KEY INSERTION SEQUENCE: B C G E F D A & J.

SPLITTING AND PROMOTING

FIGURE 9.14 ▪ Initial leaf of a B-tree with a page size of seven.

FIGURE 9.15 ▪ Splitting the leaf to accommodate the new *J* key.

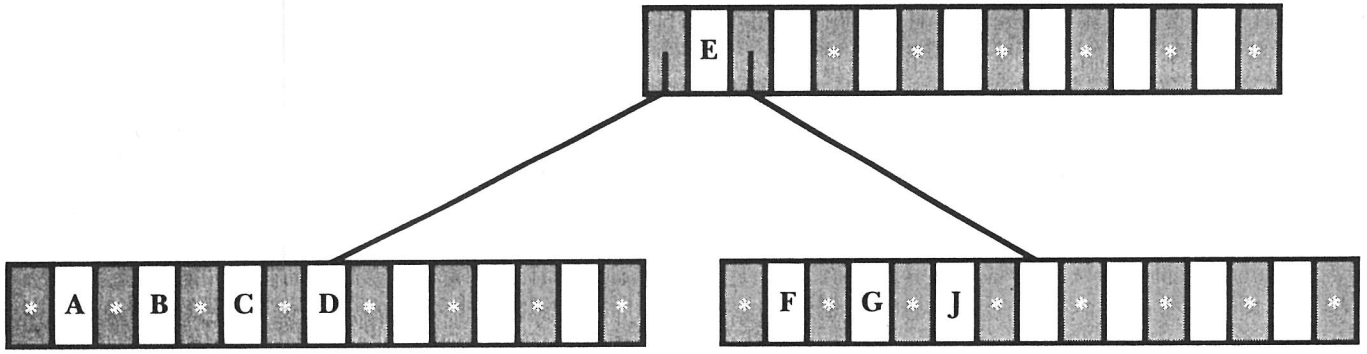


FIGURE 9.16 ▪ Promotion of the *E* key into a root node.

ALGORITHMS FOR B-TREE SEARCHING

```
FUNCTION: search (RRN, KEY, FOUND_RRN , FOUND_POS)

    if RRN == NIL then   /* stopping condition for the recursion */
        return NOT FOUND
    else
        read page RRN into PAGE
        look through PAGE for KEY, setting POS equal to the
            position where KEY occurs or should occur.
        if KEY was found then
            FOUND_RRN := RRN      /* current RRN contains the key */
            FOUND_POS := POS
            return FOUND
        else /* follow CHILD reference to next level down  */
            return(search(PAGE.CHILD[POS], KEY, FOUND_RRN, FOUND_POS))
        endif
    endif

end FUNCTION
```

FIGURE 9.20 ▪ Function *search (RRN, KEY, FOUND_RRN, FOUND_POS)* searches recursively through the B-tree to find KEY. Each invocation searches the page referenced by RRN. The arguments FOUND_RRN and FOUND_POS identify the page and position of the key, if it is found. If *search()* finds the key, it returns FOUND. If it goes beyond the leaf level without finding the key, it returns NOT FOUND.

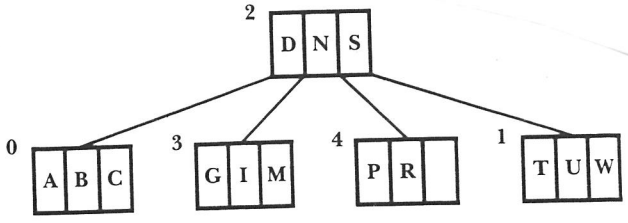


FIGURE 9.21 ▪ B-tree used for the search example.

```
FUNCTION: insert (CURRENT_RRN, KEY PROMO_R_CHILD, PROMO_KEY)

    if CURRENT_RRN = NIL then      /* past bottom of tree */
        PROMO_KEY := KEY
        PROMO_R_CHILD := NIL
        return PROMOTION       /* promote original key and NIL */
    else
        read page at CURRENT_RRN into PAGE
        search for KEY in PAGE.
        let POS := the position where KEY occurs or should occur.

    if KEY found then
        issue error message indicating duplicate key
        return ERROR

    RETURN_VALUE := insert(PAGE.CHILD[POS], KEY, P_B_RRN, P_B_KEY)

    if RETURN_VALUE == NO PROMOTION or ERROR then
        return RETURN_VALUE

    elseif there is space in PAGE for P_B_KEY then
        insert P_B_KEY and P_B_RRN (promoted from below) in PAGE
        return NO PROMOTION
    else
        split(P_B_KEY, P_B_RRN, PAGE,PROMO_KEY, PROMO_R_CHILD, NEWPAGE)
        write PAGE to file at CURRENT_RRN
        write NEWPAGE to file at rrn PROMO_R_CHILD
        return PROMOTION    /* promoting PROMO_KEY and PROMO_R_CHILD */
    endif

end FUNCTION
```

FIGURE 9.24 ▪ Function *insert (CURRENT_RRN, KEY, PROMO_R_CHILD, PROMO_KEY)* inserts a KEY in a B-tree. The insertion attempt starts at the page with relative record number CURRENT_RRN. If this page is not a leaf page, the function calls itself recursively until it finds KEY in a page or reaches a leaf. If it finds KEY, it issues an error message and quits, returning ERROR. If there is space for KEY in PAGE, KEY is inserted. Otherwise, PAGE is split. A split assigns the value of the middle key to PROMO_KEY and the relative record number of the newly created page to PROMO_R_CHILD so that insertion can continue on the recursive ascent back up the tree. If a promotion does occur, *insert()* indicates this by returning PROMOTION. Otherwise, it returns NO PROMOTION.

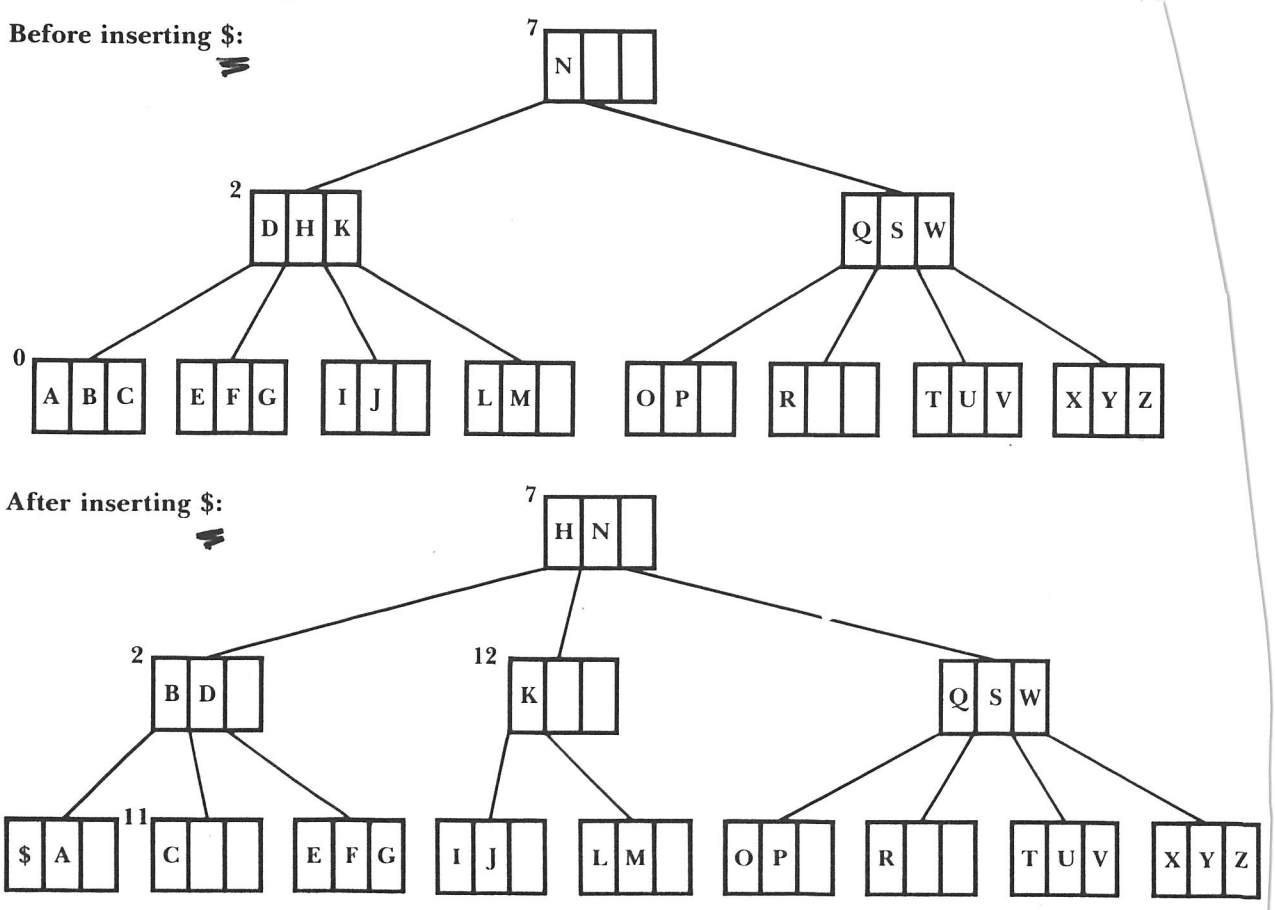


FIGURE 9.22 ▪ The effect of adding $ to the tree constructed in Fig. 9.18.

```
PROCEDURE: split (I_KEY, I_RRN, PAGE, PROMO_KEY, PROMO_R_CHILD, NEWPAGE)

    copy all keys and pointers from PAGE into a working page that
        can hold one extra key and child.

    insert I_KEY and I_RRN into their proper places in the working page.

    allocate and initialize a new page in the B-tree file to hold NEWPAGE.

    set PROMO_KEY to value of middle key, which will be promoted after
        the split.

    set PROMO_R_CHILD to RRN of NEWPAGE.

    copy keys and child pointers preceding PROMO_KEY from the working
        page to PAGE.

    copy keys and child pointers following PROMO_KEY from the working
        page to NEWPAGE.

end PROCEDURE
```

FIGURE 9.25 ▪ *Split (I_KEY, I_RRN, PAGE, PROMO_KEY, PROMO_R_CHILD, NEWPAGE)*, a procedure that inserts I_KEY and I_RRN, causing overflow, creates a new page called NEWPAGE, distributes the keys between the original PAGE and NEWPAGE, and determines which key and RRN to promote. The promoted key and RRN are returned via the arguments PROMO_KEY and PROMO_R_CHILD.

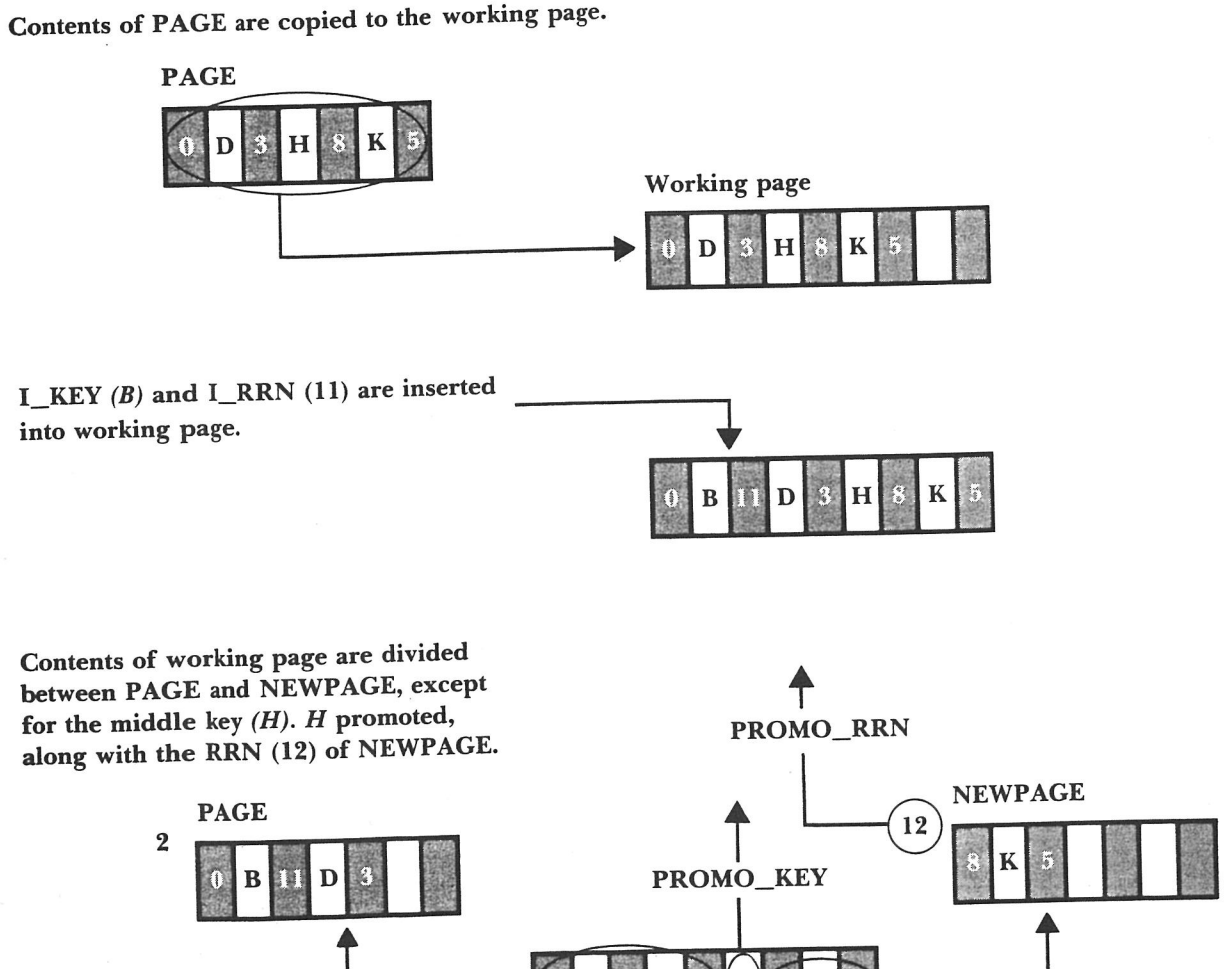
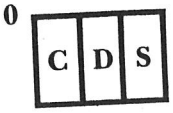


FIGURE 9.26 ▪ The movement of data in *split()*.

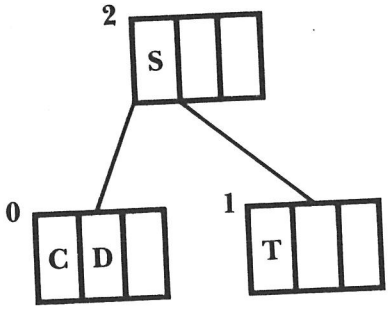
KEY INSERTION SEQUENCE:

C S D T
A M P I
B W N G
U R K E
H O L J
Y Q Z F
X V

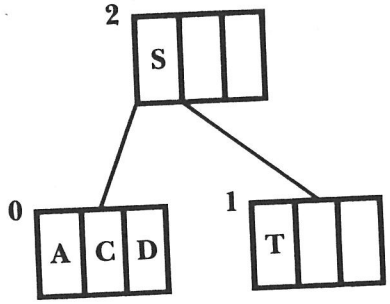
Insertion of *C*, *S*, and *D* into the initial page:



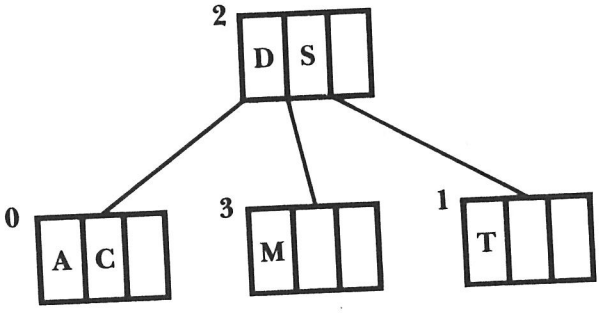
Insertion of *T* forces the split and the promotion of *S*:



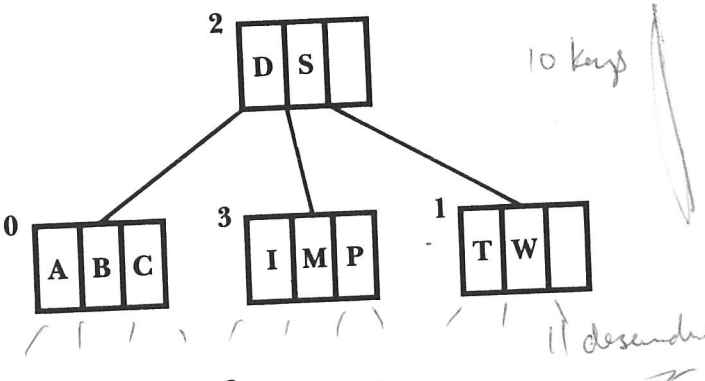
A added without incident:



Insertion of *M* forces another split and the promotion of *D*:



P, *I*, *B*, and *W* inserted into existing pages:



Insertion of *N* causes another split, followed by the promotion of *N*. *G*, *U*, and *R* are added to existing pages:

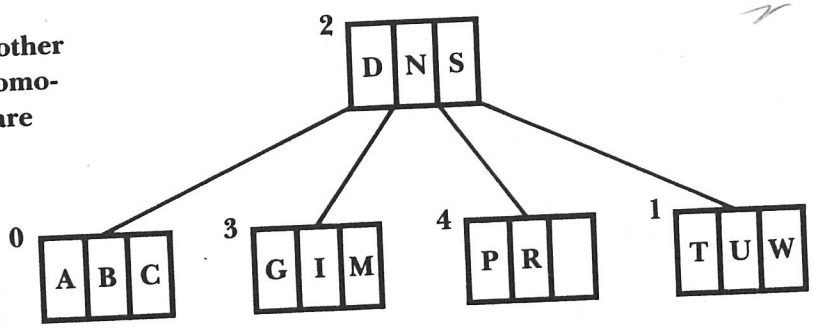
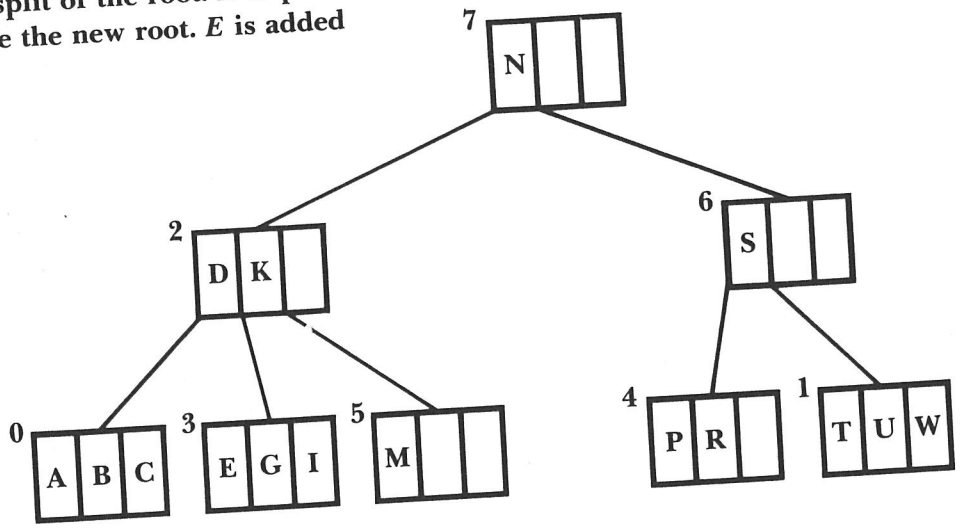
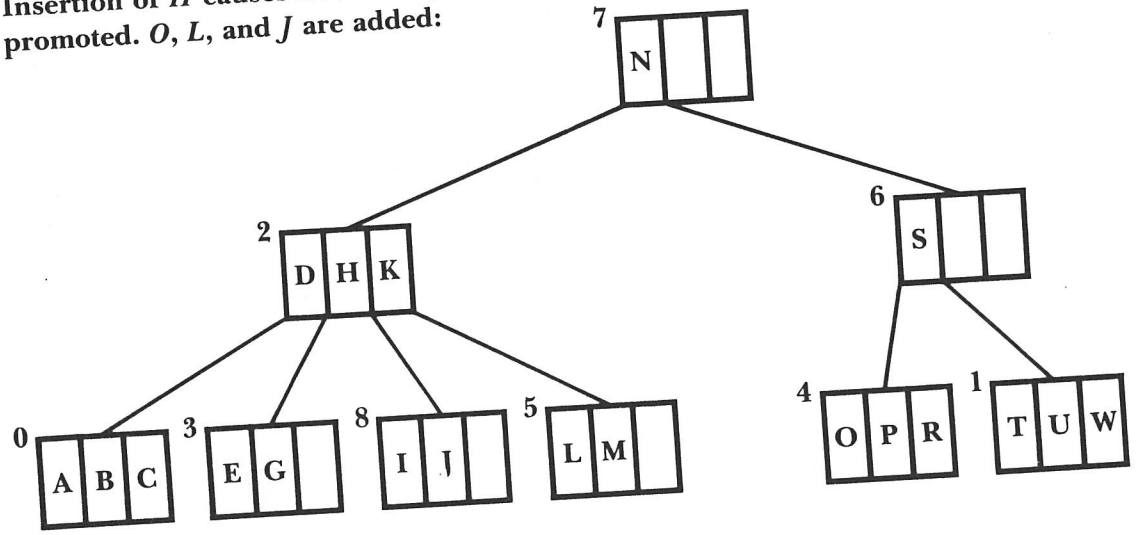


FIGURE 9.17 ▪ Growth of a B-tree, Part I. The tree grows to a point at which splitting of the root is imminent.

Insertion of *K* causes a split at leaf level, followed by the promotion of *K*. This causes a split of the root. *N* is promoted to become the new root. *E* is added to a leaf:



Insertion of *H* causes a leaf to split. *H* is promoted. *O*, *L*, and *J* are added:



Insertion of *Y* and *Q* force two more leaf splits and promotions. Remaining letters are added:

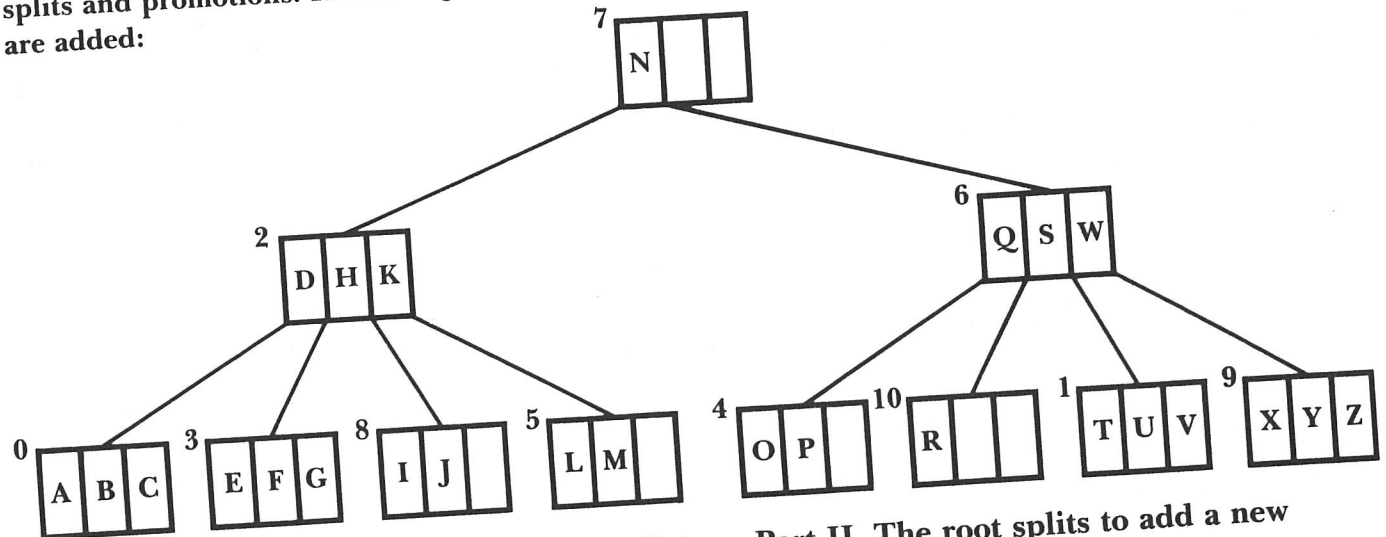
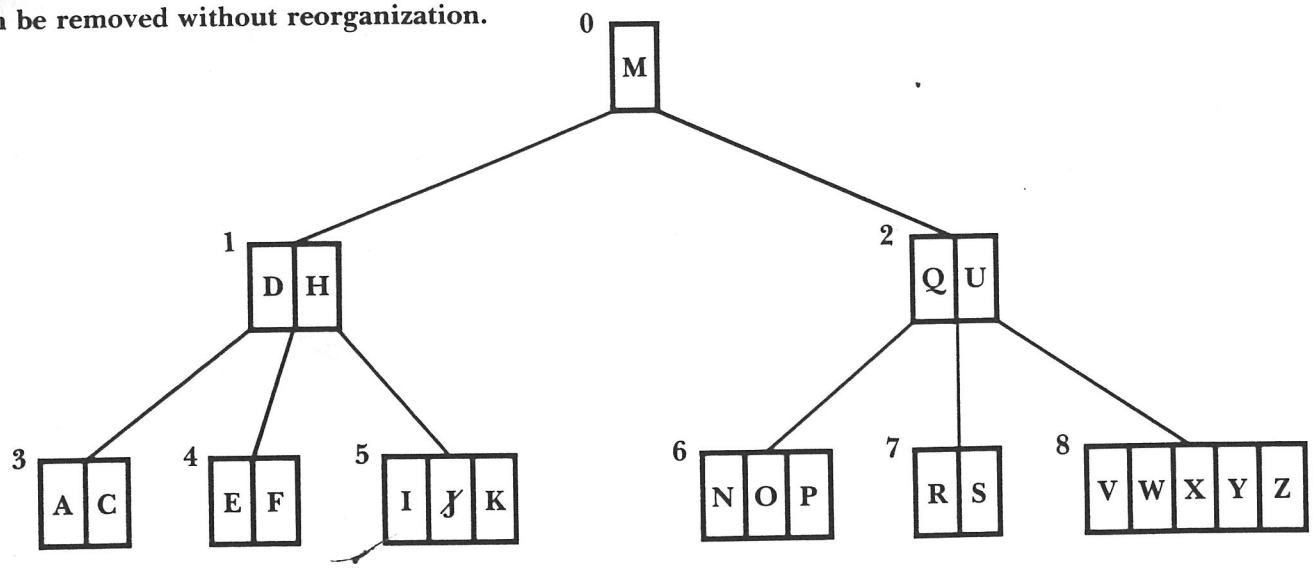
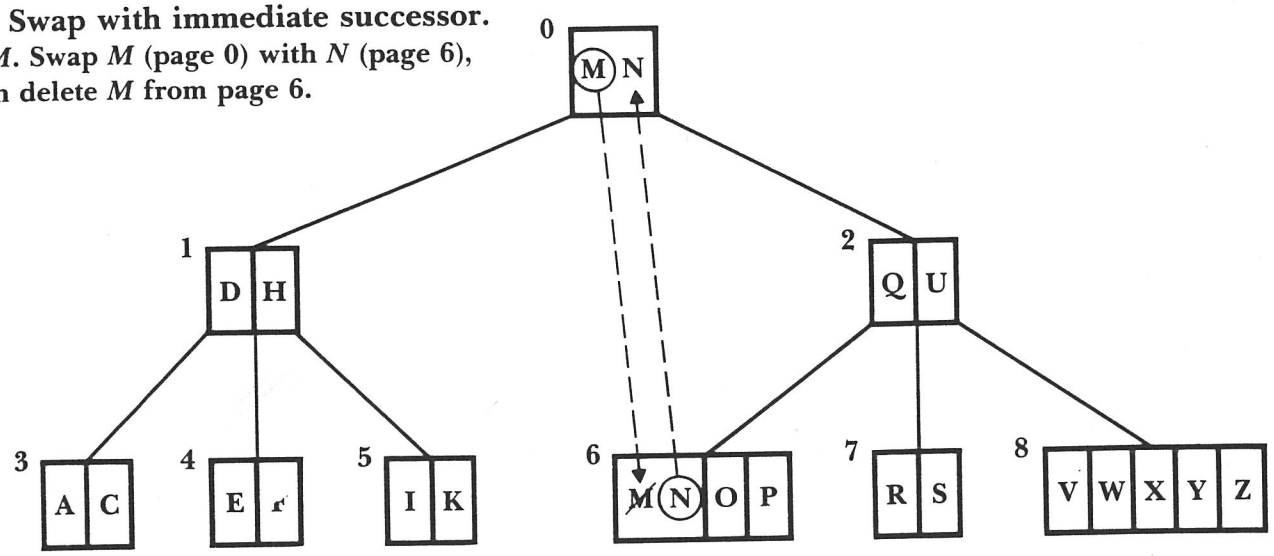


FIGURE 9.18 ▪ Growth of a B-tree, Part II. The root splits to add a new level; remaining keys are inserted.

Case 1: No action.
Delete *J* from page 5. Since page 5 has more than the minimum number of keys, *J* can be removed without reorganization.



Case 2: Swap with immediate successor.
Delete *M*. Swap *M* (page 0) with *N* (page 6), and then delete *M* from page 6.



Case 3: Redistribution.
Delete *R*. Underflow occurs. Redistribute keys among pages 2, 7, and 8 to restore balance between leaves.

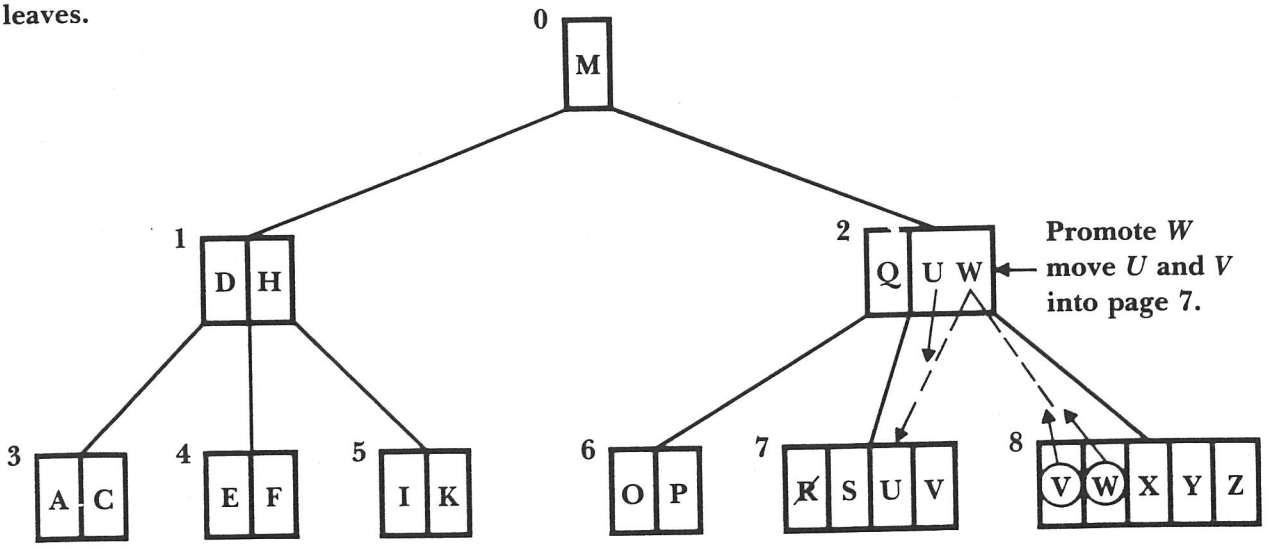
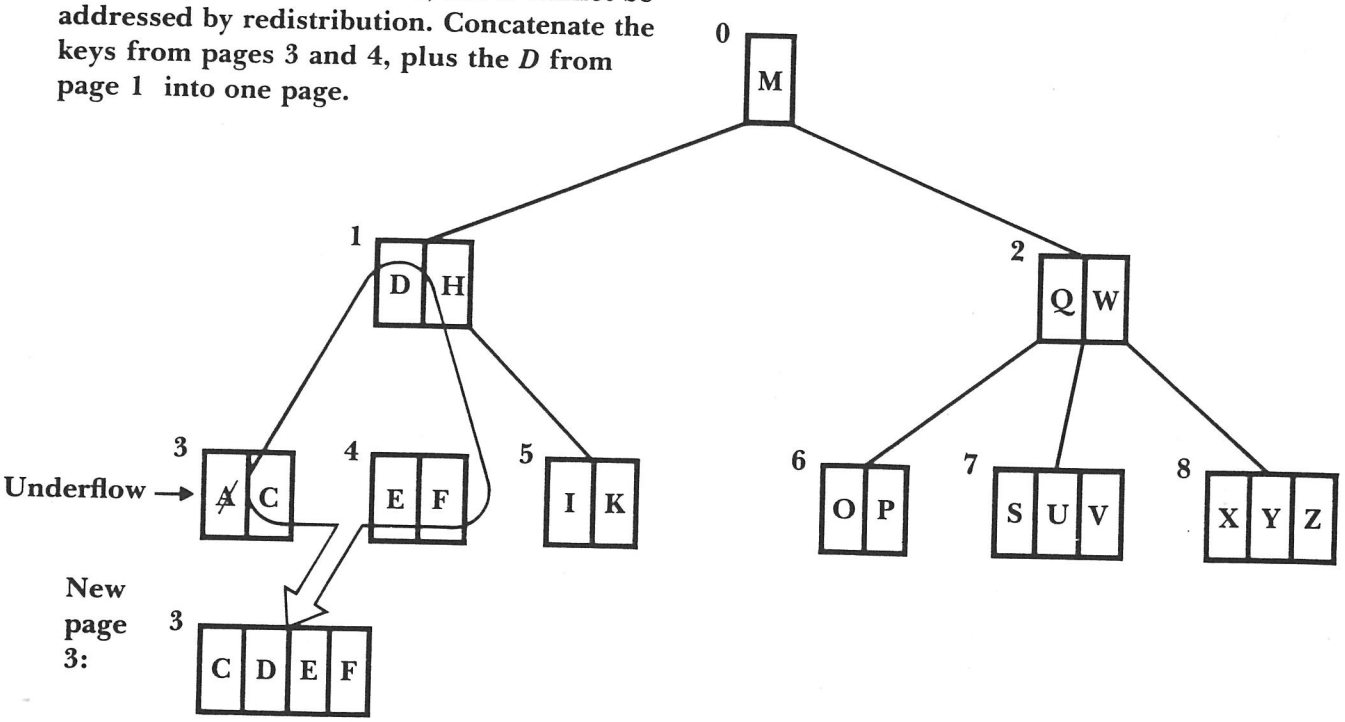
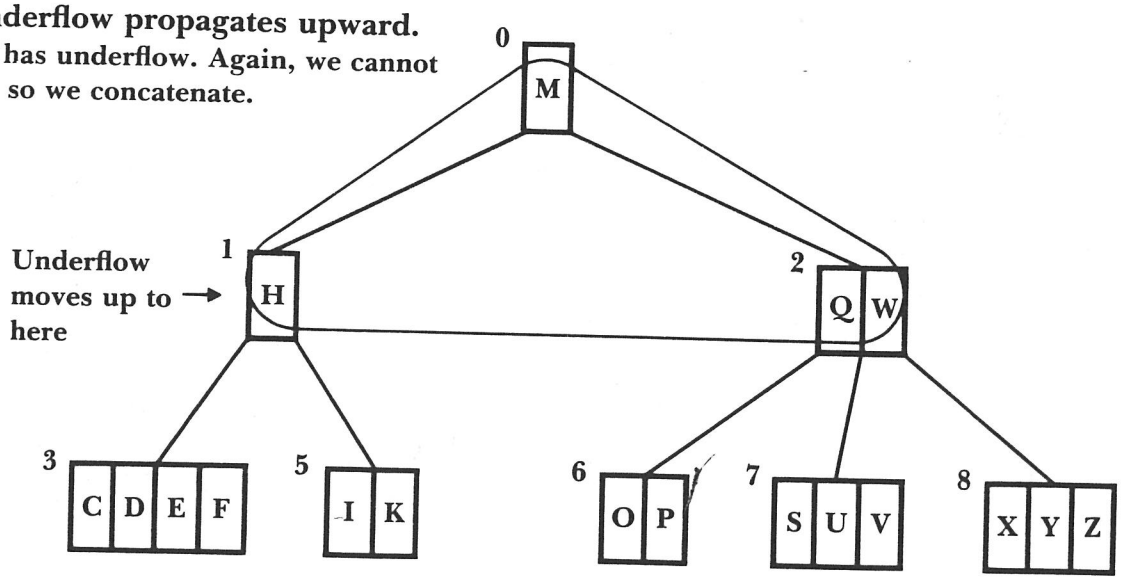


FIGURE 9.29 ▪ Six situations that can occur during deletions.

Case 4: Concatenation.
Delete *A*. Underflow occurs, but it cannot be addressed by redistribution. Concatenate the keys from pages 3 and 4, plus the *D* from page 1 into one page.



Case 5: Underflow propagates upward.
Now page 1 has underflow. Again, we cannot redistribute, so we concatenate.



Case 6: Height of tree decreased.
Since the root contains only one key, it is absorbed into the new root.