Chapter 1

EXTERNAL MEMORY DATA STRUCTURES

Lars Arge

Abstract

In many massive dataset applications the data must be stored in space and query efficient data structures on external storage devices. Often the data needs to be changed dynamically. In this chapter we discuss recent advances in the development of provably worst-case efficient external memory dynamic data structures. We also briefly discuss some of the most popular external data structures used in practice.

1. INTRODUCTION

Massive datasets often need to be stored in space efficient data structures on external storage devices. These structures are used to store a dynamically changing dataset such that queries can be answered efficiently. Many massive dataset applications involve geometric data (for example, points, lines, and polygons) or data which can be interpreted geometrically. Such applications often perform queries which correspond to searching in massive multidimensional geometric databases for objects that satisfy certain spatial constraints. Typical queries include reporting the objects intersecting a query region, reporting the objects containing a query point, and reporting objects near a query point.

While development of practically efficient (and ideally also multipurpose) external memory data structures (or *indexes*) has always been a main concern in the database community, most data structure research in the algorithms community has focused on worst-case efficient internal memory data structures. Recently, however, there has been some cross-

*Chapter to appear in *Handbook of Massive Datasets*, J. Abello, P. M. Pardalos, and M. G. C. Resende (Eds.), Kluwer Academic Publishers, 2001. Draft of July 2001.

fertilization between the two areas. In this chapter we discuss recent advances in the development of worst-case efficient external memory data structures. We will concentrate on data structures for geometric objects but mention other structures when appropriate. We also briefly discuss some of the most popular external data structures used in practice.

Model of computation. Accurately modeling memory and disk systems is a complex task (Ruemmler and Wilkes 1994). The primary feature of disks we want to model is their extremely long access time relative to that of internal memory. In order to amortize the access time over a large amount of data, typical disks read or write large blocks of contiguous data at once and therefore the standard two-level disk model has the following parameters (Aggarwal and Vitter 1988, Vitter and Shriver 1994, Knuth 1998):

$$\begin{aligned} N &= \text{number of objects in the problem instance;} \\ T &= \text{number of objects in the problem solution;} \\ M &= \text{number of objects that can fit into internal memory;} \\ B &= \text{number of objects per disk block;} \end{aligned}$$

where $B < M < N$. An *I/O operation* (or simply *I/O*) is the operation of reading (or writing) a block from (or into) disk. Refer to Figure 1.1. Computation can only be performed on objects in internal memory. The measures of performance in this model are the number of I/Os used to solve a problem, as well as the amount of space (disk blocks) used and the internal memory computation time.

Several authors have considered more accurate and complex multi-level memory models than the two-level model. An increasingly popular approach to increase the performance of I/O systems is to use several disks in parallel so work has especially been done in multi disk models. See e.g. the recent survey by Vitter (1999a). We will concentrate on the two-level one-disk model, since the data structures and data struc-

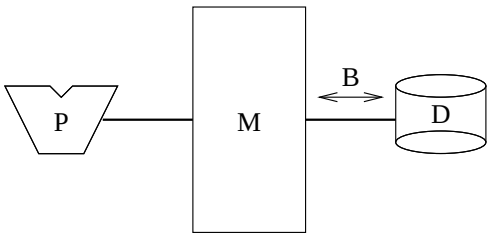

Figure 1.1 Disk model; An I/O moves B contiguous elements between disk and main memory (of size M).

ture design techniques developed in this model often work well in more complex models. For brevity we will also ignore internal computation time.

Outline of chapter. The rest of this chapter is organized as follows. In Section 2. we discuss the B-tree, the most fundamental (one-dimensional) external data structure, as well as recent variants and extensions of the structure. In Section 3. we illustrate some of the important techniques and ideas used in the development of provably I/O-efficient data structures for higher-dimensional problems. We do so through a discussion of a data structure for the stabbing query problem. In Section 4. we discuss external point location and a general method for obtaining a dynamic data structure from a static one. In Section 5. and Section 6. we discuss data structures for 3-sided and general (4-sided) two-dimensional range searching, respectively, and in Section 7. we survey various extensions of these structures. Section 8. contains a survey of external data structures for proximity queries, and in Section 10. we discuss the so-called buffer trees, which can often be used in I/O-efficient algorithms.

Several of the worst-case efficient structures we consider are simple enough to be of practical interest. Still, there are many good reasons for developing simpler (heuristic) and general purpose structures without worst-case performance guarantees, and a large number of such structures have been developed in the database community. Even though the focus of this chapter is on provably worst-case efficient data structures, in Section 9. we give a short survey of some of the major classes of such heuristic-based structures. The reader is referred to recent surveys for a more complete discussion (Agarwal and Erickson 1999, Gaede and Günther 1998, Nievergelt and Widmayer 1997).

Throughout the chapter we assume that the reader is familiar with basic internal memory data structures and design and analysis methods, such as balanced search trees and amortized analysis—see e.g. Cormen et al. (1990).

2. B-TREES

The B-tree is the most fundamental external memory data structure (Bayer and McCreight 1972, Comer 1979, Knuth 1998, Huddleston and Mehlhorn 1982). The B-tree corresponds to an internal memory balanced search tree. It uses linear space—$O(N/B)$ disk blocks—and supports insertions and deletions in $O(\log_B N)$ I/Os. One-dimensional range queries, asking for all elements in the tree in a query interval

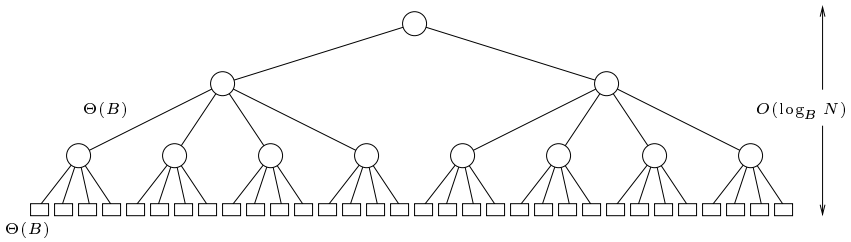

Figure 1.2 B-tree; All internal nodes (except possibly the root) have fan-out $\Theta(B)$ and there are $\Theta(N/B)$ leaves. The tree has height $O(\log_B N)$.

$[q_1, q_2]$, can be answered in $O(\log_B N + T/B)$ I/Os, where T is the number of reported elements.

The space, update, and query bounds obtained by the B-tree are the bounds we would like to obtain in general for more complicated problems. The bounds are significantly better than the bounds we would obtain if we just used an internal memory data structure and virtual memory. The $O(N/B)$ space bound is obviously optimal and the $O(\log_B N + T/B)$ query bound is optimal in a comparison model of computation. Note that the query bound consists of an $O(\log_B N)$ search-term corresponding to the familiar $O(\log N)$ internal memory search-term,[1] and an $O(T/B)$ reporting-term accounting for the $O(T/B)$ I/Os needed to report T elements. Recently, the above bounds have been obtained for a number of problems (e.g. Arge and Vitter 1996, Arge et al. 1999b, Vengroff and Vitter 1996, Agarwal et al. 2000b, Callahan et al. 1995, Govindarajan et al. 2000) but higher lower bounds have also been established for some problems (Subramanian and Ramaswamy 1995, Arge et al. 1999b, Hellerstein et al. 1997, Kanellakis et al. 1996, Koutsoupias and Taylor 1998, Samoladas and Miranker 1998, Kanth and Singh 1999). We discuss these results in later sections.

B-trees come in several variants, like B^+ and B^* trees (see e.g. Bayer and McCreight 1972, Comer 1979, Huddleston and Mehlhorn 1982, Arge and Vitter 1996, Knuth 1998, Agarwal et al. 1999, and their references). A basic B-tree is a $\Theta(B)$-ary tree (with the root possibly having smaller degree) built on top of $\Theta(N/B)$ leaves. The degree of internal nodes, as well as the number of elements in a leaf, is typically kept in the range $[B/2 \ldots B]$ such that a node or leaf can be stored in one disk block. All leaves are on the same level and the tree has height $O(\log_B N)$—refer to Figure 1.2. In the most popular B-tree variants, the N data elements are stored in the leaves (in sorted order) and each internal node holds

[1] We use $\log N$ to denote $\log_2 N$.

$\Theta(B)$ "routing" (or "splitting") elements used to guide searches. As we will see in later sections, it can sometimes be useful to use a B-tree with fan-out $\Theta(B^c)$ for some constant $0 < c \leq 1$. If we keep $\Theta(N/B)$ leaves, every such tree will use $O(N/B)$ space and have height $O(\log_{B^c} N) = O(\log_B N)$.

To answer a range query $[q_1, q_2]$ on a B-tree we first search down the tree for q_1 and q_2 using $O(\log_B N)$ I/Os, and then we report the elements in the $O(T/B)$ leaves between the leaves containing q_1 and q_2. We perform an insertion in $O(\log_B N)$ I/Os by first searching down the tree for the relevant leaf l. If there is room for the new element in l we simply store it there. If not, we *split* l into two leaves l' and l'' of approximately the same size and insert the new element in the relevant leaf. The split of l results in the insertion of a new routing element in the parent of l, and thus the need for a split may propagate up the tree. Propagation of splits can often be avoided by *sharing* some of the (routing) elements of the full node with a non-full sibling. A new (degree 2) root is produced when the root splits and the height of the tree grows by one. Similarly, we can perform a deletion in $O(\log_B N)$ I/Os by first searching for the relevant leaf l and then removing the deleted element. If this results in l containing too few elements we either *fuse* it with one of its siblings (corresponding to deleting l and inserting its elements in the sibling), or we perform a *share* operation by moving elements from a sibling to l. As splits, fuse operations may propagate up the tree and eventually result in the height of the tree decreasing by one.

In internal memory, an N element search tree can be built in optimal $O(N \log N)$ time simply by inserting the elements one by one. In external memory we would use $O(N \log_B N)$ I/Os to build a B-tree using the same method. Interestingly, this is not optimal since Aggarwal and Vitter (1988) showed that sorting N elements in external memory takes $\Theta(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os. We can build a B-tree in the same bound by first sorting the elements and then build the tree level-by-level bottom-up.

2.1 B-TREE VARIANTS AND EXTENSIONS

Recently, several important variants and extensions of B-trees have been considered. In the following we further discuss weight- and level-balanced B-trees, persistent B-trees, as well as string B-trees.

Weight-balanced B-trees. The *weight-balanced B-tree* developed by Arge and Vitter (1996) are similar to normal B-trees in that all leaves are on the same level and rebalancing is done by splitting and fusing nodes. However, instead of requiring the degree of a node to be $\Theta(B^c)$,

we require the weight (or size) of a node v to be $\Theta(B^{ch})$ if v is the root of a subtree of height h. The weight of v is defined as the number of elements in the leaves of the subtree rooted in v. The constraint actually means that v has degree $\Theta(B^c)$ and thus the tree has height $O(\log_B N)$. It also means that the children of v are of approximately the same size $\Theta(B^{c(h-1)})$. In normal B-trees their sizes can differ by a factor exponential in h. Weight-balanced B-trees can be viewed as an external version of BB[α]-trees (Nievergelt and Reingold 1973)—however, weight-balanced B-trees have also been used as a simple alternative to BB[α]-trees in internal memory structures (Arge and Vitter 1996).

After performing an insertion or deletion in a leaf l of a weight-balanced B-tree the weight constraint may be violated in nodes on the path from the root to l. In order to rebalance the tree we perform a split or fuse operation on each of these $O(\log_B N)$ nodes. A key property of a weight-balanced B-tree is that after performing a rebalance operation (split or fuse) on a weight $\Theta(B^{ch})$ node v, $\Theta(B^{ch})$ updates have to be performed below v before another rebalance operation needs to be performed on v. This means that even if the cost of a rebalance operation is $O(B^{ch})$ I/Os, the amortized complexity of an update remains $O(\log_B N)$. The cost of a rebalance operation could for example be $O(B^{ch})$ if v stores a size $\Theta(B^{ch})$ secondary structure that needs to be rebuilt when v splits (for example, a structure on the $\Theta(B^{ch})$ elements below v). The property also suggests a simple rebalancing strategy based on *partial-rebuilding* (see e.g. Overmars 1983); Instead of splitting or fusing nodes on the path from the root to l, we can simply rebuild the tree rooted in the highest unbalanced node on this path. Since the (sub-) tree can be rebuilt in a linear number of I/Os we obtain an $O(\log_B N)$ amortized update bound. Weight-balanced B-trees have been used in numerous efficient data structures, most recently in an elegant so-called *cache-oblivious* B-tree structure by Bender et al. (2000). This structure obtains B-tree-like update and query bounds without explicitly using the (possibly unknown) block size B (see also Frigo et al. 1999). We will discuss other applications in later sections.

Level-balanced B-trees. Apart from the operations discussed above, we sometimes need to be able to perform *divide* and *merge* operations on B-trees. A divide operation at element x constructs two trees containing all elements less than and greater than x, respectively. A merge operation performs the inverse operation. A divide operation can be performed in $O(\log_B N)$ I/Os by first splitting all nodes on the path from the root to the leaf containing x, constructing two trees, and then performing fuse/share operations on the relevant subset of the same

nodes in order to reestablish the B-tree invariant for the two trees. Similarly, a merge operation can also be performed in $O(\log_B N)$ I/Os using $O(\log_B N)$ split/share operations (Mehlhorn 1984).

In some applications we need to be able to traverse a path in a B-tree from a leaf to the root. To do so we need a *parent-pointer* from each node to its parent. Maintaining such pointers during a rebalance operation (split, fuse or merge) on a node v requires $\Theta(B)$ I/Os since we need to update parent pointers of $\Theta(B)$ of v's children. This results in a B-tree update, divide, or merge operation taking $O(B \log_B N)$ I/Os. However, using simple modifications of standard B-trees or weight-balanced B-trees, update operations can still be performed in $O(\log_B N)$ I/Os since it can be guaranteed that $\Theta(B)$ updates have to be performed below a node v between rebalance operations on v.

Recently, Agarwal et al. (1999) developed a variant of B-trees in which divide and merge operations can also be supported I/O-efficiently while maintaining parent pointers. The main idea in the so-called *level-balanced B-trees* is to use a global balance condition instead of the local degree or weight conditions used in B-trees or weight-balanced B-trees. More precisely, a constraint is imposed on the number of nodes on each level of the tree. When the constraint is violated the whole subtree at that level and above is rebuilt. The structure uses $O(N/B)$ space, supports query in $O(\log_B N)$ I/Os, and update, divide, and merge operations in $O(\log_B^2 N)$ I/Os amortized.[2] Level-balanced B-trees e.g. have applications in dynamic maintenance of planar st-graphs (Agarwal et al. 1999).

Persistent B-trees. In some database applications we need to be able to update the current database while querying both the current and earlier versions of the database (data structure). One simple but very inefficient way of supporting this functionality is to copy the whole data structure every time an update is performed. Another and much more efficient way is through the (partially) *persistent* technique (Sarnak and Tarjan 1986, Driscoll et al. 1989), also sometimes referred to as the *multiversion* method (Becker et al. 1996, Varman and Verma 1997). Instead of making copies of the structure, the idea in this technique is to maintain one structure at all times but for each element keep track of the time interval at which it is really present in the structure. A B-tree can be made persistent as follows: Each data element is augmented with an *existence interval* consisting of the time at which the element was inserted and (possibly) the time at which it was deleted. We say

[2] The precise bounds are actually slightly better and more complicated (Agarwal et al. 1999).

that an element is *alive* in its existence interval. All elements are stored in a slightly modified B-tree where we also associate a *node existence interval* with each node. Apart from the normal B-tree constraint on the number of elements in a node, we also maintain that a node contains $\Theta(B)$ alive elements in its existence interval. This means that for a given time t, the nodes with existence intervals containing t make up a B-tree on the elements alive at that time. Thus we can perform range queries in $O(\log_B N + T/B)$ on any version (at any time) of the tree as usual (remembering to disregard dead elements in the visited nodes). Here N is the number of updates performed.

An insertion in a persistent B-tree is performed almost like a normal insertion. We first find the relevant leaf l and if there is room for it we insert the new elements. Otherwise we have an *overflow* and to handle this we first copy all alive elements in l and make the current time the endpoint of the existence interval of l (corresponding to deleting l at the current time). Depending on how many elements we copied, we either construct one new leaf on them, split them into two equal size groups and construct two new leaves on them, or we copy the alive elements from one of l's siblings and construct one or two leaves out of all the copied elements—this corresponds to performing split or fuse operations on the alive elements in l and its sibling. In all cases we make sure that there is room for $\Theta(B)$ future updates in each of the new leaves. We then insert the new element in the relevant leaf and set the start time of the existence interval of all new leaves to the current time. Finally, we insert references to the new leaves in l's parent and (persistently) delete the reference to l. This may result in similar overflow operations cascading up one path to the root of the structure.

In order to perform a deletion we first update the existence interval of the relevant element in leaf l. As the element is not deleted, we do not need to perform a fuse operation as in a normal B-tree. However, the deletion may result in l containing less than the minimum allowed number of alive elements. If this is the case we copy the alive elements from l and one of its siblings and construct one or two new leaves as during an insertion. We also update the references in l's parent as previously, possibly resulting in similar updates up one path of the tree.

Both insertions and deletions can be handled in $O(\log_B N)$ I/Os since in both cases we touch a constant number of nodes on the $O(\log_B N)$ level of the structure. In total we construct $O(N/B)$ leaves since we construct $O(1)$ new leaves only when $\Theta(B)$ updates have been performed on an existing leaf. A similar argument can be applied to the nodes on each level of the tree and thus we can prove that the structure uses $O(N/B)$ space in total.

Several times in later sections we will construct a data structure by performing N insertion and deletions on an initially empty persistent B-tree, and then use the resulting (static) structure to answer queries. Using the above update algorithms, the construction takes $O(N \log_B N)$ I/Os. Utilizing the *distribution-sweeping* technique, Goodrich et al. (1993) showed how to construct the structure (perform the N updates without doing queries) more efficiently in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os. Their method requires that every pair of elements in the structure can be compared—even a pair of elements not present in the structure at the same time. Unfortunately, as we will see in later sections, when working with geometric objects (such as line segments) we will not always be able to compare any two elements. It should be noted that the $O(N \log_B N)$ construction algorithm—that is, the update algorithm described above—also requires every pair of elements to be comparable, since elements can be used as routing elements in the internal nodes of the structure long after they have been deleted. Thus when performing an update or query with element e at time t, we might have to compare e with elements not alive at time t. However, by storing data elements in all nodes of the tree (not just the leaves) and using slightly different update algorithms, we can eliminate this problem such that the $O(N \log_B N)$ algorithm only compares elements present in the structure at the same time (Arge and Teh 2000).

String B-trees. In the B-tree variants discussed so far, the elements—and thus the routing elements in internal nodes—have been of unit size. In string applications a data element (string of characters) can often be arbitrarily long or different elements can be of different length. This means that we cannot use the strings as routing elements and at the same time maintain a large fan-out of internal nodes. We could store pointers to strings in the internal nodes and obtain fan-out $\Theta(B)$ but searching would then be inefficient since we could be forced to perform a lot of I/Os to route a query through a node. Ferragina and Grossi (1995) (see also Ferragina and Grossi 1996) recently presented an elegant solution to this problem called the *string B-tree*. From a high-level point of view, a string B-tree on K strings of total length N is just a B-tree built on N pointers to the N suffixes of the K strings in lexicographical order. To route a query string q through the $\Theta(B)$ string pointers in an internal node, each such node contains a *blind trie* data structure. A blind trie is a variant of the compacted trie (Knuth 1998, Morrison 1968), which fits in one disk block. Routing q through a node v requires one I/O to load the blind trie, as well as some extra I/Os to scan parts of q and the strings corresponding to the pointers stored in v. However, since the scanned

parts of q correspond to parts which will not be scanned again further down the tree, we can charge the I/Os to those parts of q and obtain an optimal $O(\log_B N + |q|/B)$ search bound. Ferragina and Grossi also showed how to insert or delete a string q in $O(|q| \log_B N)$ I/Os amortized. Other results on string B-trees and external string processing have been obtained by Crauser and Ferragina (1999), Ferragina and Luccio (1998), Farach et al. (1998) and Arge et al. (1997).

3. INTERVAL MANAGEMENT

After considering the one-dimensional B-trees, we now turn to data structures for more complicated and higher-dimensional problems like range searching. In internal memory many elegant data structures have been developed for such problems—see e.g. the recent survey by Agarwal and Erickson (1999). Unfortunately, most of these structures are not efficient when mapped to external memory—mainly because they are normally based on binary trees. The main challenge when developing efficient external structures is to use B-trees as base structures, that is, to use multiway trees instead of binary trees. Recently, some progress has been made in the development of provably I/O-efficient data structures based on multi-way trees. In this section we illustrate some of the techniques and ideas used in the development of these structures through the *stabbing query problem*. The stabbing query problem is the problem of maintaining a dynamically changing set of (one-dimensional) intervals such that given a query point q all intervals containing q can be reported efficiently.

The static version of the stabbing query problem (the set of intervals is fixed) can easily be solved I/O-efficiently using a sweeping idea and a persistent B-tree (Arge et al. 1999b, Chazelle 1986, Ramaswamy 1997). To illustrate this, consider sweeping N intervals along the x-axis starting at $-\infty$, inserting each interval in a B-tree when its left endpoint is reached and deleting it again when its right endpoint is reached. To answer a stabbing query with q we simply have to report all intervals in the B-tree at "time" q—refer to Figure 1.3. Thus following the discussion in Section 2., the structure uses $O(N/B)$ space and can be constructed in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os. Queries can be answered in $O(\log_B N + T/B)$ I/Os.

Following earlier attempts of Kanellakis et al. (1996) (see also Subramanian and Ramaswamy 1995, Ramaswamy and Subramanian 1994, Blankenagel and Güting 1990, Icking et al. 1987), a dynamic structure for the problem was developed by Arge and Vitter (1996). This structure can be viewed as an external version of the interval tree (Edelsbrunner

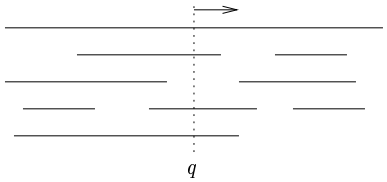

Figure 1.3 Static solution to stabbing query problem using persistence.

1983a;b). It consists of a fan-out $\Theta(\sqrt{B})$ weight-balanced B-tree $\mathcal{T}$ on the endpoints of the intervals (the base tree), with the intervals stored in secondary structures associated with the internal nodes of $\mathcal{T}$ as described below. A range X_v (containing all points below v) can be associated with each node v in a natural way. This range is subdivided into $\Theta(\sqrt{B})$ subranges associated with the children of v. For illustrative purposes we call the subranges *slabs* and the left (right) endpoint of such a slab a *slab boundary*. Refer to Figure 1.4. The $\Theta(\sqrt{B}^2) = \Theta(B)$ contiguous sets of slabs are called *multislabs*. An example of a multislab is $X_{v_2}X_{v_3}$ in Figure 1.4. We assign an interval I to the node v where I contains one or more of the slab boundaries of v but not any of the slab boundaries associated with v's parent. Each node v of $\mathcal{T}$ contains $\Theta(B)$ secondary structures used to store the set of intervals I_v assigned to v; a *left slab list* and a *right slab list* for each of the $\Theta(\sqrt{B})$ slabs, a *multislab list* for each of the $\Theta(B)$ multislabs, as well as an *underflow structure*. A right slab list contains intervals from I_v with the right endpoint in the corresponding slab, sorted according to the right endpoint. Similarly, a left slab list contains intervals with the left endpoint in a slab, sorted according to the left endpoint. A multislab list stores intervals which span the corresponding multislab but not any wider multislab. If the number of intervals stored in a multislab list is less than B we instead store them in the underflow structure. This means that the underflow

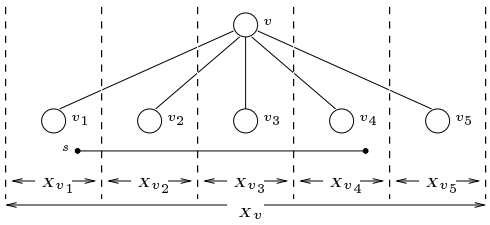

Figure 1.4 Node v in base tree of external interval tree. The range X_v associated with v is divided into 5 slabs.

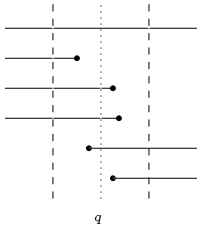

Figure 1.5 Querying a node with q.

structure contains $O(B^2)$ intervals. An interval is thus stored in at most three structures; two slab lists and possibly in a multislab list or the underflow structure. For example, interval s in Figure 1.4 is stored in the left slab list of the first slab and in the right slab list of the fourth slab, as well as in the multislab list corresponding to the second and third slab. Thus the structure uses $O(N/B)$ space.

In order to answer a stabbing query q we search down $\mathcal{T}$ for the leaf containing q, reporting all the relevant intervals among the intervals stored in secondary structures of the node we pass. In node v we report all intervals in multislab lists containing q, as well as all intervals in the underflow structure containing q. We also traverse and report the intervals in the right (left) slab list of the slab containing q from the largest toward the smallest—according to right (left) endpoint—until we meet an interval that does not contain q. No other intervals in the list can contain q—refer to Figure 1.5. If T_v is the number of intervals reported in v we use $O(T_v/B)$ I/Os to report intervals from the slab and multislab lists. There is no $O(\log_B N)$-term since we do not search in any of the lists. If we implement the underflow structure using the static structure based on a persistent B-tree described above, we can also find the relevant intervals in this structure in $O(\log_B B^2 + T_v/B) = O(1 + T_v/B)$ I/Os. Since there are $O(\log_B N)$ nodes on the search path, we in total use $O(\sum_v (1 + T_v/B)) = O(\log_B N + T/B)$ I/Os to answer a query.

To insert a new interval we first use $O(\log_B N)$ I/Os to search down $\mathcal{T}$ for the node where the interval needs to be inserted in secondary structures. In this node we insert the interval in a left and right slab list and possibly in a multislab list. If these lists are implemented using B-trees we can do so in $O(\log_B N)$ I/Os. We may also need to insert the interval in the underflow structure. The structure is static but since it has size $O(B^2)$ we can use a global rebuilding idea to make it dynamic (Overmars 1983); we simply store the update in a special "update block" and once B updates have been collected we rebuild the structure using $O(\frac{B^2}{B} \log_{M/B} \frac{B^2}{B})$ I/Os. Assuming $M > B^2$, that is, that the internal memory is capable of holding B blocks, this is $O(B)$ and we obtain an $O(1)$ amortized update bound. Arge and Vitter (1996) have shown how to make this worst-case, even without the assumption on the main memory size. To complete the insertion, we also need to insert the new endpoints in the base tree $\mathcal{T}$ and rebalance the tree using split and share operations. Performing split or share operations may be costly since they result in the need for restructuring of the secondary structures. However, since this restructuring can be performed in a linear number of I/Os in the size of the secondary structures and as $\mathcal{T}$ is implemented as

a weight-balanced B-tree (Section 2.), we can obtain an $O(1)$ amortized I/O bound for a rebalance operation. Thus in total we can perform an insertion in $O(\log_B N)$ I/Os amortized. The bound can even be made worst-case using standard lazy rebuilding techniques. Deletions can be handled in $O(\log_B N)$ I/Os in a similar way. Variants of the external interval tree structure—as well as experimental results on applications of it in isosurface extraction[3]—have been considered by Chiang and Silva (Chiang and Silva 1997, Chiang et al. 1998, Chiang and Silva 1999).

The above solution to the stabbing query problem illustrates many of the problems encountered when developing I/O-efficient dynamic data structures, as well as the techniques commonly used to overcome these problems. As already discussed, the main problem is that in order to be efficient, external tree data structures need to have large fan-out. In the above example this resulted in the need for what we called multislabs. To handle multislabs efficiently we used the notion of underflow structure, as well as the fact that we could decrease the fan-out of $\mathcal{T}$ to $\Theta(\sqrt{B})$ while maintaining the $O(\log_B N)$ tree height. The underflow structure—implemented using sweeping and a persistent B-tree—solved a static version of the problem on $O(B^2)$ interval in $O(1 + T_v/B)$ I/Os. The structure was necessary since if we had just stored the intervals in multislab lists we might have ended up spending $\Theta(B)$ I/Os to visit the $\Theta(B)$ multislab lists of a node without reporting more than $O(B)$ intervals in total. This would have resulted in an $\Omega(B \log_B N + T/B)$ query bound. We did not store intervals in multislab lists containing $\Omega(B)$ intervals in the underflow structure, since the I/Os spent on visiting such lists during a query can always be charged to the $O(T_v/B)$-term in the query bound. The idea of charging some of the query cost to the output size is often called *filtering* (Chazelle 1986), and the idea of using a static structure on $O(B^2)$ elements in each node has been called the *bootstrapping* paradigm (Vitter 1999a;b). Finally, the ideas of weight-balancing and global rebuilding were used to obtain worst-case efficient update bounds. In Section 5. we will discuss another example of the use of all the above ideas.

4. PLANAR POINT LOCATION

The *planar point location* problem is defined as follows: Given a planar subdivision with N vertices (i.e., a decomposition of the plane into polygonal regions induced by a straight-line planar graph), construct a

[3]Based on a sweeping idea and a persistent list, Agarwal et al. (1998) described an efficient static structure for terrain contour line extraction.

data structure so that the face containing a query point $p = (x, y)$ can be reported efficiently. We will concentrate on the problem of finding the first segment of the subdivision hit by a vertical ray emanating at p (a *vertical ray shooting query*)—refer to Figure 1.6. After answering this query, the face containing the query point can easily be found (Overmars 1985).

In internal memory, a lot of work has been performed on the point location problem—see e.g. the survey by Snoeyink (1997). Sarnak and Tarjan (1986) presented a very simple solution to the static problem based on persistence. Their solution is similar to the static solution to the interval management problem discussed in the previous section. It is based on the fact that a vertical line l imposes a natural order on the segments in the subdivision intersected by l. This means that if we sweep the subdivision from left the right ($-\infty$ to ∞) with a vertical line, inserting a segment in a persistent search tree when its left endpoint is encountered and deleting it again when its right endpoint is encountered, we can answer a point location query $p = (x, y)$ by searching for the position of y in the tree at "time" x. Note that in this method the elements (segments) present in the persistent structure at different times cannot necessarily be compared. As discussed in Section 2., this means that we cannot use the $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ algorithm of Goodrich et al. (1993) to construct the same structure in external memory but have to use the less efficient $O(N \log_B N)$ I/O algorithm. However, we do obtain a linear space external point location data structure that answers queries in $O(\log_B N)$ I/Os. Goodrich et al. (1993) discussed another $O(\log_B N)$ query data structure based on a parallel fractional cascading technique by Tamassia and Vitter (1996). They did not analyze how many I/Os are needed to construct the structure. Several structures which can answer a batch of queries I/O-efficiently have also been proposed (Goodrich et al. 1993, Arge et al. 1995; 1998, Crauser et al. 1998, Vahrenhold and Hinrichs 2000)

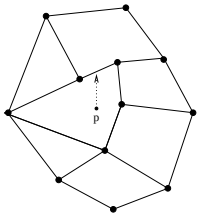

Figure 1.6 Vertical ray shooting query with p.

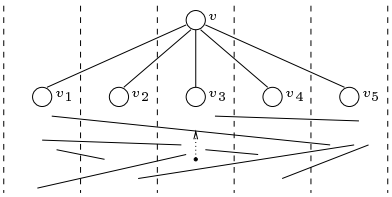

Figure 1.7 Answering a query on segments in I_v. Answer can be in two slab lists and $O(B)$ multislab lists.

Recently, progress has been made in the development of I/O-efficient dynamic point location structures. In the dynamic problem we can change the subdivision dynamically (insert and delete edges/segments and vertices). Agarwal et al. (1999) developed a dynamic structure for *monotone* subdivisions and Arge and Vahrenhold (2000) developed a structure for the general problem. Both structures are based on the external interval tree structure described in the previous section. The main idea is to store the segments of the subdivision (or rather their projection onto the x-axis) in a structure very similar to an interval tree. Doing so a query with $p = (x, y)$ can be answered similarly to a stabbing query with x, except that in each node v visited by the query procedure a ray shooting query is answered on the segments in I_v. The global ray shooting query can then be answered by choosing the lowest segment among the $O(\log_B N)$ segments found this way. We can answer a query on the segments in I_v by answering the query on the segments in two slab lists and $O(B)$ multislab lists (refer to Figure 1.7). Using ideas also utilized in several internal memory structures (Cheng and Janardan 1992, Baumgarten et al. 1994), we can answer queries on the slab lists in $O(\log_B N)$ I/Os with a slightly modified B-tree (Agarwal et al. 1999). It is also easy to answer a ray shooting query on a multislab list in $O(\log_B N)$ I/Os using a B-tree storing the segments in y-order. However, if we query each of the $\Theta(B)$ multislab lists individually we will end up using $O(B \log_B N)$ I/Os to answer the query in v. Agarwal et al. (1999) improved this to $O(\log_B N)$, obtaining an overall query bound of $O(\log_B^2 N)$, by storing the segments in all multislab lists in one combined structure as described below.

Given two segments in the same multislab list we can easily determine which segment is above the other (formally a segment s is above a segment t if there exists a vertical line l intersecting both s and t such that the intersection between l and s is above the intersection between l and t). On the other hand, two segments in different multislab lists might not be comparable (if they cannot be intersected by the same vertical line) and therefore we cannot just build a B-tree on the segments in all multislab lists of a node v and use that to answer a query. Agarwal et al. (1999) used the fact that the segments only have endpoints on $\Theta(\sqrt{B})$ different lines (we imagine cutting the segments in the multislabs at slab boundaries) to construct an efficient structure. They also used that, as shown by Arge et al. (1995), N segments in the plane can be sorted in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os—a set of segments is sorted if for any two comparable segments s and t, if s is above t then s appears after t in the sorted order. More precisely, the *multislab list structure* is constructed as follows: Let $\mathcal{R}$ denote the sorted set of multislab segments in a node.

We first construct a fan-out $\sqrt{B}$ B-tree on $\mathcal{R}$. For a node w in the tree, let $\mathcal{R}_w$ denote the subsequence of $\mathcal{R}$ stored in the subtree rooted at w. To guide the processing of queries, we store certain segments of $\mathcal{R}_w$ in each internal node w; let $w_1, \ldots, w_{\sqrt{B}}$ denote the children of an internal node w. For $1 \leq i, j \leq \sqrt{B}$, we define μ_{ij} to be the maximal segment of $\mathcal{R}_{w_i}$ that intersects the jth vertical slab. We store all $\Theta(B)$ segments μ_{ij} at w in $O(1)$ blocks. In this way the structure requires $O(N/B)$ space and can be constructed in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os. To answer a query with $p = (x, y)$, we follow a path from the root to a leaf z of the B-tree so that $\mathcal{R}_z$ contains the result of the query. At each node w visited by the procedure we do the following: If p lies in the interior of the rth slab, we define $E_w = \{\mu_{ir} \mid 1 \leq i \leq \sqrt{B}\}$. The definition of μ_{ij} ensures that if μ_{ir} is the first (lowest) segment of E_w intersected by an upward ray emanating in p, then the tree rooted at w_i contains the first segment of $\mathcal{R}$ hit by an upward ray emanating in p. We therefore visit w_i next. In this way a query can be answered in $O(\log_B N)$ I/Os. One way of thinking of the multislab list structure is as a fan-out $\sqrt{B}$ B-tree for each of the $\sqrt{B}$ slabs, all stored in the same structure; When answering a query in the rth slab, E_w of all nodes make up a fan-out $\sqrt{B}$ B-tree on the segments intersecting the slab.

The main problem in making the above point location structure dynamic is making the multislab list structure dynamic. The problem is that inserting a new segment may change the total order $\mathcal{R}$ considerably; refer to Figure 1.8. Agarwal et al. (1999) used special features of monotone subdivisions to limit such changes and obtained an $O(\log_B^2 N)$ multislab list structure update bound. This is also the global update bound since only one multislab list structure needs to be updated when performing an insertion or deletion and since the rest of the structure can be easily updated in $O(\log_B N)$ I/Os using standard B-tree and weight-balanced B-tree techniques. Arge and Vahrenhold (2000) extended the structure to work for general subdivisions. To do so they used a new general dynamization technique discussed in the next subsection. Using this method on the multislab list structure they first developed a dy-

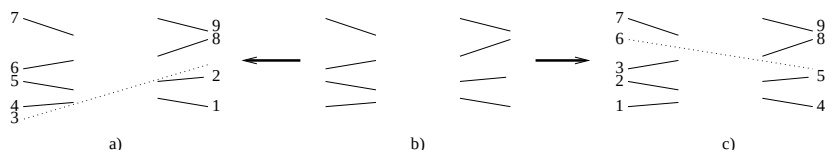

Figure 1.8 Inserting a new segment may change total order significantly. b) Original segments. a) and c) illustrate different insertions and resulting total orders.

namic version of this structure, which supports updates in $O(\log_B N)$ I/Os and answers queries in $O(\log_B^2 N)$ I/Os. Using a technique similar to fractional cascading (Chazelle and Guibas 1986, Mehlhorn and Näher 1990) they improved the query performance to $O(\log_B N)$, obtaining a linear space point location structure supporting updates and queries in $O(\log_B^2 N)$ I/Os.

4.1 THE LOGARITHMIC METHOD

The general method for transforming a static external memory data structure into an efficient dynamic structure is an external version of the *logarithmic method* (Bentley 1979) (see also Overmars 1983). In internal memory, the main idea in this method is to partition the set of N elements into $\log N$ subsets of exponentially increasing size 2^i, $i = 0, 1, 2, \ldots$, and build a static structure $\mathcal{D}_i$ for each of these subsets. Queries are then performed by querying each $\mathcal{D}_i$ and combining the answers, while insertions are performed by finding the first empty $\mathcal{D}_i$, discarding all structures $\mathcal{D}_j$, $j < i$, and building $\mathcal{D}_i$ from the new element and the $\sum_{l=0}^{i-1} 2^l = 2^i - 1$ elements in the discarded structures.

To make the logarithmic method I/O-efficient we need to decrease the number of subsets to $\log_B N$, which in turn means increasing the size of $\mathcal{D}_i$ to B^i. When doing so $\mathcal{D}_j$, $j < i$, does not contain enough objects to build $\mathcal{D}_i$ (since $1 + \sum_{l=0}^{i-1} B^l < B^i$). However, it turns out that if we can build a static structure I/O-efficiently enough, this problem can be resolved and we can make a modified version of the method work in external memory. Consider a static structure $\mathcal{D}$ that can be constructed in $O(\frac{N}{B} \log_B N)$ I/Os and that answers queries in $O(\log_B N)$ I/Os (note that $O(\frac{N}{B} \log_{M/B} \frac{N}{B}) = O(\frac{N}{B} \log_B N)$ if $M > B^2$). We partition the N elements into $\log_B N$ sets such that the ith set has size *less than* $B^i + 1$ and construct an external memory static data structure $\mathcal{D}_i$ for each set—refer to Figure 1.9. To answer a query, we simply query each

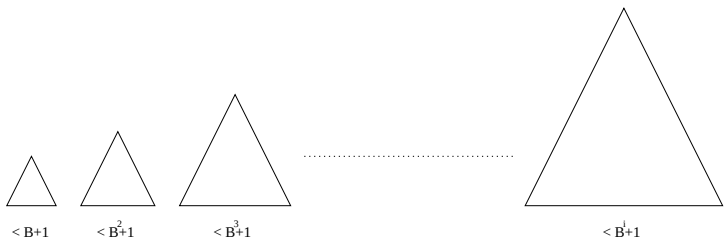

Figure 1.9 Logarithmic method; $\log_B N$ structures—$\mathcal{D}_i$ contains less than $B^i + 1$ elements. $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_j$ do not contain enough elements to build $\mathcal{D}_{j+1}$ of size B^{j+1}.

$\mathcal{D}_i$ and combine the results using $O(\sum_{j=1}^{\log_B N} \log_B |\mathcal{D}_j|) = O(\log_B^2 N)$ I/Os. We perform an insertion by finding the first structure $\mathcal{D}_i$ such that $\sum_{j=1}^{i} |\mathcal{D}_j| \leq B^i$, discarding all structures $\mathcal{D}_j$, $j \leq i$, and building a new D_i from the elements in these structures using $O((B^i/B) \log_B B^i) = O(B^{i-1} \log_B N)$ I/Os. Now because of the way $\mathcal{D}_i$ was chosen, we know that $\sum_{j=1}^{i-1} |\mathcal{D}_j| > B^{i-1}$, which means that at least B^{i-1} objects are moved from lower indexed structures to $\mathcal{D}_i$. If we divide the $\mathcal{D}_i$ construction cost between these objects, each object is charged $O(\log_B N)$ I/Os. Since an object never moves to a lower indexed structure we can at most charge it $O(\log_B N)$ times during N insertions. Thus the amortized cost of an insertion is $O(\log_B^2 N)$ I/Os. Note that the key to making the method work is that the factor of B we lose when charging the construction of a structure of size B^i to only B^{i-1} objects is offset by the $1/B$ factor in the construction bound. Arge and Vahrenhold (2000) show how deletions can also be handled I/O-efficiently using a global rebuilding idea.

5. 3-SIDED PLANAR RANGE SEARCHING

In Section 3. we discussed the stabbing query problem. This problem is equivalent to performing *diagonal corner queries*—a special case of *2-sided range queries*—on a set of points in the plane. Consider mapping an interval $[x, y]$ to the point (x, y) in the plane. Finding all intervals containing a query point q then corresponds to finding all points (x, y) such that $x \leq q$ and $y \geq q$. Refer to Figure 1.10. In this section we consider the more general *3-sided planar range searching* problem: Given a set of points in the plane the solution to a 3-sided query (q_1, q_2, q_3) consists of all points (x, y) with $q_1 \leq x \leq q_2$ and $y \geq q_3$. Refer to Figure 1.11.

Following several earlier attempts (Ramaswamy and Subramanian 1994, Subramanian and Ramaswamy 1995, Blankenagel and Güting 1990, Icking et al. 1987), Arge et al. (1999b) developed an optimal dynamic structure for the 3-sided planar range searching problem. The structure

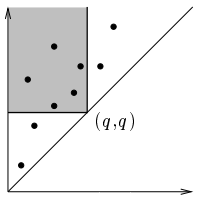

Figure 1.10 Diagonal corner query.

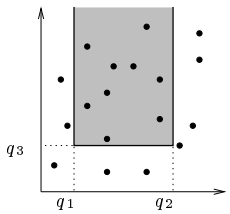

Figure 1.11 3-sided query.

uses many of the ideas already discussed for the interval tree structure in Section 3.: *Bootstrapping* using a static structure, *filtering*, and a *weight-balanced* B-tree. In fact, the I/O-optimal static solution to the problem can be obtained using the same persistence idea as the one used in the interval case. This time we imagine sweeping the plane with a horizontal line from $y = \infty$ to $y = -\infty$ and inserting the x-coordinate of points in a persistent B-tree as they are met. To answer a query (q_1, q_2, q_3) we perform a one-dimensional range query $[q_1, q_2]$ on the B-tree at "time" q_3. Following the discussion in Section 2., the structure obtained this way uses linear space and queries can be answered in $O(\log_B N + T/B)$ I/Os. It can be constructed in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os.

Using the static solution and the general dynamization method discussed in the previous section, we can immediately obtain a dynamic solution to the problem with $O(\log_B^2 N)$ query and update bounds. The optimal dynamic structure however is an external version of the internal memory *priority search tree* structure (McCreight 1985). Like the external interval tree, the external priority search tree consists of a base B-tree on the x-coordinates of the points. As previously, each internal node corresponds naturally to an x-range, which is divided into $\Theta(B)$ slabs by the x-ranges of its children. In each node v we store $O(B)$ points for each of v's $\Theta(B)$ children v_i, namely the B points with the highest y-coordinates in the x-range of v_i (if existing) that have not been stored in ancestors of v. We store the $O(B^2)$ points in the linear space static structure discussed above (the "$O(B^2)$–structure") such that a 3-sided query on them can be answered in $O(T/B)$ I/Os. As in Section 3., we can update the $O(B^2)$–structure in $O(1)$ I/Os using an "update block" and a global rebuilding technique. Since every point is stored in precisely one $O(B^2)$–structure, the structure uses $O(N/B)$ space in total.

To answer a 3-sided query (q_1, q_2, q_3) we start at the root of the external priority search tree and proceed recursively to the appropriate subtrees; when visiting a node v we query the $O(B^2)$–structure and report the relevant points, and then we advance the search to some of the children of v. The search is advanced to child v_i if v_i is either along the leftmost search path for q_1 or the rightmost search path for q_2, or if the entire set of points corresponding to v_i in the $O(B^2)$–structure were reported—refer to Figure 1.12. The query procedure reports all points in the query range since if we do not visit child v_i corresponding to a slab completely spanned by the interval $[q_1, q_2]$, it means that at least one of the points in the $O(B^2)$–structure corresponding to v_i does not satisfy the query. This in turn means that none of the points in the subtree rooted at v_i can satisfy the query. That we use $O(\log_B N + T/B)$ I/Os to answer a query can be seen as follows. In every internal node

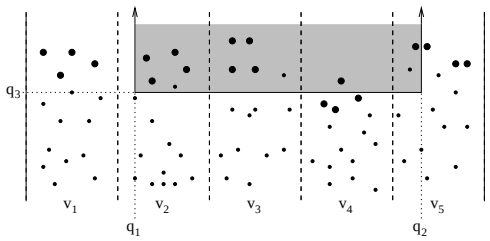

Figure 1.12 Internal node v with children v_1, v_2, ..., v_5. The points in bold are stored in the $O(B^2)$–structure. To answer a 3-sided query we report the relevant of the $O(B^2)$ points and answer the query recursively in v_2, v_3, and v_5. The query is not extended to v_4 because not all of the points from v_4 in the $O(B^2)$–structure satisfy the query.

v visited by the query procedure we spend $O(T_v/B)$ I/Os, where T_v is the number of points reported. There are $O(\log_B N)$ nodes visited on the search paths in the tree to the leaf containing q_1 and the leaf containing q_2 and thus the number of I/Os used in these nodes adds up to $O(\log_B N + T/B)$. Each remaining visited internal node v is not on the search path but it is visited because $\Theta(B)$ points corresponding to it were reported when we visited its parent. Thus the cost of visiting these nodes adds up to $O(T/B)$, even if we spend a constant number of I/Os in some nodes without finding $\Theta(B)$ points to report. Note how we again are using the filtering idea, that is, we are are charging some of our search cost to the points we output as a result of the query.

To insert a point $p = (x, y)$ in the external priority search tree we search down the tree for the leaf containing x, until we reach the node v where p needs to be inserted in the $O(B^2)$–structure. Insertion of p in v (may) result in the $O(B^2)$–structure containing one too many points from the slab corresponding to the child v_j containing x. Therefore, apart from inserting p in the $O(B^2)$–structure, we also remove the point p' with the lowest y-coordinate among the points corresponding to v_j. We insert p' recursively in the tree rooted in v_j. Since we use $O(1)$ I/Os in each of the nodes on the search path, the insertion takes $O(\log_B N)$ I/Os. We also need to insert x in the base B-tree. This may result in split and/or fuse operations and each such operation may require rebuilding an $O(B^2)$–structure. Using weight-balanced B-tress, Arge et al. (1999b) showed how the rebalancing after an insertion can be performed in $O(\log_B N)$ I/Os worst case. Deletions can be handled in $O(\log_B N)$ I/Os in a similar way.

6. GENERAL 2D RANGE SEARCHING

After discussing 2- and 3-sided planar range queries we are now ready to consider general (4-sided) range queries. Given a set of points in the plane we want to be able to find all points contained in a query rectangle. While linear space and $O(\log_B N + T/B)$ query structures exist for the two special cases, Subramanian and Ramaswamy (1995) proved that one cannot obtain an $O(\log_B N + T/B)$ query bound using less than $\Theta(\frac{N/B \log(N/B)}{\log \log_B N})$ disk blocks.[4] This lower bound holds in a natural external memory version of the *pointer machine model* (Chazelle 1990). A similar bound in a slightly different model where the search component of the query is ignored was proved by Arge et al. (1999b). This *indexability model* was defined by Hellerstein et al. (1997) and considered by several authors (Kanellakis et al. 1996, Koutsoupias and Taylor 1998, Samoladas and Miranker 1998). Note that linear space and logarithmic query structures for the *range counting problem* (where only the number of points in the query rectangle, and not the points themselves, need to be reported) can be developed in a slightly different model of computation (see Agarwal et al. 2001a, Zhang et al. 2001, and references therein).

Based on a sub-optimal linear space structure for answering 3-sided queries, Subramanian and Ramaswamy (1995) developed the *P-range tree* that uses optimal $O(\frac{N/B \log(N/B)}{\log \log_B N})$ space but uses more than the optimal $O(\log_B N + T/B)$ I/Os to answer a query. Using their optimal structure for 3-sided queries, Arge et al. (1999b) obtained an optimal structure. We discuss the structure below. In practical applications involving massive datasets it is often crucial that external data structures use linear space. We discuss this further in Section 9. Grossi and Italiano (Grossi and Italiano 1999a;b) developed the elegant linear space *cross-tree* data structure which answers queries in $O(\sqrt{N/B} + T/B)$ I/Os. This is optimal for linear space data structures—as e.g. proven by Kanth and Singh (1999). The *O-tree* of Kanth and Singh (1999) obtains the same bounds using ideas similar to the ones used by van Kreveld and Overmars (1991) in *divided k-d trees*. Below, after discussing the $O(\log_B N + T/B)$ query structure, we also discuss the cross-tree further.

Logarithmic query structure. The $O(\log_B N + T/B)$ query data structure is based on ideas from the corresponding internal memory data structure due to Chazelle (1986). It uses both the external interval tree discussed in Section 3. and the external priority search tree discussed

[4] In fact, this bound even holds for a query bound of $O(\log_B^c N + T/B)$ for any constant c.

in Section 5. The structure consists of a fan-out $\log_B N$ base tree over the x-coordinates of the N points. As usual an x-range is associated with each node v and it is subdivided into $\log_B N$ slabs by v's children $v_1, v_2, \ldots, v_{\log_B N}$. We store *all* the points in the x-range of v in four secondary data structures associated with v. Two of the structures are priority search trees for answering 3-sided queries—one for answering queries with the opening to the left and one for queries with the opening to the right. We also store the points in a linear list sorted by y-coordinate. For the fourth structure, we imagine linking together for each child v_i the points in the x-range of v_i in y-order, producing a polygonal line monotone with respect to the y-axis. We project all the segments produced in this way onto the y-axis and store them in an external interval tree. With each segment endpoint we also store a pointer to the corresponding point in a child node. Since we use linear space on each of the $O(\log_{\log_B N}(N/B)) = O(\log(N/B)/\log\log_B N)$ levels of the tree, the structure uses $O(\frac{N/B \log(N/B)}{\log\log_B N})$ disk blocks in total.

To answer a 4-sided query $q = (q_1, q_2, q_3, q_4)$ we first find the topmost node v in the base tree where the x-range $[q_1, q_2]$ of the query contains a slab boundary. Consider the case where q_1 lies in the x-range of v_i and q_2 lies in the x-range of v_j—refer to Figure 1.13. The query q is naturally decomposed into three parts, consisting of a part in v_i, a part in v_j, and a part completely spanning nodes v_k, for $i < k < j$. The points contained in the first two parts can be found in $O(\log_B N + T/B)$ I/Os using the 3-sided structures corresponding to v_i and v_j. To find the points in the third part we query the interval tree associated with v

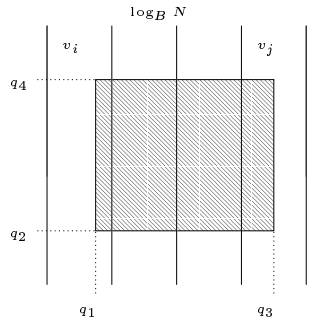

Figure 1.13 The slabs corresponding to a node v in the base tree. To answer a query (q_1, q_2, q_3, q_4) we need to answer 3-sided queries on the points in slab v_i and slab v_j, and a range query on the points in slabs between v_i and v_j.

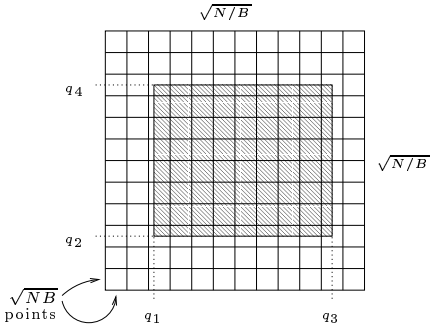

Figure 1.14 Basic squares. To answer a query (q_1, q_2, q_3, q_4) we check points in two vertical and two horizontal slabs, and report points in basic squares completely covered by the query.

with the y-value q_2. This way we obtain the $O(\log_B N)$ segments in the structure containing q_2, and thus (a pointer to) the bottommost point contained in the query for each of the nodes v_{i+1}, $v_{i+2}, \ldots, v_{j-1}$. We then traverse the $j - i - 1 = O(\log_B N)$ relevant sorted lists and output the remaining points using $O(\log_B N + T/B)$ I/Os.

To insert or delete a point, we need to perform $O(1)$ updates on each of the $O(\log(N/B)/\log\log_B N)$ levels of the base tree. Each of these updates takes $O(\log_B N)$ I/Os. We also need to update the base tree. Using a weight-balanced B-tree, Arge et al. (1999b) showed how this can be done in $O((\log_B N)(\log \frac{N}{B})/\log\log_B N)$ I/Os.

Linear space structure. The linear space cross-tree structure of Grossi and Italiano (Grossi and Italiano 1999a;b) consists of two levels. The lower level partitions the plane into $\Theta(\sqrt{N/B})$ vertical slabs and $\Theta(\sqrt{N/B})$ horizontal slabs containing $\Theta(\sqrt{NB})$ points each, forming an irregular grid of $\Theta(N/B)$ *basic squares*—refer to Figure 1.14. Each basic square can contain between 0 and $\sqrt{N/B}$ points. The points are grouped and stored according to the vertical slabs—points in vertically adjacent basic squares containing less than B points are grouped together to form groups of $\Theta(B)$ points and stored in blocks together. The points in a basic square containing more than B points are stored in a B-tree. Thus the lower level uses $O(N/B)$ space. The upper level consists of a linear space search structure which can be used to determine the basic square containing a given point—for now we can think of the structure as consisting of a fan-out $\sqrt{B}$ B-tree $\mathcal{T}_V$ on the $\sqrt{N/B}$ vertical slabs and a separate fan-out $\sqrt{B}$ B-tree $\mathcal{T}_H$ on the $\sqrt{N/B}$ horizontal slabs.

In order to answer a query (q_1, q_2, q_3, q_4) we use the upper level search tree to find the vertical slabs containing q_1 and q_3 and the horizontal slabs containing q_2 and q_4 using $O(\log_B N)$ I/Os. We then explicitly check all points in these slabs and report all the relevant points. In doing so we use $O(\sqrt{NB}/B) = O(\sqrt{N/B})$ I/Os to traverse the vertical slabs and $O(\sqrt{NB}/B + \sqrt{N/B}) = O(\sqrt{N/B})$ I/Os to traverse the horizontal slabs (the $\sqrt{N/B}$-term in the latter bound is a result of the slabs being blocked vertically—a horizontal slab contains $\sqrt{N/B}$ basic squares). Finally, we report all points corresponding to basic squares fully covered by the query. To do so we use $O(\sqrt{N/B} + T/B)$ I/Os since the slabs are blocked vertically. In total we answer a query in $O(\sqrt{N/B} + T/B)$ I/Os.

In order to perform an update we need to find and update the relevant basic square. We may also need to split slabs (insertion) or merge slabs

with neighbor slabs (deletions). In order to do so efficiently while still being able to answer a range query I/O-efficiently, the upper level is actually implemented using a *cross-tree* $\mathcal{T}_{HV}$. $\mathcal{T}_{HV}$ can be viewed as a cross product of $\mathcal{T}_V$ and $\mathcal{T}_H$: For each pair of nodes $u \in \mathcal{T}_H$ and $v \in \mathcal{T}_V$ on the same level we have a node (u, v) in $\mathcal{T}_{HV}$, and for each pair of edges $(u, u') \in \mathcal{T}_H$ and $(v, v') \in \mathcal{T}_V$ we have an edge $((u, v), (u', v'))$ in $\mathcal{T}_{HV}$. Thus the tree has fan-out $O(B)$ and uses $O((\sqrt{N/B})^2) = O(N/B)$ space. Grossi and Italiano (Grossi and Italiano 1999a;b) showed how we can use the cross-tree to search for a basic square in $O(\log_B N)$ I/Os and how the full structure can be used to answer a range query in $O(\sqrt{N/B} + T/B)$ I/Os. They also showed that if $\mathcal{T}_H$ and $\mathcal{T}_V$ are implemented using weight-balanced B-trees, the structure can be maintained in $O(\log_B N)$ I/Os during an update.

7. SPECIAL AND HIGHER-DIMENSIONAL RANGE SEARCHING

As we have seen in the preceding sections, two-dimensional external range searching is theoretically relatively well understood. In contrast, little theoretical work has been done on higher-dimensional range searching and on special cases of higher-dimensional range searching. In this section we survey such results.

Range searching. Vengroff and Vitter (1996) presented a data structure for 3-dimensional range searching with a logarithmic query bound. With recent modifications (Vitter 1999a) their structure answers queries in $O(\log_B N + T/B)$ I/Os and uses $O(\frac{N}{B} \log^3 \frac{N}{B} / \log \log_B^3 N)$ space. More generally, they presented structures for answering $(3 + k)$-sided queries (k of the dimensions, $0 \leq k \leq 3$, have finite ranges) in $O(\log_B N + T/B)$ I/Os using $O(\frac{N}{B} \log^k \frac{N}{B} / \log \log_B^k N)$ space.

As mentioned, space use is often as crucial as query time when manipulating massive datasets. The linear space cross-tree of Grossi and Italiano (Grossi and Italiano 1999a;b), as well as the O-tree of Kanth and Singh (1999), can be extended to support d-dimensional range queries in $O((N/B)^{1-1/d} + T/B)$ I/Os. Updates can be performed in $O(\log_B N)$ I/Os. Divide and merge operations can also be performed on the cross-tree in $O((N/B)^{1-1/d} + T/B)$ I/Os and the structure can be used in the design of dynamic data structures for several other problems (Grossi and Italiano 1999a;b).

Halfspace range searching. Given a set of points in d-dimensional space, a halfspace range query asks for all points on one side of a

query hyperplane. Halfspace range searching is the simplest form of non-isothetic (non-orthogonal) range searching. The problem was first considered in external memory by Franciosa and Talamo (1994; 1997). Based on an internal memory structure due to Chazelle et al. (1985), Agarwal et al. (2000b) described a simple optimal $O(\log_B N + T/B)$ query, $O(N/B)$ space structure for the 2-dimensional case. The number of I/Os used to construct the structure is $O(N(\log N)\log_B N)$. They also described a structure for the 3-dimensional case, answering queries in $O(\log_B N + T/B)$ expected I/Os but requiring $O((N/B)\log(N/B))$ space. This structure is based on an internal memory result of Chan (2000) and the expected number of I/Os needed to construct the structure is $O((N/B)(\log(N/B))\log_B N)$.

Based on the internal memory *partition trees* of Matoušek (1992), Agarwal et al. (2000b) also gave a linear space data structure for answering d-dimensional halfspace range queries in $O((N/B)^{1-1/d+\epsilon} + T/B)$ I/Os for any constant $\epsilon > 0$. The structure can be constructed in $O(N \log N)$ I/Os and using partial rebuilding it can support updates in $O((\log(N/B))\log_B N)$ expected I/Os amortized. Using an improved $O(N \log_B N)$ construction algorithm, Agarwal et al. (2000a) obtained an $O(\log_B^2 N)$ amortized and expected update I/O-bound for the planar case. Agarwal et al. (2000b) also showed how the query bound of the structure can be improved at the expense of extra space. They also discussed how their linear space structure can be used to answer very general queries—more precisely, how all points within a query polyhedron with m faces can be found in $O(m(N/B)^{1-1/d+\epsilon} + T/B)$ I/Os.

Range searching on moving points. Recently there has been an increasing interest in external memory data structures storing continuously moving objects. A key goal is to develop structures that only need to be changed when the velocity or direction of an object changes (as opposed to continuously).

Kollios et al. (1999b) presented initial work on storing moving points in the plane such that all points inside a query range at query time t can be reported in a provably efficient number of I/Os. Their results were improved and extended by Agarwal et al. (2000a) who developed a linear space structure that answers a query in $O((N/B)^{1/2+\epsilon} + T/B)$ I/Os for any constant $\epsilon > 0$. A point can be updated using $O(\log_B^2 N)$ I/Os. The structure is based on partition trees and can also be used to answer queries where two time values t_1 and t_2 are given and we want to find all points that lie in the query range at any time between t_1 and t_2. Using the notion of *kinetic data structures* introduced by Basch et al. (1999), as well as a persistent version of the range searching structure by Arge et al.

(1999b) discussed in Section 6., Agarwal et al. (2000a) also developed a number of other structures with improved query performance. One of these structures has the property that queries in the near future are answered faster than queries further away in time. Further structures with this property were developed by Agarwal et al. (2001c).

8. PROXIMITY QUERIES

Proximity queries such as nearest neighbor and closest pair queries have become increasingly important in recent years, for example because of their applications in similarity search and data mining.

Callahan et al. (1995) developed the first worst-case efficient external proximity query data structures. Their structures are based on an external version of the *topology trees* of Frederickson (1993) called *topology B-trees*, which can be used to dynamically maintain arbitrary binary trees I/O-efficiently. Using topology B-trees and ideas from an internal structure of Bespamyatnikh (1998), Callahan et al. (1995) designed a linear space data structure for dynamically maintaining the *closest pair* of a set of points in d-dimensional space. The structure supports updates in $O(\log_B N)$ I/Os. The same result was obtained by Govindarajan et al. (2000) using the *well-separated pair decomposition* of Callahan and Kosaraju (Callahan and Kosaraju 1995a;b). Govindarajan et al. (2000) also show how to dynamically maintain a well-separated pair decomposition of a set of d-dimensional points using $O(\log_B N)$ I/Os per update.

Using topology B-trees and ideas from an internal structure due to Arya et al. (1994), Callahan et al. (1995) developed a linear space data structure for the dynamic *approximate nearest neighbor* problem. Given a set of points in d-dimensional space, a query point p, and a parameter ϵ, the approximate nearest neighbor problem consists of finding a point q with distance at most $(1+\epsilon)$ times the distance of the actual nearest neighbor of p. The structure answers queries and supports updates in $O(\log_B N)$ I/Os. Agarwal et al. (2000a) designed I/O-efficient data structures for answering approximate nearest neighbor queries on a set of moving points.

In some applications we are interested in finding not only the nearest but all the k nearest neighbors of a query point. Based on their 3-dimensional halfspace range searching structure, Agarwal et al. (2000b) described a structure that uses $O((N/B)\log(N/B))$ space to store N points in the plane such that a k nearest neighbors query can be answered in $(\log_B N + k/B)$ I/Os.

9. PRACTICAL GENERAL-PURPOSE STRUCTURES

Although several of the worst-case efficient (and often optimal) data structures discussed in the previous sections are simple enough to be of practical interest, they are often not the obvious choices when deciding which data structures to use in a real-world application. There are several reasons for this, one of the most important being that in real applications involving massive datasets it is practically feasible to use data structures of size cN/B only for a very small constant c. Since fundamental lower bounds often prevent logarithmic worst-case search cost for even relatively simple problems when restricting the space use to linear, we need to develop heuristic structures which perform well in most practical cases. Space restrictions also motivate us not to use structures for single specialized queries but instead design general structures that can be used to answer several different types of queries. Finally, implementation considerations often motivate us to sacrifice worst-case efficiency for simplicity. All of these considerations have led to the development of a large number of general-purpose data structures that often work well in practice, but which do not come with worst-case performance guarantees. Below we quickly survey the major classes of such structures. The reader is referred to more complete surveys for details (Agarwal and Erickson 1999, Gaede and Günther 1998, Nievergelt and Widmayer 1997, Greene 1989, Orenstein 1990, Samet 1990b).

Range searching in d-dimensions is the most extensively researched problem. A large number of structures have been developed for this problem, including space filling curves (see e.g. Orenstein 1986, Abel and Mark 1990, Asano et al. 1997), grid-files (Nievergelt et al. 1984, Hinrichs 1985), various quad-trees (Samet 1990a;b), kd-B tress (Robinson 1981)—and variants like Buddy-trees (Seeger and Kriegel 1990), hB-trees (Lomet and Salzberg 1990, Evangelidis et al. 1997) and cell-trees (Günther 1989)—and various R-trees (Guttman 1984, Greene 1989, Sellis et al. 1987, Beckmann et al. 1990, Kamel and Faloutsos 1994). Often these structures are broadly classified into two types, namely *space driven* structures (like quad-trees and grid-files), which partition the embedded space containing the data points and *data driven* structures (like kd-B trees and R-trees), which partition the data points themselves. Agarwal et al. (2001b) described a general framework for efficient construction and updating of many of the above structures.

As mentioned above, we often want to be able to answer a very diverse set of queries, like halfspace range queries, general polygon range queries, and point location queries, on a single data structure. Many of the above

data structures can easily be used to answer many such different queries and that is one main reason for their practical success. Recently, there has also been a lot of work on extensions—or even new structures—which also support e.g. moving objects (see e.g. Wolfson et al. 1998; 1999, Salzberg and Tsotras 1999, Kollios et al. 1999a, Šaltenis et al. 2000, Pfoser et al. 2000, Tayeb et al. 1998, and references therein) or proximity queries (see e.g. Berchtold et al. 1997; 1998b; 1996, Ciacca et al. 1997, Korn et al. 1996, Papadopoulos and Manolopoulos 1997, Roussopoulos et al. 1995, Seidl and Kriegel 1997, Sproull 1991, White and Jain 1996, Katayama and Satoh 1997, Hjaltason and Samet 1995, Gaede and Günther 1998, Agarwal and Erickson 1999, Nievergelt and Widmayer 1997, and references therein). However, as discussed, most often no guarantee on the worst-case query performance is provided for these structures.

So far we have mostly discussed point data structures. In general, we are interested in storing objects such as lines and polyhedra with a spatial extent. Like in the point case, a large number of heuristic structures, many of which are variations of the ones mentioned above, have been proposed for such objects. However, almost no worst-case efficient structures are known. In practice a *filtering/refinement* method is often used when managing objects with spatial extent. Instead of directly storing the objects in the data structure we store the *minimal bounding (axis-parallel) rectangle* containing each object together with a pointer to the object itself. When answering a query we first find all the minimal bounding rectangles fulfilling the query (the *filtering* step) and then we retrieve the objects corresponding to these rectangles and check each of them to see if they fulfill the query (the *refinement* step). One way of designing data structures for rectangles (or even more general objects) is to transform them into points in higher-dimensional space and store these points in one of the point data structures discussed above (see e.g. Gaede and Günther 1998, Nievergelt and Widmayer 1997, for a survey). However, a structure based on another idea has emerged as especially efficient for storing and querying minimal bounding rectangles. Below we further discuss this so-called R-tree and its many variants.

R-trees. The R-tree, originally proposed by Guttman (1984), is a multiway tree very similar to a B-tree; all leaf nodes are on the same level of the tree and a leaf contains $\Theta(B)$ data rectangles. Each internal node v (except maybe for the root) has $\Theta(B)$ children. For each of its children v_i, v contains the minimal bounding rectangle of all the rectangles in the tree rooted in v_i. An R-tree has height $O(\log_B N)$ and uses $O(N/B)$ space. An example of an R-tree is shown in Figure 1.15. Note that there

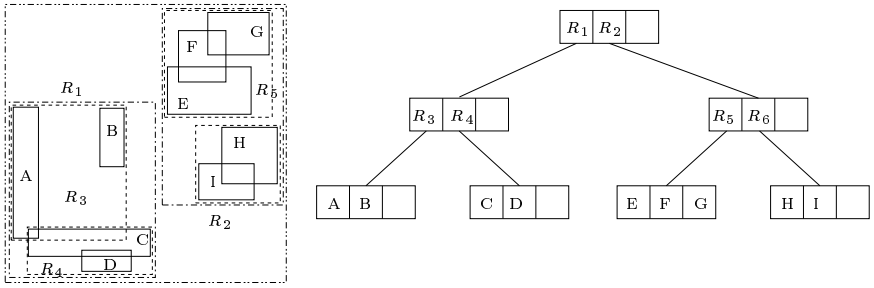

Figure 1.15 R-tree constructed on rectangles A, B, C, ..., I ($B = 3$).

is no unique R-tree for a given set of data rectangles and that minimal bounding rectangles stored within an R-tree node can overlap.

In order to query an R-tree to find, say, all rectangles containing a query point p, we start at the root and recursively visit all children whose minimal bounding rectangle contains p. This way we visit all internal nodes whose minimal bounding rectangle contains p. There can be many more such nodes than actual data rectangles containing p and intuitively we want the minimal bounding rectangles stored in an internal node to overlap as little as possible in order to obtain a query efficient structure.

An insertion can be performed in $O(\log_B N)$ I/Os like in a B-tree. We first traverse the path from the root to the leaf we choose to insert the new rectangle into. The insertion might result in the need for node splittings on the same root-leaf path. As insertion of a new rectangle can increase the overlap in a node, several heuristics for choosing which leaf to insert a new rectangle into, as well as for splitting nodes during rebalancing, have been proposed (Greene 1989, Sellis et al. 1987, Beckmann et al. 1990, Kamel and Faloutsos 1994). The R*-tree variant of Beckmann et al. (1990) seems to result in the best performance in many cases. Deletions are also performed similarly to deletions in a B-tree but we cannot guarantee an $O(\log_B N)$ bound since finding the data rectangle to delete may require many more I/Os. Rebalancing after a deletion can be performed by merging nodes like in a B-tree but some R-tree variants instead delete a node when it underflows and reinsert its children into the tree (often referred to as "forced reinsertion"). The idea is to try to obtain a better structure by forcing a global reorganization of the structure instead of the local reorganization a node merge constitutes.

Constructing an R-tree using repeated insertion takes $O(N \log_B N)$ I/Os and does not necessarily result in a good tree in terms of query performance. Therefore several sorting based $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/O construc-

tion algorithms have been proposed (Roussopoulos and Leifker 1985, Kamel and Faloutsos 1993, DeWitt et al. 1994, Leutenegger et al. 1996, Berchtold et al. 1998a). These algorithms are more than a factor of B faster than the repeated insertion algorithm and several of them produce an R-tree with practically better query performance than an R-tree built by repeated insertion. Still, no better than a linear worst-case query I/O-bound has been proven for any of them. Very recently, however, de Berg et al. (2000) and Agarwal et al. (2001d) presented R-tree construction algorithms resulting in R-trees with provably efficient worst-case query performance measured in terms of certain parameters describing the input data. They also discussed how these structures can be efficiently maintained dynamically.

10. BUFFER TREES

In internal memory we can sort N elements in optimal $O(N \log N)$ time using $\Theta(N)$ operations on a dynamic balanced search tree. Using the same algorithm and a B-tree in external memory results in an algorithm using $O(N \log_B N)$ I/Os. This is a factor of $\frac{B \log_B N}{\log_{M/B}(N/B)}$ away from optimal. In order to obtain an optimal sorting algorithm we need a structure that supports updates in $O(\frac{1}{B} \log_{M/B} \frac{N}{B})$ I/Os. The inefficiency of the B-tree sorting algorithm is a consequence of the B-tree being designed to be used in an "on-line" setting where queries should be answered immediately—updates and queries are handled on an individual basis. This way we are not able to take full advantage of the large internal memory. It turns out that in an "off-line" environment where we are only interested in the overall I/O use of a series of operations and where we are willing to relax the demands on the query operations, we can develop data structures on which a series of N operations can be performed in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os in total. To do so we use the *buffer tree* technique developed by Arge (1995a).

Basically the buffer tree is just a fan-out $\Theta(M/B)$ B-tree where each internal node has a buffer of size $\Theta(M)$. The tree has height $O(\log_{M/B} \frac{N}{B})$; refer to Figure 1.16. Operations are performed in a "lazy" manner: In order to perform an insertion we do not (like in a normal B-tree) search all the way down the tree for the relevant leaf. Instead, we wait until we have collected a block of insertions and then we insert this block in the buffer of the root (which is stored on disk). When a buffer "runs full" its elements are "pushed" one level down to buffers on the next level. We can do so in $O(M/B)$ I/Os since the elements in the buffer fit in main memory and the fan-out of the tree is $O(M/B)$. If the buffer of any of the nodes on the next level becomes full by this process, the

buffer-emptying process is applied recursively. Since we push $\Theta(M)$ elements one level down the tree using $O(M/B)$ I/Os (that is, we use $O(1)$ I/Os to push one block one level down), we can argue that every block of elements is touched a constant number of times on each of the levels of the tree. Thus, not counting rebalancing, inserting N elements requires $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os in total, or $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ amortized. Arge (1995a) showed that rebalancing can be handled in the same bound.

The basic buffer tree supporting insertions only can be used in an I/O-efficient sorting algorithm. Arge (1995a) showed how deletions and (one-dimensional) range queries can also be supported I/O-efficiently using buffers. The range queries are *batched* in the sense that we do not obtain the result of a query immediately. Instead parts of the result will be reported at different times as the query is pushed down the tree. This means that the data structure can only be used in algorithms where future updates and queries do not depend on the result of the queries. Luckily this is the case in many plane-sweep algorithms (Edelsbrunner and Overmars 1985, Arge 1995a). In general, problems where the entire sequence of updates and queries is known in advance, and the only requirement on the queries is that they must all eventually be answered, are known as *batched dynamic problems* (Edelsbrunner and Overmars 1985). Using the idea of multislabs discussed in Section 3., Arge (1995a) also showed how to implement a buffered segment tree, and Arge et al. (1998) showed how to use this data structure in a technique for solving a general class of high-dimensional problems.

The buffer tree technique has been used to develop several data structures which in turn have been used to develop algorithms in many different areas (Arge et al. 1995; 1997; 1999a, Kumar and Schwabe 1996, Arge 1995b, Fadel et al. 1999, Buchsbaum et al. 2000, van den Bercken et al. 1997; 1998, Hutchinson et al. 1997, Brengel et al. 1999, Sanders 1999). External buffered *priority queues* have been extensively researched be-

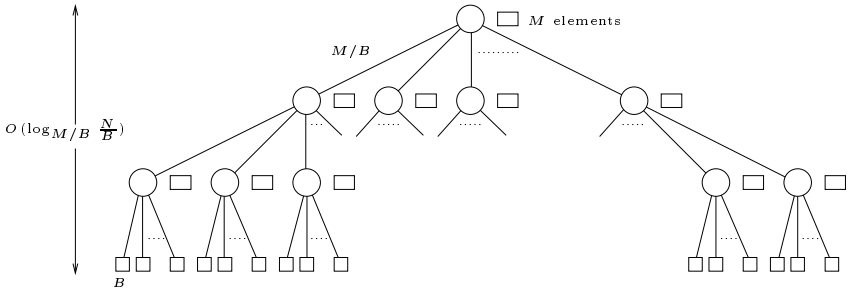

Figure 1.16 Buffer tree; Fan-out M/B tree where each node has a buffer of size M. Operations are performed in a lazy way using the buffers.

cause of their applications in graph algorithms. Arge (1995a) showed how to perform deletemin operations on a basic buffer tree in amortized $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ I/Os. Note that in this case the deletemin occurs right away, that is, it is not batched. This is accomplished by periodically computing the $O(M)$ smallest elements in the structure and storing them in internal memory. Fadel et al. (1999) developed a similar buffered heap. Using a partial rebuilding idea, Brodal and Katajainen (1998) developed a worst-case efficient external priority queue. A sequence of B operations on this structure requires $O(\log_{M/B}\frac{N}{B})$ I/Os. Using the buffer tree technique on a tournament tree, Kumar and Schwabe (1996) developed a priority queue supporting update operations in $O(\frac{1}{B}\log\frac{N}{B})$ I/Os. They also showed how to use their structure in several efficient external graph algorithms (see e.g Abello et al. 1998, Agarwal et al. 1998, Arge et al. 2000b, Buchsbaum et al. 2000, Chiang et al. 1995, Hutchinson et al. 1999, Kumar and Schwabe 1996, Maheshwari and Zeh 1999, Munagala and Ranade 1999, Nodine et al. 1996, Ullman and Yannakakis 1991, Maheshwari and Zeh 2001, Feuerstein and Marchetti-Spaccamela 1993, Arge et al. 2000a; 2001, Meyer 2001, Zeh 2001, for other results on external graph algorithms and data structures). Note that if the priority of an element is known, an update operation can be performed in $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ I/Os on a buffer tree using a delete and an insert operation.

11. CONCLUSIONS

In this chapter we have discussed recent advances in the development of provably efficient external memory dynamic data structures, mainly for geometric objects. Such structures are often crucial in massive dataset applications. We have discussed some of the most important techniques utilized to obtain efficient structures.

Even though a lot of progress has been made, many problems still remain open. For example, $O(\log_B N)$-query and space efficient structures still need to be found for many higher-dimensional problems. The practical performance of many of the worst-case efficient structures also need to be researched.

Acknowledgments

The author thanks the National Science Foundation for partially supporting this work through ESS grant EIA–9870734, RI grant EIA–9972879 and CAREER grant EIA–9984099, and Tammy Bailey, Tavi Procopiuc, Jan Vahrenhold, as well as an anonymous reviewer, for comments on earlier drafts of this chapter.

Bibliography

Abel, D. J. and Mark, D. M. (1990). A comparative analysis of some two-dimensional orderings. *Intl. J. Geographic Informations Systems*, 4(1):21–31.

Abello, J., Buchsbaum, A. L., and Westbrook., J. R. (1998). A functional approach to external graph algorithms. In *Proc. Annual European Symposium on Algorithms, LNCS 1461*, pages 332–343.

Agarwal, P. K., Arge, L., Brodal, G. S., and Vitter, J. S. (1999). I/O-efficient dynamic point location in monotone planar subdivisions. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 1116–1127.

Agarwal, P. K., Arge, L., and Erickson, J. (2000a). Indexing moving points. In *Proc. ACM Symp. Principles of Database Systems*, pages 175–186.

Agarwal, P. K., Arge, L., Erickson, J., Franciosa, P., and Vitter, J. (2000b). Efficient searching with linear constraints. *Journal of Computer and System Sciences*, 61(2):194–216.

Agarwal, P. K., Arge, L., and Govindarajan, S. (2001a). External range counting. Manuscript.

Agarwal, P. K., Arge, L., Murali, T. M., Varadarajan, K., and Vitter, J. S. (1998). I/O-efficient algorithms for contour line extraction and planar graph blocking. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 117–126.

Agarwal, P. K., Arge, L., Procopiuc, O., and Vitter, J. S. (2001b). A framework for index bulk loading and dynamization. In *Proc. Annual International Colloquium on Automata, Languages, and Programming*.

Agarwal, P. K., Arge, L., and Vahrenhold, J. (2001c). A time responsive indexing scheme for moving points. In *Proc. Workshop on Algorithms and Data Structures*.

Agarwal, P. K., de Berg, M., Gudmundsson, J., Hammer, M., and Haverkort, H. J. (2001d). Box-trees and R-trees with near-optimal query time. In *Proc. ACM Symp. on Computational Geometry*, pages 124–133.

Agarwal, P. K. and Erickson, J. (1999). Geometric range searching and its relatives. In Chazelle, B., Goodman, J. E., and Pollack, R., editors, *Advances in Discrete and Computational Geometry*, volume 223 of *Contemporary Mathematics*, pages 1–56. American Mathematical Society, Providence, RI.

Aggarwal, A. and Vitter, J. S. (1988). The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127.

Arge, L. (1995a). The buffer tree: A new technique for optimal I/O-algorithms. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 334–345. A complete version appears as BRICS technical report RS-96-28, University of Aarhus.

Arge, L. (1995b). The I/O-complexity of ordered binary-decision diagram manipulation. In *Proc. Int. Symp. on Algorithms and Computation, LNCS 1004*, pages 82–91. A complete version appears as BRICS technical report RS-96-29, University of Aarhus.

Arge, L., Brodal, G. S., and Toma, L. (2000a). On external memory MST, SSSP and multi-way planar graph separation. In *Proc. Scandinavian Workshop on Algorithms Theory*.

Arge, L., Ferragina, P., Grossi, R., and Vitter, J. (1997). On sorting strings in external memory. In *Proc. ACM Symp. on Theory of Computation*, pages 540–548.

Arge, L., Hinrichs, K. H., Vahrenhold, J., and Vitter, J. S. (1999a). Efficient bulk operations on dynamic R-trees. In *Proc. Workshop on Algorithm Engineering, LNCS 1619*, pages 328–347.

Arge, L., Meyer, U., Toma, L., and Zeh, N. (2001). On external-memory planar depth first search. In *Proc. Workshop on Algorithms and Data Structures*.

Arge, L., Procopiuc, O., Ramaswamy, S., Suel, T., and Vitter, J. S. (1998). Theory and practice of I/O-efficient algorithms for multidimensional batched searching problems. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 685–694.

Arge, L., Samoladas, V., and Vitter, J. S. (1999b). On two-dimensional indexability and optimal range search indexing. In *Proc. ACM Symp. Principles of Database Systems*, pages 346–357.

Arge, L. and Teh, S.-M. (2000). Unpublished results.

Arge, L., Toma, L., and Vitter, J. S. (2000b). I/O-efficient algorithms for problems on grid-based terrains. In *Proc. Workshop on Algorithm Engineering and Experimentation*.

Arge, L. and Vahrenhold, J. (2000). I/O-efficient dynamic planar point location. In *Proc. ACM Symp. on Computational Geometry*, pages 191–200.

Arge, L., Vengroff, D. E., and Vitter, J. S. (1995). External-memory algorithms for processing line segments in geographic information systems. In *Proc. Annual European Symposium on Algorithms, LNCS 979*, pages 295–310. To appear in special issues of *Algorithmica* on Geographical Information Systems.

Arge, L. and Vitter, J. S. (1996). Optimal dynamic interval management in external memory. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 560–569.

Arya, S., Mount, D. M., Netanyahu, N. S., Silverman, R., and Wu, A. (1994). An optimal algorithm for approximate nearest neighbor searching. In *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms*, pages 573–582.

Asano, T., Ranjan, D., Roos, T., Welzl, E., and Widmayer, P. (1997). Space-filling curves and their use in the design of geometric data structures. *Theoret. Comput. Sci.*, 181(1):3–15.

Basch, J., Guibas, L. J., and Hershberger, J. (1999). Data structures for mobile data. *Journal of Algorithms*, 31(1):1–28.

Baumgarten, H., Jung, H., and Mehlhorn, K. (1994). Dynamic point location in general subdivisions. *Journal of Algorithms*, 17:342–380.

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

Becker, B., Gschwind, S., Ohler, T., Seeger, B., and Widmayer, P. (1996). An asymptotically optimal multiversion B-tree. *VLDB Journal*, 5(4):264–275.

Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B. (1990). The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 322–331.

Bender, M. A., Demaine, E. D., and Farach-Colton, M. (2000). Cache-oblivious B-trees. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 339–409.

Bentley, J. L. (1979). Decomposable searching problems. *Information Processing Letters*, 8(5):244–251.

Berchtold, S., Böhm, C., Keim, D. A., and Kriegel, H.-P. (1997). A cost model for nearest neighbor search in high-dimensional data spaces. In *Proc. ACM Symp. Principles of Database Systems*, pages 78–86.

Berchtold, S., Böhm, C., and Kriegel, H.-P. (1998a). Improving the query performance of high-dimensional index structures by bulk load operations. In *Proc. Conference on Extending Database Technology, LNCS 1377*, pages 216–230.

Berchtold, S., Ertl, B., Keim, D. A., Kriegel, H.-P., and Seidl, T. (1998b). Fast nearest neighbor search in high-dimensional spaces. In *Proc. IEEE International Conference on Data Engineering*, pages 209–218.

Berchtold, S., Keim, D. A., and Kriegel, H.-P. (1996). The X-tree: An index structure for high-dimensional data. In *Proc. International Conf. on Very Large Databases*, pages 28–39.

Bespamyatnikh, S. N. (1998). An optimal algorithm for closets pair maintenance. *Discrete and Computational Geometry*, 19:175–195.

Blankenagel, G. and Güting, R. H. (1990). XP-trees—External priority search trees. Technical report, FernUniversität Hagen, Informatik-Bericht Nr. 92.

Brengel, K., Crauser, A., Ferragina, P., and Meyer, U. (1999). An experimental study of priority queues in external memory. In *Proc. Workshop on Algorithm Engineering, LNCS 1668*, pages 345–358.

Brodal, G. S. and Katajainen, J. (1998). Worst-case efficient external-memory priority queues. In *Proc. Scandinavian Workshop on Algorithms Theory, LNCS 1432*, pages 107–118.

Buchsbaum, A. L., Goldwasser, M., Venkatasubramanian, S., and Westbrook, J. R. (2000). On external memory graph traversal. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 859–860.

Callahan, P., Goodrich, M. T., and Ramaiyer, K. (1995). Topology B-trees and their applications. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 381–392.

Callahan, P. B. and Kosaraju, S. R. (1995a). Algorithms for dynamic closest-pair and n-body potential fields. In *Proc. 6th ACM-SIAM Sympos. Discrete Algorithms*, pages 263–272.

Callahan, P. B. and Kosaraju, S. R. (1995b). A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *Journal of the ACM*, 42(1):67–90.

Chan, T. M. (2000). Random sampling, halfspace range reporting, and construction of ($\leq k$)-levels in three dimensions. *SIAM Journal of Computing*, 30(2):561–575.

Chazelle, B. (1986). Filtering search: a new approach to query-answering. *SIAM J. Comput.*, 15(3):703–724.

Chazelle, B. (1990). Lower bounds for orthogonal range searching: I. the reporting case. *Journal of the ACM*, 37(2):200–212.

Chazelle, B. and Guibas, L. J. (1986). Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162.

Chazelle, B., Guibas, L. J., and Lee, D. T. (1985). The power of geometric duality. *BIT*, 25(1):76–90.

Cheng, S. W. and Janardan, R. (1992). New results on dynamic planar point location. *SIAM J. Comput.*, 21(5):972–999.

Chiang, Y.-J., Goodrich, M. T., Grove, E. F., Tamassia, R., Vengroff, D. E., and Vitter, J. S. (1995). External-memory graph algorithms. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 139–149.

Chiang, Y.-J. and Silva, C. T. (1997). I/O optimal isosurface extraction. In *Proc. IEEE Visualization*, pages 293–300.

Chiang, Y.-J. and Silva, C. T. (1999). External memory techniques for isosurface extraction in scientific visualization. In Abello, J. and Vitter, J. S., editors, *External memory algorithms and visualization*, pages 247–277. American Mathematical Society, DIMACS series in Discrete Mathematics and Theoretical Computer Science.

Chiang, Y.-J., Silva, C. T., and Schroeder, W. J. (1998). Interactive out-of-core isosurface extraction. In *Proc. IEEE Visualization*, pages 167–174.

Ciacca, P., Patella, M., and Zezula, P. (1997). M-tree: An efficient access method for similarity search in metric spaces. In *Proc. International Conf. on Very Large Databases*, pages 426–435.

Comer, D. (1979). The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137.

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. (1990). *Introduction to Algorithms*. The MIT Press, Cambridge, Mass.

Crauser, A. and Ferragina, P. (1999). On constructing suffix arrays in external memory. In *Proc. Annual European Symposium on Algorithms, LNCS, 1643*, pages 224–235.

Crauser, A., Ferragina, P., Mehlhorn, K., Meyer, U., and Ramos, E. (1998). Randomized external-memory algorithms for some geometric problems. In *Proc. ACM Symp. on Computational Geometry*, pages 259–268.

de Berg, M., Gudmundsson, J., Hammar, M., and Overmars, M. (2000). On R-trees with low stabbing number. In *Proc. Annual European Symposium on Algorithms*, pages 167–178.

DeWitt, D. J., Kabra, N., Luo, J., Patel, J. M., and Yu, J.-B. (1994). Client-server paradise. In *Proceedings of VLDB Conference*, pages 558–569.

Driscoll, J. R., Sarnak, N., Sleator, D. D., and Tarjan, R. (1989). Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124.

Edelsbrunner, H. (1983a). A new approach to rectangle intersections, part I. *Int. J. Computer Mathematics*, 13:209–219.

Edelsbrunner, H. (1983b). A new approach to rectangle intersections, part II. *Int. J. Computer Mathematics*, 13:221–229.

Edelsbrunner, H. and Overmars, M. (1985). Batched dynamic solutions to decomposable searching problems. *Journal of Algorithms*, 6:515–542.

Evangelidis, G., Lomet, D., and Salzberg, B. (1997). The hb^{π}-tree: A multi-attribute index supporting concurrency, recovery and node consolidation. *The VLDB Journal*, 6(1):1–25.

Fadel, R., Jakobsen, K. V., Katajainen, J., and Teuhola, J. (1999). Heaps and heapsort on secondary storage. *Theoretical Computer Science*, 220(2):345–362.

Farach, M., Ferragina, P., and Muthukrishnan, S. (1998). Overcoming the memory bottleneck in suffix tree construction. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 174–183.

Ferragina, P. and Grossi, R. (1995). A fully-dynamic data structure for external substring search. In *Proc. ACM Symp. on Theory of Computation*, pages 693–702.

Ferragina, P. and Grossi, R. (1996). Fast string searching in secondary storage: Theoretical developments and experimental results. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 373–382.

Ferragina, P. and Luccio, F. (1998). Dynamic dictionary matching in external memory. *Information and Computation*, 146(2):85–99.

Feuerstein, E. and Marchetti-Spaccamela, A. (1993). Memory paging for connectivity and path problems in graphs. In *Proc. Int. Symp. on Algorithms and Computation, LNCS 762*, pages 416–425.

Franciosa, P. and Talamo, M. (1997). Time optimal halfplane search on external memory. Unpublished manuscript.

Franciosa, P. G. and Talamo, M. (1994). Orders, k-sets and fast halfplane search on paged memory. In *Proc. Workshop on Orders, Algorithms and Applications (ORDAL'94), LNCS 831*, pages 117–127.

Frederickson, G. N. (1993). A structure for dynamically maintaining rooted trees. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 175–184.

Frigo, M., Leiserson, C. E., Prokop, H., and Ramachandran, S. (1999). Cache-oblivious algorithms. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 285–298.

Gaede, V. and Günther, O. (1998). Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231.

Goodrich, M. T., Tsay, J.-J., Vengroff, D. E., and Vitter, J. S. (1993). External-memory computational geometry. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 714–723.

Govindarajan, S., Lukovszki, T., Maheshwari, A., and Zeh, N. (2000). I/O-efficient well-separated pair decomposition and its applications. In *Proc. Annual European Symposium on Algorithms*, pages 220–231.

Greene, D. (1989). An implementation and performance analysis of spatial data access methods. In *Proc. IEEE International Conference on Data Engineering*, pages 606–615.

Grossi, R. and Italiano, G. F. (1999a). Efficient cross-tree for external memory. In Abello, J. and Vitter, J. S., editors, *External Memory Algorithms and Visualization*, pages 87–106. American Mathematical Society, DIMACS series in Discrete Mathematics and Theoretical Computer Science. Revised version available at ftp://ftp.di.unipi.it/pub/techreports/TR-00-16.ps.Z.

Grossi, R. and Italiano, G. F. (1999b). Efficient splitting and merging algorithms for order decomposable problems. *Information and Computation*, 154(1):1–33.

Günther, O. (1989). The design of the cell tree: An object-oriented index structure for geometric databases. In *Proc. IEEE International Conference on Data Engineering*, pages 598–605.

Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 47–57.

Hellerstein, J. M., Koutsoupias, E., and Papadimitriou, C. H. (1997). On the analysis of indexing schemes. In *Proc. ACM Symp. Principles of Database Systems*, pages 249–256.

Hinrichs, K. H. (1985). *The grid file system: Implementation and case studies of applications.* PhD thesis, Dept. Information Science, ETH, Zürich.

Hjaltason, G. R. and Samet, H. (1995). Ranking in spatial databases. In *Proc. of Advances in Spatial Databases, LNCS 951*, pages 83–95.

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

Hutchinson, D., Maheshwari, A., Sack, J.-R., and Velicescu, R. (1997). Early experiences in implementing the buffer tree. In *Proc. Workshop on Algorithm Engineering*, pages 92–103.

Hutchinson, D., Maheshwari, A., and Zeh, N. (1999). An external-memory data structure for shortest path queries. In *Proc. Annual Combinatorics and Computing Conference, LNCS 1627*, pages 51–60.

Icking, C., Klein, R., and Ottmann, T. (1987). Priority search trees in secondary memory. In *Proc. Graph-Theoretic Concepts in Computer Science, LNCS 314*, pages 84–93.

Kamel, I. and Faloutsos, C. (1993). On packing R-trees. In *Proc. International Conference on Information and Knowledge Management*, pages 490–499.

Kamel, I. and Faloutsos, C. (1994). Hilbert R-tree: An improved R-tree using fractals. In *Proc. International Conf. on Very Large Databases*, pages 500–509.

Kanellakis, P. C., Ramaswamy, S., Vengroff, D. E., and Vitter, J. S. (1996). Indexing for data models with constraints and classes. *Journal of Computer and System Sciences*, 52(3):589–612.

Kanth, K. V. R. and Singh, A. K. (1999). Optimal dynamic range searching in non-replicating index structures. In *Proc. International Conference on Database Theory, LNCS 1540*, pages 257–276.

Katayama, N. and Satoh, S. (1997). The SR-tree: An index structure for high-dimensional nearest-neighbor queries. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 369–380.

Knuth, D. E. (1998). *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading MA, second edition.

Kollios, G., Gunopulos, D., and Tsotras, V. J. (1999a). Nearest neighbor queries in a mobile environment. In *Proc. International Workshop on Spatio-Temporal Database Management, LNCS 1678*, pages 119–134.

Kollios, G., Gunopulos, D., and Tsotras, V. J. (1999b). On indexing mobile objects. In *Proc. ACM Symp. Principles of Database Systems*, pages 261–272.

Korn, F., Sidiropoulos, N., Faloutsos, C., Siegel, E., and Protopapas, Z. (1996). Fast nearest neighbor search in medical image databases. In *Proc. International Conf. on Very Large Databases*, pages 215–226.

Koutsoupias, E. and Taylor, D. S. (1998). Tight bounds for 2-dimensional indexing schemes. In *Proc. ACM Symp. Principles of Database Systems*, pages 52–58.

Kumar, V. and Schwabe, E. (1996). Improved algorithms and data structures for solving graph problems in external memory. In *Proc. IEEE Symp. on Parallel and Distributed Processing*, pages 169–177.

Leutenegger, S. T., López, M. A., and Edgington, J. (1996). STR: A simple and efficient algorithm for R-tree packing. In *Proc. IEEE International Conference on Data Engineering*, pages 497–506.

Lomet, D. and Salzberg, B. (1990). The hB-tree: A multiattribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems*, 15(4):625–658.

Maheshwari, A. and Zeh, N. (1999). External memory algorithms for outerplanar graphs. In *Proc. Int. Symp. on Algorithms and Computation, LNCS 1741*, pages 307–316.

Maheshwari, A. and Zeh, N. (2001). I/O-efficient algorithms for bounded treewidth graphs. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 89–90.

Matoušek, J. (1992). Efficient partition trees. *Discrete Comput. Geom.*, 8:315–334.

McCreight, E. (1985). Priority search trees. *SIAM Journal of Computing*, 14(2):257–276.

Mehlhorn, K. (1984). *Data Structures and Algorithms 1: Sorting and Searching*. Springer-Verlag, EATCS Monographs on Theoretical Computer Science.

Mehlhorn, K. and Näher, S. (1990). Dynamic fractional cascading. *Algorithmica*, 5:215–241.

Meyer, U. (2001). External memory bfs on undirected graphs with bounded degree. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 87–88.

Morrison, D. R. (1968). PATRICIA: Practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15:514–534.

Munagala, K. and Ranade, A. (1999). I/O-complexity of graph algorithm. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 687–694.

Nievergelt, J., Hinterberger, H., and Sevcik, K. (1984). The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71.

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

Nievergelt, J. and Widmayer, P. (1997). Spatial data structures: Concepts and design choices. In van Kreveld, M., Nievergelt, J., Roos, T., and Widmayer, P., editors, *Algorithmic Foundations of GIS*, pages 153–197. Springer-Verlag, LNCS 1340.

Nodine, M. H., Goodrich, M. T., and Vitter, J. S. (1996). Blocking for external graph searching. *Algorithmica*, 16(2):181–214.

Orenstein, J. (1986). Spatial query processing in an object-oriented database system. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 326–336.

Orenstein, J. (1990). A comparison of spatial query processing techniques for native and parameter spaces. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 343–352.

Overmars, M. H. (1983). *The Design of Dynamic Data Structures.* Springer-Verlag, LNCS 156.

Overmars, M. H. (1985). Range searching in a set of line segments. In *Proc. 1st Annu. ACM Sympos. Comput. Geom.*, pages 177–185.

Papadopoulos, A. and Manolopoulos, Y. (1997). Performance of nearest neighbor queries in R-trees. In *Intl. Conference on Database Theory, LNCS 1186*, pages 394–408.

Pfoser, D., Jensen, C. S., and Theodoridis, Y. (2000). Novel approaches to the indexing of moving objects trajectories. In *Proc. International Conf. on Very Large Databases*, pages 395–406.

Ramaswamy, S. (1997). Efficient indexing for constraint and temporal databases. In *Proc. International Conference on Database Theory, LNCS 1186*, pages 419–431.

Ramaswamy, S. and Subramanian, S. (1994). Path caching: A technique for optimal external searching. In *Proc. ACM Symp. Principles of Database Systems*, pages 25–35.

Robinson, J. (1981). The K-D-B tree: A search structure for large multidimensional dynamic indexes. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 10–18.

Roussopoulos, N., Kelley, S., and Vincent, F. (1995). Nearest neighbor queries. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 71–79.

Roussopoulos, N. and Leifker, D. (1985). Direct spatial search on pictorial databases using packed R-trees. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 17–31.

Ruemmler, C. and Wilkes, J. (1994). An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28.

Salzberg, B. and Tsotras, V. J. (1999). A comparison of access methods for time evolving data. *ACM Computing Surveys*, 31(2):158–221.

Samet, H. (1990a). *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison Wesley, MA.

Samet, H. (1990b). *The Design and Analyses of Spatial Data Structures.* Addison Wesley, MA.

Samoladas, V. and Miranker, D. (1998). A lower bound theorem for indexing schemes and its application to multidimensional range queries. In *Proc. ACM Symp. Principles of Database Systems*, pages 44–51.

Sanders, P. (1999). Fast priority queues for cached memory. In *Proc. Workshop on Algorithm Engineering and Experimentation, LNCS 1619*, pages 312–327.

Sarnak, N. and Tarjan, R. E. (1986). Planar point location using persistent search trees. *Communications of the ACM*, 29:669–679.

Seeger, B. and Kriegel, H.-P. (1990). The buddy-tree: An efficient and robust access method for spatial data base systems. In *Proc. International Conf. on Very Large Databases*, pages 590–601.

Seidl, T. and Kriegel, H.-P. (1997). Efficient user-adaptable similarity search in large multimedia databases. In *Proc. International Conf. on Very Large Databases*, pages 506–515.

Sellis, T., Roussopoulos, N., and Faloutsos, C. (1987). The R^+-tree: A dynamic index for multi-dimensional objects. In *Proc. International Conf. on Very Large Databases*, pages 507–518.

Snoeyink, J. (1997). Point location. In Goodman, J. E. and O'Rourke, J., editors, *Handbook of Discrete and Computational Geometry*, chapter 30, pages 559–574. CRC Press LLC, Boca Raton, FL.

Sproull, R. F. (1991). Refinements to nearest neighbor searching in k-dimensional trees. *Algorithmica*, 6(4):579–589.

Subramanian, S. and Ramaswamy, S. (1995). The P-range tree: A new data structure for range searching in secondary memory. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 378–387.

Tamassia, R. and Vitter, J. S. (1996). Optimal cooperative search in fractional cascaded data structures. *Algorithmica*, 15(2):154–171.

Tayeb, J., Ulusoy, O., and Wolfson, O. (1998). A quadtree-based dynamic attribute indexing method. *The Computer Journal*, 41(3):185–200.

Ullman, J. D. and Yannakakis, M. (1991). The input/output complexity of transitive closure. *Annals of Mathematics and Artificial Intellegence*, 3:331–360.

Vahrenhold, J. and Hinrichs, K. H. (2000). Planar point-location for large data sets: To seek or not to seek. In *Proc. Workshop on Algorithm Engineering*.

van den Bercken, J., Seeger, B., and Widmayer, P. (1997). A generic approach to bulk loading multidimensional index structures. In *Proc. International Conf. on Very Large Databases*, pages 406–415.

van den Bercken, J., Seeger, B., and Widmayer, P. (1998). A generic approach to processing non-equijoins. Technical Report 14, Philipps-Universität Marburg, Fachbereich Matematik und Informatik.

van Kreveld, M. J. and Overmars, M. H. (1991). Divided k-d trees. *Algorithmica*, 6:840–858.

Varman, P. J. and Verma, R. M. (1997). An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):391–409.

Vengroff, D. E. and Vitter, J. S. (1996). Efficient 3-D range searching in external memory. In *Proc. ACM Symp. on Theory of Computation*, pages 192–201.

Vitter, J. S. (1999a). External memory algorithms and data structures. In Abello, J. and Vitter, J. S., editors, *External Memory Algorithms and Visualization*, pages 1–38. American Mathematical Society, DIMACS series in Discrete Mathematics and Theoretical Computer Science.

Vitter, J. S. (1999b). Online data structures in external memory. In *Proc. Annual International Colloquium on Automata, Languages, and Programming, LNCS 1644*, pages 119–133.

Vitter, J. S. and Shriver, E. A. M. (1994). Algorithms for parallel memory, I: Two-level memories. *Algorithmica*, 12(2–3):110–147.

Šaltenis, S., Jensen, C. S., Leutenegger, S. T., and López, M. A. (2000). Indexing the positions of continuously moving objects. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 331–342.

White, D. A. and Jain, R. (1996). Similarity indexing with the SS-tree. In *Proc. IEEE International Conference on Data Engineering*, pages 516–523.

Wolfson, O., Sistla, A. P., Chamberlain, S., and Yesha, Y. (1999). Updating and querying databases that track mobile units. *Distributed and Parallel Databases*, 7(3):257–287.

Wolfson, O., Xu, B., Chamberlain, S., and Jiang, L. (1998). Moving objects databases: Issues and solutions. In *Intl. Conf. on Scientific and Statistical Database Management*, pages 111–122.

Zeh, N. (2001). I/o-efficient planar separators and applications. Manuscript.

Zhang, D., Markowetz, A., Tsotras, V., Gunopulos, D., and Seeger, B. (2001). Efficient computation of temporal aggregates with range predicates. In *Proc. ACM Symp. Principles of Database Systems*, pages 237–245.