CSCI 104

Rafael Ferreira da Silva

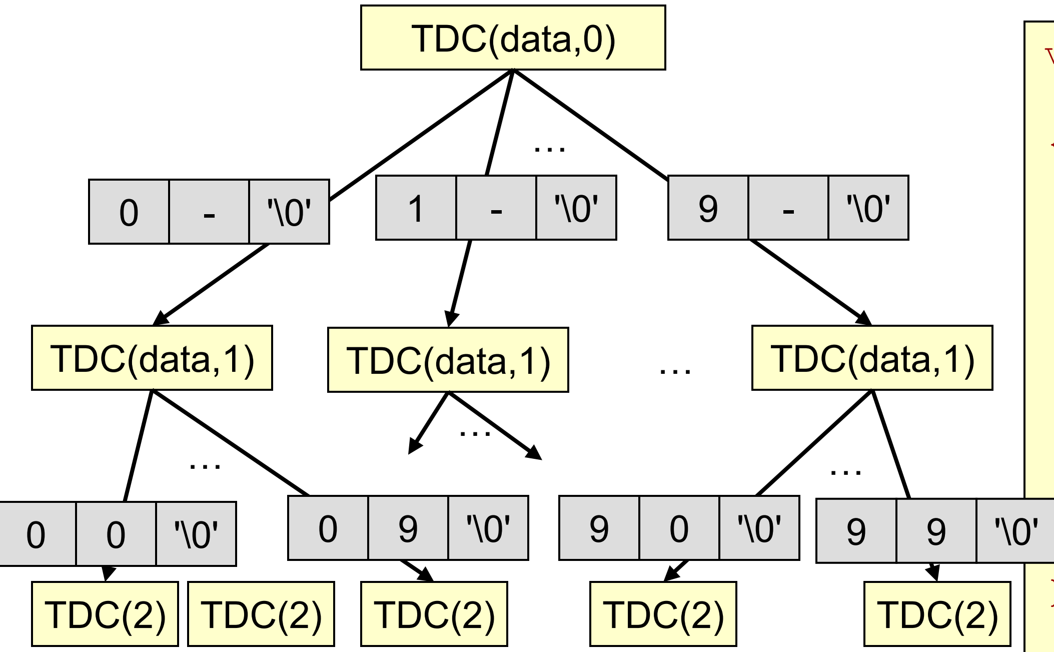
rafsilva@isi.edu

Slides adapted from: Mark Redekopp and David Kempe

BACKTRACK SEARCH ALGORITHMS

Generating All Combinations

- Recursion offers a simple way to generate all combinations of N items from a set of options, S
  - Example: Generate all 2-digit decimal numbers (N=2, S={0,1,…,9})




```
void TwoDigCombos(char data[3],
                  int curr)
{
  if(curr == 2 )
    cout << data;
  else {
    for(int i=0; i < 10; i++){
      // set to i
      data[curr] = '0'+i;
      // recurse
      TwoDigCombos(data, curr+1);
  }
}
```

Get the Code

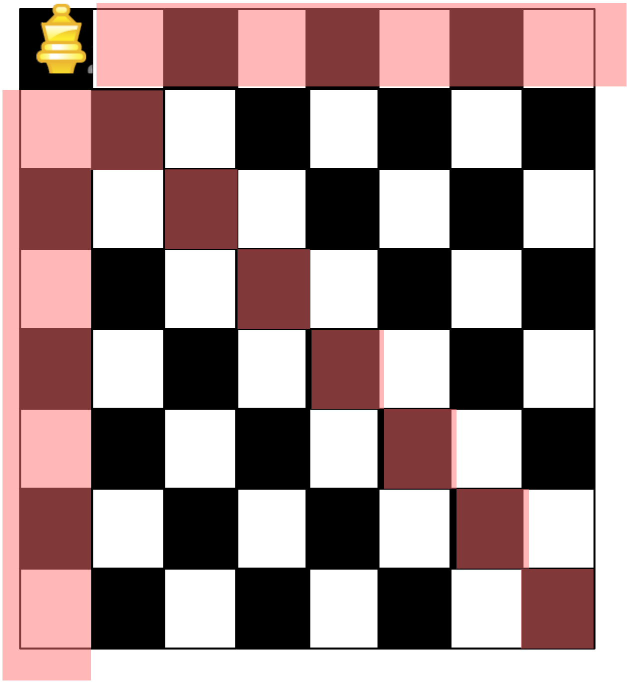
- In-class exercises
  - nqueens-allcombos
  - nqueens
- On your VM
  - $ mkdir nqueens
  - $ cd nqueens
  - $ wget http://ee.usc.edu/~redekopp/cs104/nqueens.tar
  - $ tar xvf nqueens.tar

Recursive Backtracking Search

- Recursion allows us to "easily" enumerate all solutions to some problem
- Backtracking algorithms…
  - Are often used to solve constraint satisfaction problem or optimization problems
    - Several items that can be set to 1 of N values under some constraints
  - Stop searching down a path at the first indication that constraints won't lead to a solution
- Some common and important problems can be solved with backtracking
- Knapsack problem
  - You have a set of objects with a given weight and value. Suppose you have a knapsack that can hold N pounds, which subset of objects can you pack that maximizes the value.
  - Example:
    - Knapsack can hold 35 pounds
    - Object A: 7 pounds, $12 ea. Object B: 10 pounds, $18 ea.
    - Object C: 4 pounds, $7 ea. Object D: 2.4 pounds, $4 ea.
- Other examples:
  - Map Coloring
  - Traveling Salesman Problem
  - Sudoku
  - N-Queens

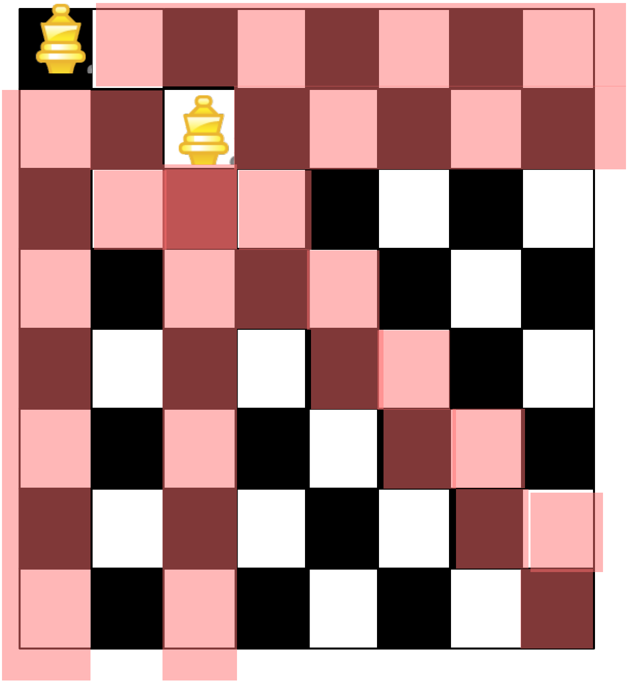
N-Queens Problem

- Problem: How to place N queens on an NxN chess board such that no queens may attack each other
- Fact: Queens can attack at any distance vertically, horizontally, or diagonally
- Observation: Different queen in each row and each column
- Backtrack search approach:
  - Place 1st queen in a viable option then, then try to place 2nd queen, etc.
  - If we reach a point where no queen can be placed in row i or we've exhausted all options in row i, then we return and change row i-1



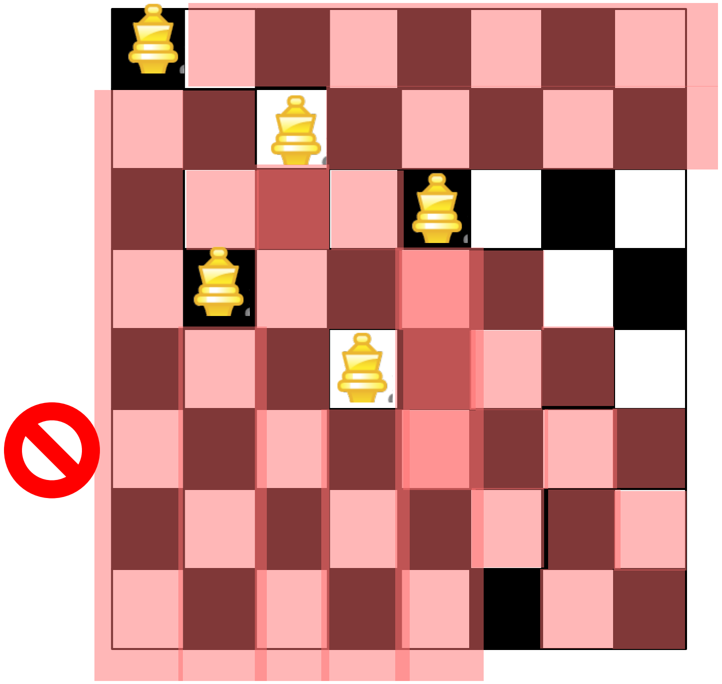
8x8 Example of N-Queens

- Now place 2[nd] queen



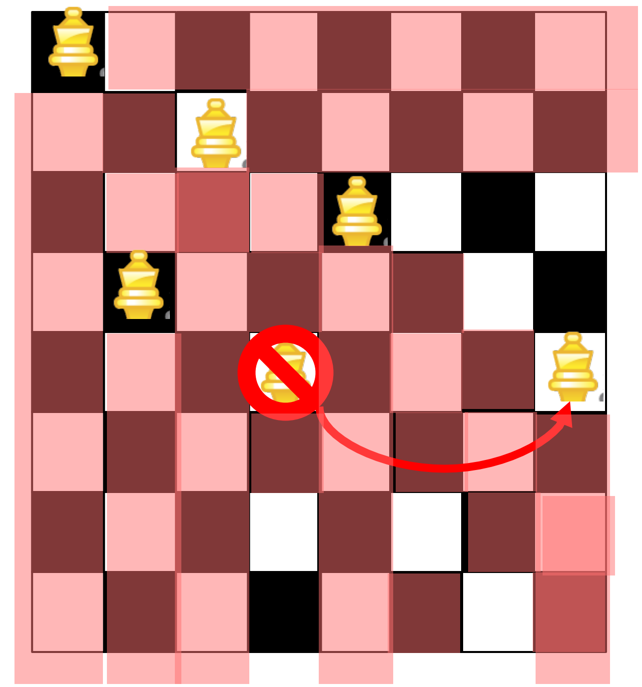
8x8 Example of N-Queens

- Now place others as viable
- After this configuration here, there are no locations in row 6 that are not under attack from the previous 5
- BACKTRACK!!!



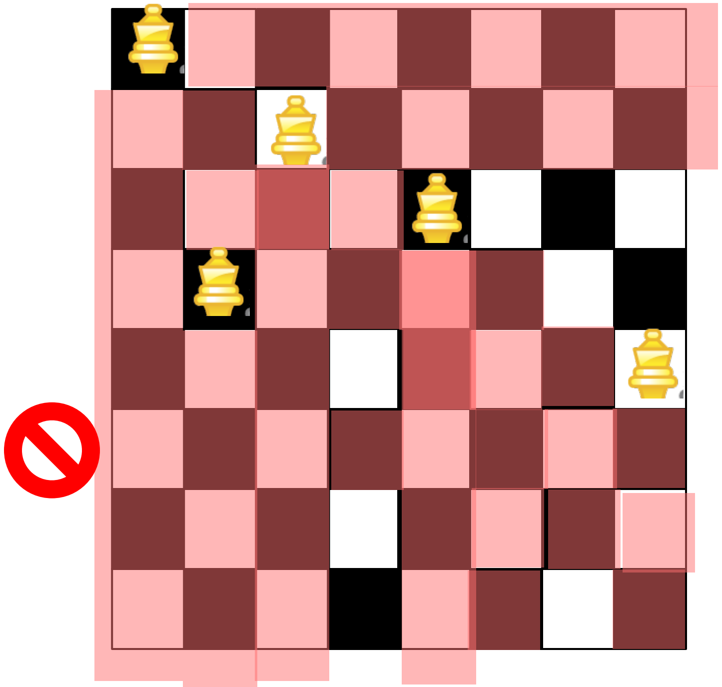
8x8 Example of N-Queens

- Now place others as viable
- After this configuration here, there are no locations in row 6 that is not under attack from the previous 5
- So go back to row 5 and switch assignment to next viable option and progress back to row 6



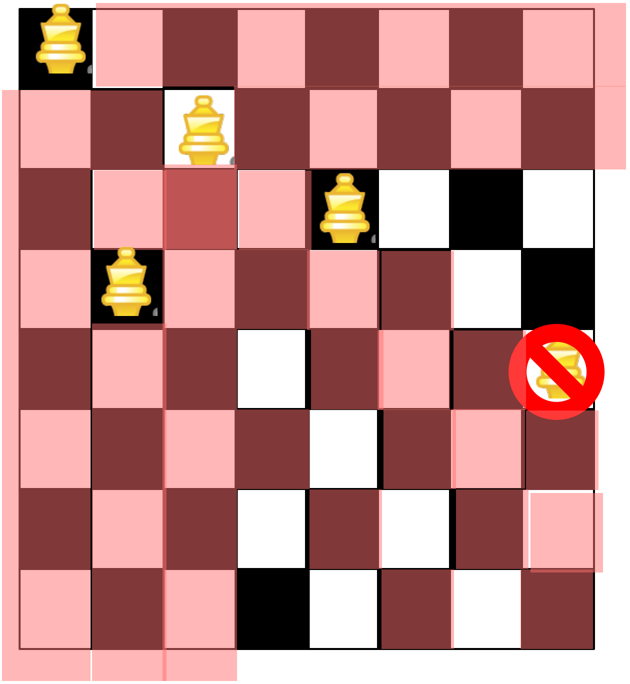
8x8 Example of N-Queens

- Now place others as viable
- After this configuration here, there are no locations in row 6 that is not under attack from the previous 5
- Now go back to row 5 and switch assignment to next viable option and progress back to row 6
- But still no location available so return back to row 5



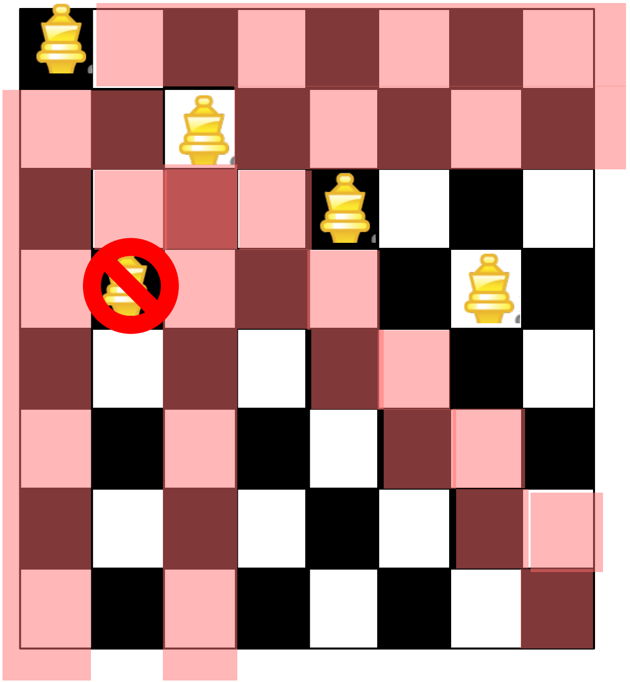
8x8 Example of N-Queens

- Now place others as viable
- After this configuration here, there are no locations in row 6 that is not under attack from the previous 5
- Now go back to row 5 and switch assignment to next viable option and progress back to row 6
- But still no location available so return back to row 5
- But now no more options for row 5 so return back to row 4
- BACKTRACK!!!!



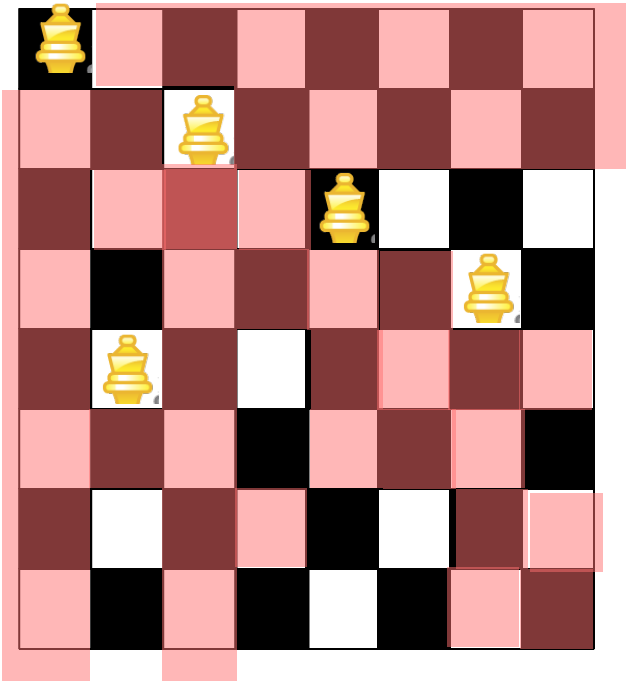
8x8 Example of N-Queens

- Now place others as viable
- After this configuration here, there are no locations in row 6 that is not under attack from the previous 5
- Now go back to row 5 and switch assignment to next viable option and progress back to row 6
- But still no location available so return back to row 5
- But now no more options for row 5 so return back to row 4
- Move to another place in row 4 and restart row 5 exploration



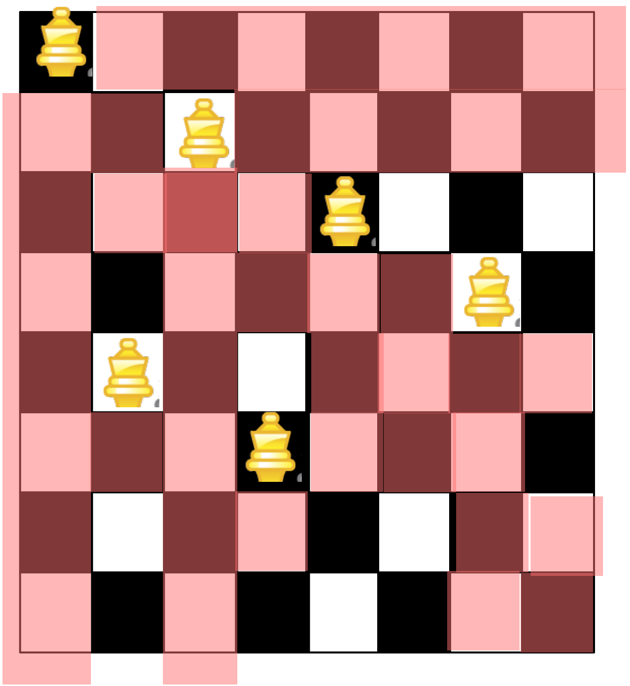
8x8 Example of N-Queens

- Now place others as viable
- After this configuration here, there are no locations in row 6 that is not under attack from the previous 5
- Now go back to row 5 and switch assignment to next viable option and progress back to row 6
- But still no location available so return back to row 5
- But now no more options for row 5 so return back to row 4
- Move to another place in row 4 and restart row 5 exploration



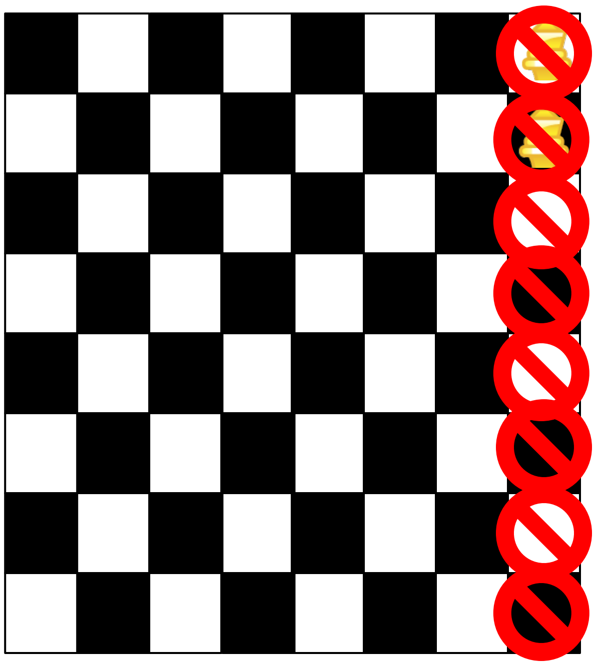
8x8 Example of N-Queens

- Now a viable option exists for row 6
- Keep going until you successfully place row 8 in which case you can return your solution
- What if no solution exists?



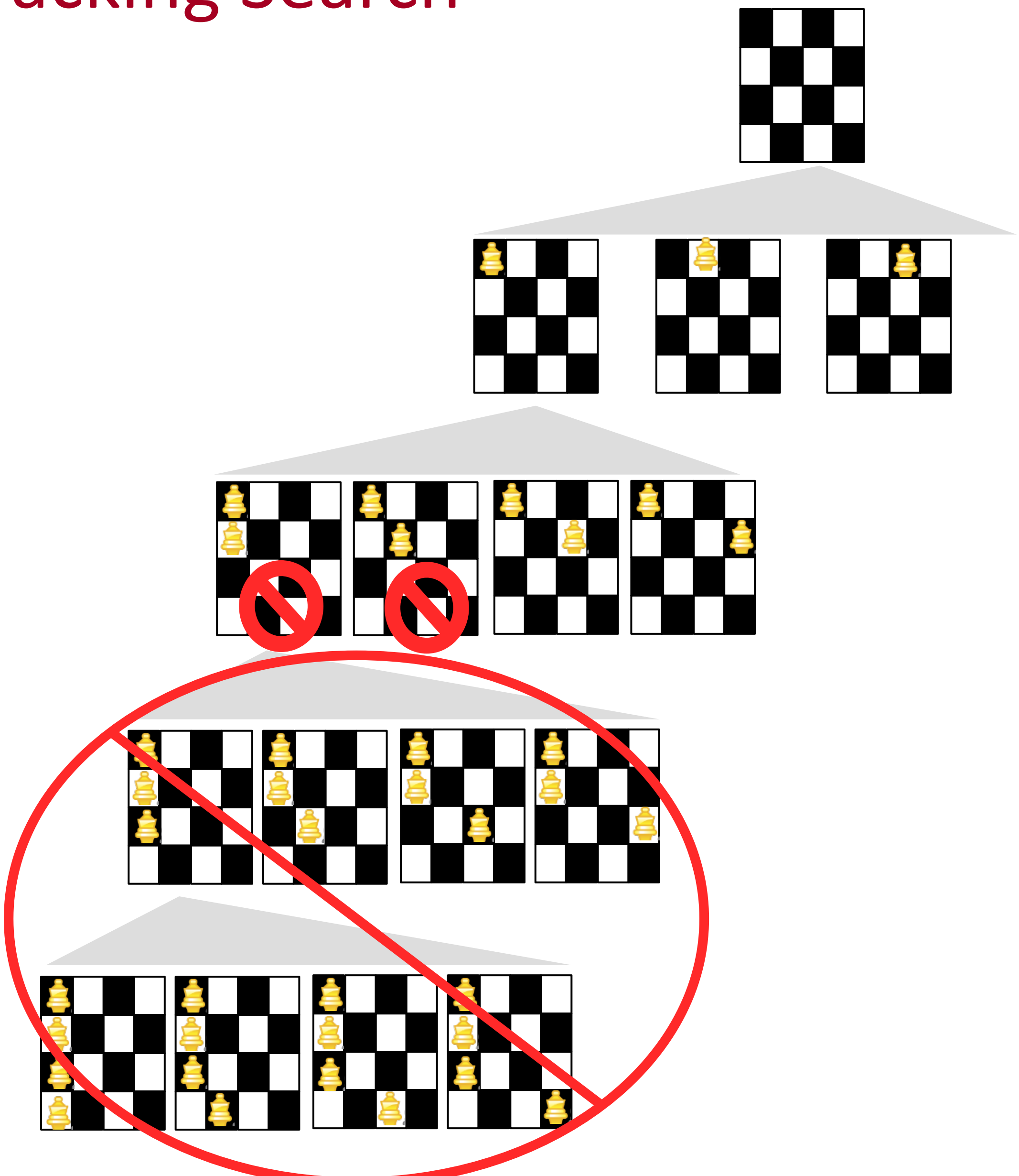
8x8 Example of N-Queens

- Now a viable option exists for row 6
- Keep going until you successfully place row 8 in which case you can return your solution
- What if no solution exists?
  - Row 1 queen would have exhausted all her options and still not find a solution



Backtracking Search

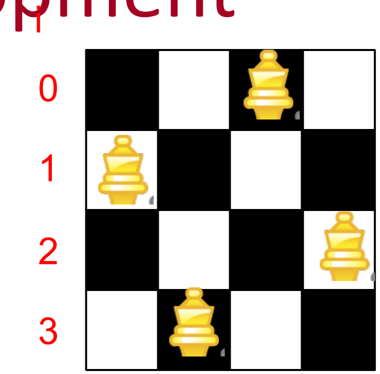
- Recursion can be used to generate all options
  - 'brute force' / test all options approach
  - Test for constraint satisfaction only at the bottom of the 'tree'
- But backtrack search attempts to 'prune' the search space
  - Rule out options at the partial assignment level



Brute force enumeration might test only once a possible complete assignment is made (i.e. all 4 queens on the board)

N-Queens Solution Development

- Let's develop the code
- 1 queen per row
  - Use an array where index represents the queen (and the row) and value is the column
- Start at row 0 and initiate the search [i.e. search(0)]
- Base case:
  - Rows range from 0 to n-1 so STOP when row == n
  - Means we found a solution
- Recursive case
  - Recursively try all column options for that queen
  - But haven't implemented check of viable configuration…



Index = Queen i in row i

q[i] = column of queen i

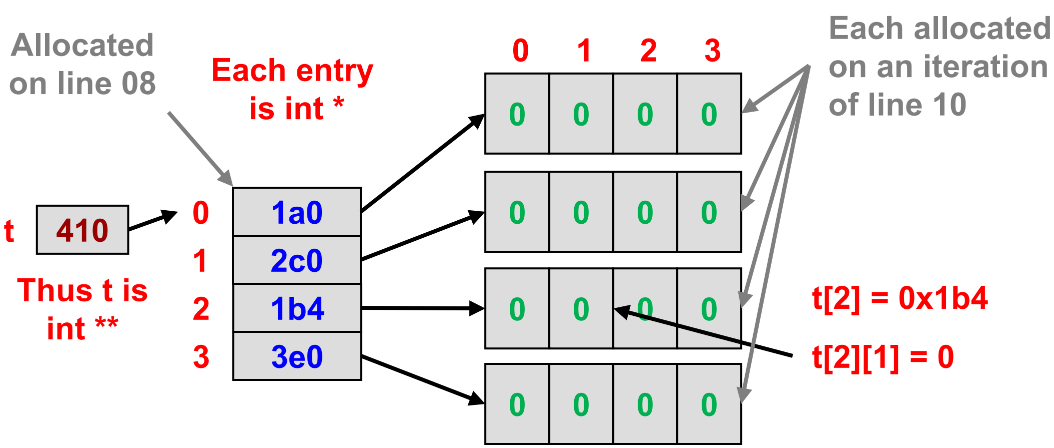
0	1	2	3
2	0	3	1

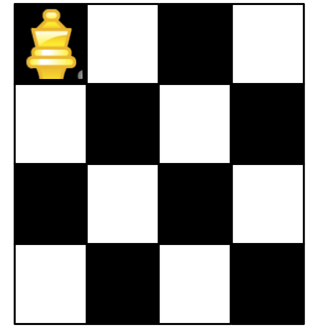
```
int *q;   // pointer to array storing
          // each queens location
int n;    // number of board / size

void search(int row)
{
  if(row == n)
    printSolution(); // solved!
  else {
   for(q[row]=0; q[row]<n; q[row]++){
     search(row+1);
  }
}
```

N-Queens Solution Development

- To check whether it is safe to place a queen in a particular column, let's keep a "threat" 2-D array indicating the threat level at each square on the board
  - Threat level of 0 means SAFE
  - When we place a queen we'll update squares that are now under threat
  - Let's name the array 't'
- Dynamically allocating 2D arrays in C/C++ doesn't really work
  - Instead conceive of 2D array as an "array of arrays" which boils down to a pointer to a pointer





0	0	1	1	1
1	1	1	0	0
2	1	0	1	0
3	1	0	0	1

Index = Queen i in row i

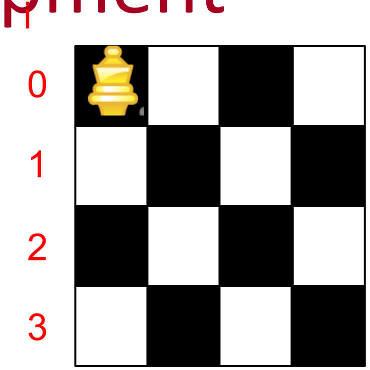
q[i] = column of queen i

0	1	2	3
0			

```
00 int *q;   // pointer to array storing
01           // each queens location
02 int n;    // number of board / size
03 int **t;  // thread 2D array
04
05 int main()
06 {
07   q = new int[n];
08   t = new int*[n];
09   for(int i=0; i < n; i++){
10     t[i] = new int[n];
11     for(int j = 0; j < n; j++){
12       t[i][j] = 0;
13     }
14   }
15   search(0); // start search
16   // deallocate arrays
17   return 0;
18 }
```

N-Queens Solution Development

- After we place a queen in a location, let's check that it has no threats
- If it's safe then we update the threats (+1) due to this new queen placement
- Now recurse to next row
- If we return, it means the problem was either solved or more often, that no solution existed given our placement so we remove the threats (-1)
- Then we iterate to try the next location for this queen



Index = Queen i in row i

q[i] = column of queen i

	0	1	2	3
q	0			

```
int *q;   // pointer to array storing
          // each queens location
int n;    // number of board / size
int **t; // n x n threat array
void search(int row)
{
  if(row == n)
    printSolution(); // solved!
  else {
   for(q[row]=0; q[row]<n; q[row]++){
     // check that col: q[row] is safe
     if(t[row][q[row]] == 0){
       // if safe place and continue
       addToThreats(row, q[row], 1);
       search(row+1);
       // if return, remove placement
       addToThreats(row, q[row], -1);
} } }
```

Safe to place queen in upper left

t	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

Now add threats

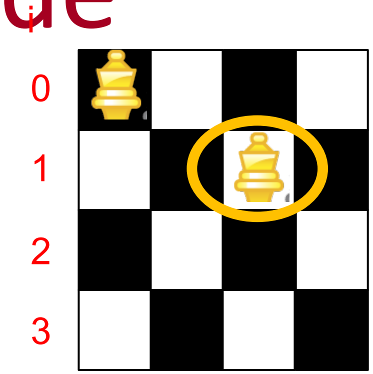
t	0	1	2	3
0	0	1	1	1
1	1	1	0	0
2	1	0	1	0
3	1	0	0	1

Upon return, remove threat and iterate to next option

t	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

addToThreats Code

- Observations
  - Already a queen in every higher row so addToThreats only needs to deal with positions lower on the board
    - Iterate row+1 to n-1
  - Enumerate all locations further down in the same column, left diagonal and right diagonal
  - Can use same code to add or remove a threat by passing in change
- Can't just use 2D array of booleans as a square might be under threat from two places and if we remove 1 piece we want to make sure we still maintain the threat



Index = Queen i in row i

q[i] = column of queen i

0	1	2	3
0			

```
void addToThreats(int row, int col, int change)
{
  for(int j = row+1; j < n; j++){
    // go down column
    t[j][col] += change;
    // go down right diagonal
    if( col+(j-row) < n )
       t[j][col+(j-row)] += change;
    // go down left diagonal
    if( col-(j-row) >= 0)
       t[j][col-(j-row)] += change;
  }
}
```

t	0	1	2	3
0	0	1	1	1
1	1	1	0	0
2	1	0	1	0
3	1	0	0	1

t	0	1	2	3
0	0	1	1	1
1	1	1	0	0
2	1	1	2	1
3	2	0	1	1

USC Viterbi School of Engineering

N-Queens Solution

```
int *q;   // queen location array
int n;    // number of board / size
int **t; // n x n threat array

int main()
{
  q = new int[n];
  t = new int*[n];
  for(int i=0; i < n; i++){
    t[i] = new int[n];
    for(int j = 0; j < n; j++){
      t[i][j] = 0;
    }
  }
  // do search
  if( ! search(0) )
     cout << "No sol!" << endl;
  // deallocate arrays
  return 0;
}
```

```
void addToThreats(int row, int col, int change)
{
  for(int j = row+1; j < n; j++){
    // go down column
    t[j][col] += change;
    // go down right diagonal
    if( col+(j-row) < n )
       t[j][col+(j-row)] += change;
    // go down left diagonal
    if( col-(j-row) >= 0)
       t[j][col-(j-row)] += change;
  }
}

bool search(int row)
{
  if(row == n){
    printSolution(); // solved!
    return true;
  }
  else {
   for(q[row]=0; q[row]<n; q[row]++){
     // check that col: q[row] is safe
     if(t[row][q[row]] == 0){
       // if safe place and continue
       addToThreats(row, q[row], 1);
       bool status = search(row+1);
       if(status) return true;
       // if return, remove placement
       addToThreats(row, q[row], -1);
     }
   }
   return false;
} }
```

General Backtrack Search Approach

- Select an item and set it to one of its options such that it meets current constraints
- Recursively try to set next item
- If you reach a point where all items are assigned and meet constraints, done…return through recursion stack with solution
- If no viable value for an item exists, backtrack to previous item and repeat from the top
- If viable options for the 1st item are exhausted, no solution exists
- Phrase:
  - Assign, recurse, unassign

General Outline of Backtracking Sudoku Solver

```
bool sudoku(int **grid, int r, int c)
{
  if( allSquaresComplete(grid) )
    return true;
  }
  // iterate through all options
  for(int i=1; i <= 9; i++){
    grid[r][c] = i;
    if( isValid(grid) ){
      bool status = sudoku(...);
      if(status) return true;
    }
  }
  return false;
}
```

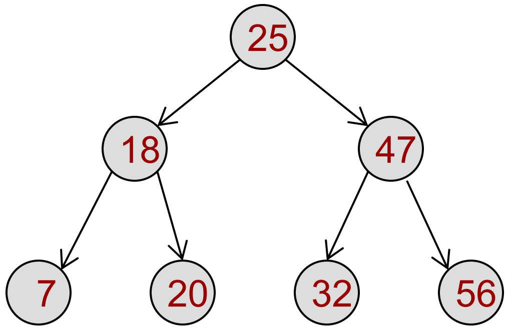
Assume r,c is current square to set and grid is the 2D array of values

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

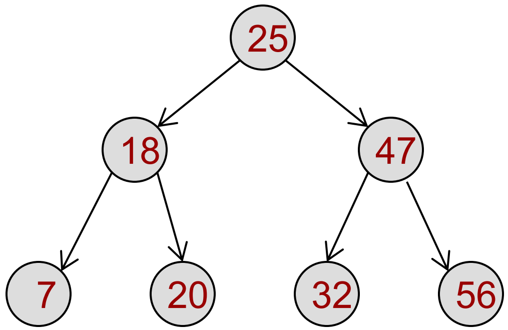


If we wanted to print the values in sorted order would you use an pre-order, in-order, or post-order traversal?

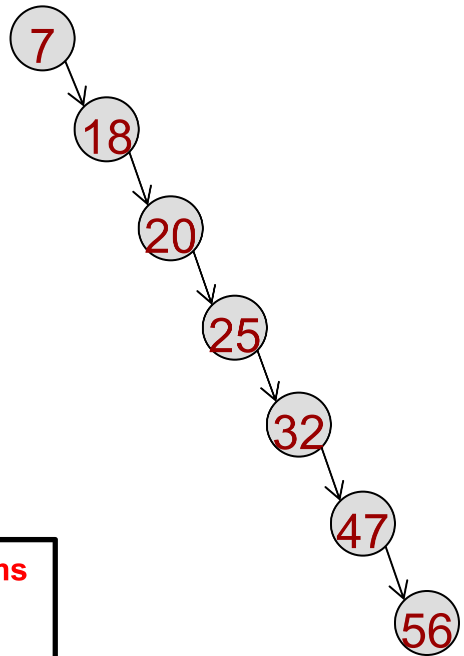
BST Insertion

- 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
- https://www.cs.usfca.edu/~galles/visualization/BST.html

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



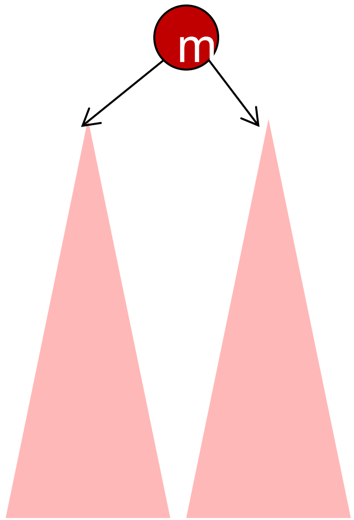
Insertion Order: 7, 18, 20, 25, 32, 47, 56



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

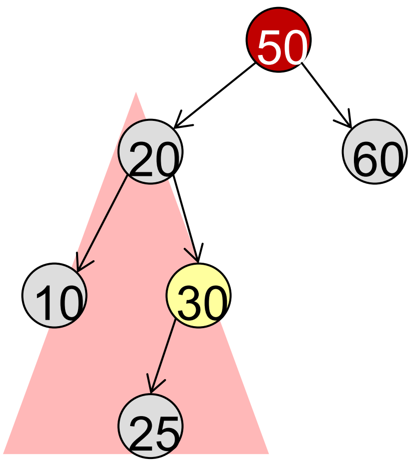
Successors & Predecessors

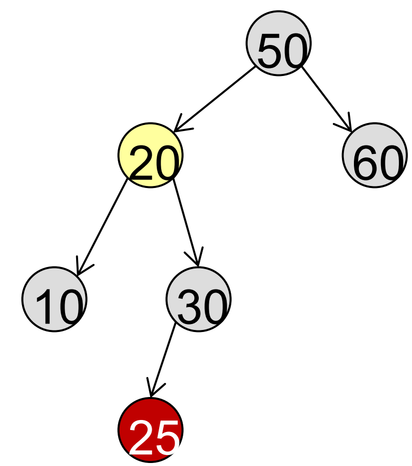
- Let's take a quick tangent that will help us understand how to do **BST Removal**
- Given a node in a BST
  - Its predecessor is defined as the next smallest value in the tree
  - Its successor is defined as the next biggest value in the tree
- Where would you expect to find a node's successor?
- Where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

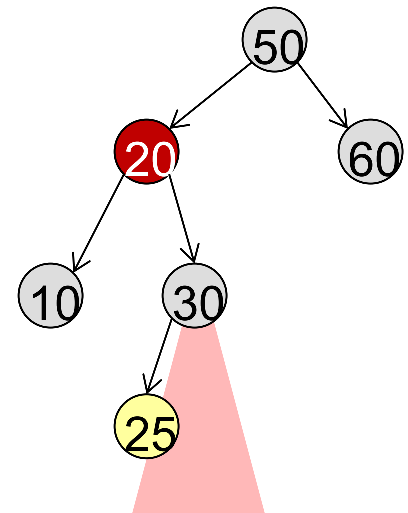




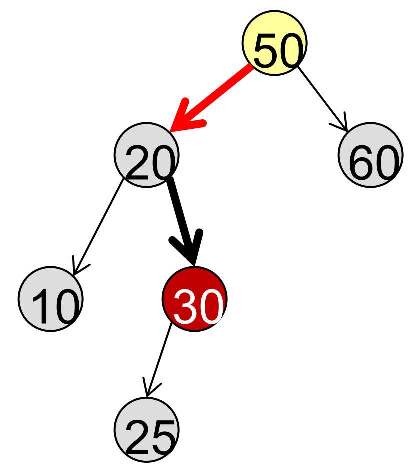
Successors

- 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

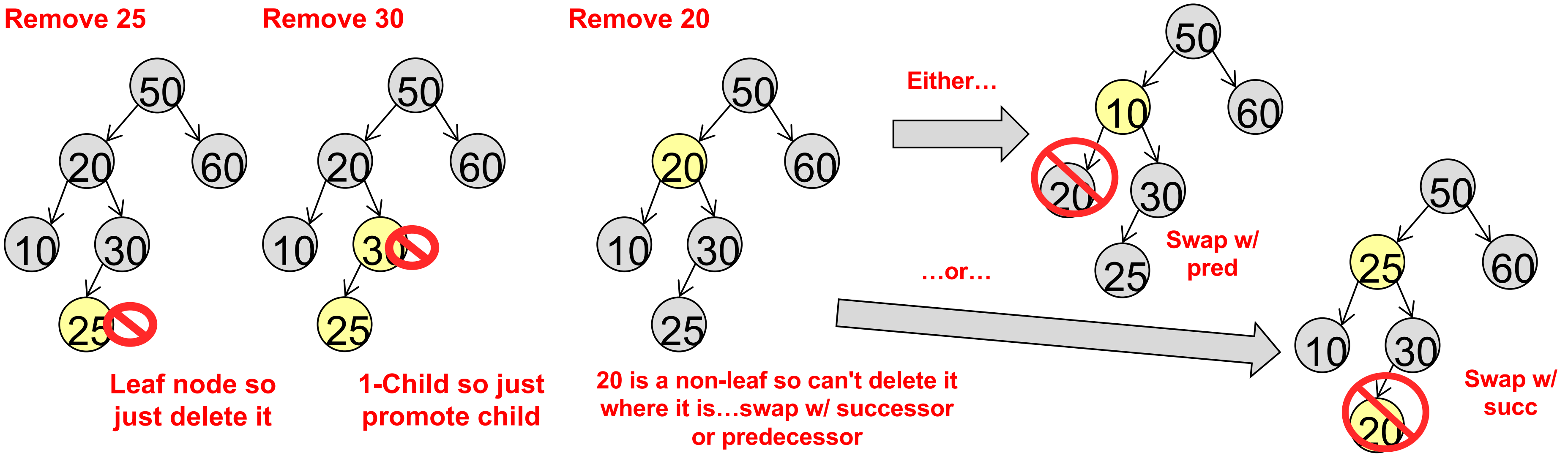


Succ(30)=50



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



BST Efficiency

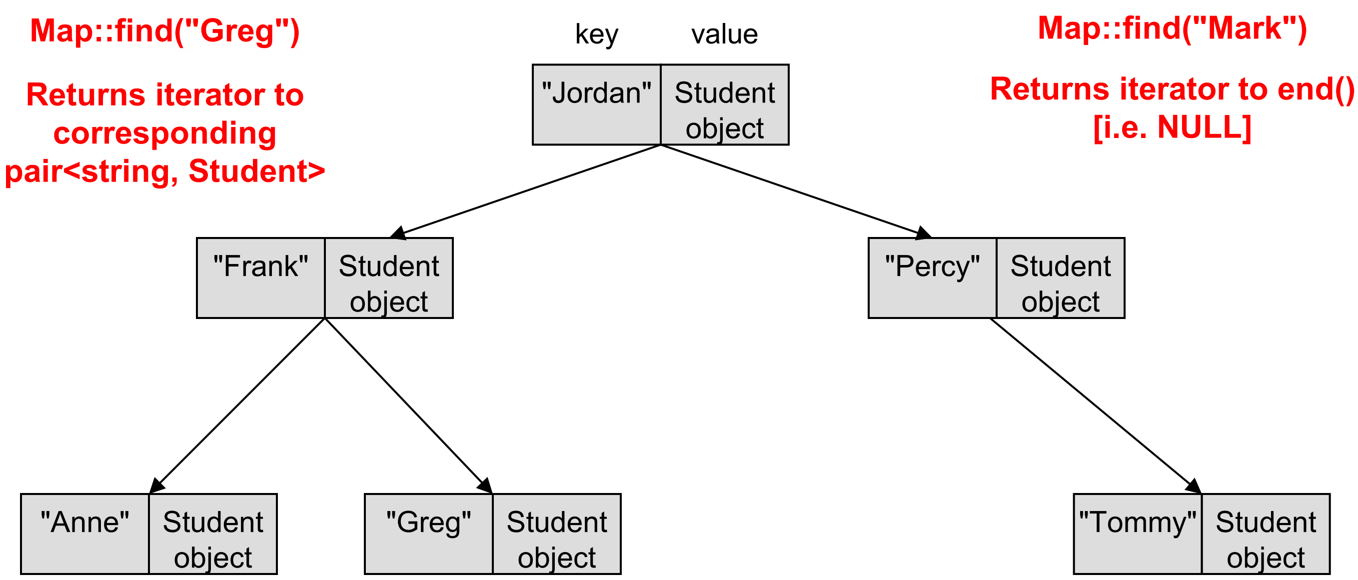
- Insertion
  - Balanced: O(log n)
  - Unbalanced: O(n)
- Removal
  - Balanced : O(log n)
  - Unbalanced: O(n)
- Find/Search
  - Balanced : O(log n)
  - Unbalanced: O(n)

```
#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!

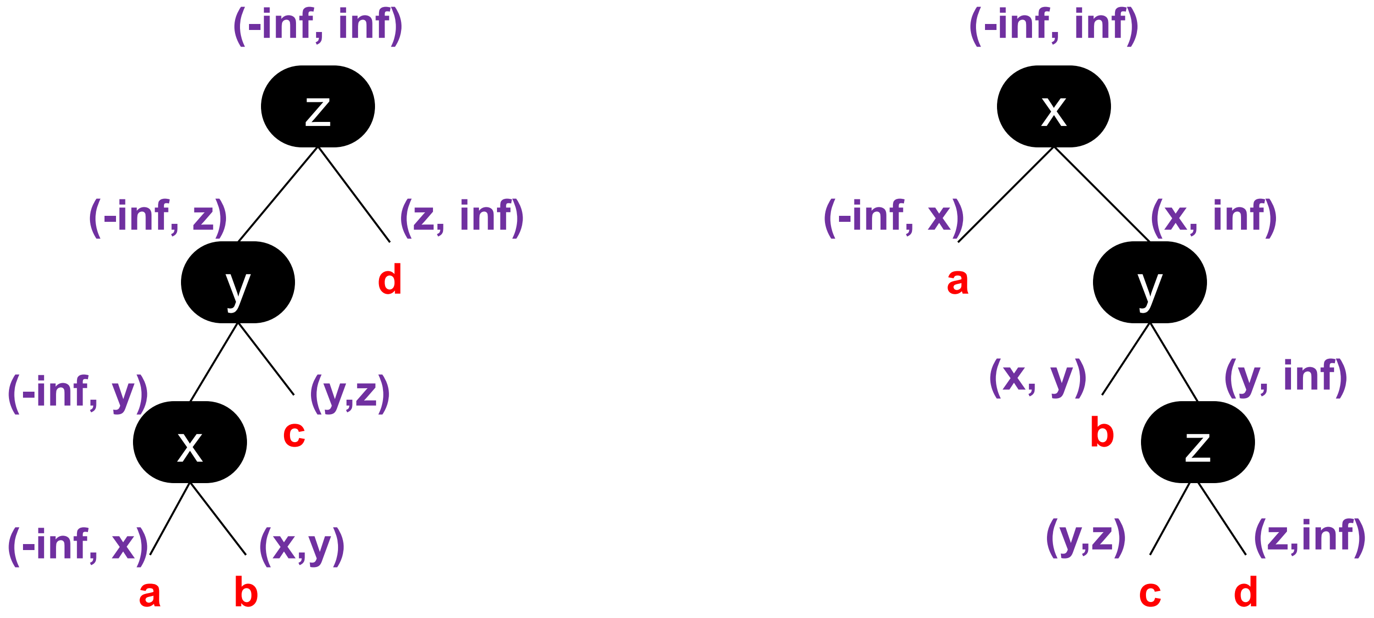


The key to balancing…

TREE ROTATIONS

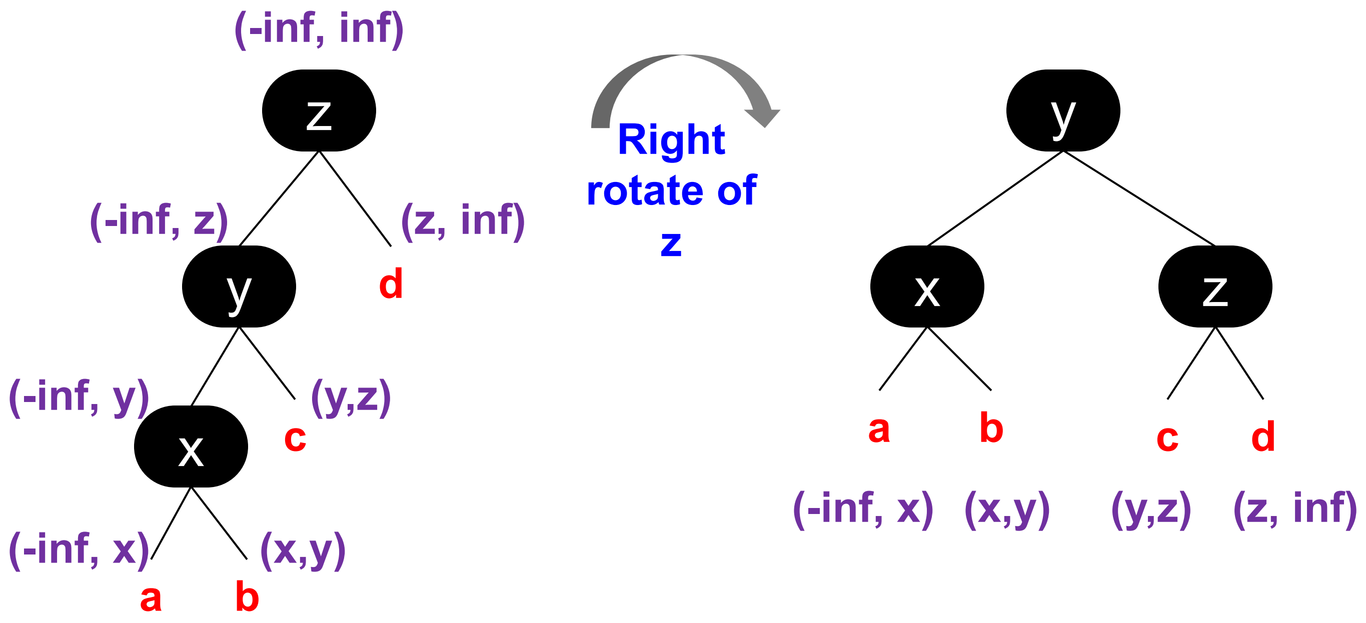
BST Subtree Ranges

- Consider a binary search tree, what range of values could be in the subtree rooted at each node
  - At the root, any value could be in the "subtree"
  - At the first left child?
  - At the first right child?



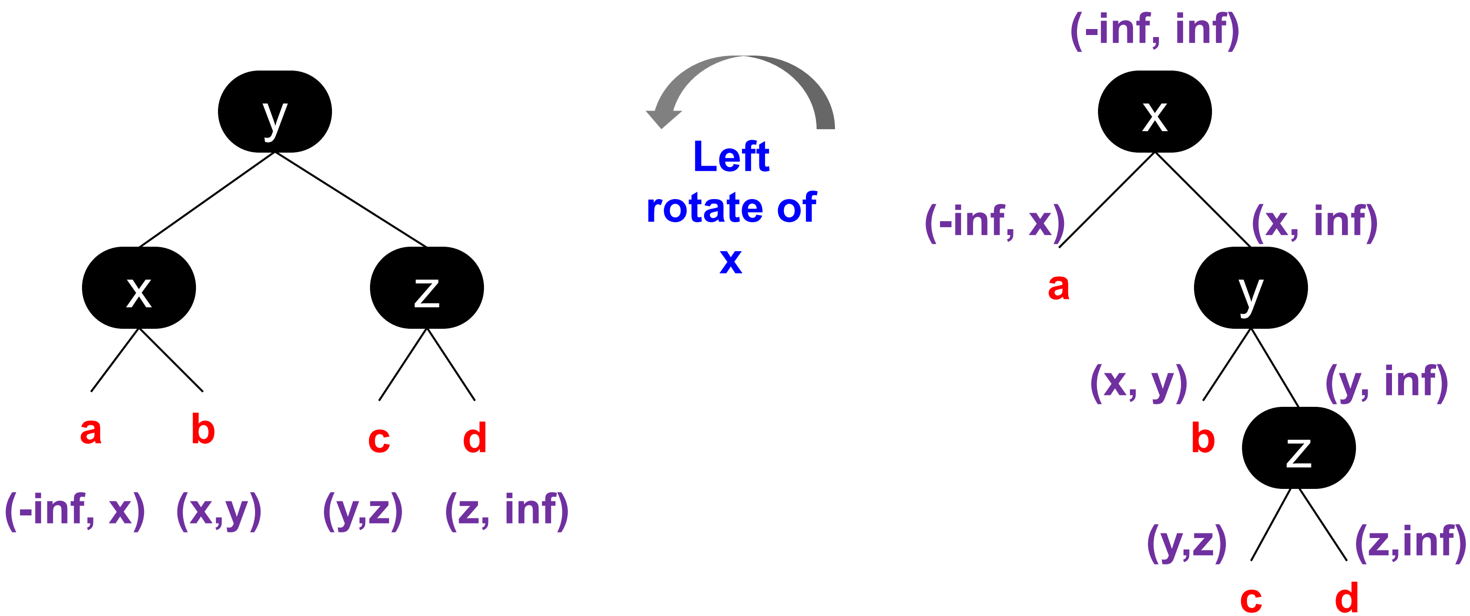
Right Rotation

- Define a right rotation as taking a left child, making it the parent and making the original parent the new right child
- Where do subtrees a, b, c and d belong?
  - Use their ranges to reason about it…



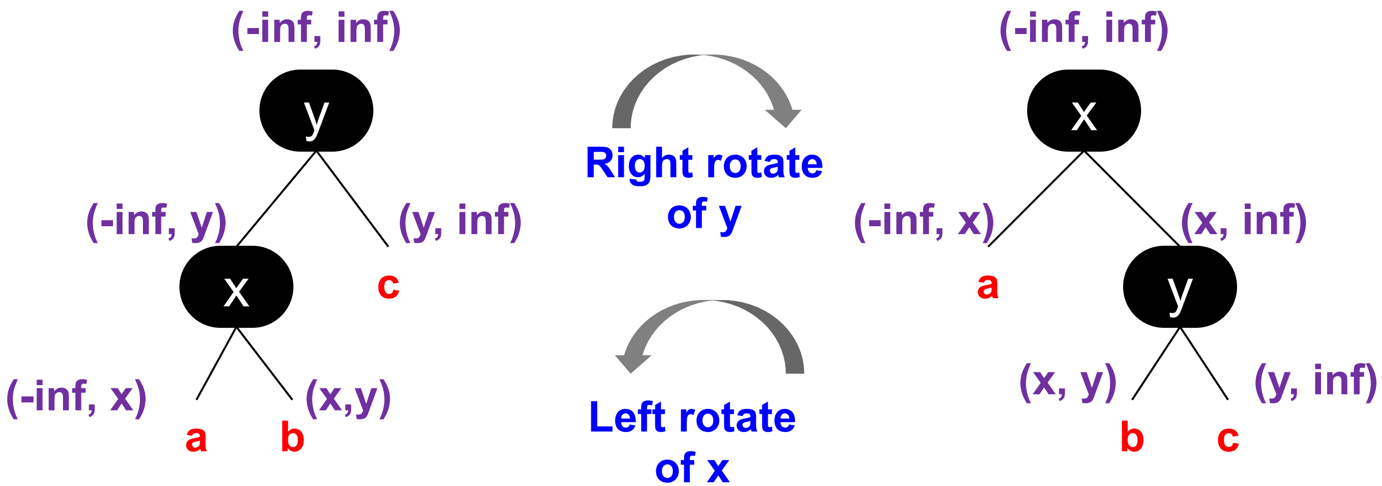
Left Rotation

- Define a left rotation as taking a right child, making it the parent and making the original parent the new left child
- Where do subtrees a, b, c and d belong?
  - Use their ranges to reason about it…



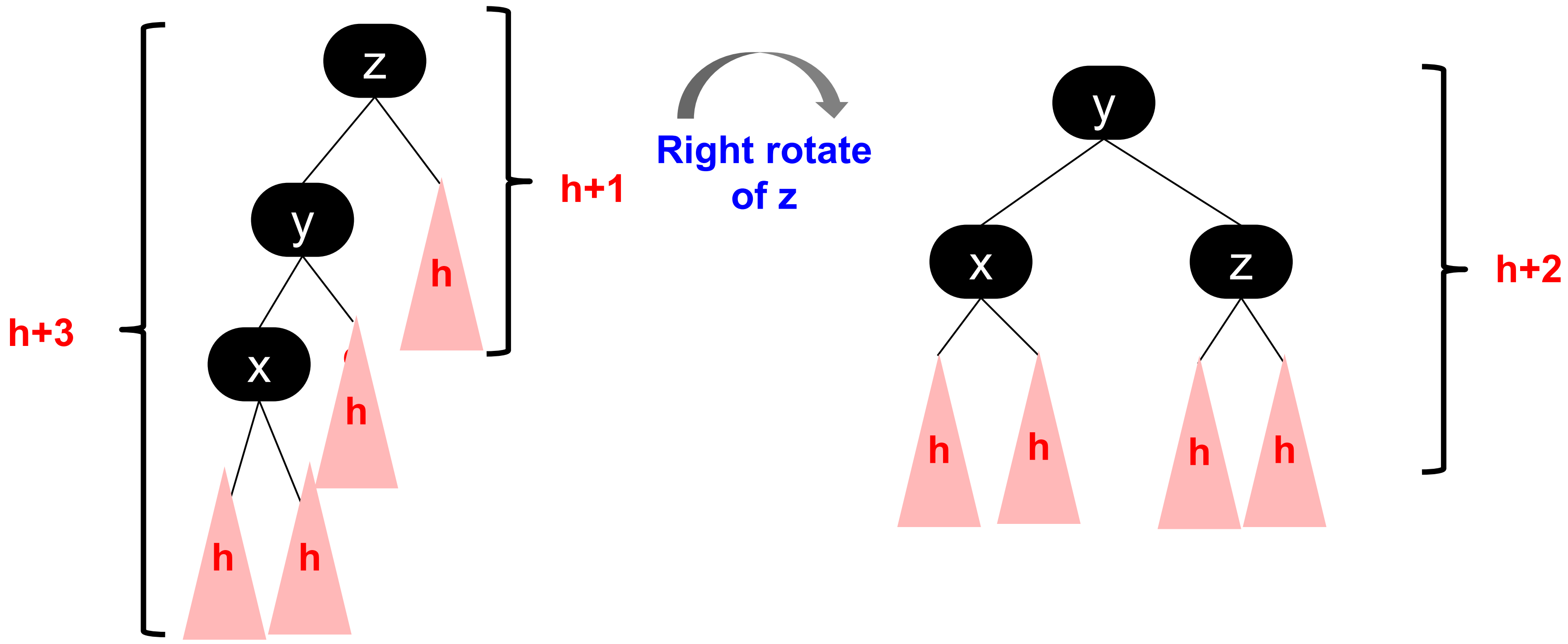
Rotations

- Define a right rotation as taking a left child, making it the parent and making the original parent the new right child
- Where do subtrees a, b, and c belong?
  - Use their ranges to reason about it…



Rotation's Effect on Height

- When we rotate, it serves to re-balance the tree



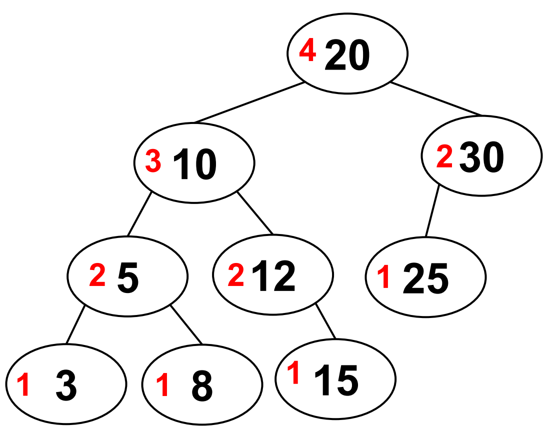
Let's always **specify the parent node** involved in a rotation (i.e. the node that is going to move **DOWN**).

Self-balancing tree proposed by Adelson-Velsky and Landis

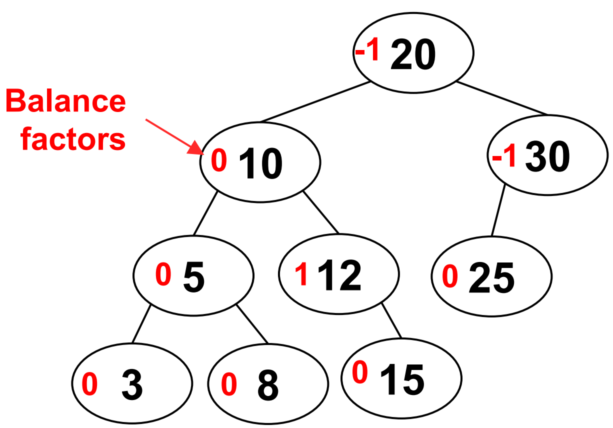
AVL TREES

AVL Trees

- A binary search tree where the **height difference** between left and right subtrees of a node is **at most 1**
  - Binary Search Tree (BST): Left subtree keys are less than the root and right subtree keys are greater
- Two implementations:
  - Height: Just store the height of the tree rooted at that node
  - Balance: Define b(n) as the balance of a node = (Right – Left) Subtree Height
    - Legal values are -1, 0, 1
    - Balances require at most 2-bits if we are trying to save memory.
    - **Let's use balance for this lecture.**



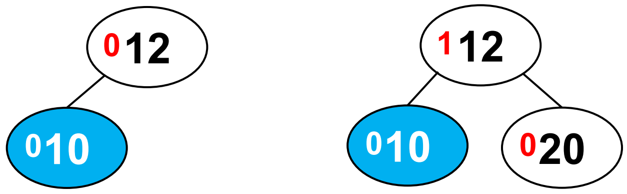
AVL Tree storing Heights



AVL Tree storing balances

Adding a New Node

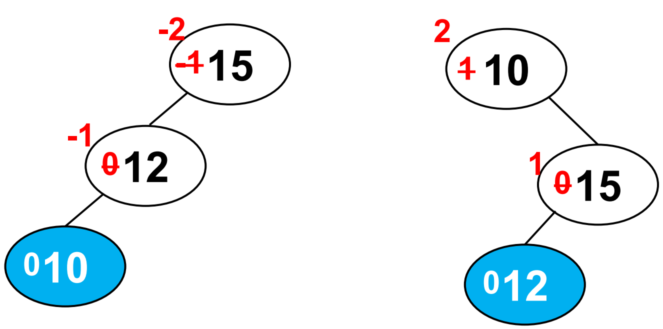
- Once a new node is added, can its parent be out of balance?
  - No, assuming the tree is "in-balance" when we start.
  - Thus, our parent has to have
    - A balance of 0
    - A balance of 1 if we are a new left child or -1 if a new right child
  - Otherwise it would not be our parent or the parent would have been out of balance already



USC Viterbi School of Engineering

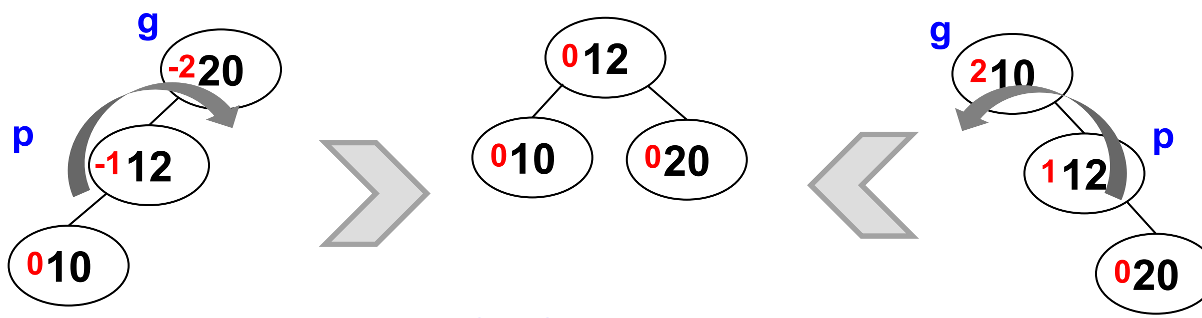
Losing Balance

- If our parent is not out of balance, is it possible our grandparent is out of balance?
- Sure, so we need a way to re-balance it

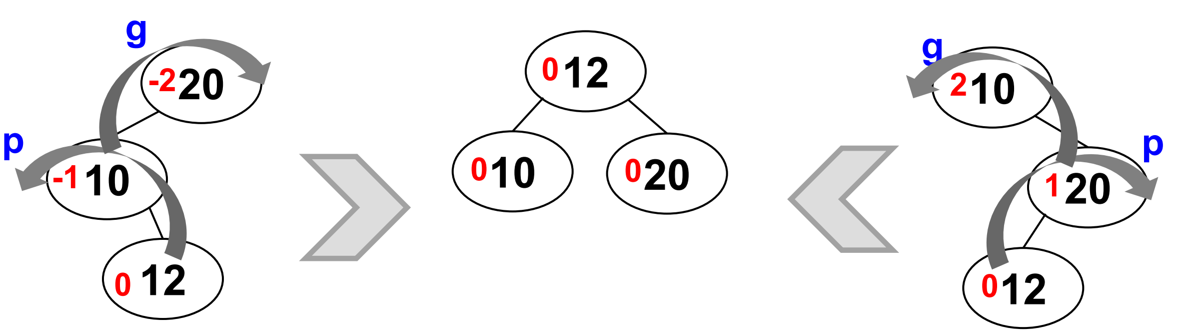


To Zig or Zag

- The rotation(s) required to balance a tree is/are dependent on the grandparent, parent, child relationships
- We can refer to these as the zig-zig case and zig-zag case
- Zig-zig requires 1 rotation
- Zig-zag requires 2 rotations (first converts to zig-zig)



Left-left or Right-right
(a.k.a. Zig-zig)
[One left/right rotation of g]



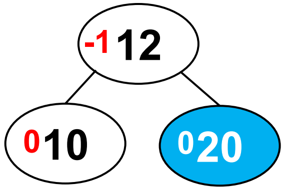
Left-right or Right-left
(a.k.a. Zig-zag)
[Rotate p then g]

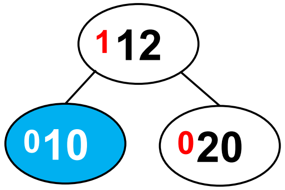
Disclaimer

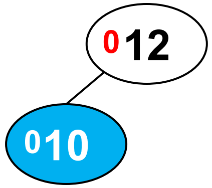
- There are many ways to structure an implementation of an AVL tree…the following slides represent just 1
  - Focus on the bigger picture ideas as that will allow you to more easily understand other implementations

Insert(n)

- If empty tree => set as root, b(n) = 0, done!
- Insert n (by walking the tree to a leaf, p, and inserting the new node as its child), set balance to 0, and look at its parent, p
  - If b(p) = -1, then b(p) = 0. Done!
  - If b(p) = +1, then b(p) = 0. Done!
  - If b(p) = 0, then update b(p) and call insert-fix(p, n)







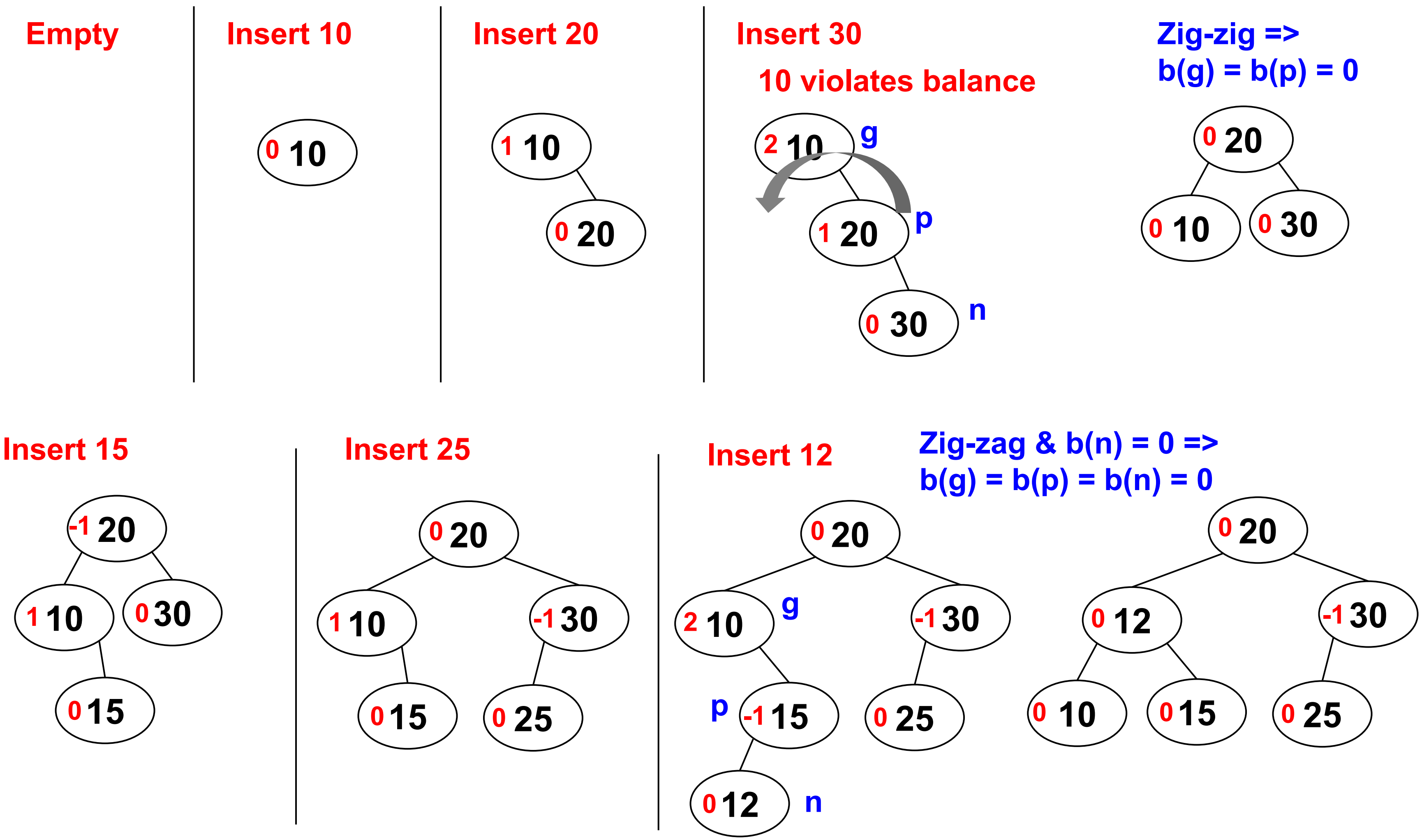
Insert-fix(p, n)

- Precondition: p and n are balanced: {+1,0,-1}
- Postcondition: g, p, and n are balanced: {+1,0,-1}
- If p is null or parent(p) is null, return
- Let g = parent(p)
- Assume p is left child of g [For right child swap left/right, +/-]
  - g.balance += -1
  - if g.balance == 0, return
  - if g.balance == -1, insertFix(g, p)
  - If g.balance == -2
    - If zig-zig then rotateRight(g); p.balance = g.balance = 0
    - If zig-zag then rotateLeft(p); rotateRight(g);
      - if n.balance == -1 then p.balance = 0; g.balance(+1); n.balance = 0;
      - if n.balance == 0 then p.balance = 0; g.balance(0); n.balance = 0;
      - if n.balance == +1 then p.balance = -1; g.balance(0); n.balance = 0;

Note: If you perform a rotation, you will NOT need to recurse. You are done!

Insertion

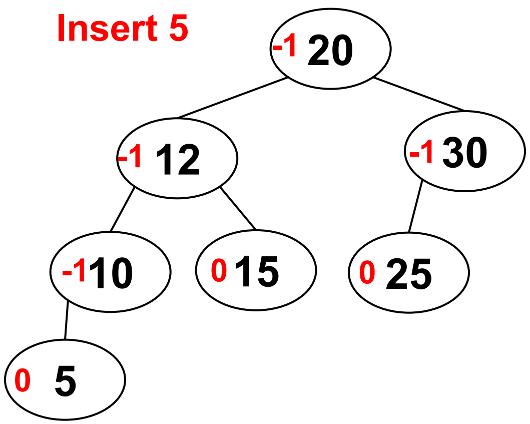
- Insert 10, 20, 30, 15, 25, 12, 5, 3, 8



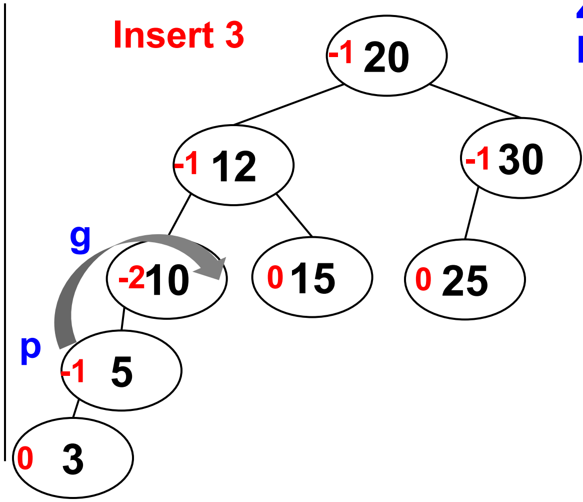
Insertion

- Insert 10, 20, 30, 15, 25, 12, 5, 3, 8

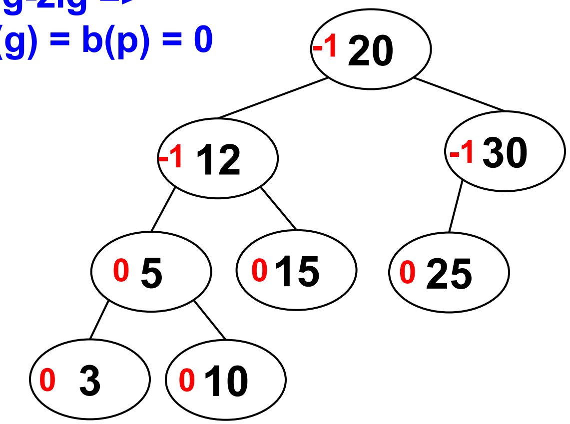
Insert 5



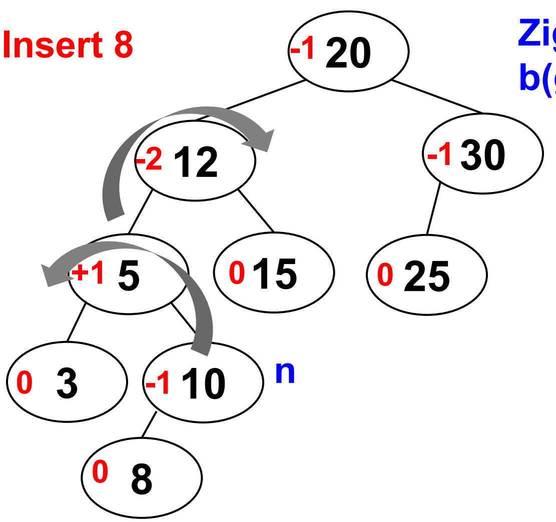
Insert 3



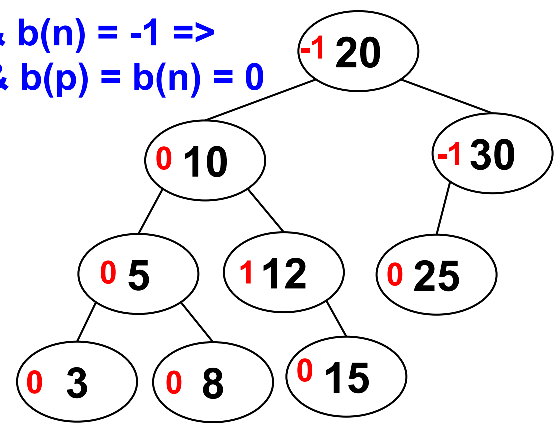
Zig-zig =>
b(g) = b(p) = 0



Insert 8

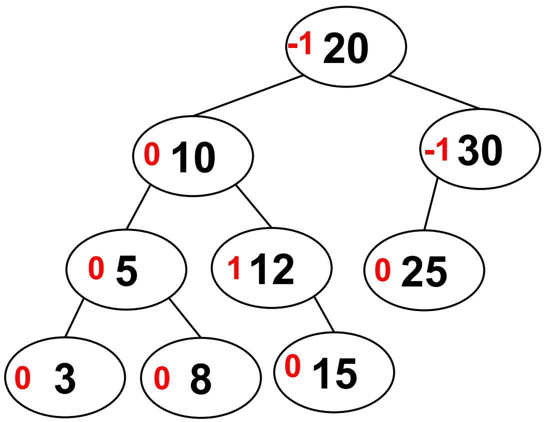


Zig-zag & b(n) = -1 =>
b(g) = 1 & b(p) = b(n) = 0



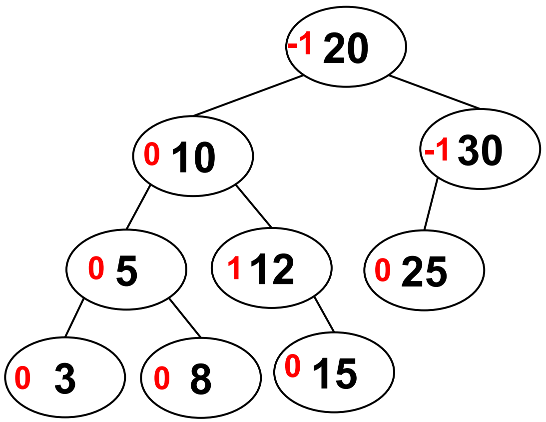
Insertion Exercise 1

- Insert key=28



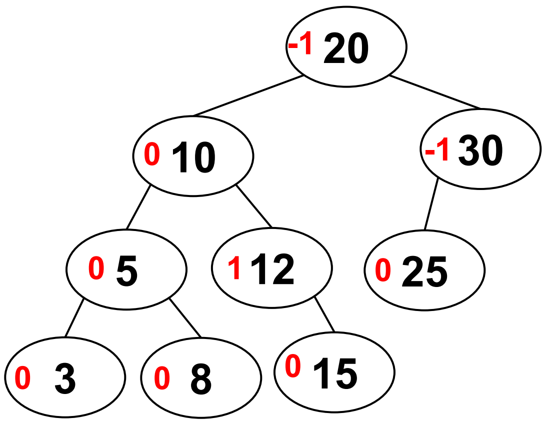
Insertion Exercise 2

- Insert key=17



Insertion Exercise 3

- Insert key=2



Remove Operation

- Remove operations may also require rebalancing via rotations
- The key idea is to update the balance of the nodes on the ancestor pathway
- If an ancestor gets out of balance then perform rotations to rebalance
  - Unlike insert, performing rotations does not mean you are done, but need to continue
- There are slightly more cases to worry about but not too many more

Remove

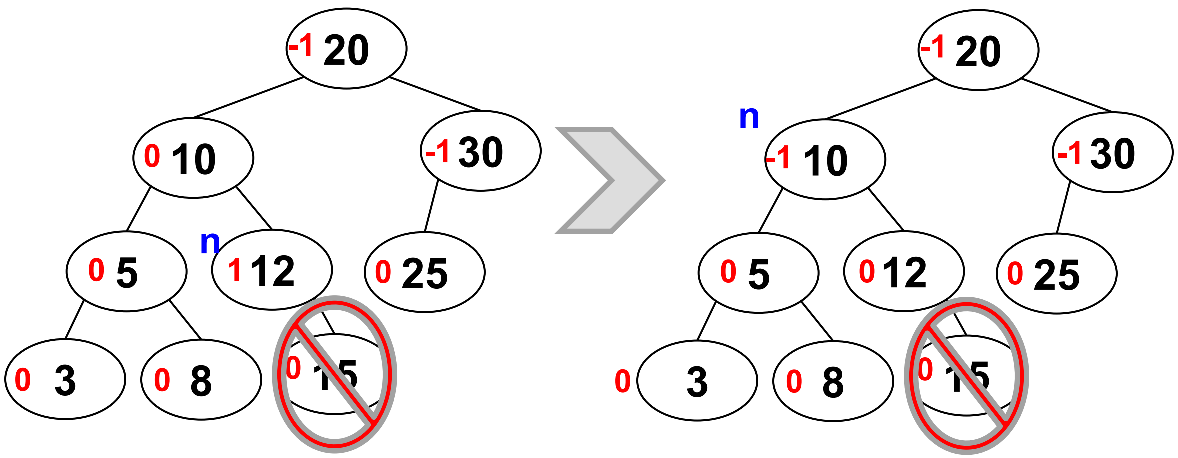
- Let n = node to remove (perform BST find) and p = parent(n)
- If n has 2 children, swap positions with in-order successor and perform the next step
  - Now n has 0 or 1 child guaranteed
- If n is not in the root position determine its relationship with its parent
  - If n is a left child, let diff = +1
  - if n is a right child, let diff = -1
- Delete n and update tree, including the root if necessary
- removeFix(p, diff);

RemoveFix(n, diff)

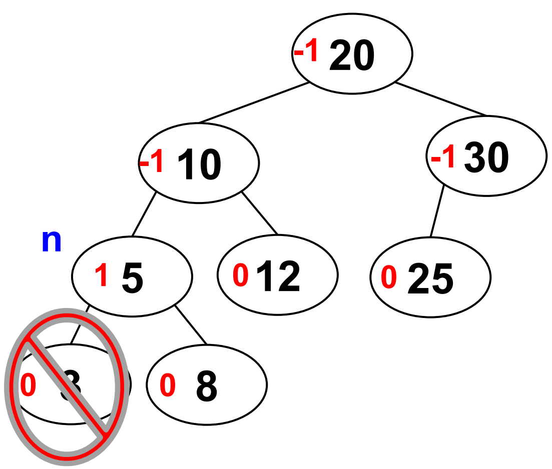
- If n is null, return
- Let ndiff = +1 if n is a left child and -1 otherwise
- Let p = parent(n). Use this value of p when you recurse.
- If balance of n would be -2 (i.e. balance(n) + diff == -2)
  - [Perform the check for the mirror case where balance(n) + diff == +2, flipping left/right and -1/+1]
  - Let c = left(n), the taller of the children
  - If balance(c) == -1 or 0 (zig-zig case)
    - rotateRight(n)
    - if balance(c) == -1 then balance(n) = balance(c) = 0, removeFix(p, ndiff)
    - if balance(c) == 0 then balance(n) = -1, balance(c) = +1, done!
  - else if balance(c) == 1 (zig-zag case)
    - rotateLeft(c) then rotateRight(n)
    - Let g = right(c)
    - If balance(g) == +1 then balance(n) = 0, balance(c) = -1, balance(g) = 0
    - If balance(g) == 0 then balance(n) = balance(c) = 0, balance(g) = 0
    - If balance(g) == -1 then balance(n) = +1, balance(c) = 0, balance(g) = 0
    - removeFix(parent(p), ndiff);
- else if balance(n) == 0 then balance(n) += diff, done!
- else balance(n) = 0, removeFix(p, ndiff)

Remove Examples

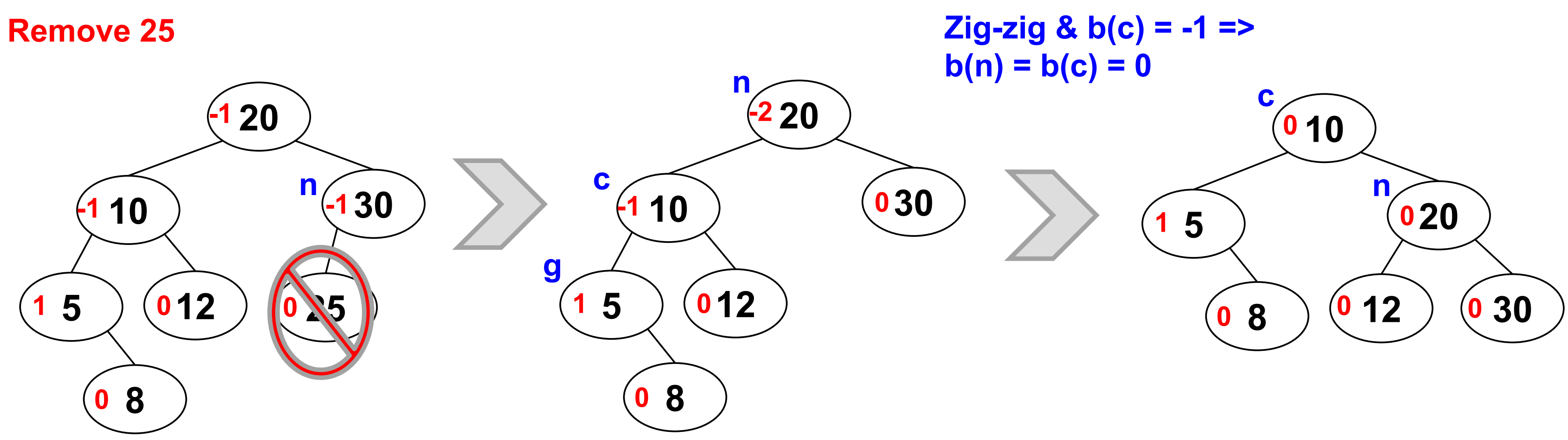
Remove 15



Remove 3

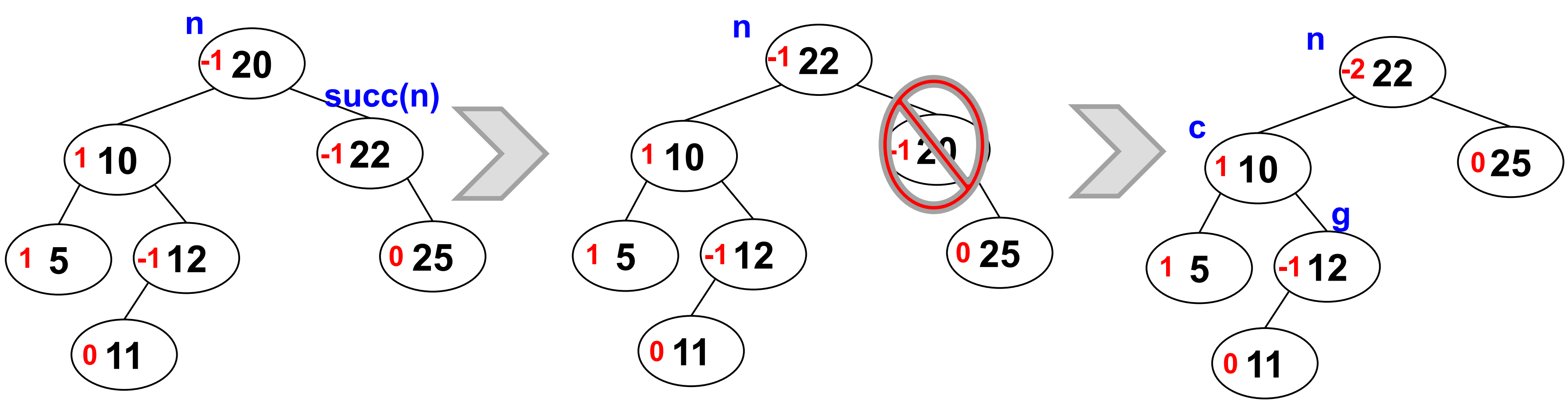


Remove Examples

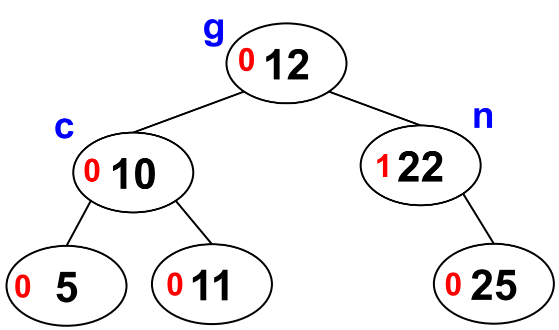


Remove Examples

Remove 20

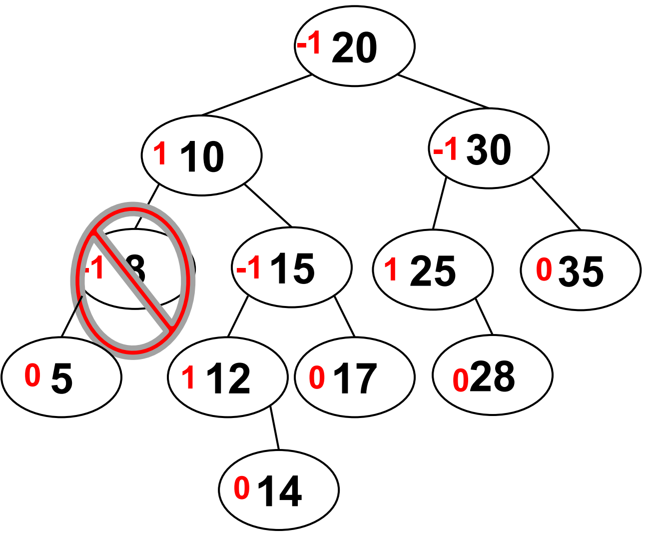


Zig-zag & b(g) = -1 =>
b(n) = +1, b(c) = 0, b(g) = 0



Remove Example 1

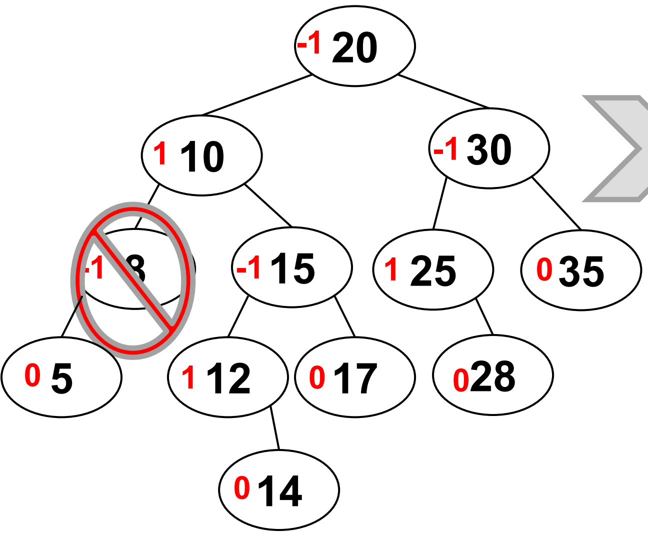
Remove 8



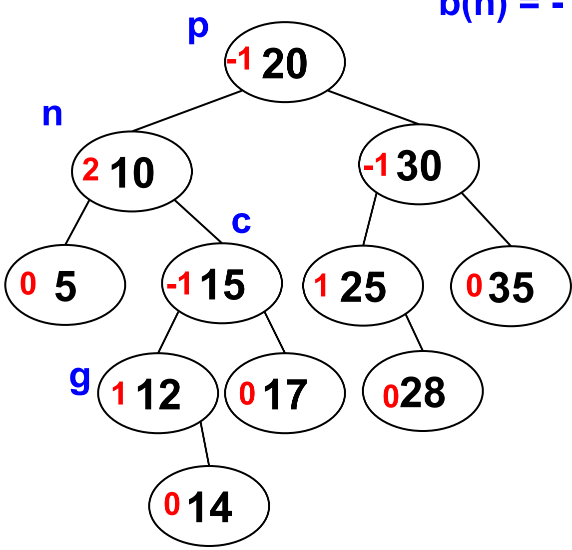
USC Viterbi School of Engineering

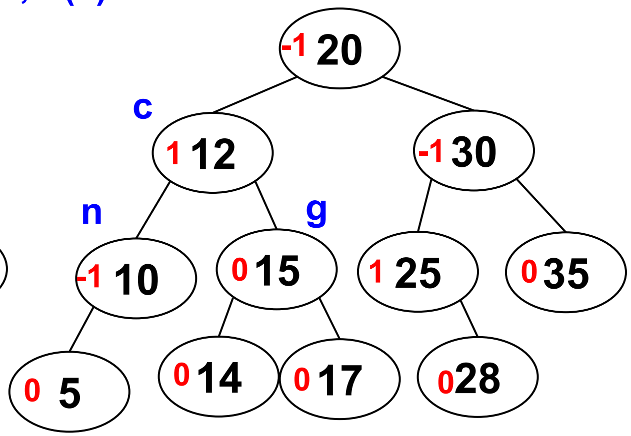
Remove Example 1

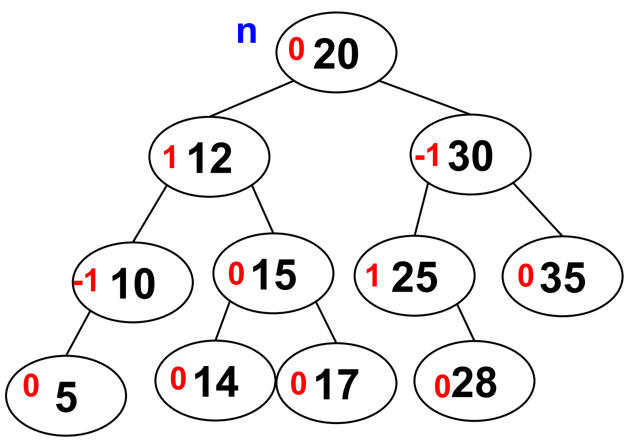
Remove 8



Zig-zag & b(1) = 0 =>
b(n) = -1, b(c) = 0

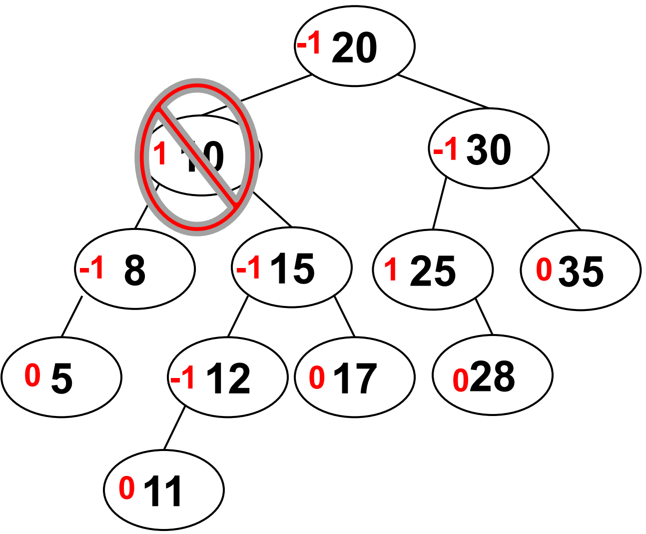






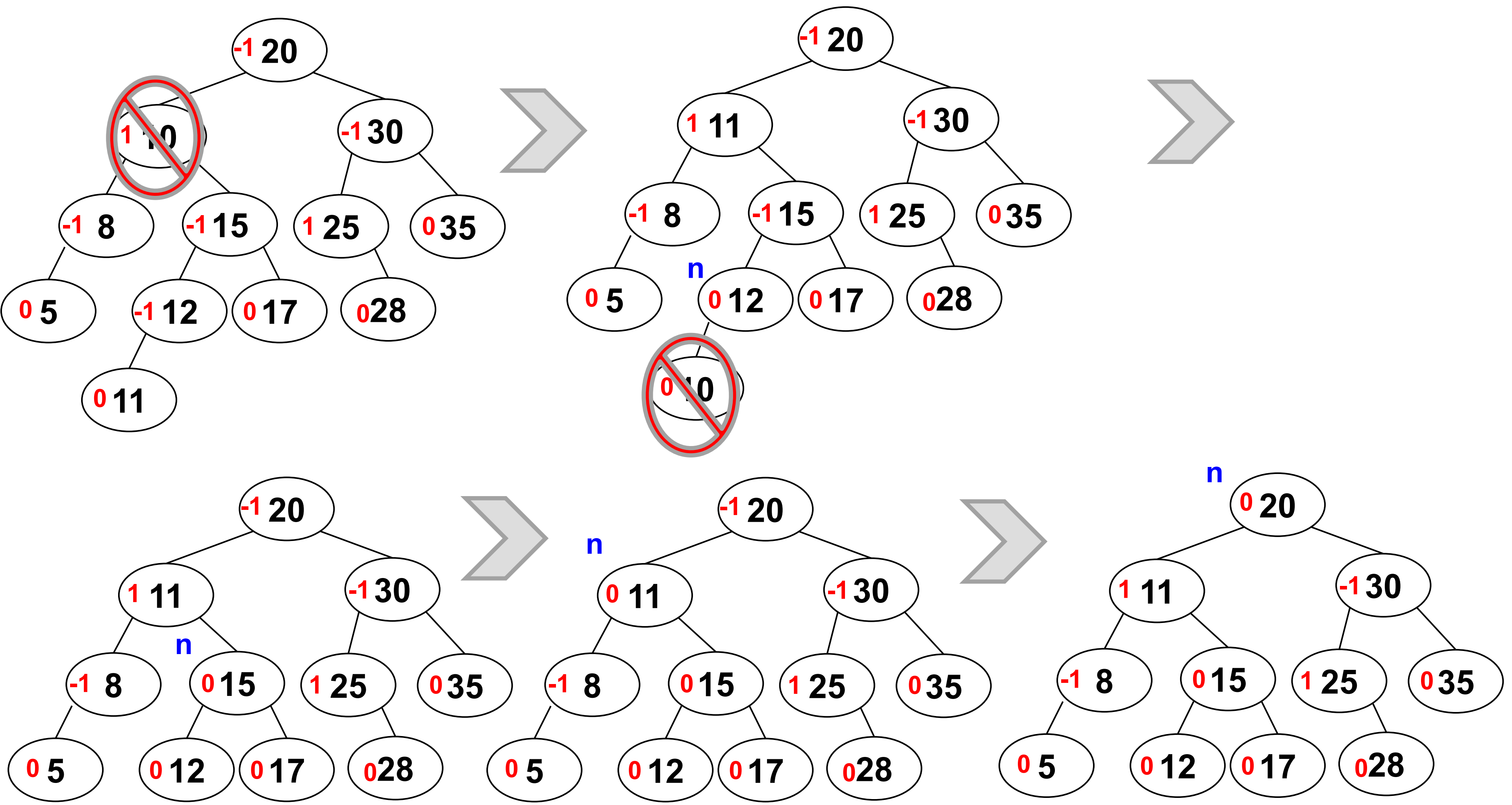
Remove Example 2

Remove 10



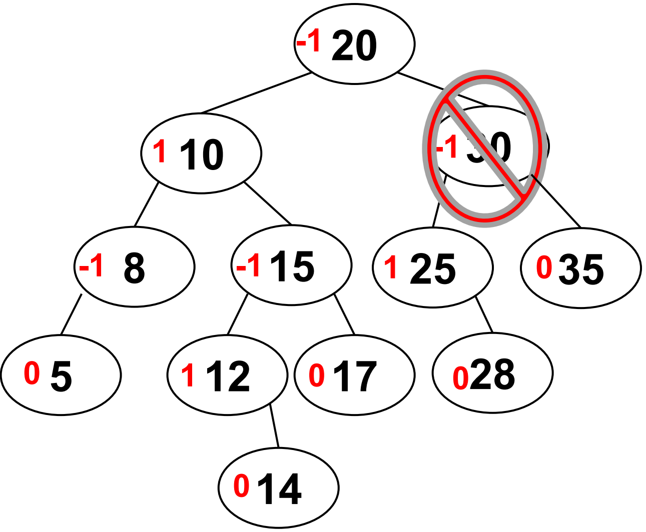
Remove Example 2

Remove 10



Remove Example 3

Remove 30

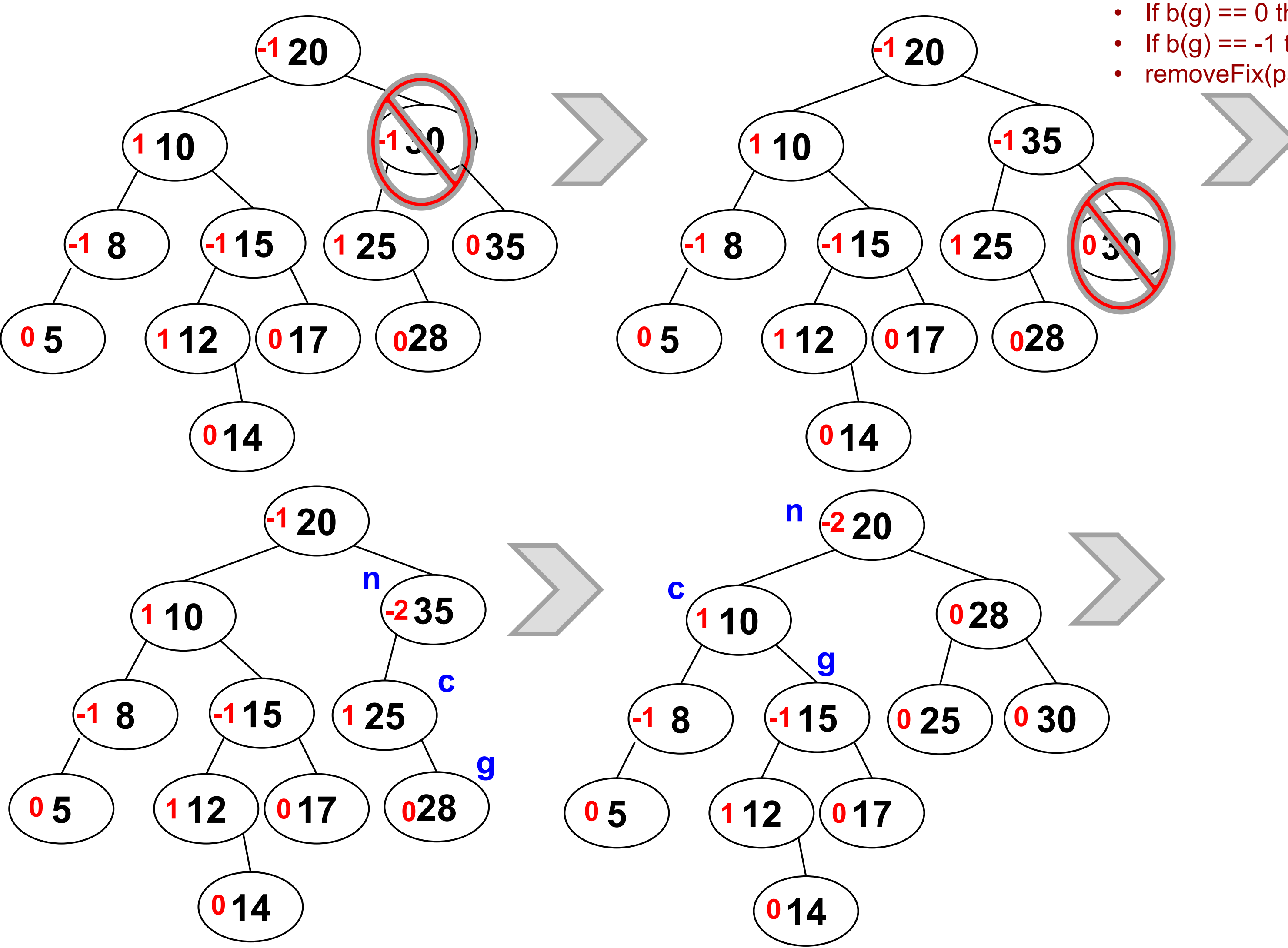


Remove Example 3

else if b(c) == 1 (zig-zag case)

- rotateLeft(c) then rotateRight(n)
- Let g = right(c), b(g) = 0
- If b(g) == +1 then b(n) = 0, b(c) = -1, b(g) = 0
- If b(g) == 0 then b(n) = b(c) = 0, b(g) = 0
- If b(g) == -1 then b(n) = +1, b(c) = 0, b(g) = 0
- removeFix(parent(p), ndiff);

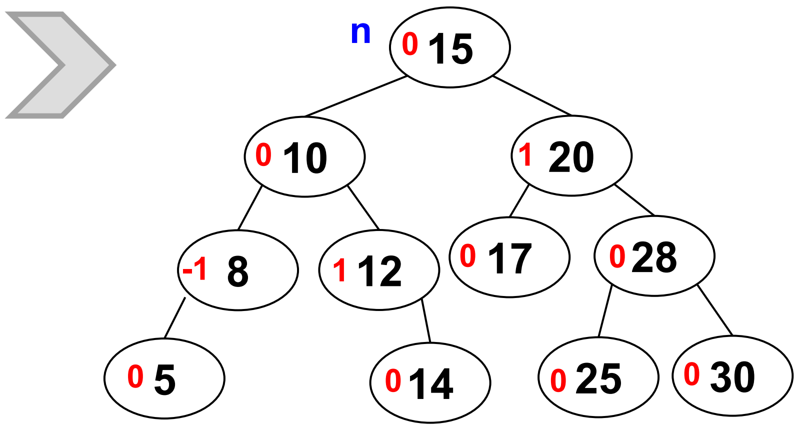
Remove 30



Remove Example 3 (cont)

else if b(c) == 1 (zig-zag case)
- rotateLeft(c) then rotateRight(n)
- Let g = right(c), b(g) = 0
- If b(g) == +1 then b(n) = 0, b(c) = -1, b(g) = 0
- If b(g) == 0 then b(n) = b(c) = 0, b(g) = 0
- If b(g) == -1 then b(n) = +1, b(c) = 0, b(g) = 0
- removeFix(parent(p), ndiff);

Remove 30 (cont.)



Online Tool

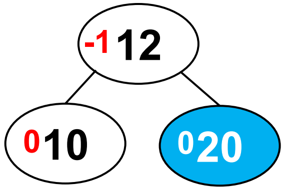
- https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

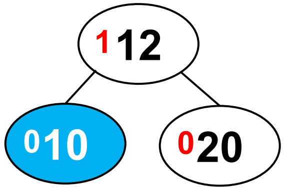
Distribute these 4 to students

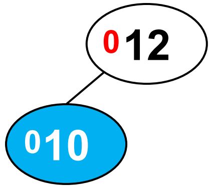
FOR PRINT

Insert(n)

- If empty tree => set as root, b(n) = 0, done!
- Insert n (by walking the tree to a leaf, p, and inserting the new node as its child), set balance to 0, and look at its parent, p
  – If b(p) = -1, then b(p) = 0. Done!
  – If b(p) = +1, then b(p) = 0. Done!
  – If b(p) = 0, then update b(p) and call insert-fix(p, n)







Insert-fix(p, n)

- Precondition: p and n are balanced: {-1,0,-1}
- Postcondition: g, p, and n are balanced: {-1,0,-1}
- If p is null or parent(p) is null, return
- Let g = parent(p)
- Assume p is left child of g [For right child swap left/right, +/-]
  - g.balance += -1
  - if g.balance == 0, return
  - if g.balance == -1, insertFix(g, p)
  - If g.balance == -2
    - If zig-zig then rotateRight(g); p.balance = g.balance = 0
    - If zig-zag then rotateLeft(p); rotateRight(g);
      - if n.balance == -1 then p.balance = 0; g.balance(+1); n.balance = 0;
      - if n.balance == 0 then p.balance = 0; g.balance(0); n.balance = 0;
      - if n.balance == +1 then p.balance = -1; g.balance(0); n.balance = 0;

Note: If you perform a rotation, you will NOT need to recurse. You are done!

Remove

- Let n = node to remove (perform BST find) and p = parent(n)
- If n has 2 children, swap positions with in-order successor and perform the next step
  - Now n has 0 or 1 child guaranteed
- If n is not in the root position determine its relationship with its parent
  - If n is a left child, let diff = +1
  - if n is a right child, let diff = -1
- Delete n and update tree, including the root if necessary
- removeFix(p, diff);

RemoveFix(n, diff)

- If n is null, return
- Let ndiff = +1 if n is a left child and -1 otherwise
- Let p = parent(n). Use this value of p when you recurse.
- If balance of n would be -2 (i.e. balance(n) + diff == -2)
  - [Perform the check for the mirror case where balance(n) + diff == +2, flipping left/right and -1/+1]
  - Let c = left(n), the taller of the children
  - If balance(c) == -1 or 0 (zig-zig case)
    - rotateRight(n)
    - if balance(c) == -1 then balance(n) = balance(c) = 0, removeFix(p, ndiff)
    - if balance(c) == 0 then balance(n) = -1, balance(c) = +1, done!
  - else if balance(c) == 1 (zig-zag case)
    - rotateLeft(c) then rotateRight(n)
    - Let g = right(c)
    - If balance(g) == +1 then balance(n) = 0, balance(c) = -1, balance(g) = 0
    - If balance(g) == 0 then balance(n) = balance(c) = 0, balance(g) = 0
    - If balance(g) == -1 then balance(n) = +1, balance(c) = 0, balance(g) = 0
    - removeFix(parent(p), ndiff);
- else if balance(n) == 0 then balance(n) += diff, done!
- else balance(n) = 0, removeFix(p, ndiff)

OLD ALTERNATE METHOD

Insert

- Root => set balance, done!
- Insert, v, and look at its parent, p
  - If b(p) = -1, then b(p) = 0. Done!
  - If b(p) = +1, then b(p) = 0. Done!
  - If b(p) = 0, then update b(p) and call insert-fix(p)

Insert-Fix

- For input node, v
  - If v is root, done.
  - Invariant: b(v) = {-1, +1}
- Find p = parent(v) and assume v = left(p) [i.e. left child]
  - If b(p) = 1, then b(p) = 0. Done!
  - If b(p) = 0, then b(p) = -1. Insert-fix(p).
  - If b(p) = -1 and b(v) = -1 (zig-zig), then b(p) = 0, b(v) = 0, rightRotate(p) Done!
  - If b(p) = -1 and b(v) = 1 (zig-zag), then
    - u = right(v), b(u) = 0, leftRotate(n), rightRotate(p)
    - If b(u) = -1, then b(v) = 0, b(p) = 1
    - If b(u) = 1, then b(v) = -1, b(p) = 0
    - Done!