Politechnika Śląska
Wydział Informatyki, Elektroniki i Informatyki

Fundamentals of Computer Programming

«Bank Application»

author: Oliwia Mlonek

instructor: mgr inż. Krzysztof Pasterak

year: 2019/2020

lab group: Monday, 15:45 - 17:15

deadline: 2019-02-03

1 Project’s topic

My idea for this project was to create a program visualizing the operation of a real mobile banking application (available for example on a smartphone) using dynamic data structures. The user has the option to create, edit and overview their accounts in the app and perform transaction such as transfers.

2 Analysis of the task

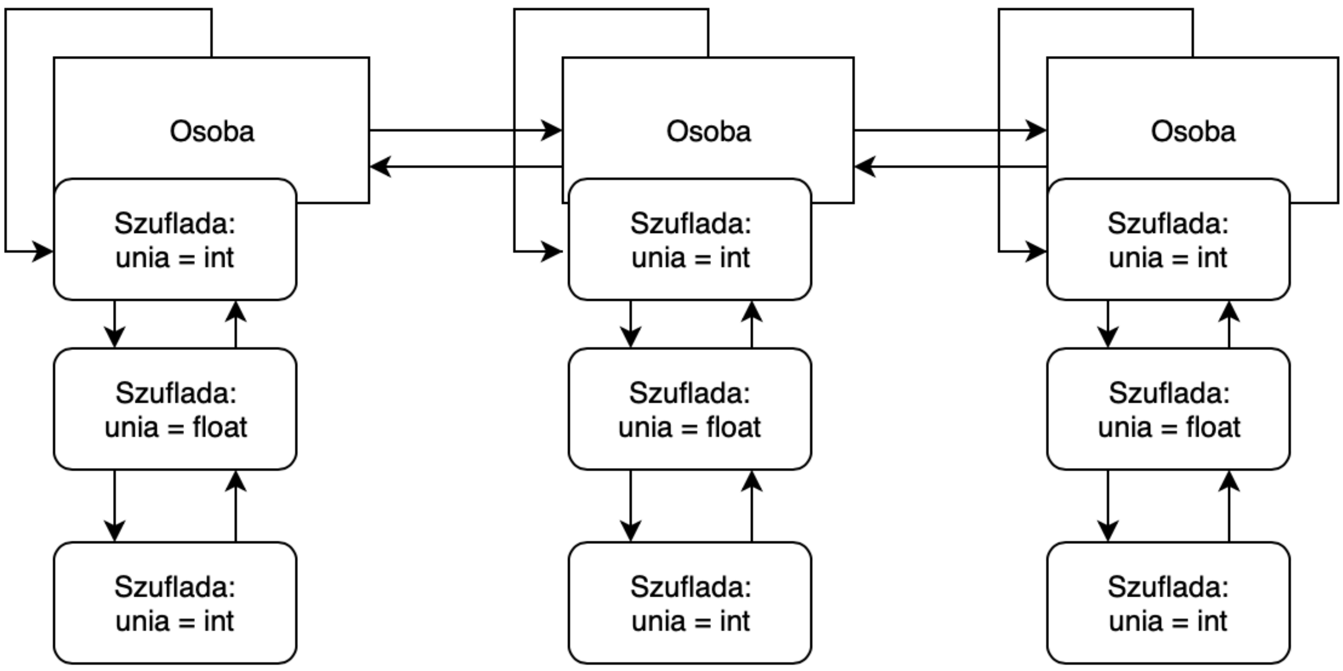
The task focuses on storing data in the doubled-linked list of list. Mostly lists are created by function that reads data from the database text file 'Dane' and creates a list of lists based on it.

2.1 Data structures

The structure used in the list is a doubled-linked list of lists. The first "person" list stores personal data that can be stored in string variables - name, password and account type. The main problem to solve was the idea of creating second doubled-linked list, for each "person" node, which would store numeric data variables of different types (integer and float) in separate nodes.

This second list was named "szuflada" and its functionality is based on a different data structure - union. This union is contained in every "szuflada" node. They are similar to structures, however only one component of the union can be used at a time. That allows to create individual nodes, each of which can have one type of variable from the union.

The figure below presents the scheme of this list of lists:



2.2 Algorithms

The program sorts list of lists (in ascending order) using "Bubble Sort" based on the account number stored inside the "szuflada" list in every "osoba" node. However, program doesn't use the arrays as in the basic example implementation of this algorithm. We treat our nodes as array cells, and because in my version of this algorithm we have the exact number of nodes in the list, we can easily get to, for example, 6th 'cell' using a simple while loop. Average complexity of the "Bubble Sort": n^2.

The implementation in the program:

```
void bubbleSort(osoba* start)
{
   int swapped;
   struct osoba *ptr1;
   struct osoba *lptr = NULL;
   if (start == NULL)
      return ;
   do
   {
      swapped = 0;
      ptr1 = start;
      while (ptr1->next != lptr)
      {
         if (ptr1->info->type == 1 && ptr1->info->d.some_int > ptr1->next->info-
            >d.some_int)
         {
            swapNode(&start, ptr1->info->d.some_int, ptr1->next->info->d.some_int);
            swapped = 1;
            ptr1 = ptr1->prev;
         }
         ptr1 = ptr1->next;
      }
      lptr = ptr1;
   }

   while (swapped);
   return;

};
```

Another algorithm used in the program, after sorting, is “Binary search algorithm”. This search algorithm finds the position of a target value within a sorted array, next compares the target value to the middle element of the array. If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this until the target value is found. If the search ends with the remaining half being empty, the target is not in the array. Average complexity of the “Binary search algorithm”: $O(logn)$.

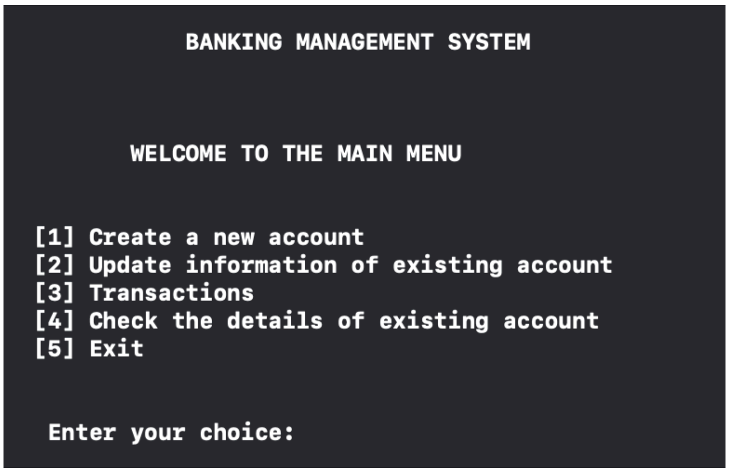
Again, since I want to make all operations using lists, instead of creating an array with the data, I modified the algorithms so it treats the nodes of the previously sorted list of lists as the array cells.

The implementation:

```
int existing = 0;
  while( existing == 0){

    int liczba, l, p, s, num_found;

    l = 0;
    p = count-1;
    osoba* find = Check;
    cout << "Enter your account number: ";
    cin >> liczba;

    while (true)
    {

      if (l > p)
      {
        break;
      }

      s = (l+p)/2;

      for(int x = 0; x < s; x++)
      {

        if(find->info->type == 1){
          num_found = find->info->d.some_int;
          if (num_found == liczba)
          {
            existing = 1;
            break;
          }
        }
        find = find->next;
      }

      if (num_found == liczba)
      {
        break;
      }
      else if (num_found < liczba)
        l = s+1;
      else
        p = s-1;
    }

    if(existing == 0)
      cout<< "No such account. “<<endl;
}
```

3 External specification

To set up an account and use the app one must first log in with the password received at the stationary bank facility. In the project, these passwords are kept in the file ‘password.txt’. Only then user can create an account in the app and set his own password to protect that account. Thanks to this function, it is possible to make transfers safely, change personal data, e.t.c. The application always asks for a password before performing any operation. The application is controlled via the main screen menu (which redirects to the selected option) as illustrated in the photo below:



```
             BANKING MANAGEMENT SYSTEM



         WELCOME TO THE MAIN MENU


[1] Create a new account
[2] Update information of existing account
[3] Transactions
[4] Check the details of existing account
[5] Exit


 Enter your choice:
```

The user using the application is repeatedly asked to enter different types of data such as account number or password. To avoid an error, the correctness of typed numeric variables (int, float) is checked. When an incorrect character (letter or symbol) appears at the beginning or in the middle of the variable, the program notifies of incorrect input data and asks to enter it again or, in some cases, allows to return to the main menu. In addition, the program notifies you of any successful operation and also alerts you in the event of errors and incorrect input.

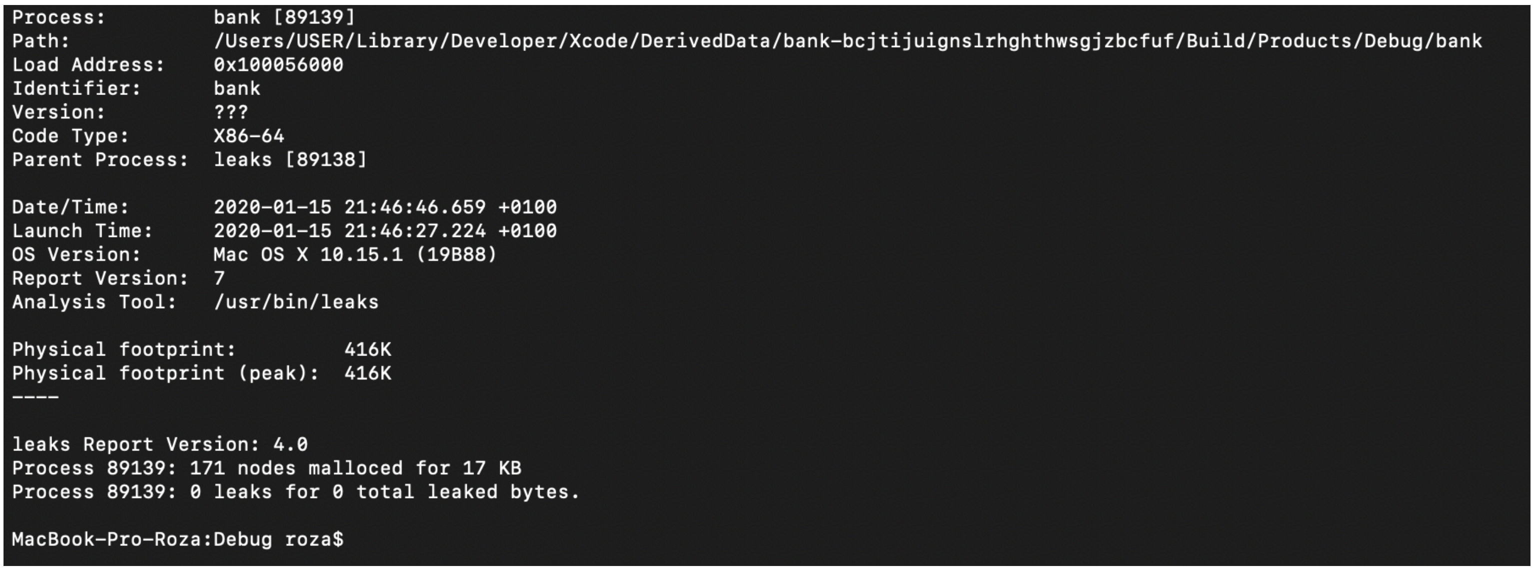
4 Internal specification

Technical documentation in the doxygen style is attached to this report in a separate file (refman.pdf).

5 Testing

The program has been tested with various types of files. Incorrect files (numbers in incorrect format) are detected and an error message is printed. The program was tested with many different “potential clients” who had different account states and types. If any wrong input had been typed, mistake had been found or operation could not had been executed, the proper message was printed.

The program was tested on the macOS 10.15.1 operating system and was also checked for memory leakage using memory leak detecting utility called “Leaks” and no leak was detected :



```
Process:         bank [89139]
Path:            /Users/USER/Library/Developer/Xcode/DerivedData/bank-bcjtijuignslrhghthwsgjzbcfuf/Build/Products/Debug/bank
Load Address:    0x100056000
Identifier:      bank
Version:         ???
Code Type:       X86-64
Parent Process:  leaks [89138]

Date/Time:       2020-01-15 21:46:46.659 +0100
Launch Time:     2020-01-15 21:46:27.224 +0100
OS Version:      Mac OS X 10.15.1 (19B88)
Report Version:  7
Analysis Tool:   /usr/bin/leaks

Physical footprint:         416K
Physical footprint (peak):  416K
----

leaks Report Version: 4.0
Process 89139: 171 nodes malloced for 17 KB
Process 89139: 0 leaks for 0 total leaked bytes.

MacBook-Pro-Roza:Debug roza$
```

6 Conclusions

The application “pretends” to be real mobile application and offers many possibilities available in real bank applications. The creation required prediction of many potential scenarios (eg mistakes in user input, empty data file), frequent work on files, data protection and comparison of many datasets. The most challenging tasks are: manual memory management without any memory leaks, navigating through lists of lists without losing the pointer to the first item and implementing both algorithms without using arrays.