AUGMENTATION OF B-TREE SEARCH OPERATION

Viral Kapadia
Department of Computer Engineering, The M S University of baroda, Vadodara, Gujarat, India

ABSTRACT

B-tree, a well known tree data structure has arguably been one of the best data structure to use in database and filesystems. The traversal of Btree performs an inorder walk. This paper focuses on finding the ith element efficiently from a tree by defining a new attribute size and rank. The augmented function used here to find the ith element in a b-tree operates at O(log t n) complexity without violating the insertion and deletion algorithms. Here ‘t’ signifies the minimum degree of the tree.

I. INTRODUCTION

A tree (strictly speaking, a free tree) is an acyclic, connected, undirected graph. Equivalently, a tree may be defined as an undirected graph in which there exists exactly one path between any given pair of nodes. Since a tree is a kind of graph, the same representations used to implement graphs can be used to implement trees. Tree structures support various basic dynamic set operations including Search, Predecessor, Successor, Minimum, Maximum, Insert, and Delete in time proportional to the height of the tree[1]. Ideally, a tree will be balanced and the height will be log n where n is the number of nodes in the tree. To ensure that the height of the tree is as small as possible and therefore provide the best running time, a balanced tree structure like a red-black tree, AVL tree, or b-tree must be used.

Height, depth and level:

It is easy to confuse the terms used to describe the position of a node in a rooted tree. Height and level, for instance, are similar concepts, but not the same.

• The height of a node is the number of edges in the longest path from the node in question to a leaf.
• The depth of a node is the number of edges in the path from the root to the node in question.
• The level of a node is equal to the height of the root of the tree minus the depth of the node concerned.
A. STRUCTURE OF B-TREE

Unlike a binary-tree, each node of a b-tree may have a variable number of keys and children[1]. The keys are stored in non-decreasing order. Each key has an associated child that is the root of a subtree containing all nodes with keys less than or equal to the key but greater than the preceding key. A node also has an additional rightmost child that is the root for a subtree containing all keys greater than any keys in the node.

A b-tree has a minimum number of allowable children for each node known as the minimization factor. If t is this minimization factor, every node must have at least t - 1 keys. Under certain circumstances, the root node is allowed to violate this property by having fewer than t - 1 keys. Every node may have at most 2t - 1 keys or, equivalently, 2t children.

Since each node tends to have a large branching factor (a large number of children), it is typically necessary to traverse relatively few nodes before locating the desired key. If access to each node requires a disk access, then a b-tree will minimize the number of disk accesses required.

The minimization factor is usually chosen so that the total size of each node corresponds to a multiple of the block size of the underlying storage device. This choice simplifies and optimizes disk access. Consequently, a b-tree is an ideal data structure for situations where all data cannot reside in primary storage and accesses to secondary storage are comparatively expensive (or time consuming).

B. HEIGHT OF B-TREE

For n greater than or equal to one, the height of an n-key b-tree T of height h with a minimum degree t greater than or equal to 2[2],

\[ h \leq \log_t \frac{n + 1}{2} \]

Ref.(For a proof of the above inequality, refer to Cormen, Leiserson, and Rivest pages 383-384.)

The worst case height is O(log n). Since the "branchiness" of a b-tree can be large compared to many other balanced tree structures, the base of the logarithm tends to be large; therefore, the number of nodes visited during a search tends to be smaller than required by other tree structures. Although this does not affect the asymptotic worst case height, b-trees tend to have smaller heights than other trees with the same asymptotic height.

C. OPERATION OF B-TREE

We shall focus here only on the search operation of B-tree. The ordinary search operation is implemented as follows[6]:

```
B-Tree-Search(x, k)
  i <- 1
  while i <= n[x] and k > key[i][x]
    do i <- i + 1
      if i <= n[x] and k = key[i][x]
        then return (x, i)
      if leaf[x]
        then return NIL
      else Disk-Read(c[i][x])
        return B-Tree-Search(c[i][x], k)
```
The search operation on a b-tree is analogous to a search on a binary tree. Instead of choosing between a left and a right child as in a binary tree, a b-tree search must make an n-way choice. The correct child is chosen by performing a linear search of the values in the node. After finding the value greater than or equal to the desired value, the child pointer to the immediate left of that value is followed. If all values are less than the desired value, the rightmost child pointer is followed. Of course, the search can be terminated as soon as the desired node is found. Since the running time of the search operation depends upon the height of the tree.

Here, we search the element E in the given tree in an inorder fashion. The solid lines show the traversal as follows:

II. AUGMENTING THE SEARCHING OPERATION

OS-SELECT(x, i ,b )
\[ r \leftarrow 1 \]
For(j=1 to x.child[b].count) do
\[ r+ \leftarrow x.child[b].no\_key[j] \]
End For
if i = r
then return x
elseif i < r
then return OS-SELECT(x.child[b], i ,0 )
else
then increment b by1
if (x.count=b)
then return OS-SELECT(x.child[b],i-r,0)
else
return OS-SELECT(x,i-r,b)
The no_key[] indicates the size given to each element. The size is calculated as

\[ \text{size} = \text{left.size} + 1. \]

And for the last element of the node, it is calculated as

\[ \text{size} = \text{left.size} + \text{right.size} + 1. \]

And the leaf element are assigned size as 1. This retrieves the \( i^{\text{th}} \) element from the b-tree by comparing the rank of root node with \( i \). And then following the path as described above it retrieves the element. The black solid line shows the path traversed by the modified algorithm.

### III. RESULT ANALYSIS

<table>
<thead>
<tr>
<th>No of data</th>
<th>Measures</th>
<th>B-Tree Search</th>
<th>Modified B-Tree Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>Physical memory usage</td>
<td>1449984</td>
<td>503808</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1466368</td>
<td>503808</td>
</tr>
<tr>
<td></td>
<td>Execution time in millisecond</td>
<td>80</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td></td>
<td>64</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>Ticks</td>
<td>236273488</td>
<td>189992825</td>
</tr>
<tr>
<td></td>
<td></td>
<td>189910886</td>
<td>178269096</td>
</tr>
<tr>
<td>5000</td>
<td>Physical memory usage</td>
<td>503808</td>
<td>503808</td>
</tr>
<tr>
<td></td>
<td></td>
<td>503808</td>
<td>557056</td>
</tr>
<tr>
<td></td>
<td>Execution time in millisecond</td>
<td>75</td>
<td>74</td>
</tr>
<tr>
<td></td>
<td></td>
<td>77</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td>Ticks</td>
<td>220483098</td>
<td>219300312</td>
</tr>
<tr>
<td></td>
<td></td>
<td>228157886</td>
<td>222804901</td>
</tr>
</tbody>
</table>

### IV. CONCLUSIONS

From the above table we can conclude that Modified search function acquires less of physical memory usage.

Moreover the Ticks i.e. Clocks per Cycle also gives a significant reduction.

The execution time of both the functions is significantly analogous.
V. ACKNOWLEDGMENT

We are highly indebted to Prof. Viral Kapadia for his sincere efforts. He constantly guided us. We would like to express our gratitude towards our parents & members of Birla Vishvakarma Mahavidyalaya for their kind co-operation and encouragement which help us in implementation of this paper.

REFERENCES