Technical Report CS74004-R

ROOT-HEAVY DIRECTORY TREE ON DIRECT ACCESS STORAGE DEVICES

Stewart N. T. Shen

Department of Computer Science

College of Arts and Sciences

Virginia Polytechnic Institute & State University

Blacksburg, Virginia 24061

April 1974

I. Introduction

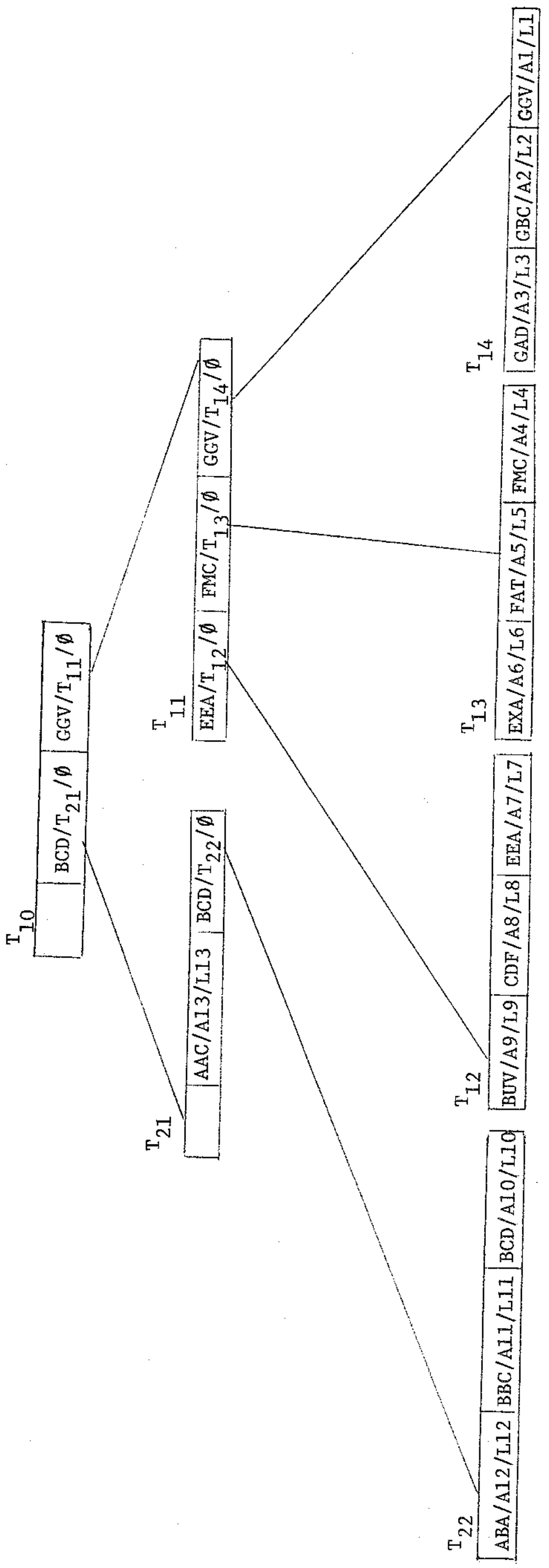
There are many applications in information storage and retrieval that require the use of direct access storage devices (DASD) such as the disk drive and the drum. The typical structure of information for such a system is in the form of a directory and a file of records. The directory contains the keys and the pointers pointing to the associated records or lists of records in the file of records. In a typical application, the entire file of records and the most, if not all, of the directory are stored on a DASD.

The process of finding where a record or a list of records associated with a particular key is located is called directory decoding. This process is essentially to find the particular key in the directory. The decoding technique is directly dependent upon how the directory is structured. When the directory takes the form of a tree structure,some multiway tree is usually used on a DASD. A binary tree structure is ineffficient on a DASD because to use it to decode a key we need to make about $\log_2 N$ accesses, where N is the number of keys. N may be a million in an actual application and this size would require 20 or so accesses to decode a key. Accesses on a DASD is very time consuming compared to operations in the central memory. Multiway trees with may more keys per node can greatly reduce the number of DASD accesses hence can improve the overall search efficiency significantly.

There are many different kinds of multiway directory trees that are used. Some examples are the tree with fixed length key-word truncation [2. pp. 93-98] and the tree with variable length key-word unique truncation [2. pp. 98-104]. This paper proposed an approach which can be applied to modify such trees so that the directory decoding efficiency can be improved.

The proposed appraoch is the root-heavy directory tree. Since different kinds of multiway directory trees are constructed using different techniques and criteria, it would be rather awkward to propose a general definition of the root-heavy directory tree and to present a general algorithm to construct one using any given multiway directory tree. In this paper, a fixed length key-word directory tree similar to the fixed length key-word truncation directory tree of Lefkovitz [2. pp. 93-98] is used as the original directory tree and based on it the algorithm and the definition of the root-heavy directory tree are all presented. The purpose of doing so is to obtain a concise analysis. Both the definition and the algorithm of the root-heavy directory tree can be easily extended to some other types of multiway directory trees. The analysis of the root-heavy directory trees based on other types of multiway directory trees would be a little more complex.

An example would be helpful to illustrate the proposed approach. Let us consider a directory with the following fixed length keys: AAC, ABA, BBC, BCD, BUV, CDF, EEA, EXA, FAT, FMC, GAD, GBC, and GGV. Assume that the complete directory is stored on a movable head disk drive with the tree structure as in Figure 1. The use of only a small number of keys is to simplify our example. An application may have up to thousands or millions of keys.



<u>Figure 1</u>. The Original Directory Tree

In Figure 1, Tij is used to designate j^{th} track in cylinder i; Ak is the symbolic link address; Lk is the corresponding list length; and Ø is used to indicate that the triplet containing it is a reference to another track in the directory and not to the list in the file.

The following definition will be helpful in our further discussions:

Definition 1. A physical record on a DASD is said to be fully filled if what is unused on it is only the desired reserve space. To fill up a physical record is to record as much information on the physical record as possible leaving the desired reserve space unused.

According to Landauer [1], 10% reserve space is adequate for most applications. To construct a directory tree as in Figure 1, the standard procedure is to work from the leaves back to the root and to try to fill up each physical record along the way if possible. In Figure 1, it is assumed that 3 triplets will fill up a physical record. In actuallity, a track of a disk may in fact accommodate about 200 such triplets.

A characteristic of a directory tree constructed in the conventional manner is that the tree tends to be light at the root, that is, the root is not fully filled. Also, the left descendents of the root may not be fully filled either.

II. Root-Heavy Directory Tree

Before considering a definition of the root-heavy directory tree, let us first look at the following algorithm. In the algorithm, the root of the directory tree is filled up by moving some elements (An element is a triplet in Figure 1.) from its right son into it. Its right son is then filled up using the same method. This operation terminates at the right leaf. All the left descendents of the root which are not fully filled are filled up in similar operations in which the descendents are treated as roots of the appropriate subtrees.

Algorithm RHDT:

```
Step 1.  Let ROOT be the root.
Step 2.  Let CURRENT  be ROOT.
Step 3.  If CURRENT is fully filled then go to Step 6.
         If CURRENT has no descendents then go to Step 6.
Step 4.  Try to fill up CURRENT.
         Take away from the right son of CURRENT as many elements as
         possible from the leftmost element on and insert them immed-
         iately in front of the rightmost element of CURRENT so that
         no overflow in CURRENT will occur.
Step 5.  Let CURRENT be the right son of CURRENT.
         Go to Step 3.
Step 6.  If ROOT has no descendent then terminate.
         Otherwise, let ROOT be the left son of ROOT and go to Step 2.
```

The application of Algorithm RHDT to the directory tree in Figure 1 will produce the directory tree in Figure 2.

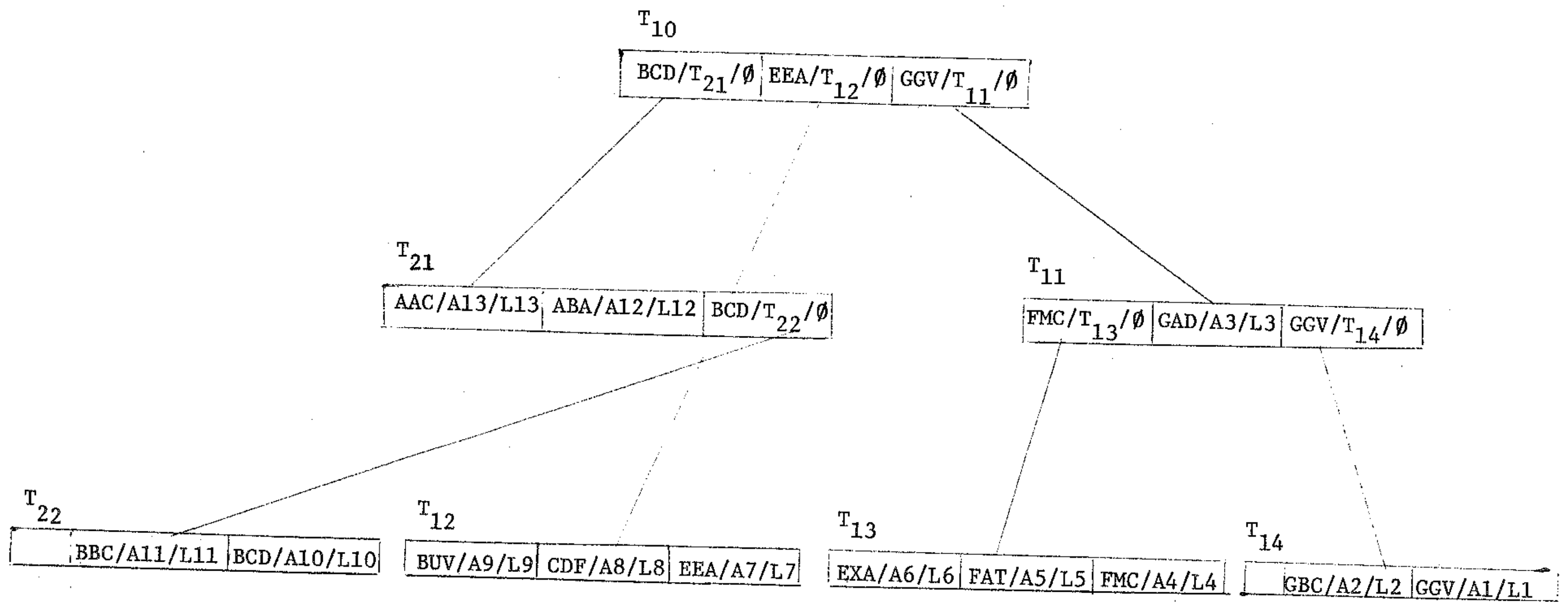


Figure 2. The Root-Heavy Directory Tree

Figure 2 illustrates that Algorithm RHDT produces directory trees in which the roots of all the subtrees of order more than zero are fully filled. In fact, only the left leaf and the right leaf of the tree may be not fully filled. A definition of the root-heavy directory tree may now be given.

Definition 2. A root-heavy directory tree is a directory tree constructed according to Algorithm RHDT using a conventionaly constructed multiway directory tree as input.

A comparison of the decoding efficiencies of the two directory trees in Figure 1 and Figure 2 respectively are given in table 1. Sequential searches in nodes are assumed. Note that the superiority of the root-heavy directory tree is onesided. More general comparisons are given in the next section.

Table 1

Comparisons On The Directory Trees

Key	Original (Figure 1) Number of accesses	Original (Figure 1) Number of comparisons	Root-Heavy (Figure 2) Number of accesses	Root-Heavy (Figure 2) Number of comparisons	Superiority of Root-Heavy Number of accesses	Superiority of Root-Heavy Number of comparisons
AAC	2	2	2	2	0	0
ABA	3	4	2	3	1	1
BBC	3	5	3	5	0	0
BCD	3	6	3	6	0	0
BUV	3	4	2	3	1	1
CDF	3	5	2	4	1	1
EEA	3	6	2	5	1	1
EXA	3	5	3	5	0	0
FAT	3	6	3	6	0	0
FMC	3	7	3	7	0	0
GAD	3	6	2	5	1	1
GBC	3	7	3	7	0	0
GGV	3	8	3	8	0	0
Total	38	71	33	66	5	5

Theorem 3. To fill up a node which has m element-spaces unused and which is at kth level, with the leaf level considered to be level one, will save

$$\frac{b^{k-1}-1}{b-1} \cdot m$$

accesses, assuming each key in the original subtree of its right son is to be decoded once.

Proof. The process of filling up the node involves moving m elements from its right son into it, moving m elements from its right son's right son into its right son, and so on. Move m elements from k-1th level into kth level yields a saving of $m \cdot b^{k-2}$ accesses. Thus, all these moves yield a total saving of $\sum_{i=2}^{k} m \cdot b^{k-i} = \frac{b^{k-1}}{b-1} \cdot m$ □

Theorem 4. To fill up a node which has m element-spaces unused and which is at kth level will save $\frac{b^{k-1}-1}{b-1} \cdot m$ comparisons, assuming each key in the original subtree of its right son is to be decoded once and assuming sequential search is used in each node.

Proof. The number of comparisons necessary to reach the b-m-1 elements which are shifted to the left-hand side in a node is not affected by the movings. To reach all the elements left in the leaf level require the same number of comparisons as before. To reach an element that is moved up needs one less comparison than before. The savings on comparisons to reach the keys associated with all the moved up elements is $\sum_{i=2}^{k} m \cdot b^{k-i} = \frac{b^{k-1}-1}{b-1} \cdot m$. This is also the total saving. □

The above theorems describe the storage requirement and the decoding efficiency of the root-heavy directory tree. There are two other aspects that one should also consider for the purpose of information storage and retrieval. One aspect is the creation efficiency of the root-heavy directory tree. To create a root-heavy directory tree involves an extra step. However, this extra step amounts to only a few updatings. In other words, this step is not time consuming. In fact, for a directory tree with fixed length keys this step can even be avoided by directly creating the root-heavy directory tree. Such a procedure will require only a few extra calculations. The other aspect that one should consider is the updating efficiency of the root-heavy directory tree. As far as filling an element or deleting an element in a node is concerned, the root-heavy directory tree offers no difference because a node has the same amount of reserve space. On the other hand, the speed of reaching a desired node is some times faster for a root-heavy directory tree.

Sequential search is assumed in Theorem 4. In actual applications, binary search is almost always used. When binary search is used, a root-heavy directory tree does not have absolute advantage over the original directory tree any more in the number of comparisons necessary to decode the keys. For some keys the numbers of comparisons necessary will become in fact even larger in the case of the root-heavy directory tree. However, the processing time for the increases in comparisons for these keys are usually very small because, for large N, $\log_2 N$ increases very slowly as N increases. Another point should be mentioned is that when binary search is used in searching a node, the node search time is usually negligible as compared to the node access time.

V. Bibliography

1. Landauer, W. L. "The Balanced Tree and Its Utilization in Information Retrieval", IEEE Transactions on Computers, EC-12, No. 5, Dec. 1963, pp. 863-871.

2. Lefkovitz, D. File Structures For On-Line Systems, Hayden, N. J., 1969.

Acknowledgments

The author wishes to thank B. Claybrook for valuable discussions.