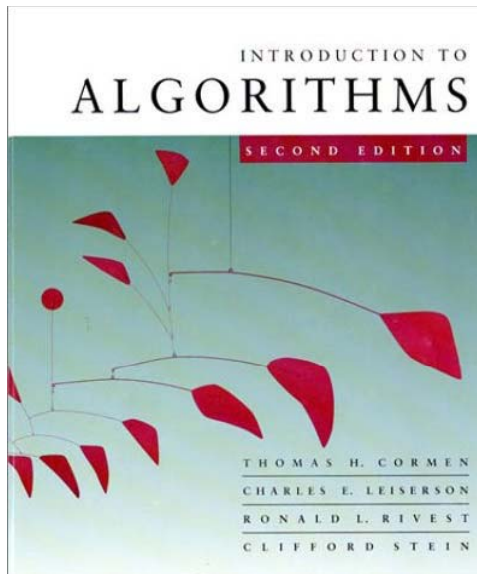
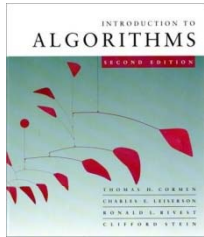


CS 3343 -- Fall 2010



B-trees

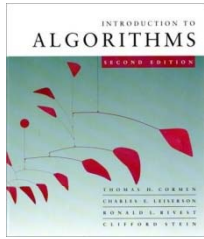
Carola Wenk



External memory dictionary

Task: Given a large amount of data that does not fit into main memory, process it into a dictionary data structure

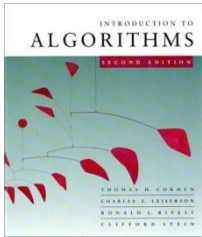
- Need to minimize number of disk accesses
- With each disk read, read a whole block of data
- Construct a balanced search tree that uses one disk block per tree node
- Each node needs to contain more than one key



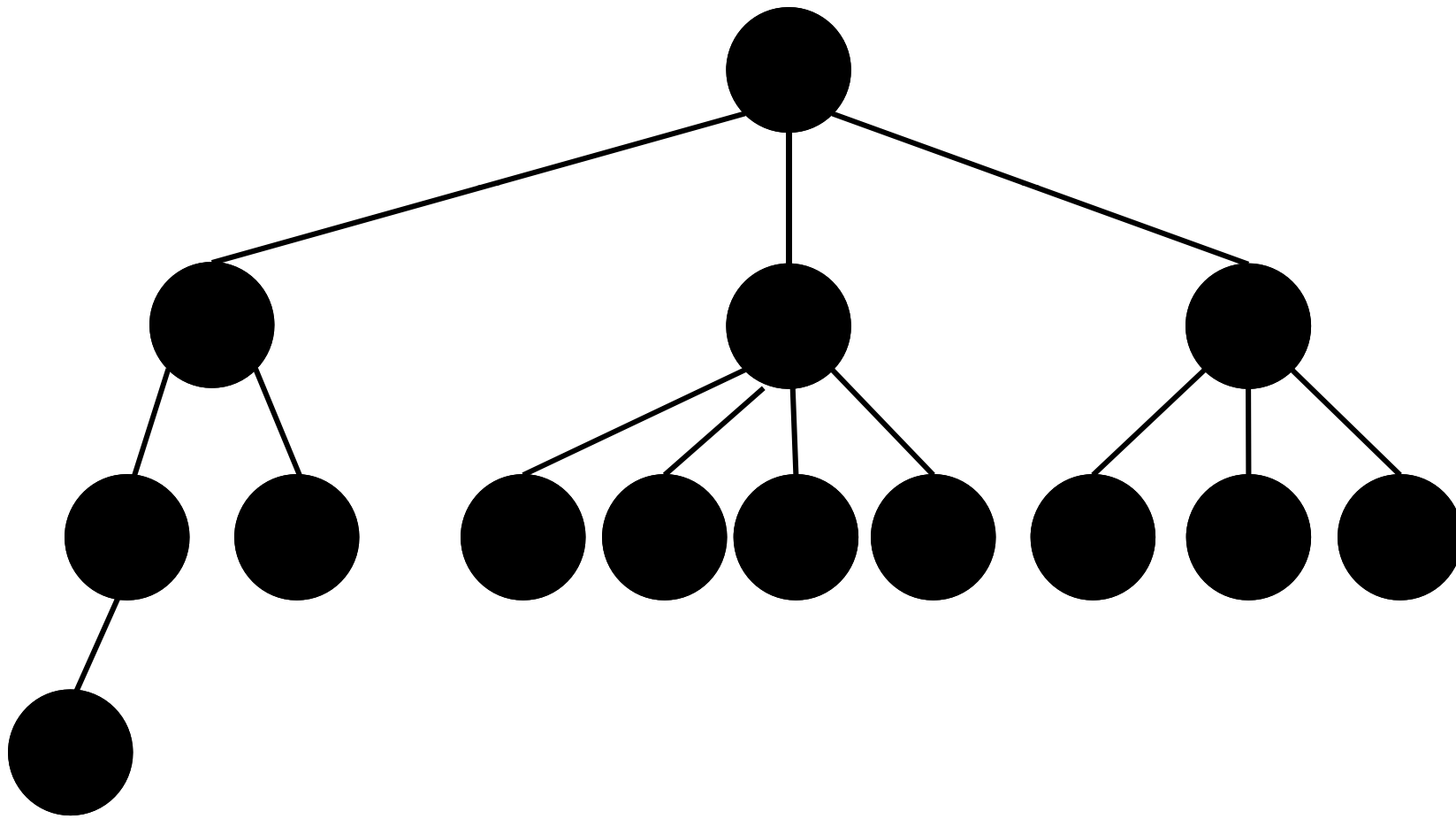
k -ary search trees

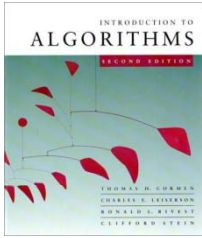
A k -ary search tree T is defined as follows:

- For each node x of T :
 - x has at most k children (i.e., T is a k -ary tree)
 - x stores an ordered list of pointers to its children, and an ordered list of keys
 - For every internal node: $\#keys = \#children - 1$
 - x fulfills the **search tree property**:
keys in subtree rooted at i -th child $\leq i$ -th key $<$
keys in subtree rooted at $(i+1)$ -st child

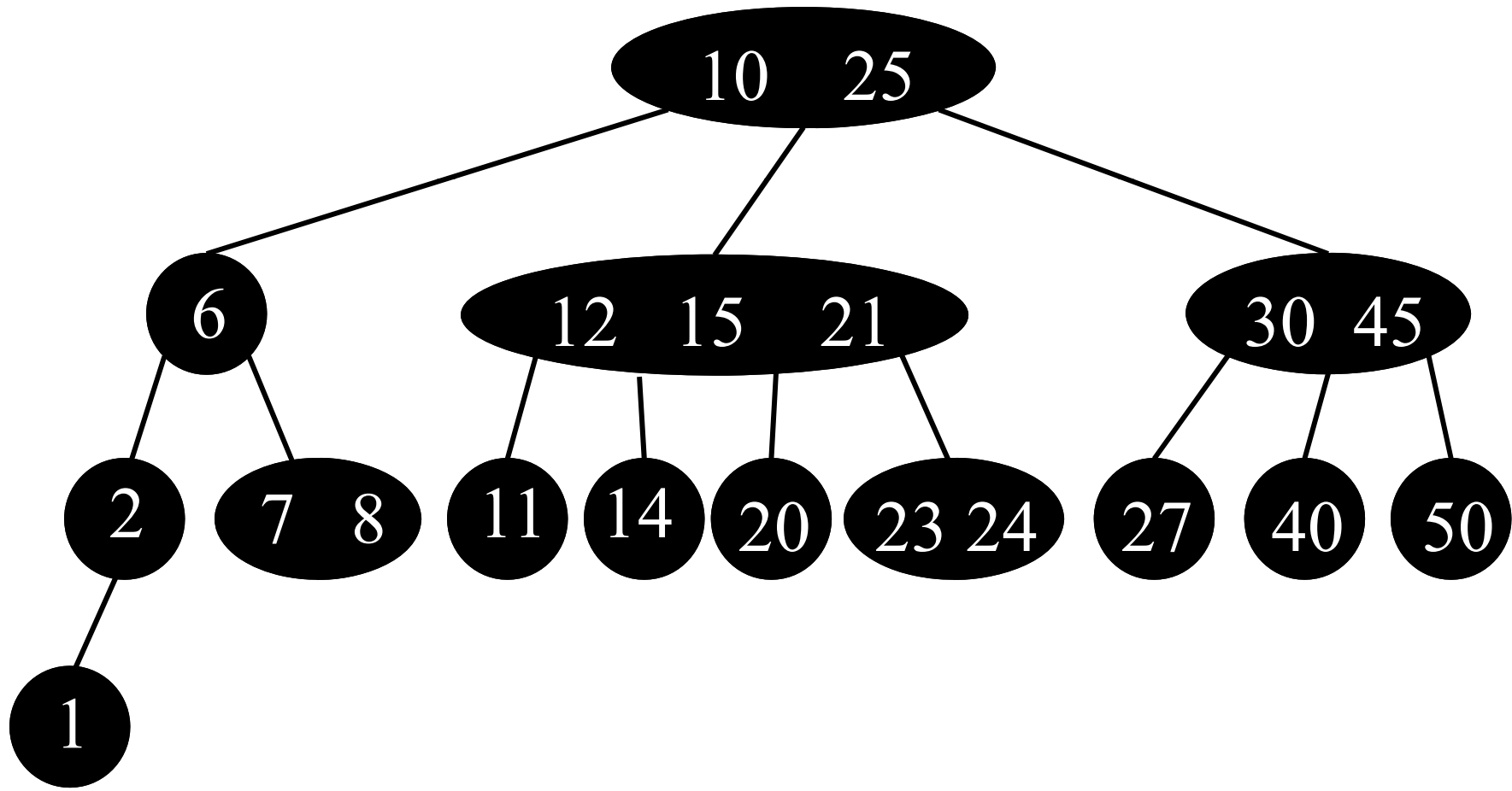


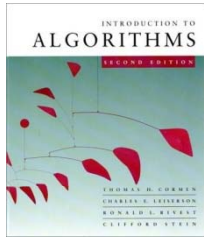
Example of a 4-ary tree





Example of a 4-ary search tree

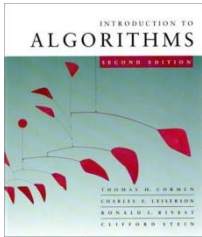




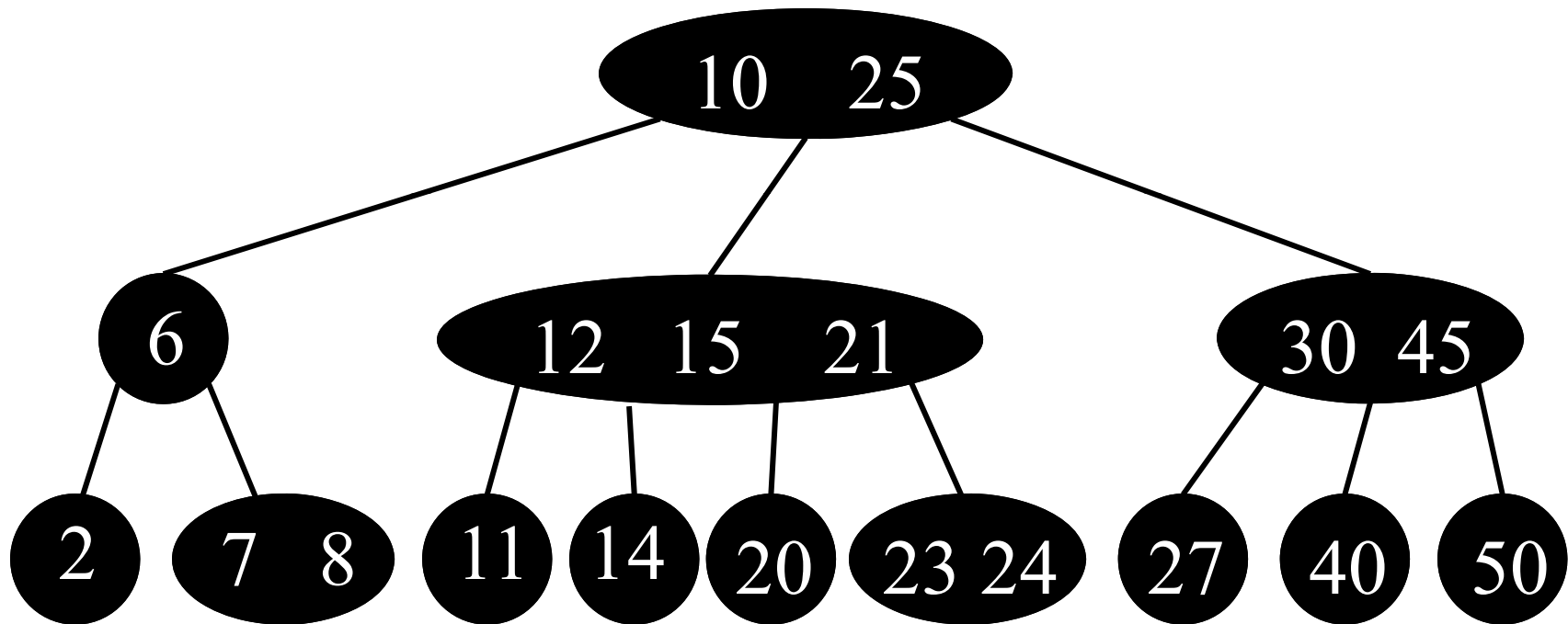
B-tree

A **B-tree** T with **minimum degree** $k \geq 2$ is defined as follows:

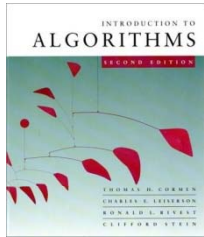
1. T is a $(2k)$ -ary search tree
2. Every node, except the root, stores at least $k-1$ keys
(every internal non-root node has at least k children)
3. The root must store at least one key
4. All leaves have the same depth



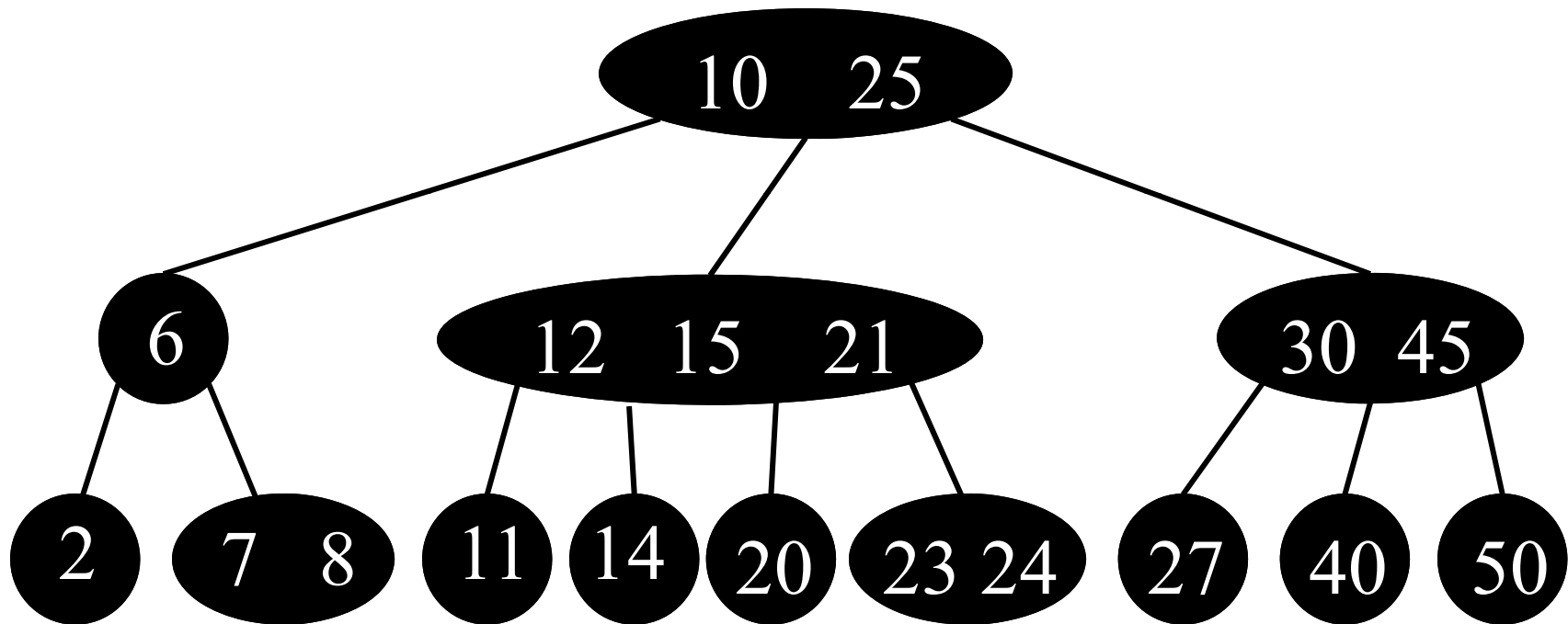
B-tree with $k=2$



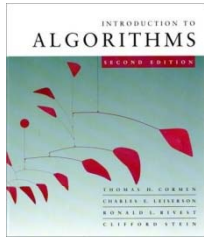
1. T is a $(2k)$ -ary search tree



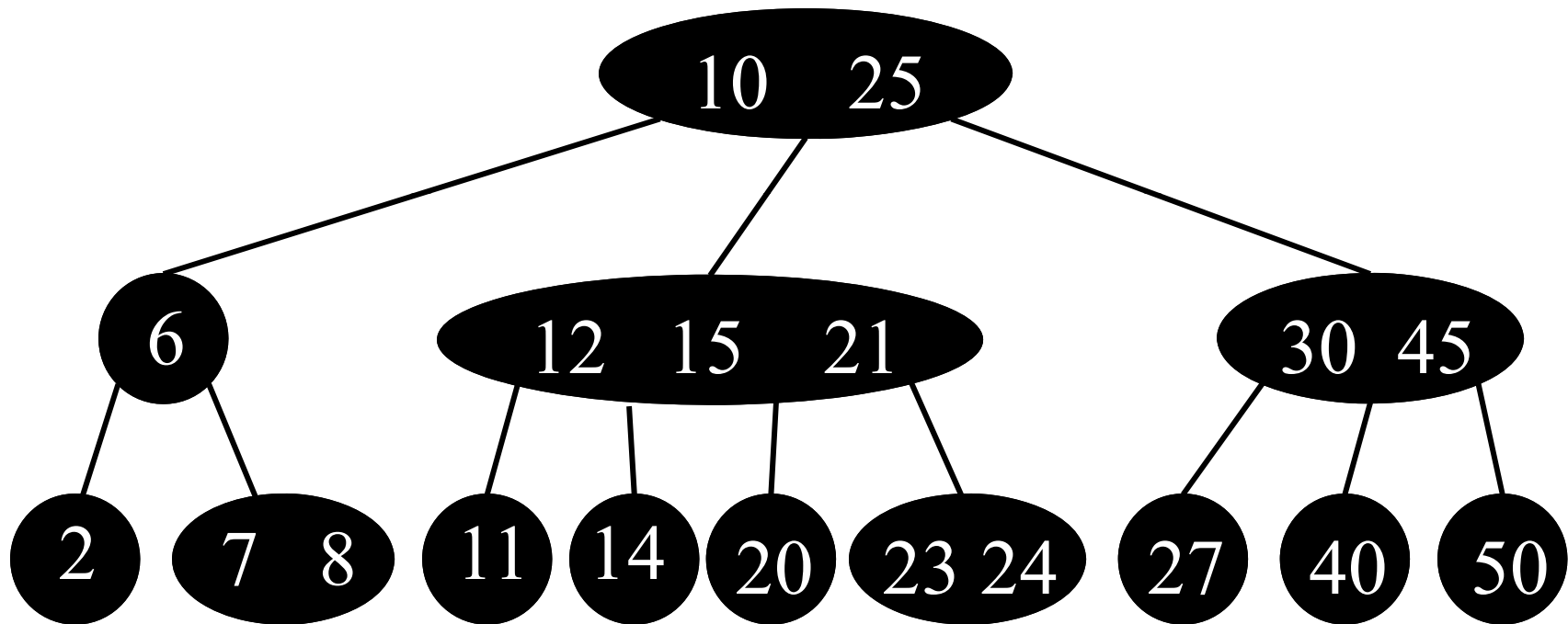
B-tree with $k=2$



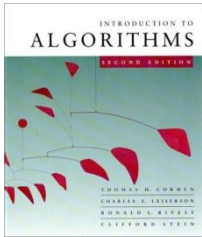
2. Every node, except the root, stores at least $k-1$ keys



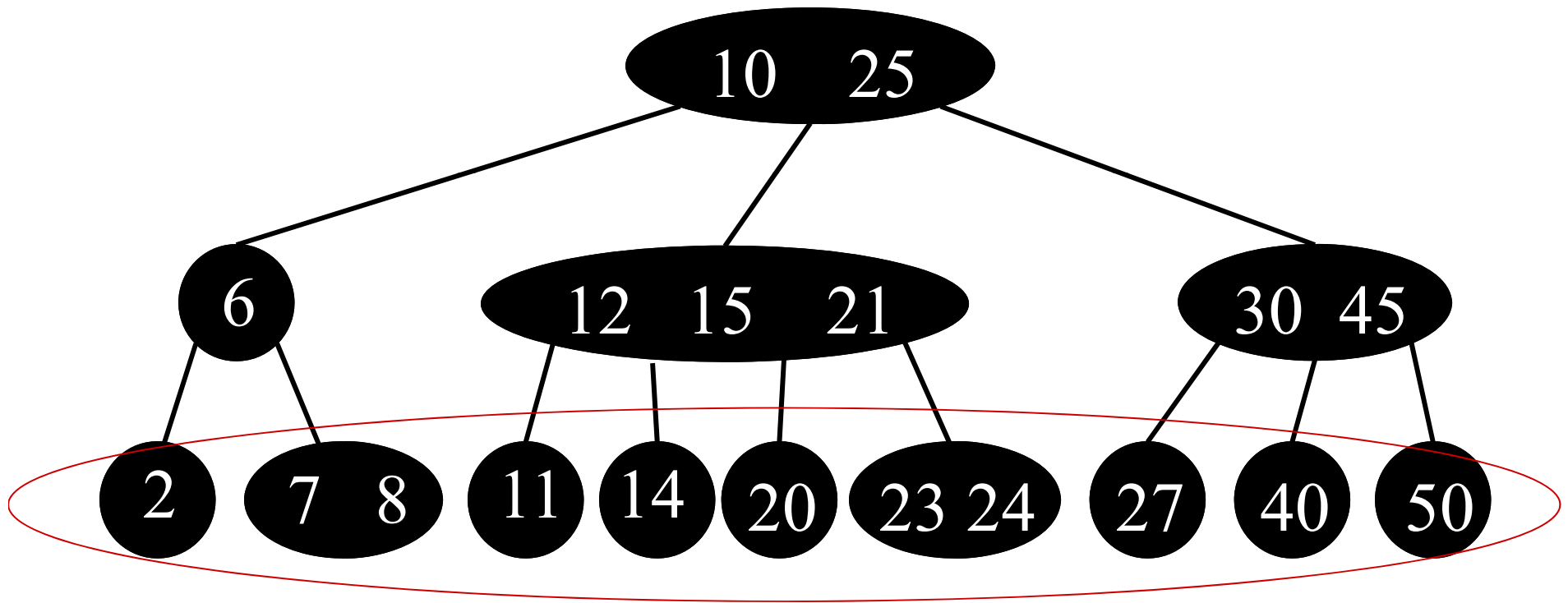
B-tree with $k=2$



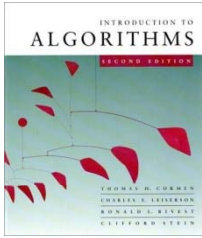
3. The root must store at least one key



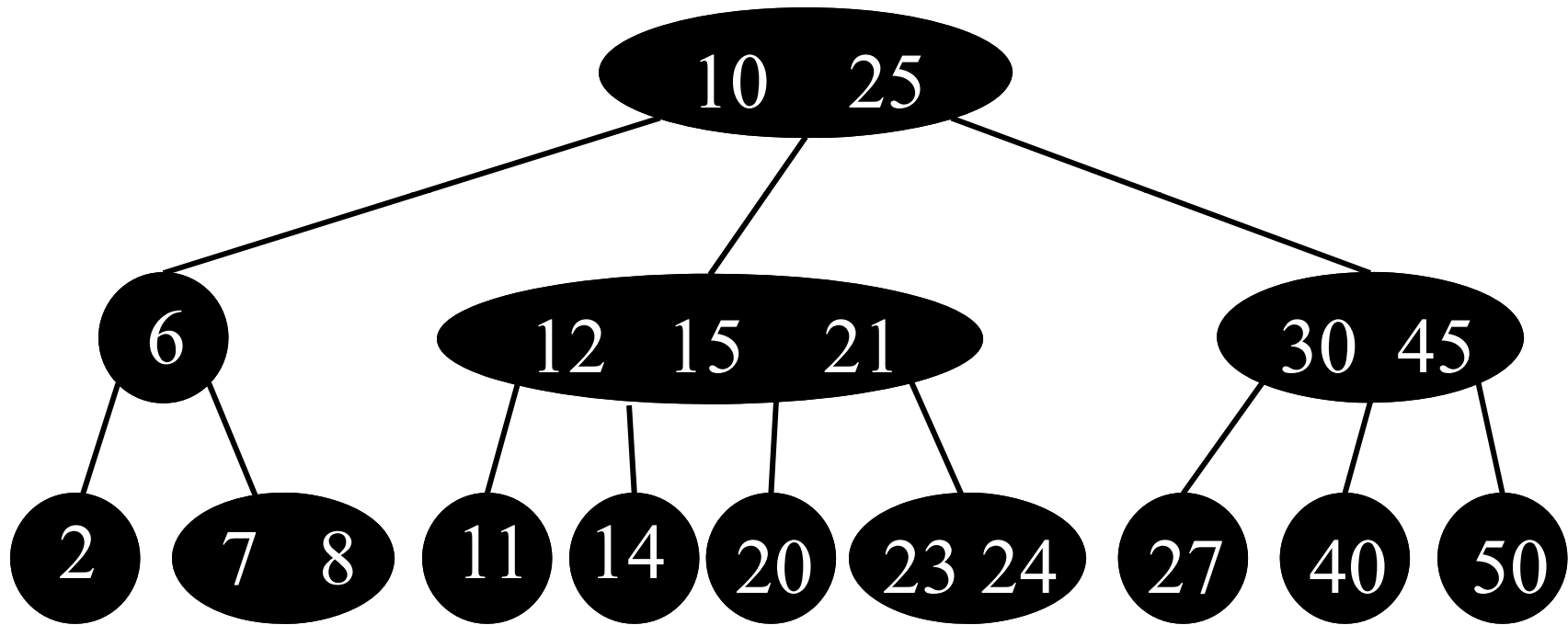
B-tree with $k=2$



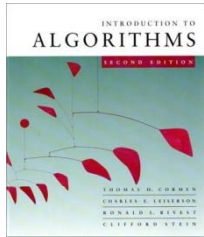
4. All leaves have the same depth



B-tree with $k=2$



Remark: This is a $(2,3,4)$ -tree.

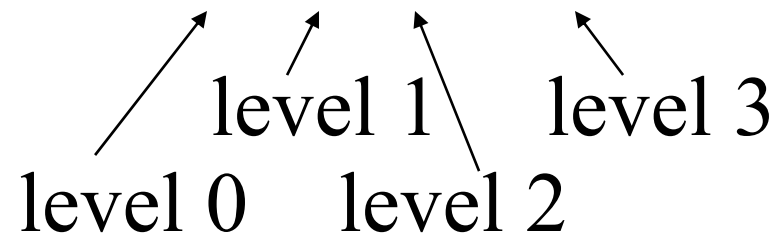


Height of a B-tree

Theorem: For a B-tree with minimum degree $k \geq 2$ which stores n keys has height h holds:

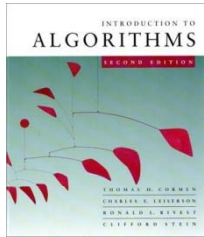
$$h \leq \log_k (n+1)/2$$

Proof: #nodes $\geq 1 + 2 + 2k + 2k^2 + \dots + 2k^{h-1}$



$$n = \#keys \geq 1 + (k-1) \sum_{i=0}^{h-1} 2k^i = 1 + 2(k-1) \cdot \frac{k^h - 1}{k-1} = 2k^h - 1$$





B-tree search

B-TREE-SEARCH(x, key)

$i \leftarrow 1$

while $i \leq \#keys$ of x **and** $key > i$ -th key of x | Linear
do $i \leftarrow i+1$ | Search

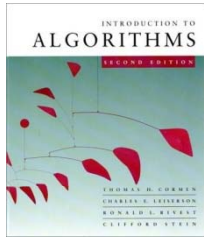
if $i \leq \#keys$ of x **and** $key = i$ -th key of x
then return (x, i)

if x is a leaf

then return **NIL**

else $b = \text{DISK-READ}(i\text{-th child of } x)$

return **B-TREE-SEARCH**(b, key)

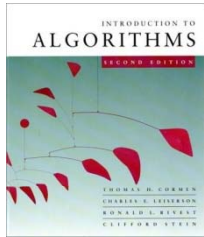


B-tree search runtime

- $O(k)$ per node
- Path has height $h = O(\log_k n)$
- CPU-time: $O(k \log_k n)$

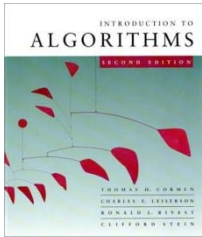
- Disk accesses: $O(\log_k n)$

disk accesses are more expensive than CPU time



B-tree insert

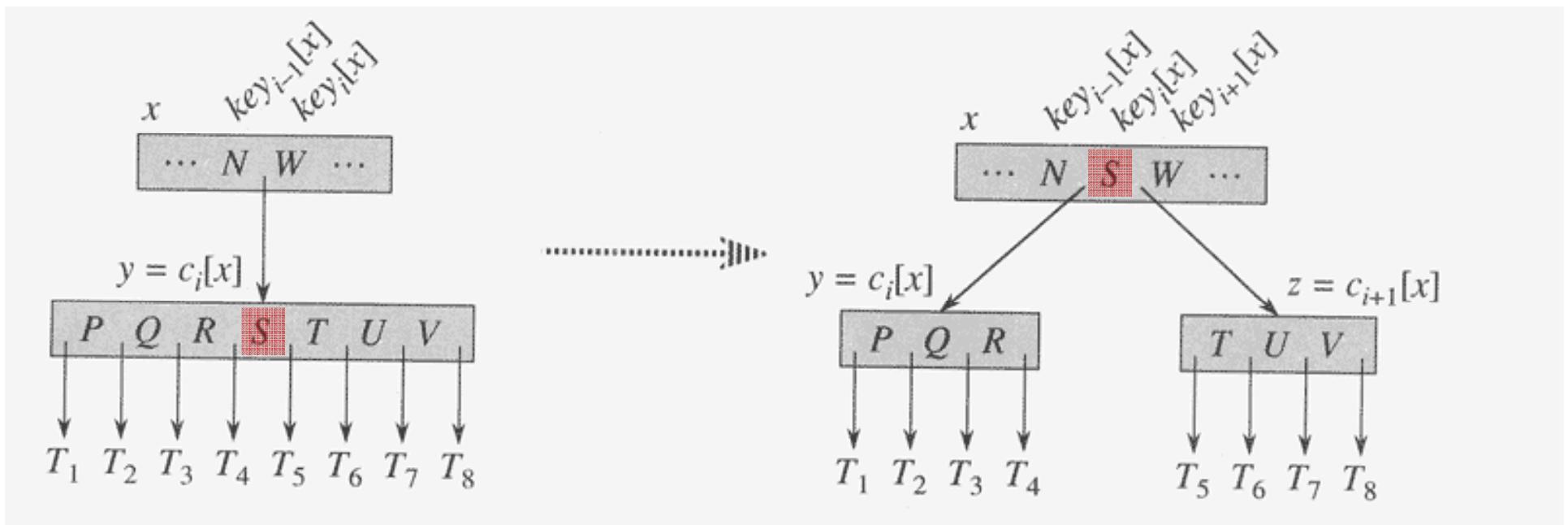
- There are different insertion strategies. We just cover one of them
- Make one pass down the tree:
 - The goal is to insert the new *key* into a leaf
 - Search where *key* should be inserted
 - **Only descend into non-full nodes:**
 - If a node is full, split it. Then continue descending.
 - **Splitting of the root node is the only way a B-tree grows in height**

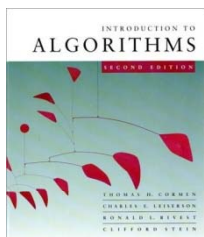


B-TREE-SPLIT-CHILD(x, i, y)

has $2k-1$ keys

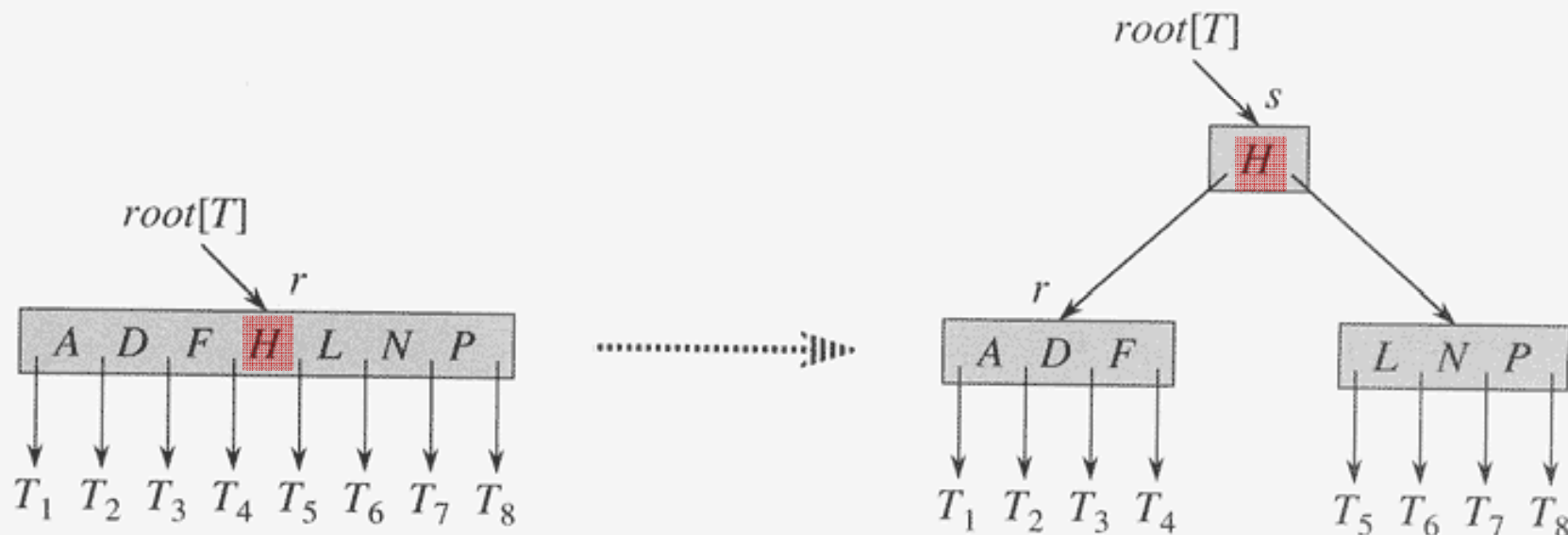
- Split full node y into two nodes y and z of $k-1$ keys
- Median key S of y is moved up into y 's parent x
- Example below for $k = 4$

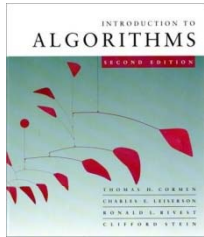




Split root: B-TREE-SPLIT-CHILD(s, l, r)

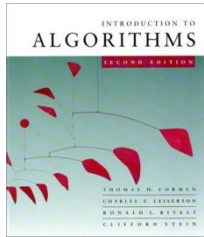
- The **full** root node r is split in two.
- A new root node s is created
- s contains the median key H of r and has the two halves of r as children
- Example below for $k = 4$





B-TREE-INSERT(T, key)

```
 $r = \text{root}[T]$   
if (# keys in  $r$ ) =  $2k-1$  // root  $r$  is full  
    //insert new root node:  
     $s \leftarrow \text{ALLOCATE-NODE}()$   
     $\text{root}[T] \leftarrow s$   
    // split old root  $r$  to be two children of new root  $s$   
    B-TREE-SPLIT-CHILD( $s, 1, r$ )  
    B-TREE-INSERT-NONFULL( $s, key$ )  
else B-TREE-INSERT-NONFULL( $r, key$ )
```



B-TREE-INSERT-NONFULL(x, key)

if x is a leaf **then**

insert key at the correct (sorted) position in x

DISK-WRITE(x)

else

find child c of x which by the search tree property
should contain key

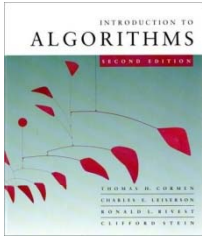
DISK-READ(c)

if c is full **then** // c contains $2k-1$ keys

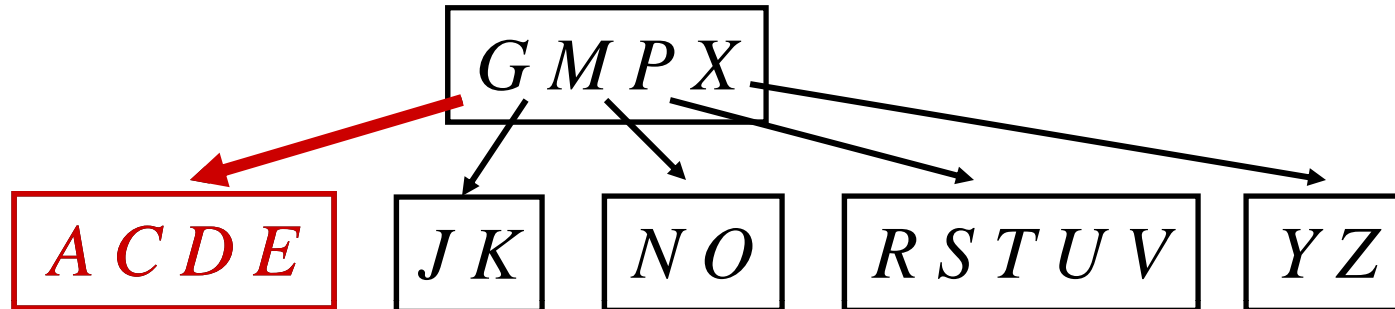
B-TREE-SPLIT-CHILD(x, i, c)

c =child of x which should contain key

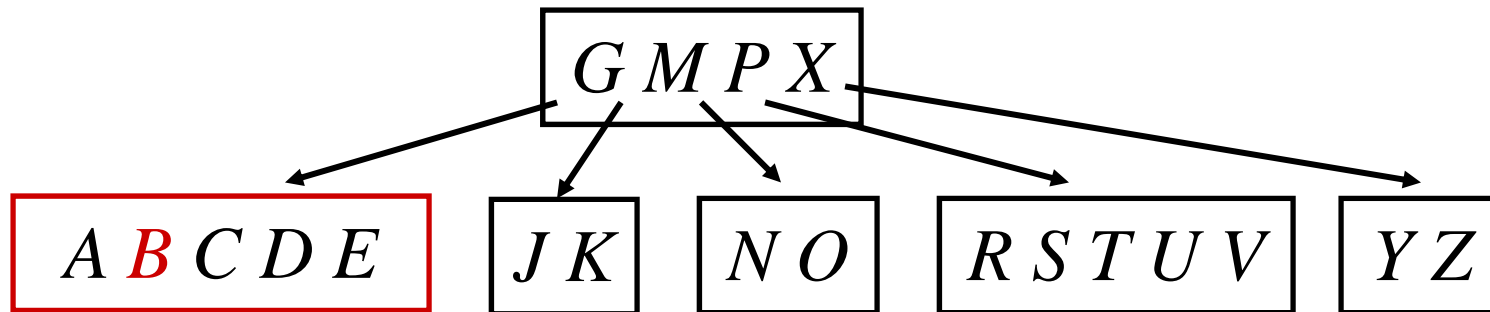
B-TREE-INSERT-NONFULL(c, key)

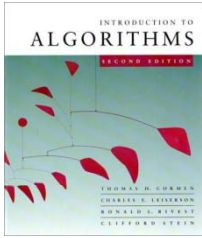


Insert example ($k=3$)

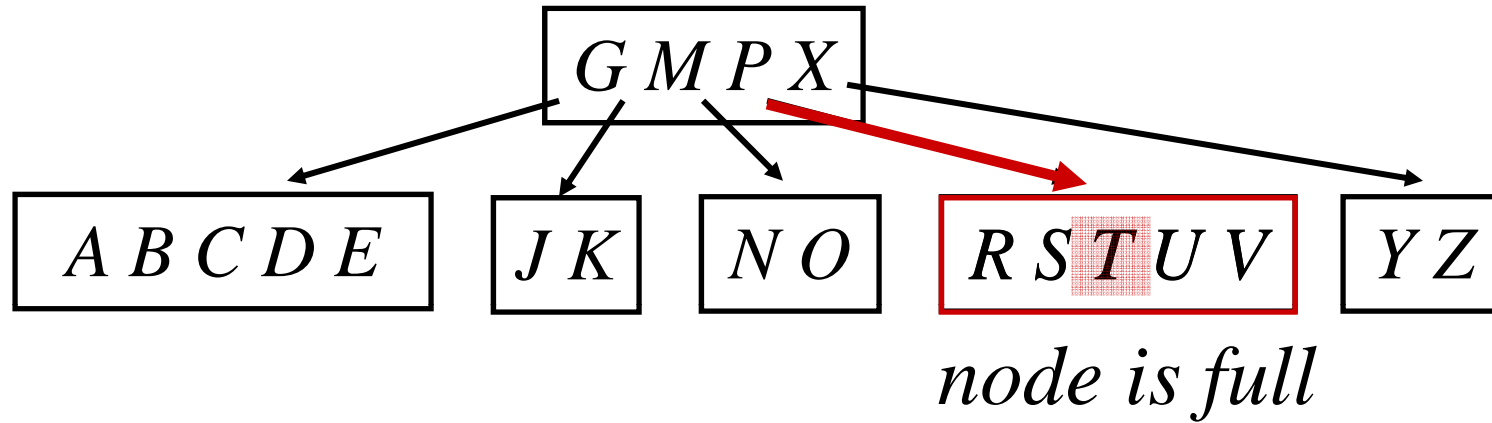


- Insert B :

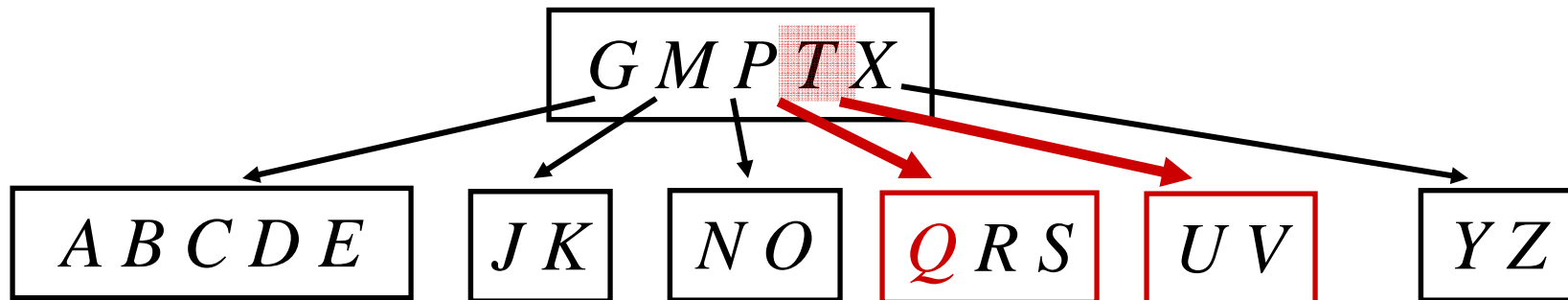


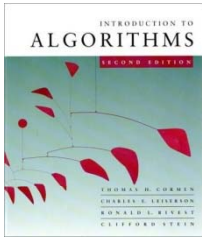


Insert example ($k=3$) -- cont.

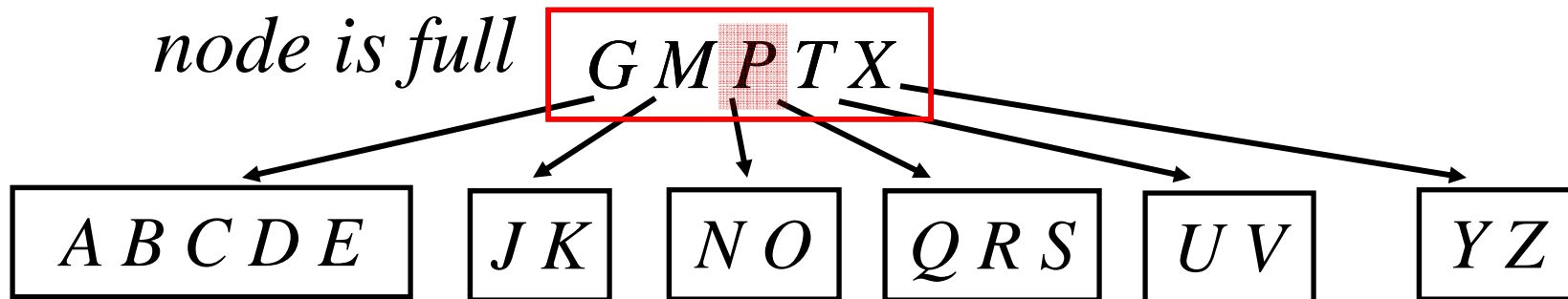


- Insert Q :

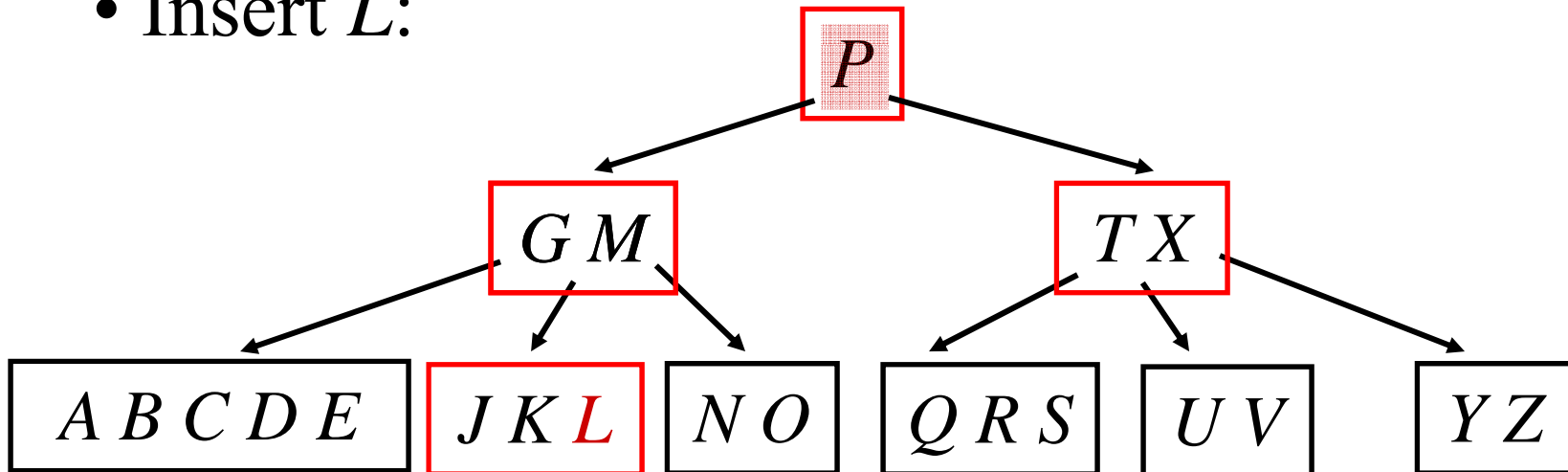


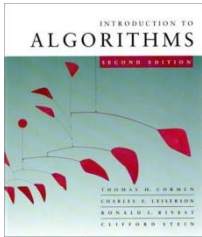


Insert example ($k=3$) -- cont.

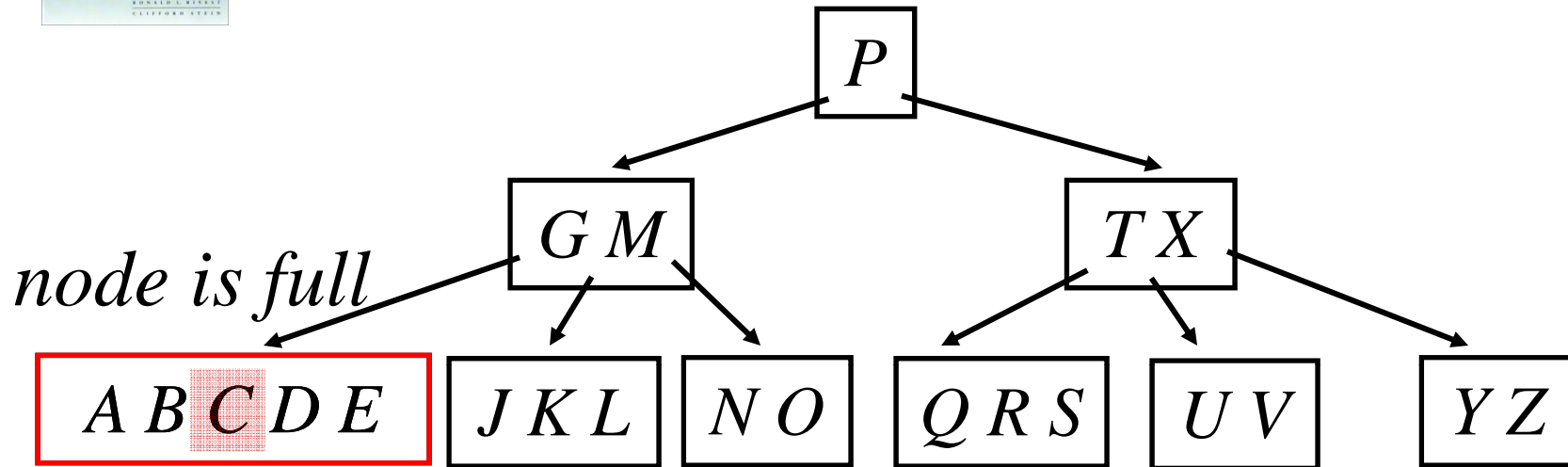


- Insert L :

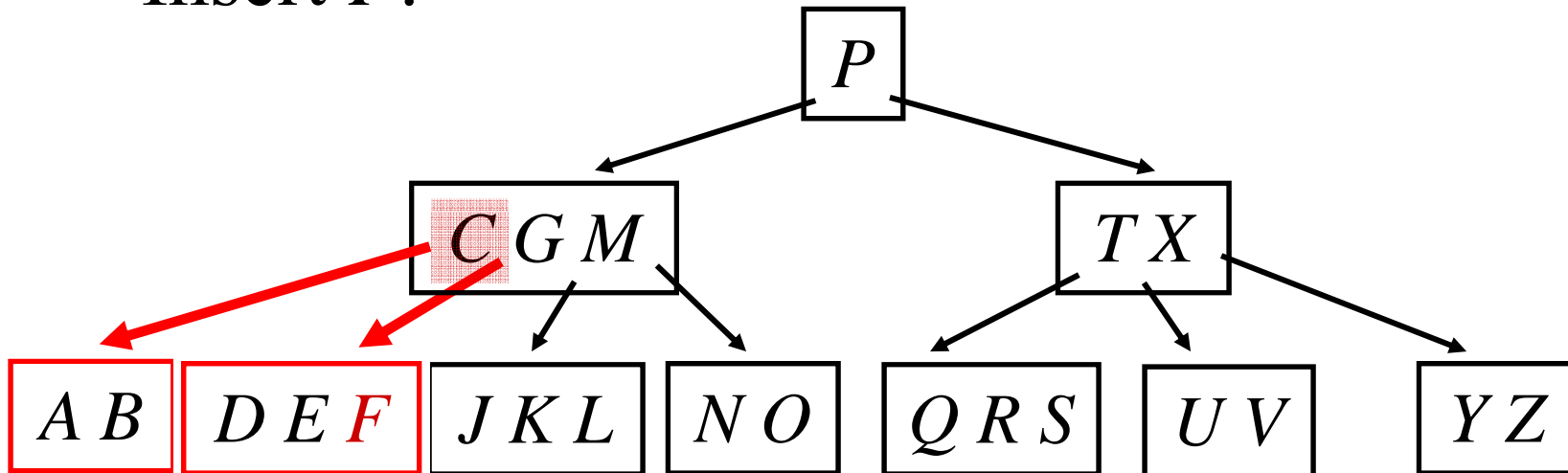


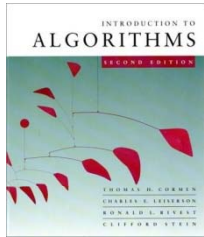


Insert example ($k=3$) -- cont.



- Insert F :



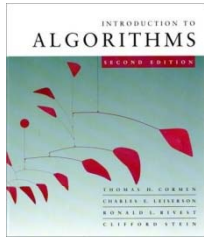


Runtime of B-TREE-INSERT

- $O(k)$ runtime per node
- Path has height $h = O(\log_k n)$
- CPU-time: $O(k \log_k n)$

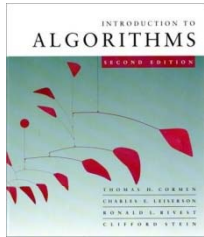
- Disk accesses: $O(\log_k n)$

disk accesses are more expensive than CPU time



Deletion of an element

- Similar to insertion, but a bit more complicated; see book for details
- If sibling nodes get not full enough, they are **merged** into a single node
- Same complexity as insertion



B-trees -- Conclusion

- B-trees are balanced $2k$ -ary search trees
- The **degree** of each node is **bounded from above and below** using the parameter k
- All leaves are at the same height
- **No rotations** are needed: During insertion (or deletion) the balance is maintained by **node splitting** (or **node merging**)
- The tree grows (shrinks) in height only by **splitting** (or **merging**) the root