Advanced Algorithms

Course Code: 21CS7E12
Module 4

TEXT BOOK

Introduction to Algorithms- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein PHI, 3rd Edition, 2009

Module 4
B-trees and Fibonacci Heaps

1.Introduction and Definition of B-trees
2 Basic operations of B-trees
3 Introduction to Fibonacci heaps
4 Structure of Fibonacci heaps
5 Mergeable-heap operations

Module 4
B Tree

1.Introduction and Definition of B-trees

2 Basic operations of B-trees

Introduction to B-Tree

- B-trees are balanced search tree.
- More than two children are possible.

- B-Tree, stores all information in the leaves and stores only keys and Child pointer.

- If an internal B-tree node x contains n[x] keys then x has n[x]+1 children.

Example of B-Tree

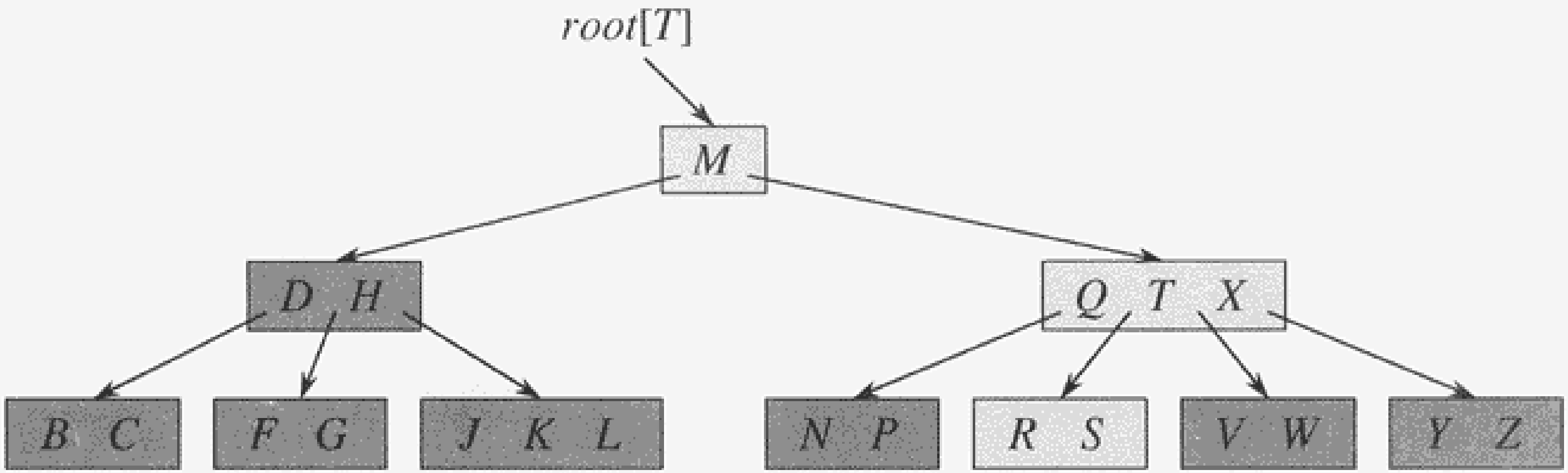


Figure 18.1 A B-tree whose keys are the consonants of English. An internal node x containing $n[x]$ keys has $n[x] + 1$ children. All leaves are at the same depth in the tree. The lightly shaded nodes are examined in a search for the letter R.

Another example of B-Tree

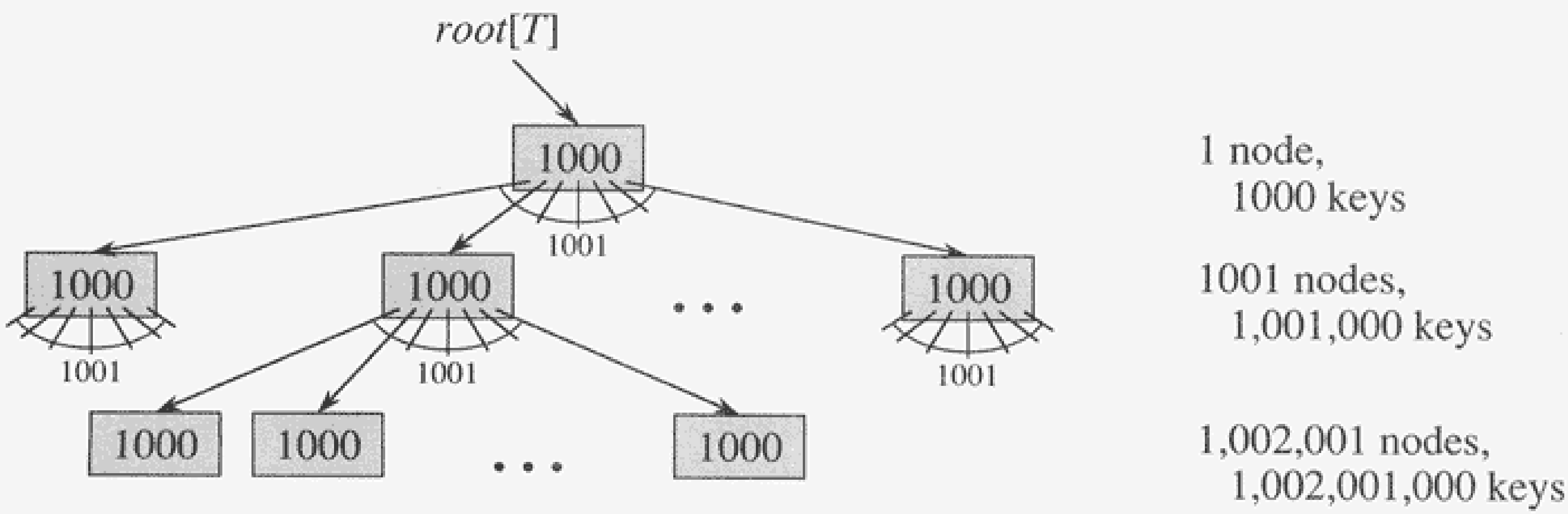


Figure 18.3 A B-tree of height 2 containing over one billion keys. Each internal node and leaf contains 1000 keys. There are 1001 nodes at depth 1 and over one million leaves at depth 2. Shown inside each node x is $n[x]$, the number of keys in x.

Application of B-Tree

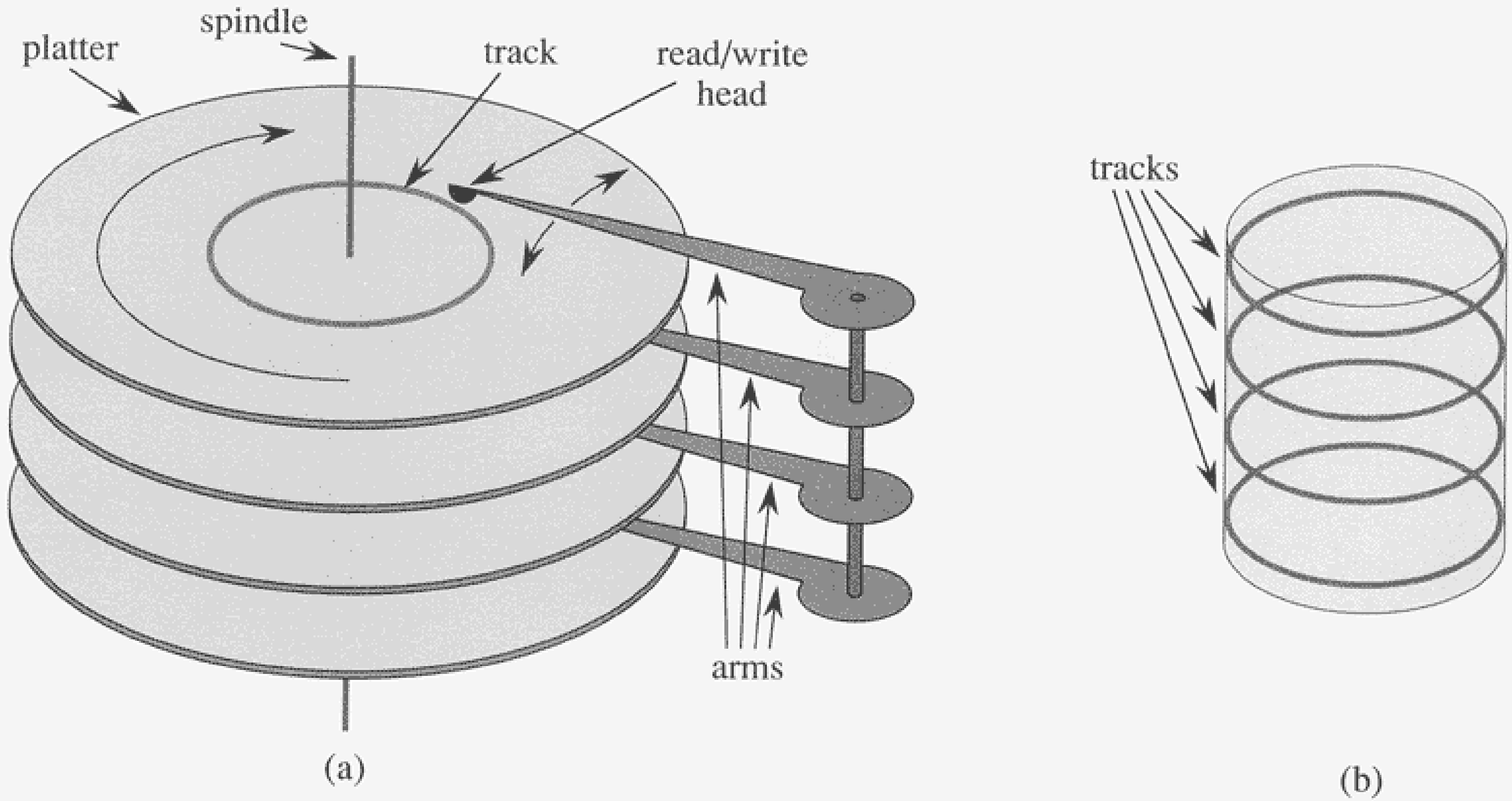


Figure 18.2 **(a)** A typical disk drive. It is composed of several platters that rotate around a spindle. Each platter is read and written with a head at the end of an arm. The arms are ganged together so that they move their heads in unison. Here, the arms rotate around a common pivot axis. A track is the surface that passes beneath the read/write head when it is stationary. **(b)** A cylinder consists of a set of covertical tracks.

It designed to work on magnetic disks or other (direct access) secondary storage devices.

Properties of B-Tree

- A B-Tree T is a rooted tree having the following properties:
  1. Every node x has the following fields:
     1. n[x] the no. of keys currently stored in node x
     2. The n[x] keys themselves, stored in non-decreasing order, so that $key_1[x] \le key_2[x]\ldots\ldots \le key_{n-1}[x] \le key_n[x]$.
     3. Leaf[x], a Boolean value that is TRUE if x is a leaf and FALSE if x is an internal node.
  2. Each internal node x also contains n[x]+1 pointers (Childs) $c_1[x]$, $c_2[x]$,---------$c_{n[x]+1}[x]$.
  3. All leaves have the same depth, which is the tree's height h.

Properties of B-Tree (cont.)

4. There are lower and upper bounds on the no. of keys a node can contains: these bounds can be expressed in terms of a fixed integer $t \geq 2$ called the minimum degree of B-Tree.
   - Every node other than the root must have at least t-1 keys, then root has at least t children if the tree is non empty the root must have at least one key.
   - Every node can contain at most 2t-1 keys. Therefore, an internal node can have at most 2t children we say that a node is full if it contains exactly 2t-1 keys.

Height of B-tree

- Theorem:

  If $n \geq 1$, then for any n-key B-tree T of height h and minimum degree $t \geq 2$,

$$h \leq \log_t (n+1)/2$$

- Proof:

– The root contains at least one key

– All other nodes contain at least t-1 keys.

– There are at least 2 nodes at depth 1, at least 2t nodes at depth 2, at least $2t^{i-1}$ nodes at depth i and $2t^{h-1}$ nodes at depth h

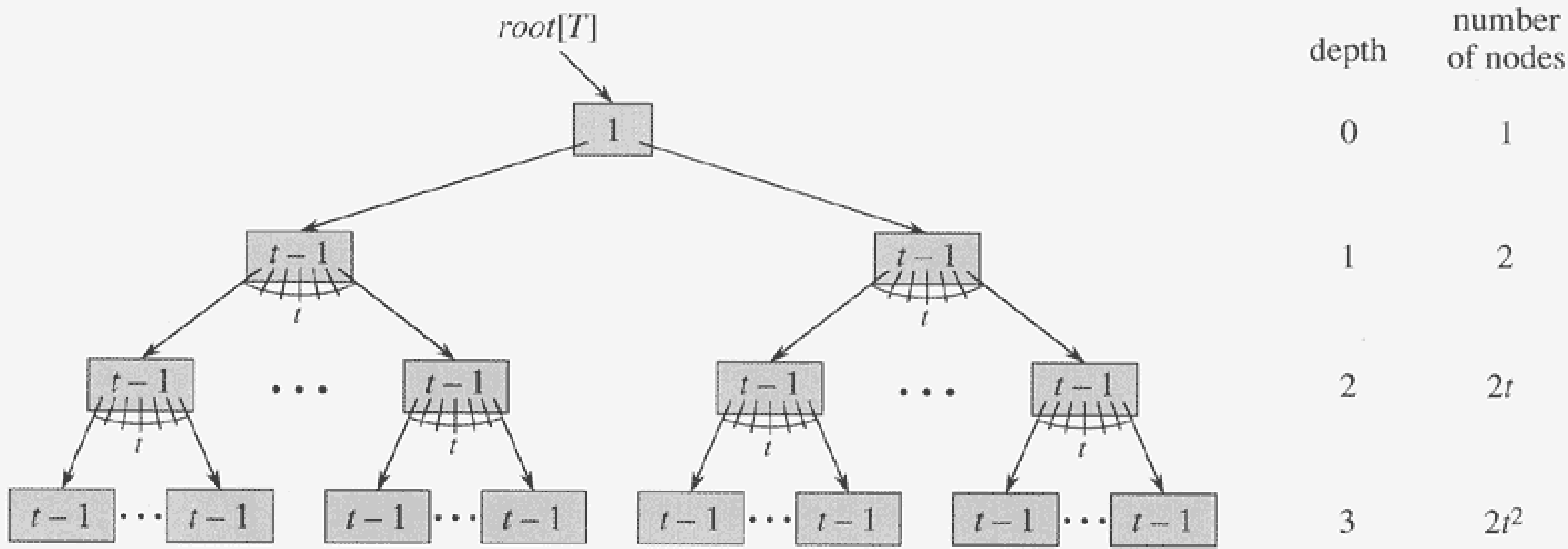


Figure 18.4 A B-tree of height 3 containing a minimum possible number of keys. Shown inside each node x is $n[x]$.

Basic operation on B-tree

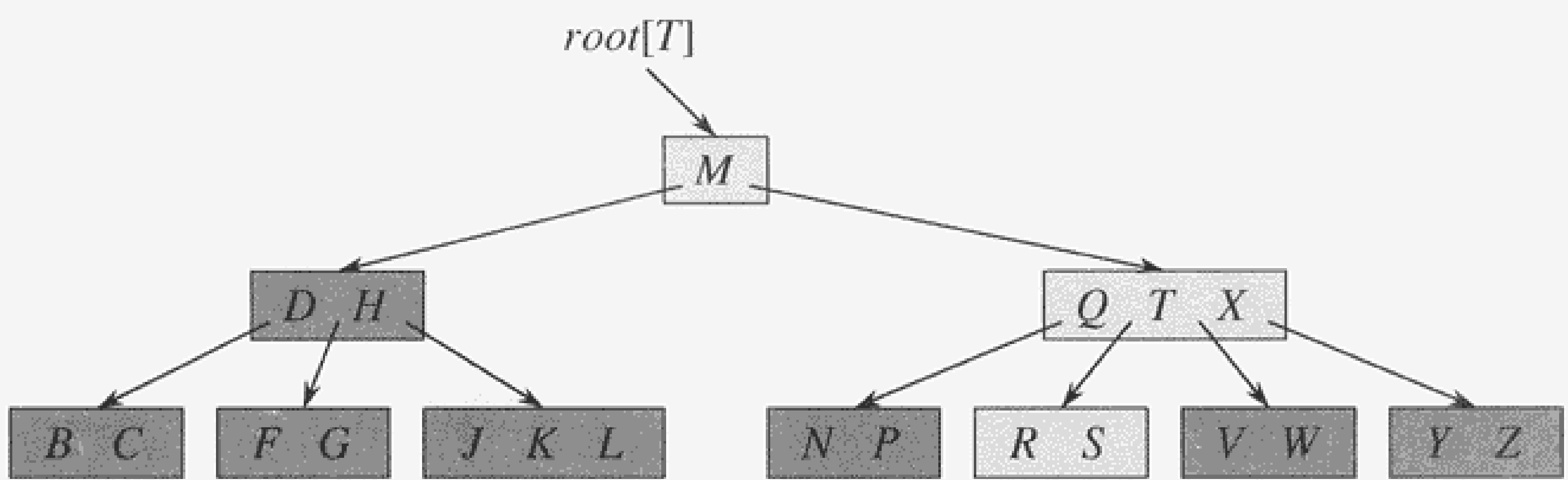
- B-TREE-SEARCH :-Searching in B Tree
- B-TREE-INSERT :-Inserting key in B Tree
- B-TREE-CREATE :-Creating a B Tree
- B-TREE-DELETE :- Deleting a key from B Tree

X is a subtree and k is searching element

```
B-TREE-SEARCH(x, k)
i ← 1
while i ≤ n[x] and k > key_i[x]
    do i ← i + 1
if i ≤ n[x] and k = key_i[x]
    then return (x, i)
if leaf[x]
    then return NIL
    else DISK-READ(c_i[x])
         return B-TREE-SEARCH(c_i[x], k)
```

Complexity=O(t)

Complexity= $O(\log_t n)$



Creating an empty B tree

```
B-TREE-CREATE(T)
x ← ALLOCATE-NODE()
leaf[x] ← TRUE
n[x] ← 0
DISK-WRITE(x)
root[T] ← x
```

Insert

- Cannot just create a new leaf node and insert it
  – resulting tree is not B-tree
- Insert new key into an existing leaf node
- If leaf node is full
  – Split full node y (with 2t-1) keys around its median $key_t[y]$ into two nodes each having t-1 keys
  – Move the median key into y's parent.
  – If parent is full, recursively split, all the way to the root node if necessary.
  – If root is full, split root - height of tree increase by one.

INSERT

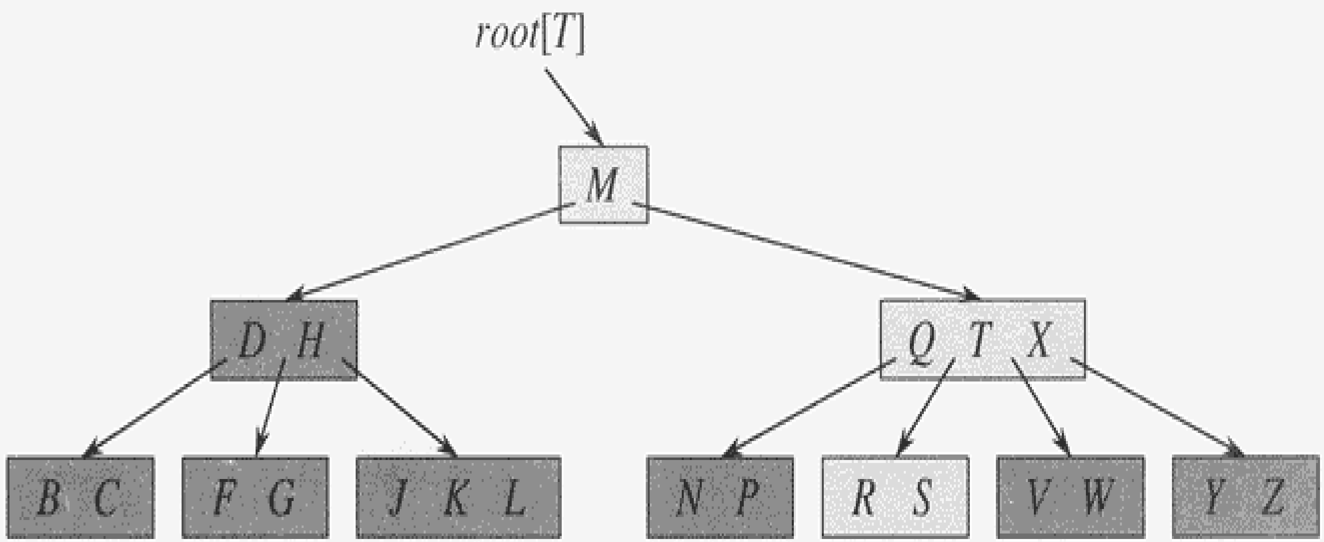
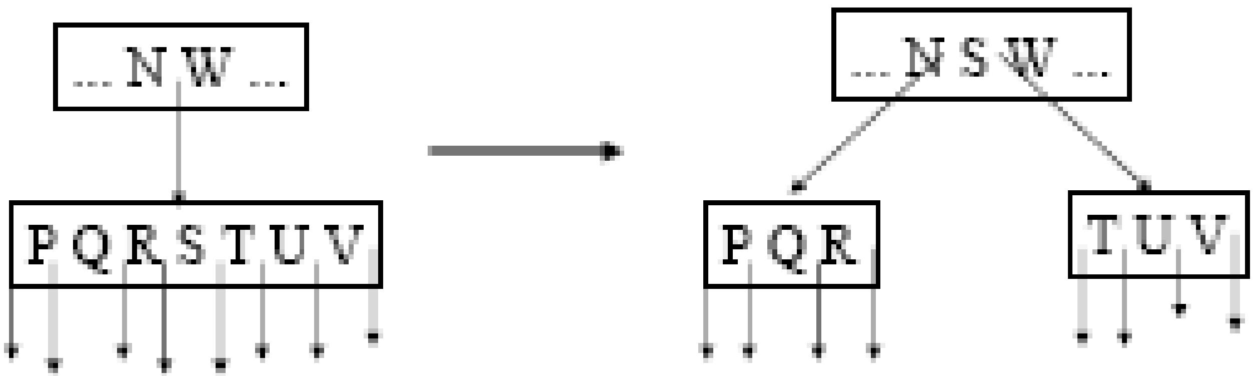


Figure 18.1 A B-tree whose keys are the consonants of English. An internal node x containing $n[x]$ keys has $n[x] + 1$ children. All leaves are at the same depth in the tree. The lightly shaded nodes are examined in a search for the letter R.

Split



- Splitting a node with t=4

B-TREE-SPLIT-CHILD ALGORITHM

```
B-TREE-SPLIT-CHILD(x,i,y)
z← ALLOCATE-NODE()
leaf [z]← leaf [y]
n[z]←t-1
for j← 1 to t-1
      do key_j [z] ← key_{j+t} [y]
if not leaf [y]
      then for j←1 to t
                  do c_j [z]← c_{j+t} [y]
n[y]← t-1
```

cont…….

10. for j $\leftarrow$ n[x] +1 downto i+1
11. do c_{j+1} [x] $\leftarrow$ c_j [x]
12. c_{j+1} [x] $\leftarrow$ z
13. for j $\leftarrow$ n[x] downto i
14. do key_{j+1} [x] $\leftarrow$ key_j [x]
15. key_i [x] $\leftarrow$ key_t [y]
16. n[x] $\leftarrow$ n[x] +1
17. DISK-WRITE(y)
18. DISK-WRITE(z)
19. DISK-WRITE(x)

B-TREE-INSERT ALGORITHM

B-TREE-INSERT(T,k)

1. r ← root[T]
2. If (n[r] = 2t-1)
3. **then** s← Allocate-Node()
4. root[T] ← s
5. leaf[s] ← FALSE
6. n[s]←0
7. c_1[s]←r
8. B-TREE-SPLIT-CHILD(s,1,r)
9. B-TREE-INSERT-NONFULL(s,k)
10. **else** B-TREE-INSERT-NONFULL(r,k)

Algorithm: Insert Key into B-Tree

1. Start at the Root:

Begin at the root node of the B-tree.

2. Traverse the Tree:

- Find the appropriate leaf node where the new key should be inserted.
- While traversing:
  - If the current node is a leaf, stop.
  - If the current node is not a leaf, follow the appropriate child pointer based on the value of the key compared to the keys in the current node.

3. Check Space in the Leaf Node:

- If the leaf node has fewer than $2t-1$ keys:
  - Insert the key in sorted order in the node.
- Otherwise:
  - The leaf node is full. Split it and propagate a key to the parent.

4. Splitting a Full Node:

- If a node overflows (contains 2t keys), split it:
  - Divide the 2t keys into two nodes:
    - Left node: Contains the first $t-1$ keys.
    - Right node: Contains the last $t-1$ keys.
  - Promote the middle key (the t-th key) to the parent node.
- If the parent node also overflows due to this promotion, repeat the splitting process upward, possibly splitting the root.

5. Handle Root Overflow:

- If the root node overflows, create a new root:
  - The new root contains the promoted middle key.
  - The two split nodes become its children.
- This increases the height of the B-tree by 1.

6. Repeat Until Key is Inserted:

- Continue the process until the key is

Process of finding the appropriate position for the key in the tree.

1. Start from the Root:

•Compare the key with the keys in the current node.

•Traverse the child pointer corresponding to the key range where the new key belongs.

2. Repeat Until a Leaf is Reached:

•For each internal node encountered:

- ➢Find the correct child node by comparing the key with the keys in the node.
- ➢Move to that child node.

3. Insert into the Leaf Node:

•Once at the appropriate leaf, insert the key into the correct position based on sorted order.

Example

Consider inserting the key 15 into a B-tree with t=3.

Traverse from the root to the correct child node where 15 belongs.

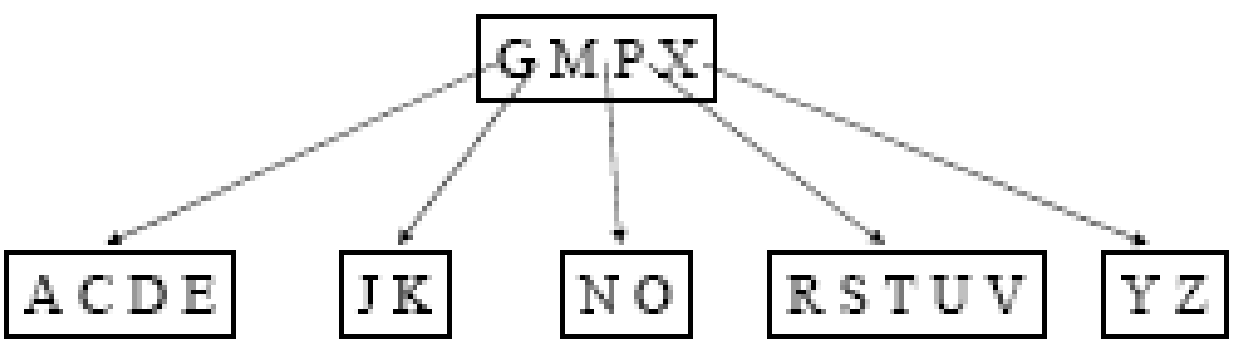
If the leaf node is not full, insert 15 in sorted order.

If the leaf node is full:

Split the node, promoting the middle key to the parent.

Adjust pointers and repeat if necessary.

- Can also insert in a single pass down the tree, instead of going down during search then recursively splitting on the way up
  - Split full nodes encountered on the way down during search.
  - Example:



Initial tree, t=3

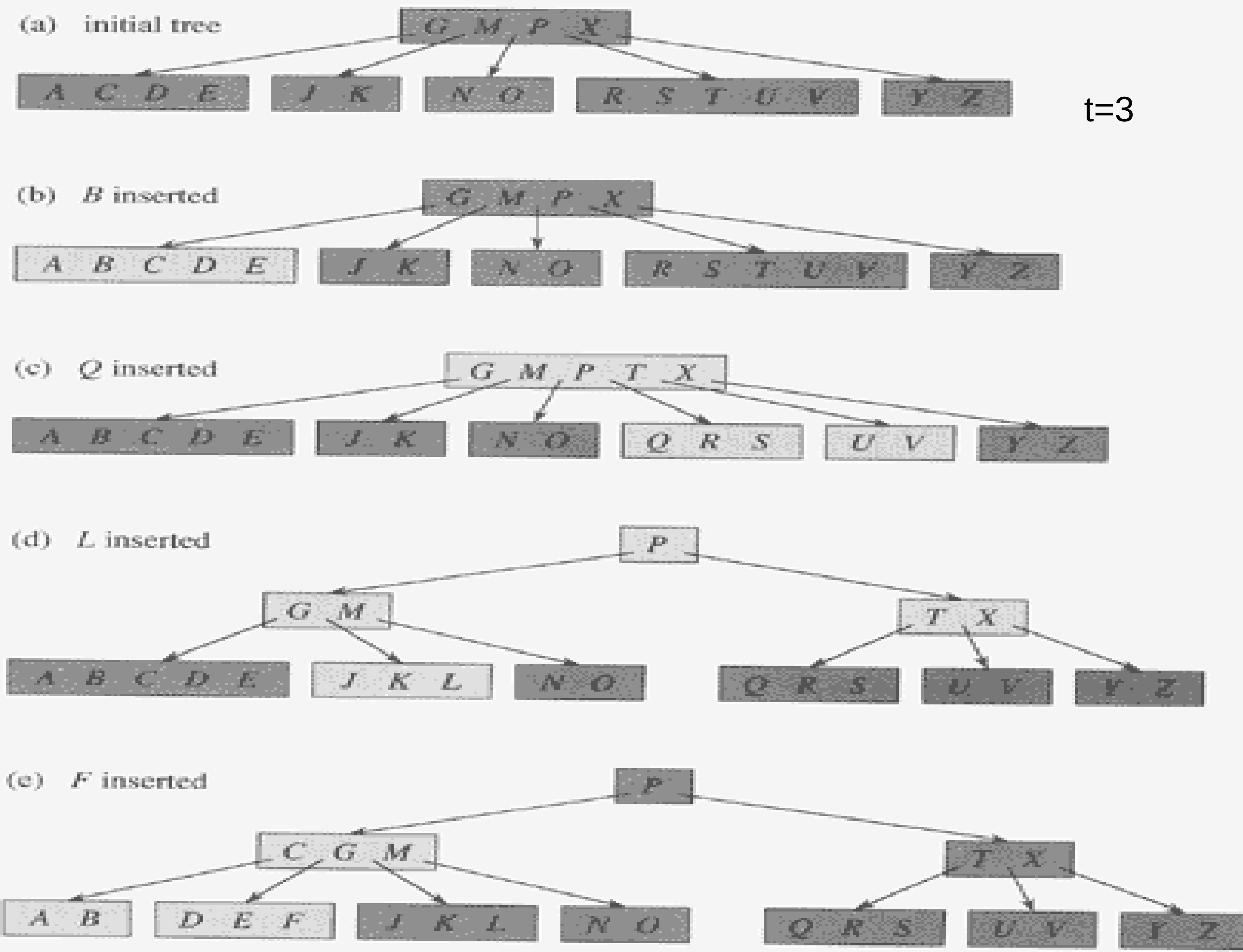
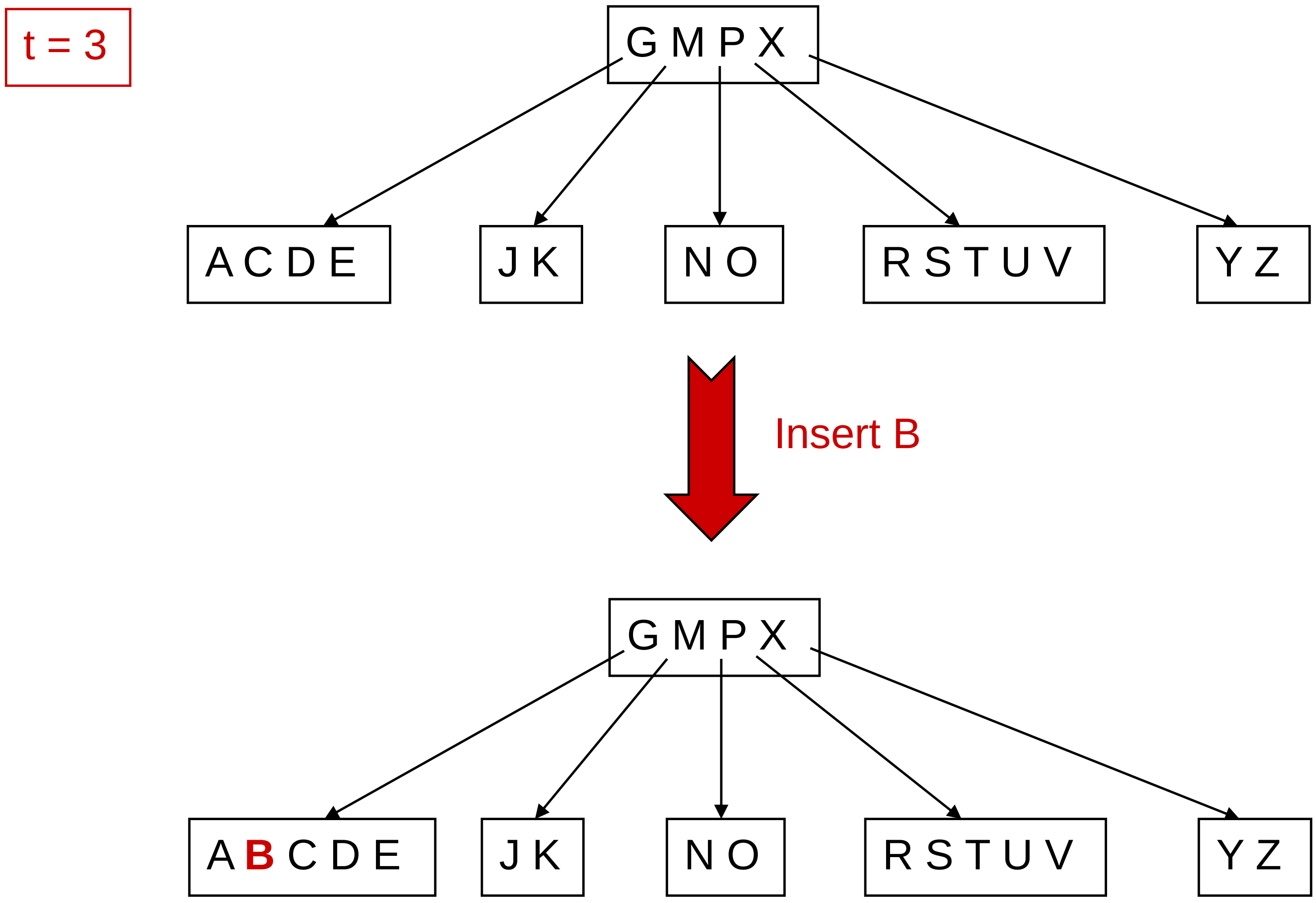
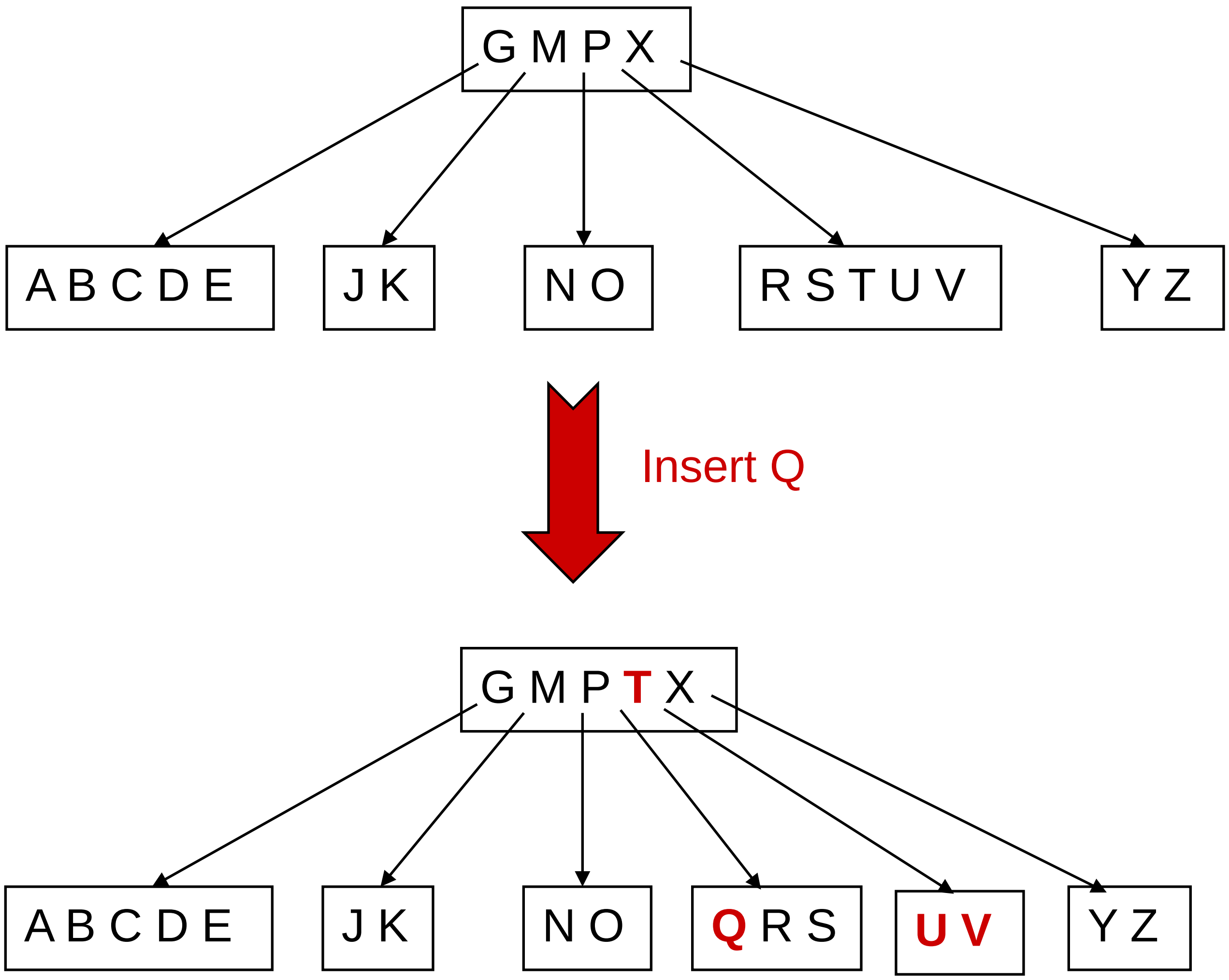


Figure 18.7 Inserting keys into a B-tree. The minimum degree t for this B-tree is 3

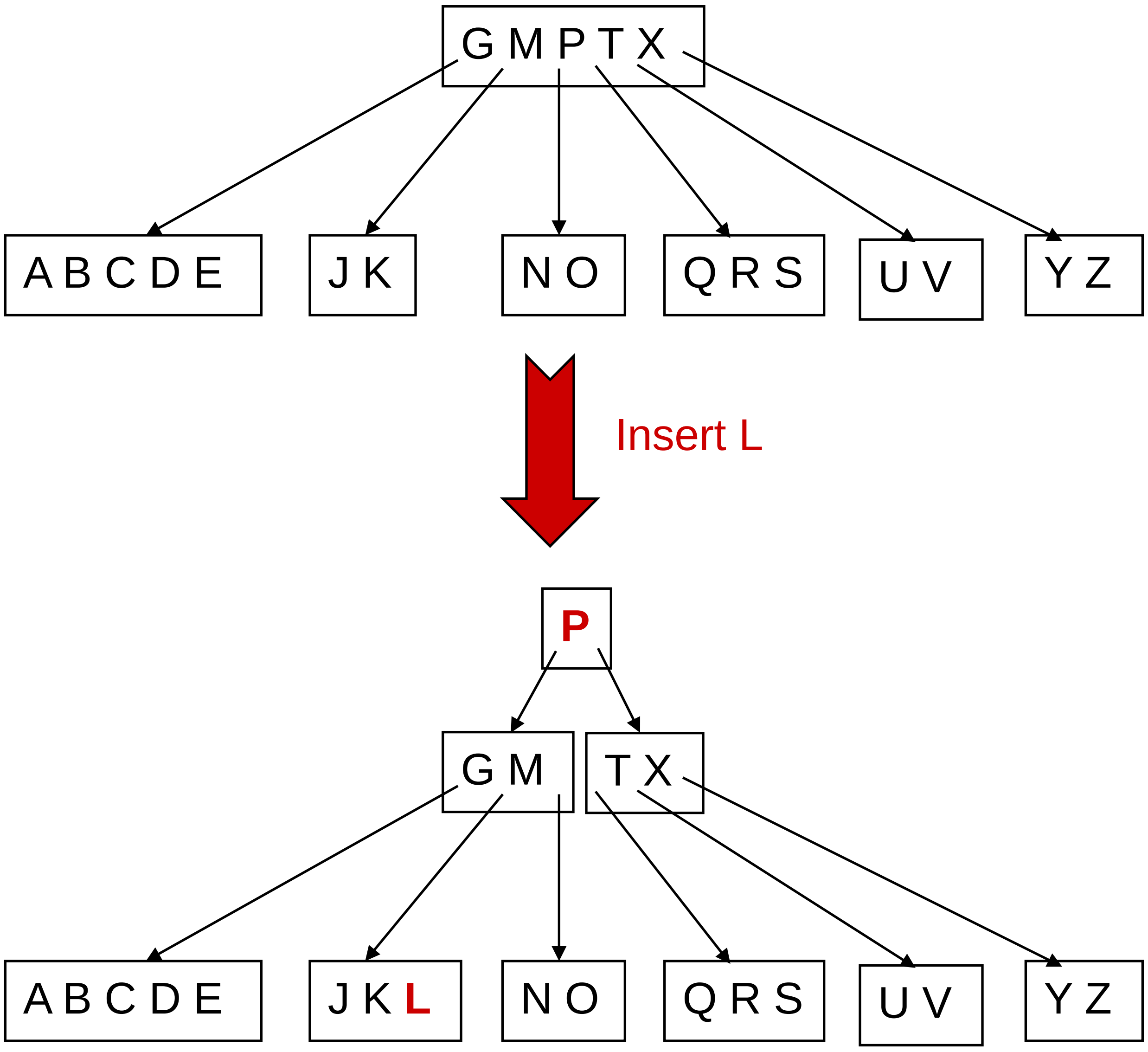
Insert Example



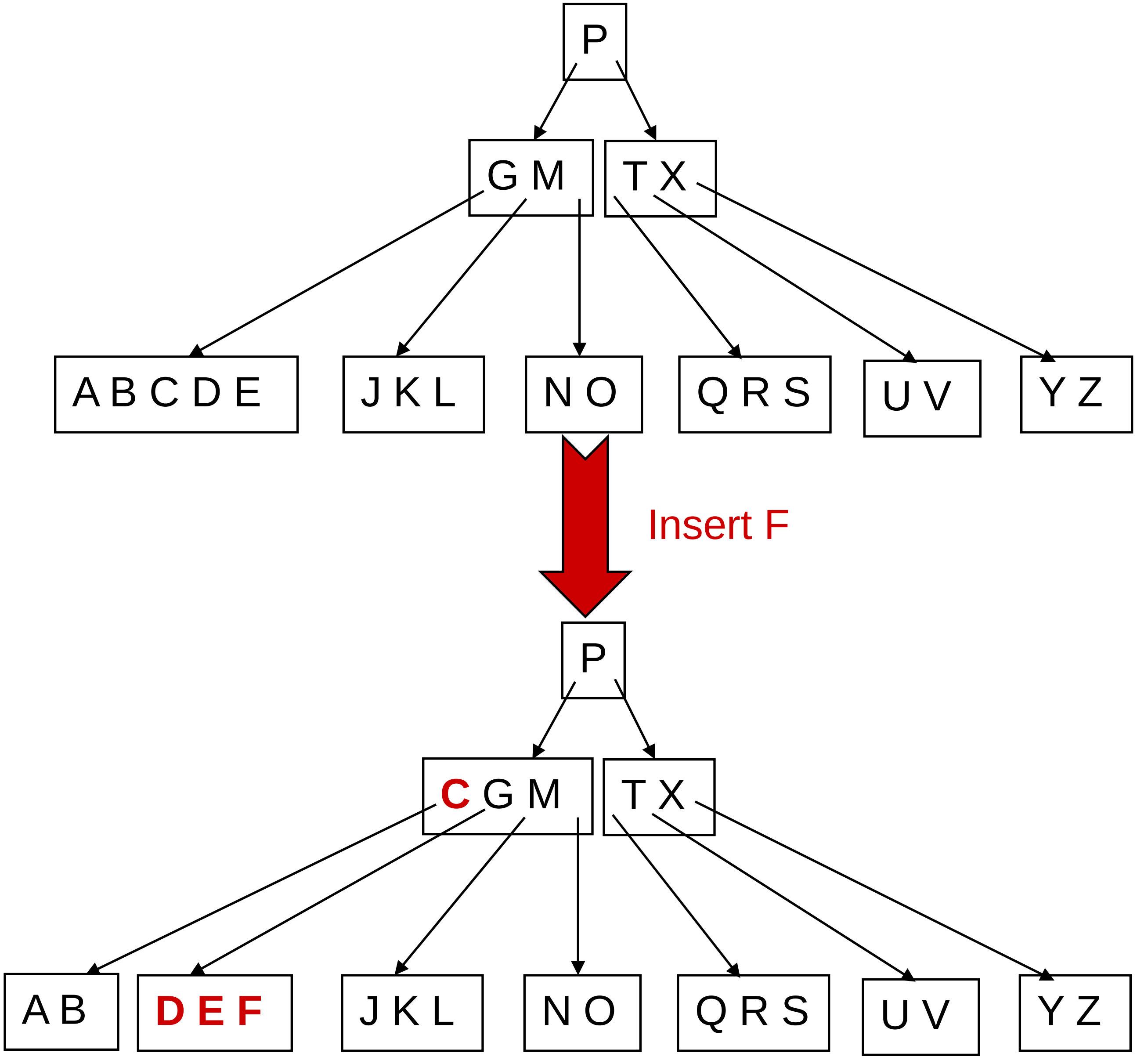
Insert Example (Continued)



Insert Example (Continued)

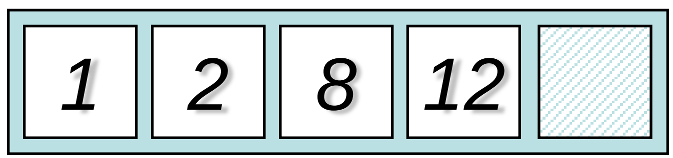


Insert Example (Continued)



Constructing a B-tree

- Suppose we start with an empty B-tree and keys arrive in the following order:1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45
- We want to construct a B-tree of degree 3
- The first four items go into the root:

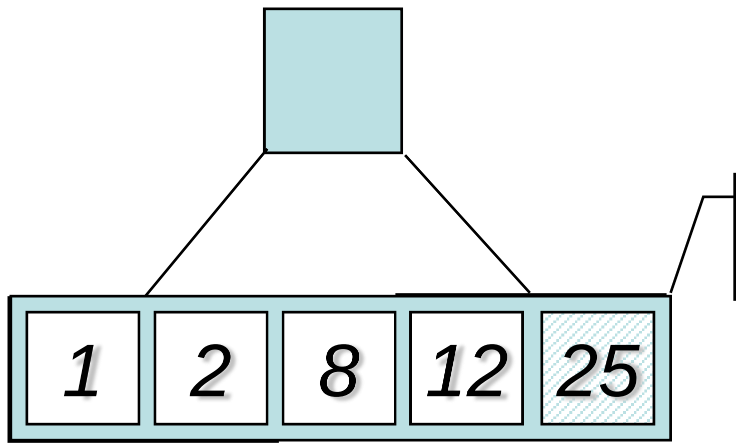


- To put the fifth item in the root would violate condition 5
- Therefore, when 25 arrives, pick the middle key to make a new root

1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

Constructing a B-tree

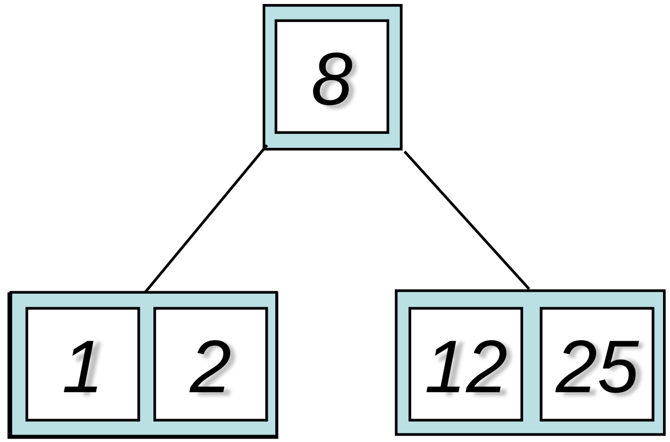
Add 6 to the tree



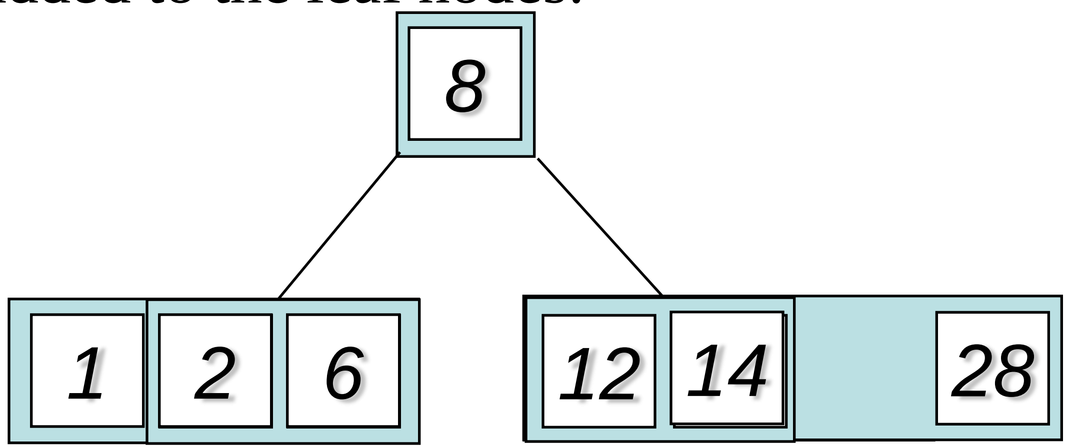
Exceeds Order. Promote middle and split.

1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

Constructing a B-tree (contd.)



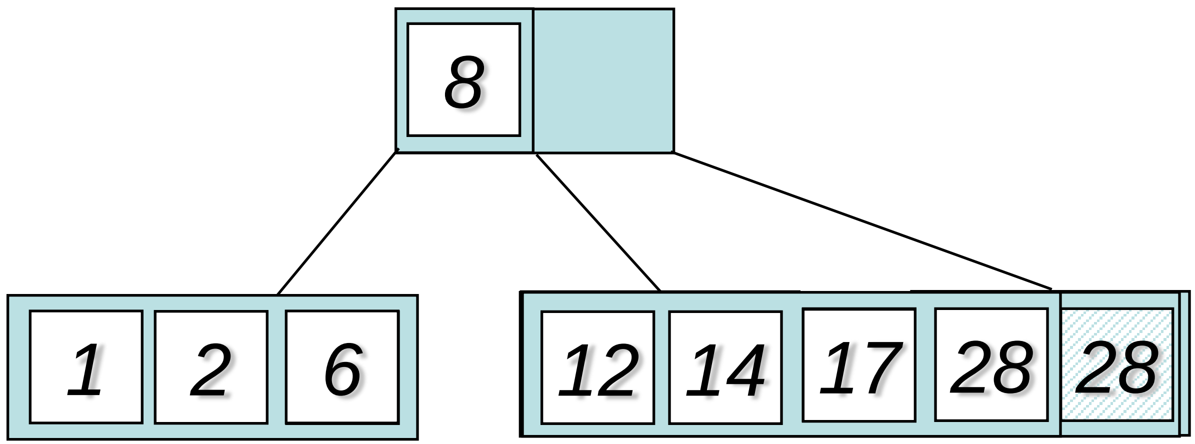
6, 14, 28 get added to the leaf nodes:



1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

Constructing a B-tree (contd.)

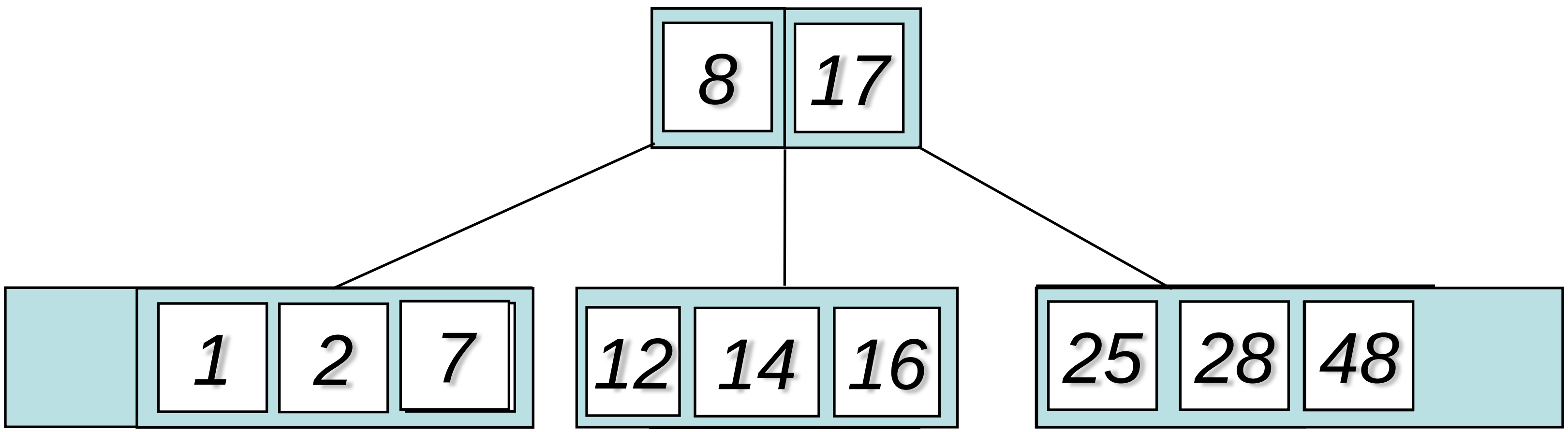
Adding 17 to the right leaf node would over-fill it, so we take the middle key, promote it (to the root) and split the leaf



Constructing a B-tree (contd.)

1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

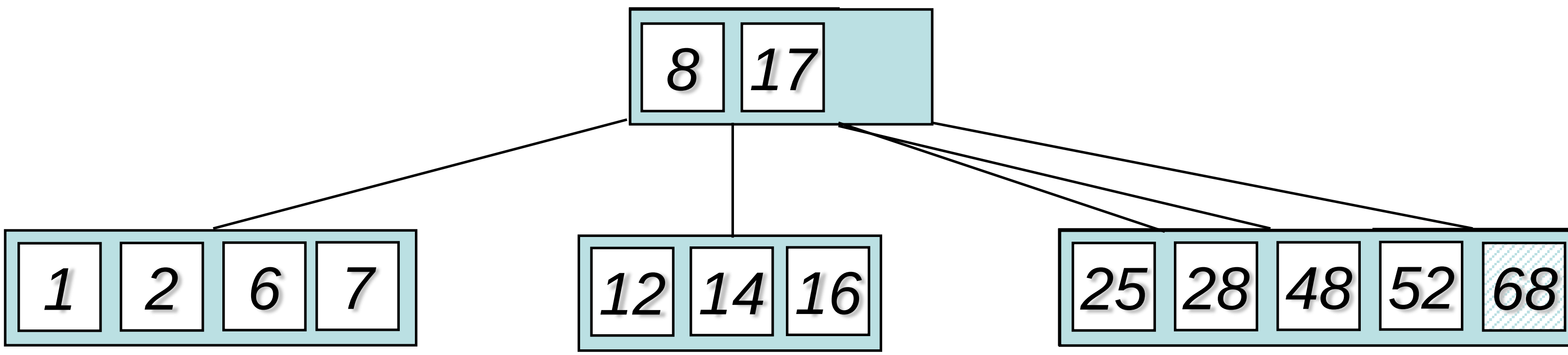
7, 52, 16, 48 get added to the leaf nodes



Constructing a B-tree (contd.)

1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

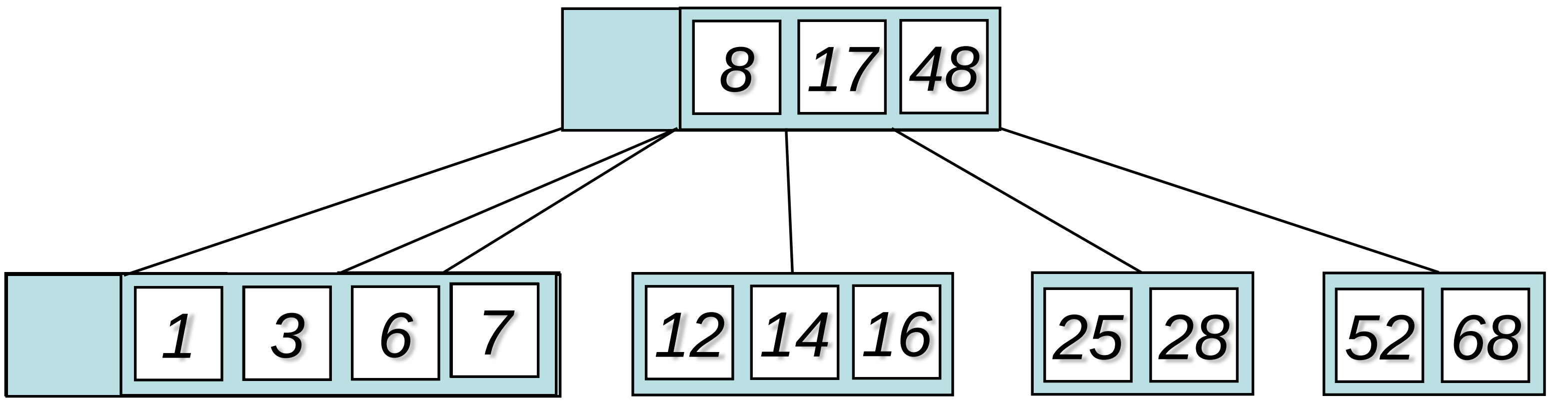
Adding 68 causes us to split the right most leaf, promoting 48 to the root



1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

Constructing a B-tree (contd.)

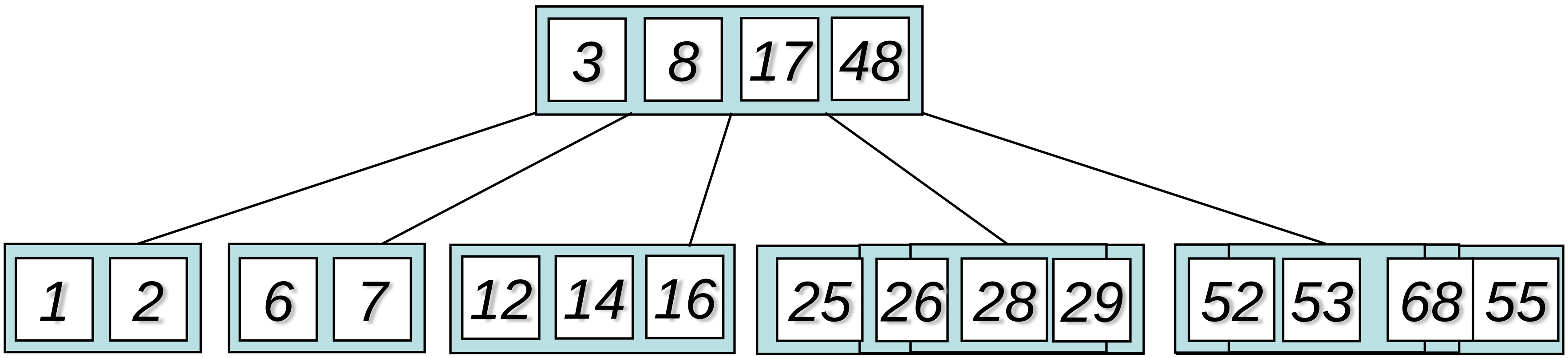
Adding 3 causes us to split the left most leaf



1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

Constructing a B-tree (contd.)

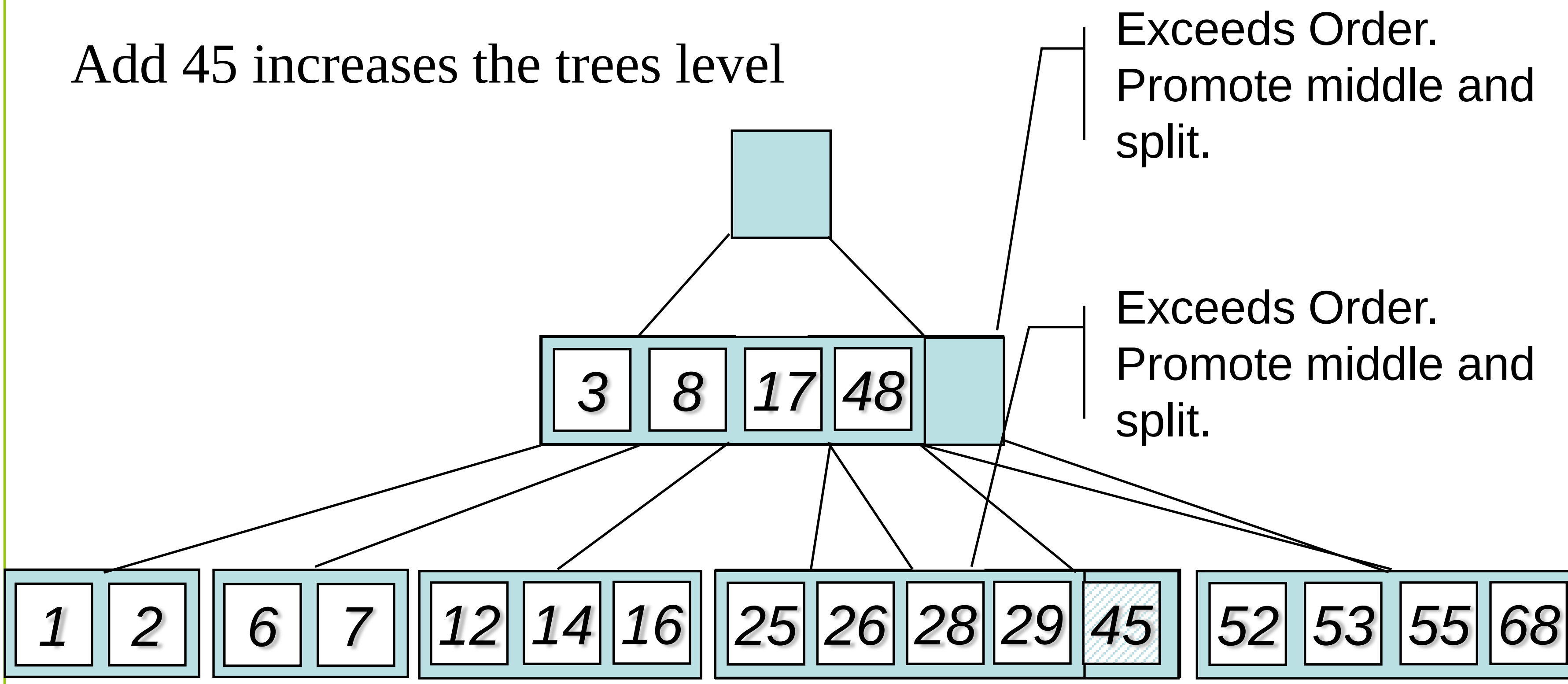
Add 26, 29, 53, 55 then go into the leaves



Constructing a B-tree (contd.)

1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

Add 45 increases the trees level



Exercise in Inserting a B-Tree

1) Insert the following keys in B-tree when t=3 :

- 3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56

2) Difference between B-Tree and Red black tree.

39

Aspect	Red-Black Tree	B-Tree
Definition	A type of self-balancing binary search tree.	A self-balancing search tree designed for databases and storage systems.
Node Structure	Each node has a single key (binary search tree).	Each node can store multiple keys (min: t−1, max: 2t−1).
Child Structure	Each node has at most two children.	Each node can have multiple children (min: t, max: 2t).
Balancing Method	Balance is maintained using **coloring** (red/black) and **rotations** during insertions and deletions.	Balance is maintained by splitting or merging nodes during insertions and deletions.
Height	Height is approximately $O(\log n)$ (binary tree).	Height is also $O(\log n)$, but generally shorter than a red-black tree due to larger branching factor.

Aspect	Red-Black Tree	B-Tree
Use Case	Used for in-memory data structures (e.g., compilers, network routing).	Used for disk-based storage systems (e.g., databases, filesystems).
Traversal Complexity	Traversal involves multiple nodes (binary structure).	Traversal is efficient due to fewer nodes and larger fanout.
Search Efficiency	Search time is $O(\log n)$, but nodes are smaller.	Search time is $O(\log n)$, with fewer disk I/O operations due to multi-key nodes.
Space Utilization	Stores one key per node, less space-efficient.	Optimized for storage efficiency with multiple keys per node.
Memory Access	Frequently accesses small amounts of data, which is CPU-cache-friendly.	Designed for minimizing disk I/O with fewer accesses and larger blocks.
Rotations	Requires frequent rotations during balancing.	No rotations; rebalancing involves splits or merges.
Applications	- Symbol tables in compilers. - Network routing tables.	- Database indexing (e.g., MySQL, PostgreSQL). - Filesystem indexing (e.g., NTFS, EXT).

Example Problem

As a function of the minimum degree t, what is the maximum number of keys that can be stored in a B-tree of height h?

Key properties of a B-tree:

Minimum degree t: This is the minimum number of children a node in the B-tree can have.

Every internal node (except the root) has at least **t** children.

Each internal node can have at most **$2t-1$** keys and $2t$ children.

Height h: This is the number of levels in the tree, with the root being at height 0.

At level i, there can be up to $\mathbf{(2t)^i}$ nodes, and each of these nodes can hold up to **$2t-1$** keys.

Total number of keys:

At level 0, the root can hold up to **$2t-1$ keys.**

At level 1, there are up to $2t$ nodes, each holding up to **$2t-1$ keys.**

At level 2, there are up to $(2t)^2$ nodes, each holding up to **$2t-1$ keys.**

This continues until **level h,**
where Number of nodes$=1+2t+(2t)^2+\cdots+(2t)^h$
Here,

First term (a) = 1
Common ratio (r) = 2t
Number of terms = h+1
Sum of the Geometric Series is given by:
$\mathbf{S=a\cdot(r^n-1)/r-1}$

Substituting these values:
$\mathbf{S=((2t)^{h+1}-1)/2t-1}$

Maximum Number of Keys
Each node can have at most $2t-1$ keys.
So, the maximum number of keys $\mathbf{N_{max}}$ is:
$\mathbf{N_{max}}=(S-1)\cdot(2t-1)$
After simplifying:
Nmax= $\mathbf{(2t)^{h+1}-1}$

This represents the maximum number of keys a B-tree of height h and minimum degree t can hold.

Example Problem

Suppose that we insert the keys {1,2,........,n} into an empty B-tree with minimum degree 2. How many nodes does the final B-tree have?

B-tree Structure

Minimum Degree t=2:

Each node (except the root) contains at least t−1 key and at most 2t−1==3 keys.

The root can have fewer keys but must have at least 1 key (if not empty).

Each internal node has between 2 and 4 children (one more than the number of keys).

Node Splitting:

•A node splits when it becomes full, i.e., when it contains 2t−1=3 keys.

•During splitting:

- The middle key moves up to the parent.
- The node splits into two child nodes.

•This increases the number of nodes in the tree.

Contd…..

Step-by-Step Analysis

1. Keys in the Root

- Initially, the root node can hold up to $2t-1=3$ keys.
- When the root splits, it creates two child nodes, and the tree grows in height.

2. Number of Splits

- For n keys, the total number of splits determines the number of nodes.
- Each split adds a new node to the tree.
- At any point in the tree's growth:
  - The number of nodes in the tree is related to how keys are distributed among all levels.

3. Approximation for Total Nodes

- When inserting n keys into a B-tree:
  - A key distributes across nodes to ensure balance.
  - Each node can hold approximately **2 keys** on average (halfway between $t-1$ and $2t-1$).
- The total number of nodes is approximately: Number of Nodes≈[n/2]

```
B-TREE-INSERT-NONFULL(x, k)
i ← n[x]
if leaf[x]
   then while i ≥ 1 and k < key_i[x]
             do key_{i+1}[x] ← key_i[x]
                i ← i − 1
        key_{i+1}[x] ← k
        n[x] ← n[x] + 1
        DISK-WRITE(x)
   else while i ≥ 1 and k < key_i[x]
             do i ← i − 1
        i ← i + 1
        DISK-READ(c_i[x])
        if n[c_i[x]] = 2t − 1
          then B-TREE-SPLIT-CHILD(x, i, c_i[x])
               if k > key_i[x]
                  then i ← i + 1
        B-TREE-INSERT-NONFULL(c_i[x], k)
```

Deleting from B-Trees

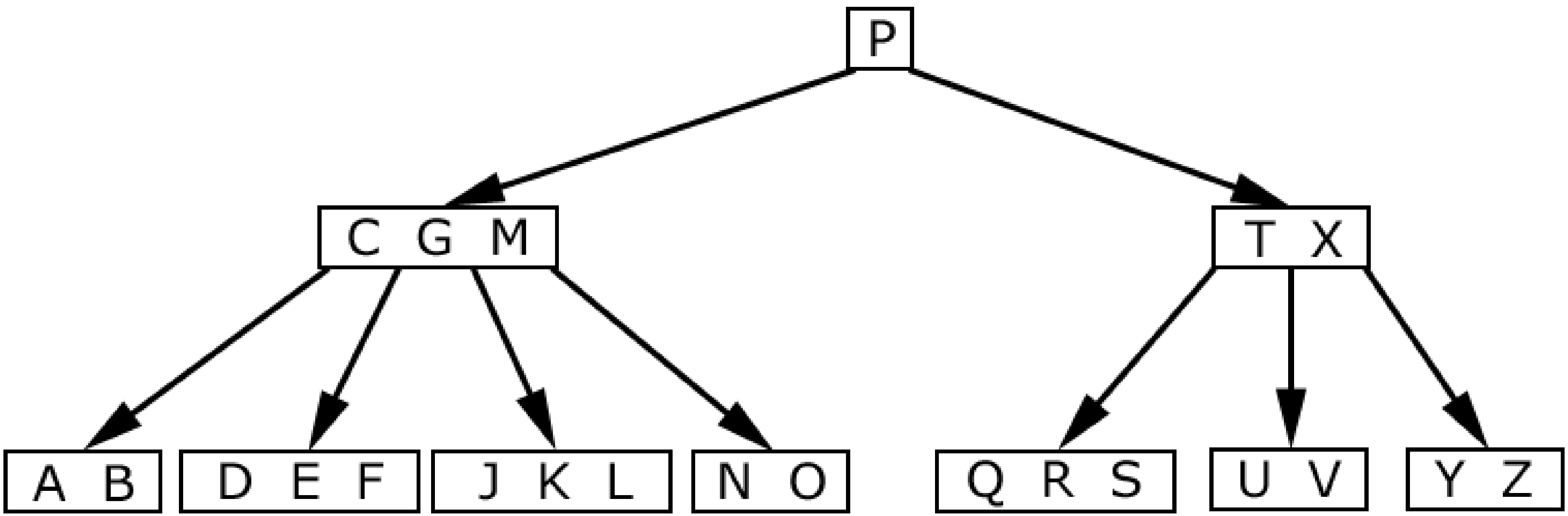
The Concept

- You can delete a key entry from any node.
- ->Therefore, you must ensure that before/after deletion, the B-Tree maintains its properties.
- When deleting, you have to ensure that a node doesn’t get *too small* (minimum node size is T – 1). We prevent this by *combining* nodes together.

Lets look at an example:

We're given this valid B-Tree

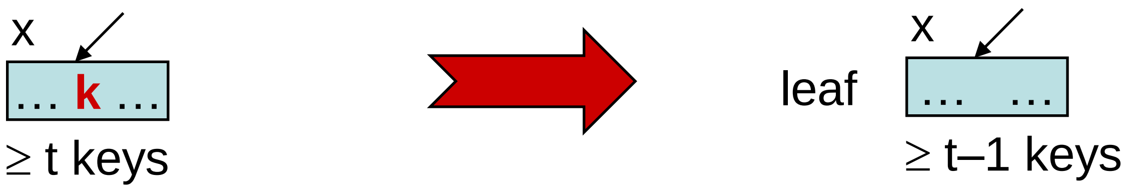
Note: T = 3



Source: Introduction to Algorithms, Thomas H. Cormen

Deletion Cases

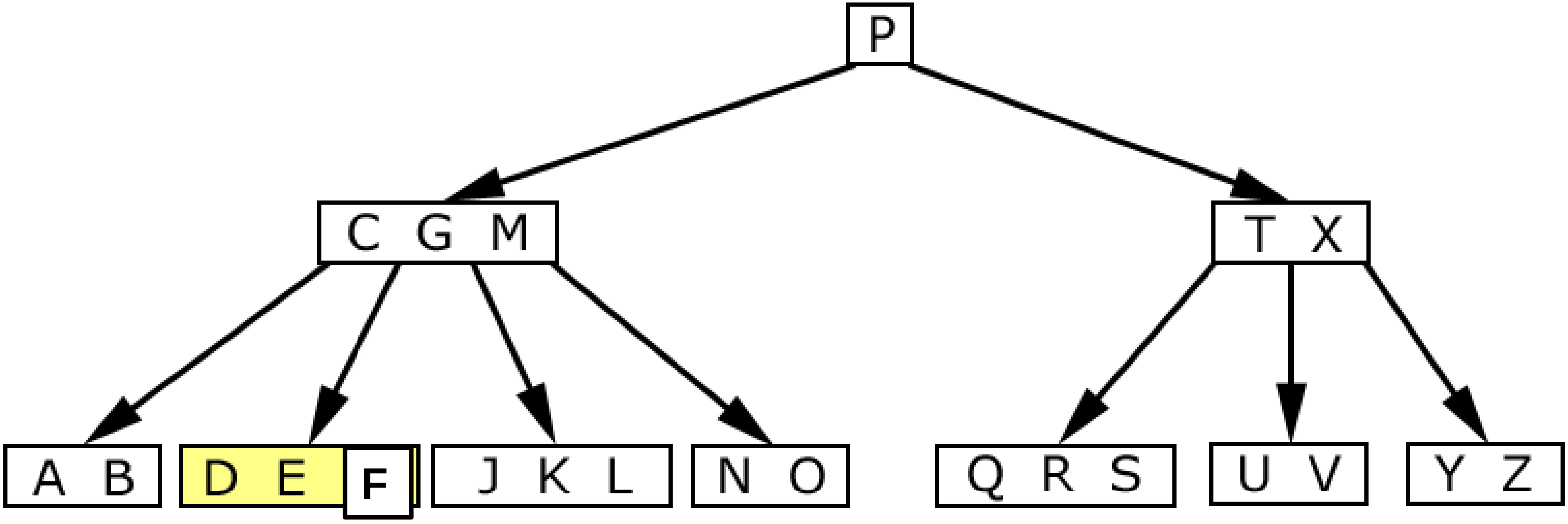
- **Case 1:** If the key k is in node x and x is a leaf node having atleast t keys - then delete k from x.



Simple Deletion

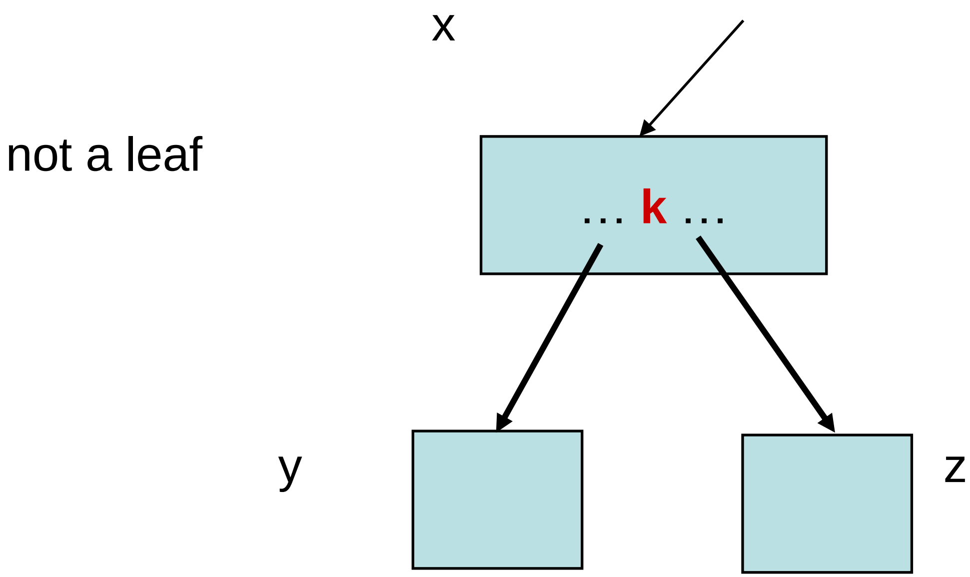
Case 1: We delete “F”

Result: We remove “F” from the leaf node. No further action needed.



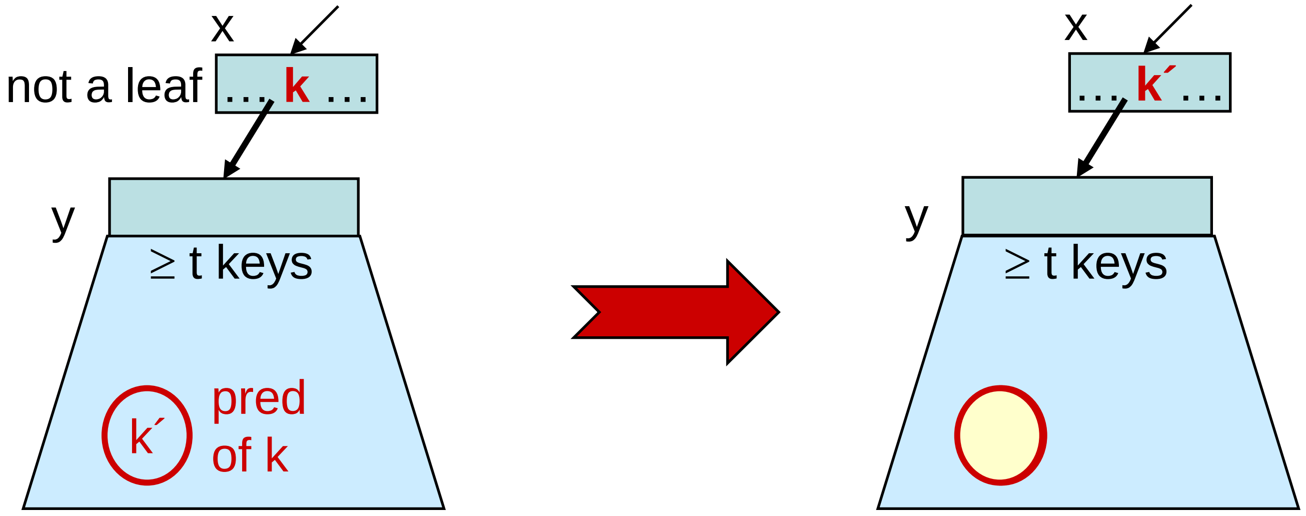
Deletion Cases (Continued)

- **Case 2:** If the child key k is in node x and x is an internal node, do the following:



Deletion Cases (Continued)

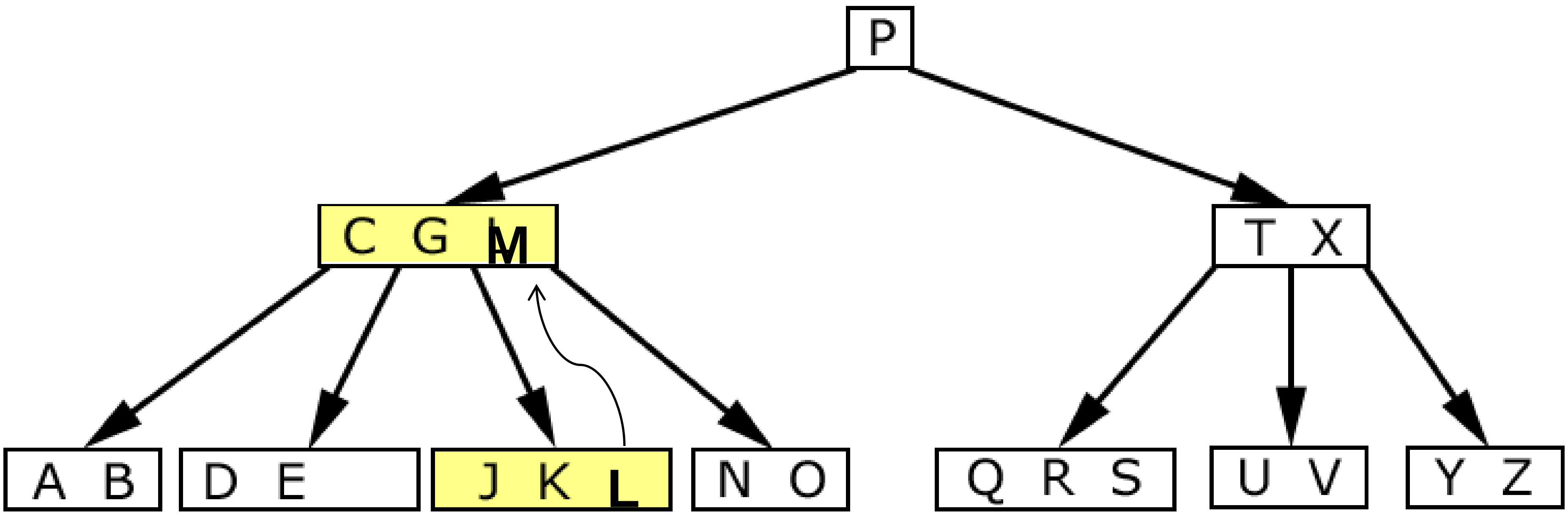
- **Subcase a:** If the child y that precedes k has at least t keys then find predecessor k´ of k in subtree rooted at y, recursively delete k´ and replace k by k´ in x.



Deleting and shifting

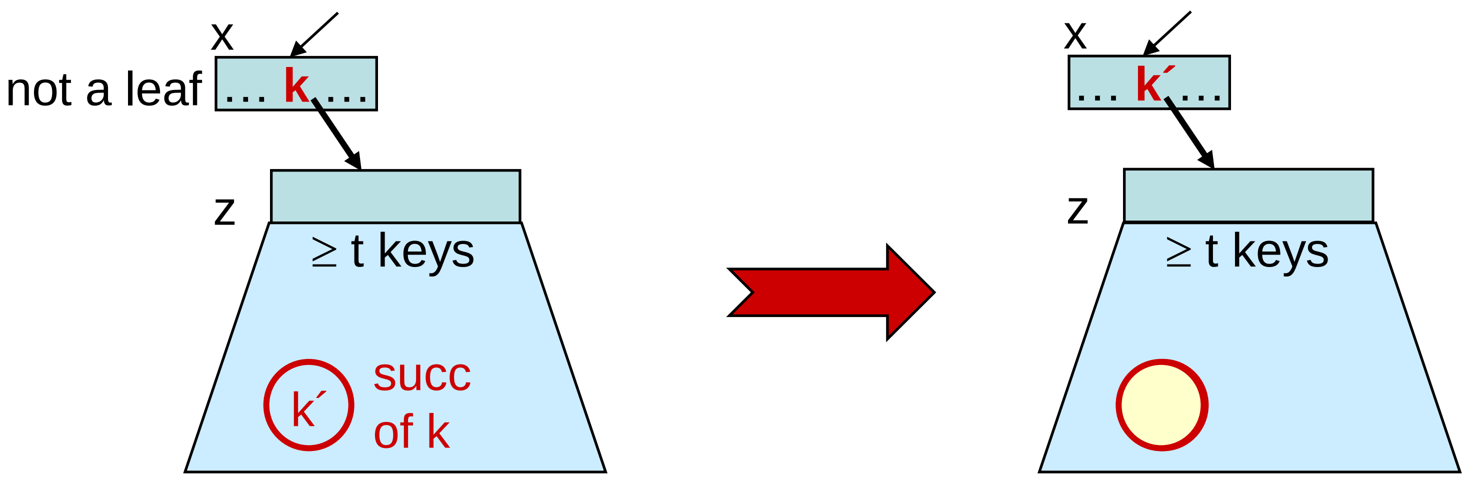
Case 2a: We deleted “M”

Result: We remove “M” from the parent node. Since there are four nodes and two letters, we move “L” to replace “M”. Now, the “N O” node has a parent again.



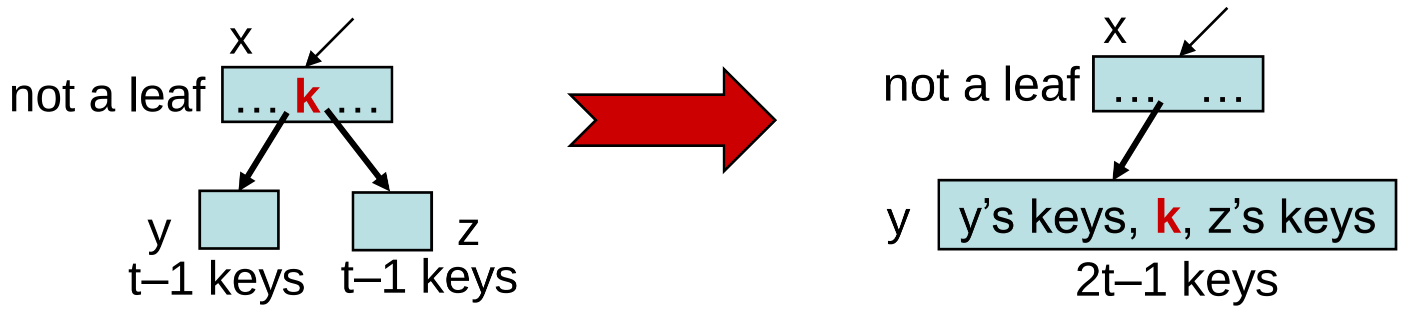
Deletion Cases (Continued)

Subcase B: Symmetrically, if the child z that follows k in node x has at least t keys then find successor k´ of k in subtree rooted at z, recursively delete k´and replace k by k´ in x.



Deletion Cases (Continued)

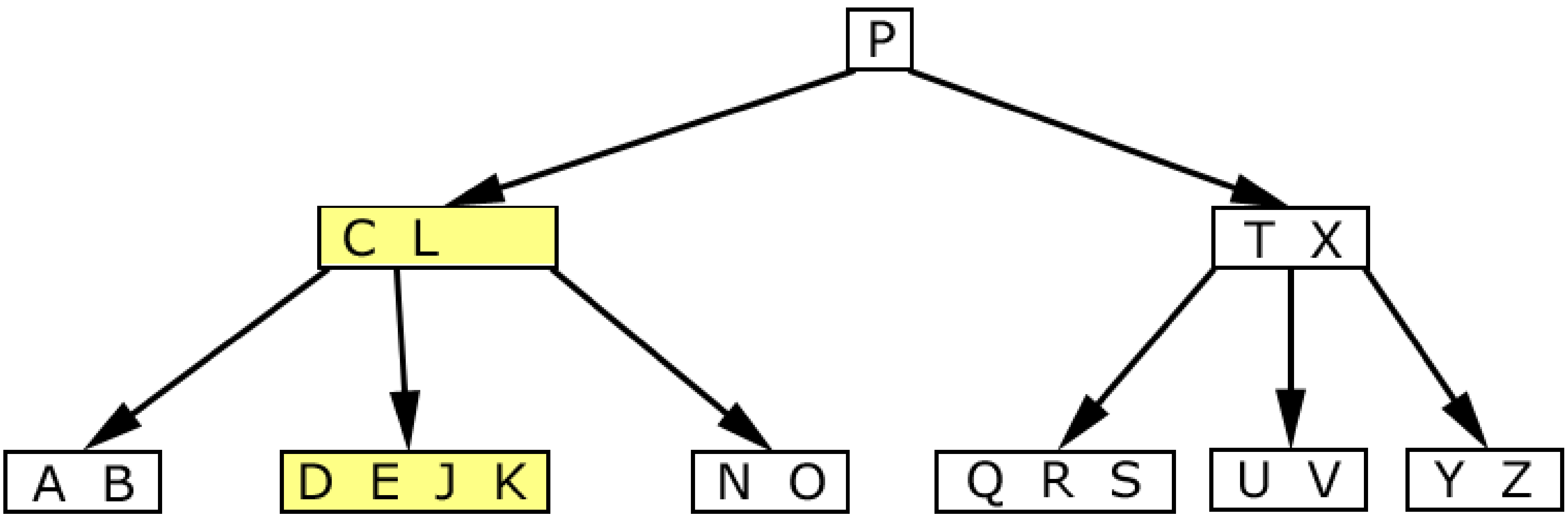
Subcase C: y and z both have t–1 keys -- merge k and z into y, free z, recursively delete k from y.



Combining and Deleting

Case 2c: Now, we delete “G”

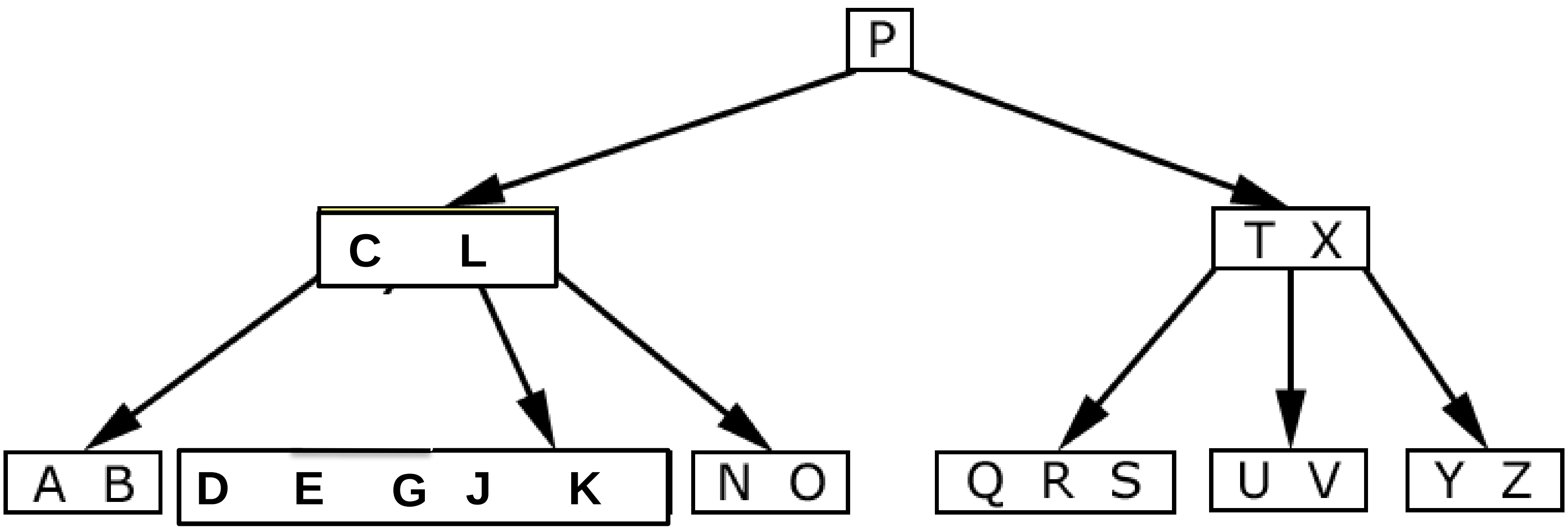
Result: First, we combine nodes “DE” and “JK”. Then, we push down “G” into the “DEJK” node and delete it as a leaf.



Combining and Deleting

Case 2c: Now, we delete “G”

Result: First, we combine nodes “DE” and “JK”. Then, we push down “G” into the “DEJK” node and delete it as a leaf.

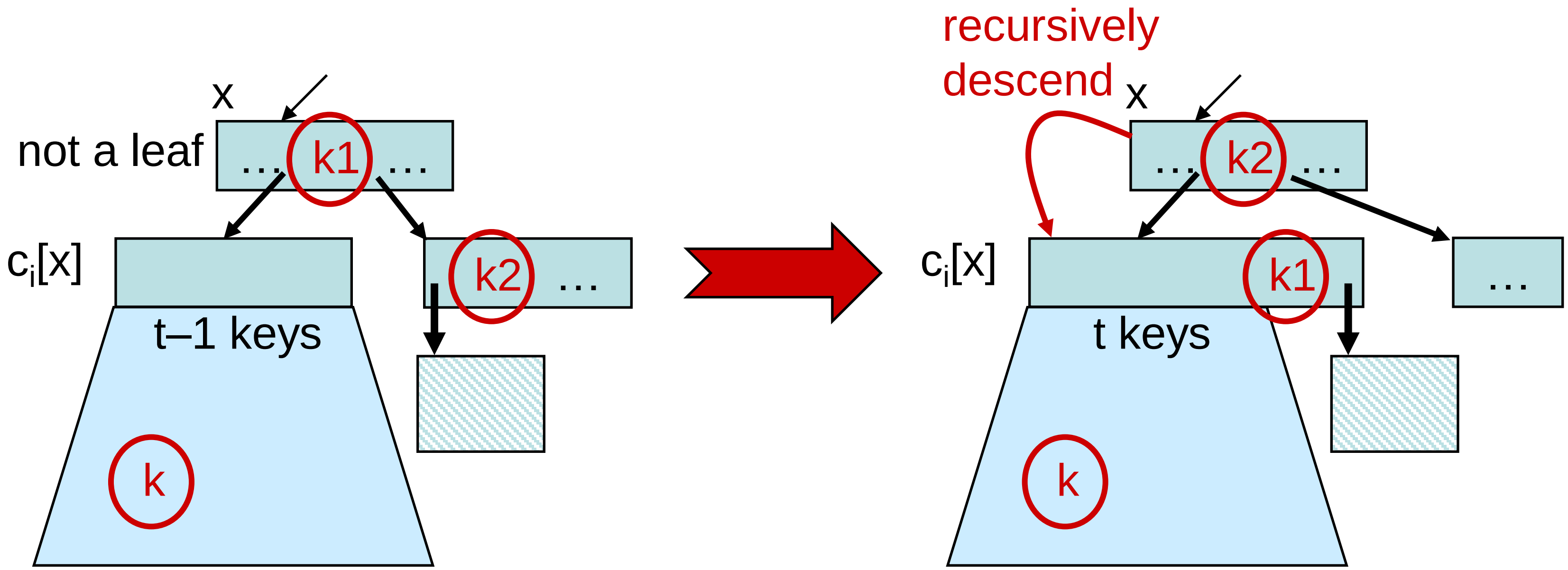


Deletion Cases (Continued)

Case 3: k not in internal node. Let $c_i[x]$ be the root of the subtree that must contain k, if k is in the tree.
If $c_i[x]$ has at least t keys, then recursively descend; otherwise, execute 3.A and 3.B as necessary.

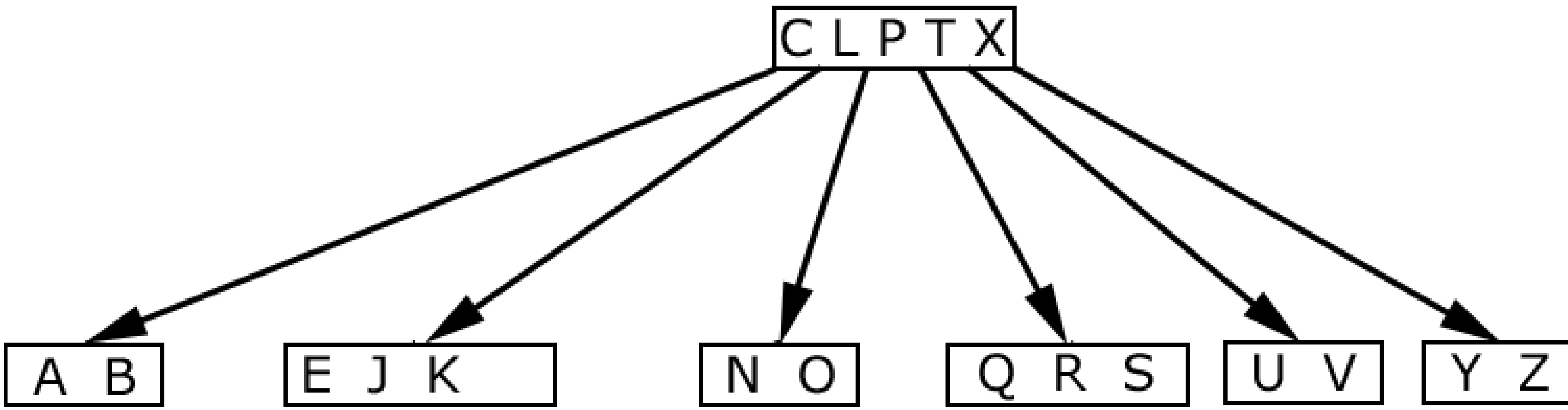
Deletion Cases (Continued)

Subcase A: $c_i[x]$ has t–1 keys, some sibling has at least t keys.

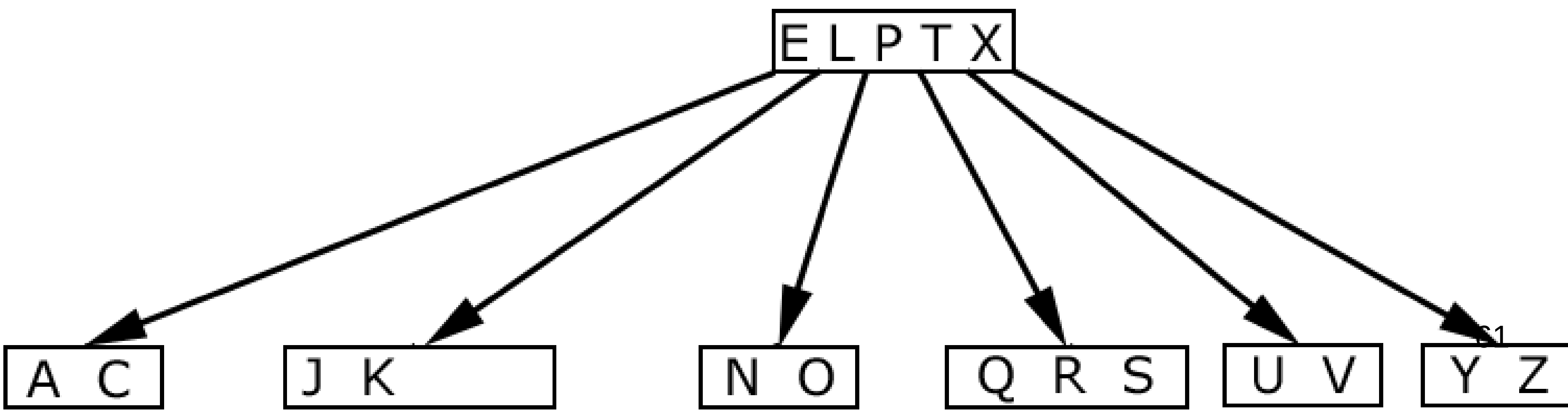


Deleting “B”

Before:

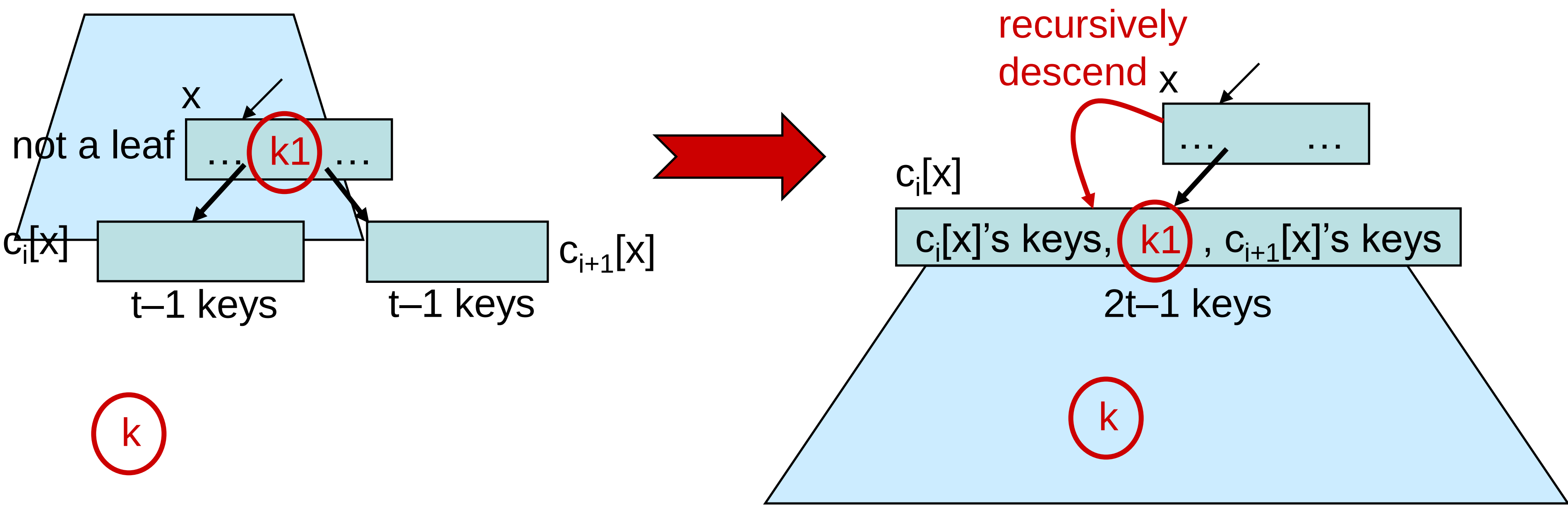


After: Deleted “B”, Demoted “C”, Promoted “E”



Deletion Cases (Continued)

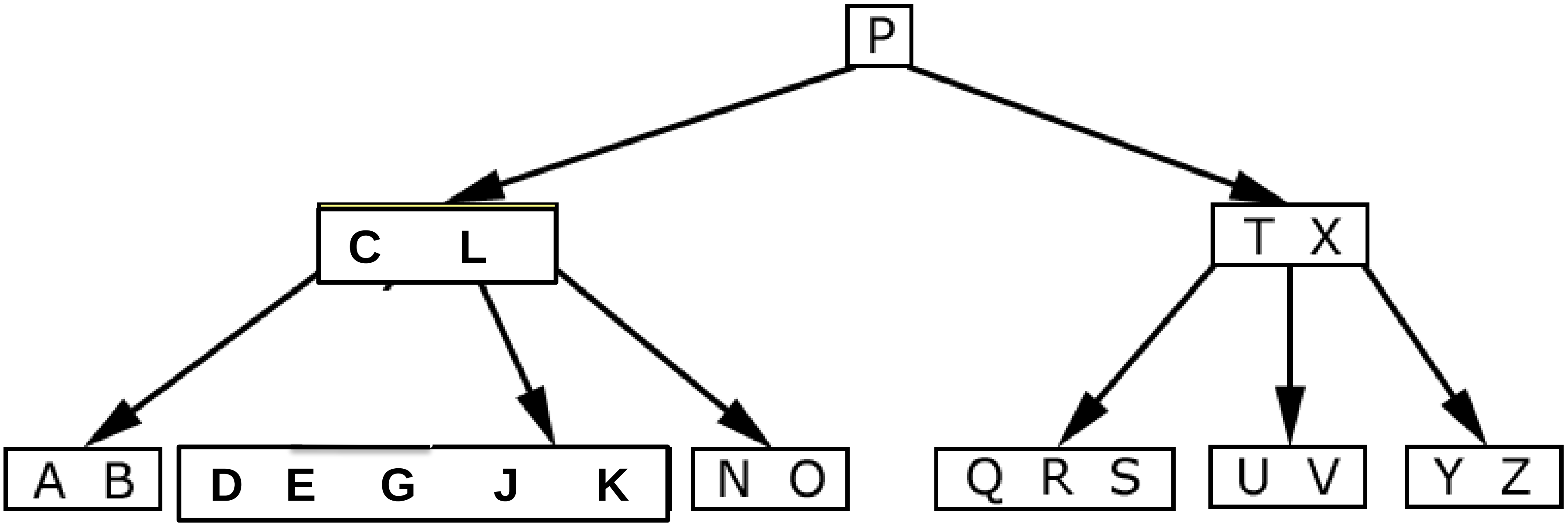
Subcase B: $c_i[x]$ and sibling both have t–1 keys.



Combining and Deleting

Case 3b: Now, we delete “D”

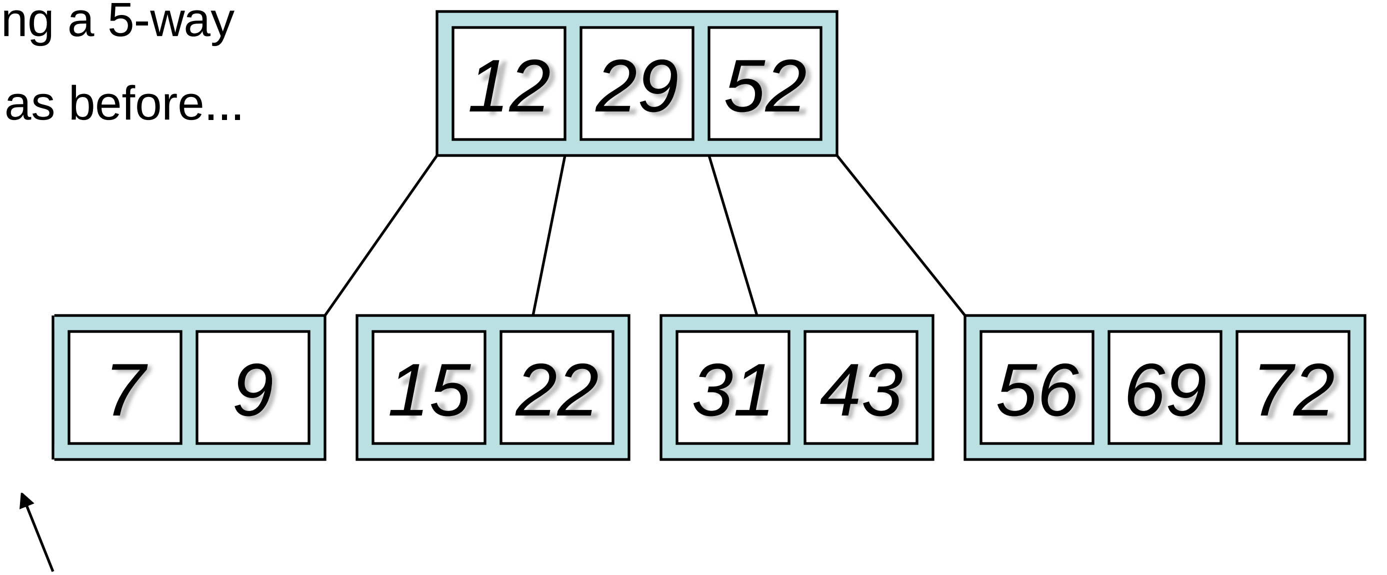
Result: First, we combine nodes “DE” and “JK”. Then, we push down “G” into the “DEJK” node and delete “D” as a leaf.



Type #1: Simple leaf deletion

Assuming a 5-way

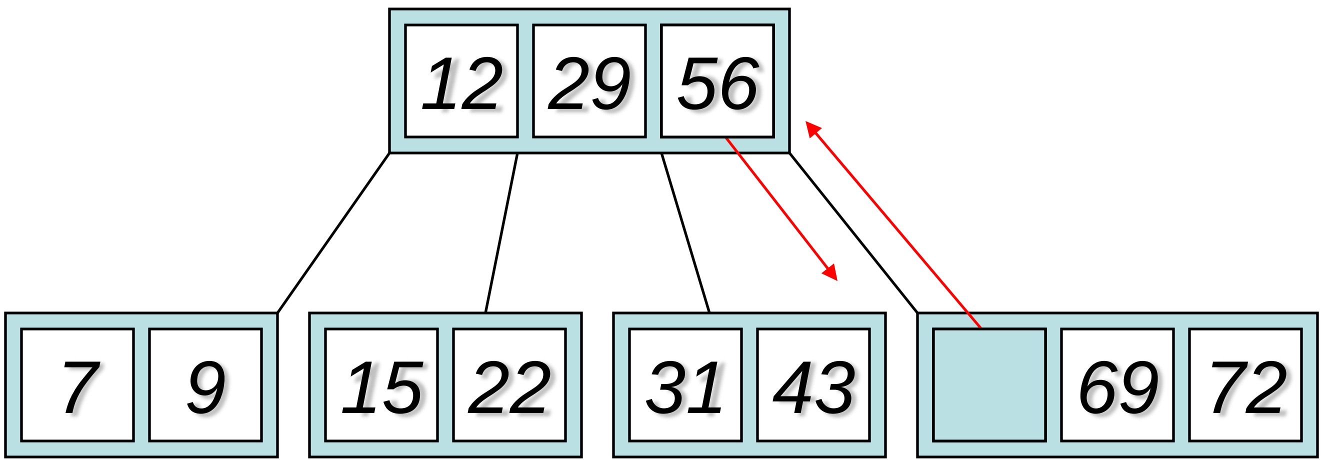
B-Tree, as before...



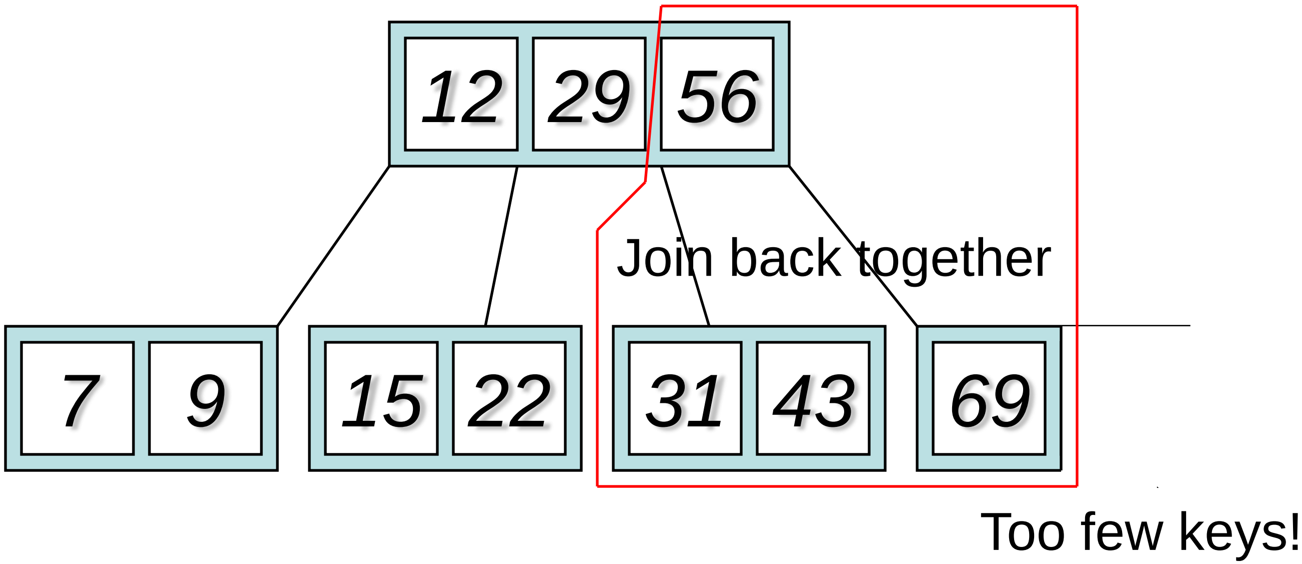
Delete 2: Since there are enough

keys in the node, just delete it

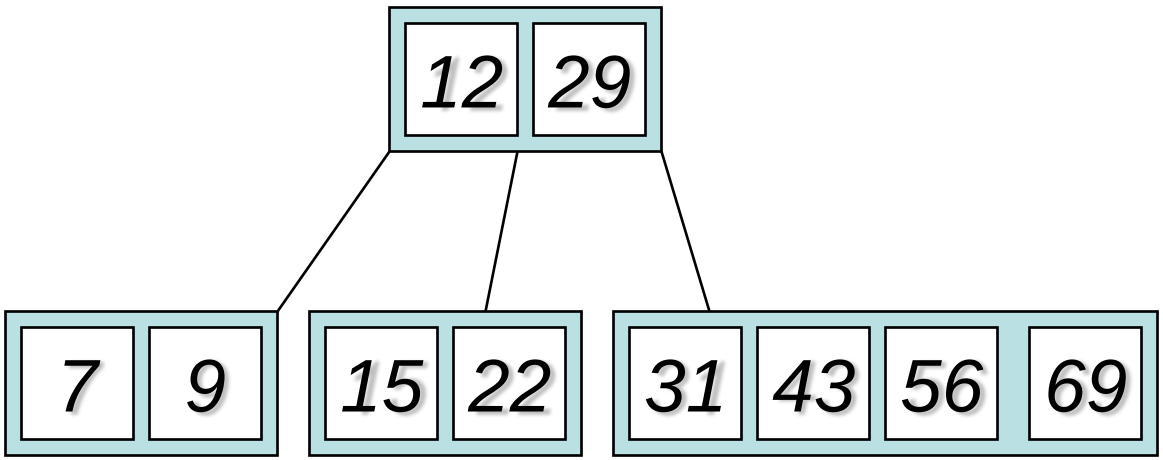
Type #2: Simple non-leaf deletion



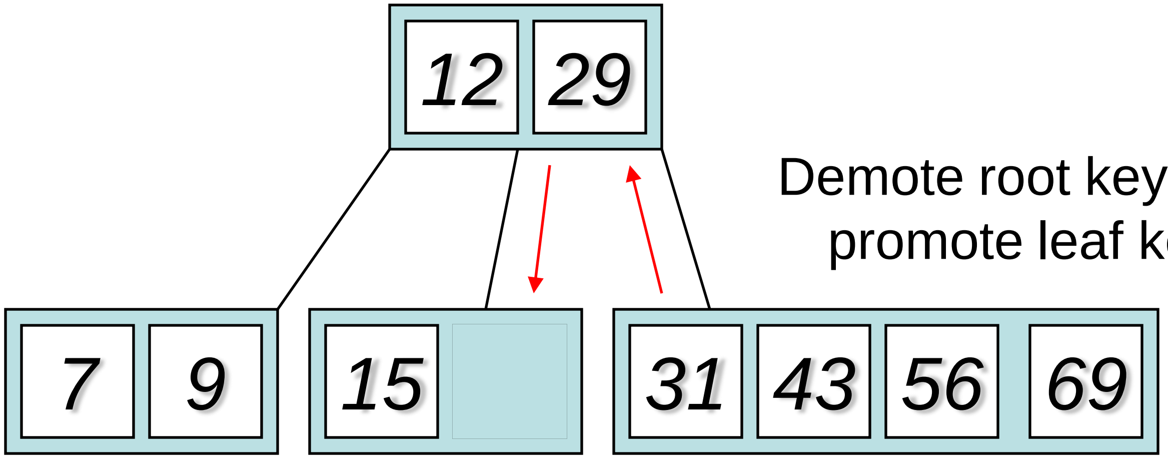
Type #4: Too few keys in node and its siblings



Type #4: Too few keys in node and its siblings



Type #3: Enough siblings



Demote root key and promote leaf key

Type #3: Enough siblings