***Operating Systems***

***Spring 2020***

***Project 1: Disk space analyzer tool***

**Rinal Hamdy (900171803) - Ahmed Ibrahim (900121237)**

**Andrew Kamal (900170228) - Samah Ayman (900171848)**

**Introduction**

This project is an implementation of a disk space analyzer by which the users can get a closer picture of the files and folders occupying the disk. Disk space analyzers are becoming one of the essential types of disk utility tools for system administrators to allow them or other users to visualize how the disk space is occupied by different applications and documents. It is also beneficial to pinpoint and prevent important directories and partitions from running out of space. Providing such analysis tools enable their users to detect the most promising ways to clean up the disk and have more space available on it.

**Background**

Disk analyzer tools are provided to check disk space usage details and to provide clear information about what exactly is consuming the space provided on the disk by what percentage. Such information is very important to help people determine what items to remove or to move to another place to free up some space in the disk to avoid any deterioration in the system performance due to the insufficiency of free space. There can be so much information enclosed in these file systems, so keeping track of all these files and folders is almost impossible without a software that can get it done in seconds. There are many different technical implementations to achieve this goal of analyzing the disk. Thus, in the survey, our concern is to assess the most popular and well-functioning disk analyzer tools that are available for Linux operating system in order to get a sense of the good features in each of them and understand their drawbacks and consequently, build our own disk analysis tool that combines all the good features in one place and avoid the drawbacks in each of them. Our end goal is to provide a tool that’s reliable and good enough to beat all the already existing tools. The specifications that will be followed in the design of the disk analyzer tool are time and space complexity, memory usage, and parallelism.

**Related work**

This paper presents an analysis of the performance of different disk analyzer implementations. Various techniques have been developed over the years to accomplish this task, so determining which method is better will be determined according to two main criteria which are the data structure used to represent and keep the information of the directory and the interface provided with the analyzer tool and how effective it is in telling the information.

A.)  **Disk usage analyzer**, also known as, “Baobab”, this tool uses tree data structures to store the information related to the files and the directories. The graphical user interface provided by this program can be tailored according to the content being used scanned. This program uses a function that mainly relies on some APIs to traverse all the contents of the directory path given to be analyzed and return the size of each and the overall size.

B.) **QDirStat** is another disk analyzer implementation that also uses trees to store the content of the path provided to it. What is added to this implementation is that this program provides clean up tools to help the user clean up the disk when it is occupied by many files. However, the interface of this program makes it very hard on the user to understand and visualize the content that is consuming his disk space as it appears in rectangular and square grids and they are not intuitive at all

C.) **Ncurses Disk Usage (ncdu)** is another tool that uses linked lists as their main data structure to store the contents of the directory. This tool is a text-based user interface; it doesn’t provide any visual illustration of the contents and the internals of the directory. This implementation by default scans the entire system automatically. It also provides clean up options to the user and these options can be tested using the keyboard commands specified in the documentation of this program. This way of giving information as text is not very indicative and it doesn’t provide the best visual aid to help the user understand the information presented in the program. This implementation also traverses the disk in the regular manner using the function that relies on the APIs to use system calls to inquire about the sizes and traverse the directory.

D.) **XDiskUsage** program is another implementation that also uses trees and displays the information in a tabular format; it looks pretty much like storing information in excel tables. This tabular format makes it easy on the user to navigate through the files and folders consuming the space in his system.

E.)  **FlightLight** program also uses trees and implements the GUI using Qt and displays the information using pie charts and it uses the listing technique described above and used by other implementations as well.

This assessment covered only five tools since they are the most popular among all other tools and discussing and assessing the rest of the analyzers will be a bit redundant since they are very much similar to the ones already described and assessed above.

From analyzing the pros and cons of each of the above implementations, the below design criteria were agreed upon to be followed during our implementation process

1. BST trees will be used as our main data structure, instead of regular trees, to provide the quickest means to search and traverse the content of the directories.

2. The GUI will represent the directory contents using pie charts as they are the most intuitive way to understand the information

3. The interface will be interactive and will provide cleanup options to the user

**Represented Classes**

There are mainly three classes that represent the backend of the program. BinarySearchTree class, which is a templated class “template <class NODE,typename ITEM> class BinarySearchTree”. The class NODE is represented by “template <typename ITEM> class BSTNode”, which is the node inserted in the tree. The class ITEM is represented by class File, which represents the data being inserted in the node of the Binary Search Tree.

Another point that is worth mentioning is that we overloaded the operators in the File class in order to allow us to compare and deal with the items that will be inserted in the tree.

**Implementation and Design scheme**

The main data structure used in this project is a binary search tree. The main reason behind choosing this data structure to work with is mainly to obtain an ordered list of objects in the quickest possible manner as well as providing the user with a quick means to build, search, and traverse his directories in an optimal way in terms of space and time complexity. Another major component in the project is the **onedir Function** that is responsible for returning the sizes of the files and directories given to it and to building the tree that will be later used in order to zoom in into some other directories inside the main directory passed to that onedir function. Upon checking the contents of the path passed to onedir function, the tree gets built such that if the checked item is detected as a file, the BST node will store the name of that file, its size, and mark it as not a directory; all of this will happen by calling the **insertFile Function** and passing to it the tree we are working on and the data object representing the file and if the item is a file, the number of files inside that path also gets incremented; the size of the whole tree gets updated too accordingly.

If the checked item is a directory, the BST node stores its name, its path, and marks it as a directory. Then, a new tree is dynamically created and assigned the output of **insertdirectory Function** to which the data object that contains the name of the directory and a BST node pointer are passed to it. This new tree serves as a subset of the original tree that represents the files and only the names of the directories. Each node in the original tree having the dir variable marked as true stores a pointer that points to a new tree whose root is the object in which the name of the directory and its path is stored. This tree that is a subset of the original tree is stored inside the tree variable of the data object. The tree that is preceding the sub tree by one level is also stored in the mytree variable of the data object. The preceding tree is also stored to keep track and to have pointers connecting all the directories that are embedded in one another. This whole data object is then inserted in the original tree via the **insert function.**

The **traverseASC function** is not just serving the purpose of traversing the whole tree. However, it builds the sub trees through passing their paths to the onedir function called inside it. This tweak is done like that because calling onedir recursively in the onedir function caused the stack to overflow when it gets a directory that has too many nested directories inside it; so this step cannot be done recursively and that’s why it is done inside the traverse function. This traverse function traverses the whole tree with everything inside it.

Then, there is another function that is called **TraverseChild** and as the name suggests, it only traverses one tree at a time without printing the contents of all the sub trees to the last level. Moreover, every node that’s marked as a directory in the tree passed to it is fully emptied and traversed in a text file whose name is the same as the name of the path stored in it. Thus, when the user wants to zoom in into a directory that has already been traversed, the files that have been generated will be checked to get the one requested from the zoom in and printed on the screen and modeled using the pie chart. These files will be saved in a specific directory called files and created by our code to reside in the users’ device. The files in this directory will all be deleted if the next selected directory to be traversed is not a subset of the current directory that has already been traversed; this will be achieved using the **RecursiveDelete Function**

**Graphical User Interface**

For the user to interact with the disk space analyzer, there has to be a user-friendly interface which allows navigating the disk space and is capable of visually representing some useful information to the user, such as the existing files and folders, their sizes, and their percentages of the disk space.

**The features provided to the user by the GUI are:**

1. Selecting a specific path from the existing directories in the disk to start navigation form

2. Zooming in and out from the selected starting point

3. Visual representation of the selected directory

4. Writing the path manually to the GUI

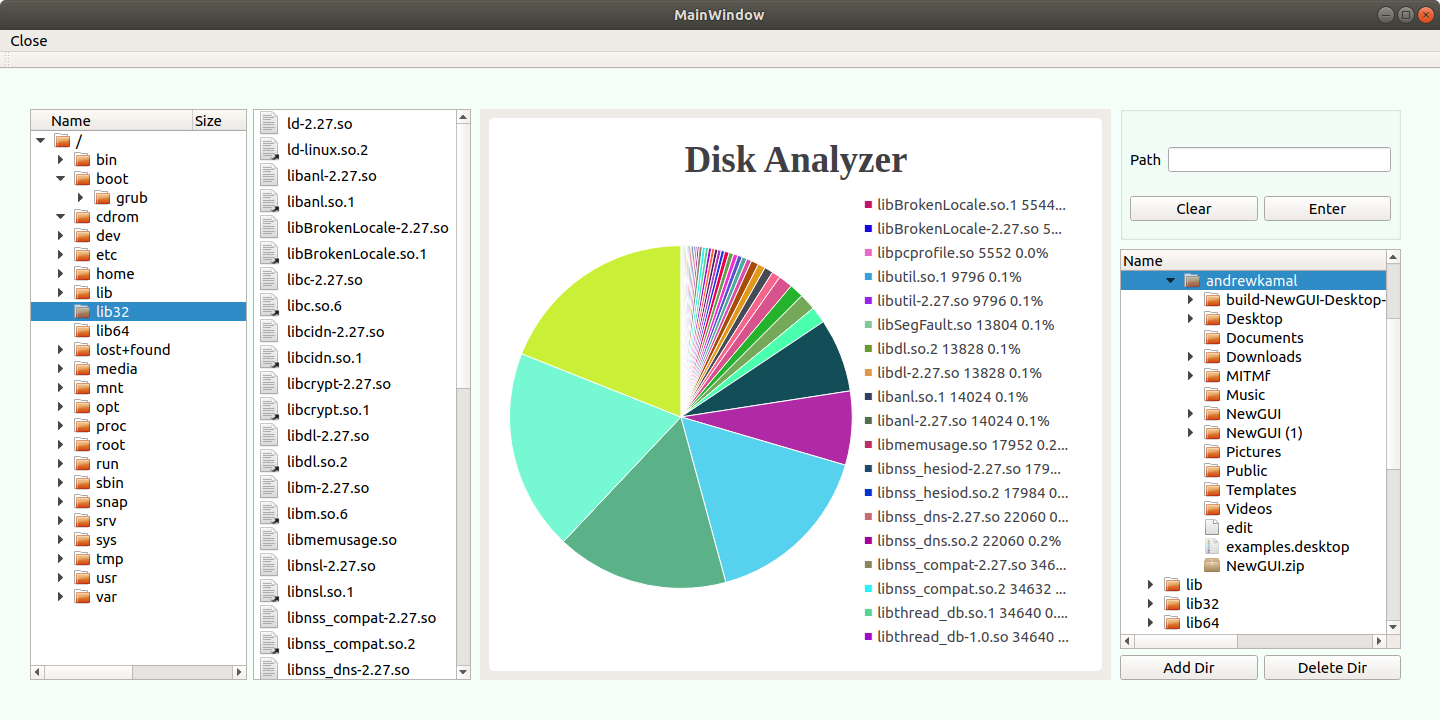
5. Adding and deleting directories from the GUI

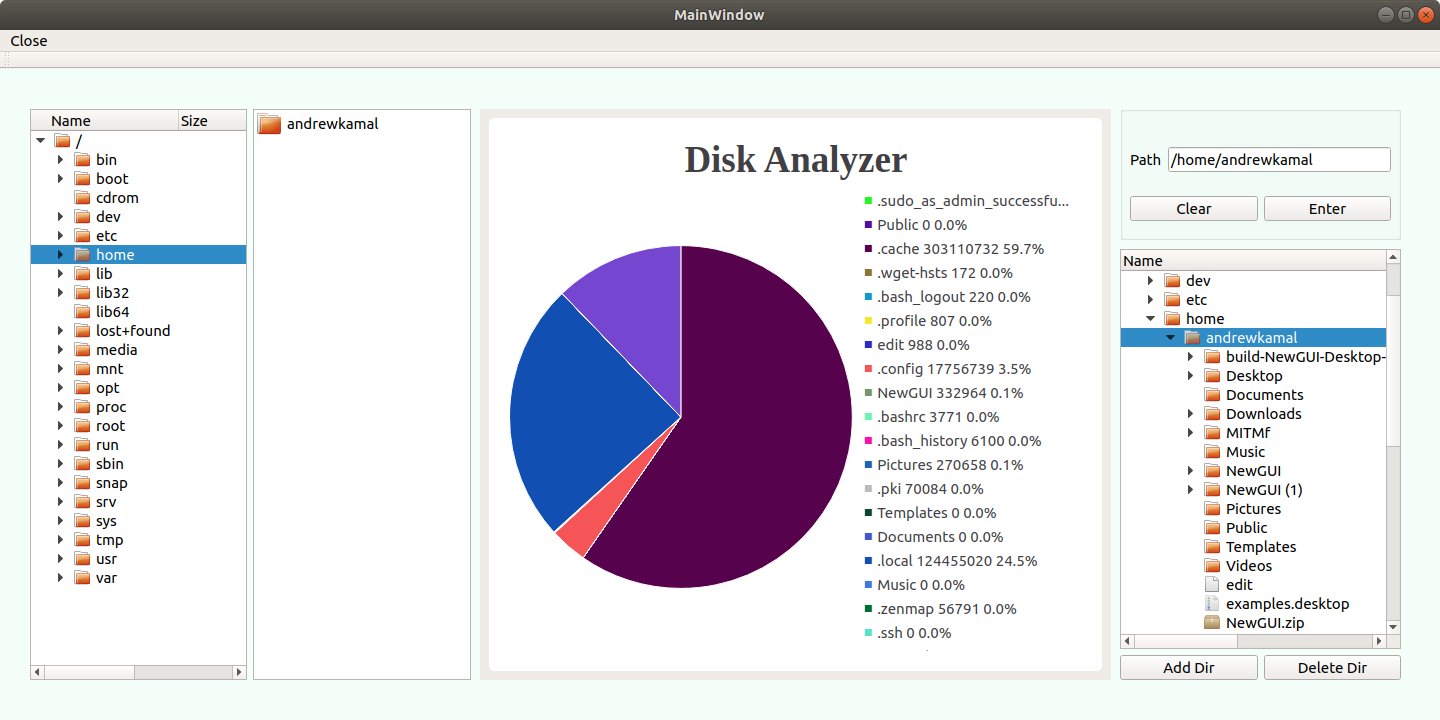
6. Displaying the total size, number of subdirectories, and number of files of the selected path

**Software used**

The GUI is designed and implemented using **Qt** that is a free and open-source widget toolkit for creating graphical user interfaces. It is also a cross-platform application that runs on various software and hardware platforms such as Linux and Windows. The reasons for choosing this software are basically its simplicity, being a free open source, and the variety of available useful toolkits and widgets which is essential to our application.

**GUI design**





**Implementation**

1. The user can select a starting path to navigate from via double clicking on a specific directory in the treeView displayed on the left side of the GUI. When the treeView is double clicked, the path to the current index is obtained. By using **QFileSystemModel Class** which provides a data model for the local filesystem, we could obtain the list of the available directories on our local filesystem, and using **QFileInfo Class**, we could obtain useful system-independent file information. The QFileInfo class has the member function absoluteFilePath() which we use to return the absolute path including the file name. The absolute path name consists of the full path and the file name, and on Unix this will always begin with the root, '/'. Then, we check if that path is available, if yes, we send that path to the function run() that is responsible for running the entire logic of our program. After that, the path is sent to createGraph() function which generates the chart to be displayed on the GUI. The previous chart is deleted, and finally the newly generated one is drawn.

2. ِZooming in and out is accomplished via a flag that checks if the currently selected path is part of the previous directory or not. The purpose of this step is to avoid redundancy in recreating an already existing, compiled text file while it already exists. If the selected path is a substring of the previous path, the flag is set to true and consequently, we go ahead to function run () to build a new tree. But, if the flag is false, we use the previously generated tree -which means that this current path is a subdirectory of the previous one. So, if we are using the previously generated tree, it means we have the capability to move in and out from the current path.

3. The visual representation of the selected directory is displayed as a Pie Chart. In order to generate the desired representation of the data, createGraph() function is implemented. The function takes the name of the file that contains the data to be displayed and returns a chartView. The idea is that each line of this file contains 2 essential data: the name of the file or directory and its size. We use 2 different vectors to push those data in, and then we fill in the slices of the chart using those 2 vectors. For drawing the pie chart, we used QtCharts/QPieSeries, QtCharts/QPieSlice, QtCharts/QChartView, and QtCharts/QLegend classes for various purposes. Those classes provided several aesthetic features to the chart, such as the colors, text, labels, percentages, alignment, size, and animation. However, in order to randomize colors of the chart, we created another generateRandom() function, which generates numbers between 0 and 255 (the RGB scale) and assigned each randomized RGB value to a slice of the chart.

4. In order to allow the user to enter the desired path to be displayed by the GUI, two buttons were used: one that takes the input path and another one to allow clearing the entered path. When the user enters the path manually, firstly, we check if that’s a valid path or not. Then, we convert the input to a path that we can use to create our graph.

The input through the message box which the user entered is of type QString so it is firstly converted to a standard string using toStdString(). That string is then passed to the member function run() which is responsible for running the entire logic of our program. Then, function utility() is applied on that string. The purpose of this function is to remove the slashes in the path name, so that it is a one string corresponding to a unique file for each directory. Then, in order to form the entire path of the directory that is going to be drawn on the graph, we need to add 2 parts to the string: the current working directory and “.txt” to access the text file in which we can find the useful data to be displayed on the chart. After that, the function **createGraph()** is called, which generates the pie chart as explained above.

Then, we need to delete the previously drawn chart including the vertical layout (that is used to select a starting path) before drawing the newly generated graph. In order to delete the previous chart, **QtAlgorithms Class** is used which includes the generic, template-based algorithms. This class has the member function **qDeleteAll()** which we used to delete the previous chart before drawing the new one, and that was done to avoid drawing multiple charts on top of each other each time the user enters or selects a new path. Then **QLayoutItem Class** is used to delete the previous vertical layout before displaying the new one. The class has the member function **takeAt()** which removes the layout item at the specified index from the layout. Finally, we draw the newly generated chart on the chartView and the vertical layout as well.

5. In order to add or delete directories from the GUI, **QDirModel Class** was used as it provides a data model for the local filesystem. A treeView of the filesystem is displayed on the GUI to allow the user to select the directory in which he wants to create a new subdirectory or the directory he wants to delete. Along with the treeView of the filesystem, two push buttons are used to allow the user to either add or delete directories.

When the add push button is clicked, a dialogue appears to the user to allow him to enter the name of the new directory. This is implemented using **QInputDialog Class** which provides a simple convenience dialog to get a single value (numbers, text, items, etc.) from the user. The member function **getText()** from this class is used to read the user input (name of the new directory to make). Then, we check if the user input string is valid or to either do nothing if invalid (empty string, for example), or go to make a new directory if valid. The member function **mkdir()** of QDirModel class is used to create a directory with the name in the parent model item. The function takes the current index from the treeView (the location at which the user wishes to make a new directory) and the name entered by the user and adds this new directory to the parent item.

When the delete push button is clicked, the GUI takes the current index in the treeView which displays the entire filesystem and does one of three actions. One, if the chosen item is invalid, the selected directory needs permission to be deleted for example, the function returns without doing anything. Two, if the selected item is a directory, **rmdir ()** is called which removes the directory corresponding to the model item index in the directory model and deletes the corresponding directory from the filesystem. Three, if the selected item is a file, **remove()** is called which removes the model item index from the directory model and deletes the corresponding file from the filesystem. The method by which the second and third options are implemented is by using the functions **fileInfo()** and **isDir()** which provide information about the underlying files and directories related to items in the model.

6. Every time the user changes its choice and the newly generated chart is displayed, the total size of this directory is printed in a message box, along with its total size in bytes and number of the internal files and directories as well.

**Conclusion**

To sum up, the described implementation of a disk analyzer tool made it possible to achieve and combine all the pros found in the previous implementations of similar tools while improving some of the issues found in them. Not only does our disk analysis tool provide visual insights about the folders and files occupying the space of a hard disk or other storage media in a very quick manner, but it also offers the user a built in tool that eases up the process of cleaning the disk; such analysis and cleanup tools will always ensure the best performance of the system that might, otherwise, get degraded because of the insufficient disk free space.

**Literature Cited**

Faisal Frafat. (2019, December 10). The 10 Best Disk Analyzer Tools for Linux System in 2020. Retrieved March 1, 2020, from <https://www.ubuntupit.com/best-disk-analyzer-tools-for-linux-system/>

Lewine, D. A. (1994). *Posix programmers guide: writing portable Unix programs with the Posix. 1 standard*. Sebastopol, CA: OReilly & Associates.

QInputDialog Class: Qt Widgets 5.14.1. (n.d.). Retrieved March 18, 2020, from <https://doc.qt.io/qt-5/qinputdialog.html>//

QDirModel Class: Qt Widgets 5.14.1. (n.d.). Retrieved March 15, 2020, from <https://doc.qt.io/qt-5/qdirmodel.html>

QFileSystemModel Class: Qt Widgets 5.14.1. (n.d.). Retrieved March 18, 2020, from <https://doc.qt.io/qt-5/qfilesystemmodel.html>

Generic Algorithms: Qt Core 5.14.1. (n.d.). Retrieved March 18, 2020, from <https://doc.qt.io/qt-5/qtalgorithms.html>

QLayoutItem Class: Qt 4.8. (n.d.). Retrieved March 19, 2020, from <https://doc.qt.io/archives/qt-4.8/qlayoutitem.html>

QFileInfo Class: Qt Core 5.14.1. (n.d.). Retrieved March 19, 2020, from <https://doc.qt.io/qt-5/qfileinfo.html>

QPieSeries Class: Qt Charts 5.14.1. (n.d.). Retrieved March 19, 2020, from <https://doc.qt.io/qt-5/qpieseries.html>

QPieSlice Class: Qt Charts 5.14.1. (n.d.). Retrieved March 20, 2020, from <https://doc.qt.io/qt-5/qpieslice.html>

QChartView Class: Qt Charts 5.14.1. (n.d.). Retrieved March 20, 2020, from <https://doc.qt.io/qt-5/qchartview.html>

QLegend Class: Qt Charts 5.14.1. (n.d.). Retrieved March 21, 2020, from <https://doc.qt.io/qt-5/qlegend.html>

https://stackoverflow.com/questions/2256945/removing-a-non-empty-directory-programmatically-in-c-or-c