III

Dictionary Structures

9

Hash Tables

Pat Morin
Carleton University

9.1 Introduction

A *set abstract data type (set ADT)* is an abstract data type that maintains a set S under the following three operations:

1. INSERT(x): Add the key x to the set.
2. DELETE(x): Remove the key x from the set.
3. SEARCH(x): Determine if x is contained in the set, and if so, return a pointer to x.

One of the most practical and widely used methods of implementing the set ADT is with *hash tables*.

Note that the three set ADT operations can easily be implemented to run in $O(\log n)$ time per operation using balanced binary search trees (See Chapter 10). If we assume that the input data are integers in the set $U = \{0, \ldots, u-1\}$ then they can even be implemented to run in sub-logarithmic time using data structures for integer searching (Chapter 39). However, these data structures actually do more than the three basic operations we require. In particular if we search for an element x that is not present in S then these data structures can report the smallest item in S that is larger than x (the *successor* of x) and/or the largest item in S that is smaller than x (the *predecessor* of x).

Hash tables do away with this extra functionality of finding predecessors and successors and only perform exact searches. If we search for an element x in a hash table and x is not present then the only information we obtain is that $x \notin S$. By dropping this extra functionality hash tables can give better performance bounds. Indeed, any reasonable hash table implementation performs each of the three set ADT operations in $O(1)$ expected time.

The main idea behind all hash table implementations discussed in this chapter is to store a set of $n = |S|$ elements in an array (the hash table) A of length $m \geq n$. In doing this, we require a function that maps any element x to an array location. This function is called a *hash function* h and the value $h(x)$ is called the *hash value* of x. That is, the element x gets stored at the array location $A[h(x)]$. The *occupancy* of a hash table is the ratio $\alpha = n/m$ of stored elements to the length of A.

The study of hash tables follows two very different lines. Many implementations of hash tables are based on the *integer universe assumption*: All elements stored in the hash table come from the universe $U = \{0, \ldots, u-1\}$. In this case, the goal is to design a hash function $h : U \to \{0, \ldots, m-1\}$ so that for each $i \in \{0, \ldots, m-1\}$, the number of elements $x \in S$ such that $h(x) = i$ is as small as possible. Ideally, the hash function h would be such that each element of S is mapped to a unique value in $\{0, \ldots, m-1\}$. Most of the hash functions designed under the integer universe assumption are number-theoretic constructions. Several of these are described in Section 9.2.

Historically, the integer universe assumption seems to have been justified by the fact that any data item in a computer is represented as a sequence of bits that can be interpreted as a binary number. However, many complicated data items require a large (or variable) number of bits to represent and this make u the size of the universe very large. In many applications u is much larger than the largest integer that can fit into a single word of computer memory. In this case, the computations performed in number-theoretic hash functions become inefficient.

This motivates the second major line of research into hash tables. This research work is based on the *random probing assumption*random probing assumption: Each element x that is inserted into a hash table is a black box that comes with an infinite random *probe sequence* $x_0, x_1, x_2, \ldots$ where each of the x_i is independently and uniformly distributed in $\{0, \ldots, m-1\}$. Hash table implementations based on the random probing assumption are described in Section 9.3.

Both the integer universe assumption and the random probing assumption have their place in practice. When there is an easily computing mapping of data elements onto machine word sized integers then hash tables for integer universes are the method of choice. When such a mapping is not so easy to compute (variable length strings are an example) it might be better to use the bits of the input items to build a good pseudorandom sequence and use this sequence as the probe sequence for some random probing data structure.

To guarantee good performance, many hash table implementations require that the occupancy α be a constant strictly less than 1. Since the number of elements in a hash table changes over time, this requires that the array A be resized periodically. This is easily done, without increasing the amortized cost of hash table operations by choosing three constants $0 < \alpha_1 < \alpha_2 < \alpha_3 < 1$ so that, whenever n/m is not the interval (α_1, α_3) the array A is resized so that its size is n/α_2. A simple amortization argument (Chapter 1) shows that the amortized cost of this resizing is $O(1)$ per update (Insert/Delete) operation.

9.2 Hash Tables for Integer Keys

In this section we consider hash tables under the integer universe assumption, in which the key values x come from the universe $U = \{0, \ldots, u-1\}$. A *hash function* h is a function whose domain is U and whose range is the set $\{0, \ldots, m-1\}$, $m \leq u$. A hash function h is said to be a *perfect hash function* for a set $S \subseteq U$ if, for every $x \in S$, $h(x)$ is unique. A perfect hash function h for S is *minimal* if $m = |S|$, i.e., h is a bijection between S and $\{0, \ldots, m-1\}$. Obviously a minimal perfect hash function for S is desirable since it

allows us to store all the elements of S in a single array of length n. Unfortunately, perfect hash functions are rare, even for m much larger than n. If each element of S is mapped independently and uniformly to a random element of $\{0, \ldots, m-1\}$ then the birthday paradox (See, for example, Feller [27]) states that, if m is much less than n^2 then there will almost surely exist two elements of S that have the same hash value.

We begin our discussion with two commonly used hashing schemes that are heuristic in nature. That is, we can not make any non-trivial statements about the performance of these schemes when storing an arbitrary set S. We then discuss several schemes that have provably good performance.

9.2.1 Hashing by Division

In *hashing by division*, we use the hash function

$$h(x) = x \bmod m \ .$$

To use this hash function in a data structure, we maintain an array $A[0], \ldots, A[m-1]$ where each element of this array is a pointer to the head of a linked list (Chapter 2). The linked list L_i pointed to by the array element $A[i]$ contains all the elements x such that $h(x) = i$. This technique of maintaining an array of lists is called *hashing with chaining*.

In such a hash table, inserting an element x takes $O(1)$ time; we compute $i = h(x)$ and append (or prepend) x to the list L_i. However, searching for and/or deleting an element x is not so easy. We have to compute $i = h(x)$ and then traverse the list L_i until we either find x or reach the end of the list. The cost of this is proportional to the length of L_i. Obviously, if our set S consists of the elements $0, m, 2m, 3m, \ldots, nm$ then all elements are stored in the list L_0 and searches and deletions take linear time.

However, one hopes that such pathological cases do not occur in practice. For example, if the elements of S are uniformly and independently distributed in U and u is a multiple of m then the expected size of any list L_i is only n/m. In this case, searches and deletions take $O(1+\alpha)$ expected time. To help avoid pathological cases, the choice of m is important. In particular, m a power of 2 is usually avoided since, in a binary computer, taking the remainder modulo a power of 2 means simply discarding some high-order bits. Taking m to be a prime not too close to a power of 2 is recommended [37].

9.2.2 Hashing by Multiplication

The implementation of a hash table using *hashing by multiplication* is exactly the same as that of hashing by division except that the hash function

$$h(x) = \lfloor mxA \rfloor \bmod m$$

is used. Here A is a real-valued constant whose choice we discuss below. The advantage of the multiplication method is that the value of m is not critical. We can take m to be a power of 2, which makes it convenient for use on binary computers.

Although any value of A gives a hash function, some values of A are better than others. (Setting $A = 0$ is clearly not a good idea.)

Knuth [37] suggests using the *golden ratio* for A, i.e., setting

$$A = (\sqrt{5} - 1)/2 = 0.6180339887\ldots$$

This choice of A is motivated by a theorem, first conjectured by Oderfeld and later proven by Sẃierczkowski [59]. This theorem states that the sequence

$$mA \bmod m,\ 2mA \bmod m,\ 3mA \bmod m,\ \ldots,\ nmA \bmod m$$

partitions the interval $(0, m)$ into $n + 1$ intervals having only three distinct lengths. Furthermore, the next element $(n + 1)mA \bmod m$ in the sequence is always contained in one of the largest intervals.[1]

Of course, no matter what value of A we select, the pigeonhole principle implies that for $u \geq nm$ then there will always exist some hash value i and some $S \subseteq U$ of size n such that $h(x) = i$ for all $x \in S$. In other words, we can always find a set S all of whose elements get stored in the same list L_i. Thus, the worst case of hashing by multiplication is as bad as hashing by division.

9.2.3 Universal Hashing

The argument used at the end of the previous section applies equally well to any hash function h. That is, if the table size m is much smaller than the universe size u then for any hash function there is some large (of size at least $\lceil u/m \rceil$) subset of U that has the same hash value. To get around this difficulty we need a collection of hash functions from which we can choose one that works well for S. Even better would be a collection of hash functions such that, for any given S, most of the hash functions work well for S. Then we could simply pick one of the functions at random and have a good chance of it working well.

Let $\mathcal{H}$ be a collection of hash functions, i.e., functions from U onto $\{0, \ldots, m-1\}$. We say that $\mathcal{H}$ is *universal* if, for each $x, y \in U$ the number of $h \in \mathcal{H}$ such that $h(x) = h(y)$ is at most $|\mathcal{H}|/m$. Consider any $S \subseteq U$ of size n and suppose we choose a random hash function h from a universal collection of hash functions. Consider some value $x \in U$. The probability that any key $y \in S$ has the same hash value as x is only $1/m$. Therefore, the expected number of keys in S, not equal to x, that have the same hash value as x is only

$$n_{h(x)} = \begin{cases} (n-1)/m & \text{if } x \in S \\ n/m & \text{if } x \notin S \end{cases}$$

Therefore, if we store S in a hash table using the hash function h then the expected time to search for, or delete, x is $O(1 + \alpha)$.

From the preceding discussion, it seems that a universal collection of hash functions from which we could quickly select one at random would be very handy indeed. With such a collection at our disposal we get an implementation of the set ADT that has $O(1)$ insertion time and $O(1)$ expected search and deletion time.

Carter and Wegman [8] describe three different collections of universal hash functions. If the universe size u is a prime number[2] then

$$\mathcal{H} = \{h_{k_1,k_2,m}(x) = ((k_1 x + k_2) \bmod u)) \bmod m : 1 \leq k_1 < u, 0 \leq k_2 < u\}$$

[1] In fact, any irrational number has this property [57]. The golden ratio is especially good because it is not too close to a whole number.

[2] This is not a major restriction since, for any $u > 1$, there always exists a prime number in the set $\{u, u + 1, \ldots, 2u\}$. Thus we can enforce this assumption by increasing the value of u by a constant factor.

is a collection of universal hash functions. Clearly, choosing a function uniformly at random from $\mathcal{H}$ can be done easily by choosing two random values $k_1 \in \{1, \ldots, u-1\}$ and $k_2 \in \{0, \ldots, u-1\}$. Thus, we have an implementation of the set ADT with $O(1)$ expected time per operation.

9.2.4 Static Perfect Hashing

The result of Carter and Wegman on universal hashing is very strong, and from a practical point of view, it is probably the strongest result most people will ever need. The only thing that could be improved about their result is to make it deterministic, so that the running times of all operations are $O(1)$ *worst-case*. Unfortunately, this is not possible, as shown by Dietzfelbinger et al. [23].

Since there is no hope of getting $O(1)$ worst-case time for all three set ADT operations, the next best thing would be to have searches that take $O(1)$ worst-case time. In this section we describe the method of Fredman, Komlós and Szemerédi [28]. This is a static data structure that takes as input a set $S \subseteq U$ and builds a data structure of size $O(n)$ that can test if an element x is in S in $O(1)$ worst-case time. Like the universal hash functions from the previous section, this method also requires that u be a prime number. This scheme uses hash functions of the form

$$h_{k,m}(x) = (kx \bmod u)) \bmod m \ .^3$$

Let $B_{k,m}(S,i)$ be the number of elements $x \in S$ such that $h_{k,m}(x) = i$, i.e., the number of elements of S that have hash value i when using the hash function $h_{k,m}$. The function $B_{k,m}$ gives complete information about the distribution of hash values of S. The main lemma used by Fredman et al. is that, if we choose $k \in U$ uniformly at random then

$$\mathrm{E}\left[\sum_{i=0}^{m-1} \binom{B_{k,m}(S,i)}{2}\right] < \frac{n^2}{m} \ . \tag{9.1}$$

There are two important special cases of this result.

In the *sparse case* we take $m = n^2/\alpha$, for some constant $0 < \alpha < 1$. In this case, the expectation in (9.1) is less than α. Therefore, by Markov's inequality, the probability that this sum is greater than or equal to 1 is at most α. But, since this sum is a non-negative integer, then with probability at least $1-\alpha$ it must be equal to 0. In other words, with probability at least $1-\alpha$, $B_{k,m}(S,i) \leq 1$ for all $0 \leq i \leq m-1$, i.e., the hash function $h_{k,m}$ is perfect for S. Of course this implies that we can find a perfect hash function very quickly by trying a small number of random elements $k \in U$ and testing if they result in perfect hash functions. (The expected number of elements that we will have to try is only $1/(1-\alpha)$.) Thus, if we are willing to use quadratic space then we can perform searches in $O(1)$ worst-case time.

In the *dense case* we assume that m is close to n and discover that, for many values of k, the hash values are distributed fairly evenly among the set $1, \ldots, m$. More precisely, if we use a table of size $m = n$, then

$$\mathrm{E}\left[\sum_{i=0}^{m-1} B_{k,m}(S,i)^2\right] \leq 3n \ .$$

[3] Actually, it turns out that any universal hash function also works in the FKS scheme [16, Section 11.5].

By Markov's inequality this means that

$$\Pr\left\{\sum_{i=0}^{m-1} B_{k,m}(S,i)^2 \leq 3n/\alpha\right\} \geq 1-\alpha \ . \tag{9.2}$$

Again, we can quickly find a value of k satisfying (9.2) by testing a few randomly chosen values of k.

These two properties are enough to build a two-level data structure that uses linear space and executes searches in worst-case constant time. We call the following data structure the FKS-α data structure, after its inventors Fredman, Komlós and Szemerédi. At the top level, the data structure consists of an array $A[0],\ldots,A[m-1]$ where $m=n$. The elements of this array are pointers to other arrays $A_0,\ldots,A_{m-1}$, respectively. To decide what will be stored in these other arrays, we build a hash function $h_{k,m}$ that satisfies the conditions of (9.2). This gives us the top-level hash function $h_{k,m}(x) = (kx \bmod u) \bmod m$. Each element $x \in S$ gets stored in the array pointed to by $A[h_{k,m}(x)]$.

What remains is to describe how we use the arrays $A_0,\ldots,A_{m-1}$. Let S_i denote the set of elements $x \in S$ such that $h_{k,m}(s) = i$. The elements of S_i will be stored in A_i. The size of S_i is $n_i = B_{k,m}(S,i)$. To store the elements of S_i we set the size of A_i to $m_i = {n_i}^2/\alpha = B_{k,n}(S,i)^2/\alpha$. Observe that, by (9.2), all the A_i's take up a total space of $O(n)$, i.e., $\sum_{i=0}^{m-1} m_i = O(n)$. Furthermore, by trying a few randomly selected integers we can quickly find a value k_i such that the hash function h_{k_i,m_i} is perfect for S_i. Therefore, we store the element $x \in S_i$ at position $A_i[h_{k_i,m_i}(x)]$ and x is the unique element stored at that location. With this scheme we can search for any value $x \in U$ by computing two hash values $i = h_{k,m}(x)$ and $j = h_{k_i,m_i}(x)$ and checking if x is stored in $A_i[j]$.

Building the array A and computing the values of $n_0,\ldots,n_{m-1}$ takes $O(n)$ expected time since for a given value k we can easily do this in $O(n)$ time and the expected number of values of k that we must try before finding one that satisfies (9.2) is $O(1)$. Similarly, building each subarray A_i takes $O({n_i}^2)$ expected time, resulting in an overall expected running time of $O(n)$. Thus, for any constant $0 < \alpha < 1$, an FKS-α data structure can be constructed in $O(n)$ expected time and this data structure can execute a search for any $x \in U$ in $O(1)$ worst-case time.

9.2.5 Dynamic Perfect Hashing

The FKS-α data structure is nice in that it allows for searches in $O(1)$ time, in the worst case. Unfortunately, it is only static; it does not support insertions or deletions of elements. In this section we describe a result of Dietzfelbinger et al. [23] that shows how the FKS-α data structure can be made dynamic with some judicious use of partial rebuilding (Chapter 10).

The main idea behind the scheme is simple: be lazy at both the upper and lower levels of the FKS-α data structure. That is, rebuild parts of the data structure only when things go wrong. At the top level, we relax the condition that the size m of the upper array A is exactly n and allow A to have size anywhere between n and $2n$. Similarly, at the lower level we allow the array A_i to have a size m_i anywhere between ${n_i}^2/\alpha$ and $2{n_i}^2/\alpha$.

Periodically, we will perform a *global rebuilding* operation in which we remove all n elements from the hash table. Some elements which have previously been marked as deleted will be discarded, thereby reducing the value of n. We put the remaining elements in a list, and recompute a whole new FKS-($\alpha/2$) data structure for the elements in the list. This data structure is identical to the standard FKS-($\alpha/2$) data structure except that, at the top level we use an array of size $m = 2n$.

Searching in this data structure is exactly the same as for the static data structure. To search for an element x we compute $i = h_{k,m}(x)$ and $j = h_{k_i,m_i}(x)$ and look for x at location $A_i[j]$. Thus, searches take $O(1)$ worst-case time.

Deleting in this data structure is done in the laziest manner possible. To delete an element we only search for it and then mark it as deleted. We will use the convention that this type of deletion does not change the value of n since it does not change the number of elements actually stored in the data structure. While doing this, we also keep track of the number of elements that are marked as deleted. When this number exceeds $n/2$ we perform a global rebuilding operation. The global rebuilding operation takes $O(n)$ expected time, but only occurs during one out of every $n/2$ deletions. Therefore, the amortized cost of this operation is $O(1)$ per deletion.

The most complicated part of the data structure is the insertion algorithm and its analysis. To insert a key x we know, because of how the search algorithm works, that we must ultimately store x at location $A_i[j]$ where $i = h_{k,m}(x)$ and $j = h_{k_i,m_i}(x)$. However, several things can go wrong during the insertion of x:

1. The value of n increases by 1, so it may be that n now exceeds m. In this case we perform a global rebuilding operation and we are done.
2. We compute $i = h_{k,m}(x)$ and discover that $\sum_{i=0}^{m-1} {n_i}^2 > 3n/\alpha$. In this case, the hash function $h_{k,m}$ used at the top level is no longer any good since it is producing an overall hash table that is too large. In this case we perform a global rebuilding operation and we are done.
3. We compute $i = h_{k,m}(x)$ and discover that, since the value of n_i just increased by one, ${n_i}^2/\alpha > m_i$. In this case, the array A_i is too small to guarantee that we can quickly find a perfect hash function. To handle this, we copy the elements of A_i into a list L and allocate a new array A_i with the new size $m_i = 2{n_i}^2/\alpha$. We then find a new value k_i such that h_{k_i,m_i} is a perfect hash function for the elements of L and we are done.
4. The array location $A_i[j]$ is already occupied by some other element y. But in this case, we know that A_i is large enough to hold all the elements (otherwise we would already be done after Case 3), but the value k_i being used in the hash function h_{k_i,m_i} is the wrong one since it doesn't give a perfect hash function for S_i. Therefore we simply try new values for k_i until we find a find a value k_i that yields a perfect hash function and we are done.

If none of the preceding 4 cases occurs then we can simply place x at location $A_i[j]$ and we are done.

Handling Case 1 takes $O(n)$ expected time since it involves a global rebuild of the entire data structure. However, Case 1 only happens during one out of every $\Theta(n)$ insertions, so the amortized cost of all occurrences of Case 1 is only $O(1)$ per insertion.

Handling Case 2 also takes $O(n)$ expected time. The question is: How often does Case 2 occur? To answer this question, consider the *phase* that occurs between two consecutive occurrences of Case 1. During this phase, the data structure holds at most m distinct elements. Call this set of elements S. With probability at least $(1 - \alpha)$ the hash function $h_{k,m}$ selected at the beginning of the phase satisfies (9.2) so that Case 2 never occurs during the phase. Similarly, the probability that Case 2 occurs exactly once during the phase is at most $\alpha(1 - \alpha)$. In general, the probability that Case 2 occurs exactly i times during a phase is at most $\alpha^i(1 - \alpha)$. Thus, the expected cost of handling all occurrences of Case 2

during the entire phase is at most

$$\sum_{i=0}^{\infty} \alpha^i (1-\alpha) i \times O(n) = O(n) \ .$$

But since a phase involves $\Theta(n)$ insertions this means that the amortized expected cost of handling Case 2 is $O(1)$ per insertion.

Next we analyze the total cost of handling Case 3. Define a *subphase* as the period of time between two global rebuilding operations triggered either as a result of a deletion, Case 1 or Case 2. We will show that the total cost of handling all occurrences of Case 3 during a subphase is $O(n)$ and since a subphase takes $\Theta(n)$ time anyway this does not contribute to the cost of a subphase by more than a constant factor. When Case 3 occurs at the array A_i it takes $O(m_i)$ time. However, while handling Case 3, m_i increases by a constant factor, so the total cost of handling Case 3 for A_i is dominated by the value of m_i at the end of the subphase. But we maintain the invariant that $\sum_{i=0}^{m-1} m_i = O(n)$ during the entire subphase. Thus, handling all occurrences of Case 3 during a subphase only requires $O(n)$ time.

Finally, we consider the cost of handling Case 4. For a particular array A_i, consider the *subsubphase* between which two occurrences of Case 3 cause A_i to be rebuilt or a global rebuilding operation takes place. During this subsubphase the number of distinct elements that occupy A_i is at most $\alpha\sqrt{m_i}$. Therefore, with probability at least $1-\alpha$ any randomly chosen value of $k_i \in U$ is a perfect hash function for this set. Just as in the analysis of Case 2, this implies that the expected cost of handling all occurrences of Case 3 at A_i during a subsubphase is only $O(m_i)$. Since a subsubphase ends with rebuilding all of A_i or a global rebuilding, at a cost of $\Omega(m_i)$ all the occurrences of Case 4 during a subsubphase do not contribute to the expected cost of the subsubphase by more than a constant factor.

To summarize, we have shown that the expected cost of handling all occurrences of Case 4 is only a constant factor times the cost of handling all occurrences of Case 3. The cost of handling all occurrences of Case 3 is no more than a constant factor times the expected cost of all global rebuilds. The cost of handling all the global rebuilds that occur as a result of Case 2 is no more than a constant factor times the cost of handling all occurrences of global rebuilds that occur as a consequence of Case 1. And finally, the cost of all global rebuilds that occur as a result of Case 1 or of deletions is $O(n)$ for a sequence of n update operations. Therefore, the total expected cost of n update operation is $O(n)$.

9.3 Random Probing

Next we consider hash table implementations under the random probing assumption: Each element x stored in the hash table comes with a random sequence $x_0, x_1, x_2, \ldots$ where each of the x_i is independently and uniformly distributed in $\{1, \ldots, m\}$.[4] We begin with a discussion of the two basic paradigms: hashing with chaining and open addressing. Both these paradigms attempt to store the key x at array position $A[x_0]$. The difference between these two algorithms is their *collision resolution strategy*, i.e., what the algorithms do when a user inserts the key value x but array position $A[x_0]$ already contains some other key.

[4] A variant of the random probing assumption, referred to as the *uniform hashing* assumption, assumes that $x_0, \ldots, x_{m-1}$ is a random permutation of $0, \ldots, m-1$.

9.3.1 Hashing with Chaining

In *hashing with chaining*, a collision is resolved by allowing more than one element to live at each position in the table. Each entry in the array A is a pointer to the head of a linked list. To insert the value x, we simply append it to the list $A[x_0]$. To search for the element x, we perform a linear search in the list $A[x_0]$. To delete the element x, we search for x in the list $A[x_0]$ and splice it out.

It is clear that insertions take $O(1)$ time, even in the worst case. For searching and deletion, the running time is proportional to a constant plus the length of the list stored at $A[x_0]$. Notice that each of the at most n elements not equal to x is stored in $A[x_0]$ with probability $1/m$, so the expected length of $A[x_0]$ is either $\alpha = n/m$ (if x is not contained in the table) or $1 + (n-1)/m$ (if x is contained in the table). Thus, the expected cost of searching for or deleting an element is $O(1+\alpha)$.

The above analysis shows us that hashing with chaining supports the three set ADT operations in $O(1)$ expected time per operation, as long as the occupancy, α, is a constant. It is worth noting that this does not require that the value of α be less than 1.

If we would like more detailed information about the cost of searching, we might also ask about the *worst-case search time* defined as

$$W = \max\{\text{length of the list stored at } A[i] : 0 \leq i \leq m-1\} \ .$$

It is very easy to prove something quite strong about W using only the fact that the length of each list $A[i]$ is a binomial$(n, 1/m)$ random variable. Using Chernoff's bounds on the tail of the binomial distribution [13], this immediately implies that

$$\Pr\{\text{length of } A[i] \geq \alpha c \ln n\} \leq n^{-\Omega(c)} \ .$$

Combining this with Boole's inequality ($\Pr\{A \text{ or } B\} \leq \Pr\{A\} + \Pr\{B\}$) we obtain

$$\Pr\{W \geq \alpha c \ln n\} \leq n \times n^{-\Omega(c)} = n^{-\Omega(c)} \ .$$

Thus, with very high probability, the worst-case search time is logarithmic in n. This also implies that $\mathrm{E}[W] = O(\log n)$. The distribution of W has been carefully studied and it is known that, *with high probability*, i.e., with probability $1 - o(1)$, $W = (1+o(1)) \ln n / \ln \ln n$ [33, 38].[5] Gonnet has proven a more accurate result that $W = \Gamma^{-1}(n) - 3/2 + o(1)$ with high probability. Devroye [18] shows that similar results hold even when the distribution of x_0 is not uniform.

9.3.2 Hashing with Open Addressing

Hashing with open addressing differs from hashing with chaining in that each table position $A[i]$ is allowed to store only one value. When a collision occurs at table position i, one of the two elements involved in the collision must move on to the next element in its probe sequence. In order to implement this efficiently and correctly we require a method of marking elements as deleted. This method could be an auxiliary array that contains one bit for each element of A, but usually the same result can be achieved by using a special key value **del** that does not correspond to any valid key.

[5] Here, and throughout this chapter, if an asymptotic notation does not contain a variable then the variable that tends to infinity is implicitly n. Thus, for example, $o(1)$ is the set of non-negative functions of n that tend to 0 as $n \to \infty$.

To search for an element x in the hash table we look for x at positions $A[x_0]$, $A[x_1]$, $A[x_2]$, and so on until we either (1) find x, in which case we are done or (2) find an empty table position $A[x_i]$ that is not marked as deleted, in which case we can be sure that x is not stored in the table (otherwise it would be stored at position x_i). To delete an element x from the hash table we first search for x. If we find x at table location $A[x_i]$ we then simply mark $A[x_i]$ as deleted. To insert a value x into the hash table we examine table positions $A[x_0]$, $A[x_1]$, $A[x_2]$, and so on until we find a table position $A[x_i]$ that is either empty or marked as deleted and we store the value x in $A[x_i]$.

Consider the cost of inserting an element x using this method. Let i_x denote the smallest value i such that x_{i_x} is either empty or marked as deleted when we insert x. Thus, the cost of inserting x is a constant plus i_x. The probability that the table position x_0 is occupied is at most α so, with probability at least $1-\alpha$, $i_x = 0$. Using the same reasoning, the probability that we store x at position x_i is at most

$$\Pr\{i_x = i\} \leq \alpha^i(1-\alpha) \tag{9.3}$$

since the table locations $x_0, \ldots, x_{i-1}$ must be occupied, the table location x_i must not be occupied and the x_i are independent. Thus, the expected number of steps taken by the insertion algorithm is

$$\sum_{i=1}^{\infty} i \Pr\{i_x = i\} = (1-\alpha)\sum_{i=1}^{\infty} i\alpha^{i-1} = 1/(1-\alpha)$$

for any constant $0 < \alpha < 1$. The cost of searching for x and deleting x are both proportional to the cost of inserting x, so the expected cost of each of these operations is $O(1/(1-\alpha))$.[6]

We should compare this with the cost of hashing with chaining. In hashing with chaining,the occupancy α has very little effect on the cost of operations. Indeed, any constant α, even greater than 1 results in $O(1)$ time per operation. In contrast, open addressing is very dependent on the value of α. If we take $\alpha > 1$ then the expected cost of insertion using open addressing is infinite since the insertion algorithm never finds an empty table position. Of course, the advantage of hashing with chaining is that it does not require lists at each of the $A[i]$. Therefore, the overhead of list pointers is saved and this extra space can be used instead to maintain the invariant that the occupancy α is a constant strictly less than 1.

Next we consider the worst case search time of hashing with open addressing. That is, we study the value $W = \max\{i_x : x \text{ is stored in the table at location } i_x\}$. Using (9.3) and Boole's inequality it follows almost immediately that

$$\Pr\{W > c\log n\} \leq n^{-\Omega(c)}.$$

Thus, with very high probability, W, the worst case search time, is $O(\log n)$. Tighter bounds on W are known when the probe sequences $x_0, \ldots, x_{m-1}$ are random permutations of $0, \ldots, m-1$. In this case, Gonnet[29] shows that

$$\mathrm{E}[W] = \log_{1/\alpha} n - \log_{1/\alpha}(\log_{1/\alpha} n) + O(1).$$

[6]Note that the expected cost of searching for or deleting an element x is proportional to the value of α *at the time* x *was inserted.* If many deletions have taken place, this may be quite different than the current value of α.

Open addressing under the random probing assumption has many nice theoretical properties and is easy to analyze. Unfortunately, it is often criticized as being an unrealistic model because it requires a long random sequences $x_0, x_1, x_2, \ldots$ for each element x that is to be stored or searched for. Several variants of open addressing discussed in the next few sections try to overcome this problem by using only a few random values.

9.3.3 Linear Probing

Linear probing is a variant of open addressing that requires less randomness. To obtain the probe sequence $x_0, x_1, x_2, \ldots$ we start with a random element $x_0 \in \{0, \ldots, m-1\}$. The element x_i, $i > 0$ is given by $x_i = (i + x_0) \bmod m$. That is, one first tries to find x at location x_0 and if that fails then one looks at $(x_0 + 1) \bmod m$, $(x_0 + 2) \bmod m$ and so on.

The performance of linear probing is discussed by Knuth [37] who shows that the expected number of probes performed during an unsuccessful search is at most

$$(1 + 1/(1 - \alpha)^2)/2$$

and the expected number of probes performed during a successful search is at most

$$(1 + 1/(1 - \alpha))/2 \ .$$

This is not quite as good as for standard hashing with open addressing, especially in the unsuccessful case.

Linear probing suffers from the problem of *primary clustering*. If j consecutive array entries are occupied then a newly inserted element will have probability j/m of hashing to one of these entries. This results in $j + 1$ consecutive array entries being occupied and increases the probability (to $(j + 1)/m$) of another newly inserted element landing in this cluster. Thus, large clusters of consecutive elements have a tendency to grow larger.

9.3.4 Quadratic Probing

Quadratic probing is similar to linear probing; an element x determines its entire probe sequence based on a single random choice, x_0. Quadratic probing uses the probe sequence $x_0, (x_0 + k_1 + k_2) \bmod m, (x_0 + 2k_1 + 2^2 k_2) \bmod m, \ldots$. In general, the ith element in the probe sequence is $x_i = (x_0 + ik_1 + i^2 k_2) \bmod m$. Thus, the final location of an element depends quadratically on how many steps were required to insert it. This method seems to work much better in practice than linear probing, but requires a careful choice of m, k_1 and k_2 so that the probe sequence contains every element of $\{0, \ldots, m-1\}$.

The improved performance of quadratic probing is due to the fact that if there are two elements x and y such that $x_i = y_j$ then it is not necessarily true (as it is with linear probing) that $x_{i+1} = y_{j+1}$. However, if $x_0 = y_0$ then x and y will have exactly the same probe sequence. This lesser phenomenon is called *secondary clustering*. Note that this secondary clustering phenomenon implies that neither linear nor quadratic probing can hope to perform any better than hashing with chaining. This is because all the elements that have the same initial hash x_0 are contained in an implicit chain. In the case of linear probing, this chain is defined by the sequence $x_0, x_0 + 1, x_0 + 2, \ldots$ while for quadratic probing it is defined by the sequence $x_0, x_0 + k_1 + k_2, x_0 + 2k_1 + 4k_2, \ldots$

9.3.5 Double Hashing

Double hashing is another method of open addressing that uses two hash values x_0 and x_1. Here x_0 is in the set $\{0, \ldots, m-1\}$ and x_1 is in the subset of $\{1, \ldots, m-1\}$ that is

relatively prime to m. With double hashing, the probe sequence for element x becomes $x_0, (x_0 + x_1) \bmod m, (x_0 + 2x_1) \bmod m, \ldots$. In general, $x_i = (x_0 + ix_1) \bmod m$, for $i > 0$. The expected number of probes required by double hashing seems difficult to determine exactly. Guibas has proven that, asymptotically, and for occupancy $\alpha \leq .31$, the performance of double hashing is asymptotically equivalent to that of uniform hashing. Empirically, the performance of double hashing matches that of open addressing with random probing regardless of the occupancy α [37].

9.3.6 Brent's Method

Brent's method [5] is a heuristic that attempts to minimize the average time for a successful search in a hash table with open addressing. Although originally described in the context of double hashing (Section 9.3.5) Brent's method applies to any open addressing scheme. The *age* of an element x stored in an open addressing hash table is the minimum value i such that x is stored at $A[x_i]$. In other words, the age is one less than the number of locations we will probe when searching for x.

Brent's method attempts to minimize the total age of all elements in the hash table. To insert the element x we proceed as follows: We find the smallest value i such that $A[x_i]$ is empty; this is where standard open-addressing would insert x. Consider the element y stored at location $A[x_{i-2}]$. This element is stored there because $y_j = x_{i-2}$, for some $j \geq 0$. We check if the array location $A[y_{j+1}]$ is empty and, if so, we move y to location $A[y_{j+1}]$ and store x at location $A[x_{i-2}]$. Note that, compared to standard open addressing, this decreases the total age by 1. In general, Brent's method checks, for each $2 \leq k \leq i$ the array entry $A[x_{i-k}]$ to see if the element y stored there can be moved to any of $A[y_{j+1}], A[y_{j+2}], \ldots, A[y_{j+k-1}]$ to make room for x. If so, this represents a decrease in the total age of all elements in the table and is performed.

Although Brent's method seems to work well in practice, it is difficult to analyze theoretically. Some theoretical analysis of Brent's method applied to double hashing is given by Gonnet and Munro [31]. Lyon [44], Munro and Celis [49] and Poblete [52] describe some variants of Brent's method.

9.3.7 Multiple-Choice Hashing

It is worth stepping back at this point and revisiting the comparison between hash tables and binary search trees. For balanced binary search trees, the average cost of searching for an element is $O(\log n)$. Indeed, it easy to see that for at least $n/2$ of the elements, the cost of searching for those elements is $\Omega(\log n)$. In comparison, for both the random probing schemes discussed so far, the expected cost of search for an element is $O(1)$. However, there are a handful of elements whose search cost is $\Theta(\log n / \log \log n)$ or $\Theta(\log n)$ depending on whether hashing with chaining or open addressing is used, respectively. Thus there is an inversion: Most operations on a binary search tree cost $\Theta(\log n)$ but a few elements (close to the root) can be accessed in $O(1)$ time. Most operations on a hash table take $O(1)$ time but a few elements (in long chains or with long probe sequences) require $\Theta(\log n / \log \log n)$ or $\Theta(\log n)$ time to access. In the next few sections we consider variations on hashing with chaining and open addressing that attempt to reduce the worst-case search time W.

Multiple-choice hashing is hashing with chaining in which, during insertion, the element x has the choice of $d \geq 2$ different lists in which it can be stored. In particular, when we insert x we look at the lengths of the lists pointed to by $A[x_0], \ldots, A[x_{d-1}]$ and append x to $A[x_i]$, $0 \leq i < d$ such that the length of the list pointed to by $A[x_i]$ is minimum. When searching for x, we search for x in each of the lists $A[x_0], \ldots, A[x_{d-1}]$ *in parallel*. That is, we

look at the first elements of each list, then the second elements of each list, and so on until we find x. As before, to delete x we first search for it and then delete it from whichever list we find it in.

It is easy to see that the expected cost of searching for an element x is $O(d)$ since the expected length of each the d lists is $O(1)$. More interestingly, the worst case search time is bounded by $O(dW)$ where W is the length of the longest list. Azar et al. [3] show that

$$\mathrm{E}[W] = \frac{\ln \ln n}{\ln d} + O(1) \ . \tag{9.4}$$

Thus, the expected worst case search time for multiple-choice hashing is $O(\log \log n)$ for any constant $d \geq 2$.

9.3.8 Asymmetric Hashing

Asymmetric hashing is a variant of multiple-choice hashing in which the hash table is split into d blocks, each of size n/d. (Assume, for simplicity, that n is a multiple of d.) The probe value x_i, $0 \leq i < d$ is drawn uniformly from $\{in/d, \ldots, (i+1)n/d - 1\}$. As with multiple-choice hashing, to insert x the algorithm examines the lengths of the lists $A[x_0], A[x_1], \ldots, A[x_{d-1}]$ and appends x to the shortest of these lists. In the case of ties, it appends x to the list with smallest index. Searching and deletion are done exactly as in multiple-choice hashing.

Vöcking [64] shows that, with asymmetric hashing the expected length of the longest list is

$$\mathrm{E}[W] \leq \frac{\ln \ln n}{d \ln \phi_d} + O(1) \ .$$

The function ϕ_d is a generalization of the *golden ratio*, so that $\phi_2 = (1 + \sqrt{5})/2$. Note that this improves significantly on standard multiple-choice hashing (9.4) for larger values of d.

9.3.9 LCFS Hashing

LCFS hashing is a form of open addressing that changes the collision resolution strategy.[7] Reviewing the algorithm for hashing with open addressing reveals that when two elements collide, priority is given to the first element inserted into the hash table and subsequent elements must move on. Thus, hashing with open addressing could also be referred to as *FCFS (first-come first-served) hashing*.

With LCFS (last-come first-served) hashing, collision resolution is done in exactly the opposite way. When we insert an element x, we always place it at location x_0. If position x_0 is already occupied by some element y because $y_j = x_0$ then we place y at location y_{j+1}, possibly displacing some element z, and so on.

Poblete and Munro [53] show that, after inserting n elements into an initially empty table, the expected worst case search time is bounded above by

$$\mathrm{E}[W] \leq 1 + \Gamma^{-1}(\alpha n) \left(1 + \frac{\ln \ln(1/(1-\alpha))}{\ln \Gamma^{-1}(\alpha n)} + O\left(\frac{1}{\ln^2 \Gamma^{-1}(\alpha n)} \right) \right) \ ,$$

[7] Amble and Knuth [1] were the first to suggest that, with open addressing, any collision resolution strategy could be used.

where Γ is the gamma function and

$$\Gamma^{-1}(\alpha n) = \frac{\ln n}{\ln \ln n}\left(1 + \frac{\ln \ln \ln n}{\ln \ln n} + O\left(\frac{1}{\ln \ln n}\right)\right) \quad .$$

Historically, LCFS hashing is the first version of open addressing that was shown to have an expected worst-case search time that is $o(\log n)$.

9.3.10 Robin-Hood Hashing

Robin-Hood hashing [9, 10, 61] is a form of open addressing that attempts to equalize the search times of elements by using a fairer collision resolution strategy. During insertion, if we are trying to place element x at position x_i and there is already an element y stored at position $y_j = x_i$ then the "younger" of the two elements must move on. More precisely, if $i \leq j$ then we will try to insert x at position x_{i+1}, x_{i+2} and so on. Otherwise, we will store x at position x_i and try to to insert y at positions y_{j+1}, y_{j+2} and so on.

Devroye et al. [20] show that, after performing n insertions on an initially empty table of size $m = \alpha n$ using the Robin-Hood insertion algorithm, the worst case search time has expected value

$$\mathrm{E}[W] = \Theta(\log \log n)$$

and this bound is tight. Thus, Robin-Hood hashing is a form of open addressing that has doubly-logarithmic worst-case search time. This makes it competitive with the multiple-choice hashing method of Section 9.3.7.

9.3.11 Cuckoo Hashing

Cuckoo hashing [50] is a form of multiple choice hashing in which each element x lives in one of two tables A or B, each of size $m = n/\alpha$. The element x will either be stored at location $A[x_A]$ or $B[x_B]$. There are no other options. This makes searching for x an $O(1)$ time operation since we need only check two array locations.

The insertion algorithm for cuckoo hashing proceeds as follows:[8] Store x at location $A[x_A]$. If $A[x_A]$ was previously occupied by some element y then store y at location $B[y_B]$. If $B[y_B]$ was previously occupied by some element z then store z at location $A[z_A]$, and so on. This process ends when we place an element into a previously empty table slot or when it has gone on for more than $c \log n$ steps. In the former case, the insertion of x completes successfully. In the latter case the insertion is considered a failure, and the entire hash table is reconstructed from scratch using a new probe sequence for each element in the table. That is, if this reconstruction process has happened i times then the two hash values we use for an element x are $x_A = x_{2i}$ and $x_B = x_{2i+1}$.

Pagh and Rodler [50] (see also Devroye and Morin [19]) show that, during the insertion of n elements, the probability of requiring a reconstruction is $O(1/n)$. This, combined with the fact that the expected insertion time is $O(1)$ shows that the expected cost of n insertions in a Cuckoo hashing table is $O(n)$. Thus, Cuckoo hashing offers a somewhat simpler alternative to the dynamic perfect hashing algorithms of Section 9.2.5.

[8] The algorithm takes its name from the large but lazy cuckoo bird which, rather than building its own nest, steals the nest of another bird forcing the other bird to move on.

9.4 Historical Notes

In this section we present some of the history of hash tables. The idea of hashing seems to have been discovered simultaneously by two groups of researchers. Knuth [37] cites an internal IBM memorandum in January 1953 by H. P. Luhn that suggested the use of hashing with chaining. Building on Luhn's work, A. D. Linh suggested a method of open addressing that assigns the probe sequence $x_0, \lfloor x_0/10 \rfloor, \lfloor x_0/100 \rfloor, \lfloor x_0/1000 \rfloor, \ldots$ to the element x.

At approximately the same time, another group of researchers at IBM: G. M. Amdahl, E. M. Boehme, N. Rochester and A. L. Samuel implemented hashing in an assembly program for the IBM 701 computer. Amdahl is credited with the idea of open addressing with linear probing.

The first published work on hash tables was by A. I. Dumey [24], who described hashing with chaining and discussed the idea of using remainder modulo a prime as a hash function. Ershov [25], working in Russia and independently of Amdahl, described open addressing with linear probing.

Peterson [51] wrote the first major article discussing the problem of searching in large files and coined the term "open addressing." Buchholz [7] also gave a survey of the searching problem with a very good discussion of hashing techniques at the time. Theoretical analyses of linear probing were first presented by Konheim and Weiss [39] and Podderjugin. Another, very influential, survey of hashing was given by Morris [47]. Morris' survey is the first published use of the word "hashing" although it was already in common use by practitioners at that time.

9.5 Other Developments

The study of hash tables has a long history and many researchers have proposed methods of implementing hash tables. Because of this, the current chapter is necessarily incomplete. (At the time of writing, the hash.bib bibliography on hashing contains over 800 entries.) We have summarized only a handful of the major results on hash tables in internal memory. In this section we provide a few references to the literature for some of the other results. For more information on hashing, Knuth [37], Vitter and Flajolet [63], Vitter and Chen [62], and Gonnet and Baeza-Yates [30] are useful references.

Brent's method (Section 9.3.6) is a collision resolution strategy for open addressing that reduces the expected search time for a successful search in a hash table with open addressing. Several other methods exist that either reduce the expected or worst-case search time. These include *binary tree hashing* [31, 45], *optimal hashing* [31, 54, 55], Robin-Hood hashing (Section 9.3.10), and *min-max hashing* [9, 29]. One interesting method, due to Celis [9], applies to any open addressing scheme. The idea is to study the distribution of the ages of elements in the hash table, i.e., the distribution give by

$$D_i = \Pr\{x \text{ is stored at position } x_i\}$$

and start searching for x at the locations at which we are most likely to find it, rather than searching the table positions $x_0, x_1, x_2 \ldots$ in order.

Perfect hash functions seem to have been first studied by Sprugnoli [58] who gave some heuristic number theoretic constructions of minimal perfect hash functions for small data sets. Sprugnoli is responsible for the terms "perfect hash function" and "minimal perfect hash function." A number of other researchers have presented algorithms for discovering minimal and near-minimal perfect hash functions. Examples include Anderson and Anderson [2], Cichelli [14, 15], Chang [11, 12], Gori and Soda [32], and Sager [56]. Berman et al. [4]

and Körner and Marton [40] discuss the theoretical limitations of perfect hash functions. A comprehensive, and recent, survey of perfect hashing and minimal perfect hashing is given by Czech et al. [17].

Tarjan and Yao [60] describe a set ADT implementation that gives $O(\log u/\log n)$ worst-case access time. It is obtained by combining a trie (Chapter 28) of degree n with a compression scheme for arrays of size n^2 that contain only n non-zero elements. (The trie has $O(n)$ nodes each of which has n pointers to children, but there are only a total of $O(n)$ children.) Although their result is superseded by the results of Fredman et al. [28] discussed in Section 9.2.4, they are the first theoretical results on worst-case search time for hash tables.

Dynamic perfect hashing (Section 9.2.5) and cuckoo hashing (Section 9.3.11) are methods of achieving $O(1)$ worst case search time in a dynamic setting. Several other methods have been proposed [6, 21, 22].

Yao [65] studies the *membership problem*. Given a set $S \subseteq U$, devise a data structure that can determine for any $x \in U$ whether x is contained in S. Yao shows how, under various conditions, this problem can be solved using a very small number of memory accesses per query. However, Yao's algorithms sometimes derive the fact that an element x is in S without actually finding x. Thus, they don't solve the set ADT problem discussed at the beginning of this chapter since they can not recover a pointer to x.

The "power of two random choices," as used in multiple-choice hashing, (Section 9.3.7) has many applications in computer science. Karp, Luby and Meyer auf der Heide [34, 35] were the first to use this paradigm for simulating PRAM computers on computers with fewer processors. The book chapter by Mitzenmacher et al. [46] surveys results and applications of this technique.

A number of table implementations have been proposed that are suitable for managing hash tables in external memory. Here, the goal is to reduce the number of disk blocks that must be accessed during an operation, where a disk block can typically hold a large number of elements. These schemes include *linear hashing* [43], *dynamic hashing* [41], *virtual hashing* [42], *extendible hashing* [26], *cascade hashing* [36], and *spiral storage* [48]. In terms of hashing, the main difference between internal memory and external memory is that, in internal memory, an array is allocated at a specific size and this can not be changed later. In contrast, an external memory file may be appended to or be truncated to increase or decrease its size, respectively. Thus, hash table implementations for external memory can avoid the periodic global rebuilding operations used in internal memory hash table implementations.

Acknowledgment

The author is supported by a grant from the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] O. Amble and D. E. Knuth. Ordered hash tables. *The Computer Journal*, 17(2):135–142, 1974.

[2] M. R. Anderson and M. G. Anderson. Comments on perfect hashing functions: A single probe retrieving method for static sets. *Communications of the ACM*, 22(2):104, 1979.

[3] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. *SIAM Journal on Computing*, 29(1):180–200, 1999.

[4] F. Berman, M. E. Bock, E. Dittert, M. J. O'Donnell, and D. Plank. Collections of functions for perfect hashing. *SIAM Journal on Computing*, 15(2):604–618, 1986.

[5] R. P. Brent. Reducing the storage time of scatter storage techniques. *Communications of the ACM*, 16(2):105–109, 1973.

[6] A. Brodnik and J. I. Munro. Membership in constant time and almost minimum space. *SIAM Journal on Computing*, 28:1627–1640, 1999.

[7] W. Buchholz. File organization and addressing. *IBM Systems Journal*, 2(1):86–111, 1963.

[8] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[9] P. Celis. Robin Hood hashing. Technical Report CS-86-14, Computer Science Department, University of Waterloo, 1986.

[10] P. Celis, P.-Å. Larson, and J. I. Munro. Robin Hood hashing. In *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science (FOCS'85)*, pages 281–288. IEEE Press, 1985.

[11] C. C. Chang. An ordered minimal perfect hashing scheme based upon Euler's theorem. *Information Sciences*, 32(3):165–172, 1984.

[12] C. C. Chang. The study of an ordered minimal perfect hashing scheme. *Communications of the ACM*, 27(4):384–387, 1984.

[13] H. Chernoff. A measure of the asymptotic efficient of tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, 23:493–507, 1952.

[14] R. J. Cichelli. Minimal perfect hash functions made simple. *Communications of the ACM*, 23(1):17–19, 1980.

[15] R. J. Cichelli. On Cichelli's minimal perfect hash functions method. *Communications of the ACM*, 23(12):728–729, 1980.

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, Massachussetts, 2nd edition, 2001.

[17] Z. J. Czech, G. Havas, and B. S. Majewski. Perfect hashing. *Theoretical Computer Science*, 182(1-2):1–143, 1997.

[18] L. Devroye. The expected length of the longest probe sequence when the distribution is not uniform. *Journal of Algorithms*, 6:1–9, 1985.

[19] L. Devroye and P. Morin. Cuckoo hashing: Further analysis. *Information Processing Letters*, 86(4):215–219, 2002.

[20] L. Devroye, P. Morin, and A. Viola. On worst case Robin-Hood hashing. *SIAM Journal on Computing*. To appear.

[21] M. Dietzfelbinger and F. Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proceedings of the 17th International Colloquium on Automata, Languages, and Programming (ICALP'90)*, pages 6–19, 1990.

[22] M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger. Polynomial hash functions are reliable. In *Proceedings of the 19th International Colloquium on Automata, Languages, and Programming (ICALP'92)*, pages 235–246, 1992.

[23] M. Dietzfelbinger, A. R. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.

[24] A. I. Dumey. Indexing for rapid random access memory systems. *Computers and Automation*, 5(12):6–9, 1956.

[25] A. P. Ershov. On programming of arithmetic operations. *Doklady Akademii Nauk SSSR*, 118(3):427–430, 1958.

[26] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing — a fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, 1979.

[27] W. Feller. *An Introduction to Probability Theory and its Applications*. John Wiley & Sons, New York, 1968.

[28] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.

[29] G. H. Gonnet. Expected length of the longest probe sequence in hash code searching. *Journal of the ACM*, pages 289–304, 1981.

[30] G. H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures: in Pascal and C*. Addison-Wesley, Reading, MA, USA, 2nd edition, 1991.

[31] G. H. Gonnet and J. I. Munro. Efficient ordering of hash tables. *SIAM Journal on Computing*, 8(3):463–478, 1979.

[32] M. Gori and G. Soda. An algebraic approach to Cichelli's perfect hashing. *Bit*, 29(1):2–13, 1989.

[33] N. L. Johnson and S. Kotz. *Urn Models and Their Applications*. John Wiley & Sons, New York, 1977.

[34] R. Karp, M. Luby, and F. Meyer auf der Heide. Efficient PRAM simulation on a distributed memory machine. Technical Report TR-93-040, International Computer Science Institute, Berkeley, CA, USA, 1993.

[35] R. M. Karp, M. Luby, and F. Meyer auf der Heide. Efficient PRAM simulation on a distributed memory machine. In *Proceedings of the 24th ACM Symposium on the Theory of Computing (STOC'92)*, pages 318–326. ACM Press, 1992.

[36] P. Kjellberg and T. U. Zahle. Cascade hashing. In *Proceedings of the 10th International Conference on Very Large Data Bases (VLDB'80)*, pages 481–492. Morgan Kaufmann, 1984.

[37] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 2nd edition, 1997.

[38] V. F. Kolchin, B. A. Sevastyanov, and V. P. Chistyakov. *Random Allocations*. John Wiley & Sons, New York, 1978.

[39] A. G. Konheim and B. Weiss. An occupancy discipline and its applications. *SIAM Journal of Applied Mathematics*, 14:1266–1274, 1966.

[40] J. Körner and K. Marton. New bounds for perfect hashing via information theory. *European Journal of Combinatorics*, 9(6):523–530, 1988.

[41] P.-Å. Larson. Dynamic hashing. *Bit*, 18(2):184–201, 1978.

[42] W. Litwin. Virtual hashing: A dynamically changing hashing. In *Proceedings of the 4th International Conference on Very Large Data Bases (VLDB'80)*, pages 517–523. IEEE Computer Society, 1978.

[43] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Proceedings of the 6th International Conference on Very Large Data Bases (VLDB'80)*, pages 212–223. IEEE Computer Society, 1980.

[44] G. E. Lyon. Packed scatter tables. *Communications of the ACM*, 21(10):857–865, 1978.

[45] E. G. Mallach. Scatter storage techniques: A unifying viewpoint and a method for reducing retrieval times. *The Computer Journal*, 20(2):137–140, 1977.

[46] M. Mitzenmacher, A. W. Richa, and R. Sitaraman. The power of two random choices: A survey of techniques and results. In P. Pardalos, S. Rajasekaran, and J. Rolim, editors, *Handbook of Randomized Computing*, volume 1, chapter 9. Kluwer, 2001.

[47] R. Morris. Scatter storage techniques. *Communications of the ACM*, 11(1):38–44, 1968.

[48] J. K. Mullin. Spiral storage: Efficient dynamic hashing with constant performance. *The Computer Journal*, 28(3):330–334, 1985.

[49] J. I. Munro and P. Celis. Techniques for collision resolution in hash tables with open addressing. In *Proceedings of 1986 Fall Joint Computer Conference*, pages 601–610. ACM Press, 1999.

[50] R. Pagh and F. F. Rodler. Cuckoo hashing. In *Proceedings of the 9th Annual European Symposium on Algorithms (ESA 2001)*, volume 2161 of *Lecture Notes in Computer Science*, pages 121–133. Springer-Verlag, 2001.

[51] W. W. Peterson. Addressing for random-access storage. *IBM Journal of Research and Development*, 1(2):130–146, 1957.

[52] P. V. Poblete. Studies on hash coding with open addressing. M. Math Essay, University of Waterloo, 1977.

[53] P. V. Poblete and J. Ian Munro. Last-come-first-served hashing. *Journal of Algorithms*, 10:228–248, 1989.

[54] G. Poonan. Optimal placement of entries in hash tables. In *ACM Computer Science Conference (Abstract Only)*, volume 25, 1976. (Also DEC Internal Tech. Rept. LRD-1, Digital Equipment Corp. Maynard Mass).

[55] R. L. Rivest. Optimal arrangement of keys in a hash table. *Journal of the ACM*, 25(2):200–209, 1978.

[56] T. J. Sager. A polynomial time generator for minimal perfect hash functions. *Communications of the ACM*, 28(5):523–532, 1985.

[57] V. T. Sós. On the theory of diophantine approximations. i. *Acta Mathematica Budapest*, 8:461–471, 1957.

[58] R. Sprugnoli. Perfect hashing functions: A single probe retrieving method for static sets. *Communications of the ACM*, 20(11):841–850, 1977.

[59] S. Świerczkowski. On successive settings of an arc on the circumference of a circle. *Fundamenta Mathematica*, 46:187–189, 1958.

[60] R. E. Tarjan and A. C.-C. Yao. Storing a sparse table. *Communications of the ACM*, 22(11):606–611, 1979.

[61] A. Viola and P. V. Poblete. Analysis of linear probing hashing with buckets. *Algorithmica*, 21:37–71, 1998.

[62] J. S. Vitter and W.-C. Chen. *The Design and Analysis of Coalesced Hashing*. Oxford University Press, Oxford, UK, 1987.

[63] J. S. Vitter and P. Flajolet. Analysis of algorithms and data structures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, chapter 9, pages 431–524. North Holland, 1990.

[64] B. Vöcking. How asymmetry helps load balancing. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science (FOCS'99)*, pages 131–140. IEEE Press, 1999.

[65] A. C.-C. Yao. Should tables be sorted? *Journal of the ACM*, 28(3):615–628, 1981.

10

Balanced Binary Search Trees

Arne Andersson
Uppsala University

Rolf Fagerberg
University of Southern Denmark

Kim S. Larsen
University of Southern Denmark

10.1 Introduction

Balanced binary search trees are among the most important data structures in Computer Science. This is because they are efficient, versatile, and extensible in many ways. They are used as a black-box in numerous algorithms and even other data structures.

The main virtue of balanced binary search trees is their ability to maintain a dynamic set in sorted order, while supporting a large range of operations in time logarithmic in the size of the set. The operations include search, insertion, deletion, predecessor/successor search, range search, rank search, batch update, split, meld, and merge. These operations are described in more detail in Section 10.2 below.

Data structures supporting the operations search, insertion, deletion, and predecessor (and/or successor) search are often denoted *ordered dictionaries*. In the comparison based model, the logarithmic performance of balanced binary search trees is optimal for ordered dictionaries, whereas in the RAM model, faster operations are possible [13, 18]. If one considers *unordered dictionaries*, i.e., only the operations search, insertion, and deletion, expected constant time is possible by hashing.

10.2 Basic Definitions

10.2.1 Trees

There are many ways to define trees. In this section, we define a tree as a hierarchical organization of a collection of nodes. For alternatives to our exposition, see the chapter on trees.

A *tree* can be empty. If it is not empty, it consists of one node, which is referred to as the *root* of the tree, and a collection of trees, referred to as *subtrees*. Thus, a tree consists of many smaller trees, each with their own root. We use r to denote the single node which is the root of the entire tree.

We only consider *finite* trees, i.e., every collection of subtrees is finite, and there are no infinite chains of nonempty subtrees. Furthermore, we only consider *ordered* trees, meaning that the collection of subtrees of a node is an ordered sequence rather than just a set. If every nonempty tree has exactly two subtrees, then the tree is called *binary*. In this case, we refer to the two subtrees as the *left* and *right* subtrees.

We use u, v, w, etc. to denote nodes and T to denote trees, applying apostrophes, index, etc. to increase the name space. For a node u, we use $u.l$ and $u.r$ to denote the left and right subtree, respectively, of the tree rooted by u. However, when no confusion can occur, we do not necessarily distinguish between nodes and subtrees. Thus, by the subtree v, we mean the subtree rooted at the node v and by T we mean the entire tree or the root of the tree.

We use the standard genealogical terminology to denote nodes in the vicinity of a designated node. Thus, if u is the root of a tree and v is the root of a subtree of u, then v is referred to as a *child* of u. By analogy, this defines *grandchildren*, *parent*, *grandparent*, and *sibling*.

The set of nodes *belonging* to a nonempty tree is its root, along with all the nodes belonging to its subtrees. For an empty tree, this set is of course empty. If a node v belongs to the subtree of u, then v is a *descendant* of u, and u is an *ancestor* of v. An ancestor or descendant v of a node u is *proper* if $u \neq v$.

Quite often, it is convenient to refer to empty subtrees as real nodes, in which case they are referred to as *external* nodes (or leaves). The remaining nodes are then referred to as *internal* nodes. It is easy to prove by induction that the number of external nodes is always one larger than the number of internal nodes.

The number of nodes belonging to a tree is referred to as its *size* (or its *weight*). In some applications, we define the size of the tree to be the number of internal nodes in the tree, but more often it is convenient to define the size of the tree to be the number of external nodes. We use n to denote the size of the tree rooted by r, and $|u|$ to denote the size of the subtree rooted by u.

A *path* in a tree is a sequence of nodes $u_1, u_2, \ldots, u_k$, $k \geq 1$, such that for $i \in \{1, \ldots, k-1\}$, u_{i+1} is a child of u_i. Note that the length of such a path is $k-1$. The *depth* of a node u in the tree T is the length of the path from the root of T to u, and the *height* of a tree T is the maximal depth of any external node.

10.2.2 Binary Trees as Dictionaries

When trees are used to implement the abstract data type *dictionary*, nodes have associated values. A dictionary basically organizes a set of *keys*, which must be elements drawn from a total ordering, and must usually supply at least the operations search, insertion, and deletion. There may be additional information associated with each key, but this does not

lead to any conceptual complications, so here we simply focus on the keys.

When a tree is used as a dictionary, each node stores one key, and we impose the following ordering invariant (the *in-order* invariant): for each node u in the tree, every key in $u.l$ is strictly smaller than $u.k$, and every key in $u.r$ is strictly larger than $u.k$. A tree organized according to this invariant is referred to as a binary *search* tree.

An important implication of this ordering invariant is that a sorted list of all the keys in the tree can be produced in linear time using an *in-order traversal* defined recursively as follows. On an empty tree, do nothing. Otherwise, recurs on the left subtree, report the root key, and then recurs on the right subtree.

Many different operations can be supported by binary search tree implementations. Here, we discuss the most common. Using the ordering invariant, we can devise a searching procedure of asymptotic time complexity proportional to the height of the tree. Since searching turns out to be at the heart of most of the operations of interest to us, unless we stipulate otherwise, all the operations in the following inherit the same complexity.

Simple Searching

To *search* for x in a tree rooted by u, we first compare x to $u.k$. If they are equal, a positive response is given. Otherwise, if x is smaller than $u.k$, we search recursively in $u.l$, and if x is larger, we search in $u.r$. If we arrive at an empty tree, a negative response is given. In this description, we have used *ternary* comparisons, in that our decisions regarding how to proceed depend on whether the search key is less than, equal to, or greater than the root key. For implementation purposes, it is possible to use the more efficient *binary* comparisons [12].

A characteristic feature of search trees is that when a searching fails, a nearest neighbor can be provided efficiently. Dictionaries supporting predecessor/successor queries are referred to as *ordered*. This is in contrast to hashing (described in a chapter of their own) which represents a class of unordered dictionaries. A *predecessor* search for x must return the largest key less than or equal to x. This operation as well as the similar *successor* search are simple generalizations of the search strategy outlined above. The case where x is found on the way is simple, so assume that x is not in the tree. Then the crucial observation is that if the last node encountered during the search is smaller than x, then this node is the predecessor. Otherwise, the predecessor key is the largest key in the left subtree of the last node on the search path containing a key smaller than x. A successor search is similar.

Simple Updates

An *insertion* takes a tree T and a key x not belonging to T as arguments and adds a node containing x and two empty subtrees to T. The node replaces the empty subtree in T where the search for x terminates.

A *deletion* takes a tree T and a key x belonging to T as arguments and removes the node u containing x from the tree. If u's children are empty trees, u is simply replaced by an empty tree. If u has exactly one child which is an internal node, then this child is replacing u. Finally, if u has two internal nodes as children, u's predecessor node v is used. First, the key in u is overwritten by the key of v, after which v is deleted. Note that because of the choice of v, the ordering invariant is not violated. Note also that v has at most one child which is an internal node, so one of the simpler replacing strategies described above can be used to remove v.

More Searching Procedures

A *range* search takes a tree T and two key values $k_1 \leq k_2$ as arguments and returns all keys x for which $k_1 \leq x \leq k_2$. A range search can be viewed as an in-order traversal, where we do not recurs down the left subtree and do not report the root key if k_1 should be in the right subtree; similarly, we do not recurs down the right subtree and do not report the root key if k_2 should be in the left subtree. The complexity is proportional to the height of the tree plus the size of the output.

A useful technique for providing more complex operations efficiently is to equip the nodes in the tree with additional information which can be exploited in more advanced searching, and which can also be maintained efficiently. A *rank* search takes a tree T and an integer d between one and n as arguments, and returns the dth smallest key in T. In order to provide this functionality efficiently, we store in each node the size of the subtree in which it is the root. Using this information during a search down the tree, we can at each node determine in which subtree the node must be located and we can appropriately adjust the rank that we search for recursively. If the only modifications made to the tree are small local changes, this extra information can be kept up-to-date efficiently, since it can always be recomputed from the information in the children.

Operations Involving More Trees

The operation *split* takes a key value x and tree T as arguments and returns two trees; one containing all keys from T less than or equal to x and one with the remaining keys. The operations is destructive, meaning that the argument tree T will not be available after the operation. The operation *meld* takes two trees as arguments, where all keys in one tree are smaller than all keys in the other, and combines the trees into one containing all the keys. This operation is also destructive. Finally, *merge* combines the keys from two argument trees, with no restrictions on keys, into one. Also this operation is destructive.

10.2.3 Implementation of Binary Search Trees

In our discussion of time and space complexities, we assume that some standard implementation of trees are used. Thus, in analogy with the recursive definition, we assume that a tree is represented by information associated with its root, primarily the key, along with pointers (references) to its left and right subtrees, and that this information can be accessed in constant time.

In some situations, we may assume that additional pointers are present, such as *parent-pointers*, giving a reference from a node to its parent. We also sometimes use *level-pointers*. A *level* consists of all nodes of the same depth, and a level-pointer to the right from a node with key k points to the node at the same level with the smallest key larger than k. Similar for level-pointers to the left.

10.3 Generic Discussion of Balancing

As seen in Section 10.2, the worst case complexity of almost all operations on a binary search tree is proportional to its height, making the height its most important single characteristic.

Since a binary tree of height h contains at most $2^h - 1$ nodes, a binary tree of n nodes has a height of at least $\lceil \log(n+1) \rceil$. For static trees, this lower bound is achieved by a tree where all but one level is completely filled. Building such a tree can be done in linear time (assuming that the sorted order of the keys is known), as discussed in Section 10.5 below. In the dynamic case, however, insertions and deletions may produce a very unbalanced

tree—for instance, inserting elements in sorted order will produce a tree of height linear in the number of elements.

The solution is to rearrange the tree after an insertion or deletion of an element, if the operation has made the tree unbalanced. For this, one needs a *definition of balance* and a *rebalancing algorithm* describing the rearrangement leading to balance after updates. The combined balance definition and rebalancing algorithm we denote a *rebalancing scheme*. In this section, we discuss rebalancing schemes at a generic level.

The trivial rebalancing scheme consists of defining a balanced tree as one having the optimal height $\lceil \log(n+1) \rceil$, and letting the rebalancing algorithm be the rebuilding of the entire tree after each update. This costs linear time per update, which is exponentially larger than the search time of the tree. It is one of the basic results of Computer Science, first proved by Adel'son-Vel'skiĭ and Landis in 1962 [1], that logarithmic update cost can be achieved simultaneously with logarithmic search cost in binary search trees.

Since the appearance of [1], many other rebalancing schemes have been proposed. Almost all reproduce the result of [1] in the sense that they, too, guarantee a height of $c \cdot \log(n)$ for some constant $c > 1$, while handling updates in $O(\log n)$ time. The schemes can be grouped according to the ideas used for definition of balance, the ideas used for rebalancing, and the exact complexity results achieved.

10.3.1 Balance Definitions

The balance definition is a structural constraint on the tree ensuring logarithmic height. Many schemes can viewed as belonging to one of the following three categories: schemes with a constraint based on the *heights of subtrees*, schemes with a constraint based on the *sizes of subtrees*, and schemes which can be seen as *binarizations of multi-way search tree schemes* and which have a constraint inherited from these. The next section will give examples of each.

For most schemes, balance information is stored in the nodes of the tree in the form of single bits or numbers. The structural constraint is often expressed as an invariant on this information, and the task of the rebalancing algorithm is to reestablish this invariant after an update.

10.3.2 Rebalancing Algorithms

The rebalancing algorithm restores the structural constraint of the scheme if it is violated by an update. It uses the balance information stored in the nodes to guide its actions.

The general form of the algorithm is the same in almost all rebalancing schemes—balance violations are removed by working towards the root along the search path from the leaf where the update took place. When removing a violation at one node, another may be introduced at its parent, which is then handled, and so forth. The process stops at the root at the latest.

The violation at a node is removed in $O(1)$ time by a local restructuring of the tree and/or a change of balance information, giving a total worst case update time proportional to the height of the tree. The fundamental restructuring operation is the *rotation*, shown in Figure 10.1. It was introduced in [1]. The crucial feature of a rotation is that it preserves the in-order invariant of the search tree while allowing one subtree to be moved upwards in the tree at the expense of another.

A rotation may be seen as substituting a connected subgraph T consisting of two nodes with a new connected subgraph T' on the same number of nodes, redistributing the keys (here x and y) in T' according to in-order, and redistributing the subtrees rooted at leaves

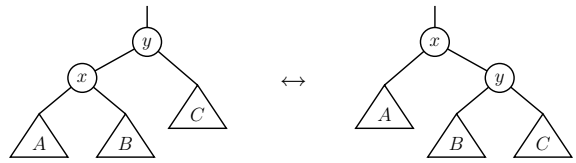

FIGURE 10.1: Rotation.

of T by attaching them as leaves of T' according to in-order. Described in this manner, it is clear that in-order will be preserved for *any* two subgraphs T and T' having an equal number of nodes. One particular case is the *double rotation* shown in Figure 10.2, so named because it is equivalent to two consecutive rotations.

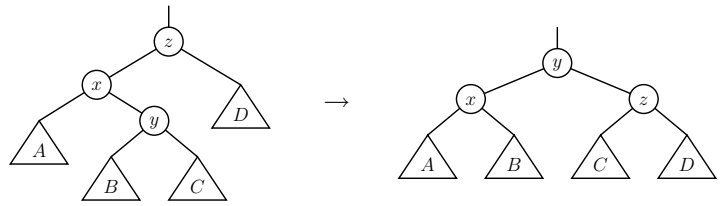

FIGURE 10.2: Double rotation.

Actually, any such transformation of a connected subgraph T to another T' on the same number of nodes can be executed through a series of rotations. This can be seen by noting that any connected subgraph can be converted into a right-path, i.e., a tree where all left children are empty trees, by repeated rotations (in Figure 10.1, if y but not x is on the rightmost path in the tree, the rotation will enlarge the rightmost path by one node). Using the right-path as an intermediate state and running one of the conversions backwards will transform T into T'. The double rotation is a simple case of this. In a large number of rebalancing schemes, the rebalancing algorithm performs at most one rotation or double rotation per node on the search path.

We note that rebalancing schemes exist [34] where the rebalancing along the search path is done in a top-down fashion instead of the bottom-up fashion described above. This is useful when several processes concurrently access the tree, as discussed in Section 10.8.

In another type of rebalancing schemes, the restructuring primitive used is the rebuilding of an entire subtree to perfect balance, where perfect balance means that any node is the median among the nodes in its subtree. This primitive is illustrated in Figure 10.3. In these rebalancing schemes, the restructuring is only applied to one node on the search path for the update, and this resolves all violations of the balance invariant.

The use of this rebalancing technique is sometimes termed *local* or *partial rebuilding* (in contrast to global rebuilding of data structures, which designates a periodically rebuilding of the entire structure). In Section 10.5, we discuss linear time algorithms for rebalancing a (sub-)tree to perfect balance.

10.3.3 Complexity Results

Rebalancing schemes can be graded according to several complexity measures. One such measure is how much rebalancing work is needed after an update. For this measure, typical

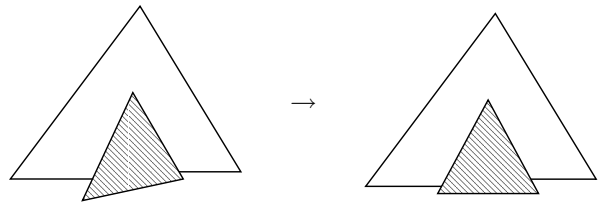

FIGURE 10.3: Rebuilding a subtree.

values include amortized $O(\log n)$, worst case $O(\log n)$, amortized $O(1)$, and worst case $O(1)$. Values below logarithmic may at first sight seem useless due to the logarithmic search time of balanced search trees, but they are relevant in a number of settings. One setting is finger search trees (described in a chapter of their own in this book), where the search for the update point in the tree does not start at the root and hence may take sub-logarithmic time. Another setting is situations where the nodes of the tree are annotated with information which is expensive to update during restructuring of the tree, such that rotations may take non-constant time. This occurs in Computational Geometry, for instance. A third setting is concurrent access to the tree by several processes. Searching the tree concurrently is not a problem, whereas concurrent updates and restructuring may necessitate lockings of nodes in order to avoid inconsistencies. This makes restructuring more expensive than searches.

Another complexity measure is the exact height maintained. The majority of schemes maintain a height bounded by $c \cdot \log n$ for some constant $c > 1$. Of other results, splay trees [70] have no sub-linear bound on the height, but still perform searches in amortized $O(\log n)$ time. Splay trees are described in a chapter of their own in this book. In the other direction, a series of papers investigate how close c can get to the optimal value one, and at what rebalancing cost. We discuss these results in Section 10.7.

One may also consider the exact amount of balance information stored in each node. Some schemes store an integer, while some only need one or two bits. This may effect the space consumption of nodes, as a single bit may be stored implicitly, e.g., as the sign bit of a pointer, or by storing subtrees out of order when the bit is set. Schemes even exist which do not need to store any information at all in nodes. We discuss these schemes in Section 10.6

Finally, measures such as complexity of implementation and performance in practice can also be considered. However, we will not discuss these here, mainly because these measures are harder to quantify.

10.4 Classic Balancing Schemes

10.4.1 AVL-Trees

AVL-trees where introduced in 1962 in [1], and are named after their inventors Adel'son-Vel'ski and Landis. They proposed the first dictionary structure with logarithmic search and update times, and also introduced the rebalancing technique using rotations.

The balance definition in AVL-trees is based on the height of subtrees. The invariant is that for any node, the heights of its two subtrees differ by at most one. Traditionally, the balance information maintained at each node is $+1$, 0, or -1, giving the difference in heights between the right subtree and the left subtree. This information can be represented by two bits. Another method is to mark a node when its height is larger than its siblings. This requires only one bit per node, but reading the balance of a node now involves visiting

its children. In the other direction, storing the height of each node requires $\log\log n$ bits of information per node, but makes the rebalancing algorithms simpler to describe and analyze.

By induction on h, it is easily proved that for an AVL-tree of height h, the minimum number of nodes is $F_{h+2} - 1$, where F_i denotes the i'th Fibonacci number, defined by $F_1 = F_2 = 1$ and $F_{j+2} = F_{j+1} + F_j$. A well-known fact for Fibonacci numbers is that $F_i \geq \Phi^{i-2}$, where Φ is the golden ratio $(\sqrt{5}+1)/2 \approx 1.618$. This shows that the height of an AVL-tree with n nodes is at most $\log_\Phi(n+1)$, i.e., AVL-trees have a height bound of the type $c \cdot \log n$ with $c = 1/\log\Phi \approx 1.440$.

After an update, violations of the balance invariant can only occur at nodes on the search path from the root to the update point, as only these nodes have subtrees changed. The rebalancing algorithm resolves these in a bottom-up fashion. At each node, it either performs a rotation, performs a double rotation, or just updates balance information, with the choice depending on the balance of its child and grandchild on the search path. The algorithm stops when it can guarantee that no ancestor has a balance problem, or when the root is reached.

In AVL-trees, the rebalancing algorithm has the following properties: After an insertion, change of balance information may take place any number of steps towards the root, but as soon as a rotation or double rotation takes place, no further balance problems remain. Hence, only $O(1)$ structural change is made. In contrast, after a deletion it may happen that rotations are performed at all nodes on the search path. If only insertions take place, the amortized amount of rebalancing work, including updating of balance information, can be shown [58] to be $O(1)$. The same is true if only deletions take place [75]. It is not true in the fully dynamic case, as it is easy to find an AVL-tree where alternating insertions and deletions of the same key require rebalancing along the entire search path after each update.

10.4.2 Weight-Balanced Trees

Weight-balanced trees were proposed in 1973 by Nievergelt and Reingold [62], and have a balance definition based on the sizes of subtrees. Here, the size of a subtree is most conveniently defined as the number of external nodes (empty trees) in the subtree, and the size, also denoted the *weight*, of a node is the size of its subtree. The balance invariant of weight-balanced trees states that for any node, the ratio between its own weight and the weight of its right child (or left) is in the interval $[\,\alpha\,,\,1-\alpha\,]$ for some fixed value $\alpha > 0$. This ratio is denoted the *balance* of the node. Since a node of weight three must have subtrees of weight two and one, we must have $\alpha \leq 1/3$. Weight-balanced trees are also called BB[α]-trees, which stands for trees of bounded balance with parameter α.

By the balance criterion, for any node v the weight of the parent of v is at least a factor $1/(1-\alpha)$ larger than the weight of v. A tree of height k therefore has a root of weight at least $1/(1-\alpha)^k$, which shows that the height of a weight-balanced tree with n nodes is at most $\log_{1/(1-\alpha)}(n+1)$, i.e., weight-balanced trees have a height bound of the type $c \cdot \log n$ with $c = -1/\log(1-\alpha) > 1.709$.

The balance information stored in each node is its weight, for which $\log n$ bits are needed. After an update, this information must be updated for all nodes on the search path from the root to the update point. Some of these nodes may now violate the balance criterion. The rebalancing algorithm proposed in [62] resolves this unbalance in a bottom-up fashion along the search path using either a rotation or a double rotation at each violating node. The choice of rotation depends on the weight of the children and the grandchildren of the node.

In [62], the rebalancing algorithm was claimed to work for α in the interval $[\,0\,,\,1-1/\sqrt{2}\,]$, but Blum and Mehlhorn [20] later observed that the correct interval is $(2/11\,,\,1-1/\sqrt{2}\,]$. They also showed that for α strictly inside this interval, the rebalancing of an unbalanced node restores its balance to a value in $[\,(1+\delta)\alpha\,,\,1-(1+\delta)\alpha\,]$, where δ depends on the choice of α. This implies that when the node becomes unbalanced again, the number of updates which have taken place below it since it was last rebalanced is at least a fraction (depending on α) of its current weight. This feature, unique to weight-balanced trees, has important applications, e.g., for data structures in Computational Geometry. A number of these structures are binary search trees where each node has an associated secondary structure built on the elements in the subtree of the node. When a rotation takes place, the structures of the nodes taking part in the rotation will have to be rebuilt. If we attribute the cost of this rebuilding evenly to the updates which have taken place below the node since it was last involved in a rotation, then, as an example, a linear rebuilding cost of the secondary structure will amount to a constant attribution to each of these updates. As the search path for an update contains $O(\log n)$ nodes, any single update can at most receive this many attributions, which implies an amortized $O(\log n)$ update complexity for the entire data structure.

The same analysis allows BB[α]-trees to be maintained by local rebuilding instead of rotations in amortized $O(\log n)$ time, as first noted by Overmars and van Leeuwen [69]: After an update, the subtree rooted at the highest unbalanced node (if any) on the search path is rebuilt to perfect balance. Since a rebuilding of a subtree leaves all nodes in it with balance close to 1/2, the number of updates which must have taken place below the node since it was last part of a rebuilding is a constant fraction of its current weight. The rebuilding uses work linear in this weight, which can be covered by attributing a constant amount of work to each of these updates. Again, each update is attributed $O(\log n)$ work. This scheme will work for any $\alpha \leq 1/3$.

For the original rebalancing algorithm using rotations, a better analysis can be made for α chosen strictly inside the interval $(2/11\,,\,1-1/\sqrt{2}\,]$: The total work per rebalancing operation is now $O(1)$, so the work to be attributed to each update below a node is $O(1/w)$, where w is the weight of the node. As noted above in the proof of the height bound of weight-balanced trees, w is exponentially increasing along the search path from the update point to the root. This implies that each update is attributed only $O(1)$ work in total, and also that the number of rotations taking place at a given height decreases exponentially with the height. This result from [20] seems to be the first on $O(1)$ amortized rebalancing in binary search trees. The actual time spent after an update is still logarithmic in weight-balanced trees, though, as the balance information needs to be updated along the entire search path, but this entails no structural changes.

Recently, the idea of balancing by weight has been applied to multi-way search trees [14], leading to trees efficient in external memory which posses the same feature as weight-balanced binary trees, namely that between each rebalancing at a node, the number of updates which have taken place below the node is proportional to the weight of the node.

10.4.3 Balanced Binary Trees Based on Multi-Way Trees.

The B-tree [17], which is treated in another chapter of this book, is originally designed to handle data stored on external memory. The basic idea is to associate a physical block with a high-degree node in a multi-way tree. A B-tree is maintained by merging and splitting nodes, and by increasing and decreasing the number of layers of multi-way nodes. The smallest example of a B-tree is the 2-3-tree [2], where the nodes have degree 2 or 3. In a typical B-tree implementation, the degree of a node is much larger, and it varies roughly

within a factor of 2.

The concept of multi-way nodes, splitting, and merging, has also proven to be very fruitful in the design of balancing schemes for binary trees. The first such example is the binary B-tree [15], a binary implementation of 2-3-trees. Here, the idea is to organize binary nodes into larger chunks of nodes, here called *pseudo-nodes*. In the binary version of a 2-3-tree, a node of degree 2 is represented by one binary node, while a node of degree 3 is represented as two binary nodes (with the additional constraint that one of the two nodes is the right child of the other). In the terms of binary nodes grouped into pseudo-nodes, it is convenient to say that edges within a pseudo-node are *horizontal* while edges between pseudo-nodes are *vertical*.

As a natural extension of binary B-trees, Bayer invented *Symmetric Binary Trees*, or SBB-trees [16]. The idea was that, instead of only allowing a binary node to have one horizontal outgoing edge to its right child, we can allow both left- and right-edges to be horizontal. For both binary B-trees and Symmetric Binary B-trees, Bayer designed maintenance algorithms, where the original B-tree operations split, merge, and increase/decrease number of levels were implemented for the pseudo-nodes.

Today, SBB-trees mostly appear under the name *red-black trees* [34]. Here, the horizontal and vertical edges are represented by one "color" per node. (Both notations can be represented by one bit per node.) SBB/red-black trees are binary implementations of B-trees where each node has degree between 2 and 4.

One advantage with SBB-trees/red-black trees is that a tree can be updated with only a constant number of rotations per insertion or deletion. This property is important for example when maintaining priority search trees [56] where each rotation requires $\Theta(\log n)$ time.

The first binary search tree with $O(1)$ rotations per update was the half-balanced trees by Olivié [66]. Olivié's idea was to use *path-balancing*, where the quotient between the shortest and longest path from each node is restricted to be at most $1/2$, and he showed that this path-balance could be maintained with $O(1)$ rotations per update. It turns out to be the case that half-balanced trees and SBB/red-black trees are structurally equivalent, although their maintenance algorithms are different. It has also been proven by Tarjan [73] that SBB/red-black trees can be maintained by $O(1)$ rotations. These algorithms can also be generalized to maintain pseudo-nodes of higher degree, resulting in binary B-tree implementations with lower height [8], still requiring $O(1)$ rotations per update.

The mechanism behind the constant number of rotations per update can be explained in a simple way by examining three cases of what can happen during insertion and deletion in a binary B-tree representation.

- When a pseudo-node becomes too large, it can be split into two pseudo-nodes without any rotation; we just need to change the balance information.
- Also, when a pseudo-node becomes too small and its sibling has minimal size, these two nodes can be merged without any rotation; we just change balance information.
- In all other cases, when a pseudo-node becomes too small or too large, this will be resolved by moving nodes between the pseudo-node and its sibling and no splitting or merging will take place.

From these three basic facts, it can be shown that as soon as the third case above occurs, no more rebalancing will be done during the same update. Hence, the third case, requiring rotations, will only occur once per update. For details, we refer to the literature [8, 73].

Binary B-trees can also be used to design very simple maintenance algorithms that are

easy to code. This is illustrated by AA-trees [5, 77]. AA-trees are actually the same as Bayer's binary version of 2-3-trees, but with design focused on simplicity. Compared with normal red-black tree implementations, AA-trees require very few different cases in the algorithm and much less code for implementation.

While binary B-trees and SBB/red-black trees deal with small pseudo-nodes, the stratified trees by van Leeuwen and Overmars [76] use large pseudo-nodes arranged in few layers. The concept of stratification does not imply that all pseudo-nodes have similar size; it is mainly a way to conceptually divide the tree into layers, using the notion of merging and splitting.

10.5 Rebalancing a Tree to Perfect Balance

A basic operation is the rebalancing operation, which takes a binary tree as input and produces a balanced tree. This operation is important in itself, but it is also used as a subroutine in balancing schemes (see Section 10.6).

It is quite obvious that one can construct a perfectly balanced tree from an ordered tree, or a sorted list, in linear time. The most straightforward way is to put the elements in sorted order into an array, take the median as the root of the tree, and construct the left and right subtrees recursively from the upper and lower halves of the array. However, this is unnecessarily cumbersome in terms of time, space, and elegance.

A number of restructuring algorithms, from the type mentioned above to more elegant and efficient ones based on rotations, can be found in the literature [26, 27, 33, 54, 72]. Of these, the one by Stout and Warren [72] seems to be most efficient. It uses the following principle:

1. *Skew.* Make right rotations at the root until no left child remains. Continue down the right path making right rotations until the entire tree becomes one long rightmost path (a "vine").
2. *Split.* Traverse down the vine a number of times, each time reducing the length of the vine by left rotations.

If we start with a vine of length $2^p - 1$, for some integer p, and make one rotation per visited node, the resulting vine will be of length $2^{p-1} - 1$ after the first pass, $2^{p-2} - 1$ after the second pass, etc., until the vine is reduced to a single node; the resulting tree is a perfectly balanced tree. If the size of the tree is $2^p - 1$, this will work without any problem. If, however, the size is not a power of two, we have to make some special arrangements during the first pass of left rotations. Stout and Warren solved the problem of how to make evenly distributed rotations along the vine in a rather complicated way, but there is a simpler one. It has never before been published in itself, but has been included in demo software and in published code [6, 11].

The central operation is a split operation that takes as parameters two numbers p_1 and p_2 and compresses a right-skewed path of p_1 nodes into a path of p_2 nodes ($2p_2 \geq p_1$). The simple idea is to use a counter stepping from $p_1 - p_2$ to $p_2(p_1 - p_2)$ with increment $p_1 - p_2$. Every time this counter reaches or exceeds a multiple of p_2, a rotation is performed. In effect, the operation will make $p_1 - p_2$ evenly distributed left rotations.

With this split operation available, we can do as follows to rebalance a tree of size n (n internal nodes): First, skew the tree. Next, find the largest integer b such that b is an even power of 2 and $b - 1 \leq n$. Then, if $b - 1 < n$, call Split with parameters n and $b - 1$. Now, the vine will have proper length and we can traverse it repeatedly, making a left rotation at each visited node, until only one node remains.

In contrast to the Stout-Warren algorithm, this algorithm is straightforward to implement. We illustrate it in Figure 10.4. We describe the five trees, starting with the topmost:

1. A tree with 12 internal nodes to be balanced.
2. After Skew.
3. With $n = 12$ and $b = 8$, we call split with parameters 12 and 7, which implies that five evenly distributed rotations will be made. As the result, the vine will be of length 7, which fulfills the property of being $2^p - 1$.
4. The next split can be done by traversing the vine, making one left rotation at each node. As a result, we get a vine of length 3 (nodes 3, 6, and 10).
5. After the final split, the tree is perfectly balanced.

10.6 Schemes with no Balance Information

As discussed above, a balanced binary search tree is typically maintained by local constraints on the structure of the tree. By keeping structure information in the nodes, these constraints can be maintained during updates.

In this section, we show that a plain vanilla tree, without any local balance information, can be maintained efficiently. This can be done by coding the balance information implicitly (Section 10.6.1) or by using global instead of local balance criteria, hereby avoiding the need for balance information (Section 10.6.2). Splay trees [70] also have no balance information. They do not have a sub-linear bound on their height, but still perform searches in amortized $O(\log n)$ time. Splay trees are described in a chapter of their own in this book.

10.6.1 Implicit Representation of Balance Information

One idea of how to remove the need for local balance information is to store the information implicitly. There are two main techniques for this: coding information in the way empty pointers are located or coding information by changing the order between left and right children.

In both cases, we can easily code one bit implicitly at each internal node, but not at external nodes. Therefore, we weed to use balance schemes that can do with only one bit per internal node and no balance information at external nodes.

As an example, we may use the AVL-tree. At each node, we need to keep track of whether the two subtrees have the same height or if one of them is one unit higher than its sibling. We can do this with one bit per internal node by letting the bit be 1 if and only if the node is higher than its sibling. For external nodes we know the height, so no balance information is needed there.

The assumption that we only need one bit per internal node is used in the two constructions below.

Using Empty Pointers

As pointed out by Brown [24, 25], the explicitly stored balance information may in some classes of balanced trees be eliminated by coding the information through the location of empty pointers. We use a tree of pseudo-nodes, where a pseudo-node contains two consecutive elements, stored in two binary nodes. The pseudo-node will have three outgoing pointers, and since the two binary nodes are consecutive, one of the three pointers will be

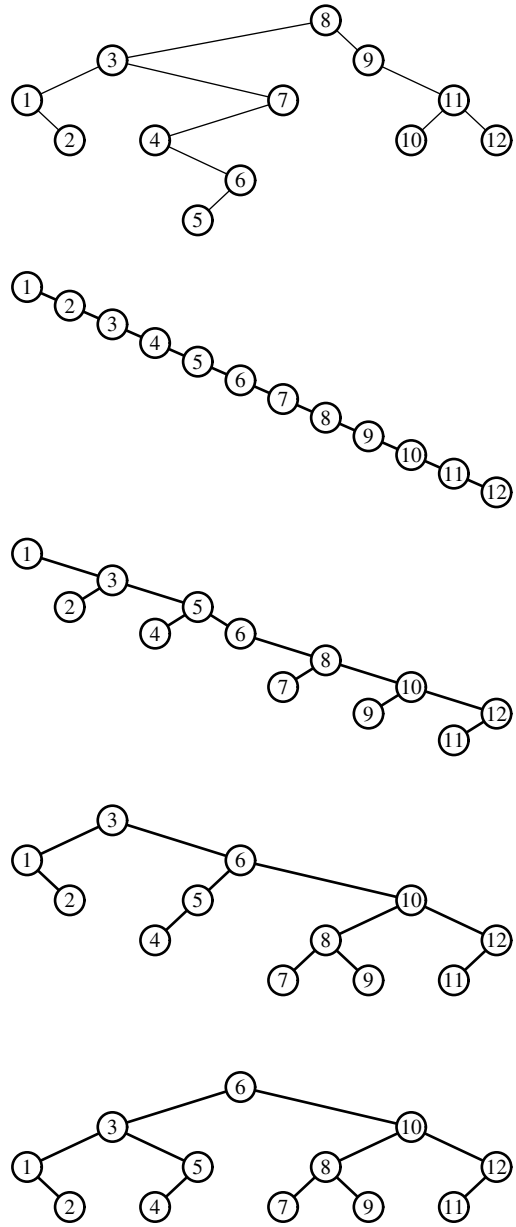

FIGURE 10.4: Rebalancing a binary search tree.

empty. By varying which of the two nodes become parent, we can arrange the pseudo-node in two ways. These two different structures is used to represent bit values 0 and 1, respectively; by checking the position of the empty pointer, we can compute the bit value. In order for this to work, we allow the pseudo-nodes at the bottom of the tree to contain one or two binary nodes.

During insertion, we traverse down the tree. If the inserted element lies between the two keys in a visited pseudo-node, we replace it by one of the elements in the pseudo-node and

continue down the tree with that element instead. At the bottom of the tree, if we find a pseudo-node with only one key, we just add the new key. If, on the other hand, we find a pseudo-node with two keys, we split it into two pseudo-nodes which will cause an insertion in the tree of pseudo-nodes. Rotations etc. can be done with pseudo-nodes instead of ordinary binary nodes. (If a rotation involves the lowest level of the tree of pseudo-nodes, some care has to be taken in order to maintain the invariant that only the lowest pseudo-nodes may contain a single node.)

Deletions are handled correspondingly. If the deleted element is contained in an internal pseudo-node, we replace it by its predecessor or successor, which resides at the bottom of the tree; in this way we ensure that the deletion occurs at the bottom. If the deletion occurs at a pseudo-node with two binary nodes, we just remove the node, if the pseudo-node contains only one node, a deletion occurs in the tree of pseudo-nodes.

Despite the pseudo-nodes, the tree is really just a binary search tree where no balance information is explicitly stored. Since each pseudo-node has internal height 2, and the number of pseudo-nodes is less than n, the height of the binary tree is $O(\log n)$. A drawback is that the height of the underlying binary tree will become higher by the use of pseudo-nodes. Instead of n internal nodes we will have roughly $n/2$ pseudo-nodes, each of height 2. In the worst case, the height of the binary tree will be doubled.

Swapping Pointers

Another possibility for coding information into a structure is to use the ordering of nodes. If we redefine binary search trees, such that the left and right subtree of a node are allowed to change place, we can use this possibility to encode one bit per node implicitly. By comparing the keys of the two children of a node, the one-bit information can be extracted. During search, we have to make one comparison extra at each node. This idea has been used by Munro and Suwanda [59–61] to achieve implicit implementation of binary search trees, but it can of course also be used for traditional pointer-based tree structures.

10.6.2 General Balanced Trees

In the following, we use $|T|$ to denote the weight (number of leaves) in a tree T. We also use $|v|$ to denote the weight of a subtree rooted at node v. It should be noted that for a tree T storing n keys in internal nodes, $|T| = n + 1$

Instead of coding balance information into the structure of the tree, we can let the tree take any shape, as long as its height is logarithmic. Then, there is no local balance criterion to maintain, and we need no balance information in the nodes, not even implicitly coded. As we show below, the tree can still be maintained efficiently.

When maintaining trees this way, we use the technique of *partial rebuilding*. This technique was first introduced by Overmars and van Leeuwen [68, 69] for maintaining weight-balanced trees. By making a partial rebuilding at node v, we mean that the subtree rooted at v is rebuilt into a perfectly balanced tree. The cost of such rebalancing is $\Theta(|v|)$. In Section 10.5, we discuss linear time algorithms for rebalancing a (sub-)tree to perfect balance.

Apart from the advantage of requiring no balance information in the nodes, it can be shown [7] that the constant factor for general balanced trees is lower than what has been shown for the maintenance of weight-balanced trees by partial rebuilding.

The main idea in maintaining a general balanced tree is to let the tree take *any* shape as long as its height does not exceed $\log |T|$ by more than a specified constant factor. The key observation is that whenever the tree gets too high by an insertion, we can find a node where partial rebuilding can be made at a low amortized cost. (Since deletions do not

increase the height of the tree, we can handle deletions efficiently by rebuilding the entire tree after a large number of elements have been deleted.)

We use two constants $c > 1$, and $b > 0$, and we maintain a balanced tree T with maximum height $\lceil c \log |T| + b \rceil$.

No balance information is used, except two global integers, containing $|T|$, the number of leaves in T, and $d(T)$, the number of deletions made since the last time the entire tree T was rebalanced.

Updates are performed in the following way:

Insertion: If the depth of the new leaf exceeds $\lceil c \log(|T|+d(T)) \rceil$, we back up along the insertion path until we find the lowest node v, such that $h(v) > \lceil c \log |v| \rceil$. The subtree v is then rebuilt to perfect balance. The node v is found by explicitly traversing the subtrees below the nodes on the path from the inserted leaf to v, while counting the number of leaves. The cost for this equals the cost for traversing the subtree below v once, which is $O(|v|)$.

Deletion: $d(T)$ increases by one. If $d(T) \geq (2^{b/c} - 1)|T|$, we rebuild T to perfect balance and set $d(T) = 0$.

First, we show that the height is low enough. Since deletions do not increase the height of T, we only need to show that the height is not increased too much by an insertion. We prove this by induction. Assume that

$$h(T) \leq \lceil c \log(|T| + d(T)) \rceil \tag{10.1}$$

holds before an insertion. (Note that the height of an empty tree is zero.) During the insertion, the height condition can only be violated by the new node. However, if such a violation occurs, the partial rebuilding will ensure that Inequality 10.1 holds after the insertion. Hence, Inequality 10.1 holds by induction. Combining this with the fact that $d(T) < (2^{b/c} - 1)|T|$, we get that $h(T) \leq \lceil c \log |T| + b \rceil$.

Next, we show that the maintenance cost is low enough. Since the amortized cost for the rebuilding of the entire tree caused by deletions is obviously O(1) per deletion, we only need to consider insertions.

In fact, by the way we choose where to perform rebuilding, we can guarantee that *when a partial rebuilding occurs at node v, $\Omega(v)$ updates have been made below v since the last time v was involved in a partial rebuilding.* Indeed, this observation is the key observation behind general balanced trees.

Let v_H be v's child on the path to the inserted node. By the way v is selected by the algorithm, we know the following about v and v_h:

$$\begin{aligned} h(v) &> \lceil c \log |v| \rceil && (10.2)\\ h(v_H) &\leq \lceil c \log |v_H| \rceil && (10.3)\\ h(v) &= h(v_h) + 1 && (10.4) \end{aligned}$$

Combining these, we get

$$\lceil c \log |v| \rceil < h(v) = h(v_H) + 1 \leq \lceil c \log |v_H| \rceil + 1 \tag{10.5}$$

and, thus

$$\begin{aligned} \log |v| &< \log |v_H| + 1/c \\ |v_H| &> 2^{-1/c}|v| && (10.6) \end{aligned}$$

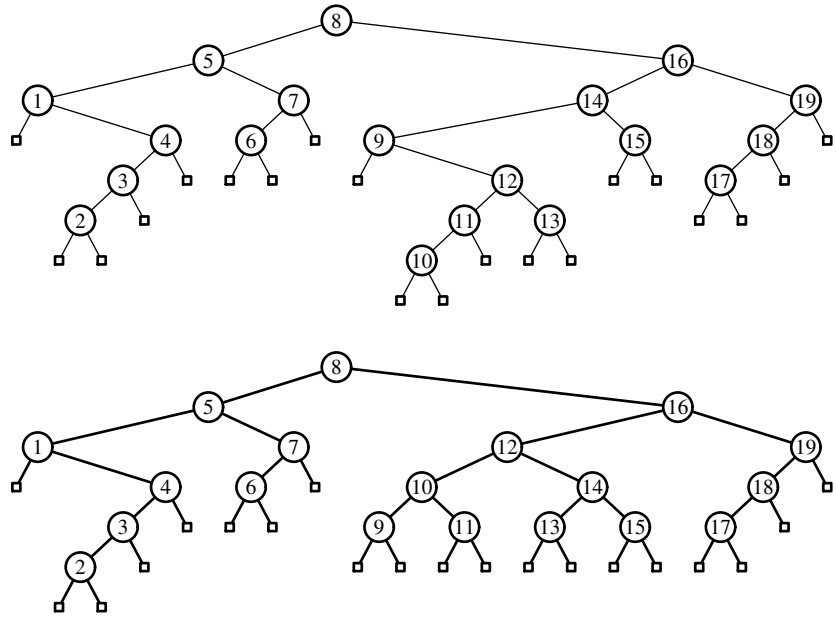

FIGURE 10.5: Upper tree: A GB(1.2)-tree which requires rebalancing. Lower tree: After partial rebuilding.

Since $2^{-1/c} > 1/2$, we conclude that the weight of v_H is $\Theta(v)$ larger than the weight of v's other child. The only way this difference in weight between the two children can occur is by insertions or deletion below v. Hence, $\Omega(v)$ updates must have been made below v since the last time v was involved in a partial rebuilding. In order for the amortized analysis to hold, we need to reserve a constant cost at v for each update below v. At each update, updates are made below $O(\log n)$ nodes, so the total reservation per update is $O(\log n)$.

Since the tree is allowed to take any shape as long as its height is low enough, we call this type of balanced tree *general balanced trees* [7]. We use the notation GB-trees or GB(c)-trees, where c is the height constant above. (The constant b is omitted in this notation.) (The idea of general balanced trees have also been rediscovered under the name scapegoat trees [33].)

Example. The upper tree in Figure 10.5 illustrates a GB(1.2)-tree where five deletions and some insertions have been made since the last global rebuilding. When inserting 10, the height becomes 7, which is too high, since $7 > \lceil c \log(|T|+d(T))\rceil = \lceil 1.2 \log(20+5)\rceil = 6$. We back up along the path until we find the node 14. The height of this node is 5 and the weight is 8. Since $5 > \lceil 1.2 \log 8\rceil$, we can make a partial rebuilding at that node. The resulting tree is shown as the lower tree in Figure 10.5.

10.6.3 Application to Multi-Dimensional Search Trees

The technique of partial rebuilding is an attractive method in the sense that it is useful not only for ordinary binary search trees, but also for more complicated data structures, such as multi-dimensional search trees, where rotations cannot be used efficiently. For example, partial rebuilding can be used to maintain logarithmic height in k-d trees [19]

under updates [57, 68, 69]. A detailed study of the use of partial rebuilding can be found in Mark Overmars' Ph.D. thesis [68]. For the sake of completeness, we just mention that if the cost of rebalancing a subtree v is $O(P(|v|))$, the amortized cost of an update will be $O\left(\frac{P(n)}{n}\log n\right)$. For example, applied to k-d trees, we get an amortized update cost of $O(\log^2 n)$.

10.7 Low Height Schemes

Most rebalancing schemes reproduce the result of AVL-trees [1] in the sense that they guarantee a height of $c \cdot \log(n)$ for some constant $c > 1$, while doing updates in $O(\log n)$ time. Since the height determines the worst-case complexity of almost all operations, it may be reasonable to ask exactly how close to the best possible height $\lceil \log(n+1) \rceil$ a tree can be maintained during updates. Presumably, the answer depends on the amount of rebalancing work we are willing to do, so more generally the question is: given a function f, what is the best possible height maintainable with $O(f(n))$ rebalancing work per update?

This question is of practical interest—in situations where many more searches than updates are performed, lowering the height by factor of (say) two will improve overall performance, even if it is obtained at the cost of a larger update time. It is also of theoretical interest, since we are asking about the inherent complexity of maintaining a given height in binary search trees. In this section, we review the existing answers to the question.

Already in 1976, Maurer et al. [55] proposed the k-neighbor trees, which guarantee a height of $c \cdot \log(n)$, where c can be chosen arbitrarily close to one. These are unary-binary trees, with all leaves having the same depth and with the requirement that between any two unary nodes on the same level, at least $k-1$ binary nodes appear. They may be viewed as a type of $(1,2)$-trees where the rebalancing operations exchange children, not only with neighboring nodes (as in standard (a,b)-tree or B-tree rebalancing), but with nodes a horizontal distance k away. Since at each level, at most one out of k nodes is unary, the number of nodes increases by a factor of $(2(k-1)+1)/k = 2 - 1/k$ for each level. This implies a height bound of $\log_{2-1/k} n = \log(n)/\log(2-1/k)$. By first order approximation, $\log(1+x) = \Theta(x)$ and $1/(1+x) = 1 - \Theta(x)$ for x close to zero, so $1/\log(2-1/k) = 1/(1+\log(1-1/2k)) = 1 + \Theta(1/k)$. Hence, k-trees maintain a height of $(1+\Theta(1/k))\log n$ in time $O(k \log n)$ per update.

Another proposal [8] generalizes the red-black method of implementing $(2,4)$-trees as binary trees, and uses it to implement (a,b)-trees as binary trees for $a = 2^k$ and $b = 2^{k+1}$. Each (a,b)-tree node is implemented as a binary tree of perfect balance. If the underlying (a,b)-tree has t levels, the binary tree has height at most $t(k+1)$ and has at least $(2^k)^t = 2^{kt}$ nodes. Hence, $\log n \geq tk$, so the height is at most $(k+1)/k \log n = (1+1/k)\log n$. As in red-black trees, a node splitting or fusion in the (a,b)-tree corresponds to a constant amount of recoloring. These operations may propagate along the search path, while the remaining rebalancing necessary takes place at a constant number of (a,b)-tree nodes. In the binary formulation, these operations involve rebuilding subtrees of size $\Theta(2^k)$ to perfect balance. Hence, the rebalancing cost is $O(\log(n)/k + 2^k)$ per update.

Choosing $k = \lfloor \log\log n \rfloor$ gives a tree with height bound $\log n + \log(n)/\log\log(n)$ and update time $O(\log n)$. Note that the constant for the leading term of the height bound is now one. To accommodate a non-constant k, the entire tree is rebuilt when $\lfloor \log\log n \rfloor$ changes. Amortized this is $O(1)$ work, which can be made a worst case bound by using incremental rebuilding [68].

Returning to k-trees, we may use the method of non-constant k also there. One possibility

is $k = \Theta(\log n)$, which implies a height bound as low as $\log n + O(1)$, maintained with $O(\log^2 n)$ rebalancing work per update. This height is $O(1)$ from the best possible. A similar result can be achieved using the general balanced trees described in Section 10.6: In the proof of complexity in that section, the main point is that the cost $|v|$ of a rebuilding at a node v can be attributed to at least $(2^{-1/c} - 1/2)|v|$ updates, implying that each update is attributed at most $(1/(2^{-1/c} - 1/2))$ cost at each of the at most $O(\log n)$ nodes on the search path. The rebalancing cost is therefore $O(1/(2^{-1/c} - 1/2) \log n)$ for maintaining height $c \cdot \log n$. Choosing $c = 1 + 1/\log n$ gives a height bound of $\log n + O(1)$, maintained in $O(\log^2 n)$ amortized rebalancing work per update, since $(2^{-1/(1+1/\log n))} - 1/2)$ can be shown to be $\Theta(1/\log n)$ using the first order approximations $1/(1 + x) = 1 - \Theta(x)$ and $2^x = 1 + \Theta(x)$ for x close to zero.

We note that a binary tree with a height bound of $\log n + O(1)$ in a natural way can be embedded in an array of length $O(n)$: Consider a tree T with a height bound of $\log n + k$ for an integer k, and consider n ranging over the interval $[2^i; 2^{i+1}[$ for an integer i. For n in this interval, the height of T never exceeds $i + k$, so we can think of T as embedded in a virtual binary tree T' with $i + k$ completely full levels. Numbering nodes in T' by an in-order traversal and using these numbers as indexes in an array A of size $2^{i+k} - 1$ gives an embedding of T into A. The keys of T will appear in sorted order in A, but empty array entries may exist between keys. An insertion into T which violates the height bound corresponds to an insertion into the sorted array A at a non-empty position. If T is maintained by the algorithm based on general balanced trees, rebalancing due to the insertion consists of rebuilding some subtree in T to perfect balance, which in A corresponds to an even redistribution of the elements in some consecutive segment of the array. In particular, the redistribution ensures an empty position at the insertion point.

In short, the tree rebalancing algorithm can be used as a maintenance algorithm for a sorted array of keys supporting insertions and deletions in amortized $O(\log^2 n)$ time. The requirement is that the array is never filled to more than some fixed fraction of its capacity (the fraction is $1/2^{k-1}$ in the example above). Such an amortized $O(\log^2 n)$ solution, phrased directly as a maintenance algorithm for sorted arrays, first appeared in [38]. By the converse of the embedding just described, [38] implies a rebalancing algorithm for low height trees with bounds as above. This algorithm is similar, but not identical, to the one arising from general balanced trees (the criteria for when to rebuild/redistribute are similar, but differ in the details). A solution to the sorted array maintenance problem with worst case $O(\log^2 n)$ update time was given in [78]. Lower bounds for the problem appear in [28, 29], with one of the bounds stating that for algorithms using even redistribution of the elements in some consecutive segment of the array, $O(\log^2 n)$ time is best possible when the array is filled up to some constant fraction of its capacity.

We note that the correspondence between the tree formulation and the array formulation only holds when using partial rebuilding to rebalance the tree—only then is the cost of the redistribution the same in the two versions. In contrast, a rotation in the tree will shift entire subtrees up and down at constant cost, which in the array version entails cost proportional to the size of the subtrees. Thus, for pointer based implementation of trees, the above $\Omega(\log^2 n)$ lower bound does not hold, and better complexities can be hoped for.

Indeed, for trees, the rebalancing cost can be reduced further. One method is by applying the idea of *bucketing*: The subtrees on the lowest $\Theta(\log K)$ levels of the tree are changed into buckets holding $\Theta(K)$ keys. This size bound is maintained by treating the buckets as (a, b)-tree nodes, i.e., by bucket splitting, fusion, and sharing. Updates in the top tree only happen when a bucket is split or fused, which only happens for every $\Theta(K)$ updates in the bucket. Hence, the amortized update time for the top tree drops by a factor K. The buckets themselves can be implemented as well-balanced binary trees—using the schemes

above based on k-trees or general balanced trees for both top tree and buckets, we arrive at a height bound of $\log n + O(1)$, maintained with $O(\log \log^2 n)$ amortized rebalancing work. Applying the idea recursively inside the buckets will improve the time even further. This line of rebalancing schemes was developed in [3, 4, 9, 10, 42, 43], ending in a scheme [10] maintaining height $\lceil \log(n+1) \rceil + 1$ with $O(1)$ amortized rebalancing work per update.

This rather positive result is in contrast to an observation made in [42] about the cost of maintaining exact optimal height $\lceil \log(n+1) \rceil$: When $n = 2^i - 1$ for an integer i, there is only one possible tree of height $\lceil \log(n+1) \rceil$, namely a tree of i completely full levels. By the ordering of keys in a search tree, the keys of even rank are in the lowest level, and the keys of odd rank are in the remaining levels (where the rank of a key k is defined as the number of keys in the tree that are smaller than k). Inserting a new smallest key and removing the largest key leads to a tree of same size, but where all elements previously of odd rank now have even rank, and vice versa. If optimal height is maintained, all keys previously in the lowest level must now reside in the remaining levels, and vice versa—in other words, the entire tree must be rebuilt. Since the process can be repeated, we obtain a lower bound of $\Omega(n)$, even with respect to amortized complexity. Thus, we have the intriguing situation that a height bound of $\lceil \log(n+1) \rceil$ has amortized complexity $\Theta(n)$ per update, while raising the height bound a trifle to $\lceil \log(n+1) \rceil + 1$ reduces the complexity to $\Theta(1)$.

Actually, the papers [3, 4, 9, 10, 42, 43] consider a more detailed height bound of the form $\lceil \log(n+1) + \varepsilon \rceil$, where ε is any real number greater than zero. For ε less than one, this expression is optimal for the first integers n above $2^i - 1$ for any i, and optimal plus one for the last integers before $2^{i+1} - 1$. In other words, the smaller an ε, the closer to the next power of two is the height guaranteed to be optimal. Considering tangents to the graph of the logarithm function, it is easily seen that ε is proportional to the fraction of integers n for which the height is non-optimal.

Hence, an even more detailed formulation of the question about height bound versus rebalancing work is the following: Given a function f, what is the smallest possible ε such that the height bound $\lceil \log(n+1) + \varepsilon \rceil$ is maintainable with $O(f(n))$ rebalancing work per update?

In the case of amortized complexity, the answer is known. In [30], a lower bound is given, stating that no algorithm using $o(f(n))$ amortized rebuilding work per update can guarantee a height of $\lceil \log(n+1) + 1/f(n) \rceil$ for all n. The lower bound is proved by mapping trees to arrays and exploiting a fundamental lemma on density from [28]. In [31], a balancing scheme was given which maintains height $\lceil \log(n+1) + 1/f(n) \rceil$ in amortized $O(f(n))$ time per update, thereby matching the lower bound. The basic idea of the balancing scheme is similar to k-trees, but a more intricate distribution of unary nodes is used. Combined, these results show that for amortized complexity, the answer to the question above is

$$\varepsilon(n) \in \Theta(1/f(n)).$$

We may view this expression as describing the inherent amortized complexity of rebalancing a binary search tree, seen as a function of the height bound maintained. Using the observation above that for any i, $\lceil \log(n+1) + \varepsilon \rceil$ is equal to $\lceil \log(n+1) \rceil$ for n from $2^i - 1$ to $(1 - \Theta(\varepsilon))2^{i+1}$, the result may alternatively be viewed as the cost of maintaining optimal height when n approaches the next power of two: for $n = (1-\varepsilon)2^{i+1}$, the cost is $\Theta(1/\varepsilon)$. A graph depicting this cost appears in Figure 10.6.

This result holds for the fully dynamic case, where one may keep the size at $(1-\varepsilon)2^{i+1}$ by alternating between insertions and deletions. In the semi-dynamic case where only insertions take place, the amortized cost is smaller—essentially, it is the integral of the function in Figure 10.6, which gives $\Theta(n \log n)$ for n insertions, or $\Theta(\log n)$ per insertion.

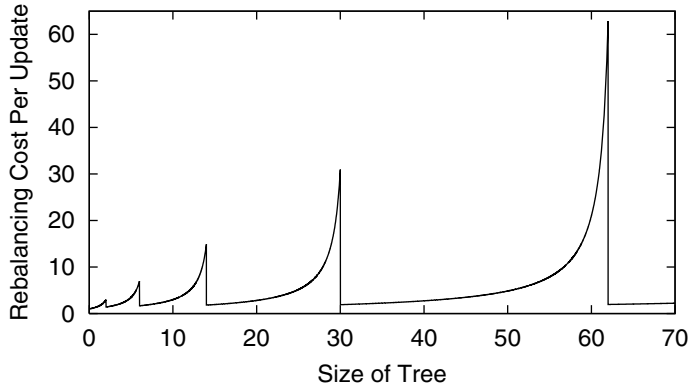

FIGURE 10.6: The cost of maintaining optimal height as a function of tree size.

More concretely, we may divide the insertions causing n to grow from 2^i to 2^{i+1} into i segments, where segment one is the first 2^{i-1} insertions, segment two is the next 2^{i-2} insertions, and so forth. In segment j, we employ the rebalancing scheme from [31] with $f(n) = \Theta(2^j)$, which will keep optimal height in that segment. The total cost of insertions is $O(2^i)$ inside each of the i segments, for a combined cost of $O(i2^i)$, which is $O(\log n)$ amortized per insertion. By the same reasoning, the lower bound from [30] implies that this is best possible for maintaining optimal height in the semi-dynamic case.

Considering worst case complexity for the fully dynamic case, the amortized lower bound stated above of course still applies. The best existing upper bound is height $\lceil \log(n+1) + \min\{1/\sqrt{f(n)}, \log(n)/f(n)\}\rceil$, maintained in $O(f(n))$ worst case time, by a combination of results in [4] and [30]. For the semi-dynamic case, a worst case cost of $\Theta(n)$ can be enforced when n reaches a power of two, as can be seen by the argument above on odd and even ranks of nodes in a completely full tree.

10.8 Relaxed Balance

In the classic search trees, including AVL-trees [1] and red-black trees [34], balancing is tightly coupled to updating. After an insertion or deletion, the updating procedure checks to see if the structural invariant is violated, and if it is, the problem is handled using the balancing operations before the next operation may be applied to the tree. This work is carried out in a bottom-up fashion by either solving the problem at its current location using rotations and/or adjustments of balance variables, or by carrying out a similar operation which moves the problem closer to the root, where, by design, all problems can be solved.

In relaxed balancing, the tight coupling between updating and balancing is removed. Basically, any restriction on when rebalancing is carried out and how much is done at a time is removed, except that the smallest unit of rebalancing is typically one single or double rotation. The immediate disadvantage is of course that the logarithmic height guarantee disappears, unless other methods are used to monitor the tree height.

The advantage gained is flexibility in the form of extra control over the combined process of updating and balancing. Balancing can be "turned off" during periods with frequent searching and updating (possibly from an external source). If there is not too much correlation between updates, the tree would likely remain fairly balanced during that time. When the frequency drops, more time can be spend on balancing. Furthermore, in multi-processor

environments, balancing immediately after an update is a problem because of the locking strategies with must be employed. Basically, the entire search path must be locked because it may be necessary to rebalance all the way back up to the root. This problem is discussed as early as in [34], where top-down balancing is suggested as a means of avoiding having to traverse the path again bottom-up after an update. However, this method generally leads to much more restructuring than necessary, up to $\Theta(\log n)$ instead of $O(1)$. Additionally, restructuring, especially in the form of a sequence of rotations, is generally significantly more time-consuming than adjustment of balance variables. Thus, it is worth considering alternative solutions to this concurrency control problem.

The advantages outlined above are only fully obtained if balancing is still efficient. That is the challenge: to define balancing constraints which are flexible enough that updating without immediate rebalancing can be allowed, yet at the same time sufficiently constrained that balancing can be handled efficiently at any later time, even if path lengths are constantly super-logarithmic.

The first partial result, dealing with insertions only, is from [41]. Below, we discuss the results which support insertion as well as deletion.

10.8.1 Red-Black Trees

In standard red-black trees, the balance constraints require that no two consecutive nodes are red and that for any node, every path to a leaf has the same number of black nodes. In the relaxed version, the first constraint is abandoned and the second is weakened in the following manner: Instead of a color variable, we use an integer variable, referred to as the *weight* of a node, in such a way that zero can be interpreted as red and one as black. The second constraint is then changed to saying that for any node, every path to a leaf has the same sum of weights. Thus, a standard red-black tree is also a relaxed tree; in fact, it is the ideal state of a relaxed tree. The work on red-black trees with relaxed balance was initiated in [64, 65].

Now, the updating operations must be defined so that an update can be performed in such a way that updating will leave the tree in a well-defined state, i.e., it must be a relaxed tree, without any subsequent rebalancing. This can be done as shown in Fig. 10.7. The operations are from [48].

The trees used here, and depicted in the figure, are assumed to be leaf-oriented. This terminology stems from applications where it is convenient to treat the external nodes differently from the remaining nodes. Thus, in these applications, the external nodes are not empty trees, but real nodes, possibly of another type than the internal nodes. In database applications, for instance, if a sequence of sorted data in the form of a linked list is already present, it is often desirable to build a tree on top of this data to facilitate faster searching. In such cases, it is often convenient to allow copies of keys from the leaves to also appear in the tree structure. To distinguish, we then refer to the key values in the leaves as keys, and refer to the key values in the tree structure as *routers*, since they merely guide the searching procedure. The ordering invariant is then relaxed, allowing keys in the left subtree of a tree rooted by u to be smaller than *or equal to* $u.k$, and the size of the tree is often defined as the number of leaves. When using the terminology outlined here, we refer to the trees as *leaf-oriented* trees.

The balance problems in a relaxed tree can now be specified as the relations between balance variables which prevent the tree from being a standard red-black tree, i.e., consecutive red nodes (nodes of weight zero) and weights greater than one. Thus, the balancing scheme must be targeted at removing these problems. It is an important feature of the design that the global constraint on a standard red-black tree involving the number of black nodes is

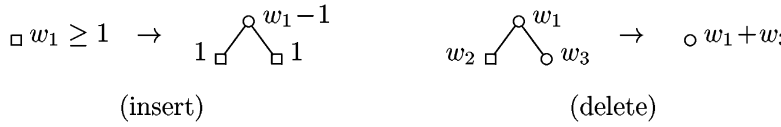

FIGURE 10.7: Update operations.

not lost after an update. Instead, the information is captured in the second requirement and as soon as all weight greater than one has been removed, the standard constraint holds again.

The strategy for the design of balancing operations is the same as for the classical search trees. Problems are removed if this is possible, and otherwise, the problem is moved closer to the root, where all problems can be resolved. In Fig. 10.8, examples are shown of how consecutive red nodes and weight greater than one can be eliminated, and in Fig. 10.9, examples are given of how these problems may be moved closer to the root, in the case where they cannot be eliminated immediately.

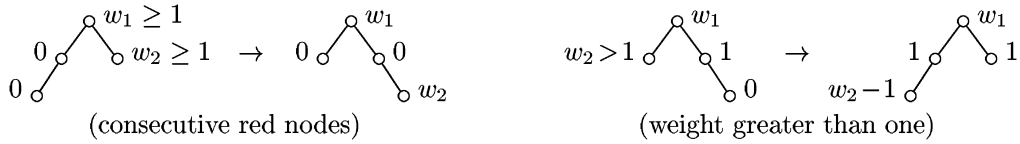

FIGURE 10.8: Example operations eliminating balance problems.

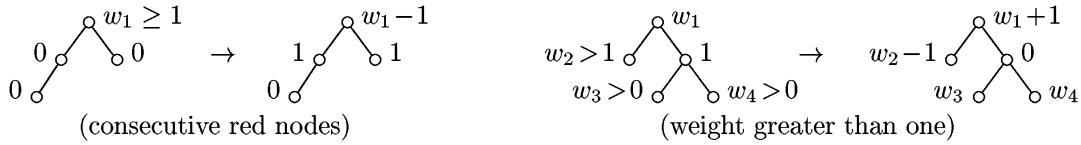

FIGURE 10.9: Example operations moving balance problems closer to the root.

It is possible to show complexity results for relaxed trees which are similar to the ones which can be obtained in the classical case. A logarithmic bound on the number of balancing operations required to balance the tree in response to an update was established in [23]. Since balancing operations can be delayed any amount of time, the usual notion of n as the number of elements in the tree at the time of balancing after an update is not really meaningful, so the bound is logarithmic in N, which is the maximum number of elements in the tree since it was last in balance. In [22], amortized constant bounds were obtained and in [45], a version is presented which has fewer and smaller operations, but meets the same bounds. Also, restructuring of the tree is worst-case constant per update. Finally, [48] extends the set of operations with a group insertion, such that an entire search tree

can be inserted in between two consecutive keys in amortized time $O(\log m)$, where m is the size of the subtree.

The amortized bounds as well as the worst case bounds are obtained using potential function techniques [74]. For group insertion, the results further depend on the fact that trees with low total potential can build [40], such that the inserted subtree does not increase the potential too dramatically.

10.8.2 AVL-Trees

The first relaxed version of AVL-trees [1] is from [63]. Here, the standard balance constraint of requiring that the heights of any two subtrees differ by at most one is relaxed by introducing a slack parameter, referred to as a *tag* value. The tag value, t_u, of any node u must be an integer greater than or equal to -1, except that the tag value of a leaf must be greater than or equal to zero. The constraint that heights may differ by at most one is then imposed on the *relaxed height* instead. The relaxed height $rh(u)$ of a node u is defined as

$$rh(u) = \begin{cases} t_u, & \text{if } u \text{ is a leaf} \\ \max(rh(u.l), rh(u.r)) + 1 + t_u, & \text{otherwise} \end{cases}$$

As for red-black trees, enough flexibility is introduced by this definition that updates can be made without immediate rebalancing while leaving the tree in a well-defined state. This can be done by adjusting tag values appropriately in the vicinity of the update location. A standard AVL-tree is the ideal state of a relaxed AVL-tree, which is obtained when all tag values are zero. Thus, a balancing scheme aiming at this is designed.

In [44], it is shown that a scheme can be designed such that the complexities from the sequential case are met. Thus, only a logarithmic number of balancing operations must be carried out in response to an update before the tree is again in balance. As opposed to red-black trees, the amortized constant rebalancing result does not hold in full generality for AVL-trees, but only for the semi-dynamic case [58]. This result is matched in [46].

A different AVL-based version was treated in [71]. Here, rotations are only performed if the subtrees are balanced. Thus, violations of the balance constraints must be dealt with bottom-up. This is a minimalistic approach to relaxed balance. When a rebalancing operation is carried out at a given node, the children do not violate the balance constraints. This limits the possible cases, and is asymptotically as efficient as the structure described above [52, 53].

10.8.3 Multi-Way Trees

Multi-way trees are usually described either as (a, b)-trees or B-trees, which are treated in another chapter of this book. An (a, b)-tree [37, 57] consists of nodes with at least a and at most b children. Usually, it is required that $a \geq 2$ to ensure logarithmic height, and in order to make the rebalancing scheme work, b must be at least $2a - 1$. Searching and updating including rebalancing is $O(\log_a n)$. If $b \geq 2a$, then rebalancing becomes amortized $O(1)$. The term B-trees [17] is often used synonymously, but sometimes refers to the variant where $b = 2a - 1$ or the variant where $b = 2a$.

For (a, b)-trees, the standard balance constraints for requiring that the number of children of each node is between a and b and that every leaf is at the same depth are relaxed as follows. First, nodes are allowed to have fewer than a children. This makes it possible to perform a deletion without immediate rebalancing. Second, nodes are equipped with a tag value, which is a non-positive integer value, and leaves are only required to have the same

relaxed depth, which is the usual depth, except that all tag values encountered from the root to the node in question are added. With this relaxation, it becomes possible to perform an insertion locally and leave the tree in a well-defined state.

Relaxed multi-way trees were first considered in [63], and complexity results matching the standard case were established in [50]. Variations with other properties can be found in [39]. Finally, a group insertion operation with a complexity of amortized $O(\log_a m)$, where m is the size of the group, can be added while maintaining the already achieved complexities for the other operations [47, 49]. The amortized result is a little stronger than usual, where it is normally assumed that the initial structure is empty. Here, except for very small values of a and b, zero-potential trees of any size can be constructed such the amortized results starting from such a tree hold immediately [40].

10.8.4 Other Results

Even though there are significant differences between the results outlined above, it is possible to establish a more general result giving the requirements for when a balanced search tree scheme can be modified to give a relaxed version with corresponding complexity properties [51]. The main requirements are that rebalancing operations in the standard scheme must be local constant-sized operations which are applied bottom-up, but in addition, balancing operation must also move the problems of imbalance towards the root. See [35] for an example of how these general ideas are expressed in the concrete setting of red-black trees.

In [32], it is demonstrated how the ideas of relaxed balance can be combined with methods from search trees of near-optimal height, and [39] contains complexity results made specifically for the reformulation of red-black trees in terms of layers based on black height from [67].

Finally, performance results from experiments with relaxed structures can be found in [21, 36].

References

[1] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organisation of information. *Doklady Akadamii Nauk SSSR*, 146:263–266, 1962. In Russian. English translation in *Soviet Math. Doklady*, 3:1259–1263, 1962.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, Massachusetts, 1983.

[3] A. Andersson. Optimal bounds on the dictionary problem. In *Proceeding of the Symposium on Optimal Algorithms*, volume 401 of *Lecture Notes in Computer Science*, pages 106–114. Springer-Verlag, 1989.

[4] A. Andersson. *Efficient Search Trees*. PhD thesis, Department of Computer Science, Lund University, Sweden, 1990.

[5] A. Andersson. Balanced search trees made simple. In *Proceedings of the Third Workshop on Algorithms and Data Structures*, volume 709 of *Lecture Notes in Computer Science*, pages 60–71. Springer-Verlag, 1993.

[6] A. Andersson. A demonstration of balanced trees. Algorithm animation program (for macintosh) including user's guide. Can be downloaded from author's homepage, 1993.

[7] A. Andersson. General balanced trees. *Journal of Algorithms*, 30:1–28, 1999.

[8] A. Andersson, C. Icking, R. Klein, and T. Ottmann. Binary search trees of almost optimal height. *Acta Informatica*, 28:165–178, 1990.

[9] A. Andersson and T. W. Lai. Fast updating of well-balanced trees. In *Proceedings of*

the Second Scandinavian Workshop on Algorithm Theory, volume 447 of *Lecture Notes in Computer Science*, pages 111–121. Springer-Verlag, 1990.

[10] A. Andersson and T. W. Lai. Comparison-efficient and write-optimal searching and sorting. In *Proceedings of the Second International Symposium on Algorithms*, volume 557 of *Lecture Notes in Computer Science*, pages 273–282. Springer-Verlag, 1991.

[11] A. Andersson and S. Nilsson. An efficient list implementation. In *JavaOne Conference*, 1999. Code can be found at Stefan Nilsson's home page.

[12] Arne Andersson. A note on searching in a binary search tree. *Software – Practice & Experience*, 21(10):1125–1128, 1991.

[13] Arne A. Andersson and Mikkel Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proceedings of the Thirty Second Annual ACM Symposium on Theory of Computing*, pages 335–342. ACM Press, 2000.

[14] Lars Arge and Jeffrey Scott Vitter. Optimal external memory interval management. *SIAM Journal on Computing*, 32(6):1488–1508, 2003.

[15] R. Bayer. Binary B-trees for virtual memory. In *Proceedings of the ACM SIGIFIDET Workshop on Data Description, Access and control*, pages 219–235, 1971.

[16] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, 1972.

[17] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[18] Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, 65(1):38–72, 2002.

[19] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[20] N. Blum and K. Mehlhorn. On the average number of rebalancing operations in weight-balanced trees. *Theoretical Computer Science*, 11:303–320, 1980.

[21] Luc Bougé, Joaquim Gabarró, Xavier Messeguer, and Nicolas Schabanel. Concurrent rebalancing of AVL trees: A fine-grained approach. In *Proceedings of the Third Annual European Conference on Parallel Processing*, volume 1300 of *Lecture Notes in Computer Science*, pages 421–429. Springer-Verlag, 1997.

[22] Joan Boyar, Rolf Fagerberg, and Kim S. Larsen. Amortization results for chromatic search trees, with an application to priority queues. *Journal of Computer and System Sciences*, 55(3):504–521, 1997.

[23] Joan F. Boyar and Kim S. Larsen. Efficient rebalancing of chromatic search trees. *Journal of Computer and System Sciences*, 49(3):667–682, 1994.

[24] M. R. Brown. A storage scheme for height-balanced trees. *Information Processing Letters*, 7(5):231–232, 1978.

[25] M. R. Brown. Addendum to "A storage scheme for height-balanced trees". *Information Processing Letters*, 8(3):154–156, 1979.

[26] H. Chang and S. S. Iynegar. Efficient algorithms to globally balance a binary search tree. *Communications of the ACM*, 27(7):695–702, 1984.

[27] A. C. Day. Balancing a binary tree. *Computer Journal*, 19(4):360–361, 1976.

[28] P. F. Dietz, J. I. Seiferas, and J. Zhang. A tight lower bound for on-line monotonic list labeling. In *Proceedings of the Fourth Scandinavian Workshop on Algorithm Theory*, volume 824 of *Lecture Notes in Computer Science*, pages 131–142. Springer-Verlag, 1994.

[29] P. F. Dietz and J. Zhang. Lower bounds for monotonic list labeling. In *Proceedings of the Second Scandinavian Workshop on Algorithm Theory*, volume 447 of *Lecture Notes in Computer Science*, pages 173–180. Springer-Verlag, 1990.

[30] Rolf Fagerberg. Binary search trees: How low can you go? In *Proceedings of the Fifth Scandinavian Workshop on Algorithm Theory*, volume 1097 of *Lecture Notes in Computer Science*, pages 428–439. Springer-Verlag, 1996.

[31] Rolf Fagerberg. The complexity of rebalancing a binary search tree. In *Proceedings of Foundations of Software Technology and Theoretical Computer Science*, volume 1738 of *Lecture Notes in Computer Science*, pages 72–83. Springer-Verlag, 1999.

[32] Rolf Fagerberg, Rune E. Jensen, and Kim S. Larsen. Search trees with relaxed balance and near-optimal height. In *Proceedings of the Seventh International Workshop on Algorithms and Data Structures*, volume 2125 of *Lecture Notes in Computer Science*, pages 414–425. Springer-Verlag, 2001.

[33] I Galperin and R. L. Rivest. Scapegoat trees. In *Proceedings of The Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 165–174, 1993.

[34] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual IEEE Symposium on the Foundations of Computer Science*, pages 8–21, 1978.

[35] S. Hanke, Th. Ottmann, and E. Soisalon-Soininen. Relaxed balanced red-black trees. In *Proceedings of the 3rd Italian Conference on Algorithms and Complexity*, volume 1203 of *Lecture Notes in Computer Science*, pages 193–204. Springer-Verlag, 1997.

[36] Sabina Hanke. The performance of concurrent red-black tree algorithms. In *Proceedings of the 3rd International Workshop on Algorithm Engineering*, volume 1668 of *Lecture Notes in Computer Science*, pages 286–300. Springer-Verlag, 1999.

[37] Scott Huddleston and Kurt Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

[38] Alon Itai, Alan G. Konheim, and Michael Rodeh. A sparse table implementation of priority queues. In *Proceedings of the 8th International Colloquium on Automata, Languages and Programming*, volume 115 of *Lecture Notes in Computer Science*, pages 417–431. Springer-Verlag, 1981.

[39] Lars Jacobsen and Kim S. Larsen. Complexity of layered binary search trees with relaxed balance. In *Proceedings of the Seventh Italian Conference on Theoretical Computer Science*, volume 2202 of *Lecture Notes in Computer Science*, pages 269–284. Springer-Verlag, 2001.

[40] Lars Jacobsen, Kim S. Larsen, and Morten N. Nielsen. On the existence and construction of non-extreme (a, b)-trees. *Information Processing Letters*, 84(2):69–73, 2002.

[41] J. L. W. Kessels. On-the-fly optimization of data structures. *Communications of the ACM*, 26:895–901, 1983.

[42] T. Lai. *Efficient Maintenance of Binary Search Trees*. PhD thesis, Department of Computer Science, University of Waterloo, Canada., 1990.

[43] T. Lai and D. Wood. Updating almost complete trees or one level makes all the difference. In *Proceedings of the Seventh Annual Symposium on Theoretical Aspects of Computer Science*, volume 415 of *Lecture Notes in Computer Science*, pages 188–194. Springer-Verlag, 1990.

[44] Kim S. Larsen. AVL trees with relaxed balance. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 888–893. IEEE Computer Society Press, 1994.

[45] Kim S. Larsen. Amortized constant relaxed rebalancing using standard rotations. *Acta Informatica*, 35(10):859–874, 1998.

[46] Kim S. Larsen. AVL trees with relaxed balance. *Journal of Computer and System Sciences*, 61(3):508–522, 2000.

[47] Kim S. Larsen. Relaxed multi-way trees with group updates. In *Proceedings of*

the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pages 93–101. ACM Press, 2001.

[48] Kim S. Larsen. Relaxed red-black trees with group updates. *Acta Informatica*, 38(8):565–586, 2002.

[49] Kim S. Larsen. Relaxed multi-way trees with group updates. *Journal of Computer and System Sciences*, 66(4):657–670, 2003.

[50] Kim S. Larsen and Rolf Fagerberg. Efficient rebalancing of b-trees with relaxed balance. *International Journal of Foundations of Computer Science*, 7(2):169–186, 1996.

[51] Kim S. Larsen, Thomas Ottmann, and E. Soisalon-Soininen. Relaxed balance for search trees with local rebalancing. *Acta Informatica*, 37(10):743–763, 2001.

[52] Kim S. Larsen, E. Soisalon-Soininen, and Peter Widmayer. Relaxed balance using standard rotations. *Algorithmica*, 31(4):501–512, 2001.

[53] Kim S. Larsen, Eljas Soisalon-Soininen, and Peter Widmayer. Relaxed balance through standard rotations. In *Proceedings of the Fifth International Workshop on Algorithms and Data Structures*, volume 1272 of *Lecture Notes in Computer Science*, pages 450–461. Springer-Verlag, 1997.

[54] W. A. Martin and D. N. Ness. Optimizing binary trees grown with a sorting algorithm. *Communications of the ACM*, 15(2):88–93, 1972.

[55] H. A. Maurer, T. Ottmann, and H.-W. Six. Implementing dictionaries using binary trees of very small height. *Information Processing Letters*, 5:11–14, 1976.

[56] E. M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, 1985.

[57] Kurt Mehlhorn. *Sorting and Searching*, volume 1 of *Data Structures and Algorithms*. Springer-Verlag, 1986.

[58] Kurt Mehlhorn and Athanasios K. Tsakalidis. An amortized analysis of insertions into AVL-trees. *SIAM Journal on Computing*, 15(1):22–33, 1986.

[59] J. I. Munro. An implicit data structure for the dictionary problem that runs in polylog time. In *Proceedings of the 25th Annual IEEE Symposium on the Foundations of Computer Science*, pages 369–374, 1984.

[60] J. I. Munro. An implicit data structure supporting insertion, deletion and search in $O(\log^2 n)$ time. *Journal of Computer and System Sciences*, 33:66–74, 1986.

[61] J. I. Munro and H. Suwanda. Implicit data structures for fast search and update. *Journal of Computer and System Sciences*, 21:236–250, 1980.

[62] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM Journal on Computing*, 2(1):33–43, 1973.

[63] O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrency control in database structures with relaxed balance. In *Proceedings of the 6th ACM Symposium on Principles of Database Systems*, pages 170–176, 1987.

[64] Otto Nurmi and Eljas Soisalon-Soininen. Uncoupling updating and rebalancing in chromatic binary search trees. In *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 192–198, 1991.

[65] Otto Nurmi and Eljas Soisalon-Soininen. Chromatic binary search trees—a structure for concurrent rebalancing. *Acta Informatica*, 33(6):547–557, 1996.

[66] H. J. Olivie. A new class of balanced search trees: Half-balanced binary search trees. *R. A. I. R. O. Informatique Theoretique*, 16:51–71, 1982.

[67] Th. Ottmann and E. Soisalon-Soininen. Relaxed balancing made simple. Technical Report 71, Institut für Informatik, Universität Freiburg, 1995.

[68] M. H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*. Springer-Verlag, 1983.

[69] Mark H. Overmars and Jan van Leeuwen. Dynamic multi-dimensional data structures

based on quad- and K-D trees. *Acta Informatica*, 17(3):267–285, 1982.

[70] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985.

[71] Eljas Soisalon-Soininen and Peter Widmayer. Relaxed balancing in search trees. In *Advances in Algorithms, Languages, and Complexity*, pages 267–283. Kluwer Academic Publishers, 1997.

[72] Q. F. Stout and B. L. Warren. Tree rebalancing in optimal time and space. *Communications of the ACM*, 29(9):902–908, 1986.

[73] R. E. Tarjan. Updating a balanced search tree in $O(1)$ rotations. *Information Processing Letters*, 16:253–257, 1983.

[74] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

[75] Athanasios K. Tsakalidis. Rebalancing operations for deletions in AVL-trees. *R.A.I.R.O. Informatique Théorique*, 19(4):323–329, 1985.

[76] J. van Leeuwen and M. H. Overmars. Stratified balanced search trees. *Acta Informatica*, 18:345–359, 1983.

[77] M. A. Weiss. *Data Structures and Algorithm Analysis*. The Benjamin/Cummings Publishing Company, 1992. Several versions during the years since the first version.

[78] Dan E. Willard. A density control algorithm for doing insertions and deletions in a sequentially ordered file in good worst-case time. *Information and Computation*, 97(2):150–204, 1992.

11

Finger Search Trees

Gerth Stølting Brodal
University of Aarhus

11.1 Finger Searching

One of the most studied problems in computer science is the problem of maintaining a sorted sequence of elements to facilitate efficient searches. The prominent solution to the problem is to organize the sorted sequence as a balanced search tree, enabling insertions, deletions and searches in logarithmic time. Many different search trees have been developed and studied intensively in the literature. A discussion of balanced binary search trees can be found in Chapter 10.

This chapter is devoted to *finger search trees*, which are search trees supporting *fingers*, i.e., pointers to elements in the search trees and supporting efficient updates and searches in the vicinity of the fingers.

If the sorted sequence is a *static* set of n elements then a simple and space efficient representation is a sorted array. Searches can be performed by binary search using $1+\lfloor \log n \rfloor$ comparisons (we throughout this chapter let $\log x$ to denote $\log_2 \max\{2, x\}$). A finger search starting at a particular element of the array can be performed by an *exponential search* by inspecting elements at distance $2^i - 1$ from the finger for increasing i followed by a binary search in a range of $2^{\lfloor \log d \rfloor} - 1$ elements, where d is the rank difference in the sequence between the finger and the search element. In Figure 11.1 is shown an exponential search for the element 42 starting at 5. In the example $d = 20$. An exponential search requires

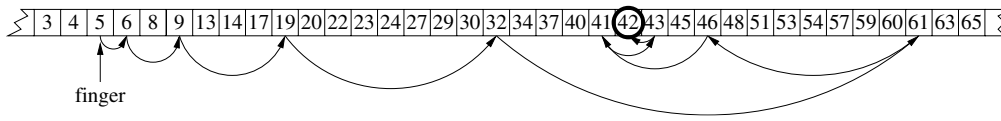

FIGURE 11.1: Exponential search for 42.

$2 + 2\lfloor \log d \rfloor$ comparisons.

Bentley and Yao [5] gave a close to optimal static finger search algorithm which performs $\sum_{i=1}^{\log^* d - 1} \log^{(i)} d + \mathcal{O}(\log^* d)$ comparisons, where $\log^{(1)} x = \log x$, $\log^{(i+1)} x = \log(\log^{(i)} x)$, and $\log^* x = \min\{i \mid \log^{(i)} x \leq 1\}$.

11.2 Dynamic Finger Search Trees

A dynamic finger search data structure should in addition to finger searches also support the insertion and deletion of elements at a position given by a finger. This section is devoted to an overview of existing dynamic finger search data structures. Section 11.3 and Section 11.4 give details concerning how three constructions support efficient finger searches: The level linked (2,4)-trees of Huddleston and Mehlhorn [26], the randomized skip lists of Pugh [36, 37] and the randomized binary search trees, treaps, of Seidel and Aragon [39].

Guibas et al. [21] introduced finger search trees as a variant of B-trees [4], supporting finger searches in $\mathcal{O}(\log d)$ time and updates in $\mathcal{O}(1)$ time, assuming that only $\mathcal{O}(1)$ movable fingers are maintained. Moving a finger d positions requires $\mathcal{O}(\log d)$ time. This work was refined by Huddleston and Mehlhorn [26]. Tsakalidis [42] presented a solution based on AVL-trees, and Kosaraju [29] presented a generalized solution. Tarjan and van Wyk [41] presented a solution based on red-black trees.

The above finger search tree constructions either assume a fixed constant number of fingers or only support updates in amortized constant time. Constructions supporting an arbitrary number of fingers and with worst case update have been developed. Levcopoulos and Overmars [30] presented a search tree that supported updates at an arbitrary position in worst case $\mathcal{O}(1)$ time, but only supports searches in $\mathcal{O}(\log n)$ time. Constructions supporting $\mathcal{O}(\log d)$ time searches and $\mathcal{O}(\log^* n)$ time insertions and deletions were developed by Harel [22, 23] and Fleischer [19]. Finger search trees with worst-case constant time insertions and $\mathcal{O}(\log^* n)$ time deletions were presented by Brodal [7], and a construction achieving optimal worst-case constant time insertions and deletions were presented by Brodal et al. [9].

Belloch et al. [6] developed a space efficient alternative solution to the level linked (2,4)-trees of Huddleston and Mehlhorn, see Section 11.3. Their solution allows a single finger, that can be moved by the same performance cost as (2,4)-trees. In the solution no level links and parent pointers are required, instead a special $\mathcal{O}(\log n)$ space data structure, *hand*, is created for the finger that allows the finger to be moved efficiently.

Sleator and Tarjan introduced *splay trees* as a class of self-adjusting binary search trees supporting searches, insertions and deletions in amortized $\mathcal{O}(\log n)$ time [40]. That splay trees can be used as efficient finger search trees was later proved by Cole [15, 16]: Given an $\mathcal{O}(n)$ initialization cost, the amortized cost of an access at distance d from the preceding access in a splay tree is $\mathcal{O}(\log d)$ where accesses include searches, insertions, and deletions. Notice that the statement only applies in the presence of one finger, which always points to the last accessed element.

All the above mentioned constructions can be implemented on a pointer machine where the only operation allowed on elements is the comparison of two elements. For the Random Access Machine model of computation (RAM), Dietz and Raman [17, 38] developed a finger search tree with constant update time and $\mathcal{O}(\log d)$ search time. This result is achieve by tabulating small tree structures, but only performs the comparison of elements. In the same model of computation, Andersson and Thorup [2] have surpassed the logarithmic bound in the search procedure by achieving $\mathcal{O}\left(\sqrt{\frac{\log d}{\log \log d}}\right)$ query time. This result is achieved by

considering elements as bit-patterns/machine words and applying techniques developed for the RAM to surpass lower bounds for comparison based data structures. A survey on RAM dictionaries can be found in Chapter 39.

11.3 Level Linked (2,4)-Trees

In this section we discuss how (2,4)-trees can support efficient finger searches by the introduction of *level links*. The ideas discussed in this section also applies to the more general class of height-balanced trees denoted (a, b)-trees, for $b \geq 2a$. A general discussion of height balanced search trees can be found in Chapter 10. A throughout treatment of level linked (a, b)-trees can be found in the work of Huddleston and Mehlhorn [26, 32].

A (2,4)-tree is a height-balanced search tree where all leaves have the same depth and all internal nodes have degree two, three or four. Elements are stored at the leaves, and internal nodes only store search keys to guide searches. Since each internal node has degree at least two, it follows that a (2,4)-tree has height $\mathcal{O}(\log n)$ and supports searches in $\mathcal{O}(\log n)$ time.

An important property of (2,4)-trees is that insertions and deletions given by a finger take amortized $\mathcal{O}(1)$ time (this property is not shared by $(2, 3)$-trees, where there exist sequences of n insertions and deletions requiring $\Theta(n \log n)$ time). Furthermore a (2,4)-tree with n leaves can be split into two trees of size n_1 and n_2 in amortized $\mathcal{O}(\log \min(n_1, n_2))$ time. Similarly two (2,4)-trees of size n_1 and n_2 can be joined (concatenated) in amortized $\mathcal{O}(\log \min(n_1, n_2))$ time.

To support finger searches (2,4)-trees are augmented with level links, such that all nodes with equal depth are linked together in a double linked list. Figure 11.2 shows a (2,4)-tree augmented with level links. Note that all edges represent bidirected links. The additional level links are straightforward to maintain during insertions, deletions, splits and joins of (2,4)-trees.

To perform a finger search from x to y we first check whether y is to the left or right of x. Assume without loss of generality that y is to the right of x. We then traverse the path from x towards the root while examining the nodes v on the path and their right neighbors until it has been established that y is contained within the subtree rooted at v or v's right neighbor. The upwards search is then terminated and at most two downwards searches for y is started at respectively v and/or v's right neighbor. In Figure 11.2 the pointers followed during a finger search from J to T are depicted by thick lines.

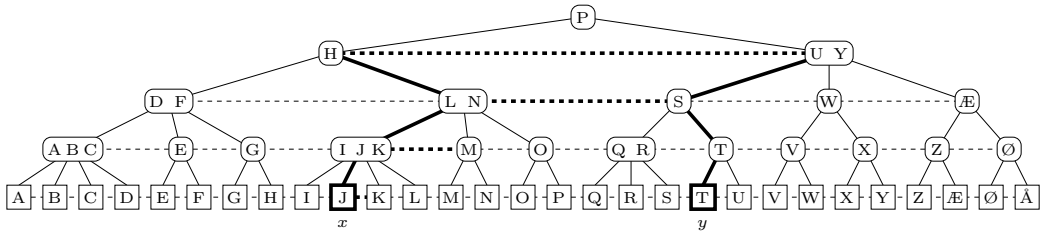

FIGURE 11.2: Level linked (2,4)-trees.

The $\mathcal{O}(\log d)$ search time follows from the observation that if we advance the upwards search to the parent of node v then y is to the right of the leftmost subtree of $v's$ right

neighbor, i.e. d is at least exponential in the height reached so far. In Figure 11.2 we advance from the internal node labeled "L N" to the node labeled "H" because from "S" we know that y is to the right of the subtree rooted at the node "Q R".

The construction for level linked (2,4)-trees generalizes directly to level linked (a, b)-trees that can be used in external memory. By choosing $a = 2b$ and b such that an internal node fits in a block in external memory, we achieve *external memory* finger search trees supporting insertions and deletions in $\mathcal{O}(1)$ memory transfers, and finger searches with $\mathcal{O}(\log_b n)$ memory transfers.

11.4 Randomized Finger Search Trees

Two randomized alternatives to deterministic search trees are the randomized binary search trees, *treaps*, of Seidel and Aragon [39] and the *skip lists* of Pugh [36, 37]. Both treaps and skip lists are elegant data structures, where the randomization facilitates simple and efficient update operations.

In this section we describe how both treaps and skip lists can be used as efficient finger search trees without altering the data structures. Both data structures support finger searches in *expected* $\mathcal{O}(\log d)$ time, where the expectations are taken over the random choices made by the algorithm during the construction of the data structure. For a general introduction to randomized dictionary data structures see Chapter 13.

11.4.1 Treaps

A treap is a rooted binary tree where each node stores an element and where each element has an associated random priority. A treap satisfies that the elements are sorted with respect to an inorder traversal of tree, and that the priorities of the elements satisfy heap order, i.e., the priority stored at a node is always smaller than or equal to the priority stored at the parent node. Provided that the priorities are distinct, the shape of a treap is uniquely determined by its set of elements and the associated priorities. Figure 11.3 shows a treap storing the elements A,B,...,T and with random integer priorities between one and hundred.

The most prominent properties of treaps are that they have expected $\mathcal{O}(\log n)$ height, implying that they provide searches in expected $\mathcal{O}(\log n)$ time. Insertions and deletions of elements can be performed in expected at most two rotations and expected $\mathcal{O}(1)$ time, provided that the position of insertion or deletion is known, i.e. insertions and deletions given by a finger take expected $\mathcal{O}(1)$ time [39].

The essential property of treaps enabling expected $\mathcal{O}(\log d)$ finger searches is that for two elements x and y whose ranks differ by d in the set stored, the expected length of the path between x and y in the treap is $\mathcal{O}(\log d)$. To perform a finger search for y starting with a finger at x, we ideally start at x and traverse the ancestor path of x until we reach the *least common ancestor* of x and y, $\mathrm{LCA}(x, y)$, and start a downward tree search for y. If we can decide if a node is $\mathrm{LCA}(x, y)$, this will traverse exactly the path from x to y. Unfortunately, it is nontrivial to decide if a node is $\mathrm{LCA}(x, y)$. In [39] it is assumed that a treap is extended with additional pointers to facilitate finger searches in expected $\mathcal{O}(\log d)$ time. Below an alternative solution is described not requiring any additional pointers than the standard left, right and parent pointers.

Assume without loss of generality that we have a finger at x and have to perform a finger search for $y \geq x$ present in the tree. We start at x and start traversing the ancestor path of x. During this traversal we keep a pointer ℓ to the last visited node that can potentially

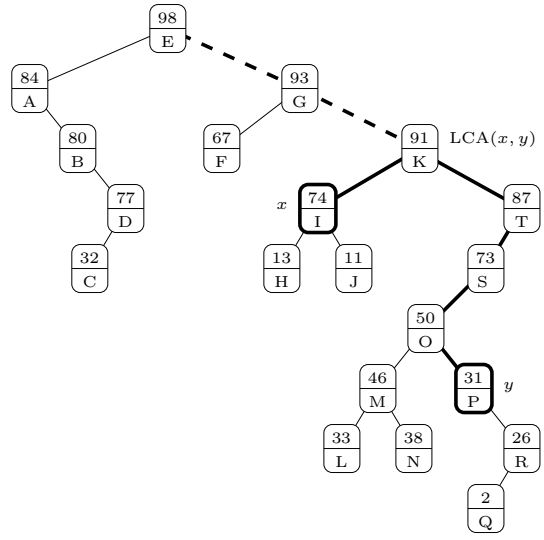

FIGURE 11.3: Performing finger searches on treaps.

be LCA(x, y). Whenever we visit a node v on the path from x to the root there are three cases:

(1) $v \leq x$, then x is in the right subtree of v and cannot be LCA(x, y); we advance to the parent of v.

(2) $x < v \leq y$, then x is in the left subtree of v and LCA(x, y) is either y or an ancestor of y; we reset $\ell = v$ and advance to the parent of v.

(3) $x < y < v$, then LCA(x, y) is in the left subtree of v and equals ℓ.

Unfortunately, after LCA(x, y) has been visited case (1) can happen $\omega(\log d)$ times before the search is terminated at the root or by case (3). Seidel and Aragon [39] denote these extra nodes visited above LCA(x, y) the *excess path* of the search, and circumvent this problem by extending treaps with special pointers for this.

To avoid visiting long excess paths we extend the above upward search with a concurrent downward search for y in the subtree rooted at the current candidate ℓ for LCA(x, y). In case (1) we always advance the tree search for y one level down, in case (2) we restart the search at the new ℓ, and in (3) we finalize the search. The concurrent search for y guarantees that the distance between LCA(x, y) and y in the tree is also an upper bound on the nodes visited on the excess path, i.e. we visit at most twice the number of nodes as is on the path between x and y, which is expected $\mathcal{O}(\log d)$. It follows that treaps support finger searches in $\mathcal{O}(\log d)$ time. In Figure 11.3 is shown the search for $x = I$, $y = P$, LCA$(x, y) = K$, the path from x to y is drawn with thick lines, and the excess path is drawn with dashed lines.

11.4.2 Skip Lists

A skip list is a randomized dictionary data structure, which can be considered to consists of expected $\mathcal{O}(\log n)$ levels. The lowest level being a single linked list containing the elements in sorted order, and each succeeding level is a random sample of the elements of the previous level, where each element is included in the next level with a fixed probability, e.g. 1/2. The

pointer representation of a skip is illustrated in Figure 11.4.

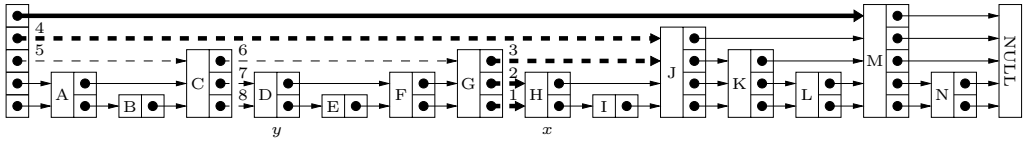

FIGURE 11.4: Performing finger searches on skip list.

The most prominent properties of skip lists are that they require expected linear space, consist of expected $\mathcal{O}(\log n)$ levels, support searches in expected $\mathcal{O}(\log n)$ time, and support insertions and deletions at a given position in expected $\mathcal{O}(1)$ time [36, 37].

Pugh in [36] elaborates on the various properties and extensions of skip lists, including pseudo-code for how skip lists support finger searches in expected $\mathcal{O}(\log d)$ time. To facilitate backward finger searches, a finger to a node v is stored as an expected $\mathcal{O}(\log n)$ space finger data structure that for each level i stores a pointer to the node to the left of v where the level i pointer either points to v or a node to the right of v. Moving a finger requires this list of pointers to be updated correspondingly.

A backward finger search is performed by first identifying the lowest node in the finger data structure that is to the left of the search key y, where the nodes in the finger data structure are considered in order of increasing levels. Thereafter the search proceeds downward from the identified node as in a standard skip list search.

Figure 11.4 shows the situation where we have a finger to H, represented by the thick (solid or dashed) lines, and perform a finger search for the element D to the left of H. Dashed (thick and thin) lines are the pointers followed during the finger search. The numbering indicate the other in which the pointers are traversed.

If the level links of a skip list are maintained as double-linked lists, then finger searches can be performed in expected $\mathcal{O}(\log d)$ time by traversing the existing links, without having a separate $\mathcal{O}(\log n)$ space finger data structure

11.5 Applications

Finger search trees have, e.g., been used in algorithms within computational geometry [3, 8, 20, 24, 28, 41] and string algorithms [10, 11]. In the rest of this chapter we give examples of the efficiency that can be obtained by applying finger search trees. These examples typically allow one to save a factor of $\mathcal{O}(\log n)$ in the running time of algorithms compared to using standard balanced search trees supporting $\mathcal{O}(\log n)$ time searches.

11.5.1 Optimal Merging and Set Operations

Consider the problem of merging two sorted sequences X and Y of length respectively n and m, where $n \leq m$, into one sorted sequence of length $n + m$. The canonical solution is to repeatedly insert each $x \in X$ in Y. This requires that Y is searchable and that there can be inserted new elements, i.e. a suitable representation of Y is a balanced search tree. This immediately implies an $\mathcal{O}(n \log m)$ time bound for merging. In the following we discuss how finger search trees allow this bound to be improved to $\mathcal{O}(n \log \frac{m}{n})$.

Hwang and Lin [27] presented an algorithm for merging two sorted sequence using optimal $\mathcal{O}(n \log \frac{m}{n})$ comparisons, but did not discuss how to represent the sets. Brown and Tarjan [12] described how to achieve the same bound for merging two AVL trees [1]. Brown and Tarjan subsequently introduced *level linked* (2,3)-trees and described how to achieve the same merging bound for level linked (2,3)-trees [13].

Optimal merging of two sets also follows as an application of finger search trees [26]. Assume that the two sequences are represented as finger search trees, and that we repeatedly insert the n elements from the shorter sequence into the larger sequence using a finger that moves monotonically from left to right. If the ith insertion advances the finger d_i positions, we have that the total work of performing the n finger searches and insertions is $\mathcal{O}(\sum_{i=1}^{n} \log d_i)$, where $\sum_{i=1}^{n} d_i \leq m$. By convexity of the logarithm the total work becomes bounded by $\mathcal{O}(n \log \frac{m}{n})$.

Since sets can be represented as sorted sequences, the above merging algorithm gives immediately raise to optimal, i.e. $\mathcal{O}\left(\log \binom{n+m}{n}\right) = \mathcal{O}(n \log \frac{m}{n})$ time, algorithms for set union, intersection, and difference operations [26]. For a survey of data structures for set representations see Chapter 33.

11.5.2 Arbitrary Merging Order

A classical $\mathcal{O}(n \log n)$ time sorting algorithm is binary merge sort. The algorithm can be viewed as the merging process described by a balanced binary tree: Each leaf corresponds to an input element and each internal node corresponds to the merging of the two sorted sequences containing respectively the elements in the left and right subtree of the node. If the tree is balanced then each element participates in $\mathcal{O}(\log n)$ merging steps, i.e. the $\mathcal{O}(n \log n)$ sorting time follows.

Many divide-and-conquer algorithms proceed as binary merge sort, in the sense that the work performed by the algorithm can be characterized by a treewise merging process. For some of these algorithms the tree determining the merges is unfortunately fixed by the input instance, and the running time using linear merges becomes $\mathcal{O}(n \cdot h)$, where h is the height of the tree. In the following we discuss how finger search trees allow us to achieve $\mathcal{O}(n \log n)$ for unbalanced merging orders to.

Consider an arbitrary binary tree $\mathcal{T}$ with n leaves, where each leaf stores an element. We allow $\mathcal{T}$ to be arbitrarily unbalanced and that elements are allowed to appear at the leaves in any arbitrary order. Associate to each node v of $\mathcal{T}$ the set $\mathcal{S}_v$ of elements stored at the leaves of the subtree rooted at v. If we for each node v of $\mathcal{T}$ compute $\mathcal{S}_v$ by merging the two sets of the children of v using finger search trees, cf. Section 11.5.1, then the total time to compute all the sets $\mathcal{S}_v$ is $\mathcal{O}(n \log n)$.

The proof of the total $\mathcal{O}(n \log n)$ bound is by structural induction where we show that in a tree of size n, the total merging cost is $\mathcal{O}(\log(n!)) = \mathcal{O}(n \log n)$. Recall that two sets of size n_1 and n_2 can be merged in $\mathcal{O}\left(\log \binom{n_1+n_2}{n_1}\right)$ time. By induction we get that the total merging in a subtree with a root with two children of size respectively n_1 and n_2 becomes:

$$\begin{aligned} & \log(n_1!) + \log(n_2!) + \log \binom{n_1+n_2}{n_1} \\ &= \log(n_1!) + \log(n_2!) + \log((n_1+n_2)!) - \log(n_1!) - \log(n_2!) \\ &= \log((n_1+n_2)!) \ . \end{aligned}$$

The above approach of arbitrary merging order was applied in [10, 11] to achieve $\mathcal{O}(n \log n)$ time algorithms for finding repeats with gaps and quasiperiodicities in strings. In both these

algorithms $\mathcal{T}$ is determined by the suffix-tree of the input string, and the S_v sets denote the set of occurrences (positions) of the substring corresponding to the path label of v.

11.5.3 List Splitting

Hoffmann et al. [25] considered how finger search trees can be used for solving the following *list splitting* problem, that e.g. also is applied in [8, 28]. Assume we initially have a sorted list of n elements that is repeatedly split into two sequences until we end up with n sequences each containing one element. If the splitting of a list of length k into two lists of length k_1 and k_2 is performed by performing a simultaneous finger search from each end of the list, followed by a split, the searching and splitting can be performed in $\mathcal{O}(\log \min(k_1, k_2))$ time. Here we assume that the splitting order is unknown in advance.

By assigning a list of k elements a potential of $k - \log k \geq 0$, the splitting into two lists of size k_1 and k_2 releases the following amount of potential:

$$\begin{aligned}(k - \log k) - (k_1 - \log k_1) - (k_2 - \log k_2) \\ = \quad -\log k + \log \min(k_1, k_2) + \log \max(k_1, k_2) \\ \geq \quad -1 + \log \min(k_1, k_2)\ ,\end{aligned}$$

since $\max(k_1, k_2) \geq k/2$. The released potential allows each list splitting to be performed in amortized $\mathcal{O}(1)$ time. The initial list is charged $n - \log n$ potential. We conclude that starting with a list of n elements, followed by a sequence of at most $n - 1$ splits requires total $\mathcal{O}(n)$ time.

11.5.4 Adaptive Merging and Sorting

The area of *adaptive sorting* addresses the problem of developing sorting algorithms which perform $o(n \log n)$ comparisons for inputs with a limited amount of disorder for various definitions of measures of disorder, e.g. the measure Inv counts the number of pairwise insertions in the input. For a survey of adaptive sorting algorithms see [18].

An adaptive sorting algorithm that is optimal with respect to the disorder measure Inv has running time $\mathcal{O}(n \log \frac{\text{Inv}}{n})$. A simple adaptive sorting algorithm optimal with respect to Inv is the *insertion sort* algorithm, where we insert the elements of the input sequence from left to right into a finger search tree. Insertions always start at a finger on the last element inserted. Details on applying finger search trees in insertion sort can be found in [13, 31, 32].

Another adaptive sorting algorithm based on applying finger search trees is obtained by replacing the linear merging in binary merge sort by an *adaptive merging* algorithm [14, 33–35]. The classical binary merge sort algorithm alway performs $\Omega(n \log n)$ comparisons, since in each merging step where two lists each of size k is merged the number of comparisons performed is between k and $2k - 1$.

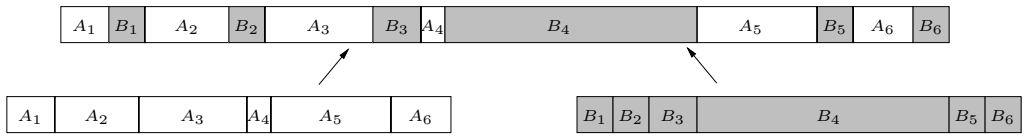

FIGURE 11.5: Adaptive merging.

The idea of the adaptive merging algorithm is to identify consecutive blocks from the input sequences which are also consecutive in the output sequence, as illustrated in Figure 11.5. This is done by repeatedly performing a finger search for the smallest element of the two input sequences in the other sequence and deleting the identified block in the other sequence by a split operation. If the blocks in the output sequence are denoted $Z_1, \ldots, Z_k$, it follows from the time bounds of finger search trees that the total time for this adaptive merging operation becomes $\mathcal{O}(\sum_{i=1}^{k} \log |Z_i|)$. From this merging bound it can be argued that merge sort with adaptive merging is adaptive with respect to the disorder measure Inv (and several other disorder measures). See [14, 33, 34] for further details.

Acknowledgment

This work was supported by the Carlsberg Foundation (contract number ANS-0257/20), BRICS (Basic Research in Computer Science, www.brics.dk, funded by the Danish National Research Foundation), and the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

References

[1] G. M. Adel'son-Vel'skii and Y. M. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk SSSR*, 146:263–266, 1962. English translation in *Soviet Math. Dokl.*, 3:1259–1262.

[2] A. Anderson and M. Thorup. Tight(er) worst case bounds on dynamic searching and priority queues. In *Proc. 32nd Annual ACM Symposium On Theory of Computing*, pages 335–342, 2000.

[3] M. Atallah, M. Goodrich, and K.Ramaiyer. Biased finger trees and three-dimensional layers of maxima. In *Proc. 10th ACM Symposium on Computational Geometry*, pages 150–159, 1994.

[4] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[5] J. L. Bentley and A. C.-C. Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–87, 1976.

[6] G. E. Blelloch, B. M. Maggs, and S. L. M. Woo. Space-efficient finger search on degree-balanced search trees. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 374–383. Society for Industrial and Applied Mathematics, 2003.

[7] G. S. Brodal. Finger search trees with constant insertion time. In *Proc. 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 540–549, 1998.

[8] G. S. Brodal and R. Jacob. Dynamic planar convex hull. In *Proc. 43rd Annual Symposium on Foundations of Computer Science*, pages 617–626, 2002.

[9] G. S. Brodal, G. Lagogiannis, C. Makris, A. Tsakalidis, and K. Tsichlas. Optimal finger search trees in the pointer machine. *Journal of Computer and System Sciences, Special issue on STOC 2002*, 67(2):381–418, 2003.

[10] G. S. Brodal, R. B. Lyngsø, C. N. S. Pedersen, and J. Stoye. Finding maximal pairs with bounded gap. *Journal of Discrete Algorithms, Special Issue of Matching Patterns*, 1(1):77–104, 2000.

[11] G. S. Brodal and C. N. S. Pedersen. Finding maximal quasiperiodicities in strings. In *Proc. 11th Annual Symposium on Combinatorial Pattern Matching*, volume 1848 of *Lecture Notes in Computer Science*, pages 397–411. Springer-Verlag, 2000.

[12] M. R. Brown and R. E. Tarjan. A fast merging algorithm. *Journal of the ACM*, 26(2):211–226, 1979.

[13] M. R. Brown and R. E. Tarjan. Design and analysis of a data structure for representing sorted lists. *SIAM Journal of Computing*, 9:594–614, 1980.

[14] S. Carlsson, C. Levcopoulos, and O. Petersson. Sublinear merging and natural mergesort. *Algorithmica*, 9(6):629–648, 1993.

[15] R. Cole. On the dynamic finger conjecture for splay trees. part II: The proof. *SIAM Journal of Computing*, 30(1):44–85, 2000.

[16] R. Cole, B. Mishra, J. Schmidt, and A. Siegel. On the dynamic finger conjecture for splay trees. part I: Splay sorting $\log n$-block sequences. *SIAM Journal of Computing*, 30(1):1–43, 2000.

[17] P. F. Dietz and R. Raman. A constant update time finger search tree. *Information Processing Letters*, 52:147–154, 1994.

[18] V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys*, 24:441–476, 1992.

[19] R. Fleischer. A simple balanced search tree with $O(1)$ worst-case update time. *International Journal of Foundations of Computer Science*, 7:137–149, 1996.

[20] L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. Tarjan. Linear time algorithms for visibility and shortest path problems inside simple polygons. *Algorithmica*, 2:209–233, 1987.

[21] L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts. A new representation for linear lists. In *Proc. 9th Ann. ACM Symp. on Theory of Computing*, pages 49–60, 1977.

[22] D. Harel. Fast updates of balanced search trees with a guaranteed time bound per update. Technical Report 154, University of California, Irvine, 1980.

[23] D. Harel and G. S. Lueker. A data structure with movable fingers and deletions. Technical Report 145, University of California, Irvine, 1979.

[24] J. Hershberger. Finding the visibility graph of a simple polygon in time proportional to its size. In *Proc. 3rd ACM Symposium on Computational Geometry*, pages 11–20, 1987.

[25] K. Hoffmann, K. Mehlhorn, P. Rosenstiehl, and R. E. Tarjan. Sorting Jordan sequences in linear time using level/linked search trees. *Information and Control*, 68(1-3):170–184, 1986.

[26] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

[27] F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly ordered sets. *SIAM Journal of Computing*, 1(1):31–39, 1972.

[28] R. Jacob. *Dynamic Planar Convex Hull*. PhD thesis, University of Aarhus, Denmark, 2002.

[29] S. R. Kosaraju. Localized search in sorted lists. In *Proc. 13th Ann. ACM Symp. on Theory of Computing*, pages 62–69, 1981.

[30] C. Levcopoulos and M. H. Overmars. A balanced search tree with $O(1)$ worst-case update time. *Acta Informatica*, 26:269–277, 1988.

[31] H. Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Transactions on Computers*, C-34:318–325, 1985.

[32] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. Springer-Verlag, 1984.

[33] A. Moffat. Adaptive merging and a naturally natural merge sort. In *Proceedings of the 14th Australian Computer Science Conference*, pages 08.1–08.8, 1991.

[34] A. Moffat, O. Petersson, and N. Wormald. Further analysis of an adaptive sorting

algorithm. In *Proceedings of the 15th Australian Computer Science Conference*, pages 603–613, 1992.

[35] A. Moffat, O. Petersson, and N. C. Wormald. Sorting and/by merging finger trees. In *Algorithms and Computation: Third International Symposium, ISAAC '92*, volume 650 of *Lecture Notes in Computer Science*, pages 499–508. Springer-Verlag, 1992.

[36] W. Pugh. A skip list cookbook. Technical Report CS-TR-2286.1, Dept. of Computer Science, University of Maryland, College Park, 1989.

[37] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[38] R. Raman. *Eliminating Amortization: On Data Structures with Guaranteed Response Time*. PhD thesis, University of Rochester, New York, 1992. Computer Science Dept., U. Rochester, tech report TR-439.

[39] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996.

[40] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[41] R. Tarjan and C. van Wyk. An $o(n \log \log n)$ algorithm for triangulating a simple polygon. *SIAM Journal of Computing*, 17:143–178, 1988.

[42] A. K. Tsakalidis. AVL-trees for localized search. *Information and Control*, 67(1-3):173–194, 1985.

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

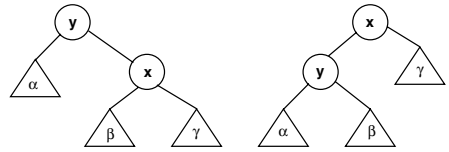

FIGURE 12.1: parent(x) is the root— edge xy is rotated (Zig case).

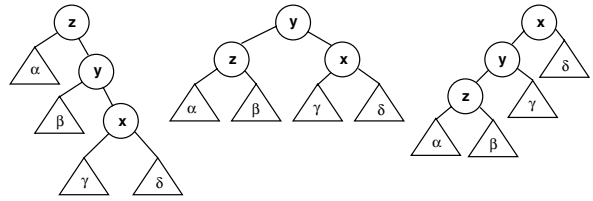

FIGURE 12.2: x and parent(x) are both right children (Zig-Zig case) —first edge yz is rotated then edge xy.

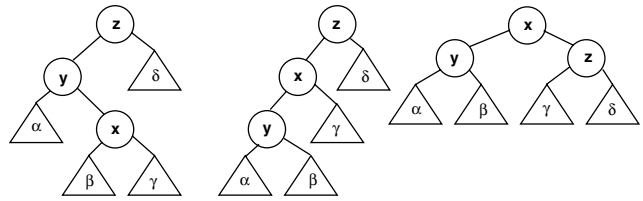

FIGURE 12.3: x is a right child while parent(x) is a left child (Zig-Zag case)— first edge xy is rotated then edge xz.

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

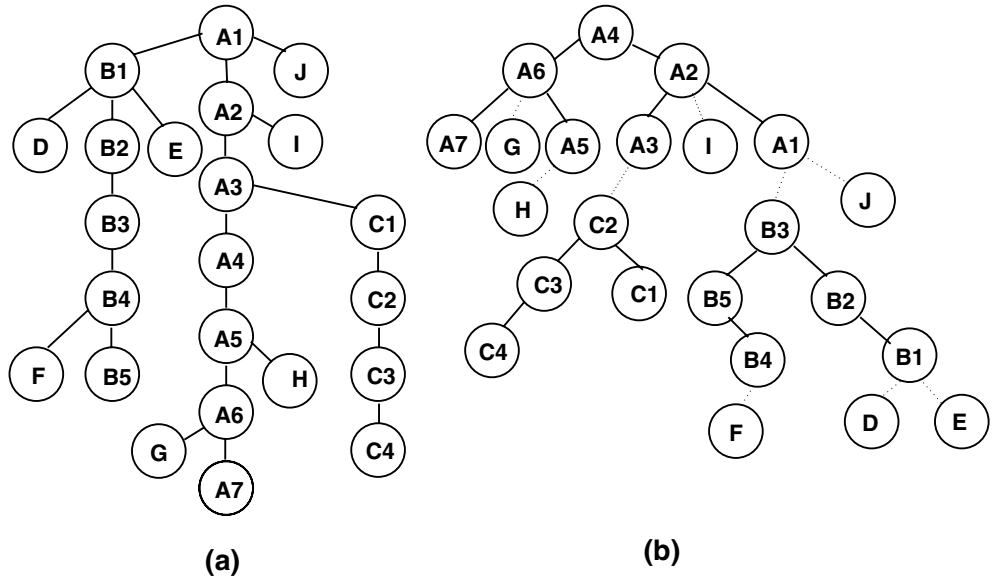

FIGURE 12.4: (a) Original Tree (b) Virtual Trees: Solid and dashed children.

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,

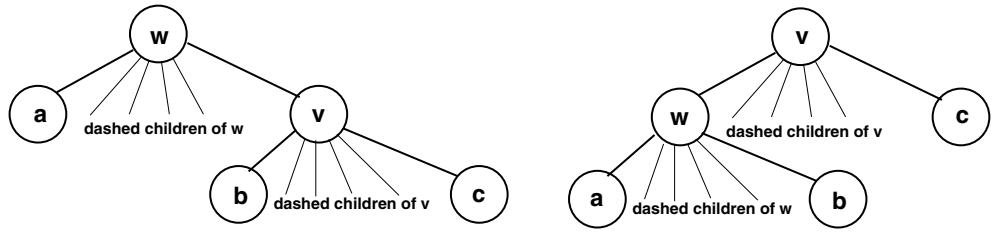

FIGURE 12.5: Rotation in Solid Trees— rotation of edge (v, w).

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

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 $t \in \overline{S}$

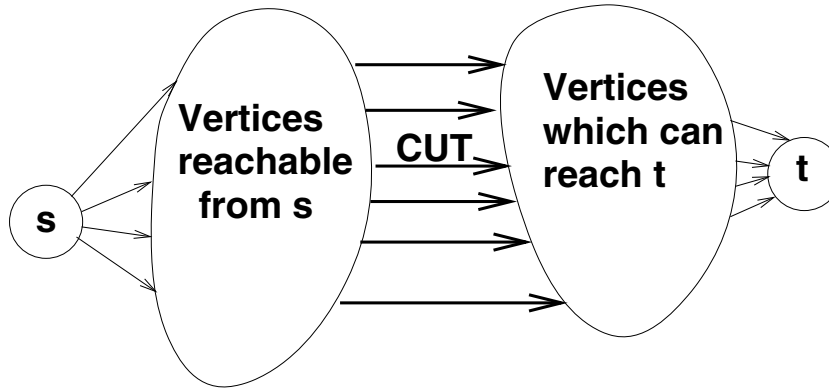

FIGURE 12.6: $s - t$ Cut.

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

design goals. In one of such design goals, we seek to minimize the total cost of a sequence of operations as opposed to the cost of individual operations. Such data structures are said to be designed for minimizing the amortized costs of operations. Randomization provides yet another avenue for exploration. Here, the goal will be to limit the expected costs of operations and ensure that costs do not exceed certain threshold limits with overwhelming probability.

In this chapter we discuss about the *Dictionary* ADT which deals with sets whose elements are drawn from a fixed universe U and supports operations such as *insert*, *delete* and *search*. Formally, we assume a linearly ordered universal set U and for the sake of concreteness we assume U to be the set of all integers. At any time of computation, the *Dictionary* deals only with a finite subset of U. We shall further make a simplifying assumption that we deal only with sets with distinct values. That is, we never handle a multiset in our structure, though, with minor modifications, our structures can be adjusted to handle multisets containing multiple copies of some elements. With these remarks, we are ready for the specification of the *Dictionary* ADT.

DEFINITION 13.1 [Dictionary ADT] Let U be a linearly ordered universal set and S denote a finite subset of U. The Dictionary ADT, defined on the class of finite subsets of U, supports the following operations.
Insert (x, S) : For an $x \in U, S \subset U$, generate the set $S \bigcup \{x\}$.
Delete (x, S) : For an $x \in U, S \subset U$, generate the set $S - \{x\}$.
Search (x, S) : For an $x \in U, S \subset U$, return TRUE if $x \in S$ and return FALSE if $x \notin S$.

Remark : When the universal set is evident in a context, we will not explicitly mention it in the discussions. Notice that we work with sets and not multisets. Thus, *Insert (x,S)* does not produce new set when x is in the set already. Similarly *Delete (x, S)* does not produce a new set when $x \notin S$.

Due to its fundamental importance in a host of applications ranging from compiler design to data bases, extensive studies have been done in the design of data structures for dictionaries. Refer to Chapters 3 and 10 for data structures for dictionaries designed with the worst-case costs in mind, and Chapter 12 of this handbook for a data structure designed with amortized cost in mind. In Chapter 15 of this book, you will find an account of *B-Trees* which aim to minimize the disk access. All these structures, however, are deterministic. In this sequel, we discuss two of the interesting randomized data structures for Dictionaries. Specifically

- We describe a data structure called *Skip Lists* and present a comprehensive probabilistic analysis of its performance.
- We discuss an interesting randomized variation of a search tree called *Randomized Binary Search Tree* and compare and contrast the same with other competing structures.

13.2 Preliminaries

In this section we collect some basic definitions, concepts and the results on randomized computations and probability theory. We have collected only the materials needed for the topics discussed in this chapter. For a more comprehensive treatment of randomized algorithms, refer to the book by Motwani and Raghavan [9].

13.2.1 Randomized Algorithms

Every computational step in an execution of a *deterministic algorithm* is uniquely determined by the set of all steps executed prior to this step. However, in a *randomized algorithm*, the choice of the next step may not be entirely determined by steps executed previously; the choice of next step might depend on the outcome of a random number generator. Thus, several execution sequences are possible even for the same input. Specifically, when a randomized algorithm is executed several times, even on the same input, the running time may vary from one execution to another. In fact, the running time is a random variable depending on the random choices made during the execution of the algorithm. When the running time of an algorithm is a random variable, the traditional worst case complexity measure becomes inappropriate. In fact, the quality of a randomized algorithm is judged from the statistical properties of the random variable representing the running time. Specifically, we might ask for bounds for the expected running time and bounds beyond which the running time may exceed only with negligible probability. In other words, for the randomized algorithms, there is no bad input; we may perhaps have an *unlucky* execution.
The type of randomized algorithms that we discuss in this chapter is called *Las Vegas* type algorithms. A *Las Vegas* algorithm always terminates with a correct answer although the running time may be a random variable exhibiting wide variations. There is another important class of randomized algorithms, called *Monte Carlo* algorithms, which have fixed running time but the output may be erroneous. We will not deal with *Monte Carlo* algorithms as they are not really appropriate for basic building blocks such as data structures. We shall now define the notion of efficiency and complexity measures for *Las Vegas* type randomized algorithms.

Since the running time of a *Las Vegas* randomized algorithm on any given input is a random variable, besides determining the expected running time it is desirable to show that the running time does not exceed certain threshold value with very high probability. Such threshold values are called *high probability bounds* or *high confidence bounds*. As is customary in algorithmics, we express the estimation of the expected bound or the high-probability bound as a function of the size of the input. We interpret an execution of a *Las Vegas* algorithm as a *failure* if the running time of the execution exceeds the expected running time or the high-confidence bound.

DEFINITION 13.2 [Confidence Bounds] Let α, β and c be positive constants. A randomized algorithm A requires resource bound $f(n)$ with

1. $n-$*exponential* probability or very high probability, if for any input of size n, the amount of the resource used by A is at most $\alpha f(n)$ with probability $1 - O(\beta^{-n})$, $\beta > 1$. In this case $f(n)$ is called a *very high confidence bound*.
2. n − *polynomial* probability or high probability, if for any input of size n, the

amount of the resource used by A is at most $\alpha f(n)$ with probability $1 - O(n^{-c})$. In this case $f(n)$ is called a *high confidence bound*.

3. $n - log$ probability or very good probability, if for any input of size n, the amount of the resource used by A is at most $\alpha f(n)$ with probability $1 - O((\log n)^{-c})$. In this case $f(n)$ is called a *very good confidence bound*.
4. $high - constant$ probability, if for any input of size n, the amount of the resource used by A is at most $\alpha f(n)$ with probability $1 - O(\beta^{-\alpha})$, $\beta > 1$.

The practical significance of this definition can be understood from the following discussions. For instance, let A be a *Las Vegas* type algorithm with $f(n)$ as a high confidence bound for its running time. As noted before, the actual running time $T(n)$ may vary from one execution to another but the definition above implies that, for any execution, on any input, $Pr(T(n) > f(n)) = O(n^{-c})$. Even for modest values of n and c, this bound implies an extreme rarity of failure. For instance, if $n = 1000$ and $c = 4$, we may conclude that the chance that the running time of the algorithm A exceeding the threshold value is one in zillion.

13.2.2 Basics of Probability Theory

We assume that the reader is familiar with basic notions such as *sample space, event* and basic *axioms of probability*. We denote as $Pr(E)$ the probability of the event E. Several results follow immediately from the basic axioms, and some of them are listed in Lemma 13.1.

LEMMA 13.1 The following laws of probability must hold:

1. $Pr(\phi) = 0$
2. $Pr(E^c) = 1 - Pr(E)$
3. $Pr(E_1) \leq Pr(E_2)$ if $E_1 \subseteq E_2$
4. $Pr(E_1 \cup E_2) = Pr(E_1) + Pr(E_2) - Pr(E_1 \cap E_2) \leq Pr(E_1) + Pr(E_2)$

Extending item 4 in Lemma 13.1 to countable unions yields the property known as *sub additivity*. Also known as *Boole's Inequality*, it is stated in Theorem 13.1.

THEOREM 13.1 *[Boole's Inequality]* $Pr(\cup_{i=1}^{\infty} E_i) \leq \sum_{i=1}^{\infty} Pr(E_i)$

A probability distribution is said to be *discrete* if the sample space S is finite or countable. If $E = \{e_1, e_2, ..., e_k\}$ is an event, $Pr(E) = \sum_{i=1}^{k} Pr(\{e_i\})$ because all elementary events are mutually exclusive. If $|S| = n$ and $Pr(\{e\}) = \frac{1}{n}$ for every elementary event e in S, we call the distribution a *uniform distribution* of S. In this case,

$$\begin{aligned} Pr(E) &= \sum_{e \in E} Pr(e) \\ &= \sum_{e \in E} \frac{1}{n} \\ &= |E|/|S| \end{aligned}$$

which agrees with our intuitive and a well-known definition that probability is the ratio of the favorable number of cases to the total number of cases, when all the elementary events are equally likely to occur.

13.2.3 Conditional Probability

In several situations, the occurrence of an event may change the uncertainties in the occurrence of other events. For instance, insurance companies charge higher rates to various categories of drivers, such as those who have been involved in traffic accidents, because the probabilities of these drivers filing a claim is altered based on these additional factors.

DEFINITION 13.3 [Conditional Probability] The *conditional probability* of an event E_1 given that another event E_2 has occurred is defined by $Pr(E_1/E_2)$ ("$Pr(E_1/E_2)$" is read as "the probability of E_1 given E_2.").

LEMMA 13.2 $Pr(E_1/E_2) = \frac{Pr(E_1 \cap E_2)}{Pr(E_2)}$, provided $Pr(E_2) \neq 0$.

Lemma 13.2 shows that the conditional probability of two events is easy to compute. When two or more events do not influence each other, they are said to be independent. There are several notions of independence when more than two events are involved. Formally,

DEFINITION 13.4 [Independence of two events] Two events are *independent* if $Pr(E_1 \cap E_2) = Pr(E_1)Pr(E_2)$, or equivalently, $Pr(E_1/E_2) = Pr(E_1)$.

DEFINITION 13.5 [Pairwise independence] Events $E_1, E_2, \ldots E_k$ are said to be *pairwise independent* if $Pr(E_i \cap E_j) = Pr(E_i)Pr(E_j)$, $1 \leq i \neq j \leq n$.

Given a partition $S_1, \ldots, S_k$ of the sample space S, the probability of an event E may be expressed in terms of mutually exclusive events by using conditional probabilities. This is known as the *law of total probability in the conditional form*.

LEMMA 13.3 [Law of total probability in the conditional form] For any partition $S_1, ..., S_k$ of the sample space S, $Pr(E) = \sum_{i=1}^{k} Pr(E/S_i)\ Pr(S_i)$.

The law of total probability in the conditional form is an extremely useful tool for calculating the probabilities of events. In general, to calculate the probability of a complex event E, we may attempt to find a partition $S_1, S_2, \ldots, S_k$ of S such that both $Pr(E/S_i)$ and $Pr(S_i)$ are easy to calculate and then apply Lemma 13.3. Another important tool is *Bayes' Rule*.

THEOREM 13.2 *[Bayes' Rule] For events with non-zero probabilities,*

1. $Pr(E_1/E_2) = \frac{Pr(E_2/E_1)Pr(E_1)}{Pr(E_2)}$
2. *If* $S_1, S_2, ..., S_k$ *is a partition,* $Pr(S_i/E) = \frac{Pr(E/S_i)Pr(S_i)}{\sum_{j=1} Pr(E/S_j)Pr(S_j)}$

Proof Part (1) is immediate by applying the definition of conditional probability; Part (2) is immediate from Lemma 13.3.

Random Variables and Expectation

Most of the random phenomena are so complex that it is very difficult to obtain detailed information about the outcome. Thus, we typically study one or two numerical parameters that we associate with the outcomes. In other words, we focus our attention on certain real-valued functions defined on the sample space.

DEFINITION 13.6 A *random variable* is a function from a sample space into the set of real numbers. For a random variable X, $R(X)$ denotes the *range* of the function X.

Having defined a random variable over a sample space, an event of interest may be studied through the values taken by the random variables on the outcomes belonging to the event. In order to facilitate such a study, we supplement the definition of a random variable by specifying how the probability is assigned to (or distributed over) the values that the random variable may assume. Although a rigorous study of random variables will require a more subtle definition, we restrict our attention to the following simpler definitions that are sufficient for our purposes.

A random variable X is a *discrete random variable* if its range $R(X)$ is a finite or countable set (of real numbers). This immediately implies that any random variable that is defined over a finite or countable sample space is necessarily discrete. However, discrete random variables may also be defined on uncountable sample spaces. For a random variable X, we define its *probability mass function (pmf)* as follows:

DEFINITION 13.7 [Probability mass function] For a random variable X, the *probability mass function* $p(x)$ is defined as $p(x) = Pr(X = x)$, $\forall x \in R(X)$.

The probability mass function is also known as the *probability density function*. Certain trivial properties are immediate, and are given in Lemma 13.4.

LEMMA 13.4 The probability mass function $p(x)$ must satisfy

1. $p(x) \geq 0$, $\forall x \in R(X)$
2. $\sum_{x \in R(X)} p(x) = 1$

Let X be a discrete random variable with probability mass function $p(x)$ and range $R(X)$. The *expectation* of X (also known as the *expected value* or *mean* of X) is its average value. Formally,

DEFINITION 13.8 [Expected value of a discrete random variable] The *expected value* of a discrete random variable X with probability mass function $p(x)$ is given by $E(X) = \mu_X = \sum_{x \in R(X)} xp(x)$.

LEMMA 13.5 The expected value has the following properties:

1. $E(cX) = cE(X)$ if c is a constant

2. (Linearity of expectation) $E(X+Y) = E(X) + E(Y)$, provided the expectations of X and Y exist

Finally, a useful way of computing the expected value is given by Theorem 13.3.

THEOREM 13.3 *If $R(X) = \{0, 1, 2, ...\}$, then $E(X) = \sum_{i=1}^{\infty} Pr(X \geq i)$.*

Proof

$$\begin{aligned} E(X) &= \sum_{i=0}^{\infty} iPr(X = i) \\ &= \sum_{i=0}^{\infty} i(Pr(X \geq i) - Pr(X \geq i+1)) \\ &= \sum_{i=1}^{\infty} Pr(X \geq i) \end{aligned}$$

13.2.4 Some Basic Distributions

Bernoulli Distribution

We have seen that a coin flip is an example of a random experiment and it has two possible outcomes, called *success* and *failure*. Assume that success occurs with probability p and that failure occurs with probability $q = 1 - p$. Such a coin is called p-biased coin. A coin flip is also known as *Bernoulli Trial*, in honor of the mathematician who investigated extensively the distributions that arise in a sequence of coin flips.

DEFINITION 13.9 A random variable X with range $R(X) = \{0, 1\}$ and probability mass function $Pr(X = 1) = p$, $Pr(X = 0) = 1 - p$ is said to follow the Bernoulli Distribution. We also say that X is a Bernoulli random variable with parameter p.

Binomial Distribution

Let $\left(\begin{array}{c} n \\ k \end{array} \right)$ denote the number of k-combinations of elements chosen from a set of n elements. Recall that $\left(\begin{array}{c} n \\ k \end{array} \right) = \frac{n!}{k!(n-k)!}$ and $\left(\begin{array}{c} n \\ 0 \end{array} \right) = 1$ since $0! = 1$. $\left(\begin{array}{c} n \\ k \end{array} \right)$ denotes the *binomial coefficients* because they arise in the expansion of $(a+b)^n$.

Define the random variable X to be the number of successes in n flips of a p-biased coin. The variable X satisfies the *binomial distribution*. Specifically,

DEFINITION 13.10 [Binomial distribution] A random variable with range $R(X) = \{0, 1, 2, \ldots, n\}$ and probability mass function

$$Pr(X = k) = b(k, n, p) = \left(\begin{array}{c} n \\ k \end{array} \right) p^k q^{n-k}, \quad for \quad k = 0, 1, \ldots, n$$

satisfies the *binomial distribution*. The random variable X is called a binomial random variable with parameters n and p.

THEOREM 13.4 *For a binomial random variable X, with parameters n and p, $E(X) = np$ and $Var(X) = npq$.*

Geometric Distribution

Let X be a random variable X denoting the number of times we toss a p-biased coin until we get a success. Then, X satisfies the *geometric distribution*. Specifically,

DEFINITION 13.11 [Geometric distribution] A random variable with range $R(X) = \{1, 2, \ldots, \infty\}$ and probability mass function $Pr(X = k) = q^{k-1}p$, for $k = 1, 2, \ldots, \infty$ satisfies the *geometric distribution*. We also say that X is a geometric random variable with parameter p.

The probability mass function is based on $k-1$ failures followed by a success in a sequence of k independent trials. The mean and variance of a geometric distribution are easy to compute.

THEOREM 13.5 *For a geometrically distributed random variable X, $E(X) = \frac{1}{p}$ and $Var(X) = \frac{q}{p^2}$.*

Negative Binomial distribution

Fix an integer n and define a random variable X denoting the number of flips of a p-biased coin to obtain n successes. The variable X satisfies a negative binomial distribution. Specifically,

DEFINITION 13.12 A random variable X with $R(X) = \{0, 1, 2, ...\}$ and probability mass function defined by

$$\begin{aligned} Pr(X = k) &= \binom{k-1}{n-1} p^n q^{k-n} \quad if\ k \geq n \\ &= 0 \quad if\ 0 \leq k < n \end{aligned} \tag{13.1}$$

is said to be a *negative binomial random variable* with parameters n and p.

Equation (13.1) follows because, in order for the n^{th} success to occur in the k^{th} flip there should be $n-1$ successes in the first $k-1$ flips and the k^{th} flip should also result in a success.

DEFINITION 13.13 Given n identically distributed independent random variables $X_1, X_2, \ldots, X_n$, the sum

$$S_n = X_1 + X_2 + \cdots + X_n$$

defines a new random variable. If n is a finite, fixed constant then S_n is known as the *deterministic sum* of n random variables.

On the other hand, if n itself is a random variable, S_n is called a *random sum*.

THEOREM 13.6 *Let $X = X_1 + X_2 + \cdots + X_n$ be a deterministic sum of n identical independent random variables. Then*

1. *If X_i is a Bernoulli random variable with parameter p then X is a binomial random variable with parameters n and p.*
2. *If X_i is a geometric random variable with parameter p, then X is a negative binomial with parameters n and p.*
3. *If X_i is a (negative) binomial random variable with parameters r and p then X is a (negative) binomial random variable with parameters nr and p.*

Deterministic sums and random sums may have entirely different characteristics as the following theorem shows.

THEOREM 13.7 *Let $X = X_1 + \cdots + X_N$ be a random sum of N geometric random variables with parameter p. Let N be a geometric random variable with parameter α. Then X is a geometric random variable with parameter αp.*

13.2.5 Tail Estimates

Recall that the running time of a *Las Vegas* type randomized algorithm is a random variable and thus we are interested in the probability of the running time exceeding a certain threshold value.

Typically we would like this probability to be very small so that the threshold value may be taken as the figure of merit with high degree of confidence. Thus we often compute or estimate quantities of the form $Pr(X \geq k)$ or $Pr(X \leq k)$ during the analysis of randomized algorithms. Estimates for the quantities of the form $Pr(X \geq k)$ are known as *tail estimates*. The next two theorems state some very useful tail estimates derived by Chernoff. These bounds are popularly known as *Chernoff bounds*. For simple and elegant proofs of these and other related bounds you may refer [1].

THEOREM 13.8 *Let X be a sum of n independent random variables X_i with $R(X_i) \subseteq [0,1]$. Let $E(X) = \mu$. Then,*

$$Pr(X \geq k) \leq \left(\frac{\mu}{k}\right)^k \left(\frac{n-\mu}{n-k}\right)^{n-k} \quad \text{for } k > \mu \tag{13.2}$$

$$\leq \left(\frac{\mu}{k}\right)^k e^{k-\mu} \quad \text{for } k > \mu \tag{13.3}$$

$$Pr(X \geq (1+\epsilon)\mu) \leq \left[\frac{e^\epsilon}{(1+\epsilon)^{(1+\epsilon)}}\right]^\mu \quad \text{for } \epsilon \geq 0 \tag{13.4}$$

THEOREM 13.9 *Let X be a sum of n independent random variables X_i with $R(X_i) \subseteq [0,1]$. Let $E(X) = \mu$. Then,*

$$Pr(X \leq k) \leq \left(\frac{\mu}{k}\right)^k \left(\frac{n-\mu}{n-k}\right)^{n-k} \quad k < \mu \tag{13.5}$$

$$\leq \left(\frac{\mu}{k}\right)^k e^{k-\mu} \quad k < \mu \tag{13.6}$$

$$Pr(X \leq (1-\epsilon)\mu) \leq e^{-\frac{\mu\epsilon^2}{2}}, \quad \text{for } \epsilon \in (0,1) \tag{13.7}$$

Recall that a deterministic sum of several geometric variables results in a negative binomial random variable. Hence, intuitively, we may note that only the upper tail is meaningful for this distribution. The following well-known result relates the upper tail value of a negative binomial distribution to a lower tail value of a suitably defined binomial distribution. Hence all the results derived for lower tail estimates of the binomial distribution can be used to derive upper tail estimates for negative binomial distribution. This is a very important result because finding bounds for the right tail of a negative binomial distribution directly from its definition is very difficult.

THEOREM 13.10 *Let X be a negative binomial random variable with parameters r and p. Then, $Pr(X > n) = Pr(Y < r)$ where Y is a binomial random variable with parameters n and p.*

13.3 Skip Lists

Linked list is the simplest of all dynamic data structures implementing a *Dictionary*. However, the complexity of *Search* operation is $O(n)$ in a *Linked list*. Even the *Insert* and *Delete* operations require $O(n)$ time if we do not specify the exact position of the item in the list. *Skip List* is a novel structure, where using randomness, we construct a number of progressively smaller lists and maintain this collection in a clever way to provide a data structure that is competitive to balanced tree structures. The main advantage offered by skip list is that the codes implementing the dictionary operations are very simple and resemble list operations. No complicated structural transformations such as *rotations* are done and yet the expected time complexity of *Dictionary* operations on *Skip Lists* are quite comparable to that of AVL trees or splay trees. *Skip Lists* are introduced by Pugh [6].

Throughout this section let $S = \{k_1, k_2, \ldots, k_n\}$ be the set of keys and assume that $k_1 < k_2 < \ldots < k_n$.

DEFINITION 13.14 Let $S_0, S_1, S_2, \ldots, S_r$ be a collection of sets satisfying

$$S = S_0 \supseteq S_1 \supseteq S_2 \supseteq \cdots \supset S_r = \phi$$

Then, we say that the collection $S_0, S_1, S_2, \ldots, S_r$ defines a *leveling with r levels on S*. The keys in S_i are said to be in level i, $0 \leq i \leq r$. The set S_0 is called the base level for the leveling scheme. Notice that there is exactly one empty level, namely S_r. The level number $l(k)$ of a key $k \in S$ is defined by

$$l(k) = max\{i \mid k \in L_i\}.$$

In other words, $k \in S_0, S_1, S_2, \ldots, S_{l(k)}$ but $k \notin S_{l(k)+1} \cdots S_r$.

For an efficient implementation of the dictionary, instead of working with the current set S of keys, we would rather work with a leveling of S. The items of S_i will be put in the *increasing order* in a linked list denoted by L_i. We attach the special keys $-\infty$ at the beginning and $+\infty$ at the end of each list L_i as sentinels. In practice, $-\infty$ is a key value that is smaller than any key we consider in the application and $+\infty$ denotes a key value larger than all the possible keys. A leveling of S is implemented by maintaining all the lists

$L_0, L_1, L_2, \ldots, L_r$ with some more additional links as shown in Figure 13.1. Specifically, the box containing a key k in L_i will have a pointer to the box containing k in L_{i-1}. We call such pointers *descent pointers*. The links connecting items of the same list are called *horizontal pointers*. Let B be a pointer to a box in the skip list. We use the notations $Hnext[B]$, and $Dnext[B]$, for the horizontal and descent pointers of the box pointed by B respectively. The notation $key[B]$ is used to denote the key stored in the box pointed by B. The name of the skip list is nothing but a pointer to the box containing $-\infty$ in the rth level as shown in the figure. From the Figure 13.1 it is clear that L_i has horizontal pointers that skip over several intermediate elements of L_{i-1}. That is why this data structure is called the *Skip List*.

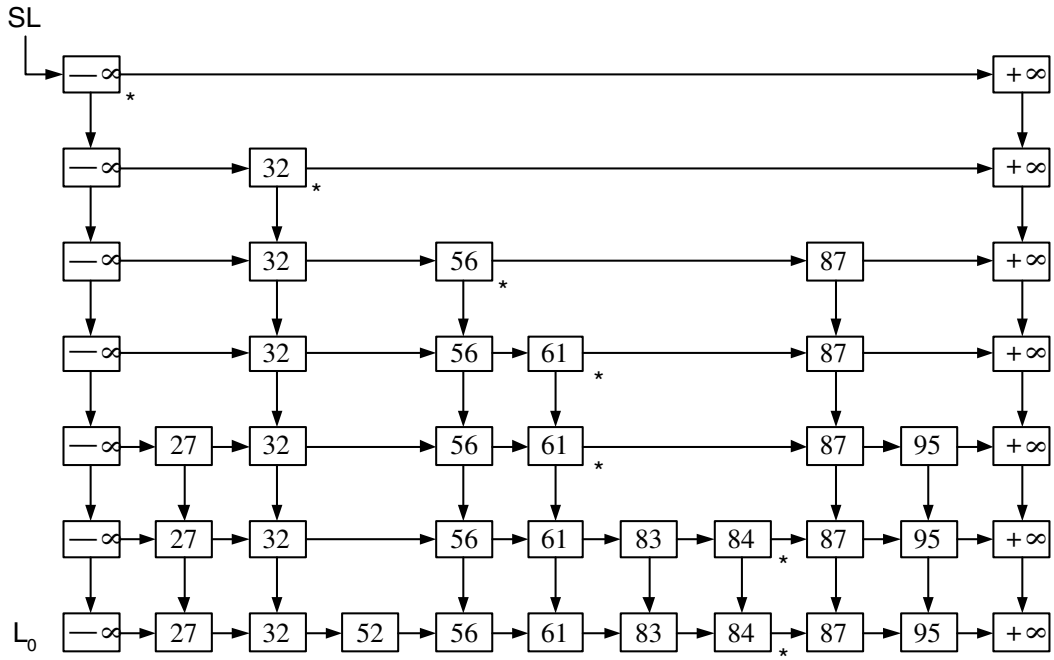

FIGURE 13.1: A Skip List. The starred nodes are marked by Mark(86,SL).

How do we arrive at a leveling for a given set? We may construct S_{i+1} from S_i in a systematic, deterministic way. While this may not pose any problem for a static search problem, it will be extremely cumbersome to implement the dynamic dictionary operations. This is where randomness is helpful. To get S_{i+1} from S_i, we do the following. For each element k of S_i toss a coin and include k in S_{i+1} iff we get *success* in the coin toss. If the coin is p-biased, we expect $\mid S_{i+1} \mid$ to be roughly equal to $p \mid S_i \mid$. Starting from S, we may repeatedly obtain sets corresponding to successive levels. Since the coin tosses are independent, there is another useful, and equivalent way to look at our construction. For each key k in S, keep tossing a coin until we get a *failure*. Let h be the number of successes before the first failure. Then, we include k in h further levels treating $S = S_0$ as the base level. In other words, we include k in the sets $S_1, S_2, \ldots, S_h$. This suggests us to define a random variable Z_i for each key $k_i \in S_0$ to be the number of times we toss a p-biased coin

before we obtain a *failure*. Since Z_i denotes the number of additional copies of k_i in the skip list, the value $maximum\{Z_i : 1 \leq i \leq n\}$ defines the highest nonempty level. Hence, we have,

$$r = 1 + maximum\{Z_i : 1 \leq i \leq n\} \tag{13.8}$$

$$\mid SL \mid = n + Z_1 + Z_2 + \cdots + Z_n + 2r + 2 \tag{13.9}$$

where r is the number of levels and $\mid SL \mid$ is the number of boxes or the space complexity measure of the *Skip List*. In the expression for $\mid SL \mid$ we have added n to count the keys in the base level and $2r + 2$ counts the sentinel boxes containing $+\infty$ and $-\infty$ in each level.

13.4 Structural Properties of Skip Lists

13.4.1 Number of Levels in Skip List

Recall that $r = 1 + max_i\{Z_i\}$. Notice that $Z_i \geq k$ iff the coin tossed for k_i gets a run of at least k successes right from the beginning and this happens with probability p^k. Since $r \geq k$ iff at least one of $Z_i \geq k - 1$, we easily get the following fact from Boole's inequality

$$Pr(r \geq k) \leq np^{k-1}$$

Choosing $k = 4\log_{1/p} n + 1$, we immediately obtain a high confidence bound for the number of levels. In fact,

$$\begin{aligned} Pr(r \geq 4\log n + 1) &\leq nn^{-4} \\ &= \frac{1}{n^3} \end{aligned} \tag{13.10}$$

We obtain an estimation for the expected value of r, using the formula stated in theorem (13.3) as follows:

$$\begin{aligned} E(r) &= \sum_{i=1}^{\infty} Pr(r \geq i) \\ &= \sum_{i=1}^{4\log n} Pr(r \geq i) + \sum_{i > 4\log n} Pr(r \geq i) \\ &\leq \sum_{i=1}^{4\log n} 1 + \sum_{i > 4\log n}^{\infty} np^{i-1} \\ &= 4\log n + np^{4\log n}(1 + p + p^2 + \cdots) \\ &= 4\log n + n \cdot \frac{1}{n^4} \cdot (1-p)^{-1} \text{ if base of the log is 1/p} \\ &\leq 4\log n + 1 \text{ for sufficiently large } n \end{aligned}$$

Thus $E(r) = O(\log n)$
Hence,

THEOREM 13.11 *The expected number of levels in a skip list of n elements is $O(\log n)$. In fact, the number is $O(\log n)$ with high probability.*

13.4.2 Space Complexity

Recall that the space complexity, $\mid SL \mid$ is given by

$$\mid SL \mid = Z_1 + Z_2 + \cdots + Z_n + n + 2r + 2.$$

As we know that $r = O(\log n)$ with high probability, let us focus on the sum

$$Z = Z_1 + Z_2 + \cdots + Z_n.$$

Since Z_i is a geometric random variable with parameter p, Z is a negative binomial random variable by theorem 13.6.

Thus $E(Z) = \frac{n}{p} = O(n)$.

We can in fact show that Z is $O(n)$ with very high probability.

Now, from theorem (13.10) $Pr(Z > 4n) = Pr(X < n)$ where X is a binomial distribution with parameters $4n$ and p. We now assume that $p = 1/2$ just for the sake of simplicity in arithmetic.

In the first Chernoff bound mentioned in theorem (13.9), replacing n, μ and k respectively with $4n$, $2n$ and n, we obtain,

$$\begin{aligned} Pr(X < n) &\leq \left(\frac{2n}{n}\right)^n \left(\frac{2n}{3n}\right)^{3n} \\ &= \left(2 \cdot \frac{2^3}{3^3}\right)^n \\ &= \left(\frac{16}{27}\right)^n \\ &= \left(\frac{27}{16}\right)^{-n} \end{aligned}$$

This implies that $4n$ is in fact a very high confidence bound for Z. Since $\mid SL \mid = Z+n+2r+2$, we easily conclude that

THEOREM 13.12 *The space complexity of a skip list for a set of size n is $O(n)$ with very high probability.*

13.5 Dictionary Operations

We shall use a simple procedure called *Mark* in all the dictionary operations. The procedure *Mark* takes an arbitrary value x as input and marks in each level the box containing the largest key that is less than x. This property implies that insertion, deletion or search should all be done next to the marked boxes. Let $M_i(x)$ denote the box in the i^{th} level that is marked by the procedure call *Mark*(x, SL). Recall the convention used in the linked structure that name of a box in a linked list is nothing but a pointer to that box. The keys in the marked boxes $M_i(x)$ satisfy the following condition :

$$key[M_i(x)] < x \leq key[Hnext[M_i(x)]] \quad \text{for all} \;\; 0 \leq i \leq r. \tag{13.11}$$

The procedure *Mark* begins its computation at the box containing $-\infty$ in level r. At any current level, we traverse along the level using horizontal pointers until we find that the

next box is containing a key larger or equal to x. When this happens, we mark the current box and use the *descent pointer* to reach the next lower level and work at this level in the same way. The procedure stops after marking a box in level 0. See Figure 13.1 for the nodes marked by the call *Mark(86,SL)*.

```
Algorithm Mark(x,SL)
-- x is an arbitrary value.
-- r is the number of levels in the skip list SL.
1. Temp  = SL.
2. For i = r down to 0 do
     While (key[Hnext[Temp]] < x)
            Temp = Hnext[Temp];
     Mark the box pointed by Temp;
     Temp = Dnext[Temp];
3. End.
```

We shall now outline how to use marked nodes for *Dictionary* operations. Let $M_0(x)$ be the box in level 0 marked by *Mark(x)*. It is clear that the box next to $M_0(x)$ will have x iff $x \in S$. Hence our algorithm for *Search* is rather straight forward.

To insert an item in the *Skip List*, we begin by marking the *Skip List* with respect to the value x to be inserted. We assume that x is not already in the *Skip List*. Once the marking is done, inserting x is very easy. Obviously x is to be inserted next to the marked boxes. But in how many levels we insert x? We determine the number of levels by tossing a coin on behalf of x until we get a *failure*. If we get h successes, we would want x to be inserted in all levels from 0 to h. If $h \geq r$, we simply insert x at all the existing levels and create a new level consisting of only $-\infty$ and $+\infty$ that corresponds to the empty set. The insertion will be done starting from the base level. However, the marked boxes are identified starting from the highest level. This means, the *Insert* procedure needs the marked boxes in the reverse order of its generation. An obvious strategy is to push the marked boxes in a stack as and when they are marked in the *Mark* procedure. We may pop from the stack as many marked boxes as needed for insertion and then insert x next to each of the popped boxes.

The deletion is even simpler. To delete a key x from S, simply mark the *Skip List* with respect to x. Note that if x is in L_i, then it will be found next to the marked box in L_i. Hence, we can use the *horizontal pointers* in the marked boxes to delete x from each level where x is present. If the deletion creates one or more empty levels, we ignore all of them and retain only one level corresponding to the empty set. In other words, the number of levels in the *Skip List* is reduced in this case. In fact, we may delete an item "on the fly" during the execution of the *Mark* procedure itself. As the details are simple we omit pseudo codes for these operations.

13.6 Analysis of Dictionary Operations

It is easy to see that the cost of *Search*, *Insert* and *Delete* operations are dominated by the cost of *Mark* procedure. Hence we shall analyze only the *Mark* procedure. The *Mark* procedure starts at the 'r'th level and proceeds like a downward walk on a staircase and ends at level 0. The complexity of *Mark* procedure is clearly proportional to the number of edges it traversed. It is advantageous to view the same path as if it is built from level 0 to level r. In other words, we analyze the building of the path in a direction opposite to the direction in which it is actually built by the procedure. Such an analysis is known as *backward analysis*.

Henceforth let P denote the path from level 0 to level r traversed by *Mark(x)* for the given fixed x. The path P will have several vertical edges and horizontal edges. (Note that at every box either P moves vertically above or moves horizontally to the left). Clearly, P has r vertical edges. To estimate number of horizontal edges, we need the following lemmas.

LEMMA 13.6 Let the box b containing k at level i be marked by Mark(x). Let a box w containing k be present at level $i+1$. Then, w is also marked.

Proof Since b is marked, from (13.11), we get that there is no value between k and x in level i. This fact holds good for L_{i+1} too because $S_{i+1} \subseteq S_i$. Hence the lemma.

LEMMA 13.7 Let the box b containing k at level i be marked by Mark(x). Let $k \notin L_{i+1}$. Let u be the first box to the left of b in L_i having a "vertical neighbor" w. Then w is marked.

Proof Let $w.key = u.key = y$. Since b is marked, k satisfies condition (13.11). Since u is the first node in the left of b having a vertical neighbor, none of the keys with values in between y and x will be in L_{i+1}. Also, $k \notin L_{i+1}$ according to our assumption in the lemma. Thus y is the element in L_{i+1} that is just less than x. That is, y satisfies the condition (13.11) at level $i+1$. Hence the w at level $i+1$ will be marked.

Lemmas (13.6) and (13.7) characterize the segment of P between two successive marked boxes. This allows us to give an incremental description of P in the following way.

P starts from the marked box at level 0. It proceeds vertically as far as possible (lemma 13.6) and when it cannot proceed vertically it proceeds horizontally. At the "first" opportunity to move vertically, it does so (lemma 13.7), and continues its vertical journey. Since for any box a vertical neighbor exists only with a probability p, we see that P proceeds from the current box vertically with probability p and horizontally with probability $(1-p)$.

Hence, the number of horizontal edges of P in level i is a geometric random variable, say, Y_i, with parameter $(1-p)$. Since the number of vertical edges in P is exactly r, we conclude,

THEOREM 13.13 *The number of edges traversed by Mark(x) for any fixed x is given by $\mid P \mid = r + (Y_0 + Y_1 + Y_2 + \cdots + Y_{r-1})$ where Y_i is a geometric random variable with parameters $1-p$ and r is the random variable denoting the number of levels in the Skip List.*

Our next mission is to obtain a high confidence bound for the size of P. As we have already derived high confidence bound for r, let focus on the sum $H_r = Y_0 + Y_1 + \cdots + Y_{r-1}$ for a while. Since r is a random variable H_r is not a deterministic sum of random variables but a *random sum* of random variables.

Hence we cannot directly apply theorem (13.6) and the bounds for negative binomial distributions.

Let X be the event of the random variable r taking a value less than or equal to $4 \log n$. Note that $P(\overline{X}) < \frac{1}{n^3}$ by (13.10).

From the law of total probability, Boole's inequality and equation (13.10) we get,

$$\begin{aligned} Pr(H_r > 16\log n) &= Pr([H_r > 16\log n] \cap X) + Pr([H_r > 16\log n] \cap \overline{X}) \\ &= Pr([H_r > 16\log n] \cap r \leq 4\log n) + Pr([H_r > 16\log n] \cap \overline{X}) \\ &\leq \sum_{k=0}^{4\log n} Pr(H_k > 16\log n) + Pr(\overline{X}) \\ &\leq (1 + 4\log n) Pr(H_{4\log n} > 16\log n) + \frac{1}{n^3} \end{aligned}$$

Now $Pr(H_{4\log n} > 16\log n)$ can be computed in a manner identical to the one we carried out in the space complexity analysis. Notice that $H_{4\log n}$ is a deterministic sum of geometric random variables. Hence we can apply theorem (13.10) and theorem (13.9) to derive a high confidence bound. Specifically, by theorem (13.10),

$$Pr(H_{4\log n} > 16\log n) = Pr(X < 4\log n),$$

where X is a binomial random variable with parameters $16\log n$ and p. Choosing $p = 1/2$ allows us to set $\mu = 8\log n$, $k = 4\log n$ and replace n by $16\log n$ in the first inequality of theorem (13.9). Putting all these together we obtain,

$$\begin{aligned} Pr(H_{4\log n} > 16\log n) &= Pr(X < 4\log n) \\ &\leq \left(\frac{8\log n}{4\log n}\right)^{4\log n} \left(\frac{8\log n}{12\log n}\right)^{12\log n} \\ &= \left(2 \cdot \frac{2^3}{3^3}\right)^{4\log n} \\ &= \left(\frac{16}{27}\right)^{4\log n} \\ &< \left(\frac{1}{8}\right)^{\log n} \\ &= \frac{1}{n^3} \end{aligned}$$

Therefore

$$\begin{aligned} Pr(H_r > 16\log n) &< (1 + 4\log n)\left(\frac{1}{n^3}\right) + \frac{1}{n^3} \\ &< \frac{1}{n^2} \quad if \quad n \geq 32 \end{aligned}$$

This completes the derivation of a high confidence bound for H_r. From this, we can easily obtain a bound for expected value of H_r. We use theorem (13.3) and write the expression for $E(H_r)$ as

$$E(H_r) = \sum_{i=1}^{16\log n} Pr(H_r \geq i) + \sum_{i > 16\log n} Pr(H_r \geq i).$$

The first sum is bounded above by $16\log n$ as each probability value is less than 1 and by the high confidence bound that we have established just now, we see that the second sum is dominated by $\sum_{i=1}^{\infty} 1/i^2$ which is a constant. Thus we obtain,

$$E(H_r) \leq 16 \log n + c = O(\log n).$$

Since $P = r + H_r$ we easily get that $E(\mid P \mid) = O(\log n)$ and $\mid P \mid = O(\log n)$ with probability greater than $1 - O(\frac{1}{n^2})$. Observe that we have analyzed *Mark(x)* for a given fixed x. To show that the high confidence bound for any x, we need to proceed little further. Note that there are only $n + 1$ distinct paths possible with respect to the set $\{-\infty = k_0, k_1, ..., k_n, k_{n+1} = +\infty\}$, each corresponding to the x lying in the internal $[k_i, k_{i+1}), i = 0, 1, ..., n$.

Therefore, for any x, *Mark(x)* walks along a path P satisfying $E(\mid P \mid) = O(\log n)$ and $\mid P \mid = O(\log n)$ with probability greater than $1 - O(\frac{1}{n})$.

Summarizing,

THEOREM 13.14 *The Dictionary operations Insert, Delete, and Search take $O(\log n)$ expected time when implemented using Skip Lists. Moreover, the running time for Dictionary operations in a Skip List is $O(\log n)$ with high probability.*

13.7 Randomized Binary Search Trees

A *Binary Search Tree* (BST) for a set S of keys is a binary tree satisfying the following properties.

(a) Each node has exactly one key of S. We use the notation $v.key$ to denote the key stored at node v.

(b) For all node v and for all nodes u in the left subtree of v and for all nodes w in the right subtree of v, the keys stored in them satisfy the so called *search tree property*:

$$u.key < v.key < w.key$$

The complexity of the dictionary operations are bounded by the height of the binary search tree. Hence, ways and means were explored for controlling the height of the tree during the course of execution. Several clever balancing schemes were discovered with varying degrees of complexities. In general, the implementation of balancing schemes are tedious and we may have to perform a number of rotations which are complicated operations. Another line of research explored the potential of possible randomness in the input data. The idea was to completely avoid balancing schemes and hope to have '*short*' trees due to randomness in input. When only random insertion are done, we obtain so called *Randomly Built Binary Tree* (RBBT). RBBTs have been shown to have $O(\log n)$ expected height.

What is the meaning of random insertion? Suppose we have already inserted the values $a_1, a_2, a_3, \cdots, a_{k-1}$. These values, when considered in sorted order, define k intervals on the real line and the new value to be inserted, say x, is equally likely to be in any of the k intervals.

The first drawback of RBBT is that this assumption may not be valid in practice and when this assumption is not satisfied, the resulting tree structure could be highly skewed and the complexity of search as well as insertion could be as high as $O(n)$. The second major drawback is when deletion is also done on these structures, there is a tremendous degradation in the performance. There is no theoretical results available and extensive

empirical studies show that the height of an RBBT could grow to $O(\sqrt{n})$ when we have arbitrary mix of insertion and deletion, even if the randomness assumption is satisfied for inserting elements. Thus, we did not have a satisfactory solution for nearly three decades.

In short, the randomness was not preserved by the deletion operation and randomness preserving binary tree structures for the dictionary operations was one of the outstanding open problems, until an elegant affirmative answer is provided by Martinez and Roura in their landmark paper [3].

In this section, we shall briefly discuss about structure proposed by Martinez and Roura.

DEFINITION 13.15 [Randomized Binary Search Trees] Let T be a binary search tree of size n. If $n = 0$, then $T = NULL$ and it is a random binary search tree. If $n > 0$, T is a random binary search tree iff both its left subtree L and right subtree R are independent random binary search trees and

$$\Pr\{Size(L) = i | Size(T) = n\} = \frac{1}{n},\ 0 \leq i \leq n.$$

The above definition implies that every key has the same probability of $\frac{1}{n}$ for becoming the root of the tree. It is easy to prove that the expected height of a RBST with n nodes is $O(\log n)$. The RBSTs possess a number of interesting structural properties and the classic book by Mahmoud [2] discusses them in detail. In view of the above fact, it is enough if we prove that when insert or delete operation is performed on a RBST, the resulting tree is also an RBST.

13.7.1 Insertion in RBST

When a key x is inserted in a tree T of size n, we obtain a tree T' of size $n + 1$. For T', as we observed earlier, x should be in the root with probability $\frac{1}{n+1}$. This is our starting point.

```
Algorithm Insert(x, T)
- L is the left subtree of the root
- R is the right subtree of the root
1. n = size(T);
2. r = random(0, n);
3. If (r = n) then
      Insert_at_root(x, T);
4. If (x < key at root of T) then
      Insert(x, L);
   Else
      Insert(x, R);
```

To insert x as a root of the resulting tree, we first split the current tree into two trees labeled $T_<$ and $T_>$, where $T_<$ contains all the keys in T that are smaller than x and $T_>$ contains all the keys in T that are larger than x. The output tree T' is constructed by placing x at the root and attaching $T_<$ as its left subtree and $T_>$ as its right subtree. The algorithm for splitting a tree T into $T_<$ and $T_>$ with respect to a value x is similar to the partitioning algorithm done in *quicksort*. Specifically, the algorithm *split(x,T)* works as

follows. If T is empty , nothing needs to be done; both $T_<$ and $T_>$ are empty. When T is non-empty we compare x with Root(T).key. If $x <$ Root(T).key, then root of T as well as the right subtree of T belong to $T_>$. To compute $T_<$ and the remaining part of $T_>$ we recursively call *split*(x, L), where L is the left subtree for the root of T. If $x >$ Root(T).key, $T_<$ is built first and recursion proceeds with *split*(x, R). The details are left as easy exercise.

We shall first prove that $T_<$ and $T_>$ are independent *Random Binary Search Trees*. Formally,

THEOREM 13.15 *Let $T_<$ and $T_>$ be the BSTs produced by split(x, T). If T is a random BST containing the set of keys S, then $T_<$ and $T_>$ are RBBTs containing the keys $S_< = \{y \in S \mid y < x\}$ and $S_> = \{y \in S \mid y > x\}$, respectively.*

Proof Let size(T) $= n > 0$, $x >$ Root(T).key, we will show that for any $z \in S_<$, the probability that z is the root of $T_<$ is $1/m$ where $m =$ size($T_<$). In fact,

$$\begin{aligned} & \Pr(\ z \text{ is root of } T_< \mid \text{root of } T \text{ is less than } x) \\ = \ & \frac{\Pr(\ z \text{ is root of } T_< \text{ and root of } T \text{ is less than } x)}{\Pr(\text{root of } T \text{ is less than } x)} \\ = \ & \frac{1/n}{m/n} = \frac{1}{m}. \end{aligned}$$

The independence of $T_<$ and $T_>$ follows from the independence of L and R of T and by induction.

We are now ready to prove that randomness is preserved by insertion. Specifically,

THEOREM 13.16 *Let T be a RBST for the set of keys S and $x \notin S$, and assume that insert(s, T) produces a tree, say T' for $S \cup \{x\}$. Then, T' is a RBST.*

Proof A key $y \in S$ will be at the root of T' iff

1) y is at root of T.
2) x is not inserted at the root of T'

As 1) and 2) are independent events with probability $\frac{1}{n}$ and $\frac{n}{n+1}$, respectively, it follows that Prob(y is at root of T') $= \frac{1}{n} \cdot \frac{n}{n+1} = \frac{1}{n+1}$. The key x can be at the root of T' only when *insert*(x, T) invokes *insert-at-root*(x, T) and this happens with probability $\frac{1}{n+1}$. Thus, any key in $S \cup \{x\}$ has the probability of $\frac{1}{n+1}$ for becoming the root of T'. The independence of left and right subtrees of T' follows from independence of left and right subtrees of T, induction, and the previous theorem.

13.7.2 Deletion in RBST

Suppose $x \in T$ and let T_x denote the subtree of T rooted at x. Assume that L and R are the left and right subtrees of T_x. To delete x, we build a new BST $T'_x =$ Join(L, R)

containing the keys in L and R and replace T_x by T'_x. Since pseudocode for deletion is easy to write, we omit the details.

We shall take a closer look at the details of the *Join* routine. We need couple of more notations to describe the procedure *Join*. Let L_l and L_r denote the left and right subtrees of L and R_l and R_r denote the left and right subtrees of R, respectively. Let a denote the root of L and b denote the root of R. We select either a or b to serve as the root of T'_x with appropriate probabilities. Specifically, the probability for a becoming the root of T'_x is $\frac{m}{m+n}$ and for b it is $\frac{n}{n+m}$ where $m = \text{size}(L)$ and $n = \text{size}(R)$.

If a is chosen as the root, the its left subtree L_l is not modified while its right subtree L_r is replaced with *Join*(L_r, R). If b is chosen as the root, then its right subtree R_r is left intact but its left subtree R_l is replaced with *Join*(L, R_l). The join of an empty tree with another tree T is T it self and this is the condition used to terminate the recursion.

```
Algorithm Join(L,R)
-- L and R are RBSTs with roots a and b and size m and n respectively.
-- All keys in L are strictly smaller than all keys in R.
-- $L_l$ and $L_r$ respectively denote the left and right subtree of L.
-- $R_l$ and $R_r$ are similarly defined for R.

1. If ( L is NULL) return R.
2. If ( R is NULL) return L.
3. Generate a random integer i in the range [0, n+m-1].
4. If ( i < m )   {* the probability for this event is m/(n+m).*}
      L_r = Join(L_l,R);
      return L;
   else           {* the probability for this event is n/(n+m).*}
      R_l = Join(L,R_l);
      return R;
```

It remains to show that *Join* of two RBSTs produces RBST and deletion preserves randomness in RBST.

THEOREM 13.17 *The Algorithm Join(L,R) produces a RBST under the conditions stated in the algorithm.*

Proof We show that any key has the probability of $1/(n+m)$ for becoming the root of the tree output by *Join(L,R)*. Let x be a key in L. Note that x will be at the root of *Join(L,R)* iff

- x was at the root of L before the *Join* operation, and,
- The root of L is selected to serve as the root of the output tree during the *Join* operation.

The probability for the first event is $1/m$ and the second event occurs with probability $m/(n+m)$. As these events are independent, it follows that the probability of x at the root of the output tree is $\frac{1}{m} \cdot \frac{m}{n+m} = \frac{1}{n+m}$. A similar reasoning holds good for the keys in R.

Finally,

THEOREM 13.18 *If T is a RBST for a set K of keys, then Delete(x,T) outputs a RBST for $K - \{x\}$.*

Proof We sketch only the chance counting argument. The independence of the subtrees follows from the properties of Join operation and induction. Let $T' = Delete(x, T)$. Assume that $x \in K$ and size of K is n. We have to prove that for any $y \in K, y \neq x$, the probability for y in root of T' is $1/(n-1)$. Now, y will be at the root of T' iff either x was at the root of T and its deletion from T brought y to the root of T' or y was at the root of T(so that deletion of x from T did not dislocate y). The former happens with a probability $\frac{1}{n} \cdot \frac{1}{n-1}$ and the probability of the later is $\frac{1}{n}$. As these events are independent, we add the probabilities to obtain the desired result.

13.8 Bibliographic Remarks

In this chapter we have discussed two randomized data structures for Dictionary ADT. Skip Lists are introduced by Pugh in 1990 [6]. A large number of implementations of this structure by a number of people available in the literature, including the one by the inventor himself. Sedgewick gives an account of the comparison of the performances of Skip Lists with the performances of other balanced tree structures [10]. See [7] for a discussion on the implementation of other typical operations such as merge. Pugh argues how Skip Lists are more efficient in practice than balanced trees with respect to Dictionary operations as well as several other operations. Pugh has further explored the possibilities of performing concurrent operations on Skip Lists in [8]. For a more elaborate and deeper analysis of Skip Lists, refer the PhD thesis of Papadakis [4]. In his thesis, he has introduced deterministic skip lists and compared and contrasted the same with a number of other implementations of Dictionaries. Sandeep Sen [5] provides a crisp analysis of the structural properties of Skip Lists. Our analysis presented in this chapter is somewhat simpler than Sen's analysis and our bounds are derived based on different tail estimates of the random variables involved.

The randomized binary search trees are introduced by Martinez and Roura in their classic paper [3] which contains many more details than discussed in this chapter. In fact, we would rate this paper as one of the best written papers in data structures. Seidel and Aragon have proposed a structure called probabilistic priority queues [11] and it has a comparable performance. However, the randomness in their structure is achieved through randomly generated real numbers(called priorities) while the randomness in Martinez and Roura's structure is inherent in the tree itself. Besides this being simpler and more elegant, it solves one of the outstanding open problems in the area of search trees.

References

[1] T.Hagerup and C.Rub, *A guided tour of Chernoff bounds*, Information processing Letters, 33:305-308, 1990.

[2] H.M.Mahmoud, *Evolution of Random Search Trees*, Wiley Interscience, USA, 1992.

[3] C.Martinez and S.Roura, *Randomized Binary search Trees*, Journal of the ACM, 45:288-323, 1998.

[4] Th.Papadakis, *Skip List and probabilistic analysis of algorithms*, PhD Thesis, University of Waterloo, Canada, 1993. (Available as Technical Report CS-93-28).

[5] S.Sen, *Some observations on Skip Lists*, Information Processing Letters, 39:173-176, 1991.

[6] W.Pugh, *Skip Lists: A probabilistic alternative to balanced trees*, Comm.ACM, 33:668-676, 1990.

[7] W.Pugh, *A Skip List cook book*, Technical report, CS-TR-2286.1, University of Maryland, USA, 1990.

[8] W.Pugh, *Concurrent maintenance of Skip Lists*, Technical report, CS-TR-2222, University of Maryland, USA, 1990.

[9] P.Raghavan and R.Motwani, *Randomized Algorithms*, Cambridge University Press, 1995.

[10] R.Sedgewick, *Algorithms*, Third edition, Addison-Wesley, USA, 1998.

[11] R.Seidel and C.Aragon, *Randomized Search Trees*, Algorithmica, 16:464-497,1996.

14

Trees with Minimum Weighted Path Length

Wojciech Rytter
New Jersey Institute of Technology and Warsaw University

14.1 Introduction

The concept of the "weighted path length" is important in data compression and searching. In case of data compression lengths of paths correspond to lengths of code-words. In case of searching they correspond to the number of elementary searching steps. By a *length of a path* we mean usually its number of edges.

Assume we have n weighted items, where w_i is the non-negative *weight* of the i^{th} item. We denote the sequence of weights by $S = (w_1 \dots w_n)$. We adopt the convention that the items have unique names. When convenient to do so, we will assume that those names are the positions of items in the list, namely integers in $[1 \dots n]$.

We consider a binary tree T, where the items are placed in vertices of the trees (in leaves only or in every node, depending on the specific problem). We define the minimum weighted path length (cost) of the tree T as follows:

$$cost(T) = \sum_{i=1}^{n} w_i level_T(i)$$

where $level_T$ is the *level function* of T, *i.e.*, $level_T(i)$ is the level (or depth) of i in T, defined to be the length of the path in T from the root to i.

In some special cases (lopsided trees) the edges can have different lengths and the path length in this case is the sum of individual lengths of edges on the path.

In this chapter we concentrate on several interesting algorithms in the area:

- Huffman algorithm constructing optimal prefix-free codes in time $O(n \log n)$, in this case the items are placed in leaves of the tree, the original order of items can be different from their order as leaves;
- A version of Huffman algorithm which works in $O(n)$ time if the weights of items are sorted
- Larmore-Hirschberg algorithm for optimal height-limited Huffman trees working in time $O(n \times L)$, where L is the upper bound on the height, it is an interesting algorithm transforming the problem to so called "coin-collector", see [21].
- Construction of optimal binary search trees (OBST) in $O(n^2)$ time using certain property of monotonicity of "splitting points" of subtrees. In case of OBST every node (also internal) contains exactly one item. (Binary search trees are defined in Chapter 3.)
- Construction of optimal alphabetic trees (OAT) in $O(n \log n)$ time: the Garsia-Wachs algorithm [11]. It is a version of an earlier algorithm of Hu-Tucker [12, 18] for this problem. The correctness of this algorithm is nontrivial and this algorithm (as well as Hu-Tucker) and these are the most interesting algorithms in the area.
- Construction of optimal lopsided trees, these are the trees similar to Huffman trees except that the edges can have some lengths specified.
- Short discussion of parallel algorithms

Many of these algorithms look "mysterious", in particular the Garsia-Wachs algorithm for optimal alphabetic trees. This is the version of the Hu-Tucker algorithm. Both algorithms are rather simple to understand in how they work and their complexity, but correctness is a complicated issue.

Similarly one can observe a mysterious behavior of the Larmore-Hirschberg algorithm for height-limited Huffman trees. Its "mysterious" behavior is due to the strange reduction to the seemingly unrelated problem of the *coin collector*.

The algorithms relating the cost of binary trees to shortest paths in certain graphs are also not intuitive, for example the algorithm for lopsided trees, see [6], and parallel algorithm for alphabetic trees, see [23]. The efficiency of these algorithms relies on the *Monge property* of related matrices. Both sequential and parallel algorithms for Monge matrices are complicated and interesting.

The area of weighted paths in trees is especially interesting due to its applications (compression, searching) as well as to their relation to many other interesting problems in combinatorial algorithmics.

14.2 Huffman Trees

Assume we have a text x of length N consisting of n different letters with repetitions. The alphabet is a finite set Σ. Usually we identify the i-th letter with its number i. The letter i appears w_i times in x. We need to encode each letter in binary, as $h(a)$, where h is a morphism of alphabet Σ into binary words, in a way to minimize the total length of encoding and guarantee that it is uniquely decipherable, this means that the extension of

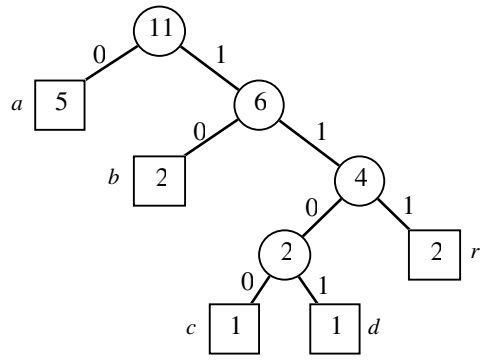

FIGURE 14.1: A Huffman tree T for the items a, b, c, d, r and the weight sequence $S = (5, 2, 1, 1, 2)$. The numbers in internal nodes are sums of weights of leaves in corresponding subtrees. Observe that weighted path length of the tree is the total sum of values in internal nodes. Hence $\mathit{HuffmanCost}(S) = 2 + 4 + 6 + 11 = 23$.

the morphism h to all words over Σ is one-to-one. The words $h(a)$, where $a \in \Sigma$, are called codewords or codes.

The special case of uniquely decipherable codes are *prefix-free* codes: none of the code is a prefix of another one. The prefix-free code can be represented as a binary tree, with left edges corresponding to zeros, and right edge corresponding to ones.

Let $S = \{w_1, w2, \ldots, w_n\}$ be the sequence of weights of items. Denote by $\mathit{HuffmanCost}(S)$ the total cost of minimal encoding (weighted sum of lengths of code-words) and by $\mathrm{HT}(S)$ the tree representing an optimal encoding. Observe that several different optimal trees are possible. The basic algorithm is a greedy algorithm designed by Huffman, the corresponding trees and codes are called Huffman trees and Huffman codes.

Example Let $\mathit{text} = \mathit{abracadabra}$. The number of occurrences of letters are

$$w_a = 5, w_b = 2, w_c = 1, w_d = 1, w_r = 2.$$

We treat letters as items, and the sequence of weights is:

$$S = (5, 2, 1, 1, 2)$$

An optimal tree of a prefix code is shown in Figure 14.1. We have, according to the definition of weighted path length:

$$\mathit{HuffmanCost}(S) = 5 * 1 + 2 * 2 + 1 * 4 + 1 * 4 + 2 * 3 = 23$$

The corresponding prefix code is:

$$h(a) = 0, h(b) = 10, h(c) = 1100, h(d) = 1101, h(r) = 111.$$

We can encode the original text *abracadabra* using the codes given by paths in the prefix tree. The coded text is then 01011101100011010101110, that is a word of length 23.

If for example the initial code words of letters have length 5, we get the compression ratio $55/23 \approx 2.4$.

14.2.1 $O(n \log n)$ Time Algorithm

The basic algorithm is the *greedy* algorithm given by Huffman. In this algorithm we can assume that two items of smallest weight are at the bottom of the tree and they are sons of a same node. The *greedy* approach in this case is to minimize the cost *locally*.

Two smallest weight items are *combined* into a single *package* with weight equal to the sum of weights of these small items. Then we proceed recursively. The following observation is used.

Observation Assume that the numbers in internal nodes are sums of weights of leaves in corresponding subtrees. Then the total weighted path length of the tree is the total sum of values in internal nodes.

Due to the observation we have for $|S| > 1$:

$$\mathit{HuffmanCost}(S) \;=\; \mathit{HuffmanCost}(S - \{u, w\}) + u + w,$$

where u, w are two minimal elements of S. This implies the following algorithm, in which we assume that initially S is stored in a min-priority queue. The algorithm is presented below as a recursive function $\mathit{HuffmanCost}(S)$ but it can be easily written without recursion. The algorithm computes only the total cost.

THEOREM 14.1 *Huffman algorithm constructs optimal tree in $O(n \log n)$ time*

Proof In an optimal tree we can exchange two items which are sons of a same father at a bottom level with two smallest weight items. This will not increase the cost of the tree. Hence there is an optimal tree with two smallest weight items being sons of a same node. This implies correctness of the algorithm.

The complexity is $O(n \log n)$ since each operation in the priority queue takes $O(\log n)$ time and there are $O(n)$ operations of Extract-Min.

```
function HuffmanCost(S)
{ Huffman algorithm: recursive version}
{ computes only the cost of minimum weighted tree }
  1. if S contains only one element u then return 0;
  2. u = Extract Min(S); w = ExtractMin(S);
  3. insert(u+w, S);
  4. return HuffmanCost(S) +u+w
```

The algorithm in fact computes only the cost of Huffman tree, but the tree can be created on-line in the algorithm. Each time we combine two items we create their father and create links son-father. In the course of the algorithm we keep a forest (collection of trees). Eventually this becomes a single Huffman tree at the end.

14.2.2 Linear Time Algorithm for Presorted Sequence of Items

There is possible an algorithm using "simple" queues with constant time operations of inserting and deleting elements from the queue if the items are *presorted*.

THEOREM 14.2 *If the weights are already sorted then the Huffman tree can be constructed in linear time.*

Proof If we have sorted queues of remaining original items and remaining newly created items (packages) then it is easy to see that two smallest items can be chosen from among 4 items, two first elements of each queue. This proves correctness.

Linear time follows from constant time costs of single queue operations.

Linear-Time Algorithm

{ linear time computation of the cost for presorted items }

1. initialize empty queues Q, S;
   total_cost = 0;
2. place original n weights into nondecreasing order into S;
   the smallest elements at the front of S;
3. **while** $|Q| + |S| > 2$ **do** {
   let u, w be the smallest elements chosen from the
   first two elements of Q and of S;
   remove u, w from $Q \cup S$; insert$(u + w,\ Q)$;
   $total\_cost\ =\ total\_cost\ +\ (u + w)$;}
4. **return** total_cost

14.2.3 Relation between General Uniquely Decipherable Codes and Prefix-free Codes

It would seem that, for some weight sequences, in the class of uniquely decipherable codes there are possible codes which beat every Huffman (prefix-free) code. However it happens that prefix-free codes are optimal within the whole class of uniquely decipherable codes. It follows immediately from the next three lemmas.

LEMMA 14.1 For each full (each internal node having exactly two sons) binary tree T with leaves $1 \ldots n$ we have:

$$\sum_{i=1}^{n} 2^{-level_T(i)} = 1$$

Proof Simple induction on n.

LEMMA 14.2 For each uniquely decipherable code S with word lengths $\ell_1, \ell_2, \ldots, \ell_k$ on the alphabet $\{0, 1\}$ we have :

$$\sum_{i=1}^{k} 2^{-\ell_i} \leq 1$$

Proof For a set W of words on the alphabet $\{0,1\}$ define:

$$C(W) \;=\; \sum_{x \in W} 2^{-|x|}$$

We have to show that $C(S) \leq 1$. Let us first observe the following simple fact.

Observation.
If S is uniquely decipherable then $C(S)^n \;=\; C(S^n)$ for all $n \geq 1$.

The proof that $C(S) \leq 1$ is now by contradiction, assume $C(S) > 1$. Let c be the length of the longest word in S. Observe that

$$C(\Sigma^k) = 1 \text{ for each } k,\; C(S^n) \leq C(\{x \in \Sigma^* \;:\; 1 \leq |x| \leq cn\;\}) \;=\; cn$$

Denote $q = C(S)$. Then we have:

$$C(S)^n = q^n \leq cn$$

For $q > 1$ this inequality is not true for all n, since

$$\lim\; q^n/(cn) \;=\; +\infty \text{ if } q > 1.$$

Therefore it should be $q \leq 1$ and $C(S) \leq 1$. This completes the proof.

LEMMA 14.3 [Kraft's inequality] There is a prefix code with word lengths $\ell_1, \ell_2, \ldots, \ell_k$ on the alphabet $\{0,1\}$ iff

$$\sum_{i=1}^{k} 2^{-\ell_i} \leq 1 \tag{14.1}$$

Proof It is enough to show how to construct such a code if the inequality holds. Assume the lengths are sorted in the increasing order. Assume we have a (potentially) infinite full binary tree. We construct the next codeword by going top-down, starting from the root. We assign to the i-th codeword, for $i = 1, 2, 3, \ldots, k$, the lexicographically first path of length ℓ_i, such that the bottom node of this path has not been visited before. It is illustrated in Figure 14.2. If the path does not exist then this means that in this moment we covered with paths a full binary tree, and the actual sum equals 1. But some other lengths remained, so it would be :

$$\sum_{i=1}^{k} 2^{-\ell_i} > 1$$

a contradiction. This proves that the construction of a prefix code works, so the corresponding prefix-free code covering all lengths exists. This completes the proof.

The lemmas imply directly the following theorem.

THEOREM 14.3 *A uniquely decipherable code with prescribed word lengths exists iff a prefix code with the same word lengths exists.*

We remark that the problem of testing unique decipherability of a set of codewords is complete in nondeterministic logarithmic space, see [47].

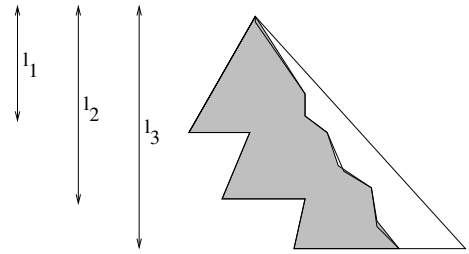

FIGURE 14.2: Graphical illustration of constructing prefix-free code with prescribed lengths sequence satisfying Kraft inequality.

14.2.4 Huffman Codes and Entropy

The performance of Huffman codes is related to a measure of information of the source text, called the *entropy* (denoted by $\mathcal{E}$) of the alphabet. Let w_a be n_a/N, where n_a is the number of occurrences of a in a given text of length N. In this case the sequence S of weights w_a is normalized $\sum_{i=1}^{n} w_i = 1$.

The quantity w_a can be now viewed as the probability that letter a occurs at a given position in the text. This probability is assumed to be independent of the position. Then, the entropy of the alphabet according to w_a's is defined as

$$\mathcal{E}(A) = -\sum_{a \in A} w_a \log w_a.$$

The entropy is expressed in bits (log is a base-two logarithm). It is a lower bound of the average length of the code words $h(a)$,

$$m(A) = \sum_{a \in A} w_a . |h(a)|.$$

Moreover, Huffman codes give the best possible approximation of the entropy (among methods based on coding of the alphabet). This is summarized in the following theorem whose proof relies on the inequalities from Lemma 14.2.

THEOREM 14.4 *Assume the weight sequence A of weights is normalized. The total cost $m(A)$ of any uniquely decipherable code for A is at least $\mathcal{E}(A)$, and we have*

$$\mathcal{E}(A) \leq \mathit{HuffmanCost}(A) \leq \mathcal{E}(A) + 1.$$

14.2.5 Huffman Algorithm for t-ary Trees

An important generalization of Huffman algorithm is to t-ary trees. Huffman algorithm generalizes to the construction of prefix codes on alphabet of size $t > 2$. The trie of the code is then an *almost full* t-ary tree.

We say that t-ary tree is almost full if all internal nodes have exactly t sons, except possibly one node, which has less sons, in these case all of them should be leaves (let us call this one node a *defect* node).

We perform similar algorithm to Huffman method for binary trees, except that each time we select t items (original or combined) of smallest weight.

There is one technical difficulty. Possibly we start by selecting a smaller number of items in the first step. If we know t and the number of items then it is easy to calculate number q of sons of the defect node, for example if $t = 3$ and $n = 8$ then the defect node has two sons. It is easy to compute the number q of sons of the defect node due to the following simple fact.

LEMMA 14.4 If T is a full t-ary tree with m leaves then $m \ modulo \ (t-1) \ = \ 1$.

We start the algorithm by combining q smallest items. Later each time we combine exactly t values. To simplify the algorithm we can add the smallest possible number of dummy items of weigh zero to make the tree full t-ary tree.

14.3 Height Limited Huffman Trees

In this section only, for technical reason, we assume that the length of the path is the number of its vertices. For a sequence S of weights the total cost is changed by adding the sum of weights in S.

Assume we have the same problem as in the case of Huffman trees with additional restriction that the height of the tree is limited by a given parameter L. A beautiful algorithm for this problem has been given by Larmore and Hirschberg, see [16].

14.3.1 Reduction to the Coin Collector Problem

The main component of the algorithm is the reduction to the following problem in which the crucial property play powers of two. We call a real number *dyadic* iff it has a finite binary representation.

Coin Collector problem:

Input: A set I of m items and dyadic number X, each element of I has a *width* and a *weight*, where each width is a (possibly negative) power of two, and each weight is a positive real number.

Output: $CoinColl(I, X)$ - the minimum total weight of a subset $S \subseteq I$ whose widths sum to X.

The following trivial lemma plays important role in the reduction of height limited tree problem to the Coin Collector problem.

LEMMA 14.5 Assume T is a full binary tree with n leaves, then

$$\sum_{v \in T} 2^{-level_T(v)} \ = \ n+1$$

Assume we have Huffman coding problem with n items with the sequence of weights weights $W \ = \ w_1, w_2, \ldots, w_n$. We define an instance of the Coin Collector problem as follows:

- $I_W \ = \ \{(i,l) \ : \ i \in [1 \ldots n], \ l \in [1, \ldots L]$,
- $width(i,l) \ = \ 2^{-l}, \quad weight(i,l) \ = \ w_i$ for each i, l
- $X_w \ = \ n+1$.

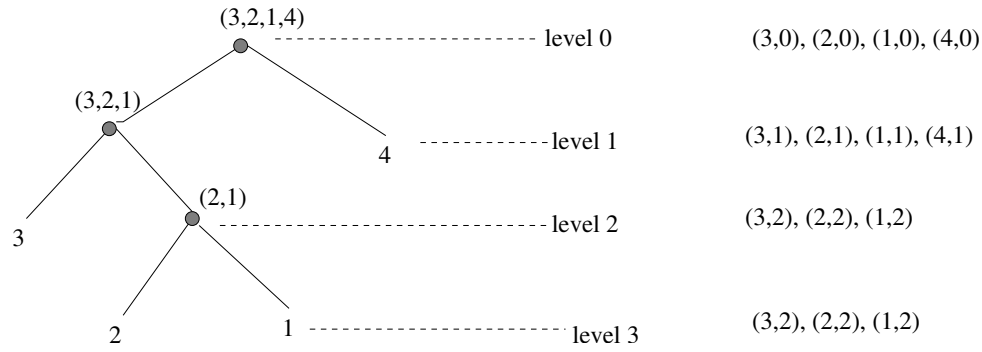

FIGURE 14.3: A Huffman tree T for the items $1, 2, 3, 4$ of height limited by 4 and the corresponding solution to the Coin Collector problem. Each node of the tree can be treated as a *package* consisting of leaves in the corresponding subtree. Assume $weight(i) = i$. Then in the corresponding Coin Collector problem we have $weight(i, h) = i,\ width(i, h) = 2^{-h}$.

The intuition behind this strange construction is to view nodes as packages consisting of of original items (elements in the leaves). The internal node v which is a *package* consisting of (leaves in its subtree) items $i_1, i_2, \ldots, i_k$ can be treated as a set of coins $(i_1, h), (i_2, h), \ldots (i_k, h)$, where h is the level of v, and $weight(i_j, h) = weight(i_j)$. The total weight of the set of coins is the cost of the Huffman tree.

Example Figure 14.3 shows optimal Huffman tree for $S = (1, 2, 3, 4)$ with height limited by 4, and the optimal solution to the corresponding Coin Collector problem. The sum of widths of the coins is 4+1, and the total weight is minimal. It is the same as the cost of the Huffman tree on the left, assuming that leaves are also contributing their weights (we scale cost by the sum of weights of S).

Hirschberg and Larmore, see [16], have shown the following fact.

LEMMA 14.6 The solution $CoinColl(I_W, X_W)$ to the Coin Collector Problem is the cost of the optimal L-height restricted Huffman tree for the sequence W of weights.

14.3.2 The Algorithm for the Coin Collector Problem

The height limited Huffman trees problem is thus reduced to the Coin Collector Problem. The crucial point in the solution of the latter problem is the fact that weights are powers of two.

Denote $MinWidth(X)$ to be the smallest power of two in binary representation of number X. For example $MinWidth(12) = 4$ since $12 = 8 + 4$. Denote by $MinItem(I)$ the item with the smallest width in I.

LEMMA 14.7 If the items are sorted with respect to the weight then the Coin Collector problem can be solved in linear time (with respect to the total number $|I|$ of coins given in the input).

Proof The recursive algorithm for the Coin Collector problem is presented below as a recursive function $CoinColl(I, X)$. There are several cases depending on the relation between the smallest width of the item and minimum power of two which constitutes the binary representation of X. In the course of the algorithm the set of weights shrinks as well as the size of X. The linear time implementation of the algorithm is given in [21].

```
function CC(I, X); {Coin Collector Problem}
{compute nnimal weight of a subset of I of total width X
x := MinItem(X); r := width(x);
if r > MinWidth(X) then
      no solution exists  else
if r = MinWidth(X) then
      return CC(I − {x}, X − r) + weight(x)  else
if r < MinWidth(X) and there is only one item of width r then
      return CC(I − {x}, X)  else
          let x, x' be two items of smallest weight and width r,
          create new item y such that
              width(y) = 2r, weight(y) = weight(x) + weight(x');
          return CC(I − {x, x'} ∪ {y}, X)
```

The last two lemmas imply the following fact.

THEOREM 14.5 *The problem of the Huffman tree for n items with height limited by L can be solved in $O(n \cdot L)$ time.*

Using complicated approach of the Least Weight Concave Subsequence the complexity has been reduced to $n\sqrt{L \log n} + n \log n$ in [1]. Another small improvement is by Schieber [49]. An efficient approximation algorithm is given in [40–42]. The dynamic algorithm for Huffman trees is given in [50].

14.4 Optimal Binary Search Trees

Assume we have a sequence of $2n + 1$ weights (nonnegative reals)

$$\alpha_0, \beta_1, \alpha_1, \beta_2, \ldots, \alpha_{n-1}, \beta_n, \alpha_n.$$

Let Tree$(\alpha_0, \beta_1, \alpha_1, \beta_2, \ldots, \alpha_{n-1}, \beta_n, \alpha_n)$ be the set of all full binary weighted trees with n internal nodes, where the i-th internal node (in the in-order) has the weight β_i, and the i-th external node (the leaf, in the left-to-right order) has the weight α_i. The in-order traversal results if we visit all nodes in a recursive way, first the left subtree, then the root, and afterwards the right subtree.

If T is a binary search tree then define the *cost* of T as follows:

$$cost(T) = \sum_{v \in T} level_T(v) \cdot weight(v).$$

Let $OPT(\alpha_0, \beta_1, \ldots, \alpha_{n-1}, \beta_n, \alpha_n)$ be the set of trees Tree$(\alpha_0, \beta_1, \ldots, \alpha_{n-1}, \beta_n, \alpha_n)$ whose cost is minimal.

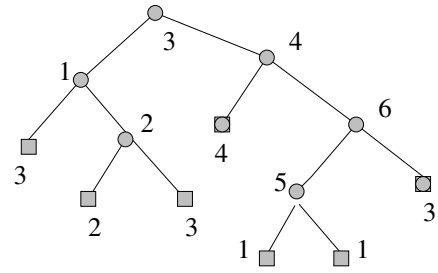

FIGURE 14.4: A binary search tree for the sequences: $\beta = (\beta_1, \beta_2, \ldots, \beta_6) = (1, 2, 3, 4, 5, 6)$, $\alpha = (\alpha_0, \alpha_1, \ldots \alpha_6) = (3, 2, 3, 4, 1, 1, 3)$. We have $cut(0, 6) = 3$.

We use also terminology from [35]. Let $K_1, \ldots K_n$ be a sequence of n weighted items (keys), which are to be placed in a binary search tree. We are given $2n + 1$ weights (probabilities): $q_0, p_1, q_1, p_2, q_2, p_3, \ldots, q_{n-1}, p_n, q_n$ where

- p_i is the probability that K_i is the search argument;
- q_i is the probability that the search argument lies between K_i and K_{i+1}.

The OBST problem is to construct an optimal binary search tree, the keys K_i's are to be stored in internal nodes of this binary search tree and in its external nodes special items are to be stored. The i-th special item K'_i corresponds to all keys which are strictly between K_i and K_{i+1}. The binary search tree is a full binary tree whose nodes are labeled by the keys. Using the abstract terminology introduced above the OBST problem consists of finding a tree $T \in OPT(q_0, p_1, q_1, p_2, \ldots, q_{n-1}, p_n, q_n)$, see an example tree in Figure 14.4.

Denote by $obst(i, j)$ the set $OPT(q_i, p_{i+1}, q_{i+1}, \ldots, q_{j-1}, p_j, q_j)$. Let $cost(i, j)$ be the cost of a tree in $obst(i, j)$, for $i < j$, and $cost(i, i) = q_i$. The sequence $q_i, p_{i+1}, q_{i+1}, \ldots, q_{j-1}, p_j, q_j$ is here the subsequence of $q_0, p_1, q_1, p_2, \ldots, q_{n-1}, p_n, q_n$, consisting of some number of consecutive elements which starts with q_i and ends with q_j. Let

$$w(i, j) = q_i + p_{i+1} + q_{i+1} + \ldots + q_{j-1} + p_j + q_j.$$

The dynamic programming approach to the computation of the OBST problem relies on the fact that the subtrees of an optimal tree are also optimal. If a tree $T \in obst(i, j)$ contains in the root an item K_k then its left subtree is in $obst(i, k-1)$ and its right subtree is in $obst(k, j)$. Moreover, when we join these two subtrees then the contribution of each node increases by one (as one level is added), so the increase is $w(i, j)$. Hence the costs obey the following *dynamic programming recurrences* for $i < j$:

$$cost(i, j) = \min\{cost(i, k-1) + cost(k, j) + w(i, j) \ : \ i < k \leq j \}.$$

Denote the smallest value of k which minimizes the above equation by $cut(i, j)$. This is the first point giving an optimal decomposition of $obst(i, j)$ into two smaller (son) subtrees. Optimal binary search trees have the following crucial property (proved in [34], see the figure for graphical illustration)

$$\textit{monotonicity property:} \quad i \leq i' \leq j \leq j' \Longrightarrow cut(i, j) \leq cut(i', j').$$

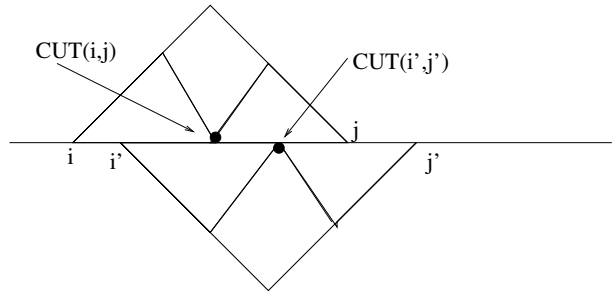

FIGURE 14.5: Graphical illustration of the monotonicity property of cuts.

The property of monotonicity, the cuts and the quadratic algorithm for the OBST were first given by Knuth. The general dynamic programming recurrences were treated by Yao [52], in the context of reducing cubic time to quadratic.

THEOREM 14.6 *Optimal binary search trees can be computed in $O(n^2)$ time.*

Proof The values of $cost(i, j)$ are computed by tabulating them in an array. Such tabulation of costs of smaller subproblems is the basis of the *dynamic programming* technique. We use the same name *cost* for this array. It can be computed in $O(n^3)$ time, by processing diagonal after diagonal, starting with the central diagonal.

In case of optimal binary search trees this can be reduced to $O(n^2)$ using additional tabulated values of the cuts in table *cut*. The k-th diagonal consists of entries i, j such that $j - i = k$. If we have computed cuts for k-th diagonal then for (i, j) on the $(k + 1)$-th diagonal we know that

$$cut(i, j - 1) \leq cut(i, j) \leq cut(i + 1, j)$$

Hence the total work on the $(k + 1)$-th diagonal is proportional to the sum of telescoping series:

$$cut(1, k + 1) - cut(0, k) + cut(2, k + 2) - cut(1, k + 1) + \\ cut(3, k + 3) - cut(2, k + 2) + \ldots cut(n - k, k) - cut(n - k - 1, k - 1),$$

which is $O(n)$. Summing over all diagonals gives quadratic total time to compute the tables of cuts and costs. Once the table $cost(i, j)$ is computed then the construction of an optimal tree can be done in quadratic time by tracing back the values of *cut*s.

14.4.1 Approximately Optimal Binary Search Trees

We can attempt to reduce time complexity at the cost of slightly increased cost of the constructed tree. A common-sense approach would be to insert the keys in the order of decreasing frequencies. However this method occasionally can give quite bad trees.

Another approach would be to choose the root so that the total weights of items in the left and right trees are as close as possible. However it is not so good in pessimistic sense.

The combination of this methods can give quite satisfactory solutions and the resulting algorithm can be linear time, see [44]. *Average* subquadratic time has been given in [29].

14.5 Optimal Alphabetic Tree Problem

The alphabetic tree problem looks very similar to the Huffman problem, except that the leaves of the alphabetic tree (read left-to-right) should be in the same order as in the original input sequence. Similarly as in Huffman coding the binary tree must be *full*, *i.e.*, each internal node must have exactly two sons.
The main difficulty is that we cannot localize easily two items which are to be combined.

Assume we have a sequence of n weighted items, where w_i is the non-negative *weight* of the i^{th} item. We write $\alpha = w_1 \dots w_n$. The sequence will be changing in the course of the algorithm.

An alphabetic tree over α is an ordered binary tree T with n leaves, where the i^{th} leaf (in left-to-right order) corresponds to the i^{th} item of The optimal alphabetic tree problem (OAT problem) is to find an alphabetic tree of minimum cost.

The Garsia-Wachs algorithm solves the alphabetic tree problem, it is a version of an earlier algorithm by Hu and Tucker, see [18]. The strangest part of the algorithm is that it permutes α, though the final tree should have the order of leaves the same as the order of items in the original sequence. We adopt the convention that the items of α have unique names, and that these names are preserved when items are moved. When convenient to do so, we will assume that those names are the positions of items in the list, namely integers in $[1 \dots n]$.

14.5.1 Computing the Cost of Optimal Alphabetic Tree

First we show how to compute only the cost of the whole tree, however this computation does not give automatically an optimal alphabetic tree, since we will be permuting the sequence of items. Each time we combine two adjacent items in the current permutation, however these items are not necessarily adjacent in the original sequence, so in any legal alphabetic tree they cannot be sons of a same father.

The alphabetic tree is constructed by reducing the initial sequence of items to a shorter sequence in a manner similar to that of the Huffman algorithm, with one important difference. In the Huffman algorithm, the minimum pair of items are combined, because it can be shown that they are siblings in the optimal tree. If we could identify two adjacent items that are siblings in the optimal alphabetic tree, we could combine them and then proceed recursively. Unfortunately, there is no known way to identify such a pair. Even a minimal pair may not be siblings. Consider the weight sequence (8 7 7 8). The second and the third items are not siblings in any optimal alphabetic tree.

Instead, the HT and GW algorithms, as well as the algorithms of [20, 22, 23, 46], operate by identifying a pair of items that have the same level in the optimal tree. These items are then combined into a single "package," reducing the number of items by one. The details on how this process proceeds differ in the different algorithms. Define, for $1 \le i < n$, the i^{th} *two-sum*:

$$TwoSum(i) = w_i + w_{i+1}$$

A pair of adjacent items $(i, i+1)$ is a *locally minimal pair* (or *lmp* for short) if

$$\begin{array}{rcll} TwoSum(i-1) & \ge & TwoSum(i) & \text{if} \quad i > 1 \\ TwoSum(i) & < & TwoSum(i+1) & \text{if} \quad i \le n-2 \end{array}$$

A locally minimal pair which is currently being processed is called the *active pair*.

The Operator *Move*. If w is any item in a list π of weighted items, define $RightPos(w)$ to be the predecessor of the nearest right larger or equal neighbor of w. If w has no right larger or equal neighbor, define $RightPos(w)$ to be $|\pi|+1$.

Let $Move(w,\pi)$ be the operator that changes π by moving w w is inserted between positions $RightPos(w)-1$ and $RightPos(w)$. For example

$$Move(7,(2,5,7,2,4,9,3,4)\ =\ (2,5,2,4,7,9,3,4)$$

.

```
function GW(π); {π is a sequence of names of items}
{restricted version of the Garsia-Wachs algorithm}
{  computing only the cost of optimal alphabetic tree }
if π = (v) (π consists of a single item)
then return 0
   else
      find any locally minimal pair (u, w) of π
      create a new item x whose weight is weight(u) + weight(w);
      replace the pair u, v by the single item x;
       { the items u, v are removed }
      Move(v, π);
      return GW(π)  + weight(v);
```

Correctness of the algorithm is a complicated issue. There are two simplified proofs, see [19, 30] and we refer to these papers for detailed proof. In [19] only the rightmost minimal pair can be processed each time, while [30] gives correctness of general algorithm when any minimal pair is processed, this is important in parallel computation, when we process simultaneously many such pairs. The proof in [30] shows that correctness is due to *well-shaped* bottom segments of optimal alphabetic trees, this is expressed as a *structural theorem* in [30] which gives more insight into the global structure of optimal alphabetic trees.

For $j > i+1$ denote by $\pi_{i,j}$ the sequence π in which elements $i, i+1$ are moved just before left of j.

THEOREM 14.7 [Correctness of the GW algorithm]
Let $(i,i+1)$ be a locally minimal pair and $RightPos(i,i+1)=j$, and let T' be a tree over the sequence $\pi_{i,j}$, optimal among all trees over $\pi_{i,j}$ in which i, $i+1$ are siblings. Then there is an optimal alphabetic tree T over the original sequence $\pi=(1,\ldots n)$ such that $T\cong T'$.

Correctness can be also expressed as equivalence between some classes of trees.
Two binary trees T_1 and T_2 are said to be **level equivalent** (we write $T_1\cong T_2$) if T_1, and T_2 have the same set of leaves (possibly in a different order) and $level_{T_1}=level_{T_2}$.

Denote by $\mathrm{OPT}(i)$ the set of all alphabetic trees over the leaf-sequence $(1,\ldots n)$ which are optimal among trees in which i and $i+1$ are at the same level. Assume the pair $(i,i+1)$ is locally minimal. Let $\mathrm{OPT}_{moved}(i)$ be the set of all alphabetic trees over the leaf-sequence $\pi_{i,j}$ which are optimal among all trees in which leaves i and $i+1$ are at the same level, where $j=RightPos(i,i+1)$.

Two sets of trees OPT and OPT$'$ are said to be *level equivalent*, written $\mathrm{OPT}\cong\mathrm{OPT}'$, if, for each tree $T\in\mathrm{OPT}$, there is a tree $T'\in\mathrm{OPT}'$ such that $T'\cong T$, and vice versa.

THEOREM 14.8
Let $(i, i+1)$ be a locally minimal pair. Then

(1) $\mathrm{OPT}(i) \cong \mathrm{OPT}_{moved}(i)$.

(2) $\mathrm{OPT}(i)$ *contains an optimal alphabetic tree* T.

(3) $\mathrm{OPT}_{moved}(i)$ *contains a tree* T' *with* i, $i+1$ *as siblings.*

14.5.2 Construction of Optimal Alphabetic Tree

The full Garsia-Wachs algorithm first computes the level tree. This tree can be easily constructed in the function $GW(\pi)$ when computing the cost of alphabetic tree. Each time we sum weights of two items (original or newly created) then we create new item which is their father with the weight being the sum of weights of sons.

Once we have a level tree, the optimal alphabetic tree can be constructed easily in linear time. Figure 14.6, Figure 14.7, and Figure 14.8 show the process of construction the level tree and construction an optimal alphabetic tree knowing the levels of original items.

LEMMA 14.8 Assume we know level of each leaf in an optimal alphabetic tree. Then the tree can be constructed in linear time.

Proof The levels give the *"shape"* of the tree, see Figure 14.8.
Assume $l_1, l_2, l_3, \ldots, l_n$ is the sequence of levels. We scan this sequence from left-to-right until we find *the first* two levels l_i, l_{i+1} which are the same. Then we know that the leaves i and $i+1$ are sons of a same father, hence we link these leaves to a newly created father and we remove these leaves, in the level sequence the pair l_i, l_{i+1} is replaced by a single level $l_i - 1$. Next we check if $l_{i-1} = l_i - 1$, if not we search to the right. We keep the scanned and newly created levels on the stack. The total time is linear.

There are possible many different optimal alphabetic trees for the same sequence, Figure 14.9 shows an alternative optimal alphabetic tree for the same example sequence.

THEOREM 14.9 *Optimal alphabetic tree can be constructed in $O(n \log n)$ time.*

Proof We keep the array of levels of items. The array *level* is global of size $(2n-1)$. Its indices are the names of the nodes, *i.e.*, the original n items and the $(n-1)$ nodes ("packages") created during execution of the algorithm. The algorithm works in quadratic time, if implemented in a naive way. Using priority queues, it works in $O(n \log n)$ time. Correctness follows directly from Theorem 14.7.

14.5.3 Optimal Alphabetic Trees for Presorted Items

We have seen that Huffman trees can be constructed in linear time if the weights are presorted. Larmore and Przytycka, see [22] have shown that slightly weaker similar result holds for alphabetic trees as well:

assume that weights of items are sortable in linear time, then the alphabetic tree problem can be solved in $O(n \log \log n)$ time.

Open problem Is it possible to construct alphabetic trees in linear time in the case when the weights are sortable in linear time?

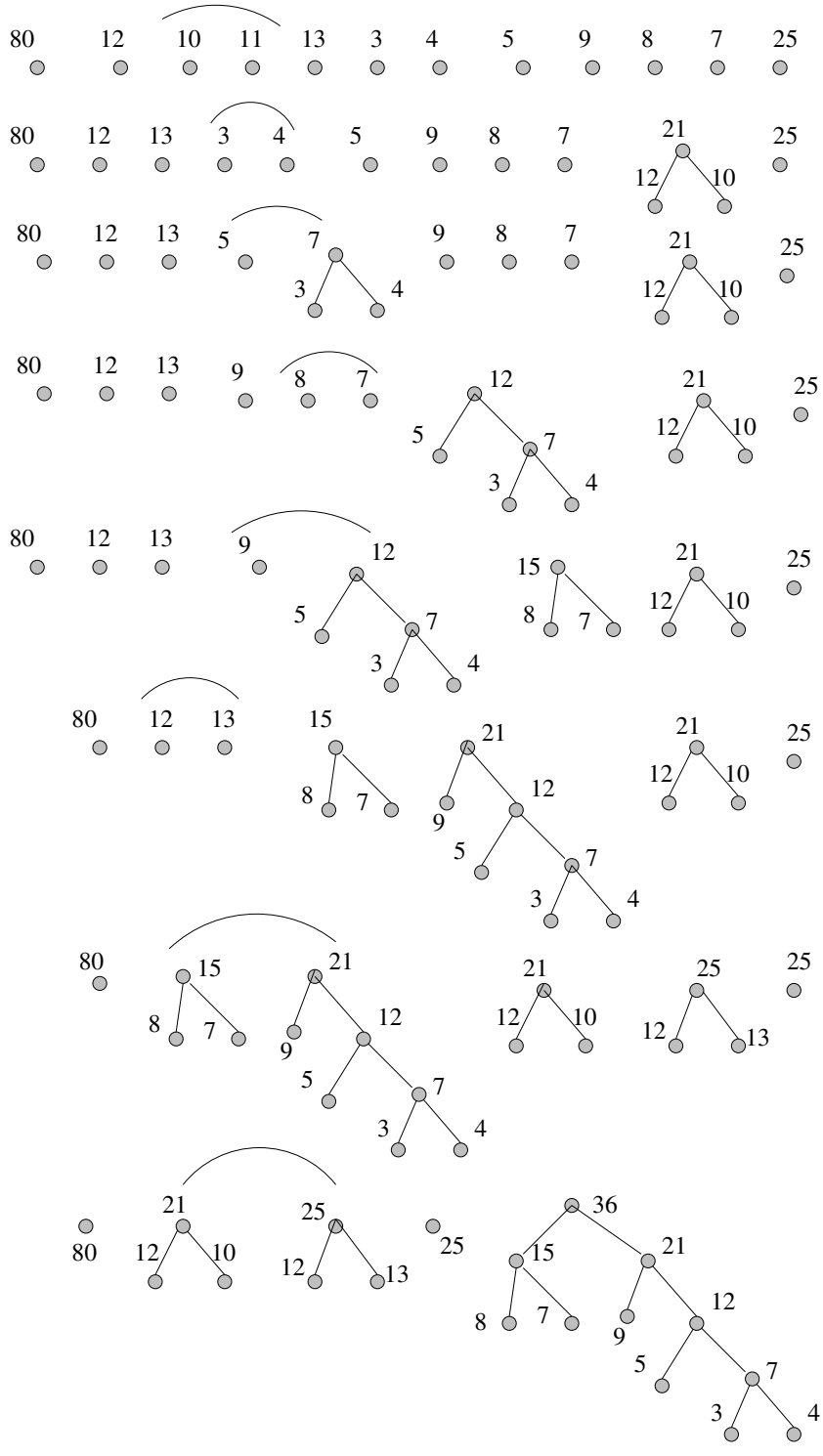

FIGURE 14.6: The first 7 phases of Garsia-Wachs algorithm.

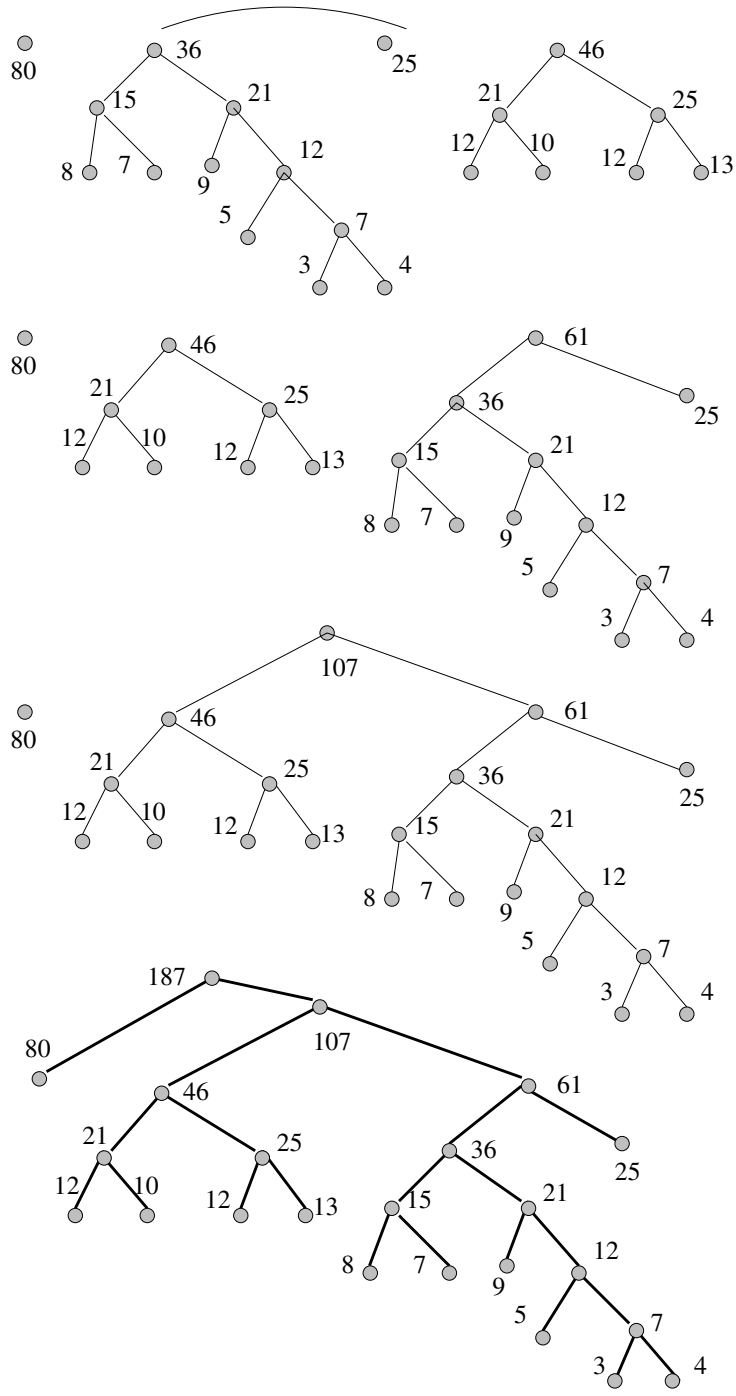

FIGURE 14.7: The last phases of Garsia-Wachs algorithm. The final tree is the level tree (but not alphabetic tree). The total cost (sum of values in internal nodes equal to 538) and the levels of original items are the same as in optimal alphabetic tree. The level sequence for the original sequence (80, 12, 10, 11, 13, 3, 4, 5, 9, 8, 7, 25) is:

$$\mathcal{L} \;=\; (\; 1,\; 4,\; 4,\; 4,\; 4,\; 7,\; 7,\; 6,\; 5,\; 5,\; 3,\;)$$

.

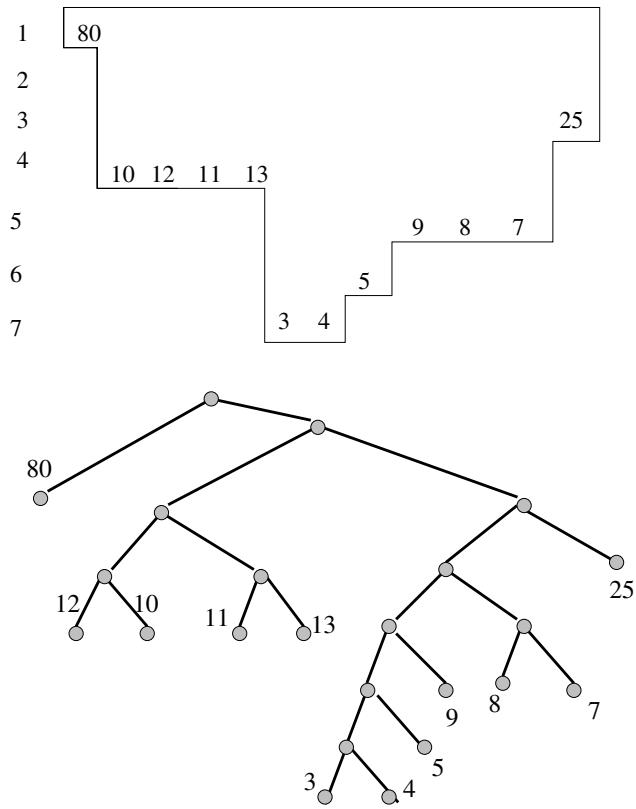

FIGURE 14.8: The shape of the optimal tree given by the level sequence and the final optimal alphabetic tree (cost=538) corresponding to this shape and to the weight sequence (80, 12, 10, 11, 13, 3, 4, 5, 9, 8, 7, 25).

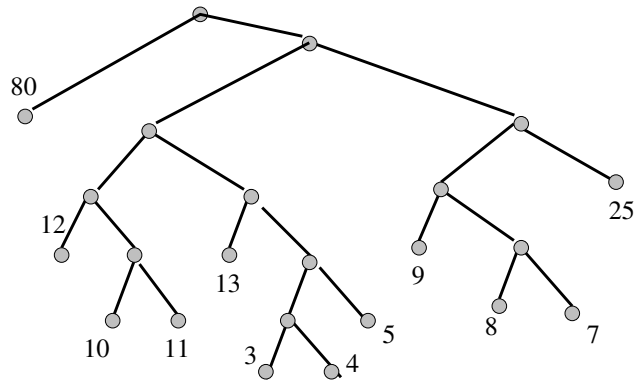

FIGURE 14.9: An alternative optimal alphabetic tree (cost = 538) for the same weight sequence.

14.6 Optimal Lopsided Trees

The problem of finding optimal prefix-free codes for unequal letter costs consists of finding a minimal cost prefix-free code in which the encoding alphabet consists of unequal cost (length) letters, of lengths α and β, $\alpha \leq \beta$. We restrict ourselves here only to binary trees. The code is represented by a *lopsided tree*, in the same way as a Huffman tree represents the solution of the Huffman coding problem. Despite the similarity, the case of unequal letter costs is much harder then the classical Huffman problem; no polynomial time algorithm is known for general letter costs, despite a rich literature on the problem, *e.g.*, [4, 15]. However there are known polynomial time algorithms when α and β are integer constants [15].

The problem of finding the minimum cost tree in this case was first studied by Karp [27] in 1961 who solved the problem by reduction to integer linear programming, yielding an algorithm exponential in both n and β. Since that time there has been much work on various aspects of the problem such as; bounding the cost of the optimal tree, Altenkamp and Mehlhorn [2], Kapoor and Reingold [26] and Savari [8]; the restriction to the special case when all of the weights are equal, Cot [10], Perl Gary and Even [45], and Choi and Golin [9]; and approximating the optimal solution, Gilbert [13]. Despite all of these efforts it is still, surprisingly, not even known whether the basic problem is polynomial-time solvable or in NP-complete.

Golin and Rote [15] describe an $O(n^{\beta+2})$-time dynamic programming algorithm that constructs the tree in a top-down fashion.

This has been improved using a different approach (monotone-matrix concepts, e.g., the *Monge property* and the SMAWK algorithm [7].

THEOREM 14.10 [6]
Optimal lopsided trees can be constructed in $O(n^{\beta})$ time.

This is the the most efficient known algorithm for the case of small β; in practice the letter costs are typically small (*e.g.*, Morse codes).

Recently a scheme of an efficient approximating algorithm has been given.

THEOREM 14.11 [24]
There is a polynomial time approximation scheme for optimal lopsided trees.

14.7 Parallel Algorithms

As a model of parallel computations we choose the *Parallel Random Access Machines* (PRAM), see [14]. From the point of view of parallel complexity two parameters are of interest: parallel time (usually we require polylogarithmic time) and total work (time multiplied by the number of processors).

The sequential greedy algorithm for Huffman coding is quite simple, but unfortunately it appears to be inherently sequential. Its parallel counterpart is much more complicated, and requires a new approach. The global structure of Huffman trees must be explored in depth.

A full binary tree T is said to be *left-justified* if it satisfies the following properties:

1. the depths of the leaves are in non-increasing order from left to right,
2. let u be a left brother of v, and assume that the height of the subtree rooted at v is at least l. Then the tree rooted at u is full at level l, which means that u has 2^l descendants at distance l.

Basic property of left-justified trees

Let T be a left-justified binary tree. Then, T consists of one leftmost branch and the subtrees hanging from this branch have logarithmic height.

LEMMA 14.9 Assume that the weights $w_1, w_2, \ldots, w_n$ are pairwise distinct and in increasing order. Then, there is Huffman tree for $(w_1, w_2, \ldots, w_n)$ that is left-justified.

The left-justified trees are used together with efficient algorithm for the CLWS problem (the *Concave Least Weight Subsequence* problem, to be defined below) to show the following fact.

THEOREM 14.12 [3]
The parallel Huffman coding problem can be solved in polylogarithmic time with quadratic work.

Hirschberg and Larmore [16] define the *Least Weight Subsequence* (LWS) problem as follows: Given an integer n, and a real-valued *weight* function $w(i, j)$ defined for integers $0 \leq i < j \leq n$, find a sequence of integers $\overline{\alpha} = (0 = \alpha_0 < \alpha_1 < \ldots < \alpha_{k-1} < \alpha_k = n)$ such that $w(\overline{\alpha}) = \sum_{i=0}^{k-1} w(\alpha_i, \alpha_{i+1})$ is minimized. Thus, the LWS problem reduces trivially to the minimum path problem on a weighted directed acyclic graph. The *Single Source* LWS problem is to find such a minimal sequence $0 = \alpha_0 < \alpha_1 < \ldots < \alpha_{k-1} < \alpha_k = m$ for all $m \leq n$. The weight function is said to be *concave* if for all $0 \leq i_0 \leq i_1 < j_0 \leq j_1 \leq n$,

$$w(i_0, j_0) + w(i_1, j_1) \leq w(i_0, j_1) + w(i_1, j_0). \tag{14.2}$$

The inequality (14.2) is also called the *quadrangle inequality* [52].

The LWS problem with the restriction that the weight function is concave is called the *Concave Least Weight Subsequence* (CLWS) problem. Hirschberg and Larmore [16] show that the LWS problem can be solved in $O(n^2)$ sequential time, while the CLWS problem can be solved in $O(n \log n)$ time. Wilber [51] gives an $O(n)$-time algorithm for the CLWS problem.

In the parallel setting, the CLWS problem seems to be more difficult. The best current polylogarithmic time algorithm for the CLWS problem uses concave matrix multiplication techniques and requires $O(\log^2 n)$ time with $n^2/\log^2 n$ processors.

Larmore and Przytycka [37] have shown how to compute efficiently CLWS in sublinear time with the total work smaller than quadratic. Using this approach they showed the following fact (which has been later slightly improved [28, 39].

THEOREM 14.13 *Optimal Huffman tree can be computed in $O(\sqrt{n} \log n)$ time with linear number of processors.*

Karpinski and Nekrich have shown an efficient parallel algorithm which *approximates* optimal Huffman code, see [5].

Similar, but much more complicated algorithm works for alphabetic trees. Again the CLWS algorithm is the main tool.

THEOREM 14.14 [23]
Optimal alphabetic tree can be constructed in polylogarithmic time with quadratic number of processors.

In case of general binary search trees the situation is more difficult. Polylogarithmic time algorithms need huge number of processors. However sublinear parallel time is easier.

THEOREM 14.15 [48] [31]
The OBST problem can be solved in **(a)** *polylogarithmic time with* $O(n^6)$ *processors,* **(b)** *in sublinear time and quadratic total work.*

References

[1] Alok Aggarwal, Baruch Schieber, Takeshi Tokuyama: Finding a Minimum-Weight k-Link Path Graphs with the Concave Monge Property and Applications. *Discrete & Computational Geometry* 12: 263-280 (1994).

[2] Doris Altenkamp and Kurt Mehlhorn, "Codes: Unequal Probabilities, Unequal Letter Costs," *J. Assoc. Comput. Mach.* **27** (3) (July 1980), 412–427.

[3] M. J. Atallah, S. R. Kosaraju, L. L. Larmore, G. L. Miller, and S-H. Teng. Constructing trees in parallel, *Proc. 1st ACM Symposium on Parallel Algorithms and Architectures* (1989), pp. 499–533.

[4] Julia Abrahams, "Code and Parse Trees for Lossless Source Encoding," *Sequences '97,* (1997).

[5] P. Berman, M. Karpinski, M. Nekrich, Approximating Huffman codes in parallel, *Proc. 29th ICALP, LNCS vol. 2380*, Springer, 2002, pp. 845-855.

[6] P. Bradford, M. Golin, L. Larmore, W. Rytter, Optimal Prefix-Free Codes for Unequal Letter Costs and Dynamic Programming with the Monge Property, *Journal of Algorithms*, Vol. 42, No. 2, February 2002, p. 277-303.

[7] A. Aggarwal, M. Klawe, S. Moran, P. Shor, and R. Wilber, Geometric applications of a matrix-searching algorithm, *Algorithmica* **2** (1987), pp. 195–208.

[8] Serap A. Savari, "Some Notes on Varn Coding," *IEEE Transactions on Information Theory*, **40** (1) (Jan. 1994), 181–186.

[9] Siu-Ngan Choi and M. Golin, "Lopsided trees: Algorithms, Analyses and Applications," *Automata, Languages and Programming*, Proceedings of the 23rd International Colloquium on Automata, Languages, and Programming (ICALP 96).

[10] N. Cot, "A linear-time ordering procedure with applications to variable length encoding," *Proc. 8th Annual Princeton Conference on Information Sciences and Systems*, (1974), pp. 460–463.

[11] A. M. Garsia and M. L. Wachs, A New algorithm for minimal binary search trees, *SIAM Journal of Computing* **6** (1977), pp. 622–642.

[12] T. C. Hu. A new proof of the T-C algorithm, *SIAM Journal of Applied Mathematics* **25** (1973), pp. 83–94.

[13] E. N. Gilbert, "Coding with Digits of Unequal Costs," *IEEE Trans. Inform. Theory,* **41** (1995).

[14] A. Gibbons, W.Rytter, Efficient parallel algorithms, Cambridge Univ. Press 1997.

[15] M. Golin and G. Rote, "A Dynamic Programming Algorithm for Constructing Opti-

mal Prefix-Free Codes for Unequal Letter Costs," *Proceedings of the 22nd International Colloquium on Automata Languages and Programming (ICALP '95),* (July 1995) 256-267.

[16] D. S. Hirschberg and L. L. Larmore, The Least weight subsequence problem, *Proc. 26th IEEE Symp. on Foundations of Computer Science* Portland Oregon (Oct. 1985), pp. 137–143. Reprinted in *SIAM Journal on Computing* **16** (1987), pp. 628–638.

[17] D. A. Huffman. A Method for the constructing of minimum redundancy codes, *Proc. IRE* **40** (1952), pp. 1098–1101.

[18] T. C. Hu and A. C. Tucker, Optimal computer search trees and variable length alphabetic codes, *SIAM Journal of Applied Mathematics* **21** (1971), pp. 514–532.

[19] J. H. Kingston, A new proof of the Garsia-Wachs algorithm, *Journal of Algorithms* **9** (1988) pp. 129–136.

[20] M. M. Klawe and B. Mumey, Upper and Lower Bounds on Constructing Alphabetic Binary Trees, *Proceedings of the 4th ACM-SIAM Symposium on Discrete Algorithms* (1993), pp. 185–193.

[21] L. L. Larmore and D. S. Hirschberg, A fast algorithm for optimal length–limited Huffman codes, *Journal of the ACM* **37** (1990), pp. 464–473.

[22] L. L. Larmore and T. M. Przytycka, The optimal alphabetic tree problem revisited, *Proceedings of the 21*st *International Colloquium*, ICALP'94, Jerusalem, LNCS 820, Springer-Verlag, (1994), pp. 251–262.

[23] L. L. Larmore, T. M. Przytycka, and W. Rytter, Parallel construction of optimal alphabetic trees, *Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures* (1993), pp. 214–223.

[24] M. Golin and G. Rote, "A Dynamic Programming Algorithm for Constructing Optimal Prefix-Free Codes for Uneq ual Letter Costs," *Proceedings of the 22nd International Colloquium on Automata Languages and Progr amming (ICALP '95),* (July 1995) 256-267. Expanded version to appear in *IEEE Trans. Inform. Theory.*

[25] R. Güttler, K. Mehlhorn and W. Schneider. Binary search trees: average and worst case behavior, *Electron. Informationsverarb Kybernet*, 16 (1980) pp. 41–61.

[26] Sanjiv Kapoor and Edward Reingold, "Optimum Lopsided Binary Trees," *Journal of the Association for Computing Machinery* **36** (3) (July 1989), 573–590.

[27] R. M. Karp, "Minimum-Redundancy Coding for the Discrete Noiseless Channel," *IRE Transactions on Information Theory,* **7** (1961) 27-39.

[28] M. Karpinski, L. Larmore, Yakov Nekrich, A work efficient algorithm for the construction of length-limited Huffman codes, to appear in Parallel Processing Letters.

[29] M. Karpinski, L. Larmore and W. Rytter, Sequential and parallel subquadratic work constructions of approximately optimal binary search trees, *the 7th ACM Symposium on Discrete Algorithms, SODA'96.*

[30] Marek Karpinski, Lawrence L. Larmore, and Wojciech Rytter. Correctness of constructing optimal alphabetic trees revisited. *Theoretical Computer Science*, 180(1-2):309-324, 10 June 1997.

[31] M. Karpinski, W. Rytter, On a Sublinear Time Parallel Construction of Optimal Binary Search Trees, Parallel Processing Letters, Volume 8 - Number 3, 1998.

[32] D. G. Kirkpatrick and T. M. Przytycka, Parallel construction of binary trees with almost optimal weighted path length, *Proc. 2*nd *Symp. on Parallel Algorithms and Architectures* (1990).

[33] D. G. Kirkpatrick and T. M. Przytycka, An optimal parallel minimax tree algorithm, *Proc. 2*nd *IEEE Symp. of Parallel and Distributed Processing* (1990), pp. 293–300.

[34] D. E. Knuth, Optimum binary search trees, *Acta Informatica* **1** (1971) pp. 14–25.
[35] D. E. Knuth. *The Art of computer programming*, Addison–Wesley (1973).
[36] L. L. Larmore, and T. M. Przytycka, Parallel construction of trees with optimal weighted path length, *Proc.* 3rd *ACM Symposium on Parallel Algorithms and Architectures* (1991), pp. 71–80.
[37] L. L. Larmore, and T. M. Przytycka, Constructing Huffman trees in parallel, SIAM J. Computing 24(6), (1995) pp. 1163-1169.
[38] L.Larmore, W. Rytter, Optimal parallel algorithms for some dynamic programming problems, IPL 52 (1994) 31-34.
[39] Ch. Levcopulos, T. Przytycka, A work-time trade-off in parallel computation of Huffman trees and concave least weight subsequence problem, Parallel Processing Letters 4(1-2) (1994) pp. 37-43.
[40] Ruy Luiz Milidiu, Eduardo Laber, The warm-up algorithm:a Lagrangian construction of length limited Huffman codes, SIAM J. Comput. 30(5): 1405-1426 (2000).
[41] Ruy Luiz Milidi, Eduardo Sany Laber: Linear Time Recognition of Optimal L-Restricted Prefix Codes (Extended Abstract). LATIN 2000: 227-236.
[42] Ruy Luiz Milidi, Eduardo Sany Laber: Bounding the Inefficiency of Length-Restricted Prefix Codes. Algorithmica 31(4): 513-529 (2001).
[43] W. Rytter, Efficient parallel computations for some dynamic programming problems, *Theo. Comp. Sci.* **59** (1988), pp. 297–307.
[44] K. Mehlhorn, *Data structures and algorithms*, vol. 1, Springer 1984.
[45] Y. Perl, M. R. Garey, and S. Even, "Efficient generation of optimal prefix code: Equiprobable words using unequal cost letters," *Journal of the Association for Computing Machinery* **22** (2) (April 1975), pp 202–214.
[46] P. Ramanan, Testing the optimality of alphabetic trees, *Theoretical Computer Science* **93** (1992), pp. 279–301.
[47] W. Rytter, The space complexity of the unique decipherability problem, IPL 16 (4) 1983.
[48] Fast parallel computations for some dynamic programming problems, *Theoretical Computer Science* (1988).
[49] Baruch Schieber, Computing a Minimum Weight k-Link Path in Graphs with the Concave Monge Property. 204-222.
[50] J. S. Vitter, "Dynamic Huffman Coding," *ACM Trans. Math. Software* 15 (June 1989), pp 158–167.
[51] R. Wilber, The Concave least weight subsequence problem revisited, *Journal of Algorithms* **9** (1988), pp. 418–425.
[52] F. F. Yao, Efficient dynamic programming using quadrangle inequalities, *Proceedings of the 12th ACM Symposium on Theory of Computing* (1980), pp. 429–435.

15

B Trees

Donghui Zhang
Northeastern University

15.1 Introduction

We have seen binary search trees in Chapters 3 and 10. When data volume is large and does not fit in memory, an extension of the binary search tree to disk-based environment is the B-tree, originally invented by Bayer and McCreight [1]. In fact, since the B-tree is always balanced (all leaf nodes appear at the same level), it is an extension of the *balanced* binary search tree. Since each disk access exchanges a whole block of information between memory and disk rather than a few bytes, a node of the B-tree is expanded to hold more than two child pointers, up to the block capacity. To guarantee worst-case performance, the B-tree requires that every node (except the root) has to be at least half full. An exact match query, insertion or deletion need to access $O(\log_B n)$ nodes, where B is the page capacity in number of child pointers, and n is the number of objects.

Nowadays, every database management system (see Chapter 60 for more on applications of data structures to database management systems) has implemented the B-tree or its variants. Since the invention of the B-tree, there have been many variations proposed. In particular, Knuth [4] defined the B*-tree as a B-tree in which every node has to be at least 2/3 full (instead of just 1/2 full). If a page overflows during insertion, the B*-tree applies a local redistribution scheme to delay splitting the node till two another sibling node is also full. At this time, the two nodes are split into three. Perhaps the best variation of the B-tree is the B+-tree, whose idea was originally suggested by Knuth [4], but whose name was given by Comer [2]. (Before Comer, Knuth used the name B*-tree to represent both B*-tree and B+-tree.) In a B+-tree, every object stays at the leaf level. Update and query algorithms need to be modified from those of the original B-tree accordingly.

The idea of the B-tree also motivates the design of many other disk-based index structures like the R-tree [3], the state-of-art spatial index structure (Chapter 21).

In this chapter, we describe the B-tree and B+-tree in more detail. In section 15.2, we briefly describe the disk-based environment and we introduce some notations. The B-tree is described in section 15.3, while the B+-tree is described in section 15.4. Finally, in section 15.5 we further discuss some related issues.

15.2 The Disk-Based Environment

Most application software deal with data. For instance, a registration application may keep the name, address and social security number of all students. The data has to be stored somewhere. There are three levels of storage. The computer CPU deals directly with the **primary storage**, which means the main memory (as well as cache). While data stored at this level can be access quickly, we cannot store everything in memory for two reasons. First, memory is expensive. Second, memory is volatile, i.e. if there is a power failure, information stored in memory gets lost.

The **secondary storage** stands for magnetic disks. Although it has slower access, it is less expensive and it is non-volatile. This satisfies most needs. For data which do not need to be accessed often, they can also be stored in the **tertiary storage**, e.g. tapes.

Since the CPU does not deal with disk directly, in order for any piece of data to be accessed, it has to be read from disk to memory first. Data is stored on disk in units called **blocks** or **pages**. Every disk access has to read/write one or multiple blocks. That is, even if we need to access a single integer stored in a disk block which contains thousands of integers, we need to read the whole block in. This tells us why internal memory data structures cannot be directly implemented as external-memory index structures.

Consider the binary search tree as an example. Suppose we implement every node as a disk block. The storage would be very inefficient. If a disk page is 8KB (=8192 bytes), while a node in the binary search tree is 16 bytes (four integers: a key, a value, and two child pointers), we know every page is only 0.2% full. To improve space efficiency, we should store multiple tree nodes in one disk page. However, the query and update will still be inefficient. The query and update need to access $O(\log_2 n)$ nodes, where n is the number of objects. Since it is possible that every node accessed is stored in a different disk page, we need to access $O(\log_2 n)$ disk pages. On the other hand, the B-tree query/update needs to access only $O(\log_B n)$ disk pages, which is a big improvement. A typical value of B is 100. Even if there are as many as billions of objects, the height of a B-tree, $\log_B n$, will be at most 4 or 5.

A fundamental question that the database research addresses is how to reduce the gap between memory and disk. That is, given a large amount of data, how to organize them on disk in such a way that they can efficiently be updated and retrieved. Here we measure efficiency by counting the total number of disk accesses we make. A disk access can be either a read or a write operation. Without going into details on how the data is organized on disk, let's make a few assumptions. First, assume each disk page is identified by a number called its *pageID*. Second, given a *pageID*, there is a function *DiskRead* which reads the page into memory. Correspondingly, there is a *DiskWrite* function which writes the in-memory page onto disk. Third, we are provided two functions which allocate a new disk page and deallocate an existing disk page.

The four functions are listed below.

- **DiskRead**: given a *pageID*, read the corresponding disk page into memory and return the corresponding memory location.
- **DiskWrite**: given the location of an in-memory page, write the page to disk.

- **AllocatePage**: find an unoccupied pageID, allocate space for the page in memory and return its memory location.
- **DeallocatePage**: given a $pageID$, mark the corresponding disk page as being unoccupied.

In the actual implementation, we should utilize a memory buffer pool. When we need to access a page, we should first check if it is already in the buffer pool, and we access the disk only when there is a buffer miss. Similarly, when we want to write a page out, we should write it to the buffer pool. An actual DiskWrite is performed under two circumstances: (a) The buffer pool is full and one page needs to be switched out of buffer. (b) The application program terminates. However, for our purposes we do not differentiate disk access and buffer pool access.

15.3 The B-tree

The problem which the B-tree aims to solve is: given a large collection of objects, each having a *key* and an *value*, design a disk-based index structure which efficiently supports query and update.

Here the query that is of interest is the *exact-match query*: given a key k, locate the value of the object with key=k. The update can be either an insertion or a deletion. That is, insert a new object into the index, or delete from the index an object with a given key.

15.3.1 B-tree Definition

A B-tree is a tree structure where every node corresponds to a disk page and which satisfies the following properties:

- A node (leaf or index) x has a value $x.num$ as the number of objects stored in x. It also stores the list of $x.num$ objects in increasing key order. The key and value of the i^{th} object ($1 \leq i \leq x.num$) are represented as $x.key[i]$ and $x.value[i]$, respectively.
- Every leaf node has the same depth.
- An index node x stores, besides $x.num$ objects, $x.num$+1 child pointers. Here each child pointer is a pageID of the corresponding child node. The i^{th} child pointer is denoted as $x.child[i]$. It corresponds to a key range ($x.key[i-1]$, $x.key[i]$). This means that in the i^{th} sub-tree, any object key must be larger than $x.key[i-1]$ and smaller than $x.key[i]$. For instance, in the sub-tree referenced by $x.child[1]$, the object keys are smaller than $x.key[1]$. In the sub-tree referenced by $x.child[2]$, the objects keys are between $x.key[1]$ and $x.key[2]$, and so on.
- Every node except the root node has to be at least half full. That is, suppose an index node can hold up to $2B$ child pointers (besides, of course, $2B$-1 objects), then any index node except the root must have at least B child pointers. A leaf node can hold more objects, since no child pointer needs to be stored. However, for simplicity we assume a leaf node holds between B and $2B$ objects.
- If the root node is an index node, it must have at least two children.

A special case of the B-tree is when $B = 2$. Here every index node must have 2 or 3 or 4 child pointers. This special case is called the **2-3-4 tree**.

Figure 15.1 shows an example of a B-tree. In particular, it's a 2-3-4 tree.

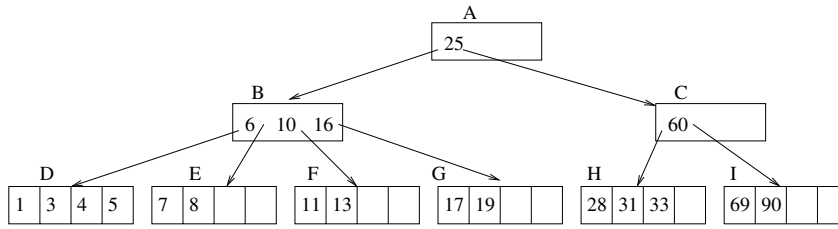

FIGURE 15.1: An example of a B-tree.

In the figure, every index node contains between 2 and 4 child pointers, and every leaf node contains between 2 and 4 objects. The root node A is an index node. Currently it has one object with key=25 and two child pointers. In the left sub-tree, every object has key<25. In the right sub-tree, every object has key>25. Every leaf node (D through I) are located at the same depth: their distance to A is 2. Currently, there are two pages which are full: an index node B and a leaf node D.

15.3.2 B-tree Query

To find the value of an object with key=k, we call the *Query* algorithm given below. The parameters are the tree root pageID and the search key k. The algorithm works as follows. It follows (at most) a single path from root to leaf. At each index node along the path, there can be at most one sub-tree whose key range contains k. A recursive call on that sub-tree is performed (step 2c). Eventually, we reach a leaf node (step 3a). If there exists an object in the node with key=k, the algorithm returns the value of the object. Otherwise, the object does not exist in the tree and $NULL$ is returned. Since the index nodes of the B-tree also stores objects, it is possible that the object with key=k is found in an index node. In this case, the algorithm returns the object value without going down to the next level (step 2a).

Algorithm *Query*($pageID$, k)
Input: $pageID$ of a B-tree node, a key k to be searched.
Output: $value$ of the object with key= k; $NULL$ if non-exist.

1. $x = DiskRead(pageID)$.
2. **if** x is an index node
   (a) If there is an object o in x s.t. $o.key = k$, return $o.value$.
   (b) Find the child pointer $x.child[i]$ whose key range contains k.
   (c) **return** Query($x.child[i]$, k).
3. **else**
   (a) If there is an object o in x s.t. $o.key = k$, return $o.value$. Otherwise, return $NULL$.
4. **end if**

As an example, Figure 15.2 shows how to perform a search query for $k = 13$. At node A, we should follow the left sub-tree since $k < 25$. At node B, we should follow the third sub-tree since $10 < k < 16$. Now we reach a leaf node F. An object with key=13 is found in the node.

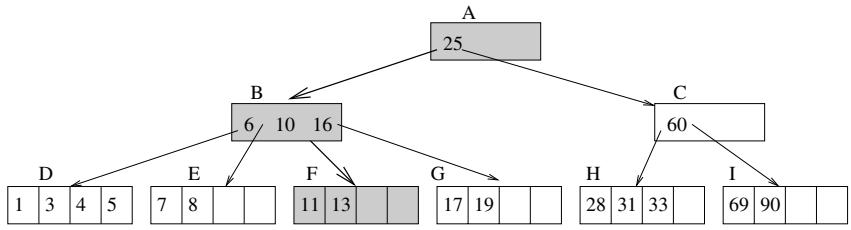

FIGURE 15.2: Query processing in a B-tree.

If the query wants to search for $k = 12$, we still examine the three nodes A, B, F. This time, no object with key=12 is found in F, and thus the algorithm returns $NULL$. If the search key is 10 instead, the algorithm only examines node A and B. Since in node B such an object is found, the algorithm stops there.

Notice that in the Query algorithm, only *DiskRead* function is called. The other three functions, e.g. *DiskWrite* are not needed as the algorithm does not modify the B-tree. Since the query algorithm examines a single path from root to leaf, the complexity of the algorithm in number of I/Os is $O(\log_B n)$, where n is the number of objects.

15.3.3 B-tree Insertion

To insert a new object with key k and value v into the index, we call the *Insert* algorithm given below.

Algorithm *Insert*($root$, k, v)
Input: $root$ pageID of a B-tree, the key k and the value v of a new object.
Prerequisite: The object does not exist in the tree.
Action: Insert the new object into the B-tree.

1. $x = DiskRead(root)$.
2. **if** x is full
   (a) $y = AllocatePage()$, $z = AllocatePage()$.
   (b) Locate the middle object o_i stored in x. Move the objects to the left of o_i into y. Move the objects to the right of o_i into z. If x is an index page, also move the child pointers accordingly.
   (c) $x.child[1] = y.pageID$, $x.child[2] = z.pageID$.
   (d) $DiskWrite(x)$, $DiskWrite(y)$, $DiskWrite(z)$.
3. **end if**
4. $InsertNotFull(x,\ k,\ v)$.

Basically, the algorithm makes sure that root page is not currently full, and then it calls the *InsertNotFull* function to insert the object into the tree. If the root page x is full, the algorithm will split it into two nodes y and z, and node x will be promoted to a higher level, thus increasing the height of the tree.

This scenario is illustrated in Figure 15.3. Node x is a full root page. It contains three objects and four child pointers. If we try to insert some record into the tree, the root node is split into two nodes y and z. Originally, x contains $x.num = 3$ objects. The left object (key=6) is moved to a new node y. The right object (key=16) is moved to a new node z.

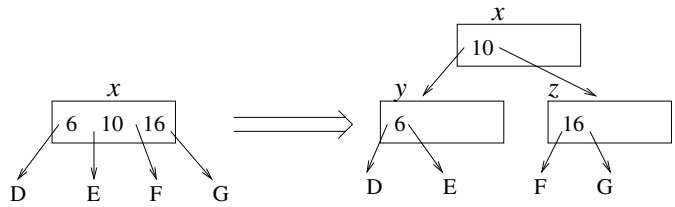

FIGURE 15.3: Splitting the root node increases the height of the tree.

The middle object (key=10) remains in x. Correspondingly, the child pointers D, E, F, G are also moved. Now, x contains only one object (key=10). We make it as the new root, and make y and z be the two children of it.

To insert an object into a sub-tree rooted by a non-full node x, the following algorithm *InsertNotFull* is used.

Algorithm *InsertNotFull*(x, k, v)
Input: an in-memory page x of a B-tree, the key k and the value v of a new object.
Prerequisite: page x is not full.
Action: Insert the new object into the sub-tree rooted by x.

1. **if** x is a leaf page
   (a) Insert the new object into x, keeping objects in sorted order.
   (b) $DiskWrite(x)$.
2. **else**
   (a) Find the child pointer $x.child[i]$ whose key range contains k.
   (b) $w = DiskRead(x.child[i])$.
   (c) **if** w is full
      i. $y = AllocatePage()$.
      ii. Locate the middle object o_j stored in w. Move the objects to the right of o_j into y. If w is an index page, also move the child pointers accordingly.
      iii. Move o_j into x. Accordingly, add a child pointer in x (to the right of o_j) pointing to y.
      iv. $DiskWrite(x)$, $DiskWrite(y)$, $DiskWrite(w)$.
      v. If $k < o_j.key$, call $InsertNotFull(w,\ k,\ v)$; otherwise, call $InsertNotFull(y,\ k,\ v)$.
   (d) **else**
      $InsertNotFull(w,\ k,\ v)$.
   (e) **end if**
3. **end if**

Algorithm *InsertNotFull* examines a single path from root to leaf, and eventually insert the object into some leaf page. At each level, the algorithm follows the child pointer whose key range contains the key of the new object (step 2a). If no node along the path is full, the algorithm recursively calls itself on each of these nodes (step 2d) till the leaf level, where the object is inserted into the leaf node (step 1).

Consider the other case when some node w along the path is full (step 2c). The node is first split into two (w and y). The right half of the objects from w are moved to y, while the middle object is pushed into the parent node. After the split, the key range of either w or y, but not both, contains the key of the new object. A recursive call is performed on the correct node.

As an example, consider inserting an object with key=14 into the B-tree of Figure 15.2. The result is shown in Figure 15.4. The child pointers that are followed are thick. When we examine the root node A, we follow the child pointer to B. Since B is full, we first split it into two, by moving the right half of the objects (only one object in our case, with key=16) into a new node B''. The child pointers to F and G are moved as well. Further, the previous middle object in B (key=10) is moved to the parent node A. A new child pointer to B'' is also generated in A. Now, since the key of the new object is 14, which is bigger than 10, we recursively call the algorithm on B''. At this node, since $14 < 16$, we recursively call the algorithm on node F. Since F is a leaf node, the algorithm finishes by inserting the new object into F. The accessed disk pages are shown as shadowed.

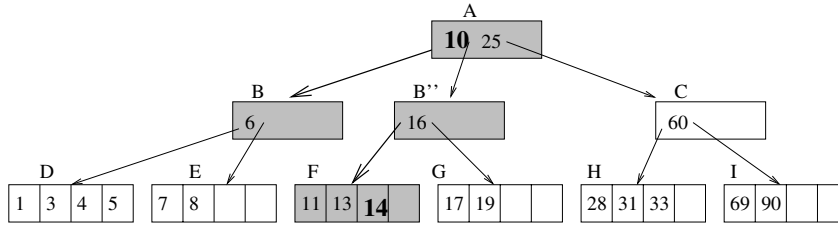

FIGURE 15.4: Inserting an object with key=14 into the B-tree of Figure 15.2 Since node B is full, it is split into two (B and B''). The object is recursively inserted into the sub-tree rooted by B''. At the lowest level, it is stored in node F.

15.3.4 B-tree Deletion

This section describes the *Delete* algorithm which is used to delete an object with key=k from the B-tree. It is a recursive algorithm. It takes (besides k) as parameter a tree node, and it will perform deletion in the sub-tree rooted by that node.

We know that there is a single path from the root node to the node x that contains k. The *Delete* algorithm examines this path. Along the path, at each level when we examine node x, we first make sure that x has at least one more element than half full (except the case when x is the root). The reasoning behind this is that in order to delete an element from the sub-tree rooted by x, the number of element stored in x can be reduced at most by one. If x has one more element than half full (minimum occupancy), it can be guaranteed that x will not underflow. We distinguish three cases:

1. x is a leaf node;
2. x is an index node which contains an object with key=k;
3. x is an index node which does not contain an object with key=k.

We first describe the *Delete* algorithm and then discuss the three cases in more detail.

Algorithm *Delete* (x, k)
Input: an in-memory node x of a B-tree, the key k to be deleted.
Prerequisite: an object with key=k exists in the sub-tree rooted by x.
Action: Delete the object from the sub-tree rooted by x.

1. **if** x is a leaf page
   (a) Delete the object with key=k from x.
   (b) $DiskWrite(x)$.
2. **else if** x does not contain the object with key=k
   (a) Locate the child $x.child[i]$ whose key range contains k.
   (b) $y = DiskRead(x.child[i])$.
   (c) **if** y is exactly half full
      i. If the sibling node z immediate to the left (right) of y has at least one more object than minimally required, add one more object to y by moving $x.key[i]$ from x to y and move that last (first) object from z to x. If y is an index node, the last (first) child pointer in z is also moved to y.
      ii. Otherwise, any immediate sibling of y is exactly half full. Merge y with an immediate sibling.

      end if
   (d) $Delete(y, k)$.
3. **else**
   (a) If the child y that precedes k in x has at least one more object than minimally required, find the predecessor k' of k in the sub-tree rooted by y, recursively delete k' from the sub-tree and replace k with k' in x.
   (b) Otherwise, y is exactly half full. We check the child z that immediately follows k in x. If z has at least one more object than minimally required, find the successor k' of k in the sub-tree rooted by z, recursively delete k' from the sub-tree and replace k with k' in x.
   (c) Otherwise, both y and z are half full. Merge them into one node and push k down to the new node as well. Recursively delete k from this new node.
4. **end if**

Along the search path from the root to the node containing the object to be deleted, for each node x we encounter, there are three cases. The simplest scenario is when x is a leaf node (step 1 of the algorithm). In this case, the object is deleted from the node and the algorithm returns. Note that there is no need to handle underflow. The reason is: if the leaf node is root, there is only one node in the tree and it is fine if it has only a few objects; otherwise, the previous recursive step has already guaranteed that x has at least one more object than minimally required.

Steps 2 and 3 of the algorithm correspond to two different cases of dealing with an index node.

For step 2, the index node x does not contain the object with key=k. Thus there exists a child node y whose key range contains k. After we read the child node into memory (step 2b), we will recursively call the *Delete* algorithm on the sub-tree rooted by y (step 2d).

However, before we do that, step 2(c) of the algorithm makes sure that y contains at least one more object than half full.

Suppose we want to delete 5 from the B-tree shown in Figure 15.2. When we are examining the root node A, we see that child node B should be followed next. Since B has two more objects than half full, the recursion goes to node B. In turn, since D has two more objects than minimum occupancy, the recursion goes to node D, where the object can be removed.

Let's examine another example. Still from the B+-tree shown in Figure 15.2, suppose we want to delete 33. The algorithm finds that the child node $y = C$ is half full. One more object needs to be incorporated into node C before a recursive call on C is performed. There are two sub-cases. The first sub-case is when one immediate sibling z of node y has at least one more object than minimally required. This case corresponds to step 2(c)i of the algorithm. To handle this case, we drag one object down from x to y, and we push one object from the sibling node up to x. As an example, the deletion of object 33 is shown in Figure 15.5.

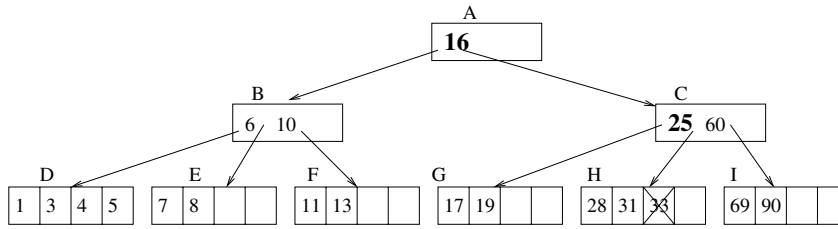

FIGURE 15.5: Illustration of step 2(c)i of the *Delete* algorithm. Deleting an object with key=33 from the B-tree of Figure 15.2. At node A, we examine the right child. Since node C only had one object before, a new object was added to it in the following way: the object with key=25 is moved from A to C, and the object with key=16 is moved from B to A. Also, the child pointer pointing to G is moved from B to C.

Another sub-case is when all immediate siblings of y are exactly half full. In this case, we merge y with one sibling. In our 2-3-4-tree example, an index node which is half full contains one object. If we merge two such nodes together, we also drag an object from the parent node of them down to the merged node. The node will then contain three objects, which is full but does not overflow.

For instance, suppose we want to delete object 31 from Figure 15.5. When we are examining node $x = C$, we see that we need to recursively delete in the child node $y = H$. Now, both immediate siblings of H are exactly half full. So we need to merge H with a sibling, say G. Besides moving the remaining object 28 from H to G, we also should drag object 25 from the parent node C to G. The figure is omitted for this case.

The third case is that node x is an index node which contains the object to be deleted. Step 3 of algorithm *Delete* corresponds to this scenario. We cannot simply delete the object from x, because we also need to decrement the number of child pointers by one. In Figure 15.5, suppose we want to delete object with key=25, which is stored in index node C. We cannot simply remove the object, since C would have one object but three child pointers left. Now, if child node G immediately to the left of key 25 had three or more objects, the algorithm would execute step 3(a) and move the last object from G into C to fill in the space of the deleted object. Step 3(b) is a symmetric step which shows that we can move an object from the right sub-tree.

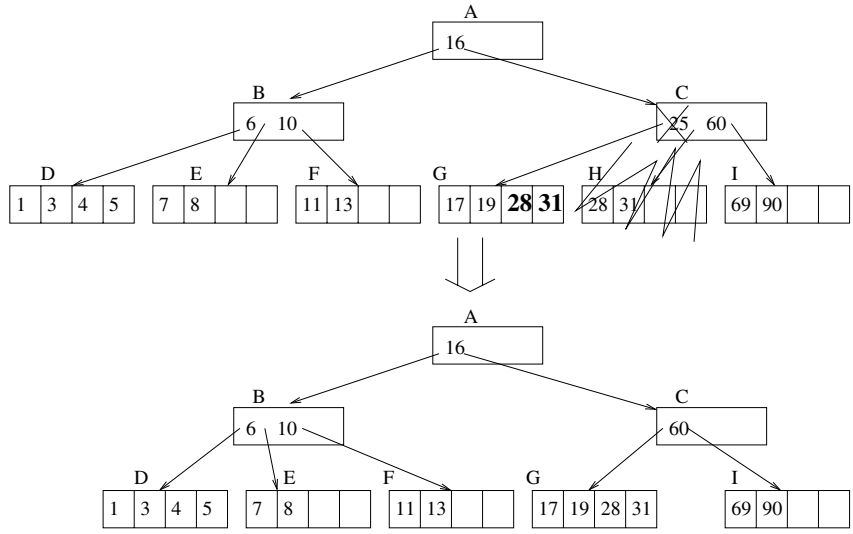

FIGURE 15.6: Illustration of step 3(c) of the *Delete* algorithm. Deleting an object with key=25 from the B-tree of Figure 15.5. At node A, we examine the right child. We see that node C contains the object with key=25. We cannot move an object up from a child node of C, since both children G and H (around key 25) are exactly half full. The algorithm merges these two nodes into one, by moving objects 28 and 31 from H to G and then deleting H. Node C loses an object (key=25) and a child pointer (to H).

However, in our case, both child nodes G and H are half full and thus cannot contribute an object. Step 3(c) of the algorithm corresponds to this case. As shown in Figure 15.6, the two nodes are merged into one.

15.4 The B+-tree

The most well-know variation of the B-tree is the B+-tree. There are two major differences from the B-tree. First, all objects in the B+-tree are kept in leaf nodes. Second, all leaf nodes are linked together as a double-linked list.

The structure of the B+-tree looks quite similar to the B-tree. Thus we omit the details. We do point out that in an index node of a B+-tree, different from the B-tree, we do not store object values. We still store object keys, though. However, since all objects are stored in the leaf level, the keys stored in index nodes act as *routers*, as they direct the search algorithm to go to the correct child node at each level.

15.4.1 Copy-up and Push-up

One may wonder where the routers in the index nodes come from. To understand this, let's look at an example. Initially, the B+-tree has a single node which is a leaf node. After $2B$ insertions, the root node becomes full. In Figure 15.7(a), if we try to insert an object to the node A when it is already full, it temporarily overflows. To handle the overflow, the B+-tree will split the node into two nodes A and B. Furthermore, a new node C is generated, which is the new root of the tree. The first key in leaf node B is **copied** up to C. The result B+-tree is shown in Figure 15.7(b).

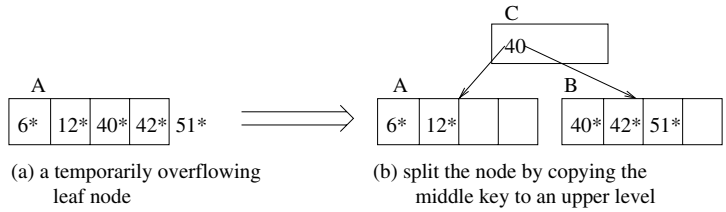

FIGURE 15.7: Illustration of a leaf-node split in the B+-tree. The middle key 40 (same as the first key in the right node) is copied up to the parent node.

We point out that a key in an index node may be validly replaced by some other keys, unlike in a leaf node. For instance, in node C of Figure 15.7(b), we can replace the key 40 to 35. As long as it is smaller than all keys in the left sub-tree and bigger than or equal to all keys in the right sub-tree, it is fine.

To emphasize the fact that the keys in a index node are different from the keys in a leaf node (a key in an index node is not a real object), in the B+-tree figures we will attach a (*) to each key in a leaf node.

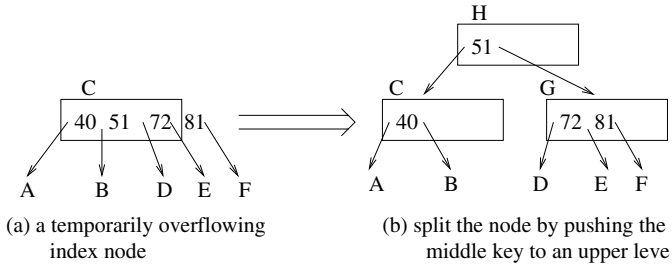

FIGURE 15.8: Illustration of an index-node split in the B+-tree. The middle key 51 is pushed up to the parent node.

As a comparison, consider the split of an index node. In Figure 15.8(a), the index node C temporarily overflows. It is split into two, C and G. Since before the split, C was the tree root, a new root node H is generated. See Figure 15.8(b). Here the middle key 51 in the original node C is **pushed** up to the parent node.

15.4.2 B+-tree Query

As in the B-tree, the B+-tree supports the *exact-match query* which finds the object with a given key. Furthermore, the B+-tree can efficiently support the *range query*, which finds the objects whose keys are in a given range.

To perform the exact-match query, the B+-tree follows a single path from root to leaf. In the root node, there is a single child pointer whose key range contains the key to be searched for. If we follow the child pointer to the corresponding child node, inside the child node there is also a single child pointer whose key range contains the object to be searched for. Eventually, we reach a leaf node. The object to be searched, if it exists, must be located in this node. As an example, Figure 15.9 shows the search path if we search key=42.

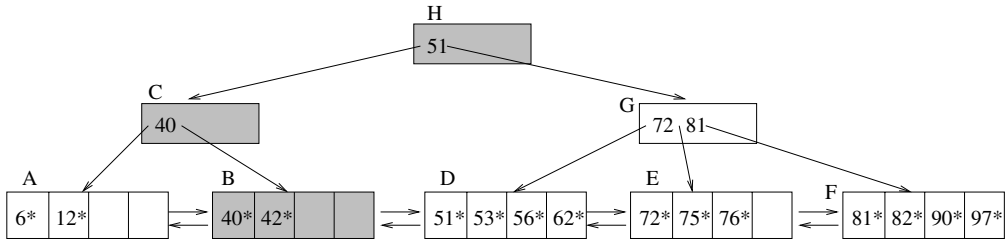

FIGURE 15.9: Illustration of the exact-match query algorithm in the B+-tree. To search for an object with key=42, nodes H, C and B are examined.

Beside the exact-match query, the B+-tree also supports the ***range query***. That is, find all objects whose keys belong to a range R. In order to do so, all the leaf nodes of a B+-tree are linked together. If we want to search for all objects whose keys are in the range R =[low, $high$], we perform an exact match query for key=low. This leads us to a leaf node l. We examine all objects in l, and then we follow the sibling link to the next leaf node, and so on. The algorithm stops when an object with key> $high$ is met. An example is shown in Figure 15.10.

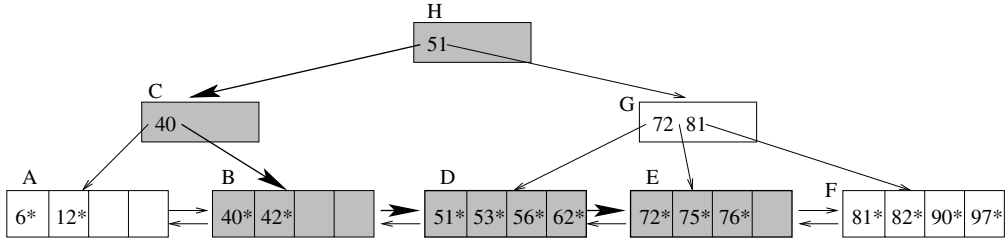

FIGURE 15.10: Illustration of the range query algorithm in the B+-tree. To search for all objects with keys in the range [42,75], the first step is to follow a path from root to leaf to find key 42 (H, C and B are examined). The second step is to follow the right-sibling pointers between leaf nodes and examine D, E. The algorithm stops at E as an object with key=76 is found.

15.4.3 B+-tree Insertion

Since all objects in the B+-tree are located at the leaf level, the insertion algorithm of the B+-tree is actually easier than that in the B-tree. We basically follow the exact-match query to find the leaf node which should contain the object if it were in the tree. Then we insert the object into the leaf node.

What needs to be taken care of is when the leaf node overflows and is split into two. In this case, a key and a child pointer are inserted into the parent node. This may in turn cause the parent node to overflow, and so on. In the worst case, all nodes along the insertion path are split. If the root node splits into two, the height of the tree increases by one. The insertion algorithm is given below.

Algorithm *Insert*(*root*, k, v)
Input: the *root* pageID of a B+-tree, the key k and the value v of a new object.
Prerequisite: the object with key=k does not exist in the tree.
Action: Insert the new object into the B+-tree.

1. Starting with the root node, perform an exact-match for key=k till a leaf node. Let the search path be $x_1, x_2, \ldots, x_h$, where x_1 is the root node, x_h is the leaf node where the new object should be inserted into, and x_i is the parent node of x_{i+1} where $1 \leq i \leq h$-1.
2. Insert the new object with key=k and value=v into x_h.
3. Let $i = h$.
   while x_i overflows
   (a) Split x_i into two nodes, by moving the larger half of the keys into a new node x_i'. If x_i is a leaf node, link x_i' into the double-linked list among leaf nodes.
   (b) Identify a key kk to be inserted into the parent level along with the child pointer pointing to x_i'. The choice of kk depends on the type of node x_i. If x_i is a leaf node, we need to perform *Copy-up*. That is, the smallest key in x_i' is copied as kk to the parent level. On the other hand, if x_i is an index node, we need to perform *Push-up*. This means the smallest key in x_i' is removed from x_i' and then stored as kk in the parent node.
   (c) **if** $i == 1$ /* the root node overflows */
      i. Create a new index node as the new root. In the new root, store one key=kk and two child pointers to x_i and x_i'.
      ii. **return**
   (d) **else**
      i. Insert a key kk and a child pointer pointing to x_i' into node x_{i-1}.
      ii. $i = i - 1$.
   (e) **end if**
   end while

As an example, Figure 15.11 shows how to insert an object with key=60 into the B+-tree shown in Figure 15.9.

15.4.4 B+-tree Deletion

To delete an object from the B+-tree, we first examine a single path from root to the leaf node containing the object. Then we remove the object from the node. At this point, if the node is at least half full, the algorithm returns. Otherwise, the algorithm tries to re-distribute objects between a sibling node and the underflowing node. If redistribution is not possible, the underflowing node is merged with a sibling.

Algorithm *Delete*(*root*, k)
Input: the *root* pageID of a B+-tree, the key k of the object to be deleted.
Prerequisite: the object with key=k exists in the tree.
Action: Delete the object with key=k from the B+-tree.

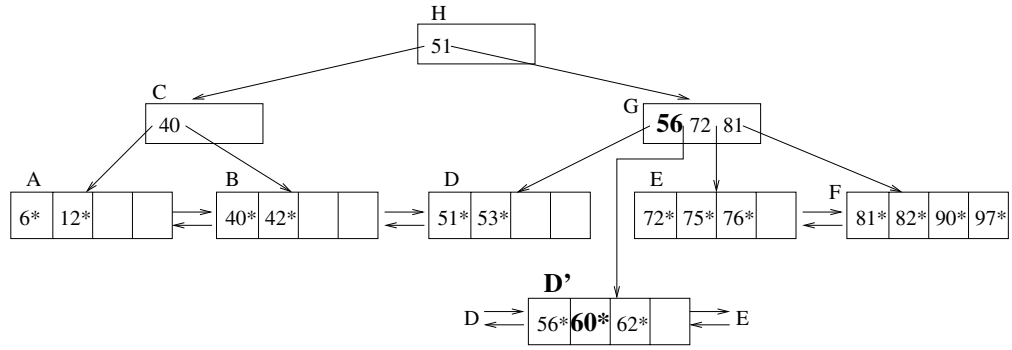

FIGURE 15.11: After inserting an object with key=60 into the B+-tree shown in Figure 15.9. Leaf node D splits into two. The middle key 56 is copied up to the parent node G.

1. Starting with the root node, perform an exact-match for key=k. Let the search path be $x_1, x_2, \ldots, x_h$, where x_1 is the root node, x_h is the leaf node that contains the object with key=k, and x_i is the parent node of x_{i+1}. ($1 \leq i \leq h$-1)
2. Delete the object with key=k from x_h.
3. If $h == 1$, **return**. This is because the tree has only one node which is the root node, and we do not care whether a root node underflows or not.
4. Let $i = h$.
   while x_i underflows
   (a) **if** an immediate sibling node s of x_i has at least one more entry than minimum occupancy
      i. Re-distribute entries evenly between s and x_i.
      ii. Corresponding to the re-distribution, a key kk in the parent node x_{i-1} needs to be modified. If x_i is an index node, kk is dragged down to x_i and a key from s is pushed up to fill in the place of kk. Otherwise, kk is simply replaced by a key in s.
      iii. **return**
   (b) **else**
      i. Merge x_i with a sibling node s. Delete the corresponding child pointer in x_{i-1}.
      ii. If x_i is an index node, drag the key in x_{i-1}, which previously divides x_i and s, into the new node x_i. Otherwise, delete that key in x_{i-1}.
      iii. $i = i - 1$.
   (c) **end if**
   end while

Step 1 of the algorithm follows a single path from root to leaf to find the object to be deleted. Step 2 deletes the object. The algorithm will finish at this point if any of the following two conditions hold. One, if the tree has a single node (step 3). Two, the leaf node is at least half full after the deletion (the while loop of step 4 is skipped).

As an example, suppose we delete object 56 and then 62 from the B+-tree shown in Figure 15.9. The deletions go to the same leaf node D, where no underflow occurs. The result is shown in Figure 15.12.

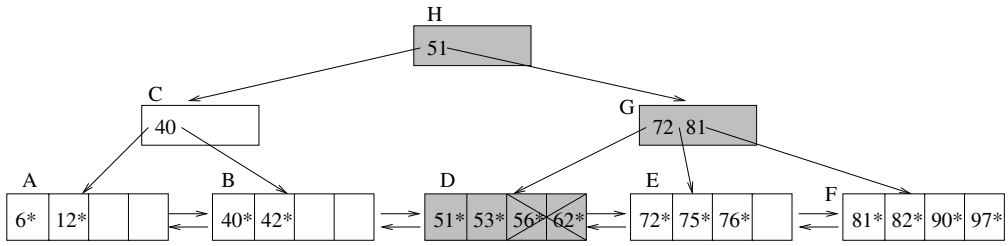

FIGURE 15.12: After deleting keys 56 and 62 from the B+-tree of Figure 15.9. Both keys are deleted from leaf node D, which still satisfies the minimum occupancy.

Now, let's try to delete key 53 from Figure 15.12. This time D underflows. Step 4 of the *Delete* algorithm handles underflows. In general, when a node x_i underflows, the algorithm tries to borrow some entries from a sibling node s, as described in step 4(a). Note that we could borrow just one entry to avoid underflow in x_i. However, this is not good because next time we delete something from x_i, it will underflow again. Instead, the algorithm **redistribute** entries evenly between x_i and s. Assume x_i has $B-1$ objects and s has $B+k$ objects, where $k \in [1..B]$. After redistribution, both x_i and s will have $B+(k-1)/2$ objects. Thus x_i can take another $(k-1)/2$ deletions before another underflow occurs.

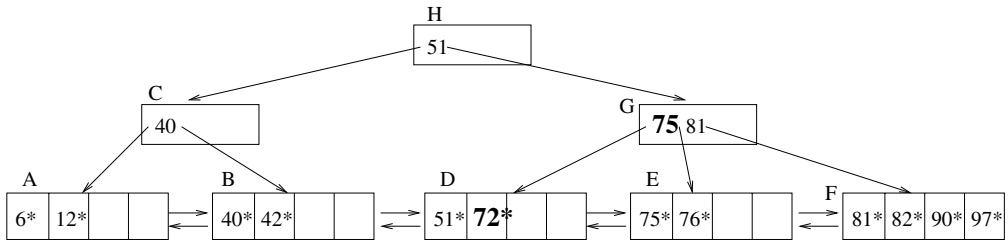

FIGURE 15.13: After deleting keys 53 from Figure 15.12. Objects in D and E are redistributed. A key in G is modified.

In our example, to delete key 53 from node D, we re-distribute objects in D and E, by moving 72* into D. As discussed in step 4(a)ii of the algorithm, we also needs to modify a key in the parent node G. In our case, since D is a leaf node, we simply replace the key 72 by 75 in node G. Here 75 is the smallest key in E. The result after the redistribution is shown in Figure 15.13. As a comparison, consider the hypothetical case when D were an index node. In this case, we would drag down the key 72 from G to D and push up a key from E to G.

Let's proceed the example further by deleting object 72 from the tree in Figure 15.13. Now, the node D underflows, and redistribution is not possible (since E, the only immediate sibling of D, is exactly half full). Step 4(b) of the *Delete* algorithm tells us to merge D and

E together. Correspondingly, a key and a child pointer need to be deleted from the parent node G. Since D is a leaf node, we simply delete the key 75 and the child pointer from G. The result is shown in Figure 15.14. As a comparison, imagine D were an index node. We would still remove key 75 and the child pointer from G, but we would keep the key 75 in node D.

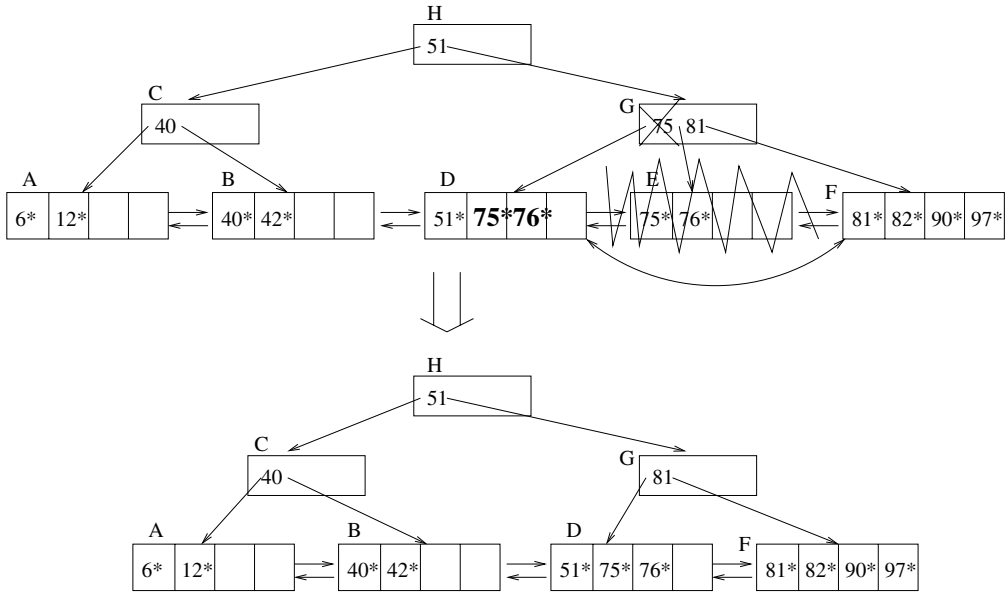

FIGURE 15.14: After deleting keys 72 from Figure 15.13. This figure corresponds to the scenario described in step 4(b) of the *Delete* algorithm. In particular, the example illustrates the merge of two leaf nodes (D and E). Node D underflows, but redistribution is not possible. From the parent node G, key 75 and child pointer to E are removed.

One may wonder why in the redistribution and the merge algorithms, the leaf node and the index node are treated differently. The reason is because when we generated an index entry, we had treated two cases differently: the case when the entry points to a leaf node and the case when the entry points to a index node. This is discussed at the beginning of Section 15.4. To generate a new entry pointing to a leaf node, we copied the smallest key from the leaf node. But to generate a new entry pointing to an index node, we pushed a key from the child node up. A key which was copied up can be safely deleted later (when merge occurs). But a key which was pushed up must be kept somewhere. If we delete it from a parent node, we should drag it down to a child node.

As a running example of merging index nodes, consider deleting object 42 from the B+-tree of Figure 15.14. Node B underflows, and it is merged with A. Correspondingly, in the parent node C, the key 40 and the child pointer to B are deleted. The temporary result is shown in Figure 15.15. It's temporary since node C underflows.

To handle the underflow of node C, it is merged with G, its sole sibling node. As a consequence, the root node H now has only one child. Thus, H is removed and C becomes the new root. We point out that to merge two index nodes C and G, a key is dragged down from the parent node (versus being deleted in the case of merging leaf nodes). The final

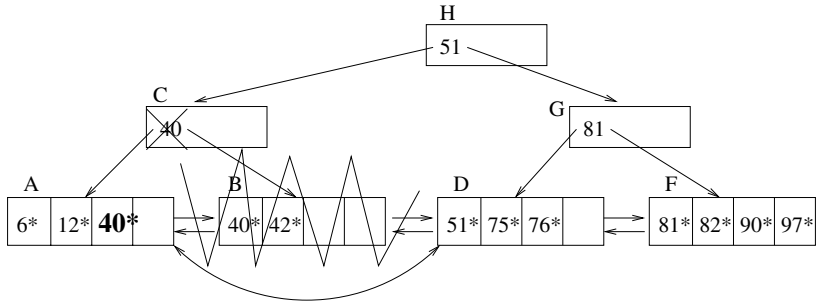

FIGURE 15.15: Temporary tree in the middle of deleting object 42 from Figure 15.14. Nodes A and B are merged. Key 40 and child pointer to B are removed from C.

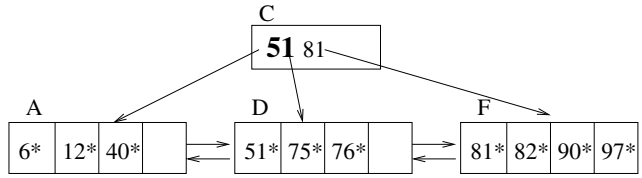

FIGURE 15.16: After the deletion of object 42 is finished. This figure illustrates an example of merging two index nodes. In particular, index nodes C and G are merged. The key 51 is dragged down from the parent node H. Since that is the only key in root H, node C becomes the new root and the height of the tree is decreased by one.

result after completing the deletion of 42* is shown in Figure 15.16.

15.5 Further Discussions

In this section we discuss various issues of the B-tree and the B+-tree.

15.5.1 Efficiency Analysis

Theorem: *In the B-tree or the B+-tree, the I/O cost of insertion, deletion and exact-match query is* $O(\log_B n)$. *In the B+-tree, the I/O cost of a range search is* $O(\log_B n + t/B)$. *Here B is the minimum page capacity in number of records, n is the total number of objects in the tree, and t is the number of objects in the range query result.*

The correctness of the theorem can be seen from the discussion of the algorithms. Basically, for both the B-tree and the B+-tree, all the insertion, deletion and exact-match query algorithms examine a single path from root to leaf. At each node, the algorithm might examine up to two other nodes. However, asymptotically the complexity of these algorithms are equal to the height of the tree. Since there are n objects, and the minimum fan-out of the tree is B, the height of the tree is $O(log_B n)$. So the complexity of the algorithms is $O(log_B n)$ as well.

For the range query in the B+-tree, $log_B n$ nodes are examined to find the leaf node that contains the low value of the query range. By following the sibling pointers in the leaf nodes, the other leaf nodes that contain objects in the query range are also found. Among

all the leaf nodes examined, except for the first and the last, every node contains at least B objects in the query result. Thus if there are t objects in the query result, the range query complexity is $O(log_B n + t/B)$.

15.5.2 Why is the B+-tree Widely Accepted?

One can safely claim that the B+-tree has been included in at least 99%, if not all, of the database management systems (DBMS). No other index structure has received so much attention. Why is that?

Let's do some calculation. First, we point out that a practical number of minimum occupancy of a B+-tree is $B = 100$. Thus the fan-out of the tree is between 100 and 200. Analysis has shown that in a real-world B+-tree, the average page capacity is about 66.7% full. Or, a page typically contains 200*66.7%=133 entries. Here is the relationship between the height of the tree and the number of objects that can hold in a typical B+-tree:

- *height=0*: B+-tree holds 133 objects on average. There is a single node, which is 66.7% full.
- *height=1*: B+-tree holds $133^2 = 17,689$ objects. There are 133 leaf nodes, each holds 133 objects.
- *height=2*: B+-tree holds $133^3 = 2,352,637$ objects.
- *height=3*: B+-tree holds $133^4 = 312,900,721$ (over 0.3 billion) objects.

The first two levels of the B+-tree contains 1+133=134 disk pages. This is very small. If a disk page is 4KB large, 134 disk pages occupy 134*4KB=536KB disk space. It's quite reasonable to assume that the first two levels of the B+-tree always stays in memory.

The calculations lead to this discovery: in a large database with 0.3 billion objects, to find one object we only need to access two disk pages! This is unbelievably good.

15.5.3 Bulk-Loading a B+-tree

In some cases, we are given a large set of records and we are asked to build a B+-tree index. Of course, we can start with an empty B+-tree and insert one record at a time using the *Insert* algorithm. However, this approach is not efficient, as the I/O cost is $O(n \cdot \log_B n)$.

Many systems have implemented the *bulk-loading* utility. The idea is as follows. First, sort the objects. Use the objects to fill in leaf nodes in sequential order. For instance, if a leaf node holds up to $2B$ objects, the $2B$ smallest objects are stored in page 1, the next $2B$ objects are stored in page 2, etc. Next, build the index nodes at one level up. Assume an index node holds up to $2B$ child pointers. Create the first index node as the parent of the first $2B$ leaf nodes. Create the second index node as the parent of the next $2B$ leaf nodes, etc. Then, build the index nodes at two levels above the leaf level, and so on. The process stops when there is only one node at a level. This node is the tree root.

If the objects are sorted already, the bulk-loading algorithm has an I/O cost of $O(n/B)$. Otherwise, the bulk-loading algorithm has asymptotically the same I/O cost as external sort, which is $O(n/B \cdot \log_B n)$. Notice that even if the bulk-loading algorithm performs a sorting first, it is still B times faster than inserting objects one at a time into the structure.

15.5.4 Aggregation Query in a B+-tree

The B+-tree can also be used to answer the *aggregation query*: "given a key range R, find the aggregate value of objects whose keys are in R". The standard SQL supports

the following aggregation operators: COUNT, SUM, AVG, MIN, MAX. For instance, the COUNT operator returns the number of objects in the query range. Here AVG can be computed as SUM/AVG. Thus we focus on the other four aggregate operators.

Since the B+-tree efficiently supports the range query, it makes sense to utilize it to answer the aggregation query as well. Let's first look at some concepts.

Associated with each aggregate operator, there exists a *init_value* and an *aggregate* function. The *init_value* is the aggregate for an empty set of objects. For instance, the *init_value* for the COUNT operator is 0. The *aggregate* function computes the aggregate value. There are two versions. One version takes two aggregate values of object set S_1 and S_2, and computes the aggregate value of set $S_1 \cup S_2$. Another version takes one aggregate value of set S_1 and an object o and computes the aggregate value of $S_1 \cup \{o\}$. For instance, if we know $COUNT_1$ and $COUNT_2$ of two sets, the COUNT for the whole set is $COUNT_1 + COUNT_2$. The COUNT of subset 1 added with an object o is $COUNT_1 + 1$. The *init_value* and the *aggregate* functions for COUNT, SUM, MIN, and MAX are shown below.

- COUNT operator:
  - $init\_value = 0$
  - $aggregate(COUNT_1, COUNT_2) = COUNT_1 + COUNT_2$
  - $aggregate(COUNT_1, object) = COUNT_1 + 1$
- SUM operator:
  - $init\_value = 0$
  - $aggregate(SUM_1, SUM_2) = SUM_1 + SUM_2$
  - $aggregate(SUM_1, object) = SUM_1 + object.value$
- MIN operator:
  - $init\_value = +\infty$
  - $aggregate(MIN_1, MIN_2) = min\{MIN_1, MIN_2\}$
  - $aggregate(MIN_1, object) = min\{MIN_1, object.value\}$
- MAX operator:
  - $init\_value = -\infty$
  - $aggregate(MAX_1, MAX_2) = max\{MAX_1, MAX_2\}$
  - $aggregate(MAX_1, object) = max\{MAX_1, object.value\}$

The B+-tree can support the aggregation query in the following way. We keep a temporary aggregate value, which is initially set to be *init_value*. A range search is performed on the B+-tree. For each object found, its value is aggregated with the temporary aggregate value on-the-fly. When all objects whose keys are in the query range are processed, this temporary aggregate value is returned.

However, this approach is not efficient, as the I/O cost is $O(\log_B n + t/B)$, which is linear to the number of objects divided by B. If the query range is large, the algorithm needs to access too many disk pages. It is ideal to find some approach whose query performance is independent to the size of the objects in the query range.

A better way is to store the local aggregate values in the tree. In more detail, along with each child pointer, we store the aggregate value of all objects in the corresponding sub-tree. By doing so, if the query range fully contains the key range of a sub-tree, we take the associated local aggregate value and avoid browsing the sub-tree. We call such a B+-tree

with extra aggregate information the ***aggregation B+-tree***. The algorithm to perform a aggregation query using the aggregation B+-tree is shown below.

Algorithm *Aggregation*(x, R)
Input: a node x of an aggregation B+-tree, the query key range R.
Action: Among objects in the sub-tree rooted by x, compute the aggregate value of objects whose keys belong to R.

1. Initialize the temporary aggregation value v as *init_value*.
2. **if** x is a leaf node
   (a) For every object o in x where $o.value \in R$, $v = aggr(v, o)$.
3. **else**
   (a) **for** every child pointer $x.child[i]$
      i. **if** the key range of $x.child[i]$ is contained in R
         $v = aggregate(v,\ x.child[i].aggr)$
      ii. **else if** the key range of $x.child[i]$ intersects R
         $y = DiskRead(x.child[i])$
         $v = aggregate(v,\ Aggregation(y, R))$
      iii. **end if**
   (b) **end for**
4. **return** v.

The algorithm starts with examining the root node. Here the child pointers are divided into three groups. (1) There are at most two child pointers whose key ranges intersect the query range R. (2) The child pointers between them have key ranges fully contained in R. (3) The child pointers outside of them have key ranges non-intersecting with R.

For child pointers in group (2), the local aggregate stored at the child pointer (represented by $x.child[i].aggr$) is aggregated to the temporary aggregate value and the examination of the sub-tree is avoided. This is shown in step 3(a)i of the algorithm. For child pointers in group (3), no object in the sub-trees will contribute to the query and the examination of the sub-trees are also avoided.

For each of the two child pointers whose key ranges intersect R, a recursive call to *Aggregation* is performed. This is shown in step 3(a)ii of the algorithm. If we go one level down, in each of the two child nodes, there can be at most one child pointer whose key range intersects R. Take the left child node as an example. If there is a child pointer whose key range intersect R, all child pointers to the left of it will be outside of R and all child pointers to the right of it will be fully contained in R. Thus the algorithm examines two paths from root to leaf.

Theorem: *The I/O cost of the Aggregation query algorithm is* $O(\log_B n)$.

The above theorem shows that the aggregation query performance is independent of the number of objects in the query range.

References

[1] R. Bayer and E. M. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Informatica*, 1, 1972.

[2] D. Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2), 1979.
[3] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data (SIGMOD)*, 1984.
[4] D. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison Wesley, 1973.