B/B+ TREES

4.7

AVL limitations

- Must be in memory

- If I don't have enough memory, I have to start doing disk accesses
  - The OS takes care of this
  - Disk is really slow compared to memory

Disk accesses

- What is the worst case for the number of disk accesses for:
  - Insert
  - Find
  - Delete

AVL Limitations

- AVL produce balanced trees, this helps keep them short and bushy
- The more bushy our tree, the better
  - Reduces the height, so reduces number of accesses
- Can we make our trees shorter and bushier?

AVL Limitations

- AVL produce balanced trees, this helps keep them short and bushy
- The more bushy our tree, the better
  - Reduces the height, so reduces number of accesses
- Can we make our trees shorter and bushier?
  - Yup, just add more children

M-ary trees

- Binary trees have 1 key and 2 kids
- M-ary trees have M children
  - Requires M-1 keys
- What is the height?

AVL limitations

- If I add more children, then my rotations become very complex.

- Just by adding a center child, I double the rotations
  - CC
  - CL
  - CR
  - RC
  - LC

More Children

- How can I maintain balance with more children?

More Children

- Require every node is at least half full
  - Otherwise degenerates to a binary tree or linked list
- How can I maintain balance with more children?
- If every leaf is at the same level, is it balanced?

B/B+ Tree

- M-ary tree with M>3
- Every node at least half full
  - Except the root
- Data is stored in leaves
- Internal nodes store M-1 keys, which guide searching
- All leaves are at the same depth

B+ tree

- Internal node values
  - No standard, many ways to do it
  - We will use the smallest value in the leaf to the right

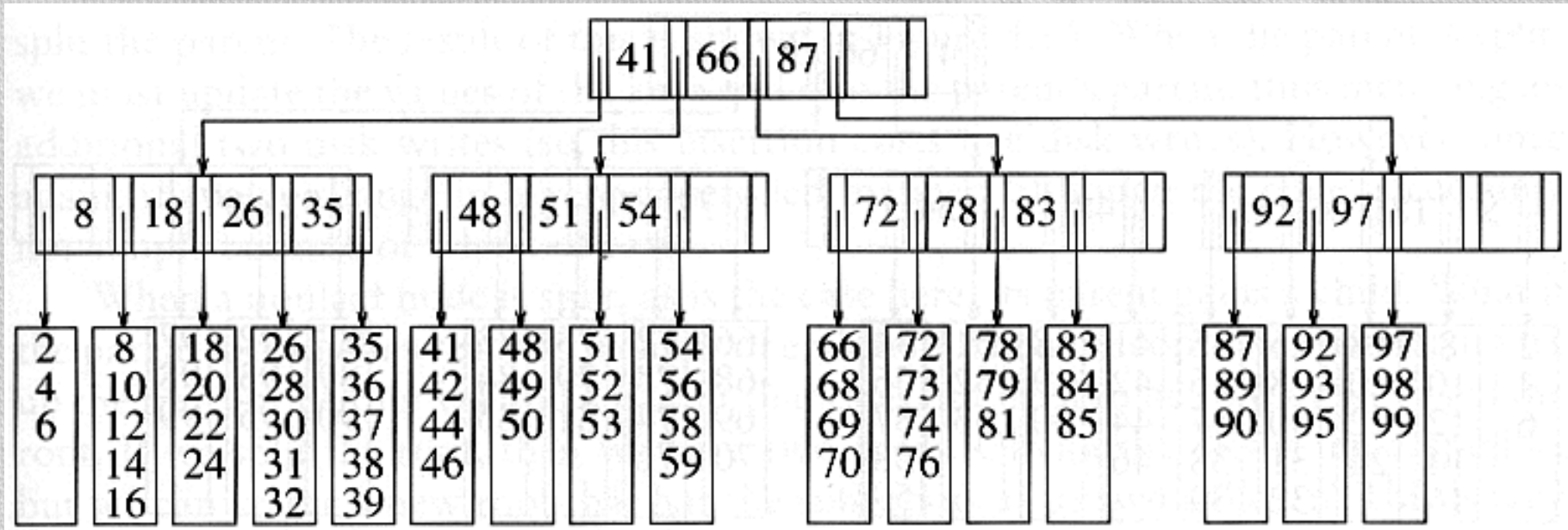


Figure 4.62 B-tree of order 5

B+ Tree

- Search
  - Similar to binary trees
    - If search is smaller than key, go left
    - If search is larger than the key, look at the next key
      - If there is not a next key, go right

Searching

- Find 41, 99, 66, 52

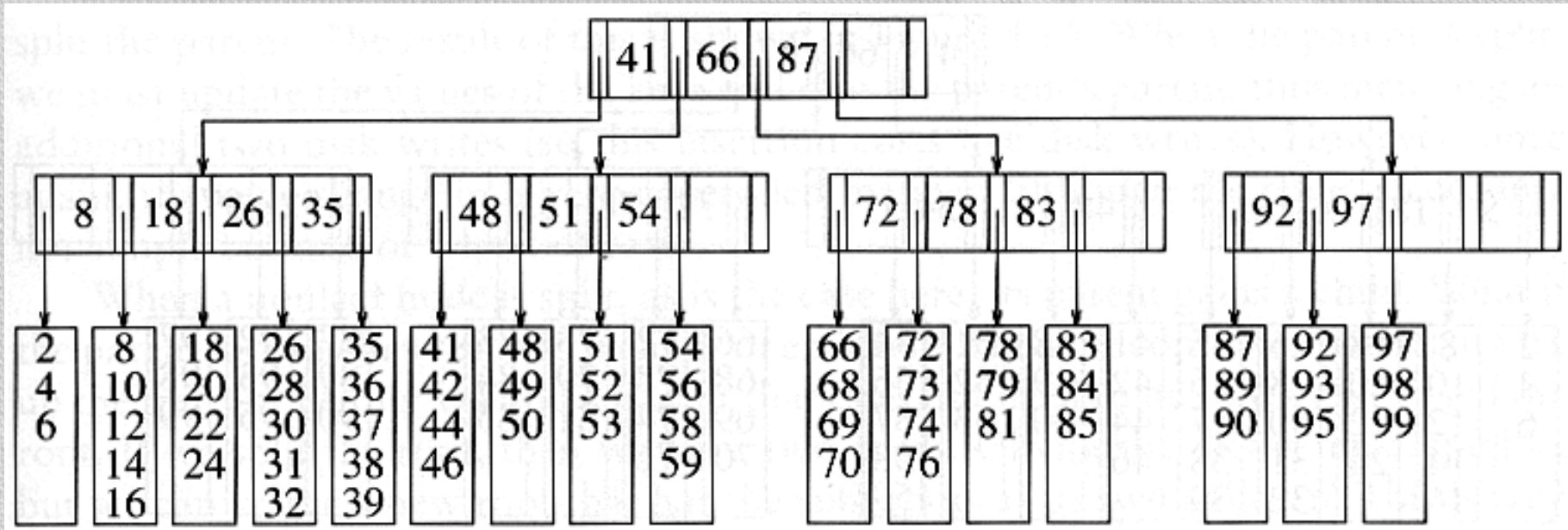


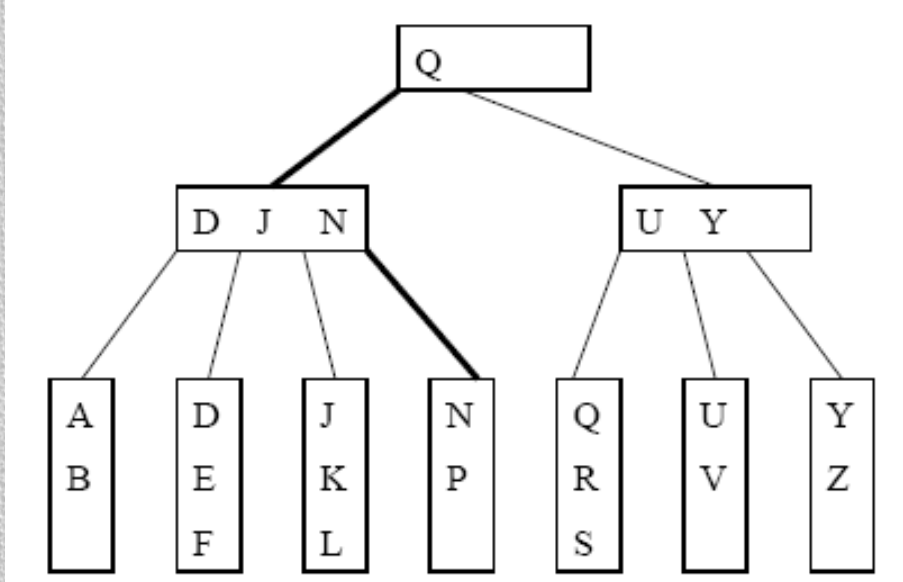
Figure 4.62 B-tree of order 5

B+ Tree

- Insert
  - Similar to binary trees on the way down
    - Find where to insert
    - When hit a leaf, insert at the leaf
  - Have to check for overflowing nodes on our way back out
  - Also have to update parent keys

Insert

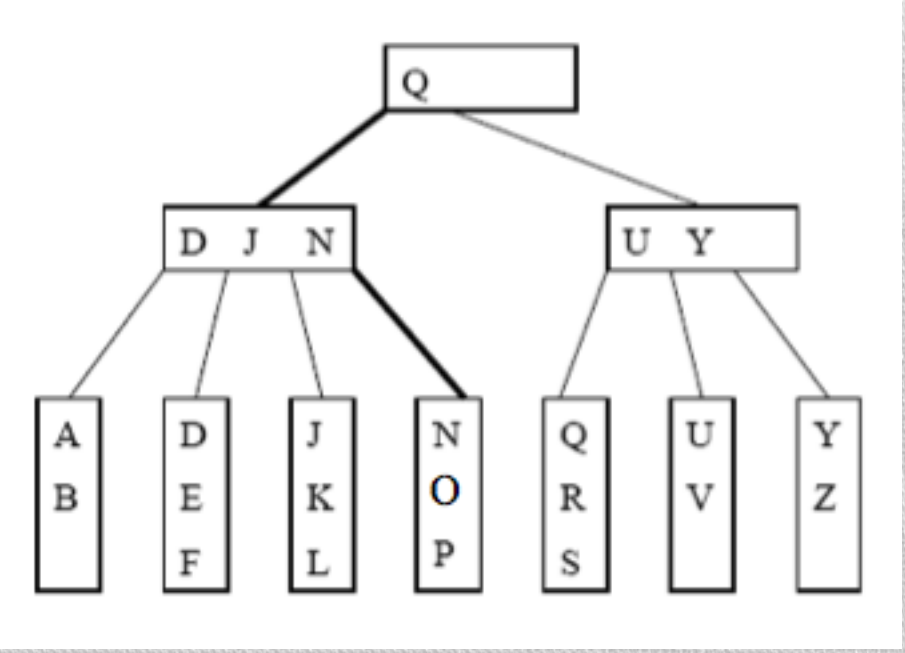
- Insert O



Search for O.

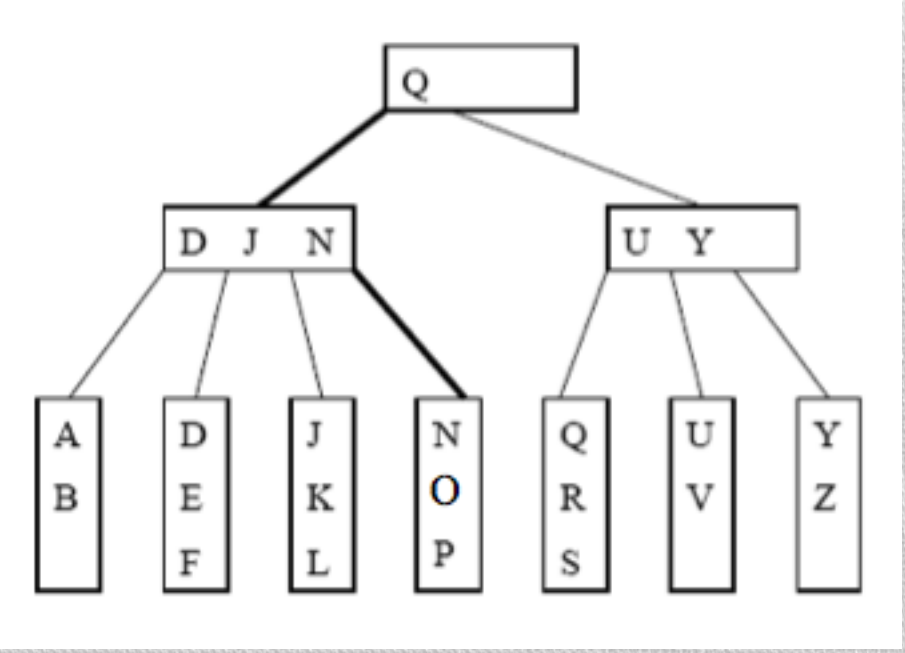
Insert Continued

- Insert O
  - After the insert



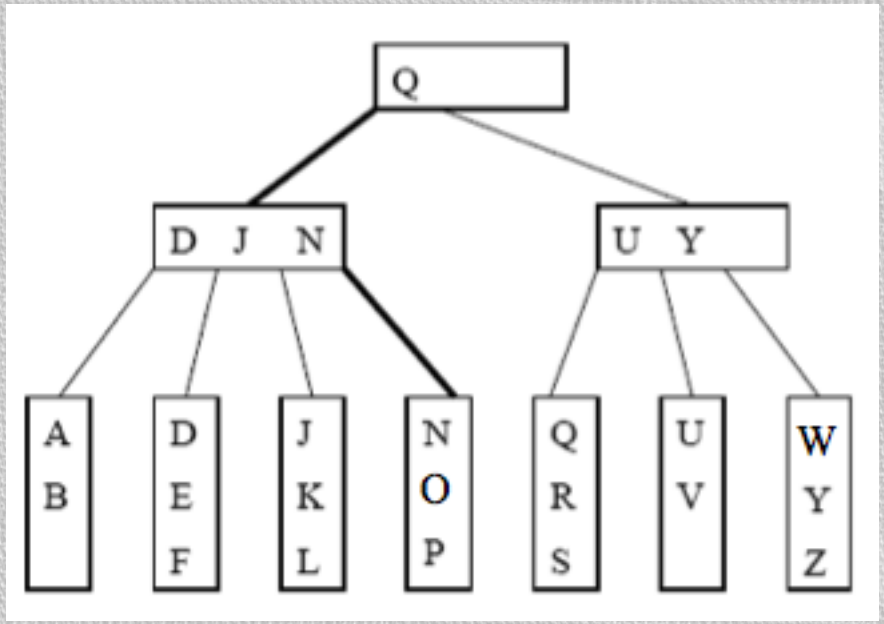
Insert

- Insert W



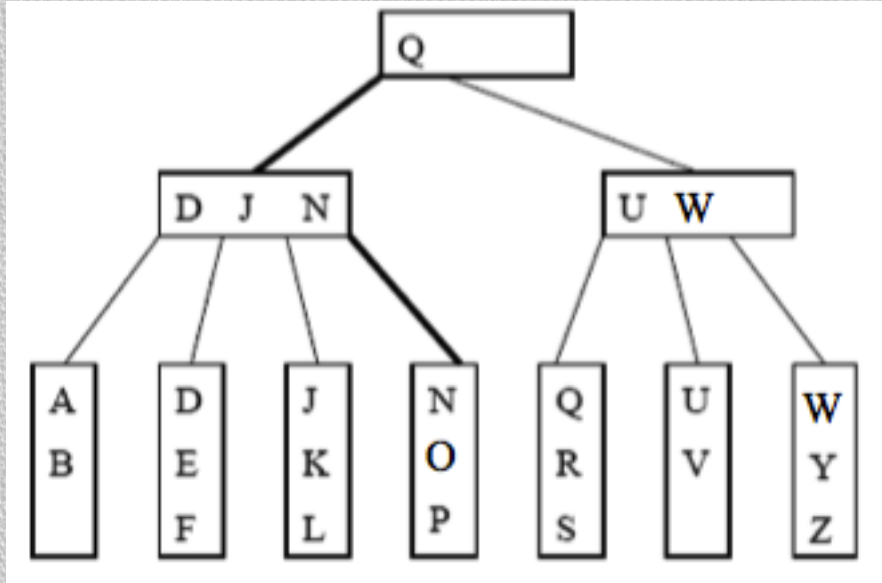
Insert Continued

- Insert W
  - After the insert



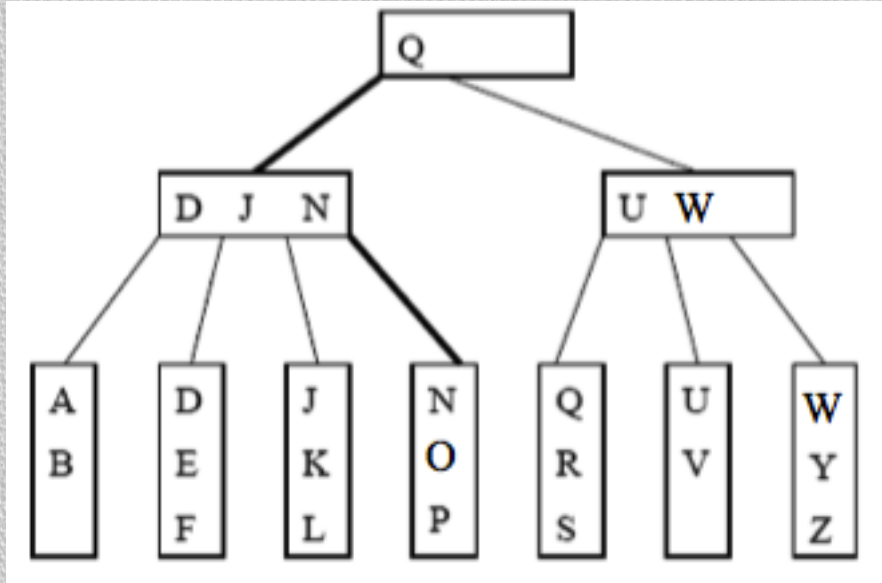
Insert Continued

- Insert W
  - Update the parent key with the smallest value in the right



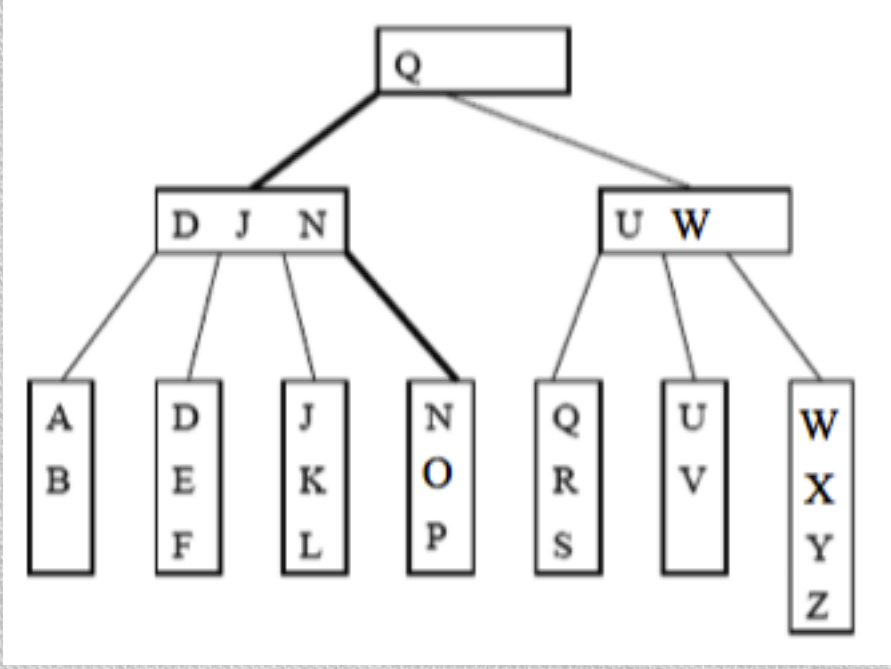
Insert Continued

- Insert X



Insert Continued

- Insert X
  - After the insert

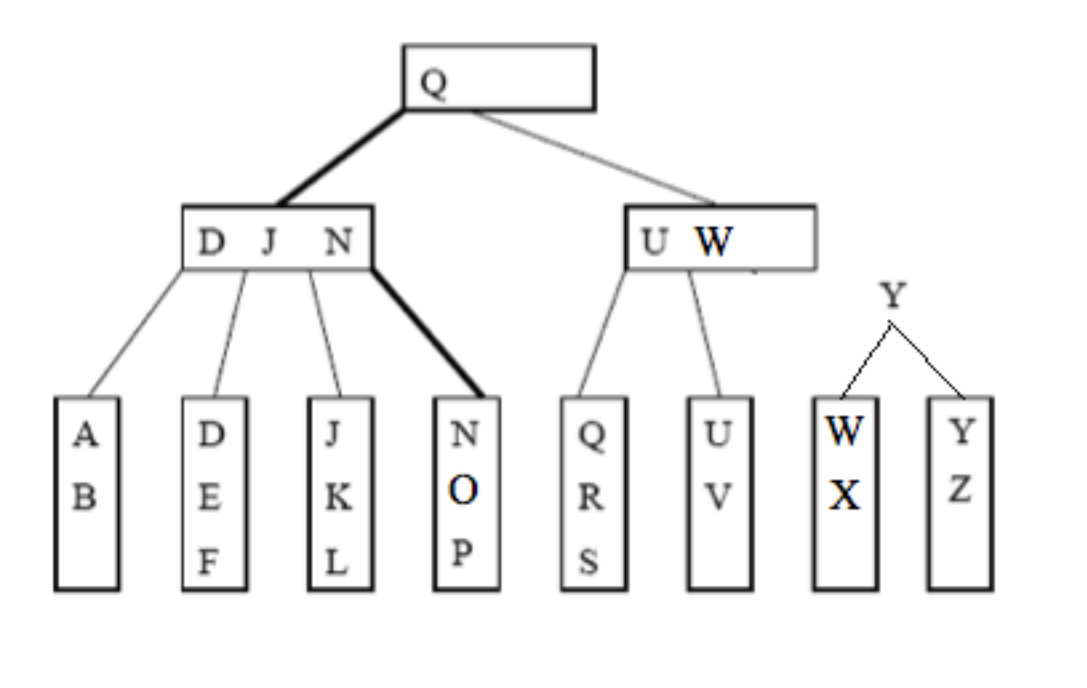


Overflowing nodes

- We split into two nodes giving half the values to each
- Choose a key for them
  - Smallest in the right
- Insert the key into the parent

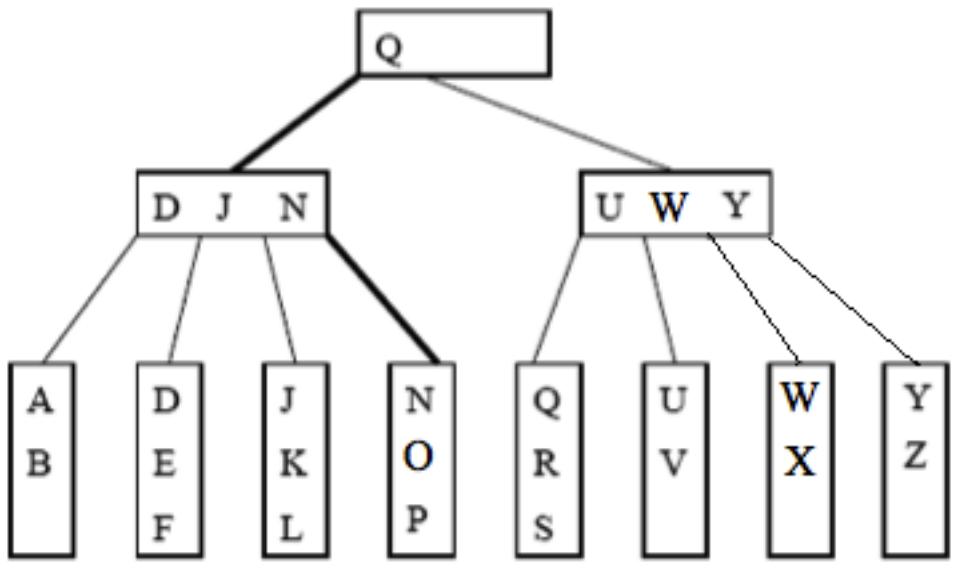
Insert Continued

- Insert X
  - Split the node, choose a key



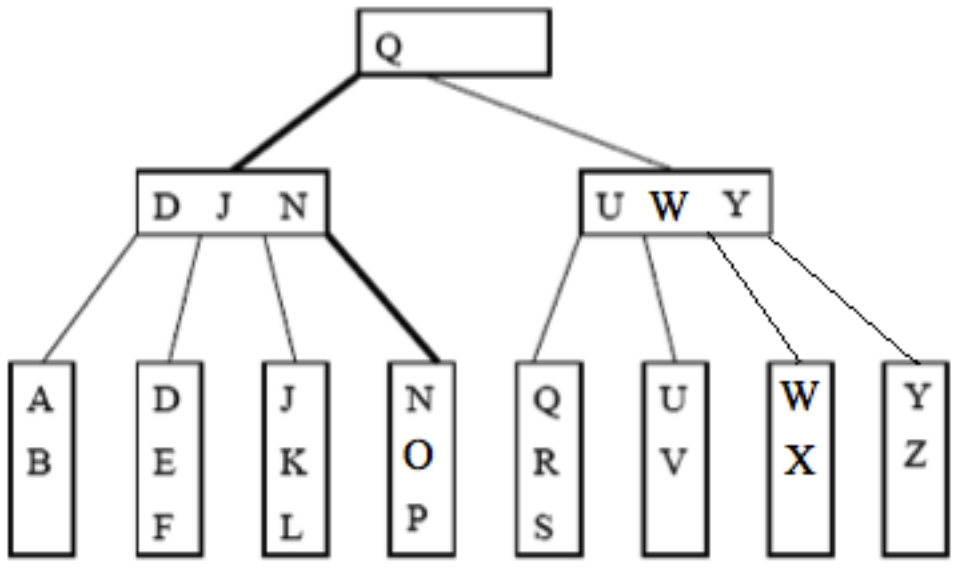
Insert Continued

- Insert X
  - Put the key in the parent



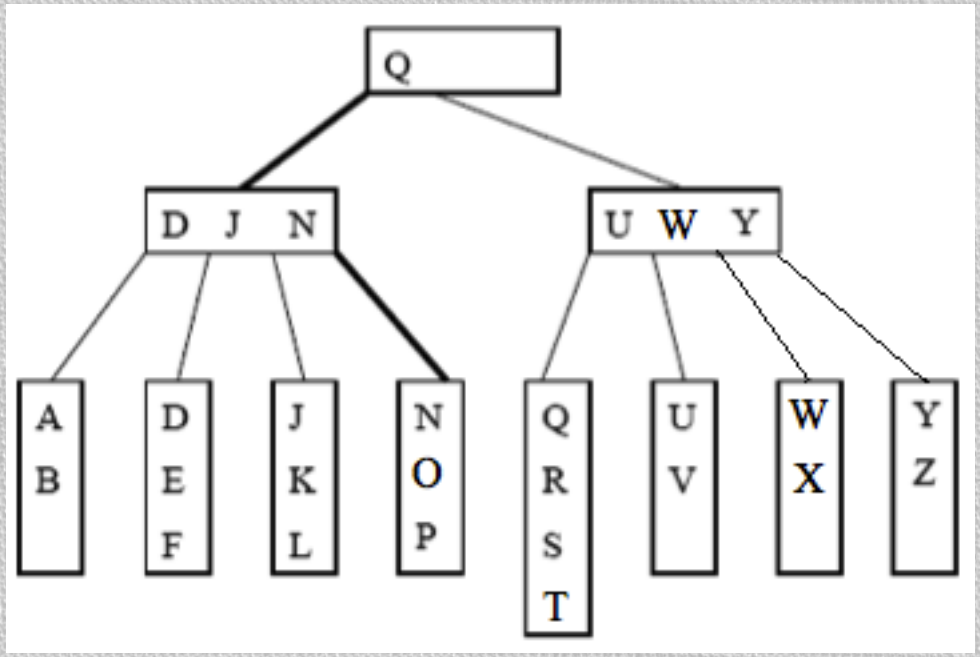
Insert

- Insert T



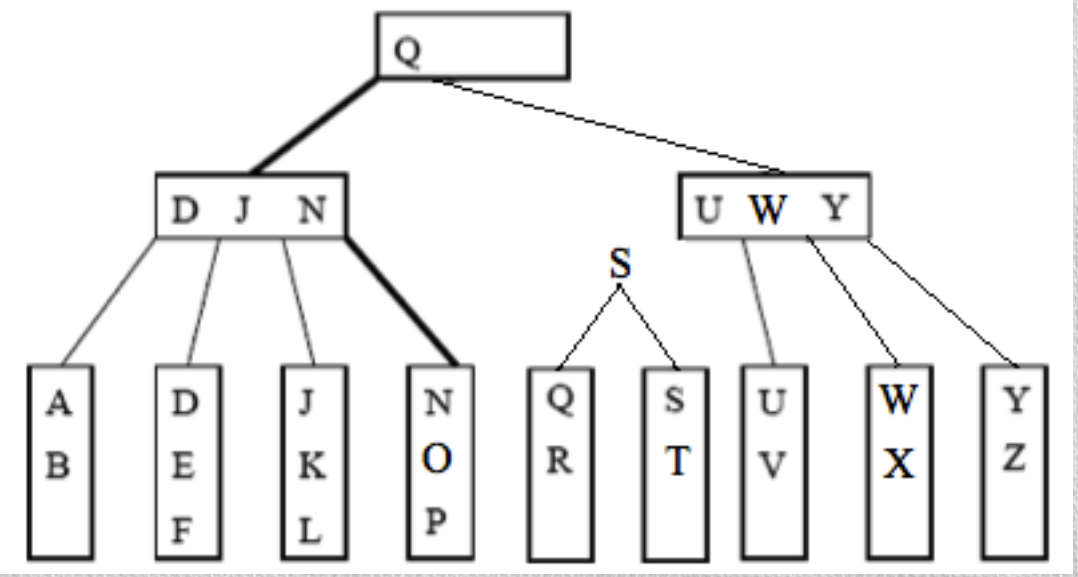
Insert Continued

- Insert T
  - After the insert



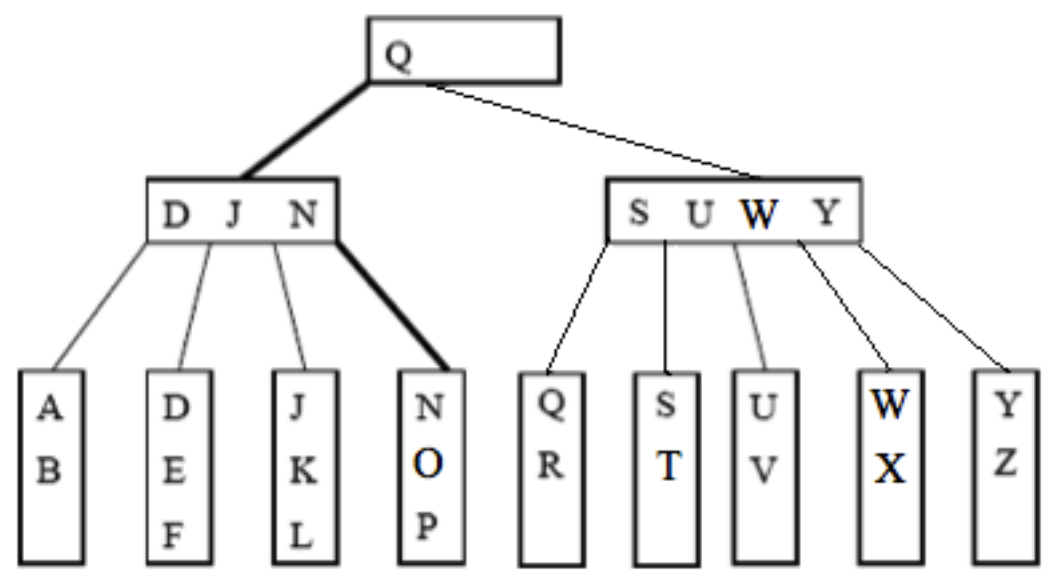
Insert Continued

- Insert T
  - Split the node, choose a key



Insert Continued

- Insert T
  - Put the key in the parent

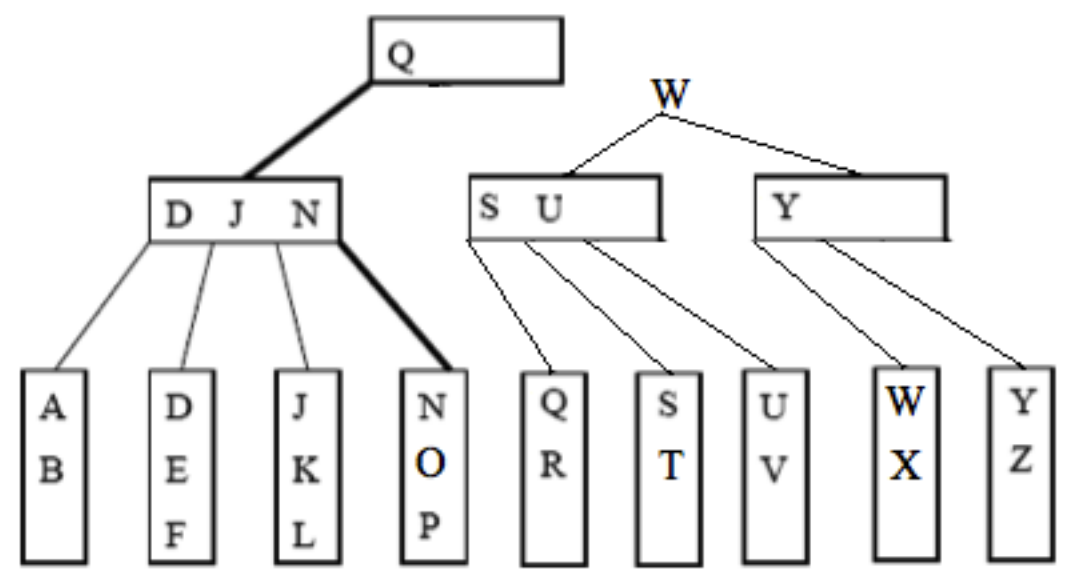


Overflowing internal nodes

- We split into two nodes giving have the values to each
- Choose a key for them
  - Smallest in the right
  - Remove this from the right node
- Insert the key into the parent

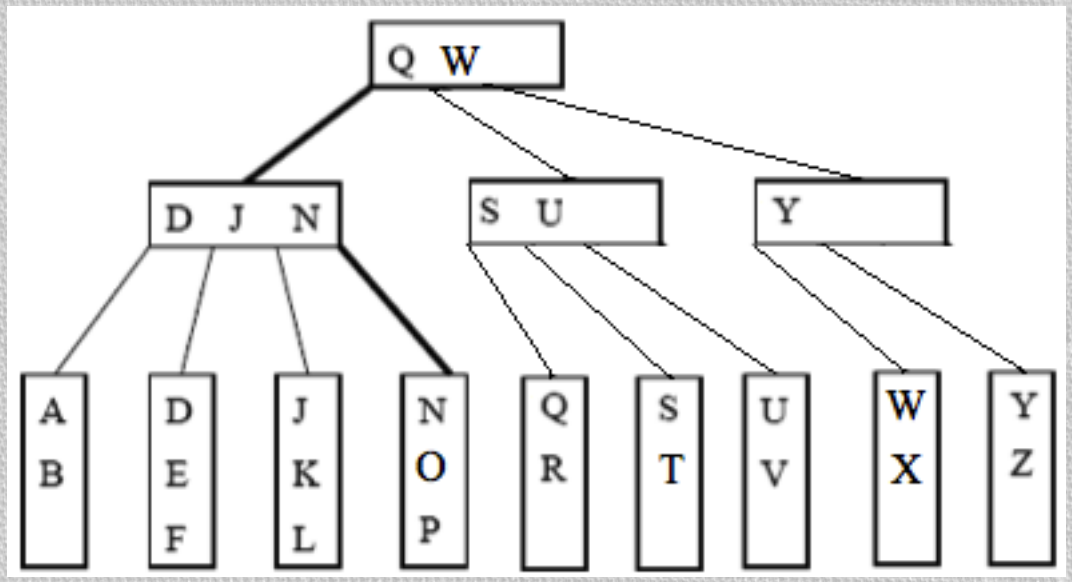
Insert Continued

- Insert T
  - Split the parent, choose a key



Insert Continued

- Insert T
  - Put the key in the parent



Overfilled nodes

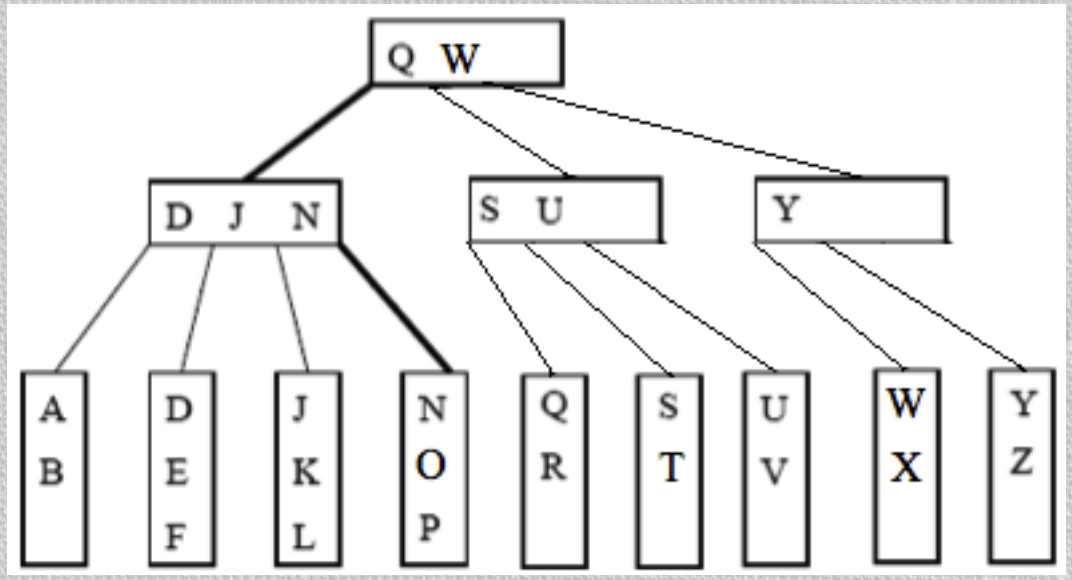
- If the root gets overfilled, we split it and add a new node on top as the new root

Delete

- Find it and delete it
- Update the keys
- Now we might have under-filled leaves
- We may have to merge them
  - If the node we merge with is full, we will just have to split them
- More practical to steal some values from a sibling
  - If the right-1 is more than half full, take the smallest value
  - If the left-1 is more than half full, take the largest one
  - If neither have enough, merge with the right

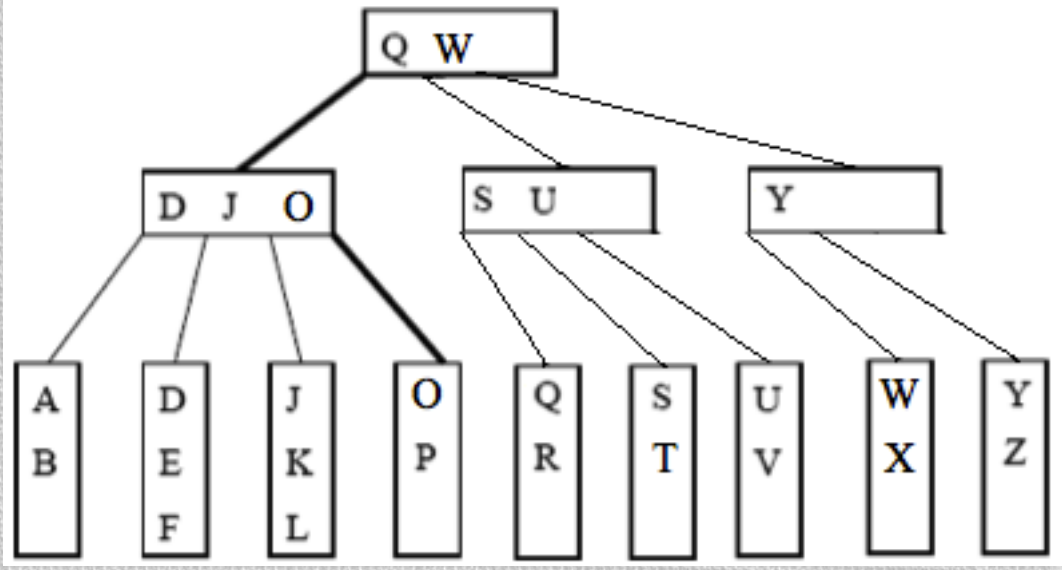
Delete Example

- Delete N



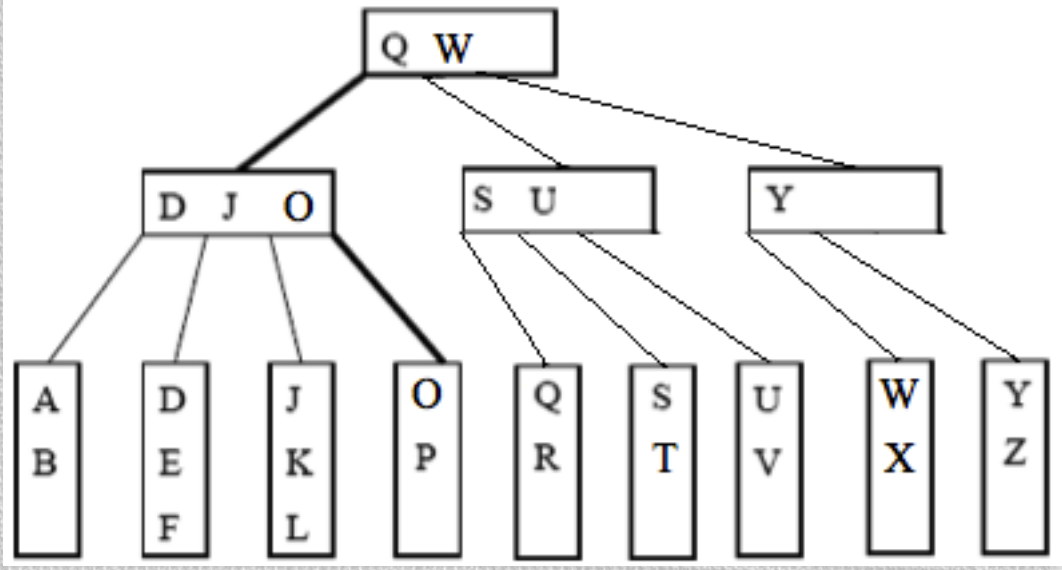
Delete Continued

- Delete N
  - After the delete, key updated



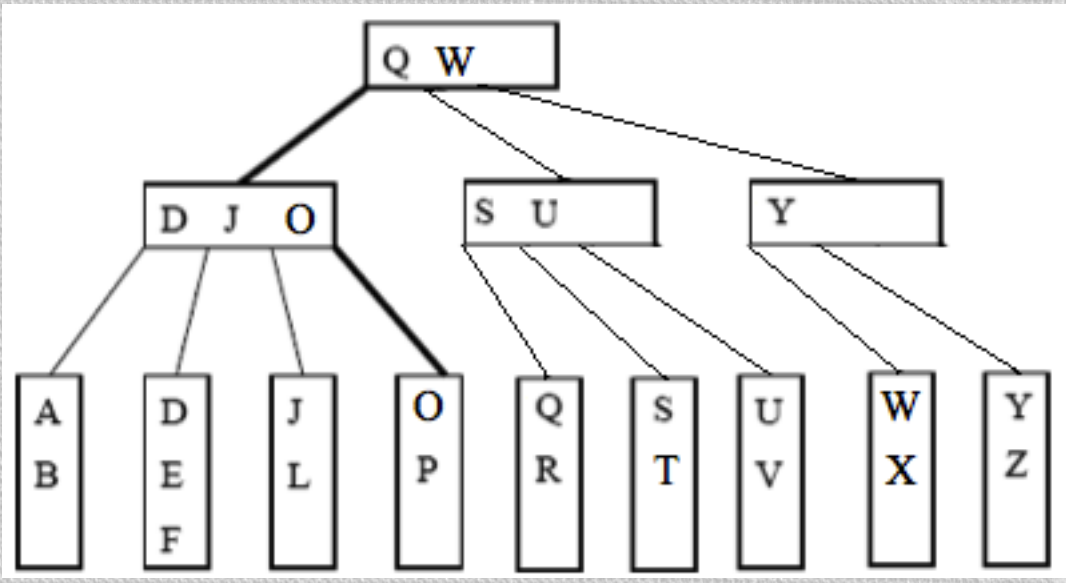
Delete

- Delete K



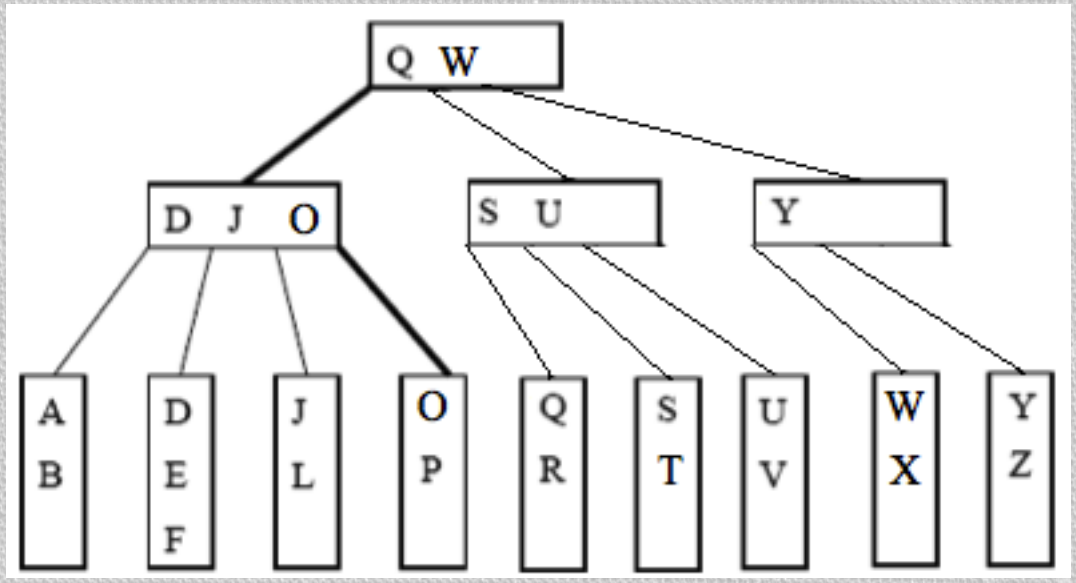
Delete Continued

- Delete K
  - After the delete, no key update needed



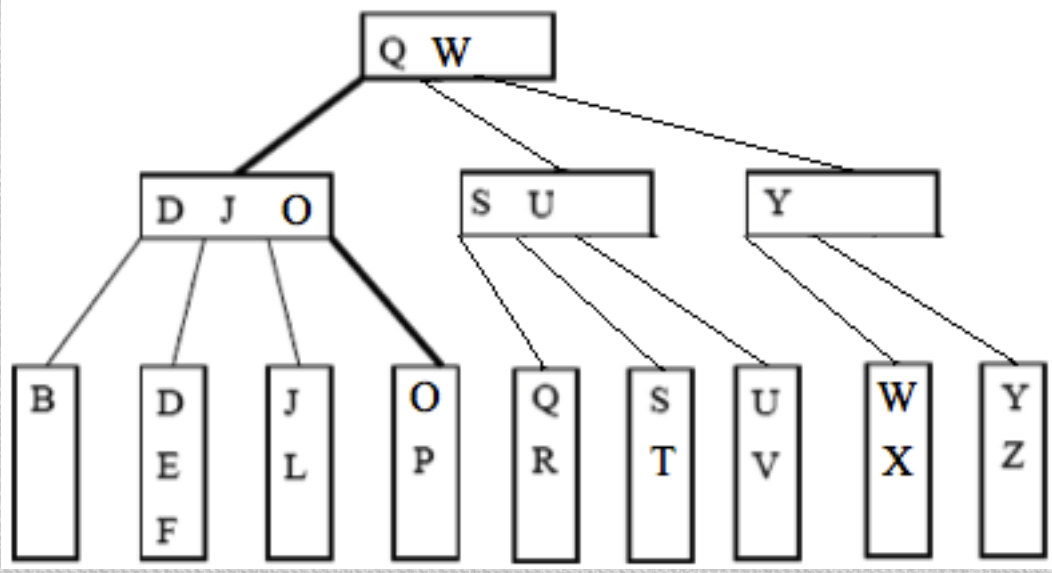
Delete

- Delete A



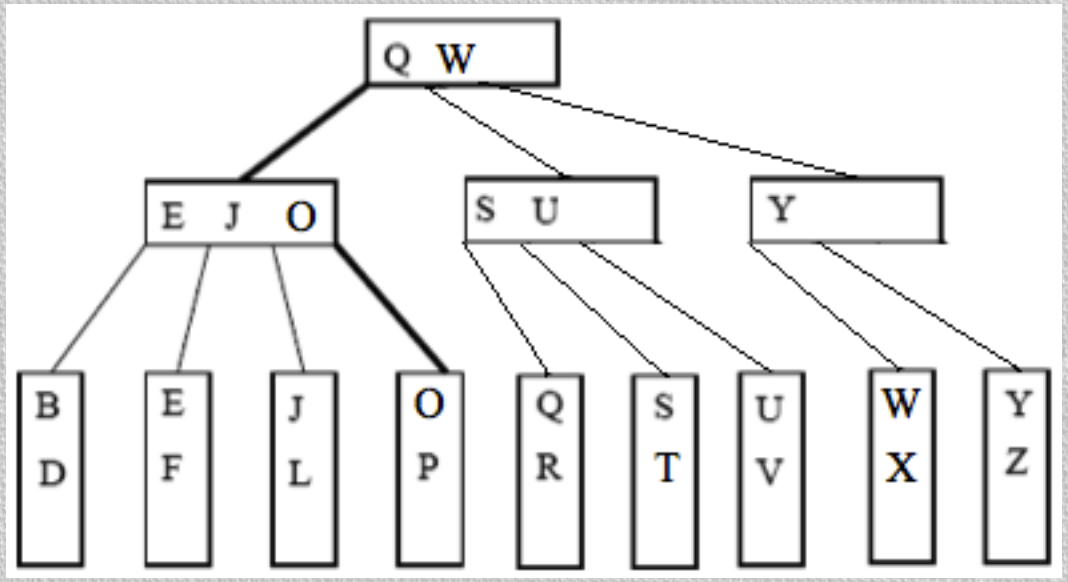
Delete Continued

- Delete A
  - The node is not full enough



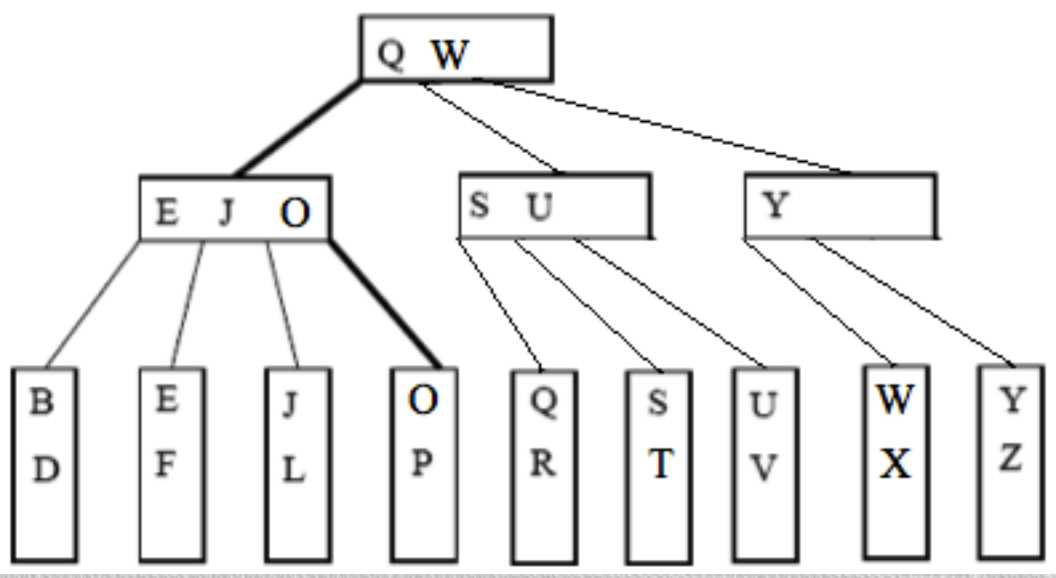
Delete Continued

- Delete A
  - After stealing the D and updating the key



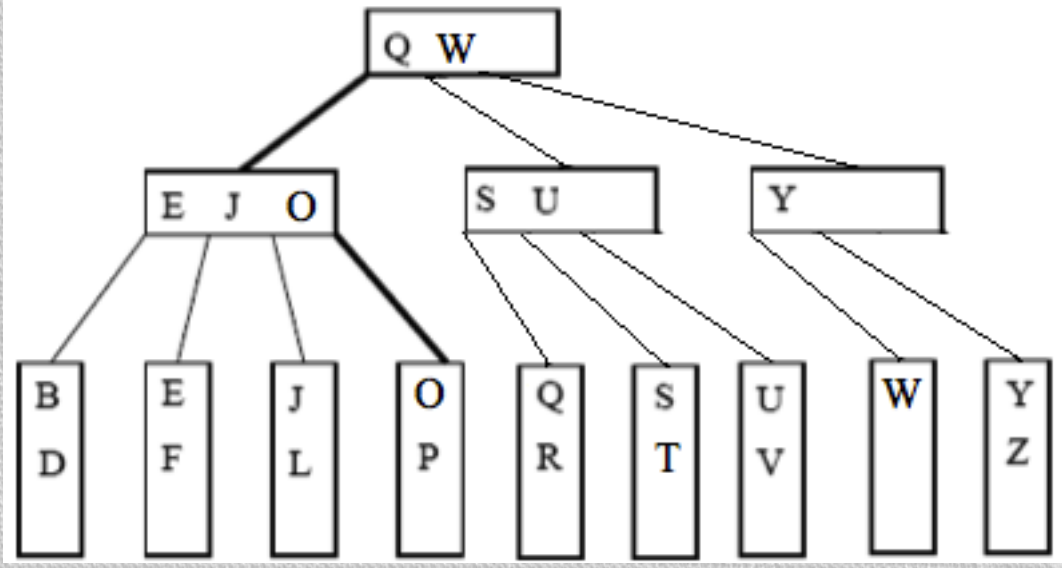
Delete

- Delete X



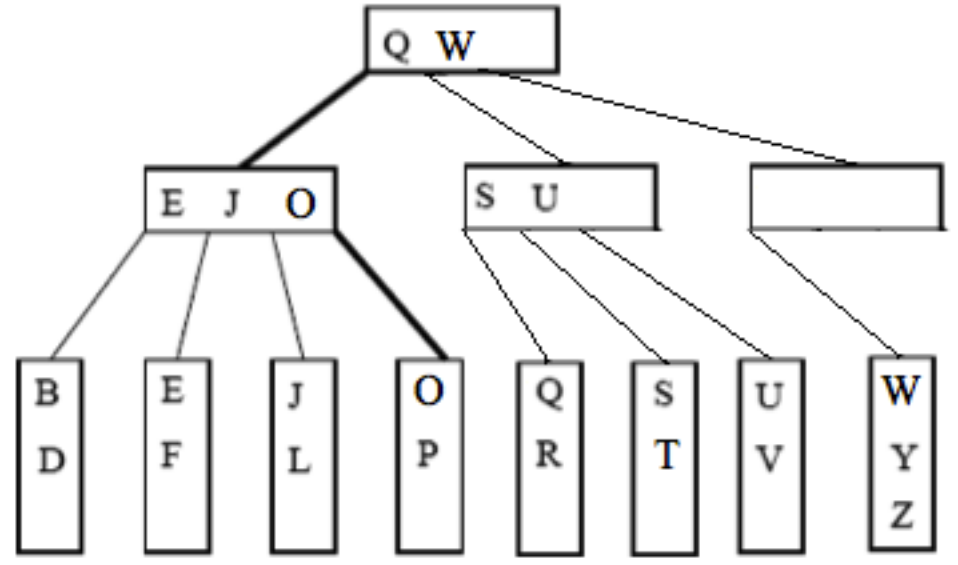
Delete Continued

- Delete X
  - The node is not full enough



Delete Continued

- Delete X
  - Since there were not enough values to fill two nodes, we merged them
  - We also updated the key

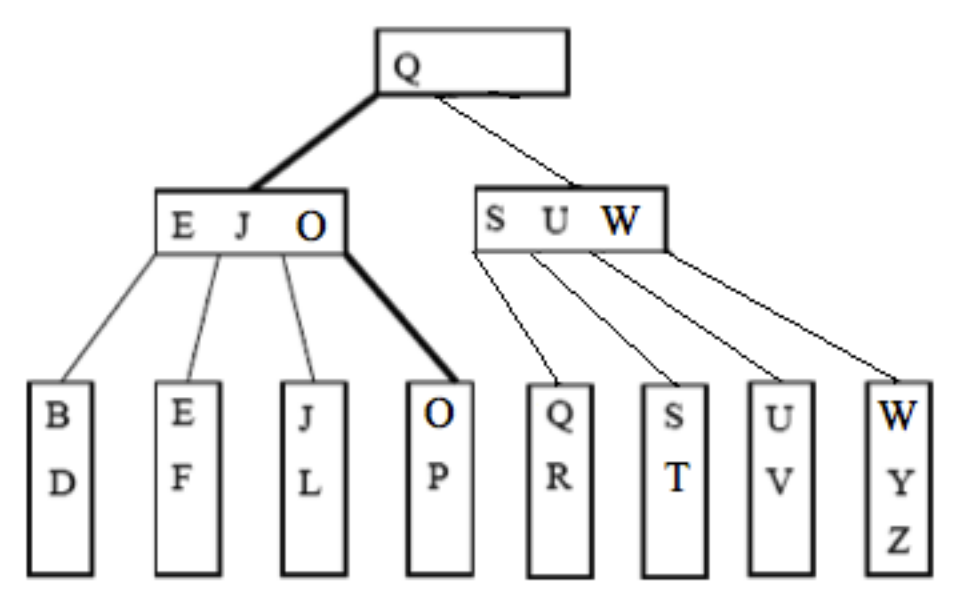


Under-filled internal nodes

- Do the same as a leaf, steal from a sibling or merge
  - Difference is we drag along the children

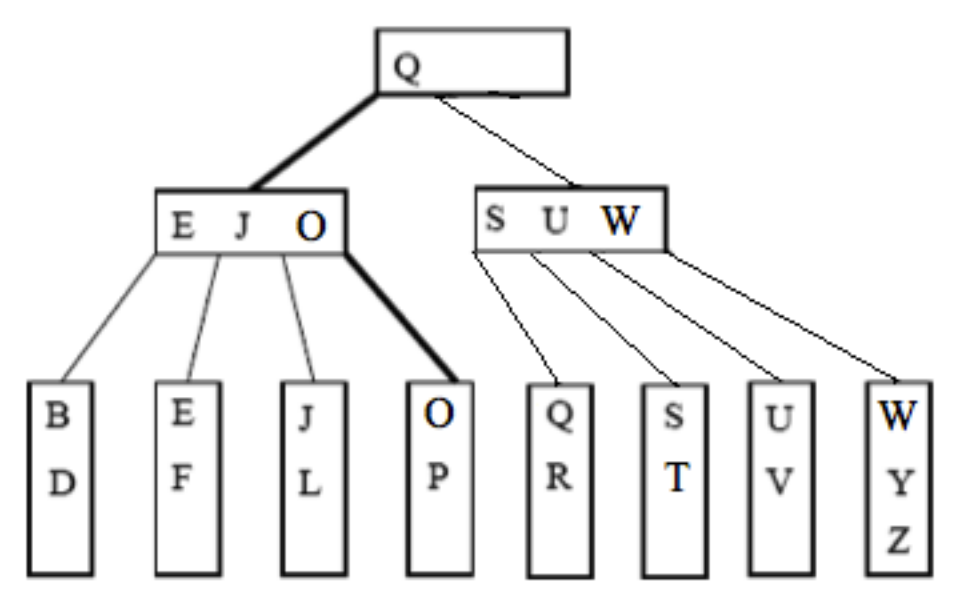
Delete Continued

- Delete X
  - After merging the internal nodes



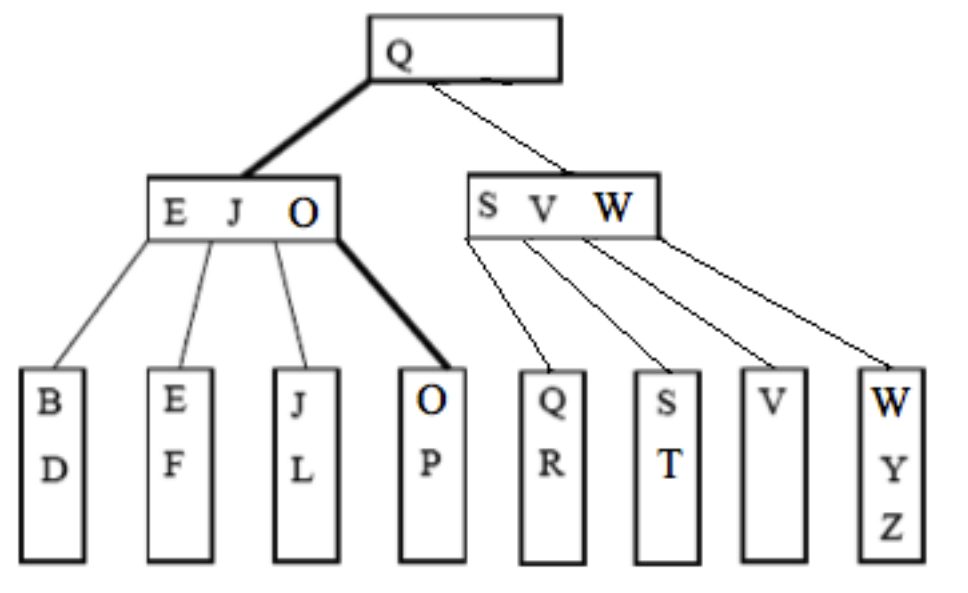
Delete

- Delete U



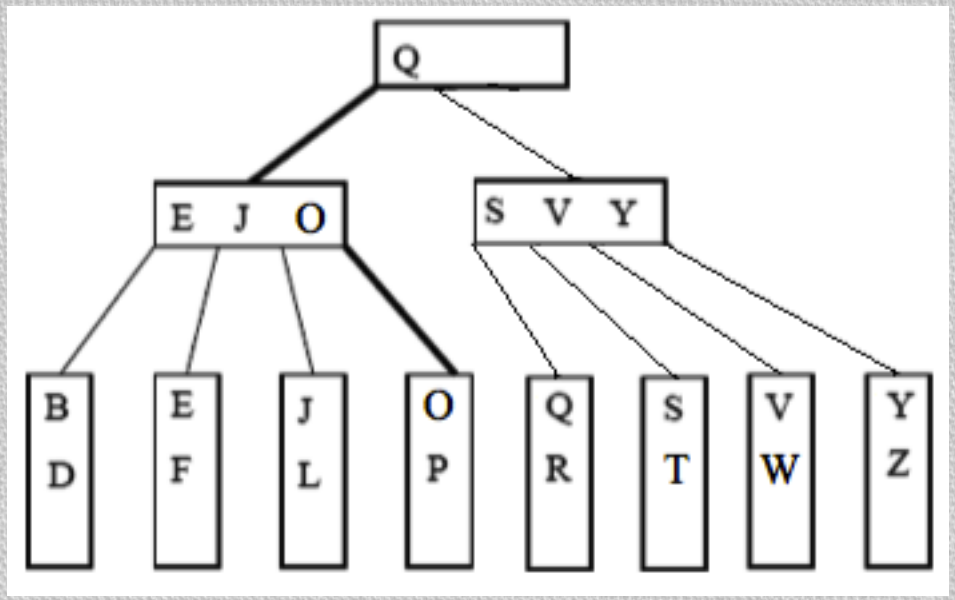
Delete Continued

- Delete U
  - After removing U and updating the parent key



Delete Continued

- Delete U
  - After stealing the W and updating the parent key



Empty Root

- If the root ever becomes empty, remove it and make the child the root