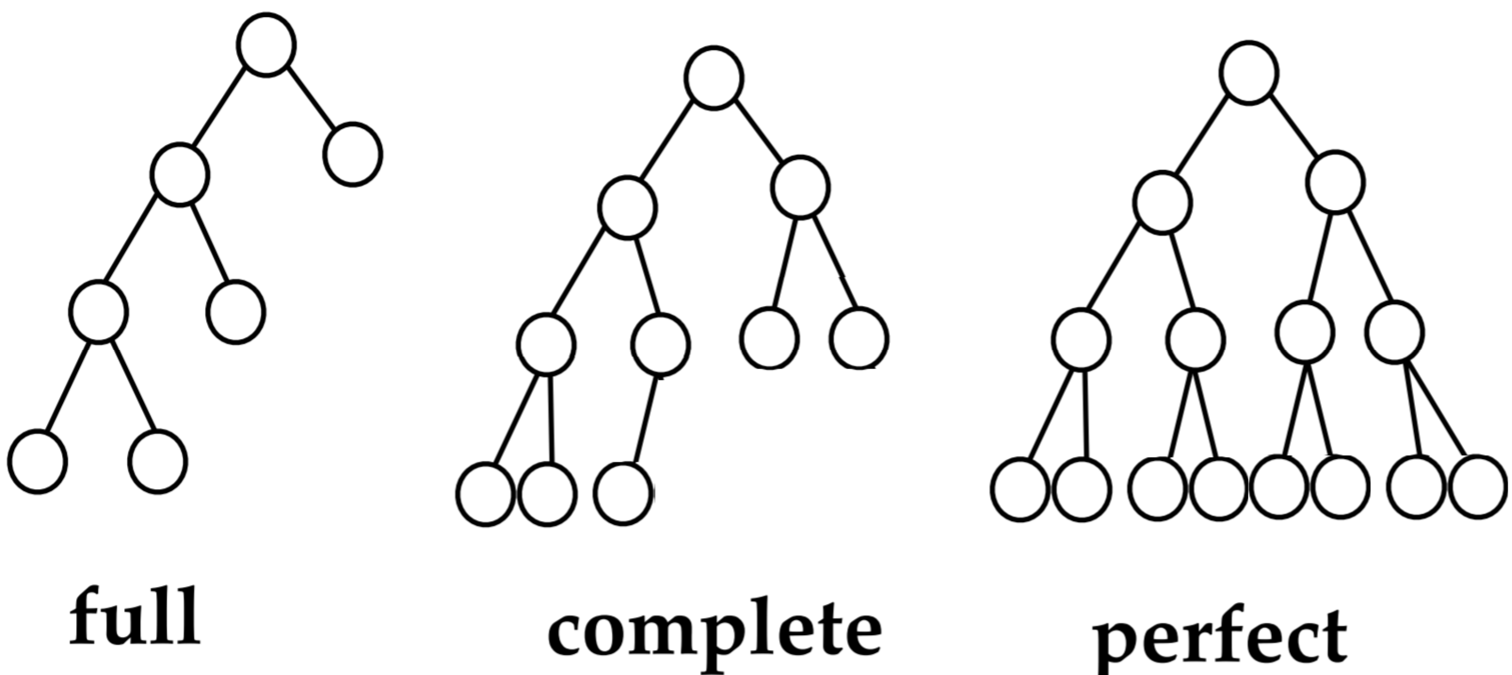
EEECE7205: Fundamentals of Computer Engineering

Binary and Balanced Search Trees

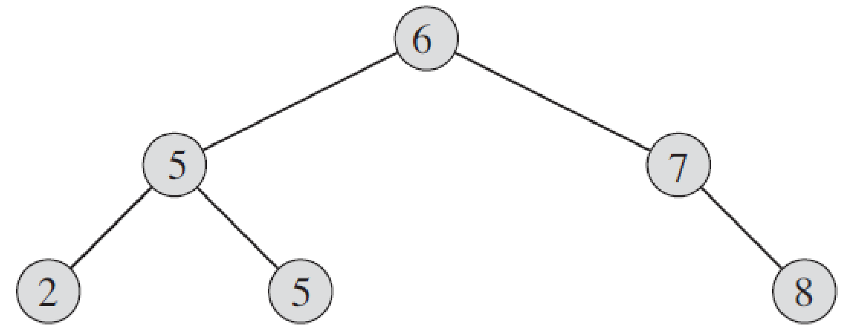
Binary Tree Types

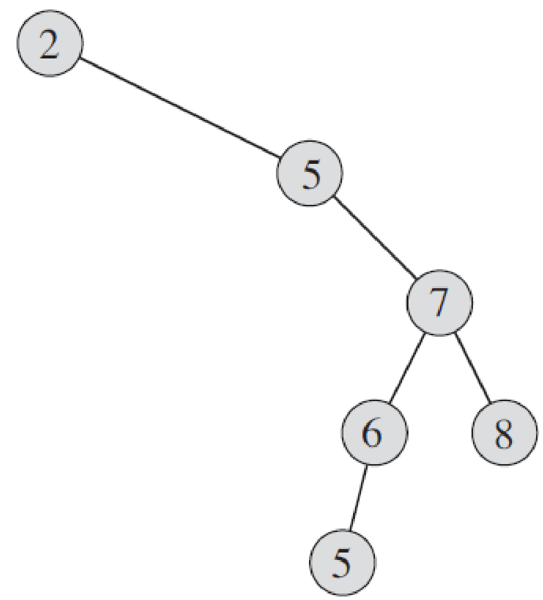
- In a **binary** tree, every node has 0, 1, or 2 children.
  - Nodes with 0 children all called **leaves**.
- In a **full** binary tree, every node has either 0 or 2 children.
- A **complete** binary tree is completely filled on all levels except the lowest level, which is filled from the left to right.
- A **perfect** binary tree is a full binary tree with all **leaves** at the same depth.



Binary Search Trees (BST)

- A binary search tree is organized, as the name suggests, in a binary tree.
- For any node *x*, the keys in the left subtree of *x* are at most *x*.*key*, and the keys in the right subtree of *x* are at least *x*.*key*.
- Different binary search trees can represent the same set of values (as shown where the top representation is more efficient than the bottom one)



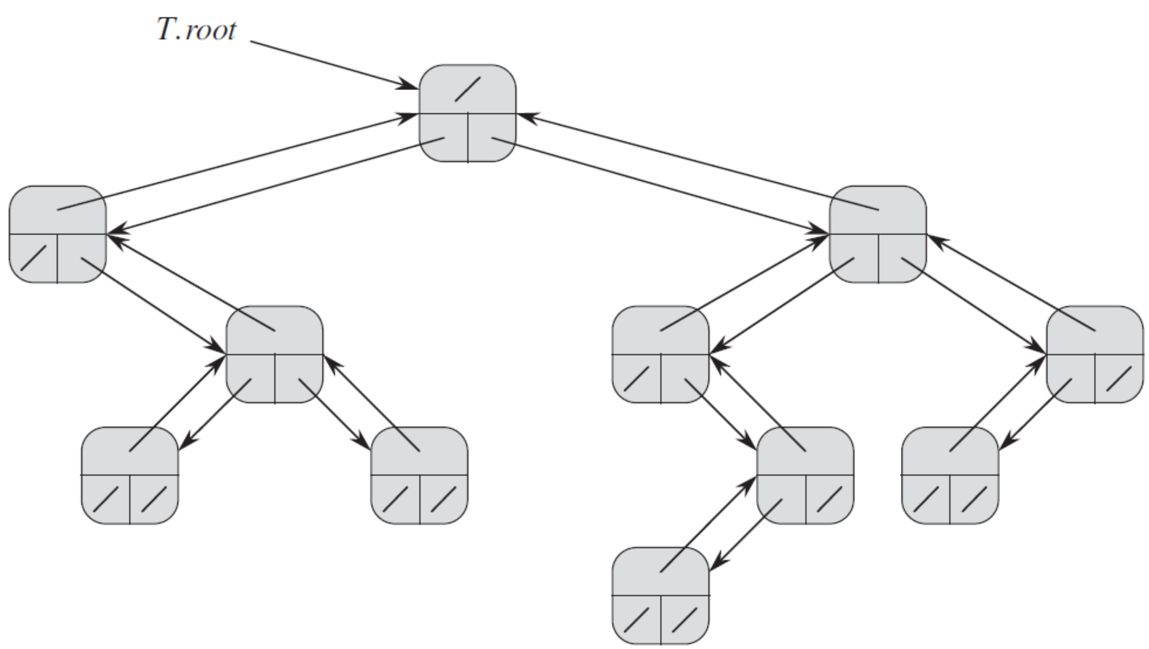


BST Property

Let x be a node in a binary search tree.
If y is a node in the left subtree of x,
then $y.key \leq x.key$.
If y is a node in the right subtree of x,
then $y.key \geq x.key$

BST Implementation

- We can represent such a tree by a **linked data structure** in which each node is an object.
- In addition to a *key* and satellite data, each node contains attributes *left*, *right*, and *p* that point to the nodes corresponding to its left child, its right child, and its parent, respectively.
- If a child or the parent is missing, the appropriate attribute contains the value NIL. The root node is the only node in the tree whose parent is NIL.



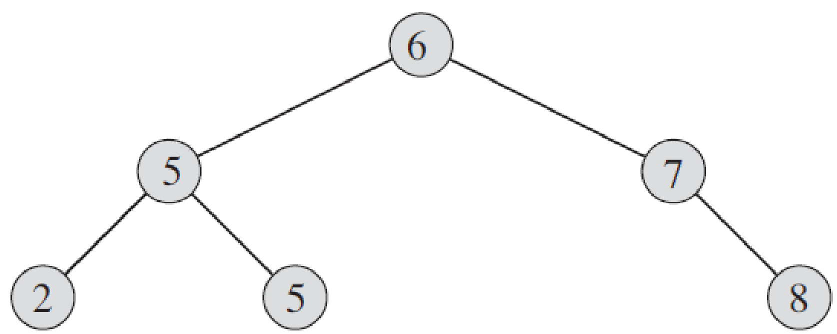
BST Operations

- The search tree data structure supports many dynamic-set operations, including SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE.
- Thus, we can use a search tree both as a **dictionary** and as a **priority queue**.
- Basic operations on a binary search tree take time proportional to the height of the tree.
- For a complete binary tree with *n* nodes, such operations run in $\Theta(\log n)$ worst-case time.

In-order BST Walk

- The binary-search-tree property allows us to print out all the keys in a binary search tree in **sorted order** by a simple recursive algorithm, called an ***inorder tree walk***.

```
INORDER-TREE-WALK(x)
if x ≠ NIL
    INORDER-TREE-WALK(x.left)
    print x.key
    INORDER-TREE-WALK(x.right)
```



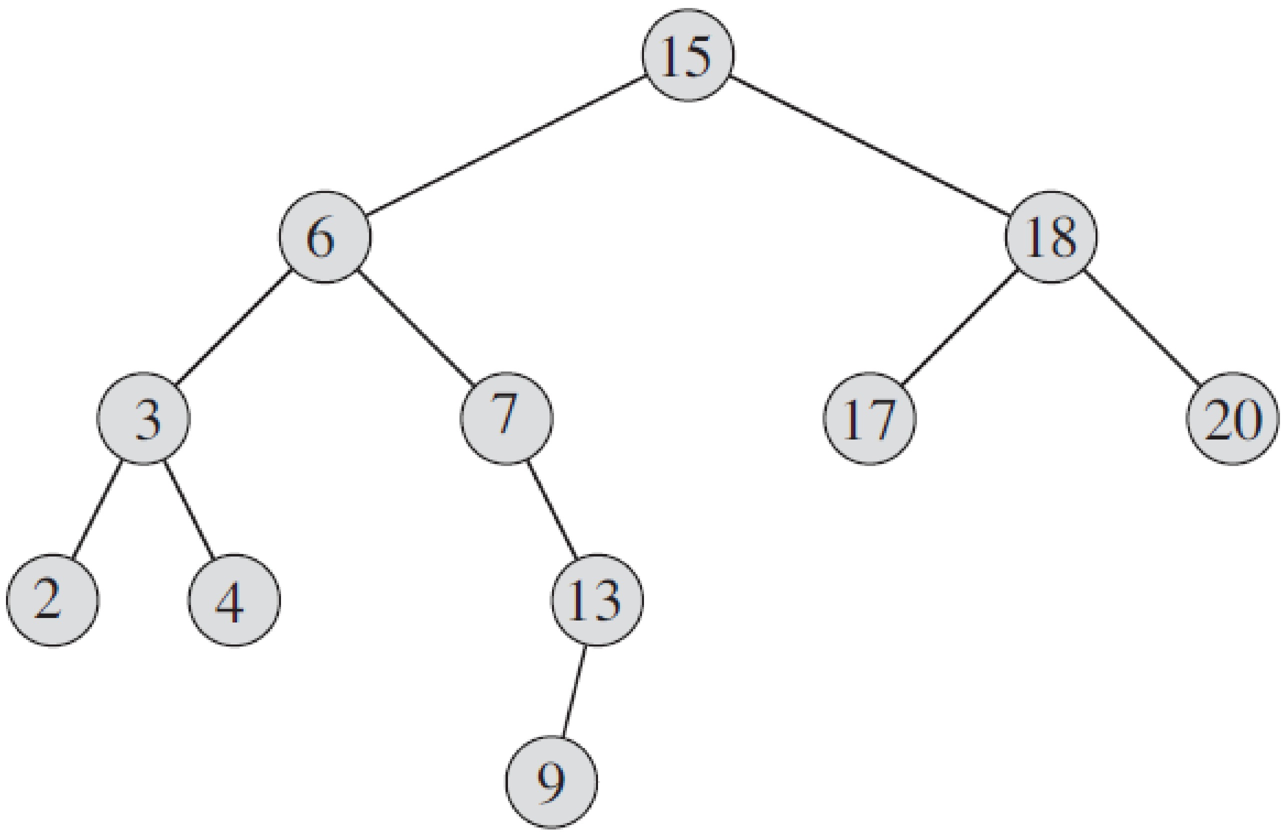
BST Search (Recursive Version)

- The following procedure searches for a node with a given key in a binary search tree.
- Given a pointer to the root of the tree and a key *k*, TREE-SEARCH returns a pointer to a node with key *k* if one exists; otherwise, it returns NIL.
- The running time of TREE-SEARCH is O(*h*), where *h* is the height of the tree.

```
TREE-SEARCH(x, k)
if x == NIL or k == x.key
    return x
if k < x.key
    return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)
```

BST Search Example

- To search for the key 13 in the shown tree, we follow the path 15 → 6 → 7 → 13 from the root.



BST Search (Iterative Version)

- We can rewrite this procedure in an iterative fashion by “unrolling” the recursion into a **while** loop. On most computers, the iterative version is more efficient

```
ITERATIVE-TREE-SEARCH(x, k)
while x ≠ NIL and k ≠ x.key
    if k < x.key
        x = x.left
    else x = x.right
return x
```

BST Minimum and Maximum

- The following procedures returns a pointer to the minimum and maximum elements in the subtree rooted at a given node *x*, which we assume to be non-NIL.
- Both procedures run in O(*h*) time on a tree of height *h* since, as in TREE-SEARCH,

```
TREE-MINIMUM(x)
1  while x.left ≠ NIL
2      x = x.left
3  return x
```

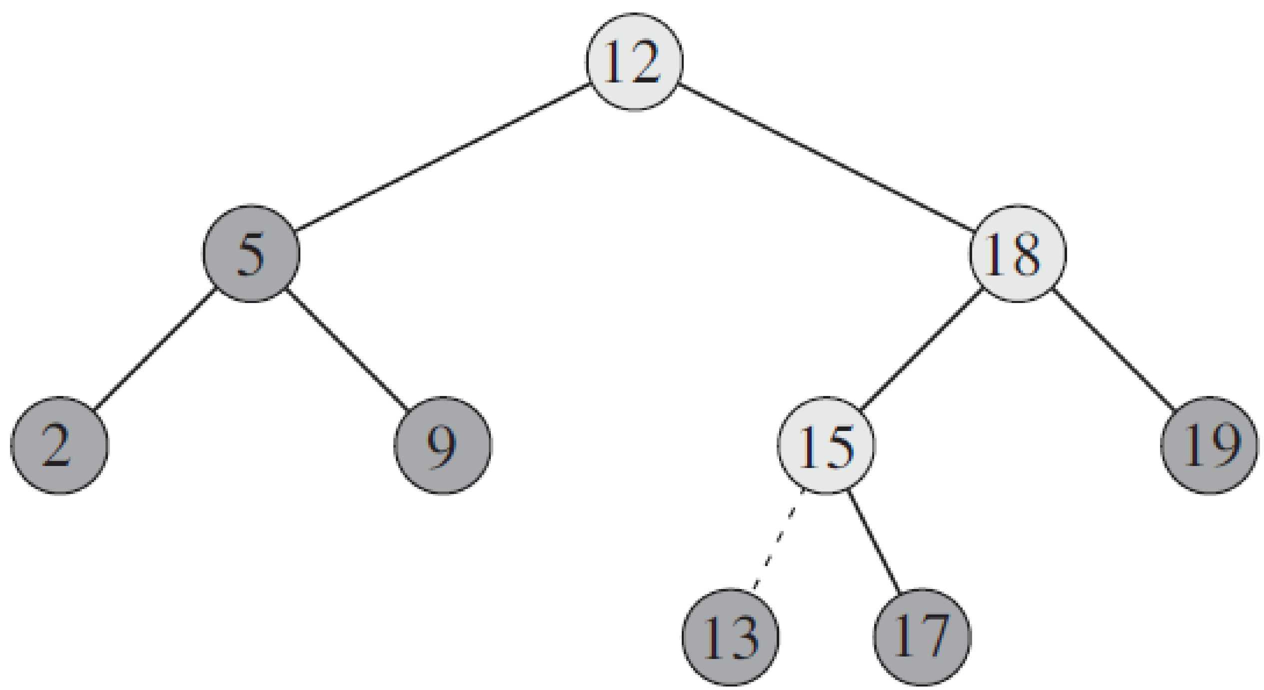
```
TREE-MAXIMUM(x)
1  while x.right ≠ NIL
2      x = x.right
3  return x
```

BST Insertion and Deletion

- The operations of insertion and deletion cause the dynamic set represented by a binary search tree to change.
- The data structure must be modified to reflect this change, but in such a way that the binary-search-tree property continues to hold.

TREE-INSERT Example

- TREE-INSERT begins at the root of the tree and the pointer *x* traces a simple path downward looking for a NIL to replace with the input item *z*.
- The procedure maintains the ***trailing pointer*** *y* as the parent of *x*.
- *Example:* Inserting an item with key 13 into a binary search tree.



BST Insertion Procedure

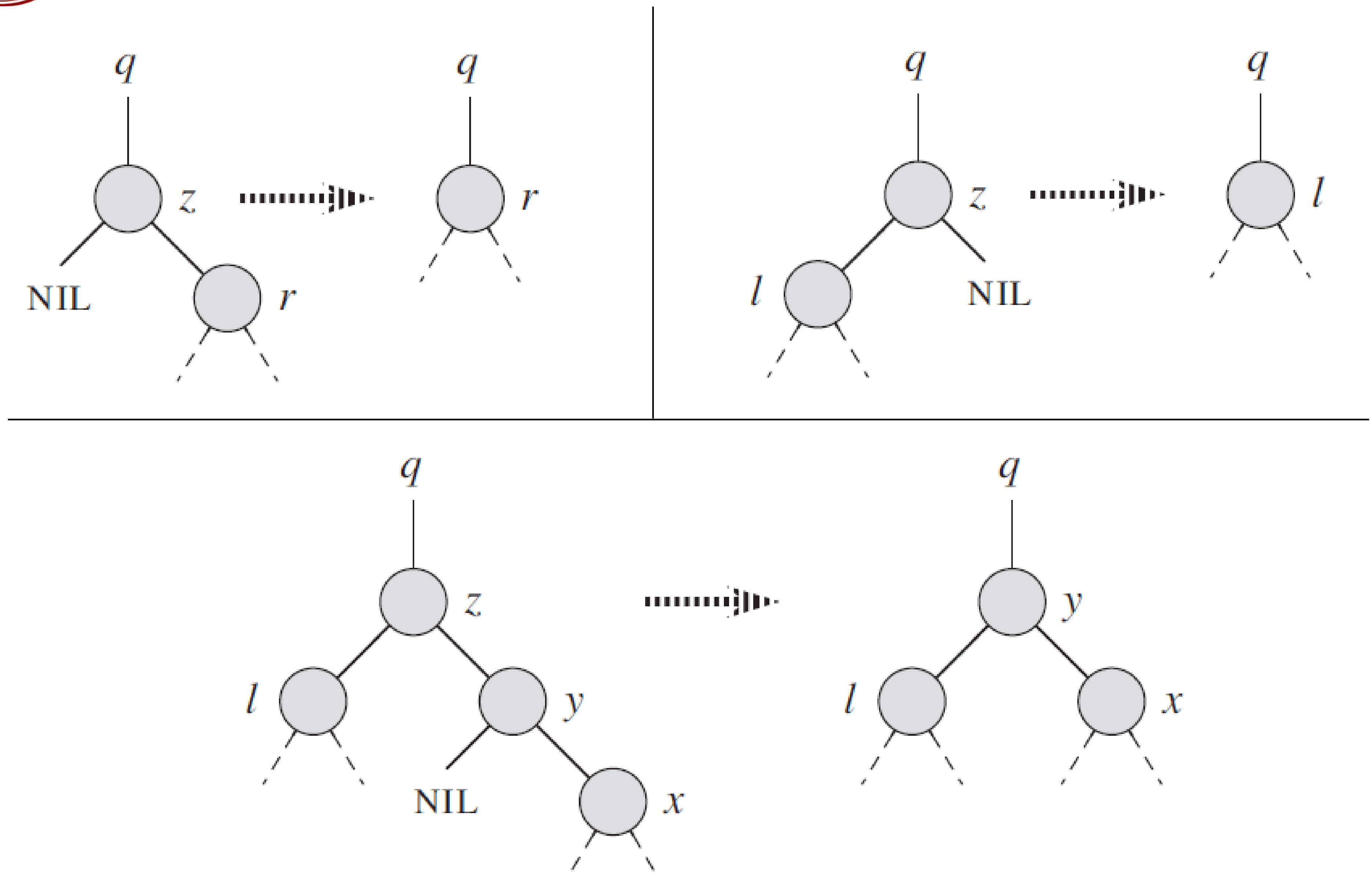
- The procedure inserts in binary search tree *T* a node *z* for which *z.key* = *v* , *z.left* = NIL, and *z.right* = NIL.
- It modifies *T* and some of the attributes of *z* in such a way that it inserts *z* into an appropriate position in the tree.
- The procedure TREE-INSERT runs in O(*h*) time on a tree of height *h*.

```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z  // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```

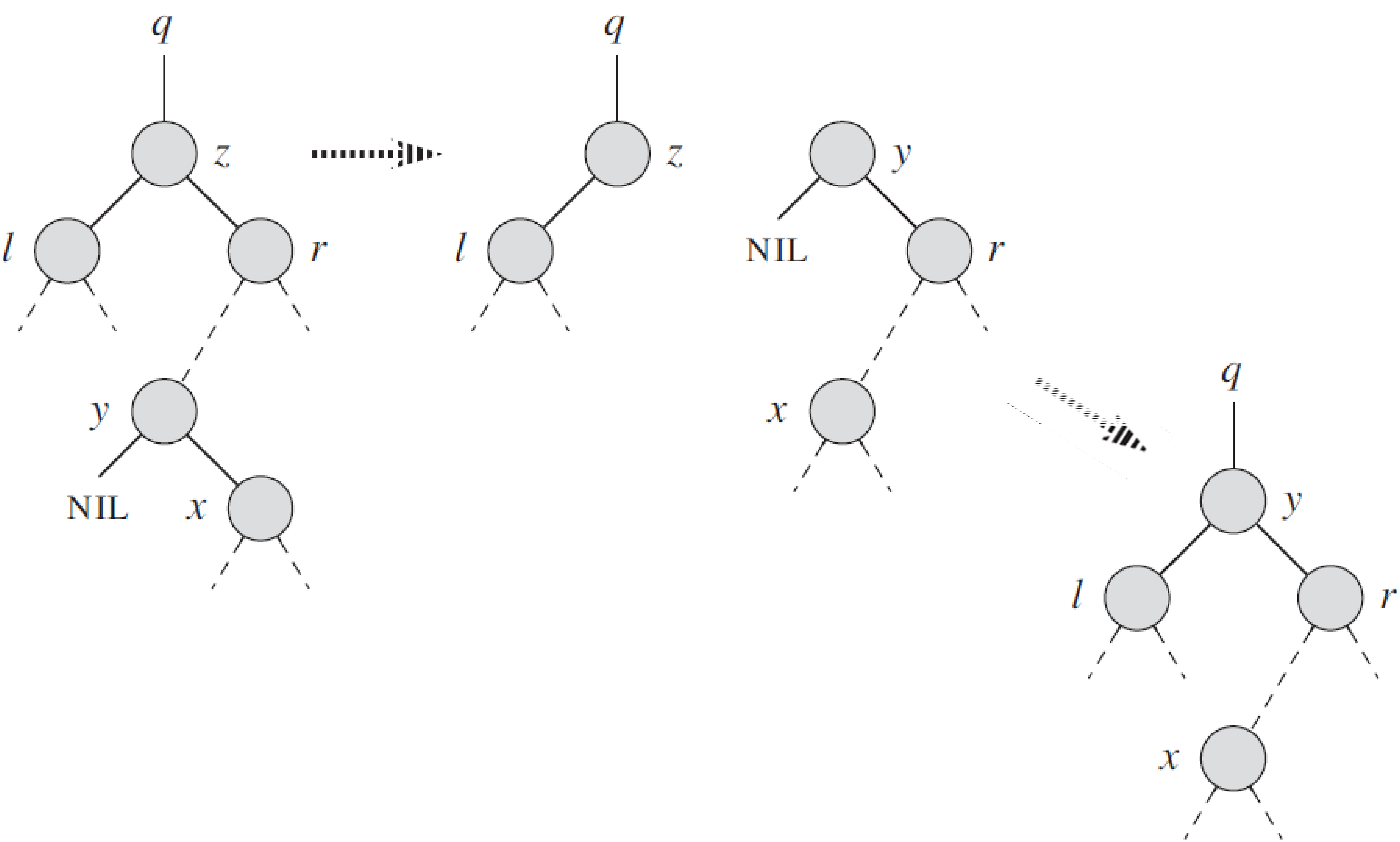
BST Deletion Procedure

- The overall strategy for deleting a node *z* from a binary search tree *T* has three cases:
  1. If *z* has no children, then we simply remove it by modifying its parent to replace *z* with NIL as its child.
  2. If *z* has just one child, then we elevate that child to take *z*'s position in the tree by modifying *z*'s parent to replace *z* by *z*'s child.
  3. If *z* has two children, then we find *z*'s successor *y*, which is the minimum element in *z*'s right subtree, and have *y* take *z*'s position in the tree. The rest of *z*'s original right subtree becomes *y*'s new right subtree, and *z*'s left subtree becomes *y*'s new left subtree.

BST Deletion Examples

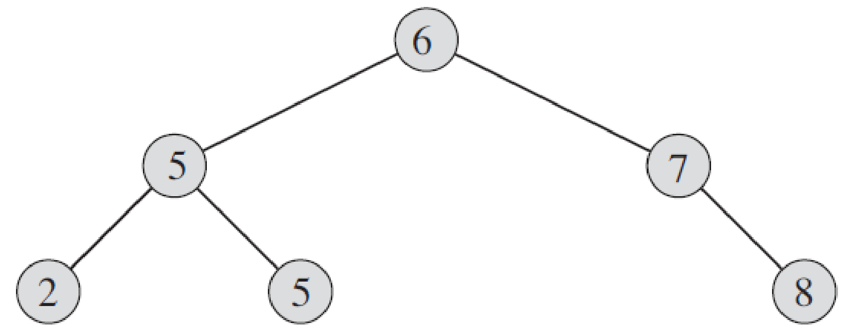


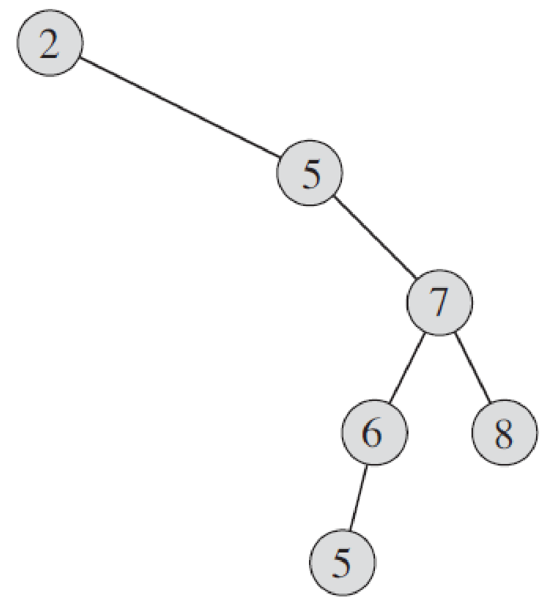
BST Deletion Examples (Cont'd)



Balanced Search Trees (1 of 2)

- A binary search tree (BST) of height *h* can support any of the basic dynamic-set operations -such as SEARCH, PREDECESSOR, SUCCESSOR, MINIMUM, MAXIMUM, INSERT, and DELETE - in *O*(*h*) time.
  - The set operations are fast if the height of the search tree is small, which is the case if the tree is **balanced**.
  - If its height is large, however, the set operations may run no faster than with a linked list.
- To balance a BST, you need to **rebuild** it by using the node with the **median** value as its root and recursively do that with its subtrees.





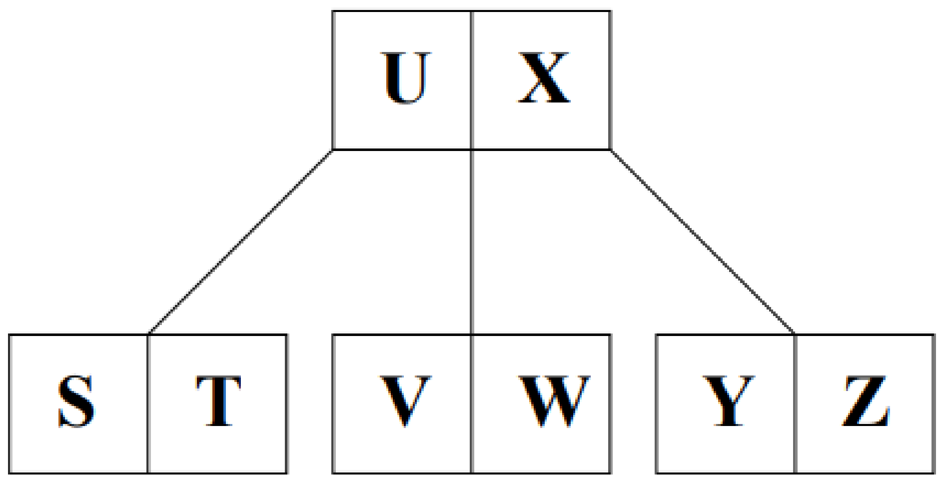
Balanced Search Trees (2 of 2)

- ***Balanced search tree*** is a search-tree data structure for which a height of $O(\log n)$ is guaranteed.

- *Examples*: B-trees, Red-black trees, AVL trees, 2-3 trees, 2-3-4 trees.

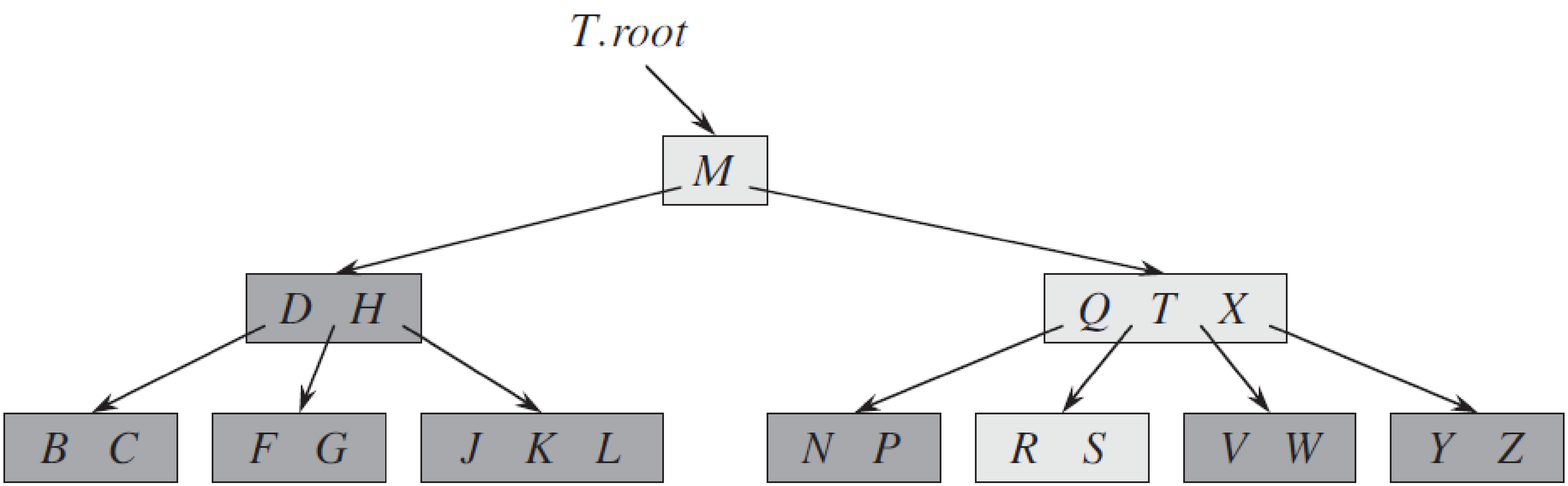
B-Trees

- B-trees are balanced search trees designed to work well on disks or other direct access secondary storage devices.
- B-trees nodes may have **many children**, from a few to thousands.
- Every n-node B-tree has height $\boldsymbol{O(\log n)}$.
- If an internal B-tree node x contains $\boldsymbol{x.n}$ keys, then x has $\boldsymbol{x.n + 1}$ children.



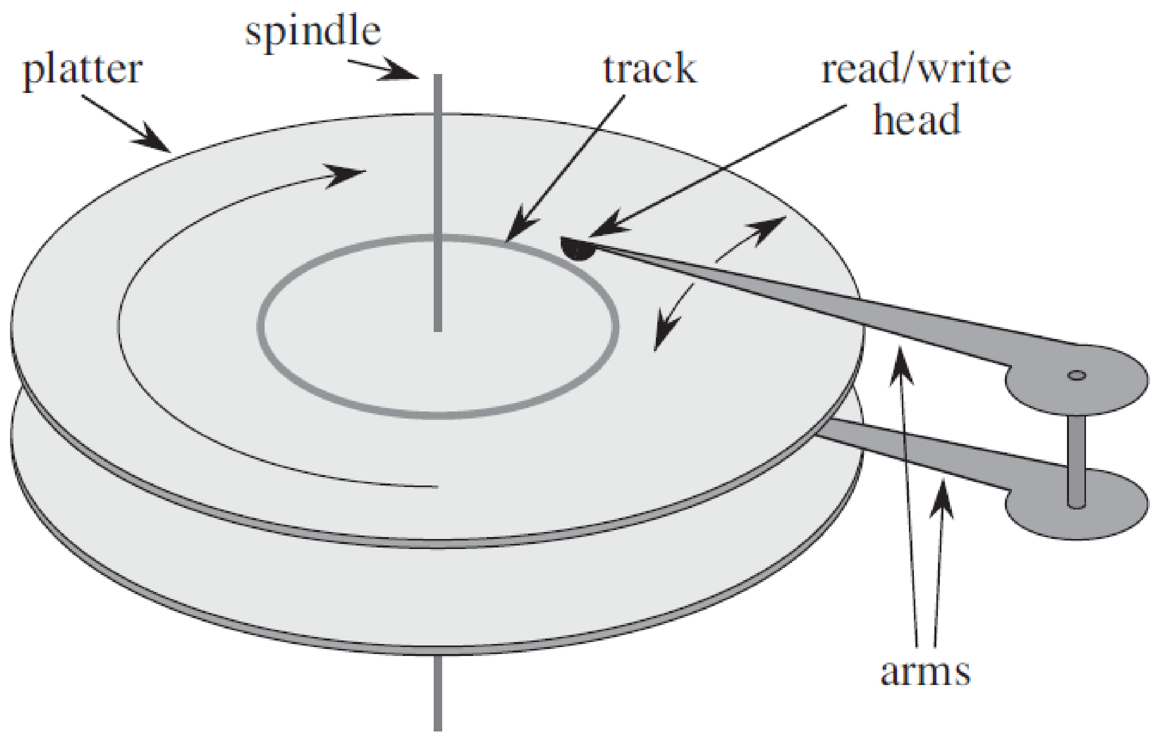
B-Tree Example

- The shown B-tree has keys representing the consonants of English.
- The lightly shaded nodes are examined in a search for the letter *R*.



Secondary Storage (1 of 2)

- Most computer systems also have ***secondary storage*** based on magnetic disks.
- The figure shows a typical disk drive that consists of one or more ***platters***, which rotate at a constant speed around a common ***spindle***.
- The drive reads and writes each platter by a ***head*** at the end of an ***arm***.
- When a given head is stationary, the surface that passes underneath it is called a ***track***.



Secondary Storage (2 of 2)

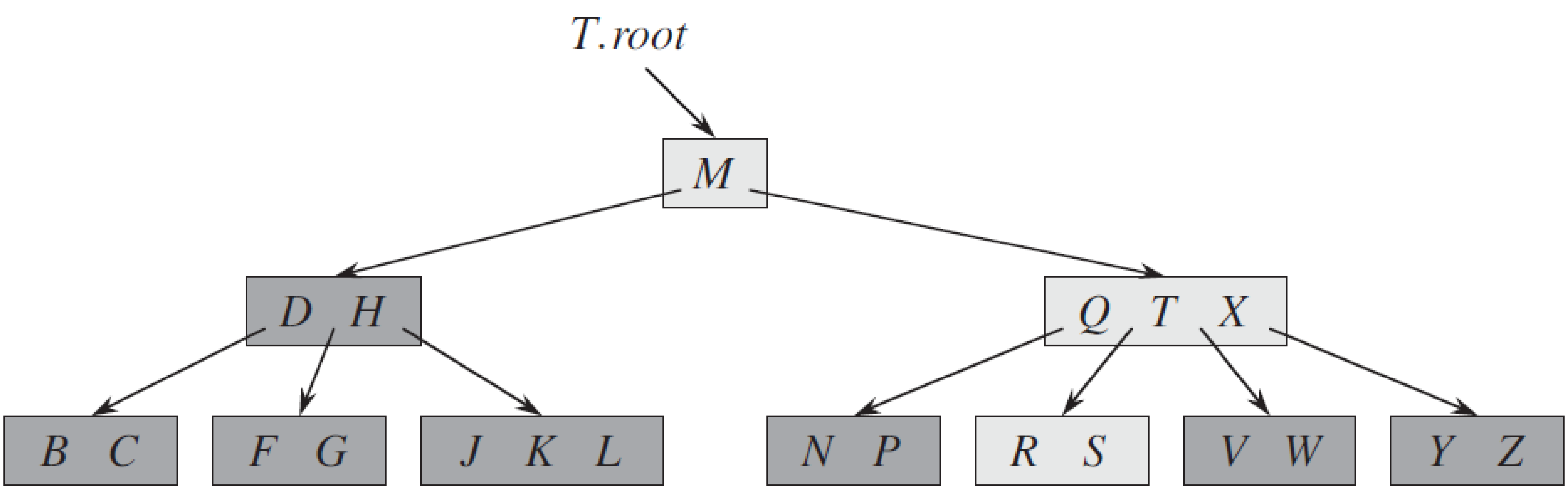
- Although disks are cheaper and have higher capacity than main memory (RAM), they are much, much slower because they have moving mechanical parts (platter rotation and arm movement).
  - A typical RAM is **100,000 times faster** than a typical disk.
- In order to amortize the time spent waiting for mechanical movements, disks access not just one byte but several at a time.
- Information is divided into a number of equal-sized ***pages*** of bytes that appear consecutively within tracks, and each disk read or write is of one or more entire pages.
  - For a typical disk, a page might be $\mathbf{2^{11}}$ to $\mathbf{2^{14}}$ bytes in length.

B-Tree and Secondary Storage

- A **B-tree node** is usually as large as **a whole page** in the disk, and this size limits the number of children a B-tree node can have.
- In a typical B-tree application, the amount of data handled is so large that all the data do not fit into main memory at once.
- The B-tree algorithms copy selected pages from disk into main memory as needed and write back onto disk the pages that have changed.
- The operation **DISK-READ(*x*)** is used to read **node *x*** from the disk into main memory before we can refer to its attributes (e.g., $x.key_i$).
- The operation **DISK-WRITE(*x*)** is used to save any changes that have been made to the attributes of node *x*.

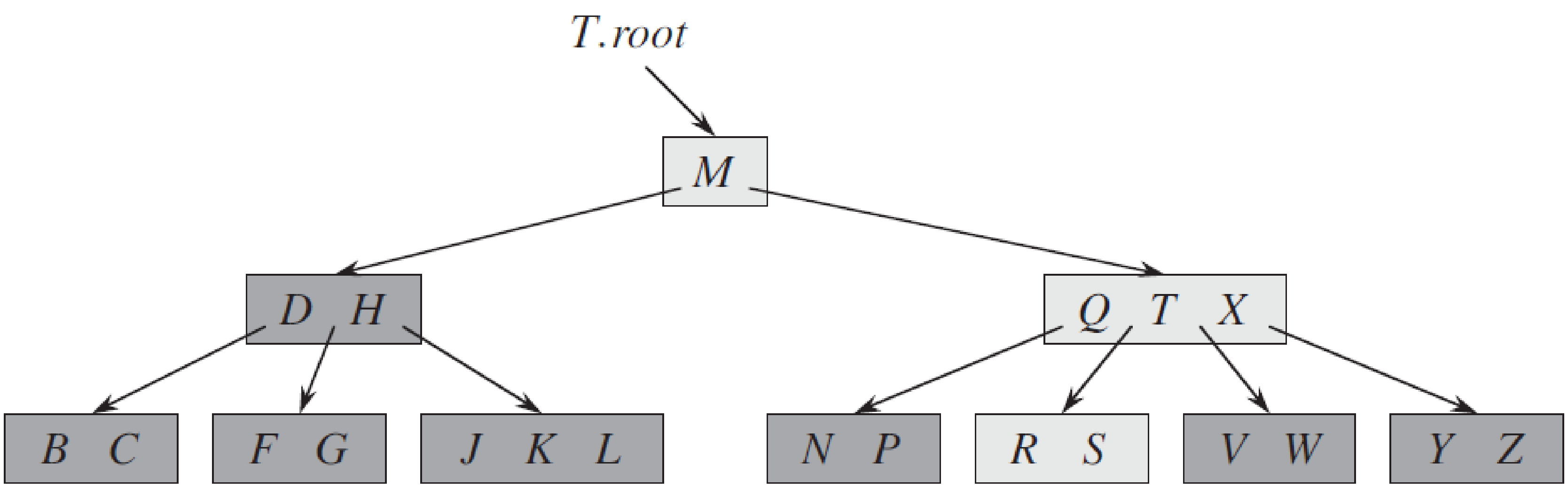
B-tree Properties (1 of 2)

- A ***B-tree*** *T* is a rooted tree (whose root is *T.root*)
- Every node *x* has the following attributes:
  a. ***x.n***, the number of keys currently stored in node *x*,
  b. the *x.n* keys themselves are stored in increasing order, so that $x.key_1 \le x.key_2 \le \dots \le x.key_{x.n}$
  c. ***x.leaf*** , a Boolean value that is TRUE if *x* is a leaf and FALSE if *x* is an internal node.



B-tree Properties (2 of 2)

- Each internal (not leaf) node x also contains $x.n + 1$ pointers $\boldsymbol{x.c_1, x.c_2, \ldots, x.c_{x.n+1}}$ to its children.
  - Leaf nodes have no children, and so their c_i attributes are undefined.
- All leaves have the same depth, which is the tree's height h.
- The keys $x.key_i$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $x.c_i$, then $k_1 \le x.key_1 \le k_2 \le x.key_2 \le \ldots \le x.key_{x.n} \le k_{x.n+1}$



The Minimum Degree of a B-tree

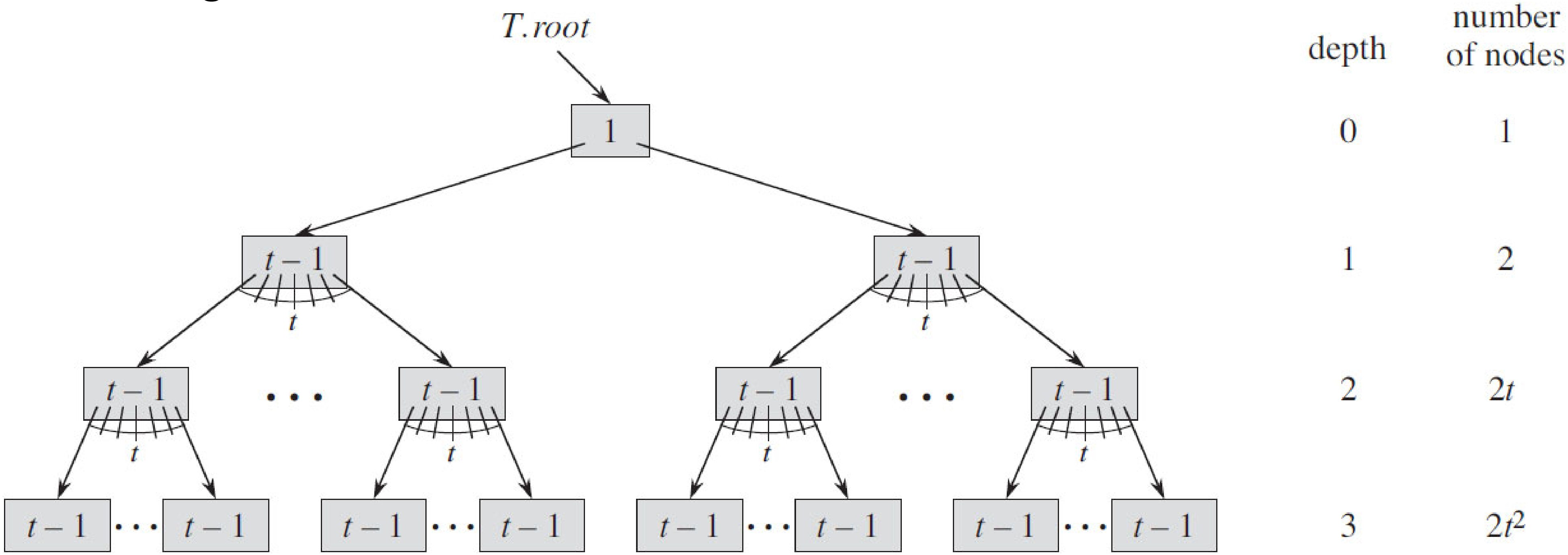
- Nodes have lower and upper bounds on the number of keys they can contain. We express these bounds in terms of a fixed integer $t \geq 2$ called the ***minimum degree*** of the B-tree:
  a. Every node other than the root must have at least ***t*** **- 1 keys**.
  b. An internal node (i.e., neither a root nor a leaf) with the least number of keys thus has ***t*** **children**.
  c. If the tree is nonempty, the root must have at least **one key**.
  d. Every node may contain at most **2*t* - 1 keys**.
  e. A non-leaf node with the most number of keys thus has **2*t* children**.
  f. We say that a node is ***full*** if it contains exactly **2*t* - 1** keys.

The Simplest B-tree

- The simplest B-tree occurs when $t = 2$, and internal nodes have from t-1 to 2t-1 keys.
- Every internal node then has either 2, 3, or 4 children, and we call this tree a ***2-3-4 tree***.
- In practice, however, much larger values of t yield B-trees with smaller height.

The Height of a B-tree

- The number of disk accesses for most operations on a B-tree is proportional to the height of the B-tree. What is the **worst-case height** of a B-tree? Worst case happens if every node has the minimum allowed number of keys.
- ***Theorem***: If $n \geq 1$, then for any n-node B-tree T of height h and minimum degree $t \geq 2$ → $h \leq \log_t \frac{n+1}{2}$
- The figure illustrates such a tree for $h = 3$.



$n-1 = 2+2t+2t^2+\ldots.. +2t^{h-1} = 2\,(t^h-1)/(t-1)$ (for max h, t=2 → $(n+1)/2 = t^h$)

Basic Operations on B-trees

- Basic operations on B-trees include:
  - B-TREE-CREATE
  - B-TREE-SEARCH
  - B-TREE-INSERT
  - B-TREE-DELETE

Creating an Empty B-Tree

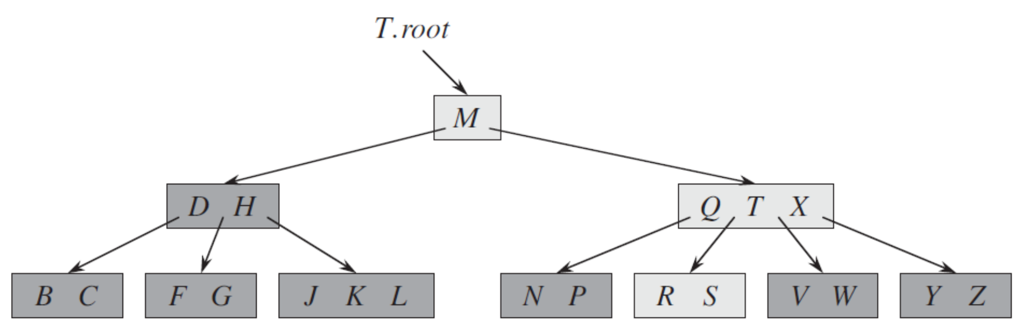
- To build a B-tree *T*, we first use B-TREE-CREATE to create an empty root node and then call B-TREE-INSERT to add new keys.
- Both of these procedures use an auxiliary procedure ALLOCATE-NODE, which allocates one disk page to be used as a new node in *O*(1) time.
  - Note here the address of a node is not a pointer to a main memory location, rather an address in the disk.

```
B-TREE-CREATE(T)
    x = ALLOCATE-NODE()
    x.leaf = TRUE
    x.n = 0
    DISK-WRITE(x)
    T.root = x
```

Searching a B-tree

- To search a B-tree instead of making a two-way branching decision at each node (as in binary search**)**, we make at each internal node *x*, an *x*.*n*+1 way **branching** decision.
- Lines 1–3 find the smallest index *i* such that $\boldsymbol{k \le x.key_i}$, or else they set *i* to *x*.*n* + 1.
- The return value here is the address of the B-Tree node where *k* is found and its index within the node.
- The procedure accesses $O(h) = O(\log_t n)$ disk pages, where *h* is the height of the B-tree and *n* is the number of nodes in the B-tree

```
B-TREE-SEARCH(x, k)
1  i = 1
2  while i ≤ x.n and k > x.key_i
3      i = i + 1
4  if i ≤ x.n and k == x.key_i
5      return (x, i)
6  elseif x.leaf
7      return NIL
8  else DISK-READ(x.c_i)
9      return B-TREE-SEARCH(x.c_i, k)
```

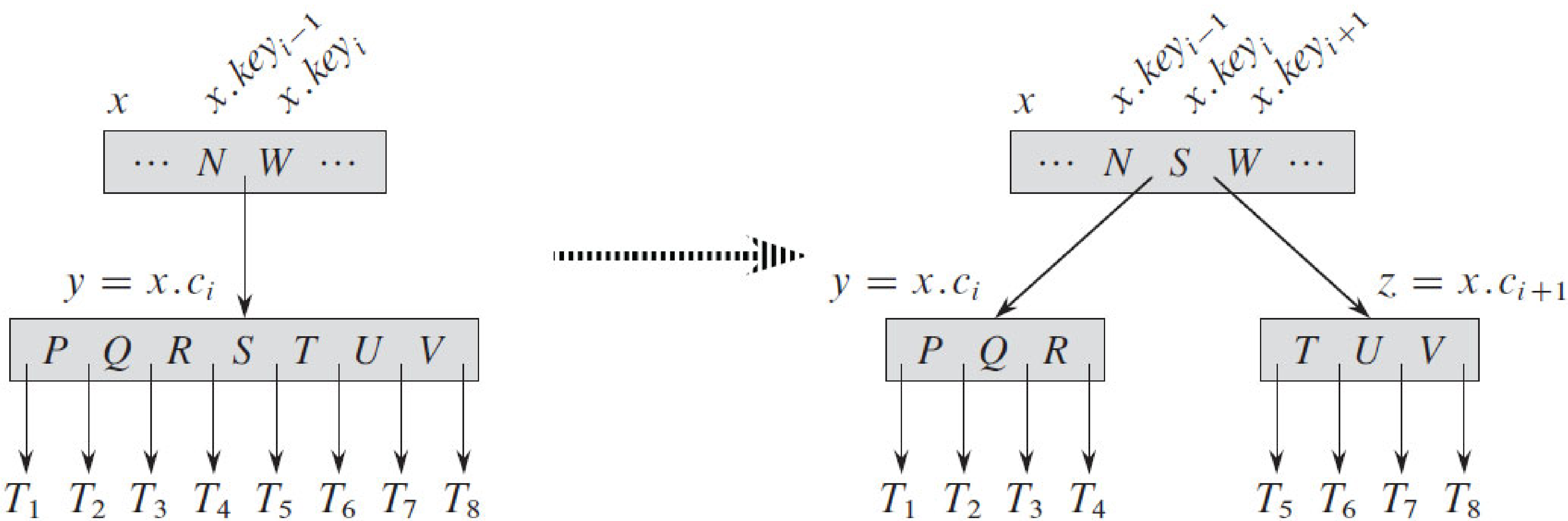


Inserting a Key into a B-Tree

- With a B-tree, however, we cannot simply create a new leaf node and insert it, as the resulting tree would fail to be a valid B-tree.
- We insert the new key into an existing **leaf** node.
- Since we cannot insert a key into a leaf node that is full, we introduce an operation that ***splits*** a full node y (having $2t$ - 1 keys) around its ***median key*** $y.key_t$ into two nodes having only t -1 keys each.
- The median key moves up into y's parent to identify the dividing point between the two new trees. But if y's parent is also full, we must split it before we can insert the new key.
- To split a full root, we will first make the root a child of a new empty root node, so that we can use B-TREE-SPLIT-CHILD.
  - The tree thus grows in height by one; splitting is the only means by which the tree grows.

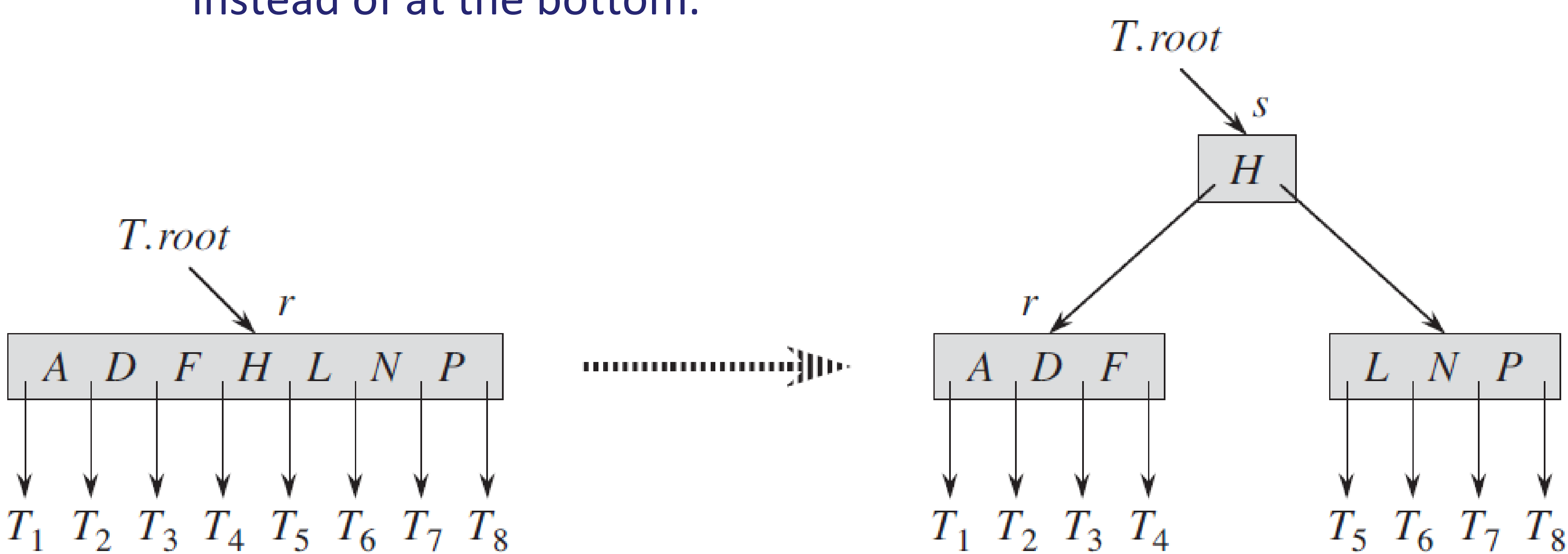
Example of Splitting a Node

- Splitting a node with $t = 4$. Node $y = x.c_i$ splits into two nodes, y and z, and the median key S of y moves up into y's parent



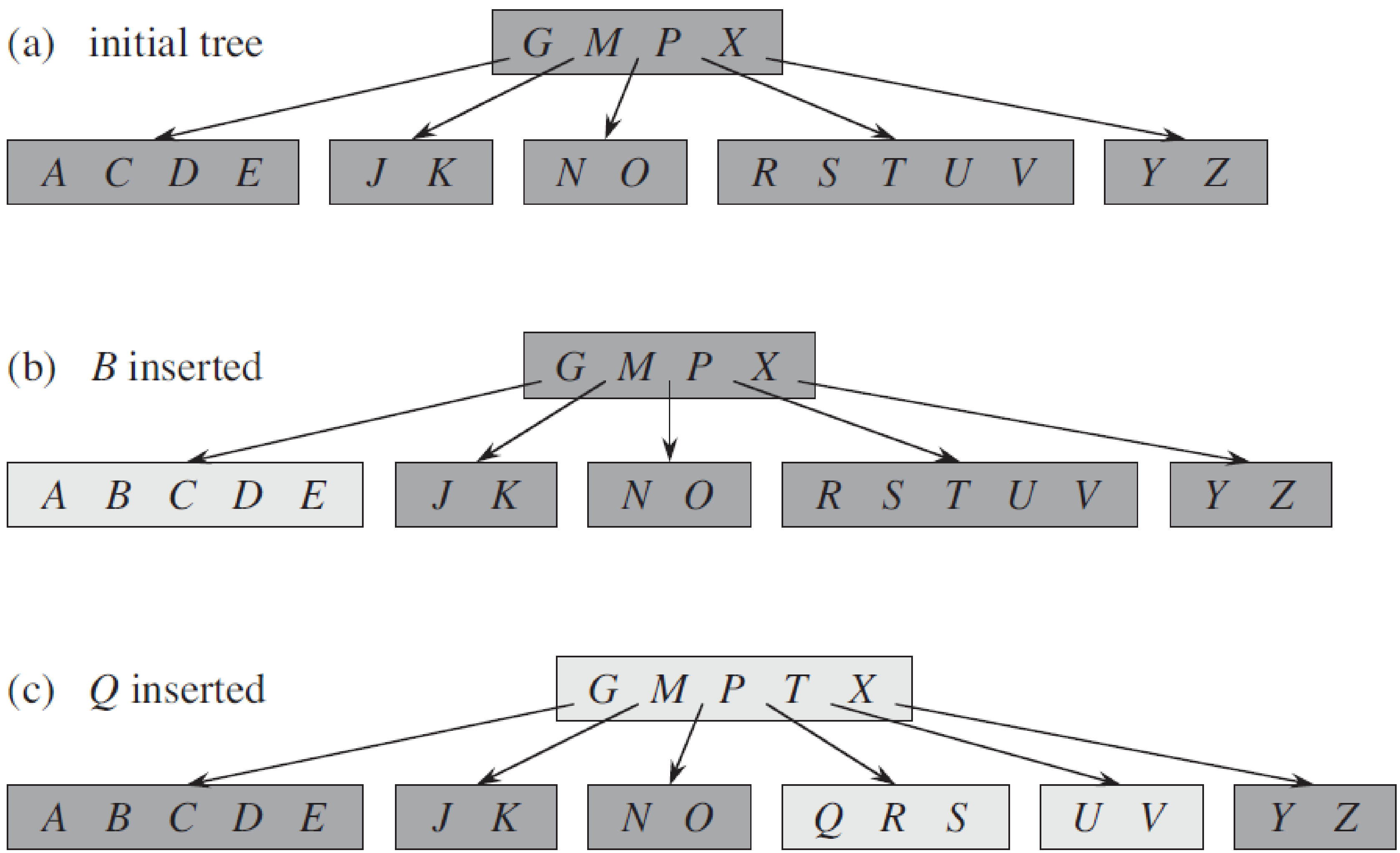
Example of Splitting the Root

- Splitting the root with $t = 4$.
- Root node r splits in two.
- A new root node s is created. The new root contains the median key of r and has the two halves of r as children.
- The B-tree grows in height by one when the root is split.
  - Unlike a binary search tree, a B-tree increases in height at the top instead of at the bottom.



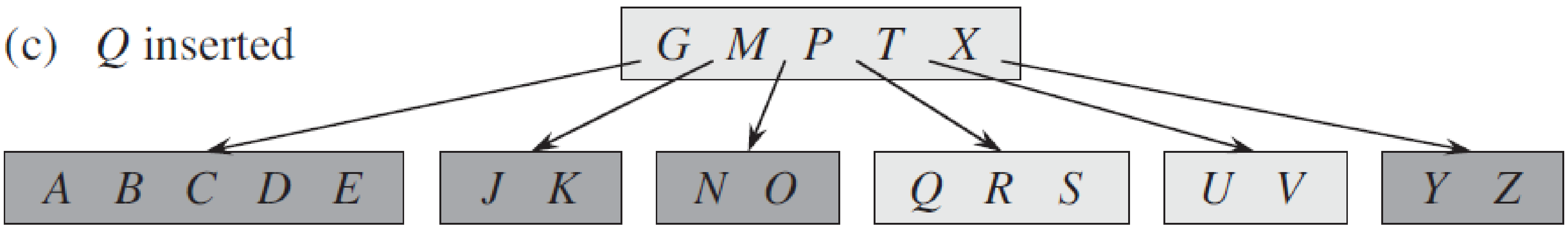
Example of Inserting a Key (1 of 2)

- The minimum degree *t* for this B-tree is 3, so a node can hold at most 5 keys and at least 2 keys. A key is inserted in an existing leaf node

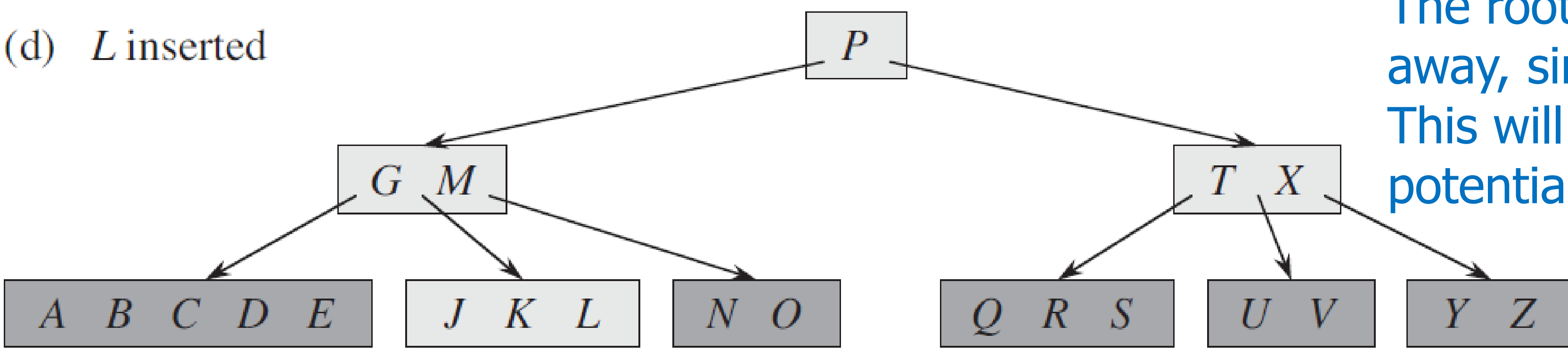


Example of Inserting a Key (2 of 2)

(c) *Q* inserted

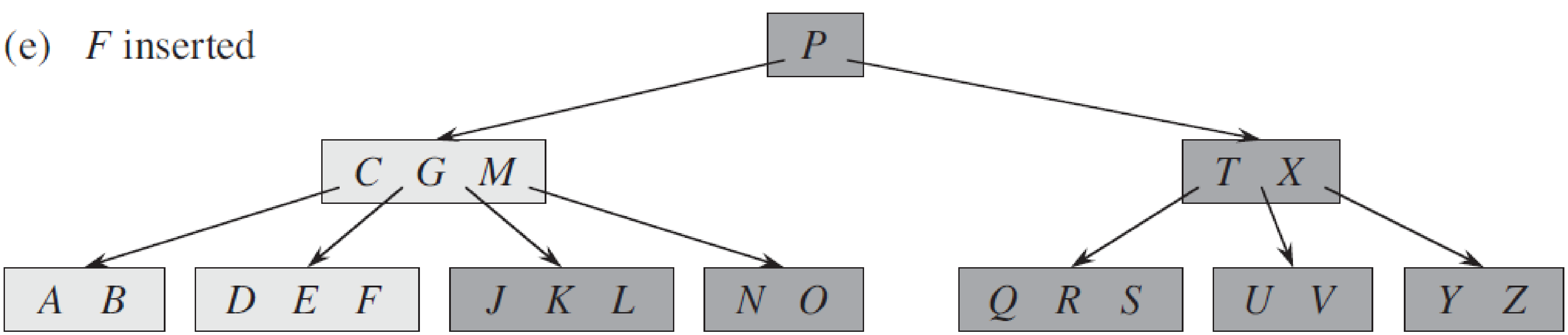


(d) *L* inserted



The root splits right away, since it is full. This will avoid any potential backtracking.

(e) *F* inserted

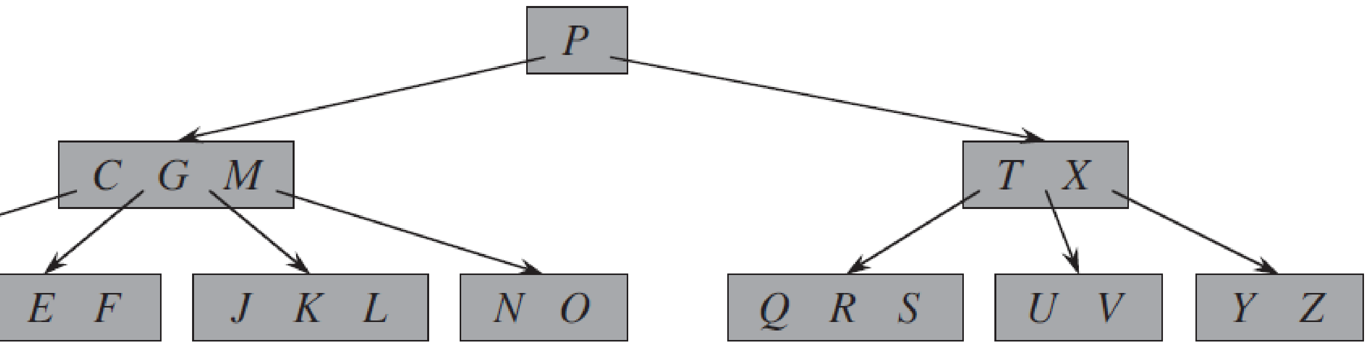


Deleting a Key from a B-Tree

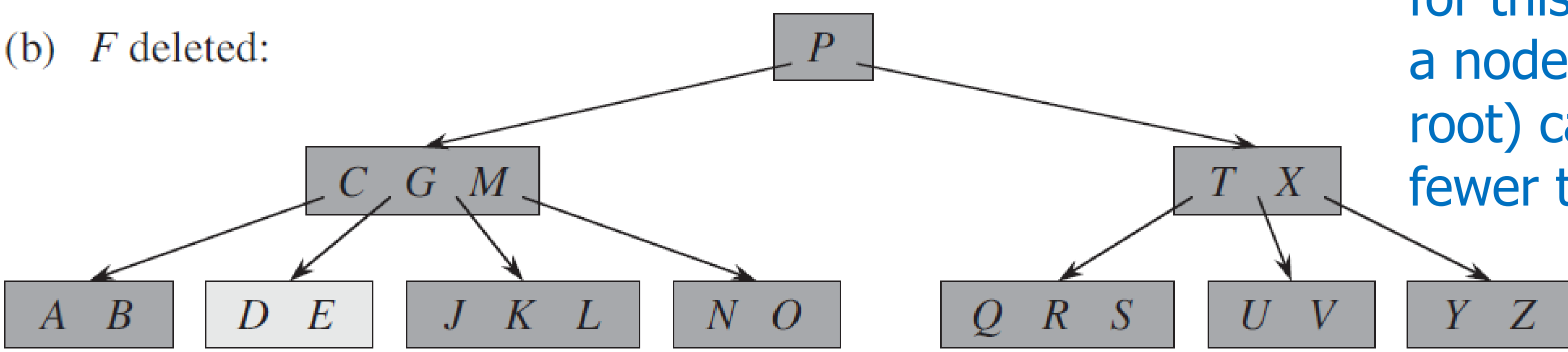
- As in insertion, we must guard against deletion producing a tree whose structure violates the B-tree properties.
  - Just as we had to ensure that a node didn't get too big due to insertion, we must ensure that a node doesn't get too small during deletion (except that the root is allowed to have fewer than the minimum number $\boldsymbol{t}$ **- 1** of keys).
- Deletion from a B-tree is a little more complicated than insertion, because we can delete a key from any node—not just a leaf—and when we delete a key from an internal node, we will have to rearrange the node's children.

Example of Deleting a Key (1 of 2)

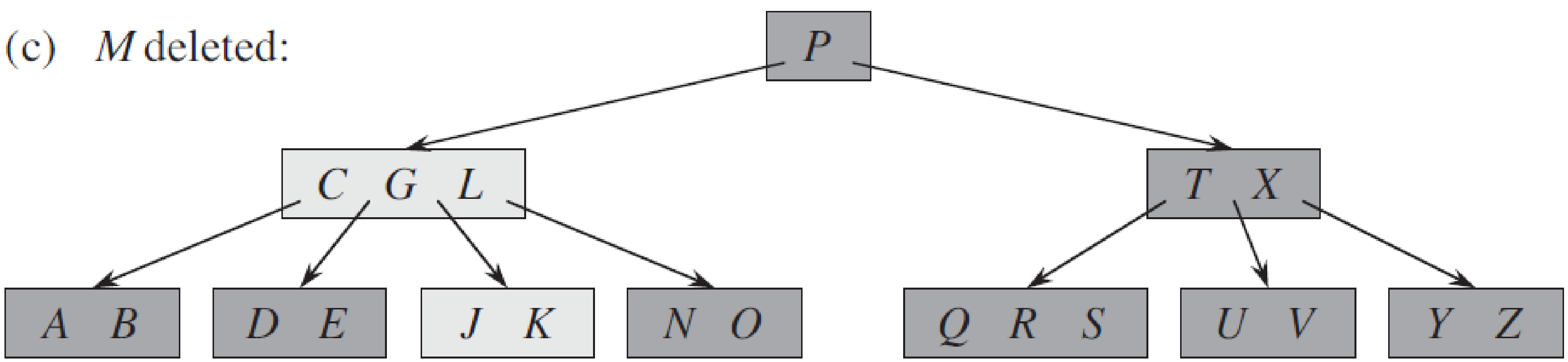
(a) initial tree



(b) *F* deleted:

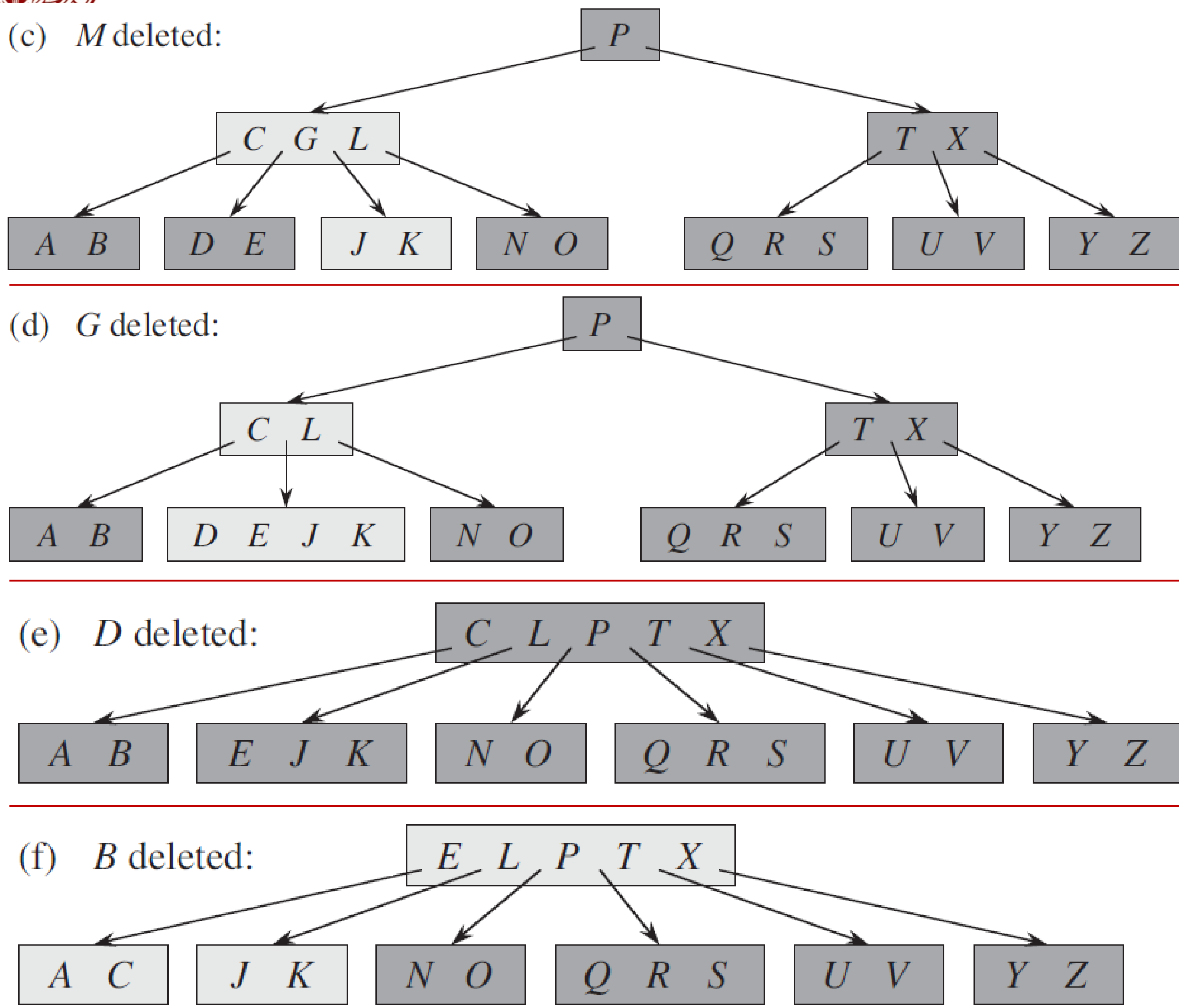


(c) *M* deleted:



- The minimum degree *t* for this B-tree is 3, so a node (other than the root) cannot have fewer than 2 keys.

Example of Deleting a Key (2 of 2)



Even merge here is not required, but it is a good practice to merge nodes with t-1 keys to avoid a potential backtracking as sometimes a key may have to be moved into a child node as with deleting key B here.