Middle East Technical University | Department of Computer Engineering

CENG 242

Programming Languages

Spring 2020-2021

Programming Exam 7

Due date: 12 June 2021, Thursday, 23:59

1 Problem Definition

In this exam, you are going to construct some n-ary trees and handle some operations on them.

1. The problem starts with the construction of the trees. For this, you are given some <parent-child> relationships in an **unordered way**, and you are expected to construct the corresponding tree(s) properly. For example:

```
addRelation(18, 35)
addRelation(20, 17)
addRelation(10, 30)
addRelation(10, 44)
addRelation(80, 100)
addRelation(35, 43)
addRelation(20, 19)
addRelation(65, 26)
addRelation(48, 60)
addRelation(10, 12)
addRelation(65, 10)
addRelation(48, 70)
addRelation(20, 18)
addRelation(17, 16)
addRelation(48, 80)
addRelation(18, 42)
addRelation(48, 90)
```

```
    The parent-child sequence given on the left
results in 3 independent trees:

tree-1:    20            tree-2:    65
         /   |   \                  /    \
        17   19   18               26     10
        |        /  \                   /  |  \
        16      35   42               12   30  44
                |
               43

tree-3:    ____48____
         /   /  \    \
        60   70   80   90
                  |
                 100
```

   As you see, child may be introduced before its parent in the input. It is ensured that any independent tree have unique nodes (i.e. distinct nodes from those of the other trees). Also, there will be no repetition in the input (like giving the same parent-child relationship). Moreover, ordering between the children of a node is not important.

2. **Node.cpp:** `Node` is the most fundamental class in this task. It represents a node holding a unique id and it is actually a tree at the same time whose starting node is itself and may have some children and a parent (if it is itself a child of another tree).

```
                ?          may have parent
                |
Node:     [ int id ]
          / / .... \ \
         ?  ?  ...  ? ?    may have children
```

This class has some some basic methods like constructor, destructor, copy cons., operator overload, etc. Each one is explained in the header file in a detailed way.

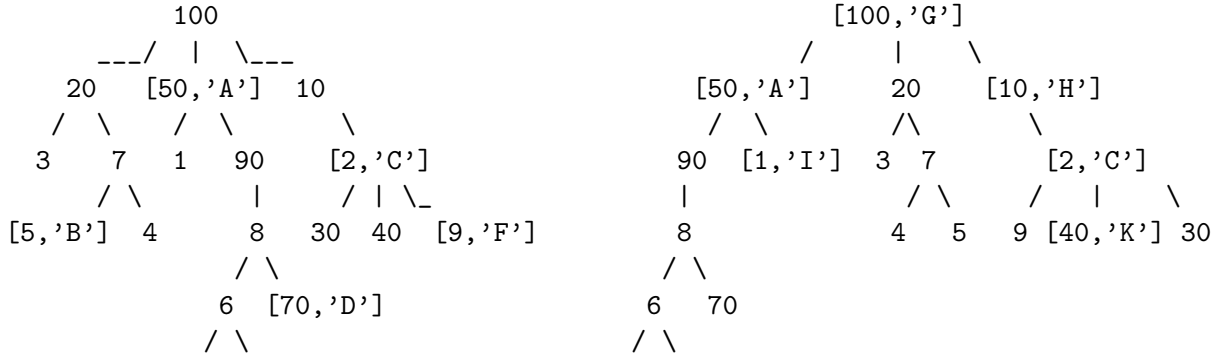
(a) **DataNode.cpp:** `DataNode` is a derivation of `Node` class. The difference from `Node` is that it also carries a char value other than id. Except than having a char data, it has the same properties with `Node` class. However, the implementation of some methods of `DataNode` may slightly differ from those of `Node`. These details are given in the header.

3. **NodeManager.cpp:** `NodeManager` is the class carrying the independent tree pieces (i.e. independent `Node` objects). This class also has 3 important methods other than constructor/destructor:

   - First, it provides to add new <parent-child> relationships via `addRelation(int parent.id, int child.id)` method. An illustration of sequential calls of this method is given above, at article 1.
   - Second, it enables to convert the type of a `Node` object into `DataNode` object via `setDataToNode(char data, int node.id)` method in which you need to find the related method and replace it with a new node of `DataNode`.
   - Third, it has `getNode(int node.id)` method which simply returns the corresponding node.

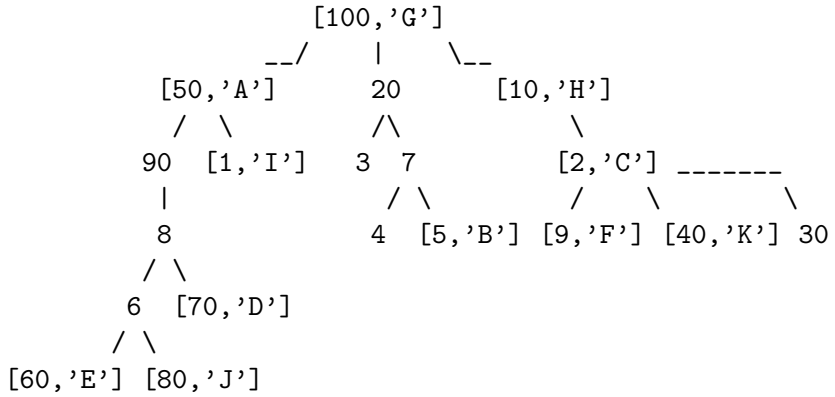
4. **Action.cpp:** `Action` is an abstract base class which inherits a single method named `act(const Node& node)` to its derivations. How the action becomes depends on the derivated class, yet we can say that it conducts some operations on the given node and returns the output tree.

   (a) **CompleteAction.cpp:** `CompleteAction` is a derivation of abstract `Action` class. Its constructor saves a `Node` object as its member. It derives the `act(const Node& node)` method as follows: The node object given in the parameter and the one kept as member are compared along their children one by one and produced a new tree (`Node` object). It is ensured that both input trees (namely `Node` object) has the same structure and nodes with corresponding ids. If there is object of `Node` type at the corresponding positions of trees, there is nothing special to do. The output tree will have the same `Node` object in that case. However, there may be `DataNode` objects at some places in the first tree instead of some bare `Node` objects whereas they are simply a `Node` object in the second tree, or vice versa. Then, in such a case the output tree will have the `DataNode` object at that position. Below, `act()` operation of `CompleteAction` class is illustrated:



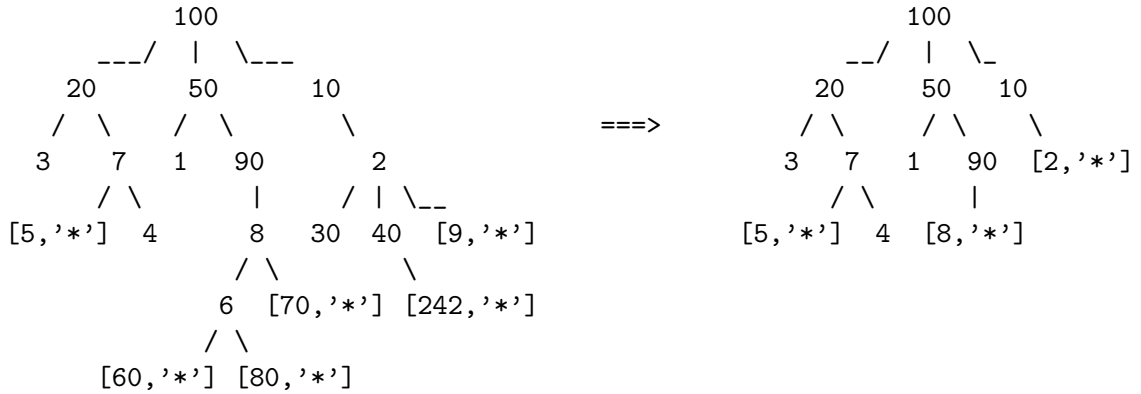
(tree1) (tree2)

Assume that a CompleteAction object was constructed previously by calling CompleteAction(tree1). After act() method of CompleteAction class is called with the tree2 as its argument, it is expected to return tree3 below:



(tree3)
(The ordering of the siblings is not important)

(b) **CutAction.cpp:** `CutAction` is a derivation of abstract `Action` class. Its constructor saves a char value as its member, say `data`. It derives the `act(const Node& node)` method as follows: It applies some cut operation on the nodes of the argument tree that satisfy a basic rule. If a node has at least 2 char value which is same with the `data` at the total of its children and grandchildren, then that node is cut such that in the output tree it is replaced with a new node of type `DataNode` which has `data` as its char value and has no child. **Note that cut operation should start from the leaves and proceeded towards the root (i.e. from down to up). If you do it in the reverse manner, the result may change!** Below, `act()` operation of `CutAction` class is illustrated:



(tree1) (tree2)

Assume that a CutAction object was constructed previously by calling CutAction('*'). After act() method of CutAction class is called with

```
the tree1 as its argument, it is expected to return the tree2.
(The ordering of the siblings is not important)
```

(c) **CompositeAction.cpp:** `CompositeAction` is a derivation of abstract `Action` class. This action is the composition of the other actions. It allows actions to be added using the `CompositeAction* addAction(const Action* action)` method (notice that this method returns a reference to the object itself, i.e., return the "this" pointer, for syntactic purposes). Note that CompositeAction is not responsible for deleting child actions when it is destructed. You may assume that there will be at least one child action added to a CompositeAction object.

5. **Exception.h:** This is implemented by the author and directly supplied to you. There is nothing that you need to implement on it. It includes an `InvalidRequest` exception which you may need in your implementations when any data is queried from a `Node` yet not `DataNode` object.

2 Regulations

1. **Implementation and Submission:** The template files "*.h" and "*.cpp" are available in the Virtual Programming Lab (VPL) activity called "PE7" on OdtuClass. At this point, you have two options:

   - You can download the template files, complete the implementation and test it with the given sample I/O on your local machine. Then submit the same file through this activity.
   - You can directly use the editor of VPL environment by using the auto-evaluation feature of this activity interactively. Saving the code is equivalent to submit a file.

   The second one is recommended. However, if you're more comfortable with working on your local machine, feel free to do it. Just make sure that your implementation can be compiled and tested in the VPL environment after you submit it. "*.h" and "*.cpp" files are given to you so that you can work on your local machines. If you choose first option, you have to submit these files as well but they will not be included into evaluation process. There is no limitation on online trials or submitted files through OdtuClass. The last one you submitted will be graded.

2. **Programming Language:** You must code your program in C++. Your submission will be compiled with g++ on VPL. You are expected to make sure your code compiles successfully with g++ using the flags -ansi -pedantic.

3. **Cheating:** We have zero tolerance policy for cheating. People involved in cheating (any kind of code sharing and codes taken from internet included) will be punished according to the university regulations.

4. **Evaluation:** Your program will be evaluated automatically using "black-box" technique so make sure to obey the specifications. No erroneous input will be used. Therefore, you don't have to consider the invalid expressions.

   Important Note: The given sample I/O's are only to ease your debugging process and NOT official. Furthermore, it is not guaranteed that they cover all the cases of required functions. As a programmer, it is your responsibility to consider such extreme cases for the functions. Your implementations will be evaluated by the official test cases to determine your actual grade after the deadline.