Fu-Yin Cherng

National Taiwan University

Balanced Search Tree

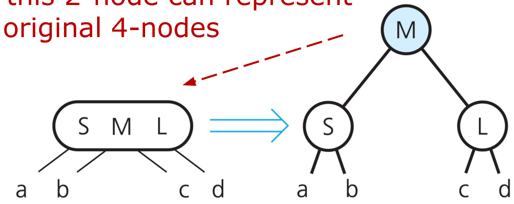
Outline

- Red-Black Trees
- AVL Trees

Red-Black Trees

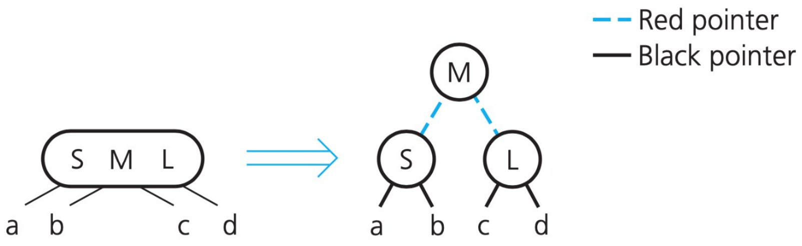
- A 2-3-4 tree requires more storage than a binary search tree
- Use special binary search tree to represent a 2-3-4 tree
  - retains the advantages of a 2-3-4 tree
  - without the storage overhead
- How?
  - split 2-nodes from 3- or 4-nodes (insertion of 2-3-4 tree)

use this 2-node can represent
the original 4-nodes



Red and Black Pointer

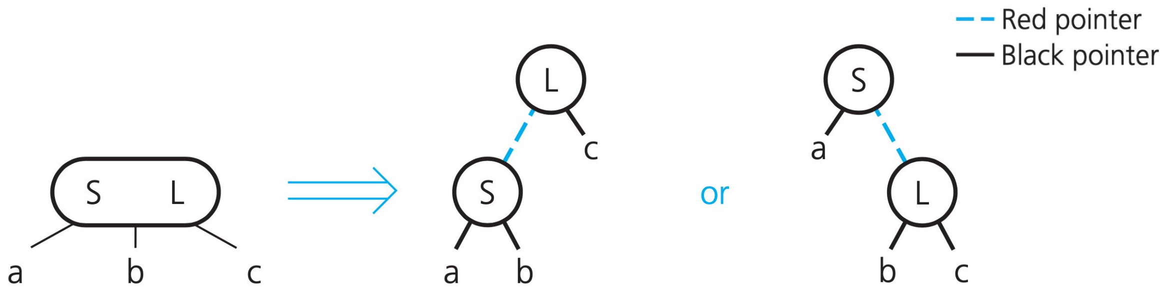
- to distinguish between original 2-nodes & 2-nodes generated from 3- and 4-nodes
- red & black child pointers
  - black: link the original 2-nodes
  - red: link the 2-nodes that now contain the values that were in a 3-node or a 4-node.



2-node generated from 4-node

Red and Black Pointer

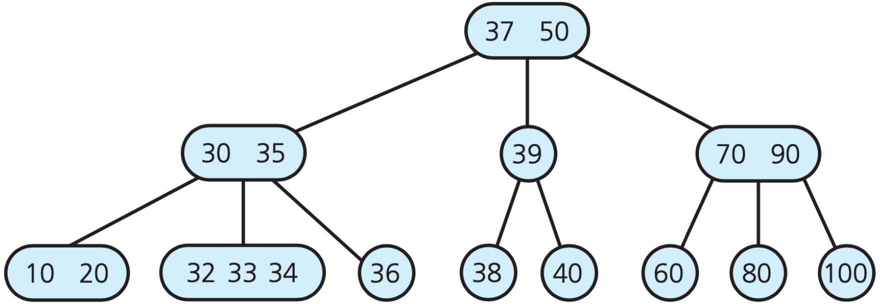
- 2 possible way to represent a 3-node as a binary tree
- a red-black representation of a 2-3-4 tree is not unique



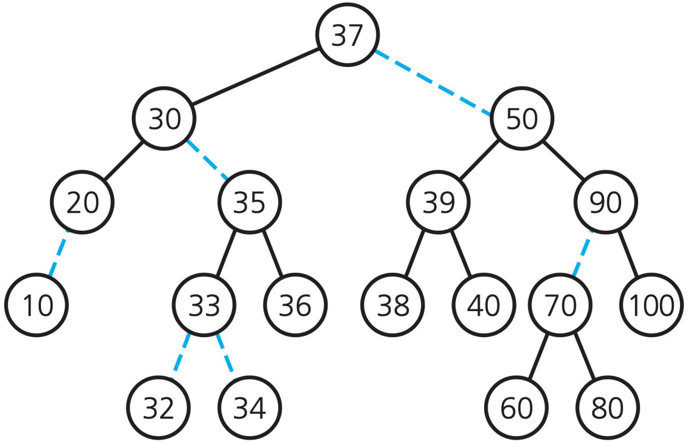
2-node generated from 3-node

Use Red-Black Trees to present 2-3-4 tree: example

Let see what nodes are changed to 2-node representation!

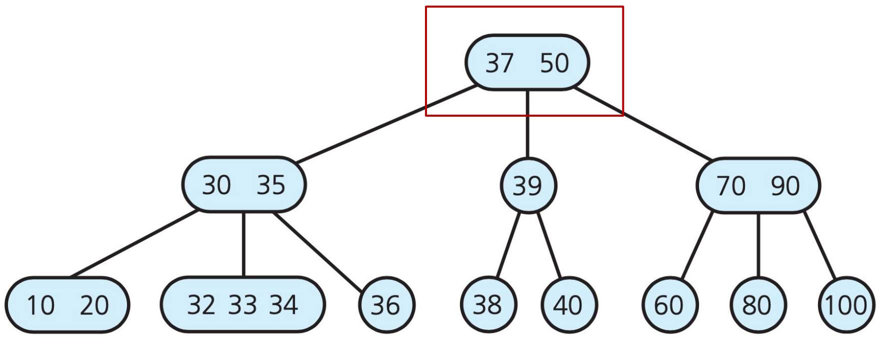


2-3-4 tree

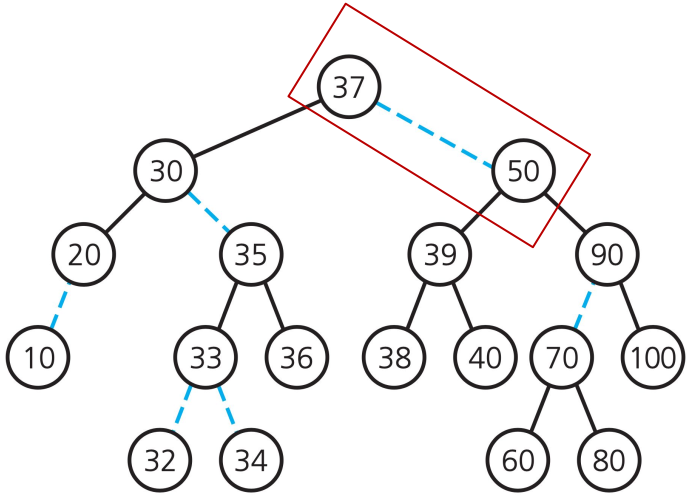


A red-black tree that represents the 2-3-4 tree

Use Red-Black Trees to present 2-3-4 tree: example

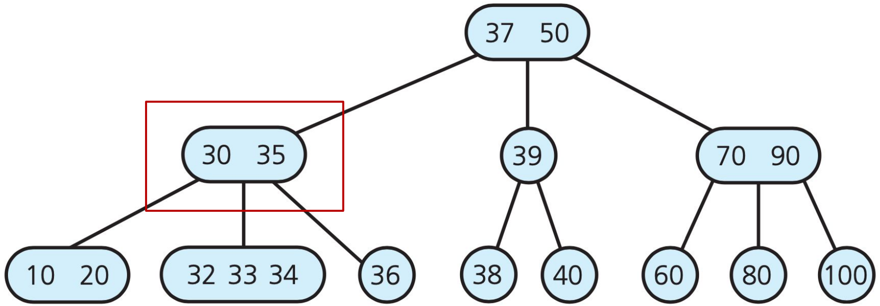


2-3-4 tree

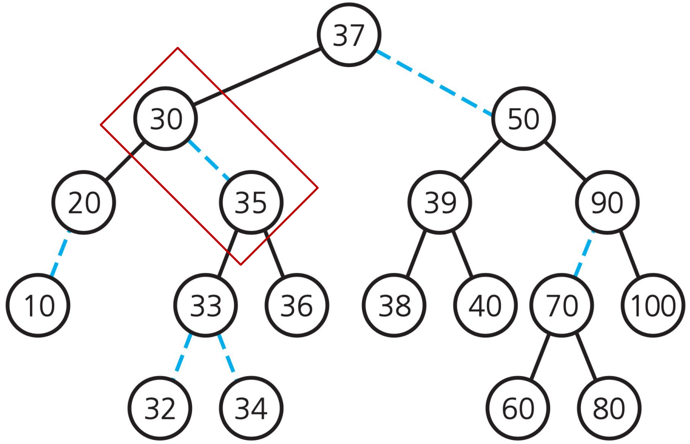


A red-black tree that represents the 2-3-4 tree

Use Red-Black Trees to present 2-3-4 tree: example

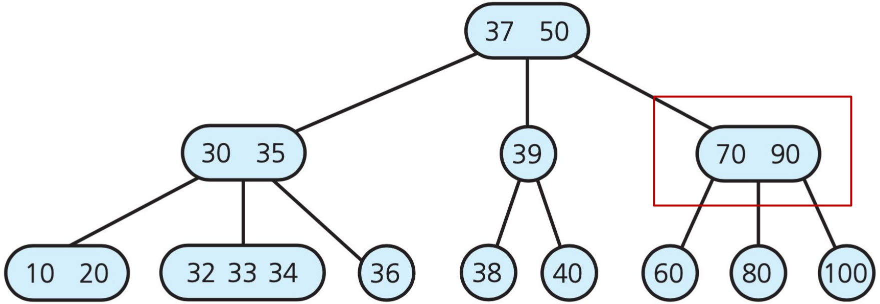


2-3-4 tree

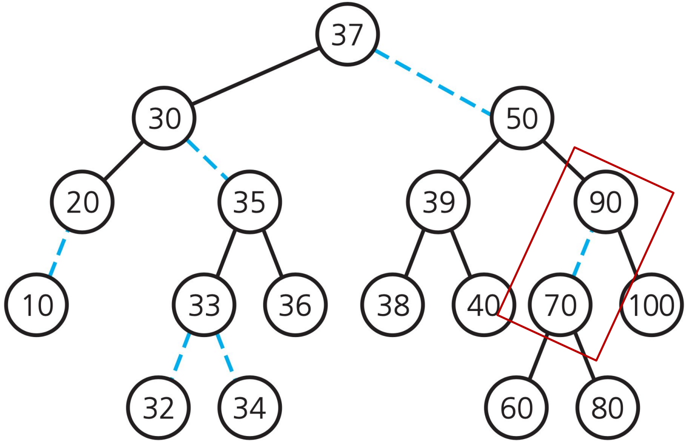


A red-black tree that represents the 2-3-4 tree

Use Red-Black Trees to present 2-3-4 tree: example

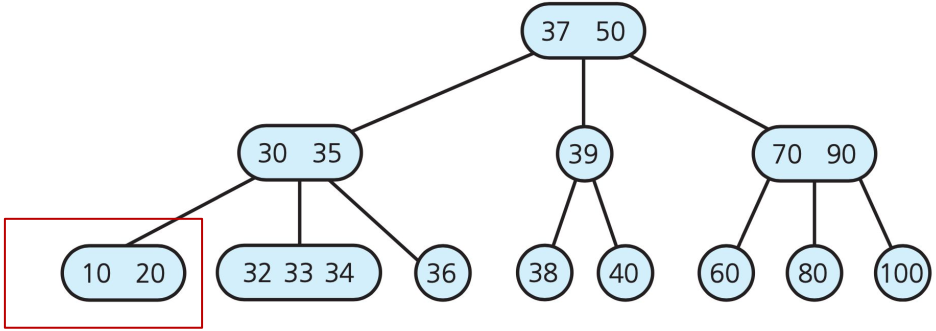


2-3-4 tree

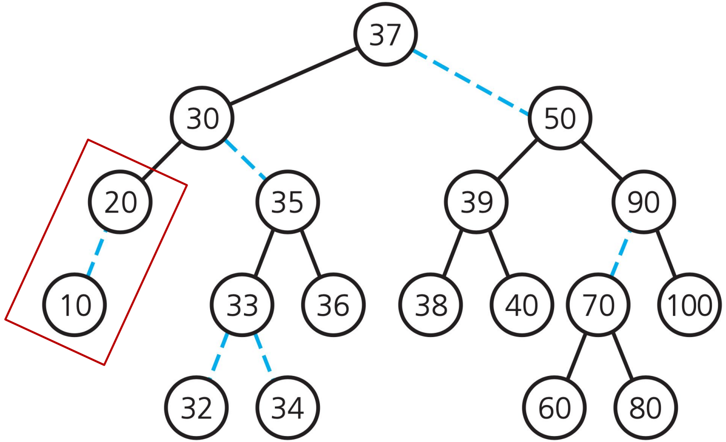


A red-black tree that represents the 2-3-4 tree

Use Red-Black Trees to present 2-3-4 tree: example

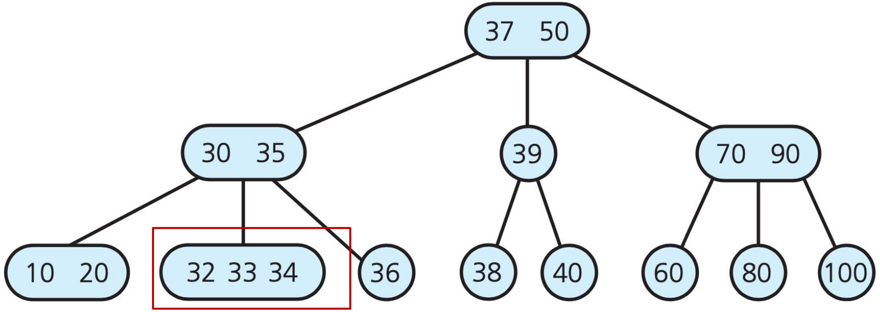


2-3-4 tree

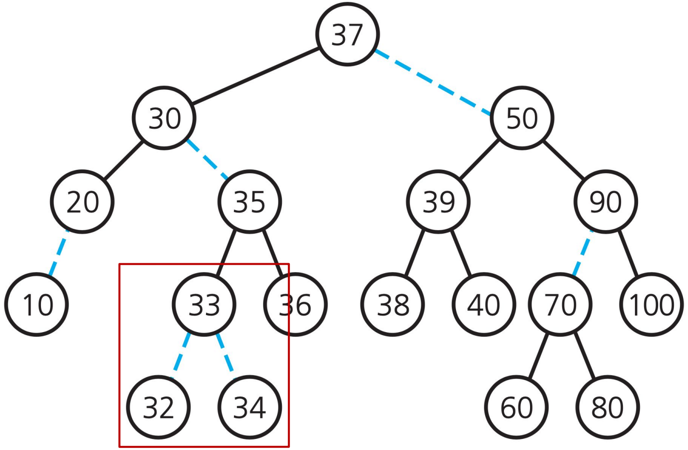


A red-black tree that represents the 2-3-4 tree

Use Red-Black Trees to present 2-3-4 tree: example

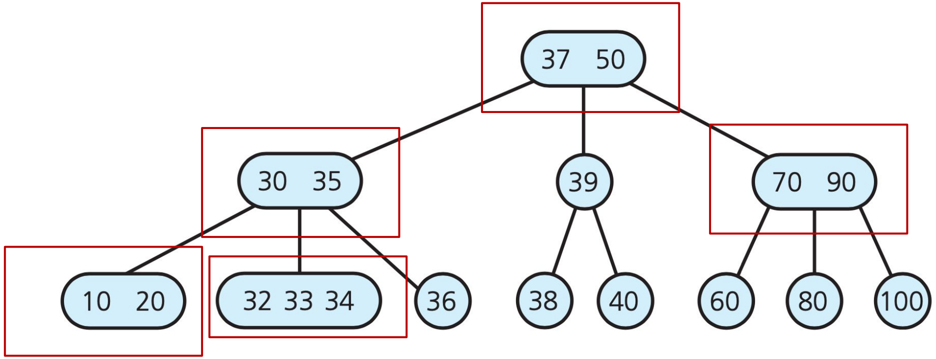


2-3-4 tree

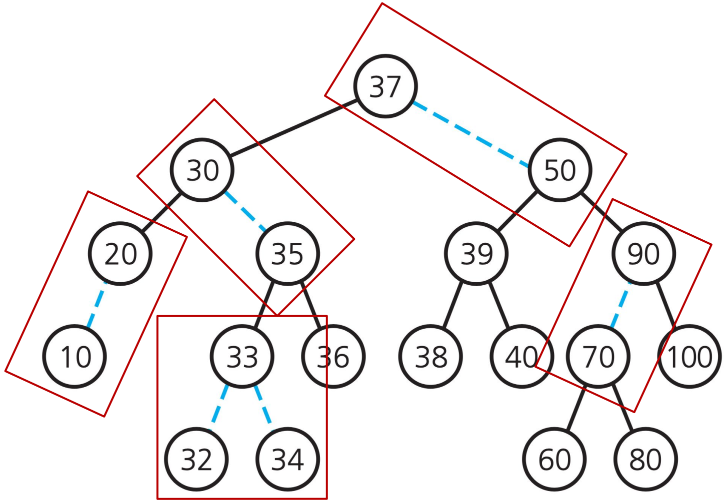


A red-black tree that represents the 2-3-4 tree

Use Red-Black Trees to present 2-3-4 tree: example



2-3-4 tree



A red-black tree that represents the 2-3-4 tree

A node in red-black tree

- Although need to store pointer colors, red-black node still require less storage than a node in 2-3-4 tree

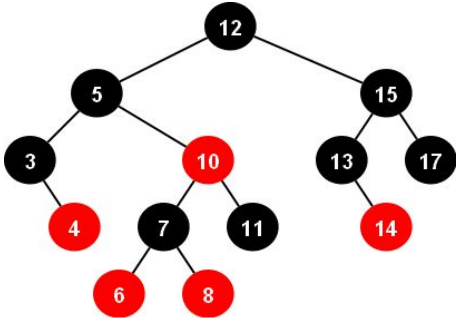
```
enum Color {RED, BLACK};

template<class ItemType>
class RedBlackNode : public BinaryNode<ItemType>{
     private:
          Color leftColor, rightColor;

     public:
          ...
}
```

Red-black tree: colored pointer or node?

- this textbook introduces red-black tree as the tree with colored pointer
- but if searching online (e.g., wiki), most tutorial introduce red-black tree as the tree with colored node
- Basically the same, the node with red pointer is the red node

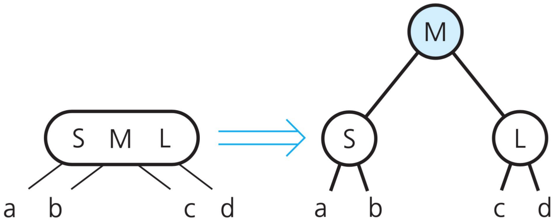


Searching and Traversing a Red-Black Tree

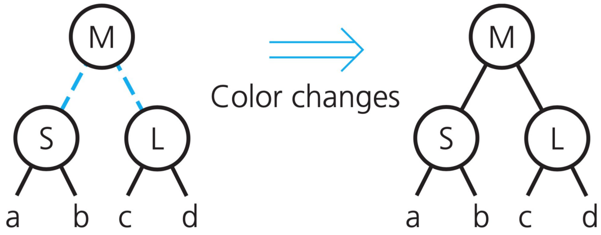
- A red-black tree is a binary search tree
- Just use the search and traversal algorithms for a binary search tree (see previous video and slides)
- simply ignore the color of pointer

Insertion of Red-black Tree

- a red-black tree actually represents a 2-3-4 tree
- just need to some adjustment
- For inseration
  - For 2-3-4 tree, split 4-node when encountered it
  - For red-black tree,
    - identify 4-node by checking color of pointer (2 red pointers)
    - change color of pointers



2-3-4 tree: split 4-node

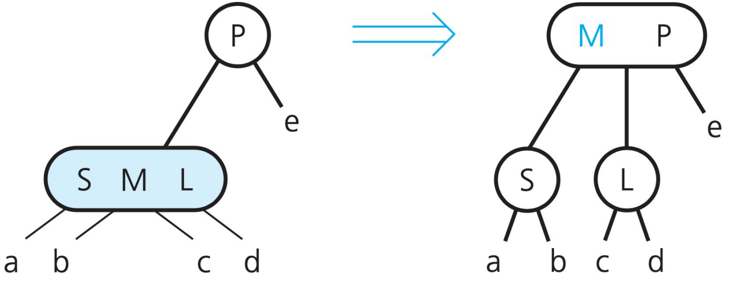


red-black tree: split 4-node

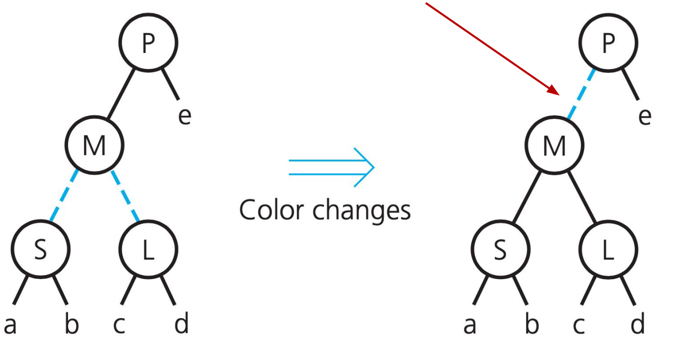
Insertion of Red-black Tree: split 4-node

- For inseration, split 4-node with 2-node parent

*M move up to parent <P>, so here is the 2-node representation of 3-node <M P> in red-black tree



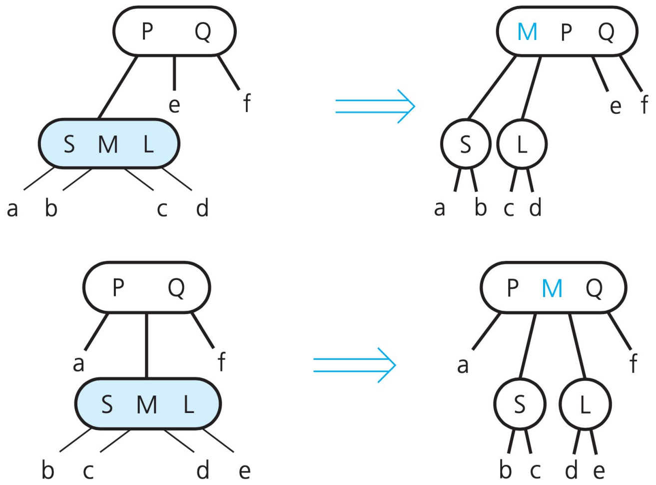
2-3-4 tree: split 4-node



red-black tree: split 4-node

Insertion of Red-black Tree: split 4-node

- For inseration, split 4-node with 3-node parent



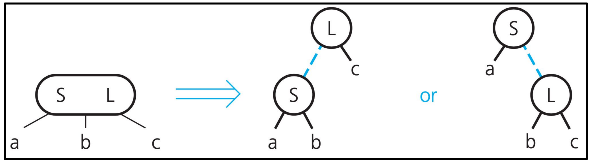
2-3-4 tree: split 4-node

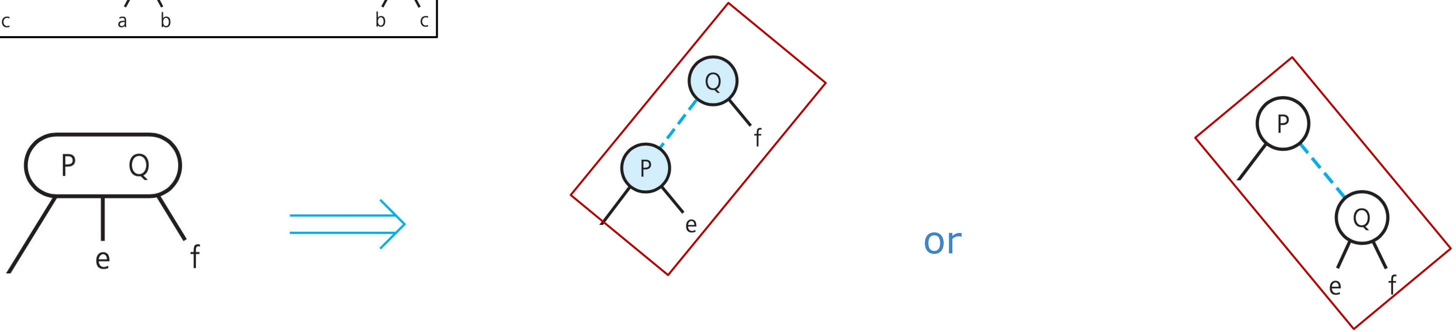
?

red-black tree: split 4-node

Insertion of Red-black Tree: split 4-node

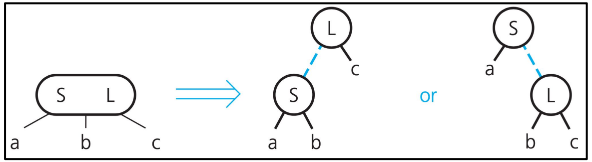
- 2 kinds of representation of 3-node parent <P Q>

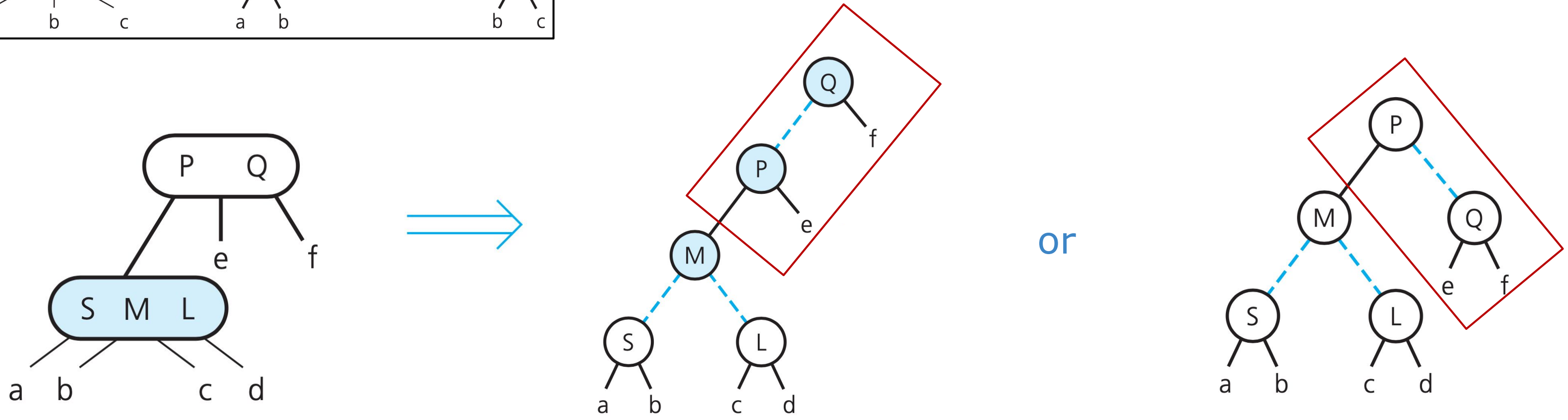




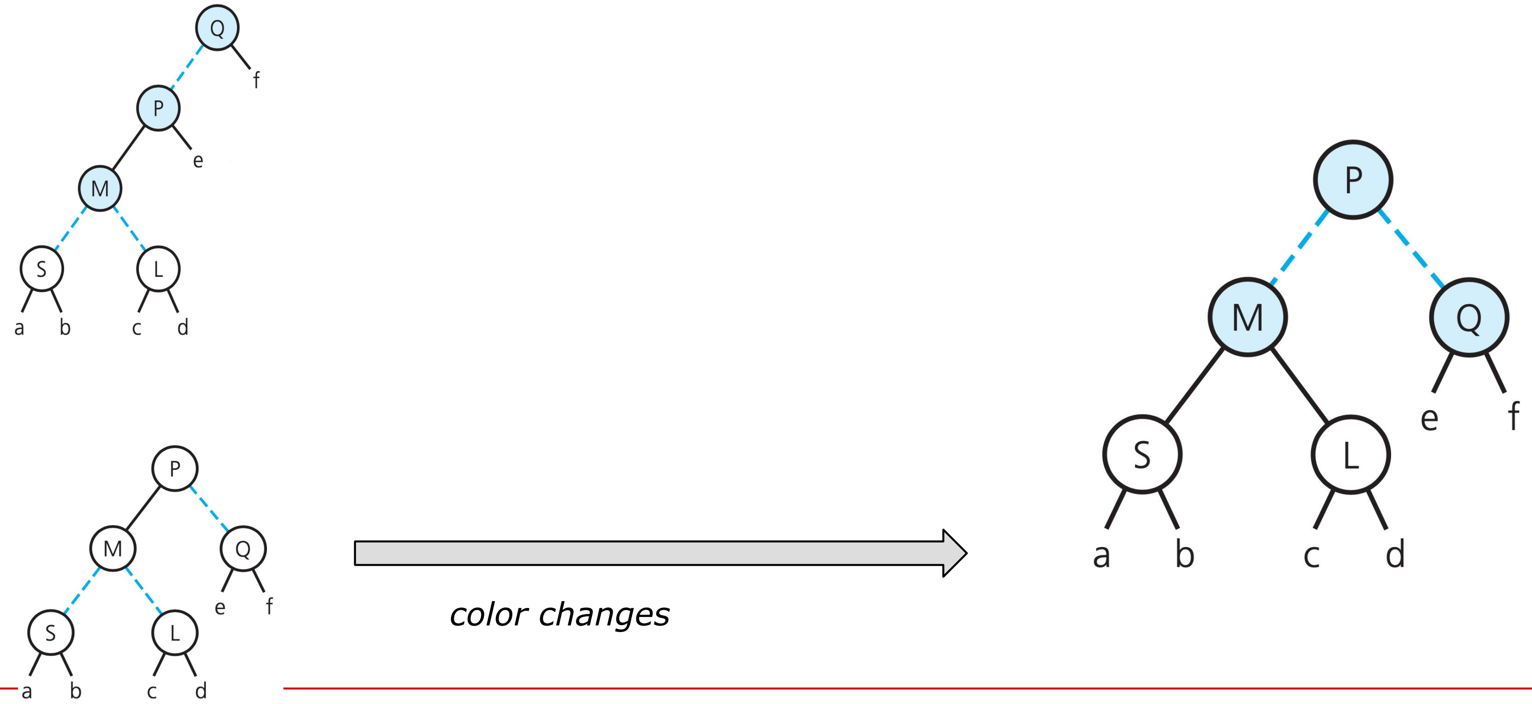
Insertion of Red-black Tree: split 4-node

- 2 kinds of representation of 3-node parent <P Q>

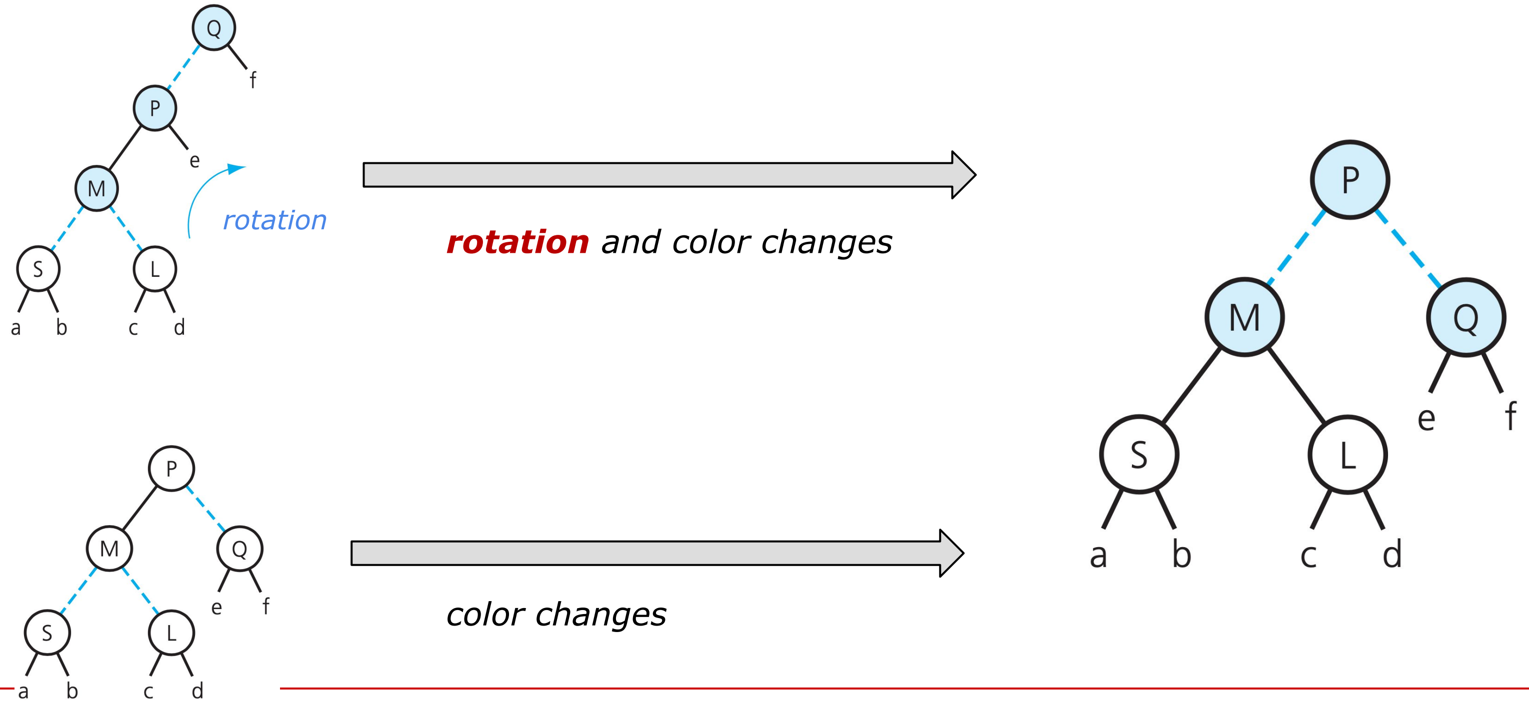




Insertion of Red-black Tree: split 4-node

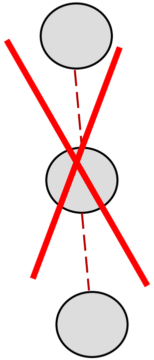


Insertion of Red-black Tree: split 4-node



Insertion of Red-black Tree: split 4-node

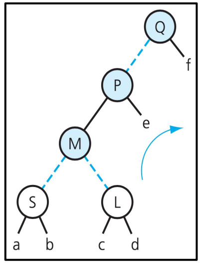
- Why and How to do rotation?
  - Why: fix the structure and follow the rules of red-black tree
    - a node with red pointer to parent cannot have red pointer to child

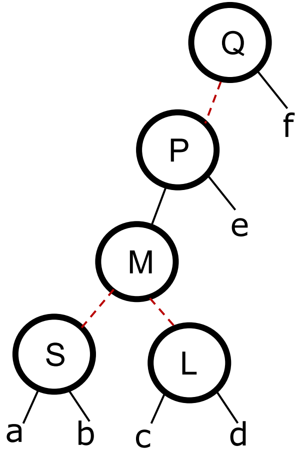


- How? There are many online tutorial of how to do rotation for red-black tree (example)
  - left, right, right-left… rotation
  - can still try to learn these rules if you like
- **But**, as we know the rules of 4-nodes from 2-3-4 tree, we can apply it to understand rotation!

Insertion of Red-black Tree: split 4-node

- split 4-node <S M L>: move up Middle value and split the Small and Large

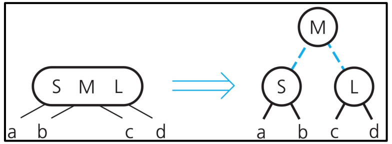




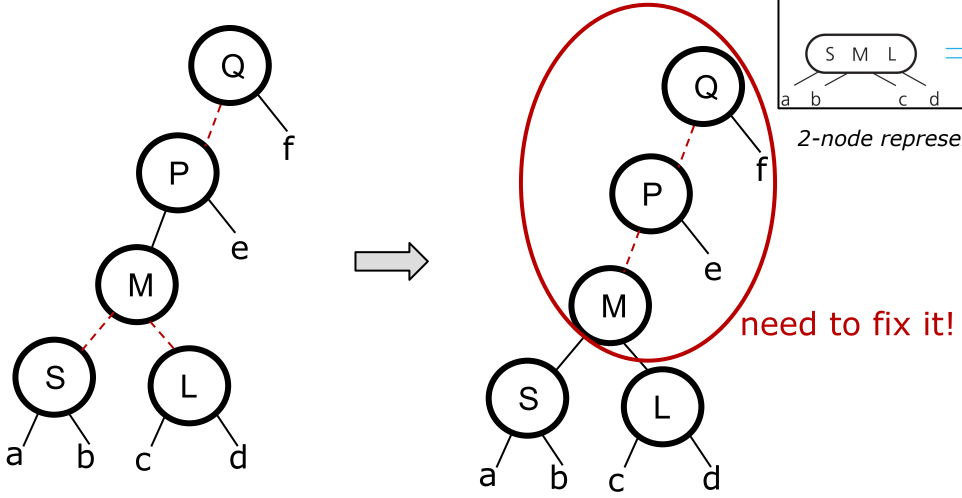
split <S M L>
Move up M ans split S and L (change color)

Insertion of Red-black Tree: split 4-node

- split 4-node <S M L>: move up Middle value and split the Small and Large



2-node representatino for 4-node

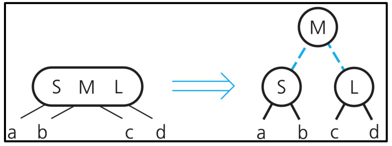


split <S M L>
Move up M ans split S and L (change color)

after moved up M, the <M P Q> violate the rules and 2-node representation for 4-node in red-black tree

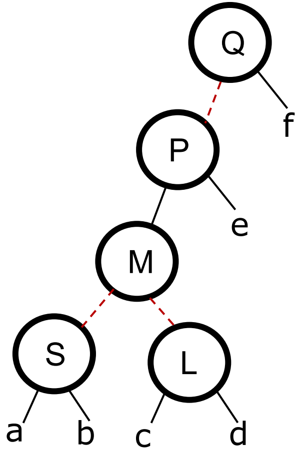
Insertion of Red-black Tree: split 4-node

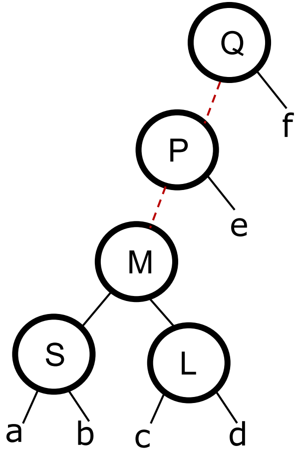
- split 4-node <S M L>: move up Middle value and split the Small and Large

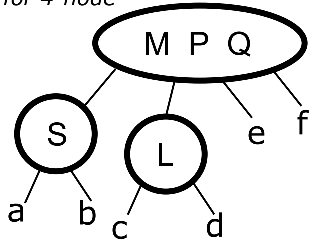


2-node representatino for 4-node

the order of placing M, P, Q is important







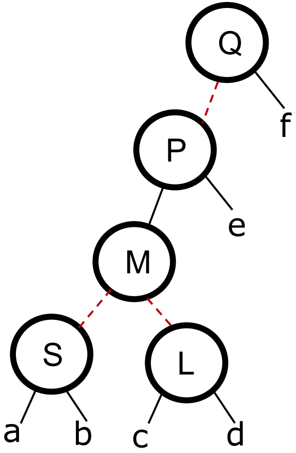
split <S M L>
Move up M ans split S and L (change color)

after moved up M, the <M P Q> violate the rules and 2-node representation for 4-node in red-black tree

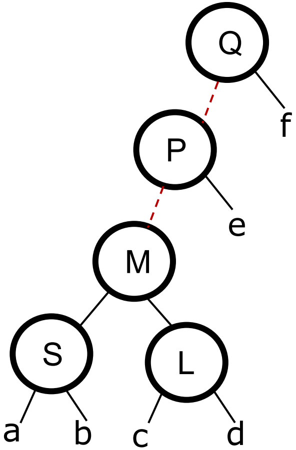
change <M P Q> back to 4-node, and then follow rules and change to 2-node representation again

Insertion of Red-black Tree: split 4-node

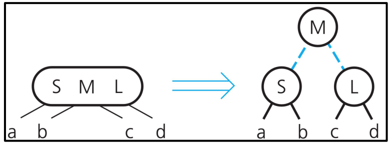
- split 4-node <S M L>: move up Middle value and split the Small and Large



split <S M L>
Move up M ans *split* S and L (*change color*)

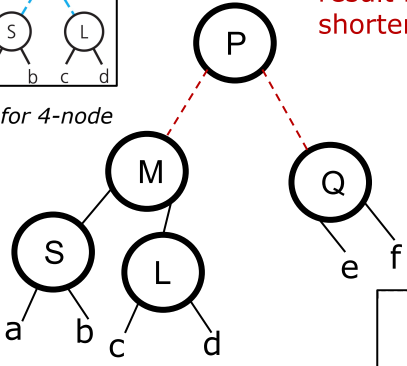


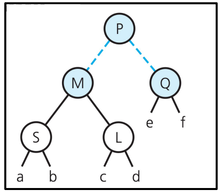
after moved up M, the <M P Q> violate the rules and 2-node representation for 4-node in red-black tree



2-node representatino for 4-node

result in shorter tree



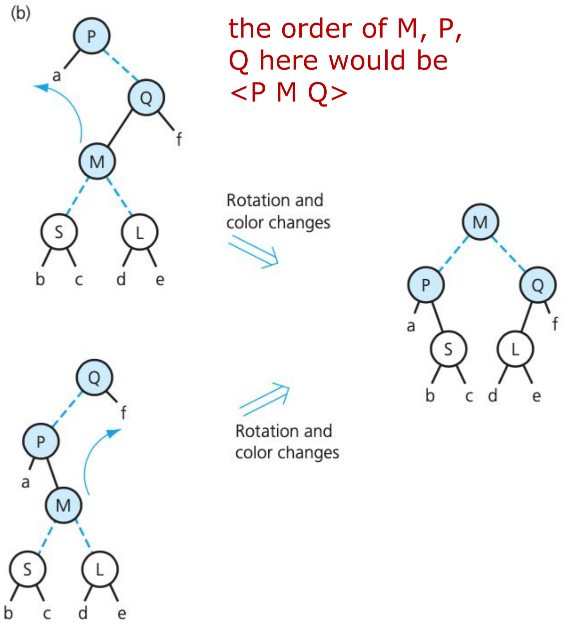


change <M P Q> back to 4-node, and then follow rules and change to 2-node representation again

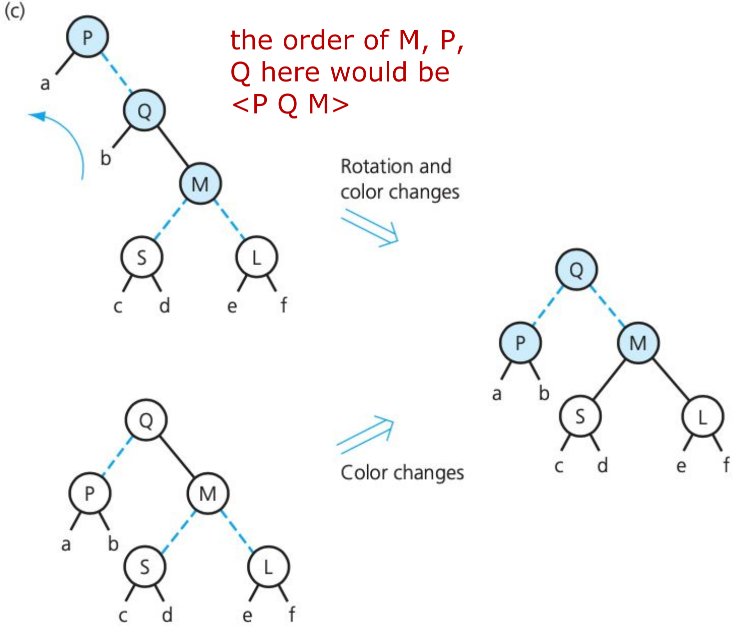
Insertion of Red-black Tree: split 4-node

- other possibilities with different orders among M, P, Q
- please try by yourself to complete splitting 4-node <S M L> in red-black tree

the order of M, P, Q here would be <P M Q>



the order of M, P, Q here would be <P Q M>



Insertion and Removal of Red-black Tree

- since we split 4-node when searching location to sert new value
- we only need to insert new value to the leaf node by using the red point
- **This textbook**
  - didn’t cover much more about insertion and removal of red-black tree (use similar methods from 2-3-4 tree)
  - focus on transformation between 2-3-4 and red-black tree, and how to split 4-node in red-balck tree

Advantage of red-black tree

- insertion and removal operation often require only color changes
- more efficient than the same operations on a 2-3-4 tree

Other resources of red-black tree

- https://en.wikipedia.org/wiki/Red%E2%80%93black_tree
- https://medium.com/swlh/red-black-tree-rotations-and-color-flips-10e87f72b142
- interactive website of red-black tree: https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

AVL Tree

- inventor: Adel'son-Vel'skii and Landis (invented in 1962)
- is a balanced binary search tree
- the heights of left and right subtree of any node differ by no more than 1
- as efficient as minimum-height binary search tree
- simply introduce the notion of AVL tree

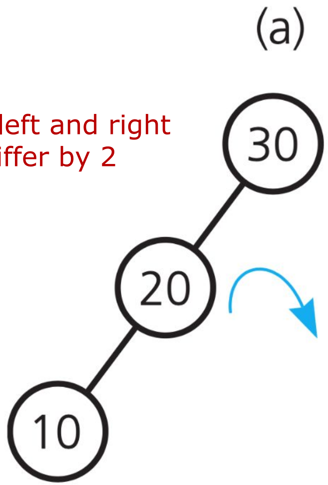
AVL Tree

- How to rearrange binary search tree to get minimum possible height ⌈log2(n + 1)⌉ (n: node numbers)?
  - save tree's value to file and then construct from the same value to new tree of minimum height
  - but, too costly if rebuilding tree every time when an insertion or removal makes tree unbalanced!
- AVL algorithm is a compromise

AVL Algorithm

- Goal: maintains a binary search tree with a height close to the minimum
  - less work than keeping the height equal to minimum
- Basic Strategy
  - monitor the shape of the binary search tree
  - after each insertion or deletion, check the if the tree is still an AVL tree
    - check if any node has left and right subtrees whose heights differ more than 1

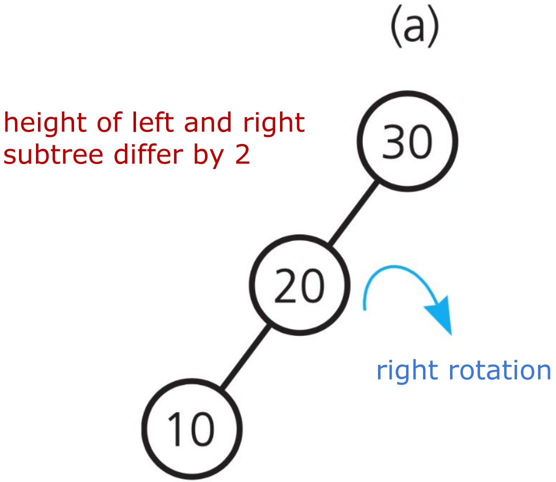
AVL Algorithm: example

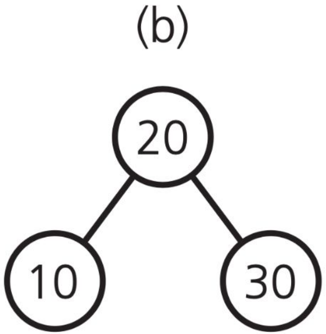


height of left and right subtree differ by 2

after a sequence of insertion and removals, get this binary search tree

AVL Algorithm: example

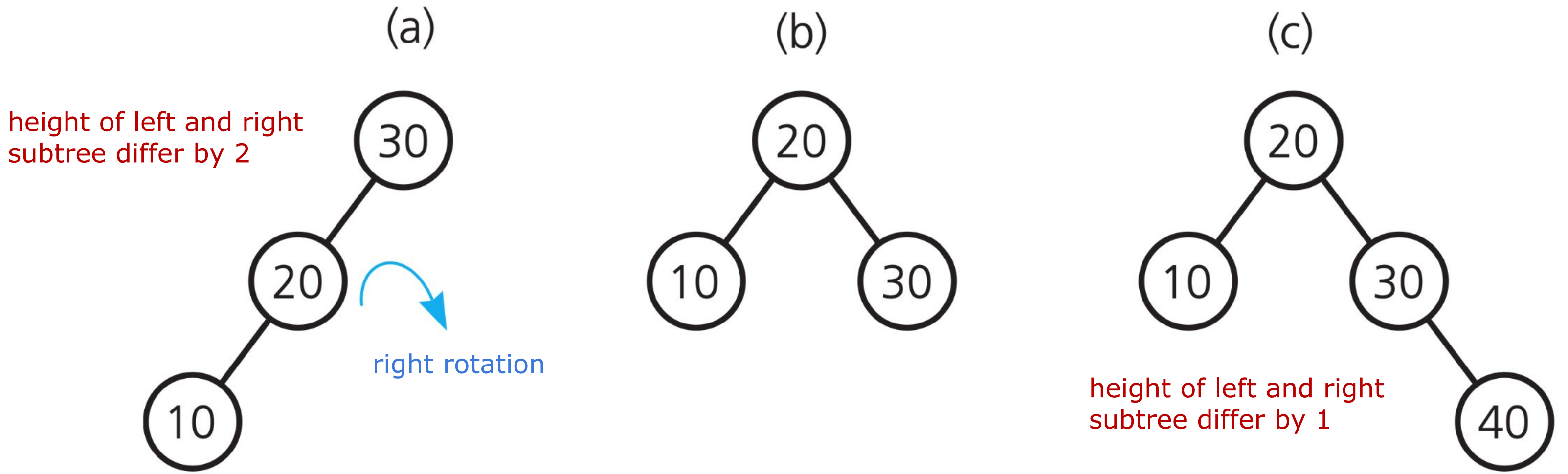




after a sequence of insertion and removals, get this binary search tree

rearrange to restore balance and keep the ordering property

AVL Algorithm: example



after a sequence of insertion and removals, get this binary search tree

rearrange to restore balance and keep the ordering property

insert <40>, but still a legit AVL tree, so no need for rotation

AVL Algorithm: Balance Factor

- compute balance factor of a AVL tree to see if rotation is needed
- AVL tree allow the absolute value of balance factor up to 1

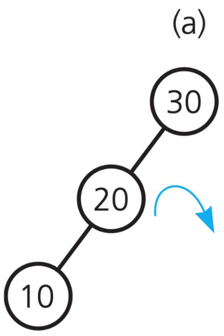
```
Balance Factor = height(right subtree) - height(left subtree)
```

AVL Algorithm: Balance Factor

- compute balance factor of a AVL tree to see if rotation is needed
- AVL tree allow the absolute value of balance factor up to 1

```
Balance Factor = height(right subtree) - height(left subtree)
```

balance factor of each node in previous example

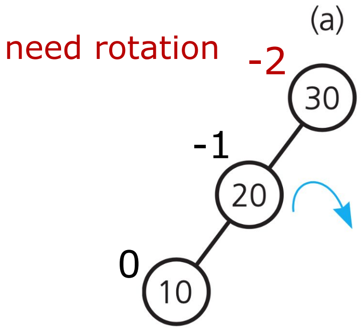


AVL Algorithm: Balance Factor

- compute balance factor of a AVL tree to see if rotation is needed
- AVL tree allow the absolute value of balance factor up to 1

```
Balance Factor = height(right subtree) - height(left subtree)
```

balance factor of each node in previous example

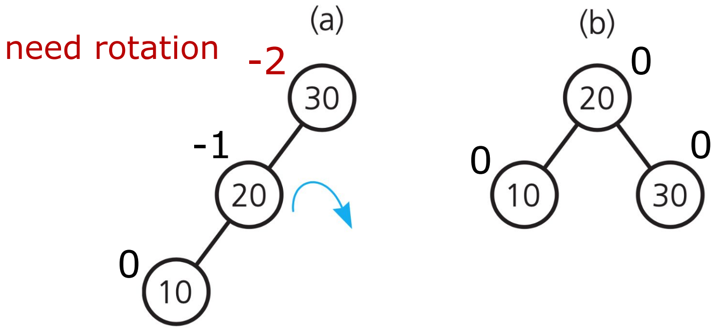


AVL Algorithm: Balance Factor

- compute balance factor of a AVL tree to see if rotation is needed
- AVL tree allow the absolute value of balance factor up to 1

```
Balance Factor = height(right subtree) - height(left subtree)
```

balance factor of each node in previous example

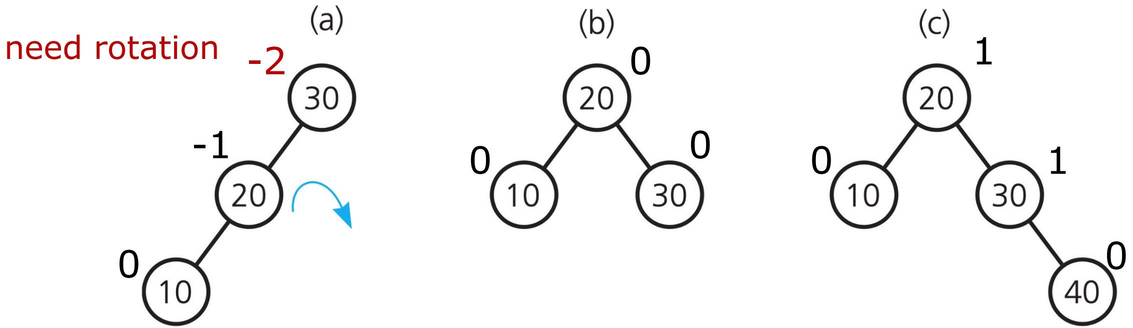


AVL Algorithm: Balance Factor

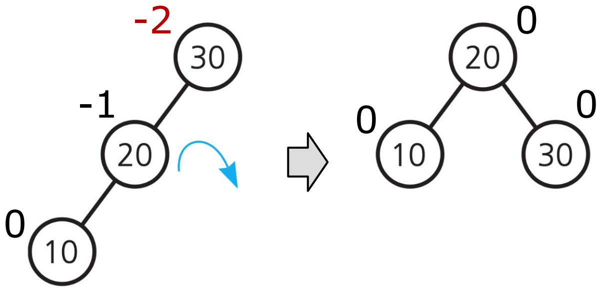
- compute balance factor of a AVL tree to see if rotation is needed
- AVL tree allow the absolute value of balance factor up to 1

```
Balance Factor = height(right subtree) - height(left subtree)
```

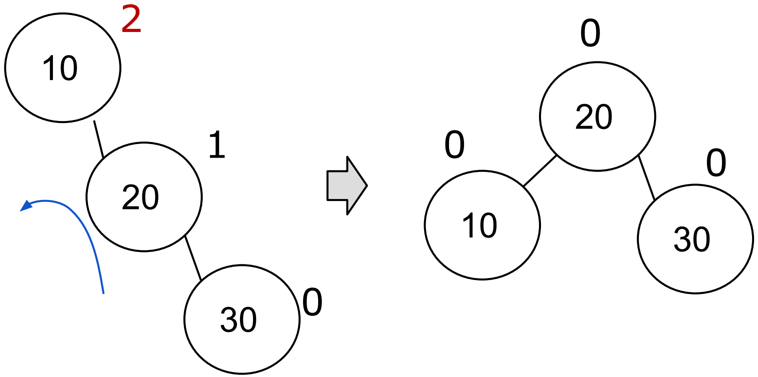
balance factor of each node in previous example



AVL Algorithm: Single Rotation



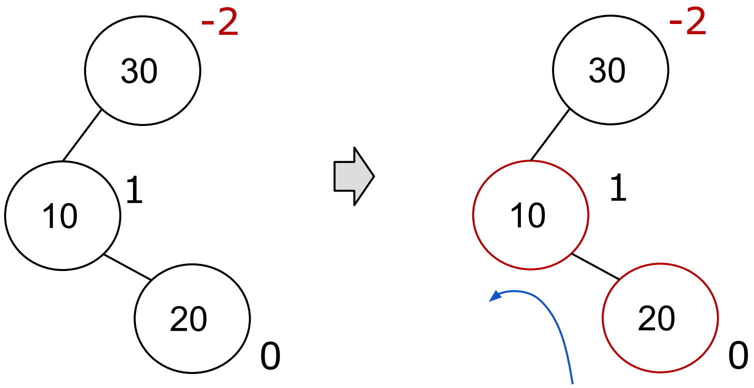
right rotation



left rotation

AVL Algorithm: Double Rotation

Left-Right Rotation

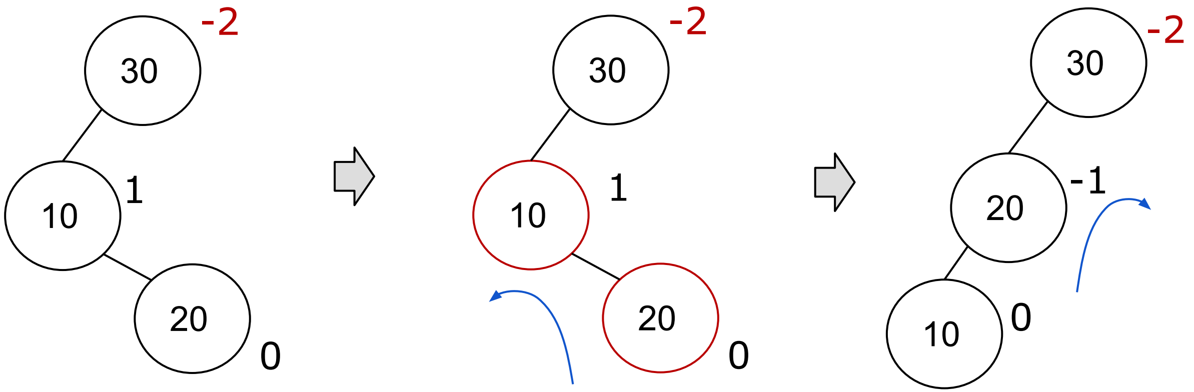


<30> is a unbalanced node

first perform left rotation on left subtree of <30>

AVL Algorithm: Double Rotation

Left-Right Rotation



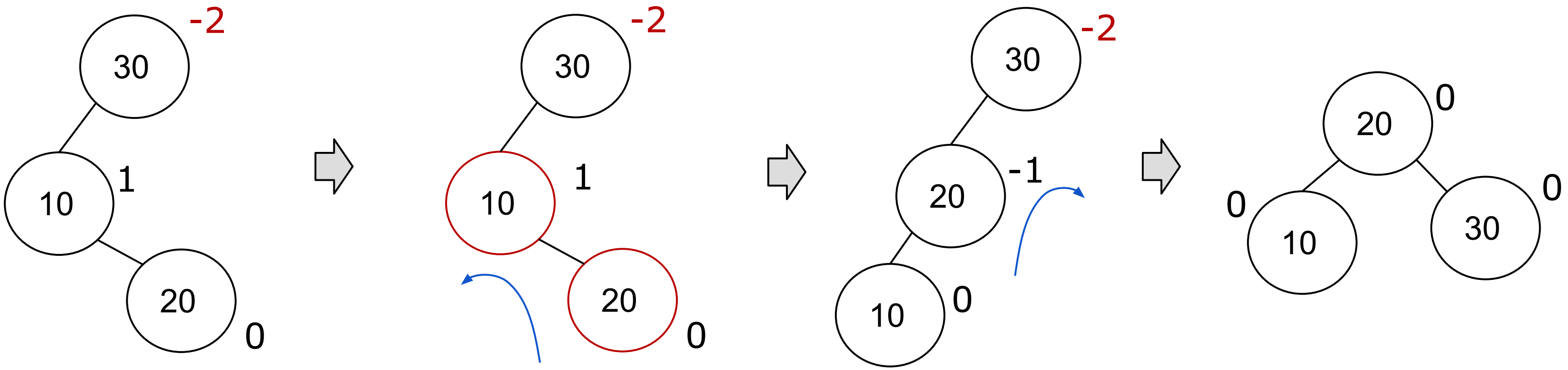
<30> is a unbalanced node

first perform left rotation on left subtree of <30>

perform right rotation

AVL Algorithm: Double Rotation

Left-Right Rotation



<30> is a unbalanced node

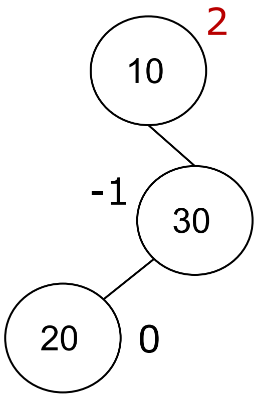
first perform left rotation on left subtree of <30>

perform right rotation

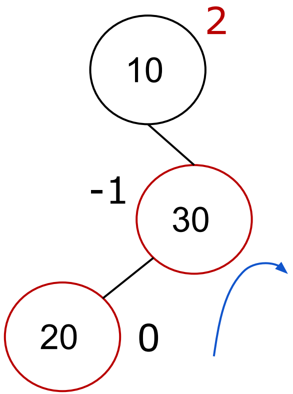
get a balanced tree

AVL Algorithm: Double Rotation

Right-left Rotation



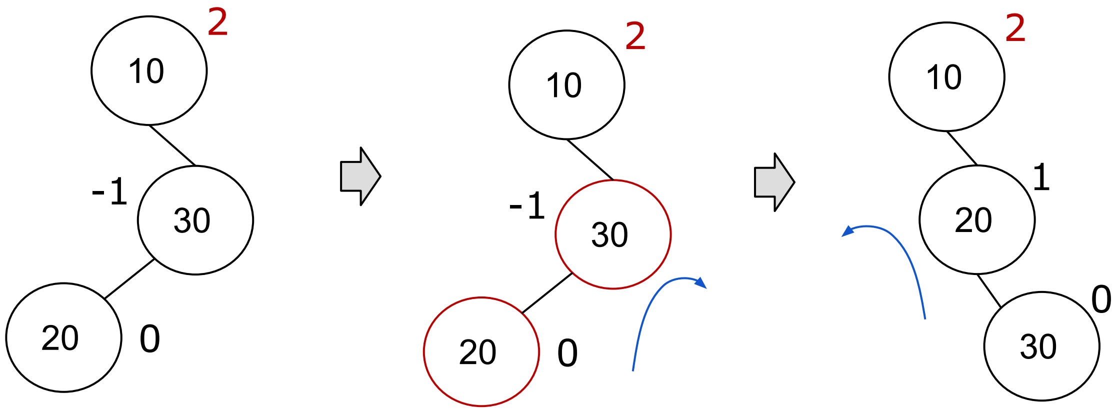
<10> is a unbalanced node



first perform right rotation on
right subtree of <30>

AVL Algorithm: Double Rotation

Right-left Rotation



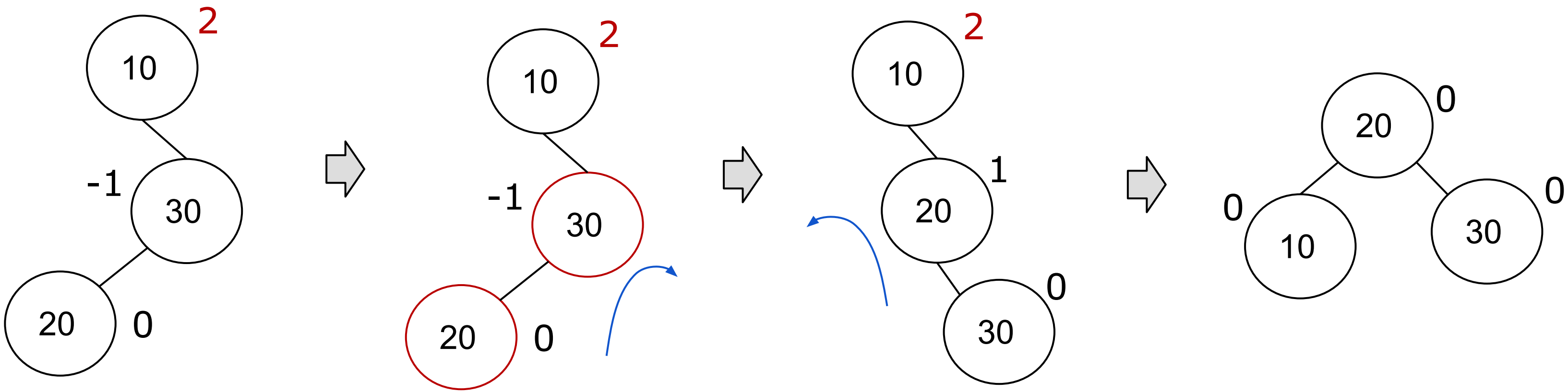
<10> is a unbalanced node

first perform right rotation on right subtree of <30>

perform left rotation

AVL Algorithm: Double Rotation

Right-left Rotation



<10> is a unbalanced node

first perform right rotation on right subtree of <30>

perform left rotation

get a balanced tree

Summary of AVL Tree

- height of AVL tree always be very close to the theoretical minimum of $\lceil \log 2(n + 1) \rceil$ (n: node numbers)
- more difficult to implement compare to 2-3-4 or red-black tree
- knowing how to count balance factor and perform different rotations to get a new balanced tree