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

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

#pragma once
#include <any>

/*
* 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 SymbolTable of the
symbol
* next:        Node* representing the next element in the LinkedList
*/
typedef struct Node {
   std::any symbol;
   int position;
   Node* next;
} Node;

class SymbolTable {
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
    */
   SymbolTable(int quotient, double max_load_factor);

   /*
```

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

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]
    * @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);
```