Priority Search Trees in Secondary Memory (Extended Abstract)

Ch. Icking, R. Klein, Th. Ottmann
Institut für Informatik
Universität Freiburg
Rheinstr. 10-12
D–7800 Freiburg, West–Germany

Abstract

In this paper we investigate how priority search trees can be adapted to secondary memory. We give an optimal solution for the static case, where the set of points to be stored is fixed. For the dynamic case we present data structures derived from B–trees and from a generalized version of red–black trees. The latter are interesting in the internal case, too, since they are better balanced than standard red–black trees, in that the ratio longest path/shortest path is smaller.

Keywords: search trees, B–tree, red–black tree, priority search tree, secondary memory.

1 Introduction

One of the most important problems in manipulating multi–dimensional data objects is how to store a set of n points in such a way that range queries and insertions/deletions can be performed efficiently. For 2–dimensional points the *priority search tree* [Mc85] has been shown to be an optimal data structure for both *three sided range queries* (see Figure 1a) and *fixed height window queries* [KNOW86] (see Figure 1b).

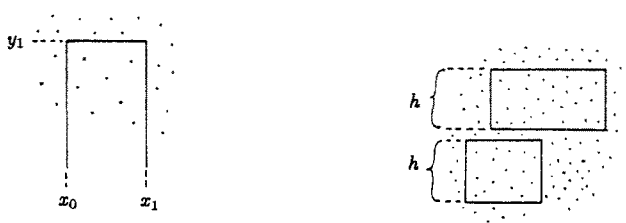

Figure 1: a) three sided range query b) fixed height window queries

If a priority search tree is implemented in internal memory, these queries can be performed within $O(\log n + k)$ time, where n is the number of points to be stored and k is the size of the answer. Insertions and deletions are possible in time $O(\log n)$. This is worst–case optimal.

Even for isothetic range queries in an arbitrary dimension d the priority search tree can help to improve the range query complexity of the range tree to $O(\log^{d-1} n)$ using $O(n \log^{d-1} n)$ storage, but this structure is not dynamic [E87].

In many applications (VLSI–Design, geographical data) the number of objects is so large that they don't fit into core. Therefore, data structures for *external memory* (hard disc) are required. In our model of complexity for secondary memory we only count the number of blocks read/written from/into external memory; main memory activity is not taken into account. Suppose that each secondary memory access transmits B units (bytes, words, points ...) of data, we can put the question like this: Is there a data structure *"External Priority Search Tree"* with $O(\log_B n + \frac{k}{B})$ half range query and $O(\log_B n)$ update complexity which doesn't use more than $O(\frac{n}{B})$ data pages? This would be obviously optimal, since that many disc accesses are necessary. Note that B must be treated here as a variable, not as a constant.

In this paper we are interested in the worst case complexity of our data structures, an important criterion for real time applications. For such purposes, other external data structures like the well–known *grid file* [NHS84], for example, do not provide us with any acceptable worst–case bound.

2 Internal Priority Search Trees

First let us consider the priority search tree in internal memory. It is a blend of leaf search trees and heaps, more formally:

Definition 1 An internal *priority search tree* is a binary tree storing a set of points (x, y) in the plane. It is a leaf search tree for the x–values of the points stored, that means for every x–value there exists one leaf node in the tree. Every node contains as a split value the maximum x–value of its left subtree, and space to store a point (possibly empty). The points are stored according to three conditions:

i) Each point (x, y) lies on the root–to–leaf path to x.

ii) The y–values of the points stored along an arbitrary root–to–leaf path are in increasing order.

iii) If a node holds a point then its father does, too.

See an example in Figure 2.

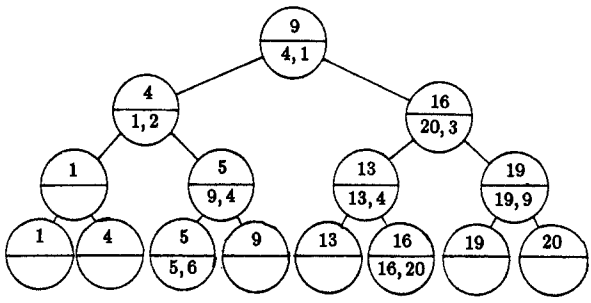

Figure 2: priority search tree

It is clear that such a priority search tree uses $O(n)$ space to store n points.

The operations on this data structure are:

a) Insert (delete) a point.

b) Given three numbers x_0, x_1 and y_1, report all points stored such that $x_0 \leq x \leq x_1$, $y \leq y_1$ (three sided range query, see Figure 1a).

The algorithms for these operations and their complexities depend on the kind of underlying search tree used. The *insert* algorithm always first inserts the x–value of the point and performs the necessary balancing operations, then "trickles down" the point into the tree, that means compares the y–values of the new point and the one in the root, stores the one with the lower y–value and continues this operation with the other point, exploring the root of the left or right subtree, according to the point's x–value, until an empty node is reached. The *delete* algorithm first deletes the point and "bubbles up" points from the subtree to fill the gap (analogous to "trickle down"), then deletes the x–value of the point and performs balancing operations, if necessary.

Most binary tree schemes use *rotations* to keep the tree balanced. It is easy to see that each rotation causes one "trickle down" and one "bubble up" in a priority search tree, so we are interested in keeping the number of rotations small.

There are several binary tree schemes that require $O(\log n)$ update time with only $O(1)$ rotations (at most 3) per update, namely *half balanced trees* [O82] and *red–black trees* [GS78]. Only the half balanced trees were designed to have this property, for red–black trees this fact was discovered later [T83], a result which [Me84a] [Me84b] seems to ignore. The algorithms differ, but the structure of these two tree schemes is equal to the structure of *symmetric binary B–trees* [B72]. Later in this paper we will use the representation of symmetric binary B–trees for red–black trees, that means red edges are drawn horizontally and black ones vertically.

Because the algorithms are easier and the balance information is smaller (just one bit per node), we prefer to use red–black trees to implement our priority search trees. In any case, using a tree with $O(1)$ rotations per update yields an optimal $O(\log n)$ complexity for updating the priority search tree.

The *range query* complexity for binary priority search trees is always $O(\log n + k)$, where k is the number of points reported, but keep in mind that we always "overshoot the mark", that means if a node's point lies in the query range, we have to examine the node's sons, even if the points stored there don't lie in the range any longer. If the tree is not binary, then this phenomenon will become very important for the range query complexity.

In the next two sections we will investigate how priority search trees can be implemented in external memory. We present two different approaches, first a modified B–tree scheme, and second a scheme using the balancing techniques of internal binary trees. In both cases the problem is how to guarantee a good range query complexity and, at the same time, an acceptable update performance.

3 Priority Search B–Trees

At first glance, a natural way of implementing priority search trees in external memory seems to be using the B–tree scheme [BM72]. Assume that to each node of the B–tree one data bucket is associated storing $\frac{B}{2}$ points and up to $\frac{B}{2}$ references (=pointer + split value) to descendant nodes, as shown in Figure 3.

Assume that, after a deletion of a point and of its corresponding x–value, two brother nodes at height i must be merged, because their total number of pointers has become less than $\frac{B}{2}$. Then, besides other difficulties, $\frac{B}{2}$ points must be trickled down the subtree, in the worst case to the leaf level, resulting in $O(Bi)$ disk accesses. Since merge operations may be necessary along a whole path in a B–tree, a single deletion would cause $O(BH^2)$ operations, where H is the height of the tree. This seems unacceptable.

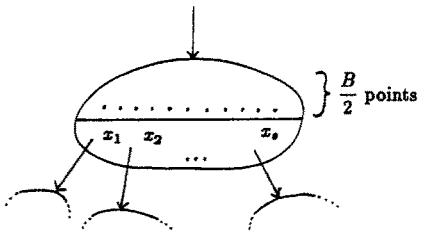

Figure 3: priority search B–tree

To overcome this deficiency, we associate with each reference to a descendant a fixed number, d, of the points stored in a node.

Definition 2 The following implementation of a priority search tree is called a *drawer–tree of capacity d.*

1. All leaf pages appear at the same level.
2. Each leaf page contains l x–values and l drawers of capacity d, where $\frac{B}{4} \leq l \leq \frac{B}{2}$.
3. Each node page contains k references to descendants and k drawers of capacity d. Here, $\frac{B}{2(d+1)} \leq k \leq \frac{B}{d+1}$, if the page is not the root, and $2 \leq k \leq \frac{B}{d+1}$, otherwise.
4. The x–value of a point stored in a drawer is contained in the subtree referred to by the associated pointer. Its y–value is not less than the y-values of all points stored in the parent drawer.
5. If a drawer is not empty then its parent drawer is full.

Figure 4 shows a fragment of a drawer–tree of capacity 2.

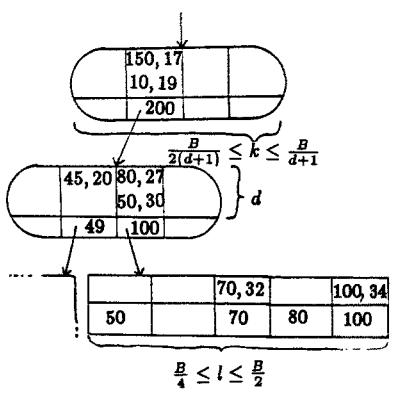

Figure 4: drawer–tree of capacity 2

If, during the insertion of the x–value of a new point, the number l (or k) of x–values (or drawer–reference units) stored in a page becomes too big, this page must be split into two pages, each of them containing about the same number of drawers. When inserting the additional reference hereby created into the parent node, the parent drawer is to be split. At most d points must be bubbled up in order to fill the two new drawers. If the number of drawers of the parent node has become too large by the last drawer split, the process continues.

Deletion of points is similar. First, the point itself is removed, and then its x–value. This may cause the contents of two pages to be redistributed among these pages, or to be merged into one page, as in the B–tree scheme.

Theorem 1 *A drawer tree of capacity d storing n points is of height*

$$H_d \leq \frac{1}{1 - \log_B 2(d+1)} \log_B \frac{4n}{B} .$$

Insertions and deletions can be performed within $O(H_d^2 d)$ steps, whereas a three–sided range query needs at most $O(H_d + \frac{k}{d})$ I/O operations. The space consumption is $O(\frac{n}{B})$.

Note that the height, H_d, is a strictly increasing function of d. By varying d, we can trade query performance for update performance, whatever seems more important. If $d = \sqrt{B}$, for example, we obtain an $O(H^2\sqrt{B})$ update time bound and an $O(H + \frac{k}{\sqrt{B}})$ query time bound, where $H \leq 2.9 \log_B n$ if $B \geq 100$.

4 Implementation Using Binary Tree Schemes

Another approach is to take the binary priority search tree as defined for the internal case, and to map several nodes onto one bucket in a suitable way. This can be done as shown in Figure 5; here portions of a fixed height h are stored in each bucket. Clearly, each bucket holds $2^h - 1$ points and has 2^h sons, and we have $H = \frac{\log_2 n}{h}$.

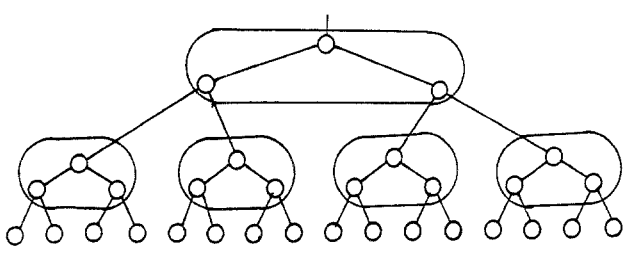

Figure 5: binary tree mapped onto buckets

Assume that we want to report all points in the half range $-\infty < x < \infty$, $y \leq y_1$, and that the points in the tree are distributed as shown in Figure 6.

Each time we report one of the points that have a maximum y–value $y \leq y_1$, we have to access the two son buckets, resulting in a number of disk accesses equal to the number k of points reported, so the range query complexity here can be as bad as $O(H+k)$. This is unacceptable because for each

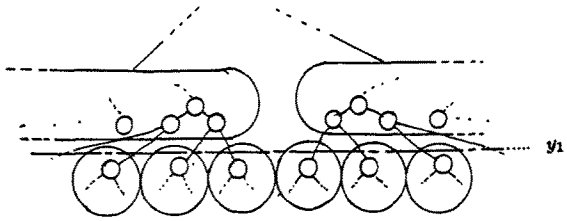

Figure 6: bad range query behavior

point reported we would have to access one bucket. This phenomenon is caused by the "overshooting the mark"-property of priority search trees mentioned in section 2.

But there is a way of getting around this difficulty. Instead of storing one point at each node of the underlying binary tree, we store a pointer to a data bucket full of points (Figure 7).

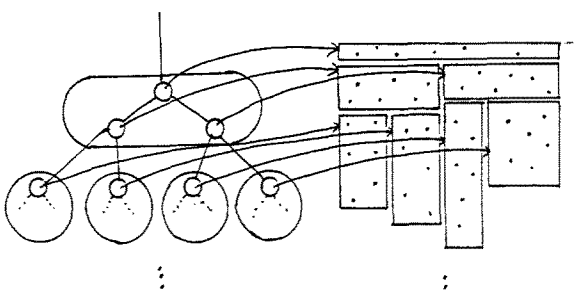

Figure 7: optimal static external priority search tree

Still the three priority search tree conditions are required to hold in an analogous way. Now for a range query we obtain the interesting property that for every bucket read the father's bucket must be fully contained in the query region, exept for the side bounderies. So our range search complexity becomes optimal $O(H + \frac{k}{B})$. Unfortunately, now we have lost any chance of performing efficient balancing operations, so this data structure, offering optimal query complexity, can only be used for the static case, where the set of points to be stored is fixed.

Theorem 2 *Static external priority search trees can be implemented using optimal $O(\frac{n}{B})$ data buckets such that three sided range queries can be performed with optimal $O(\log_B n + \frac{k}{B})$ disc accesses.*

In the rest of this section we will develop a way of dynamizing this external priority search tree without loosing too much of the optimal range query complexity.

Almost all of the various binary tree schemes known are using *rotations* for restructuring. Usually, rotations cause whole subtrees to be shifted upward or downward. In the internal case, this is easily

achieved by just redefining some pointers. But here we cannot afford moving subtrees like in Figure 5 across several data pages, for this would result in accessing all the data pages this subtree is stored in.

However, some binary tree schemes don't use rotations for rebalancing. Among them, there are the well–known 1–2 brother trees [OS76]. Here the restructuring algorithms resemble those of B–trees. Nevertheless, each of the $O(\log n)$ balancing operations that may be necessary during one insertion or deletion causes another $O(\log n)$ steps for the "trickle down"-"bubble up" operations (see section 2), resulting in a nonoptimal $O(\log^2 n)$ time complexity.

For this reason we resort to the red–black scheme that has been proven optimal in the internal case. Remember that the black depth of a node is not affected by rotations.

Therefore, the red–black tree should be mapped onto external memory in such a way that each data page contains a (sub)tree of a fixed black height h (Figure 8, remember that red edges are drawn horizontally, black ones vertically).

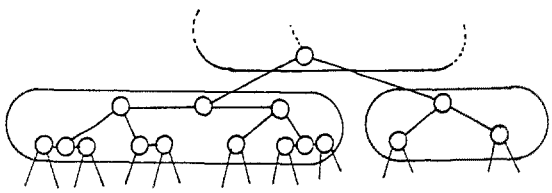

Figure 8: red–black tree mapped onto buckets

However, using the standard red–black tree scheme would result in a storage complexity of $O(\frac{n}{\sqrt{B}})$ because we have $B = 2^h$; for, every second edge can be red, but in the worst case only $\sqrt{B}$ of a data page of size B may be used (see the right subtree in Figure 8).

Definition 3 A *red–h–black tree* is a binary tree whose edges are coloured red or black such that

1. all leaves are of the same black depth
2. every leaf–root oriented path is coloured according to the scheme

$$[[\text{red}]\ \underbrace{\text{black}\ldots\text{black}}_{\text{exactly } h \text{ times}}\ ]^*\ [\text{red}]\ \underbrace{\text{black}\ldots\text{black}}_{\leq h \text{ times}}$$

See the example depicted in Figure 9. The encircled subtrees shown there are called *h–components* and will be stored on one page each.

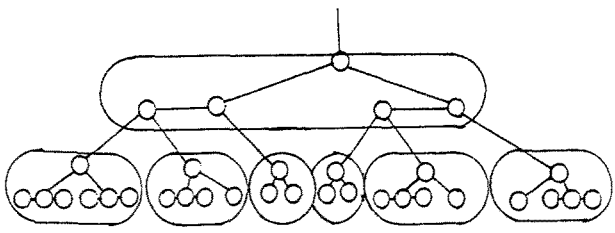

Figure 9: red–2–black tree

We briefly describe how to insert and to delete keys.

If a key is to be inserted into a component which is not already full (that is, it hasn't yet the maximum number of red edges), then the key is inserted and the component is *restructured* to maintain the defining properties. This doesn't affect the ancestors (Figure 10).

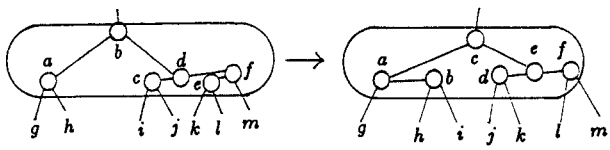

Figure 10: restructuring a red–2–black component

If the subtree stored in this component already has the maximum number of red edges then a *colour–flip* is performed, that means the component is split into two, and the former root is inserted into the fathers component (Figure 11).

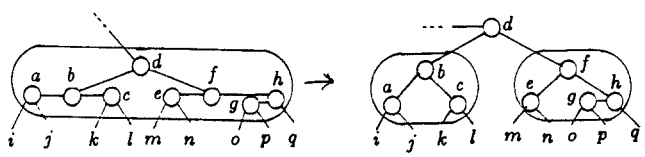

Figure 11: colour–flip in a red–2–black tree

This transformation can be visualized as pulling upward the component's subtree assuming that the black links have rigid joints, and the red links have movable ones. The root's ingoing black edge becomes red and all red edges become black .

Deletions are performed analogously, exept that two brother components may have to be rearranged simultaneously, if one of them has become underfull and the other one contains more than the minimum possible number of nodes, as in the B–tree scheme.

This results in an update complexity of $O(\log n)$. Note that after a rearrangement of one (of two) components, the algorithms stop. Thus, at most one rearrangement per update is needed. Furthermore, the structure of the binary tree is *not* affected by a colour–flip operation. This will become important for the implementation of priority search trees.

Proposition 1 *An internal red–h–black tree storing n keys has the following properties:*

i) The ratio of the lengths of a longest path and a shortest path is less or equal than $\frac{h+1}{h}$

ii) T can be updated within time $O(\log n)$*, requiring at most* $O(h)$ *structural changes.*

Now we define how priority search trees can be based on red–h–black trees. Rather than associating one point to each node, which would inevitably lead to the difficulty explained at the beginning of this section, we associate d (≥ 1) points to each node.

Here, d, h and B are related by the formula $(d+1)2^{h+1} = B$, because each bucket must be capable of storing 2^{h+1} nodes containing d points each, and 2^{h+1} pointers.

Theorem 3 *Let T be an external priority search tree based on the red–h–black tree scheme. Then insertion/deletion need $O(H_d B)$ many I/O operations in the worst case, and a three–sided range query can be performed with at most $O(H_d + \frac{k}{d})$ operations.*

Here, $H_d \leq \frac{1}{1-\log_B 2(d+1)} \log_B \frac{4n}{B}$.

The update complexity is due to the fact that a rearrangement of a component may require trickling down as many as $O(B)$ points. The range query complexity follows because for each bucket read all of the d points stored in the father node of the father component belong to the answer.

5 Concluding Remarks

We have presented two ways of implementing priority search trees in external memory. Both of them allow to trade query performance for update performance by choosing a certain parameter d. Both data structures have approximately the same query performance $O(H_d + \frac{k}{d})$ and the same storage requirements $O(\frac{n}{B})$. The red–h–black tree scheme yields an update performance of $O(H_d B)$ whereas the drawer tree scheme yields $O(H_d^2 d)$. Which of them suits a given application better depends highly on the number n of points and on the bucket size B.

In both cases H_d denotes the height of the tree depending on n, B and d:

$$H_d \leq \frac{1}{1 - \log_B 2(d+1)} \log_B \frac{4n}{B} .$$

Clearly these results should be compared with the performance of other external data structures like the grid file.

If the number of points, n, is really large then the 2–level grid file as described in [NHS84] and [H85] is not sufficient. Instead, at least $\log_B \frac{n}{B}$ levels are necessary, in general even more because the space consumption is not guaranteed to be $O(\frac{n}{B})$.

An insertion or deletion of a point can be performed in $O(\log_B \frac{n}{B})$ steps if no split or merge operations occur. The same holds for the priority search trees presented in this paper.

In a *good case*, a range query performed in the grid file needs $O(\log_B \frac{n}{B} + \frac{k}{B})$ operations, as compared with $O(H_d + \frac{k}{d})$ for our priority search schemes.

In the *worst case*, however, we can garantee a behavior that is still acceptable, whereas the grid file performance may collapse. Indeed, a range query may require to access all data pages without retrieving a single point. Even worse, the number of split operations caused by a single insertion can not be bounded in terms of n, if buckets are always split in the middle of their x– or y–intervals, as described in [H85].

Although the data structures presented here do better than this, it would be interesting to have a tight lower bound for external three–sided or standard range queries in a dynamic environment, and to find a worst case optimal data structure.

References

[B72] R. Bayer, Symmetric Binary B-trees: Data Structure and Maintenance Algorithms, Acta Informatica, 1 (1972), pp. 290-306.

[BM72] R. Bayer, E. McCreight, Organization of Large Ordered Indexes, Acta Informatica, 1 (1972), pp. 173-189.

[E87] H. Edelsbrunner, Geometrics and Algorithmics - A Tutorial in Computational Geometry, Bulletin of the EATCS, 32 (1987), pp. 118-142.

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

[H85] K. Hinrichs, The Grid File System: Implementation and Case Studies of Applications, Dissertation at the Swiss Federal Institute of Technology Zürich, ETH Zürich, 1985.

[KNOW86] R. Klein, O. Nurmi, Th. Ottmann, D. Wood, Optimal Dynamic Solutions for Fixed Windowing Problems, Proceedings of the 2nd Annual Symposium on Computational Geometry, 1986, pp. 109-115, (to appear in Algorithmica).

[Mc85] E. M. McCreight, Priority Search Trees, SIAM J. Comput., 14 (1985), pp. 257-276.

[Me84a] K. Mehlhorn, Data Structures and Algorithms 1: Sorting and Searching, EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1984.

[Me84b] K. Mehlhorn, Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry, EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1984.

[NHS84] J. Nievergelt, H. Hinterberger, K. C. Sevcik, The Grid File: An Adaptable Symmetric Multikey File Structure, ACM Transactions on Data Base Systems, 9(1) (1984), pp. 38–71.

[O82] H. J. Olivié, A New Class of Balanced Trees: Half Balanced Binary Search Trees, RAIRO Informatique Théorique, 16 (1982), pp. 51-71.

[OS76] Th. Ottmann, H.-W. Six, Eine neue Klasse von ausgeglichenen Binärbäumen, Angewandte Informatik, 9 (1976), pp. 395-400.

[T83] R. E. Tarjan, Updating a Balanced Search Tree in O(1) Rotations, Information Processing Letters, 16 (1983), pp. 253-257.