COMP261 Lecture 23

Lindsay Groves

B Trees

Victoria
UNIVERSITY OF WELLINGTON
Te Whare Wānanga
o te Ūpoko o te Ika a Māui

CAPITAL CITY UNIVERSITY

Storing large amounts of data

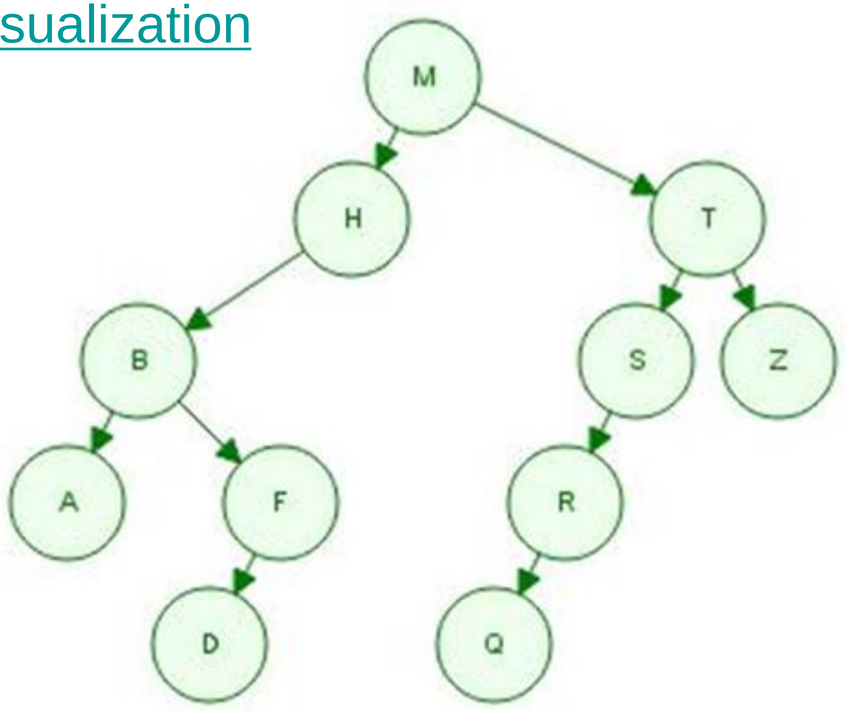
- File systems:
  - Lots of files, each file stored as lots of blocks.
- Databases:
  - large tables of data
  - each indexed by a key (or perhaps multiple alternative keys)
- How do we access the data efficiently:
  - individual items (given key)
  - sequence of all items (perhaps in order)
  - assume data is stored in files on hard drives (slow access time)
- Use some kind of index structure
  - Usually also stored in a file
- B Trees, B+ Trees: data structures and algorithms for
  - large data
  - stored on disk (ie, slow access)

COMP103 approaches:

- Efficient Set and Map implementations:
  - Hash Tables – fast, but hard to list in order
  - Binary Search Trees
  - Intended for in-core data structures

- http://www.cs.usfca.edu/~galles/visualization

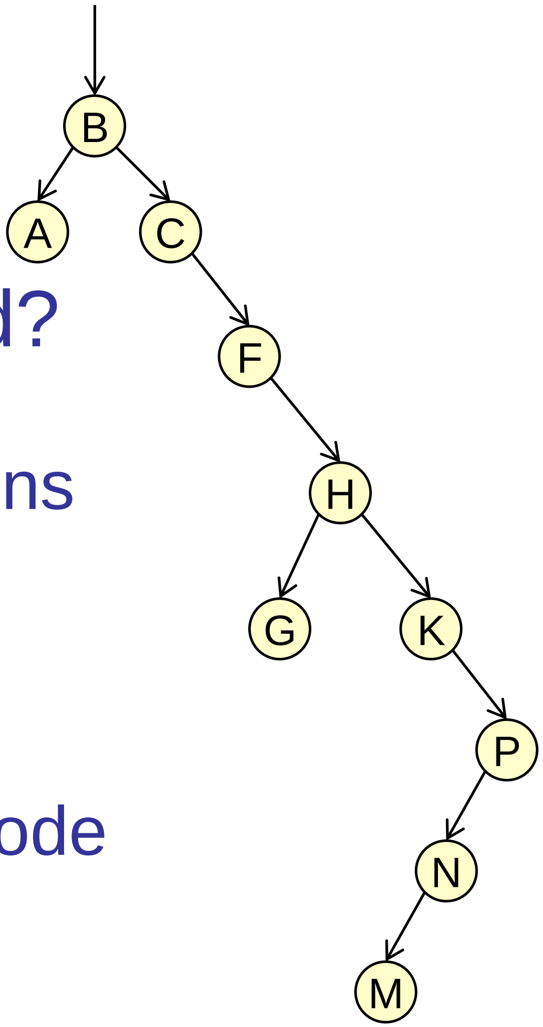
- Binary search tree
  - Add M H T S R Q B A F D Z
  - Search: Log(n)
  - What if we add in alphabetical order?



Problems with Binary Search Trees

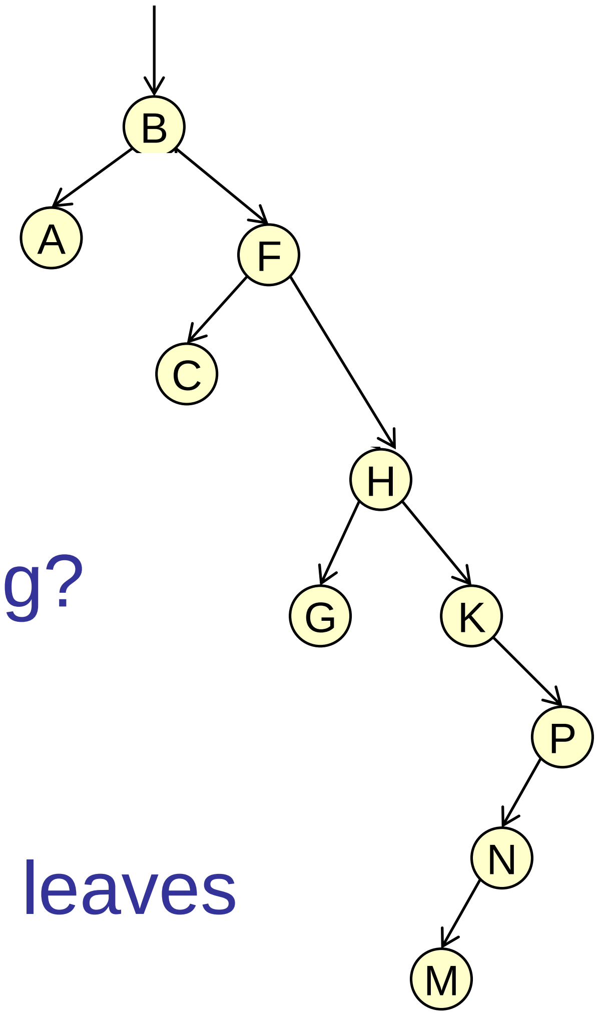
- Trees can become unbalanced
  ⇒ lookup times can become linear

- How can we keep the tree balanced?
  - ❖AVL trees: self-balanced by tree rotations
  - ❖Red-Black trees: node with a colour bit
  - ❖Splay trees: move recently accessed node to the root
  - ❖B/B+ trees: always add levels at the top!



Problems with Binary Search Trees

- Lots of pointer following
  ⇒ if each node is stored in a file, loading a node takes far longer than the comparisons we usually count!



- How can we reduce pointer following?
  - More data in each node
  - More children per node

    ⇒ "bushier" trees, fewer steps to leaves

Multiway Search Trees

- Allow each node to have more than one value and more than two children.
- An order *m* (or *m*-way) search tree has at most *m* children and *m*-1 values at each node.
  - Allows values and children be stored as arrays.
- Always have one more children than values.
  - Fill arrays from the left.
  - Think of as alternating children and values.
- As in BST, may store single values (set) or key:value pairs (map).

Multiway Search Trees

- Order values so that an in-order traversal visits values in ascending order.
  - Values at each node are ascending.
  - Values act as "separators" for lower subtrees:
    - All values in *i*th subtree are less than *i*th value and greater than (*i*+1)th value.
    - Any value between the *i*th and (*i*+1)th value at a given node must be in the *i*th subtree of that node.

Multiway Search Trees

- An order *m* search tree of height *h* has
  - At most m^{h+1}-1 values
  - At least *h* values (assuming we allow empty leaves)

- An order m search tree with *n* values has height
  - At least $\log_m n$
  - At most *h*

- Very good if tree – so long as balanced!

Multiway Search Trees

- Consider an order m tree of height 8, for differing values of *m*.

m	Max values
2	256
3	6,561
4	65,536
5	390,625
6	1,679,616
7	5,764,801
8	16,777,216
9	43,046,721
10	100,000,000

Balanced Multiway Search Trees

- How can we keep the tree balanced?
  - Keep all leaves at the same level.
  - Ensure every node is at least half full.
    - Well … as many as possible.
    - Mmm … how many is that?
- How can we maintain these properties?

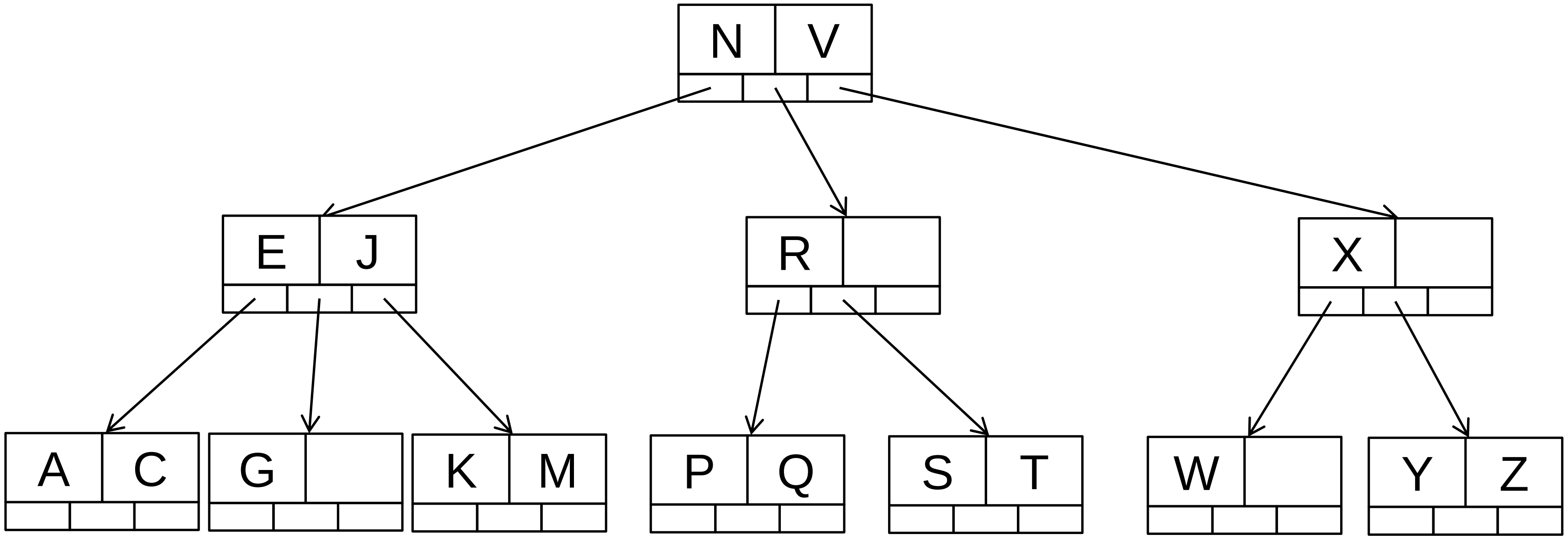
B Trees

- Variant of multiway search trees, designed for external data structures, stored in files.
- Non-leaf, non-root nodes always at least half full, i.e. have at least [*m*/2] children and [*m*/2]-1 data values.
- Leaves are always at the same level
  - Add new nodes at the root
  - Internal nodes always have at least two children
- Guarantees height is logarithmic in number of values.
- For storing files, *m* is chosen so on node occupies a disk block – makes trees very wide and very low, giving very fast access to huge amounts of data.

Examples

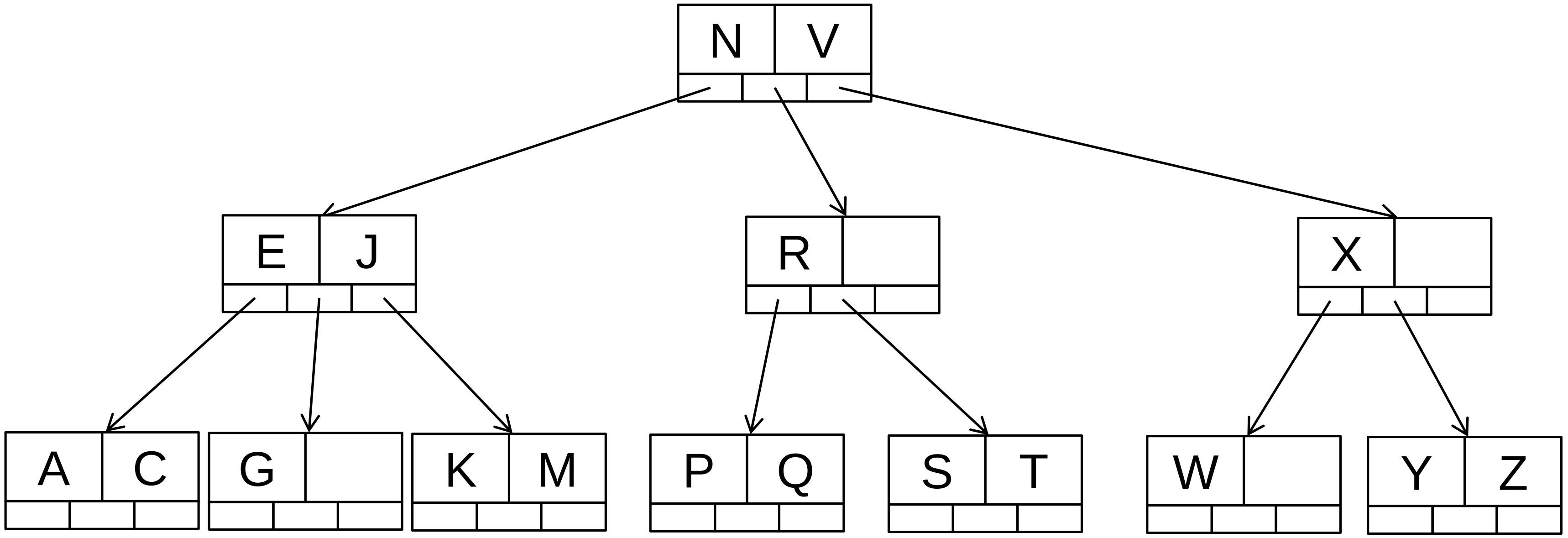
- An order 3 B tree is also called a “2-3 tree”.
- An order 4 B tree is called a “2-3-4 tree” or “2-4 tree”.
- In a 2-3 tree, every internal node has either:
  - 3 children and 2 values (“3-node”), or
  - 2 children 1 value (“2-node”)

  and leaves have 1 or 2 values



B Trees

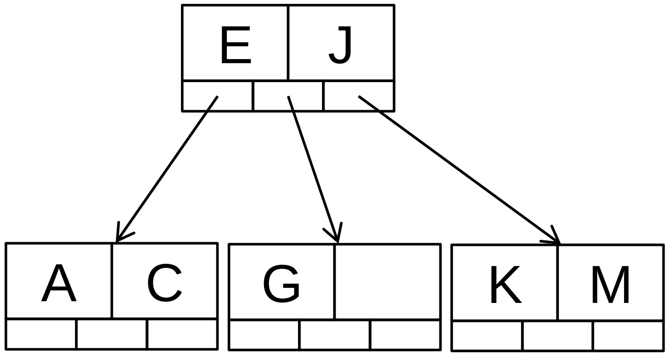
- Since an in-order traversal visits values in ascending order, we have a generalization of the ordering property for BSTs.
- For each *i* up to number of children (roughly):
  - All keys in *i*th subtree are less than *i*th key
  - And greater than (*i*-1)th key.



B Trees: Search

- Just like binary search, but more comparisons at each node
- For 2-3 tree, this is same as for ternary search tree.
- Search(key, node):

```
if   key < left_key
     return Search(key, left_child)
else if key == left_key
     return left_value
else if k < right_key
     return Search(key, middle_child)
else if key == right_key
     return right_value
else
     return Search(key, right_child)
```

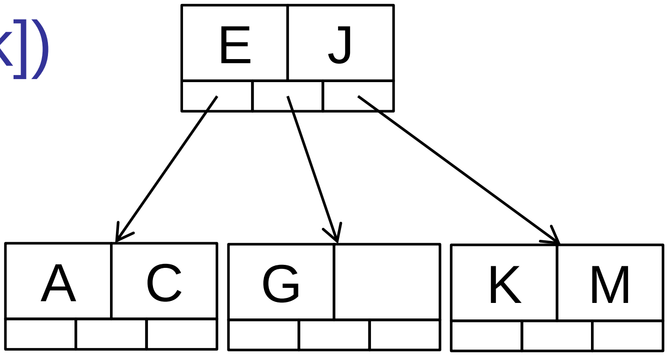


B Trees: Search

- For larger *m*, just generalize in the obvious way.

- Search(key, node):

```
    for  i = 0 to k-1     (k is number of keys in node)
        if   key <= keys[i] then break
    if  key == keys[i] then
            return values[i]
    else   return Search(key, children[k])
```



- For large *m*, use binary search to find smallest key greater than or equal to search key.

B Trees: Insert

- Search for leaf where the key should be.
- If leaf is not full, add item to the leaf.
- If leaf is full, split this node to make space
  - Spread existing values plus new one over two leaves
- Push “middle” value up to parent node

$$\begin{tikzcd}
 & & & {[\ldots \mid k'_{m/2} \mid \ldots]} \arrow[dl] \arrow[dr] & \\
{[k_1 \mid \ldots \mid k_{m-1}]} & \longrightarrow & {[k'_1 \mid \ldots \mid k'_{m/2}-1]} & & {[k'_{m/2}+1 \mid \ldots \mid k'_m]}
\end{tikzcd}$$

- $(k'_1, \ldots, k'_m) = (k_1, \ldots, k_m)$ with new key inserted in order

B Trees: Insert

To insert item (key,value) in leaf:

- Search for leaf where the key should be.
- If leaf is not full: add item to the leaf.
- If leaf is full:
    - Find the middle key from among existing items and the new item
    - Create a new leaf node to go after the current one
        - Leave items before middle key in original node
        - Put items after middle key in new leaf node,
        - Push middle item up to parent, along with pointer to new node
        - (This becomes new “separator” value in parent)

B Trees: Insert

To add new item to parent:

- If parent is not full:

    add new item to parent, and
    add new child pointer just right of new item

- else:

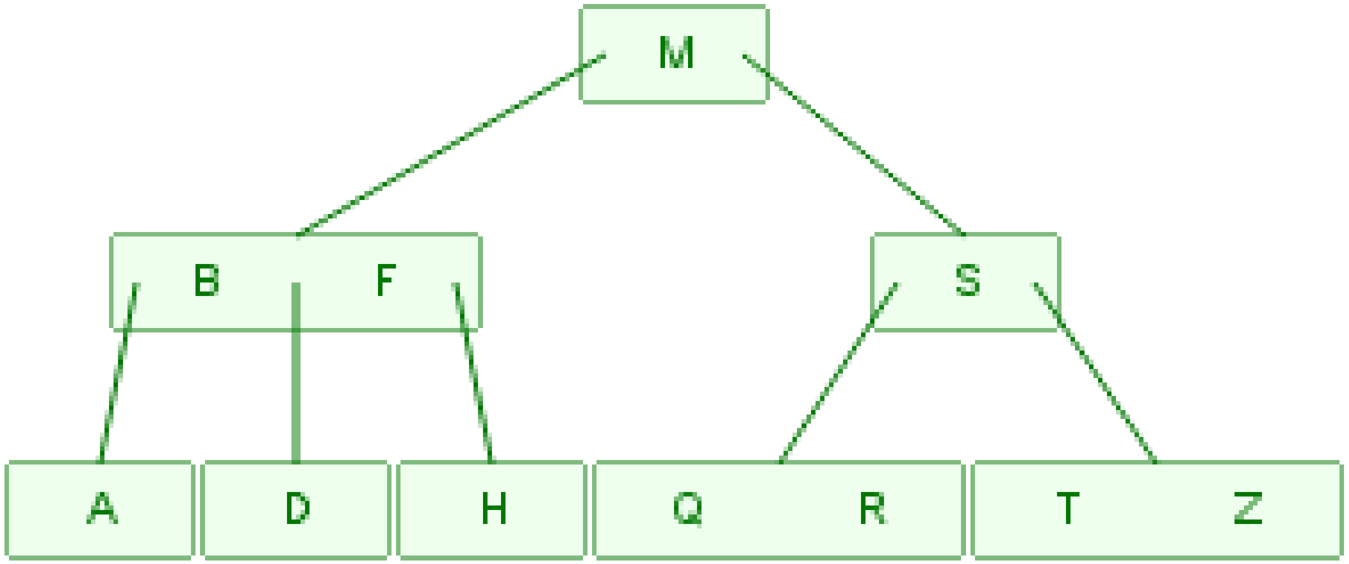
    split parent node into two nodes (like leaf)
    push middle item up to grandparent.
    add pointer to new child just right of pushed up item

If splitting goes right up to the root, so the root is split:
Create a new root node with one value and two children.

(This is why we can’t insist on root node being half full!)

2-3 B Tree: Inserting values

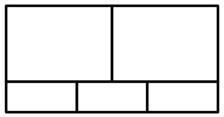
- Add M H T S R Q B A F D Z



- http://www.cs.usfca.edu/~galles/visualization/BTree.html

2-3 B Tree: Inserting values

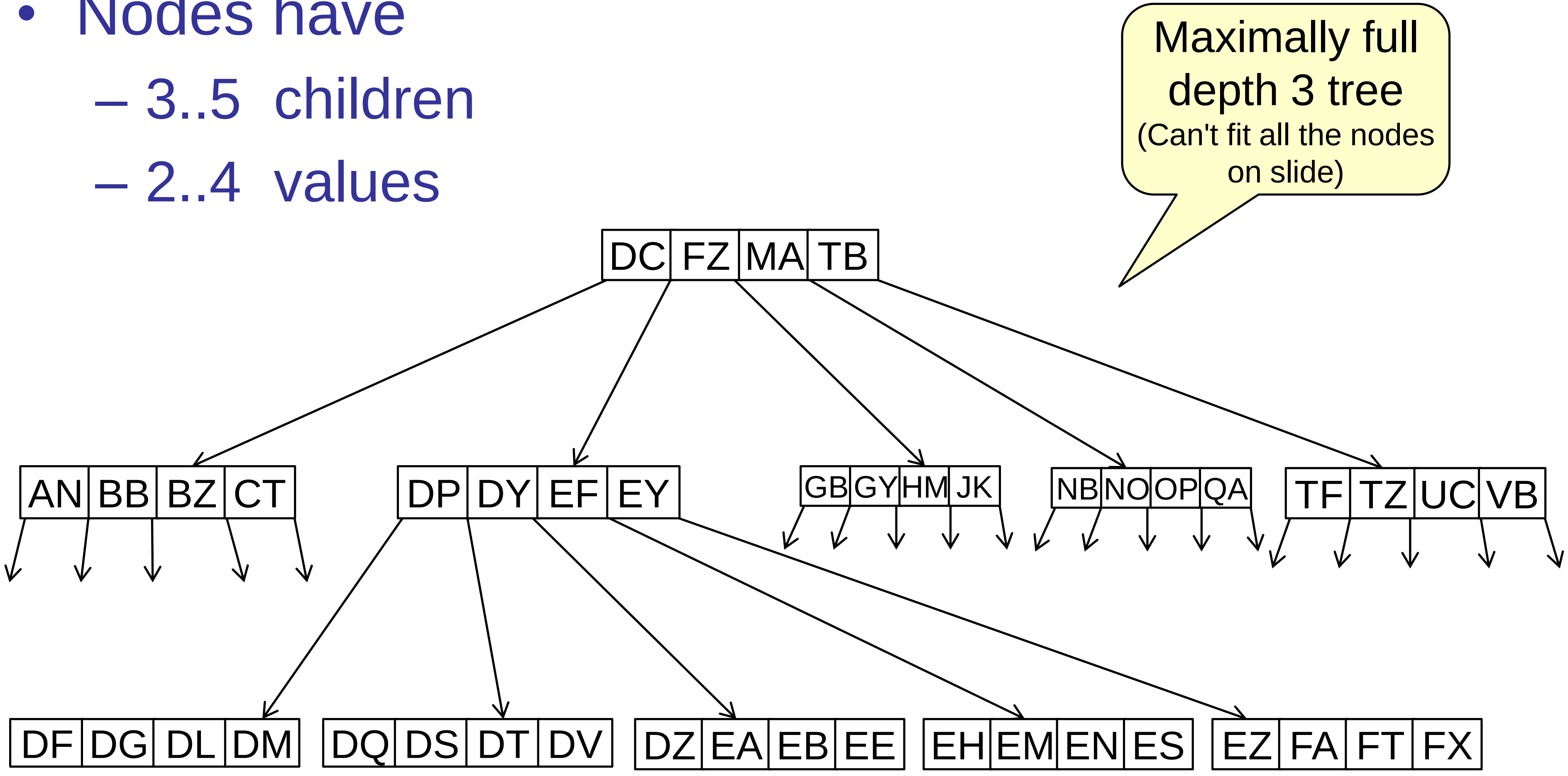
- Add 8, 5, 1, 7, 3,12, 9, 6



- http://www.cs.usfca.edu/~galles/visualization/BTree.html

Example: B Tree of order 5

- Nodes have
  - 3..5 children
  - 2..4 values

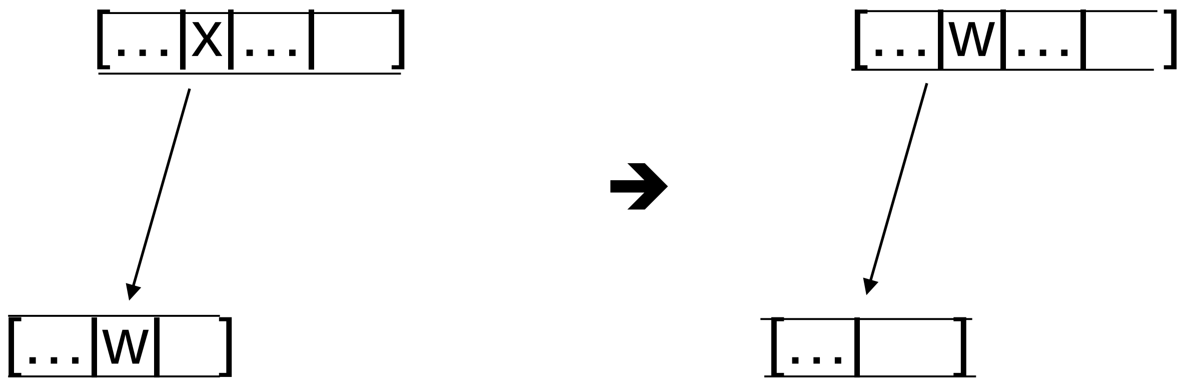


B Tree: Deletion

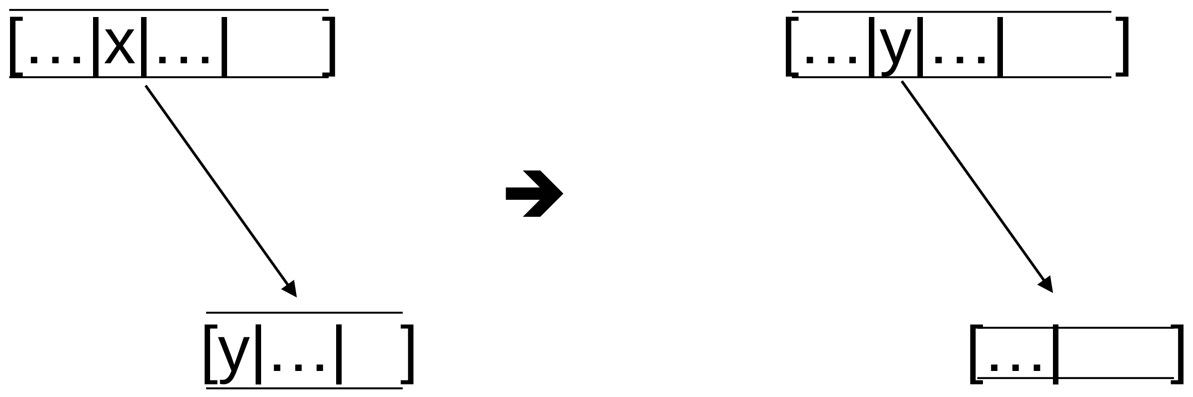
- Search for the node, say *N*, containing value (*x*) to be deleted.
- If *N* is a leaf, delete *x* from *N*.
- If *N* is an internal node, must find a value to replace it.
  - Must be next smallest or largest in tree.
  - i.e. preceding or following element in in-order traversal.
  - i.e. largest in right subtree or smallest in left subtree.
  - Must be in a leaf.
  - Delete from its leaf and copy into place where *x* was in *N*.
- In either case, if leaf is now under-full, we need to rebalance the tree.

B Tree: Deletion

- E.g. if children are leaves:



- or:



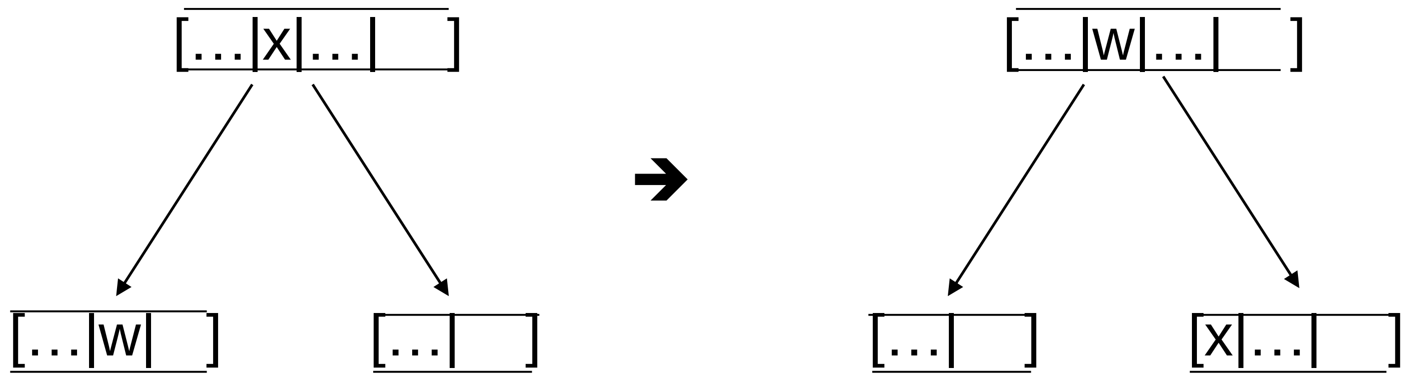
B Tree: Rebalancing

- After deleting *x*, one leaf node has one fewer values.
- If the leaf now under-full, we must rebalance the tree.
- If leaf has an adjacent sibling with more than minimum number of values, *rotate* a value from sibling, through parent to the under-full leaf;
- If no adjacent sibling has more than minimum number of values, *merge* the under-full node with a sibling into a single leaf.

B Tree: Rebalancing - Rotate

If a sibling node has more than the minimum number of values: move its smallest/largest value to the parent node, in place of the separator, and move the separator to the under-full leaf.

- If left sibling has a “spare” value:



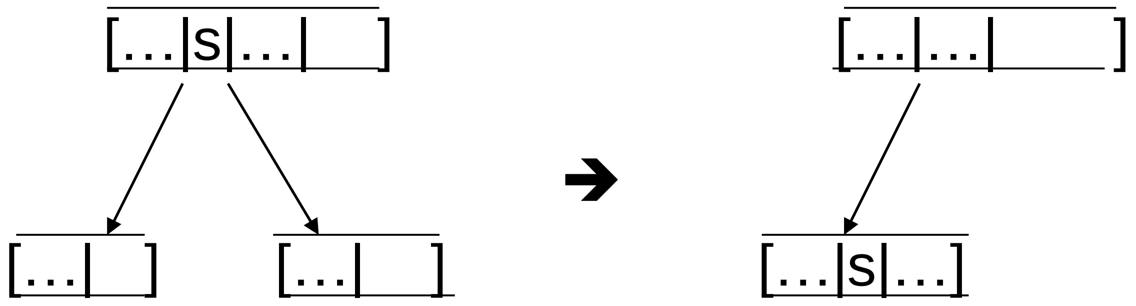
- If left sibling has a “spare” value:
  - Exercise!

B Tree: Rebalancing - Merge

If there is not a sibling with a “spare” value:
merge the under-full leaf with a sibling (which has minimum number of values)

- – Move the separator and all of the values from the sibling into the under-full node.
- – Delete the sibling and delete the separator from the parent.

- Merging with sibling on the right:

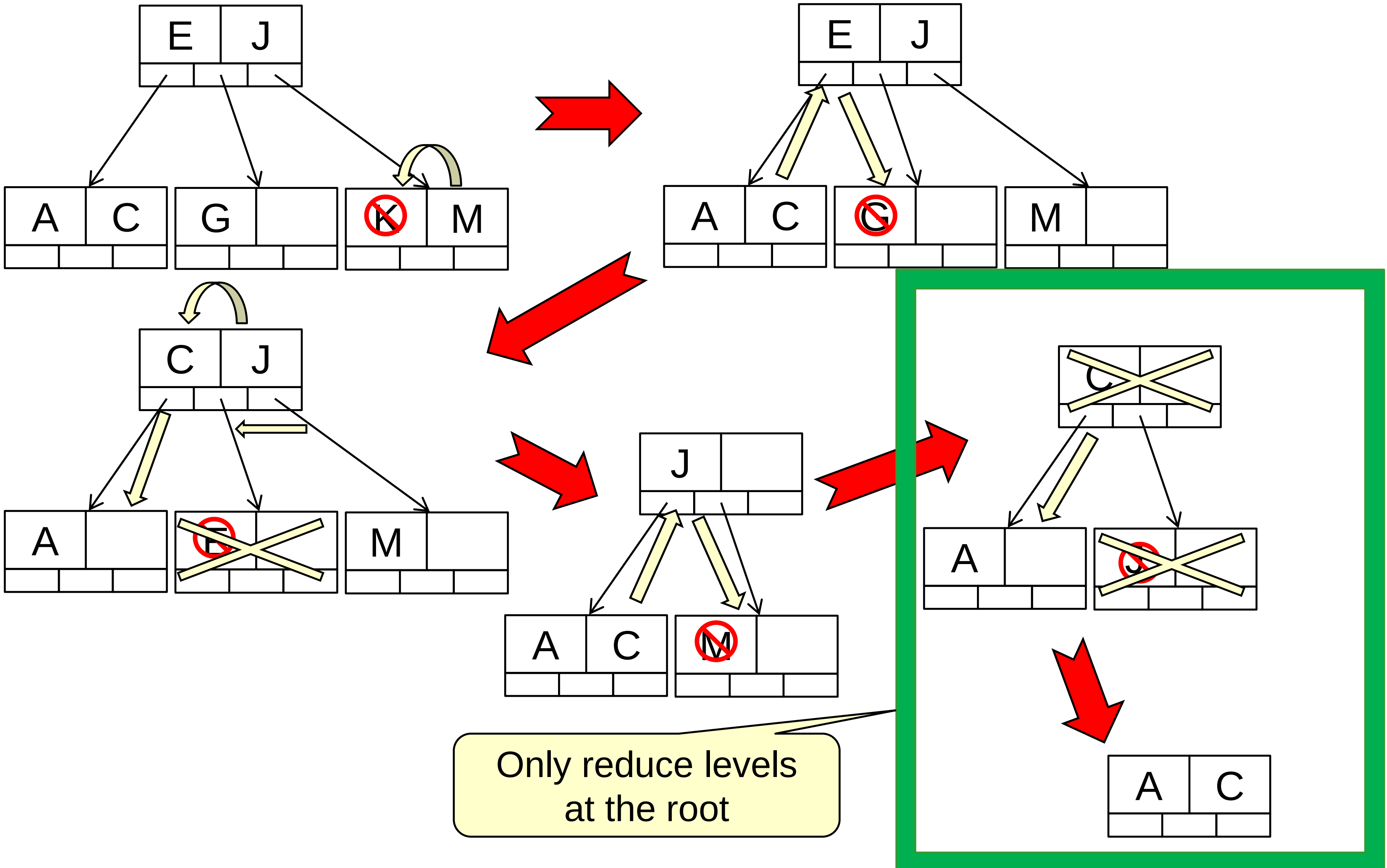


- Merging with sibling on the right: Exercise!

B Tree: Rebalancing - Merge

- After a merge, the parent has one fewer values.
- If the parent is now under-full, rebalance it.
  - So rebalancing continues back up the tree.
- If the parent is the root, we can’t rebalance
  - We can now answer our earlier question:
  - All nodes except the root are at least half full.
- If the parent becomes empty:
  - It is deleted and the new merged node becomes the root.
  - The height of the tree reduces by one.

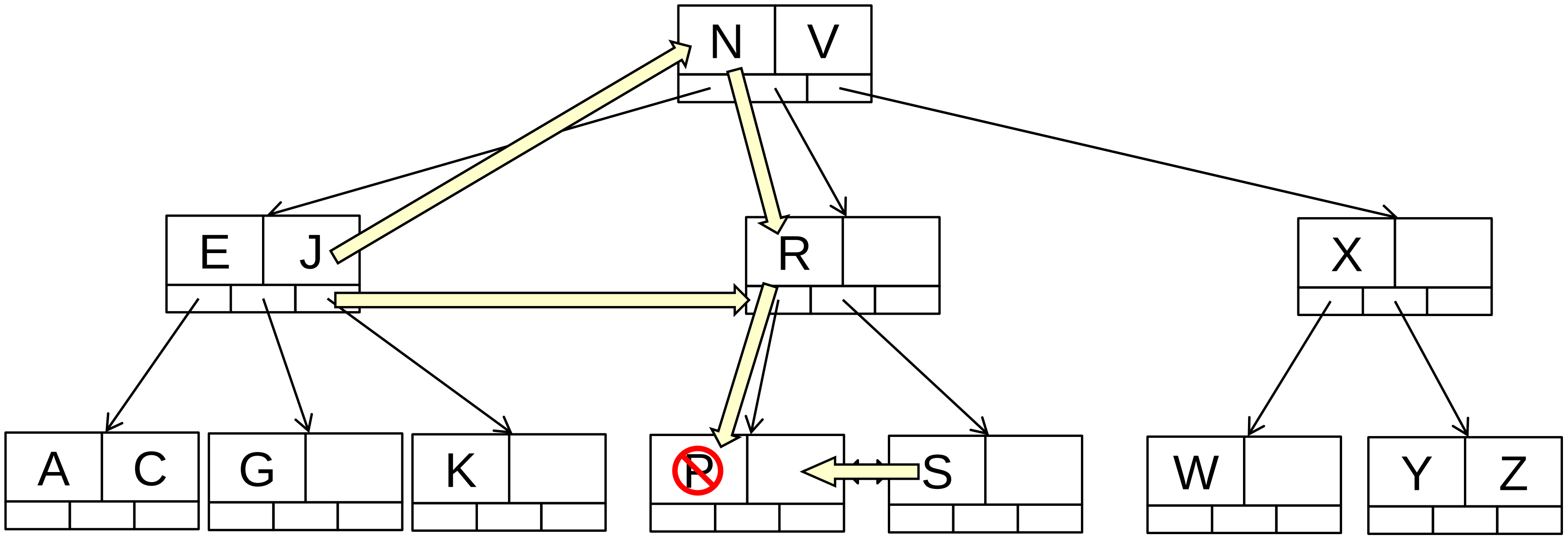
Deleting from leaves



Only reduce levels at the root

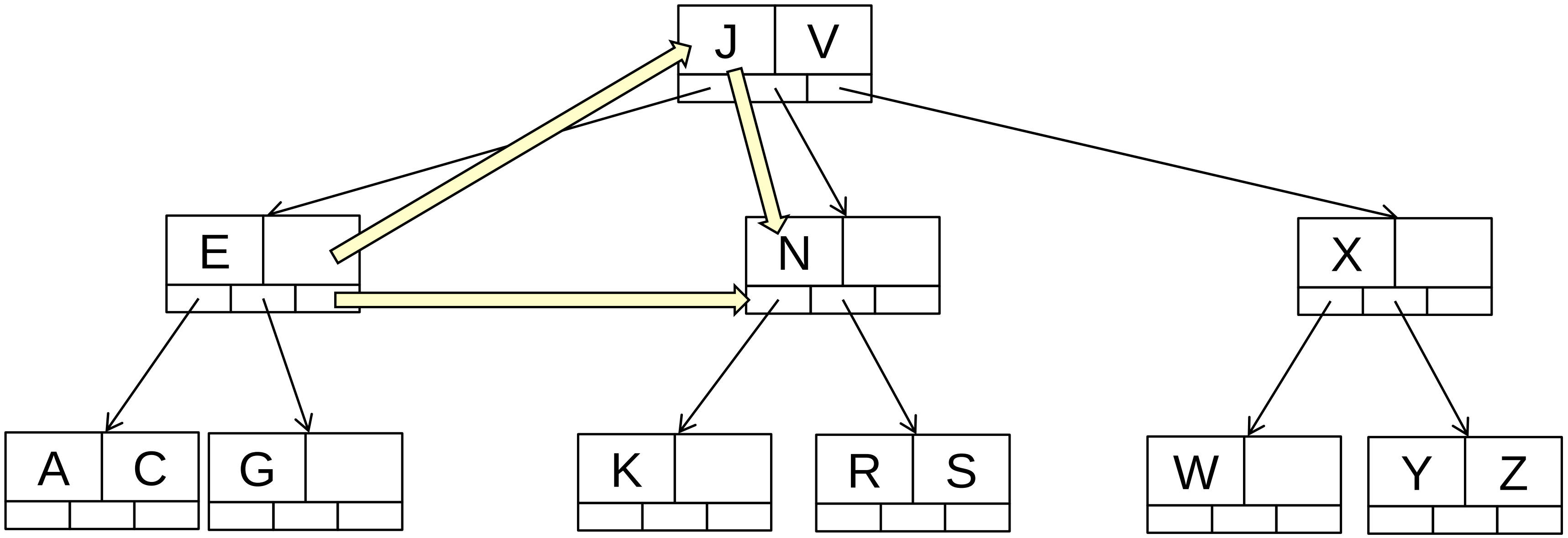
More deletions:

- When internal node becomes too empty
  - – propagate the deletion up the tree



More deletions:

- Deletion from internal node ⇒
  - – rotate key and child from sibling



Analysis

- B trees are balanced:
  - A new level is introduced only at the top
  - A level is removed only from the top

  Therefore:
  - all leaves are at the same level.
- Cost of search/add/delete:
  - $O(\log_{[m/2]}(n))$ (at worst) = depth of tree with all half full nodes
  - $O(\log_{m}(n))$ (at best) = depth of tree with full nodes
  - if 100 million items in a B tree with m = 20,

    $\log_{10}(100{,}000{,}000)$ = ? 8

    $\log_{20}(100{,}000{,}000)$ = ? 6.14
  - if billion items in a B tree with m = 100,

    $\log_{50}(1{,}000{,}000{,}000)$ = ? 5.3

    $\log_{100}(1{,}000{,}000{,}000)$ = ? 4.5