AVL-Tree Visualization using Observer and MVC Patterns

Student: Ryazanov Stanislaw
Supervisor: Trushin Dmitry

Higher School of Economics
Faculty of Computer Science
Moscow, Russia
June 4, 2023

Contents

Annotation

My project of choice is an application that implements am AVL-Tree and visualizes it via QWT or SFML. In this paper I will give a brief overview of the work done with code snippets and my comments. You can find the project on this Github page.

Keywords

Software development, design patterns, MVC pattern, Observer pattern, binary tree, AVL-Tree

1 Introduction

This project is an application that implements an AVL-Tree and visualizes it via SFML using Observer and MVC patterns. The AVL-Tree was implemented using a Node structure, implying that each individual entry in the tree is a separate object that stores a value and pointers to its children. Open-source multimedia library SFML was used for drawing the data structure on the screen. Talking of the MVC pattern, the AVL-Tree structure itself was used as a Model, the application it ran on was used as a Controller and the SFML renderer played the role of the Viewer. The connection between the Model and the Viewer was implemented via the Observer pattern, where Viewer was the observer and Model was the observable. For a more detailed review of my project read the rest of this paper.

2 Breakdown

2.1 The Application pattern

As in any worthy C++ project, the main.cpp file looks quite minimalistic:

```cpp
#include "Application.hpp"

int main()
{
    Project::Application app;
    app.execute();
}
```

Here in the main() function one may see an object of class Application being created and executed. This object handles all the logic behind the application: the AVL-Tree, input of commands and output of the tree. This pattern is used to not over-complicate the main execution file for further use.

2.2 The MVC pattern

MVC is a popular application design pattern that provides security and organization to the code. The letters in MVC abbreviation stand for "Model, View, Controller", where the Model handles the business-logic behind the program, View displays the Model to the user and Controller inputs commands and acts as a intermediary between the user, the Model and the View.

2.2.1 The Model

In our case the Model is an implementation of an AVL-Tree. I chose to create a Node-based implementation since its easier to edit trees that way. In Node.hpp file I've created the Node class:

```cpp
class Node
{
   public:
    Node(int x) : key(x), height(1), right(nullptr), left(nullptr) {};
    int key;
    Node* right;
    Node* left;

    int height;
```

```
    int updateAllHeights();
};
```

Each Node has a set of parameters: the key, which is a number assigned to it, the height of the Node in the tree and pointers to left and right 'children' Nodes. We also have an updateAllHeights() method, which updates heights of all Nodes connected to a given Node.

Now to the tree itself, here's the AVL-Tree class code:

```
class AVLTree
{
   public:
    AVLTree(Observer<pair<vector<string>, Node*>>* obs) :
        root_(nullptr) { out_.subscribe(obs); };

    AVLTree(const AVLTree&) = delete;
    AVLTree& operator=(const AVLTree&) = delete;
    AVLTree(AVLTree&&) noexcept = delete;
    AVLTree& operator=(AVLTree&&) noexcept = delete;

    // Frames
    void updateText(std::string command);
    void updateScreen(Node* root, std::string command);
    void closeWindow();
    void resize(int width, int height);

    // Utilities
    void setCmdCommand(std::string s);

    // Observer
    void subscribe(Observer<pair<vector<string>, Node*>>* obs)
        { out_.subscribe(obs); }

    // AVL-Trees
    void insert(int key);
    void remove(int key);

   private:
    Node* root_;
    vector<string> cmd_;
    Observable<pair<vector<string>, Node*>> out_ =
        [this]() { return pair(cmd_, root_); };
    sf::Text command_;
    sf::Text typeCommand_;

    void clear(Node* root);
    int balanceFactor(Node* cur) const;
    Node* returnRoot() const { return root_; }

    Node* rightRotate(Node* y);
    Node* leftRotate(Node* x);

    Node* balanceIfNeeded(Node* node);

    Node* insert(Node* node, int key);
    Node* remove(Node* node, int key);

    void notify();
};
```

A number of helper methods had been implemented to control Nodes in a specific AVL-Tree way: rightRotate() and leftRotate(), which rotate Nodes and their respective branches, balanceFactor(), which returns the differences in heights between the left and the right branches of a given Node, balanceIfNeeded(), whose purpose is to decide whether we need to rebalance a subtree or not. Here we may also see the implementation of the Observer pattern: whenever a tree is being created, it asks for a pointer to an observer, which will listen to the tree and react whenever the tree is updated.

2.2.2 The View

This part of the project is implemented using the open-source multimedia library named SFML. View is used for creating a window, calculating positions and sizes of individual Nodes and texts inside of them, changing Command Line text and rendering it all when needed:

```
class Viewer
{
   public:
    sf::RenderWindow* window_;

    Observer<pair<vector<string>, Node*>>* port() { return &in_; }

    void setupTheWindow();
    void updateFrame(Node* root);
    void handleResize(int width, int height, Node* root);
    void setText(string command);

   private:
    int x_, y_;
    sf::Font font_;
    sf::Text text_, typeCommand_;

    // frame buffers
    void drawBuffers();
    void clearBuffers();
    std::vector<sf::CircleShape> nodeBuffer_;
    std::vector<sf::VertexArray> linesBuffer_;
    std::vector<sf::Text> textBuffer_;
    std::vector<sf::Text> interfaceBuffer_;

    // rendering functions
    sf::CircleShape createCircle(int radius, int xNew, int yNew);
    sf::VertexArray createLinks(int x, int y, int xNew, int yNew, int position);
    sf::Text createKey(Node* root, int xNew, int yNew, int radius);
    void nodeDrawer(Node* root, int x, int y, int radius, int spacer, int level,
                    int position, int widthLevels);

    void getPosition(int& xNew, int& yNew, int x, int y, int radius, int spacer,
                     int widthLevels, int level, int position);

    void resizeNodes(int& spacer, int& radius, int widthLevels);

    // Observer
    void onNotify(std::pair<vector<string>, Node*> v);
    Observer<std::pair<vector<string>, Node*>> in_ =                        //
        Observer<std::pair<vector<string>, Node*>>(                         //
            [this](std::pair<vector<string>, Node*> p) { ; },               //
            [this](std::pair<vector<string>, Node*> p) { onNotify(p); },    //
            [this](std::pair<vector<string>, Node*> p) { ; });              //
};
```

Here we may see the observer being created: it handles all the on-screen changes as the AVL-Tree sends a signal informing about a change in the Model. The Viewer doesn't do anything as it gets attached and detached from a Model, but executes a proper function as the Model changes.

2.2.3 The Controller

Here's the Controller.hpp code. Controller's duty is to be an intermediary between the Model and the Viewer. It is used when text is entered, a command is sent, the window is resized or closed, etc.

```
class Controller
{
   public:
    Controller(AVLTree* tree) : command_(""), tree_(tree){};

    void handleEvent(sf::Event event);

   private:
    std::string command_;
```

```
    std::vector<std::string> msg_;
    AVLTree* tree_;

    // Utility
    void setMsgCommand(std::string s);

    // Signals
    void handleClose();
    void processCommand();
    void handleResize(sf::Event event);
    void handleKeyPress(sf::Event event);
    void handleTextEntered(sf::Event event);
    void notify();
};
```

2.3 Observer Pattern

Here's the trickiest part of this project. Observer is a behavioural application design pattern that lets the observable object (subject) have a list of subscript objects, observers (objects), that will be notified any time the subject changes its state. Let's first look at how the Observer class is implemented:

```
// prototype class
template <typename T>
class Observable;

template <typename T>
class Observer
{
    friend Observable<T>;

   public:
    template <class Tt1, class Tt2, class Tt3>
    Observer(Tt1&& onSubscribe, Tt2&& onNotify, Tt3&& onUnsubscribe)
        : onSubscribe_(std::forward<Tt1>(onSubscribe)),     //
          onNotify_(std::forward<Tt2>(onNotify)),           //
          onUnsubscribe_(std::forward<Tt3>(onUnsubscribe)) {};

    Observer(const Observer&) = delete;
    Observer& operator=(const Observer&) = delete;
    Observer(Observer&&) noexcept = delete;
    Observer& operator=(Observer&&) noexcept = delete;

    ~Observer() { unsubscribe(); }

    void unsubscribe()
    {
        if(!isSubscribed())
            return;
        Observable_->detach_(this);
        Observable_ = nullptr;
    }

    bool isSubscribed() const { return Observable_; }

   private:
    Observable<T>* Observable_ = nullptr;
    void setObservable_(Observable<T>* observable) { Observable_ = observable; }
    std::function<void(T)> onSubscribe_;
    std::function<void(T)> onNotify_;
    std::function<void(T)> onUnsubscribe_;
};
```

An observer is initiated with three functions, which are executed at subscription, notification or "unsubscription" respectively. As its only variable any observer has a pointer to an observable which it tracks. Additionally it has three methods: unsubscribe() is used to detach the observer from its observable, isSubscribed() tells if given observer is currently tracking any subject and setObservable_() makes an observer follow a subject by assigning an observable to it.

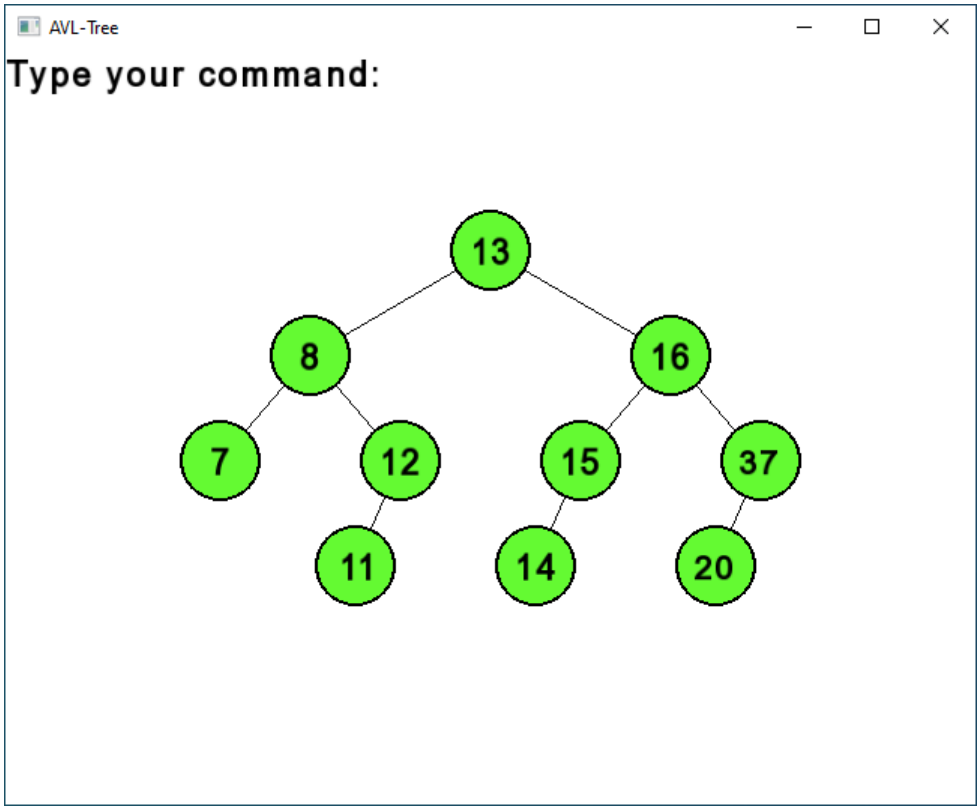
Now, let's take a look at the Observable class:

```cpp
template <typename T>
class Observable
{
    friend Observer<T>;

   public:
    // move constructor
    template <class Tt>
    Observable(Tt&& data) : data_(std::forward<Tt>(data)){};

    Observable(const Observable&) = delete;
    Observable& operator=(const Observable&) = delete;
    Observable(Observable&&) noexcept = delete;
    Observable& operator=(Observable&&) noexcept = delete;

    ~Observable()
    {
        while(!Observers_.empty())
        {
            Observers_[Observers_.size() - 1]->unsubscribe();
            Observers_.pop_back();
        }
    }

    void subscribe(Observer<T>* obs)
    {
        if(obs->isSubscribed())
            obs->unsubscribe();
        Observers_.push_back(obs);
        obs->setObservable_(this);
        obs->onSubscribe_(data_());
    }

    void notify() const
    {
        for(auto obs : Observers_)  //
            obs->onNotify_(data_());
    }

   private:
    std::function<T()> data_;
    std::vector<Observer<T>*> Observers_;
    void detach_(Observer<T>* obs)
    {
        obs->onUnsubscribe_(data_());
        auto it = find(Observers_.begin(), Observers_.end(), obs);
        Observers_.erase(it);
    }
};
```

An observable object is initiated using a custom data type, this part is exactly the beauty behind this Observer pattern implementation - it is so flexible and doesn't require anything being hard-coded into it, it will work with any C++ data structure without the need of fine-tuning the source code even a bit. Any observable also has a vector of observers which are notified when needed. The observable may subscribe an observer via subscribe(), notify observers about a change of state with notify() and unsubscribe any observer from itself.

2.4 Usage

This app provides an easy-to-use CLI: to add a Node to the tree type "ADD #", where # is a certain number and press Enter, to remove an existing node from the tree type "DEL #", where # is a certain number and press Enter. On the picture below you may see an example of an AVL-Tree I've built using this application.


AVL-Tree
Type your command:
13
8
16
7
12
15
37
11
14
20