Università degli Studi di Camerino

Scuola di Scienze e Tecnologie

Corso di Laurea in Informatica

Corso di Laurea in Informatica per la Comunicazione Digitale

Corso di Algoritmi e Strutture Dati 2024/2025

Parte di Laboratorio (6 CFU)

Istruzioni per la realizzazione del Progetto 1

ASDL2425MP1: Implementazione e Validazione di un Sistema Basato su Alberi di Merkle

**Introduzione a Merkle Tree e Merkle Proof**

Un **Merkle Tree** (o albero di Merkle) è una struttura dati ad albero binario utilizzata per garantire l’integrità e l’autenticità di un insieme di dati. Gli alberi di Merkle sono particolarmente utili in contesti come blockchain, sistemi di file distribuiti e database, dove è fondamentale verificare rapidamente se un dato appartiene a un insieme senza dover scaricare e analizzare tutto l’insieme.

**Struttura di un Merkle Tree**

Un Merkle Tree è composto da nodi, ciascuno dei quali contiene un valore hash. Esistono due tipi di nodi:

1. **Foglie**: Sono i nodi più in basso nell’albero e rappresentano gli hash dei dati originali.

2. **Nodi intermedi**: Rappresentano gli hash ottenuti combinando gli hash dei nodi figli (sinistro e destro).

L’**hash root** (o radice) è il nodo più in alto dell’albero ed è calcolato combinando gli hash intermedi fino a raggiungere un unico valore.

**Esempio di Merkle Tree**

Consideriamo un insieme di dati (in questo caso il numero di dati è una potenza di 2):

• D1 = "Alice paga Bob"

• D2 = "Bob paga Charlie"

• D3 = "Charlie paga Diana"

• D4 = "Diana paga Alice"

1. Calcoliamo gli hash delle foglie:

• H1 = Hash(D1)

• H2 = Hash(D2)

• H3 = Hash(D3)

• H4 = Hash(D4)

2. Combiniamo gli hash delle foglie per ottenere gli hash intermedi:

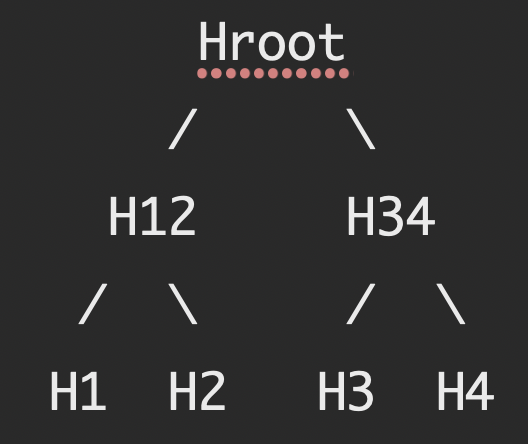
• H12 = Hash(H1 + H2) (concatenazione degli hash di D1 e D2)

• H34 = Hash(H3 + H4) (concatenazione degli hash di D3 e D4)

3. Calcoliamo l’hash root:

• Hroot = Hash(H12 + H34)

L’albero risultante avrà questa struttura:



**Merkle Proof**

Una **Merkle Proof** è una sequenza di hash necessaria per verificare se un dato appartiene a un Merkle Tree. Fornisce un modo efficiente per verificare l’integrità di un dato senza dover esaminare tutti gli altri elementi.

**Esempio di Merkle Proof**

Supponiamo di voler verificare se il dato D1 (Alice paga Bob) appartiene al Merkle Tree sopra.

**Passaggi:**

1. Forniamo l’hash di D1 (H1) e una sequenza di hash per risalire all’hash root:

• L’hash del nodo fratello: H2

• L’hash dell’altro ramo: H34

2. Combiniamo gli hash forniti:

• Calcoliamo H12 = Hash(H1 + H2)

• Calcoliamo Hroot = Hash(H12 + H34)

3. Confrontiamo il valore ottenuto (Hroot) con la radice del Merkle Tree.

Se gli hash combaciano, possiamo affermare che D1 appartiene al Merkle Tree.

**Merkle Tree con un numero di nodi che non è una potenza di 2, esempio con 5 nodi foglia.**

Quando il numero di dati non è una potenza di due, si verificherà che all’ultimo livello o in qualche livello superiore il numero di nodi non è pari e non si possono formare un numero esatto di coppie. Non è detto che ciò si verifichi a tutti i livelli, comunque.

La strategia da seguire in questi casi è la seguente: quando rimane un nodo singolo in un certo livello, invece di combinarlo con il fratello (che non esiste) ricalcoliamo l’hash sullo stesso nodo (verrà un hash diverso usando il metodo computeMD5(hash.getBytes()) della classe HashUtil fornita) e consideriamo questo nuovo hash il nodo parent del nodo singolo.

Strategie alternative potrebbero essere di duplicare il nodo e calcolare l’hash sul nodo duplicato oppure di portarlo a livello superiore così com’è. I test del progetto assumono che si usi la prima strategia indicata sopra.

**Input**

I dati sui nodi foglia sono:

• A, B, C, D, E

**Passaggi per costruire il Merkle Tree**

1. **Calcolare gli hash dei nodi foglia:**

• Hash(A), Hash(B), Hash(C), Hash(D), Hash(E)

2. **Livello 1:**

• Coppie: Hash(Hash(A) + Hash(B)) → Parent1

• Coppie: Hash(Hash(C) + Hash(D)) → Parent2

• L’ultimo nodo Parent3 = Hash(Hash(E)) *(non viene duplicato o copiato così com’è, si ricalcola l’hash sull’hash del dato singolo)*

3. **Livello 2 (genitori):**

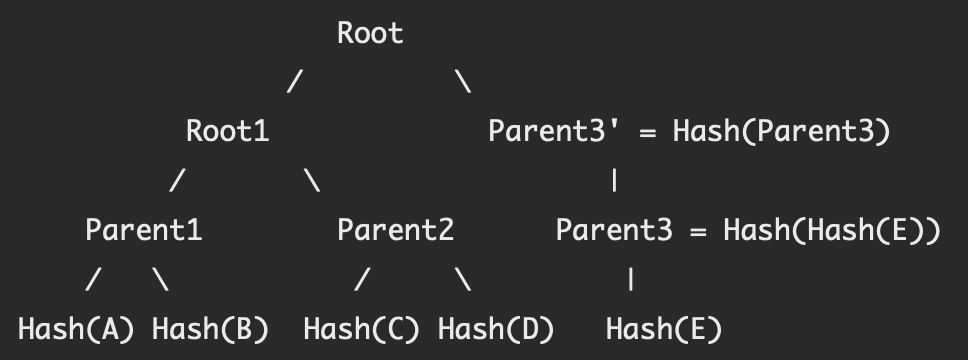
• Coppie: Hash(Parent1 + Parent2) → Root1

• Parent3’ = Hash(Parent3) *(non viene duplicato o lasciato così com’è, si ricalcola l’hash sull’hash del singolo)*

4. **Livello Root (radice):**

• Hash(Root1 + Parent3’) → Root del Merkle Tree

**Struttura finale**



**Merkle Proof: Dimostrare che C è parte dell’albero**

Per dimostrare che C è parte dell’albero, dobbiamo fornire una sequenza di hash che consenta di ricostruire la radice del Merkle Tree.

Passaggi per la proof:

1. Iniziamo con l’hash del dato da verificare: Hash(C).

2. Aggiungiamo gli hash “fratelli” a ogni livello:

* Livello 0: Hash “fratello” di Hash(C) cioè Hash(D).
* Livello 1: L’hash “fratello” di Parent2 cioè Parent1.
* Livello 2: L’hash “fratello” di Root1 cioè Parent3’.

3. Calcoliamo la radice usando questi hash e verifichiamo se corrisponde alla radice fornita.

Dato: C

Hash iniziale: Hash(C)

Merkle Proof:

1. Hash(D) (fratello sullo stesso livello)

2. Parent1 (fratello del livello superiore)

3. Parent3’ (fratello della radice)

**Ricostruzione della radice**

Passaggi:

1. Combiniamo Hash(C) e Hash(D) per calcolare Parent2 = Hash(Hash(C) + Hash(D))

2. Combiniamo Parent2 con Parent1 per calcolare Root1 = Hash(Parent1 + Parent2)

3. Combiniamo Root1 con Parent3’ per calcolare Root = Hash(Root1 + Parent3’)

Se la radice calcolata coincide con la radice fornita, il dato C è valido e parte dell’albero.

**Vantaggi di Merkle Tree e Merkle Proof**

1. **Efficienza**:

• Per verificare un dato, è necessario solo un numero ridotto di hash, anziché l’intero insieme.

2. **Integrità**:

• Un piccolo cambiamento in un dato modifica il corrispondente hash, rendendo immediato il rilevamento di manipolazioni.

3. **Scalabilità**:

• Merkle Tree può essere utilizzato su insiemi molto grandi senza la necessità di trasferire tutti i dati.

**Applicazioni**

• **Blockchain**: Utilizzati per verificare le transazioni in modo rapido.

• **Sistemi di file distribuiti**: Usati per verificare l’integrità di file in rete.

• **Database**: Per garantire che i dati non siano stati alterati.

**Nota sull’hashing**

Il progetto usa l’hash MD5 (Message Digest Algorithm 5), un algoritmo crittografico di hashing sviluppato da Ronald Rivest nel 1991. È ampiamente utilizzato per generare un valore hash univoco, chiamato digest, a partire da un input di dimensione arbitraria. L’output di MD5 è sempre un hash di 128 bit (16 byte), tipicamente rappresentato come una stringa esadecimale di 32 caratteri.

Ad esempio la stringa "Hello, World!" produce l’hash

65a8e27d8879283831b664bd8b7f0ad4

cioè 128 bit (16 byte) rappresentato come una stringa di 32 caratteri esadecimali.

Ogni carattere esadecimale corrisponde a 4 bit:

• **6**: 0110

• **5**: 0101

• **a**: 1010

eccetera.

La classe HashUtil fornita mette a disposizione due metodi, si faccia attenzione a quale si usa:

Il metodo public static String dataToHash(Object data) va usato per calcolare l’hash di un generico oggetto (i dati di tipo T del Merkle Tree) mentre il metodo public static String computeMD5(byte[] input) richiede un array di bytes su cui calcolare l’hash. Ad esempio, per calcolare l’hash della stringa “Hello, World!” di può usare HashUtil.computeMD5( “Hello, World!”.getBytes()) = 65a8e27d8879283831b664bd8b7f0ad4 o si può usare HashUtil.dataToHash( “Hello, World!”)= 55c7e2e6c31a6f0bfac559a7ea08796b

Si noti che i due hash non sono uguali! Il motivo è che il metodo dataToHash prima calcola l’hashCode (della classe Object di Java o ridefinito) dell’oggetto (che può essere di qualunque tipo ) e poi calcola md5 sull’array di bytes ottenuto dall’intero che corrisponde all’hashCode dell’oggetto.

Nel calcolo dell’hash dei nodi foglia del Merkle tree si usi il metodo dataToHash, mentre per la combinazione degli hash nei nodi interni si usi il metodo HashUtil.computeMD5(hash.getBytes())dove hash è la stringa esadecimale di 32 caratteri.

Per combinare due hash hash1 ed hash2 (anche nel caso in cui hash2 sia null) di due nodi dell’albero di Merkle e creare l’hash del parent node si può usare il seguente codice:

String combinedHash = (hash2 != null)

? hash1.getHash() + hash2.getHash()

/\* concatenazione di stringhe esadecimali \*/

: hash1.getHash();

String parentHash = HashUtil.computeMD5(combinedHash.getBytes());

In questo modo si calcoleranno gli hash in accordo ai test forniti.

**Caratteristiche principali di MD5**

1. Dimensione fissa dell’output: L’output è sempre di 128 bit, indipendentemente dalla dimensione dell’input.

2. Deterministico: Lo stesso input produce sempre lo stesso output.

3. Rapido: È progettato per essere computazionalmente efficiente.

4. Debole resistenza alle collisioni: È possibile trovare due input distinti con lo stesso hash (collisione), rendendo l’algoritmo meno sicuro per applicazioni crittografiche moderne.

Applicazioni comuni

• Controllo di integrità dei file: Verifica che i dati non siano stati alterati durante il trasferimento.

• Creazione di checksum: Utilizzato per generare identificatori univoci per file o dati.

• Confronto di dati: Confronto rapido di grandi quantità di dati tramite i loro hash.

Limiti di MD5

MD5 non è più considerato sicuro per applicazioni crittografiche:

• Collisioni note: Esistono metodi pratici per generare collisioni, rendendo MD5 vulnerabile a determinati attacchi.

• Non resistente agli attacchi pre-image: È possibile trovare un input che corrisponde a un hash dato.

Alternative moderne

A causa delle sue vulnerabilità, MD5 è stato sostituito in molte applicazioni da algoritmi più sicuri come SHA-256 e altri membri della famiglia SHA-2.

Nonostante i limiti, MD5 è ancora utile per scopi non crittografici come il controllo di integrità e l’identificazione di dati.

**Descrizione del Progetto**

L’obiettivo del progetto è implementare un sistema basato su alberi di Merkle per garantire l’integrità e l’autenticità dei dati in un insieme più ampio.

**Classi e Componenti**

Il progetto prevede cinque classi principali:

**1. HashUtil**

**Scopo**: Fornire metodi per il calcolo degli hash crittografici MD5.

**Metodi Principali:**

• dataToHash(Object data): Calcola l’hash MD5 del valore hashCode dell’oggetto data.

• computeMD5(byte[] input): Calcola l’hash MD5 di un array di byte.

• intToBytes(int value): Converte un intero in un array di byte (big-endian).

**Implementazione (fornita):**

• Utilizza l’algoritmo MD5 tramite la classe MessageDigest.

• Assicura che i valori restituiti siano in formato esadecimale.

**2**. **HashLinkedList**

**Scopo**:

Implementare una struttura dati denominata HashLinkedList: una lista concatenata che integra il calcolo e la gestione degli hash crittografici MD5 per ogni elemento contenuto.

**Metodi Principali**:

• **addAtHead(T data)**: Inserisce un elemento in testa alla lista.

• **addAtTail(T data)**: Inserisce un elemento in coda alla lista.

• **remove(T data)**: Rimuove il primo elemento che corrisponde al dato specificato.

• **getAllHashes()**: Restituisce una lista ordinata di tutti gli hash MD5 associati agli elementi della lista.

• **buildNodesString()**: Costruisce una rappresentazione testuale della lista mostrando i dati e i rispettivi hash.

• **iterator()**: Restituisce un iteratore fail-fast per scorrere gli elementi della lista in modo sicuro.

**Implementazione**:

• Ogni nodo della lista (Node) contiene il dato, l’hash MD5 calcolato tramite il metodo HashUtil.dataToHash(), e un riferimento al nodo successivo.

• L’iteratore fail-fast rileva modifiche concorrenti durante l’iterazione e lancia un’eccezione se rilevate.

• La struttura mantiene riferimenti sia al primo nodo (head) che all’ultimo (tail) per ottimizzare le operazioni di inserzione.

• L’implementazione di HashLinkedList è generica, supportando qualsiasi tipo di dato per i nodi.

**Scopo dei test forniti:** verificare il corretto funzionamento delle operazioni della classe HashLinkedList, assicurando la coerenza dei dati, la corretta gestione degli hash MD5, e la conformità alle specifiche di comportamento, anche in presenza di situazioni limite.

**3. MerkleNode**

**Scopo**: Rappresentare un nodo in un albero di Merkle.

**Metodi Principali:**

• **Costruttori**:

• Nodo foglia: accetta un hash come parametro.

• Nodo branch: accetta un hash e due nodi figli.

• isLeaf(): Determina se il nodo è una foglia.

• getHash()**,** getLeft()**,** getRight(): Restituiscono rispettivamente l’hash, il figlio sinistro e destro.

• equals **e** hashCode: Confrontano i nodi basandosi sull’hash.

**Implementazione:**

• Gli hash dei nodi foglia rappresentano direttamente l’hash di un dato.

• Gli hash dei nodi branch sono calcolati combinando gli hash dei figli (concatenazione + MD5).

**Scopo dei test forniti:**

• Verificare che i nodi foglia e branch vengano creati correttamente.

• Controllare la validità dei metodi isLeaf, equals e hashCode.

**4. MerkleProof**

**Scopo**: Gestire le prove di Merkle, consentendo la verifica dell’appartenenza di un dato o branch a un albero.

**Metodi Principali:**

• **Costruttore**:

• Accetta l’hash della radice e la lunghezza massima della prova.

• addHash(String hash, boolean isLeft): Aggiunge un hash alla prova, specificando se deve essere concatenato a sinistra.

• proveValidityOfData(Object data): Valida un dato utilizzando la prova.

• proveValidityOfBranch(MerkleNode branch): Valida un branch utilizzando la prova.

**Implementazione:**

• Utilizzare una lista concatenata (HashLinkedList) per memorizzare i passaggi della prova.

• La validazione deve calcolare iterativamente l’hash finale partendo dall’hash del dato o branch.

**Scopo dei test forniti:**

• Testare la costruzione e l’aggiunta di hash alla prova.

• Verificare la validità di dati e branch tramite le prove generate.

**5. MerkleTree**

**Scopo**: Rappresentare e gestire un albero di Merkle completo.

**Metodi Principali:**

• **Costruttore**: Costruisce un albero a partire da una HashLinkedList.

• getRoot()**,** getWidth()**,** getHeight(): Restituiscono rispettivamente la radice, la larghezza e l’altezza dell’albero.

• validateData(T data): Verifica se un dato appartiene all’albero.

• validateBranch(MerkleNode branch): Verifica se un branch è valido nell’albero.

• getMerkleProof(T data): Genera una prova di Merkle per un dato.

**Implementazione:**

• Calcolare gli hash intermedi concatenando gli hash dei figli e applicando la funzione MD5.

• Generare la prova di Merkle per un dato attraversando il cammino dalla foglia alla radice.

**Scopo dei test forniti:**

• Verificare la corretta costruzione dell’albero e il calcolo della radice.

• Testare la validità di dati e branch rispetto all’albero.

• Controllare la correttezza delle prove di Merkle.