

Report Programming Project: COP 5536 Advanced Data Structures

Naman Arora
UFID: 3979-0439
University of Florida
naman.arora@ufl.edu

Abstract—This document is intended to report the procedures and completion of programming project “*Hashtag-counter*”. The source code attached is produced conforms to ISO C standard and is tested on GNU/Linux System with GNU GCC compiler toolchain version 9.3.0. The report first goes over how the source can be compiled and run and what the author expects as output. Next section explores some interesting features of the source code.

I. COMPILATION

A. Directory Structure

The directory structure of the repository is arranged in sub directories for each module. The whole project is connect via Makefile (GNU Make) utility for compilation. The major sub directories are:

- **The *src/* Directory:** This directory is the root of the project that contains all the sub-directories. It houses the root Makefile along with the main.c source file. The test and main binaries are produced here as a result of compilation.
- **The *src/mem/* Directory:** This directory houses the implementations of both hashmap and fibonacci heap. It also has the relay Makefile which further recursively calls the internal node Makefiles for the hmap/ and heap/ implementations. This has a header mem.h which is an API interface to the hmap implementation and the project as a whole. Also the header file heap.h is intended to be used internally within the hmap implementation and must not be included in caller programs.
- **The *src/mem/heap/* Directory:** This sub-directory resides within the src/mem/ subdirectory and houses the Fibonacci Heap implementation. It has the separated doubly linked list

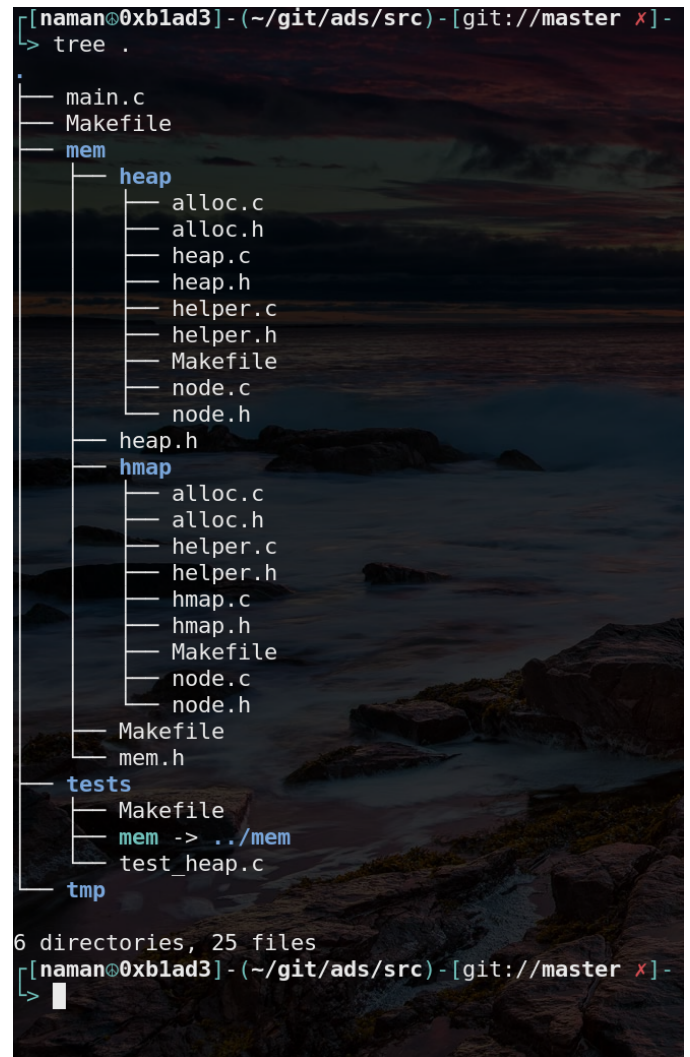
A terminal window with a dark background and a scenic image of a coastline at sunset. The terminal shows the command 'tree .' and its output, which is a directory tree. The tree starts with 'main.c' and 'Makefile' at the root. Below them is a 'mem' directory, which contains a 'heap' directory and a 'hmap' directory. The 'heap' directory contains 'alloc.c', 'alloc.h', 'heap.c', 'heap.h', 'helper.c', 'helper.h', 'Makefile', 'node.c', and 'node.h'. The 'hmap' directory contains 'alloc.c', 'alloc.h', 'helper.c', 'helper.h', 'hmap.c', 'hmap.h', 'Makefile', 'node.c', and 'node.h'. Below the 'mem' directory is a 'Makefile' and a 'mem.h' file. Further down is a 'tests' directory containing 'Makefile', 'mem -> ../mem', and 'test_heap.c'. At the bottom is a 'tmp' directory. The terminal also shows the command 'tree .' and the output '6 directories, 25 files'.

Fig. 1. Directory Structure

implementation in node.c,h, memory allocation and deallocation functions in alloc.h as well as the heap itself in heap.c,h along with a helper helper.c,h. It has the node Makefile for the heap

implementation.

- **The *src/mem/hmap/* Directory:** This subdirectory houses the implemented hash map, implemented from scratch to encapsulate the heap implementation. This hmap basically is a hybrid of Fibonacci heap and conventional hash map so that the API user has to care about a single data structure. This is also analogous to heap implementation in the way that the directory is structured.
- **The *src/tests/* Directory:** This directory was initially included to implement unit tests but due to fast approaching deadline, was just shortened to a single use case test of the heap with random number generator from Libsodium. The compilation of this test thus, requires libsodium as dependency and hence is made totally independent of the real program.

B. Compilation Instructions

The compilation requires GNU Make utility and GNU GCC compiler toolchain. It optionally requires Libsodium if tests/ is to be compiled, which is separate.

- The main binary *hashtagcounter*:
\$ make # To generate the binary
\$./hashtagcounter input_file_location output_file_location # To run
- The tests binary *test_out*:
\$ make tests # To generate tests
\$./test_out # To run the test

C. Output Expectation

The Output from the main binary is expected to be a CSV file/stream depending upon if the optional output file argument is passed. The output stream/file will have the most frequent *hashtags* as requested in the input file for each requested output.

The output from the test_out binary is supposed to a stream of lines each with a key and its output position in descending order.

II. THE CODE

Next we will gloss over the important highlights of the source code.

A. The Makefiles

The whole project is under the recursive compilation strategy of GNU Make utility and each subdirectory has its own makefile. There are three types of Makefiles,

- The Root Makefile: This is unique Makefile is from where the compilation process begins. It calls all the other types of Makefiles recursively from within.
- The Relay Makefile: This is present in the src/mem subdirectory and is just used to glue together the node Makefiles to the root one.
- The Node Makefile: There are three of these, viz. src/mem/heap/Makefile, src/mem/hmap/Makefile and src/tests/Makefile. They are the ones that actually create the object binaries.

B. The Hash Map

This is a pure implementation of the hash map including the hashing function in simplest of its form. This is intended to serve as the only visible entity to the end user and is designed with a first level API based structure. This encapsulates the underlying heap implementation. Some of the important functions/entities are:

- struct hmap: This is the main structure that encapsulates the hash map object and has embedded heap structure. Defined in src/mem/hmap/hmap.h
- struct hmap *hmap_init(char *, int): This function initializes the hash map and is required to be called for the very first time with the very first value to be inserted. Returns the generated struct hmap. Defined in src/mem/hmap/hmap.h.
- void hmap_deinit(struct hmap *): This function deletes the whole allocated memory of the hash map.
- void hmap_update(struct hmap *, char *, int): This function accepts the new value (or an existing one) and internally decides if the a similar node exists. This accordingly calls the heap insert or increase key functions.
- struct hmap_node *hmap_remove_max(struct hmap *): This function calls the underlying heap structure to remove the maximum value and returns it as an instance of the atomic map node, hmap_node.

- `struct hmap_node`: This is the atomic map instance of the hash map. It is defined in `src/mem/hmap/node.h` and contains the key value along with the pointer to the internal `heap_node` instance.
- `int hash_it(char *)`: The hashing function used for hash map. Defined in `src/mem/hmap/helper.h`.

C. The Fibonacci Heap

This is a pure implementation of the max Fibonacci heap and is housed in the subdirectory `src/mem/heap`. This is not intended to be used by end user and is coded as a level 2 API. It is an internal data structure for the previously defined hash map.

- `struct heap`: This is the main heap structure which basically contains a pointer to the current maximum node in the heap. This node is a pointer instance of `struct heap_node`. Defined in `src/mem/heap/heap.h`
- `struct heap *heap_init(int, char *)`: This is the initialization heap function, analogous to the `hmap`'s and returns the initialized `struct heap` instance.
- `void heap_de_init(struct heap *)`: This is a de-initialization function for the heap and in turn calls the traverse function to traverse in pre order format.
- `void heap_traverse(struct heap *, void (*fn)(struct heap_node *))`: This function is used to traversing the heap in post order format and apply the supplied "fn" function prototype to each level of doubly linked list of each tree.
- `struct heap_node *heap_remove_max(struct heap *)`: This function removes the maximum value of from the heap and consolidates it according to the Fibonacci heap algorithm.
- `struct heap_node *heap_meld(struct heap_node *, struct heap_node *)`: This melds two trees based on their keys while adjusting their degrees and returns the consolidated tree which is the root of both.
- `void heap_inc_key(struct heap *, struct heap_node *, int)`: This increases the key of the supplied node in the heap and adjusts the

heap in accordance with the cascading child cut algorithm of the Fibonacci heap.

D. The main driver

The main driver, in `src/main.c`, has two helper function, viz.:

- `void setup(FILE **, FILE **, int, char **)`: This function sets up the input and output file streams according to the supplied command line arguments. If no argument is provided for output file, the out file is set as `stdout`.
- `void process_file(FILE *, FILE *)`: This is the function that sets up the test hash map structure and reads the input file. This acts like an end used to the aforementioned API.

III. CONCLUSION

The programming assignment was successfully completed and was tested for 1M remove max operation on an Intel i78750H Hex core processor with 16Gb virtual memory. It took about 4 Seconds and used about 8% of the CPU. Similar and proportionally symmetrical results were obtained from testing on `thunder.cise.ufl.edu` server.

Further, this project can be refined via adding more decoupling to the two advanced data structures implemented upto a point where they can be used as independently with similar efficiency as being used together. The further updates will be pushed in the GitLab repository of the project:

<https://gitlab.com/r0ck3r008/ads>