Chapter 7. Multidimensional Data Structures

Chapter 3 was devoted to searching problems in one-dimensional space. In this chapter we will reconsider these problems in higher dimensional space and also treat a number of problems which only become interesting in higher dimensions. Let U be some ordered set and let $S \subseteq U^d$ for some d. An element $x \in S$ is a d-tuple $(x_0, \ldots, x_{d-1})$. The simplest searching problem is to specify a point $y \in U^d$ and to ask whether $y \in S$; this is called an exact match query and can in principle be solved by methods of Chapter 3. Order U^d by lexicographic order and use a balanced search tree. A very general form of query is to specify a region $R \subseteq U^d$ and to ask for all points in $R \cap S$. General region queries can only be solved by exhaustive search of set S. Special and more tractable cases are obtained by restricting the query region R to some subclass of regions. Restricting R to polygons gives us polygon searching, restricting it further to rectangles with sides parallel to the axis gives us range searching, and finally restricting the class of rectangles even further gives us partial match retrieval. In one-dimensional space balanced trees solve all these problems efficiently. In higher dimensions we will need different data structures for different types of queries; d-dimensional trees, range trees and polygon trees are therefore treated in 7.2. There is one other major difference to one-dimensional space. It seems to be very difficult to deal with insertions and deletions; i.e., the data structures described in 7.2 are mainly useful for static sets. No efficient algorithms are known as of today to balance these structures after insertions and deletions. However, there is a general approach to dynamization which we treat in 7.1. It is applicable to a wide class of problems and yields reasonably efficient dynamic data structures.

In Section 7.2.3 we discuss lower bounds. We will first prove a lower bound on the complexity of partial match retrieval where no redundancy in storage space is allowed. The lower bound implies the optimality of d-dimensional trees. The second lower bound relates the complexity of insertions, deletions and queries with a combinatorial quantity. The spanning bound implies the optimality of range trees and near-optimality of polygon trees.

Multidimensional searching problems appear in numerous applications, most notably database systems. In these applications U is an arbitrary ordered set, e.g., a set of names or a set of possible incomes. Region queries arise in these applications in a natural way; e.g., in a database containing information about persons, say name, income and number of children, we might ask for all persons with

# children = 3	, a partial match query;
# children = 3, 1000 $\leq$income $\leq$ 2000	, a range query;
income = 1000 + 1000 $\cdot$ #children	, a polygon query.

2

7.1. A Black Box Approach to Data Structures

In Chapter 3 we designed clever data structures for variants of the dictionary problem. With a little bit of unfairness one might say that all we did in Chapter 3 is the following: We started with binary search on sorted arrays and generalized it in two directions. First we generalized to weighted static trees in order to cope with insertions and deletions. Finally we combined both extensions and arrived at weighted dynamic trees. Suppose now that we want to repeat the generalization process for a different data structure, say interpolation search. Do we have to start all over again or can we profit from the development of Chapter 3? In this section we will describe some general techniques for generalization: dynamization and weighting. We start out with a static solution for some searching problem, i.e., a solution which only supports queries, but does support neither insertions and deletions nor weighted data. Then dynamization is a method which allows us to also support insertions and deletions, weighting is a method which allows us to support queries to weighted data and finally weighted dynamization combines both extensions. Of course, we cannot hope to arrive at the very best data structure by only applying general principles. Nevertheless, the general principles can give us very quickly fully dynamic solutions with reasonable running time. Also there are data structures, e.g., d-dimensional trees, where all special purpose attempts of dynamization have failed.

Binary search on sorted arrays will be our running example. Given a set of n elements one can construct a sorted array in time $O(n \log n)$ (preprocessing time is $O(n \log n)$), we can search the array in time $O(\log n)$ (query time is $O(\log n)$), and the array consumes space $O(n)$ (space requirement is $O(n)$). Dynamization produces a solution for the dictionary problem (operations Insert, Delete, Member) with running time $O(\log n)$ for Inserts and Deletes and $O(\log^2 n)$ for Member. Thus Inserts and Deletes are as fast as in balanced trees but queries are less efficient. Weighting produces weighted static dictionaries with access time $O(\log 1/p)$ for an element of probability p. This is the same order of magnitude as the special purpose solution of Section 3.4, the factor of proportionality is much larger though. Finally weighted dynamization produces a solution for the weighted, dynamic dictionary problem with running time $O((\log 1/p)^2)$ for Member operations and running time $O(\log 1/p)$ for Insert, Delete, Promote and Demote operations. Note that only access time is worse than what we obtained by dynamic weighted trees in 3.6.

Although sorted arrays are our main example, they are not an important application of our general principles. The most important applications are data structures for higher dimensional searching problems described in this chapter. In many of these cases only static solutions are known and all attempts to construct dynamic or weighted solutions by special purpose methods have failed so far. The only dynamic or weighted solutions known today are obtained by applying the general principles described in this section.

7.1.1. Dynamization

We start with a definition of searching problem.

Definition: Let T_1, T_2, T_3 be sets. A searching problem Q of type T_1, T_2, T_3 is a function $Q : T_1 \times 2^{T_2} \to T_3$. ∎

A searching problem takes a point in T_1 and a subset of T_2 and produces an answer in T_3. There are plenty of examples. In the member problem we have $T_1 = T_2$, $T_3 = \{\text{true}, \text{false}\}$ and $Q(x, S) =$ "$x \in S$". In the nearest neighbor problem in the plane we have $T_1 = T_2 = \mathbb{R}^2$, $T_3 = \mathbb{R}$ and $Q(x, S) = \delta(x, y)$, where $y \in S$ and $\delta(x, y) \leq \delta(x, z)$ for all $z \in S$. Here δ is some metric. In the inside the convex hull problem we have $T_1 = T_2 = \mathbb{R}^2$, $T_3 = \{\text{true}, \text{false}\}$ and $Q(x, S) =$ "is x inside the convex hull of point set S". In fact, our definition of searching problem is so general that just about everything is a searching problem.

A static data structure S for a searching problem supports only query operation Q, i.e., for every $S \subseteq T_2$ one can build a static data structure S such that function $Q(x, S) : T_1 \to T_3$ can be computed efficiently. We deliberately use the same name for set S and data structure S because the internal workings of structure S are of no concern in this section. We associate three measures of efficiency with structure S, query time Q_S, preprocessing time P_S and space requirement S_S.

$Q_S(n) =$ time for a query on a set of n points using data structure S.

$P_S(n) =$ time to build S for a set of n points.

$S_S(n) =$ space requirement of S for a set of n points.

We assume throughout that $Q_S(n)$, $P_S(n)/n$ and $S_S(n)/n$ are nondecreasing.

A semi-dynamic data structure D for a searching problem supports in addition operation Insert, i.e., we cannot only query D but also insert new points into D. A dynamic structure supports Insert and Delete. We use the following notation for the resource requirements of D.

$Q_D(n) =$ time for a query on a set of n points using structure D.

$S_D(n) =$ space requirement of D for a set of n points.

$I_D(n) =$ time for inserting a new point into a set of n points stored in D.

$\bar{I}_D(n) =$ amortized time for n-th insertion, i.e., (maximal total time spent on executing insertions in any sequence of n operations starting with the empty set)$/n$.

$D_D(n) =$ time for deleting a point from a set of n points stored in D.

$\bar{D}_D(n) =$ amortized time for n-th deletion, i.e., (maximal total time spent on executing deletions in any sequence of n operations (insertions, deletions, queries) starting with the empty set)$/n$.

We will next describe a general method for turning static data structures into semi-dynamic data structures. This method is only applicable to a subclass of searching problems, the decomposable searching problems.

Definition: A searching problem Q of type T_1, T_2, T_3 is **decomposable** if there is a binary operation $\sqcup : T_3 \times T_3 \to T_3$ such that for all $S \subseteq T_2$ and all partitions A, B of S, i.e., $S = A \cup B$, $A \cap B = \emptyset$, and all $x \in T_1$:

$$Q(x, S) = \sqcup(Q(x, A), Q(x, B)).$$

Moreover, $\sqcup$ is computable in constant time. ∎

In decomposable searching problems we can put together the answer to a query with respect to set S from the answers with respect to pieces A and B of S using operator $\sqcup$. We note as a consequence of the definition of decomposability that T_3 with operation $\sqcup$ is basically a commutative semigroup with unit element $Q(x, \emptyset)$. The member problem is decomposable with $\sqcup = \text{or}$, the nearest neighbor problem is decomposable with $\sqcup = \min$. However, the inside the convex hull problem is not decomposable.

Theorem 1. *Let S be a static data structure for a decomposable searching problem Q. Then there is a semi-dynamic solution D for Q with*

$$\begin{aligned} Q_D(n) &= O(Q_S(n) \cdot \log n), \\ S_D(n) &= O(S_S(n)), \\ \bar{I}_D(n) &= O((P_S(n)/n) \cdot \log n). \end{aligned}$$

Proof: The proof is based on a simple yet powerful idea. At any point of time the dynamic structure consists of a collection of static data structures for parts of S, i.e., set S is partioned into blocks S_i. Queries are answered by querying the blocks and composing the partial answers by $\sqcup$. Insertions are dealt with by suitably combining blocks.

The details are as follows. Let S be any set of n elements and let $n = \sum_{i=0} a_i 2^i$, $\sum_{i=0}^{???}$ $a_i \in \{0, 1\}$, be the binary representation of n. Let $S_0, S_1, \ldots$ be any partition of S with $|S_i| = a_i 2^i$, $0 \le i \le \log n$. Then structure D is just a collection of static data structures, one for each non-empty S_i.

The space requirement of D is easily computed as

$$\begin{aligned} S_D(n) &= \sum_i S_S(a_i 2^i) = \sum_i (S_S(a_i 2^i)/a_i 2^i) \cdot a_i 2^i \\ &\le \sum_i (S_S(n)/n) \cdot a_i 2^i = S_S(n). \end{aligned}$$

The inequality follows from our basic assumption that $S_S(n)/n$ is nondecreasing.

Next note that $Q(x,S) = \bigsqcup_{0\le i\le \log n} Q(x,S_i)$ and that there are never more than $\log n$ non-empty S_i's. Hence a query can be answered in time

$$\log n + \sum_i Q_S(a_i 2^i) \le \log n \cdot (1 + Q_S(n)) = O(\log n \cdot Q_S(n)).$$

Finally consider operation Insert(x,S). Let $n+1 = \sum \beta_i 2^i$ and let j be such that $\alpha_j = 0$, $\alpha_{j-1} = \alpha_{j-2} = \ldots = \alpha_0 = 1$. Then $\beta_j = 1$, $\beta_{j-1} = \ldots = \beta_0 = 0$. We process the $(n+1)$-th insertion by taking the new point x and the $2^j - 1 = \sum_{i=0}^{j-1} 2^i$ points stored in structures $S_0, S_1, \ldots, S_{j-1}$ and constructing a new static data structure for $\{x\} \cup S_0 \cup S_1 \cup \cdots \cup S_{j-1}$. Thus the cost of the $(n+1)$-st insertion is $P_S(2^j)$. Next note that a cost of $P_S(2^j)$ has to be paid after insertions $2^j \cdot (2\cdot l + 1)$, $l = 0, 1, 2, \ldots$, and hence at most $n/2^j$ times during the first n insertions. Thus the total cost of the first n insertions is bounded by

$$\sum_{j=0}^{\lfloor \log n \rfloor} P_S(2^j) \cdot n/2^j \le n \cdot \sum_{j=0}^{\lfloor \log n \rfloor} P_S(n)/n \le P_S(n) \cdot (\lfloor \log n \rfloor + 1).$$

Hence $\bar{I}_D(n) = O((P_S(n)/n) \cdot \log n)$. ∎

Let us apply Theorem 1 to binary search on sorted arrays. We have $S_S(n) = n$, $Q_S(n) = \log n$ and $P_S(n) = n \log n$. Hence we obtain a semi-dynamic solution for the member problem with $S_D(n) = O(n)$, $Q_D(n) = (\log n)^2$ and $\bar{I}_D(n) = (\log n)^2$. Actually, the bound on $\bar{I}_D(n)$ is overlay pessimistic. Note that we can merge two sorted arrays in linear time. Hence we can construct a sorted array of size 2^k out of a point and sorted arrays of size $1, 2, 4, 8, \ldots, 2^{k-1}$ in time $O(2^k)$ by first merging the two arrays of length 1, obtaining an array of length 2, merging it with the array of length 2, ... Plugging this bound into the bound on $\bar{I}_D(n)$ derived above yields $\bar{I}_D(n) = O(\log n)$.

There are other situations where the bounds stated in Theorem 1 are overlay pessimistic. If either $Q_S(n)$ or $P_S(n)/n$ grow fast, i.e., is of order at least n^ϵ for some $\epsilon > 0$, then better bounds hold. Suppose for example that $Q_S(n) = \Theta(n^\epsilon)$ for some $\epsilon > 0$. Then (cf. the proof of Theorem 1)

$$\begin{aligned} Q_D(n) &= \sum_i Q_S(a_i 2^i) \\ &= Q_S(2^{\lfloor \log n \rfloor}) \cdot \sum_{i=0}^{\lfloor \log n \rfloor} Q_S(a_i 2^i)/Q_S(2^{\lfloor \log n \rfloor}) \\ &= \Theta\left(n^\epsilon \cdot \sum_{i=0}^{\lfloor \log n \rfloor} 2^{(i - \lfloor \log n \rfloor)\cdot \epsilon} \right) = \Theta(n^\epsilon) = \Theta(Q_S(n)). \end{aligned}$$

Thus if either $Q_S(n)$ or $P_S(n)/n$ grows fast then the $\log n$ factor in the corresponding bound on $Q_D(n)$ or $\bar{I}_D(n)$ can be dropped.

The bound on insertion time derived in Theorem 1 is amortized. In fact, the time required to process insertions fluctuates widely. More precisely, the 2^k-th insertion takes time $P_S(2^k)$, a non-trivial amount of time indeed. Theorem 2 shows that we can turn the amortized time bound into a worst case time bound without increasing the order of query time and space requirement.

Theorem 2. *Let S be a static data structure for a decomposable searching problem. Then there is a semi-dynamic data structure D with*

$$\begin{aligned} Q_D(n) &= O(Q_S(n) \cdot \log n), \\ S_D(n) &= O(S_S(n)), \\ I_D(n) &= O(P_S(n)/n \cdot \log n). \end{aligned}$$

Proof: The basic idea is to use the construction of Theorem 1, but to spread work over time. More precisely, whenever a structure of size 2^k has to be constructed we will spread the work over the next 2^k insertions. This will have two consequences. First, the structure will be ready in time to process an overflow into a structure of size 2^{k+1} and second, the time required to process a single insertion is bounded by

$$\sum_{k=0}^{\lfloor \log n \rfloor} P(2^k)/2^k = O(P(n)/n \cdot \log n).$$

The details are as follows. The dynamic structure D consists of bags BA_0, BA_1, ... Each bag BA_i contains at most three blocks $B_i^u[1]$, $B_i^u[2]$ and $B_i^u[3]$ of size 2^i that are "in use" and at most one block B_i^c of size 2^i that is "under construction". More precisely, at any point of time blocks $B_i^u[j]$, $i \geq 0$, $1 \leq j \leq 3$, form a partition of set S, and static data structures are available for them. Furthermore, the static data structure for block B_i^c is under construction. Block B_i^c is the union of two blocks $B_{i-1}^u[j]$. We proceed as follows. As soon as two B_i^u's are available, we start building a B_{i+1}^c of size 2^{i+1} out of them. The work is spread over the next 2^{i+1} insertions, each time doing $P_S(2^{i+1})/2^{i+1}$ steps of the construction. When B_{i+1}^c is finished it becomes a B_{i+1}^u and the two B_i^u's are discarded. We have to show that there will be never more than three non-empty B_i^u's.

Lemma 1. *When we complete a B_i^c and turn it into a block in use there are at most two non-empty B_i^u's.*

Proof: Consider how blocks in BA_i develop. Consider the moment, say after the t-th insertion, when BA_i contains two B_i^u's and we start building a B_{i+1}^c out of them. The construction will be finished 2^{i+1} insertions later. Observe that BA_i got

a second B_i^u because the construction of B_i^c was completed after the t-th insertion and hence B_i^c was turned into a B_i^u. Thus B_i^c was empty after insertion t and it will take exactly 2^i insertions until it is full again and hence gives rise to a third B_i^u and it will take another 2^i insertions until it gives rise to a fourth B_i^u. Exactly at this point of time the construction of B_{i+1}^u is completed and hence two B_i^u's are discarded. Thus we can start a new cycle with just tow B_i^u's completed. ∎

It follows from Lemma 1 that there will be never more than three B_i^u's and one B_i^c for any i. Hence

$$\begin{aligned} S_D(n) &= O(S_S(n)), \\ Q_D(n) &= O(Q_S(n) \cdot \log n) \quad \text{and} \\ I_D(n) &= \sum_{i=0}^{\lfloor \log n \rfloor} O(P_S(2^i)/2^i) = O(P_S(n)/n \cdot \log n). \end{aligned}$$ ∎

The remarks following Theorem 1 also apply to Theorem 2. The "logarithmic" dynamization method described above has a large similarity to the binary number system. The actions following the insertion of a point into a dynamic structure of n elements are in complete analogy to adding a 1 to integer n written in binary. The main difference is the cost of processing a carry. The cost is $P(2^k)$ for processing a carry from the k-th position in logarithmic dynamization, whilst it is $O(1)$ in processing integers. The analogy between logarithmic dynamization and the binary number system suggests that other number systems give rise to other dynamization methods. This is indeed the case. For example, for every k one can uniquely write every integer n as

$$n = \sum_{i=1}^{k} \binom{a_i}{i}$$

with $i - 1 \leq a_i$ and $a_1 < a_2 < \cdots < a_k$ (Exercise 1). This representation gives rise to k-binomial transformation. We represent a set S of n elements by k static structures, the i-th structure holding $\binom{a_i}{i}$ elements. Then $Q_D(n) = O(Q_S(n) \cdot k)$ and $\bar{I}_D(n) = O(k \cdot n^{1/k} \cdot P_S(n)/n)$ (Exercise 1). More generally we have

Theorem 3. *Let S be any static data structure for a decomposable searching problem and let $k : \mathbb{N} \to \mathbb{N}$ be any "smooth" function. Then there is a semi-dynamic data structure D such that*

a) *if $k(n) = O(\log n)$ then*

$$\begin{aligned} Q_D(n) &= O(k(n) \cdot Q_S(n)), \\ \bar{I}_D(n) &= O(k(n) \cdot n^{1/k(n)} \cdot P_S(n)/n). \end{aligned}$$

b) *if $k(n) = \Omega(\log n)$ then*

$$\begin{aligned} Q_D(n) &= O(k(n) \cdot Q_S(n)), \\ \bar{I}_D(n) &= O(\log n / \log(k(n)/\log n) \cdot P_S(n)/n). \end{aligned}$$

Proof: The proof can be found in K. Mehlhorn, M.H. Overmars: "Optimal Dynamization of Decomposable Searching Problems", IPL 12 (1981), 93–98. The details on the definition of smoothness can be found there; functions like $\log n$, $\log\log n$, $\log\log\log n$, n, $(\log n)^2$ are smooth in the sense of Theorem 3. The proof is outlined in Exercise 2. ∎

Let us look at some examples. Taking $k(n) = \log n$ gives the logarithmic transformation (note that $n^{1/\log n} = 2$), $k(n) = k$ yields an analogue to the k-binomial transformation, $k(n) = k \cdot n^{1/k}$ yields a transformation with $Q_D(n) = O(k \cdot n^{1/k} \cdot Q_S(n))$ and $\bar{I}_D(n) = O(kP_S(n)/n)$, a dual to the k-binomial transformation, and $k(n) = (\log n)^2$ yields a transformation with $Q_D(n) = O((\log n)^2 \cdot Q_S(n))$ and $\bar{I}_D(n) = O((\log n/\log\log n) \cdot P_S(n)/n)$. Again it is possible to turn amortized time bounds into worst case time bounds by the techniques used in the proof in Theorem 2. The interesting fact about Theorem 3 is that it describes exactly how far we can go by dynamization.

Theorem 4. *Let $h, k : \mathbb{N}$ be functions. If there is a dynamization method which turns every static data structure S for any decomposable searching problem into a dynamic data structure D with $Q_D = k(n \cdot K \cdot Q_S(n))$ and $\bar{I}_D(n) = h(n) \cdot P(n)/n$ then $h(n) = \Omega(OP(k)(n))$ where*

$$OP(k)(n) = \begin{cases} \log n/\log(k(n)/\log n) & \text{,if } k(n) > 2 \cdot \log n; \\ k(n)^{1/k(n)} & \text{,if } k(n) \leq 2 \cdot \log n. \end{cases}$$

Proof: The proof can be found in K. Mehlhorn: "Lower Bounds on the Efficiency of Transforming Static Data Structures into Dynamic Data Structures", Math. Systems Theory 15, 1–16 (1981). ∎

Theorem 4 states that there is no way to considerably improve upon the results of Theorem 3. There is no way to decrease the order of the query penalty factor ($= Q_D(n)/Q_S(n)$) without simultaneously increasing the order of the update penalty factor ($= \bar{I}_D(n) \cdot n/P_S(n)$) and vice versa. Thus all combinations of query and update penalty factor described in Theorem 3 are optimal. Moreover, all optimal transformations can be obtained by an application of Theorem 3.

Turning static into semi-dynamic data structures is completely solved by Theorems 1 to 4. How about deletions? Let us consider the case of the sorted array first. At first sight deletions from sorted arrays are very costly. After all, we might have to shift a large part of the array after a deletion. However, we can do a "weak" deletion very quickly. Just mark the deleted elements and search as usual. As long as only a few, let's say no more than $1/2$ of the elements are deleted, search time is still logarithmic in the number of remaining elements. This leads to the following definition.

Definition: A decomposable searching problem together with its static structure S is deletion decomposable iff, whenever S contains n points, a point can be deleted from S in time $D_S(n)$ without increasing the query time, deletion time and storage required for S. ∎

We assume that $D_S(n)$ is non-decreasing. The Member problem with static structure sorted array is deletion decomposable with $D_S(n) = \log n$, i.e., we can delete an arbitrary number of elements from a sorted array of length n and still keep query and deletion time at $\log n$. Of course, if we delete most elements then $\log n$ may be arbitrarily large as a function of the actual number of elements stored.

Theorem 5. *Let searching problem Q together with static structure S be deletion decomposable. Then there is a dynamic structure D with*

$$\begin{aligned} Q_D(n) &= O(\log n \cdot Q_S(8 \cdot n)), \\ S_D(n) &= O(S_S(8 \cdot n)), \\ \bar{I}_D(n) &= O(\log n \cdot P_S(n)/n), \\ \bar{D}_D(n) &= O(P_S(n)/n + D_S(n) + \log n). \end{aligned}$$

Proof: The proof is a refinement of the construction used in the proof of Theorem 1. Again we represent a set S of n elements by a partition B_0, B_1, B_2, ... We somewhat relax the condition on the size of blocks B_i; namely, a B_i is either empty or $2^{i-3} < |B_i| \leq 2^i$. Here $|B_i|$ denotes the actual number of elements in block B_i. B_i may be stored in a static data structure which was originally constructed for more points but never more than 2^i points. In addition, we store all points of S in a balanced tree T. In this tree we store along with every element a pointer to the block B_i containing the element. This will be useful for deletions. We also link all elements belonging to B_i, $i \geq 0$, in a linear list.

Since $|B_i| \geq 2^{i-3}$ there are never more than $\log n + 3$ non-empty blocks. Also since the structure containing B_i might have been constructed for a set eight times the size we have

$$Q_D(n) \leq Q_S(8 \cdot n) \cdot (\log n + 3) = O(Q_S(8 \cdot n) \log n).$$

Also

$$\begin{aligned} S_D(n) &\leq \sum_i S_S(8 \cdot |B_i|) \\ &= \sum_i S_S(8 \cdot |B_i|)/(8 \cdot |B_i|) \cdot 8 \cdot |B_i| \\ &\leq S_S(8 \cdot n)/8 \cdot n \cdot \sum_i 8 \cdot |B_i| \\ &= S_S(8 \cdot n). \end{aligned}$$

It remains to describe the algorithms for insertion and deletion. We need two definitions first. A non-empty block B_i is deletion-safe if $|B_i| \geq 2^{i-2}$ and it is safe if $2^{i-2} \leq |B_i| \leq 2^{i-1}$.

Insertions are processed as follows. After an insertion of a new point x we find the least k such that $1 + |B_0| + \cdots + |B_k| \leq 2^k$. We build a new static data structure B_k for $\{x\} \cup B_0 \cup \cdots \cup B_k$ in time $P_S(2^k)$ and discard the structure for blocks $B_1, \ldots, B_k$. In addition we have to update the dictionary for a cost of $O(\log n + 2^k)$, $\log n$ for inserting the new point and 2^k for updating the new information associated with the points in the new B_k. Note that time $O(1)$ per element suffices if we chain all elements which belong to the same block in a linked list.

Lemma 2. *Insertions build only deletion-safe structures.*

Proof: This is obvious if $k = 0$. If $k > 0$ then $1 + |B_0| + \cdots + |B_{k-1}| > 2^{k-1}$ by the choice of k and hence the claim follows ∎

The algorithm for deletions is slightly more difficult. In order to delete x we first use the dictionary to locate the block, say B_i, which contains x. This takes time $O(\log n)$. Next we delete x from B_i in time $D_S(2^i)$. If $|B_i| > 2^{i-3}$ or $|B_i|$ is empty after the deletion then we are done. Otherwise, $|B_i| = 2^{i-3}$ and we have to "rebalance". If $|B_{i-1}| > 2^{i-2}$ then we interchange blocks B_i and B_{i-1}. This will cost $O(|B_i| + |B_{i-1}|) = O(2^i)$ steps for changing the dictionary; also B_i and B_{i+1} are safe after the interchange. If $|B_{i-1}| \leq 2^{i-2}$ then we join B_{i-1} and B_i — the resulting set has size at least 2^{i-3} and at most $2^{i-3} + 2^{i-2} \leq 2^{i-1}$ — and construct either a new B_{i-1} (if $|B_{i-1} \cup B_i| < 2^{i-2}$) or a new B_i (if $|B_{i-1} \cup B_i| \geq 2^{i-2}$). This will cost at most $P_S(2^i) + O(2^i) = O(P_S(s^i))$ time units; also B_{i-1} and B_i are safe after the deletion.

Lemma 3. *If a deletion from B_i causes $|B_i| = 2^{i-3}$ then B_{i-1} and B_i are safe after restructuring.*

Proof: Immediate from discussion above. ∎

Lemma 4. $\bar{D}_D(n) = O(P_S(m)/m + D_S(m) + \log m)$, *here m is the maximal size of set S during the first n updates.*

Proof: By Lemmas 2 and 3 only deletion-safe blocks are built. Hence at least 2^{i-3} points have to be deleted from a block B_i before it causes restructuring after a deletion. Hence the cost for restructuring is at most $8 \cdot P_S(m)/m$ per deletion. In addition, $\log m$ time units are required to update the dictionary and $D_S(m)$ time units to actually perform the deletion. ∎

Lemma 5. $\bar{I}_D(n) = 4 \cdot P_S(m)/m \log m$.

Proof: Consider any sequence of n insertions and deletions into an initially empty set. Suppose that we build a new B_k after the t_0-th update operation and that this update is an insertion. Then $B_0, B_1, \ldots, B_{k-1}$ are empty after the t_0-th update. Suppose also that the next time a B_l, $l \geq k$, is constructed after an insertion is after the t_1-th update operation. Then immediately before the t_1-th update $1 + |B_0| + \cdots + |B_{k-1}| > 2^{k-1}$.

We will show that $t_1 - t_0 \geq 2^{k-2}$. Assume otherwise. Then at most $2^{k-2} - 1$ points in $B_0 \cup \cdots \cup B_{k-1}$ are points which were inserted after time t_0. Hence at least 2^{k-2} points must have moved from B_k into $B_0 \cup \cdots \cup B_{k-1}$ by restructuring after a deletion. However, the restructuring algorithm constructs only deletion-safe structures and hence B_k can underflow ($|B_k| = 2^{k-3}$) at most once during a sequence of 2^{k-2} updates. Thus at most $2^{k-3} < 2^{k-2}$ points can move from B_k down to $B_0 \cup \cdots \cup B_{k-1}$ between the t_0-th and the t_1-th update, a contradiction. Thus $t_1 - t_0 \geq 2^{k-2}$.

In particular, a new B_k is constructed at most $n/2^{k-2}$ times after an insertion during the first n updates. The construction of a new B_k has cost $P_S(2^k)$ for building the B_k, $O(2^k)$ for updating the dictionary and $\log m$ for inserting the new point into the dictionary. Hence

$$\begin{aligned}\bar{I}_D(n) &= O\left(\left(\sum_{k=0}^{\log m}(P_S(2^k) \cdot n/2^{k-2} + 2^k \cdot n/2^k) + n \cdot \log m\right)/n\right) \\ &= O(\log m \cdot P_S(m)/m).\end{aligned}$$

∎∎

In our example (binary search on sorted arrays) we have $Q_S(n) = D_S(n) = \log n$ and $P_S(n) = n$ (cf. the remark following Theorem 1). Hence $Q_D(n) = O((\log n)^2)$ and $\bar{I}_D(n) = \bar{D}_D(n) = O(\log n)$. There is something funny happening here. We need balanced trees to dynamize sorted arrays. This is not a serious objection. We could do away with balanced trees if we increase the time bound for deletes to $O((\log n)^2)$. Just use the Member instruction provided by the data structure itself to process a Delete.

Theorem 5 can be generalized in several ways. Firstly, one can turn amortized bounds into worst case bounds and secondly one can choose any of the transformations outlined in Theorem 3. This yields.

Theorem 6. *Let searching problem Q together with static structure S be deletion-decomposable, and let $k(n)$ be any smooth function. Then there is a dynamic structure D with*

$$\begin{aligned}Q_D(n) &= O(k(n) \cdot Q_S(n)), \\ D_D(n) &= O(\log n + P_S(n)/n + D_S(n)), \\ I_D(n) &= \begin{cases} O(\log n/\log(k(n)/\log n) \cdot P_S(n)/n, & \text{,if } k(n) = \Omega(\log n); \\ O(k(n) \cdot n^{1/k(n)} \cdot P_S(n)/n), & \text{,if } k(n) = O(\log n). \end{cases}\end{aligned}$$

Proof: The proof combines all methods described in this section so far. It can be found in M.H. Overmars/J.v. Leeuwen: "Worst Case Optimal Insertion and Deletion Methods for Decomposable Searching Problems, IPL 12 (1981), 168–173. ∎

7.1.2. Weighting and Weighted Dynamization

In this section we describe weighting and then combine it with dynamization described in the previous section. This will give us dynamic weighted data structures for a large class of searching problems.

Definition: A searching problem $Q : T_1 \times 2^{T_2} \rightarrow T_3$ is **monotone decomposable** if there are functions $q : T_3 \rightarrow \{\text{true}, \text{false}\}$ and $\sqcup : T_3 \times T_3 \rightarrow T_3$ such that for all $x \in T_1$, $S \subseteq T_2$ and all partitions A, B of S, i.e., $A \cup B = S$, $A \cap B = \emptyset$:

$$Q(x, S) = \textbf{if } q(Q(x, A)) \textbf{ then } Q(x, A) \textbf{ else } \sqcup (Q(x, A), Q(x, B)) \textbf{ fi}$$ ∎

Again, there are plenty of examples. Member is monotone decomposable with q the identity and $\sqcup =$ or. ϵ-diameter search, i.e., $Q((x, \epsilon), S) = \textbf{true}$ if $\exists y \in S : \delta(x, y) \leq \epsilon$ is monotone decomposable with q the identity and $\sqcup =$ or. Also orthogonal range searching is monotone decomposable. Here $T_2 = \mathbb{R}^2$, $T_1 =$ all rectangles with sides parallel to the axis and $Q(R, S) = (|R \cap S| \geq 1)$.

A query $Q(x, S)$ is **successful** if there is a $y \in S$ such that $q(Q(x, \{y\}))$. If $Q(x, S)$ is successful then any $y \in S$ with $q(Q(x, \{y\}))$ is called a witness for x (with respect to S). If y is a witness for x then $Q(x, S) = Q(x, \{y\} \cup (S - \{y\})) =$ **if** $q(Q(x, \{y\}))$ **then** $Q(x, \{y\})$ **else** $\ldots$ **fi** $= Q(x, \{y\})$.

Weighting is restricted to successful searches (but cf. Exercise 6). Let $S = \{y_1, \ldots, y_n\} \subseteq T_2$ and let μ be a probability distribution on $Suc = \{x \in T_1;\ Q(x, S)$ is successful$\}$. We define a reordering π of S and a discrete probability distribution $p_1, \ldots, p_n$ on S as follows. Suppose that $\pi(1), \ldots, \pi(k-1)$ and $p_1, \ldots, p_{k-1}$, are already defined. For $y_j \in S - \{y_{\pi(1)}, \ldots, y_{\pi(k-1)}\}$ let $p(y_j) = \mu\ \{x \in Suc;\ y_j$ is a witness for x but none of $y_{\pi(1)}, \ldots, y_{\pi(k-1)}$ is$\}$. Define $\pi(k) = j$ such that $p(y_j)$ is maximal and let $p_k = p(y_{\pi(k)})$. Then $p_1 \geq p_2 \geq \cdots \geq p_n$. We assume from now on that S is reordered such that π is the identity. Then p_k is the probability that y_k is witness in a successful search but none of $y_1, \ldots, y_{k-1}$ is.

Theorem 7. *Let Q be any monotone decomposable searching problem and suppose that we have a static data structure with query time $Q_S(n)$, $Q_S(n)$ non-decreasing, for Q. Let $S = \{y_1, \ldots, y_n\} \subseteq T_2$, and let $\mu, p_1, \ldots, p_n$ be defined as above. Then there is a weighted data structure W for Q where the expected time of a successful search is at most*

$$4 \cdot \sum_i p_i \cdot Q_S(i) \leq 4 \cdot p_i \cdot Q_S(1/p_i).$$

Proof: Define $f : \mathbb{N}_0 \to \mathbb{N}_0$ by $f(0) = 0$ and $Q_S(f(i)) = 2^i$ for $i \geq 1$. Then $f(i)$ is increasing. We divide set S into blocks $B_1, B_2, \ldots$, where $B_i = \{y_j;\ f(i-1) < j \leq f(i)\}$. Then W consists of a collection of static data structures, one for each B_i. A query $Q(x, S)$ is answered by the following algorithm.

$i := 0$;
repeat $i := i + 1$ **until** $Q(x, B_i)$ is successful **od**;
output $Q(x, B_i)$.

Program 1

The correctness of this algorithm is immediate from the definition of monotone decomposability. It remains to compute the expected query time of a successful query. Let $Suc_j = \{x \in T_1;\ y_j \text{ is a witness for } x \text{ but } y_1, \ldots, y_{j-1} \text{ are not}\}$. Then $p_j = \mu(Suc_j)$. The cost of a query $Q(x, S)$ for $x \in Suc_j$ and $f(i-1) < j \leq f(i)$ is

$$\begin{aligned}
\sum_{h=1}^{i} Q_S(f(h) - f(h-1)) &\leq \sum_{h=1}^{i} Q_S(f(h)) \\
&= \sum_{h=1}^{i} 2^h \\
&\leq 4 \cdot 2^{i-1} \\
&= 4 \cdot Q_S(f(i-1)) \\
&\leq 4 \cdot Q_S(j).
\end{aligned}$$

Thus the expected cost of a successful query is

$$4 \cdot \sum_j p_j \cdot Q_S(j) \leq 4 \cdot \sum_j p_j \cdot Q_S(1/p_j).$$

The last inequality follows from $p_1 \geq p_2 \geq \cdots$ and hence $1/p_j \geq j$. ∎

It is worthwhile to go through our examples at this point. Let us look at the member problem first. If we use sorted arrays and binary search then $Q_S(n) = \log n$ and hence the expected time for a successful search is

$$4 \cdot \sum_i p_i \cdot \log i \leq 4 \cdot \sum_i p_i \cdot \log 1/p_i.$$

This bound relates quite nicely with the bounds derived in Chapter 3.4 on weighted trees. There we derived a bound of $\sum_i p_i \log 1/p_i + 1$ on the expected search time in weighted trees. Thus the bound derived by weighting is about four times the entire truth. The bound of $4 \cdot \log i$ derived now on individual searches can sometimes be considerably better than the bound of $\log 1/p_i$ derived in 3.4 (cf. Exercise 3).

Binary search is not the only method for searching sorted arrays. If the keys are drawn from a uniform distribution then interpolation search is a method with $O(\log \log n)$ expected query time. If the weights of keys are independent of key values then every block B_i is a random sample drawn from a uniform distribution and hence the expected time of a successful search is

$$O(\sum_i p_i \cdot \log \log i) = O(\sum_i p_i \cdot \log \log 1/p_i)$$

(cf. Exercise 4).

Let us finally look at orthogonal range searching in two-dimensional space. 2-dimensional trees (cf. 7.2.1 below) are a solution with $Q_S(n) = \sqrt{n}$. The weighting yields an expected search time of $\sum_i p_i \sqrt{i}$.

The construction used in the proof of Theorem 7 is optimal among a large class of algorithms, namely all algorithms which divide set S into blocks, construct static data structures for each block, and search through these blocks sequentially in some order independent of the according probability. If an element in the i-th block has higher probability than an element in the $(i-1)$-st block then interchanging the elements will reduce average search time. Thus a search with witness y_i (recall that S is reordered such that $p_1 \geq p_2 \geq \cdots$) must certainly have cost $Q_S(n_1) + \cdots + Q_S(n_k)$ where $n_1 + \cdots + n_k \geq 1$. If we assume that $Q_S(x+y) \leq Q_S(x) + Q_S(y)$ for all x, y, i.e., Q is subadditive, then $Q(n_1) + \cdots Q_S(n_k) \geq Q_S(n_1 + \cdots + n_k) \geq Q_S(i)$. Hence a search with witness y_i has cost at least $Q_S(i)$ under the modest assumption of subadditivity of Q_S. Thus the construction used in the proof of Theorem 7 is optimal because it achieves query time $O(Q_S(i))$ for a search with witness y_i for all i.

We close this section by putting all concepts together. We start with a static data structure for a monotone and deletion decomposable searching problem Q and then use dynamization and weighting to produce a dynamic weighted data structure W for Q. W supports queries on a weighted set S, i.e., a set $S = \{y_1, \ldots, y_n\}$ and weight function $w : S \to \mathbb{N}$ with query time depending on weight. It also supports operations $Promote(y, a)$ and $Demote(y, a)$, $y \in T_2$, $a \in \mathbb{N}$. $Promote(y, a)$ increases the weight of element y by a and $Demote(y, a)$ decreases the weight of y

by a. Insert and Delete are special cases of *Promote* and *Demote* (cf. 3.6 for the special case: Q = Member).

We obtain W in a two step process. In the first step we use dynamization and turn S into a dynamic data structure D with $Q_D(n) = Q_S(n) \cdot \log n$ and $U_D(n) = \max(I_D(n), D_D(n)) = O(P_S(n)/n \log n + D_S(n))$ (cf. Theorem 6). $U_D(n)$ is the time to perform an update (either Insert or Delete) on D. In the second step we use weighting and turn D into a weighted structure W. More precisely, we define f by $Q_D(f(n)) = 2^n$ and store a set $S = \{y_1, \ldots, y_n\}$ by cutting it into blocks as described in Theorem 7, i.e., block B_i contains all y_j with $f(i-1) < j \leq f(i)$. Here we assumed w.l.o.g. that $w(y_1) \geq w(y_2) \geq \cdots \geq w(y_n)$. This suffices to support queries. A query in W with witness $y \in S$ takes time $O(Q_D(w(S)/w(y)))$ by Theorem 7. Here $w(S) = \sum\{w(y); y \in S\}$.

We need to add additional data structures in order to support *Promote* and *Demote*. We store set S in a weighted dynamic tree (cf. 3.6) T. Every element in tree T points to the block B_i containing the element. Furthermore, we keep for every block B_i the weights of the points in B_i in a balanced tree. This allows us to find the smallest and largest weight in a block fast.

We are now in a position to describe a realization of $Promote(y, a)$. We first use the weighted dynamic tree to find the block which contains y. This takes time $O(\log w(S)/w(y))$. Suppose that block B_i contains y. We then run through the following routine.

---

(1) delete y from block B_i; $h := i$;
(2) **while** $w(y) + a >$ maximal weight of any element in B_h
(3) **do** delete the element with maximal weight from B_{h-1} and insert it into B_h;
(4) $h \leftarrow h - 1$;
od
(5) insert y into B_h

Program 2

The algorithm above is quite simple. If the weight of y is increased it might have to move to a block with smaller index. We make room by deleting it from the old block and moving the element with minimal weight down one block for every block. We obtain the following time bound for $Promote(y, a)$:

$$O\left[\log(w(S)/w(y)) + \sum_{h=0}^{i} \log(f(h)) + U_D(f(h))\right].$$

Here $\log w(S)/w(y)$ is the cost of searching for y in tree T and $\log f(h) + U(S(h))$ is the cost of inserting and deleting an element from a structure of size h and updating the balanced tree which holds the weights. Observing $U(n) \geq \log n$ for all n, $i \leq \lceil f^{-1}(w(S)/w(y)) \rceil$ and $U_D(\lceil f^{-1}(w(S)/w(y)) \rceil) \geq U_D(w(S)/w(y)) \geq$

$\log w(S)/w(y)$ this bound simplifies to

$$O\Big(\sum_{0\le h\le\lceil f^{-1}(w(S)/w(y))\rceil} U_D(f(h))\Big).$$

The algorithm for $Demote(y,a)$ is completely symmetric. The details are left for the reader (Exercise 5). We obtain exactly the same running time as for *Promote*, except for the fact that $w(y)$ has to be replaced by the new weight $w(y)-a$.

Theorem 8. *A static data structure with query time Q_S, preprocessing time P_S, and weak deletion time D_S for a monotone deletion decomposable searching problem can be extended to a dynamic weighted data structure W such that:*

a) *A query in weighted set S (with weight function $w: S\to\mathbb{N}$) with witness $y\in S$ takes time $O(Q_D(w(S)/w(y)))$. Here $Q_D(n)=Q_S(n)\cdot\log n$.*

b) *$Promote(y,a)$ takes time*

$$O\Big(\sum_{0\le h\le\lceil f^{-1}(w(S)/w(y))\rceil} U_D(f(h))\Big).$$

c) *$Demote(y,a)$ takes time*

$$O\Big(\sum_{0\le h\le\lceil f^{-1}(w(S)/(w(y)-a))\rceil} U_D(f(h))\Big).$$

Proof: Immediate from the discussion above. ∎

Let us look again at binary search in sorted arrays as a static data structure for Member. Then $Q_S(n)=D_S(n)=O(\log n)$ and $P_S(n)=O(n)$ (cf. the remark following Theorem 1). Hence $Q_D(n)=O((\log n)^2)$, $U_D(n)=O(\log n)$, and $f(n)=2^{\sqrt{2n}}$. A query for y with weight $w(y)$ takes time $O((\log w(S)/w(y)^2)$, the square of the search time in weighted dynamic trees. Also $Promote(y,a)$ takes time $U_D(f(\lceil f^{-1}(w(S)/w(y))\rceil))=O(\log w(S)/w(y))$ and $Demote(y,a)$ takes time $O(\log w(S)/w(y)-a))$. This is the same order as in weighted dynamic trees. Of course, weighted dynamic trees are part of the data structure W considered here. Again (cf. Theorem 5) this is not a serious objection. Since Member is the query considered here we can replace the use of weighted dynamic trees by a use of the data structure itself. This will square the time bounds for *Promote* and *Demote*. Also binary search in sorted arrays is not a very important application of weighted dynamization. In the important applications in this chapter the use of a weighted, dynamic dictionary is negligible with respect to the complexity of the data structure itself.

Dynamization and weighting are powerful techniques. They provide reasonably efficient dynamic weighted data structures very quickly which can then be used as

a reference point for more special developments. Tuning to the special case under consideration is always necessary, as weighting and dynamization tend to produce somewhat clumsy solutions if applied blindly.

7.1.3. Order Decomposable Problems

In Sections 1 and 2 we developed the theory of dynamization and weighting for decomposable searching problems and subclasses thereof. Although a large number of problems are decomposable searching problems, not all problems are. An example is provided by the inside the convex hull problem. Here we are given a set $s \subseteq \mathbb{R}^2$ and a point $x \in \mathbb{R}^2$ and are asked to decide whether $x \in CH(S)$ (= the convex hull of S). In general, there is no relation between $CH(S)$ and $CH(A)$, $CH(B)$ for arbitrary partitions A, B of S. However, if we choose the partition intelligently then there is a relation. Suppose that we order the points in S according to x-coordinate and split into sets A and B such that the x-coordinate of any point in A is no larger than the x-coordinate of any point in B. Then the convex hull of S can be constructed from $CH(A)$ and $CH(B)$ by adding a "low" and the "high" tangent.

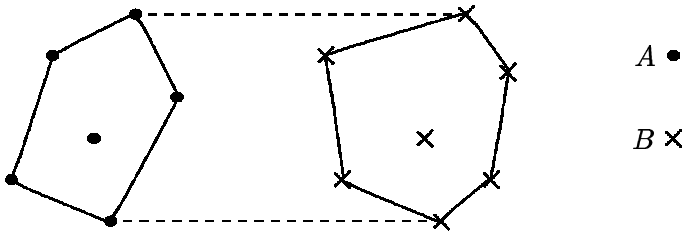


Figure 1. Convex hull of S

These tangents can be constructed in time $O(\log n)$ given suitable representations of $CH(A)$ and $CH(B)$. The details are spelled out in 8.2 and are of no importance here. We infer two things from this observation. First, if we choose A and B such that $|A| = |B| = |S|/2$ and apply the same splitting process recursively to A and B then we can construct the convex hull in time $T(n) = 2 \cdot T(n/2) + O(\log n) = O(n)$. This does not include the time for sorting S according to x-coordinate. The details are described in Theorem 9. Second, convex hulls can be maintained efficiently. If we actually keep around the recursion tree used in the construction of $CH(S)$ then we can insert a new point in S by going down a single path in this tree and redoing the construction along this path only. Since the path has length $O(\log n)$ and we spent time $O(\log n)$ in every node for merging convex hulls this will consume $O((\log n)^2)$ time units per insertion and deletion of a new point. The details are described in Theorem 10 below.

Definition: Let T_1 and T_2 be sets and let $P : 2^{T_1} \to T_2$ be a set problem. P is **order decomposable** if there is a linear order $<$ on T_1 and an operator $\sqcup$:

$T_2 \times T_2 \to T_2$ such that for every $S \subseteq T_1$, $S = \{a_1 < a_2 < \cdots < a_n\}$ and every i

$$P(\{a_1, \ldots, a_n\}) = \sqcup(P(\{a_1, \ldots, a_i\}), P(\{a_{i+1}, \ldots, a_n\})).$$

Moreover, $\sqcup$ is computable in time $C(n)$ in this situation. ∎

We assume throughout that $C(n)$ is non-decreasing. In the convex hull example we have $T_1 = \mathbb{R}^2$, $T_2 =$ the set of convex polygons in $\mathbb{R}^2$, $\sqcup$ merges two convex hulls, and $C(n) = O(\log n)$. We outlined above that convex hulls can be constructed efficiently by divide and conquer. This is true in general for order decomposable problems.

Theorem 9. *Let P be order decomposable. Then $P(S)$, $S \subseteq T_1$ can be computed in time $Sort(|S|) + T(|S|)$ where $Sort(n)$ is the time required to sort a set of n elements according to $<$, and $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(C(n))$ for $n > 1$ and $T(1) = c$ for some constant C.*

Proof: The proof is a straightforward application of divide and conquer. We first sort S in time $Sort(|S|)$ and store S in sorted order in an array. This will allow us to split S in constant time. Next we either compute $P(S)$ directly in constant time if $|S| = 1$ or we split S into sets A and B of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ respectively in constant time (if $S = \{a_1 < a_2 < \cdots < a_n\}$ then $A = \{a_1, \ldots, a_{\lfloor n/2 \rfloor}\}$ and $b = \{a_{\lfloor n/2+1 \rfloor}, \ldots, a_n\}$, compute $P(A)$ and $P(B)$ in time $T(\lfloor n/2 \rfloor)$ and $T(\lceil n/2 \rceil)$ respectively by applying the algorithm recursively, and then compute $P(S) = \sqcup(P(A), P(B))$ in time $C(n)$. Hence $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(1) + C(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(C(n))$. ∎

Recurrence $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + C(n)$ is easily solved for most C (cf. 2.1.3). In particular, $T(n) = O(n)$ if $C(n) = O(n^\epsilon)$ for some $\epsilon < 1$, $T(n) = O(C(n))$ if $C(n) = \Theta(n^{1+\epsilon})$ for some $\epsilon > 0$, and $T(n) = O(C(n) \cdot (\log n)^{k+1})$ if $C(n) = \Theta(n \cdot (\log n)^k)$ for some $k \geq 0$.

The proof of Theorem 9 reflects the close relation between order decomposable problems and divide and conquer. A non-recursive view of divide and conquer is to take any binary tree with $|S|$ leaves, to write the elements of S into the leaves (in sorted order), to solve the basic problems in the leaves and then to use operator $\sqcup$ to compute P for larger subsets of S. What tree should we use? A complete binary tree will give us the most efficient algorithm, but any reasonably weight-balanced tree will not be much worse. If we want to support insertions and deletions this is exactly what we should do. So let D be a BB[α]-tree with $|S|$ leaves for some α. (Exercise 9 shows that we *cannot* obtain the same efficiency by using (a, b)-trees, or AVL-trees, or ...). We store the elements of S in sorted order (according to $<$) in the leaves of D and use D as a search tree for S. What should we store in the internal nodes of D beside the search tree information? A first idea is to store $P(S(v))$ in node v where $S(v)$ is the set stored in the leaves below v. $P(S(v))$ is easily computed bottom-up starting at the leaves and working towards the root.

Not quite, if v has sons, x, y and we compute $P(S(v)) = \sqcup(P(S(x)), P(S(y)))$ then application of $\sqcup$ will in general destroy (the representation of) $P(S(x))$ and $P(S(y))$. Making a copy of $P(S(x))$ and $P(S(y))$ before applying $\sqcup$ might cost a lot more than $C(|S(v)|)$ and is therefore excluded. A different strategy is called for.

We store $P(S(r))$ only in the root r. In internal nodes $v \neq r$ we store two things. First, the sequence $a(v)$ of actions executed to compute $\sqcup$ applied to $P(S(x))$ and $P(S(y))$. This sequence has length $O(C(n))$. Second, the piece $P^*(S(v))$ which is left over from $P(S(v))$ when $P(S(\mathit{father}(v)))$ is computed by applying $\sqcup$ to $P(S(v))$ and $P(S(\mathit{brother}(v))$. We call tree D augmented by this additional information an **augmented** tree.

Lemma 6. *An augmented tree D for set S has space requirement $T(|S|)$ and can be constructed in time $\mathit{Sort}(|S|) + T(|S|)$ where*

$$T(n) = \max_{\alpha \leq \beta \leq 1-\alpha} [T(\beta \cdot n) + T((1-\beta) \cdot n) + O(C(n))].$$

Proof: The recursion for $T(n)$ follows from the fact that $\alpha \leq |S(x)|/|S(v)| \leq 1 - \alpha$ for any node v with sons x, y in a BB[α]-tree D. The space bound follows since at most t storage cells can be used in t time units for any t. ∎

The remark following Theorem 9 also applies to Lemma 6. In particular, $T(n) = O(n)$ if $C(n) = O(n^\epsilon)$ for some $\epsilon < 1$, and .... The space bound stated in Lemma 6 is usually overlay pessimistic. One does not use a new storage cell every time unit in general.

We will next describe how to insert into and delete from an augmented tree. We describe insertion in detail and leave deletion for the reader, deletion being very similar to insertion. Let a be a new point which we want to insert in S. Let D be an augmented tree for S. We first use D as a search tree. This will outline a path p down tree D. Let $p = v_0, v_1, \ldots, v_k$ with v_0 being the root. We walk down this path and reconstruct the $P(S(v_i))$'s as we walk down. More precisely, we start in root v_0 with $P(S(v_0))$ in our hands and use the sequences of actions $a(v_0)$ stored in v_0 and the leftover pieces $P^*(S(v_1))$ and $P^*(S(\mathit{brother}(v_1)))$ stored in v_1 and its brother to reconstruct $P(S(v_1))$ and $P(S(\mathit{brother}(v_1)))$ by running $a(v_0)$ backwards. This will take time $O(C(|S(v_0)|))$. Next we repeat this process with $v_1, \ldots, v_k$. At the end we have reconstructed $P(S(\mathit{brother}(v_i)))$, $1 \leq i \leq k$, and $P(S(v_k))$.

Lemma 7. *Let D be an augmented tree for S, $|S| = n$ and let $p = v_0, \ldots, v_k$ be a path from the root v_0 to a leaf. Then $P(S(\mathit{brother}(v_i)))$, $1 \leq i \leq k$, and $P(S(v_k))$ can be reconstructed in time $O(C(n) \cdot \log n)$.*

Proof: The algorithm outlined above has running time

$$\sum_i C(|S(v_i)|) \leq \sum_i C(n \cdot (1-\alpha)^i) \leq \sum_i C(n) = O(C(n) \cdot \log n)$$

since the depth of the tree is $O(\log n)$ and $|S(v_i)| \leq n \cdot (1-\alpha)^i$. ∎

If $C(n) = \Theta(n^\epsilon)$ for some $\epsilon > 0$ then $\sum_i C(n \cdot (1-\alpha)^i) = n^\epsilon \cdot \sum_i (1-\alpha)^{i \cdot \epsilon} = O(n^\epsilon) = O(C(n))$. In (a,b)-trees this improved claim is not true in general, i.e., there are (a,b)-trees where reconstruction along a path has cost $O(n^\epsilon \cdot \log n)$ if $C(n) = \Theta(n^\epsilon)$ (cf. Exercise 9).

The remainder of the insertion algorithm is now almost routine. We insert the new point a, walk back to the root and merge the P's as we go along. More precisely, we first compute $P(a)$, then merge it with $P(S(v_k))$, then with $P(S(\mathit{brother}(v_k)))$, ... The time bound derived in Lemma 2 applies again except that we forgot about rotations and double rotations.

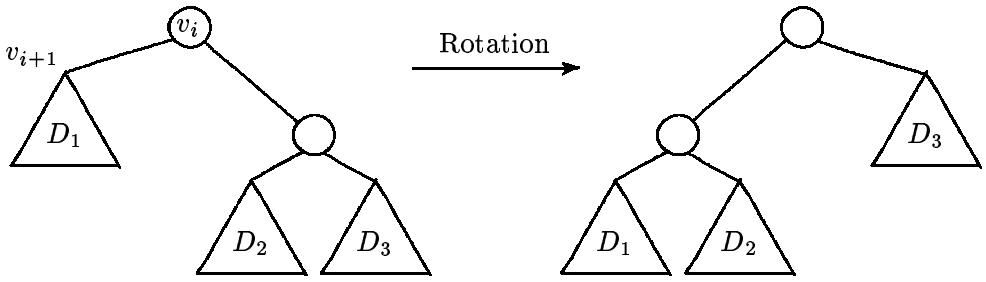


Figure 2.

Suppose that we have to rotate at node v_i and assume that v_{i+1} is the root of subtree D_1. As we walk back to the root we have already computed $P(S(v_{i+1}))$. Also $P(S(\mathit{brother}(v_{i+1})))$ is available from the top-down pass. We reverse the construction at $\mathit{brother}(v_{i+1})$ and thus compute P for the relevant nodes after the rotation. Double rotations are treated similarly, the details are left to the reader. Also it is obvious that the time bound derived in Lemma 2 does still apply, because rotations and double rotations at most require to extend the reconstruction process to a constant vicinity of the path of search. We summarize in:

Theorem 10. *Let P be an order decomposable problem with merging operator $\sqcup$ computable in time $C(n)$. Then P can be dynamized such that insertions and deletions take time $O(C(n))$ if $C(n) = \Theta(n^\epsilon)$ for $\epsilon > 0$ and time $O(C(n) \cdot \log n)$ otherwise.*

Proof: By the discussion above. The time bound follows from Lemma 2 and the remark following it. ∎

In the convex hull problem we have $C(n) = O(\log n)$. Thus we can maintain convex hulls under insertions and deletions with time bound $O((\log n)^2)$ per update. More examples of order decomposable problems are discussed in Exercises 10–20.

7.2. Multi-dimensional Searching Problems

This section is devoted to searching problems in multi-dimensional space. Let U_i, $0 \leq i < d$, be an ordered set and let $U = U_0 \times U_1 \times \cdots \times U_{d-1}$. An elment $x = (x_0, \ldots, x_{d-1}) \in U$ is also called point or record or d-tuple; it is customary to talk about points in geometric applications and about records in database applications. No such distinction is made here. Components x_i are also called coordinates or attributes.

A region searching problem is specified by a set $\Gamma \subseteq 2^U$ of regions in U. The problem is then to organize a static set $S \subseteq U$ such that the queries of the form "list all elements in $S \cap R$" or "count the number of points in $S \cap R$" can be answered efficiently for arbitrary $R \in \Gamma$. We note that region searching problems are decomposable searching problems and hence the machinery developed in 7.1.1 and 7.1.2 applies to them. Thus we automatically have dynamic solutions for region searching problems once a static solution is found. We address four types of region queries.

a) Orthogonal Range Queries: Here Γ is the set of hypercubes in U, i.e.,

$$\Gamma_{OR} = \{R;\ R = [l_0, h_0] \times [l_1, h_1] \times \cdots \times [l_{d-1}, h_{d-1}] \text{ where } l_i, h_i \in U_i \text{ and } l_i \leq h_i\}.$$

b) Partial Match Queries: Here Γ is the set of degenerated hypercubes where every side is either a single point or all of U_i, i.e.,

$$\Gamma_{PM} = \{R;\ R = [l_0, h_0] \times [l_1, h_1] \times \cdots \times [l_{d-1}, h_{d-1}] \text{ where } l_i, h_i \in U_i \text{ and either } l_i = h_i \text{ or } l_i = -\infty \text{ and } h_i = +\infty \text{ for every } i\}.$$

if $l_i = h_i$ then the i-th coordinate is specified, otherwise it is unspecified.

c) Exact Match Queries: Here Γ is the set of singletons, i.e.,

$$\Gamma_{EM} = \{R;\ R = \{x\} \text{ for some } x \in U\}.$$

d) Polygon Queries: Polygon queries are only defined for $U = \mathbb{R}^2$. We have

$$\Gamma_P = \{R;\ R \text{ is a simple polygonal region in } \mathbb{R}^2\}.$$

Exact match queries are not really a new challenge; however the three other types of problems are. There seems to be no single data structure doing well on all of them and we therefore describe three data structures: d-dimensional trees, polygon trees and range trees. d-dimensional trees and polygon trees use linear space and solve partial match queries and polygon queries in time $O(n^\epsilon)$ where ϵ depends on the type of the problem. Range trees allow us to solve orthogonal range queries

in time $O((\log n)^d)$ but they use non-linear space $O(n \cdot (\log n)^{d-1})$. In fact they exhibit a tradeoff between speed and space.

In view of Chapter 3 these results are disappointing. In one-dimensional space we could solve a large number of problems in linear space and logarithmic time, in higher dimensions all data structures mentioned above either use non-linear space or use "rootic" time $O(n^\epsilon)$ for some ϵ, $0 < \epsilon < 1$. Section 7.2.3 is devoted to lower bounds and explains this behavior. We show that partial match requires rootic time when space is restricted to its minimum and that orthogonal range queries and polygon queries either require large query or large update time. Large update time usually points to large space requirement, although it is not conclusive evidence.

7.2.1. D-dimensional Trees and Polygon Trees

We start with d-dimensional trees and show that they support partial match retrieval and orthogonal range querieswith rootic search time. However, they do not do well for arbitrary polygon queries. A discussion of why they fail for polygon retrieval leads to polygon trees.

d-dimensional trees are a straightforward, yet powerful extension of one-dimensional trees. At every level of a dd-tree we split the set according to one of the coordinates. Fairness demands that we use the different coordinates with the same frequency; this is most easily achieved if we go through the coordinates in cyclic order.

Definition: Let $S \subseteq U_0 \times \cdots \times U_{d-1}$, $|S| = n$. A dd-tree for S (starting at coordinate i) is defined as follows

1) If $d = n = 1$ then it consists of a single leaf labeled by the unique element $x \in S$.
2) If $d > 1$ or $n > 1$ then it consists of a root labeled by some element $d_i \in U_i$ and three subtrees $T_<$, $T_=$ and $T_>$. Here $T_<$ is a dd-tree starting at coordinate $(i+1) \bmod d$ for set $S_< = \{x \in S;\ x = (x_0, \ldots, x_{d-1}) \text{ and } x_i < d_i\}$, $T_>$ is a dd-tree starting at coordinate $(i+1) \bmod d$ for set $S_> = \{x \in S;\ x = (x_0, \ldots, x_{d-1}) \text{ and } x_i > d_i\}$ and $T_=$ is a $(d-1)$-dimensional tree starting at coordinate $i \bmod (d-1)$ for set $S_= = \{(x_0, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{d-1});\ x = (x_0, \ldots, x_{i-1}, d_i, x_{i+1}, \ldots, x_{d-1}) \in S\}$. ∎

Figure 3 shows a 2d-tree for set $S = \{(1, \text{II}), (1, \text{III}), (2, \text{I}), (2, \text{III}), (3, \text{I}), (3, \text{II})\}$ starting at coordinate 0. Here $U_0 = U_1 = \{1, 2, 3\}$. Arabic and roman numerals are used to distinguish coordinates.

It is very helpful to visualize 2d-trees as subdivisions of the plane. The root node splits the plane by vertical line $x_0 = 2$ into three parts: left halfplane, right halfplane and the line itself. The left son of the root then splits the left halfplane by horizontal line $x_1 = 2, \ldots$.

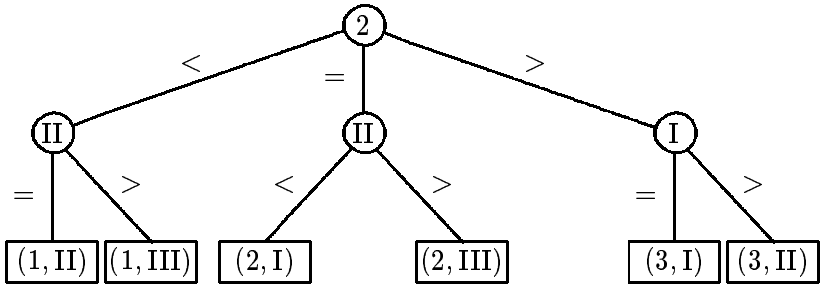


Figure 3.

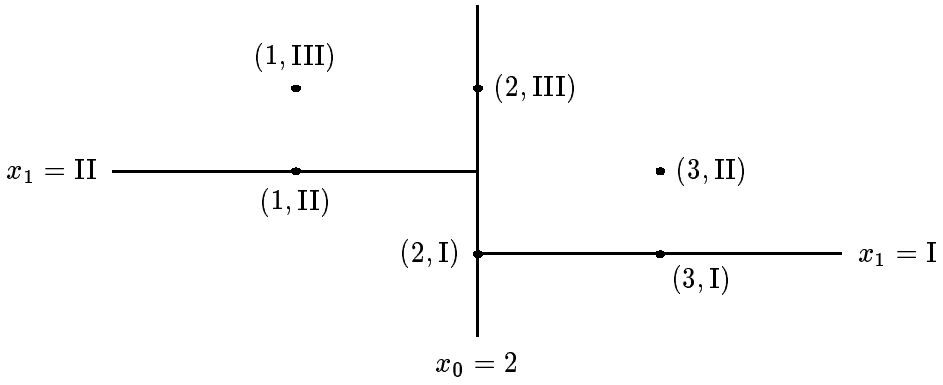


Figure 4.

The three sons of a node v in a dd-tree do not all have the same quality. The root of $T_=$ (the son via the $=$-pointer) represents a set of one smaller dimension. In general we will not be able to bound the size of this set. The roots of $T_<$ and $T_>$ (the sons via the $<$-pointer and the $>$-pointer) represent sets of the same dimension but generally smaller size. Thus every edge of a dd-tree reduces the complexity of the set represented: either in dimension or in size. In 1d-trees, i.e., ordinary search trees, only reductions in size are required.

It is clear how to perform exact match queriesin dd-trees. Start at the root, compare the search key with the value stored in the node and follow the correct pointer. Running time is proportional to the height of the tree. Our first task is therefore to derive bounds on the height of dd-trees.

Definition:

a) Let T be a dd-tree and let v be a node of T. Then $S(v)$ is the set of leaves in the subtree with root v, $d(v)$ is the **depth** of node v, and $sd(v)$, the number of $<$-pointers and $>$-pointers on the path from the root to v, is the **strong depth** of v. Node x is a **proper son** of node v if it is a son via a $<$- or $>$-pointer.

b) A dd-tree is **ideal** if $|S(x)| \leq |S(v)|/2$ for every node v and all proper sons x of v. ∎

Ideal dd-trees are a generalization of perfectly balanced 1d-trees.

Lemma 1. *Let T be an ideal dd-tree for set S, $|S| = n$.*

a) $d(v) \leq d + \log n$ for every node v of T.

b) $sd(v) \leq \log n$ for every node v of T.

Proof: a) follows from b) and the fact that at most d =-pointers can be on the path to any node v. Part b) is immediate from the definition of ideal tree. ∎

Theorem 1. *Let $S \subseteq U = U_0 \times \cdots \times U_{d-1}$, $|S| = n$.*

a) *An exact match query in an ideal dd-tree for S takes time $O(d + \log n)$.*

b) *An ideal dd-tree for S can be constructed in time $O(n \cdot (d + \log n))$.*

Proof: a) Immediate from Lemma 1, a).

b) We describe a procedure which constructs ideal dd-trees in time $O(n \cdot (d + \log n))$. Let $S_0 = \{x_0;\ (x_0, \ldots, x_{d-1}) \in S\}$ be the multi-set of 0-th coordinates of S. We use the linear time median algorithm of 2.4 to find the median d_0 of S_0. d_0 will be the label of the root. then clearly $|S_<| \leq |S|/2$ and $|S_>| \leq |S|/2$ where $S_< = \{x \in S;\ x_0 < d_0\}$ and $S_> = \{x \in S;\ x_0 > d_0\}$. We use the same algorithm recursively to construct dd-tree for $S_<$ and $S_>$ (starting at coordinate 1) and a $(d-1)$-dimensional tree for $S_=$. This algorithm will clearly construct an ideal dd-tree T for S. The bound on the running time can be seen as follows. In every node v of T we spend $O(|S(v)|)$ steps to compute the median of a set of size $|S(v)|$. Furthermore, $S(v) \cap S(w) = \emptyset$ if v and w are nodes on the same depth and hence

$$\sum_{d(v)=k} |S(v)| \leq n$$

for every k, $0 \leq k < d + \log n$. Thus the running time is bounded by

$$\begin{aligned} \sum_{v \text{ node of } T} O(|S(v)|) &= O\left(\sum_{0 \leq k \leq d + \log n} \sum_{d(v)=k} |S(v)| \right) \\ &= O(n \cdot (d + \log n)). \end{aligned}$$

∎

Insertions into dd-trees are a non-trivial problem. A first idea is to use an analogue to the naive insertion algorithm into one-dimensional trees. If x is to be inserted into tree T, search for x in T until a leaf is reached and replace that leaf by a small subtree with two leaves. Of course, the tree will not be ideal after the insertion in general. We might define weight-balanced dd-trees to remedy the situation, i.e., we choose some parameter α, say $\alpha = 1/4$, and require that $|S(x)| \leq (1 - \alpha)|S(v)|$ for every node v and all proper sons x of v. This is a generalization of BB[α]-trees. Two problems arise. Both problems illustrate a major difference between one-dimensional and multi-dimensional searching.

The first problem is that although Theorem 1 is true for weight-balanced dd-trees, Theorem 2 and 3 below are false, i.e., query time in near-ideal dd-trees may have a different order than query time in ideal trees. More precisely, partial match in ideal 2d-trees has running time $O(\sqrt{n})$ but it has running time $\Omega(n^{1/\log 8/3}) = \Omega(n^{0.706})$ in weight-balanced dd-trees, $\alpha = 1/4$ (Exercise 14). Thus weight balanced dd-trees are only useful for exact match queries.

A second problem is that weight-balanced dd-trees are hard to rebalance. Rotations are of no use since splitting is done with respect to different coordinates on different levels. Thus it is impossible to change the depth of a node as rotations do. There is a way out. Suppose that we followed path $p = v_0, v_1, \ldots$ to insert point x. Let i be minimal such that v_i goes out of balance by the insertion. Then rebalance the tree by replacing the subtree rooted at v_i by an ideal tree for set $S(v_i)$. This ideal tree can be constructed in time $O(m \cdot (d + \log m))$ where $m = |S(v_i)|$. Thus rebalancing is apparently not as simple and cheap as in one-dimensional trees. The worst case cost for rebalancing after an insertion is clearly $O(n \cdot (d + \log n))$ since we might have to rebuild the entire tree. However, amortized time bounds are much better as we will sketch. We use techniques developed in 3.5.1 (in particular in the proof of Theorem 4). We showed there (Lemmas 2 and 3 in the proof of Theorem 4), that the total number of rebalancing operations caused by nodes v with $1/(1-\alpha)^i \leq |S(v)| \leq 1/(1-\alpha)^{i+1}$ during the first n insertions (and deletions) is $O(n \cdot (1-\alpha)^i)$. A rebalancing operation caused by such a node has cost $O((1-\alpha)^{-i} \cdot (d+1))$ in weight-balanced dd-trees. Hence the total cost of restructuring a weight-balanced dd-tree during a sequence of n insertions and deletions is

$$\sum_{0 \leq i \leq O(\log n)} O(n \cdot (1-\alpha)^i \cdot (1-\alpha)^{-i} \cdot (d+1)) = O(n \cdot \log n \cdot (d + \log n)).$$

Thus the amortized cost of an insertion or deletion is $O(\log n \cdot (d + \log n))$. The details of this argument are left for Exercise 13.

Dynamization (cf. 7.1) also gives us dynamic dd-trees with $O((d+\log n) \cdot \log n)$ insertion and deletion time. Query time for exact match queriesis $O((d+\log n) \cdot \log n)$ which is not quite as good as for weight-balanced dd-trees. However, dynamization has one major advantage. The time bounds for partial match and orthogonal range queries(Theorem 2, 3 and 4 below) stay true for dynamic dd-trees.

It is about time that we move to partial match queries. Let $R = [l_0, h_0] \times \cdots \times [l_{d-1}, h_{d-1}]$ with $l_i = h_i$ or $l_i = -\infty$, $h_i = +\infty$ be a partial match query. If $l_i = h_i$ then the i-th coordinate is called specified. We use s to denote the number of specified coordinates. The algorithm for partial match queriesis an extension of the exact match algorithm. As always we start searching in the root. Suppose that the search reached node v. Suppose further that we split according to the i-th coordinate in v and that key d_i is stored in v. If the i-th coordinate is specified in query R, then the search proceeds to exactly one son of v, namely the son via the $<$-pointer if $l_i = h_i < d_i$, the son via the $=$-pointer if $l_i = h_i = d_i$, and .... If the i-th coordinate is unspecified in query R then the search proceeds to all three sons

of v. Once we reach a leaf, we return it if it belongs to region R. The correctness of this algorithm heavily depends on set S. We treat a favourable special case first: invertible sets.

Definition: $S \subseteq U = U_0 \times U_1 \times \cdots \times U_{d-1}$ is **invertible** if for all $x = (x_0, \ldots, x_{d-1}) \in S$, $y = (y_0, \ldots, y_{d-1}) \in S$: $x_i = y_i$ for some i implies $x = y$. ∎

A set is invertible if all projection functions are injective when restricted to S.

Theorem 2. *Let T be an ideal dd-tree for invertible set $S \subseteq U = U_0 \times U_1 \times \cdots \times U_{d-1}$. Then a partial match query with $s < d$ specified components takes time*

$$O(d \cdot 2^{d-s} \cdot n^{1-\frac{s}{d}}).$$

Proof: Let T' be the subtree of T consisting of all nodes visited by the search. It suffices to show that the number of nodes of T' is bounded by $O(d \cdot 2^{d-s} \cdot n^{1-s/d})$. A node of T' is called **branching** if it has a proper son and non-branching otherwise. Since S is invertible all descendants of non-branching nodes are non-branching. Hence all branching nodes can be reached by following <- and >-pointers only. A branching node of T' is a **proper branching node** if it has two proper sons.

We claim that there are at most $2^{\lceil (\log n)/d \rceil \cdot (d-s)}$ proper branching nodes in T'. This follows from the fact that at most $d-s$ out of any d consecutive nodes on any path through T' are proper branching nodes, because only $d-s$ out of d consecutive nodes split according to unspecified components. Also $d(v) = sd(v) \leq \log n$ for all branching nodes. Hence there are at most $\lceil (\log n)/d \rceil \cdot (d-s)$ proper branching nodes on any path through T' and thus the bound follows. It remains to count the improper branching nodes and the non-branching nodes in T'. Again consider any path through T'. Then there can be at most d consecutive nodes which are not proper branching nodes and hence the total number of nodes of T' is

$$\begin{aligned} O(d \cdot 2^{\lceil \frac{\log n}{d} \rceil \cdot (d-s)}) &= O(d \cdot 2^{d-s} \cdot n^{\frac{d-s}{d}}) \\ &= O(d \cdot 2^{d-s} \cdot n^{1-\frac{s}{d}}). \end{aligned}$$ ∎

The behavior of the partial match algorithm on general sets is harder to analyze. Let us look at an example first. Let $U_i = \mathbb{R}$, $0 \leq i < d$, and let $S = \{0\}^k \times \{0, \ldots, m-1\}^{d-k}$ for some m and k. Then $|S| = m^{d-k}$. Consider first partial match query R_1 which specifies the first $s = k$ coordinates as being 0 and leaves the remaining coordinates unspecified. Then the answer to the query is the entire set S and hence the running time of any algorithm must be at least linear. Consider next partial match query R_2 which specifies the first $s = k+1$ coordinates as being 0 and leaves the remaining coordinates unspecified. Then the query is "equivalent" to a partial match query in a $d - k = d - s + 1$ dimensional set with one specified coordinate. In view of Theorem 2 we therefore cannot hope to do better than $O(n^{1-1/(d-s+1)})$ time units. This is indeed the bound.

Theorem 3. *Let T be an ideal dd-tree for S, $|S| = n$. Then a partial match query with s specified components takes time*

$$O(f(d, d-s) \cdot n^{\max(\frac{1}{2}, 1-\frac{1}{d-s+1})} + (d+1) \cdot |A|).$$

Here A is the set of answers to the query and $f(d, d-s)$ is some function increasing in both arguments. f is independent of T and S.

Proof: Let T' be the subtree of T consisting of all nodes visited in the search. We split the set of nodes of T' into three classes which we count separately. A node is a tertiary node (belongs to the third class) if all descendants of v belong to A, i.e., if $S(v) \subseteq A$. The number of tertiary nodes is clearly bounded by $(d+1) \cdot |A|$. A non-tertiary node is a primary node if it is reachable without using an $=$-pointer. All other nodes of T' are secondary nodes. We will show that the number of primary and secondary nodes is bounded by

$$f(d, d-s) \cdot n^{\max(\frac{1}{2}, 1-\frac{1}{d-s+1})}$$

for some suitable function f. The proof is by induction on $d-s$ and for fixed $d-s$ by induction on s and n.

If $d = s$ then partial match is equivalent to exact match and the claim follows from Theorem 1, a). So let us assume $d > s$. If $s = 0$ then all the nodes are tertiary and the claim is trivial. This leaves the case $d > s \geq 1$. If n is small then the claim is certainly true by suitable choice of $f(d, d-s)$.

The primary nodes are easy to count. We have shown in the proof of Theorem 2 that their number is $O(d \cdot 2^{d-s} \cdot n^{1-s/d})$. It remains to count the secondary nodes.

We group the secondary nodes into maximal subtrees. If v is the root of such a subtree then v is reached via an $=$-pointer and there is no other $=$-pointer on the path to v. Thus $sd(v) = d(v) - 1$ and $|S(v)| \leq n/2^{sd(v)} \leq 2 \cdot n/2^{d(v)}$. Also there can be at most $2^{\lceil j/d \rceil \cdot (d-s)}$ such nodes v with $d(v) = j$. This follows from the fact that all nodes on the path to v are primary nodes and hence at most $\lceil j/d \rceil \cdot (d-s)$ of these nodes can be proper branching nodes; cf. the proof of Theorem 2.

In the subtree with root v we have to compute a partial match query on a $(d-1)$-dimensional set with s' specified components. Here $s' = s$ or $s' = s-1$. Also, $s' \geq 1$. Note that v and all its descendants are tertiary nodes if $s' = 0$. By induction hypothesis there are at most

$$f(d-1, d-1-s') \cdot m^{\max(\frac{1}{2}, 1-\frac{1}{d-1-s'+1})}$$

non-tertiary nodes visited in the subtree with root v where $m = |S(v)|$. For the reminder of the argument we have to distinguish two cases, $s = 1$ and $s \geq 2$.

Case 1: $s \geq 2$.
Since $d-1-s' \leq d-s$, f is increasing, and $d-s \geq 1$ we conclude that the number of non-tertiary nodes below v is bounded by $f(d-1, d-s) \cdot m^{1-1/(d-s+1)}$. We finish

the proof by summing this bound for all roots of maximal subtrees of secondary nodes. Let RT be the set of such roots. Then

$$\begin{aligned}
&\sum_{v \in RT} f(d-1, d-s) \cdot |S(v)|^{1-\frac{1}{d-s+1}} \\
&\leq f(d-1,d-s) \cdot \sum_{j \geq 1} 2^{\lceil j/d \rceil \cdot (d-s)} \cdot (2n/2^j)^{1-\frac{1}{d-s+1}} \\
&\leq (2n)^{1-\frac{1}{d-s+1}} \cdot f(d-1,d-s) \cdot 2^{(d-s)} \cdot \sum_{j \geq 1} [2^{(d-s)/d-1+1/(d-s+1)}]^j \\
&\leq (f(d,d-s) - d \cdot 2^{d-s}) \cdot n^{1-\frac{1}{d-s+1}}
\end{aligned}$$

for suitable choice of $f(d, d-s)$. Note that $(d-s)/d - 1 + 1/(d-s+1) = -s/d + 1/(d-s+1) < 0$ for $2 \leq s \leq d$. Adding the bound for the number of primary nodes proves the theorem.

Case 2: $s = 1$. Define RT as in Case 1. Since $s \geq s' \geq 1$ we have $s' = 1$. Consider the case $d = 2$ first. Then the query below v degenerates to an exact match query in a one-dimensional set, and hence there are at most $d + \log |S(v)|$ non-tertiary nodes below v. Summing this bound for all nodes in RT we obtain

$$\begin{aligned}
&\sum_{v \in RT} (d + \log |S(v)|) \\
&\leq \sum_{j=1}^{d+\lceil \log n \rceil} 2^{\lceil j/d \rceil \cdot (d-s)} \cdot (d + \log(2n/2^j)) \\
&\leq 2^{(d-s)} \cdot \sum_{k=-d}^{\lceil \log n \rceil - 1} 2^{(\lceil \log n \rceil - k)/d} \cdot (d+1+k) \\
&\text{where we used the substitution } k = \lceil \log n \rceil - j \\
&\leq 2^{d-s+1} \cdot n^{1/d} \cdot \sum_{k=-d}^{\lceil \log n \rceil} (d+1+k)/2^{k/d} \\
&\leq (f(2,1) - d \cdot 2^{d-s}) \cdot n^{\max(\frac{1}{2}, 1-\frac{1}{d-s+1})}
\end{aligned}$$

by suitable choice of $f(2,1)$; recall that $d = 2$ and $s = 1$. Adding the bound for the number of primary nodes proves the theorem.

It remains to consider the case $d \geq 3$. We infer from the induction hypothesis that the number of non-tertiary nodes below $v \in RT$ is bounded by $f(d-1, d-2) \cdot$

$|S(v)|^{1-1/(d-1)}$ in this case. Summing this bound for all $v \in RT$ we obtain

$$\sum_{f \in RT} f(d-1,d-2) \cdot |S(v)|^{1-\frac{1}{d-1}}$$

$$\leq \sum_{j=1}^{d+\lceil \log n \rceil} f(d-1,d-2) \cdot 2^{\lceil j/d \rceil \cdot (d-1)} \cdot (2n/2^j)^{1-\frac{1}{d-1}}$$

$$\leq (2n)^{1-\frac{1}{d-1}} \cdot f(d-1,d-2) \cdot 2^{d-1} \cdot \sum_{j=1}^{d+\lceil \log n \rceil} [2^{\frac{d-1}{d}+\frac{1}{d-1}-1}]^j$$

$$\leq (2n)^{1-\frac{1}{d-1}} \cdot f(d-1,d-2) \cdot c \cdot (2n)^{\frac{1}{(d-1)\cdot d}}$$

where c is a constant depending on d

$$\leq (f(d,d-1) - d \cdot 2^{d-s}) \cdot n^{1-\frac{1}{d}}$$

by suitable choice of $f(d,d-1)$. Adding the bound for the number of primary nodes proves the theorem. ∎

Theorem 3 shows that d-dimensional trees support partial match querieswith rootic running time. In particular if $d = 2$ and $s = 1$ then the running time is $O(\sqrt{n}+|A|)$ even in the case of general sets. We will see in Section 7.4.1 that this cannot be improved without increasing storage. However it is trivial to improve upon this result by using $O(d! \cdot n)$ storage.

Let $S \subseteq U = U_0 \times \cdots \times U_{d-1}$. For any of the $d!$ possible orderings of the attributes build a search tree as follows: Order S lexicographically and build a standard one-dimensional search tree for S. A partial match query with s specified components is then easily answered in time $O(d \cdot \log n + |A|)$. Assume w.l.o.g. that the first s attributes are specified, i.e., $R = [l_0,h_0] \times \cdots \times [l_{d-1},h_{d-1}]$ with $l_i = h_i$ for $0 \leq i < s$ and $l_i = -\infty$, $h_i = +\infty$ for $s \leq i < d$. Search for key $(l_0,\ldots,l_{s-1},-\infty,\ldots,-\infty)$ in tree T corresponding to the natural order of attributes. This takes time $O(d \cdot \log n)$. The answer to the query will then consist of the next $|A|$ leaves of T in increasing order. Thus logarithmic search time can be obtained at the expense of increased storage requirement. For small d, say $d = 2$, this approach is feasible and in fact we use it daily. After all, there is a German-English and an English-German dictionary and no one ever complained about the redundancy in storage.

Another remark about Theorem 3 is also in order at this place. The running time stated in Theorem 3 is for the enumerative version of partial match retrieval: "Enumerate all points in $S \cap R$". A simpler version is to count only $|S \cap R|$. If we store in every node v of a dd-tree the cardinality $|S(v)|$ of $S(v)$ then the counting version of partial match retrieval has running time $O(f(d,d-s) \cdot n^{\max(\frac{1}{2},1-\frac{1}{d-s+1})})$; cf. Exercise 15.

The next harder type of queries are orthogonal range queries. Let $R = [l_0,h_0] \times \cdots \times [l_{d-1},h_{d-1}]$ be a hypercube in $U_0 \times \cdots \times U_{d-1}$. Before we can explain the search

algorithm we need to introduce one more concept; the range of a node. We can associate a hypercube $Reg(v)$ with every node of a dd-tree in a natural way, namely $Reg(v) = \{x \in U_0 \times \cdots \times U_{d-1};$ an exact match query of x goes through $v\}$. $Reg(v)$ is easily determined recursively. If v is the root then $Reg(v) = U_0 \times \cdots \times U_{d-1}$. If v is a son of w, say via the $<$-pointer, and w is labeled with $d \in U_i$ then $Reg(v) = Reg(w) \cap \{(x_0, \ldots x_{d-1});\ x_i < d\}$.

We are now in a position to describe the search algorithm for orthogonal range query. Let R be the query hypercube. As always we start the search in the root r. Then $R \cap Reg(r) \neq \emptyset$. Assume for the inductive step that the search has reached node v with $R \cap Reg(x) \neq \emptyset$. There is at least one such son and all sons with that property can be found in time $O(1)$. Finally, if v is a leaf then we output the leaf if $v \in R$.

We analyze the running time of this algorithm only in two dimensions and leave the higher-dimensional case to the reader. The proof for the higher-dimensional case is completely analogous but somewhat more tedious.

Theorem 4. *Let T be an ideal 2d-tree for $S \subseteq U_0 \times U_1$, $|S| = n$. Then an orthogonal range query takes time*

$$O(d \cdot 4^d \cdot n^{1-\frac{1}{d}} + d \cdot |A|)$$

where A is the set of answers.

Proof: Let $R = [l_0, h_0] \times [l_1, h_1]$ be a rectangle in $U_0 \times U_1$ and let T' be the subtree of all nodes visited when answering query R. Observe first that $Reg(v) \cap R \neq \emptyset$ iff v is visited in the search. Observe next that $Reg(v) \subseteq R$ implies $S(v) \subseteq A$. Hence the number of nodes v of T' with $Reg(v) \subseteq R$ is certainly bounded by $d \cdot |A|$. It remains to count the number of nodes v with $Reg(v) \cap R \neq \emptyset$ and $Reg(v) - R \neq \emptyset$. Let N be the set of such nodes. If $v \in N$ then there must be one of the four bounding line segments of R which intersects $Reg(v)$ but does not contain $Reg(v)$. Thus $|N| \leq 4 \cdot t$ where t is the maximal number of nodes such that $Reg(v)$ intersects with but is not contained in any fixed horizontal or vertical line segment.

Claim: *Let T be an ideal 2d-tree with n leaves and let $L = \{(x, y) \in U_0 \times U_1;\ x = l_0, l_1 \leq y \leq h_1\}$ be a vertical line segment. Then the number of nodes v such that $Reg(v)$ intersects L but is not contained in L is $O(\sqrt{n})$.*

Proof: A node v of T is called a primary node if there is no $=$-pointer on the path form the root to v. Let P_d be the number of primary nodes v of depth k such that $Reg(v)$ intersects L (but is not contained in L). Then $P_0 \leq P_1 \leq 2$ and $P_{k+2} \leq 2 \cdot P_k$ follows from the observation that a vertical line can intersect at most two of the four Regions R_1, R_2, R_3, R_4 associated with the proper grandsons of any node v. This fact is illustrated in Figure 5. From $P_0 \leq P_1 \leq 2$ and $P_{k+1} \leq 2 \cdot P_k$ we infer $P_k \leq 2 \cdot 2^{k/2}$.

Next consider any primary node v of depth k such that $Reg(v)$ intersects L. Let x be the son of v via the $=$-pointer. Then $S(x) \subseteq S(v)$ and hence $|S(x)| \leq$

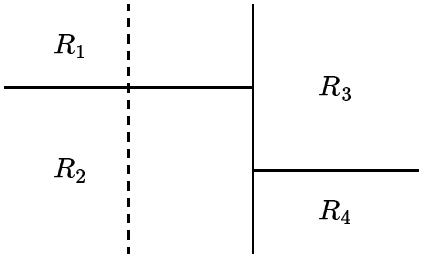


Figure 5.

$|S(v)| \leq n/2^k$. Also there are at most $2 \cdot \log S(x)$ descendants w of x such that $Reg(w)$ intersects L but is not contained in L. This can be seen as follows.

The tree with root s is a one-dimensional search tree for a set of nodes which lie either on a horizontal or a vertical line. If they lie on a horizontal line then the search below x follows exactly one path down the tree. If they lie on a vertical line (which then must be the line $x = l_0$) then $Reg(w)$ intersects L but is not contained in L iff either $l_1 \in Reg(w)$ or $h_1 \in Reg(w)$. The set of nodes w with $l_1 \in Reg(w)$ ($h_1 \in Reg(w)$) form a path in the tree with root x. Thus there are at most $2 \cdot \log |S(x)|$ descendants w of x such that $Reg(w)$ intersects L but is not contained in L.

Putting everything together we have shown that the number of nodes v in T such that $Reg(v)$ intersects L but is not contained in L is at most

$$\sum_{0 \leq k \leq \log n} 2 \cdot 2^{k/2} \cdot 2 \cdot \log n/2^k = 4 \cdot \sqrt{n} \cdot \sum_{0 \leq k \leq \log n} 2^{(k - \log n)/2} \cdot (\log n - k)$$
$$= O(\sqrt{n}).$$

This proves the claim and the theorem. ∎∎

So in two dimensions ($d = 2$) 2d-trees support even orthogonal range queries with running time $O(\sqrt{n})$. Can we stretch the use of dd-trees even further? If we want to talk about more complicated queries we have to make some additional assumptions about the U_i's. Let us assume for the sequel that $d = 2$ and $U_0 = U_1 = \mathbb{R}$. It is then natural to generalize orthogonal range queries to arbitrary polygon queries. In a database which contains persons stored by income and number of children we might ask for all persons where the income exceeds \$1000 plus \$200 for every child. This query describes a triangle in two-space. Do 2d-trees support efficient polygon searching? The answer is no (cf. Exercise 18) and the reason for this can be seen clearly in the proof of Theorem 4. A line segment in arbitrary position can intersect the regions associated with all four proper grandsons of a node v and in fact can intersect the regions of all nodes of a 2d-tree. What can we do to overcome this difficulty? First, every node v of the tree should define a subdivision of $Reg(v)$ such that any line segment can intersect only a *proper* subset of the regions in the subdivision. One possible way of achieving this is to divide $Reg(v)$ into four regions by two straight lines.

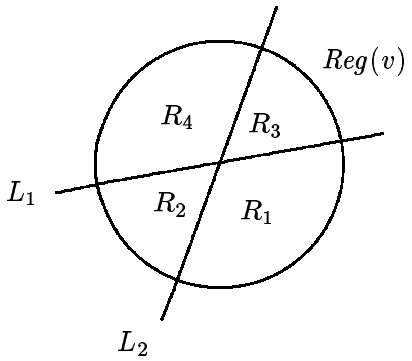


Figure 6.

Then any straight line can intersect at most three of the four regions R_1, R_2, R_3, R_4 plus a number of the "one-dimensional" regions defined by the lines themselves. With the notation of the claim in the proof of Theorem 4 we would obtain $P_{k+1} \leq 3 \cdot P_k$ and hence could hope for a search time of $3^{\log n/\log 4} = n^{\log 3/\log 4} \leq n^{0.8}$. Note that the depth of the tree will be $\log n/\log 4$ because we divide into four pieces in every step. However, we have to be careful. The arrangement above is only correct if the depth of the tree is indeed $\log n/\log 4$, i.e., if the tree is ideal. Thus lines L_1 and L_2 above have to be chosen such that $|R_i \cap S(v)| \leq \lceil |S(v)|/4 \rceil$ for $1 \leq i \leq 4$. The following lemma shows that this is always possible.

Lemma 2. *Let $S \subseteq \mathbb{R}^2$, $|S| = n$ and let n_1, n_2, n_3, n_4 be such that $n_1 + n_2 + n_3 + n_4 \leq n$. If L_1 is a line such that $n_1 + n_2$ points of S are on one side of L_1 and $n_3 + n_4$ points of S are on the other side of L_1 then there is a line L_2 such that the four open regions R_1, R_2, R_3 and R_4 defined by L_1 and L_2 contain at most n_1, n_2, n_3, n_4 points of S respectively. Also L_2 can be computed in time $O(n^2)$.*

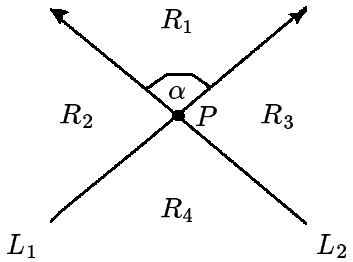


Figure 7.

Proof: For any point P on L_1 let $f(P)$ be the minimum angle α between L_1 and L_2 such that regions R_1, R_2 contain at most n_1, n_2 points respectively. Then $f(P)$ is a continuous function of P. Also $\lim_{P \to -\infty} f(P) = 0$ and $\lim_{P \to +\infty} f(P) = \pi$. Similarly define $g(P)$ be the minimum angle α between lines L_1 and L_2 such that regions R_3, R_4 contain at most n_3 and n_4 points respectively. Then $g(P)$ is a

continuos function of P and $\lim_{P\to-\infty} g(P) = \pi$ and $\lim_{P\to+\infty} g(P) = 0$. Hence there is a point P such that $f(P) = g(P)$. Then P and $f(P)$ define line L_2 with the desired property. This shows the existence of line L_2. It also shows that line L_2 can be assumed to go through two points of S. Thus there are only n^2 candidates for L_2.

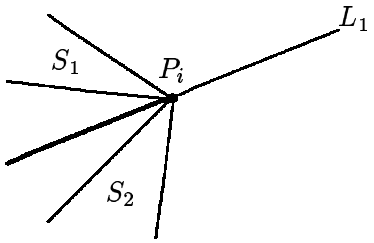


Figure 8.

Let $K_1, \ldots, K_k$, $k = n \cdot (n-1)/2$ be the lines defined by all pairs of points of S ordered according to their intersection point with L_1. Let P_i be the point of intersection of K_i and L_1. Consider any fixed P_i. Express all points of S in polar coordinates with respect to P_i and find among the $n_1 + n_2$ points "above" L_1 two points which define the n_1-th and (n_1+1)-th largest angle between line L_1 and the line defined by P_i and the point. This can be done in time $O(n_1+n_2)$ by the linear time selection algorithm 2.4. In this way we have computed a sector S_1 through which line L_2 must go if it were to intersect L_1 in P_i. In a similar way we compute sector S_2 based on the points "below" L_2. If there is a line which goes through sectors S_1 and S_2 the we are done and have found line L_2. If sectors S_1 and S_2 do not have a line in common (as it is the case in Figure 8) then we can restrict the search to one of the halflines defined by L_1 and P_i. In Figure 8 this halfline is shown bold. We summarize. In time $O(n)$ we can either determine that L_2 goes through P_i or exclude one of the halflines defined by L_1 and P_i.

This suggests that we can use binary search to find line L_2. We first compute in time $O(n^2)$ lines $K_1, \ldots, K_k$ and points $P_1, \ldots, P_k$. Next we find the median point of $P_1, \ldots, P_k$ in time $O(n^2)$. Then we are either done or can restrict the search to $k/2$ points. This decision takes time $O(n)$. Thus line L_2 can be found in $O(\log n^2)$ iterations and the cost of the i-th iteration is $O(k/2^i + n)$. Total cost is thus $O(n^2)$. ∎

Lemma 2 and the preceding discussion lead to:

Definition:

a) A 4-way polygon tree T for set $S \subseteq \mathbb{R}^2$, $|S| = n$ is defined as follows: If set S is collinear then T is an ordinary one-dimensional search tree for S. If set S is not collinear then T consists of a root r and six subtrees. There are two lines L_1 and L_2 associated with r and there is one subtree for each of the six sets $S \cap R_1$, $S \cap R_2$, $S \cap R_3$, $S \cap R_4$, $S \cap L_1$, $S \cap L_2$. Here R_1, R_2, R_3, R_4 are the four open regions defined by lines L_1 and L_2.

b) A 4-way polygon tree T is ideal if for every node v of T and son x of v: If $S(v)$ is collinear then $|S(x)| \leq \lceil |S(v)|/2 \rceil$ and if $S(v)$ is not collinear and x is one of the four sons corresponding to regions $R_1, \ldots, R_4$ then $|S(x)| \leq \lceil |S(v)|/4 \rceil$. ∎

Theorem 5. *Let $S \subseteq R^2$, $|S| = n$.*

a) *An ideal 4-way polygon tree for set S can be constructed in time $O(n^2)$.*

b) *If T is an ideal 4-way polygon tree for S and R is a polygonal region with s sides then $A = R \cap S$ can be computed in time $O(s \cdot n^{\log 3/\log 4} + |A|)$.*

Proof: a) If S is collinear then an ideal tree can be constructed in time $O(n \cdot \log n)$, the time required to sort S. If S is not collinear then lines L_1, L_2 dividing the plane into four open regions containing at most $\lceil n/4 \rceil$ points of S each can be computed in time $O(n^2)$ by Lemma 2. Hence $T(n)$, the time required to build a 4-way polygon tree for n points, satisfies the recurrence

$$T(n) \leq O(n^2 + n \cdot \log n) + 4 \cdot T(\lceil n/4 \rceil).$$

Thus $T(n) = O(n^2)$ by Theorem 2.1.3.4.

b) Let R be a polygonal region with s sides. We triangulate R (cf. Section 8.4.2) and compute $R' \cap S$ separately for each of the $s-1$ triangles R' in the triangulation. It therefore suffices to show that $A' = R' \cap S$ can be computed in time $O(n^{\log 3/\log 4} + |A'|)$ for a triangle R'. This shows that we may assume w.l.o.g. that R is a triangle.

We describe the search algorithm next. The search reaches only nodes v of the polygon tree T' with $Reg(v) \cap R \neq \emptyset$. Let us assume inductively that when the search reaches node v we have determined $Reg(v) \cap e_i$, $1 \leq i \leq 3$, for each of the three sides of triangle R. Note that $Reg(v)$ is convex and hence $Reg(v) \cap e_i$ is a line segment. Also note that $Reg(v) \subseteq R$ iff $Reg(v) \cap e_i = \emptyset$ for $1 \leq i \leq 3$ or $Reg(v) \subseteq e_i$ for some i (recall that we assume $Reg(v) \cap R \neq \emptyset$). If $Reg(v) \subseteq R$ then the search proceeds to all six (two, if $Reg(v)$ is one-dimensional) sons of v and clearly $Reg(w) \subseteq R$ for all sons w of v.

The case $Reg(v) \not\subseteq R$ is slightly more complicated. Let w be a son of v. Then $Reg(w) = Reg(v) \cap C$ where C is either a line or a cone-shaped region, as indicated in Figure 8. Then $e_i \cap Reg(w) = (e_i \cap Reg(v)) \cap (e_i \cap C)$ and hence $e_i \cap Reg(w)$ is readily computed for $1 \leq i \leq 3$. If $e_i \cap Reg(w) \neq \emptyset$ for some i then certainly $R \cap Reg(w) \neq \emptyset$ and hence the search proceeds to node w. If $e_i \cap Reg(w) = \emptyset$ for all i then the search proceeds to node w iff $c \in R$ where $c = L_1 \cap L_2$ is the intersection of the two lines which are associated with node v. Note that $Reg(w) \subseteq R$ if $c \in R$ and that $Reg(w) \cap R = \emptyset$ if $c \notin R$.

It remains to estimate the complexity of this algorithm. Let T' be the subtree of all nodes visited in the search. It suffices to bound the number of nodes of T'. If $v \in T'$ then $Reg(v) \cap R \neq \emptyset$ and hence either $Reg(v) \subseteq R$ or $Reg(v) \cap R \neq \emptyset$ and $Reg(v) - R \neq \emptyset$. In the former case we have $S(v) \subseteq A$ and hence the number of nodes with $Reg(v) \subseteq R$ is $O(|A|)$. In the latter case there must be an edge e of

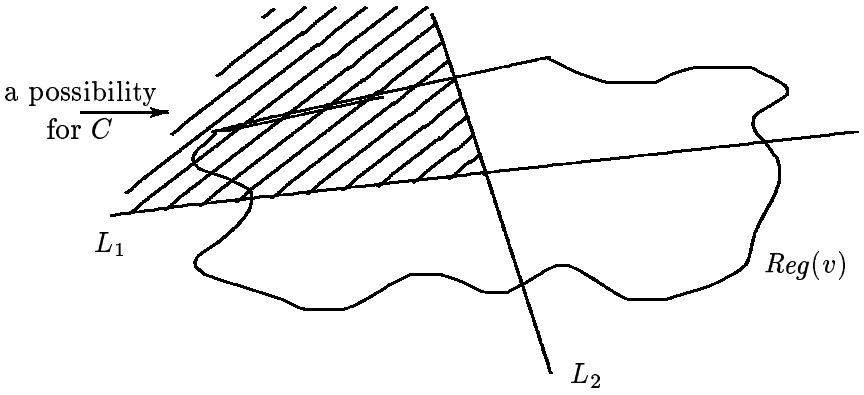


Figure 9.

region R such that $Reg(v) \cap e \neq \emptyset$ but $Reg(v)$ is not contained in e. It therefore suffices to bound t where t is the maximal number of nodes v such that $Reg(v)$ intersects but is not contained in any fixed line segment L.

Claim: $t \leq O(n^{\log 3/\log 4})$.

Proof: Let L be any line segment. Let P_k be the number of primary nodes v, i.e., $Reg(v)$ is not a line segment, of depth k such that $Reg(v) \cap L \neq \emptyset$. Then $P_1 = 1$ and $P_{k+1} \leq 3 \cdot P_k$ since L can intersect at most 3 of the four open regions associated with the sons of any primary node. Thus $P_k \leq 3^k$.

Let v be a primary node of depth k. Then v has two sons x and y which are not primary nodes. We have $S(x) \cup S(y) \subseteq S(v)$ and $|S(v)| \leq \lceil n/4^k \rceil$ since T is an ideal 4-way tree. The argument used in the proof of Theorem 4 shows that there are at most $2 \cdot \log S(x)$ descendants w of x such that $Reg(w)$ intersects L but is not contained in L. The analogous claim holds true for y. Putting both bounds together we conclude that

$$\begin{aligned} t &\leq \sum_{0 \leq k \leq \log n/\log 4} 4 \cdot P_k \cdot \log\lceil n/4^k \rceil \\ &\leq 3^{\log n/\log 4} \cdot \sum_{0 \leq k \leq \log n/\log 4} 8 \cdot 3^{k - \log n/\log 4} \cdot (\log n - 2 \cdot k) \\ &= O(3^{\log n/\log 4}) = O(n^{\log 3/\log 4}) \end{aligned}$$

∎∎

Can we improve upon 4-way polygon trees? Exercise 19 shows that one can always cut any set of n non-collinear points into $2j$ open regions such that any straight line will intersect at most $j+1$ out of these regions and such that no region contains more than $\lceil n/2j \rceil$ points. Here $j \geq 2$ is any integer. Polygon trees based on subdivisions of this form allow us to do polygon retrieval in time $O(n^{\log(j+1)/\log 2j})$. The exponent is minimized for $j = 3$ and is 0.77 in this case.

7.2.2. Range Trees and Multidimensional Divide and Conquer

D-dimensional trees support orthogonal range queries with linear space $O(n)$ and rootic time $O(n^{1-1/d})$. Range trees will allow us to trade space for time. More specifically, we can obtain polylogarithmic query time at the expense of non-linear storage or rootic query time $O(n^{\epsilon \cdot d}$ and space $O((1/\epsilon)^d \cdot n)$ for any $\epsilon > 0$. Also range trees support insertions and deletions in a natural way.

Orthogonal range queries in one-dimensional space are particularly simple. If $S \subseteq U_0$ then any ordinary balanced tree will do. We can compute $S \cap [l_0, h_0]$ by running down two paths in the tree (the search path according to l_0 and the search path according to h_0) and then listing all leaves between those paths. The query time is $O(\log n + |A|)$ and space requirement is $O(n)$. The counting version, i.e., to compute $|S \cap [l_0, h_0]|$, only takes time $O(\log n)$ if we store in every node the number of leaf descendants. This comes from the fact that we have to add up at most $O(\log n)$ counts to get the final answer, namely the counts of all nodes which are sons of a node on one of the two paths and which lie between the two paths. It is very helpful at this point to interpret search trees geometrically. We can view a search tree as a hierarchical decomposition of S into intervals, namely sets $Reg(v) \cap S$. The decomposition process is balanced, i.e., we try to split set S evenly at every step, and it is continued to the level of singleton sets. The important fact is that for every conceivable interval $[l_0, h_0]$ we can decompose $S \cap [l_0, h_0]$ into only $O(\log n)$ pieces from the decomposition. Hence the $O(\log n)$ query time for counting $S \cap [l_0, h_0]$.

This idea readily generalizes into two-dimensional (and d-dimensional space). Let $S \subseteq U_0 \times U_1$. We first project S onto U_0 and build a balanced decomposition of the projection as described above. Suppose now that we have to compute $S \cap ([l_0, h_0] \times [l_1, h_1])$. We can first decompose $[l_0, h_0]$ into $O(\log n)$ intervals. For each of these intervals we only have to solve a one-dimensional problem. This we can do efficiently if we also have data structures for al these one-dimensional problems around. Each one-dimensional problem will cost $O(\log n)$ steps and so total run time is $O((\log n)^2)$. However, space requirement goes up to $O(n \cdot \log n)$ because every point has to be stored in $\log n$ data structures for one-dimensional problems. The details are as follows.

Definition: Let $S \subseteq U_0 \times U_1 \times \cdots \times U_{d-1}$ and let $P = \{i_1, \ldots, i_k\} \subseteq \{0, \ldots, d-1\}$. Then $p(S, P) = \{(x_{i_1}, \ldots, x_{i_k});\ x \in S\}$ is the projection of S onto coordinates P. If $P = \{i\}$ then we also write $p_i(S)$ instead of $p(S, \{i\})$. ∎

Definition: Let $m \in \mathbb{N}$ and let $\alpha \in (1/4, 1 - \sqrt{2}/s)$. m is a slack parameter and α is a weight-balancing parameter. A d-fold range tree for multiset $S \subseteq U_0 \times U_1 \times \cdots \times U_{d-1}$, $|S| = n$ is defined as follows. If $d = 1$ then T is any BB[α]-tree for S. If $d > 1$ then T consists of a BB[α]-tree T_0 for $p_0(S)$. T_0 is called the primary tree. Furthermore, for every node v of T_0 with $depth(v) \in m \cdot \mathbb{Z}$ there is an auxiliary

tree $T_a(v)$. $T_a(v)$ is a $(d-1)$-fold tree for set $p(S(v), \{1, \ldots, d-1\})$. Here $S(v)$ is the set of $x = (x_0, \ldots, x_{d-1}) \in S$ such that leaf x_0 is descendant of v in T_0. ∎

The precise definition of range trees differs in two respects from the informal discussion. First, we do not insist on perfect balance. This will slightly degrade query time but will allow us to support insertions and deletions directly. Also we introduce slack parameter m which we can use to control space requirement and query time.

Lemma 3. *Let $S_m(d,n)$ be the space requirement of a d-fold tree with slack parameter m for a set of n elements. Then $S_m(d,n) = O(n \cdot (c \cdot \log n/m)^{d-1})$ where $c = 1/\log(1/(1-\alpha))$.*

Proof: Note first that the depth of a BB[α]-tree with n leaves is at most $c \cdot \log n$. Thus every point $x \in S$ is stored in the primary tree, in at most $c \cdot \log n/m$ primary trees of auxiliary trees, in at most $(c \cdot \log n/m)^2$ primary trees of auxiliary-auxiliary trees, ... Thus the total number of nodes (counting duplicates) stored in all trees and hence space requirement is

$$O(n \cdot \sum_{n \le i \le d-1} ((c \cdot \log n)/m)^i) = O(n((c \cdot \log n)/m)^{d-1}).$$ ∎

We will use two examples to illustrate the results about range trees: $m = 1$ and $m = \epsilon \log n$ for some $\epsilon > 0$. If $m = 1$ then $S_m(d,n) = O(n \cdot (c \log n)^{d-1})$ and if $m = \epsilon \log n$ then $S_m = O((c/\epsilon)^{d-1} \cdot n)$.

Lemma 4. *Ideal d-fold range trees, i.e., $|S(x)| \le \lceil S(v)/2 \rceil$ for all nodes v (primary or otherwise) and sons x of v, can be constructed in time $O(d \cdot n \cdot \log n + n \cdot ((\log n)/m)^{d-1})$. Here m is the slack parameter.*

Proof: We start by sorting S d-times, once according to the 0-th coordinate, once according to the first coordinate, ... This will take time $O(d \cdot n \log n)$. Let $T_m(d,n)$ be the time required to build an ideal d-fold tree for a set of n elements if S is sorted according to every coordinate. We will show that $T_m(d,n) = O(n \cdot ((\log n)/m)^{d-1})$. This is clearly true for $d = 1$ since $O(n)$ time suffices to build an ideal BB[α]-tree from a sorted list. For $d > 1$ we construct the primary tree in time $O(n)$ and we have to construct auxiliary trees of sizes $n_1, \ldots, n_t$. We have $n_1 + \cdots + n_t \le n \cdot (\log n)/m$ since every point is stored in $(\log n)/m$ auxiliary trees. Note that the primary tree has depth $\log n$ since it is ideal. Hence

$$\begin{aligned} T_m(d,n) &= O(n) + \sum_i T_m(d-1, n_i) \\ &= O(n) + O(\sum_i n_i \cdot (\log n/m)^{d-2}) \\ &= O(n \cdot (\log n/m)^{d-1}). \end{aligned}$$ ∎

If $m = 1$ then ideal d-fold trees can be constructed in time $O(n \cdot (\log n)^{\max(1,d-1)})$ and if$m = \epsilon \log n$ they can be constructed in time $O(d \cdot n \cdot \log n)$.

Lemma 5. *Let $Q_m(d,n)$ be the time required to answer a range query in a d-fold tree for a set of n elements. Then $Q_m(d,n) = O(\log n \cdot (c \cdot (2^m/m) \cdot \log n)^{d-1} + |A|)$. Here c and m are as in Lemma 3.*

Proof: The claim is obvious for $d = 1$. So let $d > 1$ and let $R = [l_0, h_0] \times \cdots \times [l_{d-1}, h_{d-1}]$ be an orthogonal range query. We search for l_0 and h_0 in the primary tree T_0. his will define two paths of length at most $c \cdot \log n$ in T_0. Consider one of these paths. There are at most $c \log n$ nodes v such that v is a son of one of the nodes on the paths and v is between two paths. Every such node represents a subset of points of S whose 0-th coordinate is contained in $[l_0, h_0]$. We have to solve $(d-1)$-dimensional problems on these subsets. Let v be any such node and let $v_1, \ldots, v_t$ be the closest descendants of v such that m divides $\mathit{depth}(v_i)$. Then $t \leq 2^{m-1}$ and auxiliary trees exist for all v_i's. Also we can compute $S \cap R$ by forming the union of $S(v_i) \cap ([l_1, h_1] \times \cdots \times [l_{d-1}, h_{d-1}])$ over all v_i's. Since the number of v_i's is bounded by $2 \cdot c \cdot ((\log n)/m) \cdot 2^{m-1}$ we have:

$$Q_m(d,n) \leq c \cdot (2^m/m) \cdot \log n \cdot Q_m(d-1,n) + |A|.$$

This proves Lemma 5. ∎

If $m = 1$ then $Q_m(d,n) = O(\log n \cdot (2 \cdot c \cdot \log n)^{d-1} + |A|)$ and if $m = \epsilon \log n$ then $Q_m(d,n) = O(\log n \cdot (c/\epsilon)^{d-1} \cdot n^{\epsilon \cdot d})$.

We close our discussion of range trees by discussing insertion and deletion algorithms. We will show that the amortized cost of an insertion or deletion is polylogarithmic. Suppose that point $x = (x_0, x_1, \cdots, x_{d-1})$ has to be inserted (deleted). We search for x_0 in the primary tree and insert or delete it whatever is appropriate. This has cost $O(\log n)$. Furthermore, we have to insert x into (delete x from) at most $(c \cdot \log n)/m$ auxiliary trees, $((c \cdot \log n)/m)^2$ auxiliary-auxiliary trees, ... Thus the total cost of an insertion or deletion is $O(\log n \cdot (c \log n/m)^{d-1})$ not counting the cost for rebalancing. Rebalancing is done as follows. For every (primary or auxiliary or auxiliary-auxiliary or ...) tree into which x is insert (from which x is deleted) we find a node v of minimal depth which goes out of balance. We replace the subtree rooted at v by an ideal d'-fold tree for the set $S(v)$ of descendants of v. Here $d' = d$ if v is a node of the primary tree of an auxiliary tree, ... ; $d - d'$ is called the level of node v. This will take time $O(d' \cdot q \cdot \log q + q \cdot (\log q/m)^{d'-1})$ by Lemma 4 where $q = |S(v)|$. Rebalancing on the last level ($d' = 1$) is done differently. On level 1 we use the standard algorithm for rebalancing BB[α]-trees.

Worst case insertion/deletion cost is now easily computed. It is $O(d^2 \cdot n \cdot \log n + n \cdot (\log n/m)^{d-1})$, essentially the cost of constructing a new d-fold tree from scratch. Amortized insertion/deletion cost is much smaller as we demonstrate next. We use Theorem III.5.1.4 to obtain a polylogarithmic bound on amortized insertion/deletion cost.

Note first that a point x is inserted into (deleted from) at most $((c \cdot \log n)/m)^{d-1}$ trees of level 1 for a (worst case) cost of $O(\log n)$ each. Thus total rebalancing cost on level 1 is $O(\log n \cdot (c \cdot \log n/m)^{d-1})$.

We next consider levels l, $2 \leq l$ *led.* We showed (Lemmas 2 and 3 in the proof of Theorem III.5.1.4) that the total number of rebalancing operations caused by nodes v at level l with $1/(1-\alpha)^i \leq |S(v)| \leq 1/(1-\alpha)^{i+1}$ during the first n insertions/deletions is $O(TA_{i,l} \cdot (1-\alpha)^i)$, where $TA_{i,l}$ is the total number of transactions which go through nodes v at level l with $1/(1-\alpha)^i \leq |S(v)| \leq 1/(1-\alpha)^{i+1}$; here $0 \leq i \leq c \cdot \log n$. The cost of a rebalancing operation caused by such node v is $O(l \cdot (1-\alpha)^{-(i+1)} \cdot (i+1) + ((i+1)/m)^{l-1})$ by Lemma 2. Also $TA_{i,l} \leq n \cdot ((c \cdot \log n)/m)^{d-l}$ by a simple induction on l starting with $l = d$. Thus total rebalancing cost at levels $l \geq 2$ is at most

$$\sum_{2 \leq l \leq d} \sum_{0 \leq i \leq c \cdot \log n} n \cdot ((c \cdot \log n)^{d-l} \cdot (1-\alpha)^i \cdot l \cdot (1-\alpha)^{-(i+1)} \cdot (i+1+((i+1)/m)^{l-1})$$
$$= O(\sum_{2 \leq l \leq d} n \cdot ((c \cdot \log n)^{d-l} \cdot l \cdot ((c \log n)^2 + (c \cdot \log n/m)^l \cdot (m/l))$$
$$= O(n \cdot (m^2 + m \cdot d) \cdot ((c \cdot \log n)/m)^d).$$

We summarize in

Lemma 6. *Amortized insertion/deletion cost in d-fold range trees with slack parameter m is $O((m^2 + m \cdot d) \cdot ((c \cdot \log n)/m)^d)$.*

Proof: By preceding discussion. ∎

Theorem 6. *d-fold range trees with slack parameter $m \geq 1$ and balance parameter $\alpha \in (1/4, 1-\sqrt{2}/2)$ for a set of n elements take space $O(n((c \cdot \log n)/m)^{d-1})$, support orthogonal range queries with time bound $O(\log n \cdot (c \cdot (2^m/m) \cdot \log n)^{d-1} + |A|)$, and have amortized insertion/deletion cost $O((m^2 + m \cdot d) \cdot ((c \cdot \log n)/m)^d)$. Here $c = 1/\log(1/(1-\alpha))$. In particular, we have:*

slack	*space*	*query time*	*insertion/deletion time*
1	$n \cdot (c \cdot \log n)^{d-1}$	$\log n \cdot (2 \cdot c \cdot \log n)^{d-1}$	$d \cdot (c \cdot \log n)^d$
$\epsilon \cdot \log n$	$n \cdot (c/\epsilon)^{d-1}$	$(c/\epsilon)^{d-1} \cdot n^{\epsilon \cdot d} \cdot \log n$	$(c/\epsilon)^d \cdot ((\epsilon \cdot \log n)^2 + d \cdot \epsilon \cdot \log n)$

Proof: Immediate from Lemmas 1 to 6. ∎

Search trees are always examples for divide and conquer. Range trees and dd-trees exemplify a variant of divide and conquer which is particularly useful for multidimensional problems: multidimensional divide and conquer. A problem of size n in d-space is solved by reducing it to two problems of size at most $n/2$ in d-space and one problem of size at most n in $(d-1)$-dimensional space. Range

trees (with slack parameter $m = 1$) fit well into this paradigm. A set of size n is split into two subsets of size $n/2$ each at the root. In addition, an auxiliary tree is associated with the root which solves the $(d-1)$-dimensional range query problem for the entire set. Other applications of multidimensional divide and conquer are dd-trees and polygon trees, domination problems (Exercises 22, 23) and closest point problem (Exercise 24, 25).

In fixed radius near neighbors problem we are given a set $S \subseteq \mathbb{R}^d$ and a real $\epsilon > 0$ and are asked to compute the set of all pairs $(x, y) \in S \times S$ such that $dist_2(x, y) < \epsilon$. Here $dist_2(x, y) = (\sum_{0 \le i < d}(x_i - y_i)^2)^{1/2}$ is the Euclidian or L_2-norm, but similar approaches work for other norms. We denote the set of such pairs by $\epsilon NN(S)$. Of course, $\epsilon NN(S)$ might be as large as n^2, $n = |S|$, if the points of S lie very dense. In most applications dense sets do not arise. We therefore restrict our considerations to sparse sets.

Definition: Let $\epsilon > 0$, $c > 0$. Set $S \subseteq \mathbb{R}^d$ is (ϵ, c)-sparse if for every $x \in \mathbb{R}^d$ we have $|\{y \in S;\ dist_2(x, y) < \epsilon\}| < c$, i.e., any sphere of radius ϵ contains at most c points of S. ∎

If S is (ϵ, c)-sparse then $|\epsilon NN(S)| \le c \cdot n$, i.e., the size of the output is at most linear. We apply the paradigm of multidimensional divide and conquer to solve the fixed radius near neighbors problem.

If $d = 1$ then a simple method will do. Sort set S in time $O(n \cdot \log n)$ and then make one linear scan through (the sorted version of) S. For every point $x \in S$ look at the c preceding points in the linear order and find out which of them have distance at most ϵ from x. In this way, we can produce $\epsilon NN(S)$ in time $O(c \cdot n)$ from the sorted list. Altogether we have an $O(n \cdot \log n + c \cdot n)$ algorithm in one-dimensional space.

if $d \ge 2$ then we project S onto the 0-th coordinate and find the median of the multiset $p_0(S)$ of projected points. Let that median be m. We split S into two sets A and B of $n/2$ points each, namely A contains only points $x \in S$ with $x_0 \le m$ and B contains only points $x \in S$ with $x_0 \ge m$. We apply the algorithm recursively to d-dimensional point sets A and B. This will compute all pairs $(x, y) \in \epsilon NN(S)$ where both points are in either A or B. It remains to compute pairs $(x, y) \in \epsilon NN(S)$ with $x \in A$ and $y \in B$. If $x \in A$, $y \in B$ and $(x, y) \in \epsilon NN(S)$ then x and y both belong to the slab SL of width $2 \cdot \epsilon$ around hyperplane $x_0 = m$, i.e., $SL = \{x = (x_0, \ldots, x_{d-1}) \in S;\ |x_0 - m| < \epsilon\}$. So all we have to do is to solve ϵNN on point set SL. SL is not quite $(d-1)$-dimensional. We make it $(d-1)$-dimensional by projecting the points in SL onto hyperplane $x_0 = m$, i.e., we compute $S' = \{x';$ there is $x = (x_0, \ldots, x_{d-1}) \in SL$ such that $x' = (x_1, \ldots, x_{d-1})\}$. The crucial observation is that S' is still sparse,and that $\epsilon NN(S')$ "contains" $\epsilon NN(SL)$.

Lemma 7.

a) *If S is (ϵ, c)-sparse then S' is $(\epsilon, (1 + 2^d) \cdot c)$-sparse.*

b) *If* $x, y \in SL$ *and* $dist_2(x,y) < \epsilon$ *then* $dist_2(x',y') < \epsilon$.

Proof: a) Consider any point $x' = (m, x_1, \ldots, x_{d-1})$ on hyperplane $x_0 = m$. We have to compute a bound on the number of points in the "strange" sphere $SSPH(x') = \{y \in S;\ |y_0 - m| < \epsilon$ and $(\sum_{1 \le c < d}(x_i - y_i)^2)^{1/2} < \epsilon\}$ with center x' because exactly the projections of the points in $SSPH(x')$ have distance at most ϵ from x' in $(d-1)$-dimensional set S'. It is easy to see (cf. Figure 10 for an illustration in 2-space) that $SSPH(x')$ can be covered with $(1 + 2^d)$ d-dimensional spheres of radius ϵ. Any such sphere can contain at most c points of S and hence $|SSPH(x')| \le (1 + 2^d) \cdot c$. This shows that S' is $(\epsilon, (1 + 2^d) \cdot c)$ sparse.

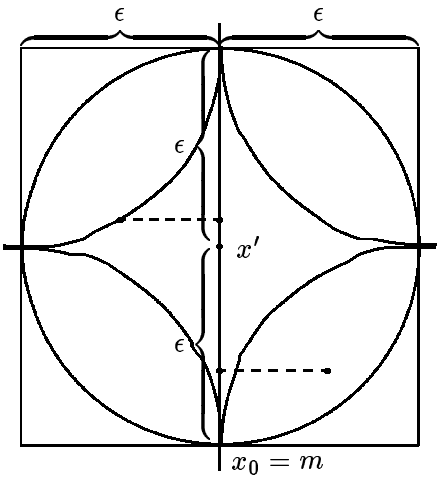


Figure 10. Illustration in 2-space.

b) obvious. ∎

Lemma 7 holds true for other norms as well; however, factor $(1+2^d)$ in 7a) depends on the norm. We infer from Lemma 7 that we can compute $\epsilon NN(SL)$ by solving the $(d-1)$-dimensional problem on S' and then going through list $\epsilon NN(S')$ and throwing out some pairs. This leads to

Theorem 7. *Let* d *be fixed and let* $S \subseteq \mathbb{R}^d$ *be* (ϵ, c)*-sparse. Then* $\epsilon NN(S)$ *can be computed in time*

$$O(n \cdot (\log n)^d/d! + (1 + \hat{c}) \cdot n \cdot (\log n)^{d-1}/(d-2)!),$$

where $\hat{c} = \prod_{2 \le i \le d}(1 + 2^i) \cdot c$ *and* $n = |S|$.

Proof: We will first derive a recurrence on $T(i,n)$, the time to compute $\epsilon NN(S)$ for any (ϵ, c)-sparse set S, $|S| = n$ and $s \subseteq \mathbb{R}^i$, $i \le d$. We have

$$T(i,n) \le 2 \cdot T(i, n/2) + T(i-1, n) + O(n)$$

since in order to solve an i-dimensional problem on n points we spend $O(n)$ time on computing the median and splitting the set and then solve two i-dimensional problems on $n/2$ points each and one $(i-1)$-dimensional problem on at most n points. Also

$$T(i,1) = 0$$

since subproblems of size 1 are trivial and

$$T(1,n) = O(n \cdot \log n + \hat{c} \cdot n)$$

since all one-dimensional problems generated are (ϵ, c)-sparse by Lemma 7a) and therefore can be solved in time $O(n \cdot \log n + \hat{c} \cdot n)$. It is not too hard to verify by induction on n and-i that $T(i,n) = O(n \cdot (\log n)^i/i! + (1+\hat{c}) \cdot n \cdot (\log n)^{i-1}/(i-1)!)$. We leave this for Exercise 25. We will rather show how one arrives at the bound for $T(i,n)$.

Observe first that it suffices to study recurrence

$$\begin{array}{ll} U(i,n) = 2 \cdot U(i,n/2) + U(i-1,n) + n & \text{for } i \geq 2, n \geq 2 \\ U(i,1) = 0 & \text{for } i \geq 1 \\ U(1,n) = n \cdot \log n + \hat{c} \cdot n & \text{for } n \geq 2 \end{array}$$

because we have $T(i,n) = O(U(i,n))$. We solve this recurrence for n a power of two. Let $F(i,k) = U(i,2^k)/2^k$. By substitution we obtain

$$\begin{array}{ll} V(i,k) = V(i,k-1) + V(i-1,k) + 1 & \text{for } i \geq 2, k \geq 1 \\ V(i,0) = 0 & \text{for } i \geq 1 \\ V(1,k) = k + \hat{c} & \text{for } k \geq 1 \end{array}$$

This further simplified by setting $V(i,k) = W(i,k) - 1$. Then

$$\begin{array}{ll} W(i,k) = W(i,k-1) + W(i-1,k) & \text{for } i \geq 2, k \geq 1 \\ W(i,0) = 1 & \text{for } i \geq 1 \\ W(1,k) = k + 1 + \hat{c} & \text{for } k \geq 1 \end{array}$$

If the boundary conditions were simpler, namely all equal to one, then this recursion has a simple combinatoric interpretation. It counts a set of paths. More precisely, if

$$\begin{array}{ll} X(i,k) = X(i,k-1) + X(i-1,k) & \text{for } i \geq 1, k \geq 1 \\ X(i,0) = X(0,k) = 1 & \text{for } i,k \geq 0 \end{array}$$

then $X(i,k)$ is exactly the set of paths from the origin $(0,0)$ to point (i,k) where the set of edges consists of unit length horizontal and vertical lines.

Every path from $(0,0)$ to (i,k) has length (number of edges) $i+k$ and contains exactly i horizontal edges. Hence $X(i,k) = \binom{i+k}{i}$. In particular, $X(1,k) = k+1$. It is now easy to express W in terms of X. Write $W(i,k) = W_1(i,k) + W_2(i,k)$ where

$$\begin{array}{lll} W_j(i,k) = W_j(i-1,k) + W_j(i,k-1) & & j = 1,2,\ i \geq 2,\ k \geq 1 \\ W_1(1,k) = k+1 & W_2(1,k) = \hat{c} & \text{for } k \geq 1 \\ W_1(i,0) = 1 & W_2(i,0) = 0 & \text{for } i \geq 1 \end{array}$$

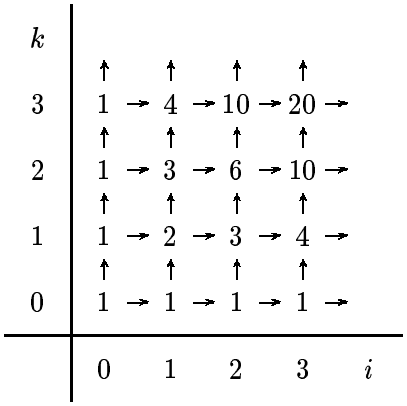


Figure 11. korrigiert,

Then $W_1(i,k) = X(i,k)$ and $W_2(i,k) = \hat{c} \cdot X(i-1,k-1)$ and therefore $W(i,k) = X(i,k) + \hat{c} \cdot X(i-1,k-1)$. Reversing all substitutions we obtain

$$T(i,n) = O\left(n \cdot \left[\binom{i+\log n}{i} + \hat{c} \cdot \binom{i-1+\log n-1}{i-1}\right] - n\right)$$

for n a power of two. Finally using the approximation

$$\binom{a+b}{a} = \frac{b^a}{a!} + \Theta\left(\frac{b^{a-1}}{(a-2)!}\right)$$

for a fixed and b growing we have

$$T(d,n) = O(n \cdot (\log n)^d/d! + (1+\hat{c}) \cdot n \cdot (\log n)^{d-1}/(d-2)!),$$

for n a power of two. It is now tedious but straight forward to verify by induction that this formula holds for all n (Exercise 25). ∎

We will next describe two improvements upon the basic algorithm for the fixed radius near neighbors problem. Presorting, the first improvement, is of general interest and was used already in the proof of Lemma 2; the strategy of finding good dividing lines, the second improvement, helps only in a few situations.

We observed already that the one-dimensional problem can be solved in linear time if set S is sorted, but that it takes time $O(n \log n)$ for general inputs. When we solve a two-dimensional problem we reduce it to a collection of one-dimensional problems of total size $O(n \log n)$. (The recurrence for the total size $S(n)$ of all one-dimensional problems generated from a two-dimensional problem is $S(n) = n + 2 \cdot S(n/2)$ which solves for $S(n) = (n \cdot \log n)$.) We have to sort all these problem instances for a total cost of $O(n \cdot (\log n)^2)$. A better strategy is to sort all of S according to y-coordinate once and then to pull out only sorted subproblems in the divide-step. If we proceed according to this strategy then all one-dimensional problem instances generated are sorted and hence can be solved in linear time. Thus two-dimensional problems can be sorted in time $O(n \cdot \log n)$. This generalizes to

Theorem 8. *Let $d \geq 2$ be fixed and let $S \subseteq \mathbb{R}^d$ be $\epsilon, c)$-sparse. Then $\epsilon NN(S)$ can be computed in time $O((1+\hat{c}) \cdot n \cdot (\log n)^{d-1}/(d-2)!)$ where $n = |S|$ and $\hat{c} = c \cdot \prod_{2 \leq i \leq d}(1+2^i)$.*

Proof: We sort S once according to the last coordinate in time $O(n \log n)$. Then we proceed as described above. With a little care all subproblems generated are also sorted. Hence we obtain the same recurrence as in the proof of Theorem 7 with the only change that $T(1,n) = O(n + \hat{c} \cdot n)$ now. This will save one factor of $\log n$ throughout. ∎

Theorems 7 and 8 derive upper bounds on the performance of a multi dimensional divide and conquer algorithm for the fixed radius near neighbor problem. Are there any sets $S \subseteq \mathbb{R}^d$ where this upper bound is actually achieved? Let us look at the two-dimensional case. If the points of S crowd into a very narrow, say width $< \epsilon$, vertical slab then all subproblems generated will have indeed maximal size and so our algorithm will run very long. A similar observation holds true in higher dimensional space. however, this observation also suggests a major improvement upon the basic algorithm. There is no a-priori reason for only looking at vertical dividing lines, we can also look for horizontal dividing lines and choose whatever is better. A "good" dividing line is a line which divides set S into (nearly) equal parts, defines a small (size $O(n)$) lower dimensional subproblem which is easy to find. Good dividing lines always exist. We content ourselves to a discussion in the two-dimensional space and leave the general case to the reader.

Lemma 8. *Let $S \subseteq \mathbb{R}^2$, $|S| = n$, be (ϵ, c)-sparse. Then there exists a line L orthogonal to one of the axes such that*

1) *no half-space defined by L contains more than $4 \cdot n/5$ points of S;*

2) *the slab of width $2 \cdot \epsilon$ around L contains at most $\sqrt{36 \cdot c \cdot n/5}$ points of S.*

Proof: For i, $i = 0, 1$, let $l_i = \min\{a;\ x_i \leq a$ for at least $n/5$ points of $S\}$ and $h_i = \min\{a;\ x_i \leq a$ for at least $4 \cdot n/5$ points of $S\}$. Next consider lines $L_{ij} = \{y \in \mathbb{R}^2;\ y_i = l_i + (2 \cdot j + 1) \cdot \epsilon\}$, $0 \leq j \leq (h_i - l_i)/2 \cdot \epsilon - 1$, and the slabs of width $2 \cdot \epsilon$ around them, i.e., $SL_{ij} = \{y \in \mathbb{R}^2;\ l_i + 2 \cdot j \cdot \epsilon < y_i < l_i + 2 \cdot (j+1) \cdot \epsilon\}$.

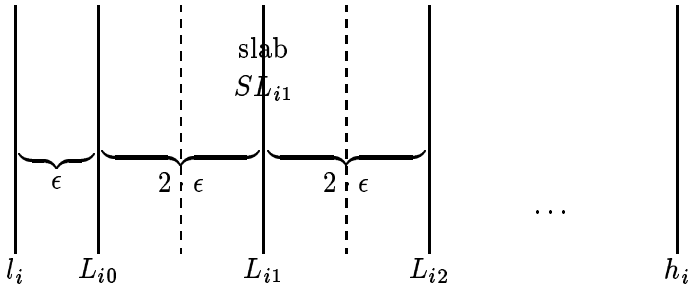


Figure 12.

Claim:

a) *For every i and j: No half-space defined by L_{ij} contains more than $4 \cdot n/5$ points of S.*

b) *For every i: If $(h_i - l_i)/2 \cdot \epsilon \geq \sqrt{n/(20 \cdot c)}$ then there is a j such that $|S \cap SL_{ij}| \leq \sqrt{36 \cdot c \cdot n/5}$.*

c) *There is an i such that $(h_i - l_i)/2 \cdot \epsilon \geq \sqrt{n/(20 \cdot c)}$.*

Proof: a) Since there are $n/5$ points x of S with $x_i < l_i$ there are clearly that many points with $x_i \leq l_i + (2 \cdot j + 1) \cdot \epsilon$. Also here are less than $4 \cdot n/5$ points $x \in S$ with $x_i < h_i$ and hence less than $4 \cdot n/5$ points $x \in S$ with $x_i \leq l_i + (2 \cdot j + 1) \cdot \epsilon < h_i$. This proves a).

b) Slabs SL_{ij}, $j \geq 0$, are pairwise disjoint and contain at most $3 \cdot n/5$ points of S together. If $(h_i - l_i)/2 \cdot \epsilon \geq \sqrt{(1/20 \cdot c) \cdot n}$ then there must be one j such that $|S \cap SL_{ij}| \leq \sqrt{36 \cdot c \cdot n/5}$.

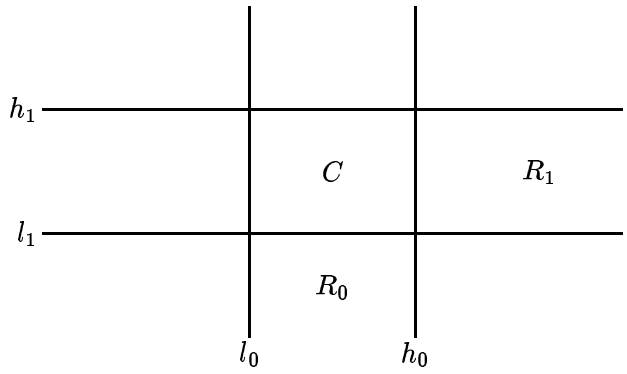


Figure 13. Illustration of part c).,

c) Assume otherwise. Then $(h_i - l_i) < \epsilon \cdot \sqrt{n/5 \cdot c}$ for $i = 0, 1$. Let $R_i = \{y \in \mathbb{R}^2;\ \ l_i \leq y_i \leq h_i\}$ and let $C = R_1 \cap R_2$. Furthermore, let $f = |C \cap S|$ and $n_i = |(R_i - C) \cap S|$. Then $f + n_i \geq 3 \cdot n/5$ since $|R_i \cap S| \geq 3 \cdot n/5$ and $n_0 + n_1 + f \leq n$ since sets $R_0 - C$, $R_1 - C$, C are pairwise disjoint. Thus $n \geq n_0 + n_1 + f = (n_0 + f) + (n_1 + f) - f \geq 6 \cdot n/5 - f$ or $f > n/5$. C is a rectangle whose sides have length at most $\epsilon \cdot \sqrt{n/5 \cdot c}$ and is hence easily covered by $n/5 \cdot c$ circles of radius ϵ. Since S is (ϵ, c)-sparse any such circle contains at most c points of S and hence $f < (n/5 \cdot c) \cdot c = n/5$, a contradiction. ∎

Note that Lemma 8 also suggests a linear algorithm for finding a good dividing line. Compute l_0, h_0, l_1, h_1 in linear time using the linear median algorithm (Section 2.4). Let us assume w.l.o.g. that $(h_0 - l_0) \geq \epsilon \cdot \sqrt{n/5 \cdot c}$. The proof of Lemma 8 shows that one of the slabs $SL_{i,j}$, $0 \leq j \leq \sqrt{n/20 \cdot c}$ contains at most $\sqrt{36 \cdot c \cdot n/5}$ points of S. The number of points in these slabs can be determined in linear time by bucket sort (Section 2.2.2). Thus a good dividing line can be determined in linear

time. We obtain the following recurrence for $T(2,n)$, the time to compute $\epsilon NN(S)$ for an (ϵ, c)-sparse set $S \subseteq \mathbb{R}^2$, $|S| = n$.

$$T(2,n) = \max_{n/5 \le n_1 \le 4 \cdot n/5} T(2,n_1) + T(2, n-n_1) + T(1, \sqrt{36 \cdot c \cdot n/5}) + O(n).$$

Since $T(1,n) = O(n \log n)$ we conclude

$$T(2,n) = \max_{n/5 \le n_1 \le 4 \cdot n/5} (T(2,n_1) + T(2, n-n_1)) + O(n).$$

Theorem 9. *The good dividing line approach to the fixed radius near neighbor problem leads to an $O(n \log n)$ algorithm in 2-dimensional space.*

Proof: In Section 3.5.1 Theorem 2a) we showed that the recurrence above has solution $T(2,n) = O(n \log n)$. ∎

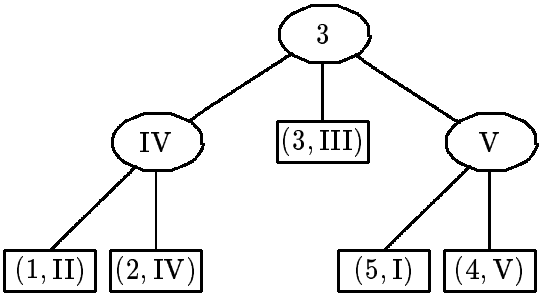
Theorem 9 also holds true in higher-dimensional space. In d-space one can always find a dividing hyperplane which splits S into nearly equal parts (1/5 to 4/5 at the worst) and such that the slab around this hyperplane contains at most $O(n^{1-1/d})$ points. This leads directly to an $O(n \log n)$ algorithm in d-space (Exercise 26).

7.2.3. Lower Bounds

This section is devoted to lower bounds. We cover two approaches. The first approach deals with partial match retrieval in minimum space and shows that rootic search time is the best we can hope for. In particular, we show that dd-trees are an optimal data structure. The second, more general approach deals with a wide class of dynamic multi-dimensional region searching problems. A region searching problem (cf. introduction to 7.2) over universe U is specified by a class $\Gamma \subseteq 2^U$ of regions. We show that the cost of insert, delete and query operations can be bounded from below by a combinatorial quantity, the spanning bound of class Γ. The spanning bound is readily computed for polygon and orthogonal range queries and can be used to show that polygon trees and range trees are nearly optimal.

7.2.3.1. Partial Match Retrieval in Minimum Space

dd-trees are a solution for the partial match retrieval problem with rootic search time and linear space. In fact, dd-trees are a minimum space solution because dd-trees are easily stored as linear arrays. The Figure 14 shows an ideal dd-tree for (invertible) set $S = \{(1, \mathrm{II}), (2, \mathrm{IV}), (3, \mathrm{III}), (4, \mathrm{V}), (5, \mathrm{I})\}$ and its representation as an array. The correspondence between tree and array is the same as for binary search (cf. Section 3.3.1).



1	II
2	IV
3	III
5	I
4	V

Figure 14. An ideal dd-tree.

The aim of this section is to show that dd-trees are an optimum minimum space solution for the partial match retrieval problem; more precisely, we show $\Omega(n^{1-1/d})$ is a lower bound on the time complexity of partial match retrieval in d-dimensional space with one specified component in a decision tree model of computation. The exact model of computation is as follows.

Let S_n be the set of permutations of elements $0, 1, \ldots, n-1$. For $\Pi_1, \ldots, \Pi_{d-1} \in S_n$ let $A(\Pi_1, \ldots, Pi_{d-1}) = \{(i, \Pi_1(i), \ldots, \Pi_{d-1}(i));\ 0 \leq i \leq n\}$ and let $I_n = \{A(\Pi_1, \ldots, \Pi_{d-1});\ \Pi_1, \ldots, \Pi_{d-1} \in S_n\}$. Then $|I_n| = (n!)^{d-1}$. I_n is the class of invertible d-dimensional sets of cardinality n with components drawn from the range $0, 1, \ldots, n-1$. We restrict ourselves to this range because in the decision tree model of computation only the relative size of elements is relevant. A decision tree algorithm for the partial match retrieval problem of size n consists of

1) a storage assignment SA which specifies for every $A \in I_n$ the way of storing A in a table $M[0\,..\,n-1, 0\,..\,d-1]$ with n rows and d columns, i.e., $SA : (S_n)^{d-1} \to S_n$ with the following interpretation. For all $\Pi_1, \ldots, \Pi_{d-1} \in S_n$ and $\Pi = SA(\Pi_1, \ldots, \Pi_{d-1})$: Tuple $(i, \Pi_1(i), \ldots, \Pi_{d-1}(i))$ of set $A(\Pi_1, \ldots, \Pi_{d-1})$ is stored in row $\Pi(i)$ of table M, i.e., $M[\Pi(i), j] = \Pi_j(i)$ for $0 \leq j \leq d-1$, $0 \leq i < n$. Here Π_0 is the identity permutation.

2) d decision trees $T_0, \ldots, T_{d-1}$. Trees T_j are ternary trees. The internal nodes of tree T_j are labelled by expressions of the form $X\ ?\ M[i, j]$ where $0 \leq i < n$. The three edges out of a node are labelled $<$, $=$ and $>$. Leaves are labelled *yes* or *no*.

A decision tree algorithm is used as follows. Let $A \in I_n$, let $y \in \mathbb{R}$ and let $j \in [0\,..\,d-1]$. In order to decide whether there is $x = (x_0, x_1, \ldots, x_{d-1}) \in A$ with $x_j = y$ we store A in table M as specified by SA and then use decision tree T_j to decide the question, i.e., we compare y with elements in the j-th column of M as prescribed by T_j.

Theorem 10. *If $SA, T_0, \ldots, T_{d-1}$ solves the partial match retrieval problem of size n in d-space then there is a j such that depth$(T_j) = \Omega(n^{1-1/d})$, i.e., the worst case time complexity of a decision tree algorithm for the partial match retrieval problem is $\Omega(n^{1-1/d})$.*

Proof: The proof consists of two parts. In the first part we reformulate the problem as a membership problem and in the second part we actually derive a lower bound.

Consider tree T_j. It decides whether there is an tuple $x = (x_0, \ldots, x_{d-1}) \in A$ with $x_j = y$, i.e., it decides membership of y in the projection of A onto the j-th coordinate. It does so by searching in an array, namely the j-the column of table M. The j-th column of table M contains n distinct elements, here integers $0, \ldots, n-1$. The crucial observation is that these n different elements appear in many different orderings. This observation leads to the following definitions.

For $0 \leq j < d$ *let* $OT(j)$ be the set of order types occurring in the j-th column, i.e.,

$$OT(j) = \{\sigma \in S_n;\ \text{ there are } \Pi_1, \ldots, \Pi_{d-1} \in S_n \text{ such that } \sigma = SA(\Pi_1, \ldots, \Pi_{d-1}) \circ \Pi_j^{-1}\}.$$

This definition needs some explanation. Let $\Pi_1, \ldots, \Pi_{d-1} \in S_n$, let $\Pi = SA(\Pi_1, \ldots, \Pi_{d-1})$, and let $A = A(\Pi_1, \ldots, \Pi_{d-1})$. When set S is stored in table M then tuple $(i, \Pi_1(i), \ldots, \Pi_{d-1}(i))$ is stored in row $\Pi(i)$ of table M, i.e., $M[\Pi(i), j] = \Pi_j(i)$. In other words, $M[\Pi(\Pi_j^{-1}(l), j]$ contains integer l, $0 \leq l < n$, i.e., $\Pi \circ \Pi_j^{-1}$ is one of order types occurring in the j-th column.

Lemma 9. *There is a j such that $|OT(jj)| \geq (n!)^{1-1/d}$.*

Proof: The discussion following the definition of $OT(j)$ shows that the mapping $(\Pi_1, \ldots, \Pi_{d-1}) \mapsto (\sigma_0, \ldots, \sigma_{d-1})$ where $\sigma_j = SA(\Pi_1, \ldots, \Pi_{d-1}) \circ \Pi_j^{-1}$ is injective. Hence $\prod_{0 \leq j \leq d-1} |OT(j)| \geq (n!)^{d-1}$. ∎

Next, we describe precisely the computational power of decision trees T_j. Let $\hat{\Pi} \subseteq S_n$ be a set of permutations. A decision tree T solves problem $SST(\Pi)$ — **s**earching **s**emi-sorted **t**ables — if for every $B = \{x_0 < x_1 < \cdots < x_{n-1}\}$, every x and every $\sigma \in \hat{\Pi}$: If B is stored in linear array $M[0\,..\,n-1]$ according to order type σ, i.e., $M[\sigma(l)] = x_l$ for $0 \leq l < n-1$, then T correctly decides $x \in B$.

Lemma 10. *T_j solves $SST(OT(j))$ for $0 \leq j \leq d-1$.*

Proof: Note first that T_j solves $SST(OT(j))$ for every $B = \{x_0 < x_1 < \cdots < x_{n-1}\}$ if it does so for $B = \{0, 1, \ldots, n-1\}$. Next let $\sigma \in OT(j)$. Then there must be $\Pi_1, \ldots, \Pi_{d-1}$ such that $\sigma = SA(\Pi_1, \ldots, \Pi_{d-1}) \circ \Pi_j^{-1}$. In particular, if our partial match retrieval algorithm is applied to set $A = A(\Pi_1, \ldots, \Pi_{d-1})$ then A is stored in table $M[0\,..\,n-1, 0\,..\,d-1]$ such that $M[\sigma(l), j] = l$ for all l; i.e., $B = \{0, \ldots, n-1\}$ is stored in the j-th column of M according to order type σ. Thus T_j solves $SST(OT(j))$. ∎

Lemma 9 and 10 reduce the partial match retrieval problem to the searching semi-sorted tables problem. Lemma 11 gives a lower bound on the complexity of the latter problem.

Lemma 11. *Let $\hat{\Pi} \subseteq S_n$ and let decision tree T solve $SST(\Pi)$.*

a) *For every injective mapping $\sigma : [0..k-1] \to [1..n]$: $|\{\Pi(k);\ \Pi \in \Pi$ and $\hat{\Pi}(i) = \sigma(i)$ for $0 \le i < k\}| \le depth(T)$.*

b) $|\hat{\Pi}| \le depth(T)^n$.

Proof: b) Is a simple consequence of part a). Namely, let $\hat{\Pi}_k = \{\Pi|_{[0..k-1]};\ \Pi \in \hat{\Pi}\}$. Then $|\hat{\Pi}_0| = 1$ and $|\hat{\Pi}_{k+1}| \le depth(T) \cdot |\hat{\Pi}_k|$ by part a). Hence $|\hat{\Pi}| = |\hat{\Pi}_n| \le depth(T)^n$.

a) Let $\sigma : [0..k-1] \to [1..n]$ be injective, let $\Pi \in \hat{\Pi}$ and let $B = \{x_0 < x_1 < \cdots < x_{n-1}\}$ be stored in table $M[0..n-1]$ according to Π. Consider a search for x, $x_{k-1} < x < x_k$. It defines a path in tree T leading to a leaf which is labelled "*no*". On this path x is compared with at most $depth(T)$ distinct table positions, say $M[i_1], \ldots, M[i_h]$, $h \le depth(T)$. We claim $\Pi(k) = i_l$ for some l, $1 \le l \le h$.

Assume otherwise. Then $T[i_l] \ne x_k$ for all l. Consider a search for $x = x_k$. it will lead to exactly the same leaf because the outcome of all comparisons is unchanged. hence T decides that x_k does not belong to B, a contradiction. We have thus shown that $\Pi(k) = i_l$ for some l, $1 \le l \le h \le depth(T)$. ∎

Theorem 10 is now an immediate consequence of Lemmas 1, 2 and 3. By Lemma 1, there is a j with $|OT(j)| \ge (n!)^{1-1/d}$. By Lemma 2, T_j solves $SST(OT(j))$ and hence has depth $|OT(j)|^{1/n}$ by Lemma 3. Finally, $|OT(j)|^{1/n} \ge ((n!)^{1-1/d})^{1/n} = ((n!)^{1/n})^{1-1/d} = \Omega(n^{1-1/d})$ since $n! \approx \sqrt{2 \cdot \pi \cdot n} \cdot (n/e)^n$ by Stirling's approximation. ∎

It is open whether Theorem 1 is also valid for more general models of computation. In particular, it is not known whether the lower bound is valid in a more general decision tree model where comparisons of the form $T[i,j] \ ? \ T[h,j]$ are also allowed. It is conceivable, that comparisons of this form can speed up searches considerably, because they can be used to infer information about the storage assignment. This point is followed up in Exercise 29. We should also emphasize at this point that the restriction to minimum space solutions which is captured in the definition of storage assignment is essential for the argument. After all, range trees provide us with polylogarithmic search time if we are willing to use non-linear space. Exercises 30–32 discuss various extensions.

7.2.3.2. The Spanning Bound

We introduce the spanning bound and use it to prove lower bounds on the complexity of polygon retrieval and orthogonal range queries.

We will first define the region searching problem in an abstract setting. Let U be the key space, let M be a commutative monoid (i.e., a set M with a commutative, associative operation $+ : M \times M \to M$ and an element $0 \in M$ such that $x + 0 = x$

for all $x \in M$) and let $\Gamma \subseteq 2^U$ be a set of regions U. The Γ-region searching problem is to (efficiently) maintain a partial function $S : U \to M$ under the operations

$Insert(x, m)$:	precondition: $x \in \operatorname{dom} S$, $x \in U$, $m \in M$
:	effect: $S \leftarrow S \cup \{(x, m)\}$
$Delete(x)$:	precondition: $x \in \operatorname{dom} S$, $x \in U$
:	effect: $\operatorname{dom} S \leftarrow \operatorname{dom} S - \{x\}$
$Query(R)$:	precondition: $R \in \Gamma$
:	effect: output $\sum_{x \in R \cap \operatorname{dom} S} S(x)$

This is in complete agreement to our previous discussion of searching problems. U is the key space. The problem is to maintain a set of pairs (x, m), where $x \in U$, $m \in M$; m is the "information" associated with key x. *Insert* and *Delete* add and delete pairs and *Query* sums the information over a region R.

Next we fix the model of computation. There is an infinite supply $v_0, v_1, v_2, \ldots$ of variables which take values in M. Initially, 0 is stored in every variable. The instruction repertoire consists $v_i \leftarrow v_j + v_k$, $v_i \leftarrow Input$, $Output \leftarrow v_i$, $i, j, k \geq 0$. Exercise 33 discusses a larger instruction repertoire. A program is given by an (infinite) state space Z, an initial state z_0 inZ corresponding to the empty function S, and three functions f_I, f_D, f_Q. Here $f_I : U \times M \times Z \to Z \times Ins^*$, $f_D : U \times Z \to Z \times Ins^*$ and $f_Q : \Gamma \times Z \to Z \times Ins^*$ where Ins^* is the set of all sequences of instructions from the repertoire. Function f_I has the following semantics. If the algorithm is in state $z \in Z$, operation $Insert(x, m)$ is to be executed, and $f_I(x, m, z) = (z', \sigma)$ then z' is the new state and sequence $\sigma \in Ins^*$ is to be executed. The first instruction of σ is of the form $v_i \leftarrow Input$ and places m into register v_i. The remaining instructions of σ are of the form $v_i \leftarrow v_j + v_k$. The semantics of f_D and f_Q are defined similarly, i.e., after a deletion a sequence of additions is executed and after a query a sequence of additions followed by an output instruction is executed.

A program Z, z_0, f_I, f_D, f_Q is correct if it is correct for all choices of monoid M. It is correct for a particular choice of M if the answers to all queries are computed correctly.

The cost of inserting(x, y) in control state z is the number of instructions in σ, where $(z', \sigma) = f_I(x, m, z)$. The cost of a sequence of operations is the sum of the costs of the operations in the sequence. We use C_n to denote the maximal cost of any sequence of n insertions, deletions and query operations (starting with empty function S).

Example 1 (One-dimensional range trees): Let $U = \mathbb{R}$, $M = (\mathbb{N}_0, +, 0)$, and let Γ be the set of intervals. The set Z of control states is the set of all BB$[\alpha]$-trees T for finite subsets of $\mathbb{R}$, z_0 is the empty tree. Let T be a BB$[\alpha]$-tree. With every node of T we associate a variable v which contains the weight (= number of leaves) in the subtree rooted at that node. An insert or delete requires the update of $O(\log n)$ variables; the update requires only additions if we start updating at the leaves. Also a query can be answered by summing $O(\log n)$ variables. ∎

The basic idea for the lower bound argument is as follows. It is intuitively clear and will be made precise below that every variable contains the sum of $S(x)$ over some subsets of U. A query for region R is then answered by summing some variables, i.e., by assembling $R \cap \operatorname{dom} S$ from smaller pieces. If all queries are "easy" to answer, then set $R \cap \operatorname{dom} S$ can be assembled from only a few pieces for every $R \in \Gamma$. This implies that we need to store information about some $x \in \operatorname{dom} S$ in many (the precise number depends on the structure of Γ) different places. If we delete x at this point then a lot of variables become useless and must be recomputed after inserting x with a different monoid value m. This argument suggests that updates are costly if queries are cheap. The lower bound is then obtained by balancing the cost of the two operations. more generally it suggests that there is a trade-off between query and update cost. In the case of range trees we have seen such a trade-off (as an upper bound) in Section 7.2.2.

Definition:

a) Let $X \subseteq U$, X finite and let $R_1, R_2, \ldots, R_l$ be all sets of the form $X \cap R$, $R \in \Gamma$. Then $F = \{Y_1, \ldots, Y_m\}$, $\emptyset \neq Y_i \subseteq X$, is a **spanning family** for X (with respect to Γ) if
   1) every R_i is the disjoint union of some Y_i's and
   2) every Y_i which is not a singleton is the disjoint union of some Y_j and Y_h.

b) For $F = \{Y_1, \ldots Y_m\}$ a spanning family define

$$t(F) = \max_i \min\{t; \quad \text{there is a representation of } R_i \text{ by } t \text{ disjoint } Y_j\text{'s in } F\}$$

and

$$\rho(F) 0 \max_{x \in X}\{d; \ x \text{ is contained in } d \ Y_j\text{'s}\}.$$

c) For $X \subseteq U$, X finite, let

$$B(X) = \min\{\max(t(F), \rho(F)); \ F \text{ is a spanning family for } X\}$$

and

$$B_n = \max\{B(X); \ X \subseteq U, |X| \leq n\}. \qquad \blacksquare$$

We can now state the main theorem of this section.

Theorem 11. *For every program* $Z, z_0, f_I, f_D, f_Q : C_n \geq \lfloor n/16 \rfloor B_n$.

Proof: We construct a sequence of operations $Op_1, Op_2, \ldots, Op_n$ of total cost at least $\lfloor n/16 \rfloor$. The construction is in three steps. in step one we show that we can restrict attention to normal form programs, in step two we associate the cost of normal form programs with the spanning bound and in step three we finally construct a hard sequence of operations.

Definition: A program Z, z_0, f_I, f_D, f_Q is in normal form if no variable is assigned to twice.

Lemma 12. *For every program there is a normal form program of the same cost.*

Proof: Lemma 12 states that space can be used intentionally wasteful and we all are experts in that. A formal argument goes as follows. Let the normal form program have variables $v'_0, v'_1, \ldots$ and control set $Z' = Z \times W$ where W is the set of finite, injective mappings from $V = \{v_0, v_1, \ldots\}$ to $V' = \{v'_0, v'_1, \ldots\}$. Any sequence σ of instructions is replaced by a sequence of instructions which assigns to unused variables only. Association $w \in W$ is updated accordingly. ∎

We open step two by fixing monoid M. Let M be the set of multi-subsets of $U \times \mathbb{N}$ with operation union. We will only consider sequences of operations $Op_1, \ldots, Op_n$ where each *Insert* is of the form $Insert(x, (x, t))$. In addition, t counts the number of times x was inserted so far. Moreover, x was deleted exactly $(t-1)$-times before it is inseted before the t-th time. Let v be any variable. Then $val(v)$, the value stored in v, is a multi subset of $U \times \mathbb{N}$. $set(v)$ is the projection of $val(v)$ on U.

Let $Op_1, Op_2, \ldots$ be a sequence of Inserts, Deletes and Queries. Let S_h denote function S after execution of $Op_1, \ldots Op_h$. Then $S_h 8x) = (x, t)$ for some t for every $x \in \text{dom}\, S_h$. t is the number of insertions $Insert(x, \ )$ in $Op_1, \ldots, Op_n$. We say that variable v is **useless** at h iff $val(v) \not\subseteq Range(S_h)$. If v is not useless at h then v is **useful** at h.

Lemma 13.

a) *If v is useless at h then v is useless at h' for all $h' \geq h$.*

b) *For every $h : F = \{set(v);\ v$ is useful at h and $set(v) \neq \emptyset\}$ is a spanning family for* dom S_h.

Proof: a) If v is useless at h then $val(v) \not\subseteq Range(S_h)$, i.e., there is a pair $(x, t) \in val(v) - Range(S_h)$. Since $(x, t) \in val(v)$ and $val(v)$ must be a sum of some of the monoid elements assigned to variables after insertions, x was inserted at least t times during $Op_1, \ldots, Op_h$. Since $(x, t) \notin Range(S_h)$ it was also deleted at least t times. Hence $(x, t) \notin Range(S_h)$ for all $h' \geq h$ by our choice of $Op_1, Op_2, \ldots$. Since $val(v)$ will never change we infer that v is useless at all $h' \geq h$.

b) We have to verify properties 1) and 2) of a spanning family. Let us verify property 2) first. If v was assigned by $v \leftarrow Input$, then $set(v)$ is a singleton. Hence if $set(v)$ is not a singleton then v was assigned by $v \leftarrow u + w$ and hence $val(v) = val(u) + val(w)$. Since v is useful at h and hence $val(v) \subseteq Range(S_h)$ we conclude that $set(v) = set(u) \cup set(w)$ and that $set(u) = set(v) \cap set(w) = \emptyset$ (For this inference it is important that we take the monoid of multi-sets and not the monoid of subsets under union). This proves property 2).

Property 1) can be seen as follows. let $R \in \Gamma$ and suppose (for the moment) that $Op_{h+1} = Query(R)$. The answer to this query, i.e., $\sum\{S(x);\ x \in R \cap \operatorname{dom} S_h\}$ is computed as a sum of some variables. Call the set of these variables A. No variables $v \in A$ can be useless at h since $val(v) \not\subseteq Range(S_h)$ implies $\sum\{val(v);\ v \in A\} \not\subseteq Range(S_h)$. Also sets $set(v)$, $v \in A$ must be pairwise disjoint by the argument used to prove property 2). ∎

We are now ready to construct sequence $Op_1, \ldots Op_n$ of cost at least $\lfloor n/16 \rfloor \cdot B_n$. Let $m = \lceil n/2 \rceil$ and let $X = \{x_1, \ldots, x_k\} \subseteq U$, $|X| \leq m$ be such that $B(X) = B_m$. The following program defines $Op_1, \ldots, Op_n$.

a) Let $Op_i = Insert(x_i, (x_i, k1)))$ for $1 \leq i \leq k$

b) **do** $\lfloor n/4 \rfloor$ times

co at this point $F = \{set(v);\ set(v) \neq \emptyset$ and v useful$\}$ is a spanning family for X and hence $B_m = B(X) \leq \max\{\rho(F), t(F)\}$ **oc**

Case 1: $t(F) \geq B_m$:
Then there is $R \in \Gamma$ such that at least B_m elements of F are needed to span $R \cap \operatorname{dom} S = R \cap X$. We let the next operation be $Query(R)$. Answering this query requires to sum at least B_m variables.

Case 2: $\rho(F) \geq B_m$:
Then there is $x \in X$ such that x is contained in at least B_m elements of F. Let the next two operations be $Delete(x)$, $Insert(x, (x, t))$ for the appropriate t. his will make all variables v with $(x, t-1) \in val(v)$ and $set(v) \in F$ useless. There are at least B_m such variables.

It remains to estimate the complexity of sequence $Op_1, Op_2, \ldots, Op_n$ defined above. Let a (b) be the number of times case 1 (2) was executed. Then $a+b \geq \lfloor n/4 \rfloor$. Also the total cost of Case 1 is at least $a \cdot B_m$. In case 2 at least $b \cdot B_m$ variables are made useless. hence at least that many variables must be assigned to. Thus

$$C_n \geq \min_{a+b=\lfloor n/4 \rfloor} \max\{a \cdot B_m, b \cdot B_m\}$$

$$\geq \lfloor n/8 \rfloor \cdot B_m \geq \lfloor n/16 \rfloor \cdot B_n$$

where the last inequality follows from

Lemma 14.

a) *$B_m \leq B_n$ for $m \leq n$.*

b) *$B_{m+n} \leq B_m + B_n$ for all m and n.*

Proof: a) Immediate from the definition.

b) Let $X \subseteq U$, $|X| = m + n$, be such that $B(X) = B_{m+n}$. Let X_1, X_2 be a partition of X with $|X_1| \leq m$, $|X_2| \leq n$. Then there are spanning families F_1 and F_2 for X_1 and X_2 respectively with $\max(t(F_i), \rho(F_i)) \leq B(X_i)$ for $i = 1, 2$. $F = F_1 \cup F_2$ is a spanning family for $X = X_1 \cup X_2$ with $t(F) = t(F_1) + t(F_2)$ and $\rho(F) = \max(\rho(F_1), \rho(F_2))$. Thus $B_{n+m} = B(X) \leq max(t(F), \rho(F)) \leq B(X_1) + B(X_2) \leq B_n + B_m$. ∎∎

The significance of Theorem 11 lies in the fact that it relates the complexity of an algorithm, a quantity which involves time and is therefore difficult to handle, with a purely combinatorial quantity, which is much easier to deal with. Before we apply the spanning bound to orthogonal range queries and polygon retrieval it is helpful to visualize spanning families in terms of graphs.

Let $\Gamma \subseteq 2^U$ be a set of regions and let $X = \{x_1, \ldots, x_n\} \subseteq U$. Let $R_1, R_2, \ldots, R_n$ be all sets of the form $X \cap R$, $R \in \Gamma$ (We may assume w.l.o.g. that the number of sets is equal to the number of points because we can always add either fictious points or regions). Furthermore, let $F = \{Y_1, \ldots, Y_m\}$ be a spanning family. Let us construct a bipartite graph G with node set $\{x_1, \ldots, x_n, R_1, \ldots, R_n\}$ and edge set $E = \{(x_i, R_i);\ x_i \in R_j\}$. For every region R_j let $S_j \subseteq \{1, \ldots, m\}$ be such that R_j is the disjoint union of Y_l, $l \in S_j$.

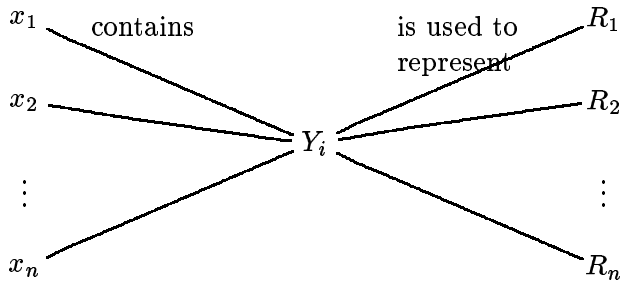


Figure 15.

We can now "factor" graph G into disjoint complete bipartite graphs as follows. For every Y_l consider the complete bipartite graph with nodes $\{x_i;\ x_i \in Y_l\}$ on the X-side and $\{R_j;\ l \in S\}$ on the R-side.

Lemma 15. *E is the disjoint union of the sets $\{(x_i, r_j);\ x_i \in Y_l$ and $l \in S_j\}$, $1 \leq l \leq m$.*

Proof: Let $(x_i, R_j) \in E$, i.e., $x_i \in R_j$. then there is exactly one l such that $x_i \in Y_l$ and $l \in S_j$. ∎

For x_i (R_j) let $\deg(x_i)$ ($\deg(R_j)$) be the degree of x_i (R_j) in the factored graph, i.e., $\deg(x_i) = |\{l;\ x_i \in Y_l\}|$ and $\deg(R_j) = |\{l;\ l \in S_j\}|$. Then $t(F) = \max_j \deg(R_j)$ and $\rho(F) = \max_i \deg(x_i)$. We want to derive lower bounds on $\max(t(F), \rho(F)) = \max_{i,j}(\deg(R_j), \deg(x_i))$ which is certainly no smaller than

$$\left[\sum_i \deg(x_i) + \sum_j \deg(R_j)\right]/2 \cdot n = \left[\sum_l (ldeg(Y_l) + rdeg(Y_l))\right]/2 \cdot n$$

Here $ldeg(Y_l) = |Y_l|$ and $rdeg(Y_l) = |\{j;\ l \in S_j\}|$. It thus suffices to prove lower bounds on the total degree of sets Y_l, $1 \leq l \leq m$.

Application 1: Polygon Retrieval

We consider a special case of polygon retrieval: line retrieval. More precisely, we assume $U = \mathbb{R}^2$ and Γ the set of all lines in $\mathbb{R}^2$, i.e., $\Gamma = \{\{(x_0, x_1) \in \mathbb{R}^2;\ ax_0 + bx_1 = c\};\ a, b, c \in \mathbb{R}, a \neq 0 \text{ or } b \neq 0\}$.

Lemma 16. *Let $S = \{y_1, y_2, \ldots, y_n\} \subseteq \mathbb{R}^2$ and let $L_1, \ldots, L_n$ be a set of n pairwise distinct lines. Let $r_i = |L_i \cap S|$ be the number of points of S on line L_i and let $F = \{Y_1, \ldots, Y_m\}$ be a spanning family for S with respect to Γ. Then*

$$\max(t(F), \rho(F)) \geq \sum_{i=1}^{n} r_i/2 \cdot n.$$

Proof: Consider any Y_l. We claim that $\min(ldeg(Y_l), rdeg(Y_l)) \leq 1$. Assume $ldeg(Y_l) \geq 2$, i.e., there are points y_j, y_h, $j \neq h$, such that $\{y_j, y_h\} \subseteq Y_l$. Since two points determine a line there is at most *one* line L_k such that $Y_l \subseteq L_k$. Thus $rdeg(Y_l) \leq 1$.

Next observe that

$$\begin{aligned} |E| &= \sum_l ldeg(Y_l) \cdot rdeg(Y_l) && \text{[by Lemma 15]} \\ &\leq \sum_l (ldeg(Y_l) + rdeg(Y_l)) && \text{[since } \min(ldeg(Y_l), rdeg(Y_l)) \leq 1\text{]} \\ &\leq 2 \cdot n \cdot \max(t(F), \rho(F)) && \text{[by discussion following Lemma 15.]} \end{aligned}$$

Thus $\max(t(F), \rho(F)) \geq |E|/2 \cdot n = \sum_{i=1}^{n} r_i/2 \cdot n$. ∎

We can now prove a lower bound on the complexity of line retrieval by exhibiting a set of lines of large total rank.

Theorem 12. *The complexity of line retrieval is $\Omega(n^{4/3})$, i.e., there is a sequence of n insertions, deletions and line retrievals of total cost $\Omega(n^{4/3})$.*

Proof: In view of Lemma 16 and Theorem 11 it suffices to construct a set of n points and n lines such that most lines contain many points.

let $A = \lfloor \sqrt{n} \rfloor$ and let $S = [1..A] \times [1..A]$. For integers i, j, a, b let $L(i, j, a, b)$ be the line through points (i, j) and $(i + a, j + b)$. We consider the set L of such lines given by $1 \leq a \leq A^{1/3}$, $1 \leq j \leq A/2$, $1 \leq b \leq a$, $\gcd(a, b)) = 1$.

Claim:

a) *If $(i, j, a, b) \neq (i', j', a', b')$ then lines $L(i, j, a, b)$, $L(i', j', a', b')$ are distinct.*

b) *The number N of lines in L satisfies $A^2 \geq N = \Omega(A^2)$.*

c) The total number of points from S on lines in L is $\Omega(n^{4/3})$.

Proof: a) Assume that the two lines are identical. Then they must have identical slopes and hence $b/a = b'/a'$. Since $\gcd(a,b) = \gcd(a',b') = 1$ we conclude $a = a'$ and $b = b'$. Next, from $(i,j) \in L(i',j',a,b)$ we conclude $(i,j) = (i',j') + x \cdot (a,b)$ for some $x \in \mathbb{R}$. Since $\gcd(a,b) = 1$ we must have $x \in \mathbb{N}$ and hence $i \equiv i' \bmod a$. From $1 \leq i, i' \leq a$ we infer $i = i'$ and hence $j = j'$.

b) The number N of lines is certainly no larger than $A/2 \cdot (A^{1/3})^3 = A^2/2$. Also it is at least

$$\begin{aligned} & A/2 \cdot \sum_{a=1}^{A^{1/3}} a \cdot |\{b;\ \gcd(a,b) = 1 \text{ and } b \leq a\}| \\ & \geq A^{4/3}/2 \cdot \sum_{a=A^{1/3}/2}^{A^{1/3}} |\{b;\ \gcd(a,b) = 1 \text{ and } b \leq a\}| \\ & = \Omega(A^{4/3} \cdot (A^{1/3})^2) = \Omega(A^2) \end{aligned}$$

since $\sum_{a=1}^{m} |\{b;\ \gcd(a,b) = 1 \text{ and } b \leq a\}| = (3/\pi^2) \cdot m^2 + O(m \cdot \log n)$ (cf. G. Hardy, E. Wright: The theory of Numbers, Fourth Edition, Oxford University Press, 1965, p. 265).

c) Every line in L contains at least $(A/2)/A^{1/3} = A^{2/3}/2$ points of S. Thus the total number of points from S on lines in L is $\Omega(A^{8/3})$ by part b) which in turn is $\Omega(n^{4/3})$. ∎

Similar arguments can be used to show lower bounds of the same order for half-space retrieval and circular queries (Exercises 34, 35). The best upper bound $n^{0.77}$ on polygon retrieval is by polygon trees. There is still a gap to close.

Application 2: Orthogonal Range Queries

The lower bound for orthogonal range queries is somewhat harder to obtain. However, there is a merit to that. It agrees with the upper bound.

Theorem 13. *The complexity of orthogonal range queries in $\mathbb{R}^d$ is $\Omega(n \cdot (\log n)^d)$, i.e., for every n there is a sequence of n insertions, deletions and orthogonal range queries of cost at least $\Omega(n \cdot (\log n)^d)$.*

Proof: We will prove a lower bound of order $(\log n)^d$ on the spanning bound. Let $A = \lfloor n^{1/d} \rfloor$, let $X = [1\,..\,A]^d$. Then $|U| = A^d$. Also we consider the following class of A^d "one-sided" range queries. For $y \in X$ let $R_y = \{x \in U;\ x \leq y\}$, where $x \leq y$ if $x_i \leq y_i$ for $0 \leq i \leq d$.

Let $F = \{Y_1, \ldots, Y_m\}$ be a spanning family for X. As above consider the complete bipartite graph associated with Y_l (cf. discusion following proof of Theorem 11), i.e., let $In(Y_l) = \{x \in X;\ x \in Y_l\}$ and let $Out(Y_l) = \{R_y;\ Y_l \text{ is used to represent } R_y, y \in X\}$. Then Y_l contributes all of $In(Y_l) \times Out(Y_l)$ to the bipartite

graph G with edge set $\{(x, R_y);\ x \leq y\}$ associated with the orthogonal range query problem.

The idea for the proof is now as follows. If Y_l contributes many edges to graph G then most edges (x, R_y) contributed by Y_l must have $x \ll y$. This suggests to weight the edges (x, R_y) of G such that the weight is a decreasing function of $y - x$. We can then hope to bound the weight of the edges covered by any Y_l from above and the weight of all edges from below. This would give the bound. What weight function should we choose? It should be symmetric with respect to th coordinates. About the simplest decreasing function with this property is to assign weight

$$w(x, y) = ((y_0 - x_0 + 1)(y_1 - x_1 + 1) \cdots (y_{d-1} - x_{d-1} + 1))^{-1}$$

to edge (x, R_y) for $x \leq y$.

Lemma 17. *For every $Y_l \in F$:*

$$\sum_{\substack{x \in In(Y_l) \\ R_y \in Out(Y_l)}} w(x, y) \leq (2 \cdot \pi)^d \cdot (|In(Y_l)| + |Out(Y_l)|)$$

Proof: Let $m_i = \max\{x_i;\ (x_0, x_1, \ldots x_{d-1}) \in In(Y_l)\}$ and let $B = \{(m_0 - x_0, \ldots, m_{d-1} - x_{d-1});\ (x_0, \ldots x_{d-1}) \in In(Y_l)\}$, $C = \{(y_0 - m_0, \ldots, y_{d-1} - m_{d-1});\ (y_0, \ldots y_{d-1}) \in Out(Y_l)\}$. Then all elements of B and C are non-negative. This is obvious for B and follows for C from the fact that $x \in In(Y_l)$, $R_y \in Out(Y_l)$ implies $x \in R_y$ and hence $x \leq y$. We have

$$\begin{aligned}
\sum_{\substack{x \in In(Y_l) \\ R_y \in Out(Y_l)}} w(x, y) &= \sum_{\substack{u \in B \\ v \in C}} w(m - u, m + v) \\
&= \sum_{u \in B, v \in C} ((u_0 + v_0 + 1) \cdots (u_{d-1} + v_{d-1} + 1))^{-1} \\
&\leq \sum_{u \in B \cup C, v \in B \cup C} ((u_0 + v_0 + 1) \cdots (u_{d-1} + v_{d-1} + 1))^{-1}
\end{aligned}$$

For $i_0 \geq 0$, $i_1 \geq 0$, $\ldots$, $i_{d-1} \geq 0$ let

$$a_{i_0 i_1 \ldots i_{d-1}} = \begin{cases} 1, & \text{if } (i_0, i_1, \ldots, i_{d-1}) \in B \cup C; \\ 0, & \text{otherwise.} \end{cases}$$

Then

$$\sum_{\substack{x\in In(Y_l)\\ R_y\in Out(Y_l)}} w(x,y) \le \sum_{\substack{i_0\ge 0,\ldots,i_{d-1}\ge 0\\ j_0\ge 0,\ldots,j_{d-1}\ge 0}} \frac{a_{i_0 i_1\ldots i_{d-1}}\cdot a_{j_0 j_1\ldots j_{d-1}}}{(i_0+j_0+1)\cdots(i_{d-1}+j_{d-1}+1)}$$

$$\le (2\cdot\pi)^d \cdot \sum_{i_0,\ldots,i_{d-1}\ge 0} a^2_{i_0 i_1\ldots i_{d-1}}$$

$$= (2\cdot\pi))^d \cdot |B\cup C|$$

$$\le (2\cdot\pi)^d\cdot(|B|+|C|) = (2\cdot\pi)^d\cdot(|In(Y_l)|+|Out(Y_l)|)$$

Here the next to last inequality follows from the following fact.

Fact: *Let $a_{i_0\ldots i_{d-1}}, a_{j_0 0 j_1\ldots j_{d-1}} \ge 0$ be d-fold subscripted variables. Then*

$$\sum_{\substack{i_0\ge 0,\ldots,i_{d-1}\ge 0\\ j_0\ge 0,\ldots,j_{d-1}\ge 0}} \frac{a_{i_0 i_1\ldots i_{d-1}}\cdot a_{j_0 j_1\ldots j_{d-1}}}{(i_0+j_0+1)\cdots(i_{d-1}+j_{d-1}+1)} \le (2\cdot\pi)^d\cdot \sum_{i_0,\ldots,i_{d-1}\ge 0} a^2_{i_0 i_1\ldots i_{d-1}}$$

Proof: Case $d=1$ is implied by Hilbert's inequality

$$\sum_{i,j\ge 0} a_i\cdot a_j/(i+j+1) \le \prod \sum_{i\ge 0} a_i^2,$$

cf. G. Hardy, J. Littlewood, G. Polya, Inequalities, Cambridge University Press, 1967, p. 235. The general case can be shown along similar lines. A complete proof can be found in M.L. Fredman, A Lower Bound on the Complexity of Orthogonal Range Queries, JACM 28 (1981), 696–705. ∎∎

Lemma 18. *For all $n\ge 1$, $A=\lfloor n^{1/d}\rfloor$*

$$\sum_{(1,\ldots,1)\le x\le y\le(A,\ldots,A)} w(x,y) = \Omega((A\cdot\log A)^d) = \Omega(n\cdot(\log n)^d)$$

Proof: We have

$$\sum_{(1,\ldots,1)\le x\le y\le(A,\ldots,A)} w(x,y)$$

$$\ge \sum_{\substack{(1,\ldots,1)\le x\le(A/2)\\ (0,\ldots,0)\le y-x\le(A/2,\ldots,A/2)}} ((y_0-x_0+1)\cdots(y_{d-1}-x_{d-1}+1))^{-1}$$

$$= (A/2)^d\cdot\left(\sum_{0\le y_0-x_0\le A/2} 1/(y_0-x_0+1)\right)^d$$

$$= \Omega((A\cdot\log A)^d) = \Omega(n\cdot(\log n)^d)$$ ∎

The proof of Theorem 13 is now easily completed. We have

$$\begin{aligned}
\max(\rho(F), t(F)) &\geq \left(\sum_l |In(Y_l)| + |Out(Y_l)|\right)/2 \cdot n \\
&\qquad \text{[by the discussion following Lemma 15]} \\
&\geq \sum_l \sum_{\substack{x \in In(Y_l) \\ R_y \in Out(Y_l)}} w(x,y)/((2\cdot\pi)^d \cdot 2 \cdot n) \\
&\qquad \text{[by Lemma 17]} \\
&= \sum_{(1,\ldots,1) \leq x \leq y \leq (A,\ldots,A)} w(x,y)/((2\cdot\pi)^d \cdot 2 \cdot n) \\
&= \Omega((\log n)^d) \\
&\qquad \text{[by Lemma 18]}
\end{aligned}$$

We have thus shown an $\Omega((\log n)^d)$ lower bound on the spanning bound of orthogonal range queries. An application of Theorem 2 finishes the proof. ∎

Theorem 13 shows that range trees are optimal. They allow to process n insertions, deletions and queries in time $O(n \cdot (\log n)^d)$ and no data structure can do better.

7.3. Exercises

1) Show that every integer n can be uniquely written as $n = \sum_{i=0}^{k} \binom{a_i}{i}$ where $i - 1 \leq a_i$ and $a_1 < a_2 < \cdots a_k$. [Hint: Use the identity $\sum_{i=0}^{k} \binom{r+i}{i} = \binom{r+k+1}{k+1}$.] Analyze the k-binomial transformation based on this representation.

2) Let $f : \mathbb{N} \to \mathbb{N}$ be any non-decreasing function with $f(i) \geq 2$ for all i. Let S be any set with n elements. Let $i = \lfloor \log n \rfloor$ and let $n - 2^i = \sum_{j \geq 0} a_j b^j$ where $b = f(i)$ and $a_j \in \mathbb{N}_0$ and $0 \leq a_j < b$.

a) Design a dynamization method based on the following representation of set S. S is represented by a large block S_{large} containing 2^i points and structures S_j, $j \geq 0$. S_j contains exactly $a_j \cdot b^j$ points of S.

b) Design a dynamization method based on the following representation of S. S is represented by a large block S_{large} containing 2^i points and structures $S_{j,l}$, $j \geq 0$, $1 \leq l \leq a_j$. A structure $S_{j,l}$ contains exactly b^j points of S.

Determine $Q_D(n)$ and $\bar{I}_D(n)$ in both cases. Reformulate your answers and prove Theorem 3.

3) Reconsider weighted trees as investigated in Section 3.4. Let $\beta_1, \ldots, \beta_n$ be a probability distribution and let $rank(i) = |\{j;\ \beta_j \geq \beta_i\}|$. Is the depth of node i in a weighted tree bounded by $O(rank(i))$?

4) Work out weighted interpolation search in detail. In particular, state precisely under what assumption the $O(\log\log n)$ bound on search time applies to all blocks constructed by weighting.

5) Describe algorithm $Demote(y, a)$ in a weighted dynamic data structure in detail. Analyze its running time.

6) Use weighting to turn sorted arrays + binary search into weighted dictionaries. Do not only support successful but also unsuccessful searches. There are probabilities associated with unsuccessful searches as well, i.e., start with a distribution $(\alpha_0, \beta_1, \ldots, \beta_n, \alpha_n)$ as in Section 3.4. [Hint: Define distribution $\gamma_1, \ldots, \gamma_n$ by $\gamma_i := \beta_i + (\alpha_{i-1} + \alpha_i)/2$ and use ideas similar to the ones used to prove Theorem 7.]

7) Do Exercise 6) for interpolation search.

8) Develop self-organizing (cf. 3.7) data structures for monotone decomposable searching problems. [Hint: Use algorithm *Promote* of Theorem 8 to implement a "Move to first group" or "Move up one group" heuristic. Choose the elements which move down carefully (randomly!).]

9) Let T be an (a, b)-tree (cf. 3.5.2) with n leaves. For a node v let $w(v)$ be the number of leaves in the subtree with root v and let $d(v)$ be the depth of v. Is there a constant $c > 1$ such that $w(v) \leq n/c^{d(v)}$ for all v and T? If not, what does this mean for the dynamization of order decomposable problem based on (a, b)-trees. In particular, is the remark following Lemma 2 valid?

10) Let $VD(S)$ be the Voronoi diagram (cf. 8.3) of point set $S \subseteq \mathbb{R}^2$. Show that Voronoi diagrams can be maintained in time $O(n)$ per insertion and deletion. [Hint: Use order decomposability.]

11) A half-space in $\mathbb{R}^2$ is a set $\{(x, y) \in \mathbb{R}, ax + by \leq c\}$ for some $a, b, c \in \mathbb{R}$. Show that the intersection of n halfspaces can be computed in time $O(n \log n)$ and that the intersection can be maintained under insertions and deletions in time $O((\log n)^2)$ per update. [Hint: The intersection is always a convex polygon. Use order decomposability and the results of Section 8.1.]

12) For $(x_1, y_1), (x_2, y_2) \in \mathbb{R}^2$ let $(x_1, y_1) \leq (x_2, y_2)$ if $x_1 \leq x_2$ and $y_1 \leq y_2$). Show that the maximal elements of a set $S \subseteq \mathbb{R}^2$, $|S| = n$, can be computed in time $O(n \log n)$ and that it can be maintained in time $O((\log n)^2)$ per insertion and deletion.

13) Define weight-balanced dd-trees (as outlined in the remarks following 7.2.1, Theorem 2). Rebalance a weight-balanced dd-tree after an insertion/deletion by replacing the largest subtree which went out of balance by an ideal dd-tree. Show that the amortized cost of an insertion/deletion is $O(8d + \log n) \cdot \log n)$.

14) Are Theorems 2, 3 and 4 true for weight-balanced dd-trees? [Hint: Consider weight-balanced $2d$-trees with $\alpha = 1/4$. Take a tree where every node of even depth has balance $1/4$ and every node of odd depth has balance $1/2$. Consider a partial match query with specified 0-th coordinate. This coordinate is chosen such that the search is always directed into the heavier subtree.]

ub 15 Show that the counting version of partial match retrieval has time complexity $O(n^{1-1/(d-s+1)})$ in ideal dd-trees.

16) Compute function $f(d, d-s)$ of Theorem 3 explicitely. Can you improve upon the argument used to prove Theorem 3 in order to get a better bound on $f(d, d-s)$?

17) Prove Theorem 2.1.4 for arbitrary d.

18) Show that an arbitrary polygon query may have linear running time in an ideal $2d$-tree.

19) A j-way subdivision of the plane consists of two infinite parallel lines L_1, L_2 and half-lines $L_3, L_4, \ldots, L_j$ such that the starting point of L_i lies on L_{i-1}, L_i intersects L_1 and is fully to the right of L_{i-1}. A j-way subdivision divides the plane into $2 \cdot j$ open regions and j one dimensional regions. Show that for every set $S \subseteq \mathbb{R}$, $|S| = n$, not all points of S collinear, there is a j-way subdivision such that $|S \cap R_i| \leq \lceil n/2j \rceil$ for any of the open regions R_i. Discuss polygon trees based on j-way subdivisions and show that they yield $O(s \cdot n^{\log(j+1)/\log 2j})$ retrieval time. Here s is the number of sides of the polygon.

20) Design a static data structure for orthogonal range queries which uses space $O(n^{1+\epsilon})$ for some $\epsilon > 0$ and has query time $O(d \cdot \log n)$. [Hint: Find a hierarchical decomposition of set S into contiguous subsets such that every contiguous subset of S can be found by using only a few pieces.]

21) Base range trees on (a, b)-trees (cf. Section 3.5.2). Reprove some or all of Lemmas 1–4 in Section 2.2.

22) For $x = (x_0, \ldots, x_{d-1})$ and $y = (y_0, \ldots, y_{d-1})$ define $x \leq y$ iff $x_i \leq y_i$ for all i. For $S \subseteq U_0 \times \cdots \times U_{d-1}$ and $x \in S$ let $rank(x) = |\{y \in S;\ y < x\}|$ be the number of points less than x. *rank* is also called the empirical cumulative distribution function. Show that $rank(x)$, $x \in S$ can be computed in time $O(n \cdot (\log n)^{\max(1,d-1)})$. [Hint: Use range trees.]

23) Let $S \subseteq U_0 \times \cdots \times U_{d-1}$ and let $\subseteq$ be defined as in Exercise 22). Show how to compute the set of maxima of S in time $O(n \cdot (\log n)^{\max(1,d-2)})$. [Hint: Use multi-dimensional divide-and-conquer.]

24) Design an algorithm for the fixed radius near neighbors problem with respect to the L_p-Norm, $p > 0$. We have $dist_p(x,y) = \left(\sum_{0 \le i < d} |x_i - y_i|^p\right)^{1/p}$. Cases $p = 1$ and $p = \infty$ are particularly interesting. Here $dist_\infty(x,y) = \max_i |x_i - y_i|$ is the city-block metric.

25) Complete the proof of Theorem 7 of Section 7.2.2.

26) Extend Lemma 6 of Section 7.2.2 to d-space, $d \ge 3$. Use the extension to generalize Theorem 9 to d-space.

27) Study the average case complexity of the ϵ-nearest neighbor problem under the following assumption. S is drawn from $[0,1]^d$ according to the uniform distribution.

28) (Closest Pair). given $S \subseteq \mathbb{R}^d$, find $x, y \in S$ such that $dist(x,y) \le dist(x',y')$ for all $x', y' \in S$, $x' \ne y'$. [Hint: Extend the algorithm for the ϵ-nearest neighbor problem.]

29) (Searching semi-sorted tables): Let $\Pi = \{\pi;$ there is a $j \in [0\,..\,n-1]$ such that $\pi(i) = (i+j) \bmod n$ for all $i\}$. Show a *linear* lower bound for $SST(\Pi)$ in the decision tree model considered in Section 7.2.2.1. Show that $O(\log n)$ comparisons suffice if comparisons of the form $T[h] \ ? \ T[k]$ are permitted! [Hint: use the proof technique of Lemma 3 to prove the lower bound, use comparisons $T[h] \ ? \ T[k]$ to find j for the upper bound.]

30) Extend Section 7.2.3.1, Theorem 1 to an average case lower bound.

31) Let $SA, T_0, \ldots, T_{d-1}$ be a solution for the partial match retrieval problem in the sense of Section 7.2.3.1. Show $\prod_{0 \le i < d} depth(T_i) = \Omega(n^{d-1})$, in particular $depth(T_0) \cdot depth(T_1) = \Omega(n)$ for $d = 2$. Modify dd-trees such that a query with specified 0-th coordinate takes time $O(n^\alpha)$ and a query with specified 1-th coordinate takes time $O(n^{1-\alpha})$. Here $0 < \alpha < 1$.

32) Show that a partial match query with s specified components takes time $\Omega(n^{1-s/d})$ in the worst case in the decision tree model.

33) Show that Section 7.2.3, Theorem 2 stays true if additional instructions $v_i \leftarrow c \cdot v_k$, **if** $v_i = v_j$ **then** $\ldots,$, $i, j \ge 0$, $c \in \mathbb{N}$, are allowed.

34) Show an $n^{4/3}$ lower bound on the complexity of half-space queries. A half-space in $\mathbb{R}^2$ is of the form $\{(x_0, x_1);\ ax_0 + bx_1 \le c\}$ for some $a, b, c \in \mathbb{R}$.

35) Show an $n^{4/3}$ lower bound on the complexity of circular queries, i.e., queries of the form $\{(x_0, x_1);\ (x_0 - a)^2 + (x_1 - b)^2 \leq c\}$.

36) Let $\Gamma \subseteq 2^n$ be a set of regions and let $B : \mathbb{N} \to \mathbb{N}$ be the spanning bound with respect to Γ. Show: there is an algorithm in the sense of Section 7.2.3.2 with $C_n = O(B_n \log n)$. [Hint: Use Section 7.1, Theorem 5 on deletion decomposable searching problems; show that there is a data structure S which supports deletions in time B_n, i.e., $D_s(n) = B_n$, and which can be built in time $n \cdot B_n$, i.e., $P_s(n) = n \cdot B_n$.]

7.4. Bibliographic Notes

Dynamization was introduced by Bentley (79) and later explored by Bentley/Saxe (80) (Theorem 1 and Exercise 1), Overmars/v. Leeuwen (81,81) (Theorems 2, 5 and 6), Mehlhorn/Overmars (81) (Theorem 3) and Mehlhorn (81) (Theorem 4). The section on weighting follows Frederickson (82) and Alt/Mehlhorn (82). Overmars (82) introduced order decomposable problems.

d-dimensional trees were introduced by Bentley (75) and Theorems 1 and 2 of Section 2.1 are taken from there. Weight-balanced dd-trees were discussed by Overmars/v. Leeuwen (82). The analysis of orthogonal range queries (Theorem 4) in dd-trees is taken from Lee/Wong (77). Theorem 3 has not appeared before.

Polygon trees with slack parameter 1 are due to Bentley (79), Luecker (78), and Willard (78). The treatment of range trees with general slack parameter seems to be new. A static trade-off between space and query time is established in Bentley/Maurer-(80). The treatment of multi-dimensional divide and conquer follows Bentley (80); Exercises 22–26 can also be found there. Monier (80) treats recurrences arising in this area.

Section 2.3.1 follows Alt/Mehlhorn/Munro (81); Exercises 29–31 can also be found there. Section 2.3.2 is the work of Fredman (81, 81, 81). Yao (82) proves lower bounds on time/space trade-offs for a similar model of computation.