BRICS RS-94-16 L. Arge: External-Storage Data Structures for Plane-Sweep Algorithms

MATEMATISK INSTITUT
AARHUS UNIVERSITET
TRYKKERIET

Basic Research in Computer Science

External-Storage Data Structures for Plane-Sweep Algorithms

Lars Arge

BRICS Report Series RS-94-16

ISSN 0909-0878 June 1994

See back inner page for a list of recent publications in the BRICS Report Series. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@daimi.aau.dk

External-Storage Data Structures for Plane-Sweep Algorithms*

Lars Arge
(large@daimi.aau.dk)

BRICS†
Computer Science Department
Aarhus University
Ny Munkegade
DK-8000 Aarhus C, Denmark.

Abstract

In this paper we develop a technique for transforming an internal memory data structure into an external storage data structure suitable for plane-sweep algorithms. We use this technique to develop external storage versions of the range tree and the segment tree. We also obtain an external priority queue. Using the first two structures, we solve the orthogonal segment intersection, the isothetic rectangle intersection, and the batched range searching problem in the optimal number of I/O-operations. Unlike previously known I/O-algorithms the developed algorithms are straightforward generalizations of the ordinary internal memory plane-sweep algorithms. Previously almost no dynamic data structures were known for the model we are working in.

*This work was partially supported by the ESPRIT II Basic Research Actions Program of the EC under contract No. 7141 (project ALCOM II) and by Aarhus University Research Foundation

†Basic Research in Computer Science, Centre of the Danish National Research Foundation.

1 Introduction

In the last few years, more and more attention has been given to Input/Output (I/O) complexity of existing algorithms and to the development of I/O-efficient algorithms. This is due to the fact that communication between fast internal memory and slower external storage is the bottleneck in many large-scale computations. The significance of this bottleneck is increasing as internal computation gets faster, and especially as parallel computing gains popularity [11].

There is a lot of literature on algorithms and data structures in an I/O-environment. Especially external sorting has been studied extensively (see e.g. [8]). However, most of the work has been done using an I/O-model where the block size (or page size) equals the size of the internal memory. This model is motivated by the observation, that the time used to move a memory-load of elements from external storage to internal memory, is the sum of the seek time (the time used to move the read-write head) and the transport time - of which the seek time is the dominant. This means that it is a good heuristic to read/write as much as possible every time one accesses the external storage. However, in modern multi-user systems one can not be sure to have control of the read-write head during the whole transport of a memory-load. As an example, a typical block size in a UNIX-environment is 8 K bytes. This means that accessed data not present in internal memory is transported from external storage to internal memory in blocks of size 8 K byte, and one can not be sure that the read-write head does not move between the transport of two consecutive blocks. This motivates a model where one distinguishes between the block size and the size of the internal memory.

1.1 I/O-Model

We will be working in an I/O-model introduced by Aggarwal and Vitter [1]. The I/O-model has the following parameters:

N = number of elements in the problem.

B = number of elements per block.

M = number of elements that fit into internal memory.

An I/O-operation in the model is a swap of B elements from internal memory with B consecutive elements from external storage. The measure of performance we will consider is the number of such I/O's needed to solve a given problem. Internal computation is free. In [16] the model is extended with a parameter D. Here the secondary storage is partitioned into D distinct disk drives, and if no two blocks come from the same disk, D blocks can be transferred per I/O. Furthermore the model can be extended such that we have more than one internal processor (see e.g. [9]). Recently a number of authors have considered further extended models, with so-called multilevel hierarchical memories (see e.g. [10] or [15]), which aim to capture the fact that large-scale computer systems contain many levels of memory - ranging from very small but fast registers, to successively larger but slower memories.

In [1] the I/O-complexity of a number of specific sorting-related problems is considered, namely sorting, fast Fourier transformation, permuting and matrix transposition. This work is extended in [9], [10] and [16] where sorting is considered in the extended versions of the model. In [3] a general connection between the comparison-complexity and the I/O-complexity of a given problem is shown.

Very recently a number of large-scale computational geometry problems have been considered in the model [13]. As pointed out in [13] large-scale problems involving geometric data are ubiquitous in many areas, e.g in spatial databases, geographical information systems, constraint logic programming, object oriented databases, statistics, virtual reality systems, and computer graphics. Another area where large-scale computational geometry problems arise and where the problems considered in this paper are of special interest, are in VLSI design, and especially in VLSI design rule checking [4]. In [13] a technique called distributed sweeping for transforming a plane-sweep algorithm into an I/O-algorithm is considered, and using this technique optimal algorithms for the following problems (with plane-sweep solutions) are designed: Orthogonal segment intersection reporting, all nearest neighbors, batched range queries, visibility from a point in the plane, pairwise rectangle intersection, measure of a union of rectangles, and the 3-d maxima problem (a sketch of the algorithms for the first two are given i [13]).

1.2 Our Results

In this paper we develop a "technique" for transforming an internal memory data structure into an external storage data structure suitable for solving ***batched dynamic searching problems*** [5]. The batched dynamic version of a searching problem is the following: given a sequence of insertions, deletions, and queries on a set of elements, report all answers to the queries when the sequence of operations is performed (in the given order) on an initially empty set. We are only interested in the overall runtime (I/O-usage). Many plane-sweep algorithms are examples of batched dynamic searching problems - namely the plane-sweep algorithms where the operations as well as the order of the operations are given at the start of the sweep. In particular this means that the operations are not depending on the answers to the queries performed during the sweep.

The main ideas in our "technique" are the following: when we want to transform a structure we group the (binary) nodes in an internal version of the structure into nodes with fan-out equaling the number of blocks that fits into internal memory. We then assign a "buffer" of size a memory-load to each of these nodes. When an element is inserted into such a tree, it is not inserted at a leaf, but in the top buffer. When a buffer of a node "runs full", the elements in the buffer are "pushed" one level down to buffers on the next level. We call this a buffer-emptying process. In this way the insertions are done in a "lazy" way, and as deletions are done in the same way, we obtain a tree where we at the same time can have an insertion and a deletion of the same element in the tree. When we want to perform a query on a transformed structure, we basically do it in the same way as when we do insertions and deletions. The query then gets batched in the sense that the result of the query may be generated (and reported) in a lazy fashion by several buffer-emptying processes, and this is the main reason why our structures are suitable for solving batched dynamic searching problems. The "laziness" also means that all I/O-bounds gets amortized.

Some work has been done on designing I/O-variants of known "internal" dynamic data structures (see e.g. [2, 7, 8, 13, 14]), but practically no work has been done in a more realistic (and thus complicated) I/O-model where the block size does not equal the size of the internal memory. Using our "technique" we develop three external storage data structures. We de-

velop an external version of a (one-dimensional) ***range tree*** structure with the operations ***insert***, ***delete***, ***batched rangesearch*** and ***write/empty*** (an operation that empties all buffers), an external version of the ***segment tree*** with the operations ***insert/delete***, ***batched search*** and ***write/empty***, and an external ***priority-queue*** with operations ***insert***, ***delete*** and ***deletemin***.

The developed structures enables us to solve the orthogonal segment intersection, the batched range searching, and the isothetic rectangle intersection problems in the optimal number of I/O-operations. Maybe more important though, we solve the problems with exactly the same plane-sweep algorithms as are used in internal memory. Until now it has been very characteristic that for a given problem the developed I/O-algorithm is quite different from an internal memory algorithm for the same problem. The three problems considered here are good examples of this, as the algorithms developed in [13] are very different from the usual internal memory plane-sweep solution. Using our "technique" we on the other hand manage to isolate all the I/O-specific parts of the algorithms in the data structures.

Using the external range tree structure we also obtain an "on-line" sorting algorithm. When we want to sort N elements, we simply insert them, one by one, in the developed tree structure, and write them in sorted order when we have inserted all of them. By the I/O-bounds we prove in this paper, it then follows that the sorting algorithm is optimal. This algorithm is the first "on-line" sorting algorithm for the model - previously known algorithms all need all the elements to be present by the start of the algorithm.

As mentioned most of the known optimal I/O-algorithms are very different from the internal memory algorithms. Furthermore almost all of them rely heavily on a subdivision technique from [1] that in a linear number of I/O's divides N input objects into $\frac{M}{B}$ almost equally sized sets. This technique in turn builds heavily on a complicated median-finding algorithm. We believe this means that most of the previously developed algorithms are of more theoretical than practical interest. On the other hand, we think that algorithms based on the data structures developed in this paper will be of practical interest. We are currently making experiments in order to verify this. We also believe that the work in this paper could be a step towards dynamic algorithms for the I/O-model. Finally - although the proof is not included in this paper - the results on the segment tree

structure also holds in the model where we have D parallel disks. This means that the sorting algorithm presented in this paper also represents an alternative to the algorithm presented in [9].

The main organization of the rest of this paper is the following: In the next section we define and discuss an I/O-version of big-O. In section 3, we shortly define the orthogonal segment intersection, the batched range searching problem, and the isothetic rectangle intersection problem. In section 4, we then solve the first problem by using our "technique" to develop the needed data structure. We also show how to develop an external priority-queue. In section 5, we then develop an external version of the segment tree in order to solve the two other problems. Finally, conclusions and open problems are given in section 6.

2 I/O-version of Big-O

In order to state the results in this paper we need the following "I/O-version" of big-O defined in [3]:

Definition 1

$$\overline{O}(f(N,M,B)) = \{g \mid \exists c \in \mathbb{R}^+ : \forall M,B \in \mathbb{N} : \exists N_0 \in \mathbb{N} : \forall N \geq N_0 : g(N,M,B) \leq c \cdot f(N,M,B)\}$$

This definition captures the idea that the leading term with respect to N is, in fact, the leading term with respect to all three variables, but on the other hand, even though M and B are less important than N, they can not be regarded as ordinary constants. For example we have $\frac{N}{B}+M\sqrt{N} \in \overline{O}(\frac{N}{B})$ whereas $N \notin \overline{O}(\frac{N}{B})$. $\overline{\Omega}$ and $\overline{\Theta}$ can bee defined similarly. As we shall see shortly the quotients $\frac{N}{B}$ (the number of blocks in the problem) and $\frac{M}{B}$ (the number of blocks that fits into internal memory) play an important role in the study of I/O-complexity. Therefore, we will use n as shorthand for $\frac{N}{B}$ and m for $\frac{M}{B}$. Furthermore we will say that an algorithm uses a linear number of I/O-operations if it uses at most $\overline{O}(n)$ I/O's to solve a problem of size N.

Before we proceed, we will make a note on the size of the characteristic parameters. First off all, as mentioned, a typical block size is 8 K bytes. If the size of an element is e.g. 80 bytes (think of a database with name,

address, tax-information and so on), this means that a typical size of B could be 100. On a SUN3 workstation, the typical size of the internal memory is 8 M byte so the size of M would typically be 100000. This again means that the value of m would be 1000. Note that m is a parameter that characterizes the machine (it is independent of the problem/the element size), while B and M depends on the particular problem in hand - or rather the size of the elements in the problem.

In the I/O-model $\overline{\Omega}(n \log_m n)$ is a lower bound on the number of I/O's needed to sort [1, 3]. It should be noted, that for realistic values of M and B this bound is always less than N. That is, if we solve the equation $n \log_m n \geq N$ with respect to N and insert the typical values for M and B, we get that N should be greater than approximately 2^{1000}. This again means that one should be extra careful with the big-O notation. For example we should be careful with the bound $\overline{O}(\frac{\log_m n}{B})$, which will be used many times in this paper. It is easy to see from definition 1 that $\frac{\log_m n}{B}+1 \in \overline{O}(\frac{\log_m n}{B})$, but for all values of n one could ever think of the 1 term will dominate. We will return to this "problem" in subsection 4.1.1 and 5.4, and here just note that even though we have modified the big-O notation so that B and M are not regarded as ordinary constants, we still have the problems that arise when working with sub-linear bounds.

3 Planar Intersection Reporting Problems

In this section we define the planar intersection problems we are going to solve, and present their "normal" internal memory solutions. For both problems a $\overline{\Omega}(n \log_m n + r)$ I/O lower bound (r is the number of *blocks* reported) follows from the $\Omega(N \log_2 N+R)$ comparison model lower bound, and the general connection between comparison and I/O-lower bounds proven in [3]. This means that the bound $\overline{O}(\frac{\log_m n}{B})$ corresponds to the $O(\log_2 N)$ bound in an internal memory data structure and therefore, it is this bound we would like to obtain on (most of) the operations on our external storage data structures.

The problem of ***orthogonal segment intersection reporting*** is defined as follows: We are given N orthogonal line segments and want to report all intersections of orthogonal lines. The optimal plane-sweep algorithm (see e.g. [12]) makes a vertical sweep with a horizontal line, inserting the

x-coordinate of vertical segments in a search tree when their top end-point are reached, and deleting them again when their bottom end-point are reached. When the sweep-line reaches a horizontal segment, a range-search operation with the two end-points of the segment is made in the tree and intersections are reported. In an internal memory model this algorithm will run in the optimal time $O(N \log_2 N + R)$.

The problem of ***isothetic rectangle intersection*** is defined similar to the orthogonal segment intersection problem. Given N isothetic rectangles in the plane we want to report all intersecting pairs. In [4] it is shown that if we - besides the orthogonal segment intersection problem - can solve the batched range searching problem in time $O(N \log_2 N + R)$, we will in total get a solution to the rectangle intersection problem with the same (optimal) time-bound.

The ***batched range search problem*** - given N points and N rectangles report for each rectangle all the points that lie inside it - can be solved with a plane-sweep algorithm in almost the same way as the orthogonal segment intersection problem - we just have to use a segment tree instead of a search tree. In a segment tree [4, 12], we can dynamically store line segments whose endpoints belong to a fixed set and perform inverse range queries, that is, given a point we can report all segments in the tree that contain the point. The optimal plane sweep algorithm makes a vertical sweep with a horizontal line, inserting a segment (rectangle) in the segment tree when the top segment of a rectangle is reached, and deleting it when the bottom segment is reached. When a point is reached we perform an inverse range query with it. As we can perform insertions and deletions of segments and inverse range queries in the segment tree in $O(\log_2 N)$ and $O(\log_2 N + R)$ time respectively, we, in total, have the desired $O(N \log_2 N + R)$ time algorithm.

4 The Buffer Tree

In this section we use our "technique" to develop an external version of the (one-dimensional) range tree. This will enable us to solve the orthogonal segment intersection problem with the "normal" internal memory algorithm. We call our structure a buffer tree. The buffer tree is just an (a, b)-tree (with $a = \frac{1}{2}m$ and $b = m$), extended with a buffer of m blocks

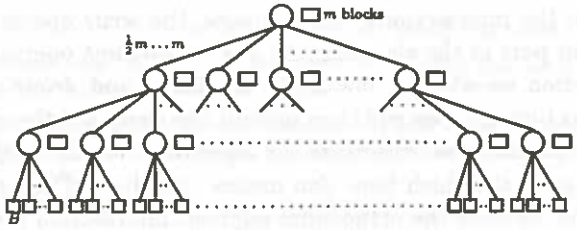


Figure 1: The buffer tree

in each node as discussed in the introduction (see figure 1). Note that a similar idea is mentioned in [15], but from [13] and [15] it appears that the authors have not managed to get optimal bounds (the logarithms are based 2 instead of m). More precisely we have:

Definition 2 *T is a buffer tree if*

- *All leaves, each consisting of B elements, have the same depth.*
- *Every internal node has a buffer of m blocks.*
- *A buffer contains at most one non-full block.*
- *The root has between 2 and m sons.*
- *All other nodes have between $\frac{1}{2}m$ and m sons.*
- *Between insertions/deletions, i.e. when the buffer tree is "balanced", every buffer contains at most $\frac{1}{2}m$ blocks.*

It is obvious that we have the following:

Lemma 1 *The height of a buffer tree with n blocks in the leaves is $\overline{O}(\log_m n)$.*

4.1 Operations on the Buffer tree

As discussed in subsection 3 we need a data structure with the three operations ***insert***, ***delete*** and ***rangesearch*** in order to solve the orthogonal segment intersection problem. We will need one more operation, namely a ***write*** or ***empty*** operation that empty all buffers of a buffer tree. We

need to run this operation by the end of the sweep-line algorithm in order to report all the intersections. Furthermore, the ***write*** operation will be an important part of the algorithm for the ***rangesearch*** operation. In the next subsection we will see how to do the ***insert*** and ***delete*** operations, and in subsection 4.1.2 we will then present the ***write*** and the ***rangesearch*** operations. Actually, we will make the algorithms for these operations a little more general (which here also means complicated) than we really need in order to solve the orthogonal segment intersection problem - we will for example balance the buffer tree dynamically, which we do not really need in the orthogonal segment intersection algorithm. We do this in order to be able to use the structure to solve other problems. We will return to this in subsection 4.2, where we use the buffer tree to solve the "on-line" sorting problem.

4.1.1 The *Insert* and *Delete* Operations.

As discussed in the introduction we do the following when we want to insert an element into or delete an element from the buffer tree; we construct a new element consisting of the element to be inserted or deleted, a time stamp, and an indication of whether the element is to be inserted or deleted. When we have collected B such elements in internal memory, we insert the block in the buffer of the root. If the buffer of the root still contains less than $\frac{1}{2}m$ blocks we stop. Otherwise, we "empty" the buffer.

If the current node is an *internal node*:

We do two of the following processes, the first on the first $\frac{1}{2}m$ blocks in the buffer and the second on the rest. This prevents "cascade exhaustion" of the buffers further down in the tree, if all elements keep "going in the same direction".

1. The elements in the buffer are loaded into internal memory.
2. The elements are sorted. If two equal elements - an insertion and a deletion - "meet" during this process, and if the time stamps "fit", then the two elements annihilates.
3. The partitioning (or routing) elements of the node are loaded into internal memory.
4. The elements are partitioned according to the partitioning elements and output to the appropriate buffers (maintaining the invariant that at most one block in a buffer is non-full).
5. If the buffer of any node contains more than $\frac{1}{2}m$ blocks the emptying-process is recursively applied on these nodes.

Figure 2: The buffer-emptying process on internal nodes.

If the current node is, a *leaf node*:

1. The elements in the buffer are loaded into internal memory and sorted.
2. The elements from the buffer are merged with the elements from the leaves while matching insert/delete-elements are removed.
3. If the resulting number of blocks is between $\frac{1}{2}m$ and m the elements are then placed in sorted order in the leaves, and the corresponding partitioning element is put in the node.
4. If the resulting number of blocks is greater than m (and less than $2m$) we do the following:
   (a) Find the median of the elements.
   (b) Partition the elements according to the median.
   (c) Replace the node with two new nodes - one for each partition - each with between $\frac{1}{2}m$ and m sons/leaves.
   (d) Insert the median among the partition elements of the father of the new nodes. If the father v has less than m sons we are finished. Otherwise v needs to be split (see figure 4). Since splitting may propagate we formulate it as a loop:

```
DO v has m+1 sons ->
  IF v is the root ->
    let x be a new node and make v its only son
  ELSE
    let x be the father of v
  FI
  Let v' be a new node
  Let v' be a new son of x immediately after v
  Split v:
    Take the rightmost (½m+1) sons away
     from v and make them sons of v'.
    Distribute the elements from the buffer of v between
     the buffer of v and v'.
  Let v=x
OD
```

Figure 3: The buffer-emptying process on leaf nodes.

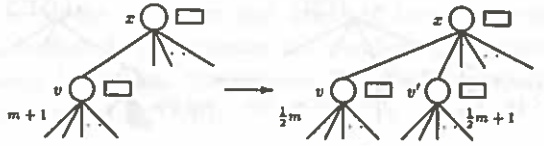


Figure 4: Splitting a node

5. If the resulting number of blocks is less than $\frac{1}{2}m$ we do the following:

   (a) Place the elements in sorted order in the leaves and the corresponding partitioning elements in the node (v). Since the number of leaves is less than $\frac{1}{2}m$ we need to rebalance. We do this by *fusing* (or *sharing*) nodes (see figure 6 and 7), and since fusion may propagate we formulate it as a loop. Let x be the father and v' a brother of v, and let $\gamma(w)$ denote the number of sons of w:

```
DO v has less than ½m sons AND
   v' has less than m − γ(v) ->
  Fuse v and v':
    Make all sons of v' sons of v and put
    the elements from the buffer of v' into the
    buffer of v (if v's buffer gets to big we
    should do a buffer-empty process on v later).
  Let v=x
  Let v' be a brother of x
  IF x does not have a brother (x is the root) AND
     x has only one son ->
    Delete x
    STOP
  FI
  Let x be the father of v.
OD
(either γ(v) ≥ ½m and we are finished or
 we can finish by sharing)
IF v has less than ½m sons ->
  Share:
    Take ½m − γ(v) sons away from v' and make
     them sons of v.
    Distribute the elements from the buffer of v' between
     the buffer of v' and v (again maybe we should do a
     buffer-empty process on v later).
FI
```

Figure 5: The buffer-emptying process on leaf nodes continued.

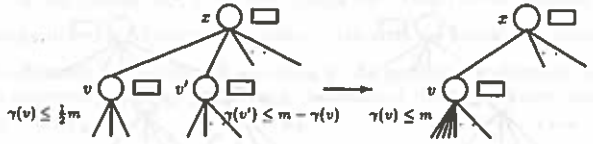


Figure 6: Fusing nodes

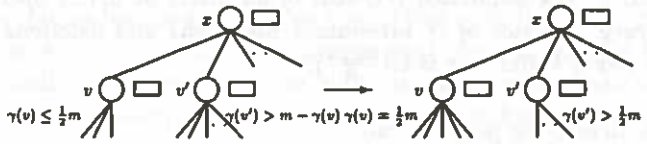


Figure 7: Sharing

The ***buffer-emptying process*** is described in figure 2, 3 and 5. The most important part of the process, namely the one used on internal nodes, is described in figure 2. The process used on leaf nodes (nodes whose sons are leaves) - which also contains the tree-rebalancing algorithms - is then presented in figure 3 and 5. Note that the rebalance algorithms are almost precisely the same as those described in [6].

It is easy to see from the algorithm that the cost of emptying a buffer, not counting recursive costs, is $\overline{O}(m)$ I/O's. Note that here it is crucial that both the buffer size and the fan-out of the nodes is $\overline{O}(m)$. It is equally easy to see that the cost of a rebalancing operation - splitting, fusing or sharing - is $\overline{O}(m)$ I/O's. Note that the constant in both of these bounds is small (less than five).

We now have the following:

Lemma 2 ***The total number of rebalancing operations in an arbitrary sequence of N intermixed insertions and deletions into an initially empty buffer tree is $\overline{O}(n)$.***

Proof In [6] it is proven that if $b \geq 2a$ then an arbitrary sequence of N intermixed insertions and deletions in an initially empty (a,b)-tree causes $O(N)$ rebalance operations - provided that we uses the balance-algorithms described above. In the buffer tree we have blocks in the leaves, so it is blocks we insert and delete in the underlying (a,b)-tree. As the use of buffers do not increase the number of rebalance operations, we immediately have that the number of rebalance operations in the buffer tree is bounded by $\overline{O}(n)$.

□

We then have the following:

Theorem 1 *The amortized I/O-cost of an insert or delete operation in an arbitrary sequence of N intermixed insertions and deletions into an initially empty buffer tree is $\overline{O}(\frac{\log_m n}{B})$.*

Proof We divide the proof in two.

First, we look at the buffer emptying cost. To this part of the proof we associate a number of credits to each block of elements in the tree. More precisely each block in the buffer of node x must hold $\overline{O}($ the height of the tree in which x is root) credits. We then have to give each block $\overline{O}(\log_m n)$ credits on insertion in the buffer of the root - that is, each operation "pays" $\overline{O}(\frac{\log_m n}{B})$ I/O's. As emptying a buffer costs $\overline{O}(m)$ I/O's and as we only do a buffer-empty process when there are at least $\frac{1}{2}m$ blocks in the buffer, this means that the blocks in the buffer can pay for the emptying-process as they all gets pushed one level down.

Next, we look at the balancing cost. A rebalancing operation costs $\overline{O}(m)$ I/O's and from lemma 2 we know that the number of such operations is bounded by $\overline{O}(n)$. This means that the total rebalancing cost of N insertions/deletions is $\overline{O}(n \cdot m)$.

This in total proves that the amortized cost of one insert or delete operation is $\overline{O}(\frac{\log_m n}{B}) + \overline{O}(\frac{m}{B}) \in \overline{O}(\frac{\log_m n}{B})$

□

It should be noted that in the rebalance-cost in the proof of Theorem 1, the $\frac{m}{B}$-term can be reduced to $\frac{1}{B}$. This can be achieved by reducing the number of rebalance-operations to $\overline{O}(\frac{n}{m})$ by reducing a to $\frac{1}{4}m$, modifying the balance-algorithm a little, and use a result due to Huddleston and Mehlhorn [6]. In [6] it is proven that using a particular rebalancing algorithm, the number of rebalance-operations in an (a,b)-tree is linear in the number of operations divided by the hysteresis ($b+1-2a$) of the tree. As discussed in section 2 it is important for the practical significance of the result that we replace the $\frac{m}{B}$-term with $\frac{1}{B}$.

Actually, we could also use a completely different balancing strategy - we could balance the tree in a top-down style instead of the bottom-up style we have described. We can make such a strategy work, if we "tune" our constants (fan-out and buffer-size) in such a way that the maximal number of elements in the buffers is guaranteed to be less that half the number of elements in the leaves of the tree. If this is the case, we can do

the rebalancing on a node when we empty it's buffer. More precisely we can do a split, a fuse or a sharing in connection with the buffer-emptying process on a node, in order to guarantee that there is room in the node to allow half of it's sons to fuse or split. In this way we make sure that rebalancing will never propagate. Unfortunately we have not been able to make this simpler strategy work when *rangesearch* operations are allowed.

4.1.2 The *Rangesearch* Operation

Normally, we perform a *rangesearch*(x_1, x_2) on a segment tree by searching down the tree for the two elements x_1 and x_2, and then report all the elements in the leaves between x_1 and x_2. The reports can however also be generated while we search down the tree, if we report all the elements in the proper (sub-) trees on our way down. This is the strategy we use in the buffer tree. When we perform a *rangesearch*, we start almost as when we do an insertion or deletion. We make a new element containing the interval $[x_1, x_2]$ and a time stamp, and insert it in the tree. We then have to modify our buffer-emptying process in order to deal with the new rangesearch-elements. When we meet a rangesearch-element in a buffer-empty process, we first determine whether the search-interval is completely contained in one of the intervals corresponding to the subtrees rooted in the sons of the node in question. If this is the case we insert the element in the corresponding buffer. Otherwise we "split" the element in two - one for x_1 and one for x_2 - and write the (sub-) trees that are completely contained in the interval. The splitting only occurs once and hereafter the rangesearch-elements are pushed downwards in the buffer-emptying processes like the insert-elements (there are no matching "delete-rangesearch-elements" however), while trees completely contained in the interval are written. As discussed in the introduction this means that our *rangesearch* operation gets batched, but this is all right as it it not important in the orthogonal segment intersection algorithm (in batched dynamic searching problems) that the result of a *rangesearch* appears right away. We should be careful here however, as problems could arise when we are rebalancing the tree - especially when two nodes, of which one of them contains a (already "split") rangesearch-element, are merged. There will be no problems though, if we always empty the buffers of nodes involved in rebalancing. This adds a $\overline{O}(m)$ term to the cost of a rebalancing-operation and we can accept that, as one of these

already uses $\overline{O}(m)$ I/O's.

The really hard part of the rangesearch-algorithm is the algorithm that writes the elements in a buffer tree. In order to get the optimal I/O-bound we would like the algorithm to use $\overline{O}(n_a)$ I/O's - where n_a is the actual number of blocks in the tree, that is, the number of blocks used by elements that are not deleted by delete-elements in the buffers of the tree. But just to get the I/O-cost down to $\overline{O}(n)$, where n is the number of blocks in the leaves of the tree, seems difficult. On the other hand, if n_d is the number of blocks deleted by delete-elements in the tree, we have that $n = n_a + n_d$. This means that if we can empty all the buffers in the tree - which includes removing all the delete elements - in $\overline{O}(n)$ I/O's, we can charge the n_d part to the delete-elements, adding $\overline{O}(\frac{1}{B})$ to the number of I/O's used by a *delete* operation, and get the desired bound. So, what we are going to do is actually to present an algorithm that gives us the *empty* operation which we discussed in section 4.1. This is not as easy as it may seem, as one should remember that the tree also contain other rangesearch-elements that may have "matching" elements among the elements we are going to delete in the writing-process. We should in other words also take care of a lot of element-rangesearch-element "hits" during the write algorithm.

Our writing algorithm runs in two major phases, and we will present it in the next two lemmas. First, we show how to do the emptying under the special assumption that all delete-elements in the tree has a "matching" insert-element in the tree and then, we develop an algorithm that from a general tree produces a tree which fulfills this assumption. Note that in the sweep-line algorithm for the orthogonal segment intersection problem this assumption is automatically fulfilled.

Lemma 3 *Given a (sub-) buffer tree T with n leaves - where only elements known to be in the tree have been deleted, and where all elements are distinct - the elements represented by T, and "matching" rangesearch-elements and elements in T, can be written in $\overline{O}(n+r)$ I/O's amortized. Here $r \cdot B$ is the number of reported elements. After this process all buffers in T are empty.*

Proof The algorithm that proves the lemma relies on two important facts:

1. All elements in the buffers of a given level of the tree are always in correct time-order compared to all *relevant* elements on higher levels. By relevant we mean that an element from the buffer of a node v was inserted in the tree before all elements in buffers of the nodes on the path from v to the root of the tree. This means that we can assume that all elements on one level were inserted before all elements on higher levels.

2. The special assumption gives us the following: If d, i and s are matching delete, insert and rangesearch-elements (that is "$i = d$" and "i is contained in s"), and if we know that they are in the following time-order d, s, i, then we can report that i is in s and "remove" i and d. If we know that the time-order is s, i (knowing that no element - especially not d - is in between s and i), we can report that i is in s and exchange their time-order. Similarly, if we know that the time-order is d, s we can again report that d is in s (because we know that there is an i matching d "on the other side of s") and exchange their time-order.

The algorithm now runs in four steps. First we process each level independently - making three lists for each level in the tree consisting of insertions, deletions and searches, respectively. Secondly, we "push" the delete-elements downwards from the top level of the tree to the leaves, and then we "push" search and insertions elements downwards. Finally, we process the leaves that is, we "merge" the elements from the buffers with the elements in the leaves. At the same time we can complete our task - that is, write the elements in the tree.

The algorithm is indicated in figure 8. As we know the order between elements from different buffers on the same level, **step one** of the algorithm will give us three sorted lists for each level in the tree. These lists respectively contain (undeleted) insertion, (unused) deletion, and rangesearch-elements from the level. During step one we also "push" the delete-elements on each level down in the time-order relative to the rangesearch-elements, and the rangesearch-elements down relative to the insertion-elements. By doing this we change the time-order between elements on the same level, such that the time order of the lists on level j is i_j, s_j, d_j. The situation after step one is pictured in figure 9 a).

Step one uses $\overline{O}(n+r)$ I/O-operations - if $r \cdot B$ is the number of elements

When we in the following write that we report "hits", we actually accumulate elements to be reported in internal memory and write/report them as soon as we have a whole block.

1. We make three lists for each level of the tree (called i_j, d_j and s_j on level j) in the following way:

   For a given level we do the following for all buffers, starting from the left:

   - We load the blocks from the buffer and use fact 2 to report all "element-search-hits" in the buffer. We also remove all "insert-delete matches" between elements from the buffer.
   - We then output the insert, delete, and rangesearch-elements in three different lists in sorted order.

2. We push the delete-elements downwards by doing the following for all levels j (from the top):

   - "merge" d_j and i_{j+1} - removing delete/insert matches.
   - "merge" d_j and s_{j+1} - reporting "hits" - in the following way:

     During the merging we keep a third list of "active" rangesearch-elements from s_{j+1}, which we - except for the B most recently added elements - keep in secondary storage.

     When we during the merge find that a rangesearch-elements has the smallest x (that is x_1) value, we insert it in this list.

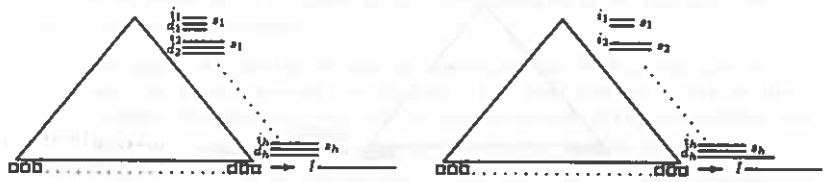
     When we find that a delete-element from d_j has the smallest x-value, we then run through the list, and report that this element is in the interval of all rangesearch-elements that have not yet "expired" - that is, whose x_2 value is less than the value of the d_j element. At the same time we remove all the rangesearch-elements that have expired from the list.
   - merge d_j and d_{j+1} into d_{j+1}.

3. We push the search and the insert-elements downwards like we did with the delete-elements in the previous step (- ending up with three lists i, d and s).

4. We process the leaves and write the result in the following way:

   - "merge" the elements in the leaves and d - removing matching elements and putting the result in a new list of elements (that is, the leaves are not touched).
   - "merge" s and the produced list - reporting "hits" like previous.
   - Write the list and i, and throw s and the list away.
   - Merge i and d.
   - We do the following for all leaf nodes, starting from the left:
     - While more than $\frac{1}{2}m$ blocks from the list go to the node, we put $\frac{1}{2}m$ blocks in the buffer and do a buffer-emptying process on it.
     - We put the remaining (less than $\frac{1}{2}m$) blocks in the buffer of the node, and continue with the next leaf node.
   - Finally we do a buffer-emptying process on all the leaf nodes.

Figure 8: The writing algorithm.



a) b)

Figure 9: a) The situation after step one - the time-order of the lists is the one indicated. b) The situation after step two - again the time-order is indicated.

reported in the step. The r contribution should be obvious, like it should be obvious that the number of I/O-operations used to make the lists is proportional to the number of blocks in the buffers of the tree. The n contribution then follows from the fact that the total number of block in the buffers is bounded by a constant times the number of leaves (there is less than $2\frac{n}{m/2}$ nodes each containing at most m blocks).

In **step two** we "push" the delete-elements downwards. For each level we first exchange the time order of d_j and i_{j+1} by "merging" them. The reason why we can do this without missing any "element-search-hits" follows from the previously mentioned time-order of the lists. The second step then corresponds to interchanging d_j and s_{j+1} (fact 2). Finally, we can merge the two "delete-lists" together. The situation after processing the last level is pictured in figure 9 b).

That the first (and third) step in step two overall uses $\overline{O}(n)$ I/O's follows from the following argument: Every level in the tree contains more nodes than all the levels above it put together. This means that the number of I/O's used to "merge" d_j and i_{j+1} is bounded by a constant times the maximal number of blocks on level j. The bound then follows from the fact that the total number of blocks in the tree is bounded by a constant times the number of leaves. That the second step overall use a linear number of I/O's, plus the number used to report "hits", follows from the same argument and a simple amortization argument.

Step three is similar to step two, and the I/O-bound again follows by an argument similar to the one used previously. In total we end up with three lists which we call i, s and d (see figure 10).

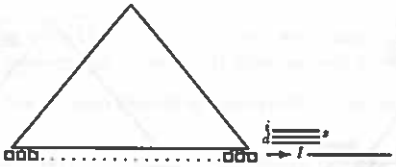


Figure 10: The situation after step 3 - again the time-order is the one indicated.

In **Step four** we report "hits" between elements in s and I - if they are not deleted by elements in d - and we then write the result we are actually after. Finally, we insert or delete the appropriate elements by doing a number of buffer-emptying process on the leaf nodes. Again it should be easy to verify that the algorithm works.

It is equally easy to verify that the merging and the writing steps (the first four steps) in the algorithm uses a linear number of I/O's in the number of leaves, plus the number of I/O's used to report hits. The buffer-emptying processes performed in step 5 (and the "merging") also use $\overline{O}(n)$ I/O's. This follows from the fact that the list consisting of elements from i and d contains $\overline{O}(n)$ block and that we need one I/O (credits) for each of these blocks in order to pay for the performed buffer-emptying processes. Finally, the bound follows from the fact that we in the last step maximally performs as many buffer-emptying processes as there are nodes on the last level - this number is $\overline{O}(\frac{n}{m})$.

Note that after step four all buffers are empty, and that we are not counting I/O's used for rebalancing as they are already accounted for.

□

We now show how to fulfill the assumptions of lemma 3:

Lemma 4 *Given a buffer tree T with n leaves, the delete-elements in the buffers of T that do not match an insert-element in T can be removed in $\overline{O}(n)$ I/O's.*

Proof The algorithm that proves this lemma runs in two major steps. In the first step, we make a list of all the delete-elements in the tree that do not match an insertion-elements, and in the second step we remove the

Note that we in step one of the algorithm copy the elements we are working with, and leave the original elements untouched.

1. For each level of the tree we make two sorted lists - one with the insertion- and one with the deletion-elements from the level. At the same time, we remove the delete-elements from these lists that match an insertion-element in the corresponding lists.

   We then "merge" the lists by doing the following for all levels j (from the top):

   - "Merge" the list of delete-elements from level j with the insert-elements from level $j+1$ - that is, remove the delete-elements that has a matching insert-element.
   - Merge the remaining delete-elements from level j with the delete-elements from level $j+1$.

2. We "merge" the list of delete-elements resulting from step one with the lists of delete-elements from the the individual levels - *from the bottom*. During the "merge" we remove the delete-elements from the lists/buffers that is in the list resulting from step one (the elements that do not have a matching insert-element).

Figure 11: The delete-removal algorithm

elements in this list from the tree (the buffers of the tree) - the algorithm is indicated in figure 11.

It should be clear that after step one of the algorithm in figure 11 we end up with a list of delete-elements from the tree that do not have a matching insert-elements. By the same argument as in lemma 3 it is easy to see that this step requires $\overline{O}(n)$ I/O's.

It should be equally easy to see that step two actually removes the unmatched delete-elements. To see that also this step uses $\overline{O}(n)$ I/O's, first note that the number of elements in the produced list must obviously be less than the total number of elements in the buffers - which again is less than twice the maximal number of elements in the buffers on the last level. This means that we can merge the "delete-list" with the list of deletes from the last level, in a number of I/O's proportional to the maximal number of blocks on the last level of the tree. When this is done the same property holds for the "delete-list" and the level before the last level - the number of remaining blocks in the "delete-list" is maximally twice the maximally number of blocks in the buffers on this level. This follows from the fact that we know that the elements in the list, actually, is in the tree. Continuing this process up the tree we removes the "unmatched" deletes in a number of I/O's proportional to the maximal number of blocks in the buffers of the tree, that is, in $\overline{O}(n)$ I/O's. □

Lemma 3 and 4 gives us the following:

Corollary 1 *The buffers of a buffer tree with n leaves can be emptied in $\overline{O}(n+r)$ I/O's amortized.*

We are now ready to prove the following:

Theorem 2 *A rangesearch(x_1, x_2) in a buffer tree with n leaves where all elements are distinct uses $\overline{O}(\frac{\log_m n}{B} + r)$ I/O-operations amortized.*

Proof When we insert a rangesearch-element in the buffer of the root, we have to pay $\overline{O}(\frac{\log_m n}{B})$ I/O's in the same way as we did in the proof of Theorem 1. The only thing that remains to be paid for are the tree-writings the rangesearch-elements cause during buffer-empty processes. From corollary 1 we have that one such tree-writing costs $\overline{O}(n'+r')$ I/O's. The r' contribution is charged to the rangesearch-elements that causes the reports. As discussed in the beginning of this section we are willing to pay $\overline{O}(\#$ blocks written), and this can be significantly smaller than $\overline{O}(n')$, as a lot of the elements in the leaves could get erased by delete-elements in the buffers. But as discussed we have $\overline{O}(n') = \overline{O}(\#$ blocks written + # blocks deleted), so we can charge the last part to the *delete* operations and get the desired bound. □

4.2 Algorithms that use the Buffer Tree

It is now easy to see that if we use the plane-sweep algorithm presented in subsection 3 for the orthogonal segment intersection problem, and remember to empty the tree when we are done with the sweep, we get the following (using Theorem 1, Theorem 2 and corollary 1):

Theorem 3 *Using the Buffer tree the orthogonal segment intersection reporting problem can be solved in $\overline{O}(n \log_m n + r)$ I/O's.*

It is equally easy to show the following, using the normal tree-sort algorithm:

Theorem 4 *Using the buffer tree N elements can be sorted in $\overline{O}(n \log_m n)$ I/O-operations.*

As previously discussed both algorithms are optimal. We believe that both algorithms have a number of advantages over known algorithms. Most important the algorithms are precisely like the internal memory algorithms that uses a balanced tree structure. Furthermore, the sorting algorithm is on-line and as mentioned it also works in the D-disk model. As discussed in the introduction we also believe that the algorithms will prove to be faster in practice than previously known algorithms.

4.3 An External Priority-queue

Normally, we can use a search-tree to implement a priority-queue because we know that the smallest element in a search-tree always is in the leftmost leaf. The same strategy can be used when using the buffer tree to implement an external priority queue. There are a couple of problems though because using the buffer tree we cannot be sure that smallest element is in the leftmost leaf, as there can be smaller elements in the buffers of the nodes on the leftmost path. If we are allowing arbitrary delete operations (not only deletes of the minimal element) we can, furthermore, have a delete-element that matches the minimal element. There is, however, a simple strategy to make a *deletemin* operation with the desired amortized I/O-bound.

When we want to perform a *deletemin* operation on the buffer tree we, first, perform a buffer-emptying process on all nodes on the path from the root to the leftmost leaf. To do this we use $\overline{O}(m \cdot \log_m n)$ I/O's. After this, we can be sure not only that the leftmost leaf consists of the B smallest elements, but that (at least) the $\frac{1}{2}m \cdot B$ smallest elements in the tree is in the sons (leaves) of the leftmost leaf-node. So, if we delete these elements and holds them in internal memory, we can "answer" the next $\frac{1}{2}m \cdot B$ *deletemin* operations without doing any I/O-operations. Of course we now also have to check insertions and deletions against the minimal elements in internal memory. This can be done in a straightforward way and a simple amortization argument gives us the following:

Theorem 5 *The amortized I/O-cost of an insert, delete or deletemin operation in an arbitrary sequence of N intermixed such operations in an initially empty buffer tree is $\overline{O}(\frac{\log_m n}{B})$.*

5 An External Segment Tree

In this section we will use our "technique" to develop an external-memory version of the segment tree. As discussed in subsection 3 this will enable us to solve the batched range searching problem, which in turn will give an I/O-algorithm for the isothetic rectangle intersection problem.

In order to construct the external version of the segment tree in the next subsection, we will give a short definition of the segment tree and the operations on it. In a segment tree all begin- and endpoints of segments are stored in sorted order in a (static) balanced binary tree. With each internal node w we associate the interval (s_w) that has the smallest value below w as its begin-point and the largest value below w as its end-point. Moreover, with w a list is associated containing all segments in the tree that contain s_w, but that do not contain $s_{\text{father}(w)}$. See figure 12 for an example of a segment tree.

To perform an inverse range query with a point x we search with x down the tree. When we pass a node w, we know that x is contained in all segments in the list associated with w and we report these. One easily

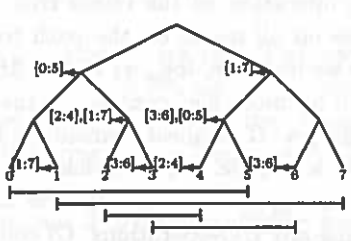


Figure 12: A segment tree containing the segments [0:5], [1:7], [2,4] and [3,6].

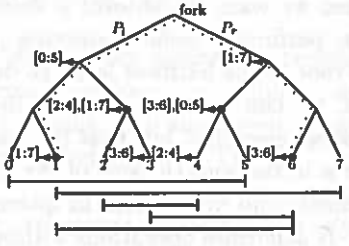


Figure 13: Insertion of [1:6]. The segment is inserted in the marked nodes.

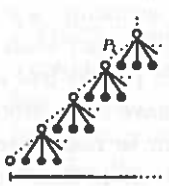


Figure 14: A segment has to be stored in all the marked nodes.

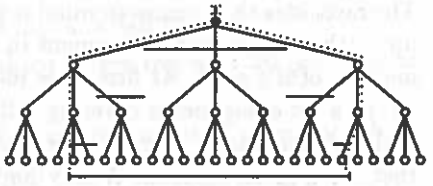


Figure 15: A segment covers all sons to the right/left of a given son in all but one node (the marked one).

verifies that in this way each segment enclosing x is reported exactly once.

An insertion of the segment $[x_1, x_2]$ corresponds to a "tour" down the tree to the two points x_1 and x_2. This tour has a structure that starts with a (possible empty) initial path from the root to a node called the *fork*, where the search for x_1 goes in one direction and the search for x_2 goes in the other direction. The two paths from the fork to x_1 and x_2 are called P_l and P_r, and either the interval is inserted in the fork, or in all right-sons of nodes of P_l which are not on P_l, as well as all left-sons of P_r which are not on P_r. See figure 13 for an example of an insertion of [1:6] into the tree from figure 12. Deletions are done in a simulate way.

It is easy to see that insertions and deletions of segments in the segment tree can be done in logarithmic time, and that inverse range queries can be done in logarithmic time plus the time used to report segments.

5.1 Definition of the Buffered Segment Tree

The first thing we should do when we are trying to construct an external version of the segment-tree is to decrease the height of the tree, that is, make each node have fan-out $\overline{\Theta}(m)$. If we just use the naive approach we are facing a serious problem - while a given segment in the binary version the segment tree is stored in at most two nodes on each level of the tree, it can be stored in $\overline{O}(m)$ nodes in a tree with fan-out m (see figure 14 for an example). This means that if we use this approach we will get something like a $\overline{O}(\frac{m}{B} \cdot \frac{\log_m n}{B})$ solution, which is not what we are after.

The next idea that comes to mind is to store the segments "a level further up" - that is, to store a segment in a node itself rather than in a (big) number of it's sons. At first, this idea may seem just as bad as the first on, as a lot of segments covering different sons have to be stored in each node. Fortunately, there is one key observation to make here, namely that for a given segment it only happens in one node that the segment does not cover all sons to the right (or to the left) of a given son. This is illustrated in figure 15, where it is indicated how a given segment only "split" in one node, namely in the node where the search-path for the left and right endpoints of the segment split. This observation enable us to construct a structure with fan-out $\overline{O}(m)$, where each segment is only stored in a constant number of nodes (independent of m) on each level, plus some constant number of times (dependent on m) in one node.

More precisely, we will do the following for a given segment: In the node where the path for the endpoints split, we will store it in a binary tree internally in the node - just like in an ordinary segment tree. In all other nodes the path for the endpoints traverse after the "split-node", we will store the segment in one list, namely, in a list that contains all segments that covers a given son and all sons to the right/left. The main reason why this will work - besides that each segment is (almost) only stored in two nodes on each level of the tree - is that the number of segment-lists in each node is $\overline{O}(m)$. This means that we (almost) can keep the cost of the buffer-emptying process on $\overline{O}(m)$, and we will then by the same arguments as used in section 4 get the desired bounds.

In order to make the whole thing work there are some technical difficulties though. These has mainly to do with the last level of nodes in the tree. We will not try to explain the details here, but ask the reader not to worry about the details (e.g. that fan-out is $\frac{1}{8}m$) in the following definition but to be patient until the *empty* algorithm is explained in subsection 5.2.2.

The formal definition of the buffered segment tree is as follows:

Definition 3 *A buffered segment tree T for storing segments with endpoints in the set E, where $|E| \leq N$ is defined as follows (see figure 16):*

- *T is a perfectly balanced $\frac{1}{8}m$-ary tree with N leaves.*
- *The following holds for each node v in T:*
  - *v has a buffer of m blocks.*
  - *A buffer contains at most one non-full block.*
  - *v contains a perfectly balanced binary tree with the $\frac{1}{8}m$ sons as leaves.*
  - *Every internal node w of v contains a list of segments whose path for the two endpoints of the segment split in v, and that spans the interval associated with w but not the interval associated with the father of w - just like in a normal (binary) segment tree.*
  - *v has two lists - l_l and l_r - for each of the $\frac{1}{8}m$ sons. l_r of a given son s contains segments in the tree that covers the x-value associated with s and all x-values associated with sons of v to the right of s. Similar with l_l.*
- *Every leaf has a list of segments, such that if the leaf represents the coordinate x, then one of the endpoints of each segment in the list is x.*
- *All the segment-lists in the tree - except for the list associated with the leaves and the nodes in the last level - contain at most one non-full block.*
- *The segment-list of the leaves and the nodes in the last level of the tree consists of blocks that are at least half-filled*
- *Between operations, i.e. when T is "balanced", every buffer contains at most $\frac{1}{2}m$ blocks.*

Lemma 5 *The height of a buffered segment tree on N endpoints is $\overline{O}(\log_m n)$.*

Proof The height is obviously $\lceil \log_{\frac{1}{8}m} N \rceil$ but we have that $\lceil \log_{\frac{1}{8}m} N \rceil = \lceil \log_{\frac{1}{8}m} nB \rceil = \lceil \log_{\frac{1}{8}m} n + \log_{\frac{1}{8}m} B \rceil \in \overline{O}(\log_m n)$. □

5.2 Operations on the Buffered Segment Tree

Normally, when we use a segment tree (in the plane-sweep algorithm) to solve the batched range searching problem, we use the operations *insert*(segment), *delete*(segment) and *search*(point). However, in our external implementation of the segment tree we will not support the *delete*

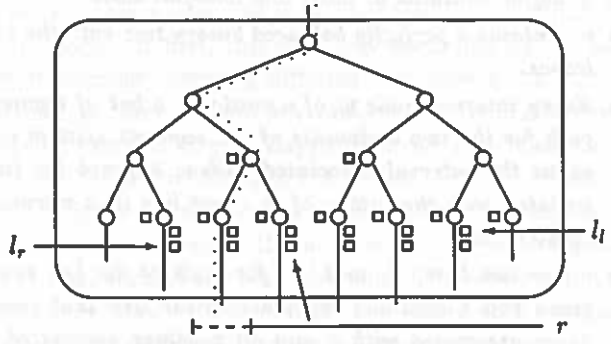


Figure 16: An internal node. The dotted line is the path to the left endpoint of the segment r. r is stored in the l_r list of the node indicated and sent down the tree along the dotted line.

operation. If we take a closer look at the algorithm described in section 3, we find that we already know at which "time" a segment should be deleted when it is inserted in the tree (this holds for all batched dynamic searching problems). So a ***delete*** operation is really not necessary. This means that we will require that a delete-time is given when a segment is inserted in the tree. One may argue that the omission of a ***delete*** operation is an unreasonable restriction to put on the data structure, but on the other hand we already assume that we know the set of x-values of segments to be inserted in the tree, before we actually start building the tree - that is, the design of the segment tree is already very much influenced by the algorithm we are going to use it in.

We will not discuss in details how we can build an empty buffered segment tree in $\overline{O}(n)$ I/O's. The main problem in doing this is that it seems that we have to use N I/O's in order to make the last level of nodes, as there is a total of N internal (binary) nodes and N segment-lists in the nodes on the last level. The solution is simply to store the internal nodes (the split value, references to the two sons, and a pointer to the corresponding segment-list, etc.) in each node in $\overline{O}(\frac{m}{B})$ blocks, and then not actually make room (make an empty block) for the empty segment-list but wait until the first time we actually output something to the list.

In the next subsection we will show how to perform the ***insert*** and the

search operations on the buffered segment tree. In order to use the buffered segment tree in the batched range searching algorithm we also - like in the orthogonal segment intersection algorithm - have to be able to empty the tree when we are done with the sweep that is, be able to empty all buffers while reporting the appropriate segments in $\overline{O}(n+r)$ I/O's. We will address this problem in section 5.2.2.

5.2.1 The *Insert* and *Search* Operations

When we want to perform an ***insert*** or a ***search*** operation on our buffered segment tree, we do exactly like we do an operation on the buffer tree. We make a new element with the segment or point in question, a time-stamp, and - if the element is an insert-element - a delete-time. When we have collected a block of such elements, we insert them in the buffer of the root. If the buffer of the root now contains more than $\frac{1}{2}m$ elements, we perform a buffer-emptying process on it. Like in the buffer tree we would like our buffer-emptying process to use $\overline{O}(m)$ I/O's.

The buffer-emptying process on the internal nodes is presented in figure 17. It is quite easy to see that the definition of the buffered segment tree is maintained by the process, so we will concentrate on the I/O-usage. It is easy to see that step 1 and 4 of the algorithm use $\overline{O}(m)$ I/O's. The main observation in order to see that step 2 uses $\overline{O}(m+r)$ I/O's, is that we use at most one I/O for each l_l and each l_r list and for each internal node that can not be paid for by reportings or by deleted elements - that is, if we during the handling of a single segment-list process more than the first block in the list, then either all the elements in the previous processed block were reported, or some of them were reported and the rest deleted. This means that if we assume that each segment has credits to pay for the deletion, then the buffer-emptying process has to pay at most one I/O per list plus the I/O's used for reportings. As the number of internal nodes is bounded by $\frac{1}{8}m$, and the number of l_r and l_l lists by $2\frac{1}{8}m$, we overall get that this step uses $\overline{O}(m+r)$ I/O's.

As far as step 3 is concerned, the main observation is that the number of I/O's used to store segments that have already split is $\overline{O}(m)$. This follows from the facts that each segment is stored in one list, and that the number of lists is $\frac{1}{4}m$, which means that the maximum number of I/O's used to store non-filled blocks is $\frac{1}{4}m$. The segments that split in

If the current node is an *internal* node:

1. Load the buffer into internal memory.
2. Sort the search-elements and report intersections –
   - with segment-elements from the buffer which do not go to the same son.
   - with segments in the l_r-lists.

     For each of the l_r lists starting from the right do the following:

     Load the first block of the list and report intersection with all points (search-elements) from the buffer going to the present son and all sons to the right of it.

     If there are more blocks in the list *and* if we actually reported any segments from the last block, then load the next block and continue as with the first block.

     If we during this process observe that any of the segment-elements have expired (if we are past their delete-time) we delete them.
   - with segments in the l_l-lists.

     Similar to l_r.
   - with segments in the lists of the internal nodes.

     For each of the internal node do the following:

     Load the first block from the list and report intersections with all points whose path goes through the node.

     If there are more blocks in the list *and* if we actually reported any elements from the last block, then load the next block and continue as with the first block.

     If we during this process observe that any of the segment-elements have expired (if we are past their delete-time) we delete them.
3. Store segments –
   - that have already split.

     Divide the elements into two groups consisting of "left-point segments" and "right-point segments", and output the segments to the appropriate list of l_r and l_l, respectively. By appropriate we mean the list of the son immediate to the right/left of the son where the endpoint goes (see figure 16).
   - that split in the node.

     For all the segment-elements that split in this node do the following:

     Store the segment in the lists of the internal nodes, just like in a normal internal segment tree. Make two new elements consisting of the segment and the right and left endpoint, respectively.
4. Output the search-elements and the segment-elements - both the "old" and the new ones produced in the previous step - to the appropriate sons.
5. If the buffer of any node contains more than $\frac{1}{2}m$ blocks, the buffer-emptying process is recursively applied to these nodes.

Figure 17: The buffer-emptying process on internal nodes.

If the current node is a *leaf* node:

We do exactly the same as with the internal nodes, except that we only output *segment-elements* to segment-lists (or to the "leaf-lists") if we have more than $\frac{B}{2}$ elements for the same list. Otherwise, we just keep the element at the buffer.

As far as the *search-elements* are concerned, we just report intersections with segments from the lists (remembering the lists in the leaves) and remove the search-elements.

Figure 18: The Buffer-emptying process on leaf nodes.

the node is a little more complicated. It is obvious that $2\log_2 \frac{1}{8}m$ is the maximum number of lists we ever need to store a segment that split in. The crucial thing is now that we can make full use of blocks, so that if x is the number of segments that split in the node, then the total number of I/O's used on these segments is $\overline{O}(m + x \cdot \frac{\log_2 m}{B})$. This follows from the fact that the maximum number of I/O's we ever have to use on non-filled blocks is bounded by the number of internal nodes. Putting it all together this means that the buffer-emptying process on an internal node costs $\overline{O}(m + r + x \cdot \frac{\log_2 m}{B})$ I/O's, if r is the number of blocks reported in the process, and x the number of elements in the buffer that "splits" in the node in question.

The buffer-emptying process for leaf nodes is presented in figure 18, and almost precisely the same arguments as the above one can be used on these nodes. However here we also have to estimate how many elements may stay in the buffer. As we have less than $4\frac{1}{8}m$ lists, and as elements are output to the lists when more than half a block goes to the same list, the maximal number of elements that can stay at the buffer is $4\frac{1}{8}m \cdot \frac{B}{2}$. As this equals $\frac{m}{4}$ blocks - which means that we as in the internal case sends $\overline{O}(m)$ blocks downwards - we can, finally, put the hole argument together:

As in the buffer tree we can imagine that we on insertion put $\overline{O}(\frac{\log_m n + \log_2 m}{B})$ credits on each element. It then follows by the same argument as we used on the buffer tree, and the fact that a segment only "splits" once that we have enough credits to pay for the buffer-emptying, except for the reportings which we charge to the relevant search-elements. As $\overline{O}(\frac{\log_m n + \log_2 m}{B}) \subseteq \overline{O}(\frac{\log_m n}{B})$ we have the following:

Theorem 6 *The amortized number of I/O-operations used by an insert or search operation in a sequence of N such operations is $\overline{O}(\frac{\log_m n}{B})$ and $\overline{O}(\frac{\log_m n}{B} + r)$, respectively.*

In subsection 5.4, we will take a closer look at how much we are "cheating" with $\overline{O}$ by removing the $\frac{\log_2 m}{B}$ term.

5.2.2 The *Empty* Operation

We will now take a closer look at the ***empty*** operation. Essentially the empty algorithm will be like the write algorithm we used on the buffer tree. Here however, we will only remove the search-elements from the tree, that is, after a ***empty*** operation the buffers of the tree will not be completely empty, as there can still be segment-elements in them. Furthermore, we will only prove that we can do the ***empty*** operation in a linear number of I/O's *amortized*. Actually, we can design an algorithm such that all buffers are totally empty after an ***empty*** operation, and such that it only takes $\overline{O}(n)$ I/O's worst-case but as this will not affect the batched range searching algorithm we will focus on the simple algorithm here.

When we is to perform a ***empty*** operation we simply level-wise perform buffer-emptying processes on (almost) all nodes in the tree, starting from the top level (the root) and ending in the second-lowest level. Actually, for the non-recursively buffer-emptying processes we only do the part of the process that involves reporting intersections and sending search-elements down the tree that is, we do not send segment-element down the tree. Then the number of nodes we do a buffer-emptying process on is bounded by $2\frac{N}{(\frac{m}{8})^2}$, and as recursively buffer-emptying processes already are paid for, we totally have to pay $\overline{O}(\cdot\frac{N}{m^2}\cdot m + r) \in \overline{O}(n + r)$ for this part of the algorithm.

Now we are left with the problem of "emptying" the buffers of the nodes on the last level. At first glance it might seem as an impossible job to do in $\overline{O}(n)$ I/O's, as there is $\frac{N}{m}$ nodes on the last level, and as one buffer-emptying process costs $\overline{O}(m)$ I/O's. However, the total number of blocks in the buffers on the last level can only be n, so we could hope to reduce the total number of I/O's used to empty these buffers correspondingly. This may again seem impossible, as we in order to empty one buffer normally have to check $\overline{O}(m)$ segment-lists for intersections. But if we had information about which of the lists where empty and which where not, we would actually be able to do the job. This follows from the fact that we have designed the tree such that each of the blocks in the

lists contains at least $\frac{B}{2}$ segment-elements. This means that if we have just one search-element that causes reporting of the elements from a give list, then the I/O-operation used can be paid for by the reporting (or by "delete-credits" if some of the segments is to be deleted). But during the hole algorithm we can easily maintain empty/non-empty information in $\overline{O}(\frac{m}{B})$ blocks for each node in the last level. In total we - apart from the I/O's charged to reportings - only use $\overline{O}(n)$ I/O's to load the buffers and $\overline{O}(\frac{m}{B})\cdot\frac{N}{m} \in \overline{O}(n)$ I/O's to load the empty/not empty information during the process of "emptying" all the buffers on the last level. This ends the proof of the following:

Theorem 7 ***The amortized number of I/O-operations used by an empty operation is*** $\overline{O}(n + r)$.

5.3 Algorithms that use the Buffered Segment Tree

Using the plane-sweep algorithm presented in section 3, and remembering to empty the buffered segment tree when we are done with the sweep, it is easy to see the following (using Theorem 6 and 7):

Theorem 8 ***The batched range searching problem can be solved in*** $\overline{O}(n\log_m n + r)$ ***I/O's.***

As discussed in section 3 this leads to the following:

Theorem 9 ***The isothetic rectangle intersection problem can be solved in*** $\overline{O}(n\log_m n + r)$ ***I/O's.***

That both these algorithms are optimal follows, as in the case with orthogonal segment intersection, by the comparison lower-bound and the result in [3].

5.4 Improving the term hidden in $\overline{O}$

In section 5.2.1 we were "hiding" a $\frac{\log_2 m}{B}$-term in the $\overline{O}$ notation. In this subsection we will take a closer look at how much this term matters by

inserting typical values for m and B in the results. We will also show how to reduce the term.

In section 5.2.1 we analyzed the amortized cost per *insert* and *search* operation to be $\frac{k_1 \log_m n}{B} + \frac{k_2 \log_2 m}{B}$ I/O's. So how big should n be in order to make the $k_2 \log_2 m$-term negligible in practice? If one takes a closer look at the buffer-emptying algorithm one finds that k_2 is approximately 1, while k_1 is somewhere between 5 and 10. If we estimate k_1 to be 10, insert the typical value for m and solves the inequality $10 \log_m n > \log_2 m$ we get that n should be greater that approximately m - that is, the problem should be bigger than a memory-load. If we estimate k_1 to be 5 we get that n should be bigger than approximately $m^2 \simeq 1000^2 \simeq 2^{20}$. This is not too bad but actually we can improve the result even further.

Consider a segment-element that splits in a given node. In the presented algorithm we store the element in $\simeq \log_2 m$ lists in order to solve the problem. But what is really the problem? - we have a segment with endpoints among m values which we want to store! Why not recursively use the method we just invented? So what we could do is that we could group the binary nodes in the internal of a given node into nodes with say x sons. Now we can use the same idea as previous, and store the segment in less than $\log_x m$ of the new nodes (in l_l and l_r lists), and in less than $\log_2 x$ binary nodes in the node where the segment splits. It is pretty easy to see that this will not in any crucial way effect the presented algorithms - we will have to check some more segment-lists during the buffer-emptying, but the number is still $\overline{O}(m)$ (this means that k_1 will be bigger than before, but we will ignore that here). Now it only remains to find the best value for x, that is, minimize the function $\frac{\log_x m + \log_2 x}{B}$. One easily finds that x should be $2^{(\sqrt{\log_2 m})}$ and this leads to a $2\sqrt{\log_2 m}$-term in the overall algorithm instead of the $\log_2 m$-term. This again reduces the estimation of how big n should be from $2^{10} - 2^{20}$ to $2^6 - 2^{12}$.

6 Conclusion and Open Problems

In this paper we have developed a technique for transforming an internal data structure into an external data structure suitable for plane-sweep algorithms. The main idea in this technique is the introduction of buffers, where the elements are not pushed downwards from one level to the next

before the buffer runs full. It would be interesting to investigate if this technique works on other data structures (e.g. the priority-search tree or the interval tree). It would also be interesting to see if the idea maybe leads to algorithms for problems that has not jet been solved (optimally) in the I/O-model - or to on-line or dynamic I/O-algorithm which is an almost unexplored area. The problem of finding all intersecting pairs of N non-orthogonal segments is an especially challenging open problem.

Almost all the I/O-bounds in this paper are amortized. We find it a very interesting problem to try to bound the worst-case number of buffer-emptying processed caused by insertion of a single block. It would also be interesting to investigate how good a I/O-bound we could obtain for an un-batched *search* operation. Another open problem in connection with the structures developed in this paper is to try to make a *delete* operation on the buffered segment tree.

Finally, it could be interesting to explore if the structures works in some of the multilevel hierarchical memory and parallel models [10] [15], or develop more general structures for these models.

Acknowledgments

The author would like to thank all the people in the algorithmic group at Aarhus University for valuable help and inspiration. Special thanks go to Mikael Knudsen for the discussions that lead to many of the results in this paper, to Sven Skyum for many computational geometry discussions, and to Peter Bro Miltersen and Erik Meineche Schmidt for help on improving the presentation of the results in this paper.

References

[1] Aggarwal, A., Vitter, J.S.: The I/O Complexity of Sorting and Related Problems. Proceedings of 14th ICALP (1987), Lecture Notes in Computer Science 267, Springer Verlag, 467-478, and: The Input/Output Complexity of Sorting and Related Problems. Communications of the ACM, Vol 31 (9) (1988) 1116-1127.

[2] Aggarwal, A., Chandra, A.K., Snir, M.: Hierarchical Memory with Block Transfer. Proceedings of 28th FOCS (1987), 204-216.

[3] Arge, L., Knudsen, M., Larsen, K.: A General Lower Bound on the I/O-Complexity of Comparison-based Algorithms. Proceedings of 3rd WADS (1993), Lecture Notes in Computer Science 709, Springer Verlag, 83-94.

[4] Bentley, J.L, Wood, D.: An Optimal Worst Case Algorithm for Reporting Intersections of Rectangles. IEEE Transactions on Computers 29 (1980), 571-577.

[5] Edelsbrunner, H., Overmars, M.H.: Batched Dynamic Solutions to Decomposable Searching Problems. Journal of Algorithms 6 (1985), 515-542.

[6] Huddleston, S., Mehlhorn, K.: A New Data Structure for Representing Sorted Lists. Acta Informatica 17 (1982), 157-184.

[7] Icking, Ch., Klein, R., Ottmann, Th.: Priority Search Trees in Secondary Memory. Proceedings of '87, Graph-Theoretic Concepts in Computer Science, Lecture Notes in Computer Science 314, Springer-Verlag, 84-93.

[8] Knuth, D.E.: The Art of Computer Programming, Vol 3: Sorting and Searching, Addison-Wesley (1973).

[9] Nodine, Mark H., Vitter, Jeffrey S.: Optimal Deterministic Sorting on Parallel Disks. Brown University CS-92-08, August 1992.

[10] Nodine, Mark H., Vitter, Jeffrey S.: Optimal Deterministic Sorting in Parallel Memory Hierarchies. Brown University CS-92-38, August 1992.

[11] Yale, N.P.: The I/O Subsystem - A Candidate for Improvement. Guest Editor's Introduction in Computer 27 (3) (march 1994), 15-16.

[12] Preparata, F., Shamos, M.: Computational Geometry, An Introduction. Text and Monographs in Computer Science, Springer-Verlag 1985.

[13] Goodrich, M.T, Tsay, J., Vengroff, D.E., Vitter, J.S.: External-Memory Computational Geometry. Proceedings of 34th FOCS (1993), 714-723.

[14] Smid, M.: Dynamic Data Structures on Multiple Storage Media. Ph.D thesis University of Amsterdam 1989.

[15] Vitter, J.S: Efficient Memory Access in Large-Scale Computation (invited paper). Proceedings of 8th STACS (1991), Lecture Notes in Computer Science 480, Springer-Verlag, 26-41.

[16] Vitter, J.S., Shriver, E.A.M.: Optimal Disk I/O with Parallel Block Transfer. Proceedings of 22nd STOC (1990), 159-169.

Recent Publications in the BRICS Report Series

RS-94-6 Mogens Nielsen and Christian Clausen. *Bisimulations, Games and Logic*. April 1994, 37 pp. Full version of paper to appear in: *New Results and Trends in Computer Science*, LNCS, 1994.

RS-94-7 André Joyal, Mogens Nielsen, and Glynn Winskel. *Bisimulation from Open Maps*. May 1994, 42 pp. Journal version of LICS '93 paper.

RS-94-8 Javier Esparza and Mogens Nielsen. *Decidability Issues for Petri Nets*. 1994, 23 pp.

RS-94-9 Gordon Plotkin and Glynn Winskel. *Bistructures, Bidomains and Linear Logic*. May 1994, 16 pp. To appear in the proceedings of ICALP '94, LNCS, 1994.

RS-94-10 Jakob Jensen, Michael Jørgensen, and Nils Klarlund. *Monadic second-order Logic for Parameterized Verification*. May 1994, 14 pp.

RS-94-11 Nils Klarlund. *A Homomorphism Concept for ω-Regularity*. May 1994.

RS-94-12 Glynn Winskel and Mogens Nielsen. *Models for Concurrency*. May 1994, 144 pp. To appear as a chapter for the *Handbook of Logic and the Foundations of Computer Science*, Oxford University Press.

RS-94-13 Glynn Winskel. *Stable Bistructure Models of PCF*. May 1994, 26 pp. *Preliminary draft*. Invited lecture for MFCS 94. To appear in the LNCS proceedings of MFCS 94.

RS-94-14 Nils Klarlund. *The Limit View of Infinite Computations*. May 1994, 16 pp. To appear in proceedings of Concur '94.

RS-94-15 Mogens Nielsen and Glynn Winskel. *Petri Nets and Bisimulations*. May 1994, 36 pp.

RS-94-16 Lars Arge. *External-Storage Data Structures for Plane-Sweep Algorithms*. June 1994, 27 pp.