Algorithmen und Datenstrukturen SoSe25

-Assignment 8-

Moritz Ruge
Matrikelnummer: 5600961
Lennard Wittenberg
Matrikelnummer: —

Juni 2025

Problem 1: Kryptographische Hashfunktionen und Blockchain

a) Kryptographische Hashfunktionen in Scala

Welche kryptographischen Hashfunktionen sind in Scala implementiert? Wie kann man sie verwenden?

Lösung: Non-cryptographic hashfunctions

Scala3 hat keine eigene Kryptographische Hashfunktion Implementiert (jedenfalls habe ich nichts gefunden).

In Scala hat man die Möglichkeit interne Hashfunktionen zu benutzten wie z.B. mit hashCode() methode[4]:

```
scala> val result = "hello".hashCode()
val result: Int = 99162322
```

Dies dient aber nicht der Kryptographischen Verschlüsselung von Werten, da es hierbei zu viele Kollisionen kommt, eher ist es zur Kontrolle von Werten gedacht.

Eine weitere Möglichkeit ist es über eine zusätzliche Scala Library zusätzliche Hashfunktionen zu benutzen: *scala.util.hashing.Hashing* [2]

```
import scala.util.hashing.Hashing


@main def run(): Unit =

  val h = summon[Hashing[String]]
  val hashWert = h.hash("Hallo")

  println(hashWert)
```

```
Output: 69490486
```

MurmurHash3

Oder auch eine Implementierung von MurmurHash3 von Rex Kerr. Auch dieser ist aber ein *non-cryptographic hashing algorithm* [3]

```
import scala.util.hashing.MurmurHash3


@main def run(): Unit =

  val text = "Hallo Welt"
  val hashWert = MurmurHash3.stringHash(text)

  println(hashWert)
```

```
Output: -608680269
```

Kryptographische Hashfunktionen

Um Kryptographische Hashfunktionen in Scala zu benutzen, müssen wir auf Bibliotheken von Java zurückgreifen. Um dies zu tun, Importieren wir z.B.:*java.security.MessageDigest* - für die Nutzung von SHA-256, MD5 oder auch SHA-1.[1]

```
import java.security.MessageDigest
```

Um z.B.: SHA-256 zu verwenden, müssen wir die vorgefertigten Methoden, getInstance(), digest(), benutzen

```
import java.security.MessageDigest


@main def run(): Unit =

  val message = "Hello World"
  val sha256 = MessageDigest.getInstance("SHA-256")
  val hashWert = sha256.digest(message.getBytes("UTF-8"))

  println(hashWert)
```

```
Output: [B@45820e51
```

- MessageDigest - ruft das Objekt auf
- getInstance() - Returns a MessageDigest object that implements the specified digest algorithm.
- digest - Performs a final update on the digest using the specified array of bytes, then completes the digest computation.

Da wir bei *println(hashWert)* eine Standard-toString-Ausgabe von einem Java-Array erhalten(also in Bytes), müssen wir die Ausgabe noch einmal in Hex-Zahlen umwandeln:

```
import java.security.MessageDigest


@main def run(): Unit =

  val message = "Hello World"
  val sha256 = MessageDigest.getInstance("SHA-256")
  val hashWert = sha256.digest(message.getBytes("UTF-8"))

  // Bytes nach Hex-String umwandeln
  val hashHex = hashWert.map("%02x".format(_)).mkString

  println(hashHex)
```

```
Output: a591a6d40bf420404a011733cfb7b190d62c65bf0bcda32b57b277d9ad9f146e
```

b) Verkettete Liste mit Hashreferenzen

Implementieren Sie in `Scala` eine einfach verkettete Liste mit Hashreferenzen. In den Knoten der einfach verketteten Liste sollen `String`-Objekte gespeichert werden. Verwenden Sie dazu eine kryptographische Hashfunktion wie in Teil (a).

```scala
import java.security.MessageDigest

// Hilfsfunktion um eine Hash fuer SHA-256 zu erzeugen Return String(Bytes), aus Aufgabe 1a
def sha256(input: String): String =
  val sha256 = MessageDigest.getInstance("SHA-256")
  val hashWert = sha256.digest(input.getBytes("UTF-8"))
  hashWert.map("%02x".format(_)).mkString

// Erzeugen der Datenstruktur fuer eine einfach verkettete Liste mit Hashreferenz
// data: inhalt des Knotens
// prevHash Der SHA-256 des vorhergehenden Knotens
// nextHash Verweisst auf den naechsten Knoten in der Liste
case class Node(data: String, prevHash: String, nextHash: Option[Node])

def buildingList(strings: List[String]): Node =
    val initialHash = "0" * 64 // setzte den ersten Hashwert auf 64 Nullen als Startwert
    var prevHash = initialHash
    var startNode: Node = null
    var previousNode: Node = null

    // wir drehen die Liste um da wir noch keine Werte fuer den zweiten Knoten haetten
    // d.h. wir fangen hinten an und rechnen die Liste umgedreht durch
    val reversed = strings.reverse

    for i <- 0 until reversed.length do
        val data = reversed(i)
        val newNode = Node(data, prevHash, Option(previousNode))
        prevHash = sha256(prevHash + data)
        previousNode = newNode
        startNode = newNode

    startNode

@main def main(): Unit =
    val liste = buildingList(List("Alice", "Bob", "Charlie"))

    def printList(node: Node): Unit =
        println(s"Daten: ${node.data}")
        println(s"PrevHash: ${node.prevHash}")
        println()
        node.nextHash.foreach(printList)

    printList(liste)
```

```
Output:
Daten: Alice
PrevHash: 5923931d867e648ec3e488074d631134d596b6a5424c5165258e9a6475fdc777

Daten: Bob
PrevHash: 12775e79fe15fd6aa0fcf6605550b6cc45ec10552c6d0b72685815af763f4774
```

```
Daten: Charlie
PrevHash: 0000000000000000000000000000000000000000000000000000000000000000
```

c) Nonce und Hash mit Nullen am Ende

Fügen Sie zu den Knoten Ihrer einfach verketteten Liste jeweils ein *Nonce* hinzu, und stellen Sie sicher, dass die Hashwerte in den Referenzen alle mit acht Nullen (in der Binärdarstellung) enden.

Wie viele Versuche sind dazu im Durchschnitt nötig?

```
import java.security.MessageDigest

// Hilfsfunktion um eine Hash fuer SHA-256 zu erzeugen Return String(Bytes), aus
    Aufgabe 1a
def sha256(input: String): String =
  val sha256 = MessageDigest.getInstance("SHA-256")
  val hashWert = sha256.digest(input.getBytes("UTF-8"))
  hashWert.map("%02x".format(_)).mkString

// Hilfsfunktion um einen Hash fue SHA-256 zu erzeigen - return Array[Bytes]
def sha256Bytes(input: String): Array[Byte] =
    val sha256 = MessageDigest.getInstance("SHA-256")
    sha256.digest(input.getBytes("UTF-8"))

// Erzeugen der Datenstruktur fuer eine einfach verkettete Liste mit Hashreferenz
// data: inhalt des Knotens
// prevHash Der SHA-256 des vorhergehenden Knotens
// nextHash Verweisst auf den naechsten Knoten in der Liste
case class Node(
    data: String,
    prevHash: String,
    hash: String,
    nextHash: Option[Node],
    Nonce: Int
)

def buildingList(strings: List[String]): Node =
    val initialHash = "0" * 64 // setzte den ersten Hashwert auf 64 Nullen als
    Startwert
    var prevHash = initialHash
    var startNode: Node = null
    var previousNode: Node = null

    // wir drehen die Liste um da wir noch keine Werte fuer den zweiten Knoten haetten
    // d.h. wir fangen hinten an und rechnen die Liste umgedreht durch
    val reversed = strings.reverse

    for i <- 0 until reversed.length do
        val data = reversed(i)
        val (nonce, hash) = findValidNonce(data, prevHash)
        val newNode = Node(data, prevHash, hash, Option(previousNode), nonce)
        prevHash = hash
        previousNode = newNode
        startNode = newNode

    startNode // return die startNode

// Checken ob die letzten Acht zeichen Nullen sind
def hashEndsWithEightZeroBits(hash: Array[Byte]): Boolean =
    hash.last == 0 // oder 0.toByte was aber das gleiche ist... 0

// Suche nach gueltiger Nonce
def findValidNonce(data: String, prevHash: String): (Int, String) =
    var nonce = 0
    var hashWert: Array[Byte] = Array(1.toByte) // Startwert nicht 0, damit Schleife
```

```
    startet
    var hashHex = ""

    // while do loop - solange bis wir am Ende 8 Nullen haben; 8 Nullen = 00 in Byte
    while !hashEndsWithEightZeroBits(hashWert) do
        val input = s"$data|$nonce|$prevHash" // wir trennen die inputs um moegliche
    zufaellige Kollision zu vermeiden
        hashWert = sha256Bytes(input)
        nonce += 1

    hashHex = hashWert.map("%02x".format(_)).mkString// Formatiere Array[Bytes] in
    String um
    (nonce - 1, hashHex) // Korrektur, da nach Schleife +1 ist


@main def main(): Unit =
    val liste = buildingList(List("Alice", "Bob", "Charlie"))

    def printList(node: Node): Unit =
        println(s"Daten: ${node.data}")
        println(s"PrevHash: ${node.prevHash}")
        println(s"Nonce:    ${node.Nonce}")
        println(s"Hash:     ${node.hash}")
        println()
        node.nextHash.foreach(printList)

    printList(liste)
```

```
Output:
Daten: Alice
PrevHash: 8efdfdc4cc1b5cb65ea1570e1371eac8ae7fb84f1b65c4f2b6d7f5d85f7c5700
Nonce:    471
Hash:     53c7e0c85a3182285cf4d837dfa1d495d5c883ac94f43b970c567b4d0a314e00

Daten: Bob
PrevHash: 213f013d20bc293b36a9fa7a59444aed5a881af4f17a41de31ca27ec7cf9e100
Nonce:    101
Hash:     8efdfdc4cc1b5cb65ea1570e1371eac8ae7fb84f1b65c4f2b6d7f5d85f7c5700

Daten: Charlie
PrevHash: 0000000000000000000000000000000000000000000000000000000000000000
Nonce:    47
Hash:     213f013d20bc293b36a9fa7a59444aed5a881af4f17a41de31ca27ec7cf9e100
```

Die Ausgabe des hashes erfolgt in Hexadezimal Darstellung.
Ein Byte = 8 Bit = 00000000 bis 11111111 → was in Hex = **00** bis FF sind. Das heißt, die letzten beiden Nullen sind die Hexadezimaldarstellung für 8 Nullen in Binärdarstellung.

Druchschnittliche Versuche:

Die Kryptographische Hashfunktion SHA-256 produziert genau 256 Bit oder auch 32 Byte. Die Wahrscheinlichkeit das ein Bit entweder 0 oder 1 annimmt ist $\frac{1}{2}$. Da die Letzten 8 Bit oder auch 1 Byte Null sein sollen ergibt sich folgende Formel:

$$P(alle8Bits = 0) = (\tfrac{1}{2})^8 = \tfrac{1}{256}$$

Der Erwartungswert ist dann:

$$\text{Erwartungswert} = \frac{1}{P} = \frac{1}{\frac{1}{256}} = 256$$

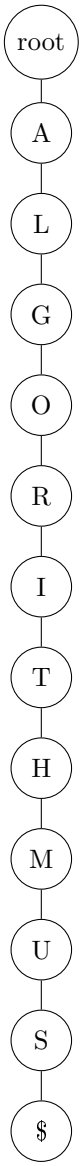
Die Wahrscheinlichkeit um am Ende Acht Nullen zu ziehen ist also $\frac{1}{256}$, bzw. braucht man ungefähr 256 versuche um den gewünschten Nonce zu berechnen.

Problem 2: Tries

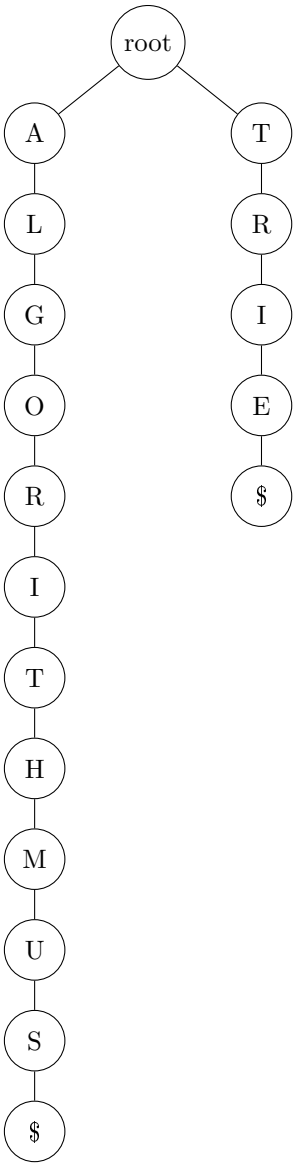
a) Zeichnen Sie einen unkomprimierten und einen komprimierten Trie für die Wörter {ALGORITHMUS, TRIE, BAUM, TORUS, BAHN, TORPEDO}.

Aufgabe 2a: unkomprimierter Trie:

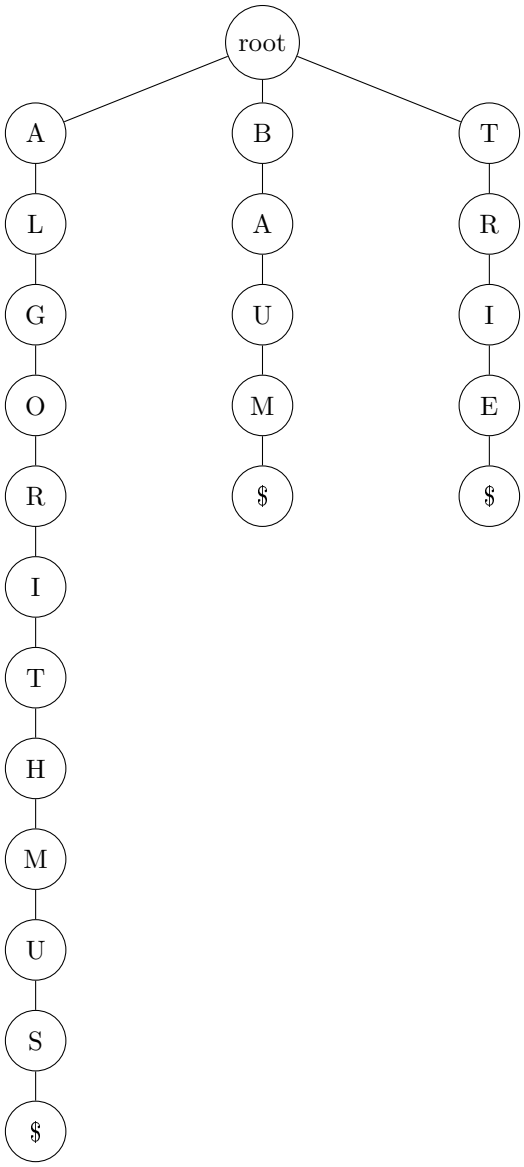
1. insert(ALGORITHMUS)



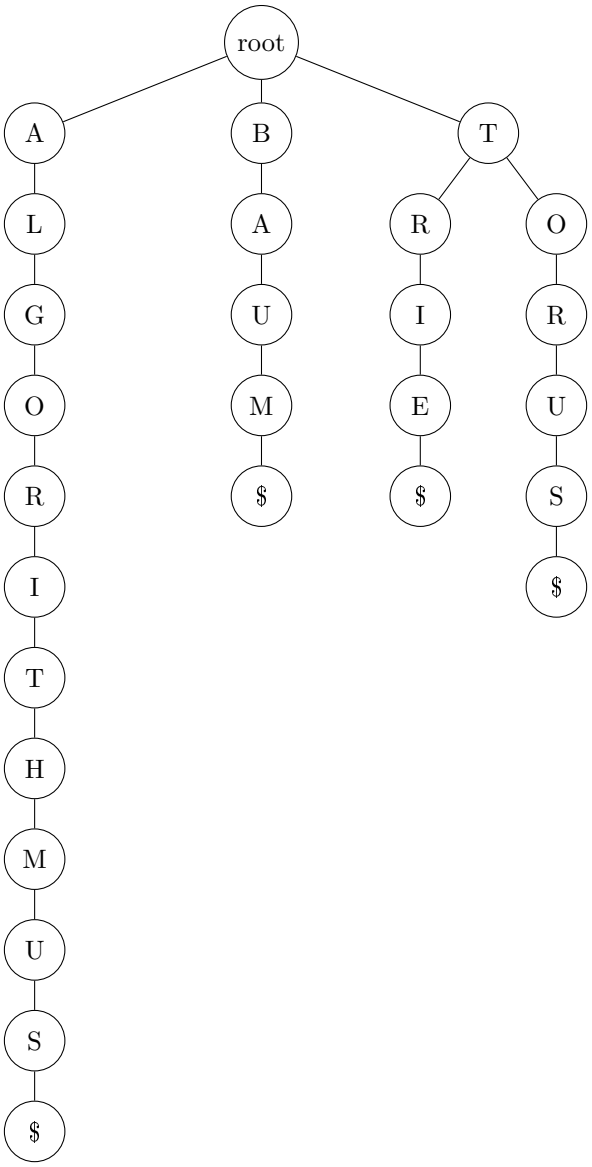
2. insert(TRIE)



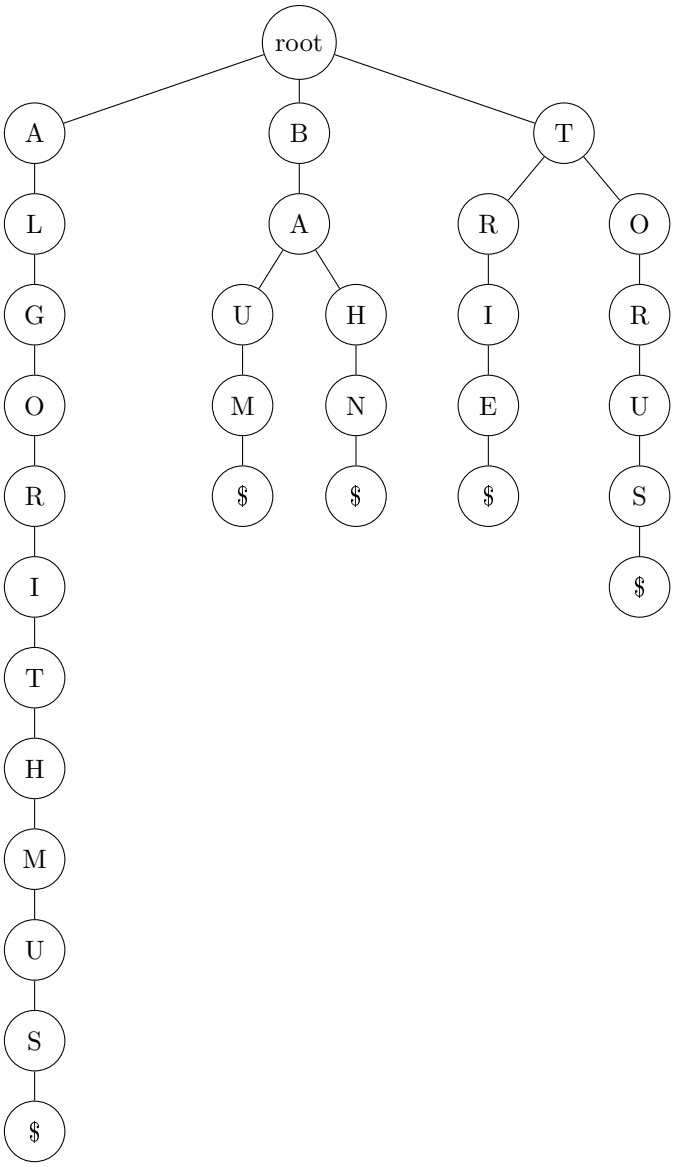
3. insert(BAUM)



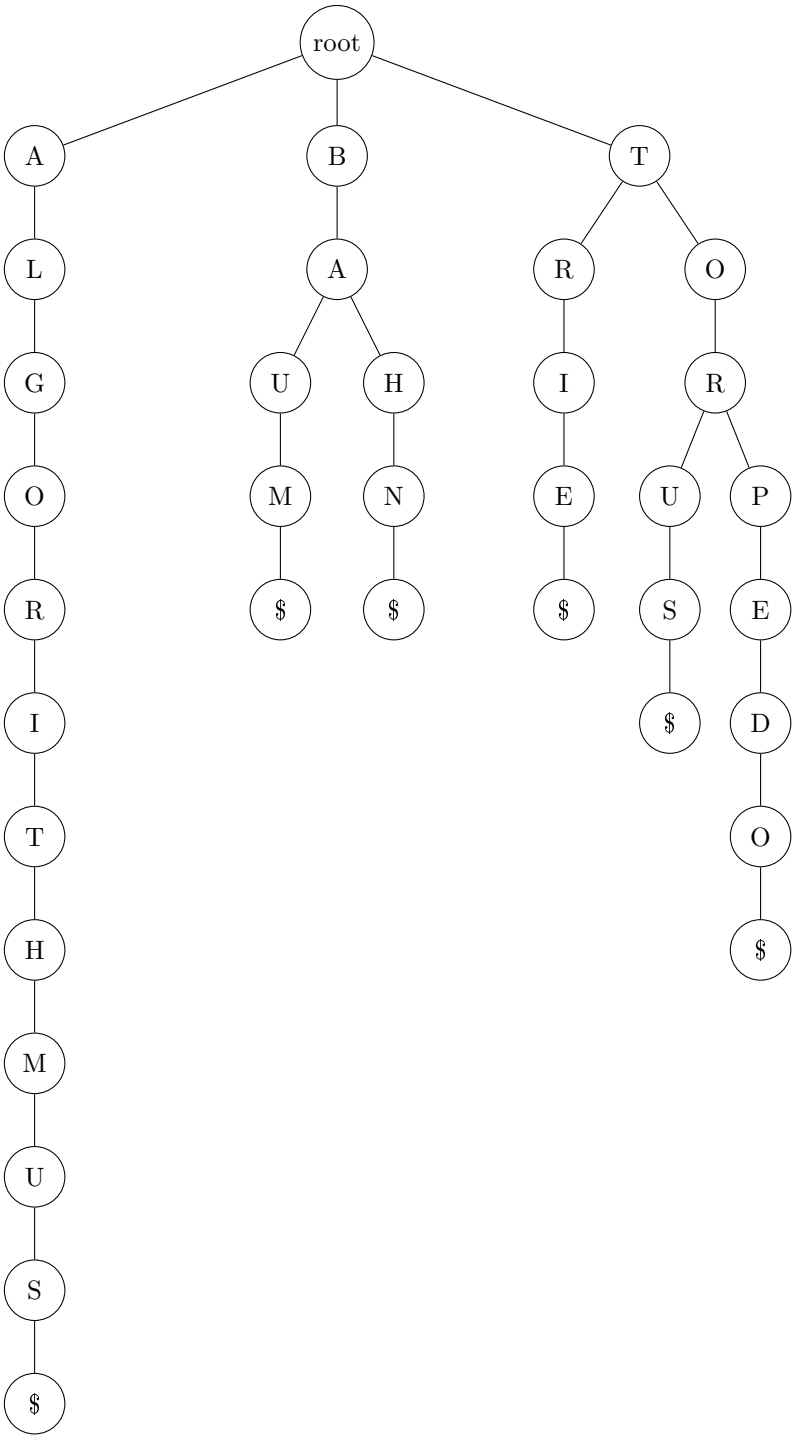
4. insert(TORUS)



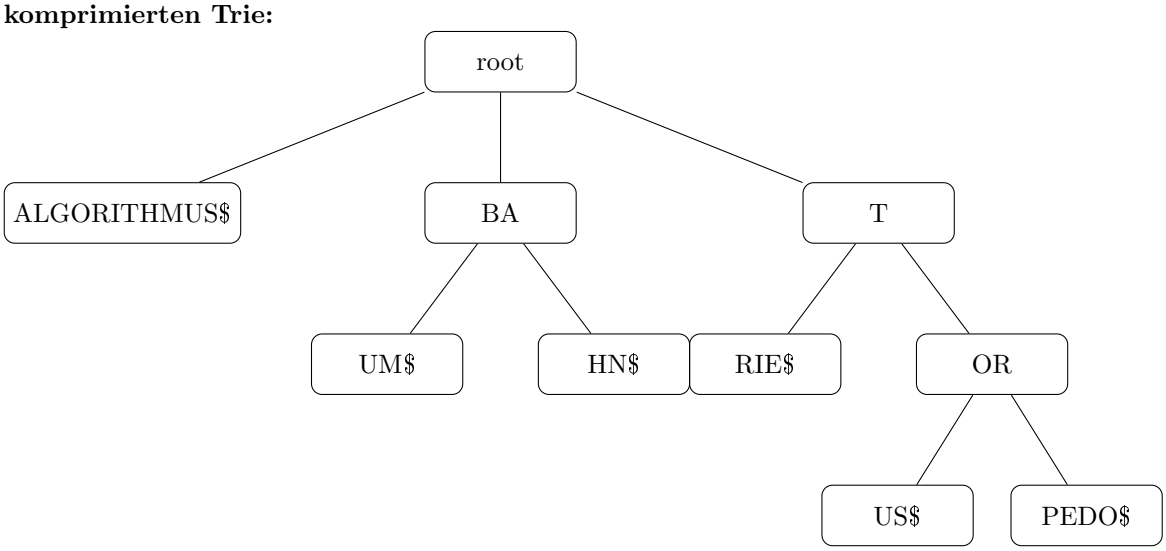
5. insert(BAHN)



6. insert(TORPEDO)



komprimierten Trie:



Enthält die Wörter: ALGORITHMUS, TRIE, BAUM, TORUS, BAHN, TORPEDO

b) Entwickeln Sie einen Algorithmus, der alle Wörter in einem unkomprimierten Trie ausgibt und dabei jede Kante höchstens zweimal besucht.

Algorithmus: Alle Wörter in einem Trie extrahieren

Beschreibung:

Der folgende Algorithmus durchläuft einen gegebenen Trie rekursiv und sammelt alle vollständigen Wörter, die im Trie enthalten sind. Jede Kante des Trie wird dabei genau einmal besucht. Die Rekursion verzweigt sich immer dann, wenn mehrere mögliche Folgeknoten für einen Eintrag existieren.

1. Übergabeparameter: `curr_node = Trie.root`, `curr_präfix = ""`, `words = []`
2. Wenn ein Eintrag des aktuell betrachteten Knotens das Stringende-Symbol (`*`) ist, dann stellt der Pfad von der Wurzel bis zu diesem Knoten ein vollständiges Wort dar.
   - ⇒ Füge dieses Wort zur Liste `words` hinzu.
3. Iteriere über alle Einträge `char` des aktuellen Knotens:
   - Wenn der Eintrag `char` *nicht* das Stringende-Symbol ist:
     - ⇒ Rufe die Funktion rekursiv auf mit:
       `curr_node = curr_node[char]`,
       `curr_präfix += char`,
       `words = words`
4. Gib die Liste `words` zurück. Diese enthält nun alle Wörter, die im Trie gespeichert sind.
5. Iteriere über die `words`-Liste und gebe jedes Wort aus.

Implementierung (Python)

```python
class Trie:
    def __init__(self):
        self.root = {}
        self.end_symbol = "*"

    def add(self, word):
        current_level = self.root
        for letter in word:
            if letter not in current_level:
                current_level[letter] = {}
            current_level = current_level[letter]
        current_level[self.end_symbol] = True

    def search_level(self, current_level, current_prefix, words):
        if self.end_symbol in current_level:
            words.append(current_prefix)
        for letter in sorted(current_level.keys()):
            if letter != self.end_symbol:
                self.search_level(current_level[letter], current_prefix + letter,
    words)
        return words

    def words_with_prefix(self, prefix):
        collected_words = []
        current_level = self.root
        for letter in prefix:
            if letter not in current_level:
                return []
            current_level = current_level[letter]
        return self.search_level(current_level, prefix, collected_words)


def main():
    trie = Trie()
    trie.add("help")
    trie.add("hello")
    trie.add("hi")
    found_words = trie.search_level(trie.root,"",[])
    for word in found_words:
        print(word)

main()
```

Ausgabe:

```
hello
help
hi
```

Anmerkung:

Der Funktionsaufruf `trie.search_level(trie.root, "", [])` zusammen mit dem anschließenden `print`-Block implementiert exakt den oben beschriebenen Algorithmus. Der Rekursionsbaum verzweigt sich bei jedem Knoten mit mehreren Kindknoten. Jede Kante im Trie wird dabei genau einmal besucht.

Problem 3: Implementierung von Tries

Beschreiben Sie kurz, wie man konkret die Operationen put(s, v), get(s), remove(s) und succ(s) auf unkomprimierten Tries implementieren kann. Dabei ist s jeweils eine nichtleere Zeichenkette und v ein Wert aus einer endlichen Wertemenge V . Geben Sie die Laufzeiten an.

Grundlegende Trie-Operationen

Die Bearbeitung dieser Aufgabe orientiert sich an der Darstellung von Pseudocode aus dem Skript.

put / add

Ziel: Fügt einen String mit einem Wert in einen Trie ein. Es wird angenommen, dass der String keine Leerzeichen enthält.

```
put(s, v):
    current = Trie.root              # Beginne bei der Wurzel des Trie, diese ist
    immer leer.
    for char in s:                   # Iteriere ueber die einzelnen Buchstaben des
    Strings.
        if char not in current:      # Existiert kein Eintrag fuer diesen Buchstaben
    ...
            current[char] = {}       # ... dann erstelle einen neuen Knoten (Dictionary
    ).
        current = current[char]      # Gehe in das naechste Level weiter.
    current["$"] = v                 # Am Ende: Markiere das Ende des Strings mit "$"
    und speichere den Wert v.
```

Laufzeit: $O(|s|)$ — Die Schleife läuft einmal für jeden Buchstaben im String $s \Rightarrow$ lineares Wachstum.

get / search

Ziel: Überprüft, ob ein gegebener String im Trie enthalten ist. Gibt den zugehörigen Wert zurück, falls vorhanden.

```
get(s):
    current = Trie.root
    for char in s:                   # Iteriere durch den Trie.
        if char not in current:
            return -1                # String ist nicht im Trie vorhanden.
        current = current[char]
    return current["$"]              # Gib den gespeicherten Wert zurueck.
```

Laufzeit: $O(|s|)$ — Linear zur Länge des eingegebenen Strings.

succ(s)

Ziel: Findet den nächsten String im Trie (alphabetisch sortiert), der mit dem Präfix s beginnt.

1. Iteriere durch den Trie wie bei `get()` oder `put()`, bis der Präfix s gefunden wurde.
2. Falls s ein vollständiger String ist:
   - Suche von diesem Knoten aus den alphabetisch kleinsten Nachfolger (Knoten mit `"$"`).

3. Falls kein solcher String existiert oder s unvollständig ist:
    - Gehe zurück zur letzten Verzweigung.
    - Suche dort den nächsten möglichen (alphabetisch kleinsten) String.

Laufzeit: $O(|s| + |\text{result}|)$ — Linear zur Länge des Präfixes plus Länge des gesuchten Ergebnisses.

remove(s)

Ziel: Entfernt einen gegebenen String aus dem Trie (falls vorhanden).

1. Iteriere durch den Trie wie bei `get()`, um die Endmarkierung des Strings zu finden.
    - Falls keine Endmarkierung gefunden wird, verlasse die Funktion.
2. Wird das Ende gefunden:
    - Lösche das `"$"`-Markierung.
    - Gehe rekursiv von unten (Blatt) zur Wurzel:
        - Wenn ein Knoten keine Kindknoten mehr hat ⇒ lösche ihn.
        - Sobald ein Knoten noch andere Kanten oder ein `"$"` enthält ⇒ stoppe.

Laufzeit: $O(|s|)$ — Die Löschoperation ist ebenfalls linear, da Schleife und Rekursion nicht verschachtelt sind. ⇒ $O(2|s|) = O(|s|)$

References

[1] Oracle Docs. *SHA-256: Java-Scala*. URL: https://docs.oracle.com/javase/8/docs/api/java/security/MessageDigest.html. (accessed: 19.06.2025).

[2] Scala 3 Docs. *Hashing*. URL: https://www.scala-lang.org/api/current/scala/util/hashing/Hashing$.html. (accessed: 19.06.2025).

[3] Scala 3 Docs. *MurmurHash3*. URL: https://www.scala-lang.org/api/current/scala/util/hashing/MurmurHash3$.html. (accessed: 19.06.2025).

[4] Geeksforgeeks. *Scala String hashCode() method with example*. URL: https://www.geeksforgeeks.org/scala/scala-string-hashcode-method-with-example/. (accessed: 19.06.2025).