Exercise Sheet 1

In this exercise sheet you will implement a *binary tree storing integers* and follow some core principles of modular programming. The purpose of the exercise is to check (for yourself) whether you are sufficiently fluent in using the C programming language. Most exercises in this course will build directly on top of what you learned in the previous semester (mainly 703004 - Introduction to Programming).

If you are struggling with this exercise sheet, please consider doing this course at a later point in your studies. Otherwise you need to be prepared for the fact that subsequent exercises will probably require greater effort from you than originally planned by us.

Also note that task 1 is longer than the typical tasks encountered in this course. We will try to keep exercises concise and focused on a particular topic, but you are required to know about the underlying details when presenting. Therefore task 2 lists a bunch of questions we might typically ask someone during the presentation of task 1.

Task 1

Implement a module (`btree.h`, `btree.c`) providing a sorted binary tree for integers. The header file is already provided and holds the signatures of required functions alongside short descriptions. Your implementation should fulfill these descriptions as closely as possible.

The `btree` type is used as a *handle* to work with the whole tree, and can also be used as storage for information concerning the entire tree. A `btree_node` holds one node of the tree, in this case an integer and references to the left and right child nodes.

- Read this task from top to bottom before starting your implementation.
- Ensure you fully understand the concept of a sorted binary tree as well as the provided header file.
- Ensure your implementation does not suffer from any memory leaks using `valgrind`.
- Ensure your implementation compiles with `-Wall -Wextra -Werror -std=c99 -O2`.
- You should not be required to change any of the provided files. If you find an error, note it down and make *reasonable* assumptions to continue.
- Keep your code clean and simple. It should be as readable as possible.
- Everything (variable names, comments, etc) must be written in English.

- Implement `btree`, `btree_node` and the required functions in `btree.c`.
    - Your implementation may (and should) utilize additional functions not stated in the header file, but make sure they are not exported (e.g. by using `static`).
    - Only the `btree_print` function produces output during normal execution, debug messages / logs of any kind.
    - Try to reuse already implemented functions (e.g. `btree_remove` could make use of `btree_maximum`).
- Ensure proper error handling.
    - Use `assert` to ensure that `NULL` is not passed to function which can not handle this case.
    - Check return values of functions that may fail, like `malloc`.
- Compare the output produced by `btree_test.c` to `btree_test_output.txt` in order to check your implementation. A matching output does not guarantee a 100% correct implementation!
- Provide a `Makefile` which provides at least the rules `all` and `clean`.
    - Use the same compile flags as previously stated.
    - `all` produces the program `testvector`.
    - `clean` removes all generated files (please don't use wildcards here).
    - Your `Makefile` should model the dependencies between all files in this task correctly.
    - Each module should be compiled first. Linking them is the final step.
    - Utilizing implicit rules is recommended.
    - Don't forget about `.PHONY`.

Task 2

This task builds on top of the previous one, but no implementation is required. Its purpose is to ensure you understand the background behind the details of the previous task. You should be able to answer the following questions without the use of any additional notes. Presenting images and examples is fine, though not required.

- When would you use a sorted binary tree over a linked list?
- What would you change if the tree had to support not just integers, but any type of data?
- Why do we use modules instead of putting everything a single file?
- Is the development cycle (i.e. changing a source file, recompiling and running tests) faster when using modules? Explain your answer.
- What is a *header guard* and why is it needed?
- Why are `btree` and `btree_node` not *defined* in the header file? What are the implications?

- Explain the `const` in the parameter list of `btree_print`, is it required?
- Explain memory leaks. Why are memory leaks bad? You should know how to use `valgrind` to check your program for memory leaks.
- What is the reason behind writing *everything* in English?
- Why should you use `static` for *non-exported* functions?
- Why should we comment our source code? Is it always need? What should the comment state? What is *self-documenting code*?
- Why should the module not output debug messages?
- Why and when should you use `assert`?
- What are the benefits of using `make` over calling the compiler by hand?
- Imagine it was your job to design the interface for the btree module (writing `btree.h`). What would you have done differently, and why?

Recommended File Structure

```
ex01/
|-- README.md
|-- btree.c
|-- btree.h
|-- Makefile
|-- btree_test.c
`-- btree_test_output.txt
```