Folder Cleaner Documentation

Problem P5 : Folder Cleaner

You are to build a Folder Cleaner. It deletes files that are:

1. Redundant files (i.e., a newer copy is present),
2. Files old for more than *N* number of months,
3. Empty files (i.e., no content),
4. Files that have not been accessed for at least M number of times.

Variations of given problem (HARDER VERSION OF THE CODE):

1) Develop an interactive command-line interface (CLI) with options for users to choose cleaning criteria, set thresholds, and interactively confirm file deletions.

In which Code Will Ask for every file whether the user wants to delete that particular file or not. (Y/N).

2)Allow users to customise cleaning rules during the cleaning process.

Ask for the types of the file. Which type(s) of files the user wants to delete.

Group : G4
Group Members Names And Contributions:

Dhola Krish Hareshbhai (202301189) --- 30%
Aryan Rathod (202301164) --- 30%
Neev Vegada (202301031) --- 20%
Tirth Koradiya (202301018) --- 20%

GitHub repo link for the source code:

Link

Introduction:

The Folder Cleaner is a command-line utility designed to assist users in organising and decluttering their directories by removing redundant or outdated files. It provides functionalities to clean up files within specified folders based on various criteria such as file redundancy, age, and content.

Usage:

Compilation: Compile the program using a C++ compiler.

Input Preparation: Prepare a text file containing folder paths and file names to clean. First line of the input file contains the folder path followed by file names in that folder.see the example input file named hello[1].txt

Also make a test folder in your local machine to run the programme on it. The test folder should contain some redundant files,empty files.

The access frequency should be mentioned in the input file for each of the files.

Execution: Run the compiled program and provide the values of threshold mons and number of accesses.

Cleaning Process:

The program will iterate through each folder specified in the input file and clean it based on the specified criteria. It will delete empty files, files older than a specified duration, files not accessed at least a specified times and redundant files based on content.

Data Structure Used:

Map (`contentMap`):

- **Purpose:** Used to store file contents as keys and corresponding file paths as values.
- **Reason for Use:**
  - Allows efficient lookup of file contents to identify duplicate files.
  - Provides constant-time complexity (O(1)) for insertion, deletion, and retrieval operations in average and worst cases.
  - Ensures uniqueness of keys, preventing duplicate entries for the same file content.
- **Time Complexity:**
  - Insertion: O(1) average case, O(log n) worst case.
  - Deletion: O(1) average case, O(log n) worst case.
  - Search: O(1) average case, O(log n) worst case.
- **Space Complexity:** O(n), where n is the number of unique file contents stored in the map.

Vector (filePaths, filenames, duplicateFiles):

- **Purpose:** Used to store file paths, filenames, and duplicate file paths temporarily during folder cleaning.
- **Reason for Use:**
  - Provides dynamic resizing, allowing flexibility in storing file-related data.
  - Allows sequential access to elements, facilitating iteration and manipulation of file-related data.
- **Time Complexity:**
  - Insertion: O(1) amortised time complexity.
  - Deletion: O(1) average case.
  - Search: O(n) linear time complexity for searching.
- **Space Complexity:** O(n), where n is the number of files or filenames stored in the vector.

Explanation:

- **Map (`contentMap`):** Chosen for efficient detection of duplicate files based on content. Offers constant-time complexity for essential operations, ensuring fast lookup and deletion. Guarantees uniqueness of file contents, preventing redundancy in storage. Balances time and space complexity considerations, suitable for handling potentially large datasets efficiently.
- **Vector (`filePaths`, `filenames`, `duplicateFiles`):** Selected for its simplicity and effectiveness in storing and managing lists of file paths, filenames, and duplicate file paths. Provides dynamic resizing, accommodating varying numbers of elements without preallocation concerns. Supports sequential access, enabling easy iteration through file-related data during folder cleaning. While not the most efficient data structure in terms of time and space complexity, its versatility and ease of use make it suitable for the task at hand.

Reason to use that Data Structure:

Vectors:

- **Space Efficiency:** Vectors provide contiguous memory allocation, which minimises memory overhead and ensures efficient usage of memory.
- **Time Efficiency:** Vectors offer O(1) time complexity for accessing elements by index, which is crucial for fast retrieval and manipulation of filenames, access frequencies, and other metadata.

Map:

- **Space Efficiency:** Maps store data in key-value pairs, which is suitable for associating file content with file paths. While they have a higher space complexity compared to arrays, their ability to map unique keys to values efficiently compensates for this.
- **Time Efficiency**: Maps offer O(log n) time complexity for common operations such as insertion, deletion, and search. This makes them well-suited for maintaining a collection of unique file contents and their corresponding file paths.

String streams:

- **Space Efficiency:** String streams provide a lightweight way to parse string data without significant additional memory overhead. They are ideal for extracting information from string representations of file data.
- **Time Efficiency**: String streams offer efficient parsing operations with a time complexity proportional to the length of the input string. This allows for quick extraction of filenames, access frequencies, and creation dates from the input file records.

Time Complexity:

- **Reading Input Files:** O(n), where n is the total number of files in all folders combined. This involves iterating over each line in the input file.
- **File Existence Check:** O(n), where n is the total number of files. This involves checking the existence of each file in the file system.
- **File Empty Check:** O(n), where n is the total number of files. This involves checking whether each file is empty.
- **File Age Check:** O(n), where n is the total number of files. This involves checking the age of each file.
- **File Access Frequency Check:** O(n), where n is the total number of files. This involves checking the access frequency of each file.
- **Duplicate File Detection:** O(n^2 * m), where n is the total number of files and m is the average number of characters in each file. This involves comparing the content of each file with every other file's content.

- **File Deletion:** O(k), where k is the number of files marked for deletion. This involves deleting the files marked for deletion.

The overall time complexity is determined by the most significant factor, which is the duplicate file detection step, resulting in **O(n^2 * m)** complexity.

Space Complexity:

- **Reading Input Files:** O(n), where n is the total number of files. This involves storing file names, metadata, and other input-related data.
- **Data Structures for Duplicate File Detection:** O(n * m), where n is the total number of files and m is the average number of characters in each file. This involves storing file contents or hashes for duplicate detection.

The overall space complexity is determined by the space required for storing file names, metadata, and file contents or hashes during duplicate file detection, resulting in **O(n * m)** complexity.

Program Flow Overview:

Initialization:

- The program imports necessary libraries such as <iostream>, <fstream>, <sstream>, <vector>, <algorithm>, <cstring>, <dirent.h>, <sys/stat.h>, <ctime>, and <map> to handle file operations, string manipulation, directory handling, and data structures. It utilizes the std namespace for streamlined access to standard library functions.

Main Function Execution:

- The main function starts by opening the input file, hello[1].txt, using an ifstream object named inputFile. It reads folder names and associated file names with their access frequencies from the input file. For each folder listed in the input file, the program passes folder names, filenames, and access frequencies to the cleanFolder function for further processing.

Clean Folder Function Execution:

- The cleanFolder function begins by prompting the user to input minimum months (N) and minimum number of accesses (m) required for file retention. It constructs full file paths for all files in the folder using the folder path and filenames provided. For each file within the folder, the function evaluates its eligibility for deletion based on specified criteria. Actions include deleting empty files, old files, and redundant files based on content. Redundancy is determined by comparing file contents and keeping track of unique content using a map.

How the Program Works for a File:

Reading Input:

- The program reads input from the provided file (hello[1].txt), which contains folder names followed by associated file names and access frequencies. It parses this input to extract folder names, filenames, and associated access frequencies for further processing.

Folder Iteration:

- For each folder listed in the input file, the program collects filenames and their associated access frequencies for evaluation.

Cleaning Process:

- The cleanFolder function processes each file within the specified folder to determine its eligibility for deletion based on the specified criteria. Files meeting the criteria are marked for deletion from the system, optimising storage space and organisation.

Handling Duplicates:

- Duplicate files (files with identical content) are identified based on their content similarity. The function retains only one copy of the duplicate files while deleting the rest, ensuring efficient use of storage resources.

Additional Considerations:

- The program's user-friendly interface prompts users to input criteria for minimum months and minimum number of accesses, providing flexibility and customization options. It employs robust file handling mechanisms to ensure efficient cleaning and optimization of storage space.

Format of the input file:

Here's an example of how the input file should be formatted:

```
<FolderName1>

<FileName1> <Number of access> <Date1>

<FileName2> <Number of access> <Date2>

...

-

<FolderName2>

<FileName1> <Number1> <Date1>

<FileName2> <Number2> <Date2>

...
```

-

...

- Each section starts with the name of a folder.
- Following the folder name, there are lines representing files within that folder.
- Each file entry consists of three parts separated by spaces:
- <FileName>: The name of the file.
- <Number of access>: Some numerical value associated with the file (possibly representing access frequency).
- <Date>: Date on which the file was modified last.
- The section ends with a hyphen -, indicating the end of files in that folder.

The input file contains the path of the folder on which the operations will be performed followed by a list of the file names as shown in the txt file named hello[1] posted on github repo.

Pseudo Code:

```
function fileExists(filename):
  // Check if the file exists
  return result of stat(filename)

function isFileEmpty(filename):
 //Check if the file is empty
  open the file
  return true if the file is empty, false otherwise

function isFileOlderThanNMonths(filename, N):
  // Check if the file is older than N months
```

```
    get the file's modification time
    calculate the difference between the current time and file's modification time in months
    return true if the difference is greater than N, false otherwise


function deleteFile(filename, reason):
    // Delete the file
    attempt to remove the file
    if removal is successful, log deletion message
    otherwise, log deletion error message


function cleanFolder(folderPath, filenames, numbers):
    // Clean the folder based on specified criteria
    prompt user for minimum months (N) and minimum access frequency (m)
    create an empty map contentMap to store file contents and their paths
    create an empty list duplicateFiles to store duplicate file paths
    for each filename in filenames:
        construct full file path using folderPath and filename
        if the file does not exist:
            log error message and continue to the next file
        if the file is empty:
            delete the file and continue to the next file
        if the file is older than N months:
            delete the file and continue to the next file
        if the file access frequency is less than m:
            delete the file and continue to the next file
        read the file content
    if the content already exists in contentMap:
      add the file path to duplicateFiles
    else:
      store the content and file path in contentMap
  for each duplicateFile in duplicateFiles:
    delete the duplicate file
  return


function main():
  open the input file containing folder names, filenames, and access numbers
  if the input file fails to open:
    log error message and exit program
  for each folder entry in the input file:
    read the folder path
    create empty lists filenames, numbers, and dates
    while there are filenames, numbers, and dates available:
      read filename, number, and date
      if filename is "-", break out of the loop
      add filename, number, and date to respective lists
    call cleanFolder function with folderPath, filenames, and numbers as arguments
  close the input file
  return
```

Here is the explanation of all the functions:

Function.1: `int main()`

Description:

This is the main function responsible for orchestrating the cleaning process of multiple folders by calling the cleanFolder function for each folder. It reads input data from a file, parses it, and passes the necessary parameters to the cleanFolder function.

Parameters:

None

Return Type:

int: Returns 0 upon successful execution.

Logic:

Open the input file ("hello[1].txt").
Read each line, representing a folder, from the input file.

For each folder:

a. Clear the vectors filenames, numbers, and dates to store file information.

b. Read each line containing file information from the input file until a "-" is encountered, indicating the end of file entries for the current folder.

c. Store the filename, number, and date in the respective vectors.

d. Call the cleanFolder function with the folder path and file information vectors.

Repeat steps 3 for all folders in the input file.

Close the input file.

Return 0 to indicate successful execution.

Time Complexity:

- Reading each line from the input file has a time complexity of O(n), where n is the number of lines in the file.
- Parsing the file information and calling the cleanFolder function for each folder has a time complexity proportional to the number of files in each folder.

Space Complexity:

- The space complexity primarily depends on the size of the input file and the number of files in each folder. However, the vectors used to store file information (filenames, numbers, and dates) contribute to the space complexity, which is linear with respect to the number of files.

Function.2: `bool fileExists(const string& filename)`

Description:

This function checks if a file exists at the specified path.

Parameters:

1. filename: A constant reference to a string representing the file path.

Return Type:

1. bool: Returns true if the file exists, otherwise false.

Logic:

1. It uses the stat function from the <sys/stat.h> header to check the file's status.
2. If the stat function returns 0, indicating success, the function returns true, indicating that the file exists.
3. If the stat function returns a non-zero value, the function returns false, indicating that the file does not exist.

Time Complexity:

1. Time complexity: O(1)
2. The stat function typically has constant time complexity, as it directly queries the file system metadata.

Space Complexity:

1. Space complexity: O(1)
2. The function uses minimal memory regardless of the file's existence.

Function.3: `bool isFileEmpty(const string& filename)`

Description:

This function checks if a file is empty by peeking into its content.

Parameters:

1. filename: A constant reference to a string representing the file path.

Return Type:

1. bool: Returns true if the file is empty, otherwise false.

Logic:

1. It opens the file using an input file stream (ifstream).
2. It uses the peek function to check the first character in the file stream.
3. If peek returns EOF (end-of-file), indicating an empty file, the function returns true.
4. If peek returns a valid character, the function returns false, indicating a non-empty file.

Time Complexity:

1. Time complexity: O(1)
2. The function reads only the first character of the file to determine if it's empty.

Space Complexity:

1. Space complexity: O(1)
2. The function uses minimal memory regardless of the file's size.

Function: `bool isFileOlderThanNMonths(const string& filename, int N)`

Description:

This function checks if a file is older than a specified number of months.

Parameters:

1. filename: A constant reference to a string representing the file path.
2. N: An integer representing the number of months.

Return Type:

1. bool: Returns true if the file is older than N months, otherwise false.

Logic:

1. It retrieves the file's modification time using the stat function.
2. It calculates the difference in time between the current time and the file's modification time in months.
3. If the difference is greater than N, the function returns true, indicating that the file is older than n months.
4. If the difference is less than N or equal to N, the function returns false, indicating that the file is not older than N months.

Time Complexity:

1. Time complexity: O(1)
2. The function involves simple arithmetic operations on time values retrieved from the file system metadata.

Space Complexity:

1. Space complexity: O(1)
2. The function uses minimal memory regardless of the file's age.

Function: `bool deleteFile(const string& filename, const string& reason)`

Description:

This function deletes a file from the filesystem and logs the action with a reason for deletion.

Parameters:

1. filename: A constant reference to a string representing the file path.
2. reason: A constant reference to a string representing the reason for deletion.

Return Type:

1. bool: Returns true if the file is successfully deleted, otherwise false.

Logic:

1. It attempts to delete the file using the remove function.
2. If remove returns 0, indicating successful deletion, the function prints the deletion action along with the provided reason and returns true.
3. If remove returns a non-zero value, indicating failure, the function logs an error message and returns false.

Time Complexity:

1. Time complexity: O(1)
2. The remove function typically has constant time complexity, as it directly interacts with the file system.

Space Complexity:

1. Space complexity: O(1)
2. The function uses minimal memory regardless of the file's size.

Function: `void cleanFolder(const string& folderPath, const vector<string>& filenames, const vector<int>& numbers)`

Description:

This function is responsible for cleaning the specified folder by deleting redundant files, files older than a certain number of months, empty files, and files that have not been accessed a minimum number of times. It operates based on the provided criteria and deletes files accordingly.

Parameters:

- folderPath: A constant reference to a string representing the path of the folder to be cleaned.
- filenames: A constant reference to a vector of strings containing the names of the files in the folder.
- numbers: A constant reference to a vector of integers representing the access frequency of each file.

Return Type:

void: This function does not return any value.

Logic:

Prompt the user to enter the minimum number of months (N) and the minimum access frequency (m) required for a file to be retained. Construct full file paths by concatenating the folder path with each filename.

Iterate over each file in the folder:

a. Check if the file exists. If not, log an error message and continue to the next file.

b. Check if the file is empty. If empty, delete the file and continue to the next file.

c. Check if the file is older than N months. If so, delete the file and continue to the next file.

d. Check if the file's access frequency is less than m. If so, delete the file and continue to the next file.

e. If the file passes all checks, read its content and store it in a map along with its path. If the content already exists in the map, mark the file as a duplicate.
Delete duplicate files.
Log the deletion of each file.

Time Complexity:

- The time complexity of this function depends on the number of files in the folder and the operations performed on each file.
- Constructing file paths, checking file existence, and accessing file attributes have a time complexity of O(n), where n is the number of files.
- Deleting files and reading file contents are generally O(1) operations.
- Iterating over files and performing checks has a time complexity of O(n), where n is the number of files.

Space Complexity:

- The space complexity is primarily determined by the size of the input vectors (filenames, numbers) and the map (contentMap) used to store file content.
- The map's space complexity depends on the number of unique file contents, which can range from O(1) to O(n), where n is the number of files.
- Overall, the space complexity is linear with respect to the number of files in the folder. O(n)

`ifstream inputFile("hello[1].txt")`

Description:

This function opens the input file named "hello[1].txt" for reading.

Return Type:

1. ifstream: Returns an input file stream object.

Logic:

1. It attempts to open the file named "hello[1].txt" using an input file stream.
2. If the file is successfully opened, the input file stream object is returned.
3. If the file cannot be opened, an error message is displayed, and the program may terminate depending on error handling.

Time Complexity:

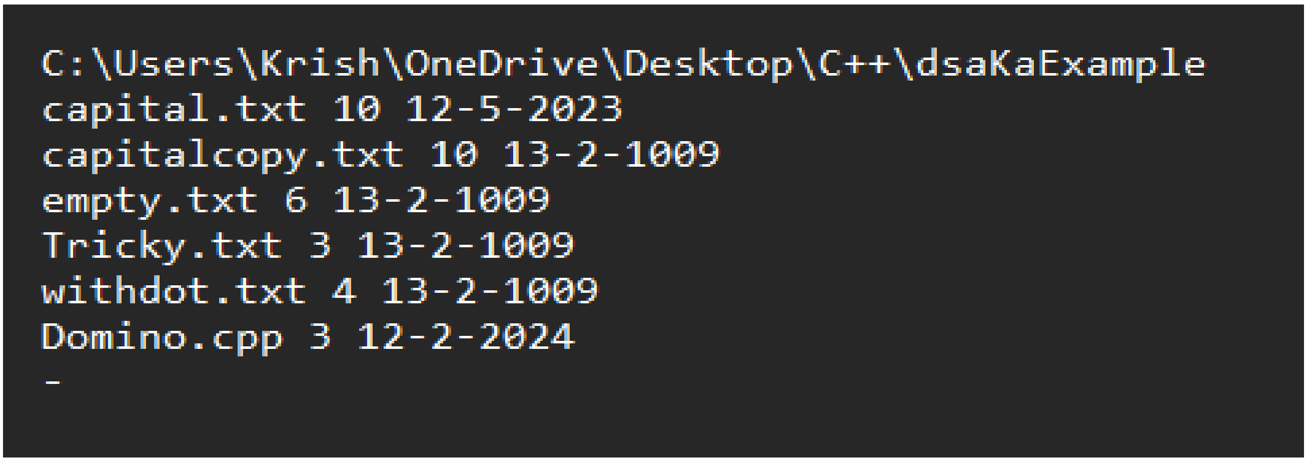
1. Time complexity: O(1)
2. Opening a file involves minimal computational overhead, typically dependent on the file system and operating system.

Space Complexity:

1. Space complexity: O(1)
2. The function uses minimal memory regardless of the file's size.

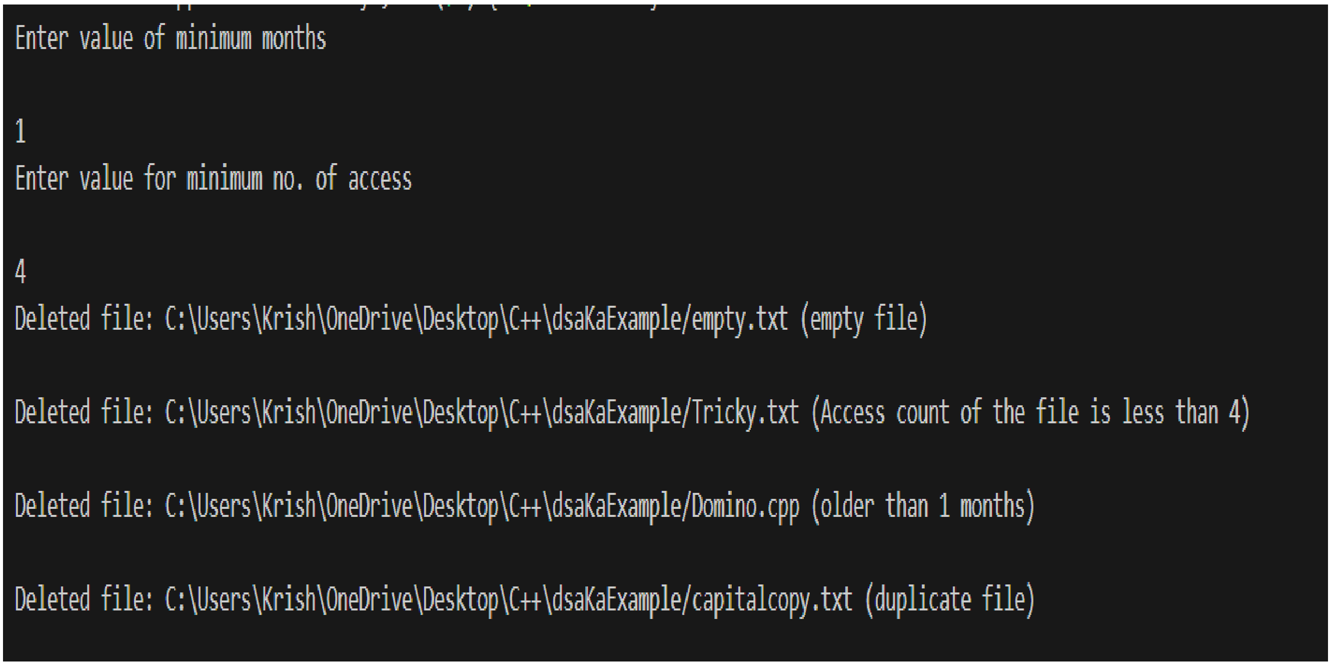
ScreenShots of Input/Output:

Input:



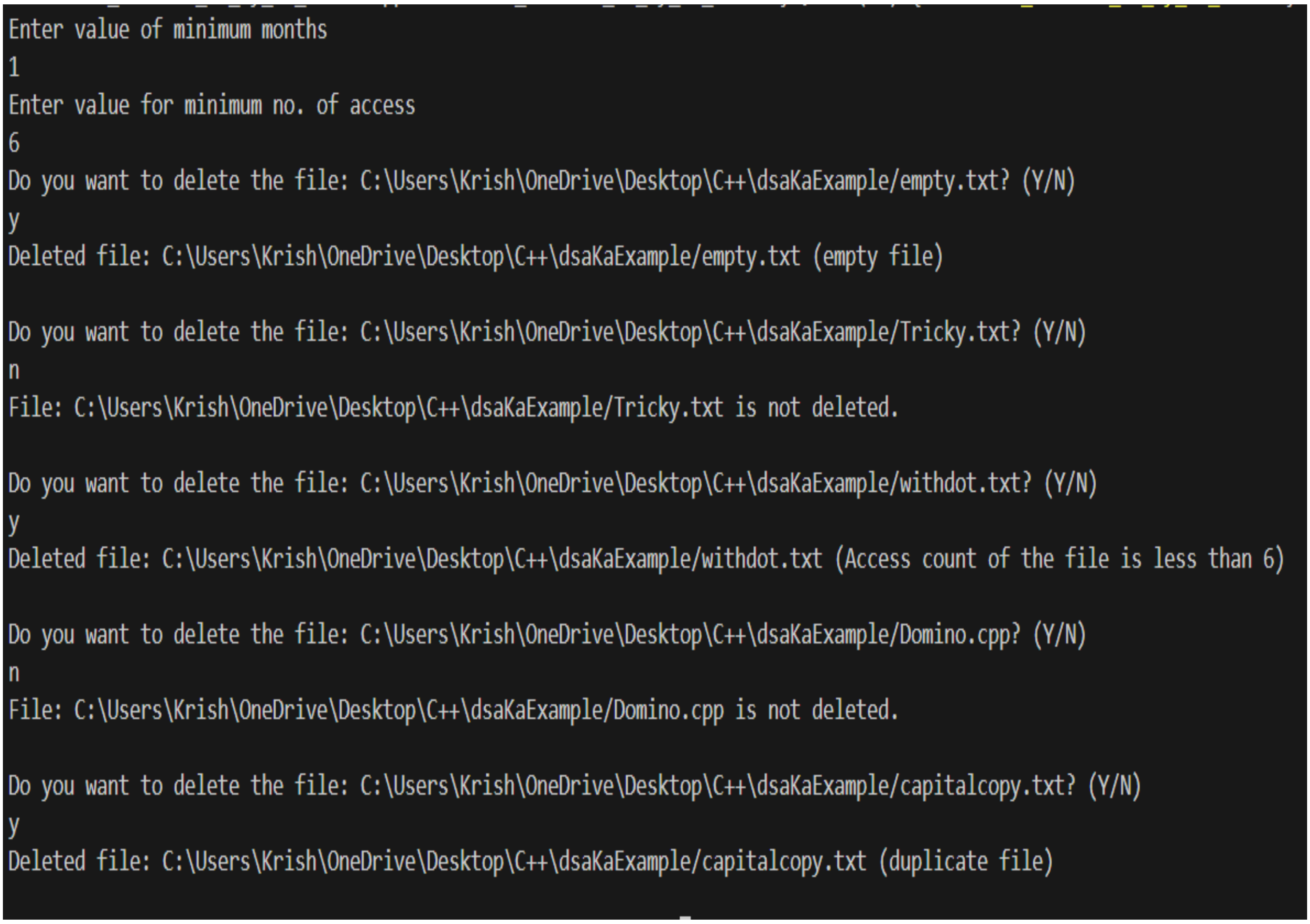
```
C:\Users\Krish\OneDrive\Desktop\C++\dsaKaExample
capital.txt 10 12-5-2023
capitalcopy.txt 10 13-2-1009
empty.txt 6 13-2-1009
Tricky.txt 3 13-2-1009
withdot.txt 4 13-2-1009
Domino.cpp 3 12-2-2024
-
```

Output (Main problem):



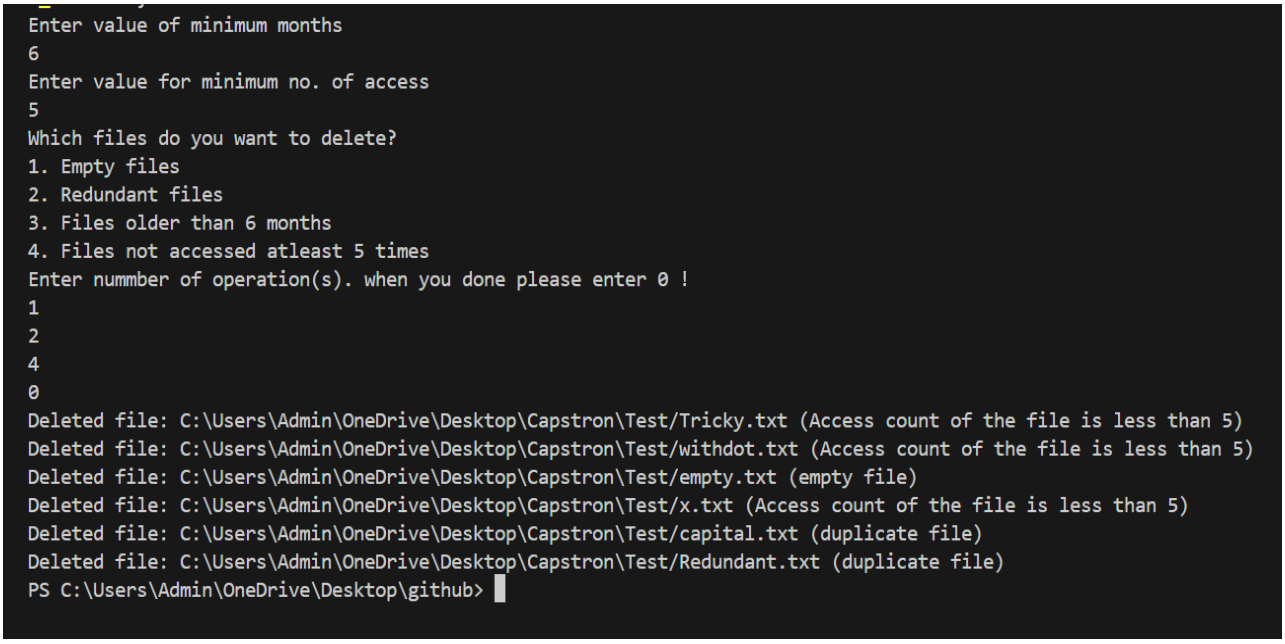
```
Enter value of minimum months

1
Enter value for minimum no. of access

4
Deleted file: C:\Users\Krish\OneDrive\Desktop\C++\dsaKaExample/empty.txt (empty file)

Deleted file: C:\Users\Krish\OneDrive\Desktop\C++\dsaKaExample/Tricky.txt (Access count of the file is less than 4)

Deleted file: C:\Users\Krish\OneDrive\Desktop\C++\dsaKaExample/Domino.cpp (older than 1 months)

Deleted file: C:\Users\Krish\OneDrive\Desktop\C++\dsaKaExample/capitalcopy.txt (duplicate file)
```

Output(Variation 1):



```
Enter value of minimum months
1
Enter value for minimum no. of access
6
Do you want to delete the file: C:\Users\Krish\OneDrive\Desktop\C++\dsaKaExample/empty.txt? (Y/N)
y
Deleted file: C:\Users\Krish\OneDrive\Desktop\C++\dsaKaExample/empty.txt (empty file)

Do you want to delete the file: C:\Users\Krish\OneDrive\Desktop\C++\dsaKaExample/Tricky.txt? (Y/N)
n
File: C:\Users\Krish\OneDrive\Desktop\C++\dsaKaExample/Tricky.txt is not deleted.

Do you want to delete the file: C:\Users\Krish\OneDrive\Desktop\C++\dsaKaExample/withdot.txt? (Y/N)
y
Deleted file: C:\Users\Krish\OneDrive\Desktop\C++\dsaKaExample/withdot.txt (Access count of the file is less than 6)

Do you want to delete the file: C:\Users\Krish\OneDrive\Desktop\C++\dsaKaExample/Domino.cpp? (Y/N)
n
File: C:\Users\Krish\OneDrive\Desktop\C++\dsaKaExample/Domino.cpp is not deleted.

Do you want to delete the file: C:\Users\Krish\OneDrive\Desktop\C++\dsaKaExample/capitalcopy.txt? (Y/N)
y
Deleted file: C:\Users\Krish\OneDrive\Desktop\C++\dsaKaExample/capitalcopy.txt (duplicate file)
```

Output (Variation 2):



```
Enter value of minimum months
6
Enter value for minimum no. of access
5
Which files do you want to delete?
1. Empty files
2. Redundant files
3. Files older than 6 months
4. Files not accessed atleast 5 times
Enter nummber of operation(s). when you done please enter 0 !
1
2
4
0
Deleted file: C:\Users\Admin\OneDrive\Desktop\Capstron\Test/Tricky.txt (Access count of the file is less than 5)
Deleted file: C:\Users\Admin\OneDrive\Desktop\Capstron\Test/withdot.txt (Access count of the file is less than 5)
Deleted file: C:\Users\Admin\OneDrive\Desktop\Capstron\Test/empty.txt (empty file)
Deleted file: C:\Users\Admin\OneDrive\Desktop\Capstron\Test/x.txt (Access count of the file is less than 5)
Deleted file: C:\Users\Admin\OneDrive\Desktop\Capstron\Test/capital.txt (duplicate file)
Deleted file: C:\Users\Admin\OneDrive\Desktop\Capstron\Test/Redundant.txt (duplicate file)
PS C:\Users\Admin\OneDrive\Desktop\github>
```