CS416

Introduction to Computer Science II

Spring 2018

8 Trees and Graphs

Chapter 15

Previously in 416

- Abstract Data Types
  - Specification
  - Stacks, Queues, Lists
  - Dictionary
- Concrete Data structures
  - Implementation
  - Lists, arrays, hash tables

Preview

- *Program State*
- Tree Abstract Data Type
- Tree data structures
  - binary trees
  - n-ary trees
- Tree algorithms
  - Tree algorithm complexity
  - Quadtrees
- Graphs

Binary Search Tree Complexity (review)

- Search for an entry in a binary search tree
  - Best case
    - Tree is <u>balanced</u>
    - Each node comparison eliminates ½ remaining nodes
    - Exactly like binary array search: $O(\log n)$
  - Worst case
    - Tree is a list
    - Same as searching list: $O(n)$
  - Average case
    - Complex analysis based on "average" balance
    - With random build order or re-balancing, it's $O(\log n)$
- There are many algorithms to re-balance trees
  - all need to delete nodes

BinarySearchTree.remove

- Removing nodes from trees is a challenge
  - This version finds a node, then deletes it

```
public Data remove( String str )
{
  Node ret = null;
  ret = findNode( _root, str );
  if ( ret == null )
    return null;
  Data d = ret.data;
  removeNode( ret ); // hard part
  return d;
}
```

removeNode(Node)

- High-level algorithm: three different cases

<u>removeNode(Node n):</u>

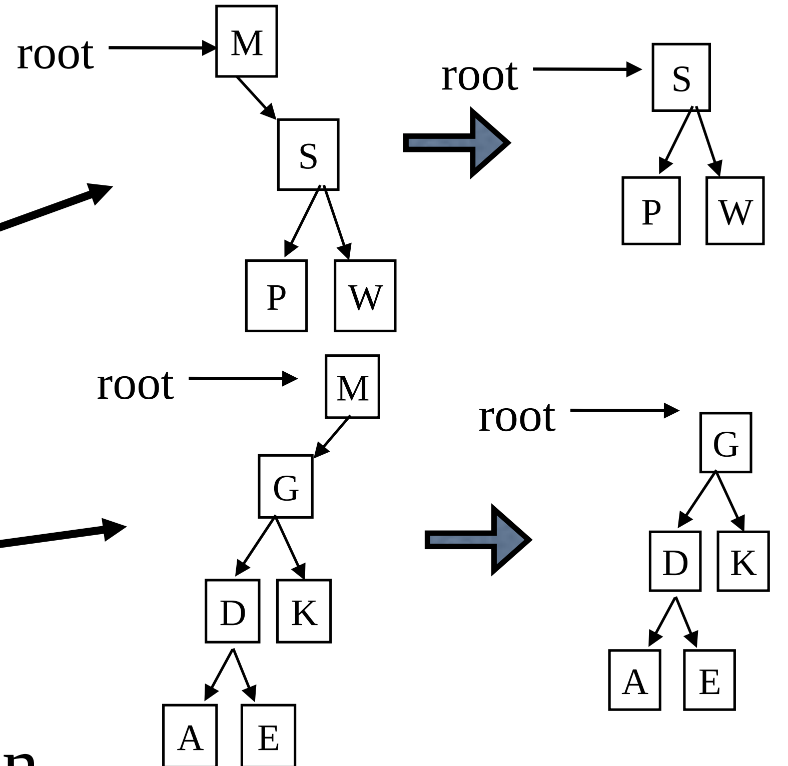
```
  p = n.parent(); // get parent (somehow)
  if ( p == null )    // n is the root of tree
     removeRoot()
  else if p.left == n  // removing parent's left node
     removeLeft( p, n )
  else                        // removing parent's right node
     removeRight( p, n )
```

removeRoot

- High-level algorithm

<u>removeRoot():</u>

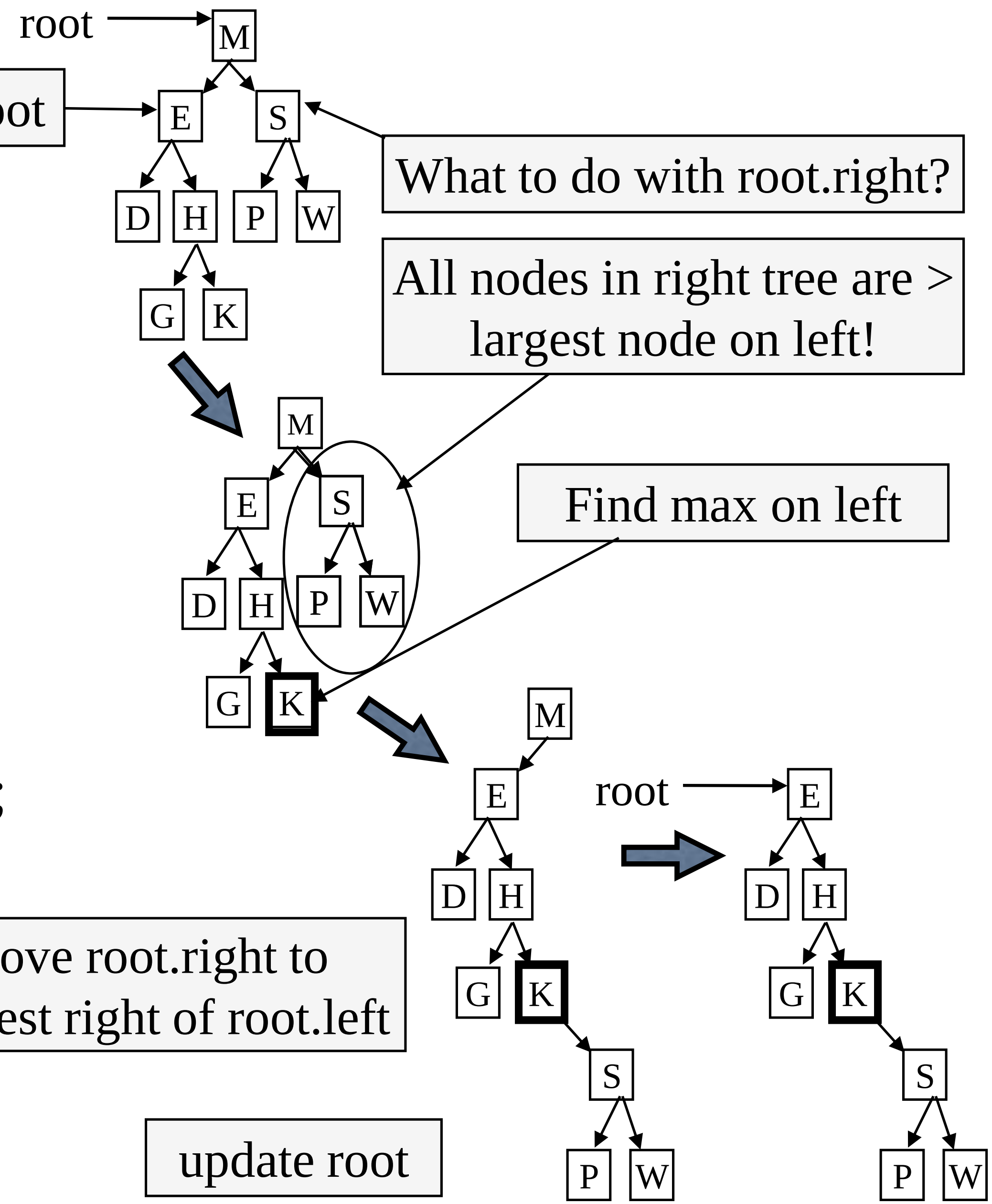
```
  if root.left == null
      root = root.right;
  else if root.right == null
      root = root.left;
  else // hard case, root has 2 children
      // we'll make left the new root
      addToFarRight( root.left, root.right );
 root = root.left;
```



removeRoot (cont)

<u>removeRoot():</u>

```
  if root.left == null
      root = root.right;
  else if root.right == null
      root = root.left;
  else
      // make left the new root
      addToFarRight( root.left, root.right );
root = root.left;
```



Could have chosen the right node as the new root. Best to choose the one that will maintain most balance.

addToFarRight

- The last option can be encapsulated into a method

```
addToFarRight( Node addTo, Node subtree )
  // add the subtree to rightmost descendant of addTo
  while ( addTo.right != null )
      addTo = addTo.right
  addTo.right =  subtree
```

removeLeft

- Algorithm to remove parent.left (E)

<u>emoveLeft(Node p, Node n):</u>

if n.left == null

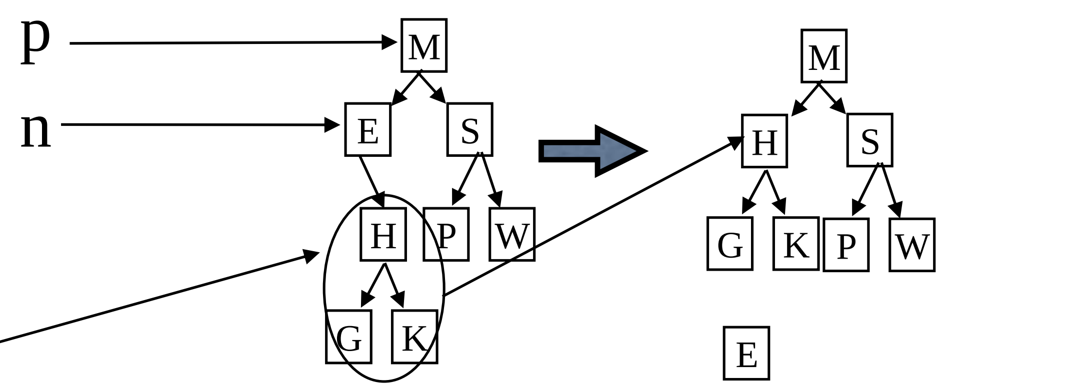
  p.left = n.right

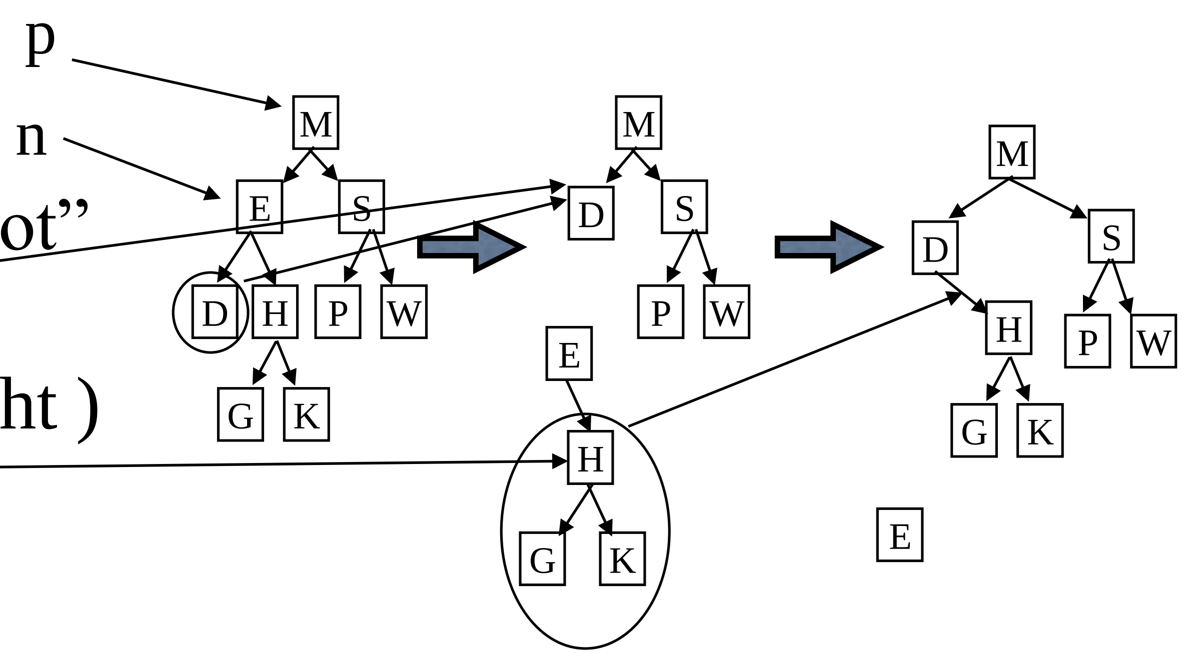
else

  // make left new subtree “root”

  p.left = n.left

  addToFarRight(p.left, n.right)





Could have chosen the right node as the new root. Best to choose the one that will maintain most balance.

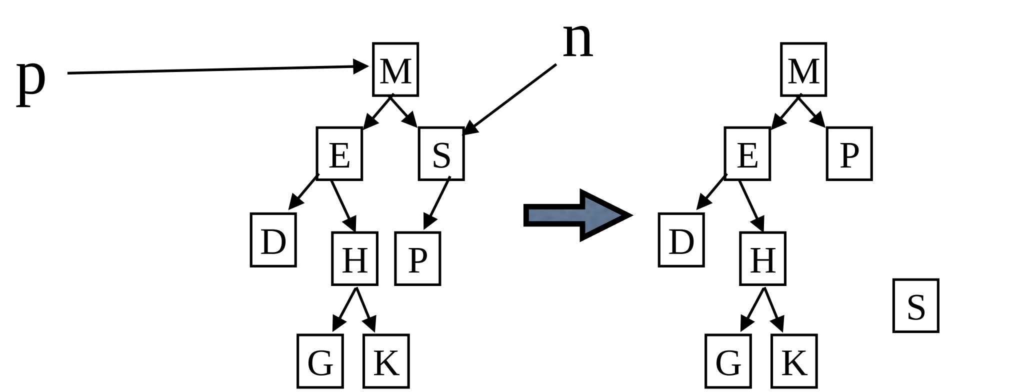
removeNode: left child

- High-level algorithm

```
remove( Node n ):
  p = n.parent
  if ( p == null )
    removeRoot()
  else if p.left == n
    removeLeft( p, n )
  else
    removeRight( p, n )
```

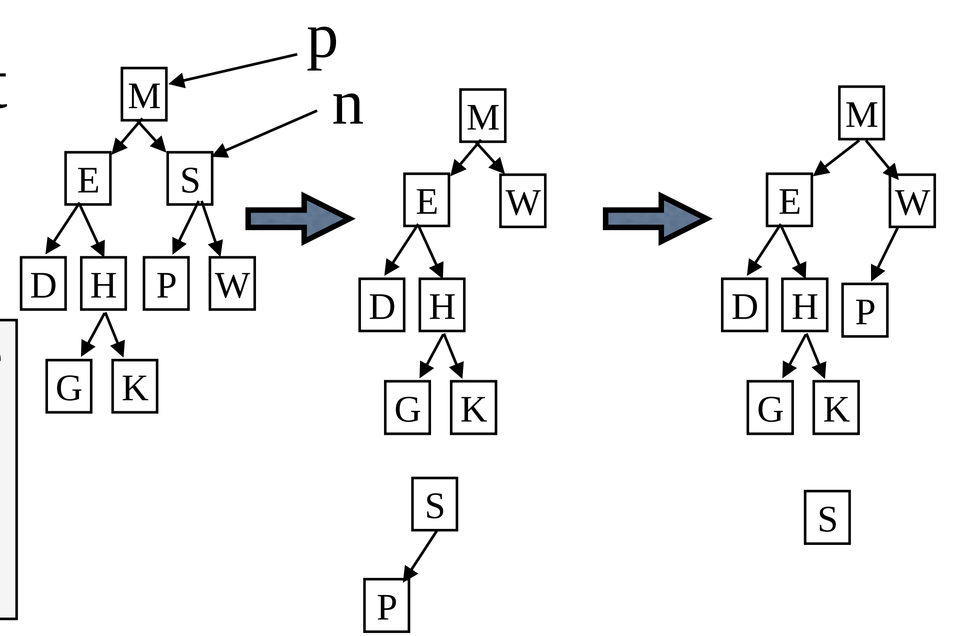
removeRight

- Algorithm to remove parent.right (S)



<u>removeRight(Node p, Node n):</u>

```
if  n.right  == null
    p.right = n.left
else // make right new subtree root
    p.right = n.right
    addToFarLeft( p.right, n.left )
```

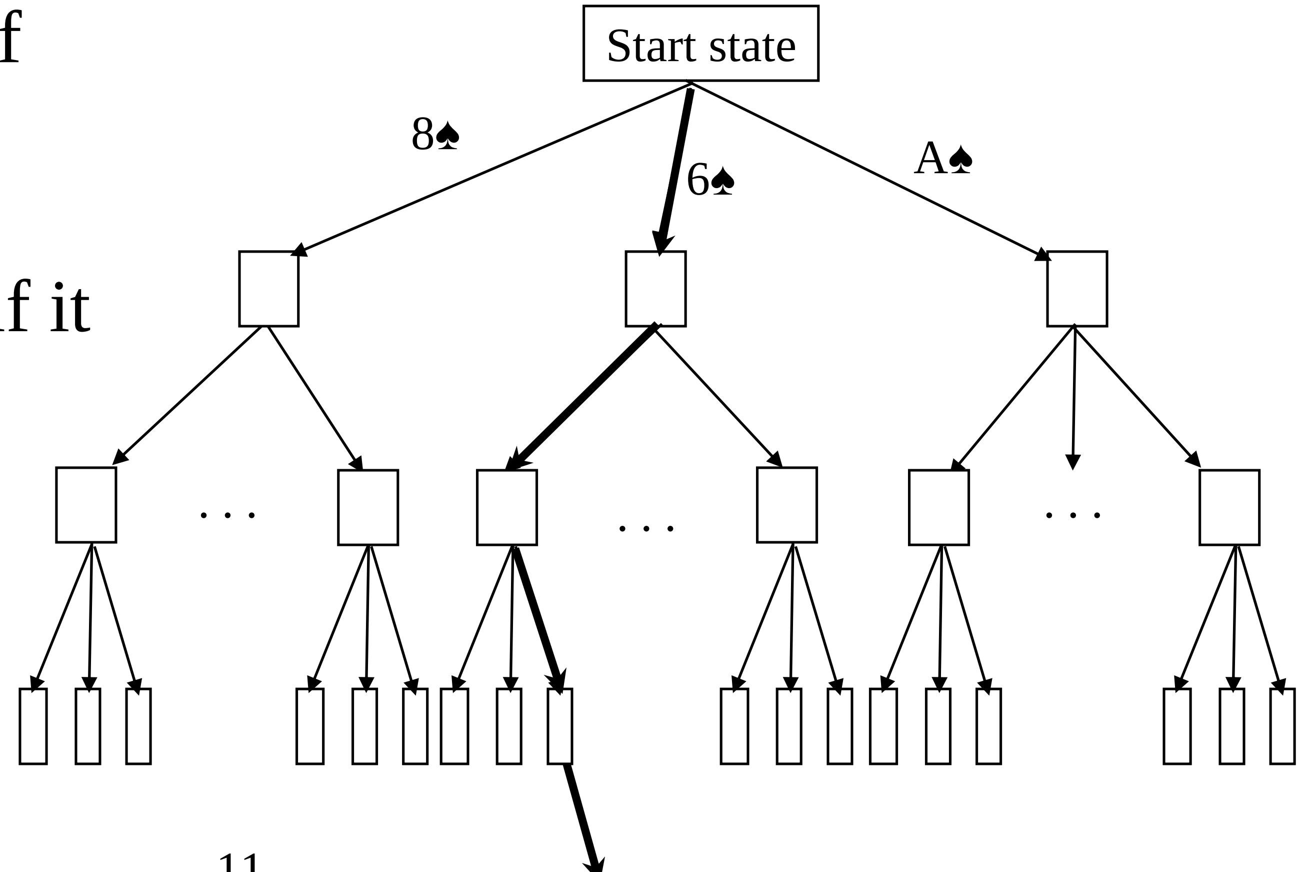


addToFarLeft is nearly identical to *addToFarRight*.

Arbitrary decision to use right here rather than left; again a good choice would be one that is most balanced

State Tree

- A *state representation tree* (or just *state tree*) can represent <u>alternative</u> state changes that occur based on which play option is chosen at any particular point in time
- A *path* from the *root* of the tree to a *leaf* node represents <u>one</u> “game” played to completion, if it is a *complete* tree.



Using a State Tree to Play Solitaire

- Given a deck of cards and rules for a solitaire game
  - shuffle and "deal" the cards
  - simulate playing <u>all</u> possible variations of the game starting in this state
  - the state tree stores all the variations
    - each *leaf* node represents the end of a game: no more moves are possible; a leaf node's score defined by the card's left
    - each interior node is assigned a score that is the maximum of its children's scores
  - play the highest scoring variation: from the root down, choose the play (child) that leads to the highest score

Building a State Tree

- Node associated with a state of the game
  - has 1 child node for <u>each</u> valid move <u>from</u> the node
  - score of the node is the best score of its children
- Leaf node
  - has no valid moves
  - leaf score determined by end state

```
void buildTree():
root = makeNode( startState, null )

Node makeNode( GameState s, Node parent ):
node = new Node( s, parent )
moves = findAllMoves( s )
if moves.size == 0  // base case: leaf node
  node.score = getScore( s )
else
  foreach move in moves
    save state s
    do the move
    child = makeNode( s, node )
    node.children.add( child )
    if child.score is best so far
      bestChild = child
    restore state s
  node.score = bestChild.score
```

State Tree Implementation Issues

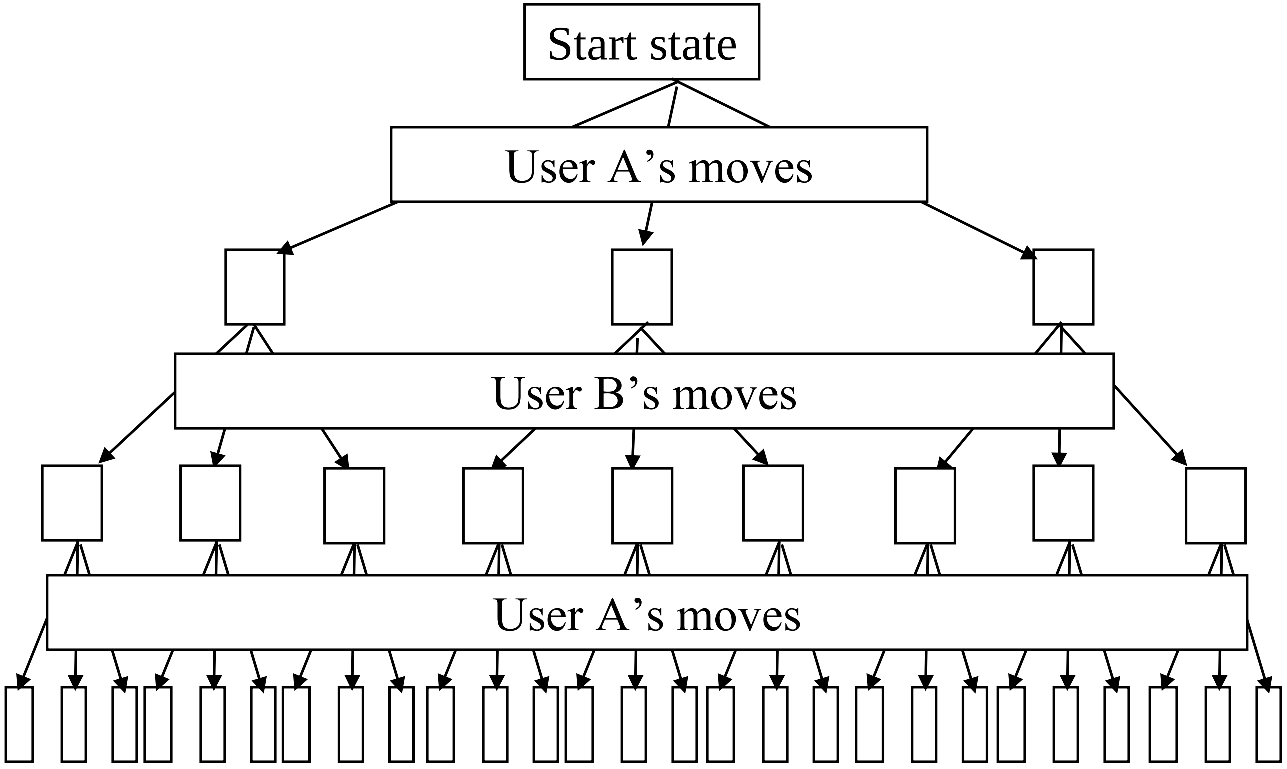
- Some additional issues include:
  - What is a "move"? Who "knows" about it?
  - How do we save and restore the "state" of a game?
    - Make **copies** of all key variables (card piles for a card game)
      - easy to generalize
      - can lead to lots of additional computation
    - Alternatively, save *differences* between states (the *move!*)
      - the "inverse" *move* restores the previous state
      - much more efficient
      - "inverse" moves in some games may be complicated

Game Trees

- *zero-sum* games with *perfect information*
  - one player’s gain is the other’s loss
  - both players know everything about past moves and what the possible next moves are *for both players*
  - checkers, chess, tic-tac-toe, nim
- For such games, we can pre-compute (in principle) a tree representing all possible move combinations
  - Each node in the tree has a child for each possible move
  - Levels in the tree alternate between users

Game Tree Example

- Each node represents a *state* of the game
- Each move changes the state
- Players alternate moves
- Tree building stops when reach a state that ends the game; mark node with winner
- "Know" all possible outcomes



Nim

- Nim is a simple 2-person game
  - throw a bunch of sticks (or rocks, or coins, …) on a table
  - players can pick up 1, 2, or 3 sticks
  - loser is the one who is forced to pick up the last stick

Building the Nim Game Tree

- Root is start state
- Nodes contain *move* information:
  - player
  - sticks removed
  - sticks left

```
void buildTree( root, sticks, depth ):
 root.child0 = makeChild( 1, sticks, depth )
 root.child1 = makeChild( 2, sticks, depth )
 root.child2 = makeChild( 3, sticks, depth )

Node makeChild( pick, sticks, depth ):
 Node node = null
 int sticksLeft = sticks - pick
 if ( sticksLeft >= 0 ) // valid move?
   node = new Node with relevant data
   buildTree( node, sticksLeft, depth+1 )
 return node
```

Can use depth % 2, to assign a user name to node

Playing with a Game Tree

- How does the program use the game tree to play against the user?
  - If it is a complete tree with all possible states
    - avoid making moves that lead to opponent win states
    - choose moves that lead to program win states
  - Sounds good, but what does it mean? What is the code like?
    - *minimax* algorithm

Minimax Algorithm

- We "know" the outcomes of the leaf nodes:
  - program wins (+1), loses (-1), or ties (0)
  - values represent value to the program, not user
- Need to compute values for internal nodes
  - value of a program move node is the *minimum* value of next level user moves (since user will make best move at that node)
  - value of a user move node, is the *maximum* value of its children, since those are program moves and program will pick the best

Note: This is a simple version of minimax.
In most cases, can't afford to build entire tree, so we stop tree creation early and replace a definitive node value with a *heuristic evaluation* function that predicts the value that would come from the lower level subtree.

Minimax Example

- Use + and - for +1/-1
- Assume no ties
- Evaluating tree from A's point of view:
  + is A win, - is A loss
- "Know" score for leaves
- Evaluate A nodes: max(all children)
- Evaluate B nodes: min(all children)

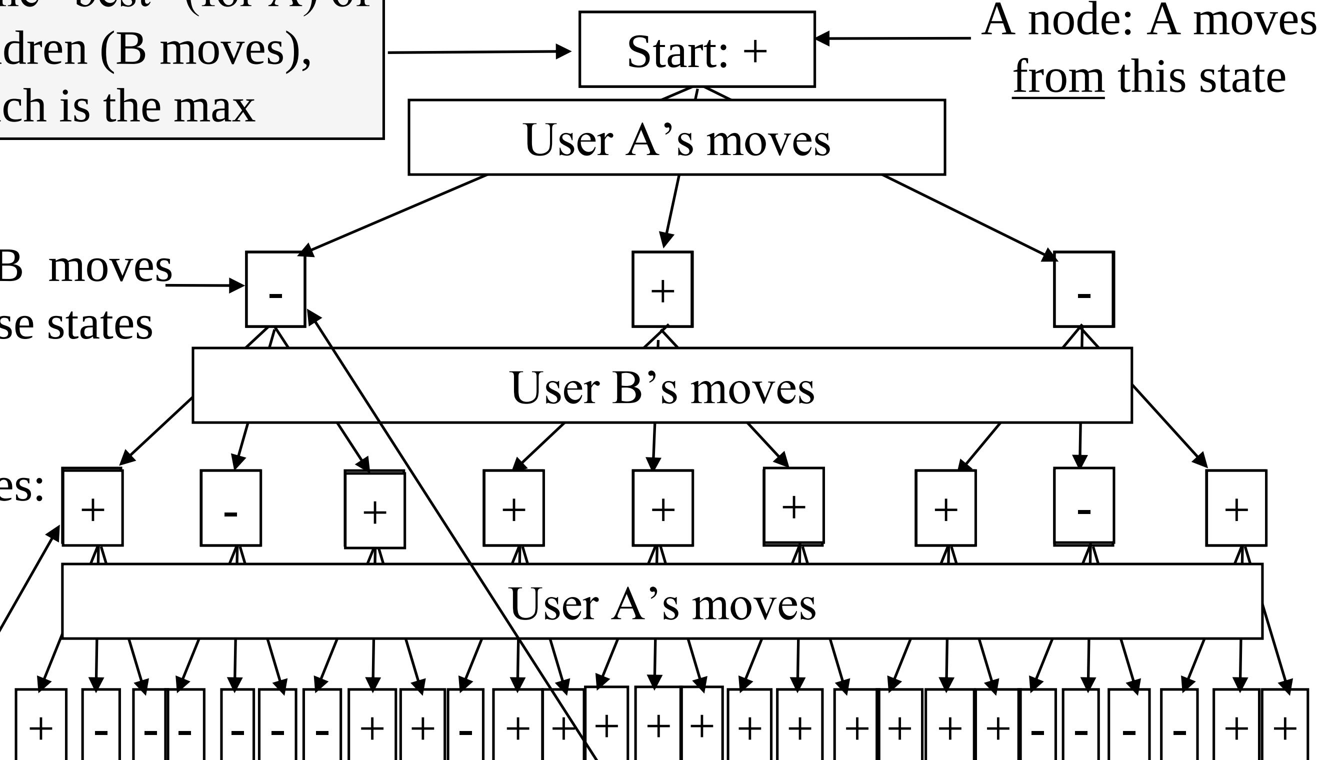
Start node's score is as good for A as the "best" (for A) of its children (B moves), which is the max

A node: A moves <u>from</u> this state

B nodes: B moves <u>from</u> these states

A nodes:

B nodes:



This A node is as good for A as the "best" (for A) of its children (B nodes): the **max**

This B node is as good for A as the "worst" (for A) of its children (B nodes): the **min**

Minimax Algorithm

- Assumptions
  - complete tree
  - no ties
  - +1 for program win
  - -1 for user win

Minimax for Nim

```
int minimax( root, depth ):
  if ( root == null )
    score = 0
  else
    if sticks == 0  // game over
      if user node
        score = 1  // program wins
      else
        score = -1 // user wins
    else  // compute score for this node
      s0 = minimax( root.child(0), depth+1 )
      s1 = minimax( root.child(1), depth+1 )
      s2 = minimax( root.child(2), depth+1 )
      if user node
        score = minIgnore0( s0, s1, s2 )
      else
        score = maxIgnore0( s0, s1, s2 )
    root.setScore( score )
  return score
```

`sticks` value is *after* the move, so 0 for user move means program won

if child is *null*, its score is 0, but need to ignore these values in computing min / max