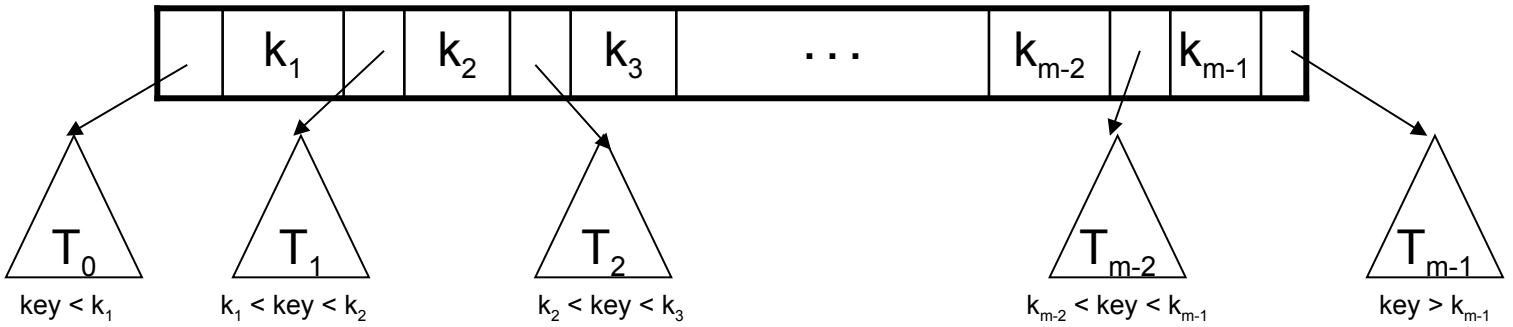
What is a Multi-way tree?

- A multi-way (or m-way) search tree of order m is a tree in which
  - Each node has at-most **m** subtrees, where the subtrees **<u>may be empty</u>**.
  - Each node consists of at least **1** and at most **m-1** distinct keys
  - The keys in each node are sorted.

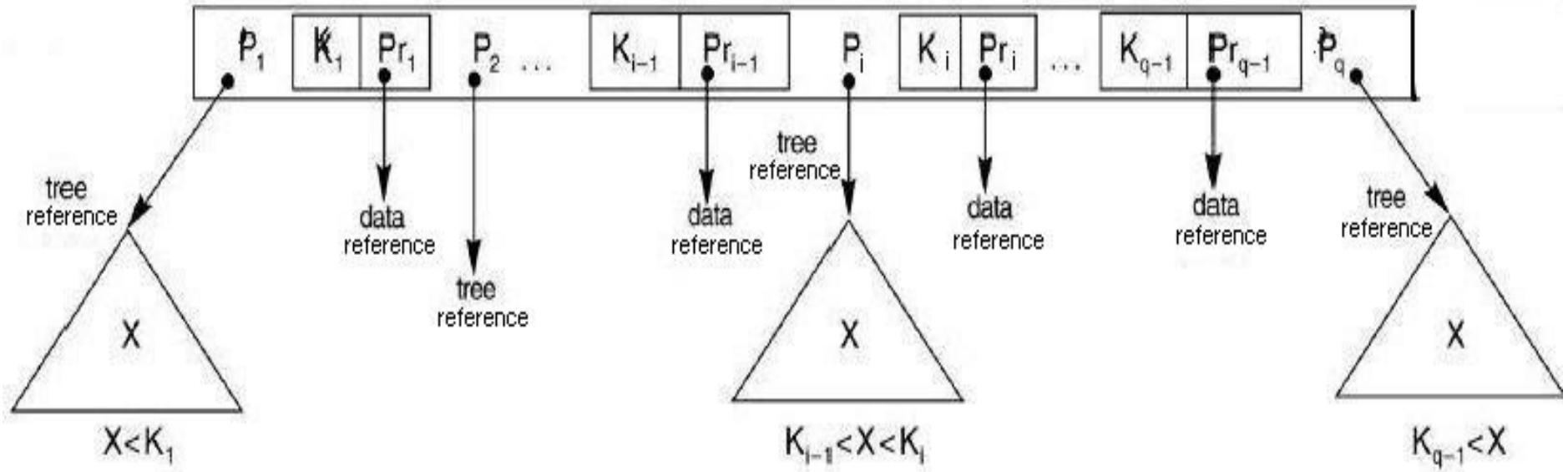


- The keys and subtrees of a non-leaf node are ordered as:

  $T_0, k_1, T_1, k_2, T_2, \ldots, k_{m-1}, T_{m-1}$ such that:

  - All keys in subtree T0 are less than k1.
  - All keys in subtree T_i , 1 <= i <= m - 2, are greater than k_i but less than k_{i+1}.
  - All keys in subtree T_{m-1} are greater than k_{m-1}

The node structure of a Multi-way tree

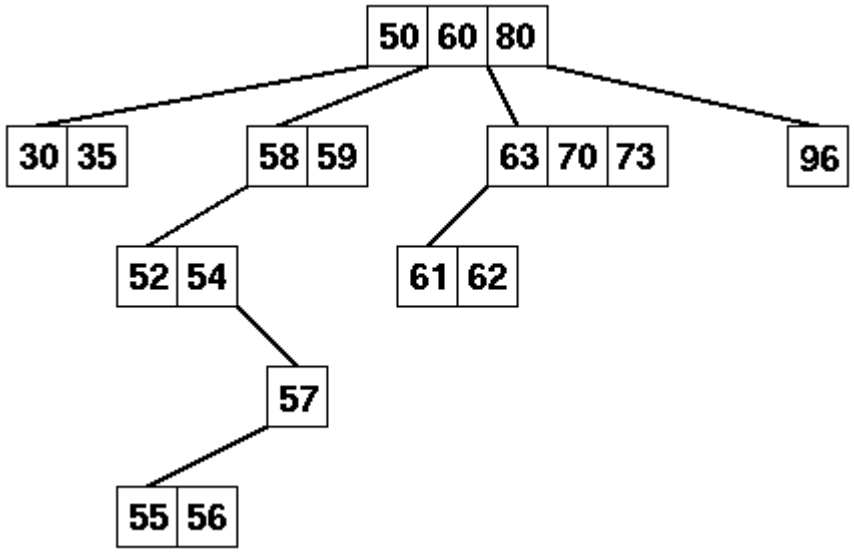


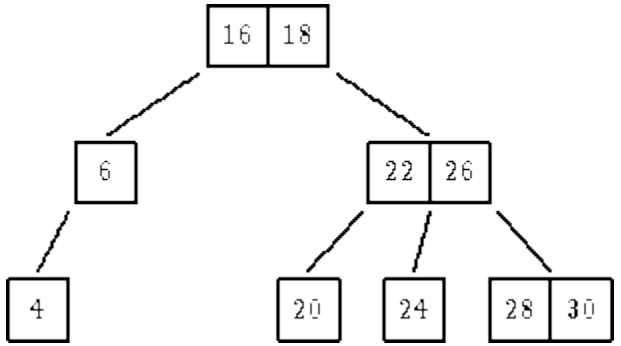
- Note:
  - Corresponding to each key there is a data reference that refers to the data record for that key in secondary memory.
  - In our representations we will omit the data references.
  - The literature contains other node representations that we will not discuss.

4000 keys, n=5

- At least 4000/(5-1) nodes (pages)
  - 1st level (root): 1 node, 4 keys, 5 sub-trees +
  - 2nd level: 5 nodes, 20 keys, 25 sub-trees +
  - 3rd level: 25 nodes, 100 keys, 125 sub-trees +
  - 4th level: 125 nodes, 500 keys,525 sub-trees +
  - 5th level: 525 nodes, 2100 keys, 2625 sub-tress +
  - 6th level: 2625 nodes, 10500 keys, ….
  - tree height = 6 (including root)

Examples of Multi-way Trees





- Note: In a multiway tree:
  - The leaf nodes need not be at the same level.
  - A non-leaf node with **n** keys may contain less than **n + 1** non-empty subtrees.

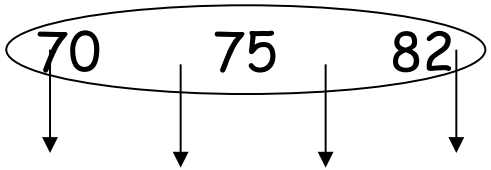
Operations on MSTs

- Search: returns pointer to node containing the key and position of key in the node
- Insert new key if not already in the tree
- Delete existing key

Insertions

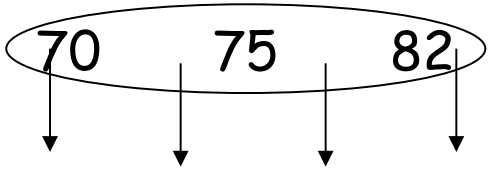
- Search & insert key it if not there
  a) the tree grows at the leafs => the tree becomes imbalanced => not equal number of disk accesses for every key
  b) the shape of the tree depends on the insertion sequence
  c) low storage utilization, many non-full nodes

insert: 70 75 82 77 71 73 84 86 87 85



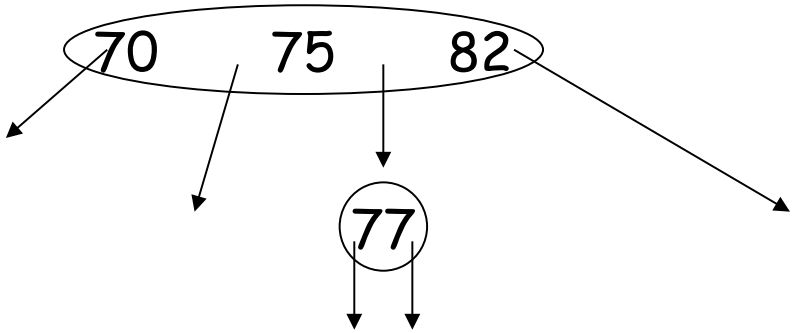
n = 4

max 3 keys/node

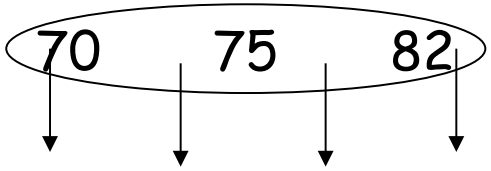


n = 4

max 3 keys/node

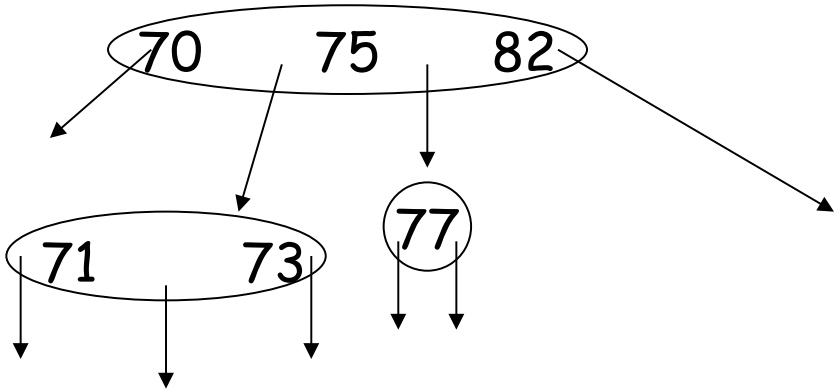


insert: 70 75 82 77 71 73 84 86 87 85

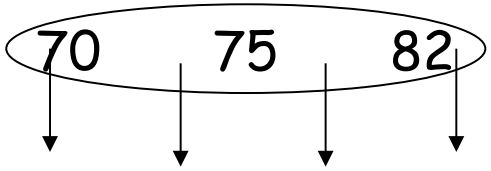


n = 4

max 3 keys/node

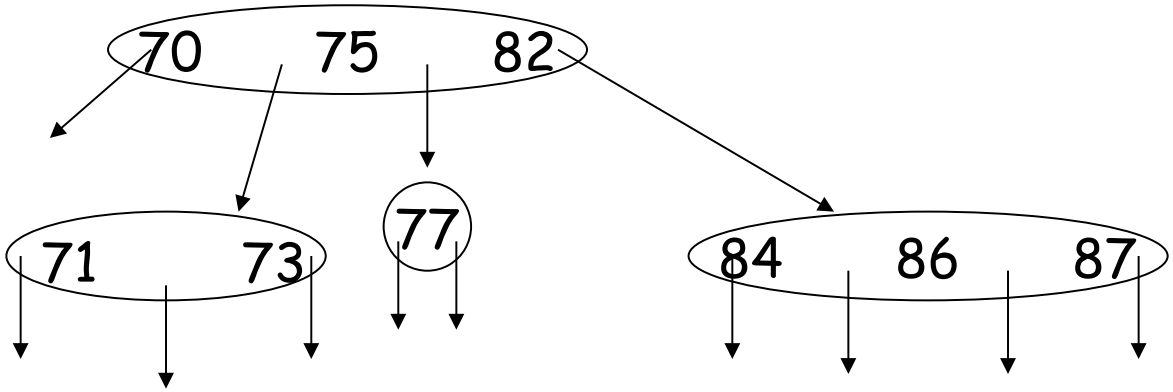


insert: 70 75 82 77 71 73 84 86 87 85

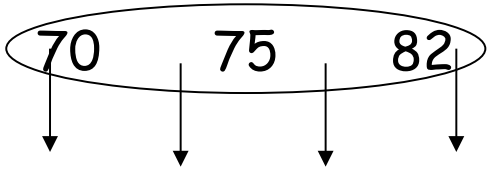


n = 4

max 3 keys/node

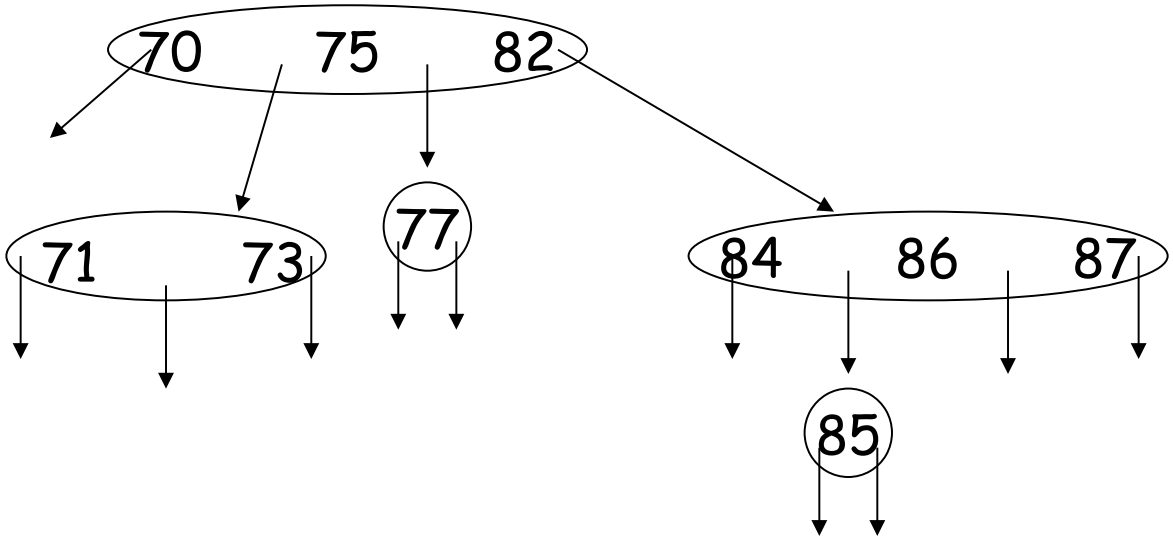


insert: 70 75 82 77 71 73 84 86 87 85



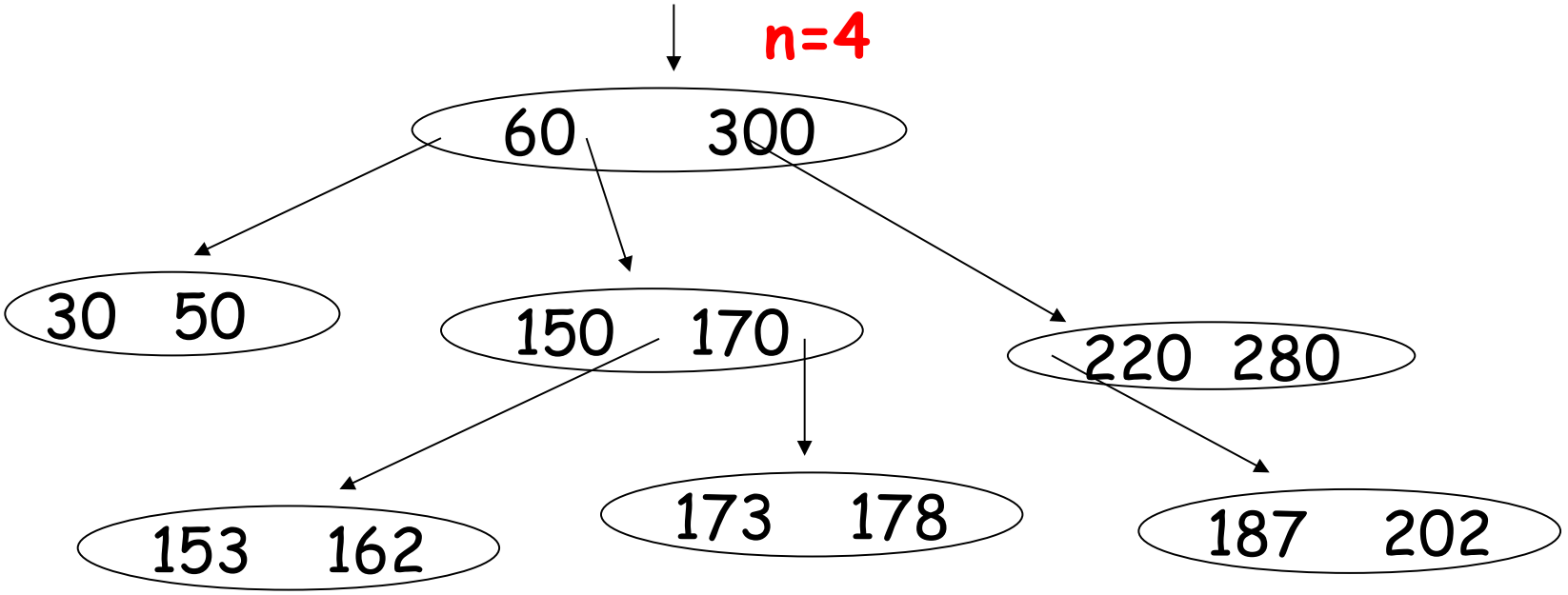
n = 4

max 3 keys/node

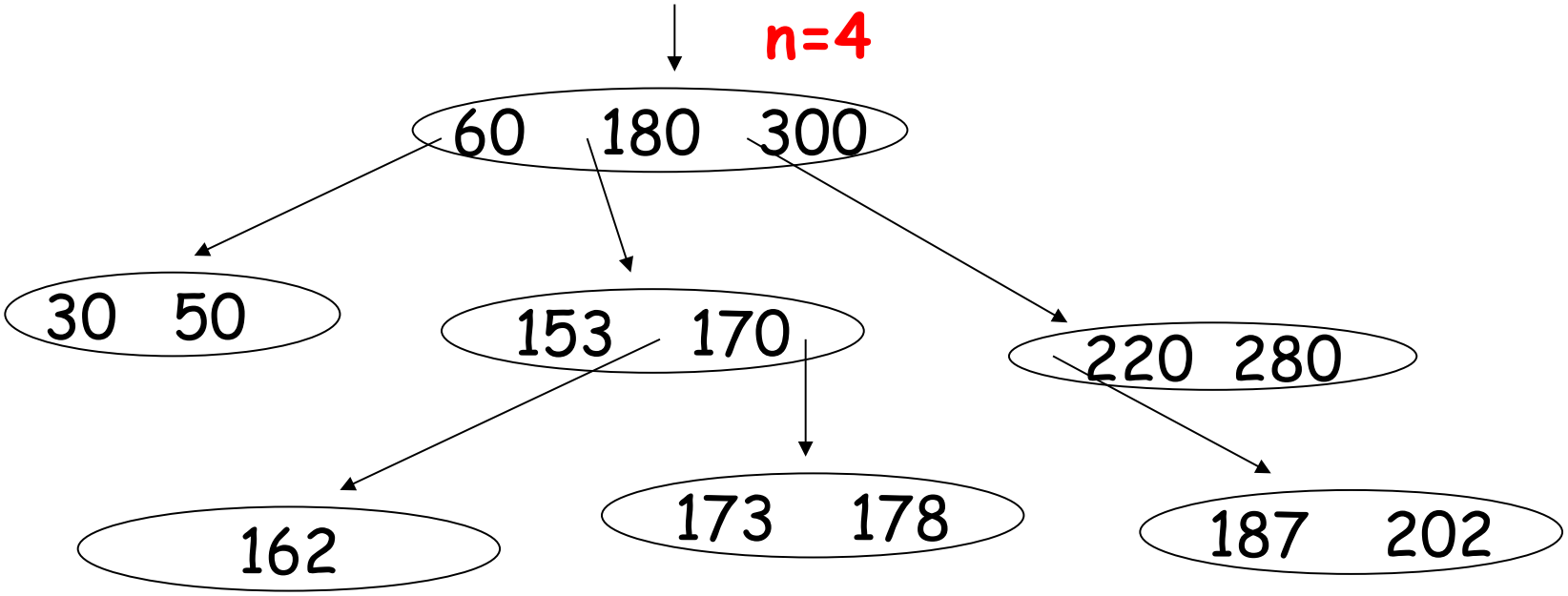


MST Deletions

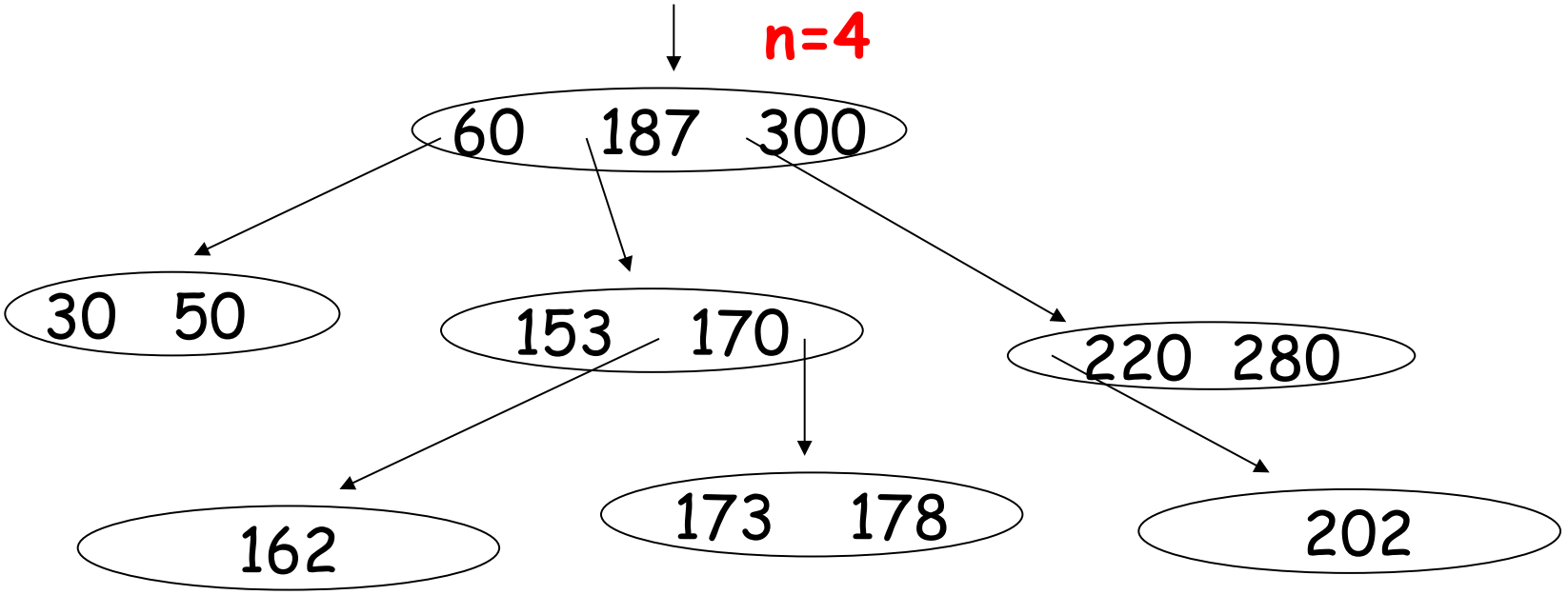
- Find and delete a key from a node
  - free the node if empty
- If the key has right or/and left sub-trees
  => find its successor or predecessor
  - min value of right subtree or max value of left subtree
- put this in the position of the key and delete the successor or predecessor
- The tree becomes imbalanced



delete 150, 180



delete 150, 180



delete 150, 180

What is a B-Tree?

- A B-tree of order m (or branching factor m), where m > 2, is either an empty tree or a multiway search tree with the following properties:
  - The root is either a leaf or it has at least two non-empty subtrees and at most m non-empty subtrees.
  - Each non-leaf node, other than the root, has at least $\lceil m/2 \rceil$ non-empty subtrees and at most m non-empty subtrees. (Note: $\lceil x \rceil$ is the lowest integer > x).
  - The number of keys in each non-leaf node is one less than the number of non-empty subtrees for that node.
  - All leaf nodes are at the same level; that is the tree is perfectly balanced.

What is a B-tree? (cont'd)

For a non-empty B-tree of order m:

This may be zero, if the node is a leaf as well

	Root node	Non-root node
Minimum number of keys	**1**	**$\lceil m/2 \rceil - 1$**
Minimum number of non-empty subtrees	2	$\lceil m/2 \rceil$
Maximum number of keys	**m - 1**	**m – 1**
Maximum number of non-empty subtrees	m	m

These will be zero if the node is a leaf as well

Insertion in B-Trees

- **OVERFLOW CONDITION:**

  A root-node or a non-root node of a B-tree of order m overflows if, after a key insertion, it contains m keys.

- **Insertion algorithm:**

  If a node overflows, split it into two, propagate the "middle" key to the parent of the node. If the parent overflows the process propagates upward. If the node has no parent, create a new root node.

- **Note: Insertion of a key always <u>starts</u> at a leaf node.**

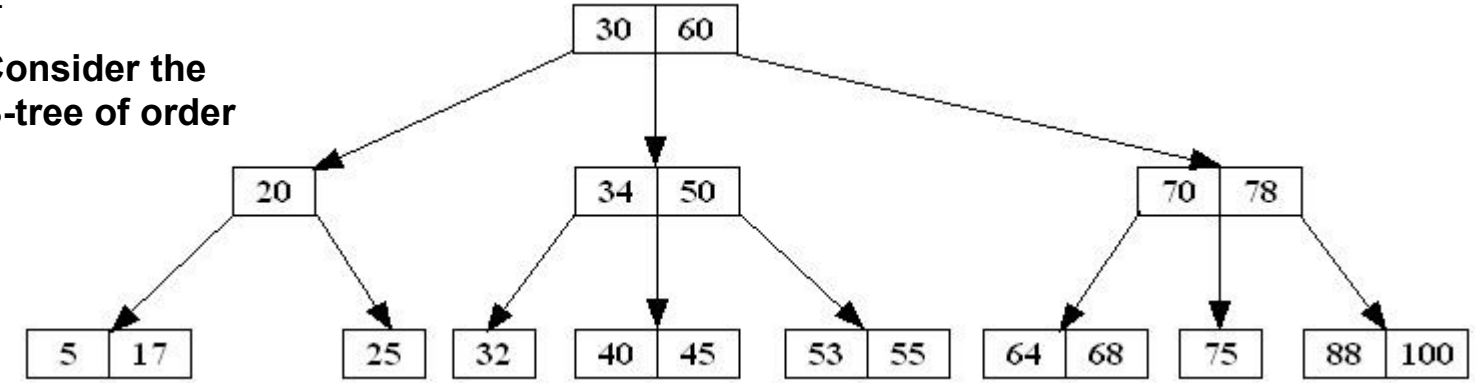
Deletion in B-Tree

- **Like insertion, deletion must be on a leaf node. If the key to be deleted is not in a leaf, swap it with either its successor or predecessor (each will be in a leaf).**

- The successor of a key **k** is the smallest key greater than **k**.
- The predecessor of a key **k** is the largest key smaller than **k**.

- IN A B-TREE THE SUCCESSOR AND PREDECESSOR, IF ANY, OF ANY KEY IS IN A LEAF NODE

Example: Consider the following B-tree of order 3:



key	predecessor	successor
20	17	25
30	25	32
34	32	40
50	45	53
60	55	64
70	68	75
78	75	88

Deletion in B-Tree

- **UNDERFLOW CONDITION**
- A non-root node of a B-tree of order m underflows if, after a key deletion, it contains $\lceil \mathbf{m / 2} \rceil$ **- 2** keys

- The root node does not underflow. If it contains only one key and this key is deleted, the tree becomes empty.

Deletion in B-Tree

- **Deletion algorithm:**

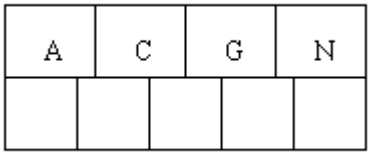
  If a node underflows, **rotate** the appropriate key from the **adjacent** right- or left-sibling if the sibling contains at least $\lceil \mathbf{m / 2} \rceil$ keys; otherwise perform a **merging**.

⇒ A key rotation must always be attempted before a merging

- There are **five** deletion cases:
  1. The leaf does not underflow.
  2. The leaf underflows and the adjacent right sibling has at least $\lceil m / 2 \rceil$ keys.
     perform a left key-rotation
  3. The leaf underflows and the adjacent left sibling has at least $\lceil m / 2 \rceil$ keys.
     perform a right key-rotation
  4. The leaf underflows and each of the adjacent right sibling and the adjacent left sibling has at least $\lceil m / 2 \rceil$ keys.
     perform either a left or a right key-rotation
  5. The leaf underflows and each adjacent sibling has $\lceil m / 2 \rceil - 1$ keys.
     perform a merging

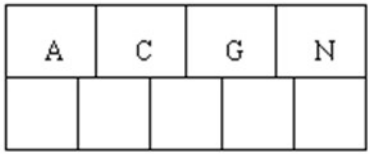
B Trees

- Insert the following letters into what is originally an empty B-tree of order 5:
- C N G A H E K Q M F W L T Z D P R X Y S
- Order 5 means that a node can have a maximum of 5 children and 4 keys.
- All nodes other than the root must have a minimum of 2 keys.
- The first 4 letters get inserted into the same node, resulting in this picture:



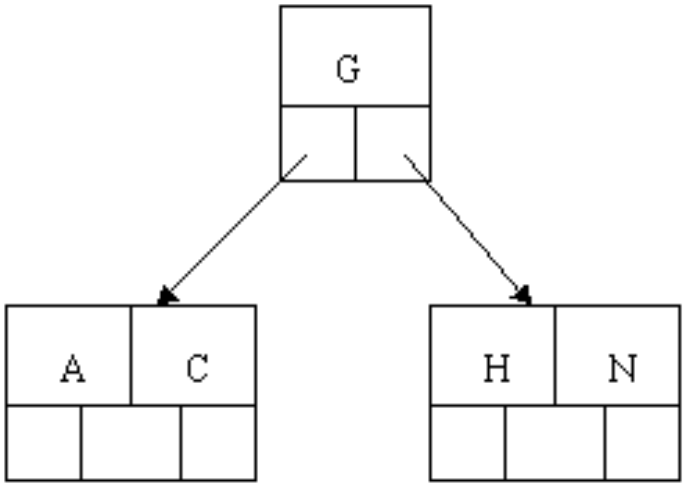
B Trees

- C N G A H E K Q M F W L T Z D P R X Y S
- insert the H - no room in this node,



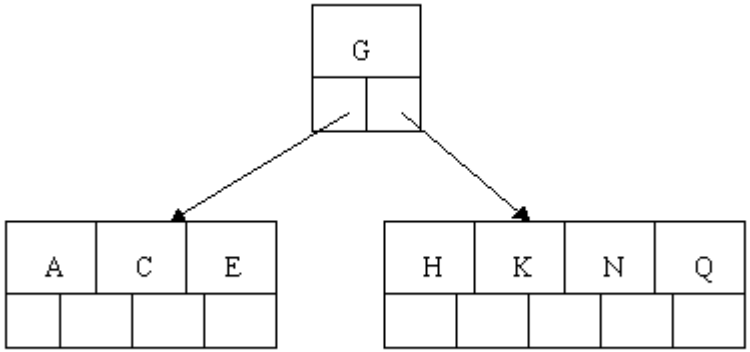
B Trees

- C N G A H E K Q M F W L T Z D P R X Y S
- insert the H - no room in this node,
- so split it into 2 nodes, moving the median item G up into a new root node.



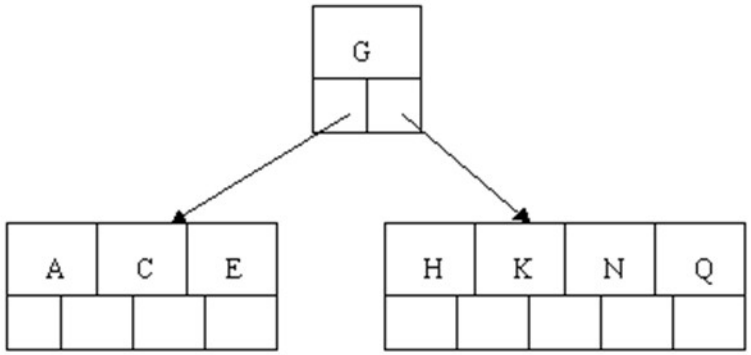
B Trees

- C N G A H E K Q M F W L T Z D P R X Y S
- Inserting E, K, and Q proceeds without requiring any splits:



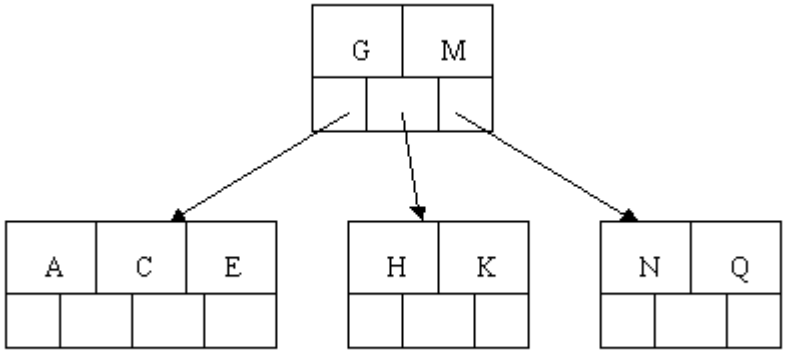
B Trees

- C N G A H E K Q M F W L T Z D P R X Y S
- Insert M



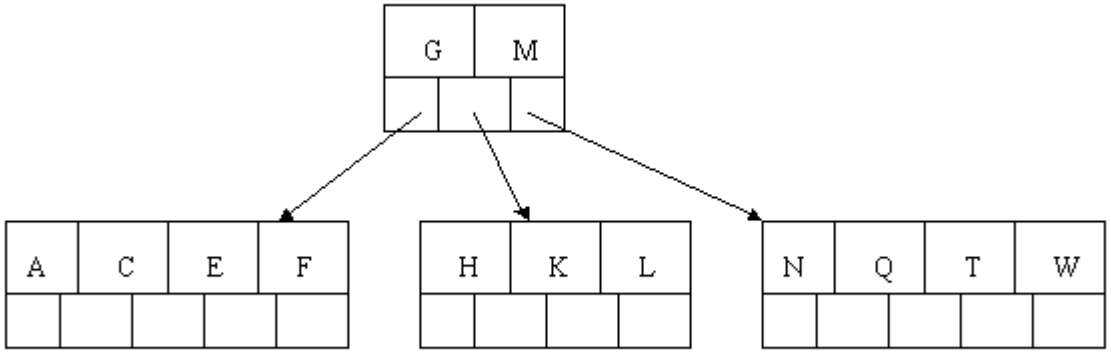
B Trees

- C N G A H E K Q M F W L T Z D P R X Y S
- Inserting M requires a split. Note that M happens to be the median key and so is moved up into the parent node.



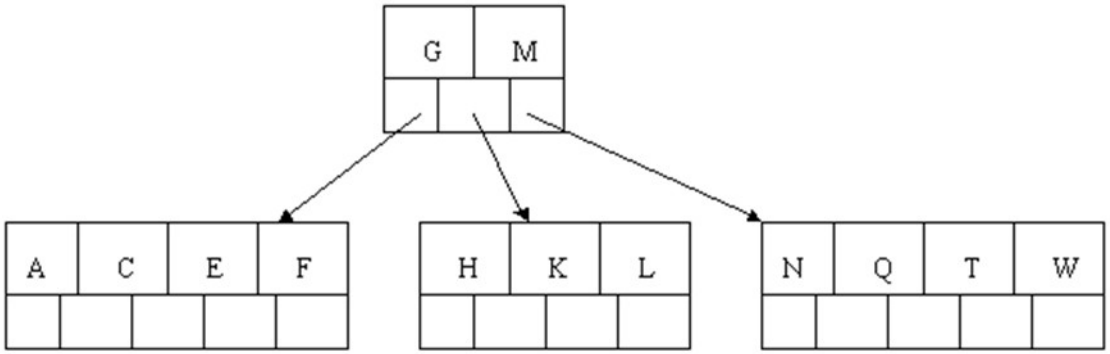
B Trees

- C N G A H E K Q M F W L T Z D P R X Y S
- The letters F, W, L, and T are then added without needing any split.



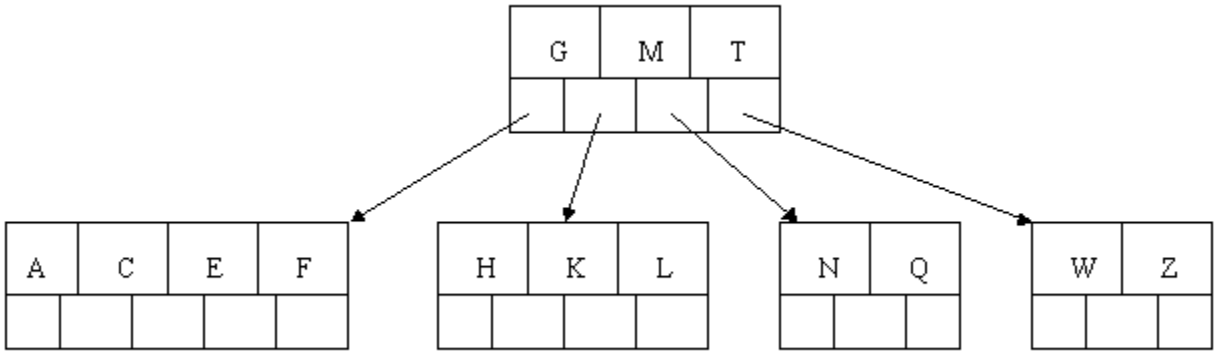
B Trees

- C N G A H E K Q M F W L T Z D P R X Y S
- Add Z



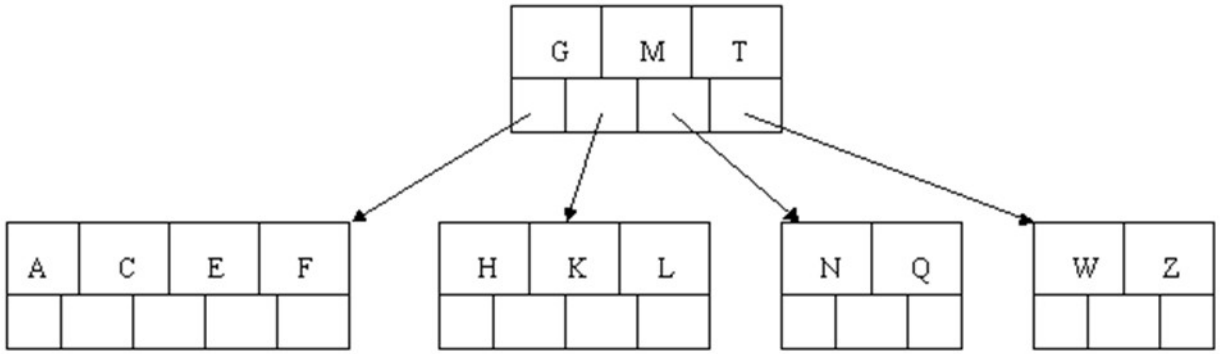
B Trees

- C N G A H E K Q M F W L T Z D P R X Y S
- When Z is added, the rightmost leaf must be split. The median item T is moved up into the parent node. Note that by moving up the median key, the tree is kept fairly balanced, with 2 keys in each of the resulting nodes.



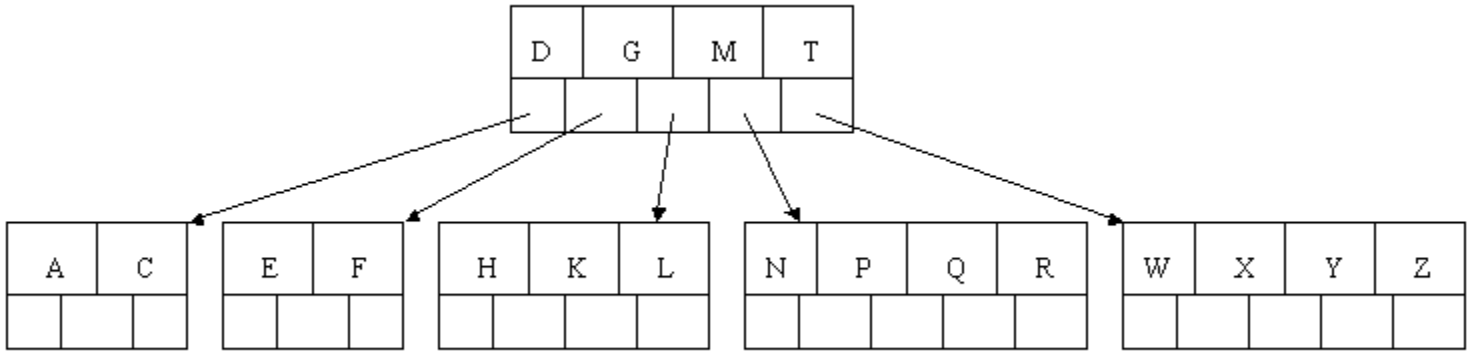
B Trees

- C N G A H E K Q M F W L T Z D P R X Y S
- insert D



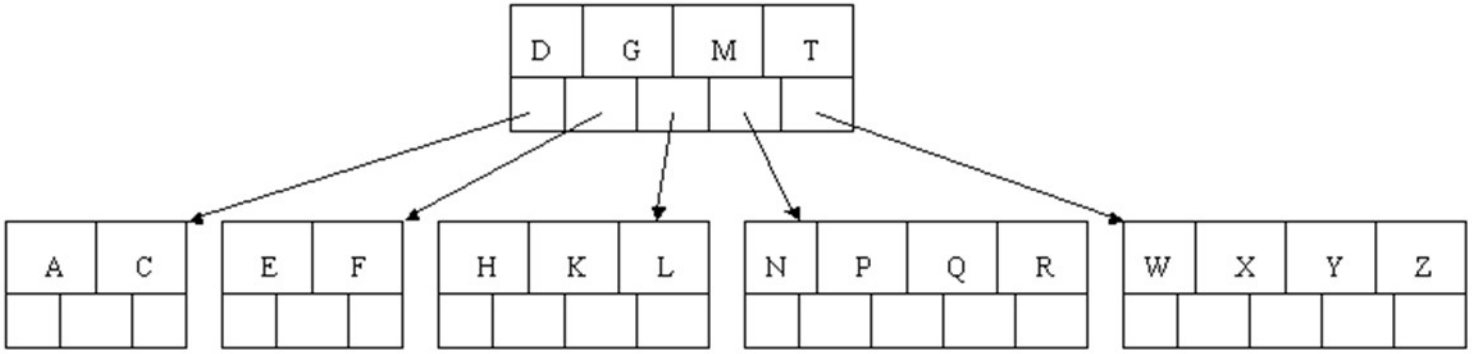
B Trees

- C N G A H E K Q M F W L T Z D P R X Y S
- The insertion of D causes the leftmost leaf to be split. D happens to be the median key and so is the one moved up into the parent node. The letters P, R, X, and Y are then added without any need of splitting:



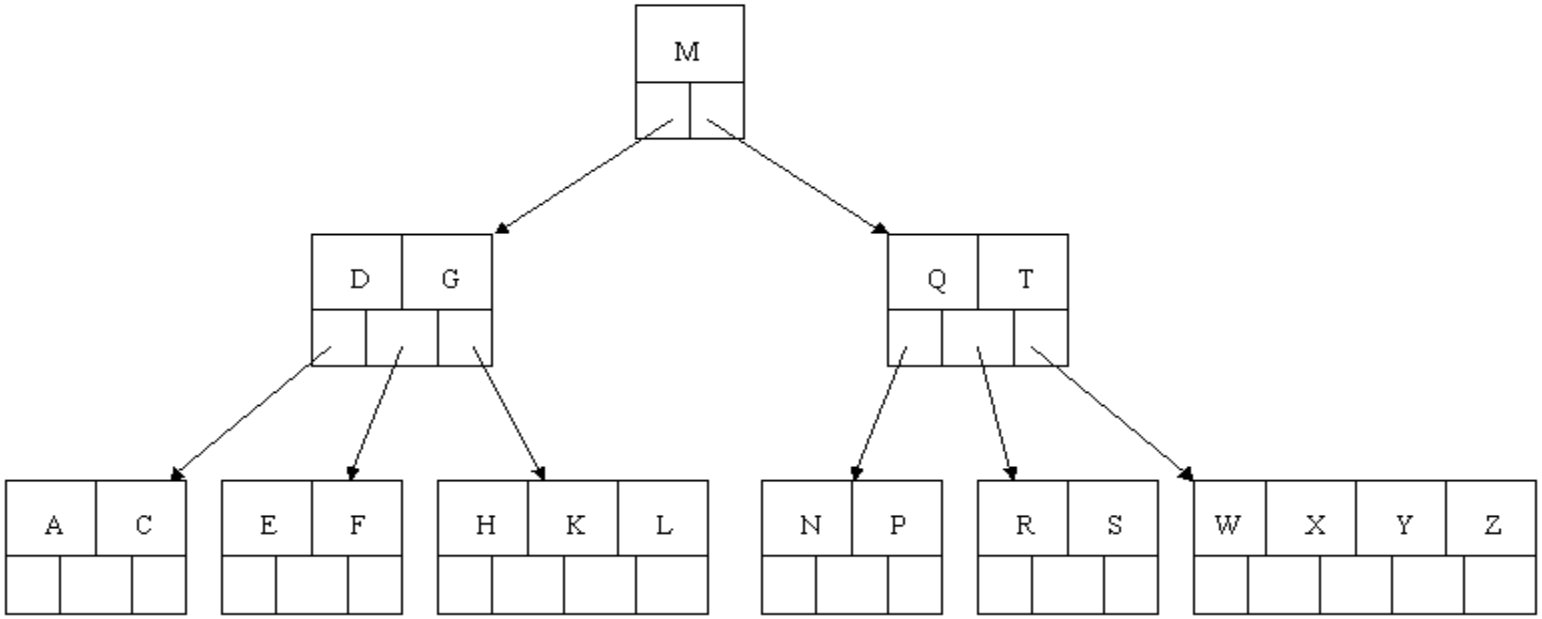
B Trees

- C N G A H E K Q M F W L T Z D P R X Y S
- Add S



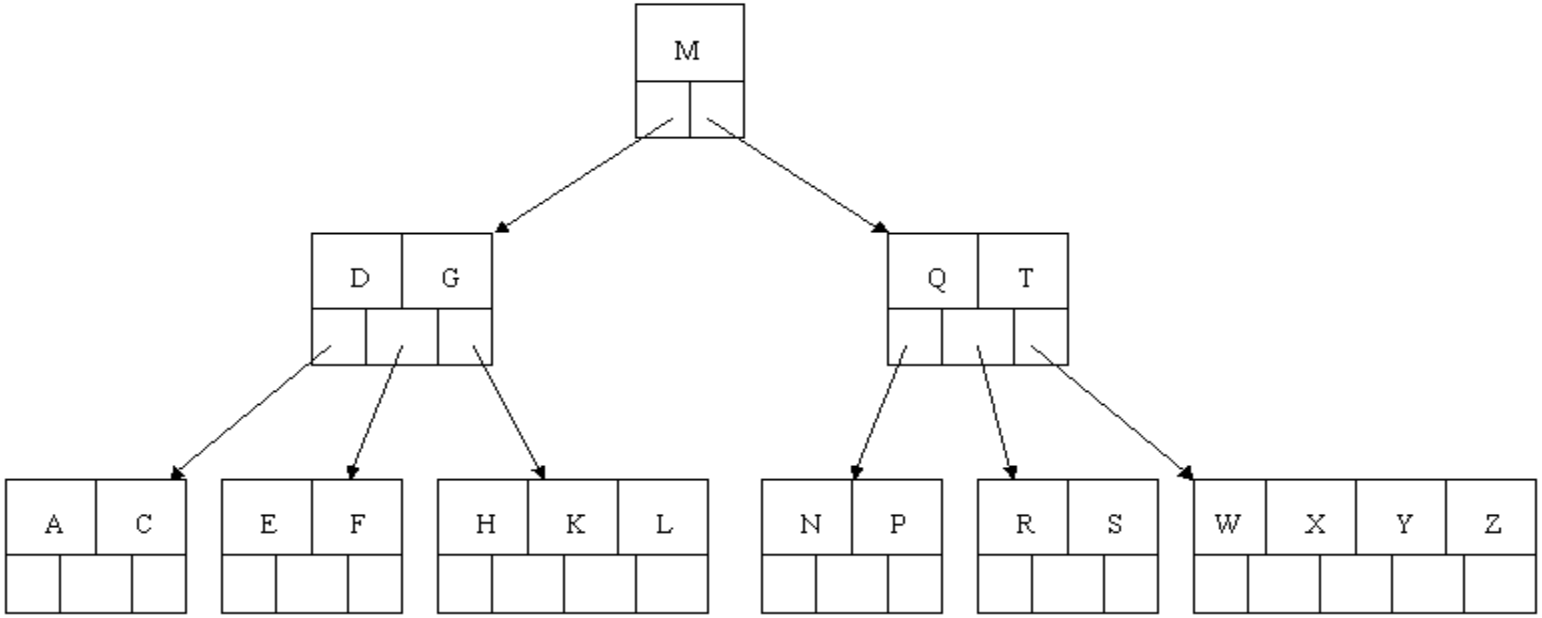
B Trees

- C N G A H E K Q M F W L T Z D P R X Y S
- when S is added, the node with N, P, Q, and R splits, sending the median Q up to the parent. The parent node is full, so it splits, sending the median M up to form a new root node.



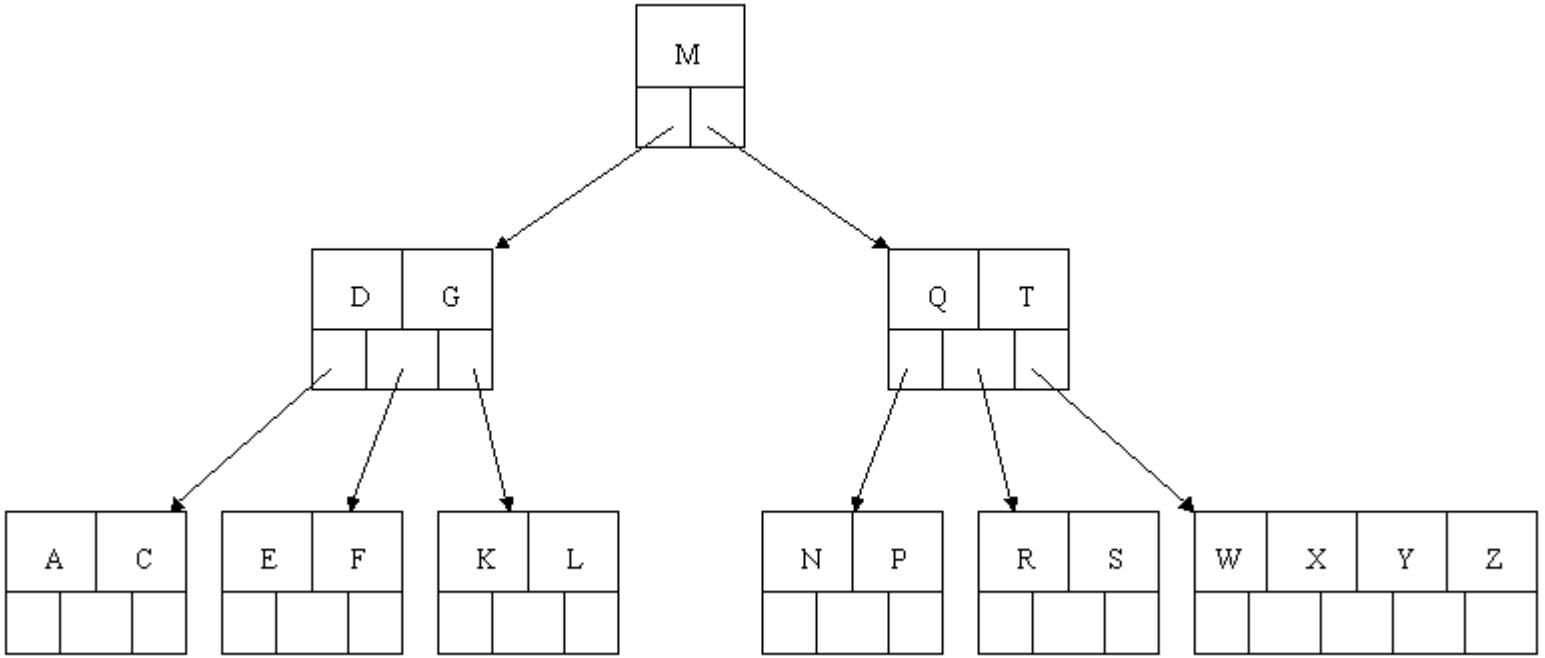
B Trees

- Delete H



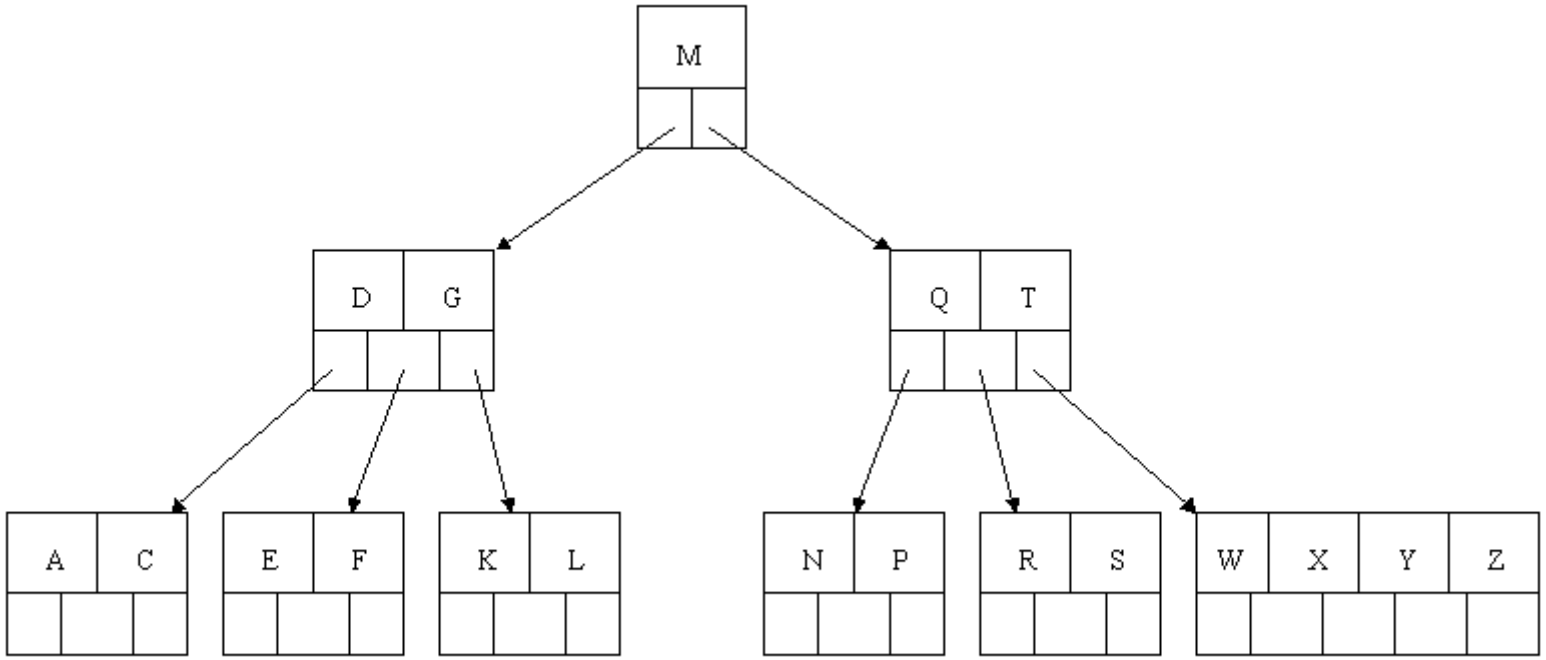
B Trees

- Delete H



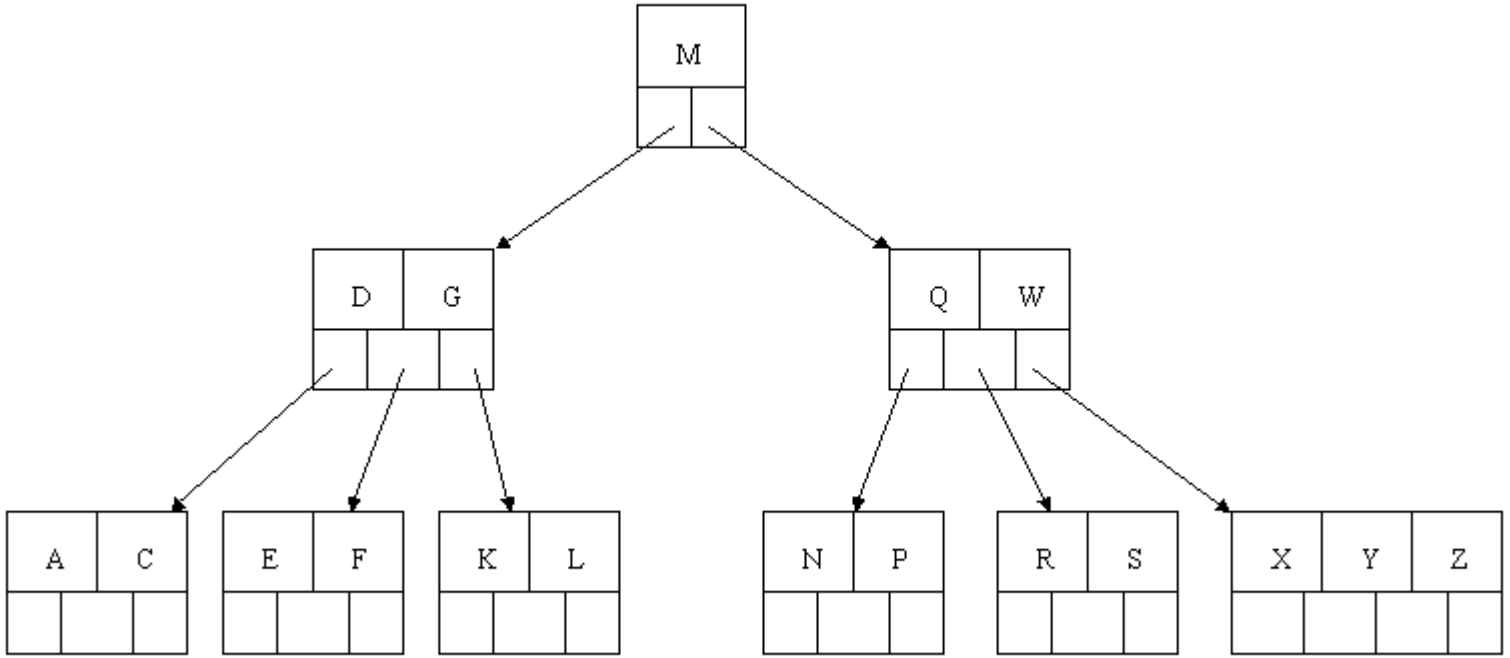
B Trees

- Next, delete T.



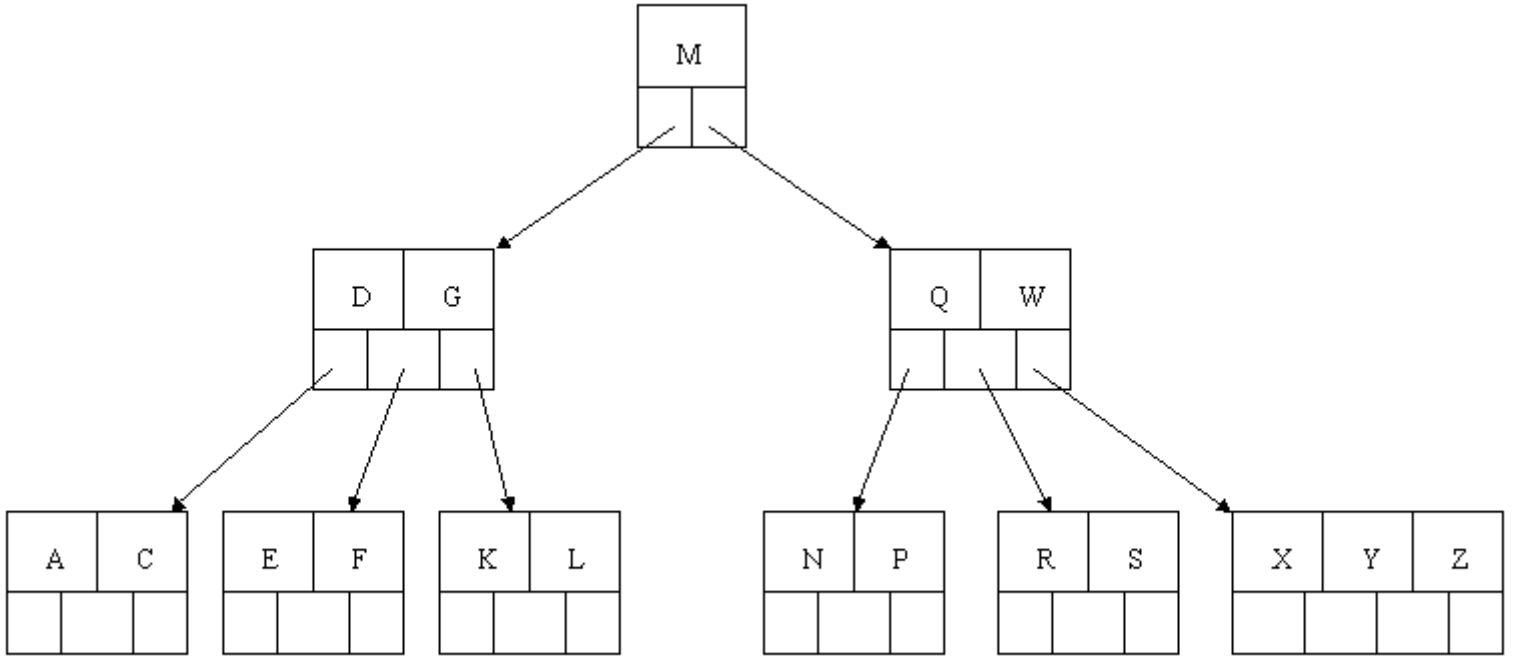
B Trees

- Since T is not in a leaf, we find its successor (the next item in ascending order)
- move W up to replace the T



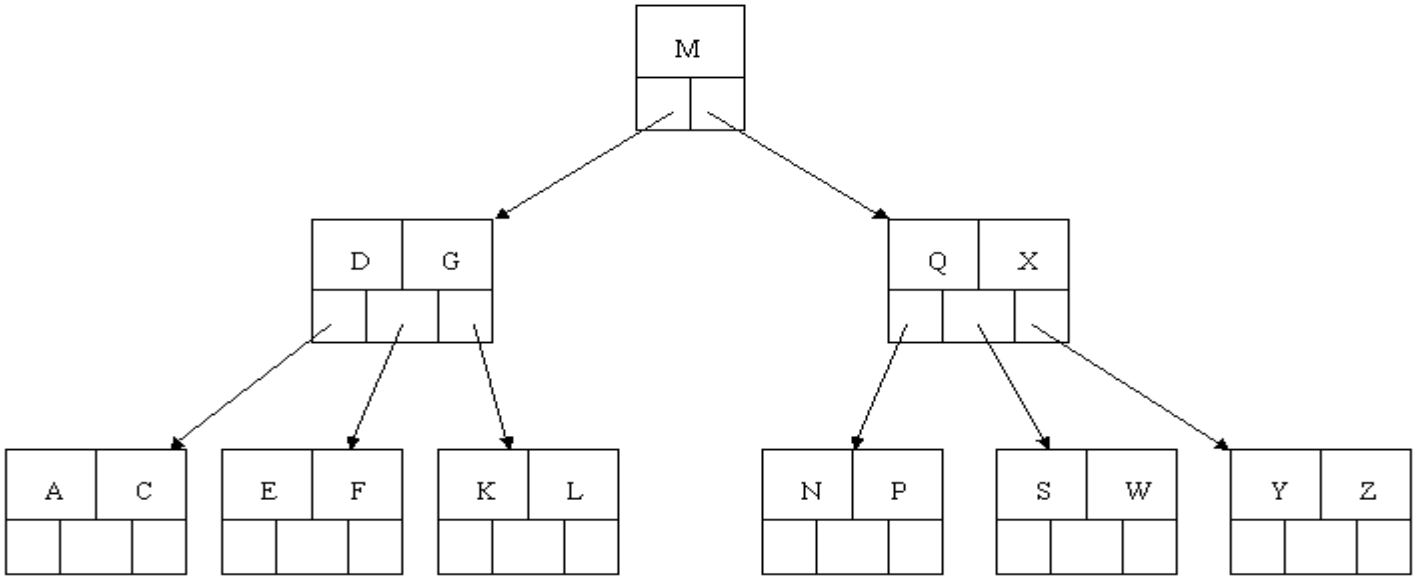
B Trees

- Next, delete R



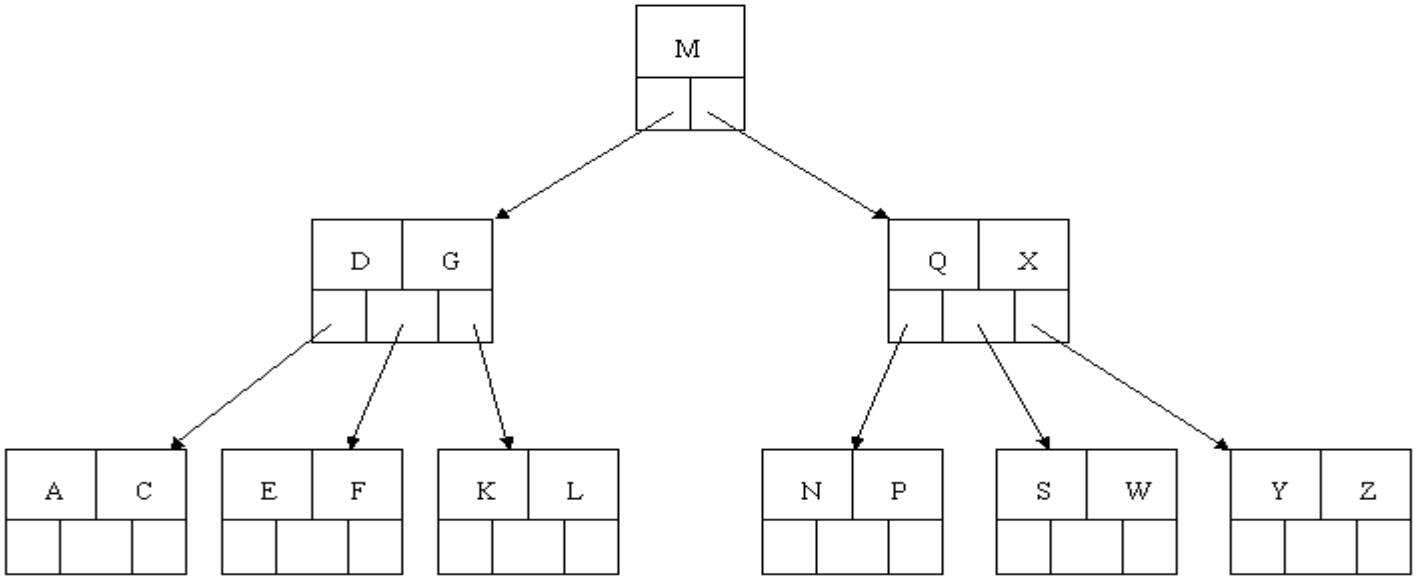
B Trees

- Next, delete R - the deletion results in a node with only one key,
- If the sibling node to the immediate left or right has an extra key, borrow a key from the parent and move a key up from this sibling.
- the successor W of S is moved down, left key X of sibling moved up.



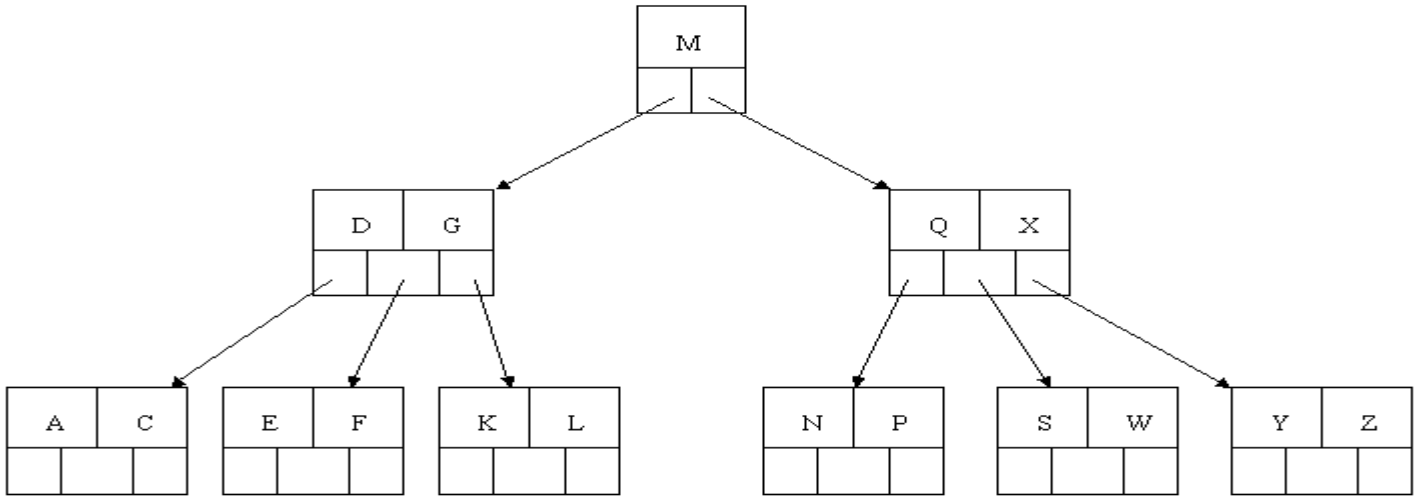
B Trees

- Finally, delete E



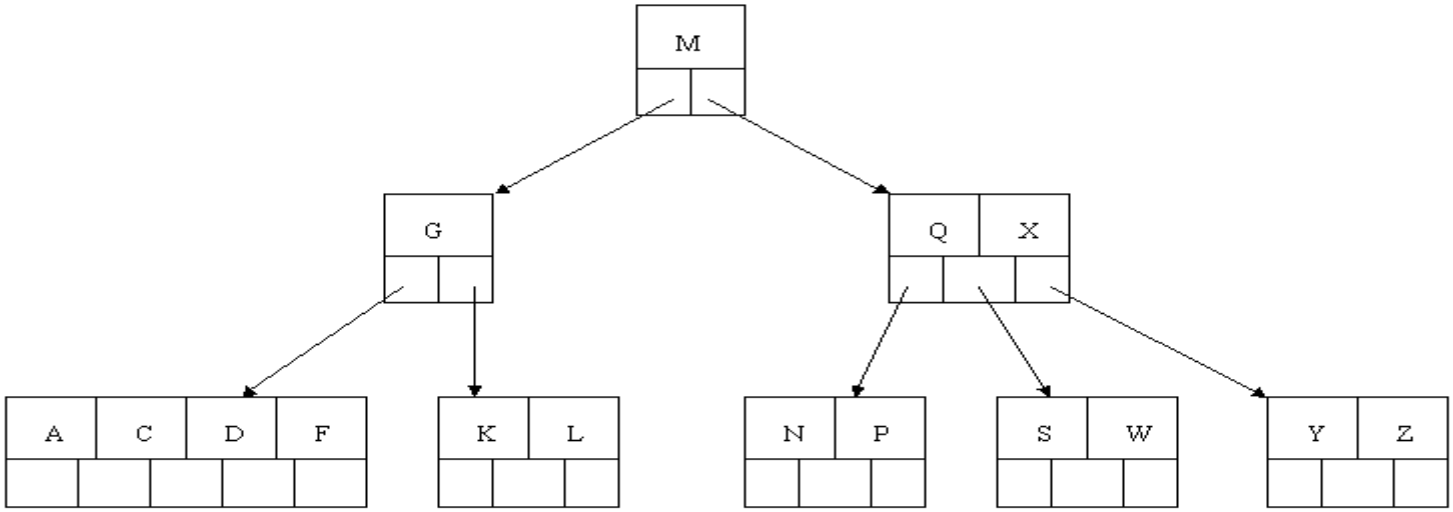
B Trees

- Finally, delete E
- Although E is in a leaf, the leaf has no extra keys, nor do the siblings to the immediate right or left
- the leaf has to be combined with one of these two siblings, move parent down (D)



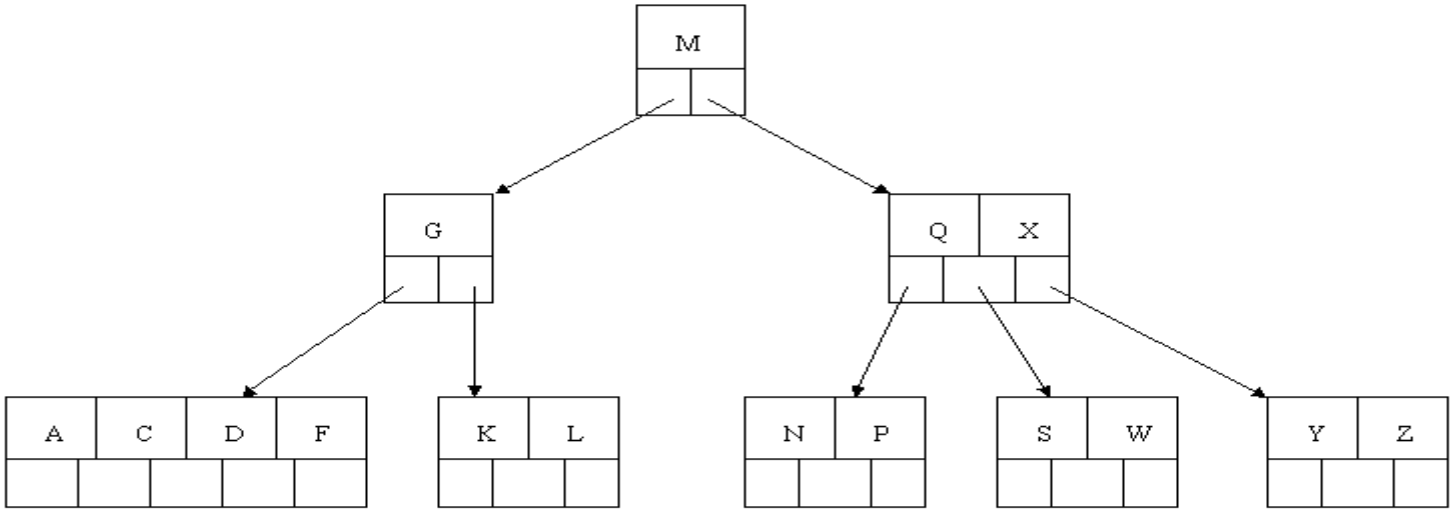
B Trees

- Finally, delete E
- Although E is in a leaf, the leaf has no extra keys, nor do the siblings to the immediate right or left
- the leaf has to be combined with one of these two siblings, move parent down (D)



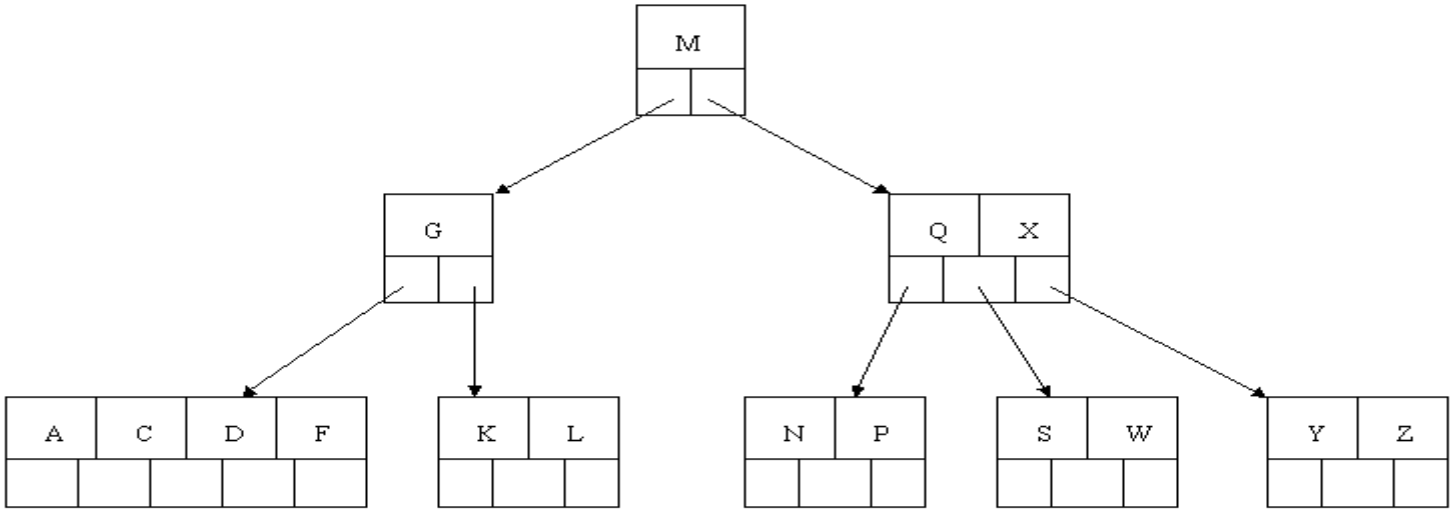
B Trees

- the parent node now contains only one key, G
- No left or right sibling to borrow key
- Move M down (Tree shrink by 1)



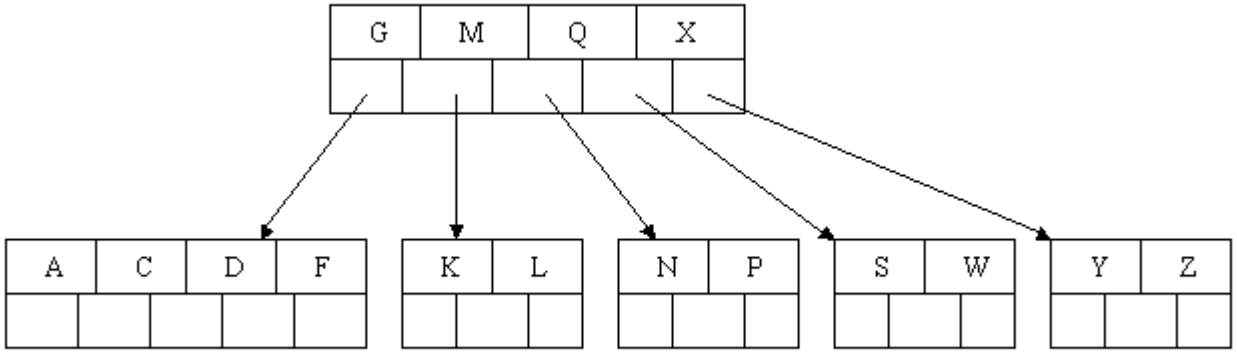
B Trees

- the parent node now contains only one key, G



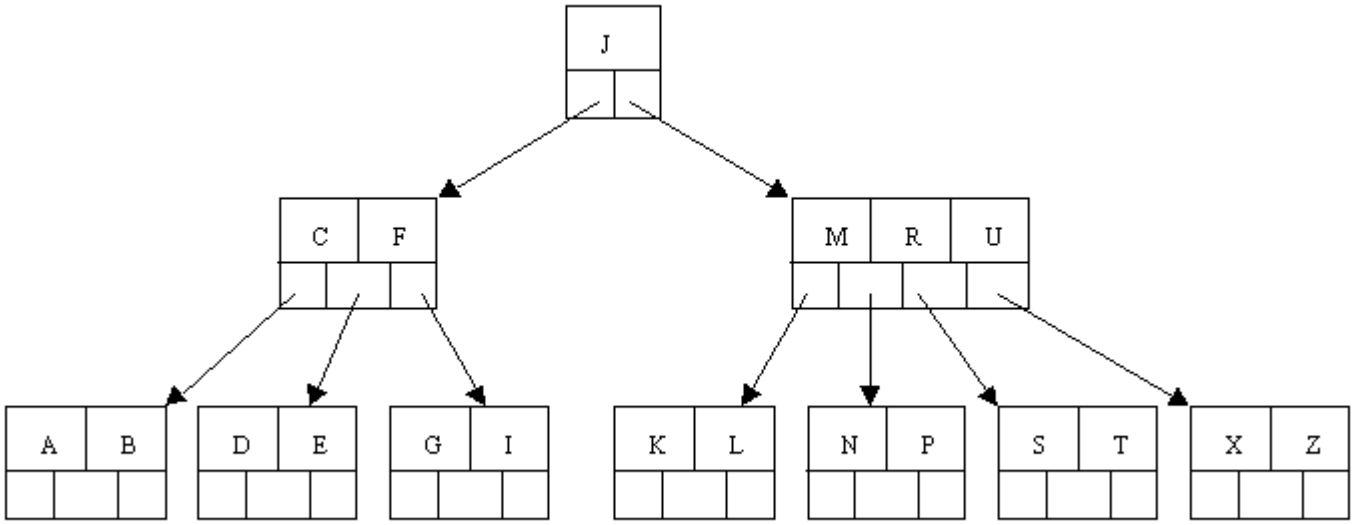
B Trees

- the parent node now contains only one key, G
- No left or right sibling to borrow key
- Move M down (Tree shrink by 1)



B Trees

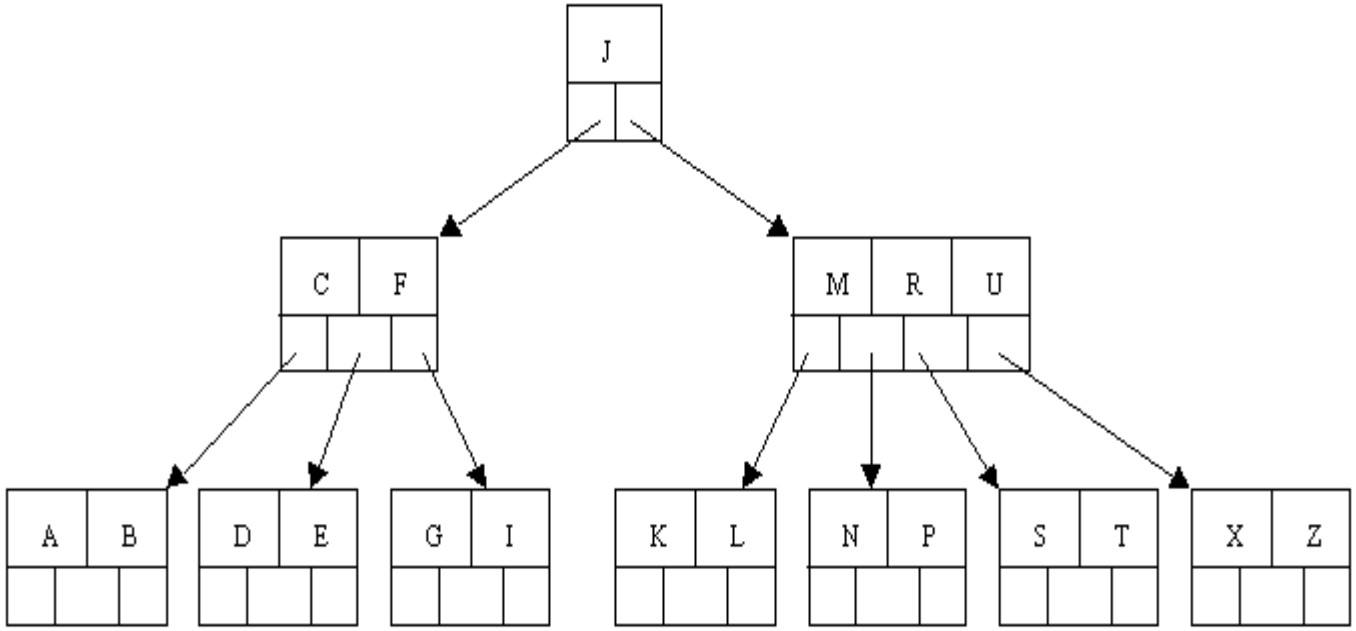
- Delete C



Delete c

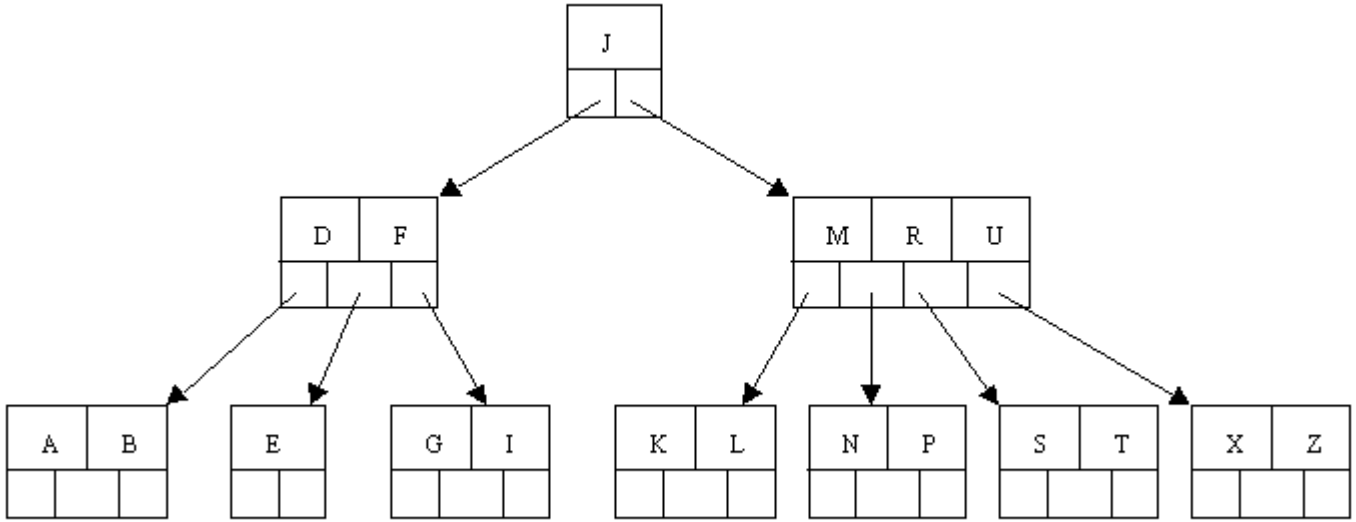
Deleting C – find immediate Successor of - D

Move D up, to replace C



Delete c

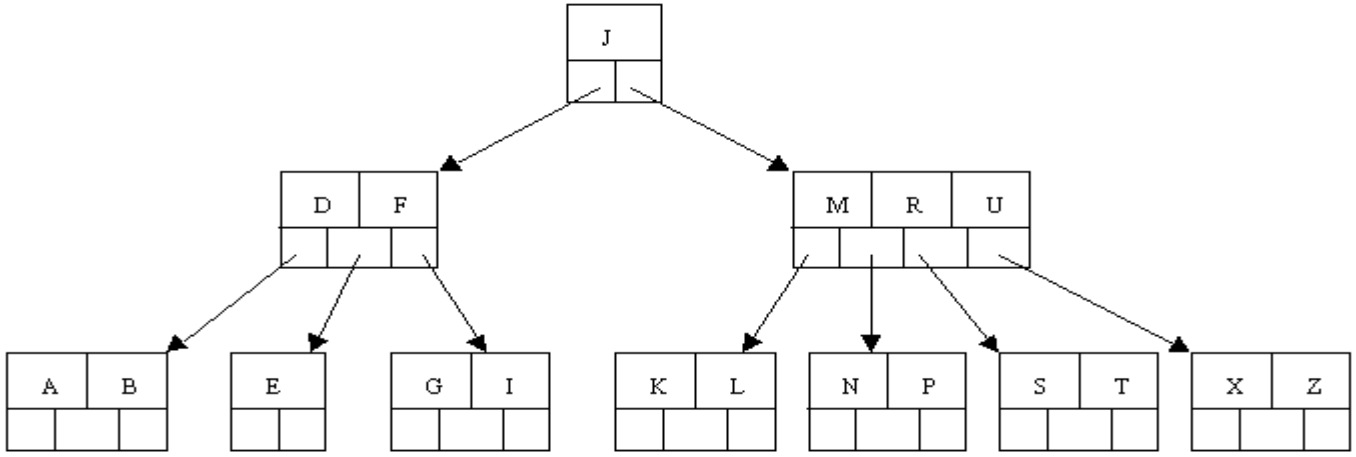
Deleting C – Leaves Only E



Delete c

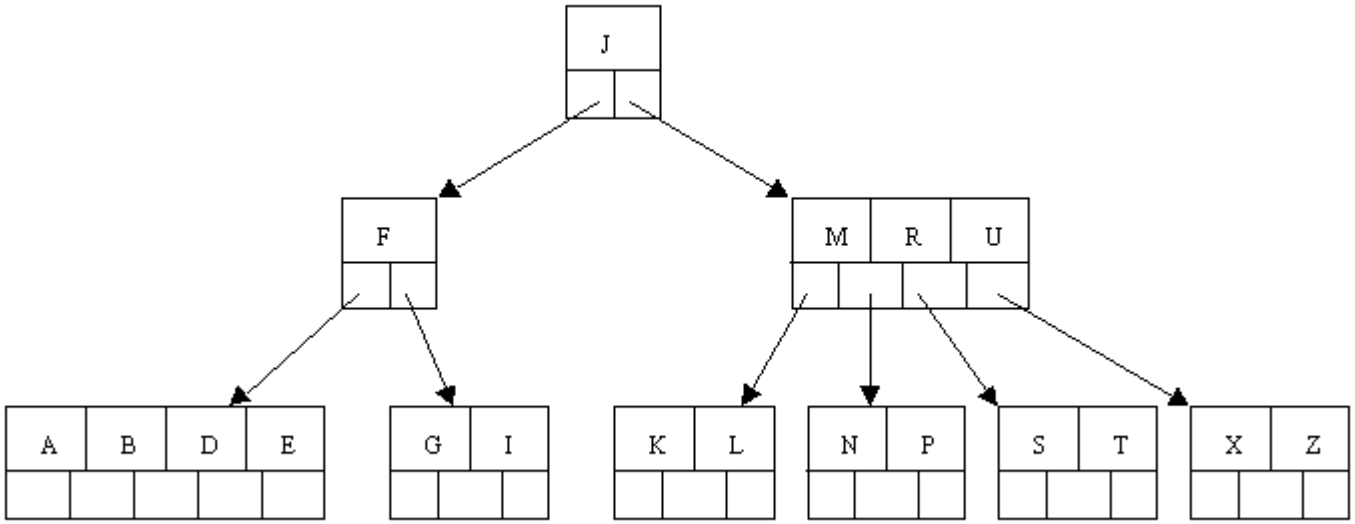
Deleting C – Leaves Only E – No sibling with spare key

Combine



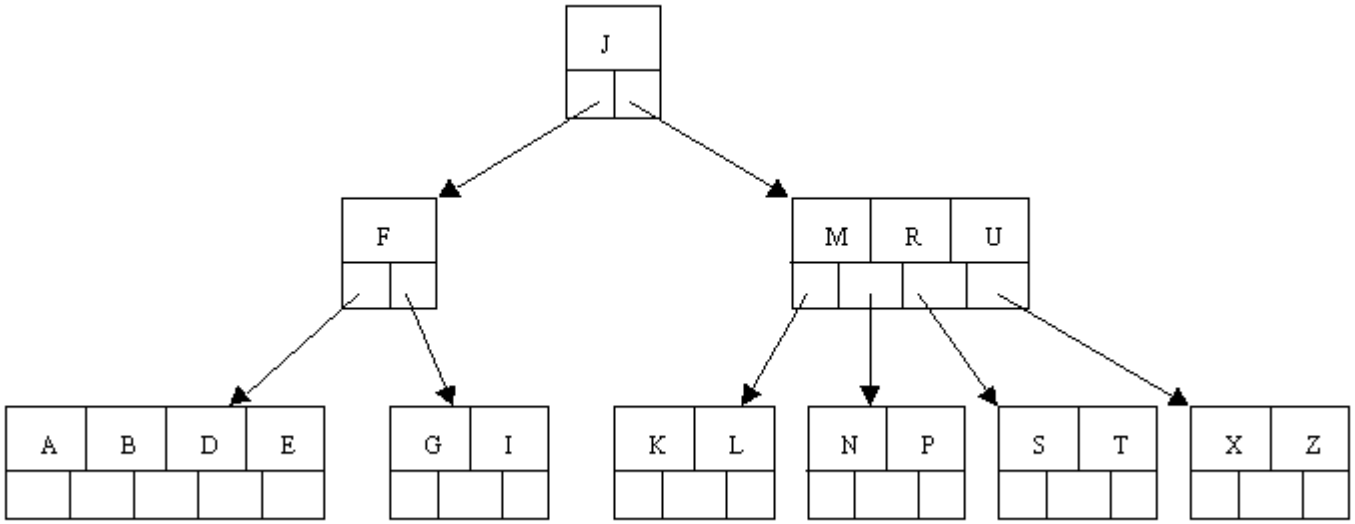
Delete c

Deleting C – consolidate with the A B node



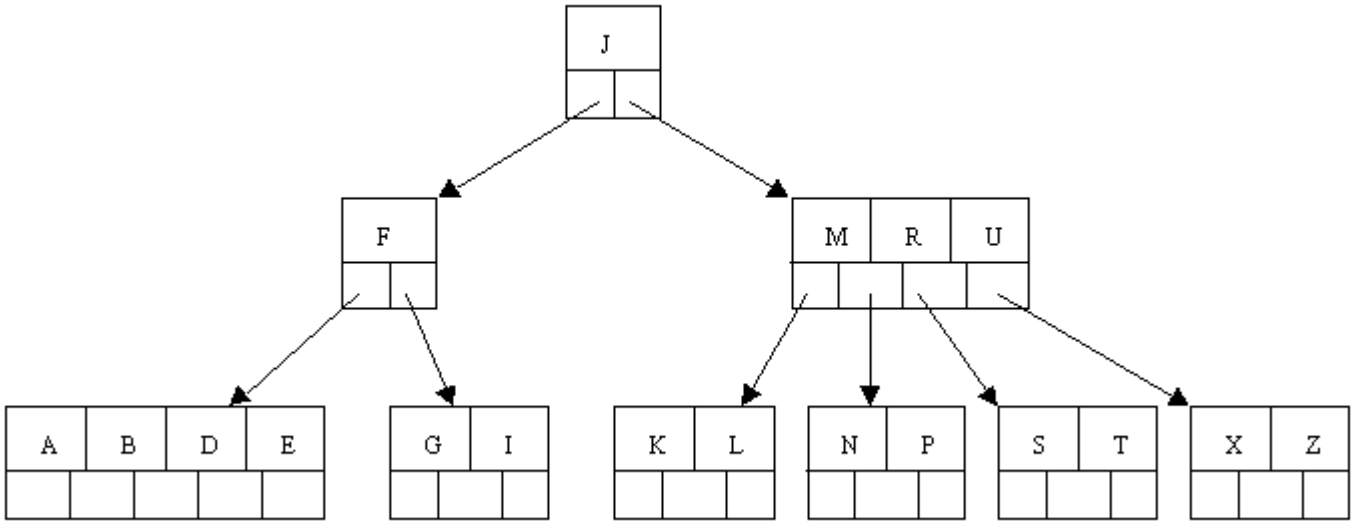
Delete c

Deleting C – F has not enough keys



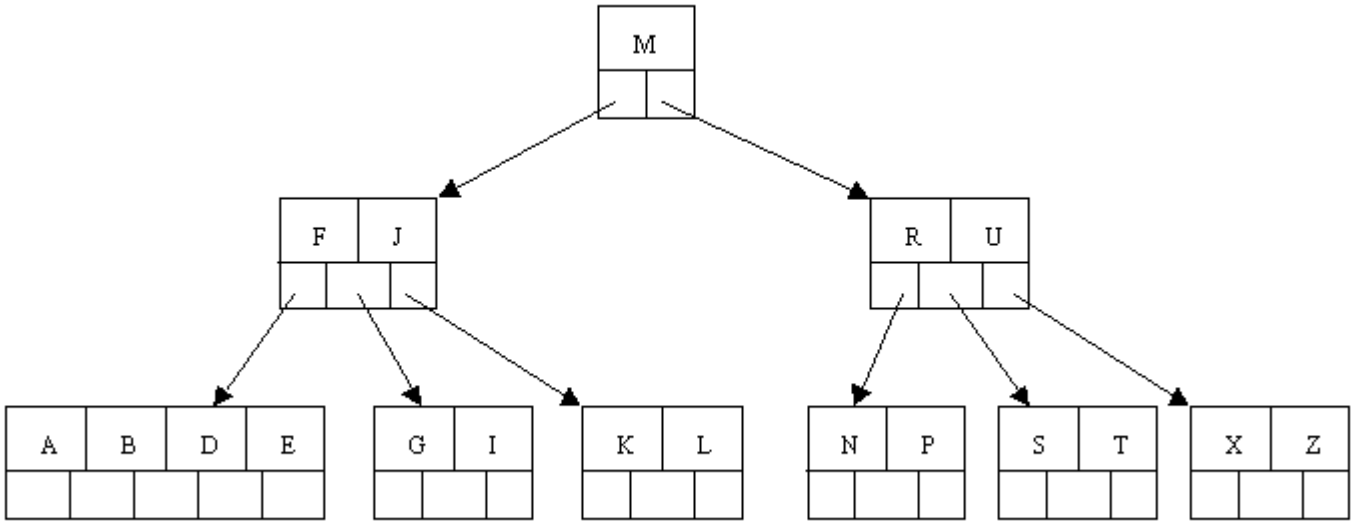
Delete c

Deleting C – F has not enough keys – borrow from sibling



Delete c

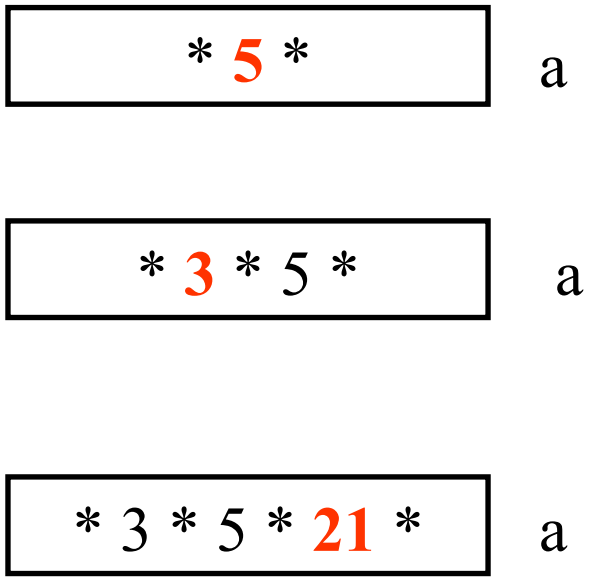
Deleting C – F has not enough keys – borrow from sibling



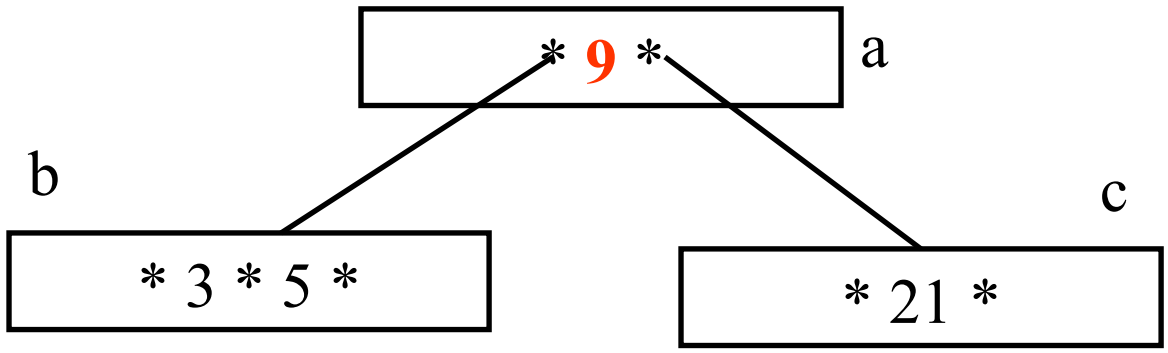
Operations

- B-Tree of order 4
  - Each node has at most 4 pointers and 3 keys, and at least 2 pointers and 1 key.
- Insert: 5, 3, 21, 9, 1, 13, 2, 7, 10, 12, 4, 8
- Delete: 2, 21, 10, 3, 4

Insert 5, 3, 21

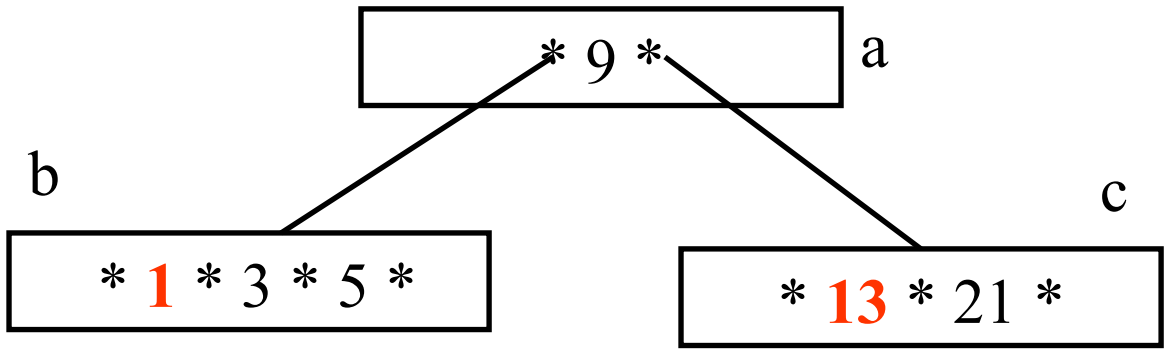


Insert 9



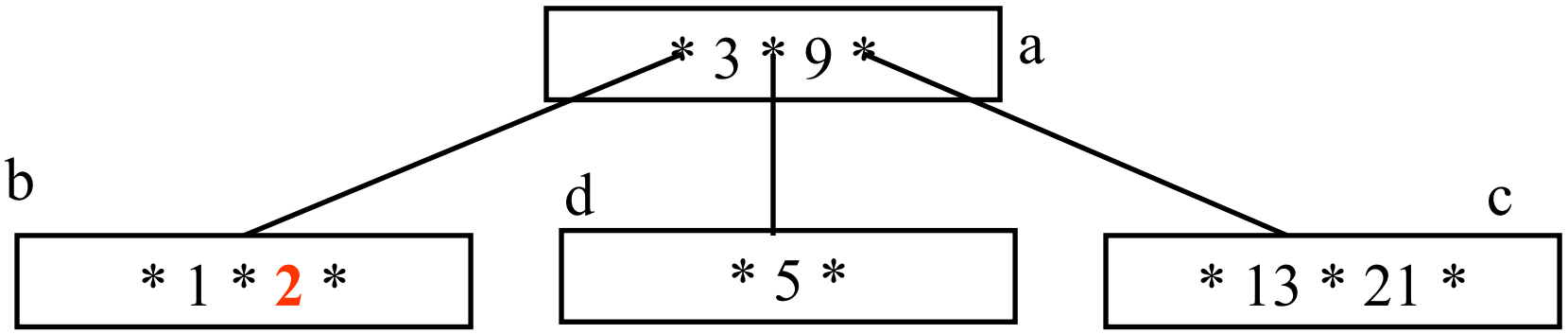
Node a splits creating 2 children: b and c

Insert 1, 13



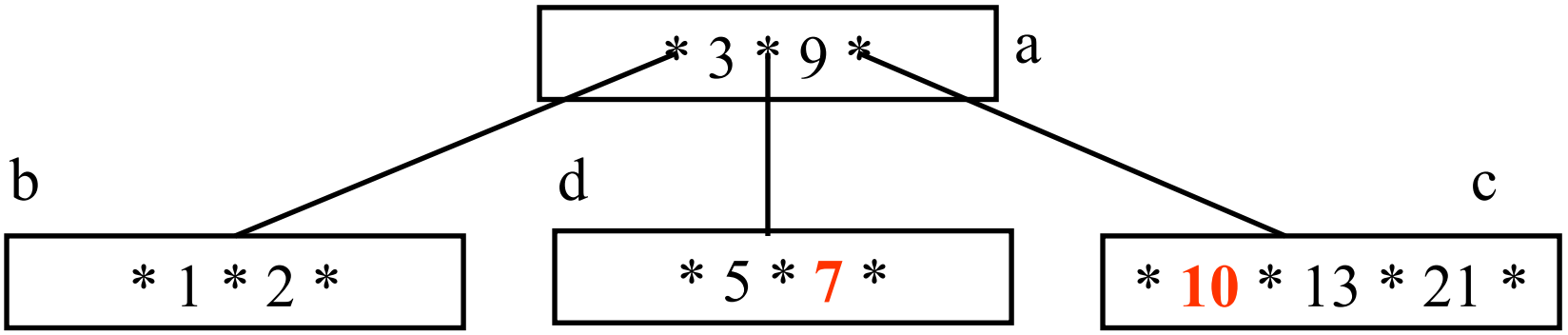
Nodes b and c have room to insert more elements

Insert 2



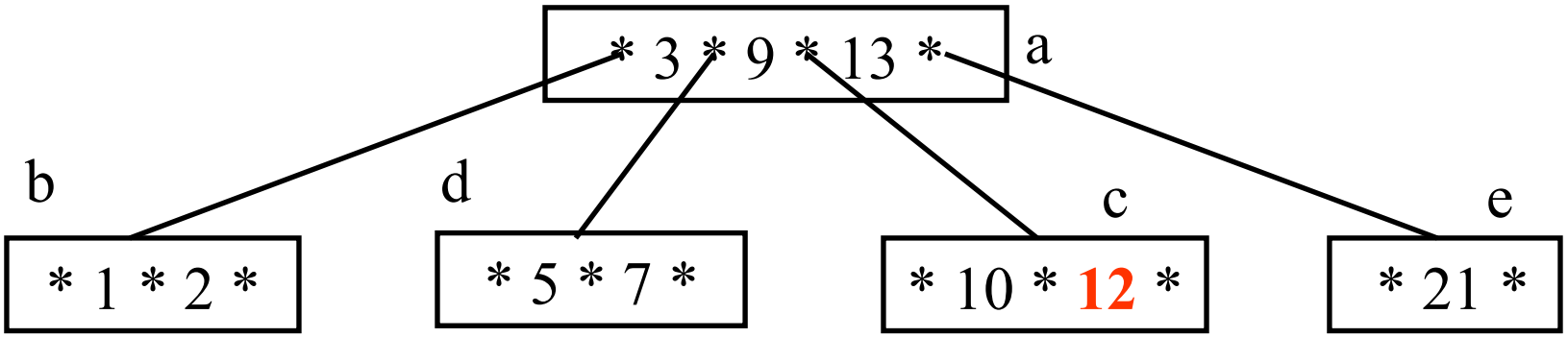
Node b has no more room, so it splits creating node d.

Insert 7, 10



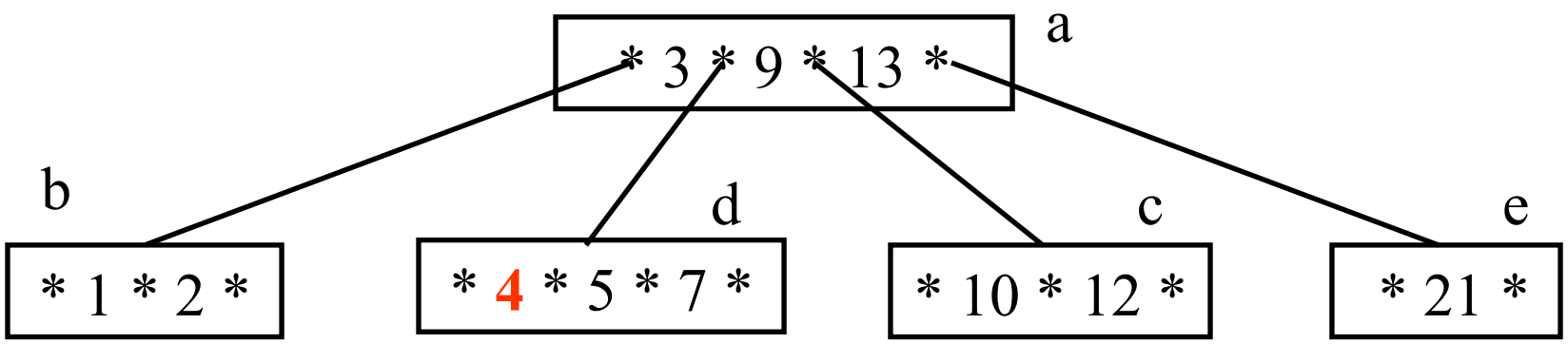
Nodes d and c have room to add more elements

Insert 12



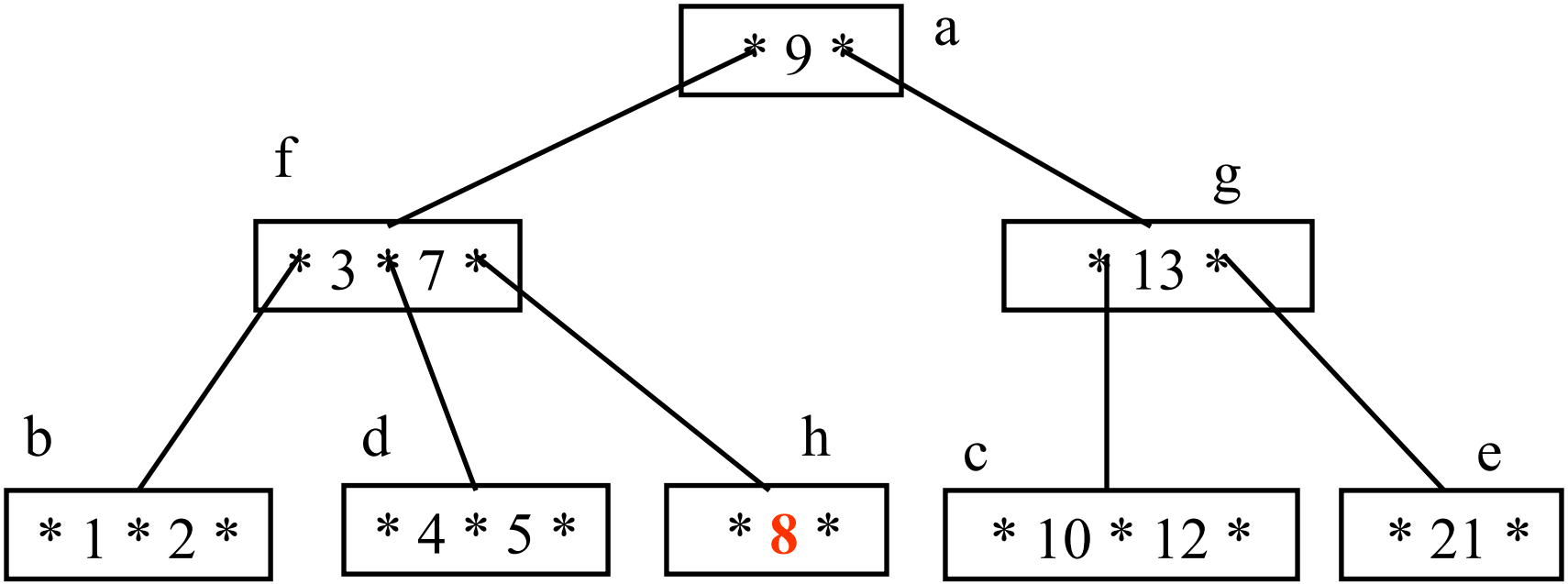
Nodes c must split into nodes c and e

Insert 4



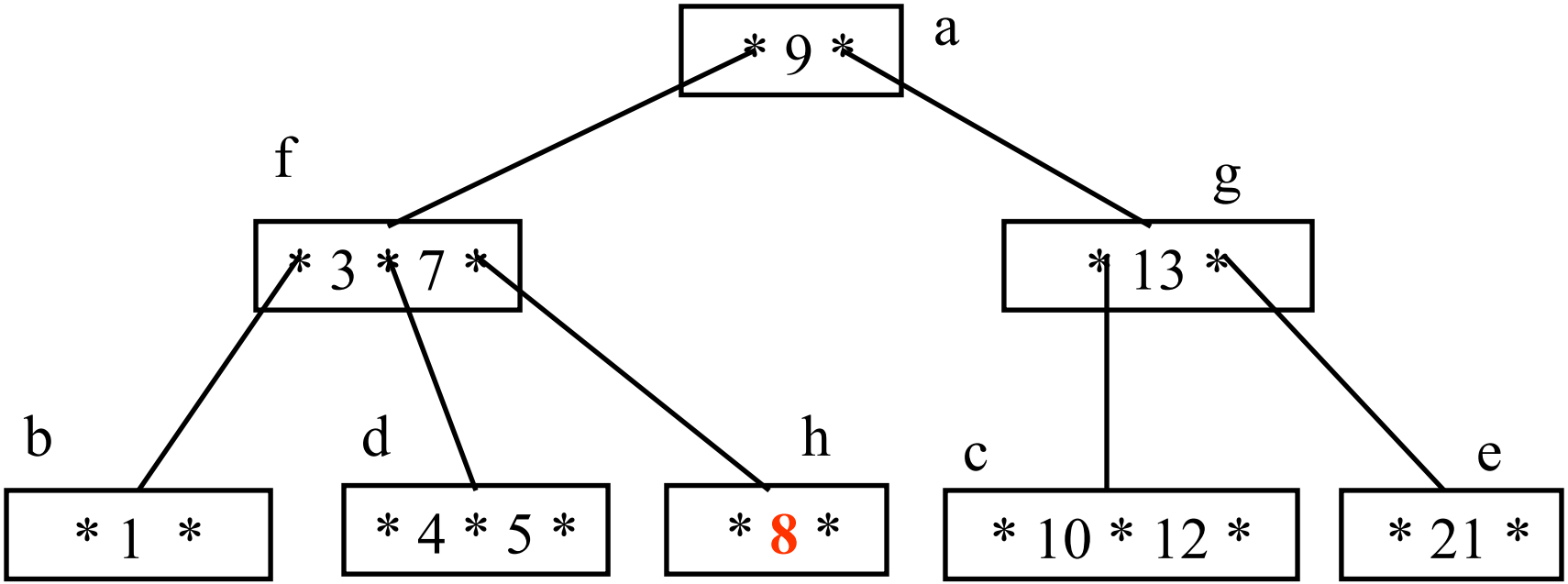
Node d has room for another element

Insert 8



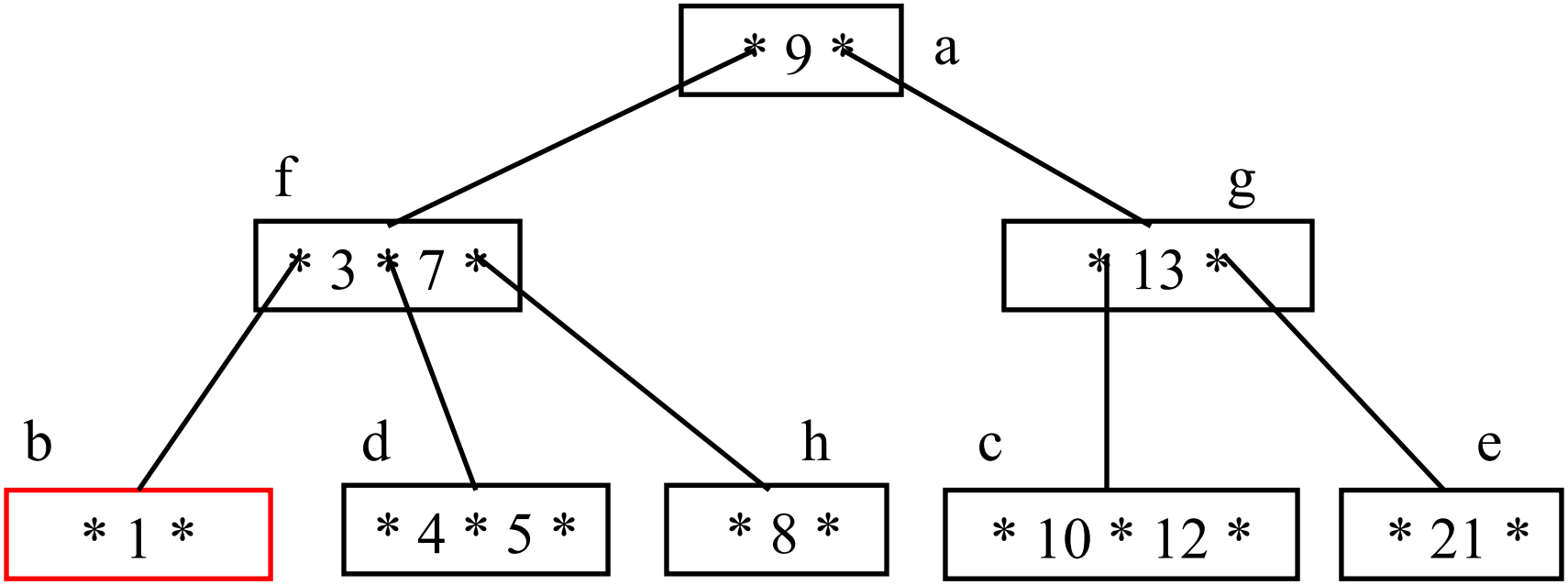
Node d must split into 2 nodes. This causes node a to split into 2 nodes and the tree grows a level.

Delete 2



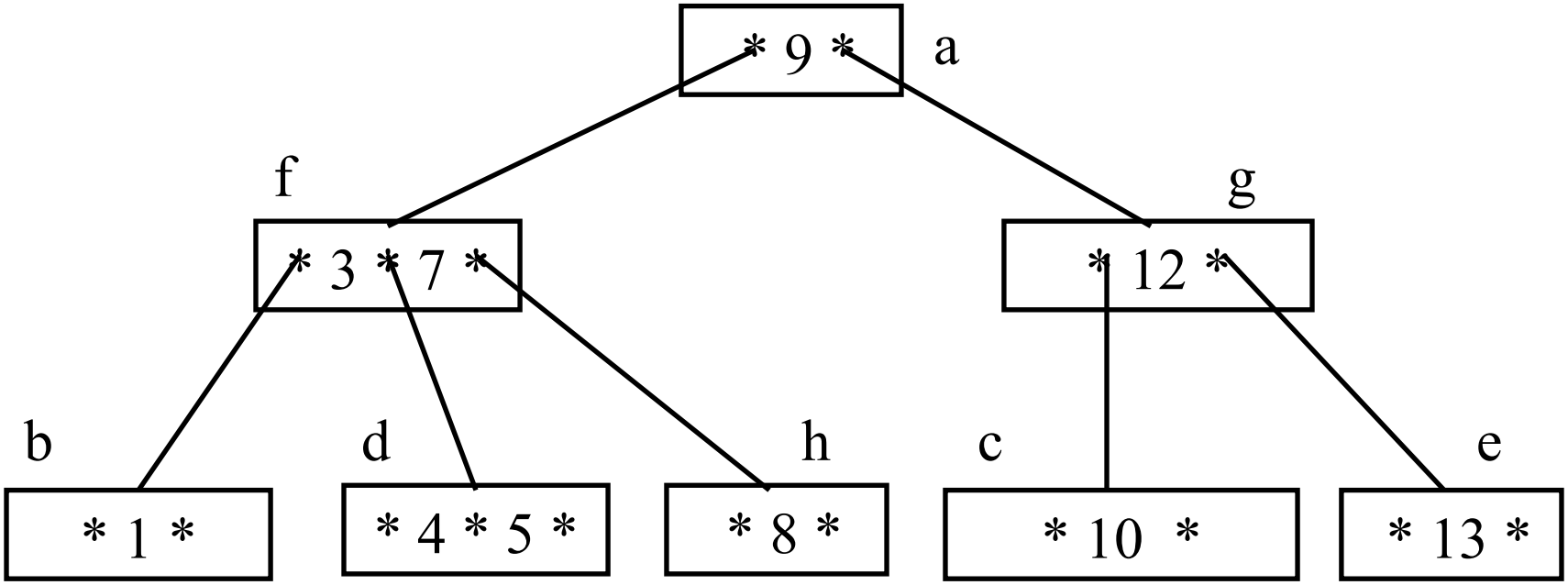
Node d must split into 2 nodes. This causes node a to split into 2 nodes and the tree grows a level.

Delete 2



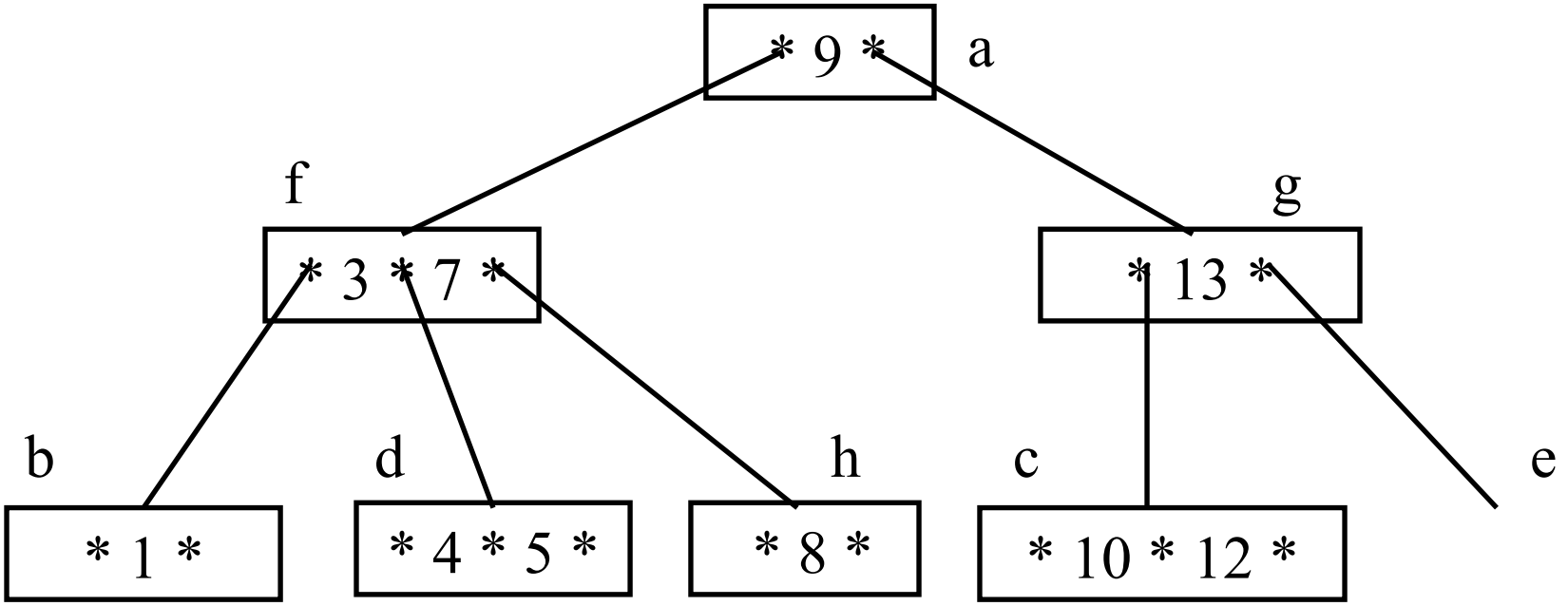
Node b can loose an element without underflow.

Delete 21



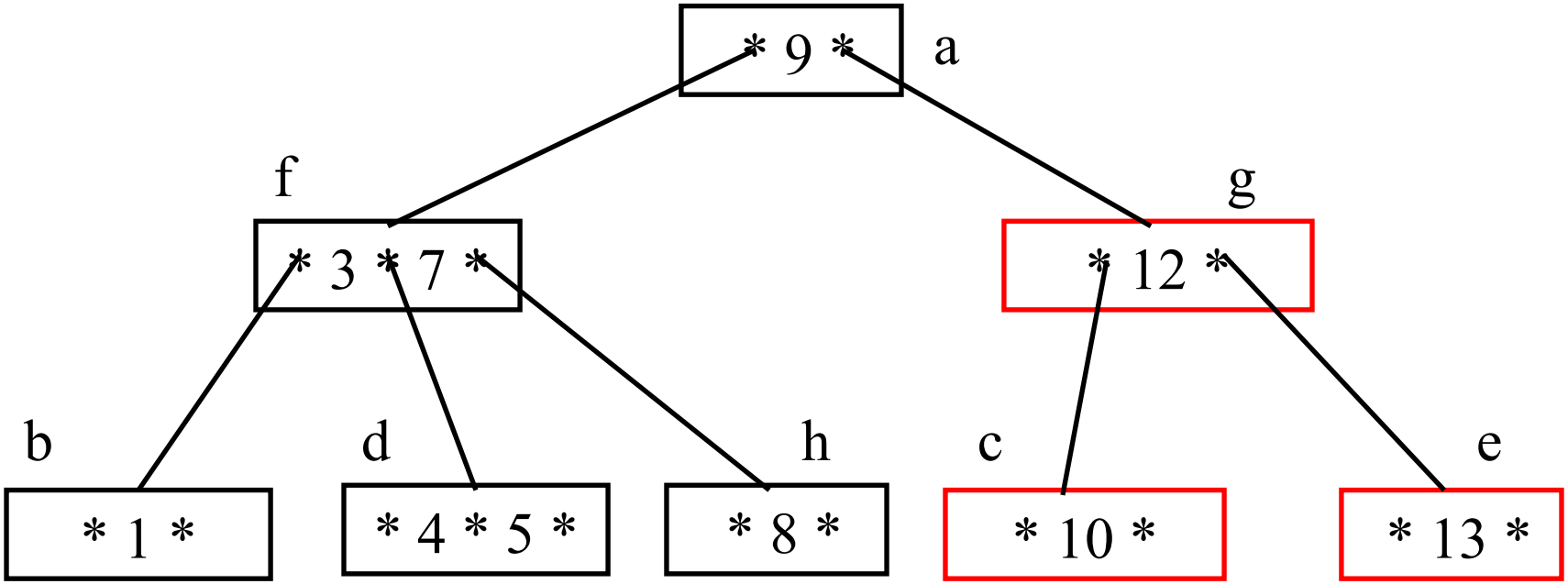
.

Delete 21



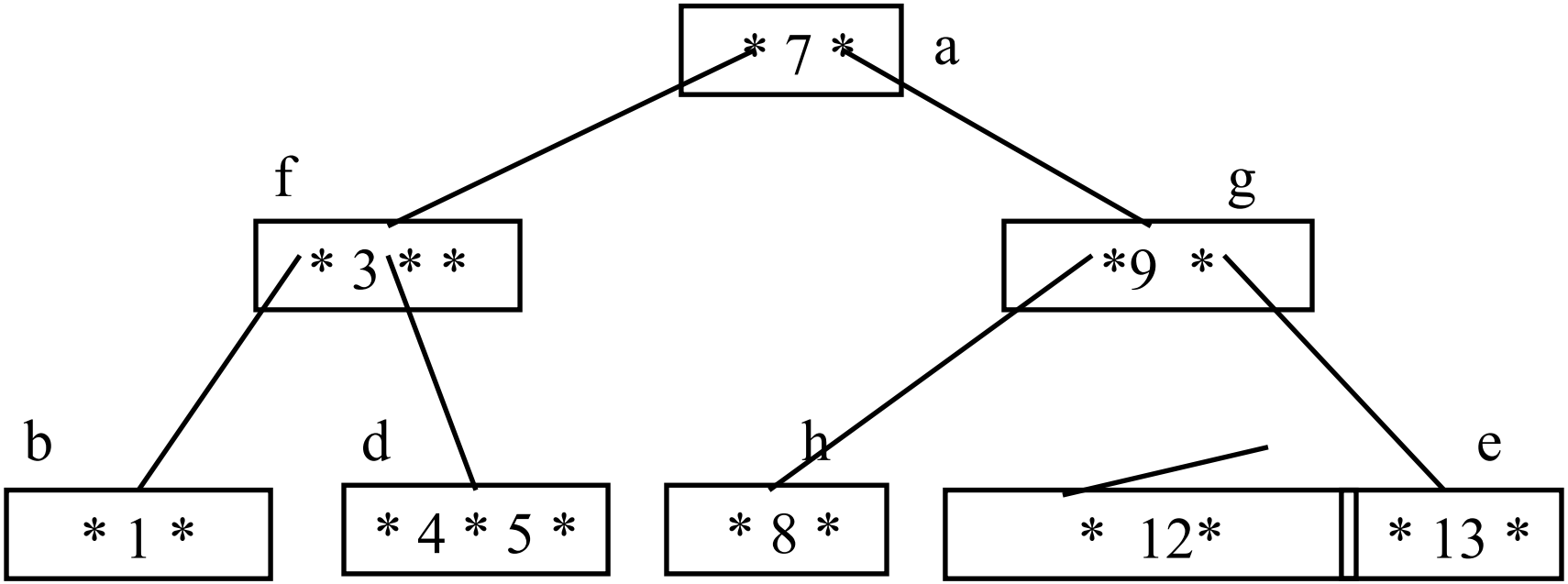
.

Delete 21

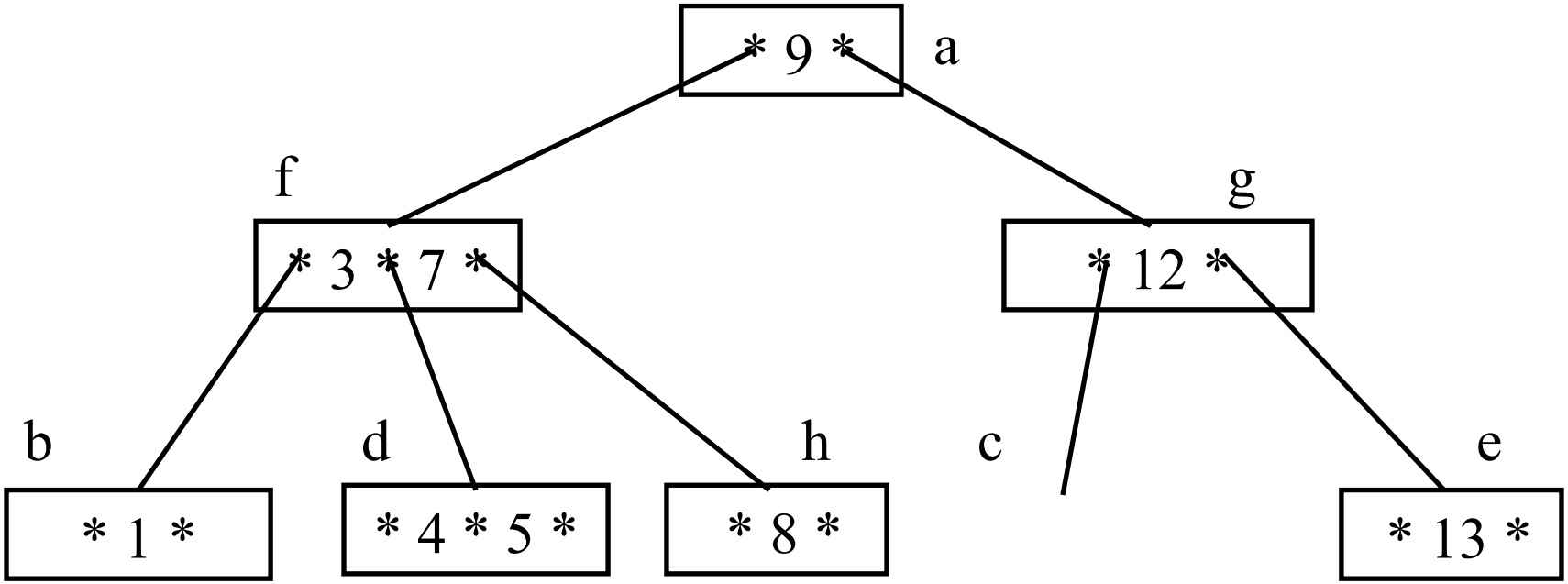


Deleting 21 causes node e to underflow, so elements are redistributed between nodes c, g, and e

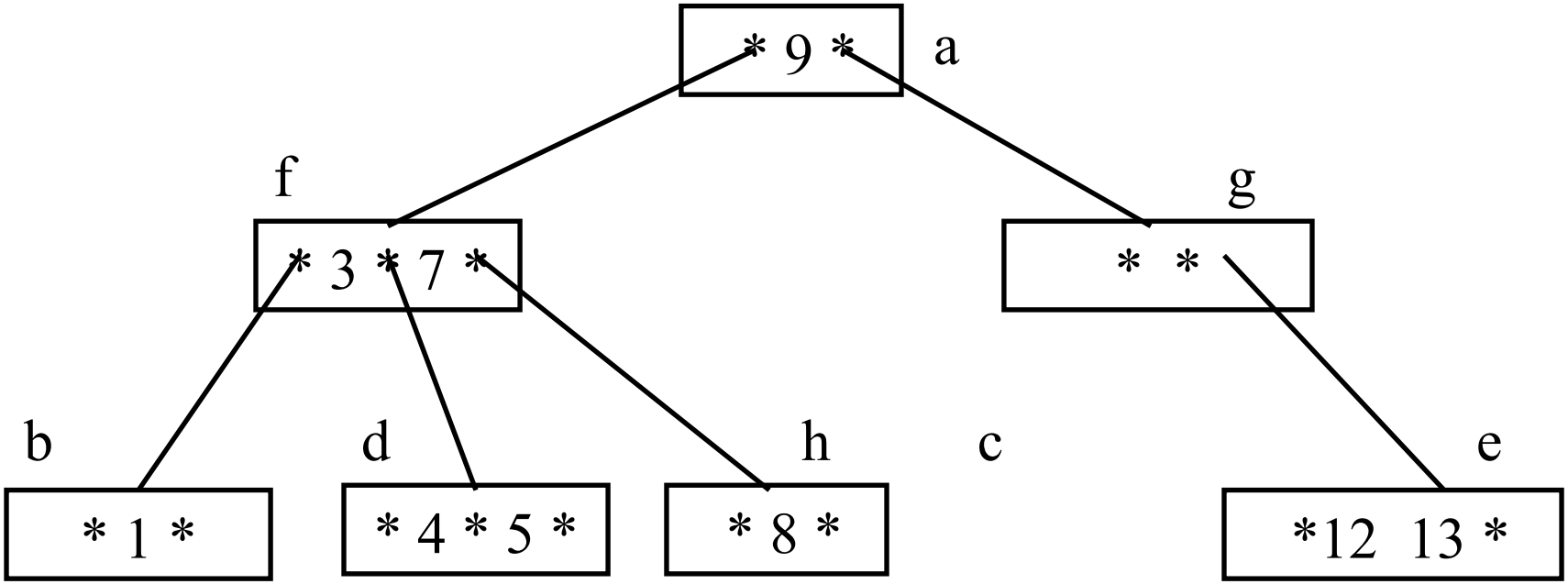
Delete 10



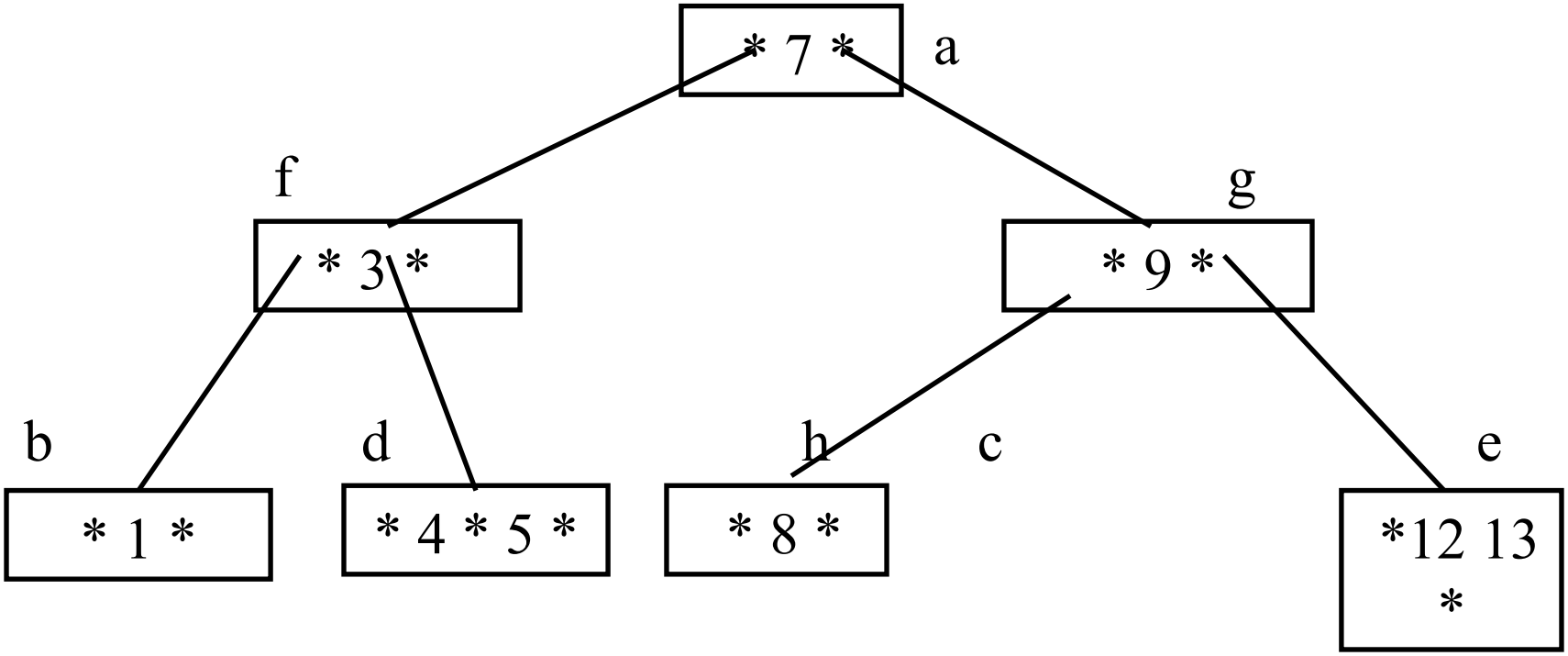
Delete 10



Delete 10



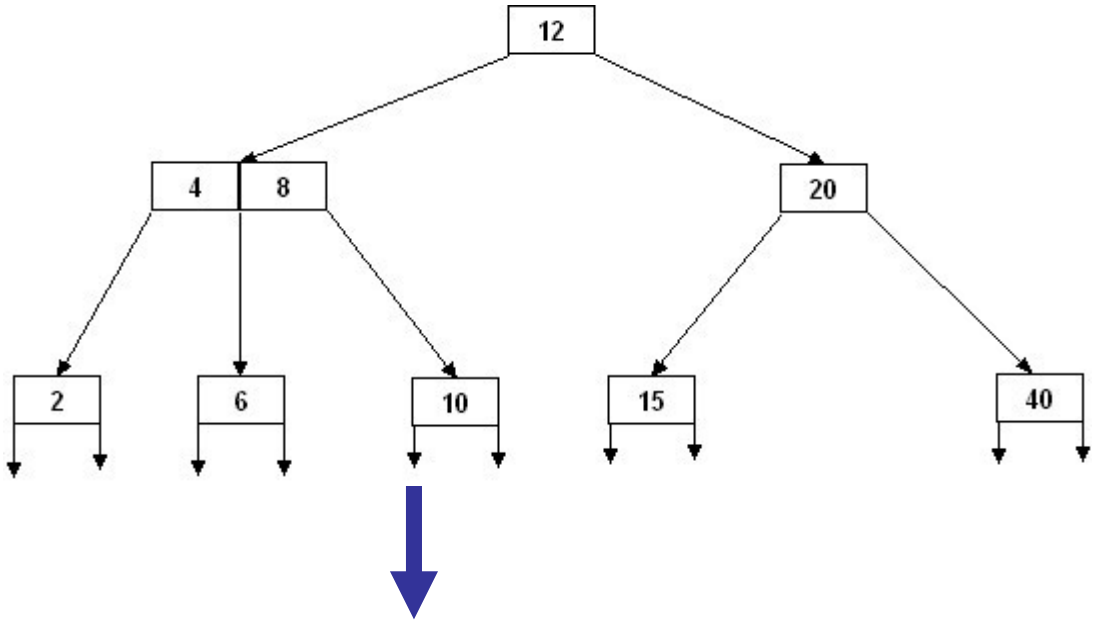
Delete 10



Merge NULL and 13 nodes

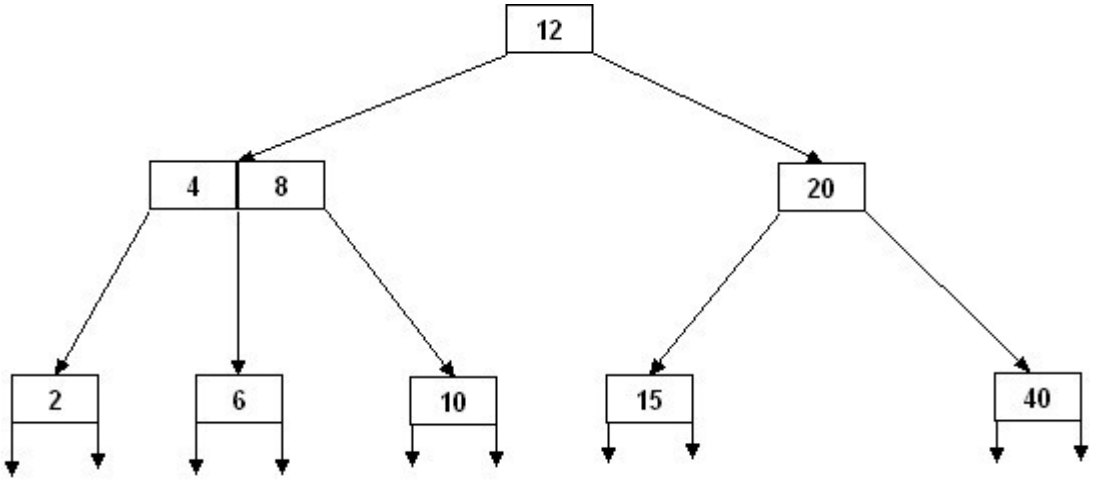
Deletion :

:Example :Delete the key **40** in the following B-tree of order **3**

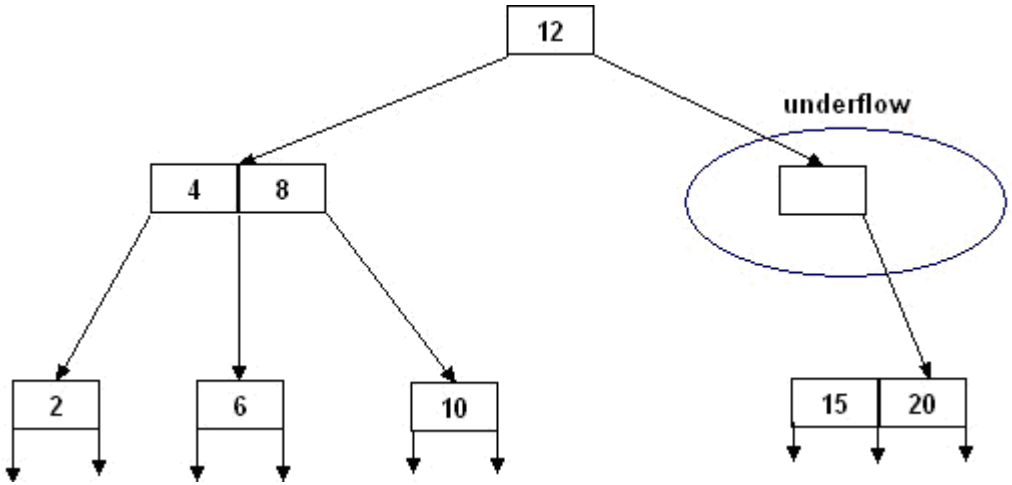


Deletion : Special Case, involves rotation and merging

:Example :Delete the key **40** in the following B-tree of order **3**

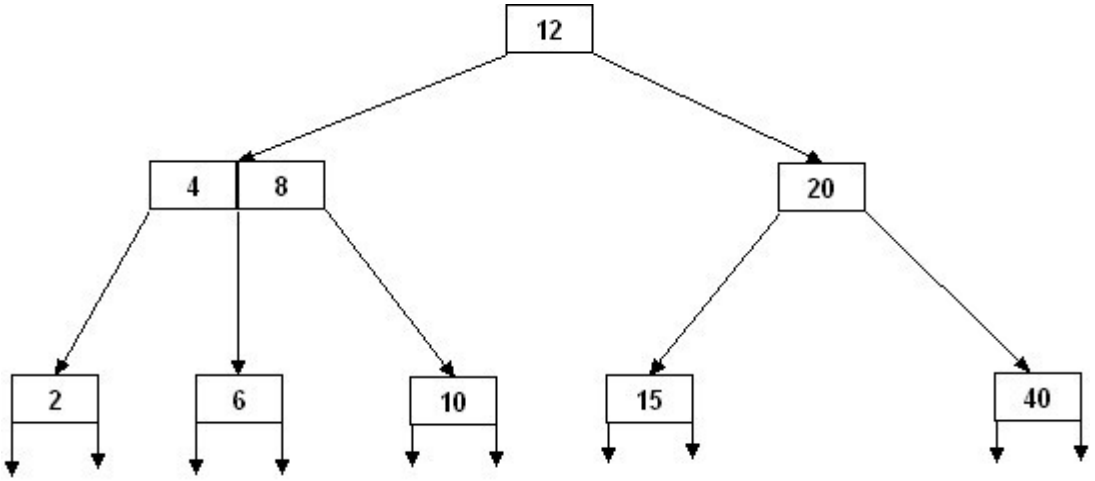


merge 15 and 20

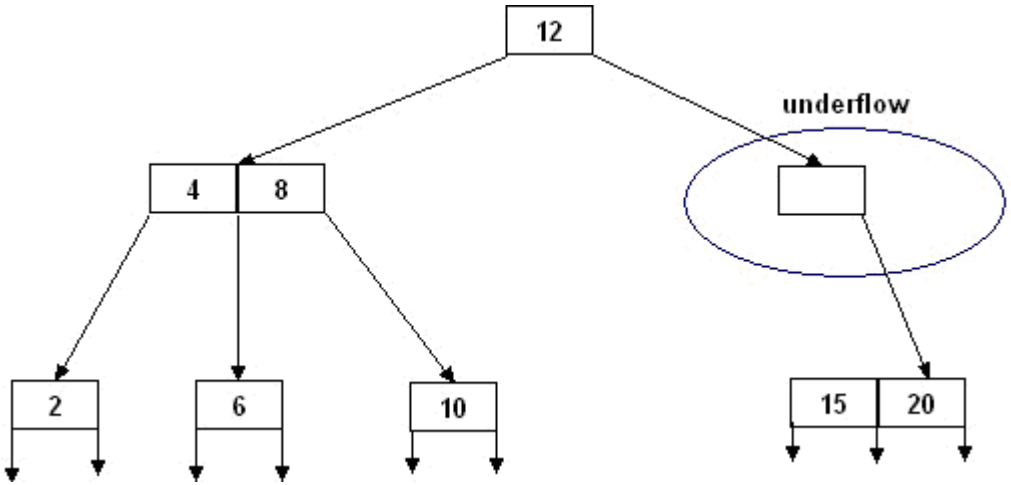


Deletion : Special Case, involves rotation and merging

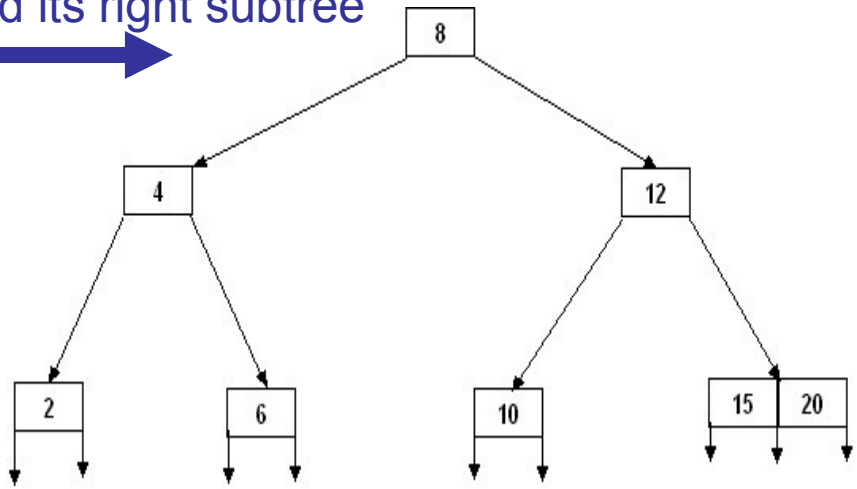
:Example :Delete the key **40** in the following B-tree of order **3**



merge 15 and 20



rotate 8 and its right subtree

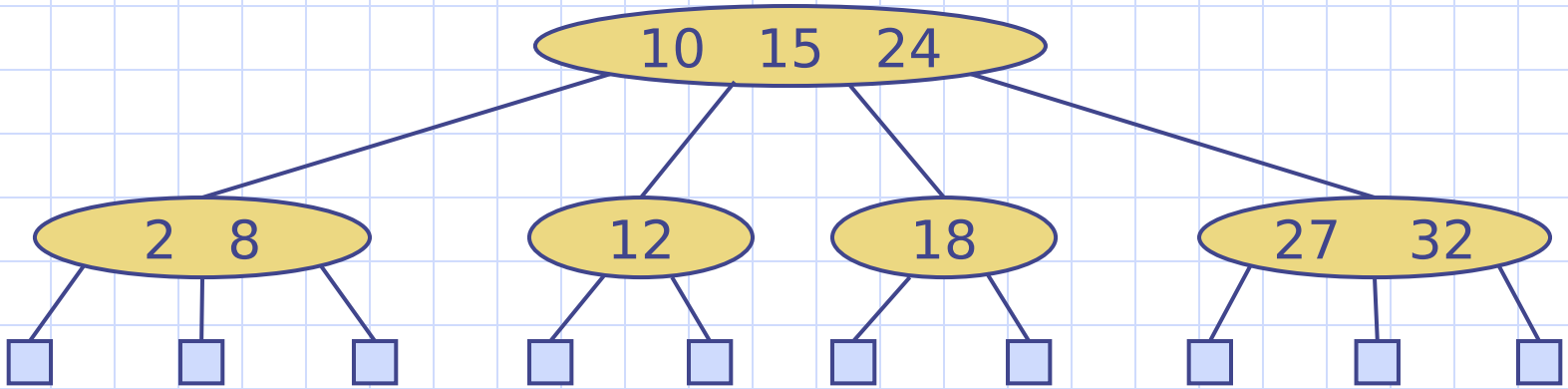


Data Structure for B-Trees

```
typedef struct {
 int Count; // number of keys stored in the current node
 ItemType Key[3]; // array to hold the 3 keys long
 Branch[4]; // array of  pointers (record numbers)
 } NodeType;
```

(2,4) Trees

- A (2,4) tree (also called 2-4 tree or 2-3-4 tree) is a multi-way search with the following properties
  - Node-Size Property: every internal node has at most four children
  - Depth Property: all the external nodes have the same depth
- Depending on the number of children, an internal node of a (2,4) tree is called a 2-node, 3-node or 4-node



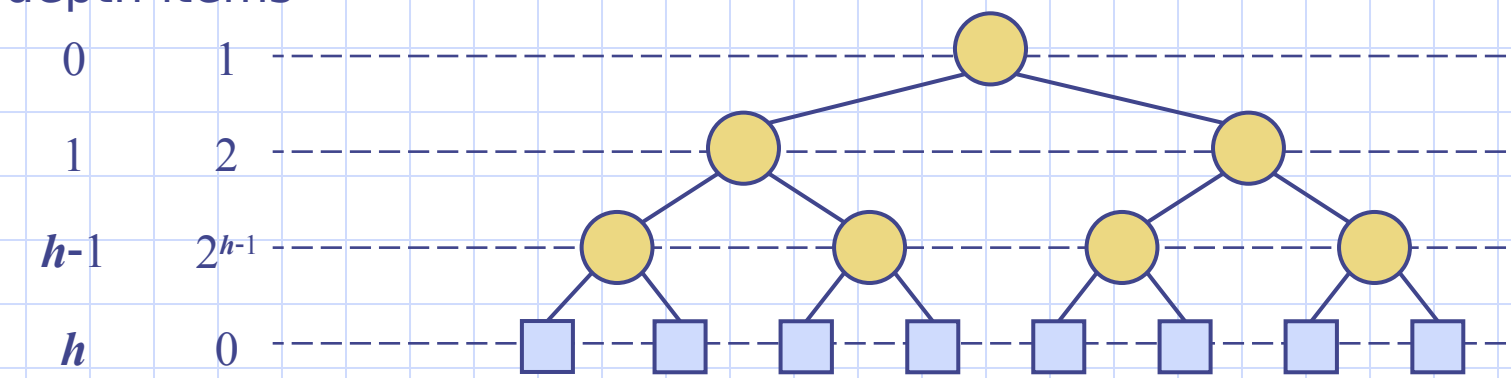
Height of a (2,4) Tree

- Theorem: A (2,4) tree storing n items has height $O(\log n)$

  Proof:
  - Let h be the height of a (2,4) tree with n items
  - Since there are at least 2^i items at depth $i = 0, \ldots, h - 1$ and no items at depth h, we have

    $$n \geq 1 + 2 + 4 + \ldots + 2^{h-1} = 2^h - 1$$

  - Thus, $h \leq \log (n + 1)$
- Searching in a (2,4) tree with n items takes $O(\log n)$ time



Analysis of Insertion

Algorithm *put*(k, o)

1. We search for key k to locate the insertion node v
2. We add the new entry (k, o) at node v
3. **while** *overflow*(v)

   if *isRoot*(v)

   create a new empty root above v

   $v \leftarrow$ *split*(v)

- Let T be a (2,4) tree with n items
  - Tree T has $O(\log n)$ height
  - Step 1 takes $O(\log n)$ time because we visit $O(\log n)$ nodes
  - Step 2 takes $O(1)$ time
  - Step 3 takes $O(\log n)$ time because each split takes $O(1)$ time and we perform $O(\log n)$ splits
- Thus, an insertion in a (2,4) tree takes $O(\log n)$ time

Analysis of Deletion

- Let T be a (2,4) tree with n items
  - Tree T has $O(\log n)$ height
- In a deletion operation
  - We visit $O(\log n)$ nodes to locate the node from which to delete the entry
  - We handle an underflow with a series of $O(\log n)$ fusions, followed by at most one transfer
  - Each fusion and transfer takes $O(1)$ time
- Thus, deleting an item from a (2,4) tree takes $O(\log n)$ time