DOI: 10.20378/irb-58919

Paging binary trees with external balancing *

Andreas Henrich
FernUniversität Hagen
5800 Hagen
West Germany

Hans-Werner Six
FernUniversität Hagen
5800 Hagen
West Germany

Peter Widmayer
Universität Freiburg
7800 Freiburg
West Germany

Abstract

We propose the partially paged binary tree principle (PPbin tree principle, for short) for maintaining binary trees which do not fit into core and hence must be (at least partially) paged on secondary storage. The PPbin tree principle can be applied to balanced as well as unbalanced binary trees. Paging a balanced binary tree results in a balanced external binary tree. However, main advantage of the new principle is that even for unbalanced binary trees it is very unlikely that long external access paths will arise. As an example, we describe the partially paged k-d tree which is used as directory in a spatial data structure. The analysis of the expected storage utilization and the expected external height proves the efficiency of the new data structure derived from the application of the PPbin tree principle.

1. Introduction

Binary search trees are useful and easy to handle data structures used in many applications. In the one-dimensional case, several balanced tree classes like AVL trees (see e.g. [AVL62], [Wir75]), brother trees [OW81] or weight-balanced trees [NR73] have been introduced. In the multidimensional case, the unbalanced k-d trees [Ben75] have been proposed.

Unfortunately, binary search trees in general suffer from the need to be kept in main memory. In this paper, we propose a new principle for maintaining binary trees which do not fit into main memory and hence must be paged on secondary storage. We call it the **Partially Paged binary tree principle** (PPbin tree principle, for short). Here, "partially paged" means that the binary tree can be maintained in core as usually as far as its size does not exceed the main memory capacity. If its size grows such that the whole tree cannot be kept in core any longer, a prefix tree whose size depends on the core size is maintained internally and only subtrees outside the prefix tree must be paged. The PPbin tree principle can be applied to many different binary tree classes. Main advantage of the PPbin tree principle is that even for unbalanced binary trees it is very unlikely that long external access paths will arise, i.e. many external pages must be traversed during a search. Hence, main application area of the PPbin tree principle are unbalanced binary trees occurring, for instance, in multidimensional geometric applications. As an example, we describe the partially paged k-d tree which is used as directory in a spatial data structure accommodating point- and non-point geometric objects. The analysis of the expected storage utilization and the expected external height proves the efficiency of the new data structure.

In [IKO87], binary priority search trees which support three sided range queries and updates with optimal worst case complexity are adapted to secondary memory. The resulting data structure is derived from B-trees [BM72] and from a generalized version of red-black trees [GS78] and therefore balanced. In contrast to our spatial data structure based on the application of the PPbin tree principle to k-d trees, however, the external priority search tree does not efficiently support regular (i.e. four sided) range queries and cannot be extended to more than two dimensions and to arbitrary (i.e. non-point) geometric objects.

Robinson [Rob81] proposes the K-D-B tree as an extension of the B-tree for storing k-dimensional points. However, the performance and storage utilization is outperformed by several other spatial data

* This work has been supported by DFG grants Si 374/1 and Wi 810/2.

structures like [NHS84], [KS88], [Free87], and our k-d PPbin tree based data structure. Furthermore, it is not suitable for non-point geometric objects.

Although unbalanced binary trees are the natural application area of the PPbin tree principle, it can also be carried over to balanced binary trees with advantage. Applying the PPbin tree principle to an AVL-tree, for example, results in a PPbin tree with logarithmic bound on the external height [HSW89].

In the next section, we explain the PPbin tree principle for one-dimensional unbalanced binary trees. In section 3, an application of the principle to the (unbalanced, multidimensional) k-d tree serving as directory in a spatial data structure is provided. In section 4, the expected storage utilization and the expected external height of the spatial data structure based on the k-d PPbin tree are analyzed.

2. The PPbin tree principle

2.1 Basic ideas and properties

We are facing the problem that the number of objects to be stored in a binary search tree exceeds the main memory capacity. Here, the whole tree or at least subtrees must be paged on secondary storage. This is not an easy task, especially if the pages should be reasonably filled and access paths should not contain too many pages in order to get the number of disk accesses for maintaining data objects small. We propose to apply the PPbin tree principle resulting in a *PPbin tree* which consists of two basic parts:

1. the internal (prefix-)tree T_i containing nodes near the root, and
2. several external subtrees stored in pages.

Furthermore, for a PPbin tree the *external balancing property* holds:

The number of external pages which are traversed on any two paths from the root to a leaf differs by at most 1.

Defining the *external height* $h_{ext}(T)$ of a PPbin tree T to be the maximal number of external pages occurring on a path in T, the external balancing property ensures that each path in T traverses either $h_{ext}(T)$ or $h_{ext}(T)$-1 external pages.

When objects are inserted into an initially empty PPbin tree, the tree grows up to a size when it cannot be kept in the dedicated part of the main memory any longer. Then a paging algorithm determines a subtree T_s to be paged on secondary storage such that the external balancing property is preserved. If T_s consists of n_s nodes, the main memory is now able to receive additional n_s nodes until a further invocation of the paging algorithm must take place.

If a paged subtree T_P grows up to a size where it cannot be kept in one page, the page is split into two. To this end, the left and right subtree of the root of T_P are stored in distinct pages. If the split page has been referenced by a node in the internal prefix tree T_i, the root of T_P is inserted into T_i. If the split page has been referenced by a node in a page, the root of T_P is inserted into this page.

Figure 2.1 shows a PPbin tree.

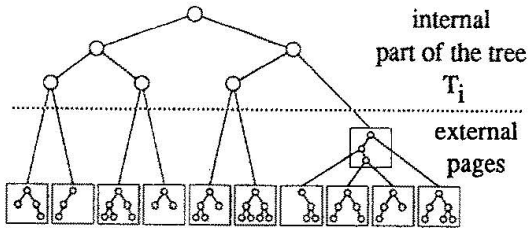


Figure 2.1: Overall structure of a PPbin tree

2.2 A closer look

We now discuss the PPbin tree in more detail by explaining the insertion of a new object into the structure. We formulate the procedures needed to perform an insertion operation in a pseudo programming language.

First, we explain procedure *Insert*, which inserts a new leaf q into a PPbin tree T.

```
PROCEDURE Insert (q, T);
{searches for the node p in the PPbin tree T which will be the father of the new leaf q and calls
TreeInsert, if T is not empty, and creates a new root for T, otherwise.}
BEGIN
    IF T is empty THEN
        let q be the root of T;
    ELSE
        search for the node p which will be the father of q;
        IF q will be the left son of p THEN dir := left ELSE dir := right END IF;
        TreeInsert(T, p, dir, q);
    END IF;
END Insert;
```

The procedure *TreeInsert* which we explain afterwards stores the new leaf q as the dir son of p (dir $\in$ {left, right}) in the PPbin tree T while preserving the external balancing property for T. For this purpose, *TreeInsert* calls the procedures *Page* and *PageSplit* which we describe next.

The procedure *Page* is called when after the insertion of an additional node the size of the internal prefix tree T_i reaches the maximal possible number n_i of internal nodes. Then *Page* searches for a subtree T_s in T_i such that paging T_s preserves the external balancing property and stores T_s in a page.

The external balancing property is preserved if T_s is a *paging candidate*, i.e. T_s fulfills the following conditions:

1. Any path from the root of T_s down to a leaf contains the minimal number of external pages (of all paths in T).
2. The height of T_s is at most h_P.

The second condition stems from the fact that a page can hold a tree up to a height of h_P.

In order to direct the search for a paging candidate in T_i the following numbers are attached to each node v in T_i:

$nep_{minl}(v)$, resp. $nep_{minr}(v)$,: the minimal number of external pages occurring on any path in T traversing the left, resp. right, son of v.

$nep_{maxl}(v)$, resp. $nep_{maxr}(v)$,: the maximal number of external pages occurring on any path in T traversing the left, resp. right, son of v.

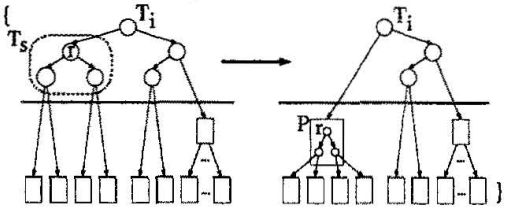
s(v): the number of nodes of the biggest paging candidate which can be reached from v.

h(v): The height of the subtree with root v in T_i.

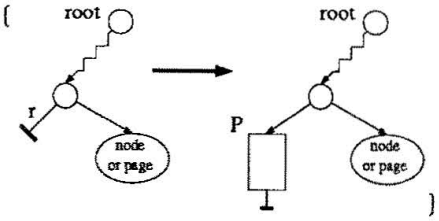
Now we can define procedure *Page*:

```
PROCEDURE Page (T);
{pages a subtree T_s of the internal prefix tree T_i.}
BEGIN
    r := root(T_i);
    WHILE r ≠ NIL AND r is a node in T_i AND
          (nep_minl(r) ≠ nep_maxl(r) OR nep_minr(r) ≠ nep_maxr(r) OR nep_minl(r) ≠ nep_minr(r) OR
           h(r) > h_P) DO
```

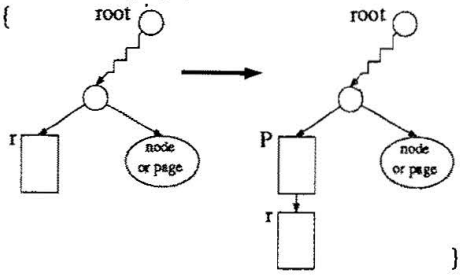
IF $nep_{minl}(r) < nep_{minr}(r)$ THEN
r := the left son of r;
ELSIF $nep_{minl}(r) > nep_{minr}(r)$ THEN
r := the right son of r;
ELSE
r := the son of r with greater s(son);
{if s(left son) = s(right son) each son might be chosen}
END IF;
END WHILE;
f := r;
allocate a new (empty) page P;
IF r is a node in T_i THEN
{



}
store the subtree T_s with root r of T_i in P;
make the pointer to r pointing to P;
ELSIF r = NIL THEN
{



}
replace r by the page P;
make P pointing to NIL;
Page(T);
ELSE {r is a page}
{



}
make the pointer to r pointing to P;
make P pointing to r
Page(T);
END IF;
REPEAT
f := the father of f in T_i;
recompute $nep_{minl}(f)$, $nep_{maxl}(f)$, $nep_{minr}(f)$, $nep_{maxr}(f)$, s(f) and h(f) from the corresponding values of the direct sons of f;
UNTIL f = root(T_i);
END **Page**;

If more than one paging candidate occurs in T_i, the paging algorithm chooses a candidate with the maximal possible number of nodes. If the search for a paging candidate ends with NIL or in a page, an empty page P is attached and *Page*(T) is invoked again. The recursive call of *Page* may happen $\lceil \frac{n_i}{2} \rceil$ times in the worst case. However, in our simulations, where 100,000 skew distributed objects have been inserted into an LSD tree (a spatial data structure with a PPbin tree as directory; see section 3) with bucket capacity 5, no empty page was created. The reason is that, if after the insertion of a new node the size of T_i reaches the main memory capacity n_i, *Page* searches for a paging candidate in T_i independently of the actual insertion which has caused the call of *Page*. Hence, a paging candidate is determined with one *Page* call if it exists at all. The performance complexity of one *Page* call is O(height(T_i)).

The procedure *PageSplit* is called when an additional node has to be stored in a page which is not able to take it without splitting. In a page a subtree is organized as a sequential heap of fixed height h_P. We choose the heap organization because the PPbin tree is "height-balanced" w.r.t external pages. Hence, it seems natural to use a height-balanced criterion inside a page, too: when the insertion of an additional node would cause the height of a paged subtree to exceed h_P, the page is split into two. After splitting the new node can be inserted.

PROCEDURE **PageSplit (p, dir, q, T_P, T);**
{The left and right subtree of the root of T_P are stored in two distinct pages P_l and P_u, and the root of T_P is inserted into T by calling *TreeInsert*. After the page split the new node q is the dir son of p (dir $\in$ {left, right}).}
BEGIN
  P := the page containing T_P;
  r := root(T_P);
  allocate two new pages P_l and P_u;
  store the left subtree of r in P_l and make P_l be the left son of r;
  store the right subtree of r in P_u and make P_u be the right son of r;
  IF p is stored in P_l THEN
    store q as the dir son of p in P_l;
  ELSE
    store q as the dir son of p in P_u;
  END IF;
  f := the node in T referencing P;
  IF P is the left son of f THEN dirf := left ELSE dirf := right END IF;
  deallocate P;
  TreeInsert(T, f, dirf, r);
END **PageSplit**;

Figure 2.2 depicts the effect of a page split.

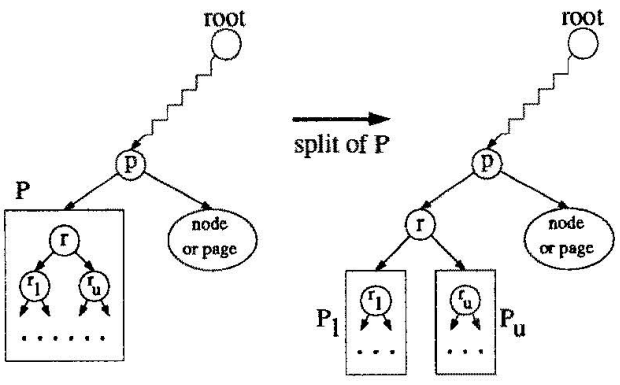


Figure 2.2: Split of page P

We are now in a position to define *TreeInsert*:

```
PROCEDURE TreeInsert (T, p, dir, q);
{stores the node q as the dir son of p (dir ∈ {left, right}) in the PPbin tree T. q is the new leaf
to be inserted or the root of a subtree whose page has been split.}
BEGIN
    IF p is a node in T_i THEN
        make q the dir son of p in T_i;
        f := q;
        REPEAT
            f := the node in T_i referencing f;
            recompute nep_minl(f), nep_maxl(f), nep_minr(f), nep_maxr(f), s(f) and h(f) from the corre-
                sponding values of the direct sons of f;
        UNTIL f = root(T_i);
        IF (number of nodes in T_i) = n_i THEN Page(T) END IF;
    ELSE {p is stored in a page}
        T_P := the paged subtree containing node p;
        IF the height of T_P will be greater than h_P after the insertion of p THEN
            PageSplit(p, dir, q, T_P, T);
        ELSE
            Store q as the dir son of p in T_P;
        END IF;
    END IF;
END TreeInsert;
```

Note that before and after the execution of *TreeInsert* the number of internal nodes is always smaller than n_i. Hence, T_i is always able to take a new node before an eventual call of the procedure *Page* takes place.

Figure 2.3 depicts the overall structure of the insertion algorithm for the PPbin tree.

The heart of the insertion algorithm for PPbin trees is the procedure *Page* which determines a paging candidate and stores it on secondary storage. Since *Page* always determines a paging candidate with the maximal number of nodes among all paging candidates, the page utilization is satisfactory and a degeneration of the external height is rather unlikely. Hence, a degeneration of the (original) binary tree is mainly reflected by a degeneration of the internal prefix tree and less by a degeneration of the external height of the PPbin tree.

The analysis of the application of the PPbin tree principle to arbitrary binary trees is extremely difficult and still open. For the class of k-d trees, the analysis provided in section 4 demonstrates that the expected external height of k-d PPbin trees is near the minimal possible external height.

2.3 The operations

Search

A search is performed similar to a search in unpaged binary trees.

Insertion

The insertion algorithm has been explained in detail in the previous section.

Deletion

In a PPbin tree T, a node is deleted similar to a node deletion in unpaged binary trees, i.e. we can restrict the discussion to the deletion of a node v with at most one son. If v is an internal node, i.e. in T_i, the deletion is trivial. If v is a node in a paged subtree T_P the height of T_P may decrease by one.

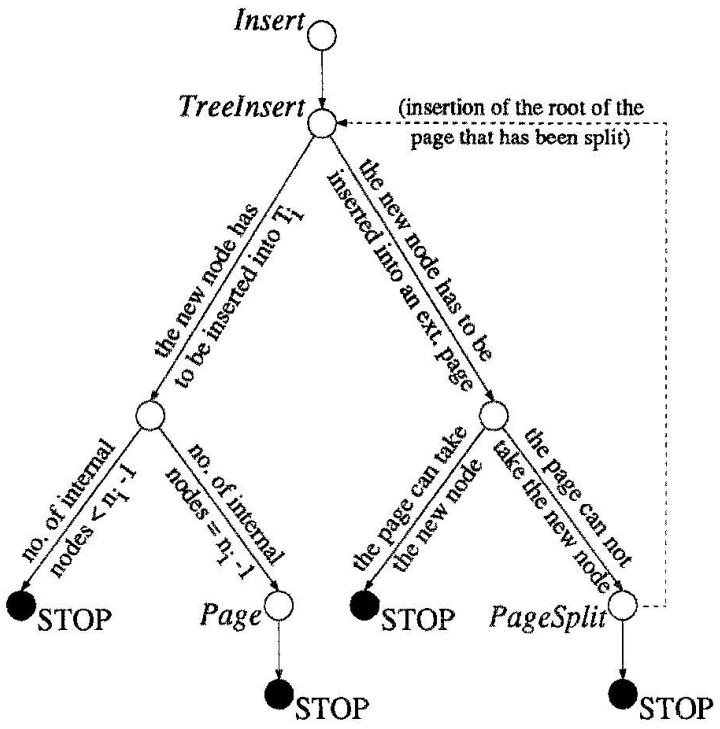


Figure 2.3: Overall structure of the insertion algorithm for the PPbin tree

If the page P storing T_P has a brother page which contains a subtree of height less than h_P, both pages can be merged. Such a merge works just inversely to a page split.

If the page P becomes empty, it can be removed from T if the external balancing property will not be violated. According to this property a path in T contains either $h_{ext}(T)$ or $h_{ext}(T)$-1 external pages. If paths of both external lengths exist, P can only be removed, if all paths containing P contain $h_{ext}(T)$ external pages. Otherwise, P has to remain in T. If all paths in T contain equally many external pages, P can also be removed.

If deletions cause the external height $h_{ext}(T)$ to decrease by one, empty pages, which could not be removed so far, can now be removed as far as the external balancing property is not violated. In any case, removing an empty page can be performed by the search procedure on the fly.

3. The LSD tree: an application of the PPbin tree principle in the multidimensional case

In this section, we present a new spatial data structure as an application of the PPbin tree principle in the multidimensional case. The data structure efficiently supports spatial queries like the retrieval of an object by its coordinates (exact match) and range queries where all objects geometrically intersecting the query region are selected for further processing or presentation on the screen. Since the set of objects varies over time, insertions and deletions of objects are supported as well.

The new structure using a partially paged k-d tree as directory has several advantages over other spatial data structures which have been proposed (see e.g. [Free87], [HSW88a], [HSW88b], [KS86], [KS88], [KW85], [NHS84], [Otoo86]):

1. It provides a great freedom for using the split strategy best suited for the actual application.
2. It is extremely insensitive to skew distributed objects.
3. The order of insertion does not influence the structure (as far as distribution dependent split strategies (see section 3.2) are applied).
4. For B data buckets the size (number of nodes) of the directory is B-1.
5. The structure can be extended to arbitrary geometric objects, i.e. non-point objects using the transformation technique ([Hin85], [SK88]). This technique, for instance, stores two-dimensional

rectangles as four-dimensional points (in a four-dimensional data structure). Unfortunately, the distribution of the four-dimensional points in general is extremely skew and therefore only data structures which are insensitive to such distributions perform well in this environment.

3.1 Basic ideas and properties

For sake of simplicity, we restrict the discussion to two-dimensional points. A generalization to arbitrary dimensions is straightforward.

Like most spatial data structures, the new structure partitions the data space into pairwise disjoint cells and stores all objects located in a cell in an associated data bucket (bucket, for short). In contrast to the grid file [NHS84], however, it is not grid oriented, i.e. a cell boundary may occur at an arbitrary position. The free choice of cell boundaries, i.e. split positions, is the basis of the graceful adaptation to arbitrary skew object distributions. Since a new split position can be chosen locally optimal, i.e. optimal with respect only to the cell to be split and independent from other existing cell boundaries, we call the structure **Local Split Decision tree** (LSD tree, for short).

Figure 3.1 shows a possible partition of the data space.

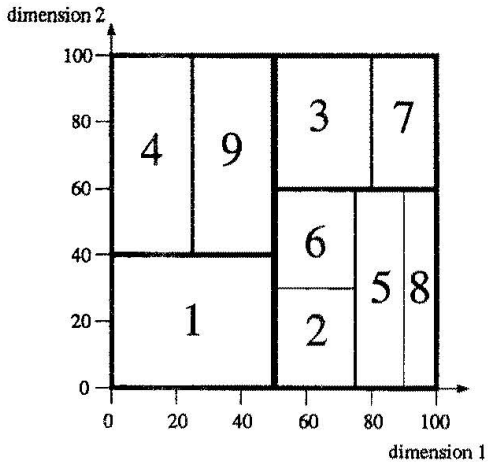


Figure 3.1: Possible partition of the data space for an LSD tree

This flexible data space partition has to be maintained by a directory. For this purpose we use a binary tree similar to a k-d tree [Ben75], which is embedded into a PPbin tree. Each node of the k-d PPbin tree represents one split decision by storing the split dimension and the split position (in that dimension). Figure 3.2 illustrates the LSD tree associated with the data space partition of Figure 3.1.

The first split is done in dimension 1 at position 50. All points whose first coordinates are smaller than, resp. greater than or equal to, 50 are stored in the left, resp. right, branch of the tree. Note that the tree levels are not necessarily alternatingly associated with data space dimensions.

Figure 3.3 shows the overall structure of the LSD tree with several external directory layers.

3.2 A closer look

In this section, we discuss the LSD tree in more detail by explaining the insertion of a new geometric object o into the structure. This insertion is performed by the procedure *LSDInsert*.

PROCEDURE **LSDInsert (o, T);**
{The search for the bucket B which will receive the new object o is guided by the directory T as in k-d trees. If B does not overflow, o is stored in B and the insertion is finished. Otherwise the procedure *BucketSplit* is called.}

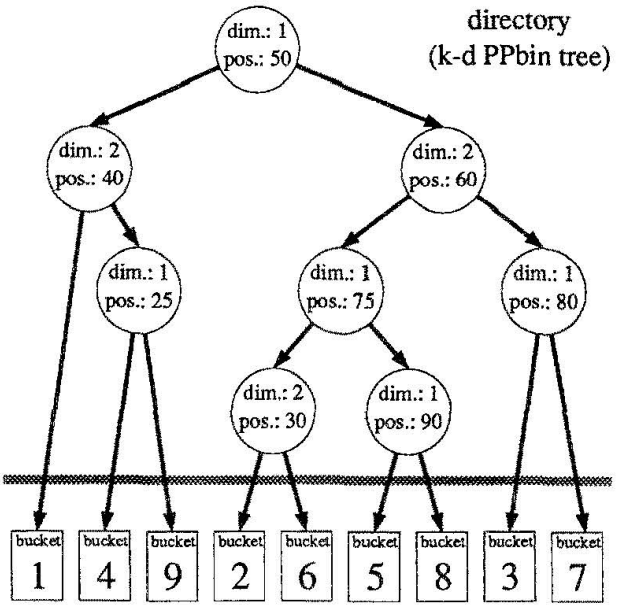


Figure 3.2: LSD tree associated with the data space partition of Figure 3.1

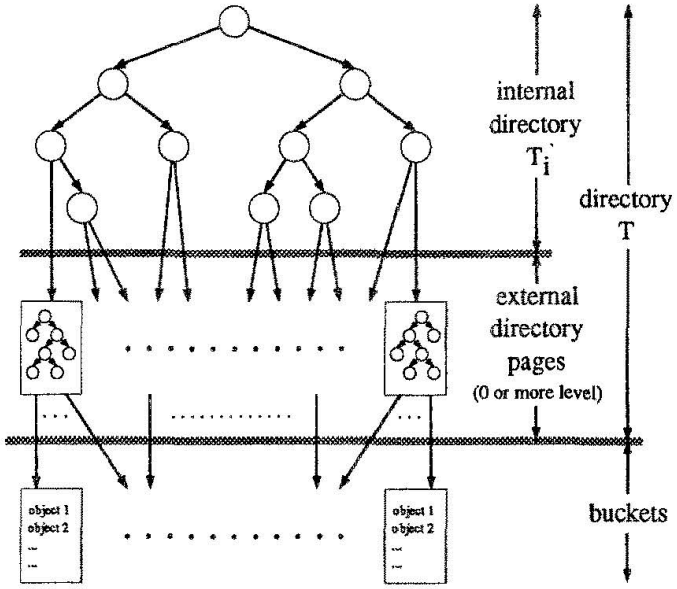


Figure 3.3: Overall structure of the LSD tree

```
BEGIN
    w := root(T);
    WHILE w is not a bucket DO
        s_dim := split dimension stored in w;
        s_pos := split position stored in w;
        IF o[s_dim] ≤ s_pos THEN w := left son of w ELSE w := right son of w END IF;
    END WHILE;
    B := w;
    IF B is not full THEN B := B ∪{o} ELSE BucketSplit(o, B, T) END IF;
END LSDInsert;
```

In *LSDInsert*, an overflowing bucket is handled by *BucketSplit*. The procedure *TreeInsert* which is used in *BucketSplit* is exactly the one described in section 2.2.

```
PROCEDURE BucketSplit (o, B, T);
{distributes object o and all objects in bucket B over two new buckets.}
BEGIN
    p := the node in T referencing B;
    IF B is the left son of p THEN dir := left ELSE dir := right END IF;
    dim := split dimension stored in p;
    ComputeSplitLine(o, B, dim, spos, sdim);
        {spos and sdim needed to split B are determined by ComputeSplitLine}
    allocate two new buckets Bl and Bu;
    create a new node q, containing spos and sdim, and referencing Bl and Bu;
    TreeInsert(T, p, dir, q);
    FOR EACH o' ∈ B ∪ {o} DO
        LSDInsert(o', T);
        {naturally, an efficient implementation will use locally available information and not
            carry out an LSDInsert for each o'}
    END FOR;
    deallocate B;
END BucketSplit;
```

Figure 3.4 depicts the effect of a bucket split.

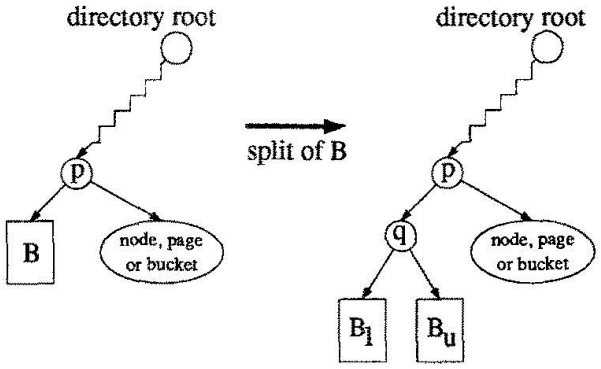


Figure 3.4: Split of data bucket B

It remains to define procedure *ComputeSplitLine*, which determines the split dimension s_{dim} and the split position s_{pos} for the overflowing bucket B. s_{dim} and s_{pos} depend on the split strategy used. We distinguish between two inherently distinct types of **split strategies**:

1. *Data dependent split strategies*
   These strategies depend only on the objects stored in the bucket to be split. A typical example is to choose for the split position the mean of all object coordinates with respect to a certain dimension.
2. *Distribution dependent split strategies*
   These strategies choose the split dimension and the split position independently of the actual objects stored in the bucket to be split (and of all other existing objects in the structure, as well). A typical example is to split a cell into two cells of equal areas. Note that this "halving split strategy" relies on the assumption of a uniform distribution of the objects.

```
PROCEDURE ComputeSplitLine (o, B, dim, spos, sdim);
BEGIN
    determine sdim and spos; {according to the split strategy used}
END ComputeSplitLine;
```

It should be obvious that the use of a binary tree as directory provides the freedom for using the split strategy best suitable for the actual application. This is an important advantage over other structures (see e.g. [Free87], [HSW88a], [KS86], [KS88], [NHS84]) where split decisions are more or less influenced by previous split decisions even if completely different data regions are concerned. Furthermore, the size of the directory is directly related to the number of buckets, i.e. for B buckets the directory contains B-1 nodes. This is in contrast to the grid file, for instance, where several entries in the directory may point to the same bucket.

3.3 The operations

Exact match

A search is guided by the k-d PPbin tree and ends in a bucket. This bucket is scanned until the object is determined or the search ends unsuccessfully.

Insertion

The insertion algorithm has been explained in detail in the previous section.

Deletion

An exact match locates the bucket B containing the object to be deleted and the object is removed from B. If the brother of B is a bucket, too, i.e. both buckets stem from the same bucket split, and the number of objects stored in both buckets is not greater than the bucket capacity, the two buckets are merged and the corresponding node is deleted from the directory as described for PPbin trees in Section 2.3.

Range query

In a range query all points located in a query region Q are reported. The function *RangeQuery* computes the set of requested objects. D(w) denotes the data region which is the union of all data cells whose corresponding buckets can be reached from the node w.

```
FUNCTION RangeQuery (Q, T) : SET OF objects;
{returns all objects which are located in the query region Q.}
BEGIN
    RangeQuery := ∅;
    stack := EmptyStack;
    w := root(T);
    PUSH(stack, w);
    WHILE NOT IsEmpty(stack) DO
        w := POP(stack);
        WHILE w is not a bucket DO
            IF Q ∩ D(right son of w) = ∅ THEN
                w := left son of w;
            ELSIF Q ∩ D(left son of w) = ∅ THEN
                w := right son of w;
            ELSE
                PUSH(stack, right son of w);
                w := left son of w;
            END IF;
        END WHILE;
        B := w;
        RangeQuery := RangeQuery ∪ {o | o is stored in B ∧ o is located in Q};
    END WHILE;
END RangeQuery;
```

Note that Q ∩ D(w) ≠ ∅ is the invariant condition of the inner WHILE-loop. Hence, if Q ∩ D(one son of w) = ∅ then Q ∩ D(other son of w) ≠ ∅.

4. Analysis of the LSD tree

In this section, we provide an analysis of the LSD tree. We estimate the expected utilization of data buckets and directory pages, the expected number of external directory layers, and the distribution of directory pages over the external layers. Since the directory of the LSD tree is a k-d PPbin tree, the analysis comprises the k-d PPbin tree, too. The analysis is based on the following assumptions:

1. The split decisions are independent of each other.
2. For each node v in T the probability that a point in D(v) is located to the left or to the right of the split line associated with v is $\frac{1}{2}$ each.

The first assumption is clearly ensured by the Local Split Decision tree. The second assumption is ensured if the split position is chosen as the mean of all object coordinates w.r.t. a certain dimension and the objects are inserted in random order. Hence, independent of the distribution of objects the assumptions are ensured in an LSD tree based on an appropriate data dependent split strategy, as far as the objects are inserted in random order.

The *level of a node* v in an LSD tree is as usual the length of the path from the root to v. The level of the root is 0. If v is a node in a page the level of v is defined as above assuming that a pointer to a page is a pointer to the root of the subtree stored in that page.

We regard the k-d PPbin tree used as directory T as an infinite binary tree with certain nodes active and others inactive. Of course, only the active nodes are actually stored in T. The number of nodes on level k is 2^k, $k \geq 0$.

The probability that a random point is located in the data region D(v) of a node v on level k in the directory T is $\frac{1}{2^k}$. If n points have been inserted into the LSD tree with directory T, the random variable X, denoting the number of points in the data region D(v) of a node v on level k is $B\left(n, \frac{1}{2^k}\right)$ distributed, i.e.

$$P(X=i) = \binom{n}{i} \cdot \left(\frac{1}{2^k}\right)^i \cdot \left(1-\frac{1}{2^k}\right)^{n-i} .$$

Since in all interesting cases $\frac{1}{2^k} < 0.1$ and $n > 30$ the binomial distribution can be approximated by a Poisson distribution:

$$P(X=i) \simeq \frac{\left(\frac{n}{2^k}\right)^i}{i!} \cdot e^{-\left(\frac{n}{2^k}\right)} .$$

The probability that a certain node v in the directory T of an LSD tree is active equals the probability that more than b (= bucket capacity) points are located in the data region D(v) because for storing more than b points a bucket split must have been performed activating v. Hence, the probability that a node v on level k in T is active is

$$\begin{aligned} P(X>b) &= 1 - P(X \leq b) \\ &\simeq 1 - \sum_{i=0}^{b} \frac{\left(\frac{n}{2^k}\right)^i}{i!} \cdot e^{-\left(\frac{n}{2^k}\right)} . \end{aligned}$$

Denoting

$$P(X \leq b) \simeq \xi_T(n,b,k) = \sum_{i=0}^{b} \frac{\left(\frac{n}{2^k}\right)^i}{i!} \cdot e^{-\left(\frac{n}{2^k}\right)}$$

the expected number of active nodes on level k in the directory T is

$$NLev_T(n,b,k) = (1 - \xi_T(n,b,k)) \cdot 2^k .$$

The expected number of active nodes in the directory T is

$$N_T(n,b) = \sum_{k=0}^{\infty} NLev_T(n,b,k)$$

and the expected number of buckets in the LSD tree is

$$B(n,b) = N_T(n,b) + 1 .$$

Figure 4.1 shows the expected bucket utilization

$$\beta(n,b) \stackrel{\text{def}}{=} \frac{n}{b \cdot B(n,b)}$$

for several bucket capacities.

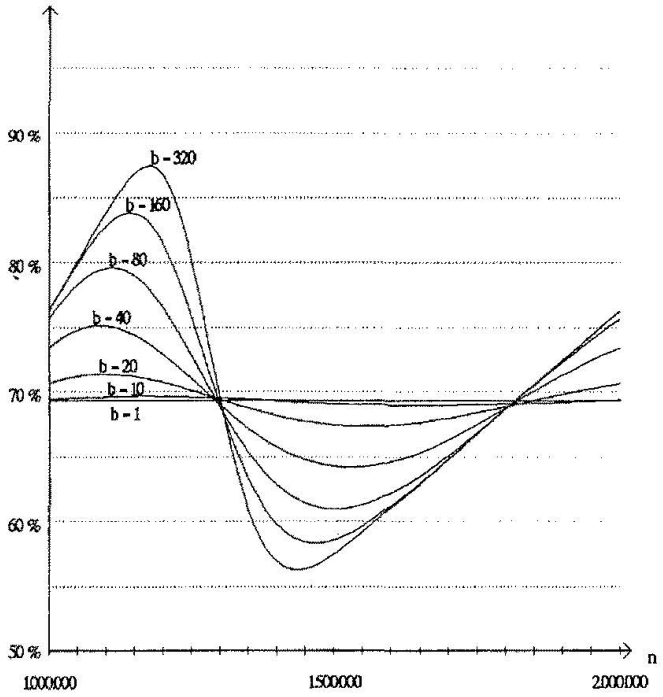


Figure 4.1: bucket utilization β(n,b)

Note that we assume that after a bucket split b instead of (the actual) b+1 objects are distributed over the two resulting buckets.

β(n,b) is a periodic function of n if b is fixed and $n \gg b$. Each doubling of n corresponds to one period.

Besides the bucket utilization we are interested in the directory page utilization. A directory page can accommodate a subtree up to a height of h_P. We start the analysis assuming that only one external layer is used.

A node v on level k in T_i is active (w.r.t. T_i) if and only if at least one node on level k+h_P in T which can be reached from v is active (w.r.t. T) (otherwise v is stored in a page). Hence, a node v on level k in T_i is inactive if and only if each of the 2^{h_P} nodes on level k+h_P in T is which can be reached from v is inactive. The probability that v is inactive is therefore

$$\xi_{T_i}(n,b,k,h_P) = [\xi_T(n,b,k+h_P)]^{2^{h_P}} .$$

Then the probability that v is active on level k in T_i is 1-ξ_{T_i}(n,b,k,h_P).

For T with one external directory layer the expected number of active nodes on level k in T_i is

$$NLev_{T_i}(n,b,k,h_P) = (1 - \xi_{T_i}(n,b,k,h_P)) \cdot 2^k ,$$

and the expected number of active nodes in T_i is

$$N_{T_i}(n,b,h_P) = \sum_{k=0}^{\infty} NLev_{T_i}(n,b,k,h_P) .$$

The expected number of directory pages in T with one external directory layer is

$$PLay_1(n,b,h_P) = N_{T_i}(n,b,h_P) + 1 .$$

Defining the expected directory page utilization as the quotient of the expected number of nodes stored in directory pages and the product of the expected number of directory pages and the page capacity, we get

$$\gamma_1(n,b,h_P) \stackrel{\text{def}}{=} \frac{N_T(n,b) - N_{T_i}(n,b,h_P)}{PLay_1(n,b,h_P) \cdot (2^{h_P} - 1)} .$$

Figure 4.2 shows the directory page utilization γ_1(n,b,h_P) for T with one external directory layer for several bucket capacities.

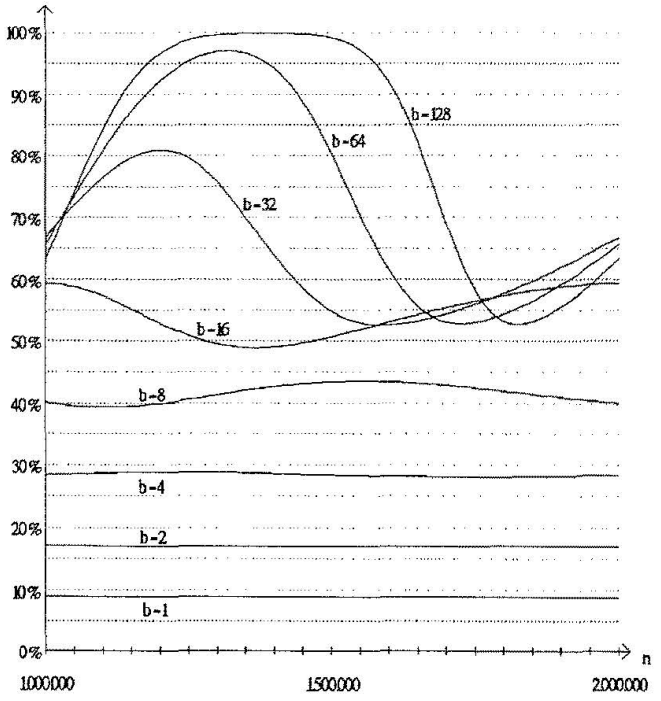


Figure 4.2: γ_1(n,b,h_P) for T with one external directory layer as a function of n (h_P = 6)

Like β(n,b), γ_1(n,b,h_P) is a periodic function of n if b is fixed and n $\gg$ b. Again, each doubling of n corresponds to one period.

We now extend the analysis of the directory page utilization if x $\geq$ 1 external directory layers are used. Note that layer 1 is the layer closest to the buckets and layer x contains the pages referenced from nodes in T_i.

Let T_i^x denote the internal part of the directory T when x external layers are used. A node v on level k in T_i^x is inactive (w.r.t T_i^x) if and only if each of the 2^{h_P} nodes on level k+h_P in T_i^{x-1} which

can be reached from v is inactive (w.r.t T_i^{x-1}). The probability that v (in T_i^x) is inactive is therefore given by the recurrence relation

$$\xi_{T_i^x}(n,b,k,h_P) = \begin{cases} [\xi_T(n,b,k+h_P)]^{2^{h_P}} & x=1 \\ \left[\xi_{T_i^{x-1}}(n,b,k+h_P,h_P)\right]^{2^{h_P}} & x>1 \end{cases}$$

The expected number of active nodes in T_i^x on level k is

$$NLev_{T_i^x}(n,b,k,h_P) = \left(1-\xi_{T_i^x}(n,b,k,h_P)\right)\cdot 2^k .$$

The expected number of active nodes in T_i^x is

$$N_{T_i^x}(n,b,h_P) = \sum_{k=0}^{\infty} NLev_{T_i^x}(n,b,k,h_P) .$$

Since for each external layer z $\in$ {1, ..., x}

$$N_{T_i^z}(n,b,h_P) = \sum_{k=0}^{\infty} NLev_{T_i^z}(n,b,k,h_P)$$

the expected number of directory pages on each external layer z is

$$PLay_z(n,b,h_P) = N_{T_i^z}(n,b,h_P)+1 .$$

The expected directory page utilization γ_z of each external layer z is given by

$$\gamma_z(n,b,h_P) \stackrel{\text{def}}{=} \begin{cases} \dfrac{N_T(n,b)-N_{T_i^z}(n,b,h_P)}{PLay_z(n,b,h_P)\cdot\left(2^{h_P}-1\right)} & z=1 \\ \dfrac{N_{T_i^{z-1}}(n,b,h_P)-N_{T_i^z}(n,b,h_P)}{PLay_z(n,b,h_P)\cdot\left(2^{h_P}-1\right)} & z>1 \end{cases}$$

Figure 4.3 shows the directory page utilization γ_z(n,b,hP) for z = 1, 2, 3, i.e. for layer 1, layer 2 and layer 3. The bucket capacity is 16.

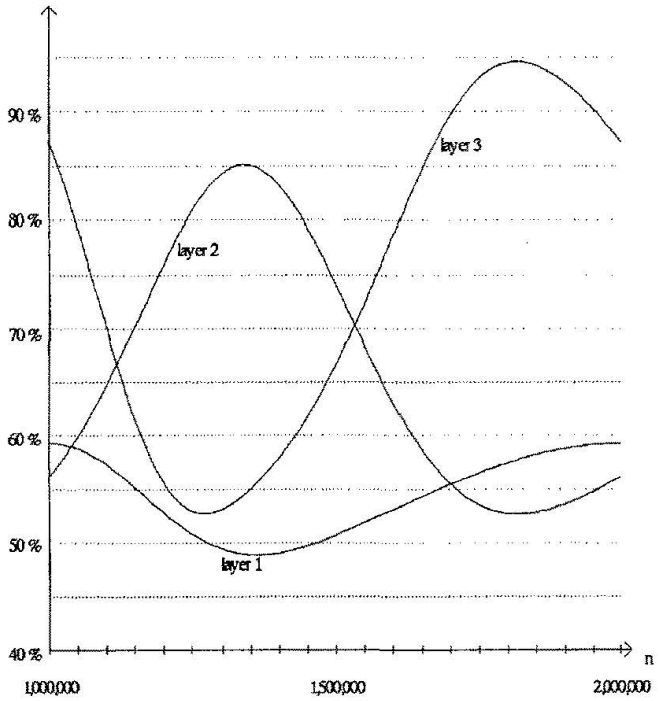


Figure 4.3: γ_z(n,b,hP) for z = 1, 2, 3, i.e. for layer 1, layer 2 and layer 3 (b = 16 and h_P = 6)

We are now in a position to derive the expected number of external layers and the expected number of directory pages if an upper bound n_i for the number of internal nodes is given.

The expected number χ of external layers can be calculated as follows:

1. If $N_T(n,b) \leq n_i$ then $\chi = 0$.
2. If $N_{T_i^1}(n,b,h_P) \leq n_i < N_T(n,b)$ then $\chi = 1$.
3. If $n_i < N_{T_i^1}(n,b,h_P)$ then choose χ such that $N_{T_i^\chi}(n,b,h_P) \leq n_i < N_{T_i^{\chi-1}}(n,b,h_P)$.

Note that χ is the expected value of the external height $h_{ext}(T)$ of T.

The expected number of directory pages on layer z is given by

$$PLay_z^*(n,b,h_P,n_i) = \begin{cases} \left\lceil \frac{N_{T_i^{z-1}}(n,b,h_P)-n_i}{\gamma_z(n,b,h_P)\cdot(2^{h_P}-1)} \right\rceil & z = \chi \\ PLay_z(n,b,h_P) & z < \chi \\ 0 & z > \chi \end{cases}$$

where $N_{T_i^0}(n,b,h_P) := N_T(n,b)$.

Table 4.1 shows, for instance, that on the average two external layers suffice for 10 million objects for realistic values of b, h_P and n_i.

n	pages on layer 1	pages on layer 2	pages on layer 3
10.000	0	0	0
100.000	238	0	0
500.000	1166	5	0
1.000.000	2332	38	0
5.000.000	13886	253	0
10.000.000	27771	526	0
20.000.000	55543	1070	3

Table 4.1: Expected number of directory pages needed on each layer if b = 16, h_P = 6 and n_i = 1000

At the end of this section, we sketch the analysis of the situation when the second assumption is relaxed. We assume that the probability that a point is located to the left, resp. to the right, of a split line is α, resp. 1-α, for $0 < \alpha < 1$.

For $\alpha > 0.5$ the probability that the leftmost path p_l in T (whose expected length is maximal among all paths in T) is shorter than k, i.e. the number of active nodes on p_l is less than k+1, is

$$\xi_{T,\alpha}(n,b,k) = \sum_{i=0}^{b} \frac{(n\cdot\alpha^k)^i}{i!} \cdot e^{-(n\cdot\alpha^k)} .$$

Hence, the probability that the length of p_l equals k is

$$\delta_\alpha(n,b,k) = \xi_{T,\alpha}(n,b,k+1) - \xi_{T,\alpha}(n,b,k) .$$

Figure 4.3 depicts δ_α as a function of k for multiple values of α. It turns out that the height of the directory of an LSD tree does not degenerate even if a clumsy split strategy is chosen.

The analysis demonstrates that under realistic assumptions the expected bucket utilization of the LSD tree as well as the expected page utilization of the k-d PPbin tree used as directory is competitive to other spatial data structures (see e.g. [Free87], [HSW88a], [KS86], [KS88], [KW85], [NHS84], [Otoo86]). Furthermore, the external height of the LSD tree does not degenerate even if a clumsy split strategy is used. Remember, however, that one of the main advantages of the LSD tree over other structures is the freedom of choosing the best suitable split strategy.

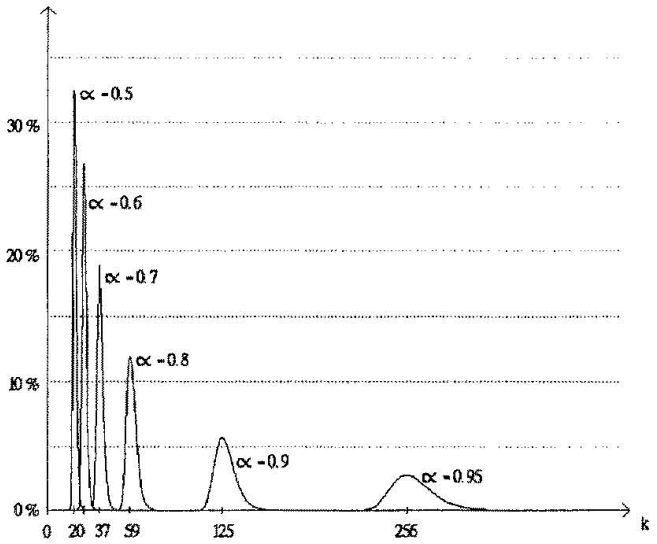


Figure 4.4: δ_α(n,b,k) as a function of k (b = 1 and n = 1,000,000)

References

[AVL62] Adelson-Velskii, G.M., Landis, E.M.: An algorithm for the organization of information', Soviet Math. Dokl. 3 (1962), 1259–1263

[BM72] Bayer, R., McCreight, E.: 'Organization and maintenance of large ordered indexes', Acta Informatika 1, 3, 173–189, 1972

[Ben75] Bentley, J.L.: 'Multidimensional Binary Search Trees Used in Database Applications', Communications of the ACM, Vol. 18, 9, 509-517, 1975

[Free87] Freeston, M.: 'The BANG file: a new kind of grid file', Proc. ACM SIGMOD Int. Conf. on Management of Data, 260-269, 1987

[GS78] Guibas, L.J., Sedgewick, R.: 'A Dichromatic Framework for Balanced trees', 19th Annual IEEE Symposium on Foundations of Computer Science, 8–21, 1978

[Hin85] Hinrichs, K.: 'The Grid File System: Implementation and Case Studies of Applications', Doctoral Thesis No. 7734, ETH Zürich, 1985

[HSW88a] Hutflez, A., Six, H.-W., Widmayer, P.: 'Globally Order Preserving Multidimensional Linear Hashing', Proc. IEEE 4th Int. Conf. on Data Engineering, 572-579, 1988

[HSW88b] Hutflez, A., Six, H.-W., Widmayer, P.: 'Twin Grid Files: Space Optimizing Access Schemes', Proc. ACM SIGMOD Int. Conf. on Management of Data, 183-190, 1988

[HSW89] Henrich, A., Six, H.-W., Widmayer, P.: unpublished manuscript

[IKO87] Icking, Ch., Klein, R., Ottmann, Th.: 'Priority Search Trees in Secondary Memory', Universität Freiburg Institut für Informatik, Bericht 4, November 1987

[KS86] Kriegel, H.-P., Seeger, B.: 'Multidimensional Order Preserving Linear Hashing with Partial Expansions', Proc. Int. Conf. on Database Theory, 203-220, 1986

[KS88] Kriegel, H.-P., Seeger, B.: 'PLOP-Hashing: A Grid File without Directory', Proc. IEEE 4th Int. Conf. on Data Engineering, 369-376, 1988

[KW85] Krishnamurthy, R., Whang, K.-Y.: 'Multilevel Grid Files', IBM Research Report, Yorktown Heights, 1985

[NHS84] Nievergelt, J., Hinterberger, H., Sevcik, K.C.: 'The Grid File: An Adaptable Symmetric Multikey File Structure', ACM Transactions on Database Systems, Vol. 9, 1, 38-71, 1984

[NR73] Nievergelt, J., Reingold, E.M.: 'Binary Search Trees of Bounded Balance', SIAM J. Computing 2 (1973) 33-43

[Otoo86] Otoo, E.J.: 'Balanced Multidimensional Extendible Hash Tree', Proc. 5th ACM SIGACT / SIGMOD Symposium on Principles of Database Systems, 100-113, 1986

[OW81] Ottmann, Th., Wood, D.: '1-2 brother trees or AVL trees revisited', Comput. J., 23 (1981), 248-255

[Rob81] Robinson, J.T.: 'The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes', Proc. ACM SIGMOD Int. Conf. on Management of Data, 10-18, 1981

[SK88] Seeger, B., Kriegel, H.-P.: 'Techniques for Design and Implementation of Efficient Spatial Access Methods', Proc. 14th Int. Conf. on VLDB, 360–371, 1988

[Wir75] Wirth, N.: 'Algorithmen und Datenstrukturen', B.G. Teubner, Stuttgart 1975