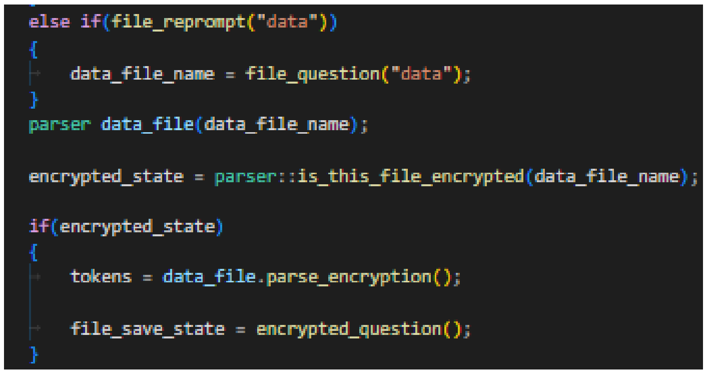
Ahmed Khan
Adam Freed
Professor Chenhansa
CS-124
6th April 2023

Post-Lab Report

Development Process: (mainly written by Ahmed)

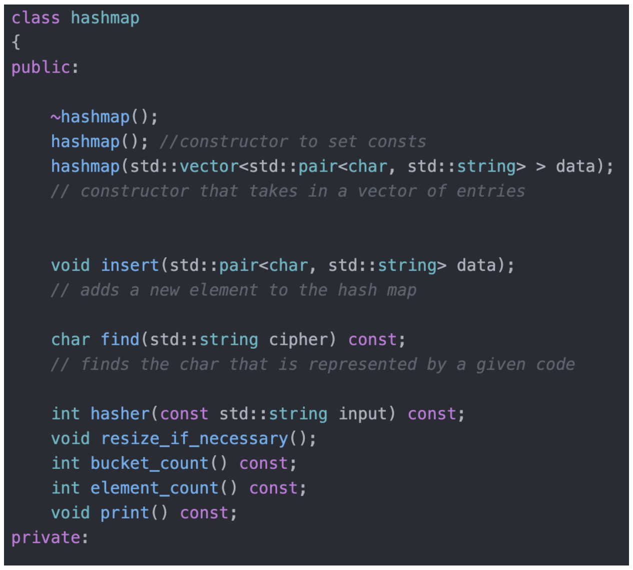
This lab differed from the previous ones due to it being a group project. This brings its own host of benefits and caveats. One would assume that two minds on one task makes it complete twice as fast, but often this is like saying 9 women can make a baby in a month. This project requires proper communication on the objective, as well as delegation of labor so there is parallel work that is more efficient than one person rather than conflicting work that is less than one person's efficiency. Therefore, a plan was set out before development on what needed to be done, a framework, a flow chart, and frequent consultation of the requirements. Secondarily, communication was established through Discord, which provided a proper interface to interact in a desktop setting, enabling the transmission of links, code, and bash commands as necessary. Third, version control was integral to the project. The problem with collaborative production is de-synchronization, as if each individual is working on their copy of the project and making their own changes, things quickly spiral out of hand. One person's bugs or unfinished code could render the entire project to a standstill until the issue is resolved, and this friction brings efficiency to below that of a single individual. I experienced this firsthand during the group project for CS-116, where my group had left me one night with a dysfunctional codebase that took me an all-nighter to untangle on top of a finals week, and they weren't even their top help. Group projects also require accountability, a show that each individual is lifting their weight and doing their part. This could not be done in real-time IDE's such as Replit. Such potential chaos was to be remedied by confronting the inevitable steep learning curve of a version control tool called Git. Through this we could delegate labor and work on individual tasks, committing them with a few lines of code after we knew it worked. When working on the same thing, the program helped us splice the code together in a manner that we almost never had to go back and check because nothing ever broke. These commits were recorded in a long and illustrious log, which could be visualized with a tree structure showing branches for feature development as well as merge conflicts. And the website showed commit history, frequency, and amount, showing when one person was or wasn't doing their part. Merge-conflicts did occur but they were resolved in short order through various strategies. What Git allowed us to do was have a say in how this was handled and have efficient commands to achieve it rather than either having no choice or having to do all of it manually. Lastly, the commit history provided a rich context for explaining the development process in retrospect. Being compelled to become familiarized with Git in a Bash and Linux environment is the professional standard, and it is good we are learning it now such that when we get to university and employment settings, it will be second nature.

We started with an online repository on git-hub and cloned it to our respective local environments. Things started small, initially used to house the flowchart and pre-lab report documents, such did provide the initial experiments with how collaboration and version control on non-code documents worked. Besides the main class that only contained two lines to call the program's wrapper class, there were several including the runtime class, hashmap and tree classes, and the parser class. Others were made in the future such as the cryptor class, but the aforementioned were created at the start. The runtime class is a wrapper class

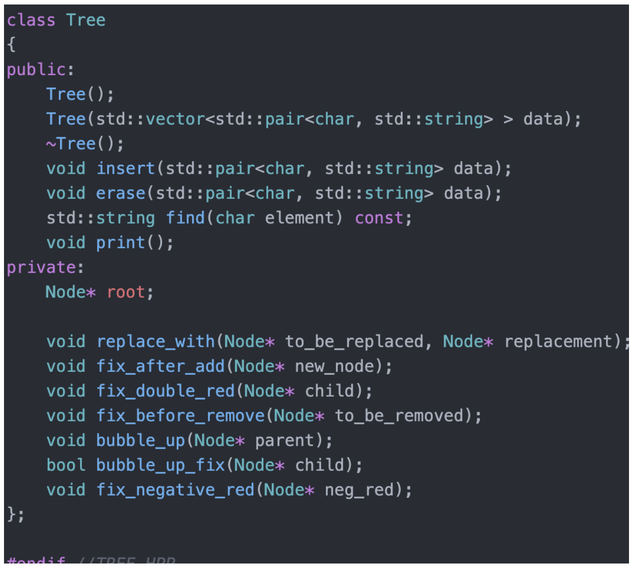


```
else if(file_reprompt("data"))
{
    data_file_name = file_question("data");
}
parser data_file(data_file_name);

encrypted_state = parser::is_this_file_encrypted(data_file_name);

if(encrypted_state)
{
    tokens = data_file.parse_encryption();

    file_save_state = encrypted_question();
}
```

that handles domain states, program flow, and data for the program at large. It is like a switchboard where things are connected from one place to another, and I only need to have a broad overview rather than being bogged down by specifics. It serves as a controlled and limited global scope, with extra functionality for being able to use class functionality in the context of the program itself. Evidenced in the image, the entire main loop consists of a single nested level, where runtime class data members are either input or output or operated upon by various classes, all of the cumbersome implementation hidden behind the function call. Above the main loop there are several console output functions which contain a litany of the various iostream code that is necessary to achieve both the clean and cohesive console output as well as the implementation in the main loop of the runtime class.



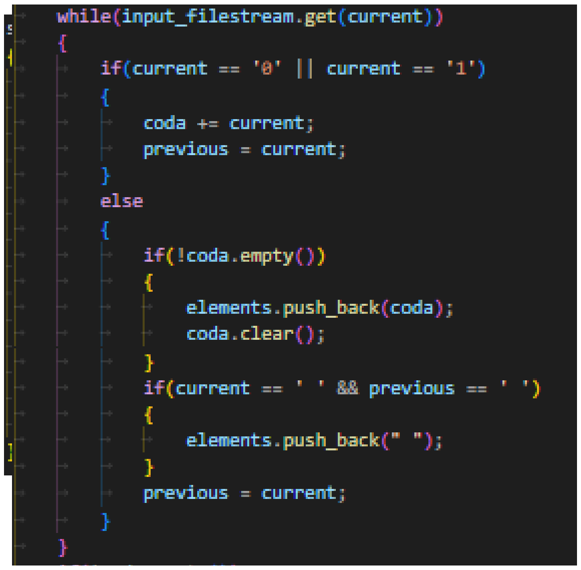
```
class hashmap
{
public:

    ~hashmap();
    hashmap(); //constructor to set consts
    hashmap(std::vector<std::pair<char, std::string> > data);
    // constructor that takes in a vector of entries


    void insert(std::pair<char, std::string> data);
    // adds a new element to the hash map

    char find(std::string cipher) const;
    // finds the char that is represented by a given code

    int hasher(const std::string input) const;
    void resize_if_necessary();
    int bucket_count() const;
    int element_count() const;
    void print() const;
private:
```



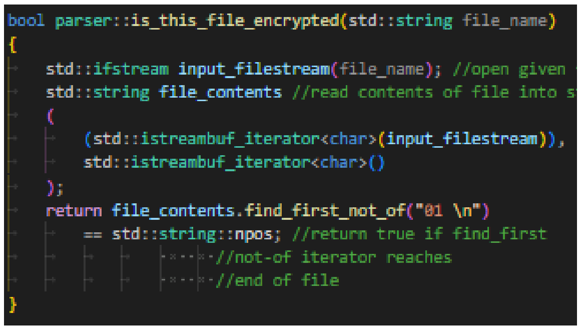
```
class Tree
{
public:
    Tree();
    Tree(std::vector<std::pair<char, std::string> > data);
    ~Tree();
    void insert(std::pair<char, std::string> data);
    void erase(std::pair<char, std::string> data);
    std::string find(char element) const;
    void print();
private:
    Node* root;

    void replace_with(Node* to_be_replaced, Node* replacement);
    void fix_after_add(Node* new_node);
    void fix_double_red(Node* child);
    void fix_before_remove(Node* to_be_removed);
    void bubble_up(Node* parent);
    bool bubble_up_fix(Node* child);
    void fix_negative_red(Node* neg_red);
};
```

The data structures came soon after as they were to exist as idealized implementations not bound to the rest of the program. We had the assumption that we could cannibalize a prior implementation of tree from a personal project, however we realized that a nestable container that did not have any automated insert functionality was not useful for our particular application of binary search so an implementation of red-black tree, according to the Big C++ textbook, was put in its place. Along such was a hash map that served as its complement. The decision of which data structure did came down to their implementations. It seemed better to have the hash-map used for decoding and the tree class for encoding, since the hash map relied on per-character string hashing, whilst the tree relied on key value comparisons. The alphabet being sequential became the key of the tree and was used to retrieve character codas, meanwhile the codas being binary strings made them the key of the hash map and was used to retrieve respective characters. This decision was also based on the fact that we didn't realize we could cast chars to ints for easy hashing.

Around the time of completing the data structures we ran into our first merge conflict (a visual depiction of such within the pitfalls section of the report), and this demonstrated the usefulness of Git. Ahmed had been making some major implementations on a tree branch, whilst there was also work going on our respective copies of main by Adam, and they all had to be reconciled eventually and they were through various commands, either letting the program decide what to keep, or overriding it, and on rare instances opening a merge editor. Almost always there were zero issues in the end.



```
while(input_filestream.get(current))
{
    if(current == '0' || current == '1')
    {
        coda += current;
        previous = current;
    }
    else
    {
        if(!coda.empty())
        {
            elements.push_back(coda);
            coda.clear();
        }
        if(current == ' ' && previous == ' ')
        {
            elements.push_back(" ");
        }
        previous = current;
    }
}
```



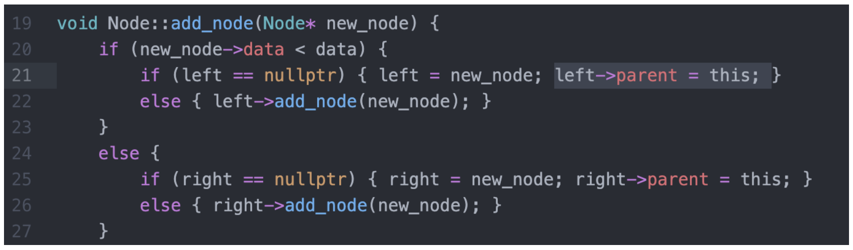
```
bool parser::is_this_file_encrypted(std::string file_name)
{
    std::ifstream input_filestream(file_name); //open given
    std::string file_contents //read contents of file into s
    (
        (std::istreambuf_iterator<char>(input_filestream)),
        std::istreambuf_iterator<char>()
    );
    return file_contents.find_first_not_of("01 \n")
        == std::string::npos; //return true if find_first
                            //not-of iterator reaches
                            //end of file
}
```

Once the data structures were completed, we began work on the parser, which relied on the data structures being fully functional to be useful. Parser has the most uses in the program by far. Interacting with files before or after an allocation of its object. It had two static functions, one to probe if the file existed through calling fstream::good(), and the other going through the text searching for non-whitelisted (other than 1, 0 or space) characters to determine if a file was encrypted or not. This affected major parts of program flow, as through the single data member “tokens” string vector would be the interface between parsing completely different types of data into a completely different format to different processing functions. If the file was encrypted, the parser would pull out individual character coda’s and dual spaces. If the file was decrypted it would take individual words from the text. Atop of all of this, the cipher class was parsed into a format of <char, string> pairs such that it could be used in both the tree and hash map data structures through the use of a string stream to break apart lines with an unspecified amount of spaces between the keys and values in the pair.

Once the data was obtained and the determination of its encryption or decryption status made, and the cipher and decipher data structures made from the parsed cipher class, they were sent to the cryptor wrapper class static functions to be processed in the appropriate manner. In the case of encryption, individual characters from words were translated via the tree into their respective codas and printed to a master string. In between sentences of codas indicating a word, another space was added to indicate an actual space, as this was the format presented in the original example encrypted file. In the case of decryption, individual codas were translated into characters, with spaces between them inserted through identifying dual spaces. Once these result strings were obtained, they were appended to the original file with the append flag std::ios_base::app in a parser member function, a class which already stored the file’s name. This removed the need to pass the name to the cryptor class and prevented file overwriting. The program was structured to allow the user to repeat the functionality of the program and be prompted whether they wanted to keep their original cipher or data files in either instance.

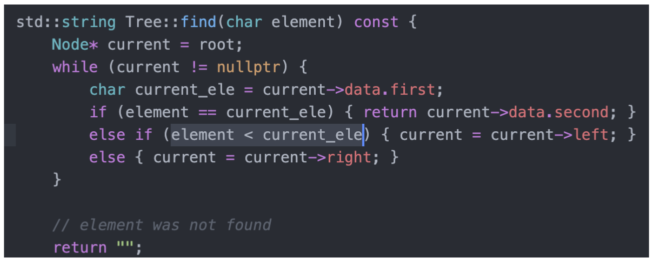
<u>Pitfalls:</u> (mainly written by Adam)

There was weird behavior when using the cryptor class as a dynamic object, it kept segfaulting. Many solutions were tried including altering the data members, passing various arguments by reference, et cetera. Doing the exact same thing with static member functions everything worked fine, so it must have been something to do with the class data members storing the token or the cipher and deciphering data structures. In retrospect it was most likely the fact that the data structure objects were not pointers, and the constructor didn't allocate any memory for the very large data structures as they were to be introduced, as they were not occupying dynamically allocated memory.



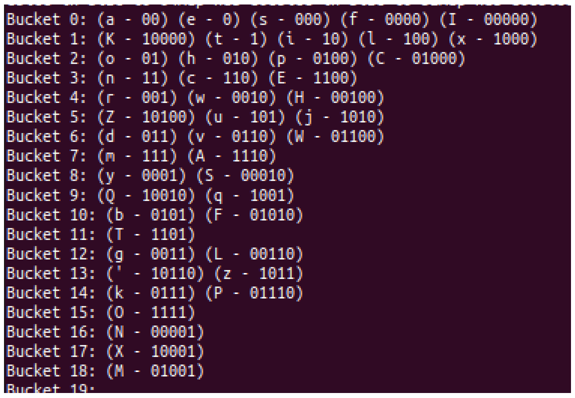
```
void Node::add_node(Node* new_node) {
    if (new_node->data < data) {
        if (left == nullptr) { left = new_node; left->parent = this; }
        else { left->add_node(new_node); }
    }
    else {
        if (right == nullptr) { right = new_node; right->parent = this; }
        else { right->add_node(new_node); }
    }
```

At first, our tree implementation did not work. Although it was supposed to be a red-black tree, it did not self-balance at all. It turned out that the function that added a Node to another one didn't properly set the new Node's parent, causing each Node's parent to be left as nullptr. Fixing this bug allowed for the red-black tree implementation to properly self-balance.



```
std::string Tree::find(char element) const {
    Node* current = root;
    while (current != nullptr) {
        char current_ele = current->data.first;
        if (element == current_ele) { return current->data.second; }
        else if (element < current_ele) { current = current->left; }
        else { current = current->right; }
    }

    // element was not found
    return "";
```

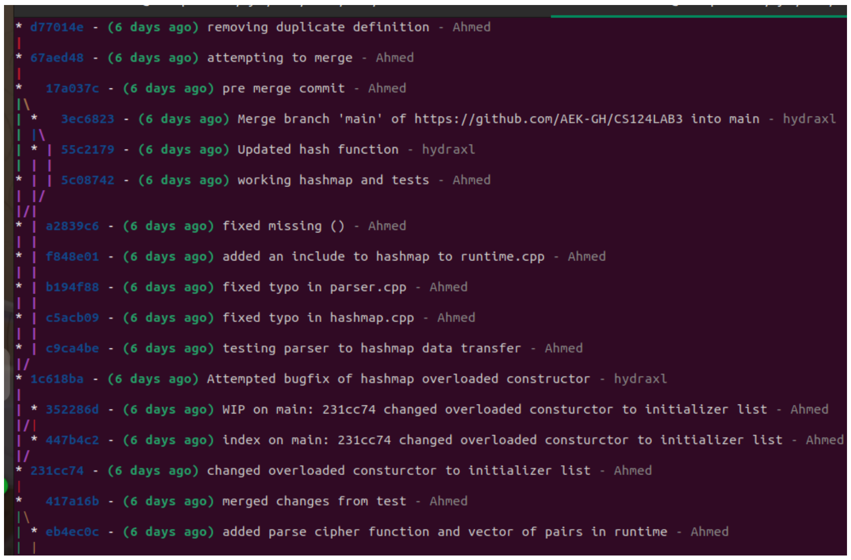
The tree class had an issue with its find function where it would search the wrong side of the tree, and then not find the item. Instead of going right, it would go left, and vice versa. This was fixed by turning a > into a <.



```
Bucket 0: (a - 00) (e - 0) (s - 000) (f - 0000) (I - 00000)
Bucket 1: (K - 10000) (t - 1) (i - 10) (l - 100) (x - 1000)
Bucket 2: (o - 01) (h - 010) (p - 0100) (C - 01000)
Bucket 3: (n - 11) (c - 110) (E - 1100)
Bucket 4: (r - 001) (w - 0010) (H - 00100)
Bucket 5: (Z - 10100) (u - 101) (j - 1010)
Bucket 6: (d - 011) (v - 0110) (W - 01100)
Bucket 7: (m - 111) (A - 1110)
Bucket 8: (y - 0001) (S - 00010)
Bucket 9: (Q - 10010) (q - 1001)
Bucket 10: (b - 0101) (F - 01010)
Bucket 11: (T - 1101)
Bucket 12: (g - 0011) (L - 00110)
Bucket 13: (' - 10110) (z - 1011)
Bucket 14: (k - 0111) (P - 01110)
Bucket 15: (O - 1111)
Bucket 16: (N - 00001)
Bucket 17: (X - 10001)
Bucket 18: (M - 01001)
Bucket 19:
```

The hashmap class always had an issue with its hashing function and bucket distributions, perhaps from a lack of understanding in how to derive unique hashes from binary strings. Every attempt would keep around a baseline of 3-4 elements in the first several buckets, or worse with like 6 or 7 in one and diminishing amounts further down. (displayed in the image on the left). Eventually 3-4 elements each in several buckets was settled on as no more possible improvements could be obtained.

The hashmap class had repeat issues with the find function. After a while of debugging, we realized it was adjusting some of the Nodes in the hash instead of just passing them if the first Node in the array was not the one being looked for.



```
* d77014e - (6 days ago) removing duplicate definition - Ahmed
|
* 67aed48 - (6 days ago) attempting to merge - Ahmed
|
*   17a037c - (6 days ago) pre merge commit - Ahmed
|\
| *   3ec6823 - (6 days ago) Merge branch 'main' of https://github.com/AEK-GH/CS124LAB3 into main - hydraxl
| |\
| * | 55c2179 - (6 days ago) Updated hash function - hydraxl
| | |
* | | 5c08742 - (6 days ago) working hashmap and tests - Ahmed
| |/
|/|
* | a2839c6 - (6 days ago) fixed missing () - Ahmed
| |
* | f848e01 - (6 days ago) added an include to hashmap to runtime.cpp - Ahmed
| |
* | b194f88 - (6 days ago) fixed typo in parser.cpp - Ahmed
| |
* | c5acb09 - (6 days ago) fixed typo in hashmap.cpp - Ahmed
| |
* | c9ca4be - (6 days ago) testing parser to hashmap data transfer - Ahmed
|/
* 1c618ba - (6 days ago) Attempted bugfix of hashmap overloaded constructor - hydraxl
|
| * 352286d - (6 days ago) WIP on main: 231cc74 changed overloaded consturctor to initializer list - Ahmed
|/|
| * 447b4c2 - (6 days ago) index on main: 231cc74 changed overloaded consturctor to initializer list - Ahmed
|/
* 231cc74 - (6 days ago) changed overloaded consturctor to initializer list - Ahmed
|
*   417a16b - (6 days ago) merged changes from test - Ahmed
|\
| * eb4ec0c - (6 days ago) added parse cipher function and vector of pairs in runtime - Ahmed
| |
```

We ran into several merge conflicts when we both had been working on code at the same time. Due to our inexperience with github, each of these required thought and effort to resolve, and often brute force methods that fortunately did not shoot us in the foot each time.

Possible improvements: (contributions by both)

Once the files have been overwritten in either instance they are treated as unencrypted and are no longer suitable for use by the program. Perhaps this is the way it should be, or perhaps the program should have a way to identify possible metadata with functionality to run the same file through a different cipher using the preexisting file name in the parser class. The reusability of the main loop does not coincide with the lack of reusability with the files, and therefore this is a possible improvement.

Unknown characters aren't handled by the program; if the program encounters a character it doesn't recognize when encoding, it inserts an empty string, aka deleting that unknown char from the encryption, and it will be entirely missing on a subsequent decryption. Conversely, when decoding, if the program encounters an unknown coda it inserts an underscore. Perhaps the program could insert a representation of an underscore in the program, but such would have to be hardcoded and might conflict with the cipher, which might also have a code for the underscore.

Our code does not handle files with multiple lines properly. Newline characters are ignored, meaning if we encrypt a file with multiple lines, then decrypt the result, everything will be placed on the same line.

This program does not have any functionality for huffman compression, because it exists purely to translate between codas and chars, rather than take in a continuous stream of binary and traverse a tree. Perhaps this is due to the fact that the initial cipher we were given took an ascii approach to encoding characters rather than a huffman approach. Just as a thought experiment, any huffman coda that contains a substring that could represent another character, it would break since the tree traversal would reach that other character first. And this was the case in the given cipher.txt.

If not for the requirements of the lab, we could have used hash tables for both encryption and decryption. And even with the requirements we could have used a hash table for encryption and the tree for decryption, since casting chars to ints provides a hash function that guarantees no collisions for any char within the unicode standard, unless the array size is below 128. Even in an array of size 128, it still minimizes the collisions because the letters are all consecutive, so they would be evenly spread after a modulus operation. Our hash function for decryption did contain collisions, such as 00 and 000 both being placed at index 0.