CSE221: Data Structures
Programming Assignment 3
Due on Thursday 25 November, 23:59

Antoine Vigneron

1 Introduction

In this assignment, you are asked to:

- Implement a general tree class as described in Lecture 13. In particular, it is *not* a binary tree class, so an internal node may have 1, 2, 3 or more children. This will be a template class, so the elements stored at the nodes could be of any class.
- Implement an algorithm that solves the minimum vertex cover problem on such a tree.

2 Implementation

You are given a header file `tree.h`, which gives the declarations of the classes and the main functions. You should complete it by writing a definition of all of these functions. You are only allowed to edit the part of the file below the comment "TYPE YOUR CODE BELOW".

You should implement all the member functions (except the default constructors which are already given), as well as the function that returns the cost of a minimum vertex cover.

3 Tree class

The tree class is very similar to the one presented in Lecture 13, and thus Sections 7.1 and 7.2. Lecture 14 gives a detailed implementation of a *binary* tree class, so you may want to look at it again before doing this assignment.

Each node records its children as an STL list of positions. In particular, it will allow you to use iterators on this list of children. (See Lecture 12.) A node also records a pointer to its parent, and an element of type E.

The two member variables of a tree are a pointer `_root` to its root, as well as its size (number of nodes) `_n`.

The default constructor makes an empty tree by setting the root pointer `_root` to NULL and the size to `_n` to 0.

Then more nodes can be created as follows:

- `addRoot(const E & e)` makes a root node containing the element `e` for an empty tree. `_root` will now point to this new node, whose parent is NULL.
- `insertAt(const Position & p, const E & e)` inserts a new node containing element `e` as a child of the node at position `p`. The position of the new node should be placed *at the end* of the list of positions of the children of `p`.

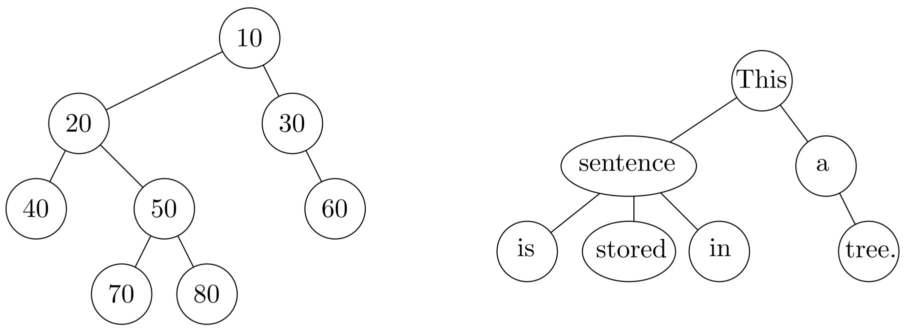

Figure 1: Two trees recorded in the files `instance3.in` and `instance8.in`.

3.1 Tree traversal

The three member functions below are related to tree traversal:

```
PositionList positions() const;
void print();
void preorder(Position p, PositionList& pl) const;
```

`preorder` computes a list of the positions of all the nodes of the subtree rooted at position *p*, and appens it at the end of the list `pl`. See Lecture 14 for an analogous function on binary trees.

`print()` prints the nodes of the tree in preorder to the standard output `cout`. It should work for both trees of integers and trees of strings (i.e. `E=int` and `E=string`). So for the trees shown in Figure 1, it should print the following:

```
10 20 40 50 70 80 30 60
This sentence is stored in a tree.
```

There should be a single space between any two elements, and a linebreak at the end.

`positions` produces a list containing the positions of all the nodes, in preorder. So the nodes should appear in the same order as above for the trees in Figure 1.

3.2 Constructor and destructor

You need to implement the constructor Tree(string filename) that builds a tree from an input file. Examples of input files are given in the instances subdirectory. An input file looks as follows:

```
8
0 10 -1
1 20 0
2 40 1
3 50 1
4 70 3
5 80 3
6 30 0
7 60 6
```

This file records the tree shown on Figure 1 (left). The first line shows the number n of nodes. Then the next lines follow this pattern:

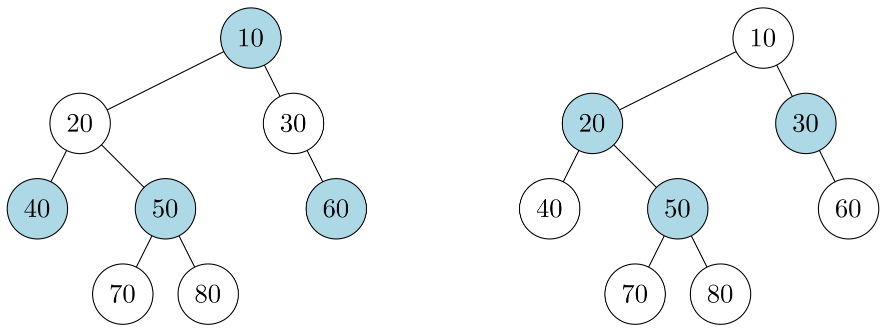

Figure 2: Vertex covers of the tree stored in `instance3.in`. (left) A vertex cover C of weight 160. (right) The minimum vertex cover C^* with weight 100.

```
node_index element parent_index
```

The indices go from 0 to $n-1$, in preorder. The root has index 0. The parent of the root is set to -1.

You also need to implement the destructor. It should be done in such a way that you don't have memory leaks.

3.3 Exceptions

If `addRoot` is called on a non-empty tree, it should throw an exception. If `parent()` is called on the root position, it should also throw an exception. In both cases, throw a standard `runtime_error`.

3.4 Remarks

One issue that you may encounter is that the C++ compiler struggles to parse some type names, due to the templates. You may be able to fix the problem by adding the keyword `typename` before. Here is an example from the attached test file `test3.cpp`:

```
T = new typename Tree<int>::Tree(filename);
```

If you remove `typename`, you get the compilation error:

```
g++ -c test3.cpp
test3.cpp: In function ‘int main()’:
test3.cpp:17:14: error: expected type-specifier
   17 |     T = new  Tree<int>::Tree(filename);
      |              ^~~~~~~~~
make: *** [Makefile:13: test3.o] Error 1
```

4 Vertex Cover

A *vertex cover* C of a graph is a subset of the vertices such that each edges has at least one its two vertices in C. When the elements stored at the nodes are numbers, a *minimum vertex cover* C^* is a vertex such that the sum of the weights (i.e. the elements stored at the nodes) is minimum. (See Figure 2.) You need to implement the function

```
int vertexCover(const Tree<int> & T);
```

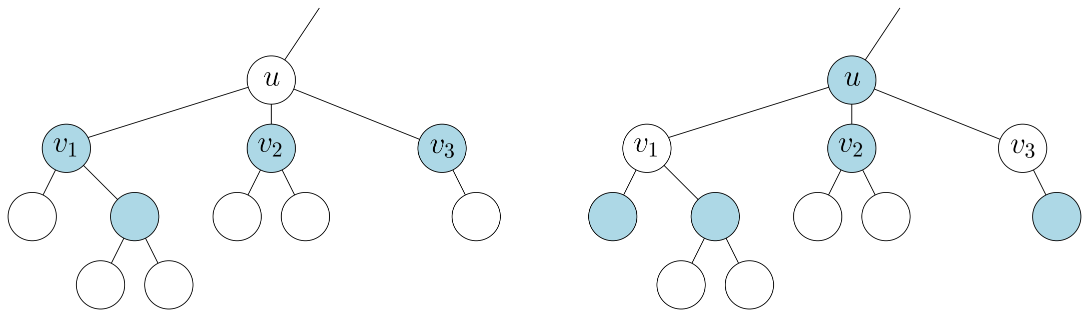

Figure 3: (left) Since u is not in the vertex cover, then its three children v_1, v_2 and v_3 are in the vertex cover. (right) Since u is in the vertex cover, its children may or may not be in the vertex cover.

that returns the *weight* of a minimum vertex cover of the tree T. So in the example from Figure 2, it should return 100. You only need to make it work for trees of integers, as all our test cases use integers, but it should also work for any type of numbers.

4.1 Solution by dynamic programming

This problem can be solved by trying all the possible subsets of nodes of T, and keeping the one that minimizes the weight. Unfortunately, there are 2^n such subsets so this algorithm would run in exponential time. Here we suggest that you use a *dynamic programming* approach, where solutions to sub-problems are recorded, and combined to obtain the solution to the original problem.

We now explain this approach. For any node u, we denote by T_u the subtree rooted at u. We denote by $w(u)$ the weight of u. So $w(u)$ is just the number (i.e. the element) stored at node u.

We denote by $C(u, 0)$ the minimum weight of a vertex cover of T_u such that u is *not* in the vertex cover. We denote by $C(u, 1)$ the minimum weight of a vertex cover of T_u such that u is in the vertex cover. Then these two quantities satisfy the relations:

$$C(u,0) = \sum_{v:\ v \text{ is a child of } u} C(v,1) \tag{1}$$

$$C(u,1) = w(u) + \sum_{v:\ v \text{ is a child of } u} \min(C(v,0), C(v,1)). \tag{2}$$

The reason is the following. If u is not in the vertex cover, then all of its children must be in the vertex cover, so that all the downward edges from u are covered. (See Figure 3.) So we obtain $C(u, 0)$ by summing up $C(v, 1)$ over all the children v of u.

On the other hand, if u is in the vertex cover, then a child v of u may or may not be in the vertex cover. We pick the best choice, which is $\min(C(v, 0), C(v, 1))$. So overall, $C(u, 1)$ is obtained by summing up the weight $w(u)$ of u with $\min(C(v, 0), C(v, 1))$ for all its children v.

So in order to solve the problem, you can compute recursively the values $C(u, 0)$ and $C(u, 1)$ for all the nodes u. The solution to the vertex cover problem is then $\min(C(r, 0), C(r, 1))$ where r is the root of the tree.

5 Testing your program

Checking programs for all the functions are provided in the ”code.zip” file. Please read the README.txt file that is provided. It is highly recommended that you run all the tests on the server as your own configuration may differ in certain aspects. (A recent Ubuntu installation should work though.)

Input files recording tree instances are also given in the subdirectory ”instances”. They are used by the checking programs.

6 Submission

You should submit your code by simply uploading your `tree.h` file to your **home directory** on the server.

7 Grading

Your program will be graded automatically. Make sure that the provided test programs compile and run without error on your code. If not, it will make it considerably more difficult to grade your program, and there will be a 10% penalty. If your program does not compile, you may want to comment out the functions that cause it not to compile.

Late submissions up to one day after the deadline will be penalized 15%, and 30% two days after the deadline. After that, your assignment will not be graded (hence you get 0).

The grading program we will use in the end may differ from the ones that have been/will be provided. In any case, some of the testing instances will be different.

8 Other rules

The general rules for the CSE department, given at the beginning of the semester (see course information folder), apply to this assignment. You may contact one of our two TAs for help:

- Seonghyeon Jue (`shjue@unist.ac.kr`)
- Hyeyun Yang (`gm1225@unist.ac.kr`)

Hyeyun Yang will be grading this assignment.

9 Typos and bugs

If you find typos in this handout, or bugs in the code that is being supplied, please send email to the TAs or to me.