**Proiect Programarea Calculatoarelor și Limbaje de Programare 2**

Compresie/decompresie de fișiere în C++

Profesor coordonator: Student:  
Neculoiu Giorgian Trofinenco Adrian-Răzvan

Olteanu Gabriela

**Alegerea temei**

Am ales tema „Compresor/Depresor de date” pentru că mi s-a părut interesantă și utilă în același timp. Trăim într-o perioadă în care lucrăm zilnic cu cantități tot mai mari de date – fie că vorbim de fișiere text, imagini sau aplicații – iar ideea de a economisi spațiu și timp prin comprimare mi s-a părut foarte practică.

În plus, am vrut să înțeleg mai bine cum funcționează, în spate, algoritmii care fac posibilă micșorarea fișierelor, fără să pierdem informații.

**Compresia și decompresia de date**

Comprimarea fișierelor este procesul prin care datele sunt transformate într-o formă mai compactă, astfel încât să ocupe mai puțin spațiu pe disc sau să poată fi transmise mai rapid. Scopul principal este optimizarea stocării și a transferului de informații.

Există două tipuri de compresie:

**Fără pierderi** (lossless) – datele originale pot fi recuperate 100%, exact cum au fost inițial. Se folosește pentru fișiere text, documente, arhive etc.

**Cu pierderi** (lossy) – anumite informații sunt eliminate definitiv pentru a obține o compresie mai mare. Se folosește la imagini, video și audio (ex: JPEG, MP3).

Decompresia este procesul invers, prin care fișierele comprimate sunt transformate înapoi la forma lor inițială.

Algoritmii de compresie analizează structura datelor și caută tipare repetitive sau redundante. Printre cei mai cunoscuți algoritmi se numără:

* Run-Length Encoding (RLE) – înlocuiește secvențele repetate cu o valoare și un contor
* Huffman Coding – folosește coduri binare mai scurte pentru caracterele frecvente
* LZ77 / LZW – bazează compresia pe secvențe detectate anterior în fluxul de date

Prin acest proiect, se dorește demonstrat modul în care funcționează compresia fără pierderi, oferind atât funcționalitate de compresie (compresor), cât și de decompresie (depresor), folosind concepte de bază din programare C++ și algoritmică.

**Algoritmi folosiți**

Pentru realizarea compresorului și depresorului de date, am implementat o succesiune de algoritmi care lucrează împreună pentru a obține o rată cât mai bună de compresie, fără pierderea informației. Mai jos sunt prezentați, în ordine, pașii principali:

### **1. Suffix Array**

Primul pas a fost construirea unui suffix array, o structură care conține toate sufixele unui șir, sortate lexicografic. Acest pas este necesar pentru a putea aplica eficient transformarea Burrows-Wheeler, deoarece ne permite să reordonăm caracterele șirului de intrare într-un mod care evidențiază secvențele repetitive.

### **2. Burrows-Wheeler Transform (BWT)**

Transformarea Burrows-Wheeler reorganizează șirul de caractere astfel încât caracterele similare să fie grupate împreună. Deși transformarea nu reduce dimensiunea fișierului, pregătește datele pentru pașii următori, care pot exploata mai eficient repetițiile.

### **3. Move-To-Front Transform (MTF)**

După aplicarea BWT, se folosește transformarea Move-To-Front, care convertește șirul rezultat într-un șir de indici. Acești indici sunt în general mici atunci când apar repetiții, ceea ce face datele rezultate mai potrivite pentru compresie prin Run-Length Encoding și Huffman.

### **4. Run-Length Encoding (RLE)**

RLE este aplicat pentru a comprima secvențele de caractere (sau indici) identice prin înlocuirea lor cu un singur simbol și un contor. Această metodă este foarte eficientă în cazul datelor cu multe repetiții, cum sunt cele rezultate după BWT și MTF.

### **5. Huffman Encoding**

Ultimul pas al compresiei este codarea Huffman, un algoritm clasic de compresie fără pierderi care folosește coduri binare de lungimi variabile pentru a reprezenta simbolurile, în funcție de frecvența apariției acestora. Simbolurile frecvente primesc coduri mai scurte, ceea ce contribuie la o compresie eficientă.

## **Descrierea funcționalității aplicației**

## Aplicația realizată este un program de tip consolă care permite utilizatorului să aleagă între două acțiuni principale: compresia sau decompresia unui fișier text.

## La rulare, aplicația solicită utilizatorului să introducă calea către fișierul text pe care dorește să îl proceseze. După aceea, utilizatorul poate alege între următoarele opțiuni:

### 1. Compresie

## Dacă se alege opțiunea de compresie, aplicația va aplica, în ordine, următorii algoritmi asupra conținutului fișierului:

## Suffix Array (pentru Burrows-Wheeler)

## Burrows-Wheeler Transform

## Move-To-Front

## Run-Length Encoding

## Huffman Encoding

## La final, rezultatul este scris într-un fișier comprimat. Pentru a marca faptul că fișierul a fost creat de această aplicație, la începutul fișierului comprimat se adaugă un identificator special: „UTCB”.

### 2. Decompresie

## În cazul în care utilizatorul alege decompresia, aplicația va verifica mai întâi dacă fișierul este valid, adică dacă începe cu semnătura „UTCB”. Această verificare previne încercarea de a decompresa fișiere care nu au fost comprimate cu această aplicație.

## Dacă fișierul este valid, se aplică pașii inversi ai fiecărui algoritm, în ordine inversă:

## Huffman Decoding

## Run-Length Decoding

## Move-To-Front Invers

## Inversa Burrows-Wheeler

## Reconstrucția textului original

## **Structura proiectului**

Proiectul a fost organizat modular, fiind împărțit în mai multe fișiere sursă de tip **.cpp** și **.h**, fiecare corespunzând unui algoritm principal din fluxul de compresie/decompresie. Această structurare a avut ca scop separarea logică a componentelor, o mai bună organizare a codului și facilitarea mentenanței sau extinderii proiectului în viitor.

Fiecare modul (algoritm) este implementat într-un fișier dedicat, care conține:

* Funcțiile necesare pentru **compresie**
* Funcțiile corespunzătoare pentru **decompresie**
* Structuri de date și utilitare locale, specifice fiecărui algoritm

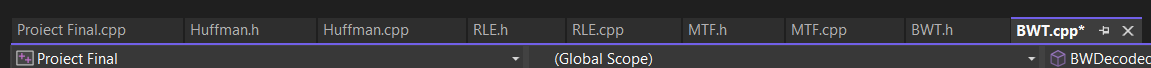
### 

### 

### **Exemple de fișieree**

* BWT.h/BWT.cpp – Burrows-Wheeler Transform, inversa sa și algoritmul pentru obținerea Suffix Array-ului
* MTF.h/MTF.cpp – Move-To-Front și invers
* RLE.h/RLE.cpp – Run-Length Encoding și decodarea RLE
* Huffman.h/Huffman.cpp – Codarea și decodarea Huffman

Fișierul principal (main.cpp) orchestrează apelurile către fiecare algoritm, în ordinea corectă, în funcție de operația dorită (compresie sau decompresie).

Această organizare modulară permite testarea individuală a fiecărei componente și oferă claritate asupra structurii generale a proiectului.

## **Structuri de date folosite**

În realizarea acestui proiect am utilizat atât structuri de date standard din biblioteca STL, cât și structuri personalizate, necesare implementării eficiente a algoritmilor de compresie și decompresie.

### **Structuri utilizate:**

* **std::vector** – folosit în majoritatea algoritmilor pentru stocarea și manipularea secvențelor de caractere, indici sau rezultate intermediare.
* **std::list** – utilizată în algoritmul **Move-To-Front**, pentru manipulări rapide de tip inserare/extragere din interiorul listei.  
  **std::pair<int, int>** – folosit pentru a stoca perechi de valori (ex: în construcția suffix array-ului), unde a fost nevoie de sortare sau comparare personalizată.
* **std::unordered\_map** – utilizat în codarea și decodarea **Huffman** și pentru maparea frecvențelor caracterelor sau a codurilor binare. Pentru cazuri speciale, am implementat funcții de hash personalizate pentru pair<int, int>.

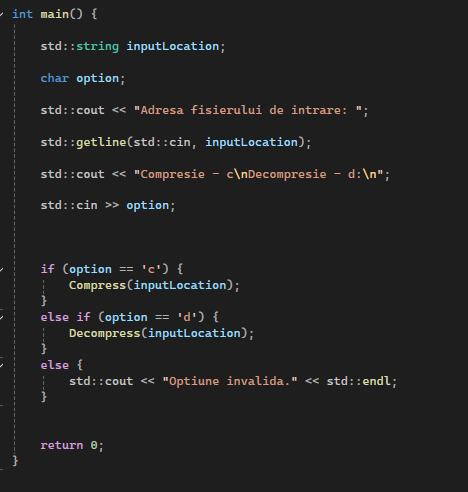
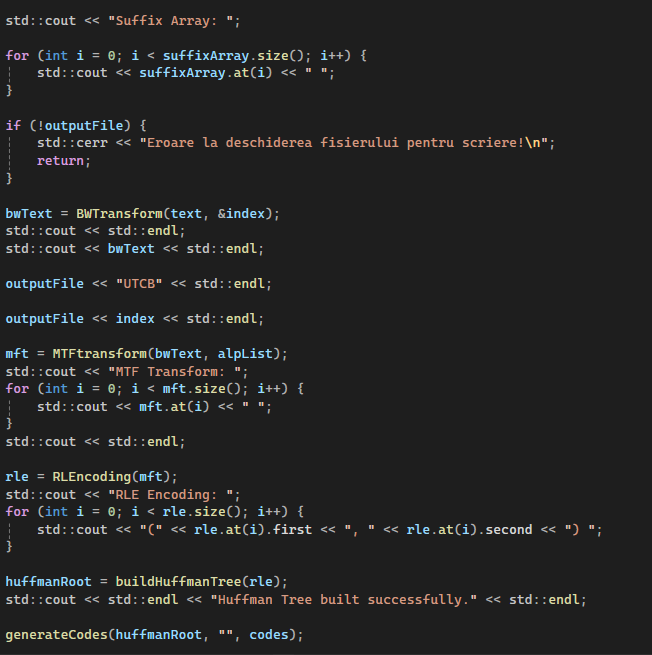
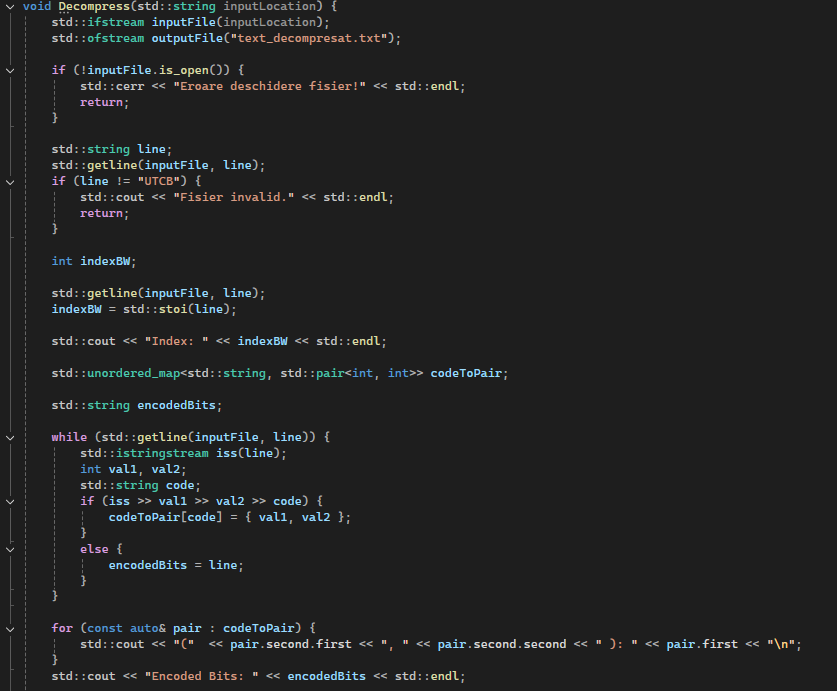
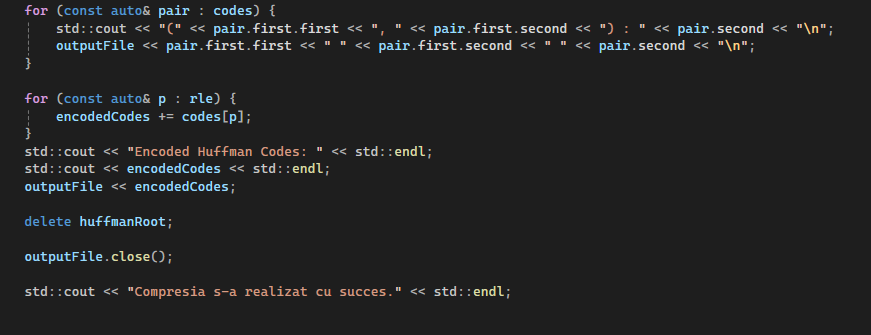
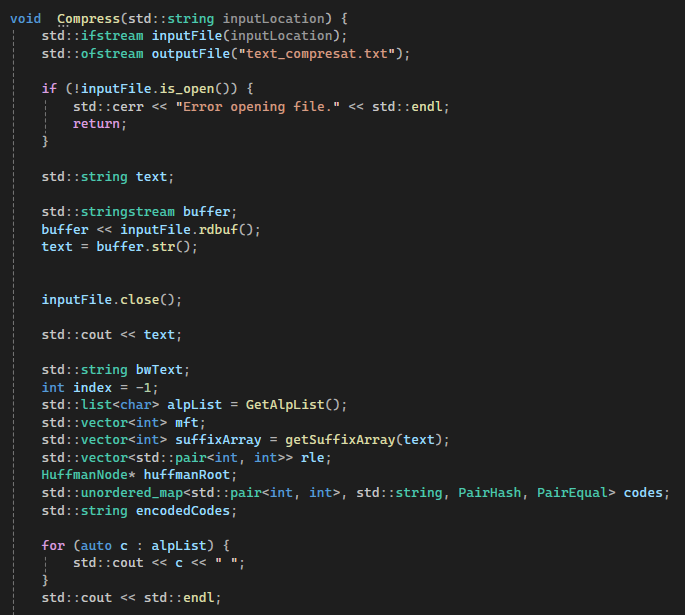
### **Structuri personalizate:**

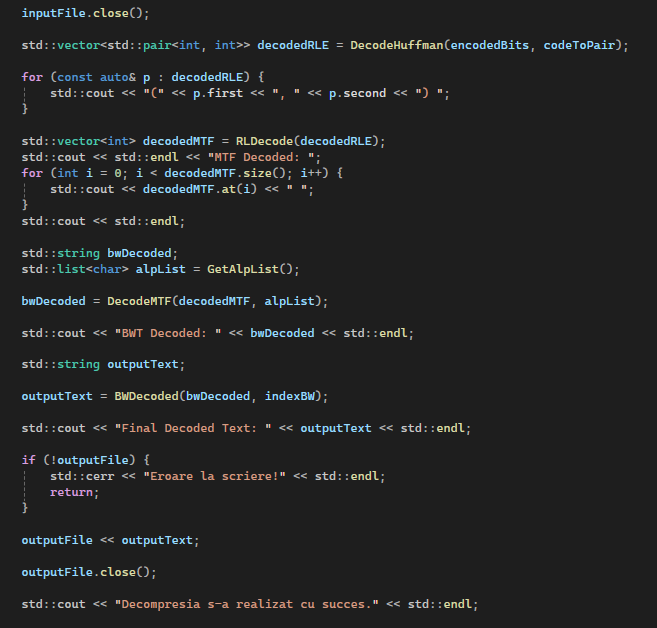
* **Structuri de comparare** – pentru a putea sorta sau compara eficient elemente de tip pair<int, int> în anumite etape ale algoritmilor, în special la construirea suffix array-ului.
* **Funcții de hash custom** – implementate pentru a permite utilizarea de pair<int, int> ca cheie într-un unordered\_map, lucru care nu este acceptat nativ de STL.
* **struct HuffmanNode** – o structură creată pentru construirea arborelui Huffman, care conține informații despre caracter, frecvență și pointeri către copiii stâng și drept ai nodului. Această structură a fost esențială în construirea arborelui binar folosit pentru generarea codurilor Huffman.

**Fișierul main.cpp**

Fișierul main.cpp are rolul de punct de intrare al aplicației și se ocupă de gestionarea interacțiunii cu utilizatorul, precum și de apelarea funcțiilor de compresie și decompresie în funcție de opțiunea selectată.

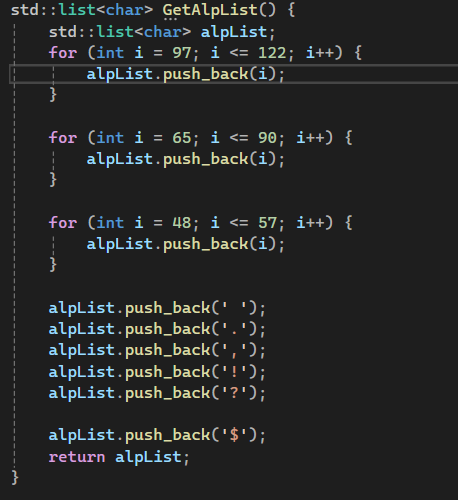
Funcționalități principale:

* **Citirea fișierului de intrare**: utilizatorul introduce de la tastatură calea către un fișier text.
* **Selectarea opțiunii**: după validarea fișierului, utilizatorul alege între:
  + 1 – Compresie  
    2 – Decompresie
* **Apelul funcțiilor corespunzătoare**: în funcție de alegerea utilizatorului, se apelează:
  + Compress() – funcție ce aplică lanțul de algoritmi de compresie.
  + Depress() – funcție care verifică dacă fișierul este valid (prin semnătura „UTCB”) și aplică decompresia.  
    



Funcții auxiliare:

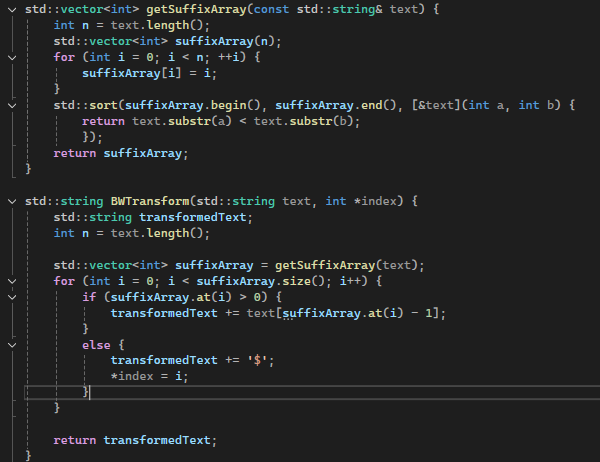
* **std::list<char> getAlphabet()** – funcție care returnează alfabetul folosit în transformarea Move-To-Front. Această funcție este importantă pentru inițializarea corectă a listei cu caracterele posibile, în ordinea dorită.

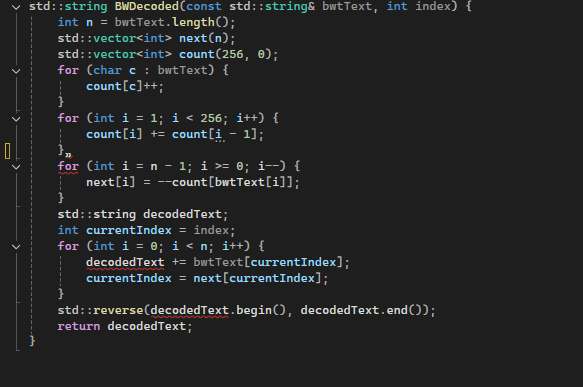


**Fișierul BWT.cpp – Burrows-Wheeler Transform**

Fișierul BWT.cpp conține implementarea transformării Burrows-Wheeler (BWT), un pas esențial în procesul de compresie, și a inversării acesteia pentru decompresie.

Funcții principale:

* getSuffixArray()  
   Construiește un vector cu toți indicii sufixelor unui șir, ordonați lexicografic. Este folosit pentru aplicarea BWT.
* BWTransform()  
   Aplică transformarea Burrows-Wheeler asupra unui șir dat. Returnează șirul transformat și reține poziția sufixului original (index) pentru a putea fi refăcut ulterior.
* BWDecoded()  
   Reface șirul original pe baza rezultatului BWT și a poziției sufixului original. Folosește un vector next construit din frecvențele caracterelor pentru a reordona corect simbolurile.

****

**Fișierul MTF.cpp – Move-To-Front Transform**

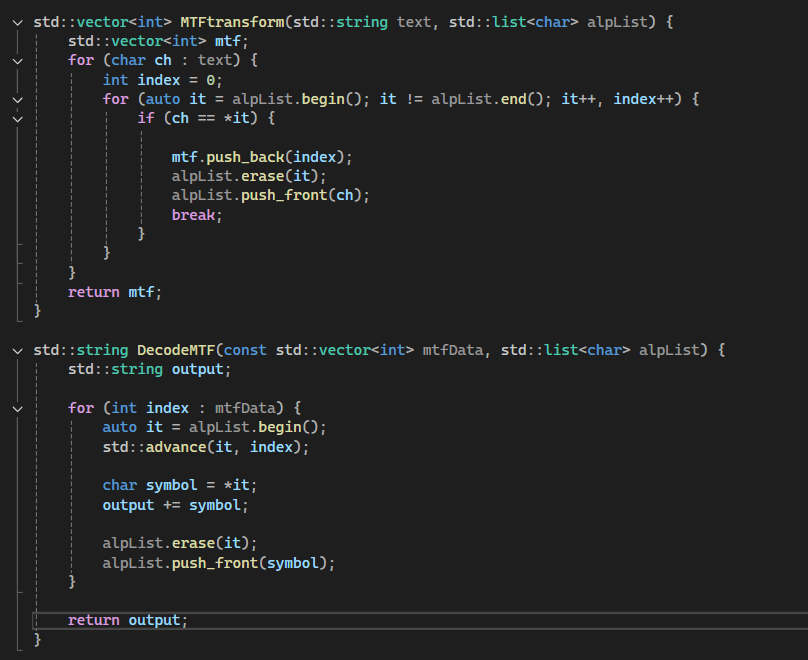
Fișierul MTF.cpp implementează transformarea Move-To-Front (MTF) și inversa acesteia, pași esențiali în procesul de compresie și decompresie, în special după aplicarea Burrows-Wheeler Transform.

Funcții principale:

* MTFtransform()  
   Primește un șir de caractere și o listă alpList (alfabetul inițial). Pentru fiecare caracter, găsește poziția sa în listă, salvează această poziție în vectorul de ieșire, apoi mută caracterul în fața listei (pentru a reflecta recența apariției).
* DecodeMTF()  
   Primește vectorul de poziții generat de MTFtransform și reconstruiește șirul original. Accesează simbolul aflat la indexul respectiv din listă, îl adaugă la rezultat, apoi îl mută în față.

Structuri folosite:

* std::list<char> – folosită pentru a păstra alfabetul într-o formă flexibilă, care permite inserarea/mutarea eficientă a caracterelor.
* std::vector<int> – stochează pozițiile simbolurilor în urma transformării MTF.

****

**Fișierul RLE.cpp – Run-Length Encoding**

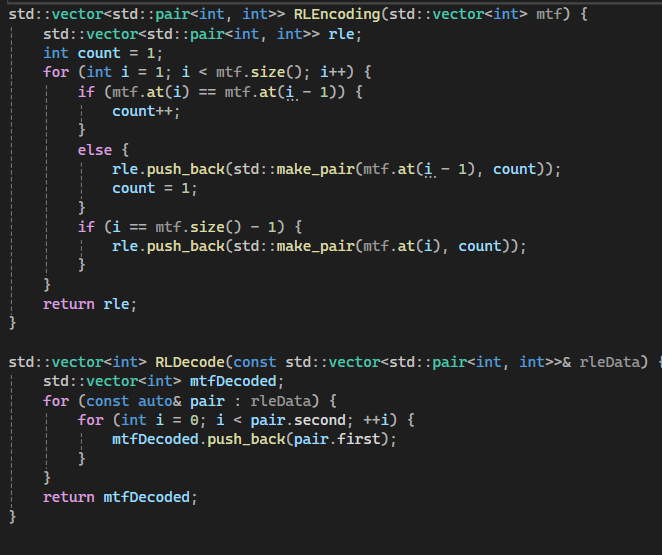
Fișierul RLE.cpp conține implementarea algoritmului Run-Length Encoding (RLE) și a decodării acestuia. Este un pas important după aplicarea Move-To-Front, deoarece optimizează stocarea secvențelor repetitive, în special a valorii 0, care apare frecvent după MTF.

Funcții principale:

* RLEncoding()  
   Primește un vector de valori întregi (ieșirea de la MTF) și returnează un vector de perechi (valoare, frecvență), unde fiecare pereche reprezintă o valoare repetată și numărul de apariții consecutive.
* RLDecode()  
   Primește un vector de perechi și reconstruiește vectorul inițial extinzând fiecare pereche de n ori valoarea respectivă.

Structuri folosite:

* std::vector<int> – pentru vectorul inițial (MTF) și rezultatul decodării.
* std::vector<std::pair<int, int>> – pentru stocarea valorilor comprimate sub formă de perechi.

****

**Fișierul Huffman.cpp – Codificare și decodificare Huffman**

Fișierul Huffman.cpp conține implementarea completă a algoritmului Huffman, utilizat ca ultim pas în compresia datelor pentru a obține o reprezentare binară optimă a informației. Codarea Huffman este extrem de eficientă deoarece atribuie coduri mai scurte simbolurilor care apar frecvent și coduri mai lungi celor rare.

Clase și structuri:

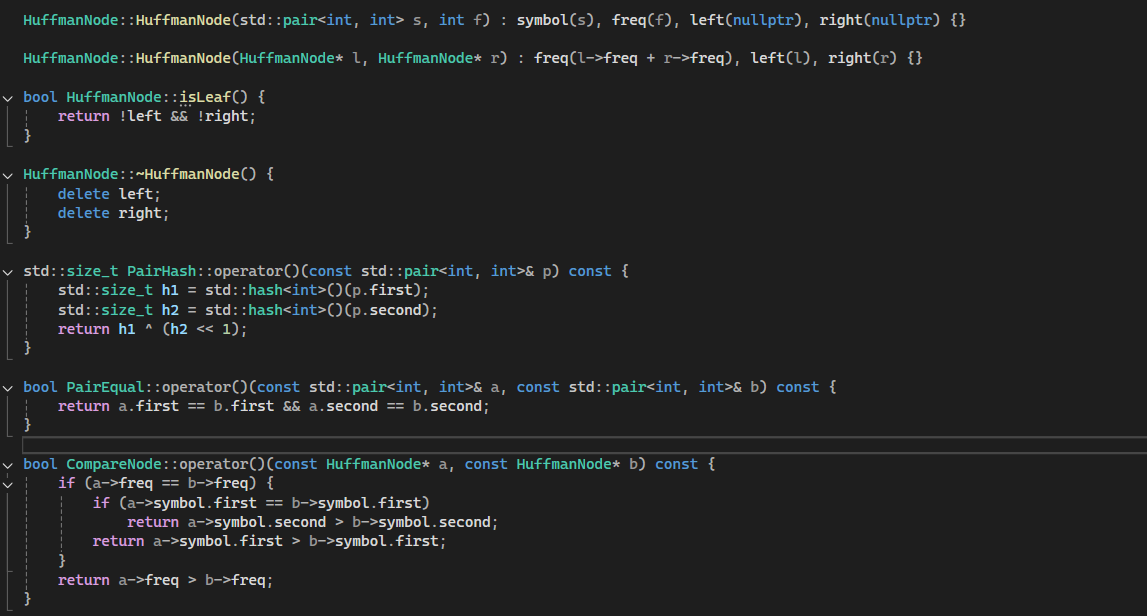
* HuffmanNode  
   Reprezintă un nod din arborele Huffman. Poate fi o frunză (care conține un simbol – în acest caz un pair<int, int> din RLE) sau un nod intern. Are frecvența totală a subarborelui și legături către copii stânga și dreapta.  
   – Constructori pentru frunze și noduri interne  
   – Funcție isLeaf() pentru a verifica dacă nodul este frunză  
   – Destructor pentru a elibera memoria recursiv
* PairHash și PairEqual  
   Sunt functori personalizați necesari pentru a folosi std::pair<int, int> ca cheie în unordered\_map. PairHash generează un hash din cele două valori ale perechii, iar PairEqual compară două perechi.
* CompareNode  
   Comparator pentru priority\_queue, astfel încât nodurile cu frecvență mai mică să fie prioritizate la construcția arborelui.

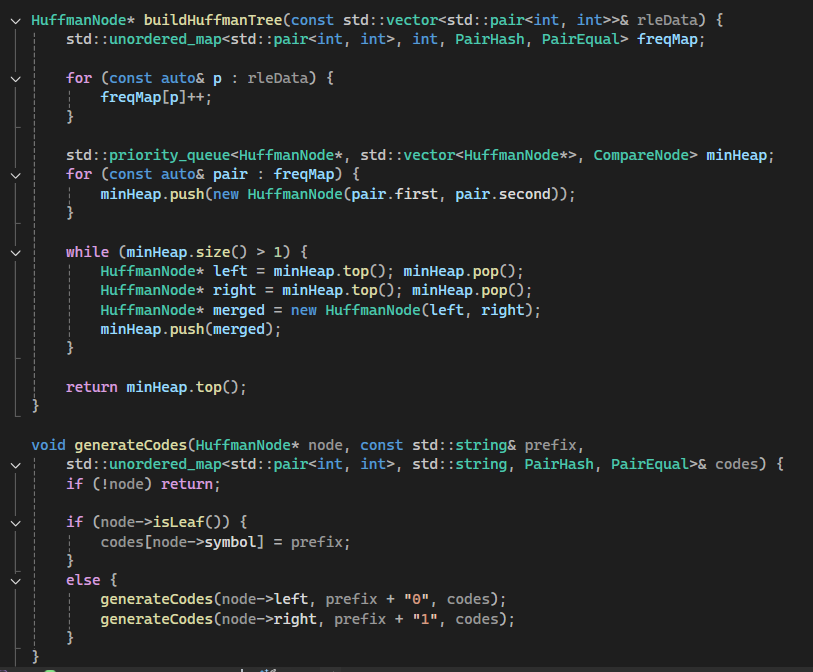
Funcții principale:

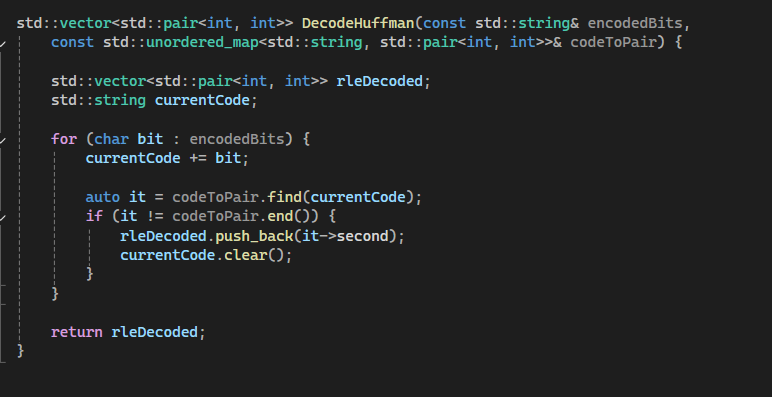
* buildHuffmanTree()  
   Primește datele RLE (perechi de valori și frecvențe) și construiește arborele Huffman.  
   – Creează un unordered\_map pentru frecvențe  
   – Adaugă toate frunzele într-un priority\_queue  
   – Conectează nodurile între ele până când rămâne un singur nod: rădăcina arborelui
* generateCodes()  
   Parcurge arborele Huffman și generează codurile binare pentru fiecare simbol. Folosește recursiv prefixul acumulat pentru a construi codul binar al fiecărui simbol și îl salvează într-un unordered\_map.
* DecodeHuffman()  
   Primește un șir de biți (std::string) și un map care leagă codurile binare de perechile originale (int, int). Reconstruiește secvența RLE originală parcurgând șirul bit cu bit și identificând codurile valide.

Structuri folosite:

* std::unordered\_map<std::pair<int, int>, int, PairHash, PairEqual> – pentru frecvențe
* std::priority\_queue – pentru construcția arborelui
* std::unordered\_map<std::pair<int, int>, std::string> – pentru coduri Huffman
* std::unordered\_map<std::string, std::pair<int, int>> – pentru decodare
* std::vector<std::pair<int, int>> – pentru datele RLE

****

****

****