1-1-1982

Comparison of tree-like structures for data maintenance.

John Chun-Hua Chen

Recommended Citation

Chen, John Chun-Hua, "Comparison of tree-like structures for data maintenance." (1982). *Theses and Dissertations.* Paper 2431.

COMPARISON OF TREE-LIKE STRUCTURES
FOR DATA MAINTENANCE

by

John Chun-Hua Chen

A Thesis
Presented to the Graduate Committee
of Lehigh University
in Candidacy for the Degree of
Master of Science
in
Computing Science

Lehigh University
December 1982

CERTIFICATE OF APPROVAL

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

Dec. 10, 1982
(date)

Professor in Charge

Head of Division

ACKNOWLEDGEMENTS

The author wishes to express his sincere thanks to his advisor, Professor S.L. Gulden, for his guidance, his helpful consultation, his inspiration and kind assistance while working for the thesis.

The author would like to give thanks to his dear parents, brother and his dearest wife for their endless encouragement and blessing.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Digital computers and their programs are among man's most logically complex artifacts. Computer process information a billion times faster than a person with pencil and paper, under the control of programs that sometimes contain hundreds of thousands of instructions. The feasibility of a proposed computer application often hinges on the efficiency with which large masses of data can be organized. Recognizing the importance of this aspect of computation, we consider some methods of searching through large amounts of data to find a particular piece of information. As we shall see, certain methods of organizing data make the search process more efficient. Since searching is such a common task in computing, a knowledge of these methods goes a long way toward making a good programmer. This paper is devoted to compare the tree-like structures and their search techniques for data maintenance. A comparative study on five different kind of tree structures was done experimentally. These are balanced binary tree structure, B-tree structure, symmetric binary B-tree structure, 2-3 tree structure and son tree structure. The number of nodes is related to the tree construction time, the height of trees and the search-time of all nodes.

1. INTRODUCTION

1.1 Trees

A tree is a finite set of elements that is either empty or contains a specified element called the root of the tree where the remaining elements are partitioned into disjoint subsets, each of which is itself a tree. These subsets are called the subtrees of the original tree. Each element of a tree is called a node of the tree.

There are many terms which are often used when referring to trees. Consider the tree in Figure 1. This tree has 14 nodes, each data item of a node being a single letter for convenience. The root is designated A, and we will normally draw trees with their root at the top. The indicated lines are not part of the tree but are used to indicate the subtree relationship.

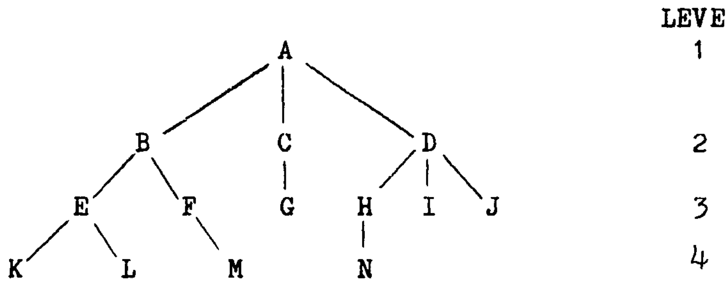


Figure 1. A sample tree

The number of subtrees of a node is called its degree. The degree of A in Fig.1 is 3, that of C is 1, and that of G is 0. A node that has degree zero is called a leaf or terminal node. The set

{K, L, M, G, N, I, J} is the set of leaf nodes of Figure 1. The other nodes are referred to as nonterminals. The roots of the subtrees of a node, X, are the children of X. X is the parent of its children. Thus the children of D are H, I, J; the parent of D is A. Children of the same parent are said to be siblings. For example H, I, and J are siblings. The degree of a tree is the maximum degree of the nodes in the tree. The tree in Figure 1 has degree 3. The ancestors of a node are all the nodes along the path from the root to that node. The ancestors of N are A, D, and H.

The level of a node is defined recursively as follows. The root is taken to be at level one. If a node is at level p, then its children are at level p+1. Figure 1 shows the levels of all nodes in that tree. The maximum level of any element of a tree is said to be its depth or height. The number of branches or edges which have to be traversed in order to proceed from the root to a node X increased by one is called the path length of X. The root has path length 1, its direct children have path length 2, etc. The path length of a tree is defined as the sum of the path lengths of all its components. It is called its internal path length.

1.2 Binary Trees

A binary tree is a tree in which each node has degree no more than two. The two subtrees at each node (possibly empty) are called its left and right subtrees.

A conventional method of picturing a binary tree is shown in Figure

2. This tree consists of ten nodes with A as its root. Its left subtree is rooted at B and its right subtree is rooted at C. This is indicated by the two branches emanating from A: to B on the left and to C on the right. The absence of a branch indicates an empty subtree. For example, the left subtree of the binary tree rooted at C and the right subtree of the binary tree rooted at E and D are both empty. The binary trees rooted at G, H, I, and J have empty right and left subtrees.

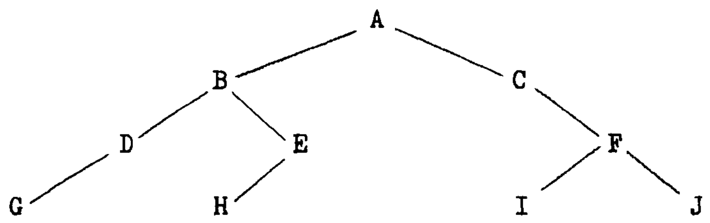


Figure 2. A Binary Tree

A complete binary tree of level n is one in which each node of level n is a leaf and in which each node of level less than n has nonempty left and right subtrees and each node at level n is a leaf. Figure 3 illustrates a complete binary tree.

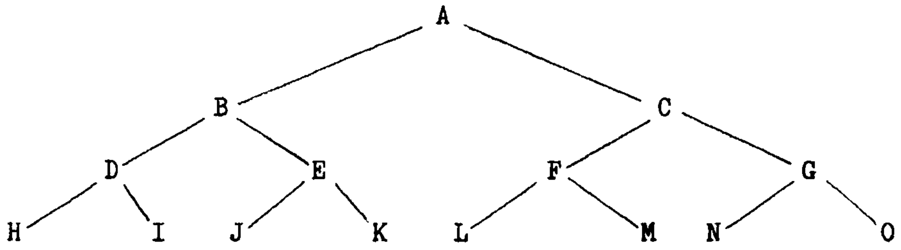


Figure 3. A complete binary tree

1.3 Applications of Binary Trees

A binary tree is a useful data structure when two-way decisions must be made at each point in a process. For example, suppose that we wanted to find all duplicates in a list of numbers. One way of doing this is to compare each number with all those that precede it. However, this involves a large number of comparisons. The number of comparisons can be reduced by using a binary tree. The first number is read and placed in a node which is established as the root of a binary tree with empty left and right subtrees. Each successive number in the list is then compared to the number in the root. If it matches, we have a duplicate. If it is smaller, the process is repeated with the left subtree, and if it is larger, the process is repeated with the right subtree. This continues until either a duplicate is found or an empty subtree is reached. In the latter case, the number is placed into a new node at that position in the tree. Figure 4 illustrates the tree that would be constructed from the input 20, 25, 9, 16, 13, 29, 7, 10, 26, 9, 32, 28, 16, 20, 10, The output would indicate that 9, 16, 20, and 10 are duplicates.

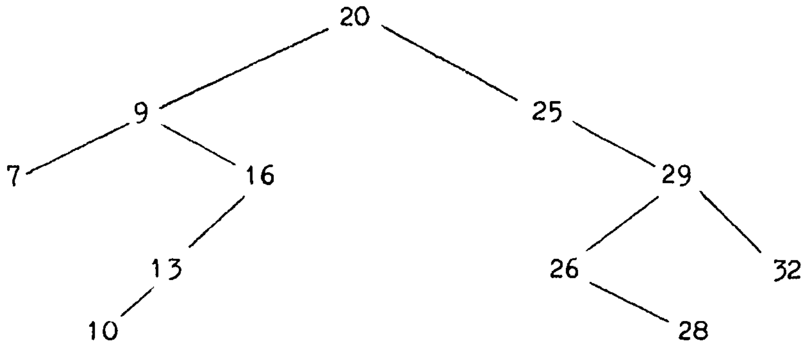


Figure 4. A binary tree constructed for finding duplicates

2. BALANCED BINARY TREE

2.1 Property of Balanced Binary Tree

A tree is said to be balanced if and only if for every node the heights of its two subtrees differ by at most 1. [AVL-balanced tree]

Each node in a balanced binary tree has a balance of 1, -1, or 0, depending on whether the height of its right subtree is greater than, less than, or equal to the height of its left subtree. The balance of each node is indicated in Figure 5.

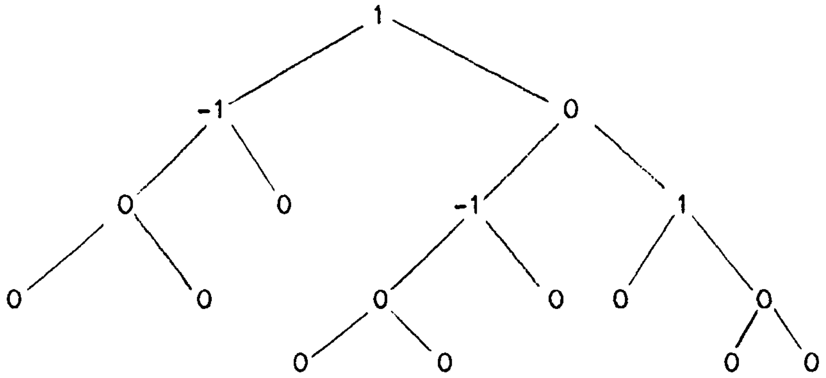


Figure 5. A balanced binary tree with indicator of each node

Suppose that we are given a balanced binary tree and insert a new node into the tree. Then the resulting tree may or may not remain balanced. Figure 6 illustrates all possible insertions that may be made to the tree of Figure 5. Each insertion that yields a balanced tree is indicated by a B. The unbalanced insertions are indicated by a U and are numbered from 1 to 12. It is easy to see that the tree becomes unbalanced only if the newly inserted node is a right descendant of a node which previously had a balance 1 (this occurs in cases U9 through U12) or if it is a left descendant of a node which previously had a balance of -1 (cases U1 through U8). In Figure 6, the youngest ancestor that becomes unbalanced in each insertion is indicated by the numbers contained in three of the nodes.

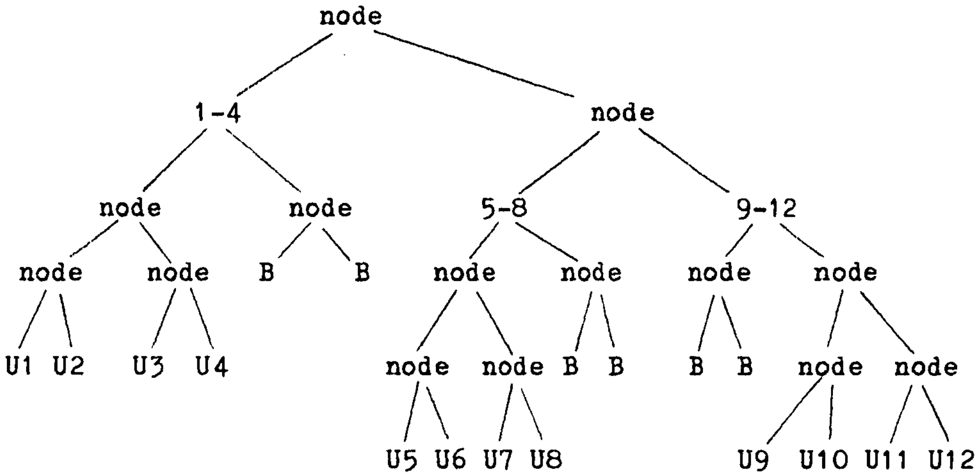


Figure 6. A balanced binary tree and possible additions

To maintain a balanced tree, it may be necessary to perform transformations called rotations on the tree. Consider the trees of Figure 7 and Figure 8, Figure 9 indicates the balancing of the tree in Figure 7 by a so-called LL rotation. Figure 10 indicates the balancing of the tree in Figure 8 by a so-called LR rotation. By symmetry there are also RR and RL rotations.

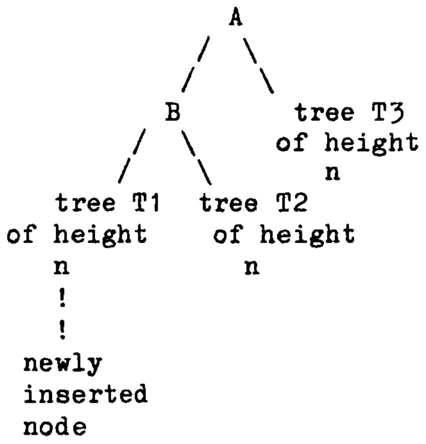


Figure 7. A sample unbalanced binary tree

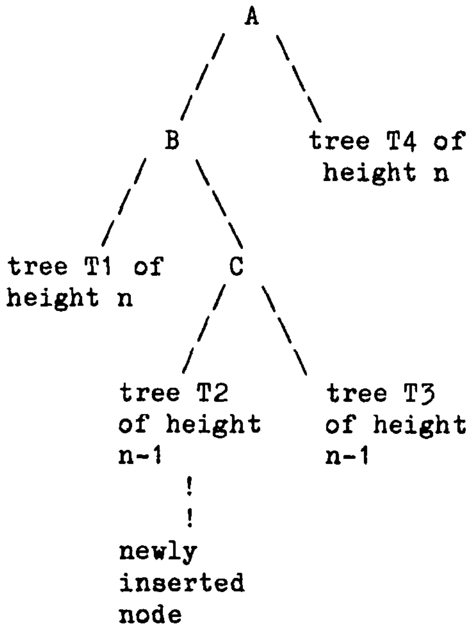


Figure 8. A sample unbalanced binary tree

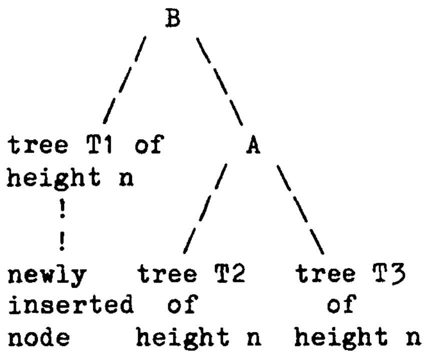


Figure 9. An unbalanced binary tree from Figure 7 after LL rotation becomes a balanced binary tree

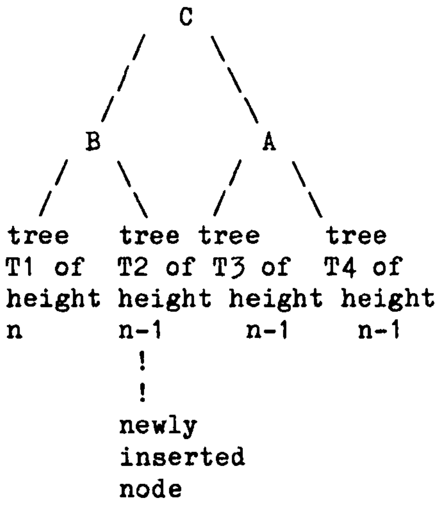


Figure 10. An unbalanced binary tree from Figure 8 after LR rotation becomes a balanced binary tree

2.2 Analysis of Balanced Binary Tree Algorithm

The following indicates an algorithm to maintain a balanced tree. See [1].

Data structure of the balanced binary tree:

```
TYPE     ref = ^node;
         node= RECORD
                  key:integer;
                  count:integer;
                  left,right:ref;
                  balance:-1,0,1
                END;


PROCEDURE search(x:integer;VAR p:ref;VAR h:boolean);
VAR p1,p2: ref;    { h=false }
BEGIN
      IF p=nil THEN
      word is not in tree; insert it
      ELSE
      IF x<p^.key THEN
      BEGIN
            search(x,p^.left,h);
            IF h THEN {left branch has grown higher}
            CASE p^.balance OF
            1: adjust p^.balance=0; h=false;
            0: adjust p^.balance=-1;
           -1: BEGIN
                   {need rotation to rebalance}
                   let p1 point to p^.left;
                   IF p1^.balance=-1
                      THEN LL rotation transformation
                      ELSE LR rotation transformation;
                   adjust p^.balance=0; h=false
                END{-1}
            END {CASE}
       END ELSE
       IF x>p^.key THEN
       BEGIN
             search(x,p^.right,h);
             IF h THEN {right branch has grown higher}
             CASE p^.balance OF
             -1: adjust p^.balance=0; h=false;
              0: adjust p^.balance=1;
              1: BEGIN
                    {need rotation to rebalance}
                    let p1 point to p^.right;
                    IF p1^.balance=1
                       THEN RR rotation transformation
                       ELSE RL rotation transformation;
                    adjust p^.balance=0; h=false
                 END {1}
```

```
          END {CASE}
     END ELSE
     word is found, increase count by 1, h=false
END {search};
```

The process of node insertion consists essentially of the following three parts:

1. Follow the search path until it is found that the key is not already in the tree.
2. Insert the new node and determine the resulting balance factor.
3. Retreat along the search path and check the balance factor at each node.

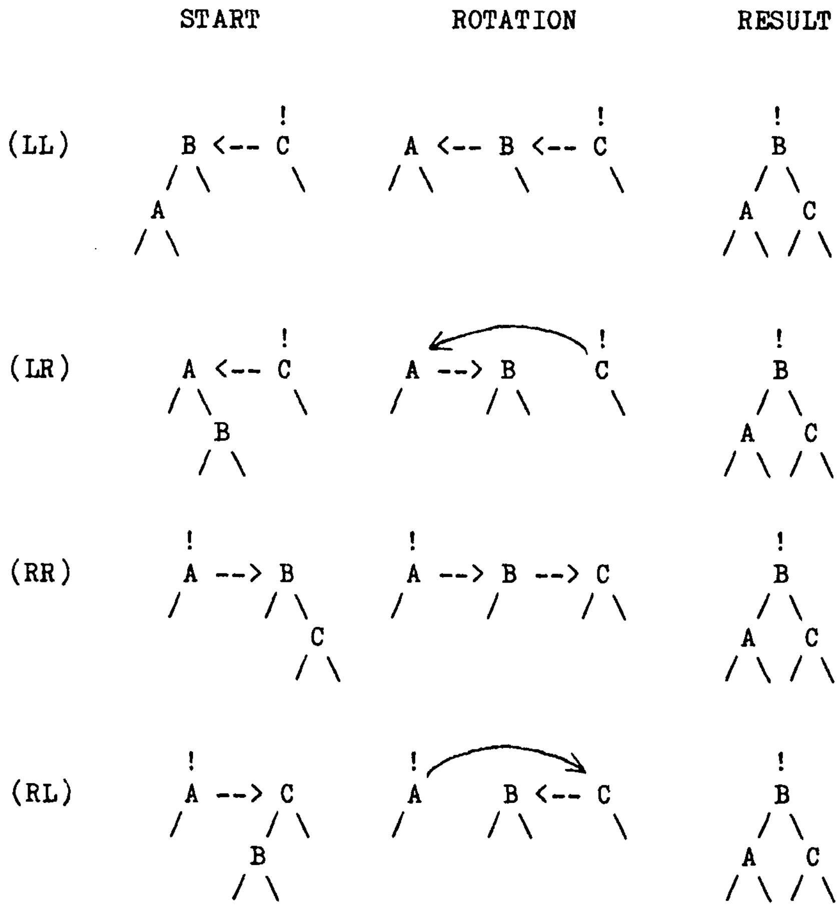
Finding the search path is straightforward, however, if it leads to a dead end (i.e., to an empty subtree designated by a pointer value nil), then the given element must be inserted in the tree at the palce of the empty subtree. Prior to insertion on a left subtree, for example, we must distinguish between the three conditions of a node's balance factor(the height of its right subtree minus the height of its left subtree) involving the subtree heights:

h(left) < h(right) +1, the previous imbalance at p will be equilibrated.

h(left) = h(right) 0, the weight is now slanted to the left.

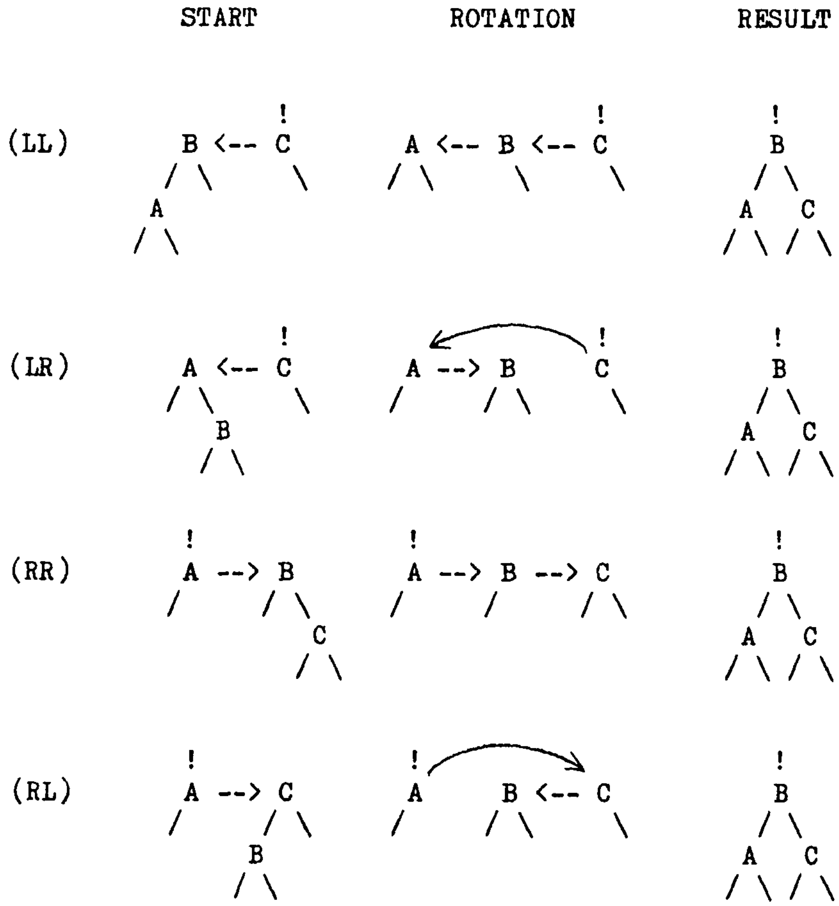
h(left) > h(right) -1, rebalancing is necessary.

The algorithm for insertion and rebalancing critically depends on the way information about the tree's balance is stored. Because of its recursive formulation it can easily accommodate an additional operation on the way back along the search path. At each step, information must be passed as to whether or not the height of the subtree had increased. Therefore, extending the procedure's parameter list by the Boolean h with the meaning that the subtree height has increased. The rebalancing operations necessary are entirely expressed as a sequence of pointer-reassignments. There are four cases that must be considered while rebalancing a tree. These are the rotations indicated in some detail below.



In an extreme case rebalancing may propagate all the way up to the root.

on the way back along the search path. At each step, information must be passed as to whether or not the height of the subtree had increased. Therefore, extending the procedure's parameter list by the Boolean h with the meaning that the subtree height has increased. The rebalancing operations necessary are entirely expressed as a sequence of pointer-reassignments. There are four cases that must be considered while rebalancing a tree. These are the rotations indicated in some detail below.



In an extreme case rebalancing may propagate all the way up to the root.

2.3 Results from Experiment

An experiment was performed in constructing balanced trees using a random number generator to provide data elements. Results are provided in the following table.

Nodes	(miliseconds) Construction Time	(miliseconds) Search Time	Internal Path Length
1,000	324.0	257.0	9,171
2,000	785.0	645.0	20,392
3,000	1,275.0	995.0	32,366
4,000	1,683.0	1,288.0	44,851
5,000	2,184.0	1,811.0	57,729
6,000	2,493.0	2,268.0	70,900
7,000	3,026.0	2,458.0	84,299
8,000	3,861.0	3,172.0	97,896
9,000	4,136.0	3,349.0	111,674
10,000	4,270.0	3,869.0	125,592

HEIGHT	NODES	NODES	NODES	NODES	NODES	NODES	NODES	NODES	NODES	NODES
1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2	2	2
3	4	4	4	4	4	4	4	4	4	4
4	8	8	8	8	8	8	8	8	8	8
5	16	16	16	16	16	16	16	16	16	16
6	32	32	32	32	32	32	32	32	32	32
7	64	64	64	64	64	64	64	64	64	64
8	128	128	128	128	128	128	128	128	128	128
9	245	256	256	256	256	256	256	256	256	256
10	331	487	506	511	512	512	512	512	512	512
11	165	622	862	951	982	1,005	1,015	1,022	1,024	1,024
12	4	357	853	1,244	1,498	1,668	1,788	1,885	1,935	1,973
13		23	261	707	1,224	1,657	2,049	2,369	2,677	2,932
14			7	76	273	634	1,051	1,495	1,943	2,402
15						13	74	206	398	645
16										1
TOTAL	1,000	2,000	3,000	4,000	5,000	6,000	7,000	8,000	9,000	10000

3. B-TREE

3.1 Properties of B-Tree

A B-tree is a data structure defined in an attempt to manage large amounts of information effeciently. In our description we will follow the standard usage and refer to the nodes as pages. In the case of a B-tree we allow a page to contain more than a single item of information. The items of information in each node are referenced by objects called keys and we shall be only concerned with the keys in a page.

A tree is called a B-tree of order n iff

1. Each page contains at most 2n keys.
2. Each page other than the root contains at least n keys.
3. If a page is not a leaf page and contains m keys then it has m+1 children.
4. All leaf pages have the same level.

Figure 11-15 shows the result of construction a B-tree of order 2(n=2) with the following insertion sequence of keys:

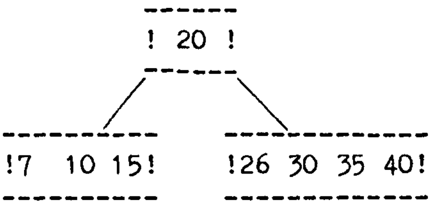
20, 40, 10, 30, 15, 35, 7, 26, 18, 22, 5, 42, 13, 46, 27,
8, 32, 38, 24, 45, 25

```
(1)          --------
             !20    !
             --------

(2)          ----------
             !20  40  !
             ----------
```

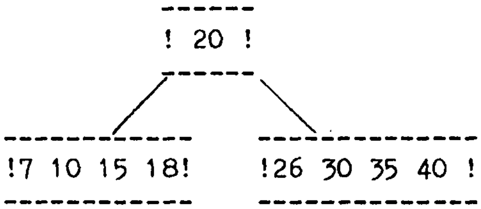
```
(3)          ------------
             ! 10 20 40 !
             ------------

(4)          ----------------
             ! 10 20 30 40  !
             ----------------

(5)          -----------------
             ! 10 15 20 30 40 !   overflow
             ------------------

             ------
             ! 20 !
             ------
              /    \
         --------   --------
         ! 10 15!   !30 40 !
         --------   --------

(6)
              -----
              !20 !
              -----
             /     \
         --------  ----------
         ! 10 15!  !30 35 40!
         --------  ----------

(7)              -----
                 ! 20!
                 -----
                /     \
        ----------   ----------
        !7  10 15!   !30 35 40!
        ----------   ----------
```

Figure 11. Construction a B-tree by insertion 20, 40, 10, 30, 15, 7

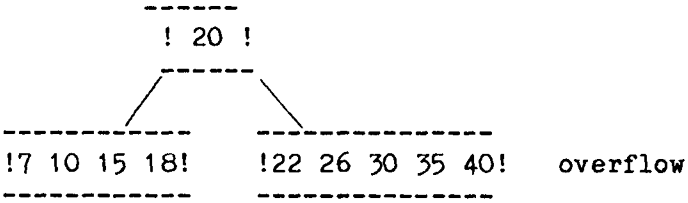
(1)

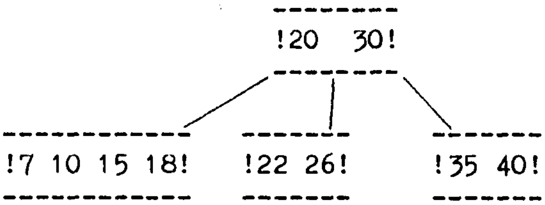

! 20 !
!7 10 15!
!26 30 35 40!

(2)


! 20 !
!7 10 15 18!
!26 30 35 40 !

(3)


! 20 !
!7 10 15 18!
!22 26 30 35 40!
overflow


!20 30!
!7 10 15 18!
!22 26!
!35 40!

(4)

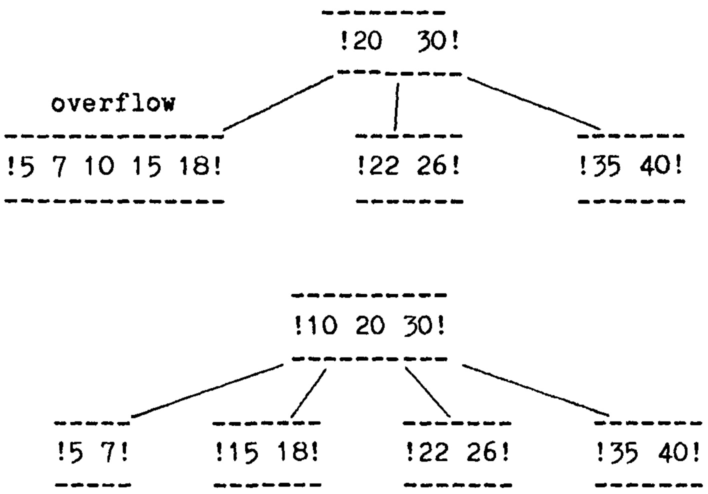
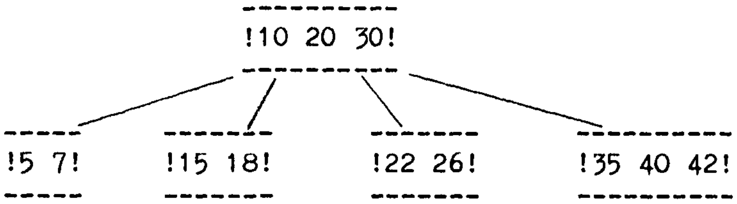
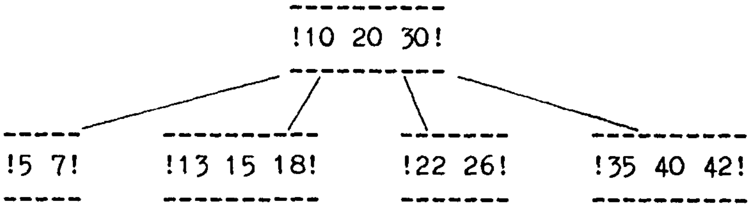


Figure 12. A B-tree after insertion 26, 18, 22, 5

(1)



(2)



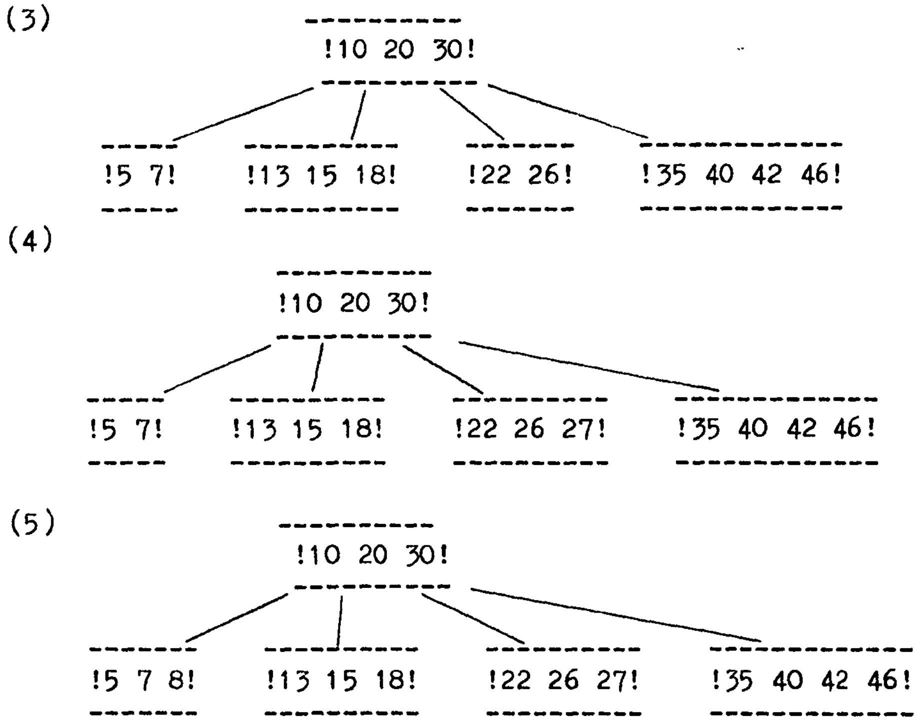
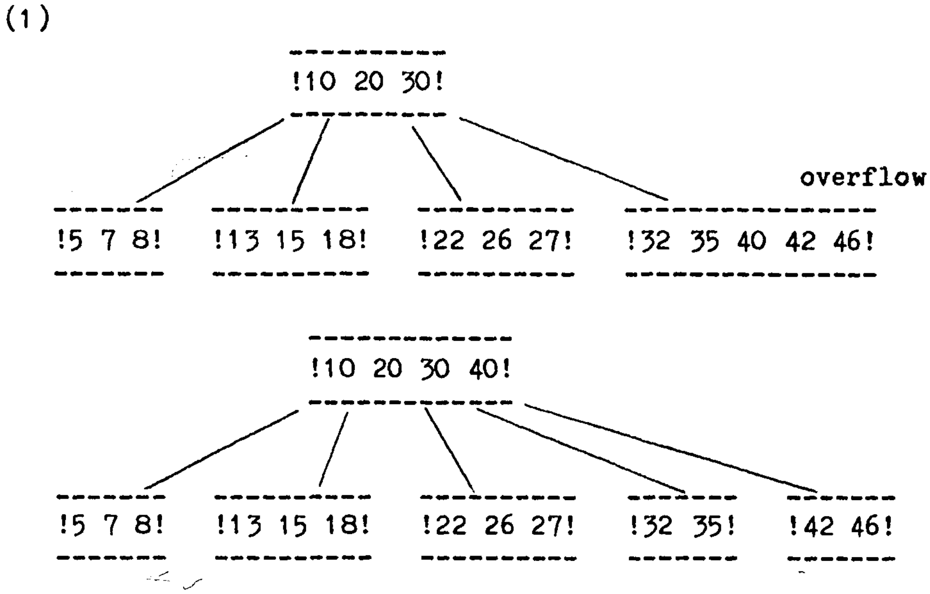


Figure 13. A B-tree after insertion 42, 13, 46, 27, 8



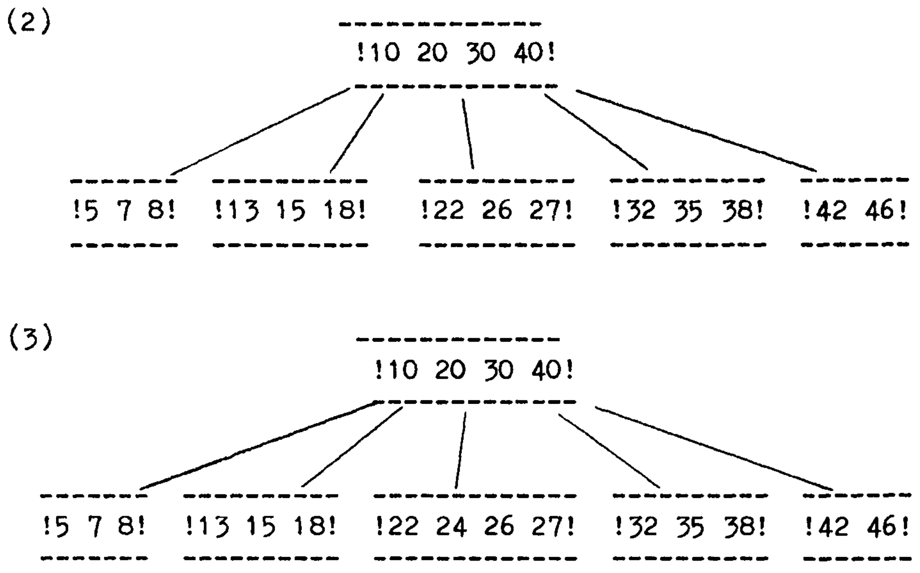
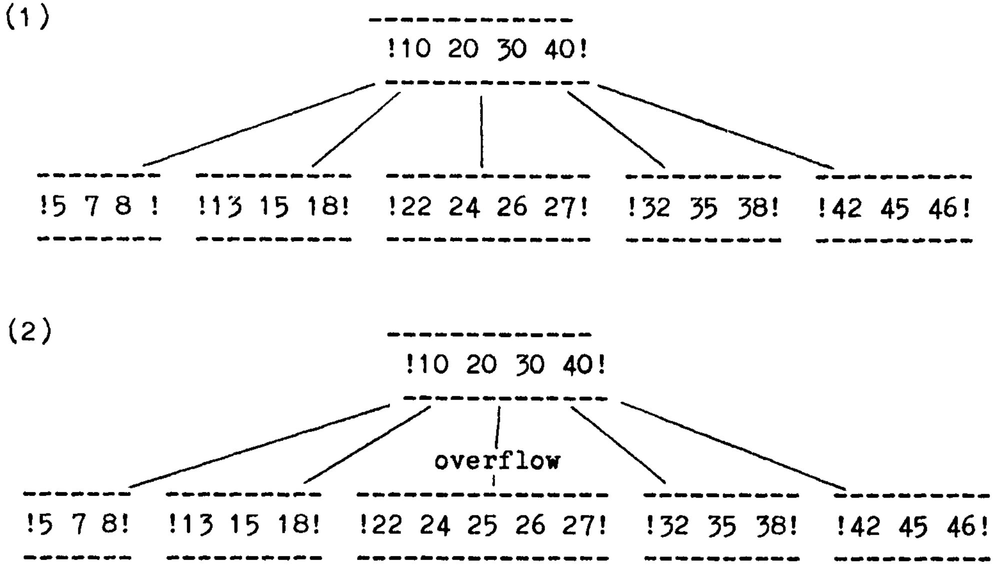


Figure 14. A B-tree after insertion 32, 38, 24



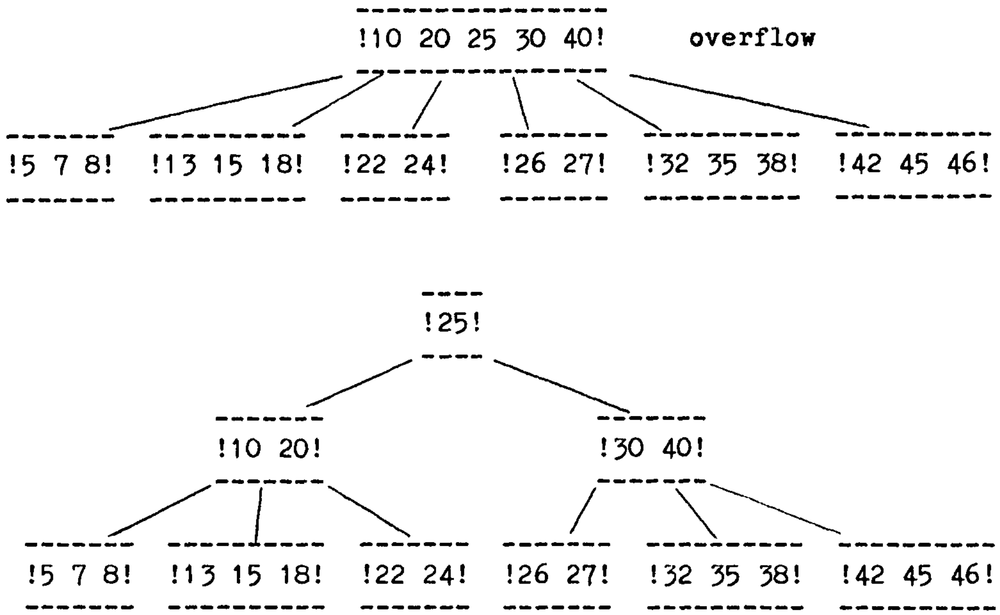


Figure 15. A B-tree after insertion 45, 25

3.2 Analysis of the B-Tree Algorithm

The data structure of page & item :

```
     item= RECORD                   page= RECORD
             key: integer;                  m:0..nn;(*no.of items*)
             p: ref;                        p0: ref;
             count: integer                 e:ARRAY[1..nn] OF item
           END;                           END;
```

```
  PROCEDURE search(x:integer;a:ref;VAR h:boolean;VAR u:item);
  BEGIN
     IF a=NIL THEN BEGIN (* x is not in tree *)
                       Assign x to item u, set h to true,
                       indicating that an item u is passed
                       up in the tree
                   END
               ELSE WITH a^ DO
                   BEGIN (* search x on page a^ *)
                       Binary array search;
                       IF found
                           THEN increment the relevant item's
                                occurrence count
                           ELSE BEGIN
                                     search(x,descendant,h,u);
```

```
                                     IF h THEN (* an item u is
                                                being passed up*)
                                          IF (no.items on a^) < 2n
                                             THEN insert u on page
                                                  a^ and set h to
                                                  false
                                             ELSE split page and
                                                  pass middle item
                                                  up
                                  END
                    END
  END;
```

The keys appear in increasing order from left to right if the B-tree is squeezed into a single level by inserting the descendants in between the keys of their ancestor page. This arrangement represents a natural extension of the organization of binary search trees, and it determines the method of searching an item with given key. Given a page as indicated below where each Ki is a key and each Pi is a page pointer. The in-page search is represented as a binary search upon the fixed array, if the search is unsuccessful, we are in one of the following situations :

1. Ki<x<Ki+1, for 1<=i<m. We continue the search on page Pi^.
2. Km<x. The search continues on page Pm^.
3. x<K1. The search continues on page PO^.

```
  page                  !
 ------------------------------------------
!  PO  K1  P1  K2  P2  ...   Pm-1  Km  Pm  !
 --!-------!-------!----!-----!---------!--
   !       !       !    !     !         !
   v       v       v    v     v         v
```

If an item is to be inserted in a page with m < 2n items, the insertion process is restricted to that page. It is only insertion into an already full page which may cause the allocation of new pages by page splitting. In particular, the split pages contain exactly n items. The insertion of an item in the ancestor page may again cause that page to overflow, thereby causing splitting to propagate. A recursive formulation is most convenient because of the property of the splitting process to propagate back along the search path.

3.3 Results from Experiment

Nodes	(miliseconds) Construction Time	(miliseconds) Search Time	Internal Path Length
1,000	563.0	452.0	4,623
2,000	1,458.0	1,116.0	11,272
3,000	1,985.0	1,708.0	16,883
4,000	2,735.0	2,229.0	22,543
5,000	3,289.0	3,217.0	33,159
6,000	4,487.0	3,645.0	39,770
7,000	4,864.0	4,404.0	46,426
8,000	5,925.0	5,360.0	53,084
9,000	6,812.0	5,788.0	59,670
10,000	8,046.0	6,364.0	66,326

HEIGHT	NODES	NODES	NODES	NODES	NODES	NODES	NODES	NODES	NODES	NODES
1	4	2	3	4	1	1	2	2	2	3
2	16	7	12	15	5	7	8	9	11	11
3	54	28	44	53	17	24	26	30	34	36
4	205	106	164	212	74	87	100	109	130	141
5	721	394	594	794	266	321	365	416	480	533
6		1,463	2,183	2,922	988	1,190	1,388	1,580	1,777	1,968
7					3,649	4,370	5,111	5,854	6,566	7,308
TOTAL	1,000	2,000	3,000	4,000	5,000	6,000	7,000	8,000	9,000	10000

4. SYMMETRIC BINARY B-TREE (SBB TREE)

4.1 Properties of SBB Tree

A symmetric binary B-tree is a B-tree which is also a binary tree. In order to compensate for the pages in the B-tree horizontal pointers are introduced to represent page items which would be considered to be all at the same level. The following conditions are required. Every node contains one key and at most two(pointers to) subtrees. Every pointer is either horizontal or vertical. There are no two consecutive horizontal pointers on any search path. All terminal nodes appear at the same level. Figure 16, and 17 show how one might construct a SBB tree with the following insertion sequences of keys:

4. SYMMETRIC BINARY B-TREE (SBB TREE)

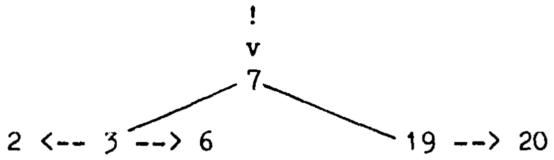
4.1 Properties of SBB Tree

A symmetric binary B-tree is a B-tree which is also a binary tree. In order to compensate for the pages in the B-tree horizontal pointers are introduced to represent page items which would be considered to be all at the same level. The following conditions are required. Every node contains one key and at most two(pointers to) subtrees. Every pointer is either horizontal or vertical. There are no two consecutive horizontal pointers on any search path. All terminal nodes appear at the same level. Figure 16, and 17 show how one might construct a SBB tree with the following insertion sequences of keys:

correction

```
(a) 19, 7, 3, 2, 20, 6, 9

(1)                     !
                        v
                        19


(2)
                        !
                        v
                  7 <-- 19


(3)
                        !
                        v
           3 <--- 7 <-- 19      two consecutive
                               siblings


                         !
                         v
                         7
                        / \
                      3   19
(4)                      !
                         v
                         7
                        / \
               2 <--- 3   19


(5)                      !
                         v
                         7
                        / \
                2 <-- 3   19 --> 20
```

(6)



(7)

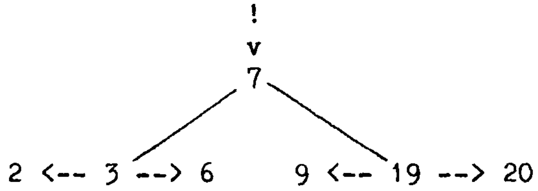
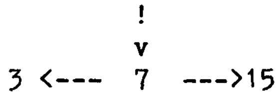


Figure 16. The development of SBB trees with insertion sequence of (a)

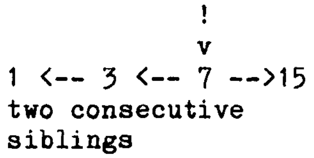
(b) 7, 3, 15, 1, 19, 6, 9

(1)

(2)



(3)



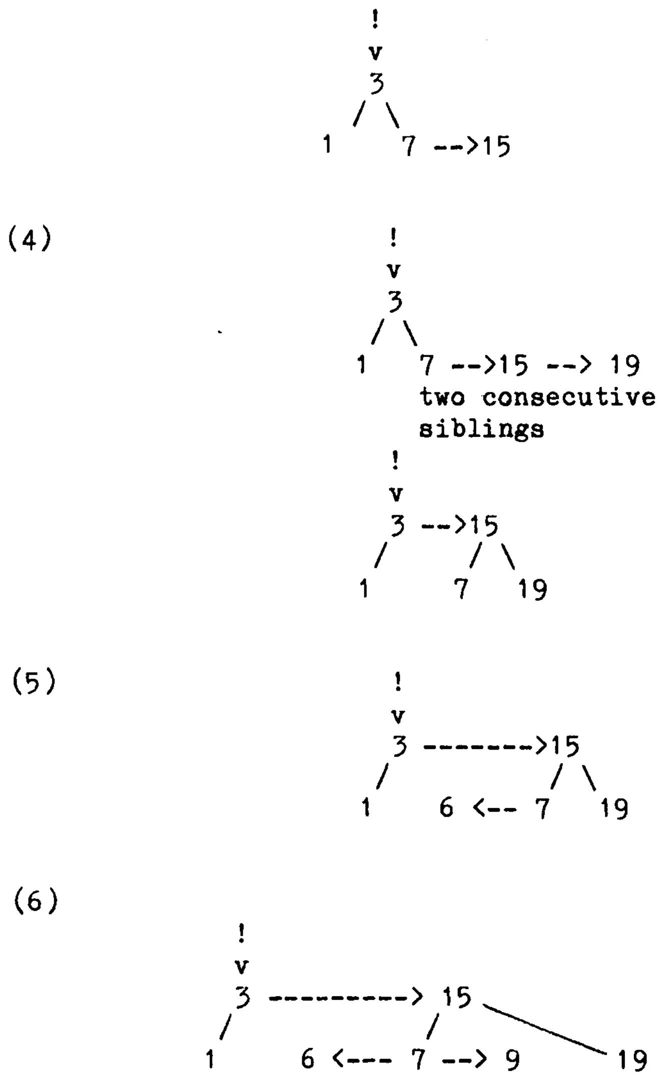


Figure 17. The development of SBB trees with insertion sequence of (b)

4.2 Analysis of the SBB Tree Algorithm

Data structure of the SBB tree:

```
TYPE      ref =^node;
          node= RECORD
                   key:integer;
                   count:integer;
```

```
                         left,right:ref;
                         lh,rh:boolean
                      END;

PROCEDURE search(x:integer;VAR p:ref; VAR h:integer);
VAR p1,p2: ref;
BEGIN
     IF p=nil THEN
     BEGIN
         word is not in tree, insert it;
         set h=2, lh, rh= false
     END ELSE
     IF x<p^.key THEN
     BEGIN
         search(x,p^.left,h);
         IF h <> 0 THEN {need rotation to rebalance}
         IF p^.lh {p has obtained a left sibling}
         THEN BEGIN
                  let p1 point to p^.left;
                  IF p1^.lh {p1 has obtained a left sibling}
                     THEN LL rotation transformation
                     ELSE IF p1^.rh {p1 has obtained a right
                                     sibling}
                             THEN LR rotation transformation
              END
         ELSE {h=0}
         BEGIN
            h=h-1;
            IF h <> 0 {p has obtained a left sibling}
            THEN p^.lh=true
         END
     END ELSE
     IF x>p^.key THEN
     BEGIN
          search(x,p^.right,h);
          IF h <> 0 THEN {need rotation to rebalance}
          IF p^.rh {p has obtained a right sibling}
          THEN BEGIN
                   let p1 point to p^.right;
                   IF p1^.right {p1 has obtained a right
                                 sibling}
                     THEN RR rotation transformation
                     ELSE IF p1^.lh {p1 has obtained a left
                                     sibling}
                             THEN RL rotation transformation
                END
            ELSE {h=0}
            BEGIN
```

```
              h:= h-1;
              IF h <> 0 {p has obtained a right sibling}
                  THEN p^.rh=true
          END
    END ELSE
    word is found in tree; increase count by 1;
    set h=0
    END {search};
```

The recursive procedure search again follows the pattern of the basic binary tree insertion algorithm. Each node now requires two bits(Boolean variables lh and rh) to indicate the nature of its two pointers. Whenever a subtree of node A without siblings grows, the root of the subtree becomes the sibling of A. The parameter h indicates whether or not the subtree with root P has changed, and it corresponds directly to the parameter h of the B-tree search program. We must distinguish between the case of a subtree(indicated by a vertical pointer) that has grown and a sibling node(indicated by a horizontal pointer) that has obtained another sibling and hence requires a page split. The problem is solved by introducing a three-valued h with the following meanings:

1. h=0 : the subtree P requires no changes of the tree structure.
2. h=1 : node P has obtained a sibling.
3. h=2 : the subtree P has increased in height.

The actions to be taken for node re-arrangement are virtually identical to those of the balanced binary tree algorithm. The implementation of the three-valued balance field(-1, 0, 1), in the case of the balanced binary tree, is replaced by three boolean fields lh, rh in the case of the SBB tree. In fact, the four cases (LL, RR, LR, RL) in the SBB tree algorithm are slightly simpler than in the balanced binary tree algorithm. It can be shown that the AVL-balanced tree is a special case of the SBB tree. [1]

4.3 Results from Experiment

Nodes	(miliseconds) Construction Time	(miliseconds) Search Time	Internal Path Length
1,000	368.0	298.0	9,358
2,000	862.0	677.0	20,772
3,000	1,258.0	902.0	32,832
4,000	1,782.0	1,269.0	45,381
5,000	2,562.0	1,952.0	58,615
6,000	3,018.0	2,436.0	72,098
7,000	3,269.0	2,853.0	87,447
8,000	3,545.0	3,103.0	100,836
9,000	4,162.0	3,231.0	115,140
10,000	4,965.0	4,096.0	129,554

HEIGHT	NODES	NODES	NODES	NOSES	NODES	NODES	NODES	NODES	NODES	NODES
1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2	2	2
3	4	4	4	4	4	4	4	4	4	4
4	8	8	8	8	8	8	8	8	8	8
5	16	16	16	16	16	16	16	16	16	16
6	32	32	32	32	32	32	32	32	32	32
7	64	64	64	64	64	64	64	64	64	64
8	128	128	128	128	128	128	128	128	128	128
9	219	256	256	256	256	256	256	256	256	256
10	277	451	508	512	512	512	512	512	512	512
11	173	529	774	912	932	972	936	962	982	989
12	67	336	718	1,054	1,274	1,473	1,332	1,527	1,606	1,677
13	9	122	349	712	1,099	1,371	1,426	1,809	2,028	2,189
14		37	116	258	529	790	1,233	1,469	1,740	2,054
15		14	20	40	125	313	730	839	1,010	1,275
16			4	1	16	51	250	275	435	557
17					2	7	60	83	153	174
18							10	13	23	52
19										10
TOTAL	1,000	2,000	3,000	4,000	5,000	6,000	7,000	8,000	9,000	10000

5. 2-3 TREE

5.1 Property of the 2-3 Tree

A 2-3 tree is a tree in which each vertex which is not a leaf has 2 or 3 children, and every path from the root to a leaf is of the same length. Nodes which are not leaf nodes contain the maximum value of the leftmost and the maximum value of the next child. Note that the tree consisting of a single vertex is a 2-3 tree. Figure 18 shows a sample 2-3 tree. Figure 19 shows the sample 2-3 tree after inserting 2. Figure 20 shows the sample 2-3 tree after inserting 7.

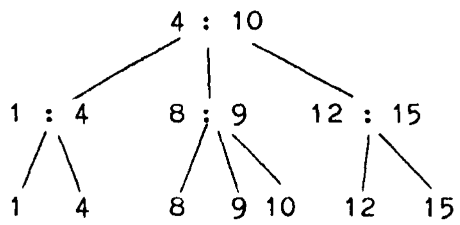


Figure 18. A sample 2-3 tree

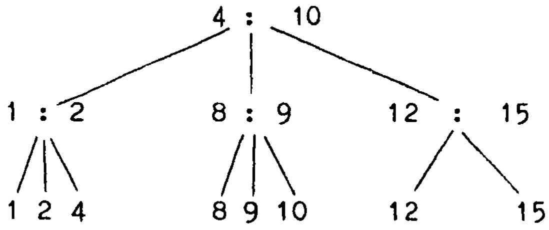


Figure 19. The sample 2-3 tree after inserting 2

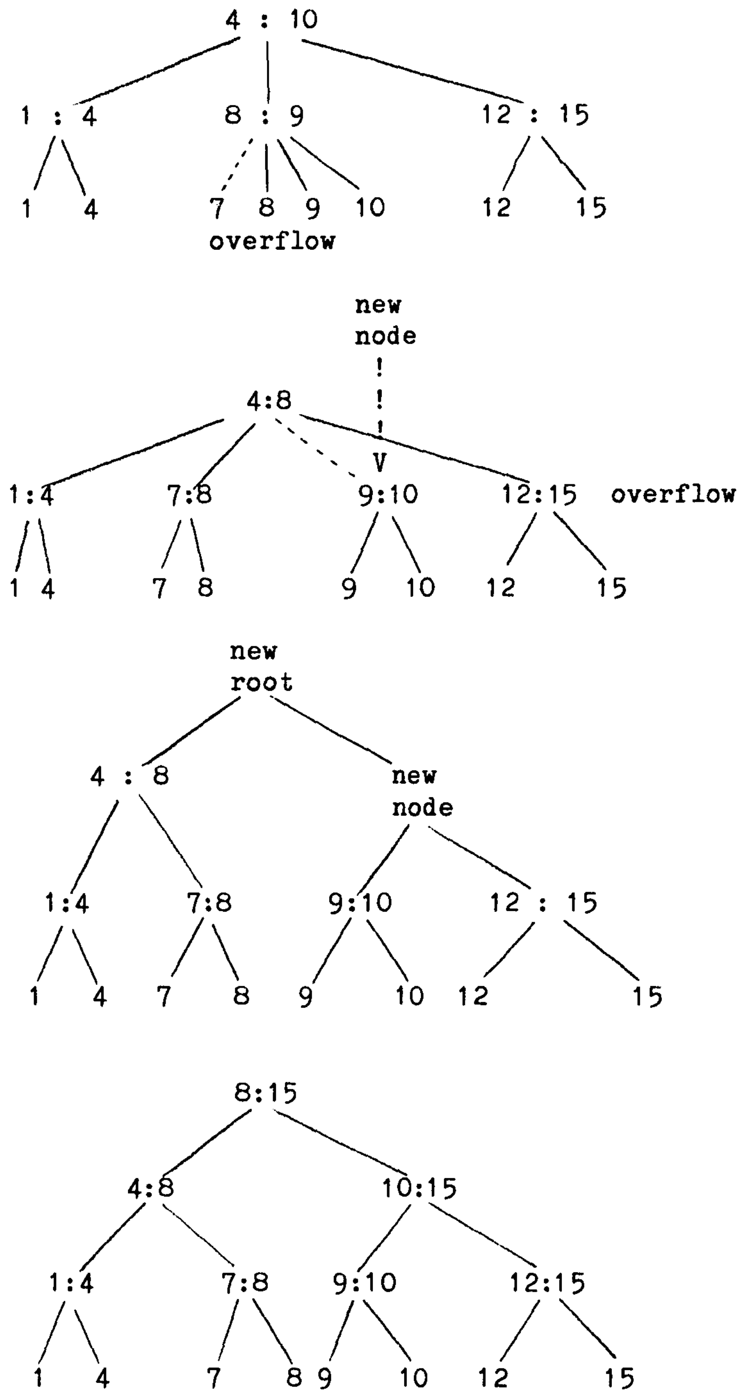


Figure 20. The sample 2-3 tree after inserting 7

5.2 Analysis of the 2-3 Tree Algorithm

To build a 2-3 tree is essentially similar to construct a binary tree, except a 2-3 tree may have 3-way decisions making because of its property that each node(except leaf) has two or three children. Therefore, we need two pieces of information (L and M) store in each node(except leaf) to indicate the direction while inserting a new element. L[v], the element L stored at node v, indicates the largest element of the subtree whose root is the left most son of node v; M[v], the element M stored at node v, indicates the largest element to the subtree whose root is the second son of node v. With these information stored in the nodes, we can search a 2-3 tree analogous to binary tree search.

To insert a new element e into a 2-3 tree, we must locate the place for the new leaf l that will contain element e. This is done by tracing the paths that are indicated by the value of L[v] and M[v]. If e<=L[v], then follow the path of the left most son of the node v. If L[v]<e<=M[v], then follow the path of the second son of node v. If none of the above cases are true, then follow the path of the right most son of the node v. Until an empty node is found, then we create a new leaf l and insert element e into leaf l, then insert l into tree.

Suppose node r already has three leaves, s1, s2, and s3. We want to insert a new leaf l to be the appropriate son of node r. First, we must re-arrange the sequence of these four children by comparing the value of each of them. To maintain the 2-3 property, we create a new node n, and assign the two right most sons as the two left most sons of n, keep the two left most sons of node r unchange. At this point, we have to up date the value of L and M that had been stored in the ancestors of node r, then we can insert node n into the tree and becomes a child of father of r. If f, father of r, had three children, we must repeat this procedure recursively until all ancestors in the tree has at most three children. Suppose f is the root of the tree, and had three children, r1, r2, r3 plus a newly created node n (assume n>r3). In this case, we create another new node n' to keep r3 and n, and create a new root to keep f and n'.

```
Procedure SEARCH(a,r);
IF any son of r is a leaf THEN RETURE r
ELSE BEGIN
        let si be the ith son of r;
        IF a<=L[r] THEN RETURE SEARCH(a,s1)
                        IF r has less than 3 children
                           THEN adjust L, M
                           ELSE addson(r)
        ELSE
             IF r has two sons or a<=M[r] THEN RETURN SEARCH(a,s2)
                                               IF r has less than
                                               3 children
                                                  THEN adjust L, M
                                                  ELSE addson(r)
             ELSE RETURN SEARCH(a,s3)
                  IF r has less than 3 children
                     THEN adjust L, M
                     ELSE addson(r)
     END


Procedure ADDSON(v);
BEGIN
     create a new vertex v';
     make the two rightmost sons of v the left and right sons of v';
     IF v is not in the leaf level
        THEN adjust L, M
```

```
      IF v has no father THEN
        BEGIN
            create a new root r;
            make v the left son and v' the right son of r
            adjust L, M
        END
      ELSE
        BEGIN
            let f be the father of v;
            make v' a son of f immediately to the right of v;
            adjust L, M
            IF f now has four sons THEN ADDSON(f)
        END
END ;
```

5.3 Results from Experiment

Nodes	(miliseconds) Construction Time	(miliseconds) Search Time	Internal Path Length
1,000	1,337.0	262.0	8,000
2,000	3,205.0	623.0	18,000
3,000	4,989.0	919.0	27,000
4,000	7,526.0	1,389.0	40,000
5,000	9,547.0	1,709.0	50,000
6,000	11,773.0	2,053.0	60,000
7,000	14,515.0	2,446.0	70,000
8,000	16,720.0	2,940.0	88,000
9,000	19,335.0	3,580.0	99,000
10,000	------	-----	----

HEIGHT	NODES
8	1,000
9	2,000
9	3,000
10	4,000
10	5,000
10	6,000
10	7,000
11	8,000
11	9,000
----	10,000

6. SON TREE

6.1 Properties of Son Tree

A binary tree is called a 1-2 tree iff each inner node is either unary or binary, and all leaves are on the same level. A 1-2 tree is called a son tree if no two unary nodes are sucessive. The son tree we discuss here is one kind of B tree. Each page contains one or two nodes, the maximum sons of each page are three, each node can have either one or two sons. The node in the page may be an empty (E) node.

```
            !
 -----------------------
!     node    node      !
 ---!-------!-------!---
    !       !       !
    v       v       v
```

Figure 23-26 shows the result of construction a son tree with the following insertion sequence of keys:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11

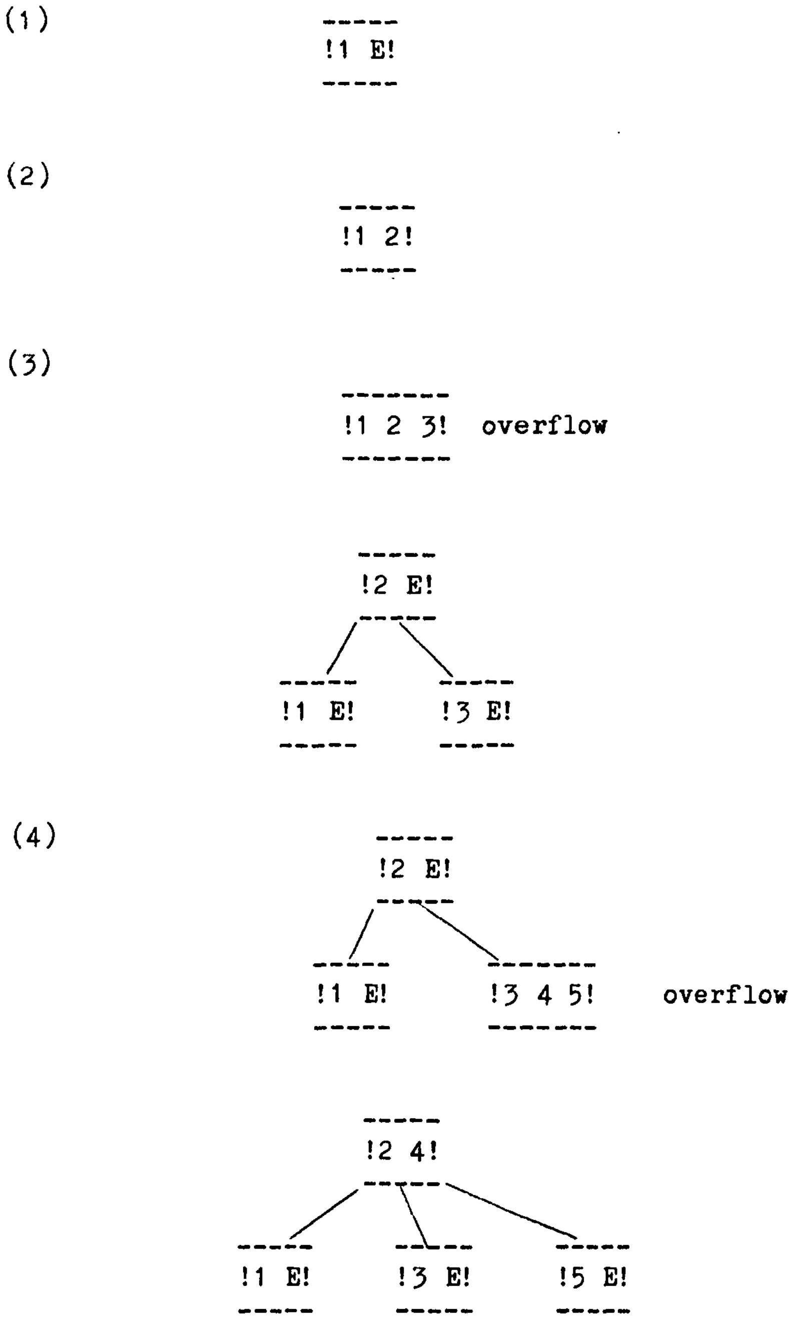
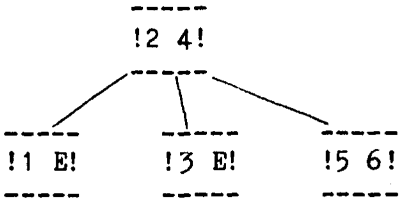
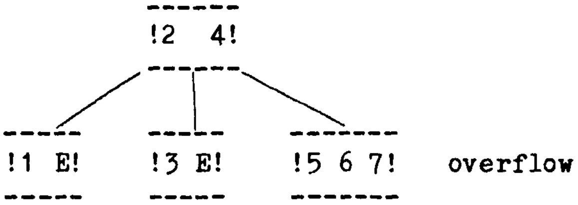


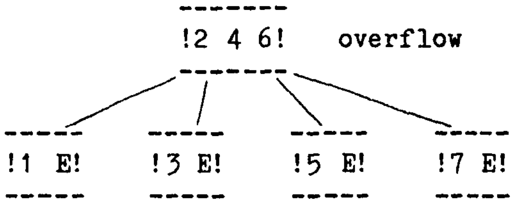
Figure 21. Construction a son tree of B-tree type by insertion 1, 2, 3, 4, 5

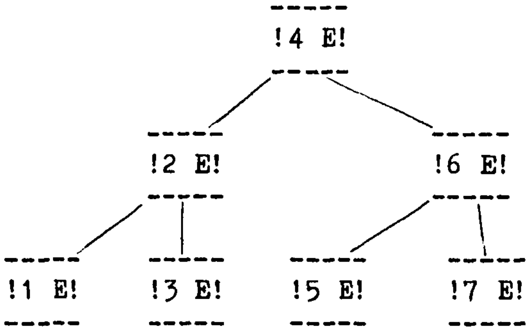
(1)


!2 4!
!1 E!
!3 E!
!5 6!

(2)


!2 4!
!1 E!
!3 E!
!5 6 7!
overflow


!2 4 6!
overflow
!1 E!
!3 E!
!5 E!
!7 E!


!4 E!
!2 E!
!6 E!
!1 E!
!3 E!
!5 E!
!7 E!

(3)

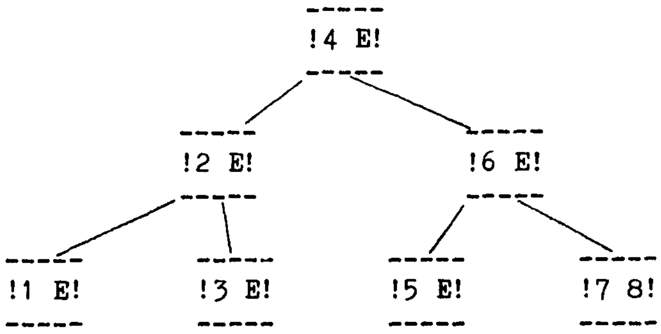
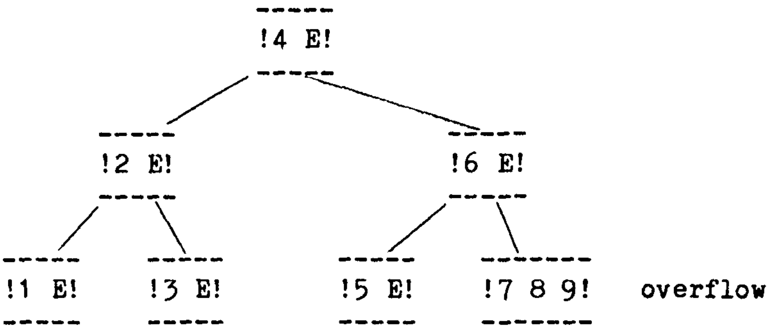
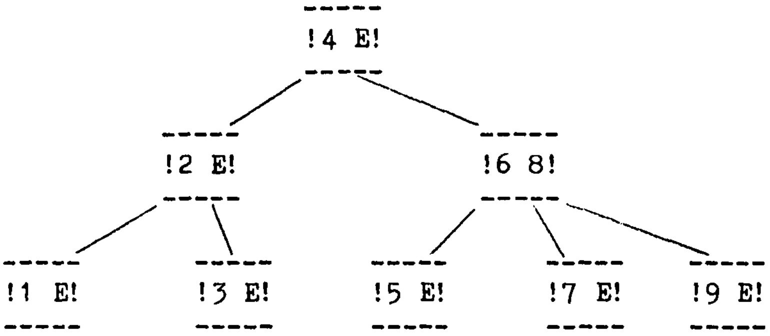


Figure 22. A son tree of B-tree type after insertion
6, 7, 8

(1)





(2)

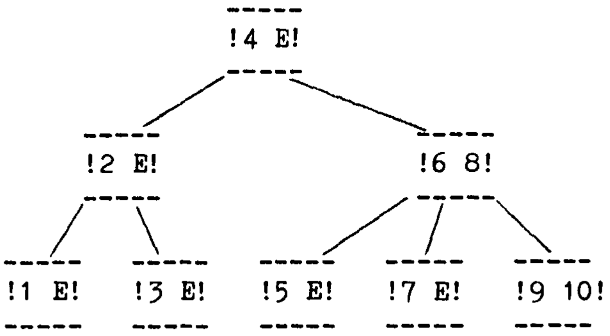
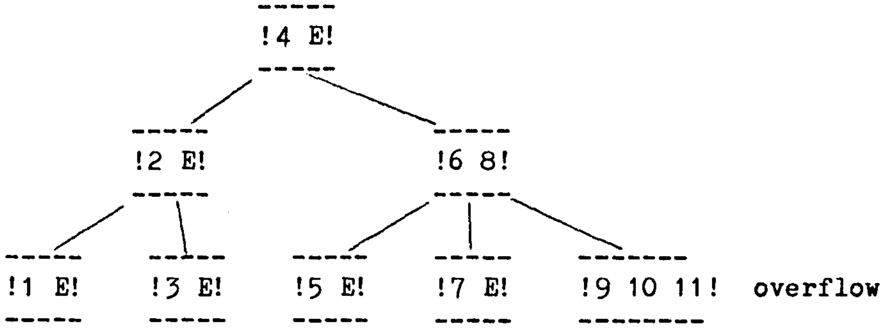
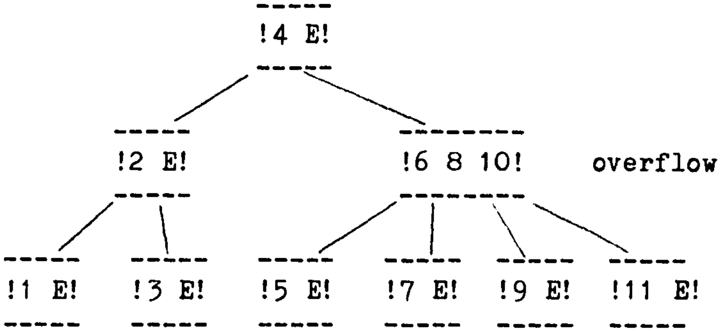


Figure 23. A son tree of B-tree type after insertion
9, 10





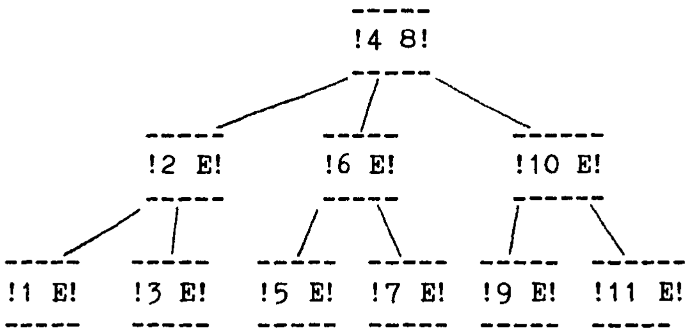


Figure 24. A son tree of B-tree type after insertion 11

6.2 Analysis of the Son Tree Algorithm

The data structure of page & item :

```
item= RECORD                    page= RECORD
        key: integer;                  m:1.. 2;(*no.of items*)
        p: ref;                        p0: ref;
        count: integer                 e:ARRAY[1.. 2] OF item
      END;                           END;
```

```
PROCEDURE search(x:integer;a:ref;VAR h:boolean;VAR u:item);
BEGIN
   IF a=NIL THEN BEGIN (* x is not in tree *)
                     Assign x to item u, set h to true,
                     indicating that an item u is passed
                     up in the tree
                 END
            ELSE WITH a^ DO
                 BEGIN (* search x on page a^ *)
                     Binary array search;
                     IF found
                         THEN increment the relevant item's
                              occurrence count
                         ELSE BEGIN
                                   search(x,descendant,h,u);
                                   IF h THEN (* an item u is
                                              being passed up*)
                                       IF (no.items on a^) < 2
                                          THEN insert u on page
                                               a^ and set h to
                                               false
```

```
                                           ELSE split page and
                                                pass middle item
                                                up
                                END
                   END
  END;
```

The procedure search is straightward binary tree search, searching key x with root a, if found, increment counter, otherwise insert an item with key x and count 1 in tree. If an item emerges to passed to a lower level, this may cause that page to overflow, then, the procedure insert will call by the recursive formulation (parameter h=true) back along the search path.

6.3 Results from Experiment

Nodes	(miliseconds) Construction Time	(miliseconds) Search Time	Internal Path Length
1,000	642.0	535.0	7,258
2,000	1,547.0	1,321.0	16,504
3,000	2,512.0	2,067.0	27,777
4,000	3,195.0	2,757.0	37,034
5,000	4,612.0	3,793.0	46,302
6,000	5,211.0	4,352.0	55,536
7,000	6,159.0	5,620.0	71,770
8,000	6,999.0	6,472.0	82,056
9,000	8,036.0	6,569.0	92,347
10,000	9,805.0	8,479.0	102,577

HEIGHT	NODES	NODES	NODES	NODES	NODES	NODES	NODES	NODES	NODES	NODES
1	2	1	1	1	2	2	1	1	1	1
2	3	3	2	3	3	4	2	3	3	3
3	9	6	5	5	7	10	5	5	5	5
4	19	16	11	13	15	22	11	11	12	15
5	44	38	25	32	41	49	24	27	31	33
6	106	91	55	78	91	109	58	65	69	78
7	245	214	130	181	224	269	135	155	175	194
8	572	498	308	415	525	626	311	354	392	439
9		1,133	746	975	1,226	1,472	726	831	938	1,042
10			1,717	2,297	2,866	3,437	1,726	1,959	2,209	2,476

11							4,001	4,589	5,165	5,714
TOTAL	1,000	2,000	3,000	4,000	5,000	6,000	7,000	8,000	9,000	10000

7. COMPARISONS OF DATA MAINTENANCE

CONSTRUCTION TIME : (miliseconds)

Nodes	Balanced Binary Tree	SBB Tree	B Tree	Son Tree	2-3 Tree
1,000	324.0	368.0	563.0	642.0	1,337.0
2,000	785.0	862.0	1,458.0	1,547.0	3,205.0
3,000	1,275.0	1,258.0	1,985.0	2,512.0	4,989.0
4,000	1,683.0	1,782.0	2,735.0	3,195.0	7,526.0
5,000	2,184.0	2,562.0	3,289.0	4,612.0	9,547.0
6,000	2,493.0	3,018.0	4,487.0	5,211.0	11,773.0
7,000	3,026.0	3,269.0	4,864.0	6,159.0	14,515.0
8,000	3,861.0	3,545.0	5,925.0	6,999.0	16,720.0
9,000	4,136.0	4,162.0	6,812.0	8,036.0	19,335.0
10,000	4,270.0	4,965.0	8,046.0	9,805.0	----

SEARCH TIME : (miliseconds)

Nodes	2-3 Tree	Balanced Binary Tree	SBB Tree	B Tree	Son Tree
1,000	262.0	257.0	298.0	452.0	535.0
2,000	623.0	645.0	677.0	1,116.0	1,321.0
3,000	919.0	995.0	902.0	1,708.0	2,067.0
4,000	1,389.0	1,288.0	1,269.0	2,229.0	2,757.0
5,000	1,709.0	1,811.0	1,952.0	3,217.0	3,793.0
6,000	2,053.0	2,268.0	2,436.0	3,645.0	4,352.0
7,000	2,446.0	2,458.0	2,853.0	4,404.0	5,620.0
8,000	2,940.0	3,172.0	3,103.0	5,360.0	6,472.0
9,000	3,580.0	3,349.0	3,231.0	5,788.0	6,569.0
10,000	---	3,869.0	4,096.0	6,364.0	8,479.0

INTERNAL PATH LENGTH :

Nodes	B Tree	2-3 Tree	SON Tree	Balanced Binary Tree	SBB Tree
1,000	4,623	8,000	7,258	9,171	9,358
2,000	11,272	18,000	16,504	20,392	20,772
3,000	16,883	27,000	27,777	32,366	32,832
4,000	22,543	40,000	37,034	44,851	45,381
5,000	33,159	50,000	46,302	57,729	58,615
6,000	39,770	60,000	55,536	70,900	72,098
7,000	46,426	70,000	71,770	84,299	87,447
8,000	53,084	88,000	82,056	97,896	100,836
9,000	59,670	99,000	92,347	111,674	115,140
10,000	66,326	---	102,577	125,592	129,554

8. CONCLUSION

From the data that obtained by comparing five tree structures, we can conclude that Balanced Binary tree has the most efficient searching structure. In the decreasing order of efficiency, follows by SBB tree, 2-3 tree, B tree and Son tree. When using recursive formulation search in the balanced tree structures(we exclude B tree and Son tree here, because they involve in-page binary search also). The difference of search time and construction time between Balanced Binary tree and SBB tree are very small, because they have similar strutures. B tree and Son tree have shorter internal path lengths but they consume more search time and construction time, because they have in-page binary search within recursive formulation search.

The Balanced Binary tree can be considered a well constructed tree structure for the general purposes of storing and searching, because it has the fastest search time and the construction time comparing it to other tree structures for very large numbers of nodes. The 2-3 tree may be used for special purposes where insertion is not an important time constraint. It has the slowest construction time. However it has fairly efficient speed on search time. The Son tree has the unique character of filling empty nodes into the tree structure, thereby, it delays the search time and construction time. The B tree may be used for large scale data bank in which insertions and deletions are necessary, but in which the primary storage of a computer is not large enough or is too costly to be used for long-time storage.

REFERENCES

1. N. Wirth, Algorithms + Data Structures = Programs (Prentice- Hall, Englewood Cliffs, N.J, 1976).
2. Data Structures for Set Manipulation Problems.
3. H. Olivie, Information Processing Letters (Feb. 1980).
4. E. Horowitz & S. Sahni, Fundamentals of Computer Algorithms, (Computer Science Press, Inc. 1978).
5. H. Lorin, Sorting and Sort Systems (Addison-Wesley Publishing Company, 1975).
6. A. Tenenbaum & M. Augenstein, Data Structures Using Pascal (Prentice-Hall, Englewood Cliffs, N.J. 1981).

VITA

John Chun-Hua Chen was born in Taiwan, Republic of China on November 1, 1950. He graduated from University of San Francisco in June 1976 with a B.S. degree in accounting specialist. From 1977 to 1979, he worked in a public C.P.A. firm as an external auditor. In January 1981, he obtained a M.A. degree in accounting.

In May 1981, the author entered the Computing and Information Science of Lehigh University as a graduate student. Since then, he worked under Professor S.L. Gulden in the field of the system programming language.