October 26, 2022
Project 1: Ledger
double-entry bookkeeping using Fortran pointers & recursion

Aim

In this project you must demonstrate your knowledge of Fortran pointers and recursion both in dynamic data structures and subprograms.

You must write a program that creates a dynamic structure similar to a binary search tree (BST) to store names of deities. Each tree node must additionally contain two dynamic sorted lists to store this deity’s debtors and creditors with corresponding amounts. This entire structure describes a ledger with double-entry bookkeeping. When the whole tree is printed, the names of deities, their debits, and credits are printed in the alphabetical order owing to the order of nodes in the tree and in the both sorted lists of each node.

You program should read a list of input lines, build the tree of nodes in the BST order, add amounts either to the list of debits or credits depending on the transaction, print the entire bookkeeping, print the net debit and credit, which must be equal, and finally delete the tree.

Input file

An example of the input file (see `Project_1/tests/in.txt` at GitHub):

```
Jupiter lent 10.00 to Poseidon
Hermes borrowed 5.00 from Demeter
Poseidon borrowed 3.00 from Zeus
Poseidon lent 2.00 to Hermes
Hermes stole 1.50 from Zeus
Athena borrowed 5.00 from Jupiter
Demeter lent 10.00 to Apollo
Poseidon lent 15.00 to Athena
Jupiter borrowed 0.50 from Demeter
```

The number of lines can be any and is not specified in the file.

Expected output

The output must have three parts: messages about deity names being added to the tree, the ledger with alphabetically-sorted names as well as non-empty lists of their debtors and creditors, and two net balances for debits and credits at the end. See `Project_1/tests/out.txt` example at GitHub:

```
Jupiter has been added.
Poseidon has been added.
Hermes has been added.
Demeter has been added.
Zeus has been added.
Wrong transaction 'stole'.
Athena has been added.
Apollo has been added.
```

```
Apollo:
    credit
        Demeter 10.00
Athena:
    credit
        Jupiter 5.00
        Poseidon 15.00
```

```
Demeter:
    debit
        Apollo 10.00
        Hermes 5.00
        Jupiter .50
Hermes:
    credit
        Demeter 5.00
        Poseidon 2.00
```

Expected output

```
Jupiter:
    debit
        Athena 5.00
        Poseidon 10.00
    credit
        Demeter .50
Poseidon:
    debit
        Athena 15.00
        Hermes 2.00
    credit
        Jupiter 10.00
        Zeus 3.00
Zeus:
    debit
        Poseidon 3.00

Net debit:  50.50
Net credit: 50.50
```

Input line format

The input line format is

`<Deity1> <transaction> <amount> <preposition> <Deity2>`

Deity1 and *Deity2* are one-word names of deities from classical mythology and can be of any length. *Transaction* can be any verb in the past tense but only *lent* and *borrowed* are legal. *Amount* is some positive quantity of money always expressed as a floating-point number with two figures after the decimal point. *Preposition* is always either *from* or *to*. Two examples of valid input lines are

```
Chronos borrowed 3.14 from Nyx
Eos lent 2.71 to Styx
```

Double-entry bookkeeping

We emulate a simple version of double-entry bookkeeping, reduced to the case of money transfers between personal accounts.

Each transaction between two deities needs two mutual records in the ledger. This British system has the following mnemonics: *debit the receiver, credit the giver*.

If one deity borrows from the other one, he must record in his credit list the second deity with the amount borrowed, and the second deity must record in her debit list the first deity with the same amount lent.

Conversely, if one deity lends to the other one, then she must record the second deity in her debit list with the amount lent, and the second deity must record the first deity in his credit list with the same amount borrowed.

This technique creates a simple checksum: if every transaction has been recorded correctly in both lists, then the net debit and net credit must always be in balance.

Transactions

For example, one transaction

```
A borrowed X.XX from B
```

is equivalent to two mutual records as if

```
A owes  X.XX to B
B loans X.XX to A
```

which must be recorded in the ledger as

```
A:
    credit
        B X.XX
B:
    debit
        A X.XX
```

The converse transaction

```
A lent X.XX to B
```

generates the opposite order of records in the ledger,

```
A:
    debit
        B X.XX
B:
    credit
        A X.XX
```

Binary search tree

For simplicity, let’s ignore transactions between the deities so far. If you use names from the example input file, then the following binary search tree must be constructed (only names are shown in the nodes):

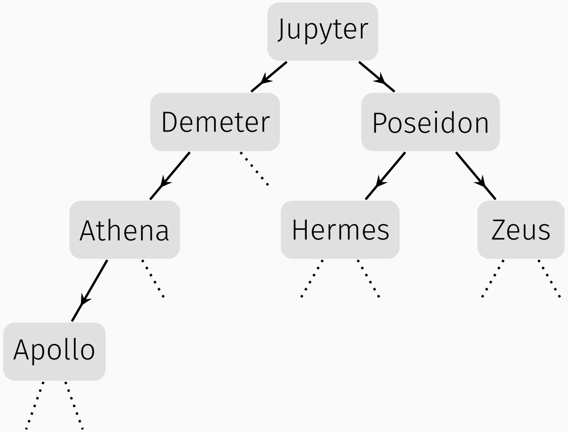

Figure 1: Binary search tree of nodes with only deity *names* and *left* and *right* pointers shown. Associated pointers are solid black edges, while disassociated (null) pointers are hanging dotted edges.

Binary search tree

Each node of the tree must contain five components:

1. the deity *name* (character string);
2. the list of *debit* (pointer to a list item);
3. the list of *credit* (same type as debit);
4. the *left* neighbor (pointer to a tree node);
5. the *right* neighbor (pointer to a tree node).

Once you read one input line, you have two names for the first and second deities. Each of these two names must be inserted into the tree. If there is no such name in the tree, then you must create a new tree node for it. Otherwise don't create it as the name is already there.

Once the tree (including two lists in each node) is built, then do an in-order walk to print for each node its name and the two lists. At the end, recursively walk the tree again to calculate the net debit and credit.

Lists of debit and credit

When both names are inserted into the tree and you have access to their nodes, you must correctly update corresponding lists of debit and credit. It is better if you use the simplest item type and subprograms that allow you to do sorted insertion without mistakes. Each item must have three components:

1) Either the *name* (character string) or the corresponding *deity* pointer to a tree node;
2) Floating-point *amount* either borrowed (for credits), or lent (for debits);
3) A pointer to the *next* item in the list.

You can implement the deity component in the item type as a character name instead of a pointer to a tree node. This is easier to code but it takes more memory per one list item. This solution will be marked less than the one that uses the tree node pointer (which is more difficult but it needs less memory).

Tree node structure

This illustrates the details of one tree node for Poseidon (see `out.txt`), built from the default input file `in.txt`.

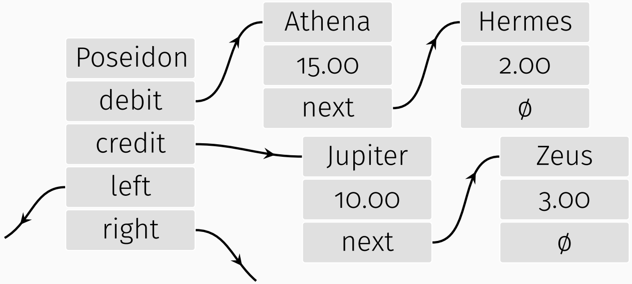

Figure 2: The structure of one tree node with the debit and credit lists shown in detail. The *next* pointer component is disassociated (∅) at the end of each list. For simplicity, list items show names instead of pointers to tree nodes.

Tree node and list entry types

An example of possible type definitions:

```fortran
type :: a_list_item
    type(a_tree_node), pointer :: deity
    real                       :: amount
    type(a_list_item), pointer :: next
end type a_list_item

type :: a_tree_node
    character(len=:),  allocatable :: name
    type(a_list_item), pointer     :: debit
    type(a_list_item), pointer     :: credit
    type(a_tree_node), pointer     :: left
    type(a_tree_node), pointer     :: right
end type a_tree_node
```

What the program should do?

- Start from an empty tree.
- Read line-by-line the input list. Parse each line in two names of deities, a transaction between them, and amount.
- If the transaction is illegal, skip this line and print a warning.
- Insert each of the two names into the tree. If for the given name the corresponding node doesn't exist, then create it and print a message that it has been added. If this node exists, don't create it again.
- For each of the two tree nodes store the amount in the two mutual lists of debit and credit. If the transaction is *borrowed*, then take name-1 node and insert the amount for name-2 into the credit list, then take name-2 node and insert the same amount for name-1 into the debit list. If the transaction is *lent*, then do the opposite. If the corresponding item already exists in any of the lists, then update it by adding the new amount to the old one.

What the program should do?

- Once all lines are parsed and the tree is built, print the ledger by walking round the tree *in-order*. For each tree node, print the name and, if they are not empty, print the two lists of debit and credit. If you did the insertion correctly both for tree nodes and for list items, then all names will appear sorted.
- Walk round the tree two times more, calculate the net amount for all debits, for all credits, and print them. If no mistake has been made, the net amounts must be equal.
- At the end, delete all tree nodes, list items, and other dynamic targets so that no memory leaks are visible in Valgrind.

Tips that might help

Don’t solve everything in one run, follow the recommended steps:

1) read input lines,
2) parse them into components,
3) declare the tree node type with no internal lists so far,
4) write the tree insertion procedure,
5) from each input line insert the two names into the tree,
6) write the tree printing procedure,
7) print the tree,
8) write the tree destruction procedure,
9) destroy the tree,
10) make sure there are no memory leaks,
11) declare the list item type,
12) update the tree node type to have two lists,

Tips that might help

13) write the sorted list insertion procedure,
14) depending on the transaction between the two names, insert the amount into the two mutual lists of the corresponding tree nodes,
15) write the list printing procedure,
16) update the tree printing procedure to print both non-empty lists,
17) print the tree in full ledger form,
18) write the list destruction procedure,
19) update the tree destruction procedure to destroy the two lists from each node,
20) destroy the entire tree with lists,
21) make sure no memory leaks are visible in Valgrind,
22) write the list reduction procedure,
23) write the tree reduction procedure, which sums either all debit or credit lists,
24) print the net balances,
25) finally destroy the whole tree.

Evaluation

The project is not all-or-nothing: your mark will depend on how far you get. You can submit partially-working code with comments explaining what you tried to do and what you would like to do but don't know how to.

Keep copies of your program at different steps of the solution. You can submit at least something that works in the case when the next step becomes difficult and you cannot solve it.

This project's mark contributes 1/3 to the total mark of the course. To pass, you need to get at least a 50% mark. You get a 100% mark if your code does everything what is expected, producing for several different input files exactly the same output files, and if there are no memory leaks.

This project is individual. You can discuss general ideas and technical details between each other, but the code you submit you must write yourself. When in doubt, I'll ask you to explain your solution.

Breakdown of marks

10%: for reading all input lines and correctly parsing them into components;

35%: for inserting nodes into the tree, messaging on creation, and treating existing ones (10%), for inserting items into sorted lists and updating amounts in existing ones (20%), for inserting into the correct pair of lists based on the transaction (5%);

15%: for printing the ledger by printing the tree (10%) with all lists of debtors and creditors (5%);

15%: for printing the net balances of debit and credit by summing each corresponding list (5%) round the tree (10%);

15%: for destroying the tree (10%) and lists (5%), — this mark will be lowered if any memory leaks are visible in Valgrind;

5%: for using correct format descriptors on output, — this mark will be 0% if the output format is loose and not exactly the same as in the provided examples;

5%: for using a tree-node pointer instead of a character string in the list-item type.