SHORT NOTE

OPTIMAL B-TREE PACKING

TIMOTHY M. KLEIN, KENNETH J. PARZYGNAT and ALAN L. THARP†
Computer Science Department, North Carolina State University, Raleigh, NC 27695-8206, U.S.A.

(Received 15 May 1990; in revised form 23 October 1990)

Abstract—This note presents an algorithm for optimally packing a B-tree. The advantages of optimal packing are greater storage utilization and faster retrieval performance. The algorithm inserts data faster than the standard B-tree insertion algorithm and provides simplicity of implementation.

Key words: B-tree, optimal packing

1. INTRODUCTION

The B-tree [1] is one of the most prominent data structures in computer science. Its ubiquity [2] may come from its utility; it is an effective structure for storing data to be accessed both sequentially and directly [3, 4]. The recent article by Chu and Knott [5] describes and analyzes several forms of the B-tree. Even though the B-tree has considerable advantages, their paper clearly indicates that its storage utilization, \( \alpha \), is a deficiency. The expected \( \alpha \) is about 69% with variation depending upon a tree's capacity order.

Because a splitting reduces the \( \alpha \) of a B-tree, the B*-tree [3, 5], or B # -tree [4], postpones the splitting operation by redistributing records among the nodes of a tree. These B-tree variants also split two buckets into three to increase \( \alpha \) but still only achieve about 85% packing at best, unless nodes beyond those adjacent to the insertion node are searched for empty space [6]. This note presents a straightforward algorithm for achieving the optimal \( \alpha \) of a B-tree. Also, an optimally packed B-tree often has a reduced depth which means faster retrieval. Rosenberg and Snyder [7] point out that space optimal trees are nearly time optimal.

One method for achieving optimal packing is to sort the records prior to insertion and redistribute already-inserted records to the left to avoid splittings [8]. This redistribution algorithm suffers from a high insertion cost when the capacity order and number of records inserted increases (measured in hours for inserting 10,000 records on a SAGE M68000-based microcomputer). The greater costs result from more movement of records among the nodes which requires more auxiliary memory accesses. If the insertions can be performed at non-peak times, the higher costs may not present a problem.

2. PREORDERING ALGORITHM

By placing the records into a special order prior to inserting them, we may eliminate most of the record movement. The special ordering places a record in a position such that it will be inserted after the node in which it is to reside has already been split. Assume that we want to insert the records with keys: AA AB AC AD AE AF AG AH. A standard approach would be to insert the records in lexical order. This order always yields pessimal packing:

\[
\begin{array}{c}
AC \\
AF \\
AA AB \\
AD AE \\
AG AH \\
\end{array}
\]

†To whom all correspondence should be addressed.
The optimal contents for the leftmost leaf node (which was the tree's original node) are:

```
AA AB AC AD
```

After the first four insertions in lexical order, the node does indeed contain these records, but the fifth insertion splits it, relocating the right-half records to the newly-created sibling.

The solution to this problem is to defer inserting the right half of the node's optimal contents (i.e. AC and AD) until after that node has been split once. To achieve this result, the algorithm arranges the records for insertion in the following order: AA AB AE AF AG AC AD AH. Now after the first four insertions, the node contains:

```
AA AB AE AF
```

The fifth insertion results in:

```
AE
|
AA AB
|
AF AG
```

Now we are free to insert AC and AD without fear that they will ever be relocated:

```
AE
|
AA AB AC AD
|
AF AG
```

Since AH is the last record to be inserted, it now goes into the rightmost leaf. But if there are instead four additional records to be inserted (AI AJ AK AL), then we must similarly defer inserting AH and AI until after the rightmost leaf node has been split once. The algorithm therefore arranges these remaining records in the following order: AJ AK AL AH AI, resulting in:

```
AE AJ
|
AA AB AC AD
|
AF AG AH AI
```

Further insertion of records AM through AX yields the optimal tree shown in Fig. 1. But note what happens now upon the insertion of AY. The root splits, leaving only AE and AJ in its left half, with no chance that the right half will ever again contain any records. This circumstance is the same as that encountered with the original single-node tree. The optimal contents of this root node are

```
AE AJ AO AT
```

The node does at one time contain these records, as seen in Fig. 1. But upon the insertion of the new record, the node splits and the right-half records are relocated. As before, the solution is to defer inserting the right half of the optimal contents until after the node has been split once.

The contents of this root node enter it not by direct insertion, since B-tree insertions always occur at the leaf level, but rather as a result of splittings among the leaf nodes which promote the middle records to parent levels. To defer inserting AO and AT into the root node, we must therefore defer processing the leaves whose splittings ultimately elevate those two records to it. In general, for any node in the final optimal tree (other than the leaves and the rightmost node on each level), we must defer processing the nodes whose splittings produce its right-half contents until after that node has been split once. This deferring is the reason for the recursive nature of the algorithm.
Fig. 1. Optimal B-tree with 24 records, capacity order = 2.

Algorithm for B-tree Insertion with Preordering for Optimal Packing

The algorithm starts with a sorted file (Oldfile) of all the records to be stored in the B-tree, and returns a new file (Newfile) containing those records in an order which will yield optimal packing when inserted by means of the standard B-tree insertion procedure.

1. Let \( d \) be the capacity order, or one-half the maximum number of records per node. (The algorithm assumes that node capacity is even.) Let \( M \) by \((2 \times d) + 1\), or the maximum number of children per node.
2. Define an element to be a single record.
3. Define a group to be \( M \) elements.
4. Write the first \( d \) elements in Oldfile to Newfile, and remove those elements from Oldfile.
5. Add the minimum sufficient number of dummy elements to Oldfile to make its total number of elements a multiple of \( M \).
6. Divide all the elements in Oldfile into groups.
7. Rearrange the elements in each group by placing the first \( d \) elements after the last element (e.g. if \( d = 2 \), then a group consisting of the elements A, B, C, D, E would become C, D, E, A, B).
8. If \((\text{the number of groups}) > d\), then:
   (a) redefine an element to be equivalent to the current definition of group;
   (b) Go to Step 3;
   else
   (c) Write to Newfile all records (except dummies) that remain in Oldfile.
9. Perform B-tree insertion with the records in Newfile.

The optimality of packing for this preordering algorithm may be proven as follows. If a B-tree has the minimal number of nodes, then it is space-optimal [7, p. 174]. By Theorem 2.2 [7, p. 179] a B-tree has minimal number of modes if and only if for all levels other than the leaf-level

\[ v_L = \lceil \frac{v_{L+1}}{M} \rceil, \]

where \( v_L \) is the number of nodes at level \( L \); the other variables are defined in the algorithm. This equality is trivial for the root node \((L = 0)\) for any multi-level B-tree, since \( v_0 = 1 \) and \( v_L \) therefore cannot exceed \( M \). For the (relatively rare) case in which the number of records is exactly sufficient to yield 100% \( \alpha \), the algorithm ensures (1) by forcing the following sequence of events for every non-root node in the final tree other than the rightmost one on each level:

1. The node’s left half fills with the records that it will contain in the final, optimal tree. Since the node now holds \( d \) records, it has \( d + 1 \) (or \( \lceil M/2 \rceil \)) children.
2. The node’s right half fills with records \( > \) (i.e. occurring later in lexical order) any that it will contain in the final tree. Since the node now holds \( 2d \) records, it must have \( 2d + 1 \) (or \( M \)) children.
3. A subsequent insertion causes the node to split, relocating its right-half records and its rightmost \( d \) children. It now holds the same records it contained after Step 1, above.
4. The node’s right half fills with the records that it will contain in the final tree. The node now holds \( 2d \) records and hence has \( M \) children (which, recursively, contain their final, optimal contents also). All records yet to be inserted into the tree are \( > \) those now in this node and its children, so this node will never be split again. (References to children do not, of course, apply to leaf nodes.)
**Table 1. Nodes visited, comparisons, right shifts and total insertion times (including sorting) using the preorder algorithm for B-tree insertion**

<table>
<thead>
<tr>
<th>Capacity order</th>
<th>Number of records</th>
<th>2000</th>
<th>4000</th>
<th>6000</th>
<th>8000</th>
<th>10,000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>nv</td>
<td>nv</td>
<td>cmp</td>
<td>shf</td>
<td>s</td>
</tr>
<tr>
<td>2000</td>
<td></td>
<td>6536</td>
<td>14,537</td>
<td>22,537</td>
<td>30,537</td>
<td>38,537</td>
</tr>
<tr>
<td>4000</td>
<td></td>
<td>41,009</td>
<td>88,465</td>
<td>139,294</td>
<td>192,691</td>
<td>249,595</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>5025</td>
<td>9295</td>
<td>13,475</td>
<td>18,775</td>
<td>22,940</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>55</td>
<td>55</td>
<td>1 min 22 s</td>
<td>1 min 43 s</td>
<td>5 min 28 s</td>
</tr>
<tr>
<td>30</td>
<td></td>
<td>3438</td>
<td>3438</td>
<td>15,348</td>
<td>21,348</td>
<td>27,348</td>
</tr>
<tr>
<td>60</td>
<td></td>
<td>25,620</td>
<td>25,620</td>
<td>274,130</td>
<td>533,140</td>
<td>696,420</td>
</tr>
<tr>
<td>120</td>
<td></td>
<td>575</td>
<td>23,975</td>
<td>46,925</td>
<td>65,705</td>
<td>69,775</td>
</tr>
<tr>
<td>25</td>
<td></td>
<td>25</td>
<td>25</td>
<td>1 min 22 s</td>
<td>1 min 52 s</td>
<td>2 min 19 s</td>
</tr>
<tr>
<td>50</td>
<td></td>
<td>3899</td>
<td>7899</td>
<td>11,899</td>
<td>17,208</td>
<td>19,889</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>168,430</td>
<td>379,410</td>
<td>627,891</td>
<td>918,871</td>
<td>1,249,851</td>
</tr>
<tr>
<td>200</td>
<td></td>
<td>53</td>
<td>53</td>
<td>2 min 08 s</td>
<td>3 min 39 s</td>
<td>4 min 67 s</td>
</tr>
</tbody>
</table>

The rightmost node on each level is the sibling created by the last splitting on that level. It is, of course, never split, so Steps 2 and 3 of the above sequence are omitted for this node; it never contains records other than those that it will hold in the final tree. Because of the above sequence, every node in the final tree is full, and hence every non-leaf node has $M$ children. Consequently, for any non-leaf level $l$, $v_{l+1} = M \times v_l$. This expression satisfies (1).

For the more common case in which the total number of records is such that the $\alpha$ must be <100%, Step 4 of the sequence must be altered for the rightmost two nodes on each (non-root) level:

4'. The records that the node's right half will contain in the final tree are inserted into it. The node will not necessarily fill, because the number of records to be stored in the tree may be depleted before there are sufficient splittings to elevate $2d$ records to this node.

In other words, the last splitting on a particular level occurs, but there are not necessarily enough remaining records to fill the newly-split node and its newly-created sibling. If these two nodes are not full, they cannot have as many as $M$ children. This deficiency is not great enough to invalidate (1) however, since each of the two nodes must, by the definition of the B-tree, have at least $d+1$ children. All other nodes on this level have $M = 2d + 1$ children. Thus, the minimum total number of children of the nodes on this level is:

$$v_{l+1} = (v_l - 2)M + 2(d + 1)$$

$$= (v_l - 2)(2d + 1) + 2d + 2$$

$$= v_l(2d + 1) - (2d + 1) + 1$$

$$= (v_l - 1)M + 1.$$  

Substitution into (1) yields:

$$0 = \lceil -1 + (1/M) \rceil,$$

which is true for all possible $M$.

### 3. Analysis of the Preordering Algorithm

Table 1 reports the results of applying the preorder B-tree insertion algorithm. The test runs were performed on an IBM PS/2 model 50Z. Table 2 provides the equivalent results for random insertion using the standard B-tree insertion algorithm. Despite the overhead of rearranging records (or pointers thereto) and the initial sort, the preordering algorithm still requires markedly less insertion time than the standard B-tree insertion algorithm with random data. These better insertion times occur because the algorithm trades auxiliary-memory shift operations (shf) for...
Table 2. Nodes visited, comparisons, right shifts and total insertion times for the standard B-tree insertion of random data

<table>
<thead>
<tr>
<th>Capacity order</th>
<th>Number of records</th>
<th>Nodes visited (nv)</th>
<th>Comparisons (cmp)</th>
<th>Right shifts (shf)</th>
<th>Total insertion time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2000</td>
<td>4000</td>
<td>6000</td>
<td>8000</td>
<td>10,000</td>
</tr>
<tr>
<td>5</td>
<td>7159 nv</td>
<td>15,281 nv</td>
<td>23,637 nv</td>
<td>33,637 nv</td>
<td>43,802 nv</td>
</tr>
<tr>
<td></td>
<td>28,827 cmp</td>
<td>63,911 cmp</td>
<td>101,517 cmp</td>
<td>140,870 cmp</td>
<td>181,550 cmp</td>
</tr>
<tr>
<td>25</td>
<td>10,690 shf</td>
<td>21,141 shf</td>
<td>31,721 shf</td>
<td>42,256 shf</td>
<td>52,866 shf</td>
</tr>
<tr>
<td></td>
<td>32 s</td>
<td>2 min 28 s</td>
<td>4 min 14 s</td>
<td>6 min 21 s</td>
<td>10 min 15 s</td>
</tr>
<tr>
<td></td>
<td>4091 av</td>
<td>10,229 av</td>
<td>16,224 av</td>
<td>22,224 av</td>
<td>28,134 av</td>
</tr>
<tr>
<td></td>
<td>65,189 cmp</td>
<td>142,274 cmp</td>
<td>222,063 cmp</td>
<td>304,090 cmp</td>
<td>390,284 cmp</td>
</tr>
<tr>
<td></td>
<td>61,993 shf</td>
<td>124,550 shf</td>
<td>184,800 shf</td>
<td>246,938 shf</td>
<td>308,066 shf</td>
</tr>
<tr>
<td></td>
<td>45 s</td>
<td>2 min 50 s</td>
<td>5 min 19 s</td>
<td>7 min 13 s</td>
<td>12 min 06 s</td>
</tr>
<tr>
<td></td>
<td>3899 nv</td>
<td>7899 nv</td>
<td>11,899 nv</td>
<td>17,208 nv</td>
<td>22,853 nv</td>
</tr>
<tr>
<td></td>
<td>89,352 cmp</td>
<td>204,467 cmp</td>
<td>350,427 cmp</td>
<td>493,132 cmp</td>
<td>641,689 cmp</td>
</tr>
<tr>
<td></td>
<td>125,033 shf</td>
<td>253,548 shf</td>
<td>376,377 shf</td>
<td>504,817 shf</td>
<td>631,657 shf</td>
</tr>
<tr>
<td></td>
<td>65 s</td>
<td>3 min 02 s</td>
<td>6 min 11 s</td>
<td>10 min 15 s</td>
<td>15 min 11 s</td>
</tr>
</tbody>
</table>

primary-memory comparisons (cmp). With the preordering algorithm, records almost always enter any given node in lexical order; this situation requires that the new record be compared with every other record in the node before its proper place is located, but it rarely requires that any records be shifted to make room for the new one. In trees with only two levels, no such shifts are ever required when using the preordering algorithm.

Nodes visited (nv) refers to the number of nodes which were examined (i.e. loaded from auxiliary memory) to determine whether the record currently being inserted belonged there. Integers were used instead of alphanumeric keys for the insertions in the test runs. In practice, the key strings themselves need not be processed with this algorithm; integers may be processed and then used as indices into the sorted file of keys (records).

Miller et al. [9] propose an algorithm for building a space-optimal B-tree in which the first step in forming the optimal structure is to compute the number of nodes on each level of an optimal tree. The records are then loaded into the structure from a sorted order by directly accessing nodes rather than using the standard B-tree insertion algorithm. Although our preordering algorithm may be less intuitive than their optimal-structure algorithm, it is easier to implement because it requires neither computing the final tree nor directly accessing the nodes. Also, the preordering algorithm is a preprocessor which allows the use of the standard B-tree insertion algorithm, which may already be available, and may in fact be the only option available, as is often the case when using commercially published software.

4. CONCLUSIONS

The advantages of an optimally packed B-tree are greater storage utilization and faster performance—a rare case in which there is not a time-space tradeoff. The preordering algorithm even has the advantage of being faster at insertion than the standard B-tree insertion algorithm. A caution in using an optimally packed B-tree is that subsequent random insertions may require splitting operations which will cause real-time delays on insertions, plus decreased storage utilization and retrieval performance. To minimize the real-time splitting requirements, place-holder records could be inserted into the tree when it is loaded. To maintain the high storage utilization and fast retrieval performance, the tree could be rebuilt (by running the algorithm) periodically at non-peak times of use.

REFERENCES