A2 Design Document

Gabriel Racz 101181470

Message Consumer Program (MCP)

The Message Consumer Program is a program that parses the input delivered by an external source and stores it in a more usable data structure. In this case, a message payload is delivered representing customer transaction data. The program stores this in an array, calculates total number of transactions as well as average transaction value.

The main goals of this implementation is to store the transaction data in an efficient and effective way that allows for minimum memory usage while maintaining the ability to efficiently query and manipulate the data set.

Implementation

To store the necessary components of a single transaction, a transaction structure is defined to store a 10 character transaction ID, an 8 character customer ID, and a float amount value. These transactions are stored in a dynamically sized array that grows to accommodate the exact amount of transactions in the message payload. After the array is finalized, calculating the average (along with any other data manipulation / calculations) is trivial.

The array that stores the transaction (tList) is an array of transaction pointers to ensure that both the array and the transactions can be stored on the heap. A pointer to this array will be passed to the processInputString function along with the address of a count variable. This function will parse the input string and return with an error code representing the success or failure of the parsing.

To read input of arbitrary length, the implementation relies on the required format of the input string. It reads in information from the stdin input buffer in chunks large enough for each of the transaction fields. While the next char in the input buffer is a ‘|’ representing a new transaction, the loop will continue. Each iteration, tList is reallocated with enough space to fit another transaction pointer. The new transaction pointer is then initialized on the heap with a call to malloc. Each of the required fields is then read in from the input buffer with calls to fgets() supplying the appropriate length for each field. Once complete, the new transaction is appended to the end of tList. As long as the input was in the right format the next character is either another ‘|’ indicating a new transaction, or an ‘\n’ representing the end of the input.

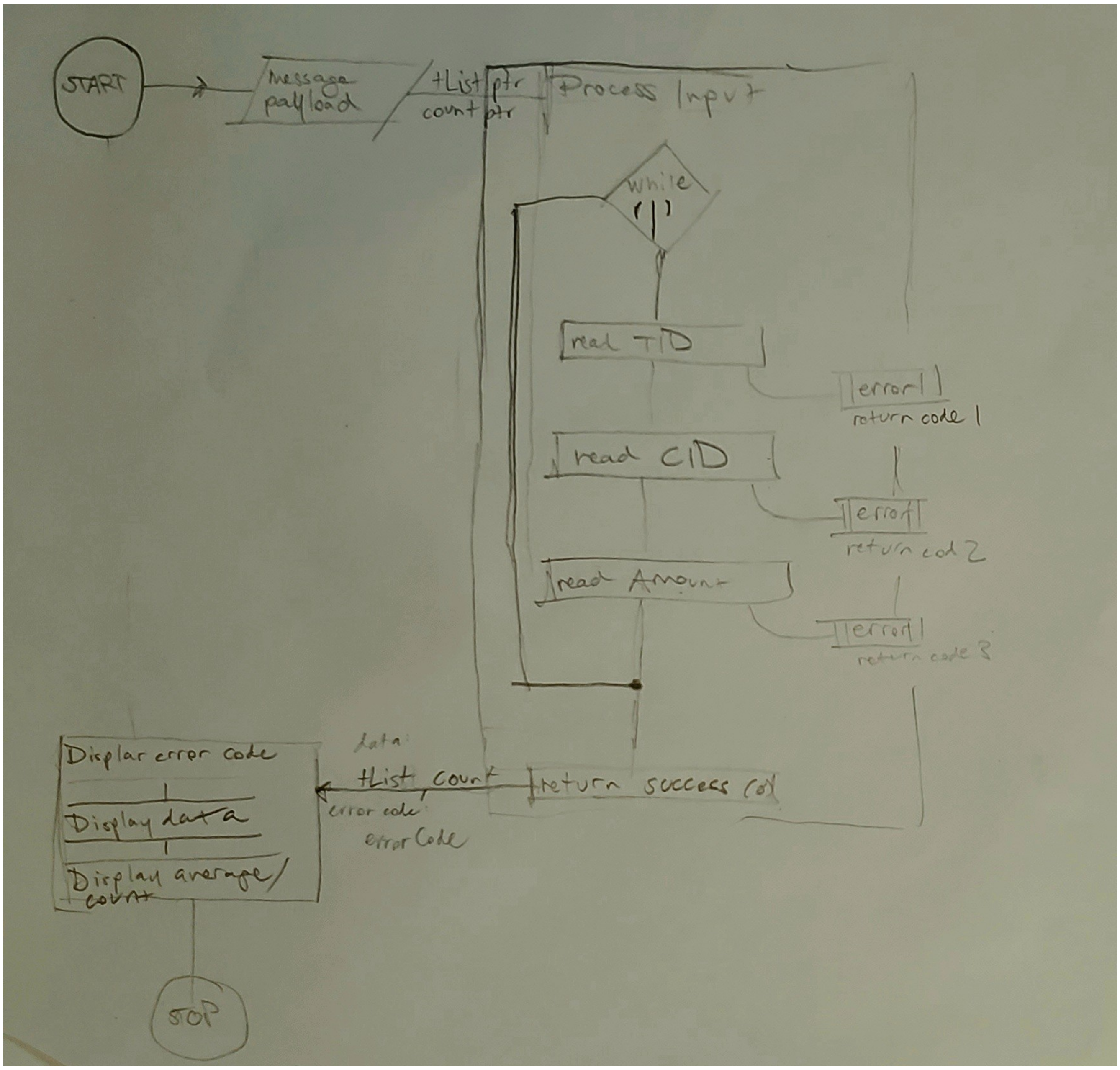
In theory this will repeat for an arbitrary large input, however, I encountered a limitation within the terminal caps any input put into the terminal to 4096 bytes. To get around this I used the command stty -icanon which puts the terminal in non-canonical mode and allows any input to be read. A better implementation would read the input from a file instead of command line entry.

Other Decisions

Because the transactions are stored one by one, if there is an error in the input of a transaction, the previous transactions can still be accessed. Also, checks are made for each transaction field so an error message can display the exact exact transaction and field that was inputted incorrectly.

A dynamic array is far better than a linked list for this implementation due to the memory saved by avoiding the Node types linking the transactions together, as well as the usability of having the transactions in an array rather than being linked together. With the array implementation, further data manipulation such as sorting, and searching, and random access are much more efficient and intuitive.

The growth factor of the dynamic array can be optimized if more detail was given about the various use cases for the program. At the moment, the program uses exactly the correct amount of memory to store every transaction because realloc is called on every iteration of the loop. This was done intentionally because the specification outlined the conservation of memory to be more important that any other aspect. Other dynamic arrays typically grow by factors of around 2 to reduce unnecessary calls to realloc especially on modern machines.


START
message payload
tList ptr
count ptr
Process Input
while (1)
read TID
error
return code 1
read CID
error
return cod 2
read Amount
error
return code 3
return success cod
data:
tList, count
error code
errorCode
Display error code
Display data
Display average/count
STOP