

CSI 33 LECTURE NOTES (Ojakian)

Topic 8: Trees and Recursion

OUTLINE

(References: Ch. 6, 7)

1. Recall recursion (then apply to linked structures)
 2. Preorder, postorder, inorder
 3. Binary Search Tree
 4. Game Trees?
-

1. Recursion

- (a) Recall some basic examples (see Python examples)
- (b) Wrapper function and Recursive Helper function

PROBLEM 1. *See recursion examples: Fibonacci and list sums*

- (c) Check that recursion works using idea of Proof-By-Induction

PROBLEM 2. *Verify the last recursion examples (Fibonacci and list sums) using Proof-By-Induction.*

2. Recursion on Python Linked Lists

- (a) Pass a node reference to the recursive function
- (b) Function returns a node reference to the properly modified linked structure
- (c) Notice pattern: link passed-in and the link modified are the same
- (d)

PROBLEM 3. *Add a recursive append and delete to linked list, thinking inductively.*

Then verify the operation of the functions using Proof-By-Induction.

3. Expression Tree and Traversals

- (a)

PROBLEM 4. *For example from book (p. 277, Figure 7.4) do three kinds of traversal.*

- (b) Inorder - corresponds infix expression
- (c) Postorder - corresponds postfix expression
- (d) Preorder - corresponds prefix expression
- (e)

PROBLEM 5. *Program in-order. Leave other two as class work or group work or homework.*

4. Binary Search Tree

- (a) Its defining property with examples.

PROBLEM 6. *From diagrams, which are and which are not BSTs?*

- (b) Recursive and non-recursive insertion: Do examples, code, and prove
- (c) Recursive deletion:
 - i. First just code and discuss the case where the deleted node has **at most ONE child**
 - ii. Discuss theory for case: deleted node has TWO children
 - iii. Code must a deleteMax method
 - iv. Then do full delete method

5. Application: Game tree (Maybe do!)

Just do for impartial games.

- (a) Create a class for NON-binary trees by modifying TreeNode to contain a list of references instead of two references.
- (b) Create an **abstract Game** class with methods:
 - i. **moves**: Given a game position, return a list of all legal positions that can be moved to, noting that returning an empty list is a valid return if there are no valid moves from the position
(this method is empty and must be specified by the particular game which is a derived class).
 - ii. **build**: Takes a game position as a start position and builds a game tree to store
(this is done generically in the Game class, but uses the particular method moves).
 - iii. **value**: Determines if the current game is First or Second player win.
 - iv. Could add a mechanism for the computer to play optimally ...