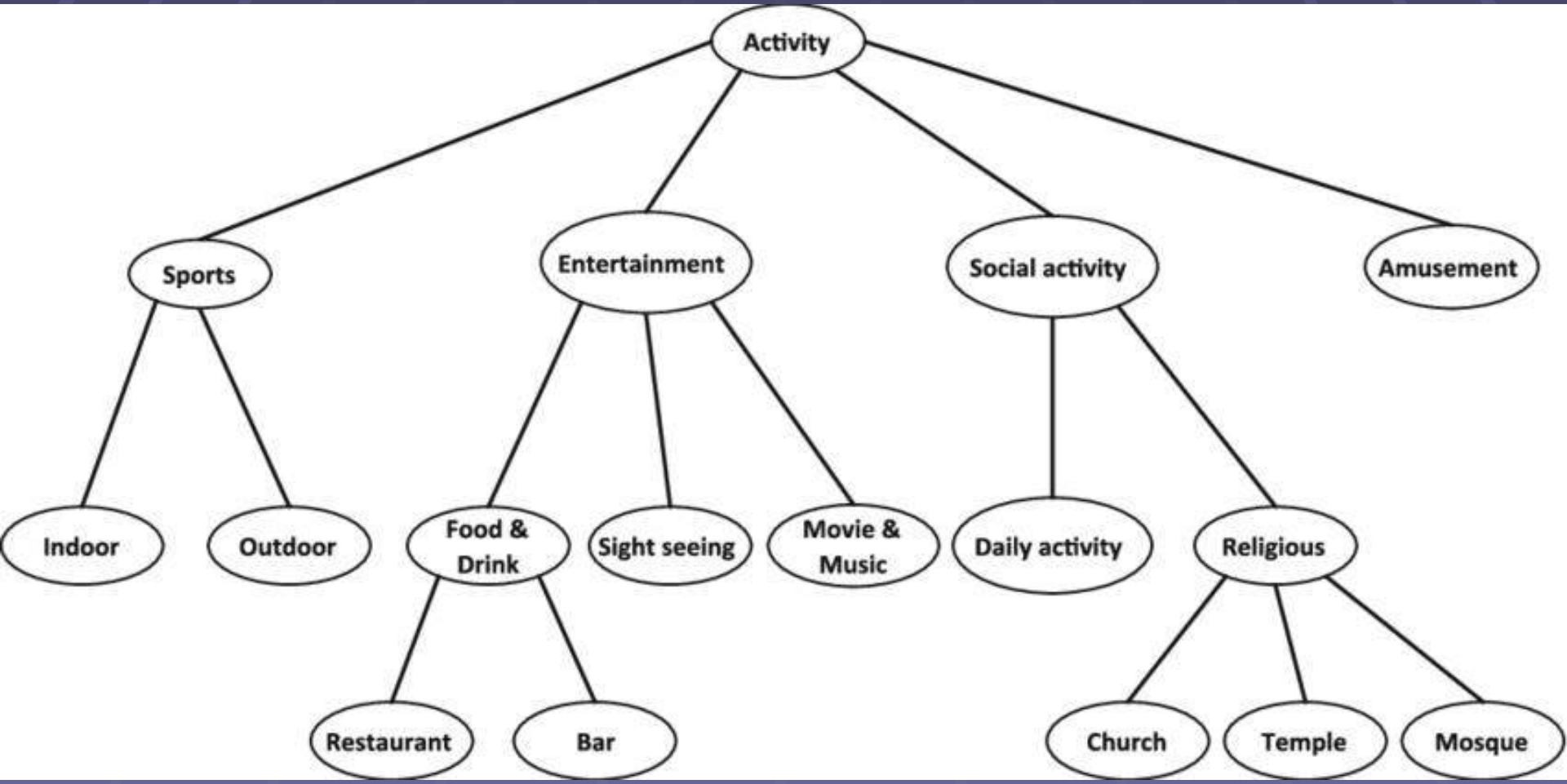
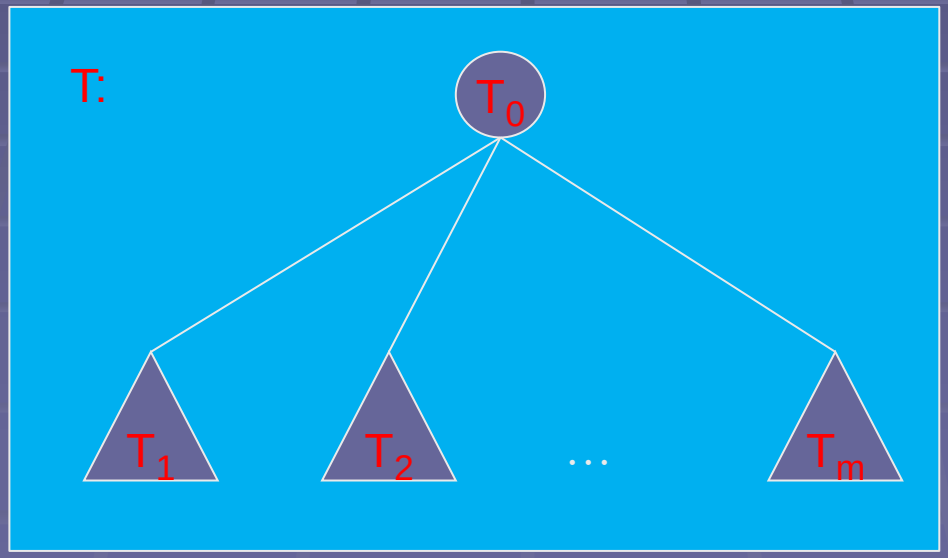
Tree

- A ***tree*** structure means that the data are organized so that items of information are related by branches



Introduction

- **Definition** (recursively): A *tree* is a finite set of one or more nodes such that
  - There is a specially designated node called *root*.
  - The remaining nodes are partitioned into n>=0 disjoint set $T_1,\ldots,T_n$, where each of these sets is a tree. $T_1,\ldots,T_n$ are called the *subtrees* of the root T_0.
- Every node in the tree is the root of some subtree



Introduction

- Some **Terminology**
  - *node*: the item of information plus the branches to each node.
  - *degree*: the number of subtrees of a node
  - *degree* of a tree: the maximum of the degree of the nodes in the tree.
  - *terminal* nodes (or *leaf*): nodes that have degree zero
  - *nonterminal* nodes: nodes that don't belong to terminal nodes.
  - *children*: the roots of the subtrees of a node X are the *children* of X
  - *parent*: X is the *parent* of its children.

Introduction

- Some **Terminology** (cont'd)
  - *siblings*: children of the same parent are said to be siblings.
  - *Ancestors* of a node: all the nodes along the path from the root to that node.
  - The *level* of a node(L): defined by letting the root be at level zero. If a node is at level *l*, then it children are at level *l+1*.
  - *Depth(D):* The depth of a node N is given as the number of edges in the path from the root R to the node N. The depth of the root node is zero.
  - *Height(H):* The height of a node N is given as the number of edges in the longest path from the node N to a leaf.
  - For overall tree, D=H=L but for a node N, D=L but D may not be equal to H.

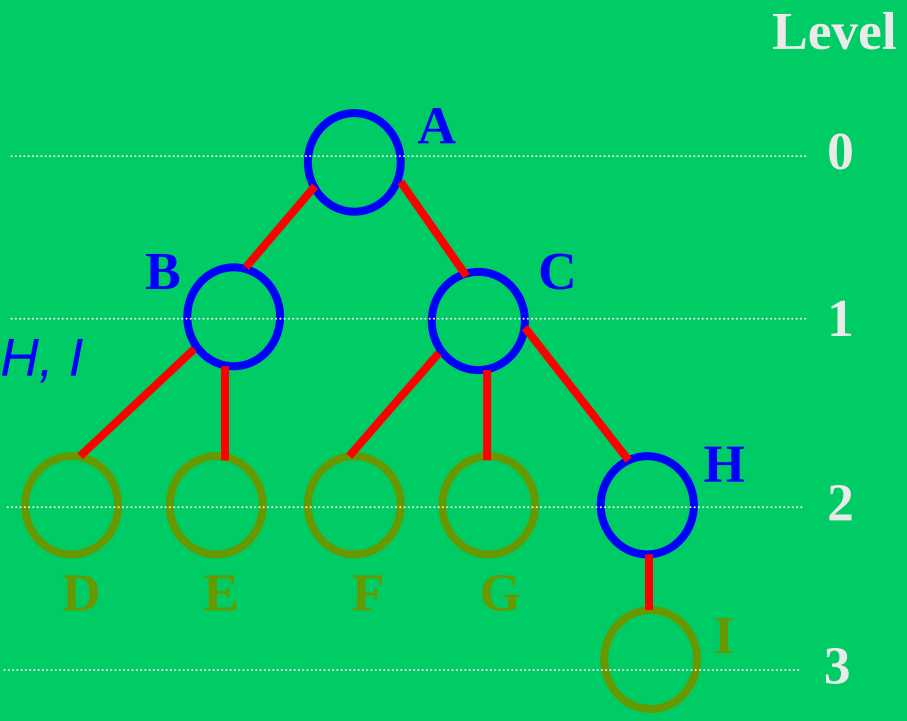
Introduction

- Another notation followed as
  - The *level of a node*: defined by letting the root be at level one. If a node is at level L, then it children are at level L+1.
  - *D = L = H*
- *Another notation followed as*
  - *Height*: the total number of nodes on the path from the root node to the deepest node in the tree. A tree with only a root node has a height of 1. Level of root is 0 and hence height = level + 1. This is followed in this lecture.

Introduction

- Example
  - *A* is the *root* node
  - *B* is the *parent* of D and E
  - *C* is the *sibling* of B
  - *D* and *E* are the *children* of B
  - *D, E, F, G, I* are *external nodes*, or *leaves*
  - *A, B, C, H* are *internal nodes*
  - The *level* of *E* is *2*
  - The *height (depth)* of the tree is *4*
  - The *degree* of node *B* is *2*
  - The *degree* of the tree is *3*
  - The *ancestors* of node *I* is *A, C, H*
  - The *descendants* of node *C* is *F, G, H, I*

Property: *(# edge*s) = (*#node*s) - 1



Introduction

- Representation Of Trees
  - List Representation
    - we can write of Figure 5.2 as a list in which each of the subtrees is also a list

(A (B (E (K, L), F), C (G), D (H (M), I, J)))

    - The root comes first, followed by a list of sub-trees

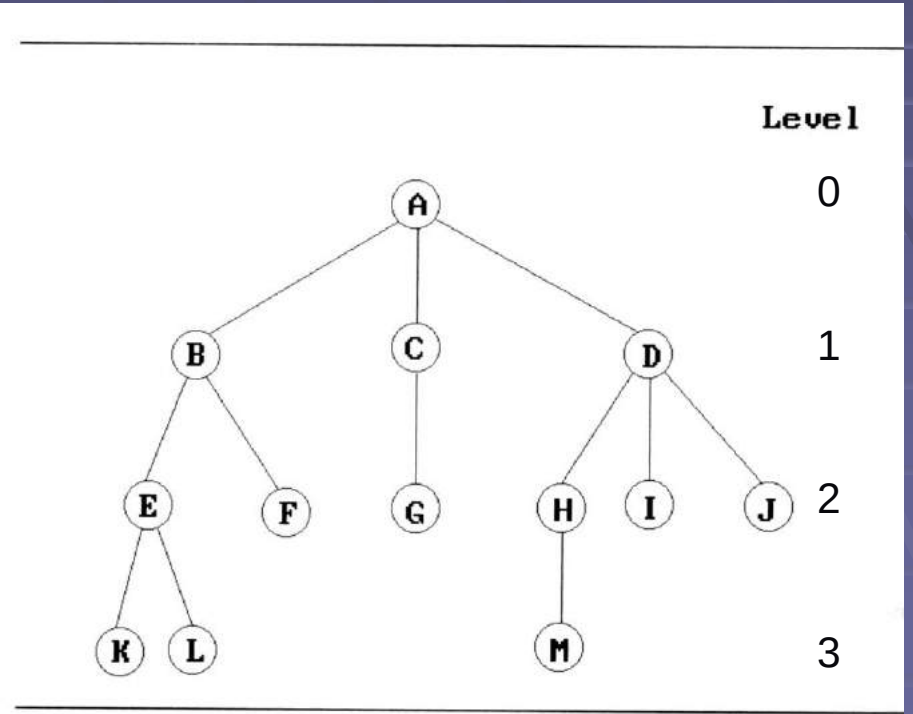


Figure 5.2: A sample tree

Binary Trees

- Binary trees are characterized by the fact that any node can have at most two branches
- **Definition** (recursive):
  - A *binary tree* is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the left subtree and the right subtree
- Thus the left subtree and the right subtree are distinguished

- Any tree can be transformed into binary tree
  - by left child-right sibling representation

Binary Trees

- The abstract data type of binary tree

---

structure *Binary_Tree* (abbreviated *BinTree*) is

objects: a finite set of nodes either empty or consisting of a root node, left *Binary_Tree*, and right *Binary_Tree*.

functions:

for all *bt*, *bt*1, *bt*2 ∈ *BinTree*, *item* ∈ *element*

BinTree Create()	::=	creates an empty binary tree
Boolean IsEmpty(*bt*)	::=	**if** (*bt* == empty binary tree) **return** *TRUE* **else return** *FALSE*
BinTree MakeBT(*bt*1, *item*, *bt*2)	::=	**return** a binary tree whose left subtree is *bt*1, whose right subtree is *bt*2, and whose root node contains the data *item*.
BinTree Lchild(*bt*)	::=	**if** (IsEmpty(*bt*)) **return** error **else return** the left subtree of *bt*.
element Data(*bt*)	::=	**if** (IsEmpty(*bt*)) **return** error **else return** the data in the root node of *bt*.
BinTree Rchild(*bt*)	::=	**if** (IsEmpty(*bt*)) **return** error **else return** the right subtree of *bt*.

Structure 5.1: Abstract data type *Binary_Tree*

Binary Trees

- Properties of binary trees
  - [***Maximum number of nodes***]:

1. The maximum number of nodes on level *i* of a binary tree is 2^i, $i \geq 0$.
2. The maximum number of nodes in a binary tree of height *k* is 2^k-1, $k \geq 1$.

  - [***Relation between number of leaf nodes and degree-2 nodes***]:

    For any nonempty binary tree, T, if n_0 is the number of leaf nodes and n_2 is the number of nodes of degree 2, then $n_0 = n_2 + 1$.

- These lemmas allow us to define full and complete binary trees

Binary Trees

- Definition:
  - A *full binary tree* of height *k* is a binary tree of death *k* having 2^k-1 nodes, $k \geq 0$.
  - A binary tree with n nodes and height k is complete *iff* its nodes correspond to the nodes numbered from 1 to n in the full binary tree of height k.
  - The height of a complete binary tree with *n* nodes is $\lceil \log_2(n+1) \rceil$

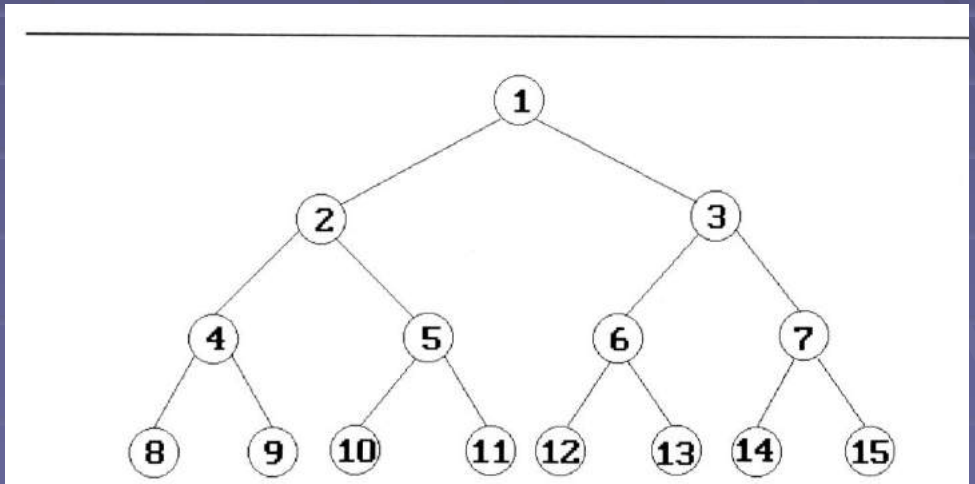


Figure 5.10: Full binary tree of depth 4 with sequential node numbers

Binary Trees

- Two special kinds of binary trees:
  (a) *skewed tree*, (b) *complete binary tree*
  - The all leaf nodes of these trees are on two adjacent levels

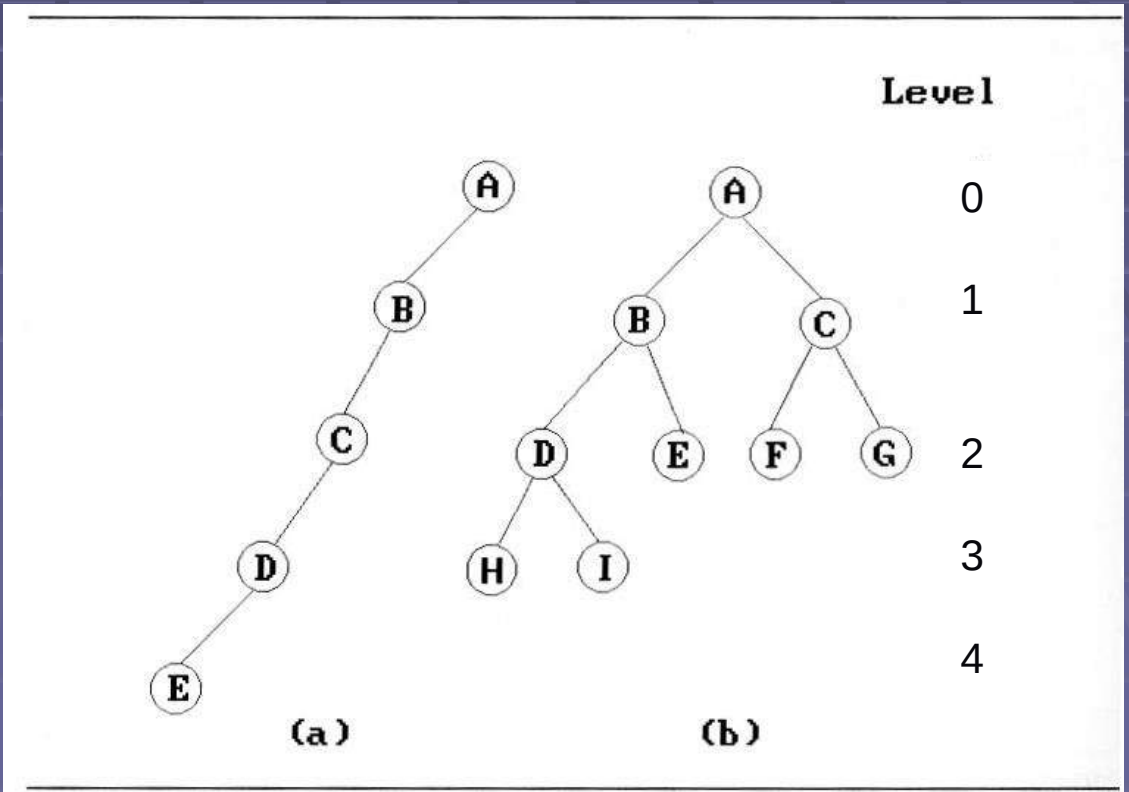
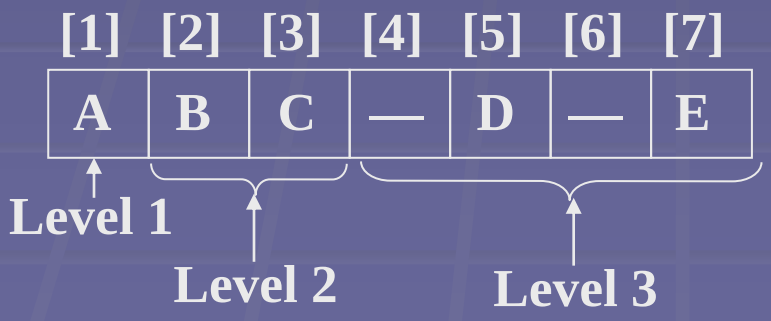


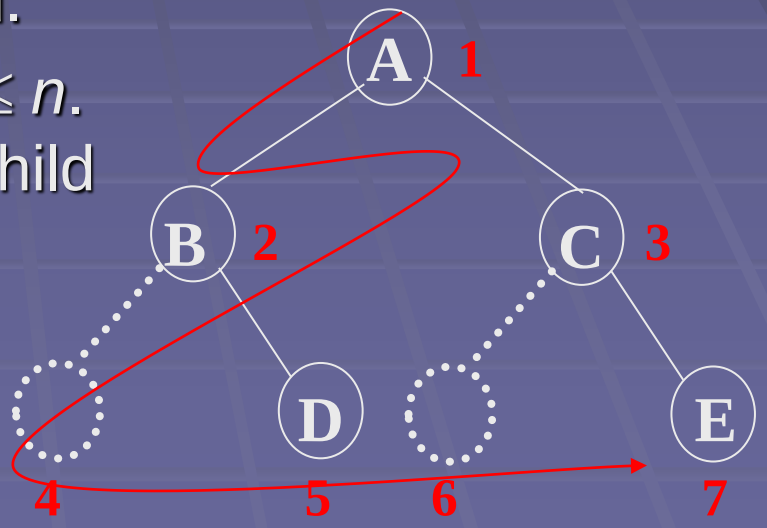
Figure 5.9: Skewed and complete binary trees

Binary Trees

- Binary tree representations (using array)
  - **Lemma 5.3:** If a complete binary tree with n nodes is represented sequentially, then for any node with index i, $1 \le i \le n$, we have

1. *parent(i)* is at $\lfloor i/2 \rfloor$ if $i \neq 1$.
   If $i = 1$, i is at the root and has no parent.
2. *LeftChild(i)* is at $2i$ if $2i \le n$.
   If $2i > n$, then i has no left child.
3. *RightChild(i)* is at $2i+1$ if $2i+1 \le n$.
   If $2i+1 > n$, then i has no left child





Binary Trees

- Binary tree representations (using array)
  - Waste spaces: in the worst case, a skewed tree of height *k* requires 2^k-1 spaces. Of these, only *k* spaces will be occupied
  - Insertion or deletion of nodes from the middle of a tree requires the movement of potentially many nodes to reflect the change in the level of these nodes

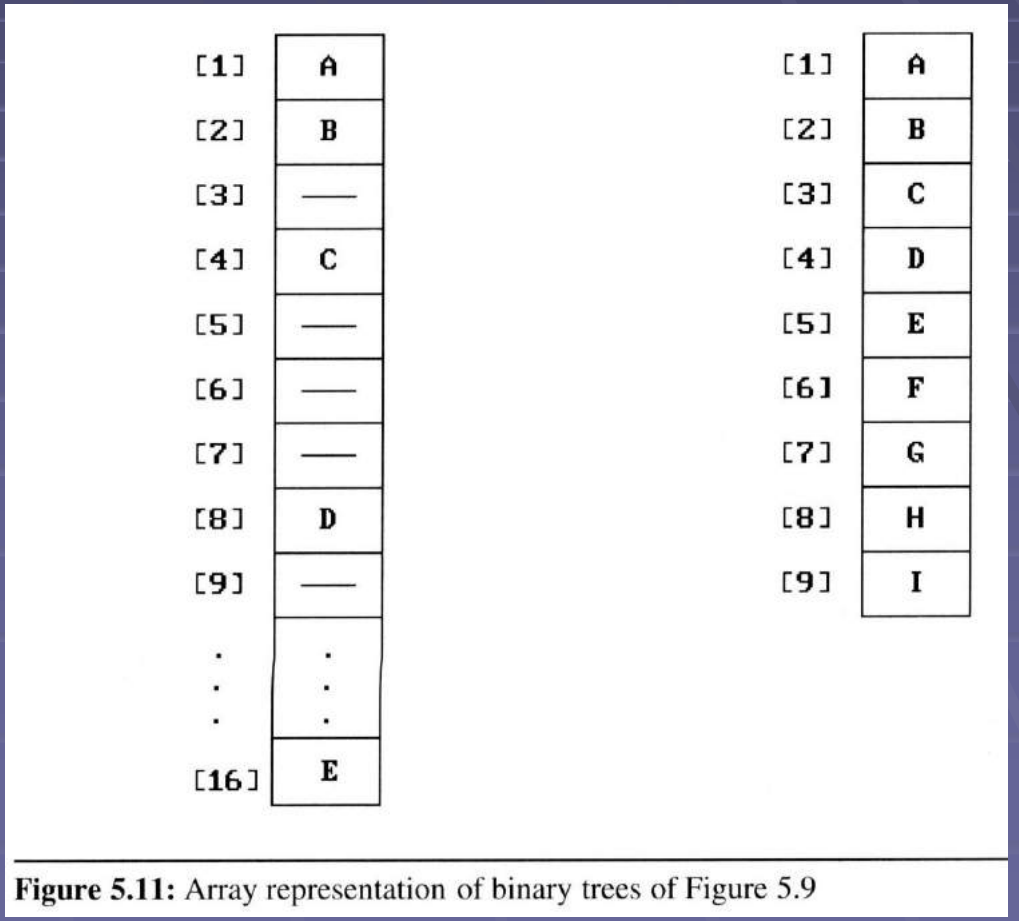
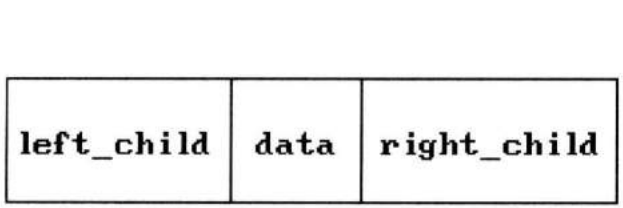


Figure 5.11: Array representation of binary trees of Figure 5.9

Binary Trees

- Binary tree representations (using link)

```
typedef struct node *tree_pointer;
typedef struct node {
        int data;
        tree_pointer left_child, right_child;
        };
```



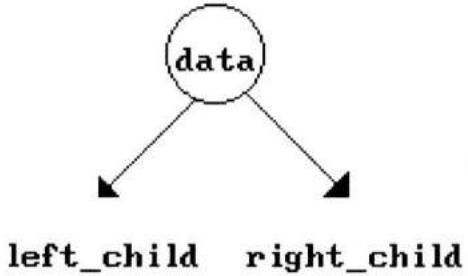


Figure 5.12: Node representation for binary trees

Create a binary tree

```
struct node *mktree(void)
{
 int c;
Struct node *np;
c=getchar();
if (c != '#')
{ np = (struct node *) malloc(sizeof(struct node));
np →data = c;
np → left_child = mktree();
np → right_child = mktree();
return np;
}
else
return NULL;
}
```

Binary Trees

- Binary tree representations (using link)

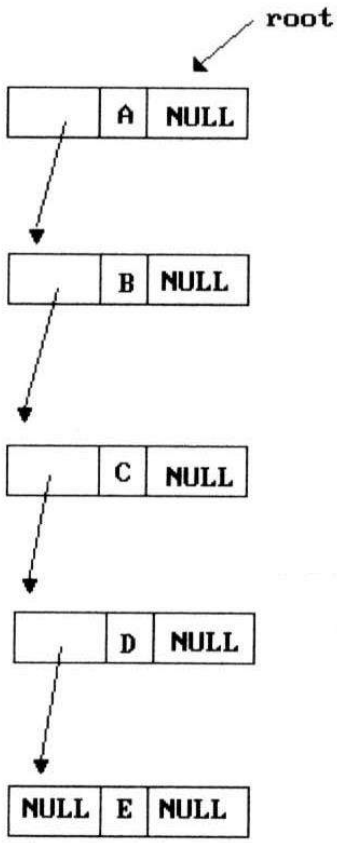
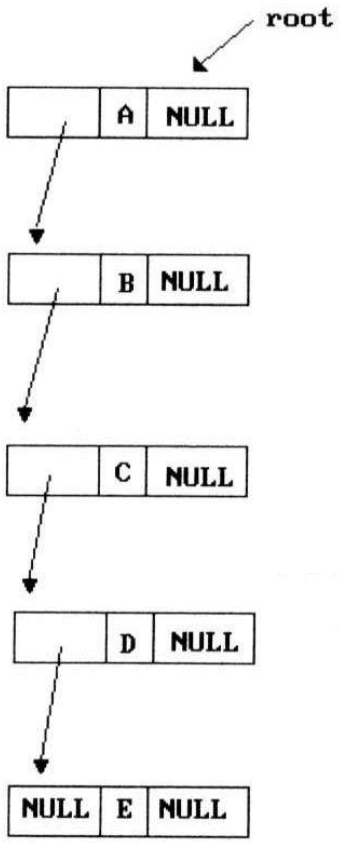


Figure 5.13: Linked representation for the binary trees of Figure 5.9

Binary Trees

- Binary tree representations (using link)



(A (B (C (D (E (#) (#)) (#)) (#)) (#)) (#))

Figure 5.13: Linked representation for the binary trees of Figure 5.9

Binary Trees

- Binary tree representations (using link)

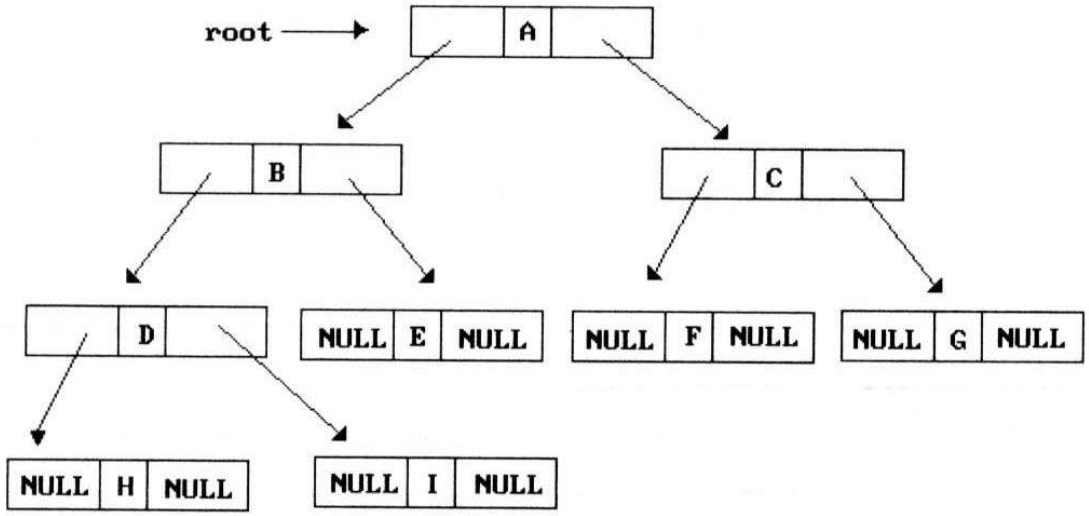
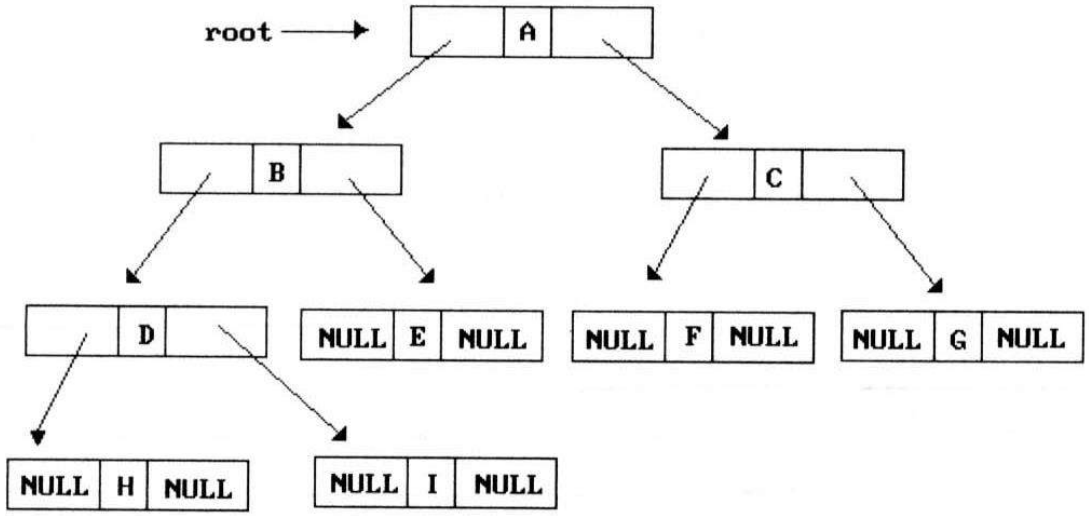


Figure 5.13: Linked representation for the binary trees of Figure 5.9

Binary Trees

- Binary tree representations (using link)



(A (B (D (H (#) (#)) (I (#)(#))) (E (#)(#))) (C (F (#)(#)) (G (#)(#))))

Figure 5.13: Linked representation for the binary trees of Figure 5.9

Any tree to Binary tree

- Left Child-
  Right Sibling
  Representation

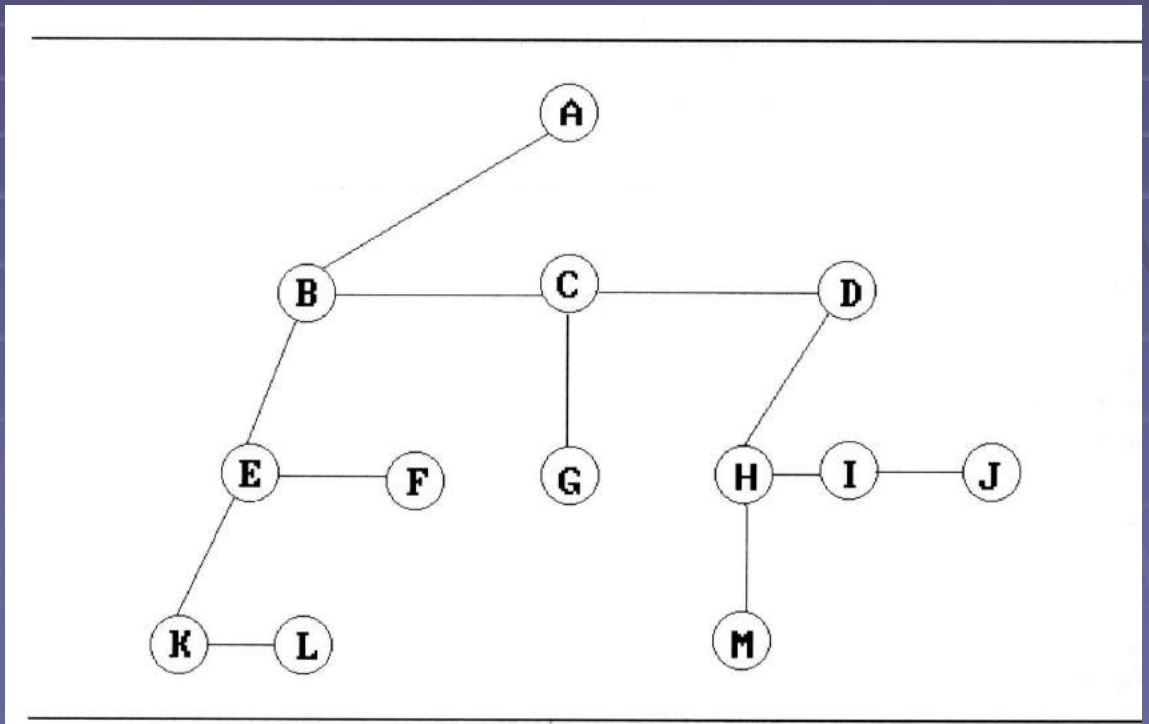


Figure 5.5: Left child-right sibling representation of a tree

Any tree to Binary tree

- Representation Of Trees (cont'd)
  - Representation As A Degree Two Tree

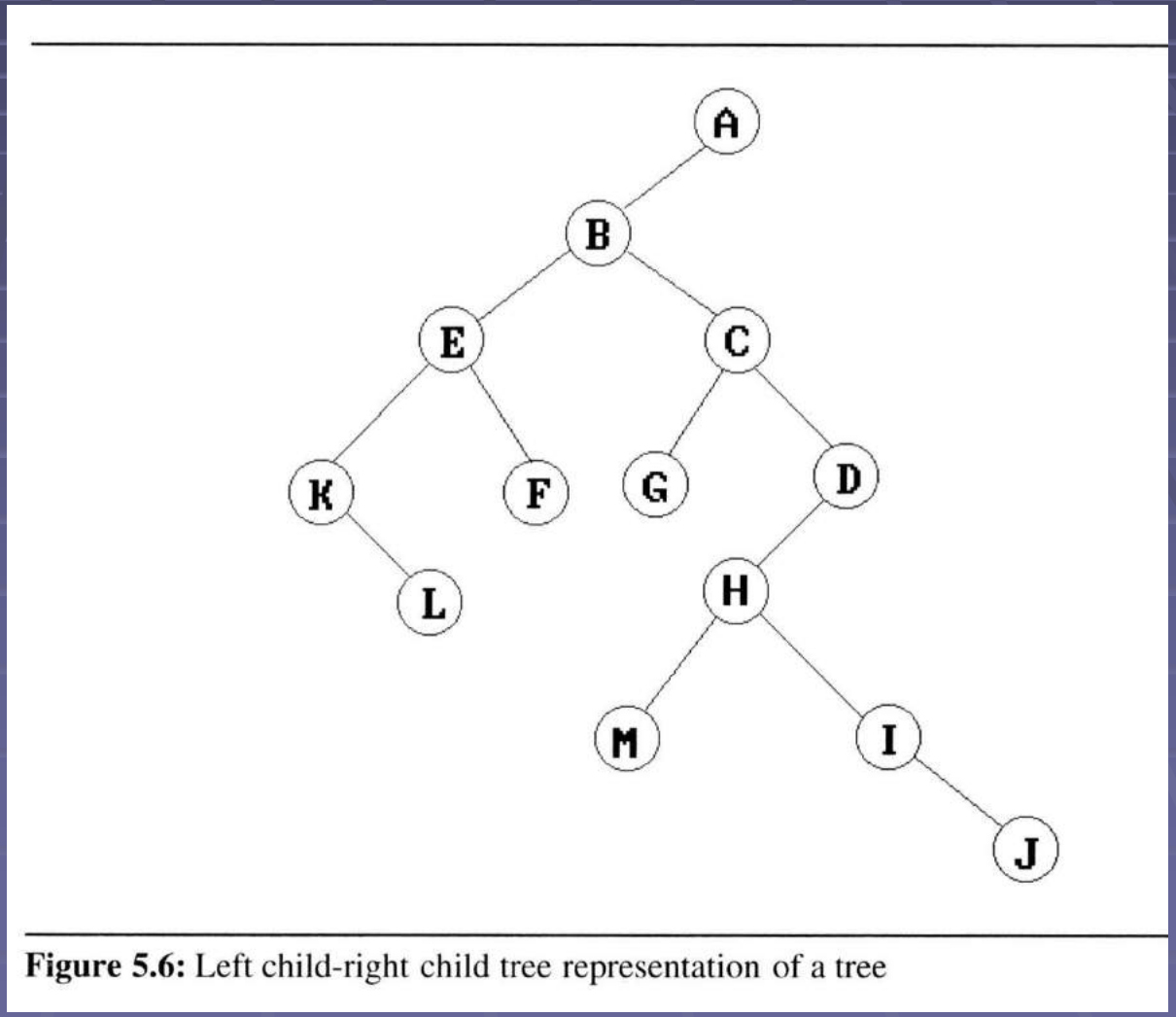


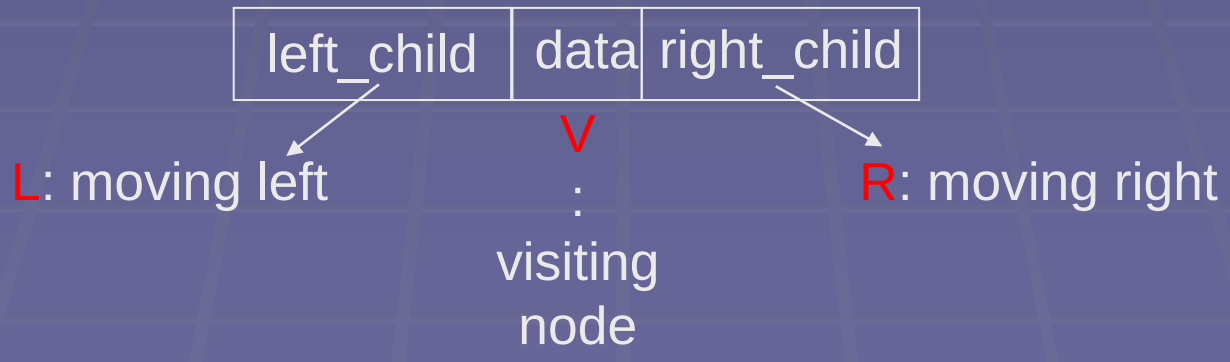
Figure 5.6: Left child-right child tree representation of a tree

Binary Tree Traversals

- How to traverse a tree or visit each node in the tree exactly once?
  - There are six possible combinations of traversal

    LVR, LRV, VLR, VRL, RVL, RLV
  - Adopt convention that we traverse left before right, only 3 traversals remain

LVR (inorder), LRV (postorder), VLR (preorder)



Binary Tree Traversals

- Arithmetic Expression using binary tree
  - inorder traversal (infix expression)

    A / B * C * D + E
  - preorder traversal (prefix expression)

    + * * / A B C D E
  - postorder traversal (postfix expression)

    A B / C * D * E +
  - level order traversal

    + * E * D / C A B

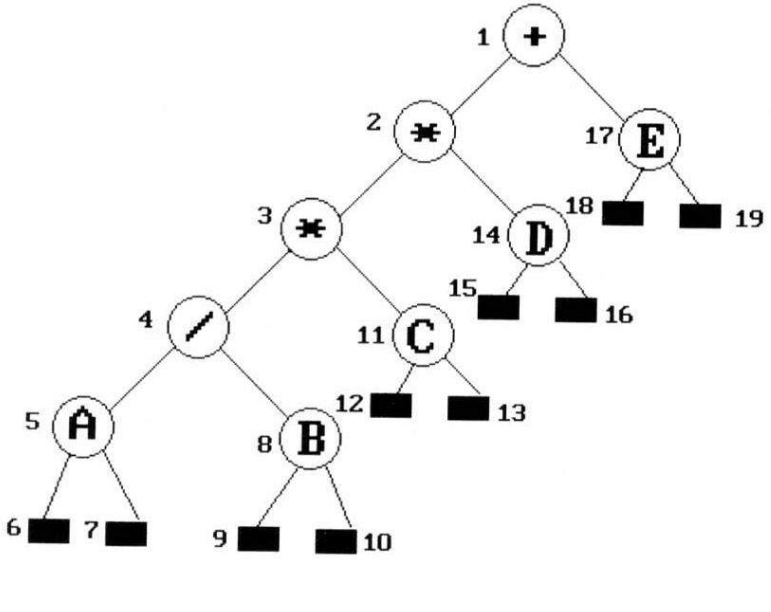


Figure 5.15: Binary tree with arithmetic expression

Binary Tree Traversals

- Inorder traversal (*LVR*) (recursive version)

output: A / B * C * D + E

```
void inorder(tree_pointer ptr)
/* inorder tree traversal */
{
   if (ptr) {
      inorder(ptr->left_child);
      printf("%d",ptr->data);
      inorder(ptr->right_child);
   }
}
```

L
V
R

Program 5.1: Inorder traversal of a binary tree

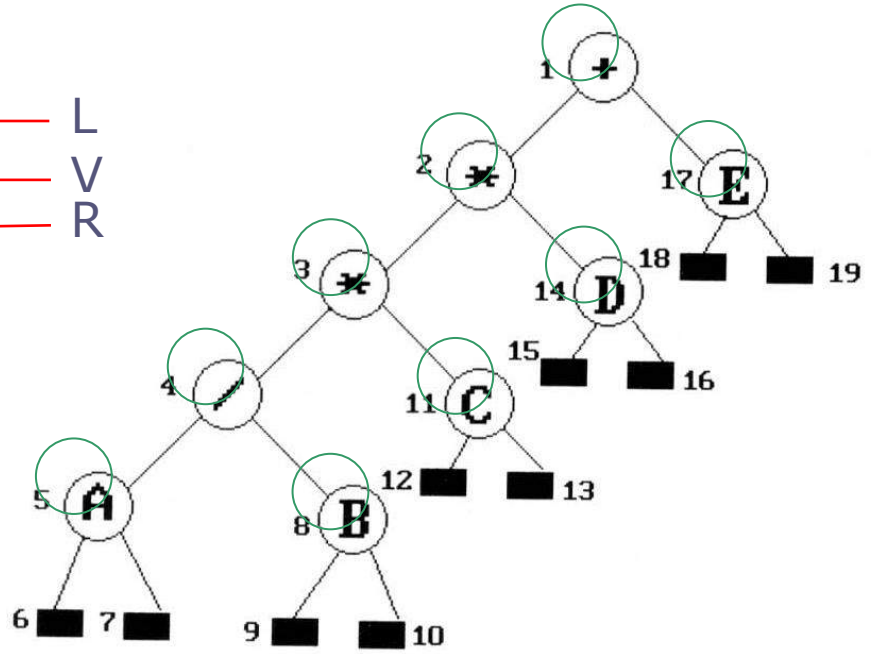


Figure 5.15: Binary tree with arithmetic expression

Binary Tree Traversals

- Preorder traversal (*VLR*) (recursive version)

output: + * * / A B C D E

```
void preorder(tree_pointer ptr)
/* preorder tree traversal */
{
   if (ptr) {
      printf("%d",ptr->data);
      preorder(ptr->left_child);
      preorder(ptr->right_child);
   }
}
```

V
L
R

Program 5.2: Preorder traversal of a binary tree

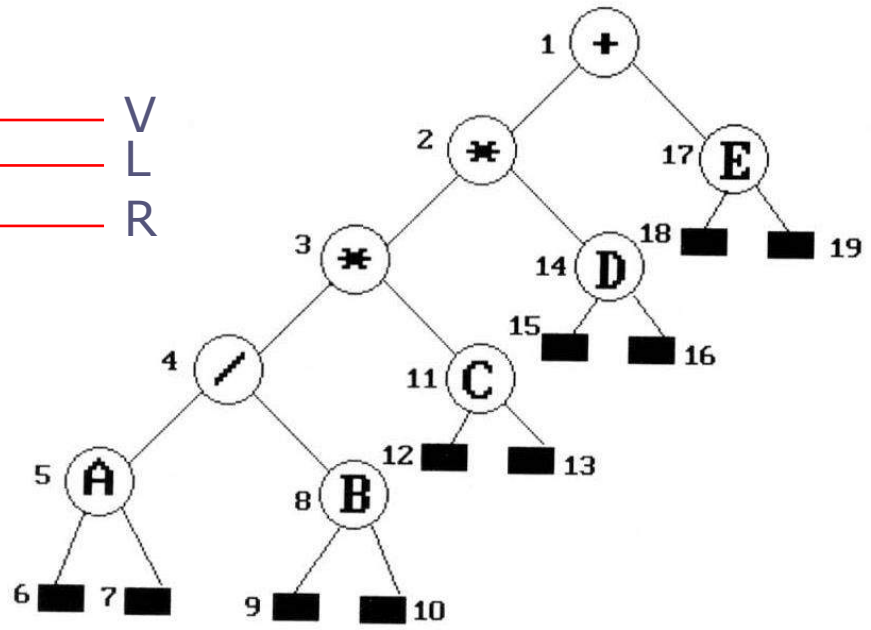


Figure 5.15: Binary tree with arithmetic expression

Binary Tree Traversals

- Postorder traversal (*LRV*) (recursive version)

output: A B / C * D * E +

```
void postorder(tree_pointer ptr)
/* postorder tree traversal */
{
   if (ptr) {
      postorder(ptr->left_child);
      postorder(ptr->right_child);
      printf("%d",ptr->data);
   }
}
```

L
R
V

Program 5.3: Postorder traversal of a binary tree

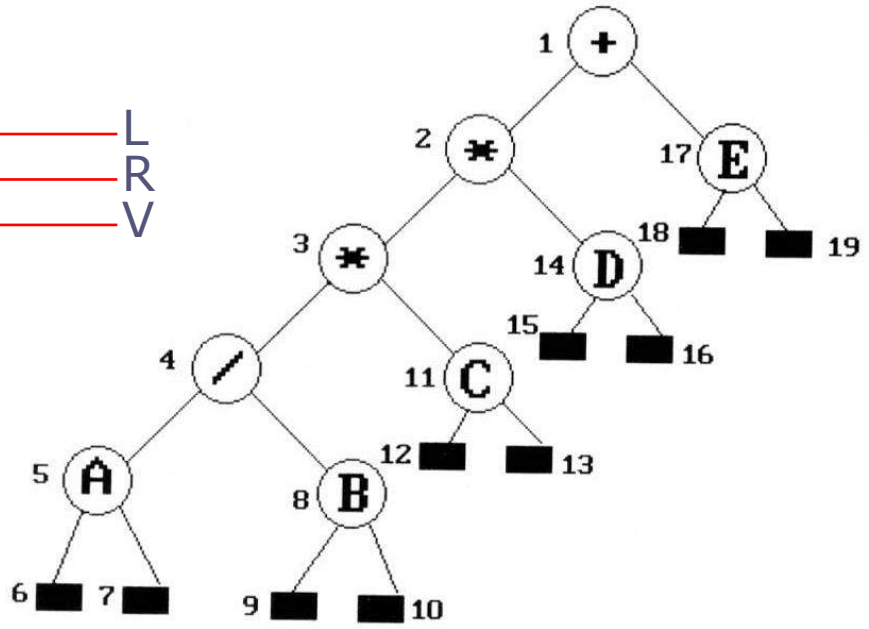


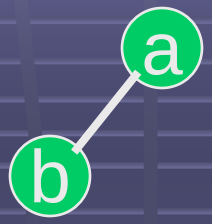
Figure 5.15: Binary tree with arithmetic expression

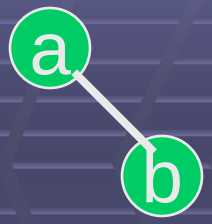
Binary Tree Construction

- Suppose that the elements in a binary tree are distinct.
- Can you construct the binary tree from which a given traversal sequence came?
- When a traversal sequence has more than one element, the binary tree is not uniquely defined.
- Therefore, the tree from which the sequence was obtained cannot be reconstructed uniquely.

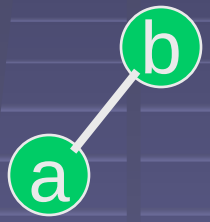
Some Examples

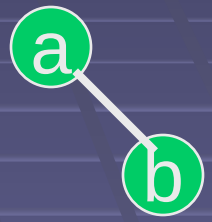
preorde
r = ab



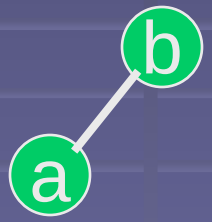


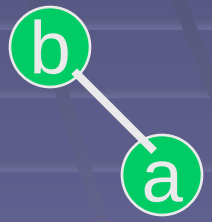
inorder
= ab



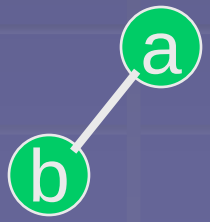


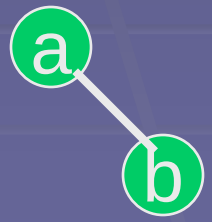
postorder
= ab





level order
= ab





Binary Tree Construction

- Can you construct the binary tree, given two traversal sequences?
- Depends on which two sequences are given.

Preorder And Postorder

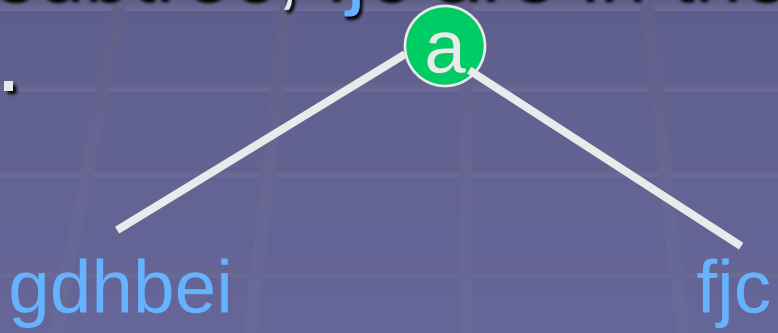
preorder = ab

postorder = ba

- Preorder and postorder do not uniquely define a binary tree.
- Nor do preorder and level order (same example).
- Nor do postorder and level order (same example).

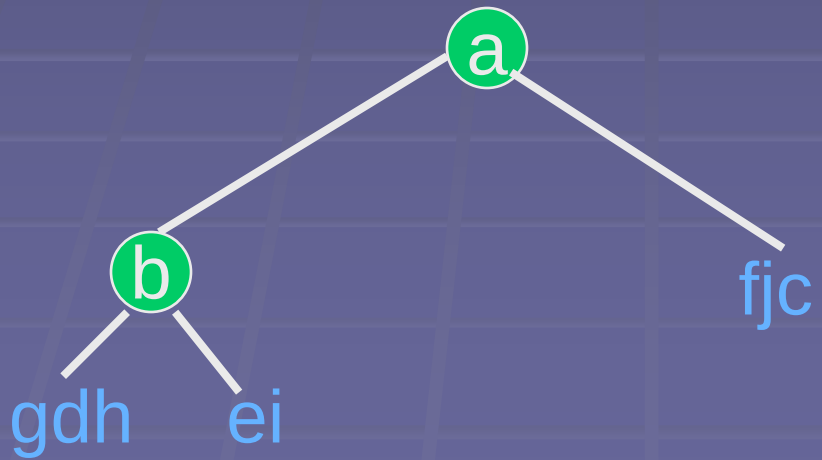
Inorder And Preorder

- inorder = g d h b e i a f j c
- preorder = a b d g h e i c f j
- Scan the preorder left to right using the inorder to separate left and right subtrees.
- a is the root of the tree; gdhbei are in the left subtree; fjc are in the right subtree.

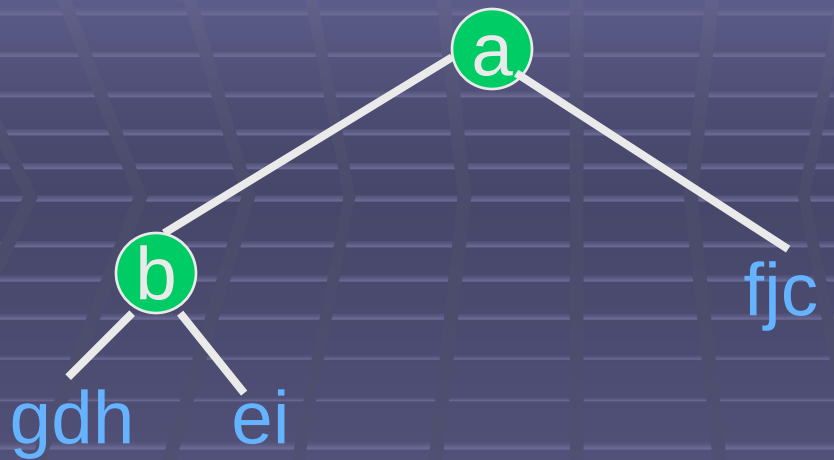


Inorder And Preorder

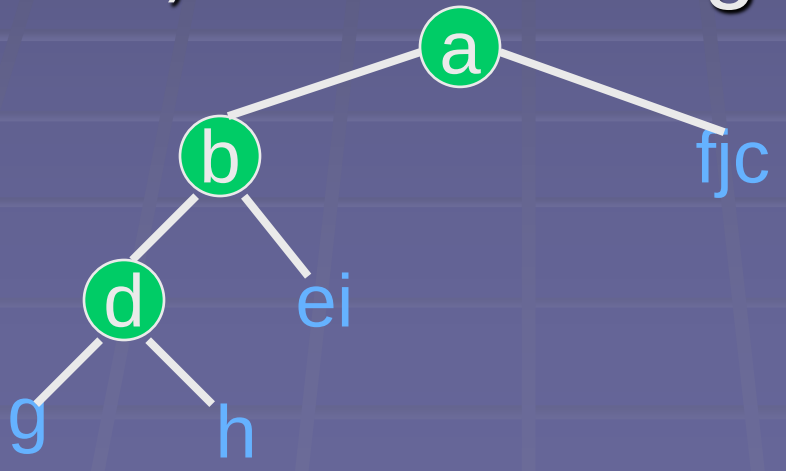
- preorder = a b d g h e i c f j
- b is the next root; gdh are in the left subtree; ei are in the right subtree.



Inorder And Preorder



- preorder = a b d g h e i c f j
- d is the next root; g is in the left subtree; h is in the right subtree.



Inorder And Postorder

- Scan postorder from right to left using inorder to separate left and right subtrees.
- inorder = g d h b e i a f j c
- postorder = g h d i e b j f c a
- Tree root is a; gdhbei are in left subtree; fjc are in right subtree.

Inorder And Level Order

- Scan level order from left to right using inorder to separate left and right subtrees.
- inorder = g d h b e i a f j c
- level order = a b c d e f g h i j
- Tree root is a; gdhbei are in left subtree; fjc are in right subtree.

Threaded Binary Trees

- Threads
  - Do you find any drawback of the above tree?
  - Too many null pointers in current representation of binary trees

    n: number of nodes

    number of non-null links: n-1

    total links: 2n

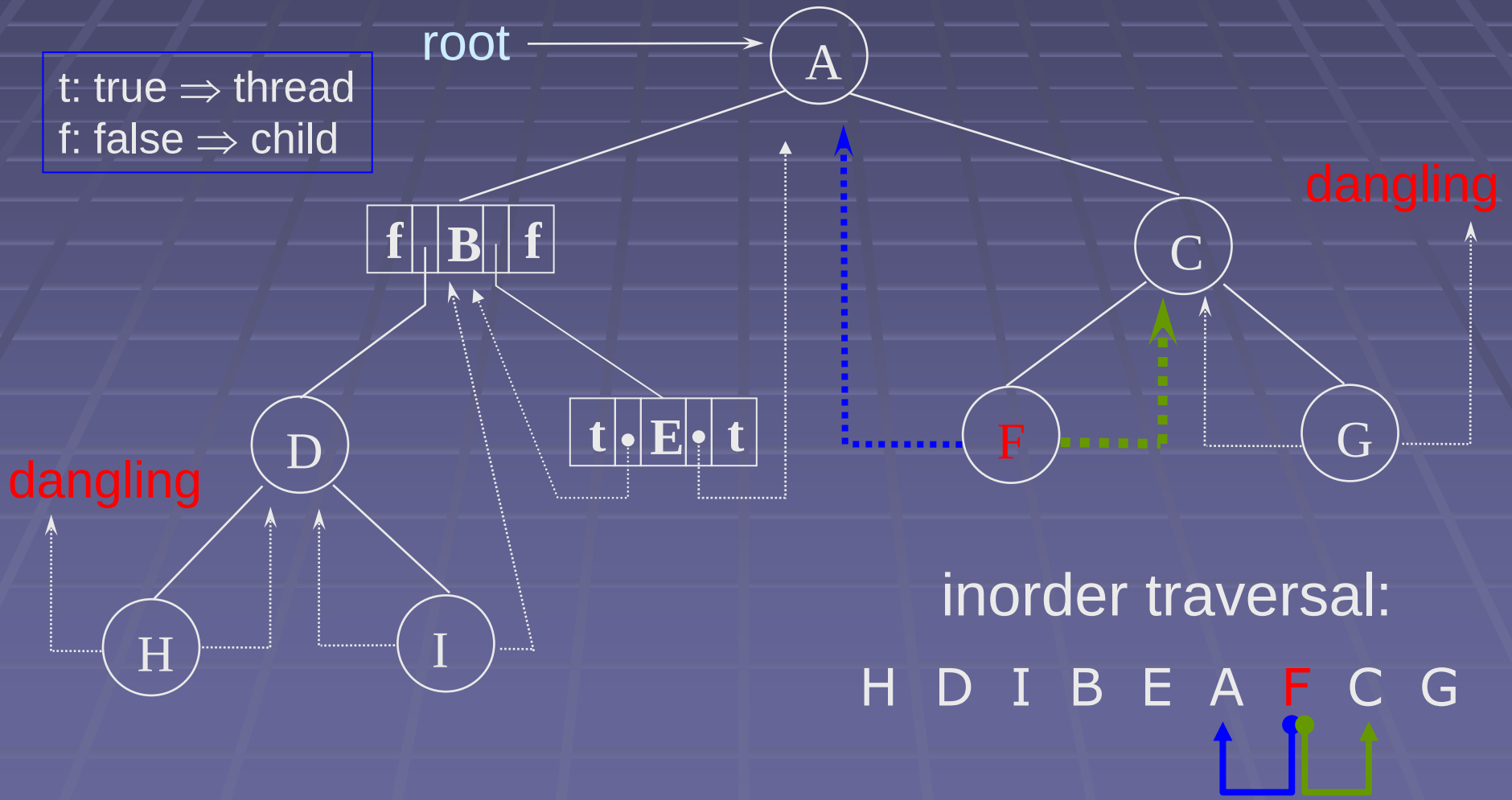
    null links: 2n-(n-1) = n+1
  - Solution: replace these null pointers with some useful “threads”

Threaded Binary Trees

- Rules for constructing the threads
  - If ptr->left_child is null,
    replace it with a pointer to the node that would be visited *before* `ptr` in an *inorder traversal*
  - If ptr->right_child is null,
    replace it with a pointer to the node that would be visited *after* `ptr` in an *inorder traversal*

Threaded Binary Trees

- A Threaded Binary Tree



Threaded Binary Trees

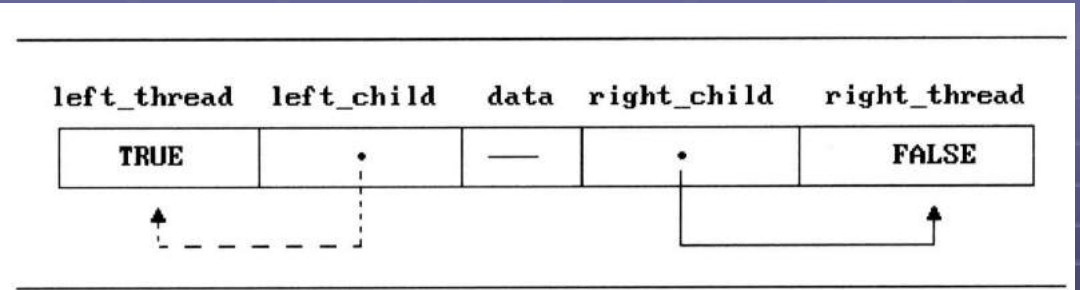


Figure 5.22: An empty threaded tree

- Two additional fields of the node structure, left-thread and right-thread
  - If ptr->left-thread=*TRUE*, then ptr->left-child contains a thread;
  - Otherwise it contains a pointer to the left child.
  - Similarly for the right-thread

```
typedef struct threaded_tree *threaded_pointer;
typedef struct threaded_tree {
        short int left_thread;
        threaded_pointer left_child;
        char data;
        threaded_pointer right_child;
        short int right_thread;
        };
```

Threaded Binary Trees

- If we don't want the left pointer of H and the right pointer of G to be dangling pointers, we may create root node and assign them pointing to the root node

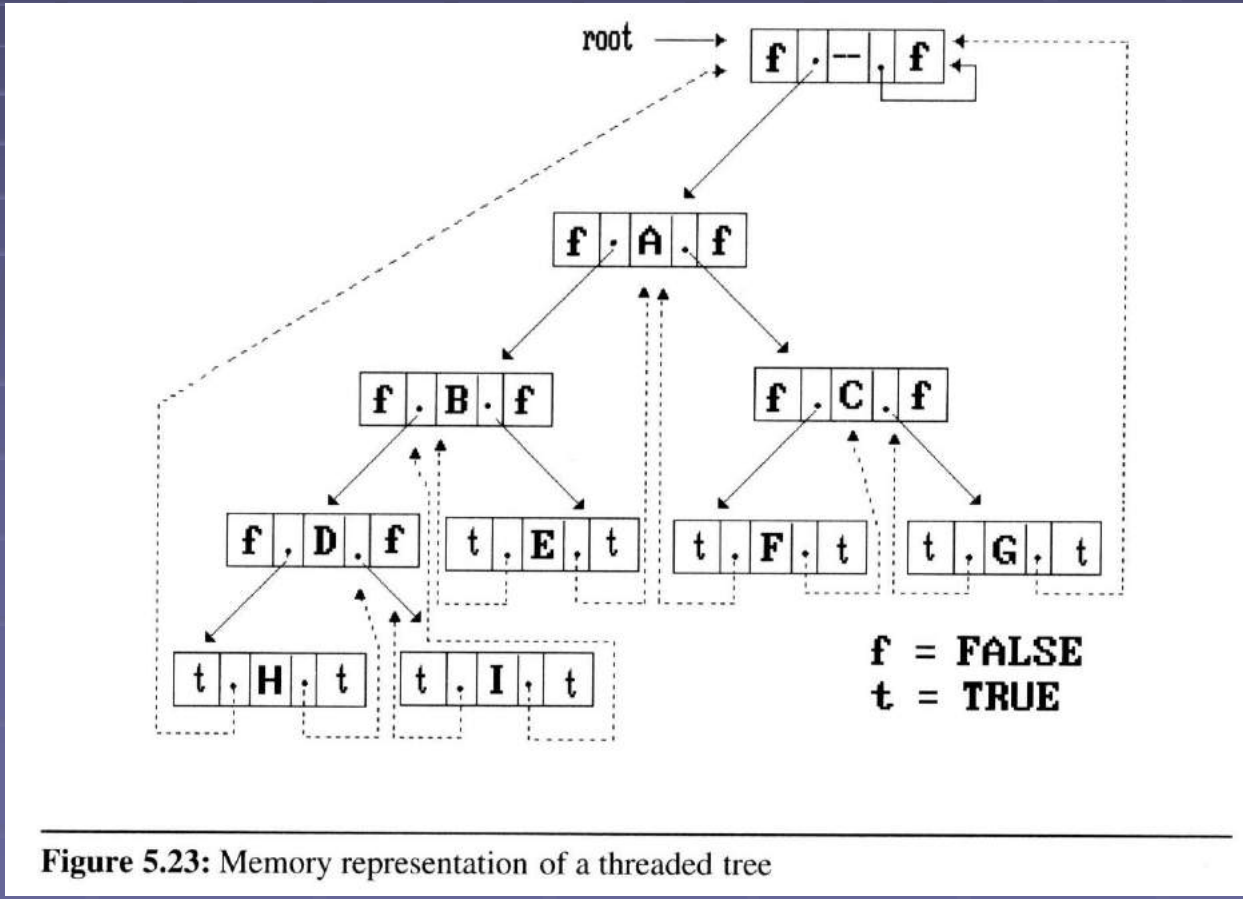


Figure 5.23: Memory representation of a threaded tree

Threaded Binary Trees

- Inorder traversal of a threaded binary tree
  - By using of threads we can perform an inorder traversal without making use of a stack (simplifying the task)
  - Now, we can follow the thread of any node, *ptr*, to the "next" node of inorder traversal

1. If *ptr->right_thread = TRUE*, the inorder successor of *ptr* is *ptr->right_child* by definition of the threads
2. Otherwise we obtain the inorder successor of *ptr* by following a path of left-child links from the right-child of *ptr* until we reach a node with *left_thread = TRUE*

Threaded Binary Trees

- Inserting A Node Into A Threaded Binary Tree
  - Insert *child* as the right child of node *parent*

1. change *parent->right_thread* to *FALSE*
2. set *child->left_thread* and *child->right_thread* to *TRUE*
3. set *child->left_child* to point to *parent*
4. set *child->right_child* to *parent->right_child*
5. change *parent->right_child* to point to *child*

Binary Search Trees

- Example: (b) and (c) are binary search trees

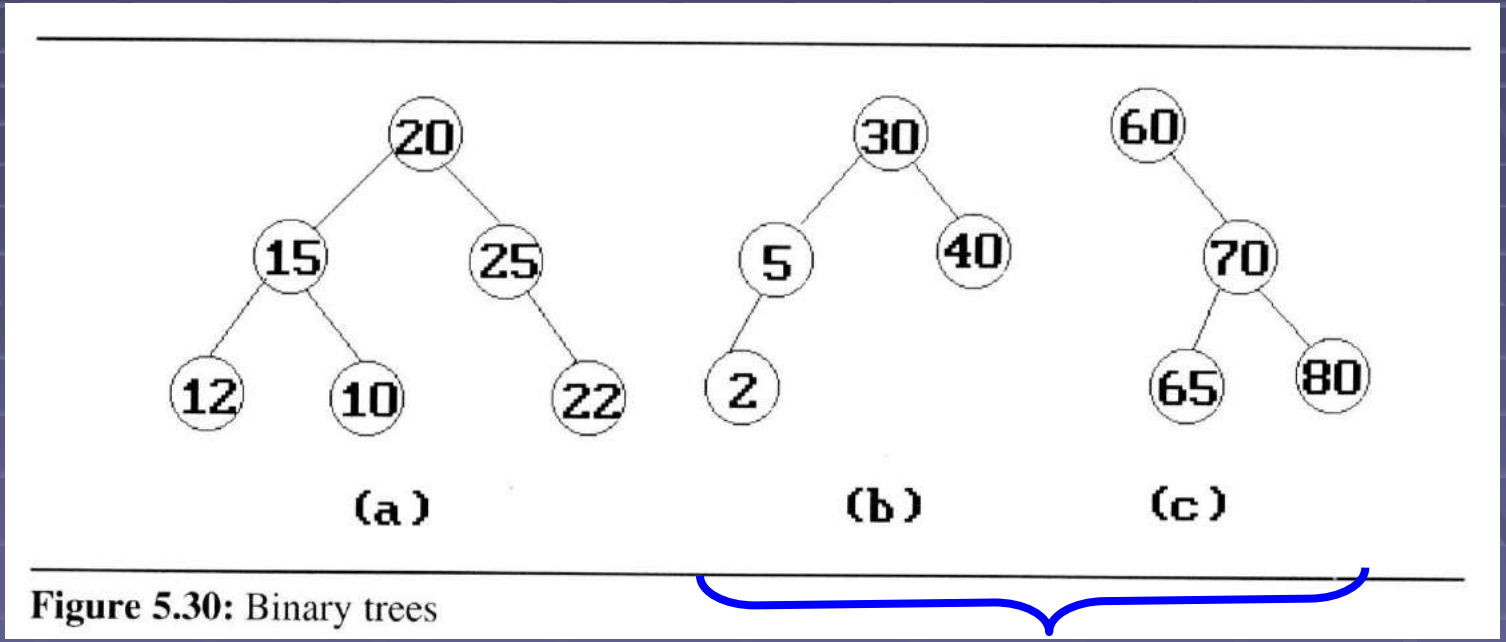
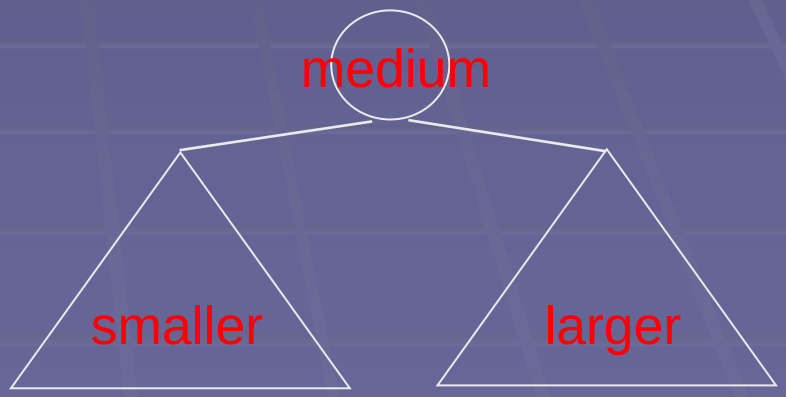
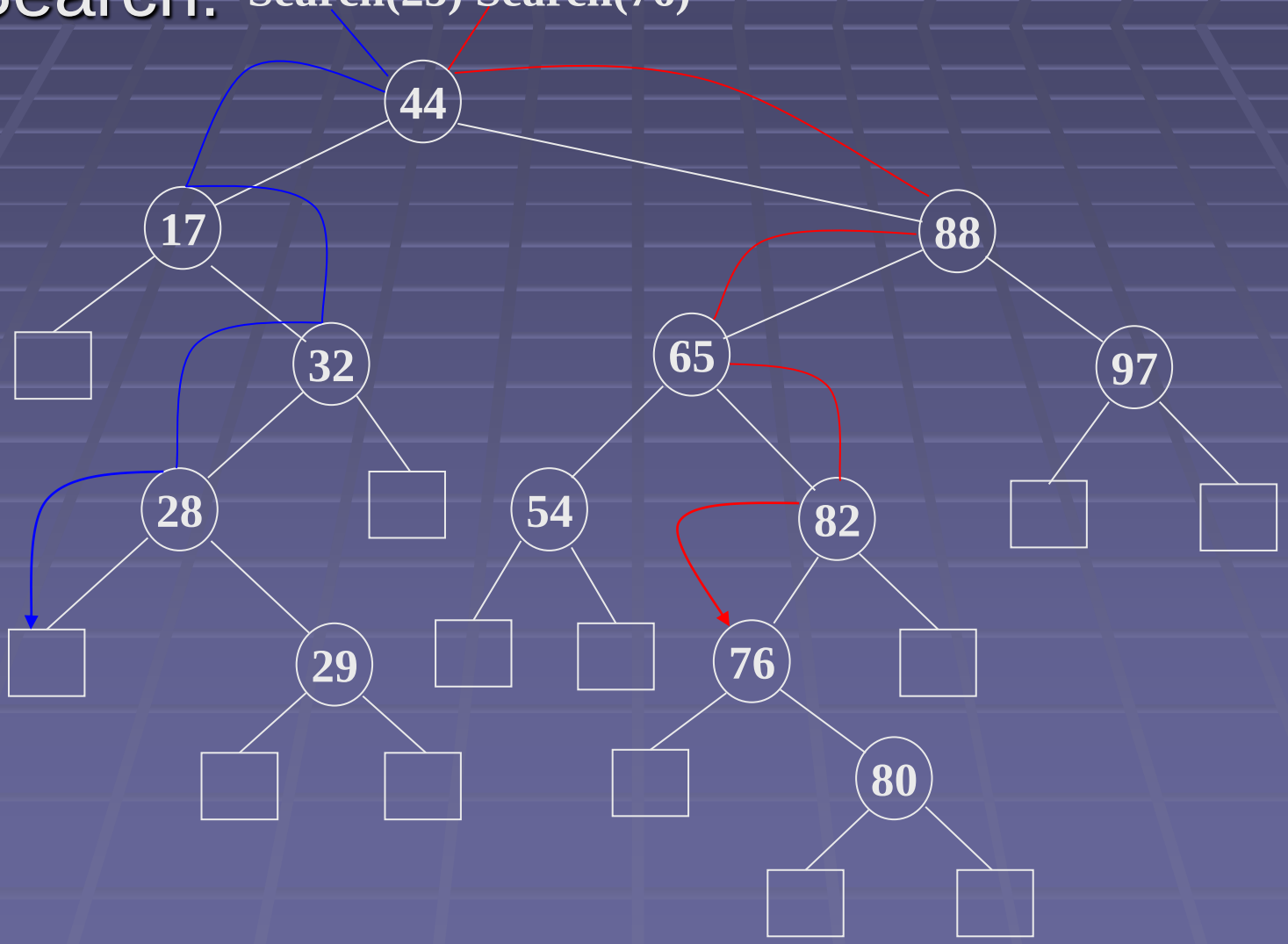


Figure 5.30: Binary trees



Binary Search Trees

- Search: Search(25) Search(76)



Binary Search Trees

- Searching a binary search tree

```
tree_pointer search(tree_pointer root, int key)
{
/* return a pointer to the node that contains key.  If
there is no such node, return NULL. */
   if (!root) return NULL;
   if (key == root->data) return root;
   if (key < root->data)
      return search(root->left_child, key);
   return search(root->right_child,key);
}
```

Program 5.15: Recursive search of a binary search tree

```
tree_pointer search2(tree_pointer tree, int key)
{
/* return a pointer to the node that contains key.  If
there is no such node, return NULL. */
   while (tree) {
      if (key == tree->data) return tree;
      if (key < tree->data)
         tree = tree->left_child;
      else
         tree = tree->right_child;
      }
   return NULL;
}
```

Program 5.16: Iterative search of a binary search tree

O(h)

Binary Search Trees

- Inserting into a binary search tree

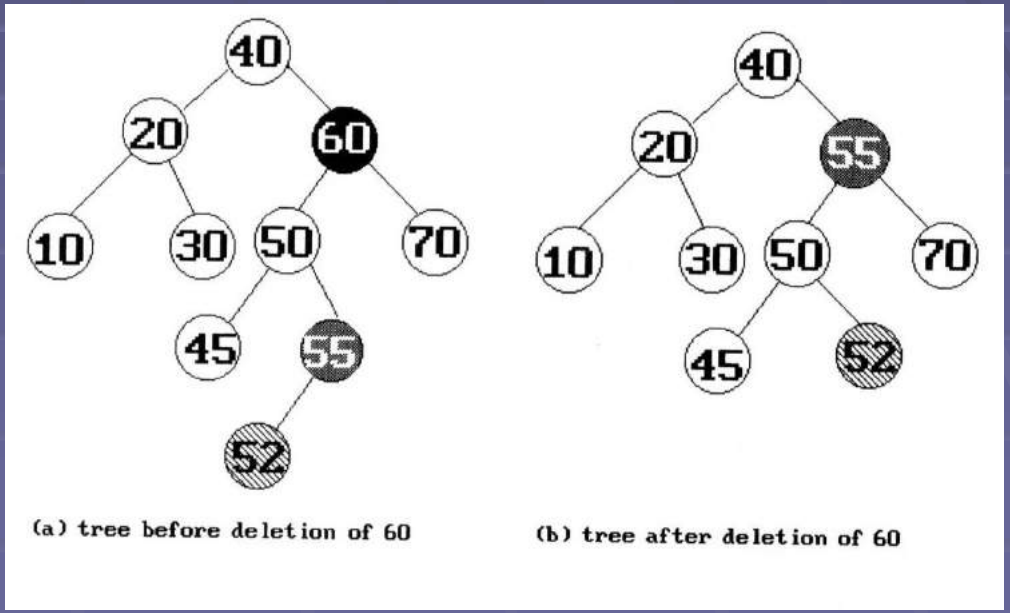
```
void insert_node(tree_pointer *node, int num)
/* If num is in the tree pointed at by node do nothing;
otherwise add a new node with data = num */
{
   tree_pointer ptr, temp = modified_search(*node, num);
   if (temp || !(*node)) {
      /* num is not in the tree */
      ptr = (tree_pointer)malloc(sizeof(node));
      if (IS_FULL(ptr)) {
         fprintf(stderr, "The memory is full\n");
         exit(1);
      }
      ptr->data = num;
      ptr->left_child = ptr->right_child = NULL;
      if (*node) /* insert as child of temp */
         if (num < temp->data) temp->left_child = ptr;
         else temp->right_child = ptr;
      else *node = ptr;
   }
}
```

An empty tree

Program 5.17: Inserting an element into a binary search tree

Binary Search Trees

- Deletion from a binary search tree
  - Three cases should be considered
  - case 1. leaf $\rightarrow$ delete
  - case 2.
    one child $\rightarrow$ delete and change the pointer to this child
  - case 3. two child $\rightarrow$ either the smallest element in the right subtree or the largest element in the left subtree



Binary Search Trees

- Height of a binary search tree
  - The height of a binary search tree with *n* elements can become as large as *n*.
  - It can be shown that when insertions and deletions are made at random, the height of the binary search tree is $O(\log_2 n)$ on the average.
  - Search trees with a worst-case height of $O(\log_2 n)$ are called *balance search trees*

Binary Search Trees

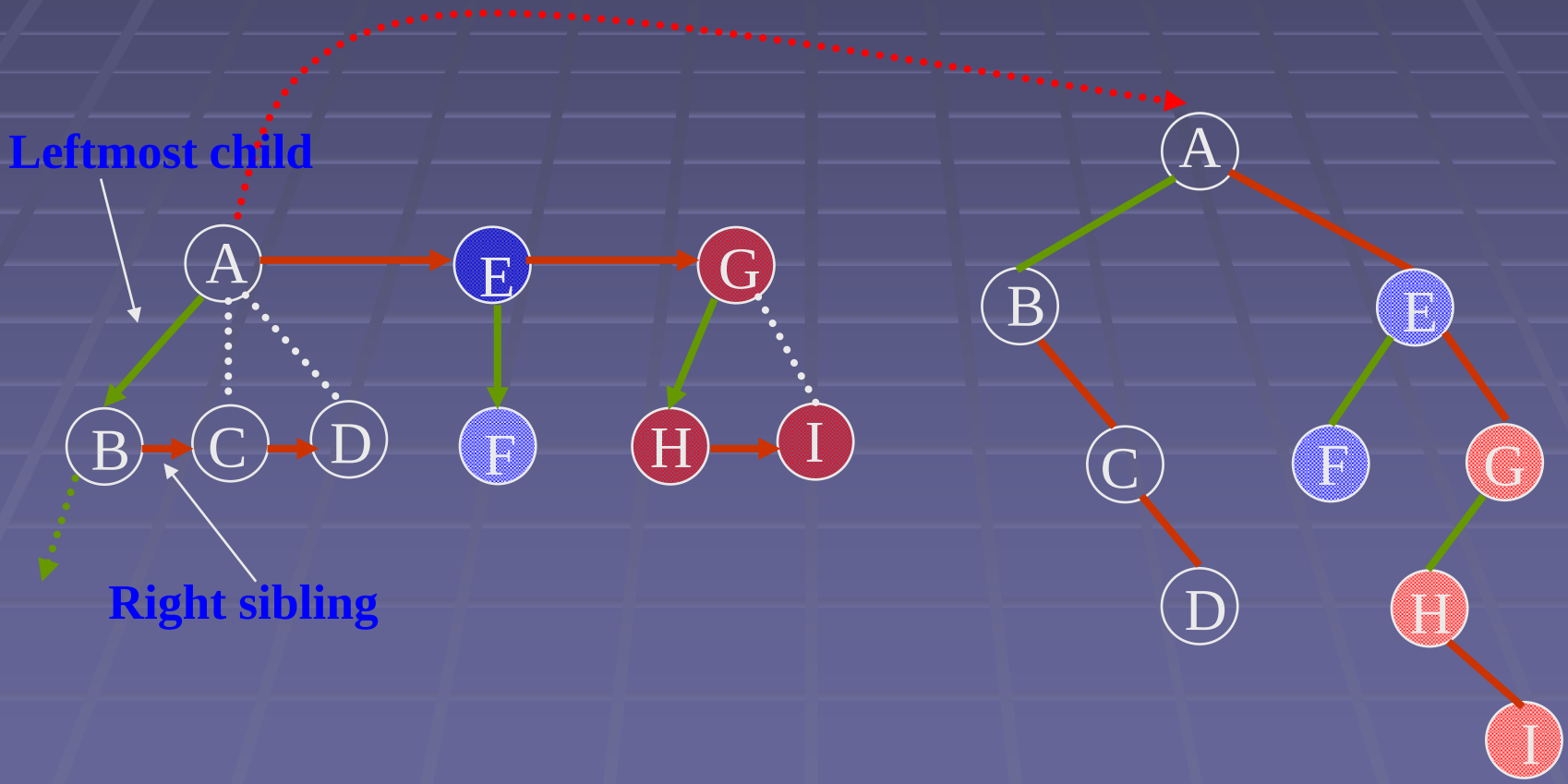
- Time Complexity
  - Searching, insertion, removal
    - O(*h*), where h is the height of the tree
  - Worst case - skewed binary tree
    - O(*n*), where n is the # of internal nodes
- Prevent worst case
  - rebalancing scheme
  - AVL, 2-3, and Red-black tree

Forests

- Definition:
  - A *forest* is a set of $n \geq 0$ disjoint trees
- Transforming a forest into a binary tree
  - Definition: If $T_1,\ldots,T_n$ is a forest of trees, then the binary tree corresponding to this forest, denoted by $B(T_1,\ldots,T_n)$:
  - is empty, if $n = 0$
  - has root equal to root(T_1); has left subtree equal to B($T_{11},T_{12},\ldots,T_{1m}$); and has right subtree equal to B($T_2,T_3,\ldots,T_n$)
    - where $T_{11},T_{12},\ldots,T_{1m}$ are the subtrees of root (T_1)

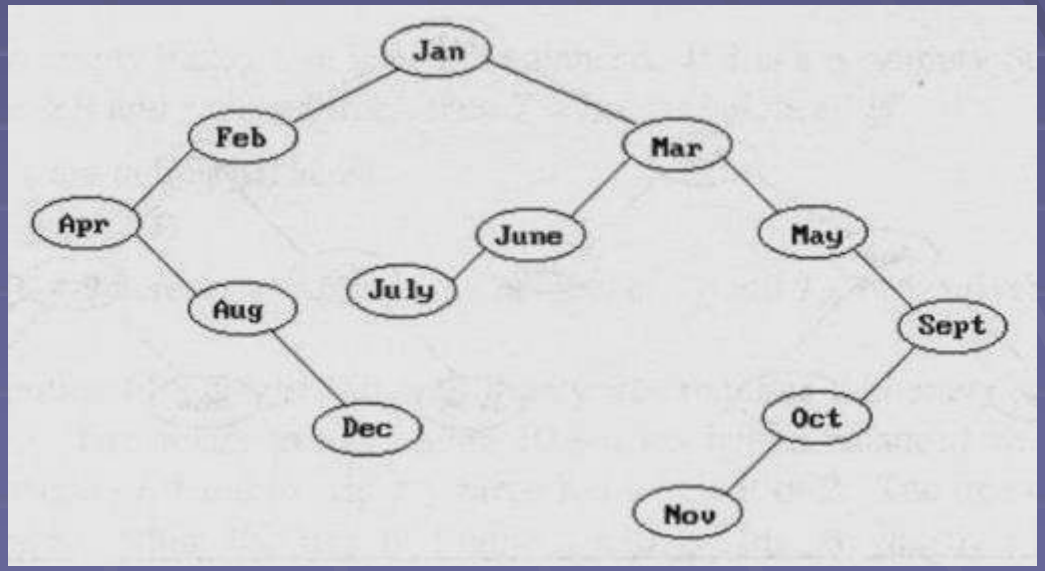
Forests

- Rotate the tree clockwise by 45 degrees



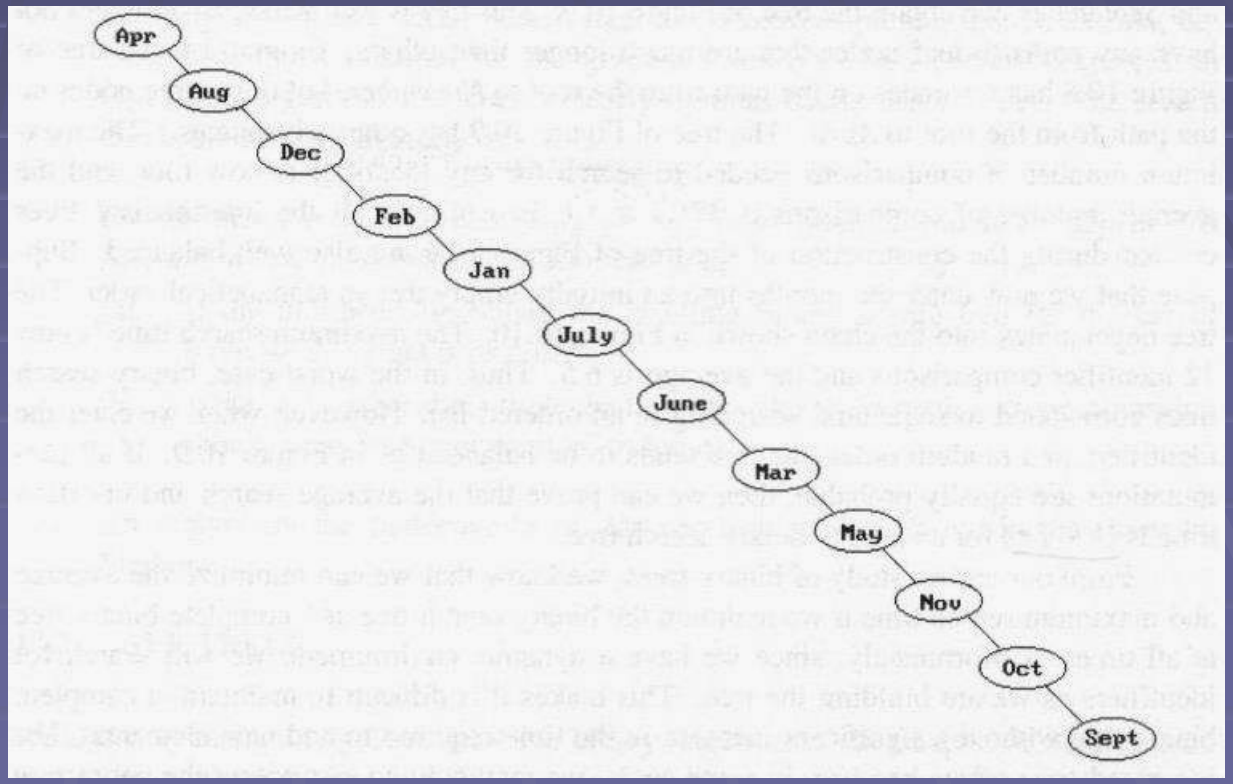
AVL Trees

- We also may maintain dynamic tables as binary search trees.
  - Figure 10.8 shows the binary search tree obtained by entering the months *January* to *December*, in that order, into an initially empty binary search tree
  - The maximum number of comparisons needed to search for any identifier in the tree of Figure 10.8 is six (for *November*).
  - Average number of comparisons is 42/12 = 3.5

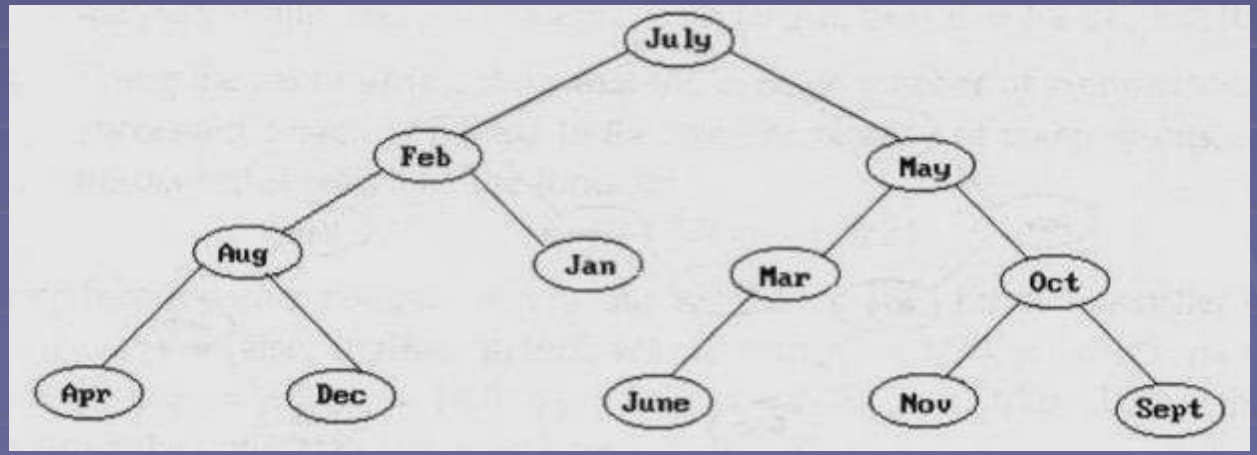


AVL Trees

- Suppose that we now enter the months into an initially empty tree in alphabetical order
  - The tree degenerates into the chain
  - number of comparisons: maximum: 12, and average: 6.5
  - in the worst case, binary search trees correspond to sequential searching in an ordered list



- Another insert sequence
  - In the order *Jul*, *Feb*, *May*, *Aug*, *Jan*, *Mar*, *Oct*, *Apr*, *Dec*, *Jun*, *Nov*, and *Sep*, by Figure 10.9.
  - Well balanced and does not have any paths to leaf nodes that are much longer than others.
  - Number of comparisons:
    maximum: 4, and average: 37/12 ≈ 3.1.
  - All intermediate trees created during the construction of Figure 10.9 are also well balanced
  - If all permutations are equally probable, then we can prove that the average search and insertion time is $O(\log n)$ for n node binary search tree

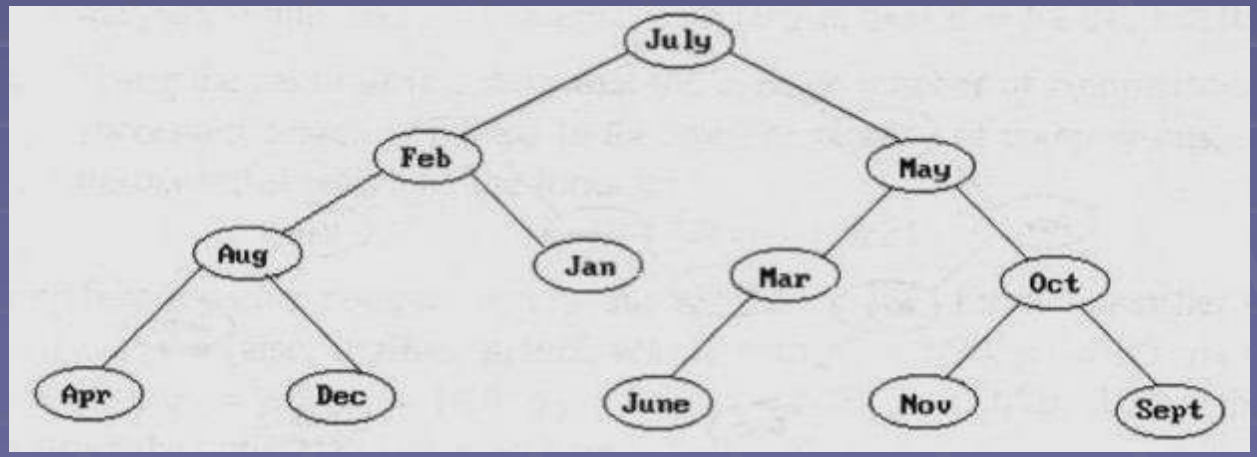


AVL Trees

- Since we have a dynamic environment, it is hard to achieve:
  - Required to add new elements and maintain a complete binary tree without a significant increasing time
- Adelson-Velskii and Landis introduced a binary tree structure (*AVL trees*):
  - Balanced with respect to the heights of the subtrees.
  - We can perform dynamic retrievals in O(log*n*) time for a tree with n nodes.
  - We can enter an element into the tree, or delete an element form it, in O(log*n*) time. The resulting tree remain height balanced.
  - As with binary trees, we may define AVL tree recursively

AVL Trees

- **Definition:**
  - An empty binary tree is height balanced. If T is a nonempty binary tree with T_L and T_R as its left and right subtrees, then T is *height balanced iff*
    - T_L and T_R are height balanced, and
    - $|h_L - h_R| \leq 1$ where h_L and h_R are the heights of T_L and T_R, respectively.
- The definition of a height balanced binary tree requires that every subtree also be height balanced



AVL Trees

- This time we will insert the months into the tree in the order
  - *Mar*, *May*, *Nov*, *Aug*, *Apr*, *Jan*, *Dec*, *Jul*, *Feb*, *Jun*, *Oct*, *Sep*
- It shows the tree as it grows, and the restructuring involved in keeping it balanced.
- The numbers by each node represent the difference in heights between the left and right subtrees of that node
- We refer to this as the balance factor of the node
- **Definition:**
  - The balance factor, *BF(T)*, of a node, *T*, in a binary tree is defined as h_L - h_R, where h_L(h_R) are the heights of the left(right) subtrees of *T*.
    For any node *T* in an AVL tree *BF(T)* = -1, 0, or 1.

AVL Trees

- Insertion into an AVL tree

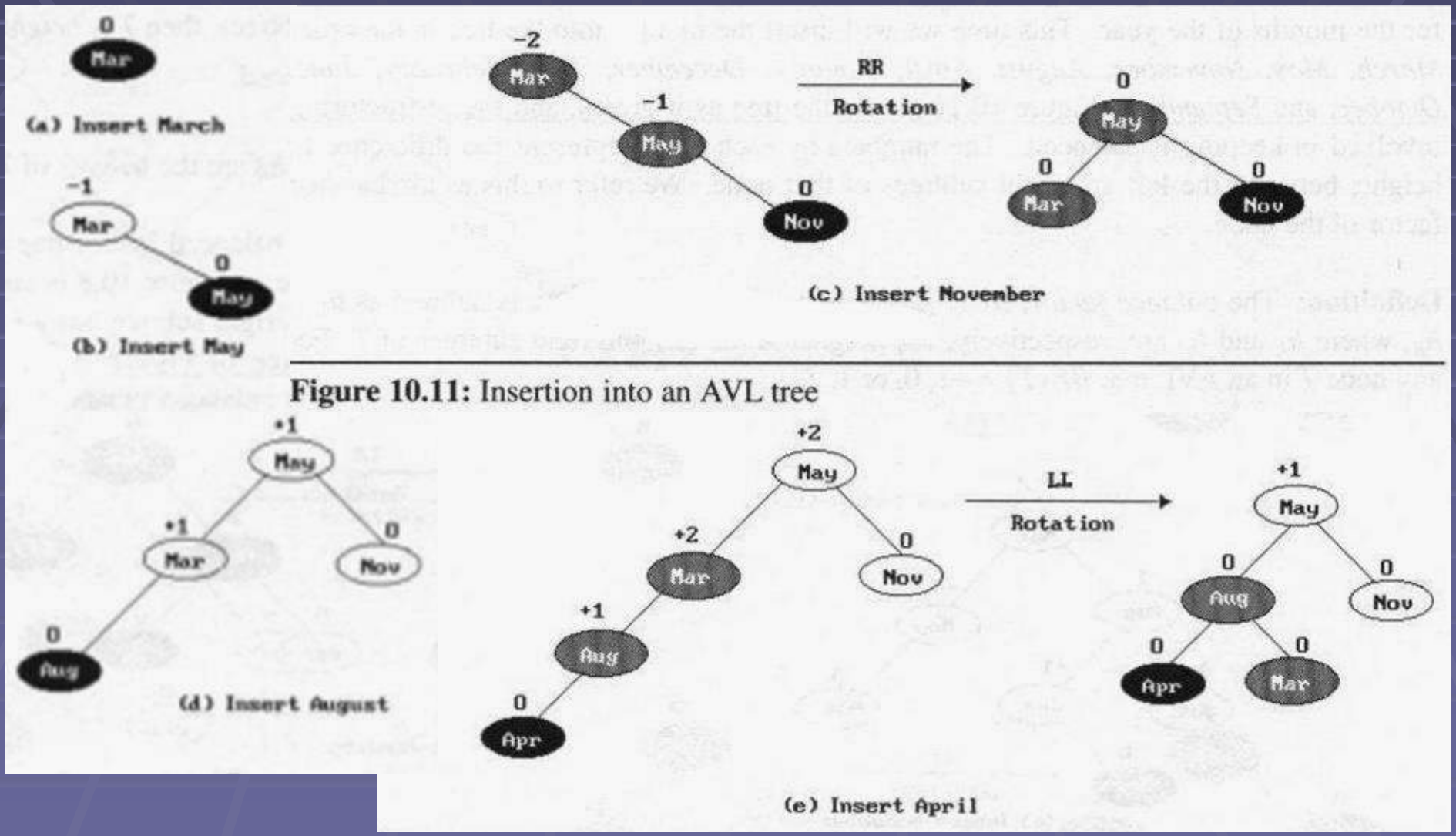
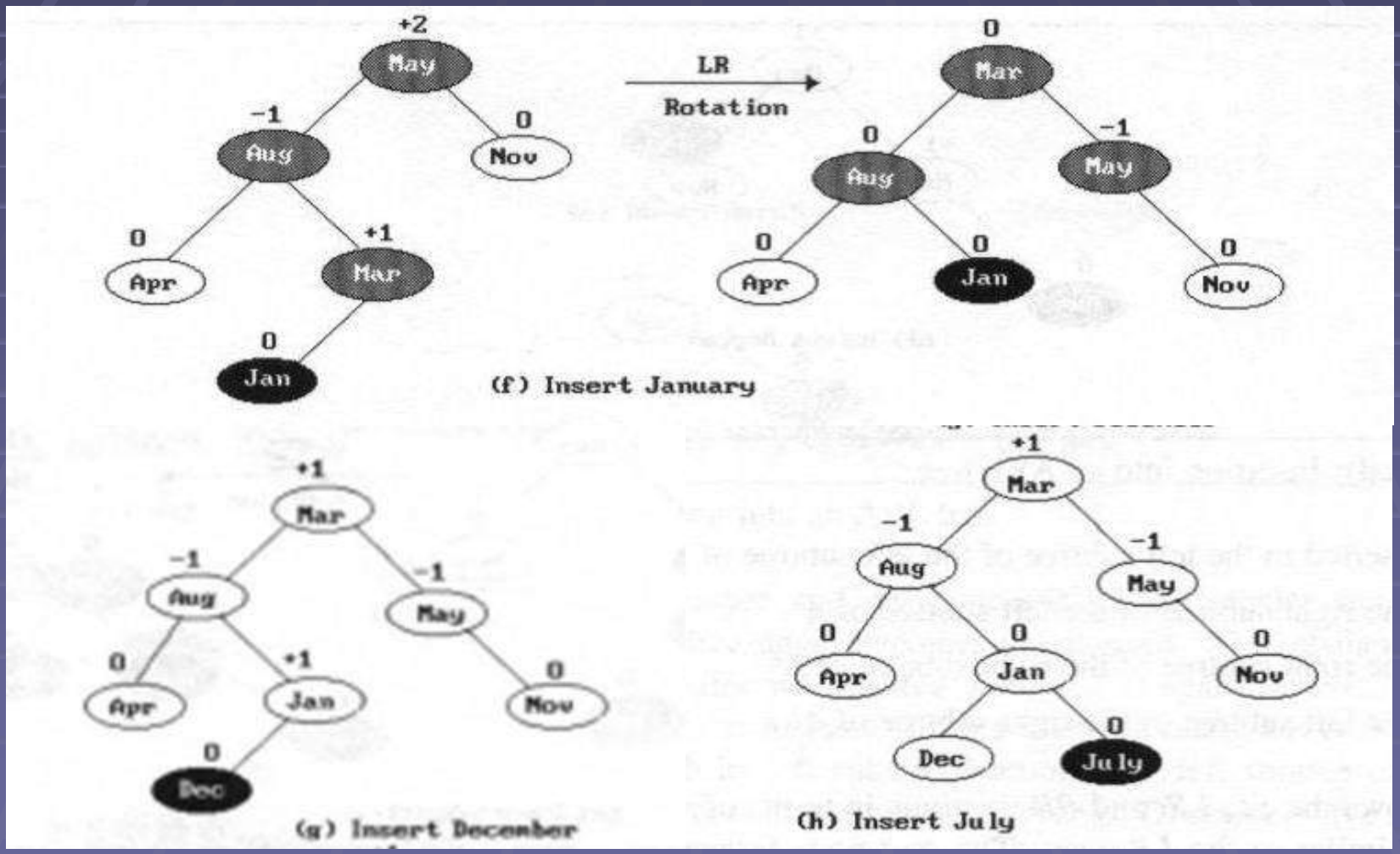


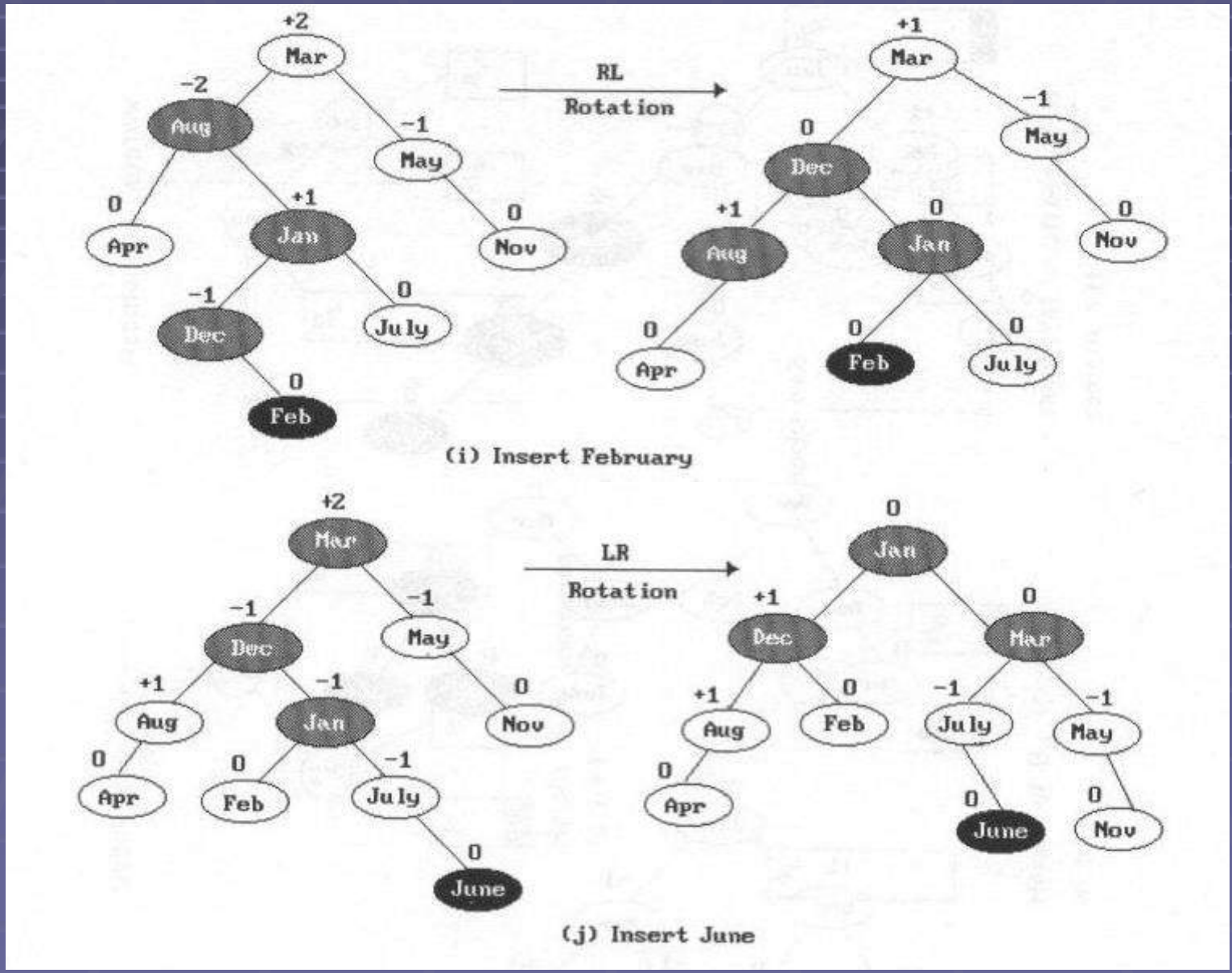
Figure 10.11: Insertion into an AVL tree

AVL Trees

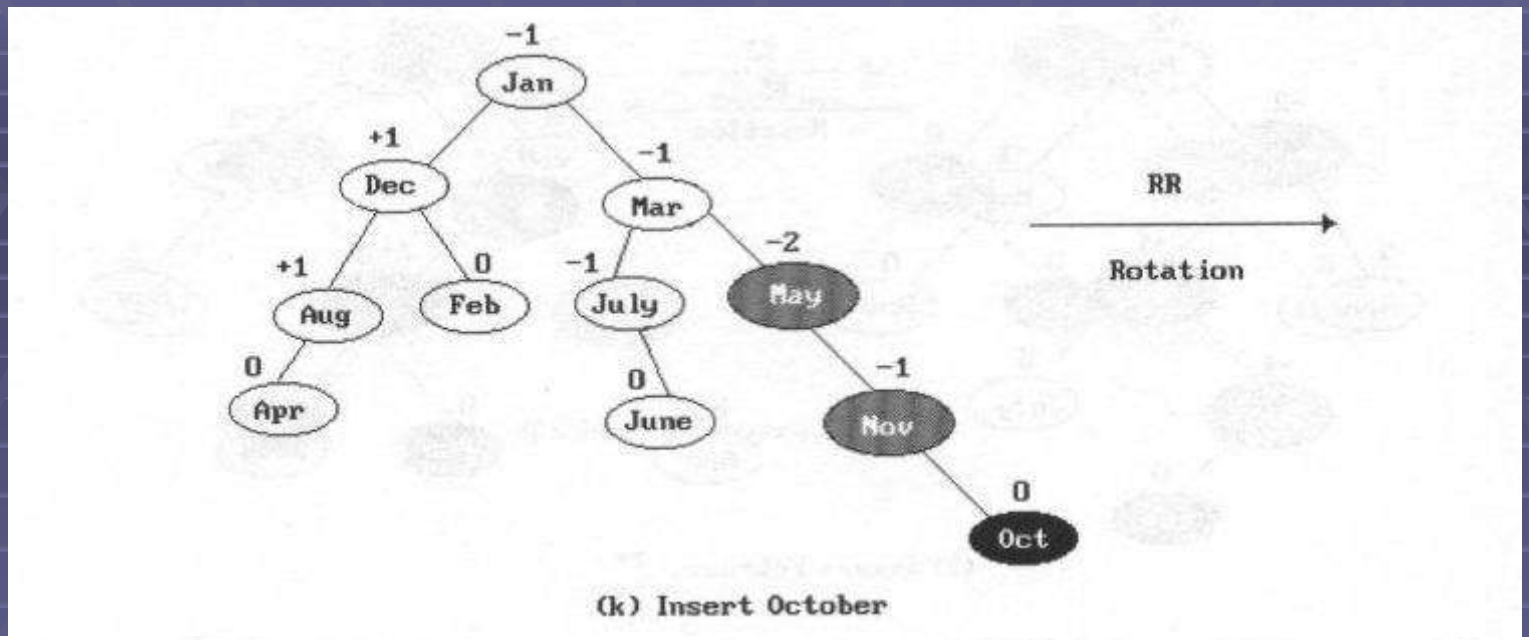
- Insertion into an AVL tree (cont'd)



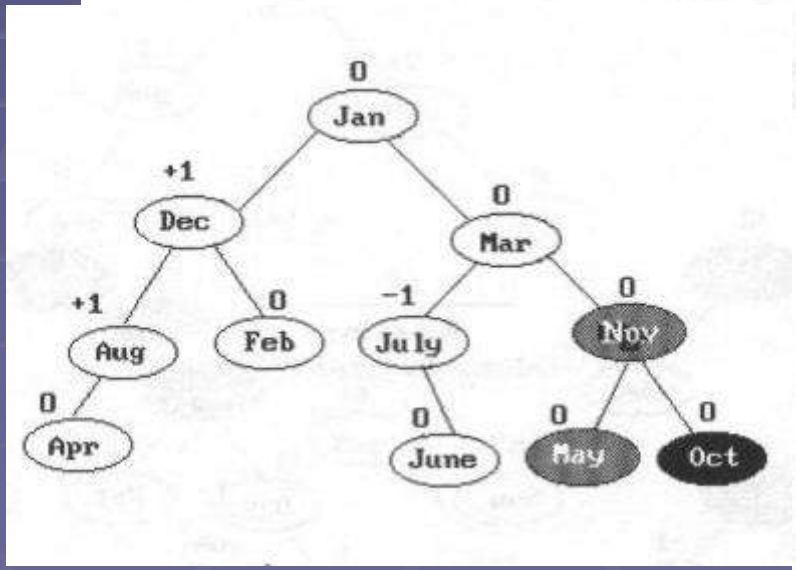
- Insertion into an AVL tree (cont'd)

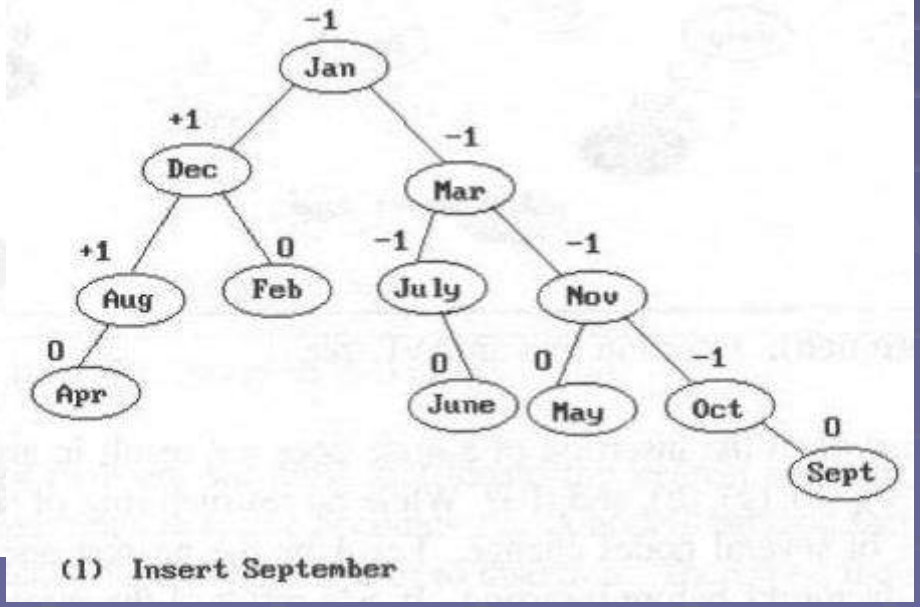


- Insertion into an AVL tree (cont'd)



(k) Insert October





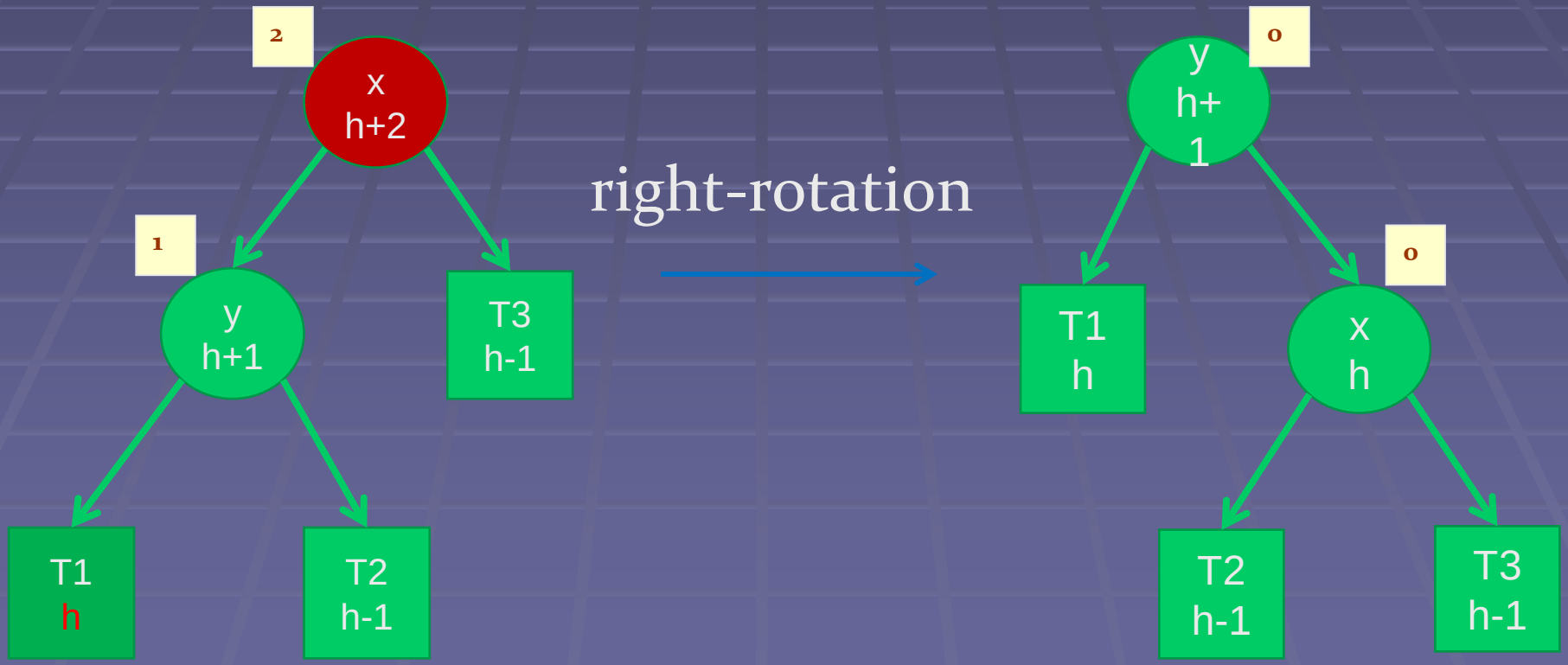
(l) Insert September

AVL Trees

- We carried out the rebalancing using four different kinds of rotations:
  LL, *RR*, *LR*, and *RL*
  - *LL* and *RR* are symmetric as are *LR* and *RL*
  - These rotations are characterized by the nearest ancestor, *A*, of the inserted node, *Y*, whose balance factor becomes ± 2.
    - *LL*: *Y* is inserted in the left subtree of the left subtree of *A*.
    - *LR*: *Y* is inserted in the right subtree of the left subtree of *A*
    - *RR*: *Y* is inserted in the right subtree of the right subtree of *A*
    - *RL*: *Y* is inserted in the left subtree of the right subtree of *A*

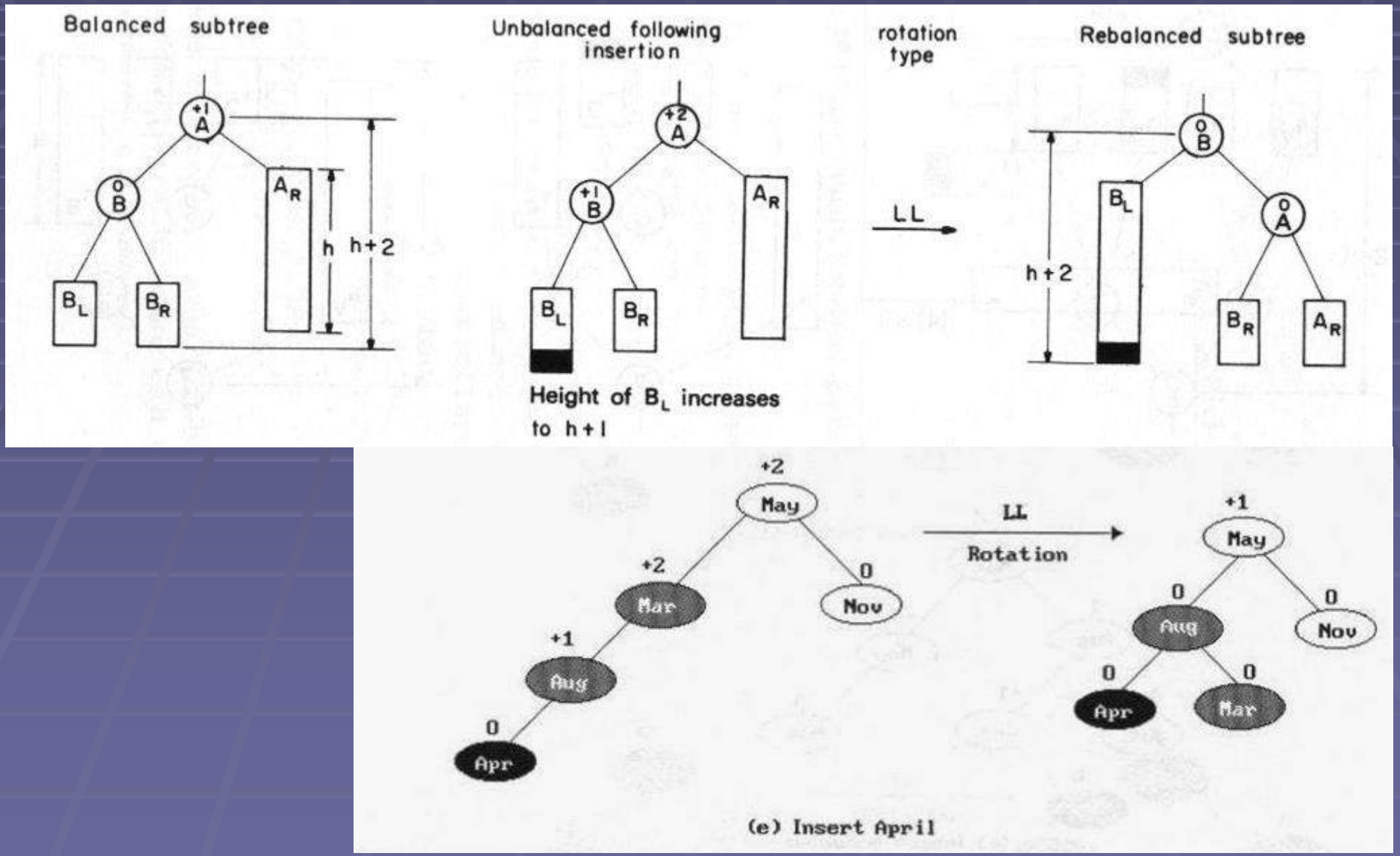
Case 1 : right-rotation to re-balance

Case 1(LL): balance-factor(x) = 2, balance-factor(x->left) >= 0
right_rotation(x) it return y as new top node



AVL Trees

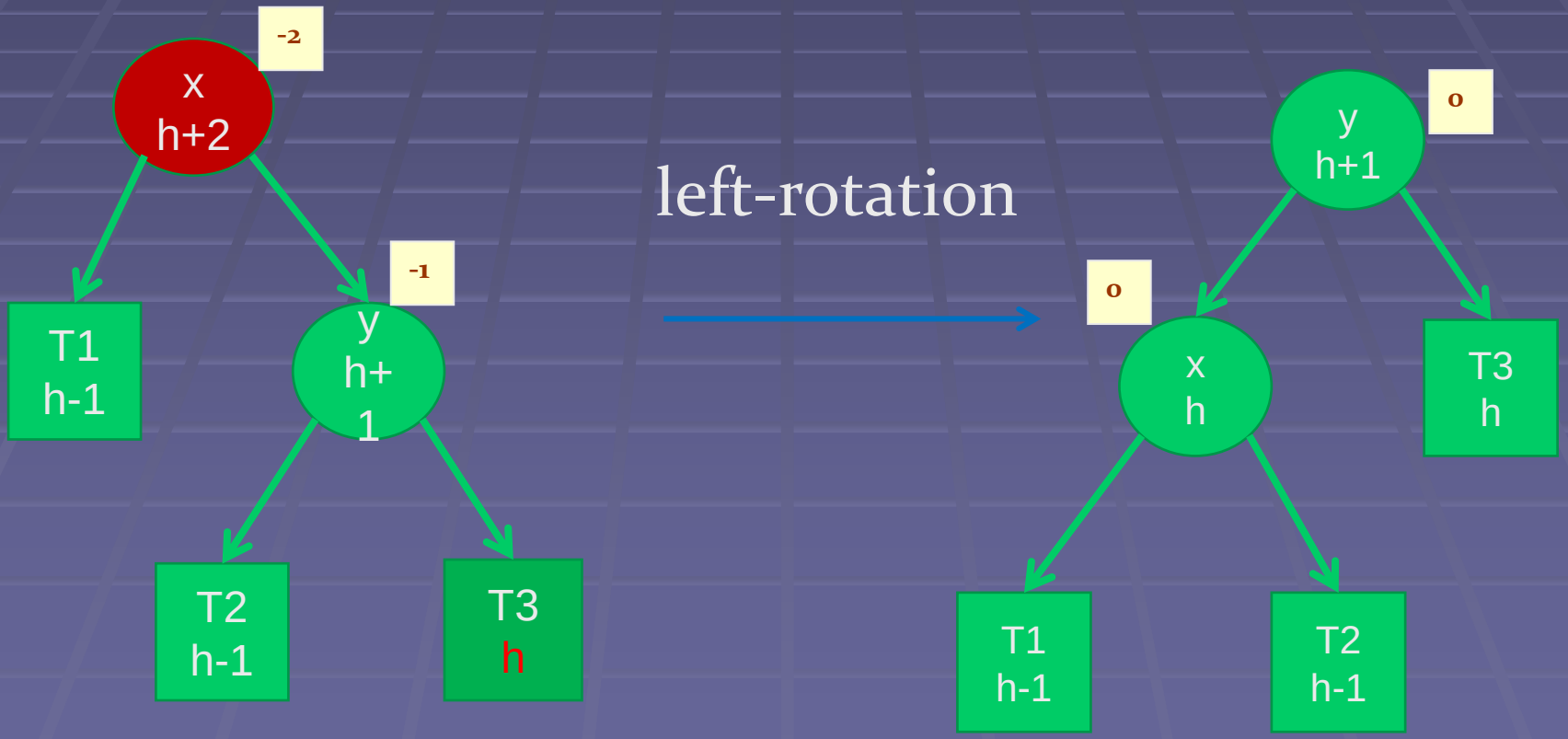
- Rebalancing rotations



left-rotation to re-balance

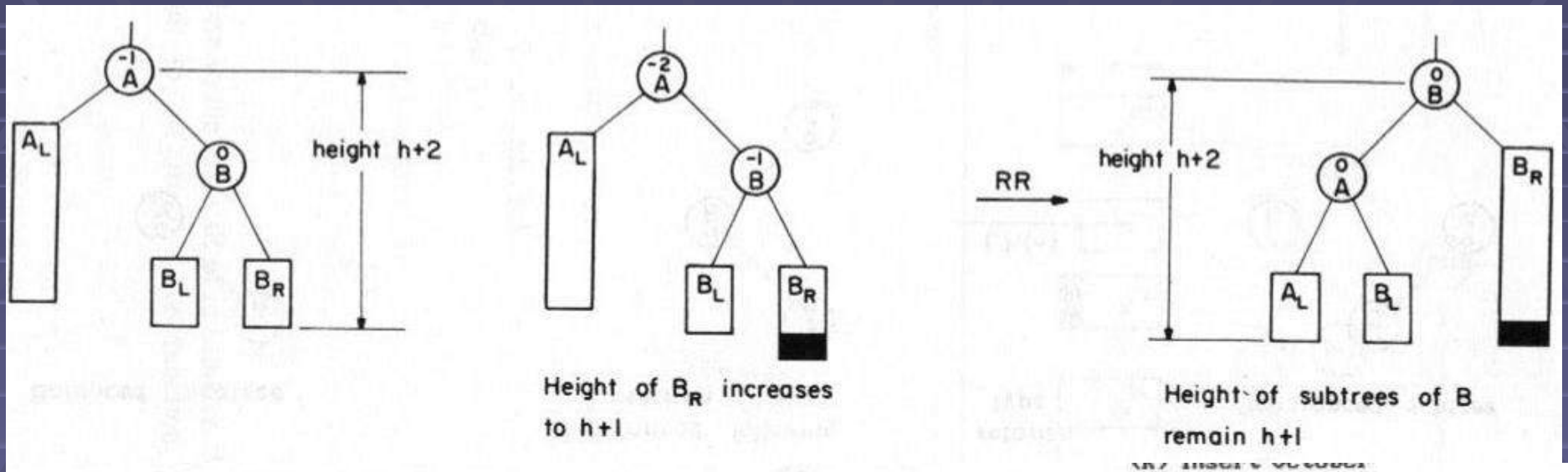
Case 2(RR): balance-factor(x) = -2, balance-factor(x->right) <= 0

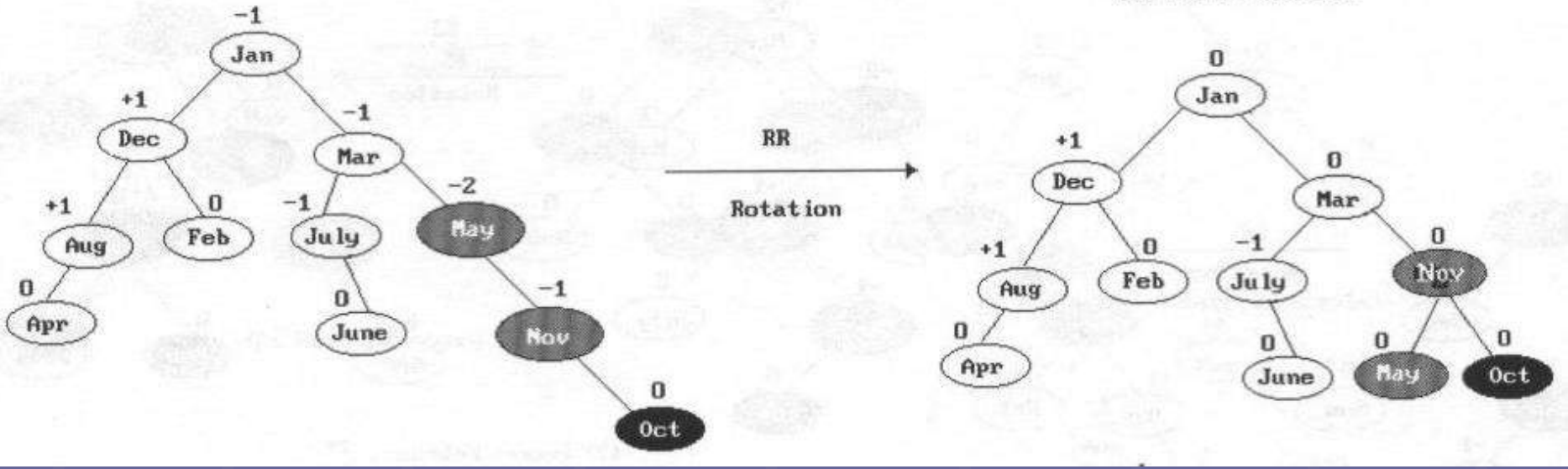
left_rotation(x);



AVL Trees

- Rebalancing rotations

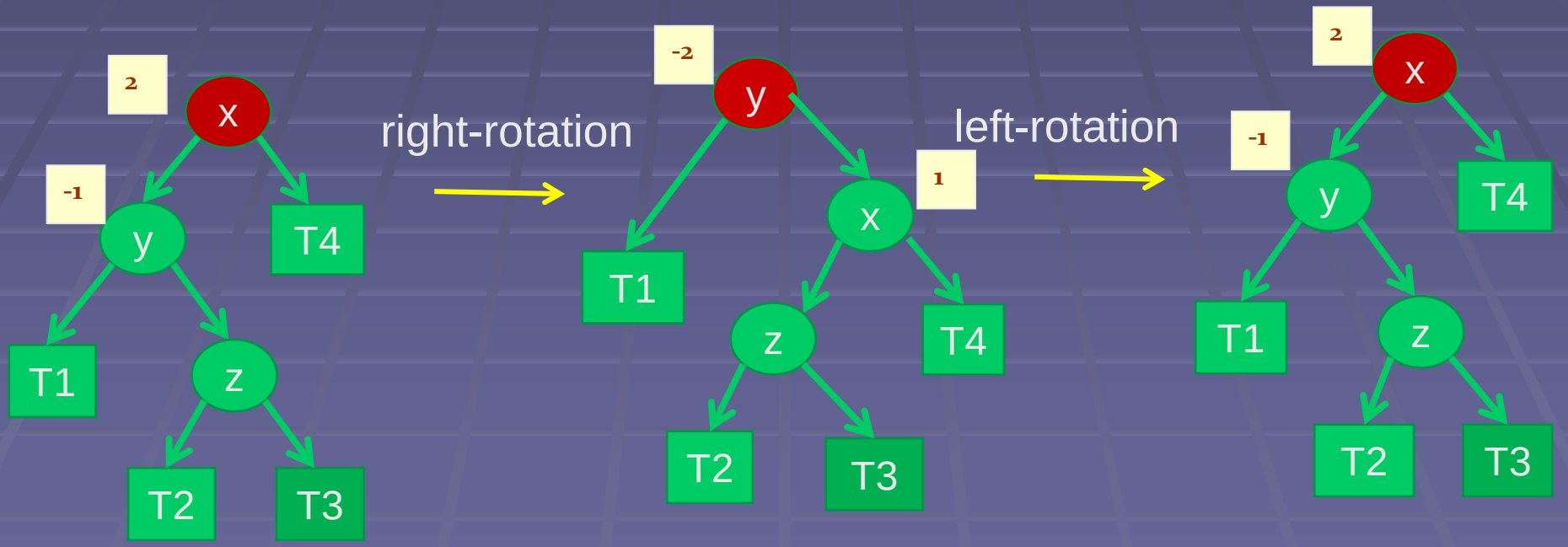




L-R-Rotation for Re-balancing

Case 3(LR): balance-factor(x) = 2, balance-factor(x->left) < 0

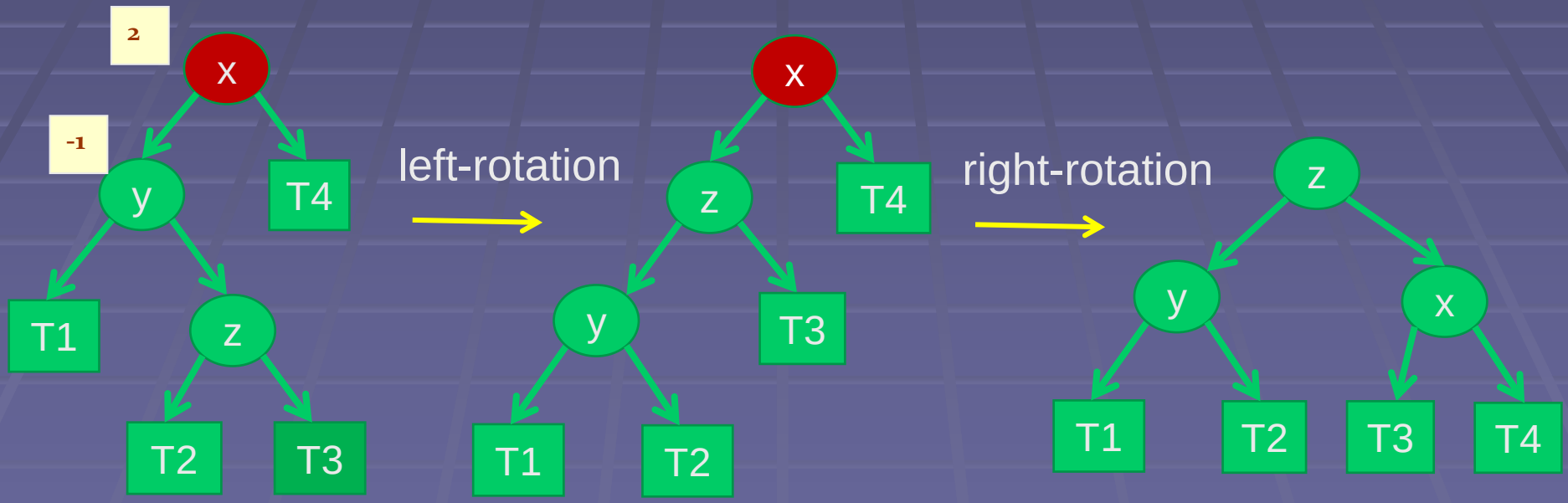
x->left = left_rotate(x->left); right_rotation(x);



L-R-Rotation for Re-balancing

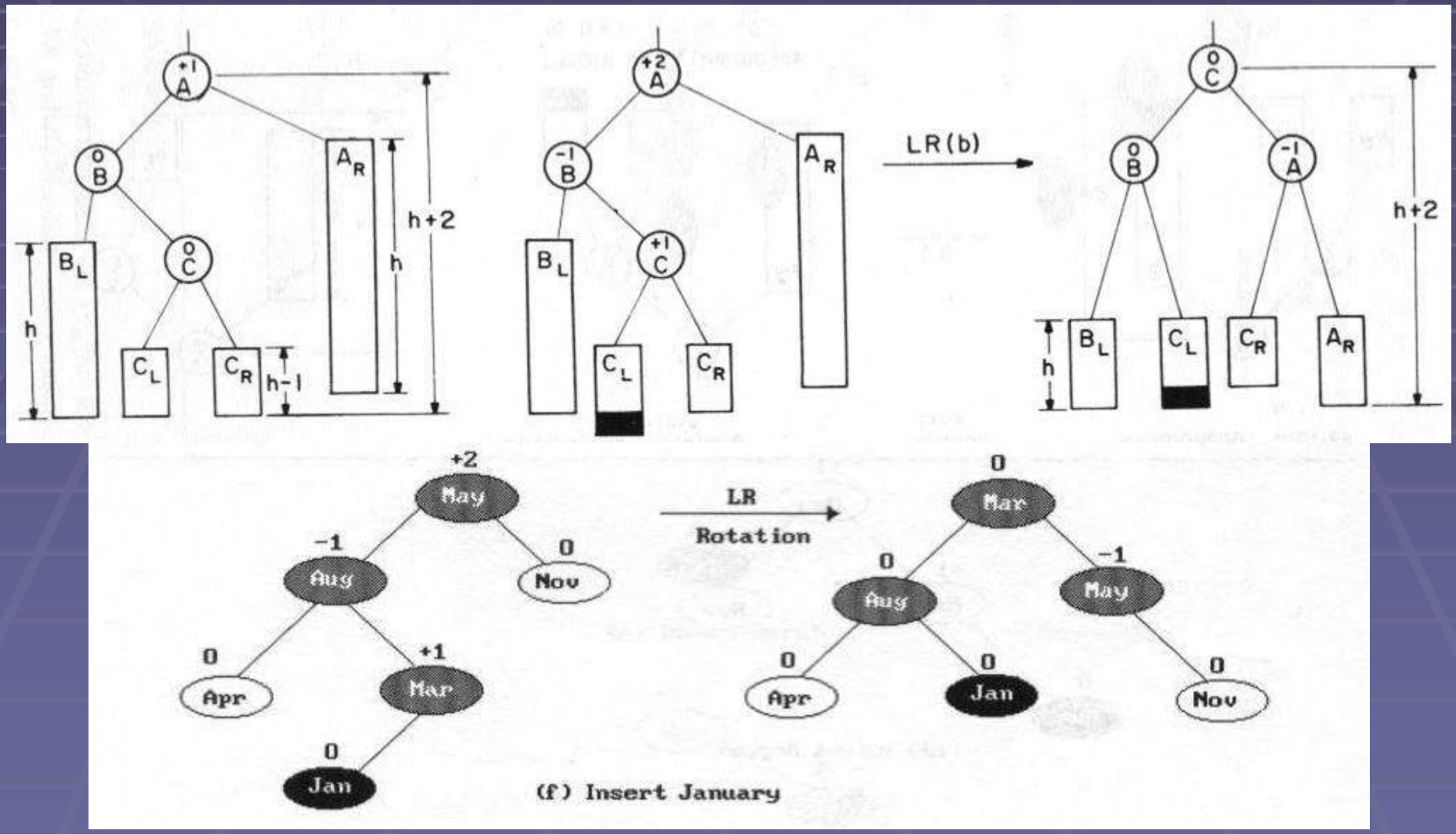
Case 3(LR) balance-factor(x) = 2, balance-factor(x->left) < 0

x->left = left_rotate(x->left); right_rotation(x);

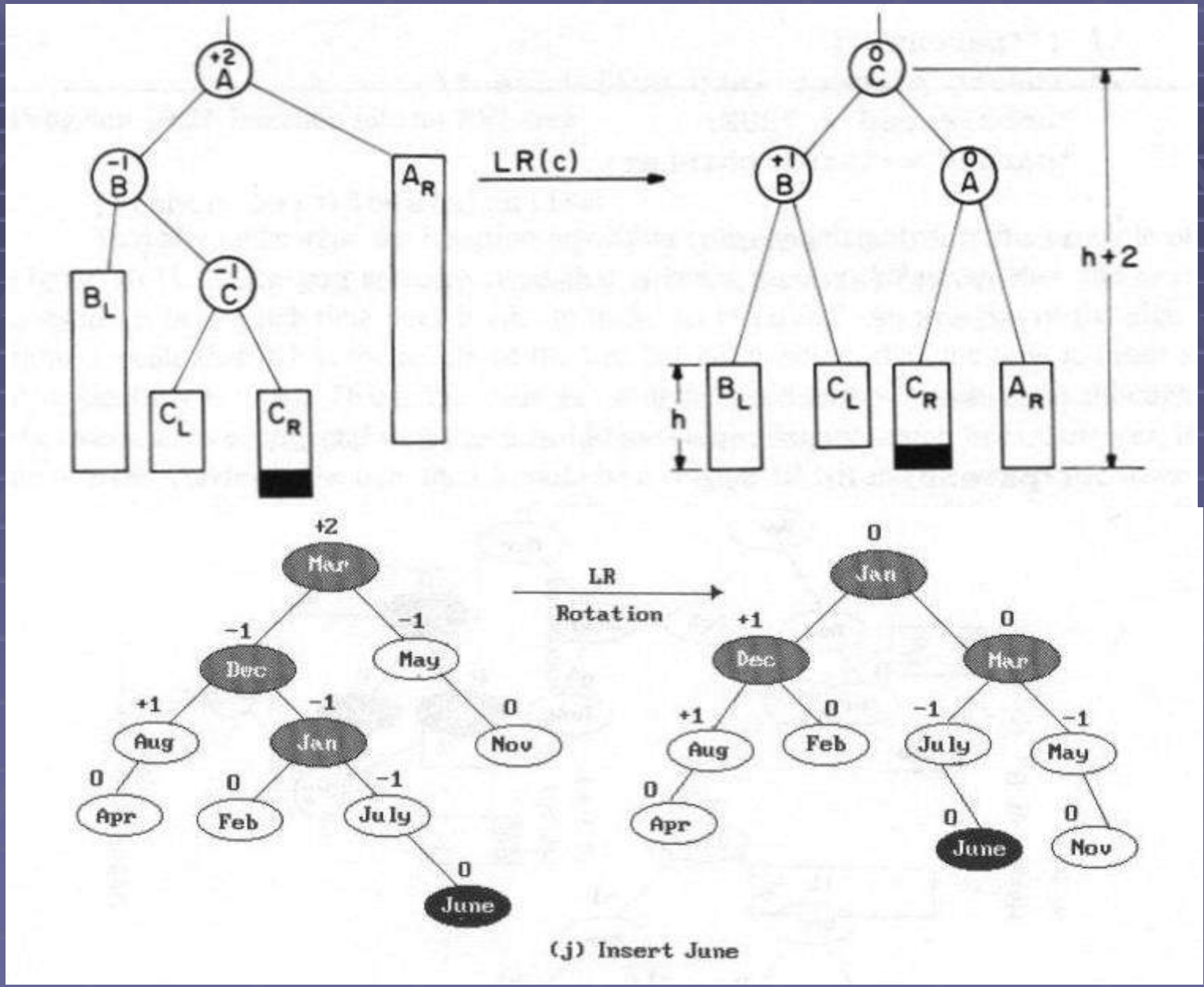


AVL Trees

- Rebalancing rotations (cont'd)

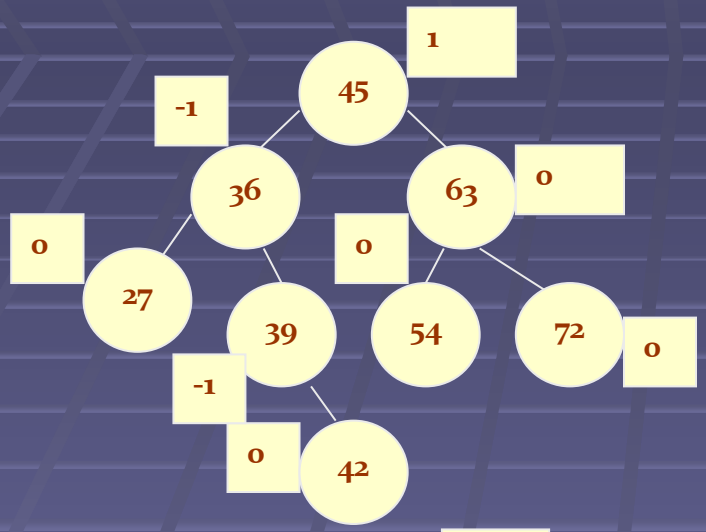


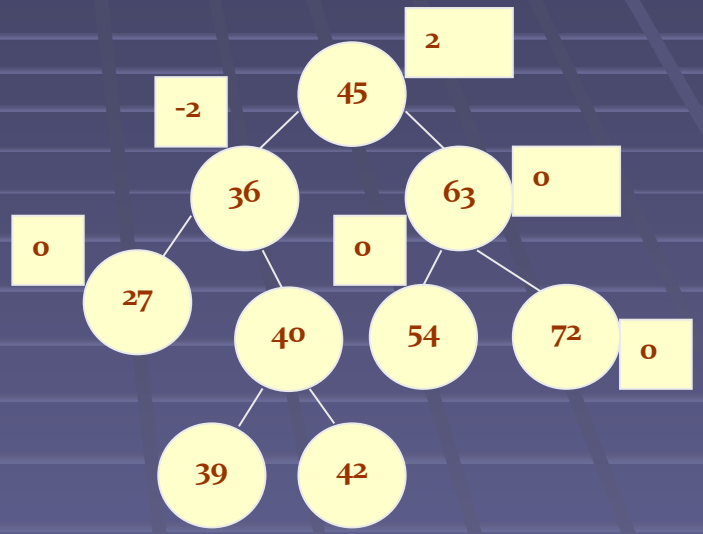
- Rebalancing rotations (cont'd)

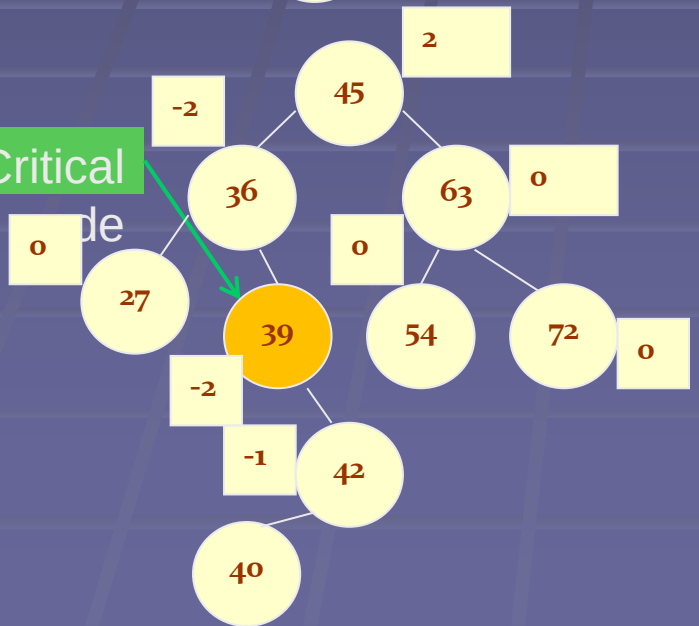


Case 4. R-L-Rotation to Balance

Example: Consider the AVL tree given below and insert 40 into it.







Critical
de

The tree is balanced using R-L rotation

```
void avl_insert(tree_pointer *parent, element x,
                                      int *unbalanced)
{
  if (!*parent) {
  /* insert element into null tree */
     *unbalanced = TRUE;
     *parent = (tree_pointer)
                              malloc(sizeof(tree_node));
     if (IS_FULL(*parent)) {
        fprintf(stderr, "The memory is full\n");
        exit(1);
     }
     (*parent)->left_child =
                 (*parent)->right_child = NULL;
     (*parent)->bf = 0;
     (*parent)->data = x;
  }
  else if (x.key < (*parent)->data.key) {
     avl_insert(&(*parent)->left_child, x, unbalanced);
     if (*unbalanced)
     /* left branch has grown higher */
        switch ((*parent)->bf) {
           case -1: (*parent)->bf = 0;
                    *unbalanced = FALSE;
                    break;
           case  0: (*parent)->bf = 1;
                    break;
           case  1: left_rotation(parent,unbalanced);
        }
  }
  else if (x.key > (*parent)->data.key) {
     avl_insert(&(*parent)->right_child, x, unbalanced);
     if (*unbalanced)
        /* right branch has grown higher */
        switch((*parent)->bf) {
           case 1 : (*parent)->bf = 0;
                    *unbalanced = FALSE;
                    break;
           case 0 : (*parent)->bf = -1;
                    break;
           case -1: right_rotation(parent, unbalanced);
        }
  }
        else {
           *unbalanced = FALSE;
           printf("The key is already in the tree");
        }
  }
```

Program 10.2: Insertion into an AVL tree

```
void left_rotation(tree_pointer *parent, int *unbalanced)
{
   tree_pointer grand_child, child;
   child = (*parent)->left_child;
   if (child->bf == 1) {
      /* LL rotation */
      (*parent)->left_child = child->right_child;
      child->right_child = *parent;
      (*parent)->bf = 0;
      (*parent) = child;
   }
   else {
   /* LR rotation */
      grand_child = child->right_child;
      child->right_child = grand_child->left_child;
      grand_child->left_child = child;
      (*parent)->left_child = grand_child->right_child;
      grand_child->right_child = *parent;
      switch(grand_child->bf) {
         case  1 : (*parent)->bf = -1;
                   child->bf = 0;
                   break;
         case 0:   (*parent)->bf = child->bf = 0;
                   break;
         case -1:  (*parent)->bf = 0;
                   child->bf = 1;
      }
      *parent = grand_child;
   }
   (*parent)->bf = 0;
   *unbalanced = FALSE;
}
```

Program 10.3: Left rotation function

AVL Trees

- **Complexity:**
  - In the case of binary search trees, if there were *n* nodes in the tree, then *h* (the height of tree) could be be *n* and the worst case insertion time would be O(*n*).
  - In the case of AVL trees, since *h* is at most (log *n*), the worst case insertion time is O(log *n*).
  - Figure 10.13 compares the worst case times of certain operations

Operation	Sequential list	Linked list	AVL tree
Search for x	$O(\log n)$	$O(n)$	$O(\log n)$
Search for kth item	$O(1)$	$O(k)$	$O(\log n)$
Delete x	$O(n)$	$O(1)^1$	$O(\log n)$
Delete kth item	$O(n-k)$	$O(k)$	$O(\log n)$
Insert x	$O(n)$	$O(1)^2$	$O(\log n)$
Output in order	$O(n)$	$O(n)$	$O(n)$

1. Doubly linked list and position of x known.
2. If position for insertion is known.

Deleting a Node from an AVL Tree

- Deleting algorithm
  - Step 1. delete node as BST
  - Step 2. if not height balanced, rebalance by rotation

If the resulting BST of step 1 is not height-balanced, find the critical node. There are four possible cases similar to the inserting. Then do corresponding rotation by the case

Rotate to rebalance

Case 1: balance-factor(x) = 2, balance-factor(x->left) >=0,

right_rotation(x);

Case 2: balance-factor(x) = 2, balance-factor(x->left) < 0,

x->left = left_rotate(x->left); right_rotation(x);

Case 3: balance-factor(x) = -2, balance-factor(x->right) <=0,

left_rotation(x);

Case 4: balance-factor(x) = -2, balance-factor(x->right) >0

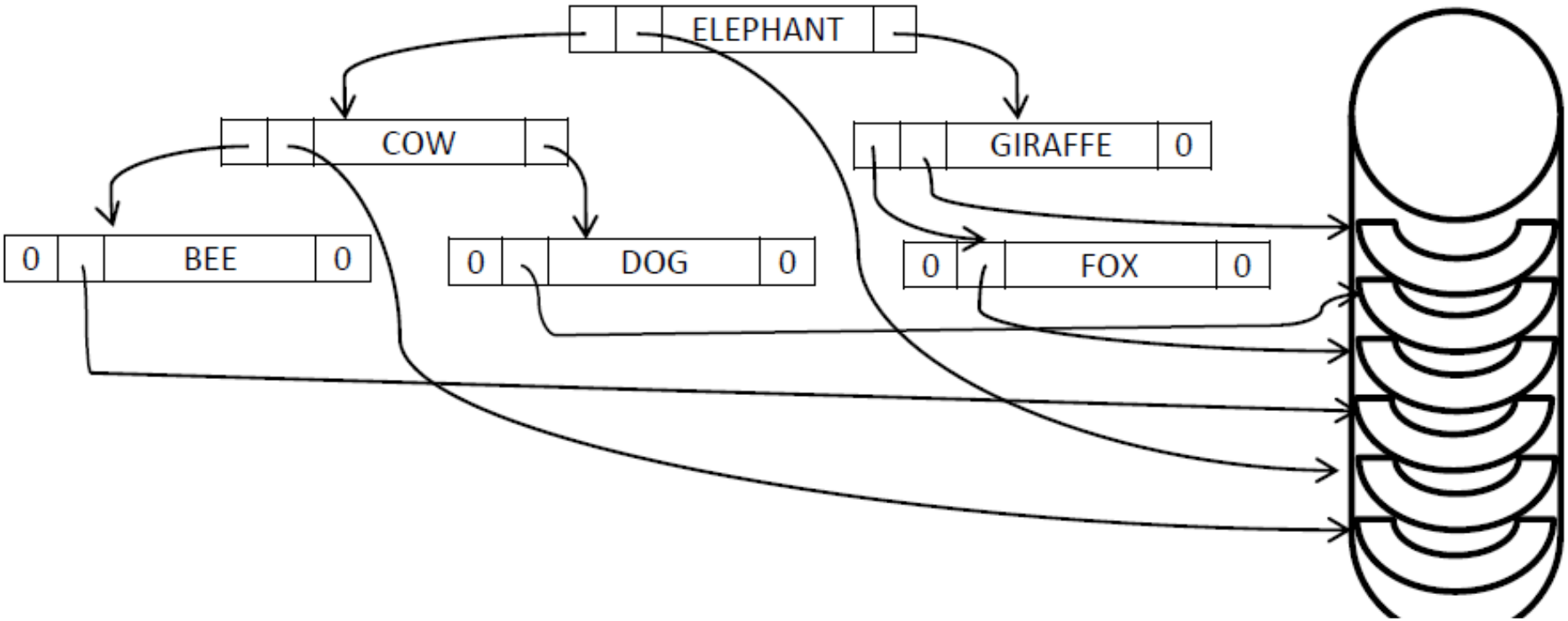
x->right = right_rotate(x->right); left_rotation(x);

Indexing

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute or set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form

- Index files are typically much smaller than the original file
- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across “buckets” using a “hash function”.

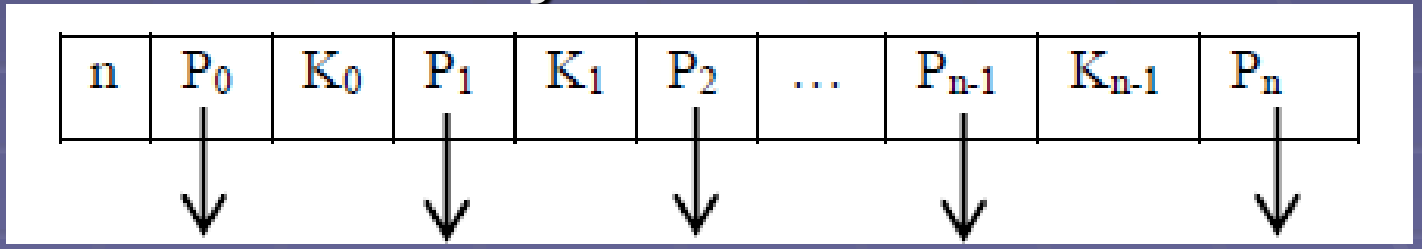
BST AS INDEX



Just as a binary search tree AVL tree can be used as an index. The maximum path length through an AVL tree is $O(log_2 n)$ to locate the address of the sought record

M-WAY SEARCH TREE

- The performance of an index can be enhanced significantly by increasing the branching factor of the tree.
- Rather than binary branching, m-way (m>2) branching can be used.
- Each node of the m-way search tree has the structure



- The $P_0, P_1, \ldots, P_n$ are poiters to the node's subtree and the $K_0, \ldots, K_{n-1}$ are key values. The requirement that each node have out-degree < = m forces n < = m – 1.

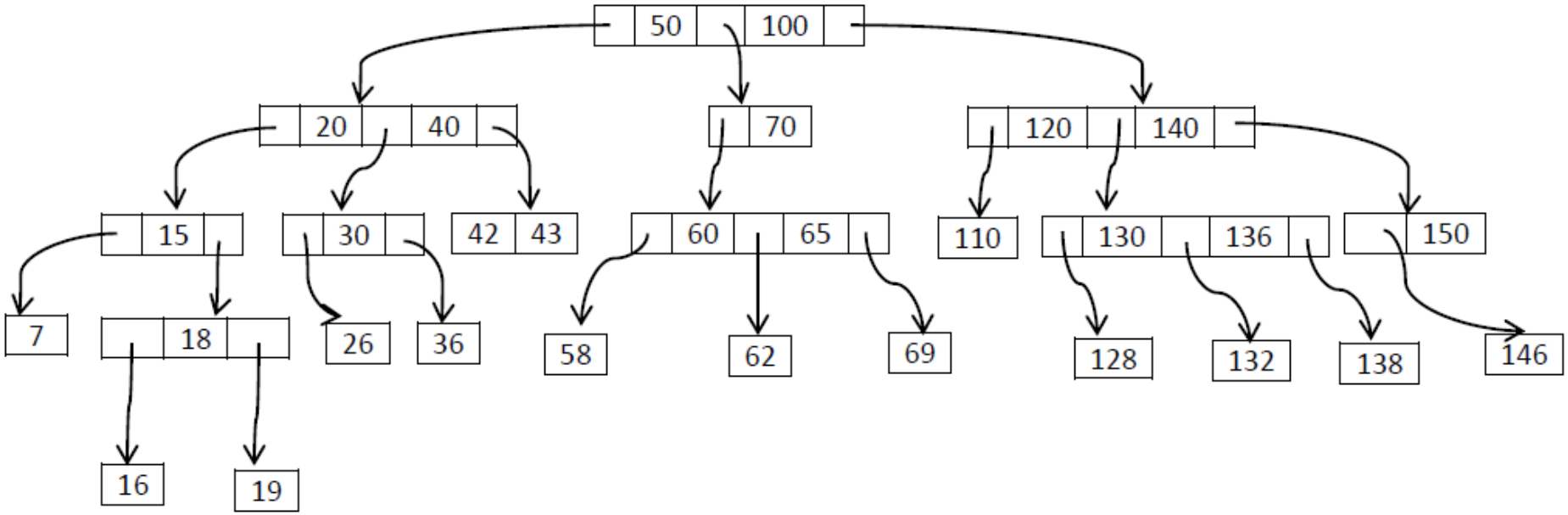
M-way search tree

- The key values in a node are in the ascending order:

  $K_i < K_{i+1}$ for i = 0, …, n – 2.

- All key values in nodes of the subtree pointed to by P_i are less than the key value K_i for $i = 0, \ldots, n - 1$.
- All key values in nodes of the subtree pointed to by P_n are greater than the key value K_{n-1}.
- The subtrees pointed by the P_i, $i = 0, \ldots, n$ are also m-way search trees.

Example: 3-way search tree



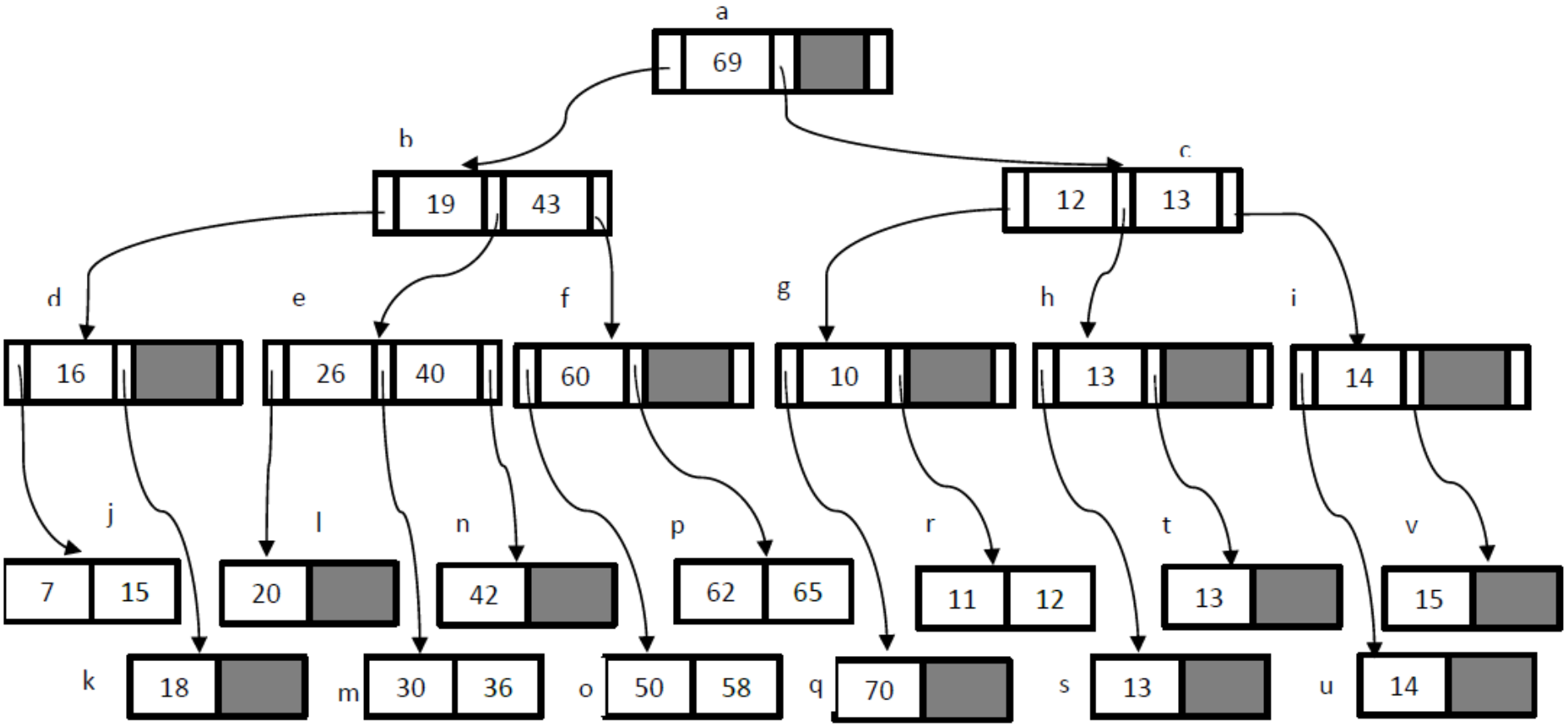
The maximum search length of an optimal m-way tree of n-keys, then, is (log_m^n)

B-TREES

- A B-tree of order m is an m-way search tree with the following properties.
  - Each node of the tree, except for the root and the leaves, has at least $[\frac{1}{2}m]$ subtrees and no more than m subtrees.
  - The root of the tree has at least two subtrees, unless it is itself a leaf.
  - All leaves of the tree are on the same level.

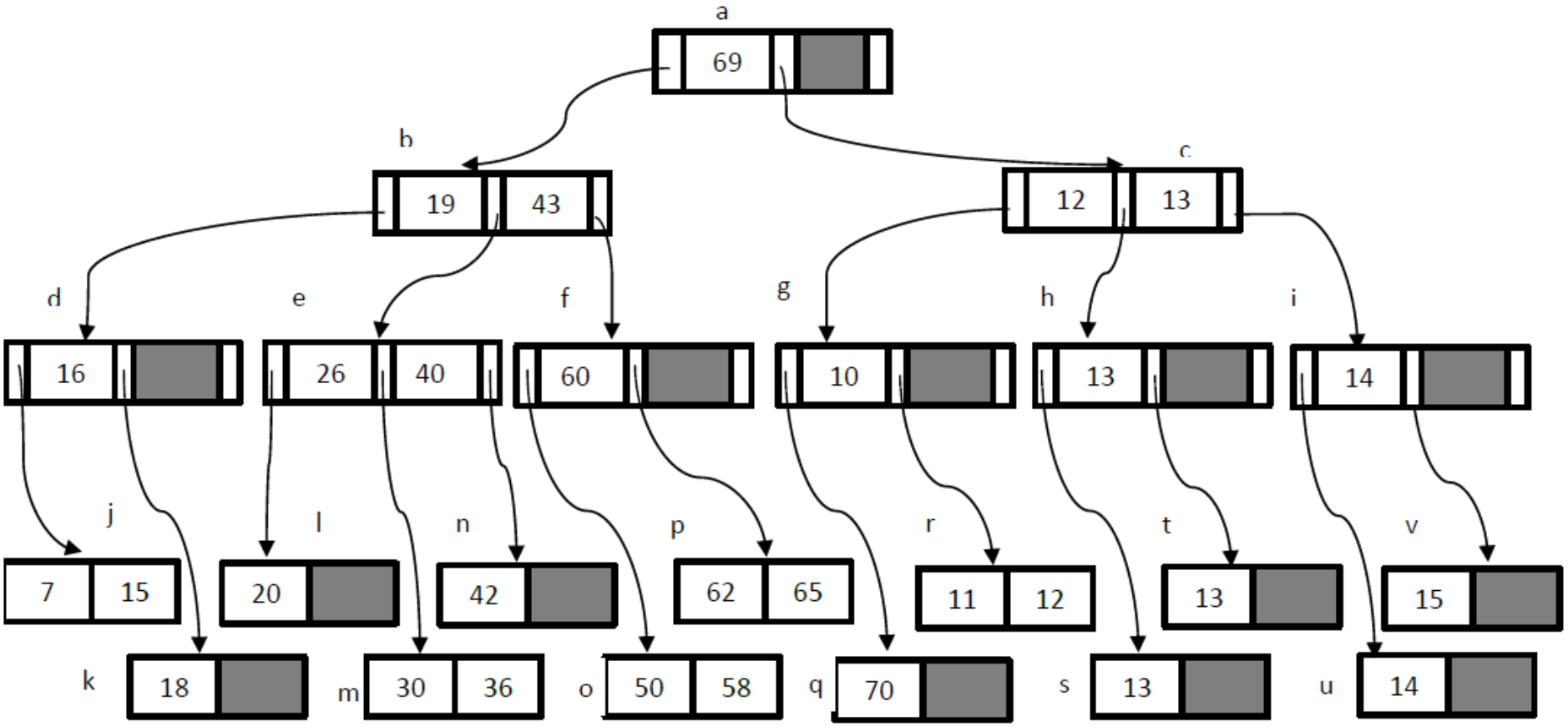
The first constraint ensure that each node of the tree is at least half full. The second constraint forces the tree to branch early. The third constraint keeps the tree nearly balanced.

3-way B-tree

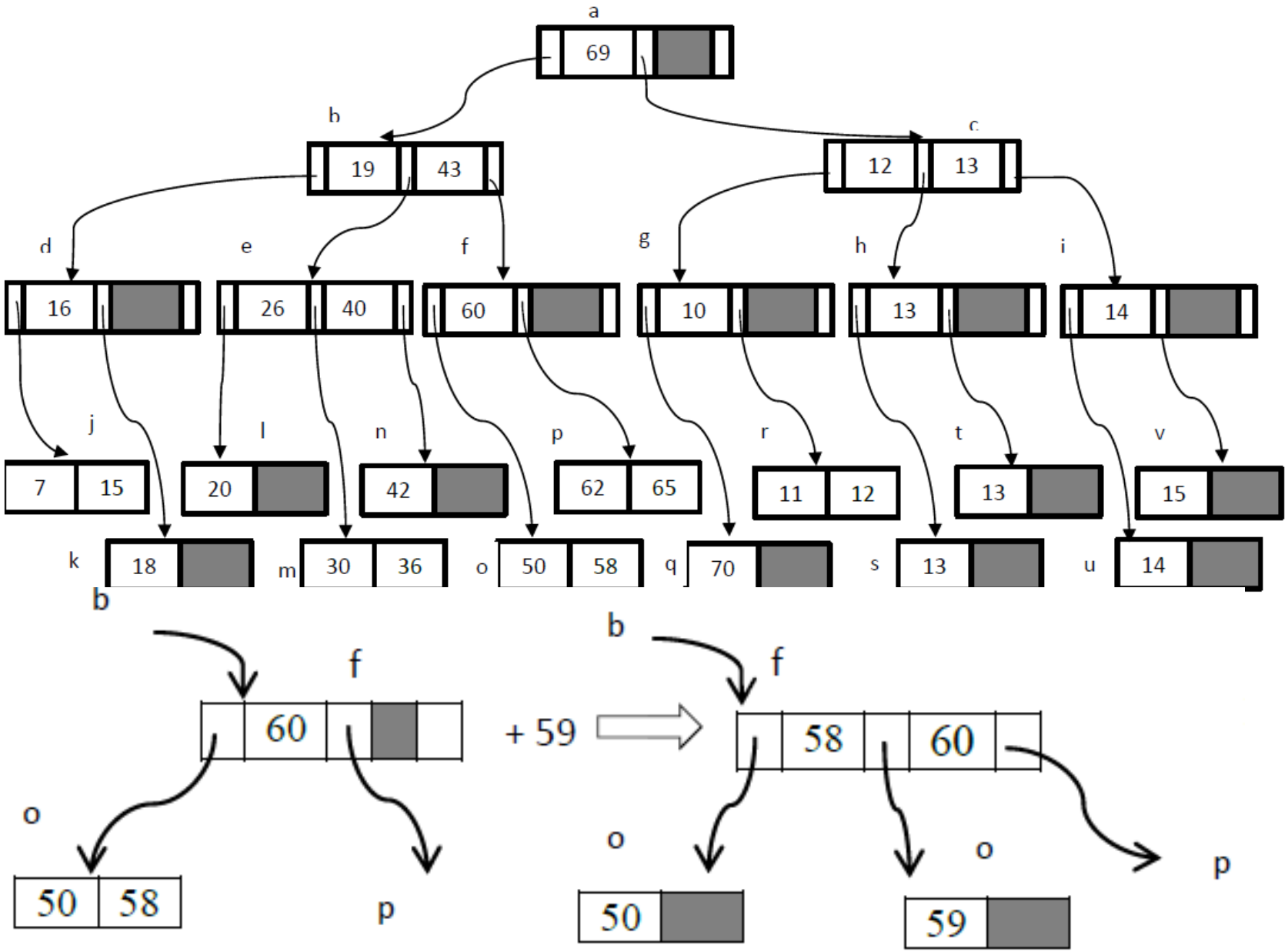


B-tree of order 3 containing keys of the previous 3-way search tree

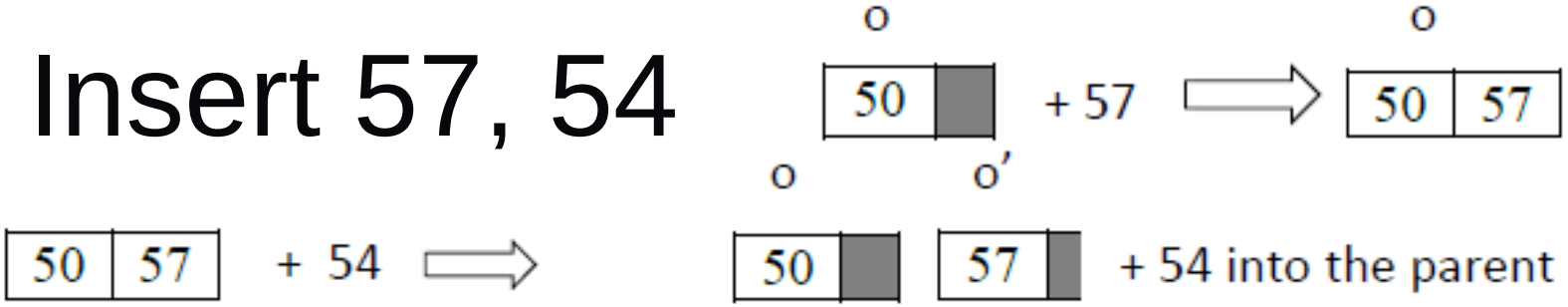
Insert 22 and 41

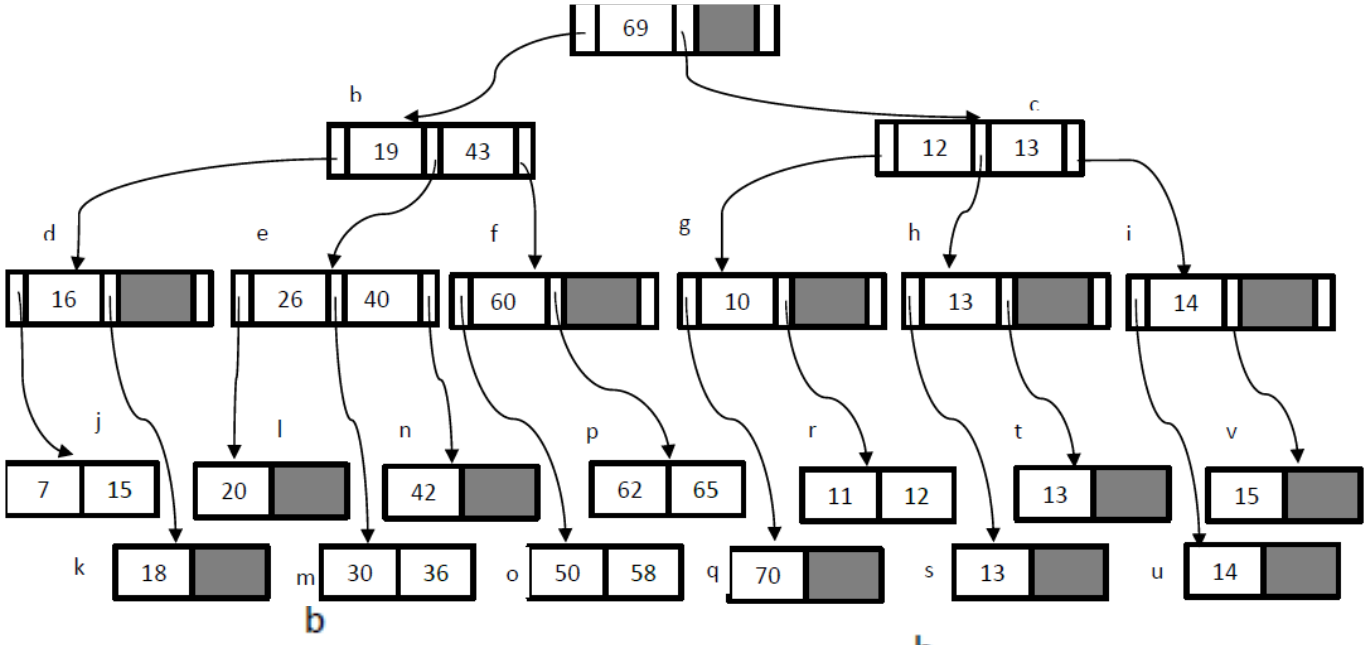


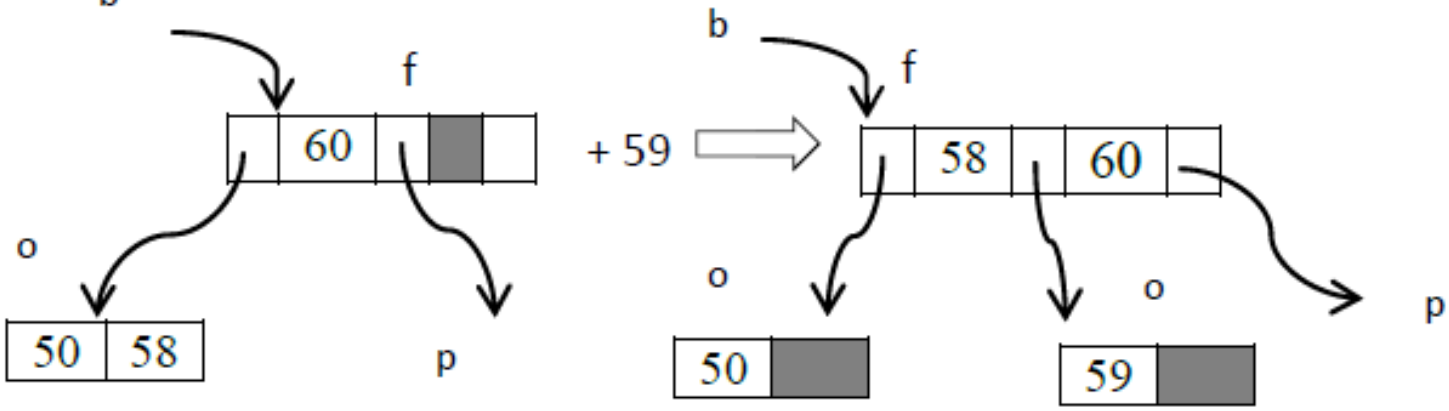
Insert 59


a
69
b
19
43
c
12
13
d
16
e
26
40
f
60
g
10
h
13
i
14
j
7
15
l
20
n
42
p
62
65
r
11
12
t
13
v
15
k
18
m
30
36
o
50
58
q
70
s
13
u
14
b
f
60
o
50
58
p
+ 59
b
f
58
60
o
50
o
59
p

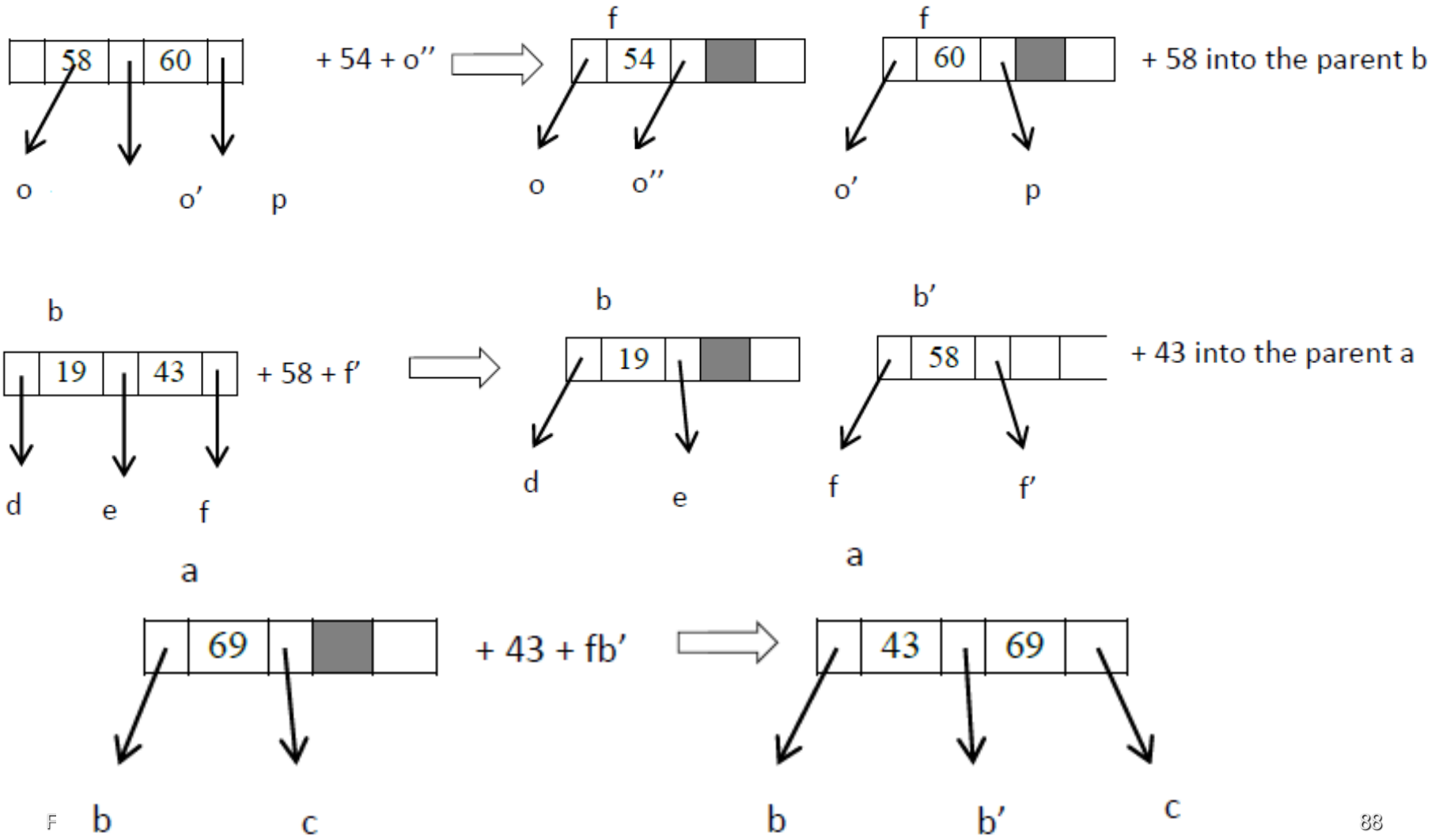
Insert 57, 54


o
50
+ 57
o
50
57
o
o'
50
57
+ 54 into the parent
50
57
+ 54

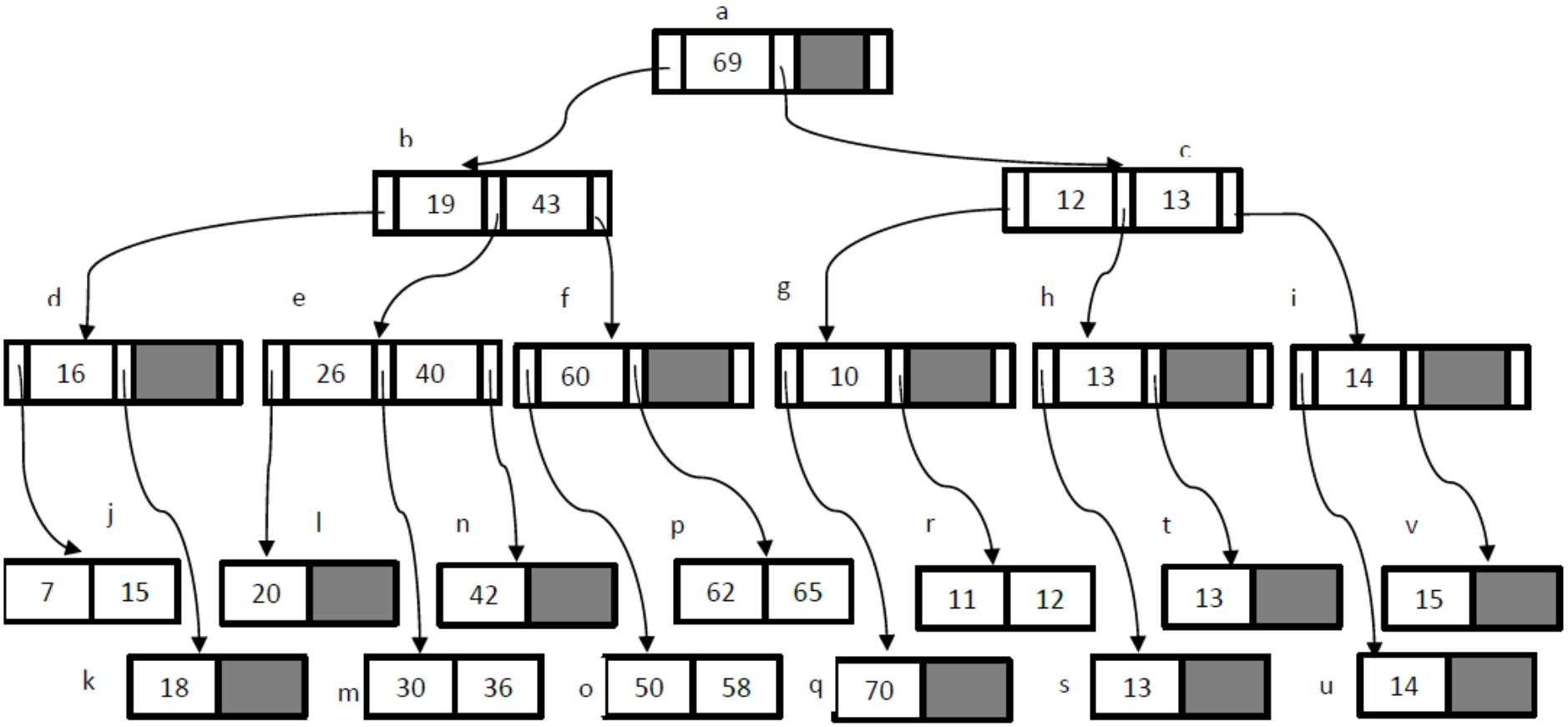

69
b
19
43
c
12
13
d
e
f
g
h
i
16
26
40
60
10
13
14
j
l
n
p
r
t
v
7
15
20
42
62
65
11
12
13
15
k
18
m
30
36
o
50
58
q
70
s
13
u
14
b


f
60
o
50
58
p
+ 59
b
f
58
60
o
50
o
59
p

Insert 58, 43

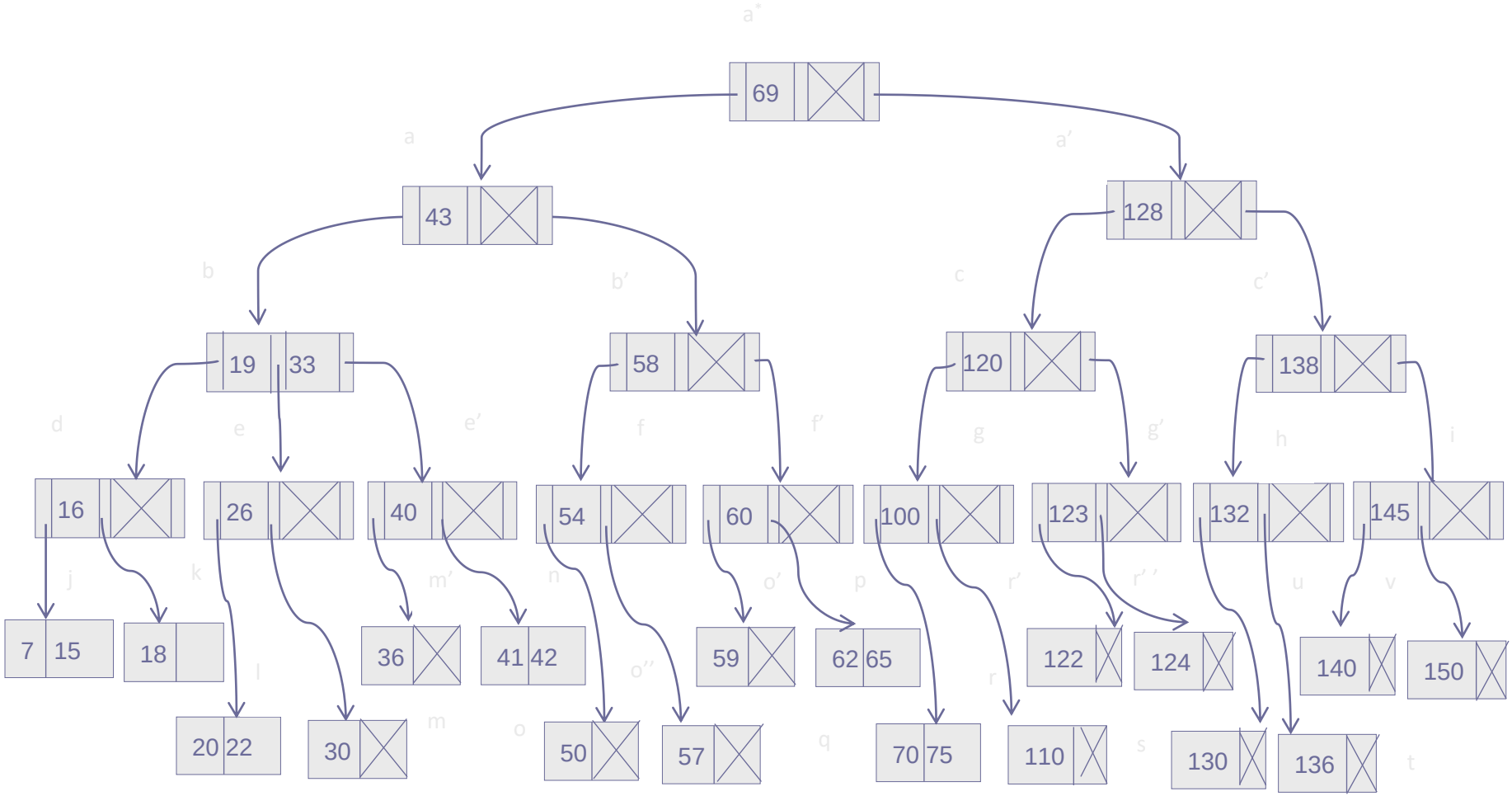

58
60
o
o'
p
+ 54 + o''
f
54
o
o''
f
60
o'
p
+ 58 into the parent b
b
19
43
d
e
f
a
+ 58 + f'
b
19
d
e
b'
58
f
f'
a
+ 43 into the parent a
69
b
c
+ 43 + fb'
43
69
b
b'
c

3-way B-tree

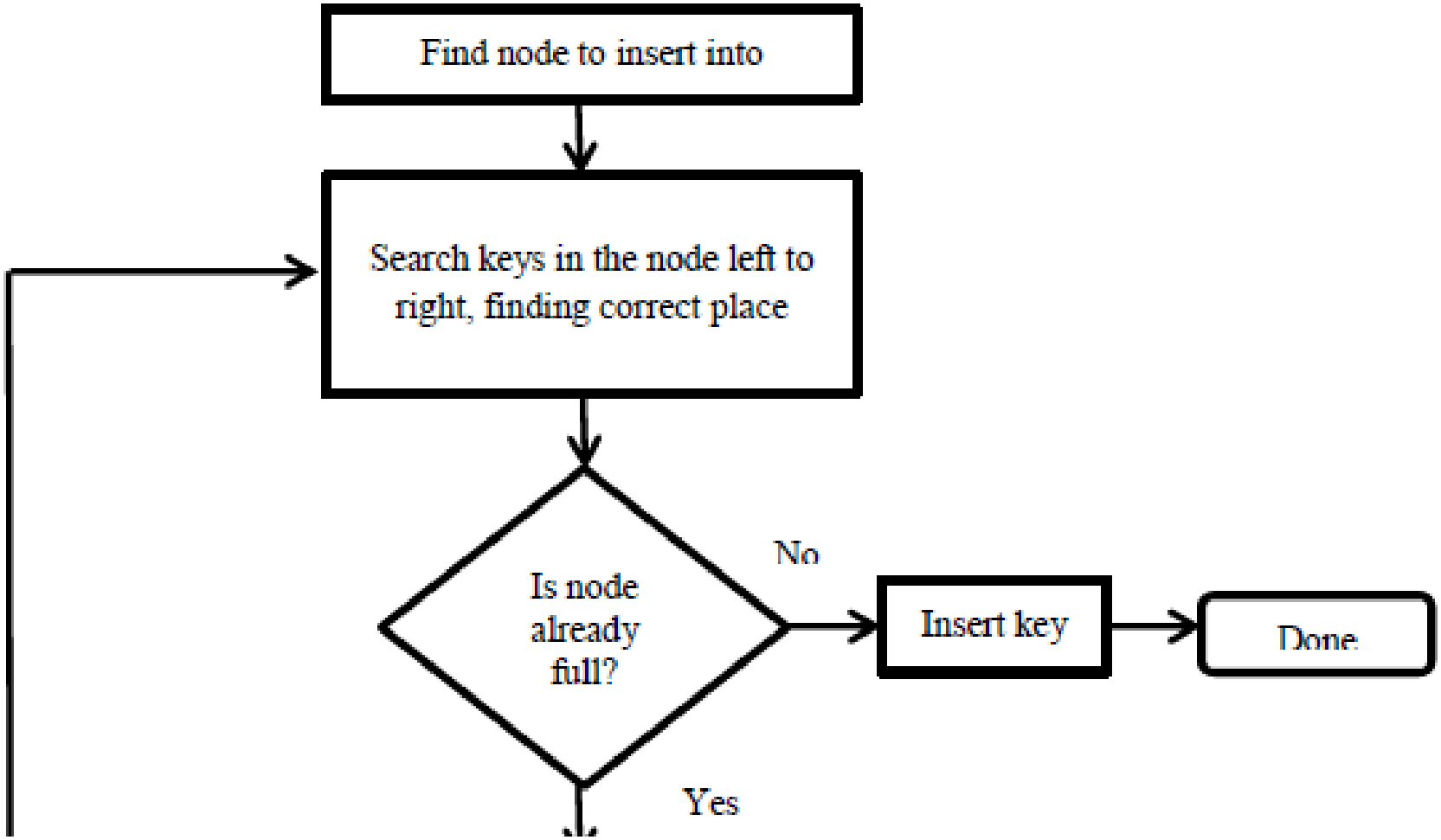


B-tree of order 3 containing keys of the previous 3-way search tree

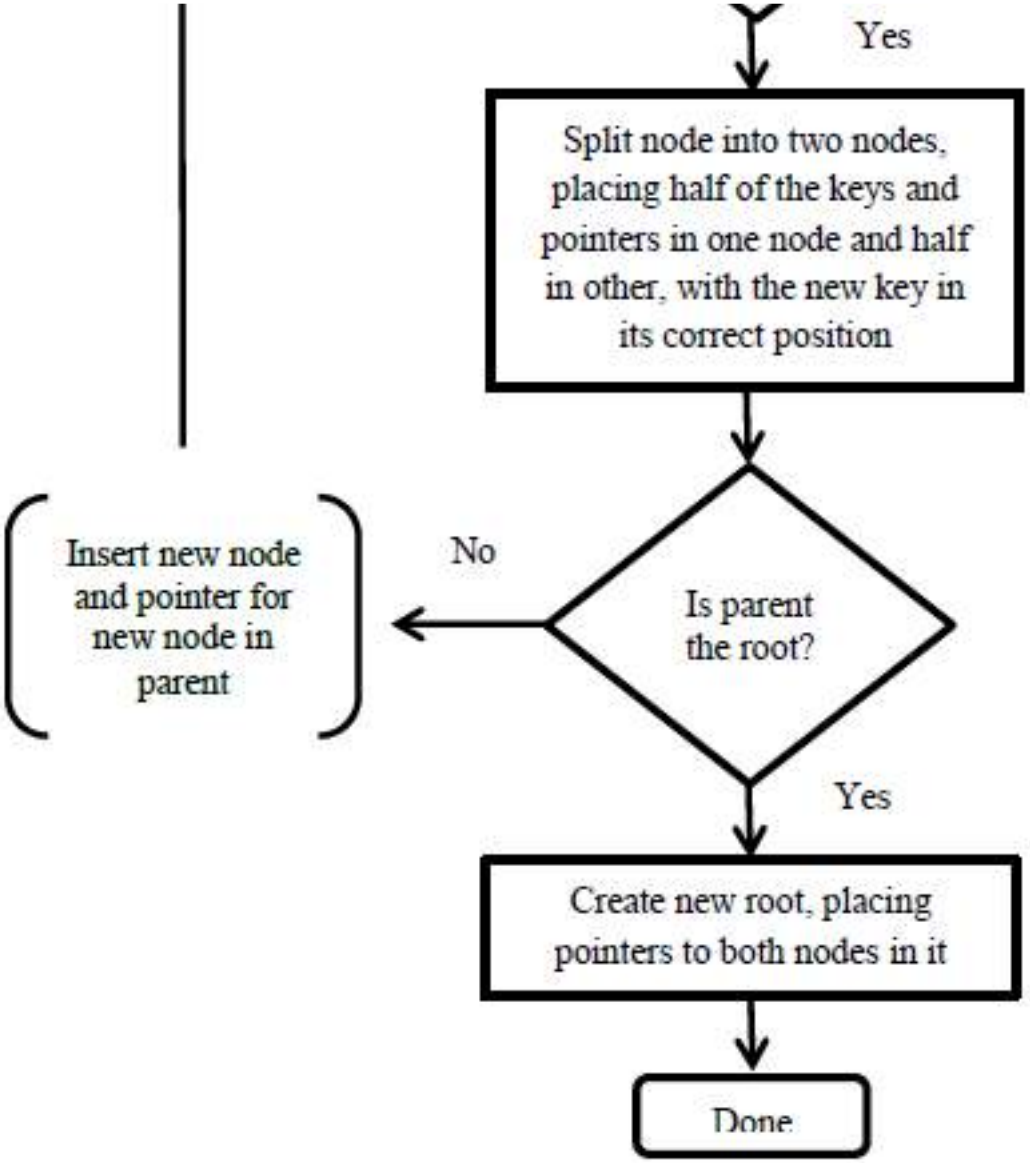
Insert 22, 41, 59, 57, 54, 33, 75, 124, 122, 123



Insertion in a B-Tree



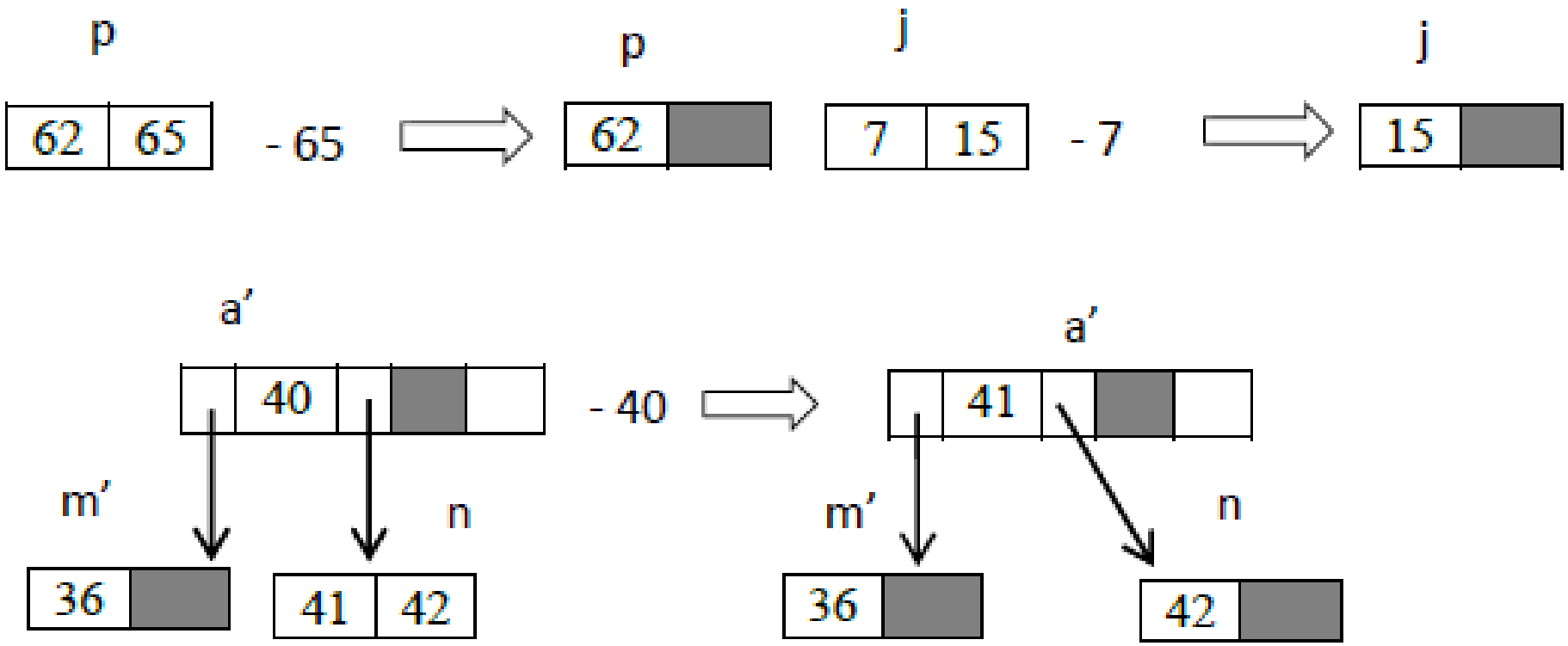
Insertion in a B-Tree



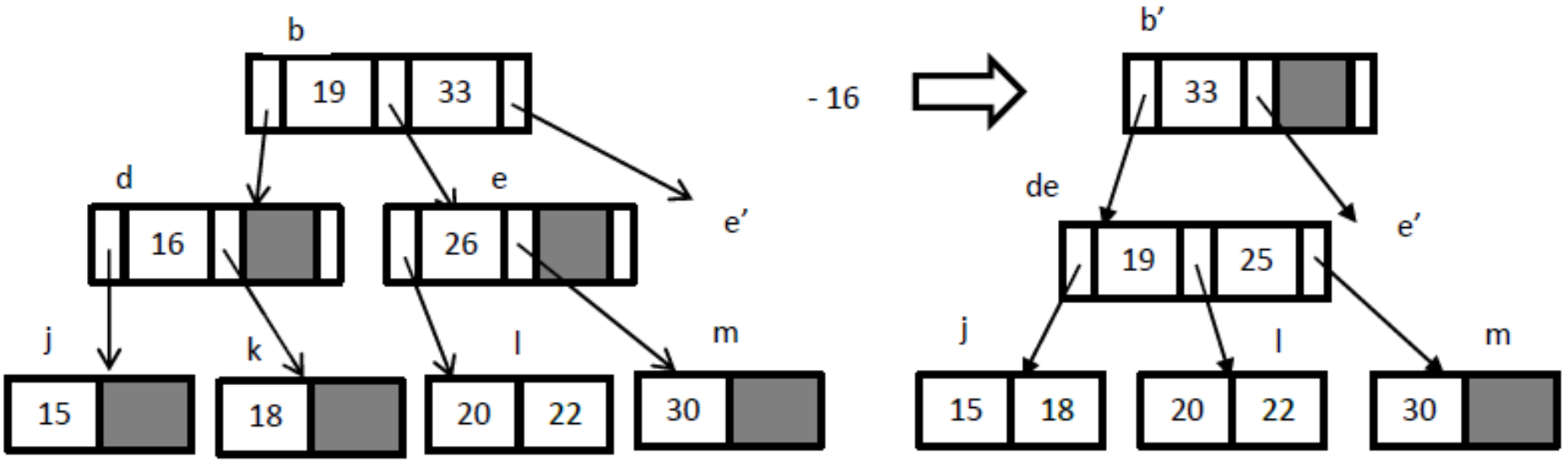
DELETING FROM A B-TREE

- Deletion of a key from a B-tree is only slightly more complicated than is insertion.
- In order to keep the B-tree a B-tree, two nodes need to be merged when one node would have fewer than the minimum required keys and pointers.
- If a merge percolates all the way to the higher level, a new root is formed and the height of the B-tree decreased by 1.

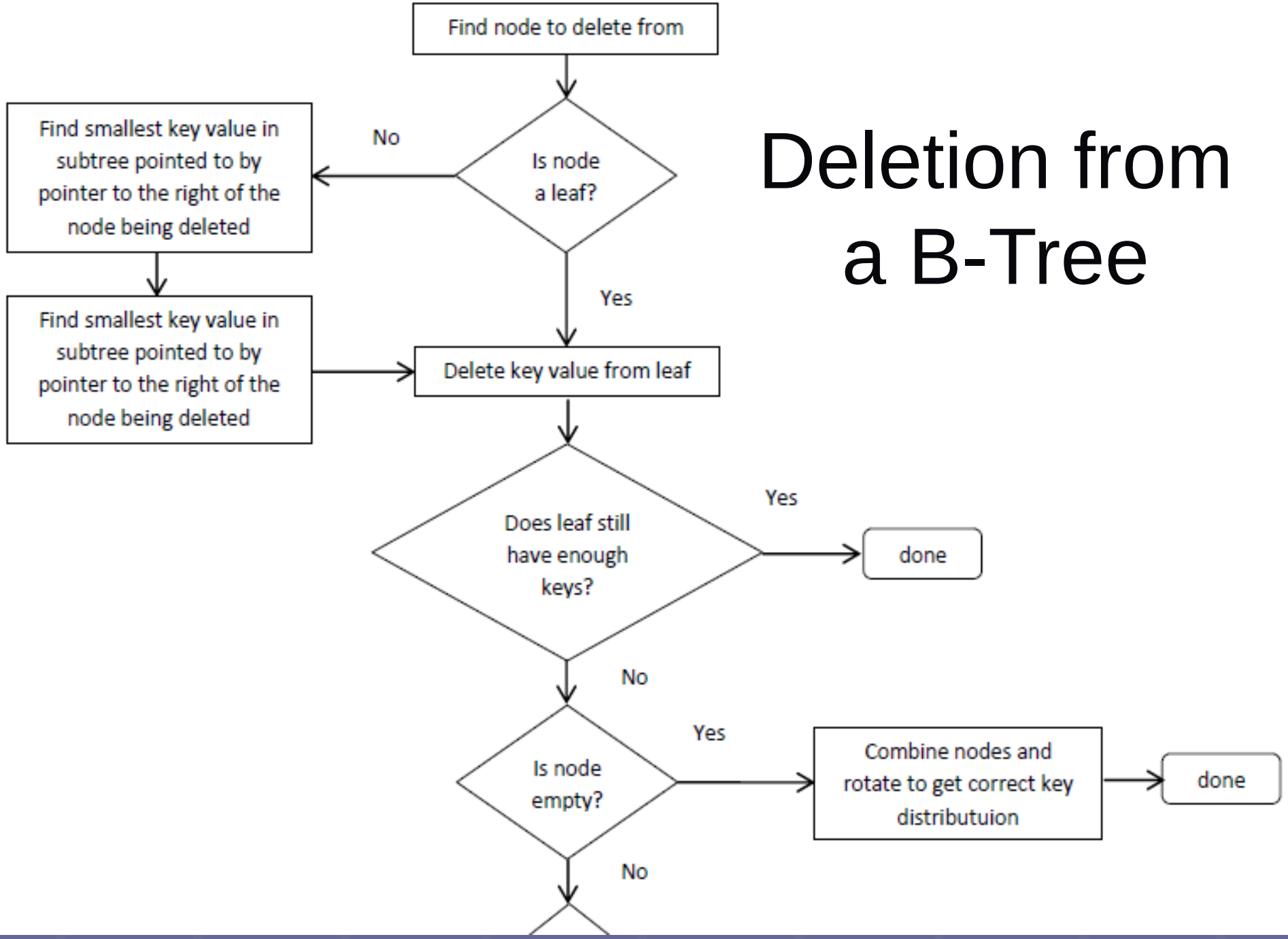
Delete 65, 7, 40

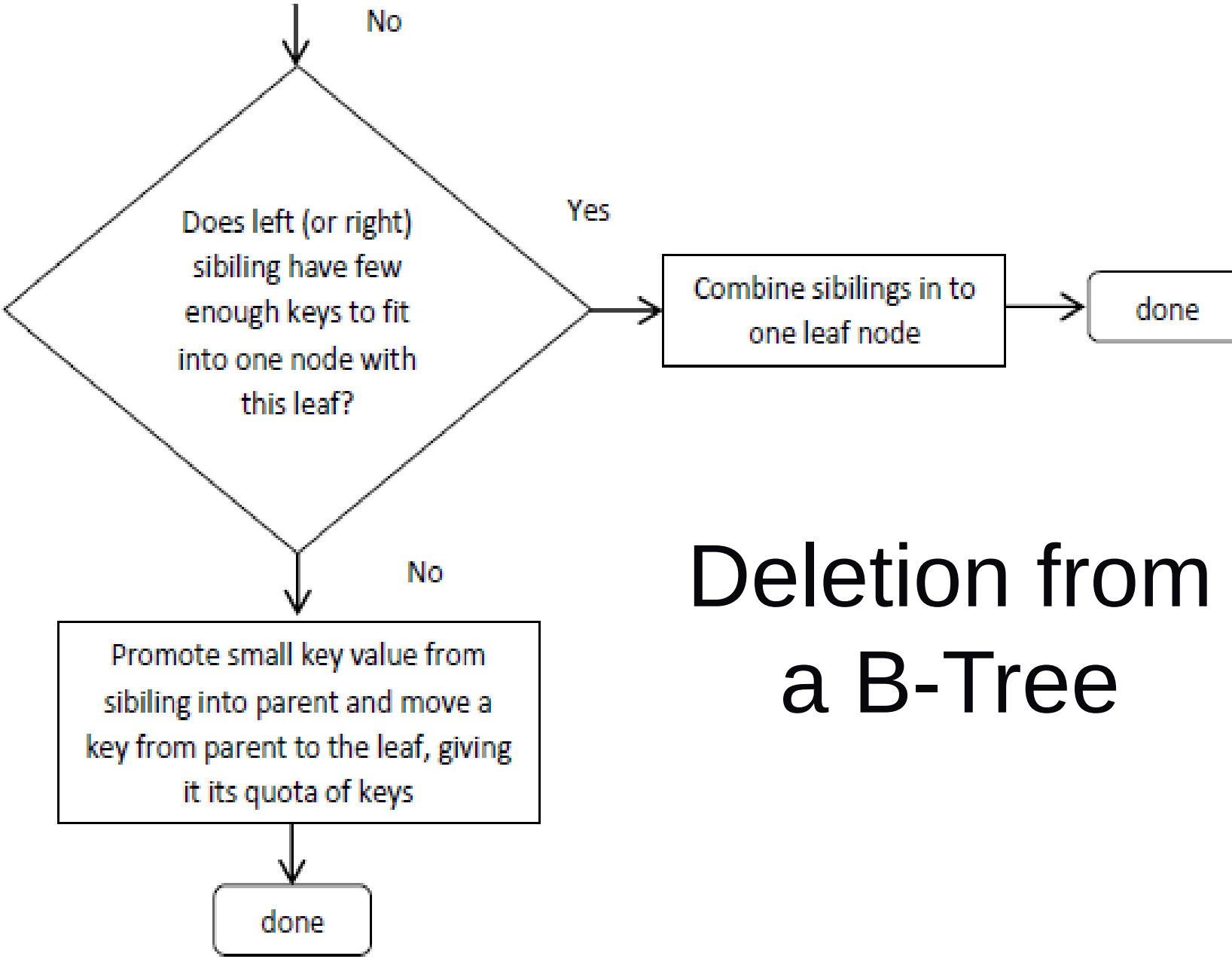


Delete 16


b
19
33
d
16
e
26
e'
j
15
k
18
l
20
22
m
30
- 16
b'
33
de
19
25
e'
j
15
18
l
20
22
m
30

Deletion from a B-Tree



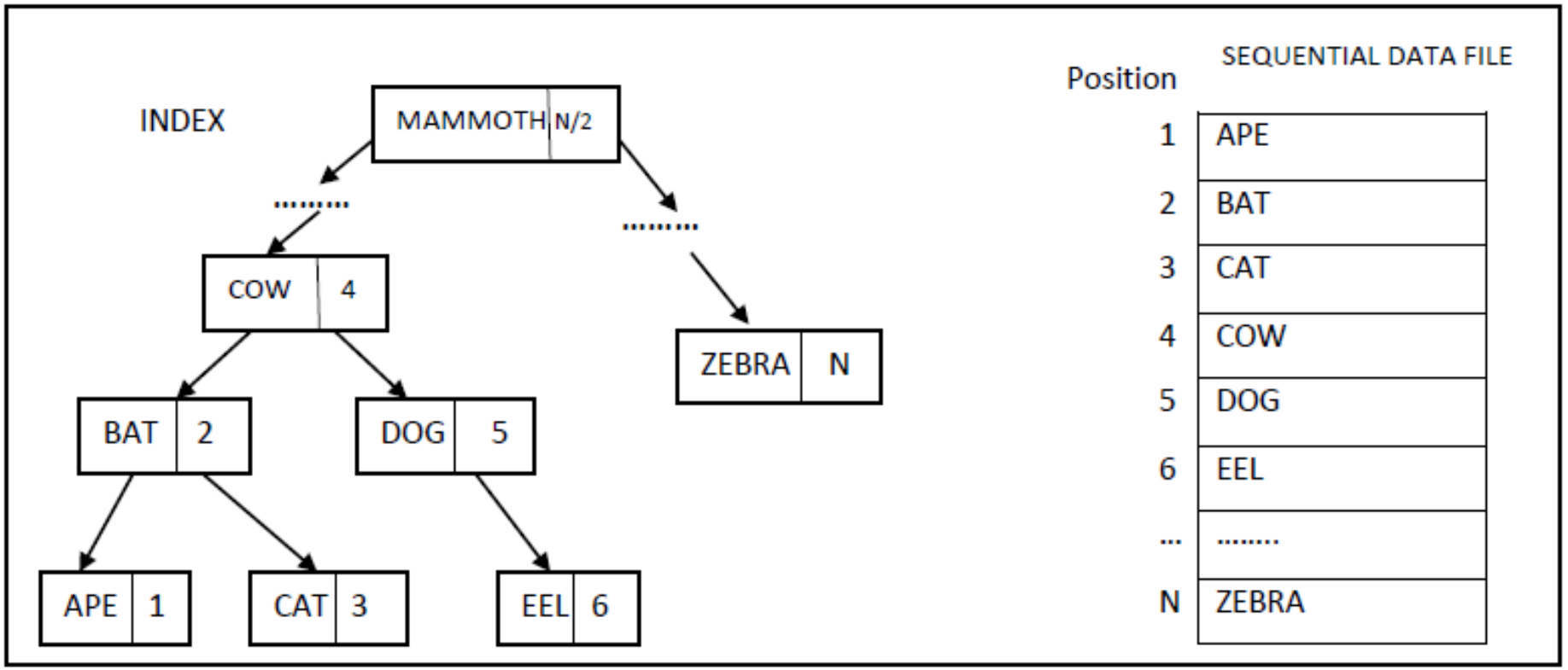


Deletion from a B-Tree

B^+ tree

- An effective way to organize a collection of records when there is the need both to access the records sequentially by some key value and also to access the records individually by that same key is indexed sequential file organization.
- An indexed sequential file provides the combination of access types that are supported by a sequential file and a relative file.

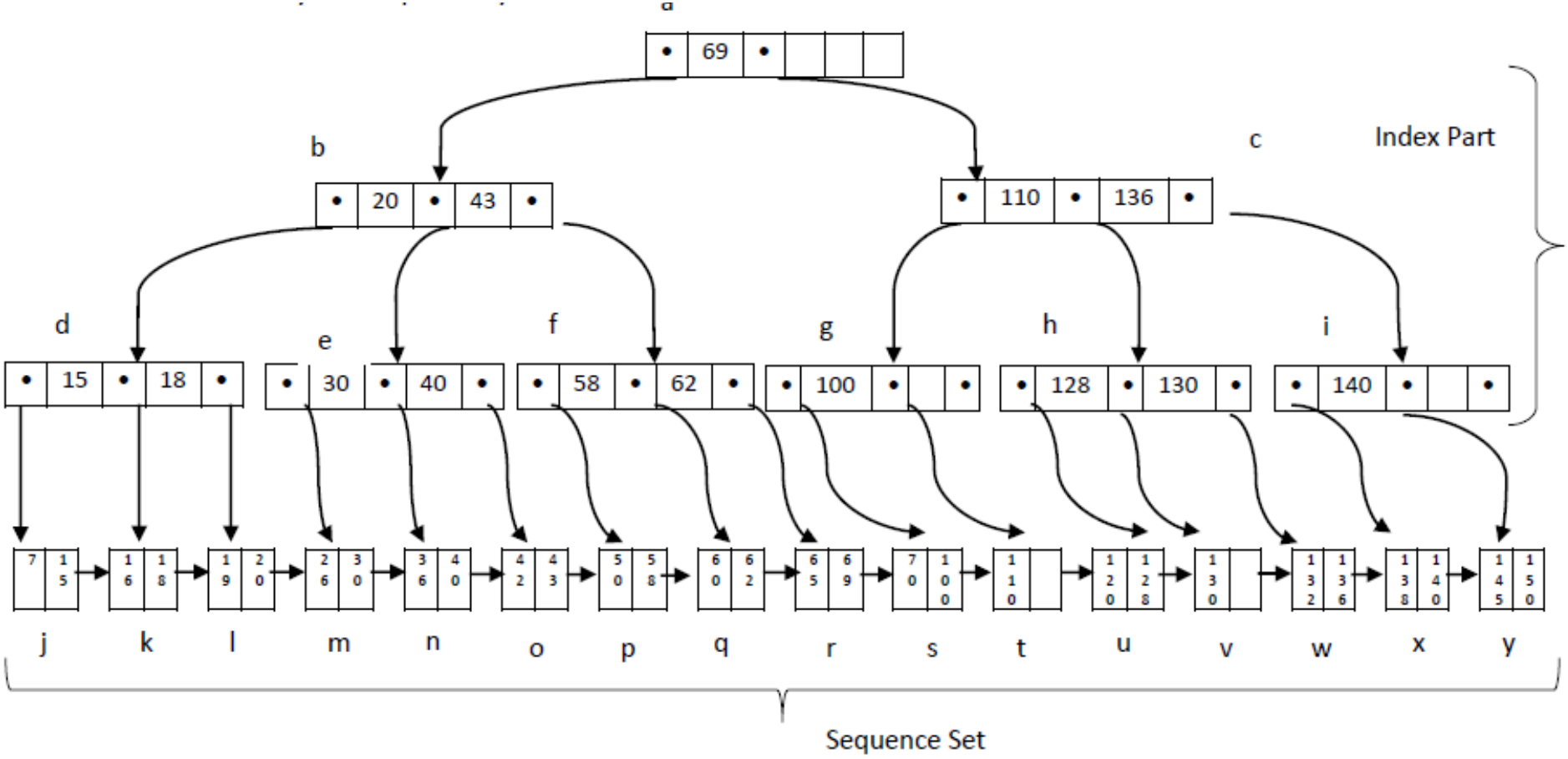
Use of a binary search tree and a sequential file to provide indexed sequential file



B^+ tree

- One of the popular techniques for implementing indexed sequential file organization is to use a variation on the basic B-tree as the B^+-tree.
- B^+ - tree contains a new set of pointers: the leaves have been connected to form a linked list of the keys in sequential order.
- The B^+ -tree has two parts: the *index* part is the interior nodes; the *sequence* set is the leaves.
- The linked leaves are an excellent aspect of a B^+ -tree; the keys can be accessed efficiently both directly and sequentially.

B^+ tree for the B-tree



Insertion: B^+ tree

- Inserting a new key value into a B^+-tree is done in almost the same way as insertion of a new key value into a basic B-tree.
- When a leaf node is split into two nodes, a copy of a low-order key value from the rightmost node is prompted to be the separator key value in the present node.
- The new node also must be inserted in the linked list of the sequence set.

Deletion: B^+ tree

- Deleting a key value from a B^+-tree is somewhat easier than if the tree were a basic B-tree.
- When a key value is deleted from a leaf, there is no need to delete that key value from the index of the tree.
- That key value still can direct searches to the proper leaves.