

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 benutzen wie z.B. mit `hashCode()` methode[4]:

```
1 scala> val result = "hello".hashCode()
2 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]

```
1 import scala.util.hashing.Hashing
2
3
4 @main def run(): Unit =
5
6   val h = summon[Hashing[String]]
7   val hashWert = h.hash("Hallo")
8
9   println(hashWert)
```

Output: 69490486

MurmurHash3

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

```
1 import scala.util.hashing.MurmurHash3
2
3
4 @main def run(): Unit =
5
6   val text = "Hallo Welt"
7   val hashWert = MurmurHash3.stringHash(text)
8
9   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.[\[1\]](#)

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

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

```
1 import java.security.MessageDigest
2
3
4 @main def run(): Unit =
5
6   val message = "Hello World"
7   val sha256 = MessageDigest.getInstance("SHA-256")
8   val hashWert = sha256.digest(message.getBytes("UTF-8"))
9
10  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:

```
1 import java.security.MessageDigest
2
3
4 @main def run(): Unit =
5
6   val message = "Hello World"
7   val sha256 = MessageDigest.getInstance("SHA-256")
8   val hashWert = sha256.digest(message.getBytes("UTF-8"))
9
10  // Bytes nach Hex-String umwandeln
11  val hashHex = hashWert.map("%02x".format(_)).mkString
12
13  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).

```
1 import java.security.MessageDigest
2
3 // Hilfsfunktion um eine Hash fuer SHA-256 zu erzeugen Return String(Bytes), aus
  Aufgabe 1a
4 def sha256(input: String): String =
5   val sha256 = MessageDigest.getInstance("SHA-256")
6   val hashWert = sha256.digest(input.getBytes("UTF-8"))
7   hashWert.map("%02x".format(_)).mkString
8
9 // Erzeugen der Datenstruktur fuer eine einfach verkettete Liste mit Hashreferenz
10 // data: inhalt des Knotens
11 // prevHash Der SHA-256 des vorhergehenden Knotens
12 // nextHash Verweist auf den naechsten Knoten in der Liste
13 case class Node(data: String, prevHash: String, nextHash: Option[Node])
14
15 def buildingList(strings: List[String]): Node =
16   val initialHash = "0" * 64 // setzte den ersten Hashwert auf 64 Nullen als
    Startwert
17   var prevHash = initialHash
18   var startNode: Node = null
19   var previousNode: Node = null
20
21   // wir drehen die Liste um da wir noch keine Werte fuer den zweiten Knoten haetten
22   // d.h. wir fangen hinten an und rechnen die Liste umgedreht durch
23   val reversed = strings.reverse
24
25   for i <- 0 until reversed.length do
26     val data = reversed(i)
27     val newNode = Node(data, prevHash, Option(previousNode))
28     prevHash = sha256(prevHash + data)
29     previousNode = newNode
30     startNode = newNode
31
32   startNode
33
34 @main def main(): Unit =
35   val liste = buildingList(List("Alice", "Bob", "Charlie"))
36
37   def printList(node: Node): Unit =
38     println(s"Daten: ${node.data}")
39     println(s"PrevHash: ${node.prevHash}")
40     println()
41     node.nextHash.foreach(printList)
42
43   printList(liste)
```

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?

```
1 import java.security.MessageDigest
2
3 // Hilfsfunktion um eine Hash fuer SHA-256 zu erzeugen Return String(Bytes), aus
  Aufgabe 1a
4 def sha256(input: String): String =
5   val sha256 = MessageDigest.getInstance("SHA-256")
6   val hashWert = sha256.digest(input.getBytes("UTF-8"))
7   hashWert.map("%02x".format(_)).mkString
8
9 // Hilfsfunktion um einen Hash fue SHA-256 zu erzeugen - return Array[Byte]
10 def sha256Bytes(input: String): Array[Byte] =
11   val sha256 = MessageDigest.getInstance("SHA-256")
12   sha256.digest(input.getBytes("UTF-8"))
13
14 // Erzeugen der Datenstruktur fuer eine einfach verkettete Liste mit Hashreferenz
15 // data: inhalt des Knotens
16 // prevHash Der SHA-256 des vorhergehenden Knotens
17 // nextHash Verweist auf den naechsten Knoten in der Liste
18 case class Node(
19   data: String,
20   prevHash: String,
21   hash: String,
22   nextHash: Option[Node],
23   Nonce: Int
24 )
25
26 def buildingList(strings: List[String]): Node =
27   val initialHash = "0" * 64 // setzte den ersten Hashwert auf 64 Nullen als
    Startwert
28   var prevHash = initialHash
29   var startNode: Node = null
30   var previousNode: Node = null
31
32   // wir drehen die Liste um da wir noch keine Werte fuer den zweiten Knoten haetten
33   // d.h. wir fangen hinten an und rechnen die Liste umgedreht durch
34   val reversed = strings.reverse
35
36   for i <- 0 until reversed.length do
37     val data = reversed(i)
38     val (nonce, hash) = findValidNonce(data, prevHash)
39     val newNode = Node(data, prevHash, hash, Option(previousNode), nonce)
40     prevHash = hash
41     previousNode = newNode
42     startNode = newNode
43
44   startNode // return die startNode
45
46 // Checken ob die letzten Acht zeichen Nullen sind
47 def hashEndsWithEightZeroBits(hash: Array[Byte]): Boolean =
48   hash.last == 0 // oder 0.toByte was aber das gleiche ist... 0
49
50 // Suche nach gueltiger Nonce
51 def findValidNonce(data: String, prevHash: String): (Int, String) =
52   var nonce = 0
53   var hashWert: Array[Byte] = Array(1.toByte) // Startwert nicht 0, damit Schleife
```

```

54   startet
55   var hashHex = ""
56
57   // while do loop - solange bis wir am Ende 8 Nullen haben; 8 Nullen = 00 in Byte
58   while !hashEndsWithEightZeroBits(hashWert) do
59       val input = s"$data|$nonce|$prevHash" // wir trennen die inputs um moegliche
60       zufaellige Kollision zu vermeiden
61       hashWert = sha256Bytes(input)
62       nonce += 1
63
64   hashHex = hashWert.map("%02x".format(_)).mkString// Formatiere Array[Byte] in
65   String um
66   (nonce - 1, hashHex) // Korrektur, da nach Schleife +1 ist
67
68 @main def main(): Unit =
69     val liste = buildingList(List("Alice", "Bob", "Charlie"))
70
71     def printList(node: Node): Unit =
72         println(s"Daten: ${node.data}")
73         println(s"PrevHash: ${node.prevHash}")
74         println(s"Nonce:    ${node.Nonce}")
75         println(s"Hash:     ${node.hash}")
76         println()
77         node.nextHash.foreach(printList)
78
79     printList(liste)

```

Output:

Daten: Alice

PrevHash: 8efdfdc4cc1b5cb65ea1570e1371eac8ae7fb84f1b65c4f2b6d7f5d85f7c5700

Nonce: 471

Hash: 53c7e0c85a3182285cf4d837dfa1d495d5c883ac94f43b970c567b4d0a314e00

Daten: Bob

PrevHash: 213f013d20bc293b36a9fa7a59444aed5a881af4f17a41de31ca27ec7cf9e100

Nonce: 101

Hash: 8efdfdc4cc1b5cb65ea1570e1371eac8ae7fb84f1b65c4f2b6d7f5d85f7c5700

Daten: Charlie

PrevHash: 00

Nonce: 47

Hash: 213f013d20bc293b36a9fa7a59444aed5a881af4f17a41de31ca27ec7cf9e100

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:

$$P(\text{alle8Bits} = 0) = \left(\frac{1}{2}\right)^8 = \frac{1}{256}$$

Der Erwartungswert ist dann:

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

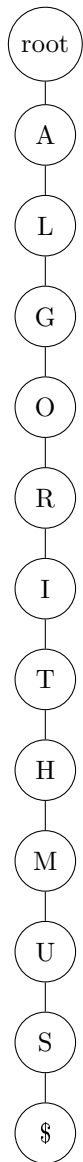
Die Wahrscheinlichkeit um am Ende Acht Nullen zu ziehen ist also $\frac{1}{256}$, bzw. braucht man ungefähr 256 versuche um den gewünschten Nonce zu berechnen.

Problem 2: Tries

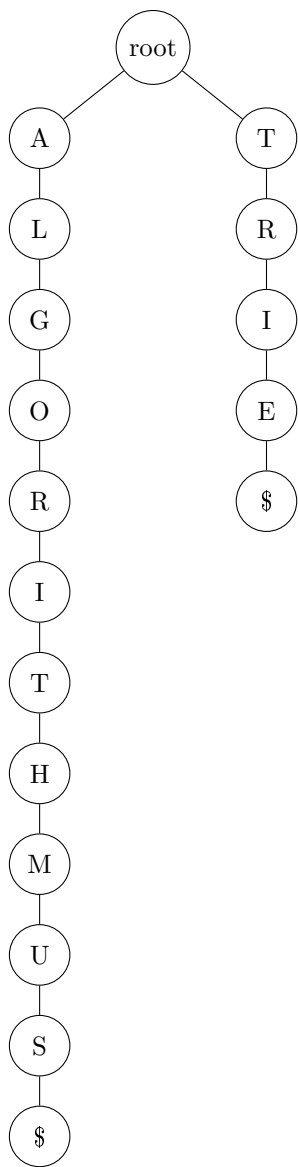
a) Zeichnen Sie einen unkomprimierten und einen komprimierten Trie für die Wörter {ALGORITHMUS, TRIE, BAUM, TORUS, BAHN, TORPEDO}.

Aufgabe 2a: unkomprimierter Trie:

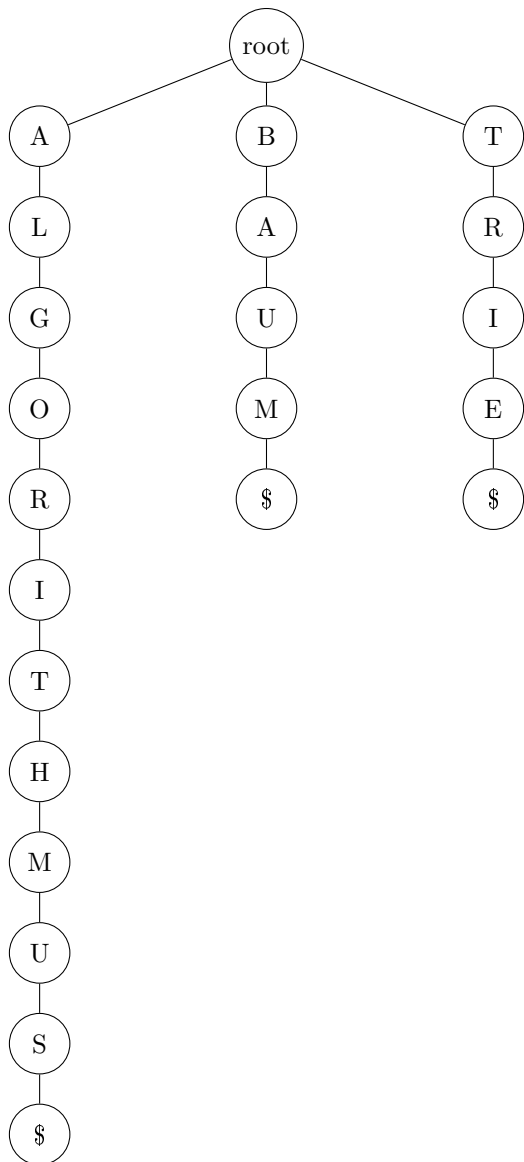
1. insert(ALGORITHMUS)



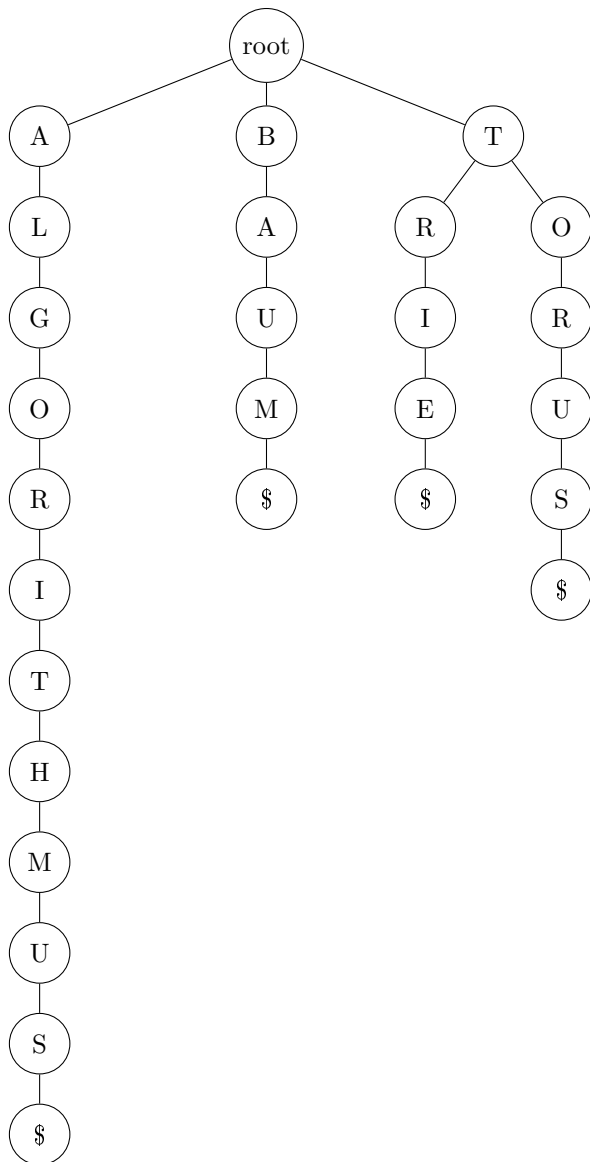
2. insert(TRIE)



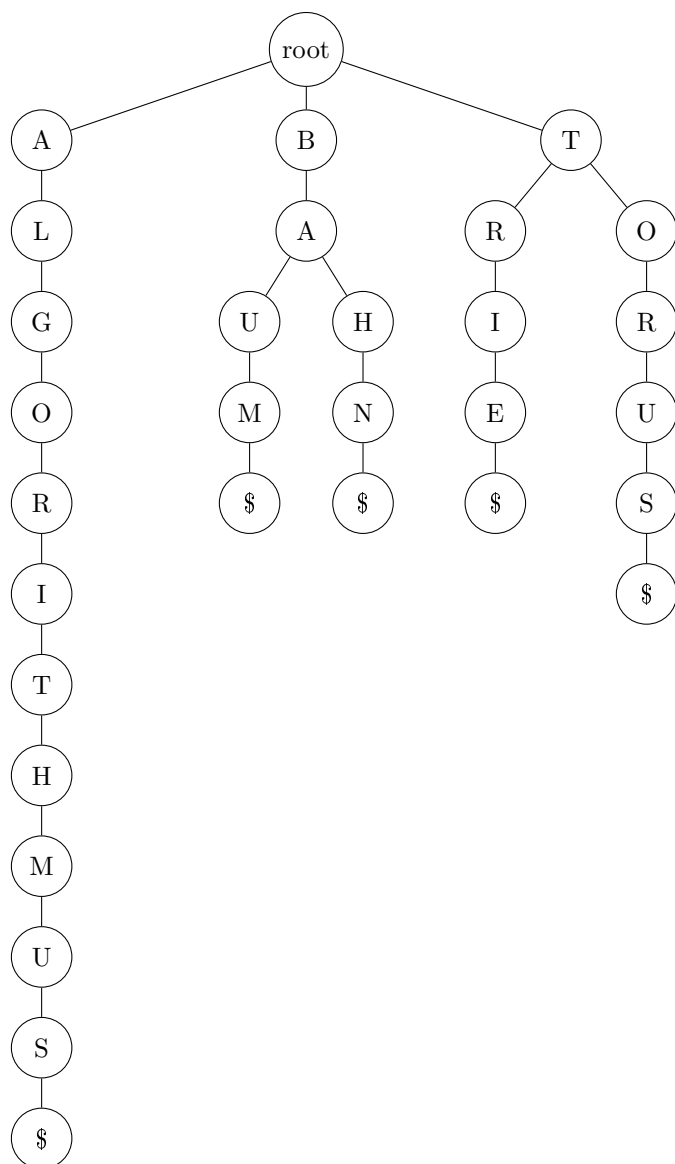
3. insert(BAUM)



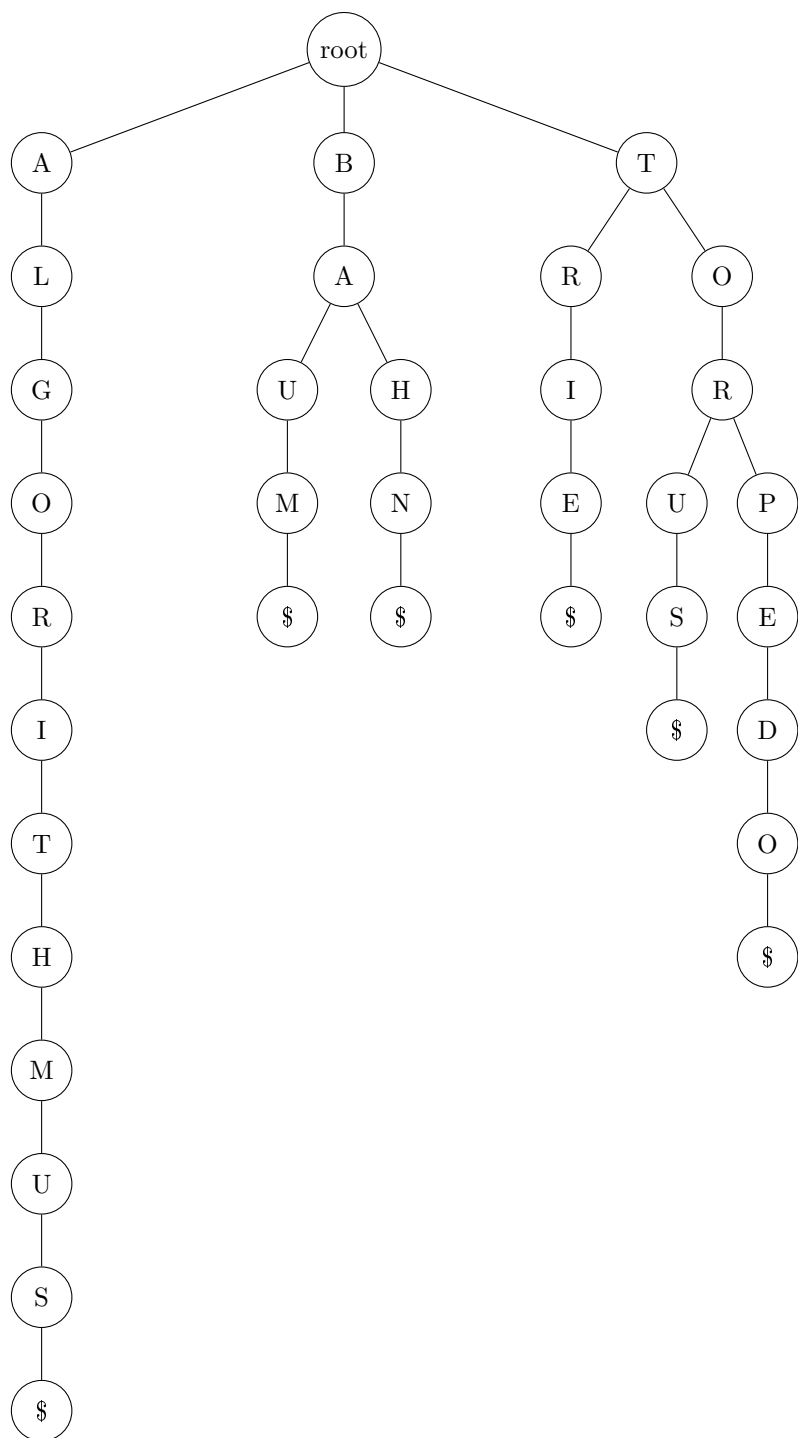
4. insert(TORUS)



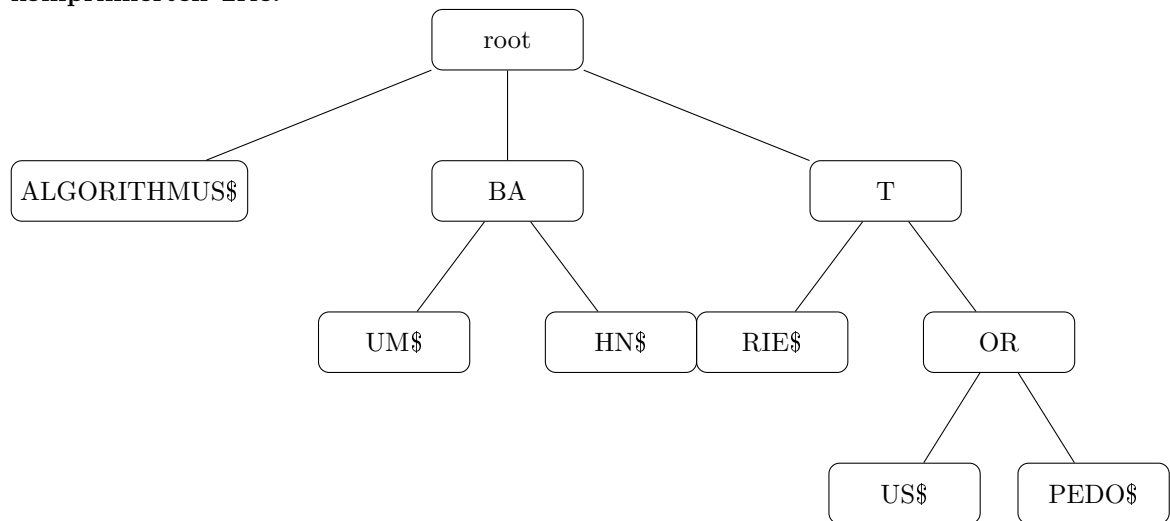
5. insert(BAHN)



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.
 - \Rightarrow 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:
 - \Rightarrow 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)

```
1 class Trie:
2     def __init__(self):
3         self.root = {}
4         self.end_symbol = "*"
5
6     def add(self, word):
7         current_level = self.root
8         for letter in word:
9             if letter not in current_level:
10                 current_level[letter] = {}
11                 current_level = current_level[letter]
12             current_level[self.end_symbol] = True
13
14     def search_level(self, current_level, current_prefix, words):
15         if self.end_symbol in current_level:
16             words.append(current_prefix)
17         for letter in sorted(current_level.keys()):
18             if letter != self.end_symbol:
19                 self.search_level(current_level[letter], current_prefix + letter,
20 words)
21         return words
22
23     def words_with_prefix(self, prefix):
24         collected_words = []
25         current_level = self.root
26         for letter in prefix:
27             if letter not in current_level:
28                 return []
29             current_level = current_level[letter]
30         return self.search_level(current_level, prefix, collected_words)
31
32 def main():
33     trie = Trie()
34     trie.add("help")
35     trie.add("hello")
36     trie.add("hi")
37     found_words = trie.search_level(trie.root, "", [])
38     for word in found_words:
39         print(word)
40
41 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 $\text{put}(s, v)$, $\text{get}(s)$, $\text{remove}(s)$ und $\text{succ}(s)$ 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.

```
1 put(s, v):
2     current = Trie.root           # Beginne bei der Wurzel des Trie, diese ist
3     immer leer.                   # Iteriere ueber die einzelnen Buchstaben des
4     for char in s:                 # Strings.
5         if char not in current:    # Existiert kein Eintrag fuer diesen Buchstaben
6             ...                    # ... dann erstelle einen neuen Knoten (Dictionary)
7             current[char] = {}
8         current = current[char]    # Gehe in das naechste Level weiter.
9     current["$"] = v              # Am Ende: Markiere das Ende des Strings mit "$"
10    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.

```
1 get(s):
2     current = Trie.root
3     for char in s:                 # Iteriere durch den Trie.
4         if char not in current:
5             return -1              # String ist nicht im Trie vorhanden.
6         current = current[char]
7     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 $\text{get}()$ oder $\text{put}()$, 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 \Rightarrow lösche ihn.
 - Sobald ein Knoten noch andere Kanten oder ein "\$" enthält \Rightarrow stoppe.

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

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](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](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).