# Tries

# Implementing **`Lexicon`**

- The **`Lexicon`** represents a set of English words.
- Main operations:
  - Add word.
  - Remove word.
  - Is word contained?
  - Is prefix contained?
- How can we efficiently implement the **`Lexicon`**?

# Sorted Array Implementation

- We could implement the `Lexicon` as an array of all the words it contains.
- To add a word: **O(*n*)**
  - Check if the word already exists.
  - If not, insert it in sorted order.
- To remove a word: **O(*n*)**
  - Find and remove it from the array.
- To see if a word exists: **O(log *n*)**
  - Perform binary search of the array for the word.
- To see if a prefix exists: **O(log *n*)**
  - Perform binary search of the array to see if the word is a prefix of some word in the array.

# A Better Implementation

- Use a Binary Search Tree.
- Adding, removing, checking for a word AND **`containsPrefix`** now run very quickly (**O(log *n*)** comparisons needed).
- Can we do any better?

# A (kinda) Better Implementation

- Use a hash table.
- Adding, removing, and checking for a word now run even faster (**O(1)** comparisons needed).
- How would you implement **`containsPrefix`**?
  - Would have to check all words in all buckets.
    - Linear search!

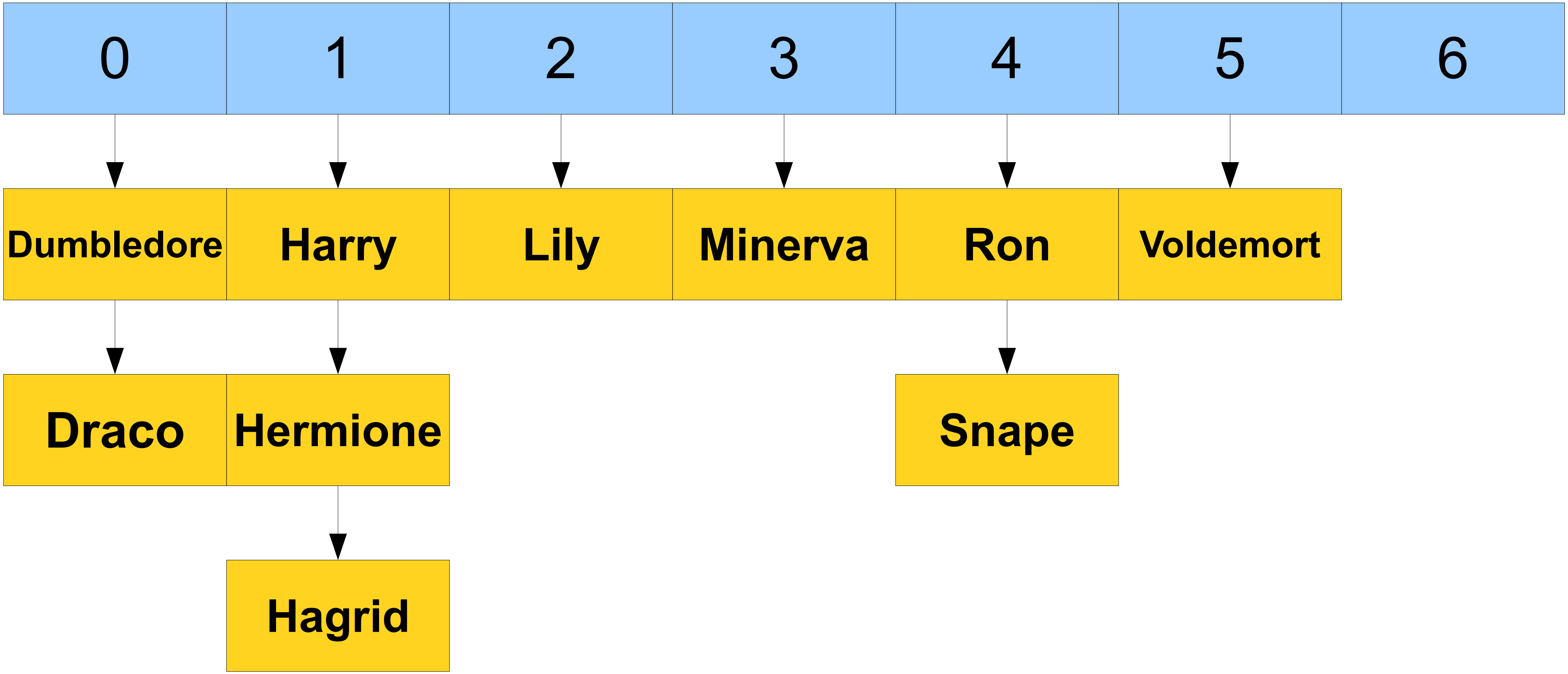
# What We Want

- What we want is a data structure that allows us to lookup, insert, remove AND check for a prefix in **O(1)**
- It isn't possible to do this with any of the structures we've covered so far.
- We need a new data structure!

# Rethinking Hashing

- Our motivation behind hashing was to put values into places where we would know to look for them.
- When storing strings as our keys, one initial idea was to break strings apart by their first letter.
- Let's look at that idea again.

# An Initial Hashing Idea



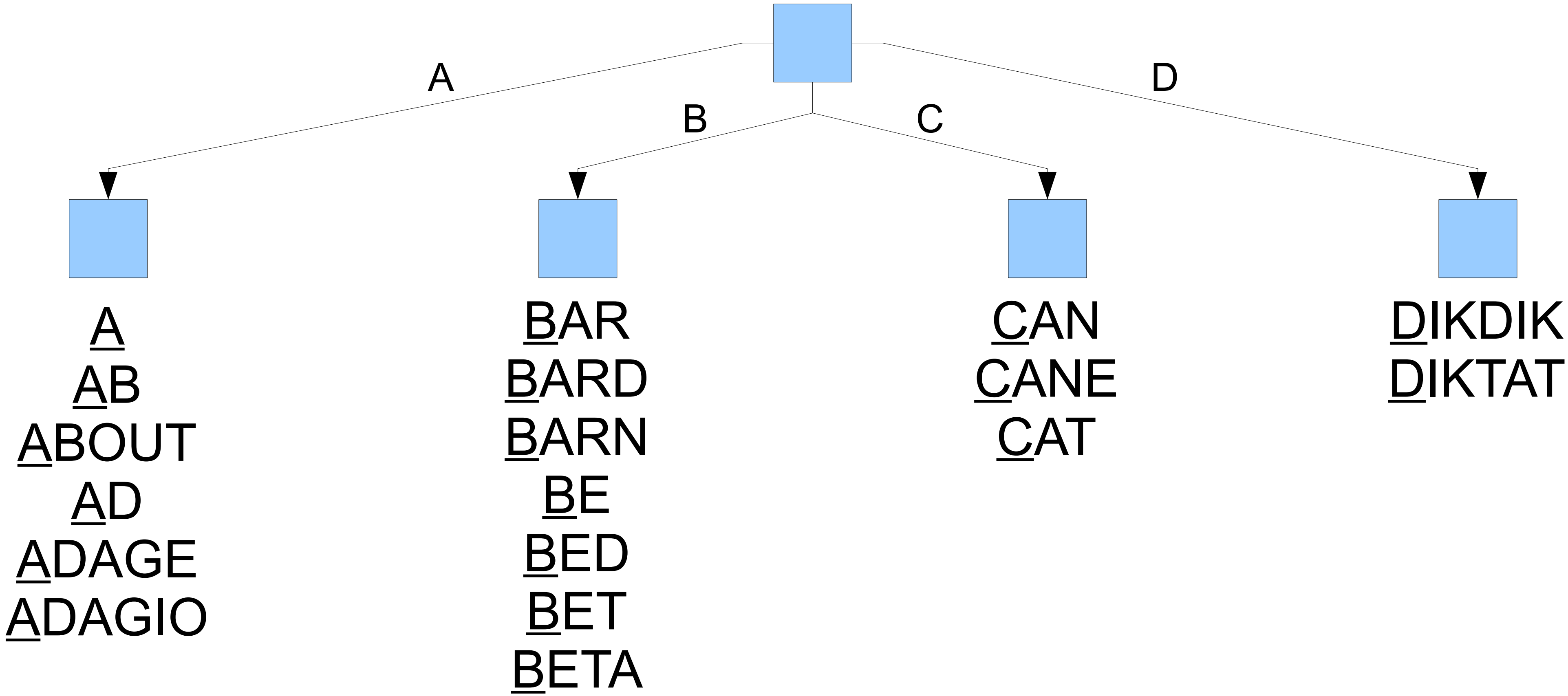
A
AB
ABOUT
AD
ADAGE
ADAGIO
BAR
BARD
BARN
BE
BED
BET
BETA
CAN
CANE
CAT
DIKDIK
DIKTAT

A
AB
ABOUT
AD
ADAGE
ADAGIO

BAR
BARD
BARN
BE
BED
BET
BETA

CAN
CANE
CAT

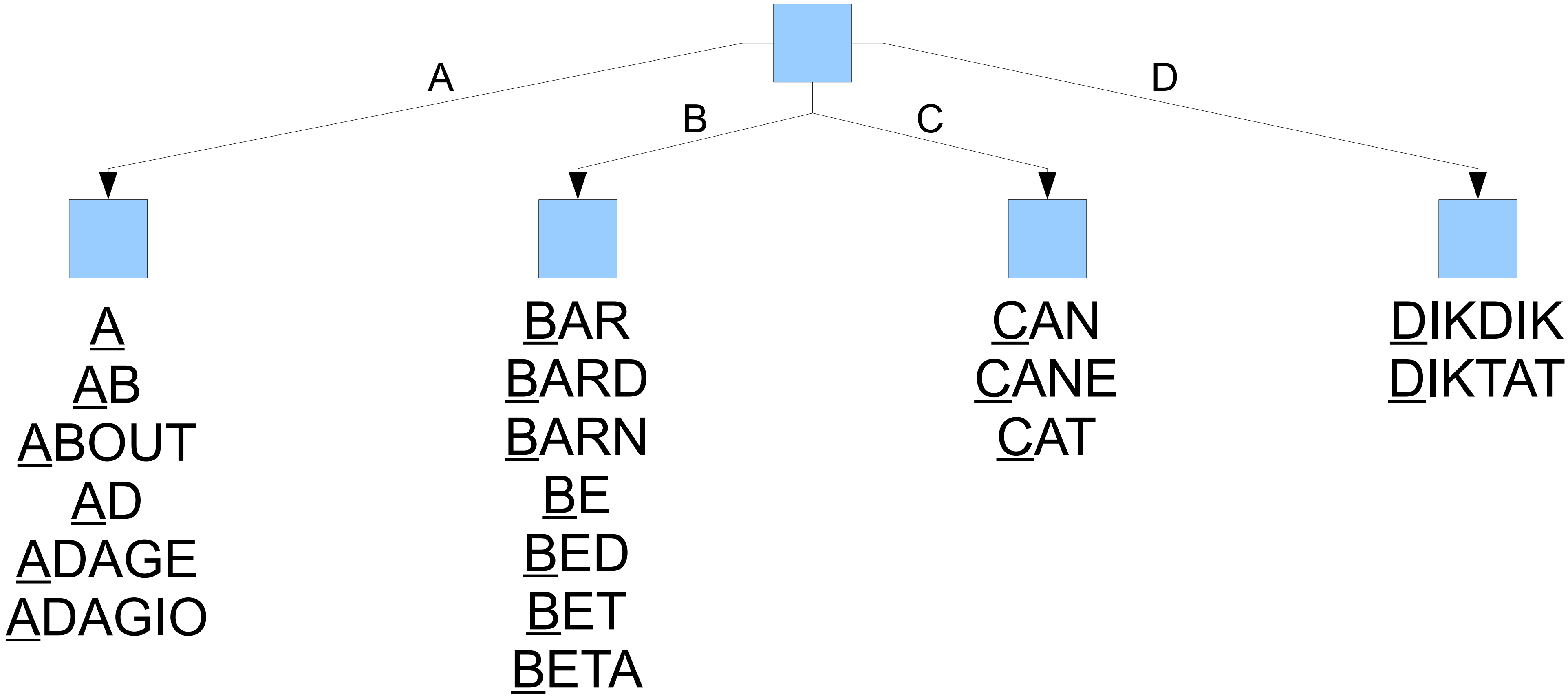
DIKDIK
DIKTAT

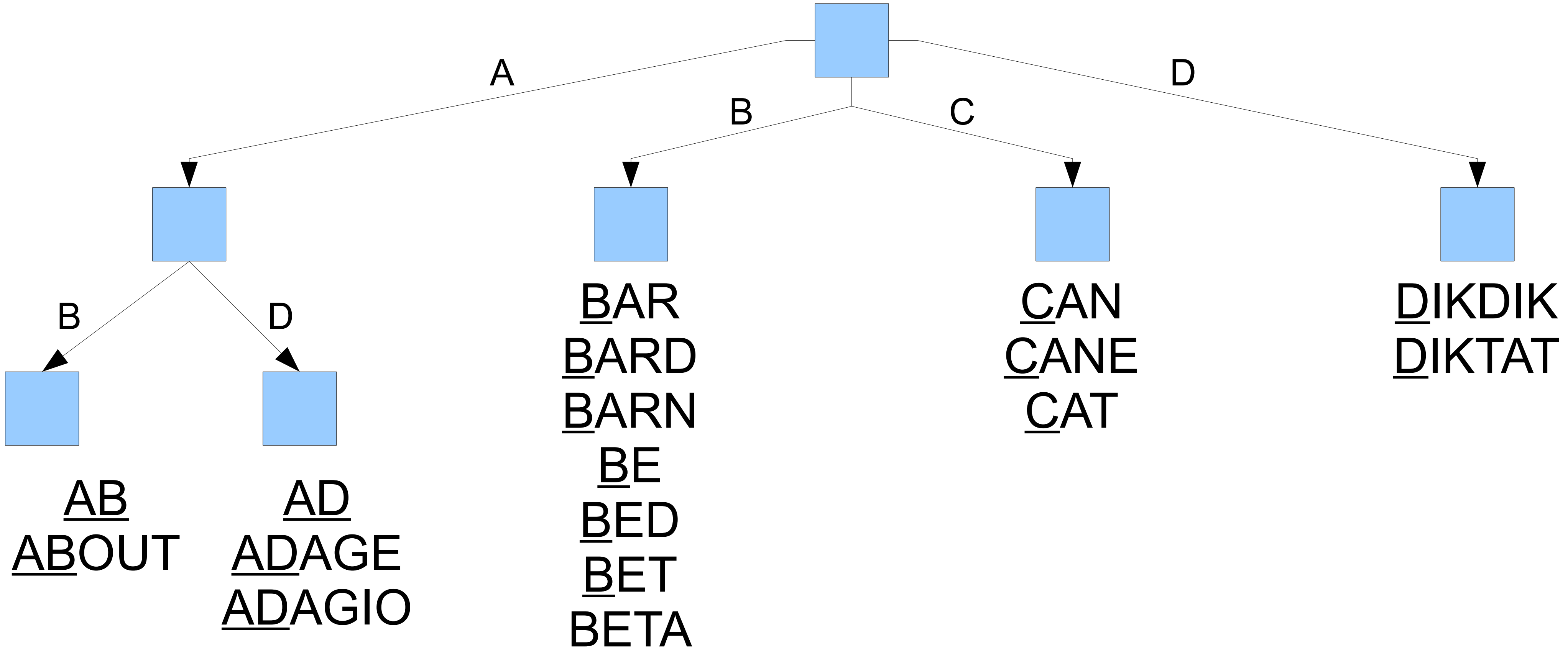

A
B
C
D
A
AB
ABOUT
AD
ADAGE
ADAGIO
BAR
BARD
BARN
BE
BED
BET
BETA
CAN
CANE
CAT
DIKDIK
DIKTAT

# How does this affect performance?

- If we assume that roughly the same number of words start with each letter, then we've sped up **`containsPrefix`** by a factor of 1/26
  - Totally not a fair assumption to make. But it still gives us a good constant factor speedup
- **`containsPrefix`** still runs in **O(*n*)** (unsorted) or **O(log(*n*))** (sorted)

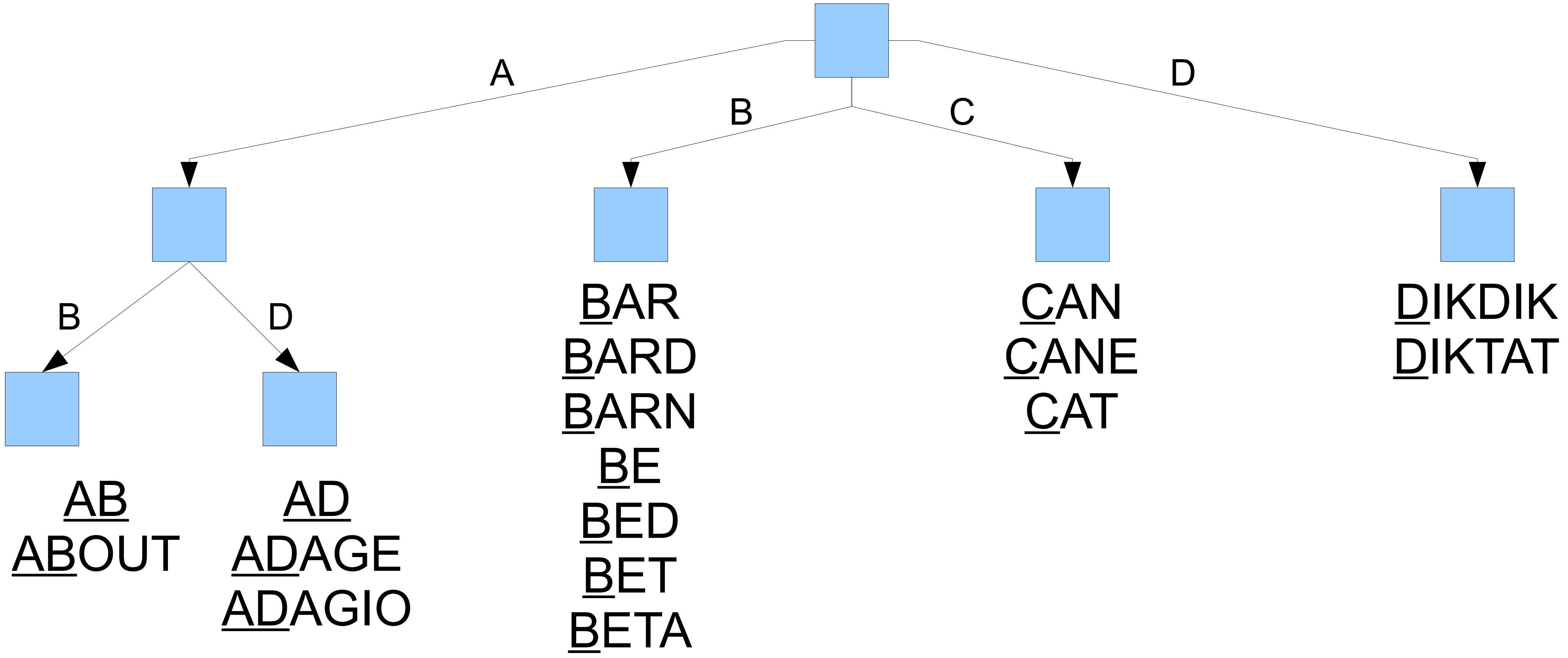
What happens if we *split again*

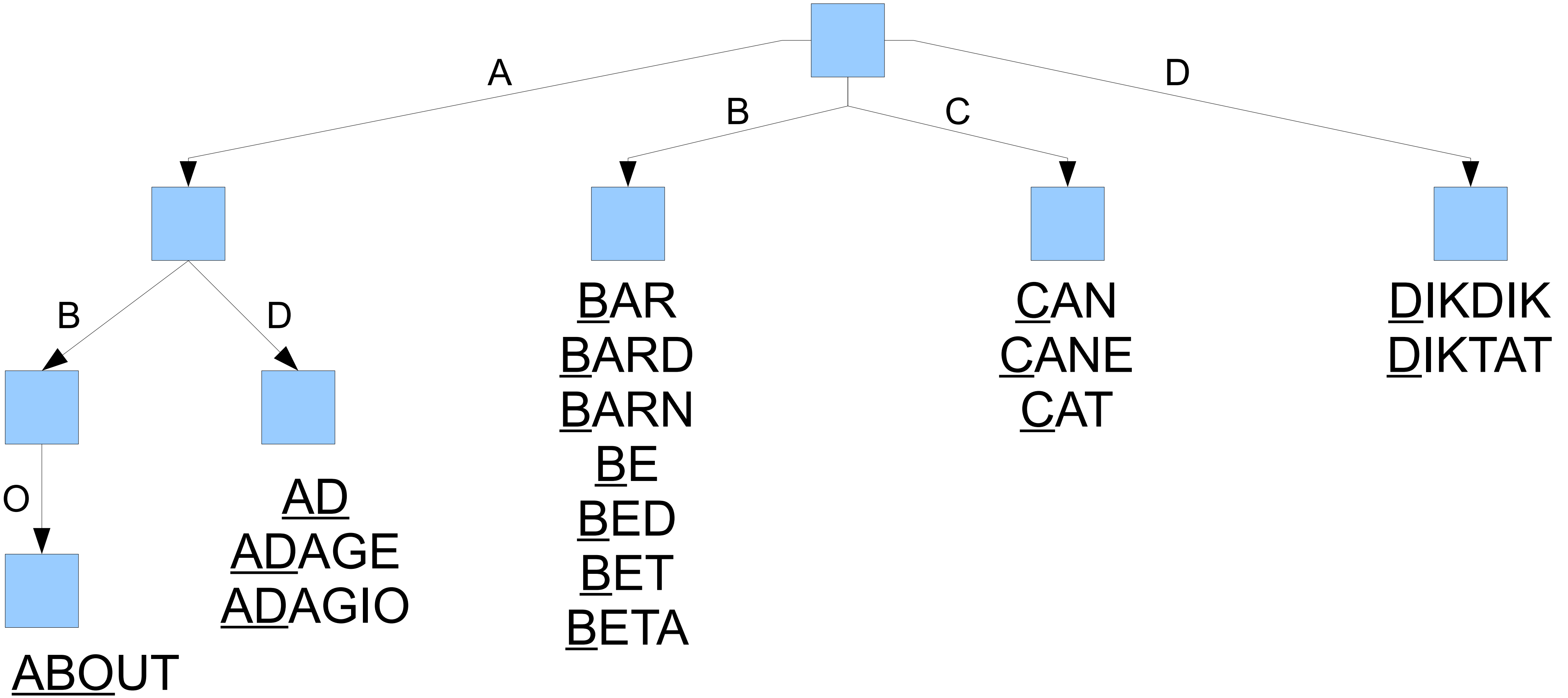

A
B
C
D
A
AB
ABOUT
AD
ADAGE
ADAGIO
BAR
BARD
BARN
BE
BED
BET
BETA
CAN
CANE
CAT
DIKDIK
DIKTAT

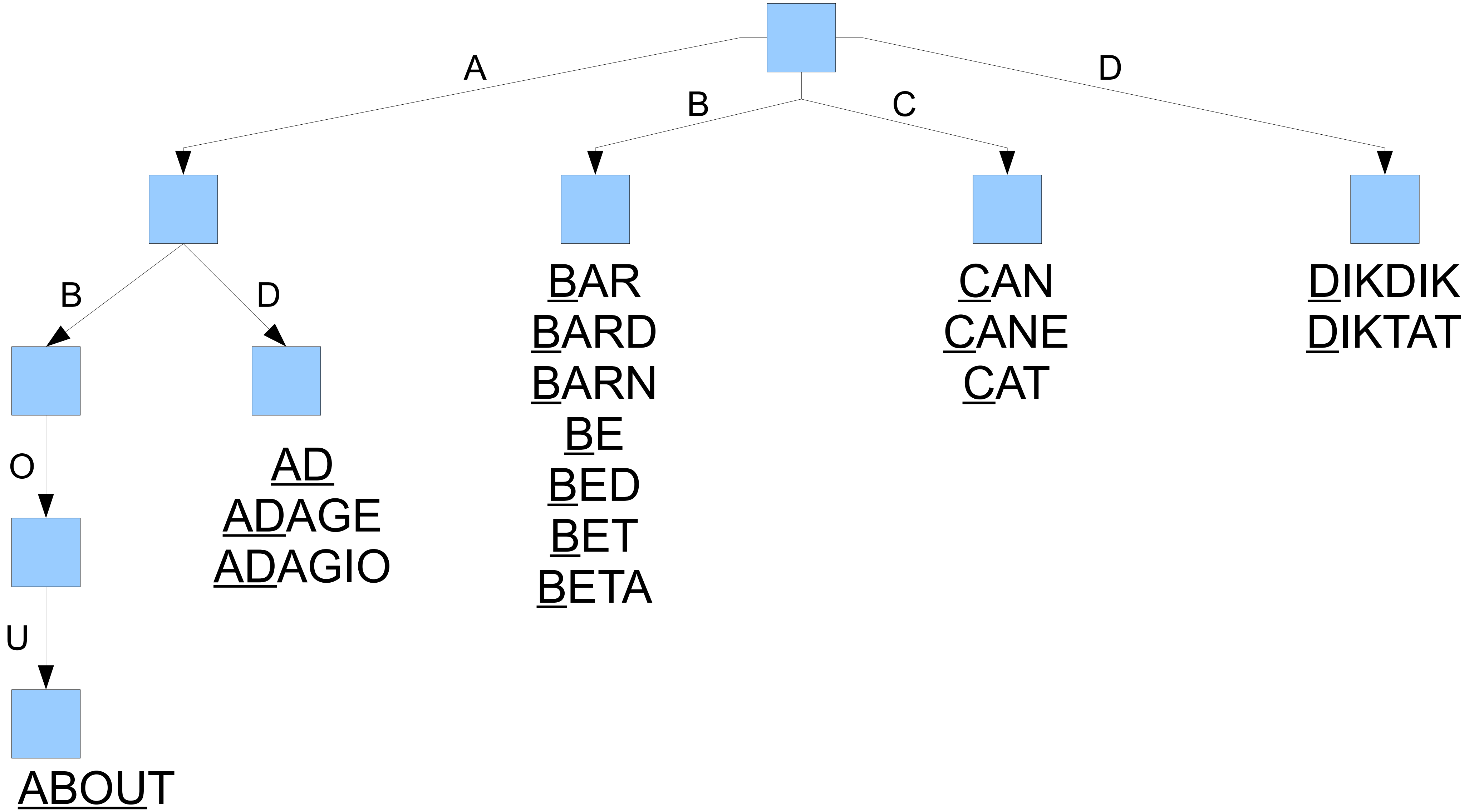

A
B
C
D
B
D
BAR
BARD
BARN
BE
BED
BET
BETA
CAN
CANE
CAT
DIKDIK
DIKTAT
AB
ABOUT
AD
ADAGE
ADAGIO

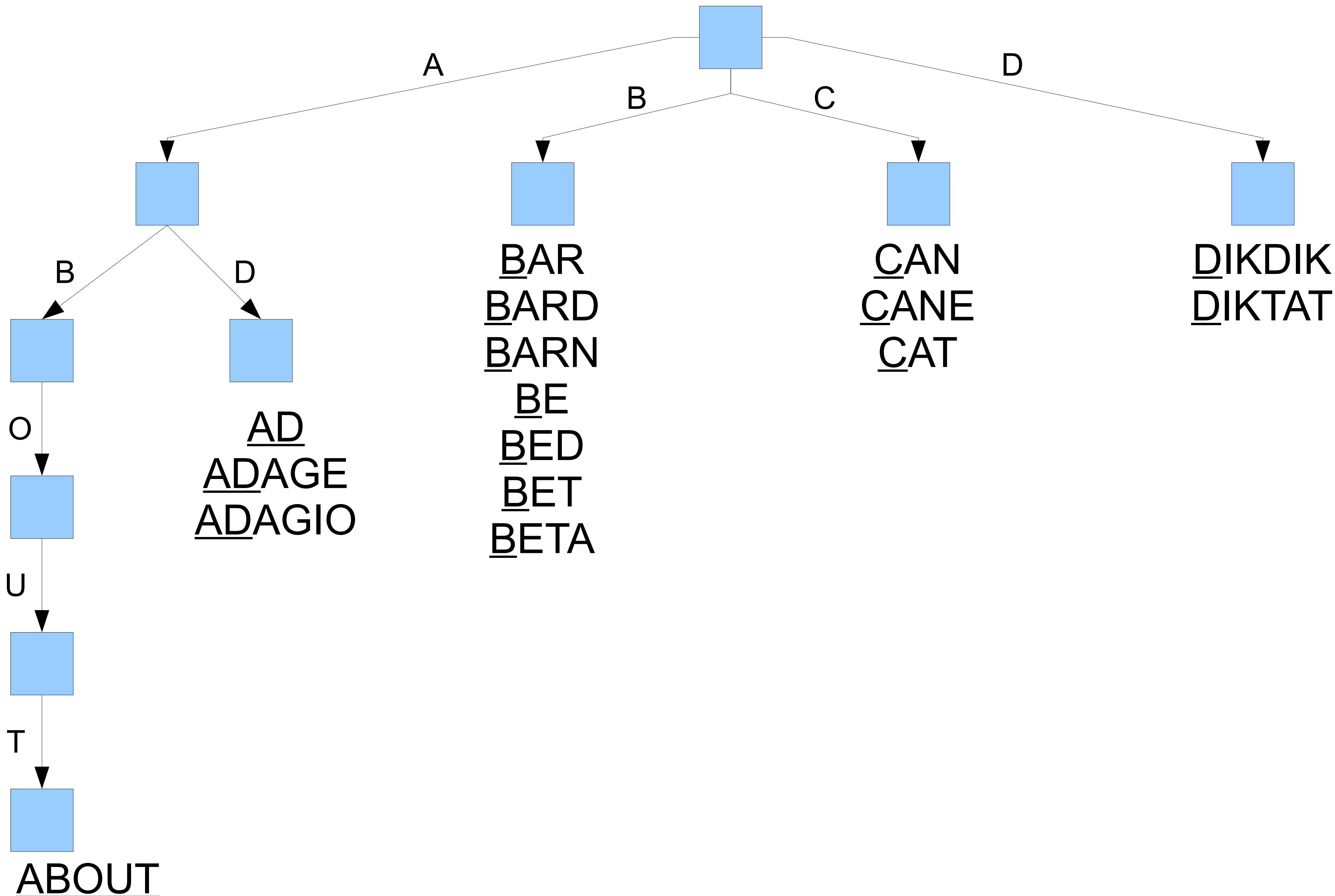
# How does this affect performance?

- This gives us another constant factor speedup on words that start with “AB” and “AD”
- What happens if we continue this process

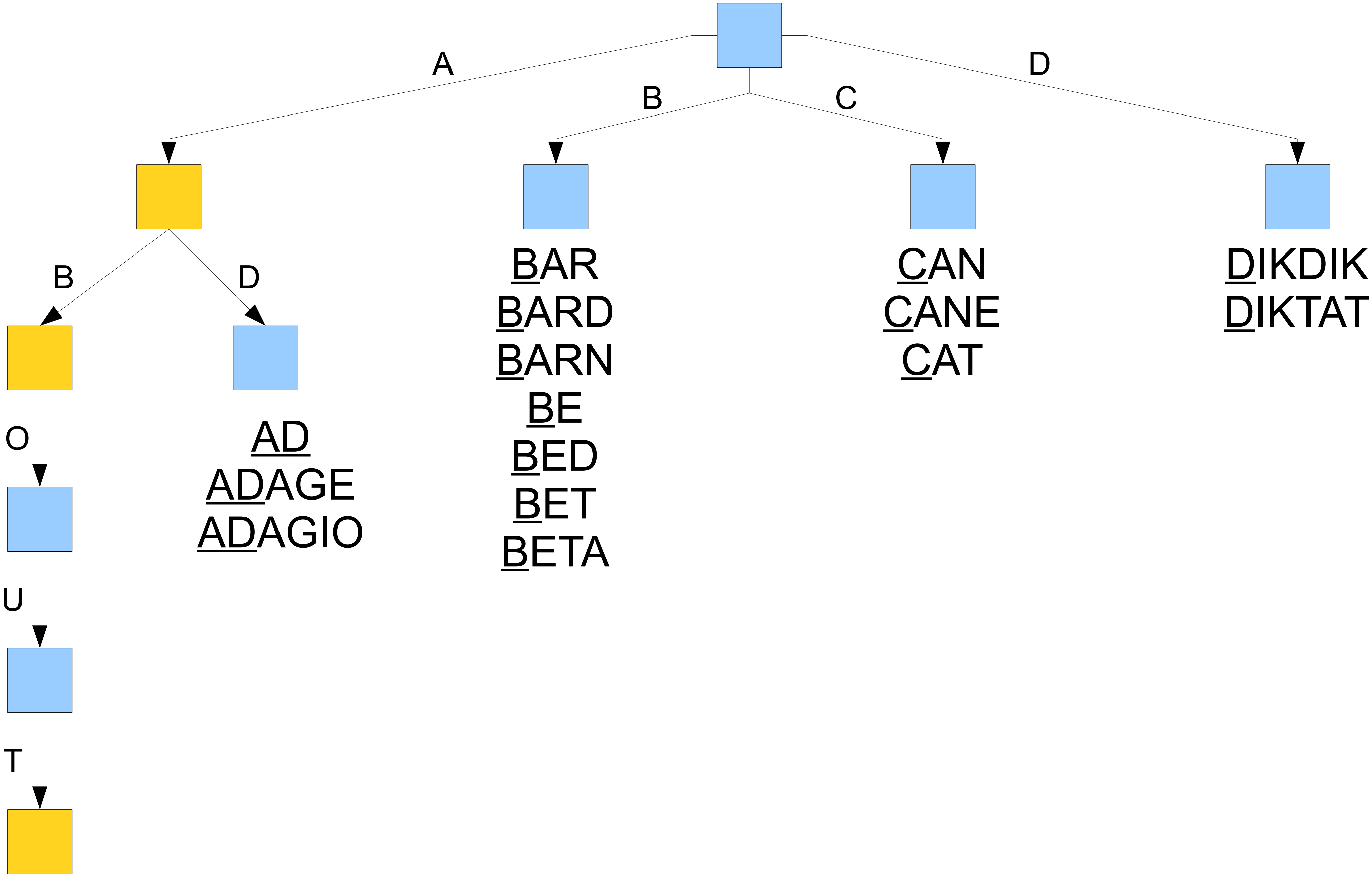

A
B
C
D
B
D
BAR
BARD
BARN
BE
BED
BET
BETA
CAN
CANE
CAT
DIKDIK
DIKTAT
AB
ABOUT
AD
ADAGE
ADAGIO

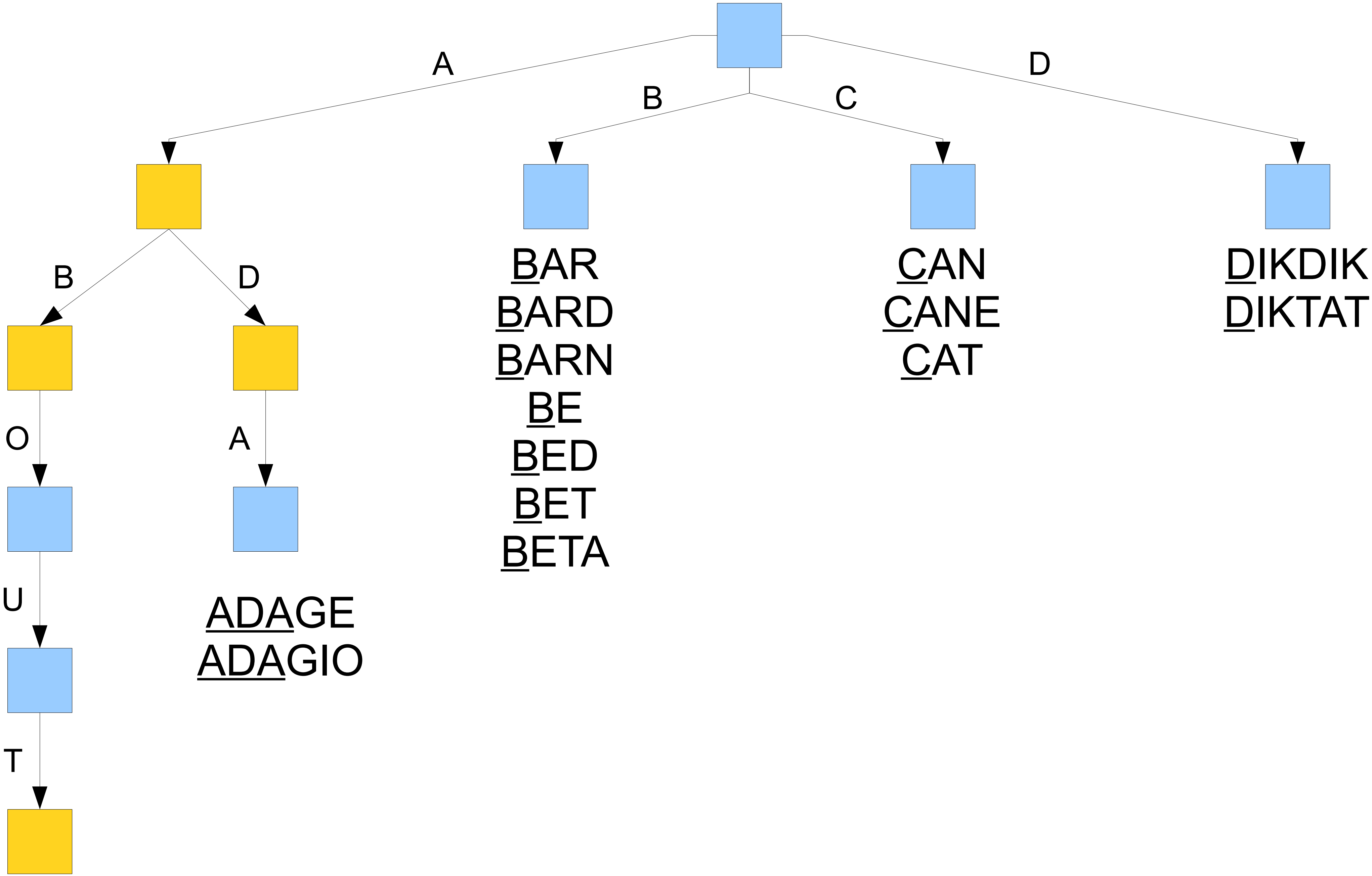

A
B
C
D
B
D
O
BAR
BARD
BARN
BE
BED
BET
BETA
CAN
CANE
CAT
DIKDIK
DIKTAT
AD
ADAGE
ADAGIO
ABOUT

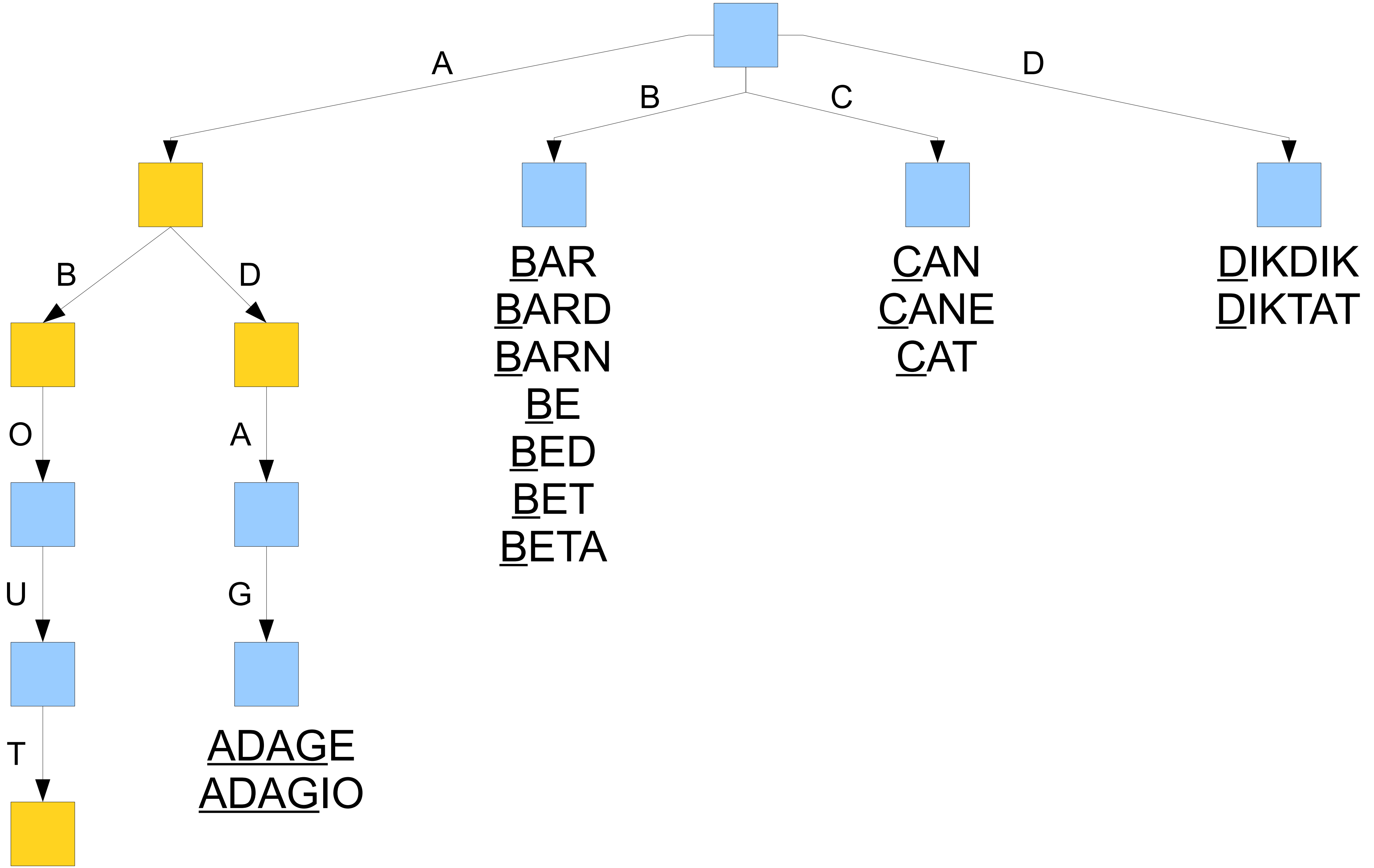

A
B
C
D
B
D
O
U
BAR
BARD
BARN
BE
BED
BET
BETA
CAN
CANE
CAT
DIKDIK
DIKTAT
AD
ADAGE
ADAGIO
ABOUT

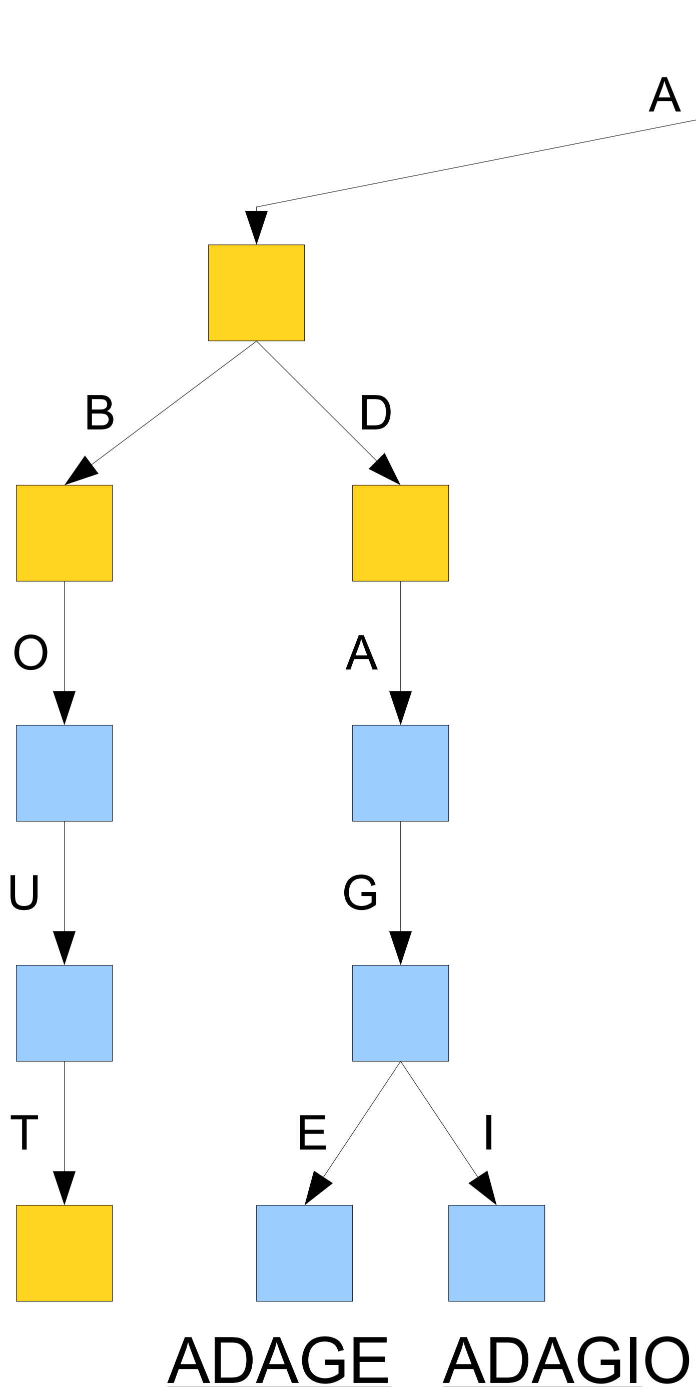


A
B
C
D
B
D
BAR
BARD
BARN
BE
BED
BET
BETA
CAN
CANE
CAT
DIKDIK
DIKTAT
O
AD
ADAGE
ADAGIO
U
T
ABOUT

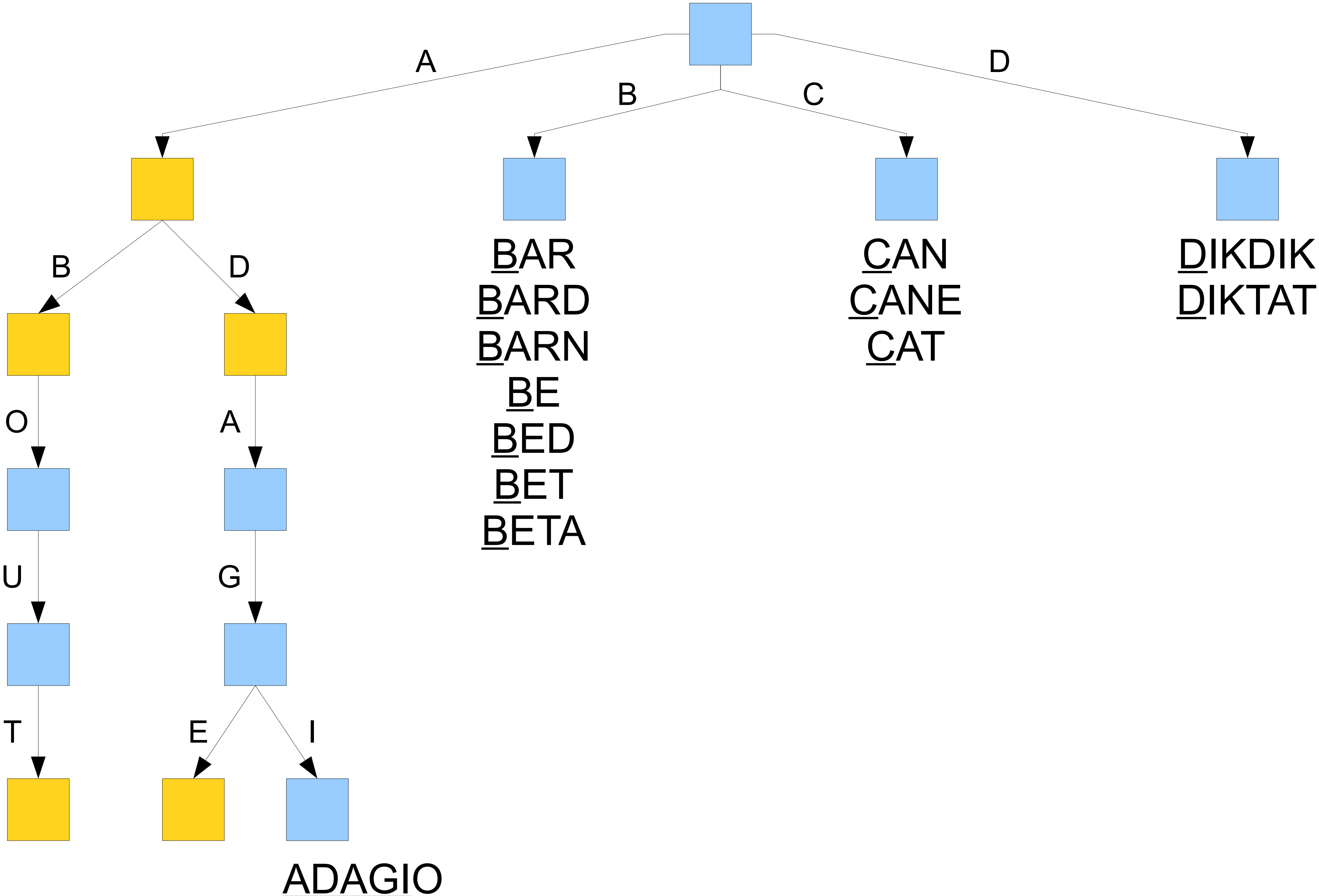

A
B
C
D
B
D
O
U
T
BAR
BARD
BARN
BE
BED
BET
BETA
CAN
CANE
CAT
DIKDIK
DIKTAT
AD
ADAGE
ADAGIO

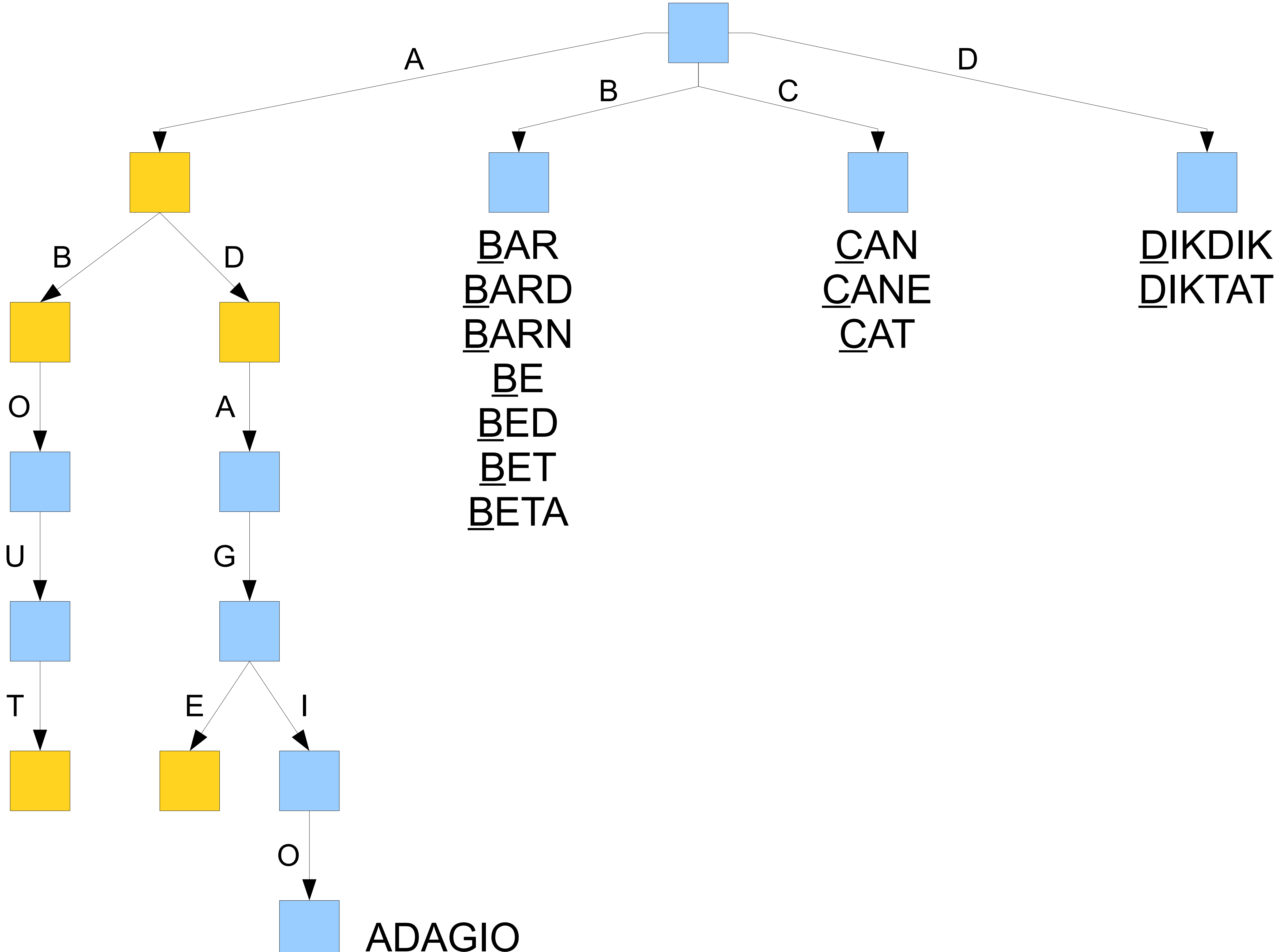

A
B
C
D
B
D
O
A
U
T
BAR
BARD
BARN
BE
BED
BET
BETA
CAN
CANE
CAT
DIKDIK
DIKTAT
ADAGE
ADAGIO

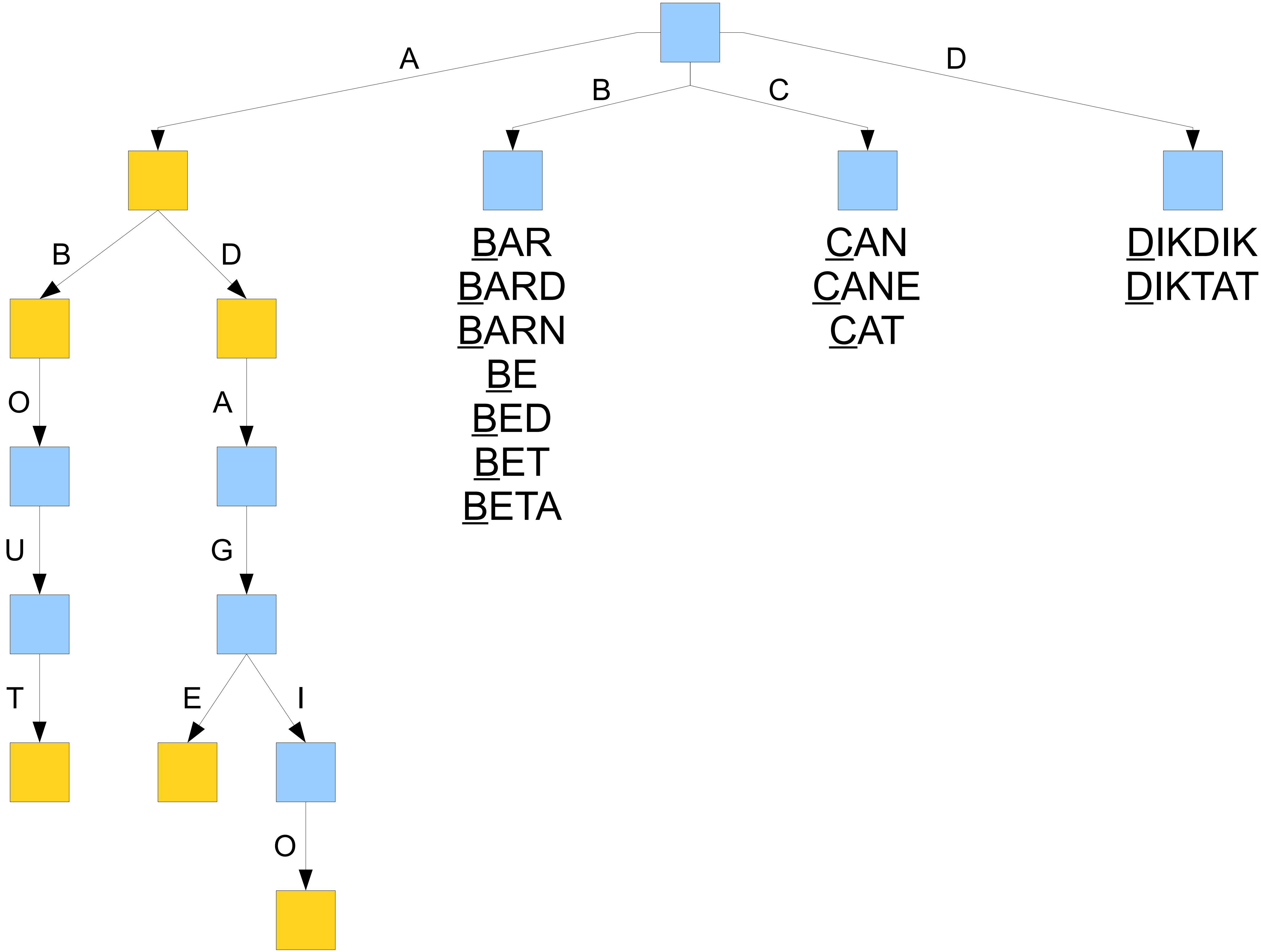

A
B
C
D
B
D
O
A
U
G
T
ADAGE
ADAGIO
BAR
BARD
BARN
BE
BED
BET
BETA
CAN
CANE
CAT
DIKDIK
DIKTAT

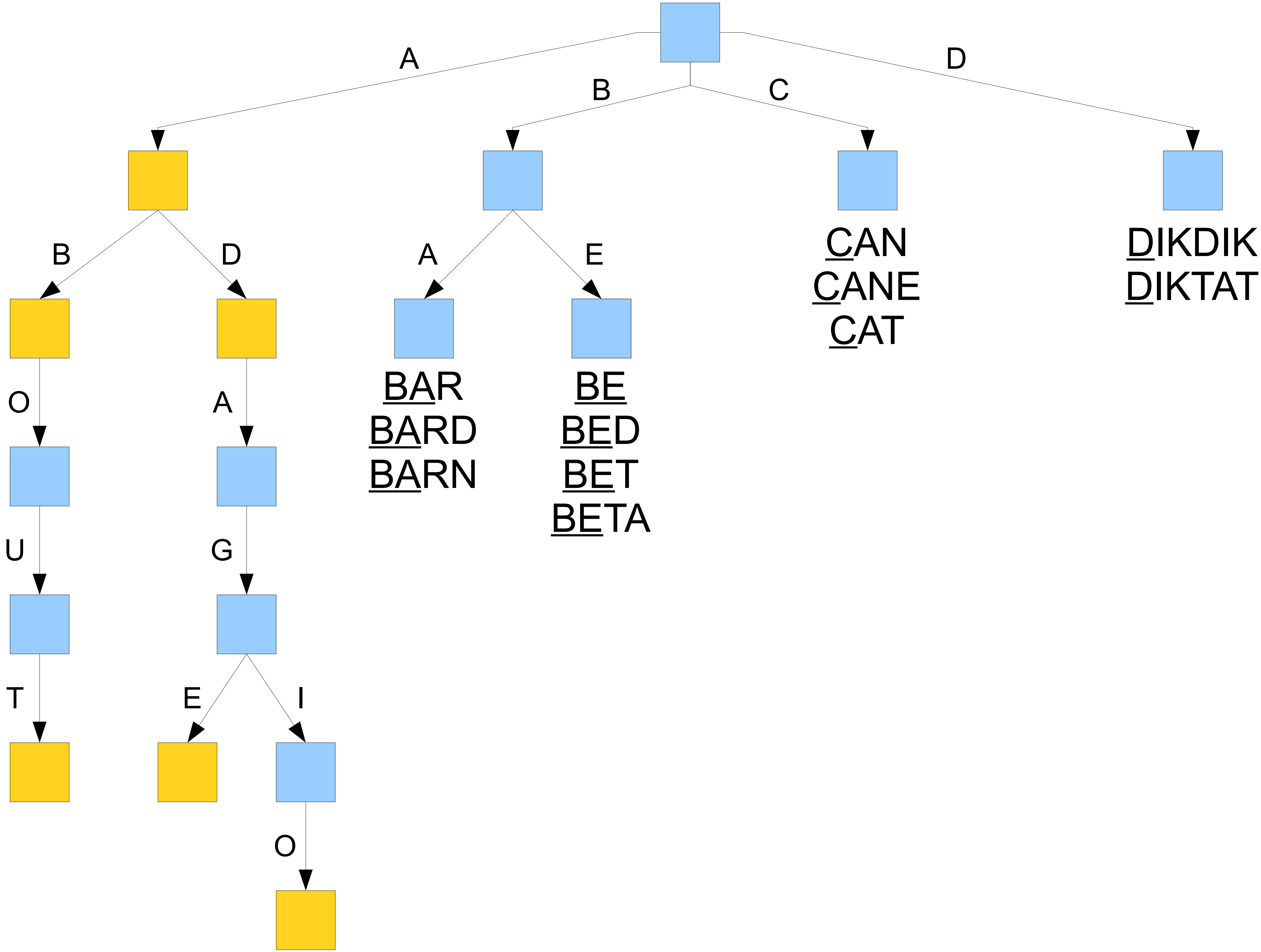


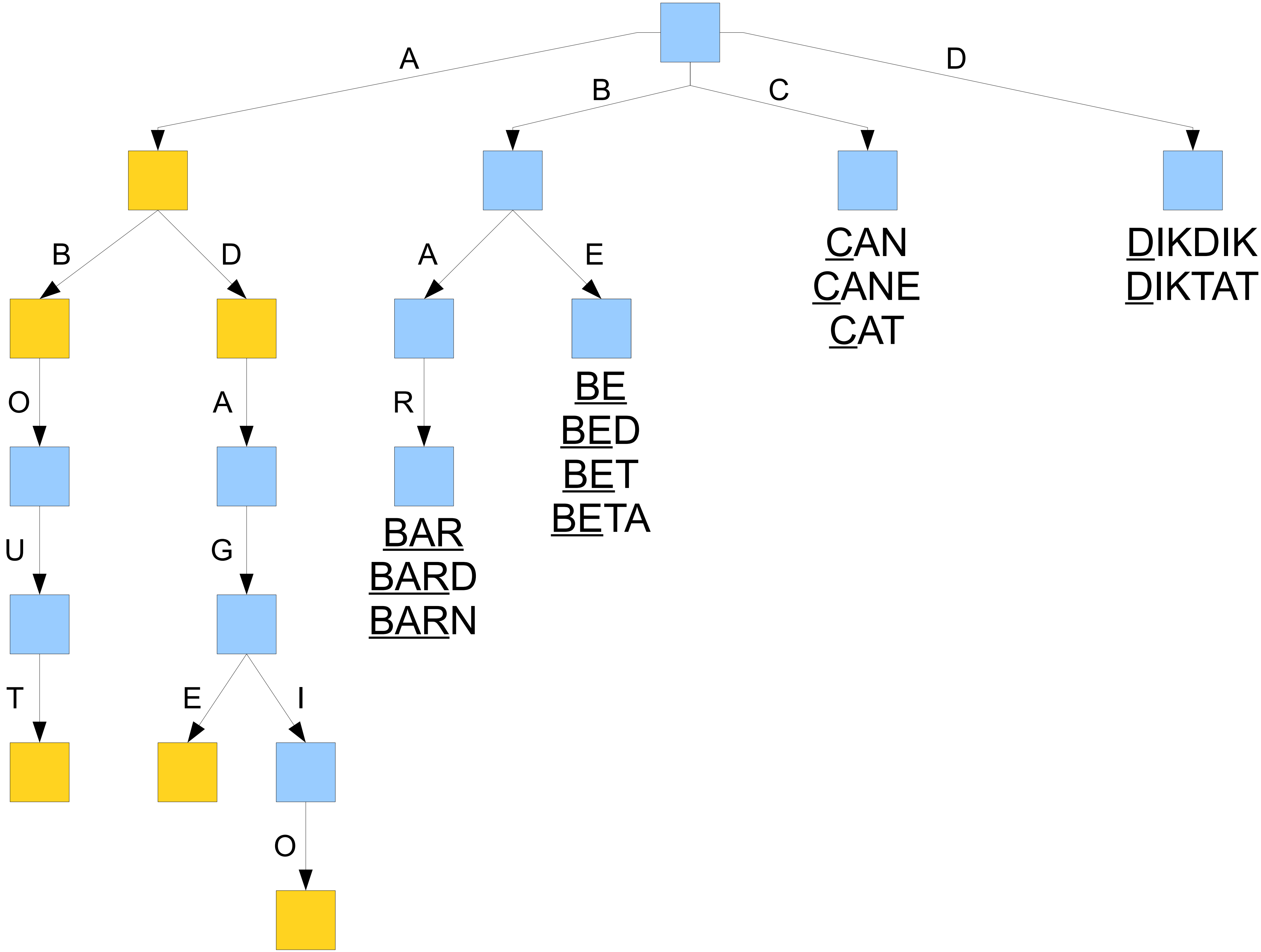
A
B
C
D
B
D
BAR
BARD
BARN
BE
BED
BET
BETA
CAN
CANE
CAT
DIKDIK
DIKTAT
O
A
U
G
T
E
I
ADAGE
ADAGIO

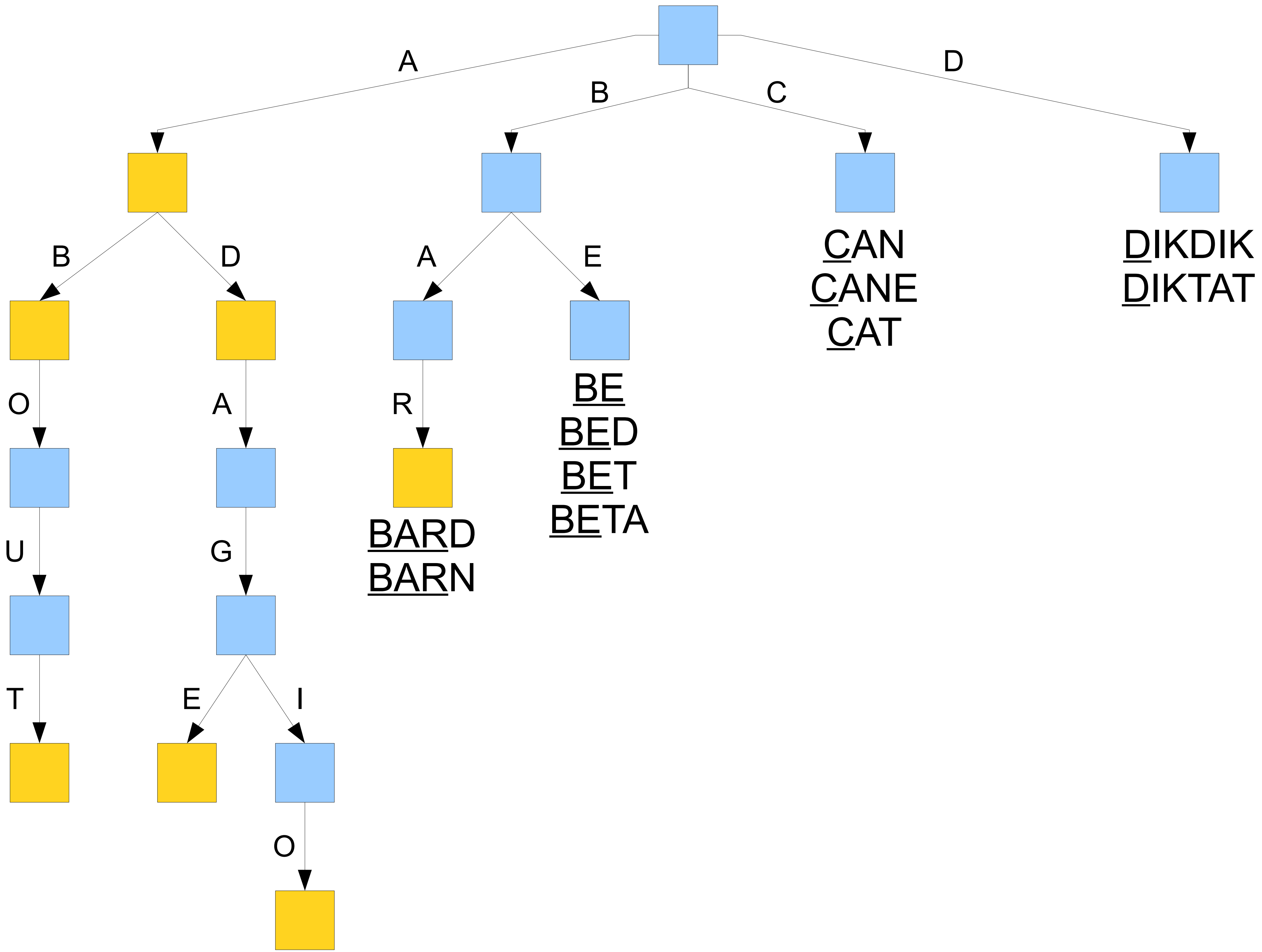

A
B
C
D
B
D
O
A
U
G
T
E
I
BAR
BARD
BARN
BE
BED
BET
BETA
CAN
CANE
CAT
DIKDIK
DIKTAT
ADAGIO

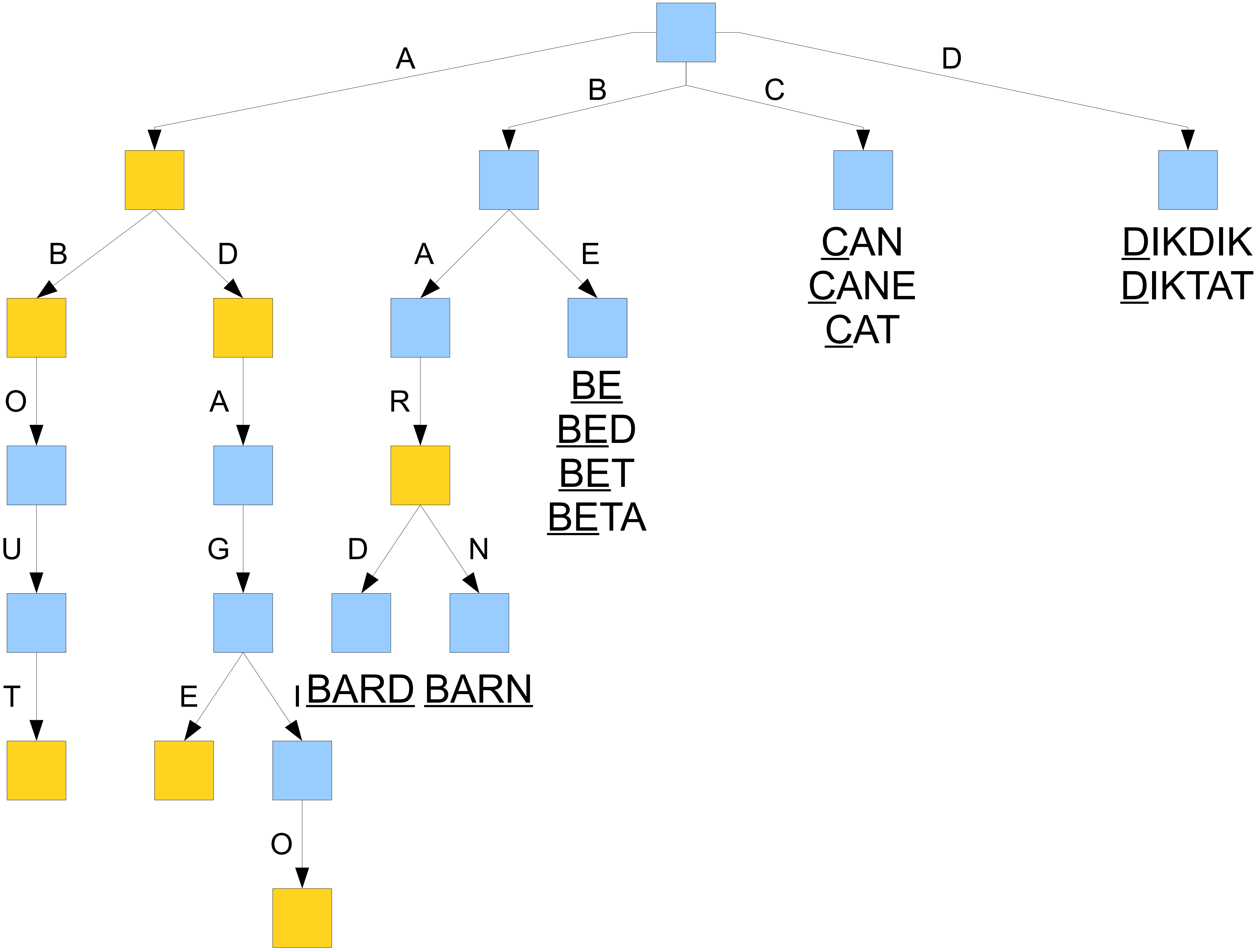

A
B
C
D
B
D
O
A
U
G
T
E
I
O
BAR
BARD
BARN
BE
BED
BET
BETA
CAN
CANE
CAT
DIKDIK
DIKTAT
ADAGIO

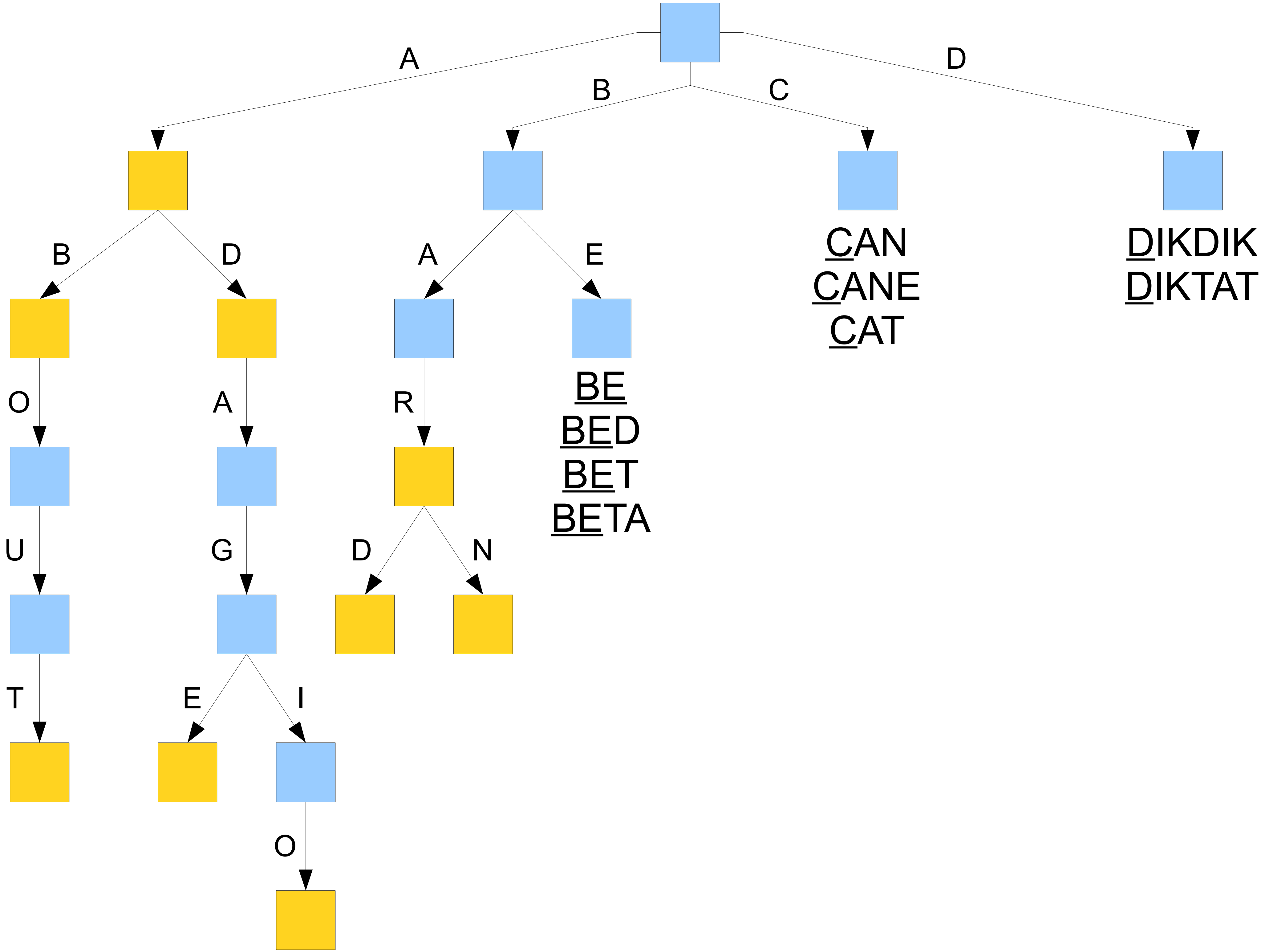

A
B
C
D
B
D
O
A
U
G
T
E
I
O
BAR
BARD
BARN
BE
BED
BET
BETA
CAN
CANE
CAT
DIKDIK
DIKTAT

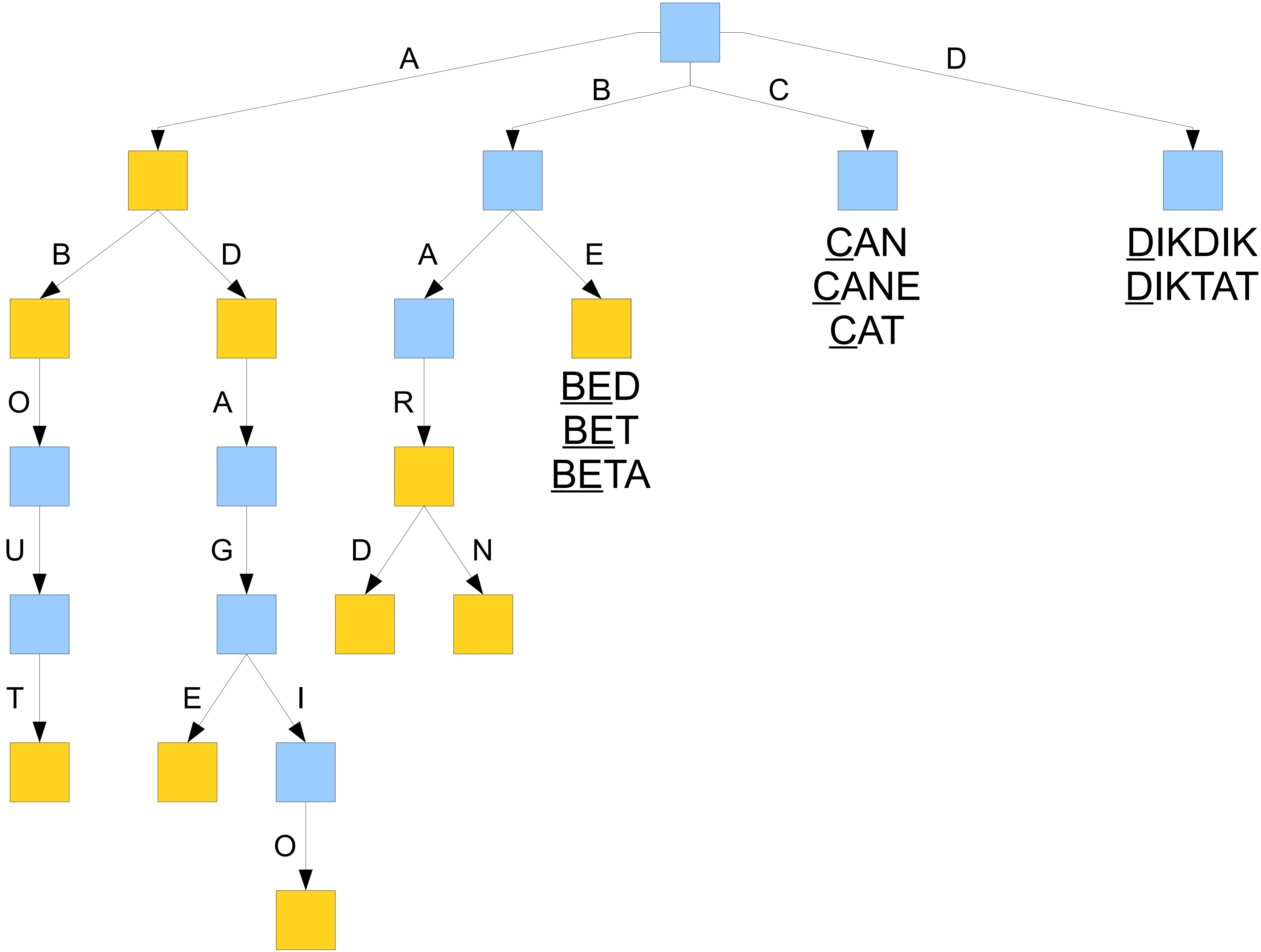

A
B
C
D
B
D
A
E
CAN
CANE
CAT
DIKDIK
DIKTAT
O
A
BAR
BARD
BARN
BE
BED
BET
BETA
U
G
T
E
I
O

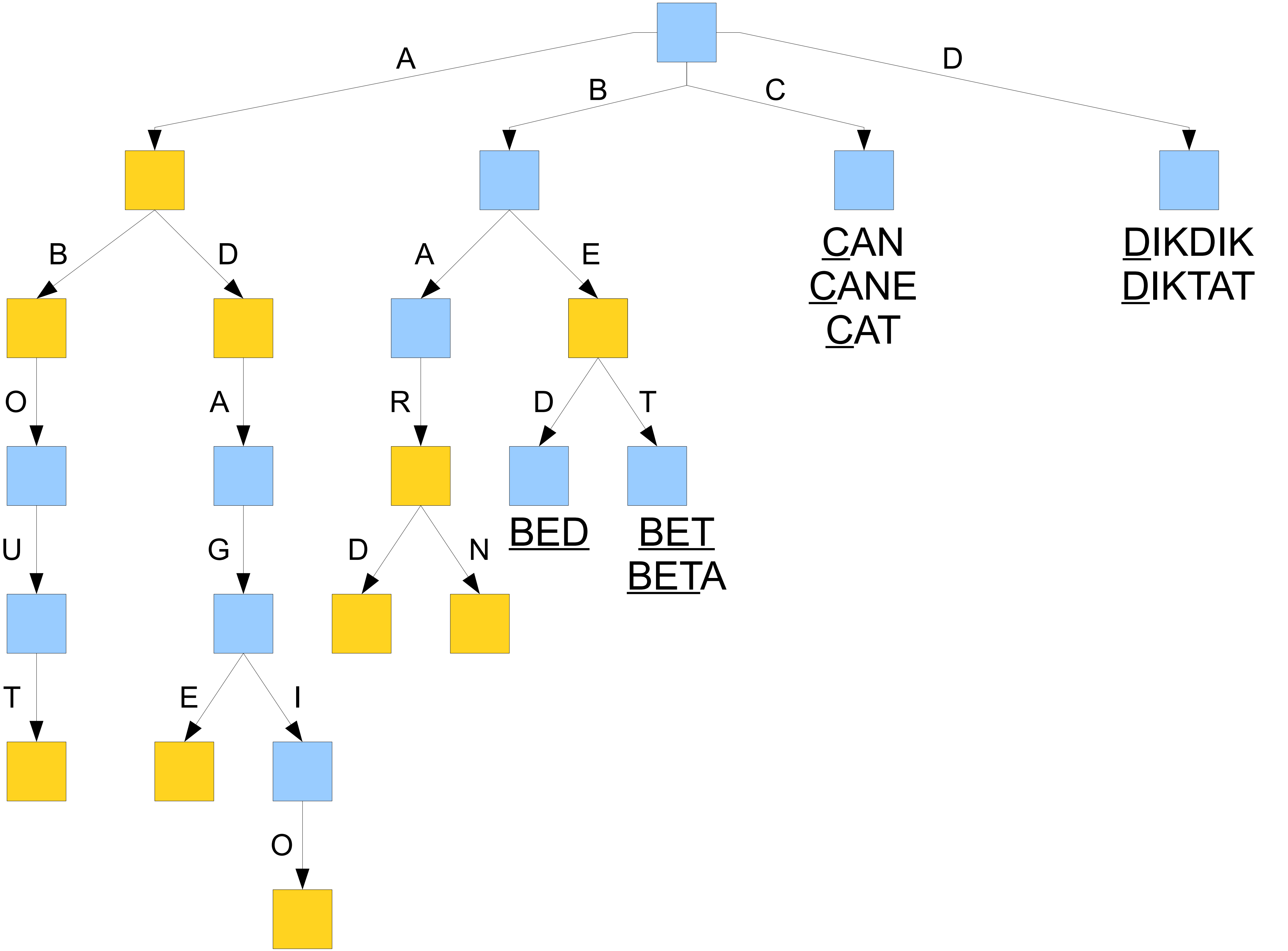

A
B
C
D
B
D
A
E
CAN
CANE
CAT
DIKDIK
DIKTAT
O
A
R
BE
BED
BET
BETA
U
G
BAR
BARD
BARN
T
E
I
O

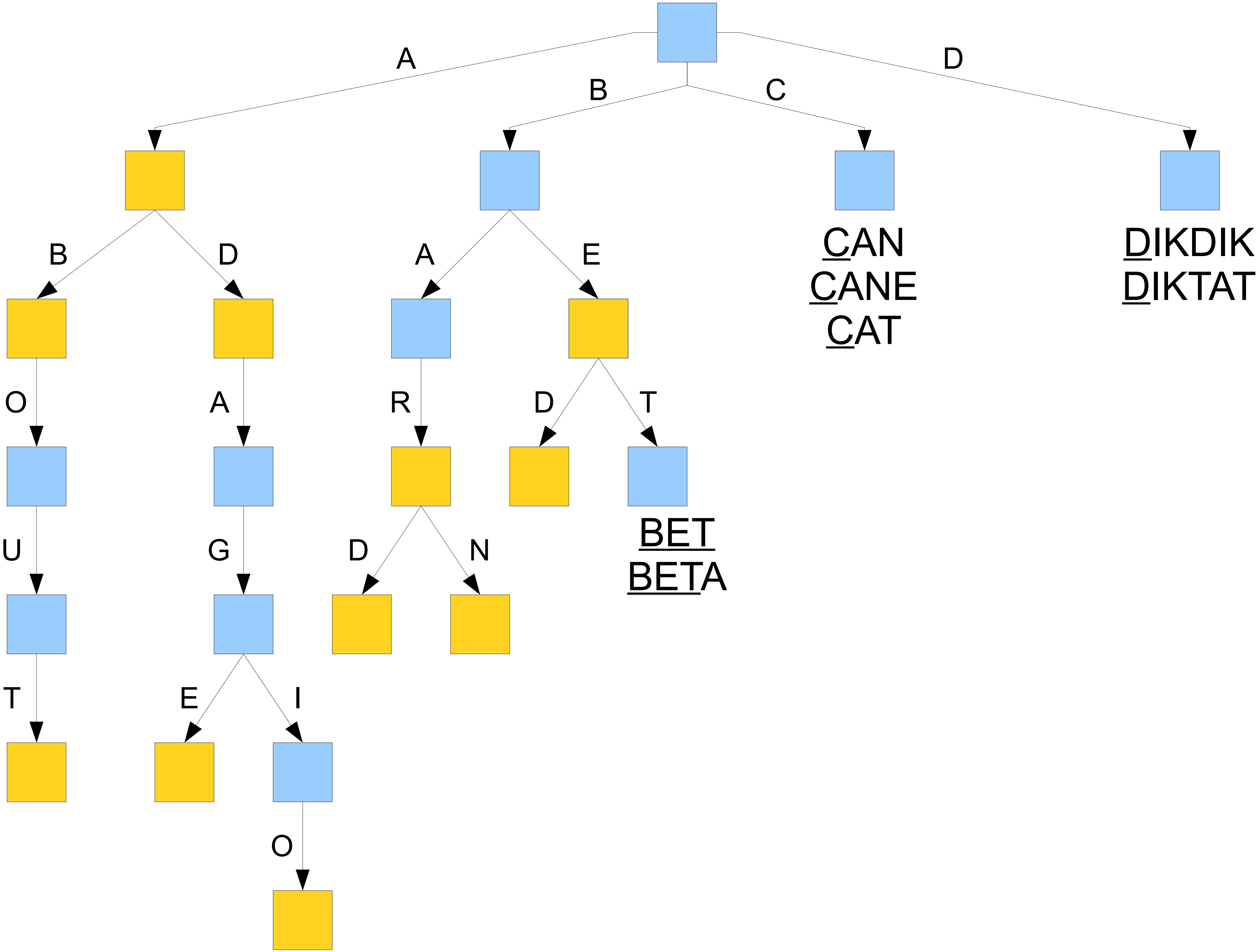

A
B
C
D
B
D
A
E
CAN
CANE
CAT
DIKDIK
DIKTAT
O
A
R
BE
BED
BET
BETA
BARD
BARN
U
G
T
E
I
O

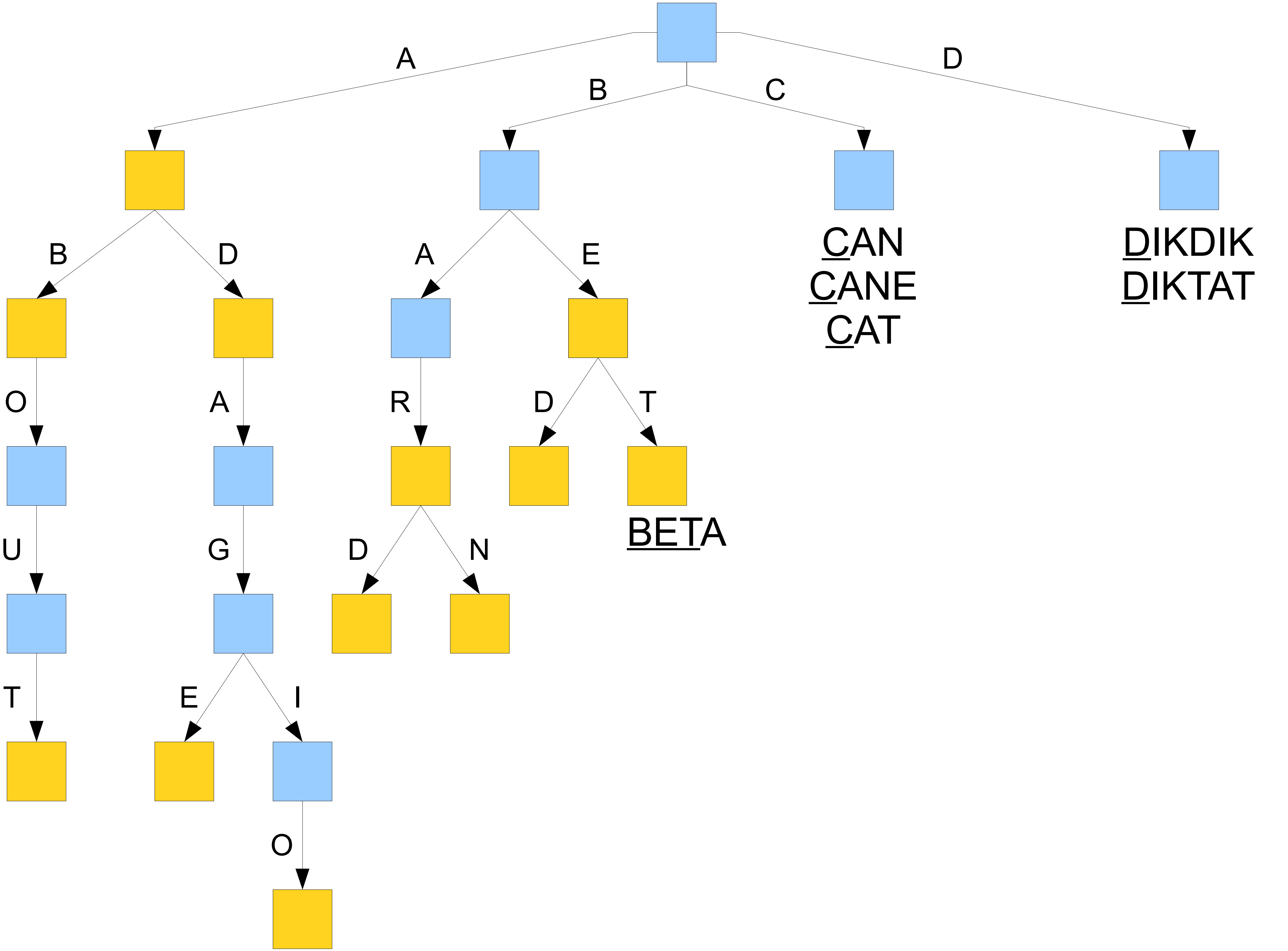

A
B
C
D
B
D
A
E
CAN
CANE
CAT
DIKDIK
DIKTAT
O
A
R
BE
BED
BET
BETA
U
G
D
N
T
E
I
BARD
BARN
O

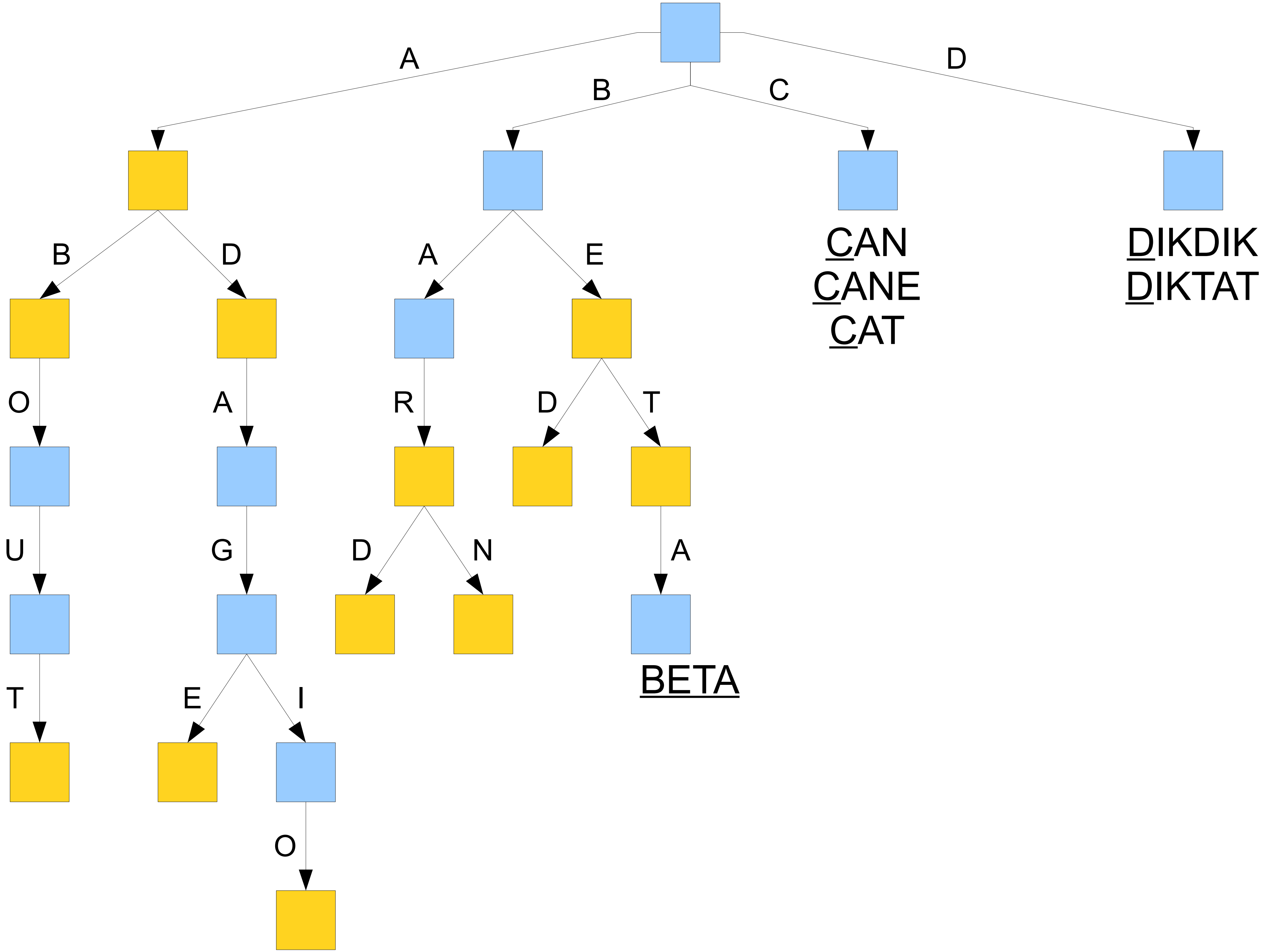

A
B
C
D
B
D
A
E
CAN
CANE
CAT
DIKDIK
DIKTAT
O
A
R
BE
BED
BET
BETA
U
G
D
N
T
E
I
O

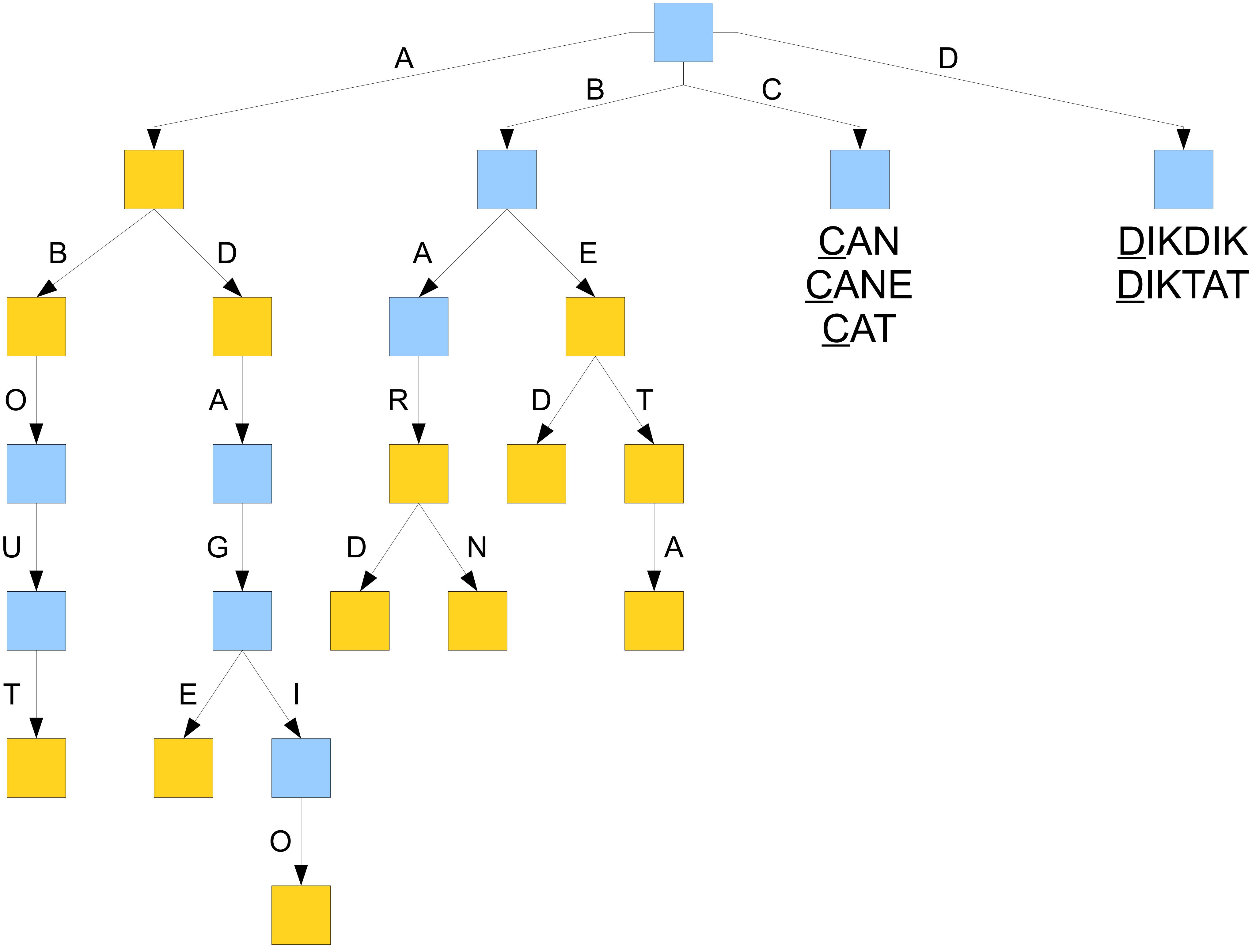

A
B
C
D
B
D
A
E
CAN
CANE
CAT
DIKDIK
DIKTAT
O
A
R
BED
BET
BETA
U
G
D
N
T
E
I
O

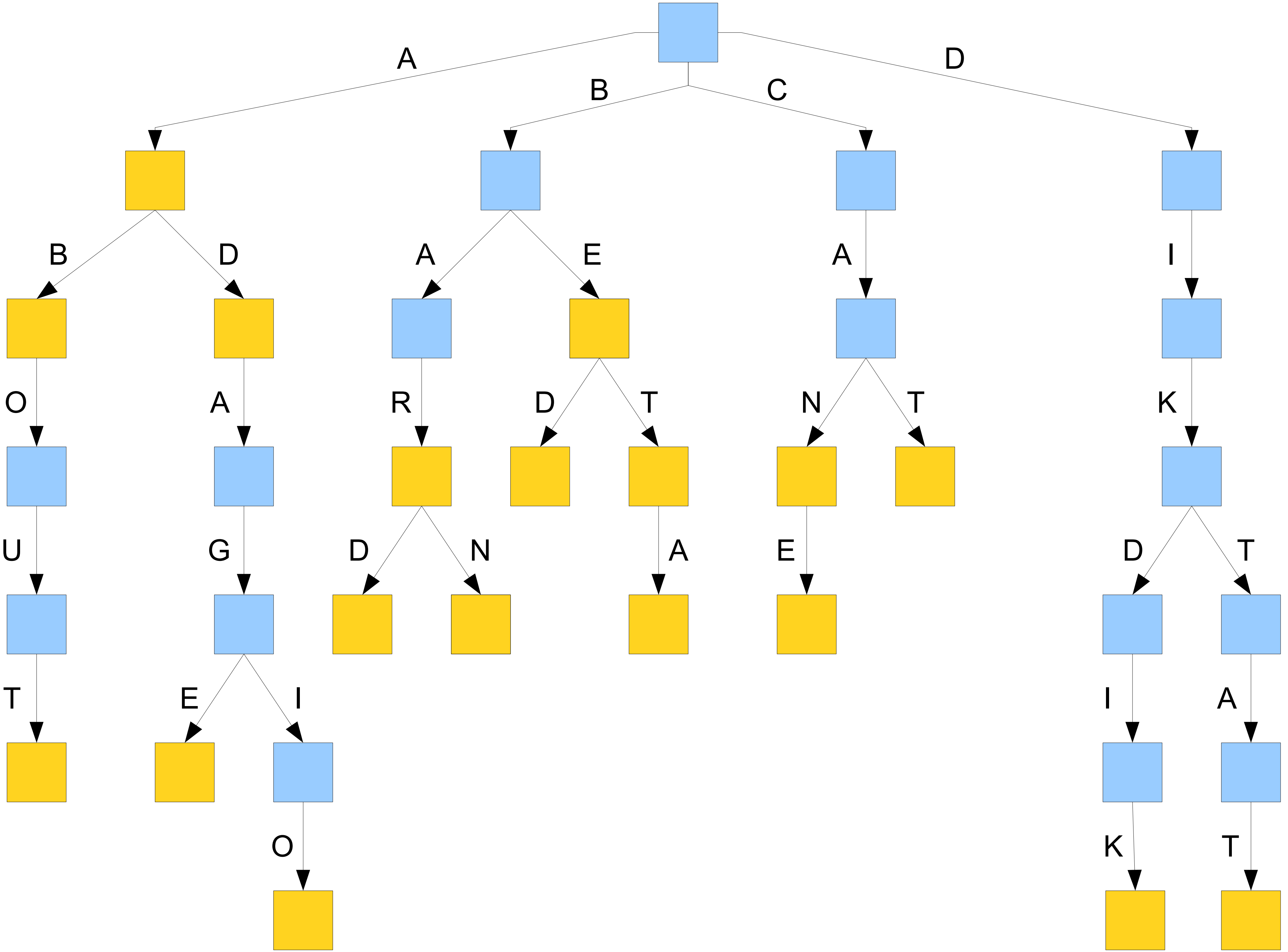

A
B
C
D
B
D
A
E
CAN
CANE
CAT
DIKDIK
DIKTAT
O
A
R
D
T
U
G
D
N
BED
BET
BETA
T
E
I
O

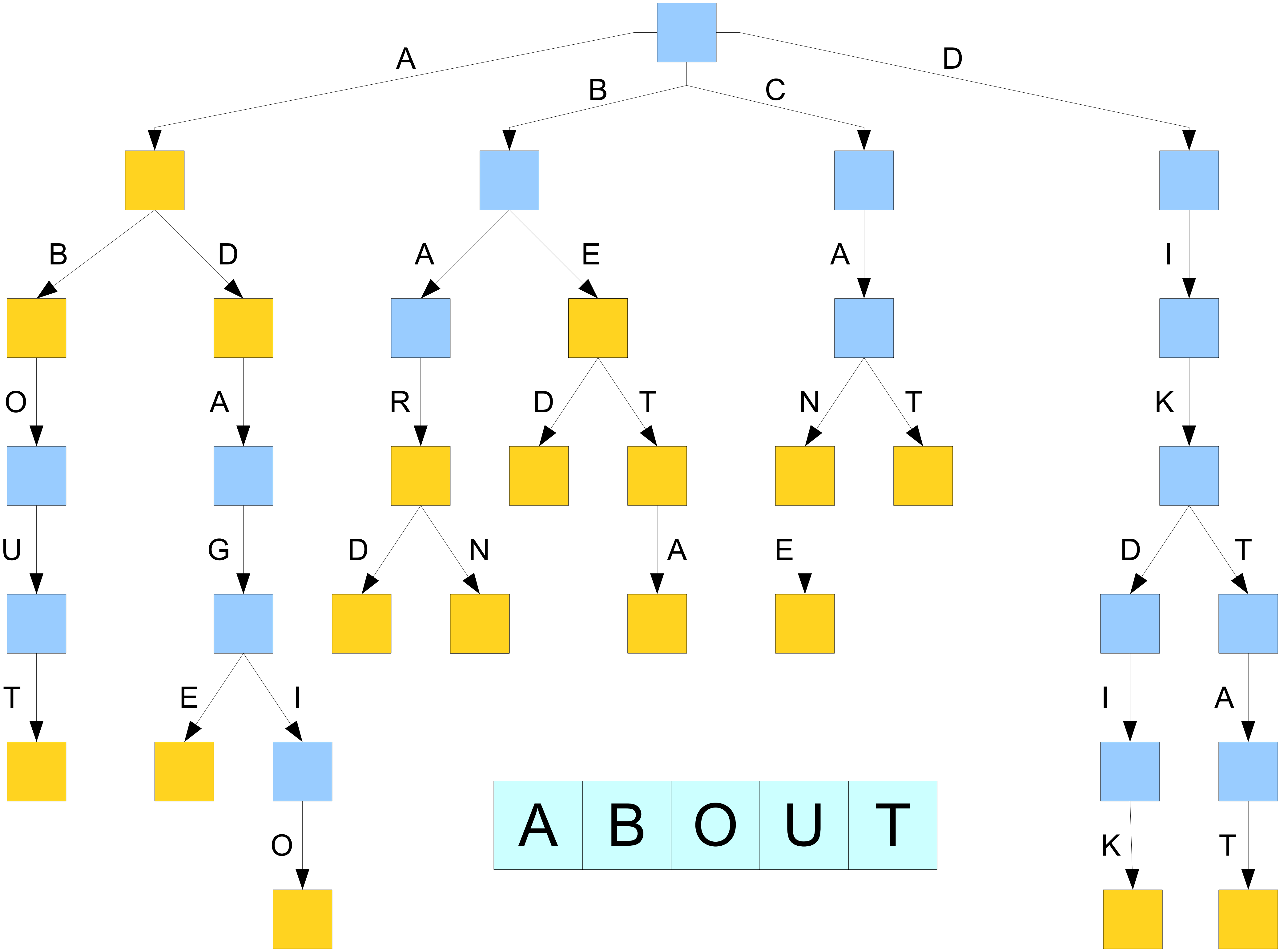

A
B
C
D
B
D
A
E
CAN
CANE
CAT
DIKDIK
DIKTAT
O
A
R
D
T
U
G
D
N
BET
BETA
T
E
I
O

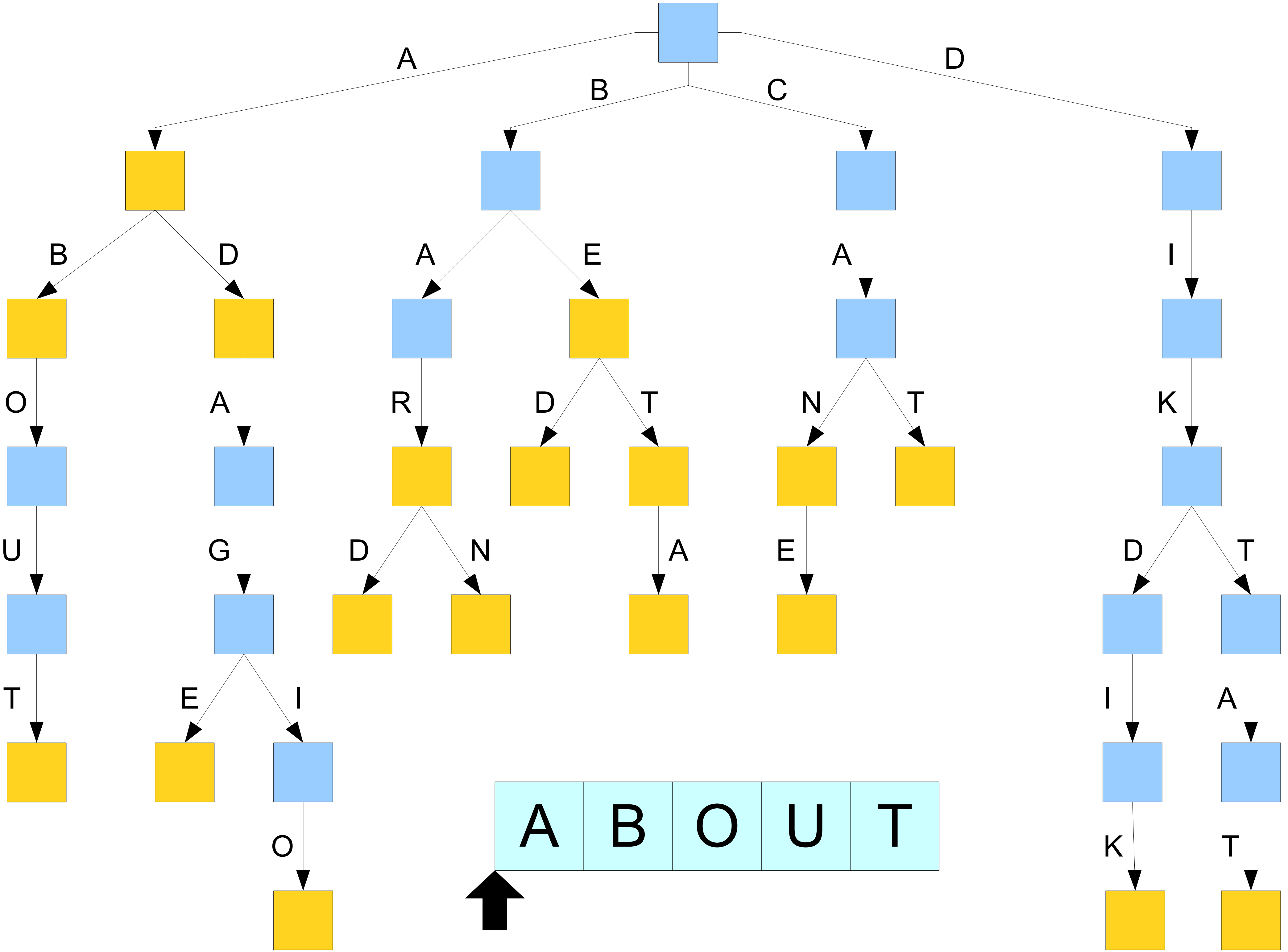

A
B
C
D
B
D
A
E
CAN
CANE
CAT
DIKDIK
DIKTAT
O
A
R
D
T
U
G
D
N
BETA
T
E
I
O

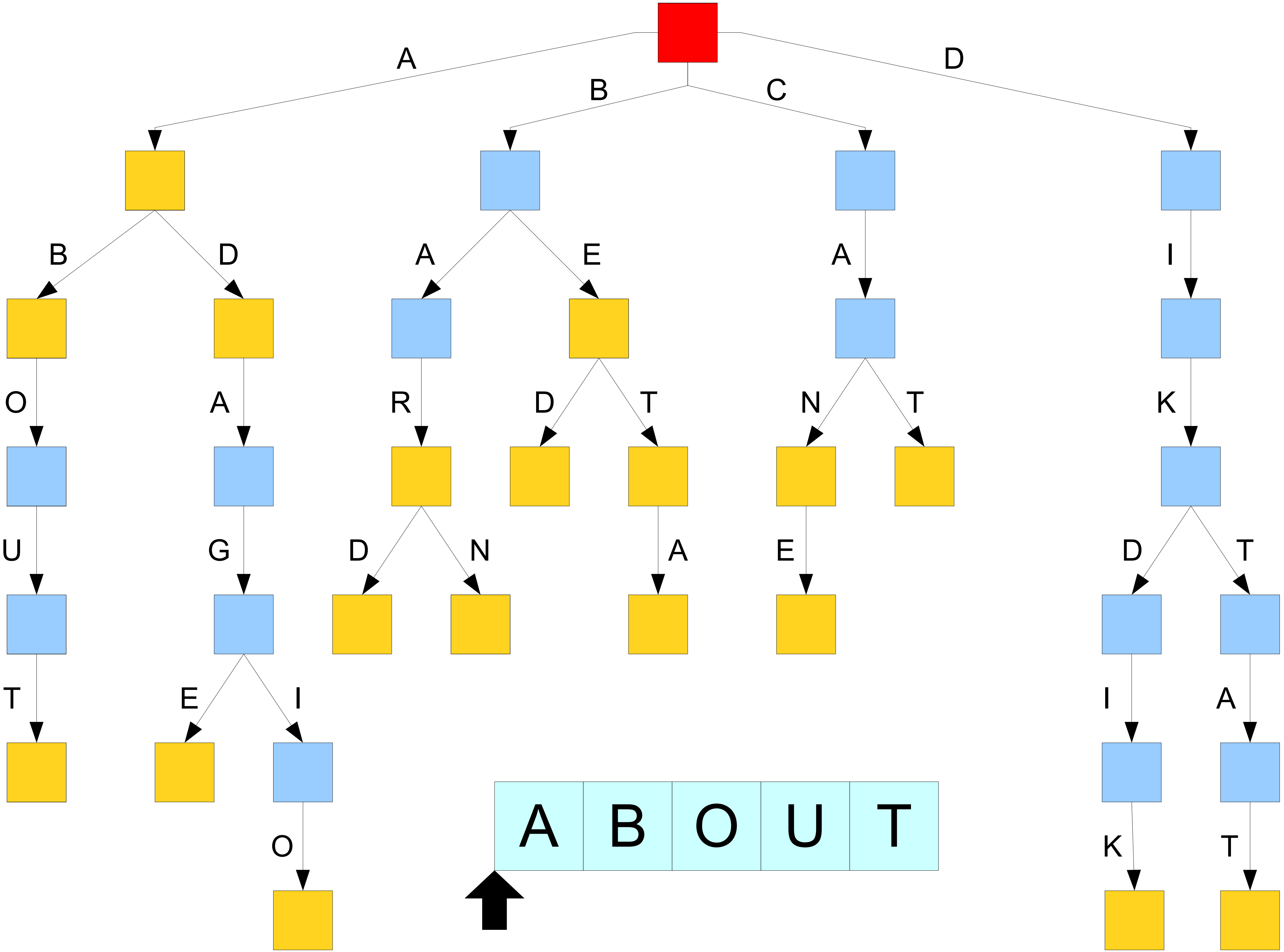

A
B
C
D
B
D
A
E
CAN
CANE
CAT
DIKDIK
DIKTAT
O
A
R
D
T
U
G
D
N
A
BETA
T
E
I
O

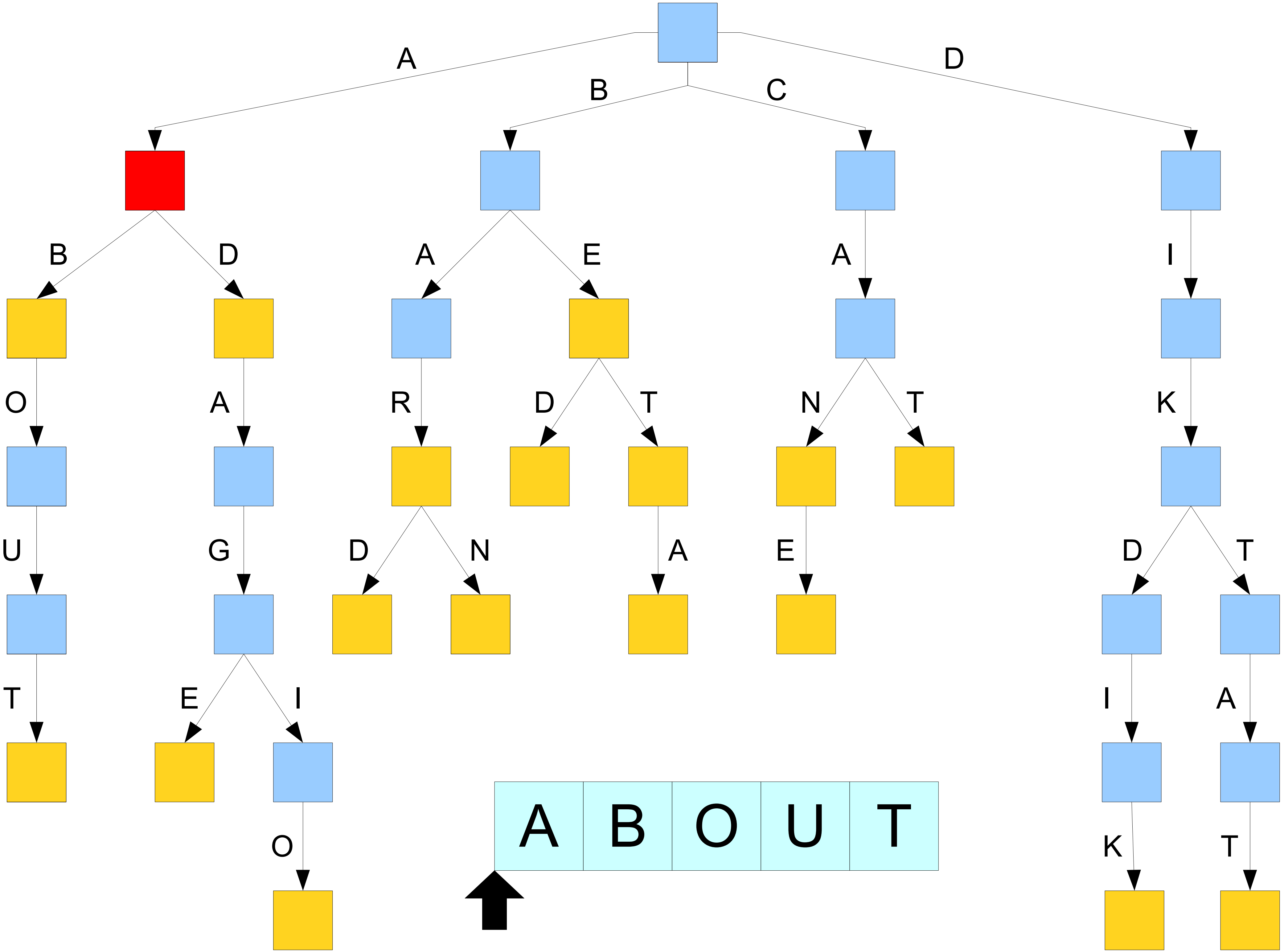

A
B
C
D
B
D
A
E
CAN
CANE
CAT
DIKDIK
DIKTAT
O
A
R
D
T
U
G
D
N
A
T
E
I
O


A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T

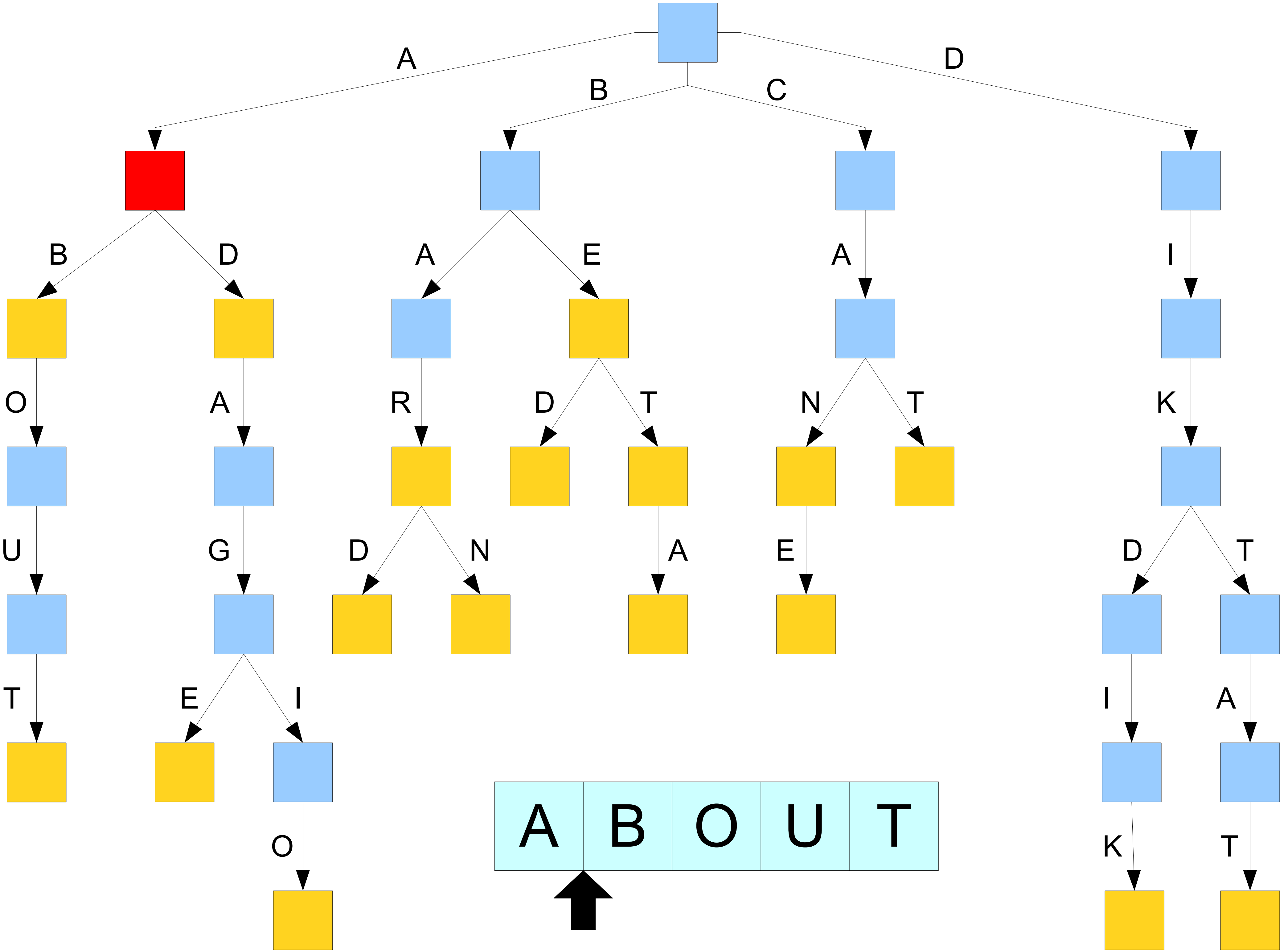

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A B O U T

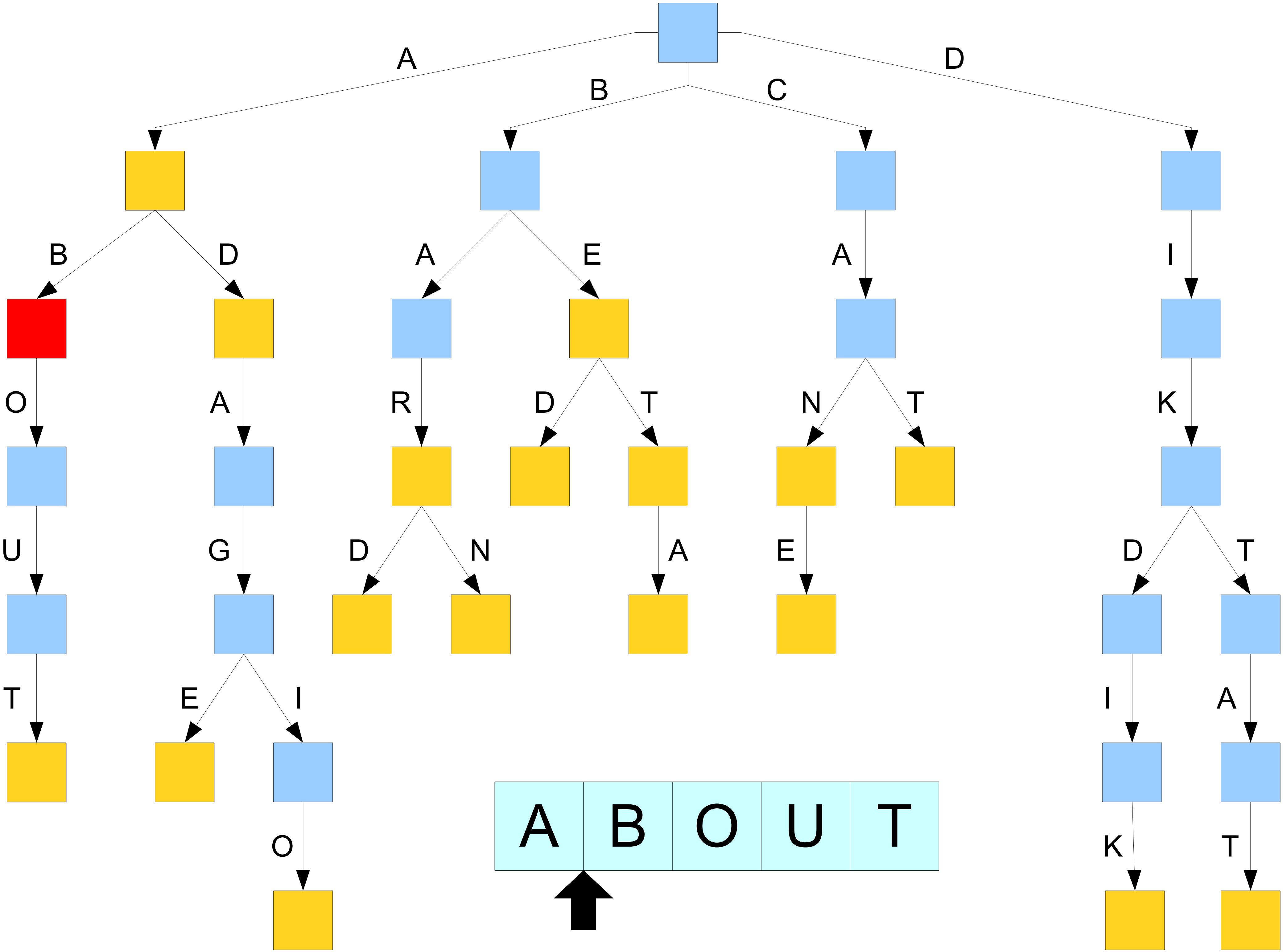

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A B O U T


A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A
B
O
U
T

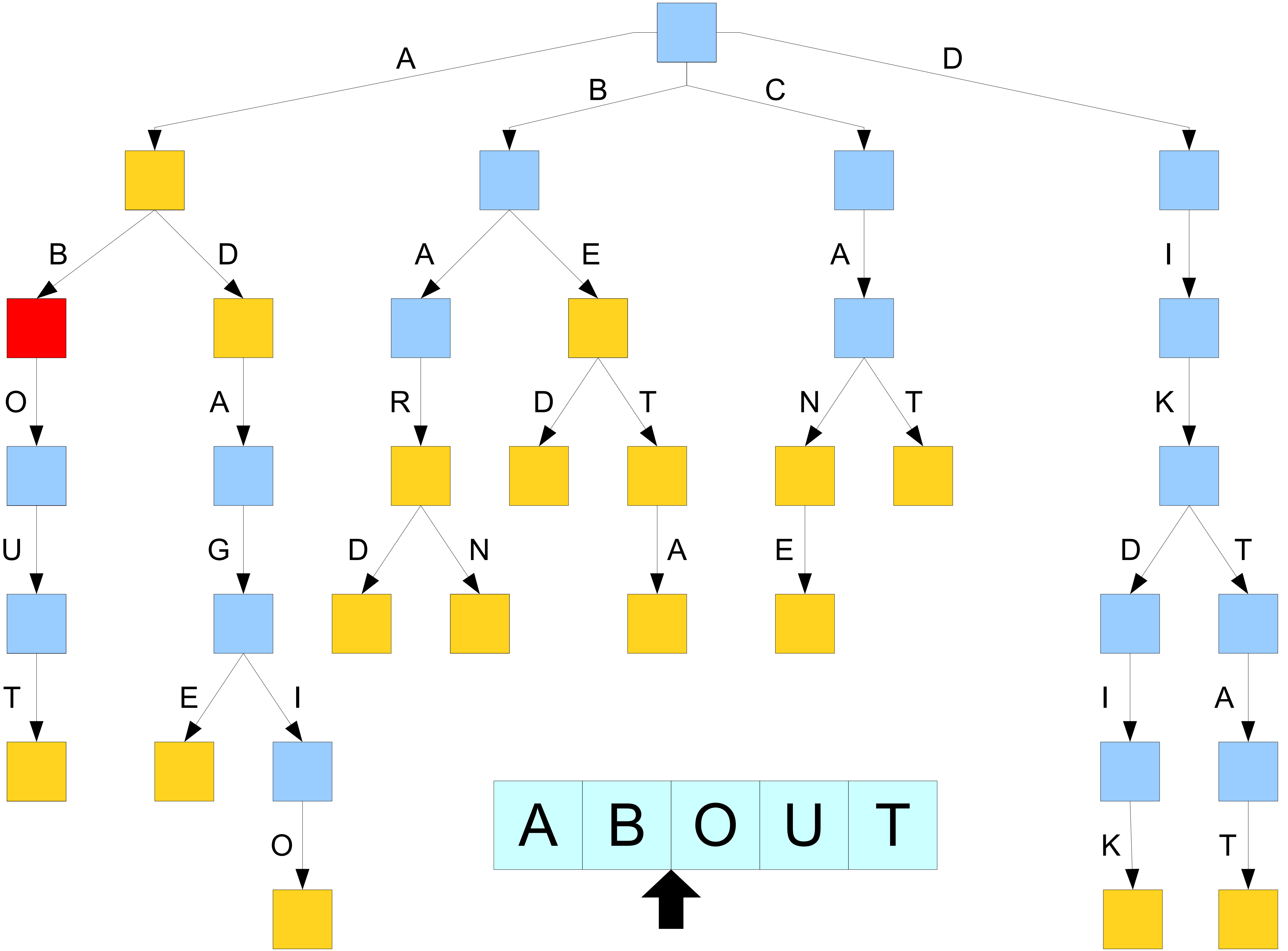


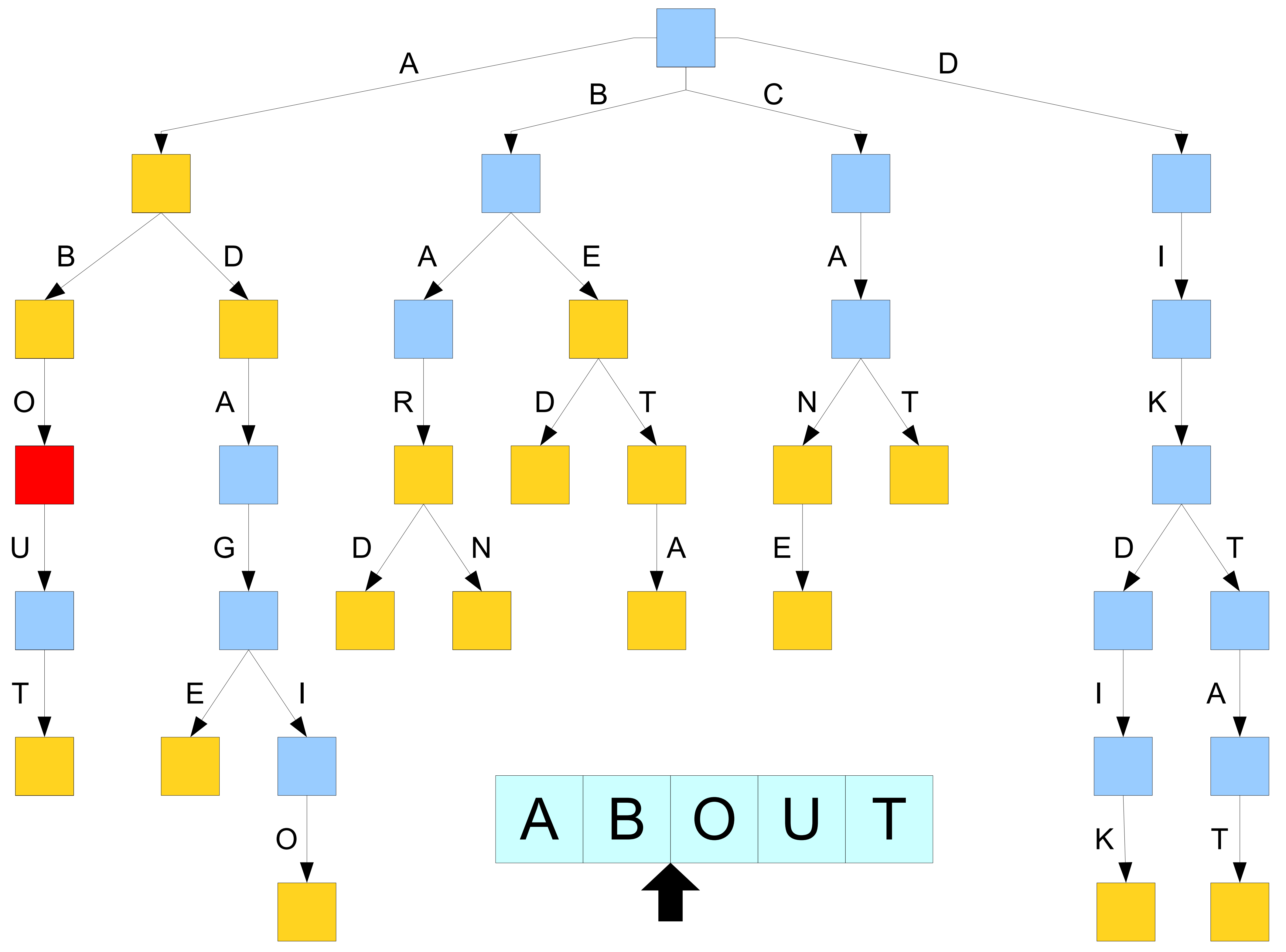
A B C D
B D A E A I
O A R D T N T K
U G D N A E D T
T E I I A
O K T
A B O U T

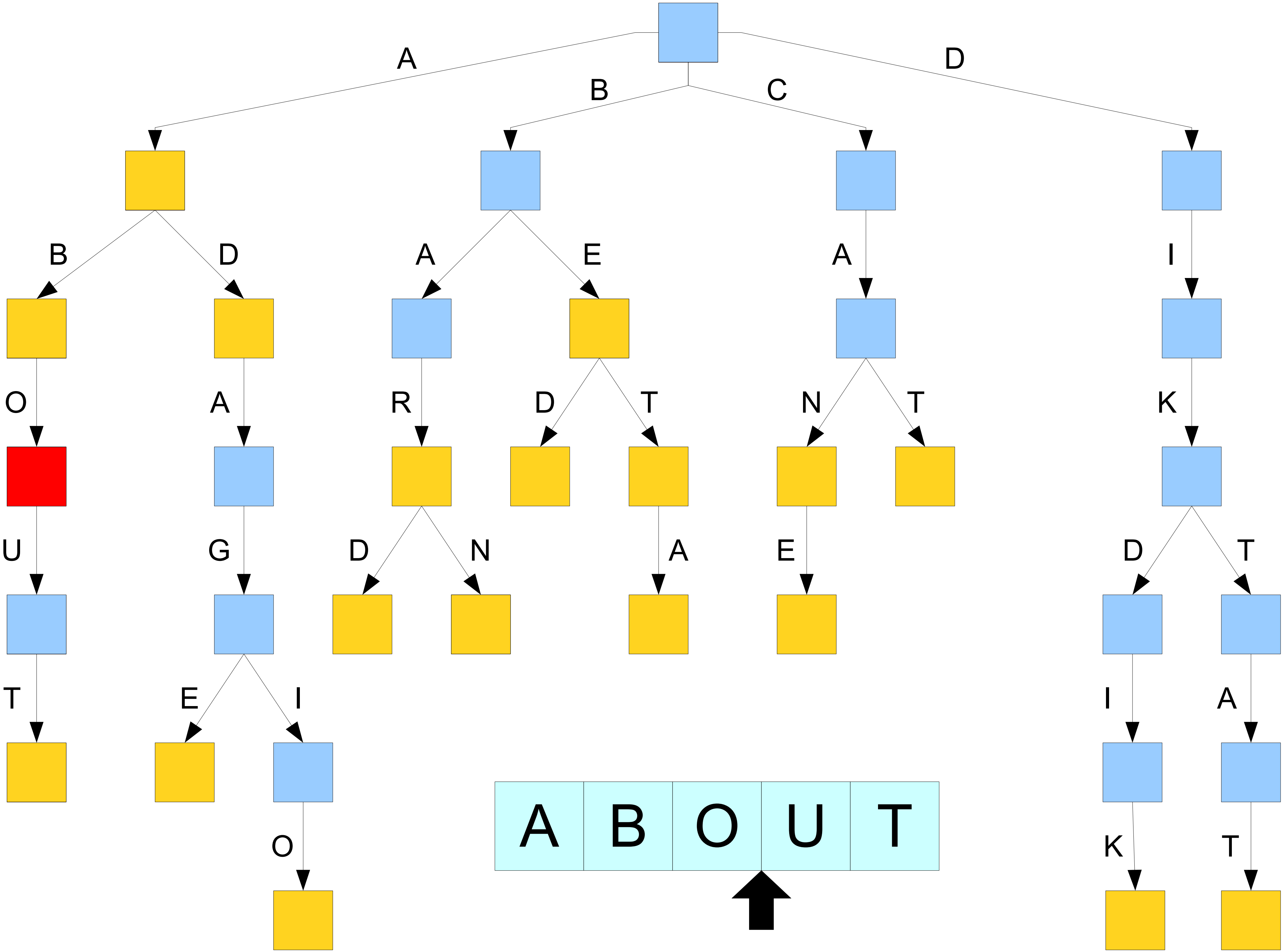

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A B O U T



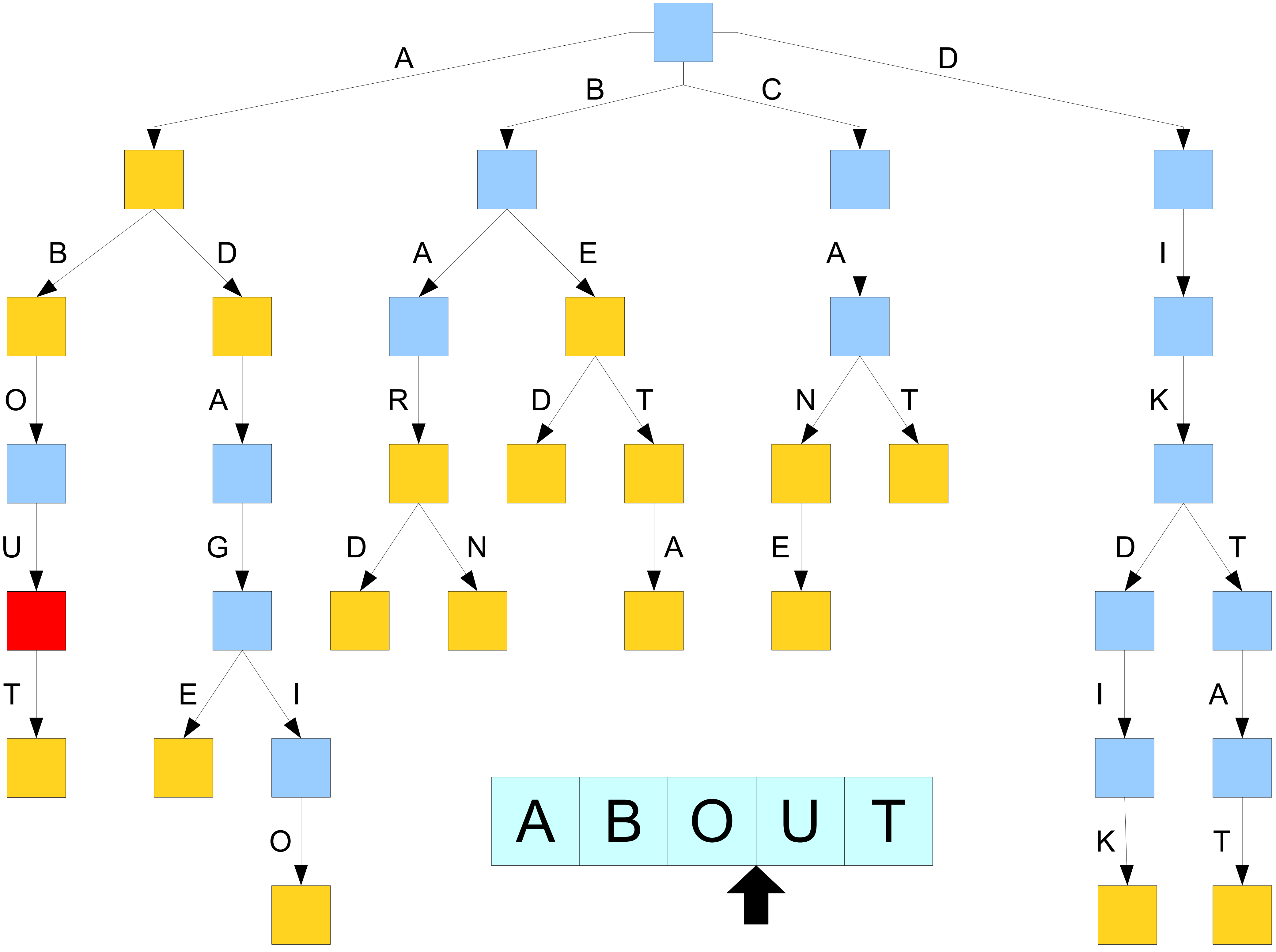
A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A
B
O
U
T

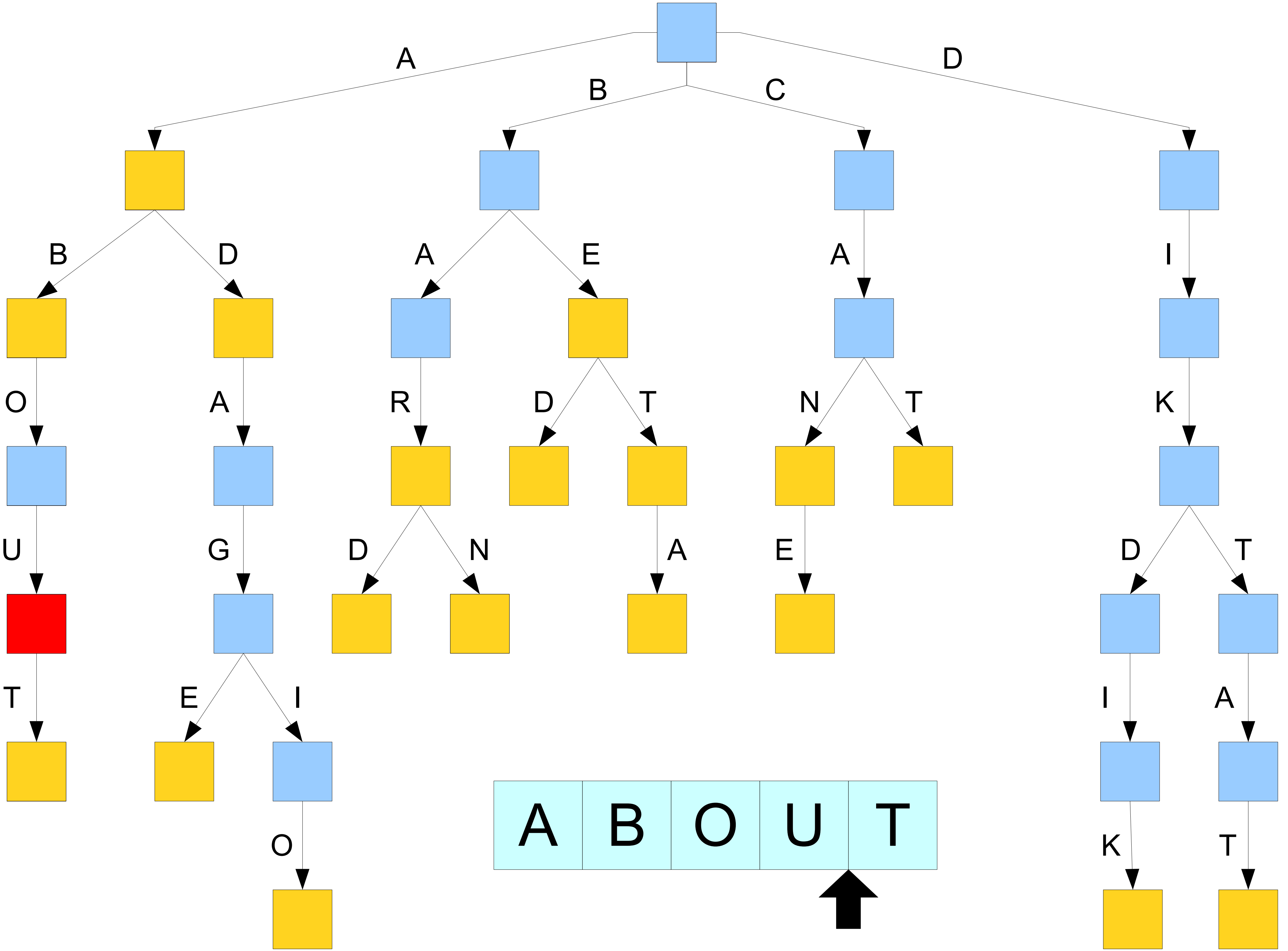

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A B O U T

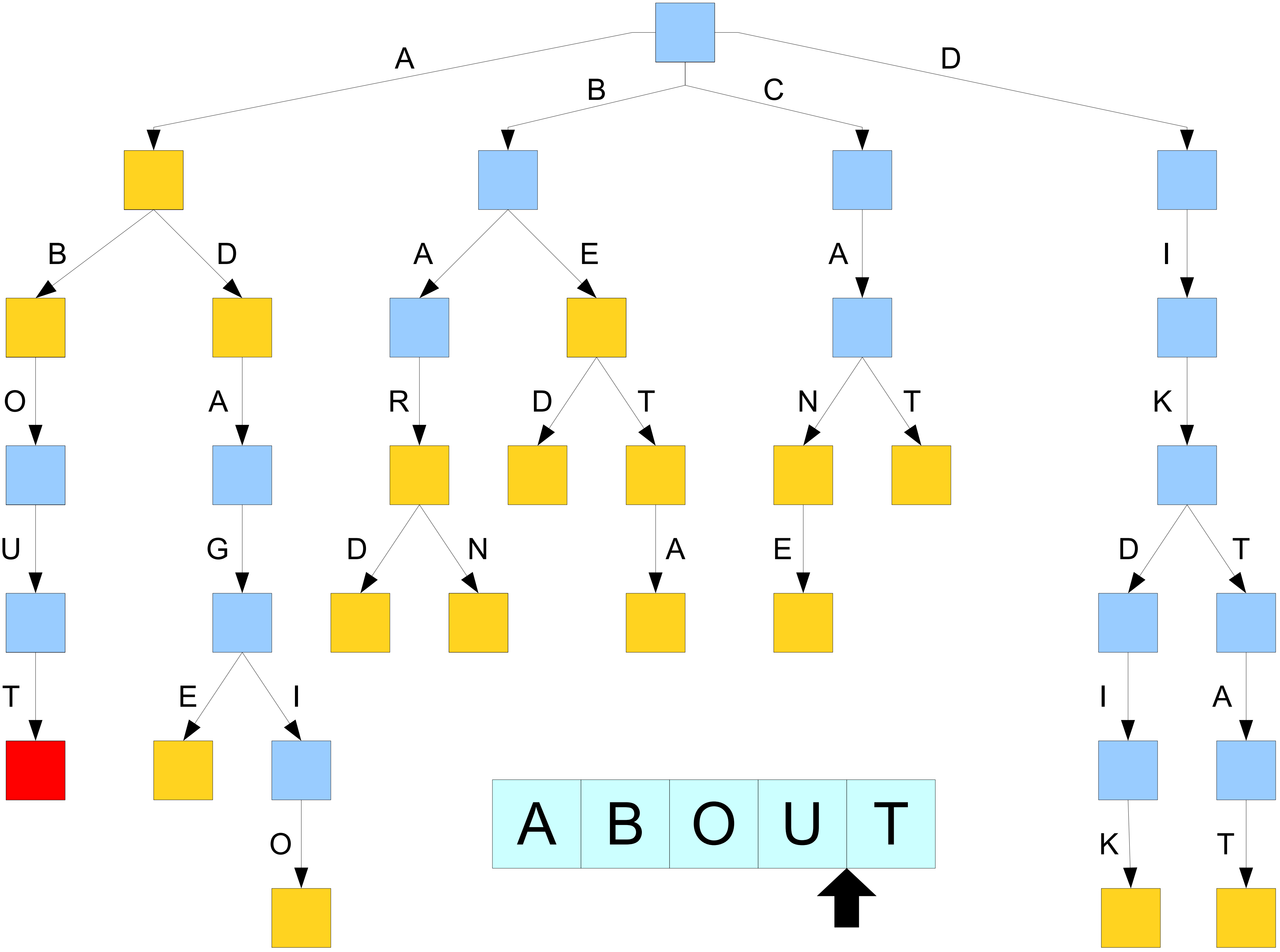

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A B O U T

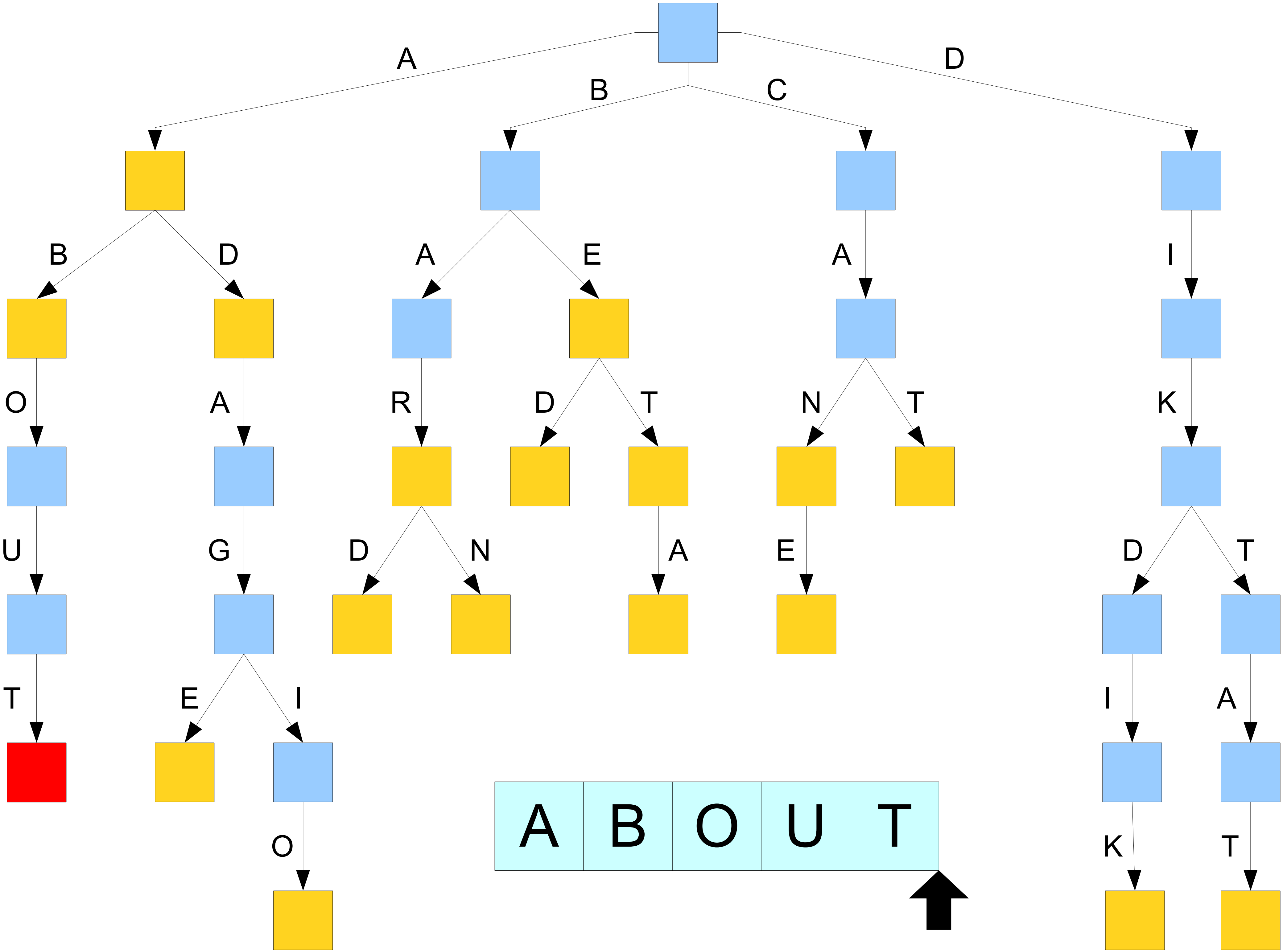


A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A B O U T

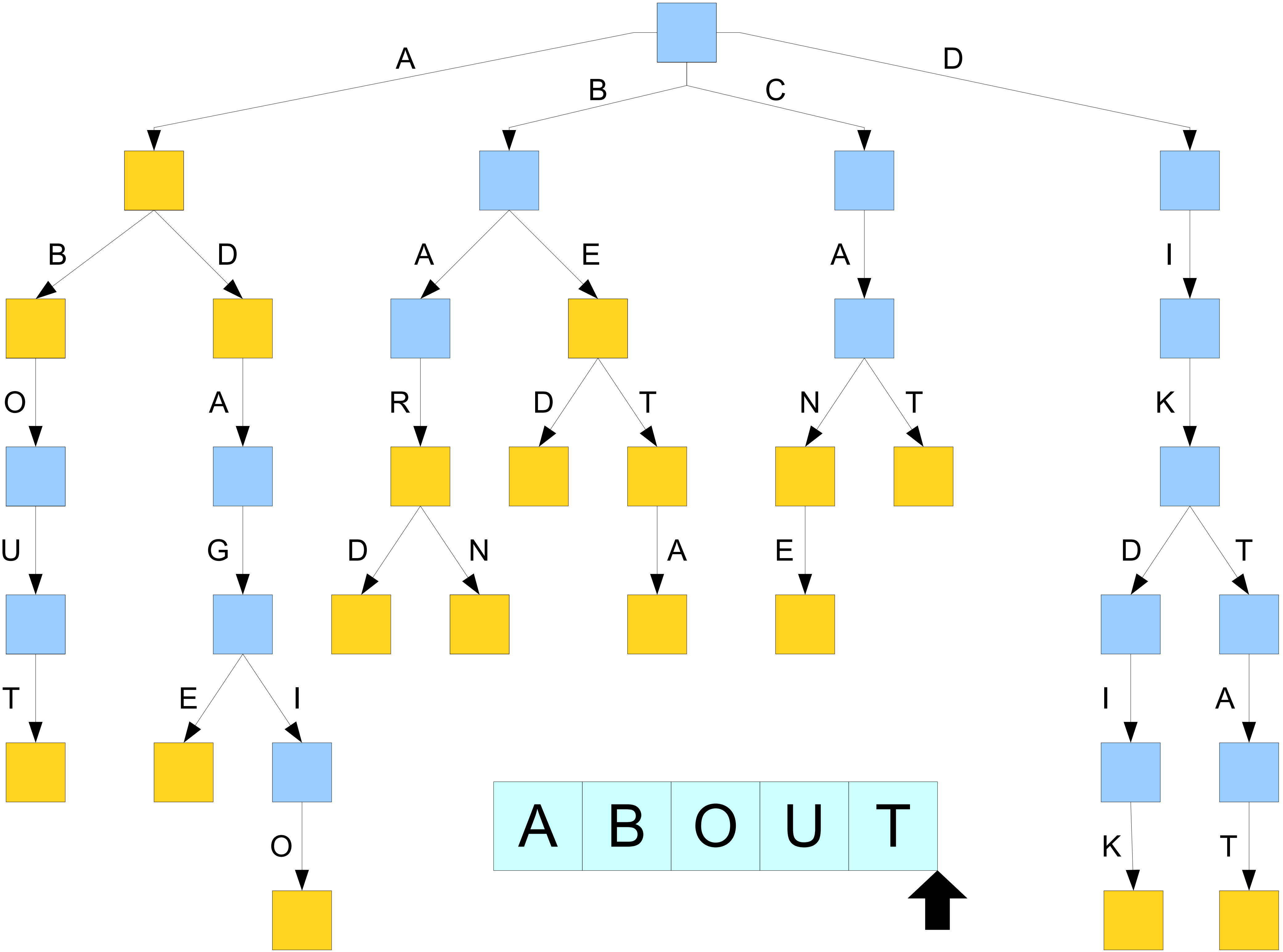

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A B O U T

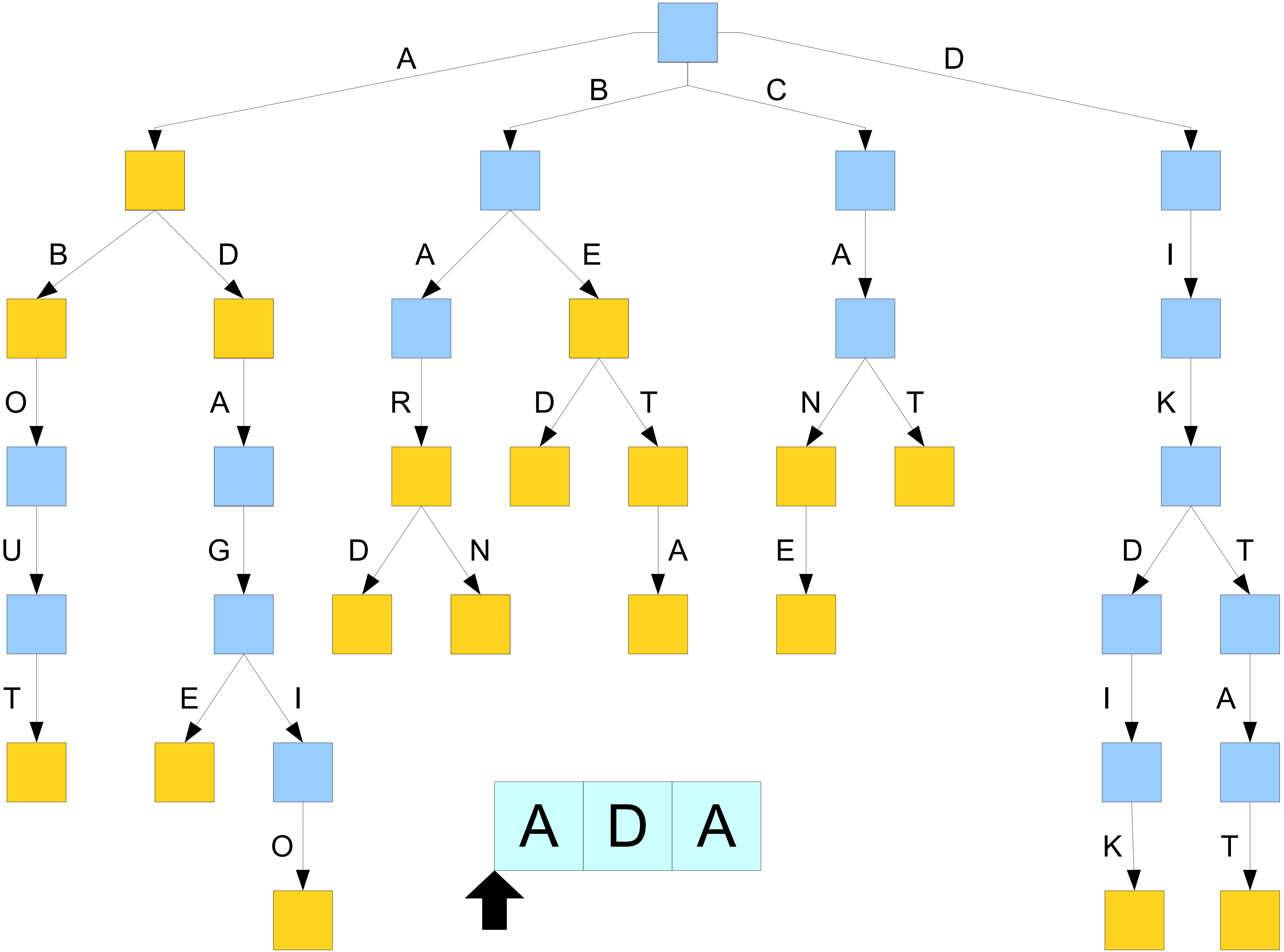

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A B O U T

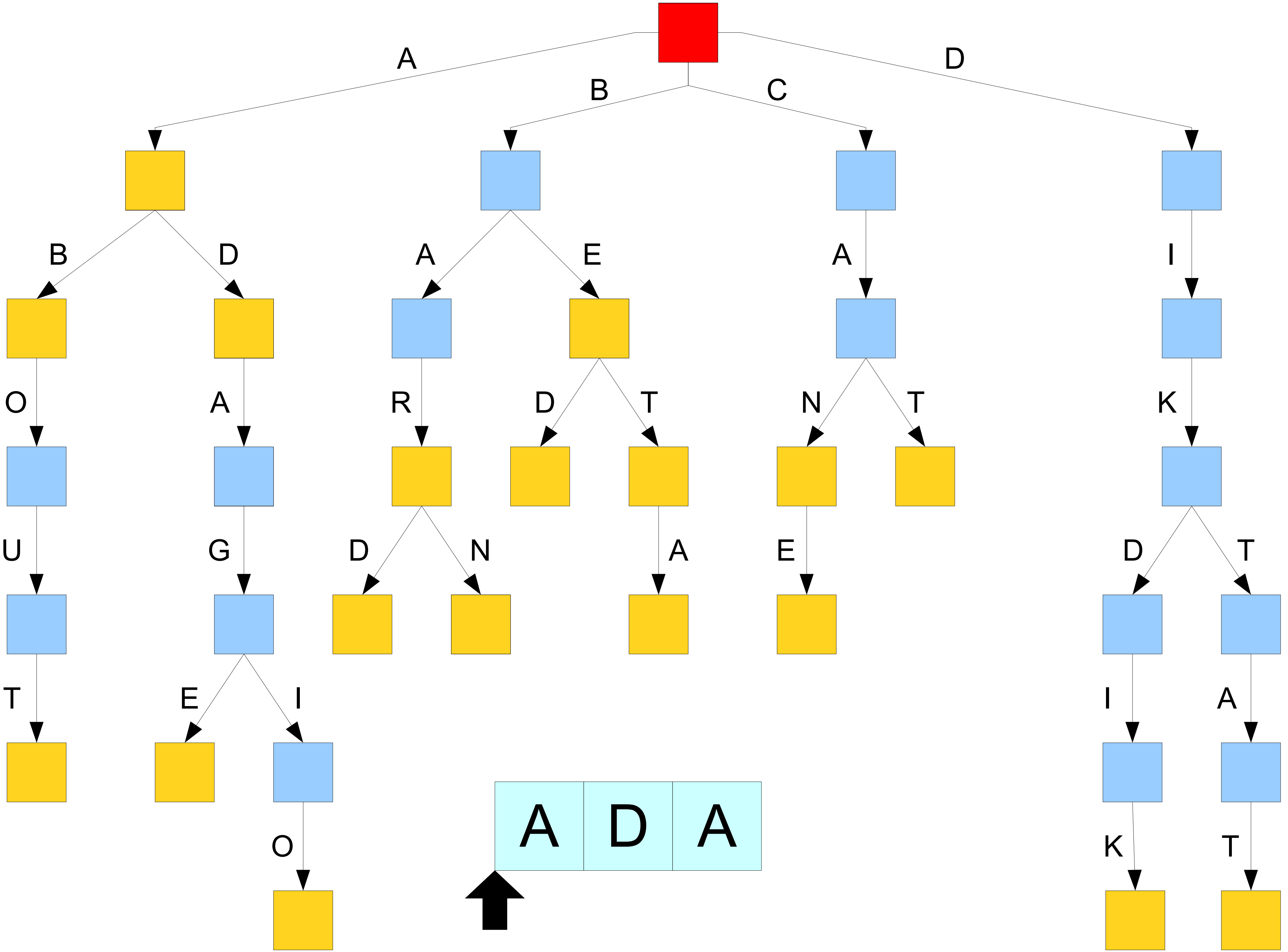

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A B O U T



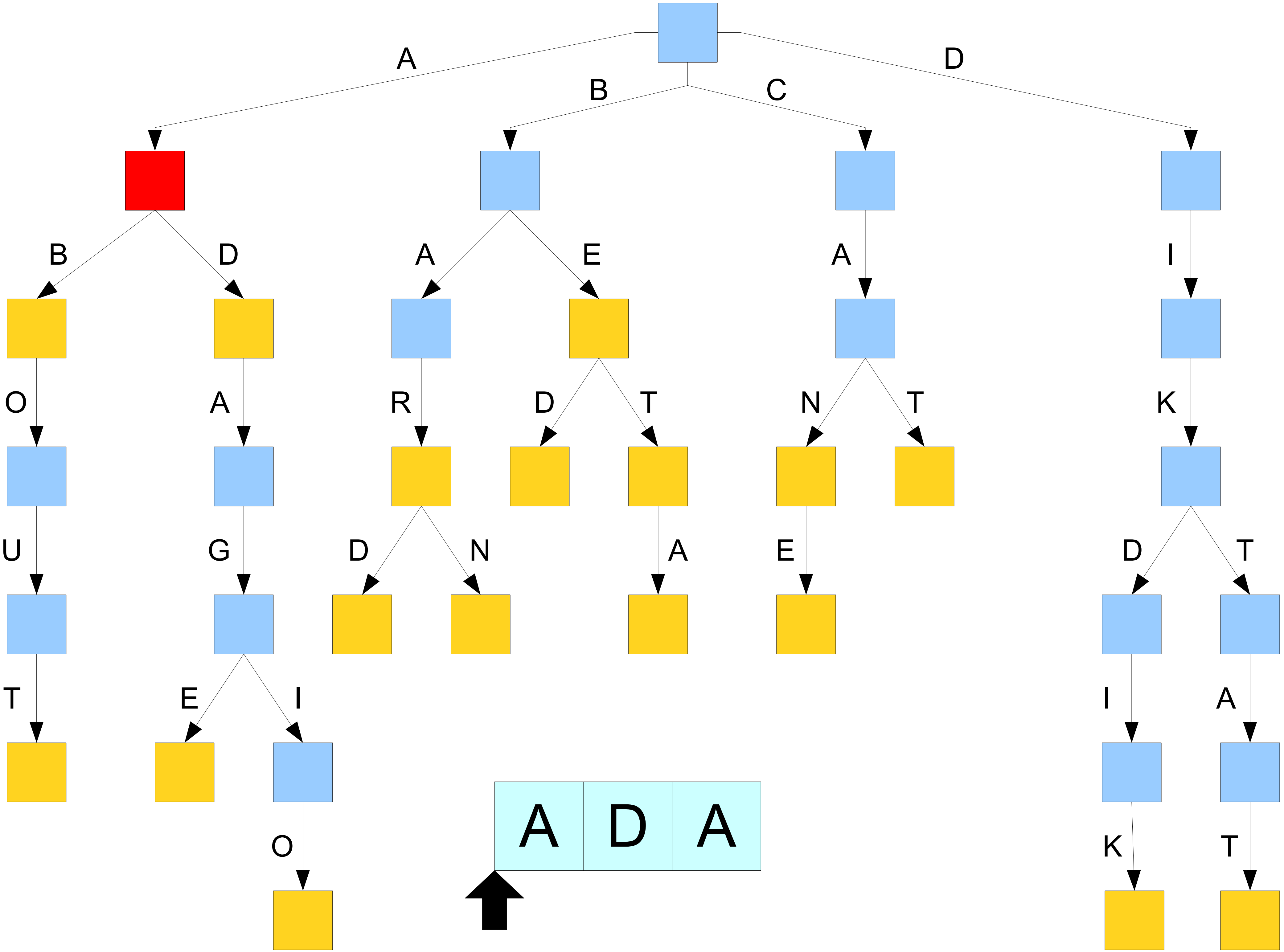
A
D
B
C
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A B O U T

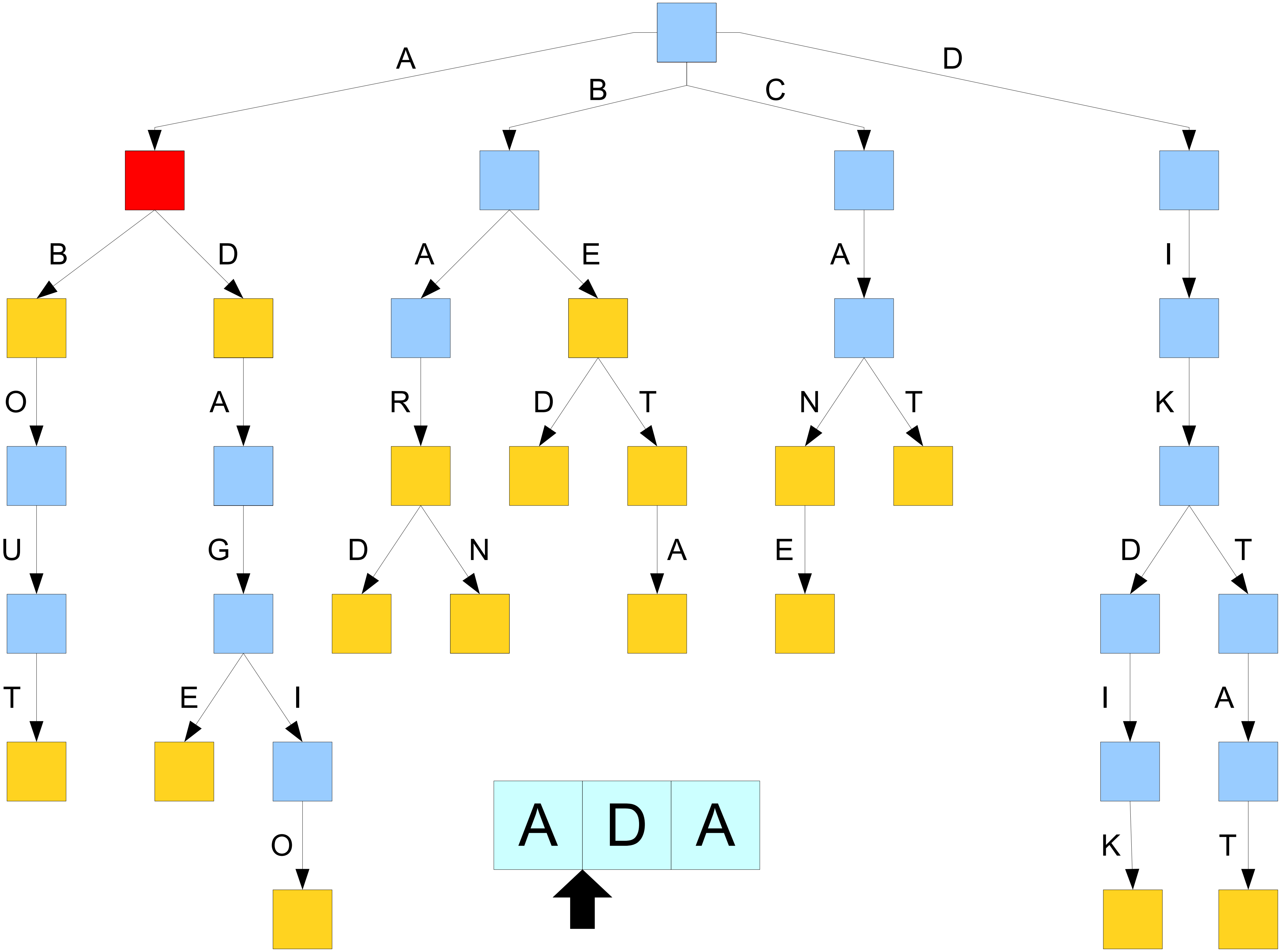

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A B O U T

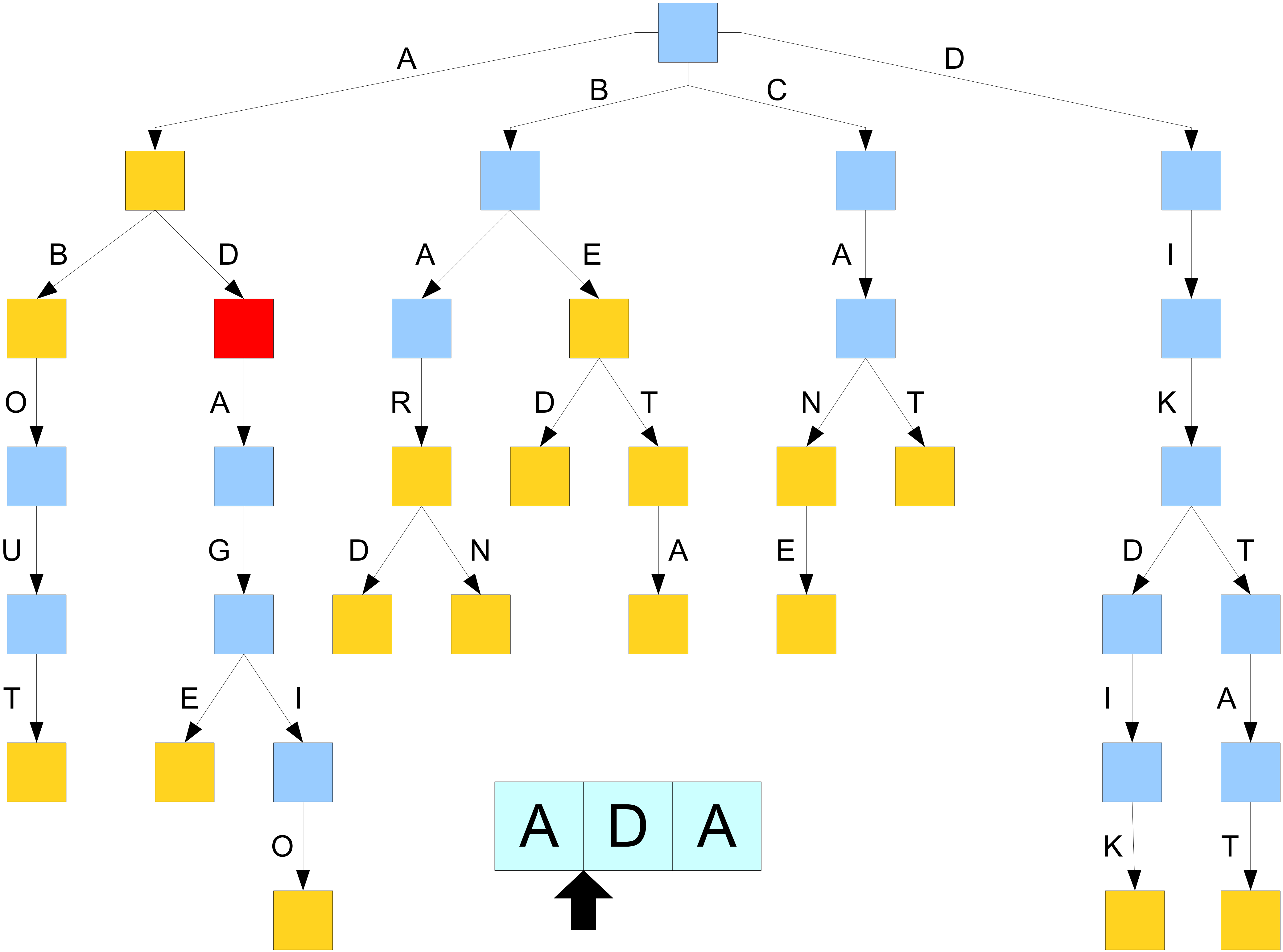

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A
D
A

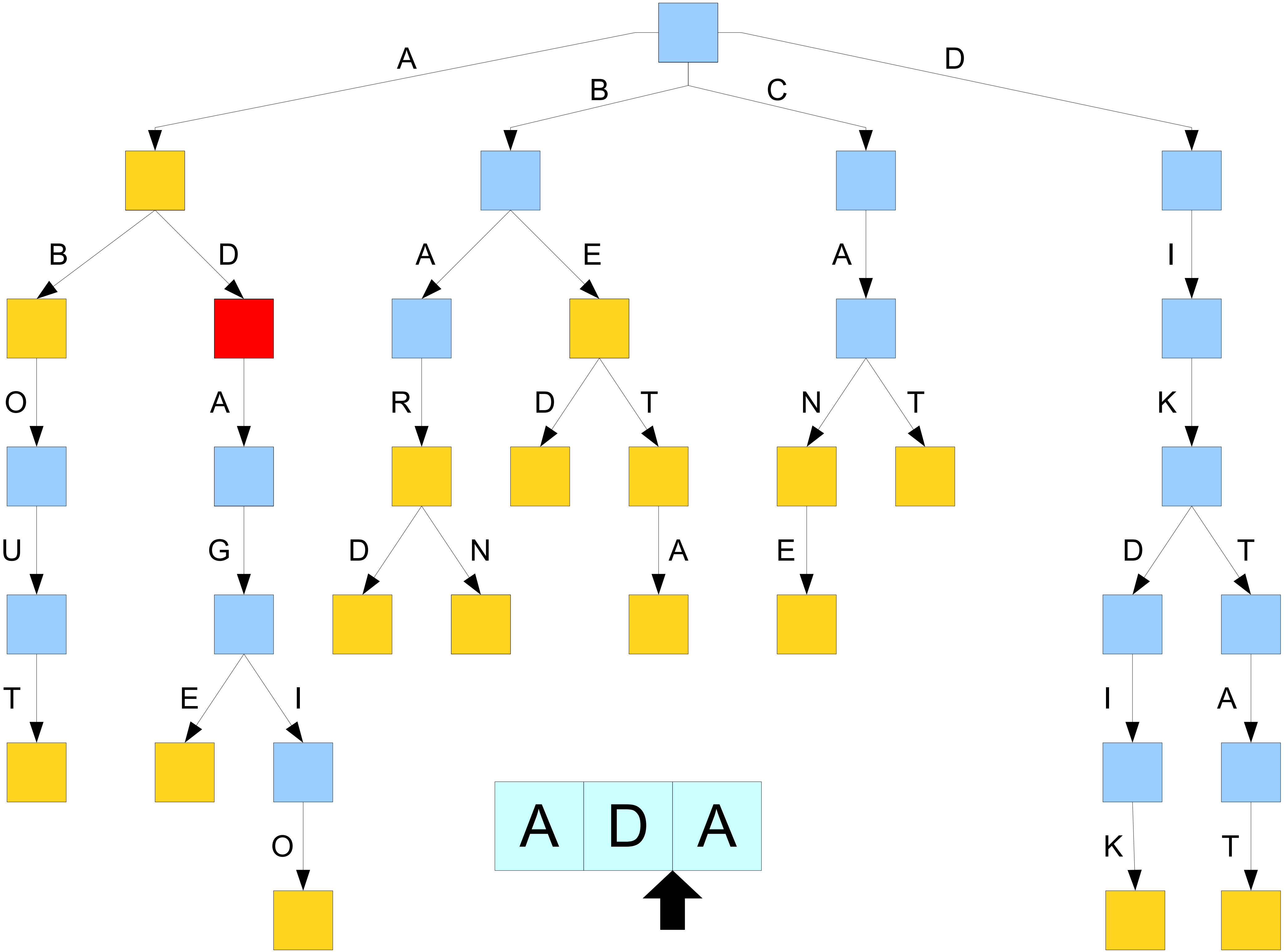


A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A
D
A

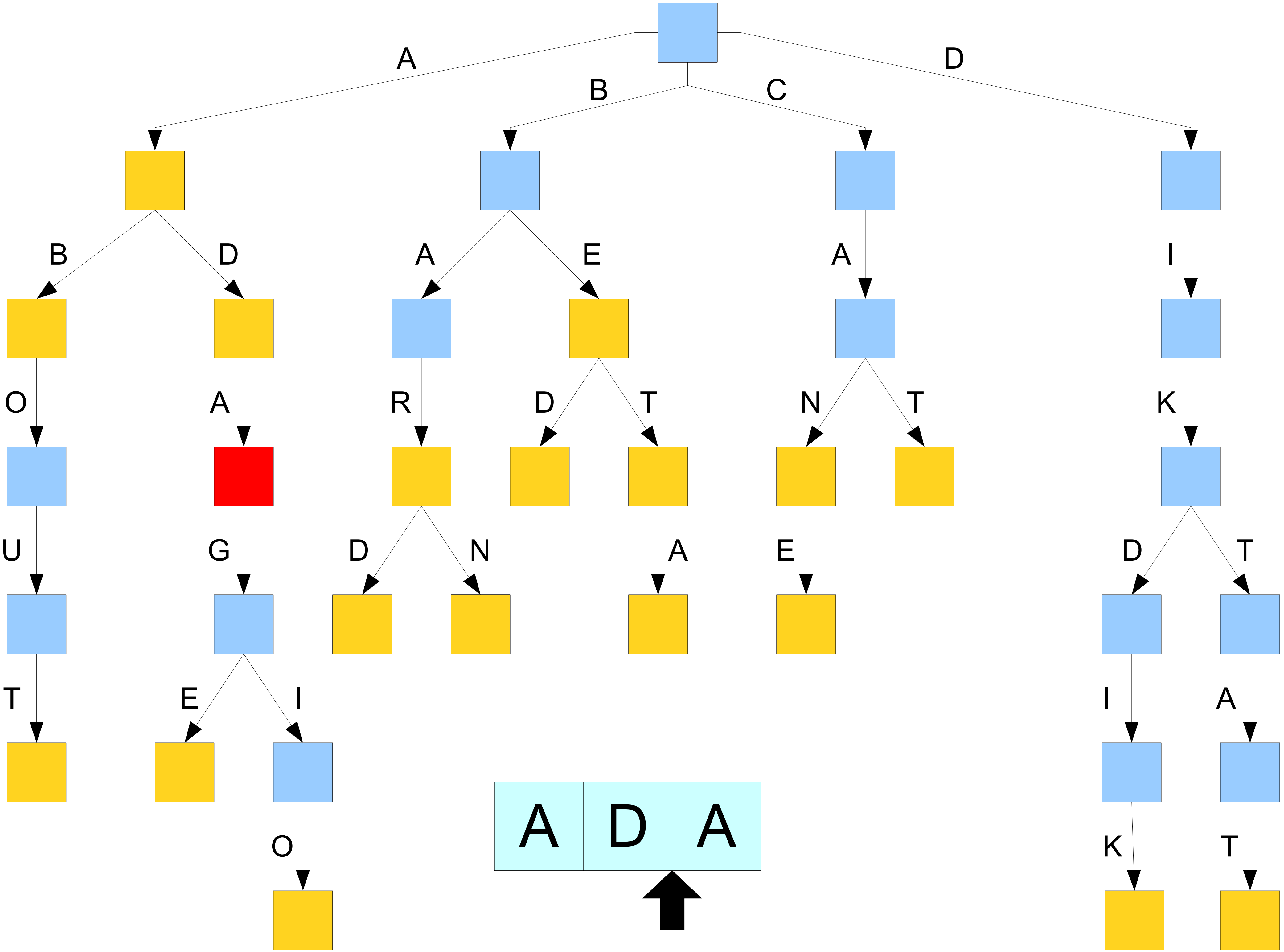

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A D A

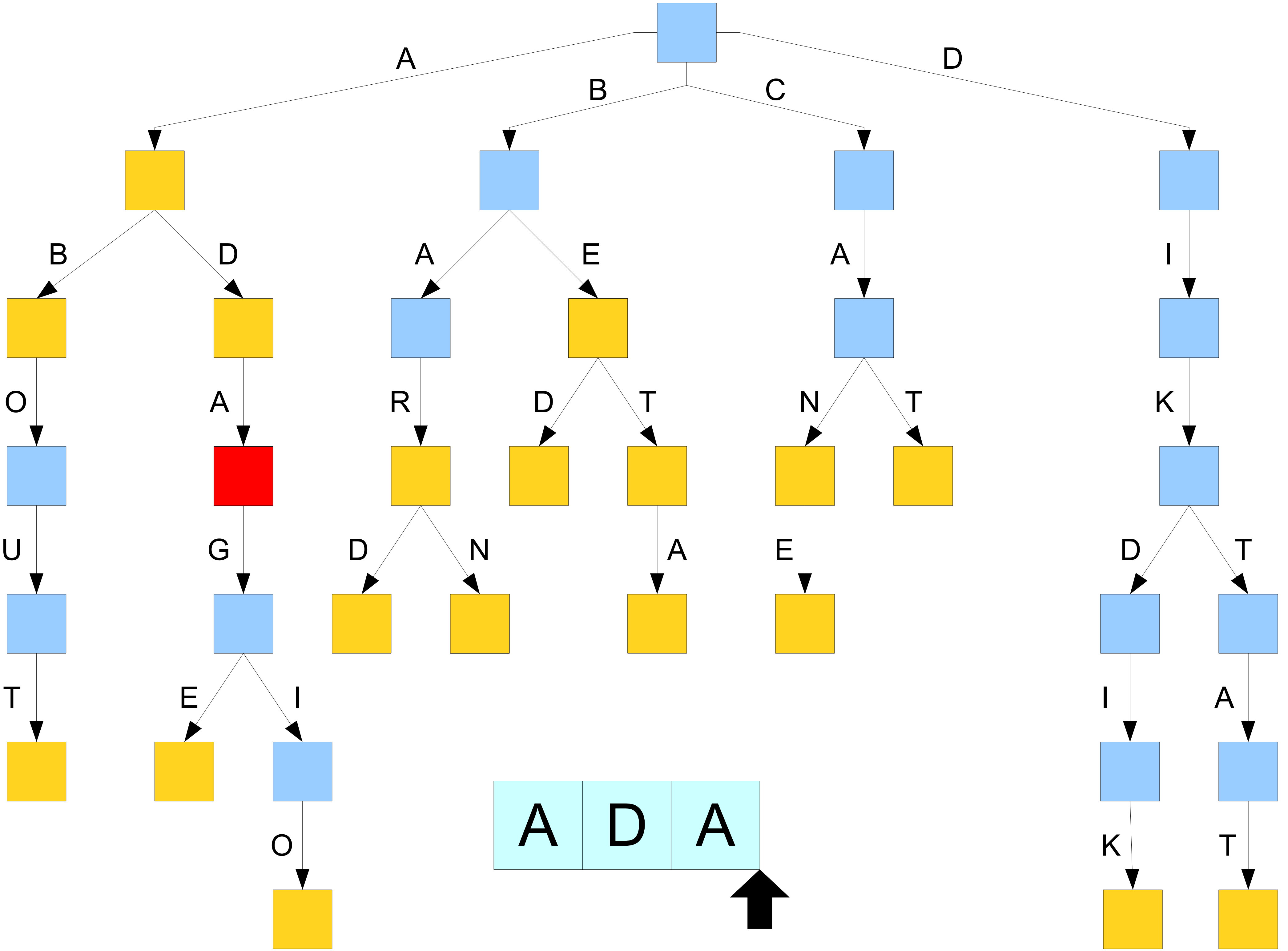

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A D A


A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A D A

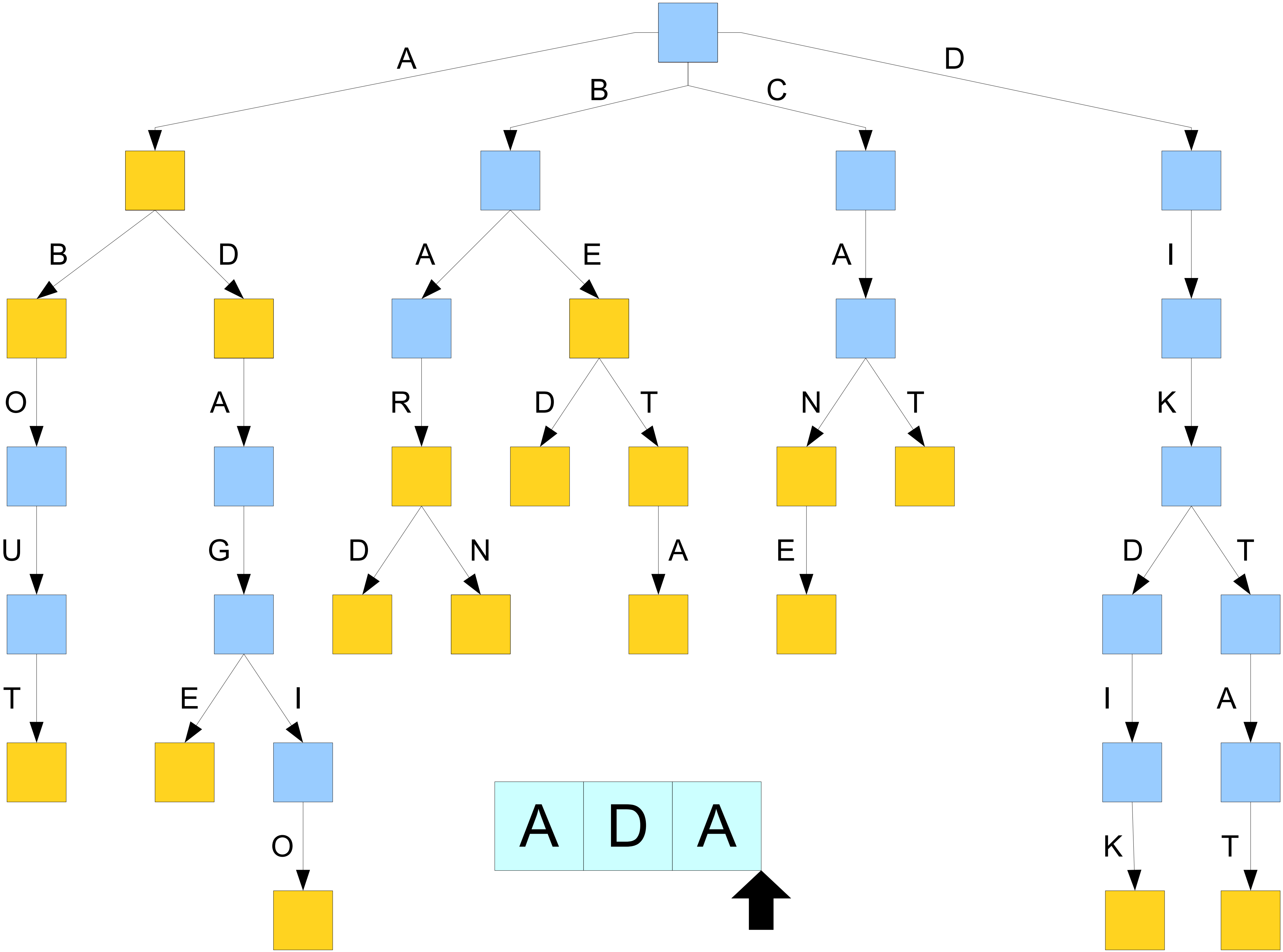


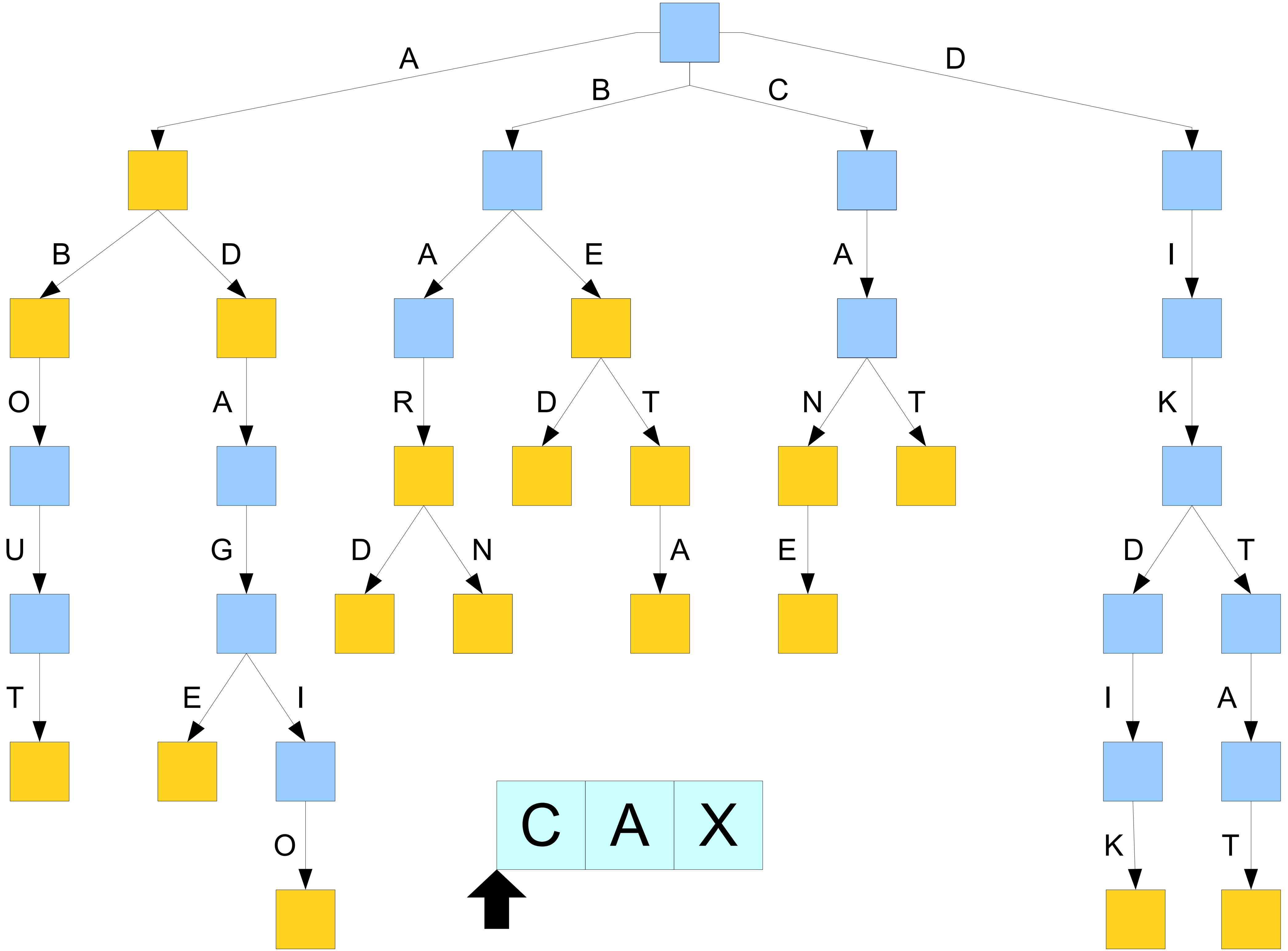
A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A
D
A


A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A D A

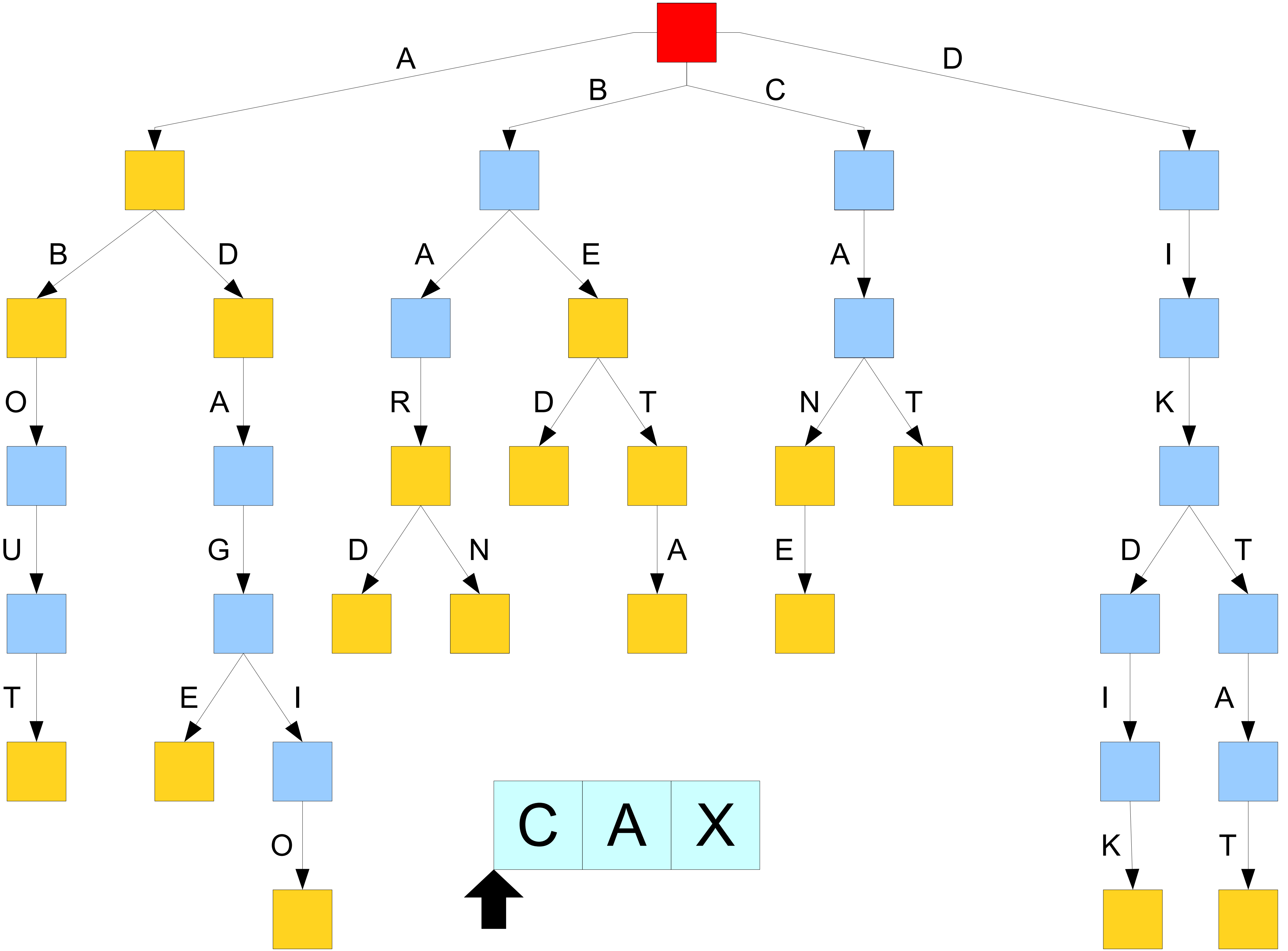


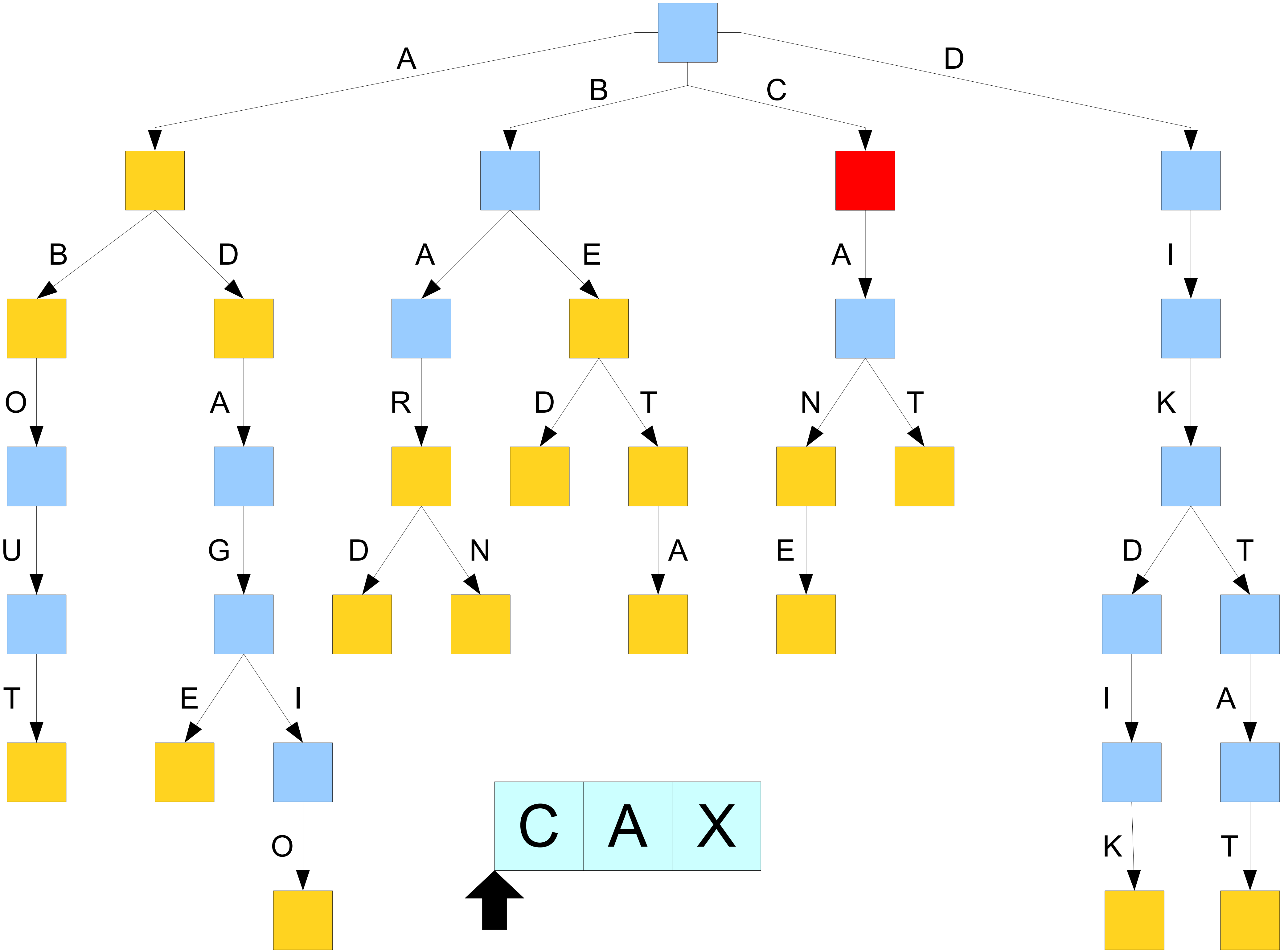
A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A D A

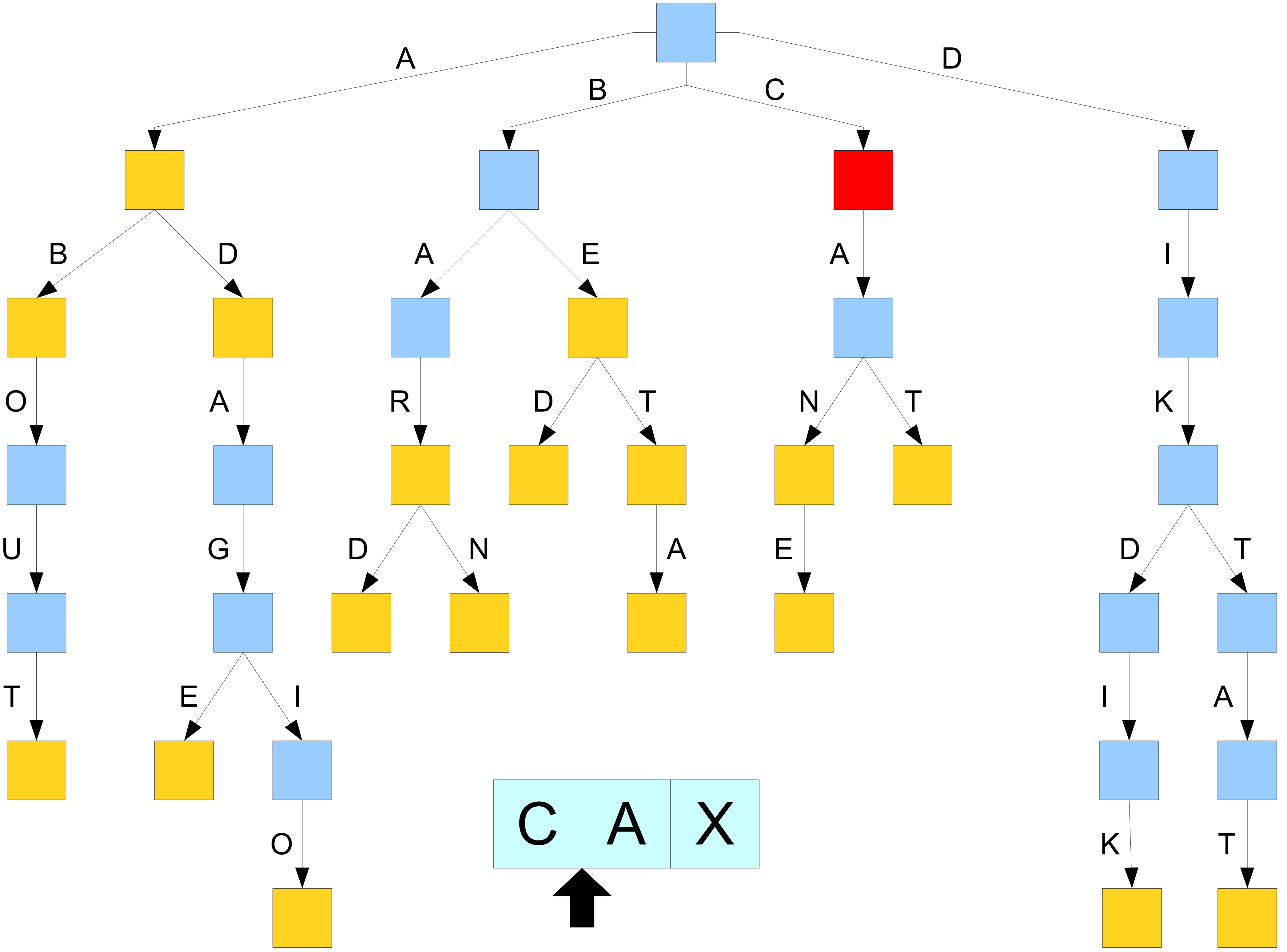

A
D
B
C
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A
D
A

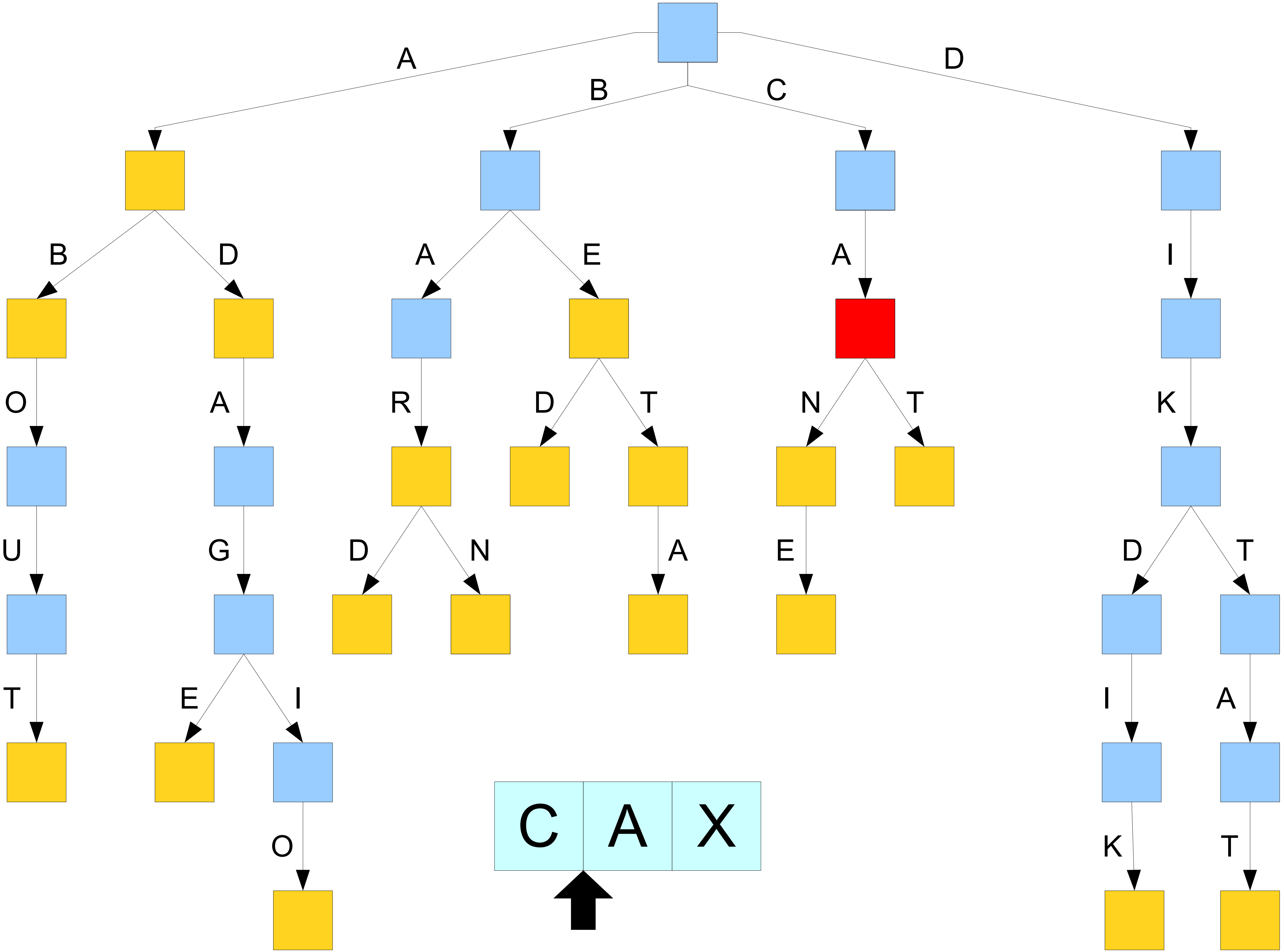


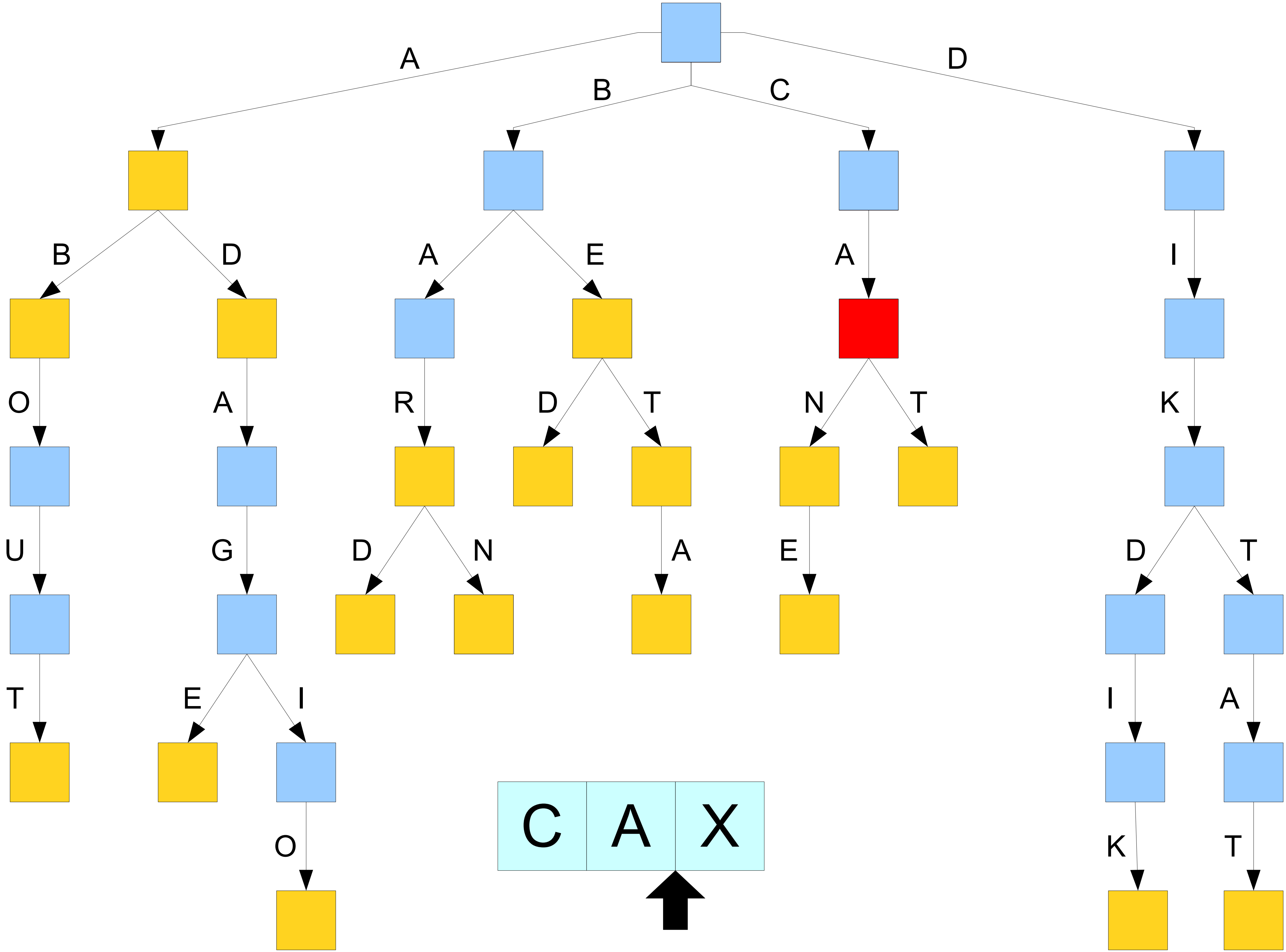
A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
C
A
X

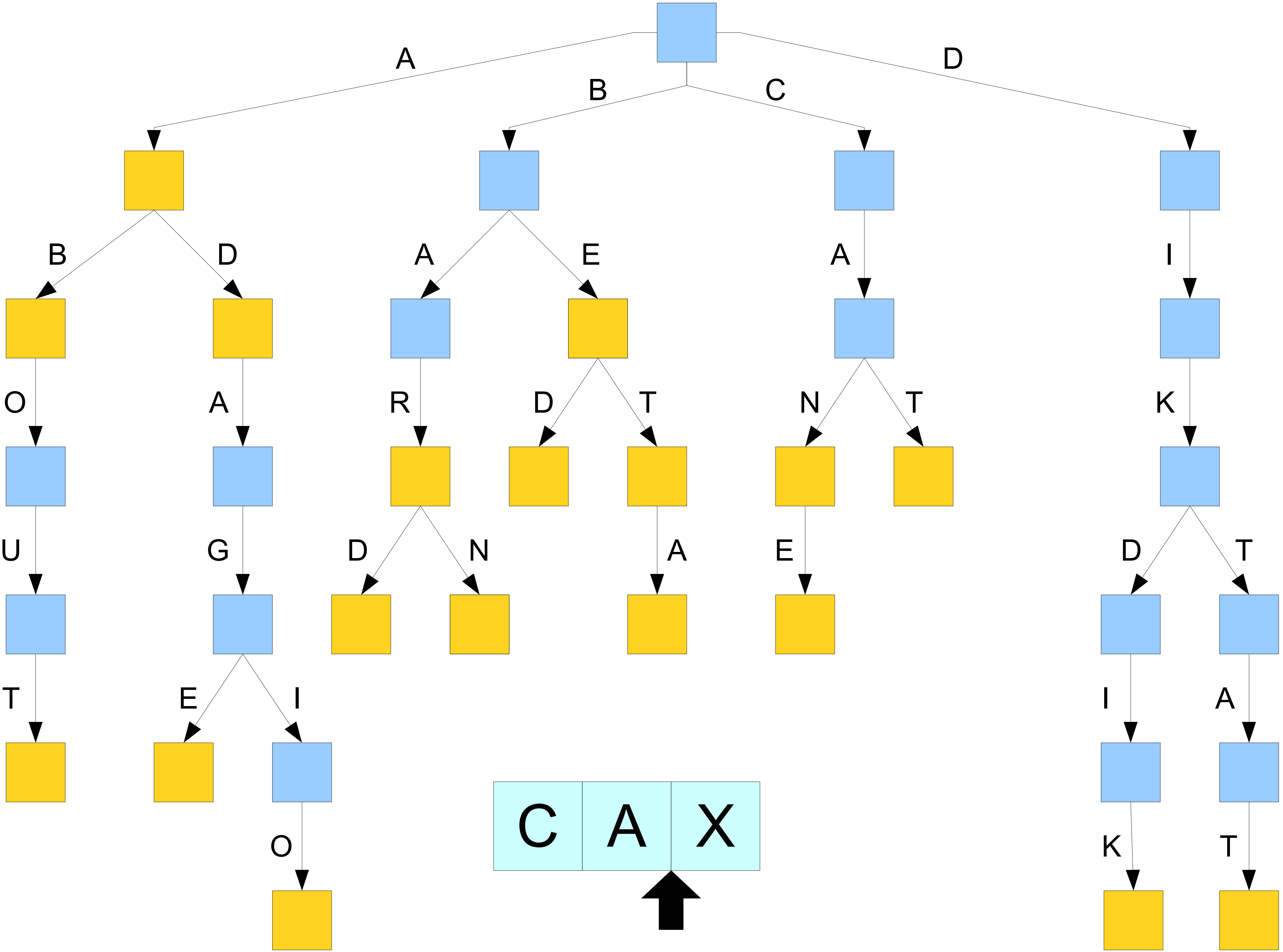

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
C A X

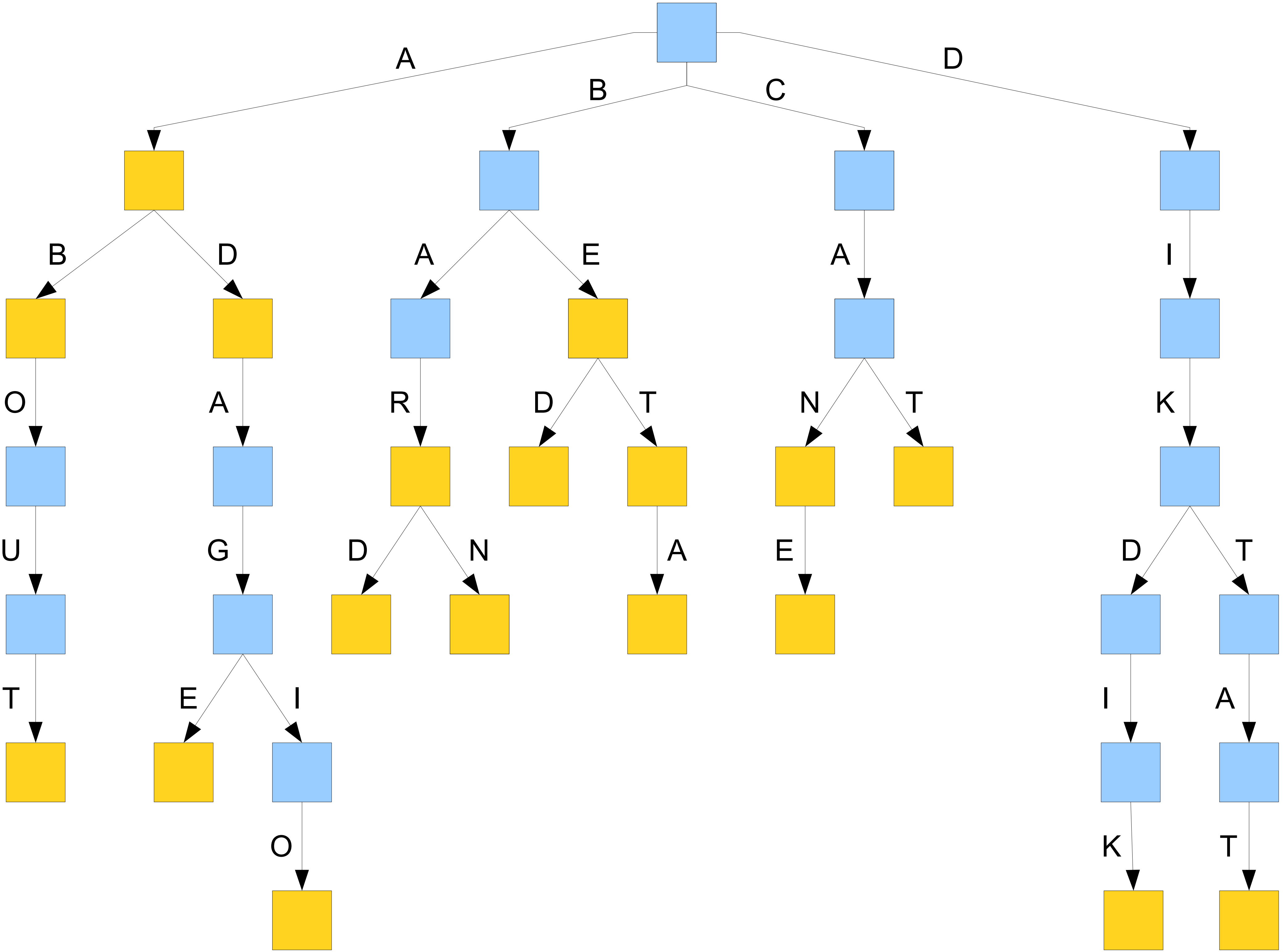

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
C
A
X
K
T


A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
C
A
X


A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
C
A
X

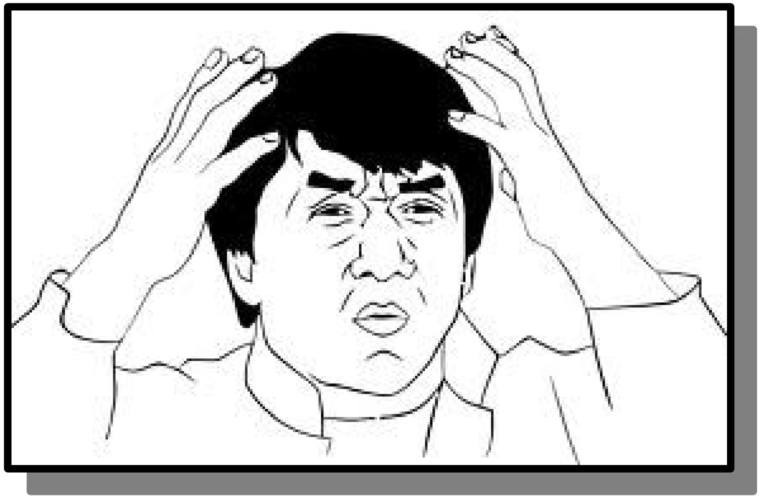

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
C
A
X


A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
C
A
X


A
D
B
C
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T

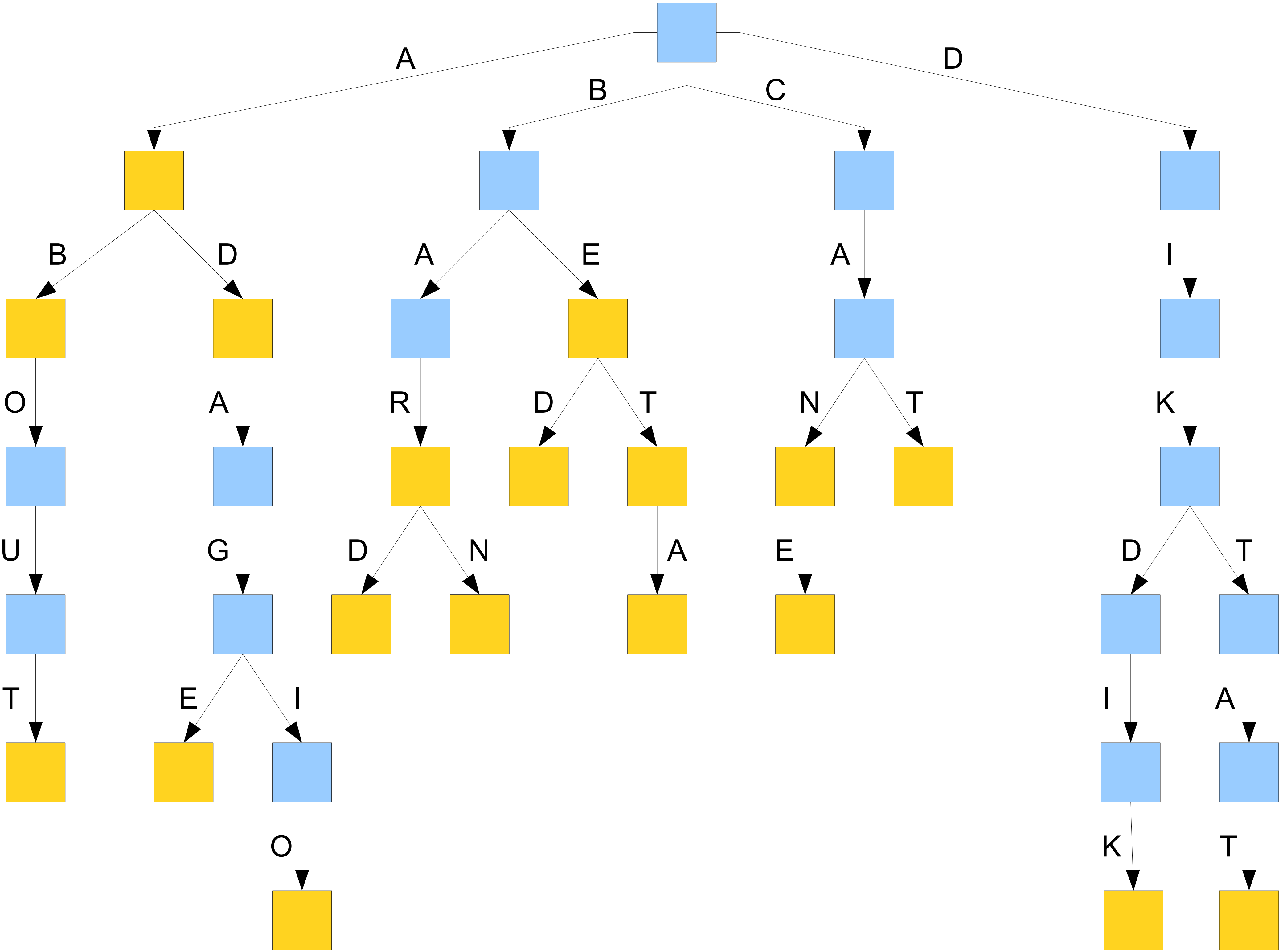
# Tries

- The data structure we have just seen is called a **trie**.
- Comes from the word re**trie**val.
- Pronounced “try,” not “tree.”

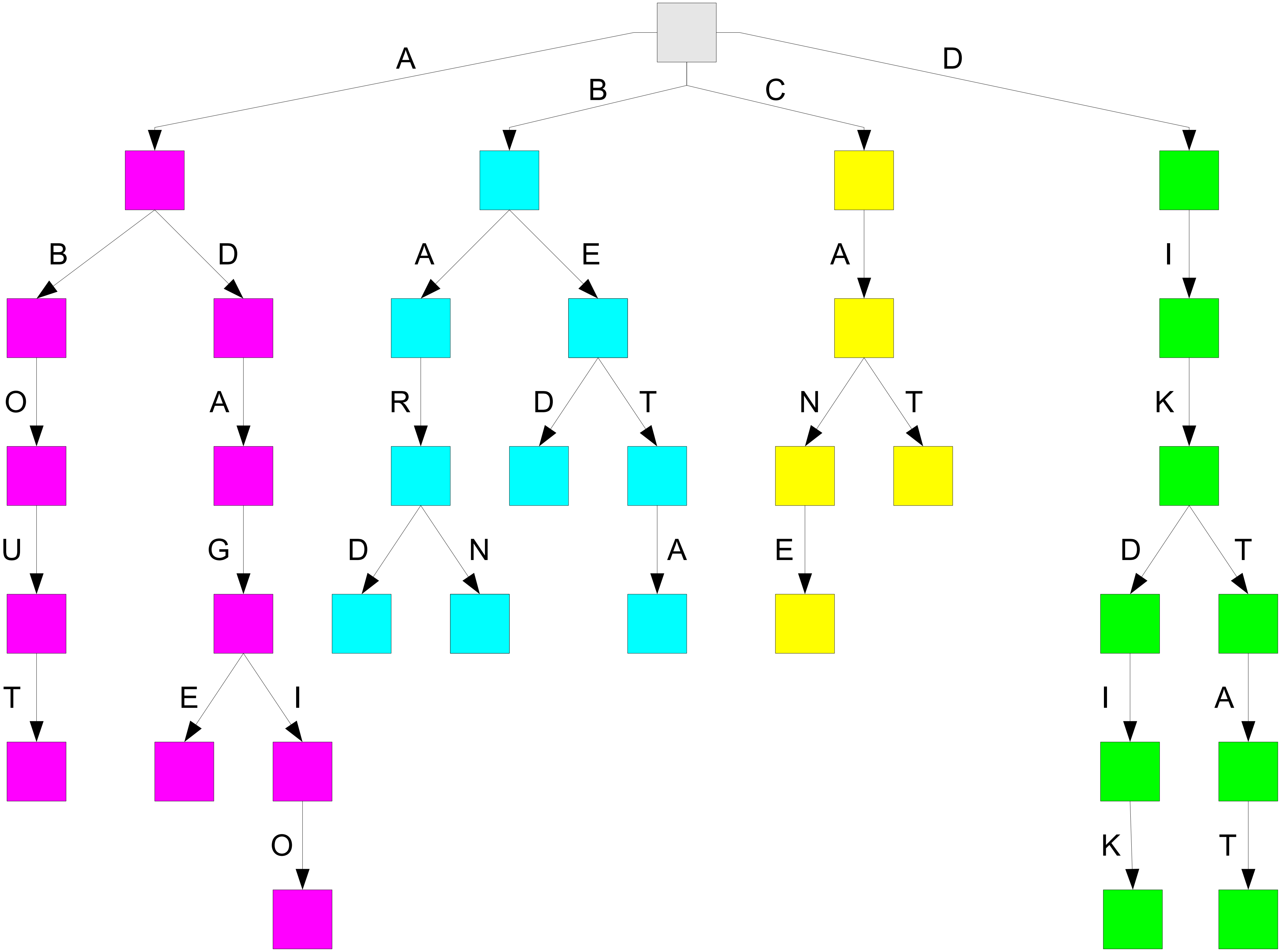


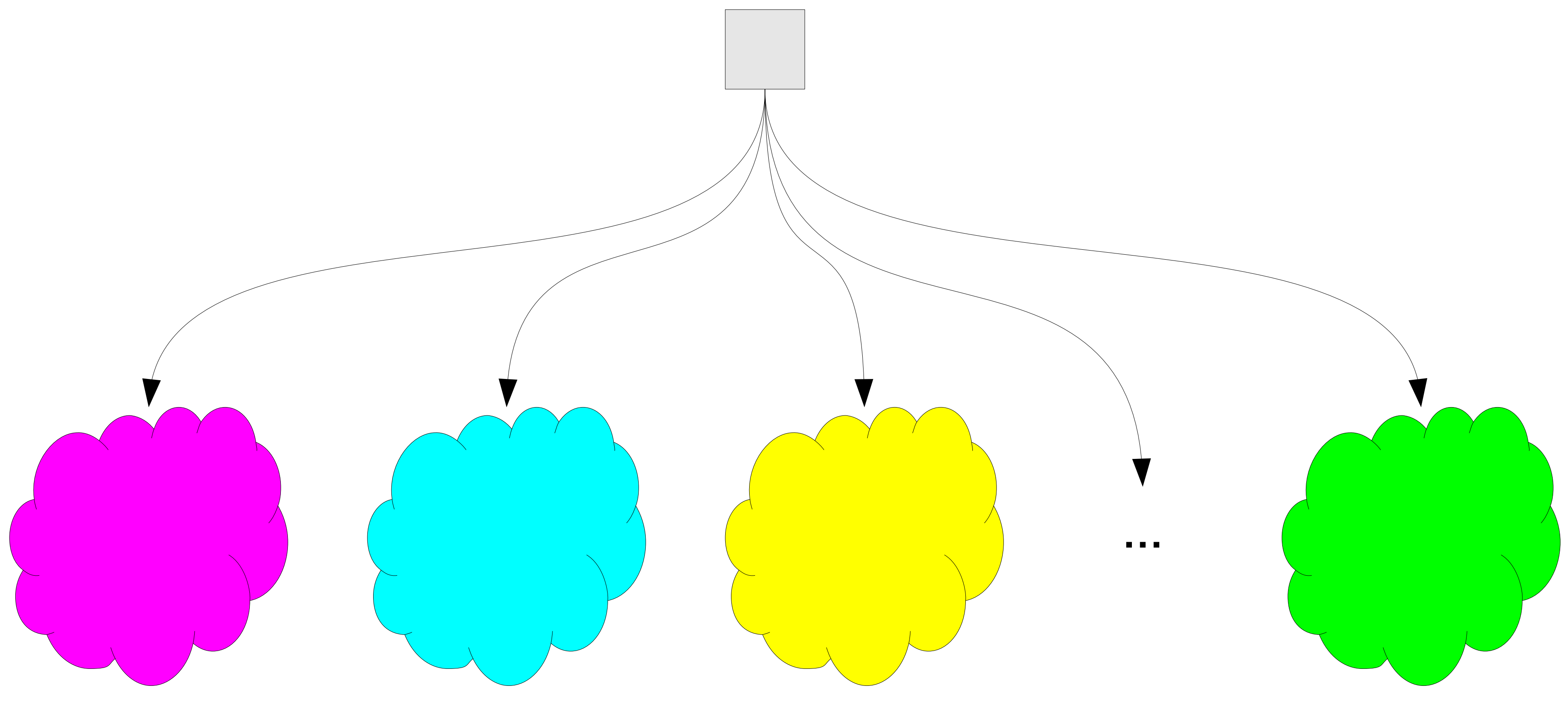
# Trie Nodes

- The pieces of the trie are called **nodes**.
- Each node stores two pieces of information:
    - Whether, at this point in the trie, you have arrived at a word, and
    - Pointers to child nodes in the trie.
- The node at the top of the trie is called the **root node**.

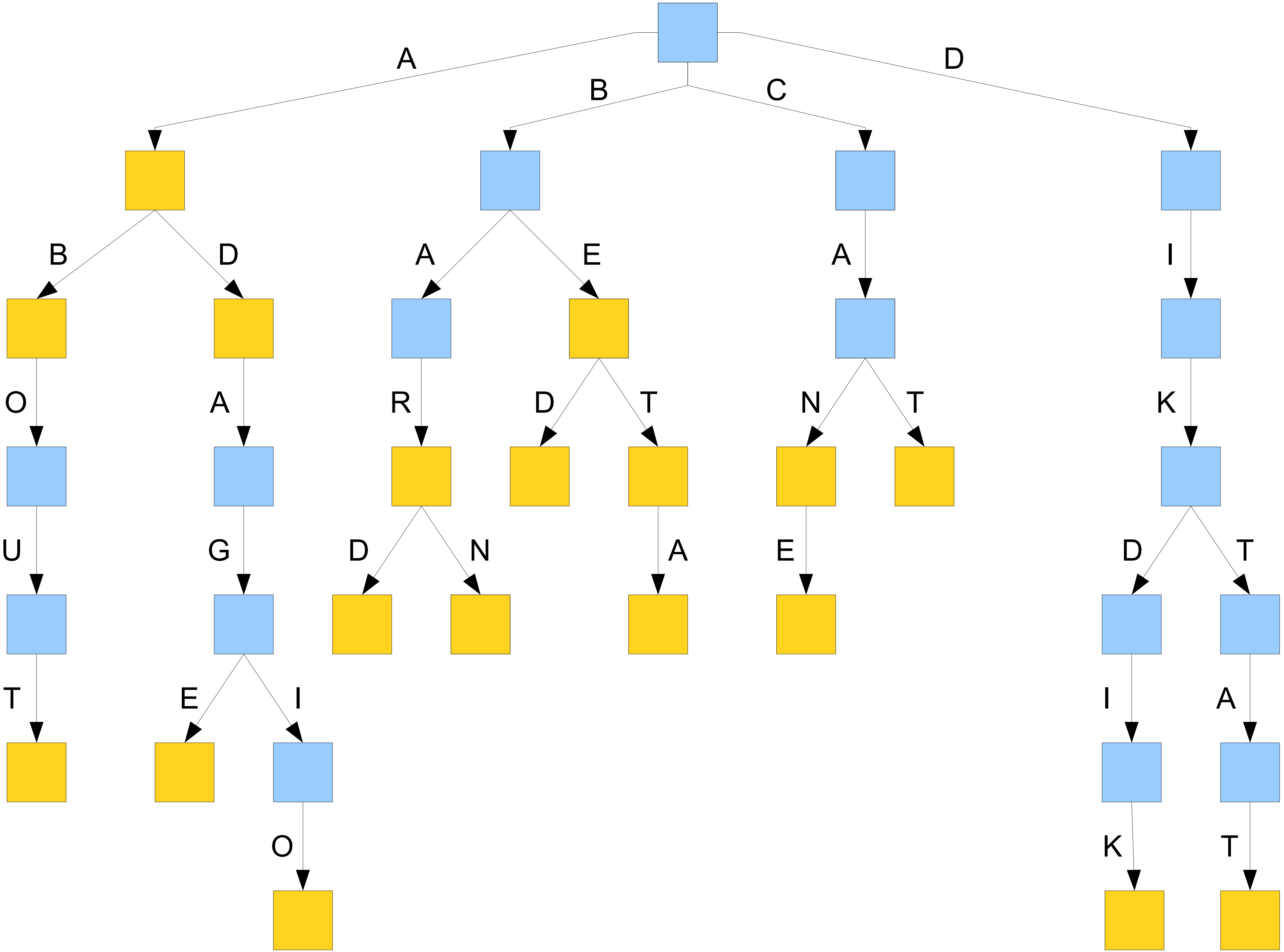


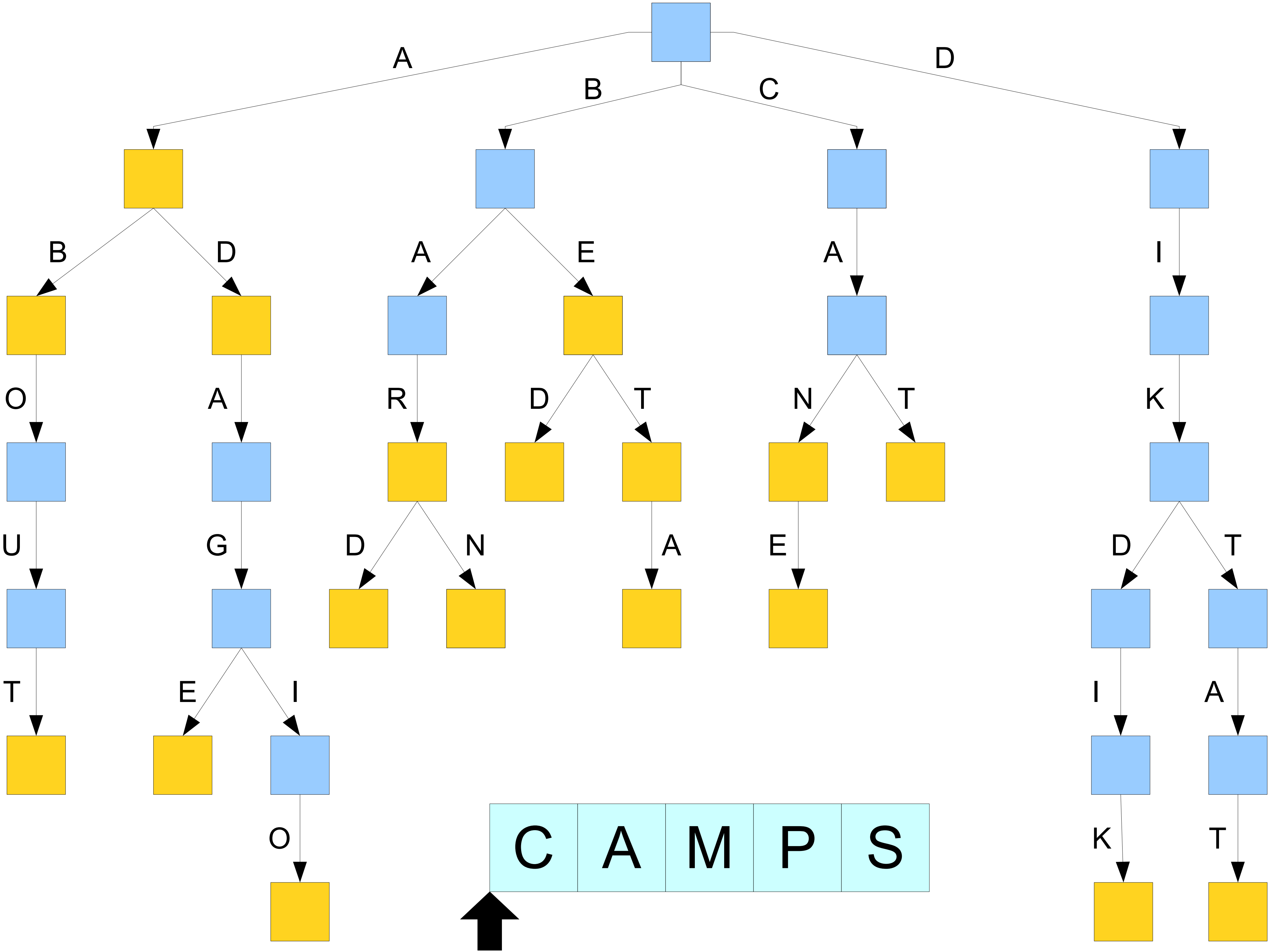
A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T

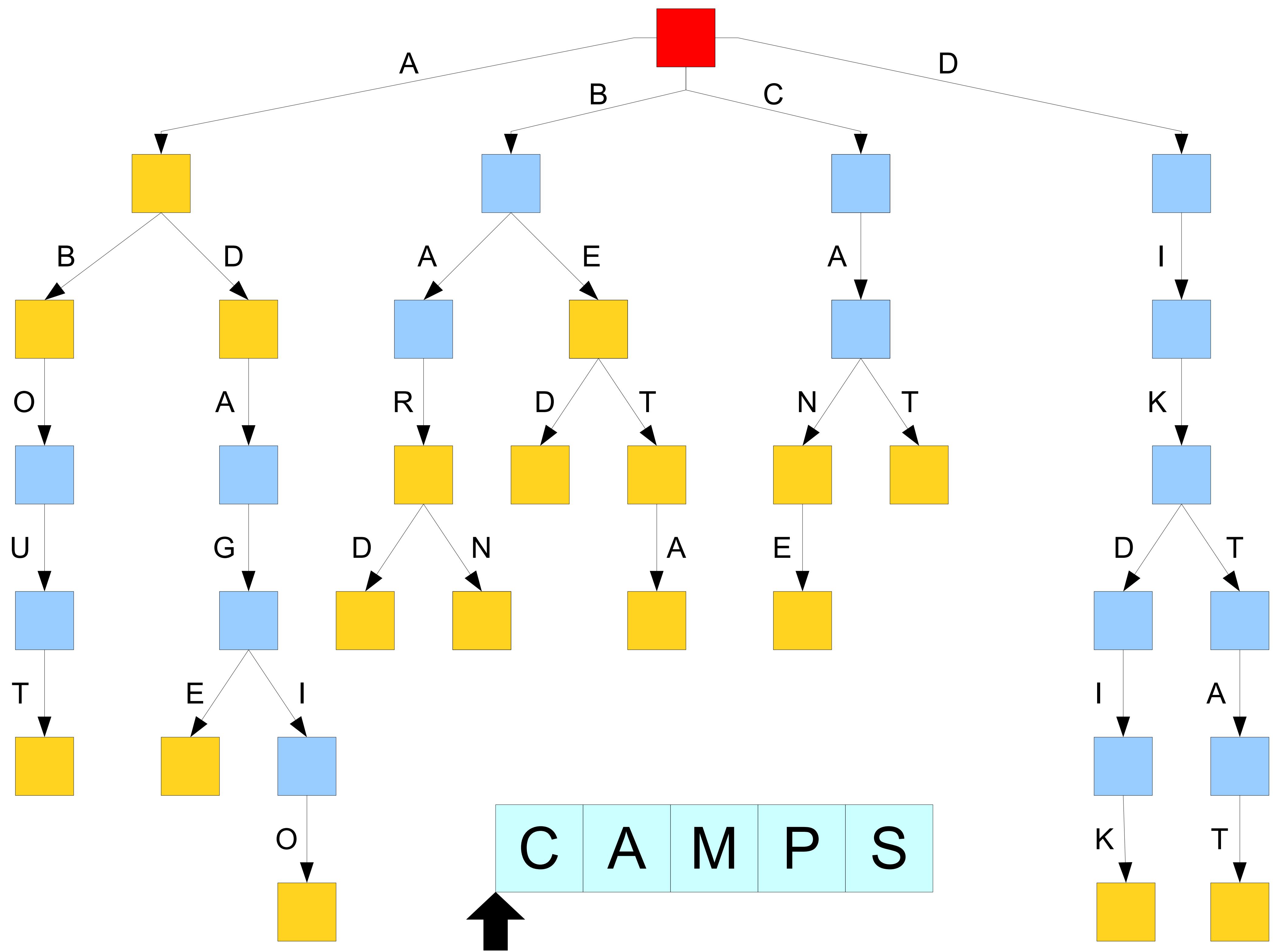

A
D
B
C
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T

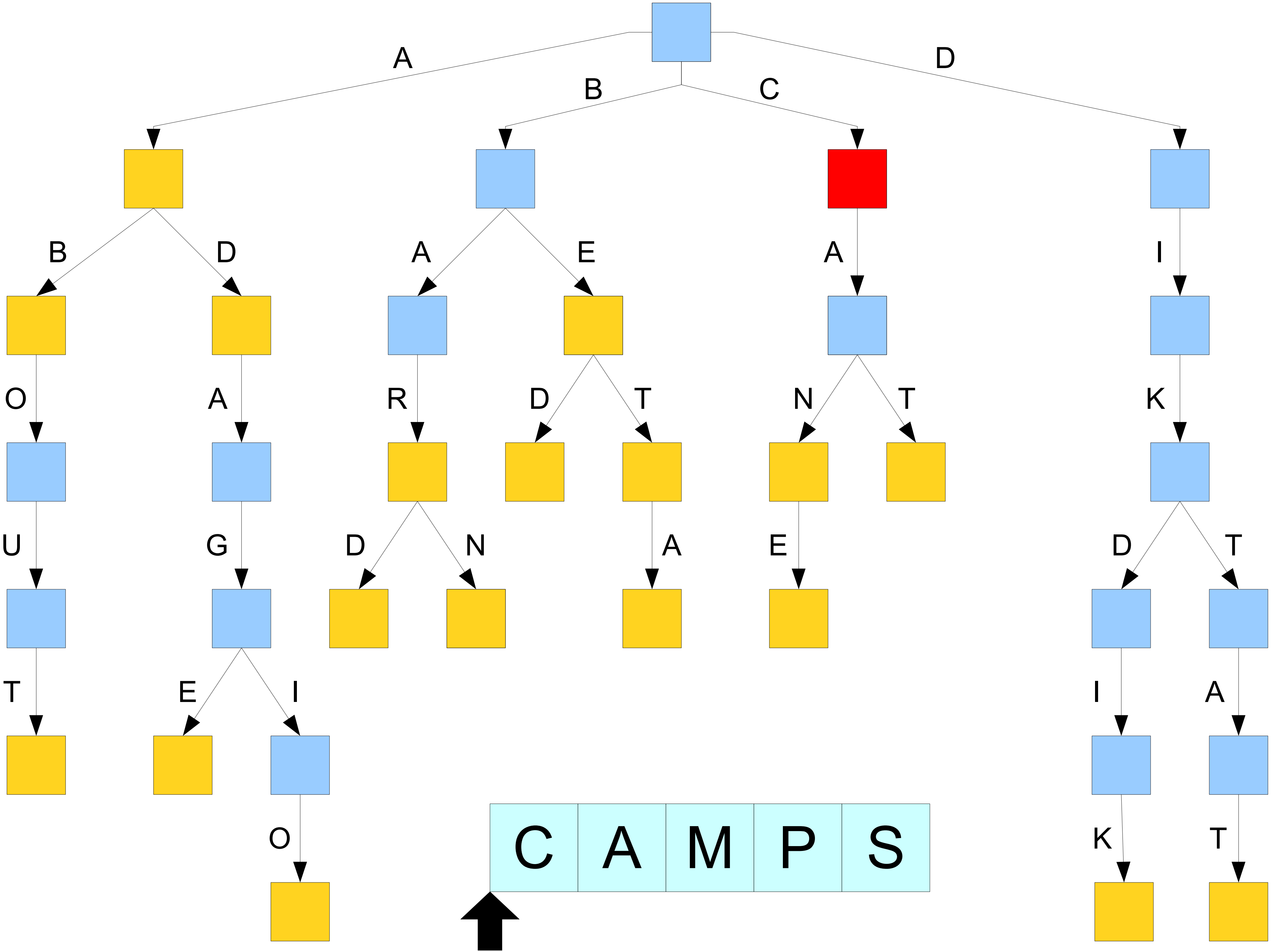


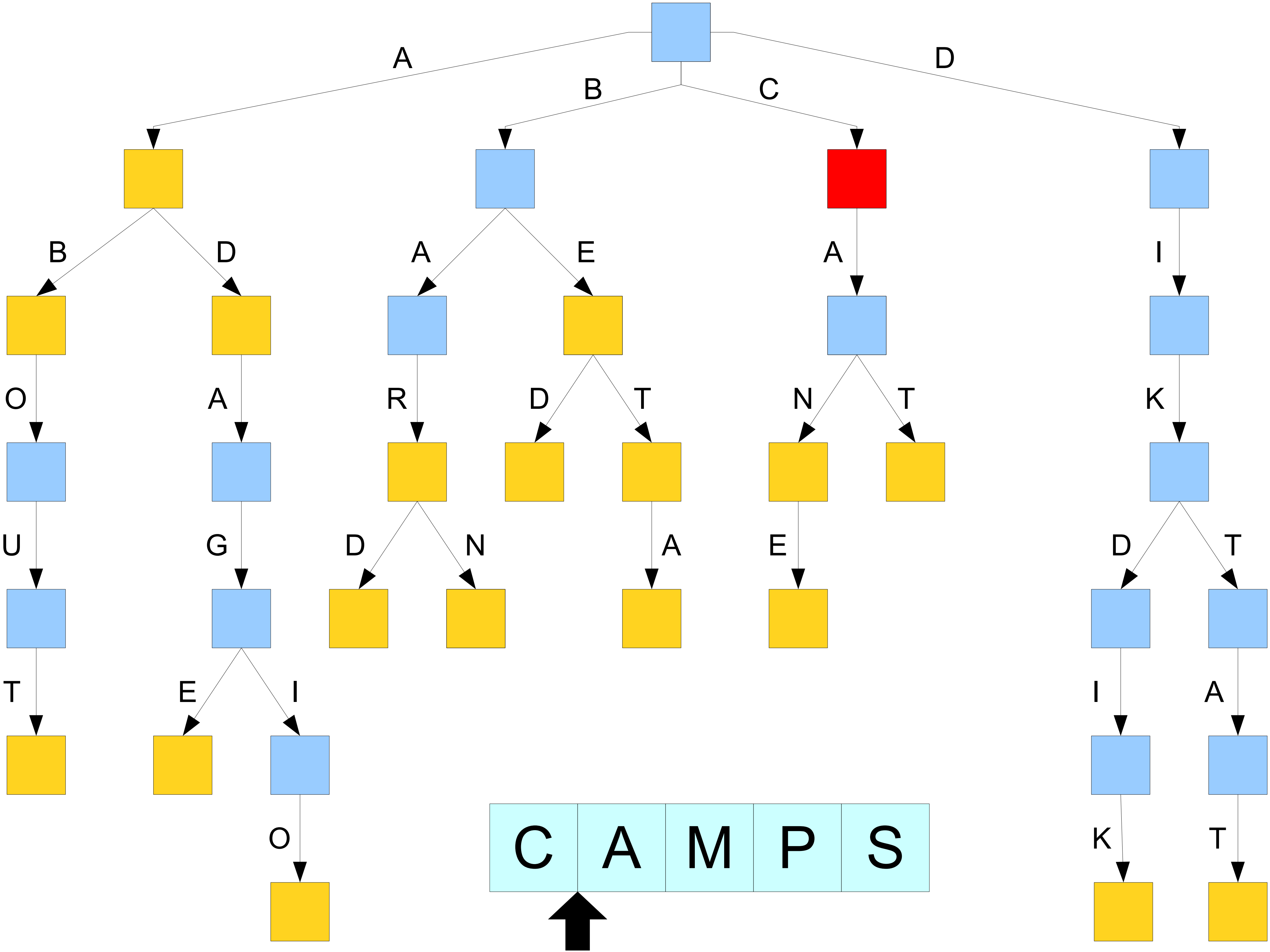
Let's trie coding up **`Lexicon`**!
(Constructor,
Destructor
Variables,
**`size`**
**`contains`**,
**`containsPrefix`**)
(OurLexicon.cpp/h)

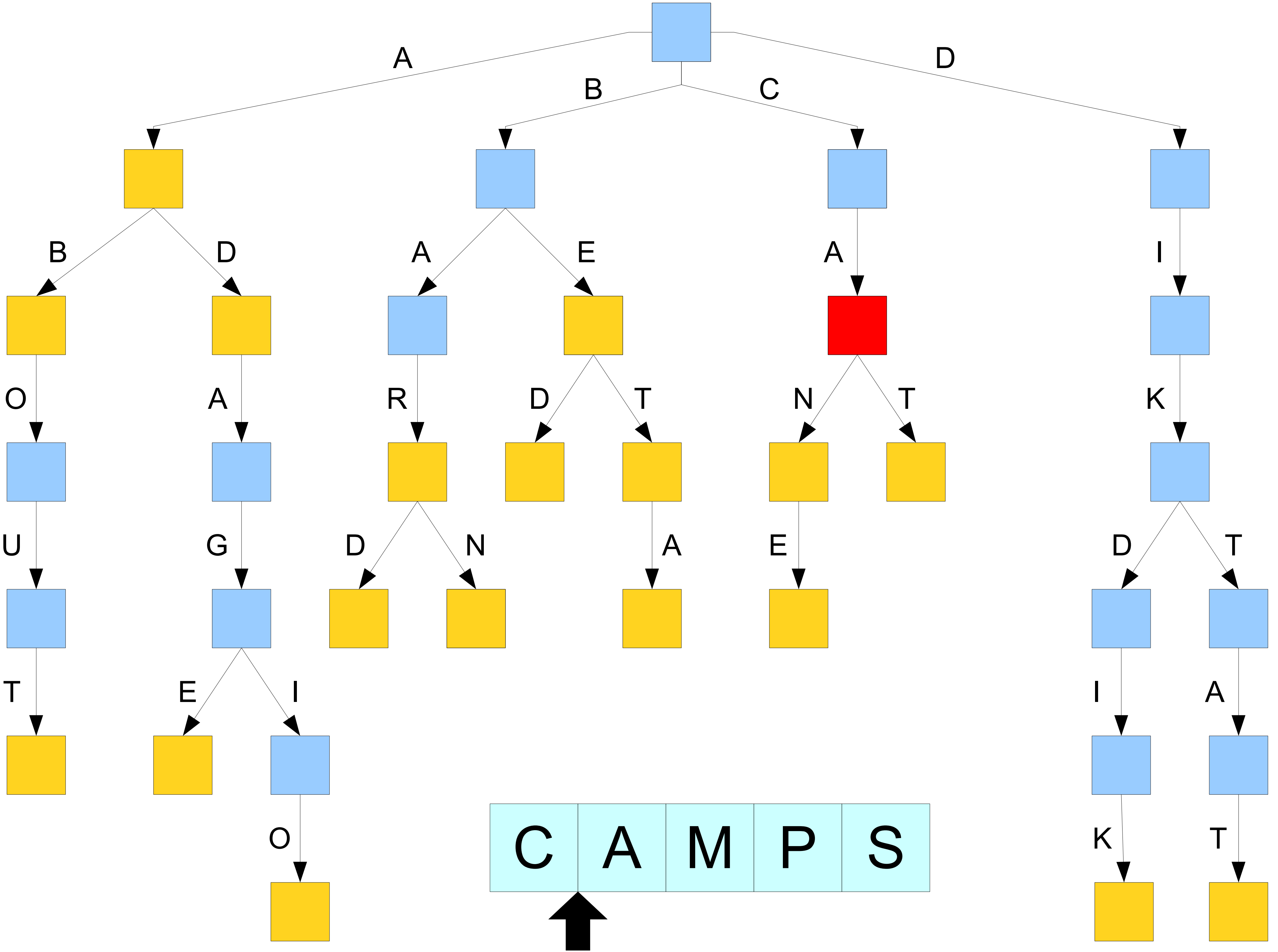

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T

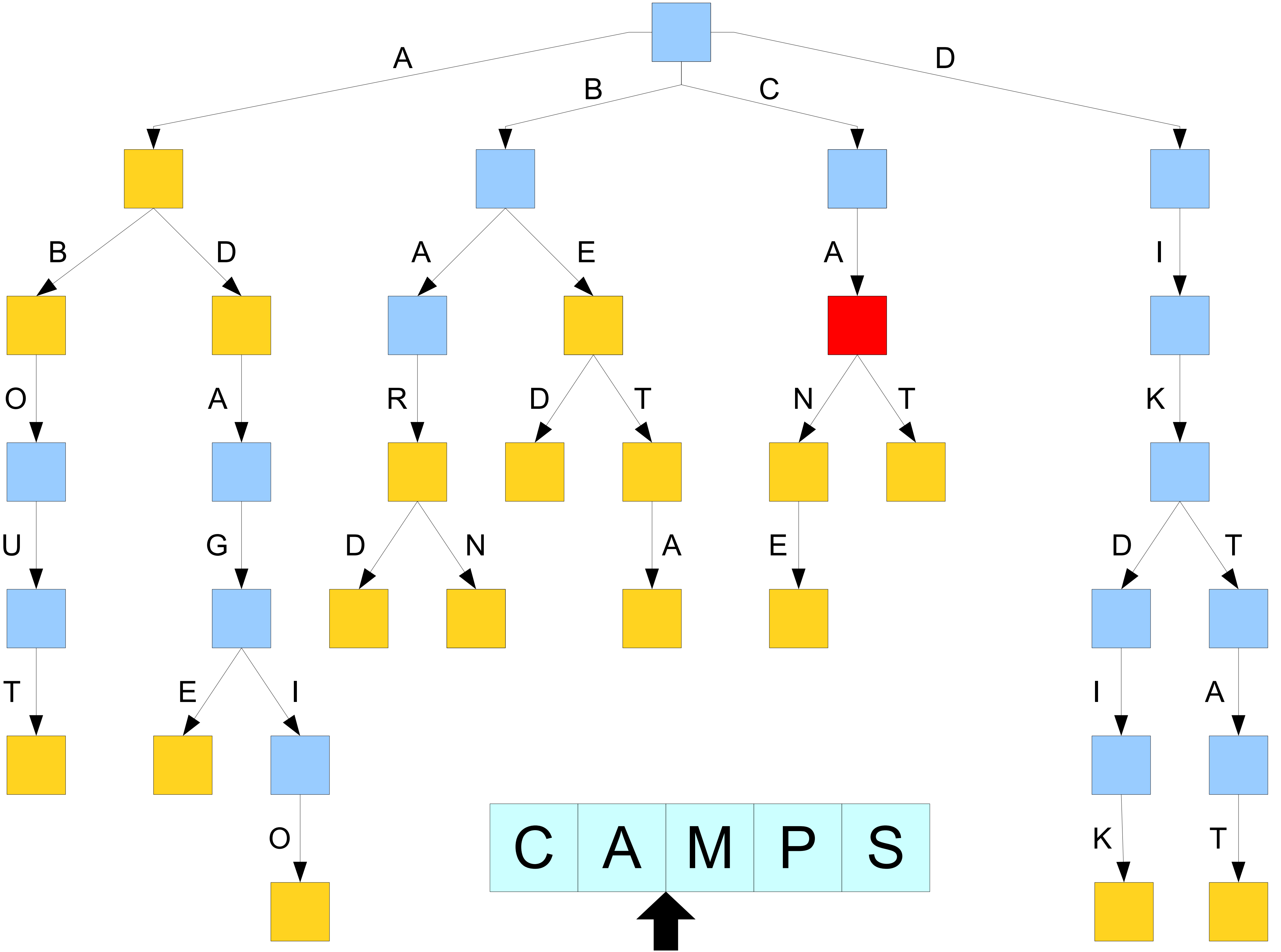

A
D
B
C
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
C A M P S

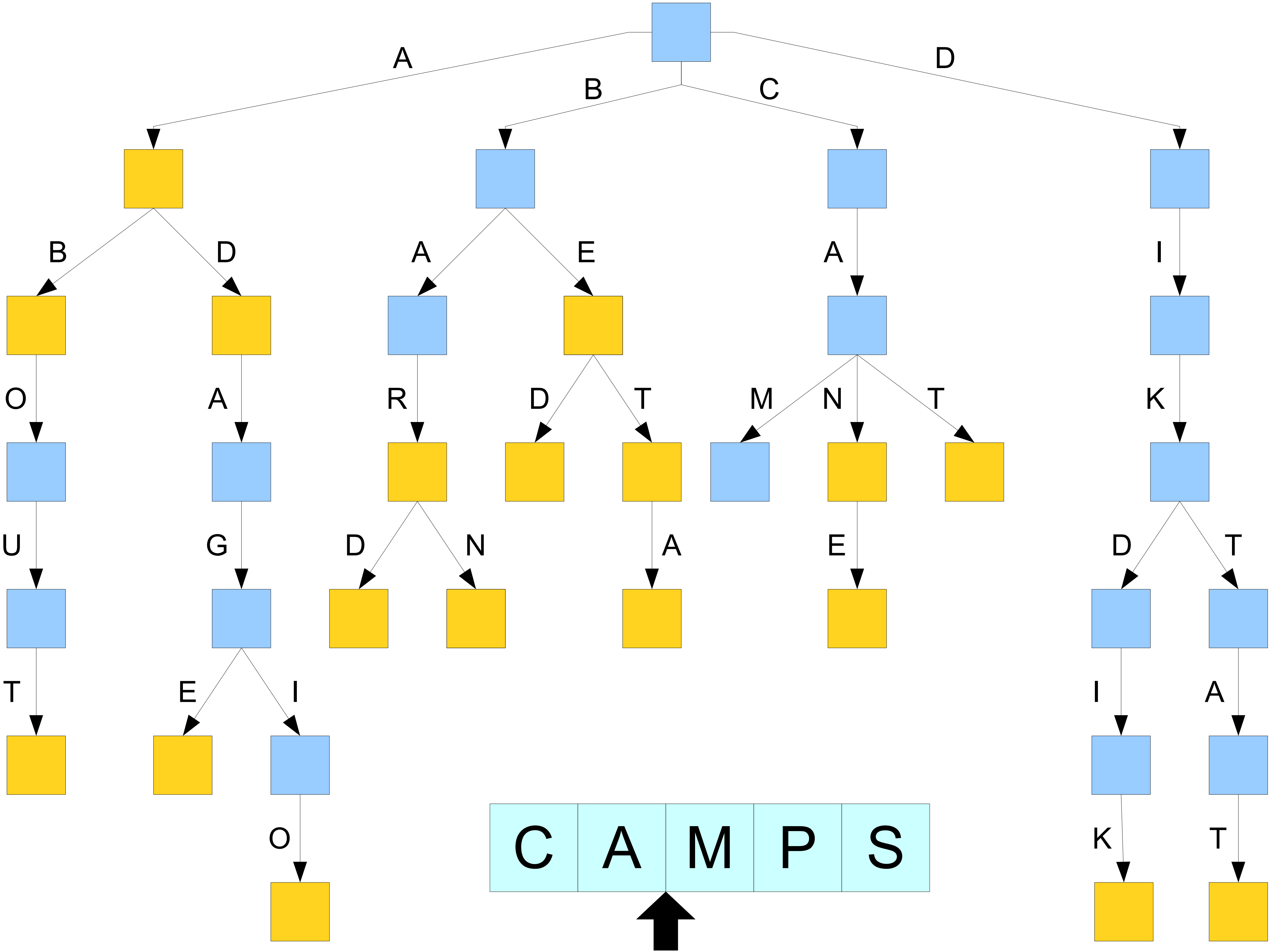

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
C A M P S

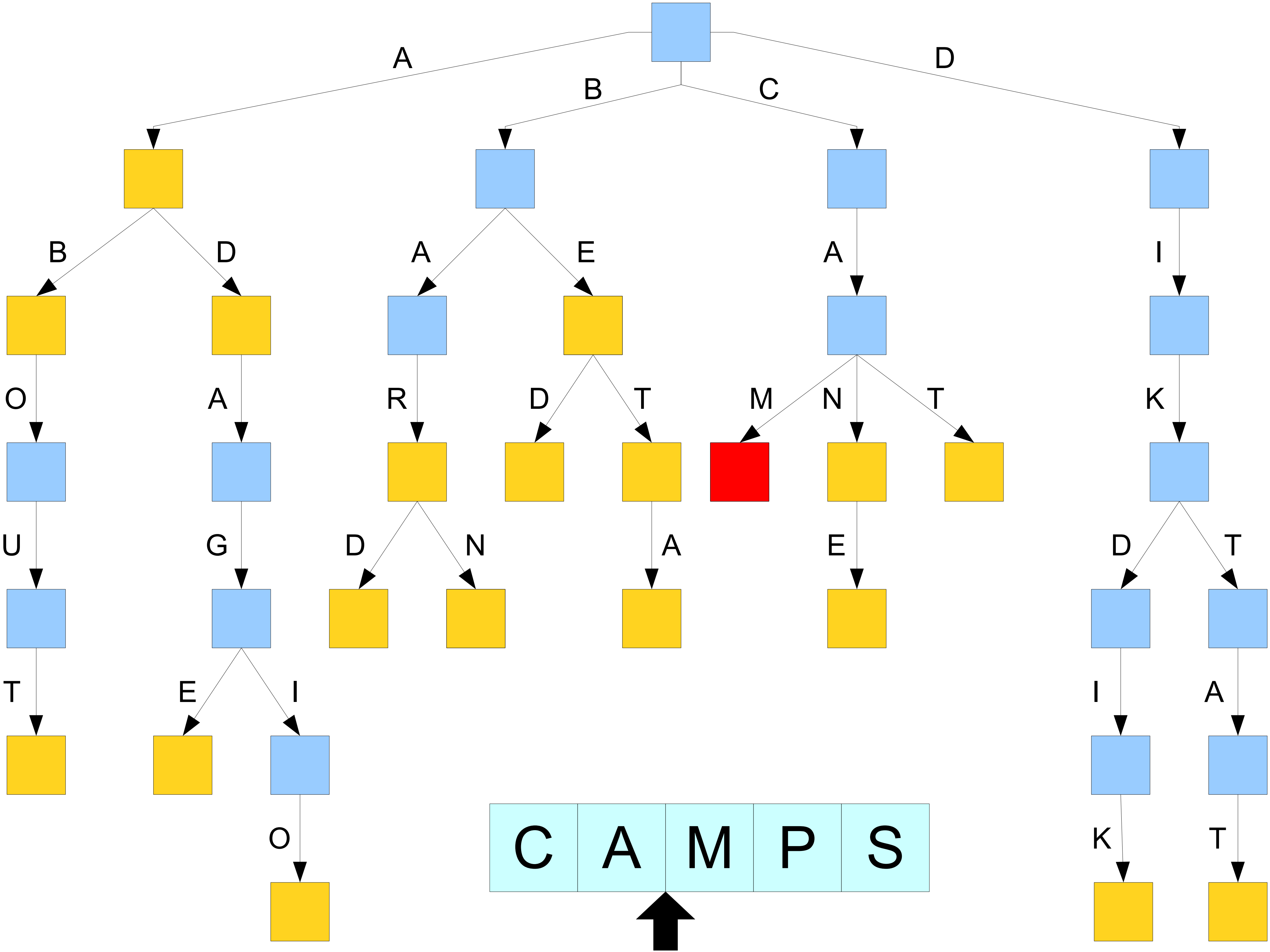

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
C A M P S

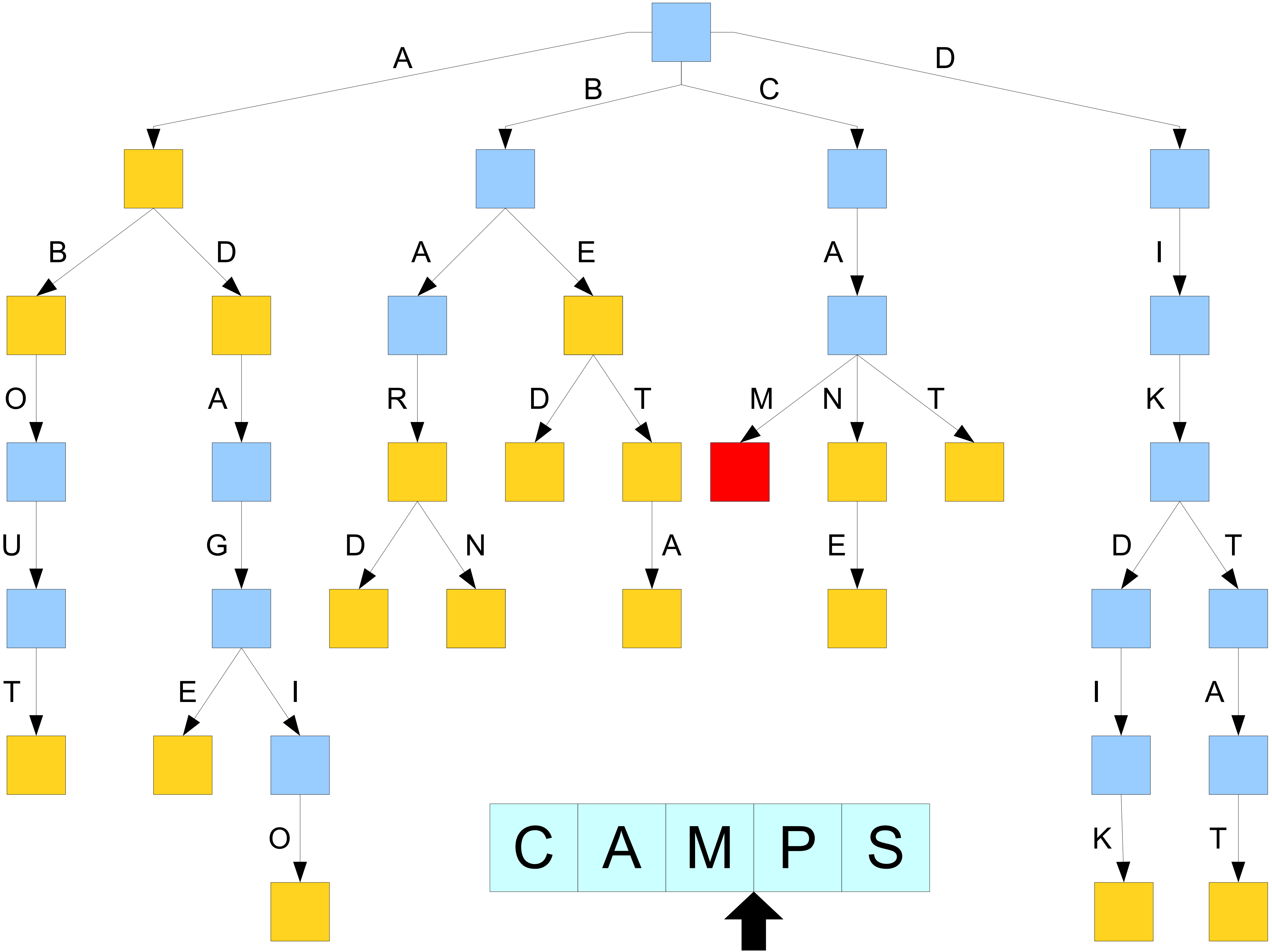

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
C A M P S

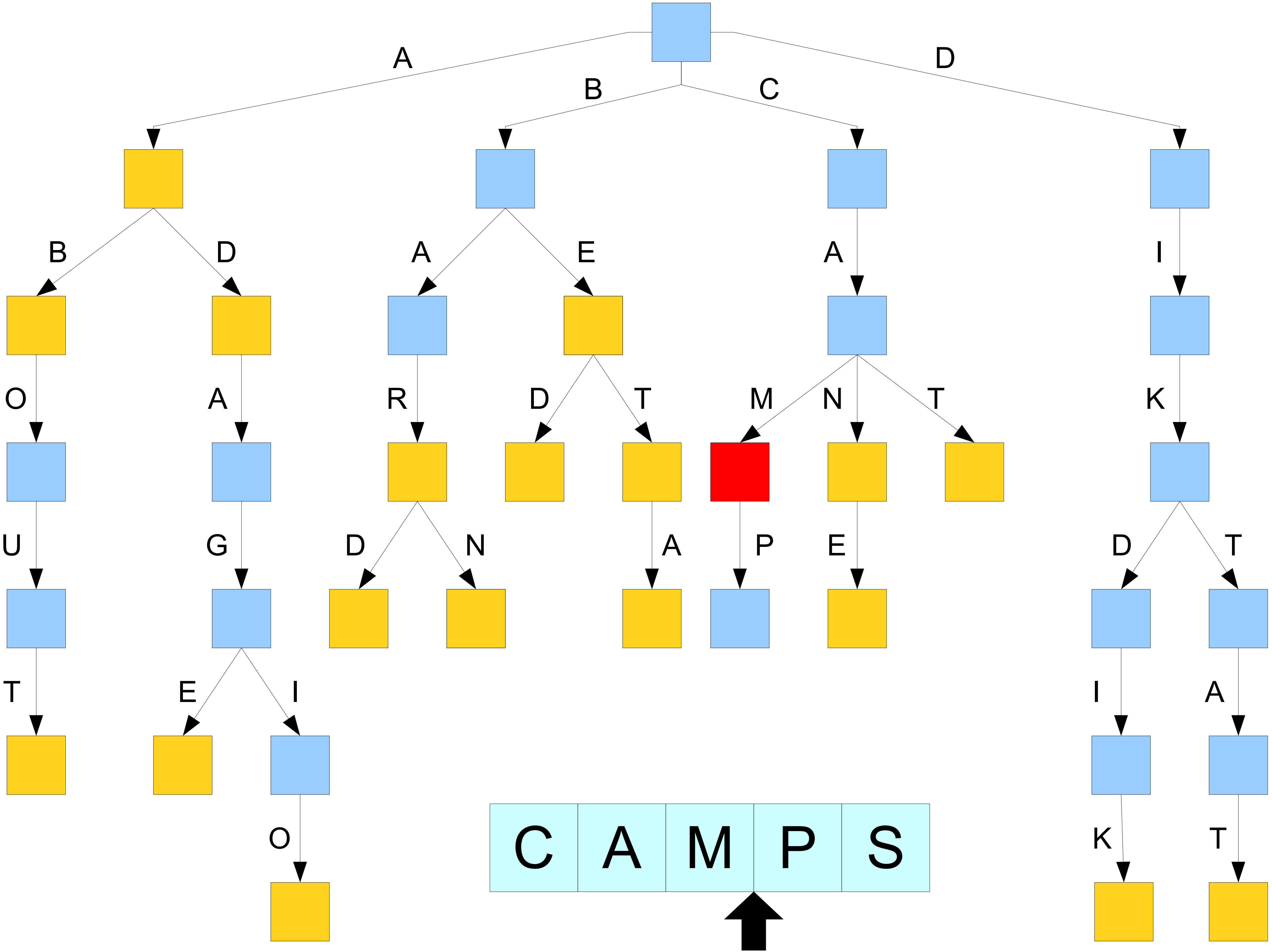

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
C A M P S

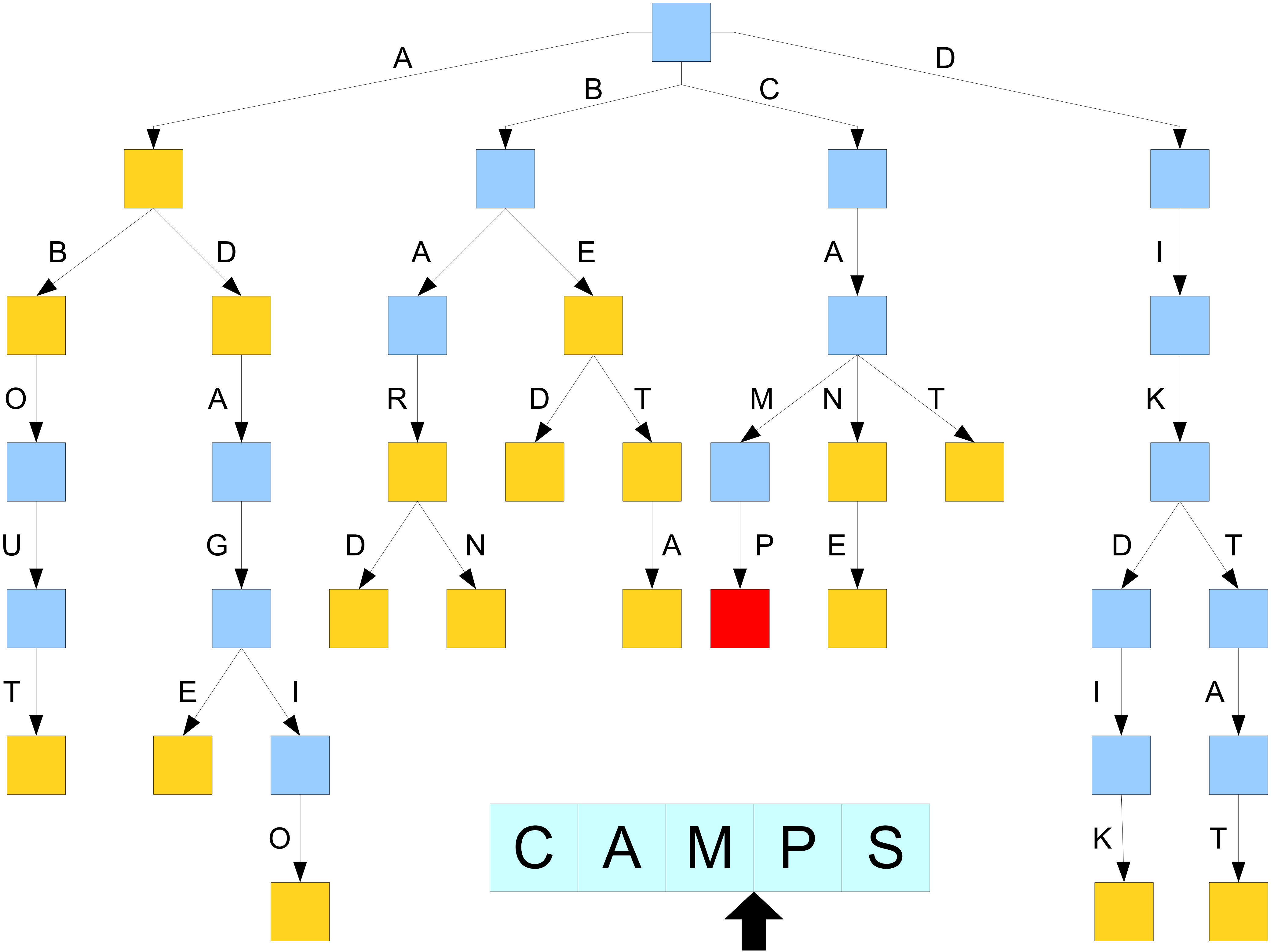

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
C
A
M
P
S


A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
C A M P S

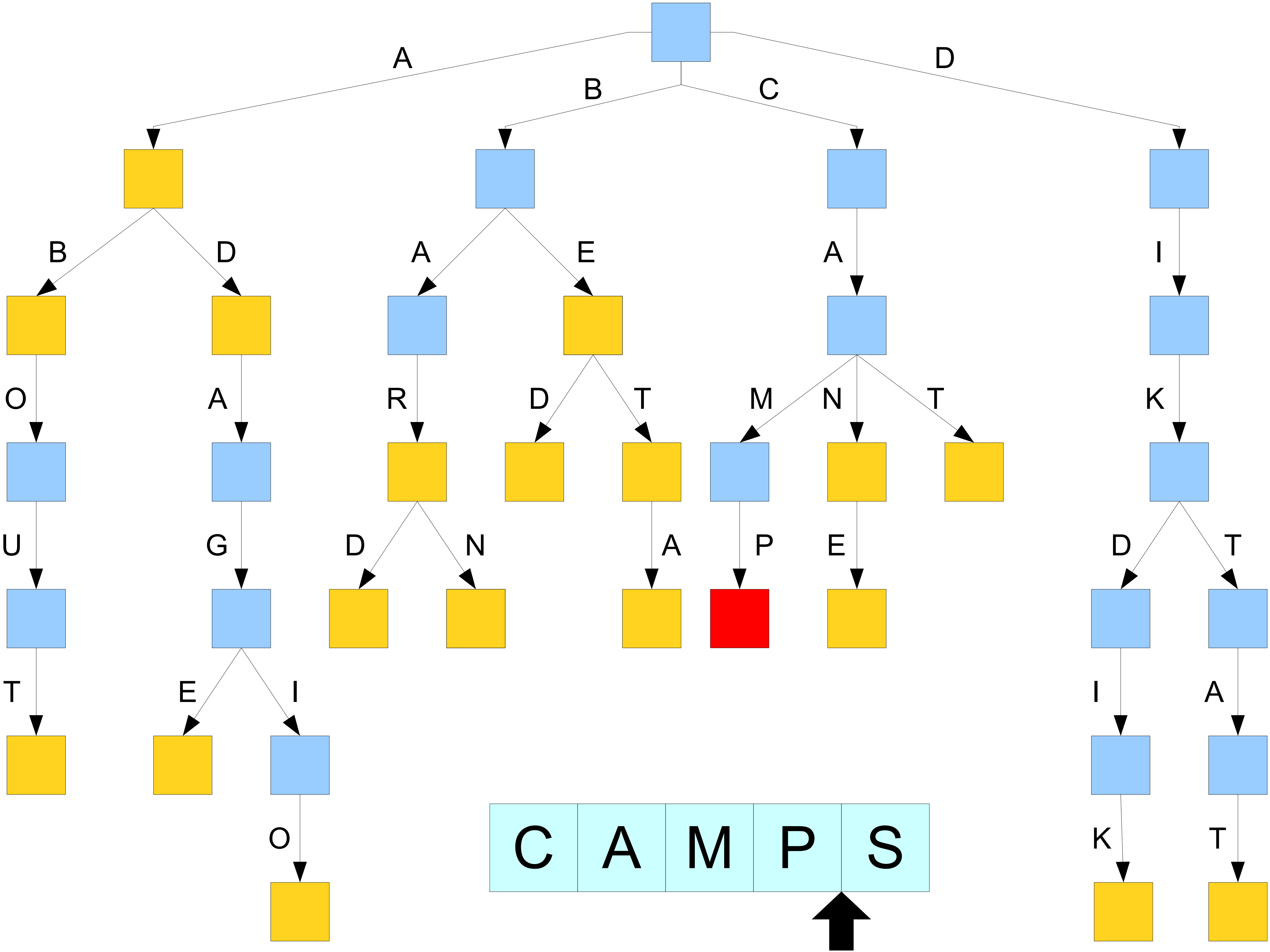

A
D
B
C
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
C A M P S


A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
C A M P S

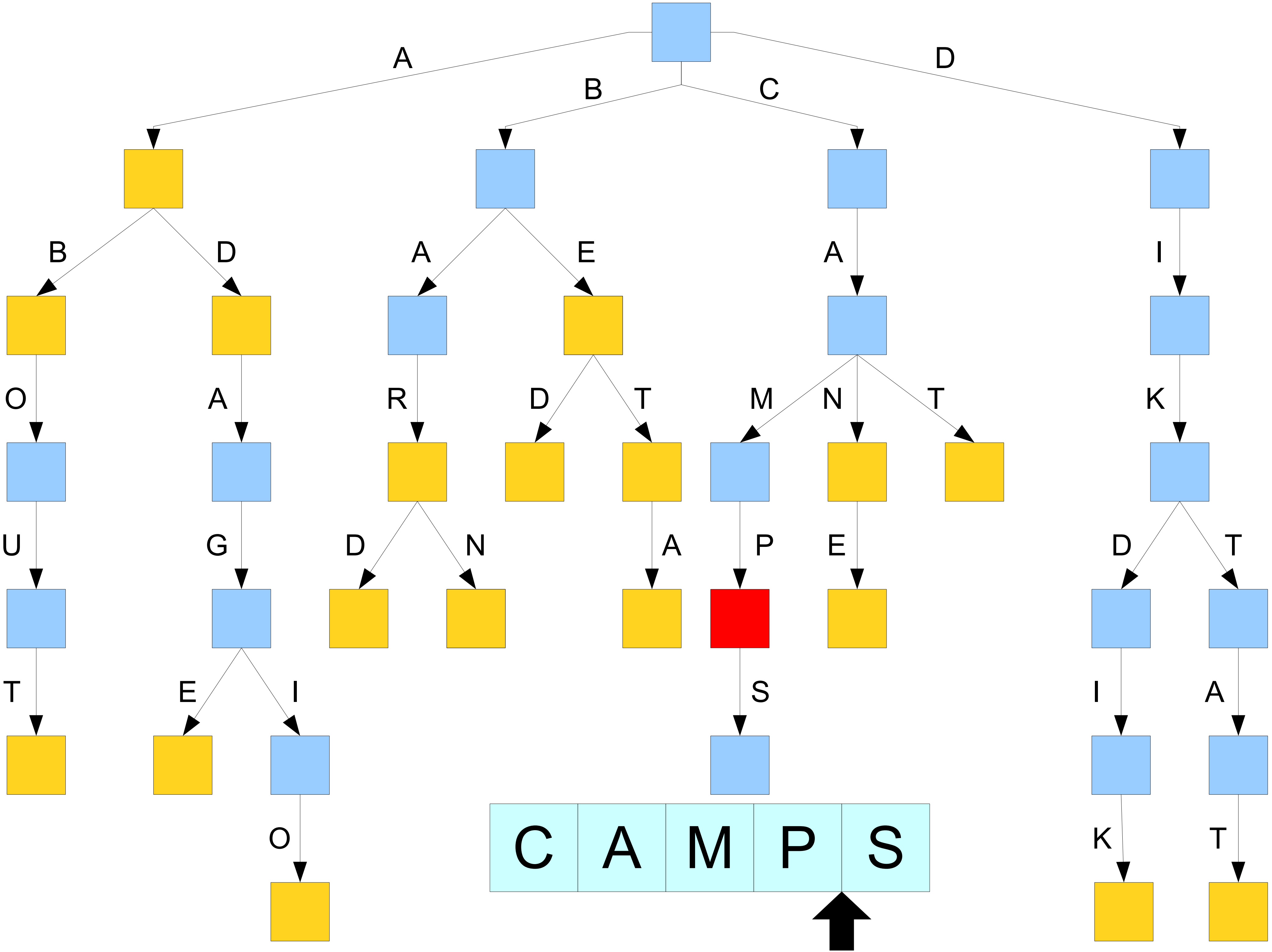

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
P
E
D
T
T
E
I
I
A
O
K
T
C A M P S

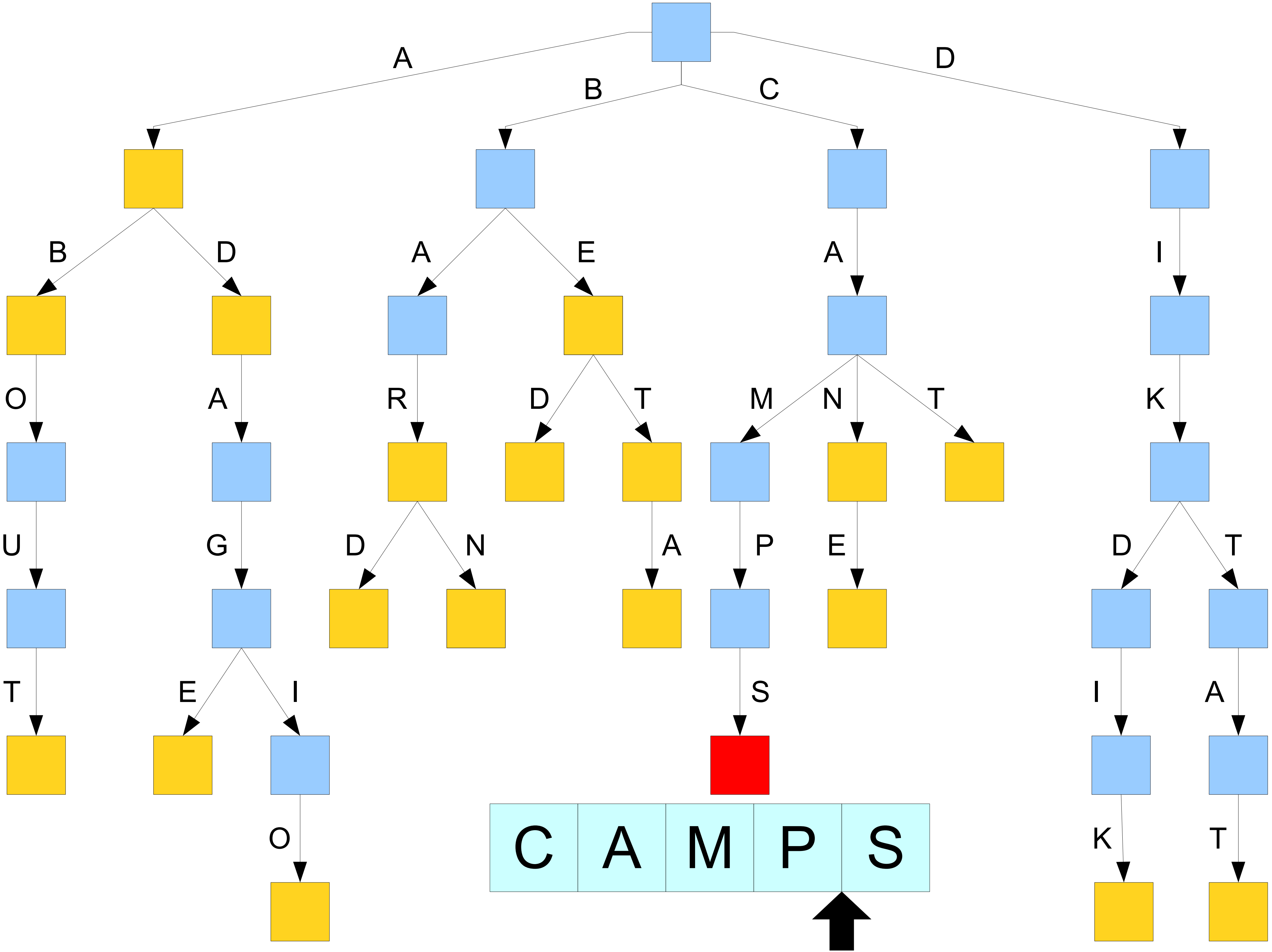


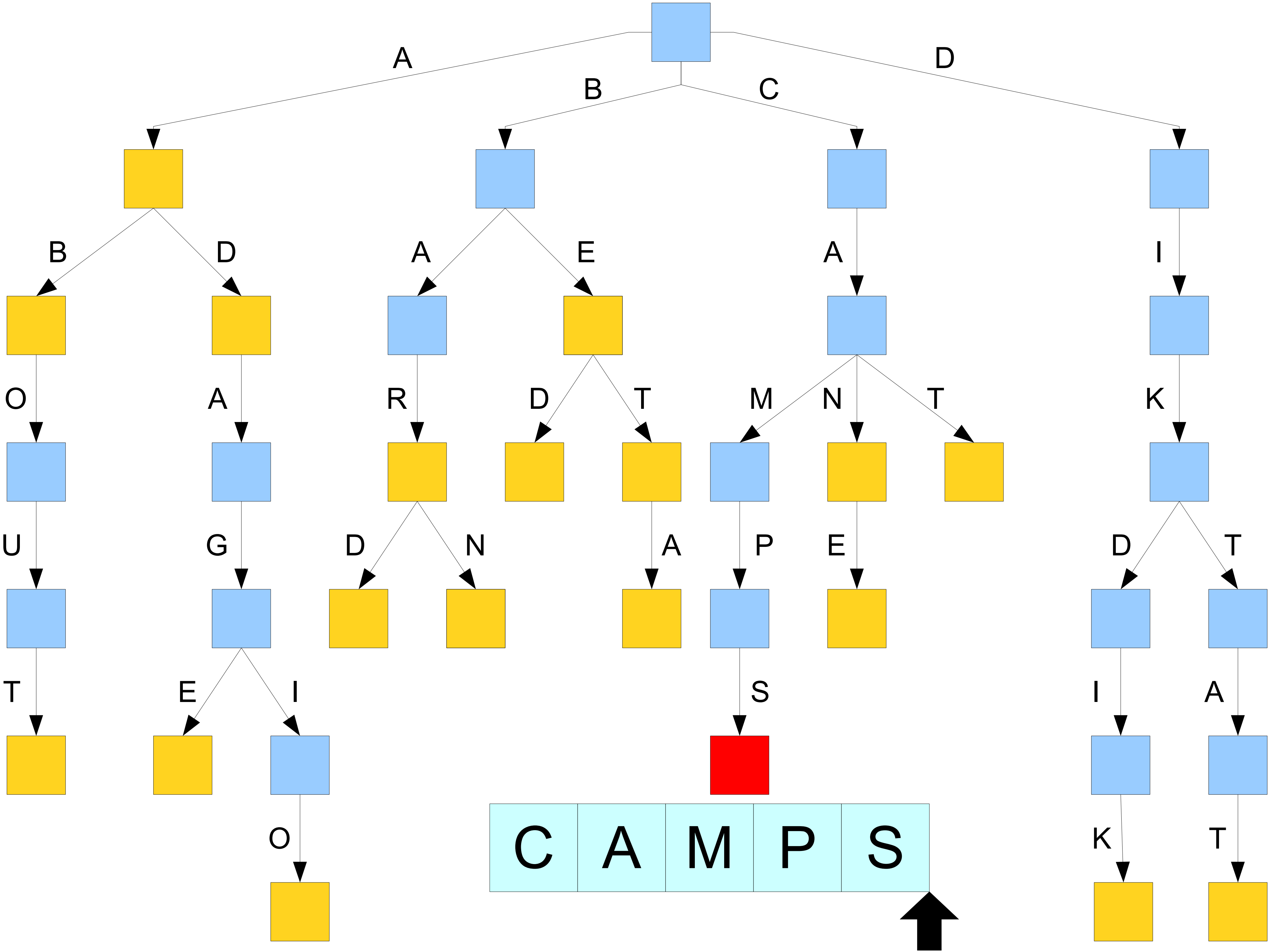
A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
P
E
D
T
T
E
I
I
A
O
K
T
C A M P S

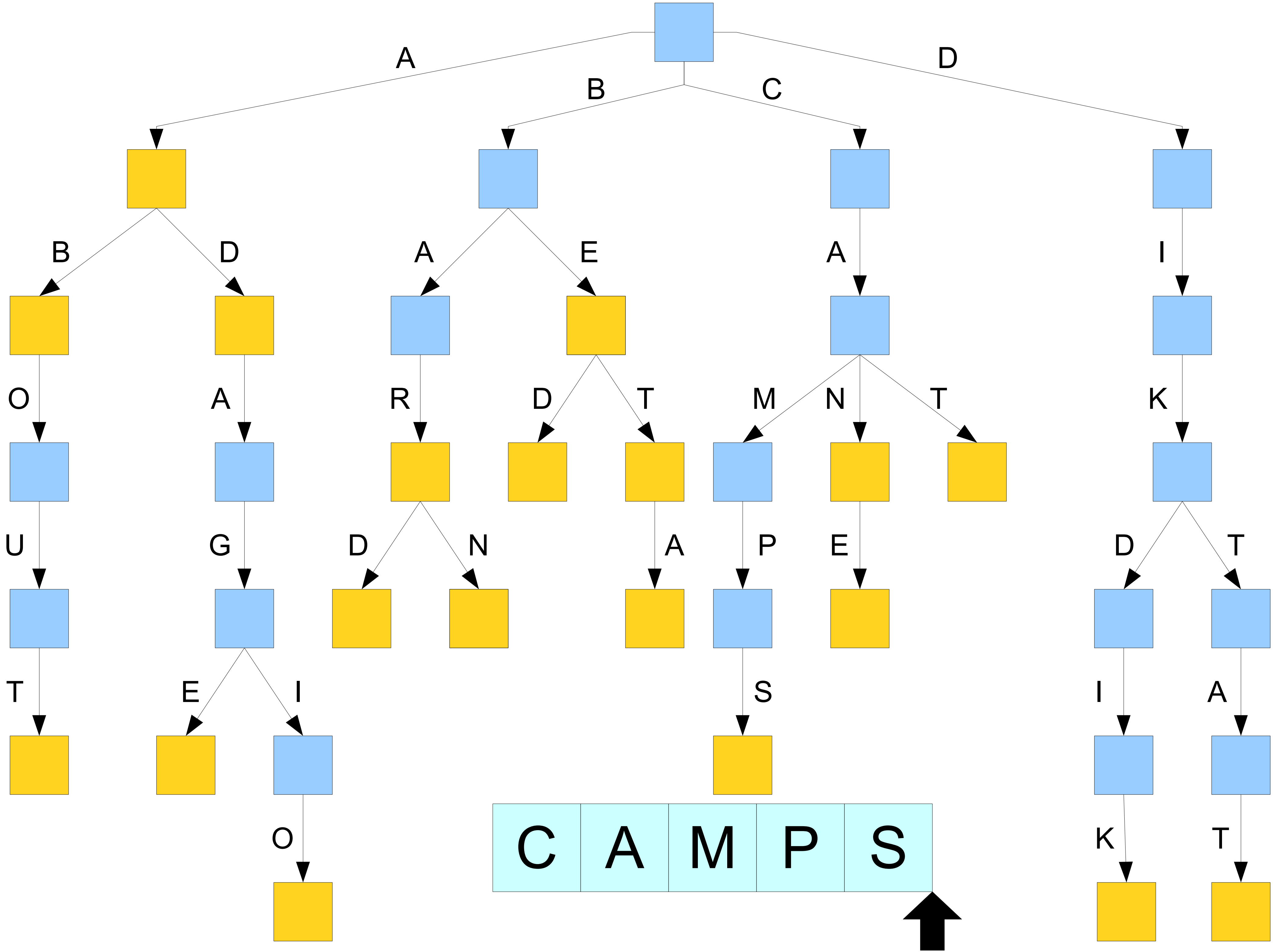


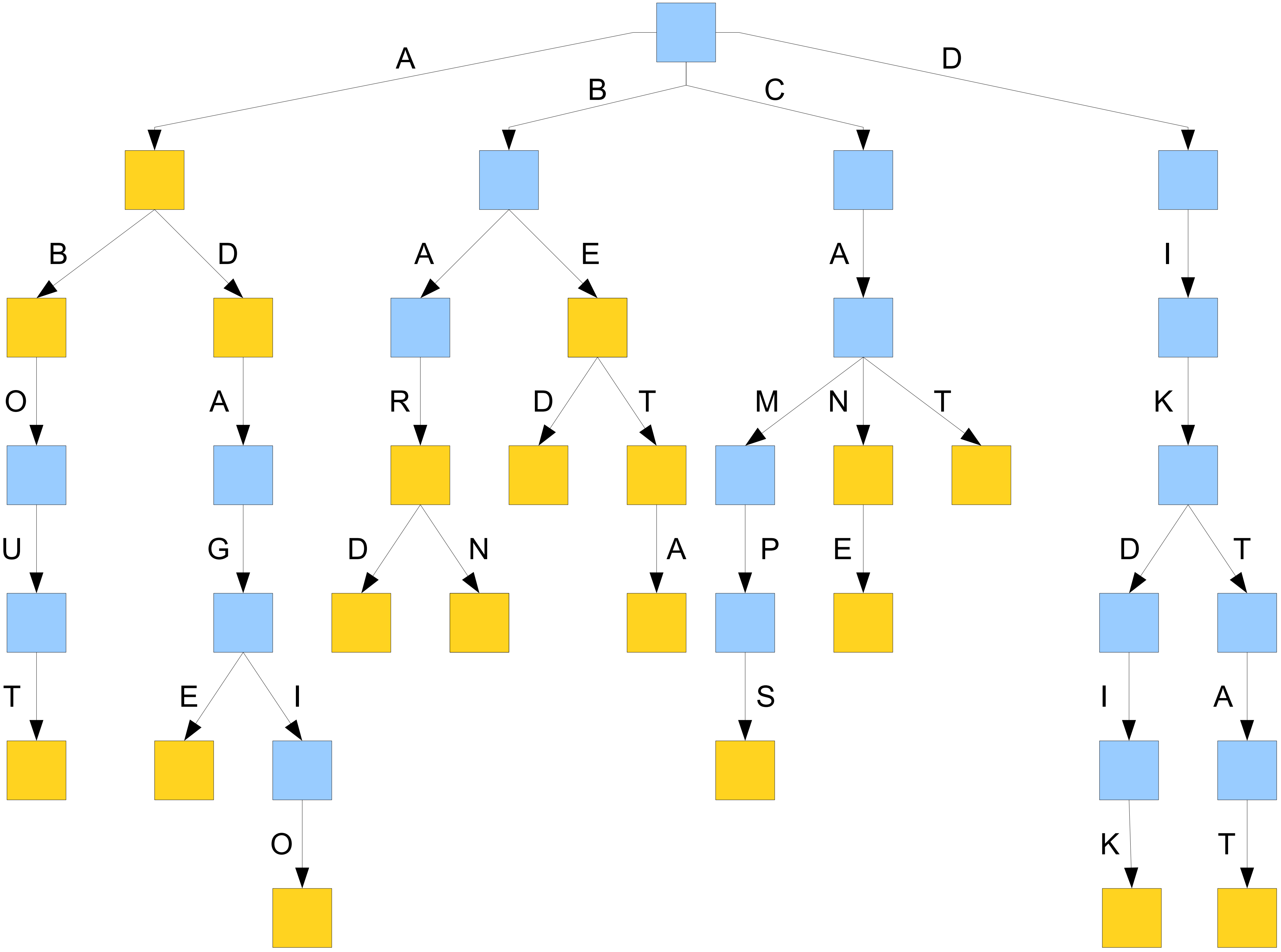
A
D
B
C
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
P
E
D
T
T
E
I
I
A
O
K
T
C A M P S

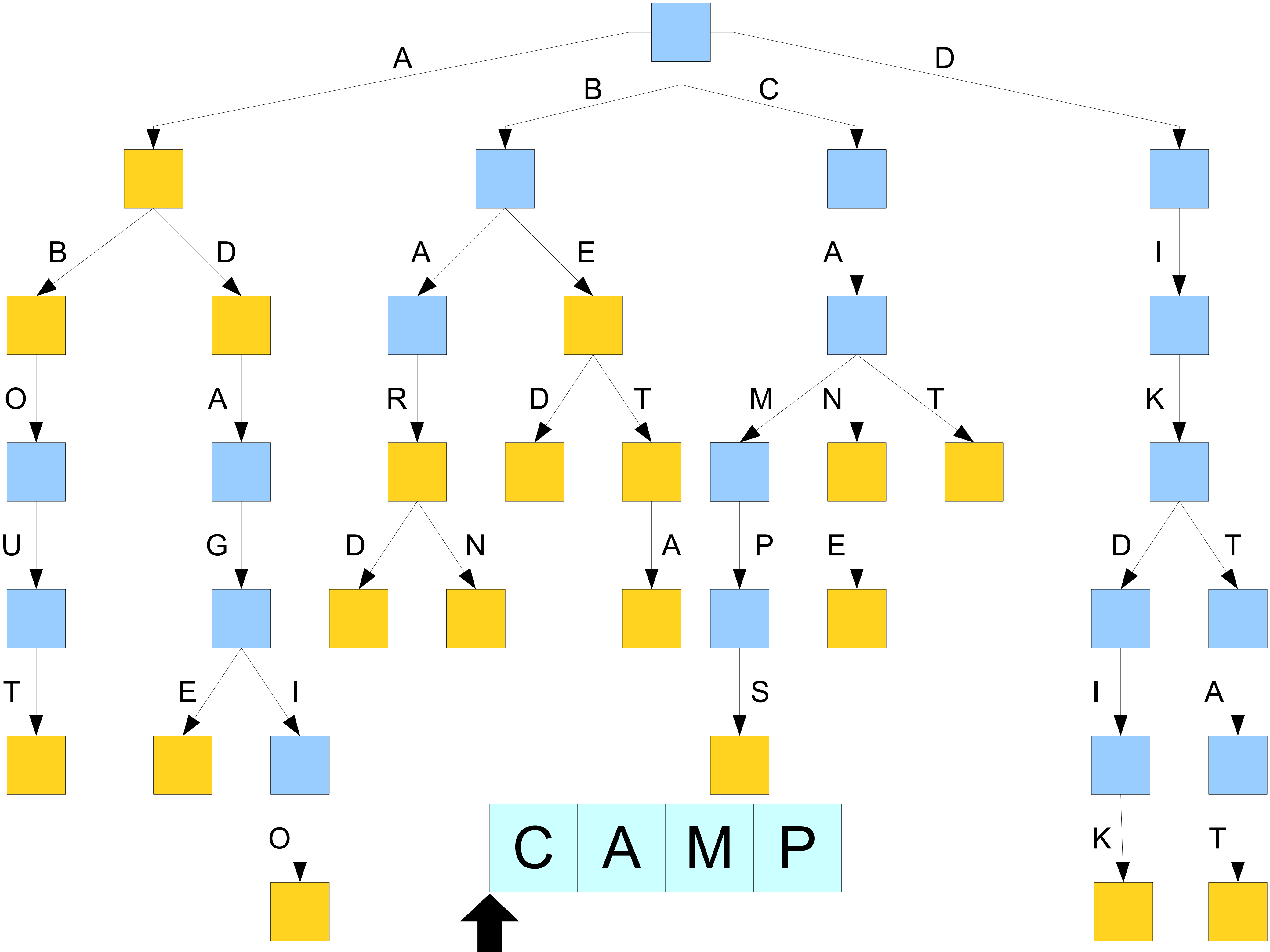

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
P
E
D
T
T
E
I
S
I
A
O
C A M P S
K
T

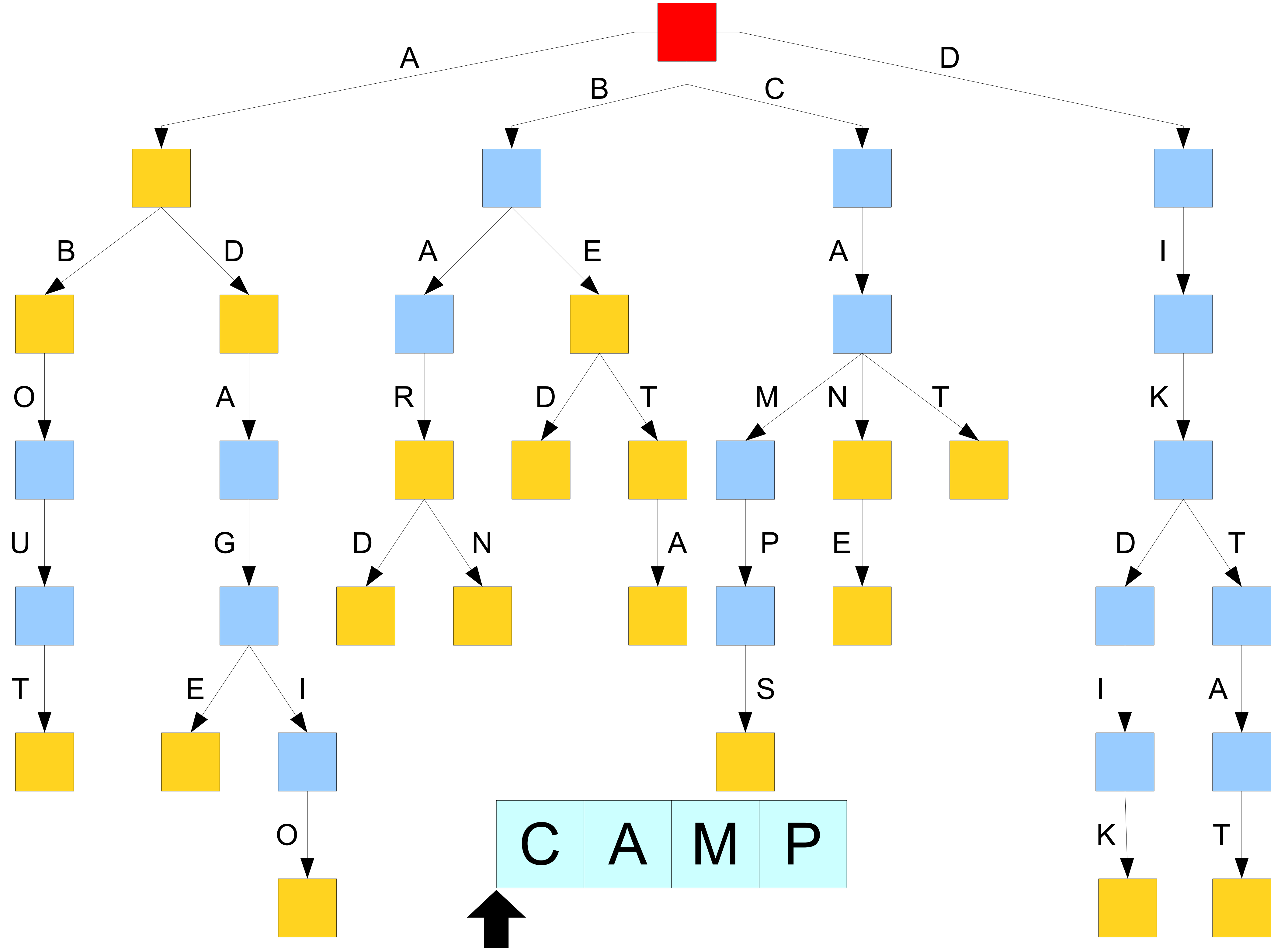

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
P
E
D
T
T
E
I
S
I
A
O
C A M P S
K
T

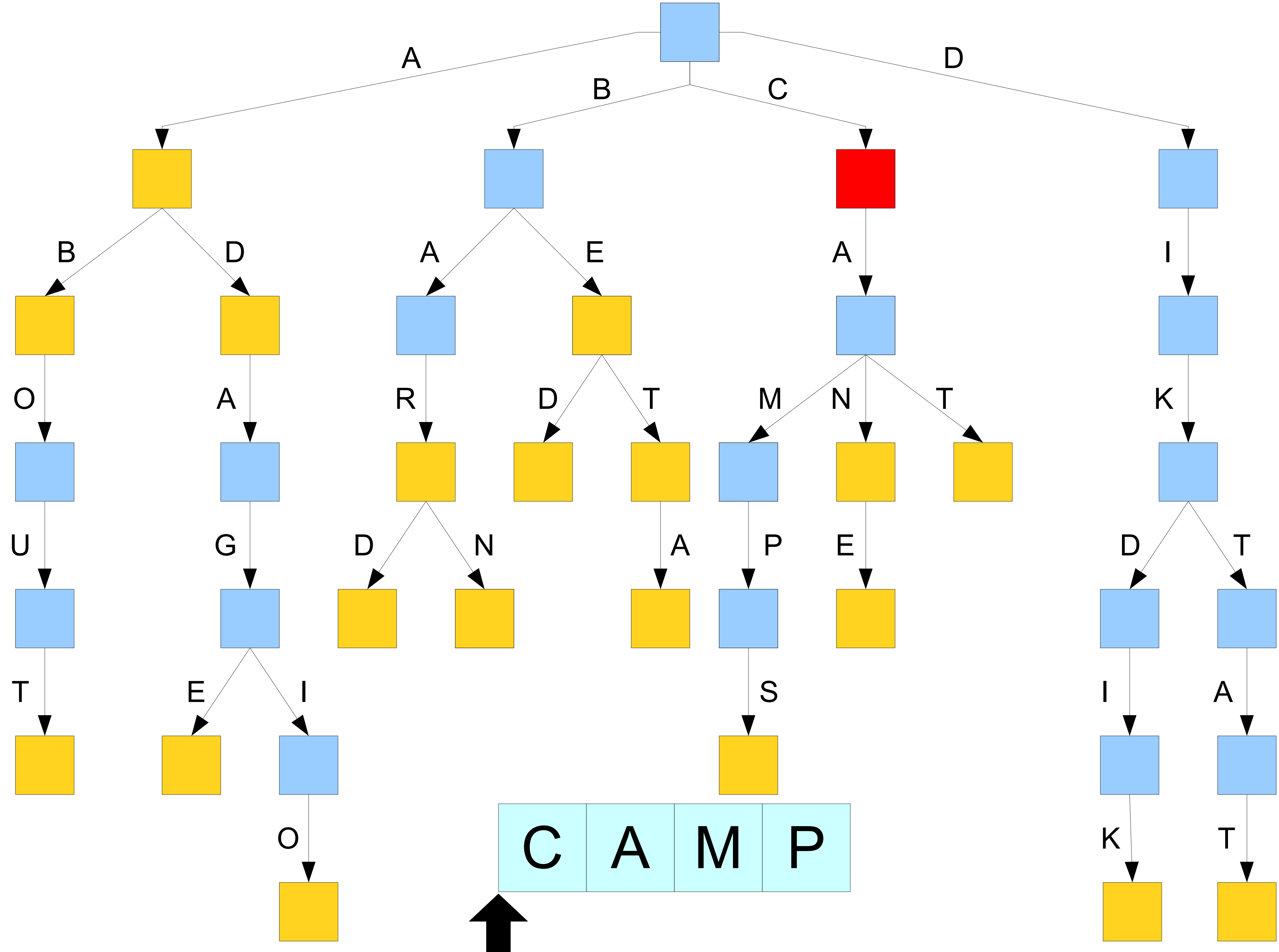

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
P
E
D
T
T
E
I
S
I
A
O
K
T
C A M P S

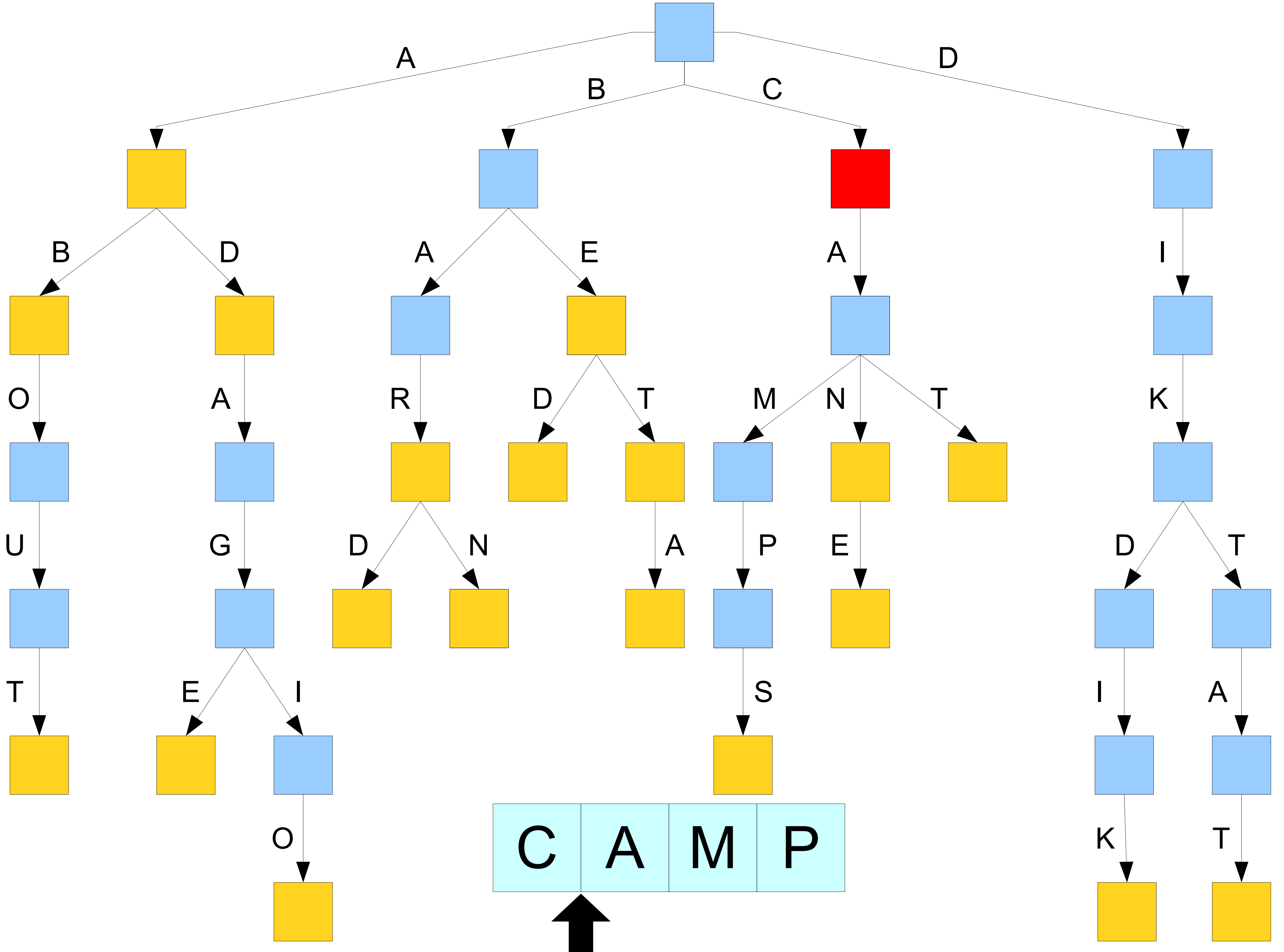

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
P
E
D
T
T
E
I
S
I
A
O
C A M P S
K
T


A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
P
E
D
T
T
E
I
S
I
A
O
K
T

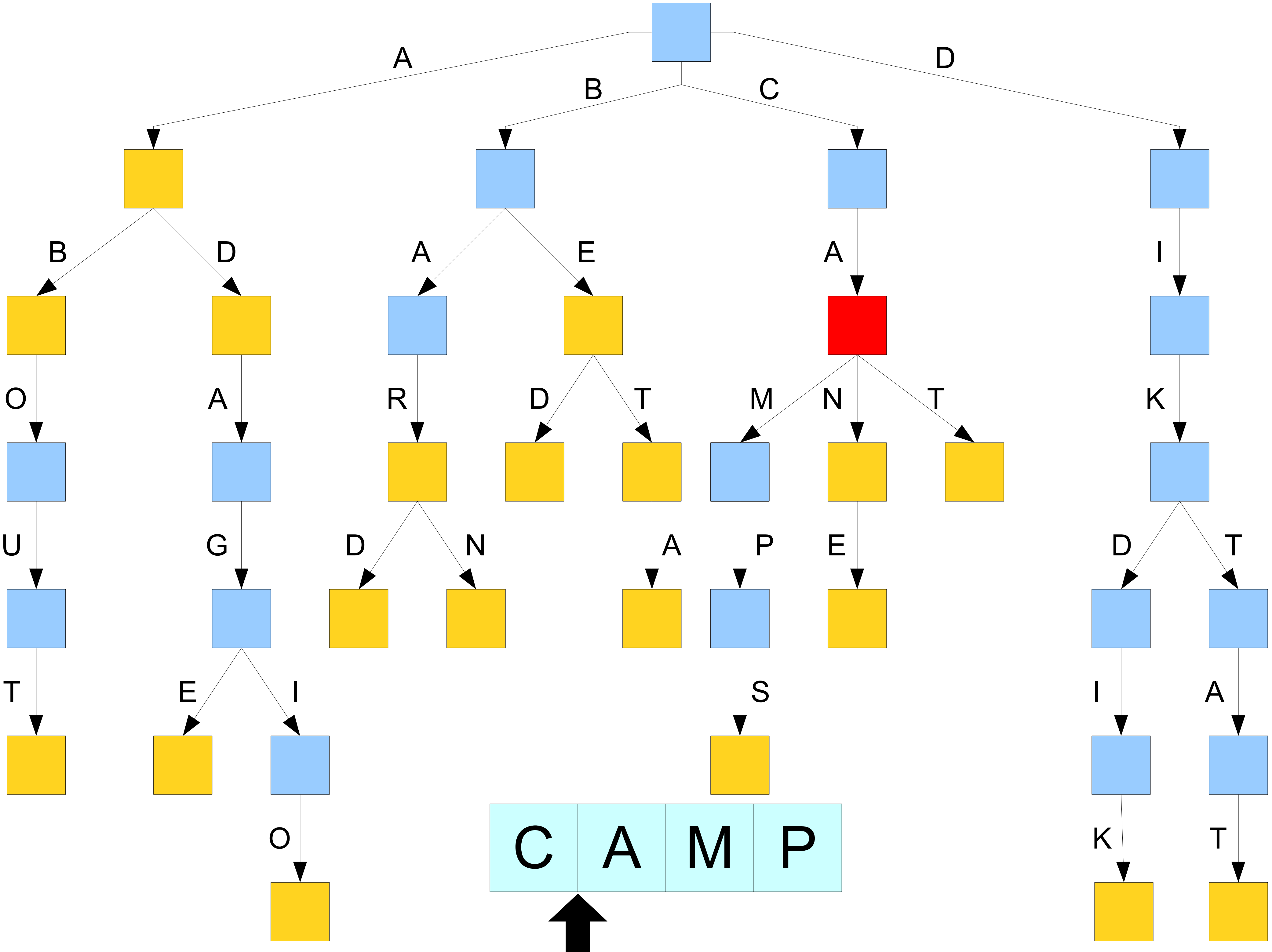

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
P
E
D
T
T
E
I
S
I
A
O
K
T
CAMP

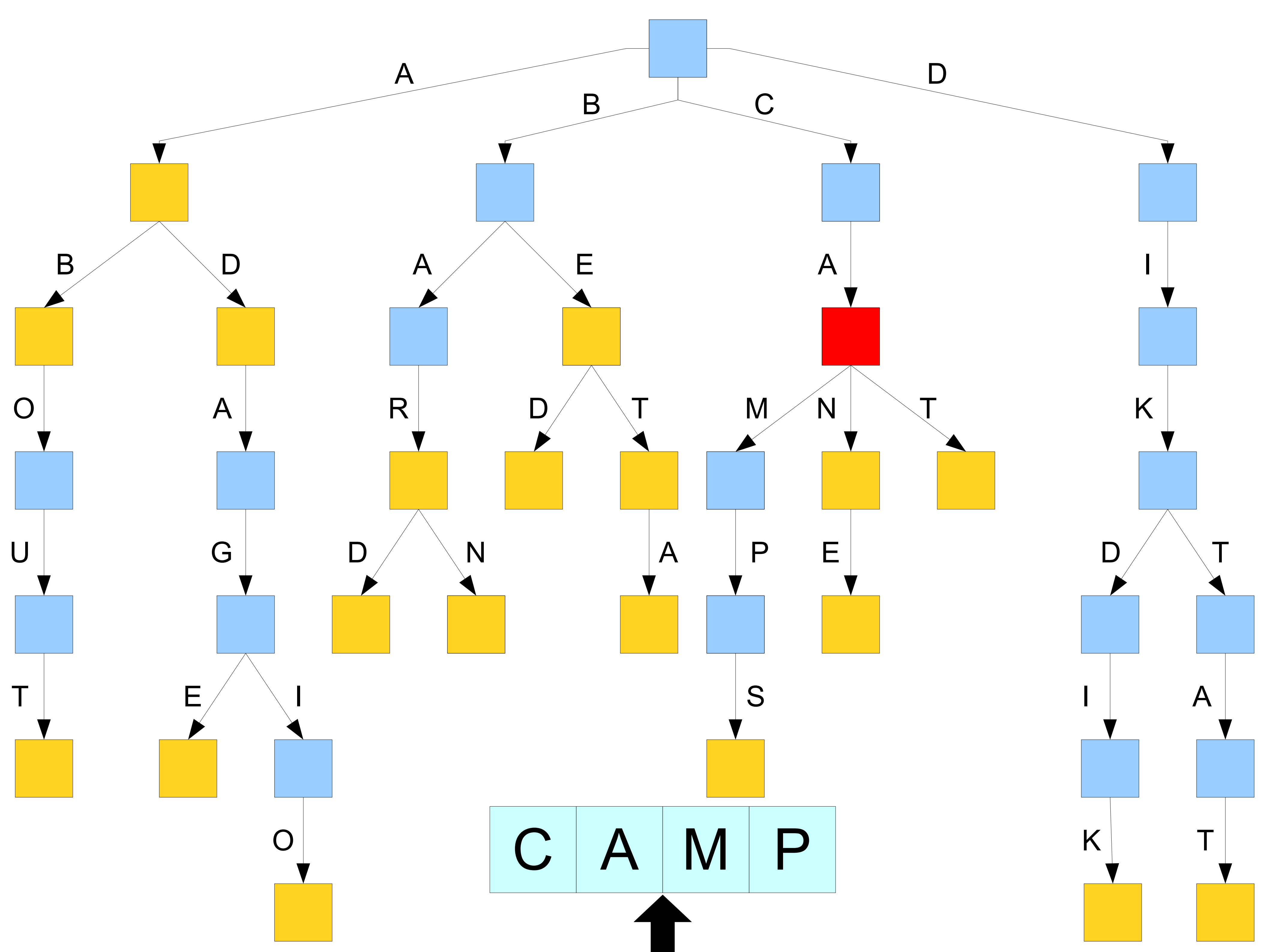

A B C D
B D A E A I
O A R D T M N T K
U G D N A P E D T
T E I S I A
O K T
C A M P

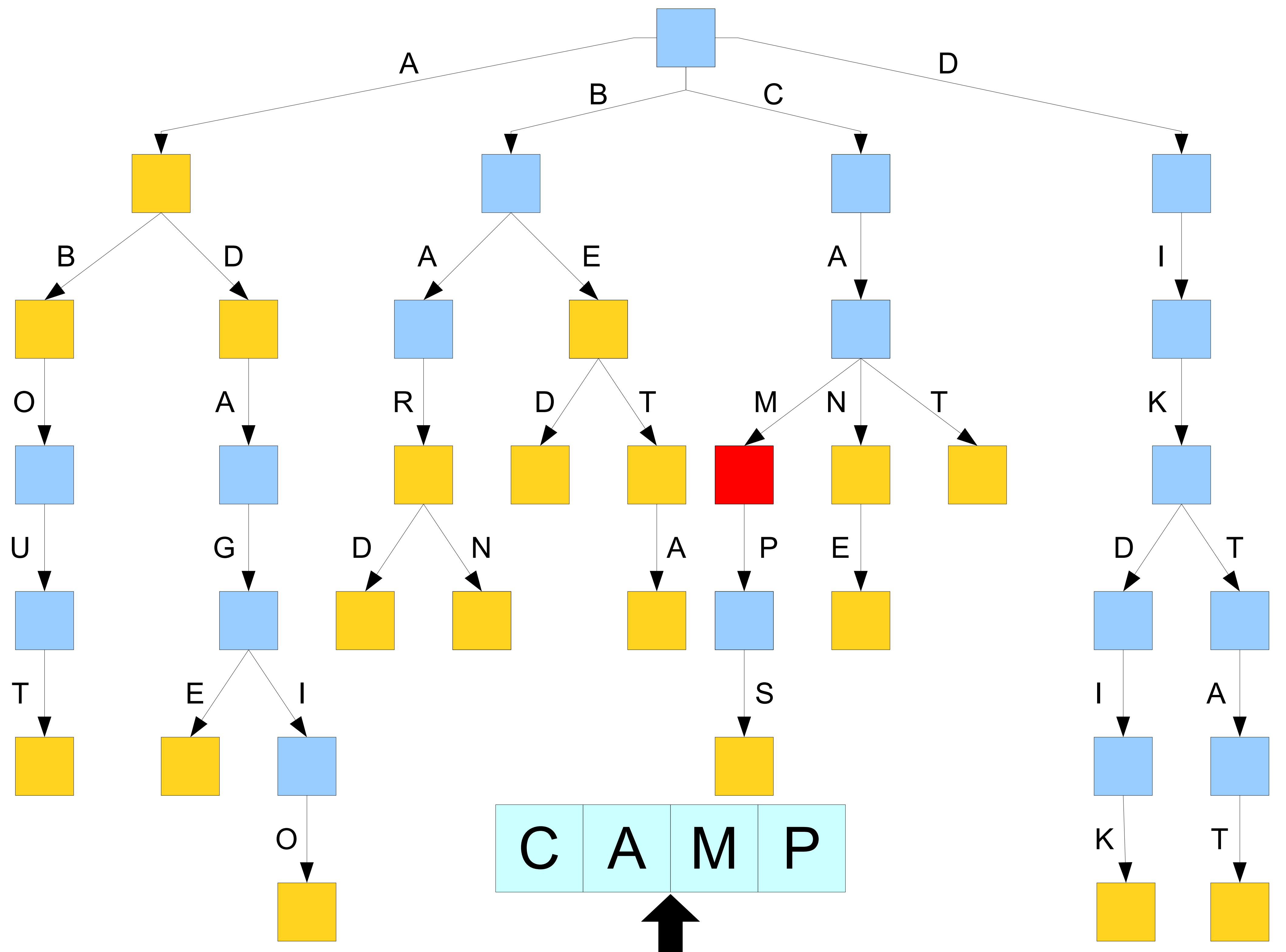

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
P
E
D
T
T
E
I
S
I
A
O
K
T
CAMP

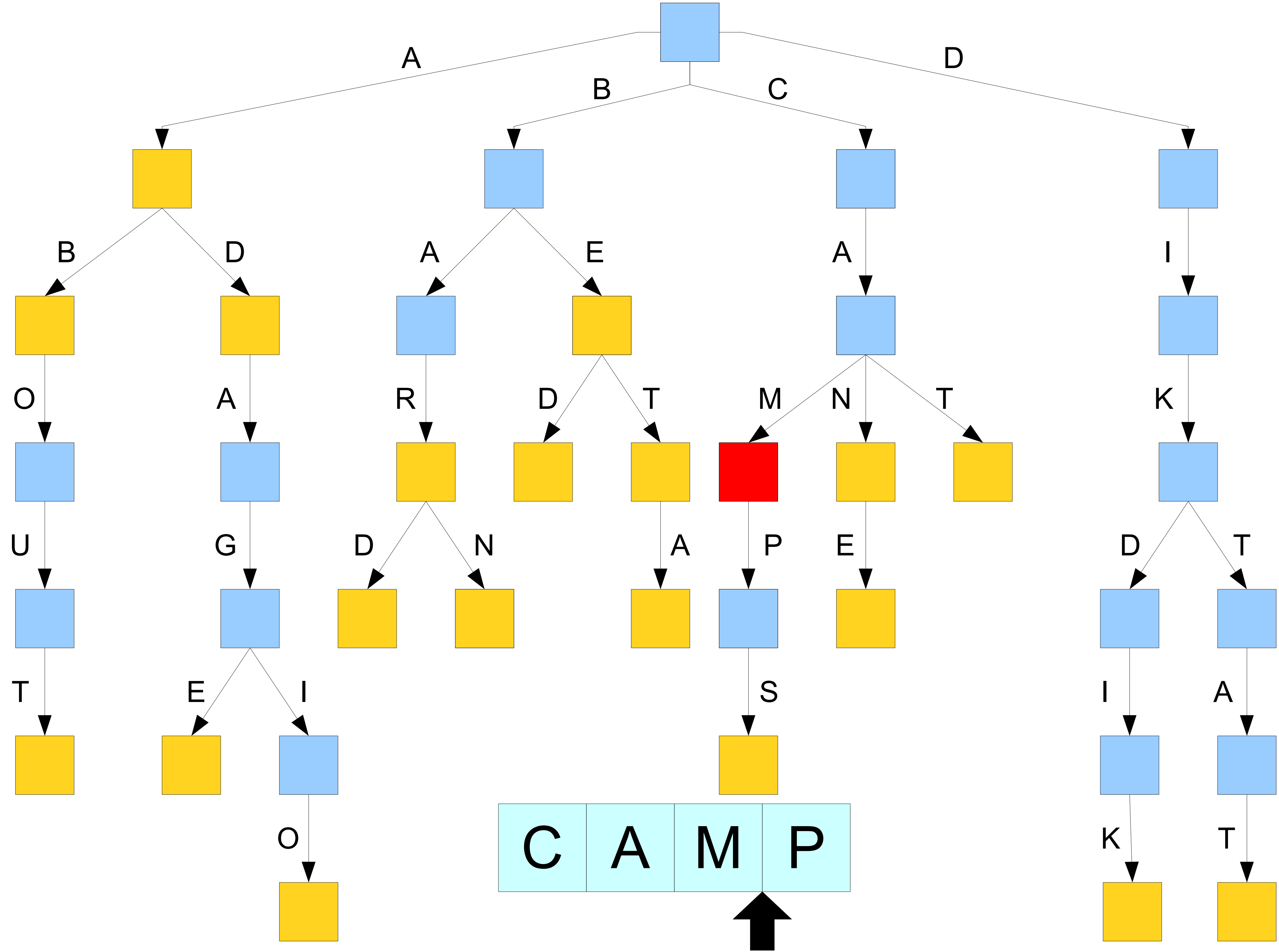


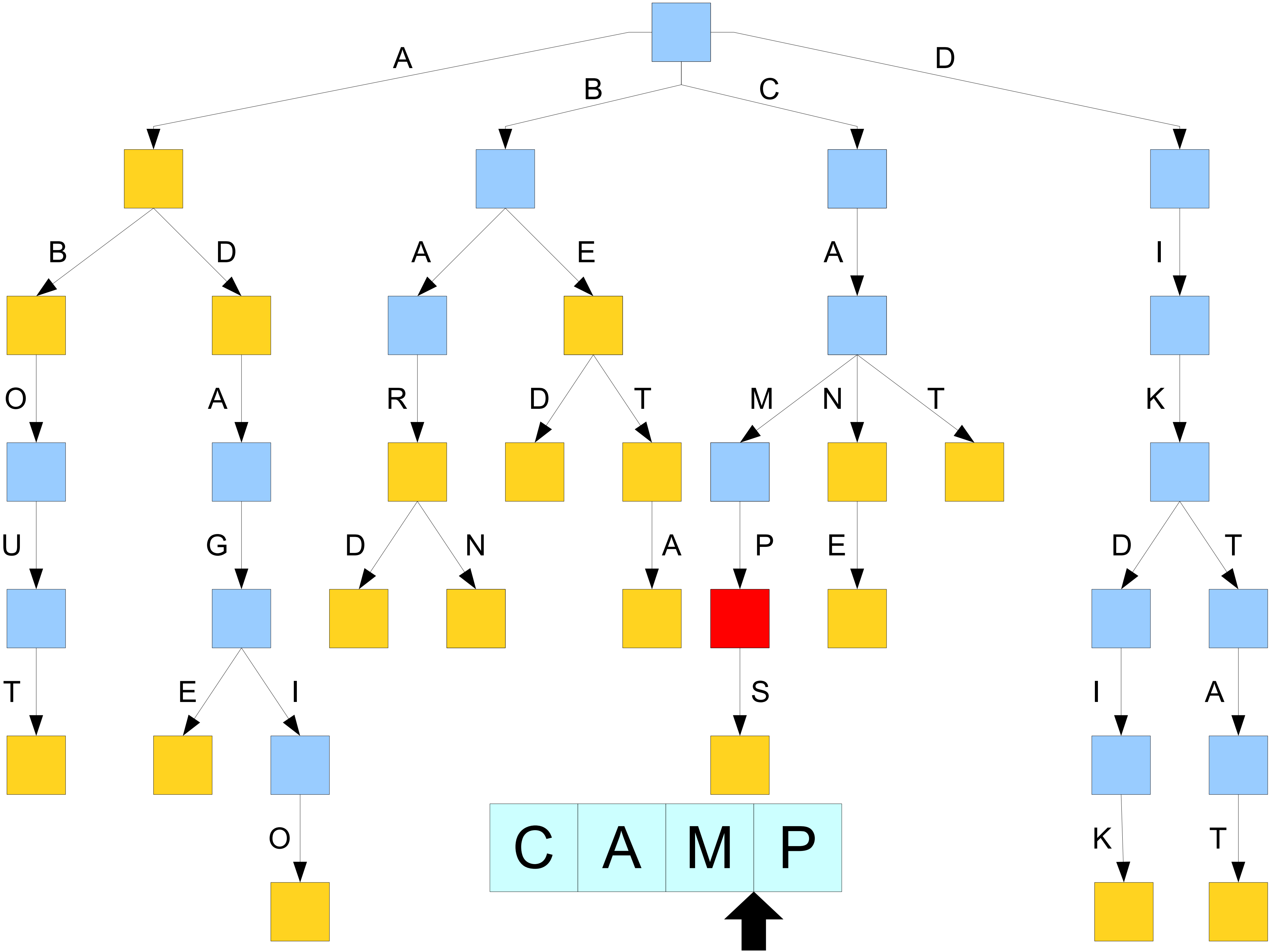
A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
P
E
D
T
T
E
I
S
I
A
O
K
T
C A M P

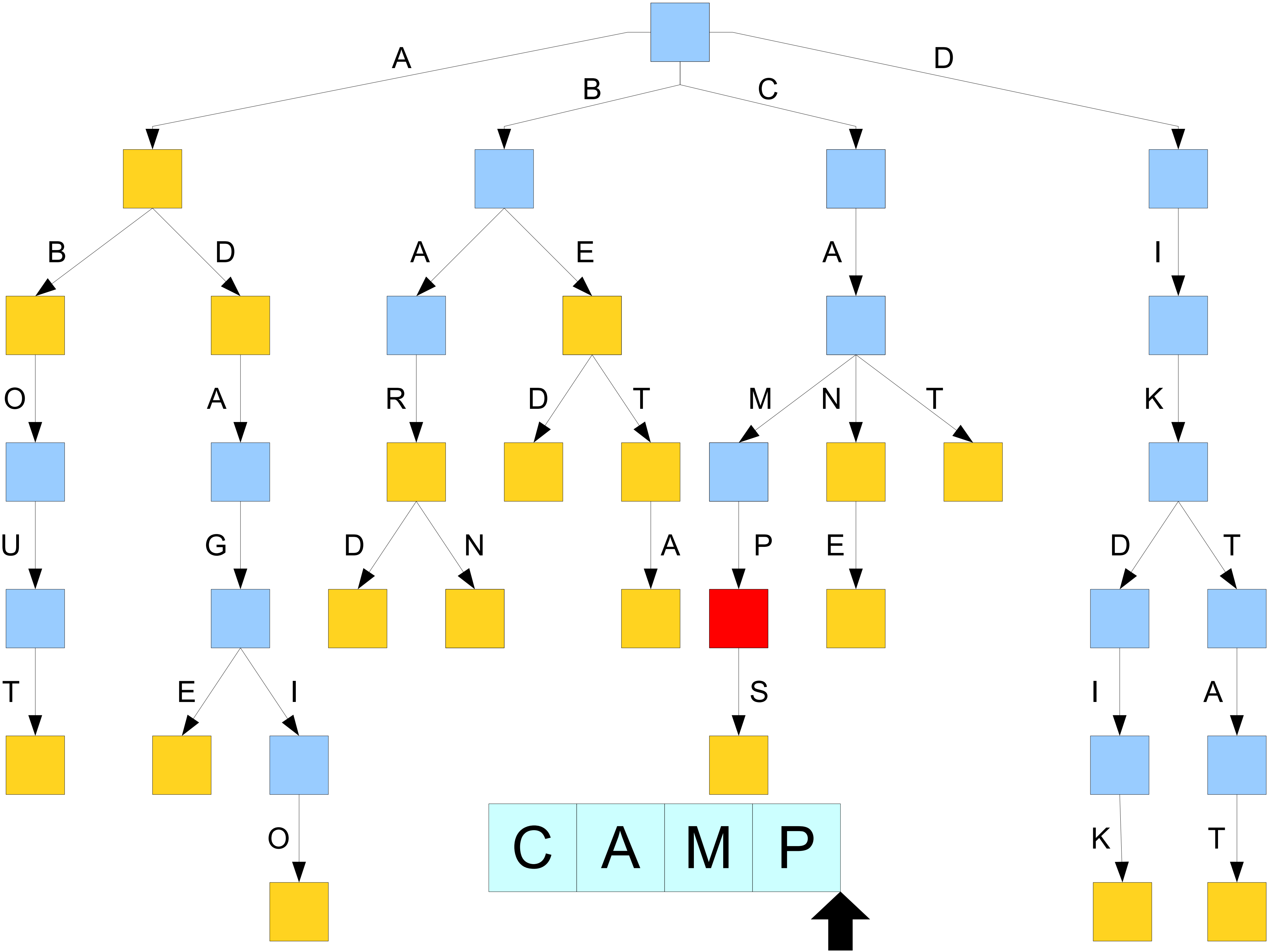

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
P
E
D
T
T
E
I
S
I
A
O
K
T
C A M P

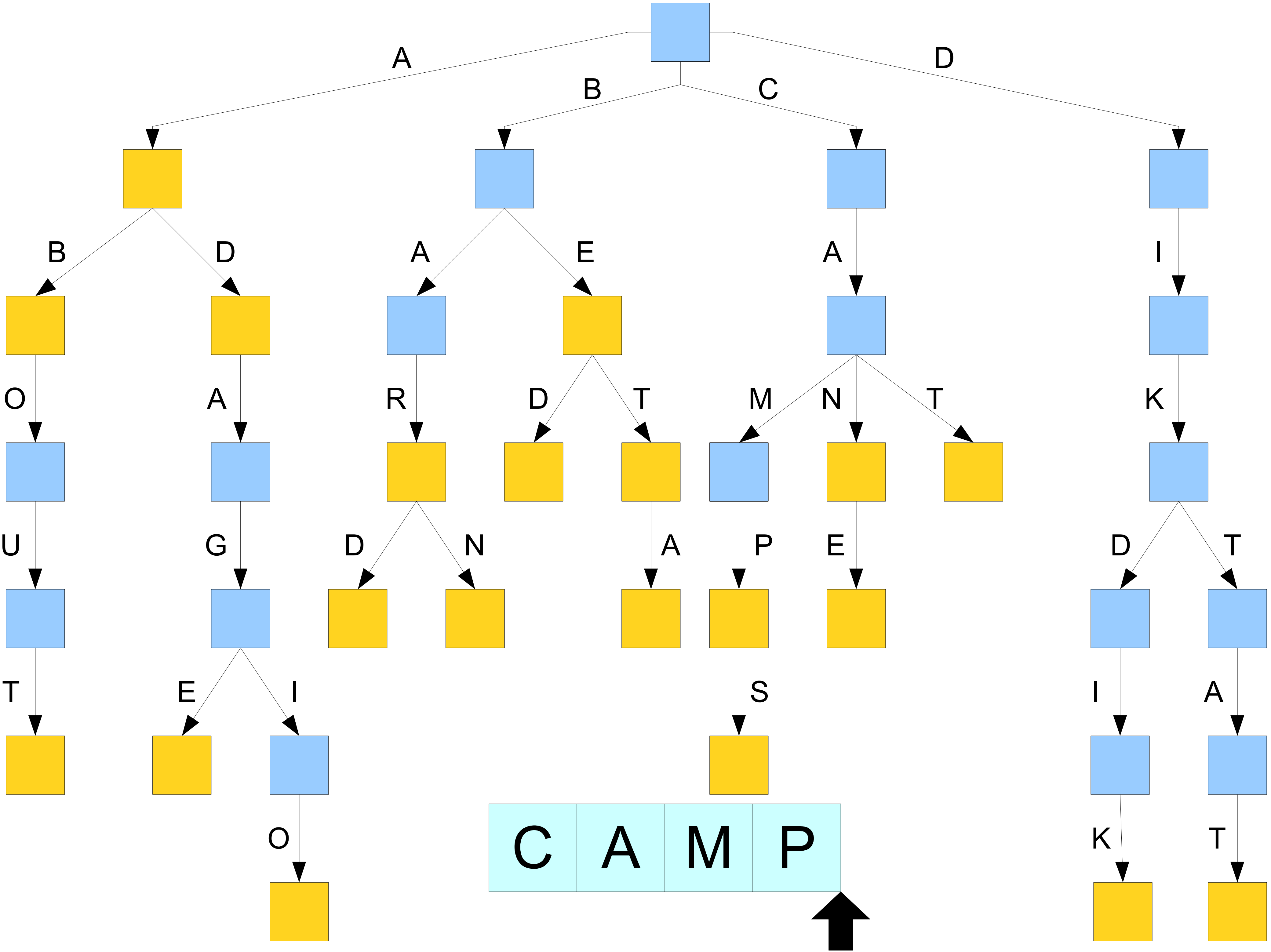

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
P
E
D
T
T
E
I
S
I
A
O
K
T
C A M P


A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
P
E
D
T
T
E
I
S
I
A
O
K
T
C A M P


A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
P
E
D
T
T
E
I
S
I
A
O
K
T
C A M P


A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
P
E
D
T
T
E
I
S
I
A
O
K
T
C
A
M
P


A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
P
E
D
T
T
E
I
S
I
A
O
K
T
C
A
M
P

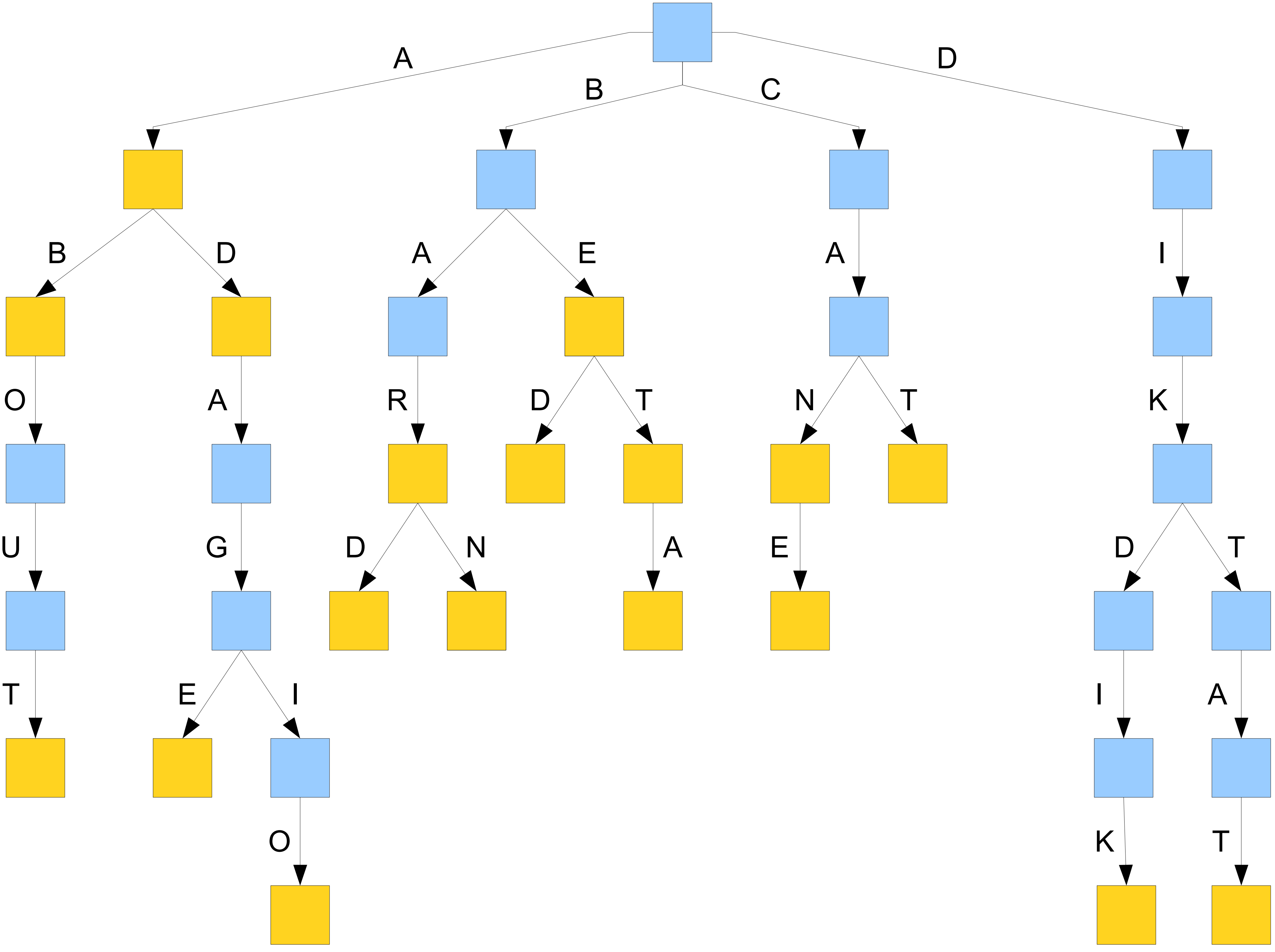

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
P
E
D
T
T
E
I
S
I
A
O
K
T
C A M P

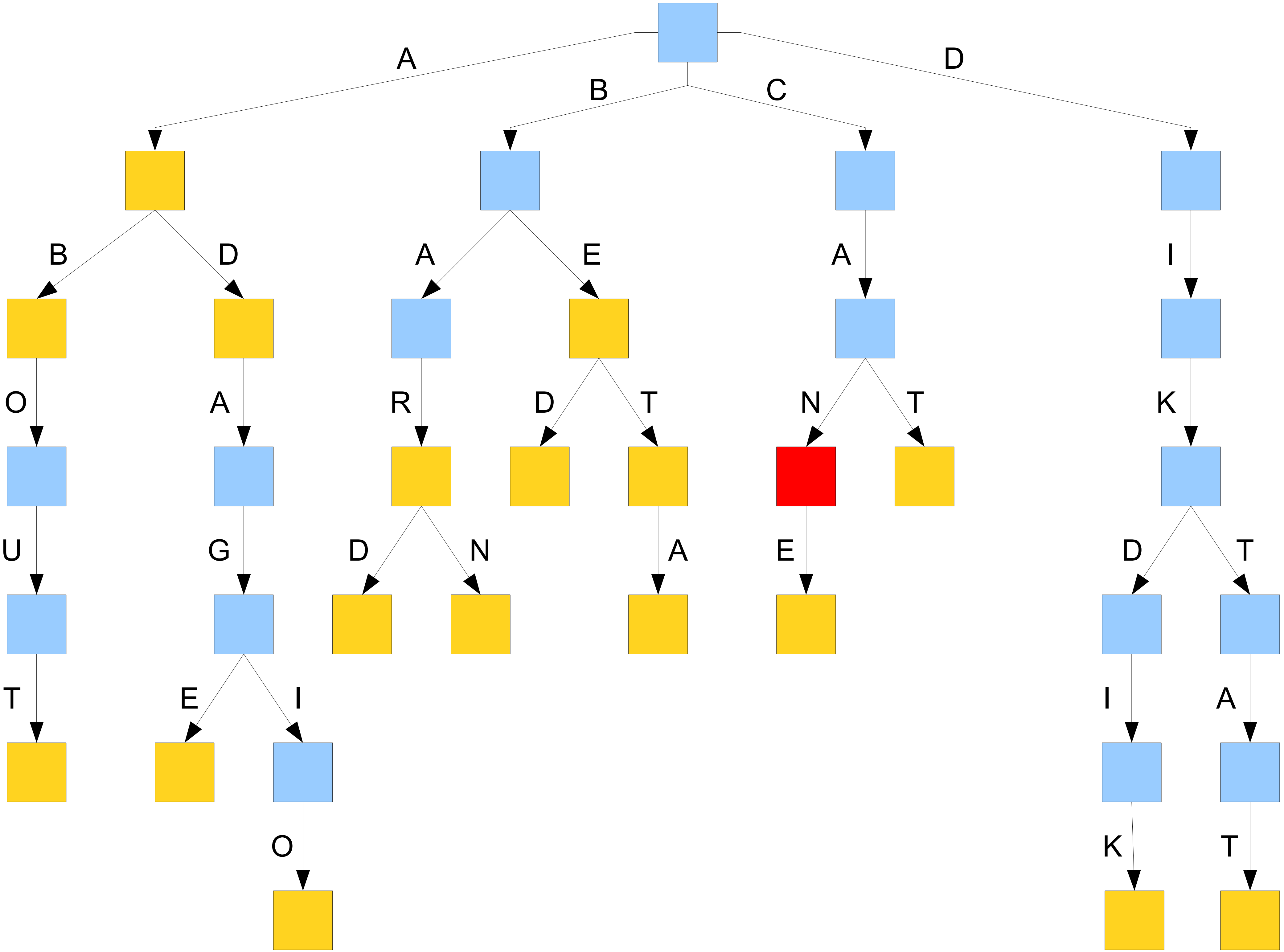
Let's trie coding up **`insert`**!

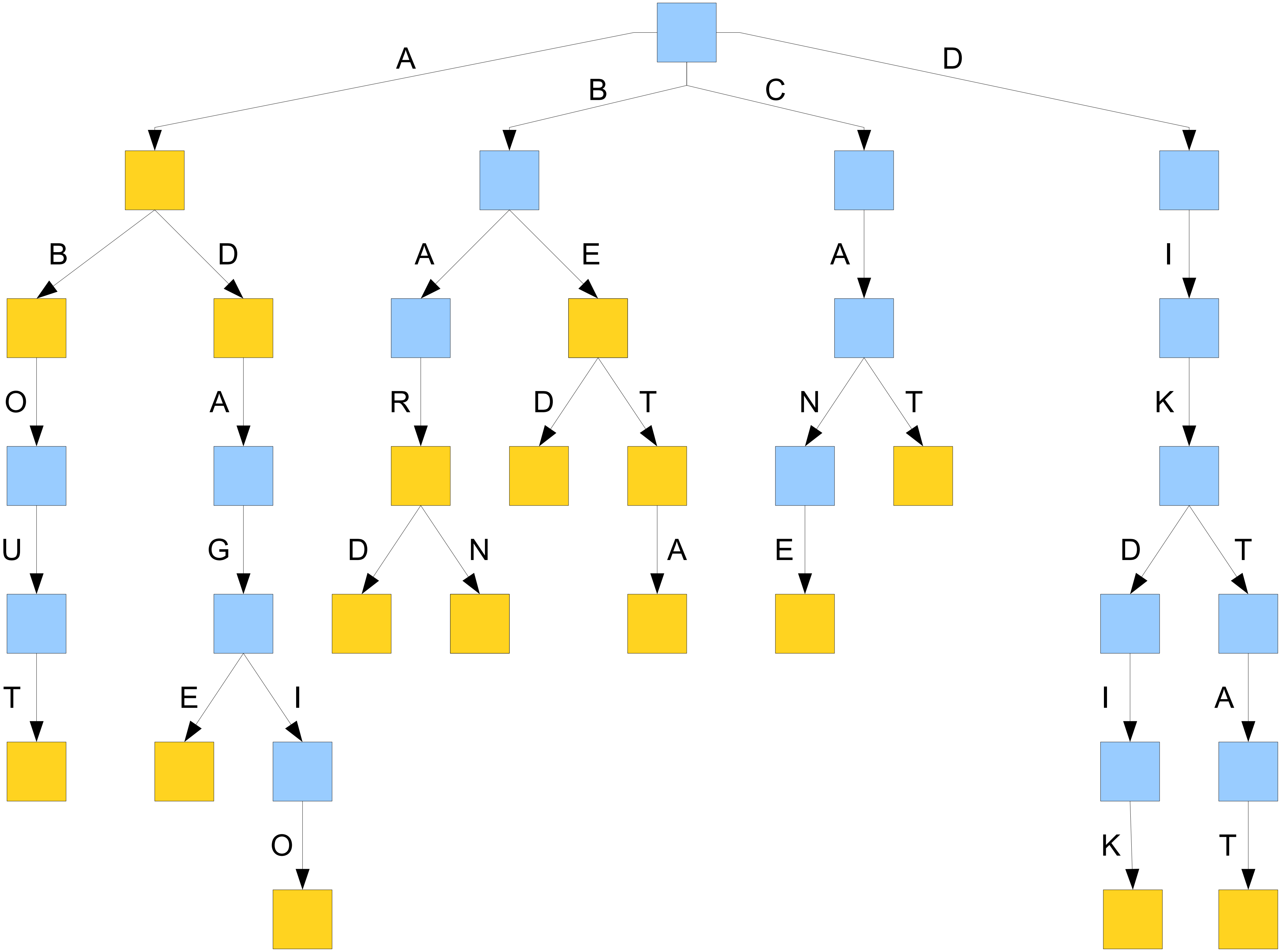
(OurLexicon.cpp/h)

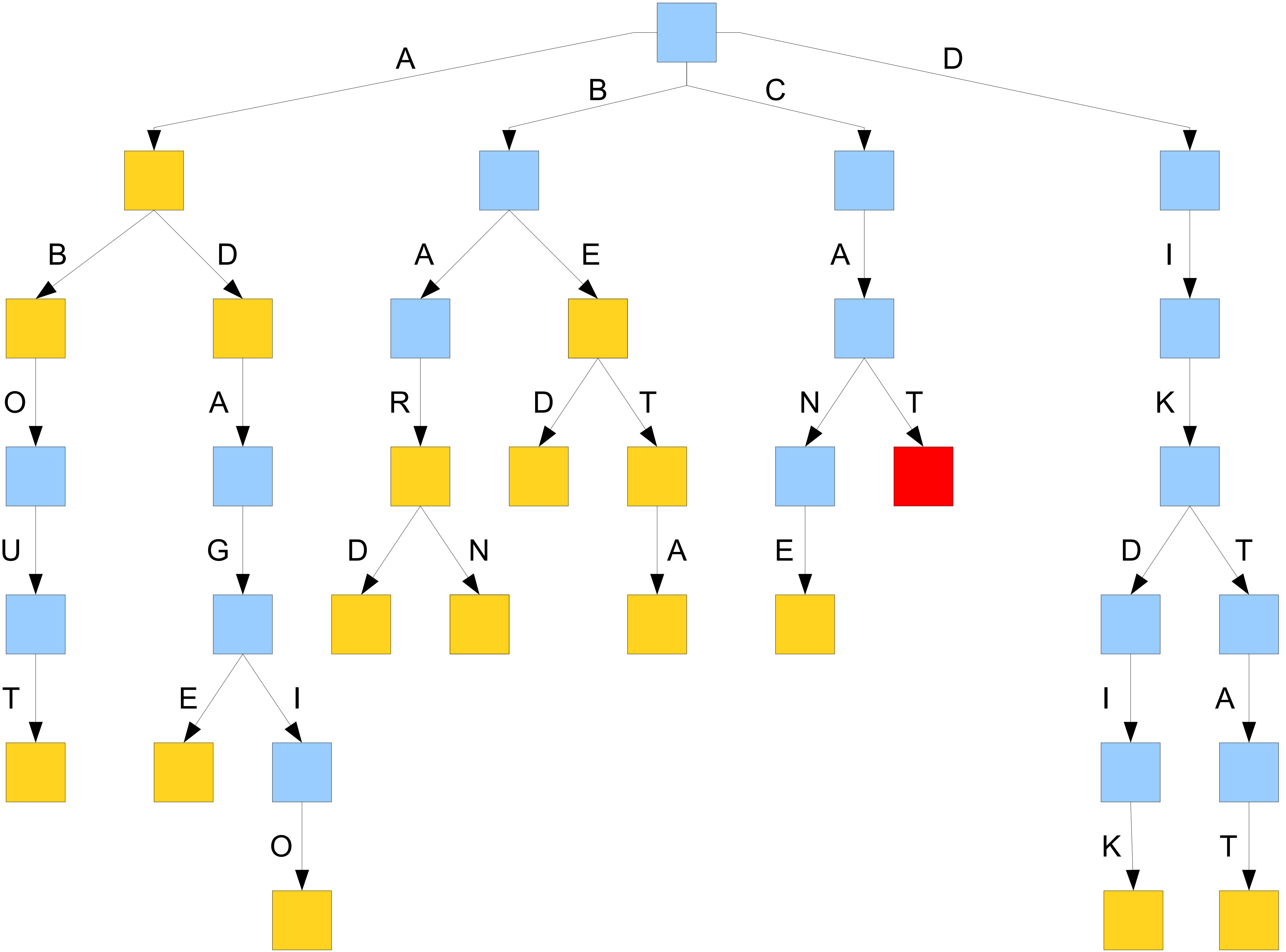
# Analyzing the Trie

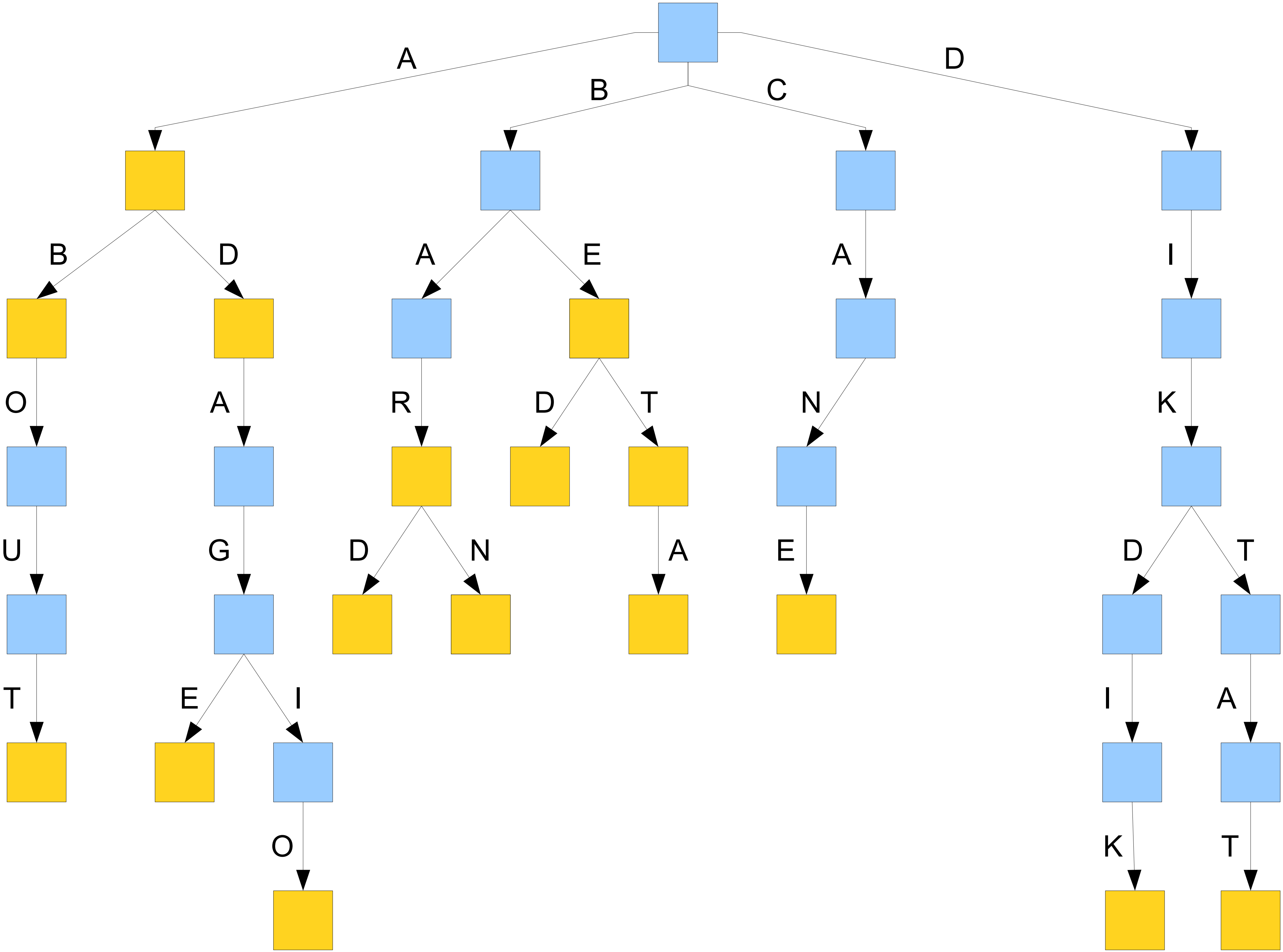
- How efficient are the operations on the trie?
- Every operation takes time proportional to the length of the string, which we'll denote ***L***.
- Time to add or look up an element is **O(*L*)**.
- Time to check if a prefix exists is **O(*L*)**.

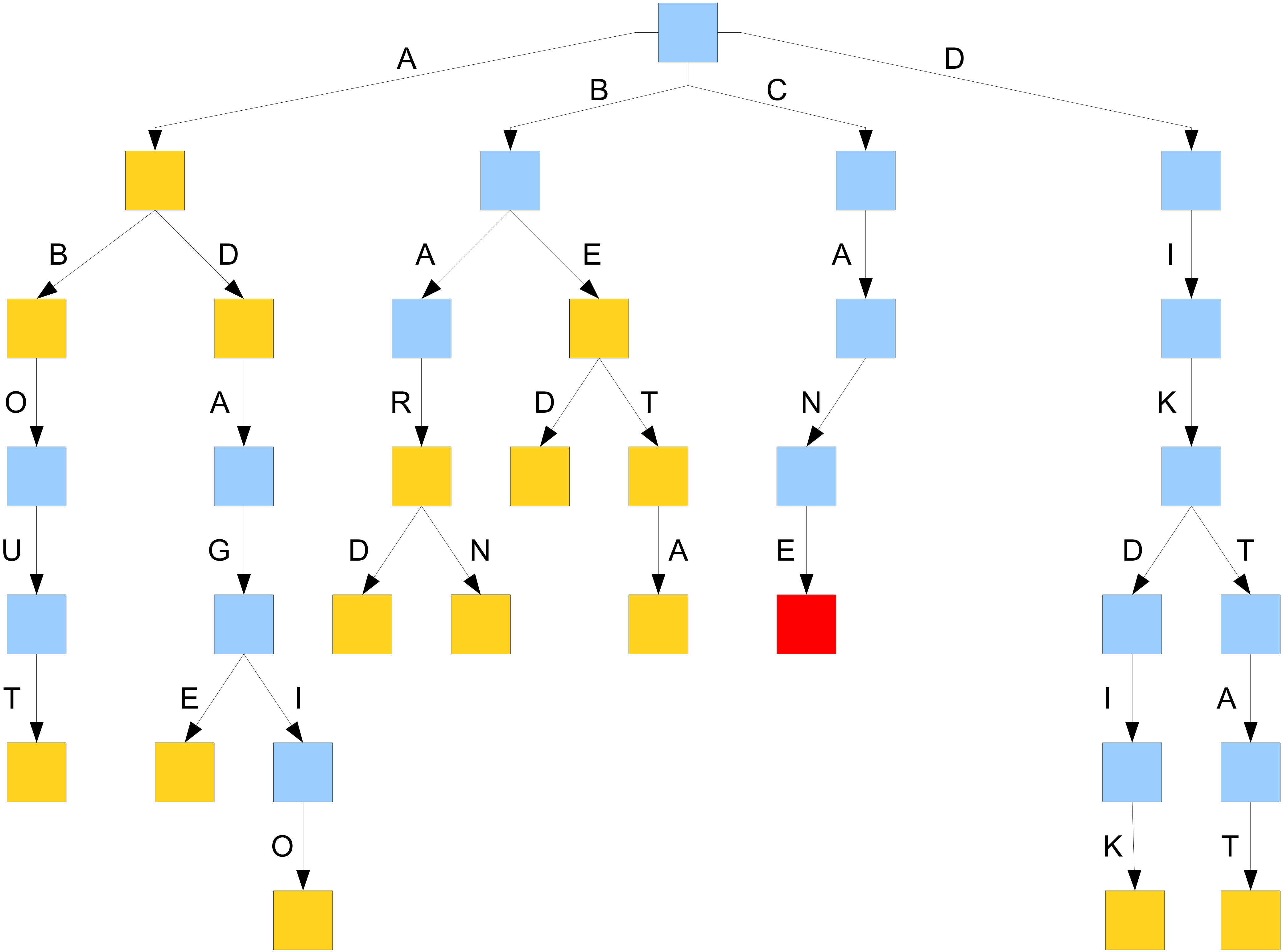

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T

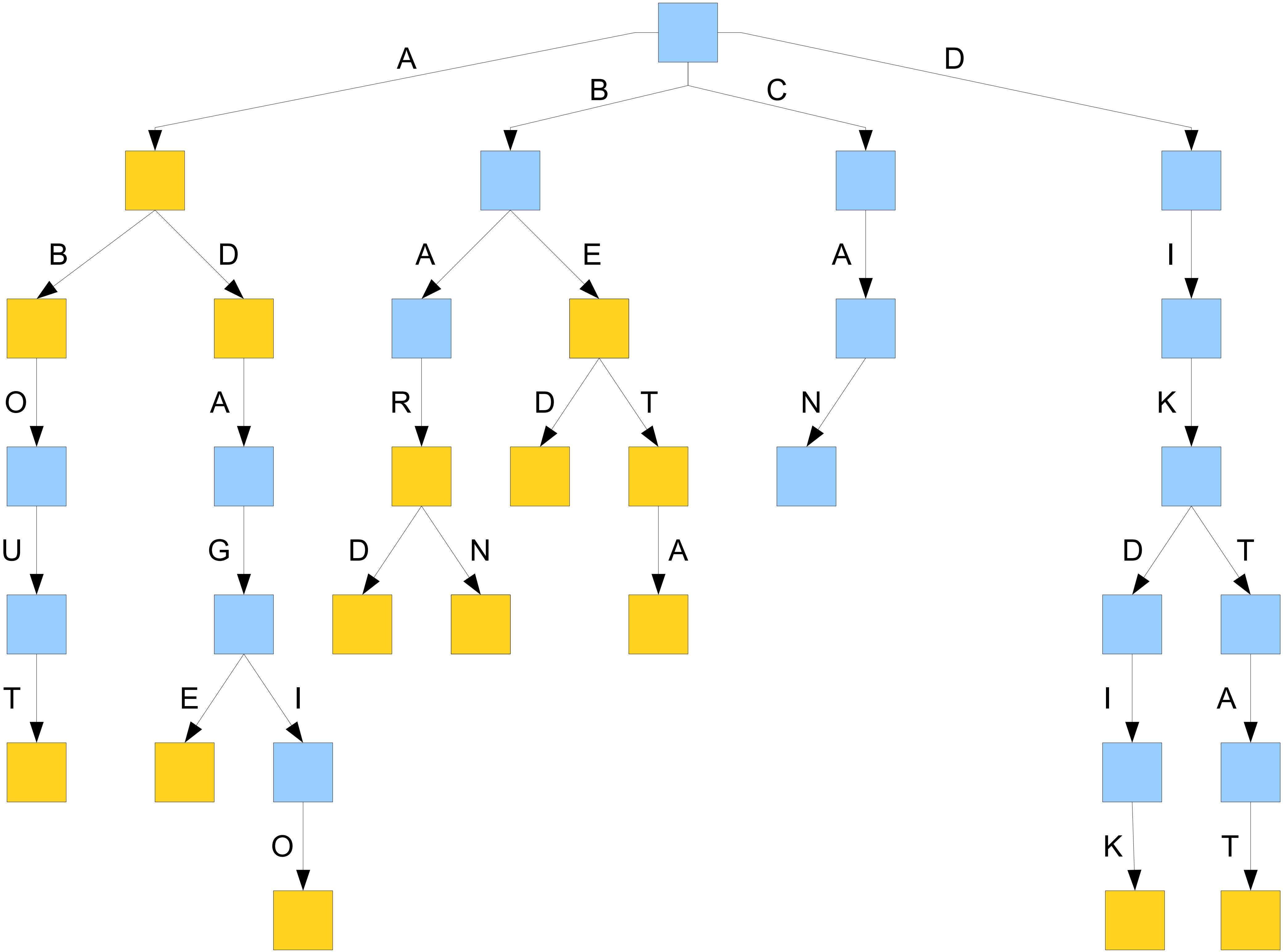

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T

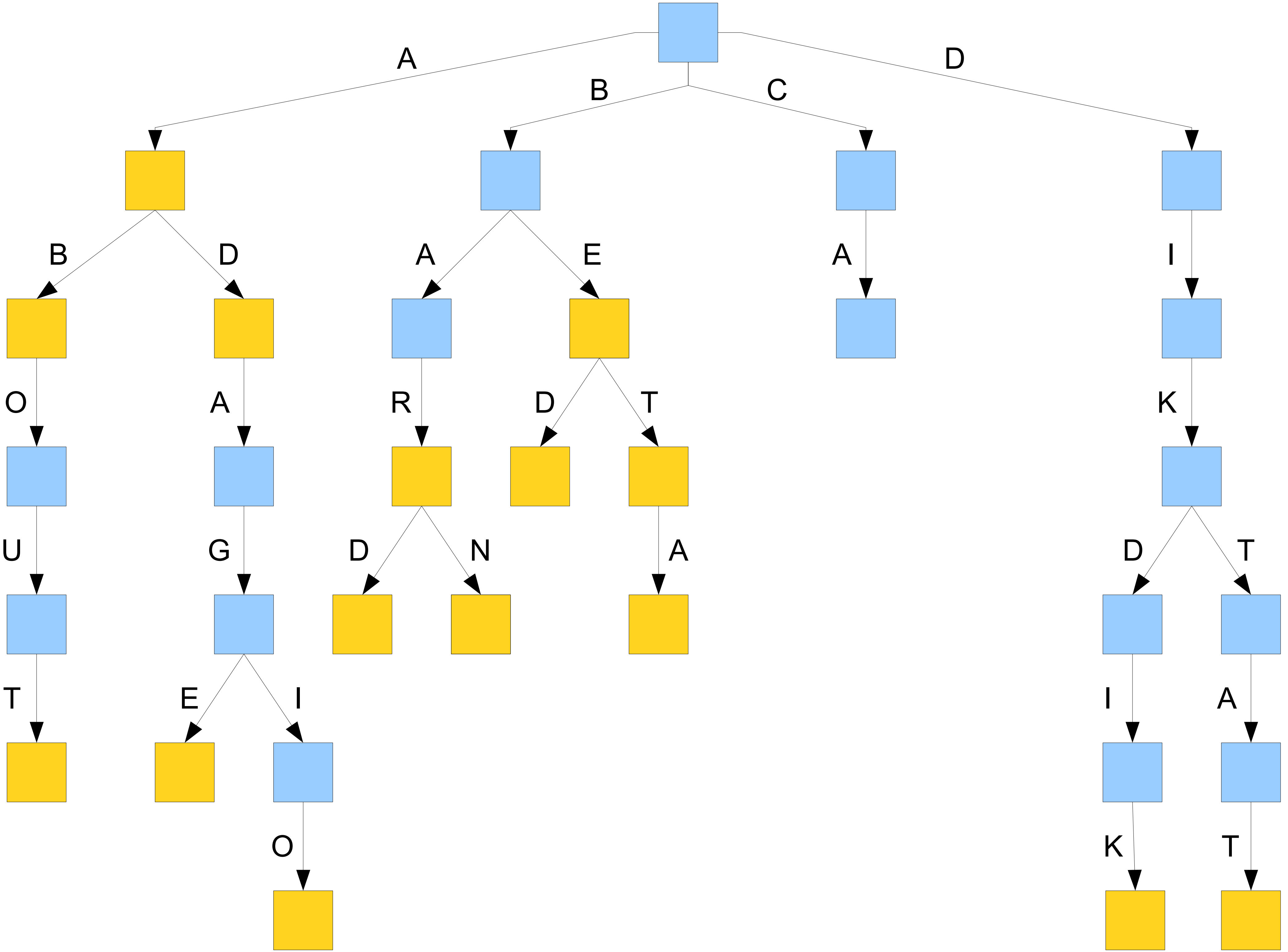

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T

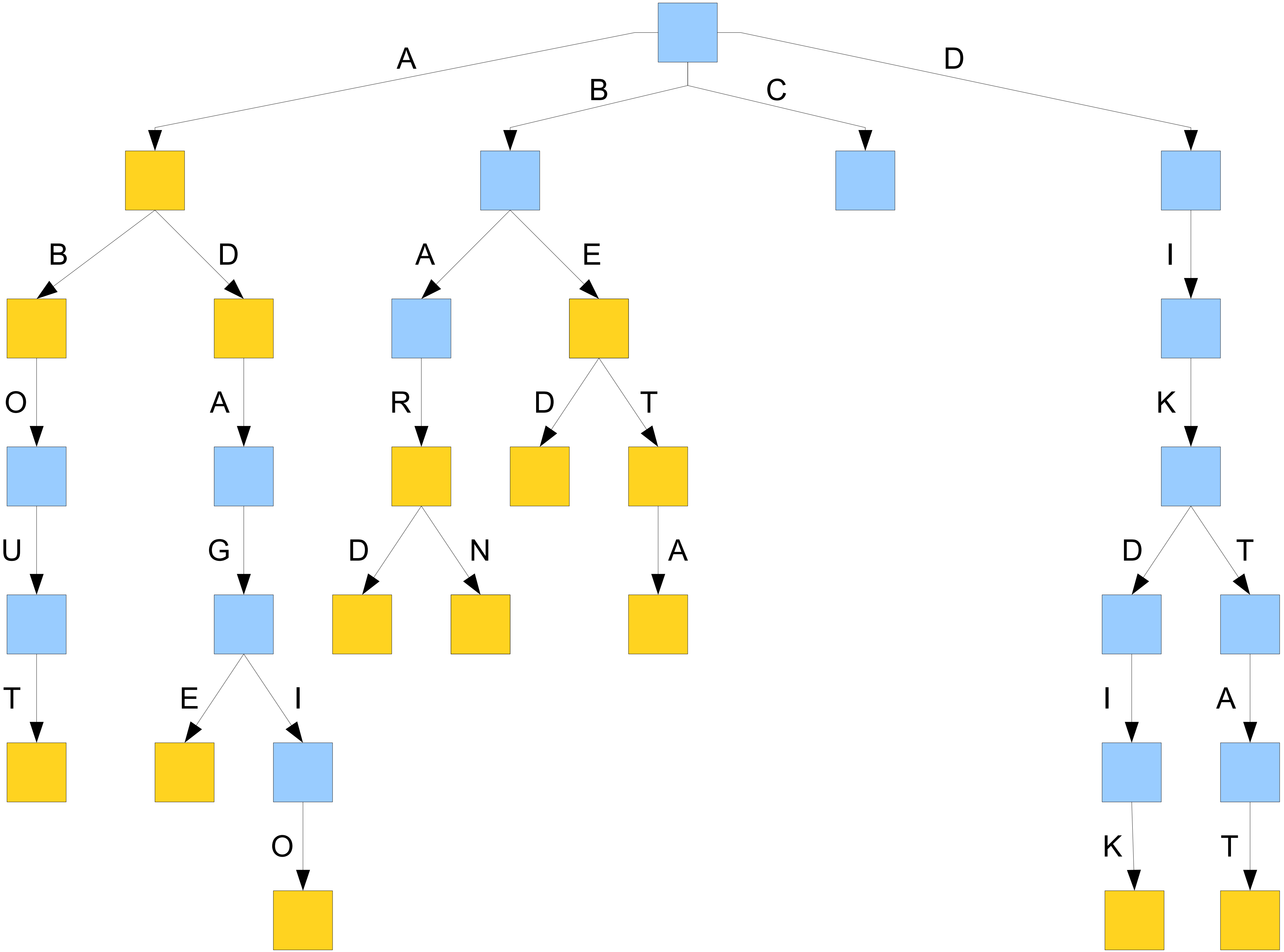

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T

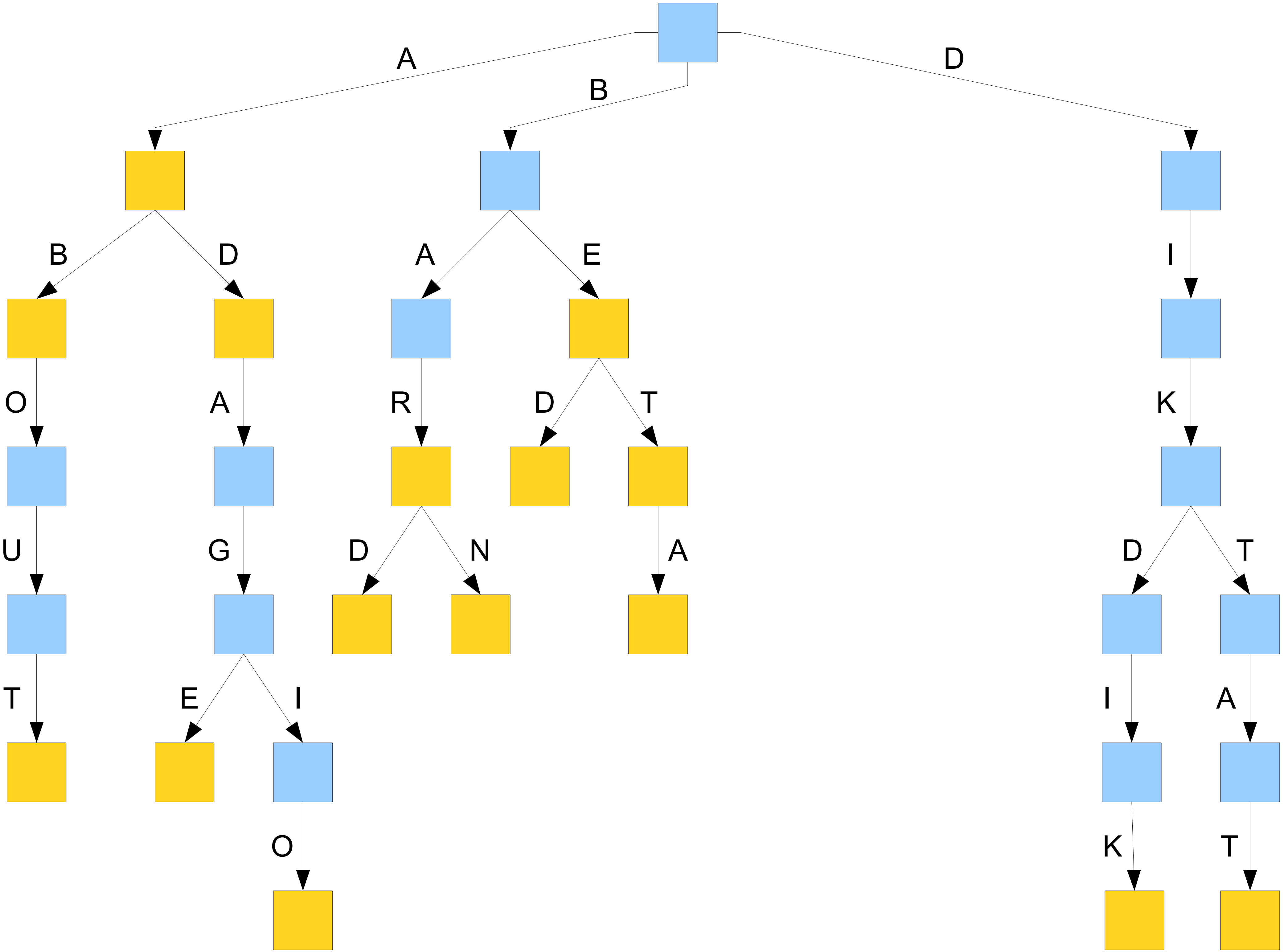

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T


A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T


A
D
B
C
B
D
A
E
A
I
O
A
R
D
T
N
K
U
G
D
N
A
D
T
T
E
I
I
A
O
K
T


A
D
B
C
B
D
A
E
A
I
O
A
R
D
T
K
U
G
D
N
A
D
T
T
E
I
I
A
O
K
T


A
B
C
D
B
D
A
E
I
O
A
R
D
T
K
U
G
D
N
A
D
T
T
E
I
I
A
O
K
T



A
B
D
B
D
A
E
I
O
A
R
D
T
K
U
G
D
N
A
D
T
T
E
I
I
A
O
K
T

# Removing from a Trie

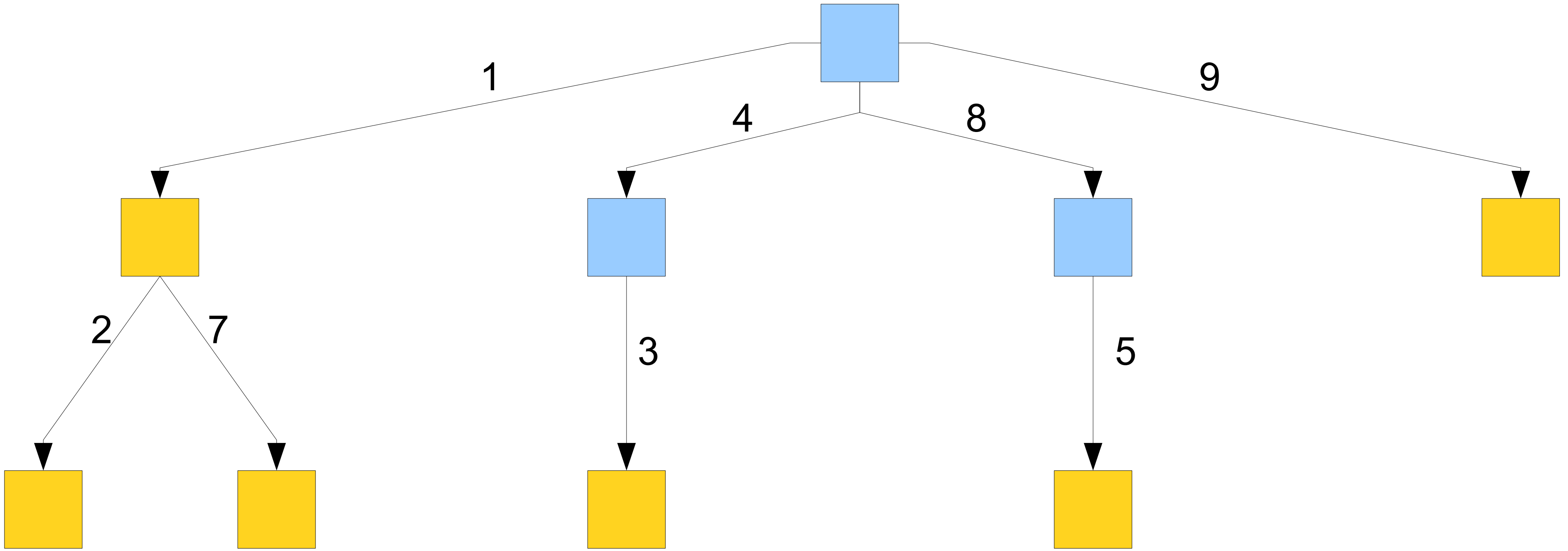
- Recurse until you reach the last node representing the word
- Mark the node as no longer containing a word.
- If the node has no children:
    - Remove that node.
        - **`delete`** and set equal to **`NULL`**
    - Repeat this process at the node one level higher up in the tree.

Let's trie coding up `Lexicon`!
(remove)
(OurLexicon.cpp/h)

# Other uses of Tries

- The Trie we wrote stores strings.  Could we use a Trie to store other data types as well?
  - Yes!

# Integer Trie



# Other uses of Tries

- We can also generalize Tries to any data type by branch on the binary representation of a piece of data
  - This is called a “Bitwise Trie”

# Tries: Pros and Cons

- Pros:

  1) **`containsPrefix()`** runs in **$O(L)$** time

  2) **`contains()`** runs in **$O(L)$** time

  3) Memory advantages by taking advantage of redundant prefixes.

     – e.g. All words that start with 'A' *share* a node representing the prefix “A”

# Tries: Pros and Cons

- Cons:
  - 1) Without redundant prefixes, Tries are super memory inefficient.
  - 2) Without some cool compression techniques that we'll go over in a couple lectures, Tries eat up a lot of memory.
    - It turns out, that these optimizations work best **when we don't modify the set of elements we are storing!**

# When do we use Tries?

- We generally use Tries when the following two properties hold:
  - Property 1: Lots of redundant prefixes
  - Property 2: The set of elements is *static* (the set of elements we want to store doesn't change over time)
- These two properties hold for languages

# Next Week

- **Graphs!**
  - How we represent and work with data with relationships (e.g. map data)