CA1 GROUP7

The first computing assignment of CS225, data structure. The implementation of **a two-stack abstract machine (2SAM)** for processing query requests on a list-based object store.

Getting Started

This abstract machine consists mainly four parts: the **database.py** to store the list-based object. the **semanticAnalysis.py** to analysis the input queries, splitting and translating them for evaluate. **stack.py**, the ADT stack used in implementation. machine.py, the machine itself that processes the query requests

Prerequisites

To run the assignment, the following environment is required

```
python 3.7.6
```

Usage

```
$ python3 machine.py
```

when executing in python console

```
:o -- output the results
:q -- quit the program
:i -- read a new query from terminal, which will be evaluated as the next query
:h -- help
```

the list-based data is stored in ./data.txt
the queries is stored in ./queries.txt
you can also write your own commands, here is the grammar

```
"PERFORMACE where(&cinema = Flora)"
"deref(PERFORMANCE where(&cinema = ((THEATRE where(&address =
Old_Village)).cinema)))"
"deref(eqjoin(PERFORMANCE cinema THEATRE cinema))"
```

1. **When using "where" in the query which represents the form "Q where phi", the & in the &cinema is necessary which represents the meaning of a keyword for PERFORMANCE. It tells the code to view the cinema not as plain text.**
2. Add brackets for functions and expressions, but the white space doesn't matter.
3. **The deref() here can read any kind of nested id lists can return the value, you don't have to dereference it twice because we have done it recursively for you.**
4. The equal join has the format: list1 name1 list2 name2. The names here denotes the way of doing equal join.
5. **No commas between parameters.**
6. See the function mapping dictionaries for the name of each operations.

7. Read **queries.txt** for more information about grammar.

Anaconda (Windows) Notes

If you are using Anaconda in Windows, using 'python3' might not work. Use 'python' instead.

Design Logic

Database

The database is based on an improved hash set, with uniquely generated id numbers to denote their positions. The database simulates the TRIE data structure to build a index tree. You can find the id numbers of sub-tree in the roots so that you can access any parts of the database at any depth.

By saying the index tree here, we basically mean that entries like PERFORMANCE whose value is actually a set are viewed as roots of the database. The strength of this structure is that we can always build the database recursively to make the object store at any depth you want.

Stack based input analysis

The input are pure strings, so we used an notation of input expression called "Reverse Polish notation". To realize it, we defined the priority of all operations. Basically we can transfer the input string "PERFORMANCE where(&cinema = Flora)" to ["PERFORMANCE", "&cinema", "Flora", "=", "where"]. Then we are able to process the instruction one by one.

Note that the "&" mark here is necessary which means to iterate the PERFORMANCE list and get attributes called "cinema", there should be no white space between "&" and "cinema".

Machine

The RES Stack is used to store the results of calculation and the ENV stack is used to store id numbers while processing queries.

The function evaluate() is the core of making function calls and the analysis the command in Reverse Polish notation. This function will loop over the id numbers in RES and ENV stack (if necessary) and recursively pop out currently needed command from the command list.

Operations

Nearly no needs of passing parameters between functions, all are based on RES stack.

Running the tests

The program is expected to execute as follows:
Note that all our tests are based on the the file queries.txt.
You can run though the machine.py code and input the ":o" which means the output to see the result of the test file.
Or you can also input ":i" to run your own tests.

The list-based data used for tests

```
["THEATRE", [["cinema", "Abaton"], ["address", "Grindle_Alley"]]]
["THEATRE", [["cinema", "Flora"], ["address", "Old_Village"]]]
["THEATRE", [["cinema", "Holi"]]]
["PERFORMANCE", [["cinema", "Flora"], ["title", "The_Piano"], ["date",
"May_7"]]]
["PERFORMANCE", [["cinema", "Holi"], ["title", "Manhattan"]]]
["PLAY", [["title", "The_Piano"], ["director", "Campio"], ["price", 10]]]
["PLAY", [["title", "Manhattan"], ["director", "Allen"], ["price", 15]]]
["NATIONALITY", [["director", "Campio"], ["country", "USA"]]]
["NATIONALITY", [["director", "Allen"], ["country", "USA"]]]
```

The operations supported

our abstract machine supports the following operators

```
aggregation operators: count, sum, min, max, average
arithmetic operators: +, −, ∗, /
comparison operators: <, >, ≤, ≥, ==, !=
other operators: In, distinct, deref, where(our selection method), projection,
cartesian, equaljoin
```

detailed tests

```
query = "PERFORMANCE where(&cinema = Flora)"
output: [9]
```

```
query = "deref(PERFORMANCE where(&cinema = Flora))"
output: ['Flora', 'The_Piano', 'May_7']]
```

```
query = "deref(eqjoin(PERFORMANCE cinema THEATRE cinema))"
output: [['Flora', 'The_Piano', 'May_7', 'Flora', 'Old_Village'], ['Holi',
'Manhattan', 'Holi']]
```

```
query = "(NATIONALITY where(&director = Campio)).country"
output: ['USA']
```

```
query = "PERFORMANCE where(&cinema = GZM)"
output: []
```

```
query = "(PERFORMANCE where(&cinema = Flora)) In [9,10,11]"
output: ['True']
```

```
query = "distinct([2,3,4,4,5,5,6,77,77,88])"
output: [2, 3, 4, 5, 6, 77, 88]
```

```
query = "distinct(NATIONALITY.country)"
output: ['USA']
```

```
query = "deref(THEATRE)"
output: [['Abaton', 'Grindle_Alley'], ['Flora', 'Old_Village'], ['Holi']]
```

```
query = "NATIONALITY x PLAY"
output: [[16, 24], [20, 24], [16, 27], [20, 27]]
```

```
query = "deref(NATIONALITY) x deref(NATIONALITY)"
output: [[['Campio', 'USA'], ['Campio', 'USA']], [['Allen', 'USA'], ['Campio',
'USA']], [['Campio', 'USA'], ['Allen', 'USA']], [['Allen', 'USA'], ['Allen',
'USA']]]
```

```
query = "(PLAY where (&director = Campio)).price > (PLAY where(&director =
Allen)).price"
output: [False]
```

```
query = "(PLAY where (&director = Campio)).price < (PLAY where(&director =
Allen)).price"
output: [True]
```

```
query = "deref(PERFORMANCE where(&cinema = ((THEATRE where(&address =
Old_Village)).cinema)))"
output: [['Flora', 'The_Piano', 'May_7']]
```

```
query = "deref(PLAY where (&price < 12))"
output: [['The_Piano', 'Campio', 10]]
```

```
query = "deref(PLAY where (&price < 20))"
output: [['The_Piano', 'Campio', 10], ['Manhattan', 'Allen', 15]]
```

```
query = "(6 + 3)*2"
output = [18.0]
```

```
query = "(7 + 8)/3"
output = [5.0]
```

```
query = "max([7,3,4])"
output = [7]
```

```
query = "min([2,3,4,5])"
output = [2]
```

```
query = "count(PLAY where(&price > 5))"
output = [2]
```

Time Complexity Analysis

EQ_join

For the equijoin operation, we do not use the basic expression $(\text{Q1 x Q2})\ \text{where}\ (\text{name1} = \text{name2})$, which needs complexity of $O(n^2)$, but use a single operation $eqjoin$ with an optimized implement based on **sorting**.

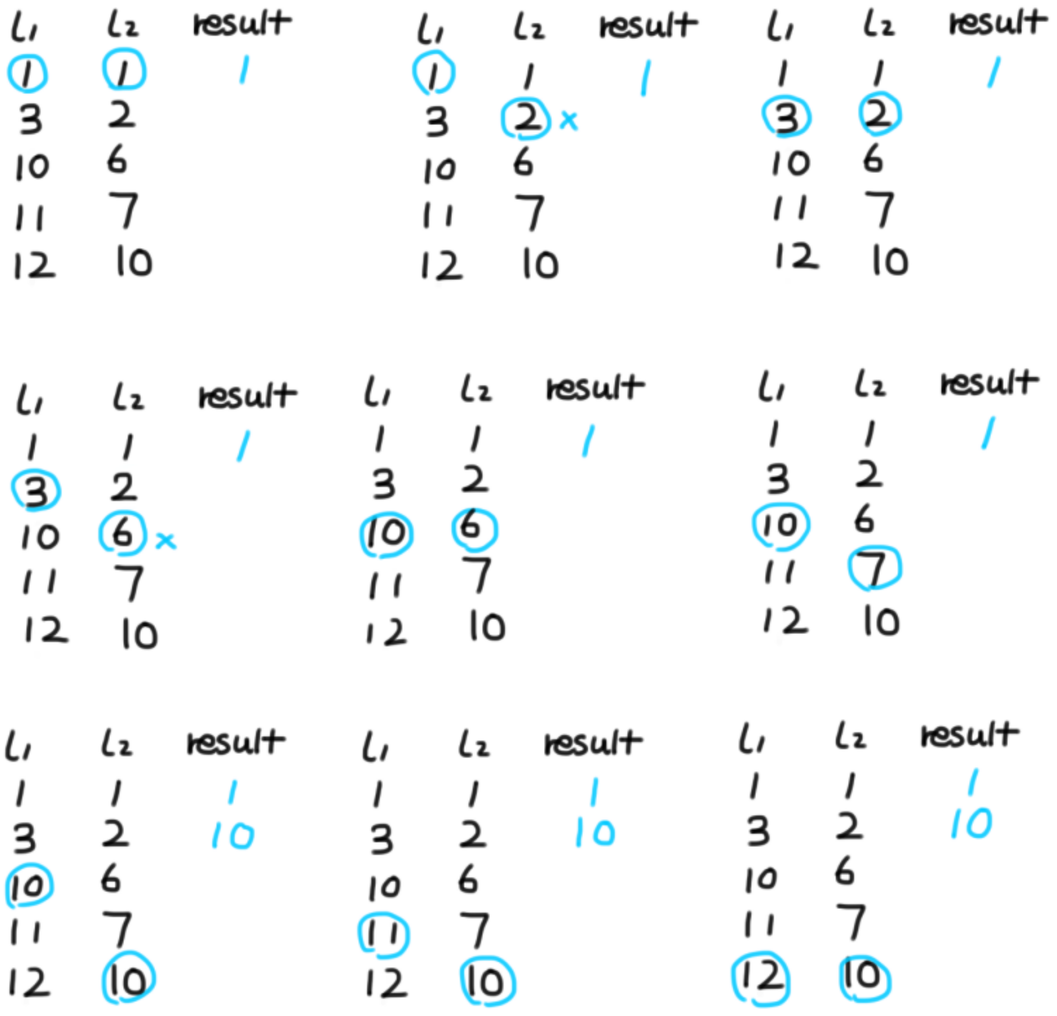
Here is our algorithm:

1. Sort records in both lists corresponding to their expected external names (name1 / name2) respectively.

   Here, we use the **merge sort**, which needs complexity of $O(nlog(n))$
2. Traverse one list. When we get a new element (call it A), start traverse another list.
   - When the elements in the other list (call it B) is equal to A corresponding to their expected external names, i.e. **val(A.name1) = val(B.name2)**, combined A and B together and put it into the resulting list.
   - Once the elements in the other list (call it B) is bigger than A corresponding to their expected external names, i.e. **val(A.name1) < val(B.name2)**, record the position of current element in the other list, then go into the next loop.
3. In the next loop, start from the position recorded in the previous loop.

Because two lists are ordered, we can guarantee that **elements after the recorded position are distinct with A** and **only the elements after the recorded position are likely equal to A.**

An small example:



The complexity of the comparing part is $O(n)$, because we only need to traverse over one list and part of another list.

Therefore, the total complexity of the algorithm is $O(n) + O(nlogn) = O(nlogn)$, which is more efficient than the origin method.

Quick database access

The core of the database is its efficient interface functions. By recursively building the database, we are able to make a record of all the nodes. For example, by giving the depth and the name to database, you can immediately read all the external names and their id numbers.

The id numbers can be used in the hash set to calculate index which makes it possible for us to read the entry. After initializing the database, searching in it has time complexity of O(1) because we already recorded all the information needed.

Note that out database currently support the nested lists of depth = 3. It's trivial to add more depth. Below is the self.roots (one dictionary with depth and names as keys) in database:

{0: {'THEATRE': [1, 4, 7], 'PERFORMANCE': [9, 13], 'PLAY': [16, 20], 'NATIONALITY': [24, 27]}, 1: {'cinema': [2, 5, 8, 10, 14], 'address': [3, 6], 'title': [11, 15, 17, 21], 'date': [12], 'director': [18, 22, 25, 28], 'price': [19, 23], 'country': [26, 29]}, 2: {}}

Authors

Group 7

- **Zhu Zhongbo** - *3180111635*
- **Guan Zimu** - *3180111630*
- **Xie Tian** - *3180111631*
- **Yang Zhaohua** - *3180111374*

Contact us

your communication really makes a difference

- Zhongbo.18@intl.zju.edu.cn
- Zhaohua.18@intl.zju.edu.cn
- tianx.18@intl.zju.edu.cn
- zimu.18@intl.zju.edu.cn