# Introducere în Blockchain
# Arbori Merkle

Iulian Oleniuc

## 1 Introducere în blockchain

Conceptul *blockchain* a fost introdus în 2008 de către Satoshi Nakamoto, pentru a servi ca un registru public de tranzacții pentru criptomoneda *Bitcoin*. Blockchainul este un sistem *descentralizat*.

### 1.1 Registrul de tranzacții

Toate tranzacțiile dintre utilizatorii Bitcoin efectuate vreodată sunt stocate într-un *ledger* (registru) public de tranzacții. Acest registru *este*, în mod intrinsec, criptomoneda în sine, deoarece, având acces la un registru valid, putem determina cu exactitate suma de bitcoini deținută în momentul curent de către fiecare utilizator în parte.

| tranzacții | | | balanțe | | |
|---|---|---|---|---|---|
| | $A$ | 300 | 300 | 0 | 0 |
| | $B$ | 200 | 300 | 200 | 0 |
| | $C$ | 500 | 300 | 200 | 500 |
| $A$ | $B$ | 90 | 210 | 290 | 500 |
| $A$ | $C$ | 50 | 160 | 290 | 550 |
| $B$ | $C$ | 30 | 160 | 260 | 580 |
| $B$ | $A$ | 70 | 230 | 190 | 580 |

În tabelul de mai sus am reprezentat un exemplu de registru, unde fiecare tranzacție (formată din expeditor, destinatar și suma trimisă) este însoțită de balanțele celor trei utilizatori ($A$, $B$ și $C$) imediat după efectuarea ei.

Observăm că, pentru ca balanțele să nu fie niciodată negative, este necesar ca unele tranzacții să *nu* aibă expeditor. În cazul de față este vorba de primele trei. Rolul acestor tranzacții speciale este de a introduce criptomonedă în sistem, aspect la care vom reveni mai târziu.

## 1.2 Nevoia de semnare a tranzacțiilor

Trebuie să ne asigurăm că autoritatea care administrează registrul nu poate falsifica tranzacții. Cu alte cuvinte, avem nevoie de un mecanism prin care să putem verifica dacă tranzacția $(A, B, x)$ a fost efectuată într-adevăr de către utilizatorul $A$.

În acest scop, tupla $(A, B, x)$ va fi *semnată digital* de către $A$. În primul rând, această semnătură este construită pe baza *cheii private* a lui $A$, deci validarea ei ne garantează că emițătorul tranzacției respective a fost cu adevărat $A$. În al doilea rând, semnătura este construită în așa fel încât schimbarea unui singur bit din tranzacție să producă o semnătură complet diferită, *indistingibilă* de cea precedentă. Verificarea semnăturii poate fi efectuată de către oricine folosind *cheia publică* a lui $A$.

Chiar dacă semnătura ne garantează că valorile $A$, $B$ și $x$ sunt cele corecte, asta nu împiedică pe nimeni să includă de două sau mai multe ori aceeași tranzacție în registru. Pentru a preveni această problemă, semnătura digitală a tranzacției este construită și peste *indexul* tranzacției respective în registru.

## 1.3 Nevoia de descentralizare

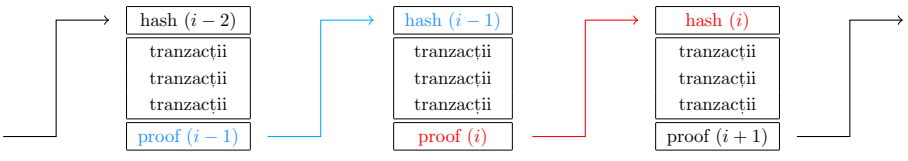
Până acum, ne-am asigurat că tranzacțiile în sine pot fi validate, dar cum ne asigurăm că și *registrul* este valid? Dacă acesta ar fi controlat de către o autoritate centrală, aceasta ar putea, de exemplu, să omită intenționat adăugarea unei anumite tranzacții la blockchain, fără ca vreun utilizator să poată măcar observa asta, cu excepția expeditorului.

Soluția oferită de tehnologia blockchain este *descentralizarea* registrului – fiecare utilizator va reține propria sa copie. În consecință, de fiecare dată când un nod vrea să facă o tranzacție, acesta o va trimite (împreună cu semnătura sa) tuturor celorlalte noduri din rețea, pentru ca fiecare dintre ele să-și poată actualiza copia propriului registru.

## 1.4 Împărțirea registrului în blocuri

Noua problemă pe care o întâmpinăm este cum ne dăm seama dacă putem avea încredere într-o tranzacție pe care tocmai am primit-o sau nu? Mai întâi însă, vom prezenta o structură mai detaliată a registrului din spatele Bitcoin.

Registrul este de fapt împărțit în *blocuri*, fiecare conținând în medie câte 2 400 de tranzacții. Aceste blocuri sunt *înlănțuite* – de unde vine și numele de *blockchain* – în așa fel încât alterarea unui singur bloc ar invalida automat toate blocurile care îi succed.



Fiecare bloc conține, pe lângă tranzacțiile aferente, două valori speciale. Prima este hash-ul blocului precedent, iar a doua este numită *proof-of-work* și influențează hash-ul blocului curent (care va fi inclus în blocul următor). Despre conceptul de *hashing* vom vorbi mai pe larg în secțiunea 2.

## 1.5 Proof-of-work

Calculul valorii proof-of-work asociate blocului curent trebuie să fie o problemă grea din punct de vedere computațional. Din acest motiv, problema aleasă este următoarea. Să se determine o valoare $x$ astfel încât hash-ul blocului curent (calculat peste hash-ul precedent, tranzacțiile din bloc și $x$) să înceapă cu $n$ zerouri. Vom afla puțin mai târziu cum este ales acest $n$, dar pentru moment vom presupune că $n = 30$.

Cum funcția hash aleasă, mai precis SHA-256, produce un output *pseudo-random*, probabilitatea ca acesta să înceapă cu cele 30 de zerouri dorite este $1/2^{30}$. Prin urmare, este nevoie de aproximativ $2^{30} \approx 10^9$ încercări pentru a obține un proof-of-work valid.

## 1.6 Identificarea tranzacțiilor false

Recapitulând, toate nodurile din rețea ascultă după tranzacții de la celelalte noduri și construiesc pe baza lor noi blocuri. Când un nod finalizează un bloc, îl trimite tuturor celorlalte noduri, pentru a-l adăuga la blockchainul personal.

Revenind la problema observată în secțiunea 1.4, să presupunem că Alice vrea să îl mintă pe Bob că i-a trimis acestuia 100 de bitcoini. Cu alte cuvinte, Alice trebuie să procedeze în așa fel încât Bob să aibă o versiune de blockchain diferită de toate celelalte, ea fiind singura care conține tranzacția falsă.

Pentru aceasta, Alice ar trebui să muncească încontinuu, menținând o versiune de blockchain diferită, special pentru Bob. Între timp, Bob continuă să primească blocuri și de la celelalte noduri, astfel că, începând din momentul în care a primit tranzacția falsă de la Alice, blockchainul lui Bob a fost bifurcat:

$$B_1 \to B_2 \to \cdots \to B_i \begin{cases} \to B'_{i+1} \to B'_{i+2} \\ \to B_{i+1} \to B_{i+2} \to B_{i+3} \to B_{i+4} \to B_{i+5} \end{cases}$$

Ideea de bază din spatele conceptului de proof-of-work este că un nod va avea întotdeauna încredere în versiunea *mai lungă* a blockchainului, deoarece în construirea ei a fost pusă mai multă muncă (prin calculul valorii proof-of-work a fiecărui bloc), ceea ce înseamnă că mai multe noduri au fost implicate și deci au încredere în ea.

Prin urmare, Alice ar trebui să valideze primul bloc înaintea tuturor celorlalte noduri din rețea, ceea ce nu este imposibil. Însă, va trebui să procedeze la fel și cu fiecare dintre următoarele blocuri. Matematic, este imposibil ca, după un număr relativ mic de blocuri, lungimea versiunii de blockchain a lui Alice s-o mai poată ajunge din urmă pe cea a blockchainului construit de restul rețelei.

## 1.7 Mineritul de bitcoini

Toate nodurile rețelei Bitcoin pot emite tranzacții, însă nu toate aleg să și *proceseze* tranzacții. Cele care totuși o fac se numesc adesea *mineri*. Minerii construiesc noi blocuri, folosindu-și resursele computaționale pentru a calcula valori proof-of-work. Ei sunt astfel cei care țin rețeaua în viață.

Rolul lor este atât de important încât sunt răsplătiți în mod intrinsec pentru munca depusă. Mai exact, primul miner care găsește un proof-of-work pentru blocul curent primește un *reward*. Acesta se manifestă sub forma unei tranzacții care se atașează la începutul blocului și conform căreia minerul primește (de nicăieri) o anumită sumă de bitcoini. Acesta este totodată unicul mod prin care se introduce criptomonedă în sistem.

Inițial, valoarea rewardului era de 50 de bitcoini. Însă, la fiecare 210 000 de blocuri minate, adică aproximativ o dată la patru ani, acesta se înjumătățește, procesul fiind cunoscut sub numele de *halving*. Astfel, se garantează că niciodată nu vor exista mai mult de

$$210\,000 \cdot (50 + 25 + 12.5 + \cdots) = 210\,000\,000$$

de bitcoini în circulație.

## 1.8 Taxe de tranzacție

Revenind la calculul proof-of-work, mai devreme spuneam că se caută o valoare ce face ca hash-ul blocului să înceapă cu un anumit număr de zerouri. Cu cât numărul este mai mare, cu atât timpul necesar unui miner pentru a obține un proof-of-work valid devine și el mai mare. Ei bine, acest număr de zerouri este schimbat periodic, în funcție de numărul de mineri din rețea, astfel încât minarea unui nou bloc să dureze aproximativ 10 minute.

Totodată, numărul de tranzacții dintr-un bloc este limitat la aproximativ 2 400. Astfel, dacă sunt efectuate foarte multe tranzacții simultan, poate dura destul de mult până când toate vor fi adăugate la blockchain. Astfel, pentru a motiva minerii să-i adauge cât mai repede tranzacția la blockchain, expeditorul poate atașa la aceasta o *taxă de tranzacție* – o sumă arbitrară de bitcoini ce va ajunge la primul miner care reușește să creeze un bloc conținând tranzacția în cauză.

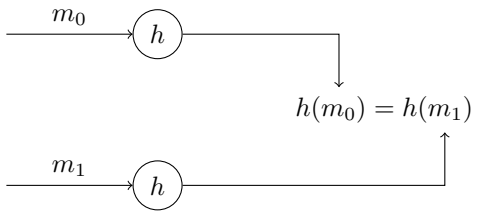
# 2 Funcții hash criptografice

O funcție *hash* (criptografică) primește ca input un șir de biți de lungime arbitrară și returnează un șir de biți de lungime fixă (cum ar fi 128 de biți). Hash-ul unui mesaj se mai numește și *digest* (rezumat). Funcțiile hash trebuie să fie ușor de calculat, adică, formal, o funcție hash trebuie să poată fi calculată printr-un algoritm determinist polinomial.

În criptografie, funcțiile hash se folosesc, printre altele, pentru semnarea digitală a mesajelor și pentru identificarea fișierelor în rețelele peer-to-peer. De obicei, când vorbim despre funcții hash, ne referim de fapt la funcțiile hash *rezistente la coliziuni*.

## 2.1 Funcții hash rezistente la coliziuni

În contextul unei funcții hash $h : X \to Z$, o *coliziunie* reprezintă o pereche de mesaje $(m_0, m_1) \in X^2$ cu proprietatea că $h(m_0) = h(m_1)$. Funcția $h$ este *rezistentă la coliziuni* dacă niciun adversar nu poate găsi pentru $h$ o coliziune $(m_0, m_1)$ într-un timp rezonabil (polinomial), decât cu probabilitate neglijabilă.




Intuitiv, pentru ca o funcție hash să fie rezistentă la coliziuni, trebuie ca outputul său să fie pseudo-random. Nu trebuie să fie aleatoriu – $h$ este deterministă – dar trebuie să *pară* aleatoriu. Adică, oricare ar fi două mesaje $m_0$ și $m_1$, dacă primim un rezumat $h_b$, unde $b \in \{0, 1\}$, trebuie să ne fie imposibil să determinăm dacă $b = 0$ sau $b = 1$ atât timp cât nu cunoaștem funcția $h$. Un alt mod de a privi această proprietate este că, dacă modificăm chiar și un singur bit din mesaj, hash-ul acestuia va arăta complet diferit față de cel inițial.

## 2.2 Paradoxul zilei de naștere

Paradoxul zilei de naștere ne spune că avem nevoie de doar 23 de oameni aleși aleatoriu pentru a fi siguri că probabilitatea ca doi dintre ei să fie născuți în aceeași zi a anului este cel puțin 1/2:

$$1 - \underbrace{\frac{365}{365} \cdot \frac{364}{365} \cdots \frac{343}{365}}_{23} \geq \frac{1}{2}$$

Morala este că, chiar dacă o funcție hash produce outputuri pseudo-aleatorii, trebuie să ne asigurăm că, în plus, codomeniul ei este suficient de mare. Altfel, un atacator poate găsi o coliziune relativ repede calculând pur și simplu hash-uri pentru inputuri random.

# 3 Arbori Merkle

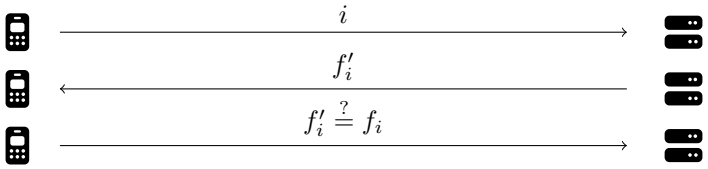
Pentru a introduce conceptul de *arbore Merkle*, vom porni de la următoarea problemă, a cărei aplicație în blockchain o vom analiza puțin mai târziu.

## 3.1 Problema inițială

Lui Mihai îi place să facă fotografii. Este atât de pasionat de arta fotografiei încât și-a umplut deja memoria telefonului cu poze efectuate pe acesta. Pentru a-și putea continua pasiunea, Mihai a decis să recurgă la un serviciu de stocare în cloud a fișierelor, cum ar fi Google Photos.

În acest sens, Mihai și-a eliberat memoria telefonului, urcând în cloud cele $n$ fotografii de pe acesta, $\langle f_1, f_2, \ldots, f_n \rangle$, și ștergându-le după aceea de pe telefon. Mai târziu, Mihai vrea să descarce din cloud fotografia $f_i$, pentru a retrăi momentul surprins în aceasta.

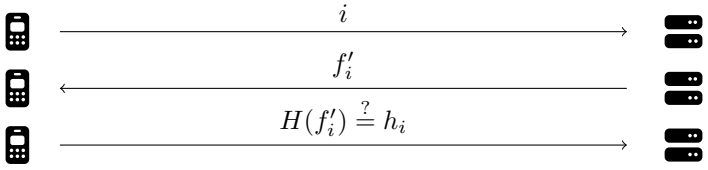
Întrebarea pe care trebuie să ne-o punem este următoarea. Cum se poate asigura Mihai că poza descărcată este într-adevăr cea pe care a urcat-o în cloud? Cu alte cuvinte, cum poate fi sigur că niciun pixel din ea nu a fost alterat, în mod intenționat sau accidental?




## 3.2 Soluția naivă

O persoană care deține un set de cunoștințe elementare despre criptografie se poate gândi la următoarea soluție. Înainte de orice schimb de fișiere, fixăm de comun acord cu serverul o funcție hash, desigur rezistentă la coliziuni, $H$.

După ce urcăm cele $n$ fișiere în cloud, în loc să le ștergem complet, vom reține pentru fiecare fișier $f_i$ hash-ul său $h_i \triangleq H(f_i)$. Astfel, tot ce avem de făcut atunci când descărcăm un fișier $f_i'$ este să verificăm dacă hash-ul său coincide cu cel stocat local, adică dacă $H(f_i') = h_i$. În caz negativ, deducem că $f_i' \neq f_i$, deci că fișierul $f_i$ a fost alterat.



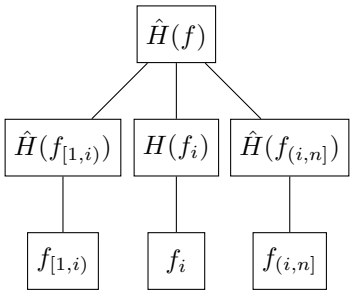

Dezavantajul acestei soluții este că, din moment ce Mihai reține câte un hash pentru fiecare fișier, telefonul său va folosi $\mathcal{O}(n)$ spațiu. Desigur, $h_i$ ocupă mult mai puțin spațiu decât $f_i$, deci constanta ascunsă de acest $\mathcal{O}(n)$ este foarte mică. Cu toate acestea, putem reduce inclusiv *ordinul* de complexitate!

## 3.3 Intuiție

Vom încerca să găsim o soluție care necesită doar $\mathcal{O}(1)$ memorie pe telefonul lui Mihai. În consecință, Mihai va reține un hash construit cumva peste *întregul* șir $f \triangleq \langle f_1, f_2, \ldots, f_n \rangle$. Să notăm cu $\hat{H}$ extensia lui $H$ la șiruri. Vom reveni în curând la definiția concretă a lui $\hat{H}$.

Pentru a verifica integritatea fișierului $f'_i$ descărcat, Mihai îi va cere serverului niște *rezultate parțiale* care să îi permită să recalculeze $\hat{H}(f)$ pe baza lor și a lui $f_i$. Dacă valoarea $\hat{H}(f')$, unde $f' \triangleq f_{[1,i)} \parallel \langle f'_i \rangle \parallel f_{(i,n]}$,[1] obținută din $f'_i$ și rezultatele respective, coincide cu $\hat{H}(f)$, concluzionăm că, într-adevăr, $f'_i = f_i$.

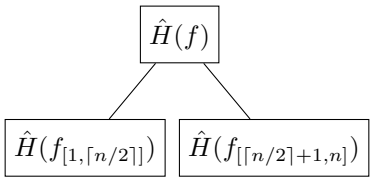
Spre exemplu, dacă funcția $\hat{H}$ respectă *proprietatea magică* prin care $\hat{H}(f) = \hat{H}(f_{[1,i)}) \oplus H(f_i) \oplus \hat{H}(f_{(i,n]}), \forall i \in \{1, 2, \ldots, n\}$, atunci este suficient să cerem serverului valorile $\hat{H}(f_{[1,i)})$ și $\hat{H}(f_{(i,n]})$.




Întâmpinăm însă două probleme. În primul rând, este greu, dacă nu chiar imposibil, să alegem o funcție $\hat{H}$ cu proprietatea de mai sus. În al doilea rând, serverul ar pierde mult timp pentru a calcula de fiecare dată cele două valori parțiale. Totodată, este impractic să *precalculăm* $\hat{H}$ pentru fiecare subsecvență posibilă a lui $f$, deoarece numărul lor este $\mathcal{O}(n^2)$.

## 3.4 Structura arborelui

Păstrăm ideea de a defini $\hat{H}(f)$ în funcție de o (singură!) partiție a lui $f$ în subsecvențe. Vom alege ca această partiție să fie formată pur și simplu din prima și respectiv din a doua jumătate a lui $f$. Formal, cele două subsecvențe vor fi $f_{[1,\lceil n/2 \rceil]}$ și $f_{[\lceil n/2 \rceil + 1, n]}$.

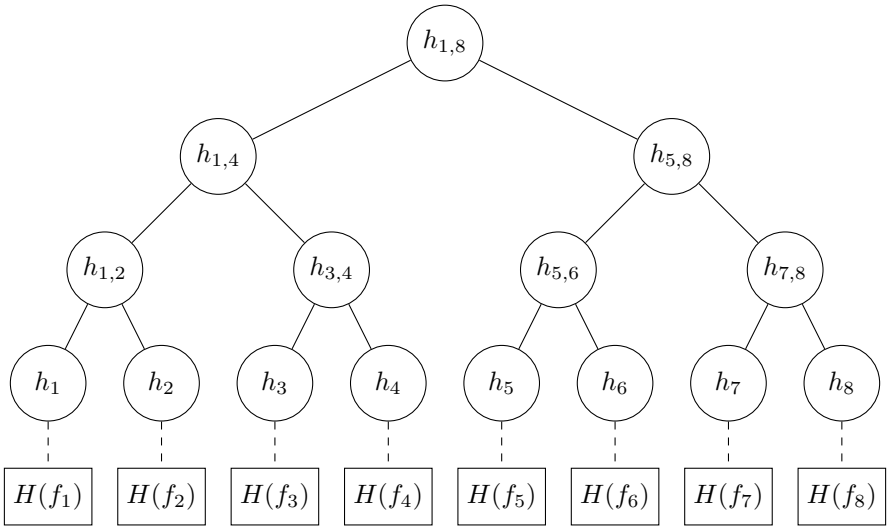



[1] Aici, notația $\parallel$ se referă la concatenarea șirurilor. Spre exemplu, $\langle 6 \rangle \parallel \langle 1, 8 \rangle = \langle 6, 1, 8 \rangle$. Apoi, notația $f_I$, unde $I$ este un interval, se referă la șirul format din elementele $f_i$, unde $i$ ia valori pe rând din mulțimea $I \cap \mathbb{N}$.

Pentru a simplifica discuția, ne vom axa doar pe cazul în care $n$ este o putere a lui 2. De asemenea, mai introducem o notație, și anume extensia lui $H$ la două fișiere: $\tilde{H}(x, y) \triangleq H(x \parallel \texttt{\#} \parallel y)$,[2] unde `#` este un caracter delimitator. Definim funcția $\hat{H}$ astfel:

$$\hat{H}(f) = \begin{cases} \tilde{H}(\hat{H}(f_{[1,n/2]}), \hat{H}(f_{[n/2+1,n]})) & \text{dacă } n > 1 \\ H(f_1) & \text{dacă } n = 1. \end{cases}$$

Tocmai am obținut o formulă recursivă pentru calculul hash-ului peste $f$. Pentru a ne referi mai ușor la hash-urile subsecvențelor șirului inițial $f$, introducem o ultimă notație, și anume $h_{l,r} \triangleq \hat{H}(f_{[l,r]})$. Calculul recursiv al lui $\hat{H}(f)$ induce un arbore a cărui structură este ilustrată mai jos (pentru $n = 8$). Acest arbore poartă numele de *arbore Merkle*.



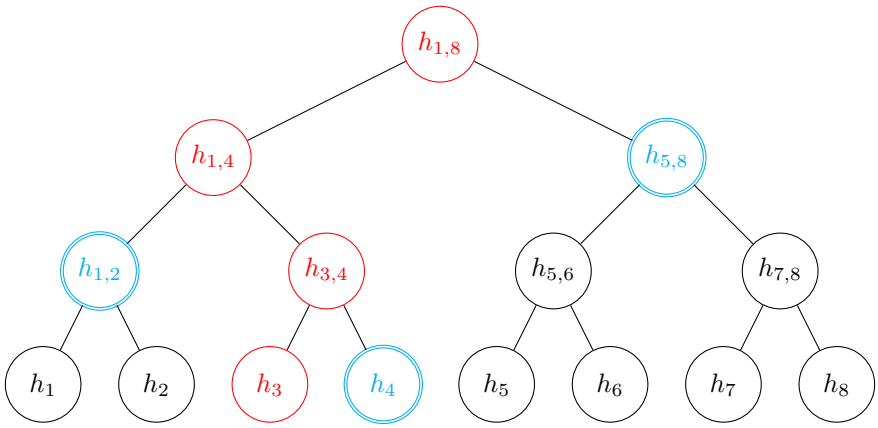

## 3.5 Dovezi Merkle

Vom analiza, în particular, cazul $(n, i) = (8, 3)$. Recapitulând, la început, Mihai și serverul construiesc în mod independent arborele Merkle corespunzător șirului de fișiere $f$. Mihai reține doar rădăcina arborelui, adică valoarea $h_{1,8}$, în timp ce serverul reține *întregul* arbore.

Apoi, Mihai primește de la server fișierul $f'_3$ și vrea să verifice dacă acesta este identic cu cel original, $f_3$. În acest scop, va trebui să recalculeze valoarea $h_{1,8}$, dar pornind cu $f'_3$ în loc de $f_3$.

Observăm că singurele noduri afectate de alterarea lui $f_3$ sunt cele marcate cu roșu în desenul de mai jos. Acestea formează de fapt tocmai drumul (unic) de la rădăcină la frunza $h_3$.

[2] Aici, notația $\parallel$ se referă la concatenarea șirurilor de caractere (sau de *bytes*). Spre exemplu, `ab` $\parallel$ `#cd` = `ab#cd`.




Prin urmare, pentru a recalcula rădăcina $h'_{1,8}$, nu este necesar să recalculăm tot arborele, ci doar nodurile roșii. Frunza $h'_3$ se calculează ușor. Ea se obține pur și simplu aplicând funcția $H$ pe fișierul $f'_3$. Pentru celelalte noduri însă, avem nevoie și de niște rezultate parțiale, care sunt stocate doar pe server.

De exemplu, $h'_{1,8}$ se obține din $h'_{1,4}$ și $h_{5,8}$. Cerând valoarea $h_{5,8}$ de la server, suntem practic scutiți de recalcularea a *jumătate* de arbore! Așadar, vestea bună este că numărul nodurilor ce trebuie cerute serverului (cele colorate cu albastru mai sus) este foarte mic. Vom reveni la el în secțiunea următoare.

Ei bine, mulțimea de noduri albastre necesare pentru a valida integritatea unui fișier $f'_i$ poartă numele de *dovadă Merkle*. Concret, calculele aferente exemplului de mai sus sunt:

$$\begin{aligned} h'_3 &\leftarrow H(f'_3) \\ h'_{3,4} &\leftarrow \tilde{H}(h'_3, h_4) \\ h'_{1,4} &\leftarrow \tilde{H}(h_{1,2}, h'_{3,4}) \\ h'_{1,8} &\leftarrow \tilde{H}(h'_{1,4}, h_{5,8}) \end{aligned}$$

## 3.6 Analiza complexității

Mai întâi, trebuie să analizăm structura arborelui Merkle. Cum arborele are $n$ frunze și pe fiecare nivel avem de două ori mai puține noduri decât pe nivelul următor, deducem că înălțimea arborelui are ordinul $\mathcal{O}(\log n)$.

Apoi, numărul de noduri ale arborelui este $1 + 2 + 4 + \cdots + n = 2n - 1$. Se poate demonstra că, chiar dacă $n$ nu este o putere a lui 2, acest număr este mărginit de $4n$.[3] Prin urmare, dimensiunea arborelui este întotdeauna $\mathcal{O}(n)$.

[3]Rezultatul nu se referă la arbori Merkle, ci la *segment trees* (arbori de intervale). Însă, aceste două tipuri de arbori au exact aceeași structură (ca dispunere a nodurilor). Prin urmare, afirmația se aplică și la arborii Merkle.

### 3.6.1 Complexitatea spațiu

Serverul trebuie să rețină tot arborele, deci va folosi $\mathcal{O}(n)$ spațiu suplimentar, ceea ce este perfect rezonabil din moment ce oricum trebuie să stocheze și fișierele, care deja ocupă $\mathcal{O}(n)$ spațiu. Mihai nu trebuie să rețină decât un hash, deci va avea nevoie de numai $\mathcal{O}(1)$ spațiu.

### 3.6.2 Timpul necesar încărcării fișierelor

Atât trimiterea celor $n$ fișiere de către Mihai cât și primirea lor de către server necesită $\mathcal{O}(n)$ timp. La acesta se adaugă doar timpul necesar construirii arborelui, lucru efectuat din nou de către ambele părți. Din moment ce, pentru fiecare nod trebuie să calculăm doar un simplu hash, deducem că timpul necesar este proporțional cu dimensiunea arborelui, deci $\mathcal{O}(n)$.

### 3.6.3 Timpul necesar unei validări de integritate

Mai întâi, Mihai primește de la server fișierul $f_i'$, operație care necesită timp constant. Apoi, Mihai recalculează nodurile roșii, al căror număr este egal cu înălțimea arborelui, deci $\mathcal{O}(\log n)$. Pentru aceasta, serverul trebuie să-i trimită lui Mihai nodurile albastre, al căror număr este egal cu înălțimea arborelui minus 1, deci tot $\mathcal{O}(\log n)$.

## 3.7 Aplicații în blockchain

Să spunem că Alice vrea să verifice dacă tocmai a primit o sumă de $x$ bitcoini de la Bob. Altfel spus, Alice vrea să verifice dacă tranzacția $t \triangleq (\texttt{Alice}, \texttt{Bob}, x)$ face parte din blockchain.

Bineînțeles, ea va fi nevoită să pună această întrebare unui nod din rețea. Nodul respectiv va trebui să caute tranzacția $t$ în fiecare nou bloc minat și să-i trimită lui Alice indexul blocului în care $t$ a fost găsit, precum și indexul lui $t$ în acel bloc. Cum poate Alice să se asigure că nodul este onest și deci că blocul respectiv chiar conține $t$ pe acea poziție?

O idee ar fi ca Alice să descarce întregul bloc, însă asta consumă în medie 1 MB, ceea ce poate fi foarte mult. Pentru a obține o soluție mai bună, ne vom folosi de *antetul* blocului, care conține, printre altele, rădăcina arborelui Merkle al tranzacțiilor sale. Astfel, este de ajuns ca nodul să-i trimită lui Alice hash-ul din rădăcină împreună cu o dovadă Merkle pentru $t$.

De menționat că ambele soluții necesită ca Alice să se bazeze pe consensul *proof-of-work* pentru validarea blocului, respectiv a antetului. În plus, observăm că nodul nu cunoaște inițial arborele Merkle al tranzacțiilor. Prin urmare, va trebui să-l construiască de fiecare dată când i se cere o dovadă Merkle. Complexitatea va fi proporțională cu mărimea blocului, timp care oricum este folosit pentru căutarea efectivă a lui $t$ în bloc.