

# Algorithmen auf Sequenzen

## Exakte Mustersuche mit mehreren Mustern

Sven Rahmann

Genominformatik  
Universitätsklinikum Essen  
Universität Duisburg-Essen  
Universitätsallianz Ruhr

- Problem der exakten Mustersuche für mehrere Muster gleichzeitig
- Anwendung: “Naive” Fehlertoleranz, z.B. {Maier, Mayer, Meier, Meyer}.
- Sei  $P := \{P_1, \dots, P_k\}$
- Sei  $m_i := |P_i|$  für  $1 \leq i \leq k$ ;  
 $m := \sum_i m_i$  (Gesamtlänge)

- Problem der exakten Mustersuche für mehrere Muster gleichzeitig
- Anwendung: “Naive” Fehlertoleranz, z.B. {Maier, Mayer, Meier, Meyer}.
- Sei  $P := \{P_1, \dots, P_k\}$
- Sei  $m_i := |P_i|$  für  $1 \leq i \leq k$ ;  
 $m := \sum_i m_i$  (Gesamtlänge)
- Einfache Lösung: Algorithmus für ein Muster (z.B. KMP)  $k$ -mal anwenden,  $\mathcal{O}(m + kn)$  Zeit
- Gewünscht:  $\mathcal{O}(m + n + z)$  Zeit; dabei  $z$ : Ausgabegröße

Anzahl der Treffer von  $P$  in Text  $T$  kann auf (mindestens) drei verschiedene Arten gezählt werden:

- 1 Überlappende Treffer; entspricht Anzahl der Paare  $(i, j)$  mit  $T[i : j] \in P$  (Standard-Definition).

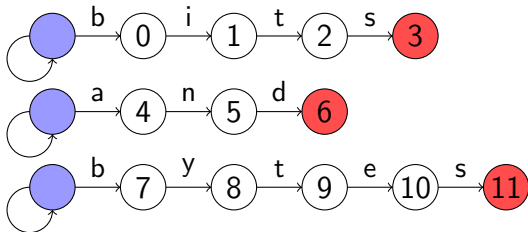
Anzahl der Treffer von  $P$  in Text  $T$  kann auf (mindestens) drei verschiedene Arten gezählt werden:

- 1 Überlappende Treffer; entspricht Anzahl der Paare  $(i, j)$  mit  $T[i : j] \in P$  (Standard-Definition).
- 2 Endpositionen von Treffern; entspricht Anzahl der  $j$ , für die es mindestens ein  $i$  gibt mit  $T[i : j] \in P$ .

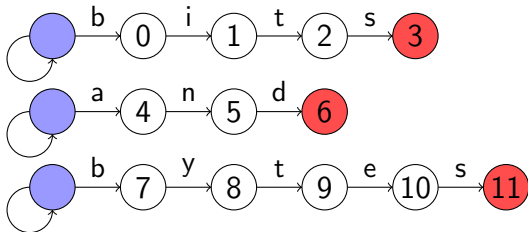
Anzahl der Treffer von  $P$  in Text  $T$  kann auf (mindestens) drei verschiedene Arten gezählt werden:

- 1** Überlappende Treffer; entspricht Anzahl der Paare  $(i, j)$  mit  $T[i : j] \in P$  (Standard-Definition).
- 2** Endpositionen von Treffern; entspricht Anzahl der  $j$ , für die es mindestens ein  $i$  gibt mit  $T[i : j] \in P$ .
- 3** Nichtüberlappende Treffer; gesucht ist eine maximale Menge von Paaren  $(i, j)$  mit  $T[i : j] \in P$ , so dass die Intervalle  $[i : j]$  paarweise disjunkt sind.

# Shift-And-Algorithmus auf Mengen



# Shift-And-Algorithmus auf Mengen



- NFA zum Finden der Wörter aus der Menge {bits, and, bytes}.
- Startzustände: blau; akzeptierend: rot
- Obwohl es mehrere Zusammenhangskomponenten gibt, handelt es sich um **einen** Automaten.



- NFA, der  $\Sigma^*P = \bigcup_{i=1}^k \Sigma^*P_k$  akzeptiert.
- Zustandsmenge  $Q := \{(i, j) \mid 1 \leq i \leq k, -1 \leq j < |P_i|\}$ .
- Zustand  $(i, j)$  repräsentiert Präfix  $P_i[..j]$  mit  $P_i \in P$ .

- NFA, der  $\Sigma^*P = \bigcup_{i=1}^k \Sigma^*P_k$  akzeptiert.
- Zustandsmenge  $Q := \{(i, j) \mid 1 \leq i \leq k, -1 \leq j < |P_i|\}$ .
- Zustand  $(i, j)$  repräsentiert Präfix  $P_i[..j]$  mit  $P_i \in P$ .
- Sei  $Q_0 = \{(i, -1) \mid 1 \leq i \leq k\}$  die Menge der Zustände, die das leere Präfix repräsentieren (Startzustände).
- Sei  $T = \{(i, |P_i| - 1) \mid 1 \leq i \leq k\}$  die Menge der Zustände, die jeweils ein Muster repräsentieren (Endzustände).
- Übergang von  $(i, j)$  nach  $(i, j + 1)$  durch Lesen von  $P_i[j + 1]$

Implementierung:

- Alle Muster werden zu einem Muster konkateniert,  
 $P \mapsto \text{bitsandbytes}$ .
- Die Startzustände werden in einem Bitfeld  $A_0$  gespeichert,  
 $A_0[j] := 1$ , wenn  $j \in \{\sum_{i=1}^g |P_i| \mid 0 \leq g < k\}$ .  
Hier  $A_0 = 000010010001$ .
- Die akzeptierenden Zustände werden ebenfalls in einem Bitfeld gespeichert,  
 $\text{accept}[j] := 1$ , wenn  $j \in \{\sum_{i=1}^g |P_i| - 1 \mid 1 \leq g \leq k\}$ .  
Hier  $\text{accept} = 100001001000$ .

Erstellung der Masken:

```
1 def create_masks(Ps, S):
2     mask = {s: 0 for s in S}
3     A_0, accept, bit = 0, 0, 1
4     for p in Ps:
5         A_0 |= bit
6         for c in p:
7             mask[c] |= bit
8             bit <<= 1
9         accept |= bit
10    return mask, A_0, accept >> 1
```

Mustersuche mit dem Shift-And-Algorithmus:

```
1 def shift_and_multi(Ps, T):
2     S, A = set("".join(Ps) + T), 0
3     mask, A_0, accept = create_masks(Ps, S)
4     for i, c in enumerate(T):
5         A = ((A << 1) | A_0) & mask[c]
6         if A & accept:
7             yield i
```

Mustersuche mit dem Shift-And-Algorithmus:

```
1 def shift_and_multi(Ps, T):
2     S, A = set("").join(Ps) + T, 0
3     mask, A_0, accept = create_masks(Ps, S)
4     for i, c in enumerate(T):
5         A = ((A << 1) | A_0) & mask[c]
6         if A & accept:
7             yield i
```

In diesem Code wird nur die Endposition eines Matches übergeben. Will man zusätzlich die Startposition kennen, so muss erkannt werden, welche akzeptierenden Zustände aktiv sind.

- Der Shift-And-Algorithmus lässt sich problemlos auf eine Menge von Mustern erweitern, solange die Gesamtlänge aller Muster kleiner als die Registergröße ist.
- Die Laufzeit beträgt  $\mathcal{O}(m + n \lceil m/W \rceil)$ .
- Es wird nur berichtet, an welchen Textpositionen mindestens ein Muster endet.

- Jedes Muster mit KMP suchen: Zeit  $\mathcal{O}(kn + m)$ .
- Gewünscht:  $\mathcal{O}(n + m)$



- Jedes Muster mit KMP suchen: Zeit  $\mathcal{O}(kn + m)$ .
- Gewünscht:  $\mathcal{O}(n + m)$
- Aho-Corasick-Algorithmus
- Datenstrukturaufbau:
  - NFA, der die NFAs der Einzelmuster verschmilzt
  - DFA daraus
  - Simulation des DFA mit lps-ähnlicher Funktion:  
Vom aktuellen “tiefsten” aktiven Zustand, finde  
“nächsthöheren” aktiven Zustand im NFA

## Definition (Trie)

Der Trie (von *retrieval*) zu einer endlichen Menge von Wörtern  $S \subset \Sigma^+$  ist ein kantenbeschrifteter Baum über der Knotenmenge  $\text{Prefixes}(S)$  mit folgender Eigenschaft:  
Der Knoten  $s$  ist genau dann ein Kind von Knoten  $t$ , wenn  $s = ta$  für ein  $a \in \Sigma$ ;  
die Kante  $t \rightarrow s$  ist dann mit  $a$  beschriftet.

## Definition (Trie)

Der Trie (von *retrieval*) zu einer endlichen Menge von Wörtern  $S \subset \Sigma^+$  ist ein kantenbeschrifteter Baum über der Knotenmenge  $\text{Prefixes}(S)$  mit folgender Eigenschaft:  
Der Knoten  $s$  ist genau dann ein Kind von Knoten  $t$ , wenn  $s = ta$  für ein  $a \in \Sigma$ ;  
die Kante  $t \rightarrow s$  ist dann mit  $a$  beschriftet.

Knoten  $v$  entspricht dem String, den die Kanten entlang des Pfades von der Wurzel zu  $v$  buchstabieren.

## Definition (Trie)

Der Trie (von *retrieval*) zu einer endlichen Menge von Wörtern  $S \subset \Sigma^+$  ist ein kantenbeschrifteter Baum über der Knotenmenge  $\text{Prefixes}(S)$  mit folgender Eigenschaft:  
Der Knoten  $s$  ist genau dann ein Kind von Knoten  $t$ , wenn  $s = ta$  für ein  $a \in \Sigma$ ;  
die Kante  $t \rightarrow s$  ist dann mit  $a$  beschriftet.

Knoten  $v$  entspricht dem String, den die Kanten entlang des Pfades von der Wurzel zu  $v$  buchstabieren.

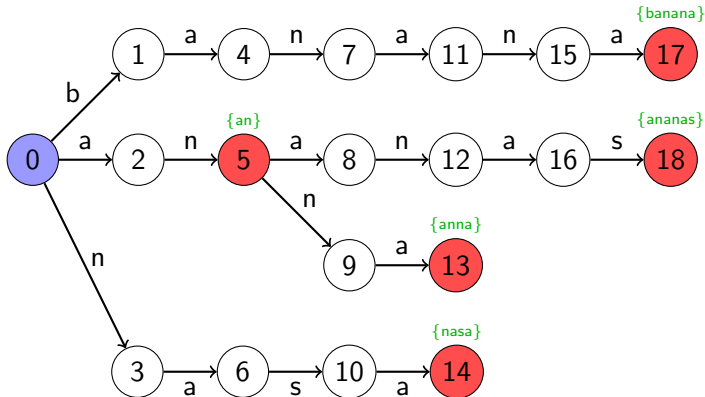
Der Aho-Corasick-Algorithmus arbeitet auf dem Trie zur Mustermenge  $P$ .

Der AC-Algorithmus verwendet die Idee der  $lps$ -Funktion des KMP-Algorithmus auf dem Trie von  $P$ . Sei

$$lps(q) := \arg \max_{p \in \text{Prefixes}(P)} \{ |p| \mid |p| < |q|, p \text{ Suffix von } q \}.$$

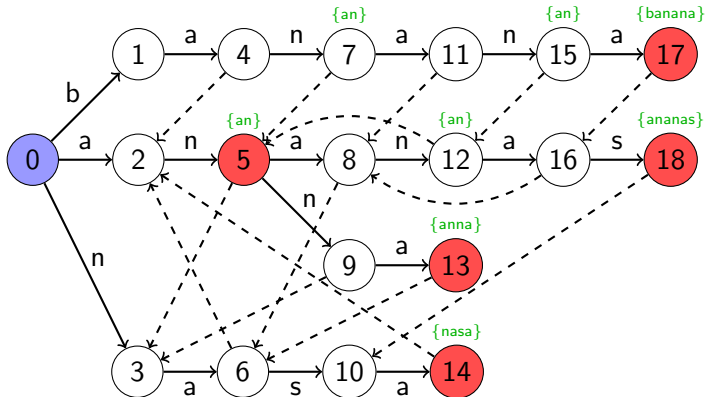
- Hierbei sind  $q$  und  $lps(q)$  keine Zahlen, sondern Knoten im Trie, die jeweils auf einen String abbilden.
- Die Funktion  $lps(q)$  verweist auf den Knoten, der zum längsten Präfix  $p$  eines Musters in  $P$  gehört, welches gleichzeitig ein echtes Suffix von  $q$  ist.

# Der Aho-Corasick-Algorithmus



Trie über Mustermenge  $P = \{an, ananas, anna, banana, nasa\}$ .  
Startzustand: blau; Wörter aus  $P$ : rot/grün

# Der Aho-Corasick-Algorithmus



Trie über Mustermenge  $P = \{an, ananas, anna, banana, nasa\}$ .  
Startzustand: blau; Wörter aus  $P$ : rot/grün  
Gestrichelte Pfeile: lps-Funktion.

Erzeugen der  $lps$ -Funktion:

- 1 Traversiere den Trie mittels Breitensuche. Beginne beim Startknoten  $\epsilon$ .  $lps[\epsilon]$  ist nicht definiert.



Erzeugen der *lps*-Funktion:

- 1 Traversiere den Trie mittels Breitensuche. Beginne beim Startknoten  $\epsilon$ .  $lps[\epsilon]$  ist nicht definiert.
- 2 Für Kinder  $q$  von  $\epsilon$  gilt:  $lps[q] = \epsilon$

Erzeugen der  $lps$ -Funktion:

- 1 Traversiere den Trie mittels Breitensuche. Beginne beim Startknoten  $\epsilon$ .  $lps[\epsilon]$  ist nicht definiert.
- 2 Für Kinder  $q$  von  $\epsilon$  gilt:  $lps[q] = \epsilon$
- 3 Für alle restlichen Knoten:
  - In Knoten  $xa$ , setze  $v := x$ .
  - Prüfe, ob Knoten  $lps[v]$  eine ausgehende Kante  $a$  hat.
  - Wenn ja, setze  $lps[xa] := lps[v]a$ .
  - Sonst gehe solange über zu  $v := lps[v]$ , bis entweder  $lps[v]$  nicht mehr existiert oder es eine ausgehende Kante  $a$  von  $v$  gibt. Im letzteren Fall setze wieder  $lps[xa] := lps[v]a$ .

Ausgabe der gefundenen Muster:

- Es findet eine Ausgabe statt, wenn der aktuelle Knoten
  - einem akzeptierenden Zustand entspricht.
  - über (mehrfache) *lps*-Transitionen einen akzeptierenden Zustand erreicht.

Ausgabe der gefundenen Muster:

- Es findet eine Ausgabe statt, wenn der aktuelle Knoten
  - einem akzeptierenden Zustand entspricht.
  - über (mehrfache) *lps*-Transitionen einen akzeptierenden Zustand erreicht.
- Die Indizes der auszugebenden Muster können
  - im Knoten als Index abgespeichert werden. Bei der Ausgabe wird den *lps*-Transitionen gefolgt, um alle Wörter auszugeben.
  - direkt im Knoten als Liste durch Konkatenation mit ihren *lps*-Vorgängern abgespeichert werden.

Implementierung:

Es wird eine Klasse `ACNode` mit folgenden Attributen angelegt:

- Ein Dictionary `targets`:  $c \rightarrow q$  mit  $c \in \Sigma, q \in \text{Prefixes}(P)$  zum speichern der Kinderknoten.
- Eine Referenz `lps` auf den `lps`-Vorgängerknoten.
- Eine Liste `out` mit den Indizes der auszugebenden Muster.

```
1 class ACNode():
2     def __init__(self):
3         self.targets = dict()
4         self.lps = None
5         self.out = []
```

Implementierung:

Die *delta*-Funktion ist ähnlich aufgebaut, wie die *delta*-Funktion des KMP, nur dass die Zustände hierbei durch Knoten abgebildet werden.

```
1 def delta(q, c):  
2     while (q.lps is not None) and (c not in q.targets):  
3         q = q.lps  
4     if c in q.targets:  
5         q = q.targets[c]  
6     return q
```

Implementierung:

Das Erstellen des Aho-Corasick-Automates ist aufgeteilt in zwei Phasen:

- 1 Erstellen des Tries.
- 2 Hinzufügen der *lps*-Kanten.

```
1 def build_AC(P) :  
2     root = build_trie(P)  
3     build_lps(root)  
4     return root
```

## Implementierung:

```
1 def build_trie(P):
2     root = ACNode()
3     for (i, p) in enumerate(P):
4         node = root
5         for c in p:
6             if c not in node.targets:
7                 node.targets[c] = ACNode()
8                 node = node.targets[c]
9         node.out.append(i)
10    return root
```



## Erstellung des Tries:

```
1 def build_trie(P):
2     root = ACNode()
3     for (i, p) in enumerate(P):
4         node = root
5         for c in p:
6             if c not in node.targets:
7                 node.targets[c] = ACNode()
8                 node = node.targets[c]
9         node.out.append(i)
10    return root
```

$P = \{$   
  an,  
  ananas,  
  anna,  
  banana,  
  nasa  
 $\}$

## Erstellung des Tries:

```
1 def build_trie(P):
2     root = ACNode()
3     for (i, p) in enumerate(P):
4         node = root
5         for c in p:
6             if c not in node.targets:
7                 node.targets[c] = ACNode()
8                 node = node.targets[c]
9         node.out.append(i)
10    return root
```

$P = \{$   
  an,  
  ananas,  
  anna,  
  banana,  
  nasa} $\}$



## Erstellung des Tries:

```
1 def build_trie(P):
2     root = ACNode()
3     for (i, p) in enumerate(P):
4         node = root
5         for c in p:
6             if c not in node.targets:
7                 node.targets[c] = ACNode()
8                 node = node.targets[c]
9         node.out.append(i)
10    return root
```

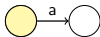
$P = \{\text{an, ananas, anna, banana, nasa}\}$



## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8                 node = node.targets[c]  
9         node.out.append(i)  
10    return root
```

$P = \{\text{an, ananas, anna, banana, nasa}\}$



## Erstellung des Tries:

```
1 def build_trie(P):
2     root = ACNode()
3     for (i, p) in enumerate(P):
4         node = root
5         for c in p:
6             if c not in node.targets:
7                 node.targets[c] = ACNode()
8             node = node.targets[c]
9         node.out.append(i)
10    return root
```

$P = \{\text{an, ananas, anna, banana, nasa}\}$



## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8                 node = node.targets[c]  
9         node.out.append(i)  
10    return root
```

$P = \{\text{an, ananas, anna, banana, nasa}\}$



## Erstellung des Tries:

```
1 def build_trie(P):
2     root = ACNode()
3     for (i, p) in enumerate(P):
4         node = root
5         for c in p:
6             if c not in node.targets:
7                 node.targets[c] = ACNode()
8             node = node.targets[c]
9         node.out.append(i)
10    return root
```

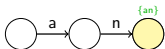
$P = \{\text{an, ananas, anna, banana, nasa}\}$



## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8             node = node.targets[c]  
9         node.out.append(i)  
10    return root
```

$P = \{\text{an},$   
 ananas,  
 anna,  
 banana,  
 nasa}

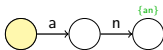




## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8                 node = node.targets[c]  
9         node.out.append(i)  
10    return root
```

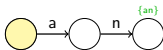
$P = \{$   
an,  
ananas,  
anna,  
banana,  
nasa  
 $\}$



## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8                 node = node.targets[c]  
9         node.out.append(i)  
10    return root
```

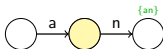
$P = \{$ an,  
 ananas,  
 anna,  
 banana,  
 nasa $\}$



## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8             node = node.targets[c]  
9         node.out.append(i)  
10    return root
```

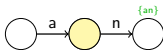
$P = \{$   
  an,  
  **ananas**,  
  anna,  
  banana,  
  nasa  
 $\}$



## Erstellung des Tries:

```
1 def build_trie(P):
2     root = ACNode()
3     for (i, p) in enumerate(P):
4         node = root
5         for c in p:
6             if c not in node.targets:
7                 node.targets[c] = ACNode()
8                 node = node.targets[c]
9         node.out.append(i)
10    return root
```

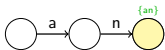
$P = \{$ an,  
ananas,  
anna,  
banana,  
nasa $\}$



## Erstellung des Tries:

```
1 def build_trie(P):
2     root = ACNode()
3     for (i, p) in enumerate(P):
4         node = root
5         for c in p:
6             if c not in node.targets:
7                 node.targets[c] = ACNode()
8             node = node.targets[c]
9         node.out.append(i)
10    return root
```

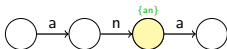
$P = \{$ an,  
ananas,  
anna,  
banana,  
nasa $\}$



## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8                 node = node.targets[c]  
9         node.out.append(i)  
10    return root
```

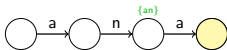
$P = \{$   
an,  
ananas,  
anna,  
banana,  
nasa  
 $\}$



## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8             node = node.targets[c]  
9         node.out.append(i)  
10    return root
```

$P = \{$ an,  
ananas,  
anna,  
banana,  
nasa $\}$



## Erstellung des Tries:

```
1 def build_trie(P):
2     root = ACNode()
3     for (i, p) in enumerate(P):
4         node = root
5         for c in p:
6             if c not in node.targets:
7                 node.targets[c] = ACNode()
8                 node = node.targets[c]
9         node.out.append(i)
10    return root
```

$P = \{$ an,  
 ananas,  
 anna,  
 banana,  
 nasa $\}$





## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8             node = node.targets[c]  
9         node.out.append(i)  
10    return root
```

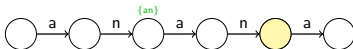
$P = \{$   
  an,  
  **ananas**,  
  anna,  
  banana,  
  nasa  
 $\}$



## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8                 node = node.targets[c]  
9         node.out.append(i)  
10    return root
```

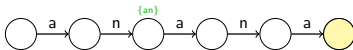
$P = \{$ an,  
 ananas,  
 anna,  
 banana,  
 nasa $\}$



## Erstellung des Tries:

```
1 def build_trie(P):
2     root = ACNode()
3     for (i, p) in enumerate(P):
4         node = root
5         for c in p:
6             if c not in node.targets:
7                 node.targets[c] = ACNode()
8             node = node.targets[c]
9         node.out.append(i)
10    return root
```

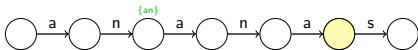
$P = \{$ an,  
ananas,  
anna,  
banana,  
nasa $\}$



## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8                 node = node.targets[c]  
9         node.out.append(i)  
10    return root
```

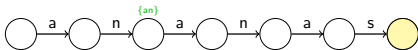
$P = \{$ an,  
 ananas,  
 anna,  
 banana,  
 nasa $\}$



## Erstellung des Tries:

```
1 def build_trie(P):
2     root = ACNode()
3     for (i, p) in enumerate(P):
4         node = root
5         for c in p:
6             if c not in node.targets:
7                 node.targets[c] = ACNode()
8             node = node.targets[c]
9         node.out.append(i)
10    return root
```

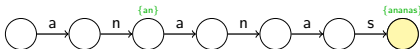
$P = \{$ an,  
ananas,  
anna,  
banana,  
nasa $\}$



## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8             node = node.targets[c]  
9         node.out.append(i)  
10    return root
```

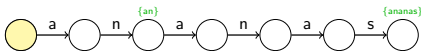
$P = \{\text{an},$   
**ananas**,  
anna,  
banana,  
nasa}



## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8                 node = node.targets[c]  
9         node.out.append(i)  
10    return root
```

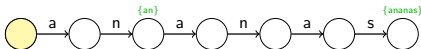
$P = \{$   
  an,  
  ananas,  
  **anna**,  
  banana,  
  nasa  
 $\}$



## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8             node = node.targets[c]  
9         node.out.append(i)  
10    return root
```

$P = \{$   
  an,  
  ananas,  
  anna,  
  banana,  
  nasa  
 $\}$

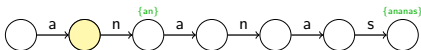




## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8             node = node.targets[c]  
9         node.out.append(i)  
10    return root
```

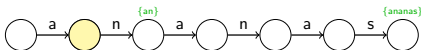
$P = \{\text{an, ananas, anna, banana, nasa}\}$



## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8                 node = node.targets[c]  
9         node.out.append(i)  
10    return root
```

$P = \{$   
  an,  
  ananas,  
  anna,  
  banana,  
  nasa  
 $\}$

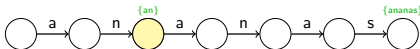


# Der Aho-Corasick-Algorithmus

## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8             node = node.targets[c]  
9         node.out.append(i)  
10    return root
```

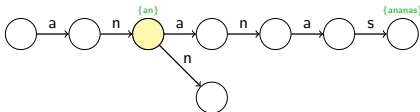
$P = \{$   
  an,  
  ananas,  
  **anna**,  
  banana,  
  nasa  
 $\}$



## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8                 node = node.targets[c]  
9         node.out.append(i)  
10    return root
```

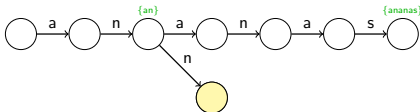
$P = \{$   
  an,  
  ananas,  
  anna,  
  banana,  
  nasa $\}$



## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8             node = node.targets[c]  
9         node.out.append(i)  
10    return root
```

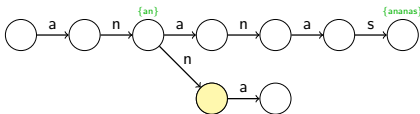
$P = \{$   
  an,  
  ananas,  
  **anna**,  
  banana,  
  nasa  
 $\}$



## Erstellung des Tries:

```
1 def build_trie(P):
2     root = ACNode()
3     for (i, p) in enumerate(P):
4         node = root
5         for c in p:
6             if c not in node.targets:
7                 node.targets[c] = ACNode()
8                 node = node.targets[c]
9         node.out.append(i)
10    return root
```

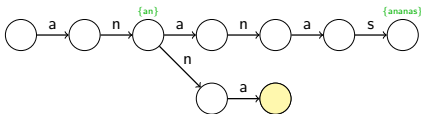
$P = \{$   
  an,  
  ananas,  
  anna,  
  banana,  
  nasa $\}$



## Erstellung des Tries:

```
1 def build_trie(P):
2     root = ACNode()
3     for (i, p) in enumerate(P):
4         node = root
5         for c in p:
6             if c not in node.targets:
7                 node.targets[c] = ACNode()
8             node = node.targets[c]
9         node.out.append(i)
10    return root
```

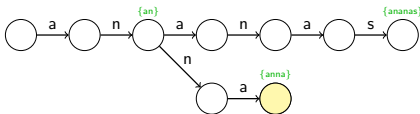
$P = \{\text{an, ananas, anna, banana, nasa}\}$



## Erstellung des Tries:

```
1 def build_trie(P):
2     root = ACNode()
3     for (i, p) in enumerate(P):
4         node = root
5         for c in p:
6             if c not in node.targets:
7                 node.targets[c] = ACNode()
8             node = node.targets[c]
9         node.out.append(i)
10    return root
```

$P = \{\text{an, ananas, anna, banana, nasa}\}$

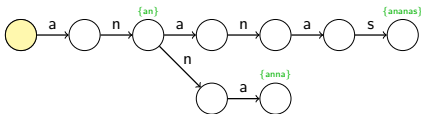




## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8                 node = node.targets[c]  
9         node.out.append(i)  
10    return root
```

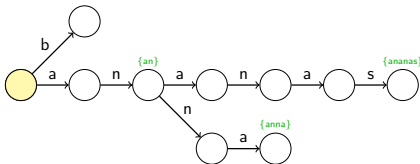
$P = \{\text{an, ananas, anna, banana, nasa}\}$



## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8                 node = node.targets[c]  
9         node.out.append(i)  
10    return root
```

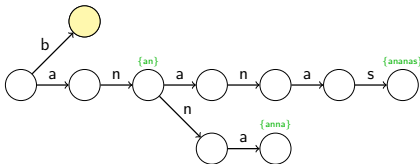
$P = \{\text{an, ananas, anna, banana, nasa}\}$



## Erstellung des Tries:

```
1 def build_trie(P):
2     root = ACNode()
3     for (i, p) in enumerate(P):
4         node = root
5         for c in p:
6             if c not in node.targets:
7                 node.targets[c] = ACNode()
8             node = node.targets[c]
9         node.out.append(i)
10    return root
```

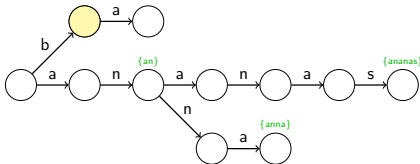
$P = \{\text{an, ananas, anna, banana, nasa}\}$



## Erstellung des Tries:

```
1 def build_trie(P):
2     root = ACNode()
3     for (i, p) in enumerate(P):
4         node = root
5         for c in p:
6             if c not in node.targets:
7                 node.targets[c] = ACNode()
8             node = node.targets[c]
9         node.out.append(i)
10    return root
```

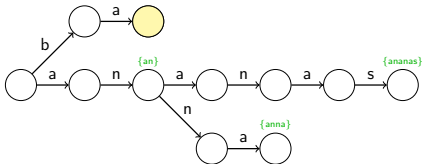
$P = \{\text{an, ananas, anna, banana, nasa}\}$



## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8                 node = node.targets[c]  
9         node.out.append(i)  
10    return root
```

$P = \{\text{an, ananas, anna, banana, nasa}\}$

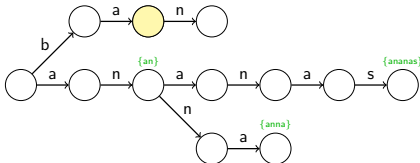


# Der Aho-Corasick-Algorithmus

## Erstellung des Tries:

```
1 def build_trie(P):
2     root = ACNode()
3     for (i, p) in enumerate(P):
4         node = root
5         for c in p:
6             if c not in node.targets:
7                 node.targets[c] = ACNode()
8             node = node.targets[c]
9         node.out.append(i)
10    return root
```

$P = \{\text{an},$   
 $\text{ananas},$   
 $\text{anna},$   
 $\text{banana},$   
 $\text{nasa}\}$

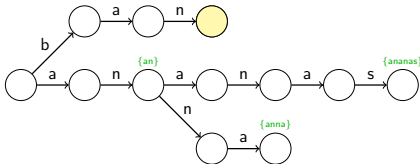


# Der Aho-Corasick-Algorithmus

## Erstellung des Tries:

```
1 def build_trie(P):
2     root = ACNode()
3     for (i, p) in enumerate(P):
4         node = root
5         for c in p:
6             if c not in node.targets:
7                 node.targets[c] = ACNode()
8             node = node.targets[c]
9         node.out.append(i)
10    return root
```

$P = \{\text{an, ananas, anna, banana, nasa}\}$

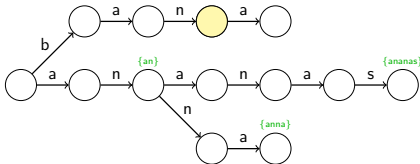


# Der Aho-Corasick-Algorithmus

## Erstellung des Tries:

```
1 def build_trie(P):
2     root = ACNode()
3     for (i, p) in enumerate(P):
4         node = root
5         for c in p:
6             if c not in node.targets:
7                 node.targets[c] = ACNode()
8                 node = node.targets[c]
9         node.out.append(i)
10    return root
```

$P = \{\text{an, ananas, anna, banana, nasa}\}$

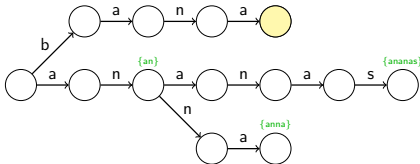




## Erstellung des Tries:

```
1 def build_trie(P):
2     root = ACNode()
3     for (i, p) in enumerate(P):
4         node = root
5         for c in p:
6             if c not in node.targets:
7                 node.targets[c] = ACNode()
8             node = node.targets[c]
9         node.out.append(i)
10    return root
```

$P = \{\text{an, ananas, anna, banana, nasa}\}$

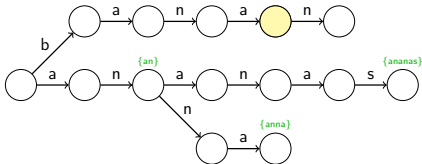


# Der Aho-Corasick-Algorithmus

## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8                 node = node.targets[c]  
9         node.out.append(i)  
10    return root
```

$P = \{\text{an, ananas, anna, banana, nasa}\}$

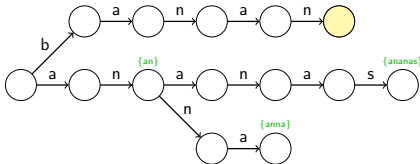


# Der Aho-Corasick-Algorithmus

## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8             node = node.targets[c]  
9         node.out.append(i)  
10    return root
```

$P = \{\text{an, ananas, anna, banana, nasa}\}$

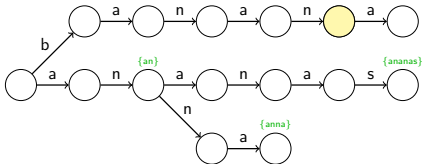


# Der Aho-Corasick-Algorithmus

## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8                 node = node.targets[c]  
9         node.out.append(i)  
10    return root
```

$P = \{\text{an, ananas, anna, banana, nasa}\}$

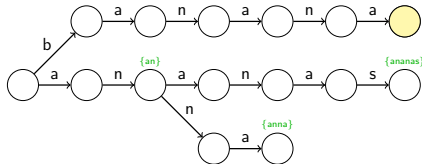


# Der Aho-Corasick-Algorithmus

## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8             node = node.targets[c]  
9         node.out.append(i)  
10    return root
```

$P = \{$   
  an,  
  ananas,  
  anna,  
  **banana**,  
  nasa  
 $\}$



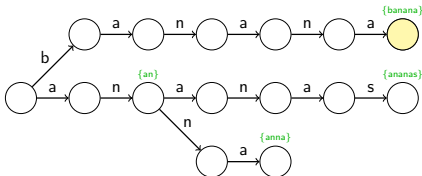
## Erstellung des Tries:

```

1 def build_trie(P):
2     root = ACNode()
3     for (i, p) in enumerate(P):
4         node = root
5         for c in p:
6             if c not in node.targets:
7                 node.targets[c] = ACNode()
8             node = node.targets[c]
9         node.out.append(i)
10    return root

```

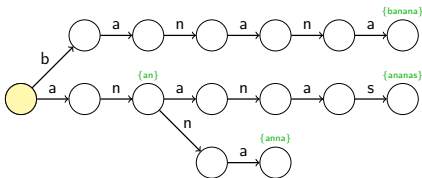
$P = \{$ 
 an,  
 ananas,  
 anna,  
 banana,  
 nasa $\}$



## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8                 node = node.targets[c]  
9         node.out.append(i)  
10    return root
```

$P = \{\text{an, ananas, anna, banana, nasa}\}$

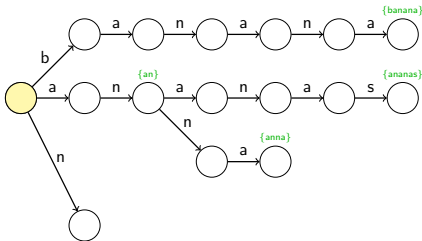


# Der Aho-Corasick-Algorithmus

## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8                 node = node.targets[c]  
9         node.out.append(i)  
10    return root
```

$P = \{\text{an, ananas, anna, banana, nasa}\}$

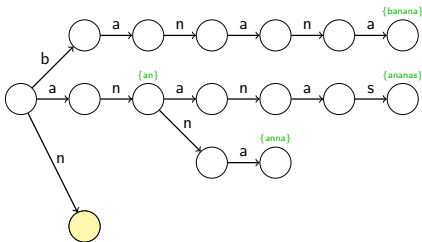




## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8                 node = node.targets[c]  
9         node.out.append(i)  
10    return root
```

$P = \{\text{an, ananas, anna, banana, nasa}\}$

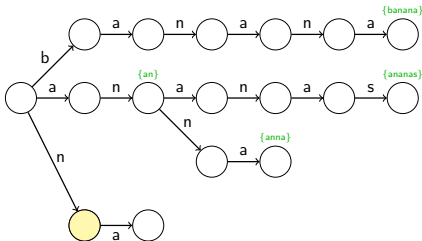


## Erstellung des Tries:

```

1 def build_trie(P):
2     root = ACNode()
3     for (i, p) in enumerate(P):
4         node = root
5         for c in p:
6             if c not in node.targets:
7                 node.targets[c] = ACNode()
8                 node = node.targets[c]
9         node.out.append(i)
10    return root
    
```

$P = \{$   
 an,  
 ananas,  
 anna,  
 banana,  
 nasa $\}$

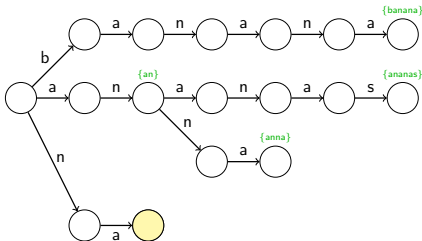


# Der Aho-Corasick-Algorithmus

## Erstellung des Tries:

```
1 def build_trie(P):
2     root = ACNode()
3     for (i, p) in enumerate(P):
4         node = root
5         for c in p:
6             if c not in node.targets:
7                 node.targets[c] = ACNode()
8             node = node.targets[c]
9         node.out.append(i)
10    return root
```

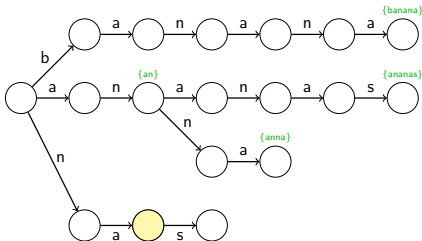
$P = \{\text{an, ananas, anna, banana, nasa}\}$



## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8                 node = node.targets[c]  
9         node.out.append(i)  
10    return root
```

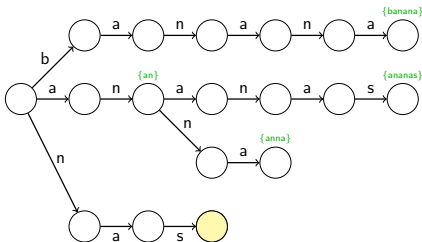
$P = \{\text{an, ananas, anna, banana, nasa}\}$



## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8             node = node.targets[c]  
9         node.out.append(i)  
10    return root
```

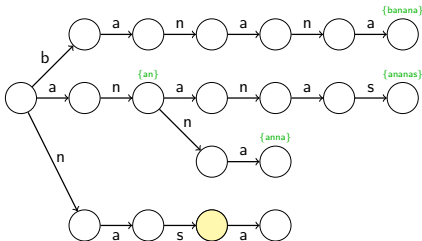
$P = \{\text{an, ananas, anna, banana, nasa}\}$



## Erstellung des Tries:

```
1 def build_trie(P):
2     root = ACNode()
3     for (i, p) in enumerate(P):
4         node = root
5         for c in p:
6             if c not in node.targets:
7                 node.targets[c] = ACNode()
8             node = node.targets[c]
9         node.out.append(i)
10    return root
```

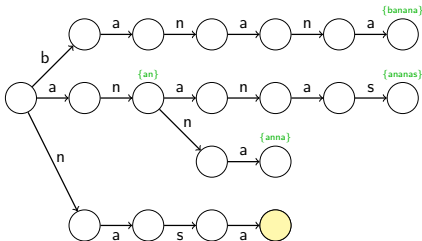
$P = \{\text{an, ananas, anna, banana, nasa}\}$



## Erstellung des Tries:

```
1 def build_trie(P):
2     root = ACNode()
3     for (i, p) in enumerate(P):
4         node = root
5         for c in p:
6             if c not in node.targets:
7                 node.targets[c] = ACNode()
8             node = node.targets[c]
9         node.out.append(i)
10    return root
```

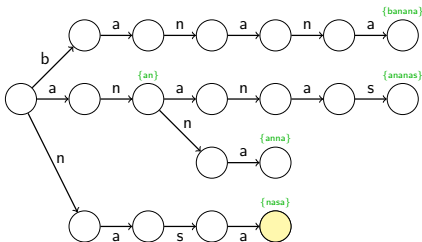
$P = \{\text{an, ananas, anna, banana, nasa}\}$



## Erstellung des Tries:

```
1 def build_trie(P):
2     root = ACNode()
3     for (i, p) in enumerate(P):
4         node = root
5         for c in p:
6             if c not in node.targets:
7                 node.targets[c] = ACNode()
8             node = node.targets[c]
9         node.out.append(i)
10    return root
```

$P = \{\text{an, ananas, anna, banana, nasa}\}$

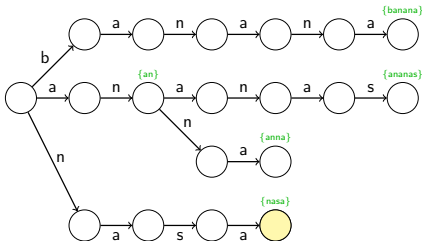




## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8                 node = node.targets[c]  
9         node.out.append(i)  
10    return root
```

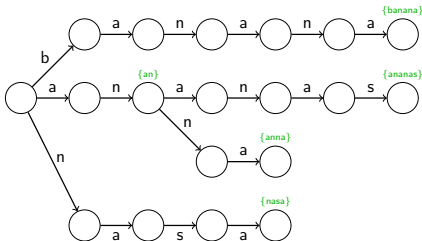
$P = \{\text{an, ananas, anna, banana, nasa}\}$



## Erstellung des Tries:

```
1 def build_trie(P):  
2     root = ACNode()  
3     for (i, p) in enumerate(P):  
4         node = root  
5         for c in p:  
6             if c not in node.targets:  
7                 node.targets[c] = ACNode()  
8                 node = node.targets[c]  
9     node.out.append(i)  
10    return root
```

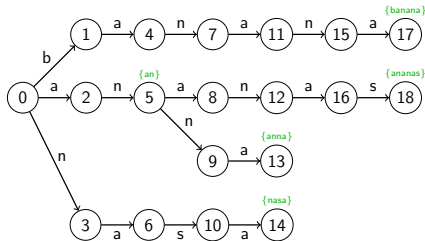
$P = \{$   
  an,  
  ananas,  
  anna,  
  banana,  
  nasa  
 $\}$



## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

Q = []

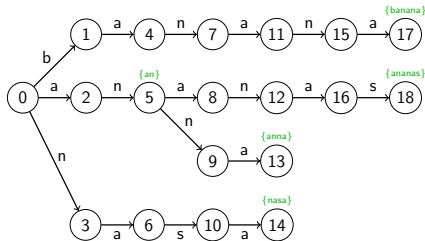


node =  
letter =  
parent =

## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

Q = [(0,  $\emptyset$ ,  $\emptyset$ )]

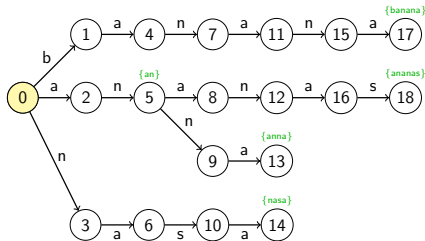


node =  
letter =  
parent =

## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

Q = []



node = 0

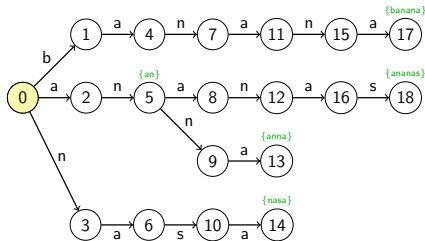
letter = {}

parent = {}

## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

Q = [(1, b, 0),  
(2, a, 0),  
(3, n, 0)]

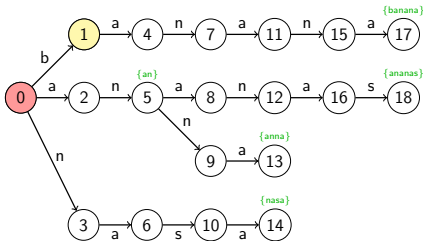


node = 0  
letter =  $\emptyset$   
parent =  $\emptyset$

## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

Q = [(2, a, 0),  
(3, n, 0)]

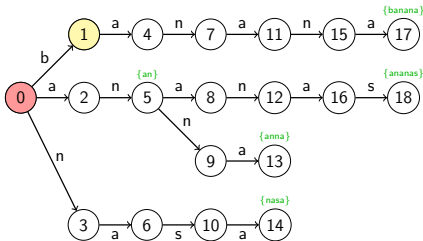


node = 1  
letter = b  
parent = 0

## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

Q = [(2, a, 0),  
(3, n, 0),  
(4, a, 1)]



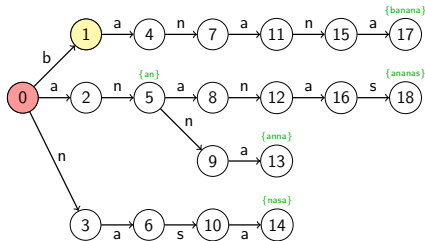
node = 1  
letter = b  
parent = 0



## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

Q = [(2, a, 0),  
(3, n, 0),  
(4, a, 1)]

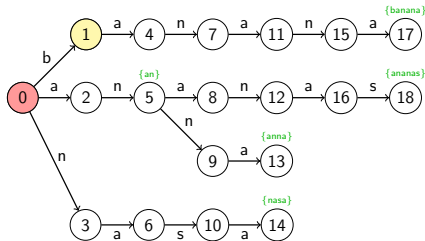


node = 1  
letter = b  
parent = 0

## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

Q = [(2, a, 0),  
(3, n, 0),  
(4, a, 1)]

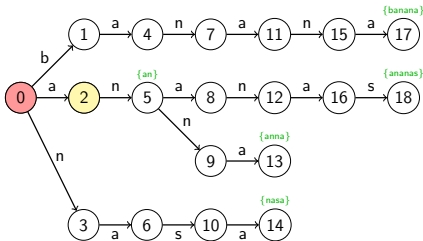


node = 1  
letter = b  
parent = 0

## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

$Q = [(3, n, 0),$   
 $(4, a, 1)]$

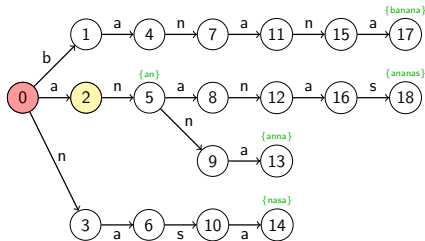


$node = 2$   
 $letter = a$   
 $parent = 0$

## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

$Q = [(3, n, 0),$   
 $(4, a, 1),$   
 $(5, n, 2)]$

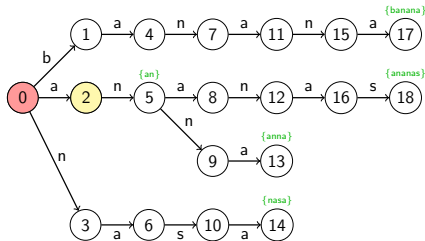


$node = 2$   
 $letter = a$   
 $parent = 0$

## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

$Q = [(3, n, 0),$   
 $(4, a, 1),$   
 $(5, n, 2)]$

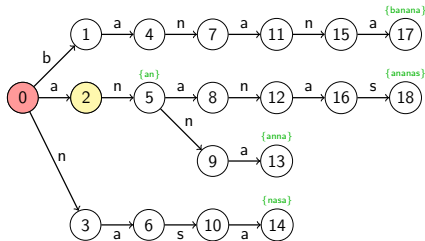


$node = 2$   
 $letter = a$   
 $parent = 0$

## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

$Q = [(3, n, 0),$   
 $(4, a, 1),$   
 $(5, n, 2)]$

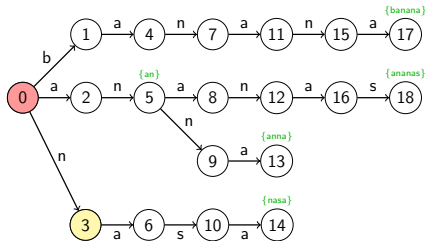


$node = 2$   
 $letter = a$   
 $parent = 0$

## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

Q = [(4, a, 1),  
(5, n, 2)]

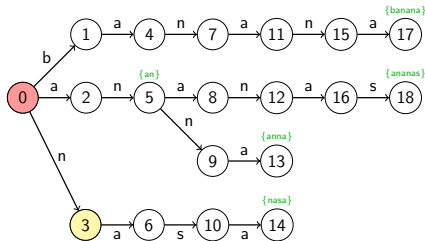


node = 3  
letter = n  
parent = 0

## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

Q = [(4, a, 1),  
(5, n, 2),  
(6, a, 3)]



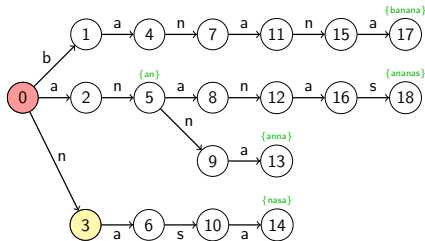
node = 3  
letter = n  
parent = 0



## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

Q = [(4, a, 1),  
(5, n, 2),  
(6, a, 3)]

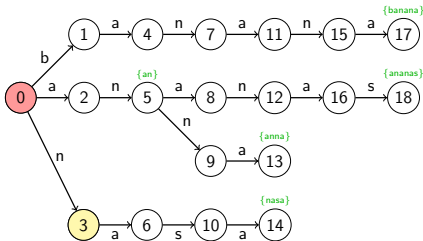


node = 3  
letter = n  
parent = 0

## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

Q = [(4, a, 1),  
(5, n, 2),  
(6, a, 3)]

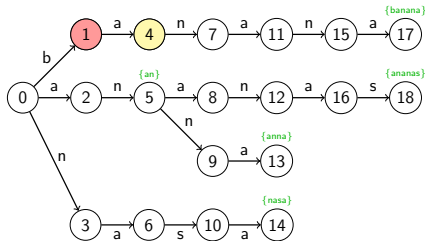


node = 3  
letter = n  
parent = 0

## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

$Q = [(5, n, 2),$   
 $(6, a, 3)]$

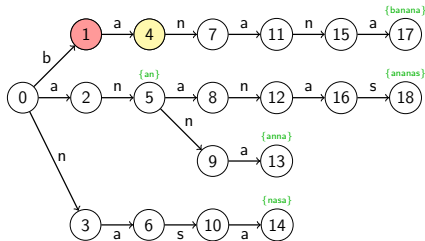


$node = 4$   
 $letter = a$   
 $parent = 1$

## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

$Q = [(5, n, 2),$   
 $(6, a, 3),$   
 $(7, n, 4)]$



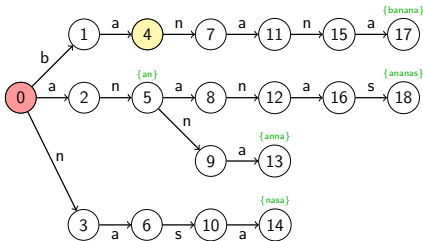
$node = 4$   
 $letter = a$   
 $parent = 1$

## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

$Q = [(5, n, 2),$   
 $(6, a, 3),$   
 $(7, n, 4)]$

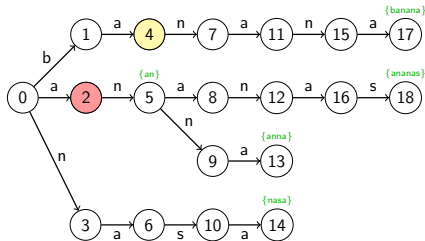
$node = 4$   
 $letter = a$   
 $parent = 1$



## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

$Q = [(5, n, 2),$   
 $(6, a, 3),$   
 $(7, n, 4)]$

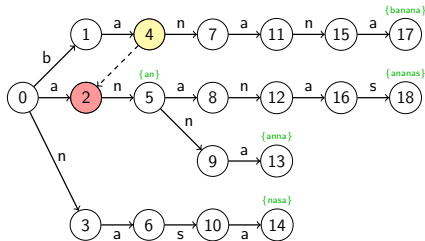


$node = 4$   
 $letter = a$   
 $parent = 1$

## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

$Q = [(5, n, 2),$   
 $(6, a, 3),$   
 $(7, n, 4)]$

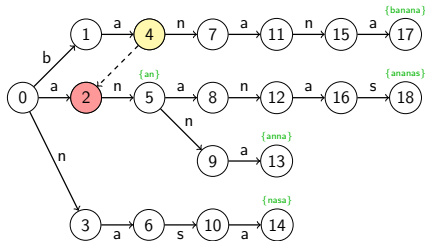


$node = 4$   
 $letter = a$   
 $parent = 1$

## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

$Q = [(5, n, 2),$   
 $(6, a, 3),$   
 $(7, n, 4)]$



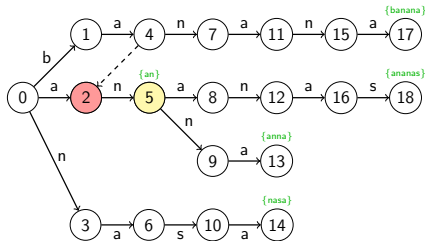
$node = 4$   
 $letter = a$   
 $parent = 1$



## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

Q = [(6, a, 3),  
(7, n, 4)]



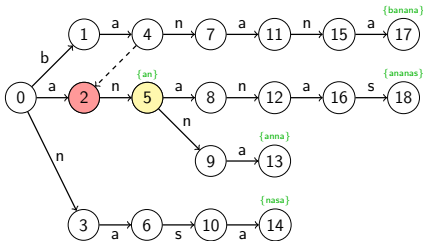
node = 5  
letter = n  
parent = 2

## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

Q = [(6, a, 3),  
(7, n, 4),  
(8, a, 5),  
(9, n, 5)]

node = 5  
letter = n  
parent = 2

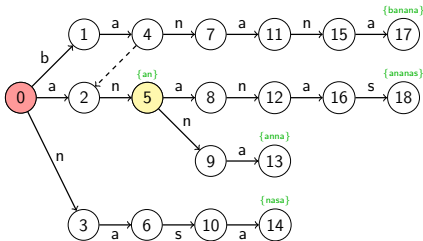


## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

Q = [(6, a, 3),  
(7, n, 4),  
(8, a, 5),  
(9, n, 5)]

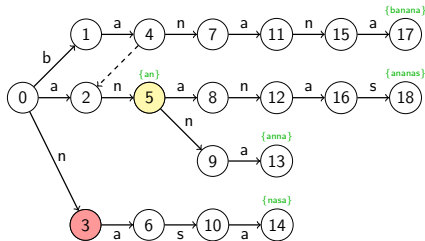
node = 5  
letter = n  
parent = 2



## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

Q = [(6, a, 3),  
(7, n, 4),  
(8, a, 5),  
(9, n, 5)]



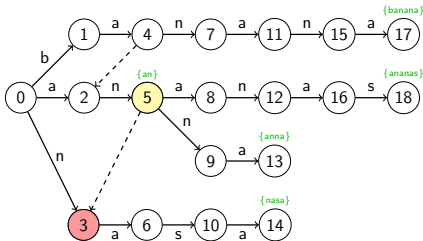
node = 5  
letter = n  
parent = 2

## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

Q = [(6, a, 3),  
(7, n, 4),  
(8, a, 5),  
(9, n, 5)]

node = 5  
letter = n  
parent = 2

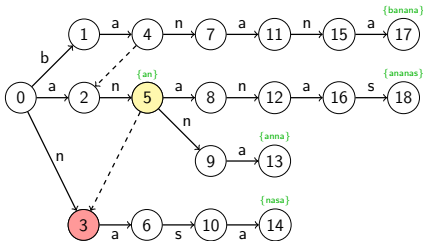


## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

Q = [(6, a, 3),  
(7, n, 4),  
(8, a, 5),  
(9, n, 5)]

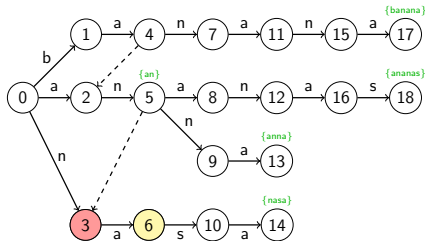
node = 5  
letter = n  
parent = 2



## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

$Q = [(7, n, 4),$   
 $(8, a, 5),$   
 $(9, n, 5)]$



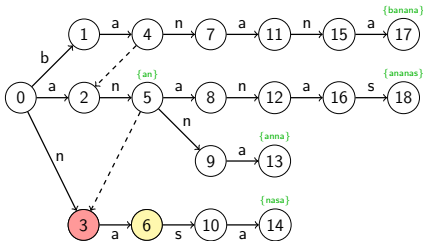
$node = 6$   
 $letter = a$   
 $parent = 3$

## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

Q = [(7, n, 4),  
(8, a, 5),  
(9, n, 5),  
(10, s, 6)]

node = 6  
letter = a  
parent = 3



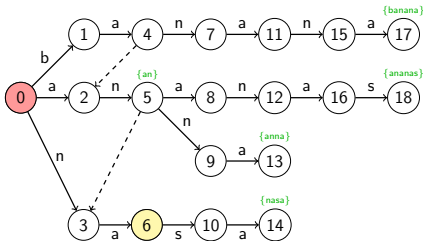


## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

Q = [(7, n, 4),  
(8, a, 5),  
(9, n, 5),  
(10, s, 6)]

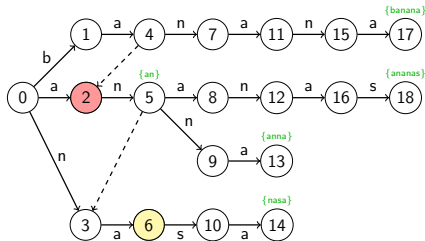
node = 6  
letter = a  
parent = 3



## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

$Q = [(7, n, 4),$   
 $(8, a, 5),$   
 $(9, n, 5),$   
 $(10, s, 6)]$



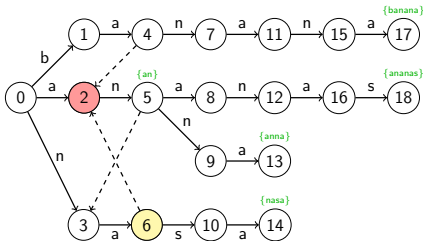
$node = 6$   
 $letter = a$   
 $parent = 3$

## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

Q = [(7, n, 4),  
(8, a, 5),  
(9, n, 5),  
(10, s, 6)]

node = 6  
letter = a  
parent = 3

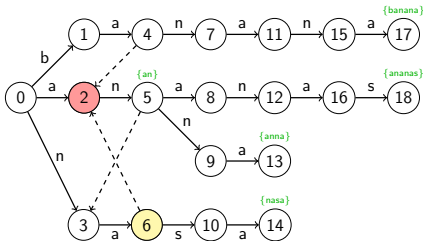


## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

Q = [(7, n, 4),  
(8, a, 5),  
(9, n, 5),  
(10, s, 6)]

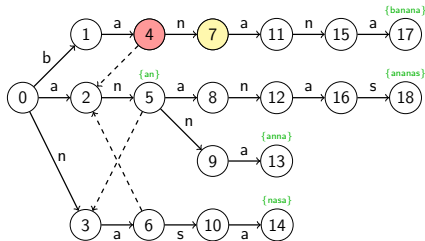
node = 6  
letter = a  
parent = 3



## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

Q = [(8, a, 5),  
(9, n, 5),  
(10, s, 6)]



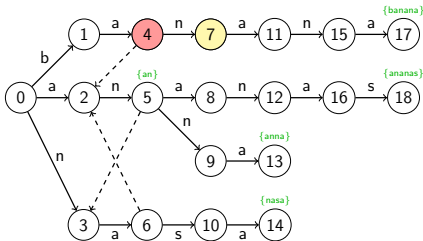
node = 7  
letter = n  
parent = 4

## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):  
2     Q = [(root, None, None)] # (child, letter to child, parent)  
3     while Q:  
4         (node, letter, parent) = Q.pop(0)  
5         Q.extend([(v, k, node) for k, v in node.targets.items()])  
6         if parent is None: continue  
7         node.lps = delta(parent.lps, letter) if parent != root else root  
8         node.out.extend(node.lps.out)
```

Q = [(8, a, 5),  
(9, n, 5),  
(10, s, 6),  
(11, a, 7)]

node = 7  
letter = n  
parent = 4

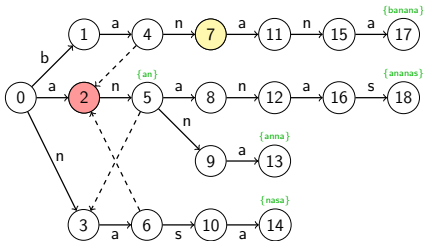


## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

Q = [(8, a, 5),  
(9, n, 5),  
(10, s, 6),  
(11, a, 7)]

node = 7  
letter = n  
parent = 4

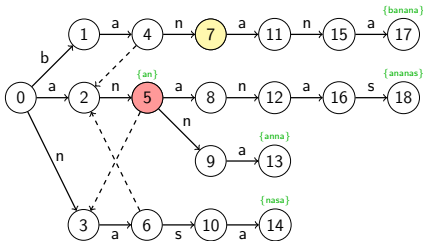


## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

Q = [(8, a, 5),  
(9, n, 5),  
(10, s, 6),  
(11, a, 7)]

node = 7  
letter = n  
parent = 4

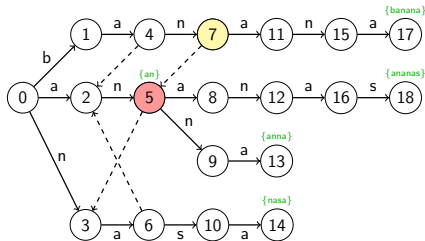




## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

Q = [(8, a, 5),  
(9, n, 5),  
(10, s, 6),  
(11, a, 7)]



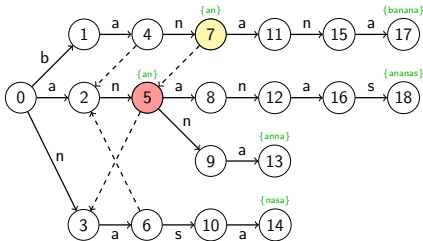
node = 7  
letter = n  
parent = 4

## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):
2     Q = [(root, None, None)] # (child, letter to child, parent)
3     while Q:
4         (node, letter, parent) = Q.pop(0)
5         Q.extend([(v, k, node) for k, v in node.targets.items()])
6         if parent is None: continue
7         node.lps = delta(parent.lps, letter) if parent != root else root
8         node.out.extend(node.lps.out)
```

Q = [(8, a, 5),  
(9, n, 5),  
(10, s, 6),  
(11, a, 7)]

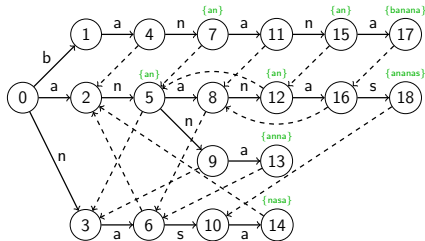
node = 7  
letter = n  
parent = 4



## Erstellung der *lps*-Kanten:

```
1 def build_lps(root):  
2     Q = [(root, None, None)] # (child, letter to child, parent)  
3     while Q:  
4         (node, letter, parent) = Q.pop(0)  
5         Q.extend([(v, k, node) for k, v in node.targets.items()])  
6         if parent is None: continue  
7         node.lps = delta(parent.lps, letter) if parent != root else root  
8         node.out.extend(node.lps.out)
```

Q =



node =  
letter =  
parent =

Mustersuche mit dem Aho-Corasick-Algorithmus:

```
1 def aho_corasick(P, T):  
2     q = build_AC(P)  
3     for (i, c) in enumerate(T):  
4         q = delta(q, c)  
5         for x in q.out:  
6             yield (i-len(P[x])+1, i+1, x)
```

- Wie beim KMP-Algorithmus wird die while-Schleife in der *delta*-Funktion amortisiert beim Erstellen des Automaten nur  $\mathcal{O}(m)$  und bei der Mustersuche nur  $\mathcal{O}(n)$  mal betreten.

- Wie beim KMP-Algorithmus wird die while-Schleife in der *delta*-Funktion amortisiert beim Erstellen des Automaten nur  $\mathcal{O}(m)$  und bei der Mustersuche nur  $\mathcal{O}(n)$  mal betreten.
- Der AC-Algorithmus braucht zum Erstellen des Automaten, für die Mustersuche und das Erkennen von Zuständen mit nichtleerer Ausgabe  $\mathcal{O}(m + n)$  Zeit.
- Die Ausgabe  $z$  kann pathologisch Größe  $\mathcal{O}(kn)$  haben, insgesamt  $\mathcal{O}(m + n + z)$  Zeit

- Der Aho-Corasick-Algorithmus ist eine Erweiterung des KMP-Algorithmus auf eine Menge von Mustern.
- Laufzeit:  $\mathcal{O}(m + n + z)$ ; dabei ist  $z$  die Größe der Ausgabe

- Der Aho-Corasick-Algorithmus ist eine Erweiterung des KMP-Algorithmus auf eine Menge von Mustern.
- Laufzeit:  $O(m + n + z)$ ; dabei ist  $z$  die Größe der Ausgabe
- Bei der gezeigten Implementierung (konkatenierte Ausgabelisten) beträgt der Speicherplatz  $O(n + km)$ , bei einer Implementierung mit Verweisen nur  $O(n + m)$ .