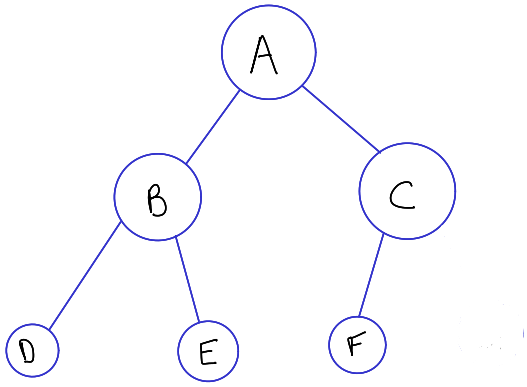
CSC148, Lab #9
week of November 25th, 2013

This document contains the instructions for lab #10 in CSC148. To earn your lab mark, you must actively participate in the lab. We mark you in order to ensure a serious attempt at learning, NOT to make careful critical judgments on the results of your work. We will use the same general rules as for the first lab (including pair programming). See the instructions at the beginning of Lab #1 to refresh your memory.

overview

If we define a **level** in a binary tree as being the nodes that are the same distance from the root, then a **complete binary tree** is a binary tree where all levels, except possibly the last, are full, and nodes are as far left as possible. The illustration below is a complete binary tree, and would remain one if we removed node **F**, but not if we removed **E** and left **F** as it is:

[A, B, C, D, E, F]



In this lab you will design and implement (some of) a complete binary tree data structure, using Python's built-in **list**. Above the tree diagram is an implementation of this tree as a Python list, where the first level is element 0, the second level is elements 1 and 2, and the third level is elements 3, 4, and 5. This data structure is useful as the basis for **heaps**, which (in turn) are used as efficient implementations of **priority queues**, which you will likely see in CSC263.

getting started

Agree on who will be student s1 and who will be student s2 for this lab. For this part, student s1 drives and student s2 navigates.

- Download compbt.py, which declares class **CompleteBinaryTree** that uses a list to store its values, as shown in the diagram. You must implement all the methods below:
- Your constructor should take an optional list parameter to initialize the tree — you should **copy** this list, rather than creating a new name referencing it. Then, discuss the following question with your partner (and other students and your TA, as needed): Why is it a good idea to make a copy of the list argument, for example with slicing (writing code equivalent to: **self.something = parameter[:]** instead of **self.something = parameter**? Write a comment in your constructor to very briefly explain the answer to this question.
- Write "private" methods in class CompleteBinaryTree, in order to insulate later methods from having to touch the underlying list mechanism directly:

  _parent(self, i): return the index of the parent of the node at index **i**, or **None** if there is no parent

  _left(self, i): return the index of the left child of the node at index **i**, or **None** if there is no left child. **i**'s left child **should** be.

  _right(self, i): return the index of the right child of the node at index **i**, or **None** if there is no right child.

  _root(self): return the index of the root of this tree, or **None** if it is an empty tree

  _get(self, i): get the item at index **i**. Raise a **NodeError** if **i** is not a valid index (you will have to create the class **NodeError**)

  _insert_next(self, item): insert item at the next available position in this CompleteBinaryTree.

  Remember to draw lots of pictures and test some examples.

When you are done, show your work to your TA and switch roles.

simple methods

For this part, student s2 drives and student s1 navigates.

Write code for the following methods in class **CompleteBinaryTree**.

inorder(self): return a list of the items stored in this tree, according to a inorder traversal. Try to make your code as generic as possible. In other words, try to write your code so that it depends on the list representation as little as possible, but uses the methods you wrote in the previous step, and perhaps a helper method.

add_new(self, item): insert item at the end of this complete binary tree. The only constraint is that the result must still be a complete binary tree.

When you are done, show your work to your TA and switch roles.

more complicated, but perhaps familiar...

For this part, student `s1` drives and student `s2` navigates.

Write code for the following method in class **CompleteBinaryTree**.

max_sum(self): return the largest sum of the items on a path from the root of this tree down to one of its leaves (the length of the path does not matter, only the value of the sum). **Precondition**: the items in this tree must be numbers, and the **max_sum** of an empty tree is 0.

When you are done, show your work to your TA. Then, please stick around to help other students in your lab section!