Case 9 : A Tree-cky Case...

Submission instructions

At the end of the practical, your Git repository must follow this architecture:

```
csharp-tpXX-firstname.lastname/
|-- README
|-- .gitignore
|-- Bonus/
    |-- GenTree.ml
    |-- Makefile (if any)
|-- Warmup/
    |-- Makefile (if any)
    |-- Warmup.cs
    |-- Program.cs
|-- Whodunit/
    |-- data/
        |-- Everything
    |-- GenTree.cs
    |-- Makefile (if any)
    |-- Program.cs
    |-- Suspects.cs
```

Do not forget to check the following requirements before submitting your work:

- You shall obviously replace `firstname.lastname` with your login.
- The `README` file is mandatory.
- There must be no solution `.sln` in the repository.
- There must be no executable `.exe` in the repository.
- There must be no `bin` or `obj` in folder in the repository.
- You must respect the prototypes of the given and asked functions.
- Remove all personal tests from your code.
- **The code MUST compile !**

README

In this file, you can write any and all comments you might have about the practical, your work, or more generally about your strengths and weaknesses. You must list and explain all the bonuses you have implemented. An empty `README` file will be considered as an invalid archive (malus).

1 Introduction

1.1 Objectives

Throughout the semester you have been using JetBrains Rider, which is what is called an IDE. The term IDE stands for 'Intergrated Development Environment' and is a powerful tool designed to greatly increase your workflow by combining essential programming tools and presenting them with a complete GUI (Graphical User Interface), but you will not be needing that this week.

Although using an IDE is seen as preferable most of the time, you might not always be able to get your hands on one or you will be working on an environment that simply does not have one installed (like next year for example). The goal of this practical is to take you out of your comfort zone and have you discover just how powerful some simple text editors can be with the right mindset (and configuration). You can see this practical as a little teaser of how you will have to work next year. We strongly advise you to play ball and not use an IDE or thunar.

Throughout this practical you will see a bit about compilation and how to write files the old fashioned way with the classic editors **Vim** and **Emacs**.

Be Careful !

We strongly advise you to work on the PIE for this practical since it is expected to compile and run on the school's environment.
Any code that fails to compile will result in a big penalty so it is crucial you compile and test your code !

One last thing...

In order to make sure you do not use Rider, the compiler we will be using will use an older version of C#. If you do not listen to us and use Rider, you might use some features that are not available and thus, your code will not compile.

2 Course

To understand the key notions of this practical, it is absolutely essential that you read the entirety of the course before starting.

2.1 Tools for this practical

The point here is to present to you the tools we want you to use for the practical. You hopefully already know most of them, but some reminders have never hurt anyone.

2.1.1 The shell

A *shell* is a user interface, usually a command-line interface (or CLI) which allows a user to interact with operating system services. In simpler terms, a shell is the big black box you type commands into that magically executes them. Here is an example of a shell use.

```
42sh ~ echo

42sh ~ echo bpl future moulette
bpl future moulette
42sh ~ ls
TPdir evalexpr.c ConfDeJulien.pptx facts.out
42sh ~ cd TPdir
42sh ~ ls
DeuxiemeMeilleurTP TP1 TP2 TP3 TP4 TP5 TP6 TP7 TP8 MeilleurTP
42sh ~ ls ..
TPdir evalexpr.c ConfDeJulien.pptx facts.out
42sh ~ ls -a ..
.evalexprtraces TPdir evalexpr.c ConfDeJulien.pptx facts.out
42sh ~ cd ..
42sh ~ ./facts.out
Plus de 90% des malwares utilisent le registre rax
```

The `42sh~` text is what we call a **prompt**. It is displayed whenever the user can type a command in the terminal.

You'll notice that each command (the text the user typed after the prompt) is constructed the same way: `program argument1 argument2 ...`

`program` is actually the path to the program you want to execute. It can be a built-in program (meaning that there is no actual executable file attached to it and that it is part of the shell program) for commands like `cd`; executable files found at a default path like `ls` or `tree`; or executable files at the specified path, like the `facts` program in our example. In that case, you need to type `./facts.out` to execute it when you are in the same directory.

Just some more reminders:

- The uppermost file in your *filesystem* is called the root. On Unix, its path is `/`
- Each file has a path from the root of the filesystem, called the absolute path. This path always starts with a '/'

- Each file also has a path relative to the position of all other files. This is called the relative path. This kind of path that does not start with '/'
- The parent directory of a file is at the relative path `..` (like in our previous example). On the other hand, `.` refers to the current directory.
- Files whose name starts with a dot are hidden files. You need to type specific commands to see them.

Reminder

Due to the way C# is compiled with csc, keep in mind that ./Program.exe will result in an error on the PIE. The correct way is to run with `mono Program.exe`.

2.1.2 Basic commands

These are the most basic commands of the shell that you should know and be able to use:

- `ls` - list files
- `pwd` - print working directory
- `cd` - change directory
- `mv` - move
- `cp` - copy
- `unzip` - unzip
- `cat` - display content of a file
- `touch` - update a file
- `mkdir` - make directory
- `rm` - remove
- `tar` - compress or decompress files

Tip

For any shell command, if you have any doubt about what a command does, you might want to use `man`, `apropos` or `whatis` followed by the command for more information.

2.2 Text editors you have to know

Since you will not be using Rider for this practical, here are a couple options for text editors that you can use.

The first, which you already know pretty well, is **Emacs**. It is a powerful tool that is extremely customizable. For those of you that were traumatized by OCaml, do note that **Emacs** is just a text/code editor, not an OCaml editor. You should be able to use **Emacs** pretty well by now.

The second option, which is more widespread at EPITA, is **Vim**. **Vim** is a program that runs directly in your terminal. When used well, it can make you work a lot more efficiently, though it is a little tough at first. The main point of **Vim** is that it is a *modal* text editor: it has many modes for editing text (insert, replace, select, and so on) that all have their own purpose. To learn the basics of **Vim**, we recommend using `vimtutor`[1] as it will explain everything you need to know. To use vim, simply type in your terminal `vim` followed by the name of the file you wish to edit.

Tip

If you have any trouble using Emacs or especially Vim, which is not exactly beginner friendly, do not hesitate to ask your ACDCs for help.

2.2.1 Vim

You will find on the Moodle page a file names **vimrc**. This is a **Vim** configuration file that provides a basic yet decent configuration to ease your work. You are free to change it or keep it as it is, it is entirely up to you. To install it, download it on the Moodle page and move it in **/afs/.confs/**. After that, launch the script **/afs/.confs/install.sh**, you **Vim** should now use the provided configuration.

2.2.2 Emacs

As for **Vim**, we offer a basic configuration for Emacs. As for Vim, it is downloadable on the TP's Moodle page and you are entirely free to change it at your will. To use it, just download copy-paste it in `/afs/.confs/emacs` and start Emacs.

[1]You only have to type this in a terminal and press enter to run it.

2.3 .NET, Mono and Compiling

You won't be using Rider for this practical, so this section will teach you everything you need to know to compile and run your code.

2.3.1 Compilation vs Interpretation

Once you have written your code, you will probably want to execute it. The problem is that your computer does not natively speak C#, Python, OCaml, or whatever language you are using. Your computer only speaks one language, *machine language*, which is a sequence of binary instructions that does not make sense for any human being. So how could one communicate with a computer to tell it what its instructions are? The answer is quite simple: you need a translator, something to translate your human readable code, the *source code*, into a *bytecode* which can be read and executed by your computer.

There are two main ways of doing so: **compilation** and **interpretation**. Compilation is the process of taking all of your code and translating it at once into the target machine language, which you will then be able to execute. On the other hand, interpretation is the process of executing each instruction of your code on the machine, without generating any executable file. Each strategy has its pros and cons.

- Compilation takes time, but once they are compiled, programs do not take much time to run. On the other hand, interpretation does not need time to generate a binary, but interpreting is not as fast.

- Compilation produces a binary which does not fit every machine, while interpreted code can be run on every machine, since it does not generate any binary. You could, however, compile your code on a different computer (on a different system) while still making it portable to the computer you have. This is called *cross-compiling*.

Those are just a few examples of the specificities of compilation and interpretation. Preferring one to the other is really just a matter of context; there is no inherently better method.

2.3.2 What is .NET?

.NET is what we call a software framework. A software framework is a kind of library, meaning a set of pre-implemented functions that a programmer can use as he or she pleases. The difference between a standard library and a framework is that a framework handles more than just pre-defined functions, as it can deal with program flow, compilers or other such things. Furthermore, a framework can be extended or overwritten by a user (in this case the programmer), though it cannot be modified.

.NET is a framework that was developed by Microsoft, and is often regarded as a solid and stable framework. It has a significant disadvantage: it was primarily developed for Microsoft Windows. .NET sets up a virtual machine which handles security, memory management and exceptions.

When you are building your project in Rider using the .NET framework, it is compiling your code into a library[2] which will then be interpreted using the .NET framework. The point of doing so is that it allows the framework to interpret code written in all the languages of the .NET framework, such as F# (the Microsoft version of OCaml) or Visual Basic.

Therefore, when you are developing with Rider, you are in some way compiling and interpreting your code. For this practical, we want you only to compile your code, and to that end we will be using **Microsoft's Visual C Sharp Compiler** (or `csc` if you prefer).

[2]A .dll library to be precise.

2.3.3 What is Mono?

As you have seen before, code can be either interpreted or compiled, with interpreters or compilers respectively. In simplified terms, Mono is a project that includes both a C# compiler and an interpreter (or Common Runtime Language). Both of these are compatible with .NET, so you will often see these two coming together as a pair. For this practical, however, you will only use the interpreter and you will compile with Microsoft's C# compiler.

As we have already seen, compiling code is seperate from executing it. Indeed, when we compile code, it will create a binary file that you can then execute. In order to compile our code, we will be using `csc`[3], which is the C# compiler. Using it is very easy: simply use the command `csc` with all the files you want to compile.

If you want to specify the name of the binary, you can use the `-out:program`, where you must replace `program` by the name of your output.

To sum up: if you have two files `Program.cs` and `Basics.cs` that you want to compile to `Basics.exe` you must type:

```
csc Program.cs Basics.cs -out:Basics.exe
```

You can then execute this binary using `mono Basics.exe`

Important

To compile, your code must contain one `Main` method. If you have none or more than one, your code will not compile.

To recap: in order to run C# programs, you need to compile the code into an intermediate language, the bytecode, that will be interpreted with what is called a JIT-compiler[4], which translates the bytecode into machine code at runtime.

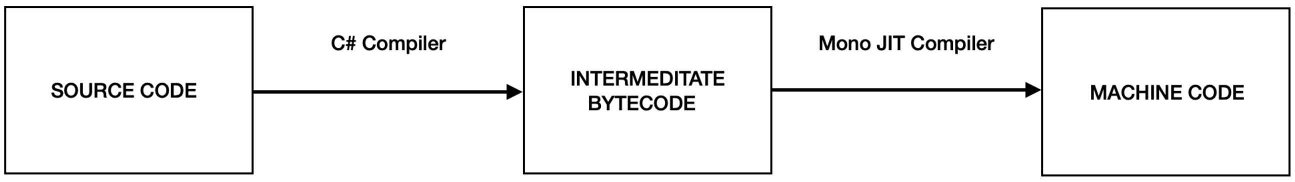


Figure 1: C#'s compilation pipeline

[3]Short for "C-Sharp Compiler".
[4]Just-In-Time Compiler

2.4 C# files

We will not provide you with any skeleton for some exercises in this practical and since you should not be using Rider, now is the perfect time to explain in detail about the structure of a *.cs* file.

```
using System;

namespace Basics
{
    public class Hello
    {
        public static void Main(string[] argv)
        {
            ConfDeJulien();
        }

        public static void ConfDeJulien()
        {
            Console.WriteLine(@"Et la tu reecris ton %eip pour faire une
            execution de code arbitraire!");
        }
    }
}
```

The example code just above is a typical *.cs* file. We can see that it is composed of three main types of blocks.

- The **using** block, which contains statements specifying which namespaces will be used. Any functions in these namespaces can be used in the namespace of the file. This block is optional.
- The **namespace** block which specifies the namespace of the file. A namespace is a way of organising various classes in groups. A class belonging to a different namespace is thus called by specifying the former.
- The **class** block which contains the definition of a class. You should all be familiar with them at this point. There can be multiple class definitions in the same file.

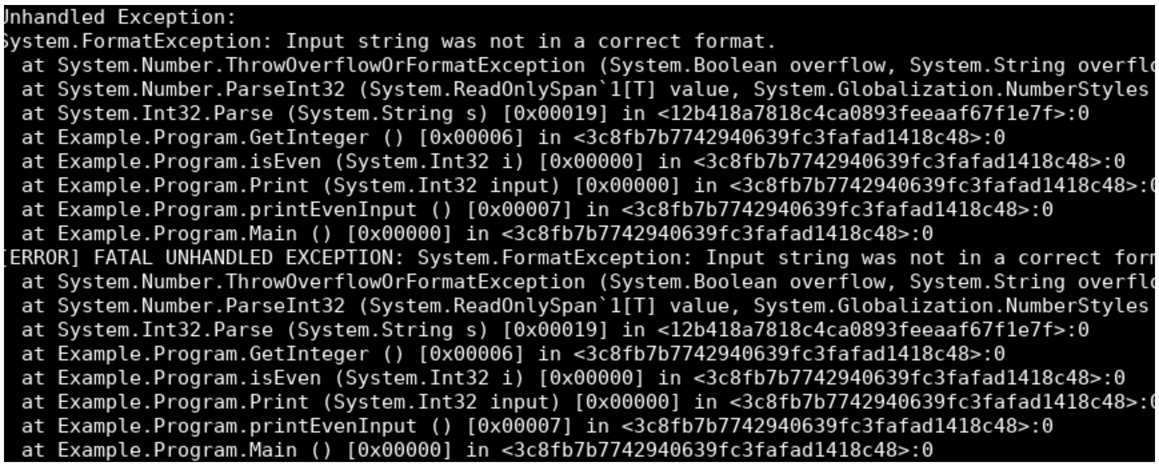
A *.cs* file needs to respect this structure to be valid and compile. An interesting point, which you might have already noticed, is that in C# all functions are part of a class. Actually, in C# everything is a class. Functions are just static methods, which means you do not need to instantiate any object to call them.

Important

Please note that the C# convention is to have only **one public class per file** and to name the class like the file.
Basics.cs will contain the Basics class, Complex.cs will contain the Complex class, and so on.

2.4.1 Runtime exception and compiler messages

In the last practical, you learnt about exceptions and how to handle it. But you never learnt about how to read an exception report, and how to deal with it. Usually Rider can handle some things for you, but not today, so it looks like a nice time to learn! First, let's check how it looks like:



```
Jnhandled Exception:
ystem.FormatException: Input string was not in a correct format.
  at System.Number.ThrowOverflowOrFormatException (System.Boolean overflow, System.String overfl
  at System.Number.ParseInt32 (System.ReadOnlySpan`1[T] value, System.Globalization.NumberStyles
  at System.Int32.Parse (System.String s) [0x00019] in <12b418a7818c4ca0893feeaaf67f1e7f>:0
  at Example.Program.GetInteger () [0x00006] in <3c8fb7b7742940639fc3fafad1418c48>:0
  at Example.Program.isEven (System.Int32 i) [0x00000] in <3c8fb7b7742940639fc3fafad1418c48>:0
  at Example.Program.Print (System.Int32 input) [0x00000] in <3c8fb7b7742940639fc3fafad1418c48>:
  at Example.Program.printEvenInput () [0x00007] in <3c8fb7b7742940639fc3fafad1418c48>:0
  at Example.Program.Main () [0x00000] in <3c8fb7b7742940639fc3fafad1418c48>:0
ERROR] FATAL UNHANDLED EXCEPTION: System.FormatException: Input string was not in a correct for
  at System.Number.ThrowOverflowOrFormatException (System.Boolean overflow, System.String overfl
  at System.Number.ParseInt32 (System.ReadOnlySpan`1[T] value, System.Globalization.NumberStyles
  at System.Int32.Parse (System.String s) [0x00019] in <12b418a7818c4ca0893feeaaf67f1e7f>:0
  at Example.Program.GetInteger () [0x00006] in <3c8fb7b7742940639fc3fafad1418c48>:0
  at Example.Program.isEven (System.Int32 i) [0x00000] in <3c8fb7b7742940639fc3fafad1418c48>:0
  at Example.Program.Print (System.Int32 input) [0x00000] in <3c8fb7b7742940639fc3fafad1418c48>:
  at Example.Program.printEvenInput () [0x00007] in <3c8fb7b7742940639fc3fafad1418c48>:0
  at Example.Program.Main () [0x00000] in <3c8fb7b7742940639fc3fafad1418c48>:0
```

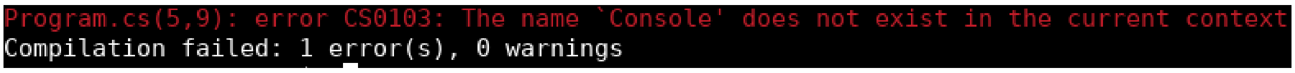
Figure 2: Example of stacktrace

What we call the *stacktrace* is the report of successive function calls that caused the crash. Pretty ugly isn't it ? But actually really easy to understand!
First thing to notice is the second line: `Input string was not in a correct format.` This is the error message, telling us why the exception **System.FormatException** got triggered in the first place.
The next few lines are some things that we don't really need to worry about since they are internal C# stuffs.
What we are interested in is the line`at Example.Program.GetInteger ():`
This is the first function that we see that belongs to one of the classes we've implemented ourselves, from this line, we know that the crash occured in the function **GetInteger** that takes no argument (because of the **()** after the function's name) and belongs to the class **Program** from the namespace **Example**. Now that we know this, we can get even more information looking at the line right above it, the presence of `System.Int32.Parse` means that the crash happened in a call to `Int32.Parse` from the **GetInteger** function.
The next lines are the functions that have been called leading to this error. In this example, the `Main` function called `printEvenInput`, which called `Print` which itself called `isEven`, which called `GetInteger` where the crash happened.

Let us now see about compiler errors. When trying to compile your code, you will most likely run into compilation errors (if not, congrats you're on your way to be a perfect programmer), meaning that the compiler cannot follow the pipeline described above because something in your code is wrong. Imagine we have this small file, that displays on the standard output a quote from the best BDE of Epita:

```csharp
public class Program
{
    static void Main()
    {
        Console.WriteLine("Votai Test.")
    }
}
```

When trying to compile it, **csc** will give us the following:



```
Program.cs(5,9): error CS0103: The name `Console' does not exist in the current context
Compilation failed: 1 error(s), 0 warnings
```

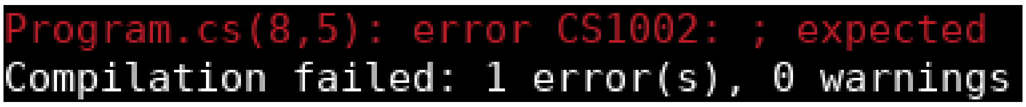
Figure 3: First compilation error, yay!

But what to deduce from that ? Once again, everything has been made so that it is easy to understand: In the file **Program.cs**, at the 9th column of the 5th line, an error occured. And this error is that we are using the name `Console` that has not been defined. This name is defined in **System**, a namespace provided in C# core language. So let's add that to our program. We now have:

```csharp
using System;

public class Program
{
    static void Main()
    {
        Console.WriteLine("Votai Test.")
    }
}
```

And this is the compiler's output:



```
Program.cs(8,5): error CS1002: ; expected
Compilation failed: 1 error(s), 0 warnings
```

Figure 4: And a second one ! :)

This one is a little bit less straight forward: It indicates an error on line 8, but that line is just a '}'. The actual error is that we forgot the ';' on the previous line. After correcting it, we can finally compile and run our program.

3 Warmup

This section is only here so you can get familiar with the editor and the shell before tackling the main part of the practical. It is also here to test your understanding of the course you have obviously read before starting this section. As you may have noticed, there is no Warmup folder in your skeleton and no this is not a mistake. You will have to create the `Warmup` folder with a `Warmup.cs` file from scratch.

In this file you are only allowed to use the System namespace[5] and you must define a custome namespace called `Warmup` and define these exercises in the `Warmup` class.

3.1 IsPalindrome

A palindrome is a word that can be read both from left to right and right to left. You must define a boolean method that will check if a given string is in fact a palindrome. The string can have any kind of character. For this exercise, we will consider a valid palindrome as only the alphabetical characters and it will not be case sensitive.

```csharp
public static bool IsPalindrome(string str);
```

```csharp
Warmup.IsPalindrome("Anna");                     // Returns true
Warmup.IsPalindrome("rAdAr");                    // Returns true
Warmup.IsPalindrome("rAdAar");                   // Returns false
Warmup.IsPalindrome("k,a;y;;;;;A;;k2930."); // Returns true
```

Caution

The input can be either empty (which will be considered an invalid argument) or null. Both of these cases must be handled and must throw an appropriate exception. This stands for the rest of the practical as well.

3.2 RotChar

By now you must be quite familiar with this exercise. The method must return the letter **c** shifted **key** times. The key can be negative. You only have to handle uppercase letters.

```csharp
public static char RotChar(char c, int key);
```

```csharp
Warmup.RotChar('A', 2);   // Returns 'C'
Warmup.RotChar('X', 5);   // Returns 'C'
Warmup.RotChar('C', -30); // Returns 'Y'
```

3.3 RotString

Encodes the string **str** by shifting each letter **key** number of times.

```csharp
Warmup.RotString("IRENE ADLER", -18);    // Returns QZMVM ILTMZ
Warmup.RotString("CISCO > MIDLAB", 42); // Returns SYISE > CYTBQR
```

[5] To learn more about namespaces:
`https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/types/namespaces`

3.4 Binary Search

```
public static int BinarySearch(int[] array, int elt) ;
```

You must implement the binary search, which searches for the position of an element in a sorted array with an average complexity of $O(\log n)$. If the element is not present in the array, you must return where it should be. If the array is null, you must raise an appropriate exception.

```
int[] array = {0,1,3,5,7,8,12};
BinarySearch(array, 0);     // Returns 0
BinarySearch(array, 1);     // Returns 1
BinarySearch(array, 5);     // Returns 3
BinarySearch(array, 8);     // Returns 5
BinarySearch(array, 42);    // Returns 7
```

3.5 Makefile (Bonus)

3.5.1 Overview

As you may have noticed, typing the same commands every time you want to test something or compile your project quickly becomes very tedious. Fortunately, there is a command that can streamline this process: **make**.

make is a program that allows the user to build projects efficiently by defining a set of rules to run in what is called a `Makefile`. The user can set up different targets that can run different sets of rules depending on their needs. The rules are composed of the commands the user would normally run.

```
42sh ~ ls
Makefile Program.cs
42sh ~ make Program
csc Program.cs
Microsoft (R) Visual C# Compiler version 3.6.0-4.20224.5 (ec77c100)
Copyright (C) Microsoft Corporation. All rights reserved.

42sh ~ ls
Makefile Program.cs Program.exe
42sh ~ make run
csc Program.cs
Microsoft (R) Visual C# Compiler version 3.6.0-4.20224.5 (ec77c100)
Copyright (C) Microsoft Corporation. All rights reserved.

mono Program.exe
Hello World !
```

In order to understand the syntax of a Makefile, let us look at a simple example:

```
variable = OCaml over all

echo:
	echo variable

echovariable:
	echo $(variable)
```

Here, you can see there is a variable containing some value and two targets called `echo` and `echovariable`. The targets both have one rule inside so when calling make, if no target is specified, the first one in the file will be executed. The first will only print the word variable because to expand the variable you need to put a dollar sign in front and between parenthesis (or curly brackets).

```
42sh ~ make
echo variable
variable
42sh ~ make echo
echo variable
variable
42sh ~ make echovariable
echo $(variable)
OCaml over all
```

Since rules are basically shell commands, you are free to put whatever you want in order to compile your project.

3.5.2 Objective

Write a Makefile[6] containing the rules `compile`, `run` and `clean` that will respectively:

- Compile the project
- Compile **and** run the project
- Remove all files that were created by the rules above

Watch out

When removing files with the **rm** command, there is no prompt so if you remove them **they are gone for good**. It is thus crucial for you to **commit** regularly your work to be able to get your work back in case this happens to you. (It really isn't fun when it happens.)

[6]You may want to take a look at this link:
`https://slashvar.github.io/2017/02/13/using-gnu-make.html`

4 Whodunit ?

There is a murderer in the family...

Sherlock is investigating a rather tricky case. A person has been murdered inside their family manor and no one knows what happened. He is sure of one thing though: someone in the family is lying. Sherlock has access to the victim's family tree and he needs your help to be able to sort out and streamline his job by implementing an easy way to evaluate and search for suspects in the tree.

Luckily, our high-functioning sociopath has devised an implementation for this exact purpose but requires your help to flesh it out.

4.1 GenTree

The detective has structured his tree as follows:

- Each node represents a family member and has a name and a degree of suspicion attached to it.
- The left and right nodes linked to the current one represent the member's parents.
- All of the members have two parents.

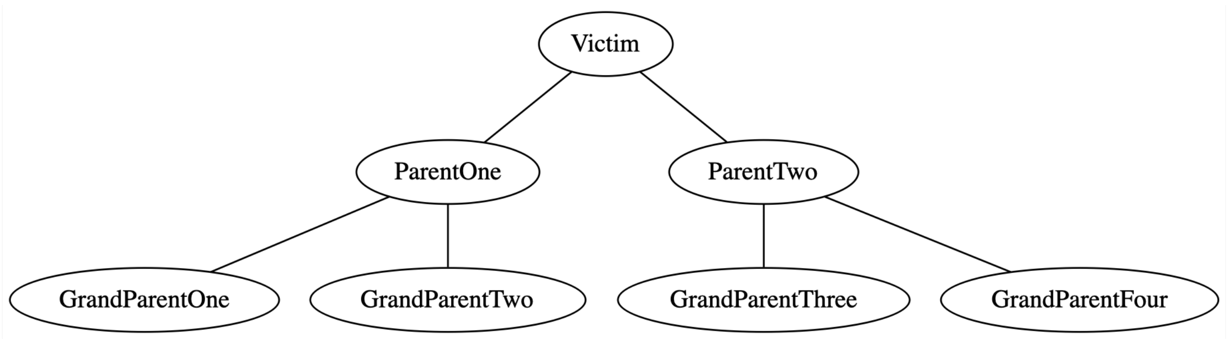


Figure 5: A simple ascendancy tree

Head's up

For simplicity's sake, we will consider that there are no family members that share the same name.

4.1.1 See what you did there

First thing to do is obviously to do a constructor for our GenTree class:

```
public GenTree(string name, int suspicion, GenTree left, GenTree right)
```

Now, Sherlock wants to have a basic way to look at the tree. He thinks that something along the lines of

```
Root (LeftChild RightChild)
```

would look pretty neat and wants your help with that since he is still busy thinking and interviewing the suspects.

```
public void PrintTree()
```

For example, using the Simpson's family tree with Bart Simpson as a root, we would get:

```
Bart (Homer (Abraham Mona) Marge (Clancy Jacqueline))
```

4.1.2 Change Name

To start off the case, Sherlock interviewed each family member. He then proceeded to write their names and put them in the tree. There is one issue though, he is not sure of the spelling as he did not have them write it down for him. He plans on asking to have it changed later on to the appropriate spelling. Meanwhile, he wants you to find a way to change the old name with the new name in the tree.

```
public static bool ChangeName(GenTree root, string oldname, string newname);
```

The method must return a boolean indicating whether the change occurred or not.

4.1.3 Find Path

The consulting detective now needs to know exactly how one person is related to another in the tree. He also needs you to keep track of the names encountered between both.

```
public static bool FindPath(GenTree root, string name, List<string> path);
```

You must not push the last name (the one you are looking for) in the list, only the nodes encountered during the search if, and only if, a path has been found. If no path exists, the list must be empty. The path starts from the last node up until the root.

4.1.4 Lowest Common Descendant

The last tool that is needed is one that allows Sherlock to be able to find the lowest common descendant of two ancestors. The lowest common descendant can be explained as the deepest node in the tree that has both nodes as ancestors in the tree.

```
public static string LowestCommonDescendant(GenTree root, string PersonA,
string PersonB);
```

For example on the figure above:

- The lowest common descendant between GrandParentOne and GrandParentTwo is ParentOne
- The lowest common descendant between GrandParentThree and ParentOne is the Victim
- The lowest common descendant between GrandParentFour and ParentTwo is the Victim

4.1.5 To Dot (Bonus)

When working with trees in general, it can become quite complicated to visualize the structure itself and what is happening when you are modifying it. Luckily, there exists an easy way to display the tree. This is done thanks to the dot language.

This language allows us to write down in a file a representation of your graph (a tree is simply an acyclic undirected graph) that can be processed by programs[7]. The syntax is as follows:

```
42sh ~ ls
Victim.dot
42sh ~ cat Victim.dot
graph Victim {
Victim -- ParentOne;
Victim -- ParentTwo;
ParentOne -- GrandParentOne;
ParentOne -- GrandParentTwo;
ParentTwo -- GrandParentThree;
ParentTwo -- GrandParentFour;
}
```

First, there is the type of the structure you are saving (which will be `graph` for this exercise). Then simply put the name of the graph, between curly braces list every link between each node and end each line with a semi-colon. Links between two nodes in the undirected graph are represented by `--`.

Write the ToDot method that translates the GenTree into a `.dot` file. The name of the file will be the name of its root.

```
public void ToDot();
```

[7]You can use this site to see your graph: `https://dreampuf.github.io/GraphvizOnline`

4.2 Suspects

In this section, Sherlock wants you to help him sort out his suspects according to the suspicion index that he assigned to each of them. To do so, he wants you to implement a decent sorting algorithm that is very reliable in terms of time complexity in both the worst and the best cases. But first it is important to transform the tree into an array.

4.2.1 Tree to array

```csharp
public Suspects(GenTree tree, int size);
```

This method is actually the constructor of the Suspects class. It goes through every node in a level order traversal and inserts each node encountered in an array of Tuples[8] containing the name and the suspicion index, the former being the only attribute of the class.

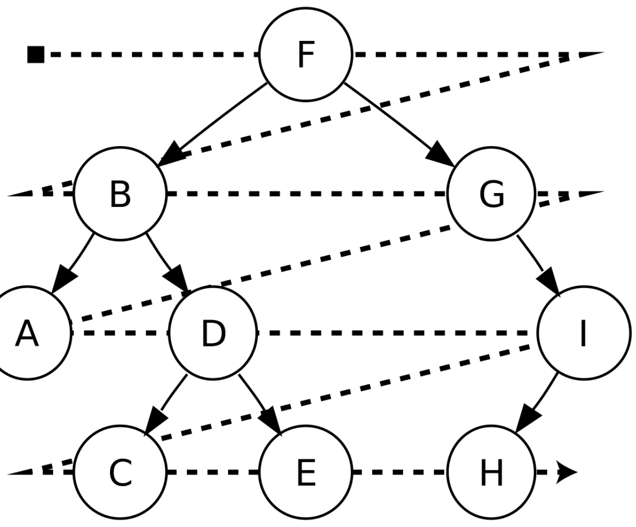


Figure 6: Level order traversal of the tree

[8]More information here:
`https://docs.microsoft.com/en-us/dotnet/api/system.tuple-2?view=net-5.0`

4.2.2 Heap Sort

Now that you have an array of suspects, it is time to sort. To do this, you will be using the Heap Sort algorithm.

```csharp
public void HeapSort();
```

A Heap is a data structure that takes the form of a tree and respects the heap property:

- If the heap is a max-heap, then the current node's key is always greater or equal to its children's keys.
- If the heap is a min-heap, then the current node is always smaller or equal to its children's keys.

Although it is explained as a tree data structure, it can also be represented as a one-dimensional array where for any node represented by the index *i*, then respectively the left child and the right child are located at indices *2 * i + 1* and *2 * i + 2*.

Specification

For the rest of the exercise, we will only be tackling max heaps.

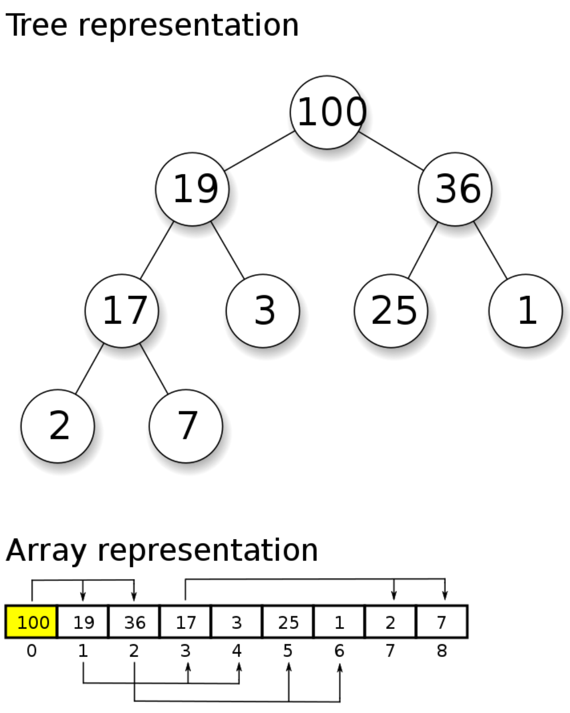


Figure 7: A binary heap with both its tree representation and its array representation

The Heap Sort algorithm is an in-place sort that utilizes the heap property, done by 'Heapifying' the array to respect the heap property at all times. The point of the sort is to always be able to get the first element (the 'root') and to add it to the end of the array. The steps can be described as followed:

Algorithm 1: HeapSort

```
BuildHeap(array) (Turn array into a max heap)
$i \leftarrow length(array) - 1$
while $i >= 0$ do
    Swap root of the heap with element at index i
    Heapify the subarray from start to i (in place)
    $i \leftarrow i - 1$
end
```

4.3 Bonus: back to S1

As you probably noticed by now, this practical was using a lot of recursion. You know what is perfect for recursion ? You guessed it: OCaml! Using the provided code in the file `Bonus/GenTree.ml` you can re-implement the functions:

```ocaml
val printTree : genTree -> unit = <fun>
(* Prints the tree passed as first parameter *)
val changeName : genTree -> string -> string -> genTree = <fun>
(* Changes the name of a individual in the tree.
The name to change is the first string while the new name
is the second one *)
val findPath : genTree -> string -> string list = <fun>
(* Returns a list of names corresponding to the path leading to a person
whose name is the second parameter in the tree given as the first parameter.
If no path is found, returns an empty list. *)
```

**There is nothing more deceptive
than an obvious fact.**