
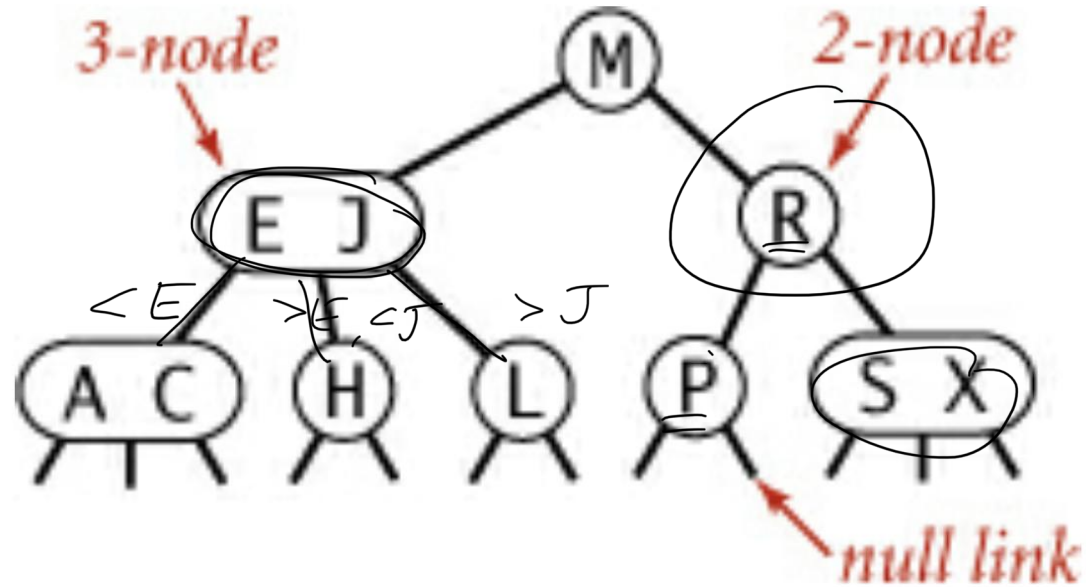


# 2-3 Trees

- An *ordered* symbol table
  - A self-balancing tree
- 
- A large, dark blue, curved shape that starts from the bottom left and extends diagonally upwards towards the right, filling the lower half of the slide.

# 2-3 Trees

Key Idea: We need a tree that is more flexible than a BST so that it can more easily balance itself.



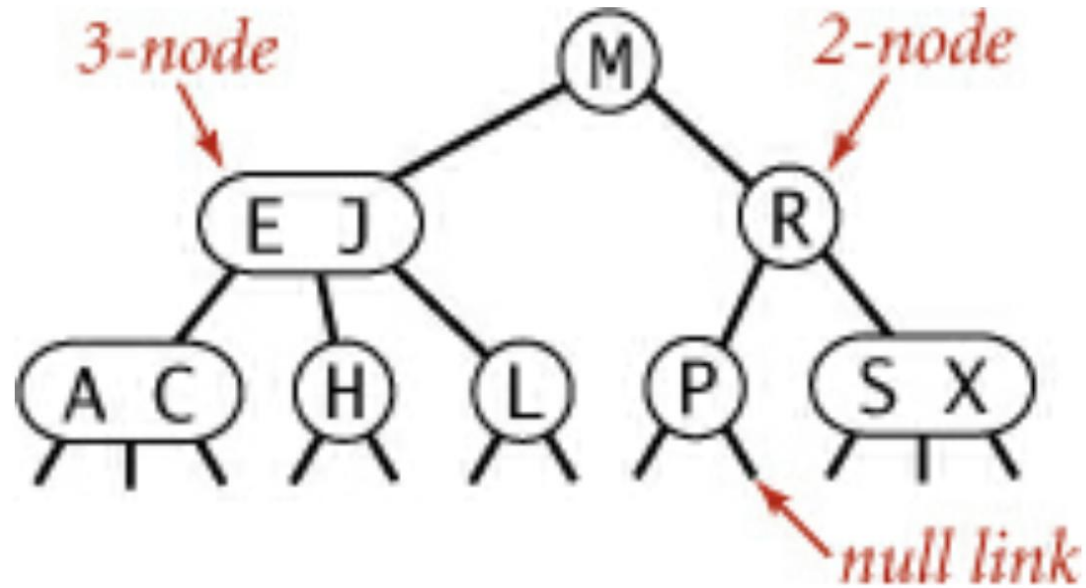
Anatomy of a 2-3 search tree

Picture from [1]

A 2-3 tree allows Nodes to have: (a) 1 key and two links or (b) 2 keys and three links

# 2-3 Trees

- Maintains Order
- Guarantees that every null link is the same distance from the root
- What is the height of the tree given  $N$  nodes?

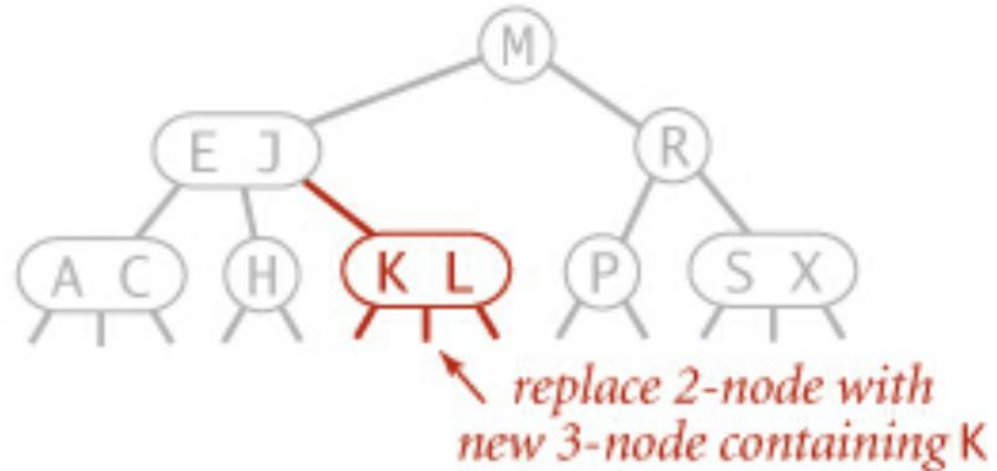
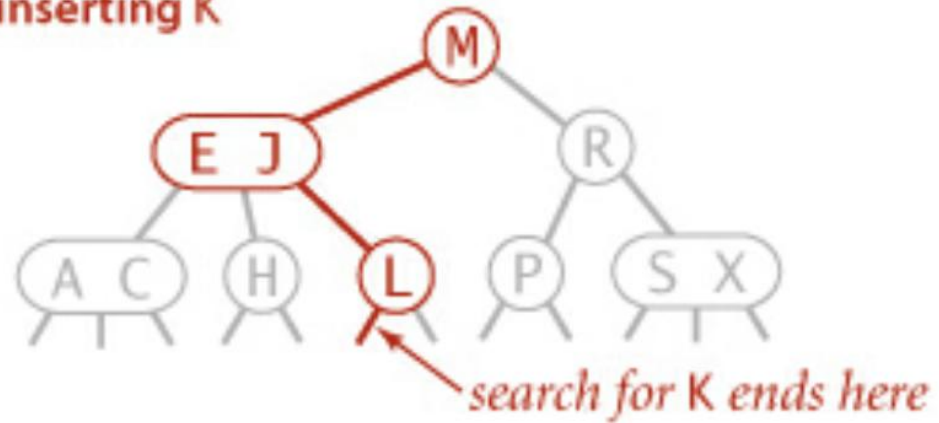


**Anatomy of a 2-3 search tree**

# put(K):

adding to a 2-node

inserting K



**Insert into a 2-node**

# put(S):

adding to a single 3-node (the root of the tree)

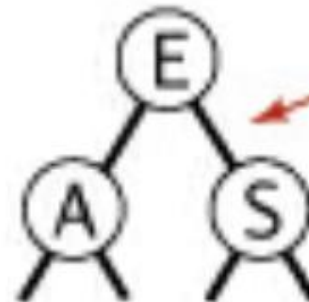
inserting S



← no room for S



← make a 4-node



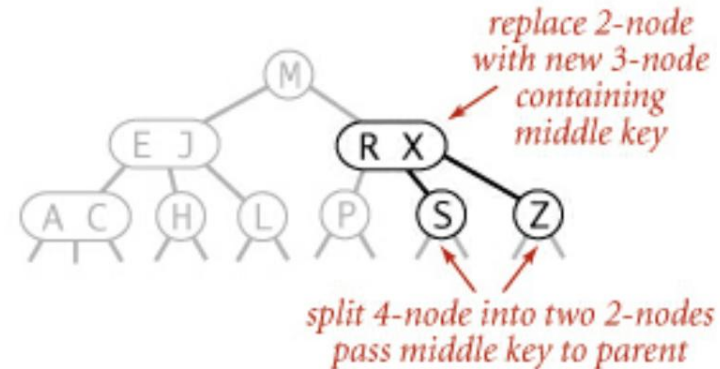
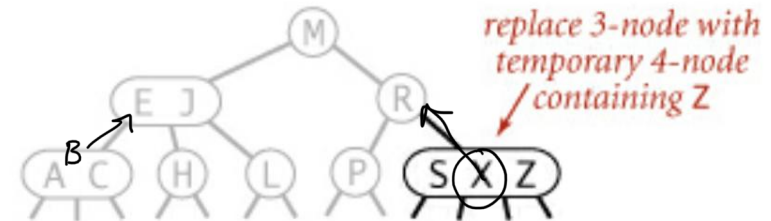
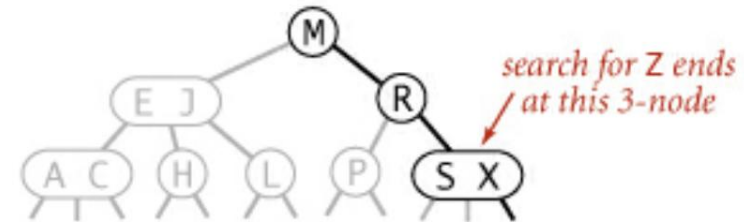
← split 4-node into this 2-3 tree

**Insert into a single 3-node**

# put(Z):

adding to a 3-node

inserting Z



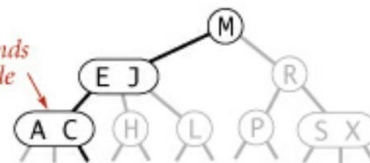
Insert into a 3-node whose parent is a 2-node

# put(D):

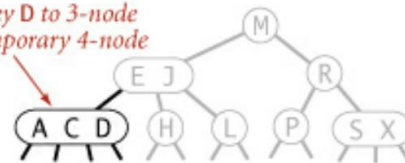
## adding to a 3-node

inserting D

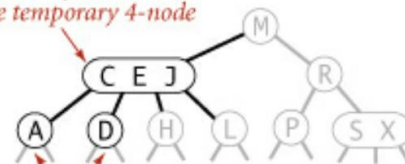
search for D ends  
at this 3-node



add new key D to 3-node  
to make temporary 4-node

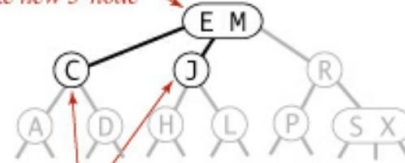


add middle key C to 3-node  
to make temporary 4-node



split 4-node into two 2-nodes  
pass middle key to parent

add middle key E to 2-node  
to make new 3-node



split 4-node into two 2-nodes  
pass middle key to parent

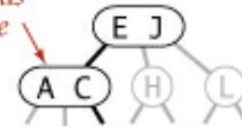
Insert into a 3-node whose parent is a 3-node

# put(D):

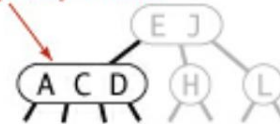
## adding to a 3-node

inserting D

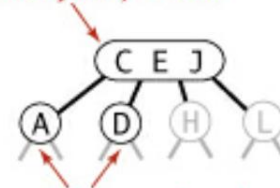
search for D ends  
at this 3-node



add new key D to 3-node  
to make temporary 4-node

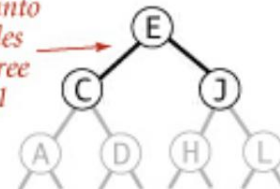


add middle key C to 3-node  
to make temporary 4-node



split 4-node into two 2-nodes  
pass middle key to parent

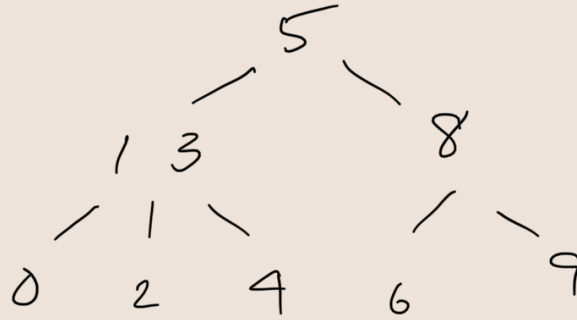
split 4-node into  
three 2-nodes  
increasing tree  
height by 1



Splitting the root



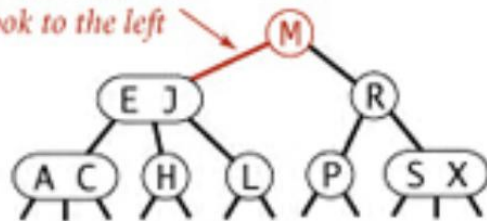
Example: ~~insert 8, insert 2, insert 3, insert 4, insert 0,~~  
~~insert 9, insert 6, insert 5, insert 1~~



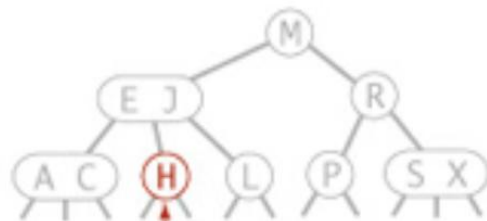
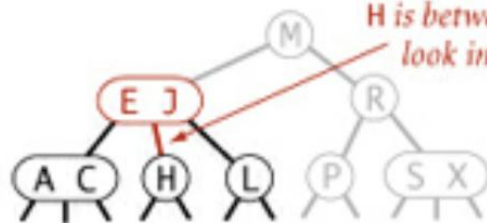
# get

successful search for H

*H is less than M so  
look to the left*



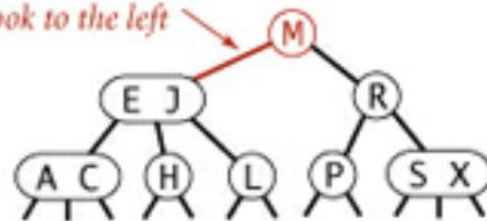
*H is between E and J so  
look in the middle*



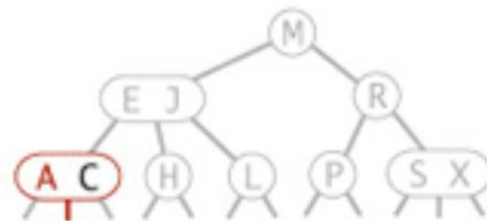
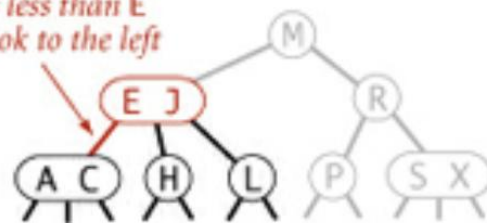
*found H so return value (search hit)*

unsuccessful search for B

*B is less than M so  
look to the left*



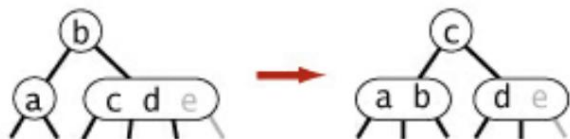
*B is less than E  
so look to the left*



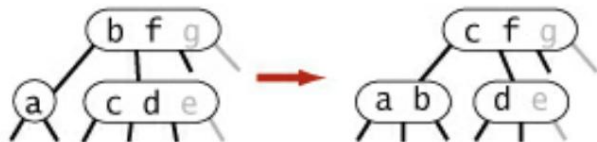
*B is between A and C so look in the middle  
link is null so B is not in the tree (search miss)*

Search hit (left) and search miss (right) in a 2-3 tree

at the root



on the way down



at the bottom



Transformations for delete the minimum

## Big Idea for Deletion:

It is easy to delete a key from a 3-node (or a 4-node) at the bottom of the tree. The tricky part is deleting a 2-node. (WHY?)

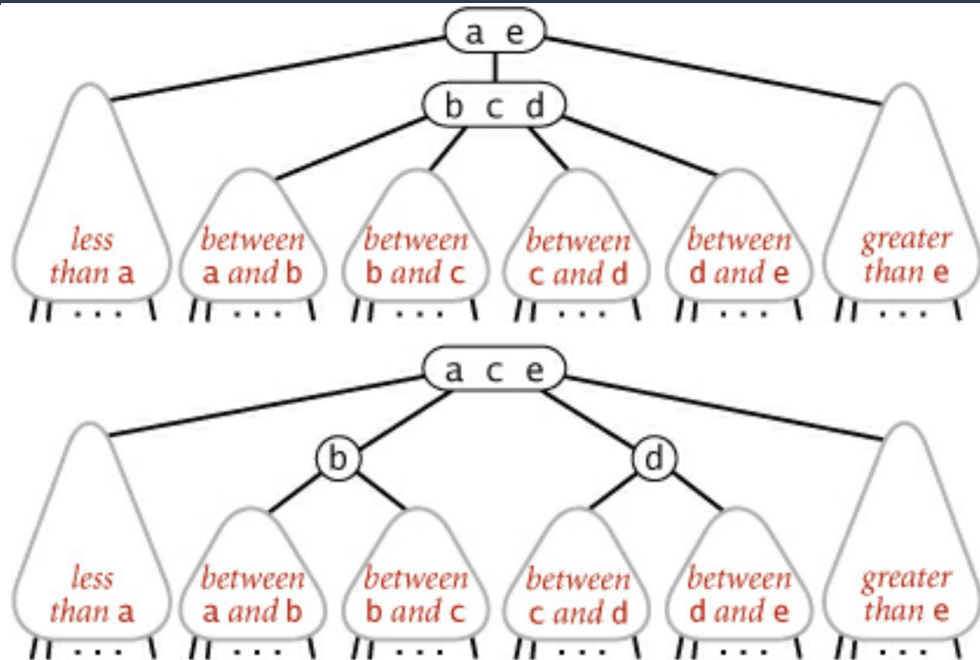
So we can transform the tree on the way down to ensure that the current node is not a 2-node. (HOW?)

Example: delete 3, delete 9, delete 5, delete 0, delete 1,  
delete 8, delete 4, delete 6, delete 7, delete 2

# Analysis of put algorithm

- **local** transformations: only the specified nodes need to be examined—number of links changed is bounded by a small constant
- **global** properties of the tree are preserved (order, balance, height\*)

\*height is increased by 1 when the root splits (all null nodes still have equal depth)



Splitting a 4-node is a local transformation that preserves order and perfect balance

root



parent is a 3-node

left



parent is a 2-node

left



middle



right



right



Splitting a temporary 4-node in a 2-3 tree (summary)

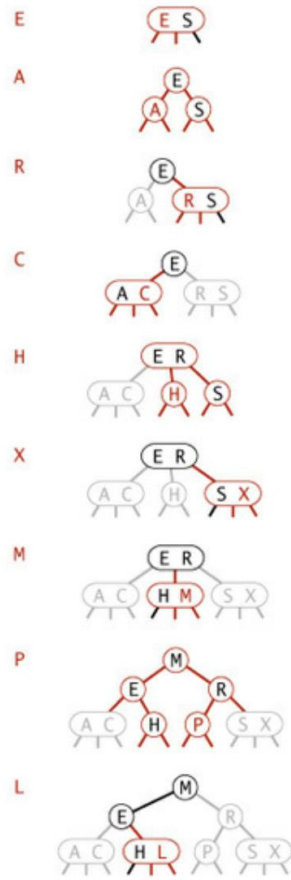
# Proposition F

Search and insert operations in a 2-3 tree with  $N$  keys are guaranteed to visit at most  $\lg N$  nodes.

- Consider the two “extremes”: the tree is made up of all 3 nodes or the tree is made up of all 2 nodes
- All transformations (local) take constant time
- Each operation touches nodes on a single path

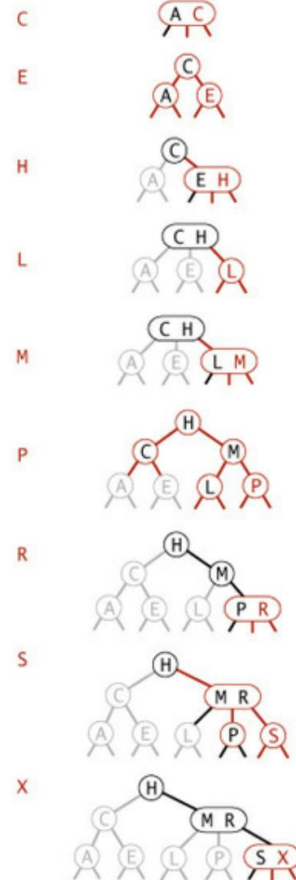
$$\log_3 N \leq h \leq \log_2 N$$

insert S



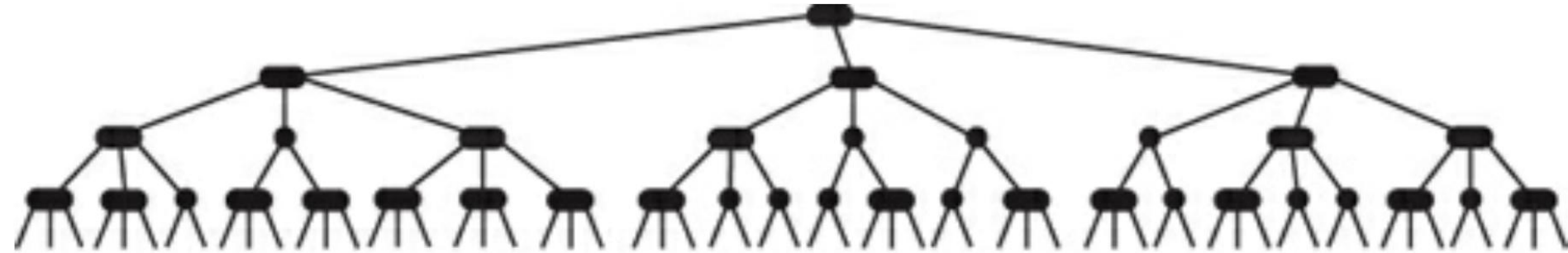
standard indexing client

insert A



same keys in increasing order





Typical 2-3 tree built from random keys

Picture from [1]

How might this be different if the keys are in decreasing order?

# 2-3 Trees

## Pros

- Good **guaranteed** worst-case performance for basic operations

## Cons

- Not “standard” trees—include two kinds of nodes
- Difficult to implement
- Implementation overhead could make it even worse to use than regular BST

# References

- [1] *Algorithms, Fourth Edition*; Robert Sedgewick and Kevin Wayne (and associated slides)
- [2] Slides from <https://www.cs.princeton.edu/~rs/talks/LLRB/RedBlack.pdf>