CMPS 2200 – Fall 2015

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

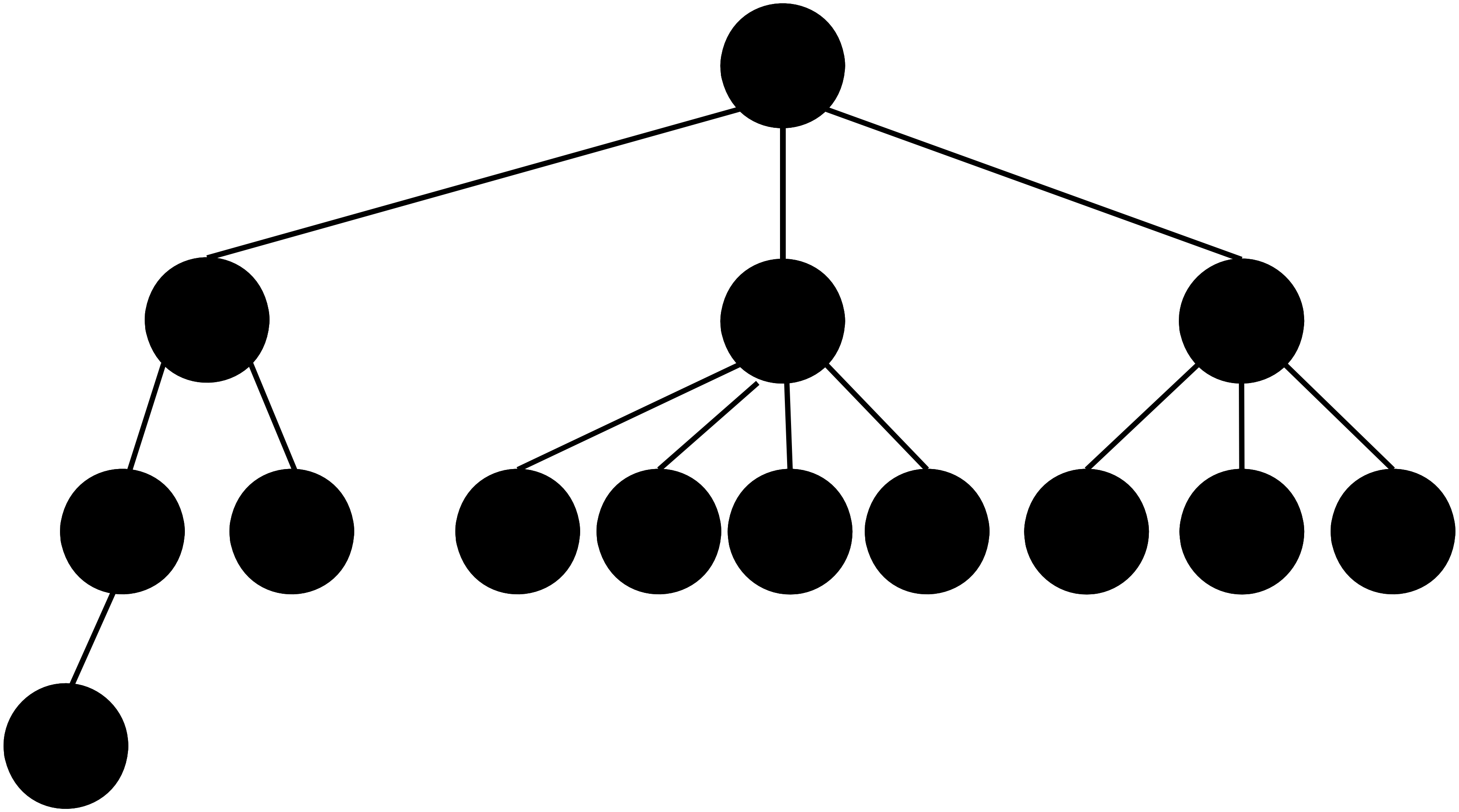
k-ary search trees

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

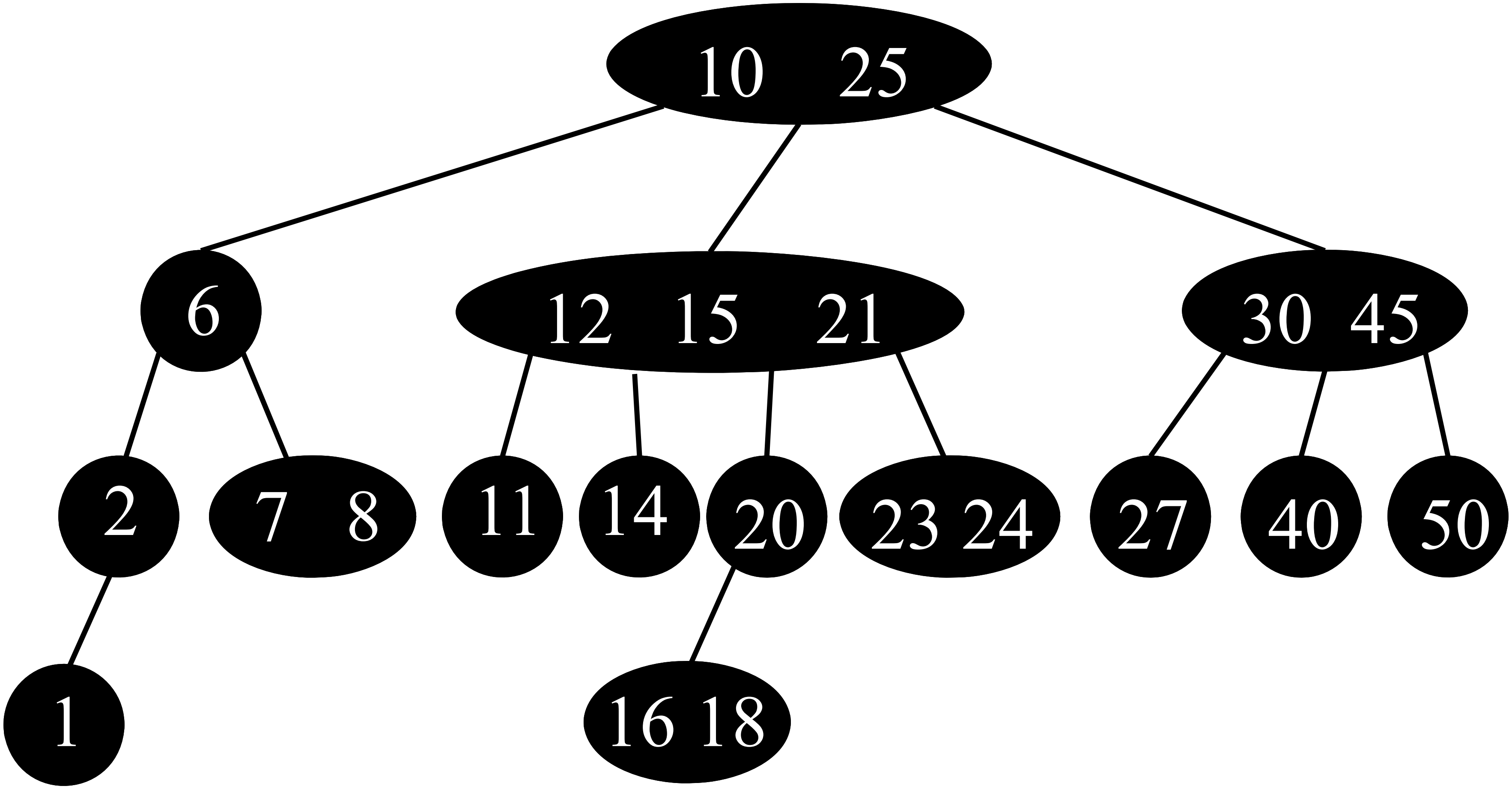
- For each node x of T:
  - x has at most k children (i.e., T is a k-ary tree)
  - x stores an ordered list of pointers to its children, and an ordered list of keys
  - For every internal node: #keys = #children-1
  - x fulfills the search tree property:

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

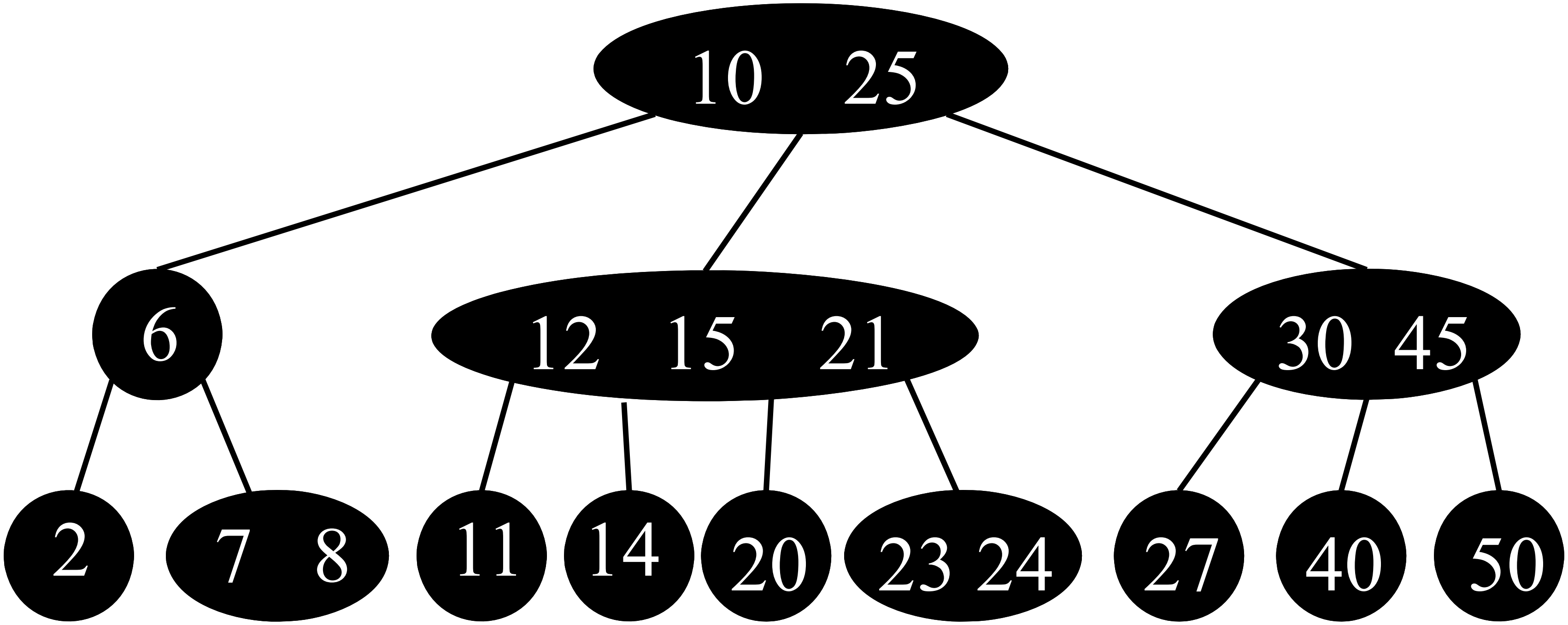


B-tree

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

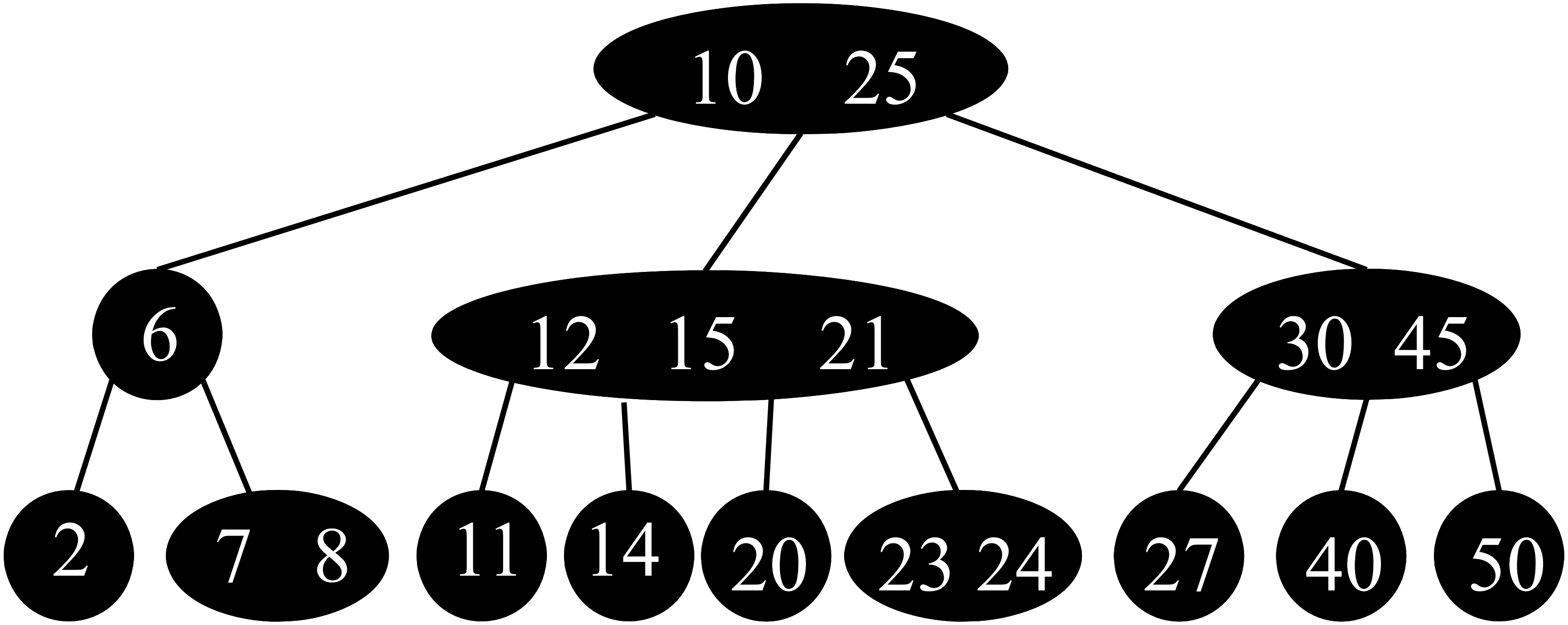
1. T is a $(2k)$-ary search tree
2. Every node, except the root, stores at least k-1 keys
   (every internal non-root node has at least k children)
3. The root must store at least one key
4. All leaves have the same depth

B-tree with k=2



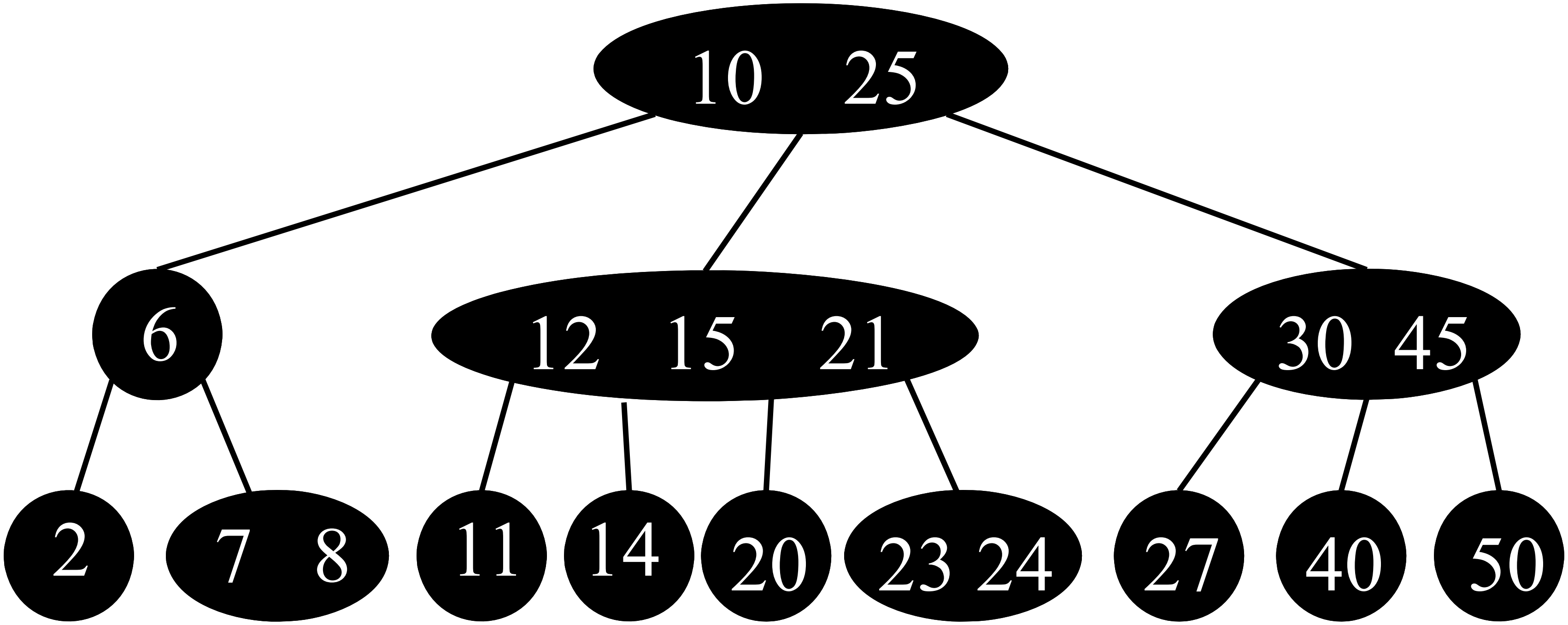
1. T is a (2k)-ary search tree

B-tree with *k*=2



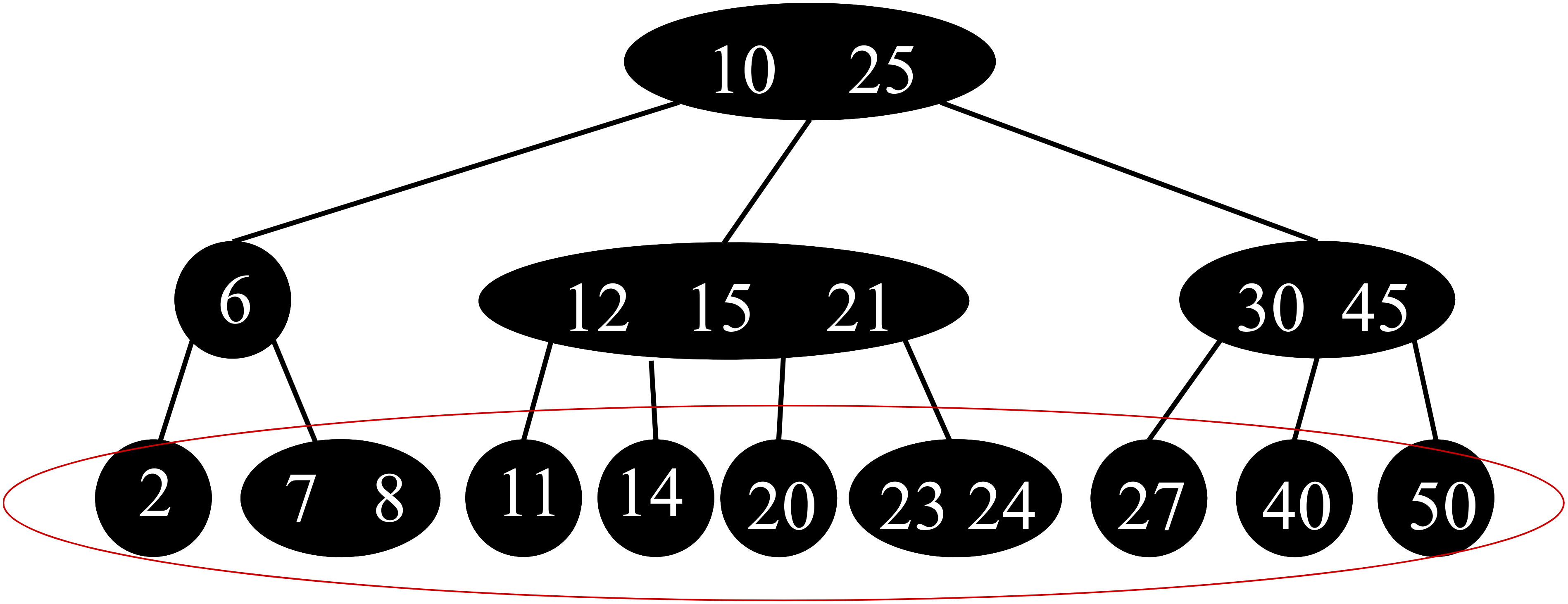
2. Every node, except the root, stores at least *k*-1 keys

B-tree with *k*=2



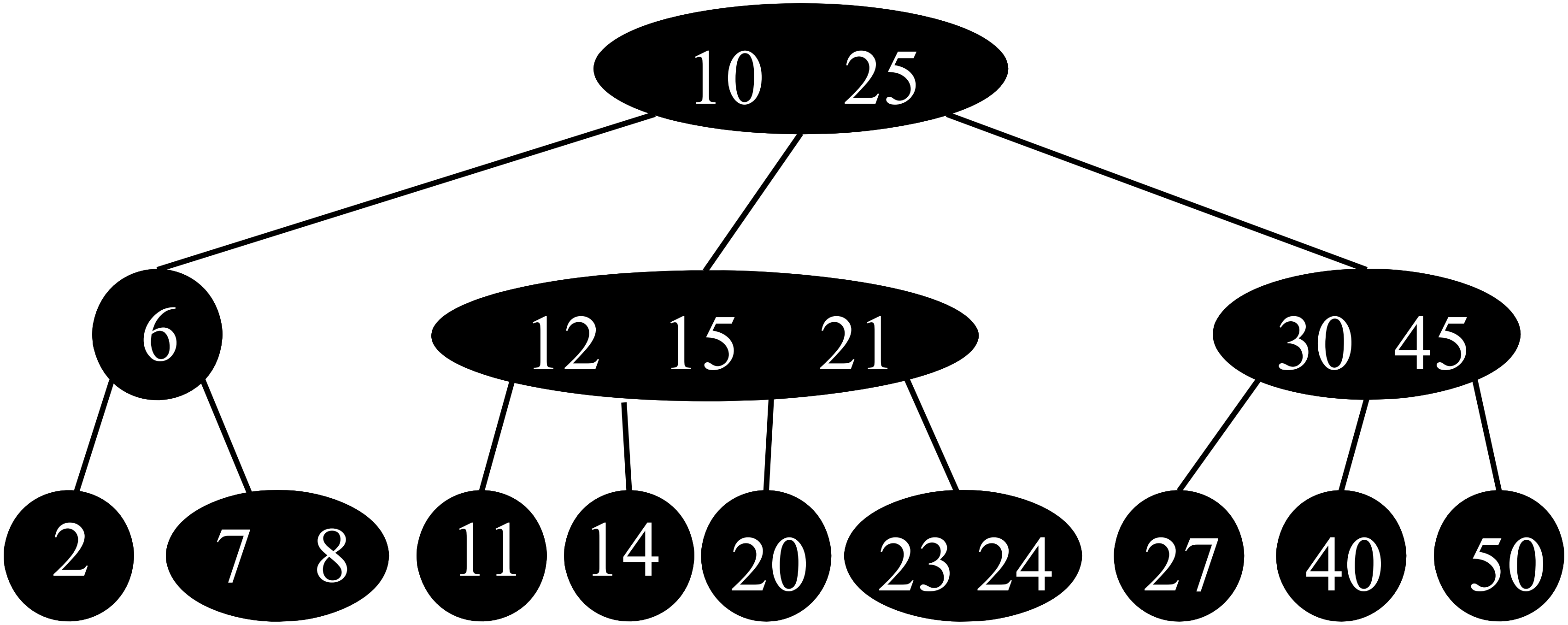
3. The root must store at least one key

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 and has height h holds:

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

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

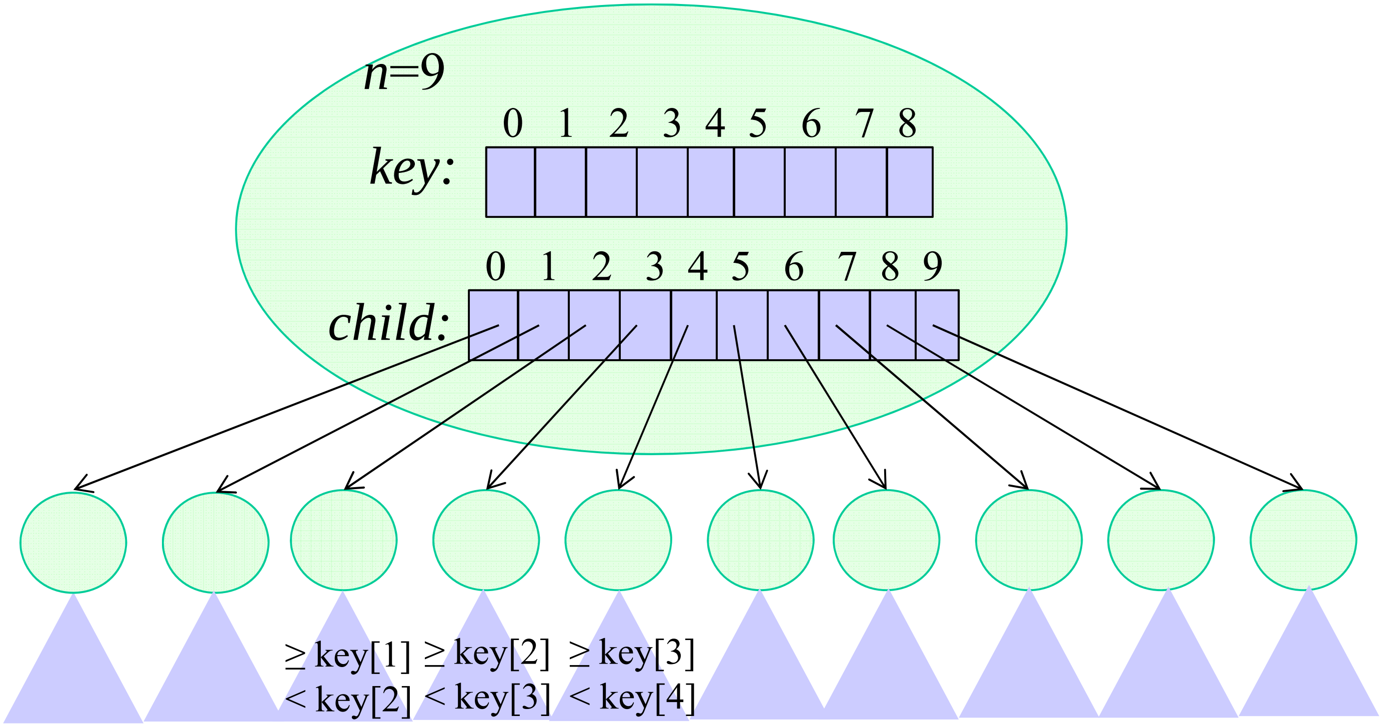
level 0 level 1 level 2 level 3

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

□

B-tree node

```
B-TREE-NODE:
    $n$                  // Number of keys
    $key[0..n-1]$        // Array of keys stored in non-decreasing
                         // order: $key[0] \le key[1] \le \ldots \le key[n-1]$
    $child[0..n]$        // Array of pointers to children nodes
```



B-tree search

```
B-TREE-SEARCH(x,key){ // Search key in node x
    i = 1
    // Find child (subtree) that contains key
    while (i < x.n ) && (key > x.key[i])
        i++
    // Found key in node
    if (i < x.n) && (key == x.key[i])
        return (x,i)

    if x is a leaf // key not found
        return null
    else { // Search for key deeper in subtree
        b=DISK-READ(x.child[i])
        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

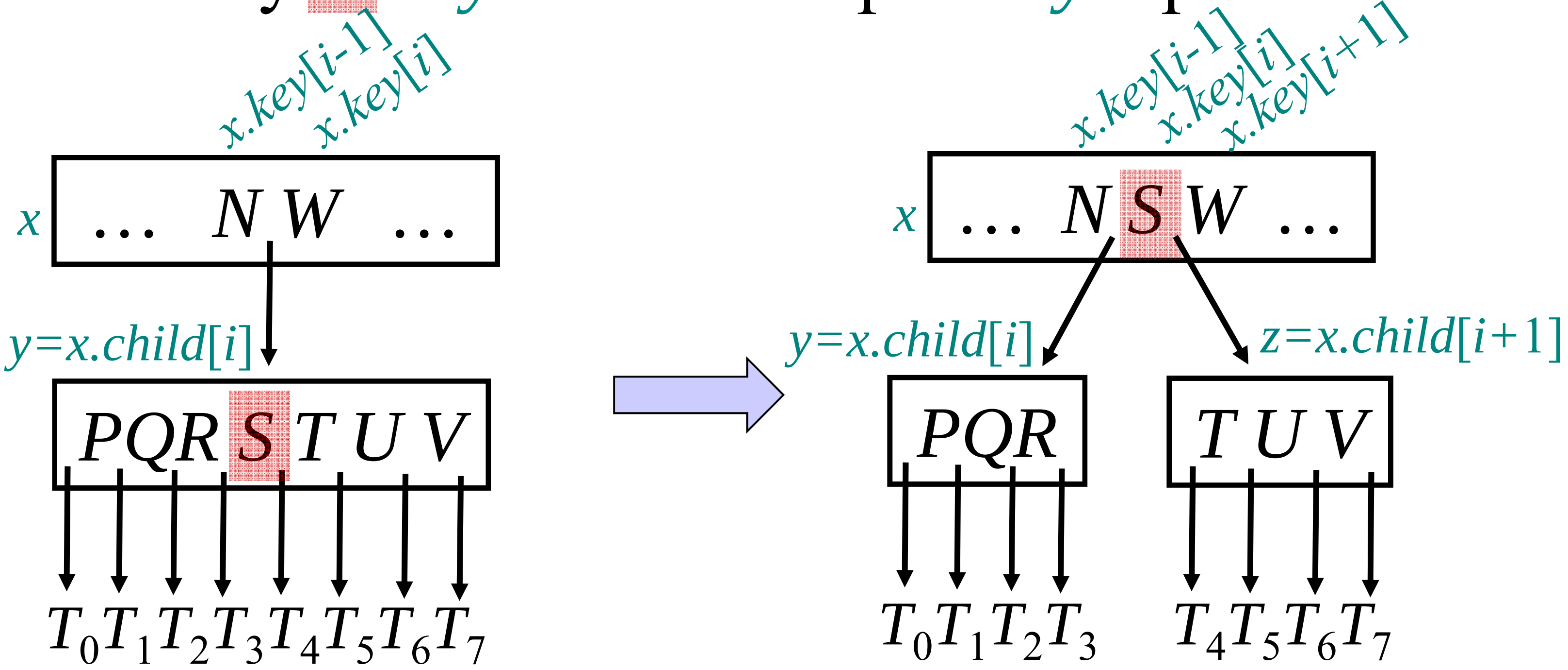
- There are different insertion strategies. We just cover one of them
- Make one pass down the tree:
  - The goal is to insert the new *key* into a leaf
  - Search where *key* should be inserted
  - **Only descend into non-full nodes:**
    - If a node is full, split it. Then continue descending.
  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

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

// full node y is i-th child of node x

- Split full node y ($\geq 2k-1$ keys) into two nodes y and z of $k-1$ keys

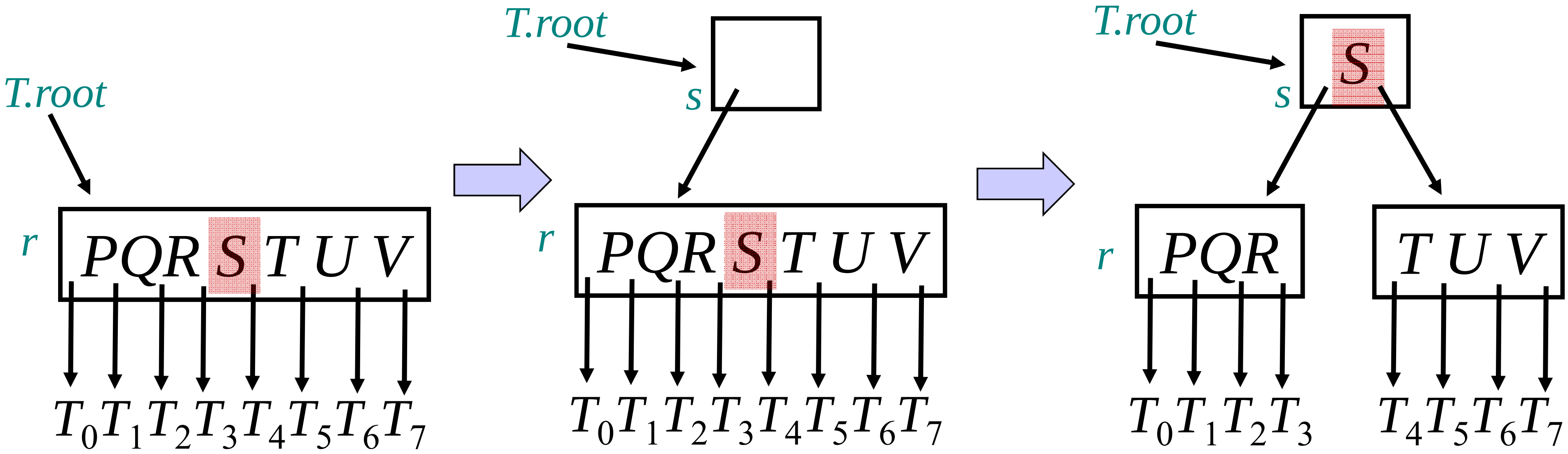
- Example below for $k = 4$:
  Median key S of y is moved up into y's parent x



Split root: B-TREE-SPLIT-CHILD(s,0)

- A new root node s is created. The **full** root node r is split in two.

- s contains the median key S of r and has the two halves of r as children
- Example below for $k = 4$



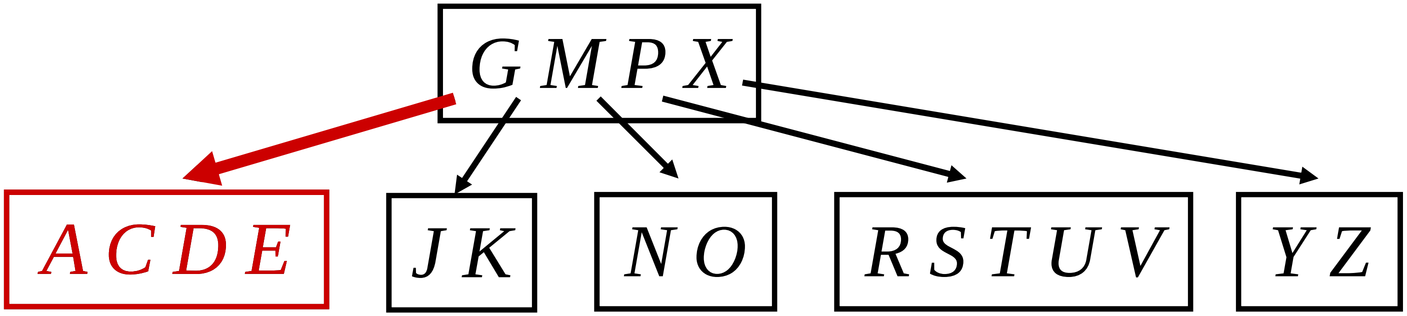
B-TREE-INSERT(*T*,*key*)

```
if (T.root.n == 2k-1) // root node is full
    // Insert new root node
    r = T.root
    T.root = ALLOCATE-NODE()
    T.child[0] = r
    // Split old root r to be two children of new root s
    B-TREE-SPLIT-CHILD(T.root,0)
    B-TREE-INSERT-NONFULL(T.root,key)
else
    B-TREE-INSERT-NONFULL(T.root,key)
```

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

```
if (x is a leaf){
    Insert key at the correct (sorted) position in x
    DISK-WRITE(x)
} else {
    c=child of x whose subtree should contain key
    DISK-READ(c)
    if (c is full){ // c contains $2k-1$ keys
        B-TREE-SPLIT-CHILD(x,i)
        c=child of x whose subtree should contain key
    }
    B-TREE-INSERT-NONFULL(c,key)
}
```

Insert example (k=3)



G M P X

A C D E

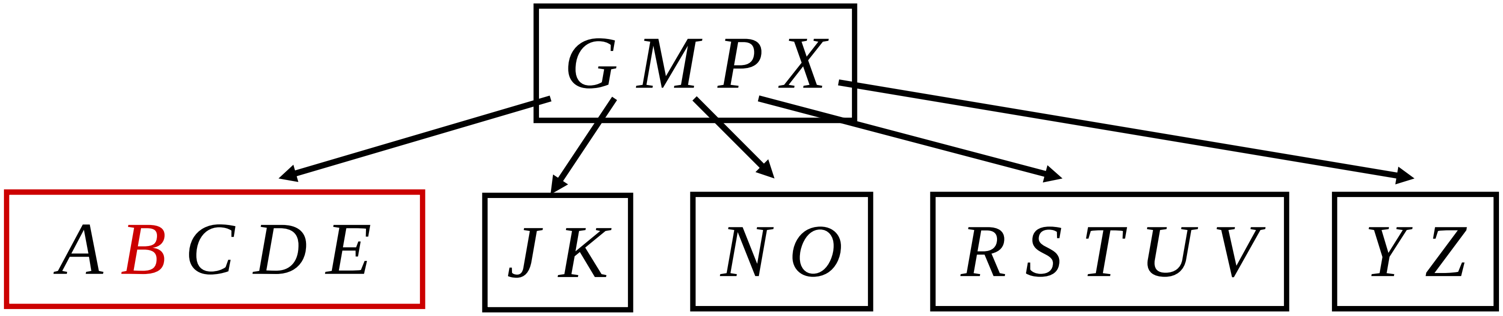
J K

N O

R S T U V

Y Z

- Insert *B*:



G M P X

A B C D E

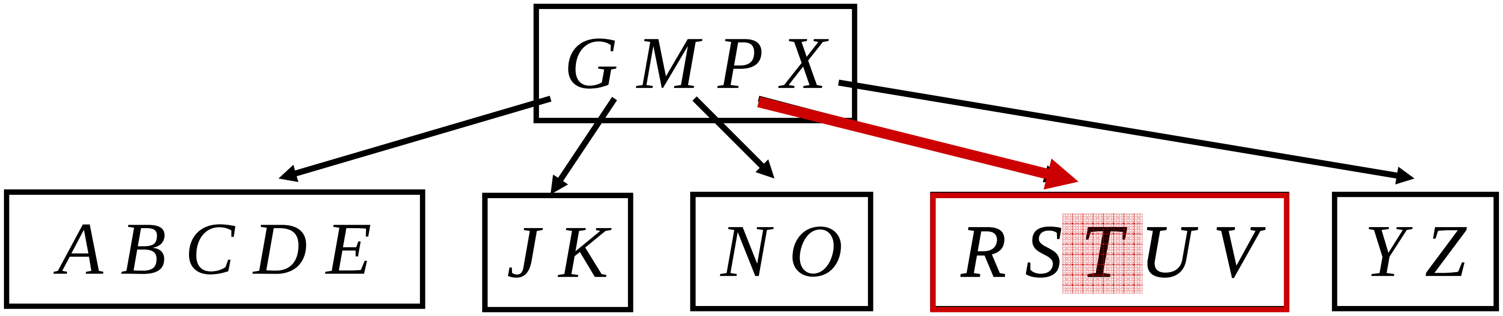
J K

N O

R S T U V

Y Z

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



G M P X

A B C D E

J K

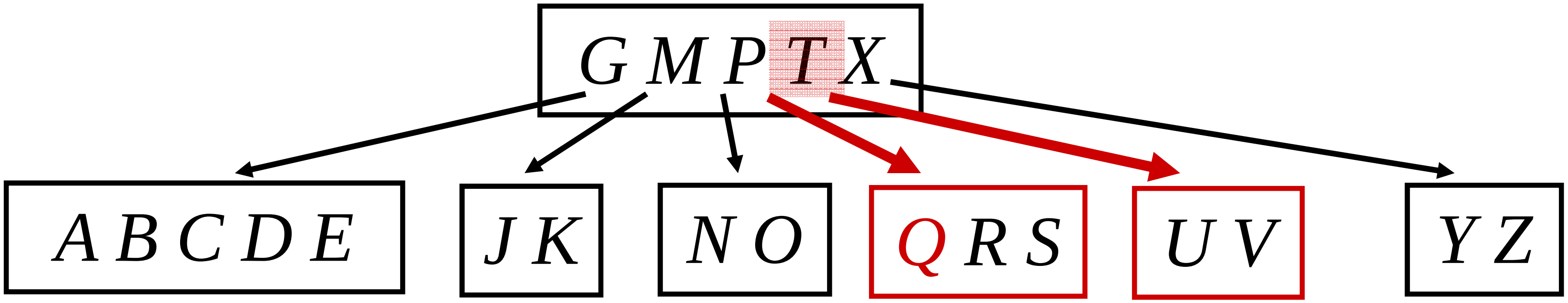
N O

R S T U V

Y Z

node is full

- Insert *Q*:



G M P T X

A B C D E

J K

N O

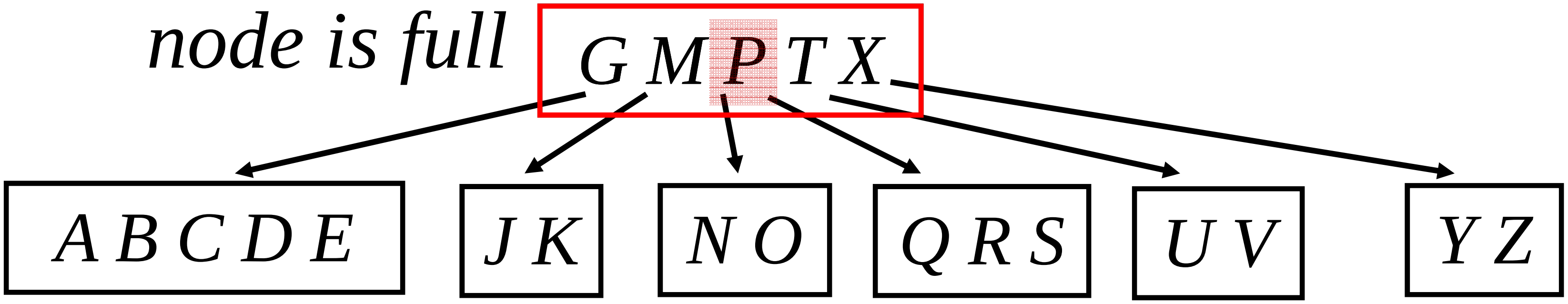
Q R S

U V

Y Z

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

node is full



G M P T X

A B C D E

J K

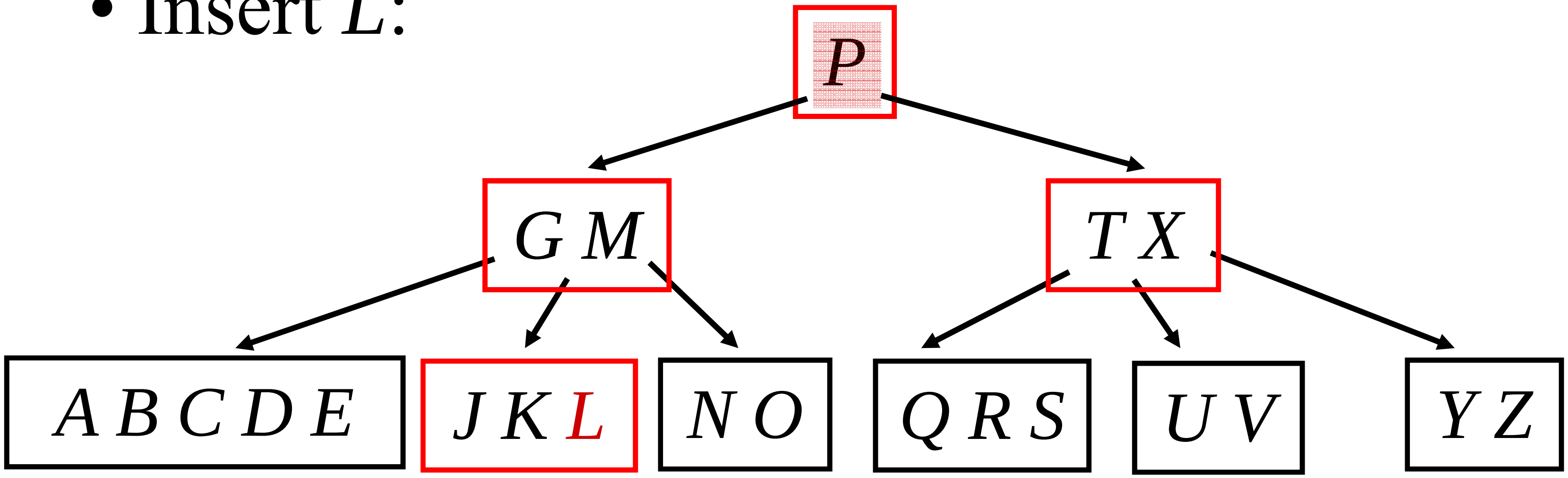
N O

Q R S

U V

Y Z

- Insert *L*:



P

G M

T X

A B C D E

J K L

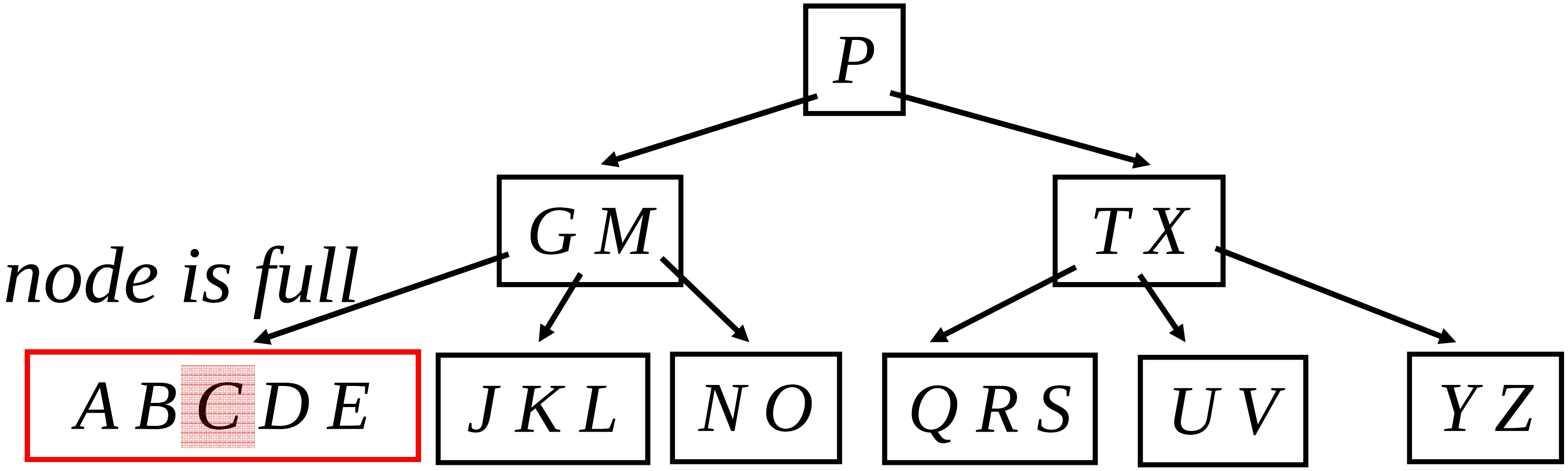
N O

Q R S

U V

Y Z

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


P

G M

T X

node is full

A B C D E

J K L

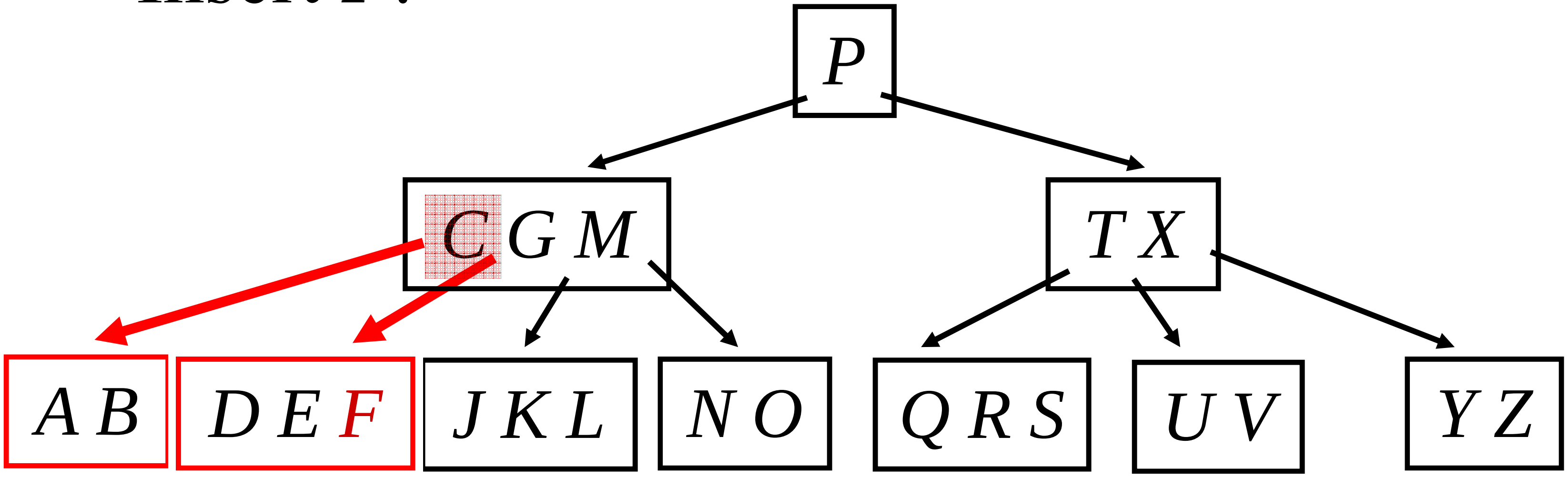
N O

Q R S

U V

Y Z

- Insert *F*:


P

C G M

T X

A B

D E F

J K L

N O

Q R S

U V

Y Z

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
- 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