Warisha Zia 23F-0534

Seema Haider 23F-0827

Nawal Tariq 23F-0776

File Driven Management system

Data Lab Project

5/11/2025

***Introduction***

The Google Drive File System is a console-based application designed to simulate a robust file management system. It leverages fundamental data structures to ensure optimized storage, retrieval, and recovery mechanisms. This system is intended to provide users with an efficient way to manage files and directories, similar to popular cloud storage services like Google Drive.

***Data Structures Used***

***1. Tree (Folder Structure)***

Purpose: Represents the hierarchical organization of directories and subdirectories.

Details:

Each folder is represented as a node in the tree.

Subfolders and files are children of these nodes.

Supports depth-first and breadth-first traversal for efficient navigation and management.

Facilitates operations like creating, deleting, and navigating through folders.

***2. Hash Table (File Metadata & Fast Lookup)***

Purpose: Stores file metadata for rapid access.

Details:

Metadata includes file size, type, creation date, and owner.

Enables quick file indexing and retrieval with O(1) complexity.

Prevents duplication and supports hashing-based conflict resolution.

**3*. Stack (Recently Deleted Files - Recycle Bin)***

Purpose: Implements LIFO (Last-In-First-Out) storage for deleted files.

Details:

Recently deleted files are stored in a stack.

Enables efficient restoration of the most recently deleted files.

Implements time-based auto-deletion of old entries to manage storage.

***4. Queue (Recent Files & Access History)***

Purpose: Stores recently accessed files in a FIFO (First-In-First-Out) manner.

Details:

Implements a Least Recently Used (LRU) cache for optimized access.

Tracks the last accessed files and prioritizes frequently used files.

The oldest file (least recently used) is removed when the queue reaches its capacity.

***5. Graph (File Sharing & User Connections)***

Purpose: Represents users as nodes and shared file relationships as edges.

Details:

Implements an adjacency list/matrix for efficient file-sharing operations.

Allows users to share files and manage access permissions.

Facilitates tracking of user interactions and file access history.

6. Linked List (File Versioning & Updates)

Purpose: Maintains a version history for each file.

Details:

Uses a doubly linked list to store multiple versions of a file.

Enables rollback to previous versions and efficient tracking of changes.

Supports version control for files, allowing users to revert to earlier states.

Functionalities by Level

***2.1 Basic Functionalities***

Folder & File Management (Tree + Hash Table):

Users can create, delete, and navigate through folders.

Each file is indexed using hashing for efficient retrieval.

File Operations (CRUD System):

Create File: Initializes a file with metadata.

Read File: Displays the content stored in a file.

Update File: Modifies the contents and stores version history.

Delete File: Moves a file to the Recycle Bin (stack).

Recycle Bin (Stack-based Recovery System): Allows retrieval of the most recently deleted file. Implements time-based auto-deletion of old entries.

Recent Files (Queue-based Access Tracking):

Tracks the last accessed files and prioritizes frequently used files.

The oldest file (least recently used) is removed when the queue reaches its capacity.

User Authentication (Graph for Login/sign in /signup/logout):

Securely manages user credentials with passwords.

Security questions are saved for password recovery.

Saves current time and date of logout for each user.

Each user is a node, and edges store login relationships (who last logged in, last logout time, etc.).

***2.2 Intermediate Functionalities***

File Search (Tree Traversal & Hashing):

Uses pre-order/post-order traversal for directory searches.

Hash table enables O(1) complexity for file lookups.

Graph-based File Sharing System:

Users (nodes) can share files using directed edges.

Implements BFS/DFS traversal for file access.

File Versioning (Linked List):

Stores multiple versions of a file in a linked list.

Allows rollback to a previous state efficiently.

Error Handling & Validation:

Prevents duplication, invalid inputs, and unauthorized access.

Implements exception handling to prevent system crashes.

***2.3 Advanced Functionalities***

Access Control & Permissions (Graph-Based User Roles):

Defines read, write, and execute permissions.

Implements user groups with hierarchical access control.

Users are nodes, and access levels are assigned based on user roles.

A directed edge between users means permission is granted.

Hierarchical control allows admins to modify access.

Compression Algorithm (Advanced Storage Optimization):

Implements run-length encoding (RLE) or dictionary-based compression.

Reduces storage requirements for large files.

Example of RLE: If a file contains AAAAABBBCC, it is stored as 5A3B2C to save space.

Dictionary-Based Compression: Common words/symbols are replaced with shorter codes to reduce file size.

Cloud Synchronization (Queue-based Background Sync):

Implements background tasks to sync local files with cloud storage.

Ensures data consistency between online and offline states.

Scalability & Optimization:

Uses balanced tree structures (AVL) for improved performance.

Implements efficient garbage collection for file system optimization.

***Conclusion***

The Google Drive File System is a comprehensive console-based application that leverages various data structures to provide efficient file management. It offers a range of functionalities from basic file operations to advanced features like access control and data compression. This system ensures optimized storage, retrieval, and recovery mechanisms, making it a robust solution for managing files in a structured manner.

By implementing this system, users can benefit from an organized and efficient way to manage their files, ensuring data integrity, security, and accessibility. The use of advanced data structures and algorithms further enhances the system's performance and scalability, making it suitable for both personal and collaborative use

***Code:***

1. #include <iostream>
2. #include <string>
3. #include <ctime>
4. #include <sstream>
5. #include <chrono>
6. #include <thread>
7. #include <cstdlib>
8. #include <iomanip>
9. #include <algorithm>
10. #include <vector>
11. #include <stack>
12. using namespace std;
13. // Color Codes as Global Variables
14. const string RED = "\033[31m";
15. const string GREEN = "\033[32m";
16. const string YELLOW = "\033[33m";
17. const string BLUE = "\033[34m";
18. const string RESET = "\033[0m";
19. // Constants
20. const size\_t MAX\_CHILDREN = 100;
21. const size\_t MAX\_FILES = 1000;
22. const size\_t RECYCLE\_BIN\_SIZE = 100;
23. const size\_t RECENT\_FILE\_LIMIT = 10;
24. const size\_t MAX\_VERSIONS = 10;
25. enum UserRole {
26. ADMIN,
27. EDITOR,
28. VIEWER
29. };
30. struct UserPermissions {
31. bool canRead;
32. bool canWrite;
33. bool canExecute;
34. UserPermissions(bool read, bool write, bool execute)
35. : canRead(read), canWrite(write), canExecute(execute) {}
36. };
37. struct UserNode {
38. string username;
39. string password;
40. string securityQuestion;
41. string securityAnswer;
42. time\_t lastLogout;
43. UserRole role;
44. UserPermissions permissions;
45. UserNode\* next;
46. UserNode(string u, string p, string q, string a, UserRole r)
47. : username(u), password(p), securityQuestion(q), securityAnswer(a), role(r),
48. permissions(getPermissionsForRole(r)), lastLogout(0), next(nullptr) {}
49. UserPermissions getPermissionsForRole(UserRole role) {
50. switch (role) {
51. case ADMIN:
52. return UserPermissions(true, true, true);
53. case EDITOR:
54. return UserPermissions(true, true, false);
55. case VIEWER:
56. return UserPermissions(true, false, false);
57. default:
58. return UserPermissions(false, false, false);
59. }
60. }
61. };
62. struct FileContent {
63. string content;
64. time\_t modifiedDate;
65. FileContent(string c = "") : content(c), modifiedDate(time(0)) {}
66. };
67. struct FileMetadata {
68. string name;
69. string type;
70. int size;
71. string owner;
72. time\_t createdDate;
73. string hash;
74. string path;
75. FileContent versions[MAX\_VERSIONS];
76. int versionCount;
77. FileMetadata() : size(0), createdDate(time(0)), versionCount(0) {}
78. };
79. struct FileNode {
80. string name;
81. bool isFolder;
82. FileNode\* children[MAX\_CHILDREN];
83. size\_t childCount;
84. FileNode\* parent;
85. FileMetadata\* metadata; // Pointer to file metadata (nullptr for folders)
86. FileNode(string n, bool f, FileNode\* p = nullptr, FileMetadata\* m = nullptr)
87. : name(n), isFolder(f), childCount(0), parent(p), metadata(m) {
88. for (size\_t i = 0; i < MAX\_CHILDREN; i++) {
89. children[i] = nullptr;
90. }
91. }
92. ~FileNode() {
93. if (!isFolder && metadata != nullptr) {
94. delete metadata;
95. }
96. }
97. };
98. struct AVLNode {
99. FileMetadata\* data;
100. AVLNode\* left;
101. AVLNode\* right;
102. int height;
103. AVLNode(FileMetadata\* data) : data(data), left(nullptr), right(nullptr), height(1) {}
104. };
105. class AVLTree {
106. private:
107. AVLNode\* root;
108. int height(AVLNode\* node) {
109. if (node == nullptr) return 0;
110. return node->height;
111. }
112. int getBalance(AVLNode\* node) {
113. if (node == nullptr) return 0;
114. return height(node->left) - height(node->right);
115. }
116. AVLNode\* rightRotate(AVLNode\* y) {
117. AVLNode\* x = y->left;
118. AVLNode\* T2 = x->right;
119. x->right = y;
120. y->left = T2;
121. y->height = max(height(y->left), height(y->right)) + 1;
122. x->height = max(height(x->left), height(x->right)) + 1;
123. return x;
124. }
125. AVLNode\* leftRotate(AVLNode\* x) {
126. AVLNode\* y = x->right;
127. AVLNode\* T2 = y->left;
128. y->left = x;
129. x->right = T2;
130. x->height = max(height(x->left), height(x->right)) + 1;
131. y->height = max(height(y->left), height(y->right)) + 1;
132. return y;
133. }
134. AVLNode\* insert(AVLNode\* node, FileMetadata\* data) {
135. if (node == nullptr) return new AVLNode(data);
136. if (data->name < node->data->name) {
137. node->left = insert(node->left, data);
138. }
139. else if (data->name > node->data->name) {
140. node->right = insert(node->right, data);
141. }
142. else {
143. return node; // Duplicate keys not allowed
144. }
145. node->height = 1 + max(height(node->left), height(node->right));
146. int balance = getBalance(node);
147. // Left Left Case
148. if (balance > 1 && data->name < node->left->data->name) {
149. return rightRotate(node);
150. }
151. // Right Right Case
152. if (balance < -1 && data->name > node->right->data->name) {
153. return leftRotate(node);
154. }
155. // Left Right Case
156. if (balance > 1 && data->name > node->left->data->name) {
157. node->left = leftRotate(node->left);
158. return rightRotate(node);
159. }
160. // Right Left Case
161. if (balance < -1 && data->name < node->right->data->name) {
162. node->right = rightRotate(node->right);
163. return leftRotate(node);
164. }
165. return node;
166. }
167. AVLNode\* minValueNode(AVLNode\* node) {
168. AVLNode\* current = node;
169. while (current->left != nullptr) {
170. current = current->left;
171. }
172. return current;
173. }
174. AVLNode\* deleteNode(AVLNode\* root, string name) {
175. if (root == nullptr) return root;
176. if (name < root->data->name) {
177. root->left = deleteNode(root->left, name);
178. }
179. else if (name > root->data->name) {
180. root->right = deleteNode(root->right, name);
181. }
182. else {
183. if ((root->left == nullptr) || (root->right == nullptr)) {
     1. AVLNode\* temp = root->left ? root->left : root->right;
     2. if (temp == nullptr) {
     3. temp = root;
     4. root = nullptr;
     5. }
     6. else {
     7. \*root = \*temp;
     8. }
     9. delete temp;
184. }
185. else {
     1. AVLNode\* temp = minValueNode(root->right);
     2. root->data = temp->data;
     3. root->right = deleteNode(root->right, temp->data->name);
186. }
187. }
188. if (root == nullptr) return root;
189. root->height = 1 + max(height(root->left), height(root->right));
190. int balance = getBalance(root);
191. // Left Left Case
192. if (balance > 1 && getBalance(root->left) >= 0) {
193. return rightRotate(root);
194. }
195. // Left Right Case
196. if (balance > 1 && getBalance(root->left) < 0) {
197. root->left = leftRotate(root->left);
198. return rightRotate(root);
199. }
200. // Right Right Case
201. if (balance < -1 && getBalance(root->right) <= 0) {
202. return leftRotate(root);
203. }
204. // Right Left Case
205. if (balance < -1 && getBalance(root->right) > 0) {
206. root->right = rightRotate(root->right);
207. return leftRotate(root);
208. }
209. return root;
210. }
211. void preOrderTraversal(AVLNode\* root) {
212. if (root != nullptr) {
213. cout << BLUE << "File: " << root->data->path << "\\" << root->data->name
     1. << " (Type: " << root->data->type << ", Size: " << root->data->size << " bytes)" << RESET << endl;
214. preOrderTraversal(root->left);
215. preOrderTraversal(root->right);
216. }
217. }
218. AVLNode\* search(AVLNode\* root, string name) {
219. if (root == nullptr) return nullptr;
220. if (root->data->name == name) {
221. return root;
222. }
223. if (root->data->name < name) {
224. return search(root->right, name);
225. }
226. return search(root->left, name);
227. }
228. public:
229. AVLTree() : root(nullptr) {}
230. void insert(FileMetadata\* data) {
231. root = insert(root, data);
232. }
233. void deleteNode(string name) {
234. root = deleteNode(root, name);
235. }
236. void preOrderTraversal() {
237. preOrderTraversal(root);
238. }
239. FileMetadata\* search(string name) {
240. AVLNode\* result = search(root, name);
241. if (result) {
242. cout << YELLOW << "File found at path: " << result->data->path << RESET << endl;
243. return result->data;
244. }
245. else {
246. cout << YELLOW << "File not found: " << name << RESET << endl;
247. return nullptr;
248. }
249. }
250. };
251. class FileSystem {
252. private:
253. FileNode\* root;
254. FileNode\* currentFolder;
255. AVLTree fileTree;
256. size\_t fileCount;
257. stack<string> recycleBin; // Changed to stack for LIFO behavior
258. string recentFiles[RECENT\_FILE\_LIMIT];
259. size\_t recentFront, recentRear, recentSize;
260. UserNode\* usersHead;
261. string formatTime(time\_t time) {
262. char buffer[80];
263. struct tm timeinfo;
264. localtime\_s(&timeinfo, &time);
265. strftime(buffer, sizeof(buffer), "%Y-%m-%d %H:%M:%S", &timeinfo);
266. return string(buffer);
267. }
268. string hashFileName(const string& fileName) {
269. unsigned long hash = 0;
270. int prime = 31;
271. for (char c : fileName) {
272. hash = hash \* prime + c;
273. }
274. stringstream ss;
275. ss << hex << hash;
276. return ss.str();
277. }
278. public:
279. FileSystem() {
280. root = new FileNode("This PC", true);
281. currentFolder = root;
282. fileCount = 0;
283. recentFront = 0;
284. recentRear = -1;
285. recentSize = 0;
286. usersHead = nullptr;
287. }
288. UserNode\* findUser(string username) {
289. UserNode\* curr = usersHead;
290. while (curr) {
291. if (curr->username == username) return curr;
292. curr = curr->next;
293. }
294. return nullptr;
295. }
296. string getCurrentPath() {
297. vector<string> pathComponents;
298. FileNode\* node = currentFolder;
299. while (node != nullptr) {
300. pathComponents.push\_back(node->name);
301. node = node->parent;
302. }
303. reverse(pathComponents.begin(), pathComponents.end());
304. string path;
305. for (size\_t i = 0; i < pathComponents.size(); i++) {
306. if (i != 0) path += "\\";
307. path += pathComponents[i];
308. }
309. return path;
310. }
311. // User management functions
312. bool signup(string username, string password, string question, string answer, UserRole role) {
313. if (findUser(username) != nullptr) {
314. cout << GREEN << "Username already exists." << RESET << endl;
315. return false;
316. }
317. UserNode\* newUser = new UserNode(username, password, question, answer, role);
318. newUser->next = usersHead;
319. usersHead = newUser;
320. cout << RED << "User signed up successfully." << RESET << endl;
321. return true;
322. }
323. bool login(string username, string password) {
324. UserNode\* user = findUser(username);
325. if (!user) {
326. cout << BLUE << "Username not found." << RESET << endl;
327. return false;
328. }
329. if (user->password == password) {
330. cout << BLUE << "Login successful." << RESET << endl;
331. return true;
332. }
333. else {
334. int choice;
335. cout << RED << "Incorrect password. Recover password?\n1. Yes\n2. No\nEnter choice: " << RESET << endl;
336. cin >> choice;
337. cin.ignore();
338. if (choice == 1) {
     1. cout << RED << "Security Question: " << user->securityQuestion << RESET << endl;
     2. string answer;
     3. cout << RED << "Enter answer: " << RESET << endl;
     4. getline(cin, answer);
     5. if (answer == user->securityAnswer) {
     6. cout << RED << "Answer correct. Enter new password: " << RESET << endl;
     7. string newPassword;
     8. getline(cin, newPassword);
     9. user->password = newPassword;
     10. cout << RED << "Password updated successfully!" << RESET << endl;
     11. return true;
     12. }
     13. else {
     14. cout << RED << "Incorrect answer. Cannot recover password." << RESET << endl;
     15. return false;
     16. }
339. }
340. cout << BLUE << "Login failed." << RESET << endl;
341. return false;
342. }
343. }
344. void logout(string username) {
345. UserNode\* user = findUser(username);
346. if (user) {
347. user->lastLogout = time(0);
348. cout << BLUE << "Logout time saved for " << username << RESET << endl;
349. }
350. else {
351. cout << BLUE << "User not found." << RESET << endl;
352. }
353. }
354. void createFolder(string folderName) {
355. if (currentFolder->childCount < MAX\_CHILDREN) {
356. currentFolder->children[currentFolder->childCount++] = new FileNode(folderName, true, currentFolder);
357. cout << BLUE << "Folder created: " << folderName << RESET << endl;
358. }
359. else {
360. cout << "Folder limit reached in current directory." << endl;
361. }
362. }
363. FileNode\* searchFolder(FileNode\* node, const string& folderName) {
364. if (node->isFolder && node->name == folderName) {
365. return node;
366. }
367. for (size\_t i = 0; i < node->childCount; ++i) {
368. FileNode\* found = searchFolder(node->children[i], folderName);
369. if (found) return found;
370. }
371. return nullptr;
372. }
373. void createFile(string fileName, FileMetadata metadata, string folderName) {
374. currentFolder = searchFolder(root, folderName); // assuming 'root' is the top-level directory
375. if (!currentFolder) {
376. cout << RED << "Folder '" << folderName << "' not found." << RESET << endl;
377. return;
378. }
379. if (currentFolder->childCount < MAX\_CHILDREN && fileCount < MAX\_FILES) {
380. FileMetadata\* fileMeta = new FileMetadata(metadata);
381. fileMeta->name = fileName;
382. fileMeta->hash = hashFileName(fileName);
383. fileMeta->path = getCurrentPath();
384. fileMeta->createdDate = time(0);
385. // Initialize first version
386. fileMeta->versions[0] = FileContent();
387. fileMeta->versionCount = 1;
388. FileNode\* newFile = new FileNode(fileName, false, currentFolder, fileMeta);
389. currentFolder->children[currentFolder->childCount++] = newFile;
390. fileTree.insert(fileMeta);
391. fileCount++;
392. cout << "File created: " << fileName << " (Hash: " << fileMeta->hash
     1. << ", Created: " << formatTime(fileMeta->createdDate) << ")" << endl;
393. }
394. else {
395. cout << "File limit reached in current directory or system file limit reached." << endl;
396. }
397. }
398. void readFile(string fileName, UserNode\* user) {
399. if (!canUserAccessFile(user, "read")) {
400. cout << RED << "Permission denied: Read access required." << RESET << endl;
401. return;
402. }
403. FileMetadata\* meta = fileTree.search(fileName);
404. if (meta && meta->versionCount > 0) {
405. cout << GREEN << "File content: " << RESET << endl;
406. cout << meta->versions[meta->versionCount - 1].content << endl;
407. cout << GREEN << "Last modified: "
     1. << formatTime(meta->versions[meta->versionCount - 1].modifiedDate) << RESET << endl;
408. }
409. else {
410. cout << RED << "File not found or has no content." << RESET << endl;
411. }
412. }
413. void updateFile(string fileName, string newContent, UserNode\* user) {
414. if (!canUserAccessFile(user, "write")) {
415. cout << RED << "Permission denied: Write access required." << RESET << endl;
416. return;
417. }
418. FileMetadata\* meta = fileTree.search(fileName);
419. if (meta) {
420. if (meta->versionCount < MAX\_VERSIONS) {
     1. meta->versions[meta->versionCount++] = FileContent(newContent);
421. }
422. else {
     1. // Shift versions to make room (remove oldest)
     2. for (size\_t j = 0; j < MAX\_VERSIONS - 1; j++) {
     3. meta->versions[j] = meta->versions[j + 1];
     4. }
     5. meta->versions[MAX\_VERSIONS - 1] = FileContent(newContent);
423. }
424. meta->size = newContent.size();
425. cout << GREEN << "File updated successfully. Version "
     1. << meta->versionCount << " created." << RESET << endl;
426. }
427. else {
428. cout << RED << "File not found." << RESET << endl;
429. }
430. }
431. void changeDirectory(string folderName) {
432. if (folderName == "..") {
433. if (currentFolder->parent) {
     1. currentFolder = currentFolder->parent;
     2. cout << BLUE << "Moved to parent folder: " << currentFolder->name << RESET << endl;
434. }
435. else {
     1. cout << BLUE << "Already at root directory." << RESET << endl;
436. }
437. return;
438. }
439. for (size\_t i = 0; i < currentFolder->childCount; i++) {
440. if (currentFolder->children[i]->isFolder && currentFolder->children[i]->name == folderName) {
     1. currentFolder = currentFolder->children[i];
     2. cout << BLUE << "Changed to folder: " << folderName << RESET << endl;
     3. return;
441. }
442. }
443. cout << GREEN << "Folder not found: " << folderName << RESET << endl;
444. }
445. void showFileVersions(string fileName) {
446. FileMetadata\* meta = fileTree.search(fileName);
447. if (meta && meta->versionCount > 0) {
448. cout << YELLOW << "Versions for " << fileName << ":" << RESET << endl;
449. for (size\_t v = 0; v < meta->versionCount; v++) {
     1. cout << "Version " << v + 1 << " ("
     2. << formatTime(meta->versions[v].modifiedDate) << "):" << endl;
     3. cout << meta->versions[v].content << endl;
450. }
451. }
452. else {
453. cout << RED << "File not found or has no versions." << RESET << endl;
454. }
455. }
456. void deleteFile(string fileName, UserNode\* user) {
457. if (!canUserAccessFile(user, "write")) {
458. cout << RED << "Permission denied: Write access required." << RESET << endl;
459. return;
460. }
461. FileMetadata\* meta = fileTree.search(fileName);
462. if (meta) {
463. if (recycleBin.size() < RECYCLE\_BIN\_SIZE) {
     1. // Store file path in recycle bin
     2. string filePath = meta->path + "\\" + fileName;
     3. recycleBin.push(filePath);
     4. // Remove from file tree
     5. fileTree.deleteNode(fileName);
     6. // Remove from folder
     7. FileNode\* folder = searchFolder(root, meta->path);
     8. if (folder) {
     9. for (size\_t i = 0; i < folder->childCount; i++) {
     10. if (!folder->children[i]->isFolder &&
         1. folder->children[i]->name == fileName) {
         2. delete folder->children[i];
         3. for (size\_t j = i; j < folder->childCount - 1; j++) {
         4. folder->children[j] = folder->children[j + 1];
         5. }
         6. folder->childCount--;
         7. fileCount--;
         8. break;
     11. }
     12. }
     13. }
     14. cout << GREEN << "File moved to recycle bin: " << fileName << RESET << endl;
464. }
465. else {
     1. cout << RED << "Recycle bin is full!" << RESET << endl;
466. }
467. }
468. else {
469. cout << RED << "File not found: " << fileName << RESET << endl;
470. }
471. }
472. void restoreFile() {
473. if (!recycleBin.empty()) {
474. string filePath = recycleBin.top();
475. recycleBin.pop();
476. // Parse file path and name
477. size\_t lastSlash = filePath.find\_last\_of('\\');
478. string fileName = filePath.substr(lastSlash + 1);
479. string folderPath = filePath.substr(0, lastSlash);
480. // Create a new file node and add it back to the current folder
481. FileMetadata\* fileMeta = new FileMetadata();
482. fileMeta->name = fileName;
483. fileMeta->path = folderPath;
484. fileMeta->createdDate = time(0);
485. FileNode\* restoredFile = new FileNode(fileName, false, currentFolder, fileMeta);
486. currentFolder->children[currentFolder->childCount++] = restoredFile;
487. fileTree.insert(fileMeta);
488. fileCount++;
489. cout << GREEN << "File restored from recycle bin: " << fileName << RESET << endl;
490. }
491. else {
492. cout << RED << "Recycle bin is empty!" << RESET << endl;
493. }
494. }
495. void searchFile(string fileName) {
496. FileMetadata\* meta = fileTree.search(fileName);
497. if (meta) {
498. cout << GREEN << "File found: " << meta->path << "\\" << meta->name << RESET << endl;
499. cout << GREEN << "Type: " << meta->type << ", Size: " << meta->size << " bytes" << endl;
500. cout << GREEN << "Created: " << formatTime(meta->createdDate) << endl;
501. cout << GREEN << "Versions: " << meta->versionCount << endl;
502. }
503. }
504. void shareFile(string fromUser, string toUser, string fileName) {
505. UserNode\* from = findUser(fromUser);
506. UserNode\* to = findUser(toUser);
507. if (from && to) {
508. cout << YELLOW << "File '" << fileName << "' shared from " << fromUser << " to " << toUser << RESET << endl;
509. }
510. else {
511. cout << BLUE << "One or both users not found." << RESET << endl;
512. }
513. }
514. void displayCurrentDirectory() {
515. cout << BLUE << "Current directory: " << getCurrentPath() << RESET << endl;
516. cout << BLUE << "Contents:" << RESET << endl;
517. // Display folders first
518. for (size\_t i = 0; i < currentFolder->childCount; i++) {
519. if (currentFolder->children[i]->isFolder) {
     1. cout << "[Folder] " << currentFolder->children[i]->name << endl;
520. }
521. }
522. // Then display files with metadata
523. for (size\_t i = 0; i < currentFolder->childCount; i++) {
524. if (!currentFolder->children[i]->isFolder) {
     1. FileMetadata\* meta = currentFolder->children[i]->metadata;
     2. if (meta) {
     3. cout << "[File] " << currentFolder->children[i]->name
     4. << " (Size: " << meta->size << " bytes, "
     5. << "Created: " << formatTime(meta->createdDate) << ")" << endl;
     6. }
     7. else {
     8. cout << "[File] " << currentFolder->children[i]->name << endl;
     9. }
525. }
526. }
527. }
528. void displayFiles() {
529. cout << GREEN << "Files in the system:" << endl;
530. fileTree.preOrderTraversal();
531. }
532. bool canUserAccessFile(UserNode\* user, string permissionType) {
533. if (!user) return false;
534. if (permissionType == "read") {
535. return user->permissions.canRead;
536. }
537. else if (permissionType == "write") {
538. return user->permissions.canWrite;
539. }
540. else if (permissionType == "execute") {
541. return user->permissions.canExecute;
542. }
543. return false;
544. }
545. void modifyUserRole(UserNode\* user, UserRole newRole) {
546. if (user) {
547. user->role = newRole;
548. user->permissions = user->getPermissionsForRole(newRole);
549. cout << "User role updated to: " << newRole << endl;
550. }
551. }
552. string runLengthEncode(string str) {
553. string result;
554. int count = 1;
555. for (size\_t i = 1; i < str.size(); i++) {
556. if (str[i] == str[i - 1]) {
     1. count++;
557. }
558. else {
     1. result += str[i - 1] + to\_string(count);
     2. count = 1;
559. }
560. }
561. result += str.back() + to\_string(count);
562. return result;
563. }
564. string runLengthDecode(string str) {
565. string result;
566. for (size\_t i = 0; i < str.size(); i += 2) {
567. char ch = str[i];
568. int count = str[i + 1] - '0';
569. result.append(count, ch);
570. }
571. return result;
572. }
573. };
574. int main() {
575. cout << YELLOW << "\t\t\t\t======================DATA PROJECT BY=======================" << RESET << endl;
576. cout << GREEN << "\t\t\t\t| |" << RESET << endl;
577. cout << GREEN << "\t\t\t\t| WARISHA |" << RESET << endl;
578. cout << GREEN << "\t\t\t\t| SEEMA |" << RESET << endl;
579. cout << GREEN << "\t\t\t\t| NAWAL |" << RESET << endl;
580. cout << YELLOW << "\t\t\t\t============================================================" << RESET << endl;
581. this\_thread::sleep\_for(chrono::seconds(10));
582. system("CLS");
583. FileSystem fs;
584. int choice;
585. string username, password, question, answer, folderName, fileName, from, to, content, fName;
586. FileMetadata fileMeta;
587. UserNode\* currentUser = nullptr;
588. while (true) {
589. cout << GREEN << "\n\t\t\t\t\t==== File System Menu ====\n" << RESET << endl;
590. cout << YELLOW << "\t\t\t\t\t1. Signup\n\t\t\t\t\t2. Login\n\t\t\t\t\t3. Create Folder\n\t\t\t\t\t4. Create File\n" << RESET;
591. cout << YELLOW << "\t\t\t\t\t5. Read File\n\t\t\t\t\t6. Update File\n\t\t\t\t\t7. Show File Versions\n" << RESET;
592. cout << YELLOW << "\t\t\t\t\t8. Delete File\n\t\t\t\t\t9. Restore File\n\t\t\t\t\t10. Change Directory\n" << RESET;
593. cout << YELLOW << "\t\t\t\t\t11. Display Current Directory\n\t\t\t\t\t12. Search File\n" << RESET;
594. cout << YELLOW << "\t\t\t\t\t13. Share File\n\t\t\t\t\t14. Run-Length Encode\n\t\t\t\t\t15. Run-Length Decode\n" << RESET;
595. cout << YELLOW << "\t\t\t\t\t16. Logout\n\t\t\t\t\t17. Display Files\n\t\t\t\t\t18. Modify User Role\n\t\t\t\t\t0. Exit\n\t\t\t\t\tChoice: " << RESET;
596. cin >> choice;
597. cin.ignore();
598. if (choice == 0) break;
599. switch (choice) {
600. case 1:
601. {
602. string username, password, question, answer;
603. int roleChoice;
604. UserRole role;
605. cout << RED << "Username: "; getline(cin, username);
606. cout << "Password: "; getline(cin, password);
607. cout << "Security Question: "; getline(cin, question);
608. cout << "Answer: "; getline(cin, answer);
609. cout << "Role (0: Admin, 1: Editor, 2: Viewer): ";
610. cin >> roleChoice;
611. cin.ignore();
612. switch (roleChoice) {
613. case 0:
     1. role = ADMIN;
     2. break;
614. case 1:
     1. role = EDITOR;
     2. break;
615. case 2:
     1. role = VIEWER;
     2. break;
616. default:
     1. cout << "Invalid role choice." << endl;
     2. continue;
617. }
618. fs.signup(username, password, question, answer, role);
619. break;
620. }
621. case 2:
622. cout << GREEN << "Username: "; getline(cin, username);
623. cout << "Password: "; getline(cin, password);
624. if (fs.login(username, password)) {
     1. currentUser = fs.findUser(username);
     2. // Set owner for files created by this user
     3. fileMeta.owner = username;
625. }
626. break;
627. case 3:
628. if (currentUser) {
     1. cout << YELLOW << "Folder Name: "; getline(cin, folderName);
     2. fs.createFolder(folderName);
629. }
630. else {
     1. cout << "You're not logged in. Please try later!" << endl;
631. }
632. break;
633. case 4:
634. if (currentUser) {
     1. cout << RED << "Folder name: "; getline(cin, fName);
     2. cout << "File Name: "; getline(cin, fileName);
     3. cout << "File Type: "; getline(cin, fileMeta.type);
     4. cout << "Initial Content: "; getline(cin, content);
     5. fileMeta.size = content.size();
     6. fs.createFile(fileName, fileMeta, fName);
     7. fs.updateFile(fileName, content, currentUser); // Set initial content
635. }
636. else {
     1. cout << "You're not logged in. Please try later!" << endl;
637. }
638. break;
639. case 5:
640. cout << GREEN << "File Name: "; getline(cin, fileName);
641. fs.readFile(fileName, currentUser);
642. break;
643. case 6:
644. cout << YELLOW << "File Name: "; getline(cin, fileName);
645. cout << "New Content: "; getline(cin, content);
646. fs.updateFile(fileName, content, currentUser);
647. break;
648. case 7:
649. cout << RED << "File Name: "; getline(cin, fileName);
650. fs.showFileVersions(fileName);
651. break;
652. case 8:
653. cout << GREEN << "File Name to Delete: "; getline(cin, fileName);
654. fs.deleteFile(fileName, currentUser);
655. break;
656. case 9:
657. fs.restoreFile();
658. break;
659. case 10:
660. cout << YELLOW << "Folder Name (or '..' for parent): "; getline(cin, folderName);
661. fs.changeDirectory(folderName);
662. break;
663. case 11:
664. fs.displayCurrentDirectory();
665. break;
666. case 12:
667. cout << RED << "File Name to Search: "; getline(cin, fileName);
668. fs.searchFile(fileName);
669. break;
670. case 13:
671. cout << GREEN << "From User: "; getline(cin, from);
672. cout << "To User: "; getline(cin, to);
673. cout << "File Name: "; getline(cin, fileName);
674. fs.shareFile(from, to, fileName);
675. break;
676. case 14:
677. cout << YELLOW << "Enter string to encode: ";
678. getline(cin, fileName);
679. cout << "Encoded: " << fs.runLengthEncode(fileName) << endl;
680. break;
681. case 15:
682. cout << RED << "Enter string to decode: ";
683. getline(cin, fileName);
684. cout << "Decoded: " << fs.runLengthDecode(fileName) << endl;
685. break;
686. case 16:
687. cout << GREEN << "Username: "; getline(cin, username);
688. fs.logout(username);
689. currentUser = nullptr;
690. break;
691. case 17:
692. fs.displayFiles();
693. break;
694. case 18:
695. if (currentUser) {
     1. cout << GREEN << "Username: "; getline(cin, username);
     2. UserNode\* user = fs.findUser(username);
     3. if (user) {
     4. cout << "New Role (0: Admin, 1: Editor, 2: Viewer): ";
     5. int newRoleChoice;
     6. cin >> newRoleChoice;
     7. cin.ignore();
     8. UserRole newRole;
     9. switch (newRoleChoice) {
     10. case 0:
     11. newRole = ADMIN;
     12. break;
     13. case 1:
     14. newRole = EDITOR;
     15. break;
     16. case 2:
     17. newRole = VIEWER;
     18. break;
     19. default:
     20. cout << "Invalid role choice." << endl;
     21. continue;
     22. }
     23. fs.modifyUserRole(user, newRole);
     24. }
     25. else {
     26. cout << "User not found." << endl;
     27. }
696. }
697. else {
     1. cout << "You're not logged in. Please try later!" << endl;
698. }
699. break;
700. default:
701. cout << RED << "Invalid choice." << endl;
702. }
703. }
704. cout << GREEN << "Exiting program." << endl;
705. cout << RED << "Exiting program." << endl;
706. cout << BLUE << "Exiting program." << endl;
707. cout << YELLOW << "Exiting program." << endl;
708. system("pause");
709. return 0;
710. }

# Results:



























































