FLCD - Symbol Table Data Structure Documentation

Source code repository: https://github.com/914-marcu-paul/FLCD-LAB3

In our implementation, the symbol table is represented by a hash table, being unique for both identifiers and constant values (a single symbol table instance in-memory).

The symbol table is written in Python 3.10 and consists of two separate classes, namely:
- The hash table class (detailed in hashtable.py), which represents the underlying data structure that saves the symbol table's representation;
- The symbol table class (detailed in symtable.py), which represents the abstraction of the symbol table (wrapping the hash table) and is called upon when operations involving it are needed.

The symbol table is constructed with a given size (representing the N number of entries in the hash table) and permits the following operations:
- add(key): adds the given key (symbol) to the hash table;
- contains(key): checks if the given symbol already exists in the hash table;
- remove(key): gets rid of the given symbol from the hash table;
- get_position(key): gets the position at which the key is located inside the hash table.

These operations call upon methods in the underlying hashtable class, where they are implemented.

As for the hash table itself, it is implemented, at a base level, through an array of doubly-linked lists (in our code we use the deque object from the Python collections module), which permits us to resolve hashing conflicts in a relatively simple manner (if two symbols hash to the same M in the 0-N indexing range of the table, they both get added to the doubly-linked list occupying the M-th slot of the deque array).

For our hashing function, we use the multiplication method with a subunitary constant included in the class fields (our hashmod). As such, for any given symbol, we simply multiply its value by the hashmod, take the fractional part of the resulting value, multiply it by the span of the table (N) and floor it to the closest integer value in order to determine the appropriate index. The simple formula is as follows: *[N * {k * hashmod}]*.
Addendum: this is written following the mathematical notations corresponding to the integer part of a given x being [x], and the fractional part being {x}.

The implementation of the add(key) method first checks if the hash table already contains the given symbol (since symbols have to be unique in a namespace) and, if not, it gets appended to the deque at the relevant index.

The contains(key) method simply returns the result of the verification of whether or not the deque at the symbol's hash contains the given symbol already, and the remove(key) method gets rid of the symbol if it is found.

The get_position(key) method returns a tuple consisting of the hash index where the symbol is found, as well as its depth in the respective doubly-linked list. This can be thought of as the X and Y coordinates of the irregular matrix that is the hash table in this implementation.

The hash table's implementation, while capable of resolving conflicts with relative ease, does not define a load factor, nor a way of rebalancing the hash table if it becomes too "lopsided", as this was not part of the laboratory requirements, and, frankly, I forgot about putting that in.