HashTable:

```cpp
//
// Created by popca on 19/10/2022.
//

// github: https://github.com/calin2110/FLCD

#pragma once
#include <any>
#include <fstream>

/*
 * Node struct that represents the Node of a LinkedList
 * symbol:      std::any which is either a string, int or bool,
 *              representing the symbol of the Node (identifier, string const, int const,
bool const)
 * position:    int representing the position in the HashTable of the symbol
 * next:        Node* representing the next element in the LinkedList
 */
typedef struct Node {
    std::any symbol;
    int position;
    Node* next;
} Node;

class HashTable {
private:
    // int representing the capacity of the hash table
    int quotient;
    // Node** representing an array of linked lists (Node*) of capacity quotient
    Node** hash_table;
    // int representing the position we're currently at in the hash table
    int current_position;
    // int representing the number of the elements currently in the hash table
    int size;
    // double representing the max load factor which we accept
    double max_load_factor;

public:
    /*
     * Given a quotient and a max_load_factor, we initialise the symbol table
     *
     * @param quotient:         int representing the max capacity of the hash table
     * @param max_load_factor:  double representing the max load factor allowed by the
hash table
     */
    explicit HashTable(int quotient = 647, double max_load_factor = 0.75);

    /*
     * Given a hash table, we want to assign it to the current hash table correctly
     * so that both can be destructed without any memory errors
```

```
     * (this is actually needed in order to not have any memory leaks)
     */
    HashTable& operator=(const HashTable& other);

    /*
     * Given a symbol, which is either a std::string, int or bool condensed into an
std::any,
     * we are trying to add it into the hash table
     *
     * @param symbol:   std::any representing the symbol we want to add into the table
     * @return:         if the symbol already exists in the table, we return it
     *                  otherwise, if it's of the right type, we add it to the beginning
of the linked list of its hash
     *                  and we update the current_position and size
     *                  if the size is greater than max_load_factor of capacity, we
resize our hash_table
     *                  then we return the added Node
     * @throws:         if the symbol is NOT int, std::string or bool, we throw an
std::invalid_argument
     */
    Node* add(const std::any& symbol);

    /*
     * Given a symbol, which is either a std::string, int or bool condensed into an
std::any,
     * we are trying to search it into the hash table
     *
     * @param symbol:   std::any representing the symbol we want to add into the table
     * @return:         if we cannot find our symbol in the linked list of its hash, we
return nullptr
     *                  otherwise, we return it
     */
    [[nodiscard]] Node* search(const std::any& symbol) const;

    /*
     * Given a hashtable, we want to output it to the ostream given
     * This is used to print the HashTable nicer to the file after using the scanning
algorithm
     */
    friend std::ostream &operator<<(std::ostream &os, HashTable& hashTable);

    /*
     * Given the symbol table, we deallocate all dynamically allocated memory
     */
    ~HashTable();

private:
    /*
     * Given a symbol, which is either a std::string, int or bool condensed into an
std::any,
     * we are trying to hash it
     *
     * @param symbol:   std::any representing the symbol we are trying to hash
```

```
     * @return:         depending on the type of symbol, we apply a different hash
function
     *                  which returns an unsigned int value between [0, quotient - 1]
     *                  value is a std::string => hash value will be the remainder of the
sum of the ascii codes
     *                                            of all characters of the std::string
when divided by quotient
     *                  value is a boolean => hash value will be 0 if the value is false
or 1 otherwise
     *                  value is an int => hash value will be the value of the remainder
when dividing
     *                                        the int by quotient
     * @throws:         std::invalid_argument if the type of the symbol is neither int,
bool or std::string
     */
    [[nodiscard]] unsigned int hash(const std::any& symbol) const;

    /*
     * When the load factor of the hash table becomes greater than max_load_factor, we
are creating a new hash table
     * with double the size and readd the elements we currently have
     * then we delete the old hash table
     */
    void resize();

    /*
     * We initialise the hash table by creating an array of Node* of size quotient
     * and at each position we are putting nullptr
     */
    void initialise_hash_table();

};

/*
 * Given two symbols, of type std::any, but both being either int, bool, or std::string
at core,
 * we want to check whether their values are equal or not
 *
 * @param symbol1:  first symbol, of type std::any
 * @param symbol2:  second symbol, of type std::any
 * @return:         true, if their types are the same and the values are the same
 *                  false, if either their types are different or the values are
different
 * @throws:         std::invalid_argument if the types are not int, std::string or bool
 */
bool symbols_are_equal(const std::any& symbol1, const std::any& symbol2);
```

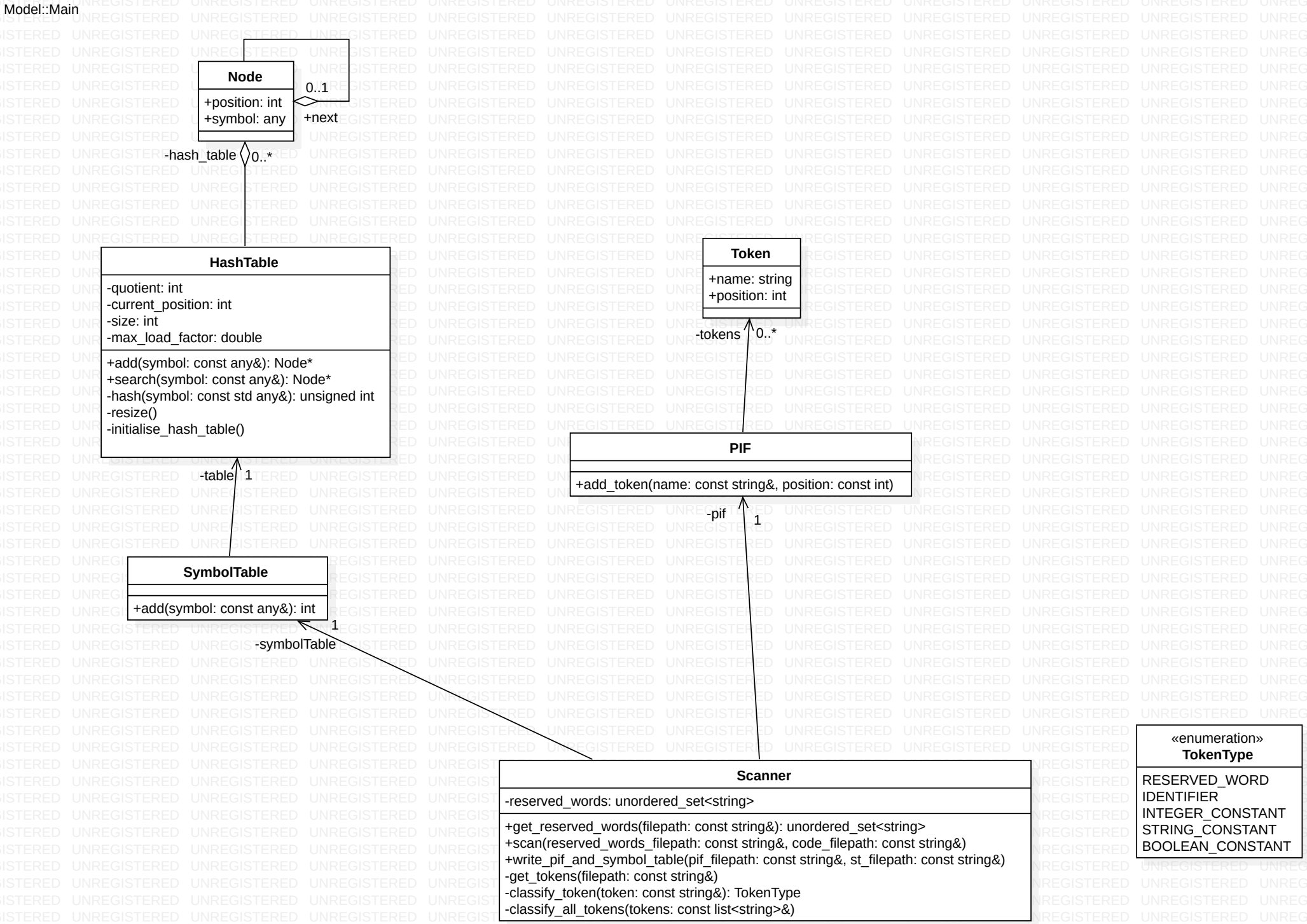
PIF:

```
//
// Created by popca on 26/10/2022.
//

#pragma once
#include <list>
#include <fstream>
#include "Token.h"

class PIF {
private:
    /*
     * PIF uses the data structure of a std::list because the tokens need to be added in
order they are found
     * so a linked list would be the most efficient, as adding to the end of a linked
list is a O(1) operation
     * and luckily, in CPP, std::list is implemented as a linked list
     */
    std::list<Token> tokens;

public:
    PIF();

    void add_token(const std::string& name, const int position);
    friend std::ostream &operator<<(std::ostream &os, PIF& pif);
};
```

Token:

```cpp
//
// Created by popca on 26/10/2022.
//

#include "Token.h"
#include <regex>

bool is_token_identifier(const std::string& token) {
    /*
     * This regex represents:
    * line starting with a letter (lower case or upper case) or underline
     * followed by a character which is either letter (lower case or upper case),
underline or digit
     * 0 or more times and then the end of the line
     */
    std::regex regex{"^[A-Z|a-z|_][A-Z|a-z|_|0-9]*$"};
    return std::regex_match(token, regex);
}

bool is_token_integer_constant(const std::string& token) {
    /*
     * This regex represents:
     * line starting with 0 and then end of line
     * or line starting with + or - or nothing, then a non-zero digit
     * followed by a digit 0 or more times and then the end of the line
     */
    std::regex regex{"^(0|^[+|-]{0,1}[1-9][0-9]*)$"};
    return std::regex_match(token, regex);
}

bool is_token_string_constant(const std::string& token) {
    /*
     * This regex represents:
     * line starting with ", then either ~ ! @ # $ ^ , . ? _ or any letter (lower case or
upper case) or any digit
    * 0 or more times, then another " and then the end of the line
     */
    std::regex regex{"^\"[~|!|@|#|$|^|,|.|?|_|A-Z|a-z|0-9]*\"$"};
    return std::regex_match(token, regex);
}

bool is_token_boolean_constant(const std::string& token) {
    /*
     * A boolean constant can be either true or false, so we do not need any regex there
     */
    return token == "true" || token == "false";
}
```

```
Token::Token(const std::string &name, const int position) {
    this->name = name;
    this->position = position;
}
```


Model::Main
Node
+position: int
+symbol: any
0..1
+next
-hash_table
0..*
HashTable
-quotient: int
-current_position: int
-size: int
-max_load_factor: double
+add(symbol: const any&): Node*
+search(symbol: const any&): Node*
-hash(symbol: const std any&): unsigned int
-resize()
-initialise_hash_table()
Token
+name: string
+position: int
-tokens
0..*
PIF
+add_token(name: const string&, position: const int)
-table
1
-pif
1
SymbolTable
+add(symbol: const any&): int
1
-symbolTable
Scanner
-reserved_words: unordered_set<string>
+get_reserved_words(filepath: const string&): unordered_set<string>
+scan(reserved_words_filepath: const string&, code_filepath: const string&)
+write_pif_and_symbol_table(pif_filepath: const string&, st_filepath: const string&)
-get_tokens(filepath: const string&)
-classify_token(token: const string&): TokenType
-classify_all_tokens(tokens: const list<string>&)
«enumeration»
TokenType
RESERVED_WORD
IDENTIFIER
INTEGER_CONSTANT
STRING_CONSTANT
BOOLEAN_CONSTANT