

# Red/Black Trees

## Copyright and AI Notice

These slides are intended for studying and teaching at Nazarbayev University in Astana, Kazakhstan.

If you want to redistribute or adapt these slides for different purposes, check the license first.

No generative AI was used in the preparation of these slides.

©Hans de Nivelles, 2025

Binary search trees can be used for the implementation of dictionaries or sets.

If the tree is balanced, complexity of lookup, insert, remove equals  $O(\log n)$ , where  $n$  is the number of keys in the tree.

In general, search trees perform worse than hash maps, because the basic operations for hash map take constant time. Moreover, walking through a tree causes many cache misses.

Trees should be used only in applications where the order matters, e.g. if one wants to search in a phone book by a prefix of the name one is looking for.

## Rotations

If the keys that are inserted into the tree are already ordered, the resulting tree will be unbalanced.

In order to prevent this, there exist different strategies that rearrange the tree in such a way to keep it sufficiently balanced.

AVL-trees were invented in 1962 by Georgy Adelson-Velsky and Evgenii Landis. It seems that they are not used any more.

In the rest of these slides I explain red/black trees.

## Rules of Red/Black Tree

A red/black tree is a binary search tree, in which the nodes are either red or black. The coloring must fulfill the following conditions:

- The root of the tree is black.
- All children of a red node are black.
- If the left child of a node is red, then the right child is also red.
- Every path from the root to a leaf contains the same number of black nodes.

Red/black trees allow  $\log(n)$  lookup, insertion and deletion. They are used in `std::map` and in `java.util.TreeMap`.

## Understanding the Rules

The rules of red/black trees are hard to understand, but not if you understand where they come from.

Red/black trees are a clever implementation of 2, 3, 4-trees.

## 2, 3, 4-Trees

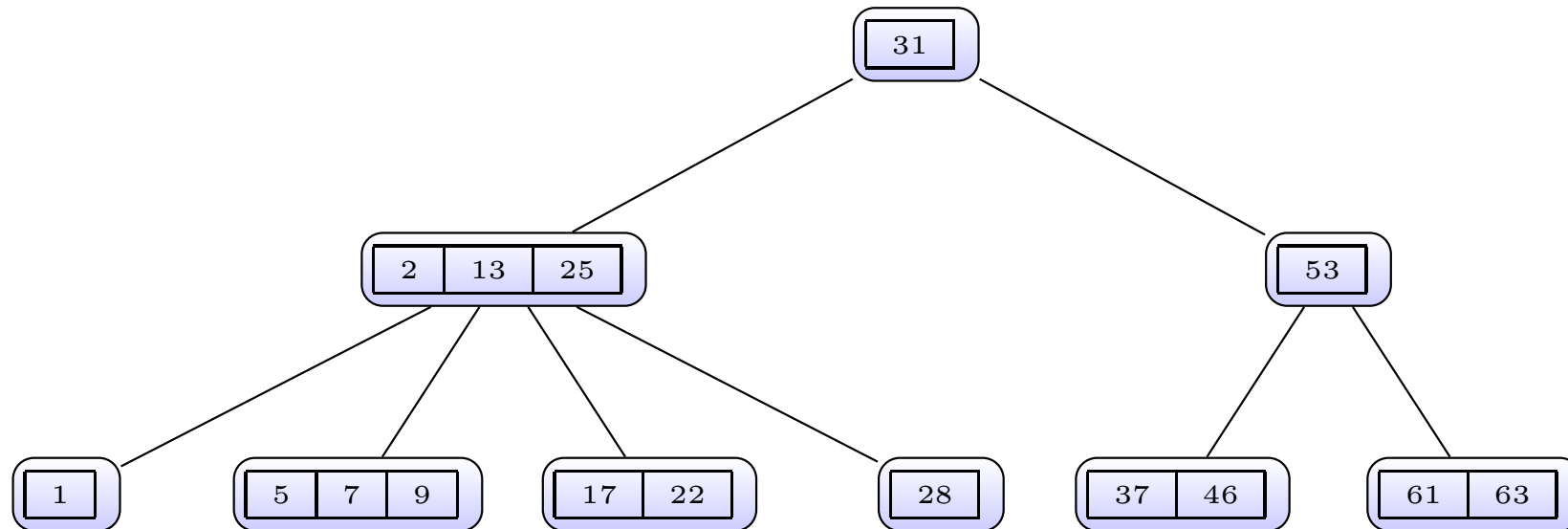
2, 3, 4-trees are a generalization of binary trees, fulfilling the following conditions:

- Every node in the tree contains 1, 2 or 3 keys, and the number of children is always one more than the number of keys.
- For every node  $n$ , the following holds:
  - All keys occurring in the first subtree of a node are smaller than the first key in this node.
  - All keys occurring in the last subtree of a node are bigger than the last key in this node.
  - Every subtree that starts between two keys of a node contains only keys that lie between these two keys.
- All paths from the root to a leaf in tree have the same length.

2, 3, 4-trees are sufficiently balanced to ensure  $O(\log n)$  lookup time.

During insertion or deletion we will permute the tree in such a way that it remains a 2, 3, 4-tree. These permutations take place on a single branch only (almost), and hence are performed in time  $O(\log n)$ .

## Example of a 2, 3, 4-Tree



It is probably not clear on the picture but the subtree containing 5, 7, 9 starts between 2 and 13.

Similarly the subtree containing 17, 22 starts between 13 and 22.

## Permutations

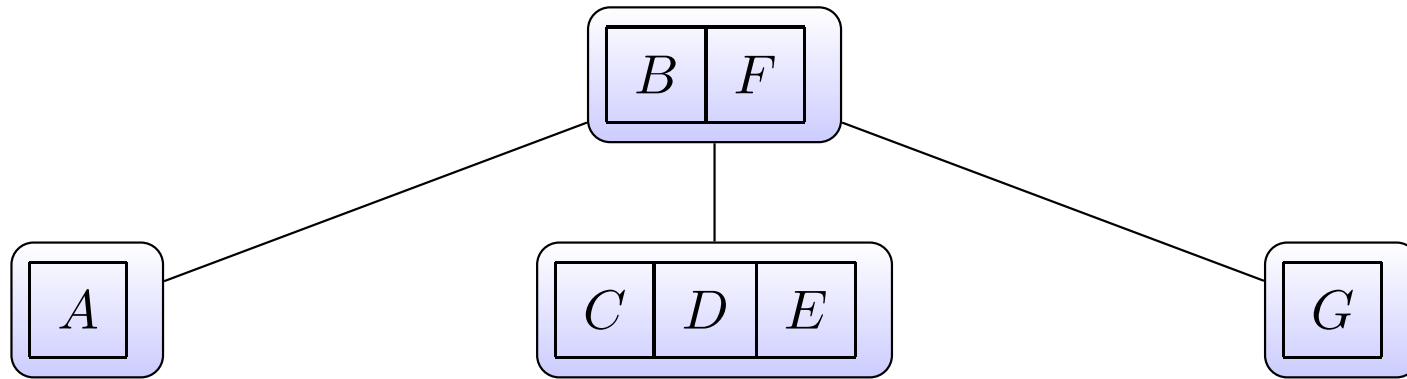
We will explain how keys are inserted into a 2, 3, 4-tree.

In order to ensure that all paths have the same length, permutations in the tree are necessary.

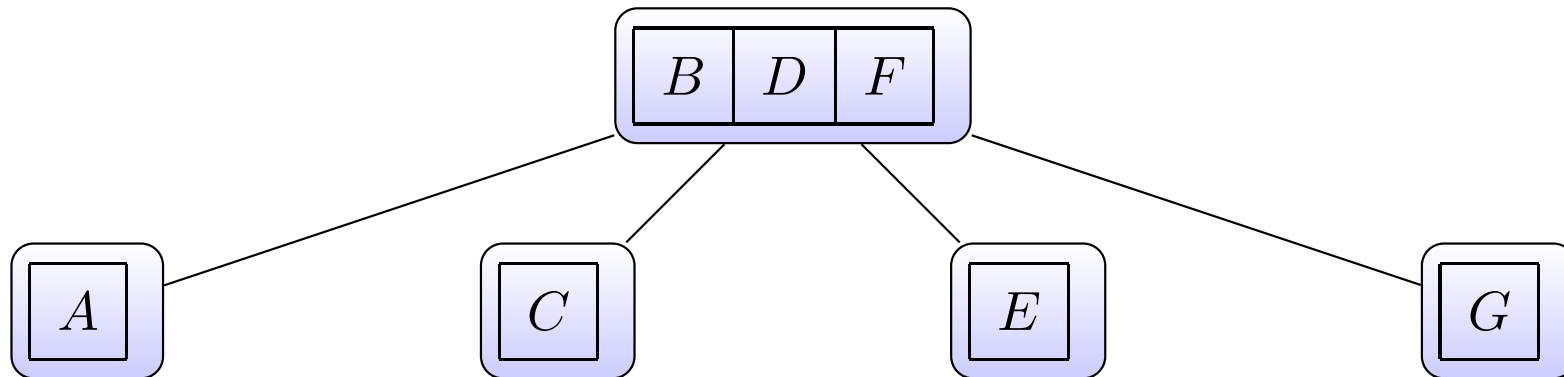
We will first explain the permutations needed for insertion, after that we give the insertion algorithm.

## Lifting Middle Value

If a node contains 3 values while its parent contains 1 or 2, the middle value can be lifted:



becomes



This is also possible when 

<i>C</i>	<i>D</i>	<i>E</i>
----------	----------	----------

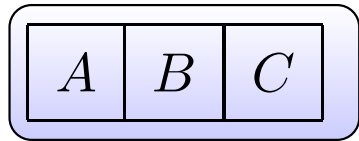
 is the first or last child.

## Lifting Middle Value (Root Case)

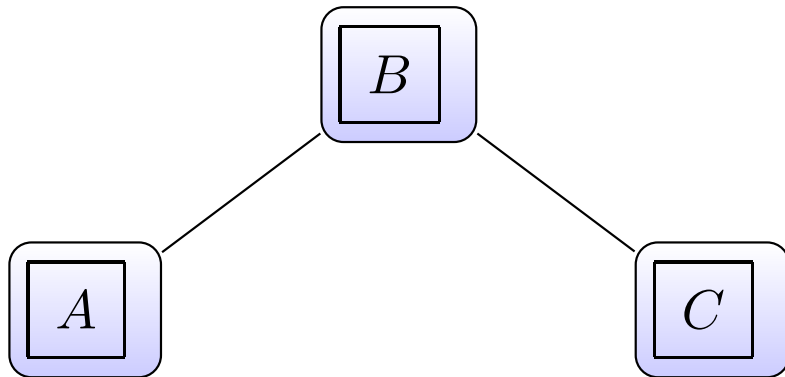
In case 

<i>A</i>	<i>B</i>	<i>C</i>
----------	----------	----------

 is the root node, we can lift as follows:



becomes



This rotation increases the depth of the tree by one.

## Inserting into 2,3,4-Tree

We can describe insertion. Assume we want to insert  $x$  into the tree:

- Find a leaf node where  $x$  can be inserted. Call this node  $n$ . Call **ensure12**( $n$ ) with node  $n$ . After that, insert  $x$  into  $n$ .
- Procedure **ensure12**( $n$ ) ensures that node  $n$  contains 1 or 2 keys. It is defined as follows:
  1. If  $n$  already has 1 or 2 keys, then return.
  2. Otherwise, if  $n$  has a parent  $p$ , recursively call **ensure12**( $p$ ). This ensures that the middle element of  $n$  can be lifted to  $p$ . Do the lifting and return. (See slide 11)
  3. If  $n$  has no parent, lift the middle element into a new node and return. (See slide 12)

## Removing from 2,3,4-Tree

Assume that we want to remove  $x$  from the tree. First find  $x$ .

If  $x$  does not occur, then we are finished.

If  $x$  is found in a leaf node, then call this node  $n$ , and continue on slide 19.

Otherwise, proceed as follows:

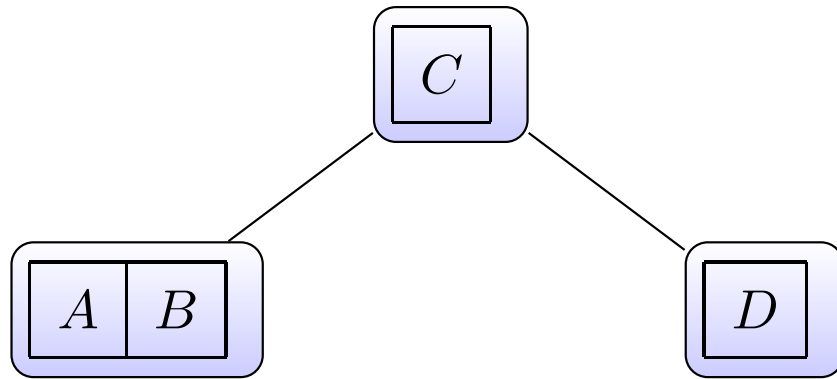
Enter the child right after  $x$ , and from there, always take the leftmost child until a leaf node is reached. Swap  $x$  with the first key in this node.

Now  $x$  stands on the wrong place in the tree, but that is no problem, because it will be deleted anyway. The procedure continues on slide 19.

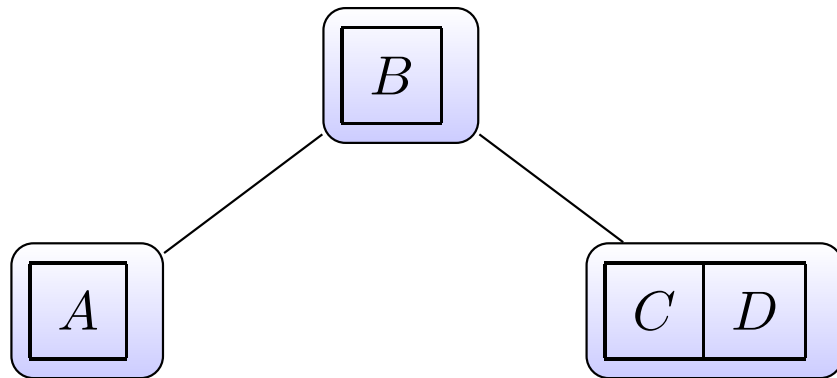
On the next slides we describe the permutations that are needed for removal.

## Stealing from Left Neighbour

If our left neighbour has more than 1 key, and we have less than 3, we can steal a key:



becomes



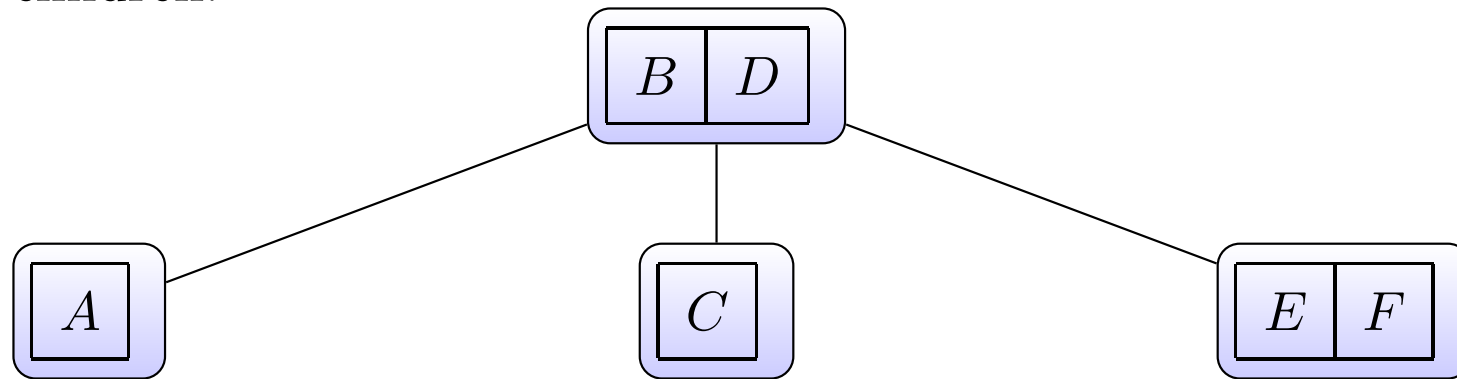
Stealing does not depend on the number of keys in the parent node.

## Stealing from Right Neighbour

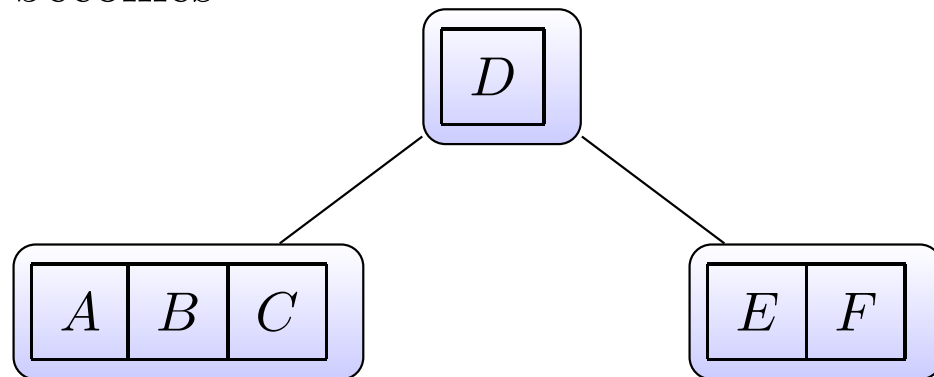
If our right neighbour has more than 1 key, and we have less than 3, we can steal a key from the right. It is the same operation as on the previous slide, but in the opposite direction.

## Merge Neighbours

If two neighbours have 1 key, and the parent has more than 1 key, the in-between key can be moved downwards while merging the children:



becomes



This is also possible when the parent node has 3 keys.

## Merge Neighbours (Root Case)

This is the opposite operation of slide 12. It decreases the depth of the tree by one.

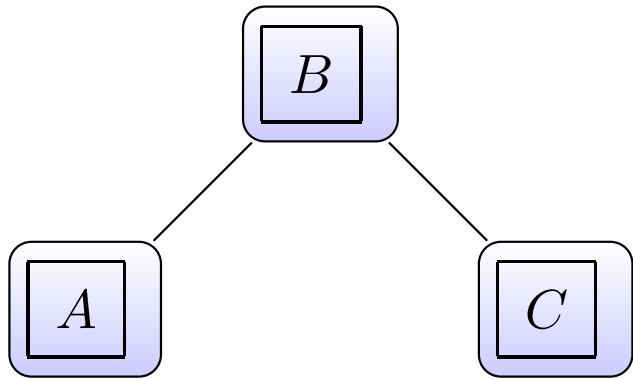
## Removing from 2,3,4-Tree (Continued)

Assume that we want to remove  $x$  from node  $n$ .

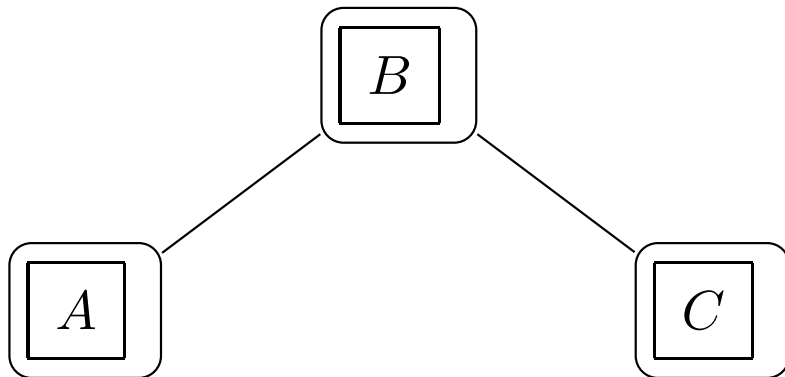
- At this point, we know that  $n$  is a leaf node. Call **ensure23**( $n$ ). Now  $x$  can be safely removed.
- Procedure **ensure23**( $n$ ) ensures that node  $n$  has 2 or 3 keys. It is defined as follows:
  1. If  $n$  already has 2 or 3 keys, then return.
  2. Otherwise, check if we have a neighbour with 2 or 3 keys. Steal from this neighbour (See slide 15 or 16) and return.
  3. If our parent is the root of the tree and contains one key, then merge  $n$  with its neighbour and return. (See slide 18).
  4. Otherwise, call **ensure23**( $p$ ), and after that merge  $n$  with one of its neighbours and return. (See slide 17).

## Mapping to red/black Trees

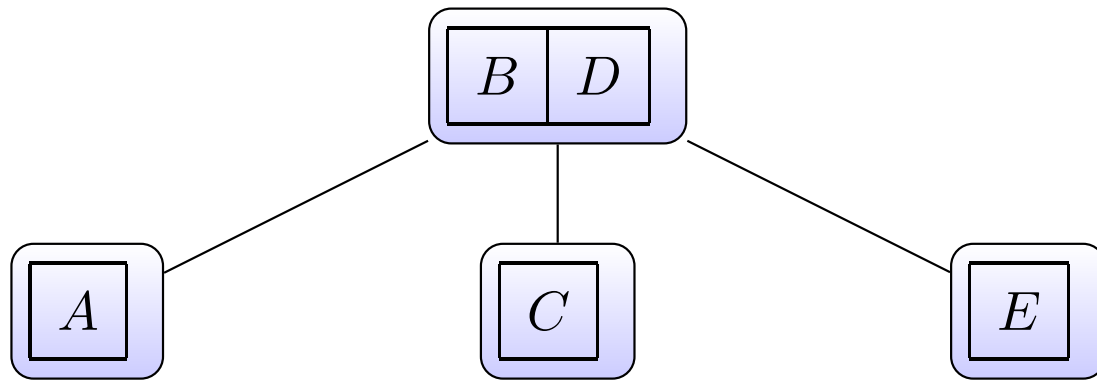
It remains to show how 2, 3, 4-trees are mapped to red/black trees:



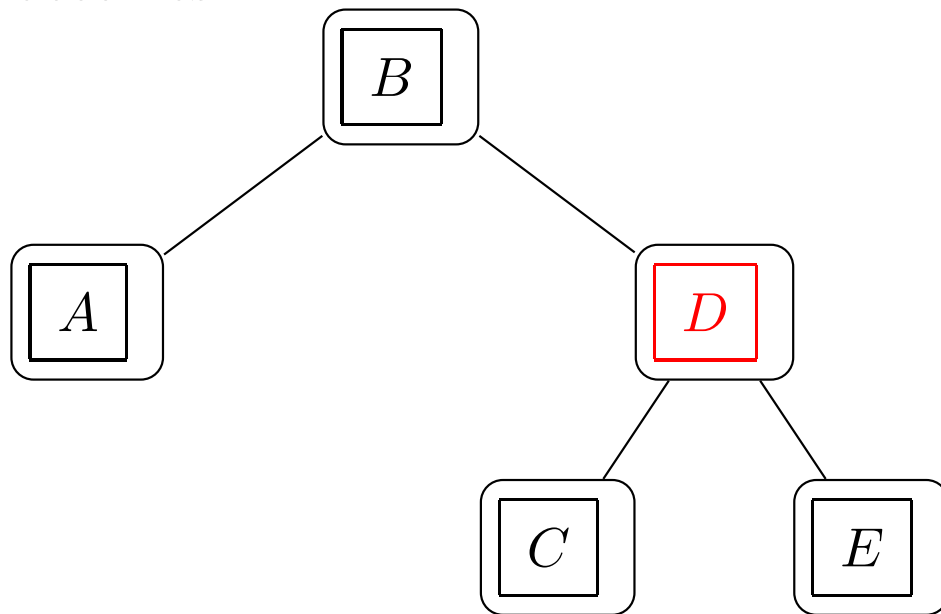
becomes



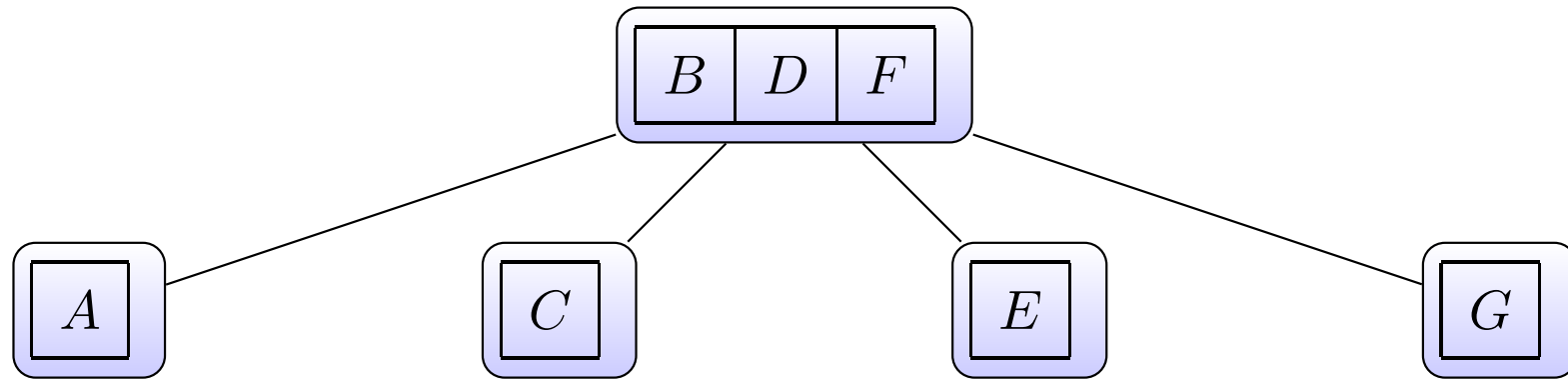
Node has 3 Children



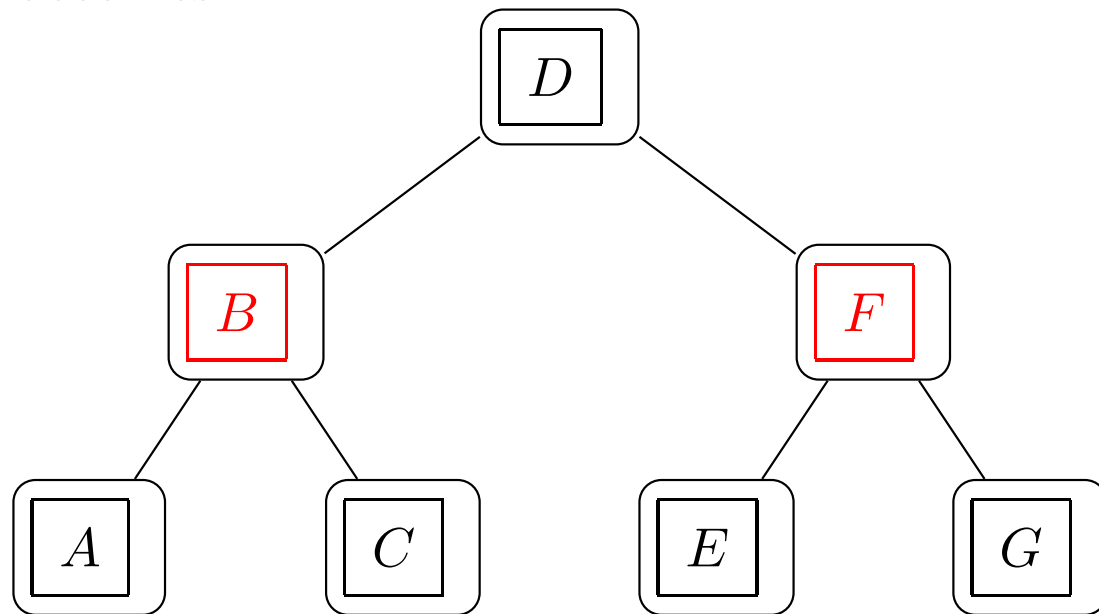
becomes



Node has 4 Children



becomes



## Final Remarks

We have shown that red/black trees are a convenient way of representing 2, 3, 4-trees.

The rules on slide 5 follow from the translations on the previous three slides.

The permutations that we gave for 2, 3, 4 trees can be redefined for red/black trees.

Red/black trees were invented by Rudolf Bayer, and RB can be considered an abbreviation of his name.