CS 234 Spring 2020 — Assignment 3

Instructors: Ten Bradley and Armin Jamshidpey
Due date: July 8, 2020 at 5:00pm

Coverage: Modules 5, and 6
This assignment consists of a programming and written component. Please read the course website carefully to ensure that you submit each component correctly. Assignment solutions are to be submitted to Markus.
Topics: Trees and Graph Interface

1 Programming Component

1. (10 marks) Using the given Contiguous class, implement the class BinaryTree using details from lectures.

   In particular, use the following details:

   - The root of the Binary Tree will be stored at index 0 of the Contiguous.
   - Given that a node is stored at index `i`:
     - The left child is stored at `2i+1`.
     - The right child is stored at `2i+2`.
     - The parent is stored at $\lfloor (i-1)/2 \rfloor$.
     - The Contiguous has size at least 64.
   - If a tree node at a given index does not exist, `None` is stored at that index.

   Submit your implementation as binaryTreeContiguous.py.

2. (10 marks) Using the given BinaryNode class, implement the class BinaryTree using details from lecture.

   In particular, use the following details:

   - BinaryTree has a field `_root` that store None if the tree is empty. Otherwise, it stores a BinaryNode.
   - If a node does not have a parent node, the parent of the node is None.
   - If a node does not have a left child, the left child is None.
   - If a node does not have a right child, the right child is None.

   Submit your implementation as binaryTreeLinked.py.

3. (5 marks) As a user of the Binary Tree class, write the function `traversal(BinaryTree)` that consumes a binary tree and returns a contiguous array containing the items in BinaryTree in in-order traversal order.

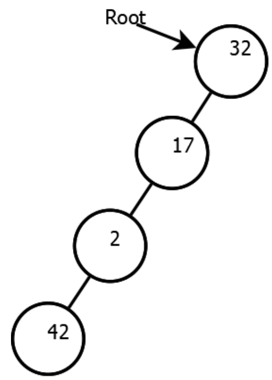
   Your implementation of traversal must be able to be used to traverse any BinaryTree that matches the interface given for P1 and P2.

   Submit your implementation as traversal.py.

Written component on the next page.

2 Written Component

1. Given the following binary tree, answer the following questions.



Each node is left child of the node above it.

   (a) (2 marks) Draw how the tree would be stored using linked memory.

   (b) (2 marks) Draw how the tree would be stored using contiguous memory.

   (c) Consider a binary tree that has `n` nodes stored similarly to the given binary tree, i.e. one node is stored per level. Answer the following questions.

      i. (2 marks) In terms of O-notation, how much memory is required to store the `n` nodes when using linked memory? Justify your answer.

      ii. (2 marks) In terms of O-notation, how much memory is required to store the `n` nodes when using contiguous memory? Justify your answer.

      iii. (2 marks) A tree with very few nodes per level is called a sparse tree. Considering the memory required for linked and contiguous implementations, which implementation would you choose if you know you are storing a sparse tree? Justify your answer.

2. (5 marks) Within an undirected simple graph with 2 or more vertices, there are always at least two vertices with the same degree.

   Write pseudocode for the function `same_degree(G)`, that consumes an undirected simple graph and returns the most common degree in the graph.

   Use the interfaces for Undirected Graph given at the end of this assignment.

   Hint: consider using ADTs we have previously learned to store data.

3. (10 marks) One definition for a tree is a graph with no cycles. Based on that definition, we can write a function Tree_To_Graph that takes an Unordered Tree as a parameter and returns an Undirected Graph.

   Write pseudocode for the function which will create a new instance of an Undirected Graph. All the nodes from the Unordered Tree will be represented as vertices in the graph, and relationships between a node, such as a parent and child node, will be represented as an edge. You may assume that all values stored in the tree are unique.

   Use the interfaces for Unordered Tree and Undirected Graph given at the end of this assignment.

3 ADT Interfaces

3.1 BinaryTree Interface

A NodeID is a unique identified for each node in a Binary Tree.
Preconditions:
For `value, parent, left_child, right_child, set_value`, `ID` is a valid NodeID in `self`.
For `add_leaf`, `ParentID` is a valid NodeID in `self` and `Side` is "Left" or "Right", or `ParentID` is `None` and `Side` is "".

Name	Returns	Mutates
`BinaryTree()`	An empty Binary Tree.	
`is_empty(self)`	True if is empty, false otherwise	
`root(self)`	The NodeID of the root of `self`.	
`value(self, ID)`	The key of the node `ID` in `self`.	
`parent(self, ID)`	The parent node of node `ID` in `self`. If `ID` does not have a parent node, returns False.	
`left_child(self, ID)`	The left child of node `ID` in	If `ID` does not have a left child, returns False.
`right_child(self, ID)`	The right child of node `ID` in	If `ID` does not have a right child, returns False.
`set_value(self, ID, Value)`		Sets the value if the node `ID` to be `Value` in `self`.
`add_leaf(self, Value, ParentID, Side)`	NodeID of inserted node.	Inserts `Value` as the `Side` child of the node `ParentID` in the correct place in `self`. If `ParentID` is None, `Value` is stored as the root node of `self` with no children nodes.

`traversal(BT)` consumes a BinaryTree `BT` and returns a contiguous array that contains the values of the nodes in `BT` in in-order traversal order.

3.2 Pair Interface

A pair is a data structure containing two values.
A pair is created using `Pair(first, second)`
These values are accessed with the first() and second() operations.

3.3 Unordered Tree Interface

A NodeID is a unique identified for each node in an Unordered Tree.
Preconditions:
For `value, parent, children`, `ID` is a valid NodeID in `self`.

Name	Returns	Mutates
`UnorderedTree()`	An empty Unordered Tree.	
`is_empty(self)`	True if is empty, false otherwise	
`root(self)`	The NodeID of the root of `self`.	
`value(self, ID)`	The key of the node `ID` in `self`.	
`parent(self, ID)`	The parent node of node `ID` in `self`. If `ID` does not have a parent node, returns False.	
`children(self, ID)`	A contiguous array of the children of the node `ID` in `self`. If `ID` does not have children, returns an empty contiguous array.	

`traversal(UT)` consumes a UnorderedTree `UT` and returns a contiguous array that contains the values of the nodes in `UT` in in-order traversal order.
Note: This is a reduced interface for an Unordered Tree that does not include operatiosn to modify the tree.

3.4 Undirected Graph Interface

When an edge is returned from a function, it is returned as a Pair containing the IDs of the two vertices the edge is between.

Precondition:
All parameters `ID`, `ID1`, and `ID2` are valid vertices in `self`.

Name	Returns	Mutates
`Graph()`	An empty Graph.	
`is_empty(self)`	True if is empty, false otherwise	
`vertices(self)`	A contiguous array of all IDs of the vertices stored in `self`.	
`edges(self)`	A contiguous array of all the edges in `self`.	
`vertex_value(self, ID)`	The value of the vertex `ID` in `self`.	
`are_adjacent(self, ID1, ID2)`	Returns True if there is an edge between the vertices `ID1` and `ID2`.	
`neighbours(self, ID)`	A contiguous array of all vertices that are adjacent to the vertex `ID` in `self`.	
`set_value(self, ID, Value)`		Sets the value of the vertex `ID` to `Value` in `self`.
`add_vertex(self, Value)`	The ID of the newly inserted vertex.	Creates a new vertex with no adjacent vertices that stores `Value` to `self`.
`add_edge(self, ID1, ID2)`		Adds an edge between the vertices `ID1` and `ID2` to `self`.