Jack Vincent Nicolas

Project 2: Parallelized grep Runner

Video Documentation Link:
https://drive.google.com/file/d/1eBVMPHfR1ymJ7X1ufP84o0CSrxBU8SEy/view?usp=share_link

Documentation:

1. All references used with purpose specified (if any; otherwise, explicitly state that you did not use any external resource)

❖ https://stackoverflow.com/questions/3554120/open-directory-using-c

I used this reference to understand how the directories work and their implementation.

❖ https://www.educative.io/blog/concatenate-string-c

I used this to learn about the different functions that can be used on string from <string.h>.

❖ https://pubs.opengroup.org/onlinepubs/009695399/basedefs/sys/types.h.html

I used this to learn and understand what is under <sys/types.h> as well as how to use them.

❖ https://stackoverflow.com/questions/29559414/how-to-get-the-output-of-grep-in-c

I used this to be able to execute the command grep by using code in C as well as get its output.

❖ https://www.gnu.org/software/libc/manual/html_node/Directory-Entries.html

I used this to learn about the different directory types.

❖ https://www.ibm.com/docs/en/i/7.1?topic=functions-main-function

I used this to understand how to get inputs through the main function.

❖ https://www.ibm.com/docs/en/zos/2.3.0?topic=functions-popen-initiate-pipe-stream-from-process

I used this to understand how popen works as well as its modes.

❖ https://stackoverflow.com/questions/10820377/c-format-char-array-like-printf

I used this to modify the string in a format just like how printf works. This allowed me to create a string which contains the command for grep.

- https://www.tutorialride.com/c-queue-programs/add-delete-items-from-string-queue-c.htm

I used this for the queue of the program.

- https://stackoverflow.com/questions/8152056/scanning-a-directory-and-modify-file-in-c

I modified this to create the absolute path of the directory for each call.

- https://uvle.upd.edu.ph/pluginfile.php/775356/mod_resource/content/1/CS_140_22_1_LAB_8_9.pdf

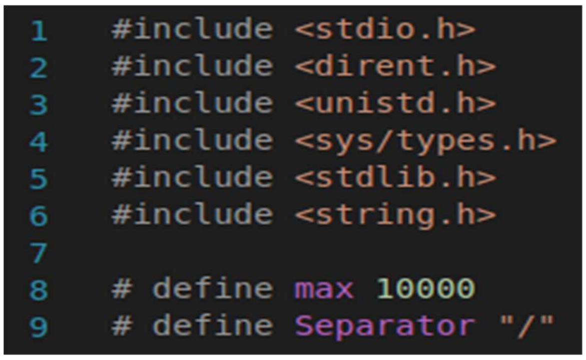
Lastly, I used this file to understand and use pthreads and semaphores.

2. For each version submitted (single-threaded version):

(a) Walkthrough of code execution with sample run having at least N = 2 on a multicore machine, one PRESENT, five ABSENTs, and six DIRs.

This is for a single-threaded version of the project.

First, this part of the code shows the libraries used in the code. It is denoted by "#include <lib>" and allows access to various functions that is helpful in the creation of the program. Then, I defined a couple of terms to have values that will make it easier to refer to them.

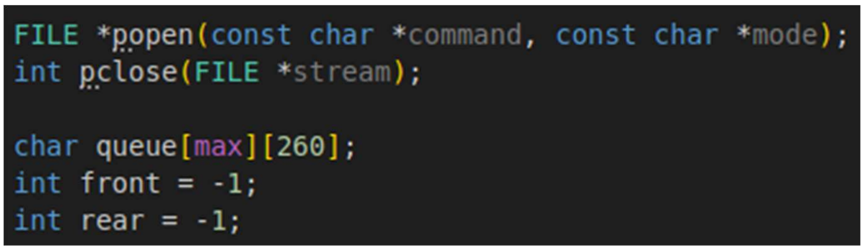


```c
#include <stdio.h>
#include <dirent.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>
#include <string.h>

# define max 10000
# define Separator "/"
```

Next, we have these global variables:

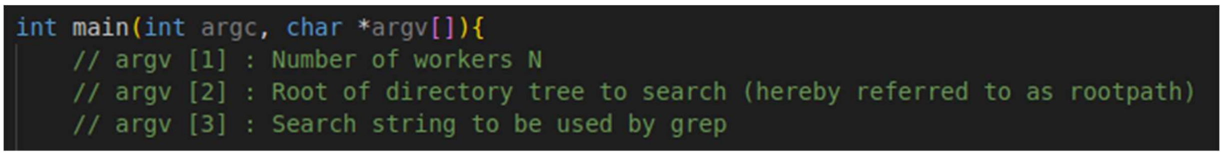
- popen opens a pipe to the shell wherein the command will be run.
- pclose closes the pipe to the shell to prevent leak.
- The queue is initialized globally so that it can always be accessed. It stores the absolute path of a directory which is also the reason for the value 260 in queue.
- Front and rear is initialized to -1 to denote that the queue is empty.



```c
FILE *popen(const char *command, const char *mode);
int pclose(FILE *stream);

char queue[max][260];
int front = -1;
int rear = -1;
```

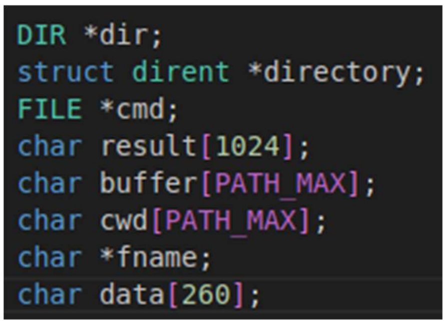
Now going into main, we have the int argc and char *argv[ ]. Argc takes in the argument count passed onto main. Argv[ ] stores the values passed onto main wherein argv[0] contains the filename, argv[1] contains the number of workers, argv[2] contains the rootpath and argv[3] contains the word to be searched for by grep.



```c
int main(int argc, char *argv[]){
    // argv [1] : Number of workers N
    // argv [2] : Root of directory tree to search (hereby referred to as rootpath)
    // argv [3] : Search string to be used by grep
```

Inside main, we initialized the variables that we are going to use.

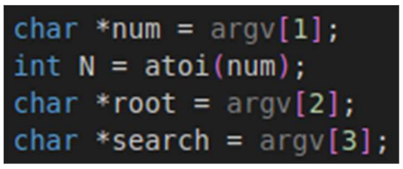
- *dir is for when we open the directory.
- *directory is for when we read the files in the directory and each file has a struct for their information.
- *cmd is where we'll use popen for the grep command.
- Result is where we'll store the result of cmd.
- Buffer is where we store the string command of grep.
- Cwd is where we store the current working directory.
- Fname is where we store the initial directory from argv[2].
- Data is where we store the string to be inserted or deleted in the queue.



```c
DIR *dir;
struct dirent *directory;
FILE *cmd;
char result[1024];
char buffer[PATH_MAX];
char cwd[PATH_MAX];
char *fname;
char data[260];
```

Next, we assigned the arguments to variables so that it is easier to refer to them.

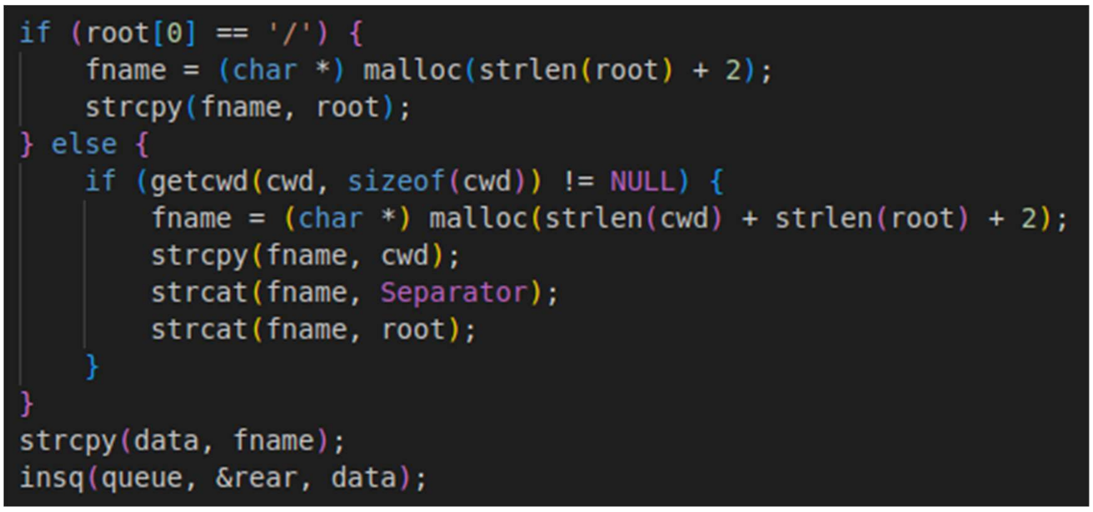
- *num stores argv[1].
- N converts the char *num into and int.
- *root stores argv[2].
- *search stores argv[3].



```c
char *num = argv[1];
int N = atoi(num);
char *root = argv[2];
char *search = argv[3];
```

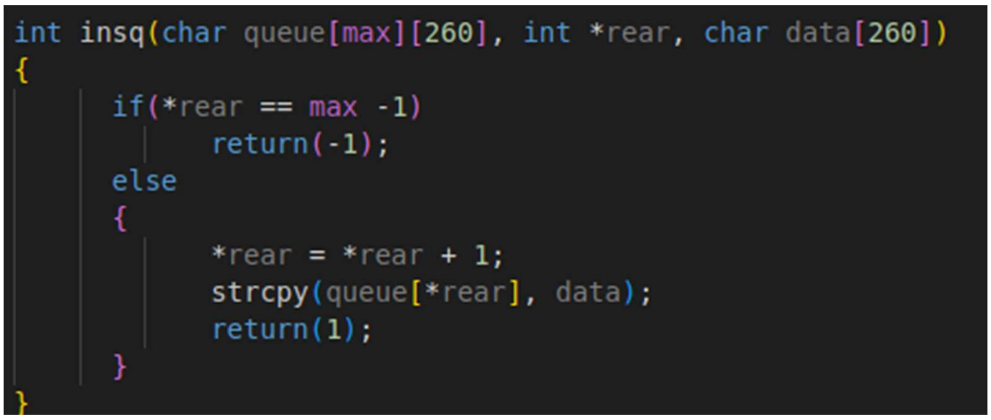
Now that we have the variables, let's move on to the first part of the code.

Here, we have an if else statement that checks the first character of the root. If it has a '/', then that is the start of the absolute path so we allocate space onto fname and transfer the root to it. Else, we get the current working directory and make sure that it isn't null, then we allocate space to fname with the length of the cwd and length of the root + 2 (for separator and an extra). Then, we copy the cwd to fname, then concatenate both the separator and root to it. After the if else statement, we copy fname to data and insert it to the queue.



```c
if (root[0] == '/') {
    fname = (char *) malloc(strlen(root) + 2);
    strcpy(fname, root);
} else {
    if (getcwd(cwd, sizeof(cwd)) != NULL) {
        fname = (char *) malloc(strlen(cwd) + strlen(root) + 2);
        strcpy(fname, cwd);
        strcat(fname, Separator);
        strcat(fname, root);
    }
}
strcpy(data, fname);
insq(queue, &rear, data);
```

The insq function inserts to the queue. It takes the queue, the rear and data variable. It checks if the queue is full by checking if rear is equals to max – 1. Else, it increments the value of rear and inserts the value of data to the queue[value of rear].



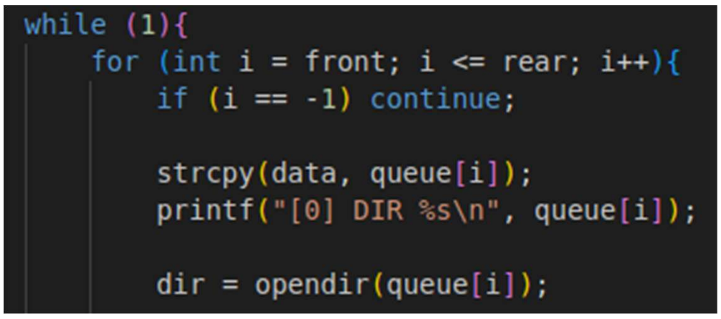
```c
int insq(char queue[max][260], int *rear, char data[260])
{
        if(*rear == max -1)
            return(-1);
        else
        {
            *rear = *rear + 1;
            strcpy(queue[*rear], data);
            return(1);
        }
}
```

Next, we have the main part of the main function.

The whole thing is enclosed in an infinite while loop.

Inside the while loop, we have a for loop which iterates through the queue.

It just continues if i = -1. Otherwise, it copies the first element of the queue to the data and prints “[0] DIR absolute_path” before it opens the directory of the queue[i].

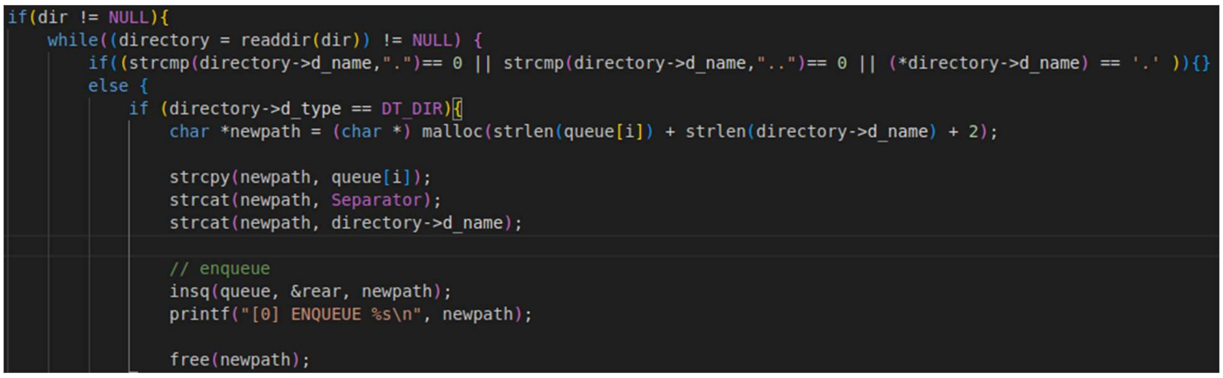


```c
while (1){
    for (int i = front; i <= rear; i++){
        if (i == -1) continue;

        strcpy(data, queue[i]);
        printf("[0] DIR %s\n", queue[i]);

        dir = opendir(queue[i]);
```

If dir isn’t null then we move on to the while loop. For the condition of the while loop, we have directory = readdir(dir), this basically reads the files in the directory continuously until there’s none left. Inside the while loop, we have an if else condition. In the if condition, though not necessary, for the sake of completeness we check if the directory’s name is “.” Or “..” . In the else, statement we have another set of if conditions. Inside the first condition, we check the type of the directory and if it is a directory we create a variable new path with malloc length of queue[i] + length of directory name + 2. Then, we copy queue[i] to new path then concatenate the Separator and the directory->d_name to it. After that we insert it to the queue and print out “[0] ENQUEUE absolute_path”. Then we free the newpath.



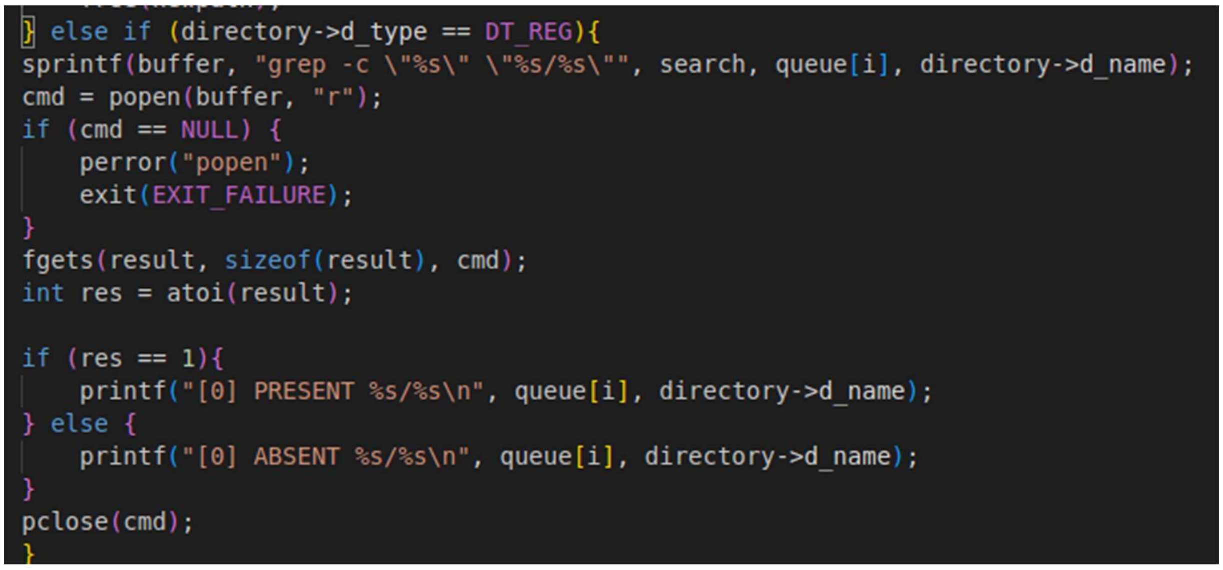
```
if(dir != NULL){
    while((directory = readdir(dir)) != NULL) {
        if((strcmp(directory->d_name,".")== 0 || strcmp(directory->d_name,"..")== 0 || (*directory->d_name) == '.' )){}
        else {
            if (directory->d_type == DT_DIR){
                char *newpath = (char *) malloc(strlen(queue[i]) + strlen(directory->d_name) + 2);

                strcpy(newpath, queue[i]);
                strcat(newpath, Separator);
                strcat(newpath, directory->d_name);

                // enqueue
                insq(queue, &rear, newpath);
                printf("[0] ENQUEUE %s\n", newpath);

                free(newpath);
```

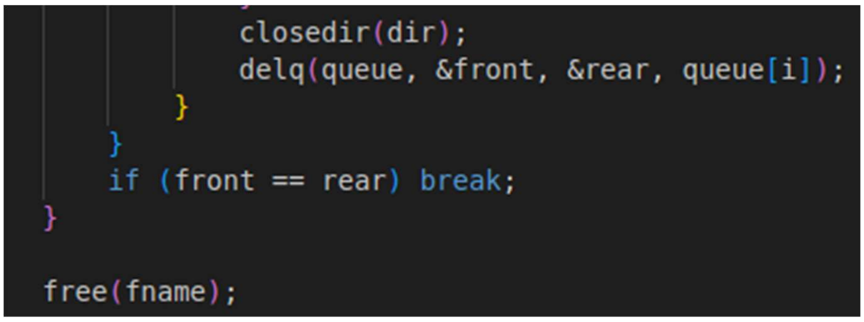
The other if (else if) condition is to check if the directory’s type is DT_REG, meaning it’s a regular file. If so, we used sprintf to create the string for the grep command and store it in buffer (grep -c word_to_be_searched absolute_path/filename). The -c in the grep command is used so that its output is 1 if it managed to find the word in the file and 0 if not. The popen then takes the buffer and runs the command and it occurs inside the cmd variable. We also have a check for cmd in case it fails, though not necessary for the testcases. Next, we have fgets which takes the return value of cmd and stores it in result variable. We then declare variable res with int, wherein we turn the result into an integer. After that, we check if res == 1 and if so, we print “[0] PRESENT absolute_path/filename” , else we print out “[0] ABSENT absolute_path/filename”. Then we use pclose to close the cmd.



```
} else if (directory->d_type == DT_REG){
sprintf(buffer, "grep -c \"%s\" \"%s/%s\"", search, queue[i], directory->d_name);
cmd = popen(buffer, "r");
if (cmd == NULL) {
    perror("popen");
    exit(EXIT_FAILURE);
}
fgets(result, sizeof(result), cmd);
int res = atoi(result);

if (res == 1){
    printf("[0] PRESENT %s/%s\n", queue[i], directory->d_name);
} else {
    printf("[0] ABSENT %s/%s\n", queue[i], directory->d_name);
}
pclose(cmd);
}
```

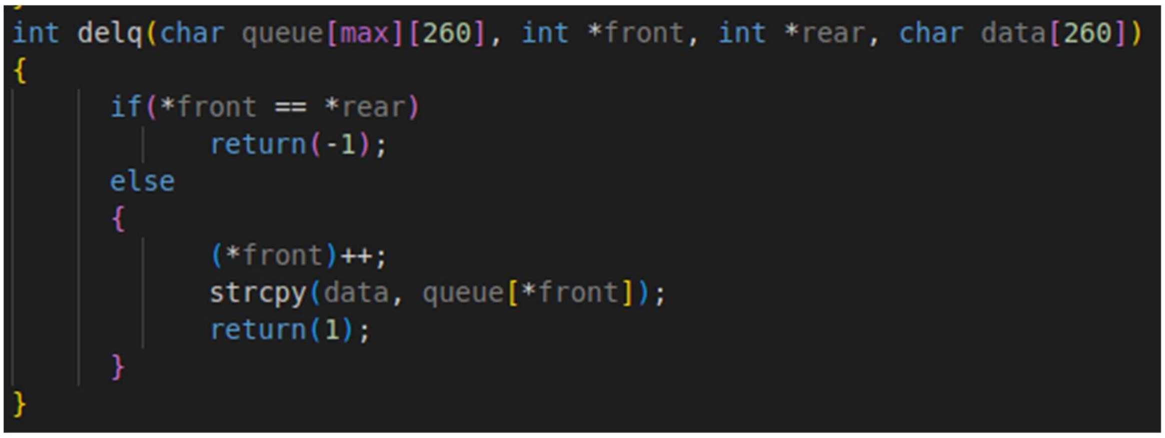
Next, when the current directory has iterated through its contents then we close the directory using closedir(dir). This is followed by a dequeue of the current directory. Then we loop infinitely thanks to the infinite while loop and the condition to break out of it is when there is no longer anything in the queue. Finally, we free fname which we used earlier on in the code.



```
                closedir(dir);
                delq(queue, &front, &rear, queue[i]);
            }
        }
        if (front == rear) break;
    }

    free(fname);
```

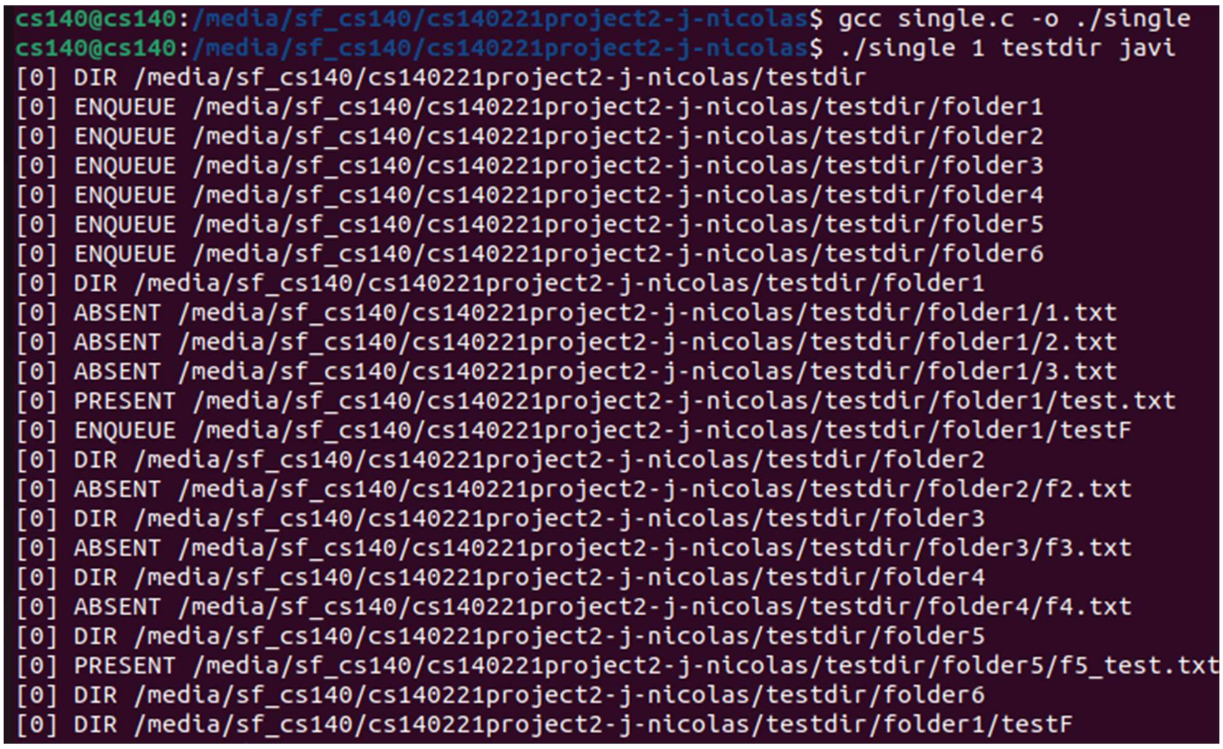
This is the function delq which basically deletes the data from the queue. It takes in the queue, the front value, the rear value and the data to be deleted.



```
int delq(char queue[max][260], int *front, int *rear, char data[260])
{
        if(*front == *rear)
            return(-1);
        else
        {
            (*front)++;
            strcpy(data, queue[*front]);
            return(1);
        }
}
```

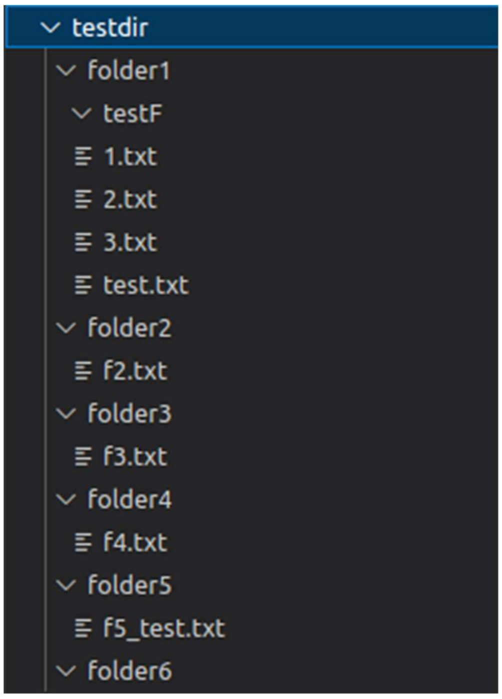
This is the sample run having neglecting N, 2 PRESENT, 6 ABSENTs, and 8 DIRs.

The initial rootpath was enqueued outside the loop which is why it wasn't printed as enqueued, but instead listed as DIR. Then, this directory was opened and enqueues all of its content since they are all folders. After this, we dequeued the initial path so the current queue has folders 1-6. Since, the directory loop is inside a while loop, we check each queue again starting from folder1, which contains txt files and a folder. We can see that there are 3 absent and 1 present and 1 enqueue. Then we move to folder 2 and check its contents and so on until the queue is empty.



```
cs140@cs140:/media/sf_cs140/cs140221project2-j-nicolas$ gcc single.c -o ./single
cs140@cs140:/media/sf_cs140/cs140221project2-j-nicolas$ ./single 1 testdir javi
[0] DIR /media/sf_cs140/cs140221project2-j-nicolas/testdir
[0] ENQUEUE /media/sf_cs140/cs140221project2-j-nicolas/testdir/folder1
[0] ENQUEUE /media/sf_cs140/cs140221project2-j-nicolas/testdir/folder2
[0] ENQUEUE /media/sf_cs140/cs140221project2-j-nicolas/testdir/folder3
[0] ENQUEUE /media/sf_cs140/cs140221project2-j-nicolas/testdir/folder4
[0] ENQUEUE /media/sf_cs140/cs140221project2-j-nicolas/testdir/folder5
[0] ENQUEUE /media/sf_cs140/cs140221project2-j-nicolas/testdir/folder6
[0] DIR /media/sf_cs140/cs140221project2-j-nicolas/testdir/folder1
[0] ABSENT /media/sf_cs140/cs140221project2-j-nicolas/testdir/folder1/1.txt
[0] ABSENT /media/sf_cs140/cs140221project2-j-nicolas/testdir/folder1/2.txt
[0] ABSENT /media/sf_cs140/cs140221project2-j-nicolas/testdir/folder1/3.txt
[0] PRESENT /media/sf_cs140/cs140221project2-j-nicolas/testdir/folder1/test.txt
[0] ENQUEUE /media/sf_cs140/cs140221project2-j-nicolas/testdir/folder1/testF
[0] DIR /media/sf_cs140/cs140221project2-j-nicolas/testdir/folder2
[0] ABSENT /media/sf_cs140/cs140221project2-j-nicolas/testdir/folder2/f2.txt
[0] DIR /media/sf_cs140/cs140221project2-j-nicolas/testdir/folder3
[0] ABSENT /media/sf_cs140/cs140221project2-j-nicolas/testdir/folder3/f3.txt
[0] DIR /media/sf_cs140/cs140221project2-j-nicolas/testdir/folder4
[0] ABSENT /media/sf_cs140/cs140221project2-j-nicolas/testdir/folder4/f4.txt
[0] DIR /media/sf_cs140/cs140221project2-j-nicolas/testdir/folder5
[0] PRESENT /media/sf_cs140/cs140221project2-j-nicolas/testdir/folder5/f5_test.txt
[0] DIR /media/sf_cs140/cs140221project2-j-nicolas/testdir/folder6
[0] DIR /media/sf_cs140/cs140221project2-j-nicolas/testdir/folder1/testF
```

This is how the folder looks like:



```
testdir
  folder1
    testF
    1.txt
    2.txt
    3.txt
    test.txt
  folder2
    f2.txt
  folder3
    f3.txt
  folder4
    f4.txt
  folder5
    f5_test.txt
  folder6
```