CSCI 104
2-3 Trees

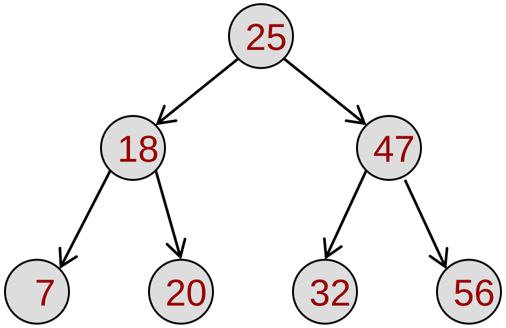
Mark Redekopp
David Kempe

Properties, Insertion and Removal

BINARY SEARCH TREES

Binary Search Tree

- Binary search tree = binary tree where all nodes meet the property that:
  - All values of nodes in left subtree are less-than or equal than the parent's value
  - All values of nodes in right subtree are greater-than or equal than the parent's value



If we wanted to print the values in sorted order would you use an pre-order, in-order, or post-order traversal?

BST Insertion

- Important: To be efficient (useful) we need to keep the binary search tree balanced
- Practice: Build a BST from the data values below
  - To insert an item walk the tree (go left if value is less than node, right if greater than node) until you find an empty location, at which point you insert the new value

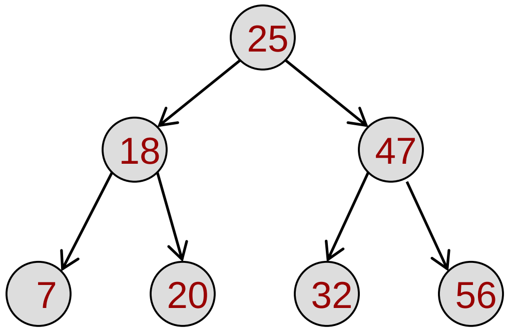
Insertion Order: 25, 18, 47, 7, 20, 32, 56

Insertion Order: 7, 18, 20, 25, 32, 47, 56

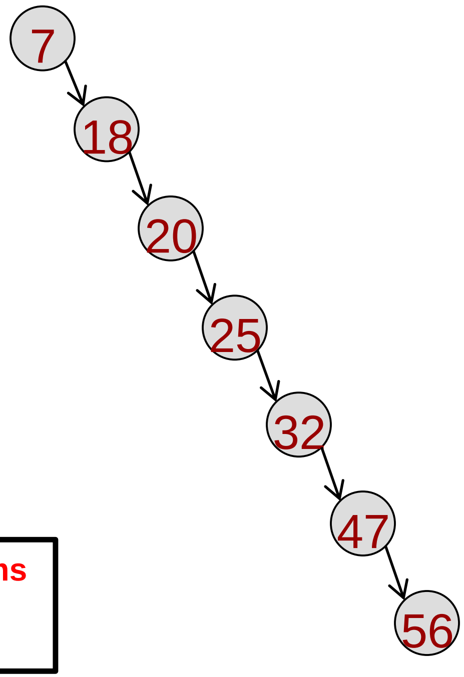
BST Insertion

- Important: To be efficient (useful) we need to keep the binary search tree balanced
- Practice: Build a BST from the data values below
  - To insert an item walk the tree (go left if value is less than node, right if greater than node) until you find an empty location, at which point you insert the new value

Insertion Order: 25, 18, 47, 7, 20, 32, 56



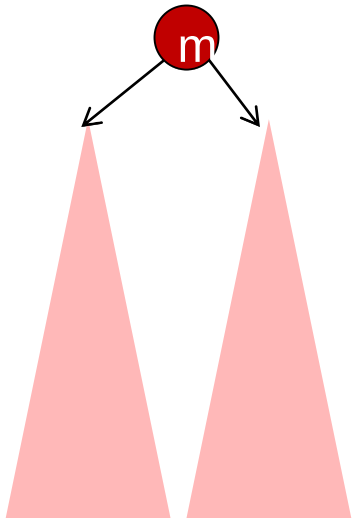
Insertion Order: 7, 18, 20, 25, 32, 47, 56



A major topic we will talk about is algorithms to keep a BST balanced as we do insertions/removals

Successors & Predecessors

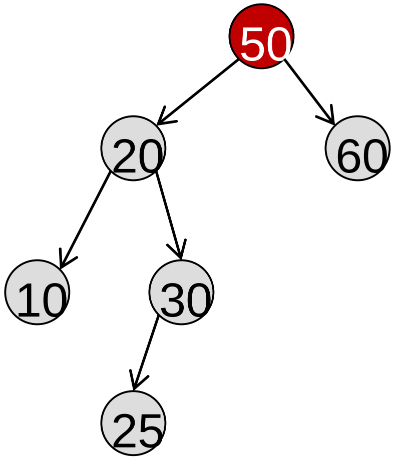
- Let's take a quick tangent that will help us understand how to do **BST Removal**
- Given a node in a BST
  - Its predecessor is defined as the next smallest value in the tree
  - Its successor is defined as the next biggest value in the tree
- Where would you expect to find a node's successor?
- Where would find a node's predecessor?



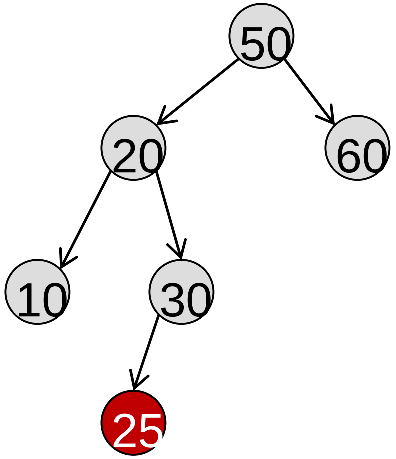
Predecessors

- If left child exists, predecessor is the right most node of the left subtree
- Else walk up the ancestor chain until you traverse the first right child pointer (find the first node who is a right child of his parent…that parent is the predecessor)
  - If you get to the root w/o finding a node who is a right child, there is no predecessor

Pred(50)



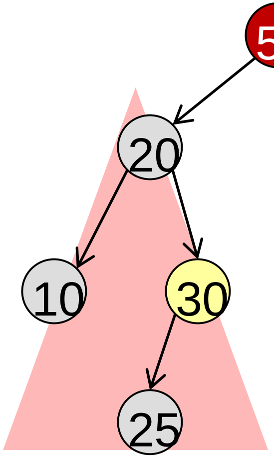
Pred(25)



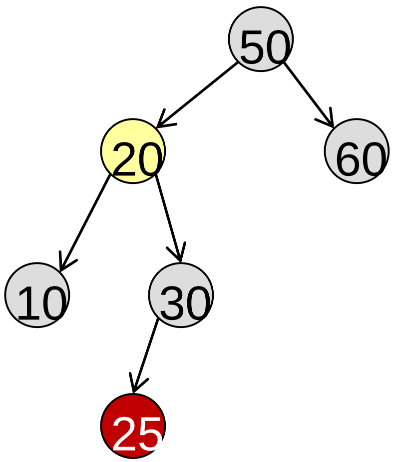
Predecessors

- If left child exists, predecessor is the right most node of the left subtree
- Else walk up the ancestor chain until you traverse the first right child pointer (find the first node who is a right child of his parent…that parent is the predecessor)
  - If you get to the root w/o finding a node who is a right child, there is no predecessor

Pred(50) = 30

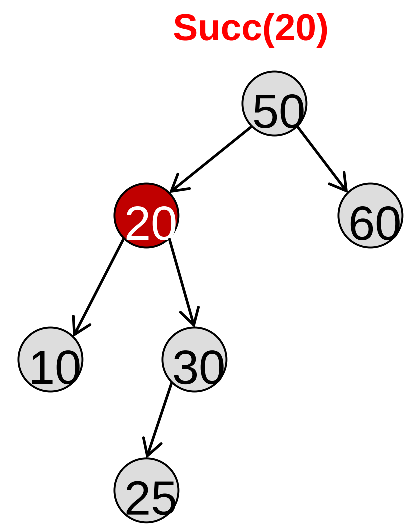


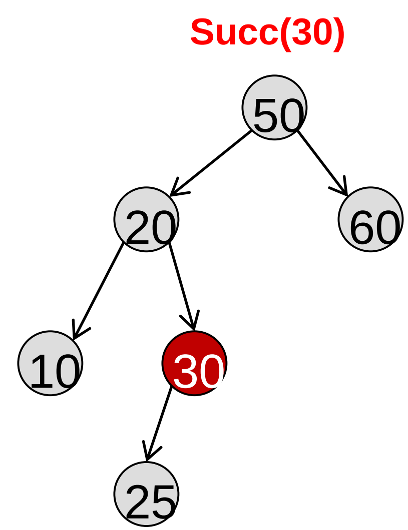
Pred(25)=20



Successors

- If right child exists, successor is the left most node of the right subtree
- Else walk up the ancestor chain until you traverse the first left child pointer (find the first node who is a left child of his parent…that parent is the successor)
  - If you get to the root w/o finding a node who is a left child, there is no successor

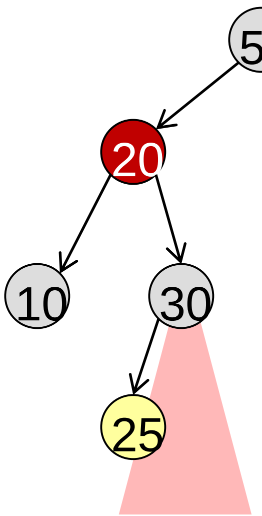




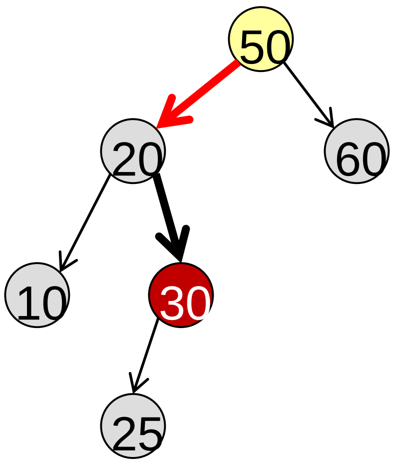
Successors

- If right child exists, successor is the left most node of the right subtree
- Else walk up the ancestor chain until you traverse the first left child pointer (find the first node who is a left child of his parent…that parent is the successor)
  - If you get to the root w/o finding a node who is a left child, there is no successor

Succ(20) = 25

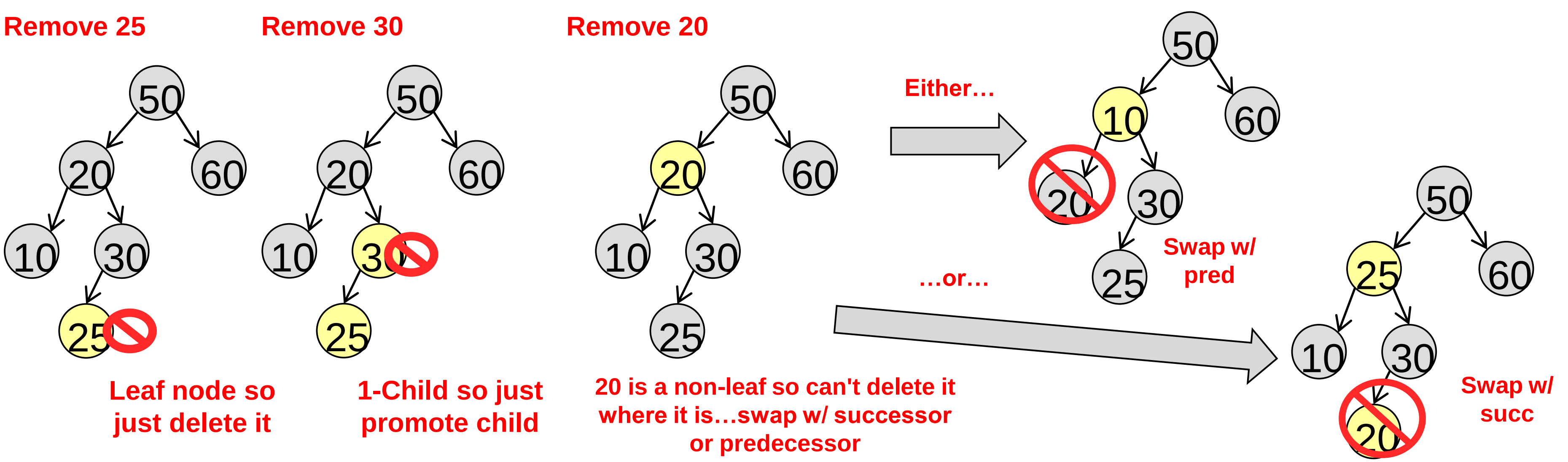


Succ(30)=50



BST Removal

- To remove a value from a BST…
  - First find the value to remove by walking the tree
  - If the value is in a leaf node, simply remove that leaf node
  - If the value is in a non-leaf node, swap the value with its in-order successor or predecessor and then remove the value
    - A non-leaf node's successor or predecessor is guaranteed to be a leaf node (which we can remove) or have 1 child which can be promoted
    - We can maintain the BST properties by putting a value's successor or predecessor in its place



Worst Case BST Efficiency

- Insertion
  - Balanced: ___________
  - Unbalanced: ___________
- Removal
  - Balanced: __________
  - Unbalanced: ___________
- Find/Search
  - Balanced: ____________
  - Unbalanced: ____________

```
#include<iostream>
using namespace std;

// Bin. Search Tree
template <typename T>
class BST
{
 public:
 BTree();
 ~BTree();
 virtual bool empty() = 0;
 virtual void insert(const T& v) = 0;
 virtual void remove(const T& v) = 0;
 virtual T* find(const T& v) = 0;
};
```

BST Efficiency

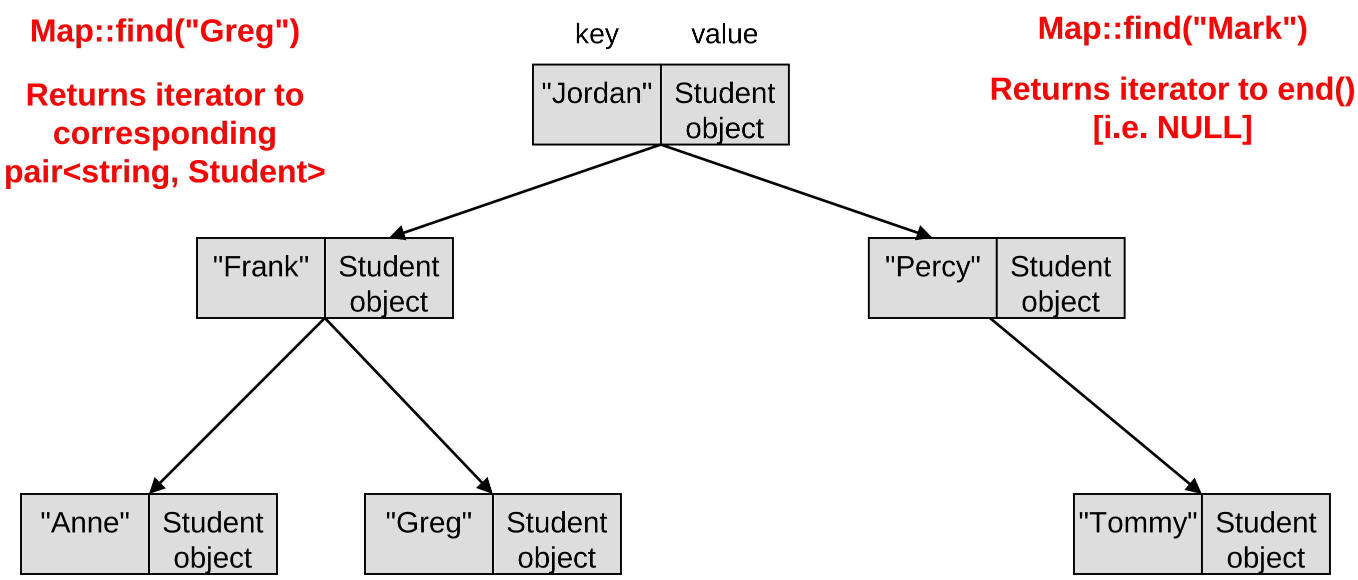
- Insertion
  - Balanced: O(log n)
  - Unbalanced: O(n)
- Removal
  - Balanced : O(log n)
  - Unbalanced: O(n)
- Find/Search
  - Balanced : O(log n)
  - Unbalanced: O(n)

```
#include<iostream>
using namespace std;

// Bin. Search Tree
template <typename T>
class BST
{
 public:
 BTree();
 ~BTree();
 virtual bool empty() = 0;
 virtual void insert(const T& v) = 0;
 virtual void remove(const T& v) = 0;
 virtual T* find(const T& v) = 0;
};
```

Trees & Maps/Sets

- C++ STL "maps" and "sets" use binary search trees internally to store their keys (and values) that can grow or contract as needed
- This allows O(log n) time to find/check membership
  - BUT ONLY if we keep the tree balanced!



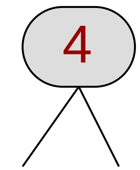
An example of B-Trees

2-3 TREES

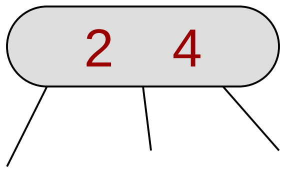
Definition

- 2-3 Tree is a tree where
  - Non-leaf nodes have 1 value & 2 children or 2 values and 3 children
  - All leaves are at the same level
- Following the line of reasoning…
  - All leaves at the same level with internal nodes having at least 2 children implies a (**full / complete**) tree
    - FULL (Recall complete just means the lower level is filled left to right but not full)
  - A full tree with n nodes implies…
    - Height that is bounded by $\log_2(n)$

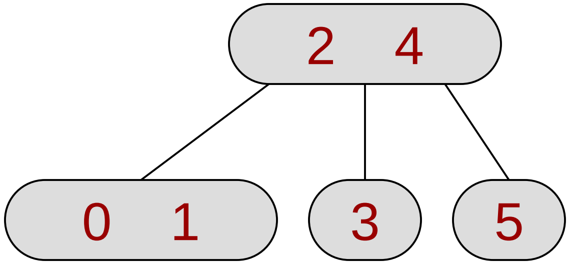
a 2 Node



a 3 Node



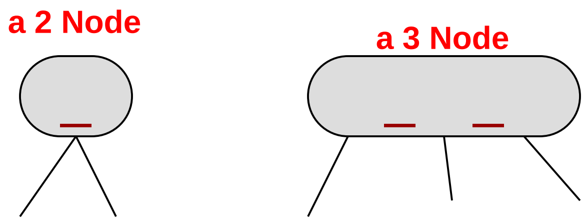
a valid 2-3 tree



Implementation of 2- & 3-Nodes

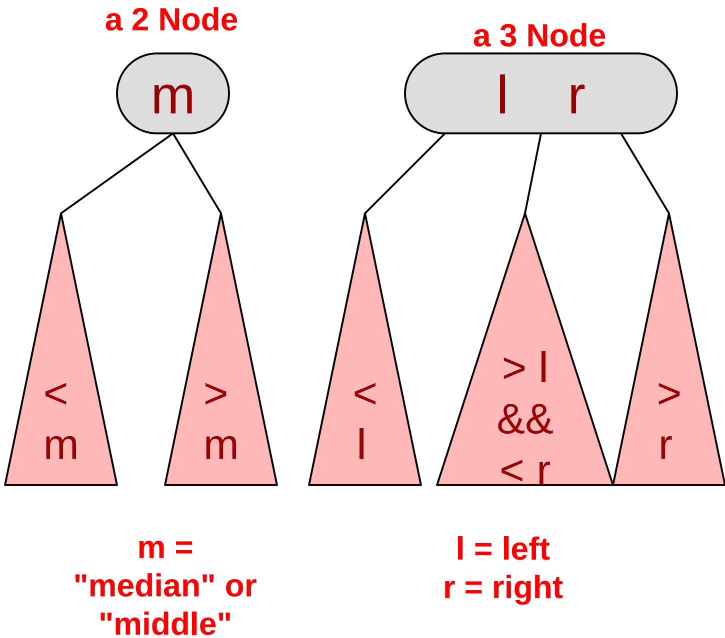
- You will see that at different times 2 nodes may have to be upgraded to 3 nodes
- To model these nodes we plan for the worst case…a 3 node
- This requires wasted storage for 2 nodes

```
template <typename T>
struct Item23 {
  T val1;
  T val2;
  Item23<T>* left;
  Item23<T>* mid;
  Item23<T>* right;
  bool twoNode;
};
```



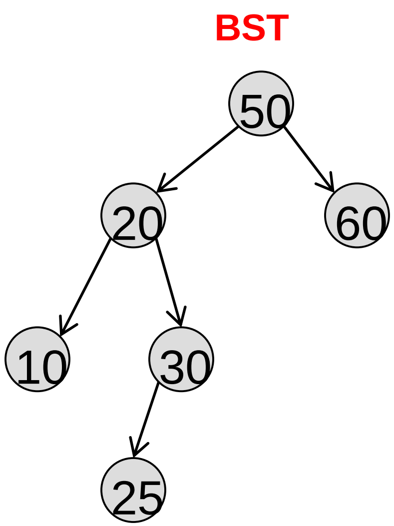
2-3 Search Trees

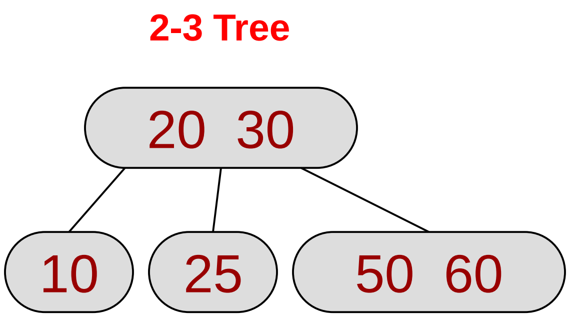
- Similar properties as a BST
- 2-3 Search Tree
  - If a 2 Node with value, *m*
    - Left subtree nodes are < node value
    - Right subtree nodes are > node value
  - If a 3 Node with value, *l* and *r*
    - Left subtree nodes are < *l*
    - Middle subtree > *l* and < *r*
    - Right subtree nodes are > *r*
- 2-3 Trees are almost always used as search trees, so from now on if we say 2-3 tree we mean 2-3 search tree



2-3 Search Tree

- Binary search tree compared to 2-3 tree
- Check if 55 is in the tree?

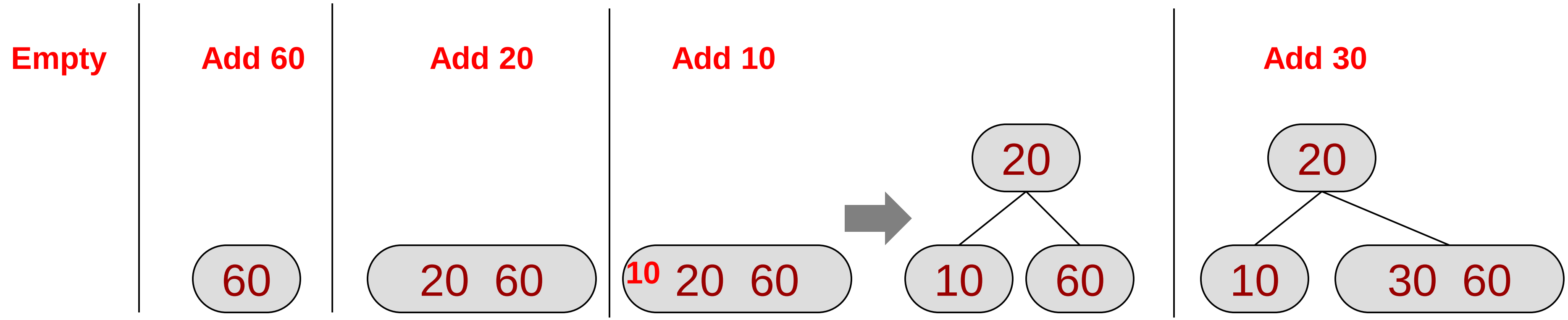




2-3 Insertion Algorithm

- Key: Since all leaves must be at the same level ("leaves always have their feet on the ground"), insertion causes the tree to "grow upward"
- To insert a value,
  - 1. walk the tree to a leaf using your search approach
  - 2a. If the leaf is a 2-node (i.e.1 value), add the new value to that node
  - 2b. Else break the 3-node into two 2-nodes with the smallest value as the left, biggest as the right, and median value promoted to the parent with smallest and biggest node added as children of the parent
  - Repeat step 2(a or b) for the parent
- Insert 60, 20, 10, 30, 25, 50, 80

Key: Any time a node accumulates 3 values, split it into single valued nodes (i.e. 2-nodes) and promote the median

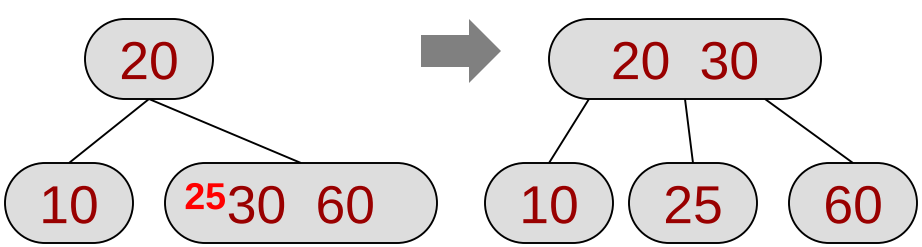


2-3 Insertion Algorithm

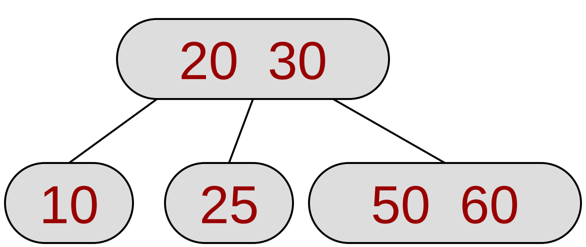
- Key: Since all leaves must be at the same level ("leaves always have their feet on the ground"), insertion causes the tree to "grow upward"
- To insert a value,
  - 1. walk the tree to a leaf using your search approach
  - 2a. If the leaf is a 2-node (i.e.1 value), add the new value to that node
  - 2b. Else break the 3-node into two 2-nodes with the smallest value as the left, biggest as the right, and median value promoted to the parent with smallest and biggest node added as children of the parent
  - Repeat step 2(a or b) for the parent
- Insert 60, 20, 10, 30, 25, 50, 80

Key: Any time a node accumulates 3 values, split it into single valued nodes (i.e. 2-nodes) and promote the median

Add 25



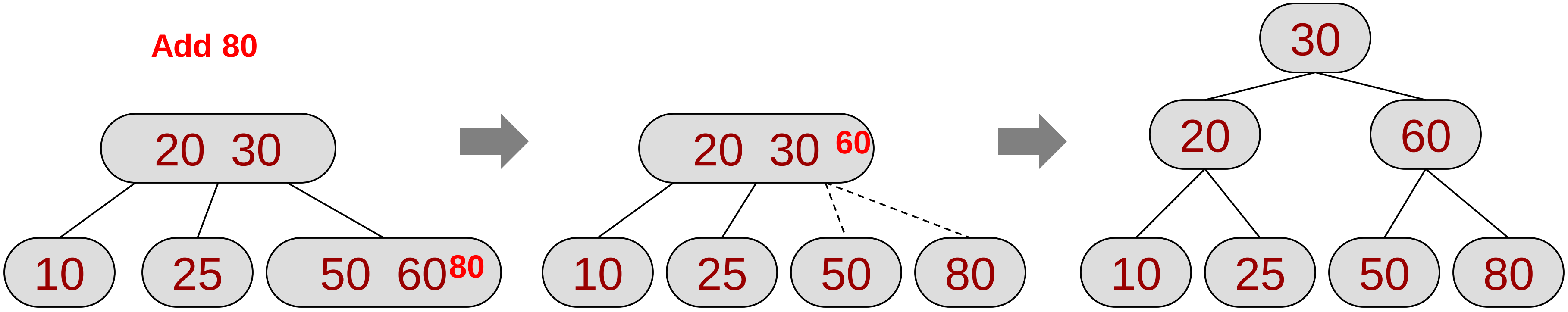
Add 50



2-3 Insertion Algorithm

- Key: Since all leaves must be at the same level ("leaves always have their feet on the ground"), insertion causes the tree to "grow upward"
- To insert a value,
  - 1. walk the tree to a leaf using your search approach
  - 2a. If the leaf is a 2-node (i.e.1 value), add the new value to that node
  - 2b. Else break the 3-node into two 2 nodes with the smallest value as the left, biggest as the right, and median value promoted to the parent with smallest and biggest node added as children of the parent
  - Repeat step 2(a or b) for the parent
- Insert 60, 20, 10, 30, 25, 50, 80

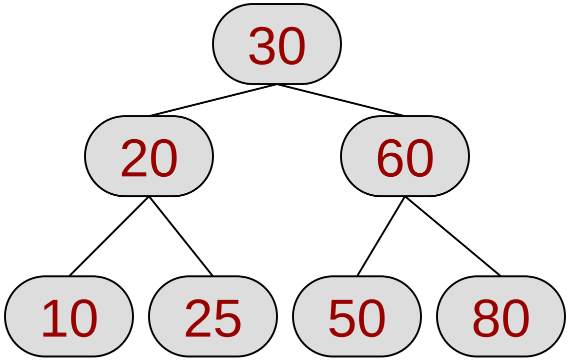
Key: Any time a node accumulates 3 values, split it into single valued nodes (i.e. 2-nodes) and promote the median



2-3 Insertion Algorithm

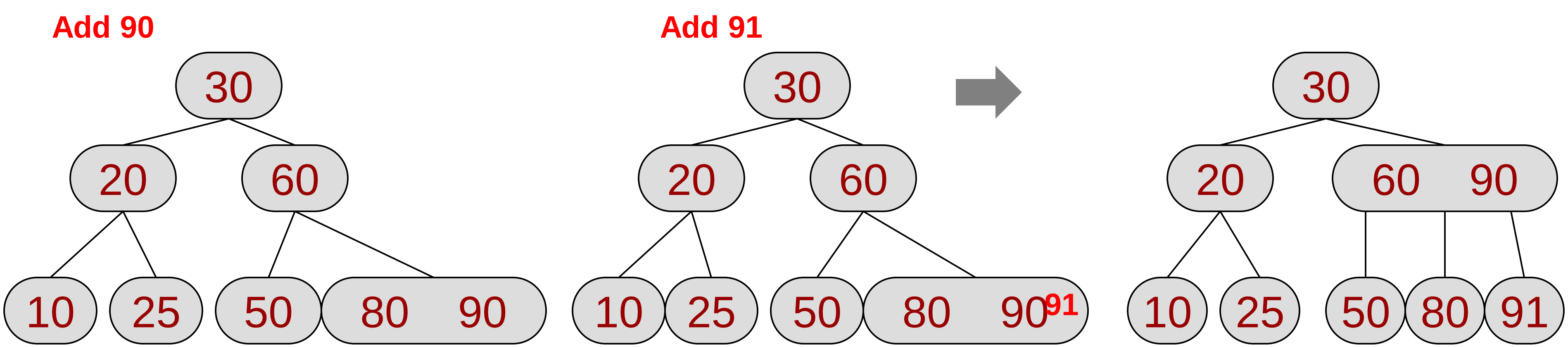
- Key: Since all leaves must be at the same level ("leaves always have their feet on the ground"), insertion causes the tree to "grow upward"
- To insert a value,
  - 1. walk the tree to a leaf using your search approach
  - 2a. If the leaf is a 2-node (i.e.1 value), add the new value to that node
  - 2b. Else break the 3-node into two 2 nodes with the smallest value as the left, biggest as the right, and median value promoted to the parent with smallest and biggest node added as children of the parent
  - Repeat step 2(a or b) for the parent
- Insert 90,91,92, 93

Add 90



2-3 Insertion Algorithm

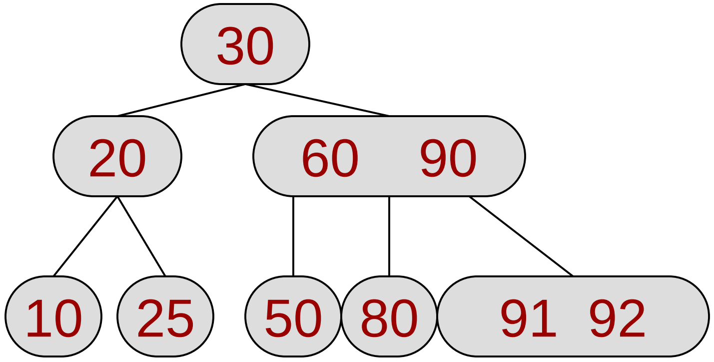
- Key: Since all leaves must be at the same level, insertion causes the tree to "grow upward"
- To insert a value,
  - 1. walk the tree to a leaf using your search approach
  - 2a. If the leaf is a 2-node (i.e.1 value), add the new value to that node
  - 2b. Else break the 3-node into two 2 nodes with the smallest value as the left, biggest as the right, and median value should be promoted to the parent with smallest and biggest node added as children of the parent
  - Repeat step 2(a or b) for the parent
- Insert 90,91,92,93



2-3 Insertion Algorithm

- Key: Since all leaves must be at the same level, insertion causes the tree to "grow upward"
- To insert a value,
  - 1. walk the tree to a leaf using your search approach
  - 2a. If the leaf is a 2-node (i.e.1 value), add the new value to that node
  - 2b. Else break the 3-node into two 2 nodes with the smallest value as the left, biggest as the right, and median value should be promoted to the parent with smallest and biggest node added as children of the parent
  - Repeat step 2(a or b) for the parent
- Insert 90,91,92,93

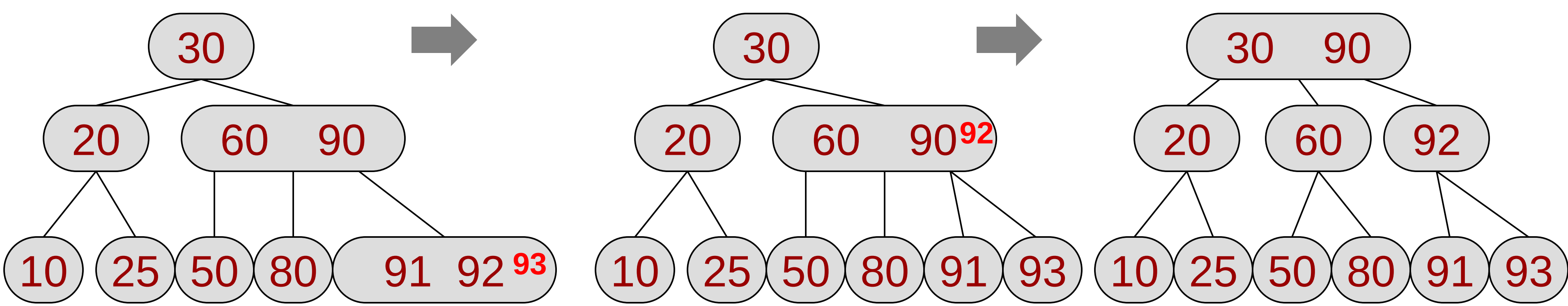
Add 92



2-3 Insertion Algorithm

- Key: Since all leaves must be at the same level, insertion causes the tree to "grow upward"
- To insert a value,
  - 1. walk the tree to a leaf using your search approach
  - 2a. If the leaf is a 2-node (i.e.1 value), add the new value to that node
  - 2b. Else break the 3-node into two 2 nodes with the smallest value as the left, biggest as the right, and median value should be promoted to the parent with smallest and biggest node added as children of the parent
  - Repeat step 2(a or b) for the parent
- Insert 90,91,92,93

Add 93



2-3 Tree Removal

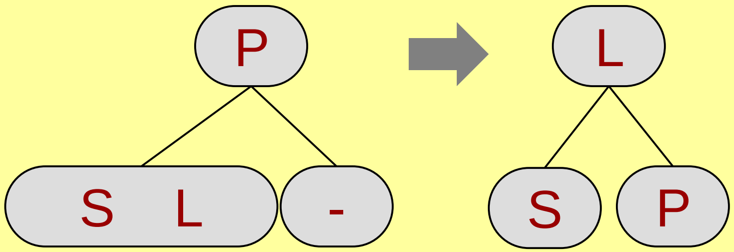
- Key: 2-3 Trees must remain "full" (leaf nodes all at the same level)
- Remove
  - 1. Find data item to remove
  - 2. If data item is not in a leaf node, find in-order successor (which is in a leaf node) and swap values (it's safe to put successor in your location)
  - 3. Remove item from the leaf node
  - 4. If leaf node is now empty, call fixTree(leafNode)

Another key: Want to get item to remove down to a leaf and then work up the tree

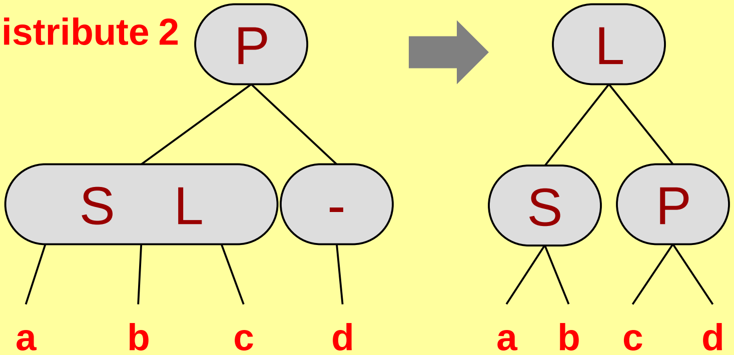
- fixTree(n)
  - If n is root, delete root and return
  - Let p be the parent of n
  - If a sibling of n has two items
    - Redistribute items between n, sibling, and p and move any appropriate child from sibling to n
  - Else
    - Choose a sibling, s, of n and bring an item from p into s redistributing any children of n to s
    - Remove node n
    - If parent is empty, fixTree(p)

Remove Cases

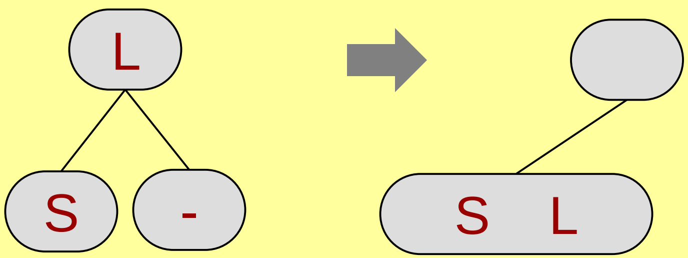
Redistribute 1



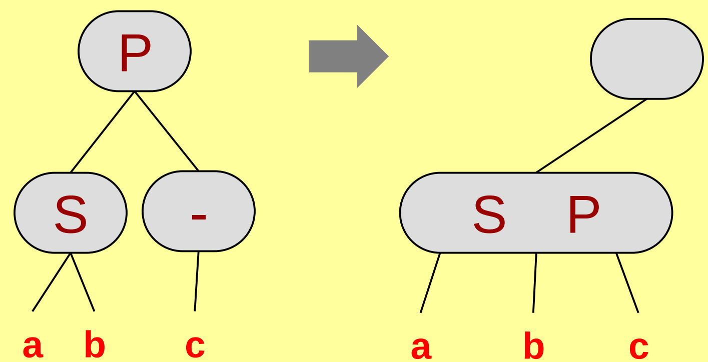
Redistribute 2



Merge 1

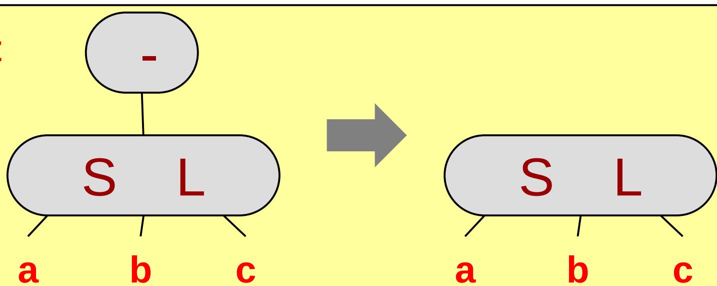


Merge 2



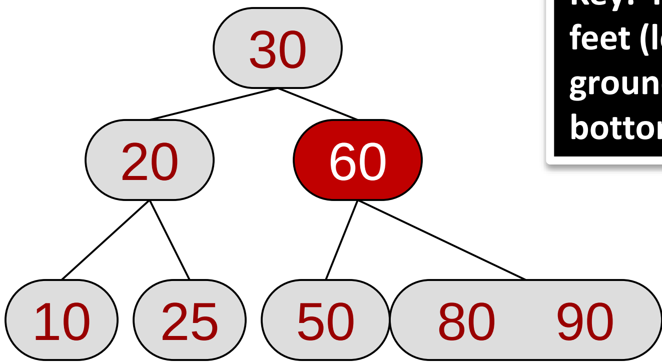
P = parent
S = smaller
L = larger

Empty root

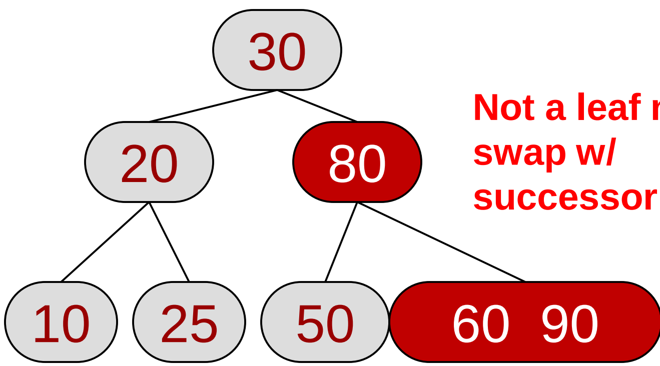


Remove Examples

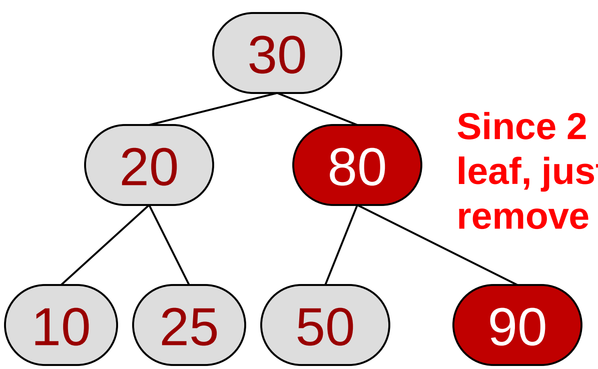
Remove 60



Key: Keep all your feet (leaves) on the ground (on the bottom row)

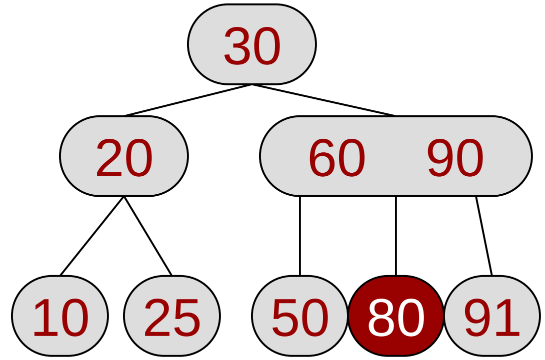


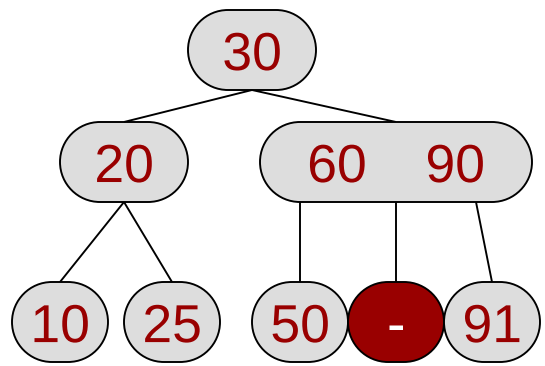
Not a leaf node so swap w/ successor at leaf



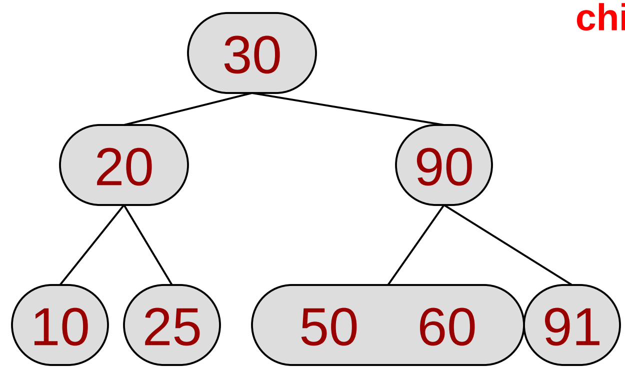
Since 2 items at leaf, just remove 60

Remove 80





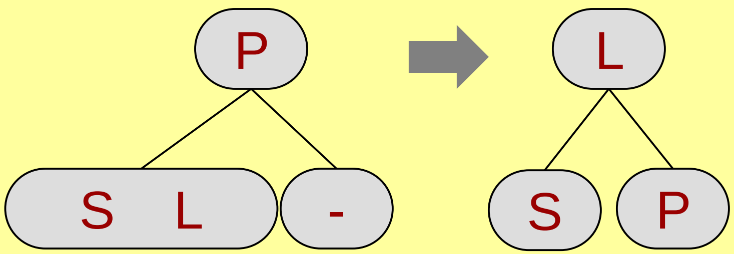
Can't just delete because a 3-node would have only 2 children



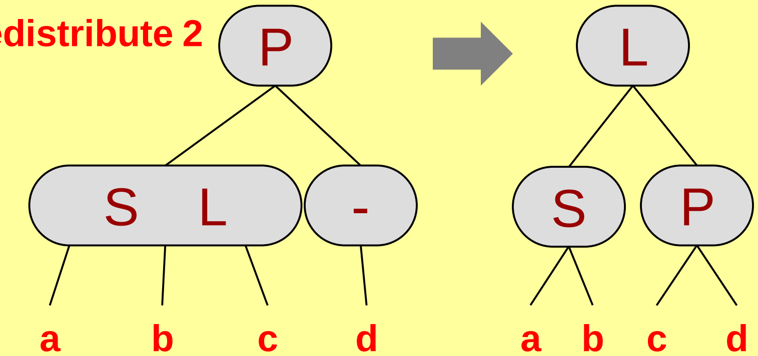
Rotate 60 down into 50 to make a 3-node at the leaf and 2-node parent

Remove Cases

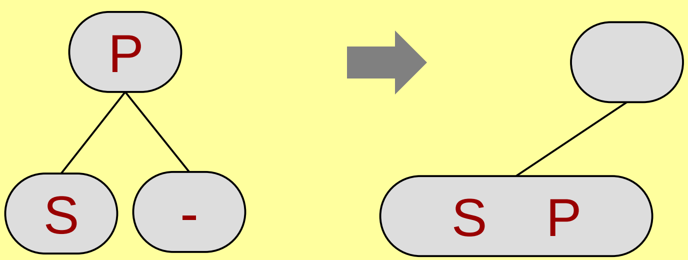
Redistribute 1



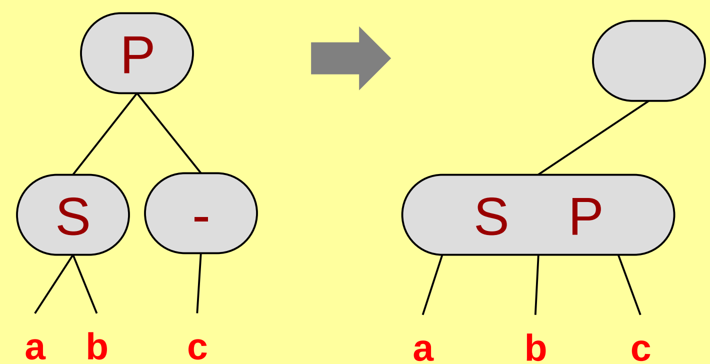
Redistribute 2



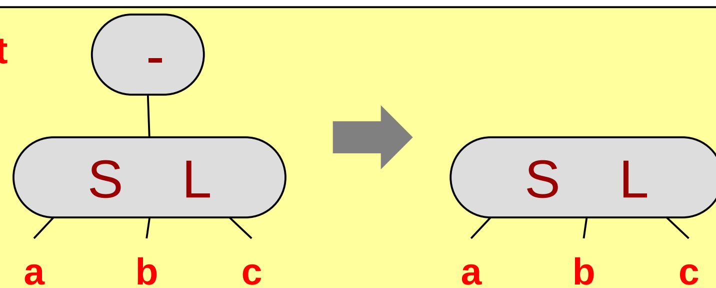
Merge 1



Merge 2

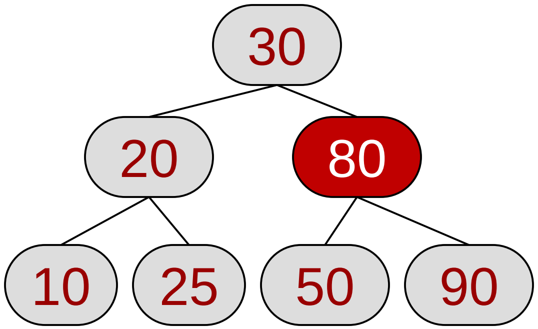


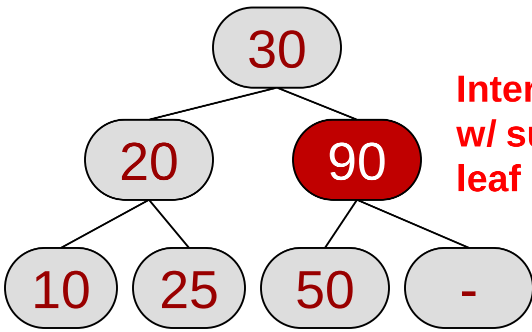
Empty root



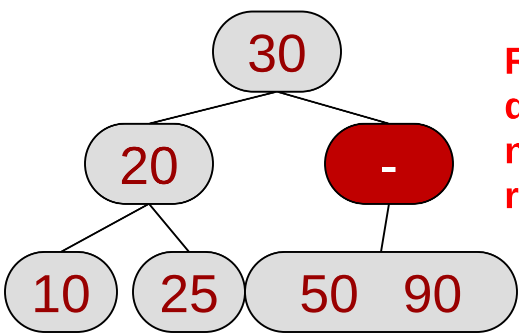
Remove Examples

Remove 80

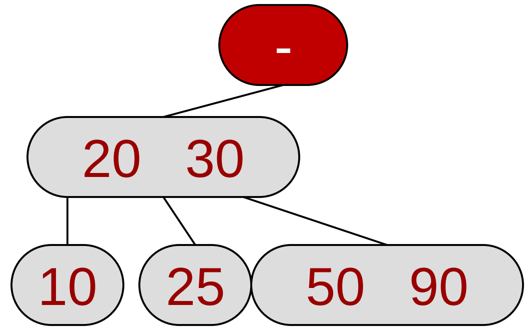




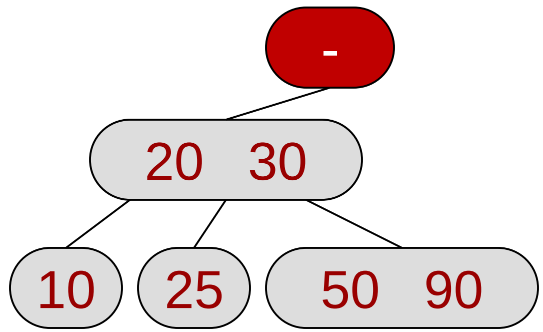
Internal so swap w/ successor at leaf



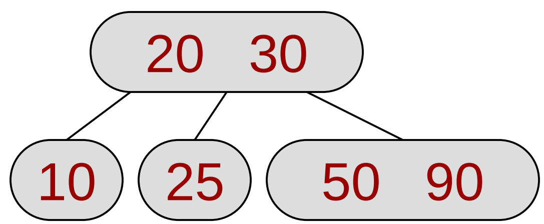
Rotate parent down and empty node up, then recurse



Rotate parent down and empty node up, then recurse

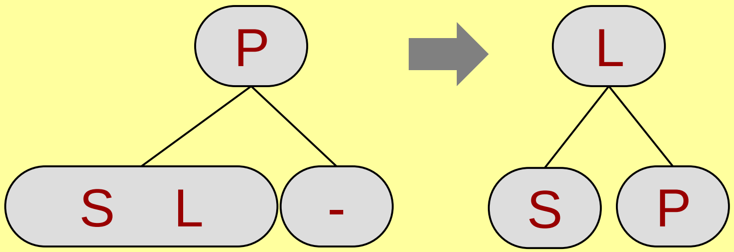


Remove root and thus height of tree decreases

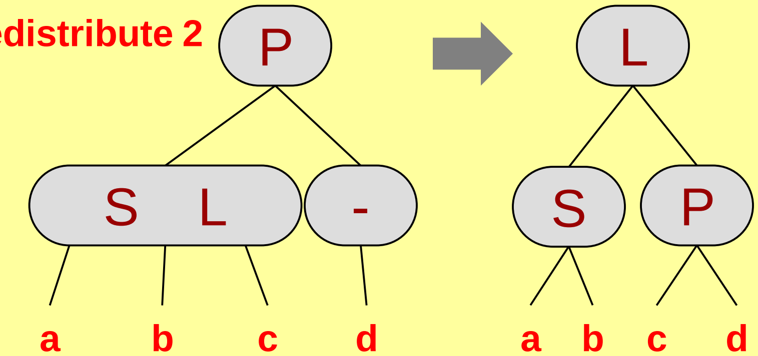


Remove Cases

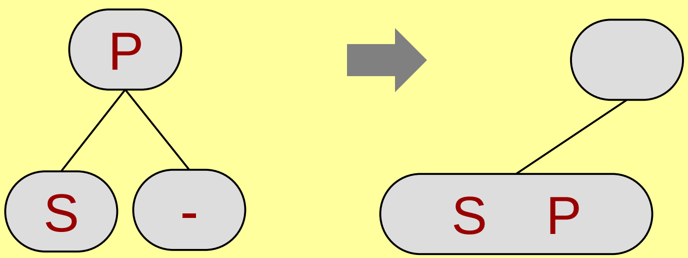
Redistribute 1



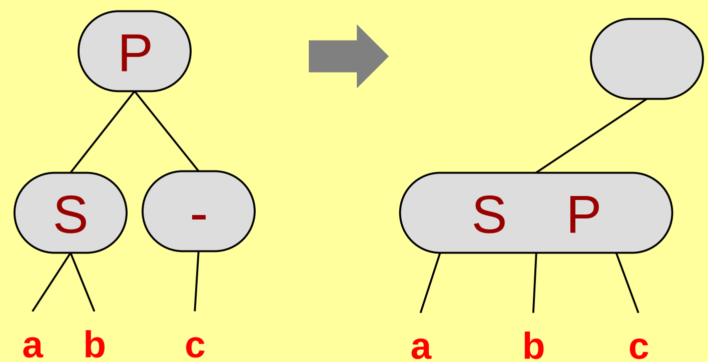
Redistribute 2



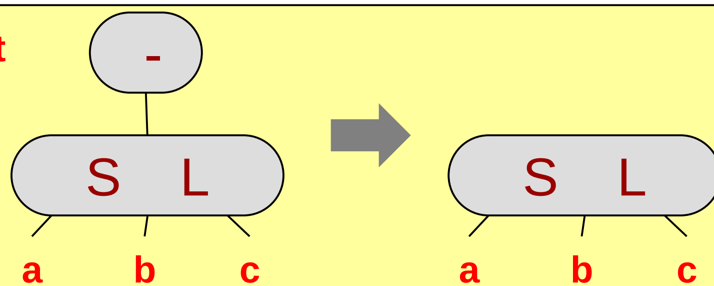
Merge 1



Merge 2

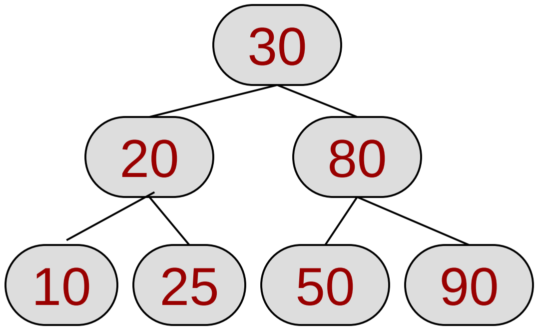


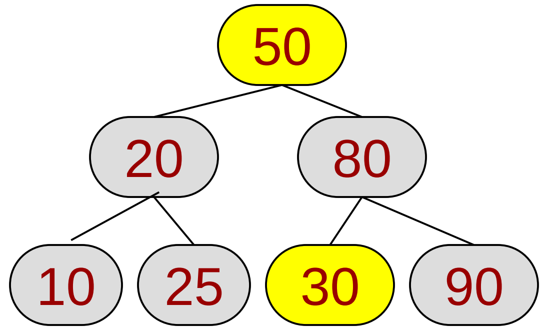
Empty root



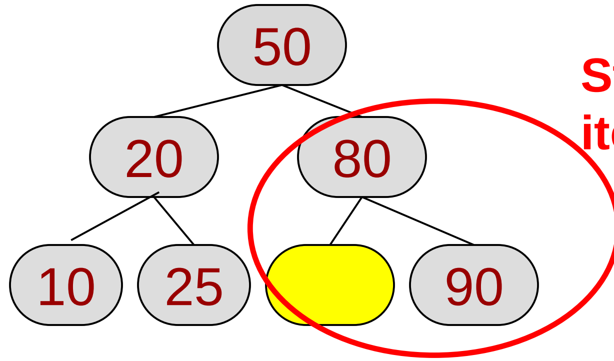
Remove Exercise 1

Remove 30

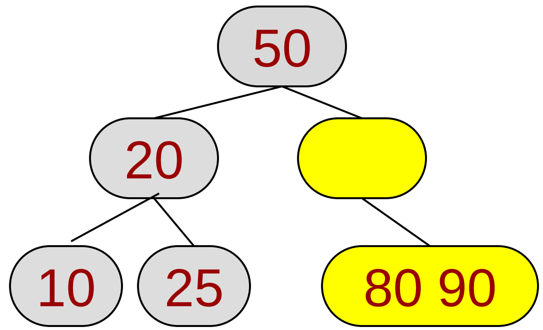




Step 1: Not a leaf, so swap with successor

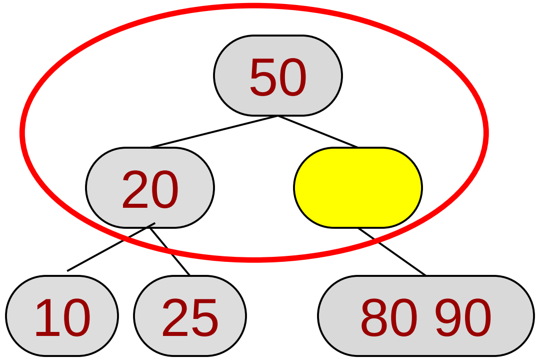


Step 2: Remove item from node

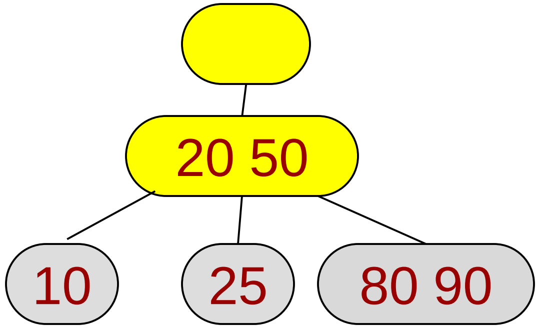


Step 3: Two values and 3 nodes, so merge. Must maintain levels.

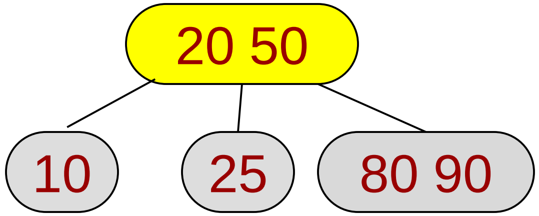
Remove Exercise 1 (cont.)



Start over with the empty parent. Do another merge



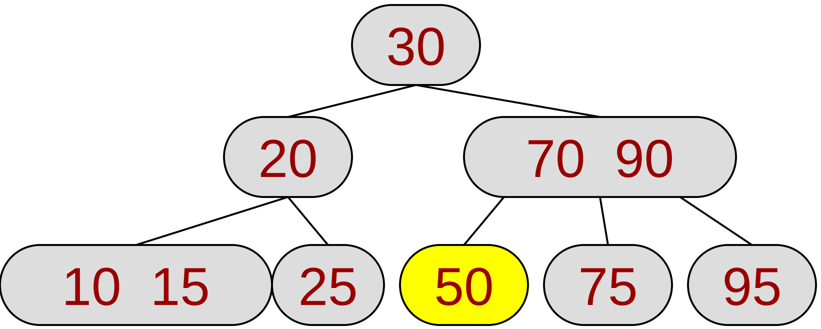
Step 4: Merge values



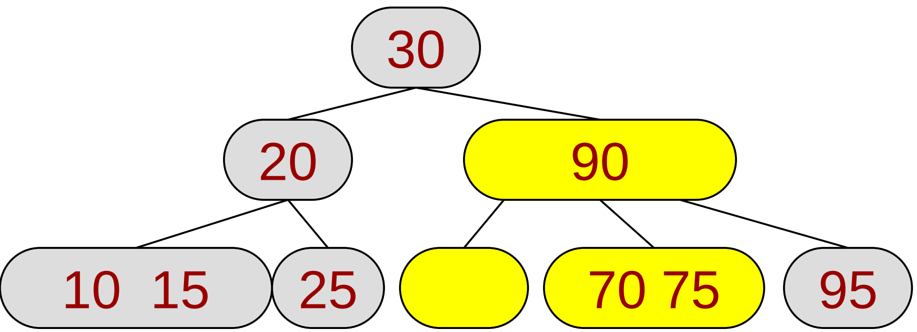
Step 5: Can delete the empty root node.

Remove Exercise 2

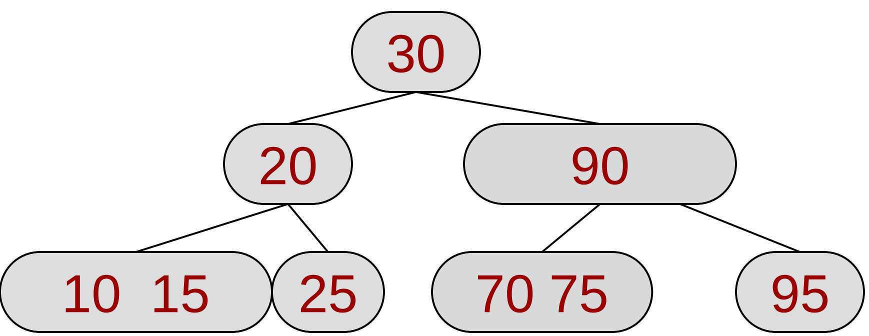
Remove 50



Step 1: It's a leaf node, so no need to find successor. Remove the item from node.

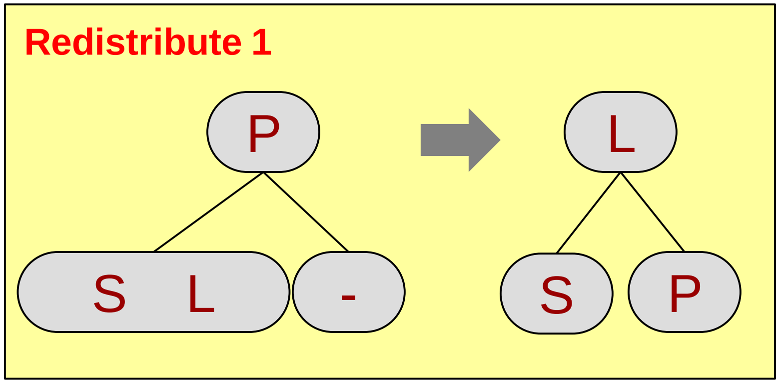


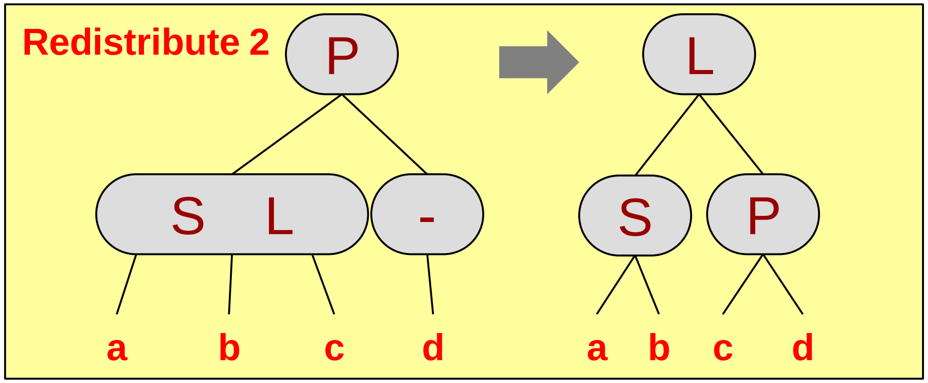
Step 2: Since no 3-node children, push a value of parent into a child.

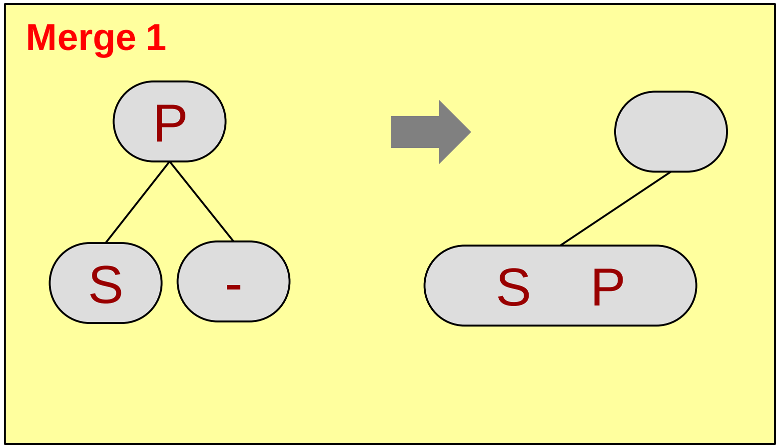


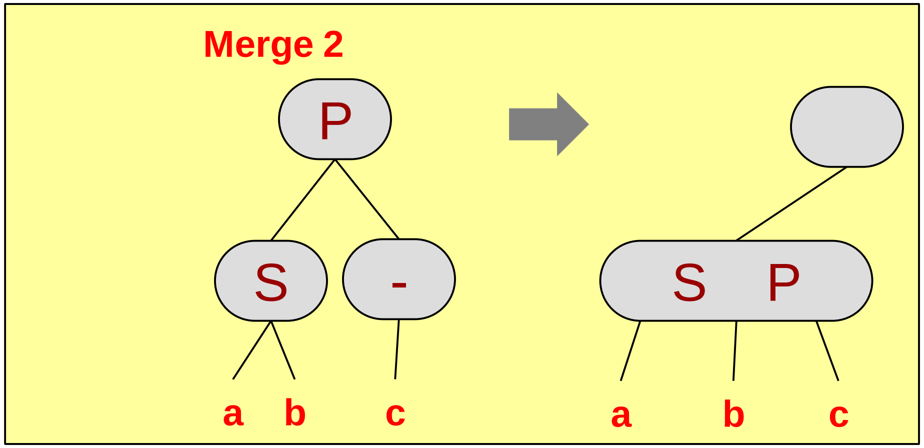
Step 3: Delete the node.

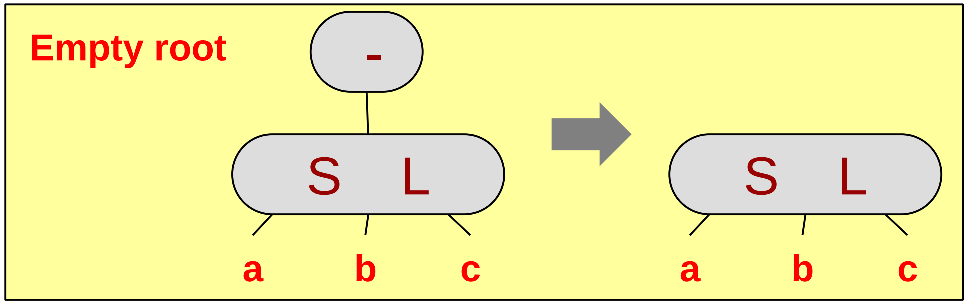
Remove Cases





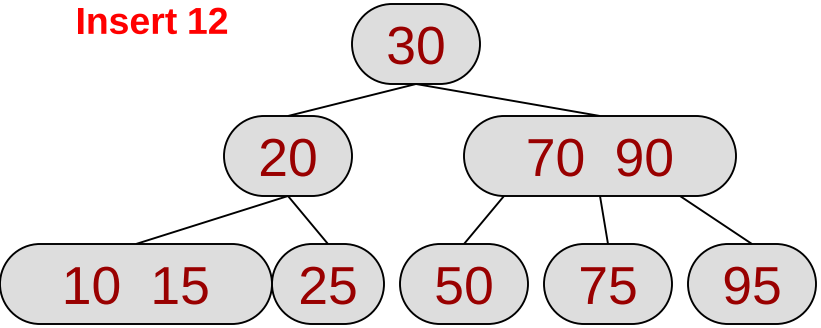






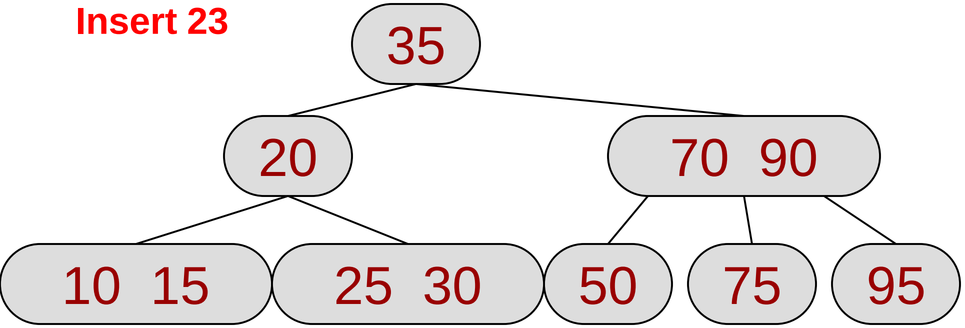
Insertion Exercise 1

Insert 12



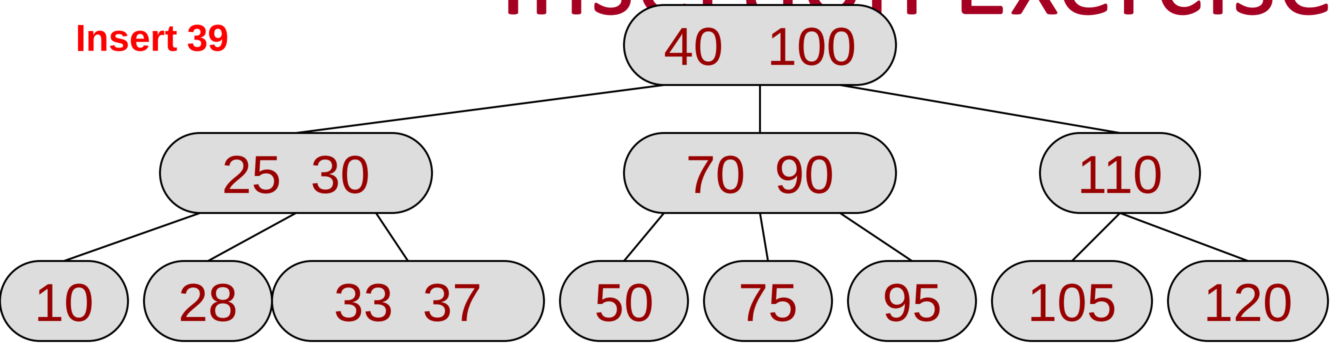
Insertion Exercise 2

Insert 23



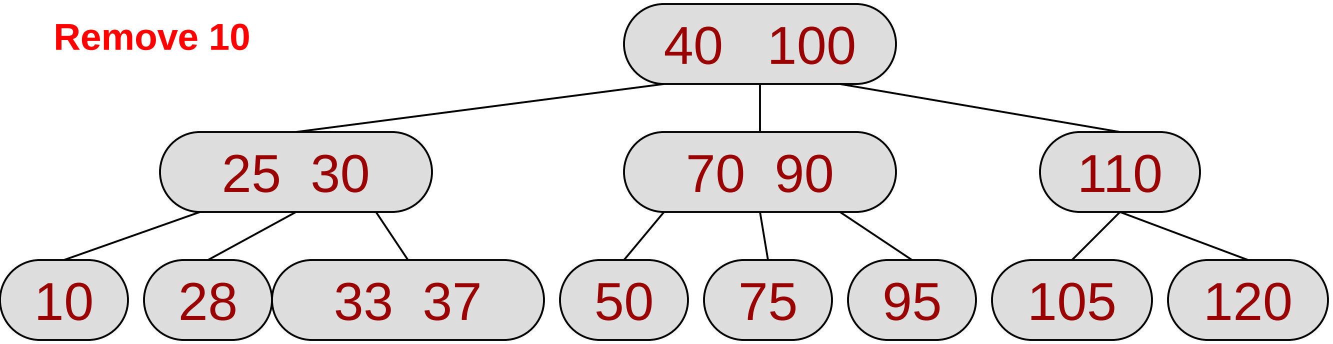
Insertion Exercise 3

Insert 39



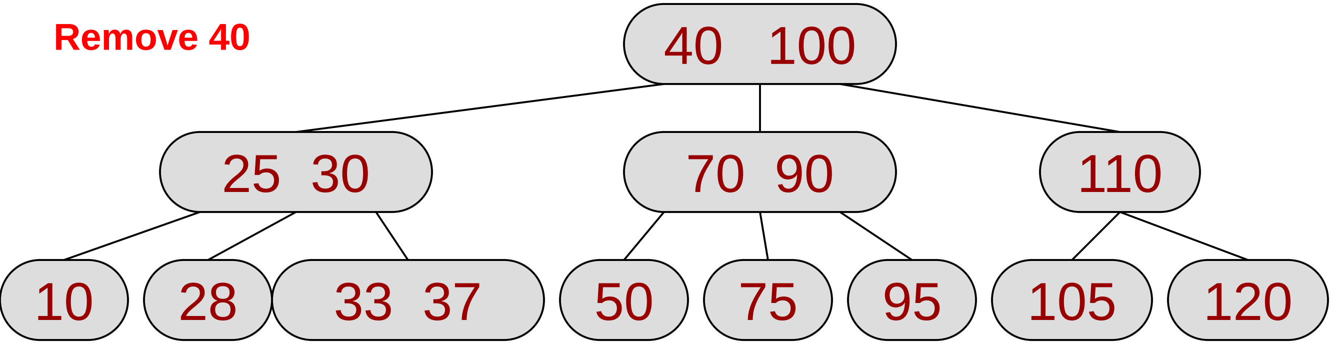
Removal Exercise 4

Remove 10



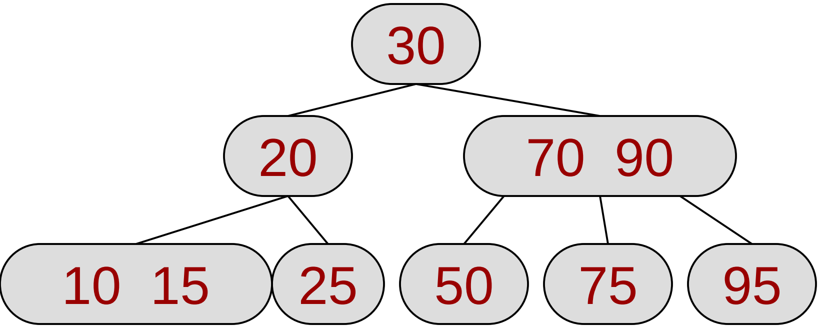
Removal Exercise 5

Remove 40



Remove Exercise 6

Remove 30



Other Resources

- http://www.cs.usfca.edu/~galles/visualization/BTree.html