Sabancı University

Faculty of Engineering and Natural Sciences
CS204 Advanced Programming
Fall 2021

Homework 4 – Stacks&queues and SU services

Due: 26/11/2021, 07:00 am

PLEASE NOTE:

Your program should be a robust one such that you have to consider all relevant user mistakes and extreme cases; you are expected to take actions accordingly!

You can NOT collaborate with your friends and discuss solutions. You have to write down the code on your own. Plagiarism will not be tolerated!

1. Introduction

This homework's aim is to make you familiar with the logic behind **queues** and **stacks** (and a bit of **simply linked lists** of linked list, again) as well as **recursions** by practically using them in a real-case scenario (you'd probably encounter many similar ones as a future programmer). Namely, SU offers a certain number of services to its academic stuff and students (such as printing, supercomputer data processing, …), implemented as functions. You will keep those SU offered services in a **data structure of linked lists**. Service requests of the students and the academic stuff (instructors) are kept separately and served in a First In First Out (FIFO) manner, thus we need **two separate queues** for both of them. Each service (function) request is consisted of set of instructions (commands) described in the next section. A certain service (function) can be part of (called from) another function (e.g. print the output data after processed from the supercomputer). As it is common in modern day systems, for this reason we will need **a single, commonly shared stack** to put on and serve in First In Last Out (FILO) manner those subsequent service (function) calls. All of this is illustrated in fig.1. Also, in order to process those consecutive function calls, you will be asked to implement **a recursive function** that calls an instance of itself at any time a new function call is issued. This will be more clarified in the 'main menu & program guide' section.

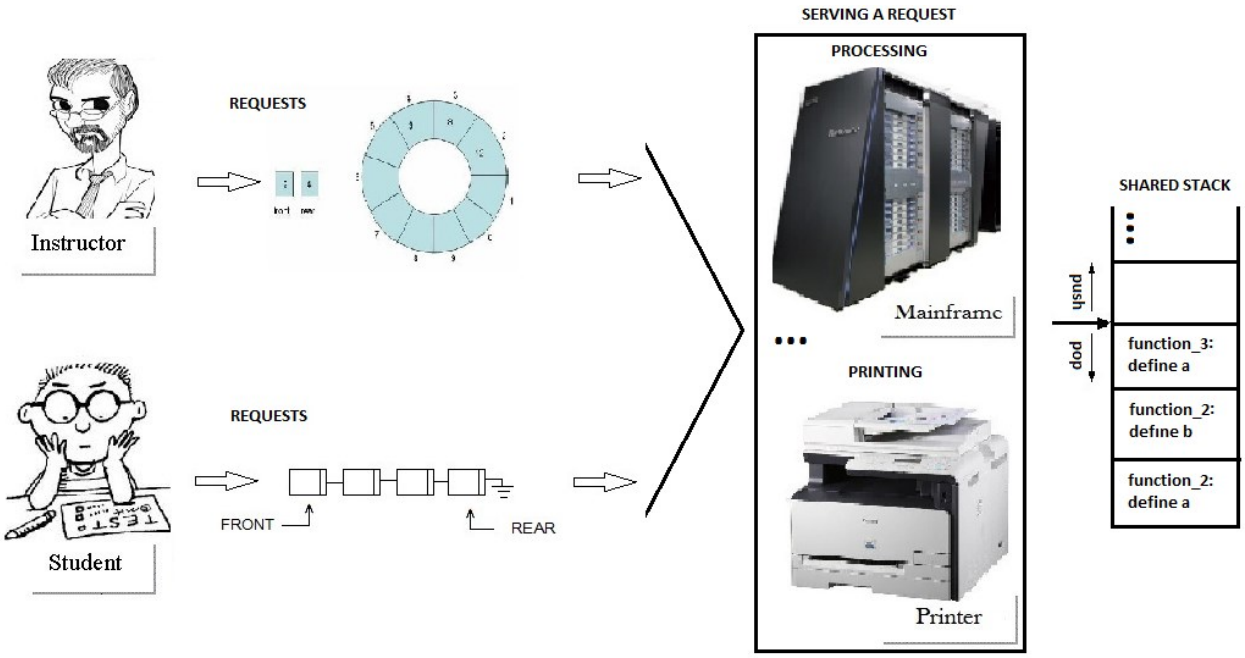


Fig.1. System overview

2. Service offers (input files)

For the time being, SU has organized its services that it offers to its academic stuff and students in separate files (fig. 2). Those will be your input files. Each service is implemented as a function that has a certain number of commands (instructions). Their aim and purpose will be described in the next section. In the first line function's name is given proceeding with a colon (':'). Afterwards in each new line you have a separate command that ends with semicolon (';'). The input format is always the same. You don't have to care about multiple empty spaces or new lines, and even letter cases.

```
function_1:
print stack;
define x;
define y;
print stack;
call function_2;
…
…
…
define b;
print stack;
```

input1.txt

```
function_2:
print stack;
define a;
define b;
print stack;
call function_3;
…
…
…
define c;
print stack;
```

input2.txt

```
function_3:
print stack;
define a;
define x;
define z;
print stack;
define c;
print stack;
```

input3.txt

Fig.2. Input file samples

For all of the read input files, your job will be to construct a simply linked list of the commands. Eventually, after reading all of your input files (in our case 3 input files), you should finish with a structure shown in fig.3. Each service (function) appears as a node in the list that has its name in it, a pointer to the next service node and another point to the commands it is consisted of. As system engineers, we decided on this approach (dynamic lists) since the number of services that SU offers (and may offer in the future) is not known in advance and that each service (function) has different numbers of commands. This way of dealing with things, not only will utilize our memory, but should also help us while processing the services when using a recursive function. We will come back to this after a while.

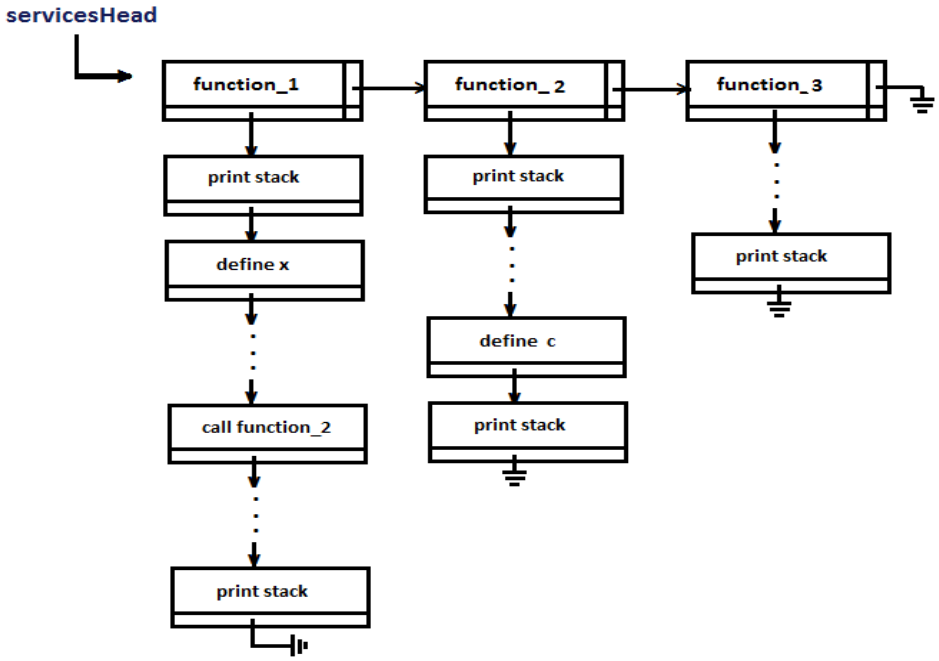


Fig.3. Linked list of Offered services (functions).

3. Logic flow of the system

Students and instructors put their corresponding requests of services in two separate queues. Since services are done in FIFO logic, you should keep in mind the order of the service requests (which request came first, e.g. the front and the rear of both of the queues). Service request are processed one at a time. Knowing that the number of instructors is known in advance we encourage you to implement instructors queue as a vector (preferable a circular one as shown in fig.1). Since the number of students is not known beforehand, we encourage you to implement students queue as a dynamic linked list (fig.1). Of course, the final decision regarding the implementations is left to you. Furthermore, instructors have priority over students. This means that even if there is only one instructor request, it will be processed before any of student's requests, even if a student request came first. Only when there are not any instructor requests (e.g. instructors queue is empty) we can proceed with serving students requests. Of course, inside a queue (be it the student's or instructor's one), serving is done in FIFO manner (fig. 1).

On the other hand, in order not to complicate things, we have only three types of commands for the services (functions):

1. **define**,
2. **print stack**, and
3. **call.**

The "**define**" command simply defines a variable, whose name comes immediately after the "define" command (e.g. define a, define x, define z). When reaching this command, we should put on the top of the shared stack (push) the defined variable.

The "**print stack**" command, in a way it will be illustrated below, prints the content of the commonly shared stack. When printing the stack content (also known as stack trace) we should also show the corresponding function that defined a certain variable. E.g. if variable *x* was defined by a "define x" command from function_1, on top of the stack we should put (and subsequently print)

something like: “function_1: define a”. During the ”print stack” command nothing should be put on top of (push-ed) or taken (pop-ed) from the shared stack (e.g. stack should not be changed).

The “**call**” command should enable us to call an existing service (function) and temporarily stop the execution of the current function while proceeding with the execution of the newly called one. For this reason we will need a structure to be available to those services (functions) that will work in a First In Last Out (FILO) order, e.g. if a function calls a function that calls a function, firstly we will need to execute the lastly called function, then the one that called it and in the end the first function. When the called function is finished, we should be able to continue with the original function from the exact place we left, thus with the instruction immediately after the call command. As we said before, we will do this by using a commonly-shared stack for the system. Furthermore, when a function finishes, all of the stack data resulting (pushed) from it, should be deleted (removed, pop-ed) from the stack. Take caution not to delete any data from the predecessor function (the one that called the current, about to finish function).

4. Main menu & program guides

Fig.4 gives to you the main menu piece of code. You should implement the corresponding functions. Fig.5 shows the main menu view (output) that will occur often during the course of the program.

```
while (true){
        cout << endl;
        cout<<"*********************************************************************"<<endl
            <<"*************** 0 - EXIT PROGRAM                        *************"<<endl
            <<"*************** 1 - ADD AN INSTRUCTOR SERVICE REQUEST   *************"<<endl
            <<"*************** 2 - ADD A STUDENT SERVICE REQUEST       *************"<<endl
            <<"*************** 3 - SERVE (PROCESS) A REQUEST           *************"<<endl
            <<"*********************************************************************"<<endl;
                cout << endl;
                int option;
                cout << "Pick an option from above (int number from 0 to 3): ";
                cin>>option;
                switch (option)
                {
                        case 0:
                                cout<<"PROGRAM EXITING ... "<<endl;
                                system("pause");
                                exit(0);
                        case 1:
                                addInstructorRequest();
                                break;
                        case 2:
                                addStudentRequest();
                                break;
                        case 3:
                                processARequest();
                                break;
                        default:
                                cout<<"INVALID OPTION!!! Try again"<<endl;
                }//switch
        }//while (true)
```

Fig.4. Main menu code.

```
*********************************************************************
*************** 0 - EXIT PROGRAM                        *************
*************** 1 - ADD AN INSTRUCTOR SERVICE REQUEST   *************
*************** 2 - ADD A STUDENT SERVICE REQUEST       *************
*************** 3 - SERVE (PROCESS) A REQUEST           *************
*********************************************************************

Pick an option from above (int number from 0 to 3): _
```

Fig 5. Main menu output.

The first main menu option is rather trivial and already implemented. We will discuss the others in the subsequent sections.

Fig.6 gives a brief overview of the recursive nature of the processARequest(string functionName) function called from the processARequest() function of the main menu. Of course, your job here is to recursively implement this function for SU system.

```
void processARequest(string functionName)
{
        //…
        while (/* there are commands to process*/)
        {
                //…
                if(/*current command*/ =="define")
                {
                        //process the define command
                }
                else if (/*current command*/=="call")
                {
                        // …
                        // the recursion goes here
                        processARequest(/*with the called functionName*/);
                }
                else
                        //print the stack …

        }// while
        //…
        // delete this function’s data from the top of the stack
        cout<<functionName<<" is finished. Clearing the stack from it's data... "
                        <<endl;
        system("pause");
}
void processARequest()
{
        if (!instructorsQueue.isEmpty())
        {
                //…
                //if instructors queue is not empty, process the next request
                processARequest(functionName);
                cout<<"GOING BACK TO MAIN MENU"<<endl;
        }
        else if (!studentsQueue.isEmpty())
        {
                //…
                //if instructors queue is empty and student’s not,
                //then process the next student request
                processARequest(functionName);
                cout<<"GOING BACK TO MAIN MENU"<<endl;

        }
        else
        {
                 // otherwise…
                cout<<"Both instructor's and student's queue is empty.\nNo request is
                processed."<<endl<<"GOING BACK TO MAIN MENU"<<endl;
        }
}
```

Fig6. Recursive nature of processARequest(string functionName)

Albeit not compulsory, yet we strongly encourage you to write separate (.h) headers and corresponding (.cpp) implementations of queue and stack that will serve the purposes of this homework and will be used in the main program. Since our aim in this homework is not to bother you with letter cases either, you can assume that they are case sensitive (thus function_3 is different from FuNctiOn_3).

This is true for the remaining of the homework, at any instance (e.g. artrim and Artrim as a student name are different).

5. Reading input files

If any of the input file fails to open (file doesn't exist, hardware problem, …) the program should terminate after displaying a proper message.

Otherwise the input files sequence should be given in the following manner (fig. 7)

```
If you want to open a service (function) defining file,
then press (Y/y) for 'yes', otherwise press any single key
y
Enter the input file name: input1.txt
Do you want to open another service defining file?
Press (Y/y) for 'yes', otherwise press anykey
Y
Enter the input file name: input2.txt
Do you want to open another service defining file?
Press (Y/y) for 'yes', otherwise press anykey
y
Enter the input file name: input3.txt
Do you want to open another service defining file?
Press (Y/y) for 'yes', otherwise press anykey
N
--------------------------------------------------------------------
PRINTIG AVAILABLE SERVICES (FUNCTIONS) TO BE CHOSEN FROM THE USERS
--------------------------------------------------------------------
```

Fig.7. Reading input files

When we are finished with the input files (and the construction of the linked lists of fig. 3), we should display all the services with their corresponding commands in the following manner (fig. 8).

Note!!!: You can assume there are no loops in the functions (e.g. a function that calls itself or a function_x that calls a function_y, which instead calls function_x again). We might also have more (or less) than three input files.

```
--------------------------------------------------------------------
PRINTIG AVAILABLE SERVICES (FUNCTIONS) TO BE CHOSEN FROM THE USERS
--------------------------------------------------------------------


function_1:
print stack, define x, define y, print stack, call function_2, print stack, defi
ne z, print stack, call function_3, print stack, define c, define b, print stack
, call function_2, print stack, define a, print stack.

function_2:
print stack, define a, define b, print stack, call function_3, print stack, defi
ne x, define y, define z, print stack, call function_3, print stack, define c, p
rint stack.

function_3:
print stack, define a, define x, define z, print stack, define c, print stack.
```

Fig.8. Displaying the read services (functions).

After this, we should be sent to the main menu.

6. Add an instructor service request

When choosing the 2nd option ('add an instructor service request'), firstly we are prompted for the name of the service (function) we are asking for. If the asked service doesn't exist, a suitable message should be printed and we are sent back to the main menu (fig. 9).

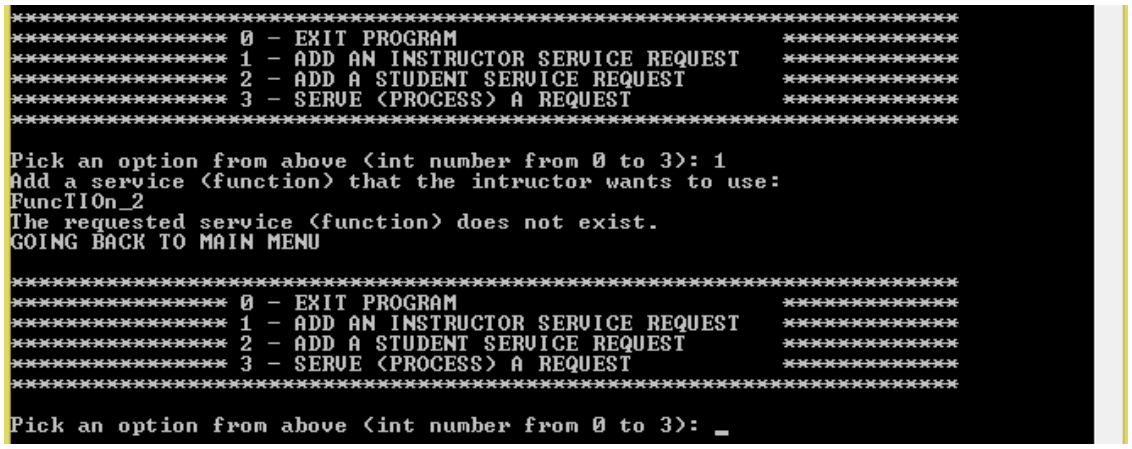


Fig. 9. Requesting unavailable service.

If the requested service exists, then we are prompted for the name and ID of the instructor. Subsequently, his/hers request is put in the instructors queue and a message is displayed and we are sent back to the main menu (fig.10)

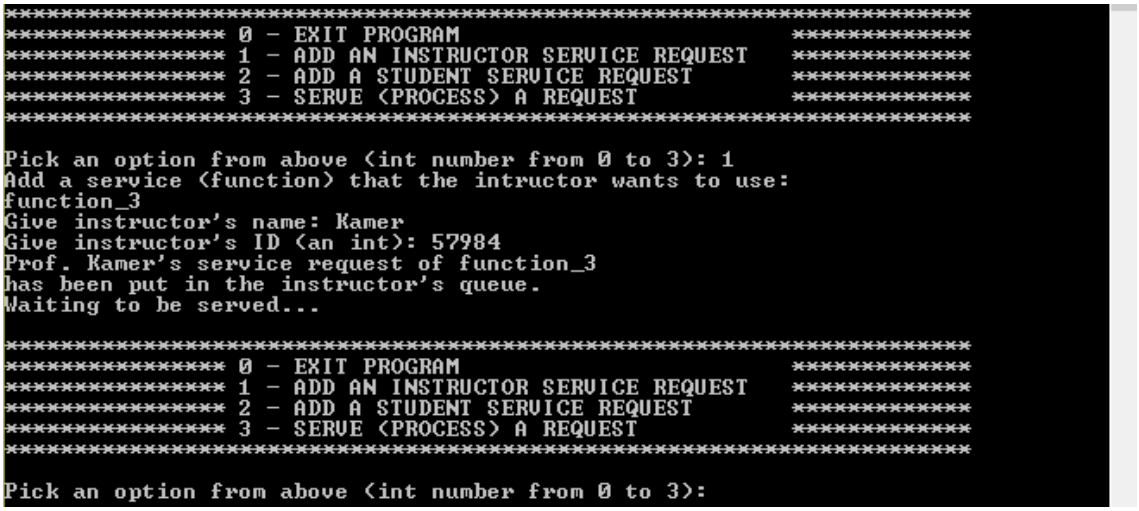


Fig. 10. Instructor's service request added to the instructors queue.

7. Add a student service request

When choosing the 3nd option ('add a student service request'), similarly as we did with the instructor, firstly we are prompted for the name of the service (function) we are asking for. If the asked service doesn't exist, a suitable message should be printed and we are sent back to the main menu (fig. 11).

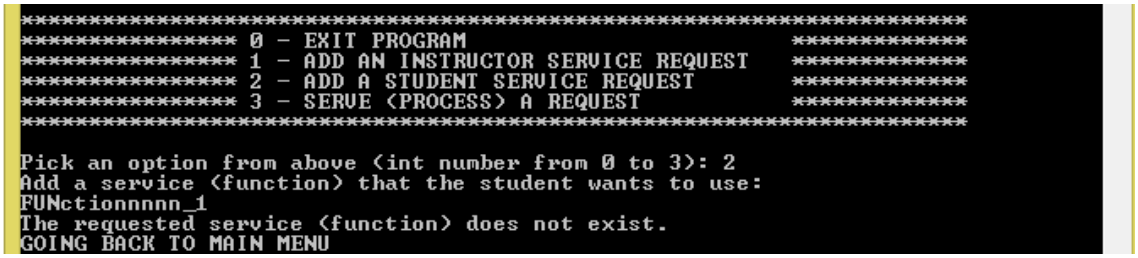


Fig. 12. Requesting unavailable service.

If the requested service exists, then we are prompted for the name and ID of the student. Subsequently, his/hers request is put in the instructors queue and a message is displayed and we are sent back to the main menu. In fig.12 we illustrate the addition of two student requests.

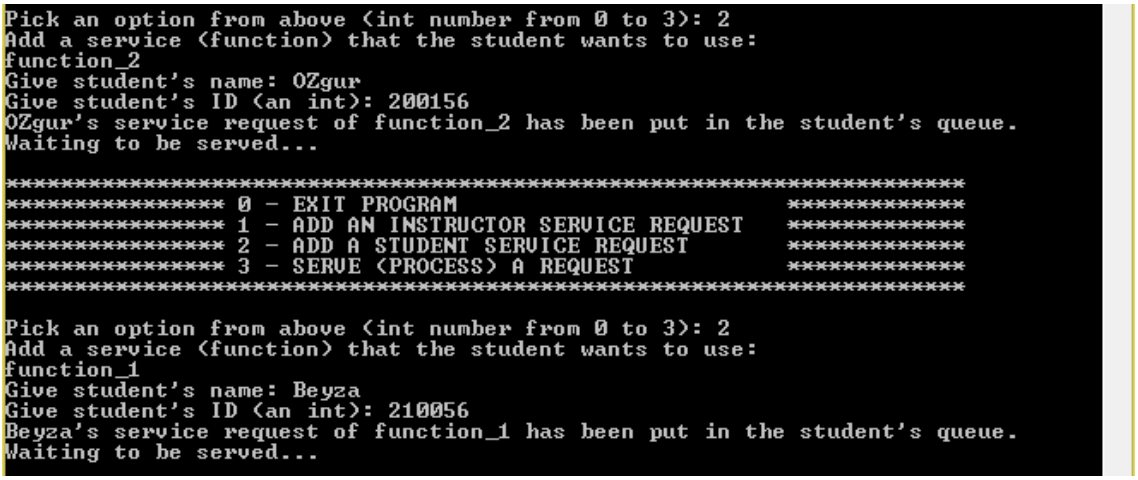


Fig. 12. Two student service requests added to the students queue.

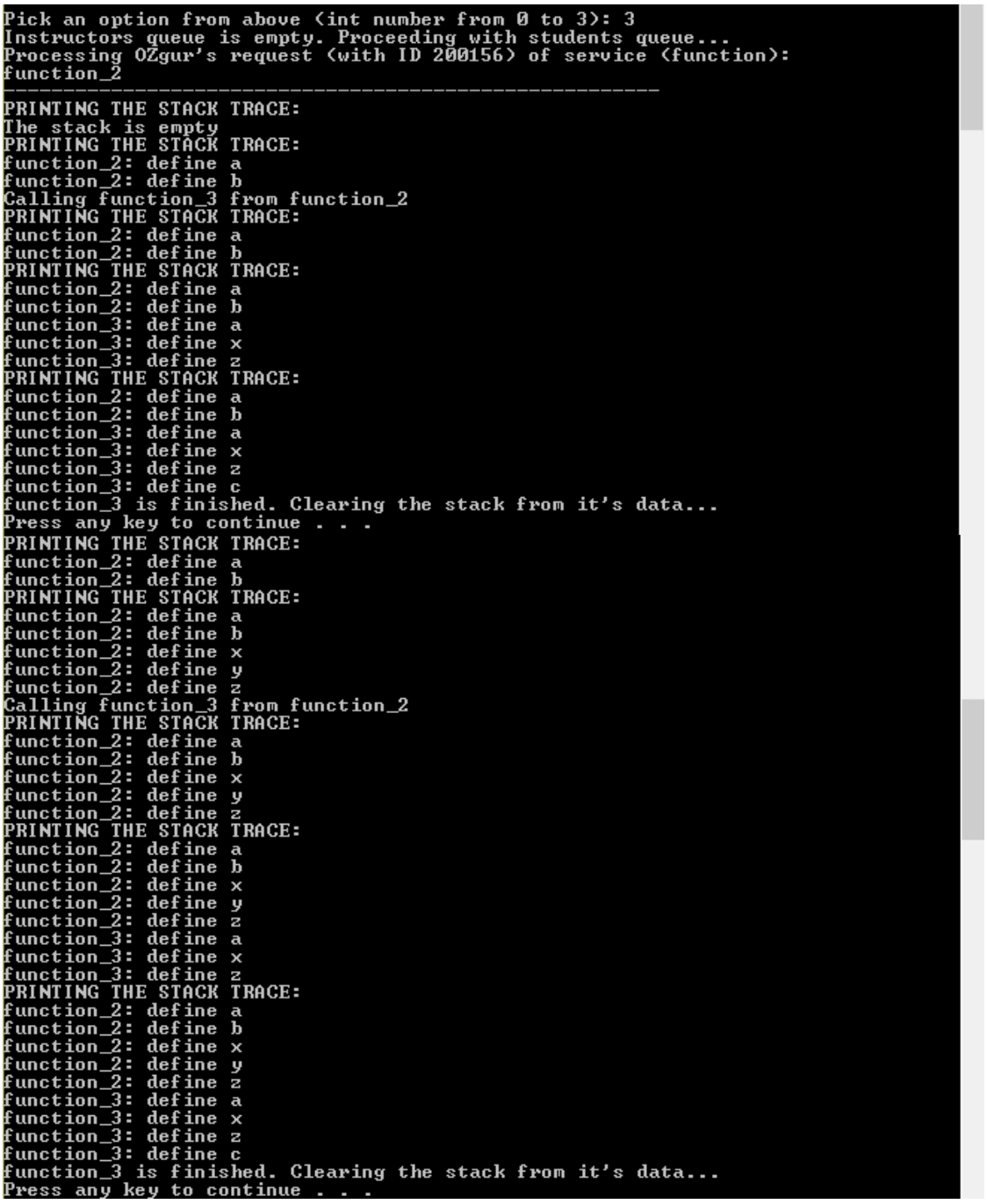
8. Serve (process) a request

When the 4th option is chosen, we serve a single request, starting with instructor's ones (if available). Whenever we encounter the 'define' command of a function, we put it in the shared stack, when we encounter the 'print stack' command we print the stack trace and when we encounter the 'call' function command, the corresponding function is called and processed recursively. You can assume that if there is a 'call' command inside a function, the called function exists in the linked list of offered services you build in the beginning. When a function is about to finish, you should pause the program for a while (use system("pause")) and clear the stack from its data (as we already described). Initially the stack is empty.

Fig. 13. Shows the output when processing an available instructor request.

Fig.13. Processing an instructor request of function_3.

If there is not any instructor request we proceed with a student one (of course, if available). Fig. 14 shows the processing of a student request (that we added in the previous section). **Note!!!**: have in mind that in our case function_2 calls function_3 twice, while function_1 calls both function_2 and function_1.



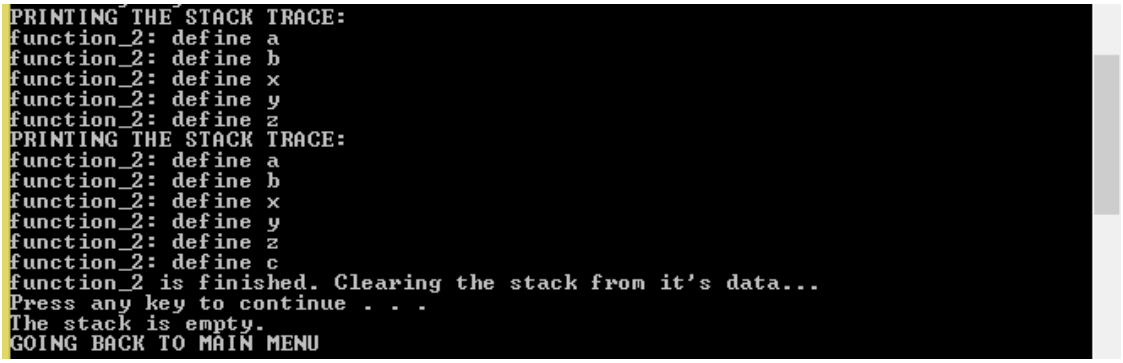


Fig. 14. Serving a student request

If both the instructors and students queues are empty, we have the following output (fig.15).

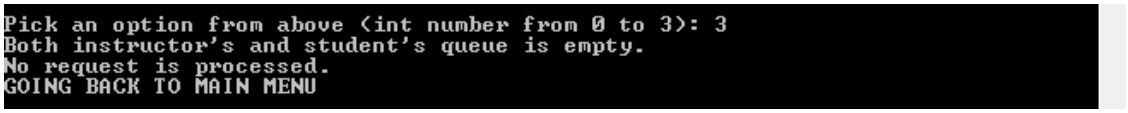


Fig. 15. No request served (empty instructor and student queues)

Some Important Rules

Although some of the information is given below, first, please read the homework submission and grading policies in the course webpage and lecture notes of the first week. In order to get a full credit, your programs must be efficient and well commented and indented. Presence of any redundant computation or bad indentation, or missing, irrelevant comments may decrease your grades if we detect them. You also have to use understandable identifier names, informative introduction and prompts. Modularity is also important; you have to use functions wherever needed and appropriate.

When we grade your homework we pay attention to these issues. Moreover, in order to observe the real performance of your codes, we are going to run your programs in *Release* mode and **we may test your programs with very large test cases**.

What and where to submit (PLEASE READ, IMPORTANT)

You should prepare (or at least test) your program using MS Visual Studio 2012 C++. We will use the standard C++ compiler and libraries of the abovementioned platform while testing your homework. You need to place your first and last name in the program (as a comment line of course).

Submissions guidelines are below. Some parts of the grading process are automatic. Students are expected to strictly follow these guidelines in order to have a smooth grading process. If you do not follow these guidelines, depending on the severity of the problem created during the grading process, 5 or more penalty points are to be deducted from the grade.

Name your cpp file that contains your program as follows:

"SUCourseUserName_YourLastname_YourName_HWnumber.cpp"

Your SUCourse user name is actually your SUNet user name which is used for checking sabanciuniv e-mails. Do NOT use any spaces, non-ASCII and Turkish characters in the file name. For example, if your SUCourse user name is cago, name is Çağlayan, and last name is Özbugsızkodyazaroğlu, then the file name must be :

Cago_Ozbugsizkodyazaroglu_Caglayan_hw2.cpp

Do not add any other character or phrase to the file name. Make sure that this file is the latest version of your homework program. Compress this cpp file using WINZIP or WINRAR programs. Please use "zip" compression. "rar" or another compression mechanism is NOT allowed. Our homework processing system works only with zip files. Therefore, make sure that the resulting compressed file has a zip extension. Check that your compressed file opens up correctly and it contains your cpp file.

You will receive no credits if your compressed zip file does not expand or it does not contain the correct file. The naming convention of the zip file is the same as the cpp file (except the extension of the file of course). The name of the zip file should be as follows:

You will receive no credits if your compressed zip file does not expand or it does not contain the correct file. The naming convention of the zip file is the same as the cpp file (except the extension of the file of course). The name of the zip file should be as follows:

SUCourseUserName_YourLastname_YourName_HWnumber.zip

For example zubzipler_Zipleroglu_Zubeyir_hw1.zip is a valid name, but

hw1_hoz_HasanOz.zip, HasanOzHoz.zip

are **NOT** valid names.

Submit via SUCourse ONLY! You will receive no credits if you submit by other means (email, paper, etc.).

Successful submission is one of the requirements of the homework. If, for some reason, you cannot successfully submit your homework and we cannot grade it, your grade will be 0.

Note: if needed, by adding certain modifications, you can use the stack and queue codes given to you in lectures&recitations. If so, you should add them to the zip file that you are going to submit. Besides those mentioned in this homework document, you shouldn't deal with extra input checks. For consistency, the questions related to HW4 should be asked to your TA Seyedpouya Seyedkazemi and the lecturer. Of course, technical help regarding HW4 can be obtained from the TAs during their OH and the lecturer.

Good Luck!
CS204 Team (Artrim Kjamilji, Seyedpouya Seyedkazemi)