A proposed Algorithm To Measure
The Behavior Of B-Tree

A PROPOSED ALGORITHM TO MEASURE THE
BEHAVIOR OF B-TREE

SHAKIR M. HUSSAIN
ASST. LECT
DEPT. OF COMPUTER SCIENCE
AL- RAFIDAIN UNIV. COLG.

ABSTRACT

This paper is a description and analysis of one
of the data structure types called a B-tree. B-trees are
balanced multi-branch tree structures which make it
possible to maintain large files with a guaranteed
efficiency. A proposed algorithm are presented here to
help us to measure the behavior (such as the tree
utilization and the leaf node utilization) of this type of
data structure.
1. B-Tree

Basically, a B-tree is a balanced tree with a predetermined maximum number of branches from each node. This maximum number of branches (known as the "order" of the tree) can be any value greater than two. An illustration of a single B-tree node is given in Figure 1.

```
  P0   k1   INFO   P1   k2   INFO   Pm-1   Km   INFO   Pm
```

Figure 1. A Sample of B-tree Node

In this node, $k_i$ represent the $i$-th key and INFO is the information (or possibly a pointer to the information) associated with this key. It should be noted here that the keys are in ascending order within any particular node. $P_{i-1}$ is a field that points to a node or group of nodes containing keys less than $k_i$ and $p_i$ points to a node or group of nodes which contains keys greater than $k_i$. Depending on stored position of the B-tree, these pointers may be either addresses in primary memory or secondary storage.

A B-tree which is made up of only one node shown in Figure 2.

```
  2   7   13   17   2
```

Figure 2. A B-tree Consisting of a Single Node

Notice that the tree is made up of one node which contains two keys and three pointers, which are null (indicated by 2). Two important facts that may be pointed out about this B-tree are that there is one level in the tree and the single node of the tree is also the root.

Figure 3 illustrates a little more complicated B-tree of order 3 (a maximum of three way branching from each node). This B-tree consists of two levels and contains four nodes. The single node on the top level (the node with keys of 25 and 35 in Figure 3) is the root node (as will be the case for all B-trees). The three nodes on the bottom level are leaf nodes due to the fact that they have null pointers (as shown in Figure 3, the nodes need not be completely full).
Figure 3. An Order Three B-Tree

B trees can have a variable number of levels as did binary trees and multibranching from the nodes as did indexed sequential, but these alone could be properties of any multi-branching tree (a tree with a variable number of branches from each node). The characteristics of B-trees which make them unique as follows:

(a) Every node has at most m sons.
(b) Every node, except the root and the leaves has at least \( \lceil m/2 \rceil \) sons (the symbol \( \lceil x \rceil \) indicates the smallest integer that is greater than or equal to \( x \)).
(c) The root node has at least two sons unless it is a leaf, in which case it is the only node in the tree (as in Figure 2).
(d) All leaves will have null pointers and will be on the same level, which in fact will be the bottom level of the tree.
(e) A non-leaf node with \( k \) sons has \( k-1 \) keys.

This property along with the first two implies that every node, except the root, will contain between \( \lceil m/2 - 1 \rceil \) and \( m-1 \) keys.

A B-tree has the important characteristic of being built from the bottom up. This implies that when a new level is added to the tree, it will be added as a new root node. Because of this method of building a B-tree, in contrast to a binary tree which is built from the top down, the B-tree constantly stays in balance; all leaf nodes are on the bottom level and all keys in the leaf nodes may be reached by the same number of probes into the tree. Since this is a balanced tree, the longest possible search will be equal to the number of levels in the B-tree. Knuth (4) has given an upper bounded on the number of levels \( L \) in a B-tree of order \( m \) with \( N \) keys to be

\[
L = 1 + \log_{m/2} \left(\frac{N+1}{2}\right)
\]
An example of a tree with \( N = 2,000,000 \) and conveniently chosen order of \( m = 200 \), the maximum number of levels and thus the maximum number of probes into the B-tree will be four. It should be recognized that the number of node probes is very important if each probe requires a reference to a secondary storage. However, if a large number of keys are contained within a node, search performance within the node itself is also a factor.

As can be seen by equation (1), the number of levels and thus the maximum number of probes into a B-tree depends not only on the number of keys in the tree but also upon the order of the B-tree. For this reason, something should be said about the selection of the order when designing a B-tree. The trade-off is between the number of levels in the tree and the size of the nodes. If the tree is stored on a secondary memory device such as disc, an ideal node size is the same as the capacity of a track. The tree can be completely contained within the main memory if storage capacity permits of the computer to eliminate the extra delay in access time due to the disc arm movement and disc rotation. But for extremely large trees this could be impossible. If the primary memory of the computer is actually a virtual store, environment then a page of the machine's memory could conveniently contain one node of the B-tree.

There are three distinct methods of constructing B-trees to organize information. A record associated with a key may be stored adjacent to the key in the tree; all records may be stored in the lowest level of the tree; and finally the records may be stored in a manner entirely independent of the tree structure. The primary application of this type of B-tree is for files that are internally structured and stored such as symbol tables.

The second type of B-tree structure incorporates the concept of storing records directly in the tree but only at the lowest level. Here the keys in the upper levels of the tree act as a set of indices to control the traversal down to the correct "block" which contains the target record. For this reason this type of tree structure is quite similar to indexed sequential. IBM's Virtual Storage Access Method (VSAM) is an example of an application of this type of B-tree.

In the third type of B-tree structure, the key is used solely as an index. The records themselves are not stored directly in the tree but are stored separately and are accessed by pointers in the tree. Thus, there is no need for the records to be physically ordered. This type of file structure is well suited to random access requirements. This method of structuring B-tree, unlike the previous two, all nodes will hold the same type of information.

Besides the three methods of structuring B-tree mentioned above, it should be evident that they may be used in a variety of applications. In fact B-trees are well suited for many applications that involve the use of files or tables of information that must be randomly accessible via a key. Algorithms are available for maintaining B-tree balanced during the insertion and deletion (1).
2. OPERATIONS ON B_TREES

In order to satisfy the five properties of B-trees as previously stated, some rigid rules must be given regarding the three basic operations to be performed on B-trees. Before discussing these operations (search, insert, and delete), it is necessary to mention that there are basically three different classes of B-trees:

[a] Those that hold information only in the leaf nodes.
[b] Those that hold information in all levels of the tree.
[c] Those that hold only pointers to the records which are stored elsewhere.

In the first class of B-tree, all levels except the bottom level of the tree contain only keys and are used as multilevel file index, similar to the indexed sequential structure. This paper will not be concerned with this type of structure.

B-trees of the second class contain information at all levels. The records are in the tree adjacent to their associated keys.

The third class contain nothing but pointers to the records which are stored external to the tree.

2.1 Search

The operation of searching for a record is the basis of all operations on a B-tree (as it is for any file structure) because all other operations depend on it. A nonsequential search for a key will always begin with the root node. For this reason it might be advisable to keep the current root node in the internal memory of the computer at all times. When searching any B-tree node, either a linear or binary search may be used since the keys in a node are stored in ascending order from left to right. If the key is found when searching a node, then the related information that needs to be "retained" is the identification of the node in which the key is found and its position within this node. If the key is found in this node, then it is easy to determine between which two keys this key should be logically located.

If the pointer of the node at this point is not the null pointer, then the successor of the node is accessed and the search process starts over as if this were the new root node (this is actually the root node for a subtree). This process continues until the key is found or the bottom of the tree is reached.

If this pointer is null however, the node is a leaf node (the node is on the bottom level of the tree) and the related information that should be retained in this case is the identification of the node the key should be in and the position which the key should occupy within this node.
If a search for key 25 is made in the tree in Figure 4, then it is immediately found in position 1 of node 1.

![B-Tree Diagram](image)

**Figure 4. An Order Three B-Tree With Three Levels**

Using the same tree, if a search is made for the key 20, then the following action would take place: the root node 1 is scanned and it is determined that the key is less than 25, therefore the left branch is taken and node 2 is accessed. The key is larger than the single key in node 2 so the right branch is taken and 5 is accessed. The key is finally located in position 1 of node 5. One final example is presented to illustrate the method used in determining that a key is absent from the tree. When searching for key 25 in Figure 4, the root node is scanned, the right branch is selected, and node 3 is accessed. The key falls between the first and second keys in node 3 so pointer 2 is used to access node 7. It is important to remember that as soon as one key in a node is found to be larger than the key sought, the scan can be stopped because all keys further to the right will also be larger. Scanning across node 7, it is determined that the key falls between the first and second keys in the node implying that the pointer 1 is to be selected for the next branch. Since this pointer is null, it is determined that the key is not in the tree but should be in position 2 of node 7. From this example, it is evident that a key cannot be determined to be absent from the tree until one node from each level has been inspected. These examples also show that any new key that will be added to the tree will be inserted into a leaf node.

### 2.2 Insertion

Insertion of a new key into a B-tree is somewhat different from insertion into an ordinary binary tree or an indexed sequential file because of the special properties of B-trees. By using the B-tree search technique, the position within the tree in which the key should be found may be determined. If, while searching for this position, the key is found to be already in the tree, then some type of error condition should be raised. If the key is not found, then it should be inserted in the correct ordered position in the appropriate leaf node (Figure 5).
action key is key is associated to a key associated with a key.

When the key associated with a key is second passed in the next order, the key will be added into an additional node in the tree as indicated by the tree diagram.

The nodes in a B-tree have a maximum (and also a minimum) size which, if exceeded during the insertion of a key, will violate one of the basic properties of B-trees. If this occurs, then a corrective action known as two-way splitting must take place. A two-way split constitutes taking an "overfull node" and dividing it into two nodes, each of which is approximately one-half full. Due to the fact that this adds one new node to the tree, a new pointer must be created in as much as all nodes in a tree except the root must be pointed to by another node. Therefore, the middle key from the node being split is moved up into the father of this node (the predecessor node of any node in the tree). This key is inserted into the father node in its correct ordered position and all other keys and pointers are shifted to the right one place. An empty pointer space is created which can now be used to point to the new node just added to the tree due to the split (Figure 6).
Figure 6: An Order Five B-tree illustrating the Two-Way Split Process During Insertion

If the father node has overflowed because of this action, the entire two-way split process is applied to the father node. Logically then, this splitting process may be propagated back up through the entire tree. If the node being split does not have a father then it must be the root node. In this case the two-way split creates a new root node which necessarily adds one level to the height of the tree (Figure 7). The fact that the number of levels in the B-tree increases only when a new root node is added to the tree shows that a B-tree is actually built from the bottom up.

a) Before Insertion of Key 17

Figure 7. An Order Two B-tree Which Adds to the Total Availability of the Numbers it Can Hold. The Empirical Formula (3):

\[ U = 10^k \]

Where

- KEYS
- NODES
- MAX

b) After Insertion of Key 11

a) Before Insertion of Key 11
Careful examination of the two way split process, illustrates that the node involved in the split will always remain on the same level (relative to the bottom). For this reason, once a node is classified as a leaf node, then it will always be a leaf as long as it is part of the tree.

As previously stated, this process of two way splitting may propagate back through the tree to the root node. If at any point of this propagation of splits a leaf node does not exceed the maximum node size after the insertion, then the process stop immediately. It is not necessary that splitting be propagated all the way to the root node of the tree.

Empirical evidence given by Bayer and McCreight (2) suggest that by using this method of insertion, the utilization of the tree will be approximately 66-70% of the total available space in the tree. The term utilization as used here is the ratio of the number of keys actually in the tree to the maximum number of keys the tree can hold. The utilization is given in terms of a percentage and defined by the formula (9):

$$ U = 100 \cdot \frac{\text{KEYS}}{\text{NODES} \cdot \text{MAX}} $$

where

- **KEYS** is the number of keys in the tree
- **NODES** is the number of nodes in the tree
- **MAX** is the maximum keys per node (one less than the order of the tree)
2.3 Deletion

Just as in the case of insertion, the deletion of a key from a B-tree is straightforward unless one of the basic properties of B-tree is violated. Because of the characteristic of B-tree being constantly in balance, a key may have to be removed from a node which may cause the size of this node to fall below the minimum node size.

There are two types of nodes in a B-tree from which a key may be deleted: a leaf node or a non-leaf node. Deleting from a leaf node does not cause immediate problems because the deletion may take place without regard to losing a pointer since all pointers in a leaf are null.

A problem does arise however, when attempting to delete a key from a non-leaf node or more directly a node with non-null pointers. This implies that a node or possibly a complete subtree below this node will be lost. To avoid this, the key that is being deleted is merely replaced by the next largest (or next smallest) key in the tree. The next largest key is the smallest key in the subtree pointed to by the pointer immediately to the right of the key being deleted. The smallest key in a subtree is found by the following pointer zero down through the tree until a null pointer is encountered, indicating a leaf. The first key in this leaf node is the smallest key of that subtree. (Similarly, the next smallest key in the tree is the largest key in the leaf subtree.) The only problem left to handle in the deletion process is if the node size of the leaf that is reduced falls below the minimum node size allowed in the tree. If this condition does not occur then the deletion of the key is completed. However, if this problem arises, one of two possible corrective actions must be taken. These two actions, which are mutually exclusive, are known as:

- cationation, and
- underflow

In the case of cationation, the node that is too small and a brother node (a node which has the same father and is immediately adjacent to the node in question) are cated and joined together to form a single node. This process can only take place if the sum of the numbers of keys in the two brother nodes being cated is strictly less than the maximum number of keys allowed in any node in the tree. The cationation will decrease the number of nodes in the tree by one so the pointer to this lost node also must be removed. The father key is the key in the father node that logically falls between the two brothers being combined. Therefore the father key also becomes part of the new node being formed which means that it is deleted from the father node along with the no longer needed pointer. This is illustrated by the subtree shown in Figure 8. When key 6 is deleted from node 2 in Figure 8a, node 2 and 3 along with father key 10 will be combined to form a new node 2 as shown in Figure 8b.
Because a node (a node section) is only being removed by one the key indexes are deleted. It is deleted but combined.

Figure 8. A Subtree of an Order Five B-Tree Illustrating the Catenation Process Used During Deletion.

Since the father node has been reduced in size, it must be tested to see if it has fallen below the minimum node size. If so, either catenation or underflow must be performed on this node. Catenation, as in the case of tree splits during insertion, may be propagated through the tree toward the root node. If, during a catenation, the father of the two brothers being combined is the root of the tree and it contains only one key, then the node formed by the catenation process is the new root of the tree, and the number of levels in the tree is reduced by one (Figure 9).

a) Before Deletion of Key 7

b) After Deletion of Key 6
b) After Deletion of Key 7

Figure 9. The deletion of a Key From an Order Five B-tree Which Results in the Construction of a New Root Node.

If concatenation is precluded by the upper limit on the node size, an "underflow" must be performed on the nodes involved.

The underflow process comprises equal distribution of the keys between the two nodes. The father key as well as the keys of the two brothers are involved in this distribution. The underflow algorithm can be described as a stepwise process where the father key is moved to the node that is too small and a key from its brother is used to replace the father key. This is repeated until the two brothers are the same size (or as near as possible). For example, let Figure 10a represents the configuration of part of an order 7 B-tree after the deletion of a key. The "underfull" node in this case is node 2 and it is the left brother. Figure 10b represents the same portion of the tree after the underflow has taken place.

2.4 Insertion

After look method if it becomes too too full because of keys of the underflow, well as the Figure 1 before excess overflow to therefore a resulting th

a) Before Underflow

| 5 | 10 | 15 | 20 |
2.4 Insertion With Overflow

After looking at the basic insertion algorithm and deletion with underflow, a method of insertion using an overflow technique to handle nodes when they become too full can be developed. This method comprises movement of keys between brothers. Overflow is very similar to underflow. When a node becomes too full because of an insertion of a key, an attempt is made to redistribute the keys of the overfull and a brother before resorting to a node split. Overfull, like underflow, can be considered to be a stepwise process involving the father node as well as the keys in the brothers.

Figure 11a is an example of a tree after a new key has been inserted but before execution of an overflow. Figure 11b illustrates the tree after completing an overflow to the right. Overflow can go in either direction to either brother; therefore an overflow to the left can be performed on the tree in Figure 11a resulting the tree in Figure 11c.
3. Properties of B-Tree Structure

Instance of leftmost direction
perform
brothers
nodes as full. An is
been sta
forces a
leaves;

The p
on B-tree
by using
The a
operator
proced
modifie

The r
needed
the tree

The r
- The n
- The n
- A sm
- inser
- The c
- A sw
to be
- The i

The p
possit
the co
SEA
'origin'
INS
a new
HEAD

Figure 11. An Order Five B-Tree Illustrating the Overflow Insertion Technique

It is important to note at this point that an overflow (like an underflow) changes only the contents and not the size of the father node. For this reason, overflows will not propagate back up through the tree.
3. Proposed Algorithm

This proposed algorithm is dedicated to the problems revolving around the structure of a B-tree, the methods of insertion and deletion in B-trees. For instance, methods of handling overfull nodes. If a node is overflowing and it is the leftmost or rightmost son of a father then an overflow may take place in only one direction. A possible variation is to attempt an overflow to a cousin node before performing a node split if the other node is full. Also if a node and its two brothers are full, a four way split requires dividing the contents of the three full nodes and producing four nodes, each of which is approximately three-fourths full. An important condition on B-tree that must be taken into consideration has been stated by Knuth (5). "The condition that all leaves be on the same level forces a characteristic behavior of B-tree. B-trees are not allowed to grow at their leaves; instead they are forced to grow at the root."

The purpose of this algorithm is to perform the three basic operations possible on B-trees: search, insert, and delete and provide an analysis of tree utilization by using different methods of insertion with trees of different order.

The algorithm is structured so that the procedures that perform the tree operations are combined in a basic package (main program). All auxiliary procedures are external to this package so that they may be conveniently modified.

The main algorithm (MAIN) performs two functions: read in the parameters needed and then call the B_TREE procedure that perform various operations on the tree.

The main parameters needed are:
- The minimum number of keys allowed in one node (MIN).
- The maximum number of keys to be in the tree (NUM).
- A switch which indicates if the method of overflow is to be used during insertion (OVERFLOW).
- The overflow direction (left or right overflow).
- A switch which indicate whether a two way split or three way split is to be used (2_W_SPLIT).
- The number of trees to be generated (#_of_trees).

The B_TREE procedure reads a transaction code to determine one of the four possible operation (insert, search, delete, and output) to be performed and call the correct procedure to carry out this task.

SEARCH is the routine that searches the tree for a particular key. The search originates with the root node. A binary search is used when "looking" for the key.

INSERT is the procedure that is called to insert a key into the B-tree. Each time a new key is added to the tree, the key count is increased by one. The routine HEAD_NODE is called only when attempting to insert a key into an empty tree.
Before inserting the key, the node size must be checked and if the node size has been exceeded, then either a two way split or an overflow will be performed according to the status of the input parameters provided in the main routine.

When the deletion of a key from the tree is requested, the procedure DELETE is invoked. First the search procedure is called to position the key in the tree. If the key is not found then the delete request is ignored. Otherwise one of two methods is used to delete the key: one method for leaf nodes and another for nonleaf nodes.

**OUTPUT procedure print the following information:**
- The number of keys in the tree.
- The order of the tree.
- The number of levels in the tree.
- The number and percentage of the available nodes used in the tree.
- The total utilization of the tree.
- The frequency count of the number of keys in the leaf nodes.
- The frequency utilization of the leaf node in the tree.
- The number and percentage of overflows that occurred during all insertions in the tree.
- The total number of sibling nodes referenced during all of these overflows.
- The total number of sibling nodes referenced during all splits that have occurred in the construction of the tree.

### 3.1 Algorithms

```c
main
    call read-parameter
    if notree = 1
        # of trees = 1
        print parameter
        MAX = 2 * MNK
        ORDER = MAX / 1
        NN = \ceil{\frac{2 + (\frac{3}{2}) \cdot MNK}{2 + \frac{3}{2} \cdot NN}}
        for tree count = 1 to # of trees
            keycount = nodeייטtings=prev_output_size = 0
            # of overflows = # of ref = 0
            # of ref = split = 0
            real = level = 0
            call B-TREE
        end for
    end if
end main
```
A TREE
  do forever
    read transaction
    if transaction is insert
      call INSERT
    if transaction is search
      call SEARCH
    if transaction is delete
      call DELETE
    if transaction is output
      call OUTPUT
    if transaction is end
      return to main
  end do
end A TREE

SEARCH
  next_node = root
  do forever
    access next_node
    perform binary search on node for key
    if key found
      save position where key is found
      return
    else
      if leaf_node
        save position where key should be
        return
      else
        determine next_node
      end
    end
  end do
end SEARCH

INSERT
  if tree empty
    current_node = 0
    key_current = key_current + 1
    call HEAD_NODE
    return to A_TREE
  else
    call SEARCH
    if key found
      return to A_TREE
    else
      do forever
        insert new key into current_node
        if first_pass
          key_current = key_current + 1
          turn on first_pass switch
        end
      end forev
If node size exceeded
  call SIZE_EX
  \[\text{also}\]
  replace current node in tree
  return to B_TREE
end
end do
end
end INSERT

SIZE_EX
  do forever
    if overflow insertion
      determine overflow direction
      get father node
      if not underflow
        call OVR
      end
      if two way split
    end
  end do
end SIZE_EX

OVERFLOW
  determine position of father key
  if (overflow direction is left and left brother exists) OR
  (overflow direction right and right brother exists)
  do forever
    \[\text{P}: \text{Left}\]
    access brother node
    if brother not full
      perform overflow to the left
      replace nodes into the tree
      return to INSERT
    end
    \[\text{P}: \text{Right}\]
    access brother node
    if brother node not exist
      call 3_W_SPLIT
    end
    if right brother not exist
      call 3_W_SPLIT
    end
    \[\text{P}: \text{Other}\]
    access brother node
    if brother node not exist
      if not both direction
        call 3_W_SPLIT
      else
        if other direction
          call 3_W_SPLIT
        end
      end
    end
end
else if left brother not exist
    call 3_W_SPLIT
    end
    end
    else
    perform overflow to the right
    replace nodes into the tree
    return to INSERT
    end
    end
    end
end do
end
end OVERFLOW

3_W_SPLIT
if not 3_w_split specified
    replace nodes into the tree
else
    set up nodes for 3-way split
    generate a new node
    calculate size of the 3 nodes
    move key from left node into new center node
    leaving left node in final form
    move father key into new center node
    move key from left node into father key
    complete the updating of the new center node
    shift father node to make room for new key
    if 3-2 tree P^_numnodes <= 3
        take father key from center node
    else
        take new father key from right node
        complete the updating of the right node
        end
        replace these brother nodes into the tree
        if father size not exceeded
        replace father node into the tree
        return to INSERT
    else
        return to SIZE_EX
    end
end 3_W_SPLIT
HEAD_NODE
increment number of levels in tree by 1
generate new node for the tree
insert the key into the new node
set up the two pointers from this node
make this node as the root of the tree
place the node into the tree
return
end HEAD_NODE

DELETE
    call SEARCH
    if key not in the tree
        return
    end
    if key in a leaf node
        remove key from leaf node
        replace node into the tree
        if node not root
            call SIZE_CHK
        else
            if tree empty
                root = level = 0
            end
        end
    end
    else
        access leaf with next largest key
        replace key being deleted with next largest key
        replace non_leaf node into the tree
        reduce size of leaf node
        replace leaf node into the tree
        call SIZE_CHK
    end
end DELETE

SIZE_CHK
    if node not too small
        return
    end
    if node is root
        return
    end
    access father node
    if leaf node rightmost sibling
        access left brother
        else
            access right brother
        end
    if deletion possible
        if left brother too small
            move father key into left brother
            move smallest key from right brother to father key

OUTPUT
print _o_key
" calculate _n_nodes = _n_usage = _n_nodes + _n
print _o_nodes
" calculate _o_keys = _o_keys + _k
print _o_keys
4. Discussion

The advantage of this method can be found through the statistical information about the tree utilization, also the number of node accesses during overflows and splits. If the tree is stored on a secondary storage device such as disc, each of these node accesses represents a disc access which can be a very important factor to show the overall efficiency of the B-tree.

Tree utilization and node accesses are very important in the construction and maintenance of B-tree but this is not the only area in which these play a vital role. When searching for a key in a B-tree the maximum length of search path is the number of levels in the tree and of course this means a node access at each level. Therefore the prediction of the number of levels that will result in the tree can be very important factor in choosing a node size for the tree.
REFERENCES


(9) Standish, T. A. Data Structure Techniques. Addison-Wesley, 1980

(10) Van Doren, J. R. "Information Organization and Retrieval." (Unpub. Class notes, Oklahoma State University. 1983)

I would like to express my appreciation to Dr. Alaa H. Alhamami, the Head Department of the Computer Science at Al-Rafidain University Colledge for his valuable suggestions and advice provided during this study.

Thanks are also due to Dr. Hilal M. Yousif, the dean of Al-Rafidain University College for his help and support.

Finally, a note of thanks to the members of the computer center / Al-Rafidain University College for their help during typing this study.