of its children. All other children will be connected by a dashed edge. To be more concrete, in any given tree, the right-most link (to its child) is kept solid, and all other links to its other children are made "dashed".

As a result, the tree will be decomposed into a collection of solid paths. The roots of solid paths will be connected to some other solid path by a dashed edge. A new data structure

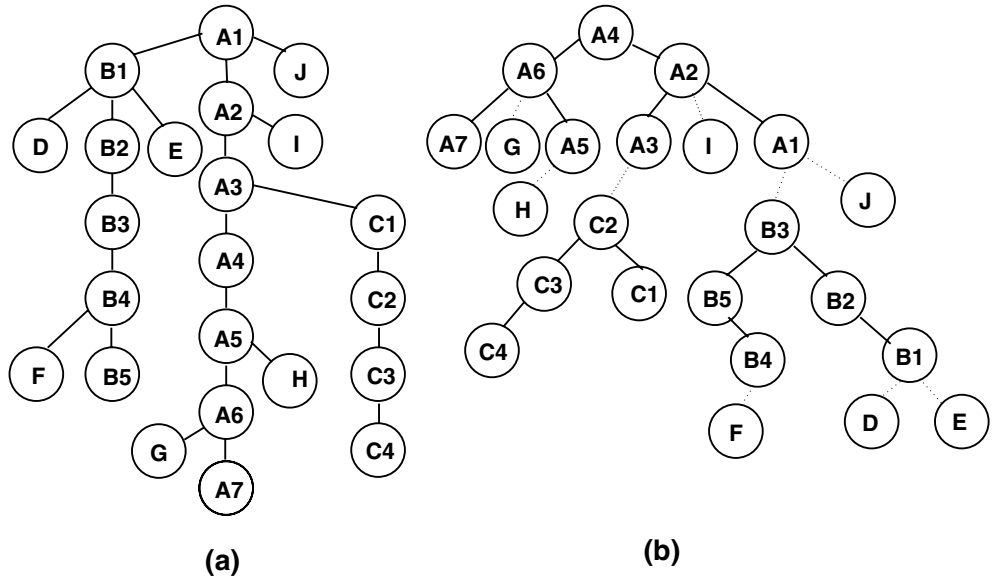

FIGURE 12.4: (a) Original Tree (b) Virtual Trees: Solid and dashed children.

called a "virtual tree" is constructed. Each linking and cutting tree T is represented by a virtual tree V, containing the same set of nodes. But each solid path of the original tree is modified or converted into a binary tree in the virtual tree; binary trees are as balanced as possible. Thus, a virtual tree has a (solid) left child, a (solid) right child and zero or more (dashed) middle children.

In other words, a virtual tree consists of a hierarchy of solid binary trees connected by dashed edges. Each node has a pointer to its parent, and to its left and right children (see Figure 12.4).

12.5.2 Solid Trees

Recall that each path is converted into a binary tree. Parent (say y) of a node (say x) in the path is the in-order (symmetric order) successor of that node (x) in the solid tree. However, if x is the last node (in symmetric order) in the solid sub-tree then its parent path will be the parent of the root of the solid sub-tree containing it (see Figure 12.4). Formally, $\text{Parent}_{\text{path}}(v) = \text{Node}(\text{Inorder}(v) + 1)$.

Note that for any node v, all nodes in the left sub-tree will have smaller inorder numbers and those in the right sub-tree will have larger inorder numbers. This ensures that all nodes in the left subtree are descendants and all nodes in the right sub-tree are ancestors. Thus, the parent (in the binary tree) of a left child will be an ancestor (in the original tree). But, parent (in the binary tree) of a right child is a descendant (in the original tree). This order, helps us to carry out add_cost effectively.

We need some definitions or notation to proceed.

Let $\text{mincost}(x)$ be the cost of the node having the minimum key value among all descendants of x in the same solid sub-tree. Then in each node we store two fields $\delta\text{cost}(x)$ and $\delta\text{min}(x)$. We define,
$\delta\text{min}(x) = \text{cost}(x) - \text{mincost}(x)$. And,

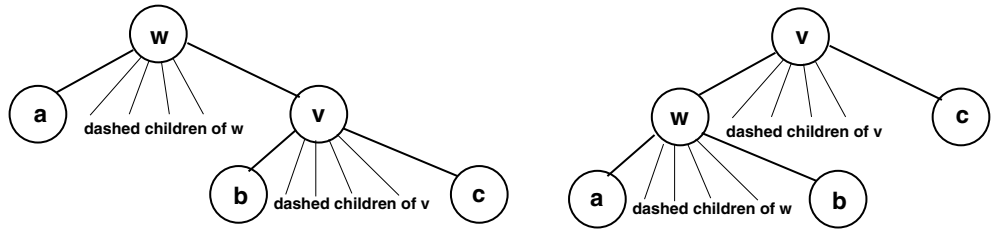

FIGURE 12.5: Rotation in Solid Trees— rotation of edge (v, w).

$$\delta\text{cost}(x) = \begin{cases} \text{cost}(x) - \text{cost}(\text{parent}(x)) & \text{if } x \text{ has a solid parent} \\ \text{cost}(x) & \text{otherwise } (x \text{ is a solid tree root}) \end{cases}$$

We will also store, size(x), the number of descendants (both solid and dashed) in virtual tree in incremental manner.

$$\delta\text{size}(x) = \begin{cases} \text{size}(\text{parent}(x)) - \text{size}(x) & \text{if } x \text{ is not the root of a virtual tree} \\ \text{size}(x) & \text{otherwise} \end{cases}$$

Thus, δsize(x) is number of descendants of parent(x), not counting the descendants of x.

FACT 12.2 *$\delta min(x) - \delta cost(x)$ =cost(parent(x))- mincost(x).*

Thus, if u and v are solid children of node z, then
mincost(z) = min{cost(z),mincost(v),mincost(w)}, or,
δmin(z) =cost(z)−mincost(z) = max{0,cost(z)−mincost(v),cost(z)−mincost(w).}
Using Fact 12.2, and the fact z =parent(u) =parent(v), we have

FACT 12.3 *If u and v are children of z, then*
$\delta min(z) = \max\{0, \delta min(u) - \delta cost(u), \delta min(v) - \delta cost(v)\}$.

For linking and cutting trees, we need two primitive operations— rotation and splicing.

12.5.3 Rotation

Let us discuss rotation first (see Figure 12.5).

Let w be the parent of v in the solid tree, then rotation of the solid edge $(v, p(v)) \equiv (v, w)$ will make $w = p(v)$ a child of v. Rotation does not have any effect on the middle children. Let a be the left solid child of w and v be the right solid child of w.

Let "non-primes" denote the values before the rotation and "primes" the values after the rotation of the solid edge (v, w). We next show that the new values $\delta\text{cost}', \delta\text{min}'$ and $\delta\text{size}'$, can be calculated in terms of old known values.

We assume that b is the left solid child of v and c is the right solid child of v.

First we calculate the new δcost′ values in terms of old δcost values. From Figure 12.5,
$\delta\text{cost}'(v)$ =cost(v)−cost(parent′(v))
=cost(v)−cost(parent(w))
=cost(v)−cost(w)+cost(w)−cost(parent(w))
= δcost(v) + δcost(w).

$\delta\text{cost}'(w) = \text{cost}(w) - \text{cost}(v)$
$= -\delta\text{cost}'(v).$

$\delta\text{cost}'(b) = \text{cost}(b) - \text{cost}(w)$
$= \text{cost}(b) - \text{cost}(v) + \text{cost}(v) - \text{cost}(w)$
$= \delta\text{cost}(b) + \delta\text{cost}(v).$

Finally,
$\delta\text{cost}'(a) = \delta\text{cost}(a)$ and $\delta\text{cost}'(c) = \delta\text{cost}(c)$.

We next compute $\delta\text{min}'$ values in terms of δmin and δcost.
$\delta\text{min}'(v) = \text{cost}(v) - \text{mincost}'(v)$
$= \text{cost}(v) - \text{mincost}(w)$
$= \text{cost}(v) - \text{cost}(w) + \text{cost}(w) - \text{mincost}(w)$
$= \delta\text{cost}(v) + \delta\text{min}(w).$

$\delta\text{min}(\ )$ of all nodes other than w will remain same, and for w, from Fact 12.3, we have,
$\delta\text{min}'(w) = \max\{0, \delta\text{min}'(a) - \delta\text{cost}'(a), \delta\text{min}'(b) - \delta\text{cost}'(b)\}$
$= \max\{0, \delta\text{min}(a) - \delta\text{cost}(a), \delta\text{min}(b) - \delta\text{cost}(b) - \delta\text{cost}(v)\}$

We finally compute $\delta\text{size}'$ in terms of δsize.
$\delta\text{size}'(w) = \text{size}'(\text{parent}'(w)) - \text{size}'(w)$
$= \text{size}'(v) - \text{size}'(w)$ (see Figure 12.5)
$= \text{size}(v) - \text{size}(b)$ (see Figure 12.5)
$= \delta\text{size}(b).$

If z is parent(w), then size(z) is unchanged.
$\delta\text{size}'(v) = \text{size}'(\text{parent}(v)) - \text{size}'(v)$
$= \text{size}(z) - \text{size}'(v)$
$= \text{size}(z) - \text{size}(w)$ as $\text{size}'(v) = \text{size}(w)$
$= \delta\text{size}(w).$

For all other nodes (except v and w), the number of descendants remains the same, hence, $\text{size}'(x) = \text{size}(x)$. Hence, for all $x \notin \{v, w\}$,
$\text{size}'(x) = \text{size}(x)$ or
$\text{size}(\text{parent}(x)) - \delta\text{size}(x) = \text{size}'(\text{parent}'(x)) - \delta\text{size}'(x)$ or
$\delta\text{size}'(x) = -\text{size}(\text{parent}(x)) + \delta\text{size}(x) + \text{size}'(\text{parent}'(x)).$

Observe that for any child x of v or w, size of parent changes. In particular,
$\delta\text{size}'(a) = -\text{size}(w) + \delta\text{size}(a) + \text{size}'(w)$
$= -\text{size}'(v) + \delta\text{size}(a) + \text{size}'(w)$
$= -\delta\text{size}'(w) + \delta\text{size}(a) = \delta\text{size}(a) - \delta\text{size}'(w)$
$= \delta\text{size}(a) - \delta\text{size}(b)$

$\delta\text{size}'(c) = -\text{size}(v) + \delta\text{size}(c) + \text{size}'(v)$
$= \text{size}(w) - \text{size}(v) + \delta\text{size}(c)$ as $\text{size}'(v) = \text{size}(w)$
$= \delta\text{size}(v) + \delta\text{size}(c).$

And finally,
$\delta\text{size}'(b) = -\text{size}(v) + \delta\text{size}(b) + \text{size}'(w)$
$= \text{size}(w) - \text{size}(v) + \delta\text{size}(b) + \text{size}'(w) - \text{size}(w)$
$= \delta\text{size}(v) + \delta\text{size}(b) + \text{size}'(w) - \text{size}'(v)$
$= \delta\text{size}(v) + \delta\text{size}(b) - \delta\text{size}'(w)$
$= \delta\text{size}(v).$

12.5.4 Splicing

Let us next look at the other operation, splicing. Let w be the root of a solid tree. And let v be a child of w connected by a dashed edge. If u is the left most child of w, then splicing at a dashed child v, of a solid root w, makes v the left child of w. Moreover the previous left-child u, now becomes a dashed child of w. Thus, informally speaking splicing makes a node the leftmost child of its parent (if the parent is root) and makes the previous leftmost child of parent as dashed.

We next analyse the changes in "cost" and "size" of various nodes after splicing at a dashed child v of solid root w (whose leftmost child is u). As before, "non-primes" denote the values before the splice and "primes" the values after the splice.

As v was a dashed child of its parent, it was a root earlier (in some solid tree). And as w is also a root,
$\delta\text{cost}'(v) = \text{cost}(v) - \text{cost}(w)$
$= \delta\text{cost}(v) - \delta\text{cost}(w)$.

And as u is now the root of a solid tree,
$\delta\text{cost}'(u) = \text{cost}(u)$
$= \delta\text{cost}(u) + \text{cost}(w)$
$= \delta\text{cost}(u) + \delta\text{cost}(w)$.

Finally, $\delta\text{min}'(w) = \max\{0, \delta\text{min}(v) - \delta\text{cost}'(v), \delta\text{min}(\text{right}(w))\text{-}\delta\text{cost}(\text{right}(w))\}$
All other values are clearly unaffected.

As no rotation is performed, δsize( ) also remains unchanged, for all nodes.

12.5.5 Splay in Virtual Tree

In virtual tree, some edges are solid and some are dashed. Usual splaying is carried out only in the solid trees. To splay at a node x in the virtual tree, following method is used. The algorithm looks at the tree three times, once in each pass, and modifies it. In first pass, by splaying only in the solid trees, starting from the node x, the path from x to the root of the overall tree, becomes dashed. This path is made solid by splicing. A final splay at node x will now make x the root of the tree. Less informally, the algorithm is as follows:

Algorithm for Splay(x)

Pass 1 Walk up the virtual tree, but splaying is done only within solid sub-tree. At the end of this pass, the path from x to root becomes dashed.

Pass 2 Walk up from node x, splicing at each proper ancestor of x. After this step, the path from x to the root becomes solid. Moreover, the node x and all its children in the original tree (the one before pass 1) now become left children.

Pass 3 Walk up from node x to the root, splaying in the normal fashion.

12.5.6 Analysis of Splay in Virtual Tree

Weight of each node in the tree is taken to be the same (say) 1. Size of a node is total number of descendants— both solid and dashed. And the rank of a node as before is $\text{rank}(x) = \log(\text{size}(x))$. We choose $\alpha = 2$, and hence the potential becomes, potential$= 2\sum_x \text{rank}(x)$. We still have to fix β. Let us analyze the complexity of each pass.

Pass 1 We fix $\beta = 1$. Thus, from Lemma 12.1, the amortized cost of single splaying is at most $6(r(t) - r(x)) + 1$. Hence, the total cost of all splays in this pass will be

$\leq 6(r(t_1) - r(x)) + 1 + 6(r(t_2) - r(p(t_1)) + 1 + \cdots + 6(r(t_k) - r(p(t_{k-1}))) + 1$
$\leq (6(r(t_1) - r(x)) + +6(r(t_k) - r(p(t_{k-1})))) + k.$
Here, k is number of solid trees in path from x to root. Or the total cost
$\leq k + (6(r(\text{root}) - r(x))) - 6(r(p(t_{k-1})) - r(t_{k-1}) + \cdots + r(p(t_1)) - r(t_1)))$
Recall that the size includes those of virtual descendants, hence each term in the bracket is non-negative. Or the total cost
$\leq k + 6(r(\text{root}) - r(x))$
Note that the depth of node x at end of the first pass will be k.

Pass 2 As no rotations are performed, actual time is zero. Moreover as there are no rotations, there is no change in potential. Hence, amortized time is also zero. Alternatively, time taken to traverse k-virtual edges can be accounted by incorporating that in β in pass 3.

REMARK 12.3 This means, that in effect, this pass can be done together with Pass 1.

Pass 3 In pass 1, k extra rotations are performed, (there is a $+k$ factor), thus, we can take this into account, by charging, 2 units for each of the k rotation in pass 3, hence we set $\beta = 2$. Clearly, the number of rotations, is exactly "k". Cost will be $6 \log n + 2$. Thus, in effect we can now neglect the $+k$ term of pass 1.

Thus, total cost for all three passes is $12 \log n + 2$.

12.5.7 Implementation of Primitives for Linking and Cutting Trees

We next show that various primitives for linking and cutting trees described in the beginning of this section can be implemented in terms of one or two calls to a single basic operation—"splay". We will discuss implementation of each primitive, one by one.

find_cost(v) We are required to find the value stored in the node v. If we splay at node v, then node v becomes the root, and δcost(v) will give the required value. Thus, the implementation is

> splay(v) and return the value at node v

find_root(v) We have to find the root of the tree containing the node v. Again, if we splay at v, then v will become the tree root. The ancestors of v will be in the right subtree, hence we follow right pointers till root is reached. The implementation is:

> splay(v), follow right pointers till last node of solid tree, say w is reached, splay(w) and return(w).

find_min(v) We have to find the node having the minimum value, on the path from v till the root of the tree containing v; in case of ties, we have to choose the node closest to the root. We again splay at v to make v the root, but, this time, we also keep track of the node having the minimum value. As these values are stored in incremental manner, we have to compute the value by an "addition" at each step.

> splay(v), use δcost( ) and δmin( ) fields to walk down to the last minimum cost node after v, in the solid tree, say w, splay(w) and return(w).

add_cost$(v, \delta x)$ We have to add a real number δx to the values stored in each and every ancestors of node v. If we splay at node v, then v will become the root and all ancestors of v will be in the right subtree. Thus, if we add δx to $\delta\text{cost}(v)$, then in effect, we are adding this value not only to all ancestors (in right subtree) but also to the nodes in the left subtree. Hence, we subtract δx from $\delta\text{cost}(\ )$ value of left child of v. Implementation is:

splay(v), add δx to $\delta\text{cost}(v)$, subtract δx from $\delta\text{cost}(\text{LCHILD}(v))$ and return

find_size(v) We have to find the number of nodes in the tree containing the node v. If we splay at the node v, then v will become the root and by definition of δsize, $\delta\text{size}(v)$ will give the required number.

splay(v) and return($\delta\text{size}(v)$).

link(v, w) If v is a root of a tree, then we have to make the tree rooted at v a child of node w.

Splay(w), and make v a middle (dashed) child of w. Update $\delta\text{size}(v)$ and $\delta\text{size}(w)$, etc.

cut(v) If v, is not a root, then we have to delete the edge from node v to its parent, thus making v a root. The implementation of this is also obvious:

splay(v), add $\delta\text{cost}(v)$ to $\delta\text{cost}(\text{RCHILD}(v))$, and break link between $\text{RCHILD}(v)$ and v. Update $\delta\text{min}(v)$, $\delta\text{size}(v)$ etc.

12.6 Case Study: Application to Network Flows

We next discuss application of linking and cutting trees to the problem of finding maximum flow in a network. Input is a directed graph $G = (V, E)$. There are two distinguished vertices s (source) and t (sink). We need a few definitions and some notations[1, 6]. Most of the results in this case-study are from[1, 6].

PreFlow $g(*, *)$ is a real valued function having following properties:

Skew-Symmetry: $g(u, v) = -g(v, u)$

Capacity Constraint: $g(u, v) \leq c(u, v)$

Positive-Flow Excess: $e(v) \equiv \sum_{w=1}^{n} g(v, w) \geq 0$ for $v \neq s$

Flow-Excess Observe that flow-excess at node v is $e(v) = \sum_{w=1}^{n} g(w, v)$ if $v \neq s$ and flow excess at source s is $e(s) = \infty$

Flow $f(*, *)$ is a real valued function having following additional property

Flow Conservation: $\sum_{w=1}^{n} f(v, w) = 0$ for $v \notin \{s, t\}$

Preflow: f is a preflow.

Value of flow: $|f| = \sum_{w=1}^{n} f(s, w)$, the net flow out of source.

REMARK 12.4 If $(u, v) \notin E$, then $c(u, v) = c(v, u) = 0$. Thus, $f(u, v) \leq c(u, v) = 0$ and $f(v, u) \leq 0$. By skew-symmetry, $f(u, v) = 0$

Cut Cut $(S, \overline{S})$ is a partition of vertex set, such that $s \in S$ and $t \in \overline{S}$

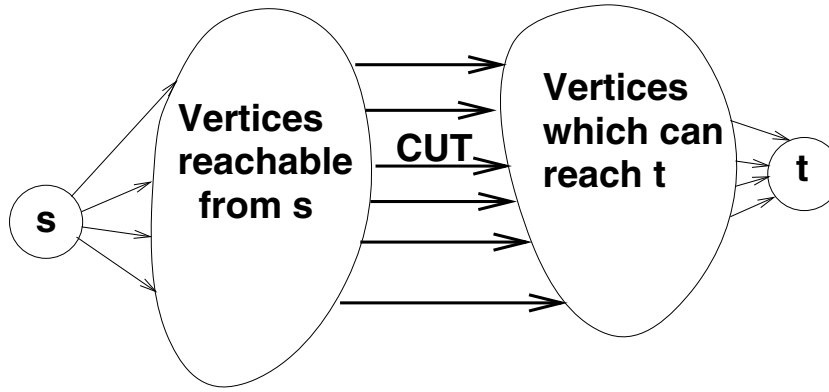

FIGURE 12.6: $s - t$ Cut.

Capacity of Cut $c(S, \overline{S}) = \sum_{v \in S, w \in \overline{S}} c(v, w)$

Pre-Flow across a Cut $g(S, \overline{S}) = \sum_{v \in S, w \notin S} g(v, w)$

Residual Capacity If g is a flow or preflow, then the residual capacity of an edge (v, w) is $r_g(v, w) = c(v, w) - g(v, w)$.

Residual Graph G_g contains same set of vertices as the original graph G, but only those edges for which residual capacity is positive; these are either the edges of the original graph or their reverse edges.

Valid Labeling A valid labeling $d(\ )$ satisfies following properties:

1. $d(t) = 0$
2. $d(v) > 0$ if $v \neq t$
3. if (v, w) is an edge in residual graph then $d(w) \geq d(v) - 1$.

A trivial labeling is $d(t) = 0$ and $d(v) = 1$ if $v \neq t$.

REMARK 12.5 As for each edge (v, w), $d(v) \leq d(w) + 1$, $\text{dist}(u, t) \geq d(u)$. Thus, label of every vertex from which t is reachable, is at most $n - 1$.

Active Vertex A vertex $v \neq s$ is said to be active if $e(v) > 0$.

The initial preflow is taken to be $g(s, v) = c(s, v)$ and $g(u, v) = 0$ if $u \neq s$.

Flow across a Cut Please refer to Figure 12.6. Observe that flow conservation is true for all vertices except s and t. In particular sum of flow (total flow) into vertices in set $S - \{s\}$ (set shown between s and cut) is equal to $|f|$ which must be the flow going out of these vertices (into the cut). And this is the flow into vertices (from cut) in set $\overline{S} - \{t\}$ (set after cut before t) which must be equal to the flow out of these vertices into t. Thus, the flow into t is $|f|$ which is also the flow through the cut.

FACT 12.4 *As,* $|f| = f(S, \overline{S}) = \sum_{v \in S, w \notin S} f(v, w) \leq \sum_{v \in S, w \notin S} c(v, w) = c(S, \overline{S})$

Thus, maximum value of flow is less than minimum capacity of any cut.

THEOREM 12.4 *[Max-Flow Min-Cut Theorem]* $\max |f| =$ *minimum cut*

Proof Consider a flow f for which $|f|$ is maximum. Delete all edges for which $(f(u,v) == c(u,v))$ to get the residual graph. Let S be the set of vertices reachable from s in the residual graph. Now, $t \notin S$, otherwise there is a path along which flow can be increased, contradicting the assumption that flow is maximum. Let $\overline{S}$ be set of vertices not reachable from s. $\overline{S}$ is not empty as $t \in S$. Thus, $(S, \overline{S})$ is an $s-t$ cut and as all edges (v,w) of cut have been deleted, $c(v,w) = f(v,w)$ for edges of cut.
$|f| = \sum_{v \in S, w \notin S} f(v,w) = \sum_{v \in S, w \notin S} c(v,w) = c(S, \overline{S})$

Push(v,w)

/* v is an active vertex and (v,w) an edge in residual graph with $d(w) = d(v)-1$ */
Try to move excess from v to w, subject to capacity constraints, i.e., send $\delta = \min\{e(v), r_g(v,w))$ units of flow from v to w.
/* $g(v,w) = g(v,w) + \delta$; $e(v) = e(v) - \delta$ and $e(w) = e(w) + \delta$; */

If $\delta = r_g(v,w)$, then the push is said to be *saturating*.

Relabel(v)

For $v \neq s$, the new distance label is

$$d(v) = \min\{d(w) + 1 | (v,w) \text{ is a residual edge } \}$$

Preflow-Push Algorithms

Following are some of the properties of preflow-push algorithms:

1. If relabel v results in a new label, $d(v) = d(w^*) + 1$, then as initial labeling was valid, $d_{\text{old}}(v) \leq d_{\text{old}}(w^*) + 1$. Thus labels can only increase. Moreover, the new labeling is clearly valid.
2. If push is saturating, edge (v,w) may get deleted from the graph and edge (w,v) will get added to the residual graph, as $d(w) = d(v) - 1$, $d(v) = d(w) + 1 \geq d(w) - 1$, thus even after addition to the residual graph, conditions for labeling to be valid are satisfied.
3. As a result of initialization, each node adjacent to s gets a positive excess. Moreover all arcs out of s are saturated. In other words in residual graph there is no path from s to t. As distances can not decrease, there can never be a path from s to t. Thus, there will be no need to push flow again out of s.
4. By definition of pre-flow, flow coming into a node is more than flow going out. This flow must come from source. Thus, all vertices with positive excess are reachable from s (in the original network). Thus, as s is initially the only node, at any stage of the algorithm, there is a path P_v to a vertex v (in the original network) along which pre-flow has come from s to v. Thus, in the residual graph, there is reverse path from v to s.
5. Consider a vertex v from which there is a path till a vertex X. As we trace back this path from X, then distance label $d(\ )$ increases by at most one. Thus, $d(v)$ can be at most $\text{dist}(v,X)$ larger than $d(X)$. That is $d(v) \leq d(X) + \text{dist}(v,X)$
6. As for vertices from which t is not reachable, s is reachable, $d(v) \leq d(s) + \text{dist}(s,v) = n + (n-1) = 2n-1$ (as $d(s) = n$).

Thus, maximum label of any node is $2n-1$.

FACT 12.5 *As label of t remains zero, and label of other vertices only increase, the number of Relabels, which result in change of labels is $(n-1)^2$. In each relabel operation we may have to look at degree(v) vertices. As, each vertex can be relabeled at most $O(n)$ times, time for relabels is $\sum O(n) \times$ degree$(v) = O(n) \times \sum$ degree$(v) = O(n) \times O(m) = O(nm)$*

FACT 12.6 *If a saturating push occurs from u to v, then $d(u) = d(v)+1$ and edge (u,v) gets deleted, but edge (v,u) gets added. Edge (u,v) can be added again only if edge (v,u) gets saturated, i.e., $d_{now}(v) = d_{now}(u)+1 \geq d(u)+1 = d(v)+2$. Thus, the edge gets added only if label increases by 2. Thus, for each edge, number of times saturating push can occur is $O(n)$. So the total number of saturating pushes is $O(nm)$.*

REMARK 12.6 Increase in label of $d(u)$ can make a reverse flow along all arcs (x,u) possible, and not just (v,u); in fact there are at most degree(u) such arcs. Thus, number of saturating pushes are $O(nm)$ and not $O(n^2)$.

FACT 12.7 *Consider the point in time when the algorithm terminates, i.e., when pushes or relabels can no longer be applied. As excess at s is ∞, excess at s could not have been exhausted. The fact that push/relabels can not be applied means that there is no path from s to t. Thus, $\overline{S}_g$, the set of vertices from which t is reachable, and S_g, set of vertices from which s is reachable, form an $s-t$ cut.*

Consider an edge (u,v) with $u \in S_g$ and $v \in \overline{S}_g$. As t is reachable from v, there is no excess at v. Moreover, by definition of cut, the edge is not present in residual graph, or in other words, flow in this edge is equal to capacity. By Theorem 12.4, the flow is the maximum possible.

12.7 Implementation Without Linking and Cutting Trees

Each vertex will have a list of edges incident at it. It also has a pointer to *current edge* (candidate for pushing flow out of that node). Each edge (u,v) will have three values associated with it $c(u,v)$, $c(v,u)$ and $g(u,v)$.

Push/Relabel(v)

Here we assume that v is an active vertex and (v,w) is current edge of v.

If $(d(w) == d(v)-1)$&& $(r_g(v,w) > 0$) then send $\delta = \min\{e(v), r_g(v,w)\}$ units of flow from v to w.
Else if v has no next edge, make first edge on edge list the current edge and Relabel(v): $d(v) = \min\{d(w)+1 | (v,w)$ is a residual edge$\}$ /* this causes $d(v)$ to increase by at least one */
Else make the next edge out of v, the current edge.

Relabeling v, requires a single scan of v's edge list. As each relabeling of v, causes $d(v)$ to go up by one, the number of relabeling steps (for v) are at most $O(n)$, each step takes $O($degree$(v))$ time. Thus, total time for all relabellings will be:

$O(\sum n\text{degree}(v)) = O(n \sum \text{degree}) = O(n \times 2m) = O(nm)$. Each non-saturating push clearly takes $O(1)$ time, thus time for algorithm will be $O(nm)+O(\#\text{non saturating pushes})$.

Discharge(v)

Keep on applying Push/Relabel(v) until either

1. entire excess at v is pushed out, OR,
2. label(v) increases.

FIFO/Queue

Initialize a queue "Queue" to contain s.

Let v be the vertex in front of Queue. Discharge(v), if a push causes excess of a vertex w to become non-zero, add w to the rear of the Queue.

Let phase 1, consist of discharge operations applied to vertices added to the queue by initialization of pre-flow.

Phase $(i + 1)$ consists of discharge operations applied to vertices added to the queue during phase i.

Let $\Phi = \max\{d(v)|v \text{ is active }\}$, with maximum as zero, if there are no active vertices. If in a phase, no relabeling is done, then the excess of all vertices which were in the queue has been moved. If v is any vertex which was in the queue, then excess has been moved to a node w, with $d(w) = d(v)-1$. Thus, $\max\{d(w)|w \text{ has now become active}\} \leq \max\{d(v)-1|v \text{ was active }\} = \Phi - 1$.

Thus, if in a phase, no relabeling is done, Φ decreases by at least one. Moreover, as number of relabeling steps are bounded by $2n^2$, number of passes in which relabeling takes place is at most $2n^2$.

Only way in which Φ can increase is by relabeling. Since the maximum value of a label of any active vertex is $n - 1$, and as a label never decreases, the total of all increases in Φ is $(n - 1)^2$.

As Φ decreases by at least one in a pass in which there is no relabeling, number of passes in which there is no relabeling is $(n - 1)^2 + 2n^2 \leq 3n^2$.

FACT 12.8 *Number of passes in FIFO algorithm is $O(n^2)$.*

12.8 FIFO: Dynamic Tree Implementation

Time for non-saturating push is reduced by performing a succession of pushes along a single path in one operation. After a non-saturating push, the edge continues to be admissible, and we know its residual capacity. [6]

Initially each vertex is made into a one vertex node. Arc of dynamic trees are a subset of admissible arcs. Value of an arc is its admissible capacity (if (u,parent(u)) is an arc, value of arc will be stored at u). Each active vertex is a tree root.

Vertices will be kept in a queue as in FIFO algorithm, but instead of discharge(v), Tree-Push(v), will be used. We will further ensure that tree size does not exceed k (k is a parameter to be chosen later). The Tree-Push procedure is as follows:

Tree-Push(v)

/* v is active vertex and (v, w) is an admissible arc */

1. /* link trees rooted at v and the tree containing w by making w the parent of v, if the tree size doesn't exceed k */.
   if v is root and (find_size(v)+find_size(w))$\leq k$, then link v and w. Arc (v, w) gets the value equal to the residual capacity of edge (v, w)
2. if v is root but find_size(v)+find_size(w) $> k$, then push flow from v to w.
3. if v is not a tree root, then send $\delta = \min\{e(v)$,find_cost(find_min(v))$\}$ units of flow from v, by add_cost$(v, -\delta)$ /* decrease residual capacity of all arcs */ and while v is not a root and find_cost(find_min(v))== 0 do

   { z := find_min(v); cut(z); /* delete saturated edge */
   $f(z$,parent(z)) := $c(z$,parent(z));
   /* in saturated edge, flow=capacity */
   f(parent(z), z) := $-c(z$,parent(z));
   }
4. But, if arc(v, w) is not admissible, replace (v, w), as current edge by next edge on v's list. If v has no next-edge, then make the first edge, the current edge and cut-off all children of v, and relabel(v).

Analysis

1. Total time for relabeling is $O(nm)$.
2. Only admissible edges are present in the tree, and hence if an edge (u, v) is cut in step (3) or in step (4) then it must be admissible, i.e., $d(u) = d(v) + 1$. Edge (v, u) can become admissible and get cut, iff, $d_{\text{then}}(v) = d_{\text{then}}(u)+1 \geq d(u)+1 = d(v) + 2$. Thus, the edge gets cut again only if label increases by 2. Thus, for each edge, number of times it can get cut is $O(n)$. So total number of cuts are $O(nm)$.
3. As initially, there are at most n-single node trees, number of links are at most n+#no_of_cuts= $n + O(nm) = O(nm)$.

Moreover, there is at most one tree operation for each relabeling, cut or link. Further, for each item in queue, one operation is performed. Thus,

LEMMA 12.2 The time taken by the algorithm is
$O(\log k \times (nm + \#\text{No\_of\_times\_an\_item\_is\_added\_to\_the\_queue}))$

Root-Nodes Let T_v denote the tree containing node v. Let r be a tree root whose excess has become positive. It can become positive either due to:

1. push from a non-root vertex w in Step 3 of the tree-push algorithm.
2. push from a root w in Step 2 /* find_size(w)+find_size(r) $> k$ */

REMARK 12.7 Push in Step 3 is accompanied by a cut (unless first push is non-saturating). As the number of cuts is $O(nm)$, number of times Step 3 (when first push is saturating) can occur is $O(nm)$. Thus, we need to consider only the times when first push was non-saturating, and the excess has moved to the root as far as push in Step 3 is concerned.

In either case let i be the pass in which this happens (i.e., w was added to the queue in pass $(i-1)$). Let I be the interval from beginning of pass $(i-1)$ to the time when $e(r)$ becomes positive.

Case 1: (T_w changes during I) T_w can change either due to link or cut. But number of times a link or a cut can occur is $O(nm)$. Thus, this case occurs at most $O(nm)$ time. Thus, we may assume that *T_w does not change during interval I.* Vertex w is added to the queue either because of relabeling of w, or because of a push in Step 2 from (say) a root v to w.

Case 2: (w is added because of relabeling) Number of relabeling steps are $O(n^2)$. Thus number of times this case occurs is $O(n^2)$. Thus, we may assume that w *was added to queue because of push* from root v to w in Step 2.

Case 3: (push from w was saturating) As the number of saturating pushes is $O(nm)$, this case occurs $O(nm)$ times. Thus we may assume that *push from w was non-saturating.*

Case 4: (edge (v,w) was not the current edge at beginning of pass $(i-1)$). Edge (v,w) will become the current edge, only because either the previous current edge (v,x) got saturated, or because of relabel(v), or relabel(x). Note, that if entire excess out of v was moved, then (v,w) will remain the current edge.
As number of saturating pushes are $O(nm)$ and number of relabeling are $O(n^2)$, this case can occur at most $O(nm)$ times. Thus, we may assume that (v,w) *was the current edge at beginning of pass* $(i-1)$.

Case 5: (T_v changes during interval I) T_v can change either due to link or cut. But the number of times a link or a cut can occur is $O(nm)$. Thus, this case occurs at most $O(nm)$ time. Thus, we may assume that *T_v has not changed during interval I.*

Remaining Case: Vertex w was added to the queue because of a non-saturating push from v to w in Step 2 and (v,w) is still the current edge of v. Moreover, T_v and T_w do not change during the interval I.
A tree at beginning of pass $(i-1)$ can participate in only one pair (T_w, T_v) as T_w, because this push was responsible for adding w to the queue. Observe that vertex w is uniquely determined by r.
And, a tree at beginning of pass $(i-1)$ can participate in only one pair (T_w, T_v) as T_v, because (v,w) was the current edge out of root v, at beginning of pass $(i-1)$ (and is still the current edge). Thus, choice of T_v will uniquely determine T_w (and conversely).
Thus, as a tree T_x can participate once in a pair as T_v, and once as T_w, and the two trees are unchanged, we have $\sum_{(v,w)} |T_v| + |T_w| \leq 2n$ (a vertex is in at most one tree). As push from v to w was in Step 2, find_size(v)+find_size(w) $> k$, or $|T_v| + |T_w| > k$. Thus, the number of such pairs is at most $2n/k$.
But from Fact 12.8, as there are at most $O(n^2)$ passes, the number of such pairs are $O(n^3/k)$.

Non-Root-Nodes Let us count the number of times a non-root can have its excess made positive. Its excess can only be made positive as a result of push in Step 2. As the number of saturating pushes is $O(nm)$, clearly, $O(nm)$ pushes in Step 2 are saturating.

If the push is non-saturating, then entire excess at that node is moved out, hence it can happen only once after a vertex is removed from Queue. If v was not a root when it was added to the queue, then it has now become a root only because of a cut. But number of cuts is $O(nm)$. Thus, we only need to consider the case when v was a root when it was

added to the queue. The root was not earlier in queue, because either its excess was then zero, or because its distance label was low. Thus, now either

1. distance label has gone up— this can happen at most $O(n^2)$ times, or
2. now its excess has become positive. This by previous case can happen at most $O(nm + (n^3/k))$ times.

Summary If k is chosen such that $nm = n^3/k$, or $k = n^2/m$, time taken by the algorithm is $O(nm \log(n^2/m))$.

12.9 Variants of Splay Trees and Top-Down Splaying

Various variants, modifications and generalization of Splay trees have been studied, see for example [2, 11, 12, 14]. Two of the most popular "variants" suggested by Sleator and Tarjan [13] are "semi-splay" and "simple-splay" trees. In simple splaying the second rotation in the "zig-zag" case is done away with (i.e., we stop at the middle figure in Figure 12.3). Simple splaying can be shown to have a larger constant factor both theoretically [13] and experimentally [11]. In semi-splay [13], in the zig-zig case (see Figure 12.2) we do only the first rotation (i.e., stop at the middle figure) and continue splaying from node y instead of x. Sleator and Tarjan observe that for some access sequences "semi-splaying" may be better but for some others the usual splay is better.

"Top-down" splay trees [13] are another way of implementing splay trees. Both the trees coincide if the node being searched is at an even depth [11], but if the item being searched is at an odd depth, then the top-down and bottom-up trees may differ ([11, Theorem 2]).

Some experimental evidence suggests [3] that top-down splay trees [11, 13] are faster in practice as compared to the normal splay trees, but some evidence suggests otherwise [16].

In splay trees as described, we first search for an item, and then restructure the tree. These are called "bottom-up" splay trees. In "top-down" splay trees, we look at two nodes at a time, while searching for the item, and also keep restructuring the tree until the item we are looking for has been located.

Basically, the current tree is divided into three trees, while we move down two nodes at a time searching for the query item

left tree: Left tree consists of items known to be smaller than the item we are searching.

right tree: Similarly, the right tree consists of items known to be larger than the item we are searching.

middle tree: this is the subtree of the original tree rooted at the current node.

Basically, the links on the access path are broken and the node(s) which we just saw are joined to the bottom right (respectively left) of the left (respectively right) tree if they contain item greater (respectively smaller) than the item being searched. If both nodes are left children or if both are right children, then we make a rotation before breaking the link. Finally, the item at which the search ends is the only item in the middle tree and it is made the root. And roots of left and right trees are made the left and right children of the root.

Acknowledgment

I wish to thank N. Nataraju, Priyesh Narayanan, C. G. Kiran Babu S. and Lalitha S. for careful reading of a previous draft and their helpful comments.

References

[1] Ravindra K. Ahuja, Thomas L. Magnanti and James B. Orlin, Network Flows (Theory, Algorithms and Applications), Prentice Hall, Inc, Englewood Cliffs, NJ, USA, 1993.

[2] S. Albers and M. Karpinkski, Randomized splay trees: Theoretical and experimental results, *Infor. Proc. Lett.*, vol 81, 2002, pp 213-221.

[3] J. Bell and G. Gupta, An Evaluation of Self-adjusting Binary Search Tree Techniques, *Software-Practice and Experience*, vol 23, 1993, 369-382.

[4] T. Bell and D. Kulp, Longest-match String Searching for Ziv-Lempel Compression, *Software-Practice and Experience*, vol 23, 1993, 757-771.

[5] R.Cole, On the dynamic finger conjecture for splay trees. Part II: The Proof, *SIAM J. Comput.*, vol 30, no. 1, 2000, pp 44-5.

[6] A.V.Goldberg and R.E.Tarjan, "A New Approach to the Maximum-Flow Problem, *JACM*, vol 35, no. 4, October 1988, pp 921-940.

[7] D. Grinberg, S. Rajagopalan, R. Venkatesh and V. K. Wei, Splay Trees for Data Compression, *SODA*, 1995, 522-530

[8] J. Iacono, Key Independent Optimality, *ISAAC 2002*, LNCS 2518, 2002, pp 25-31.

[9] D. W. Jones, Application of Splay Trees to data compression, *CACM*, vol 31, 1988, 996-1007.

[10] A. Moffat, G.Eddy and O.Petersson, Splaysort: Fast, Versatile, Practical, *Software-Practice and Experience*, vol 26, 1996, 781-797.

[11] E. Mäkinen, On top-down splaying, *BIT*, vol 27, 1987, 330-339.

[12] M.Sherk, Self-Adjusting k-ary search trees, *J. Algorithms*, vol 19, 1995, pp 25-44.

[13] D. Sleator and R. E. Tarjan, Self-Adjusting Binary Search Trees, *JACM*, vol 32, 1985

[14] A. Subramanian, An explanation of splaying, *J. Algorithms*, vol 20, 1996, pp 512-525.

[15] R. E. Tarjan, *Data Structures and Network Algorithms*, SIAM 1983.

[16] H. E. Williams, J. Zobel and S. Heinz, Self-adjusting trees in practice for large text collections, *Software-Practice and Experience*, vol 31, 2001, 925-939.

13

Randomized Dictionary Structures

C. Pandu Rangan
Indian Institute of Technology, Madras

13.1 Introduction

In the last couple of decades, there has been a tremendous growth in using randomness as a powerful source of computation. Incorporating randomness in computation often results in a much simpler and more easily implementable algorithms. A number of problem domains, ranging from sorting to stringology, from graph theory to computational geometry, from parallel processing system to ubiquitous internet, have benefited from randomization in terms of newer and elegant algorithms. In this chapter we shall see how randomness can be used as a powerful tool for designing simple and efficient data structures. Solving a real-life problem often involves manipulating complex data objects by variety of operations. We use abstraction to arrive at a mathematical model that represents the real-life objects and convert the real-life problem into a computational problem working on the mathematical entities specified by the model. Specifically, we define *Abstract Data Type (ADT)* as a mathematical model together with a set of operations defined on the entities of the model. Thus, an algorithm for a computational problem will be expressed in terms of the steps involving the corresponding ADT operations. In order to arrive at a computer based implementation of the algorithm, we need to proceed further taking a closer look at the possibilities of implementing the ADTs. As programming languages support only a very small number of built-in types, any ADT that is not a built-in type must be represented in terms of the elements from built-in type and this is where the data structure plays a critical role. One major goal in the design of data structure is to render the operations of the ADT as efficient as possible. Traditionally, data structures were designed to minimize the worst-case costs of the ADT operations. When the worst-case efficient data structures turn out to be too complex and cumbersome to implement, we naturally explore alternative