CSCI 136
Data Structures & Advanced Programming

Lecture 28

Spring 2018

Profs Bill & Jon

Administrative Details

- Lab 9 Today: Gardner’s Hex-a-Pawn
    - Another partner lab!
    - Challenging to design & debug
    - Make sure you fill out the form

Last Time

- BST Implementation details:
  - `removeTop`: detaches the root of a tree and returns a valid BST by re-assembling the children
  - `remove`: uses `removeTop` to delete a node and reattach the returned subtree to the parent of the removed node.
  - `add`: because of duplicate nodes, we should recursively call `add`.

Re-corrected: add(E value)

```
public void add(E value) {
        // add value to binary search tree
        // if there's no root, create value at root
        if (root.isEmpty())    {
                root = new BinaryTree<E>(value,EMPTY,EMPTY);
        } else {
                add(root, value);
        }
        count++;
}
```

add(BinaryTree<E> root, E value)

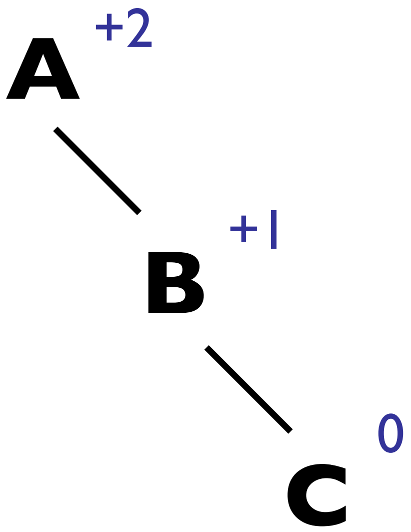
```
public void add(BinaryTree<E> root, E value) {
    BinaryTree<E> insertLocation = locate(root,value);
    E nodeValue = insertLocation.value();
    // The location returned is the successor or predecessor
    // of the to-be-inserted value
    if (ordering.compare(value, nodeValue) > 0) {
        // value > nodeValue
        insertLocation.setRight(new BinaryTree<E>(value,EMPTY,EMPTY));
    } else {
        //value <= nodeValue
        if (insertLocation.left().isEmpty()) {
            // if value is in tree, we insert just before
            insertLocation.setLeft(new BinaryTree<E>(value,EMPTY,EMPTY));
        } else {
            // to properly handle duplicates, add to tree rooted at pred
            add(predecessor(insertLocation), value);
        }
    }
}
```

Demo

- BST add demo

But What About Height?

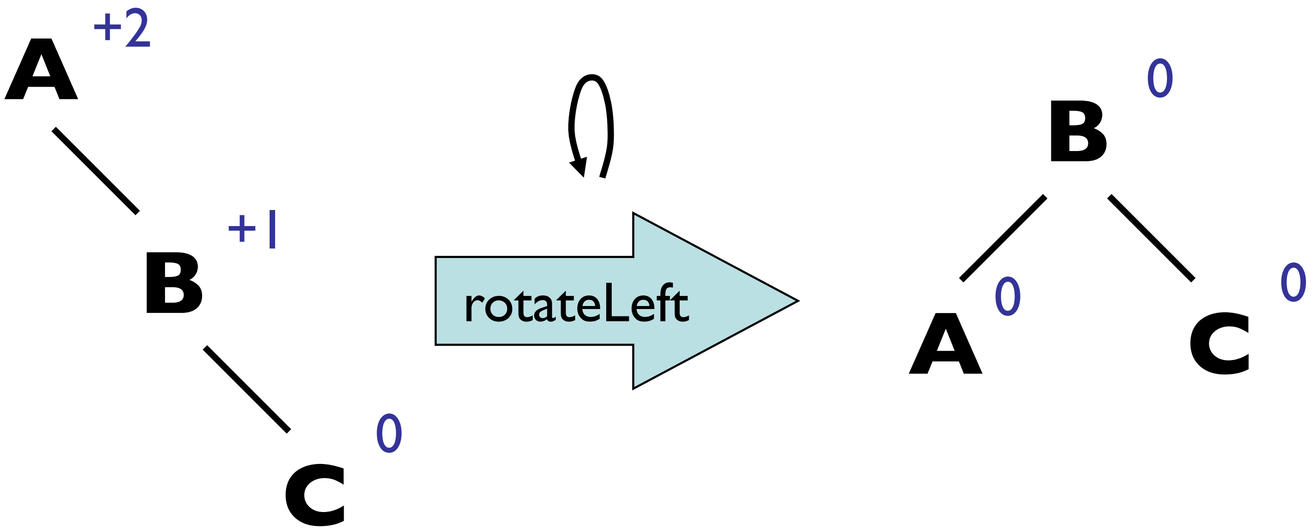
- Operations' performance all depend on *h*
- Can we design a binary search tree that is always "shallow" (minimizes h)?
- Yes! In many ways.
- AVL trees are one example
  - Named after its two inventors, G.M. Adelson-Velsky and E.M. Landis, who published a paper about AVL trees in 1962 called "An algorithm for the organization of information"



- The *balance factor* of a node is the height of its right subtree minus the height of its left subtree.
- A node with balance factor 1, 0, or -1 is considered *balanced*.
- A node with any other balance factor is considered *unbalanced* and requires rebalancing the tree.

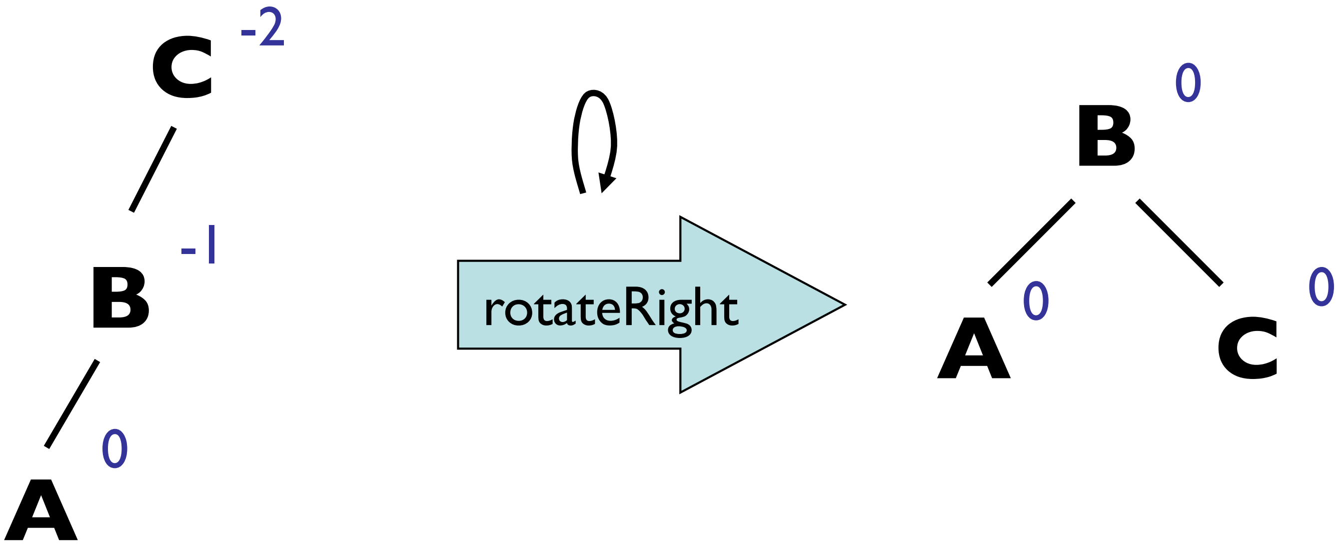
Single Rotation (Left)

Unbalanced trees can be rotated to achieve balance.

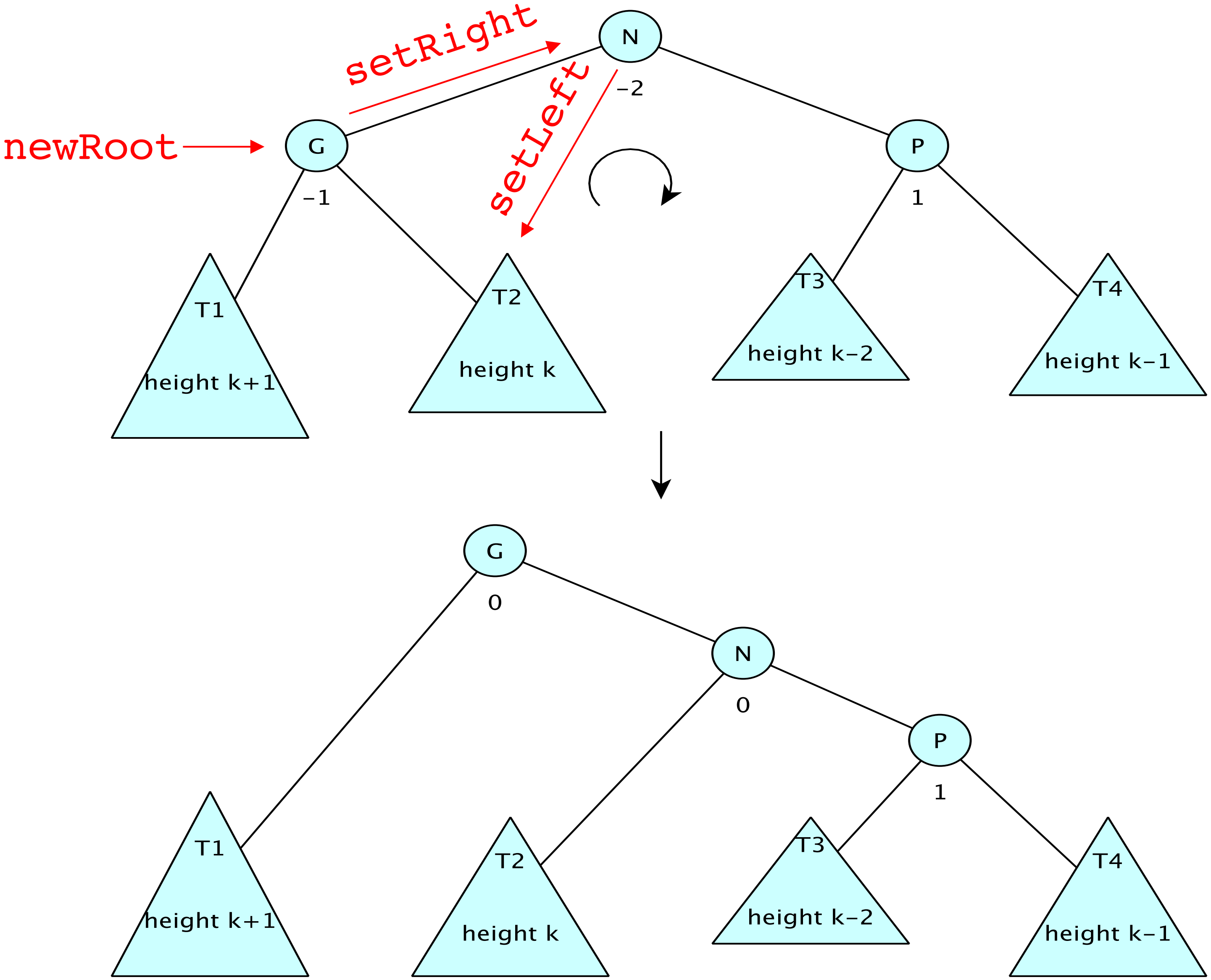


Single Rotation (Left)

Unbalanced trees can be rotated to achieve balance.



Single Right Rotation



BinaryTree rotateRight()

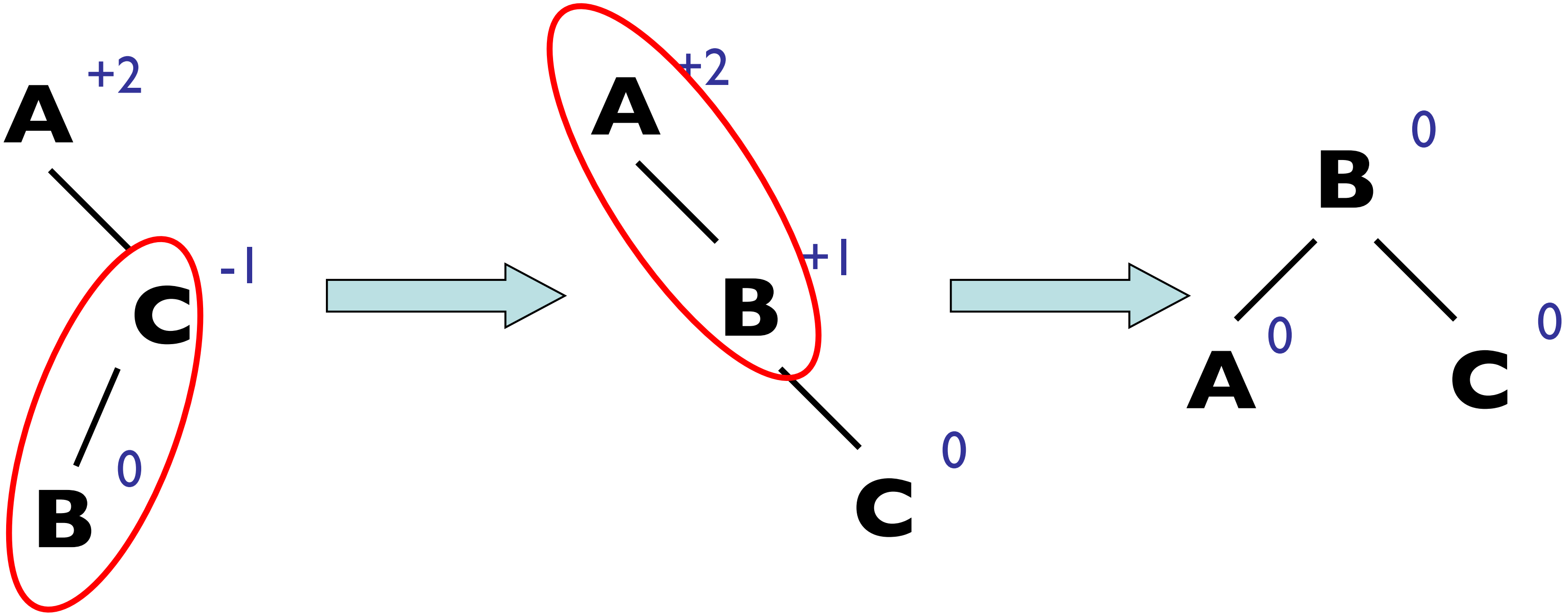
```
// pre: this has a left subtree
// post: rotates local portion of tree so left child is root
protected void rotateRight() {
    // establish pointers/relationships before mucking with the tree
    BinaryTree<E> parent = parent;
    BinaryTree<E> newRoot = left();
    boolean wasChild = parent != null;
    boolean wasLeftChild = isLeftChild();

    // rotate!
    setLeft(newRoot.right());  // hook in new root
    newRoot.setRight(this); // make old root right child of new root
    if (wasChild) {
        // update parent pointers to rotated subtree
        if (wasLeftChild) parent.setLeft(newRoot);
        else              parent.setRight(newRoot);
    }
}
```

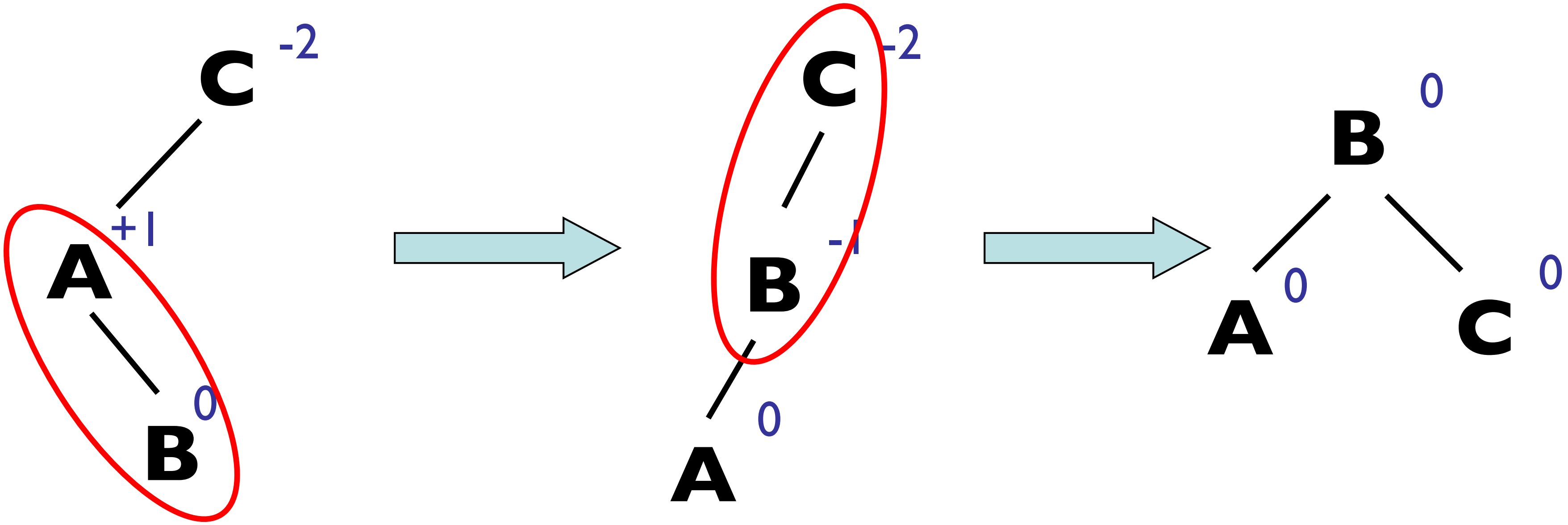
More Complicated Rotations

- Sometimes a single root rotation won’t balance the tree
    - Rotate, then rotate again!
    - We will look at Right-Left and Left-Right

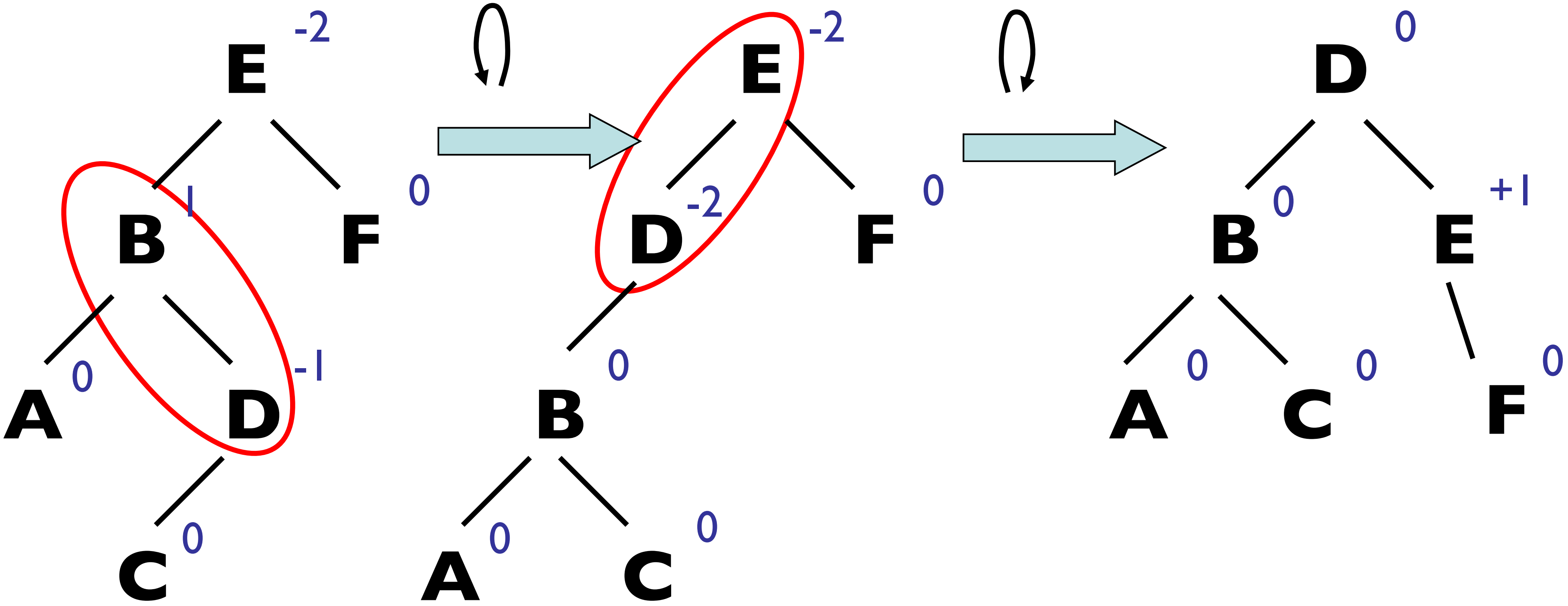
Double Rotation (Right-Left)



Double Rotation (Left-Right)



Double Rotation (Left-Right)



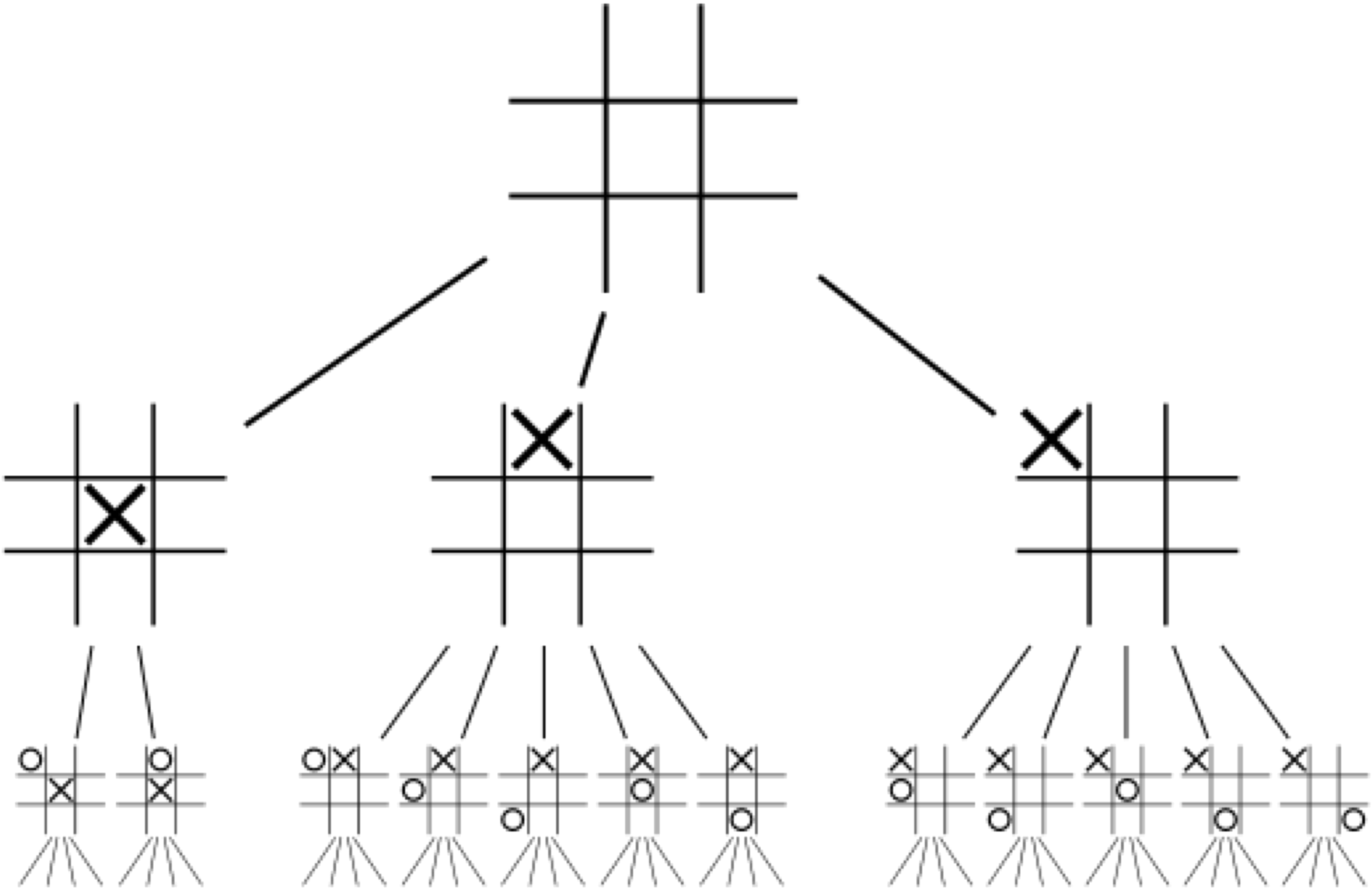
AVL Tree Facts

- A tree that is AVL except at root, where root balance factor equals ±2 can be rebalanced with at most 2 rotations
- add(v) requires at most O(log n) balance factor changes and one (single or double) rotation to restore AVL structure
- remove(v) requires at most O(log n) balance factor changes and O(log n) (single or double) rotations to restore AVL structure
- An AVL tree on n nodes has height O(log n)

AVL Trees: One of Many

- There are many strategies for tree balancing to preserve O(log n) height, including
- AVL Trees: guaranteed O(log n) height
- Red-black trees: guaranteed O(log n) height
- B-trees (not binary): guaranteed O(log n) height
  - 2-3 trees, 2-3-4 trees, red-black 2-3-4 trees, ...
- Splay trees: *Amortized* O(log n) time operations
- Randomized trees: O(log n) *expected* height

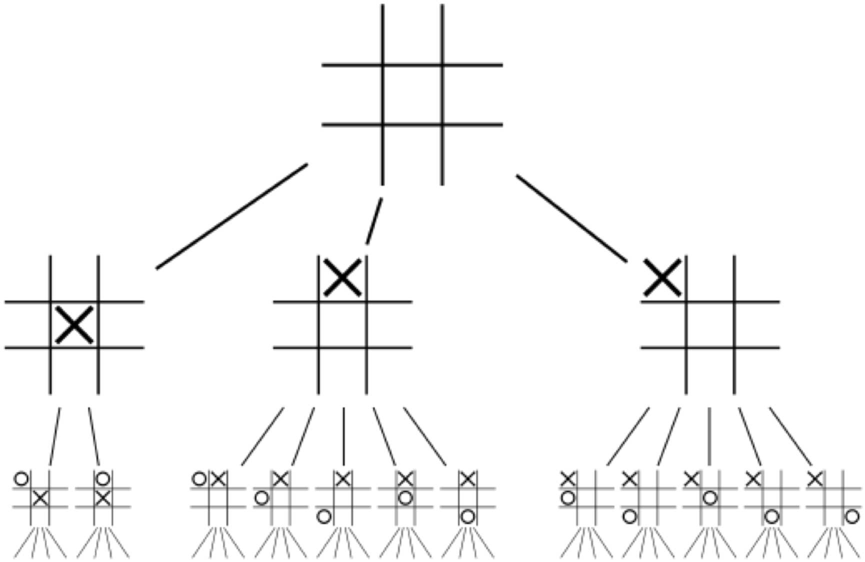
Game Trees



https://upload.wikimedia.org/wikipedia/commons/thumb/d/da/Tic-tac-toe-game-tree.svg/545px-Tic-tac-toe-game-tree.svg.png

Game Trees

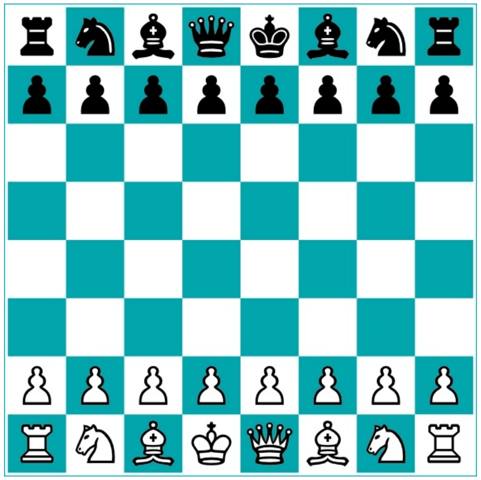
- Nodes are positions in a game (game state)
- Edges are moves (transition from one game state to another)
  - All edges to a given level represent moves by the same player
- Leaf nodes represent ending board states (winner or tie)
  - # of leaf nodes = # of ways a game can be played



https://upload.wikimedia.org/wikipedia/commons/thumb/d/da/Tic-tac-toe-game-tree.svg/545px-Tic-tac-toe-game-tree.svg.png

Game Trees

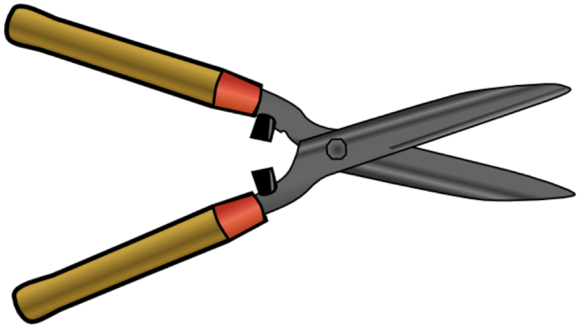
- In AI, often search the game tree and use an algorithm like **minimax** to choose the next "best move"
  - Chess, checkers, tic-tac-toe, etc.
  - What about real-time games?



http://images.all-free-download.com/images/graphiclarge/chess_board_and_pieces_clip_art_23007.jpg
https://bogku.com/product/halo-combat-evolved/

Game Trees

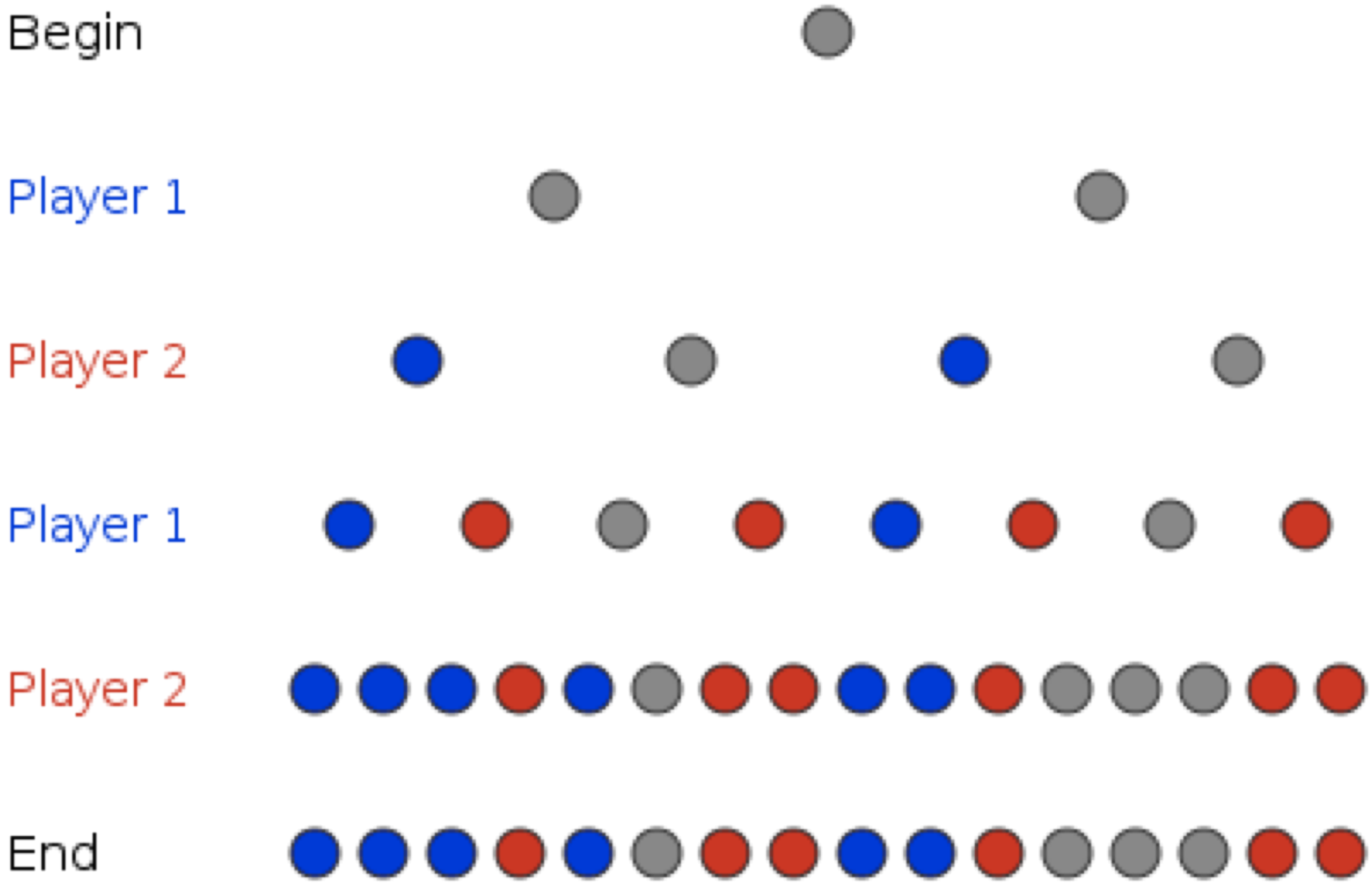
- The **complete game tree**: the root is the initial game state and the tree contains all possible moves from each position
  - You will build complete Hexapawn game trees
  - But your computer player will “prune” the losing branches



Backwards Induction (from Wikipedia)

- Pick 3 colors: player 1 win (P1W), player 2 win (P2W), and tie (T).
- Color leaves (height 0) of the game tree so that:
  - all wins for player 1 are colored P1W,
  - all wins for player 2 are colored P2W,
  - all ties are T.
- Look at height 1 nodes. For each node:
  - If any child is colored for the current player's opponent, color this for the current player's opponent
  - If all children are colored for the current player, color this node for the current player
  - Otherwise, color this node for a tie
- Repeat for each level, moving upwards, until all nodes are colored.
- The color of the root node is the outcome of optimal play.

Backwards Induction Example



https://upload.wikimedia.org/wikipedia/commons/d/d7/Arbitrary-gametree-solved.svg