Lecture Notes

Selected Topics in Data Structures (Ausgewählte Kapitel aus Datenstrukturen)

Michiel Smid

Max-Planck-Institut für Informatik

D-66123 Saarbrücken, Germany

E-mail: `michiel@mpi-sb.mpg.de`

Preface

This text contains the lecture notes for the course *Ausgewählte Kapitel aus Datenstrukturen*, which was given by the author at the Universität des Saarlandes during the winter semester 1993/94. The course was intended for 3rd/4th year students having some basic knowledge in the field of algorithm design. The course was accompanied by Übungen. The Übungsaufgaben are given in Chapter 5. The author thanks Stefan Denne for typing the manuscript and for producing the figures.

Contents

Chapter 1

Skip Lists: a randomized dictionary

We consider the *dictionary problem*: Given a set S of real numbers store them in a data structure such that the following three operations can be performed efficiently:

Search(x): Given a real number x, report the maximal element of $S \cup \{-\infty\}$ that is at most equal to x. (By introducing $-\infty$, this operation is always well-defined.)

Insert(x): Given a real number x, insert it into the data structure. (Hence, we set $S := S \cup \{x\}$.)

Delete(x): Given a real number x, delete it from the data structure. (Hence, we set $S := S \setminus \{x\}$.)

The standard data structure for this problem is a balanced binary search tree. It solves the dictionary problem with $O(\log n)$ worst-case time for each of the three operations and it uses $O(n)$ space. (Here, n denotes the size of the current set S.) Well known classes of balanced binary search trees are AVL-trees, BB[α]-trees and red-black-trees. Since these trees solve the dictionary problem optimally, we may think that the story ends here. However, anyone who has *implemented* a specific class of balanced trees will have noticed that especially the update algorithms are not trivial to code at all. Usually, the tree is rebalanced by means of rotations and double rotations. Moreover, each of these two types of rotations has a "left" and a "right" version. That is, we have to distinguish between different cases and, therefore, have to be careful.

This leads to the question whether there is an optimal data structure for the dictionary problem that is easy to implement and that is also fast in practice. In this chapter, we will introduce such a data structure: the *Skip List*. These were invented by William Pugh. Skip lists use *randomization*, i.e., they use the outcomes of random coin flips. As we will see, we (the programmer) do not have to worry about balancing: the coin flips take care that the data structure is balanced, at least with very high probability.

The rest of this chapter is organized as follows. We first define skip lists and give the algorithms that work on them. Then we give the intuition why skip lists are efficient. Finally, we give complete proofs of the complexity. Our proofs will use various notions from probability theory. These notions are recalled in Section 1.3.

1.1 Skip Lists

Throughout this chapter, we assume that we can generate random independent bits. Each bit can be generated in unit time. Put it differently, we have a fair coin. If we flip it, then we obtain a one (heads) with probability 1/2 and a zero (tails) with probability 1/2. The outcome of a coin flip is independent of previous outcomes. By flipping the coin repeatedly, we obtain a sequence of independent bits.

Let S be a set of n real numbers. We construct a sequence S_1, S_2, ..., of sets, as follows:

1. $S_1 = S$.

2. Let $i \geq 1$ and assume that S_i has been constructed already. Flip the coin independently for each element of S_i. Then S_{i+1} is the set of all elements of S_i for which the coin flip produced a one.

3. The construction stops as soon as a set S_i is empty.

Let h be the number of sets that are constructed. Then

$$\emptyset = S_h \subseteq S_{h-1} \subseteq \ldots \subseteq S_2 \subseteq S_1 = S.$$

The Skip List for S consists of the following:

1. For each $1 \leq i \leq h$, the elements of $S_i \cup \{-\infty\}$ are stored in sorted order in a linked list L_i.

2. For each $1 < i \leq h$, and each element x in L_i, there is a pointer from x to its occurrence in L_{i-1}.

Here is an example. Suppose $S = \{1, 2, 5, 7, 8, 9, 11, 12, 14, 17, 19, 20\}$. Flipping coins might lead to the sets $S_1 = S$, $S_2 = \{1, 2, 5, 8, 11, 17, 20\}$, $S_3 = \{2, 5, 11, 20\}$, $S_4 = \{11\}$ and $S_5 = \emptyset$. The corresponding skip list is shown in Figure 1.1.

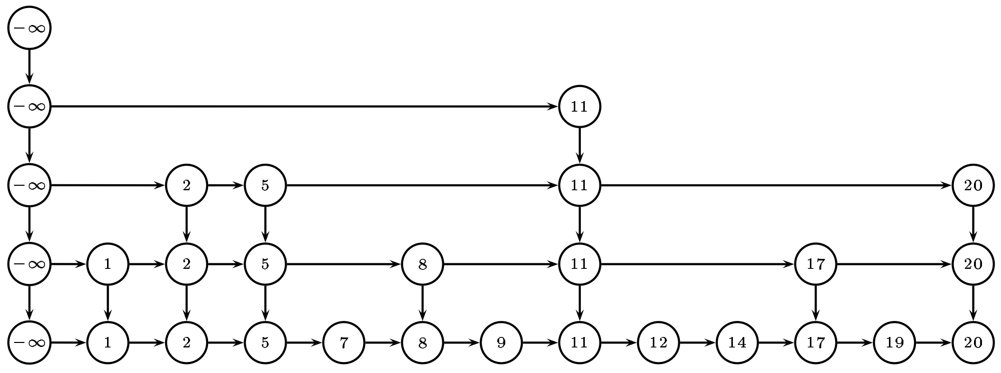

Figure 1.1: *A skip list.*

Observation 1.1.1 *The construction of the sets S_i, $i \geq 1$, defines a probability distribution on skip lists. In our example, the probability that the skip list of Figure 1.1 is obtained is exactly the probability that the coin flipping process gives the sets $S_2 = \{1, 2, 5, 8, 11, 17, 20\}$, $S_3 = \{2, 5, 11, 20\}$, $S_4 = \{11\}$ and $S_5 = \emptyset$.*

We can search in a skip list as follows. Let $x \in \mathbb{R}$. Recall that we want to find the maximal element of $S \cup \{-\infty\}$ that is at most equal to x. The search algorithm successively locates x in the lists L_h, L_{h-1}, ..., L_1:

1. Let y_h be the only element of L_h.
2. For $i = h, h-1, \ldots, 2$:
   (a) Follow the pointer from y_i in L_i to its occurrence in L_{i-1}.
   (b) Starting in y_i, walk to the right along L_{i-1}, until an element is encountered that is larger than x or the end of L_{i-1} is reached. Let y_{i-1} be the last encountered element in L_{i-1} that is at most equal to x.
3. Output y_1.

See Figure 1.2 for an example. There, we search for the number 9. The search path consists of the dashed arrows. The y-variables have values $y_5 = -\infty$, $y_4 = -\infty$, $y_3 = 5$, $y_2 = 8$, $y_1 = 9$.

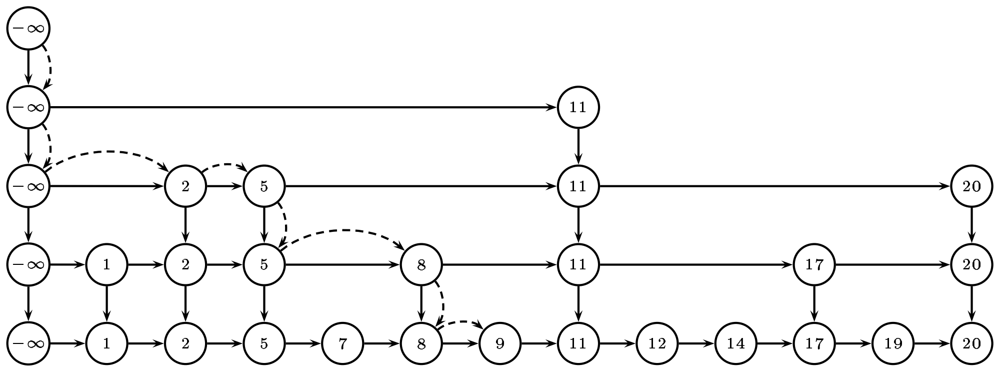

Figure 1.2: *Searching for 9.*

Exercise 1.1.1 Convince yourself that the search algorithm is correct.

Before we turn to the insert and delete algorithms, we give an alternative construction of the sets S_i:

1. For each element x in S, flip the coin until a zero comes up.

2. For $i \geq 1$, S_i is the set of all elements in S for which we flipped the coin at least i times.

Note that the sets S_i, $i \geq 1$, completely determine the skip list. The alternative construction also defines a probability distribution on skip lists.

Exercise 1.1.2 Convince yourself that the two given constructions of the sets S_i define the same probability distribution on skip lists. (Hint: In both constructions, the coin flips are independent.)

The alternative construction immediately suggests the following insert algorithm. Let $x \in \mathbb{R}$ be the element to be inserted.

1. Run the search algorithm on x. Let $y_h, y_{h-1}, \ldots, y_1$ be the elements of $L_h, L_{h-1}, \ldots, L_1$, respectively, that are computed by this algorithm. If $x = y_1$, then $x \in S$ and nothing has to be done. So assume that $x \neq y_1$.

2. Flip the coin until a zero comes up. Let l be the number of coin flips.

3. For each i, $1 \leq i \leq \min(l, h)$, add x to the list L_i, immediately after y_i.

4. If $l \geq h$, then create new lists L_{h+1}, L_{h+2}, ..., L_{l+1} storing the sets $S_{h+1} \cup \{-\infty\}$, $S_{h+2} \cup \{-\infty\}$, ..., $S_{l+1} \cup \{-\infty\}$, where $S_{h+1} = S_{h+2} = \ldots = S_l = \{x\}$ and $S_{l+1} = \emptyset$.

5. For each $1 < i \leq l$, give x in L_i a pointer to its occurrence in L_{i-1}.

6. If $l \geq h$, then for each $h+1 \leq i \leq l+1$, give $-\infty$ in L_i a pointer to its occurrence in L_{i-1}.

7. Set $h := \max(h, l+1)$.

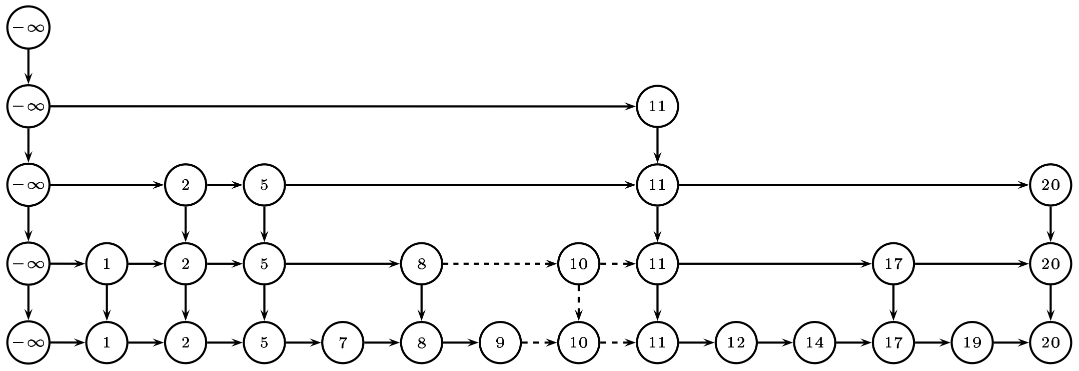

Figure 1.3: *Inserting 10. The first flip gives a one, the second a zero. Hence, 10 is added to the first two levels.*

In Figure 1.3, the skip list that arises by inserting 10 into the skip list of Figure 1.1 is depicted. The dashed pointers are the new ones.

The delete algorithm is similar. Suppose we want to delete element x.

1. Run the search element on x. Let $y_h, y_{h-1}, \ldots, y_1$ be the elements of $L_h, L_{h-1}, \ldots, L_1$, respectively, that are computed by this algorithm. If $x \neq y_1$, then $x \notin S$ and nothing has to be done. So assume that $x = y_1$.

2. For each $1 \leq i \leq h$ such that $x = y_i$, delete y_i from the list L_i.

3. For $i = h,\ h-1,\ \ldots$: if L_{i-1} only stores $-\infty$, delete the list L_i and set $h := h - 1$.

Exercise 1.1.3 Delete 11 from the skip list of Figure 1.3.

This concludes the description of the algorithms for searching and updating a skip list. Note that the word "balance" does not occur anywhere: as mentioned already, our coin takes care of this; we do not have to worry about it.

In the next sections, we will analyze the complexity of the skip list. Of course, we have to say what we mean by that. First consider the *size* of the data structure. It is clear that this size depends on the results of the coin flips that are made during the construction of the skip list. There is no *worst-case* upper bound on the size. Given a set S of real numbers, the size of a skip list for S is a *random variable.* We are interested in the *expected value* of this variable.

The running time of the search algorithm is also a random variable. The *expected search time* is the expected value of this variable. The expectation is computed by averaging over all possible outcomes of the coin flips.

Exercise 1.1.4 Suppose an adversary generates search and update operations. He wants to construct a sequence of operations that take much time to process. Assume the adversary knows the outcomes of our coin flips. Show that he can generate a sequence of update operations, followed by one search operation, such that this final operation takes linear time.

As a final remark, we have given two constructions of the sets S_i, $i \geq 1$. Both constructions define the same probability distribution on skip lists. Therefore, if we analyze the skip list, we can use properties of both constructions. As we will see, depending on what we want to prove, the properties of one construction may be more appropriate than those of the other one.

1.2 Why Skip Lists are efficient: the intuition

We start with considering the number of sets S_i, i.e., the value of h. According to the first construction, we get the set S_{i+1} by taking all elements of S_i for which the coin flip produced a one. Hence, we expect that the size of S_{i+1} is about half the size of S_i. From this, we expect that the value of h is $O(\log n)$.

Let x be any element of S. How many sets S_i contain x? According to the second construction, we flip a coin until a zero comes up. We expect that this happens after a few flips. Hence, we expect that x is contained in only few sets S_i. That is, each element of S is expected to be stored in only few lists and, therefore, the size of the skip list is $O(n)$. This also follows from the fact that $|S_{i+1}| \approx |S_i|/2$ (at least, we expect this), because this implies that the size of the skip list is proportional to $\sum_i |S_i| \approx \sum_i n/2^i = O(n)$.

Next let us consider the cost of searching for a real number x. Let C_i be the number of elements in the list L_i that are inspected by the algorithm. Then the search cost is proportional to $\sum_{i=1}^{h} C_i$. Consider a fixed i. What value of C_i do we expect? Recall that y_i and y_{i+1} are the largest elements of L_i and L_{i+1} that are at most equal to x, respectively. Let y'_{i+1} be the successor of y_{i+1} in L_{i+1}. (We assume for simplicity that y_{i+1} is not the maximal element of L_{i+1}.) Note that $y'_{i+1} > x$. Moreover, C_i is equal to the number of elements in L_i that are to the right of y_{i+1} and to the left of the successor of y_i (including this successor). See Figure 1.4. The dashed arrows form a part of the search path.

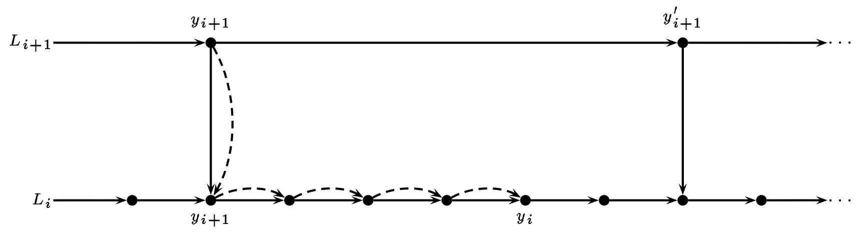

Figure 1.4: *The search algorithm at level i.*

Assume that C_i is large, say 100. Our first construction algorithm implies that all coin flips for the 99 elements of L_i that are to the right of y_{i+1} and to the left of y_i (including y_i) produced a zero. Moreover, the coin flip for y_{i+1} produced a one. But this is extremely unlikely to happen: the probability is $(1/2)^{100}$. Hence, we expect C_i to be small. That is, the search cost in the list L_i is small. Since we expect to have $O(\log n)$ such lists, the entire search algorithm should take $O(\log n)$ time. (Here, we multiply two expectations, which is in general not allowed. But remember that this section only gives intuition, not real proofs.)

The costs of the insert and delete algorithms are proportional to the number of lists L_i plus the cost of the search algorithm. Therefore, we also expect these costs to be $O(\log n)$.

Note that in this section, we only gave the intuition why skip lists are expected to be efficient. In Section 1.4, we will give rigorous proofs. First, in the next section, we recall and develop some notions from probability theory.

1.3 Some notions from probability theory

We assume that the reader has some elementary knowledge about probability theory. We recall the basic notions.

Let U be a *sample space*. The elements of U are called *elementary events*. They can be viewed as possible outcomes of an experiment. An *event* is a subset of U. Two events A and B are called *mutually exclusive* if $A \cap B = \emptyset$.

A *probability distribution* Pr on U is a function that maps events to real numbers such that

1. $\Pr(A) \geq 0$ for any event A,

2. $\Pr(U) = 1$,

3. for any finite or countably infinite sequence $A_1, A_2, \ldots$ of events that are pairwise mutually exclusive, $\Pr(\cup_i A_i) = \sum_i \Pr(A_i)$.

The real number $\Pr(A)$ is the *probability* of event A. Two events A and B are called *independent* if $\Pr(A \cap B) = \Pr(A) \cdot \Pr(B)$.

Exercise 1.3.1 Prove the following statements:
(1) The empty event $\emptyset$ has probability $\Pr(\emptyset) = 0$.
(2) If A and B are events such that $A \subseteq B$, then $\Pr(A) \leq \Pr(B)$.
(3) For any event A, $\Pr(U \setminus A) = 1 - \Pr(A)$.
(4) For events A and B, $\Pr(A \cup B) = \Pr(A) + \Pr(B) - \Pr(A \cap B) \leq \Pr(A) + \Pr(B)$.

If A and B are events such that $\Pr(B) > 0$, then the *conditional probability* of event A given that event B occurs is defined as

$$\Pr(A \mid B) := \frac{\Pr(A \cap B)}{\Pr(B)}.$$

Lemma 1.3.1 *Let B_1, B_2, ... be a finite or countably infinite sequence of events that are pairwise mutually exclusive such that $\Pr(B_i) > 0$ for all i and $\sum_i \Pr(B_i) = 1$. Let A be any event. Then,*

$$\Pr(A) = \sum_i \Pr(A \mid B_i) \cdot \Pr(B_i).$$

Proof: Let $B := \cup_i B_i$ and $B^* := U \backslash B$. Since $A = (A \cap B) \cup (A \cap B^*)$, we have $\Pr(A) = \Pr(A \cap B) + \Pr(A \cap B^*)$. It follows from our assumptions that $\Pr(B) = 1$ and, hence, $\Pr(B^*) = 0$. But then, since $\Pr(A \cap B^*) \leq \Pr(B^*)$, we also have $\Pr(A \cap B^*) = 0$. Therefore,

$$\begin{aligned} \Pr(A) &= \Pr(A \cap B) \\ &= \Pr(\cup_i (A \cap B_i)) \\ &= \sum_i \Pr(A \cap B_i) \\ &= \sum_i \Pr(A \mid B_i) \cdot \Pr(B_i). \end{aligned}$$

■

Why did we define the notion of conditional probability? One reason is the following: In some applications, it is difficult to compute $\Pr(A)$ directly. In such cases, we define events B_i, $i \geq 1$, such that the conditions of Lemma 1.3.1 are satisfied, and $\Pr(A \mid B_i)$ is easy to compute. Using Lemma 1.3.1, this allows us to compute $\Pr(A)$.

Let U be a finite or countably infinite sample space. A *random variable* X is a function from U to $\mathbb{R}$. If x is a real number, then the event "$X = x$" is defined as $\{u \in U : X(u) = x\}$. This implies that

$$\Pr(X = x) = \sum_{u \in U : X(u) = x} \Pr(\{u\}).$$

Similarly, if X and Y are random variables, then for any $x, y \in \mathbb{R}$,

$$\Pr(X = x \wedge Y = y) = \sum_{u \in U : X(u) = x \wedge Y(u) = y} \Pr(\{u\}).$$

Exercise 1.3.2 Convince yourself that

$$\Pr(X \geq x) = \sum_{u \in U : X(u) \geq x} \Pr(\{u\}).$$

Exercise 1.3.3 Prove that $\Pr(X = x) = \sum_y \Pr(X = x \wedge Y = y)$. (Hint: Use Lemma 1.3.1. Convince yourself that the summation has a finite or countably infinite number of terms.)

The random variables X and Y are called *independent* if

$$\Pr(X = x \wedge Y = y) = \Pr(X = x) \cdot \Pr(Y = y)$$

for all x and y.

A sequence $(X_i)_{i \geq 1}$ of random variables is called *pairwise independent* if for all $1 \leq i < j$ and all x and y,

$$\Pr(X_i = x \wedge X_j = y) = \Pr(X_i = x) \cdot \Pr(X_j = y).$$

The sequence is called *mutually independent*, if for all $n \geq 2$, $1 \leq i_1 < i_2 < \ldots < i_n$, and $x_1, x_2, \ldots, x_n$,

$$\Pr(\bigwedge_{j=1}^{n} (X_{i_j} = x_j)) \;=\; \prod_{j=1}^{n} \Pr(X_{i_j} = x_j).$$

See also Übungsaufgabe 5.3.

Given random variables X and Y, we can define new ones, such as $X + Y$, $X \cdot Y$, e^X, in the obvious way.

The *expected value* of a random variable X is defined as

$$E(X) := \sum_x x \cdot \Pr(X = x),$$

provided the series converges absolutely.

Exercise 1.3.4 Let X and Y be random variables and let a be a real number.
(1) Prove that $E(X + Y) = E(X) + E(Y)$.
(2) Prove that $E(a \cdot X) = a \cdot E(X)$.

(3) Assume that X and Y are independent. Prove that $E(X \cdot Y) = E(X) \cdot E(Y)$.

Exercise 1.3.5 In this exercise, we introduce and analyze the *geometric distribution*. Let $0 < p < 1$ be a real number. We have a coin that comes up zero with probability p and one with probability $1-p$. We flip this coin independently until a zero comes up. Let X be the random variable whose value is the number of times we flip the coin.

(1) Prove that $\Pr(X = k) = p(1-p)^{k-1}$ for any $k \geq 1$. Any probability distribution that satisfies this equation is called a geometric distribution.

(2) Prove that $E(X) = 1/p$.

The following indentity turns out to be useful.

Lemma 1.3.2 *Let X be a random variable that takes values in $\{0, 1, 2, \ldots\}$. Then,*

$$E(X) = \sum_{k=1}^{\infty} \Pr(X \geq k).$$

Proof: Since $\Pr(X = k) + \Pr(X \geq k+1) = \Pr(X \geq k)$, we get

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

■

If we want to compute the expected value of X according to the definition, then we have to know the probability that X has value k. In many applications, it is much easier to compute the probability that X is at least equal to k. In such situations, Lemma 1.3.2 should be applied.

Exercise 1.3.6 Let X and Y be random variables such that $X(u) \leq Y(u)$ for all $u \in U$.

(1) Prove that $\Pr(X \geq t) \leq \Pr(Y \geq t)$ for any $t \in \mathbb{R}$.
(2) Assume that X and Y take values in $\{0,1,2,\dots\}$. Prove that $E(X) \leq E(Y)$.

Lemma 1.3.3 *Let X be a random variable and let f be any function. Then*

$$E(f(X)) = \sum_x f(x) \cdot \Pr(X = x).$$

Proof: Assume that X takes on the values $x_1, x_2, x_3, \dots$ Let $y_1, y_2, y_3, \dots$ be the elements of the set $\{f(x_i) : i \geq 1\}$. Hence, the y_i's are distinct. Then, by definition,

$$E(f(x)) = \sum_{i=1}^{\infty} y_i \cdot \Pr(f(X) = y_i).$$

Since

$$\Pr(f(x) = y_i) = \sum_{k:f(x_k)=y_i} \Pr(X = x_k),$$

it follows that

$$E(f(X)) = \sum_{i=1}^{\infty} \sum_{k:f(x_k)=y_i} f(x_k) \cdot \Pr(X = x_k).$$

Since $\cup_{i=1}^{\infty}\{k : f(x_k) = y_i\} = \{1,2,3,\dots\}$ and the set on the left-hand side is a union of pairwise disjoint sets, we get

$$E(f(X)) = \sum_{j=1}^{\infty} f(x_j) \cdot \Pr(X = x_j).$$

This proves the lemma. ■

Let X and Y be random variables. The *conditional expected value* of X given that $Y = y$, is defined as

$$E(X \mid Y = y) := \sum_x x \cdot \Pr(X = x \mid Y = y).$$

Lemma 1.3.4

$$E(X) = \sum_y E(X \mid Y = y) \cdot \Pr(Y = y).$$

Proof: We know from Lemma 1.3.1 that

$$\Pr(X = x) = \sum_y \Pr(X = x \mid Y = y) \cdot \Pr(Y = y).$$

Therefore,

$$\begin{aligned} E(X) &= \sum_x x \cdot \Pr(X = x) \\ &= \sum_x x \sum_y \Pr(X = x \mid Y = y) \cdot \Pr(Y = y) \\ &= \sum_y \Pr(Y = y) \sum_x x \cdot \Pr(X = x \mid Y = y) \\ &= \sum_y \Pr(Y = y) \cdot E(X \mid Y = y). \end{aligned}$$

■

1.4 Why Skip Lists are efficient: the proofs

After our excursion to probability theory we are ready to analyze skip lists rigorously. The size of a skip list and the running times of the search and update algorithms are random variables. We will prove that their expected values are bounded by $O(n)$ and $O(\log n)$, respectively.

Recall that h denotes the number of sets S_i, $i \geq 1$, that result from our probabilistic construction. (Note that $S_h = \emptyset$). We give an upper bound on the expected value of h.

Let x be an element of S and let $h(x)$ be the number of sets S_i that contain x. Our second construction implies that $h(x)$ is distributed according to a geometric distribution with $p = 1/2$. (See Exercise 1.3.5.) Therefore, $\Pr(h(x) = k) = (1/2)^k$ and $E(h(x)) = 2$. That is, element x is expected to be contained only in S_1 and S_2.

Clearly, $h = 1 + \max\{h(x) : x \in S\}$. From $E(h(x)) = 2$ for any $x \in S$, however, we *cannot* conclude that the expected value of h is three. (See also Übungsaufgabe 5.2.)

We estimate $E(h)$ as follows. Again, consider a fixed element x of S. It follows from the second construction that for any $k \geq 1$, $h(x) \geq k$ if and only if the first $k - 1$ coin flips for x produced a one. That is,

$\Pr(h(x) \geq k) = (1/2)^{k-1}$. It is clear that $h \geq k+1$ if and only if there is an $x \in S$ such that $h(x) \geq k$. Therefore (see Exercise 1.3.1)

$$\Pr(h \geq k+1) \leq n \cdot \Pr(h(x) \geq k) = \frac{n}{2^{k-1}}.$$

This estimate does not make sense for $k < 1 + \log n$. For these values of k, we can use the trivial upper bound $\Pr(h \geq k+1) \leq 1$. Applying Lemma 1.3.2, we get

$$E(h) = \sum_{k=0}^{\infty} \Pr(h \geq k{+}1) = \sum_{k=0}^{\lceil \log n \rceil} \Pr(h \geq k{+}1) + \sum_{k=1+\lceil \log n \rceil}^{\infty} \Pr(h \geq k{+}1).$$

The first summation on the right hand side is at most equal to $1 + \lceil \log n \rceil$. The second summation is bounded from above by

$$\sum_{k=1+\lceil \log n \rceil}^{\infty} \frac{n}{2^{k-1}} = n(1/2)^{\lceil \log n \rceil - 1} \leq n(1/2)^{\log n - 1} = 2.$$

Hence we have proved that $E(h) \leq 3 + \lceil \log n \rceil$.

The expected size of a skip list can easily be computed: Let M denote the total size of the sets $S_1, S_2, \ldots, S_h$. Then $M = \sum_{x \in S} h(x)$ and, by the linearity of expectation (see Exercise 1.3.4),

$$E(M) = \sum_{x \in S} E(h(x)) = \sum_{x \in S} 2 = 2n.$$

If M' denotes the total number of nodes in the skip list, then M' is equal to M (= the number of nodes in the lists $L_i \setminus \{-\infty\}, 1 \leq i \leq h$) plus h (= the number of occurrences of $-\infty$).
Hence,

$$E(M') = E(M + h) = E(M) + E(h) \leq 2n + 3 + \lceil \log n \rceil.$$

Since each node of the skip list contains a constant amount of information (an element of $S \cup \{-\infty\}$ and at most two pointers), this proves that its expected size is bounded by $O(n)$.

Next, we estimate the expected search cost. Let x be a real number. As in Section 1.2, let C_i denote the number of elements in the list L_i

that are inspected by the algorithm when searching for x. (We do not count the element of L_i in which the algorithm starts walking to the right. Hence, C_i counts comparisons between x and elements of S. Moreover, $C_h = 0$.) The search cost is proportional to $\sum_{i=1}^{h}(1+C_i)$.

We have to be careful: It is not clear that the expected value of this summation is equal to $\sum_{i=1}^{E(h)} E(1+C_i)$. The reason is that h itself is a random variable: The linearity of expectation was only proved for the summation of a *fixed* number of random variables. (See also Übungsaufgaben 5.3 and 5.4.)

The trick is to take a fixed integer A and analyze the search costs up to level A and above level A separately. (Later, we choose A such that we get a good upper bound.)

So let A be a positive integer. We first estimate the expected search cost above level A, i.e., the total cost in the lists L_{A+1}, L_{A+2}, ..., L_h. Since this cost is at most equal to the total size of these lists, its expected value is at most equal to the expected value of $M_A := \sum_{i=A+1}^{h} |L_i|$. (See Exercise 1.3.6.)

How do we estimate the expected value of M_A? We first note that the lists L_i, $A+1 \leq i \leq h$, form a skip list for the set S_{A+1}. That is, these lists have the same properties as a skip list that is built only for S_{A+1}. Here, the notion of conditional expected value comes in. According to Lemma 1.3.4, we have

$$E(M_A) = \sum_{k=0}^{n} E(M_A \mid |S_{A+1}| = k) \cdot \Pr(|S_{A+1}| = k).$$

Note that $E(M_A \mid |S_{A+1}| = k)$ is the expected size of a skip list for a set of k elements. We have seen already that this expected value is $O(k)$.

So it remains to compute $\Pr(|S_{A+1}| = k)$. Since $|S_{A+1}| = k$ if and only if out of the n elements of S exactly k "reach" level $A+1$, we have

$$\Pr(|S_{A+1}| = k) = \binom{n}{k} (1/2)^{Ak} \left(1 - (1/2)^A\right)^{n-k}.$$

Setting $p = (1/2)^A$, we infer that the expected value of M_A is proportional to

$$\sum_{k=0}^{n} k \cdot \binom{n}{k} p^k (1-p)^{n-k} \quad = \quad \sum_{k=1}^{n} n \cdot \binom{n-1}{k-1} p^k (1-p)^{n-k}$$

$$\begin{aligned}
&= n \cdot p \sum_{k=0}^{n-1} \binom{n-1}{k} p^k (1-p)^{n-1-k} \\
&= n \cdot p \, (p + (1-p))^{n-1} \\
&= n \cdot p.
\end{aligned}$$

Putting everything together, we have proved that the expected search cost above level A is bounded by $O(n/2^A)$.

Next we estimate the expected search cost in the lists L_1, L_2, …, L_A. Recall that C_i is the number of elements of L_i that are inspected when searching for x. How do we compute $E(C_i)$? Again, we use conditional expectations: Let $l_i(x)$ be the number of elements in L_i that are at most equal to x. Then

$$E(C_i) = \sum_{k=1}^{n} E(C_i \mid l_i(x) = k) \cdot \Pr(l_i(x) = k).$$

Assume that $l_i(x) = k$. Also, assume that there is an element in L_i that is larger than x. Then, $C_i = j$ if and only if the largest $j-1$ elements of L_i that are at most equal to x do not appear in L_{i+1}, but the element that immediately precedes these $j-1$ elements does appear in L_{i+1}. (Note that the latter element may be $-\infty$, which always appears in L_{i+1}.) Hence,

$$\Pr(C_i = j \mid l_i(x) = k) \leq (1/2)^{j-1}, \; 0 \leq j \leq k.$$

This inequality also holds if x is at least equal to the maximal element of L_i. From this we obtain

$$\begin{aligned}
E(C_i \mid l_i(x) = k) &= \sum_{j=0}^{k} j \cdot \Pr(C_i = j \mid l_i(x) = k) \\
&\leq \sum_{j=0}^{k} j/2^{j-1} \\
&\leq 4.
\end{aligned}$$

This, in turn, implies that

$$E(C_i) \leq \sum_{k=1}^{n} 4 \cdot \Pr(l_i(x) = k) = 4.$$

It follows that the expected search cost up to level A is proportional to

$$E(\sum_{i=1}^{A}(1+C_i)) = \sum_{i=1}^{A}(1+E(C_i)) \leq 5A.$$

Summarizing, we have proved that the expected time to search for element x is bounded by (again, we use the linearity of expectation)

$$O(n/2^A + A).$$

This upper bound holds for any A. We take $A = \lceil \log n \rceil$ and infer that the expected search time in a skip list is $O(\log n)$. (In Section 1.6, we give an alternative proof of this. See Remark 1.6.1.)

Finally, we consider the expected update cost. It is easy to see that the expected time to insert or delete an element x in a skip list is proportional to the expected cost of searching for x. Hence, the expected update time is $O(\log n)$.

There is an important remark to be made here: An insertion (resp. deletion) of an element x results in a skip list that has the same probability distribution as a skip list for $S \cup \{x\}$ (resp. $S \setminus \{x\}$) that is built by one of our constructions. Hence, after a sequence of updates has been performed, the data structure still "behaves" as if it were just built. We analyzed skip lists under the assumption that they were built by one of our constructions. As a result, the space and time bounds we derived also hold for skip lists that have been changed by a sequence of updates.

The next theorem summarizes the results of this section.

Theorem 1.4.1 *Let S be a set of n real numbers and let SL be a skip list for S.*

1. *The expected number of levels in SL is $O(\log n)$.*
2. *The expected size of SL is $O(n)$.*
3. *For any $x \in \mathbb{R}$, it takes $O(\log n)$ expected time to search for x in SL.*
4. *We can insert and delete an element in SL in $O(\log n)$ expected time.*

1.5 Tail estimates: Chernoff bounds

In the previous section, we proved bounds for the expected size, search time and update time of a skip list. In this section, we consider so-called *tail estimates.* That is, we estimate the probability that e.g. the actual search time deviates significantly from its expected value. We saw that the expected search time is bounded by $O(\log n)$. Assume for simplicity that the constant in this bound is equal to one. Then we want to estimate the probability that the actual search time is at least $t \cdot \log n$. This probability can be bounded by *Markov's inequality*:

Lemma 1.5.1 *Let X be a random variable that takes non-negative values, and let μ be the expected value of X. Then for any $t > 0$,*

$$\Pr(X \geq t\mu) \leq 1/t.$$

Proof: Let $s = t\mu$. Then

$$\begin{aligned}
\mu &= \sum_x x \cdot \Pr(X = x) \\
&\geq \sum_{x \geq s} x \cdot \Pr(X = x) \\
&\geq \sum_{x \geq s} s \cdot \Pr(X = x) \\
&= s \cdot \Pr(X \geq s).
\end{aligned}$$

∎

Hence, the probability that the actual search time is at least $t \cdot \log n$ is less than or equal to $1/t$. This is not very impressive: The probability that the search time is more than 100 times its expected value is at most 1/100. If this upper bound were tight, then among 100 searches we would expect that one takes 100 times as long as an average search.

In this section, we prove so-called *Chernoff bounds*, which will be used in Section 1.6 to give much better tail estimates. By using more properties of the random variables that determine the search time, we will prove that the probability that it exceeds $t \cdot \log n$ is less than or equal to $n^{-9t/50}$, for $t \geq 50$. (For $t < 50$, the bound is slightly worse.) Hence, in a skip list for 1000 elements, the probability that the search time is more than 100 times its expected value is at most 10^{-54}. In practice, this means that this event will never occur.

Markov's inequality holds for any non-negative random variable. The Chernoff technique applies to random variables X that can be written as a sum $\sum_{i=1}^{n} X_i$ of mutually independent random variables X_i. In such cases, much better bounds can be obtained for $\Pr(X \geq t\mu)$.

So let X_1, X_2, …, X_n be a sequence of n mutually independent random variables and let $X = \sum_{i=1}^{n} X_i$. For any real number λ, the random variables $e^{\lambda X_1}, e^{\lambda X_2}, \ldots, e^{\lambda X_n}$ are also mutually independent. Therefore,

$$E(e^{\lambda X}) = E(e^{\lambda(X_1 + \ldots + X_n)}) = \prod_{i=1}^{n} E(e^{\lambda X_i}).$$

Now let $s > 0$ and $\lambda > 0$. Since $X \geq s$ if and only if $e^{\lambda X} \geq e^{\lambda s}$, we have $\Pr(X \geq s) = \Pr(e^{\lambda X} \geq e^{\lambda s})$. By applying Markov's inequality to the non-negative random variable $e^{\lambda X}$, we get

$$\Pr(X \geq s) = \Pr(e^{\lambda X} \geq e^{\lambda s}) \leq e^{-\lambda s} \cdot E(e^{\lambda X}).$$

This yields

$$\Pr(X \geq s) \leq e^{-\lambda s} \prod_{i=1}^{n} E(e^{\lambda X_i}) \text{ for } s > 0 \text{ and } \lambda > 0. \tag{1.1}$$

This is the basic inequality. To estimate $\Pr(X \geq s)$, we need bounds on $E(e^{\lambda X_i})$. Of course, these bounds depend on the probability distribution of X_i.

We illustrate the technique with the *geometric distribution.* (See Exercise 1.3.5.) We are given a coin that comes up with zero or one, each with probability 1/2. We flip this coin independently until a one comes up. Let T be the number of flips. Then T is distributed according to a geometric distribution, $\Pr(T = k) = (1/2)^k$ for $k \geq 1$, and $E(T) = 2$.

Now assume that we flip the coin until we have obtained a one exactly n times. We denote the number of flips by T_n. (Hence, $T = T_1$.) This random variable T_n is distributed according to a *negative binomial distribution.* Our goal is to estimate $\Pr(T_n \geq s)$.

To apply the Chernoff technique, we must express T_n as a sum of mutually independent random variables. For $1 \leq i \leq n$, let X_i denote the number of flips between the $(i-1)$-st and the i-th ones. (We exclude the flip that gives the $(i-1)$-st one, but we include the flip that gives the i-th one.) Then, $T_n = \sum_{i=1}^n X_i$, each X_i is distributed according to a geometric distribution, and $X_1, X_2, \ldots, X_n$ are mutually independent.

The expected value of T_n follows from the linearity of expectation:

$$E(T_n) = \sum_{i=1}^{n} E(X_i) = \sum_{i=1}^{n} 2 = 2n.$$

Hence Markov's inequality gives $\Pr(T_n \geq (2+t)n) \leq 2/(2+t)$. As we will see, the Chernoff technique gives a much better upper bound.

Let $0 < \lambda < \ln 2$. Then, applying Lemma 1.3.3 with $f(x) = e^{\lambda x}$, we get

$$E(e^{\lambda X_i}) = \sum_{k=1}^{\infty} e^{\lambda k} \cdot \Pr(X_i = k) = \sum_{k=1}^{\infty} (e^{\lambda}/2)^k = \frac{e^{\lambda}}{2-e^{\lambda}}.$$

Now we apply the basic inequality (1.1) with $s = (2+t)n$, where $t > 0$. We get

$$\Pr(T_n \geq (2+t)n) \leq e^{-\lambda(2+t)n} \left(\frac{e^{\lambda}}{2-e^{\lambda}}\right)^n = \left(\frac{e^{-\lambda(1+t)}}{2-e^{\lambda}}\right)^n.$$

This inequality holds for any $0 < \lambda < \ln 2$. Hence, we now choose λ such that the term on the right-hand side is minimal. It turns out that this happens for $\lambda = \ln(1 + \frac{t}{2+t})$. We find that

$$\Pr(T_n \geq (2+t)n) \leq \left(1 + \frac{t}{2}\right)^n \left(1 - \frac{t}{2+2t}\right)^{(1+t)n}.$$

Since $1 - x \leq e^{-x}$ for all x, we have

$$\left(1 - \frac{t}{2+2t}\right)^{1+t} \leq \left(e^{-t/(2+2t)}\right)^{1+t} = e^{-t/2}.$$

Moreover, $1 + t/2 \leq e^{t/4}$ for $t \geq 3$. This proves that for $t \geq 3$

$$\Pr(T_n \geq (2+t)n) \leq e^{tn/4} \cdot e^{-tn/2} = e^{-tn/4}.$$

(Compare this with the bound obtained from Markov's inequality!)

Theorem 1.5.1 *Let X_1, X_2, ..., X_n be mutually independent random variables and assume that each X_i is distributed according to a geometric distribution. Let $T_n = X_1 + X_2 + \ldots + X_n$. Then $E(T_n) = 2n$ and for any $t \geq 3$,*

$$\Pr(T_n \geq (2+t)n) \leq e^{-tn/4}.$$

Corollary 1.5.1 *Let $c \geq 1$ be a constant, and let n be a positive integer. Then for any $s \geq 5$,*

$$\Pr(T_{\lceil c \cdot \ln n \rceil} \geq s \cdot c \cdot \ln n) \leq n^{-(s-2)c/4}.$$

1.6 Tail estimates for skip lists

We use the results of the previous section to prove tail estimates for the size, search time and update time of a skip list.

Consider a skip list for a set S of n elements. Let M denote the total size of the sets S_1, S_2, ..., S_h, and let M' denote the total number of nodes of the skip list. Then, $M' = M + h$. We have seen that the expected size of the skip list is equal to $E(M') = E(M+h) \leq 2n + 3 + \lceil \log n \rceil$. We are interested in the probability that M' is at least equal to $(2+t)n$.

Clearly, if $M' \geq (2+t)n$, then $h \geq tn/2$ or $M \geq (2+t/2)n$. As a result,

$$\Pr(M' \geq (2+t)n) \;\leq\; \Pr(h \geq tn/2) + \Pr(M \geq (2+t/2)n).$$

In Section 1.4, we already estimated the first probability on the right-hand side. There, we proved that $\Pr(h \geq k+1) \leq n/2^{k-1}$ for $k \geq 1$. Hence, for $t \geq 1$ and n sufficiently large,

$$\Pr(h \geq tn/2) \leq \frac{n}{2^{tn/2-2}} = e^{\ln n + 2\ln 2 - (tn/2)\ln 2} \leq e^{-tn/8}.$$

It remains to bound $\Pr(M \geq (2+t/2)n)$. As in Section 1.4, let $h(x)$ be the number of sets S_i, $1 \leq i \leq h$, that contain x. Then, the random variables $h(x)$, $x \in S$, are mutually independent and each one is distributed according to a geometric distribution. Since $M = \sum_{x \in S} h(x)$, Theorem 1.5.1 implies that for $t \geq 6$,

$$\Pr(M \geq (2+t/2)n) \leq e^{-tn/8}.$$

This proves that for $t \geq 6$,

$$\Pr(M' \geq (2+t)n) \leq 2 \cdot e^{-tn/8},$$

i.e., it is extremely unlikely that the size of a skip list deviates much from its expected value.

The analysis of the search time is more complicated. Let $x \in \mathbb{R}$ and let $c \geq 1$ be a constant. We want to estimate the probability that searching for x takes more than $c \cdot \ln n$ steps. We analyze this cost by considering the costs up to level $\lceil c \cdot \ln n \rceil$ and above level $\lceil c \cdot \ln n \rceil$ separately.

Let T_a denote the number of nodes traversed in the levels $1 + \lceil c \cdot \ln n \rceil, 2 + \lceil c \cdot \ln n \rceil, \ldots, h$, when searching for x. We proved in Section 1.4 that the expected value of T_a is bounded by $O(n/2^{\lceil c \cdot \ln n \rceil}) = O(n^{1-c \cdot \ln 2})$. Then, Markov's inequality shows that

$$\Pr(T_a \geq 1) \leq E(T_a) = O(n^{1-c \cdot \ln 2}).$$

Let T_b be the number of nodes traversed in the levels $1, 2, \ldots, \lceil c \cdot \ln n \rceil$, when searching for x. In order to apply Theorem 1.5.1, we have to express T_b as a sum of mutually independent random variables, each one distributed according to a geometric distribution. In order to do this, we give a new construction of our skip list.

Let $S_1 = S = \langle s_{11} < s_{12} < \ldots < s_{1n_1} \rangle$, where $n_1 = n$. Moreover, let $l_1(x)$ be the number of elements of S_1 that are at most equal to x.

Let $i \geq 1$ and assume we have constructed $S_i = \langle s_{i1} < s_{i2} < \ldots < s_{in_i} \rangle$, with $n_i = |S_i|$. Let $l_i(x) = |\{y \in S_i : y \leq x\}|$. We do the following:

Stage 1: We flip our coin $n_i - l_i(x)$ times.

Stage 2: We flip the coin $l_i(x)$ times.

Stage 3: If the previous $l_i(x)$ flips only produced zeros, then we flip the coin until we get a one.

Let f_i be the total number of flips made during these stages, and denote the outcomes by $F_{i1}, F_{i2}, \ldots, F_{if_i}$. Note that $f_i \geq n_i$. We proceed as follows:

1. We define $S_{i+1} := \{s_{ij} : 1 \leq j \leq n_i \wedge F_{ij} = 1\}, n_{i+1} = |S_{i+1}|$ and $l_{i+1}(x) := \{y \in S_{i+1} : y \leq x\}$.

2. We define the random variable X_i as the number of flips after Stage 1 until the first one comes up.

3. We define the random variable C_i as the minimum of $1 + l_i(x)$ and X_i.

The construction stops as soon as an empty set S_{i+1} has been constructed. As usual, we denote the number of sets by h. Hence, we now have sets

$$\emptyset = S_h \subseteq S_{h-1} \subseteq S_{h-2} \subseteq \ldots \subseteq S_2 \subseteq S_1 = S,$$

and random variables X_1, X_2, ..., X_{h-1} and C_1, C_2, ..., C_{h-1}.

This construction defines a probability distribution on skip lists, which is the same as that of our previous two constructions. In fact, the new construction is the same as our first one, except that Stage 3 and the random variables X_i and C_i, $i \geq 1$, have been added.

Let us go back to the analysis of the search time. For convenience, we define $C_i = 0$ for all $i \geq h$. Recall that T_b is the number of nodes traversed in the levels $1, 2, \ldots, \lceil c \cdot \ln n \rceil$. We have

$$T_b = \sum_{i=1}^{\lceil c \cdot \ln n \rceil} C_i.$$

That is, we have written T_b as a sum of random variables. However, the variables $C_1, C_2, \ldots, C_{h-1}$ are not mutually independent: It is easy to see that

$$\Pr(C_{i-1} = 3n/4 \wedge C_i = 3n/4) = 0,$$

but

$$\Pr(C_{i-1} = 3n/4) \cdot \Pr(C_i = 3n/4) \neq 0.$$

Moreover, the C_i's are not distributed according to a geometric distrbution: For $j > l_i(x)$, we have $\Pr(C_i = j) = 0$.

On the other hand, it is easy to see that $C_i \le X_i$ for all $1 \le i \le h-1$. Moreover, the random variables X_i, $1 \le i \le h-1$, are mutually independent and each one is distributed according to a geometric distribution. For $i \ge h$, let X_i also denote a random variable distributed according to a geometric distribution. Since

$$T_b = \sum_{i=1}^{\lceil c \cdot \ln n \rceil} C_i \le \sum_{i=1}^{\lceil c \cdot \ln n \rceil} X_i,$$

we infer

$$\Pr(T_b \ge s \cdot c \cdot \ln n) \le \Pr(\sum_{i=1}^{\lceil c \cdot \ln n \rceil} X_i \ge s \cdot c \cdot \ln n).$$

Corollary 1.5.1 immediately gives

$$\Pr(T_b \ge s \cdot c \cdot \ln n) \le n^{-(s-2)c/4} \le n^{-sc/8},$$

for $c \ge 1$ and $s \ge 5$.

Now we can give the tail estimate for the search time . Let T denote the total number of nodes traversed when searching for x. Then $T = T_a + T_b$. Moreover, if $T \ge 1 + 5c \cdot \ln n$ then $T_a \ge 1$ or $T_b \ge 5c \cdot \ln n$. Hence,

$$\Pr(T \ge 1 + 5c \cdot \ln n) \le \Pr(T_a \ge 1) + \Pr(T_b \ge 5c \cdot \ln n).$$

Our results for the two probabilities on the right-hand side imply that for $c \ge 1$,

$$\Pr(T \ge 1 + 5c \cdot \ln n) = O(n^{1-c \cdot \ln 2} + n^{-5c/8}).$$

Taking $c = t/(5 \ln 2)$, where $t \ge 5 \ln 2 \approx 3.47$, we get

$$\begin{aligned} \Pr(T \ge 1 + t \log n) &= O(n^{1-t/5} + n^{-t/(8 \ln 2)}) \\ &= O(n^{1-t/5} + n^{-9t/50}). \end{aligned}$$

Note that this estimate only makes sense for $t > 5$. This completes the analysis of the search time. Since the update time is proportional to the search time, similar bounds can be proved for it. We summarize our result.

Theorem 1.6.1 *Let S be a set of n real numbers and let SL be a skip list for S.*

1. *For each $t \geq 6$, the probability that SL consists of at least $(2+t)n$ nodes is at most $2 \cdot e^{-tn/8}$.*
2. *For each $t > 5$ and $x \in \mathbb{R}$, the probability that a search in SL for x visits at least $1 + t \log n$ nodes is $O(n^{1-t/5} + n^{-9t/50})$.*
3. *There is a constant c such that for any sufficiently large t, the probability that an insert or delete operation in SL visits at least $t \log n$ nodes is $O(n^{-ct})$.*

Remark 1.6.1 Consider again the random variables C_i and X_i. We have seen that the total number T of nodes visited when searching for x is equal to

$$T = \sum_{i=1}^{h} C_i \leq \sum_{i=1}^{h} X_i.$$

Therefore, the expected search time $E(T)$ is at most equal to $E(\sum_{i=1}^{h} X_i)$. (See Exercise 1.3.6.) The random variables h, X_1, X_2, ..., X_h are mutually independent. (Convince yourself that this is true.) Therefore, by Übungsaufgabe 5.4,

$$E(T) \leq E(\sum_{i=1}^{h} X_i) = E(h) \cdot E(X_1) \leq (3 + \lceil \log n \rceil) \cdot 2 = O(\log n).$$

This gives an alternative proof of the logarithmic expected search time.

1.7 Further reading

Skip lists were invented by Pugh in 1989. See [19, 20, 21]. Another randomized dictionary, based on binary search trees, was introduced in 1989 by Aragon and Seidel [1].

An introduction to probability theory can be found in the book by Cormen, Leiserson and Rivest [4]. The standard book on this topic is Feller [7]. A comprehensive overview of randomized algorithms and data structures, especially in computational geometry, is the book by Mulmuley [16].

Chapter 2

The Union-Find Problem

In this chapter, we consider the well-known *Union-Find Problem*:

Given a collection of n disjoint sets S_1, S_2, ..., S_n, each containing one single element, perform a sequence of operations of the following two types:

Union(A,B,C): Combine the two disjoint sets A and B into a new set named C.

Find(x): Compute the name of the (unique) set that contains x.

The sequence of operations is given *on-line*, i.e., the next operation becomes available if the current one has been processed.

2.1 An optimal amortized solution

The data structure consists of a collection of trees. For each set A in the current collection of sets, there is a tree with $|A|$ nodes. Each node in this tree stores one element of A. Moreover, except for the root, each node contains a pointer to its parent. With the root, we store the name of the set and its size. See Figure 2.1.

Initialization: At the start of the sequence of operations, there are n trees. For $1 \leq i \leq n$, the i-th tree consists of one node that stores the element of S_i, the name of this set and its size.

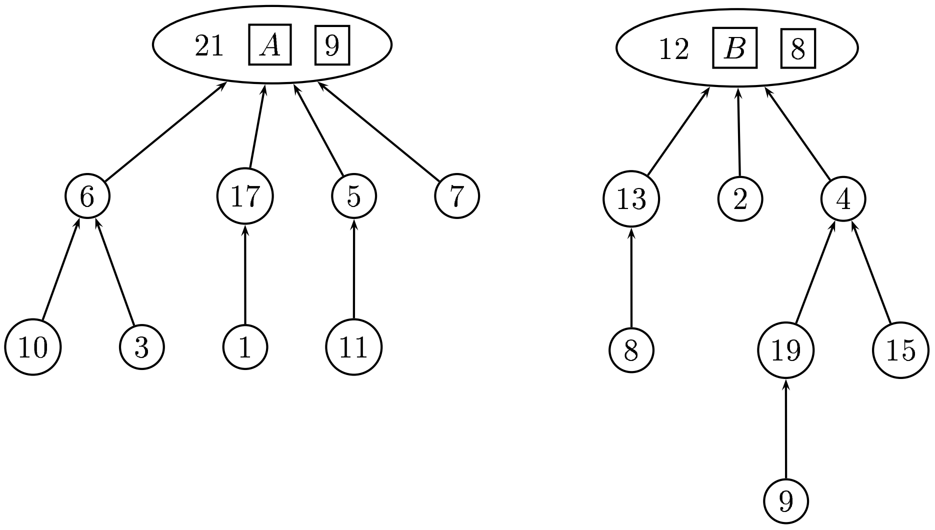

Figure 2.1: *Trees for the sets* $A = \{1, 3, 5, 6, 7, 10, 11, 17, 21\}$ *and* $B = \{2, 4, 8, 9, 12, 13, 15, 19\}$.

Now we can start with the operations:

Union: To process the operation Union(A, B, C), we are given pointers to the roots r_A and r_B of the trees that represent A and B, respectively. In these roots, we read the sizes of A and B.

1. If $|A| \leq |B|$, then we merge both trees by making r_A a child of r_B: We give r_A a pointer to r_B, and with r_B we store the name C of the new set and its size, which is $|A| + |B|$.

2. If $|B| < |A|$, then we merge both trees by making r_B a child of r_A: We give r_B a pointer to r_A, and with r_A we store the name C of the new set and its size, which is $|A| + |B|$.

See Figure 2.2 for an example.

Find: To process the operation Find(x), we are given a pointer to the node u containing x. We perform the following two steps.

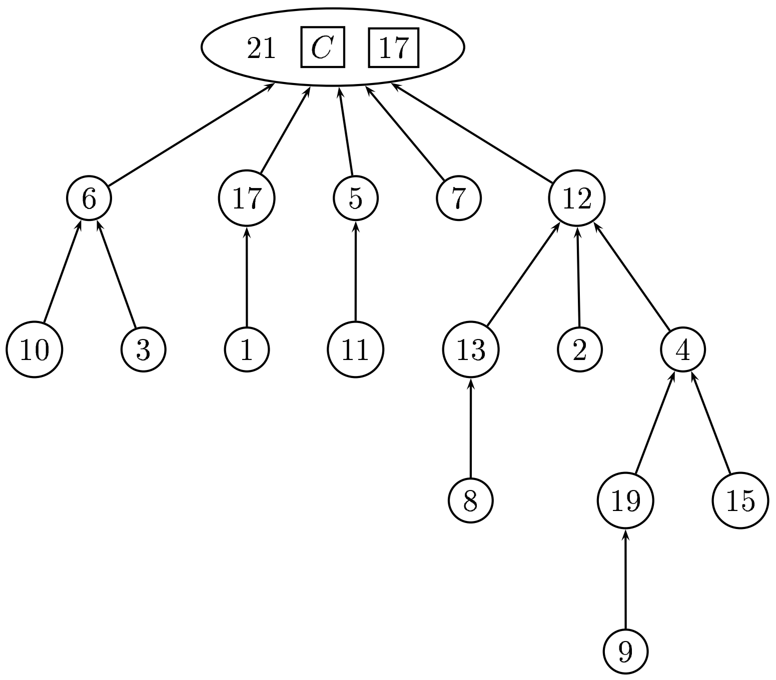

Figure 2.2: *The result of Union(A, B, C) on the two trees of Figure 2.1.*

---

1. Starting in node u, we follow parent-pointers until we reach the root r of u's tree. In r, we read the name of the set that contains x.

2. We again traverse the path from u to r. Each node $v \neq r$ on this path becomes a child of r: We give v a new parent-pointer to r.

See Figure 2.3. The process of Step 2 is called *path-compression.* Subsequent Find-operations may profit from this.

In the next sections, we will analyze the complexity of these algorithms. We will show that any sequence of m Union- and Find-operations can be processed in an amount of time that is *almost* linear in $m+n$. It turns out that this is optimal in the pointer machine model.

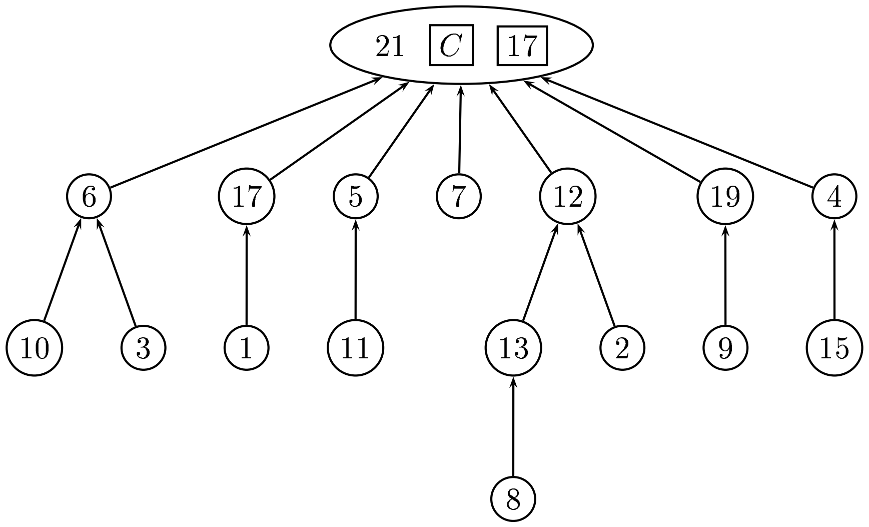

Figure 2.3: *The result of Find(19) on the tree of Figure 2.2.*

2.2 Ackermann's function and its inverse

To analyze the algorithms of the previous section, we need to introduce an extremely slowly growing function. This function is the inverse of an extremely rapidly growing function, which we define first.

We will use the following notation. If f is a function and i is a non-negative integer, then $f^{(i)}$ denotes the i-th iterate of f. That is, $f^{(0)}$ is the identity function and for $i \geq 0$, $f^{(i+1)}$ is defined by $f^{(i+1)}(x) = f(f^{(i)}(x))$ for all x.

For $k \geq 0$, we define the function $A_k : \mathbb{N} \longrightarrow \mathbb{N}$ recursively, as follows:

1. For all $x \in \mathbb{N}$, $A_0(x) := x + 1$.

2. For $k \geq 0$ and $x \in \mathbb{N}$, $A_{k+1}(x) := A_k^{(x)}(x)$.

To get an idea of the behavior of these functions, we consider a few of them. For $x = 0$, we have $A_0(0) = 1$ and $A_{k+1}(0) = A_k^{(0)}(0) = 0$ for

$k \geq 0$. For $x = 1$, we have $A_{k+1}(1) = A_k^{(1)}(1) = A_k(1) = \ldots = A_0(1) = 2$. Let $x \geq 2$. Then $A_0(x) = x + 1$ and

$$\begin{aligned} A_1(x) &= A_0^{(x)}(x) = A_0(A_0^{(x-1)}(x)) = A_0^{(x-1)}(x) + 1 \\ &= A_0(A_0^{(x-2)}(x)) + 1 = A_0^{(x-2)}(x) + 2. \end{aligned}$$

Continuing in this way, we get $A_1(x) = 2x$. For $k = 2$, we get

$$\begin{aligned} A_2(x) &= A_1^{(x)}(x) = A_1(A_1^{(x-1)}(x)) = 2 \cdot A_1^{(x-1)}(x) \\ &= 2 \cdot A_1(A_1^{(x-2)}(x)) = 2^2 \cdot A_1^{(x-2)}(x) \\ &= \ldots \ = 2^x \cdot x \geq 2^x. \end{aligned}$$

Next we consider A_3.

$$A_3(x) = A_2^{(x)}(x) = A_2(A_2^{(x-1)}(x)) \geq 2^{A_2^{(x-1)}(x)},$$

which implies

$$A_3(x) \geq \underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^{2^x}}}}}}}_{x\ 2's}.$$

The function A_4 grows so fast that we only consider $A_4(2)$:

$$A_4(2) = A_3^{(2)}(2) = A_3(A_3(2)).$$

Since

$$A_3(2) = A_2^{(2)}(2) = A_2(A_2(2)) = A_2(8) = 2048,$$

we get

$$A_4(2) = A_3(2048) \geq \underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}}_{2049}.$$

Exercise 2.2.1 Prove that for all $k \geq 0$ and $y \geq x \geq 1$,
(1) $A_k(x) \geq x$,
(2) $A_k(y) \geq A_k(x)$.

Now we can define our extremely rapidly growing function $A : \mathbb{N} \longrightarrow \mathbb{N}$:

$$A(k) := A_k(2) \text{ for } k \geq 0.$$

This function is called Ackermann's function. The function we are actually interested in is its inverse $\alpha : \mathbb{N} \longrightarrow \mathbb{N}$, defined by

$$\alpha(n) := \min\{k \geq 0 : A(k) \geq n\}.$$

We claim that for all practical applications, $\alpha(n)$ is at most 4. Indeed, let n be such that $\alpha(n) \geq 5$. Then $A(k) < n$ for $0 \leq k \leq 4$. In particular, $n > A(4)$. We have seen, however, that $A(4) = A_4(2)$ is already a number beyond comprehension.

Exercise 2.2.2 Prove that α is well-defined. That is, prove that for each $n \geq 0$, there is a $k \geq 0$ such that $A(k) \geq n$. Also, prove that $\alpha(n) \to \infty$ for $n \to \infty$.

Remark 2.2.1 In the literature, several different definitions of Ackermann's function appear. All these functions grow at roughly the same rate.

At first sight, the function α looks rather artificial. In the next section, we will prove that any sequence of m Union- and Find-operations, processed as in Section 2.1, takes $O((m+n)\alpha(n))$ time. It has been shown by Tarjan and La Poutré that this is optimal on a pointer machine. (This is a machine model having no random access.)

In fact, the function α appears in the analysis of many combinatorial an computational problems. As an example, consider a set of n line segments in the plane. See Figure 2.4. These segments induce a partition of the plane into maximally connected regions, so-called faces. The complexity of a face is defined as the number of its edges. Note that one segment can contribute several edges to one face. The maximal complexity of such a face is bounded by $O(n \cdot \alpha(n))$. Moreover, there is a constant c that for each n there are n line segments whose induced partition contains a face of complexity at least $c \cdot n \cdot \alpha(n)$. This shows that the function α appears in nature. (Well, in Euclidean nature.)

Exercise 2.2.3 Consider the partition of the plane induced by n lines. Prove that each face in this partition has complexity $O(n)$.

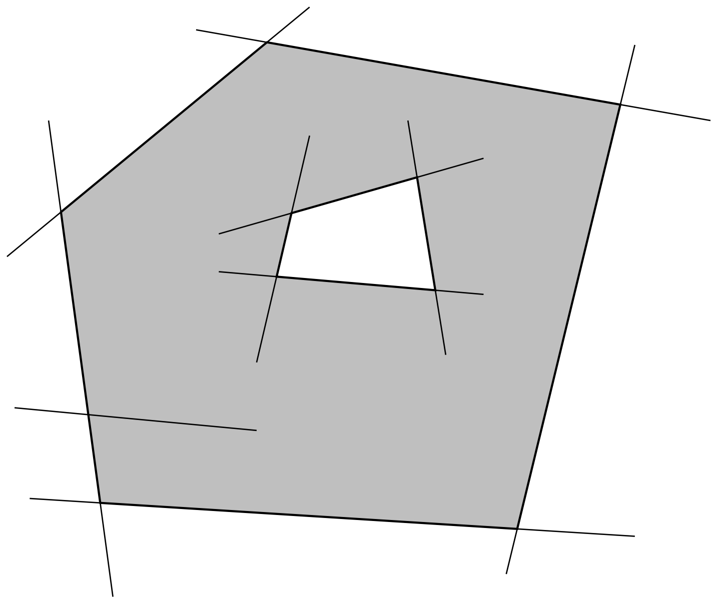

Figure 2.4: *A face of complexity 19.*

2.3 Analysis of the Union-Find algorithm

As mentioned in the previous section, we will show that our algorithms of Section 2.1 process any sequence of m Union- and Find-operations in $O((m+n)\alpha(n))$ time. In order to prove this, we need to introduce the notion of rank.

2.3.1 The rank of a node

Let σ be any sequence of m Union- and Find-operations. Assume that we process this sequence in two ways. Once, we use the algorithms of Section 2.1. In the other way, we also use these algoritms, but without

Step 2 of the Find-algorithm. We say that we process σ with and without path compression, respectively.

Observation 2.3.1 *At any moment during the processing of σ,*

1. *the contents of the trees are the same with or without path compression,*
2. *the roots of the trees are the same with or without path compression,*
3. *a node u becomes a descendant of v with path compression if and only if it does without path compression. With path compression, however, u may at some later point become a non-descendant of v.*

Let $T_t(u)$ denote the subtree rooted a u at time t in the processing of σ *without* path compression. We define the *rank* of a node u as

$$\operatorname{rank}(u) := 2 + \operatorname{height}(T_m(u)),$$

where $\operatorname{height}(T)$ denotes the height or length of a longest path in T.

Observation 2.3.2 *Without path compression,*

1. *as long as node u does not have a parent, the height of $T_t(u)$ can still increase,*
2. *once u becomes a child of another node, the tree rooted at u becomes fixed,*
3. *the height of a tree can never decrease.*

Lemma 2.3.1 *With or without path compression, if u ever becomes a descendant of v, then*

$$\operatorname{rank}(u) < \operatorname{rank}(v).$$

Proof: We know from Observation 2.3.1 that u becomes a descendant of v with path compression if and only if it does without path compression.

Assume that u becomes a descendant of v at time t. It is clear that $\text{height}(T_t(u)) < \text{height}(T_t(v))$. (Recall that $T_t(\cdot)$ is a tree during the processing of σ without path compression.) By Observation 2.3.2, we have $\text{height}(T_s(u)) = \text{height}(T_t(u))$ for all $s \geq t$. Moreover, by Observation 2.3.2 $\text{height}(T_s(v)) \geq \text{height}(T_t(v))$ for all $s \geq t$. It follows that $\text{height}(T_s(u)) < \text{height}(T_s(v))$ for all $s \geq t$. The definition of rank immediately implies that $\text{rank}(u) < \text{rank}(v)$. ∎

Lemma 2.3.2 *For all nodes u and any time t,*

$$|T_t(u)| \geq 2^{\text{height}(T_t(u))}.$$

Proof: The proof is by induction on t. For $t = 0$, the claim is true, because $|T_0(u)| = |\{u\}| = 1$ and $\text{height}(T_0(u)) = 0$. Let $t \geq 0$ and assume the claim holds at time t. Consider the $(t+1)$-st operation of σ. Let u be any node.

If $\text{height}(T_{t+1}(u)) = \text{height}(T_t(u))$, then the claim is true at time $t+1$, because $|T_t(u)| \leq |T_{t+1}(u)|$.

Otherwise, we have $\text{height}(T_{t+1}(u)) > \text{height}(T_t(u))$. Then, the $(t+1)$-st operation must be a Union-operation, and during this operation, a tree $T_t(v)$ is merged into $T_t(u)$, making v a child of u in $T_{t+1}(u)$. Then $\text{height}(T_t(v)) = \text{height}(T_{t+1}(v)) = \text{height}(T_{t+1}(u)) - 1$. By the induction hypothesis, we have $|T_t(v)| \geq 2^{\text{height}(T_t(v))}$. Moreover, since we always merge smaller trees into larger ones, $|T_t(u)| \geq |T_t(v)|$. Therefore,

$$\begin{aligned} |T_{t+1}(u)| &= |T_t(u)| + |T_t(v)| \\ &\geq 2|T_t(v)| \\ &\geq 2^{1+\text{height}((T_t(v))} \\ &= 2^{\text{height}(T_{t+1}(u))}. \end{aligned}$$

This completes the proof. ∎

Lemma 2.3.3 *For any node u,* $\text{rank}(u) \leq \lfloor \text{log} n \rfloor + 2$.

Proof: Since $|T_m(u)| \leq n$, we get from Lemma 2.3.2,

$$n \geq |T_m(u)| \geq 2^{\text{height}(T_m(u))} = 2^{\text{rank}(u)-2}.$$

Hence, $\text{rank}(u) - 2 \leq \log n$. Since $\text{rank}(u)$ is an integer, it follows that $\text{rank}(u) - 2 \leq \lfloor \log n \rfloor$. ■

Lemma 2.3.4 *For any interger $r \geq 2$,*

$$|\{u : \text{rank}(u) = r\}| \leq n/2^{r-2}.$$

Proof: If u and v are nodes such that $\text{rank}(u) = \text{rank}(v)$, then by Lemma 2.3.1, $T_m(u)$ and $T_m(v)$ are disjoint. Moreover, by Lemma 2.3.2, if $\text{rank}(u) = r$, then $|T_m(u)) \geq 2^{r-2}$. Therefore,

$$\begin{aligned} n &\geq \sum_{u:\text{rank}(u)=r} |T_m(u)| \\ &\geq \sum_{u:\text{rank}(u)=r} 2^{r-2} \\ &= |\{u : \text{rank}(u) = r\}| \cdot 2^{r-2}. \end{aligned}$$

■

Now we define a function δ which maps non-root nodes u to integers $\delta(u)$. We process the sequence σ with path compression. Recall that $\text{rank}(u)$ is an integer that is independent of time. However, the parent $\text{parent}(u)$ of u can change with time, and so can $\text{rank}(\text{parent}(u))$.

Exercise 2.3.1 Prove that the value $\text{rank}(\text{parent}(u))$ can only increase. (Hint: use Lemma 2.3.1.)

The function δ is defined as follows:

$$\delta(u) := \max\{k \geq 0 : \text{rank}(\text{parent}(u)) \geq A_k(\text{rank}(u))\}.$$

Note that the value $\delta(u)$ depends on time.

Lemma 2.3.5 *Let $n \geq 5$. For any non-root node u,*

1. *$\delta(u)$ is well-defined,*

2. *$\delta(u)$ can never decrease with time,*

3. *$0 \leq \delta(u) \leq \alpha(n) - 1$.*

Proof: We know from Lemma 2.3.1 that

$$\text{rank}(\text{parent}(u)) \geq \text{rank}(u) + 1 = A_0(\text{rank}(u)).$$

Hence the set $\{k \geq 0 : \text{rank}(\text{parent}(u)) \geq A_k(\text{rank}(u))\}$ is non-empty.

Let $k \geq \alpha(n)$ and assume that $\text{rank}(\text{parent}(u)) \geq A_k(\text{rank}(u))$. By Lemma 2.3.3, we have $\text{rank}(\text{parent}(u)) \leq \lfloor \log n \rfloor + 2$. Hence

$$n > \lfloor \log n \rfloor + 2 \geq \text{rank}(\text{parent}(u)) \geq A_k(\text{rank}(u)).$$

Since $\text{rank}(u) \geq 2$ and since the function A_k is non-decreasing (see Exercise 2.2.1), we get

$$n > A_k(2) = A(k).$$

But, since Ackermann's function is also non-decreasing, we have

$$A(0) \leq A(1) \leq A(2) \leq \ldots \leq A(k) < n.$$

Then the definition of the function α implies that $\alpha(n) > k$. This is a contradiction.

We have proved that $\text{rank}(\text{parent}(u)) < A_k(\text{rank}(u))$ for all $k \geq \alpha(n)$. As a result $\delta(u)$ is well-defined and satisfies $0 \leq \delta(u) \leq \alpha(n) - 1$. Exercise 2.3.1 implies that $\delta(u)$ can only increase. ∎

2.3.2 The analysis

We now prove the upper bound on the total time to process the sequence σ of m Union- and Find-operations with path compression.

Each Union-operation takes $O(1)$ time. Hence, the total time for all Union-operations is bounded by $O(m)$.

It remains to consider the Find-operations. The operation Find(a) takes time proportional to the length of the path from the node u containing a to the root v of u's tree. This path is traversed twice, once to find v and once for the path compression. We spend constant time, say one time unit, per node along this path.

Let x be any node on the path from u to v.

1. If x has an ancestor y such that $\delta(y) = \delta(x)$, then we charge the time unit of x to node x itself.

2. If x does not have such an ancestor, then we charge the time unit of x to the Find-operation.

The total time for the Find-operations is proportional to the total number of time units that are charged by us. We count the time units that are charged to nodes and to Find-operations separately.

First consider an operation Find(a). How many time units are charged to this operation? Let u be the node containing a and let v be the root of u's tree. Let x be any node on the path from u to v and assume we charge the time unit of x to this Find-operation. Then, for all ancestors y of x, we have $\delta(y) \neq \delta(x)$. Hence, if $\delta(x) = k$, then x is the highest node on the path from u to v whose $\delta(\cdot)$-value is k. That is, there is only one node on this path with $\delta(\cdot)$-value k whose time unit is charged to the current Find-operation. By Lemma 2.3.5, the function δ can take $\alpha(n)$ possible values. Hence, at most $\alpha(n)$ time units are charged to this operation.

This proves that we charge at most $m \cdot \alpha(n)$ time units to all Find-operations.

Now consider a node x. How many time units are charged to x over the entire computation? If we charge one time unit to x at time t, then x must have an ancestor y such that $\delta(y) = \delta(x)$. Let $k = \delta(x)$. Then at time t,

$$\text{rank}(\text{parent}(x)) \geq A_k(\text{rank}(x)),$$

and

$$\text{rank}(\text{parent}(y)) \geq A_k(\text{rank}(y)).$$

Suppose that $i \geq 1$ and

$$\text{rank}(\text{parent}(x)) \geq A_k^{(i)}(\text{rank}(x)).$$

Let v be the root of x's tree at time t. Note that $y \neq v$. We know from Lemma 2.3.1 that $\text{rank}(v) \geq \text{rank}(\text{parent}(y))$ and $\text{rank}(y) \geq \text{rank}(\text{parent}(x))$. Recall that the function A_k is non-decreasing. (See Exercise 2.2.1.) It follows that at time t,

$$\text{rank}(v) \quad \geq \quad \text{rank}(\text{parent}(y))$$

$$\begin{aligned}
&\geq A_k(\text{rank}(y)) \\
&\geq A_k(\text{rank}(\text{parent}(x))) \\
&\geq A_k(A_k^{(i)}(\text{rank}(x))) \\
&= A_k^{(i+1)}(\text{rank}(x)).
\end{aligned}$$

Since v is the parent of x at time $t+1$, we have at time $t+1$,

$$\text{rank}(\text{parent}(x)) \geq A_k^{(i+1)}(\text{rank}(x)).$$

This shows that if we charge a time unit to x for the i-th time, then

$$\text{rank}(\text{parent}(x)) \geq A_k^{(i)}(\text{rank}(x))$$

at that moment. (Exercise 2.3.1 is needed to prove this statement.) Therefore, after we have charged rank(x) time units to x, we have

$$\text{rank}(\text{parent}(x)) \geq A_k^{(\text{rank}(x))}(\text{rank}(x)) = A_{k+1}(\text{rank}(x))$$

and

$$\delta(x) \geq k+1$$

at that moment. That is, after rank(x) charges against x, the value of $\delta(x)$ increases by at least one. Since $\delta(x)$ can increase only $\alpha(n)-1$ times (it never decreases!), there can be at most rank$(x) \cdot \alpha(n)$ time units charged to x.

Now we are almost done. By Lemma 2.3.4, there are at most $n/2^{r-2}$ nodes of rank r. Hence, there are at most

$$r \cdot \alpha(n) \cdot \frac{n}{2^{r-2}} = n \cdot \alpha(n) \cdot \frac{r}{2^{r-2}}$$

time units charged to nodes of rank r. Summing over all values of r, we obtain the following upper bound on the total number of time units that are charged to nodes:

$$\sum_{r=2}^{\infty} n \cdot \alpha(n) \cdot \frac{r}{2^{r-2}} = n \cdot \alpha(n) \sum_{i=0}^{\infty} \frac{i+2}{2^i} = 6n \cdot n \cdot \alpha(n).$$

To summarize, we have shown that all Find-operations together take $O((m+n)\alpha(n))$ time.

Theorem 2.3.1 *The algorithms of Section 2.1 process any sequence of m Union- and Find-operations, starting with n singleton sets, in $O((m+n)\alpha(n))$ time.*

2.4 The single-operation complexity

Until now, we analyzed the complexity of an entire sequence of Union- and Find-operations. That is, we were interested in the total running time for processing the sequence. What about the single-operation complexity?

Exercise 2.4.1 (1) Prove that the algorithms of Section 2.1 process each Union-operation in $O(1)$ time and each Find-operation in $O(\log n)$ time.
(2) Prove that the single-operation complexity of the algorithms of Section 2.1 is $\Omega(\log n)$. That is, give a sequence of Union- and Find-operations such that at least one of them takes $\Omega(\log n)$ time.

This exercise shows that the algorithms of Section 2.1 have a single-operation complexity of $\Theta(\log n)$. In Übungsaufgabe 5.6, a family of data structures is analyzed: For each $2 \leq k \leq n$, there is a data structure for the Union-Find problem that processes each Union-operation and each Find-operation in $O(k)$ and $O(\log n / \log k)$ time, respectively. Taking $k = \lceil \log n / \log\log n \rceil$, we get a single-operation complexity of $O(\log n / \log\log n)$. In this section, we prove that we cannot do better: Any algorithm (from a broad class) for the Union-Find problem has $\Omega(\log n / \log\log n)$ single-operation complexity.

First we define the class of algorithms for which the lower bound holds. Any algorithm in this class uses linked data structures that can be viewed as graphs. We assume that these graphs are undirected. Hence, edges can be traversed in both directions. (A lower bound for undirected graphs implies the same lower bound for directed graphs. In an implementation, we have directed graphs, because edges are implemented as pointers.) The algorithm and its data structure should satisfy the following constraints:

1. To each element and to each set in the current partition, exactly one vertex is associated that contains this element and the name of this set, respectively.

2. The data structure consists of graphs, such that each graph corresponds to exactly one set in the current partition. Each such graph does not contain any edges to vertices outside the graph.

3. To process the operation Find(x), the algorithm obtains the vertex containing x. Starting in this vertex, the algorithm follows paths until it reaches the vertex that stores the name of the current set containing x.

4. To process a Union- and Find-operation, the algorithm may insert or delete any edges, as long as Constraint 2 is satisfied.

Exercise 2.4.2 Convince yourself that the algorithms of Section 2.1 and Übungsaufgabe 5.6 can be implemented within this class.

In the rest of this section, we will prove the following result.

Theorem 2.4.1 *Let $\mathcal{A}$ be any algorithm from the given class and let k be a positive integer. If $\mathcal{A}$ processes each Union-operation in $O(k)$ time, then the single-operation complexity of a Find-operation is*

$$\Omega\left(\frac{\log n}{\log k + \log\log n}\right).$$

Corollary 2.4.1 *Any algorithm from the given class has $\Omega(\log n/\log\log n)$ single operation complexity.*

Exercise 2.4.3 Prove Corollary 2.4.1.

How do we prove Theorem 2.4.1? Let $\mathcal{A}$ be any Union-Find algorithm from our class that processes each Union-operation in $O(k)$ time. Let S_1, S_2, ..., S_n be a collection of n disjoint sets, each containing one single element. We will define a sequence of Union-operations on this collection, followed by one Find-operation. This Find-operation will take $\Omega(\log n/(\log k + \log\log n))$ time.

How can we guarantee a lower bound on the time for a Find-operation? Let A be a set of size a in the current partition, and consider its graph G in the data structure. If each vertex of G has degree at most d, then there is an element $x \in A$, such that the shortest path from the vertex containing x to the vertex containing the name of A has length $\Omega(\log a/\log d)$. (We will prove that later.) That is, the operation Find(x) takes $\Omega(\log a/\log d)$ time. Hence, to prove a good lower bound, we have to construct a set A whose values of a and d are "large" and "small", respectively.

Observation 2.4.1 *During each Union-operation, algorithm $\mathcal{A}$ can insert at most $c \cdot k$ edges into the data structure, for some constant c. We assume w.l.o.g. that $c = 1$.*

Now we can define the sequence of Union-operations that "produce" an expensive Find-operation. First, we introduce some terminology. The sequence consists of *stages*:

1. All Union-operations of stage $i-1$ are processed before the first one of stage i.
2. Each Union-operation of stage i combines two disjoint sets of size 2^{i-1} into a set of size 2^i.

If A is a set that is constructed during stage i, then $G_i(A)$ denotes the graph in the data structure corresponding to A immediately after all Union-operations of stage i have been processed.

Initially there are n sets $S_1, S_2, \ldots, S_n$ of size one. For each such set S_j, let $G_0(S_j)$ be the graph corresponding to this set. Note that $G_0(S_j)$ consists of two vertices, one containing the name of S_j and one containing the element of it. We say that $S_1, S_2, \ldots, S_n$ have been constructed during stage 0.

Let $i \geq 1$ and let

$$X_{i-1} := \{A : A \text{ has been constructed during stage } i-1 \text{ and all vertices in } G_{i-1}(A) \text{ have degree at most } 1 + (i-1)k\log n\}.$$

Note that $X_0 = \{S_1, S_2, \ldots, S_n\}$.

Stage i: As long as X_{i-1} contains at least two sets, take and remove two sets A and B from X_{i-1} and process the operation Union(A, B, C).

Lemma 2.4.1 *Let $m = \lceil (\log n)/4 \rceil$. We can perform at least m stages, and $X_m \neq \emptyset$.*

Lemma 2.4.2 *Let G be any graph with a vertices, and let $d \geq 2$. Assume that each vertex of G has degree at most d. Let r be any vertex of G. Then there is a vertex u in G such that any path from u to r has length at least $\lfloor \log a / \log d \rfloor$.*

Using these lemmas (whose proofs are given below), we can prove Theorem 2.4.1: Let $A \in X_m$ and consider the graph $G_m(A)$. Let r be the vertex of this graph that contains the name of A. This graph has at least $n^{1/4}$ vertices, and each of them has degree at most $1+mk\log n$. (By the definition of our class of algorithms, any path that starts in a vertex of $G_m(A)$ lies completely within this graph!) Therefore, by Lemma 2.4.2, $G_m(A)$ contains a vertex u such that any path from u to r has length at least

$$\left\lfloor \frac{\log n^{1/4}}{\log(1+mk\log n)} \right\rfloor = \Omega\left(\frac{\log n}{\log k + \log\log n}\right).$$

If x is the element that is stored in u, then the operation Find(x) takes $\Omega(\log n/(\log k + \log\log n))$ time.

It remains to prove Lemmas 2.4.1 and 2.4.2.

Proof of Lemma 2.4.1: Since $|X_0| = n \geq 2$, we can perform the first stage. Let $i \geq 1$ and assume we have performed the first i stages. (Hence, X_{i-1} contains at least two sets.) Since each set in X_{i-1} has size 2^{i-1}, we have $|X_{i-1}| \leq n/2^{i-1}$. Hence, at most $n/2^i$ Union-operations have been processed during stage i. In particular, at most $kn/2^i$ edges have been inserted into the data structure during this stage.

Consider an operation Union(A,B,C) of stage i. Since $A, B \in X_{i-1}$, each vertex in $G_{i-1}(A)$ and $G_{i-1}(B)$ has degree at most $1+(i-1)k\log n$. Hence, if $C \notin X_i$, at least $k\log n$ edges have been inserted into $G_i(C)$ during stage i. This proves that among all sets that have been constructed during stage i, at most

$$\frac{kn/2^i}{k\log n} = \frac{n}{2^i\log n}$$

do not belong to X_i. Let Y_i be the collection of all sets that have been constructed during stage i. Then $|X_{i-1}| = 2|Y_i|$ or $|X_{i-1}| = 1+2|Y_i|$, depending on whether $|X_{i-1}|$ is even or odd, respectively. Hence,

$$\frac{|X_{i-1}|-1}{2} \leq |Y_i| = |X_i| + |Y_i \setminus X_i| \leq |X_i| + \frac{n}{2^i\log n},$$

which rewrites to

$$|X_i| \geq \frac{1}{2}|X_{i-1}| - \frac{1}{2} - \frac{n}{2^i\log n}.$$

Since $|X_0| = n \geq 2$, we can perform stage 1 and, hence, X_1 exists. In fact,

$$|X_1| \geq \frac{n}{2} - \frac{1}{2} - \frac{n}{2\log n} \geq 2,$$

hence, we can perform stage 2 and obtain a set X_2 of size

$$|X_2| \geq \frac{n}{4} - \frac{1}{4} - \frac{n}{4\log n} - \frac{1}{2} - \frac{n}{4\log n} = \frac{n}{4} - \frac{3}{4} - \frac{2n}{4\log n} \geq 2.$$

Continuing, we can perform stage i provided $|X_{i-1}| \geq 2$. In this case, we obtain a set X_i of size

$$|X_i| \geq \frac{n}{2^i} - \frac{2^i - 1}{2^i} - \frac{in}{2^i \log n}.$$

For $i = m = \lceil (\log n)/4 \rceil$, we get

$$\begin{aligned} |X_m| &\geq \frac{n}{2^{1+(\log n)/4}} - 1 - \frac{1 + (\log n)/4}{2^{(\log n)/4}} \cdot \frac{n}{\log n} \\ &= \frac{1}{4} n^{3/4} - 1 - \frac{n^{3/4}}{\log n}, \end{aligned}$$

which is at least $n^{3/4}/8 \geq 2$ for n sufficiently large. Hence, we can perform m stages. Since X_m contains at least two sets, it is non-empty. ∎

Proof of Lemma 2.4.2: Let $l = \lfloor \log a / \log d \rfloor$. Assume the lemma is false. Then for any vertex u, there is a path from u to r of length less than l. We can as well consider all paths that start in r. Taking paths of length i, we can reach at most d^i vertices of G. Hence, all paths of length less than l that start in r can reach at most

$$\sum_{i=0}^{l-1} d^i = \frac{d^l - 1}{d - 1} < d^l$$

vertices of G. On the other hand, by our assumption, these paths visit all vertices of G. Hence, $d^l > a$, which implies $l > \log a / \log d$. This is a contradiction. ∎

Remark 2.4.1 The lower bound of Theorem 2.4.1 coincides with the upper bound of Übungsaufgabe 5.6 if $k = \Omega((\log n)^{\varepsilon})$ for some $\varepsilon > 0$. For smaller values of k, there is still a gap between the upper and lower bounds. For example, for $k = \log\log n$, the upper bound is $O(\log n / \log\log\log n)$ and the lower bound is $\Omega(\log n / \log\log n)$. Closing this gap is an open problem.

2.5 Further reading

The Union-Find algorithm of Section 2.1 that uses path compression and merging smaller into larger, is due to McIlroy and Morris. They used this algorithm to construct minimum spanning trees. Theorem 2.3.1 was first proved by Tarjan [26]. The proof given here is much simpler than Tarjan's original proof. It appears in Kozen [11].

Lower bounds for the Union-Find problem have been proved by Tarjan [27] and La Poutré [12]. Section 2.4 follows Blum [3]. Übungsaufgabe 5.6 is from Smid [24].

Good references are Tarjan and van Leeuwen [28] and the survey paper by Galil and Italiano [8].

Chapter 3

Range Trees and the Post-Office Problem

This chapter discusses a problem from computational geometry:

Given a set S of n points in $\mathbb{R}^D$ preprocess them into a data structure such that for any query point $p \in \mathbb{R}^D$, we can efficiently find a point $p^* \in S$ that is nearest to p, i.e.,

$$d(p, p^*) = \min\{d(p, q) : q \in S\}.$$

Here, $d(p, q)$ denotes the Euclidean distance between p and q:

$$d(p, q) = \left(\sum_{i=1}^{D} (p_i - q_i)^2 \right)^{1/2}.$$

This problem is known as the *nearest neighbor searching problem* or *the post-office problem*: Think of S as a set of post-offices. Assume you are walking around. Suddenly, you find a letter in your pocket which you want to send. At that moment, you want to know the post-office that is closest to your current position.

In the planar case ($D = 2$), the problem can be solved optimally, i.e., with $O(\log n)$ search time and using $O(n)$ space, by means of Voronoi diagrams and point location. In higher dimensions, however, the problem gets difficult. At this moment, the best results either use a large amount of space (roughly $n^{D/2}$) or have a very high search time (roughly $n^{1-f(D)}$, where $f(D)$ goes to zero for increasing D).

In the dynamic version of the problem, we want to maintain the data structure under insertions and deletions of points. At this moment, it is not known if the dynamic planar post-office problem can be solved with polylogarithmic search time, polylogarithmic update time, using $O(n(\log n)^{O(1)})$ space.

In view of these negative results, it is natural to consider weaker versions of the post-office problem. What happens to the complexity of the problem if we replace the Euclidean metric by a simpler one? Define the L_∞-*distance* between the points p and q by

$$d_\infty(p,q) = \max\{|p_i - q_i| : 1 \leq i \leq D\}.$$

(The Euclidean distance is also called L_2-*distance.*) In the L_∞-*post-office problem*, we want to find a point $p^\infty \in S$ that is closest to the query point p w.r.t. the L_∞-metric, i.e.,

$$d_\infty(p,p^\infty) = \min\{d_\infty(p,q) : q \in S\}.$$

We can also consider the following *approximate* L_2-*post-office problem*: Let $\varepsilon > 0$ be a fixed constant. Instead of searching for the exact Euclidean neighbor p^* of p, we are satisfied with a $(1+\varepsilon)$-*approximate neighbor* of p, i.e., a point $q \in S$ such that

$$d_2(p,q) \leq (1+\varepsilon) \cdot d_2(p,p^*).$$

In this chapter, we will see that both these problems can be solved efficiently, i.e., with polylogarithmic search and update times, using $O(n(\log n)^{O(1)})$ space.

The data structure used is the *range tree*, one of the oldest data structures in computational geometry. Range trees were invented for solving the so-called *orthogonal range searching problem.* (See Übungsaufgaben 5.7 and 5.8.) We show that they can also be used to solve the L_∞-post-office problem.

In the rest of this chapter, we restrict ourselves to the planar case. For the generalization to higher dimensions, the reader is referred to the literature.

3.1 From the exact L_∞-problem to the approximate L_2-problem

Let S be a set of n points in the plane. In this section, we show that any solution for the L_∞-post-office problem can be transformed into a solution for the approximate L_2-post-office problem.

Let $p \in \mathbb{R}^2$ be a query point, and let p^* and p^∞ be the Euclidean neighbor and L_∞-neighbor of p, respectively. As a first try let us take p^∞ as an approximate L_2-neighbor of p. How large is the error? That is, what is the largest value the quotient $d_2(p, p^\infty)/d_2(p, p^*)$ can take?

Exercise 3.1.1 Prove that in the L_∞-metric, a circle with radius one centered at p is an axes-parallel square with sides of length two centered at p.

We can visualize the process of finding p^* and p^∞ as follows. To find p^*, we grow a circle centered at p until its boundary hits at a point of S. This point is the Euclidean neighbor p^* of p. Similarly, to find p^∞, we grow an L_∞-circle, which is an axes-parallel square, centered at p until its boundary hits at a point. This point is the L_∞-neighbor p^∞ of p.

This observation allows us to bound the quotient $d_2(p, p^\infty)/d_2(p, p^*)$. Let $\delta = d_\infty(p, p^*)$ and consider the axes-parallel square C with sides of length 2δ centered at p. See Figure 3.1. Note that p^* lies on the boundary of C. Since p^∞ is the L_∞-neighbor of p, it must lie inside or on the boundary of C. Hence, $d_2(p, p^\infty) \leq \sqrt{2} \cdot \delta$, and equality occurs if and only if p^∞ coincides with one of the corners of C. Similarly, $d_2(p, p^*) \geq \delta$, and equality occurs if and only if p^* coincides with the midpoint of one of the four sides of C. It follows that

$$d_2(p, p^\infty) \leq \sqrt{2} \cdot \delta \leq \sqrt{2} \cdot d_2(p, p^*).$$

That is, p^∞ is a $\sqrt{2}$-approximate L_2-neighbor of p. The error is maximal if and only if p^* is the midpoint of a side and p^∞ is a corner of C.

Now consider the circle with center p and radius $d_2(p, p^*)$. The point p^∞ lies outside or on the boundary of this circle. Hence, p^∞ must lie in the shaded region of Figure 3.1.

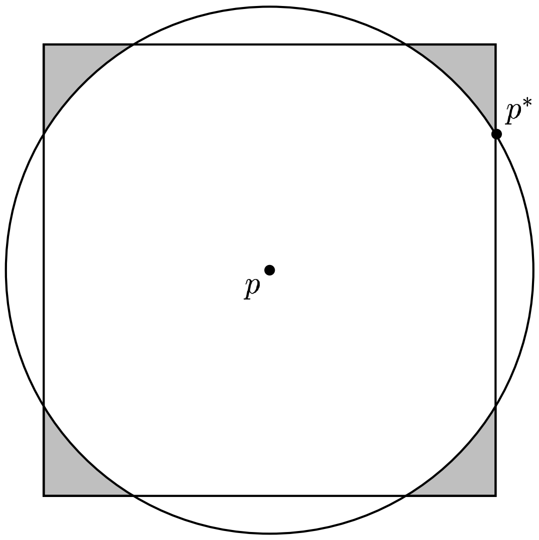

Figure 3.1: *p^* lies on the boundary of the square centered at p having sides of length 2δ, where $\delta = d_\infty(p, p^*)$. The circle is centered at p and has radius $d_2(p, p^*)$. The L_∞-neighbor p^∞ lies in the shaded region.*

We see from Figure 3.1 that the upper bound on the error depends on the angle between the line segment pp^* and the X-axis: The error can be maximal if and only if this angle is zero. On the other hand, if the angle is close to $\pi/4$, the shaded region in Figure 3.1 is very small, and the quotient $d_2(p, p^\infty)/d_2(p, p^*)$ is close to one. In fact, if the angle is exactly $\pi/4$, then p^* and p^∞ are equal.

Exercise 3.1.2 Let α be the angle between the segment pp^* and the positive X-axis. Assume that $0 \leq \alpha \leq \pi/2$. Prove that: $d_2(p, p^\infty) \leq \sqrt{2} d_2(p, p^*) \cos\alpha$ if $\alpha \leq \pi/4$, and $d_2(p, p^\infty) \leq \sqrt{2} d_2(p, p^*) \sin\alpha$ if $\alpha \geq \pi/4$.

Here is the conclusion: The L_∞-neighbor p^∞ is a good approximation for the L_2-neighbor p^* if the angle between pp^* and the positive X-axis is close to $\pi/4$. Of course, in general, this angle will not be close to $\pi/4$. Nevertheless, we can use this approach to find a $(1 + \varepsilon)$-approximate L_2-neighbor of p. The idea is to maintain a number of different coordinate systems such that there is always at least one system in which the angle between pp^* and its X-axis is close to $\pi/4$.

The details are as follows. We assume that the (XY)-coordinate system is given. Let $0 < \varepsilon \leq \pi/4$. For $0 \leq i < 2\pi/\varepsilon$, let X_i and Y_i be the directed lines that make angles of $i \cdot \varepsilon$ with the positive X- and Y-axis, respectively. Consider the (X_iY_i)-coordinate systems, $0 \leq i < 2\pi/\varepsilon$.

Lemma 3.1.1 *For each point q in the plane, there is an index i, such that the angle between the line segment from the origin to q and the positive X_i-axis lies in between $\pi/4$ and $\pi/4 + \varepsilon$.*

Proof: Let $\vec{q}$ be the line segment from the origin to q, and let γ be the angle between $\vec{q}$ and the positive X-axis. First assume that $\pi/4 \leq \gamma < 2\pi$. Let $i = \lfloor(\gamma - \pi/4)/\varepsilon\rfloor$ and let γ_i be the angle between $\vec{q}$ and the positive X_i-axis. Then $\gamma_i = \gamma - i \cdot \varepsilon$ and $\pi/4 \leq \gamma_i \leq \pi/4 + \varepsilon$.

If $0 \leq \gamma < \pi/4$, then we can take $i = \lfloor(7\pi/4 + \gamma)/\varepsilon\rfloor$. In this case, $\gamma_i = 2\pi - i \cdot \varepsilon + \gamma$ and again $\pi/4 \leq \gamma_i \leq \pi/4 + \varepsilon$. ■

For $0 \leq i < 2\pi/\varepsilon$, let S_i denote the set of points in S with coordinates in the (X_iY_i)-coordinate system. Let p be a query point and let $q^{(i)}$ be an L_∞-neighbor of p in S_i, $0 \leq i < 2\pi/\varepsilon$. Let q be the L_∞-neighbor having minimal L_2-distance to p.

Lemma 3.1.2 *q is a $(1 + \varepsilon)$-approximate L_2-neighbor of p.*

Proof: First note that the L_∞-distance depends on the coordinate system. Each (X_iY_i)-system has its own L_∞-distance function. The L_2-distance, however, is the same in all these systems. Let p^* be the exact L_2-neighbor of p. We have to show that $d_2(p, q) \leq (1+\varepsilon) \cdot d_2(p, p^*)$.

By Lemma 3.1.1, there is an index i such that the angle γ_i between the line segment from the origin to the point $p^* - p$ and the positive X_i-axis satisfies $\pi/4 \leq \gamma_i \leq \pi/4 + \varepsilon \leq \pi/2$. Note that γ_i is also the angle between the line segment from p to p^* and the positive X_i-axis. Exercise 3.1.2 implies that

$$\begin{aligned} d_2(p, q^{(i)}) &\leq \sqrt{2} d_2(p, p^*) \sin \gamma_i \\ &\leq \sqrt{2} d_2(p, p^*) \sin(\pi/4 + \varepsilon) \\ &= (\cos\varepsilon + \sin\varepsilon) d_2(p, p^*), \end{aligned}$$

where we used the formula $\sin(\alpha+\beta) = \sin\alpha\cos\beta + \cos\alpha\sin\beta$. Since $0 < \varepsilon \leq \pi/4$, we have $0 < \cos\varepsilon \leq 1$ and $0 < \sin\varepsilon \leq \varepsilon$. Therefore,

$$d_2(p, q^{(i)}) \leq (1+\varepsilon) \cdot d_2(p, p^*).$$

Now consider point q. This point has minimal L_2-distance to p among all L_∞-neighbors $q^{(j)}$, $0 \leq j < 2\pi/\varepsilon$. In particular, $d_2(p,q) \leq d_2(p, q^{(i)})$. This completes the proof. ∎

Remark 3.1.1 We showed that $d_2(p,q)/d_2(p,p^*) \leq 1+\varepsilon$. By a more careful analysis, it can be shown that in fact $d_2(p,q)/d_2(p,p^*) \leq \sqrt{2}\cos(\pi/4 - \varepsilon/2)$. See also Übungsaufgabe 5.9.

We have proved that any solution for the exact L_∞-post-office problem can be used to solve the approximate L_2-post-office problem:

Theorem 3.1.1 *Let $\varepsilon > 0$ be a constant. The complexity of the planar $(1+\varepsilon)$-approximate L_2-post-office problem is at most $O(1/\varepsilon)$ times the complexity of the planar exact L_∞-post-office problem.*

In Section 3.3, we will see how the L_∞-post-office problem can be solved using range trees. This data structure is introduced in the next section.

3.2 Range trees

Range trees are based on balanced binary search trees. We use binary trees as *leaf search trees*: Let V be a subset of $\mathbb{R} \cup \{-\infty, \infty\}$ of size n. We assume that V contains $-\infty$ and ∞. A leaf search tree for V is a binary tree storing the elements of V in its leaves, sorted from left to right. Internal nodes contain information to guide searches. That is, each internal node u contains the values

1. maxl(u), which is the maximal value stored in the left subtree of u, and

2. minr(u), which is the minimal value stored in the right subtree of u.

Exercise 3.2.1 (1) Prove that any leaf search tree for V consists of $2n - 1$ nodes.
(2) Let $x \in \mathbb{R}$. Give an algorithm that finds the smallest element of V that is at least equal to x. Similarly, show how to find the largest element of V that is at most equal to x.

Clearly, the best performance is obtained if the binary tree is *perfectly balanced*, i.e., for each internal node u, the number of leaves in the left and right subtrees of u differ by at most one. It is easy to see that such a tree has height $O(\log n)$.

Exercise 3.2.2 Give two algorithms, one bottom-up and the other top-down, to construct a perfectly balanced leaf search tree for V in $O(n \log n)$ time. If the elements of V are sorted already, the running time should be $O(n)$.

Exercise 3.2.3 Give an exact formula (a function of n) for the height of a perfectly balanced leaf search tree for V.

Now we can define the range tree. Let S be a set of n points in the plane.

Assumption 3.2.1 *All x-coordinates of the points of S are distinct, and the same is true for the y-coordinates of the points of S.*

Hence, no two points of S lie on a horizontal or vertical line. This assumption is made to simplify the algorithm. Later, we shall see how it can be removed.

Definition 3.2.1 A *range tree* for S consists of the following:

1. An *x-tree* (also called *main tree*), which is a perfectly balanced leaf search tree for the x-coordinates of the points of S and the artificial x-coordinates $-\infty$ and ∞.

2. Each node v of this tree contains a pointer to a *y-tree* (also called *associated* or *secondary structure*): Let S_v be the set of points of S whose x-coordinates are stored in the subtree of v. The y-tree of v is a perfectly balanced leaf search tree for the y-coordinates of the points of S_v and the artificial y-coordinates $-\infty$ and ∞.

See Figure 3.2 for a pictorial representation. Note that S_v is a subset of S. In particular, if v is the rightmost leaf of the x-tree, then $S_v = \emptyset$, although v stores the artificial x-coordinate ∞.

Of course, in an implementation, we store with each x- and y-coordinate (a pointer to) the corresponding point of S. Consider a node v of the x-tree. Then we can search in the set S_v for an x-coordinate as well as for a y-coordinate. This makes range trees useful for solving geometric problems.

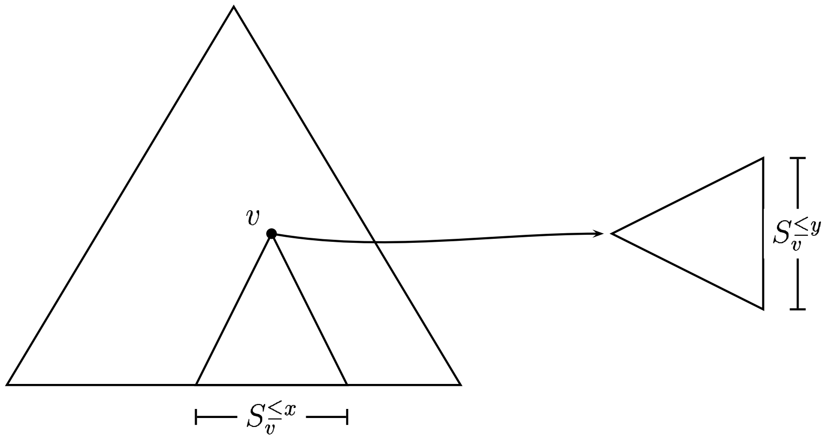

Figure 3.2: *A range tree. The subtree of v stores the x-coordinates of the points of S_v in sorted order its leaves. The y-tree of v stores the y-coordinates of these points, and the values $-\infty$ and ∞, in sorted order in its leaves.*

Let p be a point in the plane. Often we want to search (e.g. with a y-coordinate or a range of y-coordinates) in the set of all points of S that are on or to the right of the vertical line through p. Using the x-tree, we can decompose this set into $O(\log n)$ pairwise disjoint subsets, as follows. Search in the x-tree for the smallest x-coordinate that is at least equal to the x-coordinate of p. During this search, each time we move from a node v to its left son, add the right son of v to an initially empty set M. The leaf in which the search ends is also added to M. See Figures 3.3 and 3.4. In Lemma 3.2.1 below, we will show

```
**procedure** decompose($p$)     (* $p = (p_x, p_y)$ is a point in the plane *)
**begin**
   $M := \emptyset$;
   $v :=$ root of the $x$-tree;
   **while** $v \neq$ leaf
   **do if** maxl($v$) $< p_x$
       **then** $v :=$ right son of $v$
       **else** $M := M \cup \{$right son of $v\}$;
             $v :=$ left son of $v$
       **fi**
   **od**;
   $M := M \cup \{v\}$
**end**
```

Figure 3.3: *Partitioning all points of S that are to the right of p into $O(\log n)$ subsets.*

---

that $\{q \in S : q_x \geq p_x\} = \bigcup_{u \in M} S_u$. Hence, we indeed decomposed the set of all points of S that are to the right of p into $O(\log n)$ subsets. If we want to search in this set, then we can search in each set S_u, $u \in M$, separately.

Lemma 3.2.1 *Consider the set M of nodes of the x-tree that are computed by a call to the procedure* decompose*(p). Then*

$$\{q \in S : q_x \geq p_x\} = \bigcup_{u \in M} S_u,$$

and the right-hand side is a union of pairwise disjoint sets. The set M consists of $O(\log n)$ nodes.

Proof: Let v be the leaf in which the procedure decompose(p) ends, and let r be the point whose x-coordinate is stored in v. (We assume that v is not the rightmost leaf of the x-tree. In that case, the lemma is true.) Then $r_x \geq p_x$. All leaves in the subtree of any node $u \in M \setminus \{v\}$

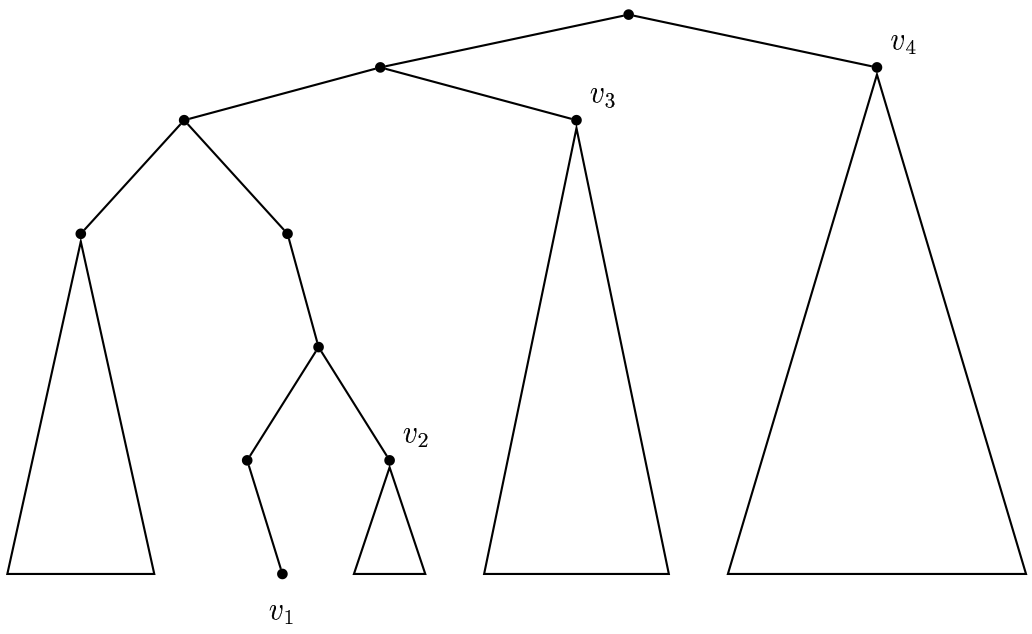

Figure 3.4: *The search for p_x ends in the leaf v_1. We have $M = \{v_1, v_2, v_3, v_4\}$.*

are to the right of v. Hence, the x-coordinates stored in these leaves are at least equal to p_x. This proves that $\bigcup_{u \in M} S_u \subseteq \{q \in S : q_x \geq p_x\}$.

To prove the converse, let q be a point of S such that $q_x \geq p_x$. Let l be the leaf that contains q_x. Then $l = v$, or l is to the right of v. If $l = v$, then $q \in \bigcup_{u \in M} S_u$, because $v \in M$. Assume $l \neq v$. Let w be the lowest common ancestor of v and l. Then v is in the left subtree of w, l is in the right subtree of w, w is on the search path to p_x, and in w this path moves to the left son of w. Therefore, the right son w' of w is contained in M. Since q_x is stored in the subtree of w', we have $q \in \bigcup_{u \in M} S_u$.

Next we prove that the sets S_u, $u \in M$, are pairwise disjoint. Let u and u' be distinct nodes of M. First note that u is not contained in the subtree of u', and u' is not contained in the subtree of u. Let w be the lowest common ancestor of u and u'. Then, u and u' are contained in different subtrees of w. This proves that $S_u \cap S_{u'} = \emptyset$.

Since each node on the search path "delivers" at most one node to the set M, it follows that this set has size $O(\log n)$. ∎

Let us analyze the size of a range tree for a set S of n points. Consider a fixed level of the x-tree, and let $u_1, u_2, \ldots, u_k$ be the nodes on this level. For $1 \leq i \leq k$, the y-tree of u_i has size $O(|S_{u_i}|)$. Note that the sets S_{u_i}, $1 \leq i \leq k$, partition S. Therefore, $\sum_{i=1}^{k} |S_{u_i}| = n$. It follows that the y-trees of the nodes $u_1, u_2, \ldots, u_k$ together have size $O(n)$. This holds for any level of the x-tree. Hence, all y-trees together have size $\Theta(n \log n)$. Since the x-tree itself has size $O(n)$, we have proved:

Lemma 3.2.2 *A range tree for a set of n points in the plane has size $\Theta(n \log n)$.*

To finish this section, we consider the problem of building a range tree. By Lemma 3.2.2, this takes $\Omega(n \log n)$ time. Here is an algorithm that builds the data structure in $O(n \log n)$ time.

1. Build the x-tree.
2. Do the following for each leaf u of the x-tree: Let p be the point whose x-coordinate is stored in u. Give u a pointer to a y-tree storing the set $\{-\infty, p_y, \infty\}$. (The y-trees of the leftmost and rightmost leaves store the sets $\{-\infty, \infty\}$.)

3. Build the y-trees of the internal nodes in a bottom-up fashion: If u is an internal node with sons v and w, such that the y-trees of v and w have been built already, then we copy and merge these y-trees. (The values $-\infty$ and ∞ are stored only once in the resulting tree.) The tree obtained in this way is the y-tree of u.

Exercise 3.2.4 Prove that this algorithm builds a range tree in $O(n \log n)$ time.

We now explain how to remove Assumption 3.2.1. In the x-tree, we store the points using the lexicographical ordering instead of the ordering by x-coordinates. The search information stored in the internal nodes become points instead of x-coordinates.

Similarly, in a y-tree, we store points using the "reversed" lexicographical ordering (a y-coordinate has higher priority than an x-coordinate). The algorithms are only slightly changed. In the procedure decompose(p), we search for the leftmost leaf that stores a point whose x-coordinate is at least equal to p_x.

3.3 Solving the L_∞-post-office problem

Recall the problem we want to solve: Preprocess a set S of n planar points in a data structure, such that for any query point $p \in \mathbb{R}^2$, we can find its L_∞-neighbor, i.e., a point $p^\infty \in S$ such that

$$d_\infty(p, p^\infty) = \min\{d_\infty(p, q) : q \in S\}.$$

We will show that this problem can be solved using range trees.

Consider a query point p. Let p^l and p^r be the L_∞-neighbors of p in the sets $\{q \in S : q_x \leq p_x\}$ and $\{q \in S : q_x \geq p_x\}$, respectively. We call these points the *left-L_∞-neighbor* and *right-L_∞-neighbor* of p, respectively. Clearly, one of them is the L_∞-neighbor of p.

We show how to find the right-L_∞-neighbor p^r of p. (This point may not be unique. Actually, we should talk about *a* right-L_∞-neighbor.) The algorithm consists of three stages. Here is a brief overview.

Stage 1: Call the procedure decompose(p) of the previous section. This procedure computes a set M of nodes such that $\{q \in S :$

$q_x \geq p_x\} = \bigcup_{u \in M} S_u$. Number these nodes v_1, v_2, ..., v_m, where $m = |M|$ and v_i is closer to the root than v_{i-1}, $2 \leq i \leq m$. (See Figure 3.4.)

Stage 2: We know that the right-L_∞-neighbor p^r is contained in the union $\bigcup_{u \in M} S_u$. In the second stage, we want to search for a node $v \in M$ such that S_v contains p^r. This turns out to be difficult. We can, however, reach the following somewhat weaker goal: We compute a node $v \in M$ and a small set $C \subseteq S$ such that $C \cup S_v$ contains p^r.

Stage 3: given node v and set C, we walk down the subtree of v. During this walk, we maintain the invariant that $C \cup S_v$ contains p^r. If v is a leaf, then the set $C \cup S_v$ is small enough to look at all its points and take the one having minimal L_∞-distance to p. This point is the right-L_∞-neighbor p^r of p.

We now discuss Stages 2 and 3 in more detail.

3.3.1 Stage 2

Run the following algorithm.

```
$C := \emptyset$; $i := 1$; *stop* := *false*;
**while** $i \leq m$ **and** *stop* = *false*
**do** search in the $y$-tree of $v_i$ for the largest resp. smallest $y$-coordinate
    that is less than resp. at least equal to $p_y$;
    let $a$ and $b$ be the points that correspond to these $y$-coordinates;
    $r :=$ the point stored in the rightmost leaf of the subtree of $v_i$;
    $\delta := r_x - p_x$;
    $R :=$ the rectangle $[p_x : r_x] \times [p_y - \delta : p_y + \delta]$;
    **if** $a$ and $b$ outside $R$
    **then** $C := C \cup \{a, b\}$;
         $i := i + 1$
    **else** $v := v_i$;
         *stop* := *true*
    **fi**
**od**
```

In words, this algorithm does the following. It visits the nodes of M from left to right (or, equivalently, from bottom to top). Consider one iteration. (See Figure 3.5.) The algorithm searches with p_y in the y-tree of v_i. This gives two points a and b of S_{v_i} between (w.r.t. the vertical direction) which p lies. The vertical lines through p and the point r define a slab whose width is denoted by δ. Note that all points of S_{v_i} lie in or on the boundary of this slab. Consider the rectangle R. If a and b are both outside R, then we add these points to C and go to the next iteration. Otherwise, if a or b is in R or on its boundary, then the while-loop stops.

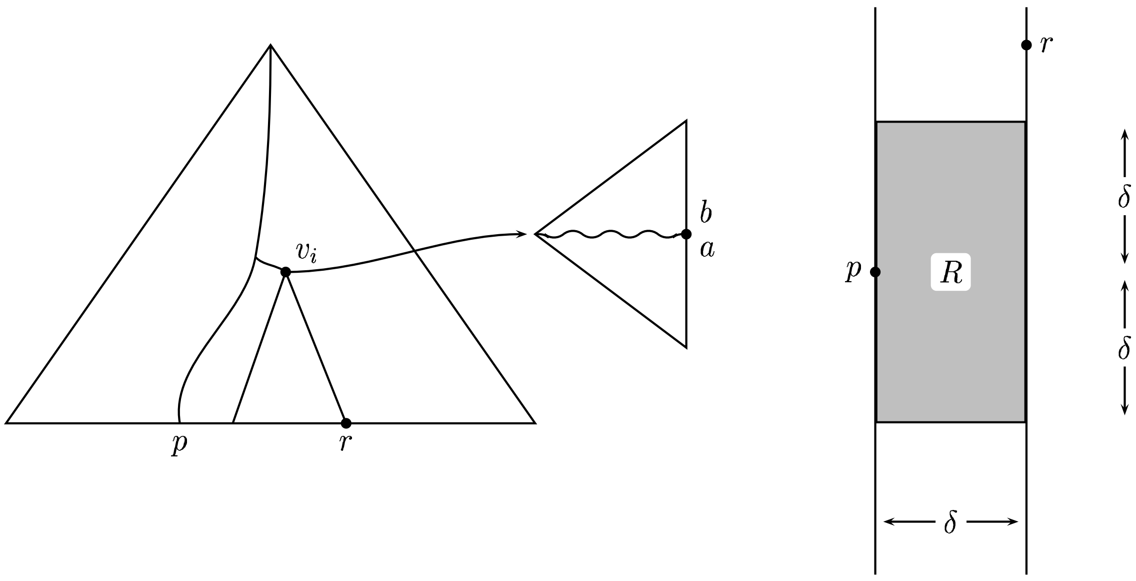

Figure 3.5: *Illustrating one iteration of Stage 2.*

Remark 3.3.1 Since the x- and y-trees also store values $-\infty$ and ∞, we have to be careful. If the rightmost leaf in the subtree of v_i stores the value ∞, then there is no point r corresponding to it. In this case, the value of δ, which is $r_x - p_x = \infty - p_x$ according to the algorithm, is set to ∞. As a result, the rectangle R is the halfplane to the right of the vertical line through p.

Similarly, the y-coordinate b_y may be ∞. Then, there is no point b corresponding to this value. In this case, we use an artificial point b which is outside rectangle R if r_x is finite, and inside R if $r_x = \infty$. A y-coordinate $a_y = -\infty$ is treated in a similar way.

We consider the variable *stop* at the end of the while-loop and distinguish the two cases where this variable has value *true* or *false*.

Lemma 3.3.1 *If the variable stop has value false after the while-loop has been completed, then the set C contains a right-L_∞-neighbor of p.*

Proof: First note that the while-loop makes m iterations. Let p^r be a right-L_∞-neighbor of p, and let i be the index such that $p^r \in S_{v_i}$. Consider the i-th iteration. The points a and b selected during this iteration are outside R.

Let q be any point of S_{v_i}. Then $p_x \leq q_x \leq r_x$ and, hence, $0 \leq q_x - p_x \leq r_x - p_x = \delta$. On the other hand, since q is outside R, we have $|q_y - p_y| > \delta$. It follows that $d_\infty(p, q) = |q_y - p_y|$. That is, for all points of S_{v_i}, the L_∞-distance to p is the same as the distance to p in the y-direction.

Assume w.l.o.g. that $d_\infty(p, a) \leq d_\infty(p, b)$. Then

$$d_\infty(p, a) = |p_y - a_y| \leq |p_y - p^r_y| = d_\infty(p, p^r).$$

On the other hand, since p^r is a right-L_∞-neighbor of p, we have $d_\infty(p, p^r) \leq d_\infty(p, a)$. This proves that $d_\infty(p, a) = d_\infty(p, p^r)$.

Hence, a is also a right-L_∞-neighbor of p. Since a is added to C during the i-th iteration, the proof is completed. ■

If the variable *stop* has value *false* at the end of the while-loop, then we can easily complete the algorithm: We consider all points of C and take the one having minimal L_∞-distance to p. By Lemma 3.3.1, this point is a right-L_∞-neighbor of p.

Lemma 3.3.2 *If the variable stop has value true after the while-loop has been completed, then the set $C \cup S_v$ contains a right-L_∞-neighbor of p.*

Proof: Let p^r be a right-L_∞-neighbor of p, and let i be the index such that $p^r \in S_{v_i}$. Let j be the integer such that during the j-th iteration, the variable *stop* is set to the value *true*. Note that $v = v_j$.

First assume that $i < j$. During the i-th iteration, the points a and b that are selected in the y-tree of v_i are outside the rectangle R. In exactly the same way as in the proof of Lemma 3.3.1, it can be shown that a or b is also a right-L_∞-neighbor of p. Since both points are added to C during the i-th iteration, the claim follows.

Next assume that $i = j$. Then the set S_v, hence also the set $C \cup S_v$, contains a right-L_∞-neighbor of p.

It remains to consider the case where $i > j$. Look what happens during the j-th iteration. Let a and b be the points in the y-tree of v_j that are selected during this iteration. At least one of them is contained in the rectangle R. Assume w.l.o.g. that a lies in R. Then $d_\infty(p, a) \leq \delta$, where δ is the x-distance between p and the rightmost point r in the subtree of v_j.

Since the x-coordinates of all points in S_{v_j} are at most equal to the x-coordinates of the point in S_{v_i}, we have $p^r_x \geq r_x$. This implies that

$$d_\infty(p, p^r) \geq p^r_x - p_x \geq r_x - p_x = \delta.$$

We have shown that $d_\infty(p, a) \leq d_\infty(p, p^r)$. On the other hand, since p^r is a right-L_∞-neighbor of p, we have $d_\infty(p, p^r) \leq d_\infty(p, a)$. This proves that $d_\infty(p, p^r) = d_\infty(p, a)$ and, hence, a is also a right-L_∞-neighbor of p. Since $a \in S_{v_j} = S_v$, the proof of the lemma is completed. ■

This concludes Stage 2. To summarize, if the variable *stop* has value *false* after the while-loop has been completed, then we find a right-L_∞-neighbor of p by looking at all point of C. In this case, the algorithm terminates. Otherwise, we know that the set $C \cup S_v$ contains a right-L_∞-neighbor of p. In this case, we proceed to the next stage.

3.3.2 Stage 3

Run the following algorithm.

```
while $v$ is not a leaf
do $w$ := left son of $v$;
```

```
        search in the $y$-tree of $w$ for the largest resp. smallest $y$-coordinate
        that is less than resp. at least equal to $p_y$;
        let $a$ and $b$ be the points that correspond to these $y$-coordinates;
        $r :=$ the point stored in the rightmost leaf of the subtree of $w$;
        $\delta := r_x - p_x$;
        $R :=$ the rectangle $[p_x : r_x] \times [p_y - \delta : p_y + \delta]$;
        if $a$ and $b$ outside $R$
        then $C := C \cup \{a, b\}$;
             $v :=$ right son of $v$
        else $v := w$
        fi
    od
```

Lemma 3.3.3 *During the while-loop, the set $C \cup S_v$ contains a right-L_∞-neighbor of p.*

Exercise 3.3.1 Prove Lemma 3.3.3.

We complete Stage 3 as follows: By looking at all points of $C \cup S_v$, we take the one having minimal L_∞-distance to p. By Lemma 3.3.3, this point is a right-L_∞-neighbor of p.

This concludes the algorithm for computing a right-L_∞-neighbor p^r of p. In a completely symmetric way, we compute a left-L_∞-neighbor p^l of p. Then, if $d_\infty(p, p^l) \leq d_\infty(p, p^r)$, p^l is an L_∞-neighbor of p. If $d_\infty(p, p^l) > d_\infty(p, p^r)$, then p^r is an L_∞-neighbor of p.

We analyze the running time of the query algorithm. By Lemma 3.2.1, Stage 1 takes $O(\log n)$ time. Consider the while-loop of Stage 2. Each iteration takes $O(\log n)$ time. Since $m = O(\log n)$, there are $O(\log n)$ iterations. Therefore, the entire while-loop takes $O((\log n)^2)$ time. If the variable *stop* has the value *false* after this loop, then we need $O(|C|)$ time to find a right-L_∞-neighbor of p. It is clear that $|C| \leq 2|M|$. Hence, $|C| = O(\log n)$. This proves that Stage 2 takes $O((\log n)^2)$ time.

In the while-loop of Stage 3, we walk down a path in the subtree of v. In each node on this path, we spend $O(\log n)$ time. Since this path has length $O(\log n)$, the entire loop takes $O((\log n)^2)$ time. Afterwards, we need $O(|C \cup S_v|)$ time to find a right-L_∞-neighbor. Since v is a leaf at this moment, we have $|S_v| = 1$. The size of C is bounded by $O(\log n)$. Therefore, this final step takes $O(\log n)$ time.

We have shown that the algorithm finds a right-L_∞-neighbor of p in $O((\log n)^2)$ time. In the same amount of time a left-L_∞-neighbor is found. Given these two points, the L_∞-neighbor of p is obtained in $O(1)$ time.

We summarize our result.

Theorem 3.3.1 *Let S be a set of n points in the plane. Using a range tree, which has size $O(n \log n)$, we can solve the L_∞-post-office-problem with a query time of $O((\log n)^2)$.*

Applying Theorem 3.1.1 gives:

Corollary 3.3.1 *Let $\varepsilon > 0$ be a constant, and let S be a set of n points in the plane. The $(1+\varepsilon)$-approximate L_2-post-office problem can be solved using $O(n \log n)$ space with a query time of $O((\log n)^2)$.*

3.4 Improving the query time: layering

We have seen that a range tree solves the two-dimensional L_∞-post-office problem with a query time of $O((\log n)^2)$. In this section, we reduce the query time to $O(\log n)$.

Consider the query algorithm of the previous section. This algorithm makes $O(\log n)$ binary searches in different y-trees, and it does some additional work. It is easily seen that the additional work takes only $O(\log n)$ time. (Here, we assume that we store with each node of the x-tree a pointer to the rightmost leaf in its subtree.) The $O(\log n)$ binary searches together take $O((\log n)^2)$ time. That is, the running time of the query algorithm is dominated by the time of these binary searches.

How can we improve the running time? The key observation is that in each y-tree, we search for the *same* element: we search for the y-coordinate of the query point p.

Let u and v be nodes of the x-tree such that u is a son of v. Assume we want to locate p_y in the y-trees of u and v. Recall that S_u and

S_v denote the points of S that are stored in the subtrees of u and v, respectively.

Assume that the y-coordinate p_y is less than all y-coordinates of the points of S_v. Then the search for the smallest element in the y-tree of v that is at least equal to p_y will end in the second leftmost leaf of this y-tree. (The leftmost leaf stores the artificial y-coordinate $-\infty$.) Where does the search in the y-tree of u end? Since $S_u \subseteq S_v$, it is clear that p_y is less than all y-coordinates of the points of S_u. Therefore, the search for the smallest element in the y-tree of u that is at least equal to p_y also ends in the second leftmost leaf.

In general, the search for p_y in the y-tree of v gives information about the result of a search for the same element in the y-tree of u. As we will see, we can use this information such that, given the position of p_y in the y-tree of v, only $O(1)$ time is needed to locate p_y in the y-tree of u. That is, we avoid making a binary search in this y-tree. The idea is to link the y-trees of u and v by pointers. This technique is called *layering*.

We change the range tree as follows:

1. As before, we have an x-tree which is a perfectly balanced leaf search tree for the x-coordinates of the points of S and the artificial x-coordinates $-\infty$ and ∞.

2. Each node v of the x-tree contains a pointer to a y-tree, which is a perfectly balanced leaf search tree for the y-coordinates of the points of S_v and the artificial y-coordinates $-\infty$ and ∞.

3. For all nodes u and v of the x-tree, such that u is a son of v, there are pointers from the y-tree of v to the y-tree of u: Let l be any leaf in the y-tree of v, and let q_y be the y-coordinate stored in l. Leaf l stores a pointer to the leftmost leaf in the y-tree of u whose y-coordinate is at least equal to q_y.

We call the resulting data structure a *layered range tree*. See Figure 3.6. Note that if q_y also occurs as a y-coordinate of a point in S_u, then the pointer from l points to the occurrence of q_y in the y-tree of u.

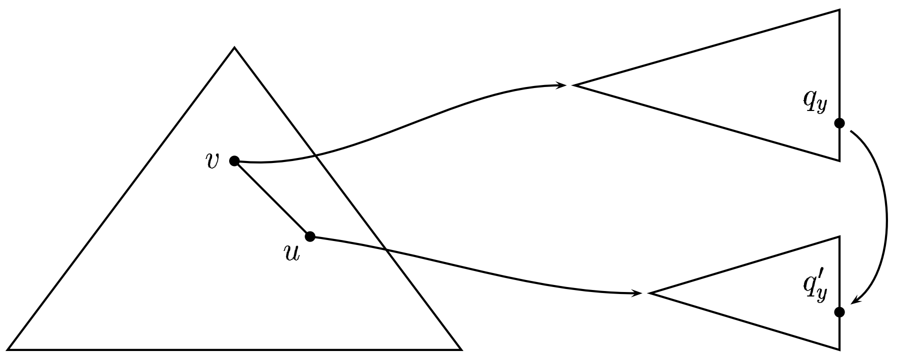

Figure 3.6: *A layered range tree. The leaf storing q_y contains a pointer to the leaf storing q'_y. If $q_y = -\infty$, then $q'_y = -\infty$. If $q_y = \infty$, the $q'_y = \infty$. If q_y is finite, then $q'_y = \min\{s_y : s = (s_x, s_y) \in S_u, s_y \geq q_y\}$.*

Exercise 3.4.1 Prove that a layered range tree still has size $O(n \log n)$ and that it can be built in $O(n \log n)$ time.

Let p be any point in the plane. Call the procedure decompose(p). (See Figure 3.3.) This gives a set M of nodes of the x-tree such that $\{q \in S : q_x \geq p_x\} = \bigcup_{u \in M} S_u$. We show how to search for the smallest elements that are at least equal to p_y, in the y-trees of all nodes $u \in M$.

We again walk down the path in the x-tree to the leftmost leaf whose point has x-coordinate at least p_x. This walk starts in the root v of the x-tree. We locate p_y in the y-tree of v. Let w be the right son of v. Then, following the pointer from the leaf in v's y-tree that stores the position of p_y to the y-tree of w, we have located p_y in w's y-tree. (See Lemma 3.4.1 below.) If w is on the path to p_x, then we proceed in the subtree of w. Otherwise, let u be the left son of v. Note that $w \in M$. We follow the pointer from the leaf in v's y-tree that stores the position of p_y to the y-coordinate of u. This gives the smallest y-coordinate in this y-tree that is at least equal to p_y. Now we proceed in the subtree of u. The complete algorithm is given in Figure 3.7.

```
procedure searchM($p$)
(* $p = (p_x, p_y)$ is a point in the plane *)
begin
   $M := \emptyset$; $v :=$ root of the $x$-tree;
   search in the $y$-tree of $v$ for the smallest $y$-coordinate that is
   at least equal to $p_y$;
   $l :=$ leaf where this search ends;
   $q :=$ point whose $y$-coordinate is stored in $l$;
   while $v \neq$ leaf
   do (* invariant: $l$ is a leaf in the $y$-tree of $v$, $l$ stores $q_y$,
       $q_y = \min\{s_y : s_y \geq p_y$ and $s_y$ stored in the $y$-tree of $v$ $\}$ *)
       $w :=$ right son of $v$;
       follow the pointer from $l$ to the leaf $l'$ in the $y$-tree of $w$;
       $q' :=$ point whose $y$-coordinate is stored in $l'$;
       if maxl($v$) $< p_x$
       then $v := w$; $l := l'$; $q := q'$
       else $M := M \cup \{w\}$; output a pointer to $l'$;
            $u :=$ left son of $v$;
            follow the pointer from $l$ to the leaf $l''$ in the $y$-tree of $u$;
            $q'' :=$ point whose $y$-coordinate is stored in $l''$;
            $v := u$; $l := l''$; $q := q''$
       fi
   od;
   $M := M \cup \{v\}$
   output a pointer to $l$
end
```

Figure 3.7: *Constructing the set M, and locating p_y in the y-tree of all nodes of M.*

Lemma 3.4.1 *During the while-loop of the procedure* searchM(p)*, the invariant is correctly maintained.*

Proof: It is clear that the invariant holds after the initialization. Consider one iteration. That is, let v be a node of the x-tree, let l be a leaf in the y-tree of v, let q be the point whose y-coordinate is stored in l, and assume that

$$q_y = \min\{s_y : s_y \geq p_y \text{ and } s_y \text{ stored in the } y\text{-tree of } v\}.$$

Let w be the right son of v, let l' be the leaf in the y-tree of w that is reached by following the pointer stored with l, and let q' be the point whose y-coordinate is stored in l'.

We will show that

$$q'_y = \min\{s_y : s_y \geq p_y \text{ and } s_y \text{ stored in the } y\text{-tree of } w\}.$$

From the definition of the pointers that link the y-tree of v with that of w, we know that $q'_y \geq q_y$. Since $q_y \geq p_y$, we infer that

$$q'_y \in \{s_y : s_y \geq p_y \text{ and } s_y \text{ stored in the } y\text{-tree of } w\}.$$

It remains to show that q'_y is the minimal element of this set. Assume this is not the case. Then there is a y-coordinate r_y stored in the y-tree of w such that $p_y \leq r_y < q'_y$. Note that $S_w \subseteq S_v$. Therefore, the y-coordinates stored in the y-tree of w form a subset of those stored in the y-tree of v. In particular, r_y is stored in the y-tree of v. Since $r_y \geq p_y$, we infer that $r_y \geq q_y$.

We have shown that $q_y \leq r_y < q'_y$, where r_y is stored in the y-tree of w. But then, the pointer from l in the y-tree of v cannot point to the leaf l' storing q'_y. This is a contradiction.

This shows the q'_y is the smallest value in the y-tree of w that is at least equal to p_y. If the search path proceeds to w, then the invariant still holds after this iteration. Otherwise, if the search path proceeds to the left son u of v, then it follows in the same way that q''_y (see Figure 3.7) is the smallest element in the y-tree of u that is at least equal to p_y. Hence, also in this case, the invariant still holds after this iteration. This completes the proof. ∎

Lemma 3.4.2 *The pointers that are reported by the procedure* searchM*(p) point to the leftmost leaves in the y-trees of all nodes of M, whose y-coordinates are at least equal to p_y.*

Proof: This follows immediately from the previous proof. ∎

We analyze the running time of the procedure searchM(p). The initialization takes $O(\log n)$ time. It is easy to see, that each iteration takes $O(1)$ time. Since there are $O(\log n)$ iterations, the while-loop takes $O(\log n)$ time. This proves that the entire procedure runs in $O(\log n)$ time. That is, by introducing the layered range tree, we reduced the time to locate p_y in the y-trees of all nodes of M, from $O((\log n)^2)$ to $O(\log n)$.

Now we return to the algorithm of Section 3.3 for finding an L_∞-neighbor of a query point. We replace Stage 1 by the procedure searchM(p). Then, Stage 2 can be performed in $O(\log n)$ time. In a similar way, the running time for Stage 3 becomes $O(\log n)$. This proves:

Theorem 3.4.1 *Let S be a set of n points in the plane. Using a layered range tree, which has size $O(n \log n)$, we can solve the L_∞-post-office problem with a query time of $O(\log n)$.*

Corollary 3.4.1 *Let $\varepsilon > 0$ be a constant and let S be a set of n points in the plane. The $(1+\varepsilon)$-approximate L_2-post-office problem can be solved using $O(n \log n)$ space with a query time of $O(\log n)$.*

3.5 Supporting insertions and deletions: partial rebuilding

Until now we only considered the static version of the post-office problem. All binary trees that occurred as substructures of the range tree were perfectly balanced. Of course, if we insert and delete points, the range tree might become unbalanced. In this section, we show how all binary trees that constitute the range tree can be kept in balance if points are inserted and deleted.

Consider a range tree for a set S of n points in the plane. We assume for simplicity that Assumption 3.2.1 holds. To insert or delete a point $p = (p_x, p_y)$, we do the following:

1. Search in the x-tree for the leftmost leaf storing an x-coordinate that is at least equal to p_x. Let w be the leaf in which this search ends.

   (a) Assume we have to insert p and assume w.l.o.g. that $p \notin S$. Let q be the point whose x-coordinate is stored in w. Note that $q_x > p_x$. We give w two new sons. The left son is a range tree for the set $\{p\}$ and the right son is a range tree for the set $\{q\}$. Finally we update the maxl(v) and minr(v) values of the nodes on the search path to w.

   (b) Assume we have to delete p and assume w.l.o.g. that $p \in S$. Let u be the father of w, and let v be the other son of u. Then we replace the subtree of u, which is a range tree for the set S_u, by the subtree of v, which is a range tree for $S_v = S_u \setminus \{p\}$. We also update the search information of the nodes on the search path to u.

2. Consider again the path of Step 1. For each node on this path, we insert or delete the value p_y in its y-tree.

3. We rebalance the data structure.

Exercise 3.5.1 Convince yourself that the structure that results from Steps 1 and 2 is a (not necessarily balanced) range tree for the set $S \cup \{p\}$ resp. $S \setminus \{p\}$.

Clearly, the problem is how to rebalance the range tree. Standard binary trees are often rebalanced by means of rotations. Consider a rotation as in Figure 3.8. Assume we apply this rotation to the x-tree. Then, v gets the y-tree of u. This is simply done by changing one pointer. To obtain the new y-tree of u, however, we have to merge the y-trees that are stored with the roots of A and B. This takes $O(|S_u|)$ time, which is large if u is close to the root of the x-tree.

By taking the binary trees from the class of BB[α]-trees, to be defined below, it can be shown that, nevertheless, this leads to an update algorithm with $O((\log n)^2)$ amortized running time. The proof of this

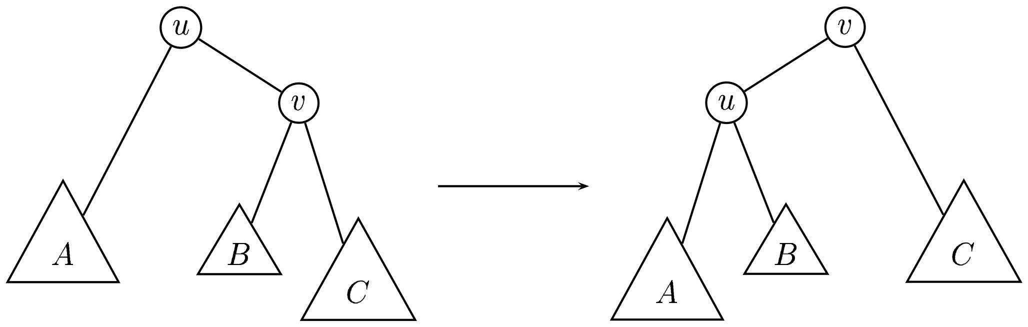

Figure 3.8: *A rotation.*

---

result is complicated. In the rest of this section, we give a much simpler technique, the *partial rebuilding technique*, that gives the same amortized update time.

Definition 3.5.1 Let $0 < \alpha \leq 1/3$ and let T be a binary tree. For each node v of T, let n_v denote the number of leaves in its subtree. The tree T is called a BB[α]-*tree*, if for all nodes u and v such that u is a son of v,

$$\alpha \leq n_u/n_v \leq 1 - \alpha.$$

Hence, if the subtree of v contains m leaves, then each of its subtrees contains at least αm and at most $(1-\alpha)m$ leaves.

Exercise 3.5.2 Prove that for each n there is a BB[α]-tree with n leaves. Why do we require that $\alpha \leq 1/3$? Prove that the height of a BB[α]-tree with n leaves is at most $c_\alpha \cdot \log n$, and determine the constant c_α.

We start with the one-dimensional case. That is, we show how to maintain a BB[α]-tree storing a set S of n real numbers, if elements are inserted and deleted in S. We store with each node v the number n_v of leaves in its subtree. Here is the algorithm to insert or delete a real number p:

1. Search for the leftmost leaf storing a value that is at least equal to p. Let w be the leaf in which this search ends.

2. Insert or delete p and update the appropriate $\text{maxl}(v)$, $\text{minr}(v)$ and n_v values.

3. Rebalance as follows: walk back from w to the root and find the highest node v that is out of balance, i.e., does not satisfy the $\text{BB}[\alpha]$-property. If there is no such v, then the tree is already a $\text{BB}[\alpha]$-tree and we are done. Otherwise, we completely rebuild the subtree of v as a perfectly balanced binary tree.

Exercise 3.5.3 Convince yourself that the rebalancing step results in a $\text{BB}[\alpha]$-tree.

Note that if node v is close to the root, rebalancing will take much time. The following lemma, however, shows that expensive rebalancing operations do not occur often. As we will see, this ensures that the given update algorithm has an amortized running time of $O(\log n)$.

Lemma 3.5.1 *Let v be a node in a* $\text{BB}[\alpha]$-tree *that is in perfect balance. Let n_v be the number of leaves in the subtree of v at the moment when it gets out of balance. Then there have been at least* $(1-2\alpha)n_v - 2$ *updates in the subtree of v.*

Proof: Let n'_v, n'_{lv} and n'_{rv} be the number of leaves in the subtree of v, the left son of v and the right son of v, respectively, at the moment when v is in perfect balance. Assume w.l.o.g./ that $n'_{lv} \leq n'_{rv}$. Then $n'_{lv} = \lfloor n'_v/2 \rfloor$. Clearly, the fastest way for node v to get out of balance is by deleting elements from its left subtree and inserting elements into its right subtree.

Suppose that at the moment when v gets out of balance, N_i insertions have taken place in the right subtree of v, and N_d deletions have taken place in its left subtree. Let n_{lv} be the number of leaves in the subtree of the left son of v at the moment when v gets out of balance. Then $n_v = n'_v + N_i - N_d$ and $n_{lv} = n'_{lv} - N_d = \lfloor n'_v/2 \rfloor - N_d$. Since node

v is out of balance at this moment, we have $n_{lv}/n_v < \alpha$. It follows that

$$\begin{aligned} \alpha n_v &> n_{lv} \\ &= \lfloor n'_v/2 \rfloor - N_d \\ &> n'_v/2 - 1 - N_d \\ &= \frac{n_v - N_i + N_d}{2} - 1 - N_d \\ &= \frac{n_v - (N_i + N_d)}{2} - 1. \end{aligned}$$

Thus $N_i + N_d > (1-2\alpha)n_v - 2$, i.e., there have been at least $(1-2\alpha)n_v - 2$ updates in the subtree of v. ■

We analyze the running time of the update algorithm. Steps 1 and 2 take time proportional to the height of the BB[α]-tree, which is bounded by $O(\log n)$. It remains to bound the time for Step 3, i.e., the time for rebalancing. We show that the amortized rebalancing time is bounded by $O(\log n)$. This will prove that the entire update algorithm has amortized update time $O(\log n)$.

Consider a node v of the BB[α]-tree, and assume we rebuild the subtree of v. Let S_v be the set of elements that are stored in the subtree of v. Note that $n_v = |S_v|$. Using the old subtree of v, we obtain the elements of S_v in sorted order, in $O(n_v)$ time. Then, in $O(n_v)$ time, we build a perfectly balanced binary tree for these elements. (See Exercise 3.2.2.) Hence, the entire rebuilding operation takes $O(n_v)$ time. We say that this visit to node v has cost $O(n_v)$. By Lemma 3.5.1, there have been $(1-2\alpha)n_v - 3$ updates in the subtree of v during which this subtree was not rebuild. During each of these updates, $O(1)$ time was spent in node v. That is, each of the provious $(1-2\alpha)n_v - 3$ visits to node v had cost $O(1)$.

This proves that the $(1-2\alpha)n_v - 2$ most recent visits to node v have total cost $O(n_v)$. Averaged over these visits, we get an upper bound of $O(1)$ per visit to node v.

To summarize, each node visited during an update operation causes $O(1)$ rebalancing costs. Since we visit $O(\log n)$ nodes during an update we get an $O(\log n)$ upper bound on the total amortized rebalancing cost. We have proved the following lemma.

Lemma 3.5.2 *Using the partial rebuilding technique, a* BB[α]-tree *can be maintained under insertions and deletions in* $O(\log n)$ *amortized time per operation.*

3.5.1 An alternative proof: the potential method

In this section we give a somewhat cleaner, but more tricky, proof of Lemma 3.5.2. We start by recalling the potential method.

Consider a data structure on which we perform a sequence of n operations. The initial data structure is denoted by D_0. For $1 \leq k \leq n$, let C_k be the cost of the k-th operation, and let D_k be the data structure that results by performing the k-th operation on D_{k-1}.

The total cost for the n operations is $\sum_{k=1}^{n} C_k$. Often it is difficult to give a good estimate for this summation. In such cases, one can try to apply the potential method:

Let Φ be a function that maps the data structure D_k to a real number $\Phi(D_k)$, $0 \leq k \leq n$. This function is called the *potential function.* Given this function, define

$$\hat{C}_k := C_k + \Phi(D_k) - \Phi(D_{k-1}), 1 \leq k \leq n.$$

That is $\hat{C}_k$ is the sum of the actual cost of the k-th operation and the increase in potential due to this operation. Now we can rewrite the total cost for the n operations:

$$\begin{aligned}\sum_{k=1}^{n} C_k &= \sum_{k=1}^{n} (\hat{C}_k - \Phi(D_k) + \Phi(D_{k-1})) \\ &= \sum_{k=1}^{n} \hat{C}_k + \Phi(D_0) - \Phi(D_n).\end{aligned}$$

Suppose that our potential function satisfies

1. $\Phi(D_0) = 0$, and
2. $\Phi(D_k) \geq 0$, for all $0 \leq k \leq n$.

Then,

$$\sum_{k=1}^{n} C_k \leq \sum_{k=1}^{n} \hat{C}_k,$$

i.e., the summation on the right hand side is an upper bound on the total cost for the n operations. The trick is to define a potential function Φ such that the summation $\sum \hat{C}_k$ can be estimated easily.

We apply this technique to analyze the amortized time of the update algorithm for BB[α]-trees. Recall that for any node v, n_v denotes the number of leaves in the subtree of v. For v an internal node, let v_l and v_r be its left an right sons, respectively. Define

$$\Delta_v := |n_{v_l} - n_{v_r}|.$$

The potential of a binary tree T is defined by

$$\Phi(T) := \gamma \sum_{v \in T : \Delta_v \geq 2} \Delta_v,$$

where γ is a constant to be fixed later.

Suppose we start with a BB[α]-tree for the empty set. Consider a sequence of n insert and delete operations. Let $T_0, T_1, T_2, \ldots, T_n$ be the sequence of BB[α]-trees obtained in this way.

First note that $\Phi(T_0) = 0$ and $\Phi(T_k) \geq 0$ for all $0 \leq k \leq n$. Also, a perfectly balanced binary tree has potential zero.

Let $1 \leq k \leq n$ and consider the k-th update operation. Let T'_k be the tree obtained after performing Steps 1 and 2 of the update algorithm. Then, T_k is obtained by performing Step 3, the rebalancing step to T'_k. Assume that during this step, we rebuild the subtree rooted at v. Hence, v is the highest node of T'_k that is out of balance.

Steps 1 and 2 take $O(\log n)$ time. Step 3 takes $O(n_v)$ time, where n_v is the number of leaves in the subtree of v in T'_k. Hence, there is a constant γ' such that the time C_k for the k-th update operation satisfies

$$C_k \leq \gamma'(\log n + n_v).$$

To estimate the increase in potential, we consider $\Phi(T_k) - \Phi(T'_k)$ and $\Phi(T'_k) - \Phi(T_{k-1})$ separately.

During the transformation from T_{k-1} into T'_k, the Δ_u values of all nodes u on the search path increase by at most one. All other Δ_u values remain unchanged. It follows that

$$\Phi(T'_k) - \Phi(T_{k-1}) \leq h_\alpha \gamma \log n,$$

where h_α is the constant that appears in the $O(\log n)$ bound on the height of a BB[α]-tree.

Consider the node v in T'_k. Because v is out of balance, we have $n_{v_l}/n_v < \alpha$ and $n_{v_r}/n_v > 1-\alpha$, or $n_{v_r}/n_v < \alpha$ and $n_{v_l}/n_v > 1-\alpha$. Assume w.l.o.g. that the first case occurs. Then,

$$n_{v_r} - n_{v_l} > (1-\alpha)n_v - \alpha n_v = (1-2\alpha)n_v.$$

If T_{kv} and T'_{kv} denote the subtrees of T_k and T'_k rooted at v, respectively, then

$$\Phi(T_k) - \Phi(T'_k) = \Phi(T_{kv}) - \Phi(T'_{kv}).$$

Since T_{kv} is perfectly balanced, its potential is zero. Moreover,

$$\Phi(T'_{kv}) \geq \gamma(n_{v_r} - n_{v_l}) > \gamma(1-2\alpha)n_v.$$

Hence,

$$\Phi(T_k) - \Phi(T'_k) < -\gamma(1-2\alpha)n_v.$$

Putting everything together, we have shown that

$$\begin{aligned}\Phi(T_k) - \Phi(T_{k-1}) &= \Phi(T_k) - \Phi(T'_k) + \Phi(T'_k) - \Phi(T_{k-1}) \\ &\leq -\gamma(1-2\alpha)n_v + h_\alpha \gamma \log n.\end{aligned}$$

This implies that

$$\begin{aligned}\hat{C}_k &= C_k + \Phi(T_k) - \Phi(T_{k-1}) \\ &\leq \gamma'(\log n + n_v) - \gamma(1-2\alpha)n_v + h_\alpha \gamma \log n.\end{aligned}$$

Note that we still have to choose the constant γ in the definition of Φ. We take $\gamma := \gamma'/(1-2\alpha)$. Then

$$\hat{C}_k \leq \gamma' \log n + h_\alpha \gamma \log n.$$

It follows that the total time for the n update operations is bounded by

$$\sum_{k=1}^{n} C_k \leq \sum_{k=1}^{n} \hat{C}_k \leq \sum_{k=1}^{n} (\gamma' + h_\alpha \gamma) \log n = O(n \log n).$$

That is, the amortized time per update operation is bounded by $O(\log n)$. This proves Lemma 3.5.2.

3.5.2 Range trees and partial rebuilding

In the beginning of Section 3.5, we already gave the basic algorithm for inserting or deleting a point p in a range tree. The algorithm consisted of three steps:

1. Search in the x-tree for the position where the x-coordinate p_x of p has to be inserted or deleted. Perform the update at this position.

2. For each node of the x-tree on the path to p_x, insert or delete p_y in its y-tree.

3. Rebalance the range tree.

Definition 3.5.2 A range tree is called a *BB*$[\alpha]$*-range tree*, if the x-tree and all the y-trees are BB$[\alpha]$-trees.

Exercise 3.5.4 Prove that the statements of Lemmas 3.2.1, 3.2.2, and Theorem 3.3.1 also hold for BB$[\alpha]$-trees.

It will be clear how we maintain BB$[\alpha]$-range trees under insertions and deletions. Step 1 can easily be performed in $O(\log n)$ time. For Step 2, we apply the partial rebuilding technique to each y-tree. Since $O(\log n)$ y-trees are updated during this step, each at an amortized cost of $O(\log n)$, Step 2 takes $O((\log n)^2)$ amortized time. After this step, all y-trees are BB$[\alpha]$-trees.

To perform Step 3, we walk the path in the x-tree to p_x in the reversed direction, and find the highest node v that is out of balance. If there is no such node, the x-tree is already a BB$[\alpha]$-tree and we are done. If v exists, then we completely rebuild the subtree of v (together with all its y-trees) as a perfectly balanced range tree. Step 3 takes $O(n_v \log n_v)$ time.

Using Lemma 3.5.1, it can be shown in exactly the same way as for BB$[\alpha]$-trees that this algorithm has an amortized update time of $O((\log n)^2)$.

Exercise 3.5.5 Prove this.

The bound on the amortized update time can also be proved with the potential method. For v a node of the x-tree, let $Y(v)$ be the y-tree of v. The potential of a range tree T is defined by

$$\Phi'(T) := \gamma \sum_{v \in x\text{-tree}:\Delta_v \geq 2} \Delta_v \log \Delta_v + \sum_{v \in x\text{-tree}} \Phi(Y(v)),$$

where Δ_v and $\Phi(\cdot)$ are as defined in Section 3.5.1.

Exercise 3.5.6 Apply the potential method to prove that the update algorithm for BB[α]-range trees has $O((\log n)^2)$ amortized time complexity.

If we combine the results of this section with those of Theorem 3.3.1 (see also Exercise 3.5.4), then we get the final result of this chapter:

Theorem 3.5.1 *Using* BB[α]-range tree*s, which have size $O(n \log n)$, we can solve the dynamic planar L_∞-post-office problem with a query time of $O((\log n)^2)$ and an amortized update time of $O((\log n)^2)$.*

Corollary 3.5.1 *Let $\varepsilon > 0$ be a constant. The dynamic $(1+\varepsilon)$-approximate L_2-post-office problem can be solved using $O(n \log n)$ space with a query time of $O((\log n)^2)$ and an amortized update time of $O((\log n)^2)$.*

Exercise 3.5.7 Give an argument why the layering technique of Section 3.4 is difficult to apply if also insertions and deletions have to be supported.

3.6 Further reading

In this chapter, we considered two topics from computational geometry: the post-office problem and range trees. For more information about these topics and geometric algorithms, see the books by Mehlhorn [14], Preparata and Shamos [18], Edelsbrunner [6] and Mulmuley [16]. The partial rebuilding technique applied to range trees is due to Lueker [13]. He investigates D-dimensional range trees for any fixed dimension $D \geq 2$. In Overmars [17], more so-called dynamization techniques can be found. For more applications of the potential method, see the book of Cormen, Leiserson and Rivest [4]. The layering technique is due to Willard and is described in [18]. A generalization of this technique that also supports insertions and deletions is given in Mehlhorn and Näher [15].

The algorithms for the L_∞-post-office problem and the approximate L_2-post-office problem are due to Kapoor and Smid [9]. For the static approximate post-office problem, there is a solution having $O(\log n)$ search time that uses $O(n)$ space. See Arya et al. [2].

A solution to Übungsaufgabe 5.8 can be found in Smid [25]. Übungsaufgaben 5.10 and 5.11 are based on results of Keil and Gutwin [10], Salowe [23] and Ruppert and Seidel [22].

Chapter 4

Mantaining order in a list

In this chapter we apply the potential method of Section 3.5.1 to analyze an algorithm for the *order maintenance problem.* In this problem, we perform the following three types of operations on a linked list L:

Insert(x, y)**:** Given a pointer to element x in L, insert element y immediately after x.

Delete(x)**:** Given a pointer to element x in L, delete it from the list.

Order(x, y)**:** Given pointers to elements x and y in L, decide if x is before y in the list.

There are two obvious solutions for this problem. We can build a binary tree having the elements of L in its leaves in the order in which they occur in this list. Then, insertions and deletions can be performed in $O(\log n)$ time, if n is the number of elements in L. To answer an order query, we determine the lowest common ancestor z of x and y, by walking up the tree. Then, x is before y in L iff x is in the left subtree of z. The time for an order query is bounded by $O(\log n)$.

In the second obvious solution, we give each element x in L a label $v(x)$, such that these labels increase if we walk along L. Then, an order query is particularly simple: x is before y in L iff $v(x) < v(y)$. To insert element y immediately after x, we put y between x and its successor z and give it label $v(y) = (v(x) + v(z))/2$. Clearly, in this way the labels still increase if we walk along L. To delete an element, we just remove it together with its label. In this solution, each operation takes only $O(1)$ time. The labels, however, can become very large numbers: If we perform operations Insert(x, y_i), $i \geq 1$, then we need more and more bits to represent the labels of the y_i's.

In this chapter, we give an extension of the second solution. Be relabeling certain elements of the list, we guarantee that each label can be represented by $O(\log n)$ bits, and each operation takes $O(1)$ amortized time.

4.1 An $O(\log n)$ amortized time solution

The data structure consists of the linked list L. Each element x is labeled with an integer $v(x)$, and it has a pointer to its successor $s(x)$. For convenience, we add an artificial element b called the *base*. This element is never deleted and it never occurs in an order query. The base has label $v(b)$ and its successor $s(b)$ is the leftmost element of L. The successor of the rightmost element of L is b. In this way, we get a circularly linked list. Note that order queries are w.r.t. the original list and, hence, these are well-defined.

We denote the current size of L by n. The labels $v(x)$ are integers from $\{0,1,2,\ldots,M-1\}$. We assume that $M \geq 2n^3$. For each element x in L we define

$$\begin{aligned} v_b(x) &:= (v(x) - v(b)) \bmod M, \\ v_b^*(x) &:= \begin{cases} M & \text{if } s(x) = b, \\ v_b(s(x)) & \text{if } s(x) \neq b. \end{cases} \end{aligned}$$

Note that we only store the value $v(x)$ with element x. Given a pointer to x, we can compute $v_b(x)$ and $v_b^*(x)$ in $O(1)$ time. The following invariant will always be maintained.

Invariant: $v_b(x) < v_b^*(x)$ for all elements x.

Lemma 4.1.1 *Let x and y be elements of L, both not equal to the base b. Then, x is before y in L iff $v_b(x) < v_b(y)$.*

Proof: Let x_1, x_2, ..., x_{n-1} be the non-base elements of L in the order in which they appear in L. The invariant implies that

$$v_b(x_i) < v_b^*(x_i) = v_b(s(x_i)) = v_b(x_{i+1}), 1 \leq i \leq n-2.$$

Hence,

$$v_b(x_1) < v_b(x_2) < \ldots < v_b(x_{n-1}).$$

■

According to this lemma, an order query can be solved in $O(1)$ time, just by computing and comparing $v_b(x)$ and $v_b(y)$. To delete element x, we remove this element together with its label $v(x)$. This also takes $O(1)$ time.

Exercise 4.1.1 Convince yourself that a deletion does not destroy the invariant.

Clearly, the difficulty is to maintain the invariant under insertions. We assume that we start with a list L containing only the base b. We give it an arbitrary label $v(b)$ from $\{0,1,2,\ldots,M-1\}$. At this moment, we have $s(b) = b$. Hence, $v_b(b) = 0$ and $v_b^*(x) = M$. Therefore, the invariant holds initially.

Consider the list L at any moment, and assume we perform the operation Insert(x, y). If $v_b(x) + 2 \leq v_b^*(x)$ then there is room to give y a label $v(y)$ such that $v_b(y)$ lies in between $v_b(x)$ and $v_b^*(x)$. What happens if $v_b(x) + 1 = v_b^*(x)$? In order to give y a label $v(y)$, we relabel the successor of x. Of course, if $v_b(s(x)) + 1 = v_b^*(s(x))$, we also relabel $s(s(x))$, and so on. In this way, we might have to relabel many elements. Be relabeling in such a way that the differences of the new values $v_b^*(x) - v_b(x)$, $v_b^*(s(x)) - v_b(s(x)), \ldots$, are large enough, we can guarantee that the amortized number of relabelings is $O(\log M)$ per insertion.

Now we can give the details. For each element x of L, define $s^0(x) :=$ x and $s^i(x) := s(s^{i-1}(x))$ for $i \geq 1$. That is, $s^i(x)$ is the i-th successor of x. Define $w_i := (v_b(s^i(x)) - v_b(x)) \bmod M$ for $0 \leq i \leq n-1$, and $w_n := M$.

Consider the operation Insert(x, y):

1. Compute the values $v_b(x)$ and $v_b^*(x)$.

2. If $v_b(x) + 2 \leq v_b^*(x)$, then we add y immediately after x and give it label
$$v(y) := \left(\left\lfloor \frac{v_b(x) + v_b^*(x)}{2} \right\rfloor + v(b) \right) \bmod M.$$

3. If $v_b(x) + 1 = v_b^*(x)$, then we do the following: We compute the minimal $j > 0$ such that $w_j \geq 2j^3$. Next, we compute integers $D \geq 1$ and f, $0 \leq f < j$ such that $w_j = D \cdot j + f$. We relabel the elements $s(x), s^2(x), \ldots, s^{j-1}(x)$: For $1 \leq i \leq j-1$, we give $s^i(x)$ the new label
$$v(s^i(x)) := \begin{cases} (v(x) + i(D+1)) \bmod M, & \text{if } 1 \leq i \leq f, \\ (v(x) + f(D+1) + (i-f)D) \bmod M, & \text{if } f+1 \leq i \leq j-1. \end{cases}$$
We now have $v_b(x) + 2 \leq v_b^*(x)$. (See Lemma 4.1.4 below.) We add y as in Step 2.

Note that during the algorithm, the value of $v(b)$ might change. Then, all values $v_b(z)$ also change. We do not have to worry about this because we do not store these values; we only store the values $v(z)$.

The correctness of the insertion algorithm follows from the following three lemmas.

Lemma 4.1.2 *Step 2 correctly maintains the invariant.*

Proof: We consider the value $v_b^*(x)$ as it was before the insertion. Note that
$$v_b(y) = (v(y) - v(b)) \bmod M = \left\lfloor \frac{v_b(x) + v_b^*(x)}{2} \right\rfloor .$$
Since $v_b(x) + 2 \leq v_b^*(x)$, we infer that $v_b(y) \geq v_b(x) + 1 > v_b(x)$ and $v_b(y) \leq v_b^*(x) - 1 < v_b^*(x)$.

After the insertion, the value of $v_b^*(x)$ is equal to $v_b(y)$. Hence, the invariant still holds for x. The value of $v_b^*(y)$ is equal to the old value of $v_b^*(x)$. Therefore, the invariant also holds for y. This completes the proof, because no other values $v_b^*(\cdot)$ are changed during Step 2. ∎

Lemma 4.1.3 *Assume Step 3 of the insertion algorithm is performed. The integer j of this step exists, and we have $D \geq 8$.*

Proof: Since $w_n = M \geq 2n^3$, there is a $j > 0$ such that $w_j \geq 2j^3$. Hence, there is also a smallest $j > 0$ having this property.

If $s(x) = b$, then $v_b(x) = v_b^*(x) - 1 = M - 1$. Since $v_b(b) = 0$, it follows that $w_1 = (v_b(s(x)) - v_b(x)) \bmod M = (0 - M + 1) \bmod M = 1$. If $s(x) \neq b$, then $w_1 = (v_b^*(x) - v_b(x)) \bmod M = 1$. That is, in both cases, we have $w_1 = 1 < 2 \cdot 1^3$. Therefore, $j \geq 2$ which implies that $D = \lfloor w_j/j \rfloor \geq 2j^2 \geq 8$. ∎

Lemma 4.1.4 *Step 3 correctly maintains the invariant. In particular, immediately before adding y, we have $v_b(x) + 2 \leq v_b^*(x)$.*

Proof: We distinguish two cases, depending on whether or not the base b gets a new label.

Assume b does not get a new label. Consider what happens until we add y at the end of Step 3. Look at the value $v_b(s^j(x))$. (This value does not change in Step 3.) First assume that $s^j(x) \neq b$. The invariant implies that $v_b(s^j(x)) > v_b(x)$. Since all labels $v_b(\cdot)$ are between zero and $M - 1$, we have $v_b(s^j(x)) - v_b(x) < M$. Therefore,

$$w_j = (v_b(s^j(x)) - v_b(x)) \bmod M = v_b(s^j(x)) - v_b(x).$$

Let $1 \leq i \leq j - 1$ and consider the new value $v_b(s^i(x))_{\text{new}}$. We have

$$v_b(s^i(x))_{\text{new}} = \begin{cases} (v_b(x) + i(D+1)) \bmod M & \text{if } 1 \leq i \leq f, \\ (v_b(x) + f(D+1) + (i-f)D) \bmod M & \text{if } f+1 \leq i \leq j-1. \end{cases}$$

Note that $v_b(x) + i(D+1) > 0$ for $1 \leq i \leq f$, and $v_b(x) + f(D+1) + (i-f)D > 0$ for $f+1 \leq i \leq j-1$. Also, both these values are at most equal to

$$v_b(x) + f(D+1) + (j-1-f)D = v_b(x) + w_j - D = v_b(s^j(x)) - D < M.$$

As a result,

$$v_b(s^i(x))_{\text{new}} = \begin{cases} v_b(x) + i(D+1) & \text{if } 1 \leq i \leq f, \\ v_b(x) + f(D+1) + (i-f)D & \text{if } f+1 \leq i \leq j-1. \end{cases}$$

Since $v_b(s^j(x))_{\text{new}} = v_b(s^j(x))_{\text{old}} = v_b(x) + Dj + f$, the latter equation also holds for $i = j$. This proves that

$$v_b^*(s^i(x))_{\text{new}} = v_b(s^{i+1}(x))_{\text{new}} \geq v_b(s^i(x))_{\text{new}} + D, \quad 0 \leq i \leq j-1.$$

Hence, immediately before we add y, we have

$$v_b(s^i(x))_{\text{new}} < v_b^*(s^i(x))_{\text{new}}, \quad 0 \leq i \leq j-1,$$

and also

$$v_b^*(x)_{\text{new}} \geq v_b(x)_{\text{new}} + D \geq v_b(x)_{\text{new}} + 2,$$

because $D \geq 8$. That is, at this moment, the invariant holds and we are in the same situation as in Step 2. Then Lemma 4.1.2 implies that the invariant still holds after y has been added.

Exercise 4.1.2 Treat the case where $s^j(x) = b$. Note that $w_j = M - v_b(x)$.

Now we analyze the case where b gets a new label. This case is more complicated, because now *all* values $v_b(\cdot)$ change in Step 3. Again, we first look at the situation immediately before we add y.

Let $1 \leq k \leq j-1$ be such that $s^k(x) = b$. First note that

$$v_b(s^i(x))_{\text{new}} = \begin{cases} (v_b(x)_{\text{new}} + i(D+1)) \bmod M & \text{if } 1 \leq i \leq f, \\ (v_b(x)_{\text{new}} + f(D+1) + (i-f)D) \bmod M & \text{if } f+1 \leq i \leq j-1. \end{cases}$$

Moreover, $v_b(s^k(x))_{\text{new}} = v_b(b)_{\text{new}} = 0$. For $k \leq i \leq j-1$, we obtain $v_b(s^i(x))_{\text{new}}$ by adding $i-k$ times the quantity $D+1$ or D to $v_b(s^k(x))_{\text{new}} = 0$, and reducing the result modulo M. Since we never add more than $f(D+1)+(j-1-f)D = w_j - D < M$ to $v_b(s^k(x))_{\text{new}} = 0$, the reduction modulo M is not necessary. This proves that

$$v_b(s^i(x))_{\text{new}} < v_b(s^{i+1}(x))_{\text{new}} = v_b^*(s^i(x))_{\text{new}}, \quad k \leq i \leq j-2.$$

Similarly, for $1 \leq i \leq k-1$, we obtain $v_b(s^i(x))_{\text{new}}$ by substracting $k-i$ times the quantity $D+1$ or D from M. (Again, the reduction modulo M is not necessary.) This proves that

$$v_b(s^i(x))_{\text{new}} < v_b^*(s^i(x))_{\text{new}}, \quad 1 \leq i \leq k-1.$$

It remains to prove that $v_b(z)_{\text{new}} < v_b^*(z)_{\text{new}}$ for all elements z whose $v(\cdot)$-label did not change, and for $z = s^{j-1}(x)$.

Let z be an element such that $v(z)_{\text{old}} = v(z)_{\text{new}}$. We claim that $v_b(z)_{\text{new}} \leq v_b(z)_{\text{old}}$. First note that

$$w_k = M - v_b(x)_{\text{old}},$$

and

$$w_j = v_b(s^j(x))_{\text{old}} + M - v_b(x)_{\text{old}}.$$

Also,

$$\begin{aligned} v_b(s^j(x))_{\text{new}} &= (v(s^j(x))_{\text{new}} - v(b)_{\text{new}}) \bmod M \\ &= (v(s^j(x))_{\text{old}} - v(b)_{\text{old}} + v(b)_{\text{old}} - v(x) + v(x) - v(b)_{\text{new}}) \bmod M \\ &= (v_b(s^j(x))_{\text{old}} - v_b(x)_{\text{old}} + v(x) - v(b)_{\text{new}}) \bmod M. \end{aligned}$$

Assume that $k \leq f$. Then

$$v_b(s^j(x))_{\text{new}} = (v_b(s^j(x))_{\text{old}} - v_b(x)_{\text{old}} - k(D+1)) \bmod M.$$

We have

$$\begin{aligned} v_b(x)_{\text{old}} + k(D+1) &= M - w_k + k(D+1) \\ &\geq M - w_k + kD \\ &= M - w_k + k\lfloor w_j/j \rfloor \\ &\geq M - 2k^3 + k \cdot 2j^2 \\ &= M + 2k(j^2 - k^2) \\ &\geq M. \end{aligned}$$

Also

$$\begin{aligned} v_b(x)_{\text{old}} + k(D+1) &\leq v_b(x)_{\text{old}} + k(w_j/j + 1) \\ &= v_b(x)_{\text{old}} + \frac{k}{j}(v_b(s^j(x))_{\text{old}} + M - v_b(x)_{\text{old}}) + k. \end{aligned}$$

This last expression is at most equal to $v_b(s^j(x))_{\text{old}} + M$ iff $k \leq (1 - k/j)w_j$. This is true because

$$w_j \geq 2j^3 \geq \frac{kj}{j-k} = \frac{k}{1-k/j}.$$

Hence we know that

$$M \leq v_b(x)_{\mathrm{old}} + k(D+1) \leq M + v_b(s^j(x))_{\mathrm{old}}.$$

Therefore,

$$\begin{array}{rcl} v_b(s^j(x))_{\mathrm{new}} & = & (v_b(s^j(x))_{\mathrm{old}} - (v_b(x)_{\mathrm{old}} + k(D+1))) \bmod M \\ & = & v_b(s^j(x))_{\mathrm{old}} - (v_b(x)_{\mathrm{old}} + k(D+1) - M) \\ & \leq & v_b(s^j(x))_{\mathrm{old}}. \end{array}$$

If $f+1 \leq k \leq j-1$, then this inequality can be proved in a similar way. Recall that we want to prove that $v_b(z)_{\mathrm{new}} \leq v_b(z)_{\mathrm{old}}$. The value $v_b(z)_{\mathrm{new}}$ is obtained by shifting $v_b(z)_{\mathrm{old}}$ over a certain amount modulo M. The value $v_b(s^j(x))_{\mathrm{new}}$ is obtained by shifting $v_b(s^j(x))_{\mathrm{old}}$ over the *same* amount. Therefore, since $v_b(s^j(x))_{\mathrm{old}} \leq v_b(z)_{\mathrm{old}}$, we also have $v_b(z)_{\mathrm{new}} \leq v_b(z)_{\mathrm{old}}$. Using the same argument, we conclude that $v_b(z)_{\mathrm{old}} < v_b^*(z)_{\mathrm{old}}$ implies $v_b(z)_{\mathrm{new}} < v_b^*(z)_{\mathrm{new}}$.

In a similar way, it can be shown that $v_b(s^{j-1}(x))_{\mathrm{new}} < v_b^*(s^{j-1}(x))_{\mathrm{new}}$. It follows that the invariant holds immediately before we add y. Since $v_b^*(x)_{\mathrm{new}} \geq v_b(x)_{\mathrm{new}} + 2$ at this moment, Lemma 4.1.2 implies that the invariant still holds after y has been added. ∎

4.2 The analysis

We saw already that order queries and deletions take $O(1)$ worst-case time. In this section, we show that the amortized insertion time is bounded by $O(\log M)$.

If $x_0 = b, x_1, x_2, \ldots, x_{n-1}$ is the sequence of elements of L obtained by walking along it from left to right, then we define

$$g_k := v_b^*(x_k) - v_b(x_k), \quad \text{for } 0 \leq k \leq n-1.$$

We call g_k a *gap*. We use the following potential function:

$$\Phi := c \cdot \sum_{k=0}^{n-1} \log M/g_k,$$

where c is a constant to be determined later.

Initially, the list only contains the base b. Then, $n = 1$, $g_0 = v_b^*(b) - v_b(b) = M$ and $\Phi = 0$.

Next, we claim that at any time, the potential is non-negative: The invariant implies that $g_k > 0$. Also $g_k \leq v_b^*(x_k) \leq M$. Hence, $\log M/g_k \geq 0$, which implies that $\Phi \geq 0$.

By the result of Section 3.5.1, we have to estimate the actual cost of any operation plus the increase in potential due to this operation.

During an order query, we do not change any labels. Therefore, the potential does not change.

Consider a deletion of element x. Note that $x \neq b$, since we never delete the base. If $x = x_k$, then the potential increases by

$$c \log \frac{M}{g_{k-1} + g_k} - c \log \frac{M}{g_{k-1}} - c \log \frac{M}{g_k} = c \log \frac{g_{k-1} g_k}{M(g_{k-1} + g_k)} \leq 0.$$

The actual cost of the deletion is constant. This proves that the amortized time for a deletion is bounded by a constant. (Of course, we already know that even the worst-case time for a deletion is $O(1)$.)

Now consider the operation Insert(x, y). If we add y between x_k and x_{k+1}, then Step 2 takes time $O(1)$. We assume w.l.o.g. that the constant in this big-O is one. This addition increases the potential by

$$\begin{aligned} c \log \frac{M}{\lceil g_k/2 \rceil} + c \log \frac{M}{\lfloor g_k/2 \rfloor} - c \log \frac{M}{g_k} &\leq c \log \frac{M}{g_k/2} + c \log M - c \log \frac{M}{g_k} \\ &= c + c \log M. \end{aligned}$$

Now consider Step 3. This step takes $O(j)$ time. Assume w.l.o.g. that the constant in the big-O is one. Renumber the indices such that before the relabeling we have gaps $g_0, g_1, g_2, \ldots, g_{j-1}$ with $\sum_{l=0}^{j-1} g_l = w_j$. Afterwards, we have f gaps of size $D + 1$ and $j - f$ gaps of size D. Therefore, the increase in potential is given by

$$c \left(f \log \frac{M}{D+1} + (j - f) \log \frac{M}{D} - \sum_{l=0}^{j-1} \log \frac{M}{g_l} \right).$$

We will prove that this expression is at most equal to $-j$, if the constant c in the definition of Φ is chosen appropriately. This will prove that the amortized cost for the entire insert algorithm is at most

$$1 + c + c \log M + j - j = O(\log M).$$

Let

$$A := f \log \frac{M}{D+1} + (j-f) \log \frac{M}{D} - j \log \frac{M}{w_j/j},$$

and

$$B := j \log \frac{M}{w_j/j} - \sum_{l=0}^{j-1} \log \frac{M}{g_l}.$$

Then we want an upper bound for $c(A+B)$. Since

$$\frac{w_j/j}{D+1} = \frac{w_j/j}{(w_j-f)/j+1} = \frac{1}{1+(1-f/j)j/w_j},$$

and

$$\frac{w_j/j}{D} = \frac{w_j/j}{(w_j-f)/j} = \frac{1}{1-f/j \cdot j/w_j},$$

we have

$$\begin{aligned} A &= j\left(\frac{f}{j}\log\frac{w_j/j}{D+1} + \frac{j-f}{j}\log\frac{w_j/j}{D}\right) \\ &= j(-f'\log(1+(1-f')p) - (1-f')\log(1-f'p)), \end{aligned}$$

where $f' := f/j$ and $p := j/w_j$. Note that $0 \le f' < 1$ and $0 \le p \le 1/8$. Maximizing over all $0 \le f' < 1$ and $0 \le p \le 1/8$ yields (see Section 4.4)

$$A \le \frac{\log e}{56} \cdot j.$$

Next we consider B. We know that $w_l := g_0 + g_1 + \ldots + g_{l-1} < 2l^3$ for $2 \le l < j, w_1 = g_0 = 1$, and $w_j \ge 2j^3$. Let $i := \lceil j/2 \rceil$. Define

$$B_1 := j \log \frac{M}{w_j/j} - \sum_{l=0}^{i-1} \log \frac{M}{g_l \frac{w_j/2}{w_i}} - \sum_{l=i}^{j-1} \log \frac{M}{g_l \frac{w_j/2}{w_j-w_i}},$$

and

$$B_2 := \sum_{l=0}^{i-1} \log \frac{M}{g_l \frac{w_j/2}{w_i}} + \sum_{l=i}^{j-1} \log \frac{M}{g_l \frac{w_j/2}{w_j-w_i}} - \sum_{l=0}^{j-1} \log \frac{M}{g_l}.$$

Note that $B = B_1 + B_2$. Observe that

$$\sum_{l=0}^{i-1} g_l \frac{w_j/2}{w_i} + \sum_{l=i}^{j-1} g_l \frac{w_j/2}{w_j - w_i} = w_j.$$

Therefore, the convexity of the function $-\log x$ implies that $B_1 \leq 0$. (See Section 4.4.) To bound B_2, we argue as follows:

$$B_2 = -i \log \frac{w_j/2}{w_i} - (j-i) \log \frac{w_j/2}{w_j - w_i}.$$

Recall that $i = \lceil j/2 \rceil$. If $j = 2i$, then $w_j \geq 2j^3 = 2 \cdot 8i^3 \geq 8w_i$. If $j = 2i - 1$, $i \geq 2$, then

$$w_j \geq 2j^3 = 2(2i-1)^3 \geq 2 \cdot 3i^3 \geq 3w_i.$$

That is, we always have $w_j \geq 3w_i$. This implies that

$$\log \frac{w_j/2}{w_i} > 0,$$

and

$$\log \frac{w_j/2}{w_j - w_i} < 0.$$

Hence,

$$\begin{aligned} B_2 &\leq -\frac{j}{2}\left(\log \frac{w_j/2}{w_i} + \log \frac{w_j/2}{w_j - w_i}\right) \\ &= -\frac{j}{2} \log \frac{r^2}{4(r-1)}, \end{aligned}$$

where $r := w_j/w_i \geq 3$. For $r \geq 3$, the function $r^2/(r-1)$ attains its minimum when $r = 3$. As a result,

$$B_2 \leq -\frac{j}{2} \log \frac{9}{8}.$$

Combining everything, we conclude that the potential increase due to the relabeling is bounded by

$$\begin{aligned} c(A+B) &= c(A + B_1 + B_2) \\ &\leq c\left(\frac{\log e}{56} \cdot j + 0 - \frac{j}{2} \log \frac{9}{8}\right). \end{aligned}$$

Taking

$$c = \frac{1}{\frac{1}{2}\log\frac{9}{8} - \frac{\log e}{56}} \approx 17,$$

we find that the potential increase is at most $-j$, which is exactly what we wanted to show. We summarize our result.

Theorem 4.2.1 *Let L be a list storing n elements and let M be an integer such that $M \geq 2n^3$. We can solve the order maintenance problem on L using labels from $\{0, 1, 2, \ldots, M-1\}$ such that order queries and deletions take $O(1)$ worst-case time, and insertions take $O(\log M)$ amortized time.*

4.3 An $O(1)$ amortized solution

In this section, we improve the result of Theorem 4.2.1. To be more precise, we first reduce the amortized insertion time from $O(\log M)$ to $O(1)$. Then we show how to maintain the assumption that $M \geq 2n^3$ and guarantee that the labels are integers consisting of $O(\log n)$ bits rather than $O(\log M)$.

To reduce the insertion time, we apply the *bucketing technique*. Let $x_1, x_2, \ldots, x_{n-1}$ be the non-base elements of L in the order in which they appear in L. We assume that $\log M$ is an even integer. Partition the elements into buckets

$$b_i := \langle x_{1+i\log M}, x_{2+i\log M}, \ldots, x_{(i+1)\log M}\rangle, \quad 0 \leq i < n/\log M.$$

Our data structure consists of the following:

1. A list B storing the names of the buckets b_0, b_1, b_2,..., in this order. For this list, we apply the labeling technique of Theorem 4.2.1. With each bucket name in B, we store the number of elements of L that are contained in the corresponding bucket.

2. The list L. Each element x of L contains a pointer to its successor in L and a pointer to the bucket in B that contains x.

3. For each $0 \leq i < n/\log M$ and $1 \leq j \leq \log M$, we give element $x_{j+i\log M}$ label $j \cdot \sqrt{M}$.

First, we give the algorithm for inserting an element. Consider an operation Insert(x, y). Recall that we get a pointer to the occurrence of x in L. Let z be the successor of x. If x and z belong to the same bucket b (this can be checked by following the pointers from x and z to B), then we give y a label which is the rounded average of the labels of x and z. We add y to L, between x and z, give it a pointer to b, and increase b's counter by one. If x and z belong to different buckets, then we give y label $\lfloor l/2 \rfloor$, where l is z's label, add y to L, give it a pointer to z's bucket and increase this bucket's counter by one.

Initially, the labels within one bucket have gaps of size $\sqrt{M}$. Therefore, our algorithm guarantees that the labels within one bucket are increasing integers, as long as the bucket contains at most $3/2 \log M$ elements. If a bucket contains $3/2 \log M$ elements, then we split it into two equal-sized buckets. We add the new bucket to B, using the insertion algorithm of Section 4.1, and give each element involved a pointer to the "its" bucket. Within each of the new buckets, we relabel the elements such that the gaps have size $\sqrt{M}$.

The deletion algorithm is similar. As soon as a bucket contains $1/2 \log M$ elements, we merge it with a neighboring bucket.

Exercise 4.3.1 Give the details of the deletion algorithm.

The algorithm for answering an order query should be clear. Consider the query Order(x, y). We follow the pointers from x and y to the buckets b_x and b_y in B, respectively. If $b_x \neq b_y$, then x is to the left of y in L iff b_x is to the left of b_y in B. The latter condition can be verified because we apply the labeling technique of Theorem 4.2.1 to the list B.

Assume the $b_x = b_y$. Then x and y belong to the same bucket. Our update algorithms ensure that the labels of the elements within one bucket are increasing. As a result, x is to the left of y in L iff x's label is smaller than y's label.

Theorem 4.3.1 *Let L be a list storing n elements and let M be an integer such that $M \geq 2n^3$. We can solve the order maintenance problem on L using labels from $\{0, 1, 2, \ldots, M-1\}$ such that order queries take $O(1)$ worst-case time, and insertions and deletions take $O(1)$ amortized time.*

Proof: To merge or split buckets, we need $O(\log M)$ time to update the list L and, by Theorem 4.2.1, $O(\log M)$ amortized time to update the list B. If such an operation occurs, however, then there must have been $\Omega(\log M)$ updates in the bucket that did not cause merge or split operations. ■

Until now, we always assumed that $M \geq 2n^3$. What happens if n becomes too large because of many insertions? Also, until now we used labels consisting of $\Theta(\log M)$ bits. If we delete many elements, i.e., if n becomes small, then $\log M$ will be very large compared to $\log n$. Is there a way to use labels of $O(\log n)$ bits?

Here is a solution. We occasionally rebuild the entire data structure. Let n_0 be the size of L at the most recent rebuilding. Then we take $M = 27/4 \cdot n_0^3$, and use the algorithm of Theorem 4.3.1, as long as n, the current size of L, is at least $n_0/2$ and at most $3n_0/2$. If n becomes too small or too large, then we completely rebuild the data structure, using a new value for M.

Between two rebuildings, the current size n satisfies $n_0/2 \leq n \leq 3n_0/2$. Therefore,

$$2n^3 \leq 2(3n_0/2)^3 = M,$$

and

$$M = 27/4 \cdot n_0^3 \leq 27/4 \cdot (2n)^3 = 54n^3.$$

That is, the condition $M \geq 2n^3$ always holds, and the labels always consist of $O(\log M) = O(\log n)$ bits.

A rebuilding operation takes $O(n_0)$ time. (Why?) However, after this operation, we do not rebuild during the next $n_0/2$ updates. Hence, rebuilding only adds $O(1)$ to the amortized update time.

We have proved our final result:

Theorem 4.3.2 *Let L be a list storing n elements. We can solve the order maintenance problem on L using labels consisting of $O(\log n)$ bits such that order queries take $O(1)$ worst-case time, and insertions and deletions take $O(1)$ amortized time.*

4.4 Convex functions and inequalities

In Section 4.2, we used the convexity of the function $-\log x$ to conclude that the value of B_1 is non-positive. In the present section, we give the details.

Definition 4.4.1 Let f be a real-valued function defined on the (finite or infinite) interval $[a : b]$. This function is called *convex* if for all $a \leq x, y \leq b$ and all $0 \leq \lambda \leq 1$,

$$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda) f(y).$$

Theorem 4.4.1 *Let f be a convex function. Let x_1, x_2, …, x_n be real numbers in the interval $[a : b]$, and let λ_1, λ_2, …, λ_n be real numbers in the interval $[0 : 1]$ such that $\sum \lambda_i = 1$. Then*

$$f(\sum_{i=1}^{n} \lambda_i x_i) \leq \sum_{i=1}^{n} \lambda_i f(x_i).$$

Proof: The theorem can be proved by induction on n. For $n = 2$, the claim follows from the definition of convexity. We give the proof for $n = 3$ and leave the general case to the reader.

Let $a \leq x, y, z \leq b$, $0 \leq \alpha, \beta, \gamma \leq 1$, such that $\alpha + \beta + \gamma = 1$. We can assume w.l.o.g. that $\alpha \neq 1$. Since

$$\alpha x + \beta y + \gamma z = \alpha x + (1-\alpha)\left(\frac{\beta}{1-\alpha} y + \frac{\gamma}{1-\alpha} z\right),$$

the convexity of f implies that

$$f(\alpha x + \beta y + \gamma z) \leq \alpha f(x) + (1-\alpha) f\left(\frac{\beta}{1-\alpha} y + \frac{\gamma}{1-\alpha} z\right).$$

Note that $\beta/(1-\alpha)$ and $\gamma/(1-\alpha)$ are real numbers in the interval $[0 : 1]$ with sum one. Therefore, again using the convexity of f, we get

$$\begin{aligned} f(\alpha x + \beta y + \gamma z) &\leq \alpha f(x) + (1-\alpha)\left(\frac{\beta}{1-\alpha} f(y) + \frac{\gamma}{1-\alpha} f(z)\right) \\ &= \alpha f(x) + \beta f(y) + \gamma f(z). \end{aligned}$$

∎

Theorem 4.4.2 *Let f be a function on $[a : b]$. If f is twice differentiable and $f''(x) \geq 0$ for all $a \leq x \leq b$, then f is convex.*

Proof: Let $x, y \in [a : b]$ and $\lambda \in [0 : 1]$. We have to show that $f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda) f(y)$. We assume w.l.o.g. that $x < y$ and $0 < \lambda < 1$.

Let $z = \lambda x + (1-\lambda)y$. Then $x < z < y$. By the mean-value theorem, there is an α, $z \leq \alpha \leq y$, such that

$$f(y) - f(z) = (y - z) \cdot f'(\alpha).$$

Similarly, there is a β, $x \leq \beta \leq z$, such that

$$f(z) - f(x) = (z - x) \cdot f'(\beta).$$

Since

$$x - z = x - \lambda x - (1-\lambda)y = (1-\lambda)(x - y)$$

and

$$y - z = y - \lambda x - (1-\lambda)y = \lambda(y - x),$$

we get

$$\begin{aligned} \lambda f(x) + (1-\lambda) f(y) - f(z) &= \lambda(f(x) - f(z)) + (1-\lambda)(f(y) - f(z)) \\ &= \lambda(x - z) \cdot f'(\beta) + (1-\lambda)(y - z) \cdot f'(\alpha) \\ &= \lambda(1-\lambda)(y - x)(f'(\alpha) - f'(\beta)). \end{aligned}$$

Note that f' is non-decreasing and $\alpha \geq \beta$. Hence the last expression is at least equal to zero. ■

Now we can return to Section 4.2. We had

$$\begin{aligned} B_1 &= j \log \frac{M}{w_j/j} - \sum_{l=0}^{i-1} \log \frac{M}{g_l \frac{w_j/2}{w_i}} - \sum_{l=i}^{j-1} \log \frac{M}{g_l \frac{w_j/2}{w_j - w_i}} \\ &= -j \log w_j/j + \sum_{l=0}^{i-1} \log g_l \frac{w_j/2}{w_i} + \sum_{l=i}^{j-1} \log g_l \frac{w_j/2}{w_j - w_i}, \end{aligned}$$

and

$$w_j = \sum_{l=0}^{i-1} g_l \frac{w_j/2}{w_i} + \sum_{l=i}^{j-1} g_l \frac{w_j/2}{w_j - w_i}.$$

By Theorem 4.4.2, the function $f(x) = -\log x$ is convex. Let

$$x_l := \begin{cases} g_l \frac{w_j/2}{w_i} & \text{if } 0 \le l \le i-1, \\ g_l \frac{w_j/2}{w_j - w_i} & \text{if } i \le l \le j-1, \end{cases}$$

and $\lambda_l := 1/j$ for $0 \le l \le j-1$. Then Theorem 4.4.1 implies that $B_1 \le 0$, which was claimed in Section 4.2.

Exercise 4.4.1 Prove that

$$x \ln \frac{x}{a} + y \ln \frac{y}{b} \ge (x+y) \ln \frac{x+y}{a+b}$$

for x, y, a, $b > 0$. (Hint: $x \ln x$ is convex.)

In Section 4.2, we also had to maximize the function

$$F(f,p) := -f \log(1 + (1-f)p) - (1-f)\log(1 - fp),$$

where $0 \le f < 1$ and $0 \le p \le 1/8$. (In Section 4.2, we wrote f' instead of f.) Here, we give the details.

Lemma 4.4.1 *For all $x > -1$, $\log(1+x) \ge \frac{x}{1+x} \cdot \log e$.*

Proof: Compute the minimum of the function $\ln(1+x) - x/(1+x)$. ∎

We consider the partial derivative of F w.r.t. p:

$$\frac{\partial F}{\partial p} = f(1-f)\left(\frac{1}{1-fp} - \frac{1}{1+(1-f)p}\right)\log e.$$

Since each term in this product is non-negative, it follows that for each fixed f, $0 \le f < 1$, the function $F(f,p)$ is non-decreasing in p. Hence,

$$F(f,p) \le -f \log(1 + (1-f)/8) - (1-f)\log(1 - f/8).$$

Applying Lemma 4.4.1 gives

$$\begin{aligned} F(f,p) &\le \left(-f \frac{(1-f)/8}{1+(1-f)/8} - (1-f)\frac{-f/8}{1-f/8}\right)\log e \\ &= \frac{f(1-f)\log e}{64(1-f/8)(1+(1-f)/8)}. \end{aligned}$$

Since $f \le 1$, $1-f \le 1$, $1 - f/8 \ge 7/8$ and $1 + (1-f)/8 \ge 1$, we conclude that

$$F(f,p) \le \frac{\log e}{56},$$

which is exactly what was claimed in Section 4.2.

4.5 Further reading

This chapter is based on Dietz and Sleator [5].

Chapter 5

Übungsaufgaben

Übungsaufgabe 5.1 Betrachte eine Skip List für eine Menge S von n Zahlen. Für p und q in S sei $R(p,q)$ die Anzahl der Elemente von S, die echt zwischen p und q liegen. Sei die Position von p in der untersten Liste L_1 bekannt. Beweise, daß man das Element q in $O(\log R(p,q))$ erwarteter Zeit lokalisieren kann. Warum ist das ein interessantes Ergebnis?

Übungsaufgabe 5.2 Gegeben seien n Kästchen und n Bälle. Wir verteilen die Bälle zufällig und unabhängig über die Kästchen.

1. Sei j mit $1 \leq j \leq n$ fest. Beweise, daß die erwartete Anzahl von Bällen in dem j–ten Kästchen gleich eins ist.

2. Für $1 \leq j \leq n$ sei L_j die Anzahl der Bälle in dem j–ten Kästchen. Wir haben in 1. gesehen, daß L_j eine Zufallsvariable mit Erwartungswert eins ist.
   Sei $L := \max_{1 \leq j \leq n} L_j$. Beweise folgende Aussagen:

   (a) $\Pr(L \geq k) \leq n \cdot \binom{n}{k} \cdot (\frac{1}{n})^k$.

   (b) $E(L) \leq \sum_{k=1}^{\infty} \min(1, \frac{n}{k!})$.

   (c) $E(L) = O(\frac{\log n}{\log \log n})$.

   (Man kann beweisen, daß $E(L) = \Theta(\frac{\log n}{\log \log n})$ ist. Siehe: G.H Gonnet, Journal of the ACM **28**, 1981, pp. 289-304.)

Übungsaufgabe 5.3 Eine Folge $(X_i)_{i\geq 1}$ von Zufallsvariablen heißt *paarweise unabhängig*, falls für alle $1 \leq i < j$ und alle r und s gilt:

$$\Pr(X_i = r \wedge X_j = s) \;=\; \Pr(X_i = r) \cdot \Pr(X_j = s).$$

Die Folge heißt *gegenseitig unabhängig*, falls für alle $n \geq 2$, $1 \leq i_1 < i_2 < \ldots < i_n$ und $r_1, r_2, \ldots, r_n$ gilt:

$$\Pr(\bigwedge_{j=1}^{n} (X_{i_j} = r_j)) \;=\; \prod_{j=1}^{n} \Pr(X_{i_j} = r_j).$$

Betrachte den folgenden Ereignisraum:

$$U = \{(123), (132), (213), (231), (312), (321), (111), (222), (333)\}.$$

Wir ziehen zufällig ein Element u aus U. Für $i = 1, 2, 3$ sei X_i die Zahl an der i–ten Stelle von u. (Wenn z.B. $u = (312)$ ist, dann ist $X_3 = 2$.) Sei N die Zufallsvariable, die gleich X_2 ist. Beweise folgende fünf Aussagen:

1. $\forall\, i, r$: $1 \leq i \leq 3$, $1 \leq r \leq 3$ gilt: $\Pr(X_i = r) = \frac{1}{3}$.

2. X_1, X_2 und X_3 sind paarweise unabhängig.

3. X_1, X_2 und X_3 sind *nicht* gegenseitig unabhängig.

4. $\sum_{i=1}^{E(N)} E(X_i) = 4$.

5. $E(\sum_{i=1}^{N} X_i) \neq \sum_{i=1}^{E(N)} E(X_i)$.

Übungsaufgabe 5.4 Sei $(X_i)_{i\geq 1}$ eine Folge von gleichverteilten Zufallsvariablen. Jedes X_i nimmt nur Werte aus $\{0, 1, 2, 3, \ldots\}$ an. Sei N eine Zufallsvariable, die auch nur Werte aus $\{0, 1, 2, 3, \ldots\}$ annimmt. Nehme an, daß die Variablen N und X_i, $i \geq 1$ gegenseitig unabhängig sind. Beweise, daß

$$E(\sum_{i=1}^{N} X_i) \;=\; E(N) \cdot E(X_1).$$

Übungsaufgabe 5.5 Sei S eine Menge von n reellen Zahlen und SL eine Skip List für S. Für $x \in \mathbb{R}$ sei $T(x)$ die Zeit, die wir brauchen, um in SL nach x zu suchen. Wir haben in der Vorlesung bewiesen, daß der Erwartungswert von $T(x)$ durch $O(\log n)$ beschränkt ist. Sei nun $T = \max\{T(x) : \; x \in \mathbb{R}\}$. Beweise, daß der Erwartungswert von T auch durch $O(\log n)$ beschränkt ist.

Übungsaufgabe 5.6 Seien k und n ganze Zahlen, so daß $2 \leq k \leq n$. Ein Baum T heißt *UF(k)-Baum* falls

1. die Wurzel von T höchstens k Kinder hat,

2. jeder Knoten in T entweder 0 oder mehr als k Enkel hat,

3. alle Blätter von T die gleiche Tiefe haben.

Löse das Union-Find–Problem mit Hilfe von UF(k)-Bäumen, so daß jede Union-Operation Zeit $O(k + \log_k n)$ und jede Find-Operation Zeit $O(\log_k n)$ kostet.
Ändere die Datenstruktur, so daß jede Union–Operation Zeit $O(k)$ und jede Find–Operation immer noch Zeit $O(\log_k n)$ braucht.
Für welchen Wert von k sind diese Zeiten minimal ?

Übungsaufgabe 5.7 Sei S eine Menge von n Punkten in der Ebene. Beweise, wie man mit Hilfe von Range Trees sogenannte *orthogonal range queries* effizient lösen kann. Eine solche Query besteht aus einem Rechteck $R = [a_1 : b_1] \times [a_2 : b_2]$ und der Aufgabe alle Punkte von S zu suchen, die in R liegen, d.h. alle Punkte $p = (p_1, p_2) \in S$, so daß $a_1 \leq p_1 \leq b_1$ und $a_2 \leq p_2 \leq b_2$.

Übungsaufgabe 5.8 In dieser Aufgabe betrachten wir eine Variante von Range Trees. Sei S ein Menge von n Punkten in der Ebene und sei m ein Parameter, so daß $1 \leq m \leq \log n$.

Ein *Range Tree mit slack Parameter* m hat eine ähnliche Struktur wie ein Range Tree. Der Unterschied ist, daß nur Knoten auf den Stufen $0, m, 2m, 3m, \ldots$ des x–Baums Zeiger auf y–Bäume enthalten.

Beweise, wie man mit Hilfe dieser Datenstruktur orthogonal range queries in Zeit $O(\frac{2^m}{m} \cdot (\log n)^2 + k)$ lösen kann, wobei k die Anzahl der Punkte in dem Queryrechteck ist. Beweise, daß diese Datenstruktur Größe $O(\frac{1}{m} \cdot n \log n)$ hat.

Übungsaufgabe 5.9 Wir betrachten außer dem XY–System auch das $X'Y'$–System, das durch eine Drehung um $\frac{\pi}{4}$ entsteht. Sei S eine Menge von Punkten in der Ebene, und sei $p \in \mathbb{R}^2$. Seien weiter p^* der L_2–Nachbar von p in S, q der L_∞–Nachbar von p in S bezüglich des XY–Systems, und q' der L_∞–Nachbar von p in S bezüglich des $X'Y'$–Systems.

Sei q_0 der Punkt aus $\{q, q'\}$, der den kleinsten L_2–Abstand zu p hat. In der Vorlesung wurde gezeigt,daß $d_2(p, q_0) \leq (1 + \frac{\pi}{4}) \cdot d_2(p, p^*)$ (Nimm $\epsilon = \frac{\pi}{4}$ in Abschnitt 3.1). Diese Abschätzung ist sehr grob, da auch gezeigt wurde, daß $d_2(p, q_0) \leq \sqrt{2} \cdot d_2(p, p^*)$. Beweise, daß $d_2(p, q_0) \leq \sqrt{1 + \frac{1}{2}\sqrt{2}} \cdot d_2(p, p^*)$.

Übungsaufgabe 5.10 Sei S eine Menge von n Punkten in der Ebene, $k > 4$ eine Konstante und $\Theta = \frac{2\pi}{k}$. Für $1 \leq i \leq k$, sei l_i die Halbgerade, die entsteht, wenn die positive X-Achse um $i \cdot \Theta$ gedreht wird. Sei C_i der Winkelausschnitt bestehend aus allen Punkten der Ebene, die zwischen l_{i-1} und l_i liegen. Für einen Punkt p in der Ebene sei $C_i^p := C_i + p := \{c + p : c \in C_i\}$.

1. Sei $i, 1 \leq i \leq k$, fest. Gib eine Datenstruktur für S an, die folgende Queries effizient löst:
   Gegeben ein Punkt p in der Ebene, finde den Punkt $p^{(i)}$ in $C_i^p \cap S$, dessen Projektion auf l_{i-1} den kleinsten Abstand zu p hat. (Also, wenn $i = 1$ und p der Ursprung, dann suchen wir den Punkt in $C_1^p \cap S$ mit minimaler x–Koordinate.)

2. Sei $p_*^{(i)}$ der Punkt in $C_i^p \cap S$ mit kleinstem Abstand zu p. Beweise, daß
$$d(p, p^{(i)}) \leq \frac{1}{\cos \Theta} \cdot d(p, p_*^{(i)}).$$

3. Löse das $(\frac{1}{\cos \Theta})$-approximative Post-Office Problem. Für jedes $\epsilon > 0$ gibt es ein k, so daß $\frac{1}{\cos \Theta} \leq 1 + \epsilon$ für $\Theta = \frac{2\pi}{k}$. Dies liefert also eine alternative Lösung für das $(1 + \epsilon)$-approximative Post-Office Problem.

Übungsaufgabe 5.11 Wir benutzen die Bezeichnungen aus Aufgabe 10. Sei $t > 1$ eine Konstante. Ein Graph G mit Knotenmenge S heißt ein *t-spanner*, falls es für jedes Paar $p, q \in S$ einen Pfad in G von p nach q gibt, mit Länge höchstens $t \cdot d(p,q)$. (Die Länge eines Pfades ist die Gesamtlänge der Kanten des Pfades.)

1. Warum enthält ein spanner mindestens $n-1$ Kanten?
   Für $k > 8$ sei G_k der Graph mit Knotenmenge S und Kantenmenge
   $$\{(p, p^{(i)}) : p \in S, 1 \leq i \leq k\}.$$
   Wir nehmen an, daß k eine Konstante ist. Dann enthält G_k $O(n)$ Kanten.

2. Seien p und q Punkte in S, so daß (p,q) keine Kante in G_k ist. Sei i der Index, so daß $q \in C_i^p$. Betrachte den Punkt $p^{(i)}$ und beweise, daß
   $$d(p^{(i)}, q) \leq d(p,q) + d(p, p^{(i)}) \cdot (\sin\Theta - \cos\Theta).$$
   Da $\Theta < \frac{\pi}{4}$, gilt insbesondere $d(p^{(i)}, q) < d(p,q)$.

3. Beweise, daß G_k zusammenhängend ist.

4. Sei $t = \frac{1}{\cos\Theta - \sin\Theta}$. Beweise, daß G_k ein t-spanner ist. Für jedes $\epsilon > 0$ gibt es ein k, so daß $\frac{1}{\cos\Theta - \sin\Theta} \leq 1 + \epsilon$ für $\Theta = \frac{2\pi}{k}$. Dies liefert also einen $(1+\epsilon)$-spanner mit nur $O(n)$ Knoten.

Bibliography

[1] C.R. Aragon and R.G. Seidel. *Randomized search trees.* Proc. 30th Annual IEEE Symp. on Foundations of Computer Science, 1989, pp. 540–545.

[2] S. Arya, D.M. Mount, N.S. Netanyahu, R. Silverman, A. Wu. *An optimal algorithm for approximate nearest neighbor searching.* Proc. 5th Annual ACM-SIAM Symp. on Discrete Algorithms, 1994, pp. 573–582.

[3] N. Blum. *On the single-operation worst-case time complexity of the disjoint set union problem.* SIAM J. Comput. **15** (1986), pp. 1021–1024.

[4] T.H. Cormen, C.E. Leiserson and R.L. Rivest. *Introduction to Algorithms.* MIT Press and McGraw-Hill, 1990.

[5] P.F. Dietz and D.D. Sleator. *Two algorithms for maintaining order in a linked list.* Proc. 19th Annual ACM Symp. Theory of Computing, 1987, pp. 365–372.

[6] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer-Verlag, Berlin, 1987.

[7] W. Feller. *An Introduction to Probability Theory and Its Applications*, Volume I, 3rd Edition. John Wiley & Sons, 1968.

[8] Z. Galil and G.F. Italiano. *Data structures and algorithms for disjoint set union problems.* ACM Computing Surveys **23** (1991), pp. 319–344.

[9] S. Kapoor and M. Smid. *New techniques for exact and approximate closest-point problems.* Proc. 10th Annual ACM Symp. on Computational Geometry, 1994, pp. 165–174.

[10] J.M. Keil and C.A. Gutwin. *Classes of graphs which approximate the complete Euclidean graph.* Discrete & Computational Geometry **7** (1992), pp. 13–28.

[11] D.C. Kozen. *The Design and Analysis of Algorithms.* Springer-Verlag, New York, 1992.

[12] J.A. La Poutré. *Lower bounds for the union-find and the split-find problem on pointer machines.* Proc. 22nd Annual ACM Symp. Theory of Computing, 1990, pp. 34–44.

[13] G.S. Lueker. *A data structure for orthogonal range queries.* Proc. 19th Annual IEEE Symp. Foundations of Computer Science, 1978, pp. 28–34.

[14] K. Mehlhorn. *Data Structures and Algorithms, Volume 3: Multi-Dimensional Searching and Computational Geometry.* Springer-Verlag, Berlin, 1984.

[15] K. Mehlhorn and S. Näher. *Dynamic fractional cascading.* Algorithmica **5** (1990), pp. 215–241.

[16] K. Mulmuley. *Computational Geometry, an Introduction through Randomized Algorithms.* Prentice Hall, Englewood Cliffs, 1994.

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

[18] F.P. Preparata and M.I. Shamos. *Computational Geometry, an Introduction.* Springer-Verlag, New York, 1985.

[19] W. Pugh. *Skip lists: a probabilistic alternative to balanced trees.* Proc. WADS, Lecture Notes in Computer Science, Vol. 382, Springer-Verlag, Berlin, 1989. pp. 437–449.

[20] W. Pugh. *A skip list cookbook.* Technical Report, University of Maryland, 1989.

[21] W. Pugh. *Skip lists: a probabilistic alternative to balanced trees.* Commun. of the ACM **33** (1990), pp. 668–676.

[22] J. Ruppert and R. Seidel. *Approximating the d-dimensional complete Euclidean graph.* Proc. 3rd Canadadian Conf. on Computational Geometry, 1991, pp. 207–210.

[23] J.S. Salowe. *On Euclidean spanner graphs with small degree.* Proc. 8th Annual ACM Symp. on Computational Geometry, 1992, pp. 186–191.

[24] M. Smid. *A data structure for the union-find problem having good single-operation complexity.* Algorithms Review **1** (1990), pp. 1–11.

[25] M. Smid. *Range trees with slack parameter.* Algorithms Review **2** (1991), pp. 77–87.

[26] R.E. Tarjan. *Efficiency of a good but not linear set union algorithm.* J. of the ACM **22** (1975), pp. 215–225.

[27] R.E. Tarjan. *A class of algorithms which require nonlinear time to maintain disjoint sets.* J. Comput. System Sci. **18** (1979), pp. 110–127.

[28] R.E. Tarjan and J. van Leeuwen. *Worst-case analysis of set union algorithms.* J. of the ACM **31** (1984), pp. 245–281.