Kurt Mehlhorn

Data Structures and Algorithms 1: Sorting and Searching

With 87 Figures

Pages 199 - 206 + 212 (213)

Springer-Verlag
Berlin Heidelberg New York Tokyo 1984
on such that $BO_i(v)$ is the total number of $(1-a)^k/\delta a^2)m$ for $a$.

For a concrete lemma 1). Choosing suggestions suggest $c$ is probably the fact that the total number of insertions is very rare.

VIII we will show. In these cases will not be used by node $v$ will node $v$ causes a new leaves then results for some non-

Let $R$ be a non-

A rotational rotation or $\rho(v)$ where $\rho(v)$ is the total cost of $m$ insertions

causes a rebalancing $f((1-a)^{-1})$ time operation is all rebalancing

\[ \begin{align*}
\left( \sum_{i=1} \sum_{v \in B_i(v)} f((1-a)^{-1}) \right) \\
\leq \left( \sum_{i=1} \sum_{v \in T(v)} f((1-a)^{-1}) \right) \text{ by lemma 2}
\end{align*} \]

and the observation that $T_{\text{a1}}(v) = 0$ for $a > \log m$ by lemma 1a.

Theorem 5 has some interesting consequences. If $f(x) = x^a$ with $a < 1$ then the total rebalancing cost is $O(m)$ and if $f(x) = x(\log x)^a$ for some $a > 0$ then the total rebalancing cost is $O(m(\log m)^{a+1})$. Thus even if $f(x)$ is fairly large the amortized rebalancing cost (i.e. rebalancing cost per insertion/deletion) is small. We will use this fact extensively in chapters VII and VIII.

III. 5.2 Height-Balanced Trees

Height-balanced trees are the second basic type of balanced tree. They come in many different kinds: AVL-trees, (2,3)-trees, B-trees, HB-trees, ... and (a,b)-trees which we describe here.

Definition: Let $a$ and $b$ be integers with $a \geq 2$ and $2a-1 \leq b$.

A tree $T$ is an $(a,b)$-tree if

a) all leaves of $T$ have the same depth
b) all nodes $v$ of $T$ satisfy $\rho(v) \leq b$
c) all nodes $v$ except the root satisfy $\rho(v) \geq a$
d) the root $r$ of $T$ satisfies $\rho(r) \geq 2$.

Here $\rho(v)$ denotes the number of sons of node $v$.

$(a,b)$-trees are known as $B$-trees if $b = 2a-1$. In our examples we always use $a = 2$ and $b = 4$. We have to address the following questions: how to store a set in an $(a,b)$-tree, how to store an $(a,b)$-tree in a computer, how to search in, insert into and delete from an $(a,b)$-tree.

Sets are stored in $(a,b)$-trees in leaf-oriented fashion. This is not
compulsory, but more convenient than node-oriented storage which we used so far. Let $S = \{x_1, ..., x_n\}$ be a subset of ordered universe $U$ and let $T$ be an $(a,b)$-tree with $n$ leaves. We store $S$ in $T$ as follows.

1) The elements of $S$ are assigned to the leaves of $T$ in increasing order from left to right.

2) To every node $v$ of $T$ we assign $\rho(v)-1$ elements $k_{i_1}(v), ..., k_{i_{\rho(v)-1}}(v)$ of $U$ such that for all $i$ ($1 \leq i \leq \rho(v)$) : $k_{i_1}(v) < k_{i_{\rho(v)-1}}(v)$ and for all leaves $w$ in the $i$-th subtree of $v$ we have $k_{i_{\rho(v)-1}}(v) < \text{CONTENT}[w] \leq k_{i_1}(v)$.

The following figure shows a $(2,4)$-tree for set $S = \{1, 3, 7, 8, 9, 10\} \subseteq \mathbb{N}$

```
      4
     /|
    2 /|
   / | \
  1 3 7
     /|
    8 9 10
```

The most simple method for storing an $(a,b)$-tree in a computer is to reserve $2b-1$ storage locations for each node of the tree, $b$ to contain the pointers to the sons and $b-1$ to contain the keys stored in the node. In general, some of these storage locations are unused, namely, whenever a node has less than $b$ sons. If a node has arity $\rho(v)$ then a fraction $(2\rho(v)-1)/(2b-1)$ of the storage locations will be used. Since $\rho(v) \geq a$ for all nodes (except the root) at least the fraction $(2a-1)/(2b-1)$ of the storage locations is used. In $(2,4)$-trees this might be as low as $3/7$. We will see in section III. 5.3.4 that storage efficiency is much larger on the average. An alternative implementation is based on red-black trees and is given at the end of the section.

Searching for an element $x$ in an $(a,b)$-tree with root $r$ is quite simple. We search down the tree starting at the root until we reach a leaf. In each node $v$, we use the sequence $k_{i_1}(v), ..., k_{i_{\rho(v)-1}}(v)$ in order to guide the search to the proper subtree. In the following program we assume that $k_0(v) < x < k_{i_{\rho(v)-1}}(v)$ for every element $x \in U$ and every node $v$ of $T$. 
which we universe $U$ and ollows.
creasing or-

$k_p(v)_{v-1}(v)$
and for all $T[w] _ {k_1(v)}$.
$\{8,9,10\} \subseteq \mathbb{N}$

$v$ - root of $T$;
while $v$ is not a leaf
do find $i$, $1 \leq i \leq p(v)-1$, such that $k_{i-1}(v) < x \leq k_i(x)$;
$v_i$ - $i$-th son of $v$;

od;
if $x = \text{CONTENT}[v]$
then success else failure fi;

The cost of a search in tree $T$ is apparently proportional to $O(b \cdot \text{height}(T))$; there are $\text{height}(T)$ iterations of the while-loop and in each iteration $O(b)$ steps are required to find the proper subtree. Since $b$ is a constant we have $O(b \cdot \text{height}(T)) = O(\text{height}(T))$ and again the height of the tree $T$ plays a crucial role.

Lemma 4: Let $T$ be an $(a,b)$-tree with $n$ leaves and height $h$. Then

a) $2 \cdot a^{h-1} \leq n \leq b^h$
b) $\log n / \log b \leq h \leq 1 + \log(n/2) / \log a$

Proof: Since each node has at most $b$ sons there are at most $b^h$ leaves. Since the root has at least two sons and every other node has at least a sons there are at least $2 \cdot a^{h-1}$ leaves. This proves a. Part b) follows from part a) by taking logarithms.

We infer from lemma 4 and the discussion preceding it that operation $\text{Access}(x,S)$ takes time $O(\log |S|)$. We will now turn to operation $\text{Insert}(x,S)$.

A search for element $x$ in tree $T$ ends in some leaf $w$. Let $v$ be the father of $w$ then we are done. If $x \neq \text{CONTENT}[w]$ then we proceed as follows:

1) We expand $v$ by giving it an additional son to the right of $w$ (we also say: we split $w$), store $x$ and $\text{CONTENT}[w]$ in $w$ and the new leaf in appropriate order and store $\min(x, \text{CONTENT}[w])$ in $v$ at the appropriate position, i.e. between the pointers to $w$ and the new leaf.

Example: Insertion of 6 into the tree of the previous example yields
2) Adding a new leaf increases the arity of $v$ by 1. If $\rho(v) \leq b$ after adding the new leaf then we are done. Otherwise we have to split $v$.

Since splitting can propagate we formulate it as a loop.

```
while $\rho(v) = b+1$
  do if $v$'s father exists
    then let $y$ be the father of $v$
    else let $y$ be a new node and make $v$ the only son of $y$
    fi;
    let $v'$ be a new node;
    expand $y$, i.e. make $v'$ an additional son of $y$ immediately to
    the right of $v$;
    split $v$, i.e. take the rightmost $(b+1)/2$ sons and keys
    $k_{(b+1)/2+1}(v), \ldots, k_b(v)$ away from $v$ and incorporate them into
    $v'$ and move key $k_{(b+1)/2}(v)$ from $v$ to $y$ (between the
    pointers to $v$ and $v'$);  
    $v = y$
  od
```

Example continued: Splitting $v$ yields
\textbf{Lemma 5.} Let $b \geq 2a-1$ and $a \geq 2$. Inserting an element $x$ into an $(a,b)$-tree for set $S$ takes time $O(\log |S|)$.

\textbf{Proof:} The search for $x$ takes time $O(\log |S|)$. Also we store the path of search in a pushdown store during the search. After adding a new leaf to hold $x$ we walk back to the root using the pushdown store and apply some number of splitting operations. If a node $v$ is split it has $b+1$ sons. It is split into nodes $v$ and $v'$ with $\lceil (b+1)/2 \rceil$ and $\lceil (b+1)/2 \rceil$ sons respectively. Since $b \geq 2a-1$ we have $\lceil (b+1)/2 \rceil \geq a$ and since $b \geq 3$ we have $\lceil (b+1)/2 \rceil \leq b$ and thus $v$ and $v'$ satisfy the arity constraint after the split. A split takes time $O(b) = O(1)$ and splits are restricted to (a final segment) of the path of search. This proves lemma 5.

Deletions are processed very similarly. Again we search for $x$, the element to be deleted. The search ends in leaf $w$ with father $v$.

1) If $x \notin \text{CONTENT}(w)$ then we are done. Otherwise, we shrink $v$ by deleting leaf $w$ and one of the keys in $v$ adjacent to the pointer to $w$ (to be specific, if $w$ is the $i$-th son of $v$ then we delete $k_i(v)$ if $i < \rho(v)$ and $k_{i-1}(v)$ if $i = \rho(v)$).

Example continued: Deleting 6 yields

\begin{center}
\begin{tikzpicture}
  \node {4} child {node {2} child {node {1}} child {node {3}}} child {node {v} child {node {7}}};
  \node {8} child {node {8}} child {node {9} child {node {10}}};
\end{tikzpicture}
\end{center}

2) Shrinking $v$ decreases $\rho(v)$ by 1. If $\rho(v)$ is still $\geq a$ then rebalancing is completed. Otherwise $v$ needs to be rebalanced by either fusing or sharing. Let $y$ be any brother of $v$.

while $\rho(v) = a-1$ and $\rho(y) = a$
do let $z$ be the father of $v$
fuse $v$ and $y$, i.e. make all sons of $y$ to sons of $v$ and move all keys from $y$ to $v$ and delete node $y$; also move one key (the key
between the pointers to \( y \) and \( v \) from \( z \) to \( v \); (note that this will shrink \( z \), i.e. decrease the arity of \( z \) by one)

if \( z \) is root of \( T \)
then if \( p(z) = 1 \) then delete \( z \) fi;
    goto completed
fi;

\( v = z; \)
let \( y \) be a brother of \( v \)

od;

\textbf{comment} we have either \( p(v) \geq a \) and rebalancing is completed or \( p(v) = a-1 \) and \( p(y) > a \) and rebalancing is completed by sharing;

if \( p(v) = a-1 \)
then \textbf{comment} we assume that \( y \) is the right brother of \( v \);
take the leftmost son away from \( y \) and make it an additional (rightmost) son of \( v \);
also move one key (the key between the pointers to \( v \) and \( y \)) from \( z \) down to \( v \) and replace it by the leftmost key of \( y \);
fi;
completely;

\textbf{Example continued}: The tree of the previous example can be either rebalanced by sharing or fusing depending on the choice of \( y \). If \( y \) is the left brother of \( v \) then fusing yields

If \( y \) is the right brother of \( v \) then sharing yields
Lemma 6: Let \( b \geq 2a-1 \) and \( a \geq 2 \). Deleting an element from an \((a,b)\)-tree for set \( S \) takes time \( O(\log |S|) \).

Proof: The search for \( x \) takes time \( O(\log |S|) \). An \((a,b)\)-tree is rebalanced after the removal of a leaf by a sequence of fusings followed by at most one sharing. Each fusing or sharing takes time \( O(b) = O(1) \) and fusings and sharings are restricted to the path of search. Finally note that a fusing combines a node with \( a-1 \) sons with a node with \( b \) sons and yields a node with \( 2a-1 \leq b \) sons.

We summarize lemmas 4, 5 and 6 in

Theorem 5: Let \( b \geq 2a-1, a \geq 2 \). If set \( S \) is represented by an \((a,b)\)-tree then operations \( \text{Access}(x,S), \text{Insert}(x,S), \text{Delete}(x,S), \text{Min}(S), \text{DeleteMin}(S) \) take time \( O(\log |S|) \).

Proof: For operations \( \text{Access}, \text{Insert} \) and \( \text{Delete} \) this is immediate from lemma 4, 5 and 6. For \( \text{Min} \) and \( \text{DeleteMin} \) one argues as in theorem 3.

\((a,b)\)-trees provide us with many different balanced tree schemes. For any choice of \( a \geq 2 \) and \( b \geq 2a-1 \) we get a different class. We will argue in III.5.3.1 that \( b \geq 2a \) is better than \( b = 2a-1 (= \text{the smallest permissible value for } b) \) on the basis that amortized rebalancing cost is much smaller for \( b \geq 2a \). So let us assume for the moment that \( b = 2a \).

What is a good value for \( a \)? We will see that the choice of \( a \) depends heavily on the intended usage of the tree. Is the tree kept in main memory, or is the tree stored on secondary memory? In the later case we assume that it costs \( C_1 + C_2 m \) time units to transport a segment of \( m \) contiguous storage locations from secondary to main storage. Here \( C_1 \) and \( C_2 \) are device dependent constants. We saw in lemma 4 that the height of an \((a,2a)\)-tree with \( n \) leaves is about \( \log n/\log a \). Let us take a closer look at the search algorithm in \((a,b)\)-tree. The loop body which is executed \( \log n/\log a \) times consists of two statements: in the first statement the proper subtree is determined for a cost of \( C_1 + C_2 a \), in the second statement attention is shifted to the son of the current node for a cost of \( C_1 + C_2 a \). Here \( C_2 = 0 \) if the tree is in main memory and \( C_1 + C_2 a \) is the cost of moving a node from secondary to main memory otherwise. Thus total search time is

\[
((C_1 + C_2 a) + C_1 + C_2 a) \log n/\log a
\]
which is minimal for a such that
\[ a \cdot \ln(a-1) = (c_1 + C_1)/(c_2 + C_2) \]

If the tree is kept in main memory then typical values for the constants are \( c_1 \approx c_2 \approx C_1 \) and \( C_2 = 0 \) and we get \( a = 2 \) or \( a = 3 \). If the tree is kept in secondary storage, say on a disk, then typical values of the constants are \( C_1 \approx c_2 \approx C_2 \) and \( C_1 \approx 1000 C_1 \). Note that \( C_1 \) is the latency time and \( C_2 \) is the time to move one storage location. In this case we obtain \( a \approx 100 \). From this coarse discussion one sees that in practice one will either use trees with small arity or trees with fairly large arity.

We close this section with the detailed description of an implementation of (2,4)-trees by red-black trees. A tree is colored (with colors red and black) if its edges are colored red and black. If \( v \) is a node we use \( bd(v) \) to denote the number of black edges on the path from the root to \( v \); \( bd(v) \) is the black depth of node \( v \). A red-black tree is a binary, colored tree satisfying the following three structural constraints:

1) all leaves have the same black depth,
2) all leaves are attached by black edges,
3) no path from the root to a leaf contains two consecutive red edges.

In the following diagrams we draw red edges as wiggled lines and black edges as straight lines.

There is a close relationship between (2,4)-trees and red-black trees. Let \( T \) be a (2,4)-tree. If we replace nodes with three (four) sons by

respective, then we obtain a red-black tree. In the example from the beginning of the section we obtain:
mentioning it. More precisely, we will derive bounds on the amortized rebalancing cost in (2,4)-trees in the next section. These bounds hold also true for red-black trees (if the programs above are used to rebalance them).

Red-black trees can also be used to implement (a,b)-trees in general. We only have to replace the third structural constraint by: "red components have at least a−1 and at most b−1 nodes" (cf. exercise 2).

Finally, we should mention that there are alternative methods for rebalancing (a,b)-trees, b ≥ 2a, and red-black trees after insertions and deletions. A very useful alternative is top-down rebalancing. Suppose that we want to process an insertion. As usual, we follow a path down the tree. However, we also maintain the invariant now that the current node is not a b-node (a node with b sons). If the current node is a b-node then we immediately split it. Since the father is a b-node (by the invariant) the splitting does not propagate towards the root. In particular, when the search reaches the leaf level the new leaf can be added without any problem. The reader should observe that b ≥ 2a is required for this strategy to work because we split b nodes instead of (b+1)-nodes now. The reader should also note that the results presented in section III.5.3.2 below are not true for the top-down rebalancing strategy. However, similar results can be shown provided that b ≥ 2a + 2 (cf. exercises 29 and 32).

Top-down rebalancing of (a,b)-trees is particularly useful in a parallel environment. Suppose that we have several processors working on the tree. Parallel searches cause no problems but parallel insertions and deletions do. The reason is that while some process modifies a node, e.g. in a split, no other process can use that node. In other words, locking protocols have to be used in order to achieve mutual exclusion. These locking protocols are fairly simple to design if searches and rebalancing operations proceed in the same direction (deadlock in a one-way street is easy to avoid), i.e. if top-down rebalancing is used. The protocols are harder to design and usually have to lock more nodes if searches and rebalancing operations proceed in opposite direction (deadlock in a two-way street is harder to avoid), i.e. if bottom-up rebalancing is used. We return to the discussion of parallel operations on (a,b)-trees at the end of section 5.3.2. In section 5.3.2 we put a result on the distribution of rebalancing operations on the levels of the tree. In particular, we will show that rebalancing operations close to the root where locking is particularly harmful are very rare.