Tries

>> Wir haben bereits einige Datenstrukturen kennengelernt, die **Key-Value-Paare** verwalten, mit denen also **Dictionaries** implementiert werden können:

Arrays: Der Key ist der Index, der Value sind die Daten an dieser Position.

Hash Tables: Der Key ist der Hashwert der Daten, der Value ist eine verkettete Liste von Daten mit diesem Hashwert.

>> **Tries** (Aussprache wie bei „to try“) kombinieren Strukturen und Arrays, um Daten schnell zugreifbar zu speichern.

Die zu suchenden Daten (also den *Key*) nutzen wir als Wegbeschreibung durch eine baumartige Datenstruktur.

Wenn man der Wegbeschreibung von Anfang bis Ende folgen kann, existieren die Daten im Trie, sonst nicht.

Anders als bei einer Hash Table gibt es **keine Kollisionen**, und keine zwei Datensätze haben denselben Pfad (außer sie sind identisch)..

Beispiel: Universitäten anhand des Gründungsjahrs finden

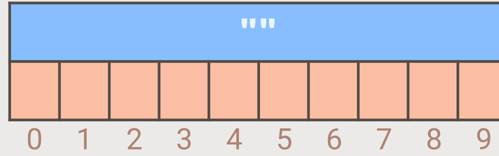
Wir wollen Key-Value-Paare speichern, wobei die **Keys vierstellige Jahreszahlen** (YYYY) sind und die **Values die Namen** von Universitäten, die in diesen Jahren gegründet wurden.

In einem Trie werden die Pfade vom zentralen Wurzelknoten zu den Blattknoten mit den Ziffern der Jahreszahl markiert.

Jeder Knoten auf dem Pfad von der Wurzel zu einem Blatt enthält ein Array mit zehn Pointern auf andere Knoten, einen pro Ziffer.

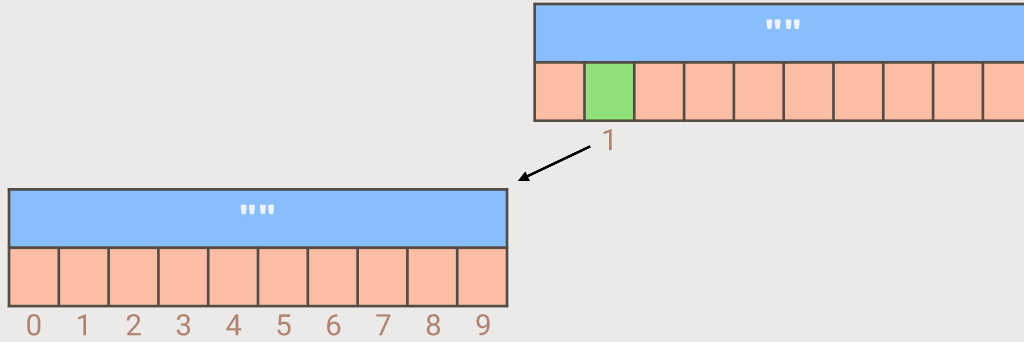
Struct für Trie im Beispiel

```c
typedef struct _trie
{
    char university[20];
    struct _trie* paths[10];
}
trie;
```



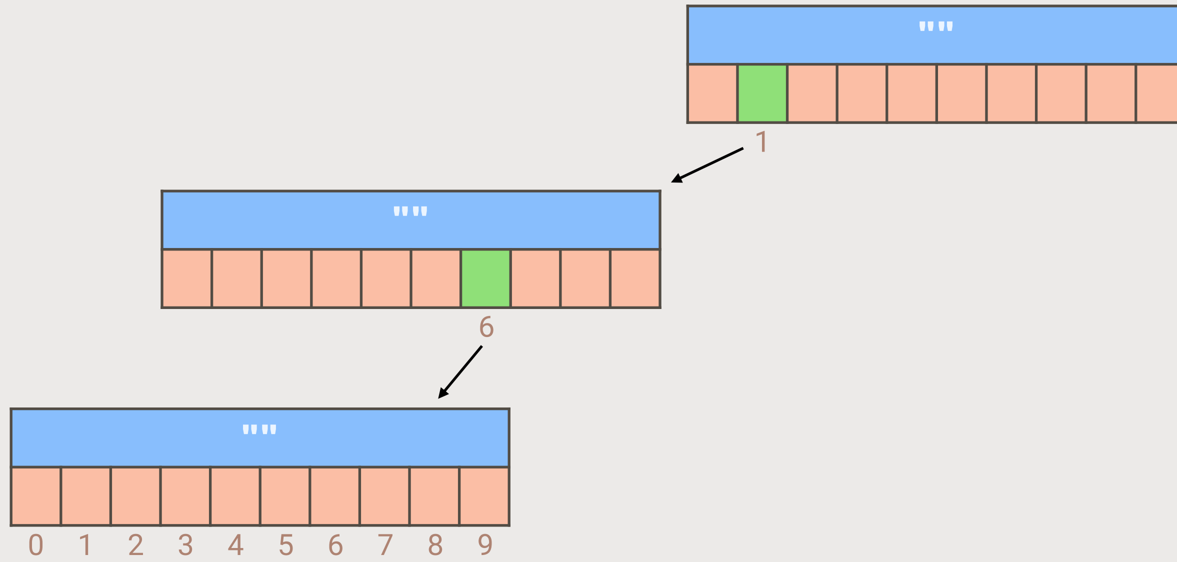
Name der Universität

Pointer-Array für je 10 Ziffern

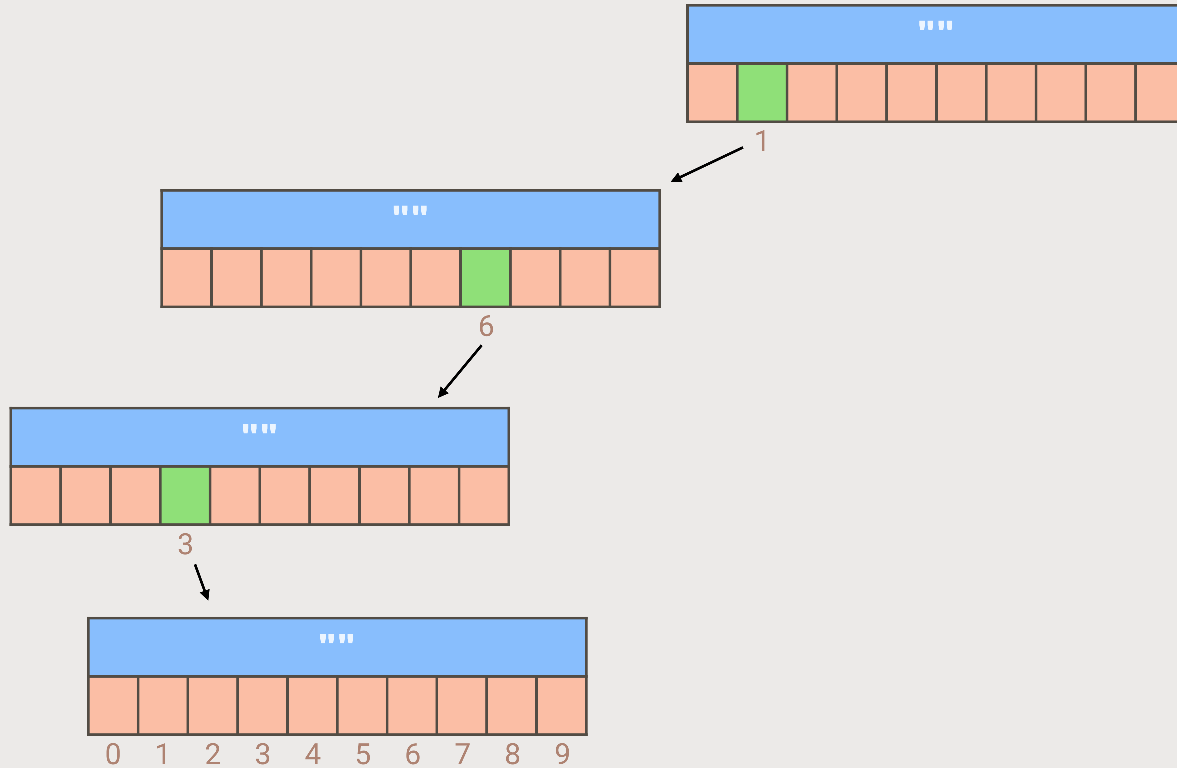


Harvard einfügen
(Gründung: 1636)

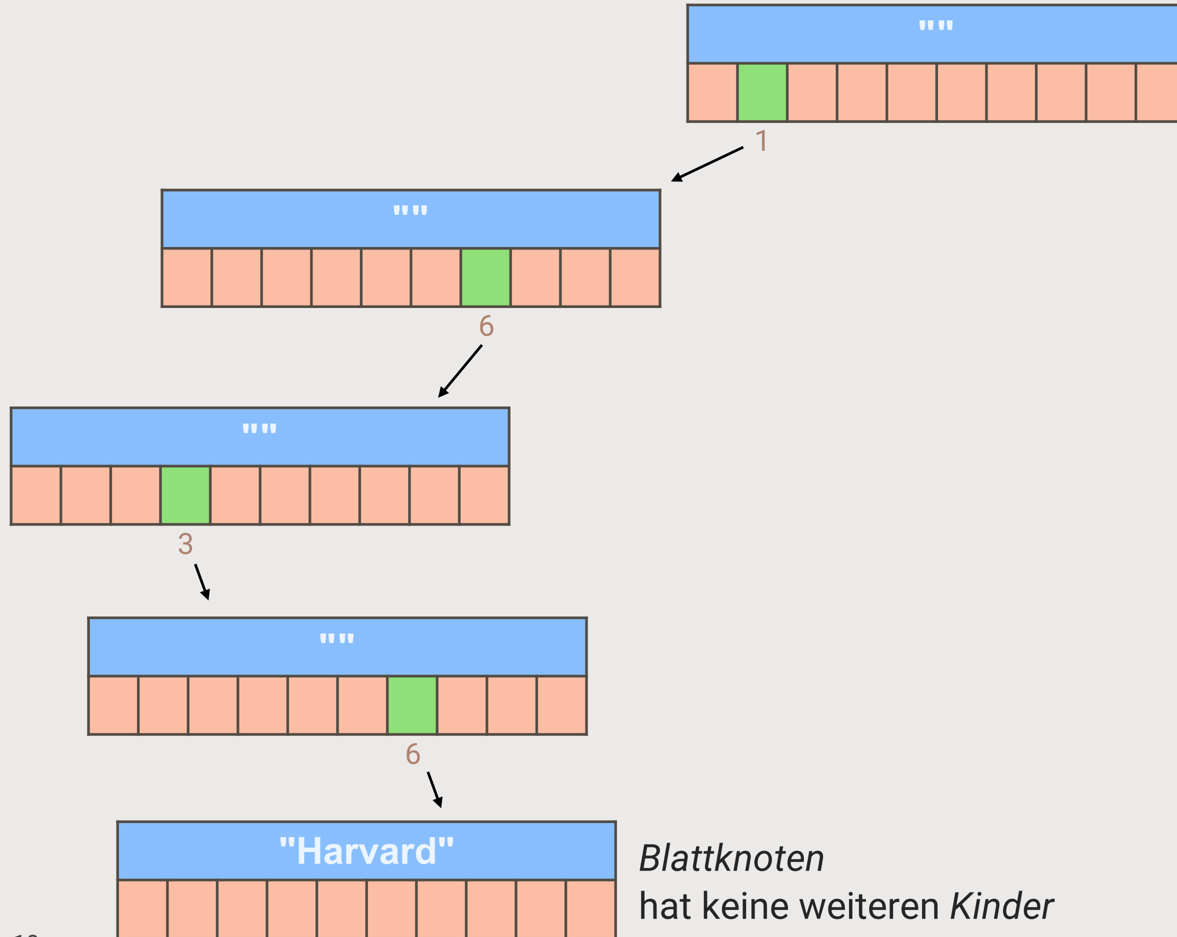
Harvard einfügen
(Gründung: 1636)



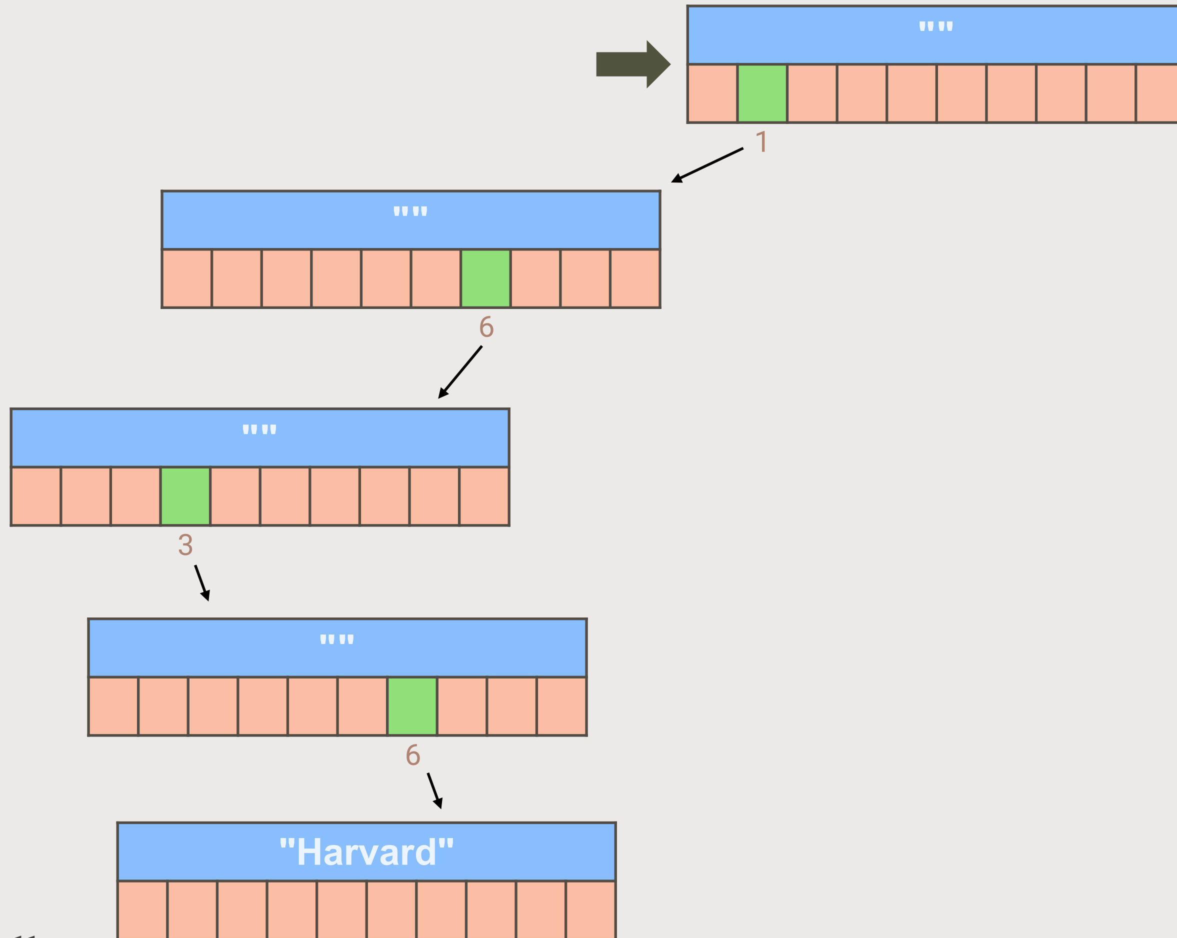
Harvard einfügen (Gründung: 1636)



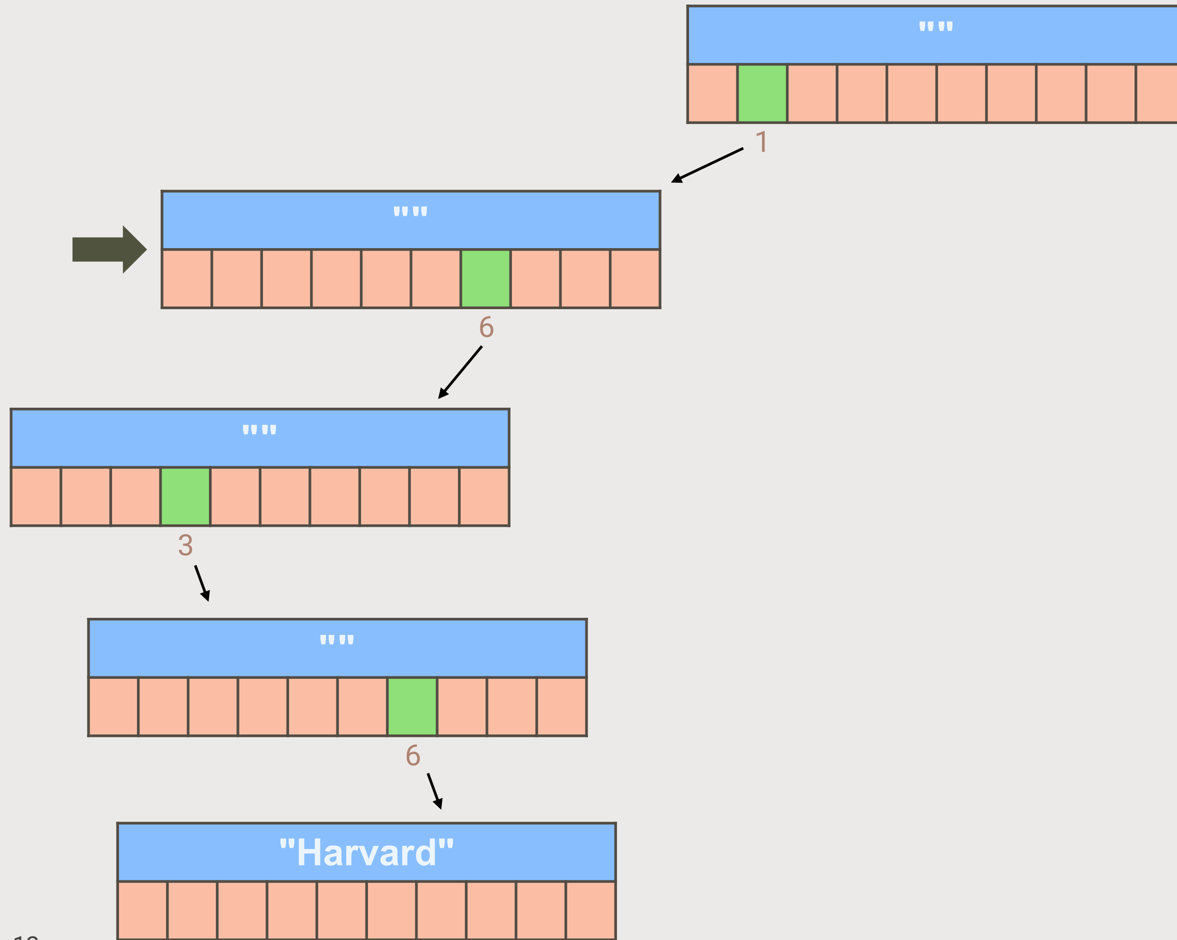
Harvard einfügen
(Gründung: 1636)



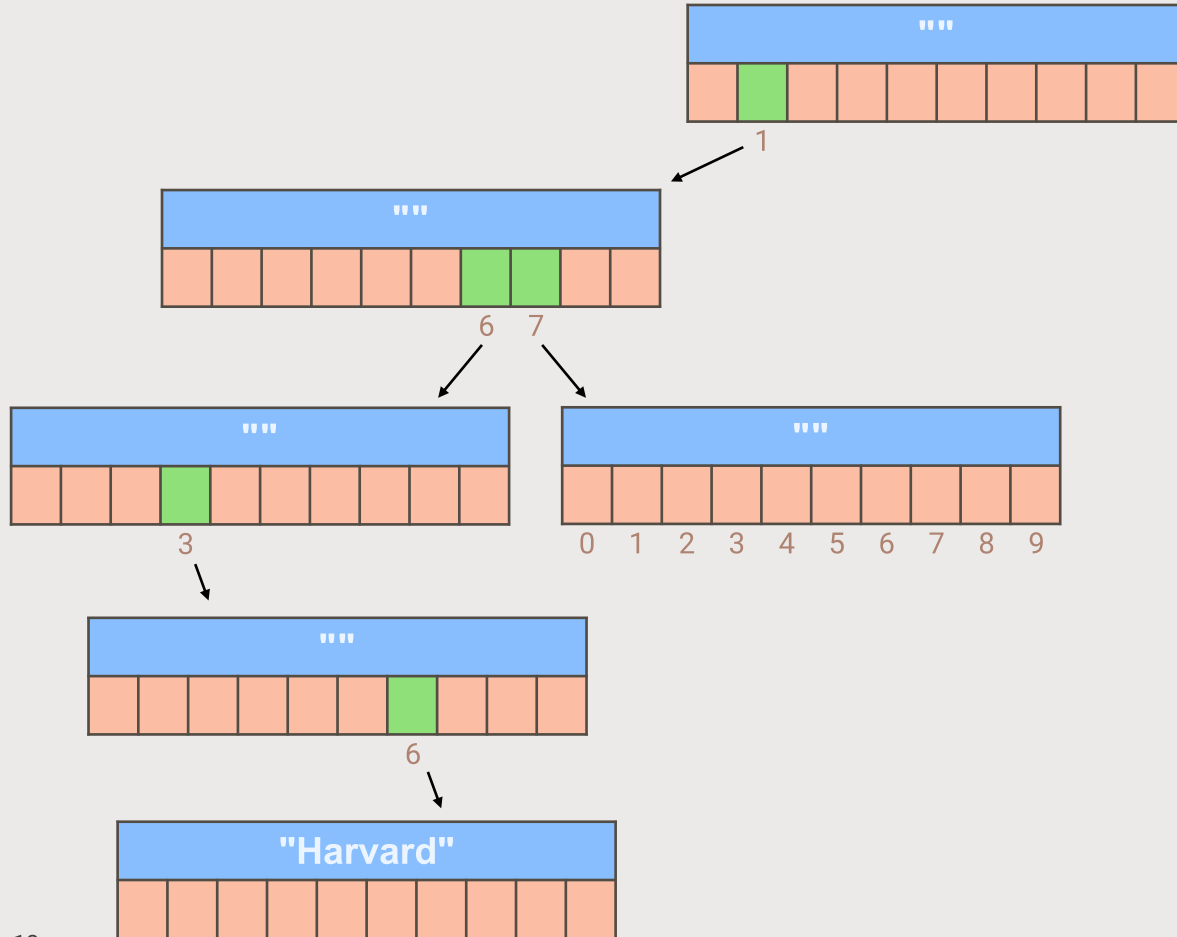
Blattknoten
hat keine weiteren *Kinder*



Yale einfügen
(Gründung: 1701)

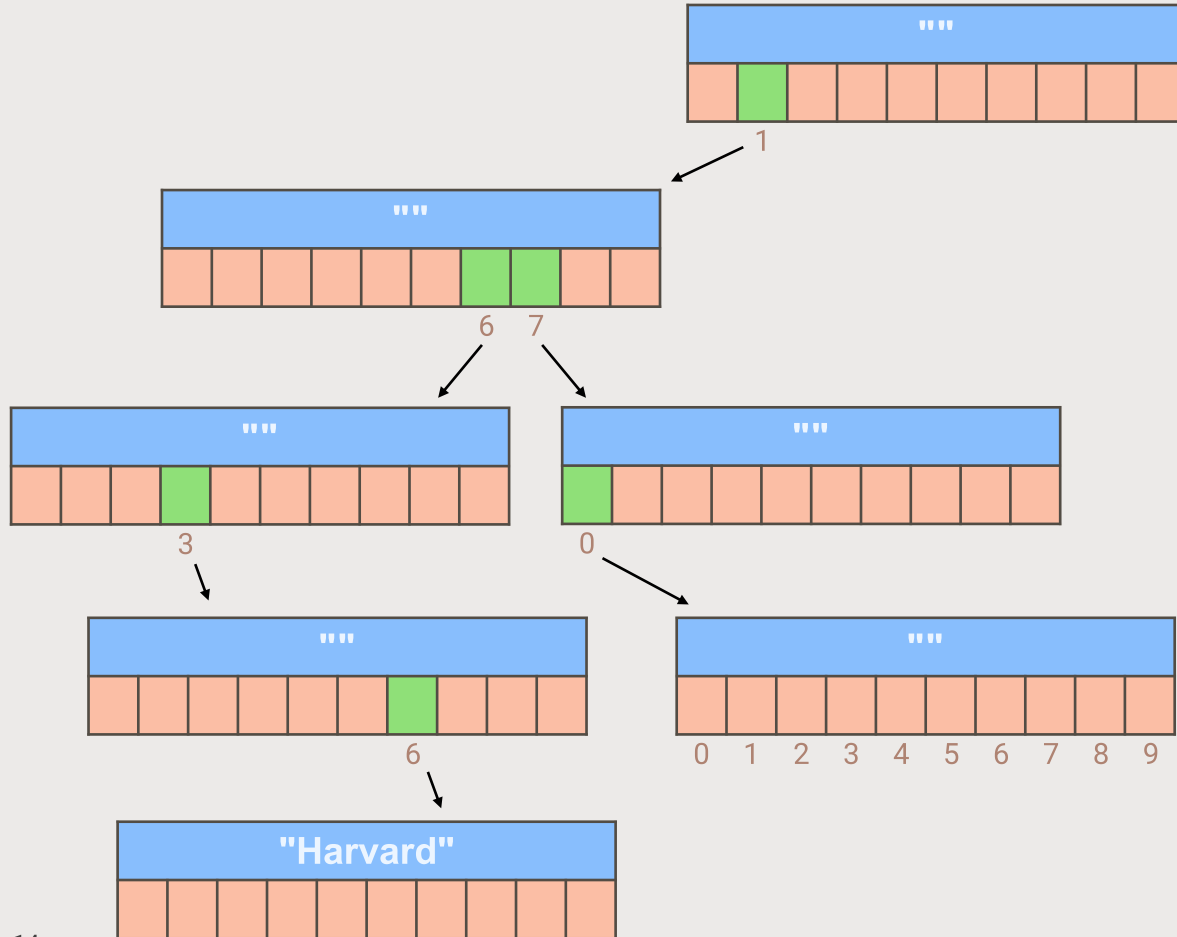


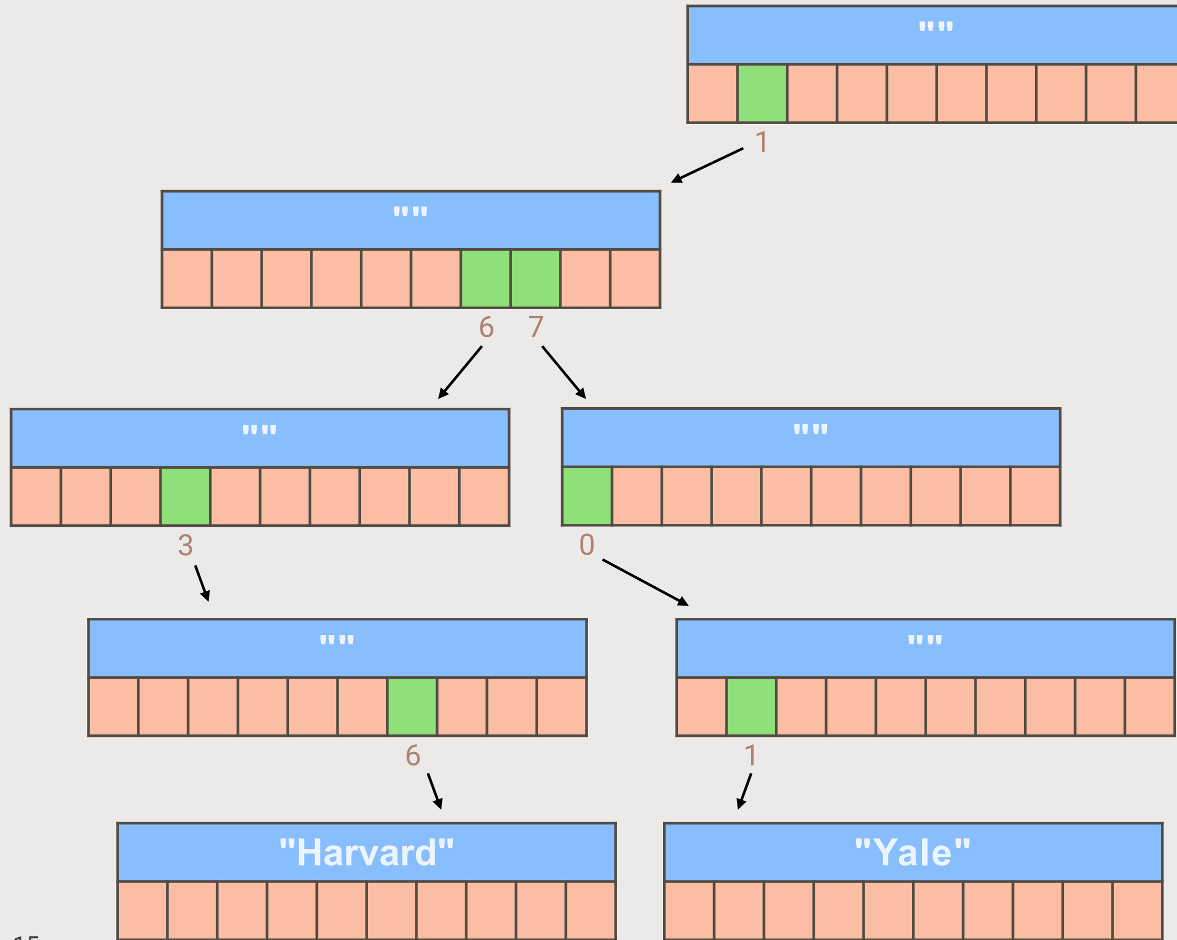
Yale einfügen
(Gründung: 1701)



Yale einfügen
(Gründung: 1701)

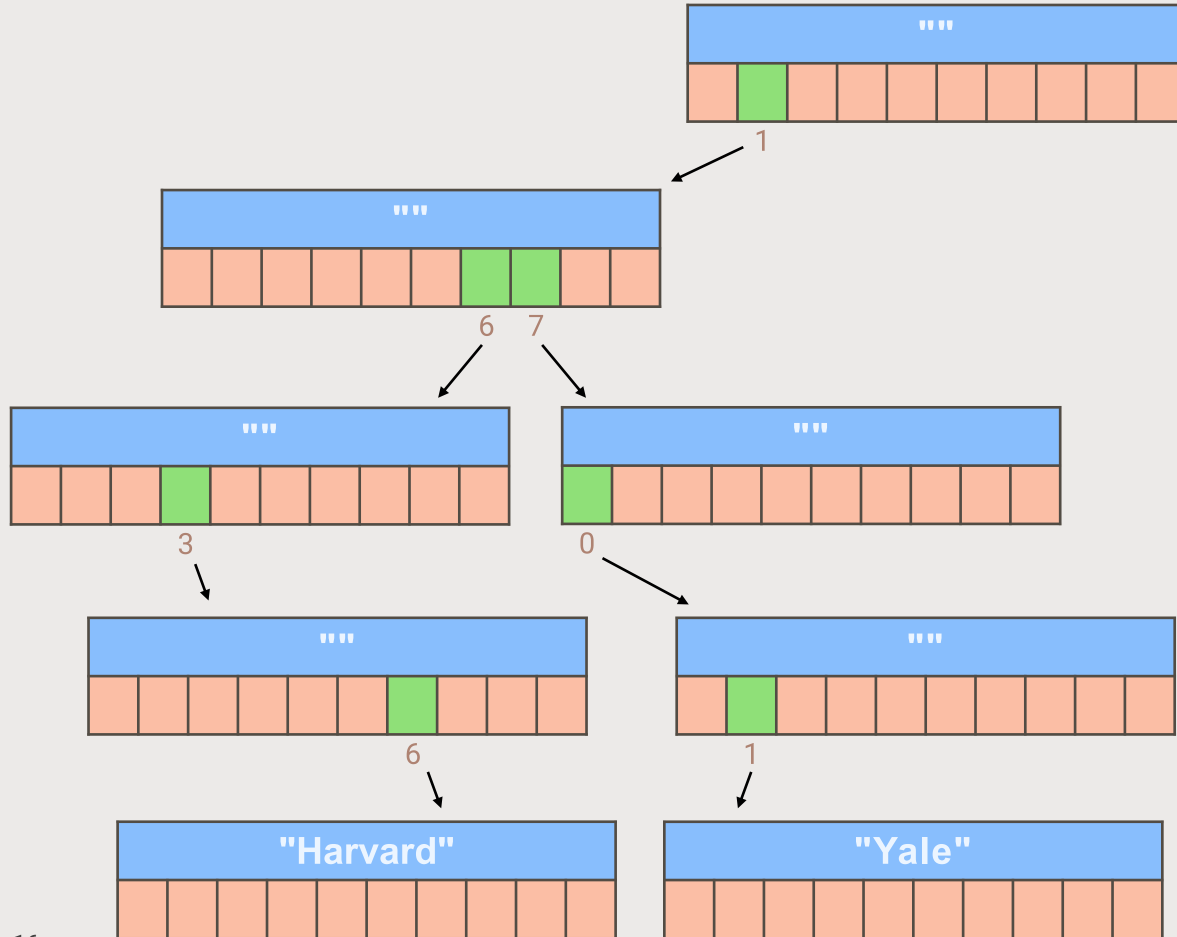
Yale einfügen
(Gründung: 1701)

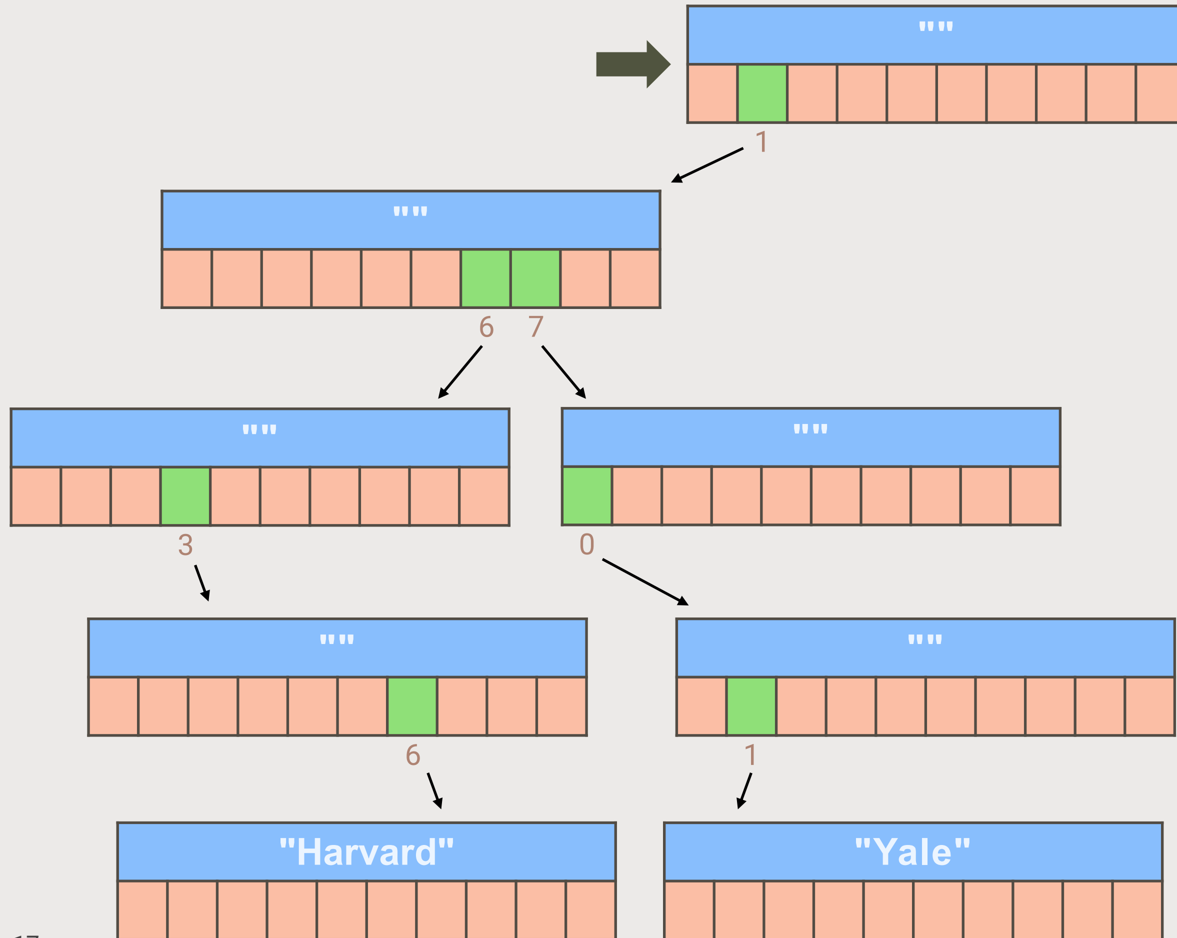




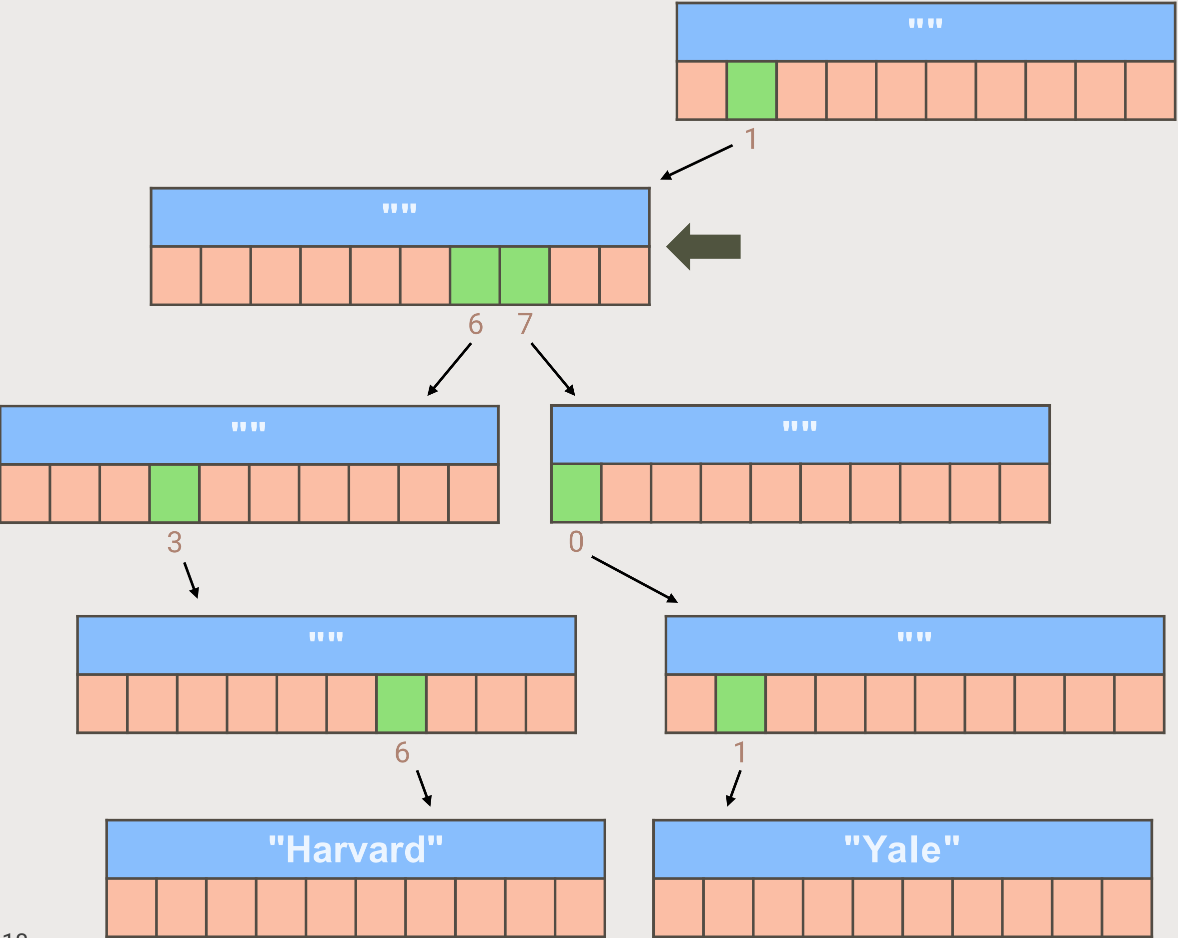
Yale einfügen
(Gründung: 1701)

Dartmouth einfügen
(Gründung: 1769)

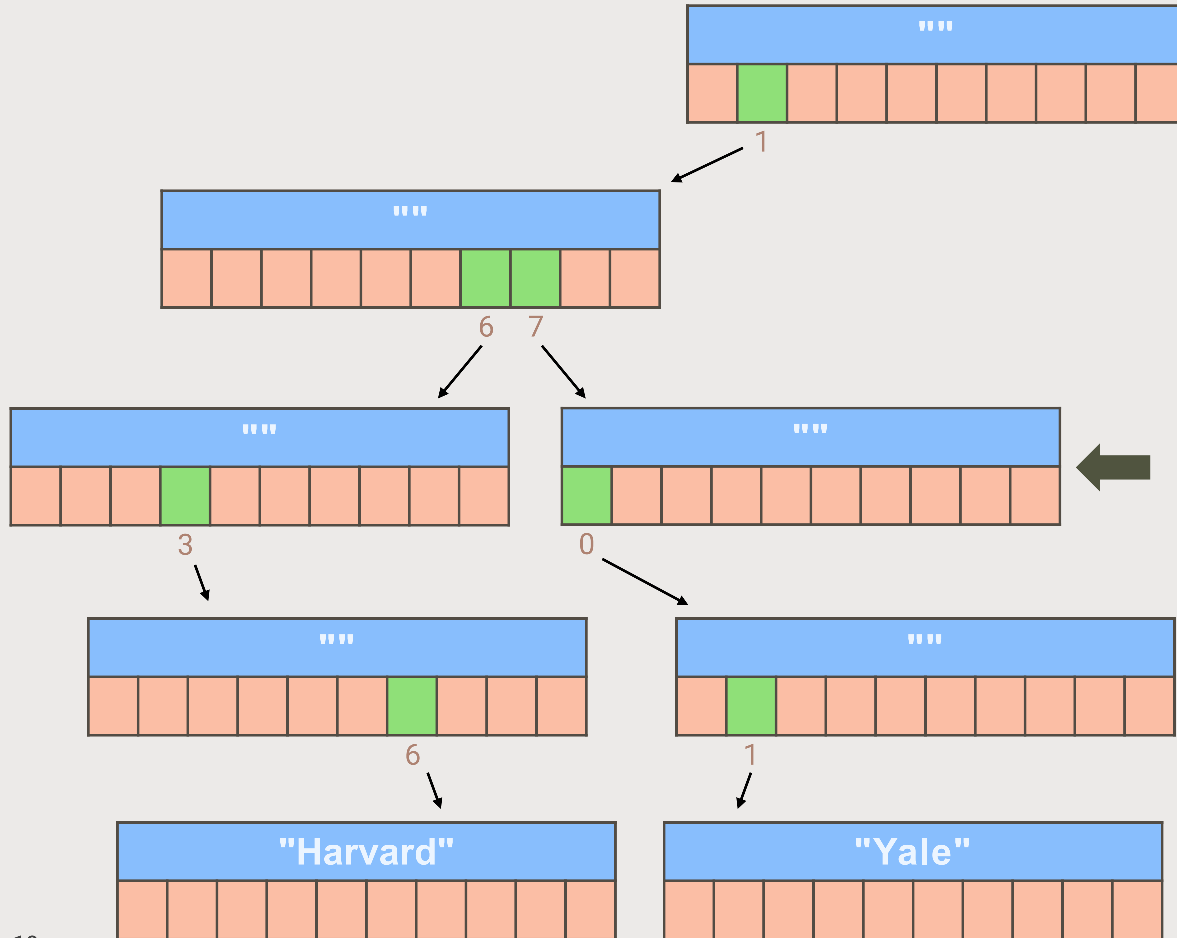




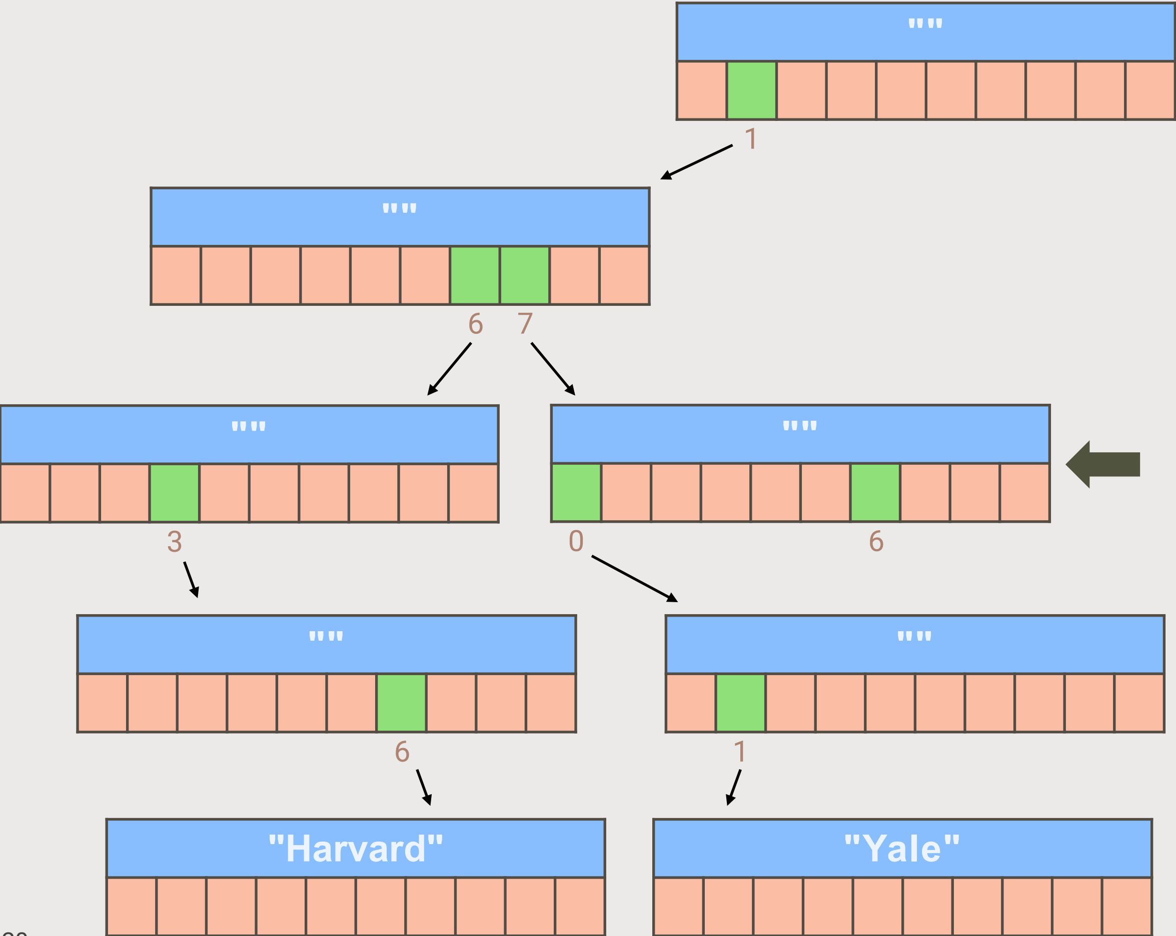
Dartmouth einfügen
(Gründung: 1769)



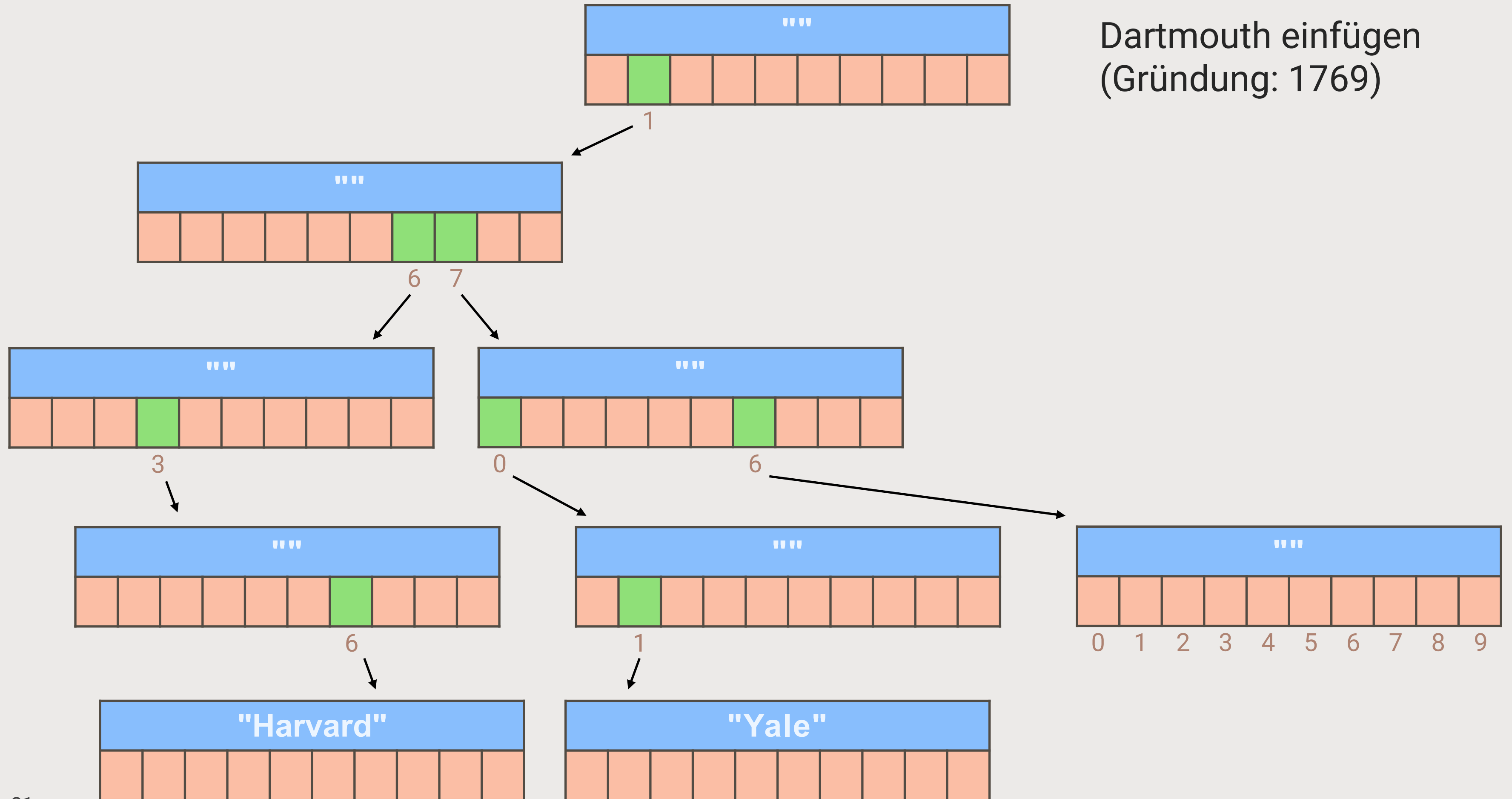
Dartmouth einfügen
(Gründung: 1769)

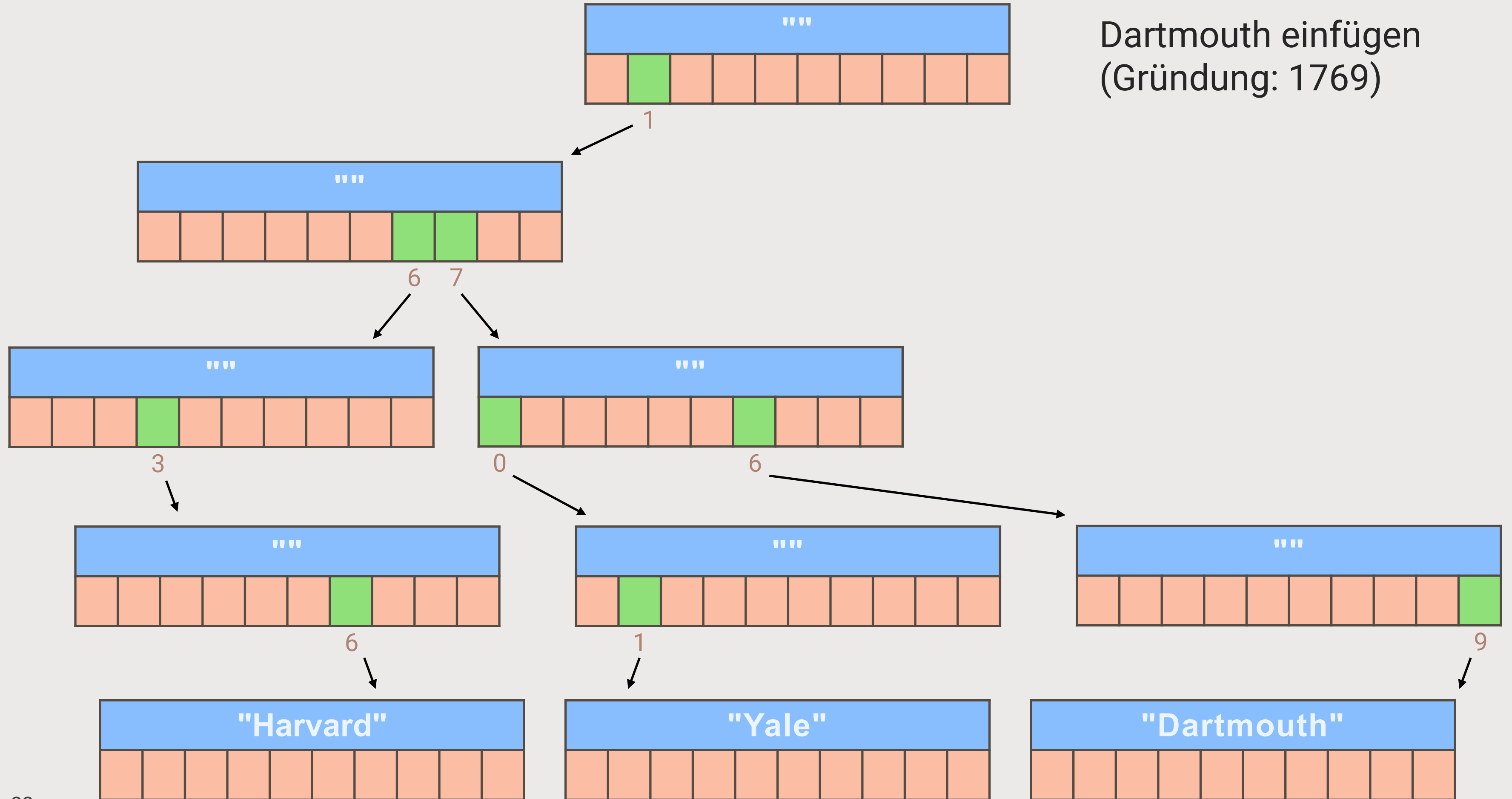


Dartmouth einfügen
(Gründung: 1769)



Dartmouth einfügen
(Gründung: 1769)


Dartmouth einfügen
(Gründung: 1769)
""
1
""
6
7
""
3
""
0
6
""
6
""
1
""
0 1 2 3 4 5 6 7 8 9
"Harvard"
"Yale"

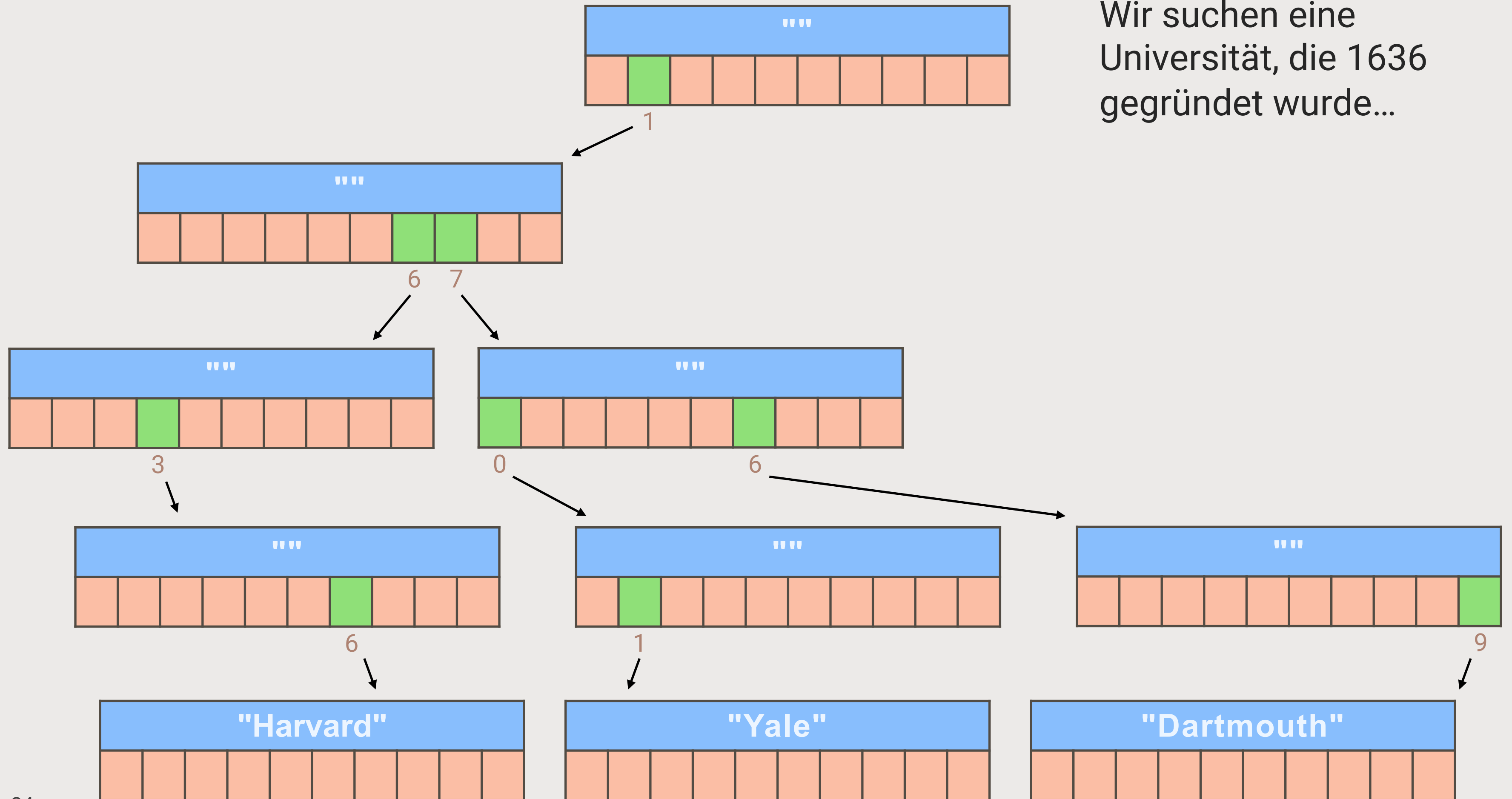

Dartmouth einfügen
(Gründung: 1769)
""
1
""
6
7
""
""
3
0
6
""
""
""
6
1
9
"Harvard"
"Yale"
"Dartmouth"

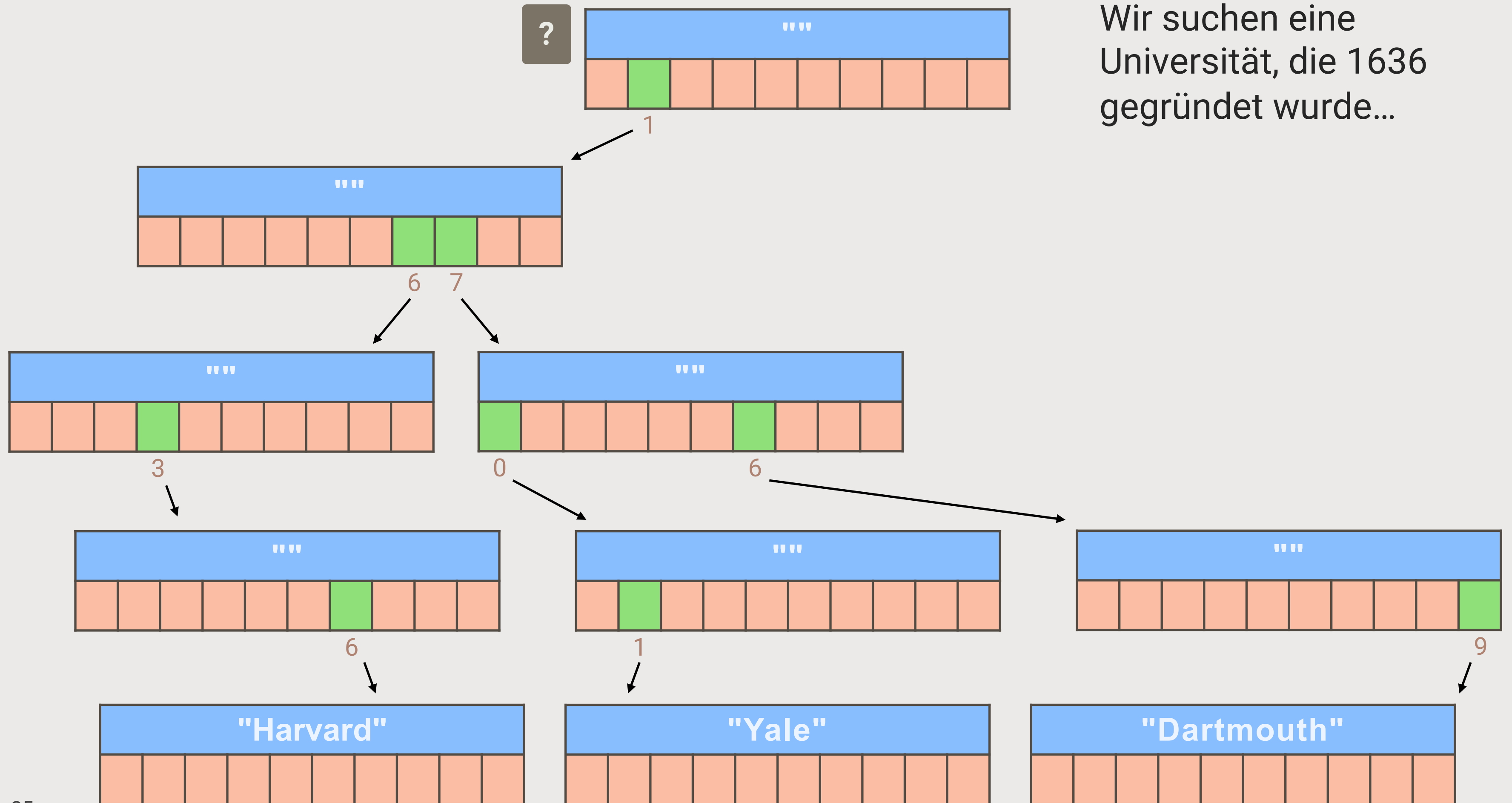
Beispiel: Suche nach einem Element

Um in unserem Trie nach einem Eintrag zu suchen, verwenden wir die Ziffern des Keys (= Jahreszahl), um von der Wurzel aus zu einem Blattknoten zu navigieren. Wenn einen Blattknoten finden, ohne in einer Sackgasse (NULL-Pointer) zu enden, haben wir den Eintrag gefunden, sonst existiert er nicht.

Wir werden sehen: Die Suche hat Laufzeit-komplexität O(1). Die Anzahl der Einträge im Trie spielt auch im Worst Case keine Rolle dafür, wie lange es dauert, einen bestimmten Eintrag *zu finden.*

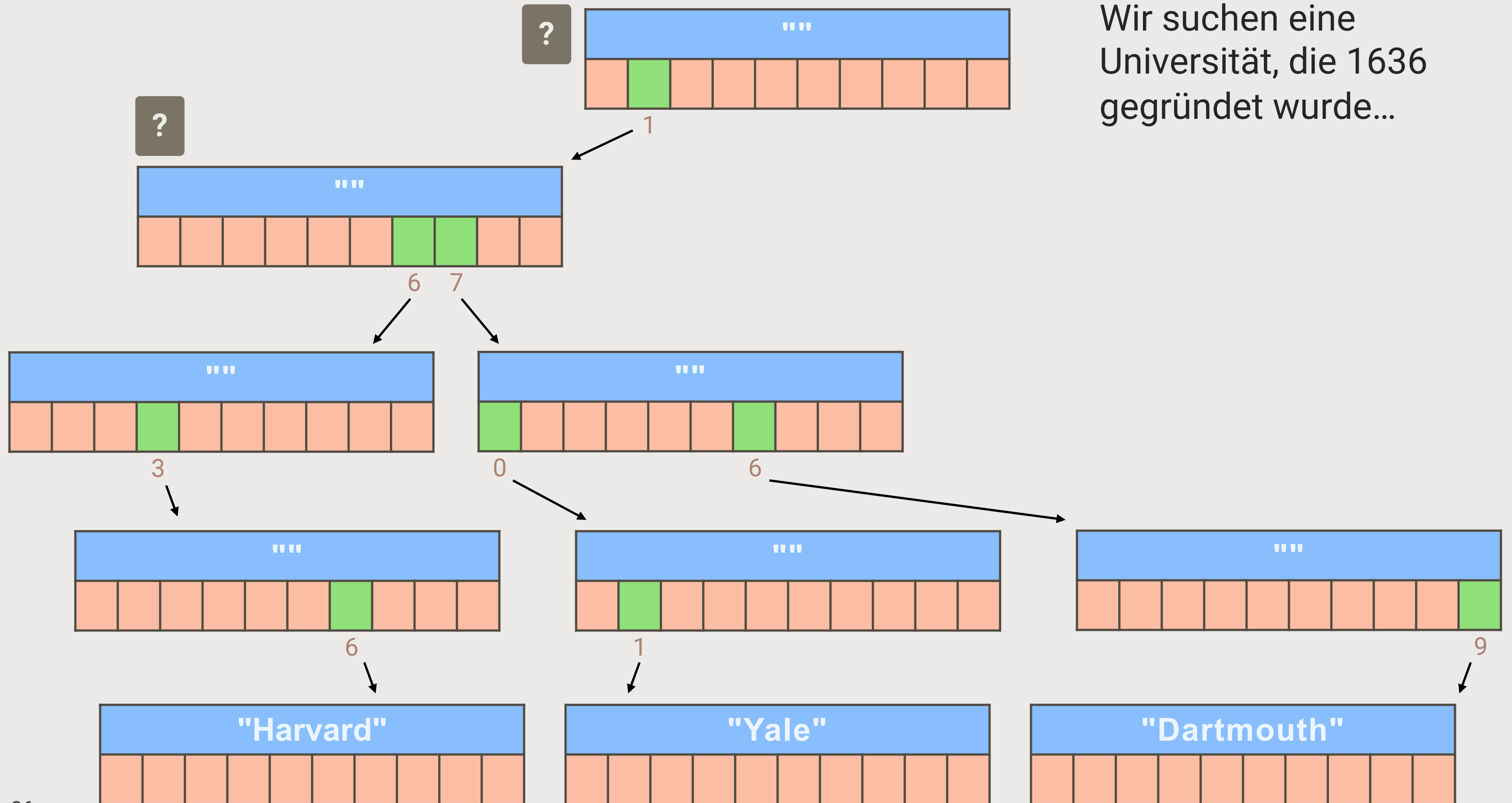
Wir suchen eine Universität, die 1636 gegründet wurde…

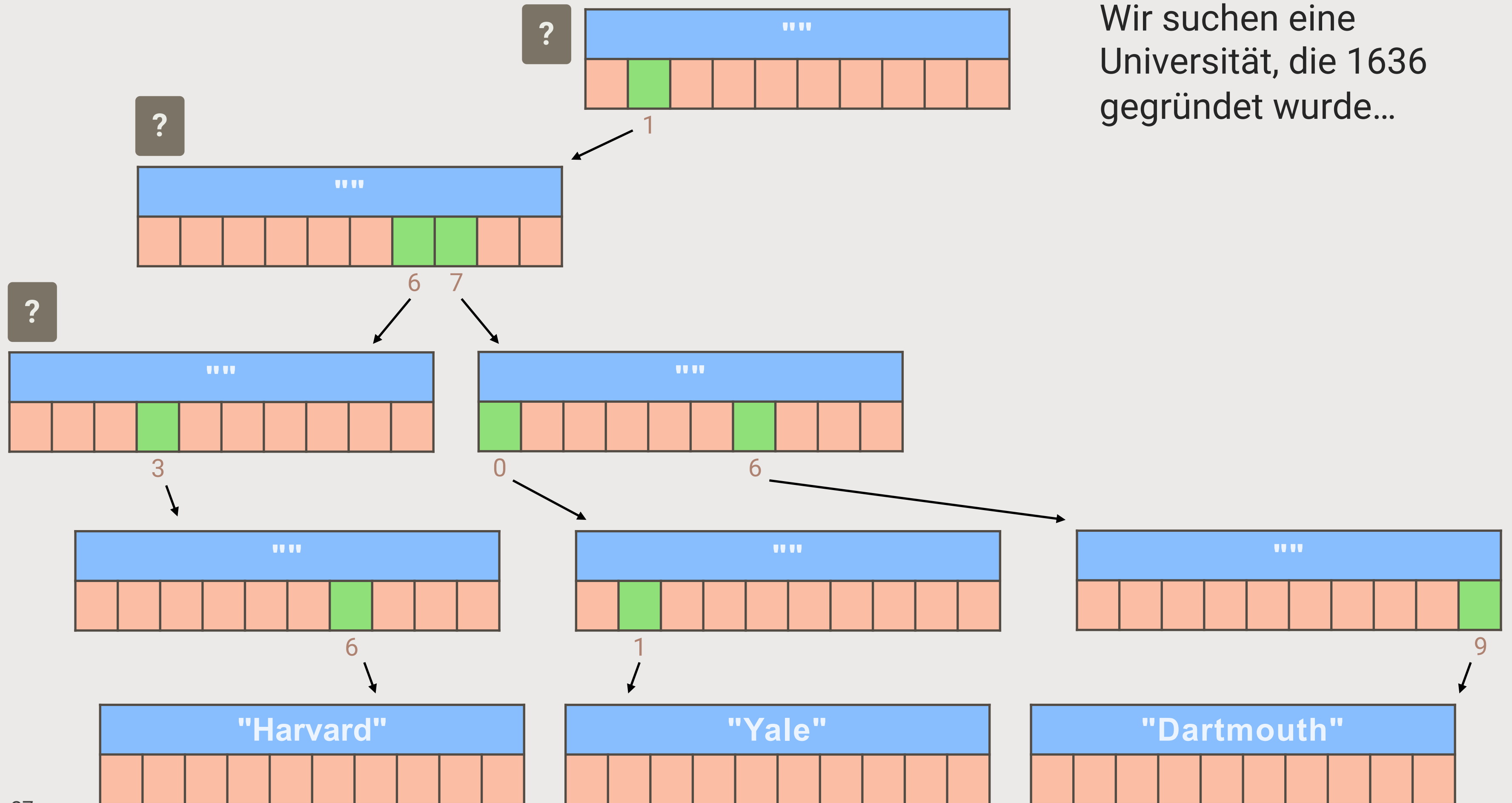




Wir suchen eine Universität, die 1636 gegründet wurde…

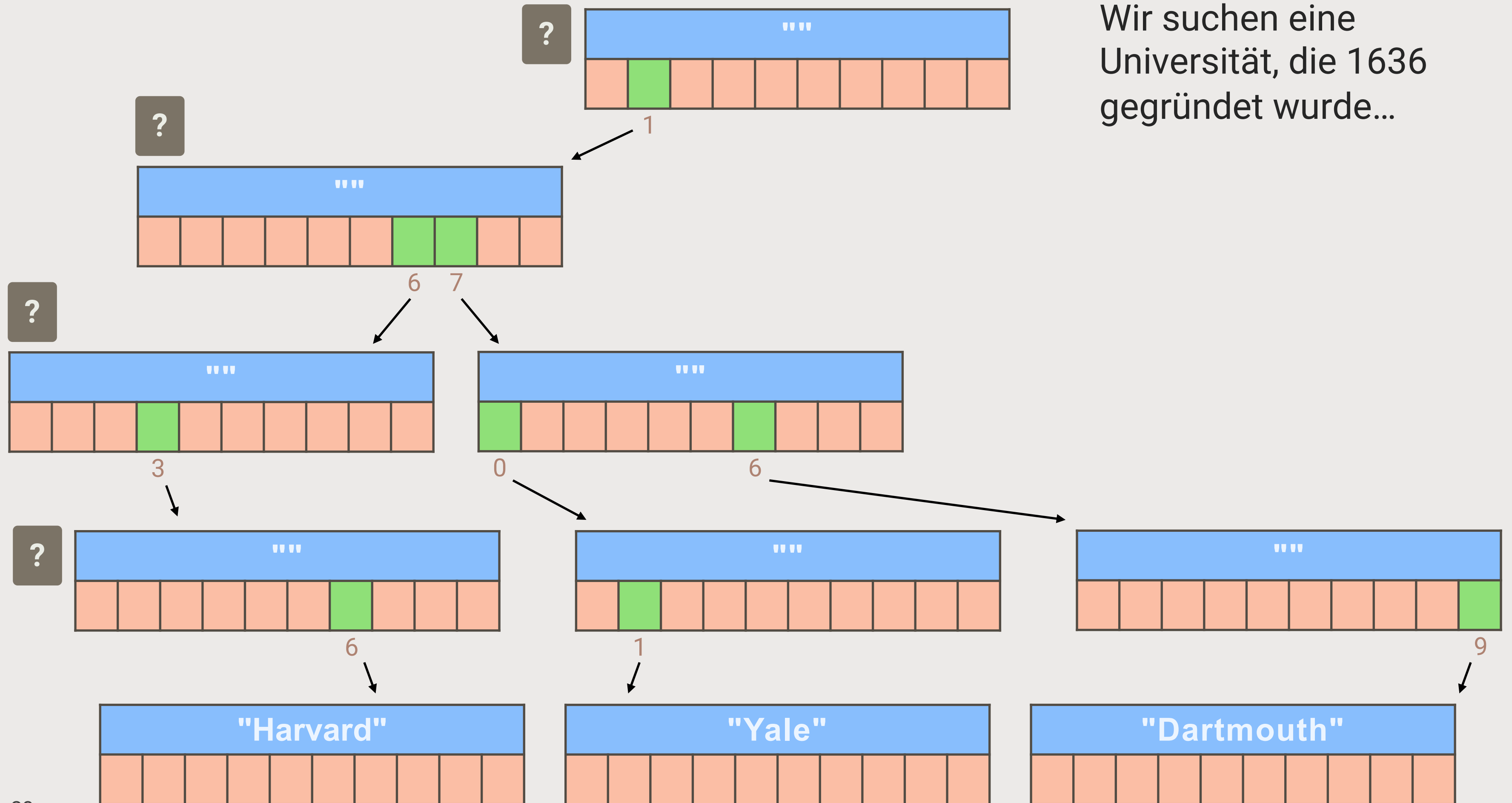
Wir suchen eine Universität, die 1636 gegründet wurde…



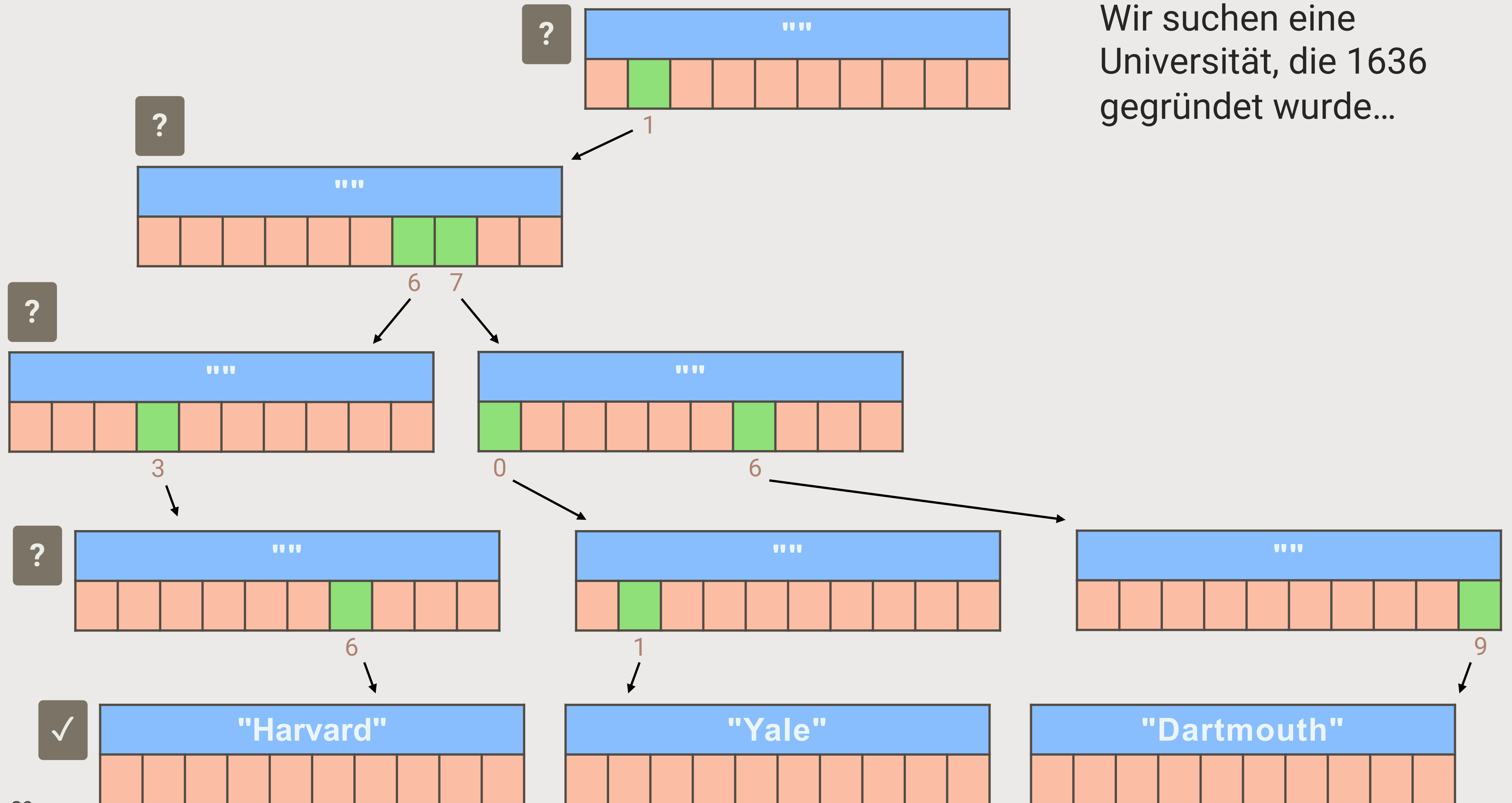


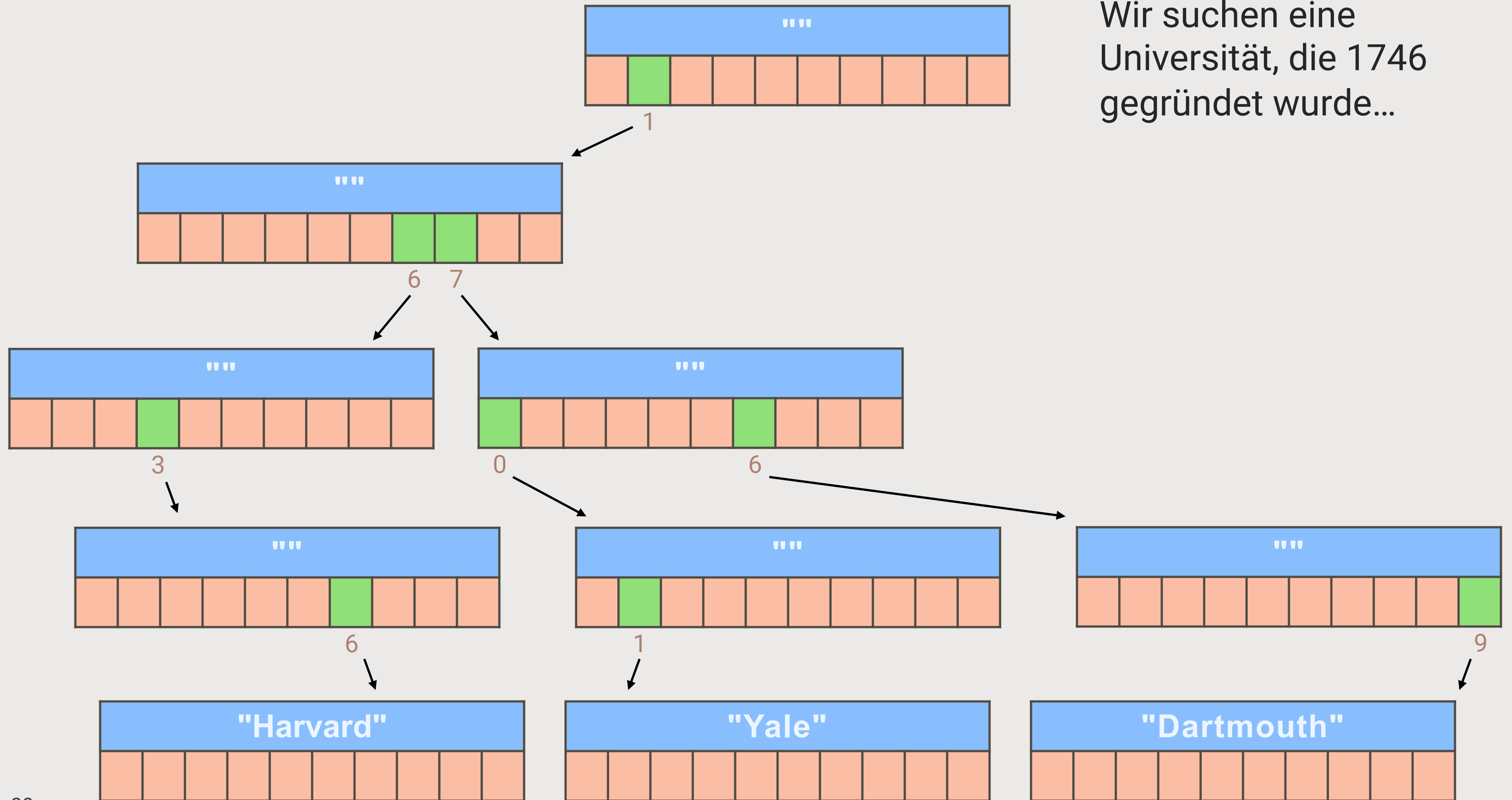
Wir suchen eine Universität, die 1636 gegründet wurde…

Wir suchen eine Universität, die 1636 gegründet wurde…

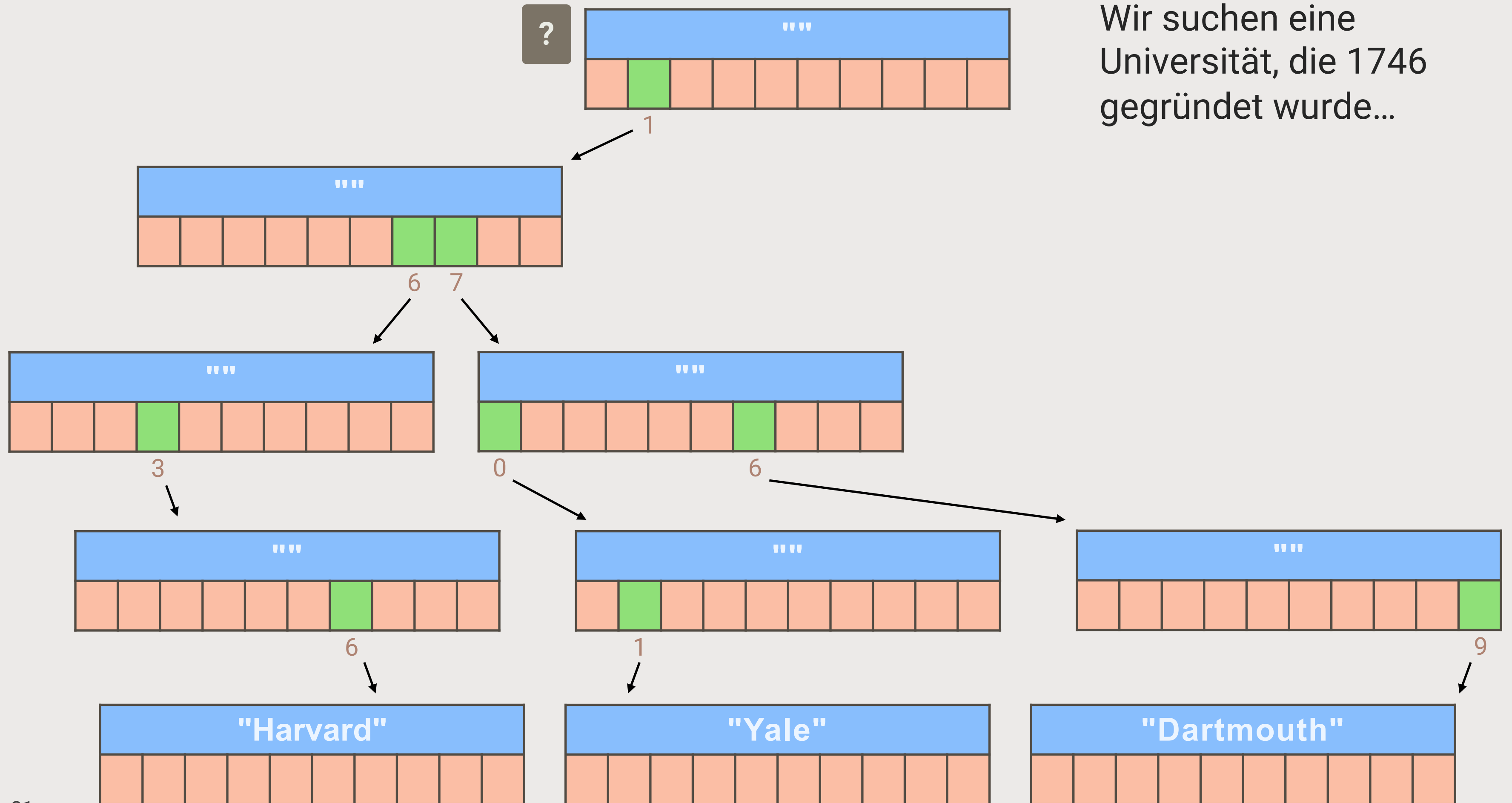


Wir suchen eine Universität, die 1636 gegründet wurde…



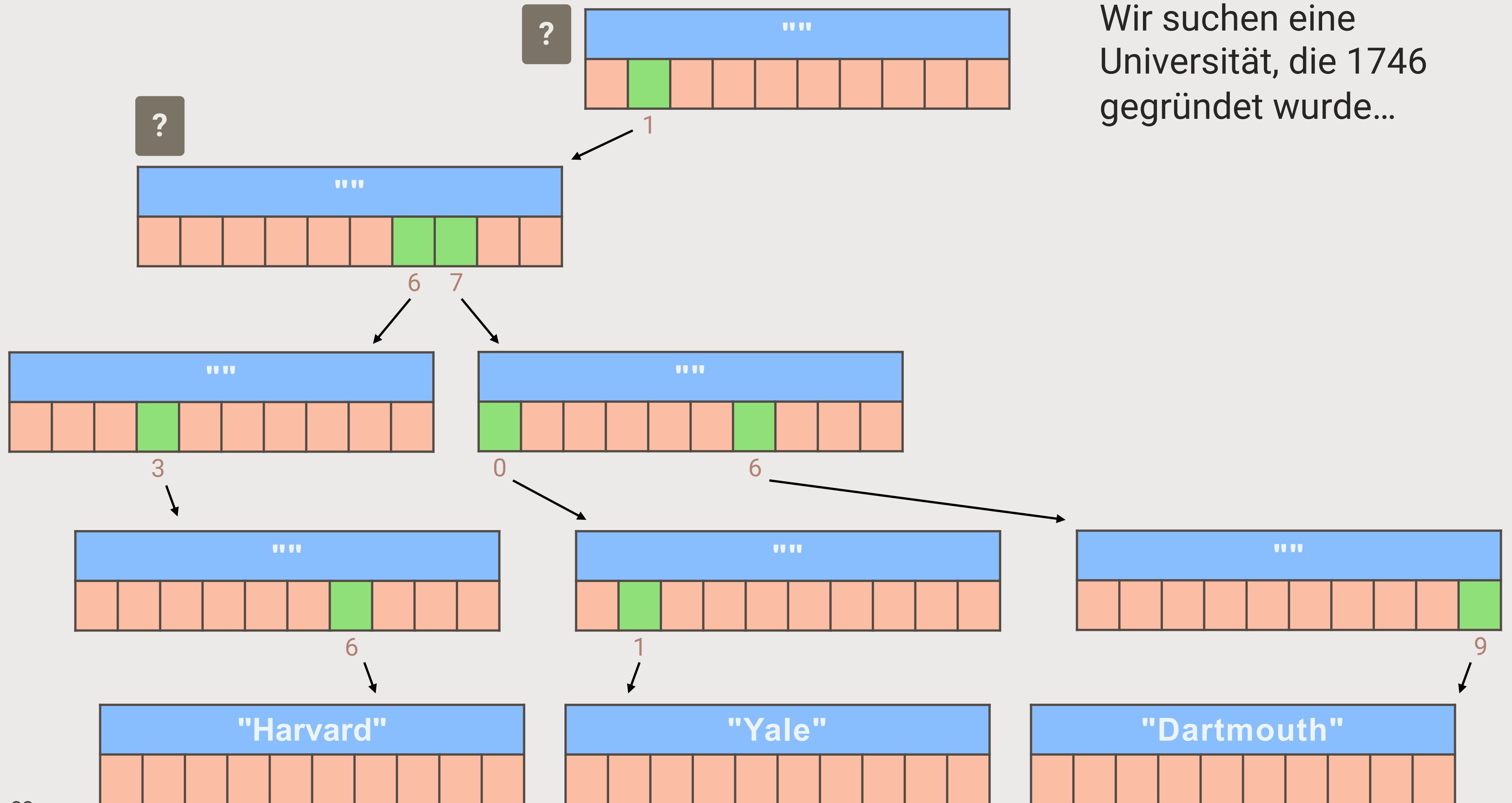


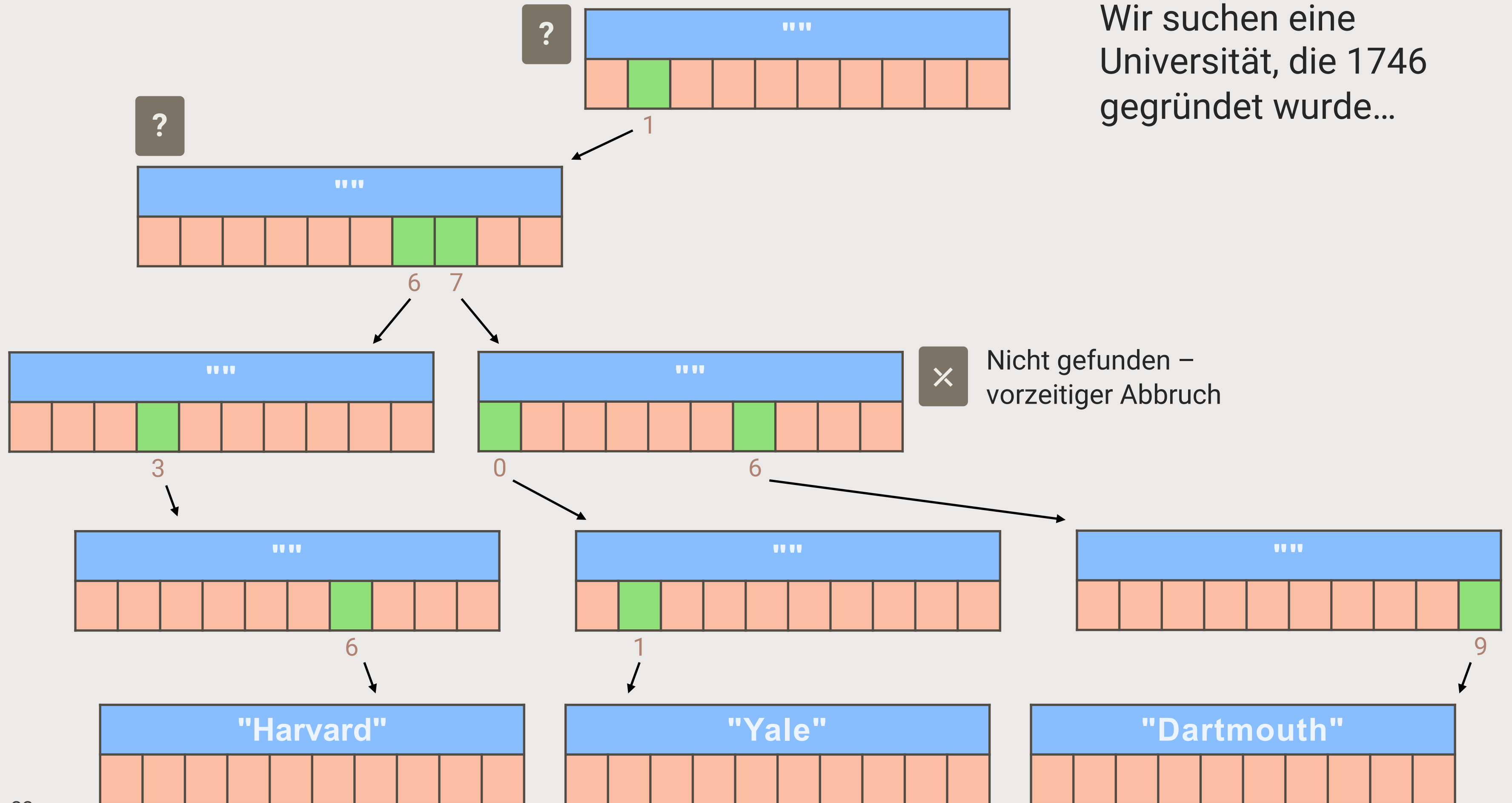
Wir suchen eine Universität, die 1746 gegründet wurde…



Wir suchen eine Universität, die 1746 gegründet wurde…

Wir suchen eine Universität, die 1746 gegründet wurde…





Wir suchen eine Universität, die 1746 gegründet wurde…

EXTRAS IN 3 MINUTEN

FRAGEN – ANTWORTEN – RÄTSEL
UND KURZE ZUSAMMENFASSUNG

Speicherplatz-optimierte Implementierung

```
struct trie_node {
    char character; // Zeichen, das dieser Knoten repräsentiert
    char* data;     // Pointer auf String (wenn Eintrag existiert; sonst NULL)
    struct trie_node* sibling;  // nächstes Zeichen auf gleicher Ebene
    struct trie_node* child;    // erstes Zeichen der nächsten Ebene
};
```

Zwei Optimierungen:

1. Wir ersetzen die Arrays durch Verkettete Listen.
   Vorteil: spart viel Speicher, da die meisten Array-Elemente NULL waren. *Nachteil:* langsamer, da kein direkter Zugriff mehr durch Indizieren möglich ist; nun müssen wir die Liste traversieren.

2. Die Nutzdaten speichern wir nicht mehr direct in der *struct*, sondern verweisen mit einem Pointer darauf. Vorteil: spart viel Speicher, weil wir nun nur in den Knoten Speicher reservieren können anstatt in allen Knoten auf dem Pfad.

Beispiel für die Speicherplatz-optimierte Implementierung

MUELLER, Max (Tel. 0951-863-2661)

M

U

E

L

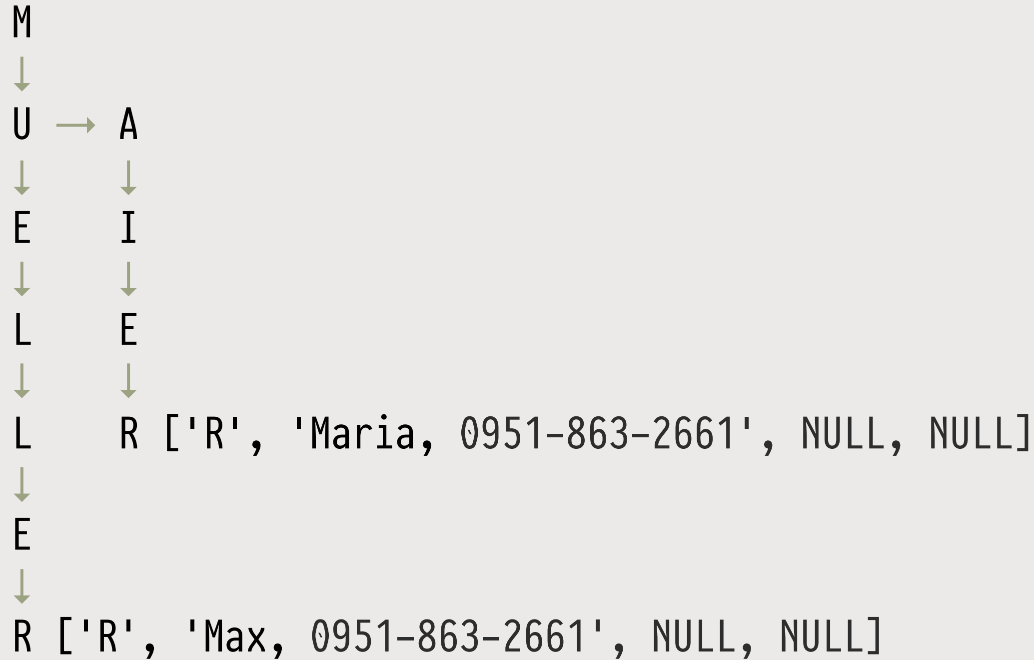
L

E

R ['R', 'Max, 0951-863-2661', NULL, NULL]

Beispiel für die Speicherplatz-optimierte Implementierung

MUELLER, Max (Tel. 0951-863-2661) und MAIER, Maria (Tel. 0951-863-2661)



Beispiel für die Speicherplatz-optimierte Implementierung

MUELLER, Max (Tel. 0951-863-2661) und MAIER, Maria (Tel. 0951-863-2661)

```
M ['M', NULL, NULL, <Pointer auf U>]
↓
U → A
↓   ↓
E   I
↓   ↓
L   E
↓   ↓
L   R ['R', 'Maria, 0951-863-2661', NULL, NULL]
↓
E
↓
R ['R', ‘Max, 0951-863-2661', NULL, NULL]
```

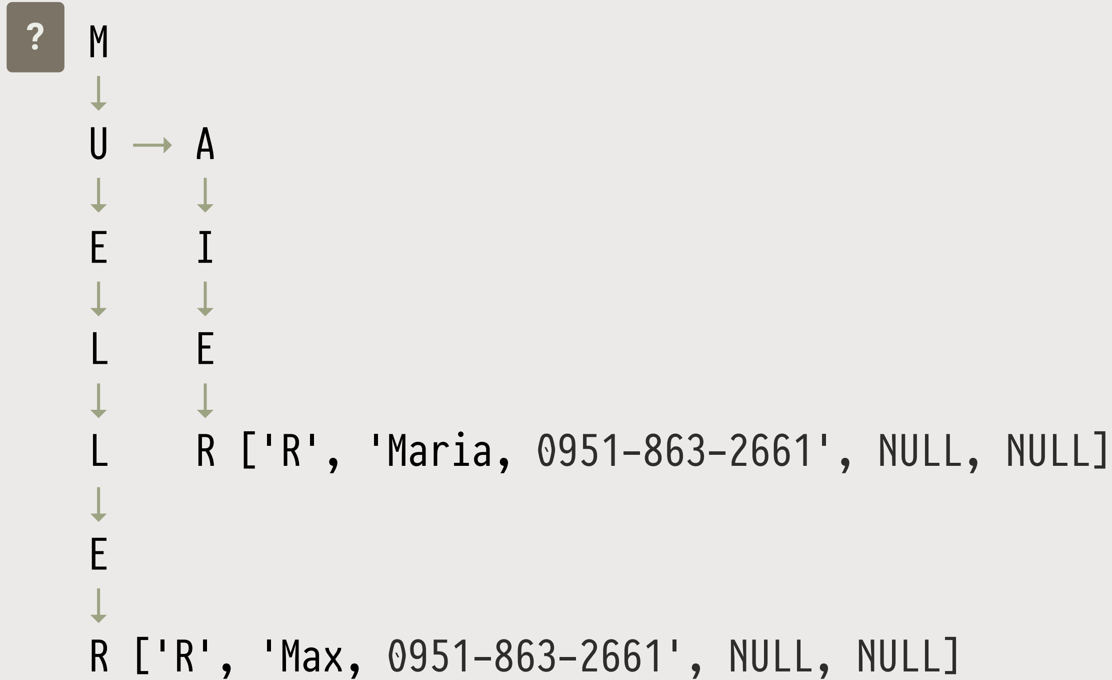
Beispiel für die Speicherplatz-optimierte Implementierung

MUELLER, Max (Tel. 0951-863-2661) und MAIER, Maria (Tel. 0951-863-2661)

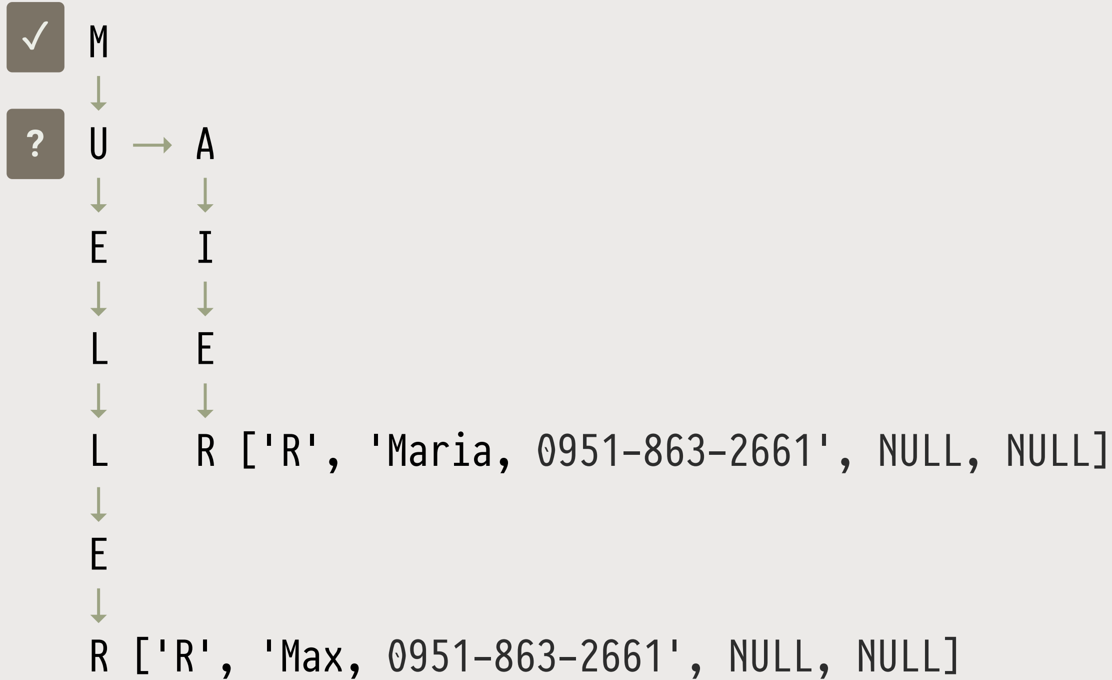
```
M ['M', NULL, NULL, <Pointer auf U>]
↓
U ['U', NULL, <Pointer auf A>, <Pointer auf E>]   → A
↓                                                  ↓
E                                                  I
↓                                                  ↓
L                                                  E
↓                                                  ↓
L                                                  R [...]
↓
E
↓
R ['R', 'Max, 0951-863-2661', NULL, NULL]
```

```c
bool find(struct trie_node* root, const char* name)
{
    struct trie_node* current = root;
    for(int i = 0; name[i] != '\0'; i++) // iteriere über Buchstaben in name
    {
        // Suche in der Liste der Geschwister
        while(current != NULL && current->character != name[i])
        {
            current = current->sibling;
        }

        // Buchstabe nicht gefunden?
        if(current == NULL)
        {
            return false;
        }

        // Gehe zur nächsten Ebene
        current = current->child;
    }
    return (current != NULL && current->data != NULL);
}
```

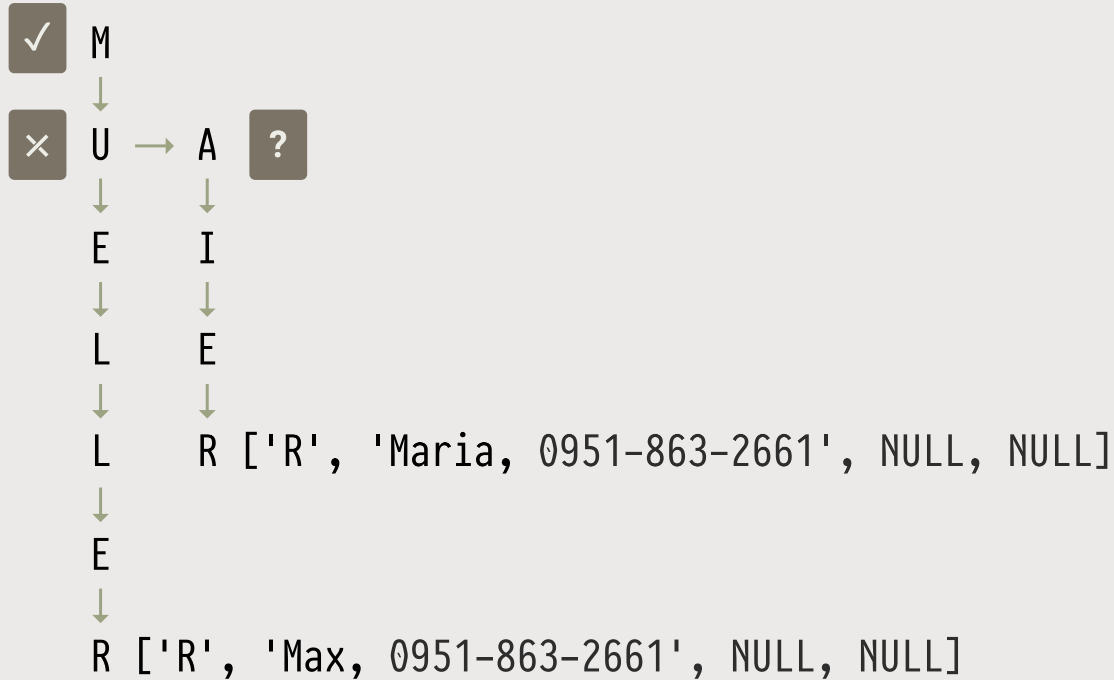
Suche nach MAIER im Trie



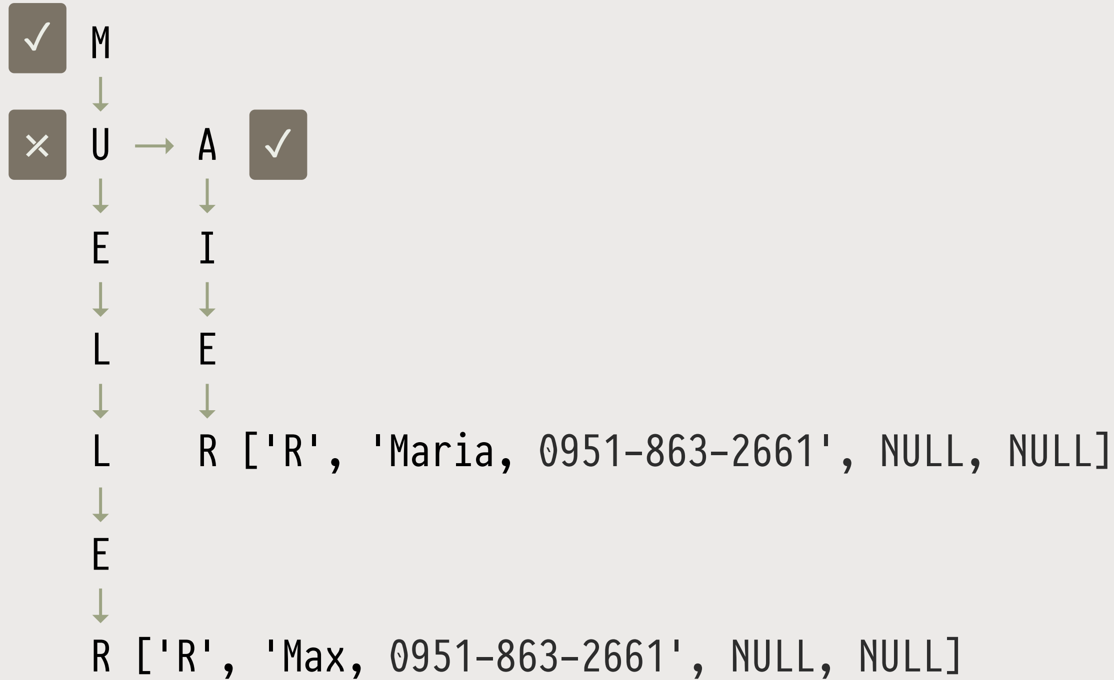
Suche nach MAIER im Trie



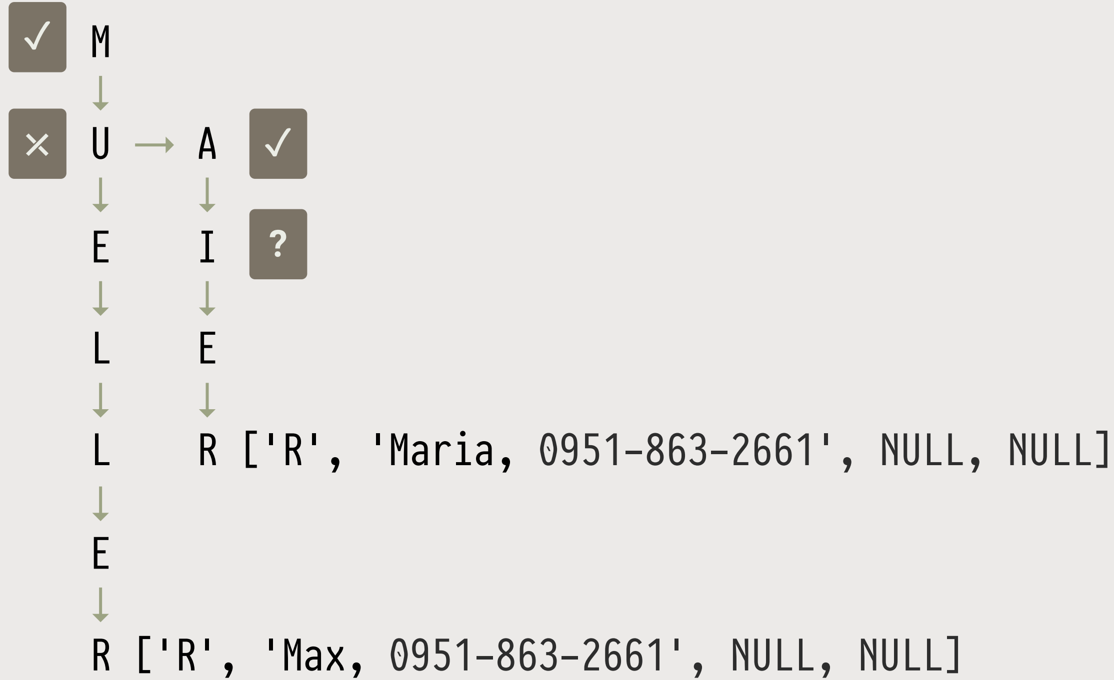
Suche nach MAIER im Trie



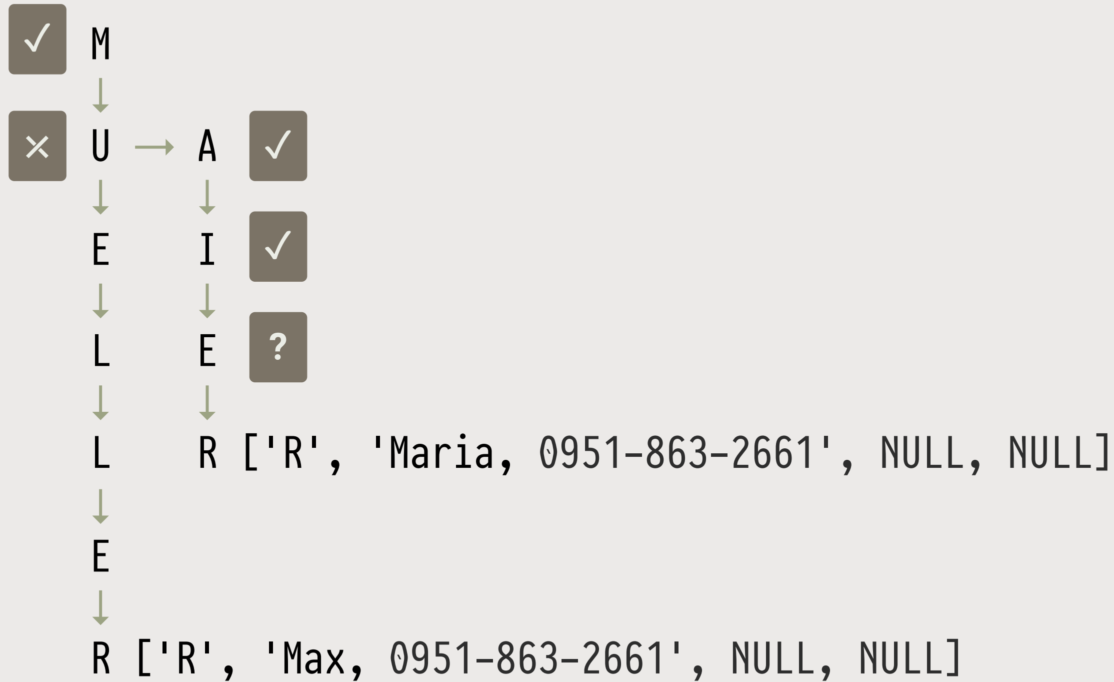
Suche nach MAIER im Trie



Suche nach MAIER im Trie



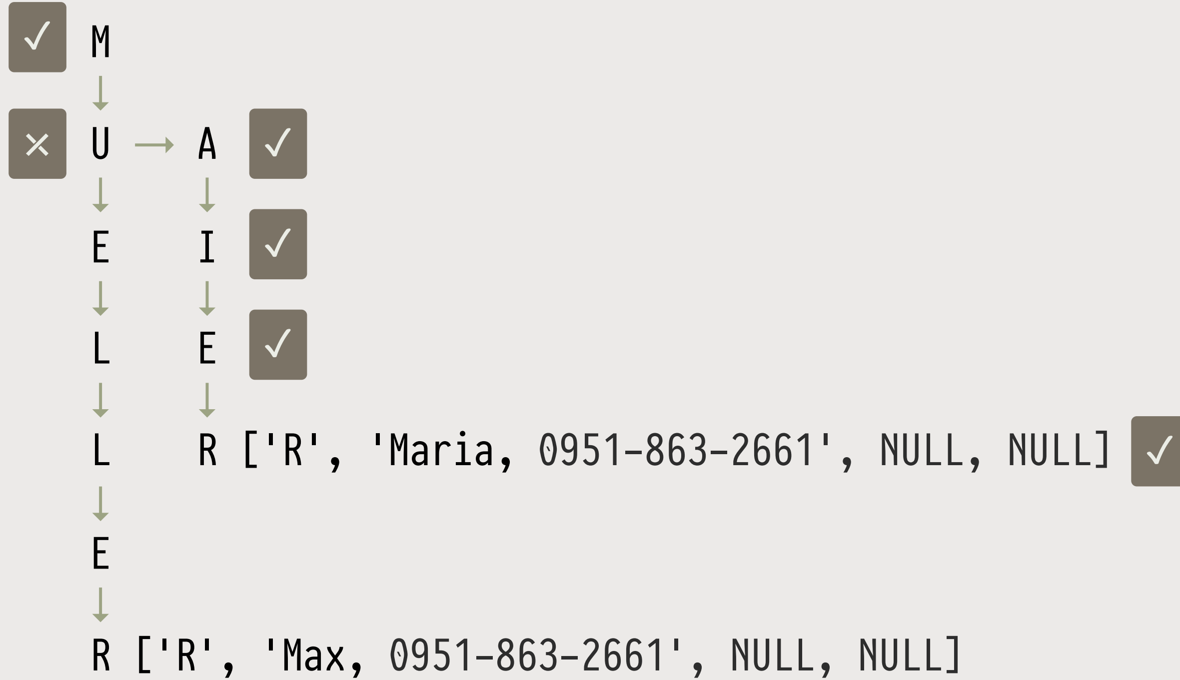
Suche nach MAIER im Trie



Suche nach MAIER im Trie

```
✓ M
  ↓
✕ U → A ✓
  ↓   ↓
  E   I ✓
  ↓   ↓
  L   E ✓
  ↓   ↓
  L   R ['R', 'Maria, 0951-863-2661', NULL, NULL] ?
  ↓
  E
  ↓
  R ['R', 'Max, 0951-863-2661', NULL, NULL]
```

Suche nach MAIER im Trie



Kombination aus Bäumen und Arrays (oder Listen)

Laufzeitkomplexität der Suche ist $O(1)$.

Diese hängt nur von der Länge des Keys, nicht von der Anzahl der gespeicherten Elemente.

Key = Wegbeschreibung durch den Baum

Laufzeitvorteile werden durch hohen Platzbedarf erkauft (sog. Time-Memory-Trade-off).

Vorteil im Vergleich zu Hashtabellen: garantiert eindeutige Pfade, keine Kollisionen.

Trade-off zwischen Speicher-bedarf und Laufzeit ist veränderbar, etwa durch Ersetzen der Arraxs durch Verkettete Listen.