APMS

A system for managing all your national agricultural needs.

Developed by:

Lazib Redhouane, S1-G1

Yahiaoui Mastene, S1-G2

Oukil Hamza, S1-G5

Besseghir Younis, S2-G7

Preface

-- Lazib Redhoaune

One can argue that most of the issues concerning the storing and usage of data around our world fall into the same root issues and solutions. Although the details might differ depending on the nature of the problem, the headlines tend to be quite similar in nature.  
Developing an agricultural management system certainly falls under these issues. We can see that building such a system requires the same steps as most others:

* Finding the optimal storage medium for the received data
* Developing a fast, efficient methods of retrieving, manipulating, and deleting the data
* Providing ways to better understand the data for future consumption.

Now these issues are quite simple on paper but figuring them out is not as simple. Depending on the context one is working with, Multiple answers with varying qualities may arise, and selecting the optimal one is unlikely considering that the entire project depend on it. There is also the factor of working as groups, as different people have different experiences, skills, and views. Thus, the results are always prone to improvement, tinkering, and development.

What we are trying to say is that although we realize that our solution to this problem is far from perfect, we claim that it serves its purpose as good as our ability can handle. It was a new experience for all of us and we can’t underestimate how much we have learned from it.

This report is but a small glimpse into the inner workings of our project, delving into our decision making, features, and even some of its limitations that we can see the possibility of improvement in.

Each section will be started by its name and author, so the distribution of our work will be visible enough. We have also structured our code well enough (hopefully) to be easy to understand, although some parts could be improved upon as well.

We hope that our work will serve as a bright model for our peers, and that its shortcomings will be a lesson for them as well.

Section 1: The problem at hand

-- Lazib Redhouane

Our objective is to build a management system to store, organize, and consume data about the farms in the entire country. It should be able to:

* Print any needed information about the lands, whether on their own or within their domain (Wilaya, City, Area, or the country itself). The needed data could be:
  + Information (ID, Name, Workers, Production, etc...)
  + Total sales (in a month, year, over a period, or even since its foundation)
  + Penalties applied to any land, at any month, for any product category, and total penalties for a certain domain.
* List the winners of the monthly competition for any category, at any month.
* Generate graphs for the sales of any land or domain, whether in a year or since their formation.
* Build the system so it can use either a BST-based database or an AVLT based database, utilize that to make a comparison between the two.

We believe that we have accomplished all these objectives in a way providing the most out of them, while giving the least headache to the user using the terminal, or the developer using the APMS class methods.

As for the time complexities of the operations, we’ve done our best to optimize the search and retrieval algorithms, and this is what we so far believe is them:

* Retrieval of any kind of information from the database: O(LogN), where N is the number of the instances of the targeted object (For example, `print\_cities\_total\_sales()` will make the N equals the number of cities in the database)
* Generating graphs:
  + Over all years: O(Y), where Y is the oldest recorded year in the database.
  + Over months of a year: O(1) (Technically O(12), but they are equivalent)

Delving deeper into the code might show a few instances where a better data structure would result in a slightly shorter time, but on the grand scheme of things, no operation should surpass these complexities in terms of time.

We will discuss more the used data structures further in the report, but beside BSTs and AVLs, we’ve used multiple data structures like vectors, hashes (in the form of unordered\_map hashes, more on that later), and a few Enums as well (if they’re considered data structures that is).

We’ve listed what every teammate has worked on by the end of the report, and each teammate has written about the work he did in this report (and edited by me, explaining the similarity in writing style).

Section 2: The data

-- Besseghir Younes

We’ve developed multiple classes that each serve as their real counterpart of the data, which then would be loaded into system and then manipulated with ease.

Instead of detailing every class, this diagram shows an overview of all the classes and the data they hold.

A diagram of a computer

Description automatically generated

* Remark 1: the Tree structure is either a BST or an AVL, depending on the used structure by the user/developer.
* Remark 2: The `Farmer` inherits from the `Worker` class.

Generally speaking, the APMS controls the user input and any interactions between the user/developer must go through it. It receives its data from the DBMS which loads it into an instance of it and parses it.

For the storing of data, there were many options to choose between. We’ve opted to use the JSON data format, which has its upsides and downsides of course, but the alternatives were not as appealing. We chose JSON because:

* Universal: We developed the system on the assumption that it’ll receive data from outside sources, requiring the developers to understand how the data is stored and structured. Using for example the CSV format would consume a lot of the developers’ time attempting to understand it, but with JSON, only a single glimpse would be sufficient.
* Fast: Even in big quantities, JSON’s performance is superb. Maybe not as fast as an SQL database for example, but it’s way less complicated in contrast, making it a great choice.
* Modular: JSON can be easily manipulated and edited, which allows us to test the system and push it to its limits in a fairly simple manner. Generating dummy data files became just a click away, allowing us to identify all kinds of issues with the system and fix them.

Now one downside is that it does take a fair bit of space especially when using inefficient data structures, but we’ve yet to confirm or deny that, for we don’t have an exact comparison between it and CSV for example when it comes to the data we want to store. Even still, its advantages are still worth it.

Now for reading it, we utilized a third-party library called nlohmann/json. Admittedly, multiple peers of ours have used it for their projects, but the reasons are apparent: Creating your own JSON parser would be a nightmare as all things can go wrong with it, so a safe method is appreciated. Unfortunately, though, it is not the fastest library in the world, and although we did our best to optimize things around, the work the library does is outside of our control.

After the data has been read by the DBMS and parsed, the APMS initializes all the classes and store them accordingly. The `Tree` container can either be a Binary Search Tree or an Avl Tree, and a comparison between the two will come later. Other than that, we’ve utilized multiple data structures such as vectors (and normal arrays), unordered maps, and multiple others. Utilizing hash was on our mind at the beginning, event implementing it in multiple places. But after some benchmarking, we found out that the std’s `unordered\_map` hash was slightly faster than our own implementation even after our tweaks with our design, so we decided to scrap our own implementation and go with the unordered\_map instead.

The heavy use of pointers can be noticed as well. This is because we wanted to allow the user to also change the data and save the changes. Although the foundation was laid (although it doesn’t have much of a benefit, the user can save the file from the interface anyways), with the saving functionality and data manipulation methods already accessible for developers, but due to some issues arising and the time constraints, we moved on to other things.  
There’s also that it removes the need of creating multiple instances of each class while keeping the benefit of faster access to them, which saves in space complexity and removes data redundancy, after all, multiple trees of pointers (8 bits each – for a 64x device, that is) is much better than two trees of the same class (+32 bits for each instance).

As for the data itself, a mix of real and dummy data was utilized. A javascript/typescript script was developed (by Redhouane) to take stored real data collected by us (Younis, Mastene) and fill it with dummy data about all kinds of lands, farmers, and products. Of course, they restrictions were put so the data could look as realistic as possible. Instructions for using it are at the end of the report.

Section 3: The usage

-- Lazib Redhouane

### For the developer

Using the system is very simple. A demo is included with the code folder, but for fast reference, you can see it in Fig.1.

A screenshot of a computer program

Description automatically generated A screen shot of a computer program

Description automatically generated

Fig.1: The default driver.cpp file Fig.2: All categories are dynamically implemented.

As one may guess, the BST version of the system is `APMSBST`, and the Avl is `APMSAVL`. We had the idea of utilizing polymorphism to keep both implementations in the same class, and had quite the progress at that, but it never worked the way we wished in the end; thus we went with what most have done to solve this issue.

An interesting fact is that unlike most other projects, Product categories are dynamic. You can see that by going through the `default\_config.hpp` file (Fig.2). We wanted to keep the project as modular as possible even if not required to. I do believe that anything that could be dynamic has to be dynamic, as I’ve yet to see a scenario where hardcoding things went well, and we tried to reflect that to our extent in our work.

### For the user

Using the system is even simpler for the user, for every choice he can make is shown to him at every step, so he doesn’t need to know anything beforehand.

A screenshot of a computer program

Description automatically generated

One thing to notice is that the terminal is colorful. This is thanks to a very simple [third-party library](https://github.com/aafulei/color-console) that allows for easy addition of colors at a negligible cost. (We found it broken due to some incompatibility with modern C++ versions, but we fixed it).

### Compiling

We have provided a ready to use driver.exe file to test the system with, but if you wish to compile it yourself, bear in mind that we have used some special configurations for our compiler script so it can compile the code correctly. They are provided at the end of this report.   
Just to point this out, the project takes a fair bit to compile. It compiles differently on our devices depending on their specs, so expect the compilation time to take a while. This is mostly due to the use of the JSON library as it has a lot of complicated content, we have no control over, but our work when standalone is fast enough.

Section 4: The choice of design

-- Oukil Hamza, Yahiaoui Mastene

It took us quite a while to decide on how our data must be structured. Storing years’ worth of data was no simple feat, especially considering that we decided from the beginning that it’ll all be stored in a single JSON file. Although most of our design choices align with our peers and with expectations, some others are quite interesting, and we’ll explain them before:

* All data will be self-encapsulated: Let’s take a City for example. It should be able to completely function on its own as it has all the data it needs like the farmers, workers, production, etc.…  
  It should not import any kind of other data from any other place. All what it needs is within it.  
  This is another advantage of JSON, as if we used CSVs for example, it would’ve been almost impossible to store the data without separating it into multiple folders. We wanted to give developers as least of work as needed. Only one data.json file is needed.
* Production is stored on multiple years: all lands have their accumulated production stored within them. You don’t need a separate file or program to retrieve the production of each land as well.
* Lands are the base; Farmers are the result: It is noticeable that we gave lands the bigger role instead of farmers. All the data about production over time has been stored and is processed through the lands, but farmers are always the result as required. We found this structure to be more logical than farmers having their production for example.
* Workers are stored over the years: An interesting choice we did was making the workers stored in the AnnualReport class instead of the Land or Farmer ones. This was intentional, as a worker may leave a Land by the next year, or another join it in the year after. It is also to store their ages dynamically, so at each year, you can see the same (and sometimes, different) workers having their ages develop, enhancing the realism of the system.
* If you can method it, method it: We designed the classes so every possible scenario requiring them is solved with ease. If you want to print the yearly report of some area, there is a method for it. If you want to print the monthly production of some land, there is a method for it. If you want to change the data, there are methods for that as well. This may have had the downside of ballooning our project side, but we believe that it was a worthy venture.

By no means do we claim our choices as optimal, but we did our best to make them serve the project well.

Section 4: Analysis of Performance

-- Yahiaoui Mastene

We’ve written a script that tests out loading, searching, and destructuring of the APMS depending on whether it uses a BST implementation or an AVL one. The results generally have aligned with what expectations we had, with the AVL implementation taking longer to be built and destructured, but is faster when it comes to looking up data to use it.

For building, we used multiple iterations of building and loading the data into the system, then destructuring the instance. As for executing, we tested out some multiple extreme cases, such as looking up for the last land in the tree. Although the data sample we were using did not include the entire country, it included enough data (+2 million lines of JSON!) to form a consensus about the conclusion.

A graph with a line and a line graph

Description automatically generated with medium confidence

Fig.3: It is noticeable how the AVL takes longer to load and created. This is expected.

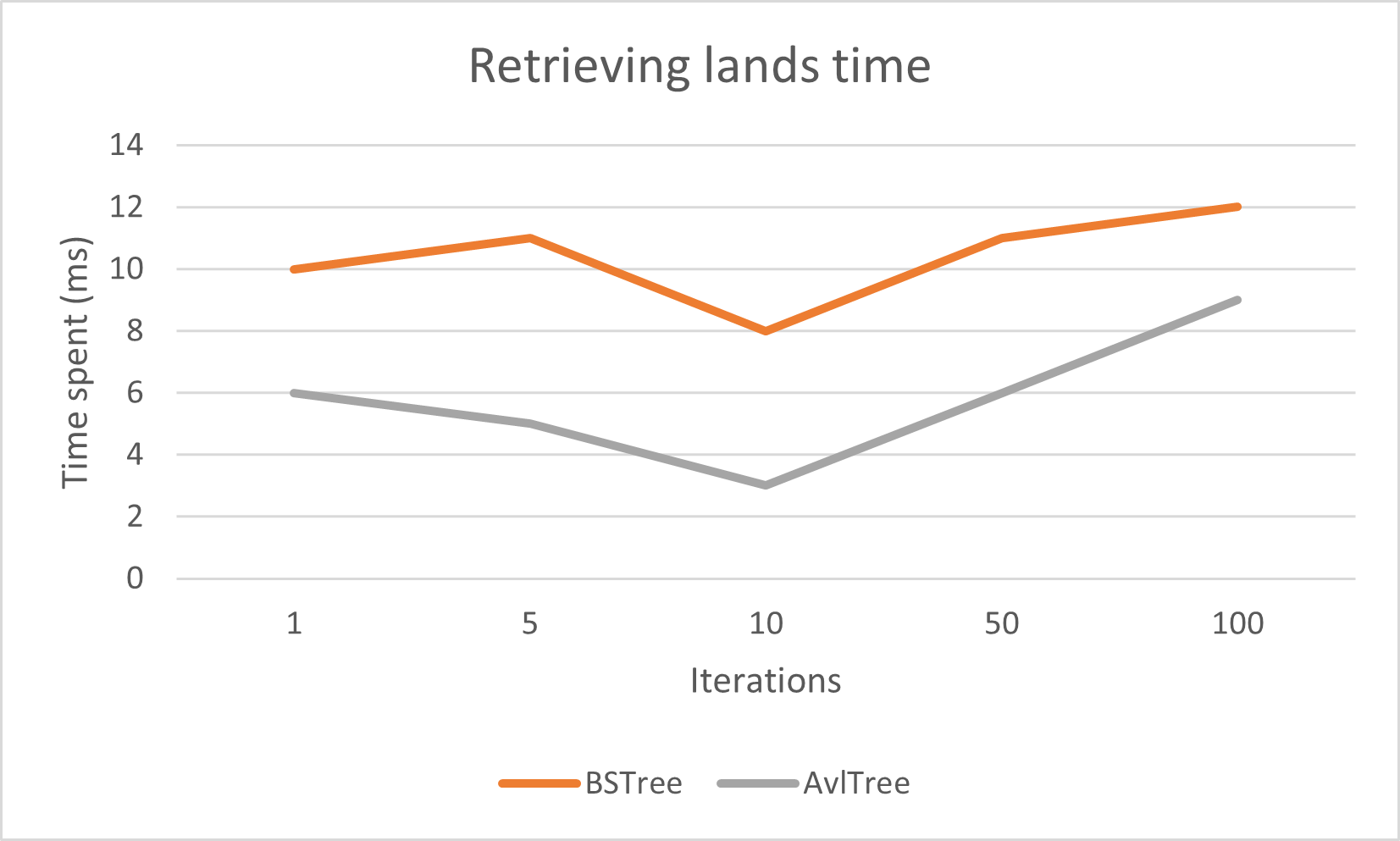


Fig.4: As expected, instruction processing time is faster for the AVL implementation.

Conclusion and Acknowledgements

Apart from the appendix, we conclude our report by saying that we are satisfied with the result of our work, hoping that it satisfies you as well. We thank everyone who supported us on this journey and our mentors for their help. It was challenging to figure out all the intricates of the design, and we’ve learned so much about the material we’ve studied this semester thanks to our application of it.

If we did well, its only thanks to allah and then to our work. Any mishaps are our own.

Appendix

## Section I - Instructions on building the system

If you are using g++, make sure that you use these parameters:

`g++ PATH/TO/driver.cpp -I(PATH/TO/lib) -g \*.cpp classes/\*.cpp ../lib/plot/\*.cpp -o driver.exe`

If you are using vscode, make sure that add/replace your compiling task at tasks.json with this:

{

      "type": "cppbuild",

      "label": "C/C++: g++.exe build active file",

      "command": "C:\\msys64\\mingw64\\bin\\g++.exe",

// Or wherever g++ is located at

      "args": [

        "-fdiagnostics-color=always",

        "-I(PATH\\TO\\lib)",

        "-g",

        "\*.cpp",

        "classes/\*.cpp",

        "../lib/plot/\*.cpp",

        "-o",

        "${fileDirname}\\${fileBasenameNoExtension}.exe"

      ],

      "options": {

        "cwd": "${fileDirname}"

      },

      "problemMatcher": [

        "$gcc"

      ],

      "group": {

        "kind": "build",

        "isDefault": true

      },

      "detail": "Task generated by Debugger."

}

Remark: Make sure to replace (PATH\\TO\\lib)", including the ()Section II - Everyone’s work

* Redhouane Lazib: Planning, writing/editing the report, building the data generator, building the user interface, developing the DBMS system.
* Yahiaoui Mastene: Creating the lower classes (Farmers, Production, Categories, etc..) and developing their methods (with coordination with Oukil Hamza), developing test programs and providing charts of their results.
* Hamza Oukil: Creating the higher classes (Wilayas, Cities, Areas, etc…) and developing their methods (with coordination with Yahiaoui Mastene), prettifying the user interface and making data retrieval clearer and better.
* Besseghir Younis: Data collection, writing the report, fixing appearing bugs, implementing the plotting functionality.

## Section III – Generating the data.

If you wish not to use the current dataset and instead use a new/custom one, you can generate it using NodeJS.

Make sure that you have [NodeJS](https://nodejs.org/en) installed on your device.

Simple navigate to `code/dataset` and run `node generator.js START\_YEAR END\_YEAR WILAYA\_COUNT`. Replace the uppercased parameters accordingly.

* Remark: any WILAYA\_COUNT higher than 58 will just make 58 because the available real data concerns only Algeria.

Data will be generated in the same directory. Move it to the APMS or change the path in `driver.cpp` and compile.

## Section IV – References

Those are all the libraries we’ve used to develop our system:

* [Nlohmann/JSON](https://github.com/nlohmann/json)
* [Console-Color](https://github.com/aafulei/color-console)
* [PbPlot](https://github.com/InductiveComputerScience/pbPlots)

Besides that, some very simple snippets and ideas (like the way to develop the average time test program) were thanks to the work of the people maintaining the [StackOverflow](https://stackoverflow.com/) site. Only the ideas were taken from there and absolutely no code was copied. Any AI tool was prohibited since the beginning of development.