The Design of the Cell Tree:  
An Object-Oriented Index Structure for Geometric Databases

Oliver Günther

Department of Computer Science
University of California
Santa Barbara, CA 93106

Abstract

This paper describes the design of the cell tree, an object-oriented dynamic index structure for geometric databases. The data objects in the database are represented as unions of convex point sets (cells). The cell tree is a balanced tree structure whose leaves contain the cells and whose interior nodes correspond to a hierarchy of nested convex polyhedra. This index structure allows quick access to the cells (and thereby to the data objects) that occupy a given location in space. Furthermore, the cell tree is designed for paged secondary memory to minimize the number of disk accesses occurring during a tree search. Point locations and range searches can therefore be carried out very efficiently using the cell tree.

1. Introduction

Modern database systems are no longer limited to business applications. Non-standard applications such as robotics, computer vision, computer-aided design, and geographic data processing are becoming increasingly important, and geometric data play a crucial role in many of these new applications.

For efficiency reasons it is essential that the special properties of geometric data objects (such as points, lines, polygons, polyhedra, or splines) be fully utilized in the database management system. In particular, special operators that are defined on geometric data need to be supported. These operators are substantially different from the operators defined on numerical data. In particular, we distinguish between

- set operators: union, intersection, set difference;
- search operators: point location, range search;
- similarity operators: translation, rotation, and scaling; and
- recognition operators.

With the possible exception of the similarity operators, all of these operators are harder to compute than most common numerical ones.

This paper discusses search operators and suitable geometric index structures to support them. Given a set of geometric objects in $d$-dimensional Euclidean space $\mathbb{E}^d$, stored in a geometric database, a range search computes those objects in the database that overlap a given search space $S \subseteq \mathbb{E}^d$. In the point search problem, which can be viewed as a degenerate range search, one determines all objects in the database that contain a given point $ACE$. Both operators require fast access to objects in the database, depending on their location in space. Therefore geometric index structures can be used efficiently to support these operators.

Indices for the computation of (one-dimensional) search operators play an important role in conventional database systems [1,2]. Indices should be dynamic with respect to updates of the database, i.e. it should be possible to perform insertions and deletions without having to reorganize the database completely. Furthermore, an index should minimize the number of page faults that occur during a search operation.

Section 2 gives a brief survey of the most well-known data structures and indices for the support of geometric search operators. Section 3 describes a scheme for a geometric database where all data objects are represented as algebraic sums of convex point sets (cells). Section 4 introduces an index for this database, viz., a new hierarchical data structure termed cell tree, and describes how to perform search operations. Section 5 gives algorithms to perform insertions and deletions.

2. Geometric Index Structures

There is a whole variety of well-known data structures for the computation of geometric search operators. Quadtrees [19] are designed to organize two-dimensional data. The decomposition process starts from a square that contains all objects to be represented, and proceeds with a recursive subdivision into four equal-sized quadrants. Corresponding to this subdivision is a tree structure of degree four (the quadtree), i.e. each node has exactly four descendants. Due to the regular decomposition, quadtrees do not adapt to the distribution of the data objects in space nor to changes in the underlying database. Objects whose boundaries do not fit into the rectilinear partition of the quadtree can only be represented approximately, or in form of an object description attached to the leaves of the quadtree. A further disadvantage of the region quadtree is that it does not take paging of secondary memory into...
account. In general, each node required during a search may cause a page fault. As quadtrees may be very deep, the paging costs incurred this way may be considerable. If quadtrees are generalized to higher dimensions, the branching factor is $2^d$ for $d$ dimensions.

Binary space partitioning (BSP) trees [4,5] are binary trees that represent a recursive subdivision of a given space into subspaces by means of $(d-1)$-dimensional hyperplanes. Each subspace is subdivided independent of its history and of the other subspaces. Each hyperplane corresponds to an interior node of the tree, and each partition corresponds to a leaf. Figure 1 gives an example of a BSP and the corresponding BSP tree. BSP trees are much more adaptive than quadtrees. However, they are typically very deep which has a negative impact on tree performance. They also do not account for paging of secondary memory.

![Fig. 1: A binary space partitioning with BSP tree.](image)

The first geometric index structure that has been designed specifically for paged memory is Robinson's k-d-B-tree [11], a generalization of Bayer's B-tree [1] to higher dimensions. k-d-B trees are designed for indexing points in arbitrary dimensions; a generalization to extended geometric objects (such as polyhedra) is not possible. The same restriction holds for three non-hierarchical point indices: Tamminen's EXCELL [23], Nievergelt's gridfile [14], and for a hash-based access method designed by Kriegel and Seeger [13]. All these index structures are dynamic, i.e. insertions and deletions of objects can be interleaved with searches and no periodic reorganization is required.

A direct generalization of these point index structures to handle extended objects is not possible. However, there are approaches to use these structures for extended objects. If the objects are $d$-dimensional intervals (boxes) they can be represented as points in 2d-dimensional space (the point space) [12]. Then the search operators can be formulated as point queries in point space and computed by means of a point index. The mapping from original space into point space has to be chosen carefully such that the image points are not distributed too unevenly [3,12,15]. The disadvantages of this method are as follows. First, the formulation of range queries in point space is much more complicated than it is in the original space. Second, the images of two intervals that are nearby in original space may be arbitrarily far apart from each other in point space [3]. Especially in the case of the k-d-B-tree, these problems do in fact cause serious performance penalties [6]. Non-hierarchical index structures such as the gridfile or EXCELL, which are based on address computation techniques, seem to be less affected by this.

Another promising approach has been proposed by Six and Widmayer [21]. It can not only be used for the management of $d$-dimensional intervals, but for arbitrary extended $d$-dimensional geometric objects. The objects are indexed by means of a layering of several point indices. Six and Widmayer use a three-layer gridfile to demonstrate the advantages of their approach.

There are also several index structures that have been designed a priori as secondary storage indices for extended objects. The first such structure was Guttman's R-tree [11], also a generalization of the B-tree to higher dimensions. R-trees are balanced trees that correspond to a nesting of $d$-dimensional intervals (Fig. 2). Each node $N$ corresponds to a disk page $D(N)$ and an interval $I(N)$. If $N$ is an interior node then all intervals corresponding to the immediate descendants of $N$ are subsets of $I(N)$ and stored on the disk page $D(N)$. If $N$ is a leaf node then $D(N)$ also contains a number of intervals that are subsets of $I(N)$. Each of these data intervals is wrapped tightly around a data object. For data objects that are not intervals themselves, the R-tree can therefore not solve a given search problem completely. One rather obtains a set of intervals whose enclosed objects may intersect the search space. One is left with the problem of bringing the objects into main memory and testing them for intersection with the search space. This step, which may require additional disk accesses and considerable computations, has not been taken into account by existing performance analyses [6,11].

![Fig. 2: An R-tree with data objects (shaded).](image)

As in the case of the B-tree, there is an upper and lower bound for the number of descendants of an interior node. The lower bound prevents the degeneration of trees and leads to an efficient storage utilization. Nodes whose number of descendants drops below the lower bound are deleted and its descendants are distributed among the remaining nodes.
(tree condensation). The upper bound can be derived from the fact that each tree node corresponds to exactly one disk page. Once a node requires more than one disk page, it is split and its descendants are distributed among the two resulting nodes. Each splitting may propagate up the tree, i.e. it may be necessary to split the ancestor node as well, and so on.

Furthermore, it should be noted that sibling nodes, i.e. nodes whose ancestor nodes are identical, may correspond to overlapping intervals. This property of the R-tree facilitates the insertion and deletion of data objects, but it may lead to performance losses during search operations. In the case of point locations, one may have to traverse several search paths. Even for range searches, the number of nodes to be inspected tends to be higher with overlaps. These problems led to the development of techniques to minimize the overlap [18] and to the R+-tree [20,22] where no overlaps are allowed (see Fig. 3).

![Fig. 3: An R+-tree with data objects (shaded).](image)

For the reasons mentioned above, the R+-tree allows the fast computation of search operators. However, the insertion and deletion of data objects may be much more complicated in turn.

First, the insertion of an object $O$ or its data interval $I_0$ may require the enlargement of several sibling intervals (i.e. intervals corresponding to sibling nodes). This is especially (but not exclusively) the case if $I_0$ overlaps several sibling intervals (see Fig. 4). Each of these enlargements may require a considerable effort because it is always necessary to test for possible overlaps with sibling intervals. $I_0$ is inserted into all corresponding subtrees; the insertion may therefore cause the creation of several leaf entries. In the R-tree, on the other hand, no more than one interval per tree level will be enlarged; possible overlaps that result from the enlargement do not matter. Consequently, each insertion causes the creation of exactly one additional leaf entry.

Second, there are situations where the enlargement step inevitably leads to overlaps (see Fig. 5). In this case, it is necessary to split one or more sibling intervals before the enlargements can take place.

![Fig. 4: The data interval $I_0$ overlaps the sibling intervals $I_1$ and $I_2$. In this case $I_0$ has to be inserted into both corresponding subtrees. $I_1$ and $I_2$ have to be enlarged in such a way that $I_0 \subseteq I_1 \cup I_2$ (without $I_0$ overlapping $I_2$).](image)

Third, node splittings have to be propagated not only up the tree (as in the case of the R-tree) but also down the tree. The splitting of an interior node $N$ corresponds to a splitting of the corresponding interval $I(N)$ by means of a hyperplane $H$. If there are intervals corresponding to descendant nodes of $N$ that intersect $H$, then these intervals have to be split along $H$ as well, and so on. It is therefore necessary to pick $H$ very carefully to avoid complicated node splittings.

Fourth, it is no more possible to maintain an upper bound on the number of leaf node entries, i.e. there may be leaves that require more than one disk page of storage space. In particular, this is inevitable if there is a point in space that is covered by more data intervals than what can be stored on one disk page. The interval covering that point certainly contains too many data intervals for one disk page. To avoid complex update algorithms, the R+-tree does not impose any upper or lower bounds on the number of descendants of an interior node. The degeneration of an R+-tree is therefore possible, and storage utilization may deteriorate.
The main goal during the design of the cell tree was to facilitate searches on data objects of arbitrary shapes, i.e. especially on data objects which are not intervals themselves. Especially in robotics and computer vision, intervals are not necessarily a good approximation of the data objects enclosed. Now each tree node corresponds not necessarily to an interval, but to a convex polyhedron. To optimize search performance, we decided to avoid overlaps between sibling polyhedra. In subsequent chapters, it will be shown how the resulting disadvantages can be partly compensated by restricting the polyhedra to be partitions of a BSP (binary space partitioning) (see Fig. 1). Therefore the cell tree can be viewed as a combination of a BSP- and an R'-tree.

3. The Geometric Database

Consider a database consisting of a collection of d-dimensional point sets in Euclidean space \( \mathbb{E}^d \). In order to support search and set operations efficiently, we represent the data objects as convex chains, i.e. as sums of convex point sets \([7,24]\). Formally, each data object \( D \) is represented as a convex chain in \( \mathbb{E}^d \):

\[
x_D = \sum_{i=1}^{m} x_i
\]

The cells \( x_i \) are \( d \)-dimensional convex closed point sets that are not necessarily bounded (cells). Note that we do not require the cells to be mutually disjoint. Disjointness is hard to maintain and provides no particular advantages for the operators we intend to support. Therefore the cells form a convex cover of the data object. We consider a point \( t \in \mathbb{E}^d \) inside \( D \) if and only if it is inside any of the cells, i.e.

\[
t \in D \iff t \in x_i \text{ for some } i=1 \ldots m
\]

Convex chains are a simple and powerful tool to describe various kinds of geometric objects (Fig. 6). Unlike simple point sets, convex chains are closed under the set operations union, intersection, and difference. Although the decomposition of the original data objects into cells will take some computation time, we believe that it will eventually pay off by making searches and updates simpler and faster [7]. Note that this decomposition is completely transparent to the user. Cells need not be seen or manipulated by the user.

4. The Cell Tree

4.1. Description

A cell tree indexes the cells in a geometric database depending on their location in space. As the R-tree, a cell tree is a height-balanced tree and each tree node corresponds to exactly one disk page. The computation of a search operator should therefore cause only a small number of page faults. The cell tree is a fully dynamic index structure; insertions and deletions can be interleaved with searches and no periodic reorganization is required.

Each leaf node entry represents a cell \( E.Z \). In addition to a description of the cell geometry, it contains the ID \( E.D \) of the data object whose convex chain \( x_D \) contains the cell \( E.Z \). \( E \) also contains any additional properties of the data object \( E.D \) that may be necessary to answer a given query. Examples for such properties are the color or the geometry of \( E.D \).

Interior (non-leaf) nodes contain entries of the form

\[(cp, P, C)\]

Here, \( cp \) is the child pointer, i.e. the address of the corresponding descendant node. \( P \) is a convex, not necessarily bounded \( d \)-dimensional polyhedron. All cells in the database that are subsets of \( P \) are in the subtree under the descendant node. The container \( C \) is a convex subset of \( P \), which also contains each cell \( P \) in the subtree, i.e. \( p \subseteq C \). \( C \) provides a more accurate localization of these cells, which may speed up search queries. In the following, \( E.cp, E.P, \) and \( E.C \) denote the corresponding attributes of an interior node entry \( E \). \( m \) is a parameter specifying the minimum number of entries in an interior node. Finally, given a node \( N \), its entry in its ancestor node is denoted by \( E.N \) and the entries in \( N \) are denoted by \( E(N) \).

A cell tree satisfies the following properties.

1. The root node has at least two entries unless it is a leaf.
2. Each interior node has at least \( m \) entries unless it is the root.
3. For each entry \( (cp, P, C) \) in an interior node, the subtree that \( cp \) points to contains a cell \( P \) if and only if \( p \subseteq P \).
4. For each entry \( (cp, P, C) \) in an interior node, the container \( C \subseteq P \) is a convex polyhedron that can be specified as the intersection of \( P \) with at most \( k \) halfspaces in \( \mathbb{E}^d \). For each cell \( p \) in the subtree pointed to by \( cp \), it is \( p \subseteq C \).
5. For each interior node \( N \), the polyhedra \( E(N).P \) form a binary space partitioning (BSP) of \( E_N \).
6. All leaves are on the same level.
Almost every node requires no more than one disk page of storage space. Figure 7 shows an example cell tree.

Property (5) has two interesting consequences. First, the polyhedra \(E_1.P\) and \(E_2.P\) corresponding to two entries \(E_1\) and \(E_2\) on the same tree level cannot overlap. Second, the entries of an interior tree node \(N\) can now be stored in a very compact manner: rather than describing the polyhedra \(E,(N).P\) explicitly, one only stores the BSP of \(E,(N).P\).

Note that the cell tree (just as the \(R^*-\)tree) does not put any upper bounds on the number of node entries. Nevertheless, it is attempted to limit the storage requirements for each node to one disk page. If a node exceeds one disk page of storage after one or more insertions, it is attempted to split that node. In most cases this is in fact possible; see section 5.3. If the splitting does not succeed, then the node is stored using overflow pages; these cases are the only ones that the word almost in property (7) refers to.

Other than the \(R^*-\)tree, however, the cell tree has a lower bound for the number of entries of an interior node. Of course, it is also possible to define the cell tree with a lower bound for the number of leaf entries. To keep insertions and deletions simple, however, we have decided not to impose such a bound.

In order to analyze the maximum number \(M\) of entries of an interior node, such that the node can still be stored on one disk page, we denote the page size by \(ps\), and the number of bytes required to store a number or a pointer by \(q\). Each interior node entry \(E,(N)\) requires \(q\) bytes for the pointer \(E,(N).cp\), and \(k\cdot d\cdot q\) bytes for those \(k\) \((d-1)\)-dimensional hyperplanes that specify the container \(E,(N).C\) if \(E,(N).P\) is known. The polyhedra \(E,(N).P\) form a BSP of \(E,(N).P\) with no more than \(M\) partitions. Therefore, the corresponding BSP-tree requires the storage of no more than \(M-1\) hyperplanes and \(2M-2\) pointers. The total number of bytes to store a full interior node is therefore

\[
M\cdot(q+k\cdot d\cdot q) + (M-1)\cdot d\cdot q + (2M-2)\cdot q
\]

\[
= q \cdot (M\cdot(k+1)\cdot d+3) - d - 2
\]

As one node corresponds to one disk page of \(ps\) bytes, we obtain

\[
M = \frac{ps/q+d+2}{(k+1)\cdot d+3}
\]

For a typical set of parameters, such as \(ps=1024\), \(q=4\), \(d=2\), and \(k=2\), one obtains \(M=28\).

A similar analysis for the maximum number of leaf node entries is not as easy as the storage requirements per leaf node entry \(E\) may vary considerably, depending on the complexity of the cell \(E.Z\).

The height of a cell tree containing \(n\) index records is bound by \(\lceil\log_m n\rceil+1\), because the branching factor of each node is at least \(m\). Except for the root and for overloaded nodes, the worst-case space utilization is \(m/M\) for interior nodes.

If a new data object is inserted into the database, one first computes a convex cover of the object. Each convex component is then inserted into the cell tree. The number of components per data object is highly data-dependent. If all data objects are convex (as it is actually the case for CAD layout data, for example), of course there would be only one component per data object. Note that the insertion of a convex component into the cell tree may cause the creation of several leaf node entries (i.e. cells). As confirmed by empirical results [5,9], the average number of cells is usually no more than twice the number of components. However, this number is highly data-dependent. In exceptional cases, where there was a lot of overlap between objects in the database, the tree height or four cells per component have been observed.

The parameters \(m\) and \(k\) are to be varied as part of the performance tuning. A large \(m\) will increase the space efficiency and decrease the height of the tree, which might in turn improve the search performance. On the other hand, a large \(m\) may cause updates to become very expensive, as tree condensations will occur more frequently and become more complex (see section 5.4). Initial tests indicate that \(m\) should be chosen small, i.e. \(m=2\) or \(m=3\) [9]. A large value for \(k\) allows a more accurate localization of the cells in a subtree, which might improve the search performance. On the other hand, \(k\) and \(M\) are inversely proportional. A large \(k\) will therefore yield a small \(M\). This might in turn increase the tree height and decrease the search performance. Furthermore, a large \(k\) makes updates more complicated.

4.2. Searching

The cell tree allows efficient searches such as to find all data objects that overlap a search space, where the search space may be of arbitrary shape. We give the algorithm for this search problem; other searches can be implemented by variations of this algorithm.

The search algorithm first computes a convex cover of the search space. For each convex
component, the search algorithm descends the tree from the root in a manner similar to a B-tree or an R-tree. At each interior node the search space is decomposed further into several disjoint convex subspaces, and a not necessarily convex remainder space. The remainder space is insignificant to the search and therefore eliminated. The convex subspaces are each passed to one of the subtrees to be decomposed recursively in the same manner. Note that this algorithm differs from the equivalent R-tree algorithm where the subspaces are allowed to overlap, thereby decreasing the search efficiency.

Algorithm \text{Search}((T,S)) \text{.} Given a cell tree with root node \( T \), find all data objects that overlap a search space \( S \).

1. \[ \text{[Decompose \( S \).]} \text{ If \( S \) is not convex, find a (small) set of convex polyhedra \( S_i \) such that } \sum S_i = S. \text{ For each } S_i, \text{ \text{Search}((T,S_i)) \text{ and stop.} } \]

2. \[ \text{[Search subtree.]} \text{ If } T \text{ is not a leaf, check each entry } E_i(T) \text{ to determine whether the \( E_i(T).C \) overlaps } S. \text{ If yes, \text{Search}((T',S \setminus E_i(T).C)) \text{ where } T' \text{ denotes the node } E_i(T).cp \text{ points to.} \]

3. \[ \text{[Search leaf node.]} \text{ If } T \text{ is a leaf, check each entry } E_i(T) \text{ to determine whether the cell } E_i(T).Z \text{ overlaps } S. \text{ If yes, return the data object } E_i(T).D. \]

Other than step 1, the main effort in this algorithm is to detect and compute overlaps between the search range \( S \) on one hand and the containers \( E_i(T).C \) or the cells \( E_i(T).Z \) on the other hand. A very efficient method to perform these computations is based on a dual representation for the search range, the containers and the cells [8]. Using this representation, the time complexity per entry for steps 2 and 3 is not much higher than in the case of the R- or \( R^+ \)-tree [9].

5. Updating the Cell Tree

5.1. Insertion

To insert a new data object, one first computes a convex cover of the object. Then each component in the cover is inserted separately into the cell tree. Note that the insertion of one component may cause the creation of more than one new leaf node entries (i.e., cells). Inserting index records for new cells is similar to insertion into a B- or R-tree. Index records are added to the leaves, nodes that overflow are split, and splits propagate up the tree (see section 5.3).

Algorithm \text{InsertObject}((T,D)). Insert a new data object \( D \) into a cell tree with root node \( T \).

1. \[ \text{[Decompose \( D \).]} \text{ If \( D \) is not convex, find a (small) set of convex polyhedra } D_i \text{ such that } \sum D_i = D. \]

   For each \( D_i \), \text{InsertComponent}((T, D_i, D)). \]

Algorithm \text{InsertComponent}((T,D,D)). Insert a component \( D \) of the data object \( D \) into the cell tree with root node \( T \).

IC1. \[ \text{[Insert into subtree.]} \text{ If } T \text{ is not a leaf, check each entry } E_i(T) \text{ to determine whether } E_i(T).P \text{ overlaps } D. \text{ If yes, expand the container } E_i(T).C \text{ to include } D \cap E_i(T).P, \text{ and } \text{InsertComponent}((T', D, D \cap E_i(T).P)) \text{ where } T' \text{ is the node } E_i(T).cp \text{ points to.} \]

IC2. \[ \text{[Insert into leaf node.]} \text{ If } T \text{ is a leaf node, insert a new entry } E \text{ into } T \text{ where } E.D = D \text{ and } E.Z = D. \text{ If } T \text{ cannot be stored on one disk page anymore, } \text{SplitNode}((T)). \]

In our current implementation, this algorithm is also used for the initialization of the cell tree: the data objects in the database are inserted one by one using \text{InsertObject}.

5.2. Deletion

In order to delete a data object \( D \) from a cell tree that indexes the object, one also first computes a convex cover of \( D \). For each component \( D_i \) in the cover, a deletion step is performed. To avoid empty leaves, a tree condensation is performed where necessary (see section 5.4).

Algorithm \text{DeleteObject}((T,D)). Delete the data object \( D \) from the cell tree with root node \( T \).

DO1. \[ \text{[Decompose \( D \).]} \text{ If \( D \) is not convex, find a (small) set of convex polyhedra } D_i \text{ such that } \sum D_i = D. \]

   For each \( D_i \), \text{DeleteComponent}((T, D_i, D_i)). \]

DO2. \[ \text{[Condense tree.]} \text{ For each leaf node } N \text{ that is marked, } \text{CondenseTree}(N). \]

Algorithm \text{DeleteComponent}((T,D,D)). Delete the component \( D \) of the data object \( D \) from the cell tree with root node \( T \).

CD1. \[ \text{[Search subtree.]} \text{ If } T \text{ is not a leaf, check each entry } E_i(T) \text{ to determine whether } E_i(T).P \text{ overlaps } D. \text{ If yes, } \text{DeleteComponent}((T', D, D \cap E_i(T).P)) \text{ where } T' \text{ denotes the node } E_i(T).cp \text{ points to.} \]

CD2. \[ \text{[Update leaf node.]} \text{ If } T \text{ is a leaf node, check each entry } E_i(T) \text{ to determine whether } E_i(T).D = D. \text{ If yes, remove the entry } E_i(T) \text{ from } T. \text{ If } T \text{ is now empty, mark } T, \text{ otherwise contract the container } E_i(T).C \text{ if possible.} \]

5.3. Node Splitting

As mentioned, it is attempted to store each leaf on no more than one disk page. If a leaf requires more storage space, it is attempted to split the leaf (and the cells it contains) in such a way that both subleaves can be stored on one disk page each. This is not always possible. An efficient algorithm to obtain a suitable splitting hyperplane is based on the plane sweep paradigm [16]. One sweeps across the cells in the leaf node along I different directions, looking for a hyperplane which (a) intersects a minimum number of cells, and (b) keeps the storage requirements of the two subleaves as balanced as possible. If this algorithm leads to a suitable splitting hyperplane, the leaf is split. In this case, the ancestor node may have to
be split as well, and so on. If the split does not succeed, the leaf is stored using overflow pages. The parameter $I$ is to be varied as part of the performance tuning. A large $I$ will cause the splitting operation to be more costly, but it may yield a better hyperplane. We obtain the following algorithm.

**Algorithm SplitNode(LN).** Given an overloaded leaf node $LN$ in a cell tree, split $LN$ along a hyperplane, and propagate the split up the tree if necessary.

**SL1. [Find hyperplane.]** Sweep across $LN$ in $I$ different directions, looking for the best splitting hyperplane such that both subleaves can be stored on one disk page. If that does not succeed, stop. Otherwise let $H_1$ and $H_2$ denote the two disjoint halfspaces that are defined by the splitting hyperplane.

**SL2. [Grow tree taller.]** If $LN$ is the root, create a new root whose only entry is $(q_0, E, CP)$. Here, the container $CP$ is a convex polyhedron with at most $k$ faces that encloses all cells in the cell tree, and $q_0$ is a pointer to $LN$.

**SL3. [Create subleaves.]** $LN_1$ and $LN_2$ are empty leaves initially. For each entry $E_i$ of $LN$, test if $E_i \cap CP$ overlaps the halfspace $H_r$ ($r=1,2$). If yes, add a new entry $E$ to $LN_r$, where $E = E_i \cap (H_r \cap CP)$, and $q_r$ is a pointer to $LN_i$.

**SL4. [Create new entries.]** Let $q_1$ and $q_2$ be pointers to $LN_1$ and $LN_2$, respectively. Create two new entries $E_{LN,1} = (q_1, E_{LN,1}, H_1 \cap CP)$, and $E_{LN,2} = (q_2, E_{LN,2}, H_2 \cap CP)$.

**SL5. [Propagate split upwards.]** If $LN$'s ancestor node $N$ has now more than $M$ entries, SplitNode($N$).

Note that the splitting of a leaf may require the representations of some of the enclosed cells to be modified (step SL3). If the convex chains of the data objects in the database are stored explicitly, it is therefore necessary to modify these structures accordingly.

When splitting interior nodes, one needs to maintain the condition that the number of partitions on each side of the splitting hyperplane is at least $m$. Furthermore, the splitting hyperplane should intersect a minimal number of containers $E_i, C$ because each such intersection has to be propagated down the tree. A large number of such intersections may cause the split to become very costly.

Note that the hyperplane $H(N)$ corresponding to the root node of N's BSP-tree does not intersect any of the polyhedra $E_i, P$ (or $E_i, C$). Therefore a split using $H(N)$ would not have to be propagated down the tree. For that reason, in the cell tree an interior node will only be split if the hyperplane $H(N)$ is suitable for splitting, i.e. if the number of partitions on both sides of $H(N)$ is at least $m$. If that is not the case, $N$ will not be split. It will rather be stored using overflow pages. Whenever another entry is added to $N$, however, another attempt will be made to split $N$ along the hyperplane $H(N)$.

**Algorithm SplitLeaf(LN).** Given an overloaded interior node $N$ in a cell tree, split $N$ along a hyperplane, and propagate the split up the tree if necessary.

**SN1. [Find hyperplane.]** Check if the number of partitions on both sides of $H(N)$ is at least $m$. If no, stop. Otherwise, let $H_1$ and $H_2$ denote the two disjoint halfspaces that are defined by the hyperplane $H(N)$.

**SN2. [Grow tree taller.]** If $N$ is the root, create a new root whose only entry is $(q_0, E, CP)$. Here, the container $CP$ is a convex polyhedron with at most $k$ faces that encloses all cells in the cell tree, and $q_0$ is a pointer to $N$.

**SN3. [Create subnodes.]** $N_1$ and $N_2$ are empty nodes initially. For each entry $E_i$ of $N$, test if $E_i, P \subseteq H_r$ ($r=1,2$). If yes, add the entry $E_i$ to $N_r$.

**SN4. [Create new entries.]** Let $q_1$ and $q_2$ be pointers to $N_1$ and $N_2$, respectively. Create two new entries $E_{N,1} = (q_1, E_{N,1}, P \cap H_1 \cap CP)$, and $E_{N,2} = (q_2, E_{N,2}, P \cap H_2 \cap CP)$.

**SN5. [Propagate split upwards.]** If $N$'s ancestor node $A$ has now more than $M$ entries, SplitNode($A$).

Due to the fact that the polyhedra $E_i, P$ are partitions of a BSP, step SN3 can be carried out very efficiently as follows. The hyperplane $H(N)$ corresponds to the root of N's BSP tree. The two BSP subtrees below that root correspond to BSP's of the polyhedra $E_{N,1} \cap H_1$ and $E_{N,2} \cap H_2$. Each partition in those BSP's is also a partition of the original BSP of $E_{N,1}$. Therefore, all that has to be done is to copy these BSP subtrees into the corresponding subnodes $N_1$ and $N_2$.

The probability that $H(N)$ is not suitable for splitting can be computed as follows. As above, let $M$ denote the maximum number of entries that can be stored on one disk page. The total number of partitions in an overloaded node is at least $M+1$. Hence, after $H(N)$ was first established (viz., when $E_{N,1}$ was split for the first time), at least $M-1$ more partitions were formed by further splittings of $E_{N,1}$. Assuming that the cells in the subtree rooted at $N$ are distributed equally across the subspace $E_{N,1}$, the probability that the number of partitions on any side of $H(N)$ is less than $m$ is

$$P(N) = \left[ 1 + \frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{m-1} \right]$$

As this probability depends on $m$, it is important to keep $m$ reasonably low. For $m=3$ and $M=28$, for example, we obtain a probability of less than one in a million.

**5.4. Tree Condensation**

The tree condensation eliminates empty leaves and propagates the elimination up the tree. Interior nodes with less than $m$ entries are deleted and the entries under these nodes are reinserted into the cell tree. See [10] for a detailed description of this algorithm.
6. Conclusions

We presented the design of a dynamic object-oriented index structure for geometric databases, termed cell tree. The purpose of the cell tree is to facilitate search queries on large databases that contain extended multidimensional objects and are stored on secondary memory. We have implemented the cell tree and are currently comparing it to the R- and R+-tree. Although the cell tree often requires more storage space and more CPU time to answer a search query, it usually obtains the results with a lower number of disk accesses than these two rival structures.

References