CSCI203

Algorithms and Data Structures

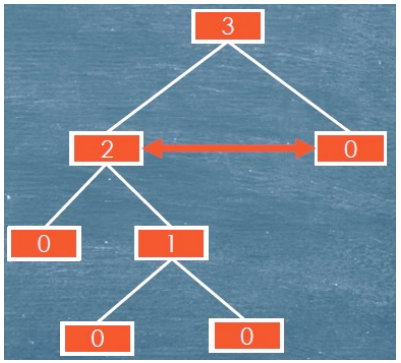
Trees (Part II)

Lecturer: Dr. Zuoxia Yu

Room 3.116

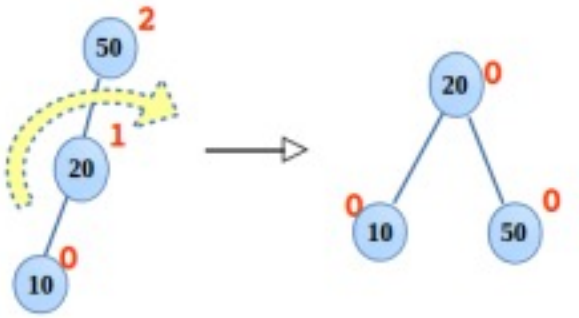
Email: zyu@uow.edu.au

AVL Trees - Review

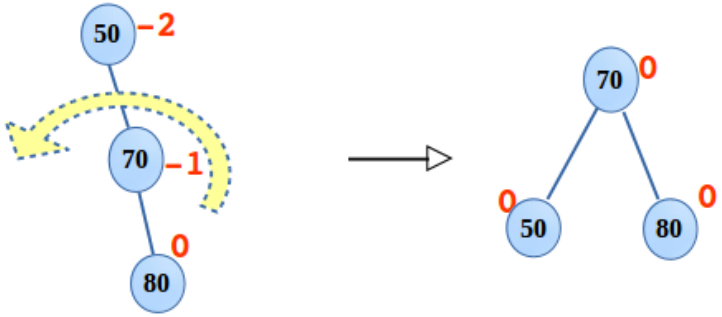


- AVL Trees are balanced binary search trees
- The balancing condition of AVL tree:
  - *Balance factor = height(Left subtree) – height(Right subtree),*
- It should be -1, 0 or 1. Other than this will cause restructuring (or balancing) the tree. Balancing performed is carried in the following operations
  - Right rotation (RR)
  - Left rotation (LL)
  - Left right double rotation(LR) (double right)
  - Right left double rotation(RL) (double left)

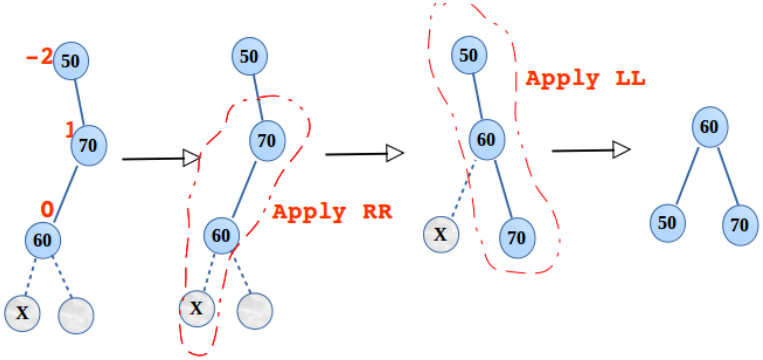
AVL Trees - Operations



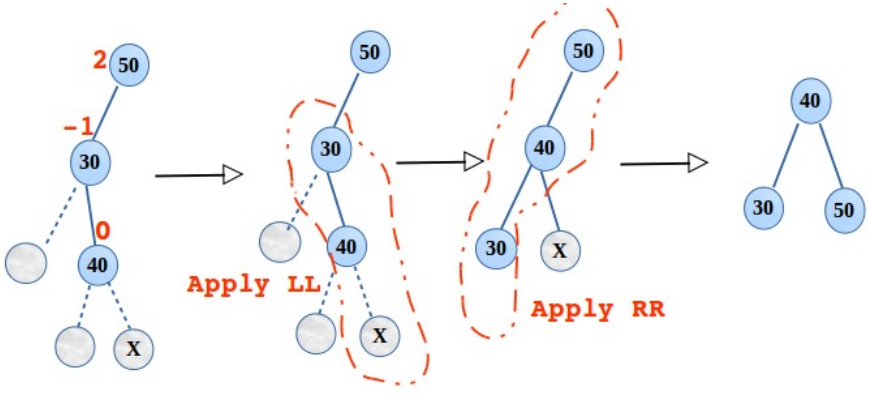
rotate_right



rotate_left



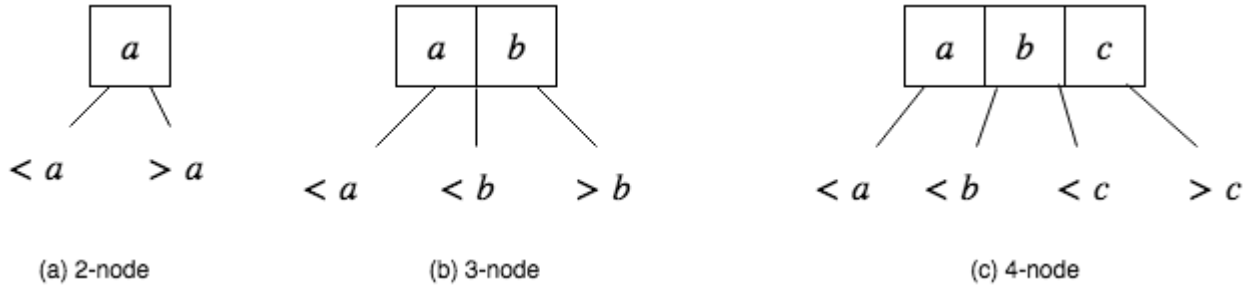
double_left



double_right

2-4 or 2-3-4 Trees - Introduction

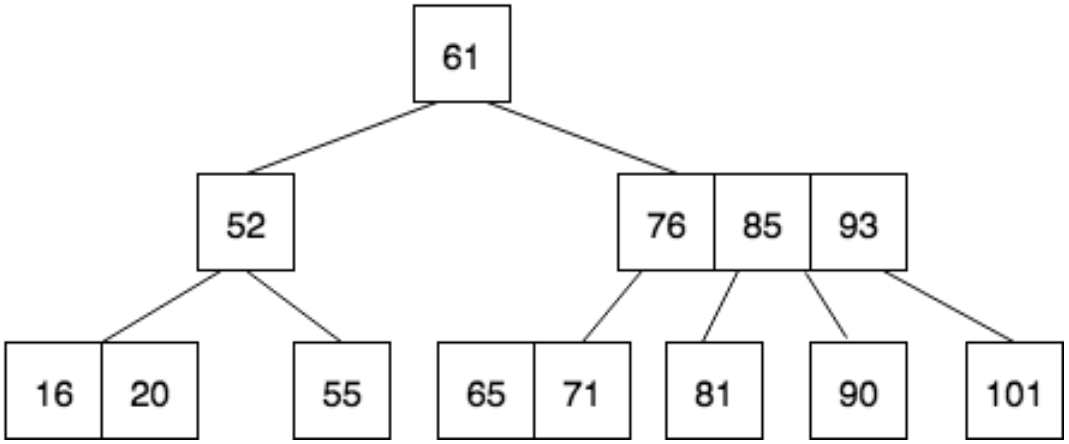
- A 2-3-4 tree is a balanced search tree having following three types of nodes.
  - 2-node has one key and two child nodes (just like binary search tree node).
  - 3-node has two keys and three child nodes.
  - 4-node has three keys and four child nodes.
- The reason behind the existence of three types is to make the tree perfectly balanced (all the leaf nodes are on the same level) after each insertion and deletion operation



(a) 2-node (b) 3-node (c) 4-node

2-4 Trees – Introduction…

- If a node has more than one keys (3-node and 4-node), the keys must be in the sorted order. This makes sure that the in-order traversal always yields the keys in sorted order



2-4 Trees – Introduction…

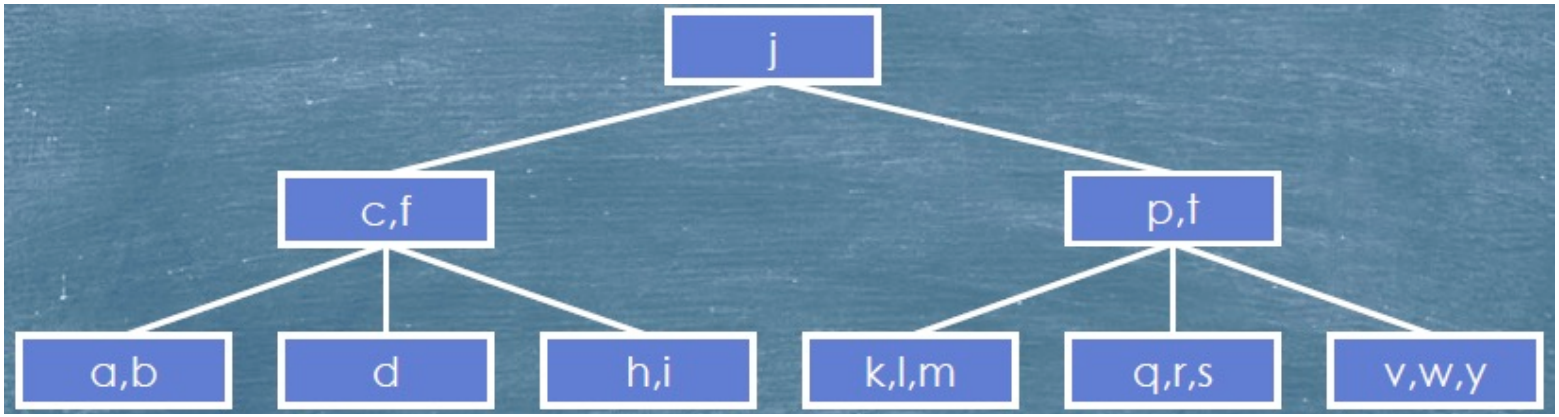
- Search
  - Compare the item to be searched with the keys of the node
  - Move to the appropriate direction. Unlike BST where we move either to the left child or to the right child, we need to make choice among three or four different paths.
- Insertion
  - A node cannot hold more than three keys. If a node is full before insertion, we split the node so that the new node can be inserted.
- Deletion
  - Delete is a bit tricker than insert operation.
  - Depending upon the location of the node containing the target to be deleted, we need to consider several cases.
  - I am going to explain each of the cases one by one later.

2-4 Trees

- A 2-4 tree is a balanced search tree and maintains its balance in a different way.
- It has the following properties:
  - Every internal node (except possibly the root) has between two and four children.
  - The keys within each node are ordered from smallest to largest.
  - Internal nodes have one less key than they have children.
  - Each such key is conceptually positioned between two consecutive children.
    - Its value is larger than the largest key in the subtree to its left and smaller than the smallest key in the subtree to its right.
  - All leaves of the tree are at the same depth.

An Example of 2-4 tree

- The following is an example of a 2-4 tree.



- Note:
  - only the keys are shown here. Each node also contains associated data.
  - a k-node contains k-1 keys and, if it is not a leaf, has k children.

Searching a 2-4 tree

- This process is similar to searching a binary tree

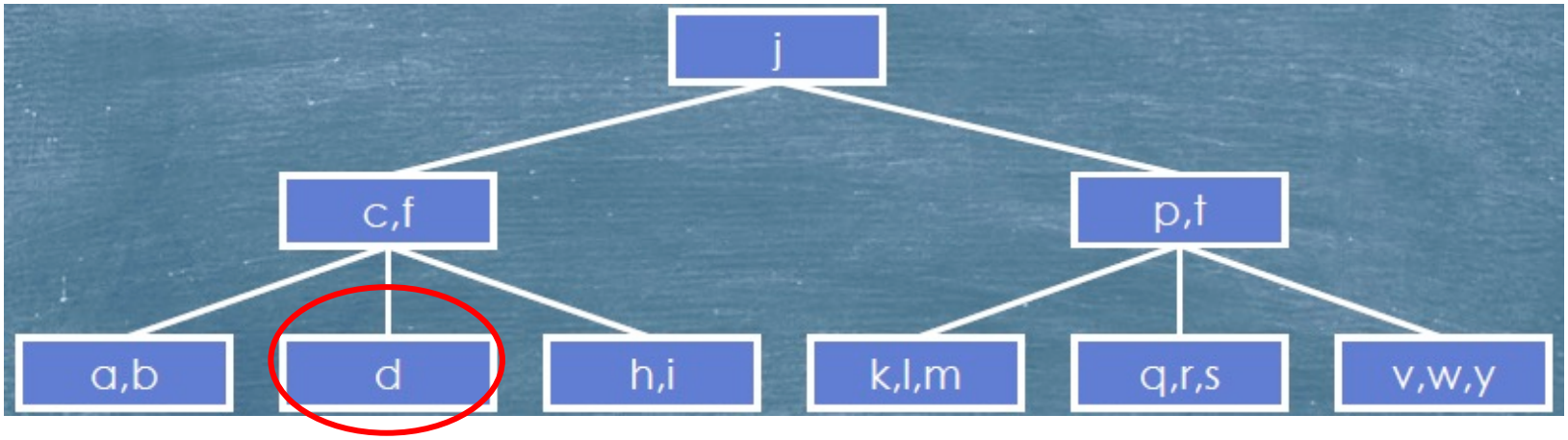
- If searching for $value$, compare value with each key
  - If $value == key$ return the associated data
  - If $value < key$ recursively search the subtree left of key
  - If $value > key$, repeat the process using the next key, if there is no next key, search the last subtree.

Insertion into a 2-4 Tree

- Find the leaf where the item is to be inserted
- Insert the item
- Update the node
  - Insertion into a 2-node -> 3-node
  - Insertion into a 3-node -> 4-node
  - Insertion into a 4-node -> 5-node
    - If an immediately adjacent sibling is not full, send a key from the parent down to this sibling, and a key up from the 5-node to the parent.
    - If all such siblings are full, split the node into two by passing the median key up to the parent, and relink subtrees if needed.
    - Repeat with the parent if necessary.
    - Create a new root layer if necessary.

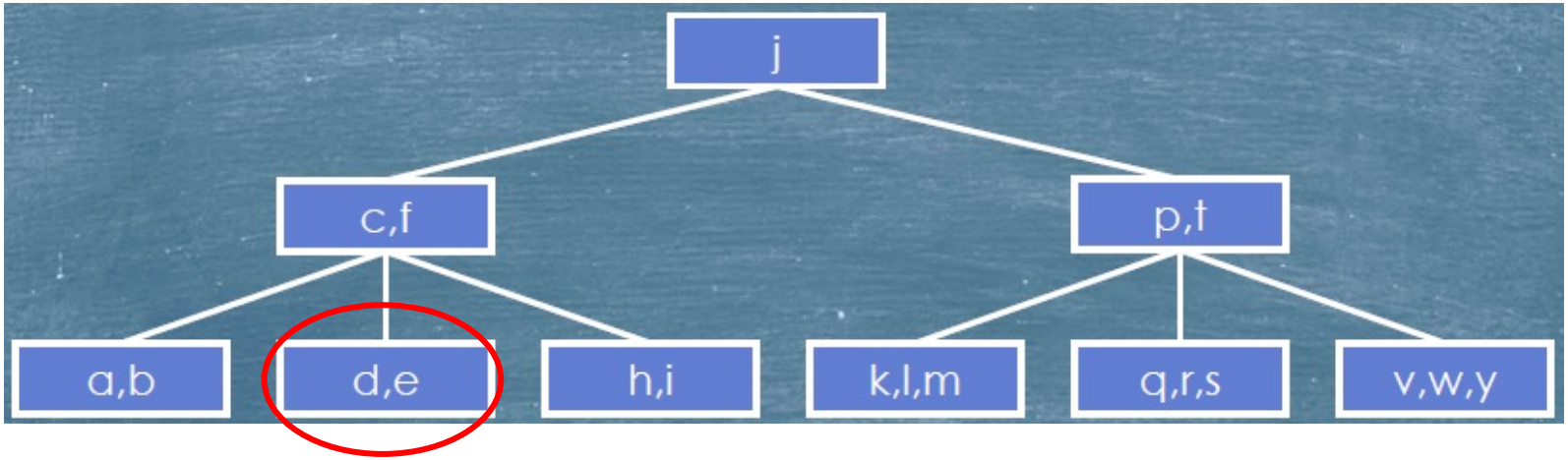
An Example of Insertion

- Insert 'e' into the following tree.



An Example of Insertion

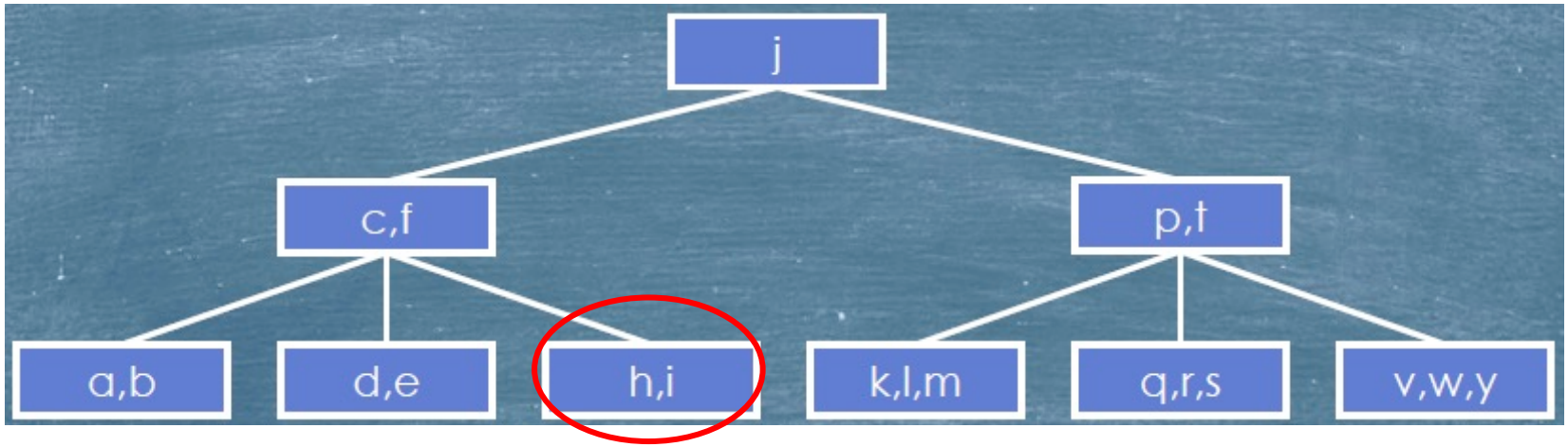
- Insert 'e' into the following tree.



- The 2-node becomes a 3-node.

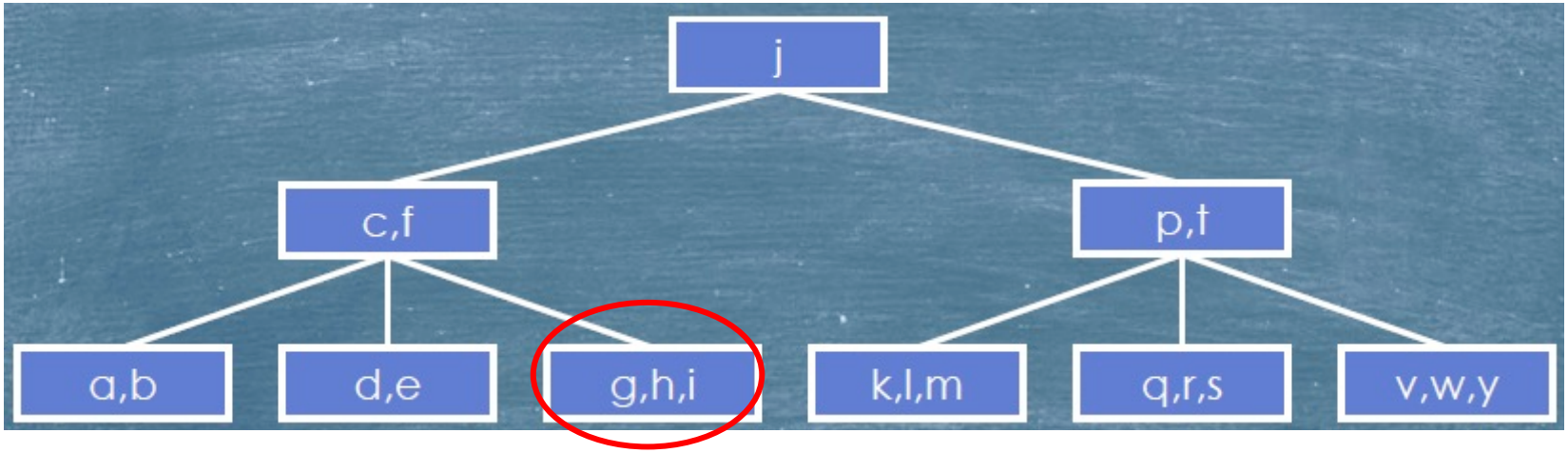
An Example of Insertion…

- Now, insert 'g'.



An Example of Insertion…

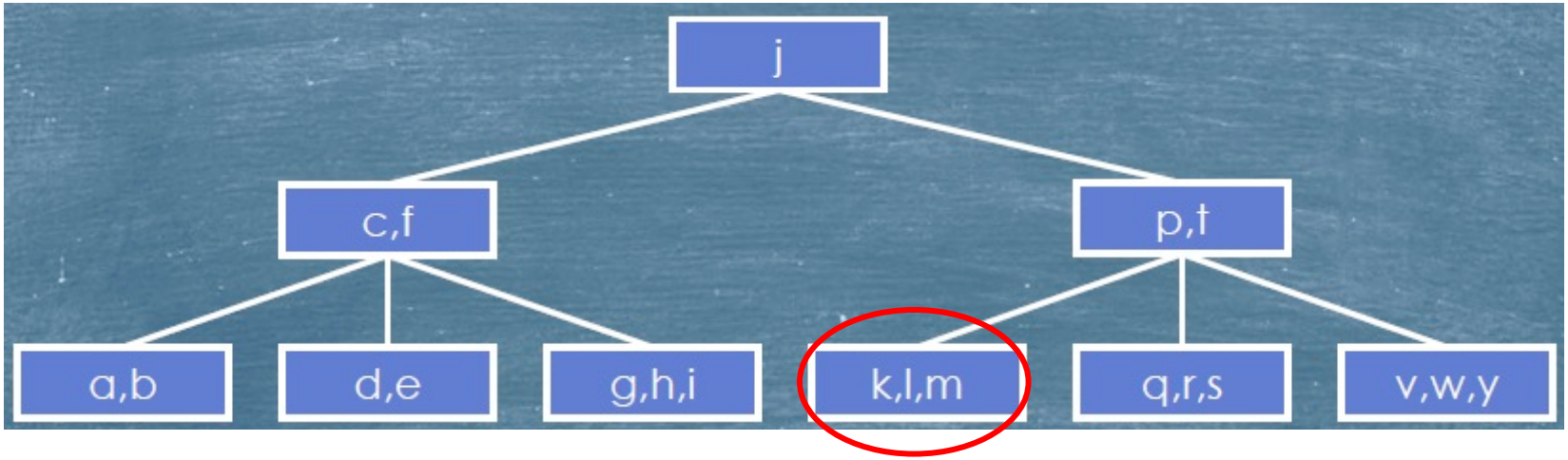
- Now, insert 'g'.



- The 3-node becomes a 4-node.

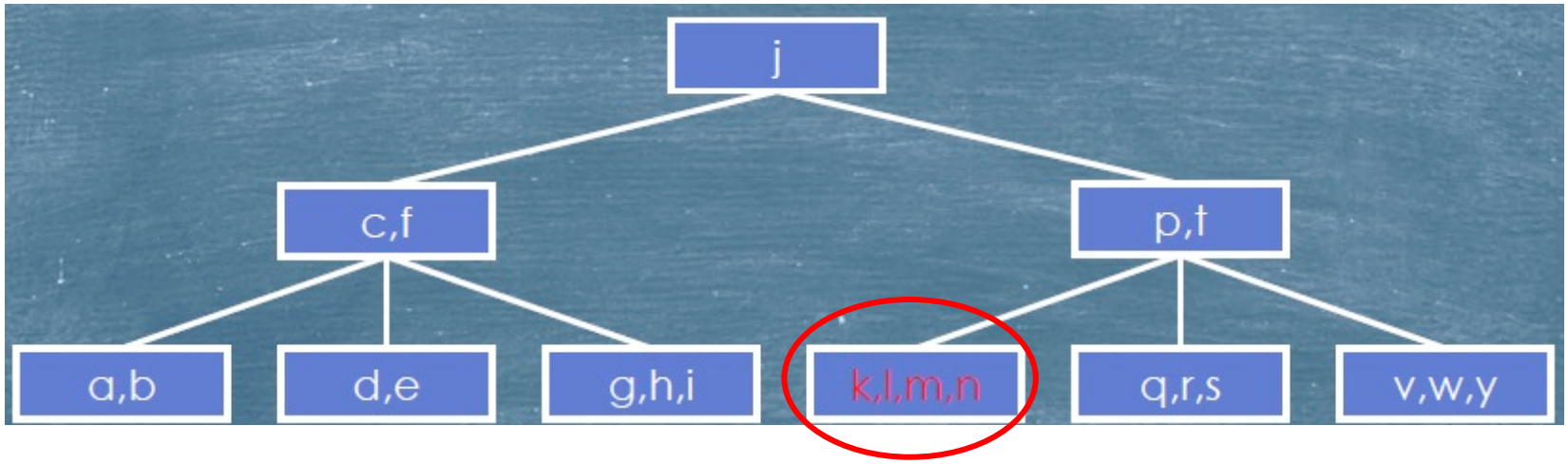
An Example of Insertion…

- Finally, insert 'n'.



An Example of Insertion…

- Finally, insert 'n'.



- The 4-node becomes a 5-node.
- This must be remedied.

Fixing a 5-node

- Consider the 5 node on the last slide:

  k,l,m,n

- To repair this, we must reduce the number of keys in the node.
- We could possibly do this by moving keys to its sibling.

  q,r,s

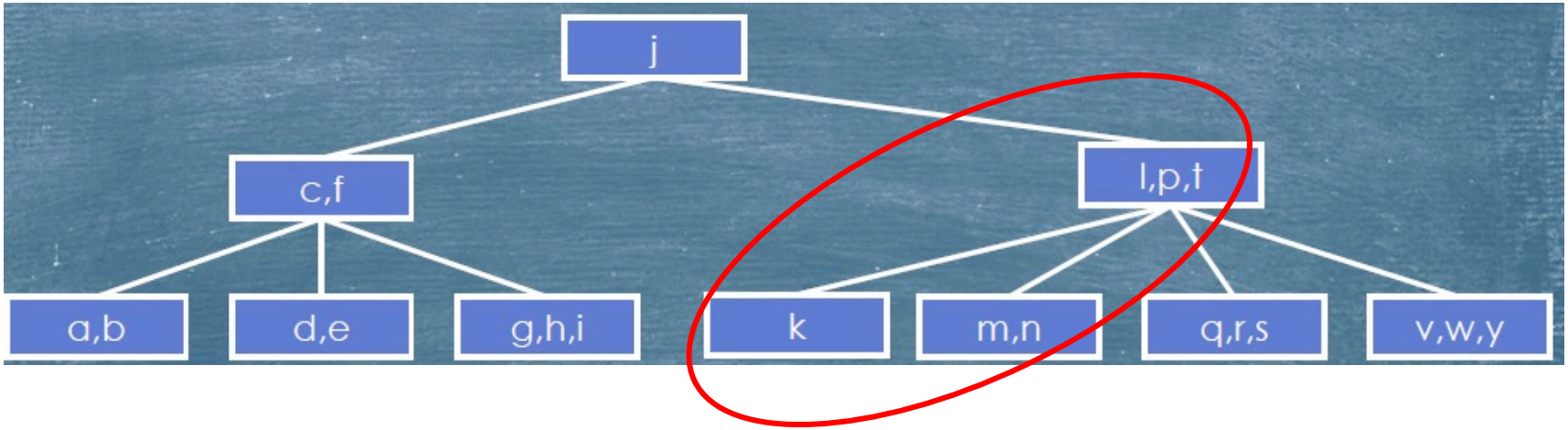
- The sibling is full so we split the node, instead.

  k m,n

- The median key 'l' moves into the node above.

Splitting a 5-node

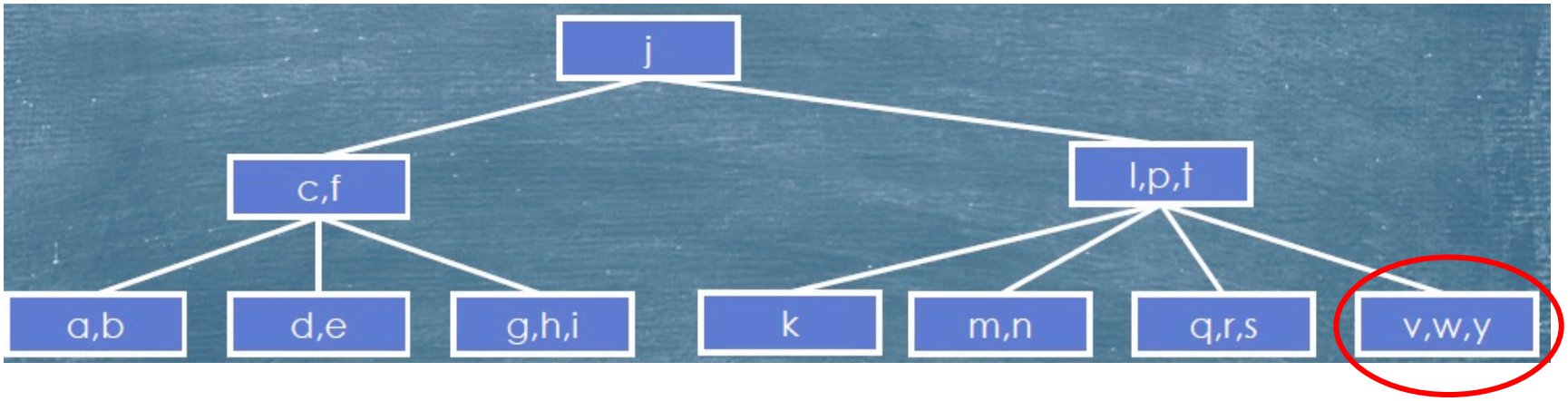
- So, the broken tree…



- …becomes this repaired tree

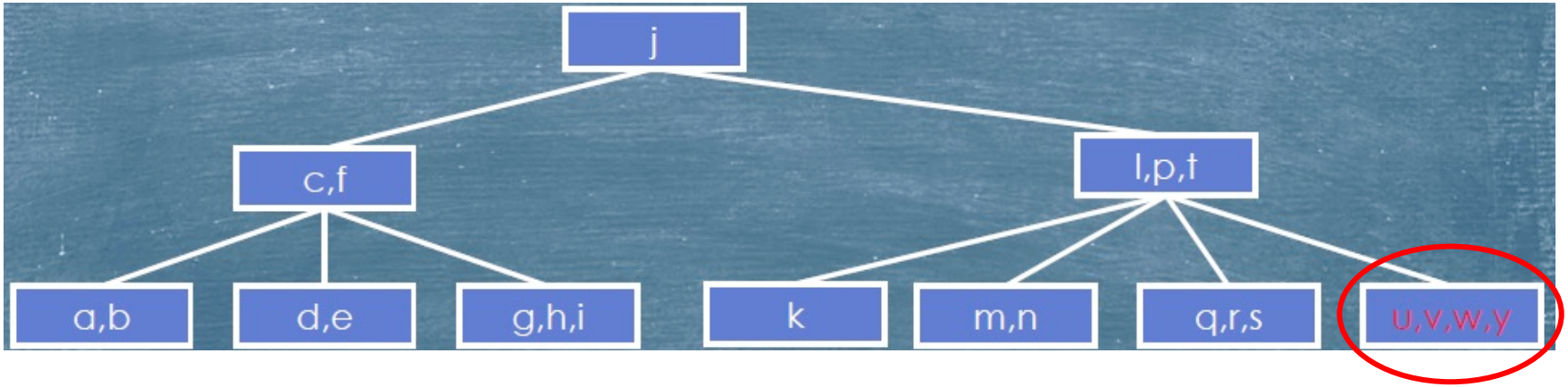
Another Insertion

- Now insert 'u'.



Another Insertion

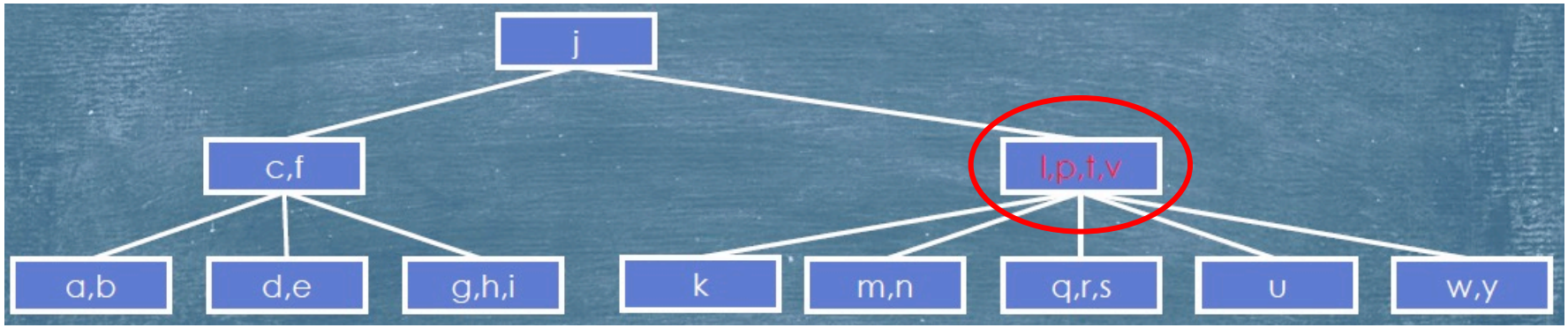
- Now insert 'u'.



- We have another 5-node.
- Again, the sibling is full.

Another Repair

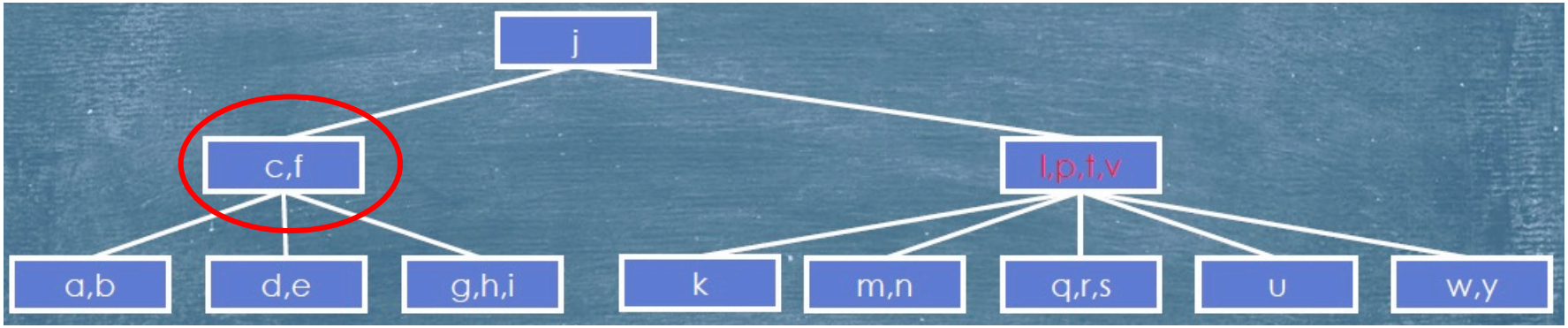
- So we split the node into two…



- But the parent is now a 5-node

Another Repair

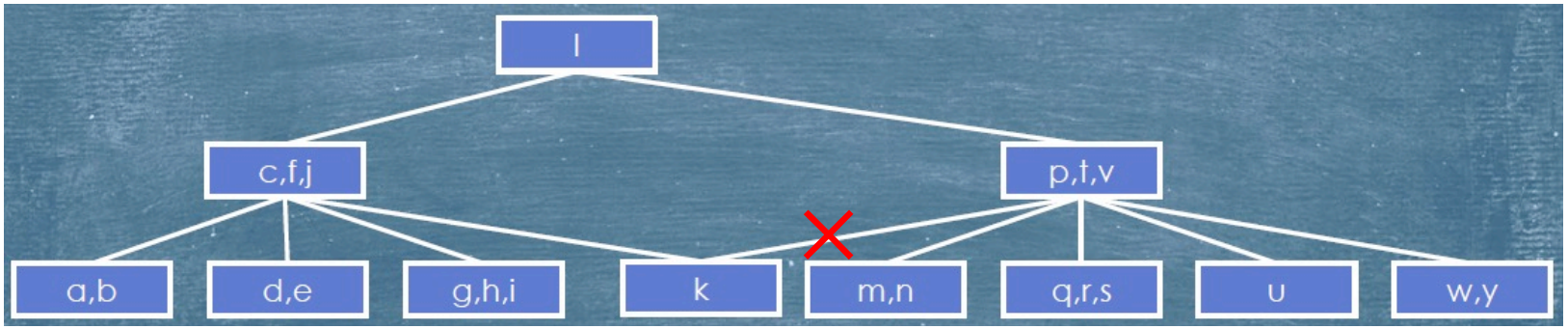
- This time, the sibling has spare room…



- …send key from parent down to sibling and key from node up to parent
- Note: the child node "k" moves across

Another Repair

- This time, the sibling has spare room…



- …send key from parent down to sibling and key from node up to parent
- Note: the child node "k" moves across

The Worst-Case Scenarios

- If the root node becomes a 5-node, split the root node and create a new root containing the median key of the old root.

- This increases the height of the tree by one

Deletion from 2-4 Trees

- Find where the item to be deleted is located.
- If this is an internal node, swap the item with its immediate in-order successor.
  - Repeat this until the item to be deleted is in a leaf node.
- Delete the item.
- Update the node.

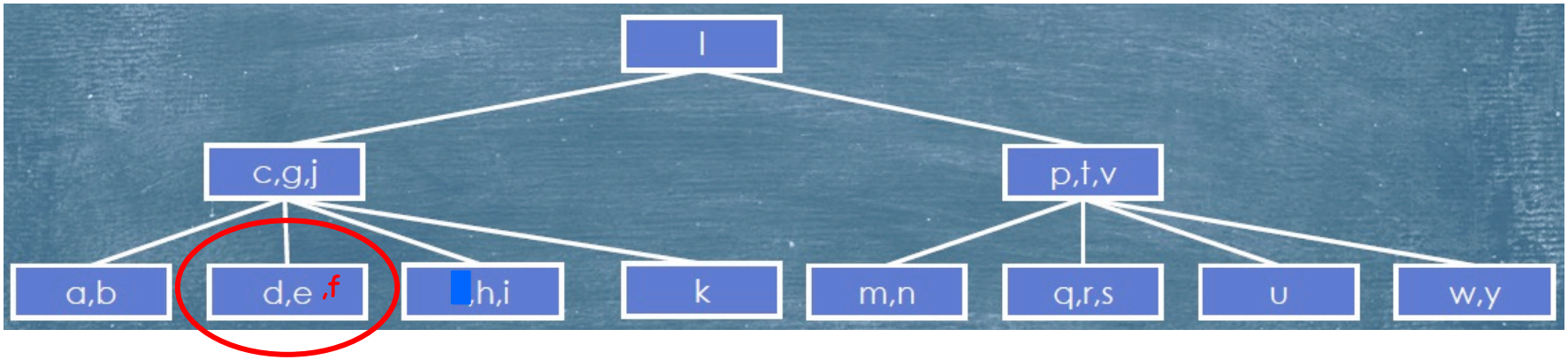
Deletion…

- Update the node:
  - Deletion from a 4-node -> 3-node.
  - Deletion from a 3-node -> 2-node.
  - Deletion from a 2-node -> 1-node.

- If the 1-node has an immediate sibling with more than one key, send a key from the sibling up to the parent, and a key from the parent down to the node and relink if necessary
- If not, remove the node and send the key down from the parent into a sibling node
  - This may cause the parent to become a 1-node
  - Fix this recursively
  - If the root becomes a 1-node, remove it.

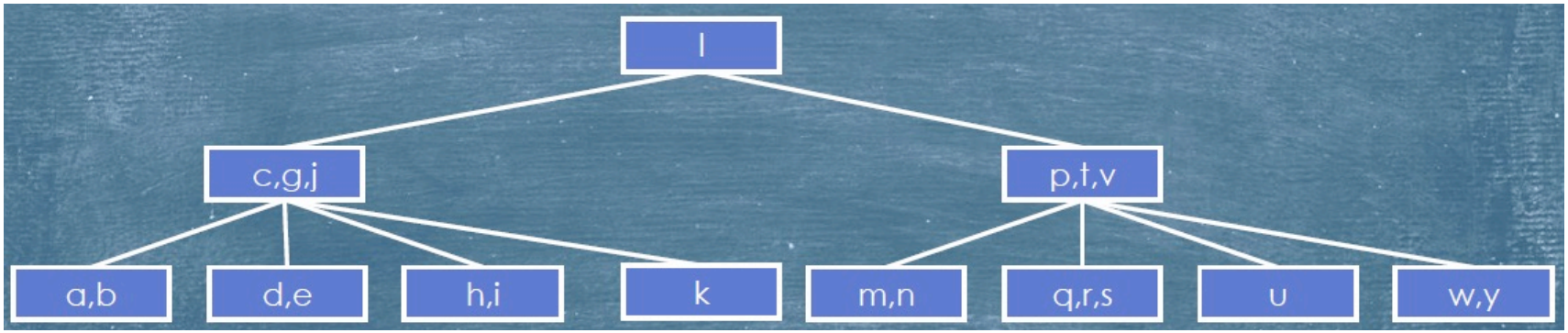
Deletion – Some Examples

- Delete 'f' from the following tree



Deletion – Some Examples

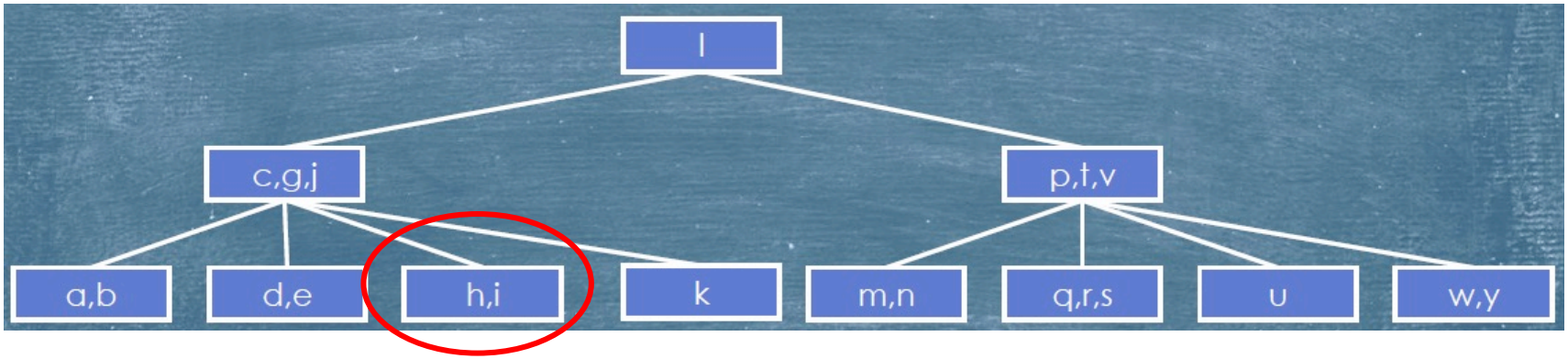
- The 'f' is now in a leaf node



- So we delete it, leaving a 3-node

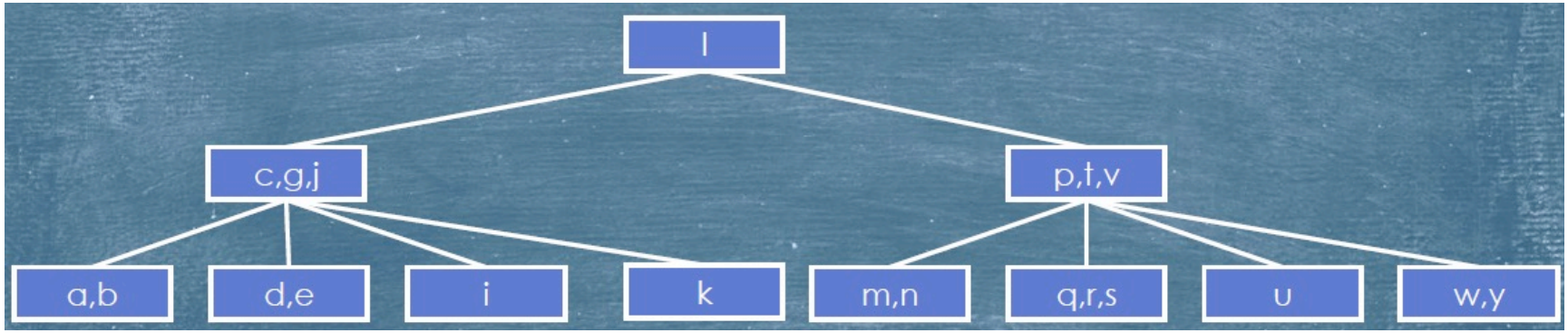
Deletion – Some Examples

- Now delete the 'h' key



Deletion – Some Examples

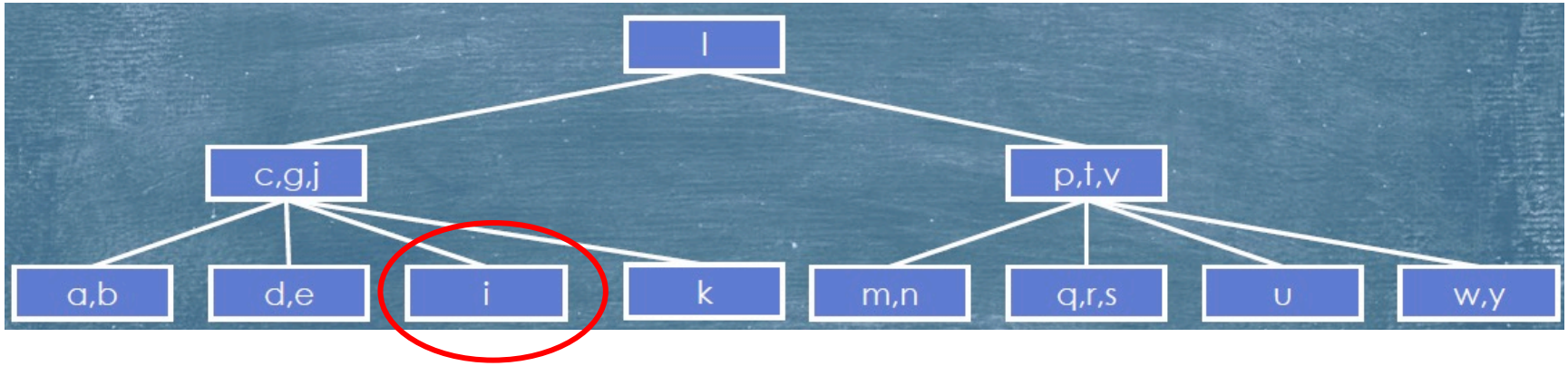
- Now delete the 'h' key



- It is already in a leaf so we just delete it
- The 3-node becomes a 2-node

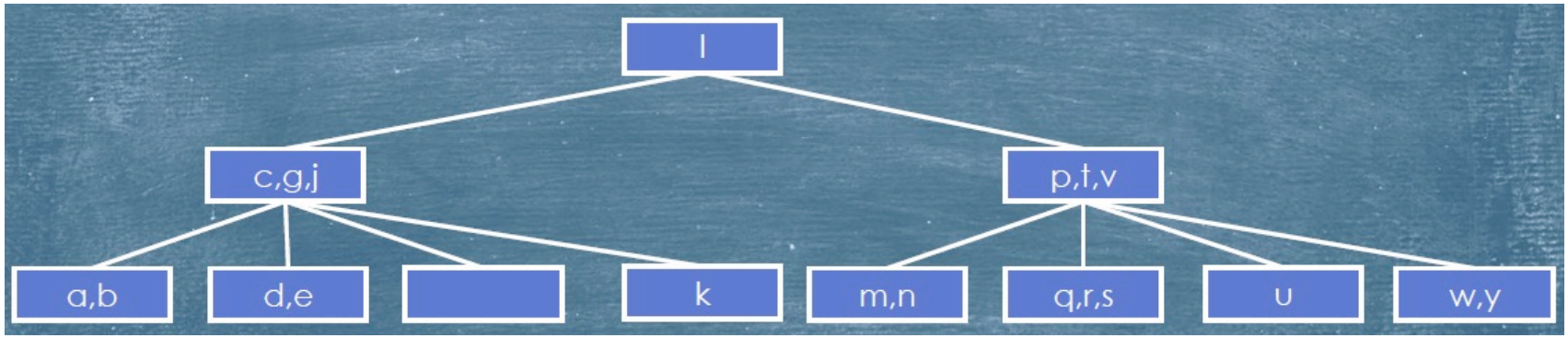
Deletion – Some Examples

- Now delete the 'i' key



Deletion – Some Examples

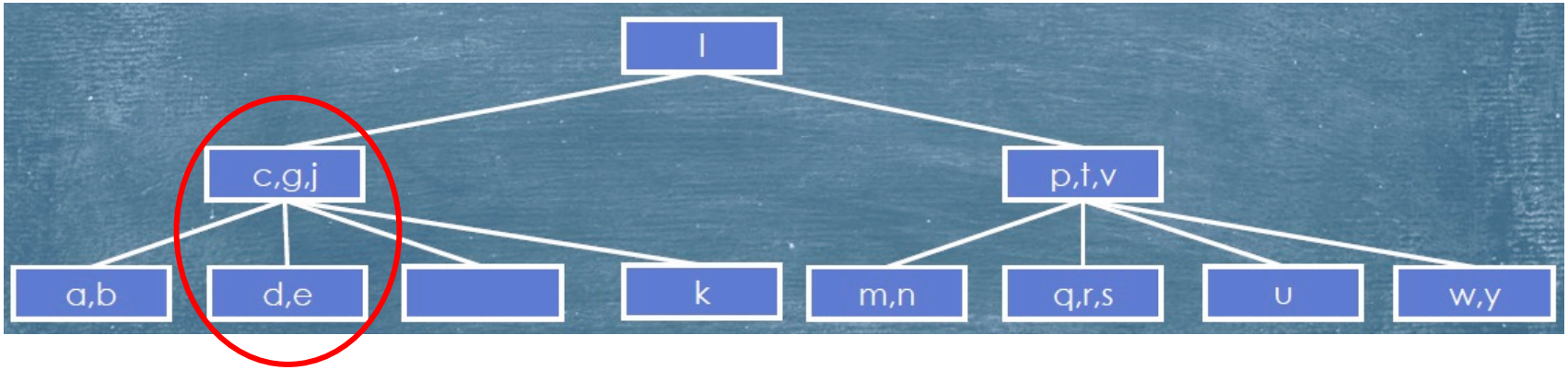
- Now delete "i" key



- It is already in a leaf so we just delete it
- The 2-node becomes a 1-node and we have a broken tree

Deletion – Some Examples

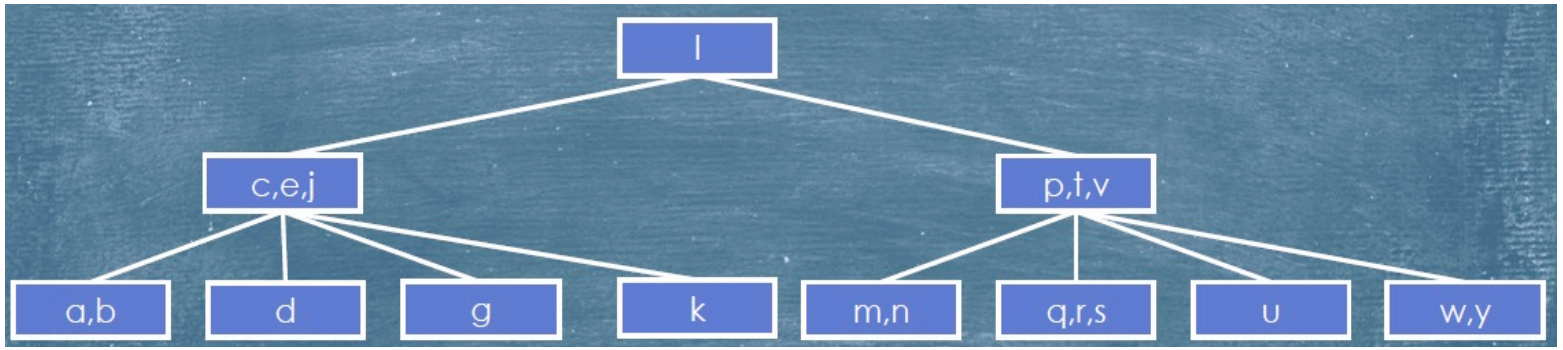
- The empty node has a sibling with more than one key



- Send a key from the sibling up to the root and a key from the root down to the (empty) 1-node

Deletion – Some Examples

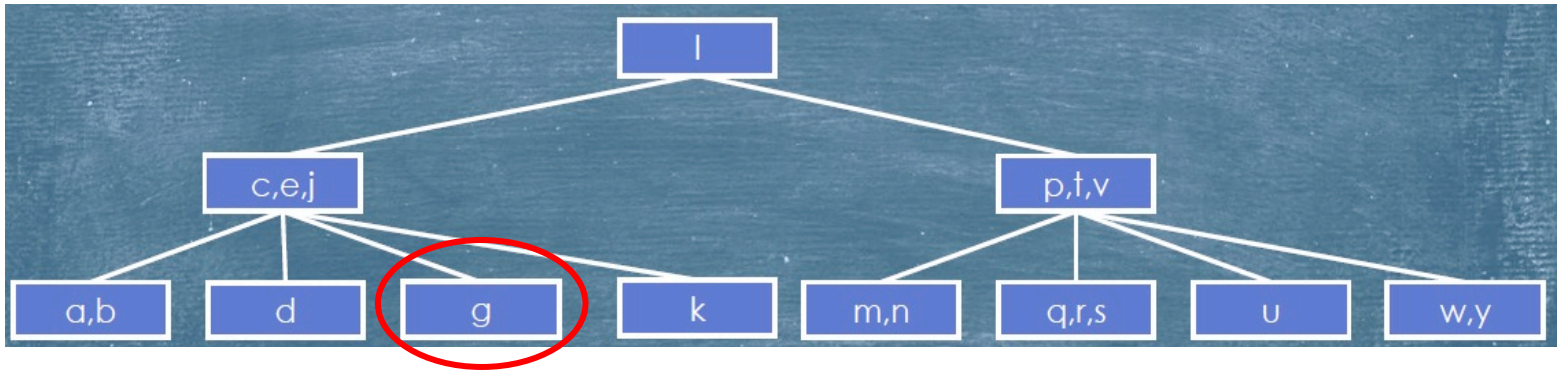
- The empty node has a sibling with more than one key



- Send a key from the sibling up to the root and a key from the root down to the (empty) 1-node

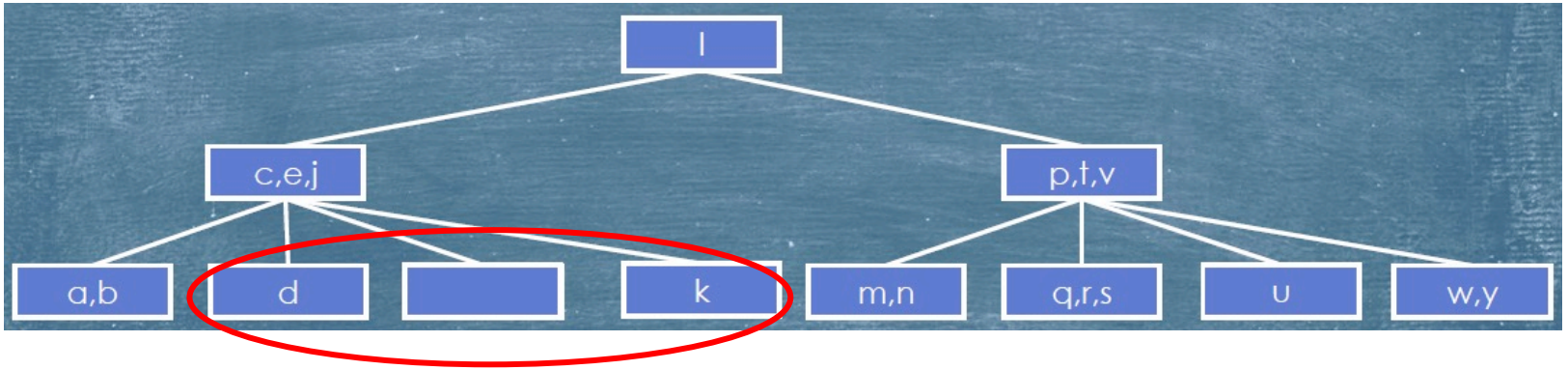
Deletion – Some Examples

- Delete the key 'g'



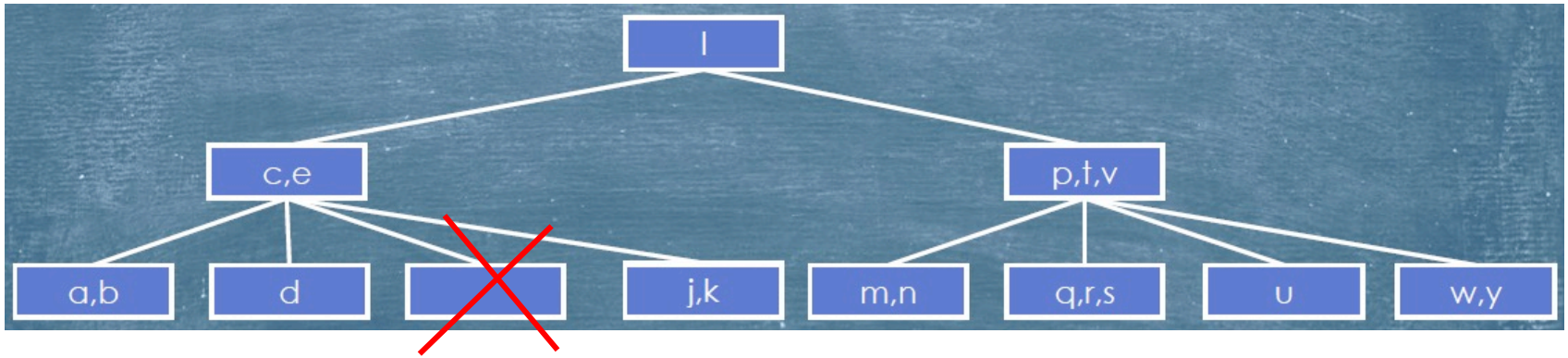
Deletion – Some Examples

- Delete the 'g' key



- This time no sibling has spare keys

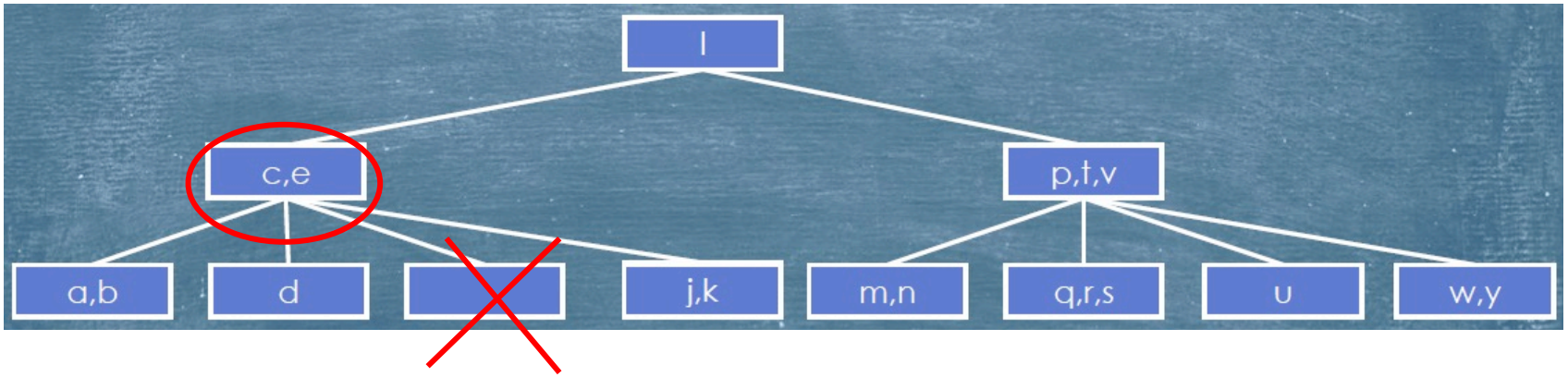
Deletion – Some Examples



- Remove the node and send a key from the parent down to a sibling
- And we are ok again

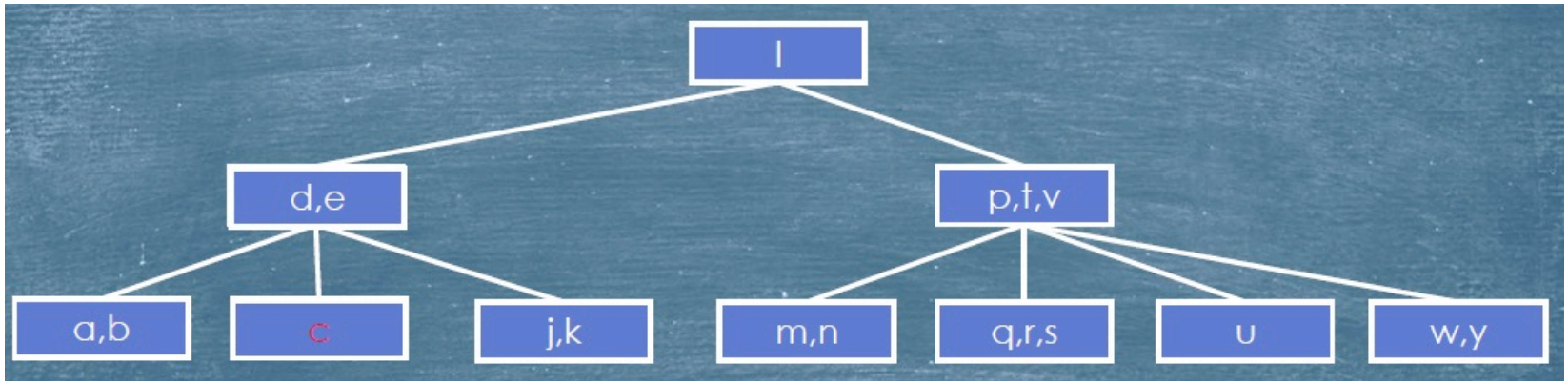
Deletion – Some Examples

- Delete the key 'c'



Deletion – Some Examples

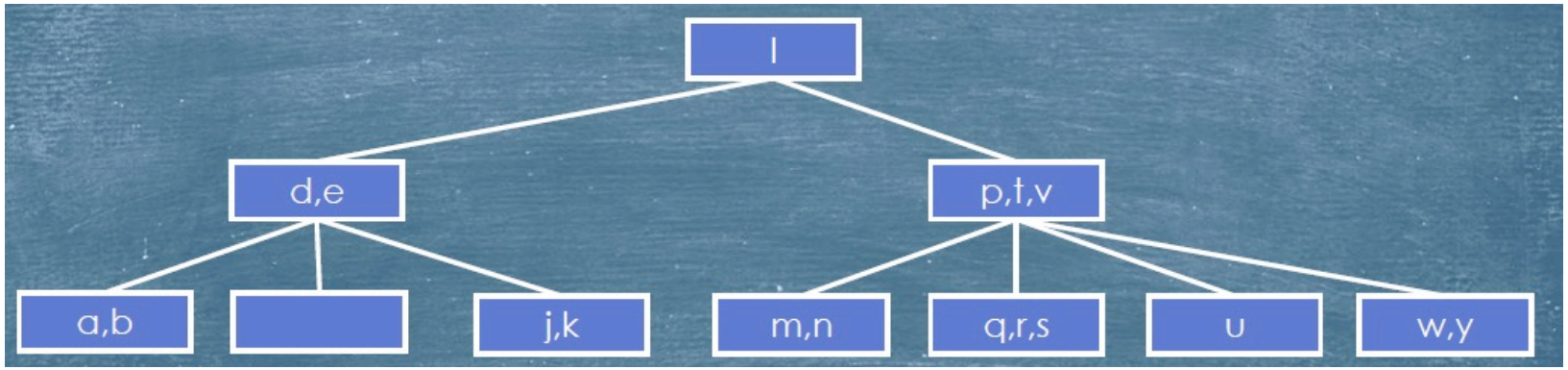
- Delete the 'c' key



- It is internal, swap with 'd'

Deletion – Some Examples

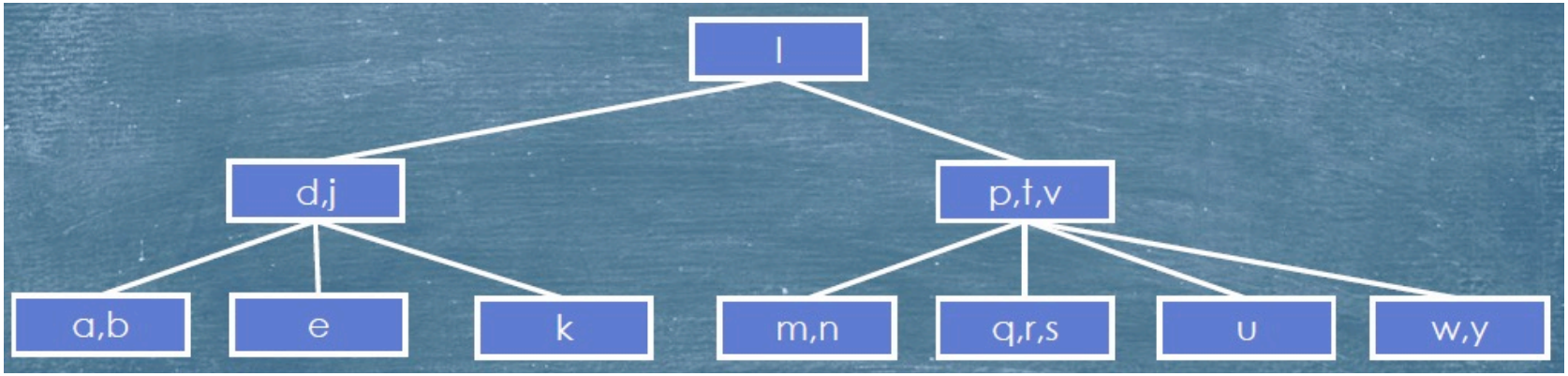
- Delete the 'c' key



- We have a 1-node
- A sibling has more then one key

Deletion: Some Examples

- A sibling has more than one key



- Move a sibling key up and a parent key down
- And we are done

2-4 Tree Efficiency

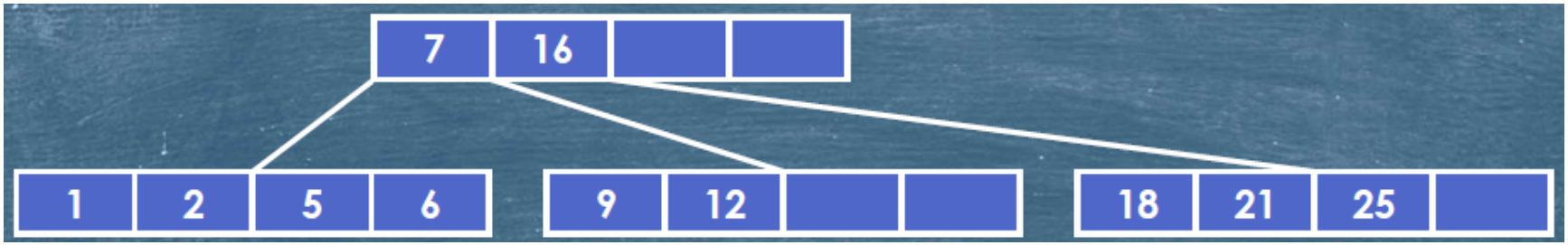
- Height of tree is $O(\log n)$
- Searching:
  - Each node checked takes $O(1)$
  - Search takes $O(\log n)$
- Insertion
  - Split takes $O(1)$ at each level
  - At most $\log n$ splits (1 per level)
  - Insertion takes $O(\log n)$
- Deletion
  - Merge takes $O(1)$
  - At most $\log n$ merges
  - Deletion takes $O(\log n)$

B-trees – tree on disk

- Generalizing 2-4 Trees - An `m-ary` search tree
- Created by Rudolph Bayer and Ed McGreight
- A B-tree of order m has the following additional properties
  - The root is either a leaf or has at least 2 children
  - All other internal nodes have at between $\lceil m/2 \rceil$ and m subtrees.
  - All non-root nodes have between $\lceil m/2 \rceil - 1$ and $m - 1$ keys inclusive.
- All leaves are at the same level

An Example of B-Tree

- This is a B-Tree with order 5 (m=5)
- Again, only the keys are shown.



- Nodes have between 2 and 4 keys.
- Internal nodes have between 3 and 5 children.

Searching B-Trees

- Similar to searching a binary tree
- If searching for value, compare value with each key
  - If $value == key$ return the associated data
  - If $value < key$ recursively search the subtree left of key
  - If $value > key$, repeat the process using the next key, if there is no next key, search the last subtree.

Inserting into B-trees

- This process is analogous to insertion into 2-4 trees.
  - Find the leaf where the item is to be inserted
  - Insert the item
  - If the node overflows (now has m keys)
    - Split the node into two by passing the median key up to the parent
      1. Repeat with the parent if necessary.
      2. Create a new root layer if necessary.

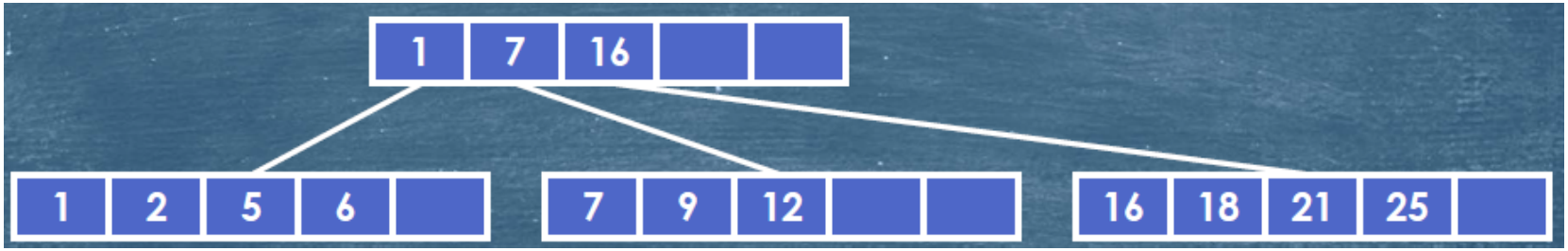
Deleting from B-Trees

This process is analogous to deletion from 2-4 Trees.

- Find the key to be deleted.
- If it is not a leave, swap it down to a leaf.
- Delete the key from the leaf.
- If the node underflows (now has $(m/2) - 1$ keys):
  - If a sibling has spare keys, borrow from the sibling and adjust the parent node.
  - If not, merge with a sibling and reduce the number of parent keys by one.
    - This may result in the parent being underfilled.
    - In this case repeat the procedure.

Variations on B-Trees

- B+ Tree:
  - Data associated with keys are only stored in the leaf nodes.
  - Each internal node contains copies of the first key of each child .
  - Our previous example now looks like this:



- Note: the nodes now contain between 3 and 5 keys.

Why do we need B-Trees?

- BST guarantees $O(\log n)$ running time for all the operations and are much more easy to implement

- Why do we still need B-trees?

Why do we need B-Trees?

- An application scenario
  - We want to store very large information into a tree
  - The information is too large to fit into memory
    - the program cannot load the whole tree into the memory
    - It can only load one node at a time from disk and then process
  - Reading from and writing to a disk is an expensive operation
  - We need to minimize the possible disk operations
- There is where B-tree come into play
  - has more branching factors and, hence, small height
  - For a tree with small height, it takes less number of operations to go from root to any leave node
  - A binary tree (e.g. AVL tree) is much deeper than a B-tree

B-Trees in Real Applications

- Although small (m=5) B-Trees are useful examples, in real applications m can be 100 or even larger.
- If m=100 we can store 1,000,000 records in a tree that has only 3 levels!
- Thus, we need only examine at most 3 nodes to find a given record.
- Compare this with 20 nodes using a BST.
- The extreme case is a 1D array, i.e., all records are kept by the root.

B-Trees and Databases

- Databases frequently contain millions, if not billions, of records.
- Often, the records are stored on disk, so the access time is significant.
- B-Trees, because they allow access with a small number of steps (disk reads), are frequently used as the storage mechanism for such databases.

B-Trees and Databases

- The Following table shows the number of records for each level of B-Tree.
- We assume m=100.

1	2	3	4	5
100	10,000	1,000,000	100,000,000	10,000,000,000

- As you can see, we can search a database of 10 billion records with only 5 disk accesses!

Related References

- Introduction to the Design and Analysis of Algorithms, A. Levitin, 3rd Ed., Pearson 2011.
  - Chapters 6.3
- Introduction to Algorithms, T. H. Cormen, 3rd Ed, MIT Press 2009.
  - Chapters 13.3, 18
- Online https://algorithmtutor.com/Data-Structures/Tree/2-3-4-Trees/