

02110

Inge Li Gørtz

Welcome

- Inge Li Gørtz.
- Reverse teaching and discussion of exercises:
 - 3 teaching assistants
 - 8.00-9.00 Group work (on exercises you could not solve at home)
 - 9.00-10.30 Discussions of your solutions in class
 - 10.30-12.00 Lecture
- Prepare and solve exercises at home.
- Use Piazza for questions during the week
- Weekly assignments (You have to pass 3 out of 9 + 2 implementation exercises in order to be able to attend the written exam).
- Prerequisites: 02105/02326 Algorithms and Data Structures I

Balanced Search Trees

2-3-4 trees

red-black trees

References: CLRS Chapter 13, Algorithms in Java
(handout)

Balanced search trees

Dynamic sets

- Search
- Insert
- Delete
- Maximum
- Minimum
- Successor
- Predecessor

This lecture: 2-3-4 trees, red-black trees

Next time: Splay trees

Dynamic set implementations

Worst case running times

Implementation	search	insert	delete	minimum	maximum	successor	predecessor
linked lists	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
ordered array	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(\log n)$	$O(\log n)$
BST	$O(h)$	$O(h)$	$O(h)$	$O(h)$	$O(h)$	$O(h)$	$O(h)$

In worst case $h=n$.

In best case $h= \log n$ (fully balanced binary tree)

Today: How to keep the trees balanced.

2-3-4 trees

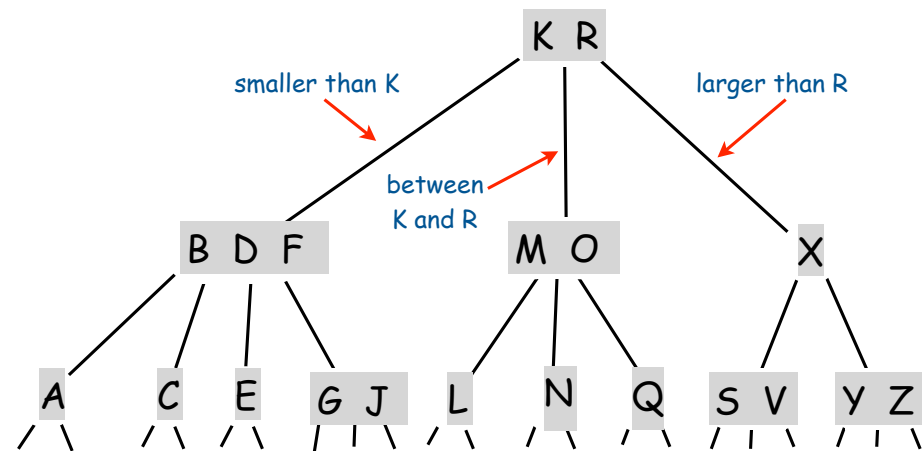
2-3-4 trees

2-3-4 trees. Allow nodes to have multiple keys.

Perfect balance. Every path from root to leaf has same length.

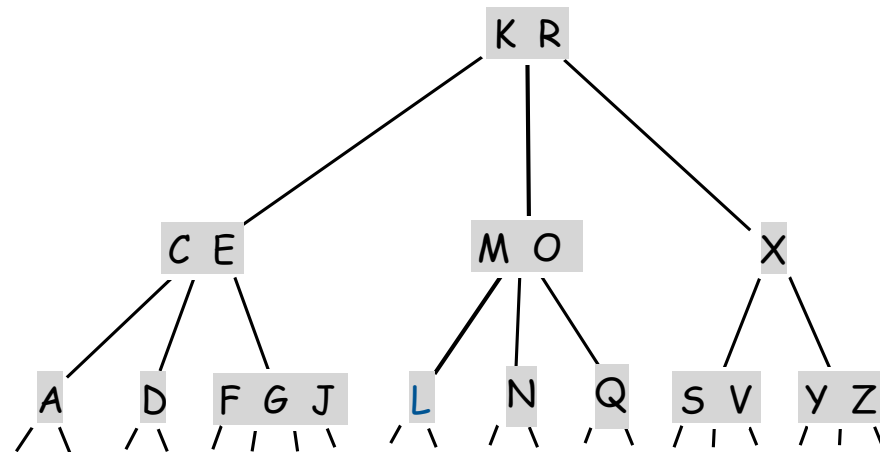
Allow 1, 2, or 3 keys per node

- 2-node: one key, 2 children
- 3-node: 2 keys, 3 children
- 4-node: 3 keys, 4 children



Searching in a 2-3-4 tree

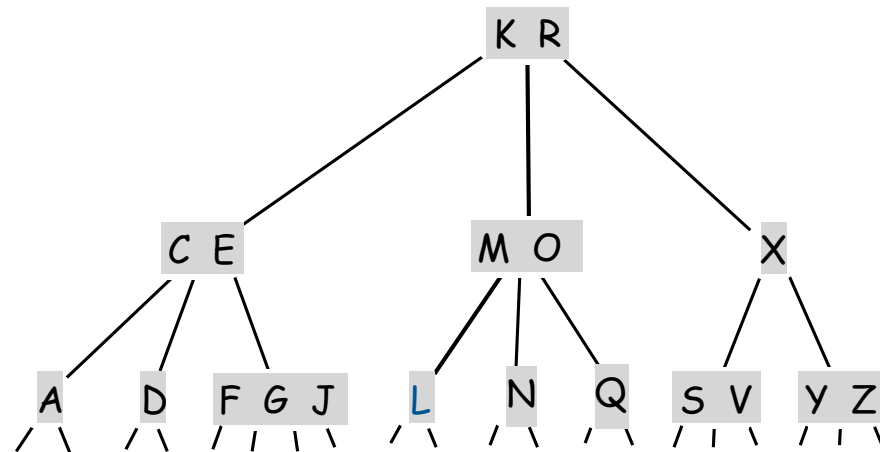
Search.



Searching in a 2-3-4 tree

Search.

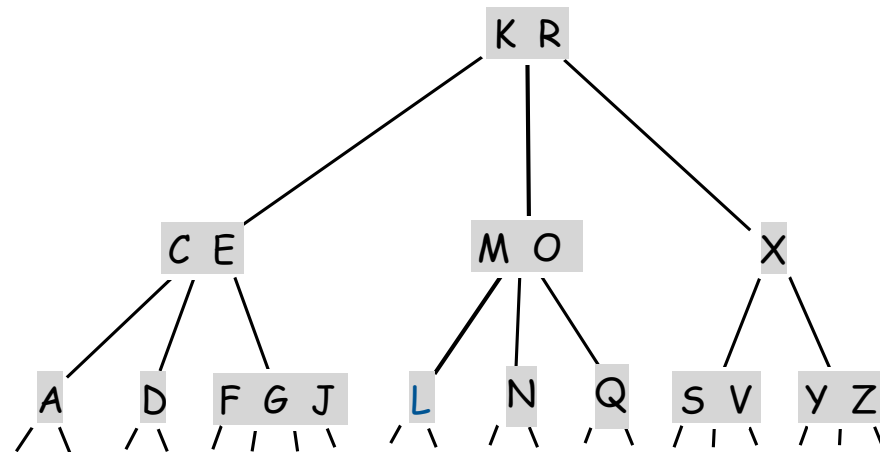
- Compare search key against keys in node.



Searching in a 2-3-4 tree

Search.

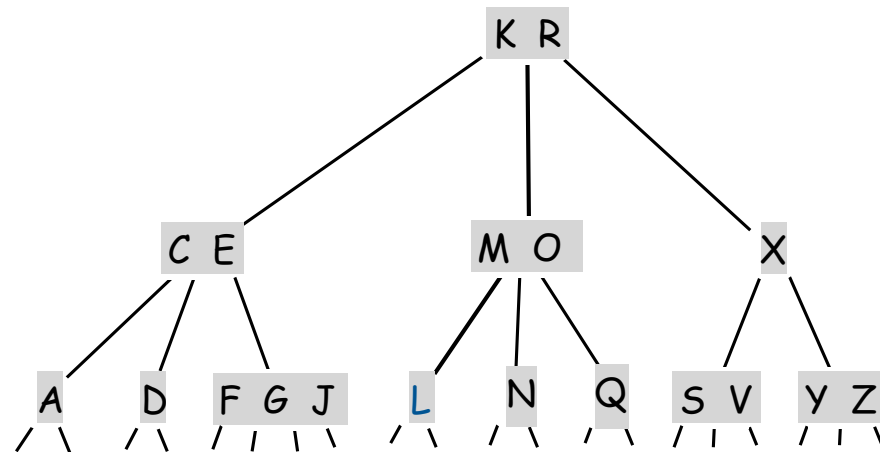
- Compare search key against keys in node.
- Find interval containing search key



Searching in a 2-3-4 tree

Search.

- Compare search key against keys in node.
- Find interval containing search key
- Follow associated link (recursively)

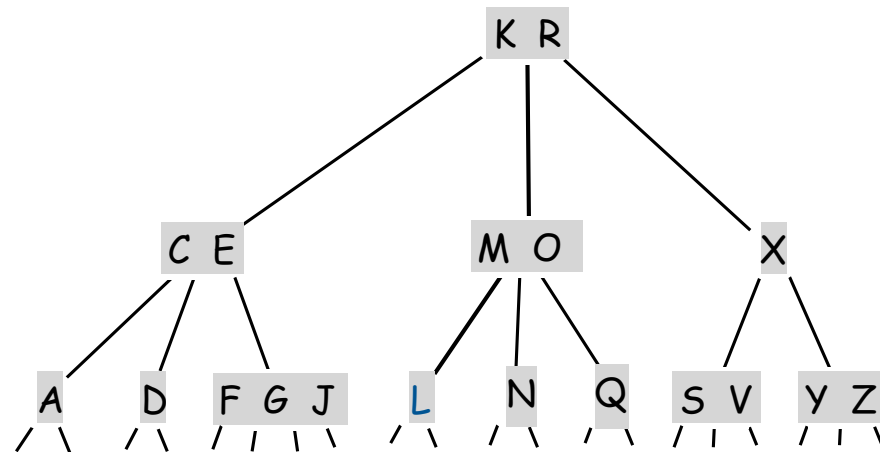


Searching in a 2-3-4 tree

Search.

- Compare search key against keys in node.
- Find interval containing search key
- Follow associated link (recursively)

Ex. Search for L

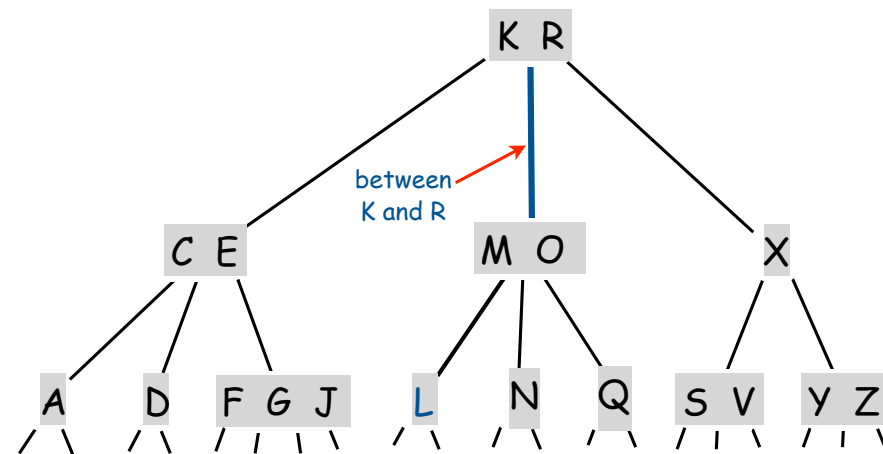


Searching in a 2-3-4 tree

Search.

- Compare search key against keys in node.
- Find interval containing search key
- Follow associated link (recursively)

Ex. Search for L

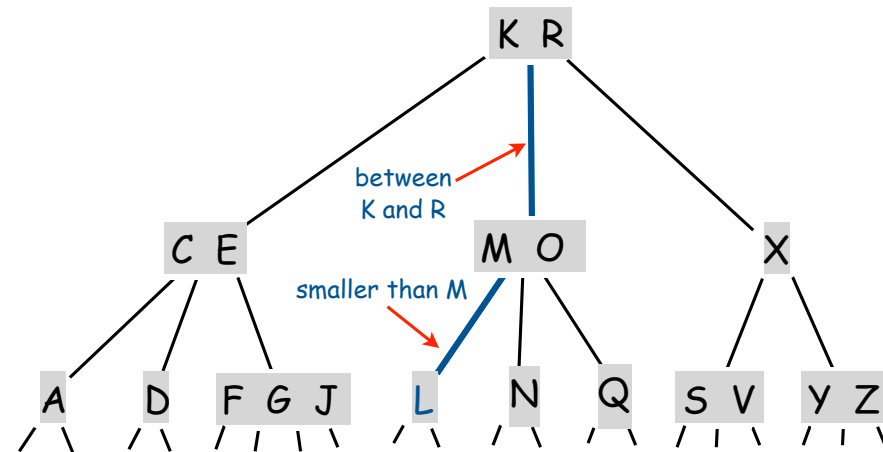


Searching in a 2-3-4 tree

Search.

- Compare search key against keys in node.
- Find interval containing search key
- Follow associated link (recursively)

Ex. Search for L

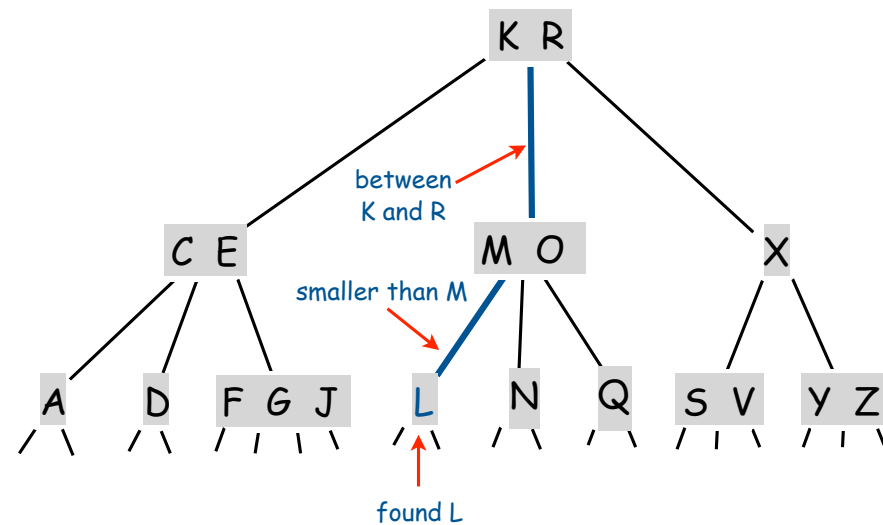


Searching in a 2-3-4 tree

Search.

- Compare search key against keys in node.
- Find interval containing search key
- Follow associated link (recursively)

Ex. Search for L

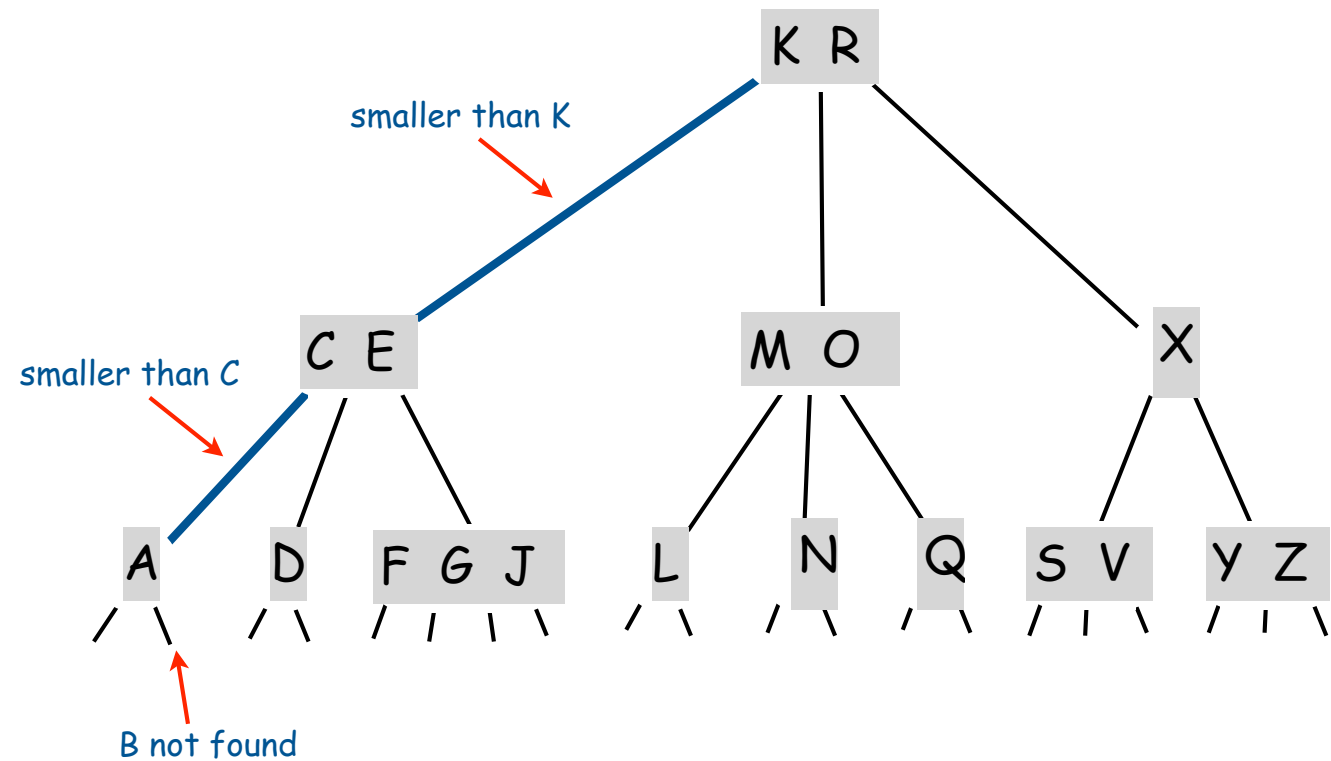


Insertion in a 2-3-4 tree

Insert.

- Search to bottom for key.

Ex. Insert B

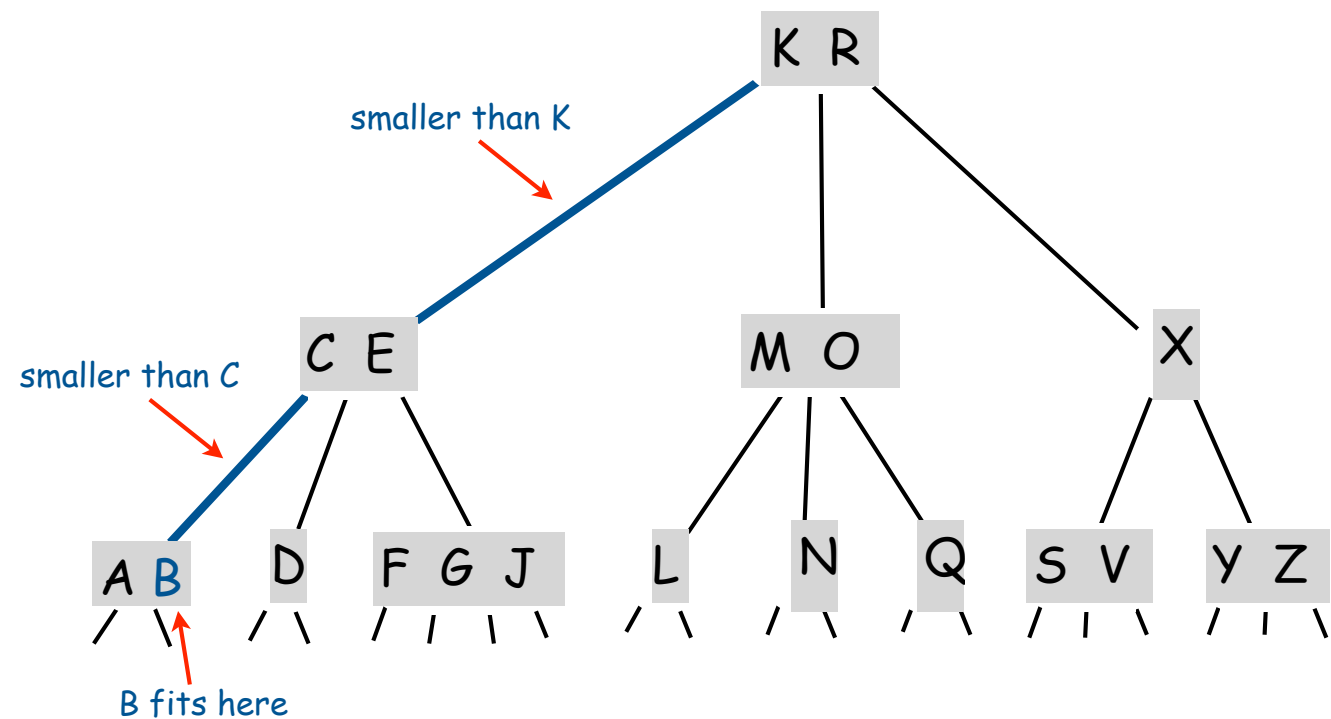


Insertion in a 2-3-4 tree

Insert.

- Search to bottom for key.
- 2-node at bottom: convert to 3-node

Ex. Insert B

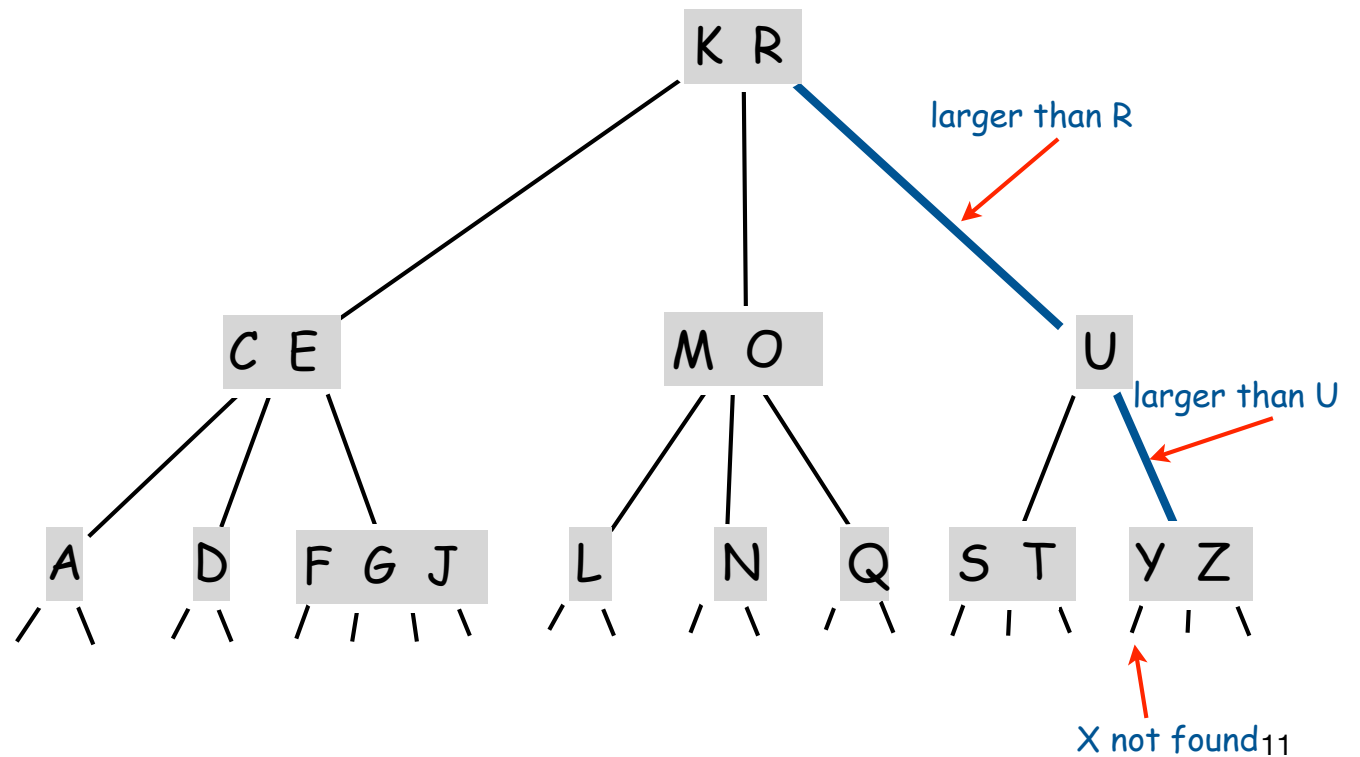


Insertion in a 2-3-4 tree

Insert.

- Search to bottom for key.
- 2-node at bottom: convert to 3-node

Ex. Insert X

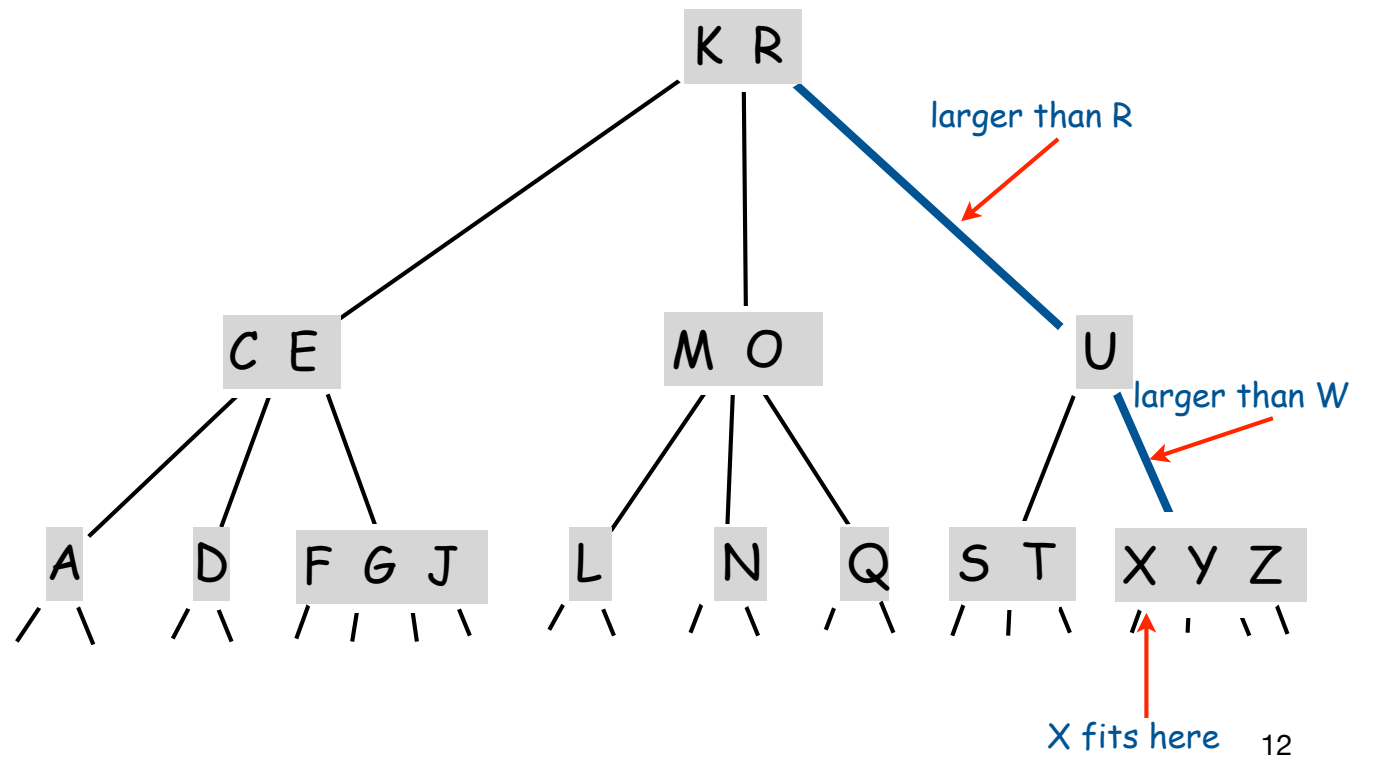


Insertion in a 2-3-4 tree

Insert.

- Search to bottom for key.
- 2-node at bottom: convert to 3-node
- 3-node at bottom: convert to 4-node

Ex. Insert X

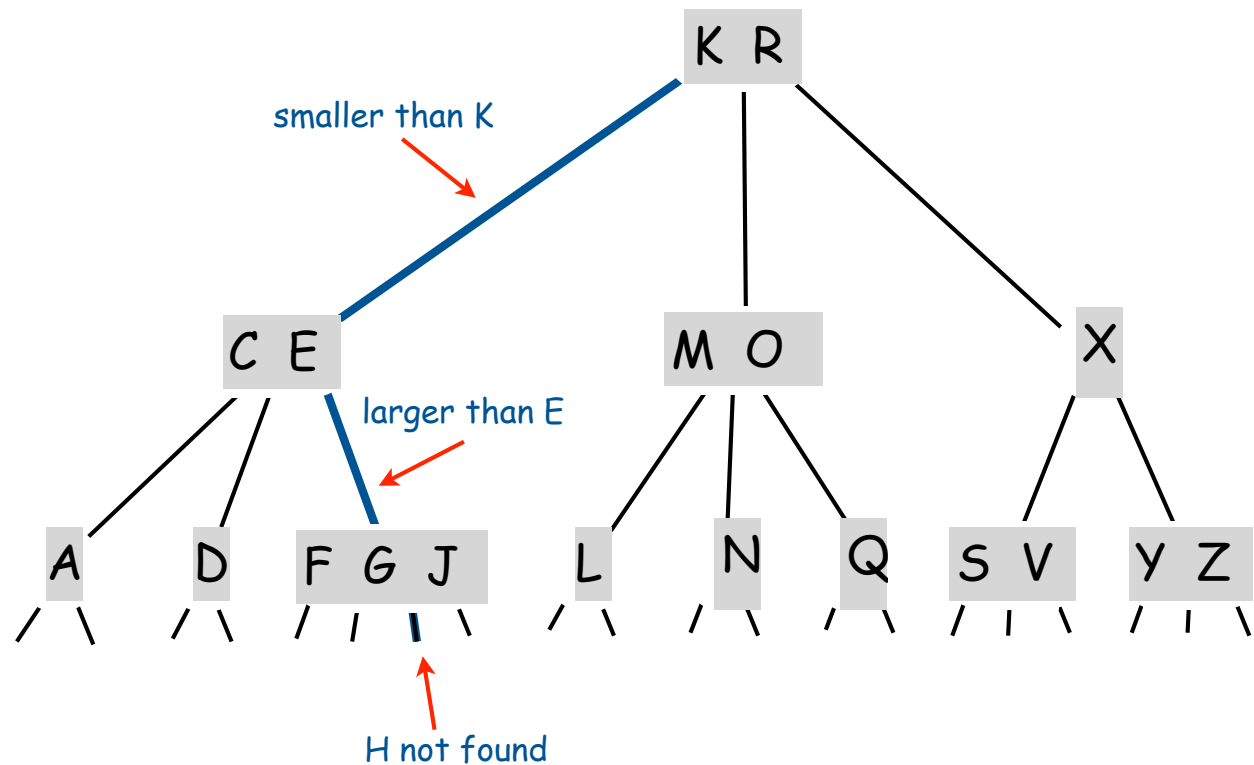


Insertion in a 2-3-4 tree

Insert.

- Search to bottom for key.
- 2-node at bottom: convert to 3-node
- 3-node at bottom: convert to 4-node

Ex. Insert H

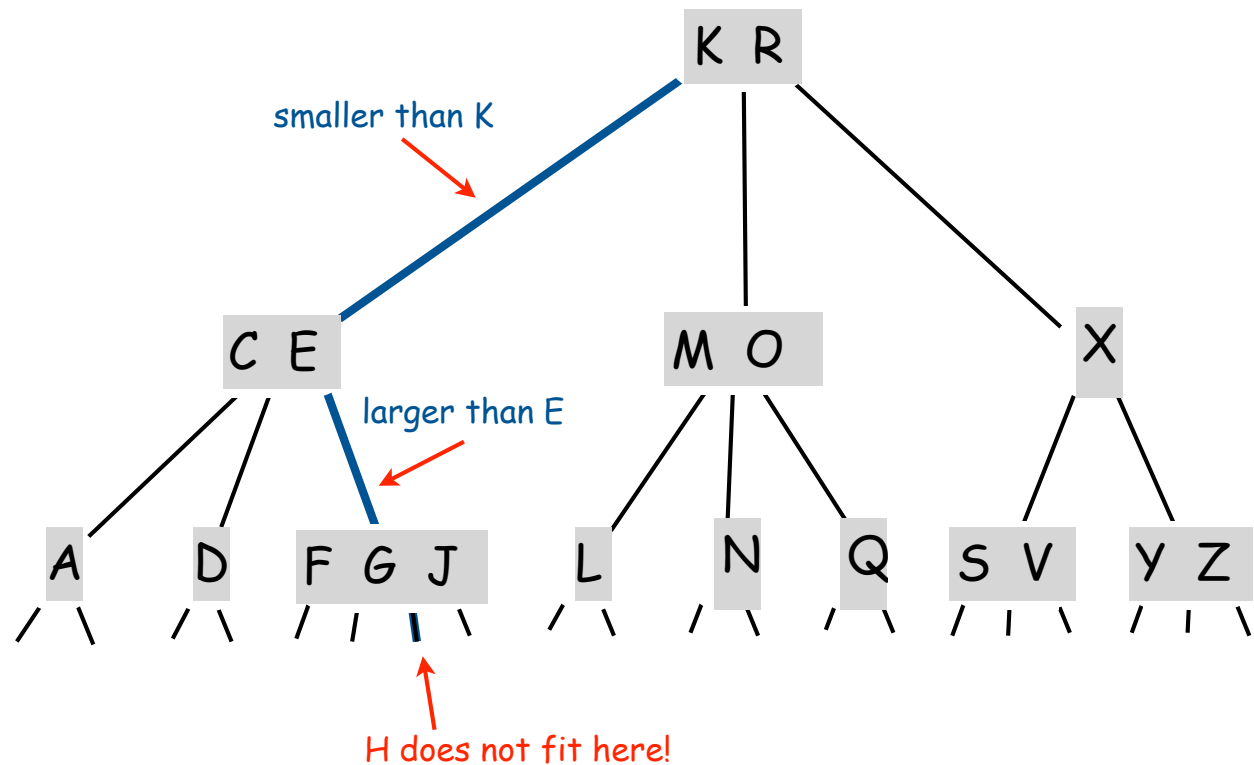


Insertion in a 2-3-4 tree

Insert.

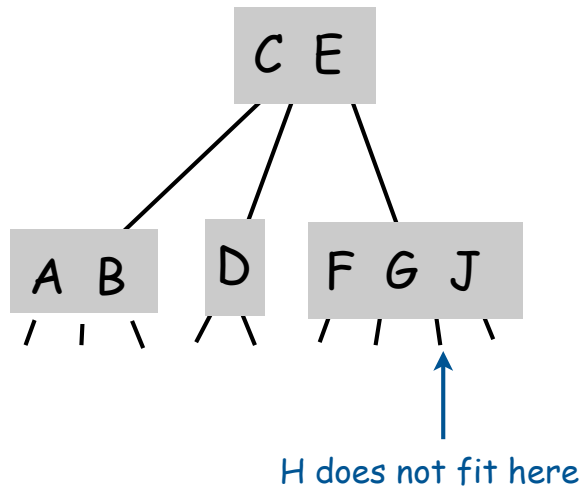
- Search to bottom for key.
- 2-node at bottom: convert to 3-node
- 3-node at bottom: convert to 4-node
- 4-node at bottom: ??

Ex. Insert H



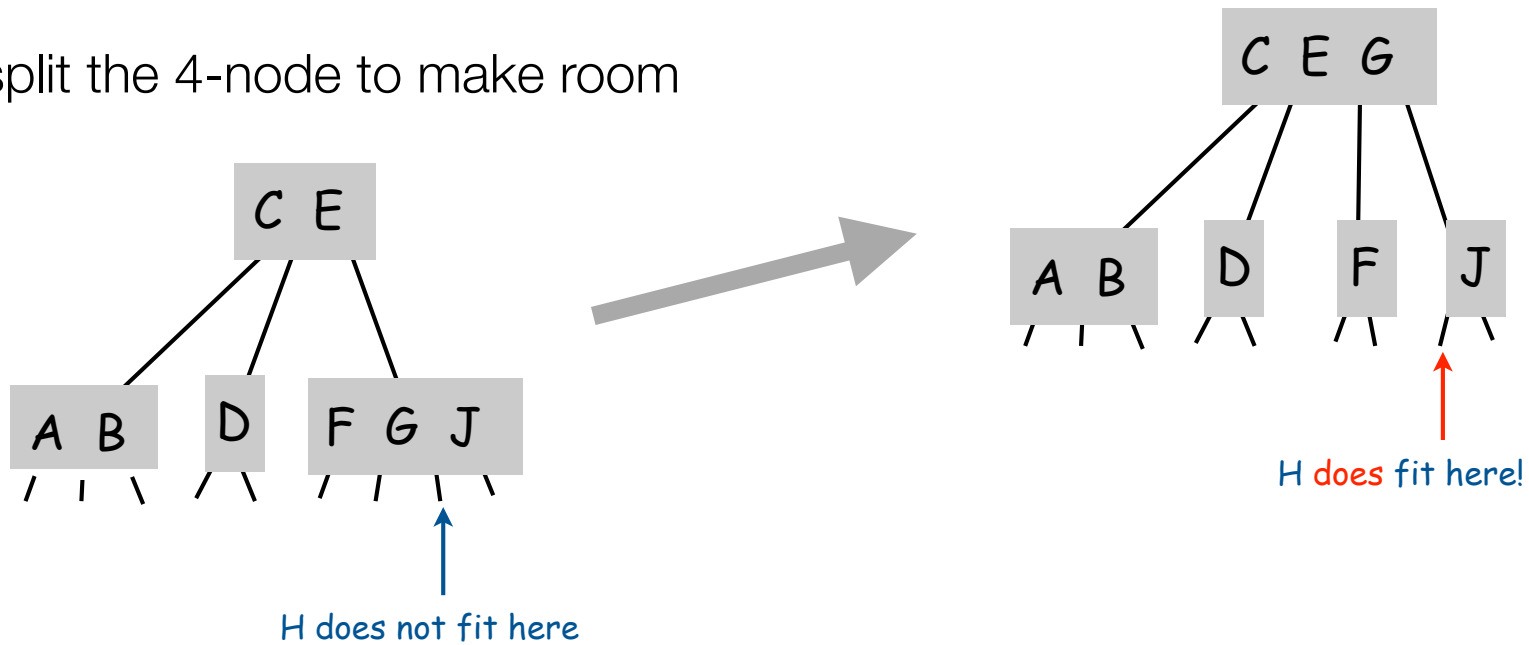
Splitting a 4-node in a 2-3-4 tree

Idea: split the 4-node to make room



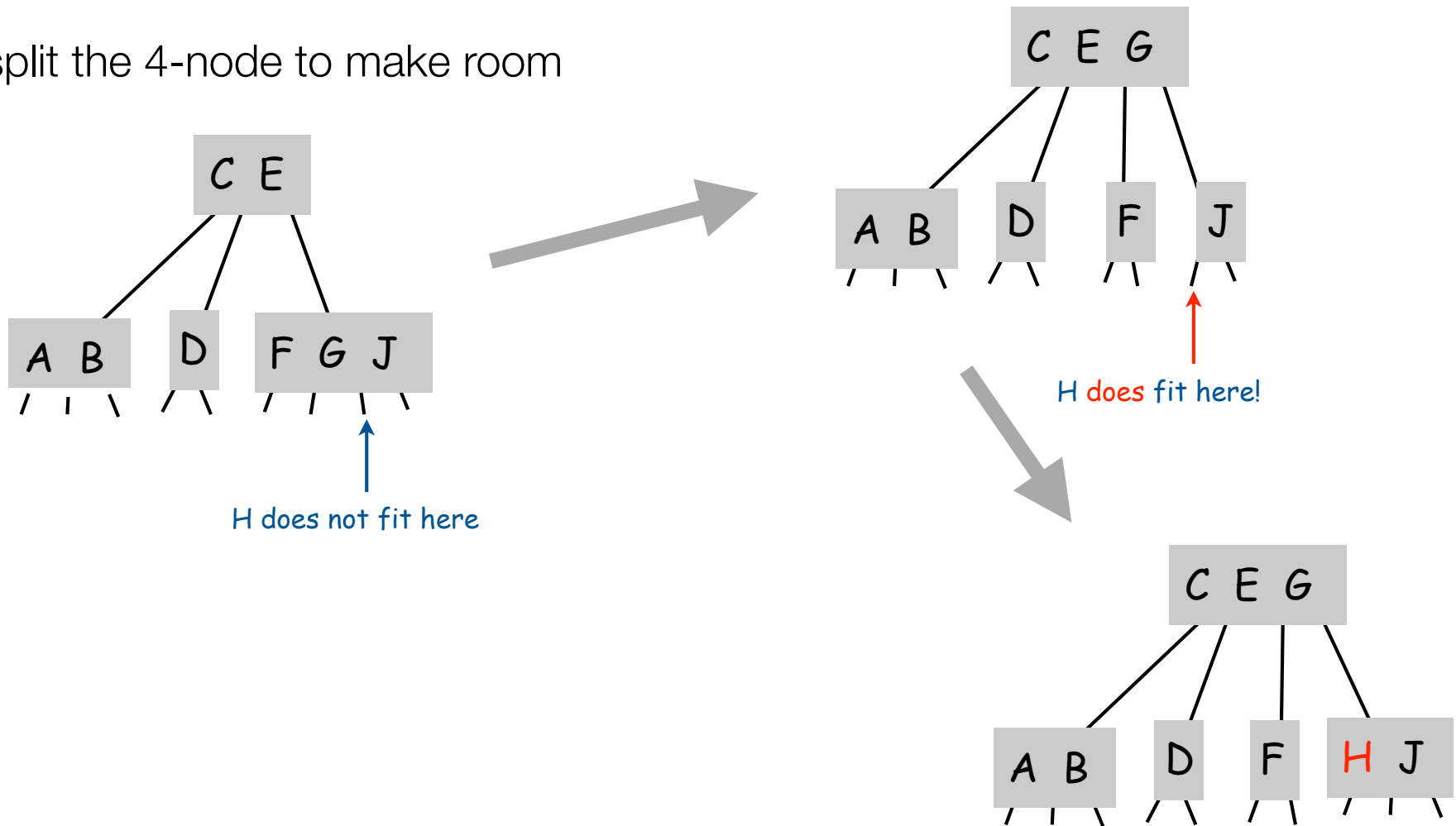
Splitting a 4-node in a 2-3-4 tree

Idea: split the 4-node to make room



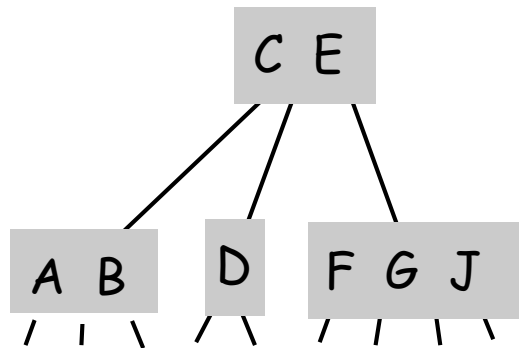
Splitting a 4-node in a 2-3-4 tree

Idea: split the 4-node to make room

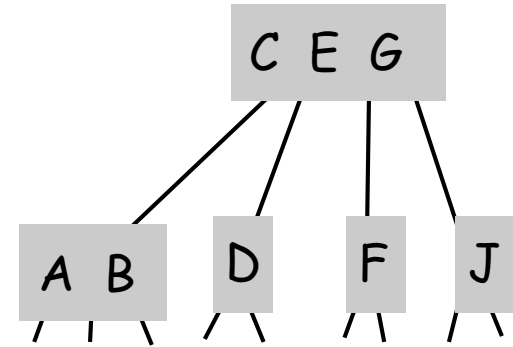


Splitting a 4-node in a 2-3-4 tree

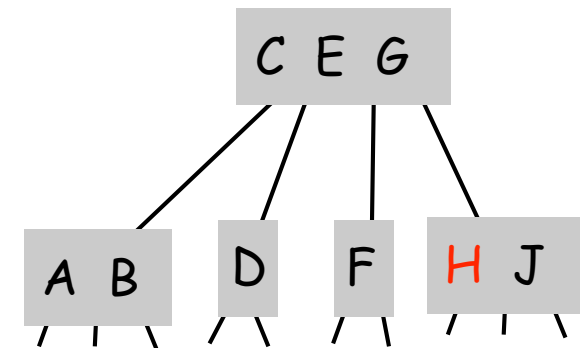
Idea: split the 4-node to make room



H does not fit here



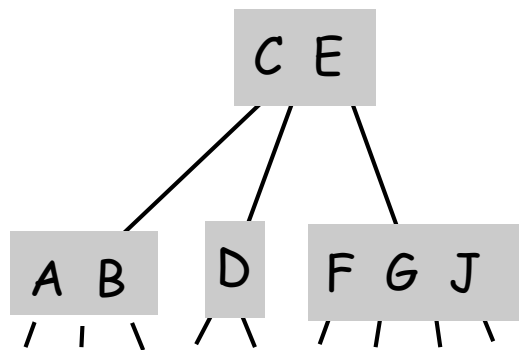
H does fit here!



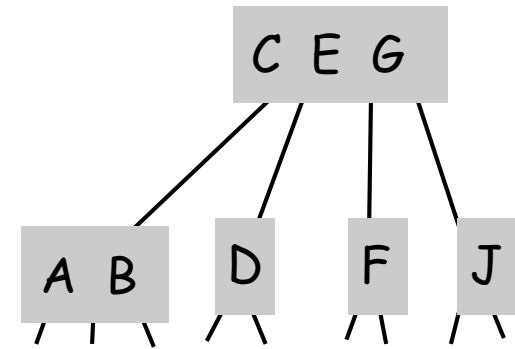
Problem: Doesn't work if parent is a 4-node

Splitting a 4-node in a 2-3-4 tree

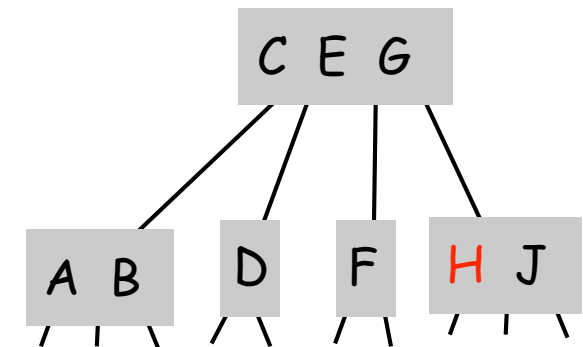
Idea: split the 4-node to make room



H does not fit here



H does fit here!

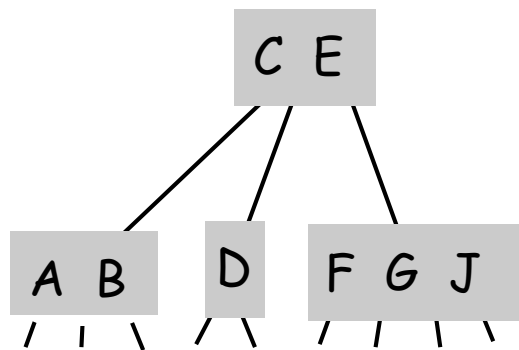


Problem: Doesn't work if parent is a 4-node

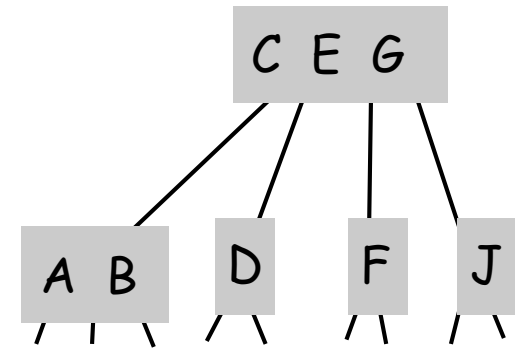
Solution 1: Split the parent (and continue splitting while necessary).

Splitting a 4-node in a 2-3-4 tree

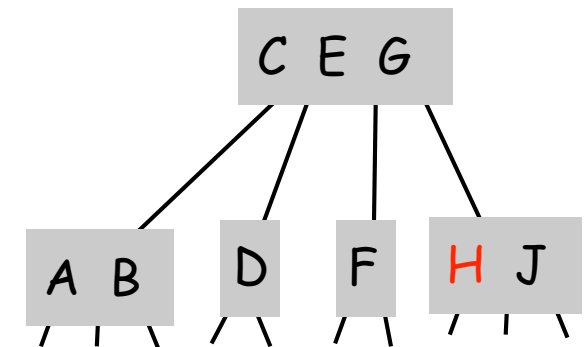
Idea: split the 4-node to make room



H does not fit here



H does fit here!



Problem: Doesn't work if parent is a 4-node

Solution 1: Split the parent (and continue splitting while necessary).

Solution 2: Split 4-nodes on the way **down**.

Splitting 4-nodes in a 2-3-4 tree

Idea: split 4-nodes on the way **down** the tree.

- Ensures last node is not a 4-node.

Splitting 4-nodes in a 2-3-4 tree

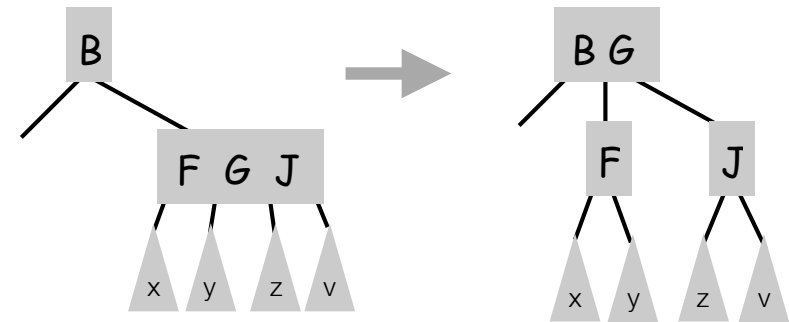
Idea: split 4-nodes on the way **down** the tree.

- Ensures last node is not a 4-node.
- Transformations to split 4-nodes:

Splitting 4-nodes in a 2-3-4 tree

Idea: split 4-nodes on the way **down** the tree.

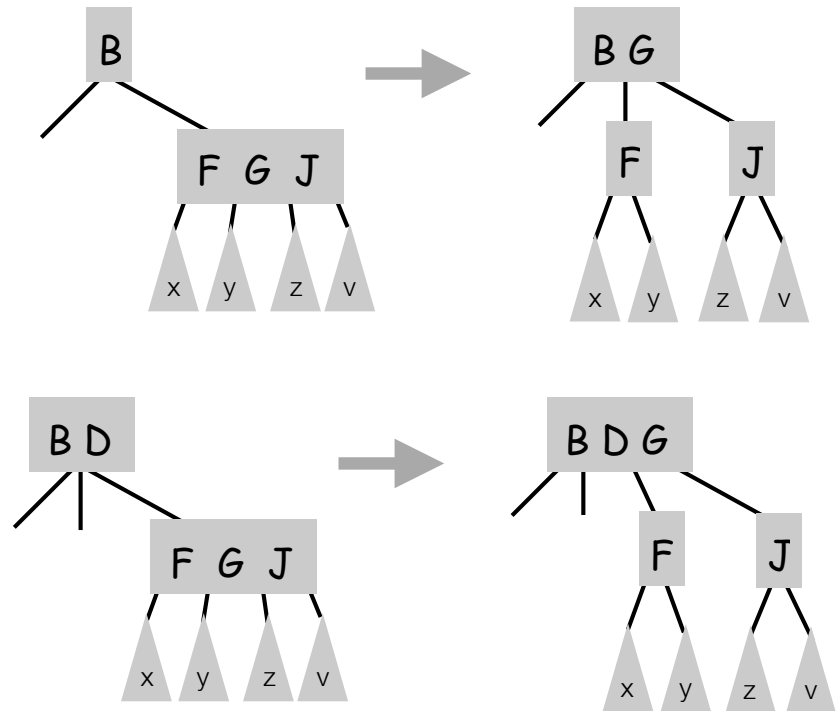
- Ensures last node is not a 4-node.
- Transformations to split 4-nodes:



Splitting 4-nodes in a 2-3-4 tree

Idea: split 4-nodes on the way **down** the tree.

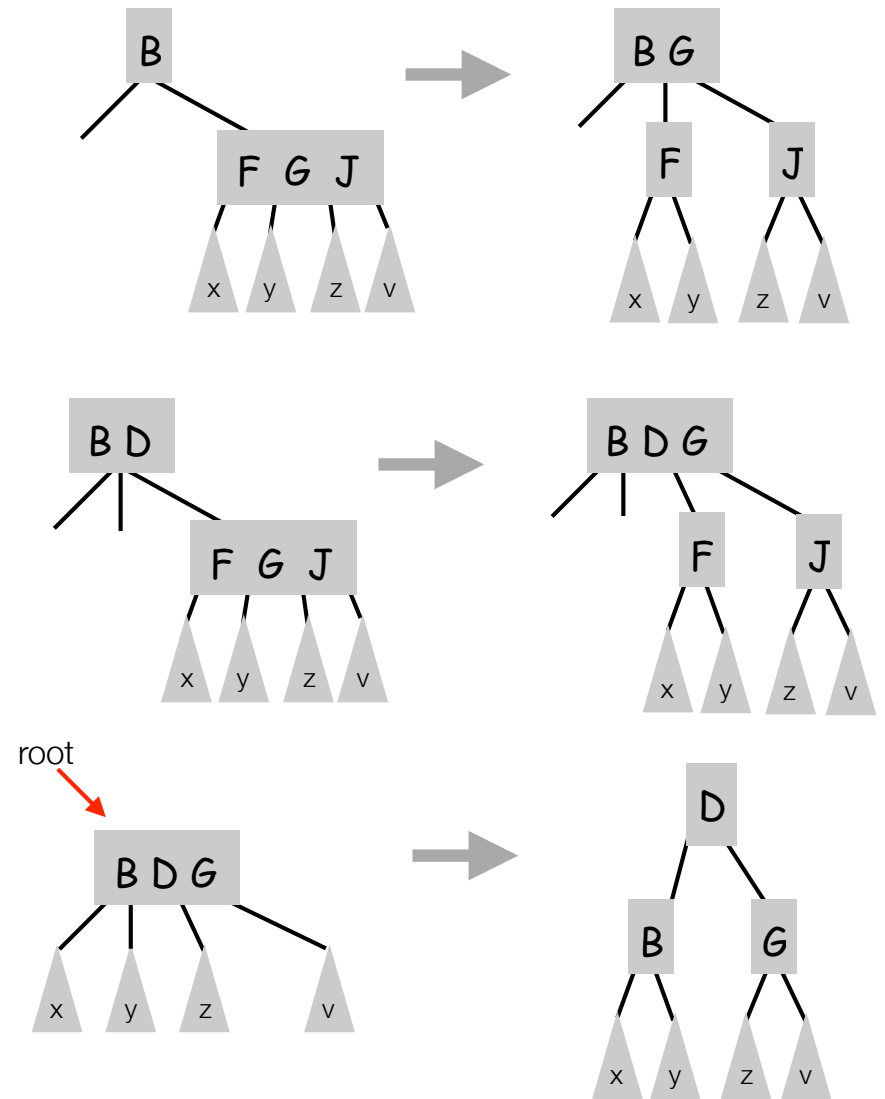
- Ensures last node is not a 4-node.
- Transformations to split 4-nodes:



Splitting 4-nodes in a 2-3-4 tree

Idea: split 4-nodes on the way **down** the tree.

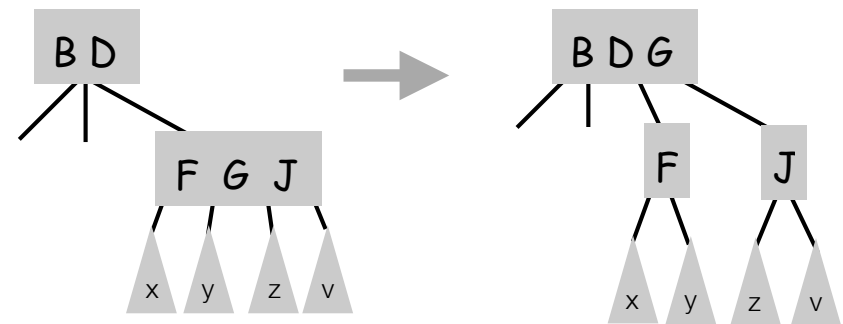
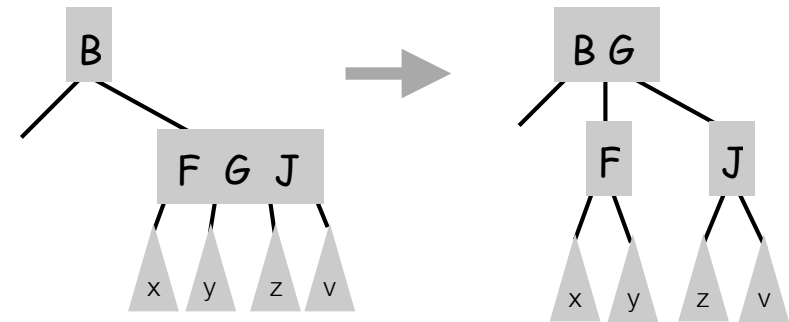
- Ensures last node is not a 4-node.
- Transformations to split 4-nodes:



Splitting 4-nodes in a 2-3-4 tree

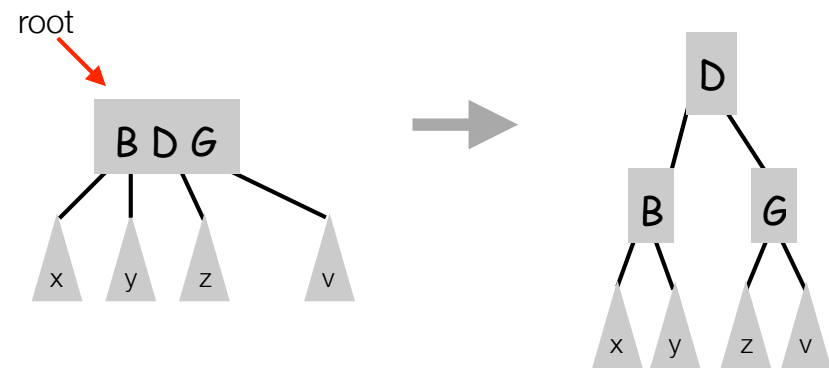
Idea: split 4-nodes on the way **down** the tree.

- Ensures last node is not a 4-node.
- Transformations to split 4-nodes:



Invariant. **Current node is not a 4-node.**

Consequence. Insertion at bottom is easy since it's not a 4-node.

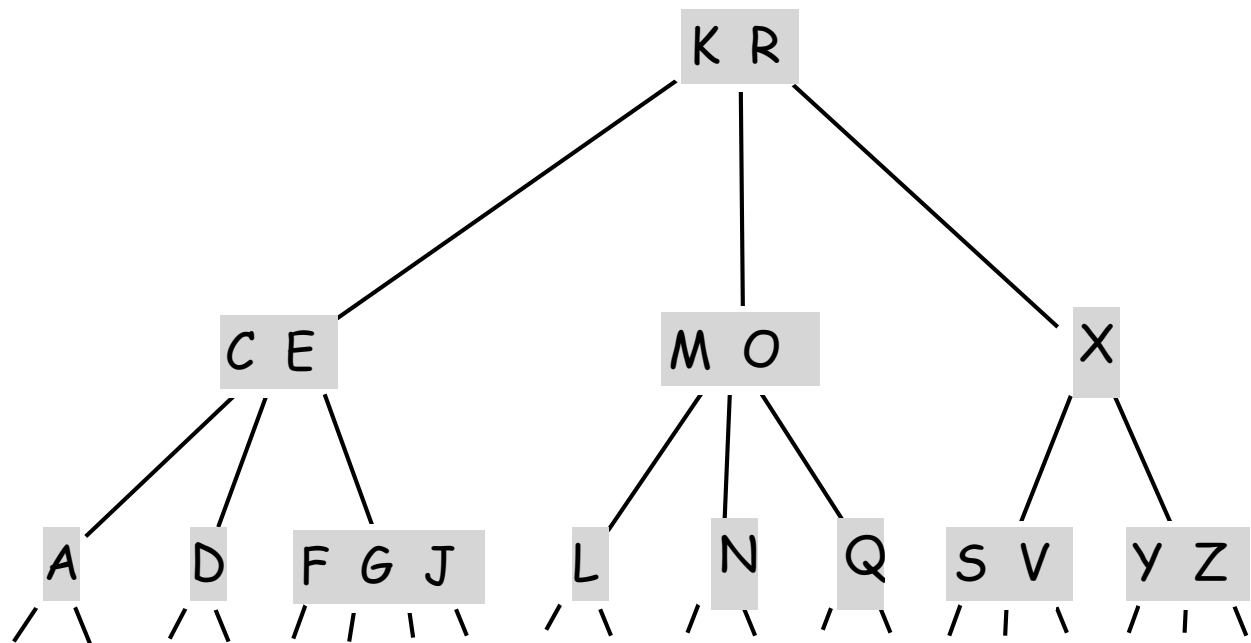


Insertion in a 2-3-4 tree

Insert.

- Search to bottom for key.
- 2-node at bottom: convert to 3-node
- 3-node at bottom: convert to 4-node
- 4-node at bottom: ??

Ex. Insert H

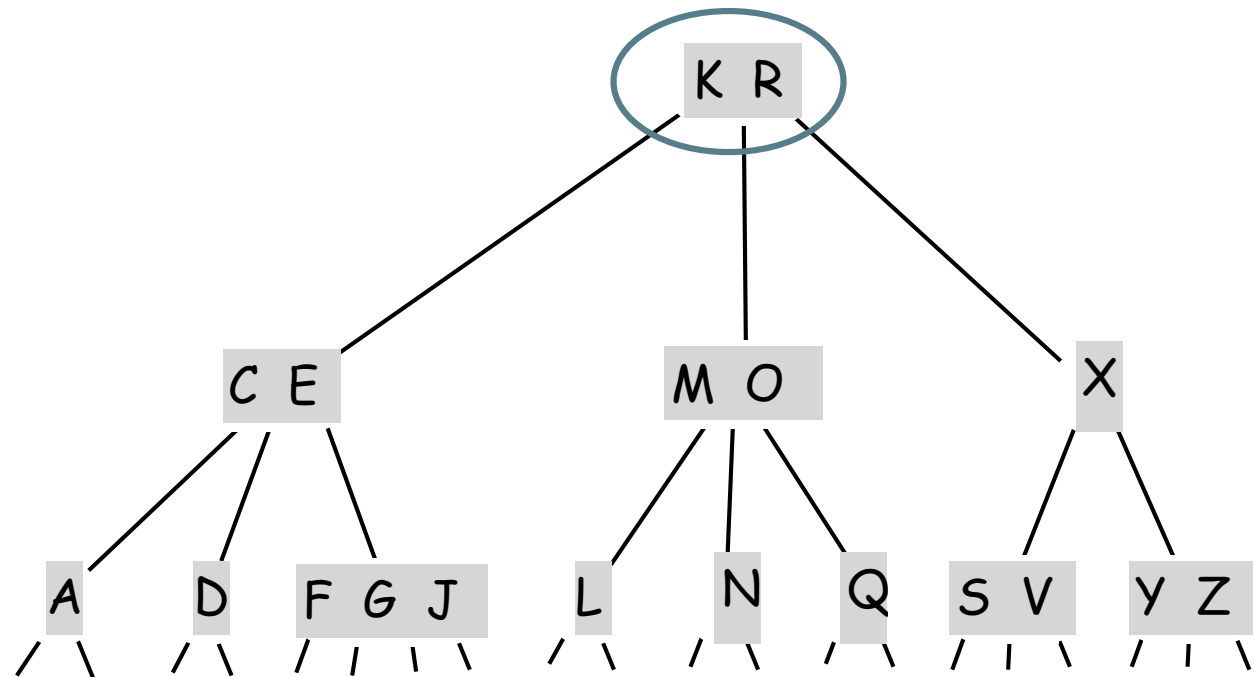


Insertion in a 2-3-4 tree

Insert.

- Search to bottom for key.
- 2-node at bottom: convert to 3-node
- 3-node at bottom: convert to 4-node
- 4-node at bottom: ??

not a 4-node



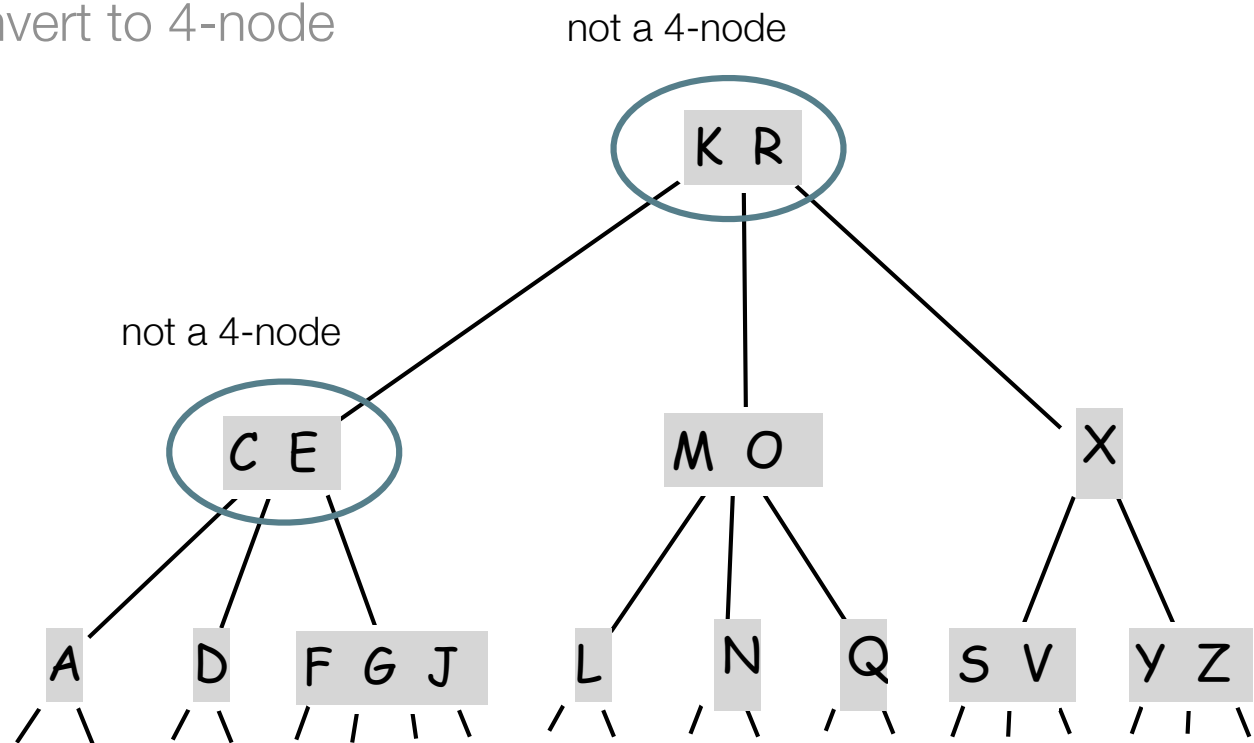
Ex. Insert H

Insertion in a 2-3-4 tree

Insert.

- Search to bottom for key.
- 2-node at bottom: convert to 3-node
- 3-node at bottom: convert to 4-node
- 4-node at bottom: ??

Ex. Insert H

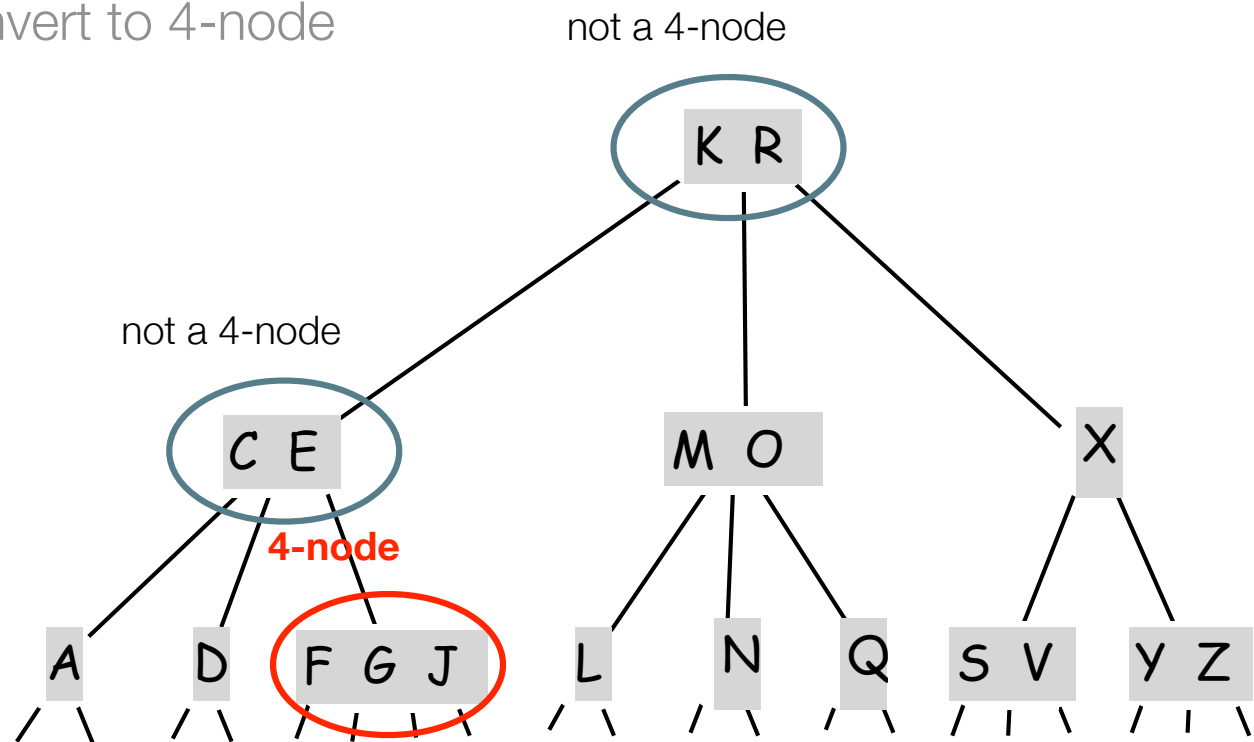


Insertion in a 2-3-4 tree

Insert.

- Search to bottom for key.
- 2-node at bottom: convert to 3-node
- 3-node at bottom: convert to 4-node
- 4-node at bottom: ??

Ex. Insert H

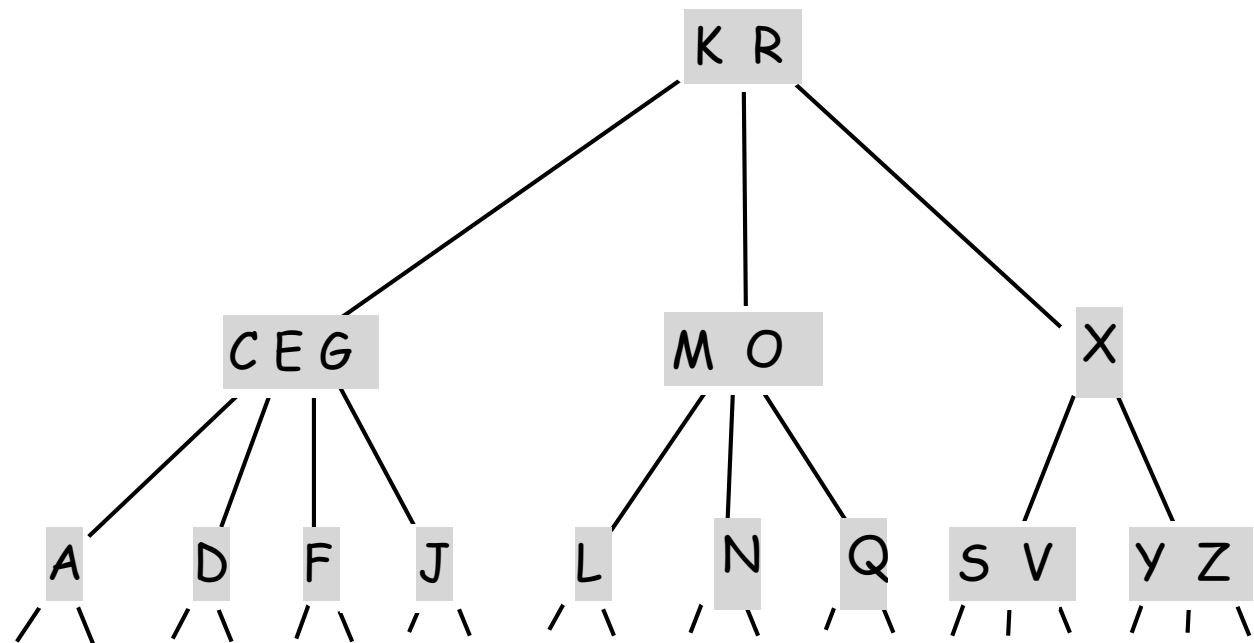


Insertion in a 2-3-4 tree

Insert.

- Search to bottom for key.
- 2-node at bottom: convert to 3-node
- 3-node at bottom: convert to 4-node
- 4-node at bottom: ??

Ex. Insert H

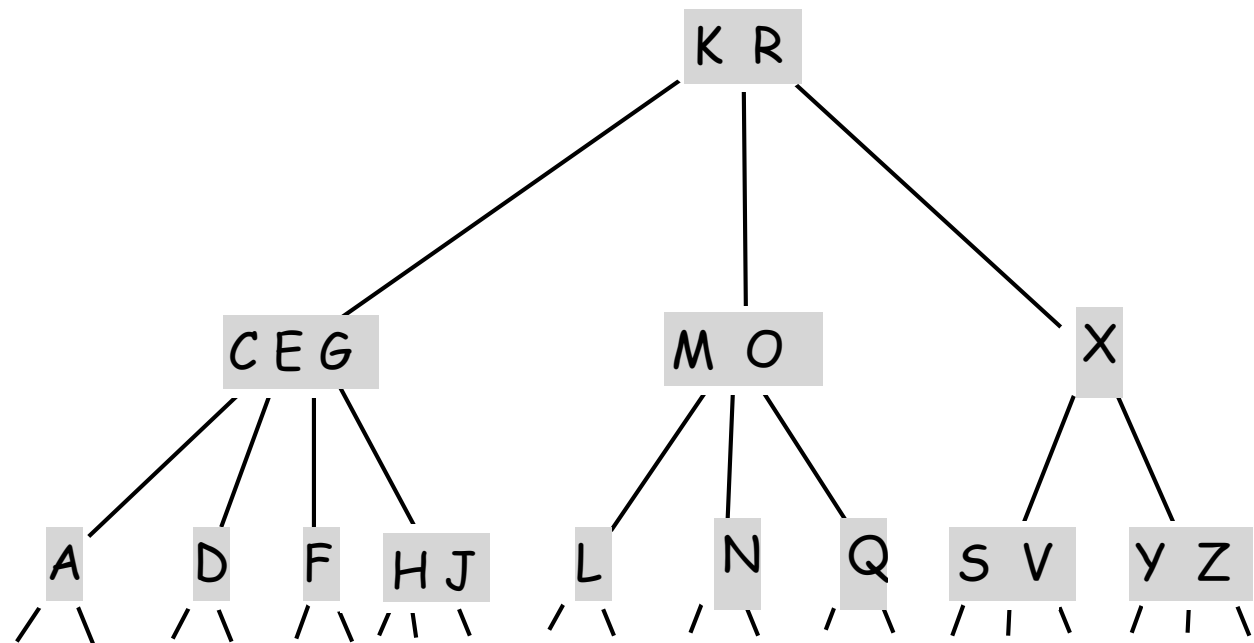


Insertion in a 2-3-4 tree

Insert.

- Search to bottom for key.
- 2-node at bottom: convert to 3-node
- 3-node at bottom: convert to 4-node
- 4-node at bottom: ??

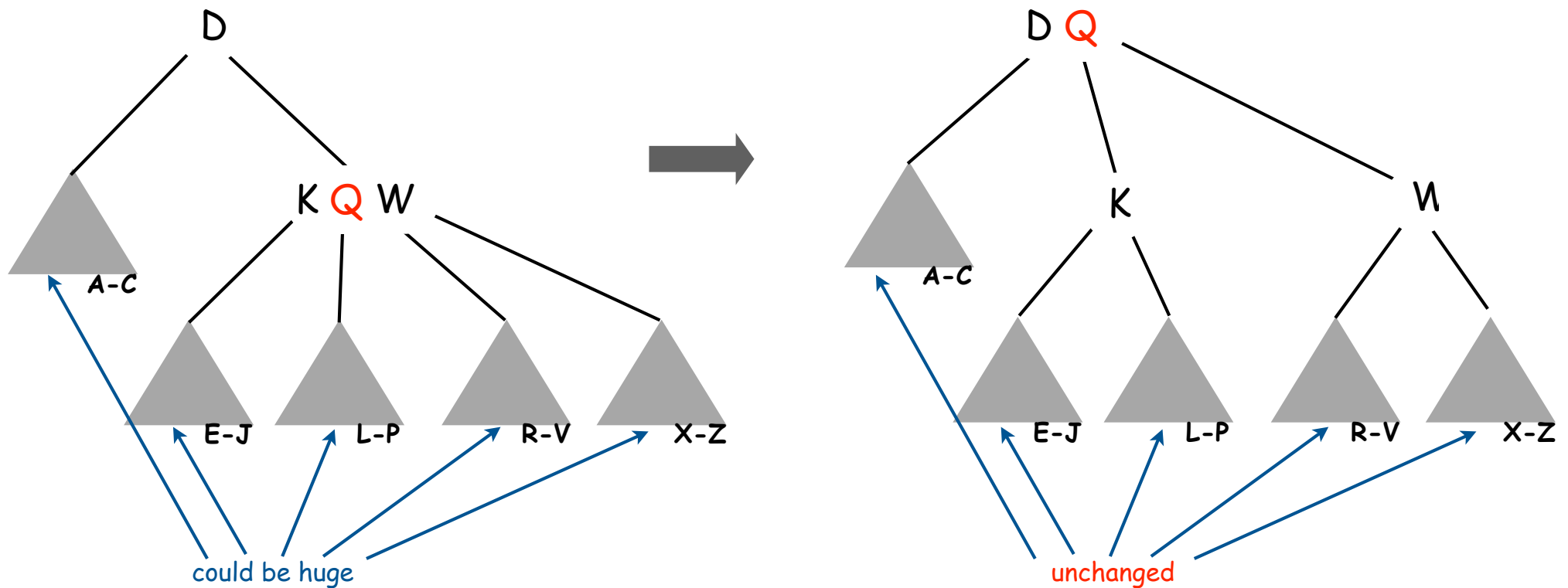
Ex. Insert H



Splitting 4-nodes in a 2-3-4 tree

Local transformations that work **anywhere** in the tree.

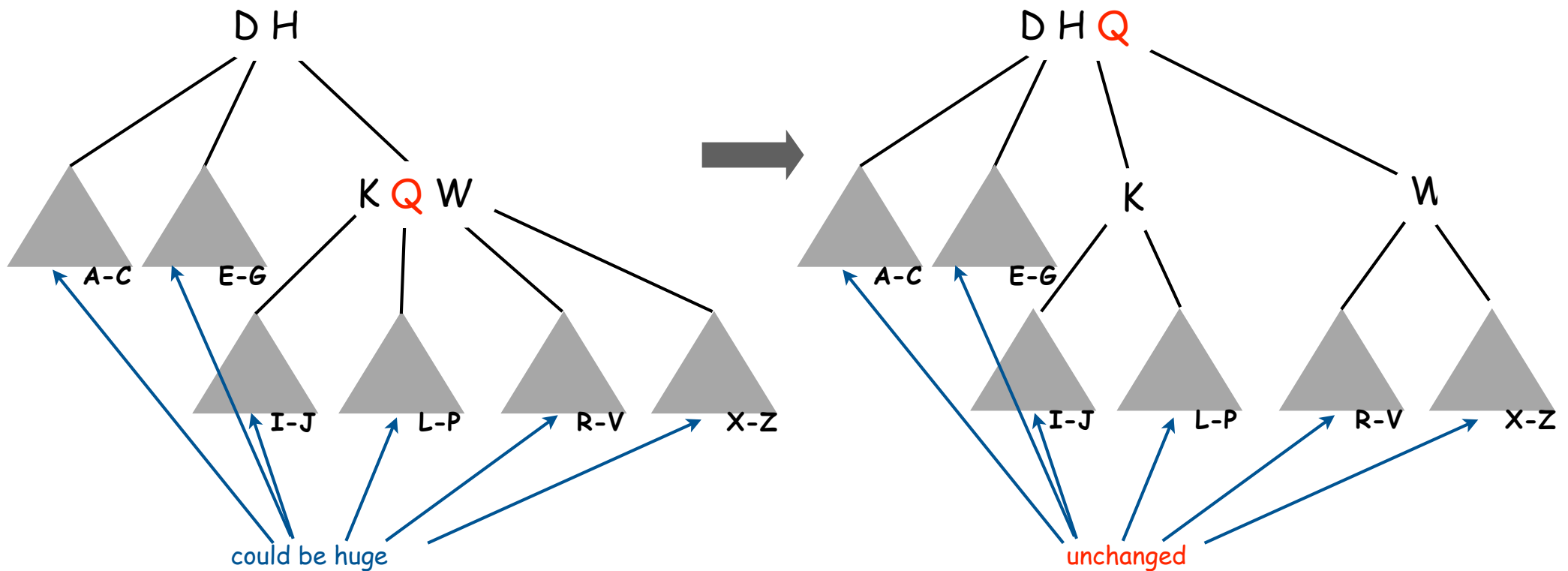
Ex. Splitting a 4-node attached to a 2-node



Splitting 4-nodes in a 2-3-4 tree

Local transformations that work *anywhere* in the tree

Ex. Splitting a 4-node attached to a 3-node



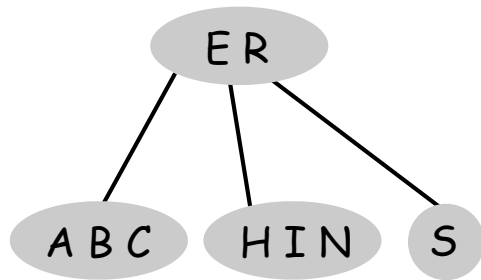
Splitting 4-nodes in a 2-3-4 tree

Local transformations that work **anywhere** in the tree.

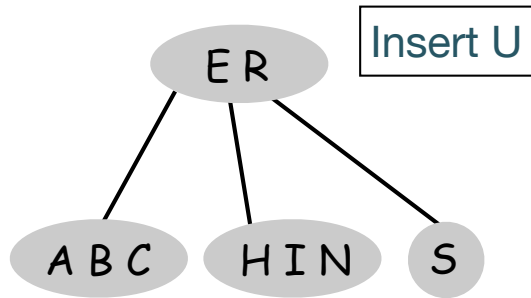
Splitting a 4-node attached to a 4-node **never happens** when we split nodes on the way down the tree.

Invariant. **Current node is not a 4-node.**

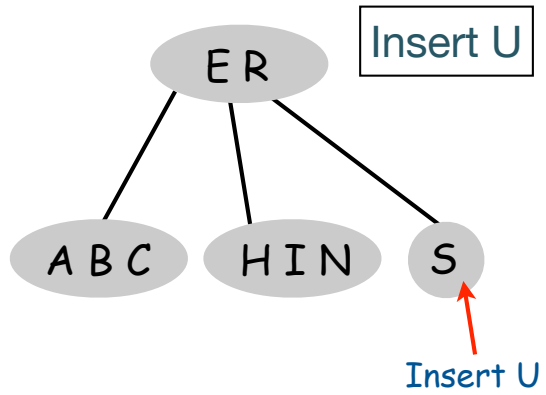
Insertion 2-3-4 trees



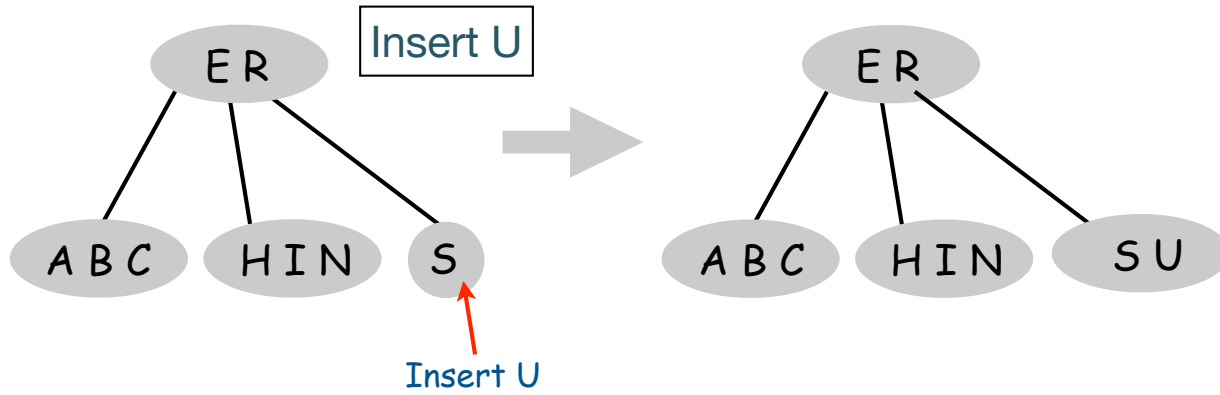
Insertion 2-3-4 trees



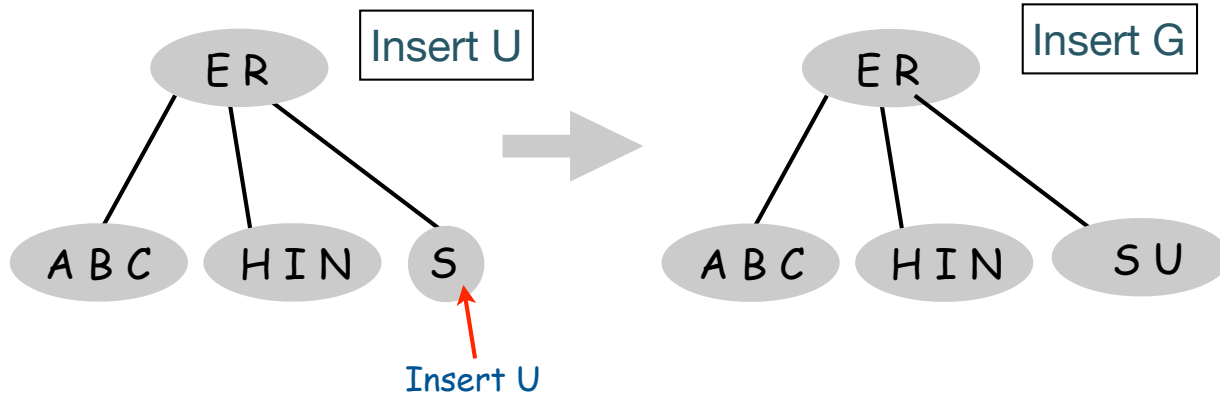
Insertion 2-3-4 trees



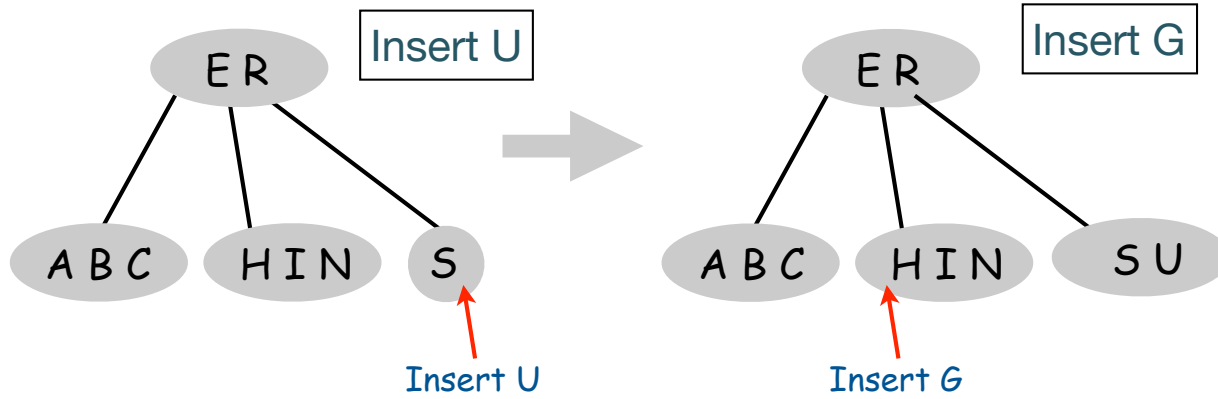
Insertion 2-3-4 trees



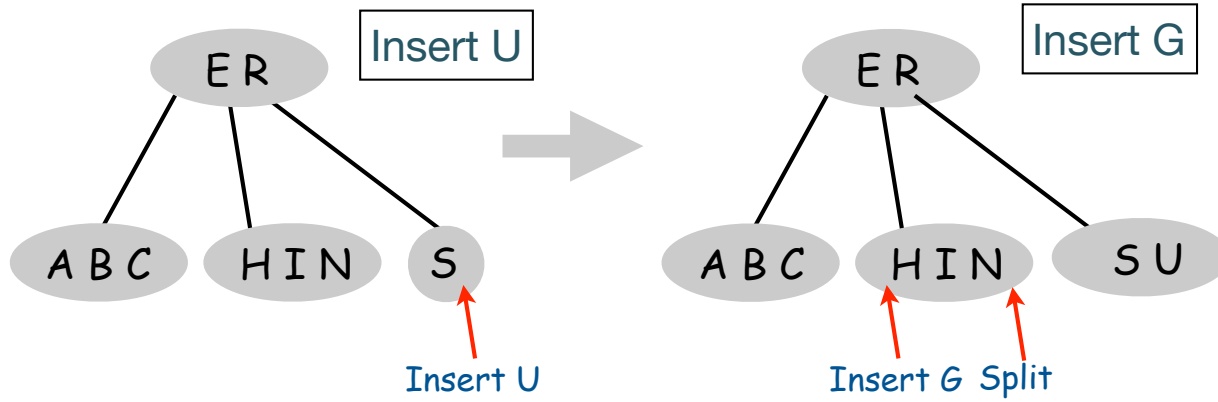
Insertion 2-3-4 trees



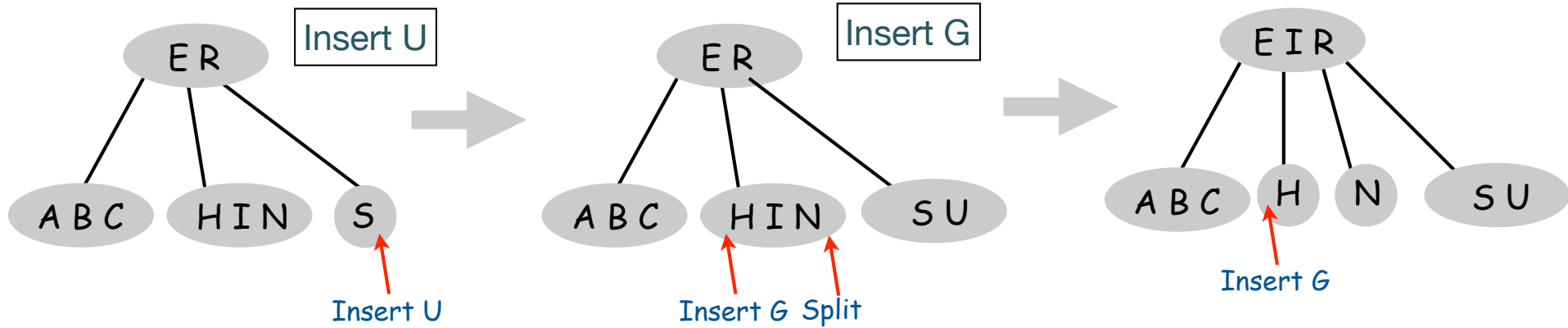
Insertion 2-3-4 trees



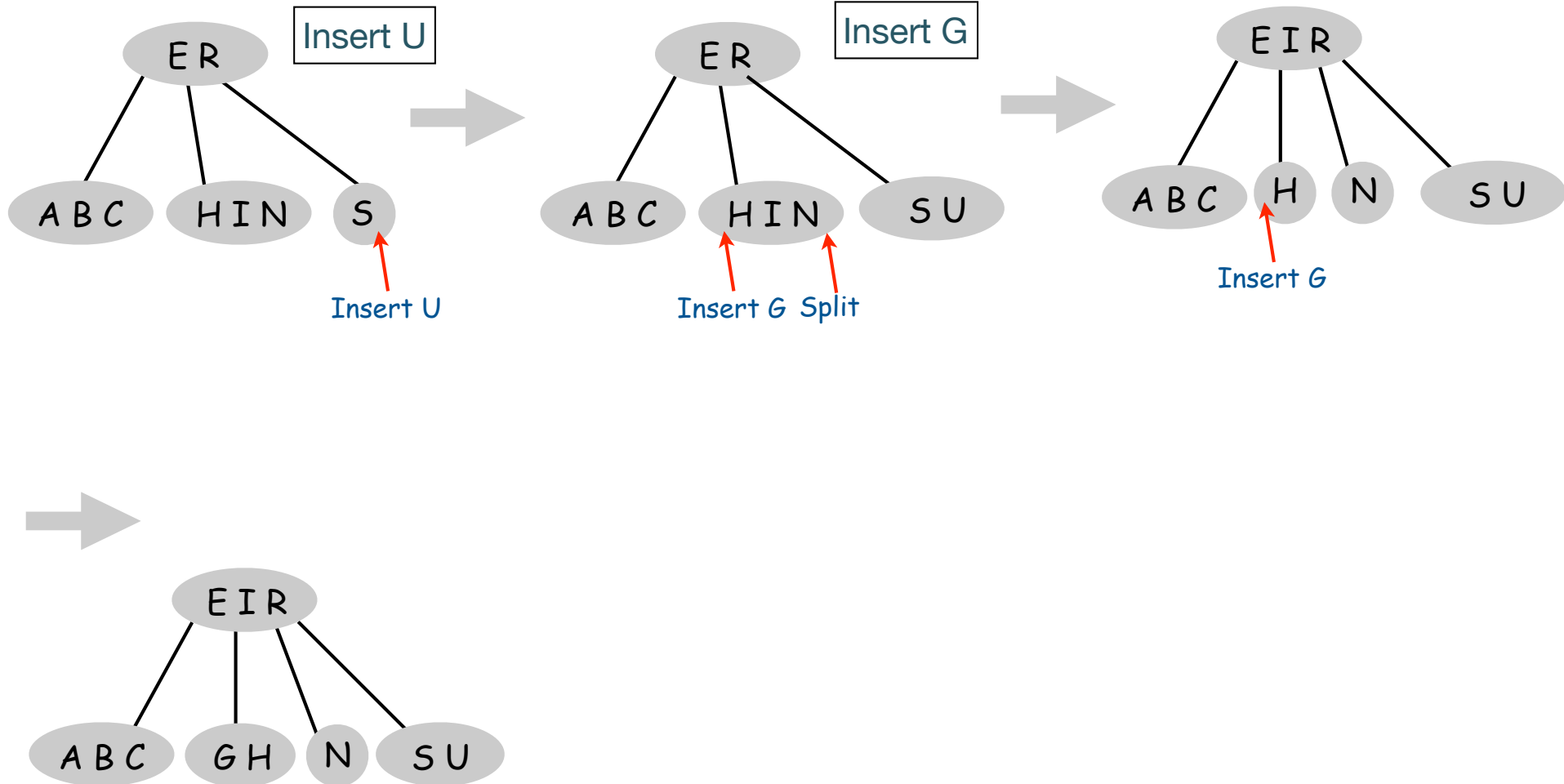
Insertion 2-3-4 trees



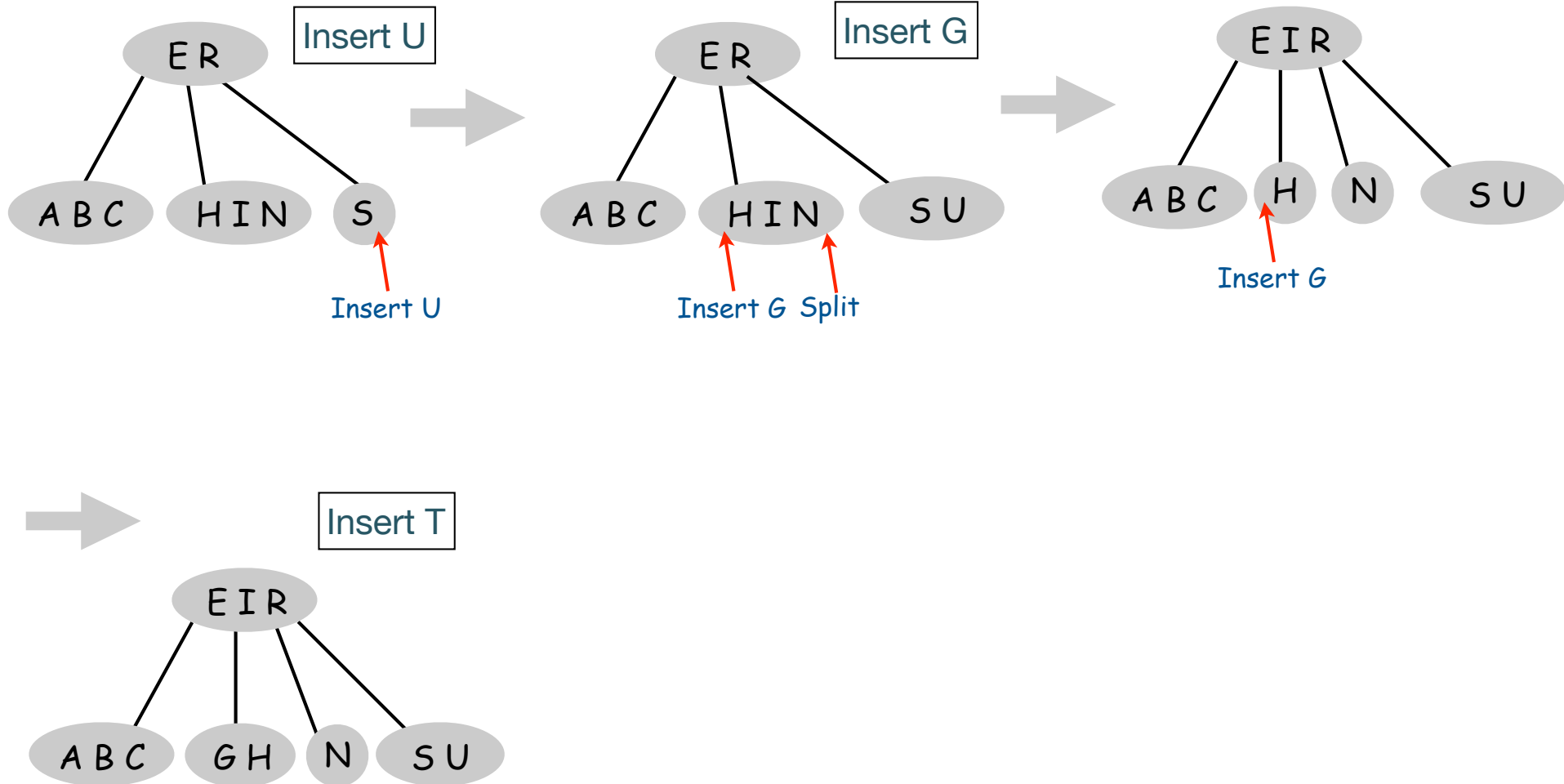
Insertion 2-3-4 trees



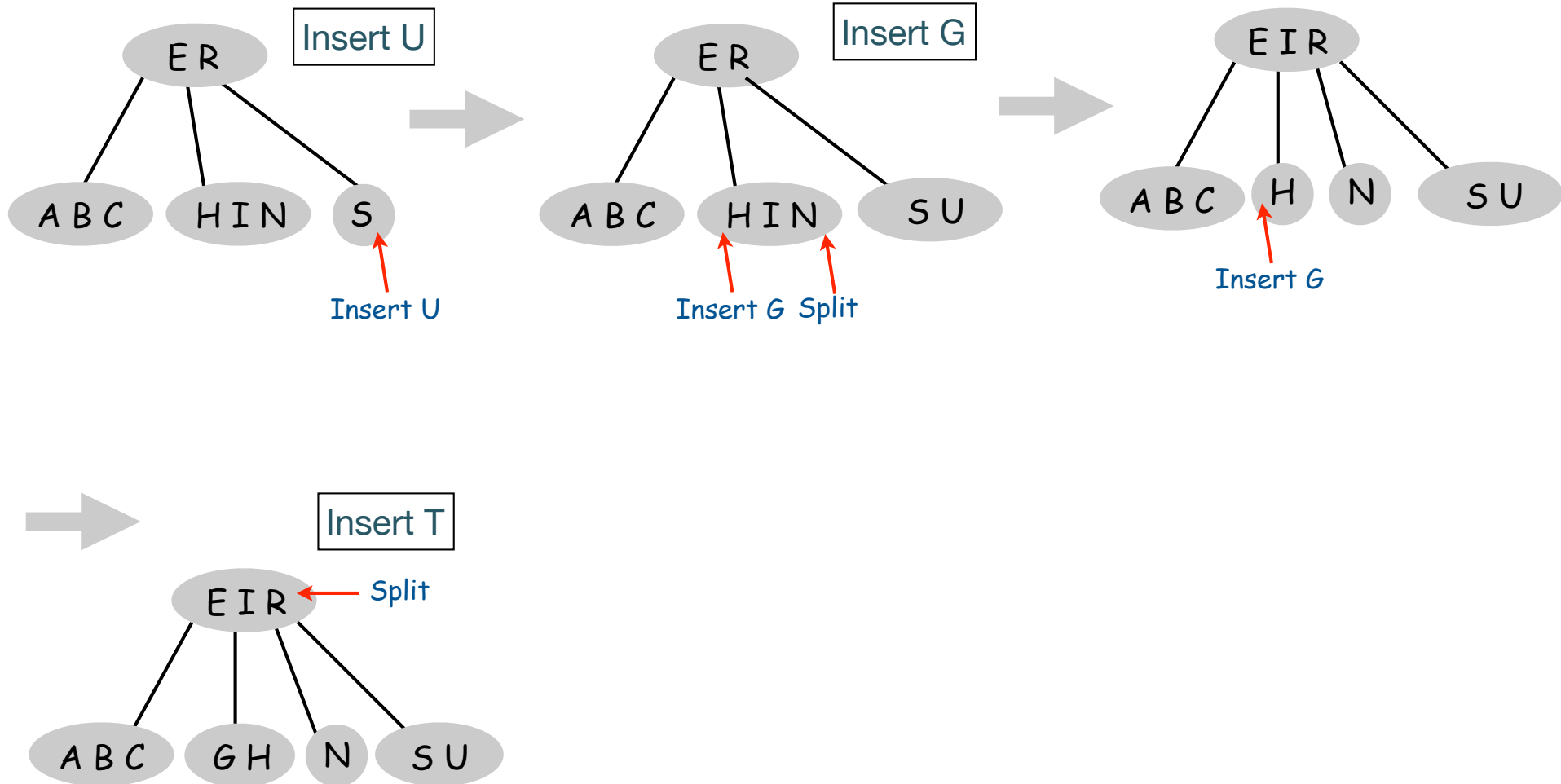
Insertion 2-3-4 trees



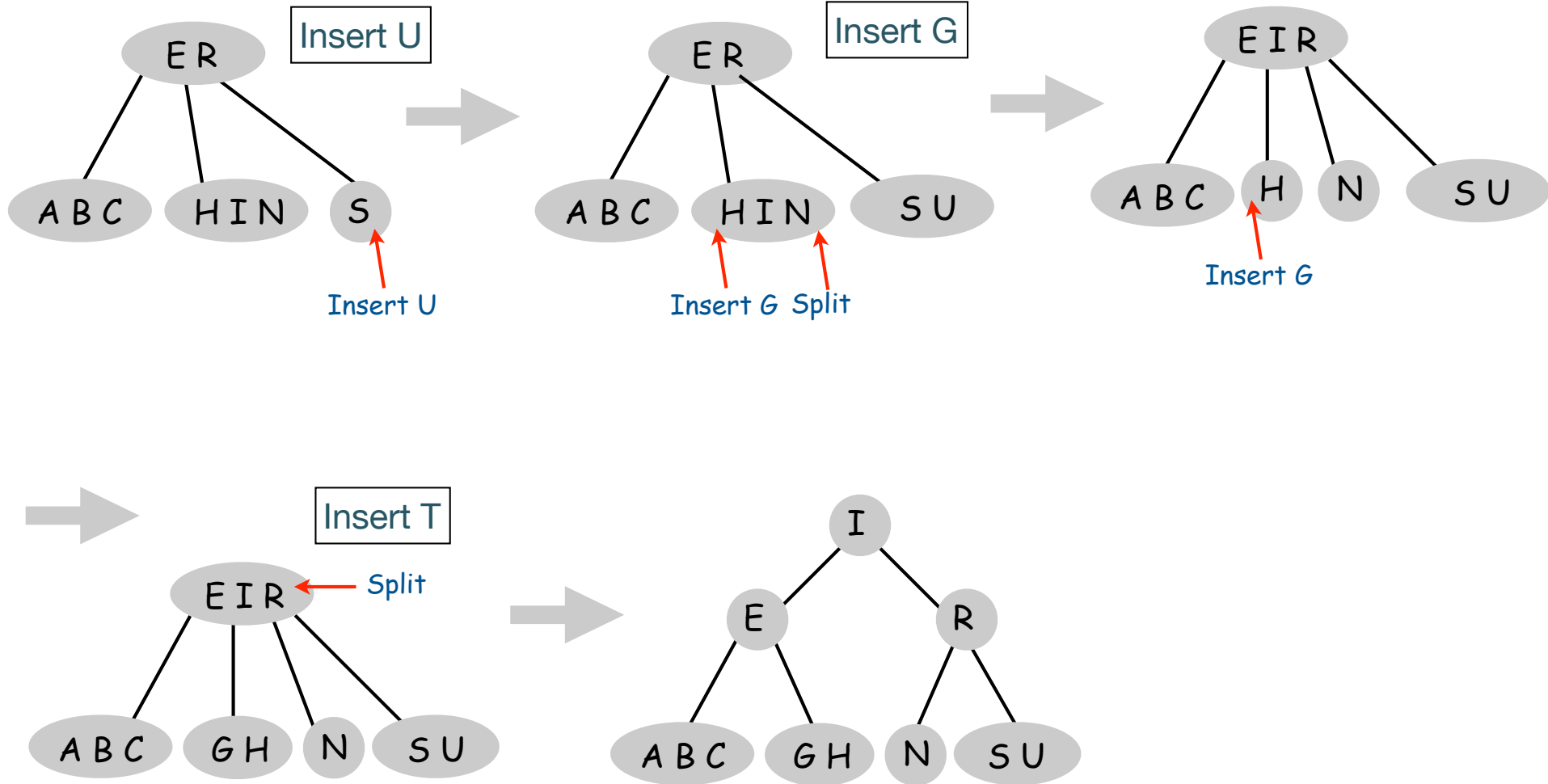
Insertion 2-3-4 trees



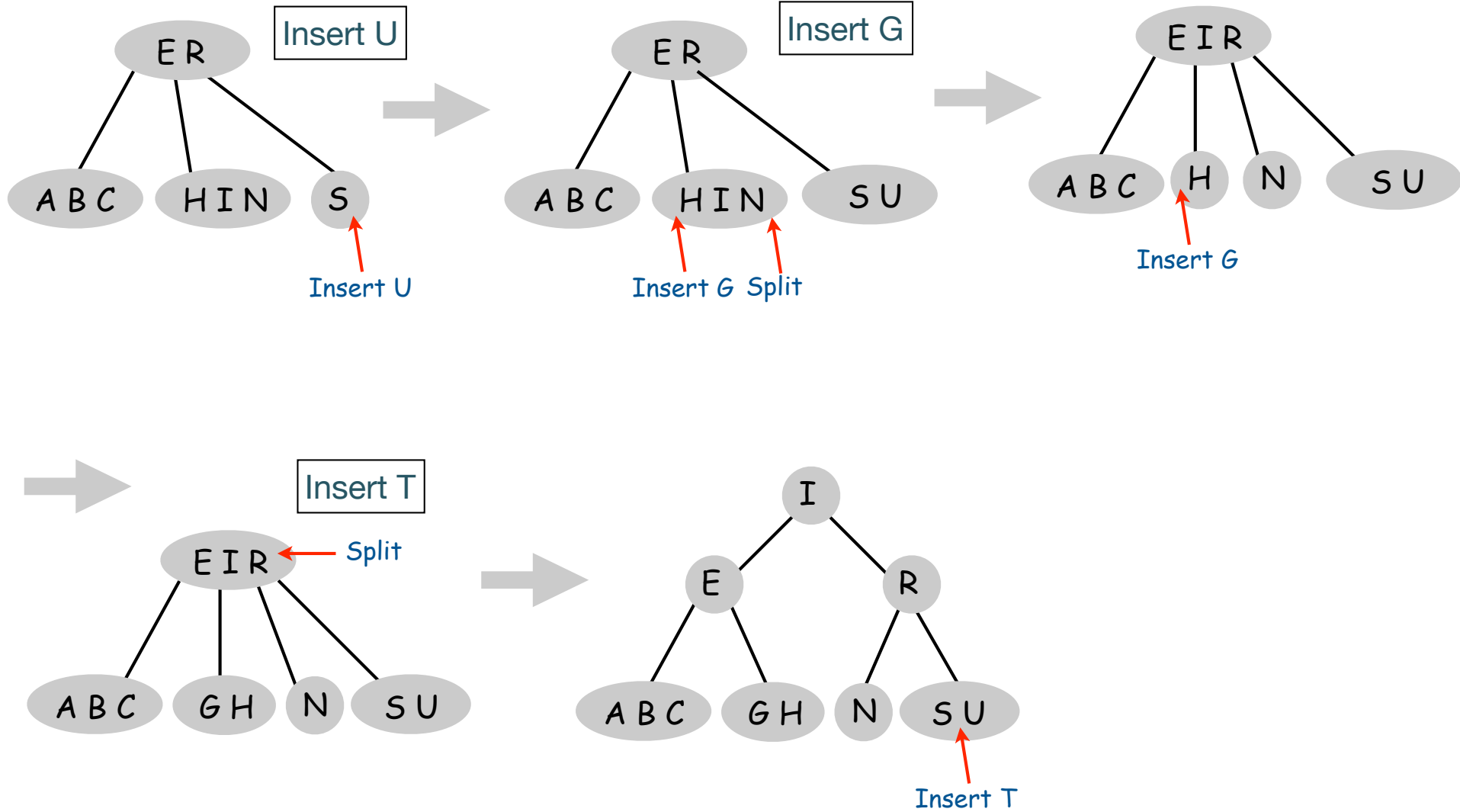
Insertion 2-3-4 trees



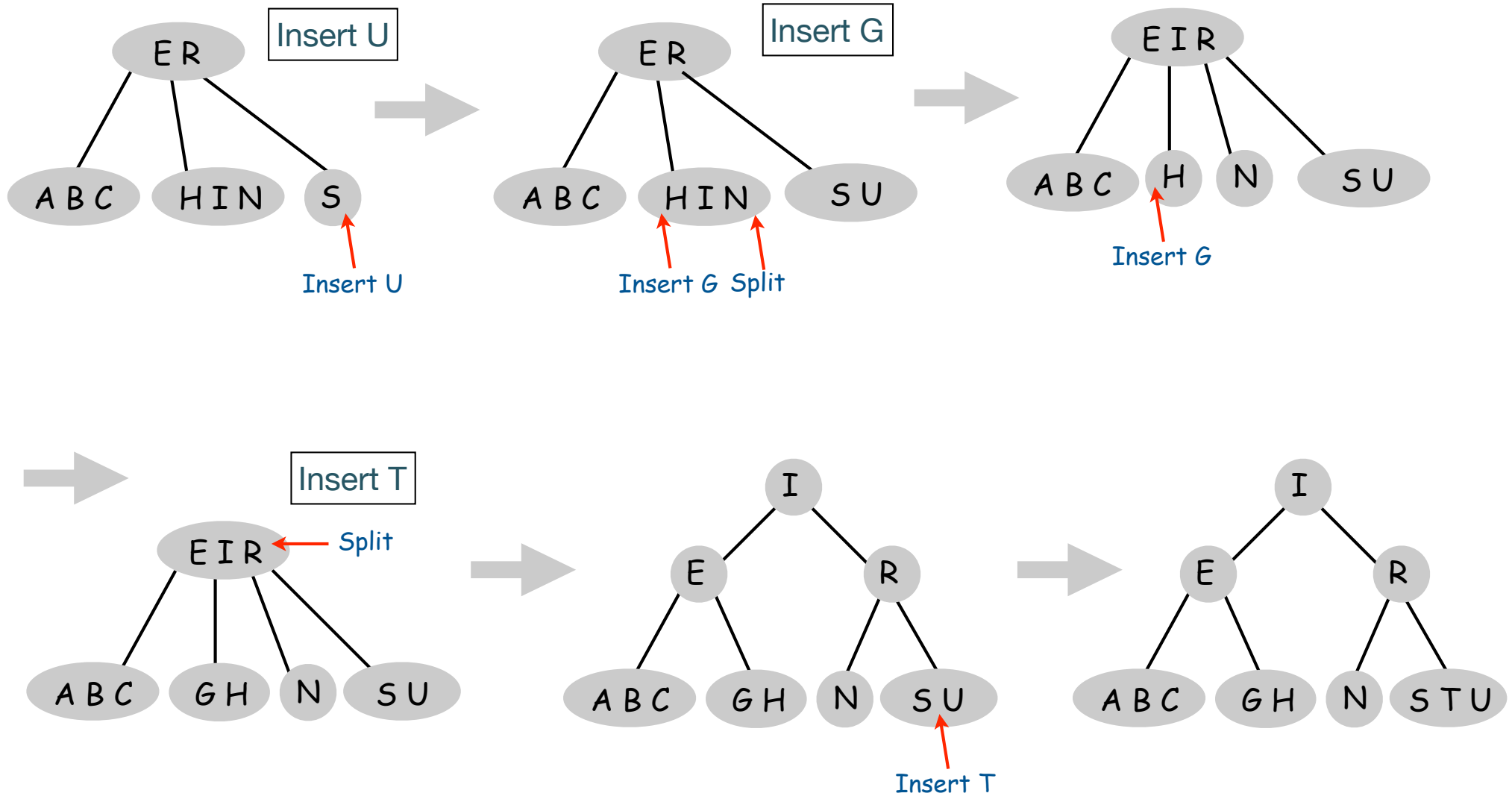
Insertion 2-3-4 trees



Insertion 2-3-4 trees



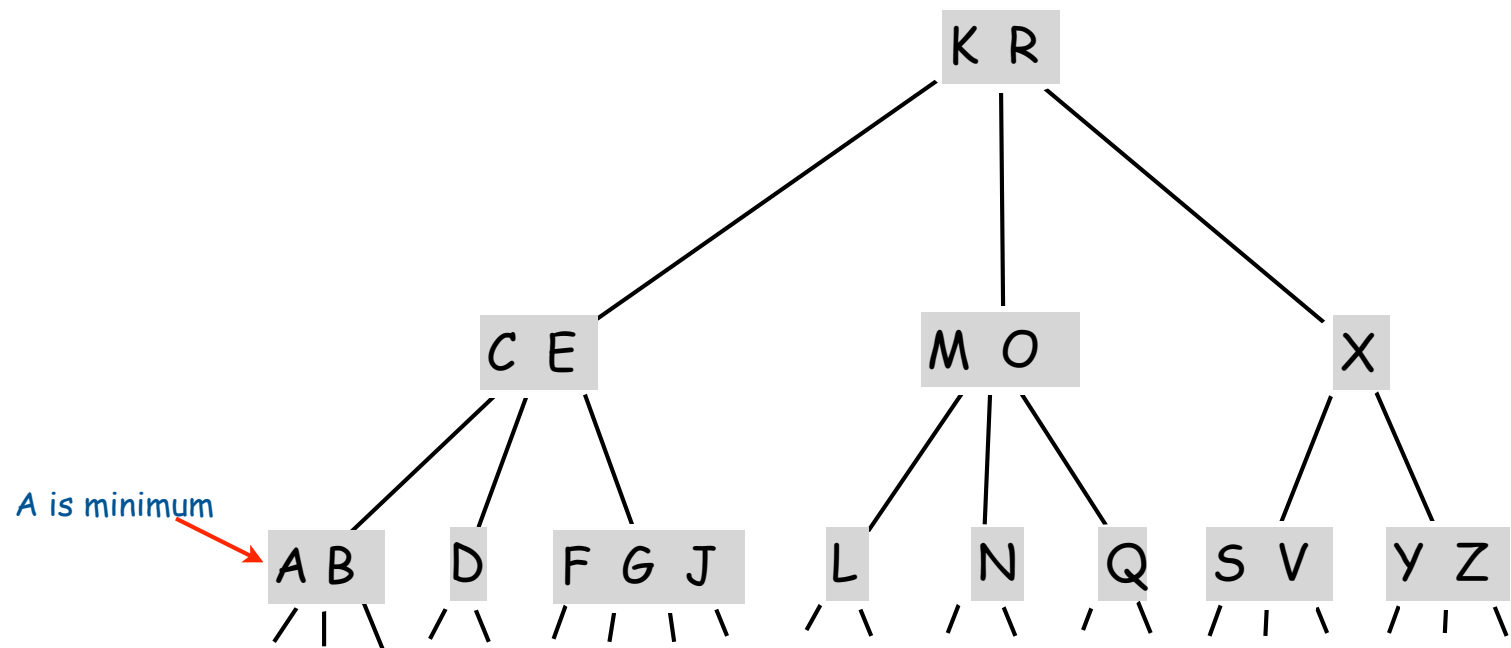
Insertion 2-3-4 trees



Deletions in 2-3-4 trees

Delete minimum:

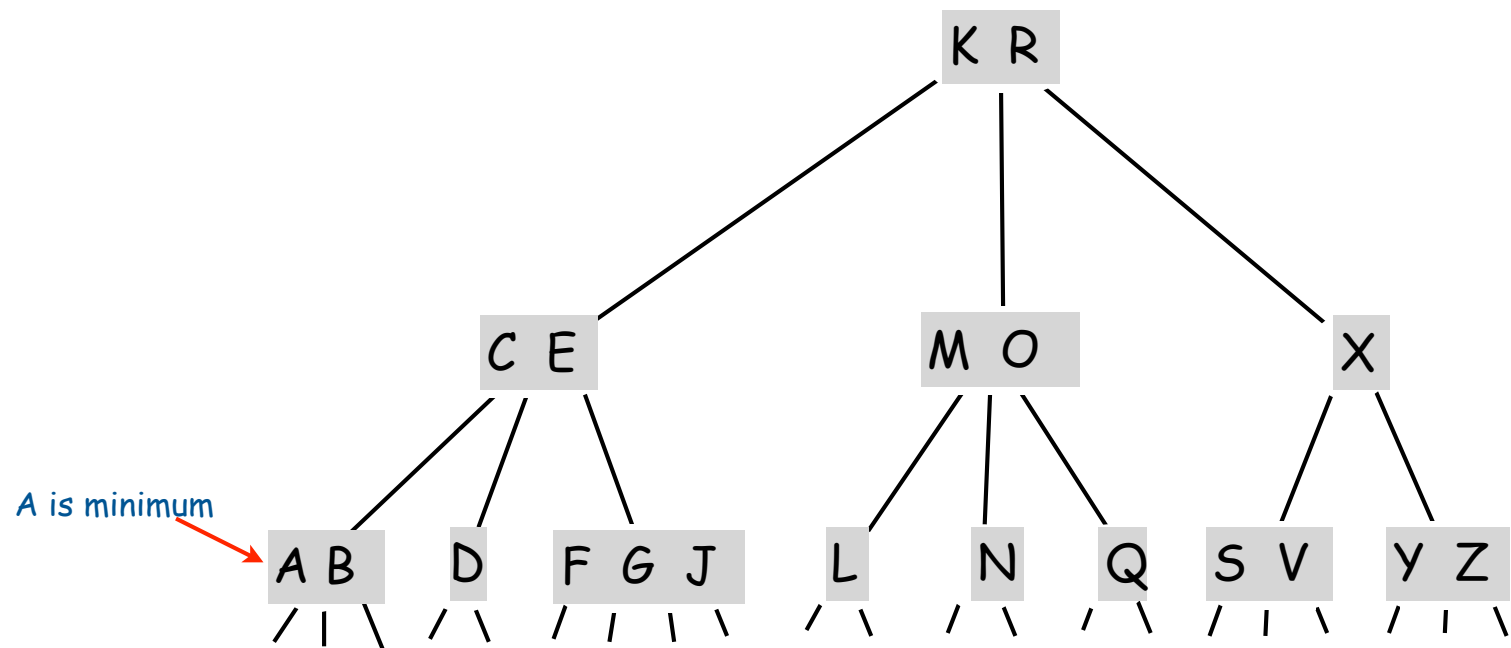
- minimum always in leftmost leaf



Deletions in 2-3-4 trees

Delete minimum:

- minimum always in leftmost leaf
- If 3- or 4-node: delete key

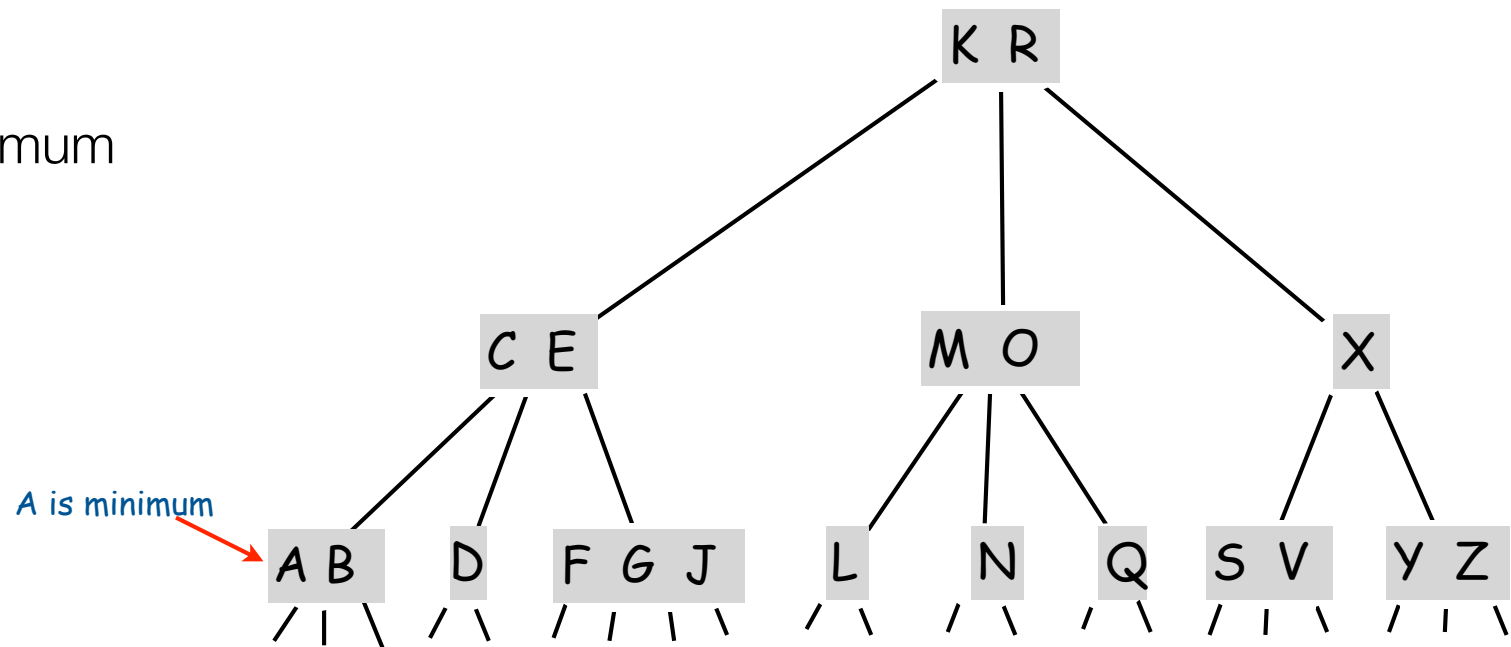


Deletions in 2-3-4 trees

Delete minimum:

- minimum always in leftmost leaf
- If 3- or 4-node: delete key

Ex. Delete minimum

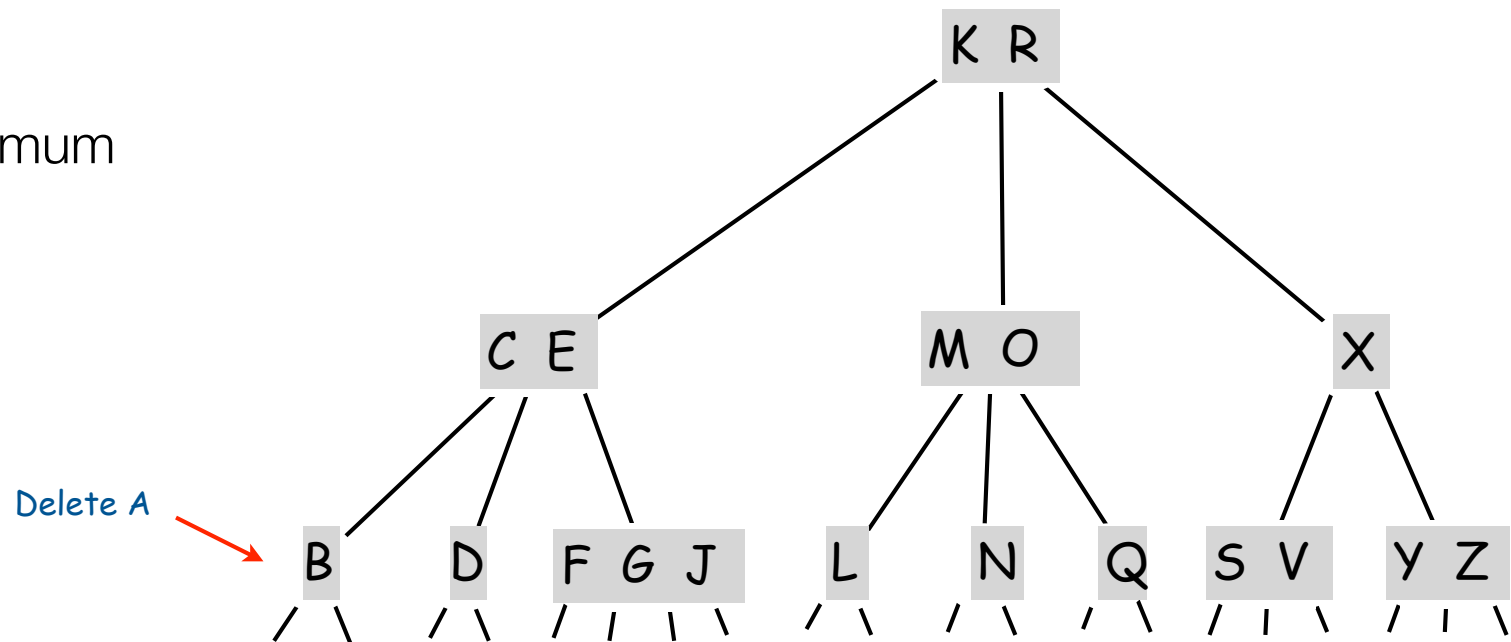


Deletions in 2-3-4 trees

Delete minimum:

- minimum always in leftmost leaf
- If 3- or 4-node: delete key

Ex. Delete minimum

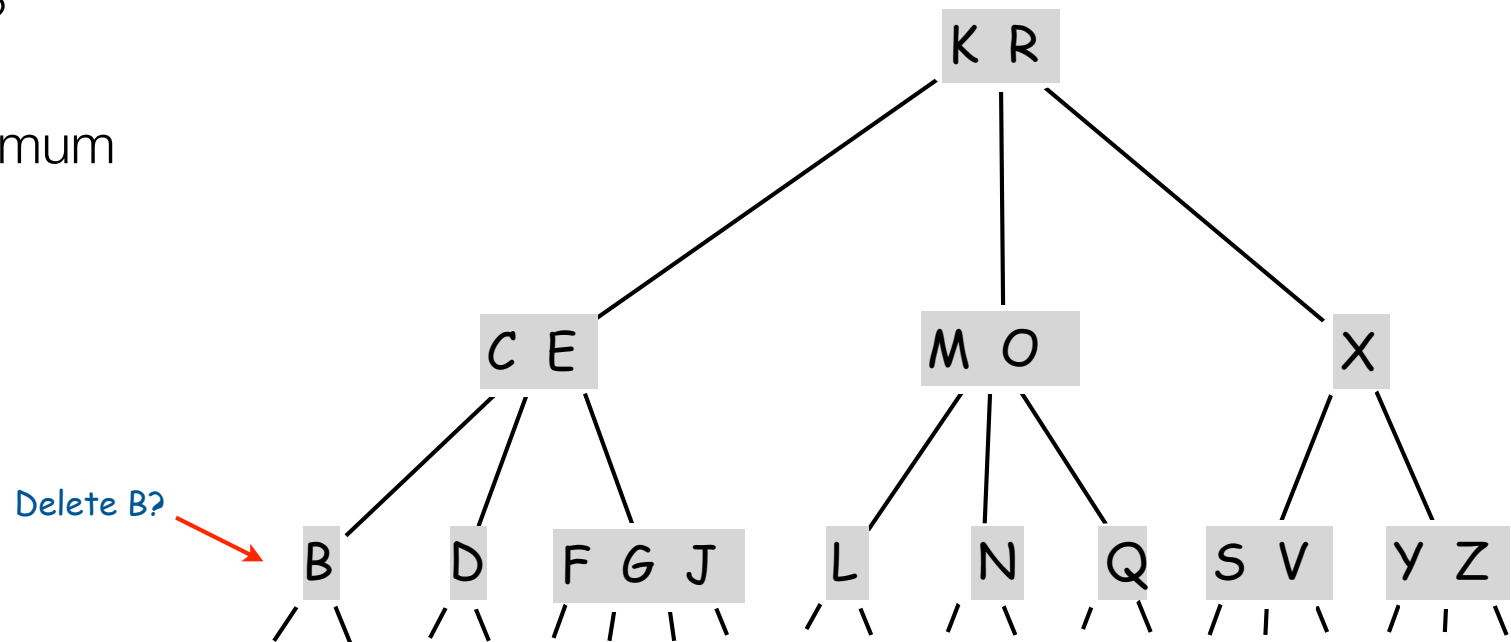


Deletions in 2-3-4 trees

Delete minimum:

- minimum always in leftmost leaf
- If 3- or 4-node: delete key
- 2-node??

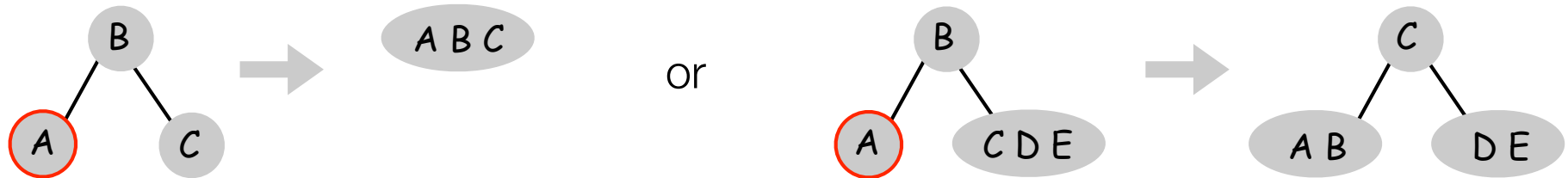
Ex. Delete minimum



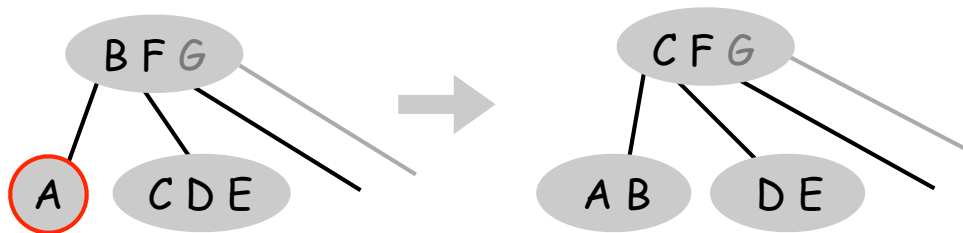
Deletions in 2-3-4 trees

Idea: On the way down maintain the invariant that current node is not a 2-node.

- Child of root and root is a 2-node:



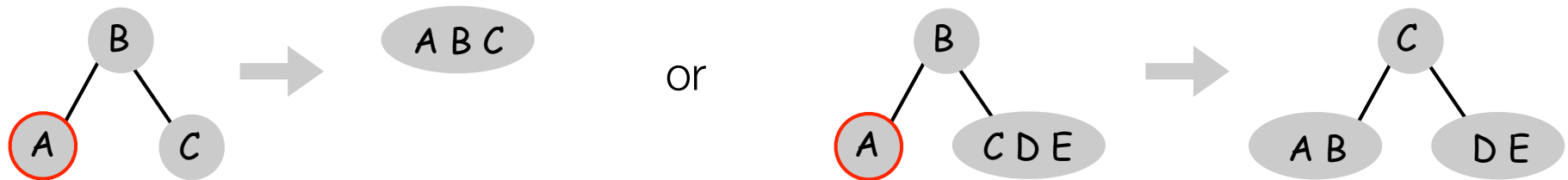
- on the way down:



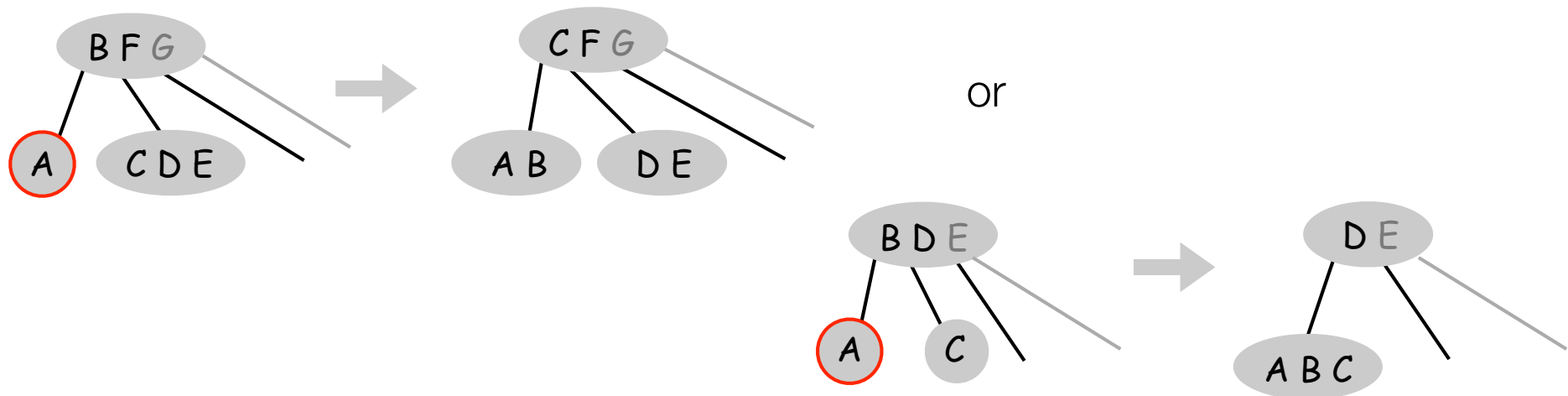
Deletions in 2-3-4 trees

Idea: On the way down maintain the invariant that current node is not a 2-node.

- Child of root and root is a 2-node:



- on the way down:

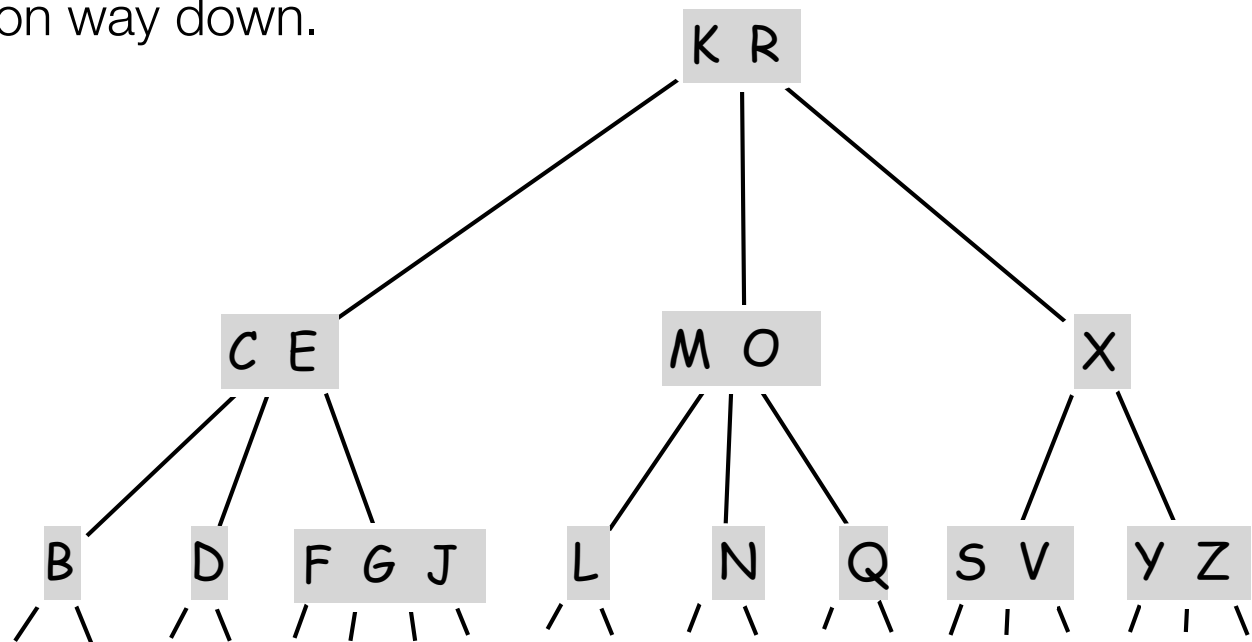


Deletions in 2-3-4 trees

Delete minimum:

- minimum always in leftmost leaf
- If 3- or 4-node: delete key
- 2-node: split/merge on way down.

Ex. Delete minimum

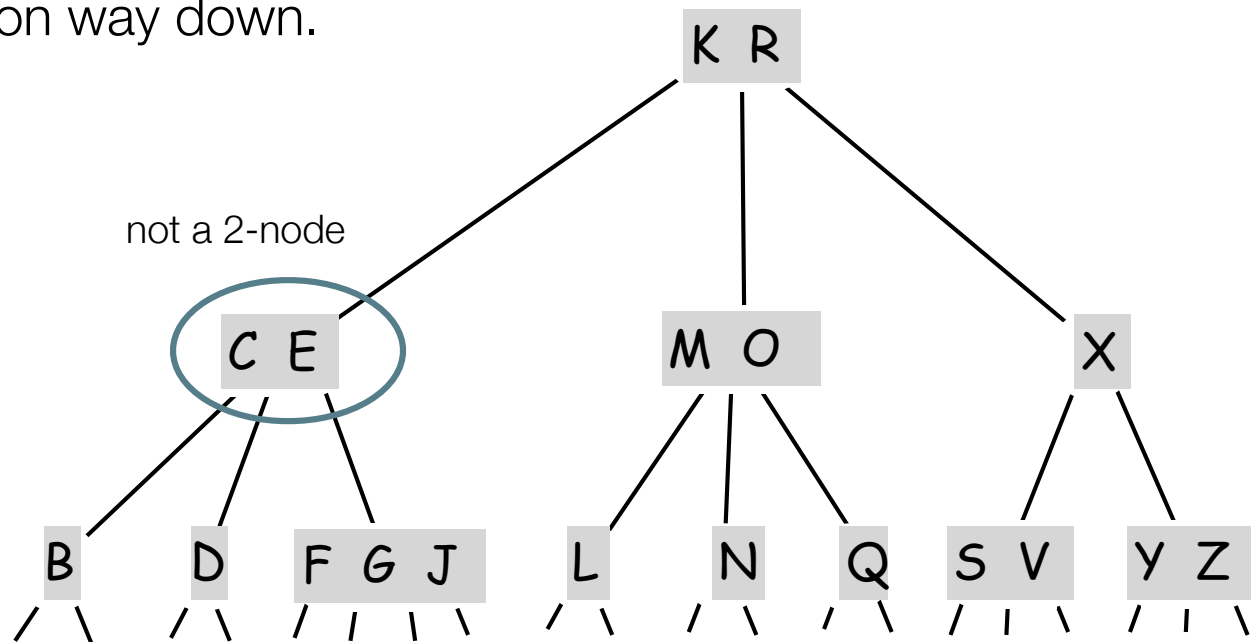


Deletions in 2-3-4 trees

Delete minimum:

- minimum always in leftmost leaf
- If 3- or 4-node: delete key
- 2-node: split/merge on way down.

Ex. Delete minimum

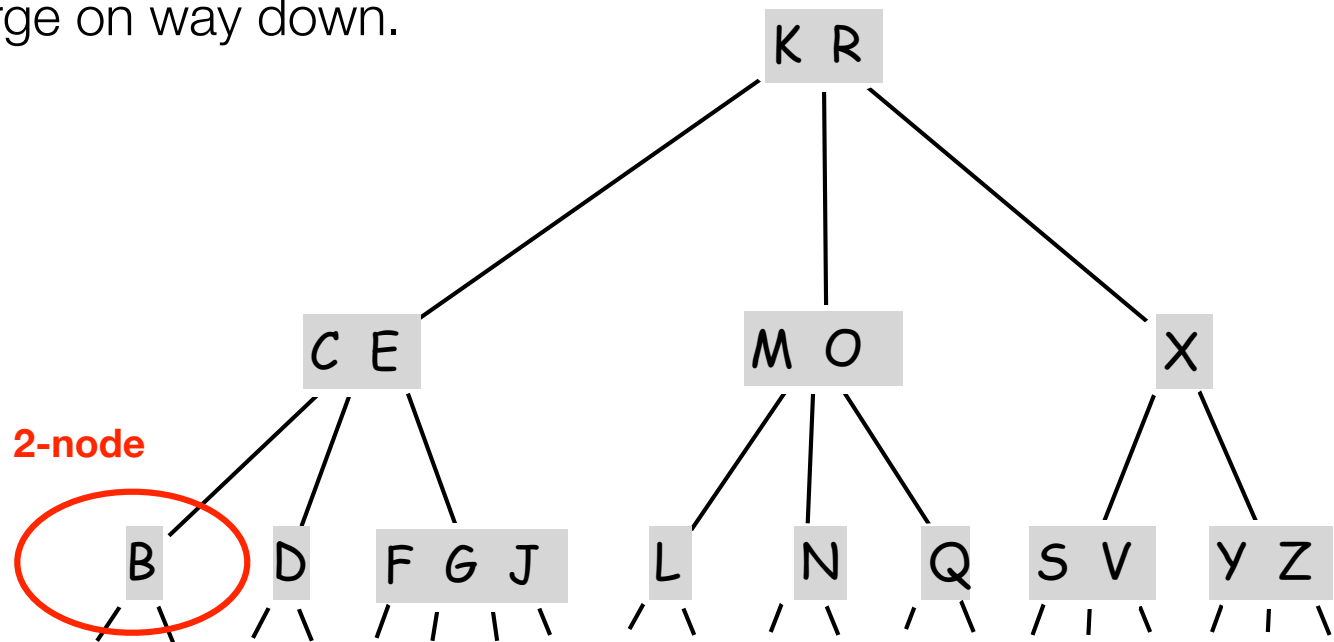


Deletions in 2-3-4 trees

Delete minimum:

- minimum always in leftmost leaf
- If 3- or 4-node: delete key
- 2-node: split/merge on way down.

Ex. Delete minimum

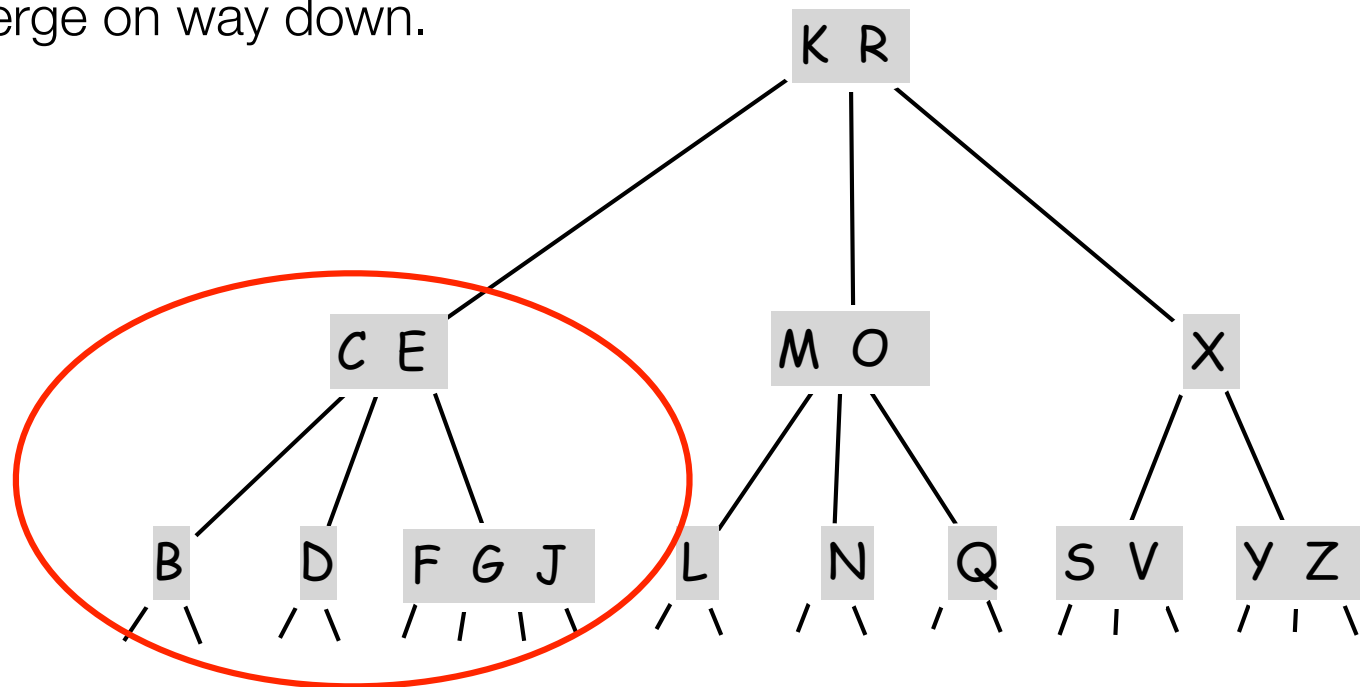


Deletions in 2-3-4 trees

Delete minimum:

- minimum always in leftmost leaf
- If 3- or 4-node: delete key
- 2-node: split/merge on way down.

Ex. Delete minimum

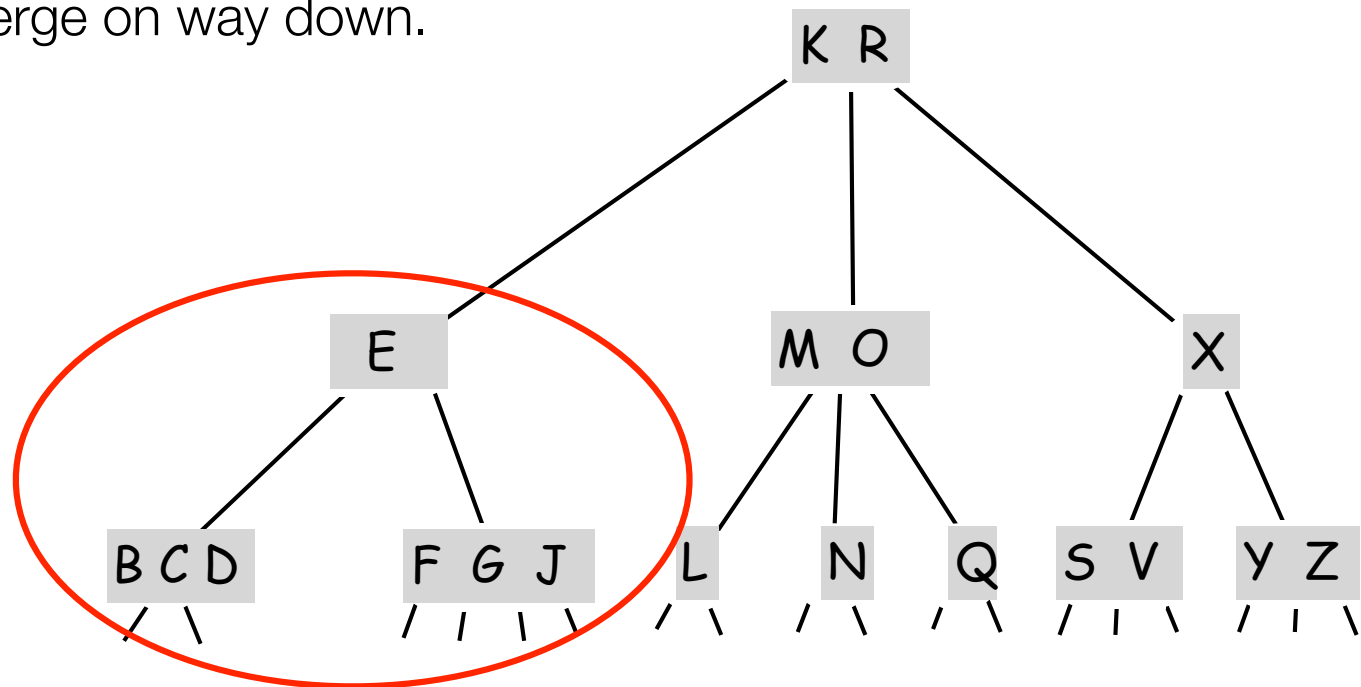


Deletions in 2-3-4 trees

Delete minimum:

- minimum always in leftmost leaf
- If 3- or 4-node: delete key
- 2-node: split/merge on way down.

Ex. Delete minimum

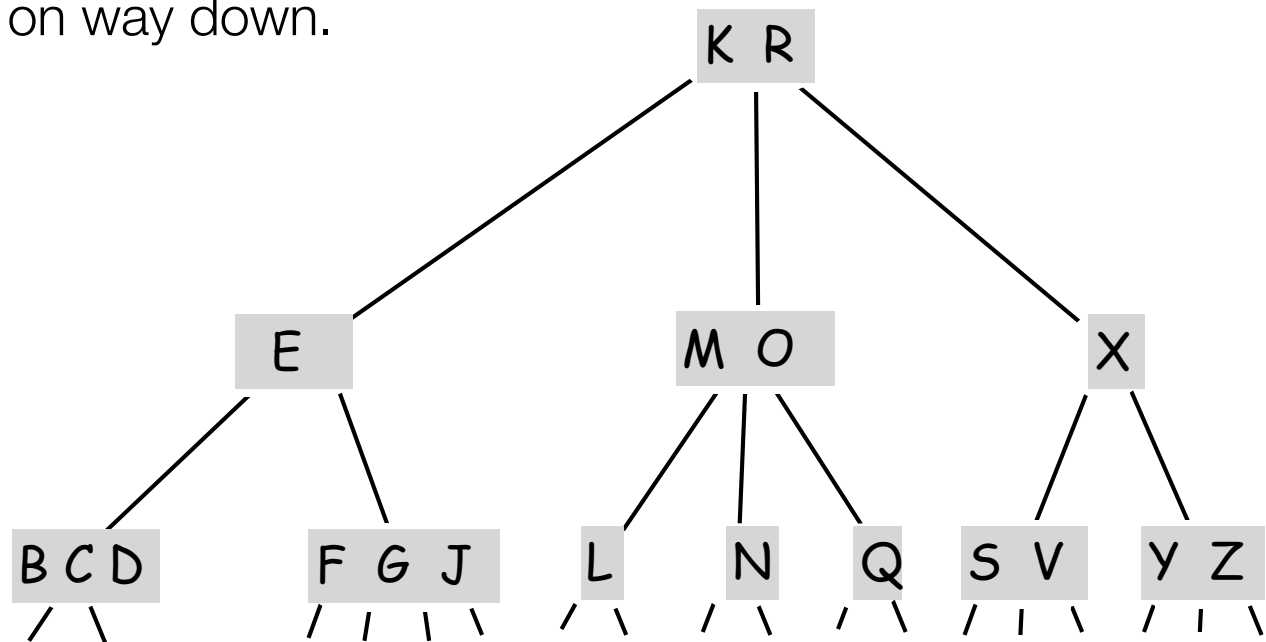


Deletions in 2-3-4 trees

Delete minimum:

- minimum always in leftmost leaf
- If 3- or 4-node: delete key
- 2-node: split/merge on way down.

Ex. Delete minimum

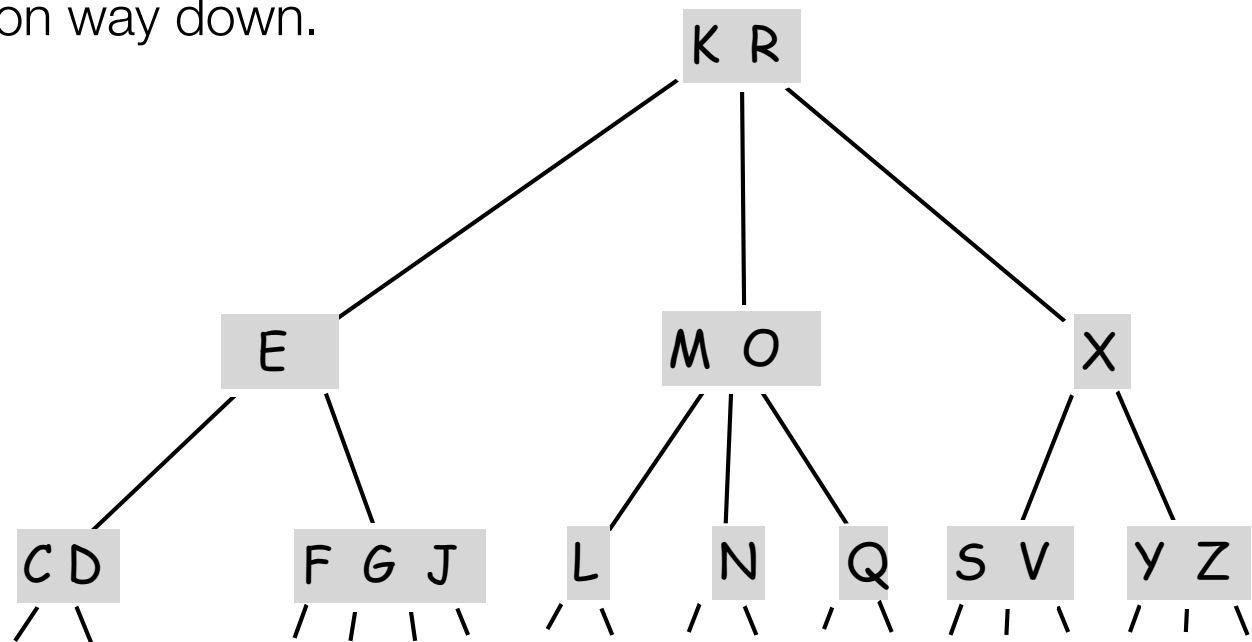


Deletions in 2-3-4 trees

Delete minimum:

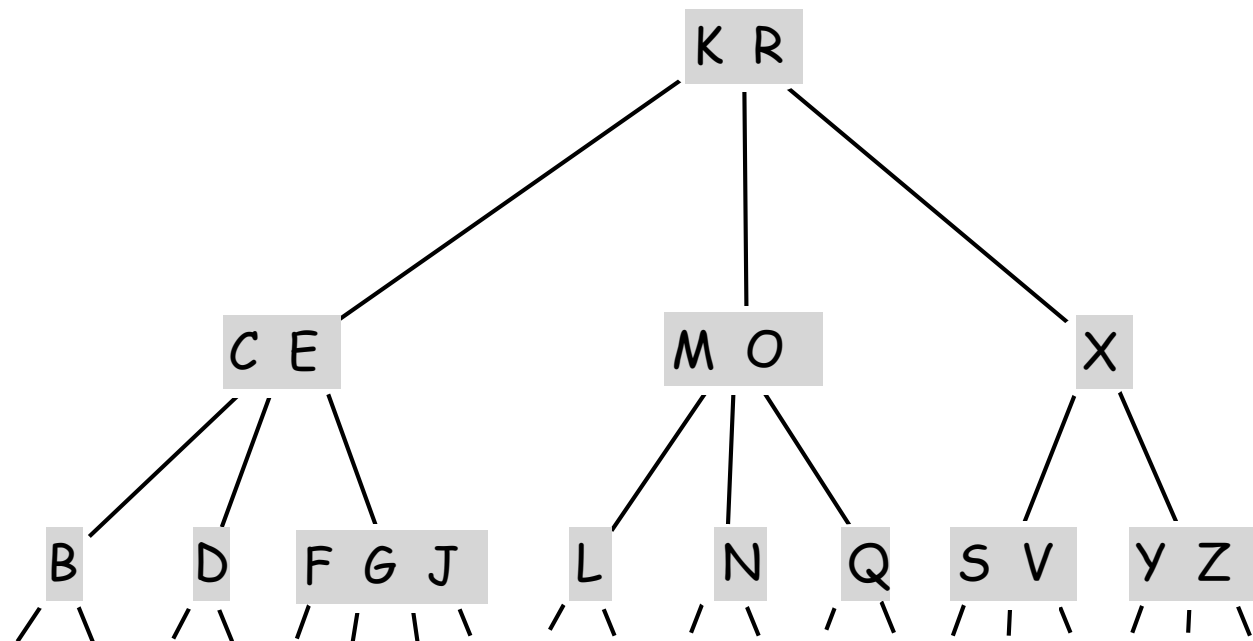
- minimum always in leftmost leaf
- If 3- or 4-node: delete key
- 2-node: split/merge on way down.

Ex. Delete minimum



Deletions in 2-3-4 trees

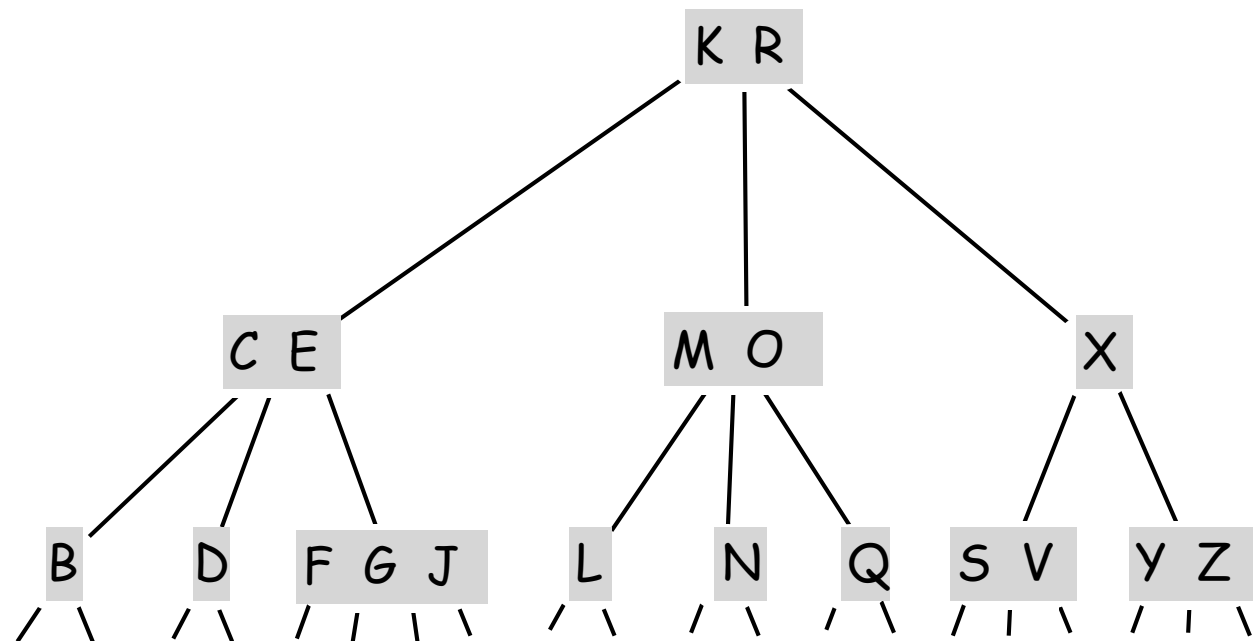
Delete:



Deletions in 2-3-4 trees

Delete:

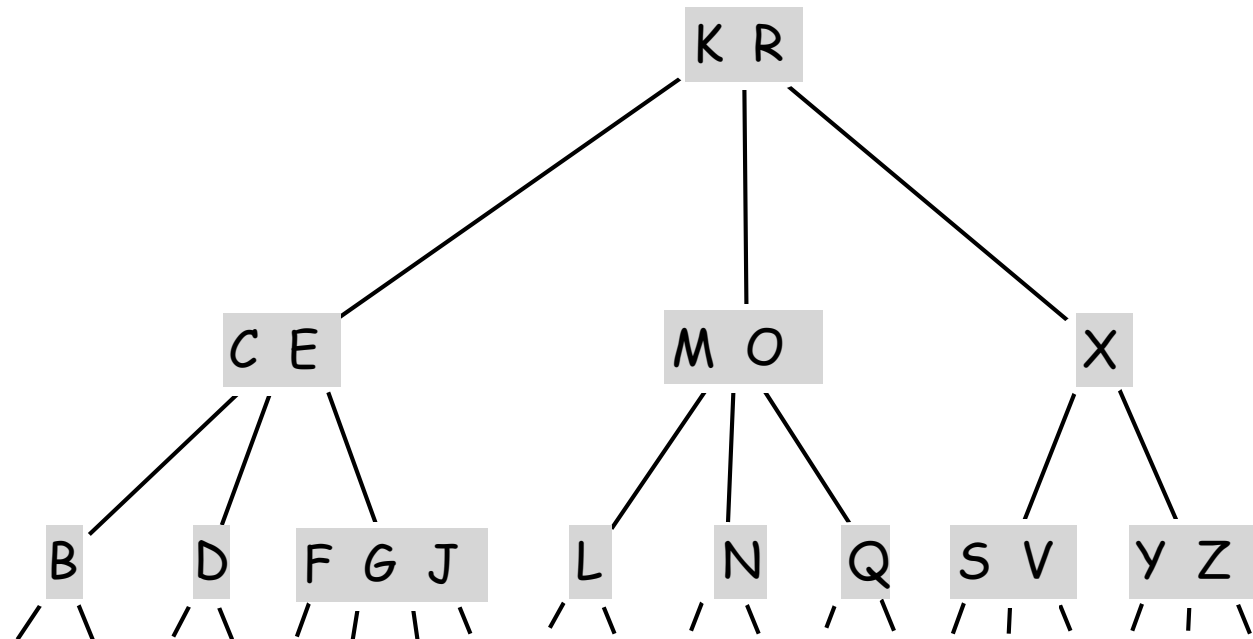
- During search maintain invariant that current node is not a 2-node



Deletions in 2-3-4 trees

Delete:

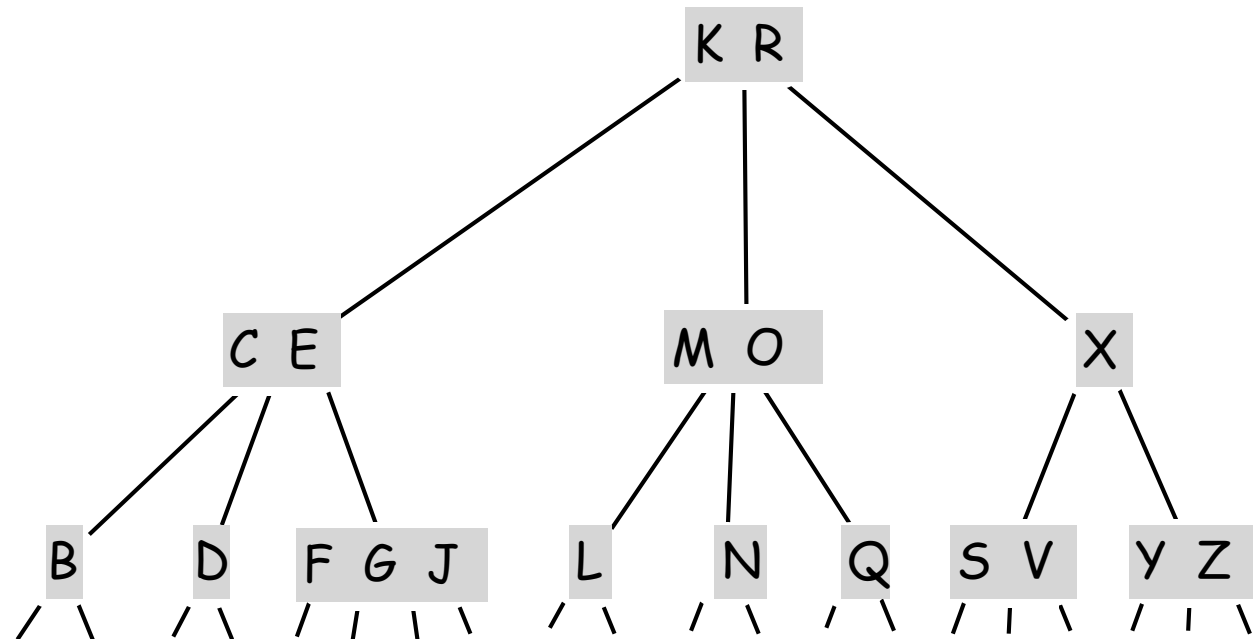
- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key



Deletions in 2-3-4 trees

Delete:

- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key
- Key not in leaf: replace with successor (always leaf in subtree) and delete successor from leaf.

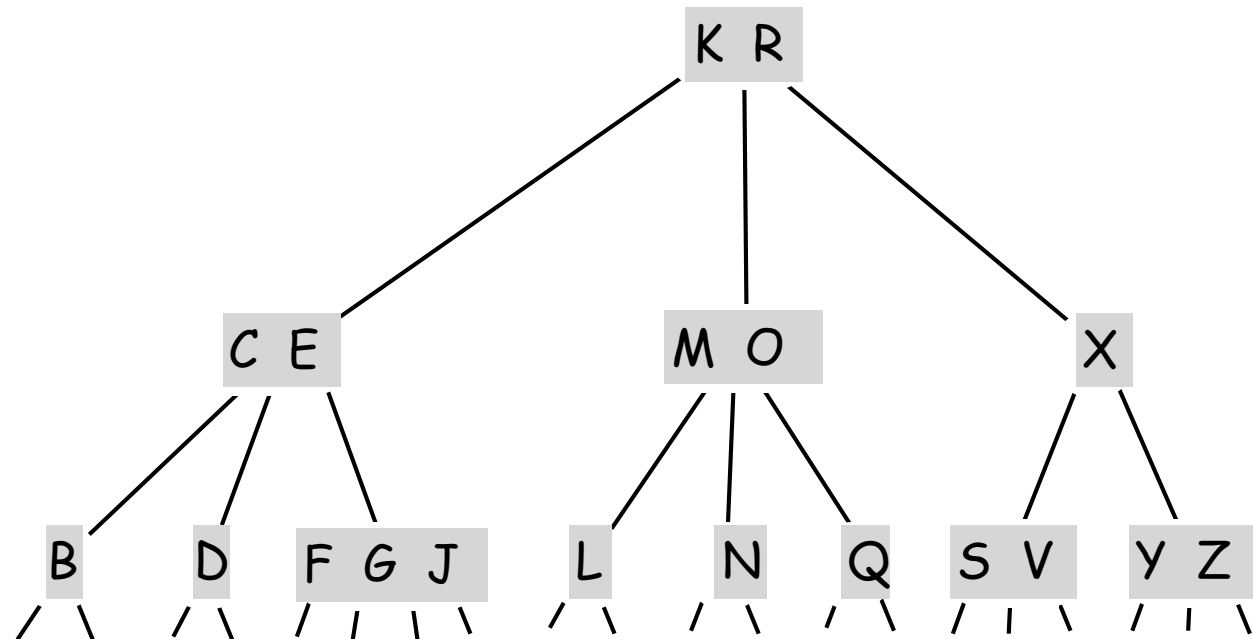


Deletions in 2-3-4 trees

Delete:

- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key
- Key not in leaf: replace with successor (always leaf in subtree) and delete successor from leaf.

Ex. Delete K



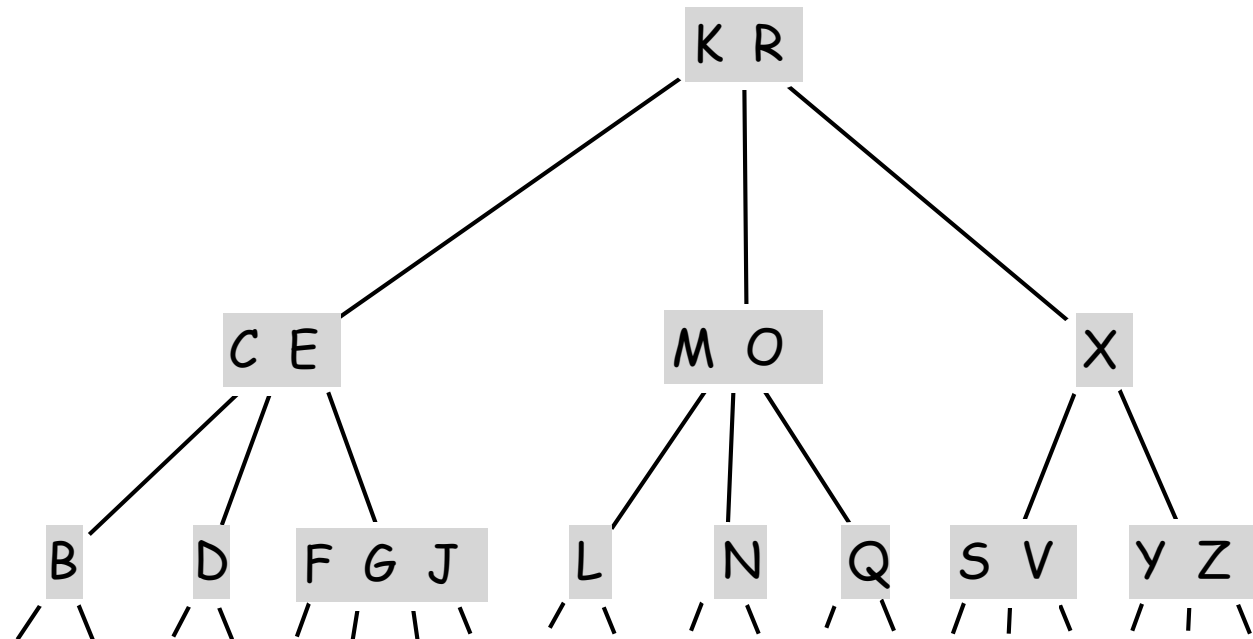
Deletions in 2-3-4 trees

Delete:

- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key
- Key not in leaf: replace with successor (always leaf in subtree) and delete successor from leaf.

Ex. Delete K

- Find successor



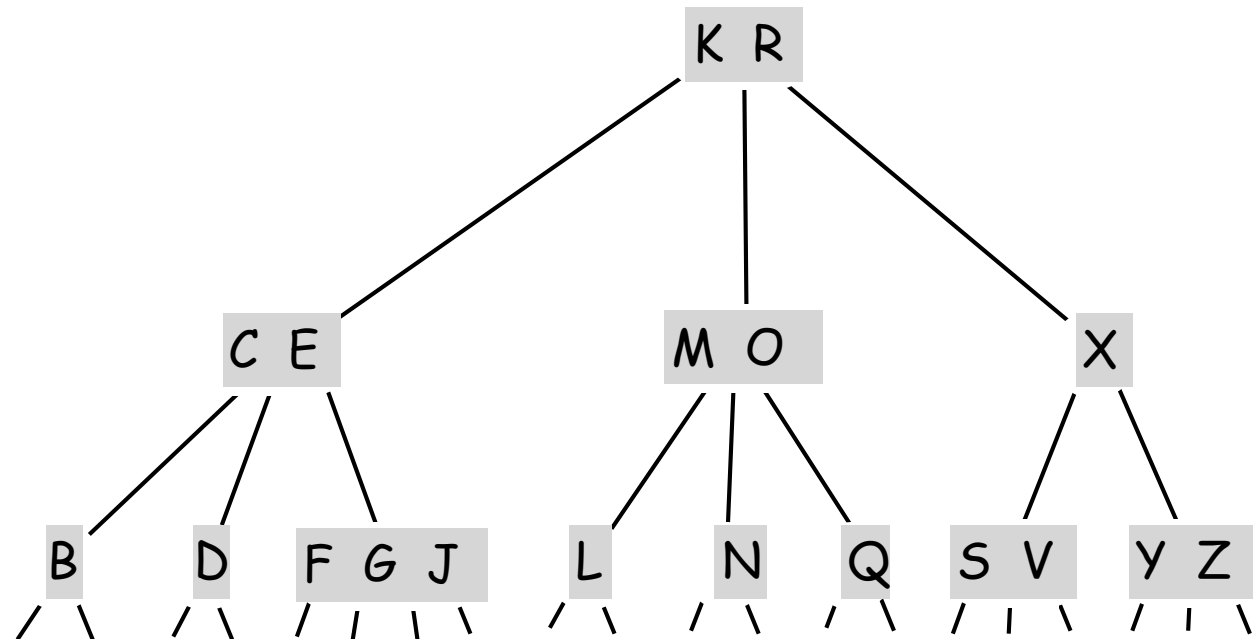
Deletions in 2-3-4 trees

Delete:

- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key
- Key not in leaf: replace with successor (always leaf in subtree) and delete successor from leaf.

Ex. Delete K

- Find successor



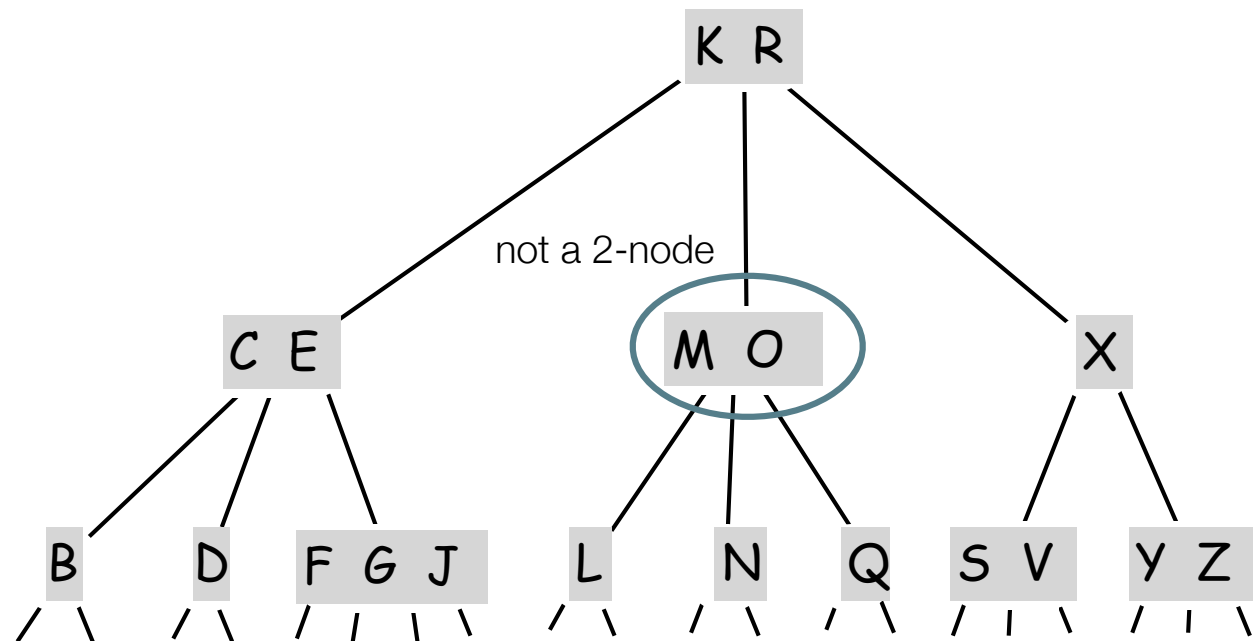
Deletions in 2-3-4 trees

Delete:

- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key
- Key not in leaf: replace with successor (always leaf in subtree) and delete successor from leaf.

Ex. Delete K

- Find successor



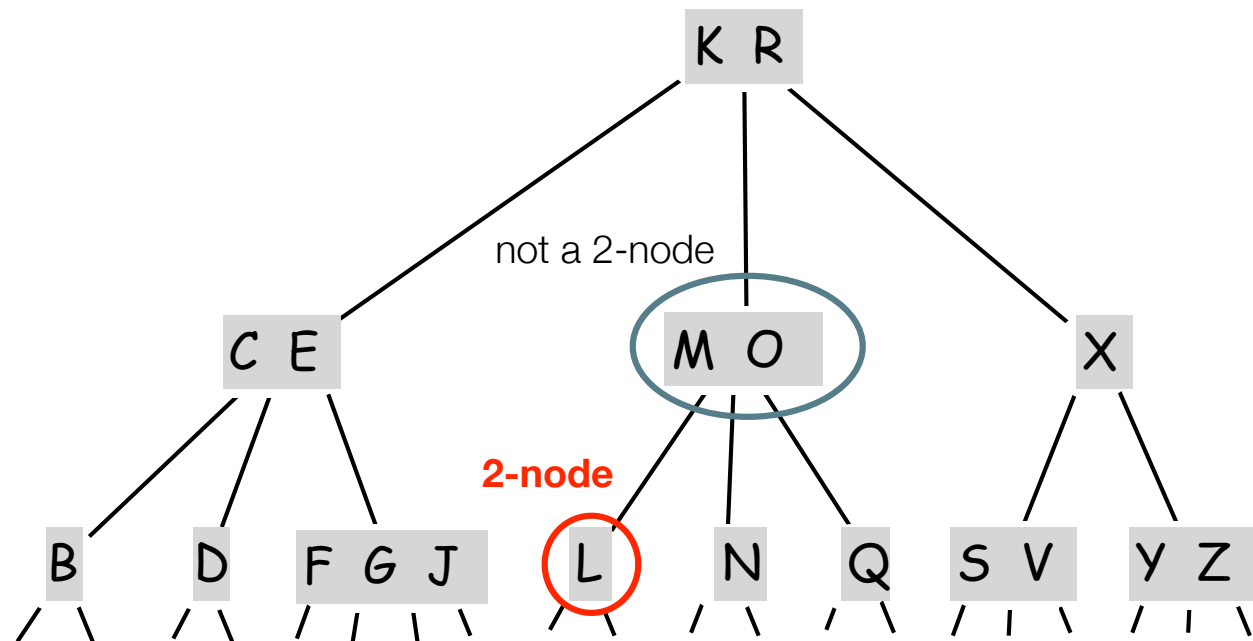
Deletions in 2-3-4 trees

Delete:

- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key
- Key not in leaf: replace with successor (always leaf in subtree) and delete successor from leaf.

Ex. Delete K

- Find successor



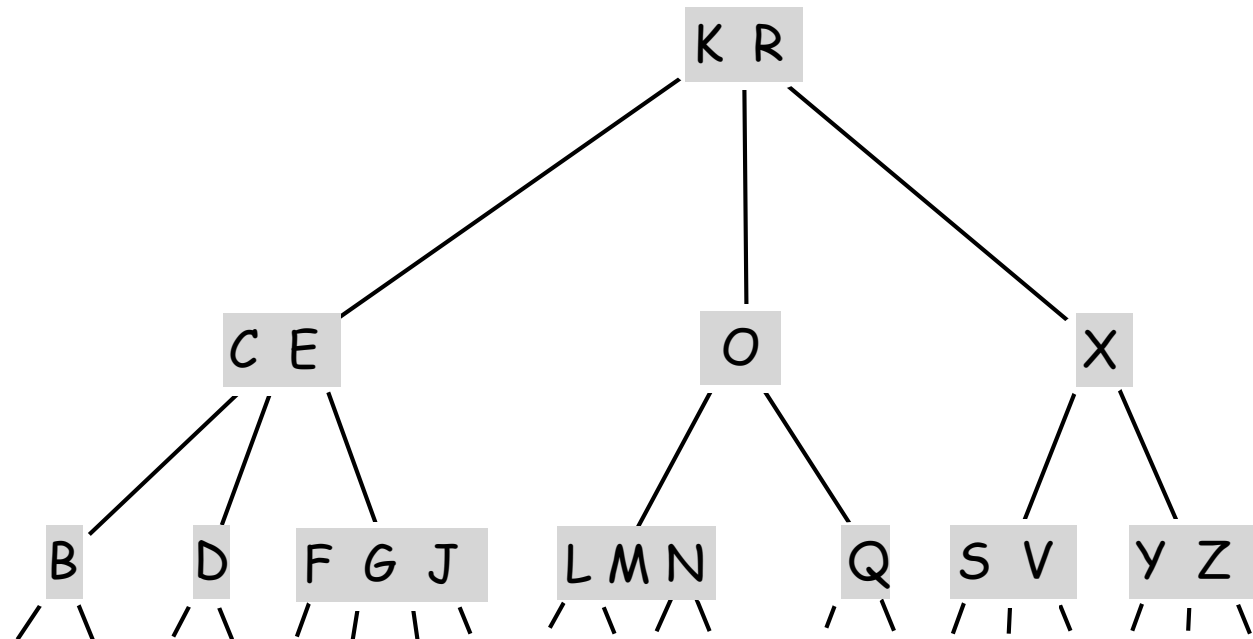
Deletions in 2-3-4 trees

Delete:

- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key
- Key not in leaf: replace with successor (always leaf in subtree) and delete successor from leaf.

Ex. Delete K

- Find successor



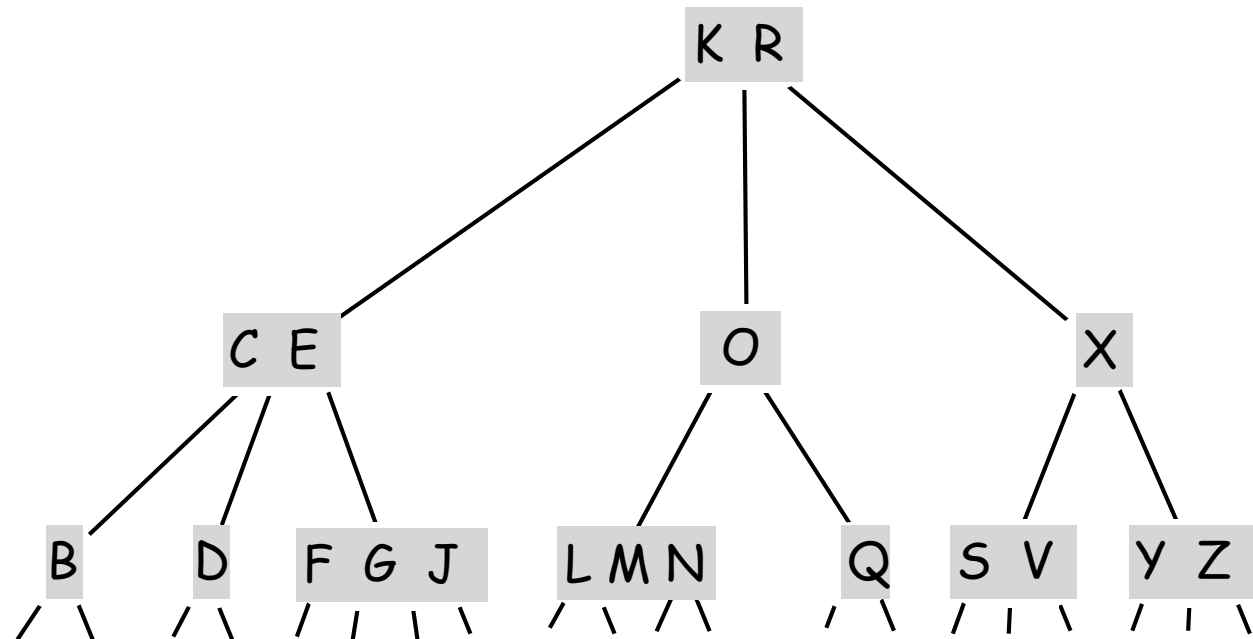
Deletions in 2-3-4 trees

Delete:

- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key
- Key not in leaf: replace with successor (always leaf in subtree) and delete successor from leaf.

Ex. Delete K

- Find successor
- Delete L from leaf



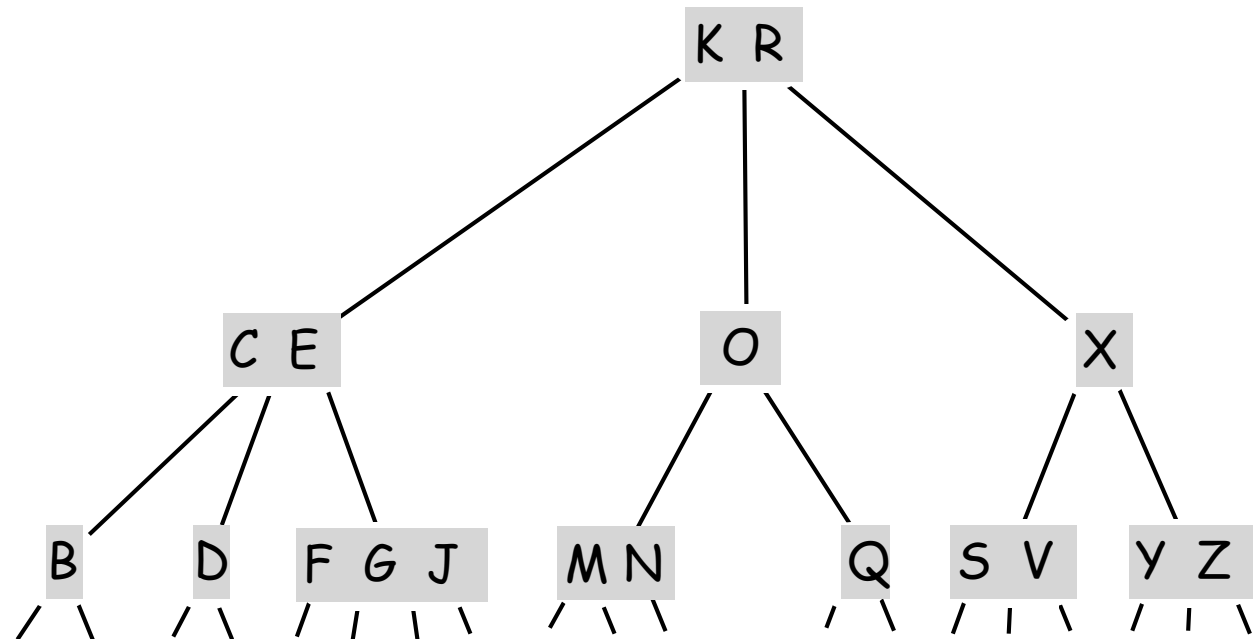
Deletions in 2-3-4 trees

Delete:

- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key
- Key not in leaf: replace with successor (always leaf in subtree) and delete successor from leaf.

Ex. Delete K

- Find successor
- Delete L from leaf



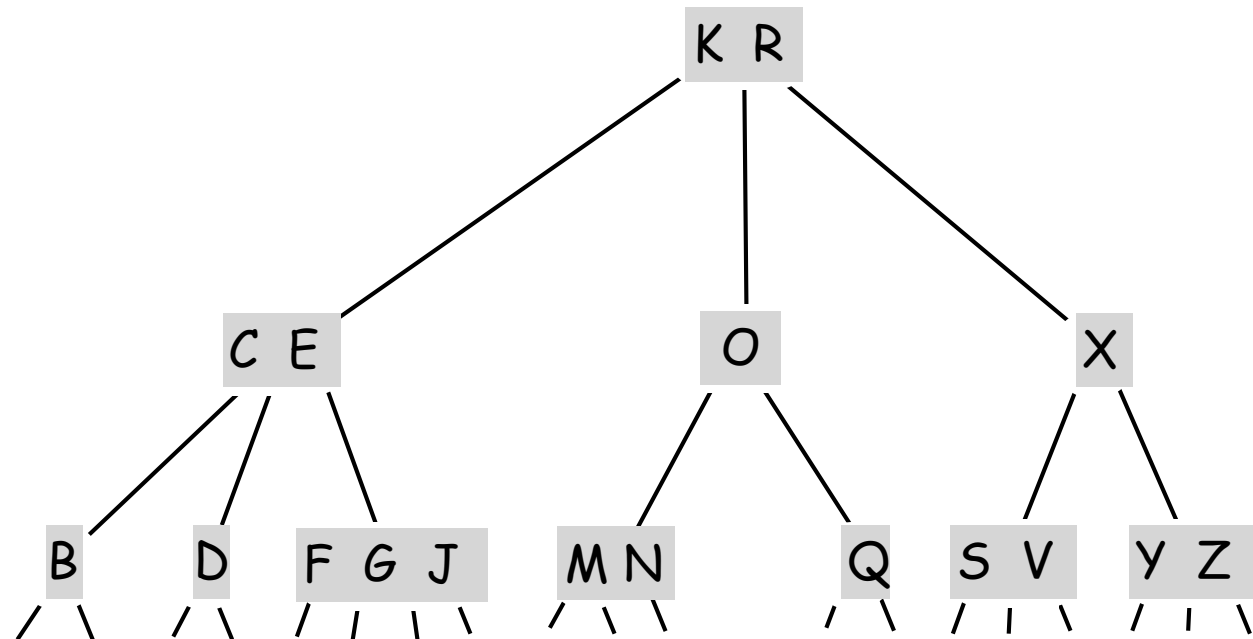
Deletions in 2-3-4 trees

Delete:

- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key
- Key not in leaf: replace with successor (always leaf in subtree) and delete successor from leaf.

Ex. Delete K

- Find successor
- Delete L from leaf
- Replace K with L



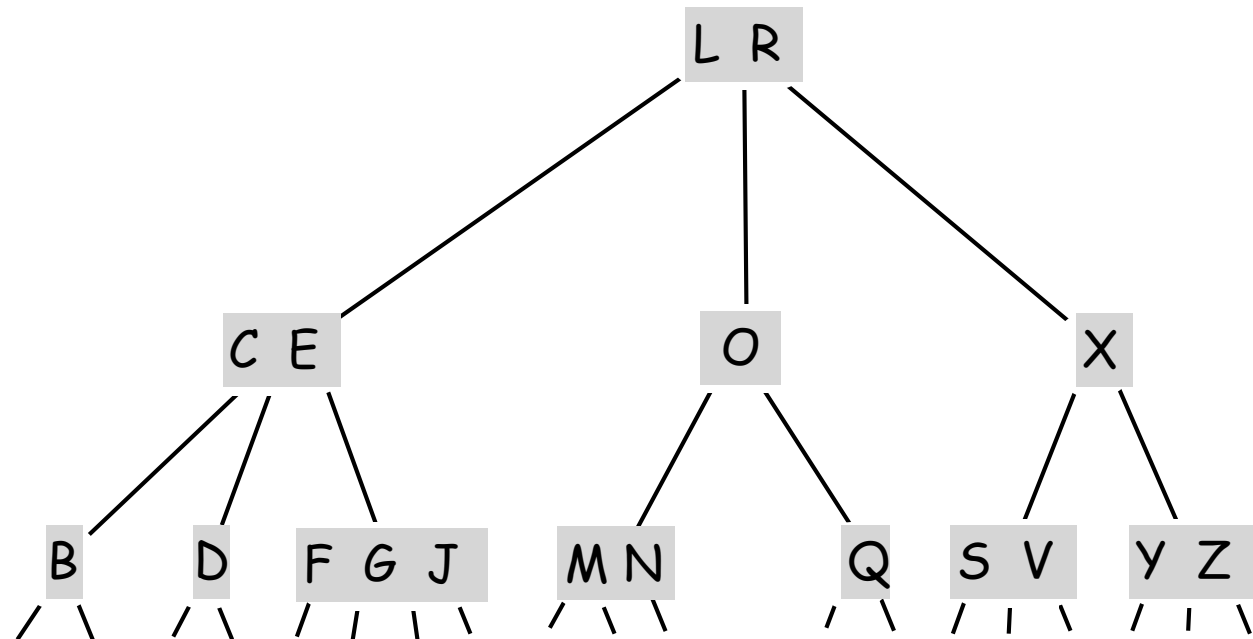
Deletions in 2-3-4 trees

Delete:

- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key
- Key not in leaf: replace with successor (always leaf in subtree) and delete successor from leaf.

Ex. Delete K

- Find successor
- Delete L from leaf
- Replace K with L



Dynamic set implementations

Worst case running times

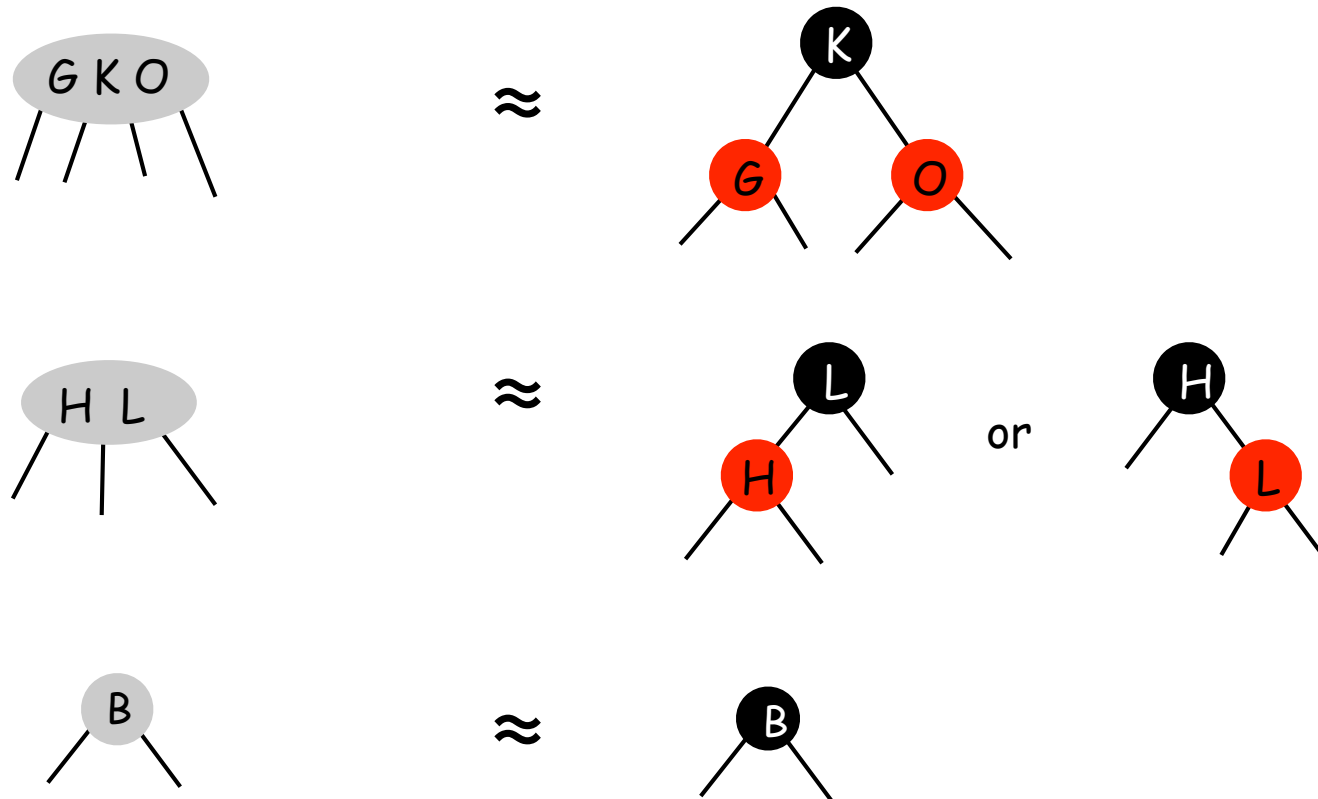
Implementation	search	insert	delete	minimum	maximum	successor	predecessor
linked lists	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
ordered array	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(\log n)$	$O(\log n)$
BST	$O(h)$	$O(h)$	$O(h)$	$O(h)$	$O(h)$	$O(h)$	$O(h)$
2-3-4 tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

Red-black trees

Red-black tree (Guibas-Sedgwick, 1979)

Represent 2-3-4 tree as a binary search tree

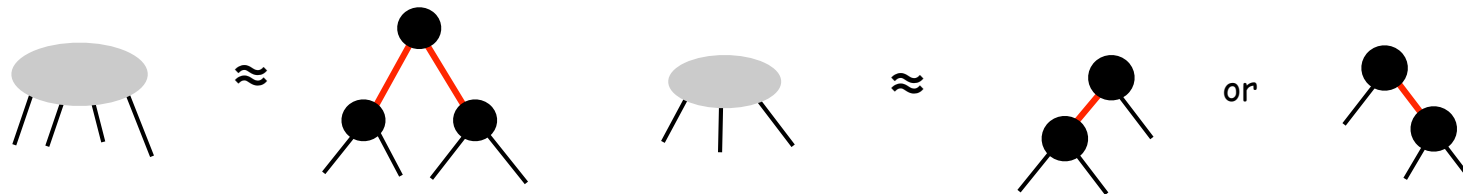
- Use colors on nodes to represent 3- and 4-nodes.



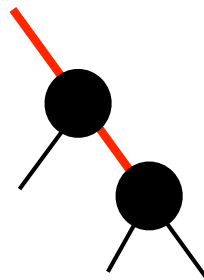
Red-black tree (Guibas-Sedgwick, 1979)

Represent 2-3-4 tree as a binary search tree

- Use colors on nodes to represent 3- and 4-nodes.



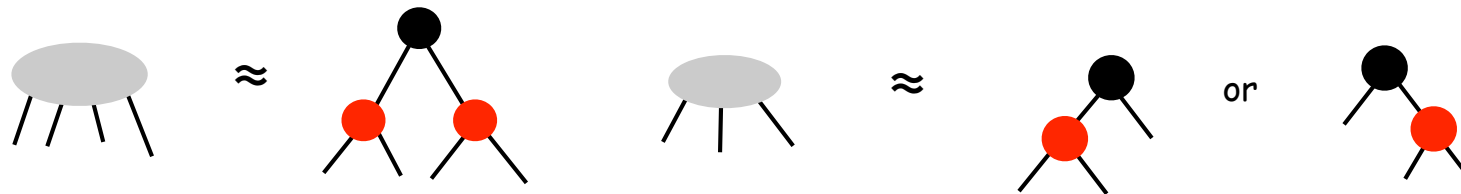
- Disallowed: 2 red nodes in-a-row.



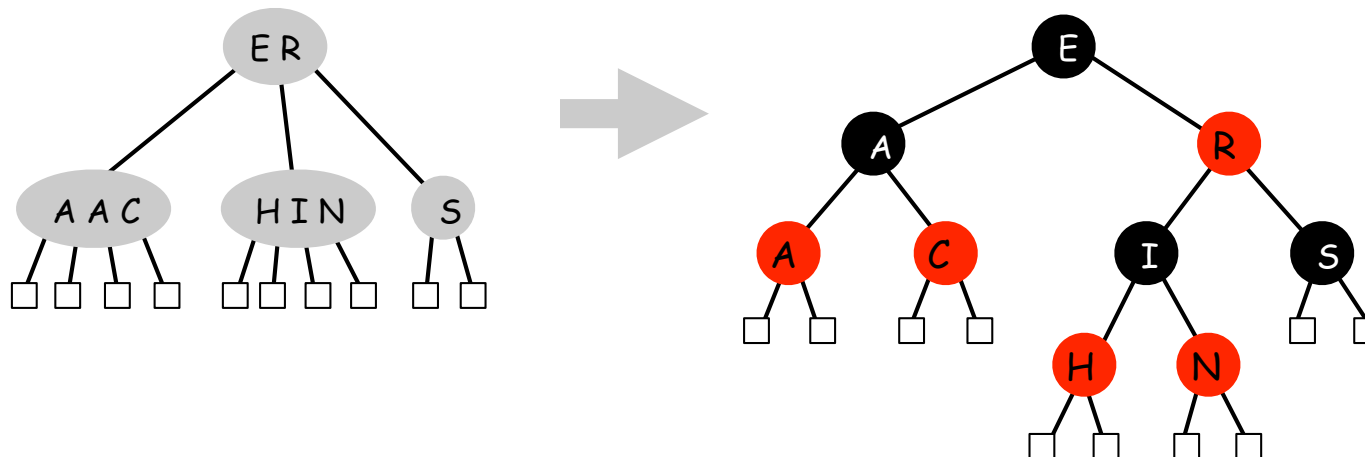
Red-black tree (Guibas-Sedgwick, 1979)

Represent 2-3-4 tree as a binary search tree

- Use colors on nodes to represent 3- and 4-nodes.



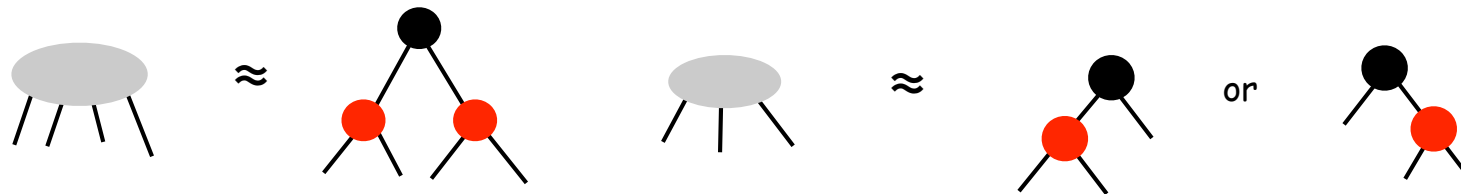
- Connection between 2-3-4 trees and red-black trees:



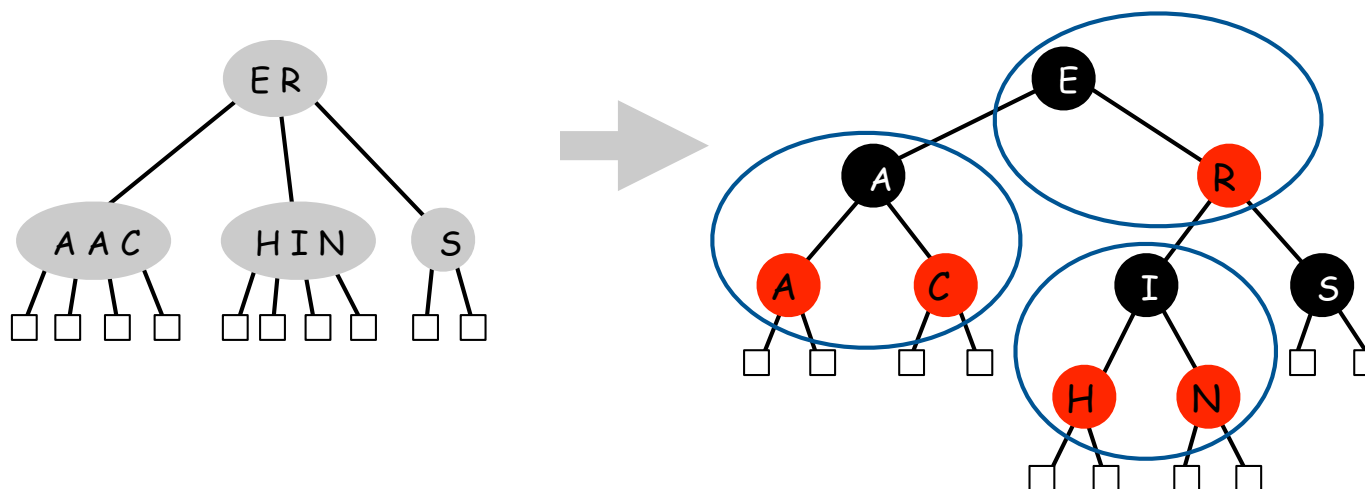
Red-black tree (Guibas-Sedgwick, 1979)

Represent 2-3-4 tree as a binary search tree

- Use colors on nodes to represent 3- and 4-nodes.



- Connection between 2-3-4 trees and red-black trees:



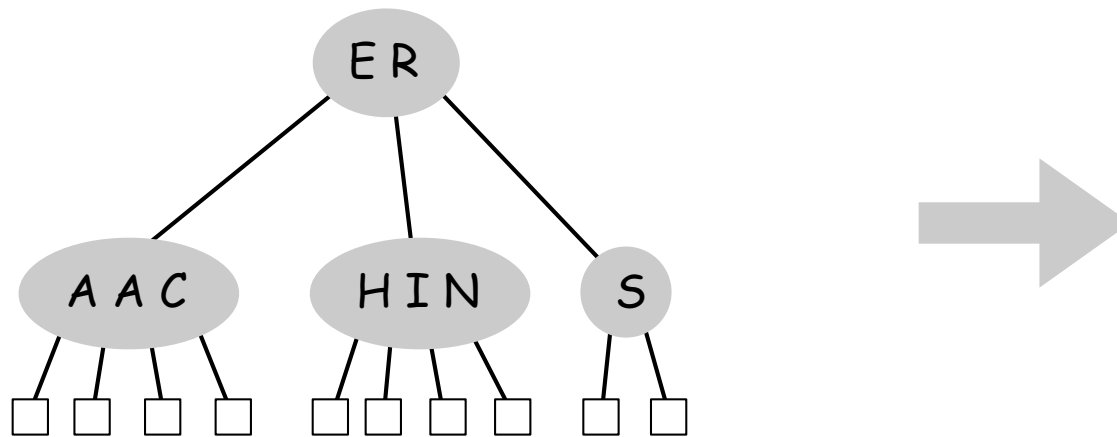
Red-black tree

Properties of red-black trees:

- The root is always black
- All root-to-leaf paths have the same number of black nodes.
- Red nodes do not have red children
- CLRS: All leaves (NIL) are black

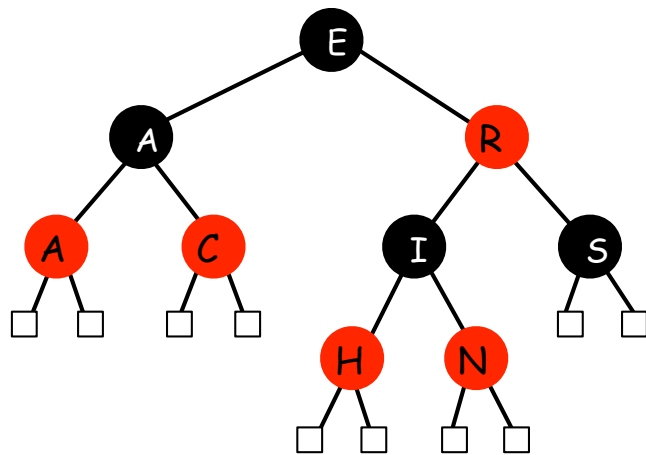
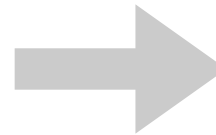
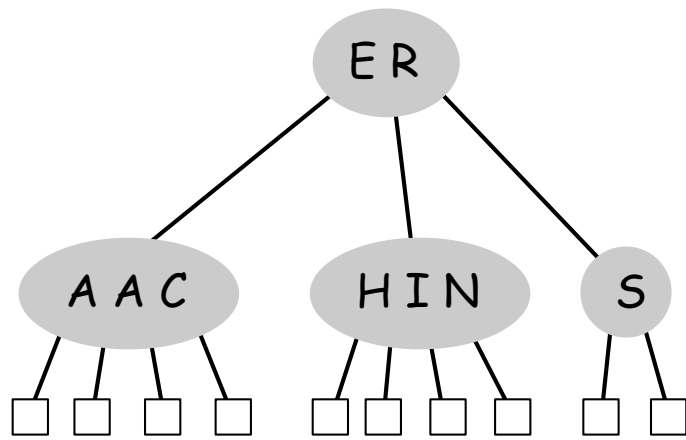
Red-black tree

Connection between 2-3-4 trees and red-black trees:



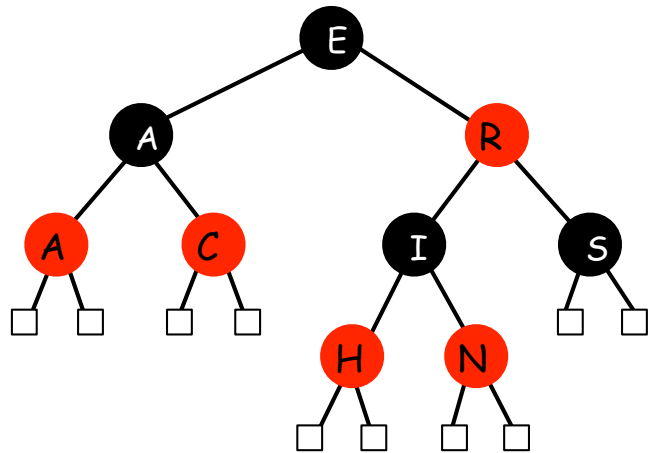
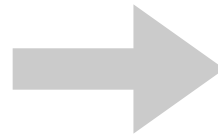
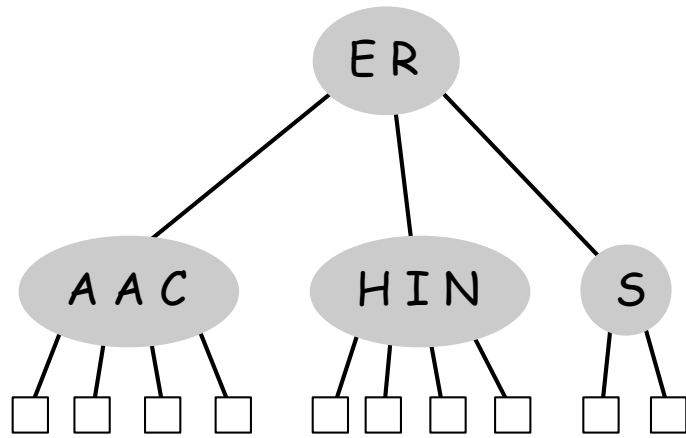
Red-black tree

Connection between 2-3-4 trees and red-black trees:

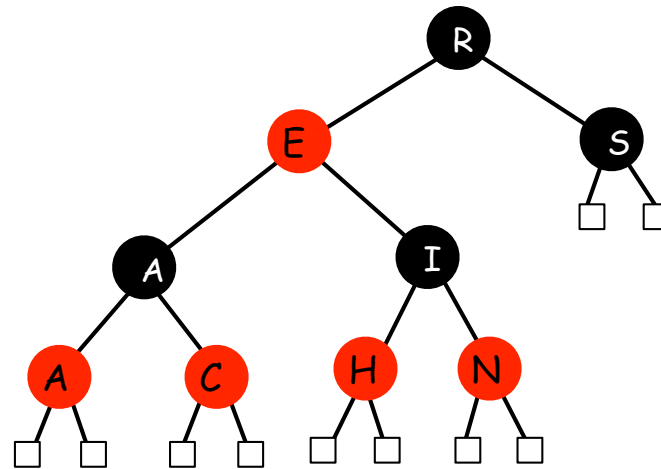


Red-black tree

Connection between 2-3-4 trees and red-black trees:

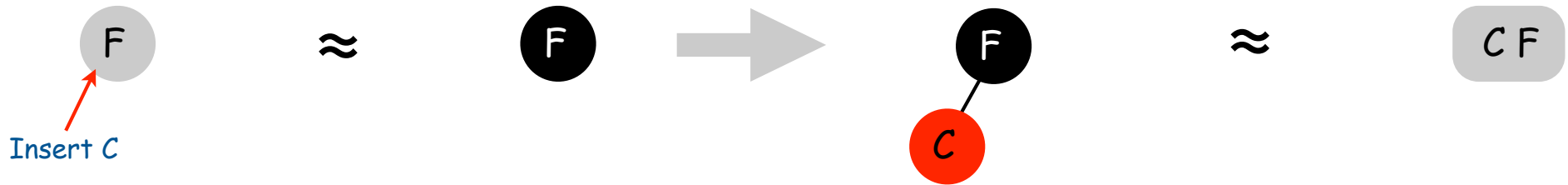


or

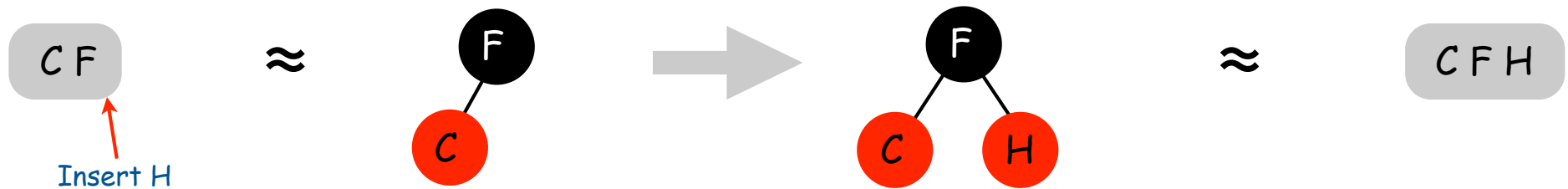


Insertion in red-black trees

Insertion in 2-node:

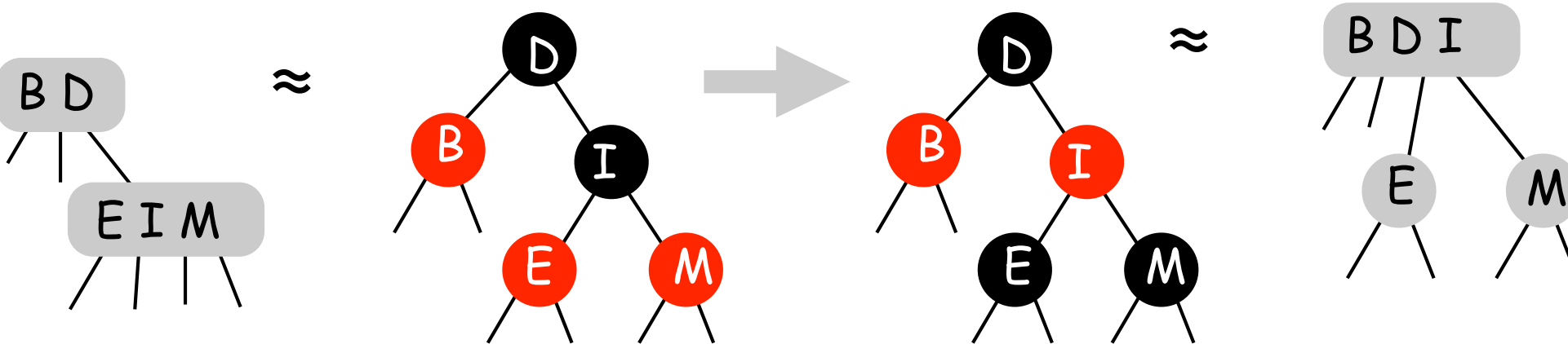
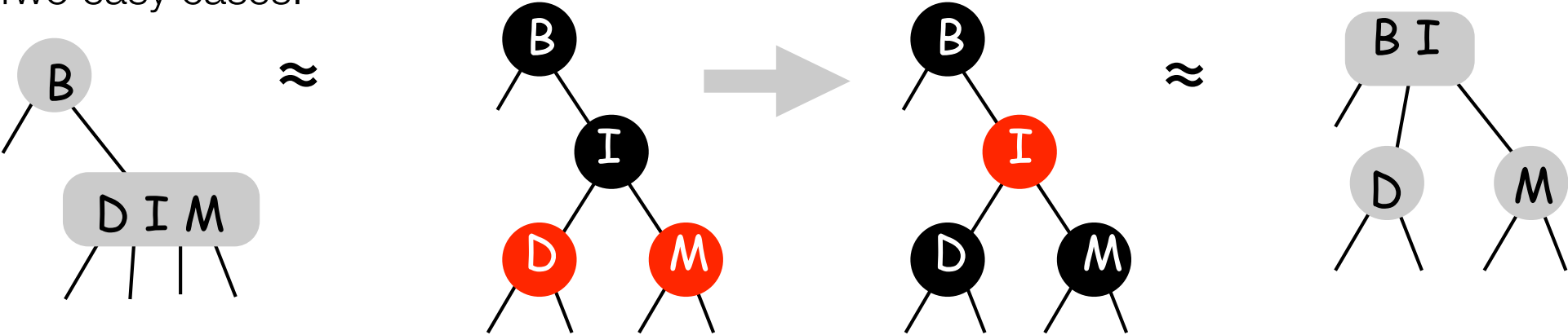


Insertion in 3-node:



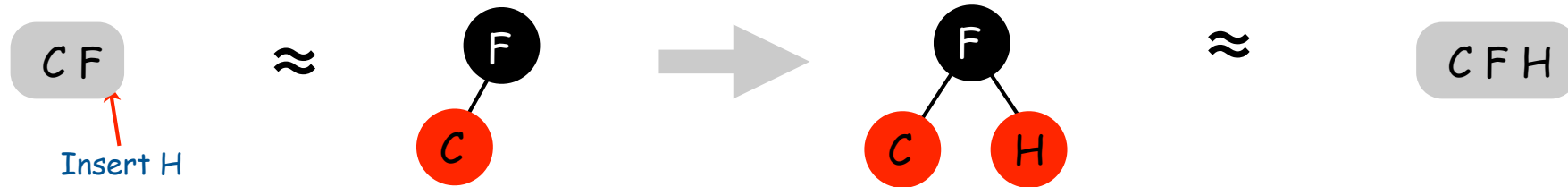
Red-black tree: Splitting 4-nodes

Two easy cases:



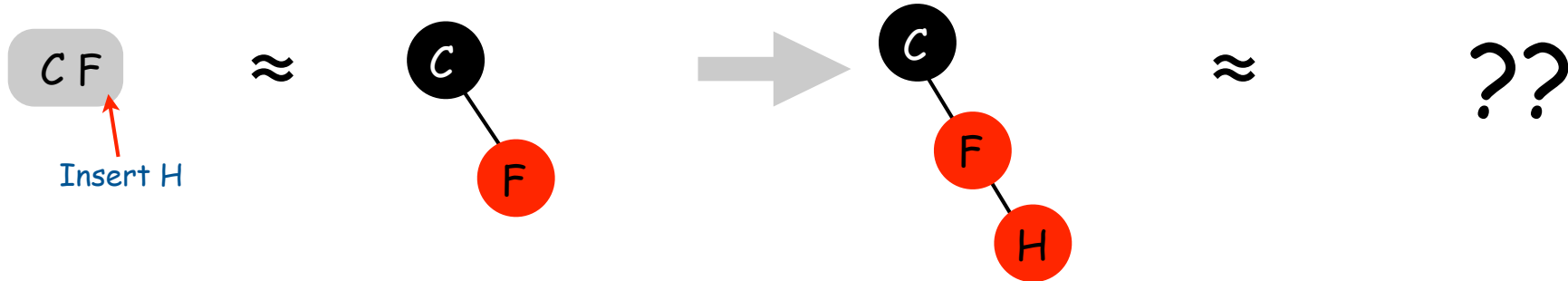
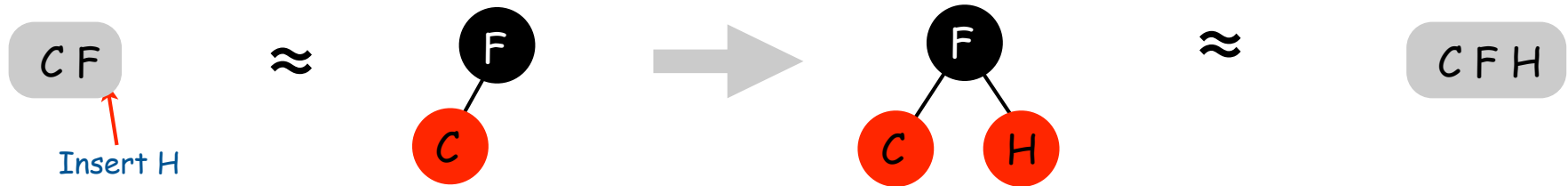
Insertion in red-black trees

Insertion in 3-node (continued):



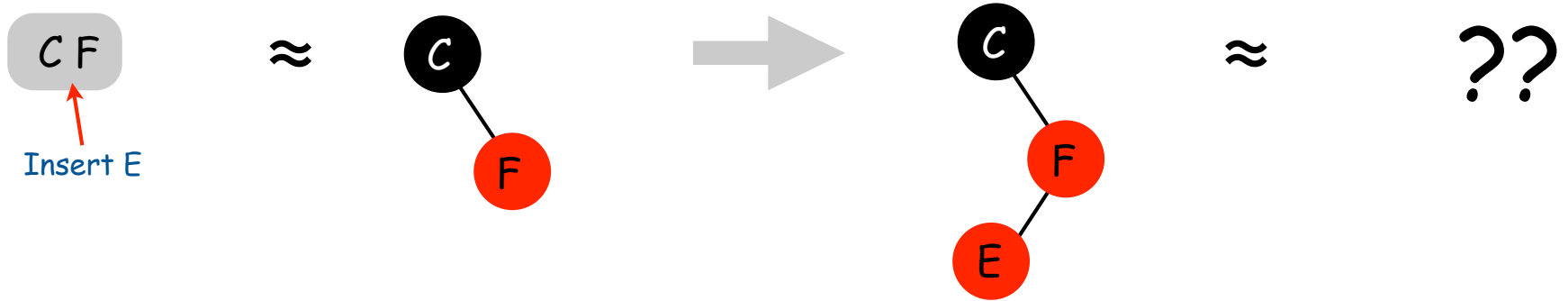
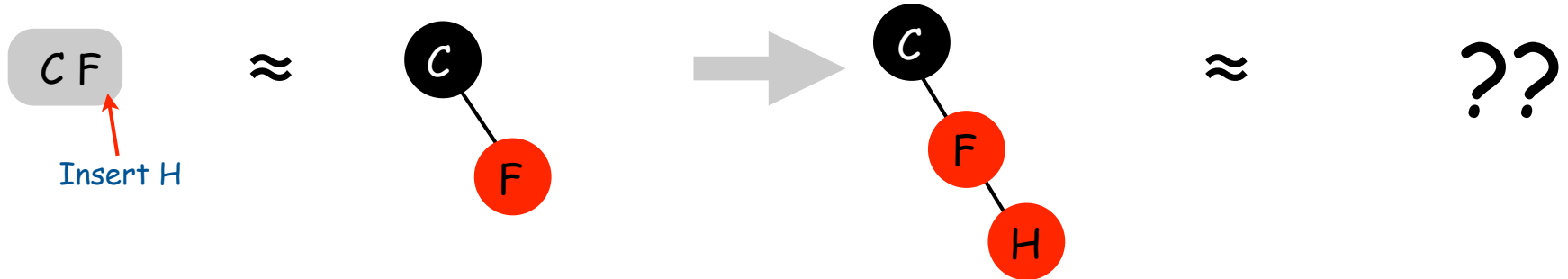
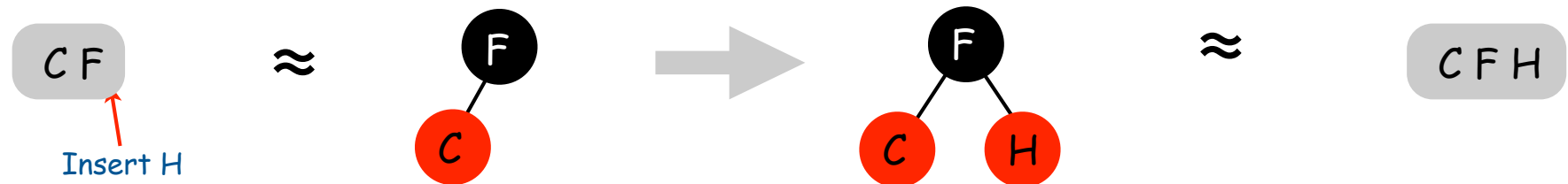
Insertion in red-black trees

Insertion in 3-node (continued):



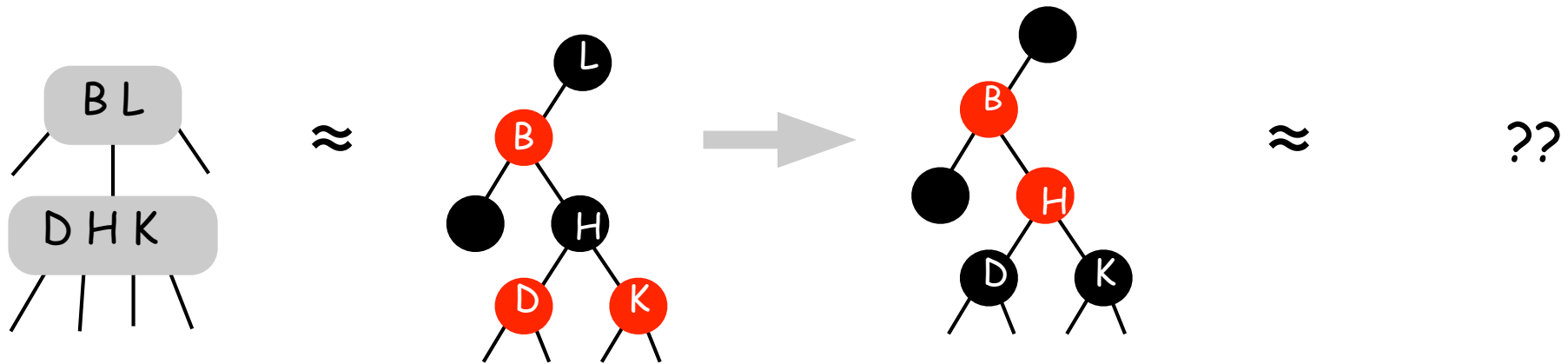
Insertion in red-black trees

Insertion in 3-node (continued):



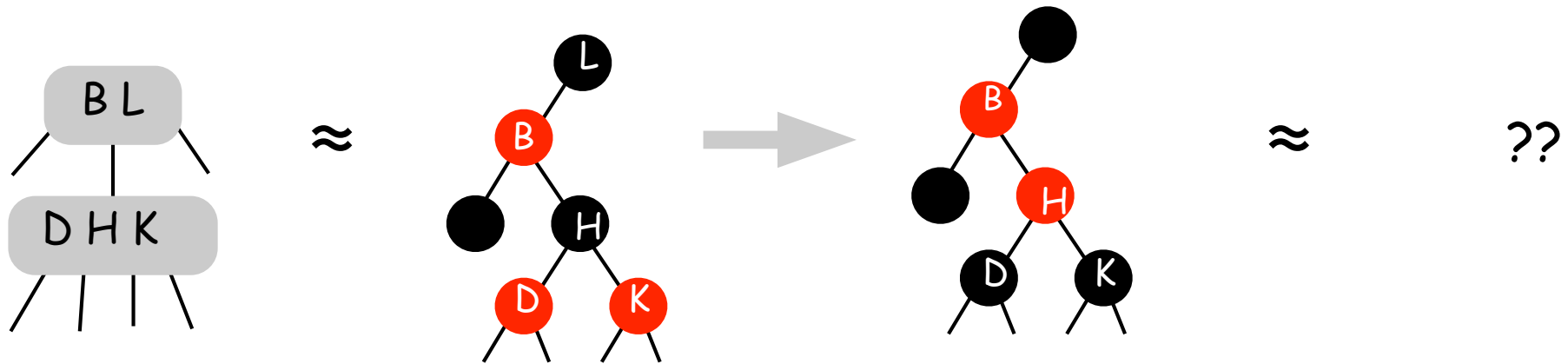
Red-black trees: Splitting of 4-nodes

Example of hard case:



Red-black trees: Splitting of 4-nodes

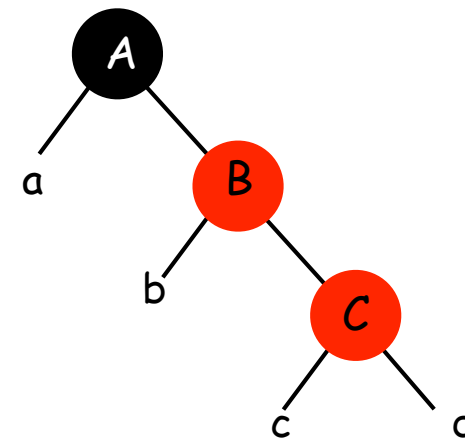
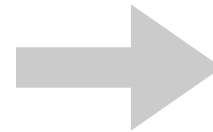
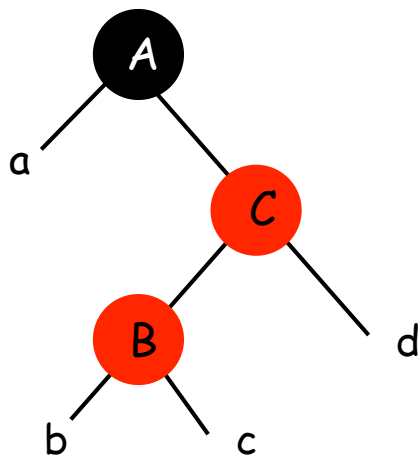
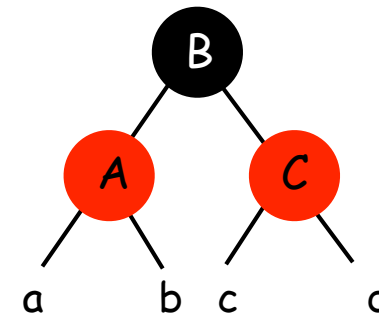
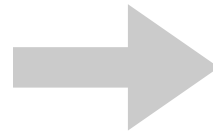
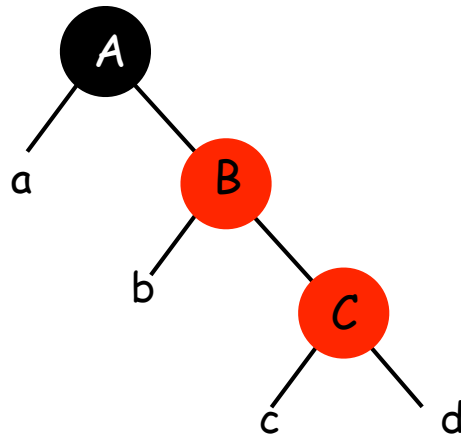
Example of hard case:



Solution: Rotations!

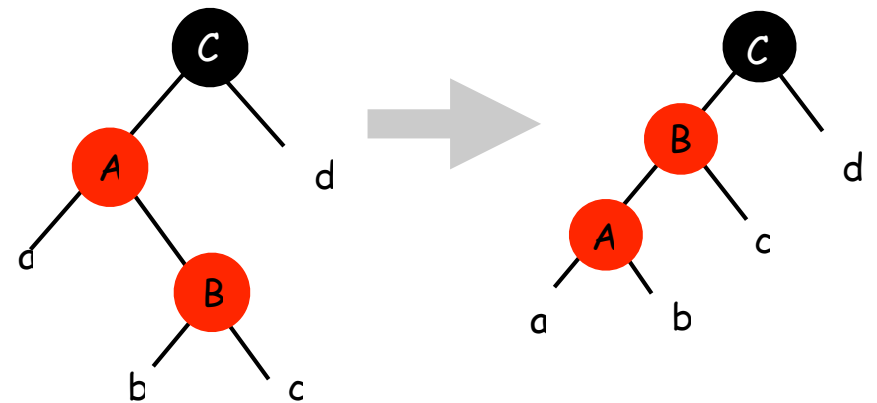
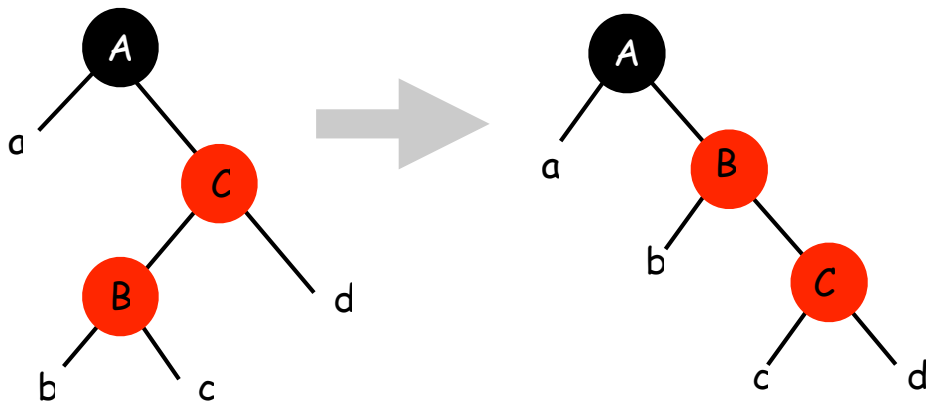
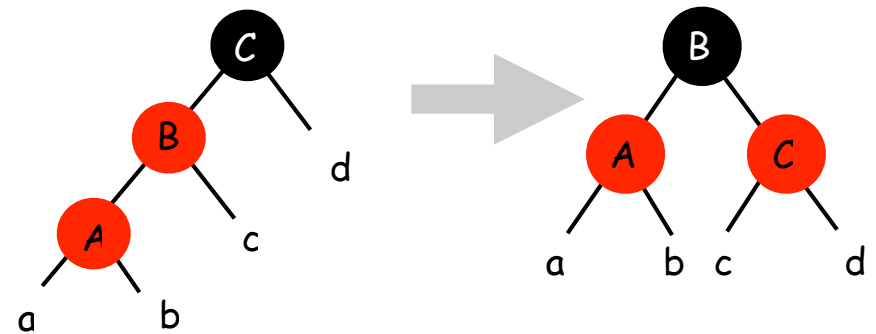
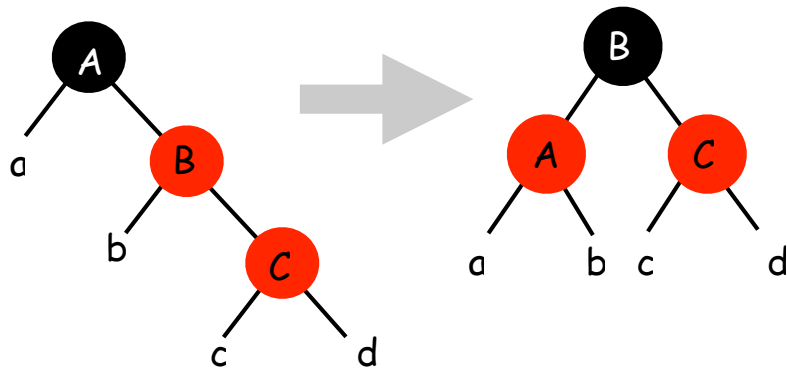
Rotations in red-black trees

Two types of rotations



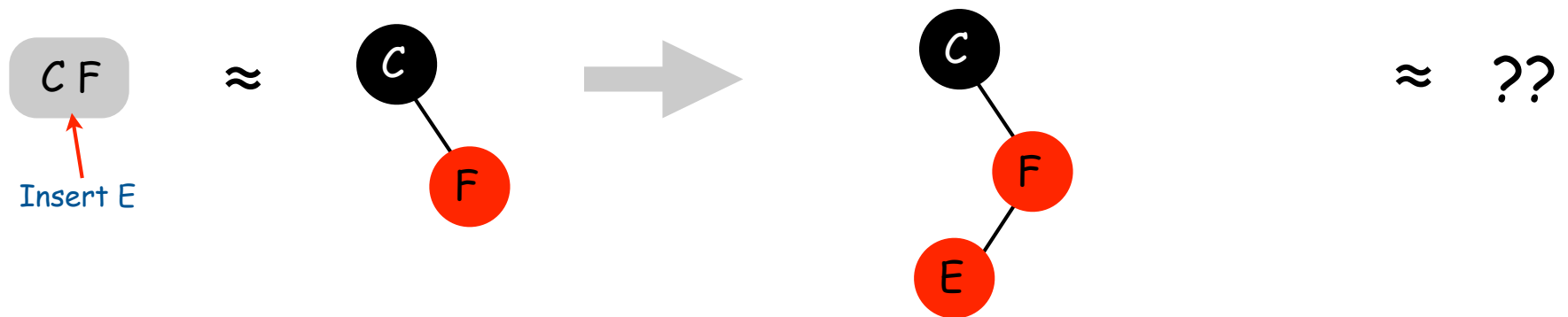
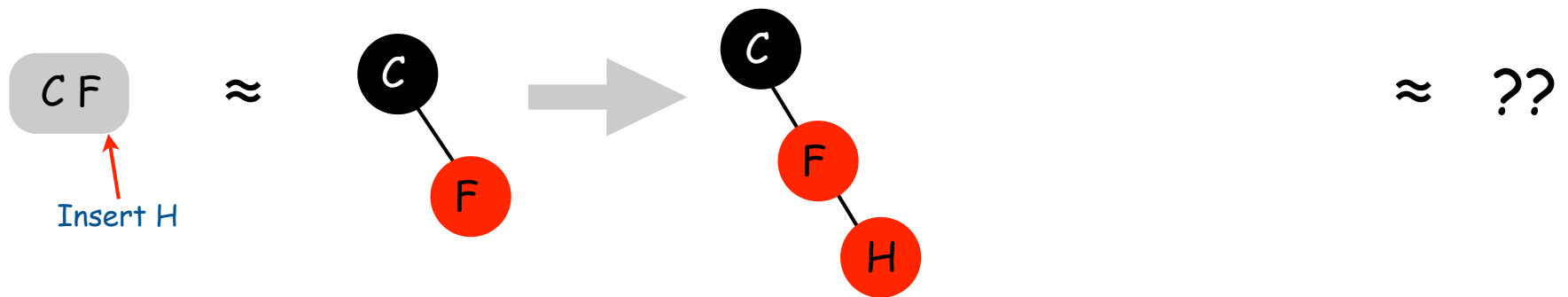
Rotations in red-black trees

Two types of rotations:



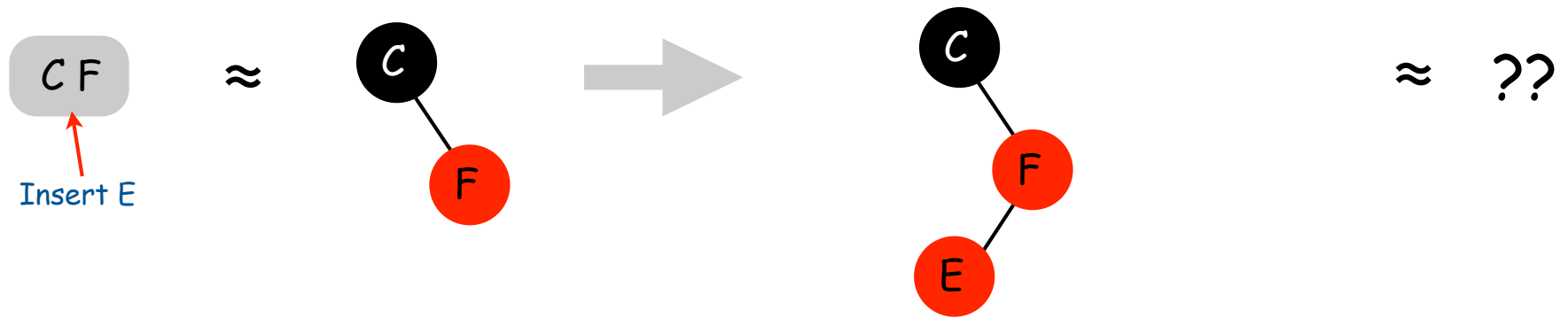
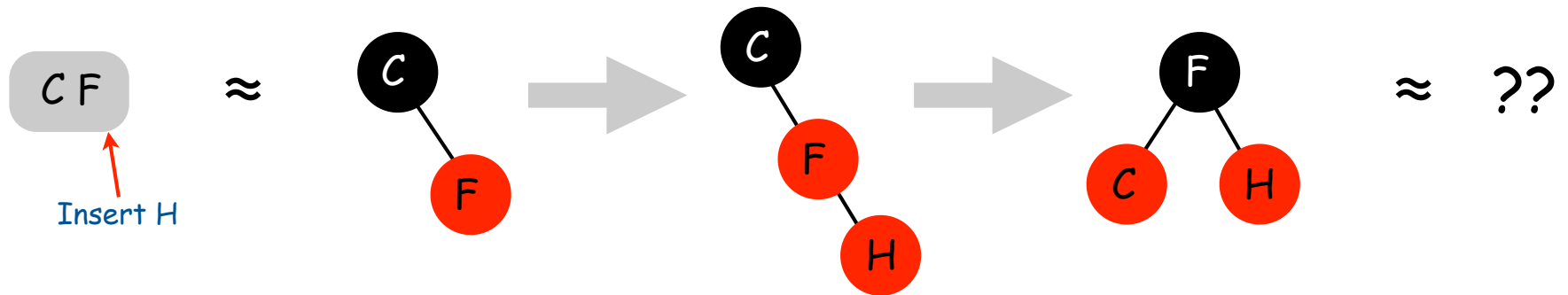
Insertion in red-black trees

Insertion in 3-node:



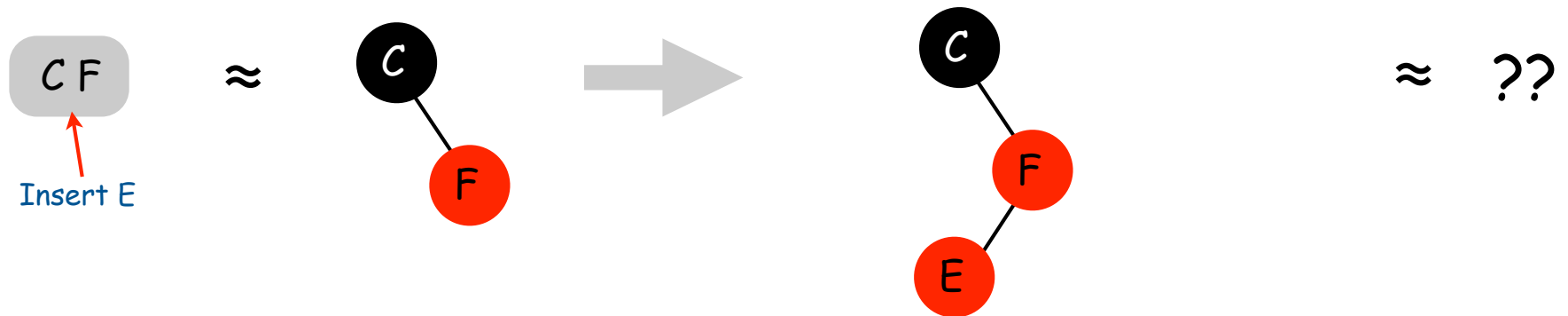
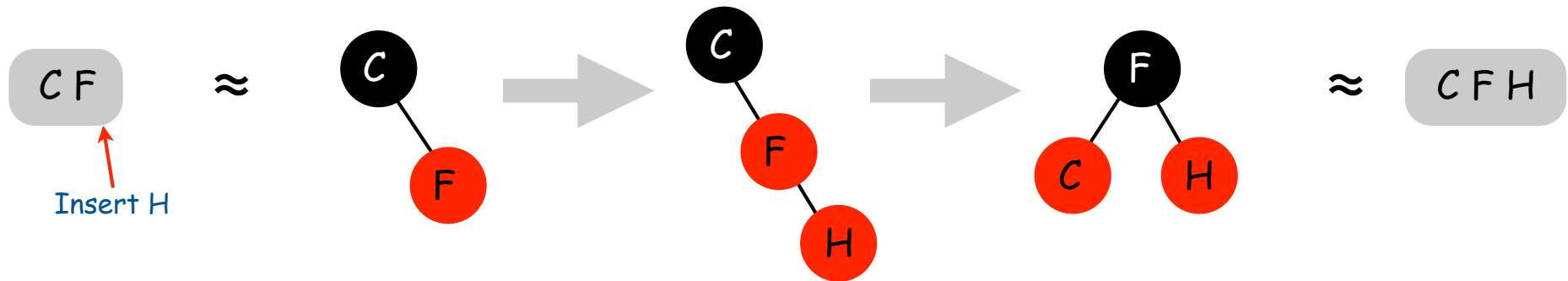
Insertion in red-black trees

Insertion in 3-node:



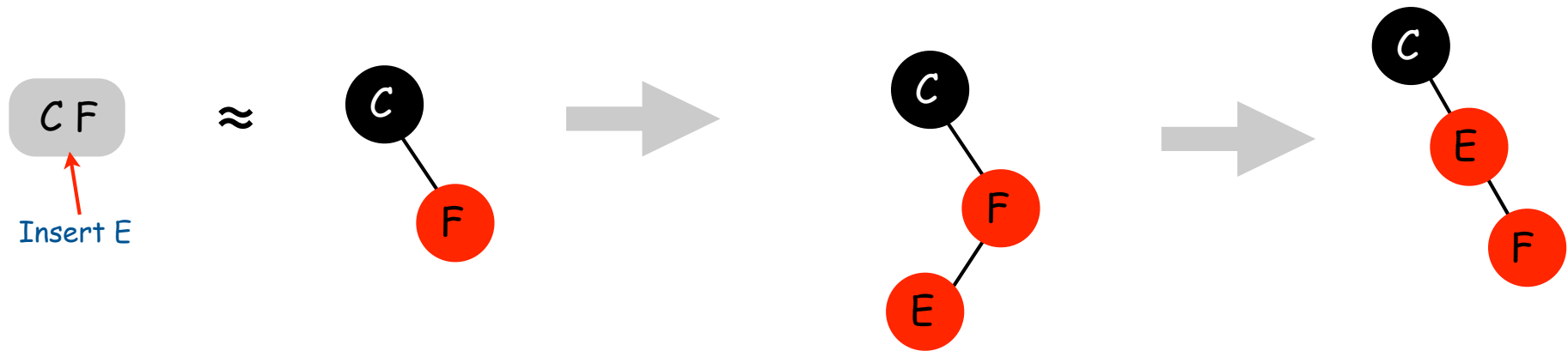
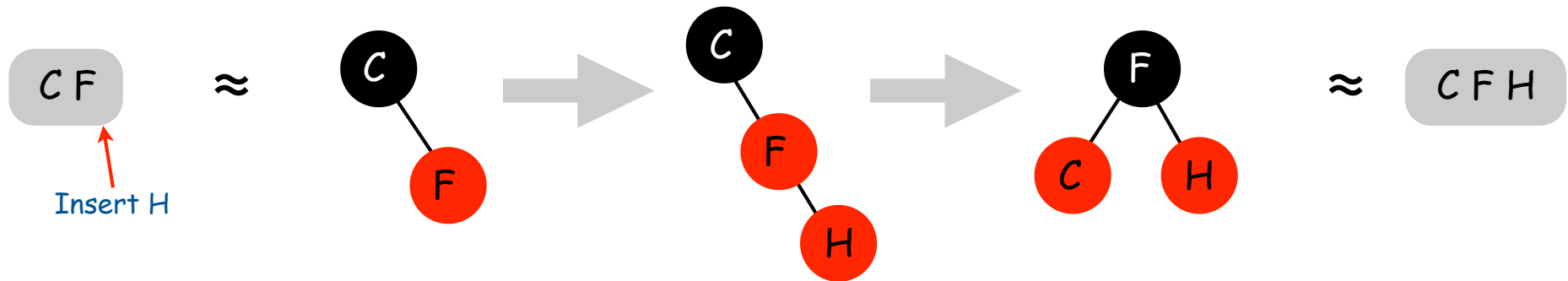
Insertion in red-black trees

Insertion in 3-node:



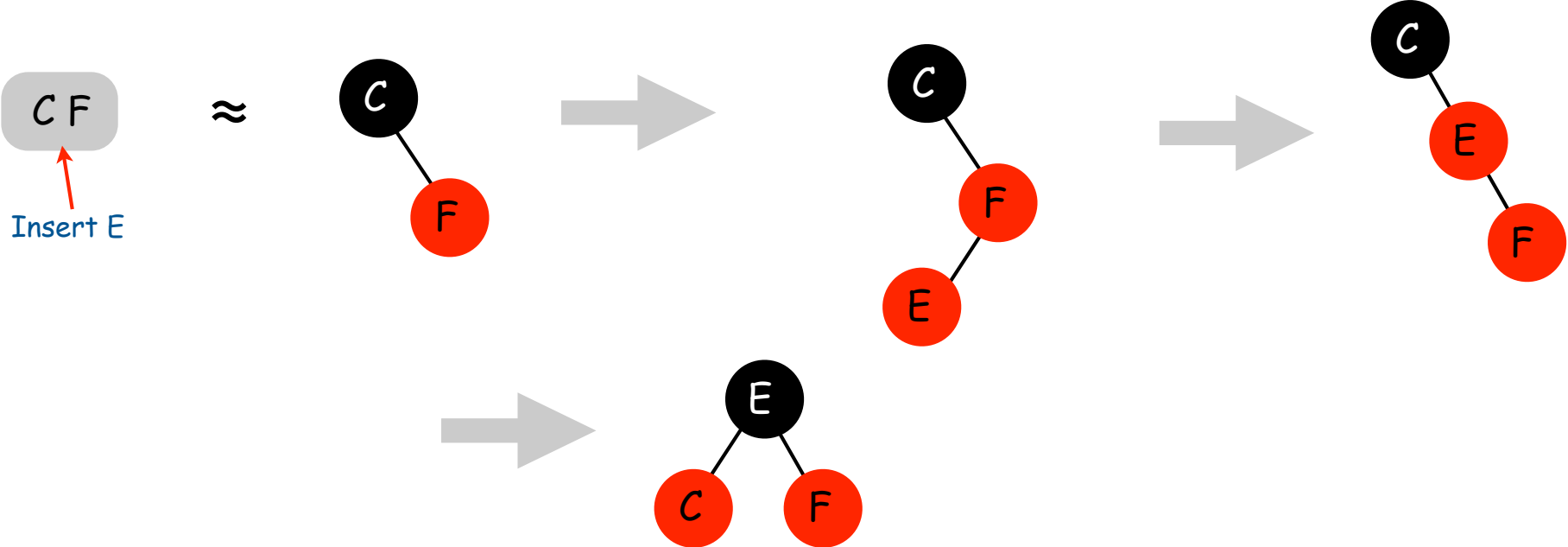
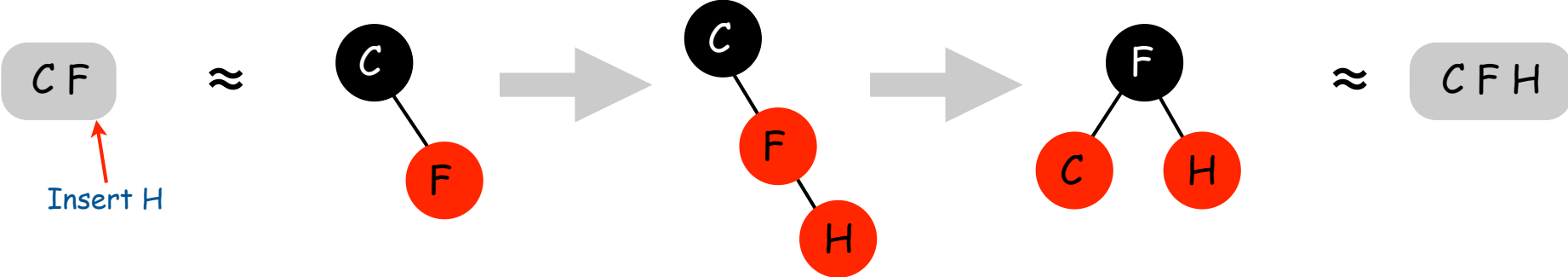
Insertion in red-black trees

Insertion in 3-node:



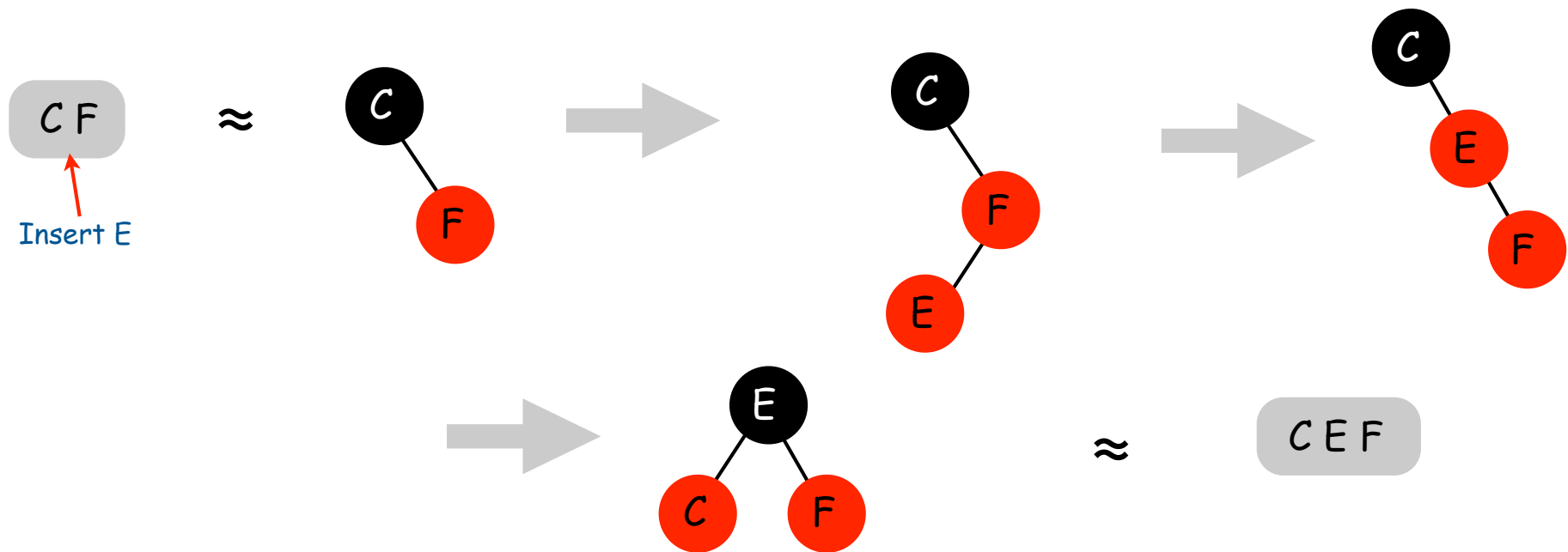
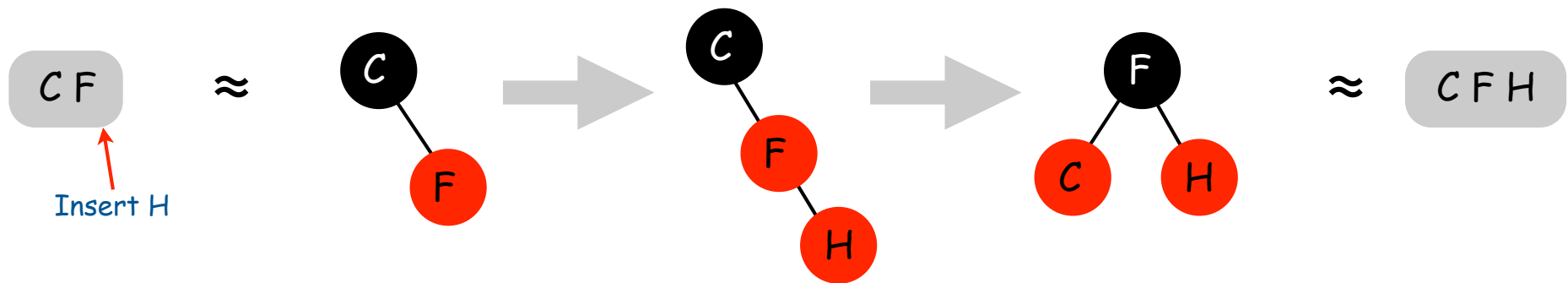
Insertion in red-black trees

Insertion in 3-node:



Insertion in red-black trees

Insertion in 3-node:



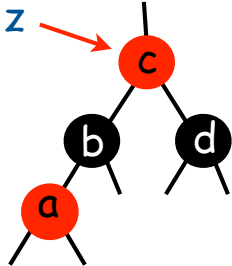
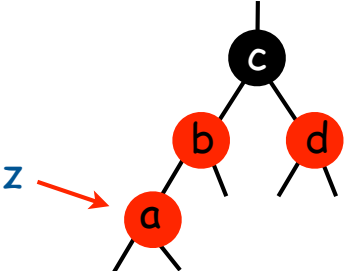
Insertion in red-black tree

Insert x:

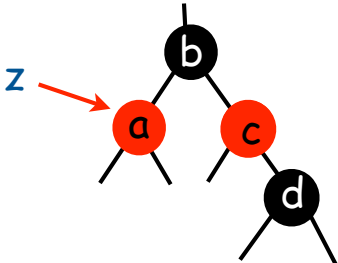
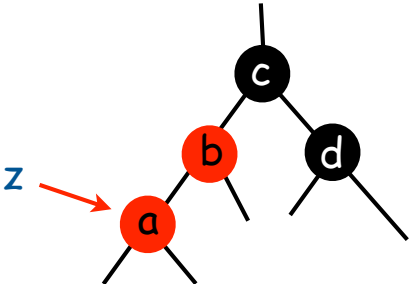
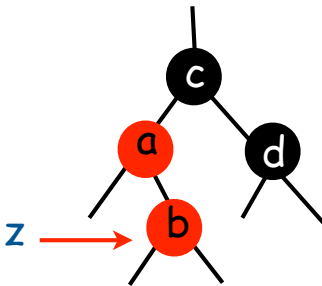
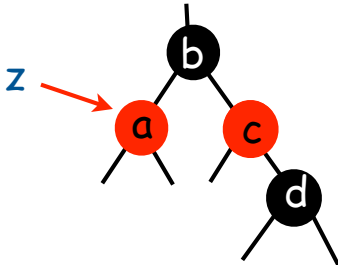
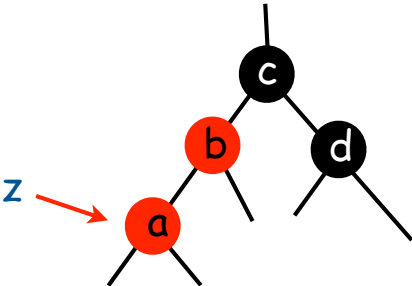
Search to bottom after key (x)

Insert red leaf

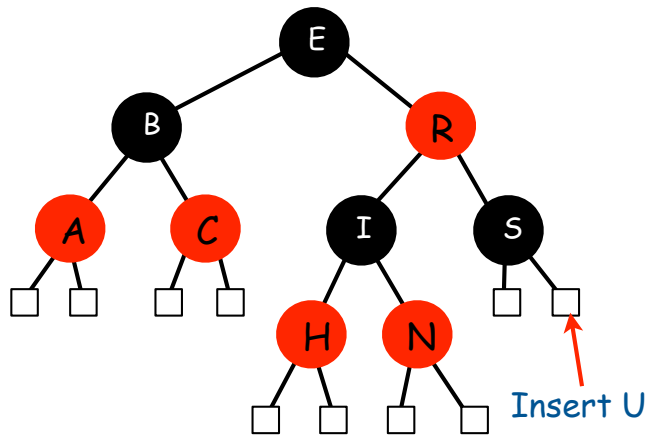
Balance: 3 cases (+ symmetric)



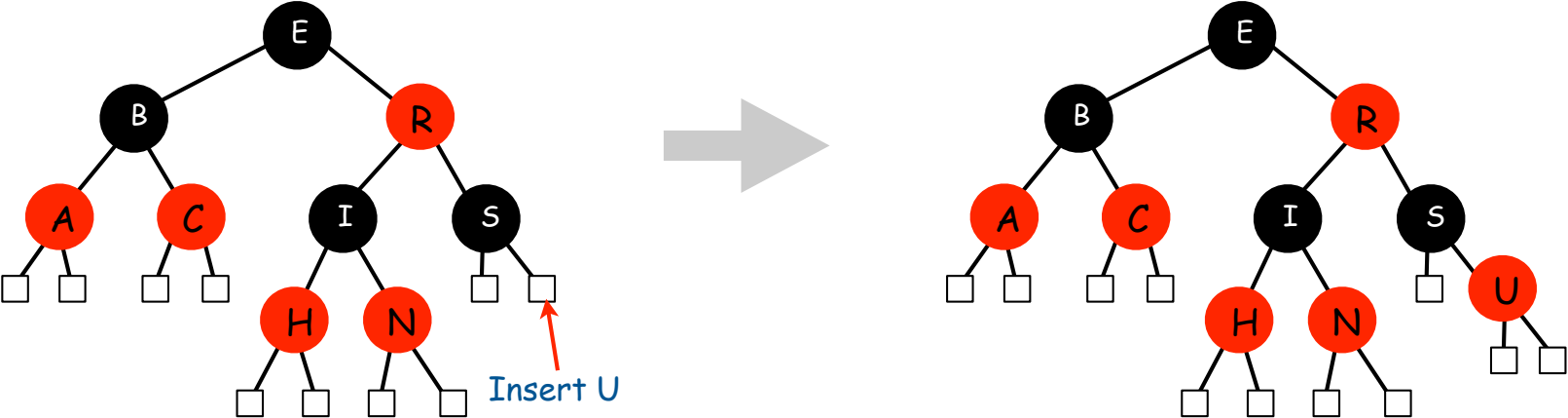
Keep
balancing
with z



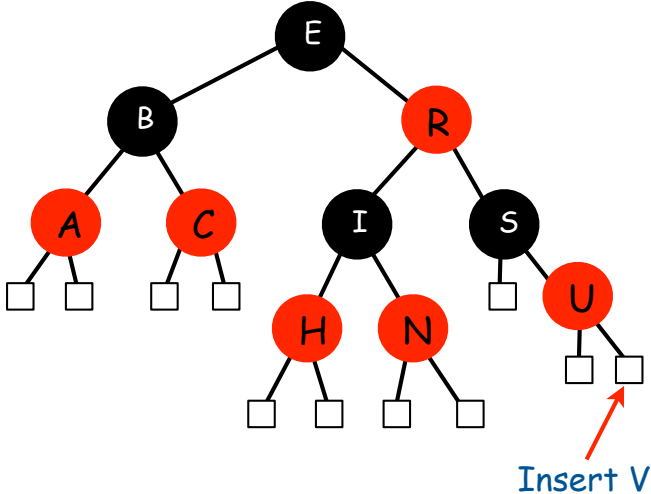
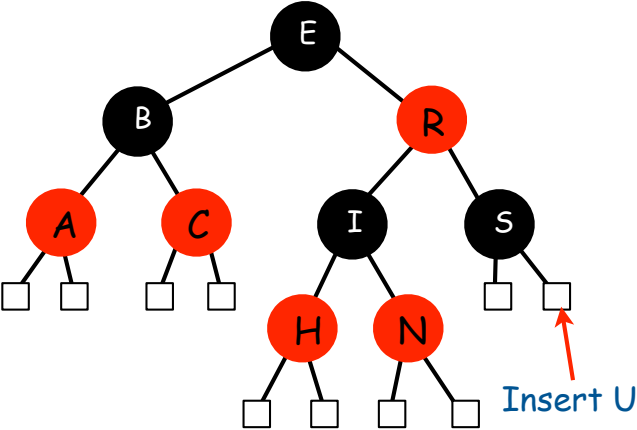
Eksempel



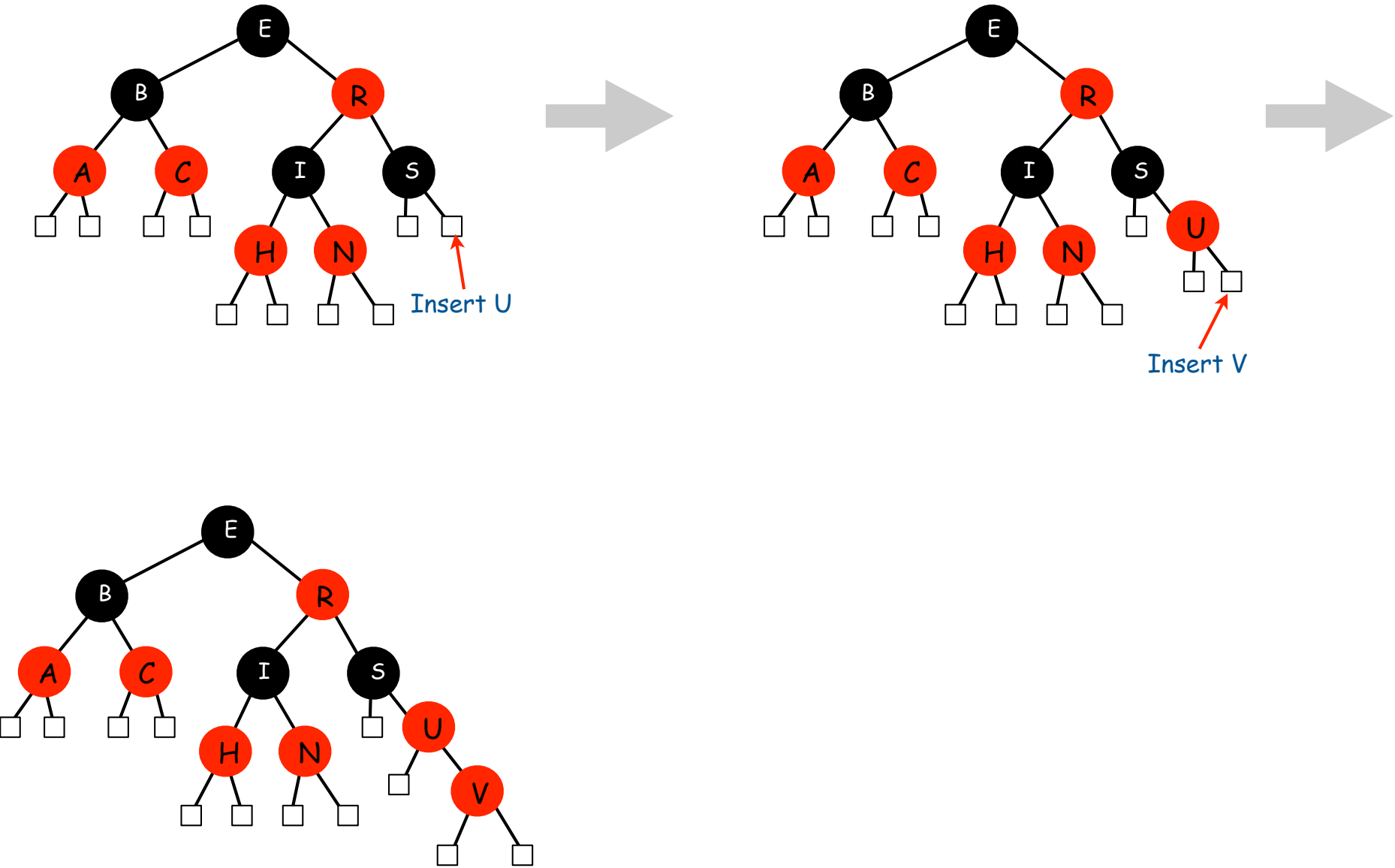
Eksempel



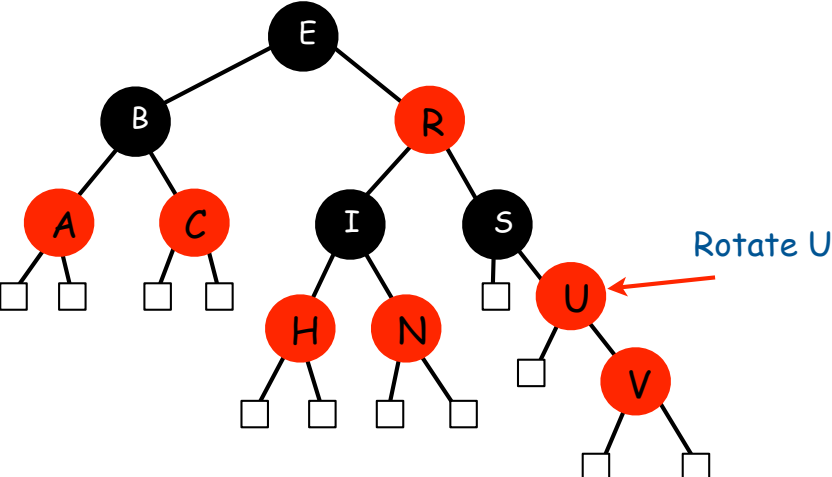
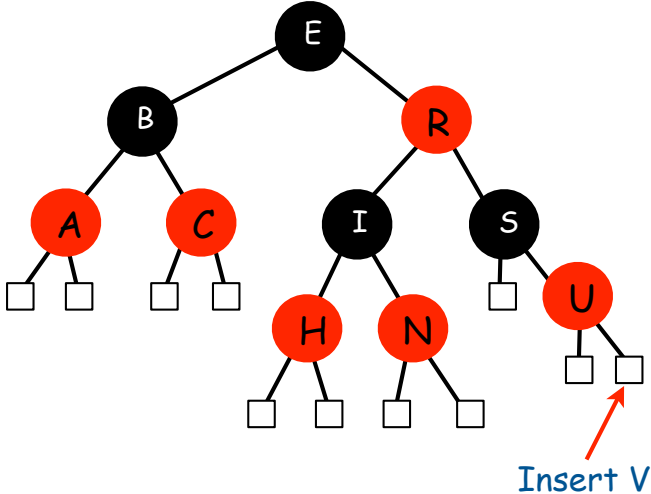
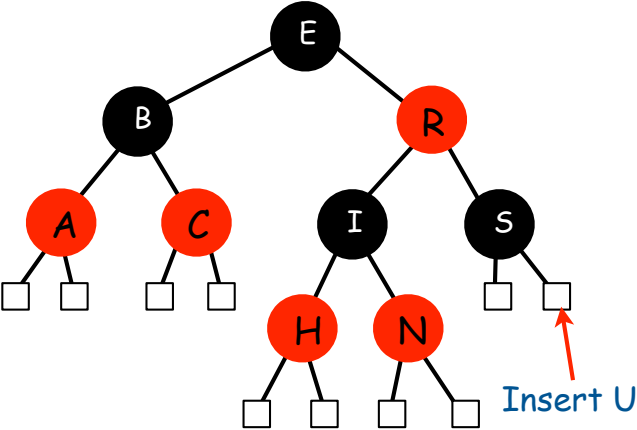
Eksempel



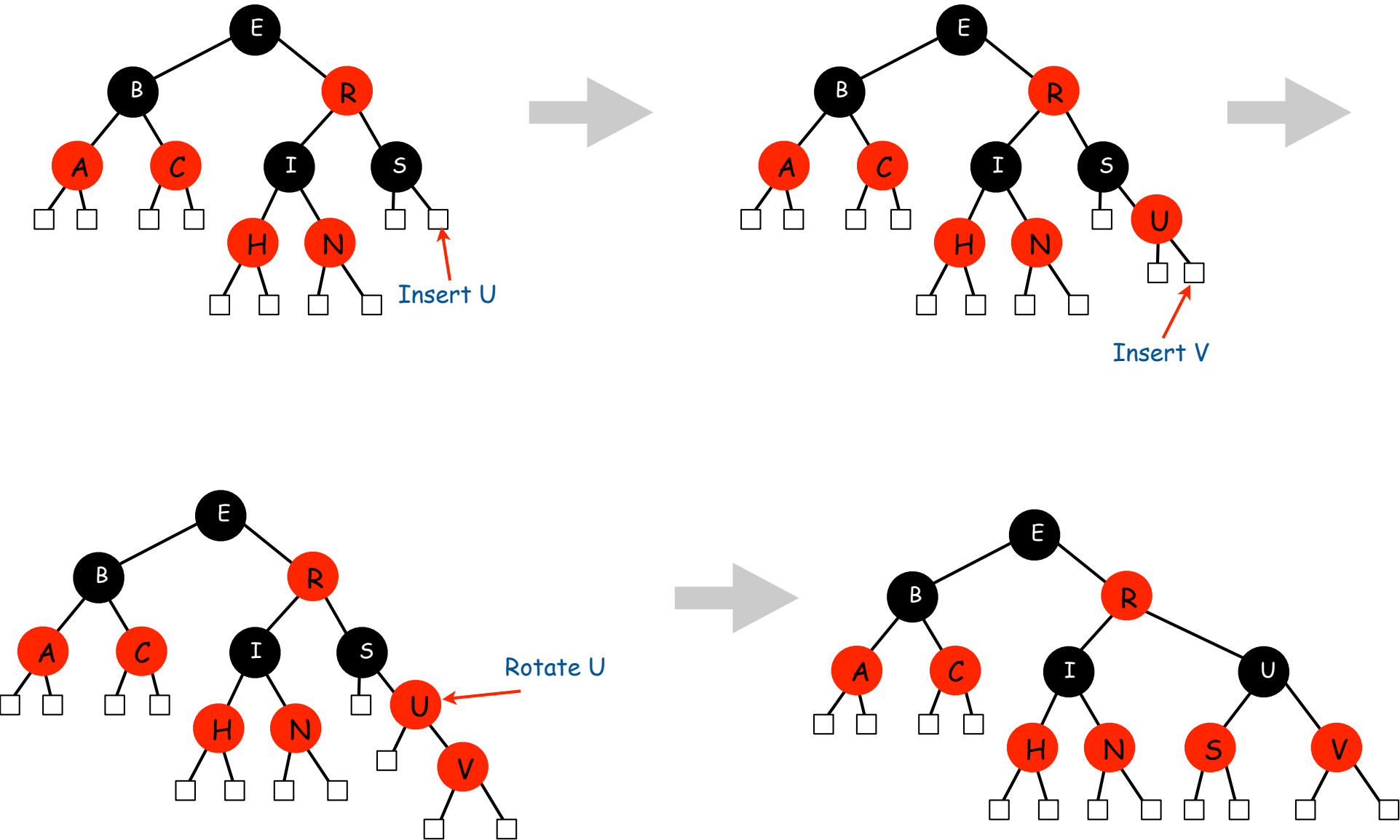
Eksempel



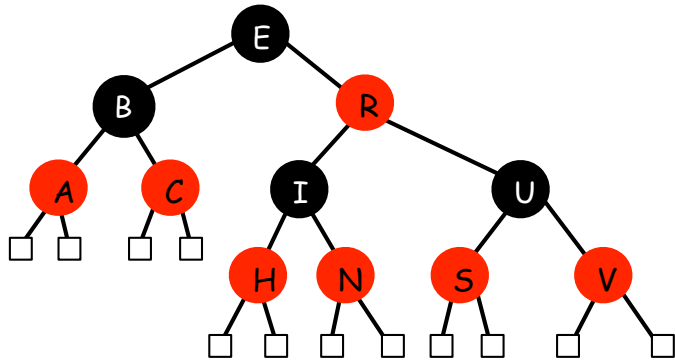
Eksempel



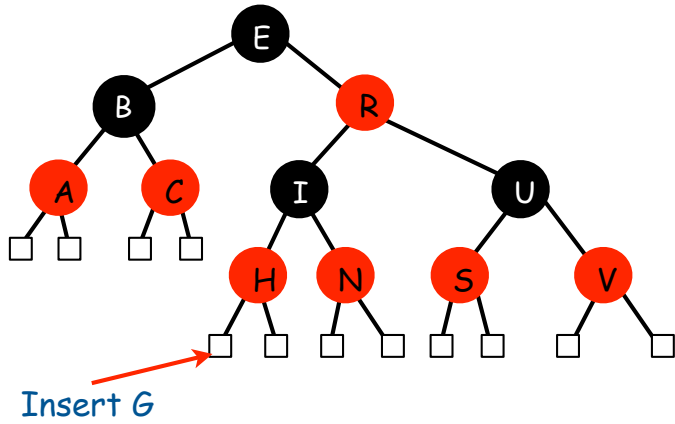
Eksempel



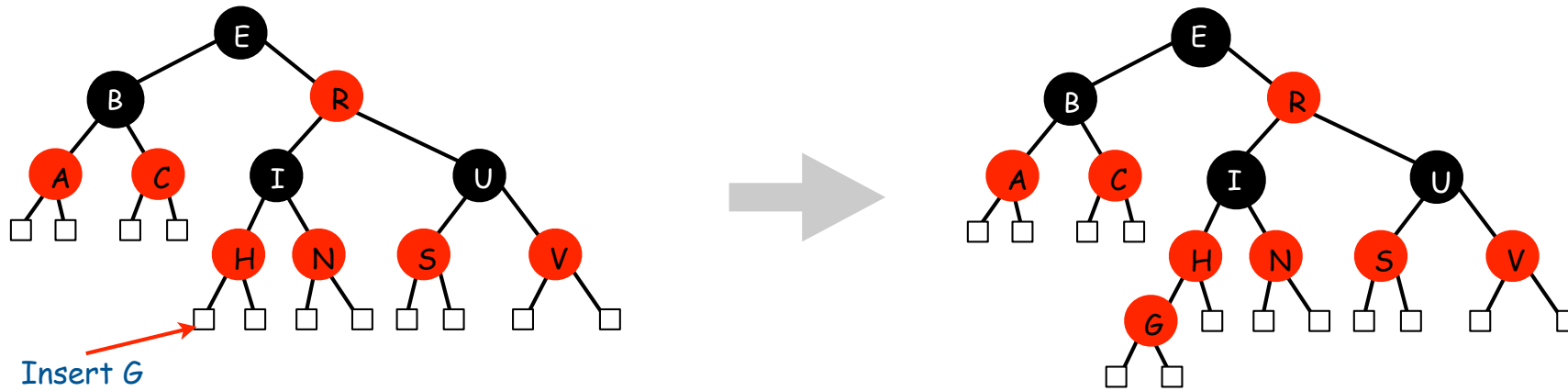
Example



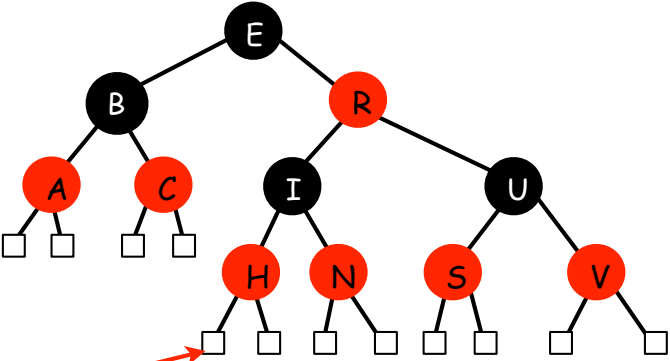
Example



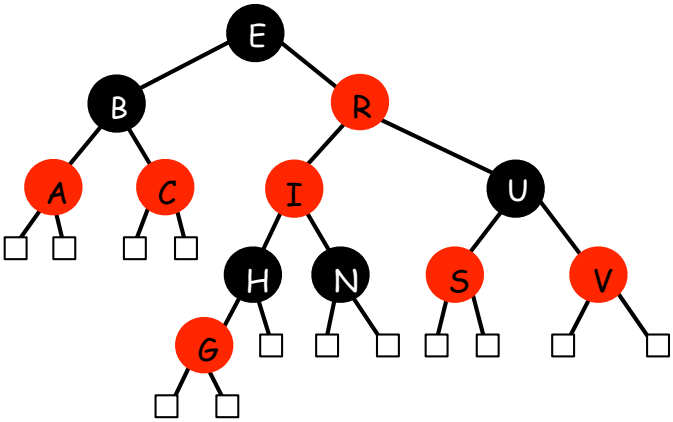
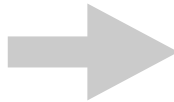
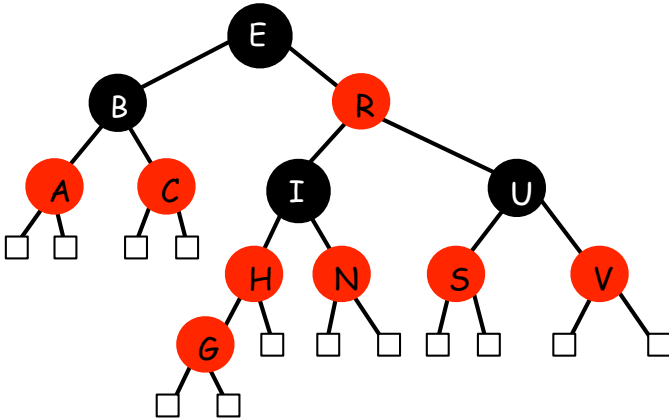
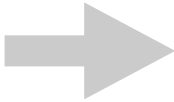
Example



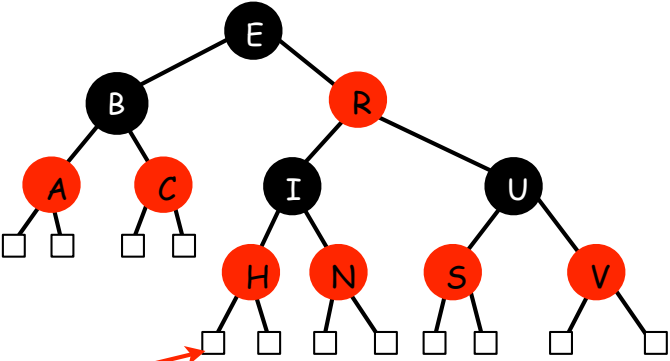
Example



Insert G

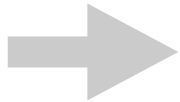
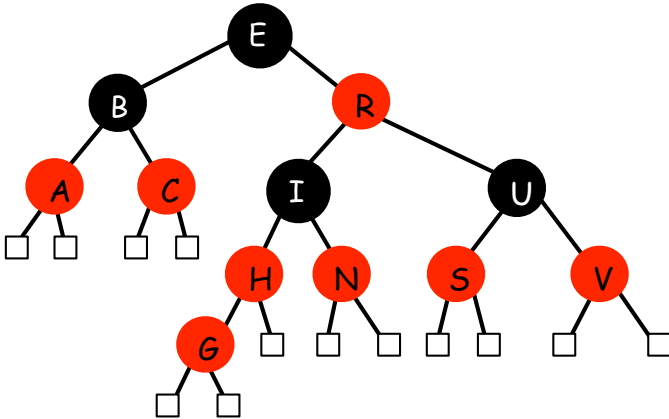
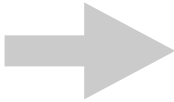
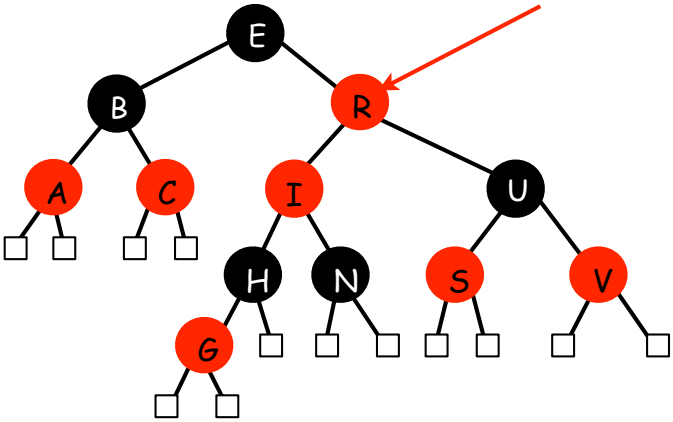


Example

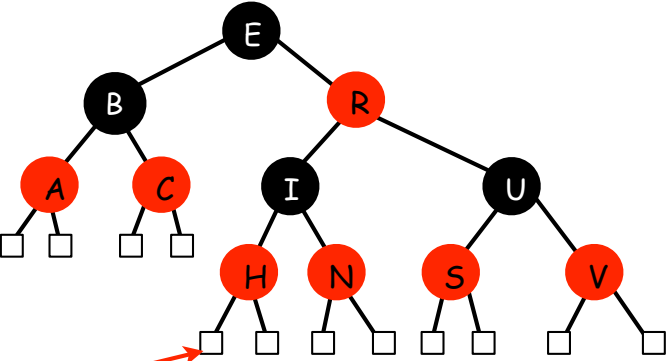


Insert G

Rotate I

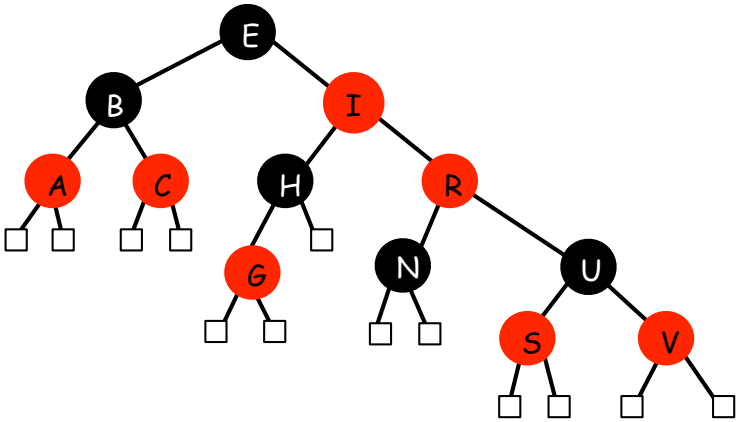
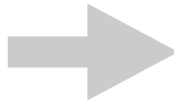
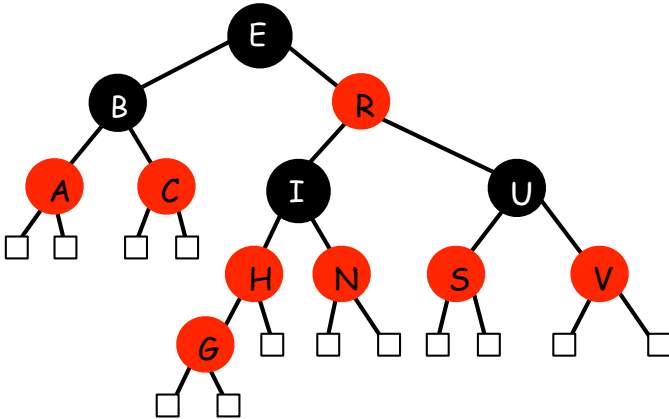
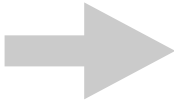
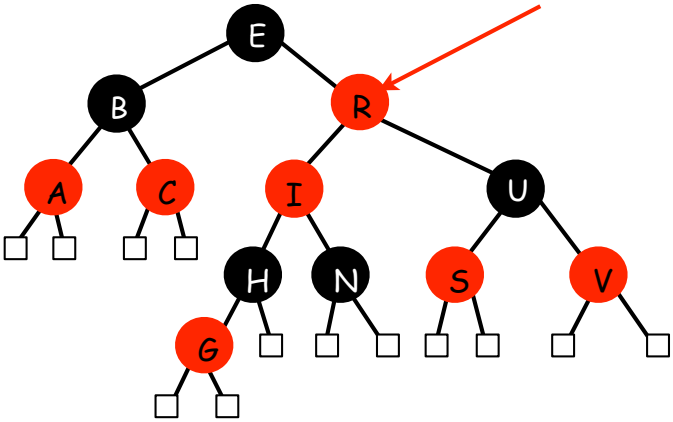


Example

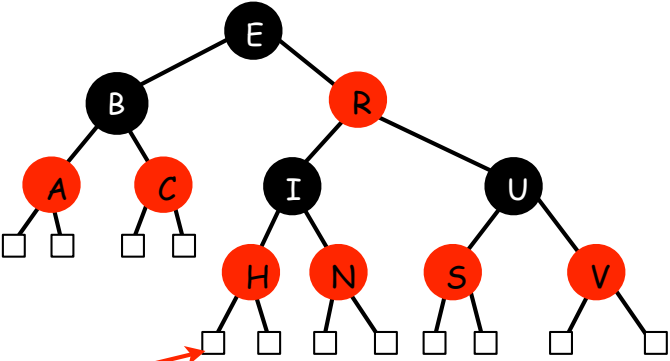


Insert G

Rotate I

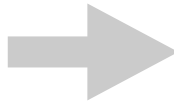
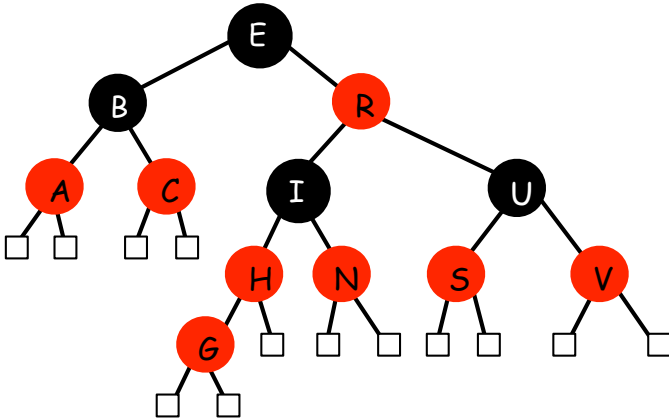
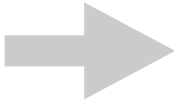
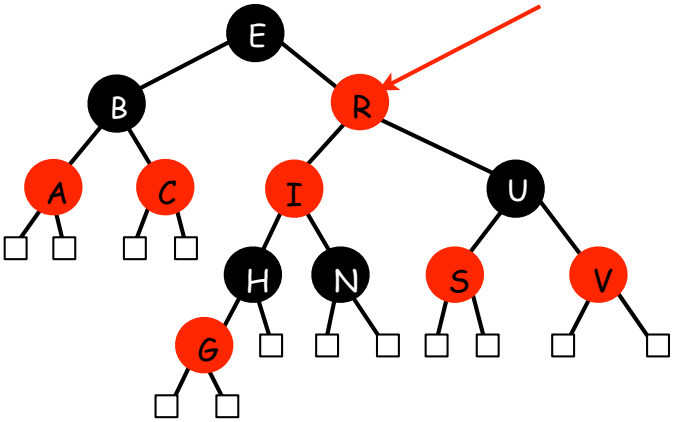


Example

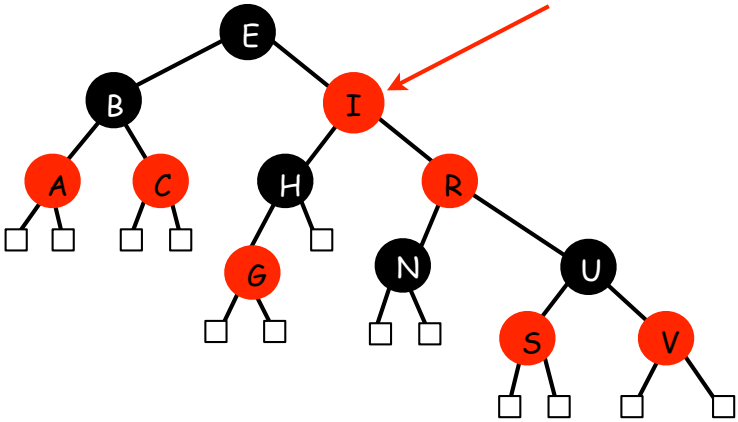


Insert G

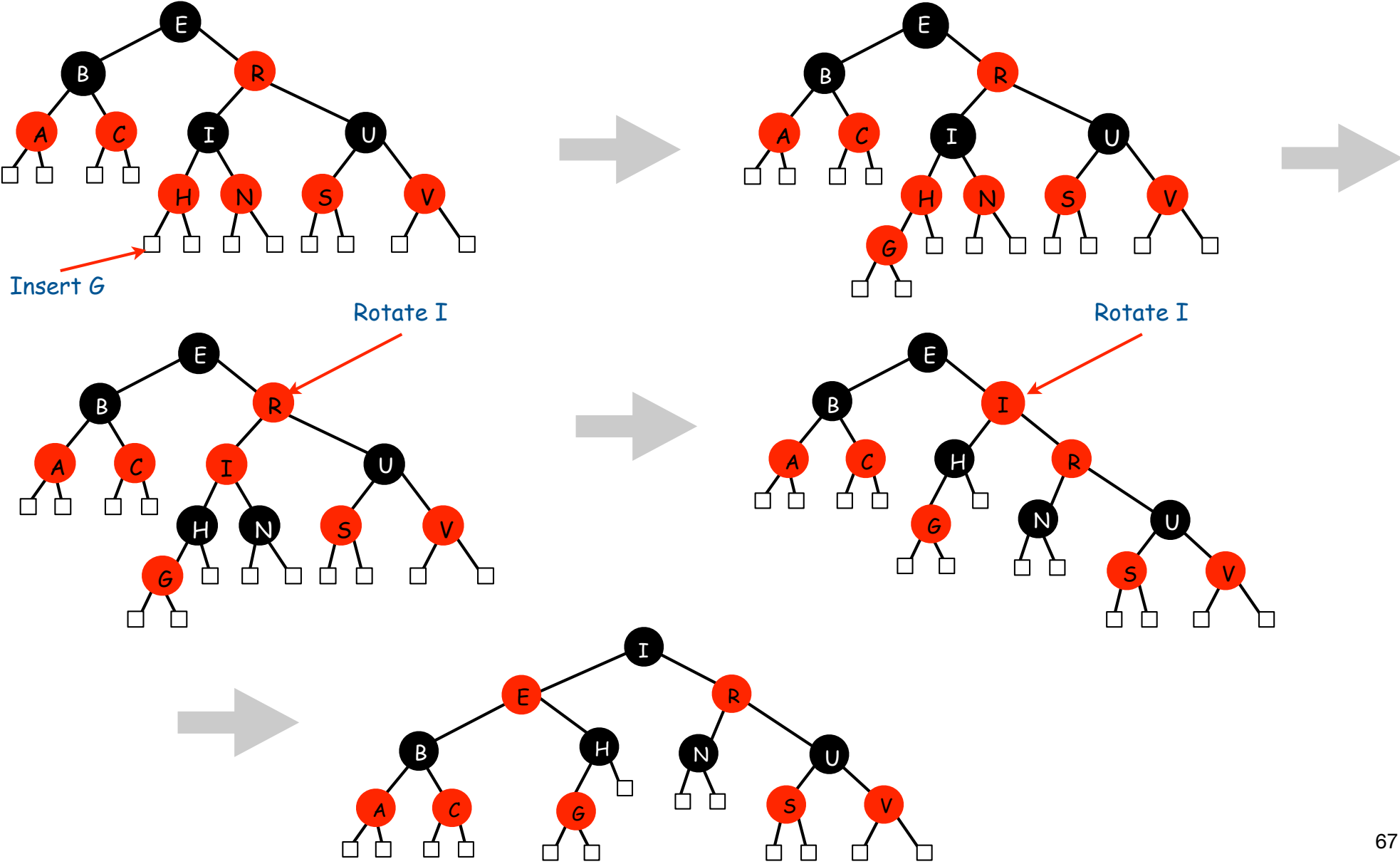
Rotate I



Rotate I



Example



Running times in red-black trees

- Time for insertion:
 - Search to bottom after key: $O(h)$
 - Insert red leaf: $O(1)$
 - Perform recoloring and rotations on way up: $O(h)$
 - Can recolor many times (but at most h)
 - At most 2 rotations.
- Total $O(h)$.
- Time for search
 - Same as BST: $O(h)$
- Height: $O(\log n)$

Dynamic set implementations

Worst case running times

Implementation	search	insert	delete	minimum	maximum	successor	predecessor
linked lists	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
ordered array	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(\log n)$	$O(\log n)$
BST	$O(h)$	$O(h)$	$O(h)$	$O(h)$	$O(h)$	$O(h)$	$O(h)$
2-3-4 tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
red-black tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

Balanced trees: implementations

Redblack trees:

Java: `java.util.TreeMap`, `java.util.TreeSet`.

C++ STL: `map`, `multimap`, `multiset`.

Linux kernel: `linux/rbtree.h`.