Tries and String Matching

Slide Courtesy : Keith Schwarz, Stanford Univ.
http://web.stanford.edu/class/cs166/

Text Processing

- String processing shows up everywhere:
  - Computational biology: Manipulating DNA sequences.
  - NLP: Storing and organizing huge text databases.
  - Computer security: Building antivirus databases.
- Many problems have polynomial-time solutions.
- **Goal:** Design theoretically and practically efficient algorithms that outperform brute-force approaches.

Outline for Today

- **Tries**
  - A fundamental building block in string processing algorithms.
- **Aho-Corasick String Matching**
  - A fast and elegant algorithm for searching large texts for known substrings.

Tries

Ordered Dictionaries

- Suppose we want to store a set of elements supporting the following operations:
  - Insertion of new elements.
  - Deletion of old elements.
  - Membership queries.
  - Successor queries.
  - Predecessor queries.
  - Min/max queries.
- Can use a standard red/black tree or splay tree to get (worst-case or expected) O(log n) implementations of each.

A Catch

- Suppose we want to store a set of strings.
- Comparing two strings of lengths r and s takes time $O(\min\{r, s\})$.
- Operations on a balanced BST or splay tree now take time $O(M \log n)$, where M is the length of the longest string in the tree.
- Can we do better?

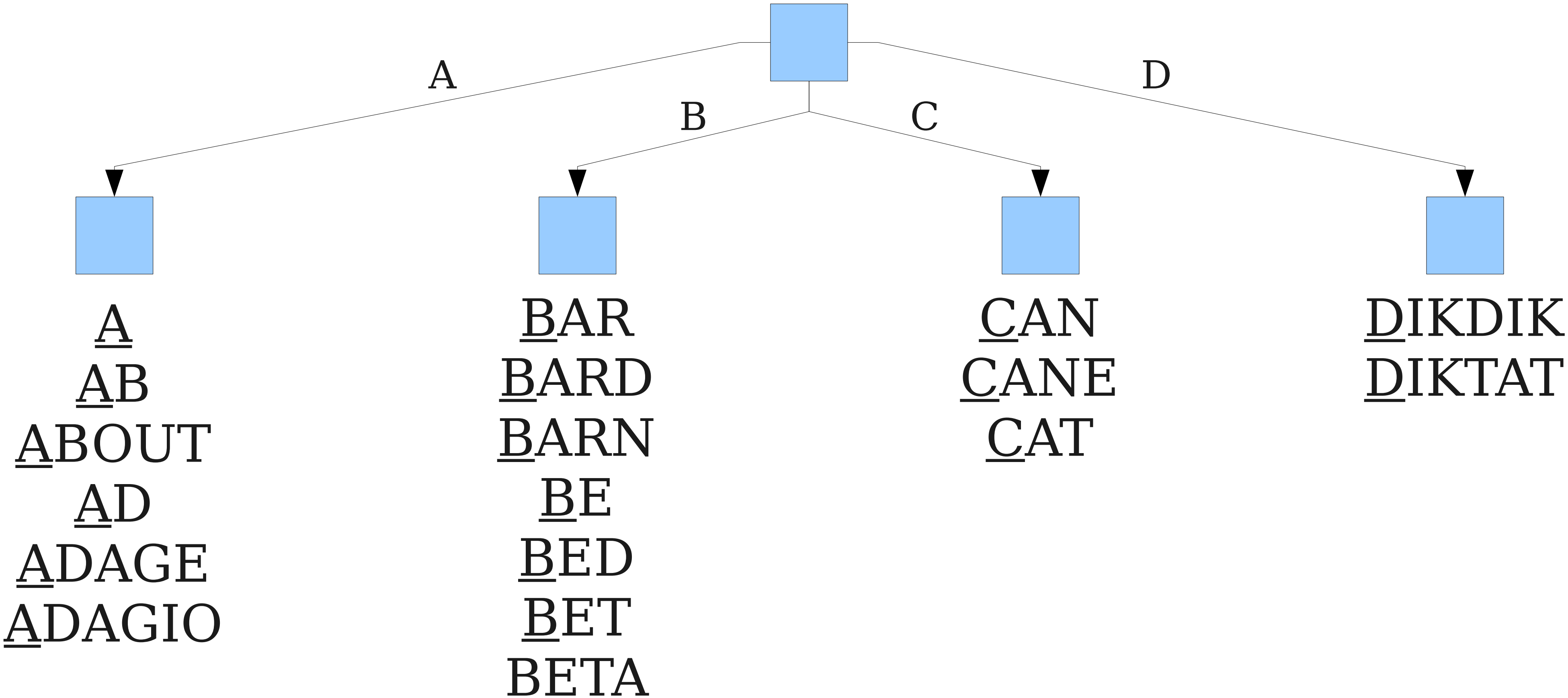
A
AB
ABOUT
AD
ADAGE
ADAGIO
BAR
BARD
BARN
BE
BED
BET
BETA
CAN
CANE
CAT
DIKDIK
DIKTAT

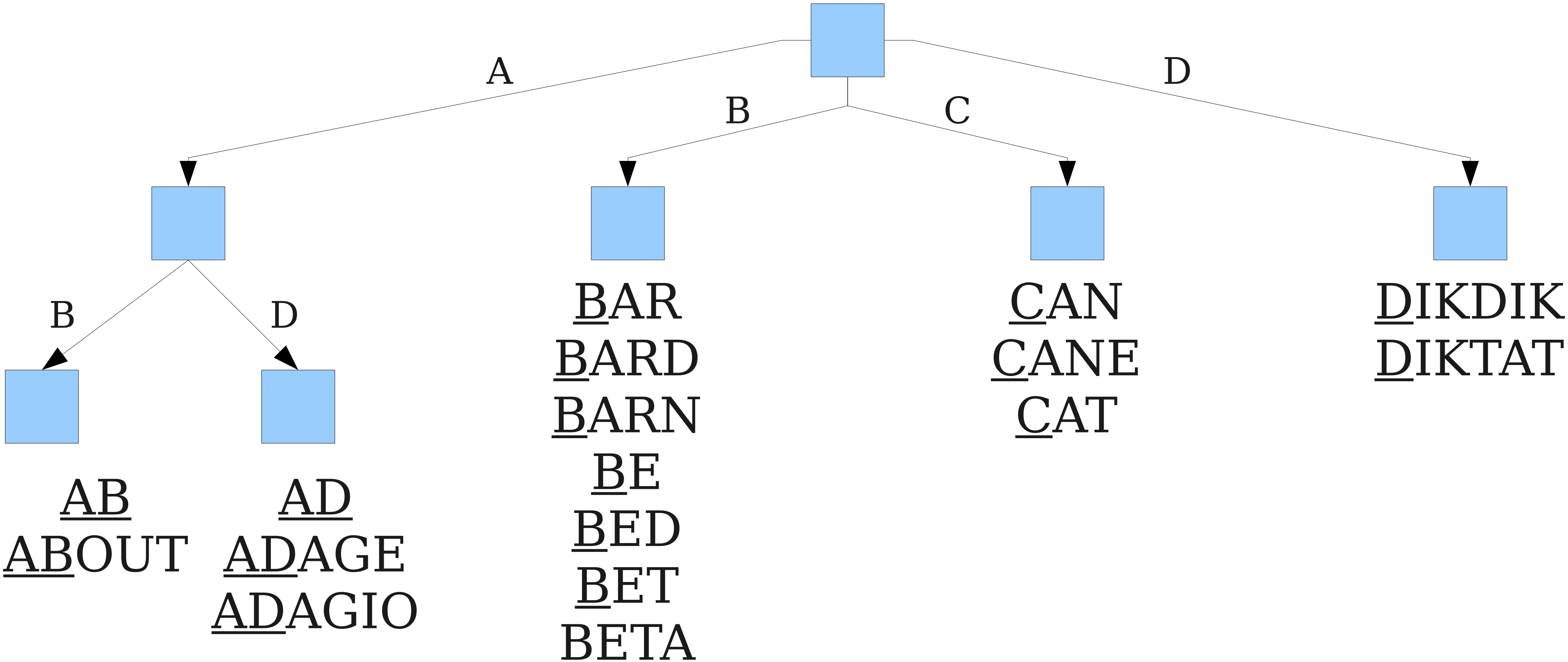
A
AB
ABOUT
AD
ADAGE
ADAGIO

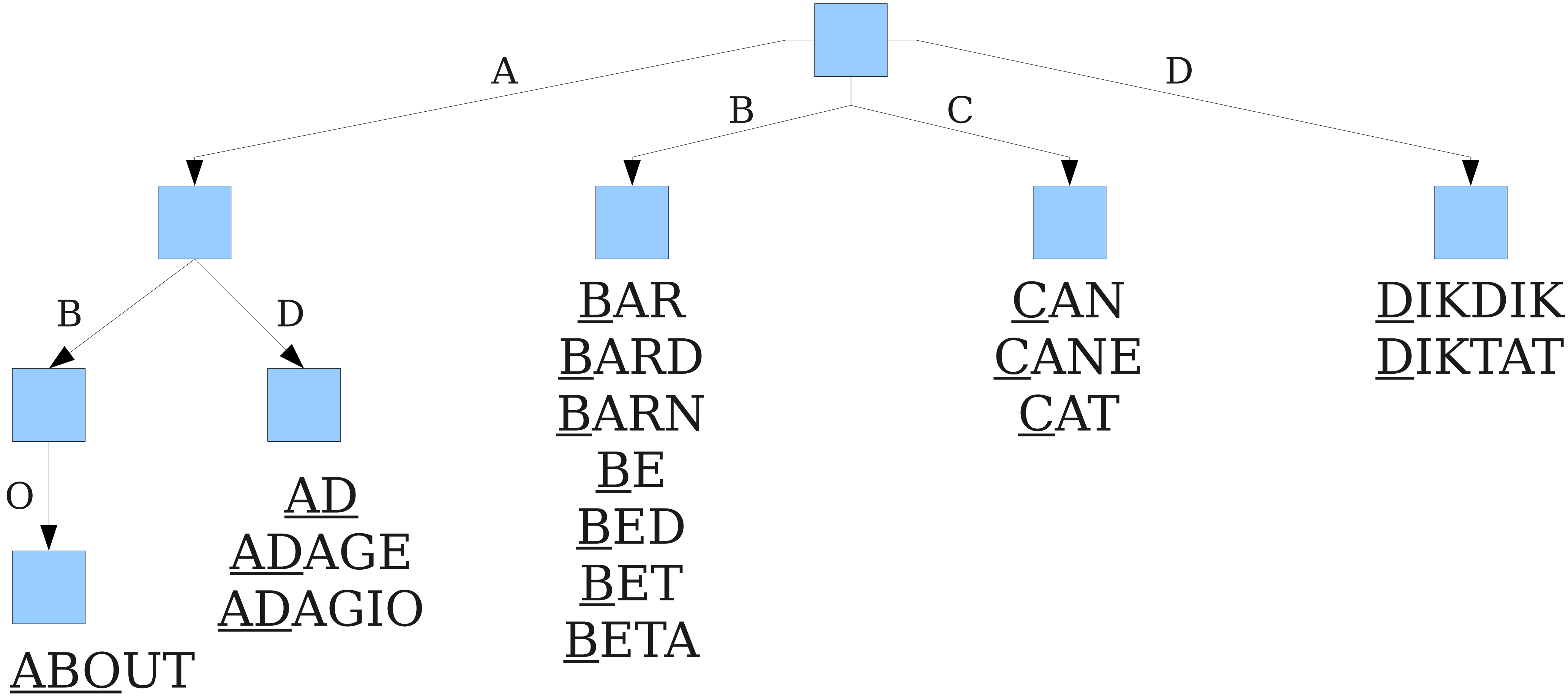
BAR
BARD
BARN
BE
BED
BET
BETA

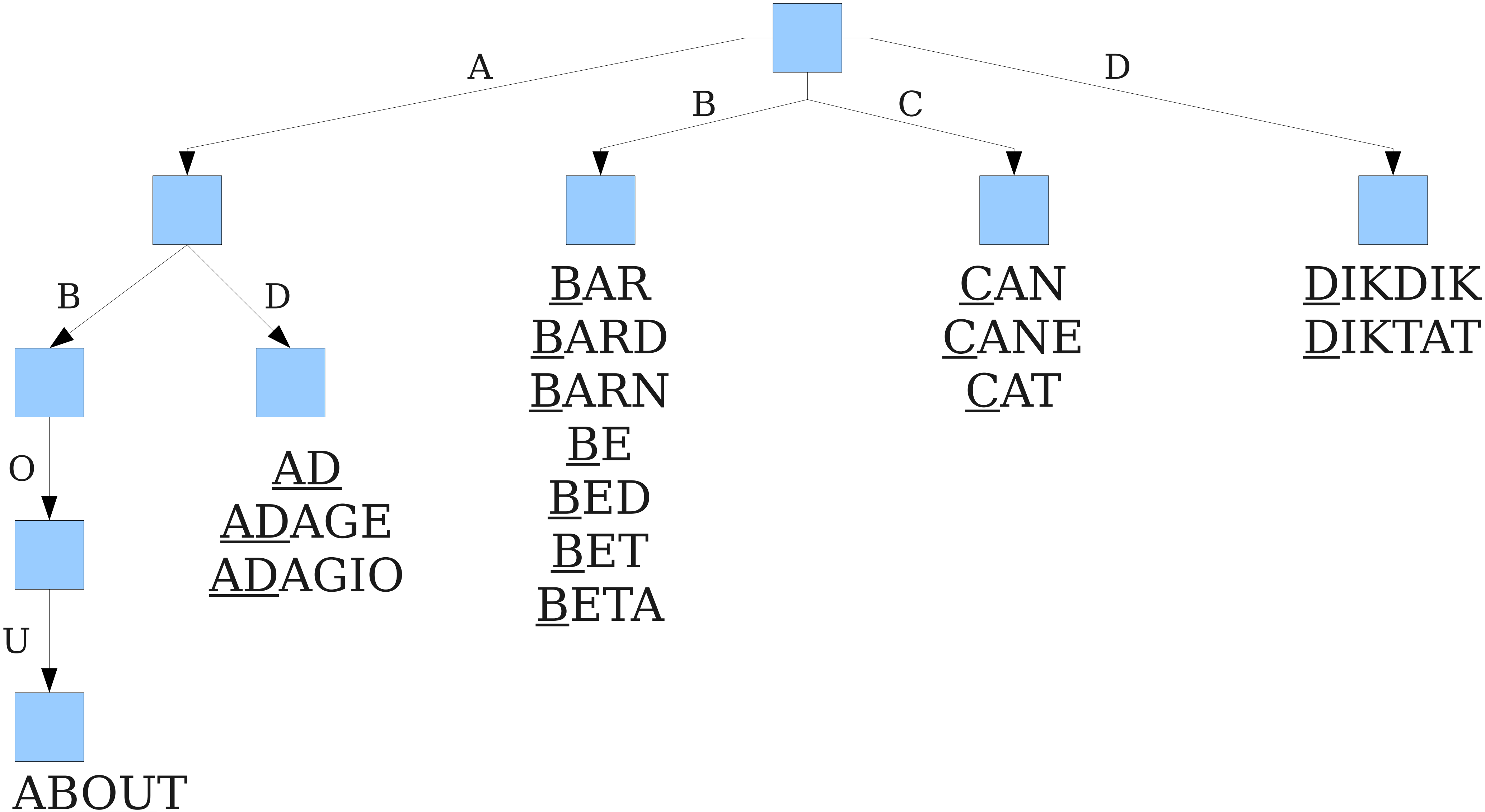
CAN
CANE
CAT

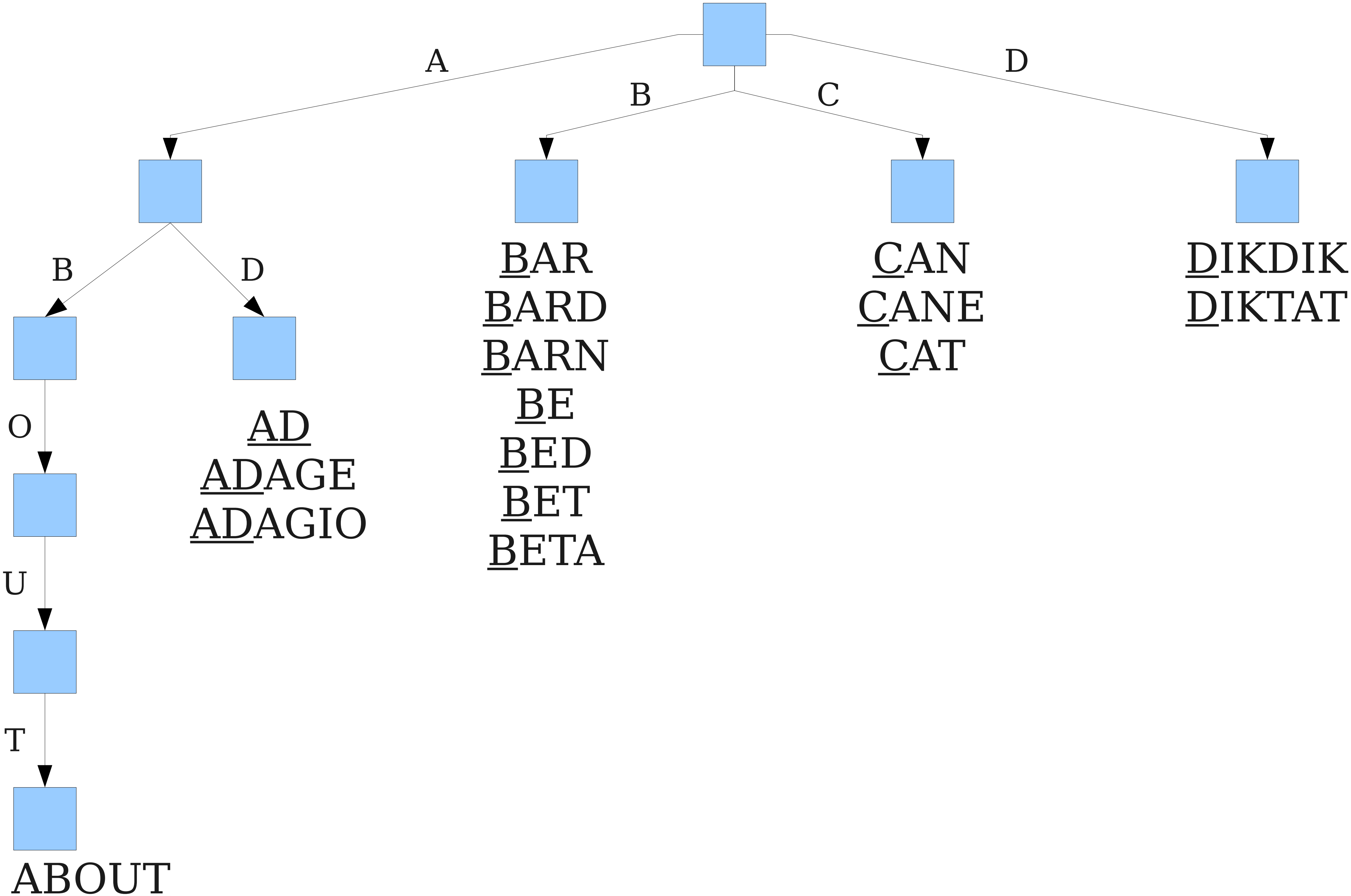
DIKDIK
DIKTAT

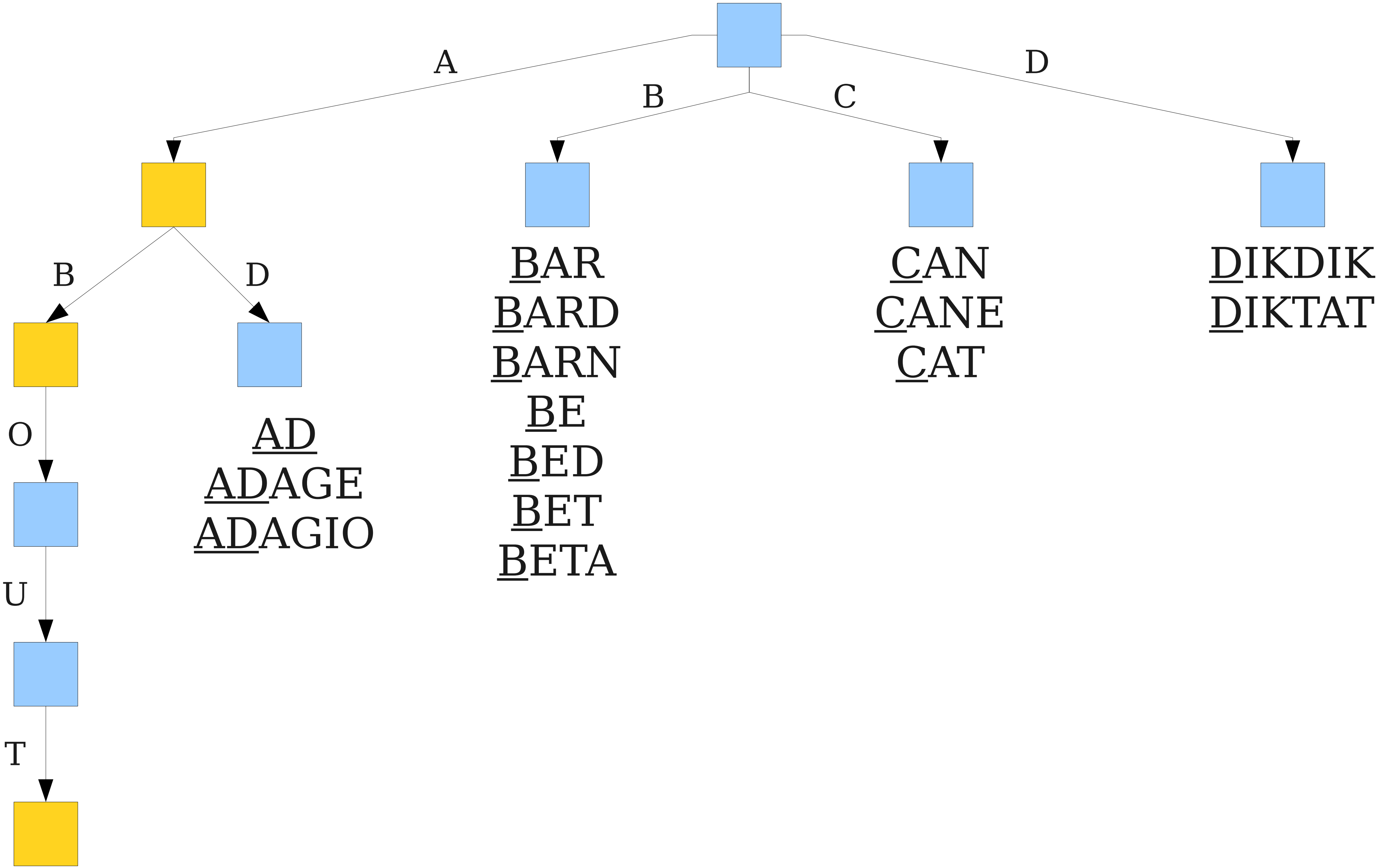

A
B
C
D
A
AB
ABOUT
AD
ADAGE
ADAGIO
BAR
BARD
BARN
BE
BED
BET
BETA
CAN
CANE
CAT
DIKDIK
DIKTAT

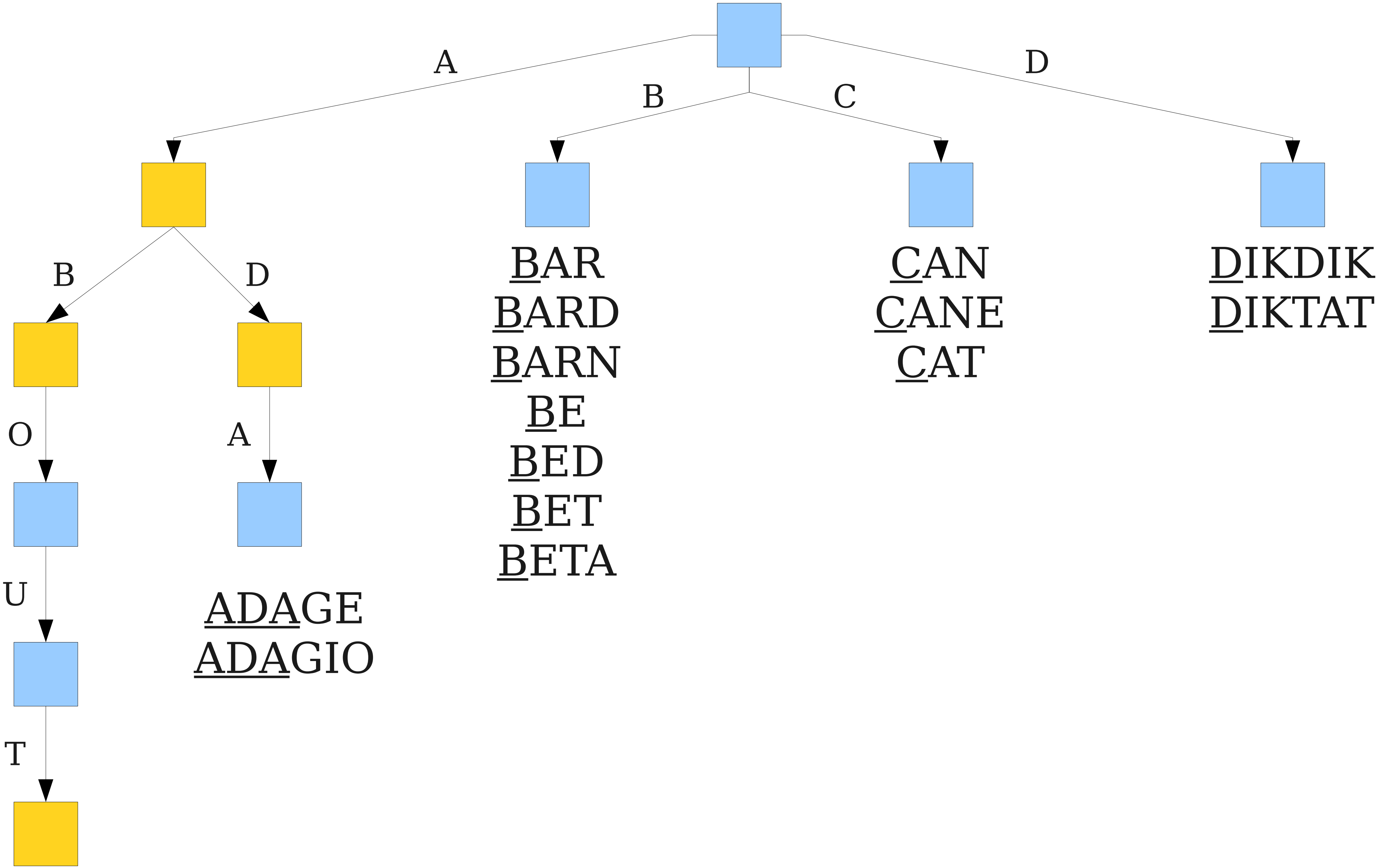

A
B
C
D
B
D
BAR
BARD
BARN
BE
BED
BET
BETA
CAN
CANE
CAT
DIKDIK
DIKTAT
AB
ABOUT
AD
ADAGE
ADAGIO

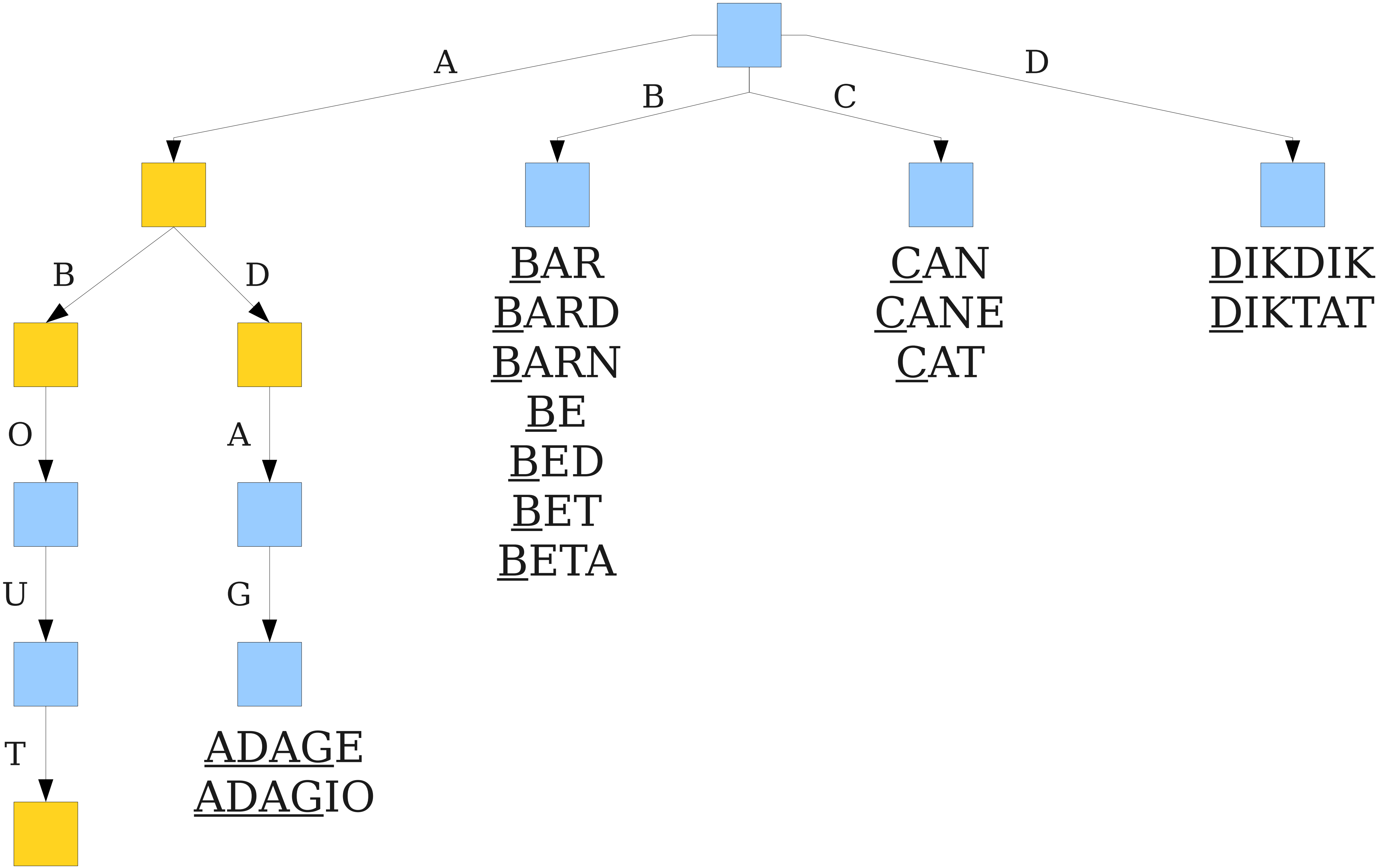

A
B
C
D
B
D
O
BAR
BARD
BARN
BE
BED
BET
BETA
CAN
CANE
CAT
DIKDIK
DIKTAT
AD
ADAGE
ADAGIO
ABOUT

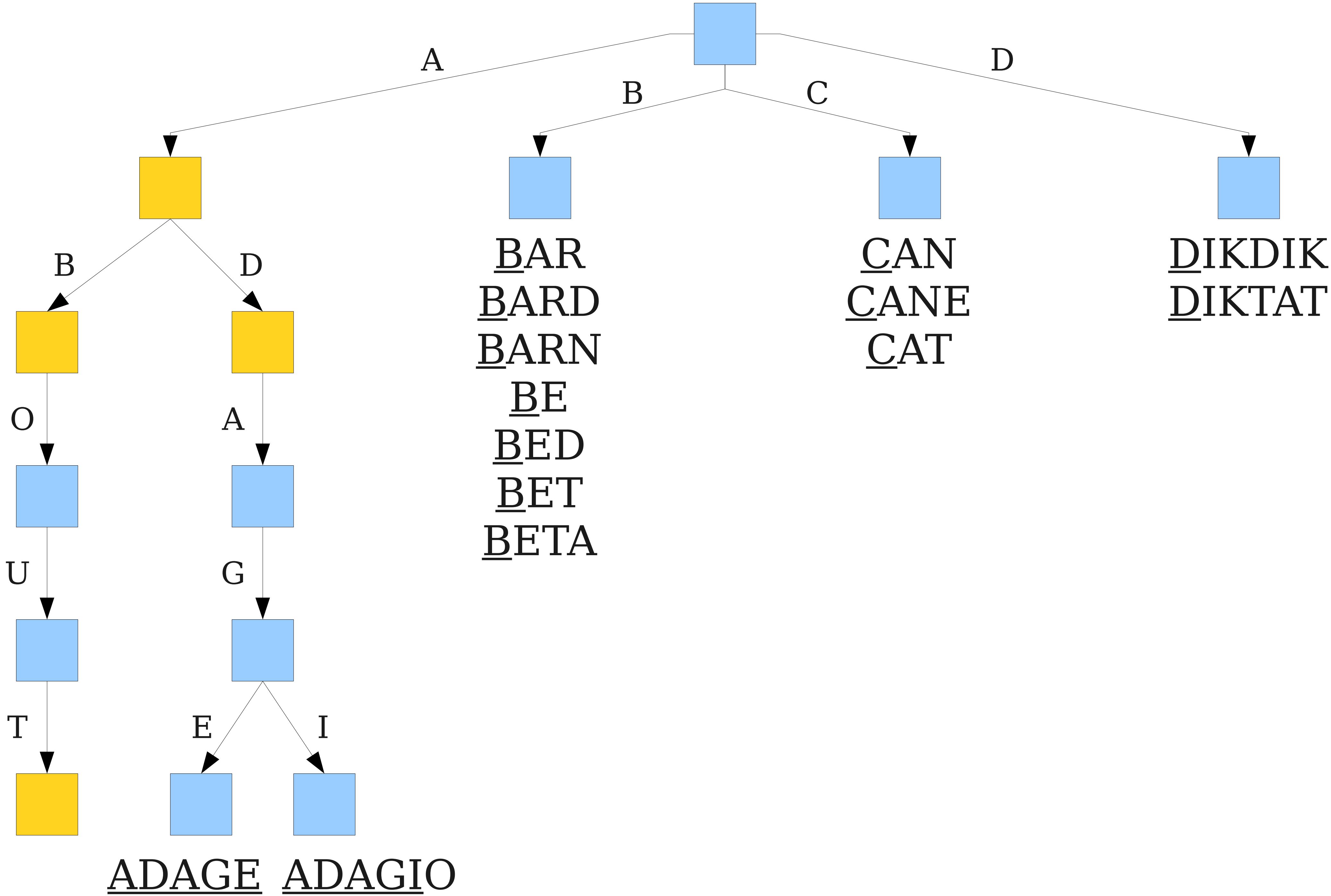

A
B
C
D
B
D
O
U
BAR
BARD
BARN
BE
BED
BET
BETA
CAN
CANE
CAT
DIKDIK
DIKTAT
AD
ADAGE
ADAGIO
ABOUT

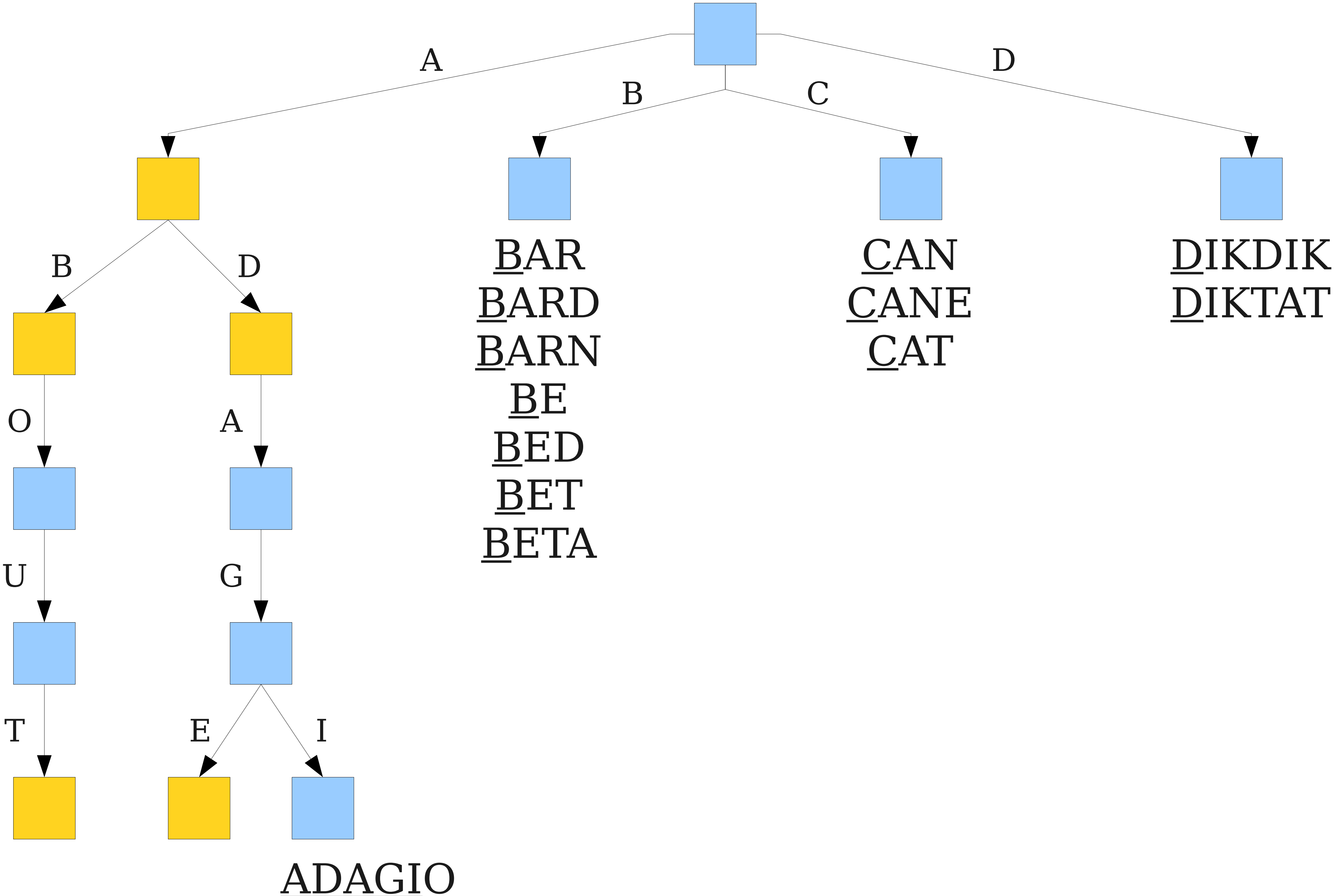

A
B
C
D
B
D
BAR
BARD
BARN
BE
BED
BET
BETA
CAN
CANE
CAT
DIKDIK
DIKTAT
O
AD
ADAGE
ADAGIO
U
T
ABOUT

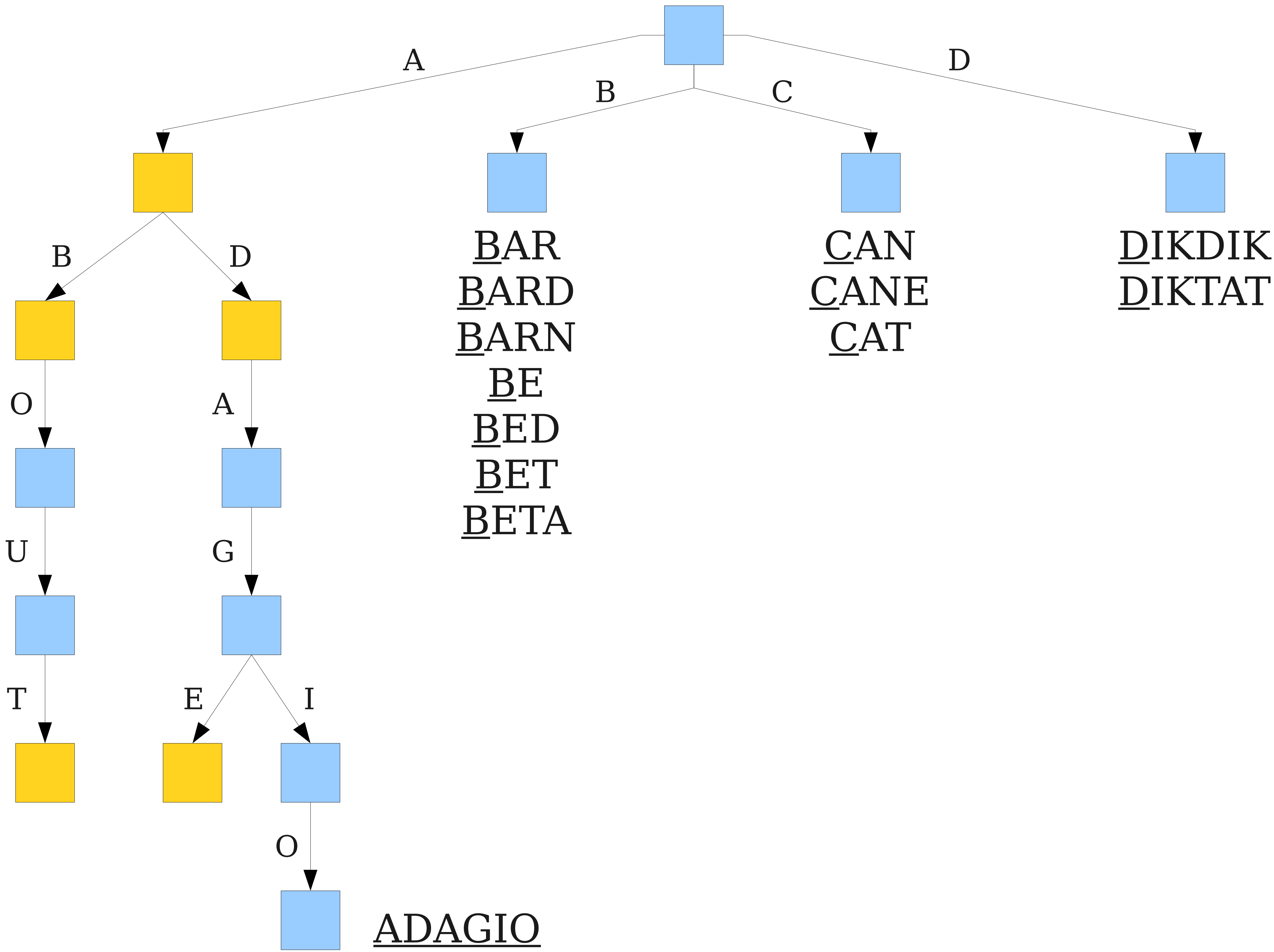

A
B
C
D
B
D
O
U
T
BAR
BARD
BARN
BE
BED
BET
BETA
CAN
CANE
CAT
DIKDIK
DIKTAT
AD
ADAGE
ADAGIO

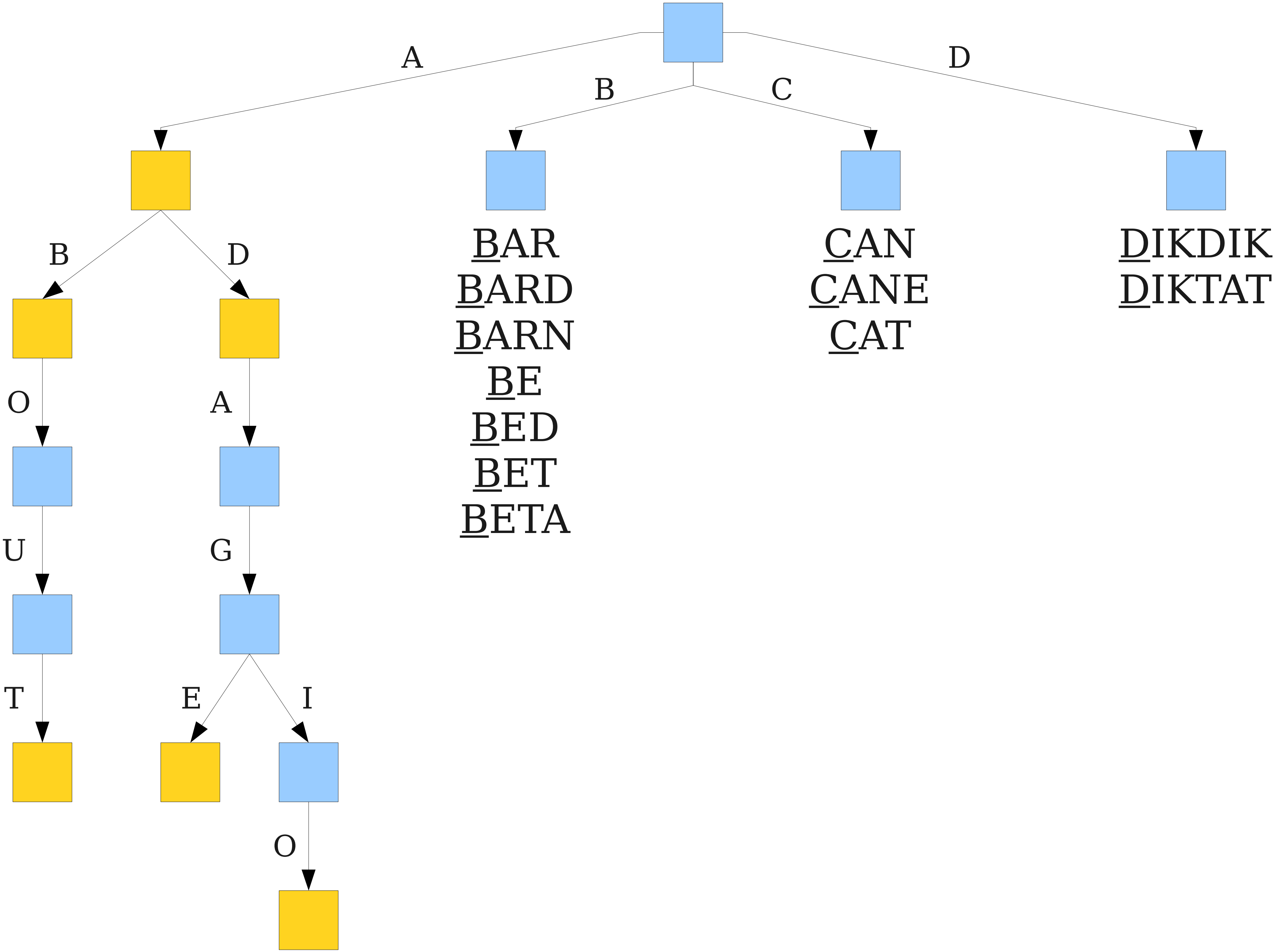

A
B
C
D
B
D
O
A
U
T
BAR
BARD
BARN
BE
BED
BET
BETA
CAN
CANE
CAT
DIKDIK
DIKTAT
ADAGE
ADAGIO

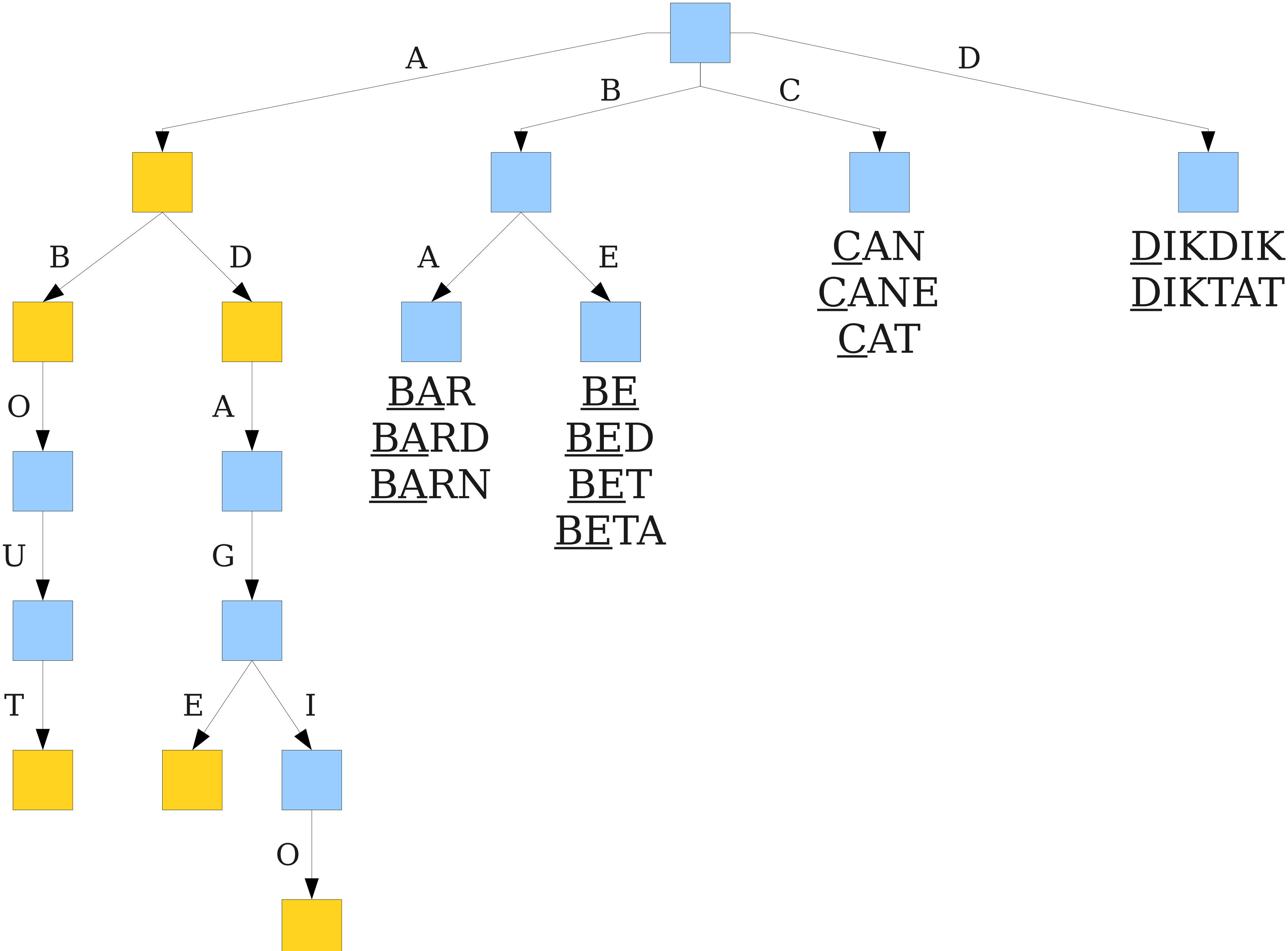

A
B
C
D
B
D
BAR
BARD
BARN
BE
BED
BET
BETA
CAN
CANE
CAT
DIKDIK
DIKTAT
O
A
U
G
T
ADAGE
ADAGIO

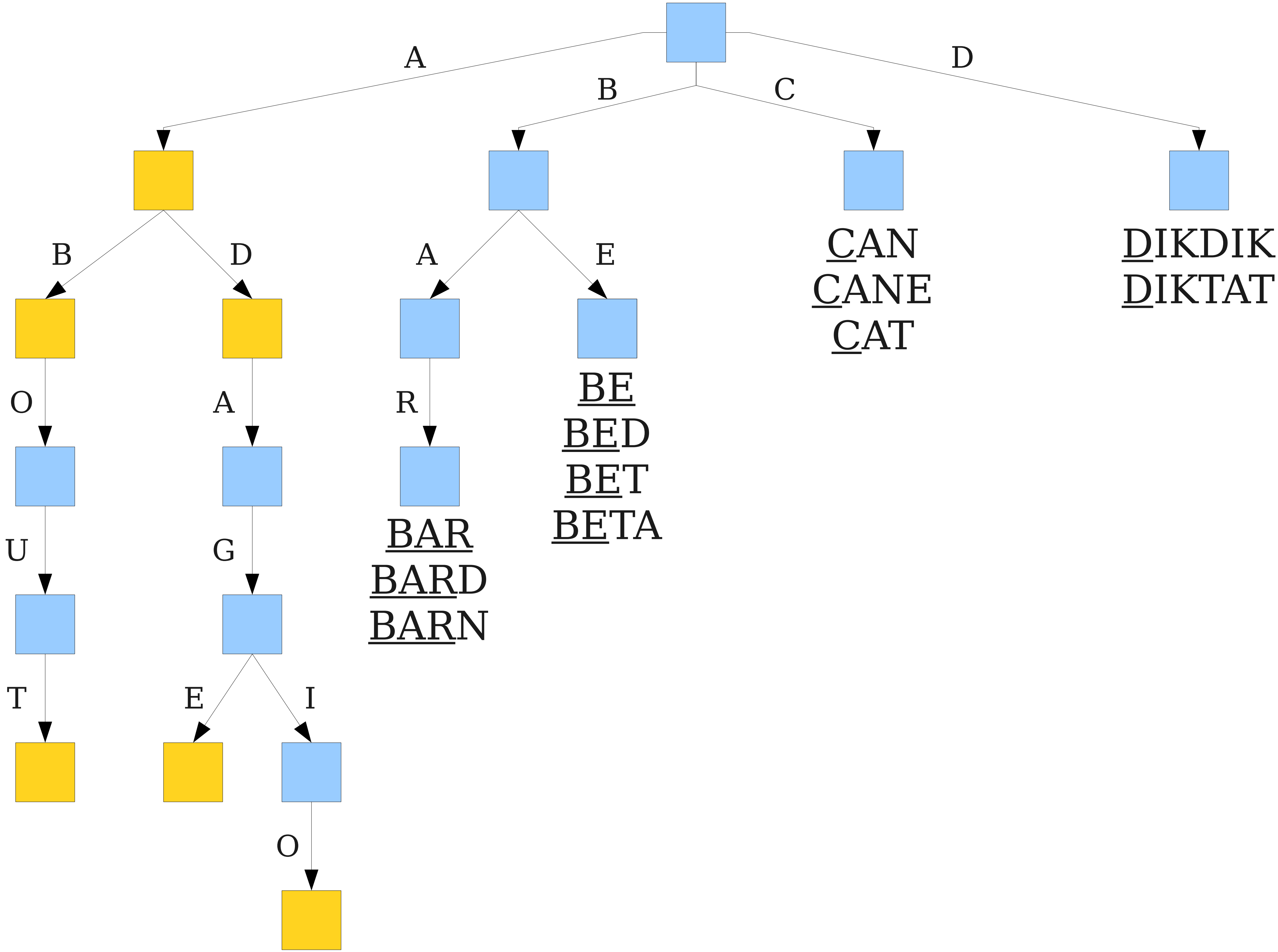

A
B
C
D
B
D
O
A
U
G
T
E
I
BAR
BARD
BARN
BE
BED
BET
BETA
CAN
CANE
CAT
DIKDIK
DIKTAT
ADAGE
ADAGIO

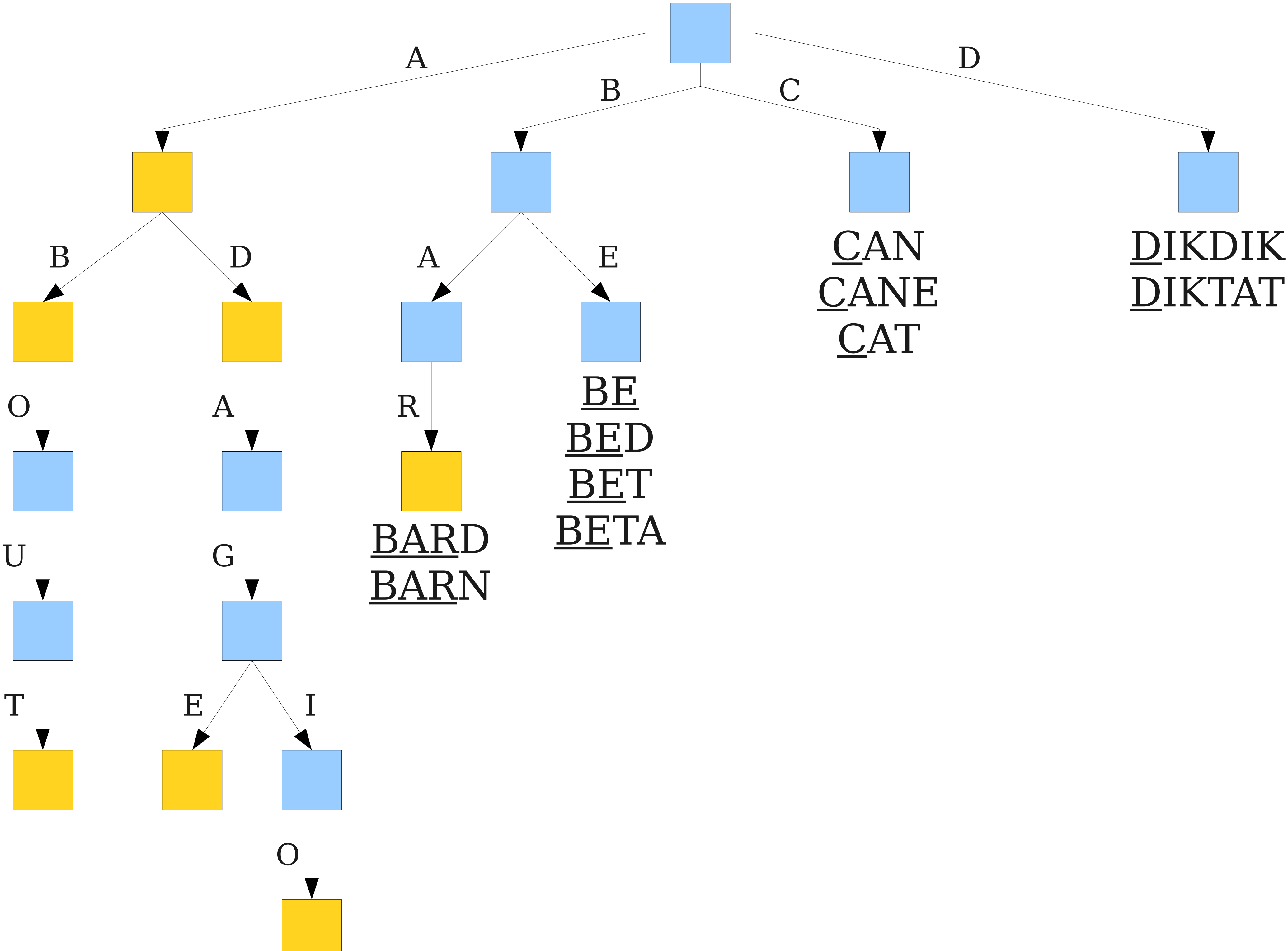

A
B
C
D
B
D
BAR
BARD
BARN
BE
BED
BET
BETA
CAN
CANE
CAT
DIKDIK
DIKTAT
O
A
U
G
T
E
I
ADAGIO

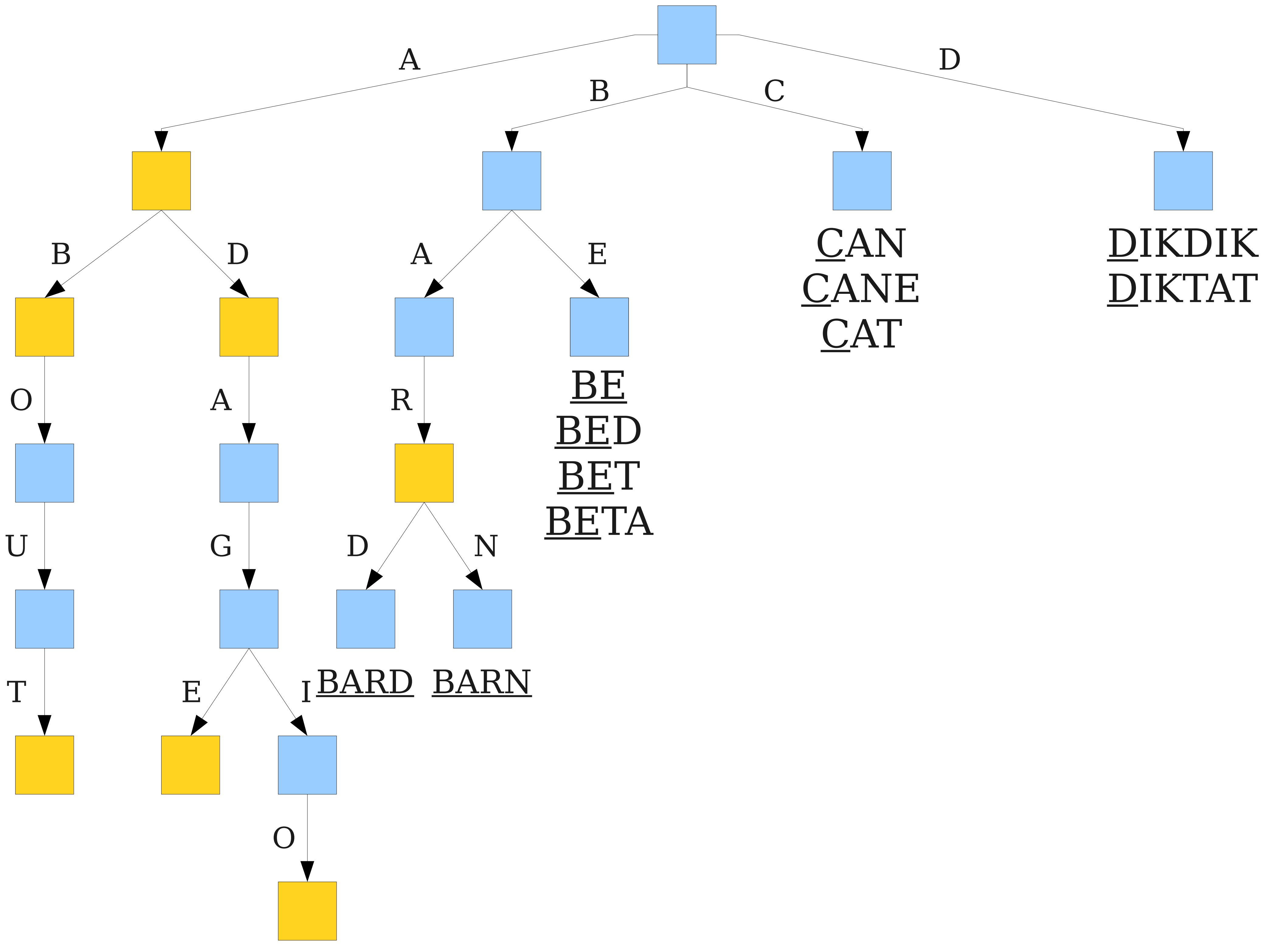

A
B
C
D
B
D
O
A
U
G
T
E
I
O
BAR
BARD
BARN
BE
BED
BET
BETA
CAN
CANE
CAT
DIKDIK
DIKTAT
ADAGIO

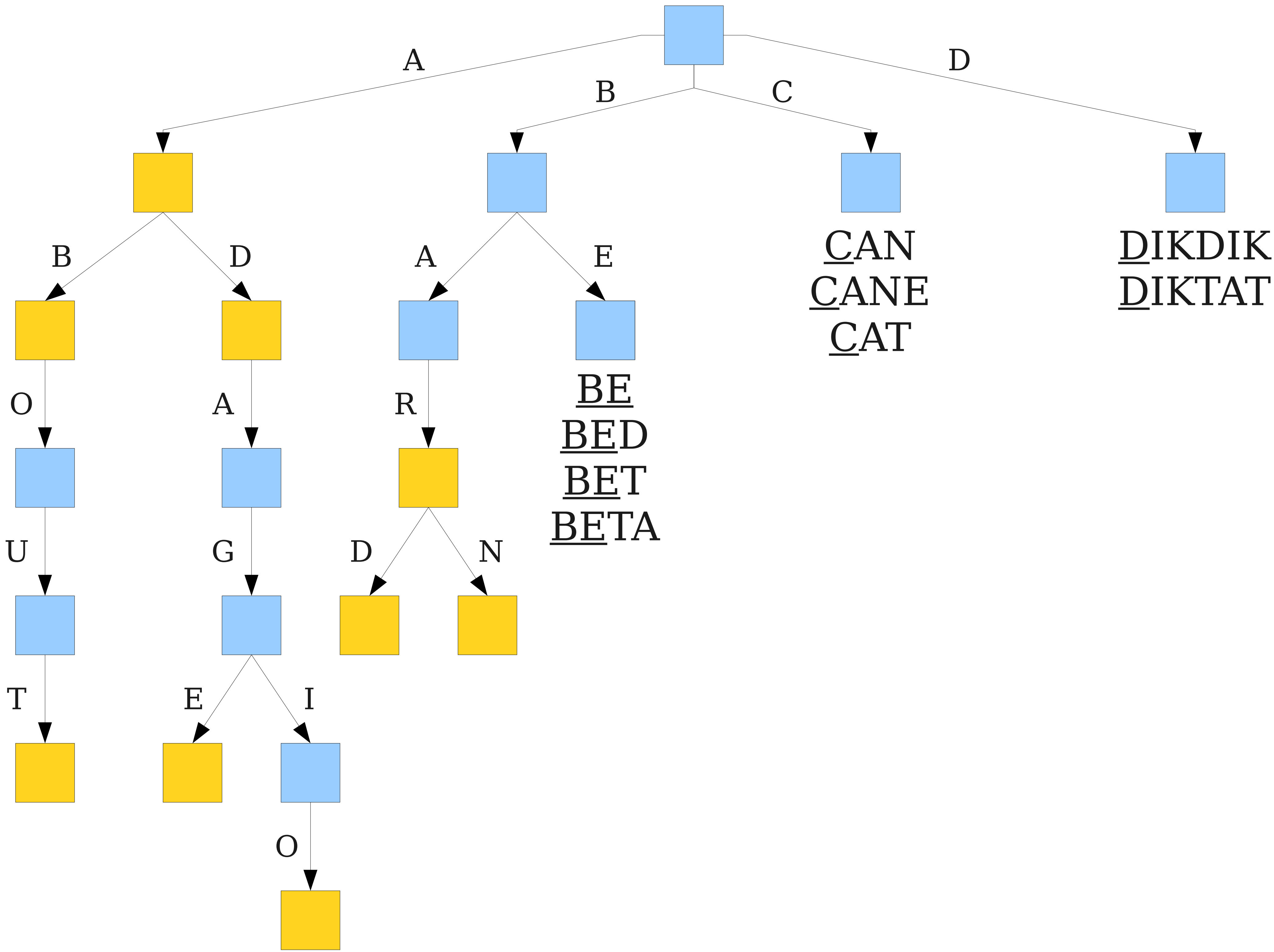

A
B
C
D
B
D
O
A
U
G
T
E
I
O
BAR
BARD
BARN
BE
BED
BET
BETA
CAN
CANE
CAT
DIKDIK
DIKTAT

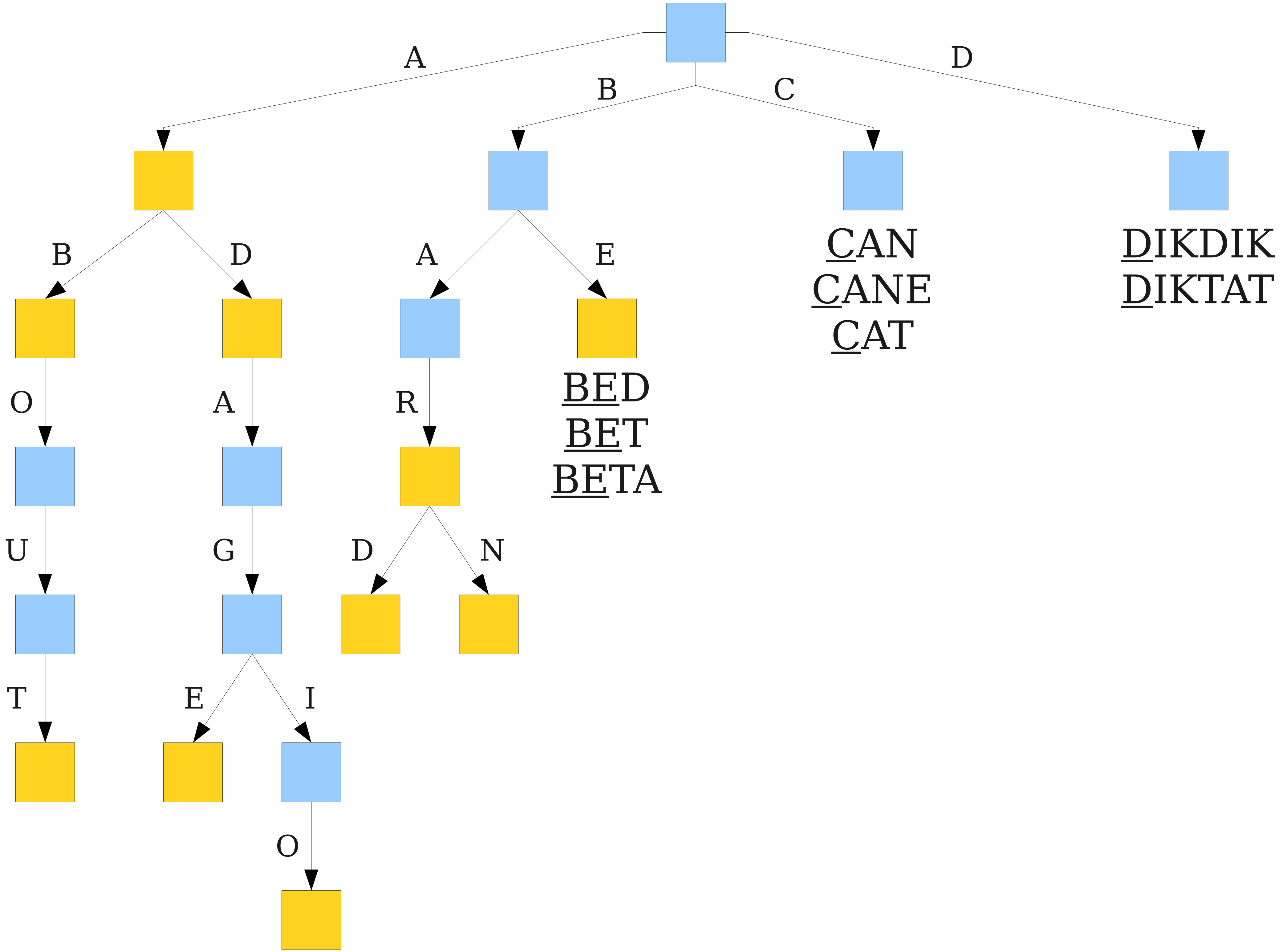

A
B
C
D
B
D
A
E
CAN
CANE
CAT
DIKDIK
DIKTAT
O
A
BAR
BARD
BARN
BE
BED
BET
BETA
U
G
T
E
I
O

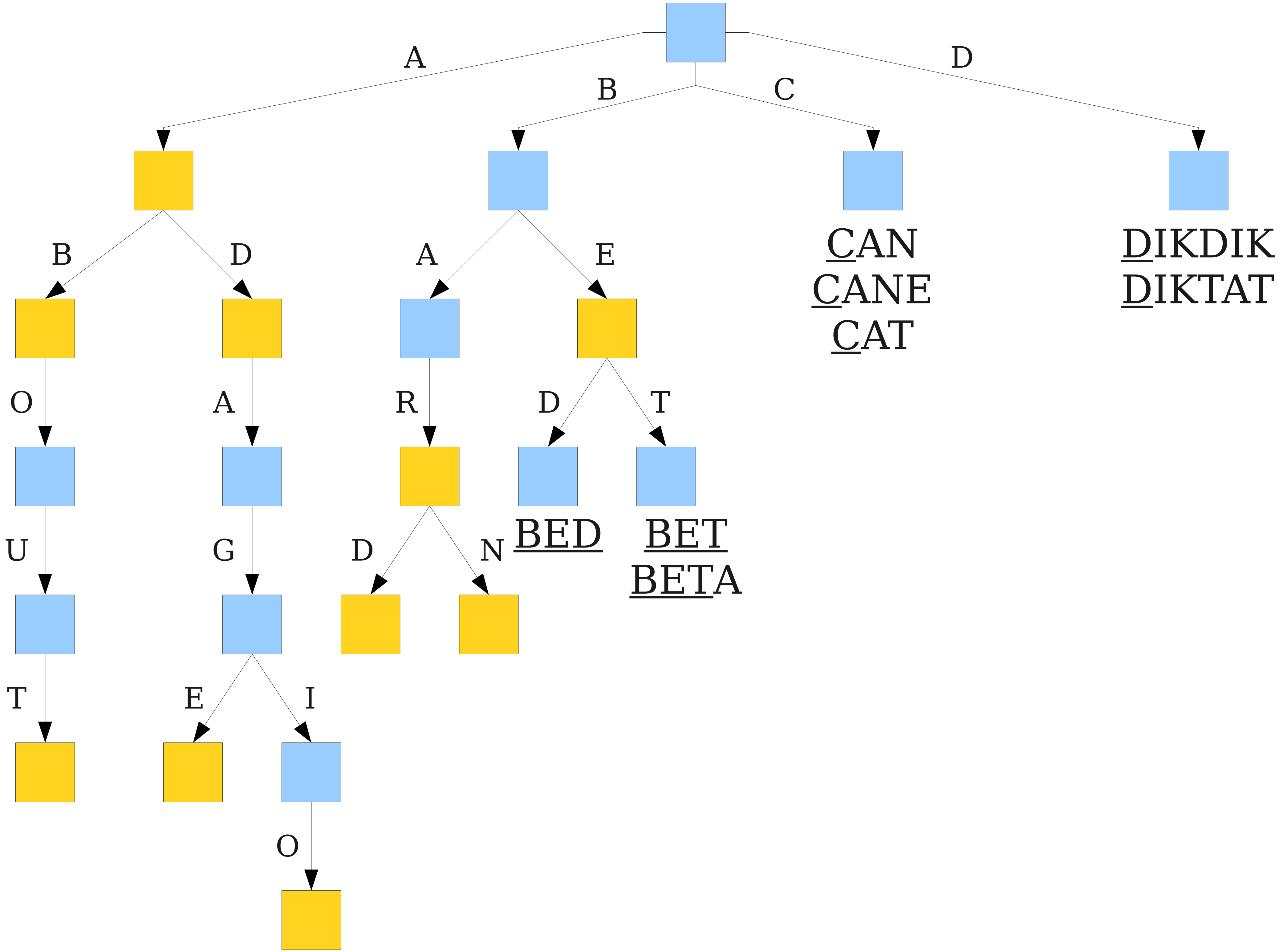

A
B
C
D
B
D
A
E
CAN
CANE
CAT
DIKDIK
DIKTAT
O
A
R
BE
BED
BET
BETA
U
G
BAR
BARD
BARN
T
E
I
O

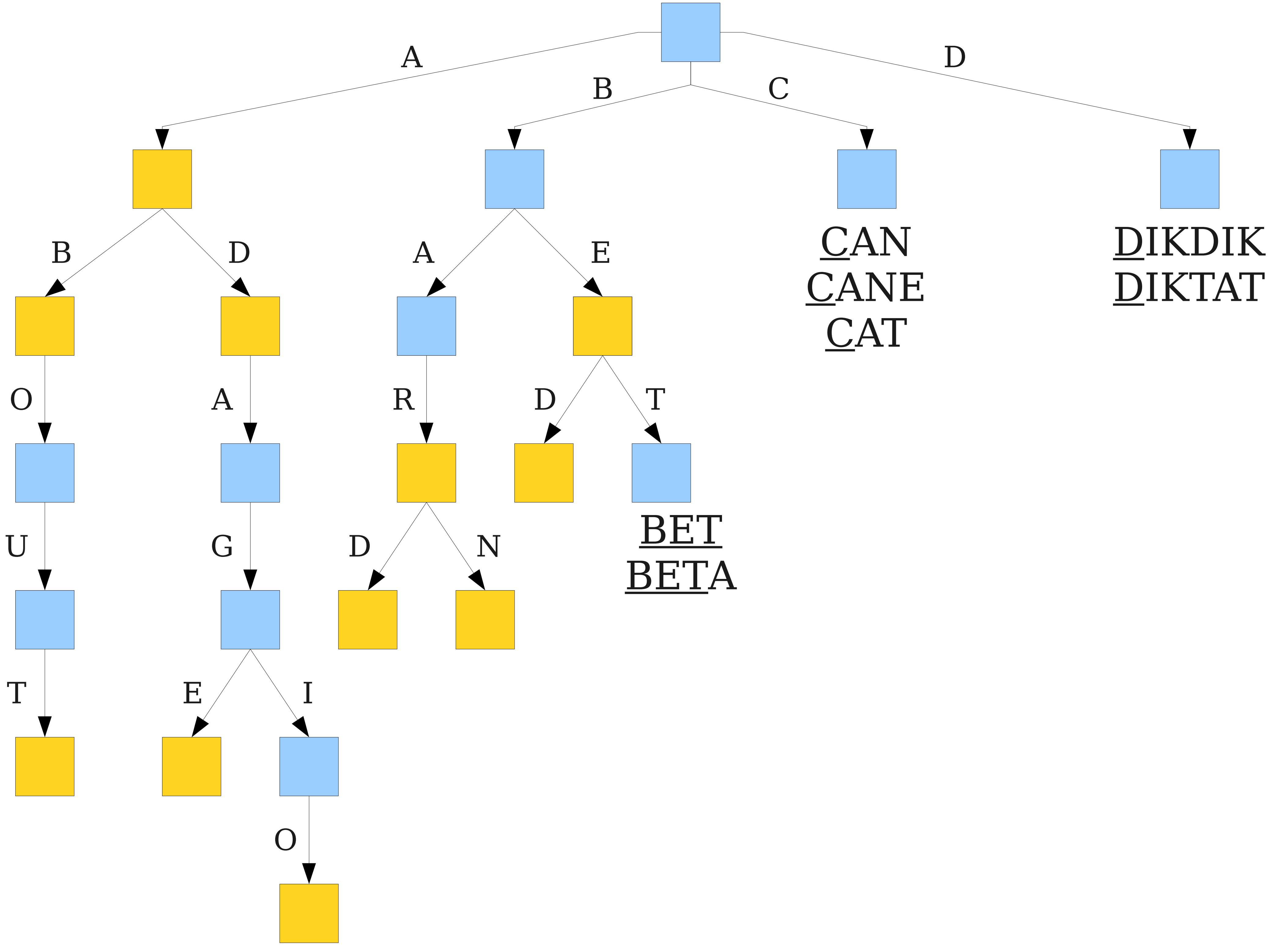

A
B
C
D
B
D
A
E
O
A
R
U
G
T
E
I
O
CAN
CANE
CAT
DIKDIK
DIKTAT
BE
BED
BET
BETA
BARD
BARN

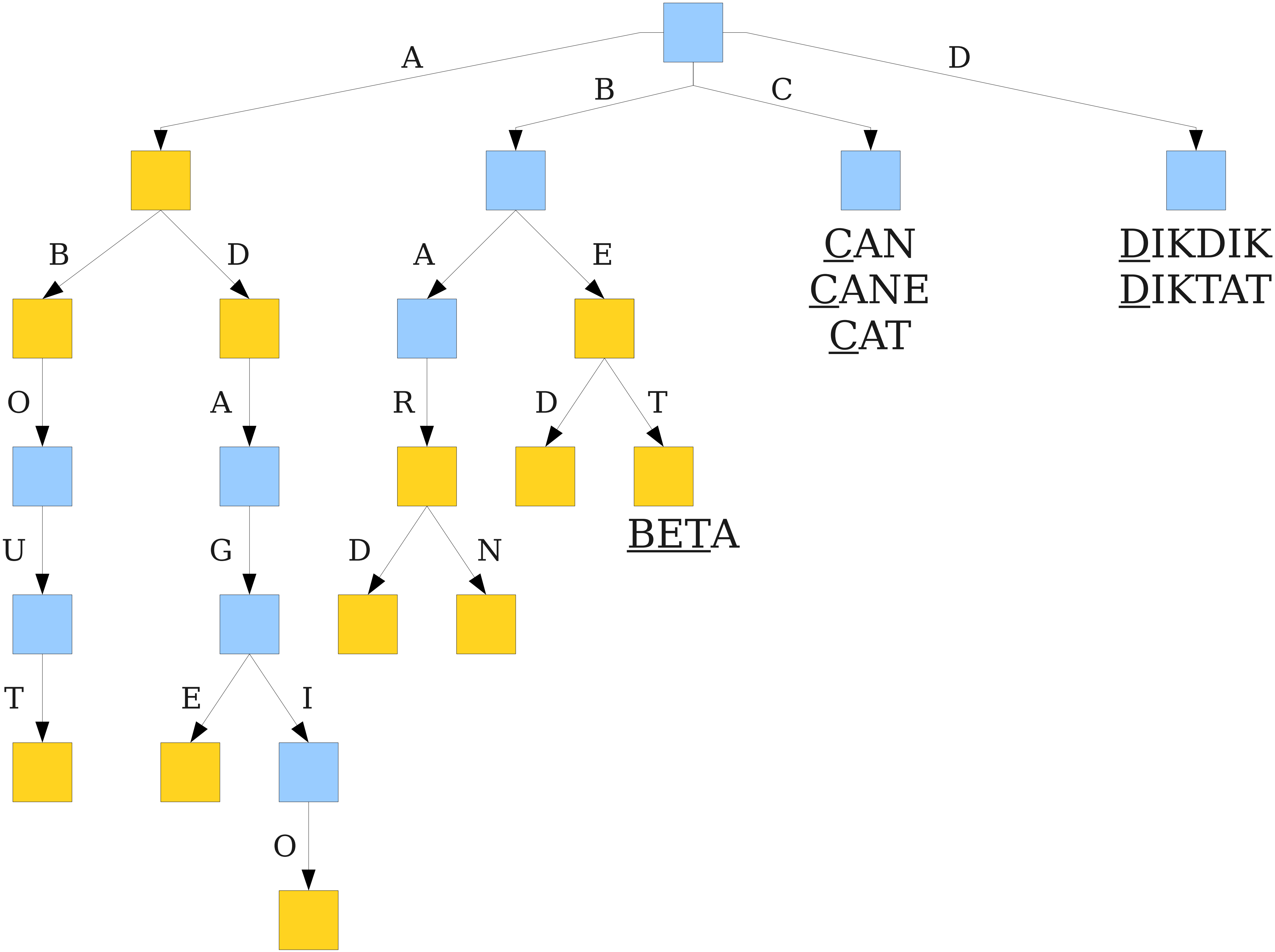

A
B
C
D
B
D
A
E
CAN
CANE
CAT
DIKDIK
DIKTAT
O
A
R
BE
BED
BET
BETA
U
G
D
N
T
E
I
BARD
BARN
O


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

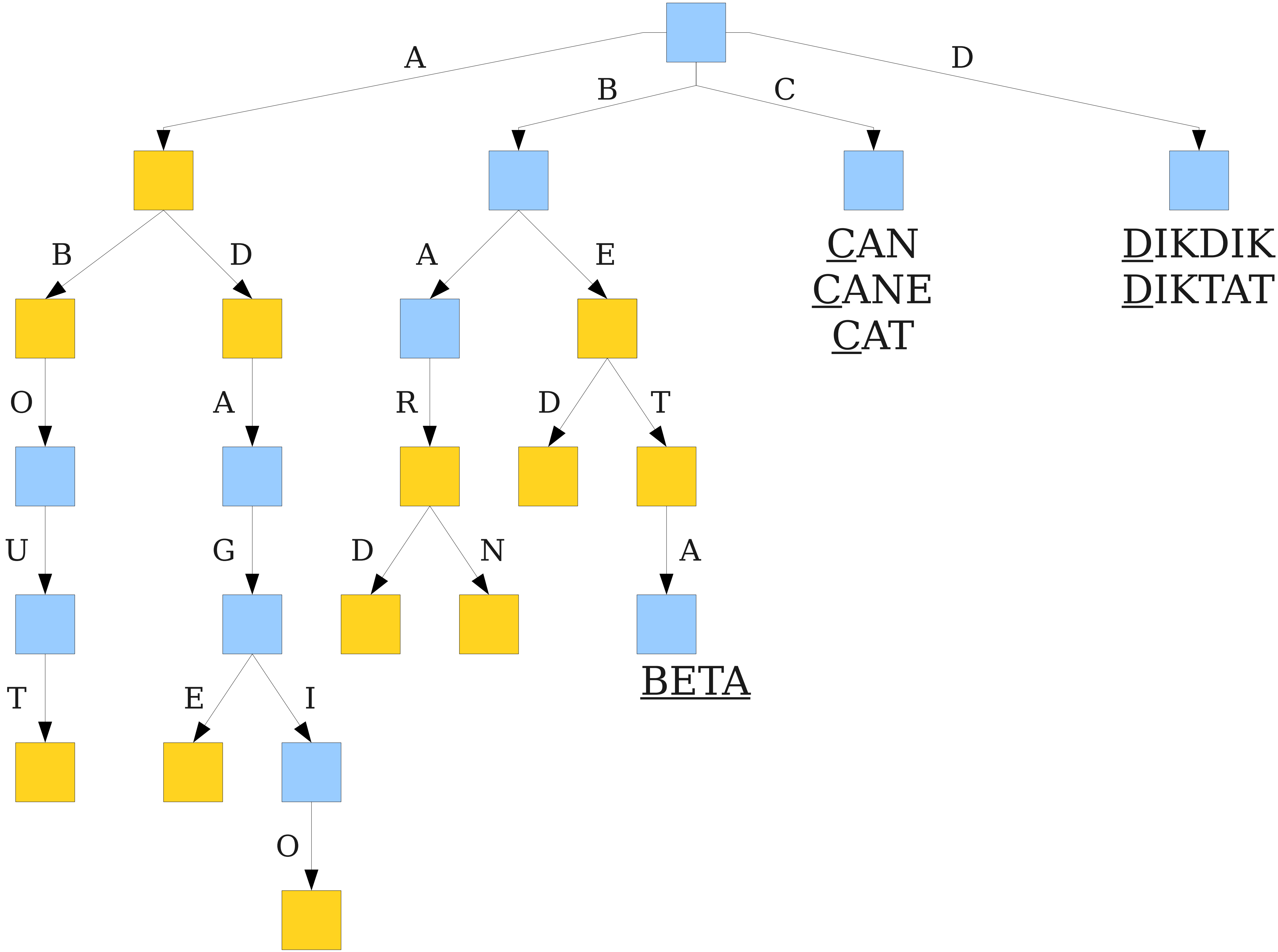

A
B
C
D
B
D
A
E
CAN
CANE
CAT
DIKDIK
DIKTAT
O
A
R
BED
BET
BETA
U
G
D
N
T
E
I
O

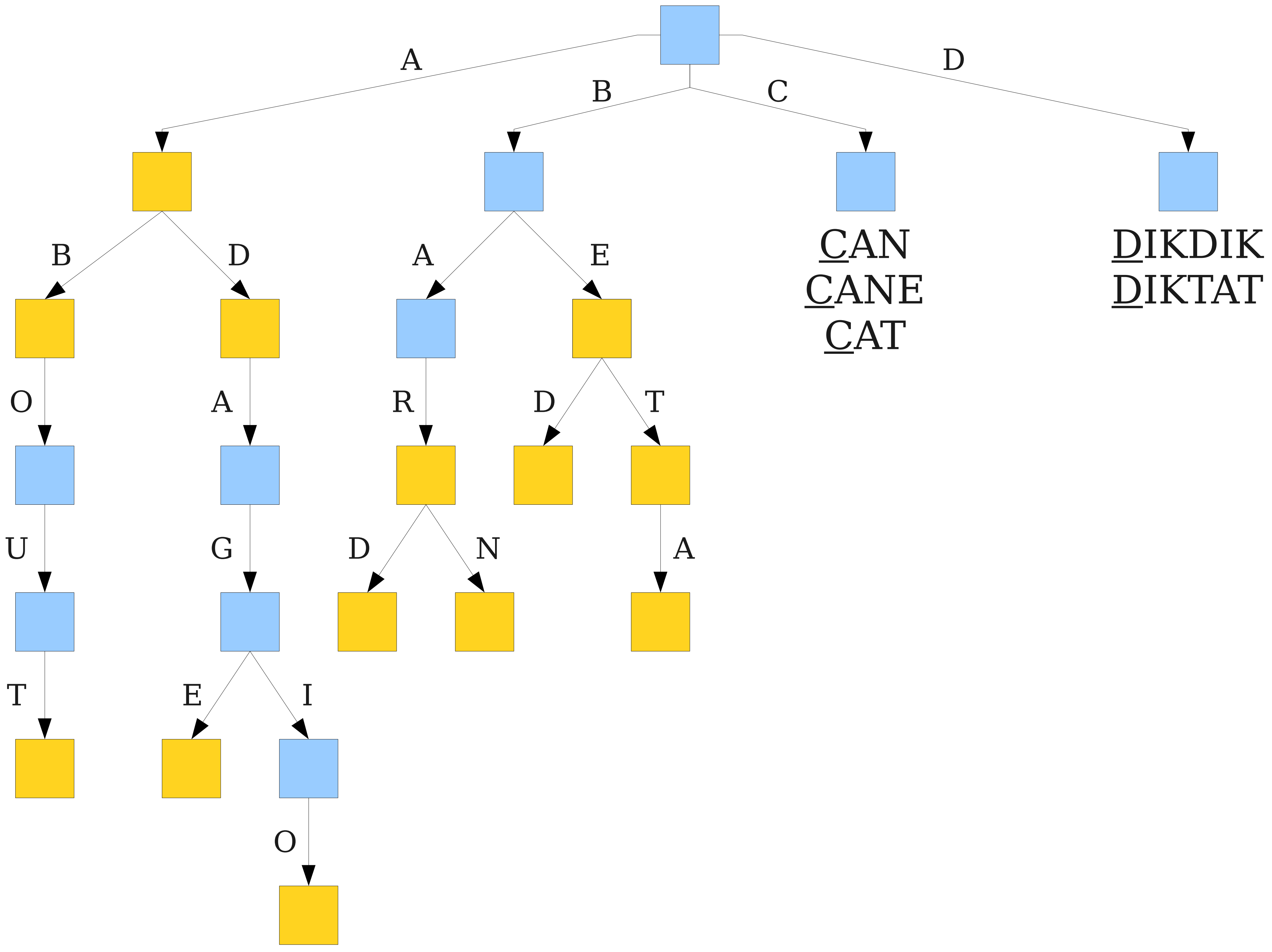

A
B
C
D
B
D
A
E
CAN
CANE
CAT
DIKDIK
DIKTAT
O
A
R
D
T
U
G
D
N
BED
BET
BETA
T
E
I
O

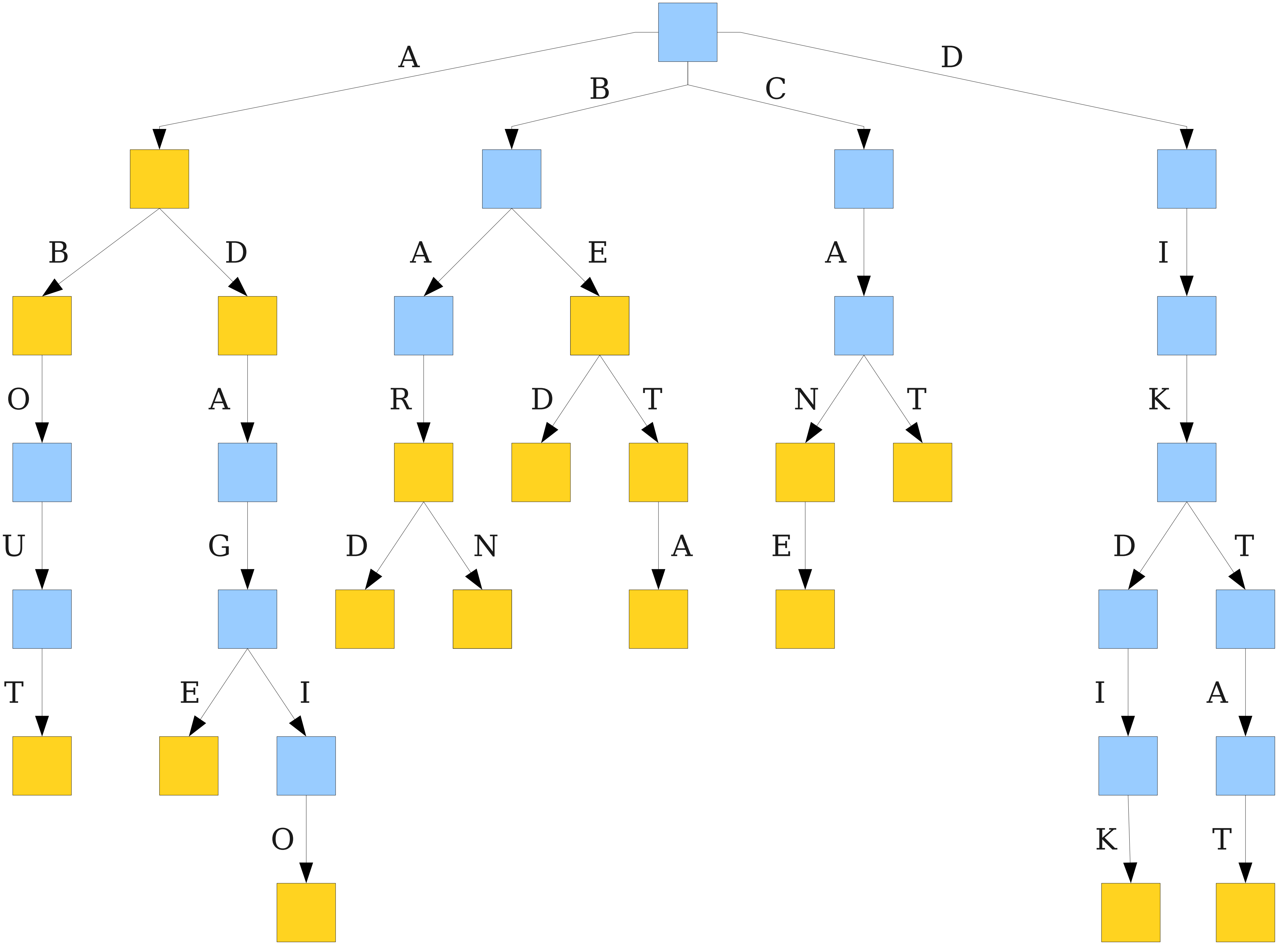

A
B
C
D
B
D
A
E
O
A
R
D
T
U
G
D
N
T
E
I
O
CAN
CANE
CAT
DIKDIK
DIKTAT
BET
BETA

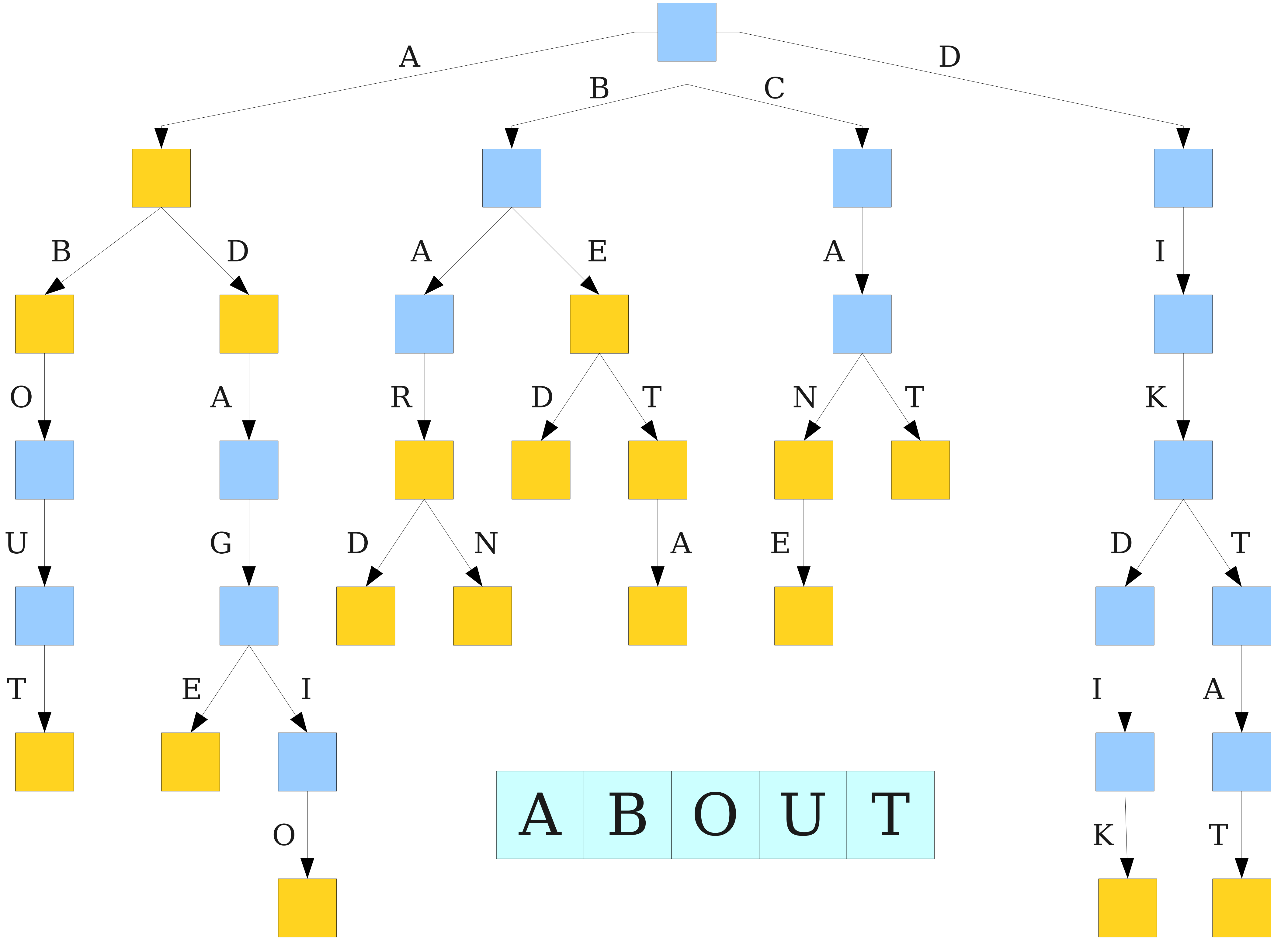


A
B
C
D
B
D
A
E
CAN
CANE
CAT
DIKDIK
DIKTAT
O
A
R
D
T
U
G
D
N
BETA
T
E
I
O

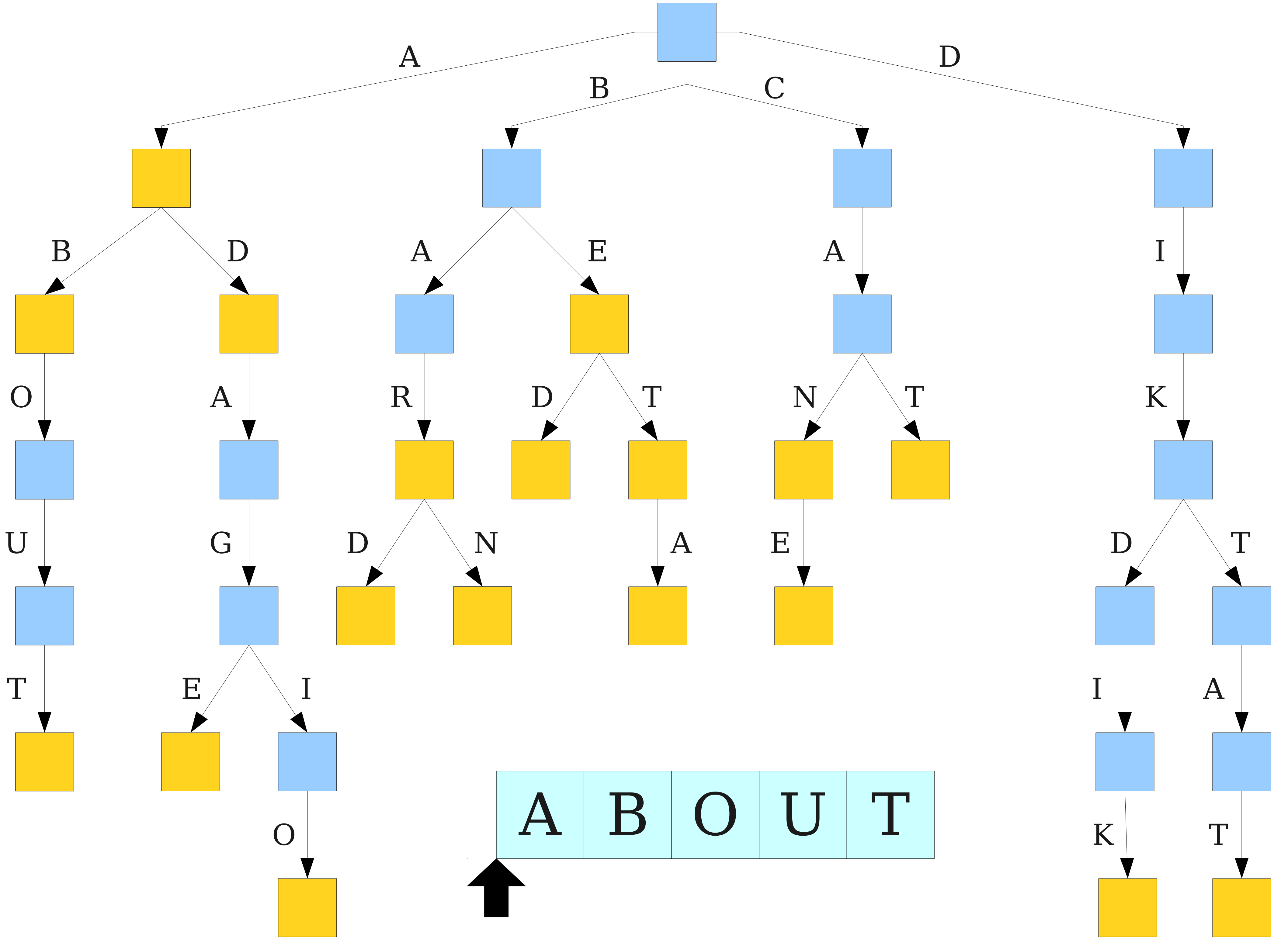

A
B
C
D
B
D
A
E
CAN
CANE
CAT
DIKDIK
DIKTAT
O
A
R
D
T
U
G
D
N
A
BETA
T
E
I
O

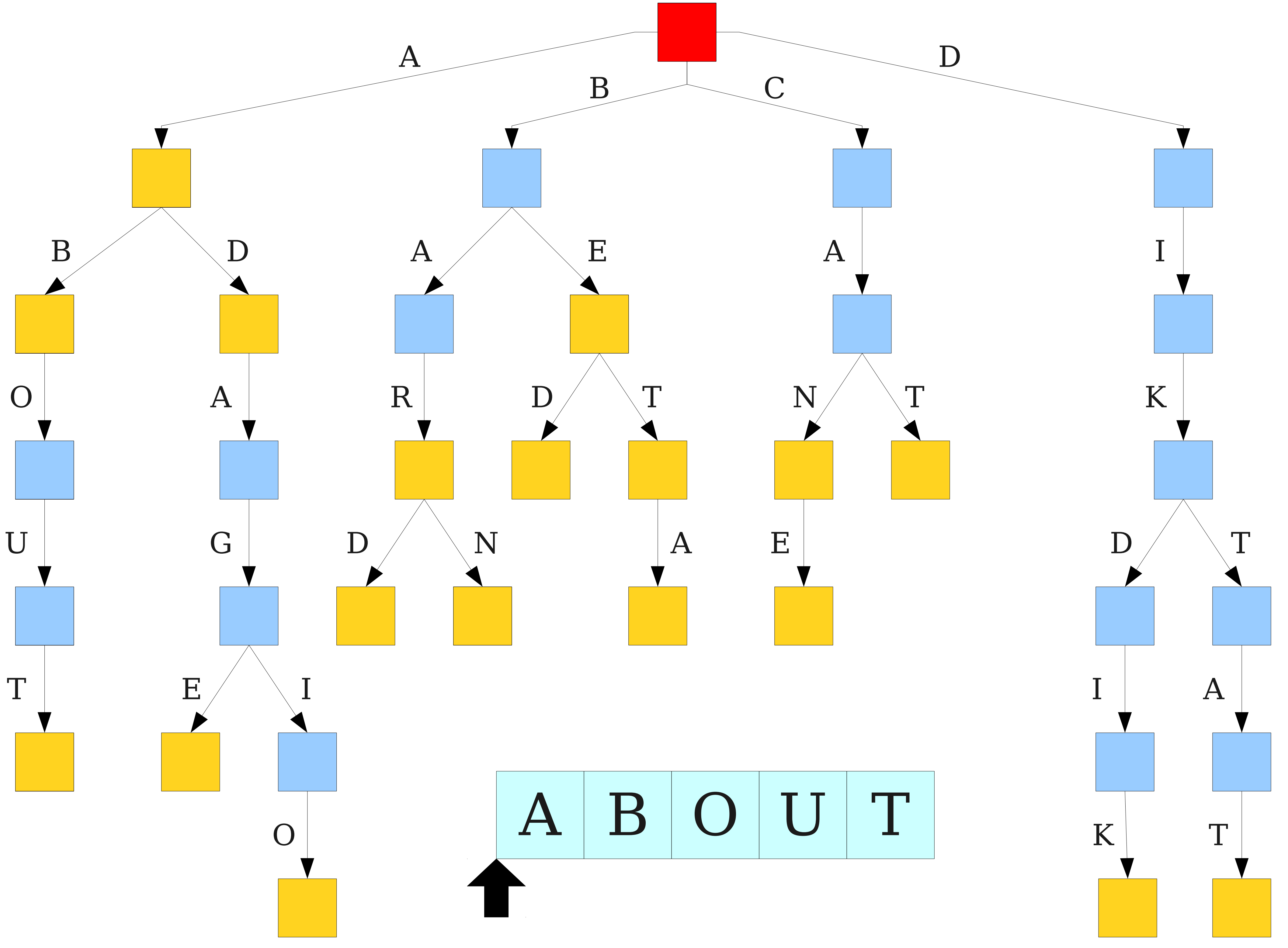

A
B
C
D
B
D
A
E
CAN
CANE
CAT
DIKDIK
DIKTAT
O
A
R
D
T
U
G
D
N
A
T
E
I
O

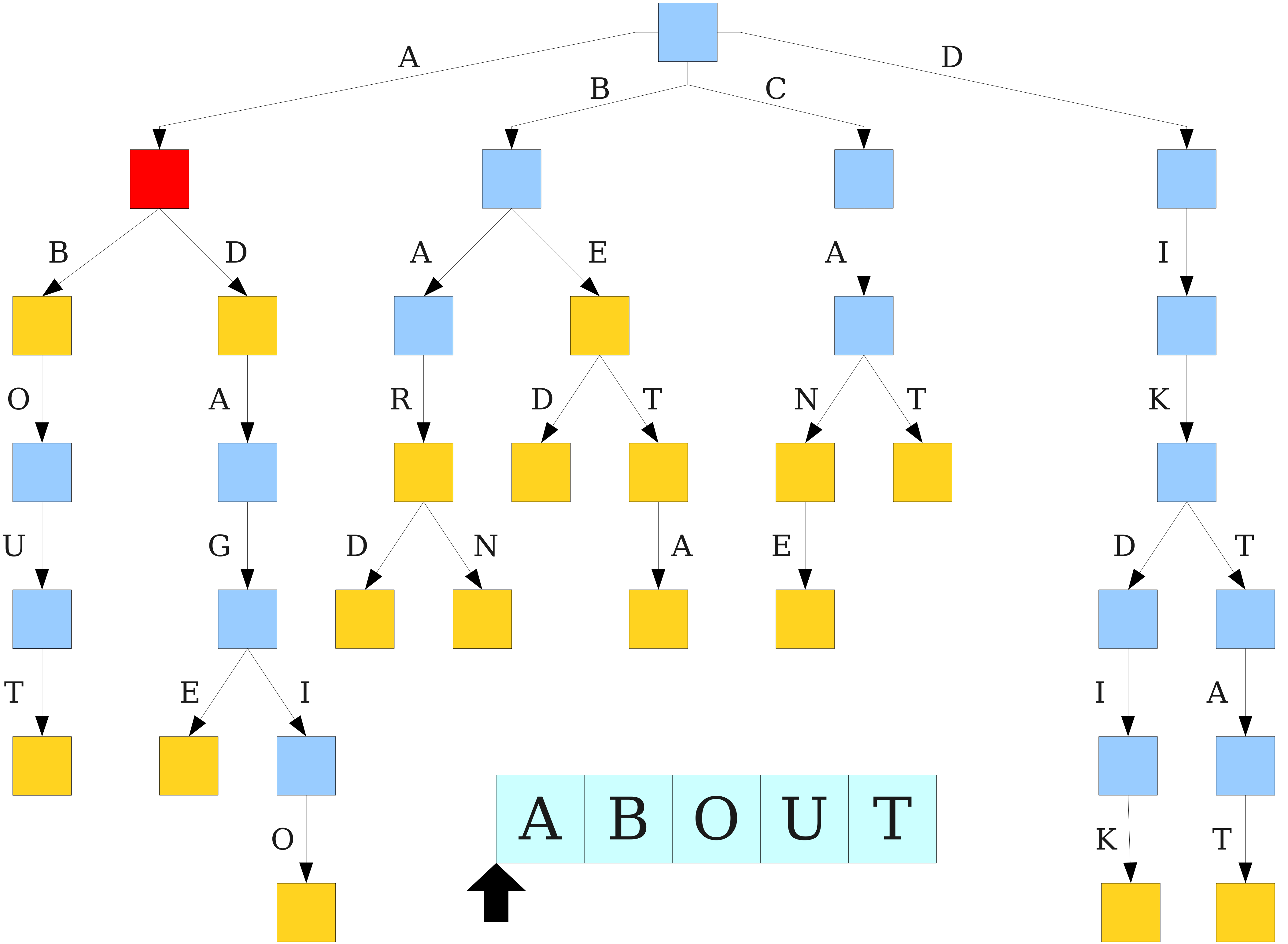

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T

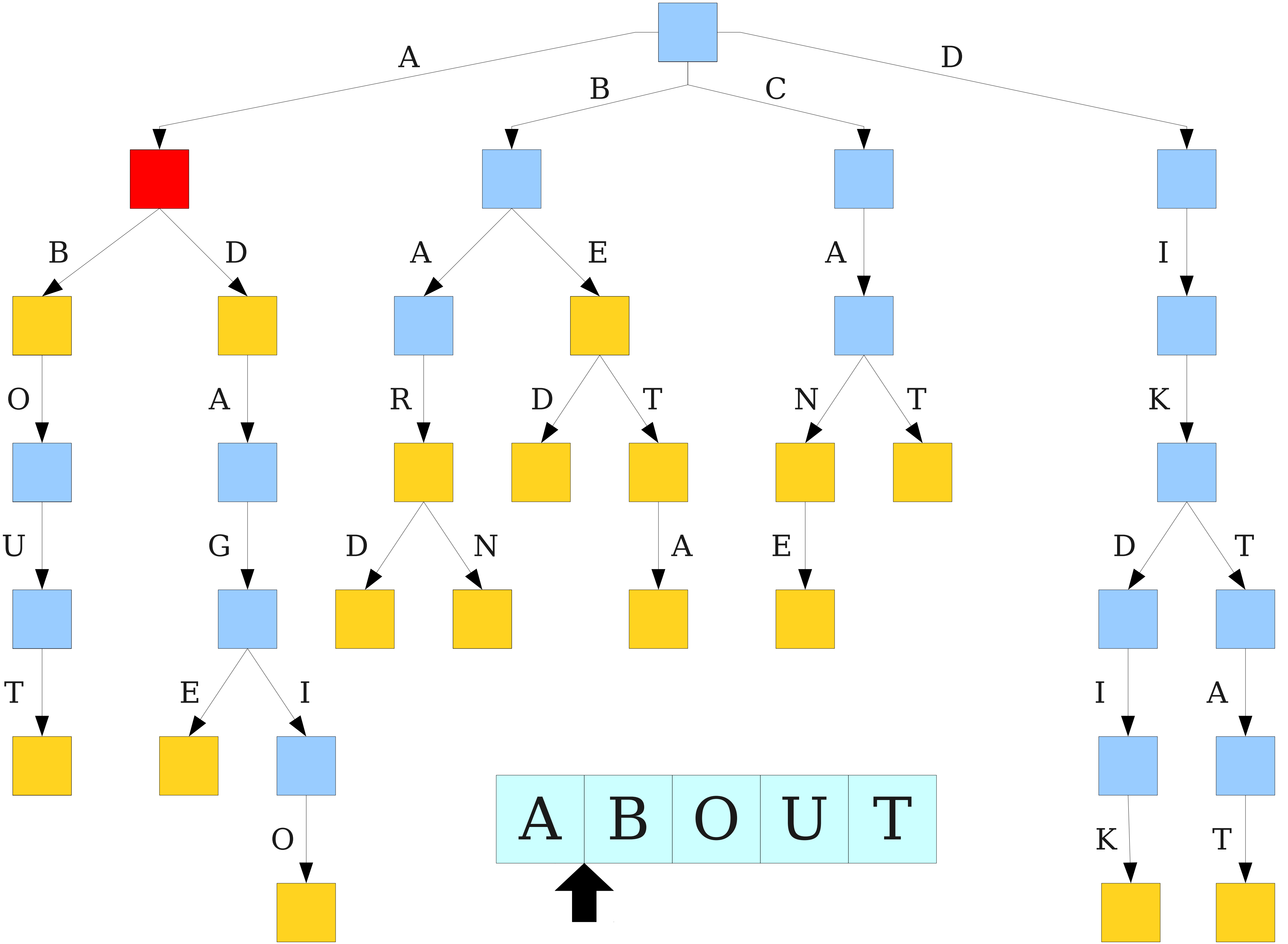


A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A B O U T

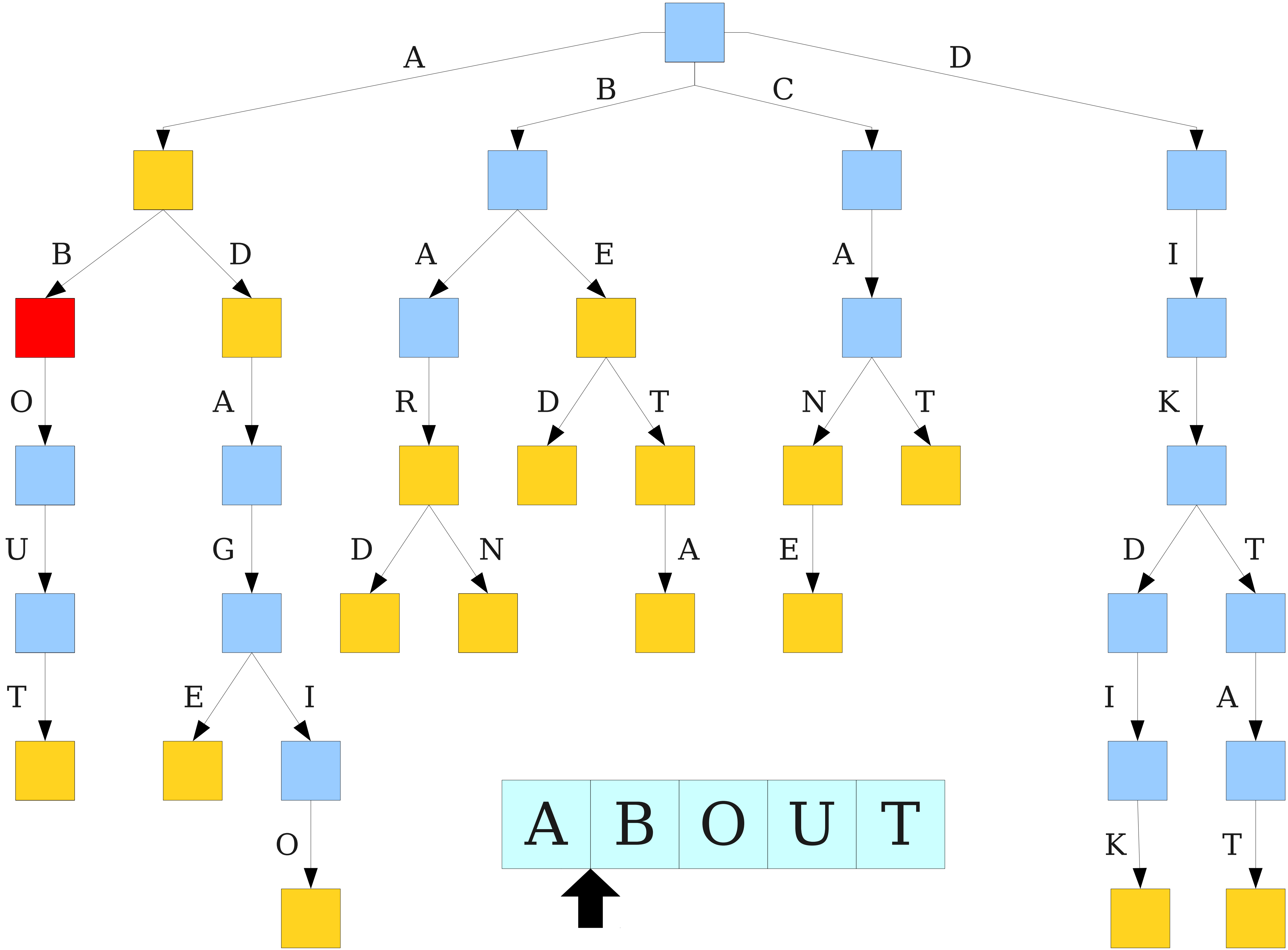

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A
B
O
U
T

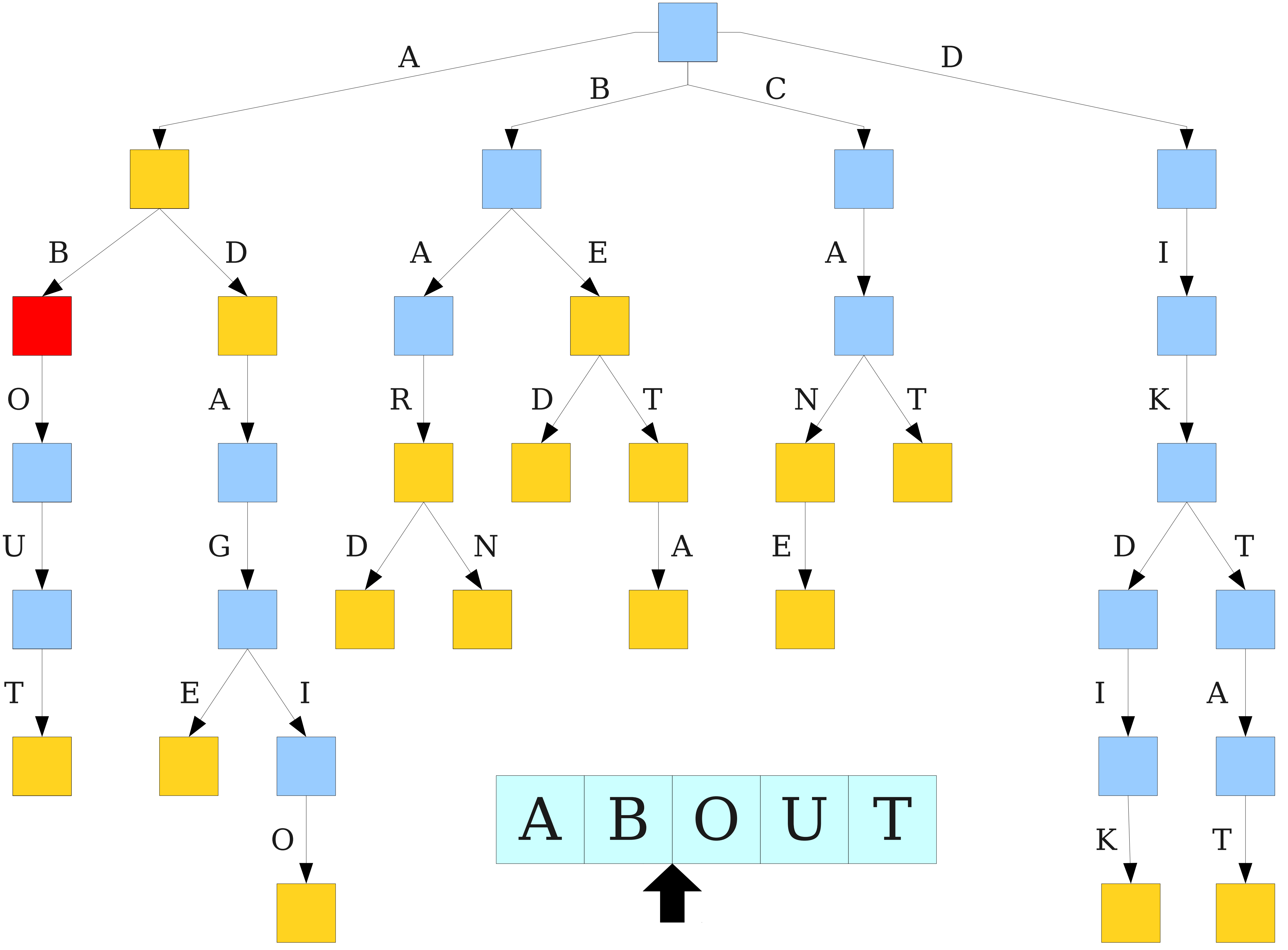

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A
B
O
U
T

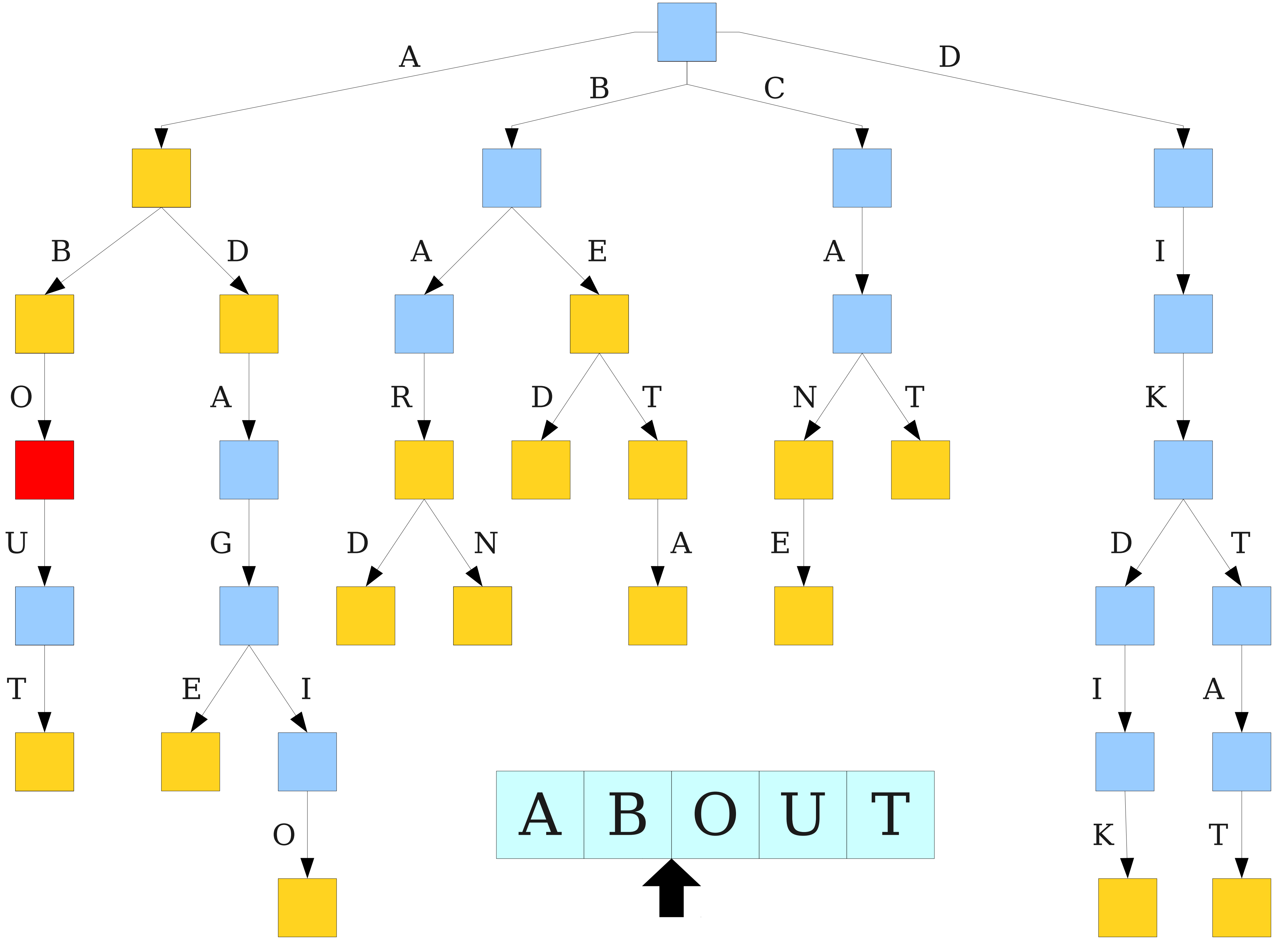

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A B O U T

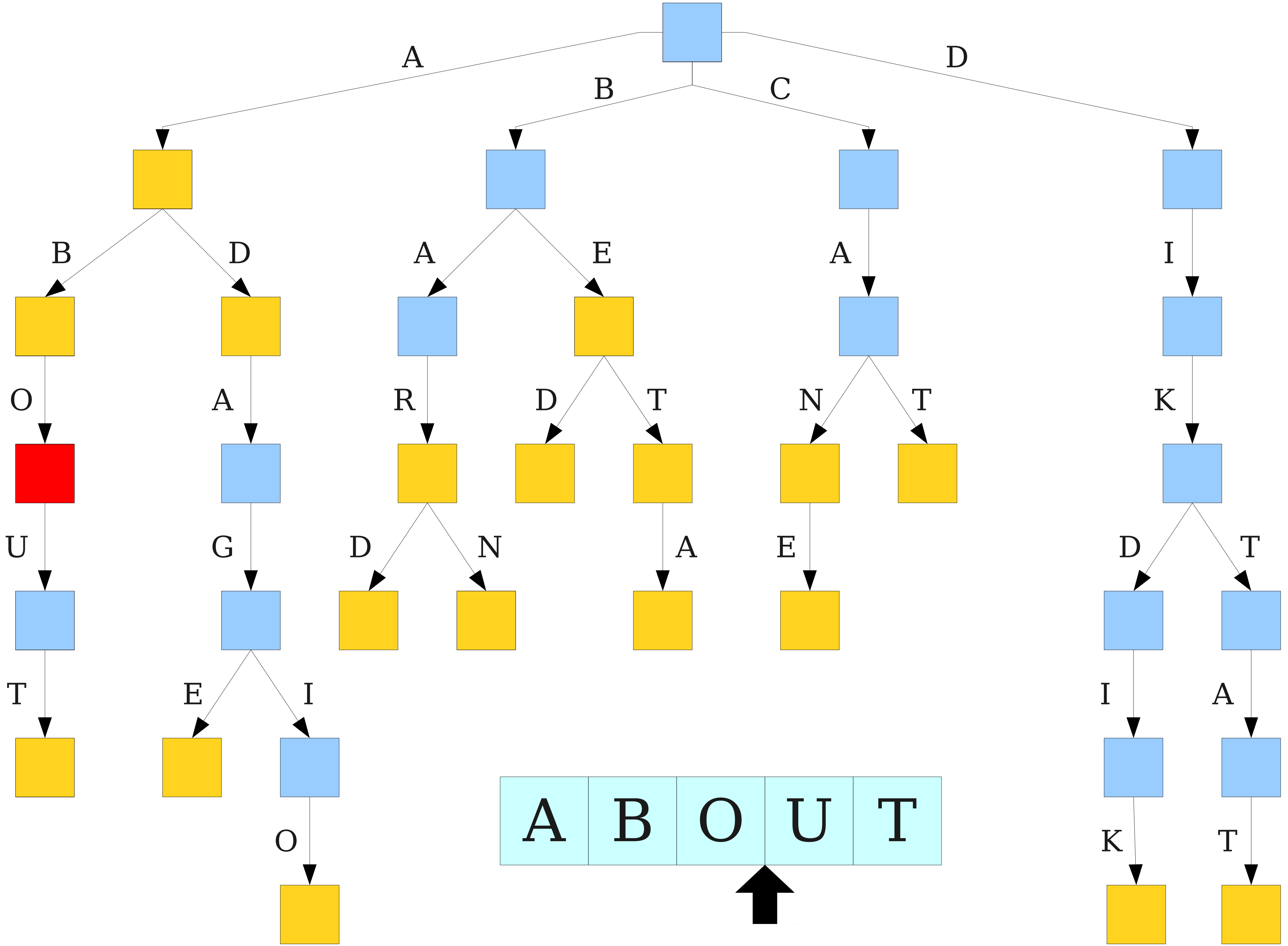


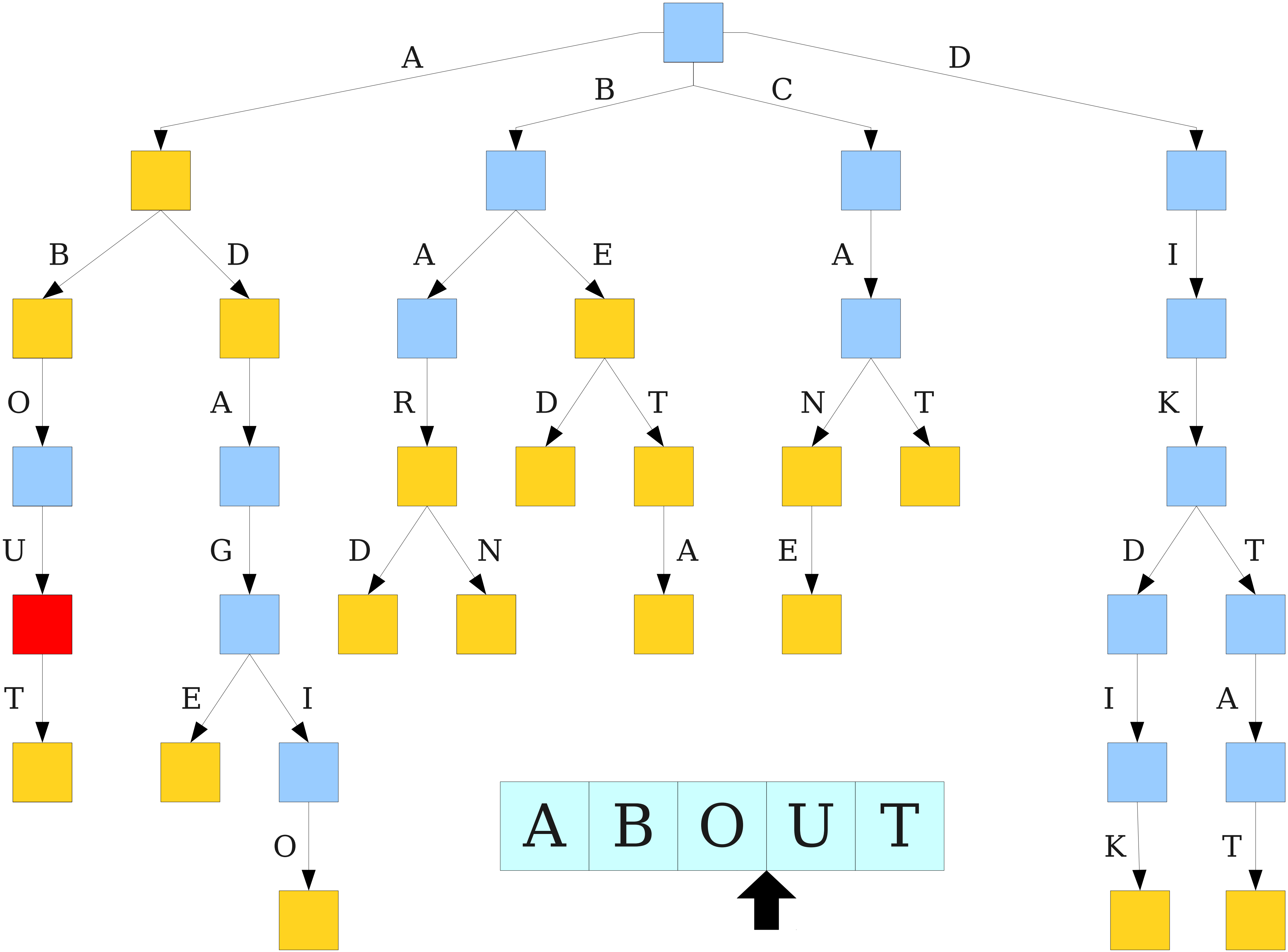
A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A B O U T


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

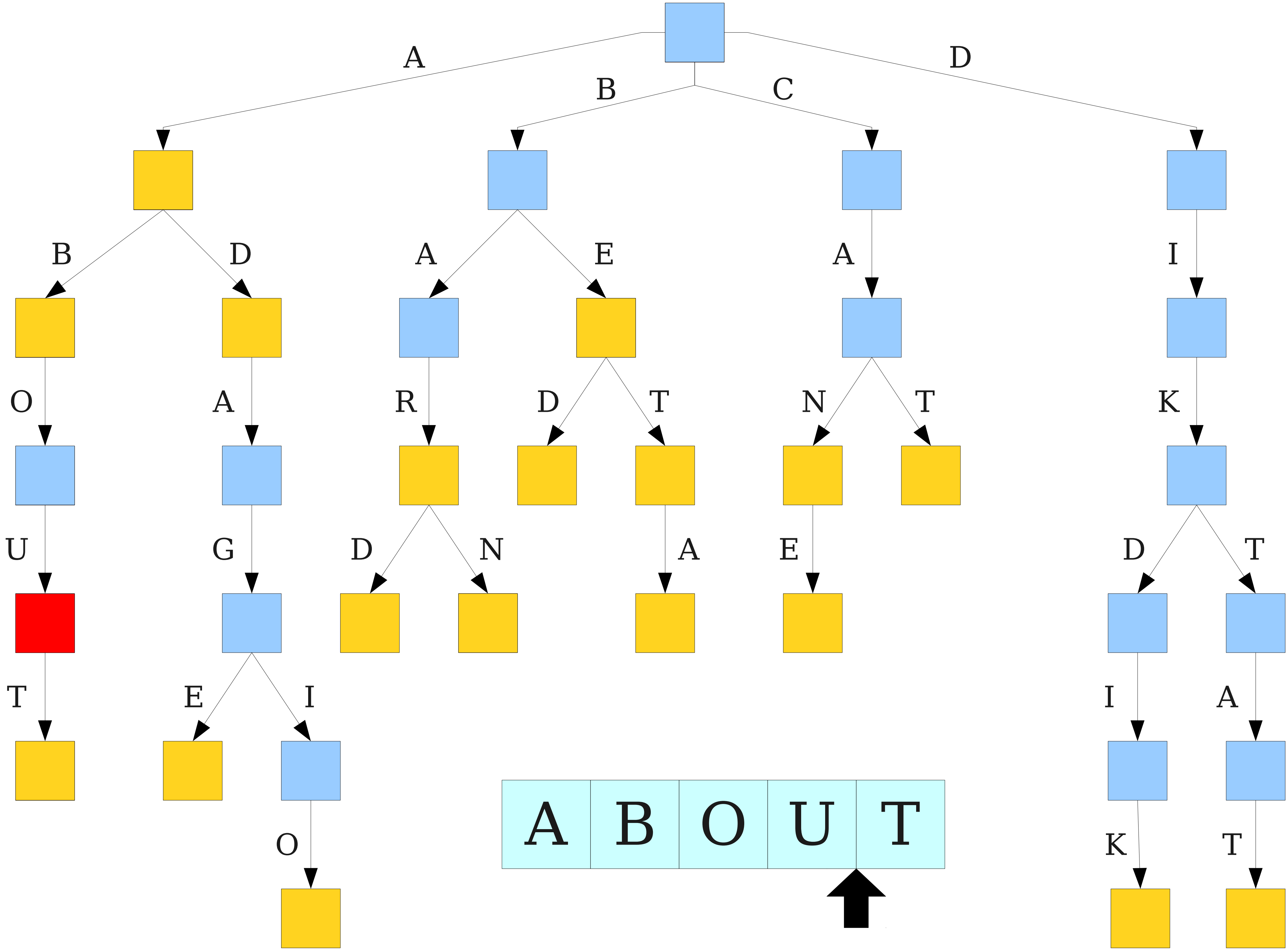

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A B O U T

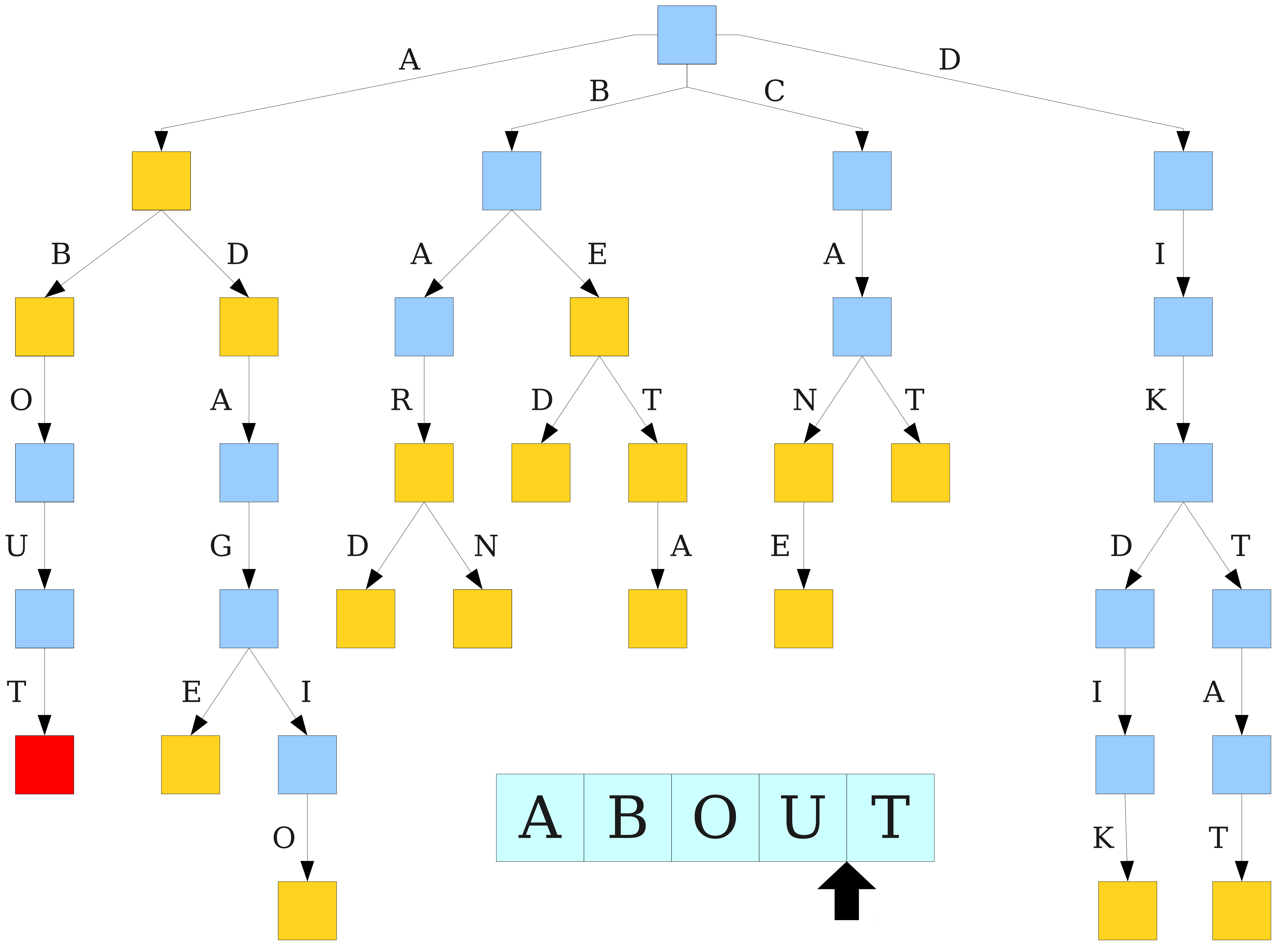

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A B O U T

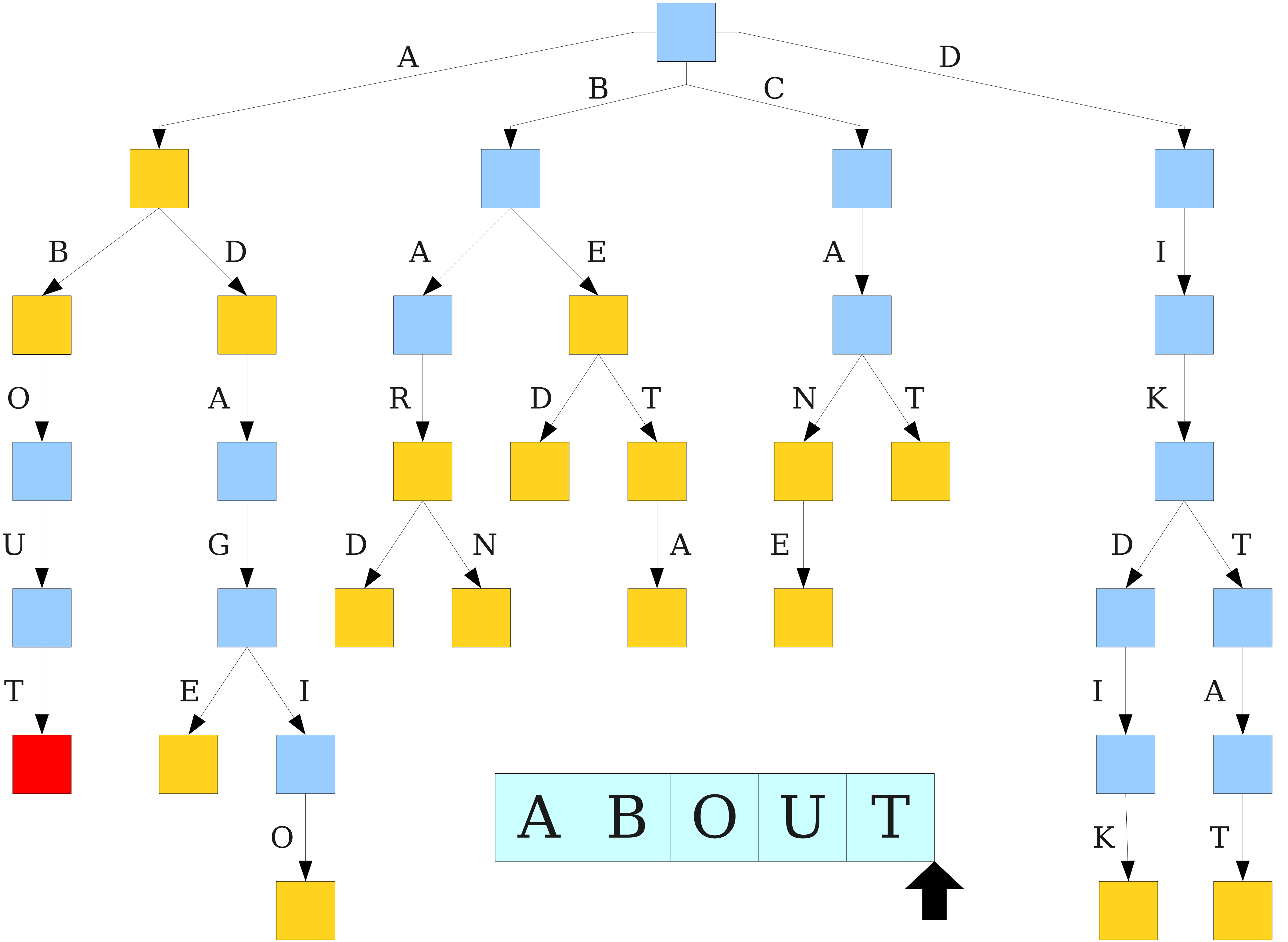

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A B O U T

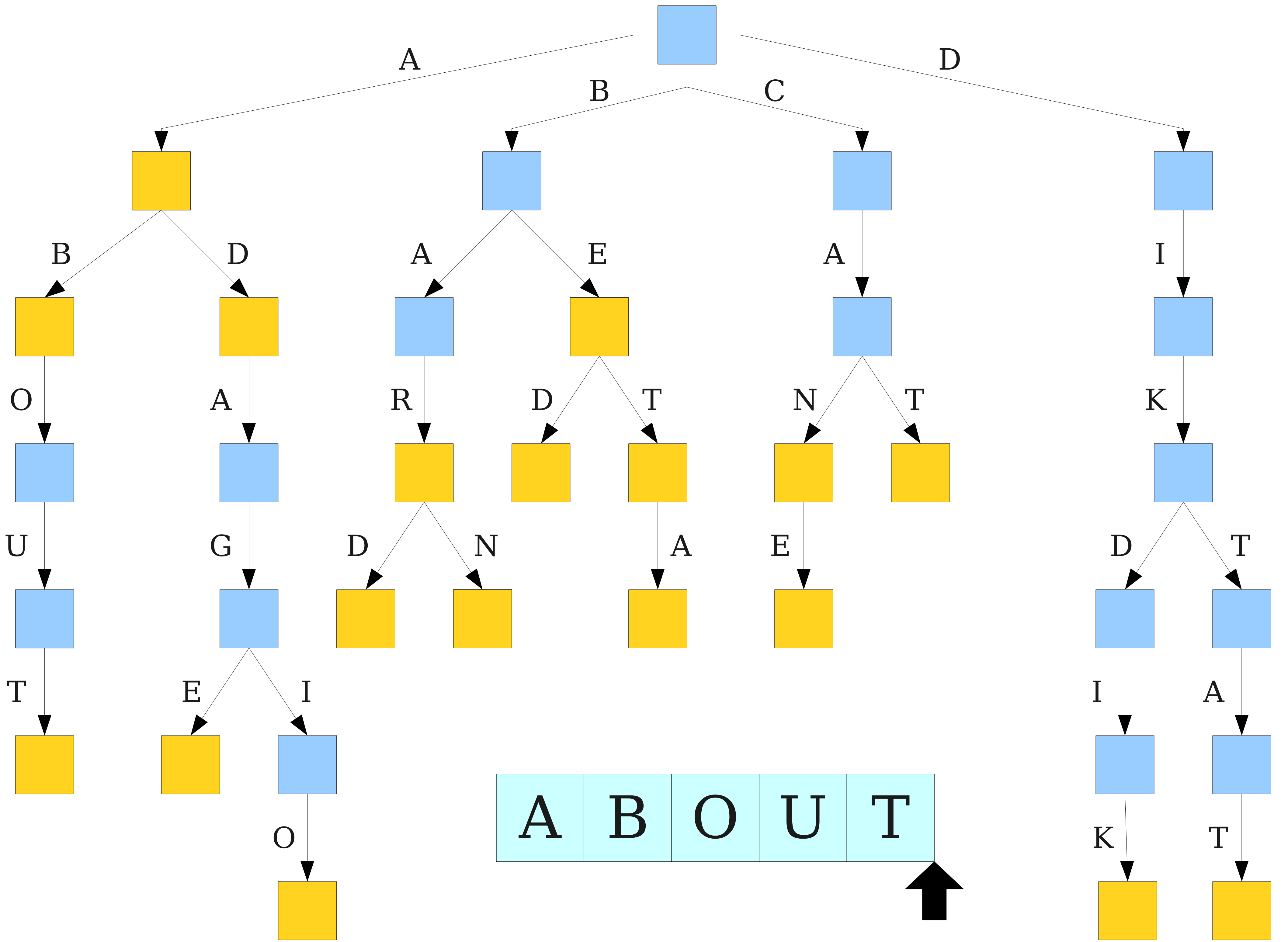


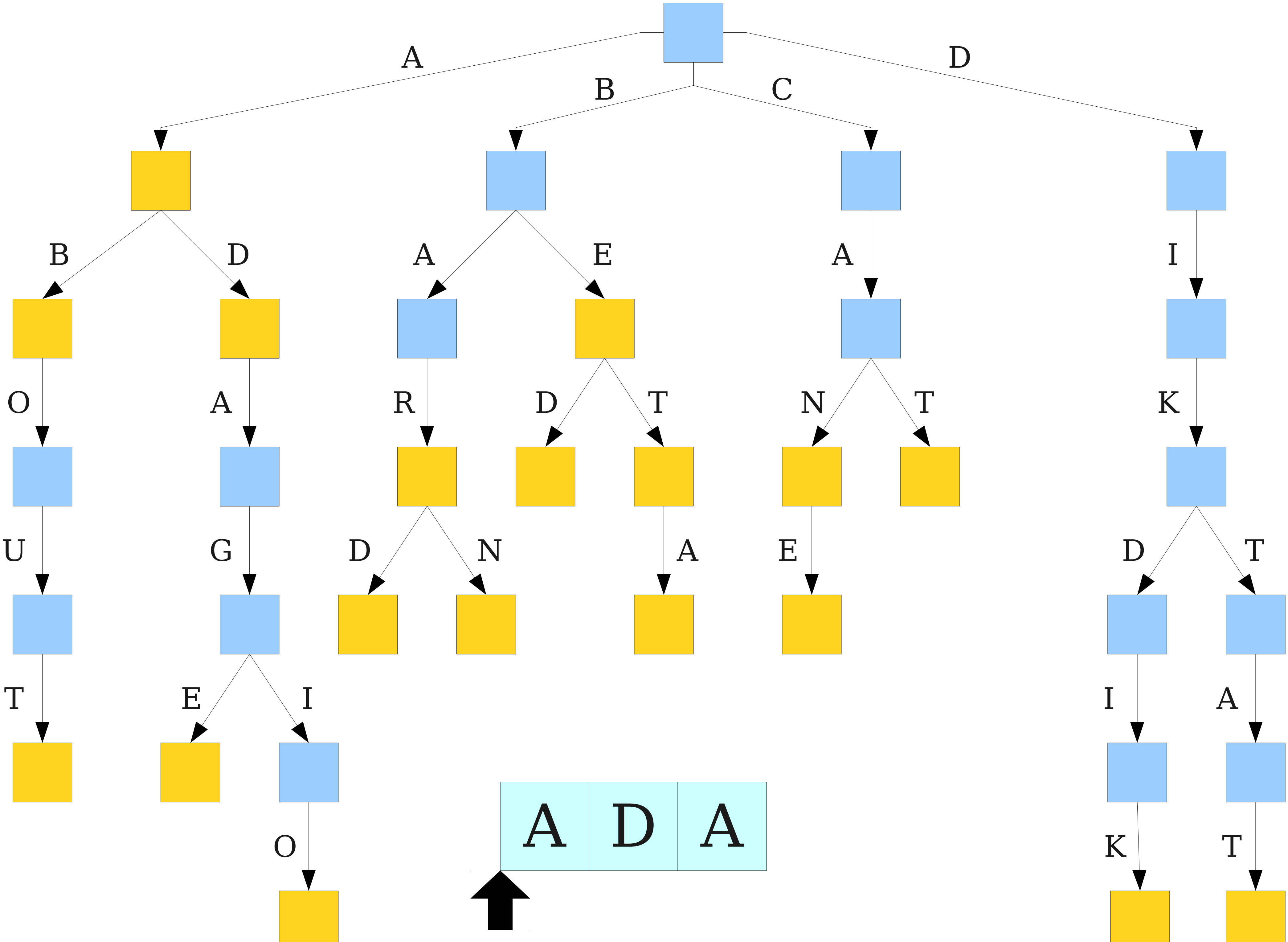
A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A B O U T

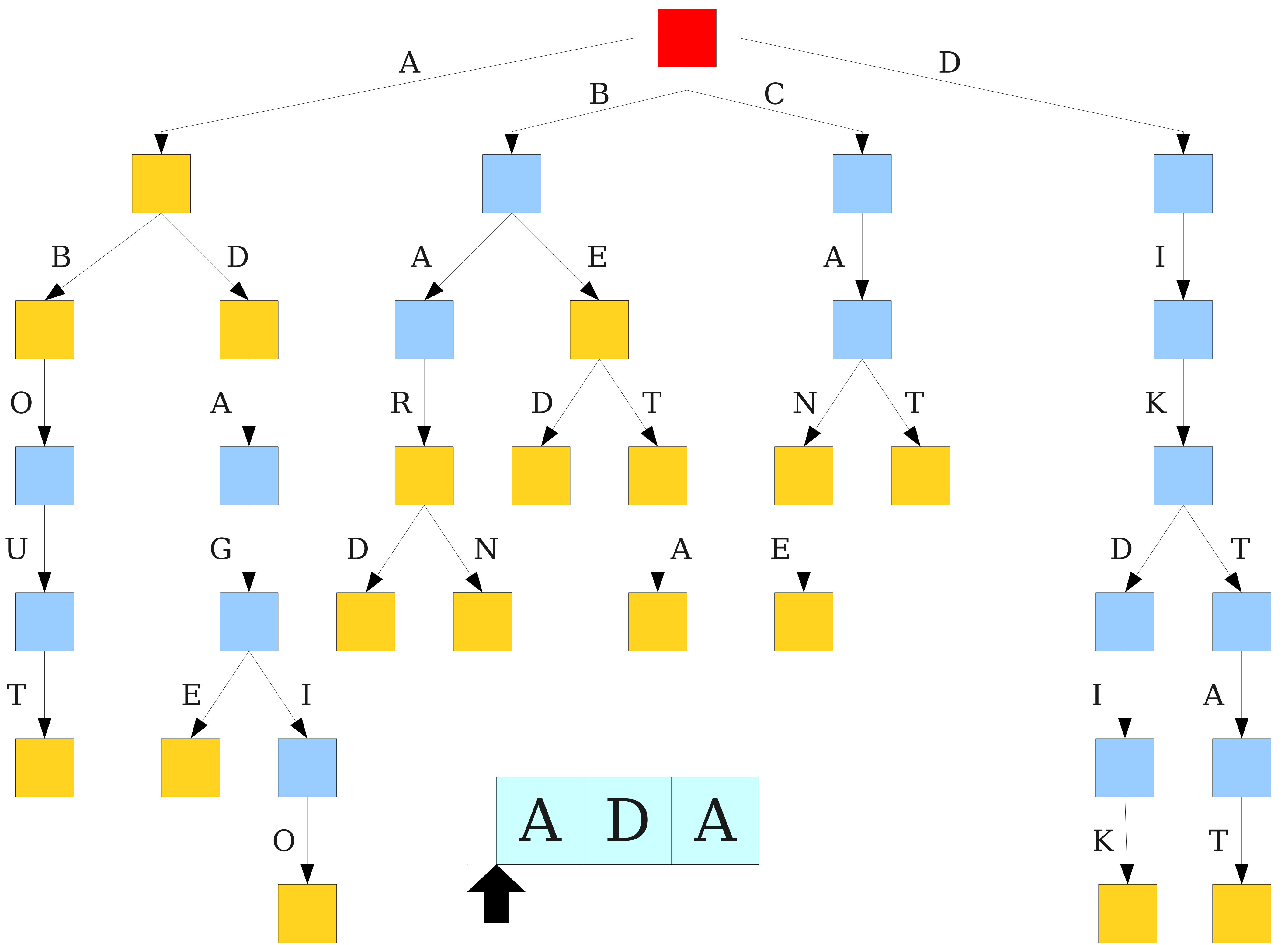

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A B O U T

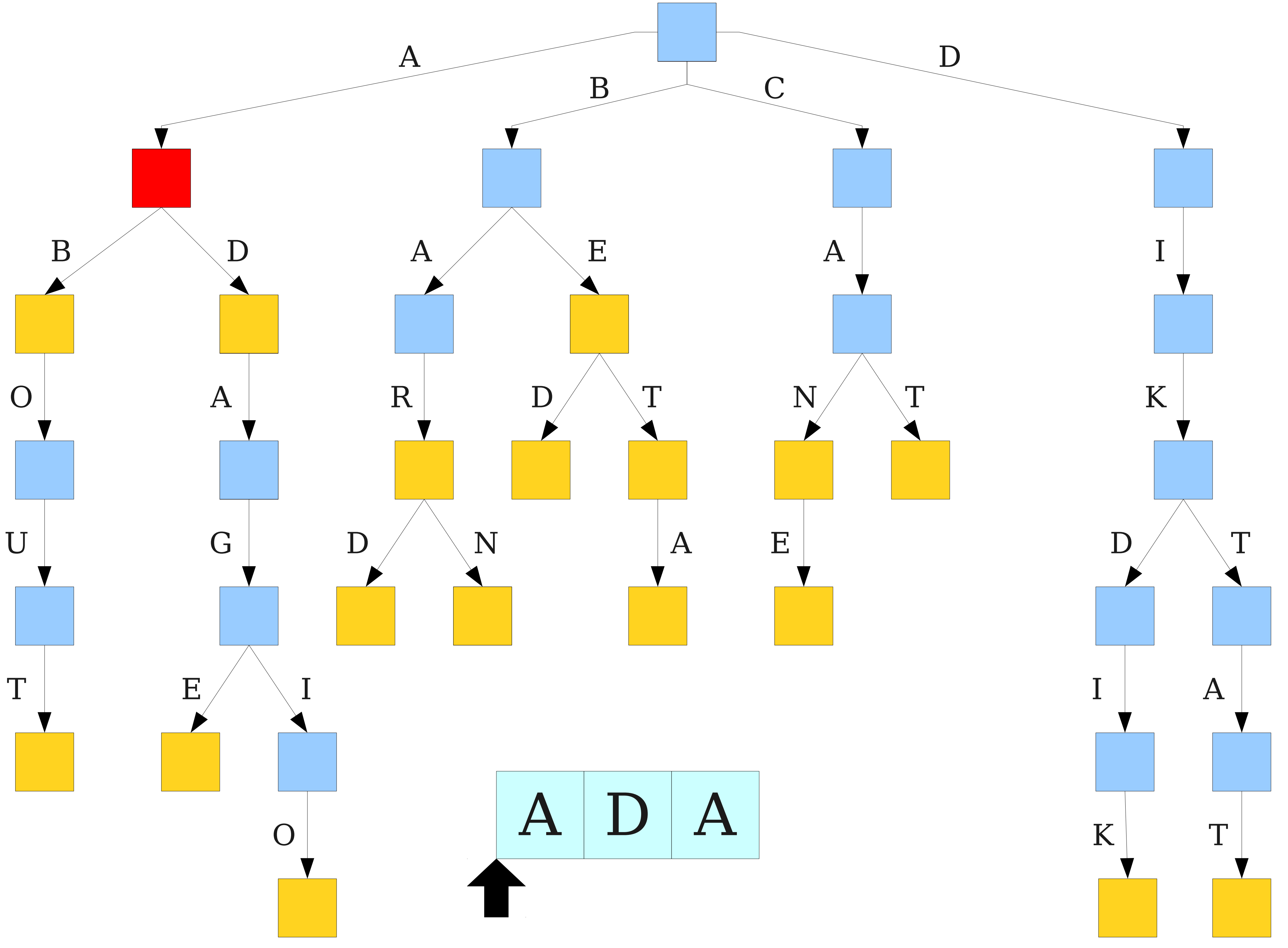

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A B O U T

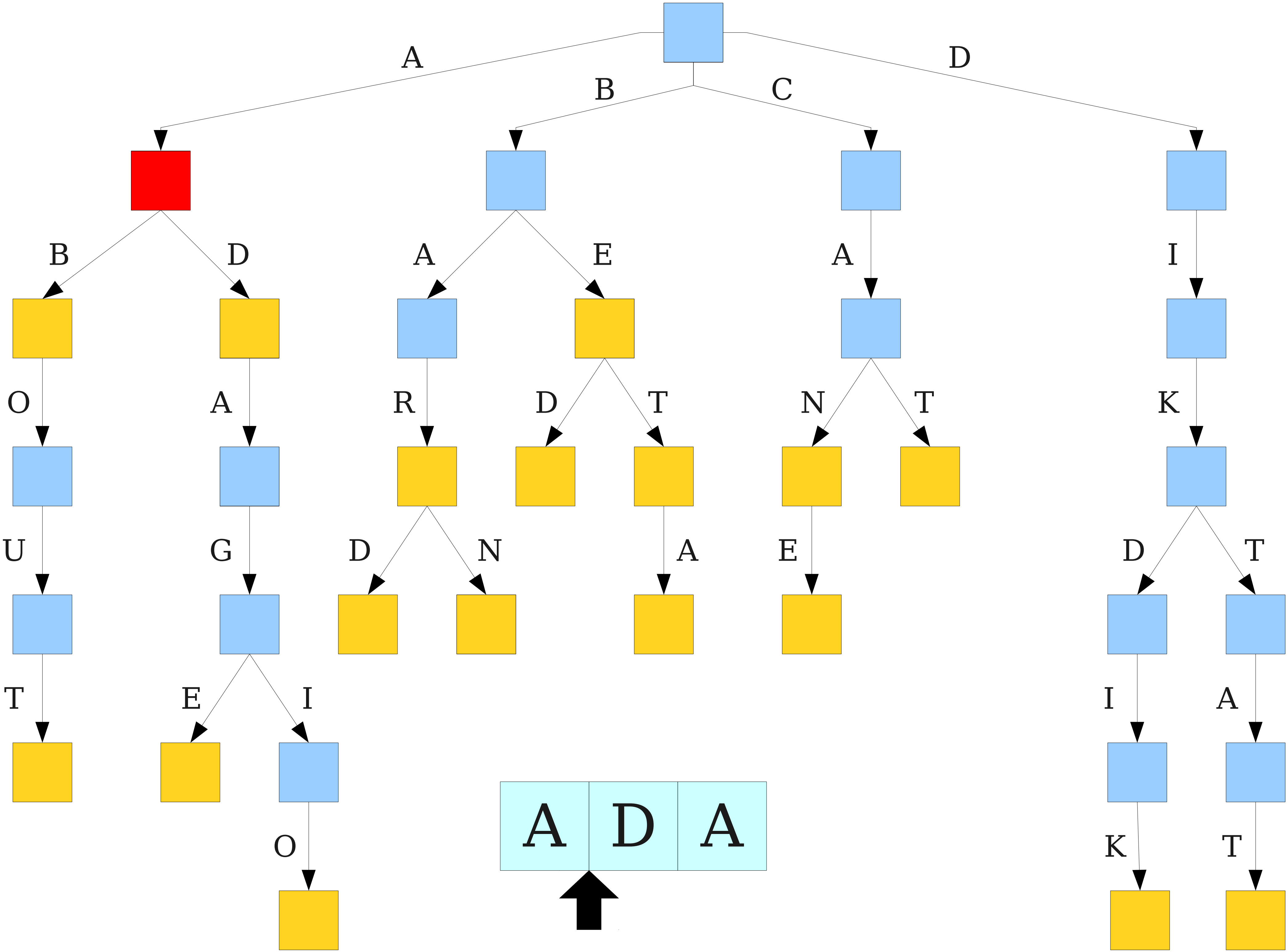

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A B O U T


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

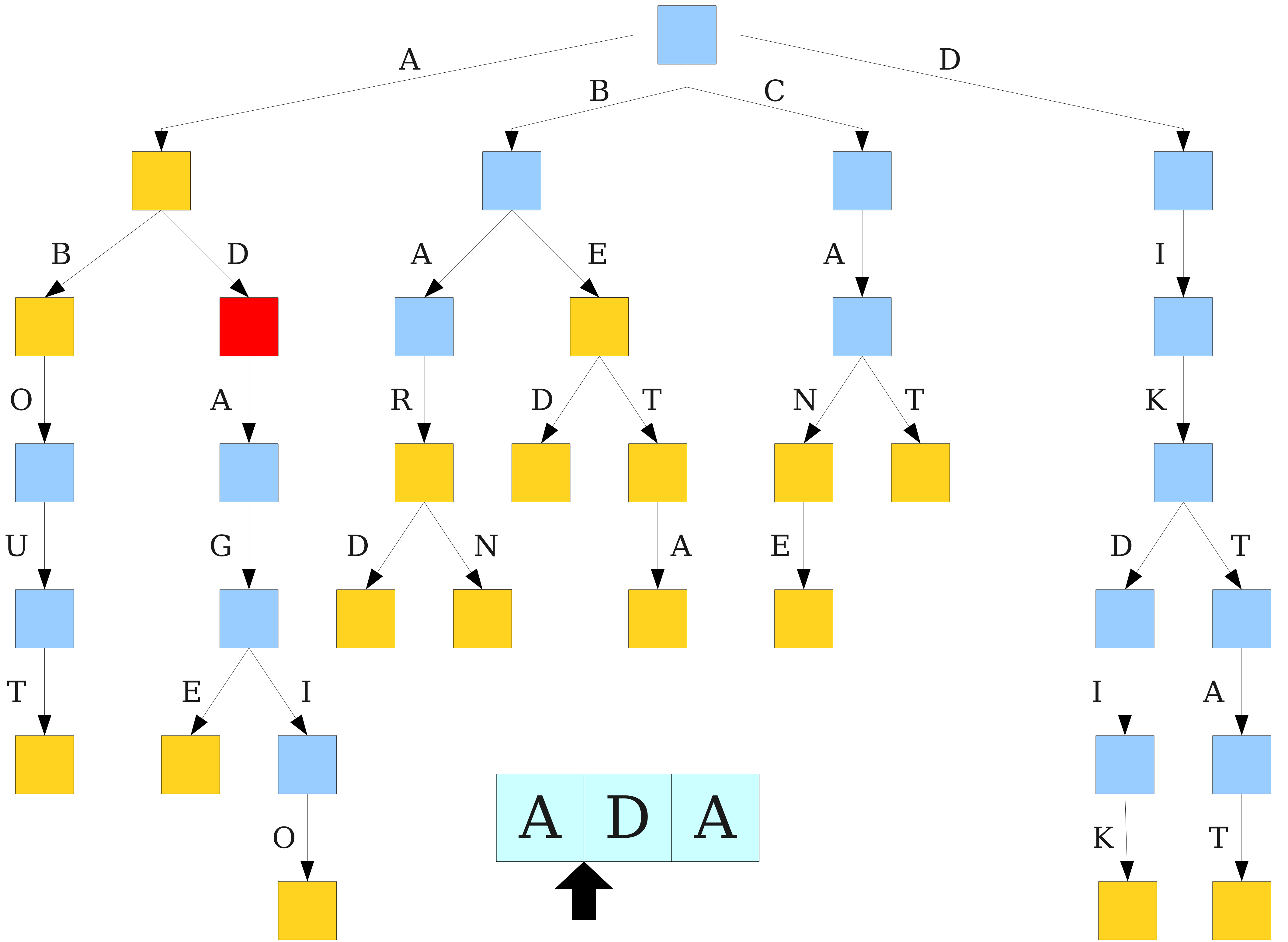

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A
D
A

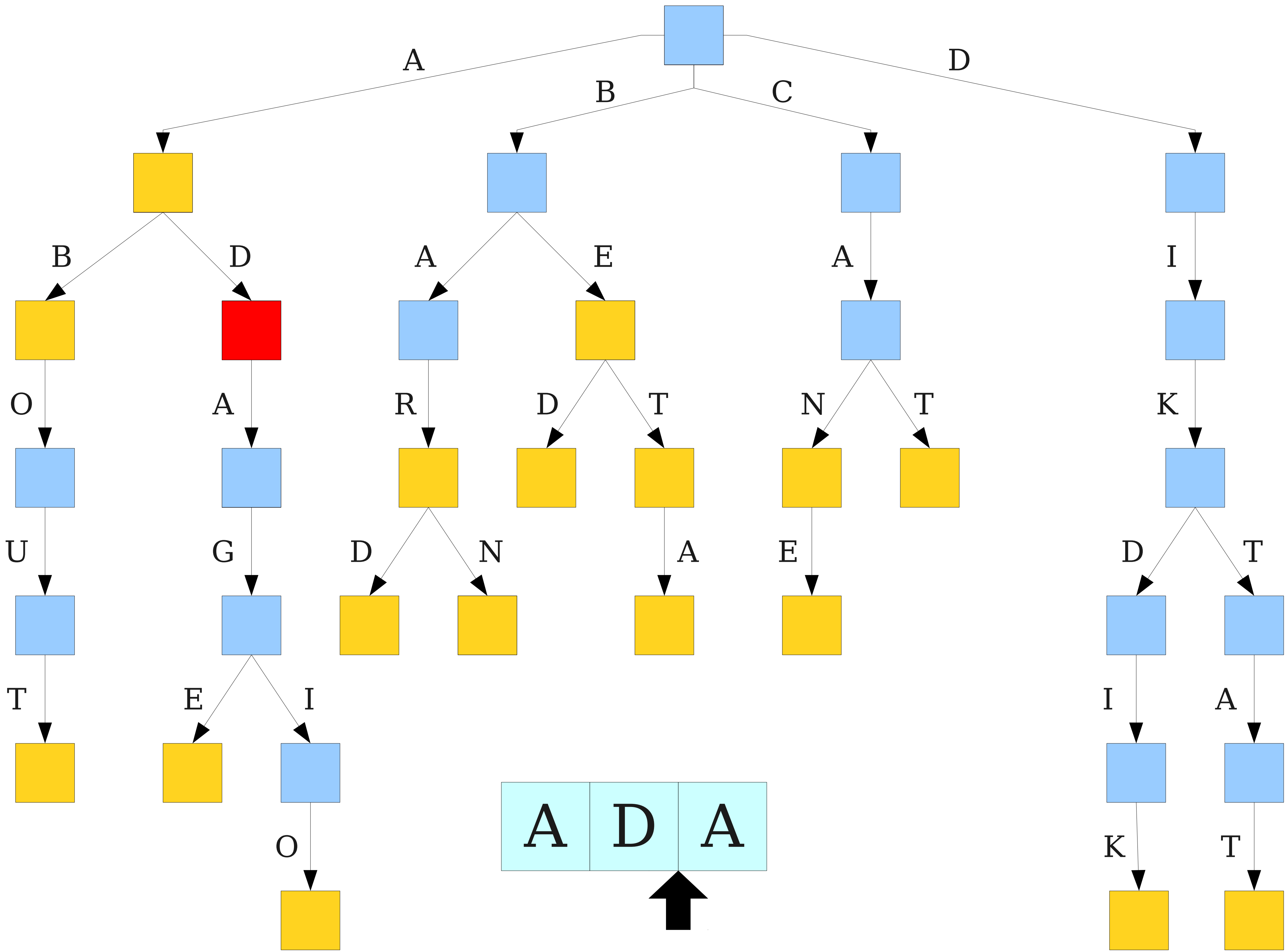

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A
D
A


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

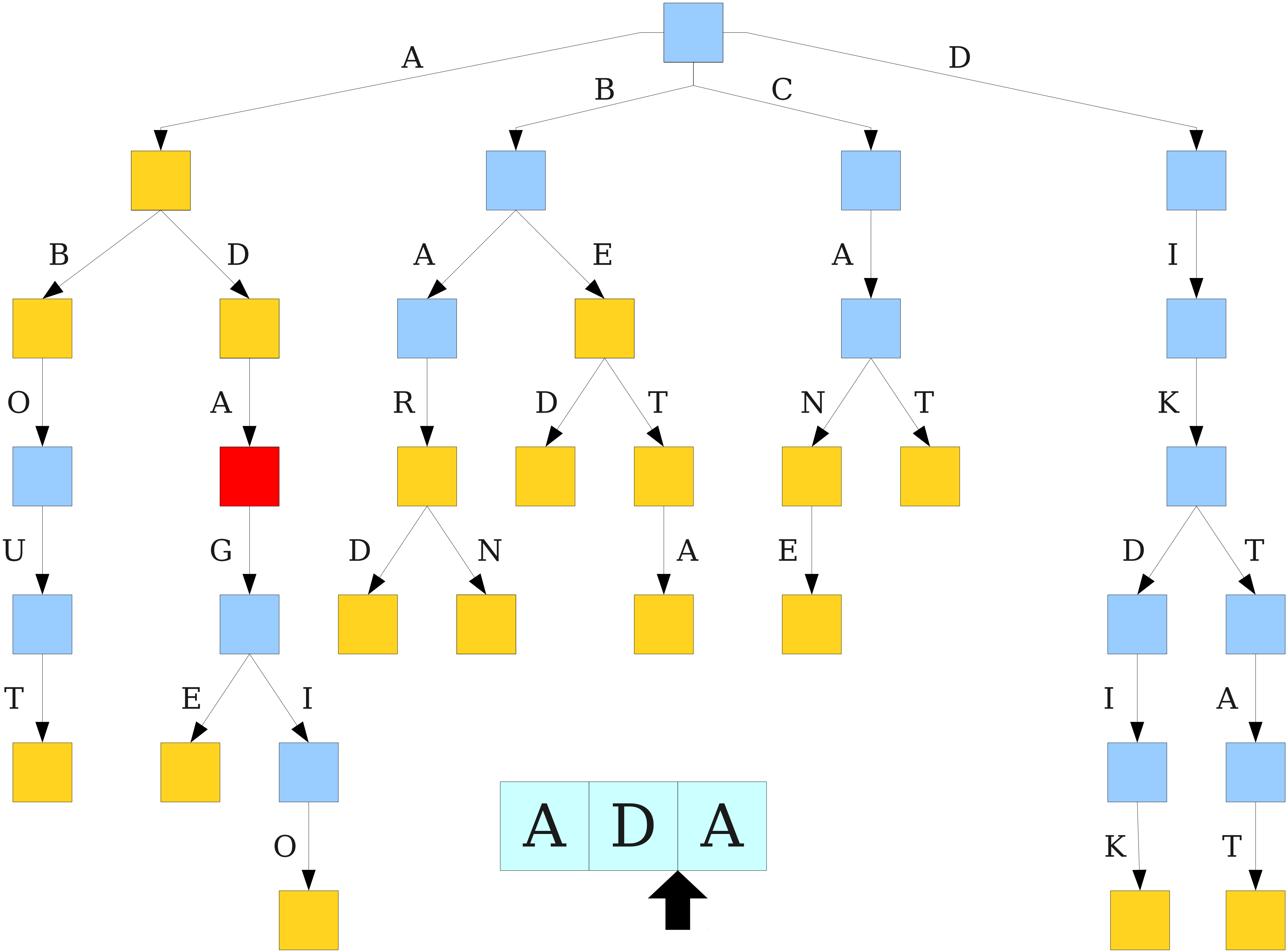


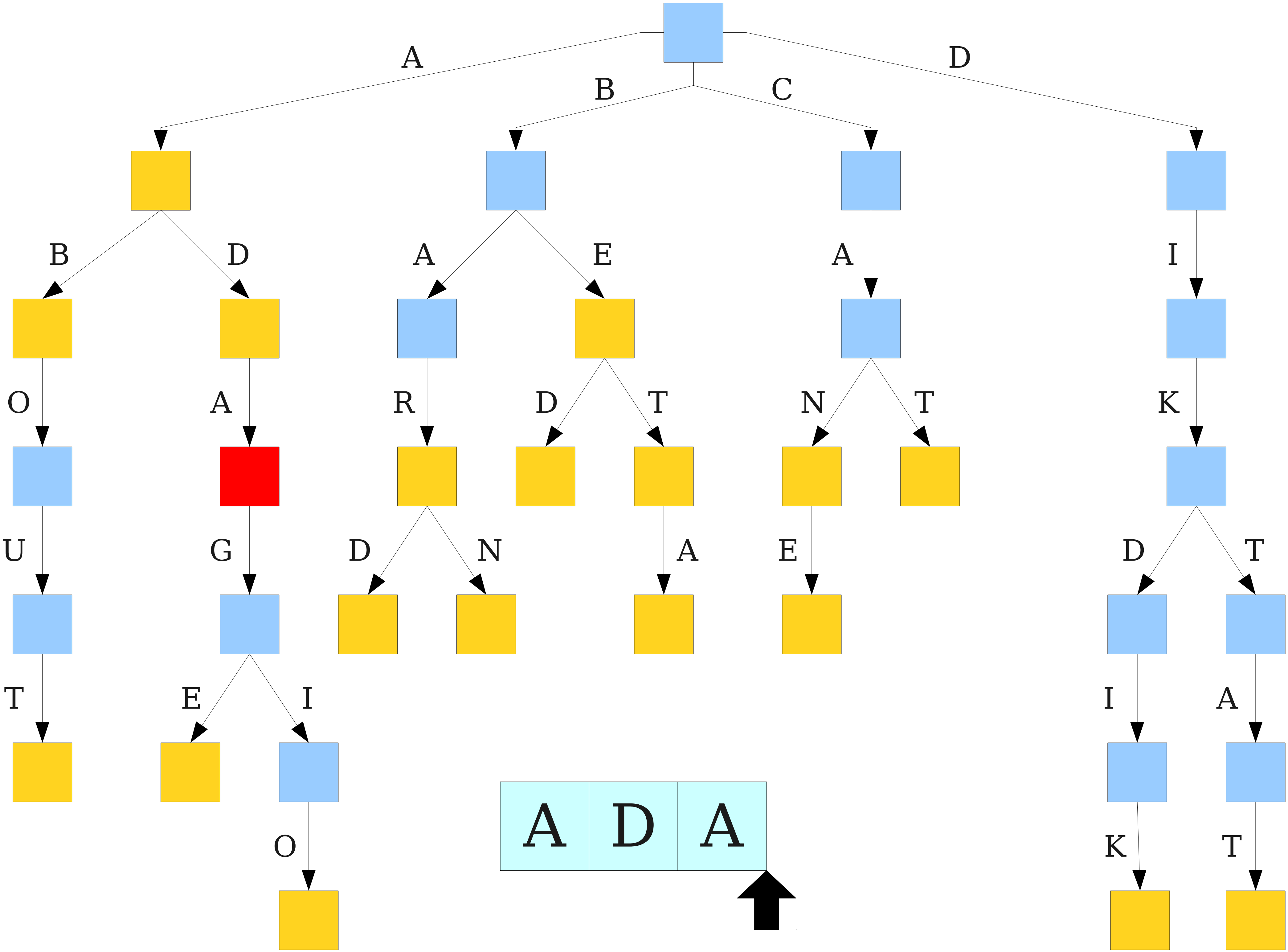
A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A D A

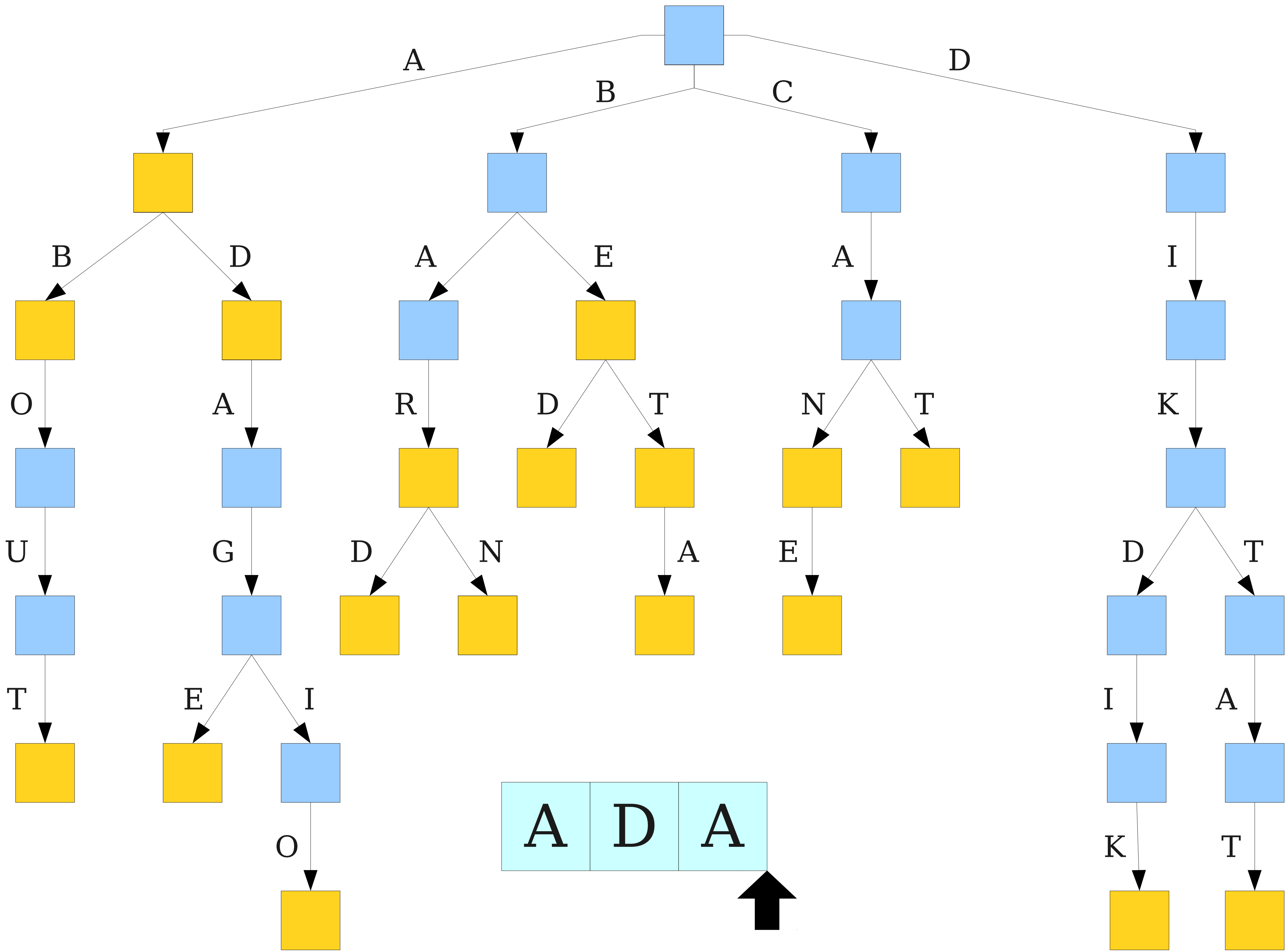

A
D
B
C
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A D A

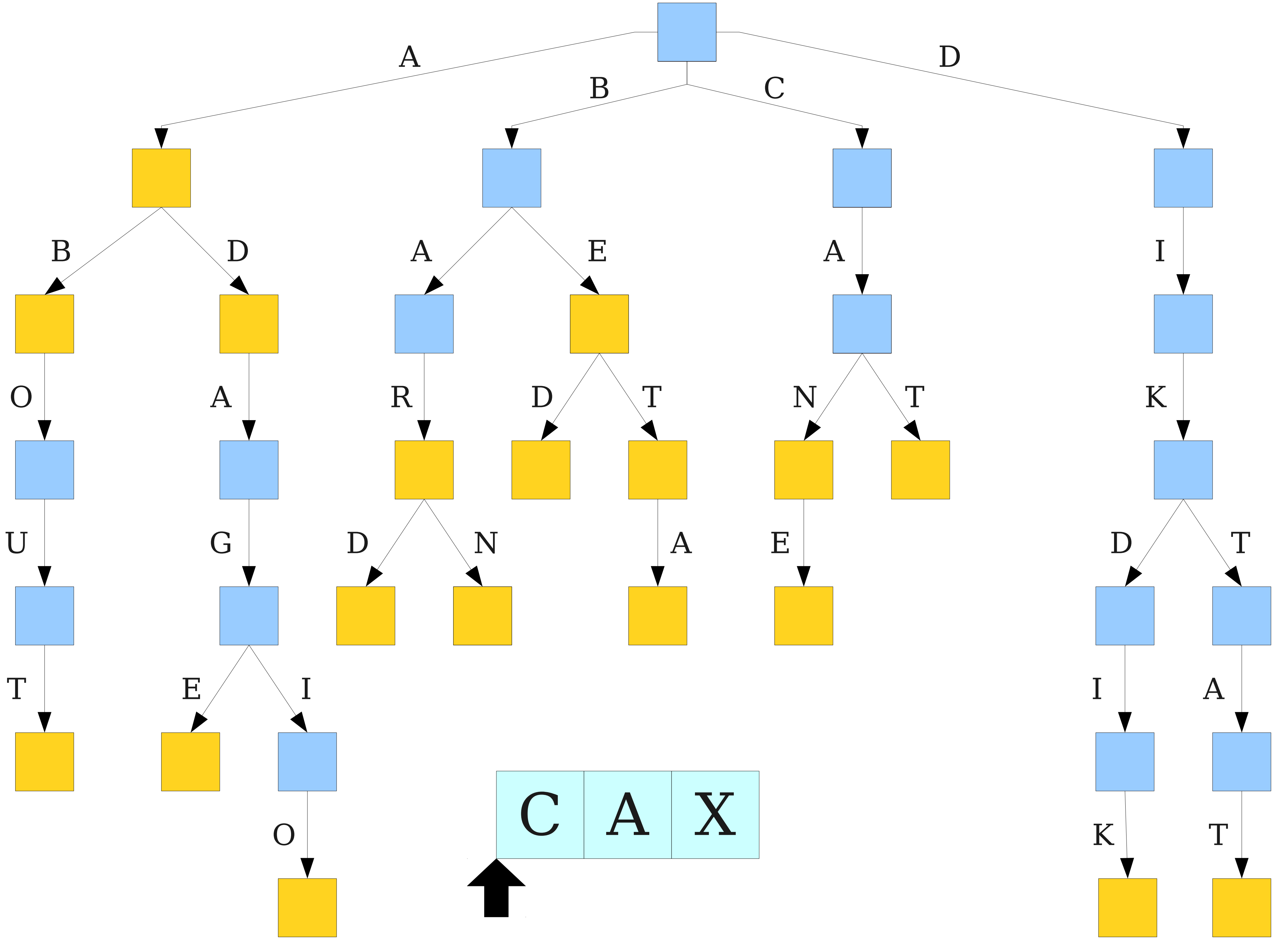


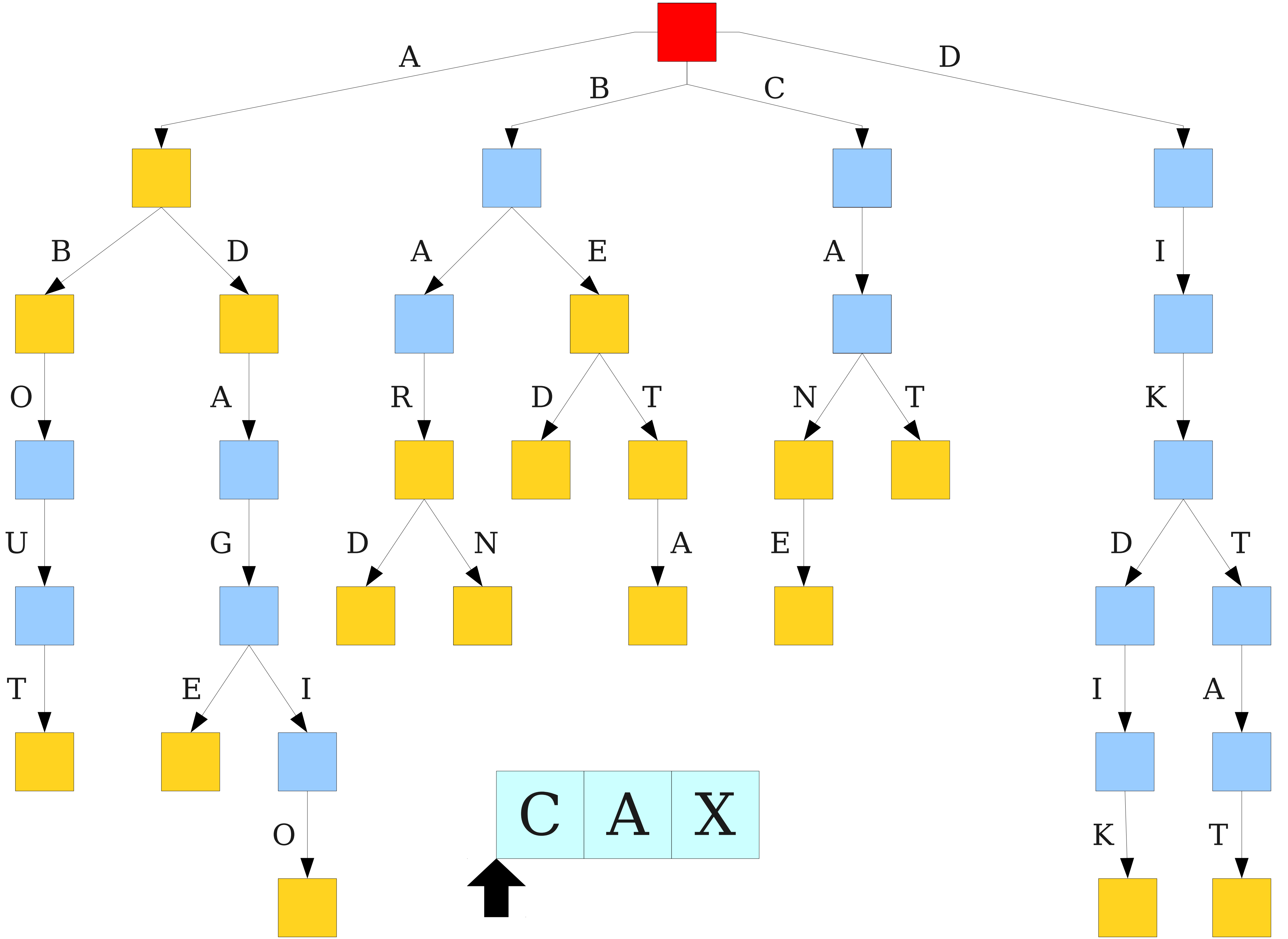
A
D
B
C
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A D A


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
A D A

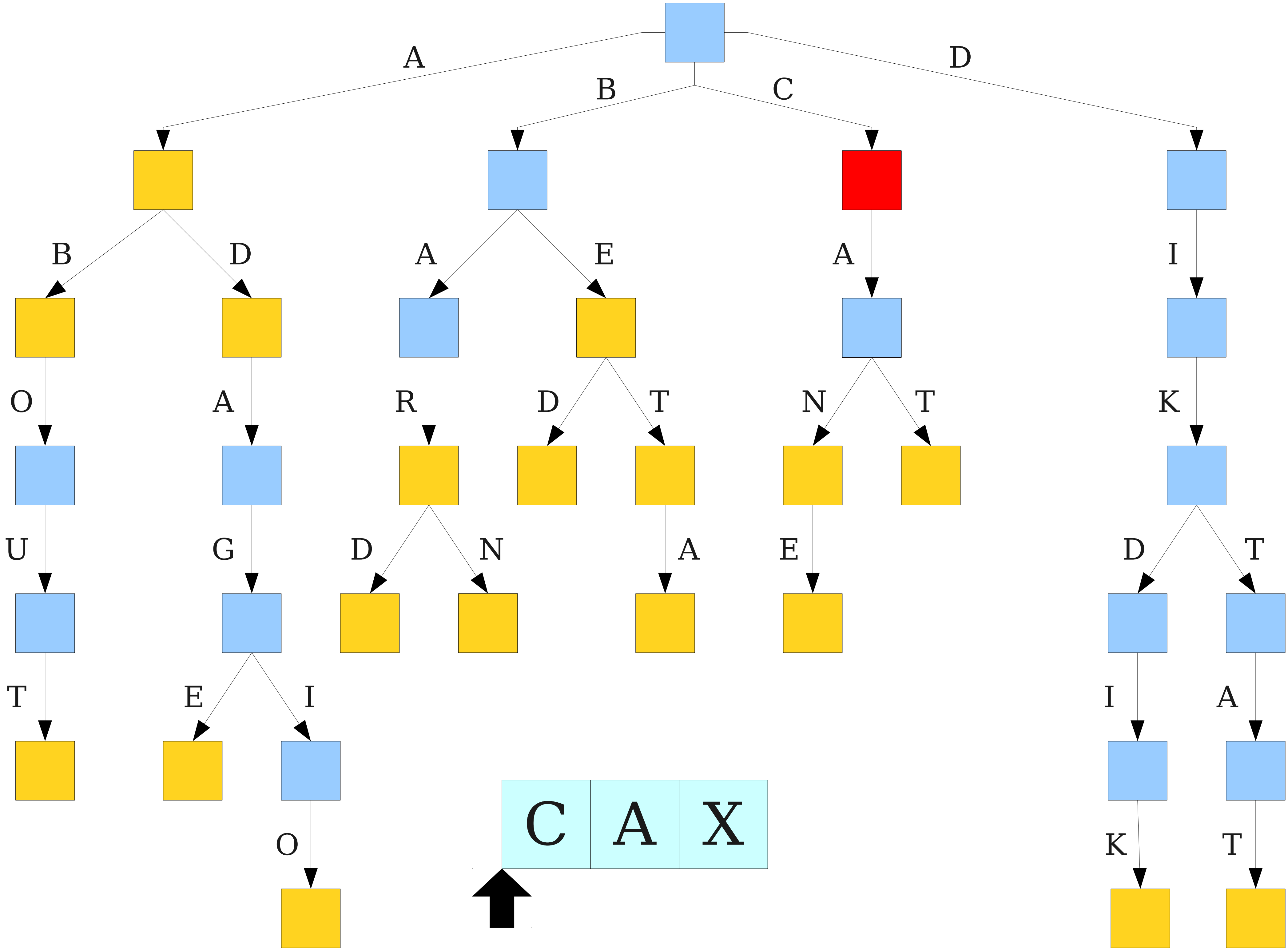

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A D A

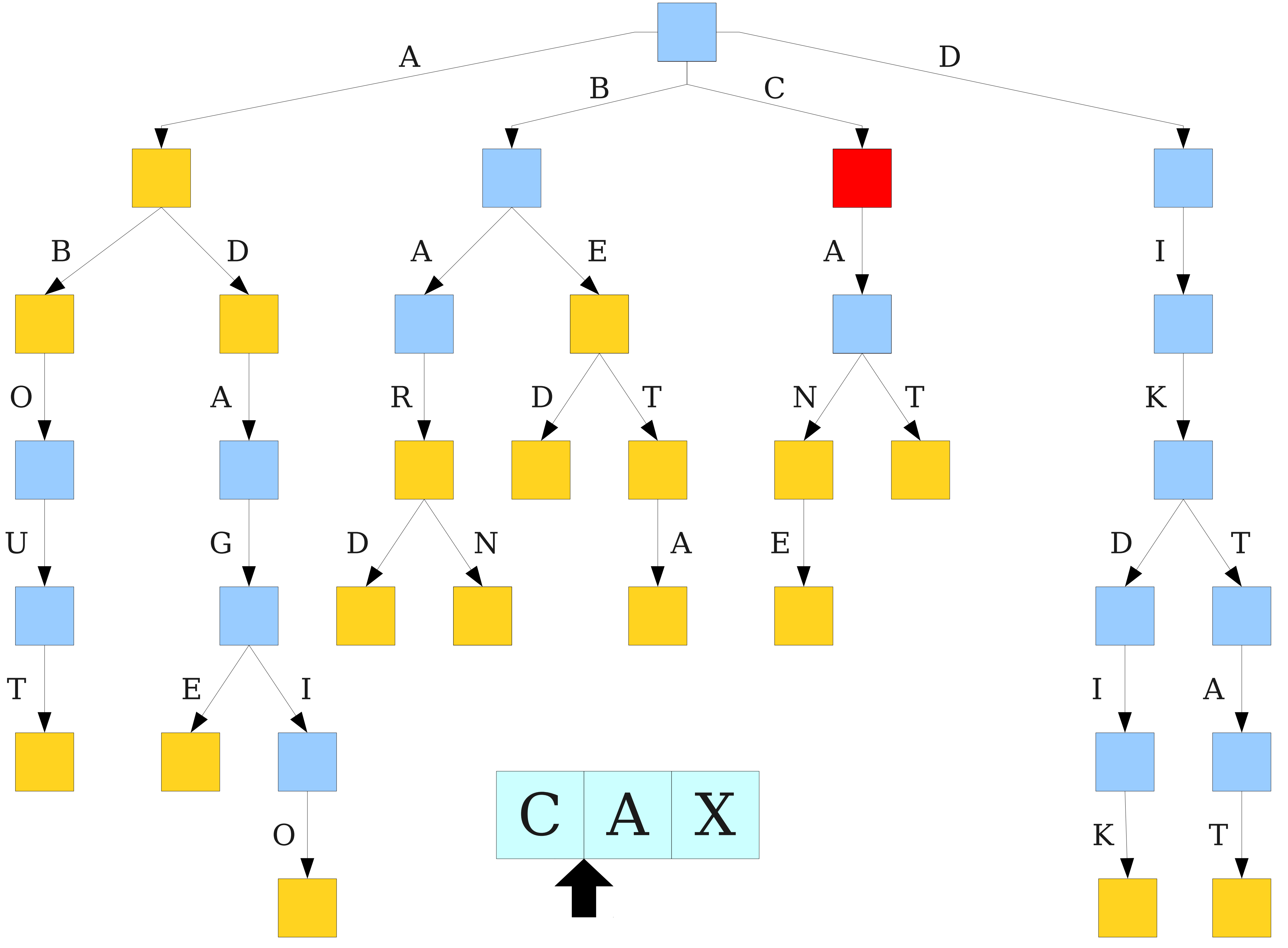

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
A D A


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
C
A
X

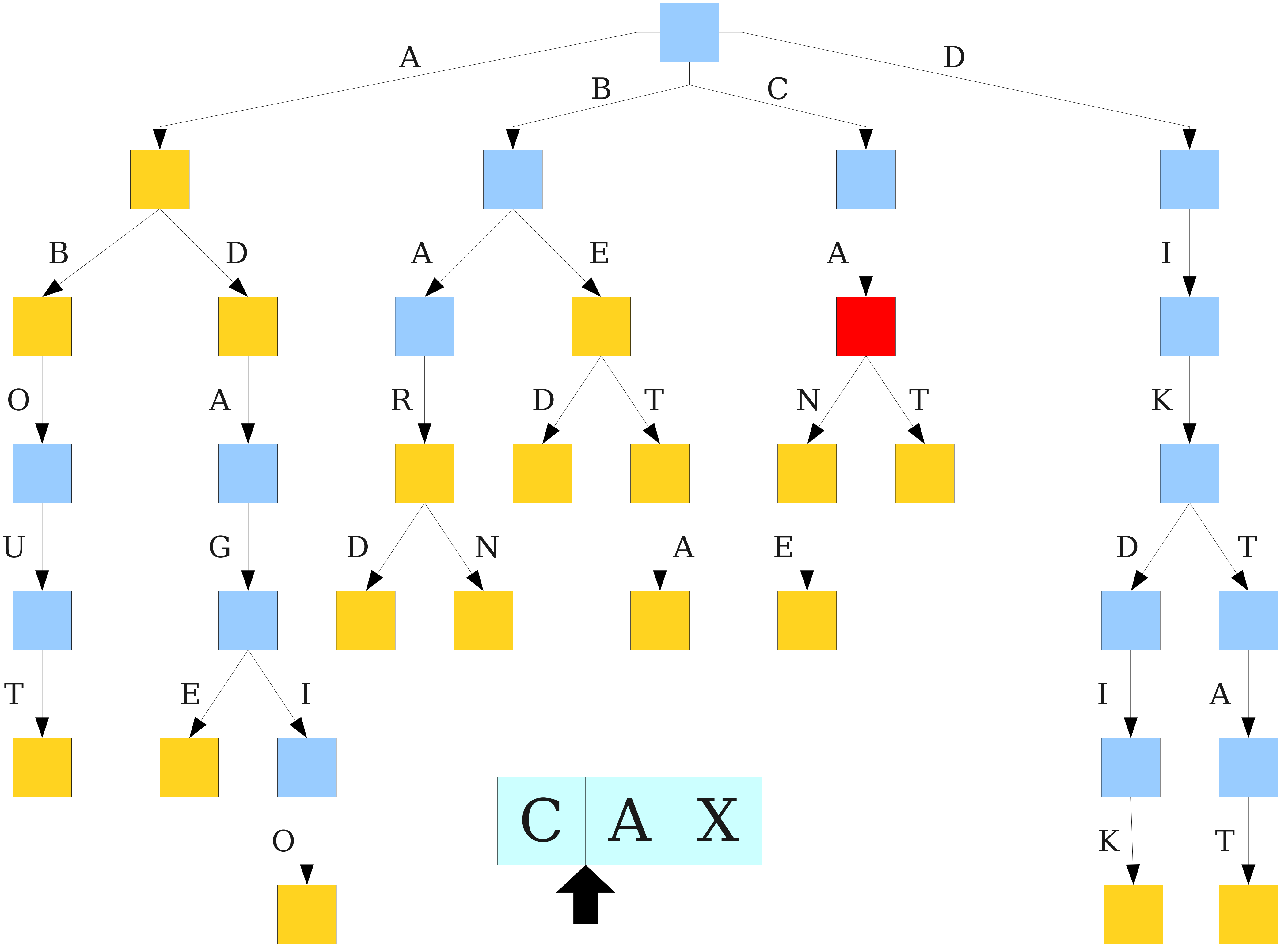


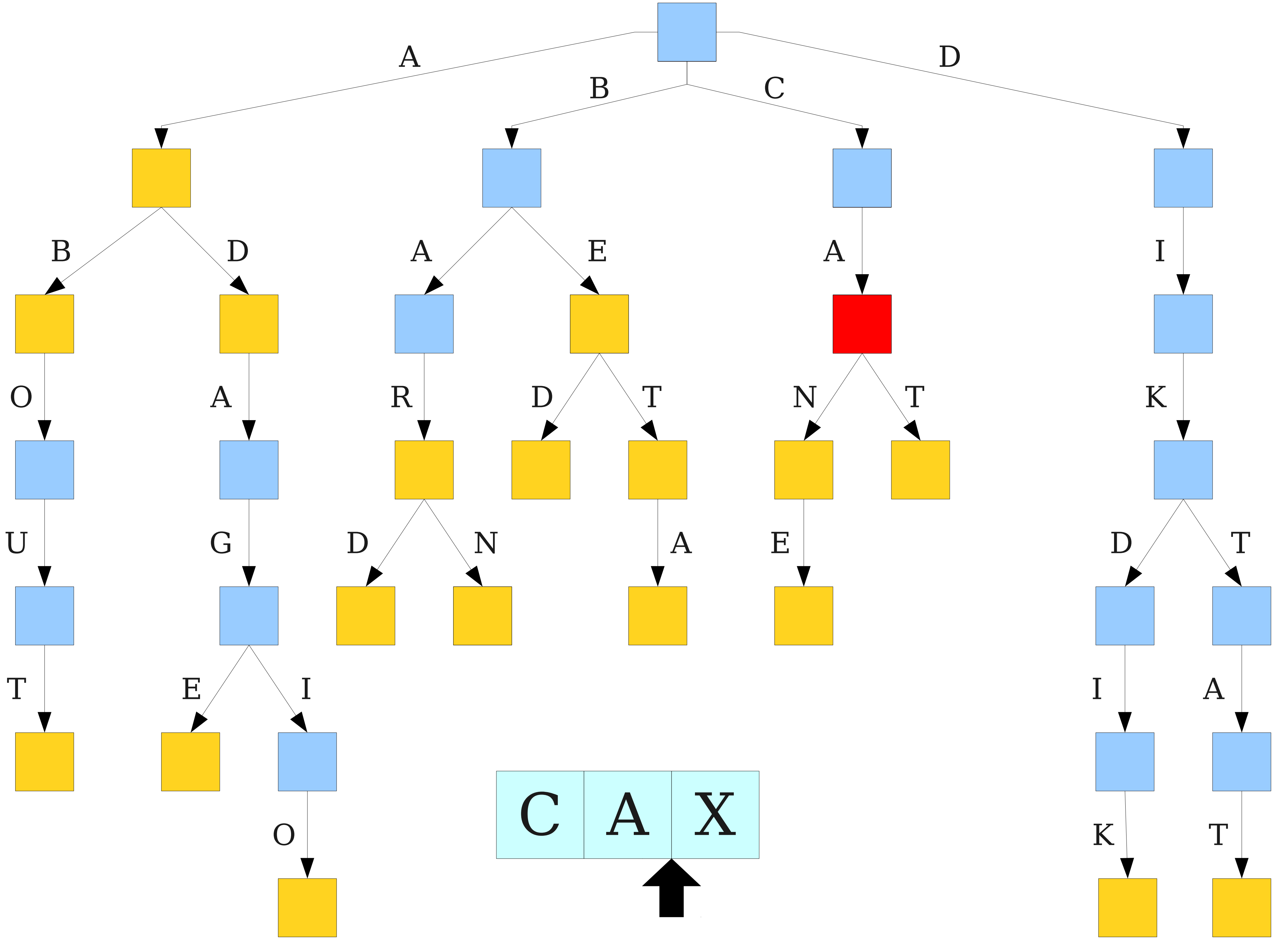
A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
C
A
X


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

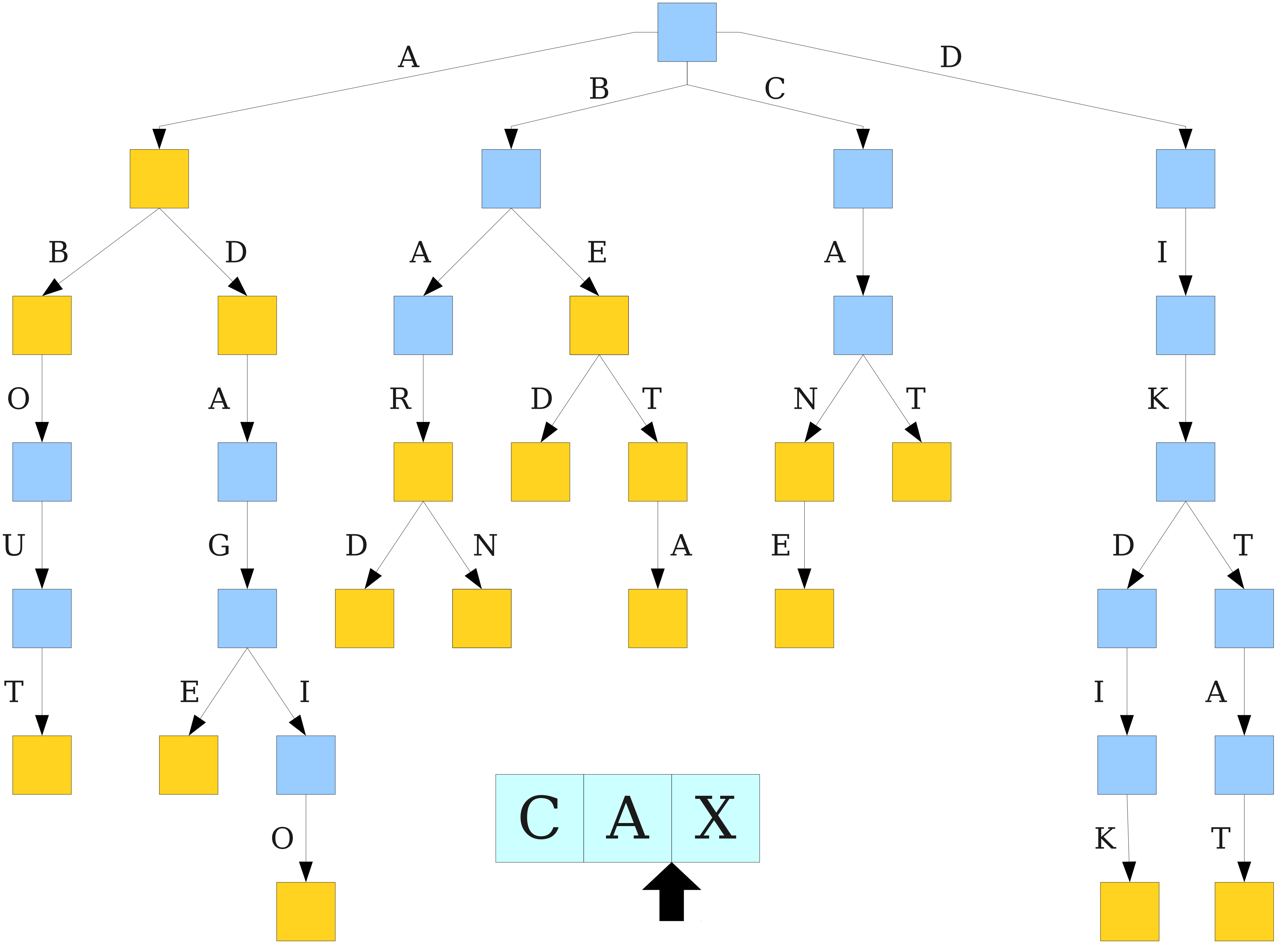


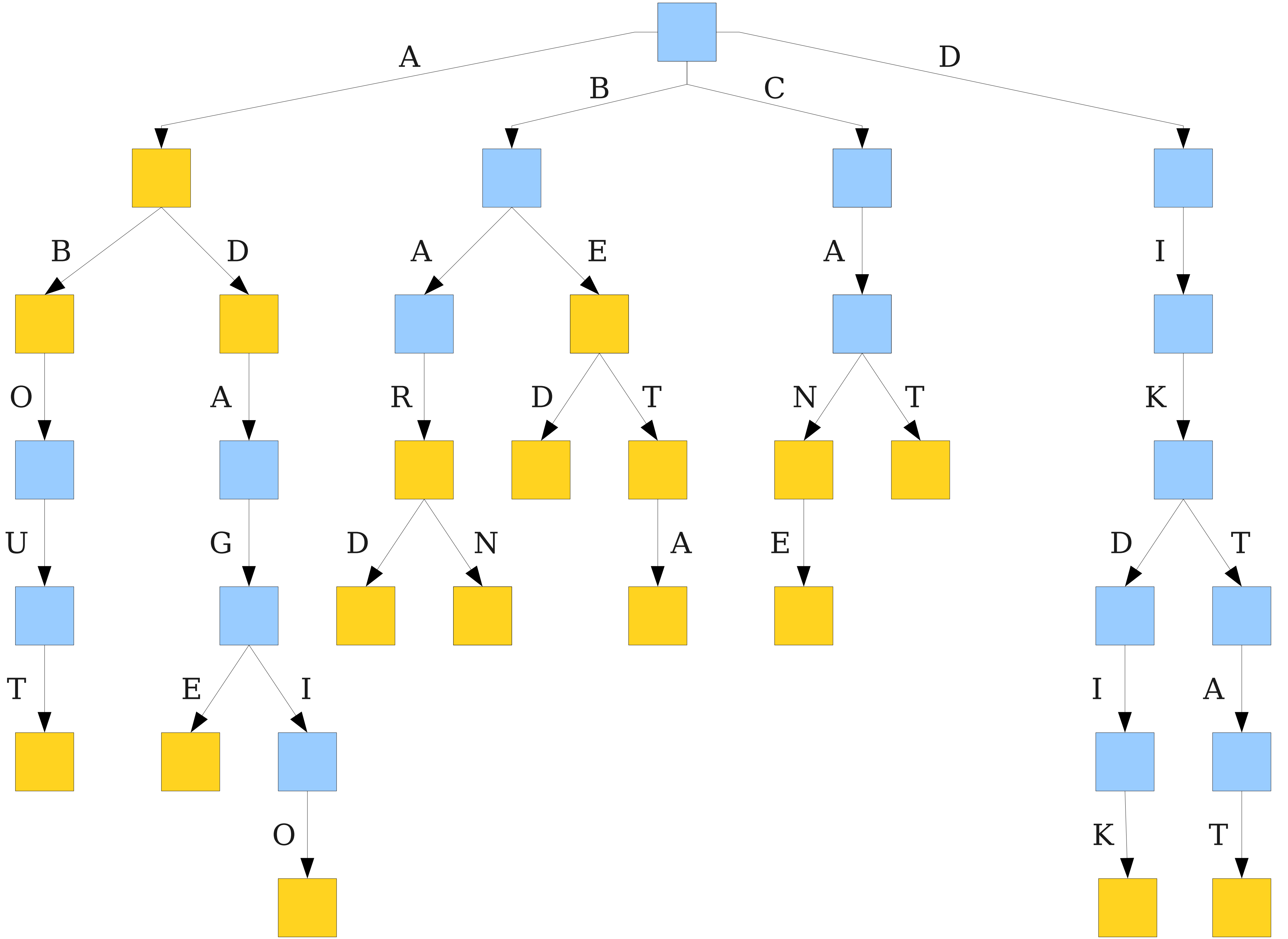
A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
C
A
X


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.”
  - Because... that's totally how “retrieval” is pronounced... I guess?

Tries, Formally

- Let Σ be some fixed alphabet.
- A **trie** is a tree where each node stores
  - A bit indicating whether the string spelled out to this point is in the set, and
  - An array of |Σ| pointers, one for each character.
- Each node *x* corresponds to some string given by the path traced from the root to that node.

Tries, Formally

Let Σ be some fixed alphabet.

A **trie** is a tree where each node stores

- A bit indicating whether the string spelled out to this point is in the set, and
- An array of |Σ| pointers, one for each character.

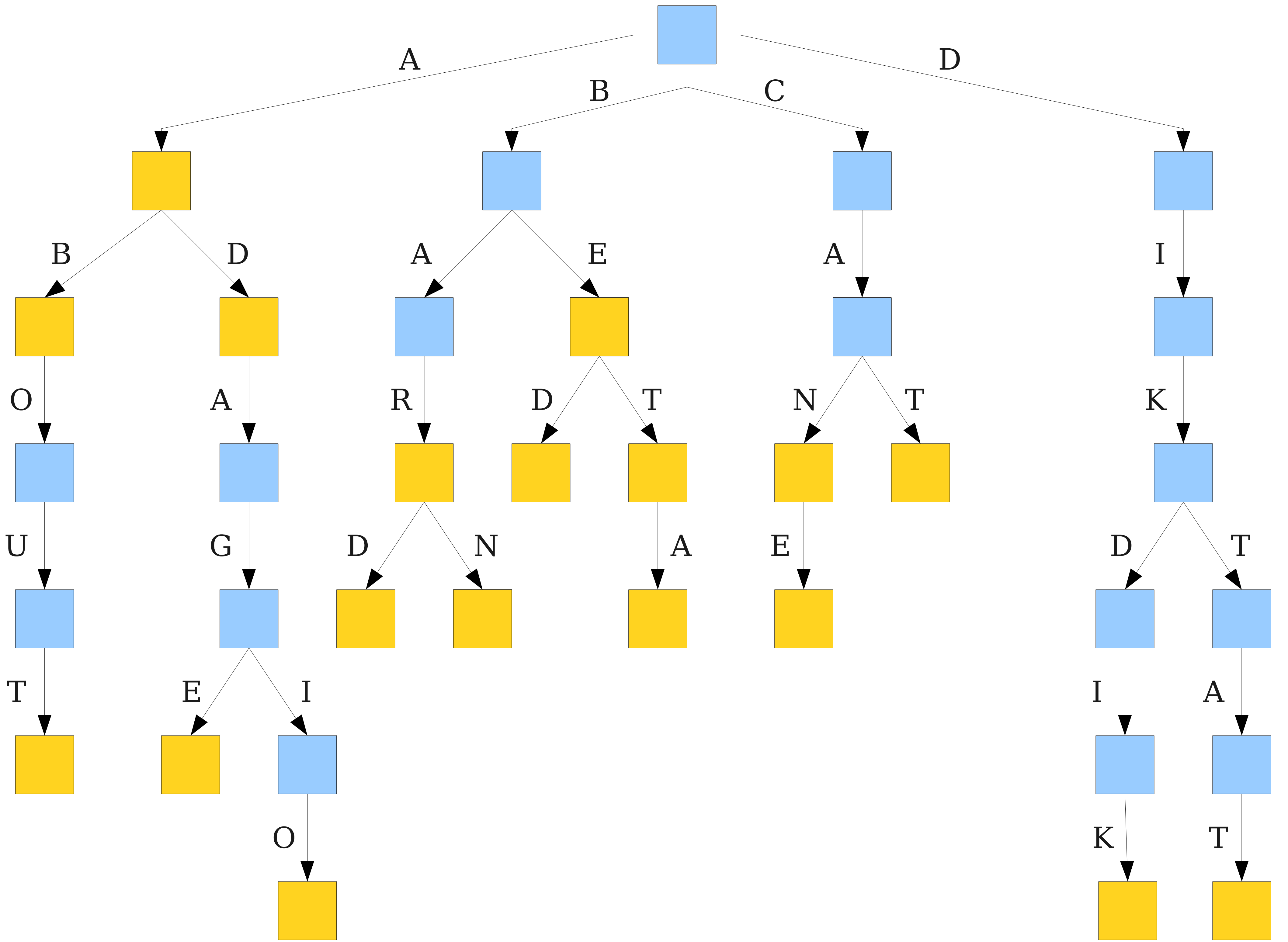
Each node *x* corresponds to some string given by the path traced from the root to that node.

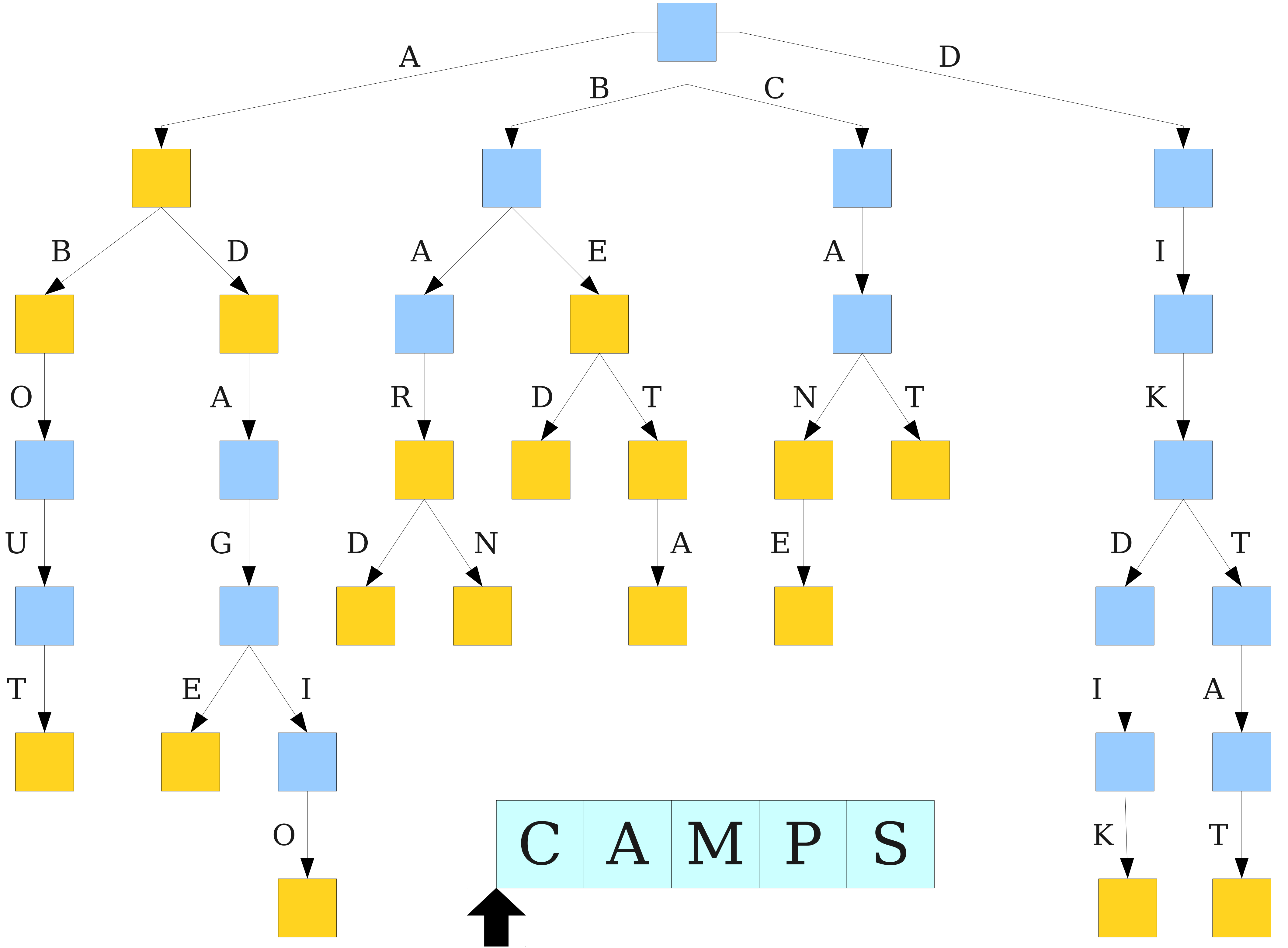
Trie Efficiency

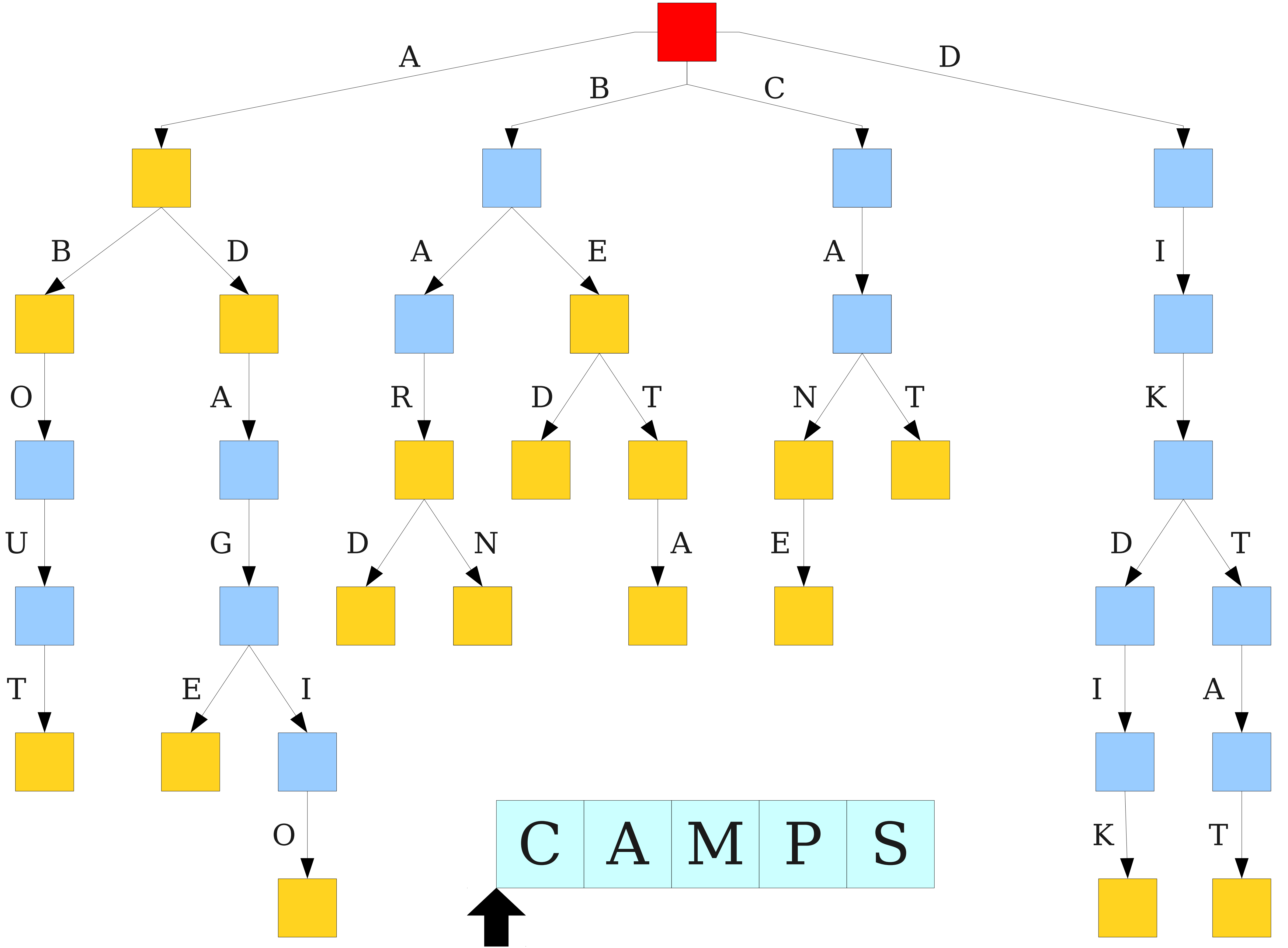
- What is the cost of looking up a string *w* in a trie?
- Follow at most $|w|$ pointers to get to the node for *w*, if it exists.
- Each pointer can be looked up in time O(1).
- Total time: $O(|w|)$.
- **Lookup time is independent of the number of strings in the trie!**

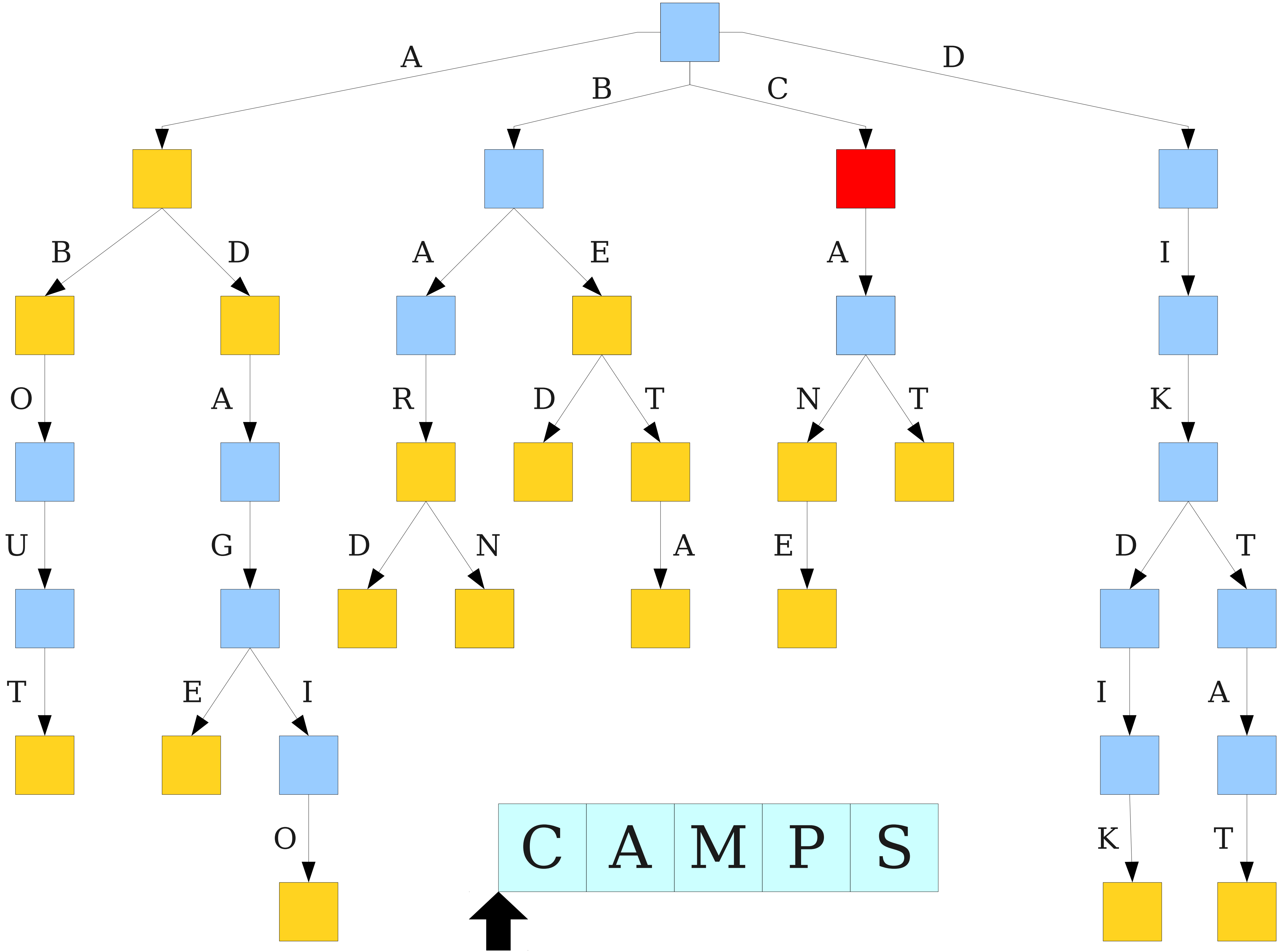
Inserting into a Trie

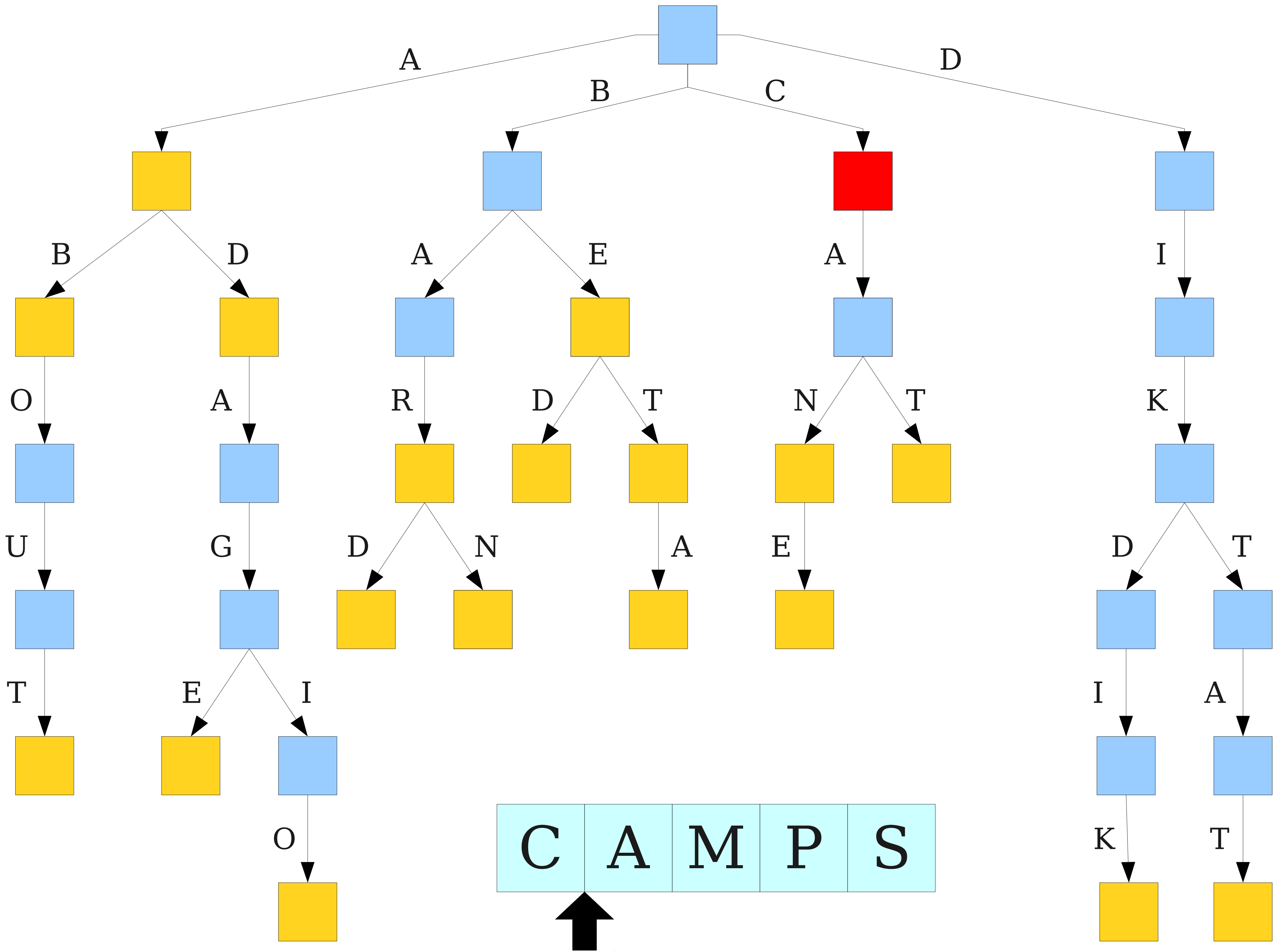
- Proceed before as if doing an normal lookup, adding in new nodes as needed.
- Set the “is word” bit in the final node visited this way.

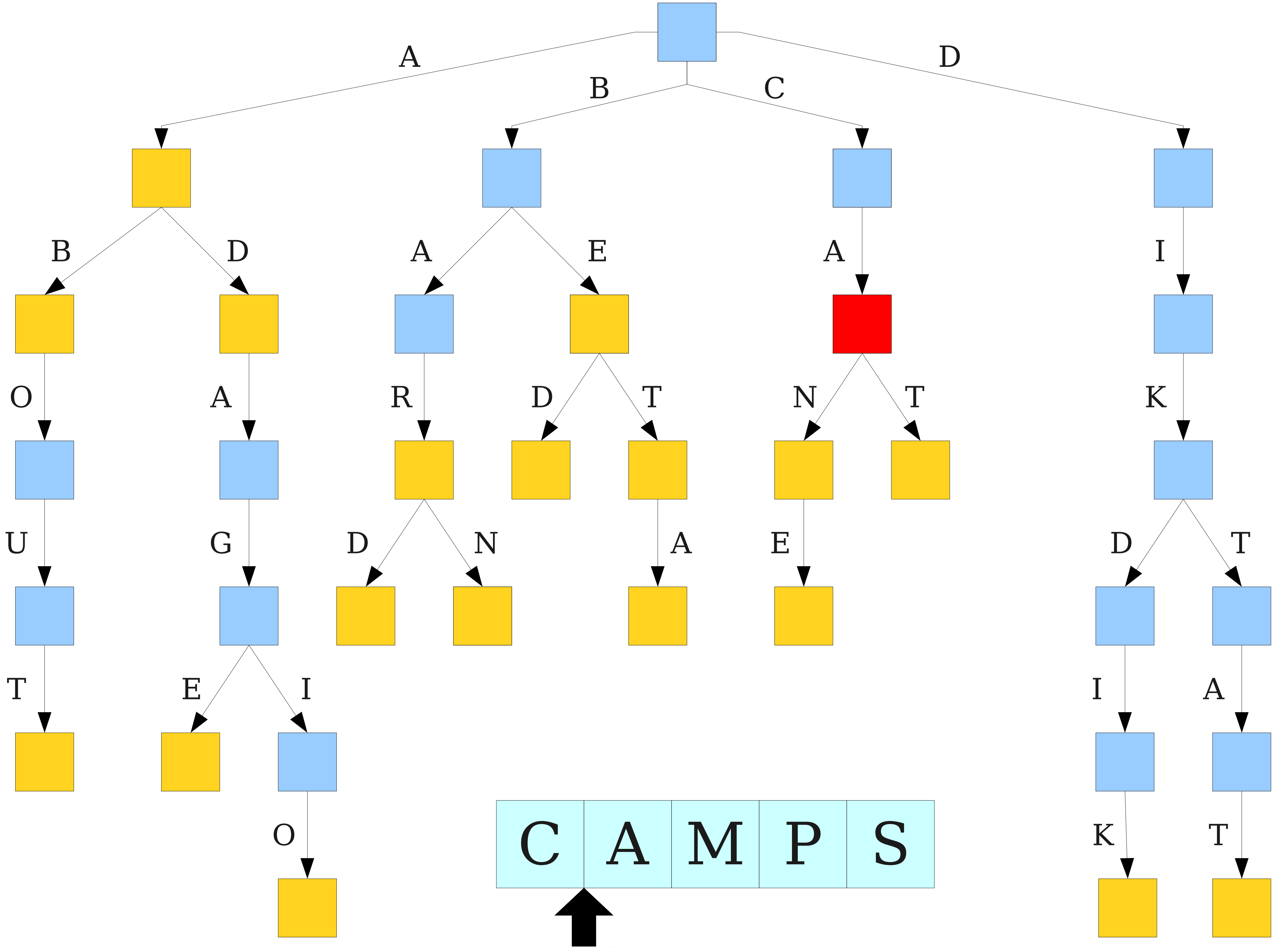

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T


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
C A M P S

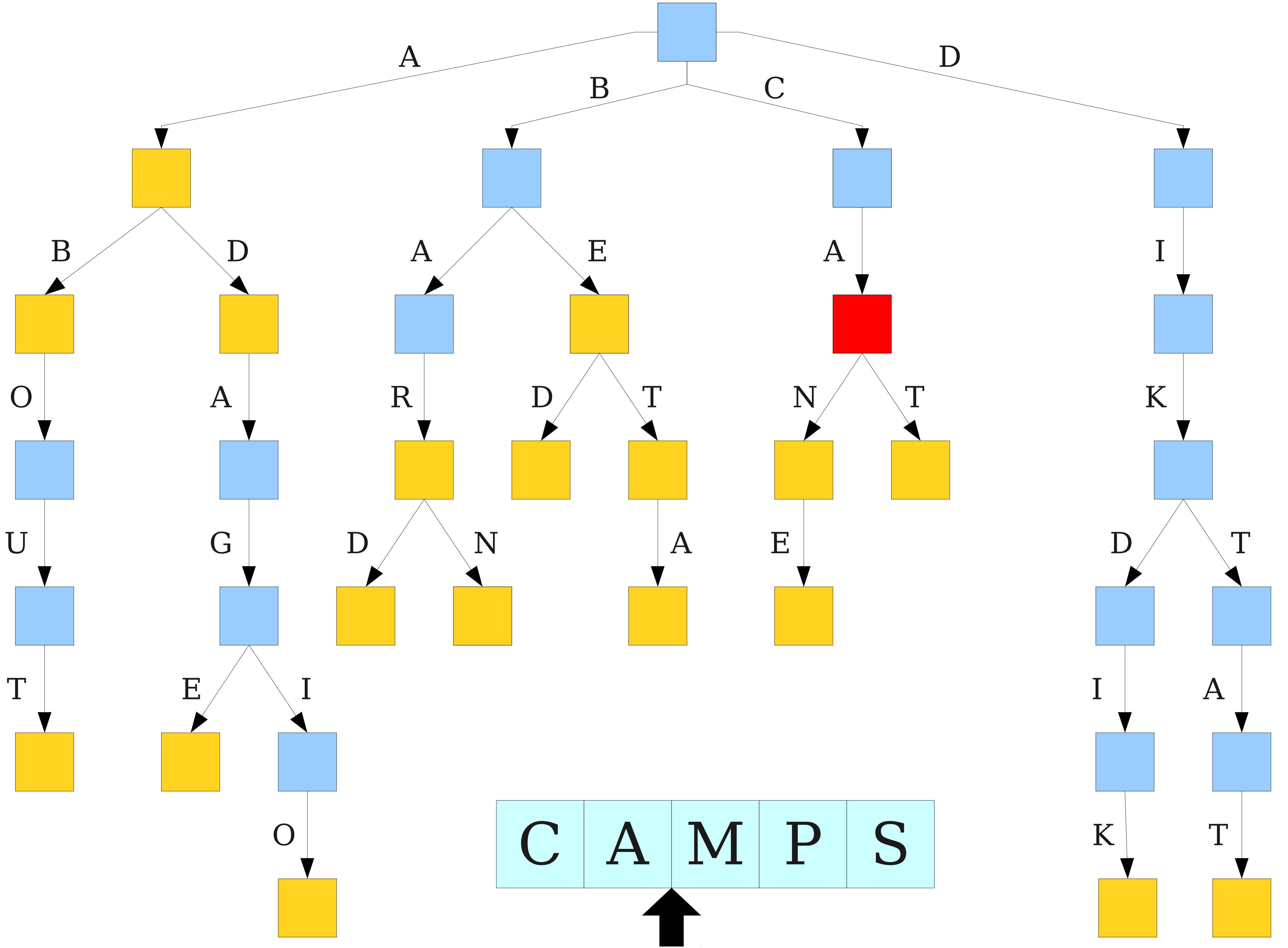

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
C A M P S
K
T


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 M P S

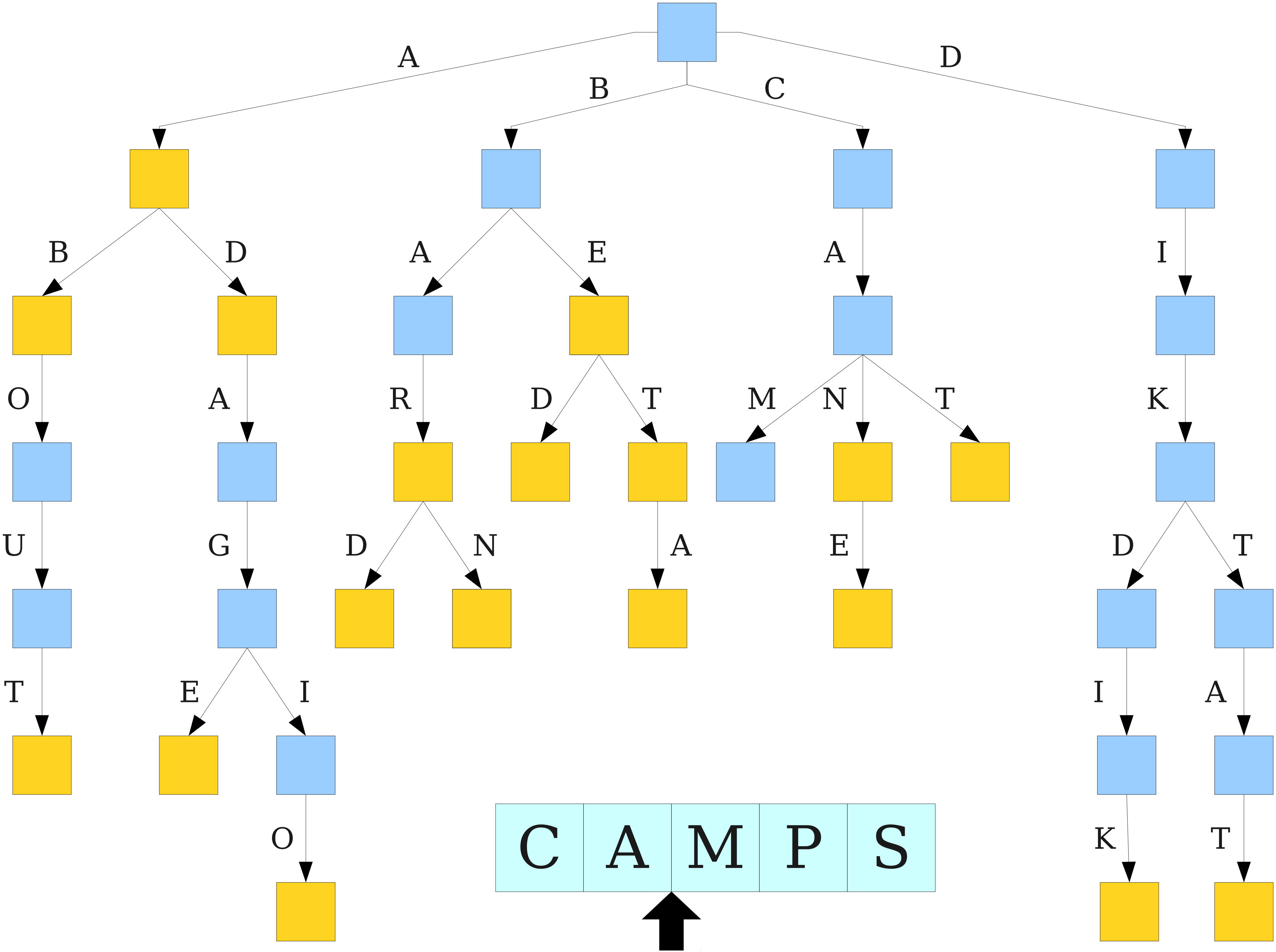

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
C A M P S

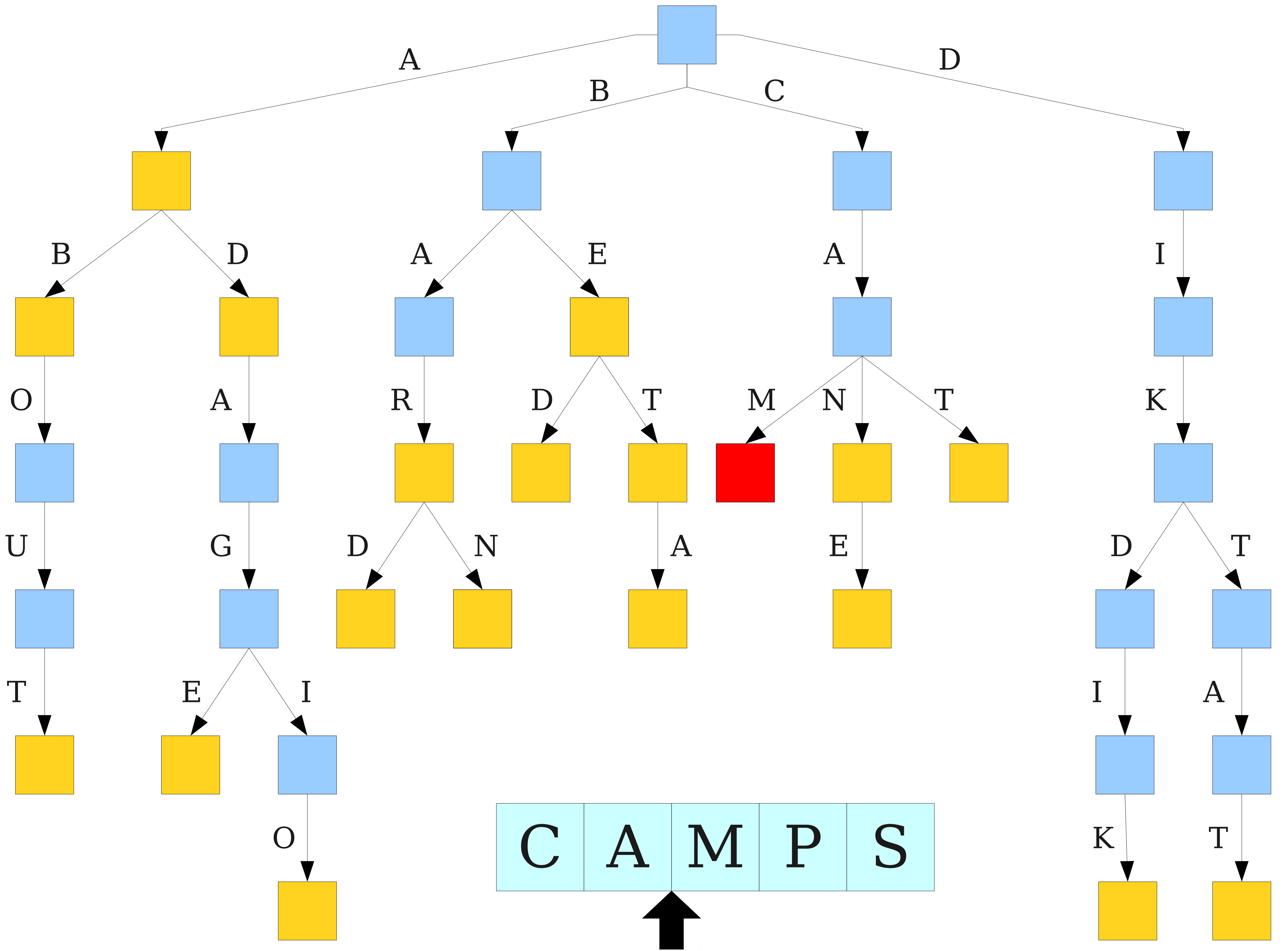


A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
C A M P S

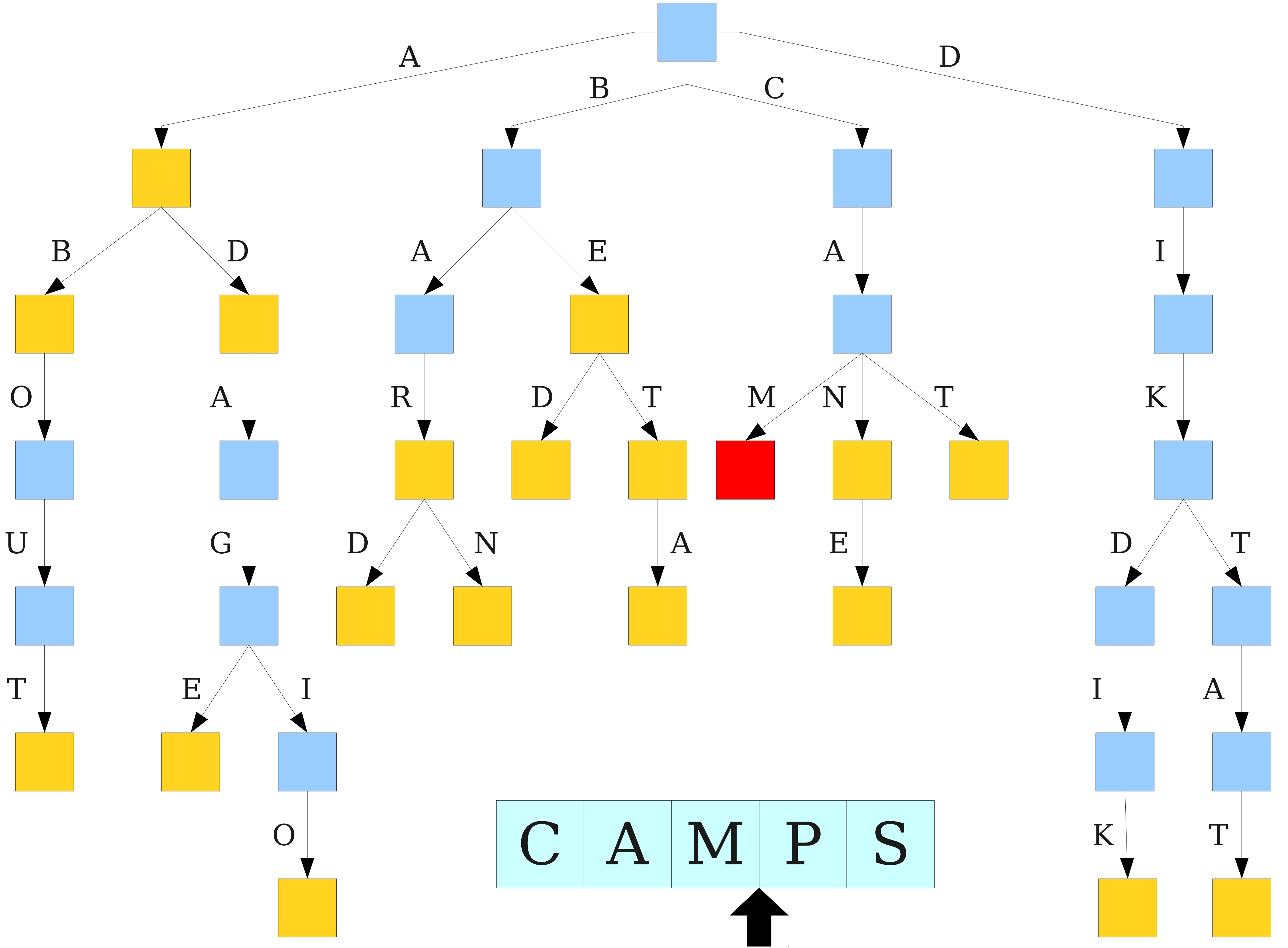


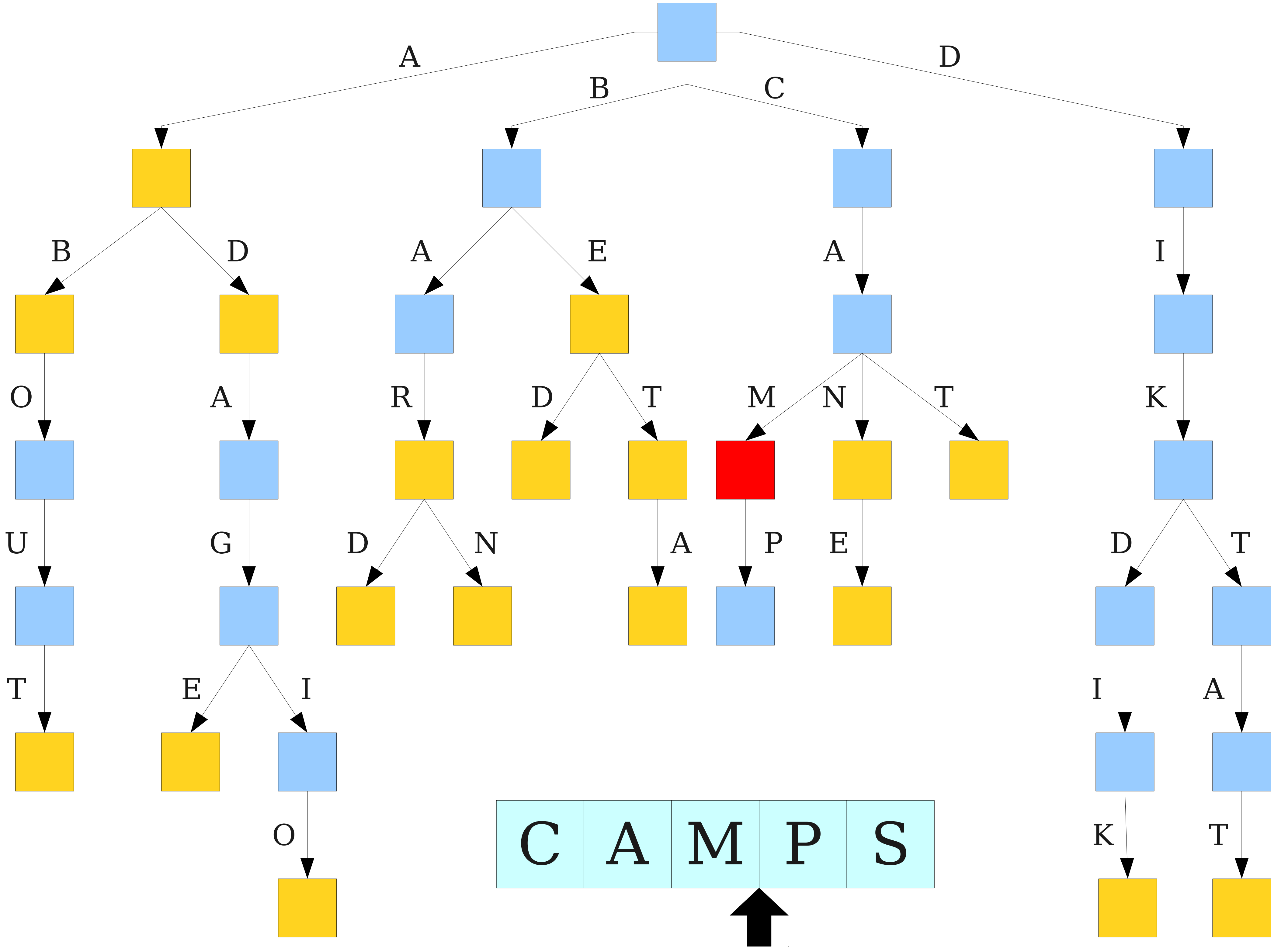
A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
C A M P S

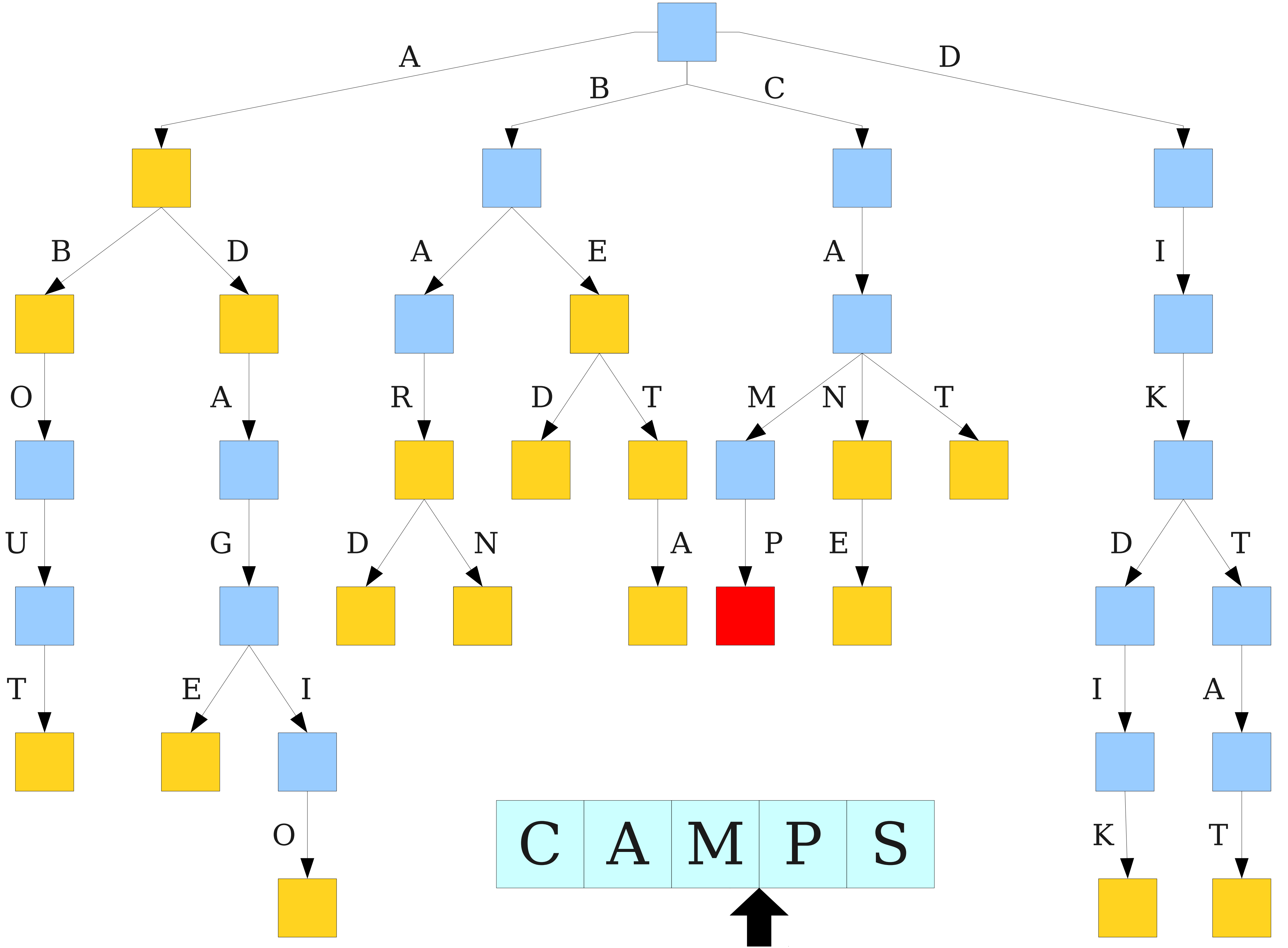

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
C
A
M
P
S

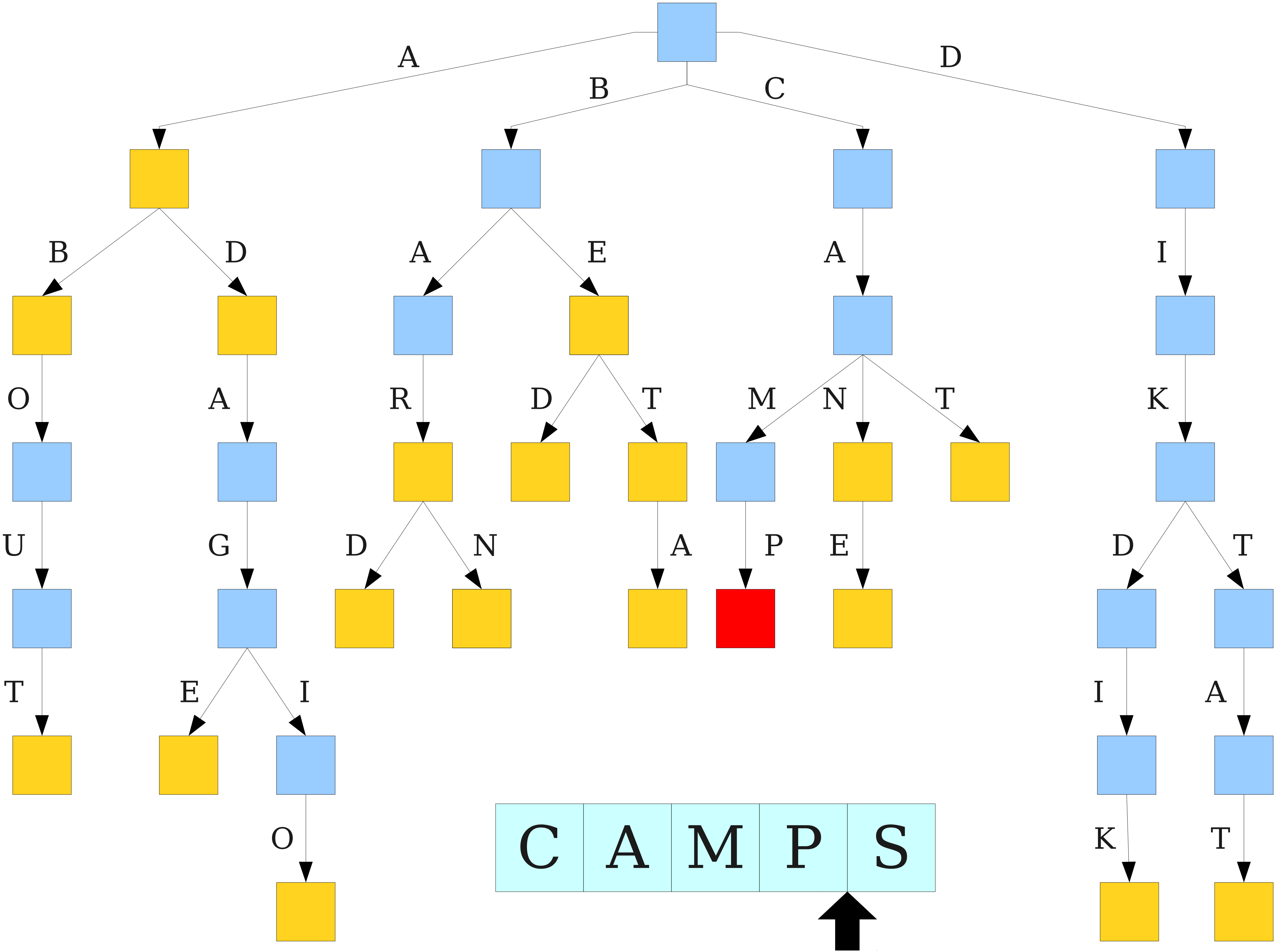


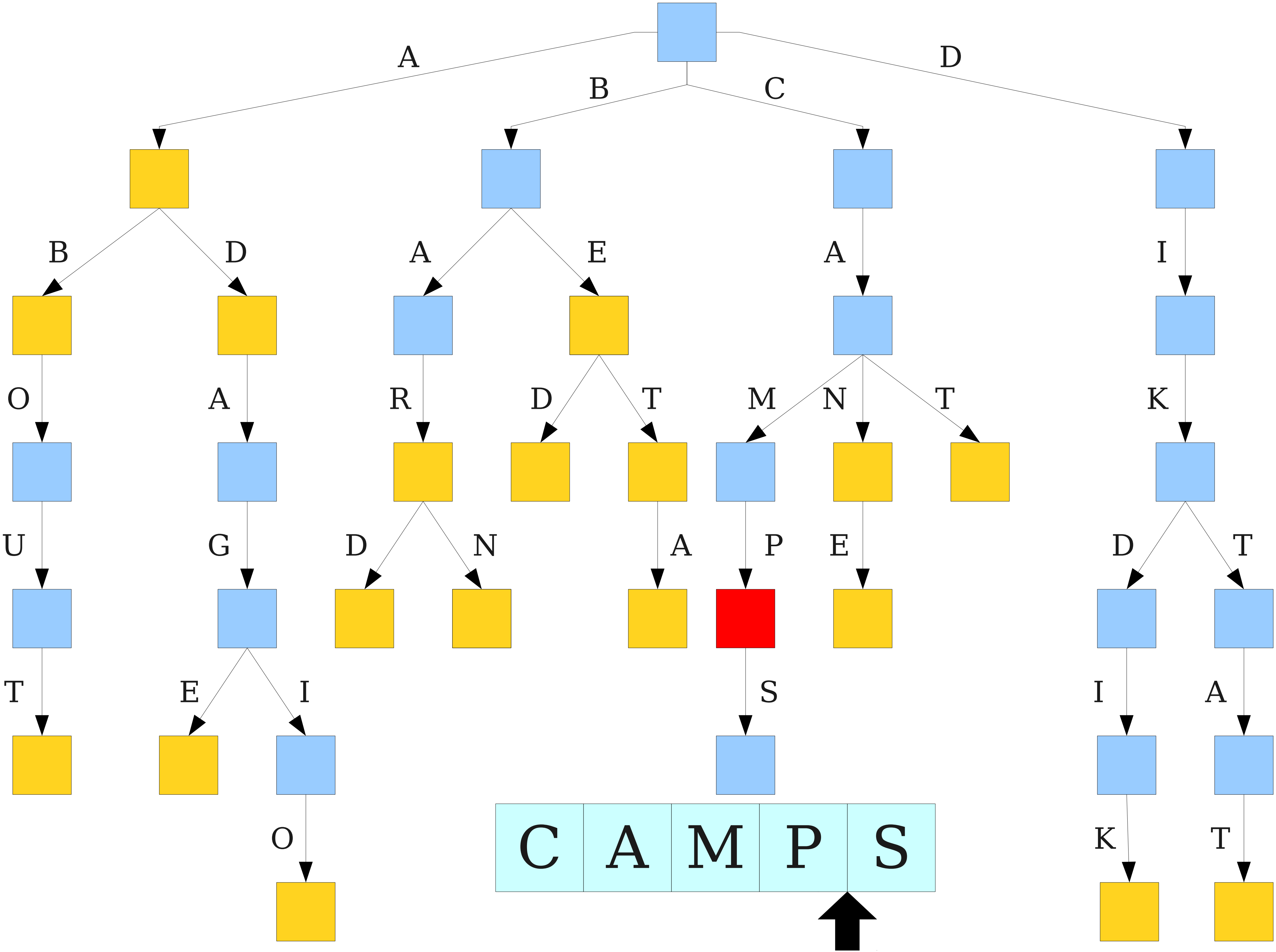
A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
C A M P S

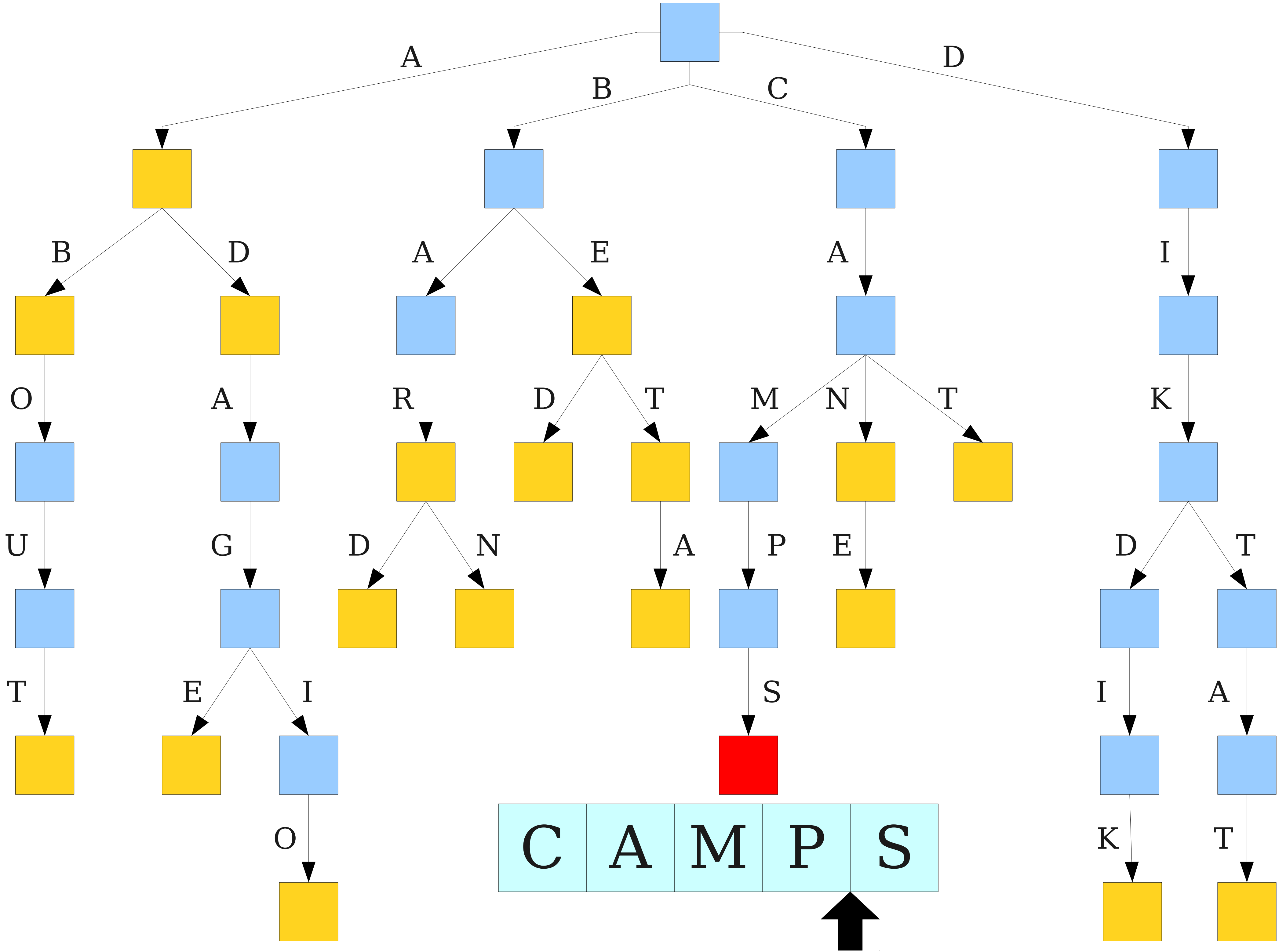

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
C A M P S


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

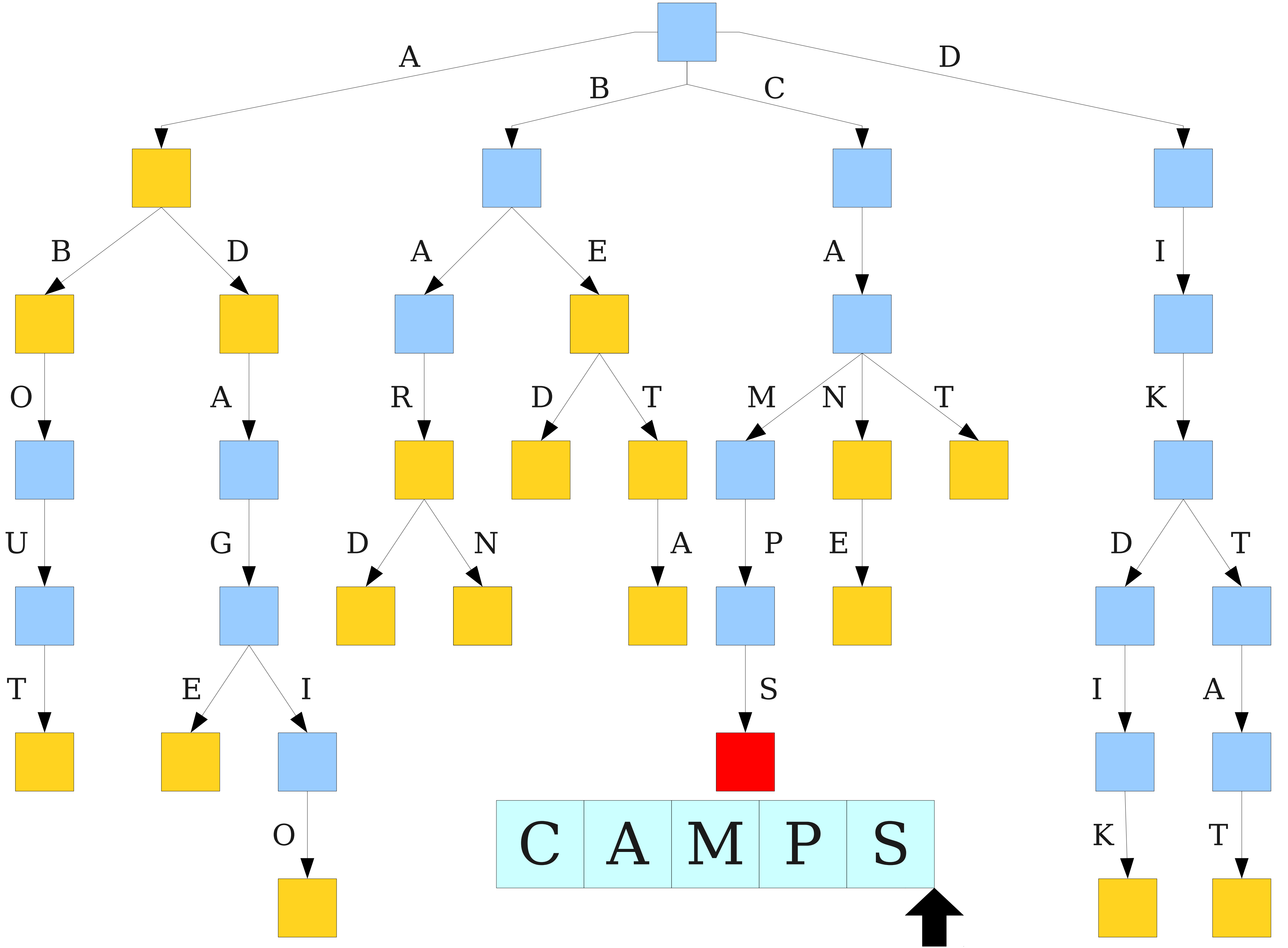

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
P
E
D
T
T
E
I
I
A
O
K
T
C A M P S

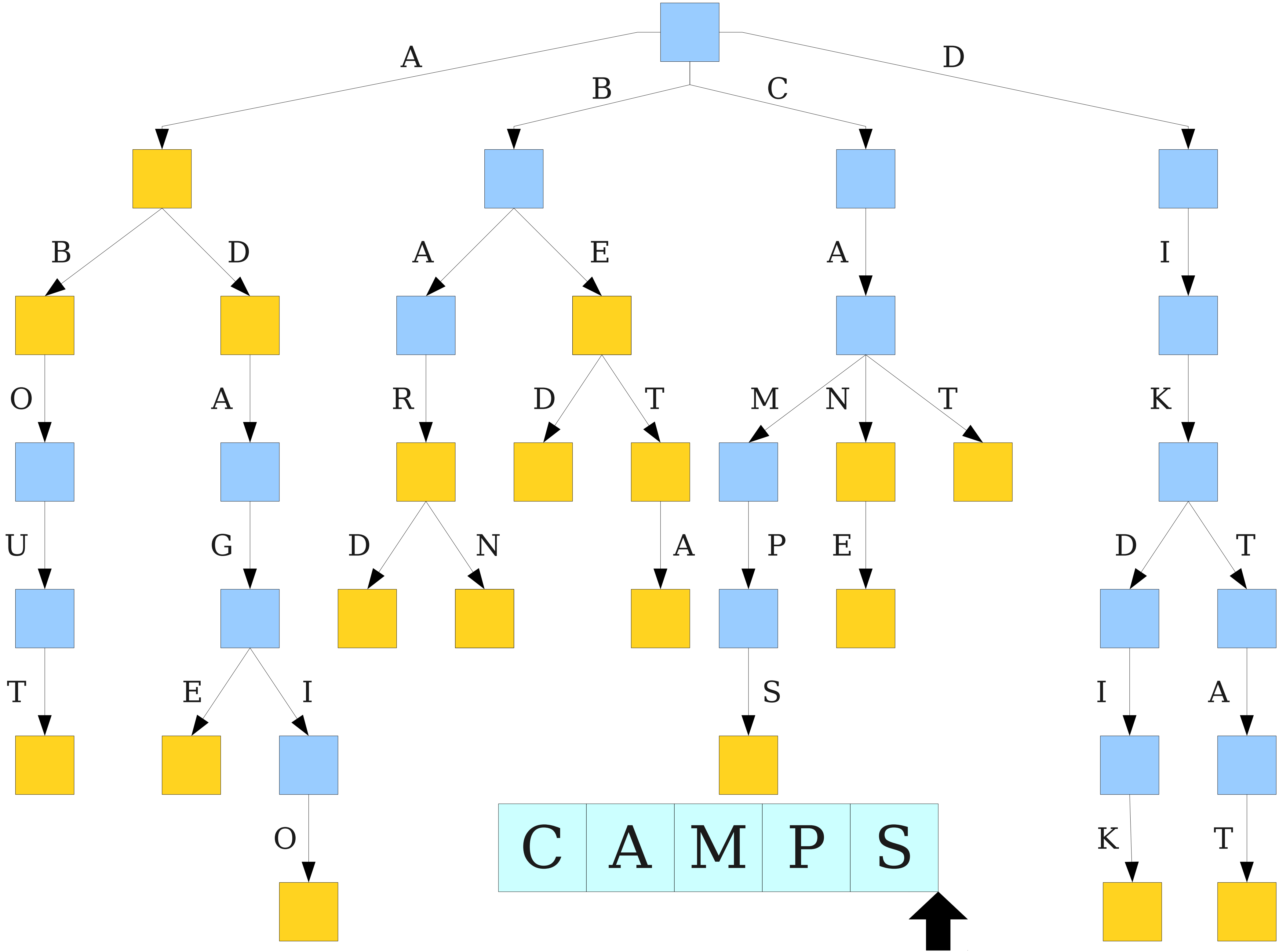

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
P
E
D
T
T
E
I
I
A
O
K
T
C A M P S

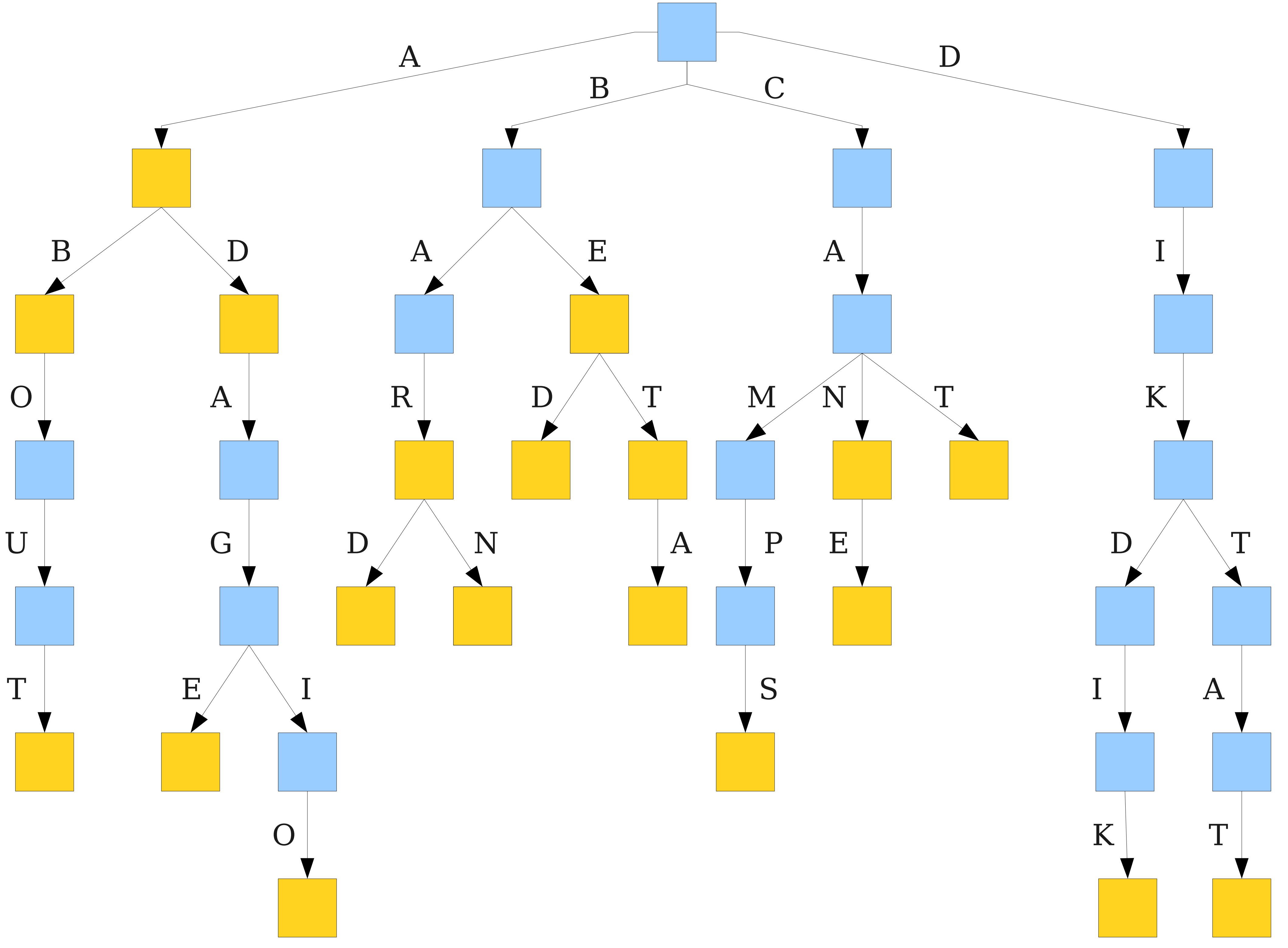

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
P
E
D
T
T
E
I
S
I
A
O
K
T
C A M P S

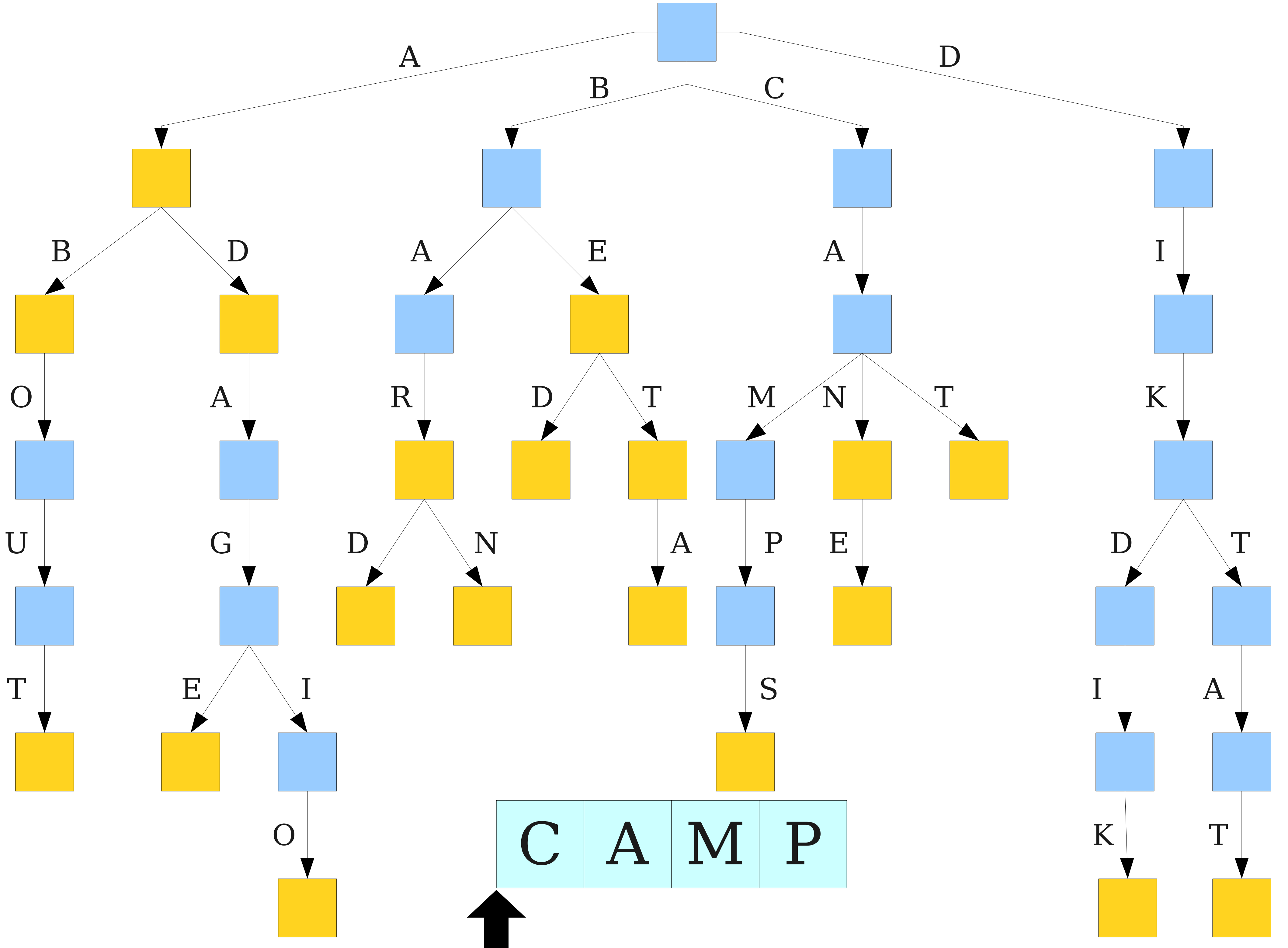


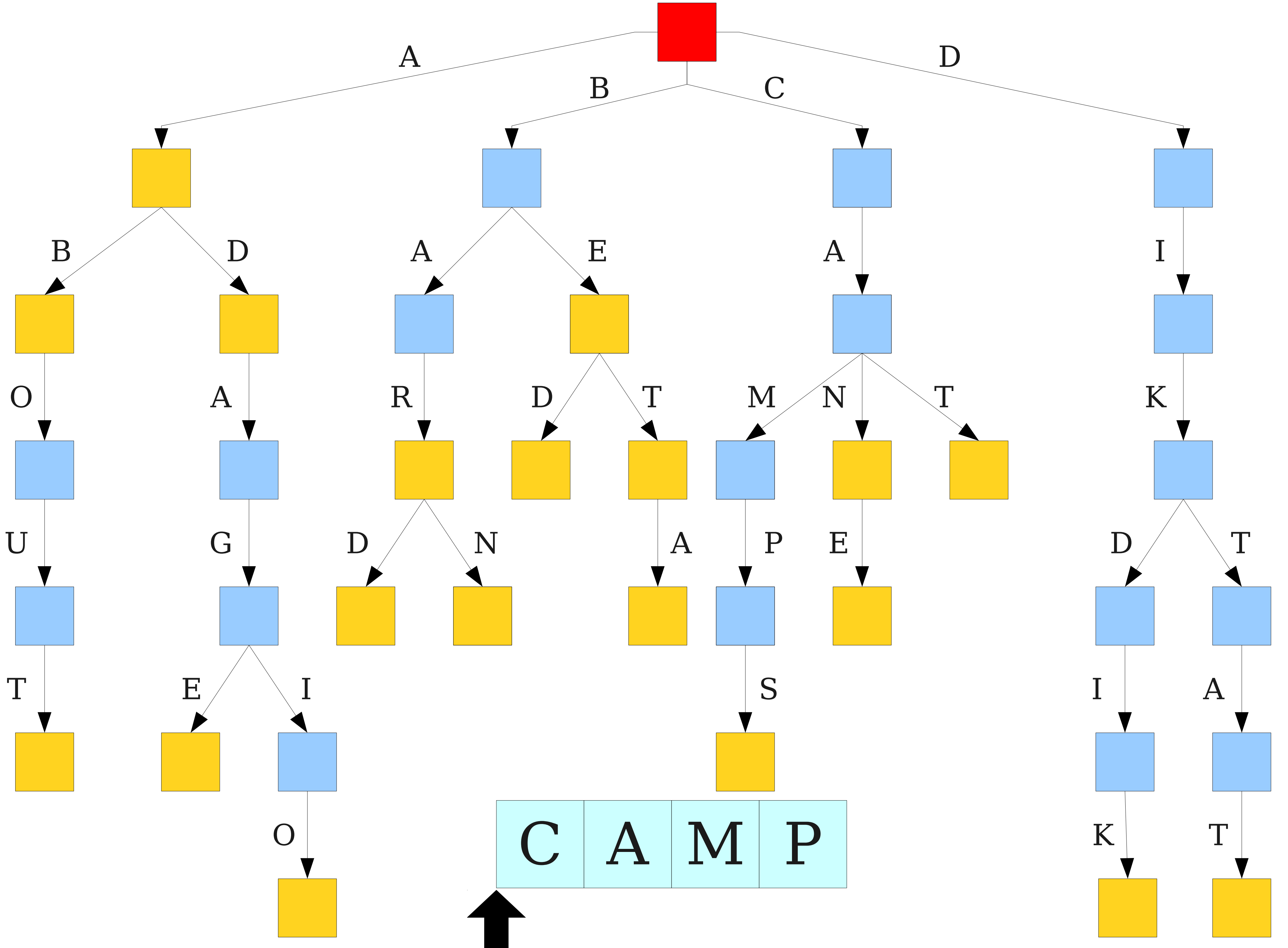
A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
P
E
D
T
T
E
I
S
I
A
O
K
T
C
A
M
P
S

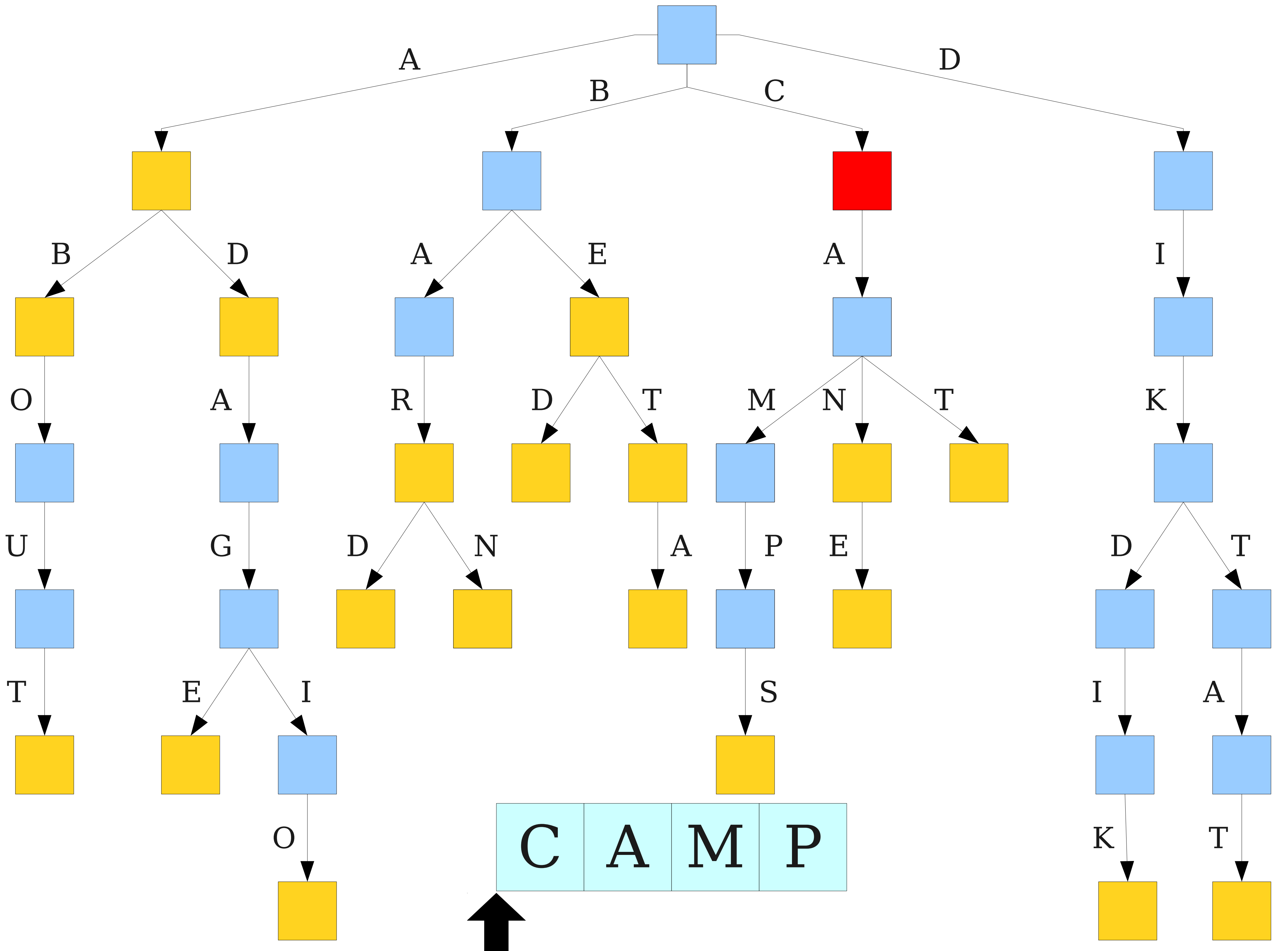

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
P
E
D
T
T
E
I
S
I
A
O
K
T
C A M P S


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 K T
C A M P S

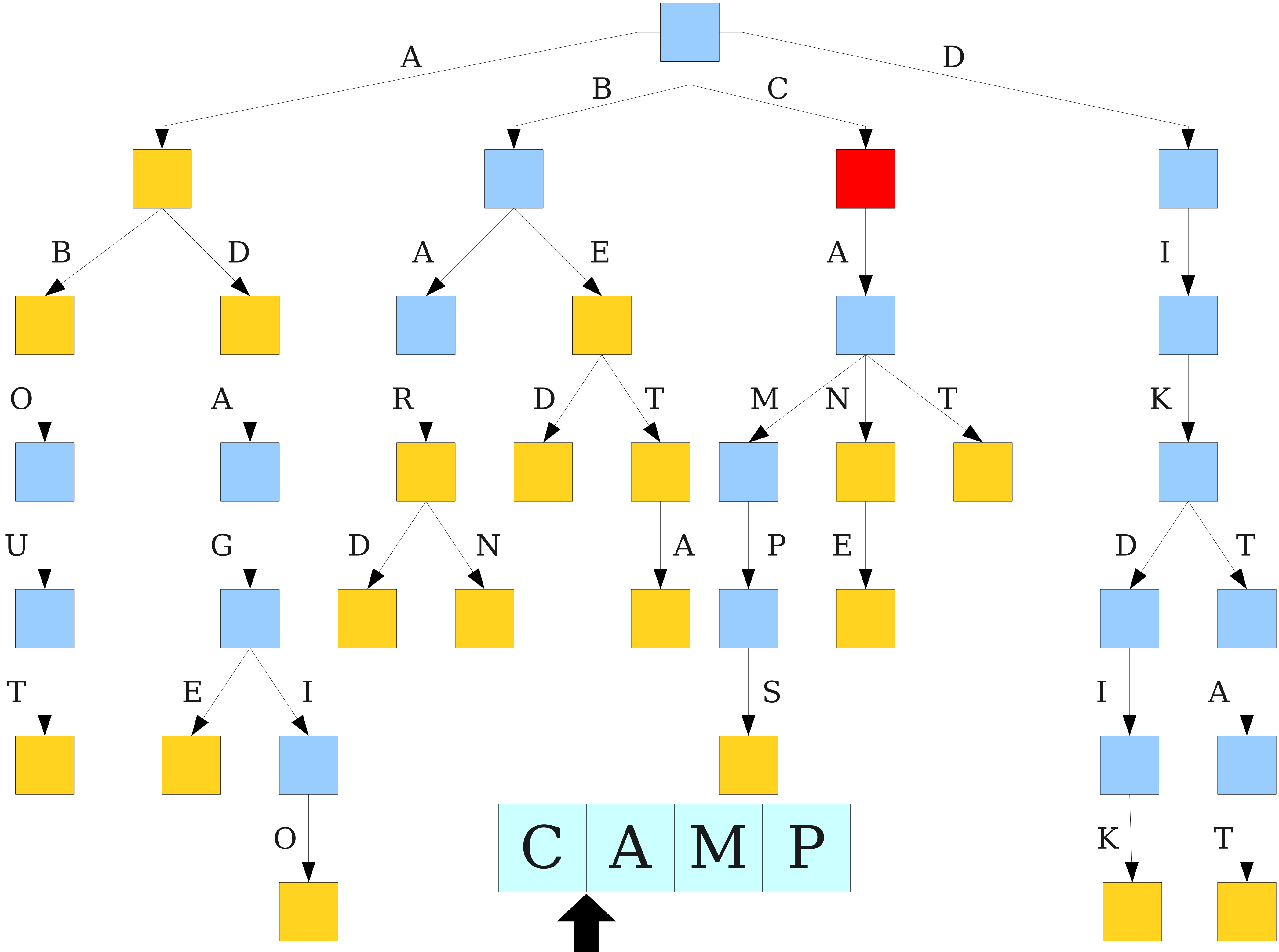

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
P
E
D
T
T
E
I
S
I
A
O
K
T

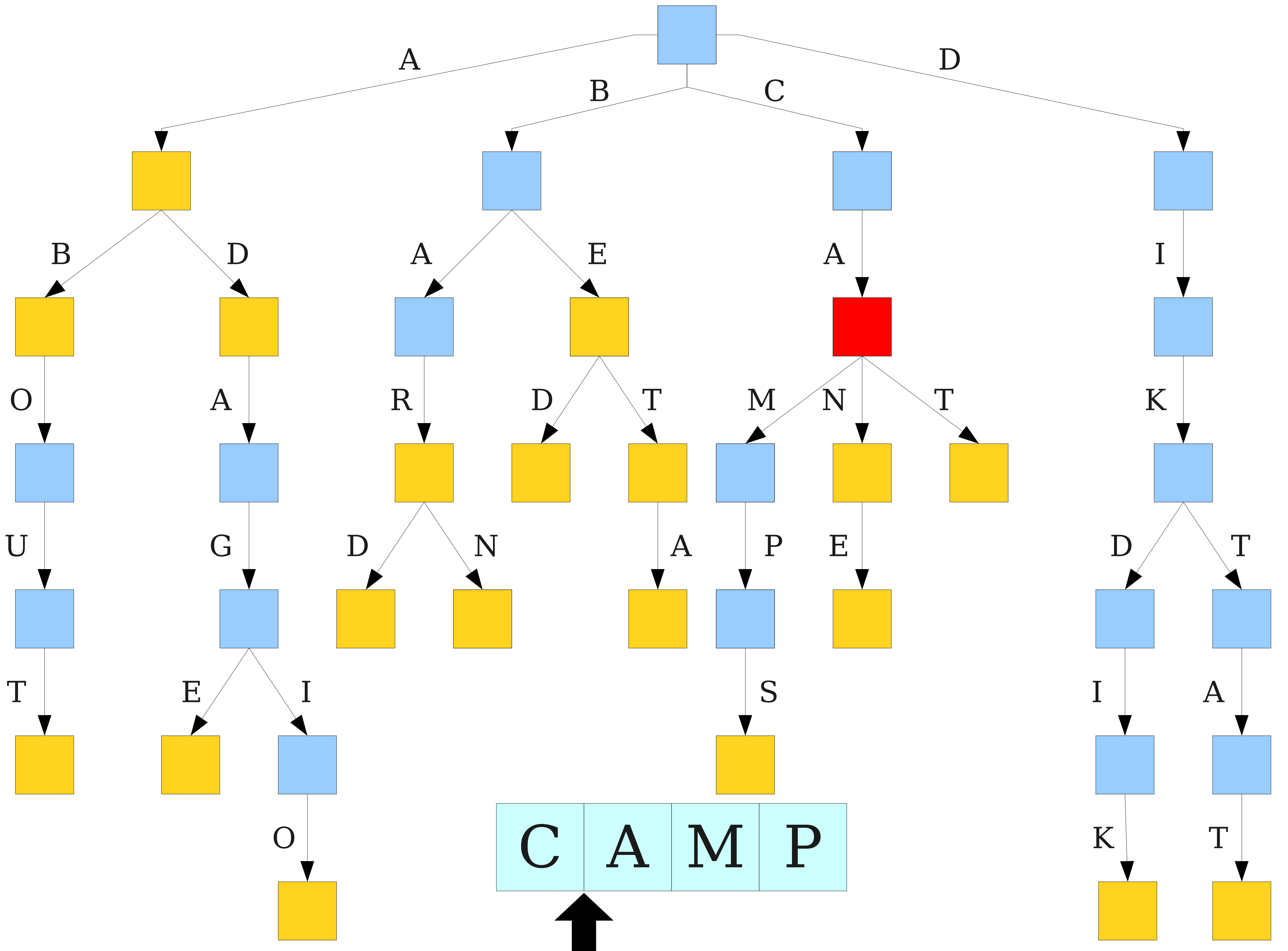

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
P
E
D
T
T
E
I
S
I
A
O
K
T
C A M P


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

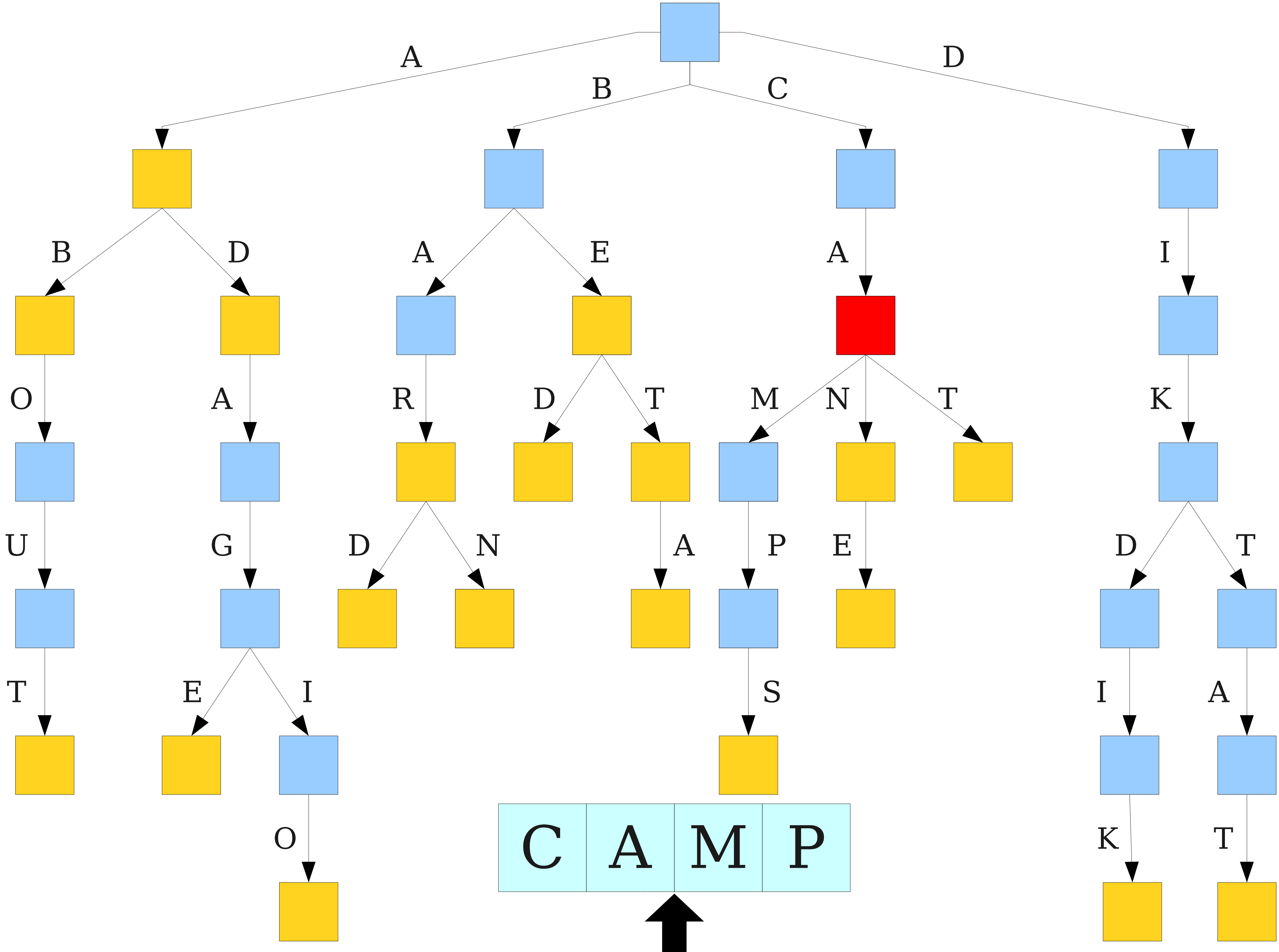


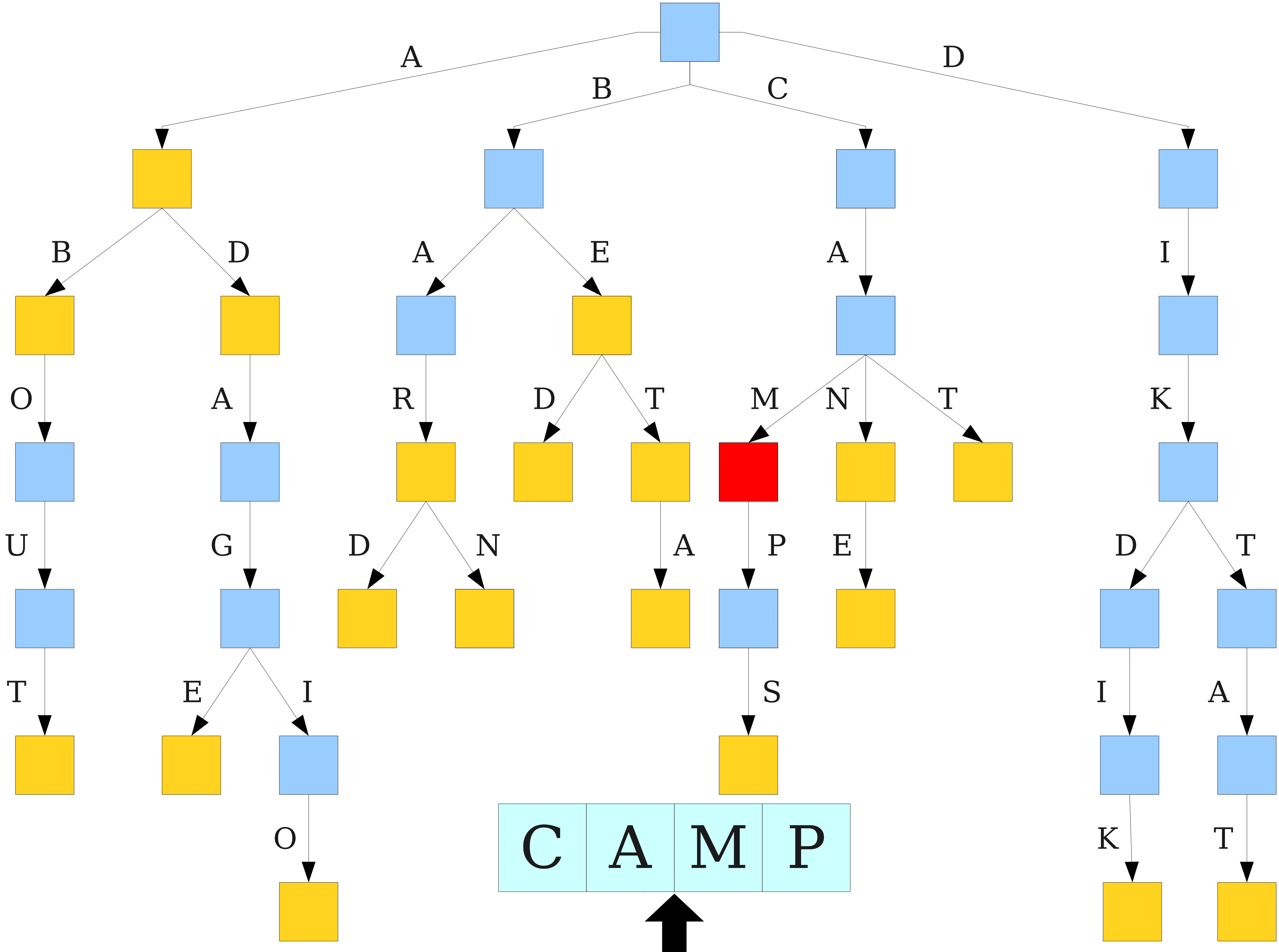
A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
P
E
D
T
T
E
I
S
I
A
O
K
T
C A M P


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

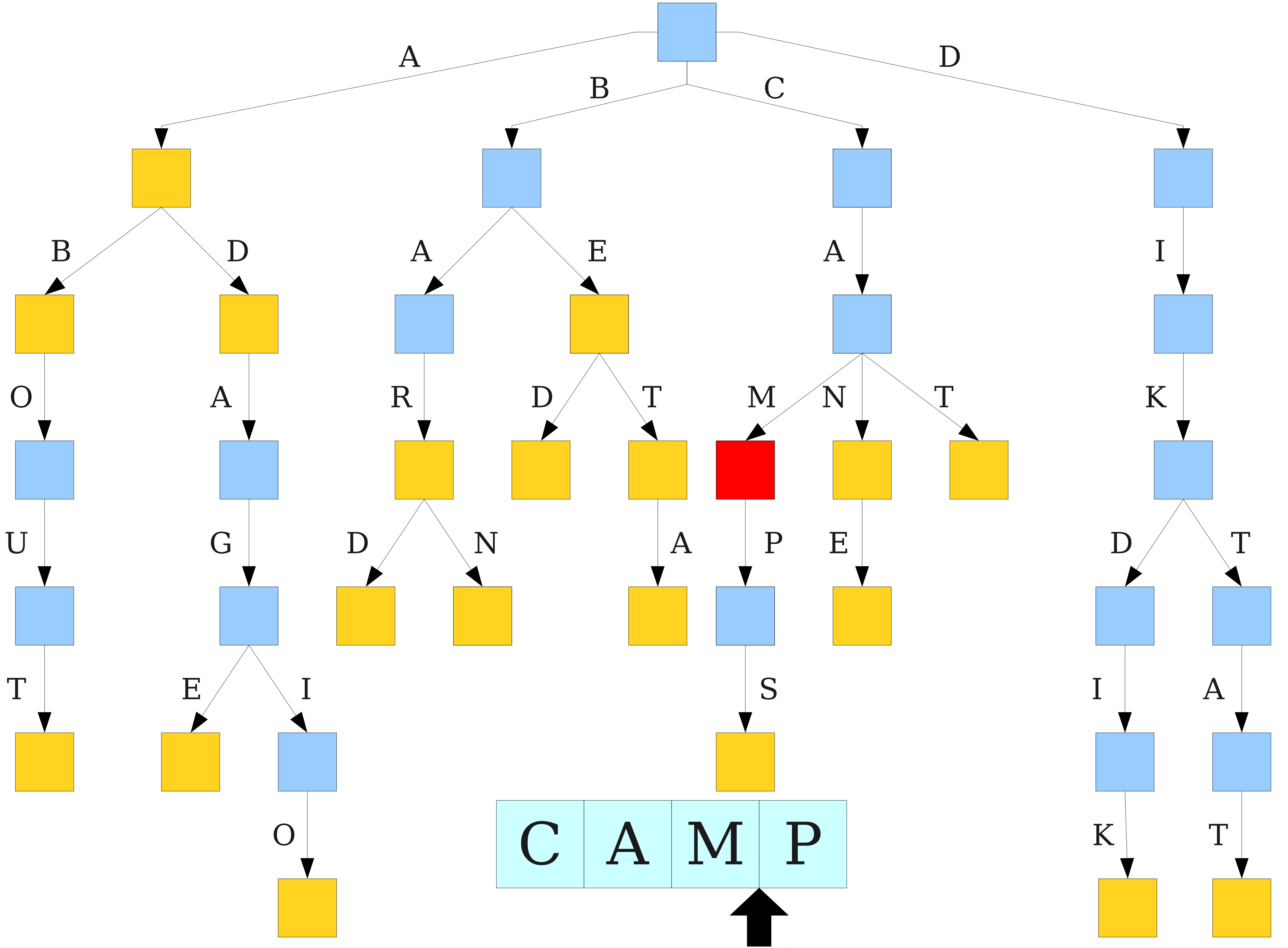


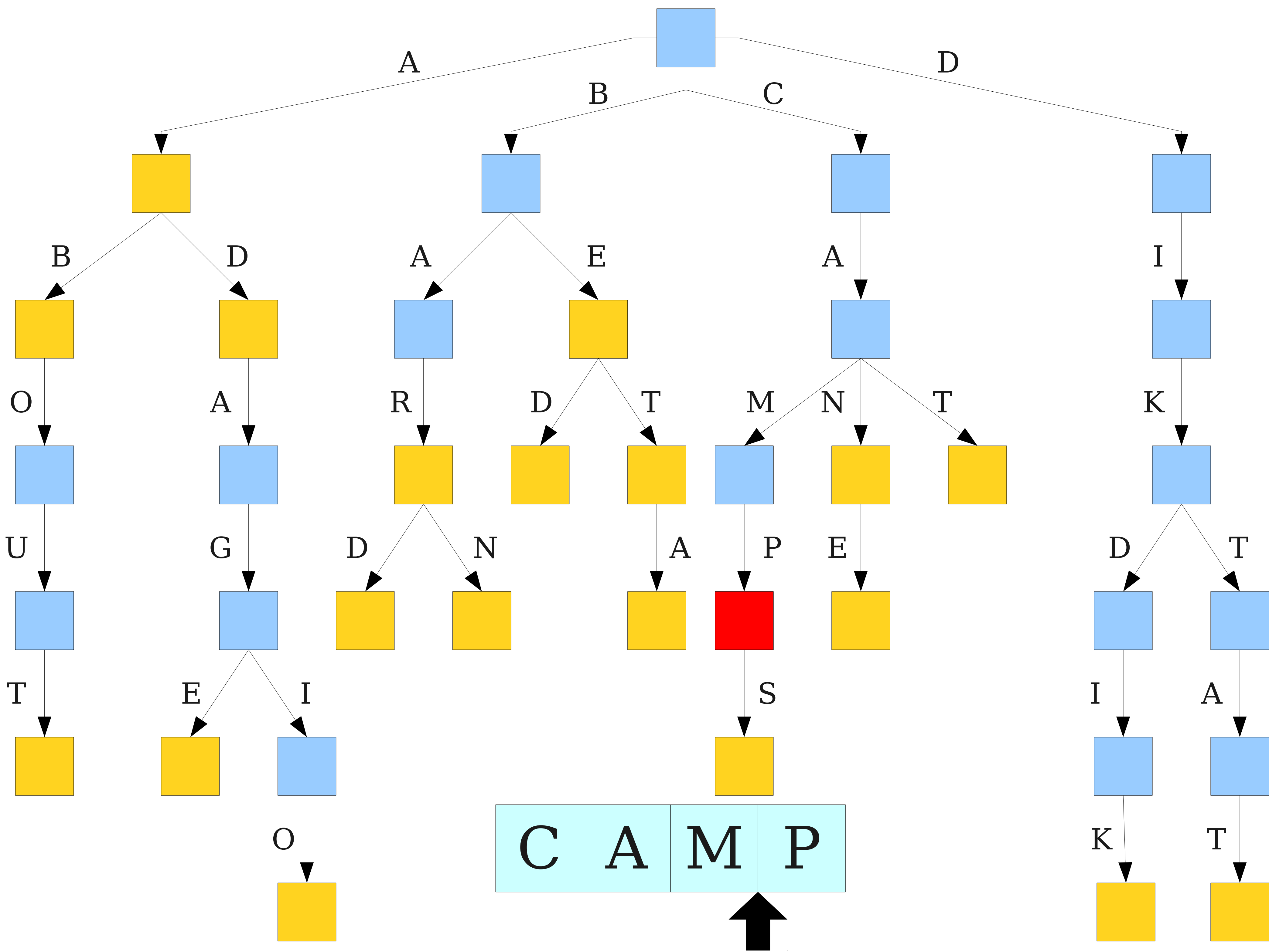
A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
P
E
D
T
T
E
I
S
I
A
O
K
T
C A M P

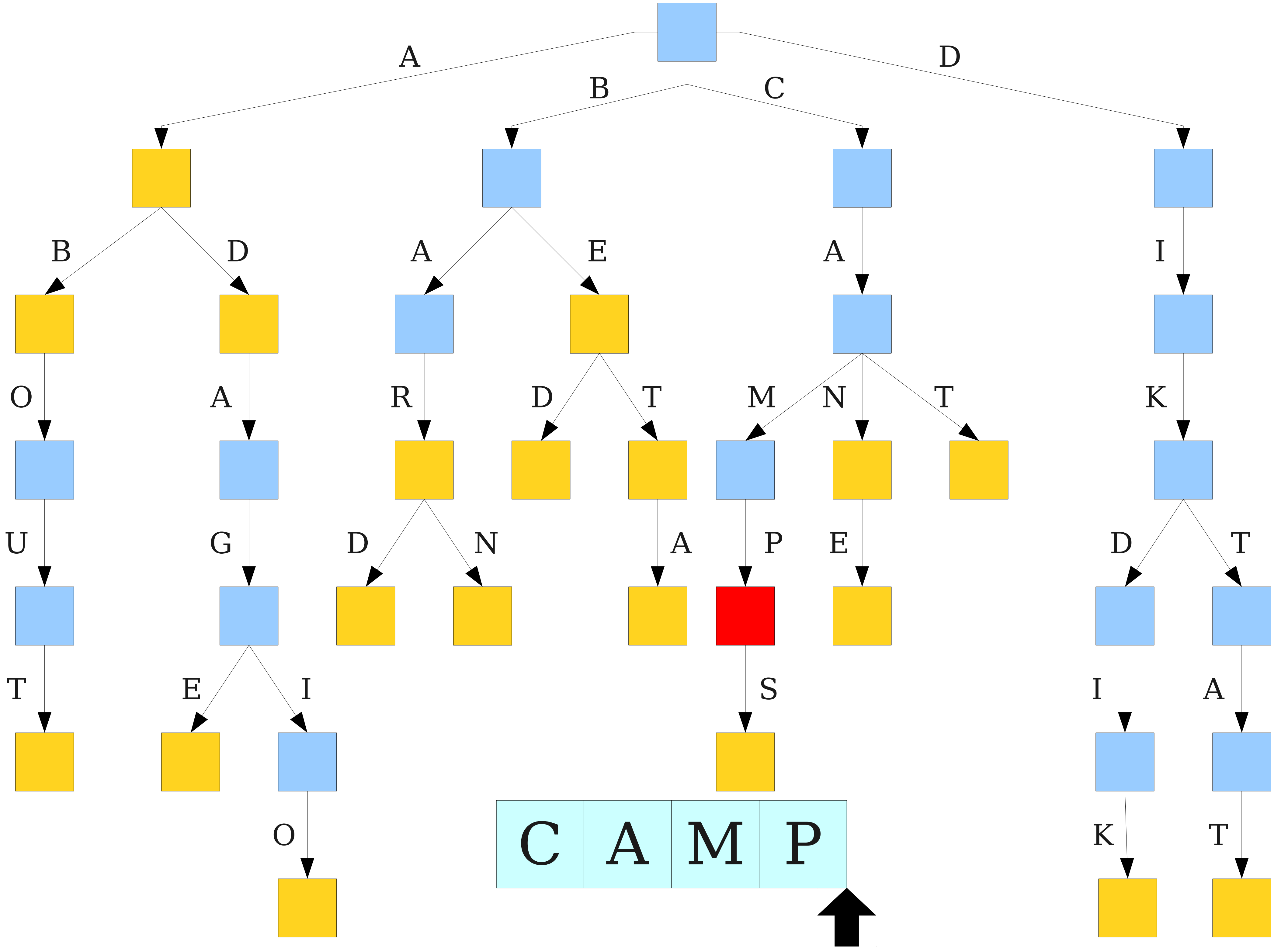

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
P
E
D
T
T
E
I
S
I
A
O
K
T
C
A
M
P

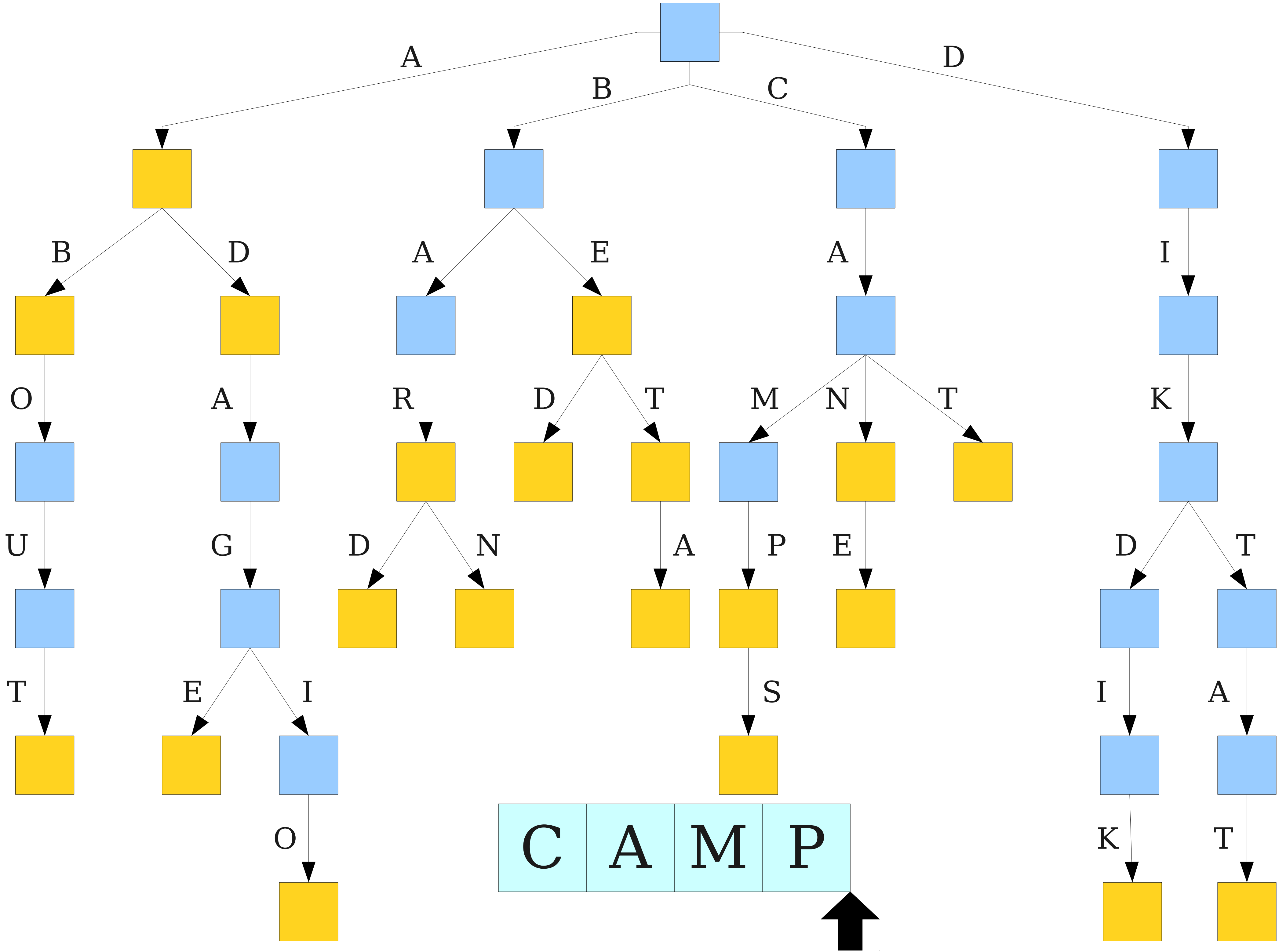


A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
P
E
D
T
T
E
I
S
I
A
O
K
T
C A M P


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
C A M P
K
T

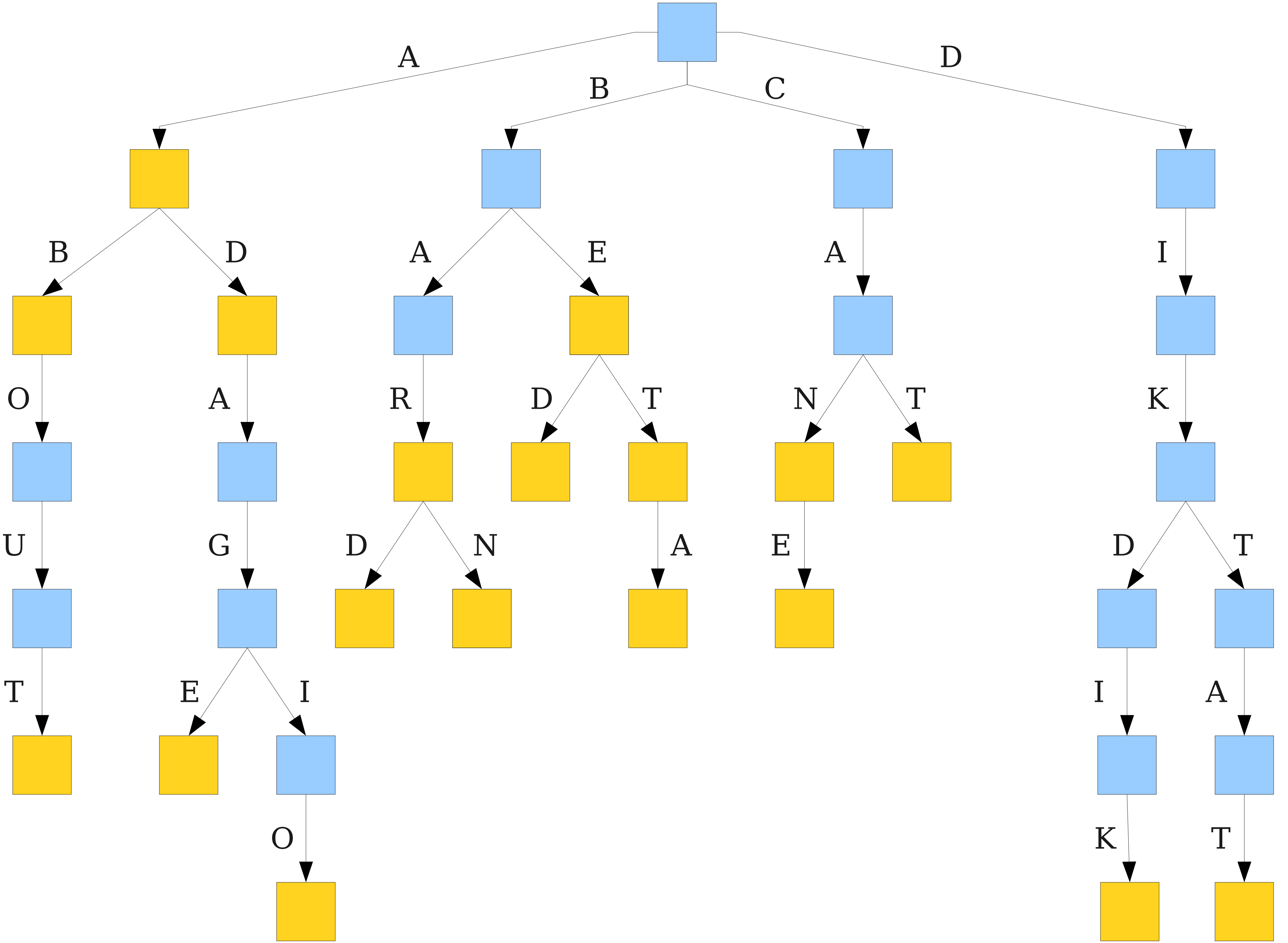

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
P
E
D
T
T
E
I
S
I
A
O
K
T
C
A
M
P

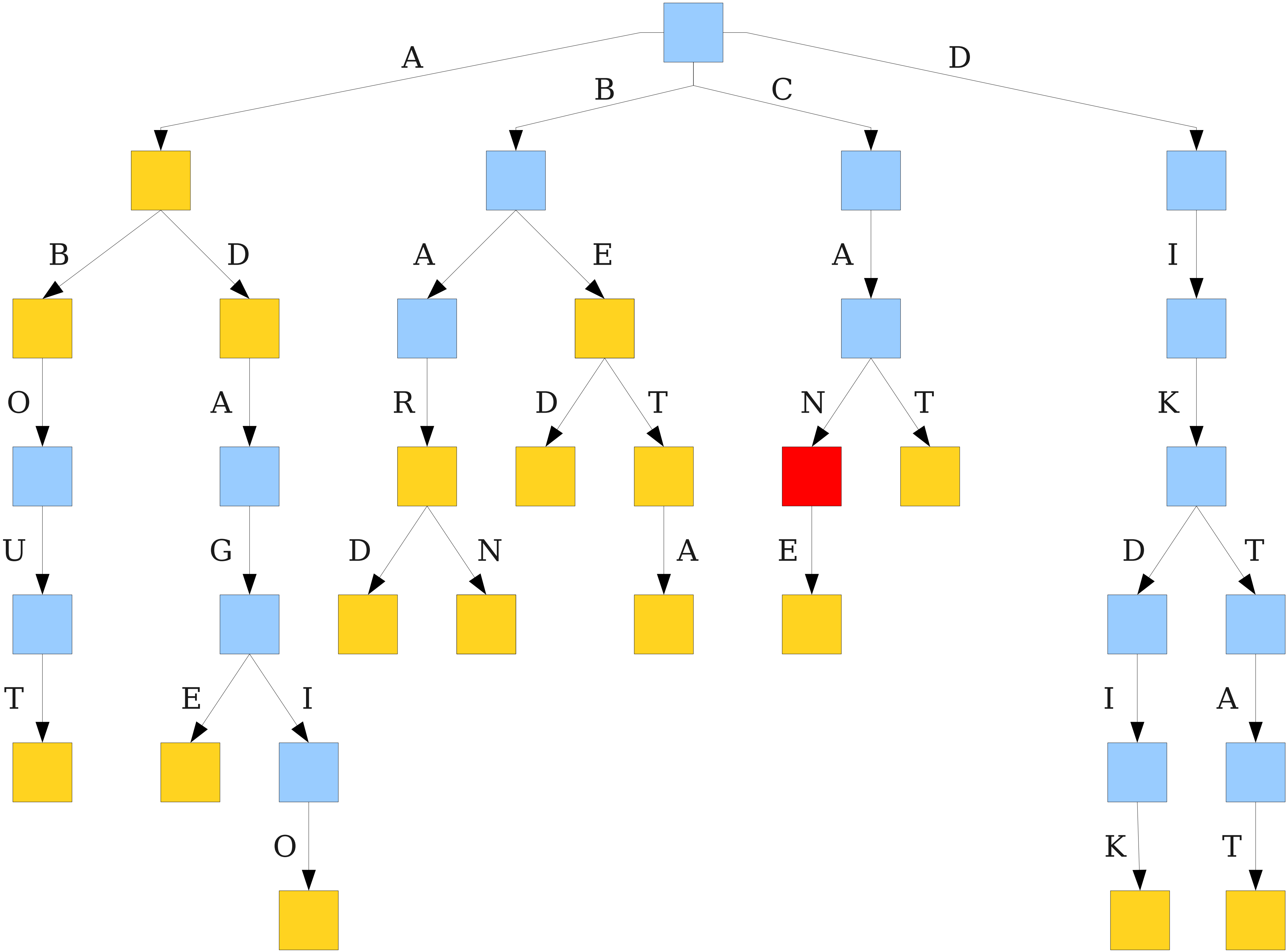

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
P
E
D
T
T
E
I
S
I
A
O
K
T
C A M P

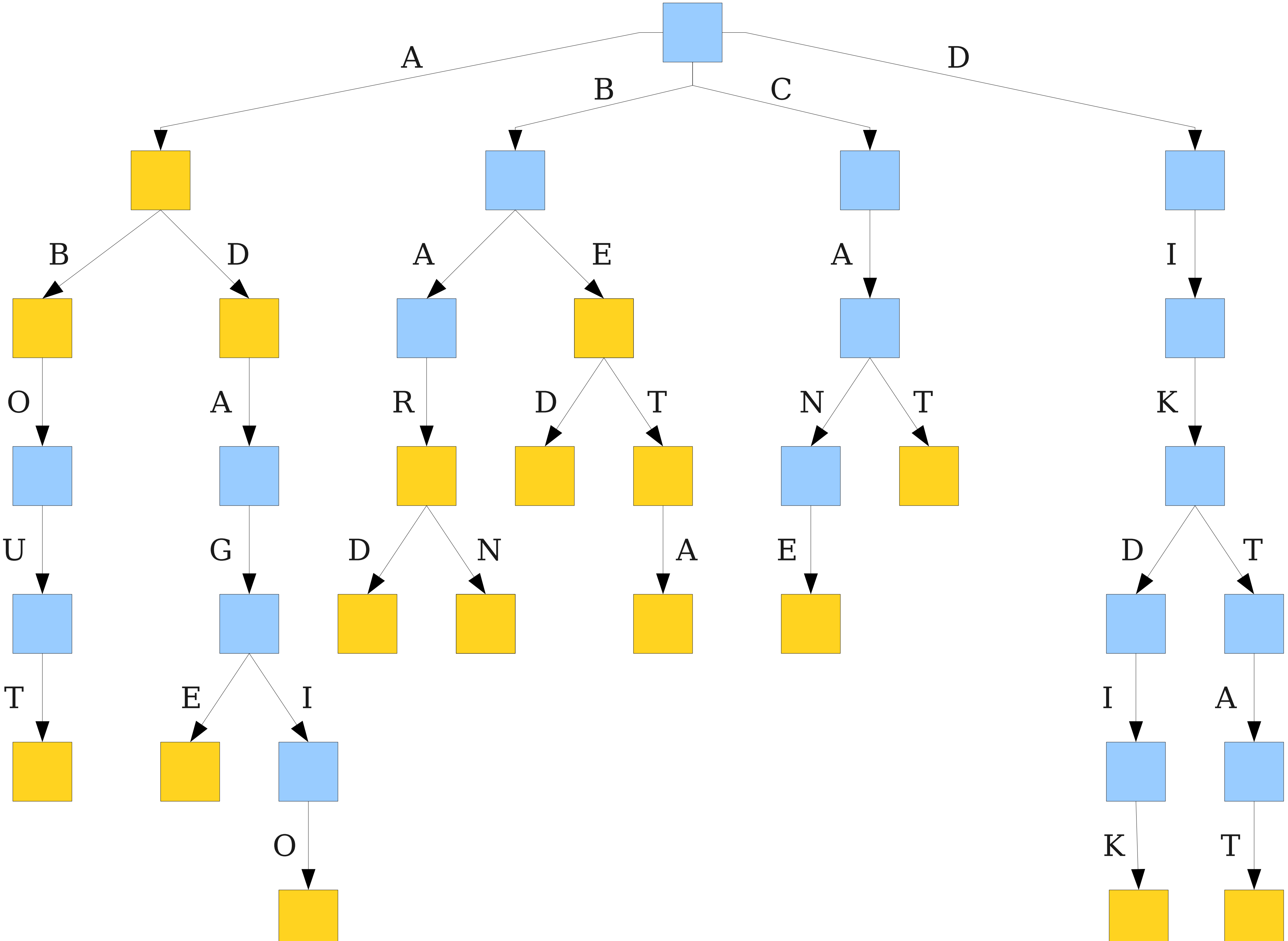

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
M
N
T
K
U
G
D
N
A
P
E
D
T
T
E
I
S
I
A
O
K
T
CAMP

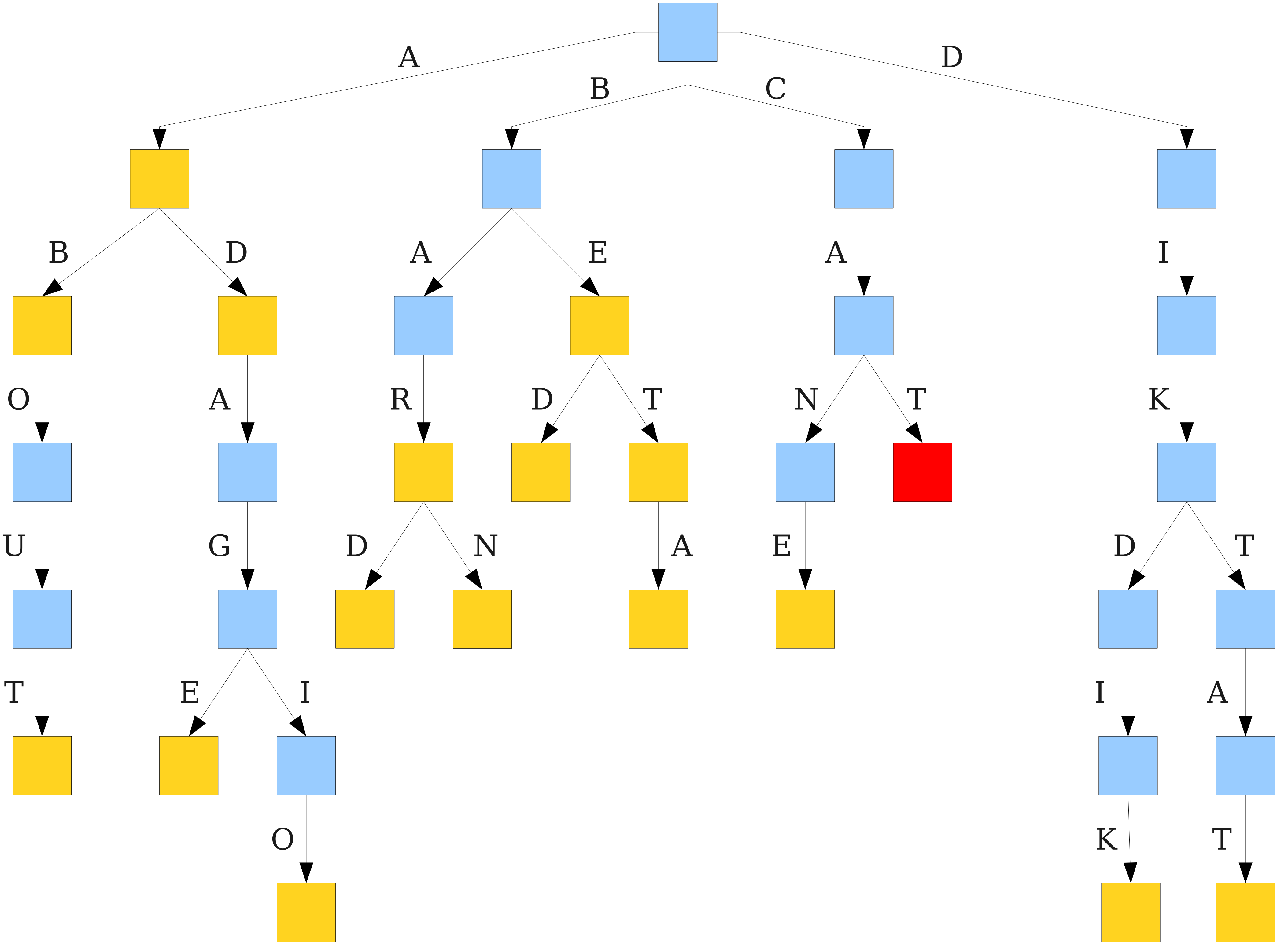
Removing from a Trie

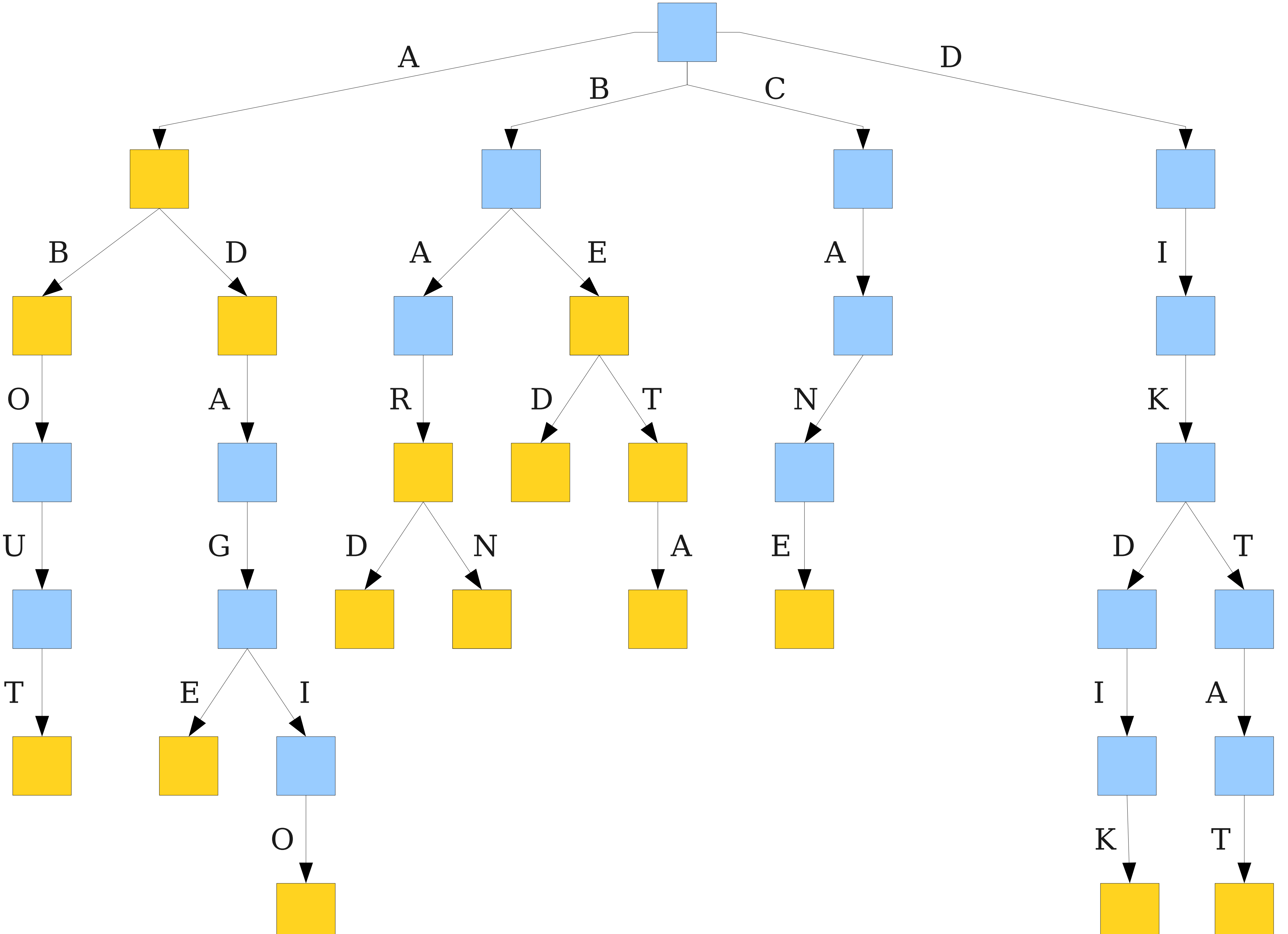
- Mark the node as no longer containing a word.
- If the node has no children:
  - Remove that node.
  - Repeat this process at the node one level higher up in the tree.


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

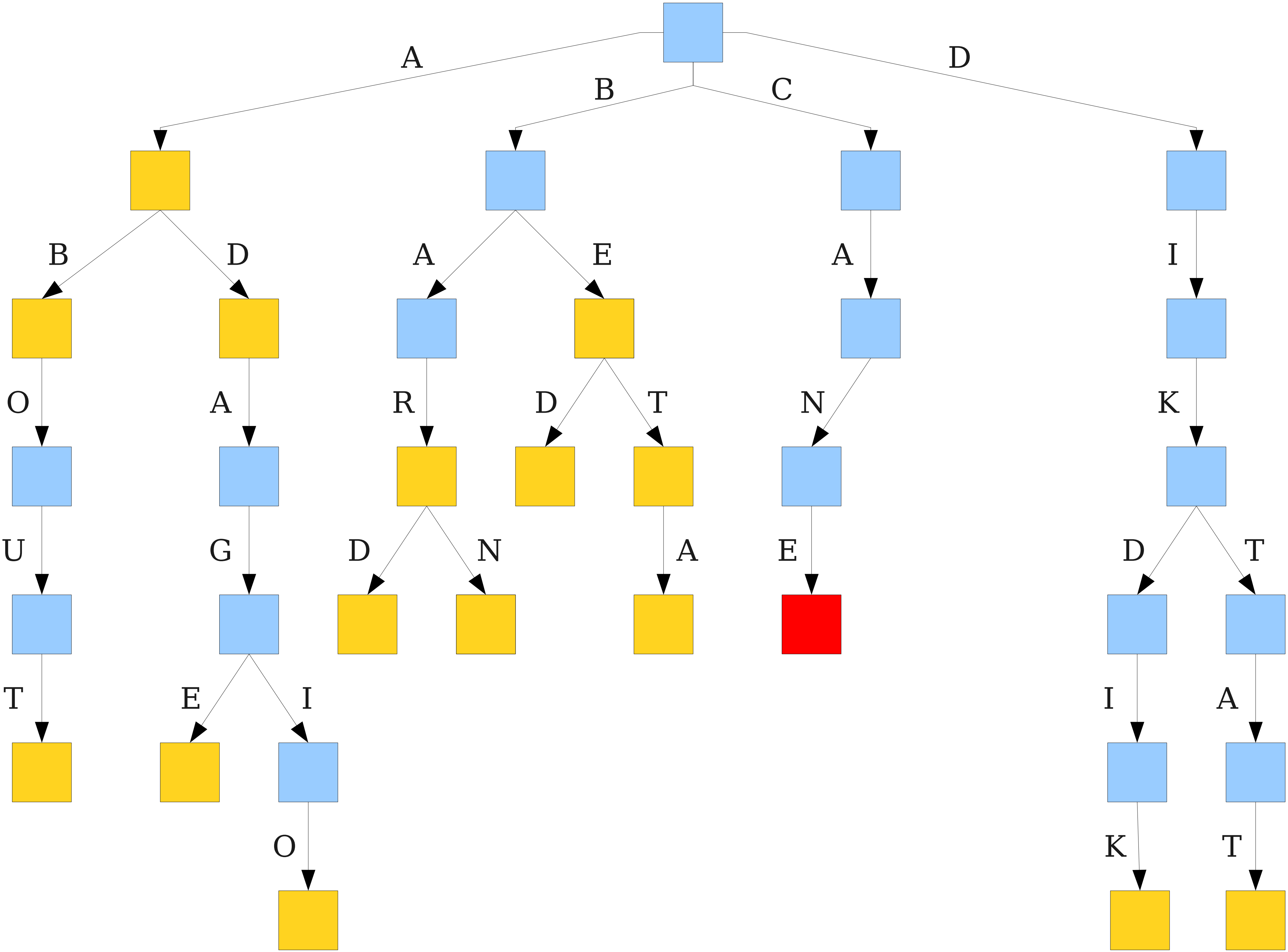

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T

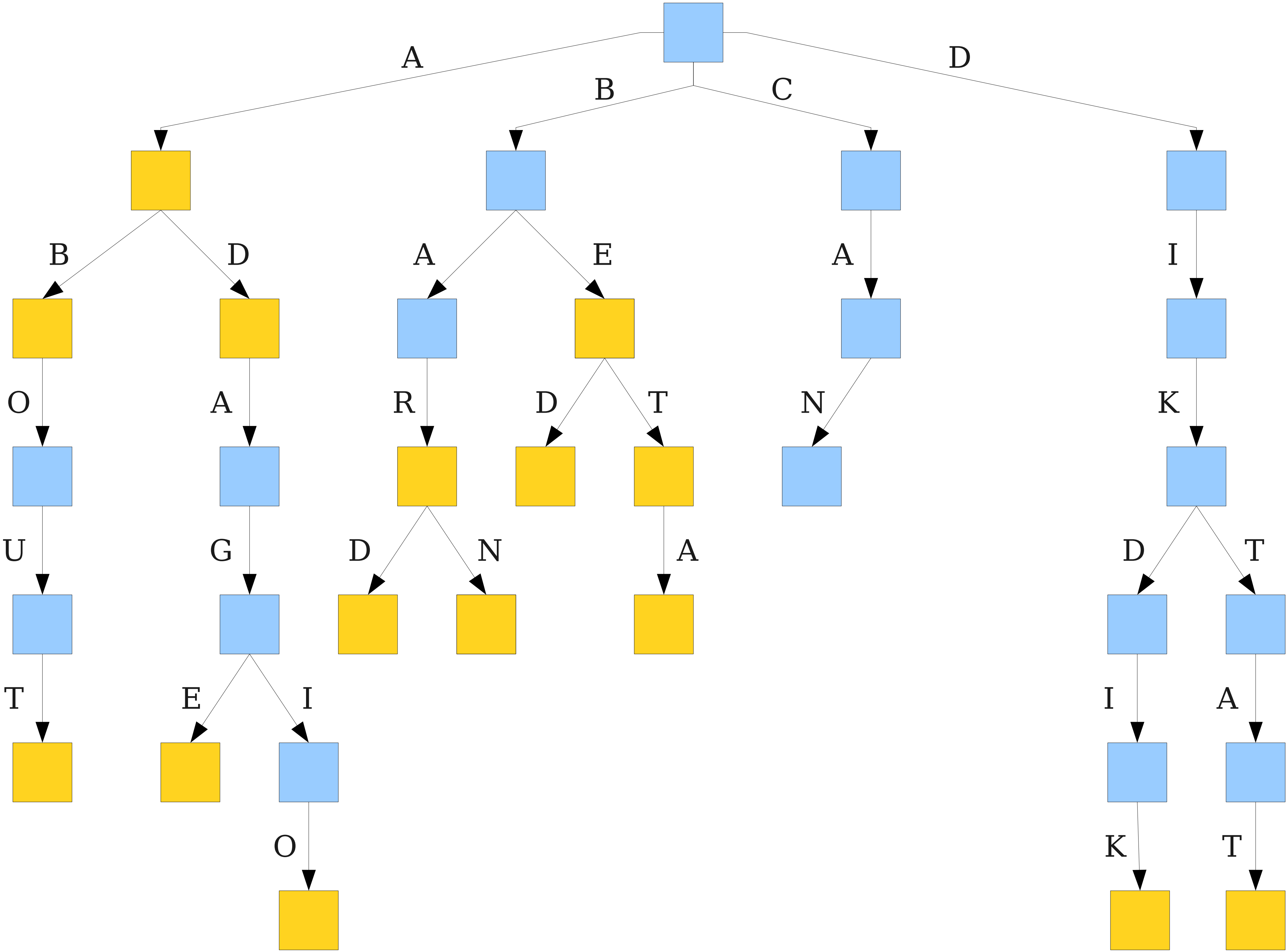

A
D
B
C
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T

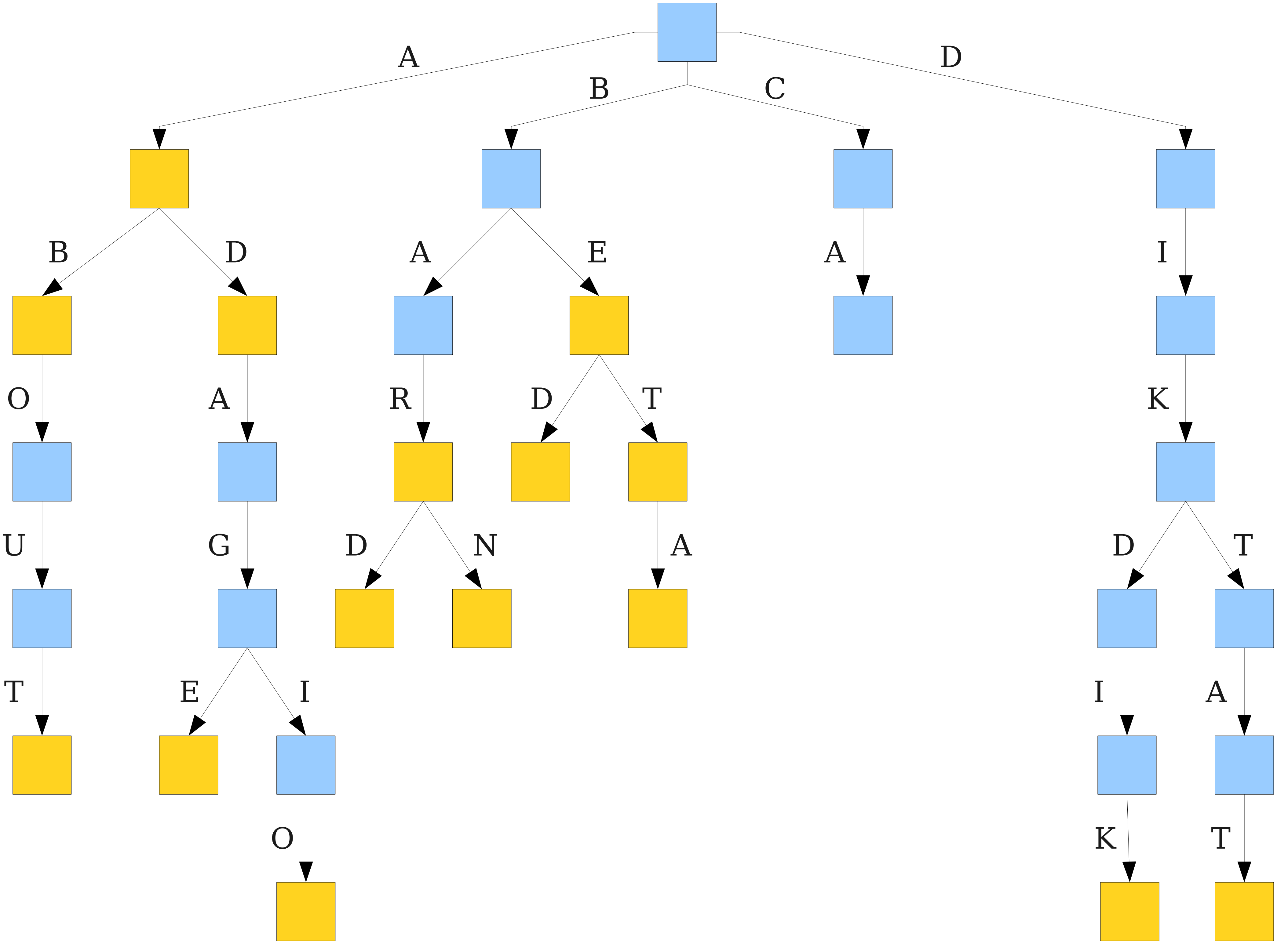

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T

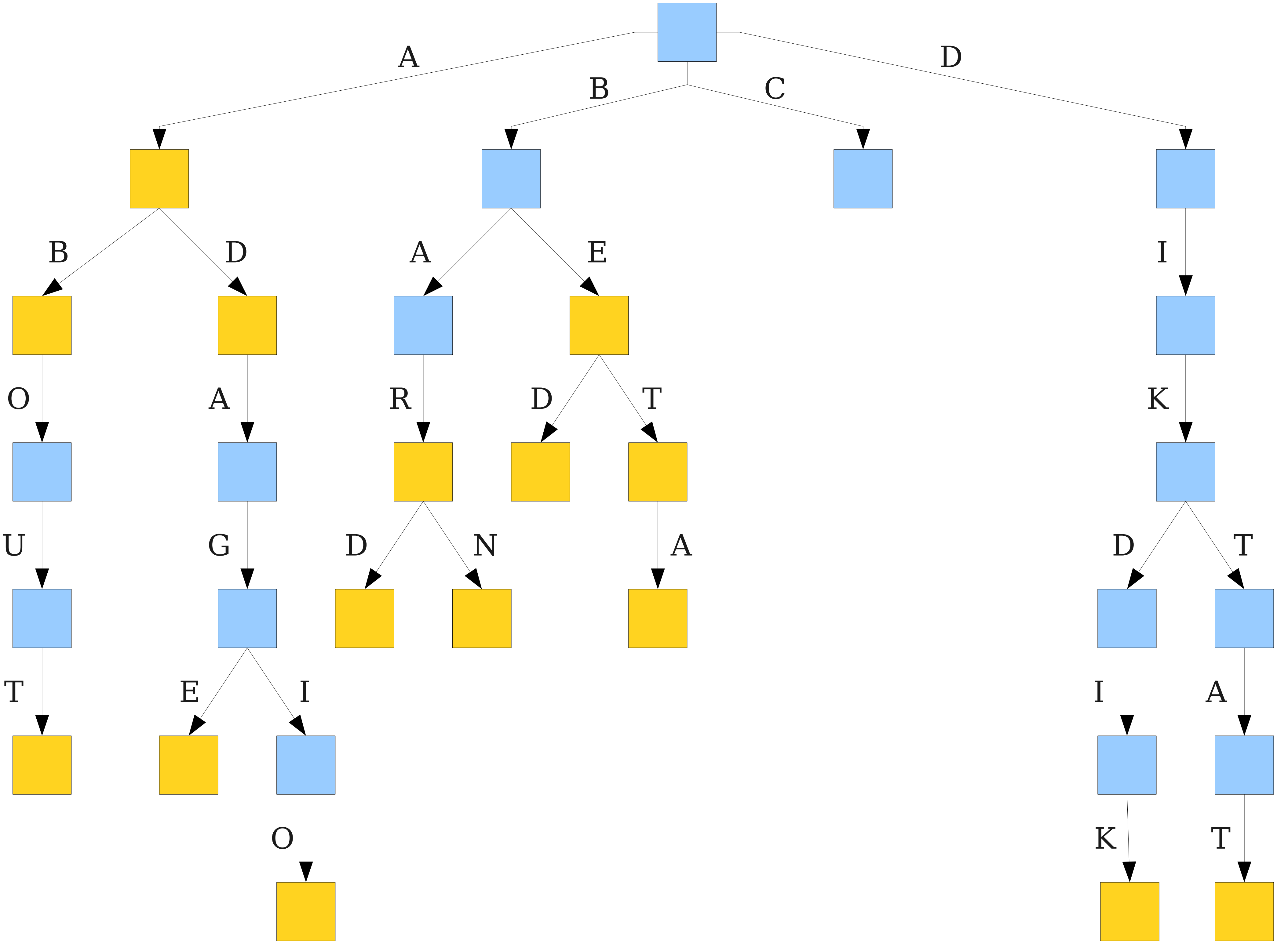


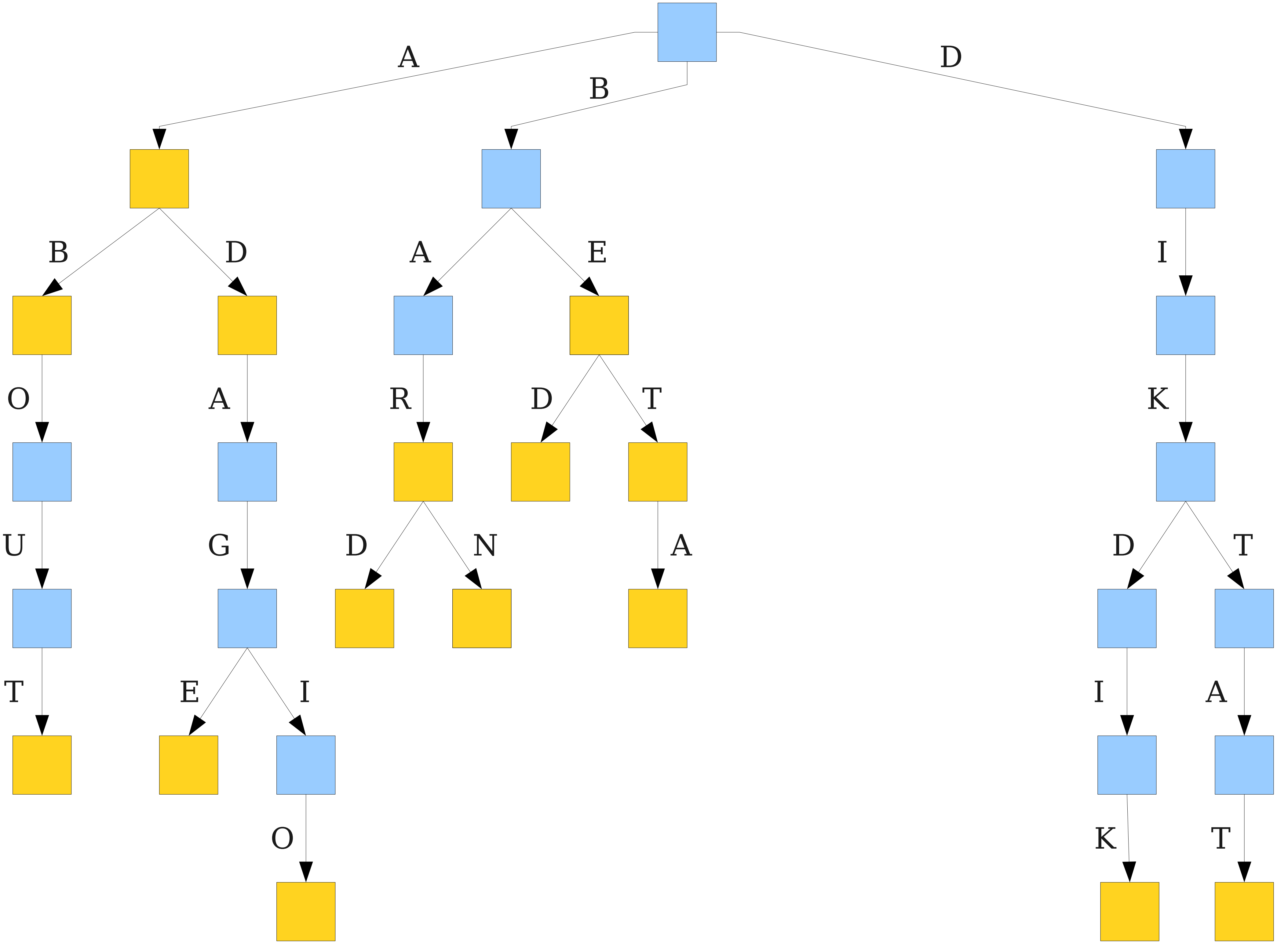
A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T


A
D
B
C
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
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
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

Space Concerns

- Although time-efficient, tries can be extremely space-inefficient.
- A trie with N nodes will need space $\Theta(N \cdot |\Sigma|)$ due to the pointers in each node.
- There are many ways of addressing this:
  - Change the data structure for holding the pointers
  - Eliminate unnecessary trie nodes (we'll see this next time).

String Matching

String Matching

- The string matching problem is the following:

  Given a text string T and a nonempty string P, find all occurrences of P in T.

- T is typically called the *text* and P is the *pattern*.
- We're looking for an exact match; P doesn't contain any wildcards, for example.
- How efficiently can we solve this problem?

The Naïve Solution

- Consider the following naïve solution: for every possible starting position for P in T, check whether the $|P|$ characters starting at that point exactly match P.
- Work per check: $O(|P|)$
- Number of starting locations: $O(|T|)$
- Total runtime: $O(|P| \cdot |T|)$.
- Is this a tight bound?

Other Solutions

- **Rabin-Karp**: Using hash functions, reduces runtime to expected $O(|P| + |T|)$, with worst-case $O(|P| \cdot |T|)$ and space $O(1)$.
- **Knuth-Morris-Pratt**: Using some clever preprocessing, reduces runtime to worst-case $O(|P| + |T|)$ and space $O(|P|)$.
- Check out CLRS, Chapter 32 for details.
  - … or don't, because KMP is a special case of the algorithm we're going to see later today.

Multi-String Searching

- Now, consider the following problem:

  Given a string T and a set of k nonempty strings $P_1, \ldots, P_k$, find all occurrences of $P_1, \ldots, P_k$ in T.

- Many applications:
  - Constructing indices: Find all occurrences of specified terms in a document.
  - Antivirus databases: Find all occurrences of specific virus fingerprints in a program.
  - Web retrieval: Find all occurrences of a set of keywords on a page.

Some Terminology

- Let $m = |T|$, the length of the string to be searched.
- Let $n = |P_1| + |P_2| + \dots + |P_k|$ be the total length of all the strings to be searched.
- Assume that strings are drawn from an alphabet Σ, where $|\Sigma| = O(1)$.

Multi-String Searching

- **Idea:** Use one of the fast string searching algorithms to search T for each of the patterns.
- Runtime for doing a single string search: $O(m + |P_i|)$
- Runtime for doing k searches: $O(km + |P_1| + \ldots + |P_k|) = O(km + n)$.
- For large k, this can be very slow.

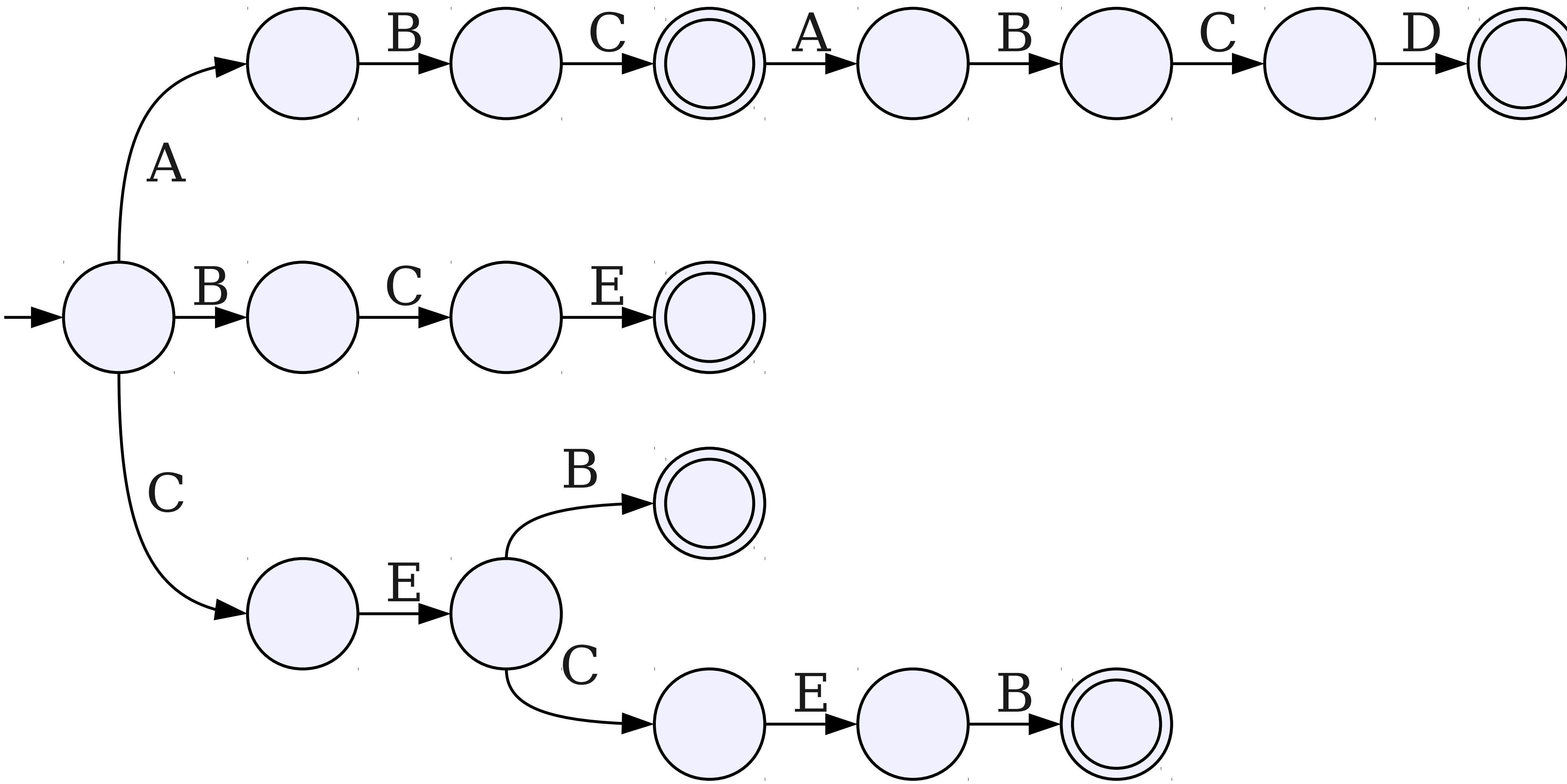
Why the Slowdown?

- Why is using an efficient string search algorithm for each pattern string slow?
- **Answer:** Each scan over the text string only searches for a single string at once.
- **Better idea:** Search for all of the strings together in parallel.

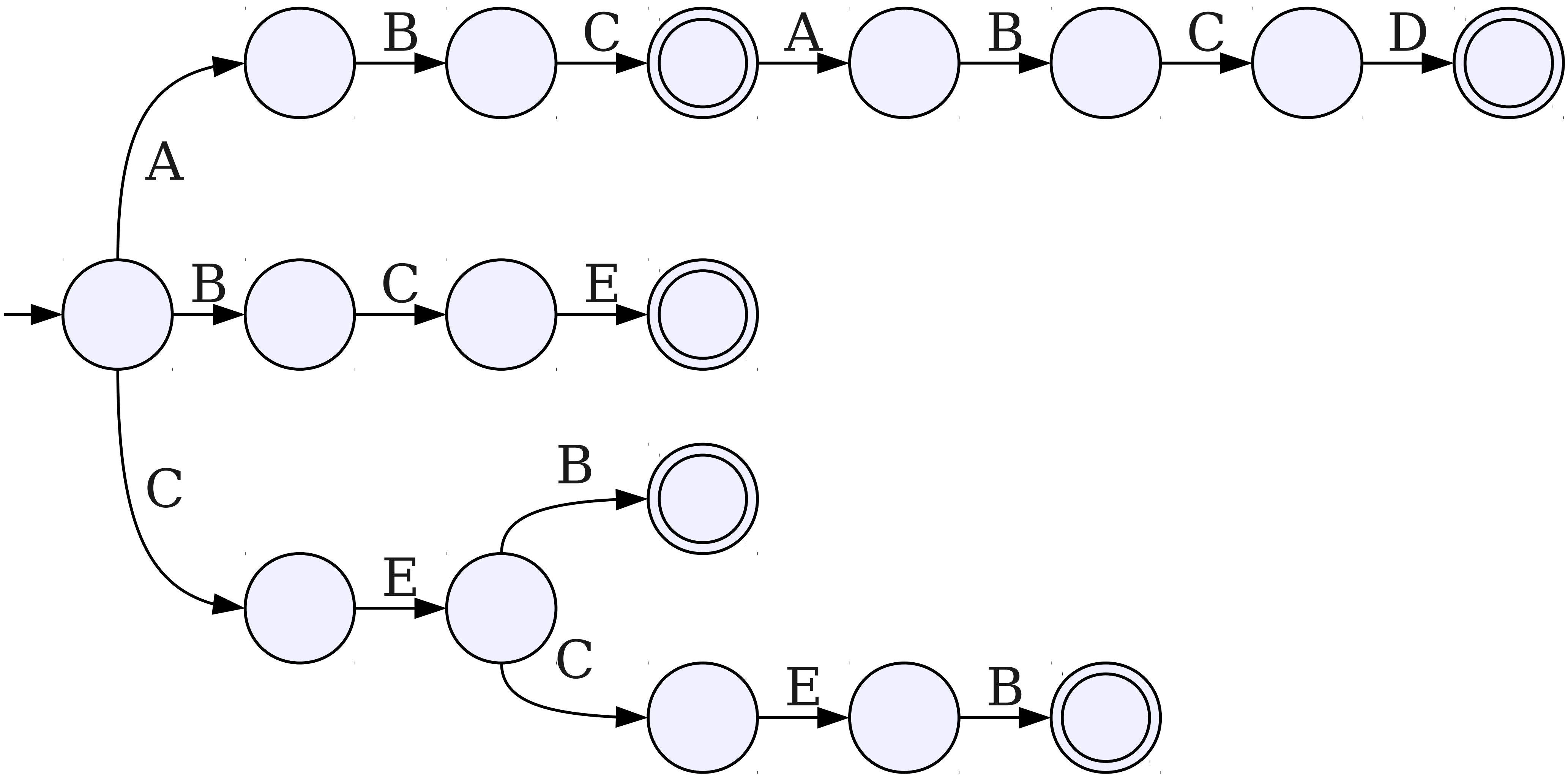
The Algorithm

- Construct a trie containing all the patterns to search for.
  - Time: O(n).
- For each character in T, search the trie starting with that character. Every time a word is found, output that word.
  - Time: O($|P_{max}|$), where P_{max} is the longest pattern string.
- Time complexity: O($m|P_{max}| + n$), which is O(mn) in the worst-case.

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC

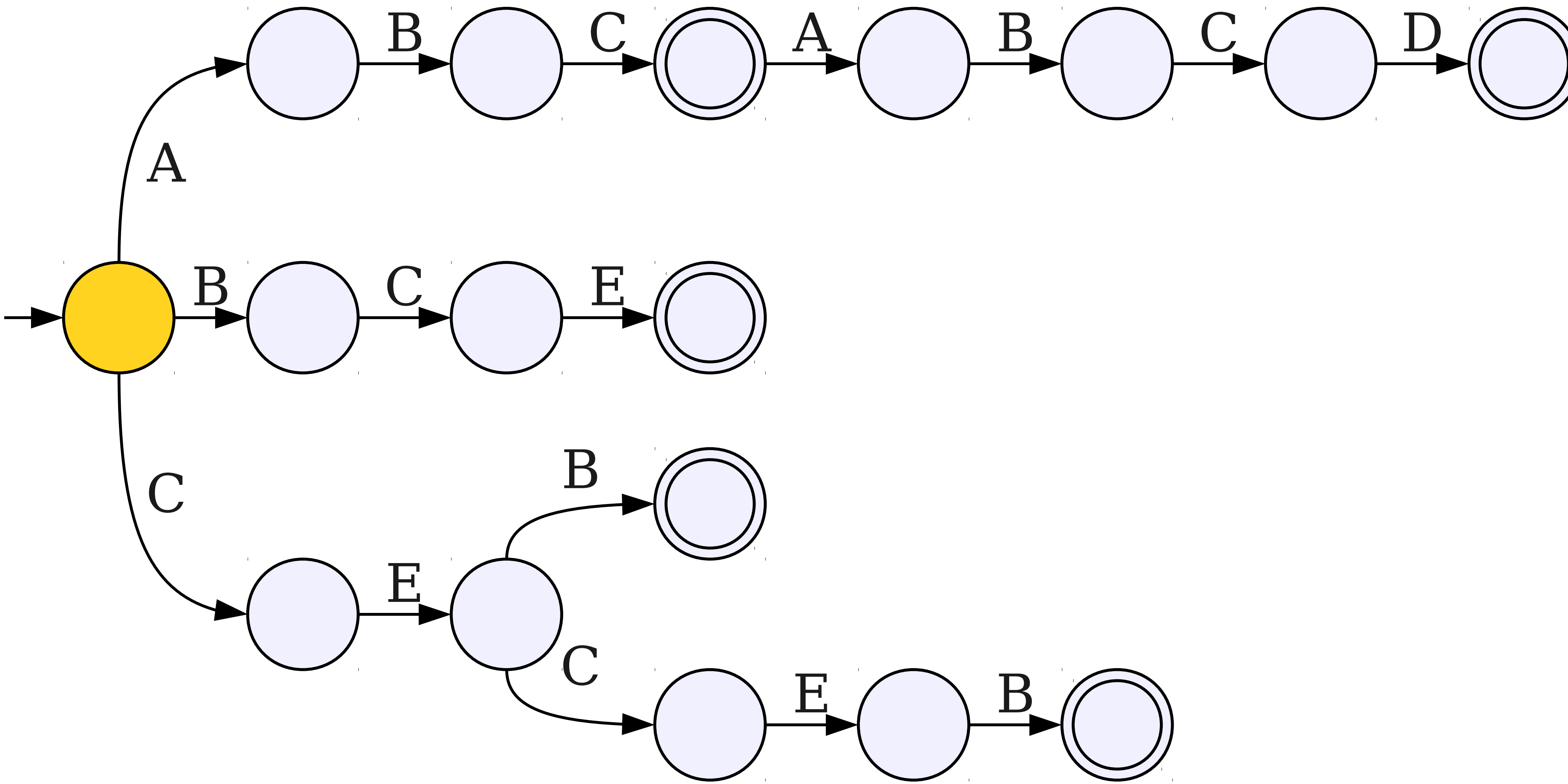


P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



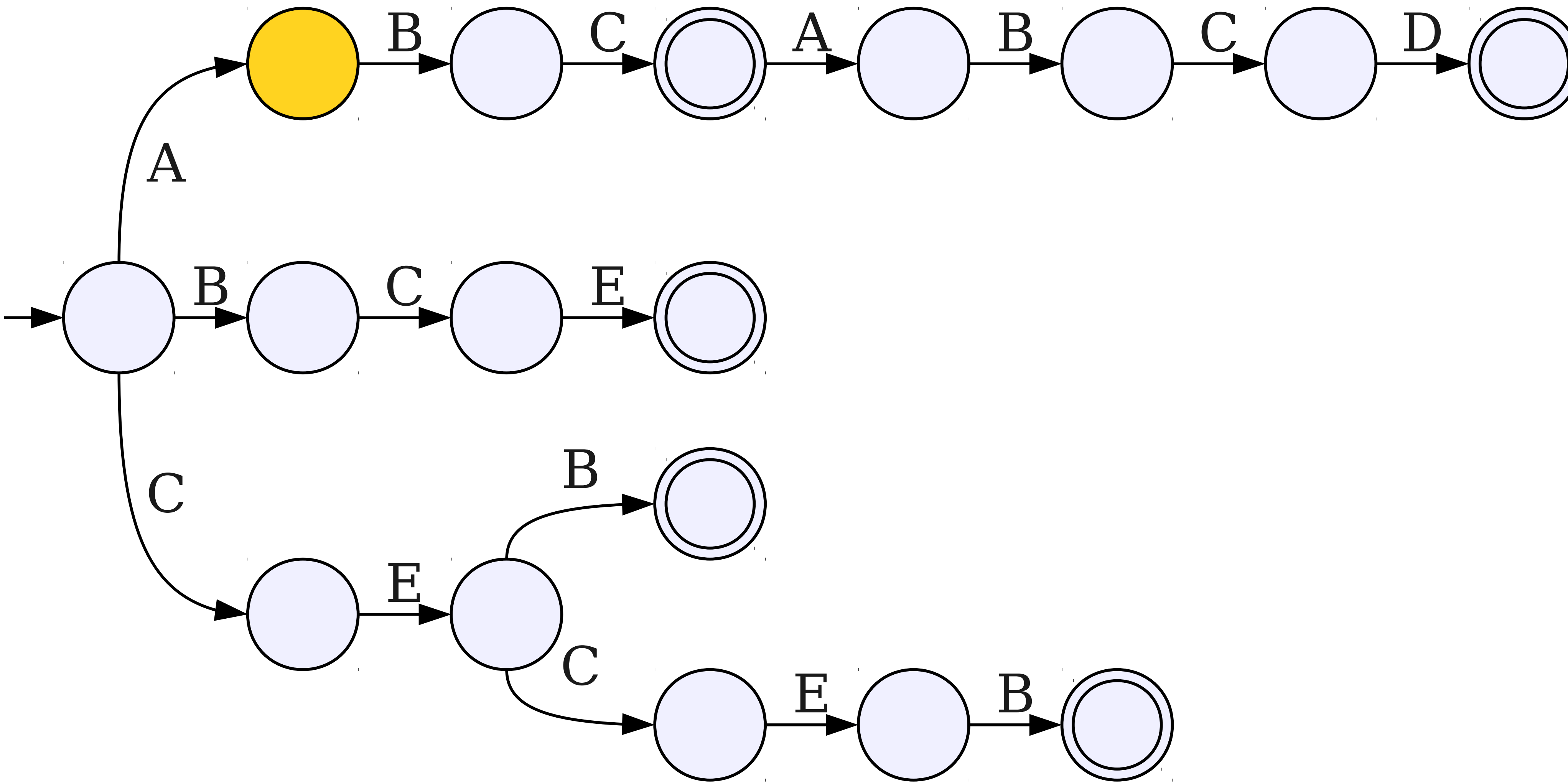
A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



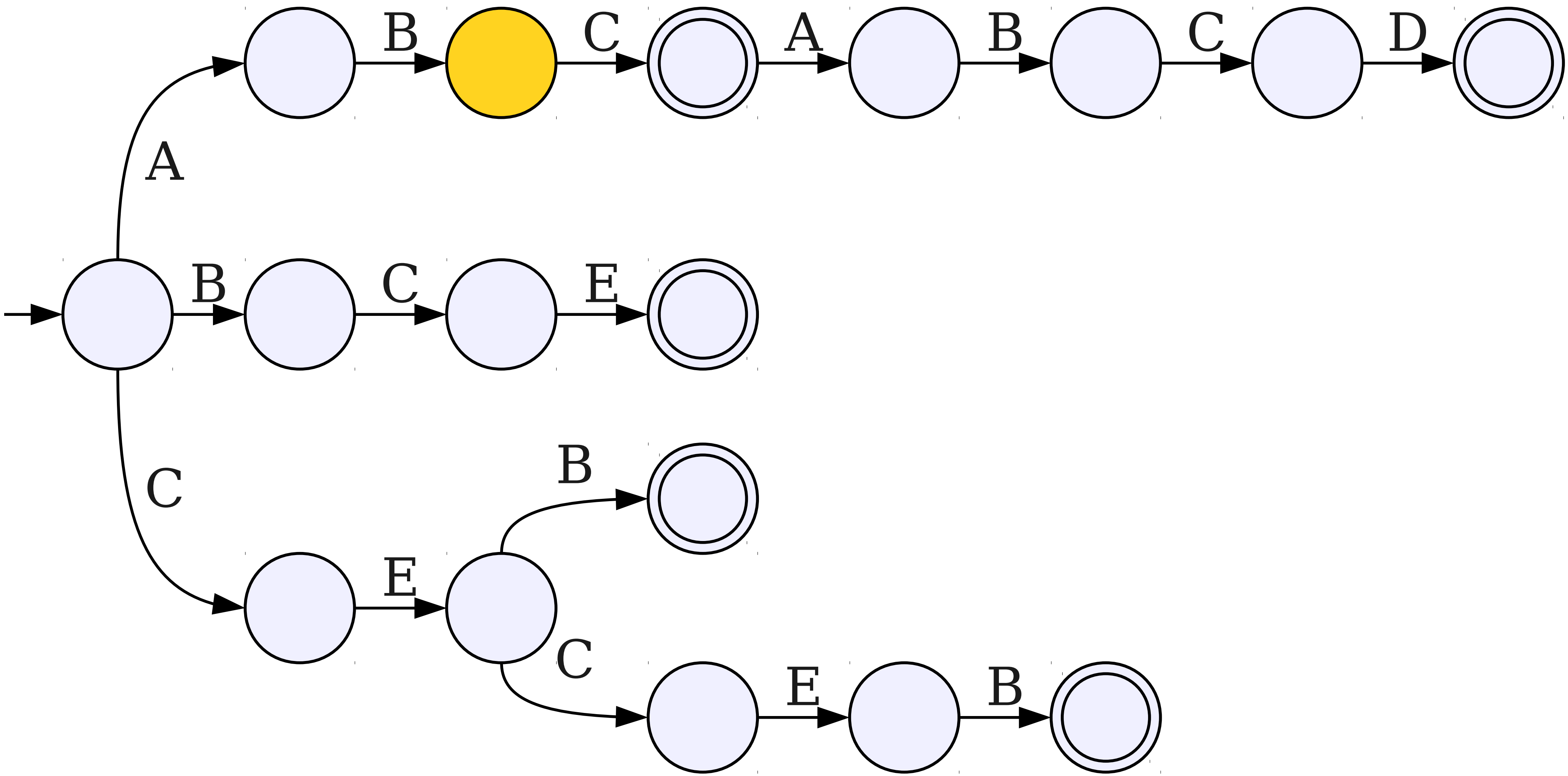
A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



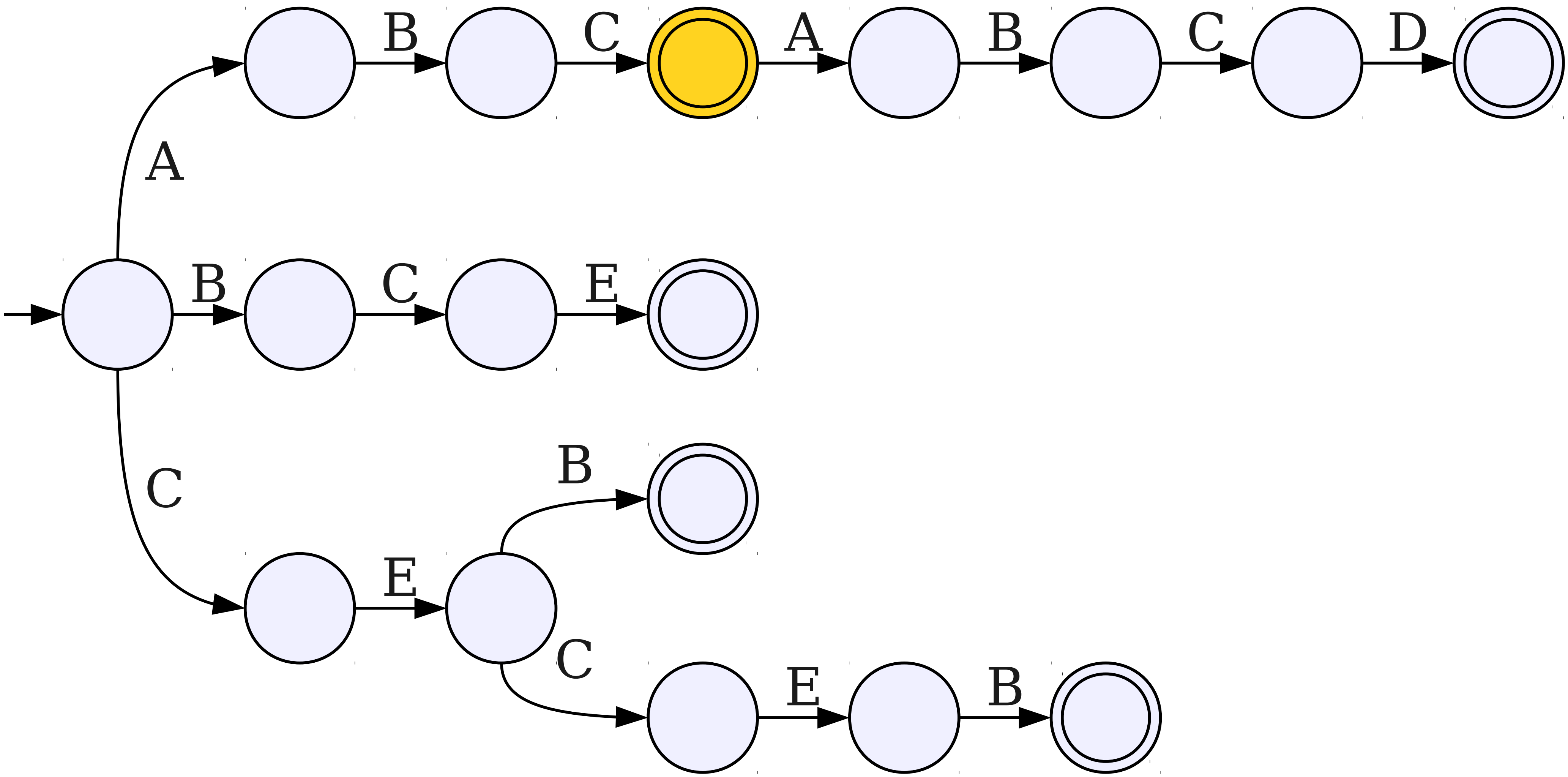
A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



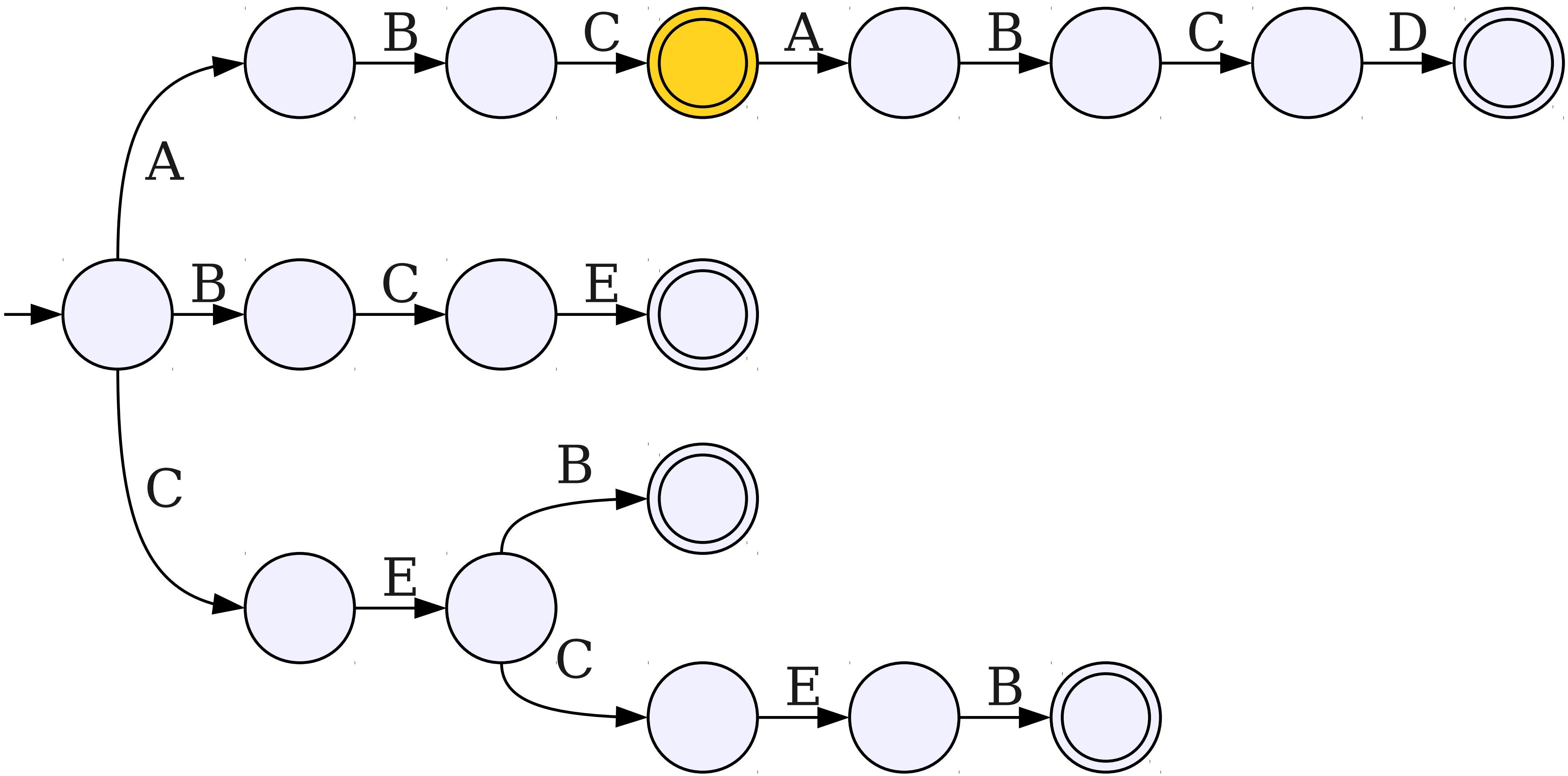
A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



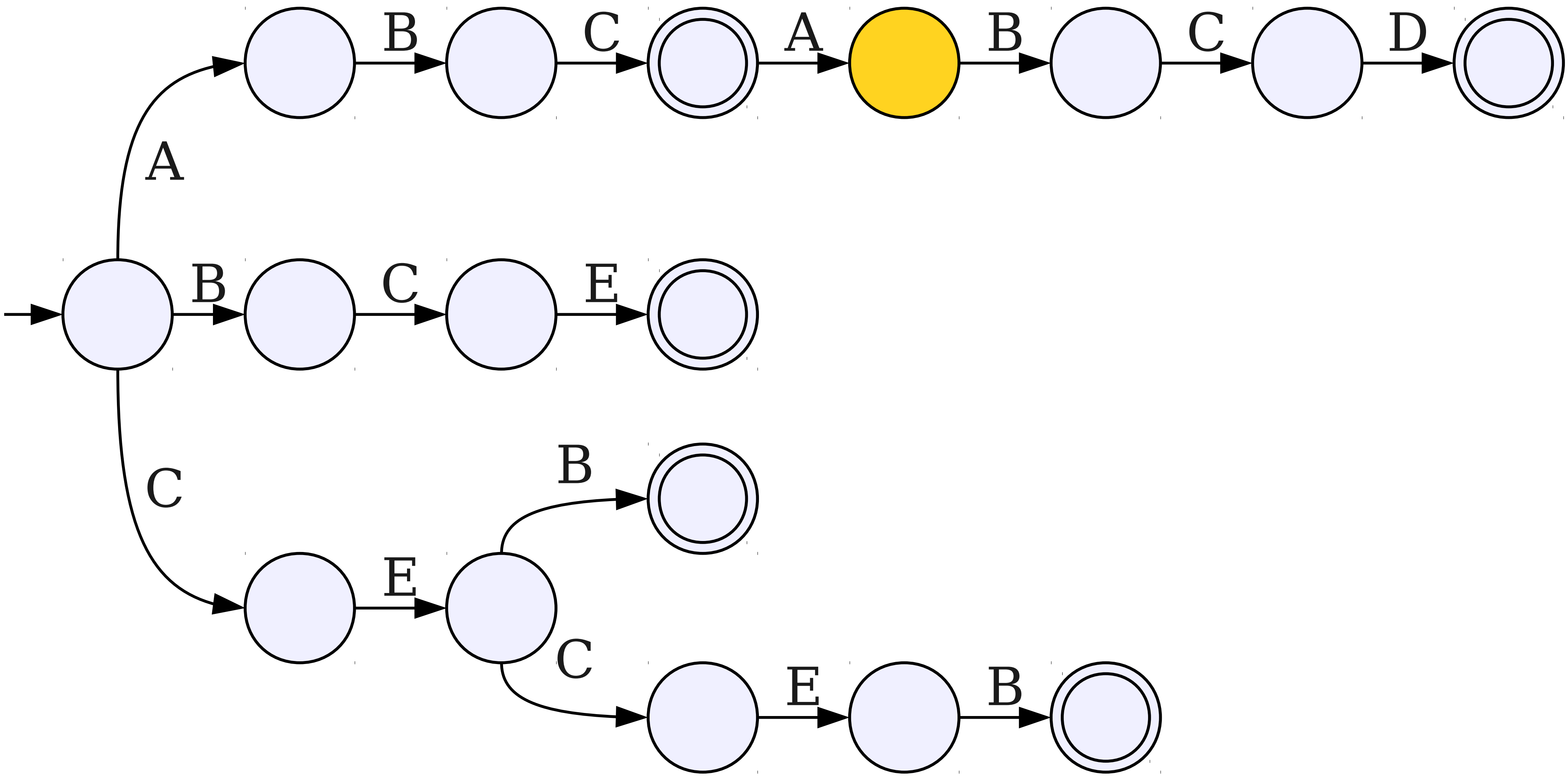
A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



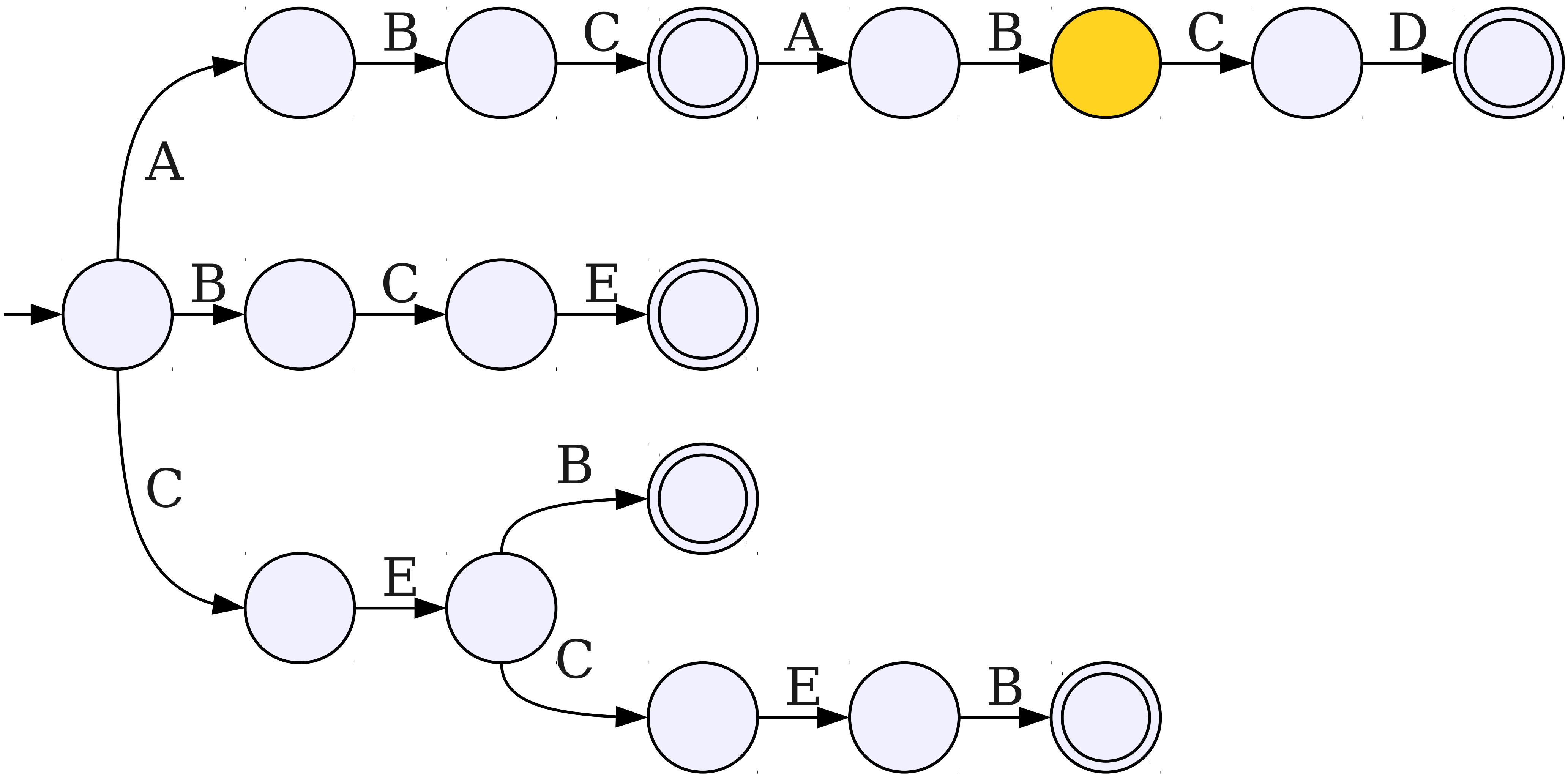
<u>A B C</u> A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



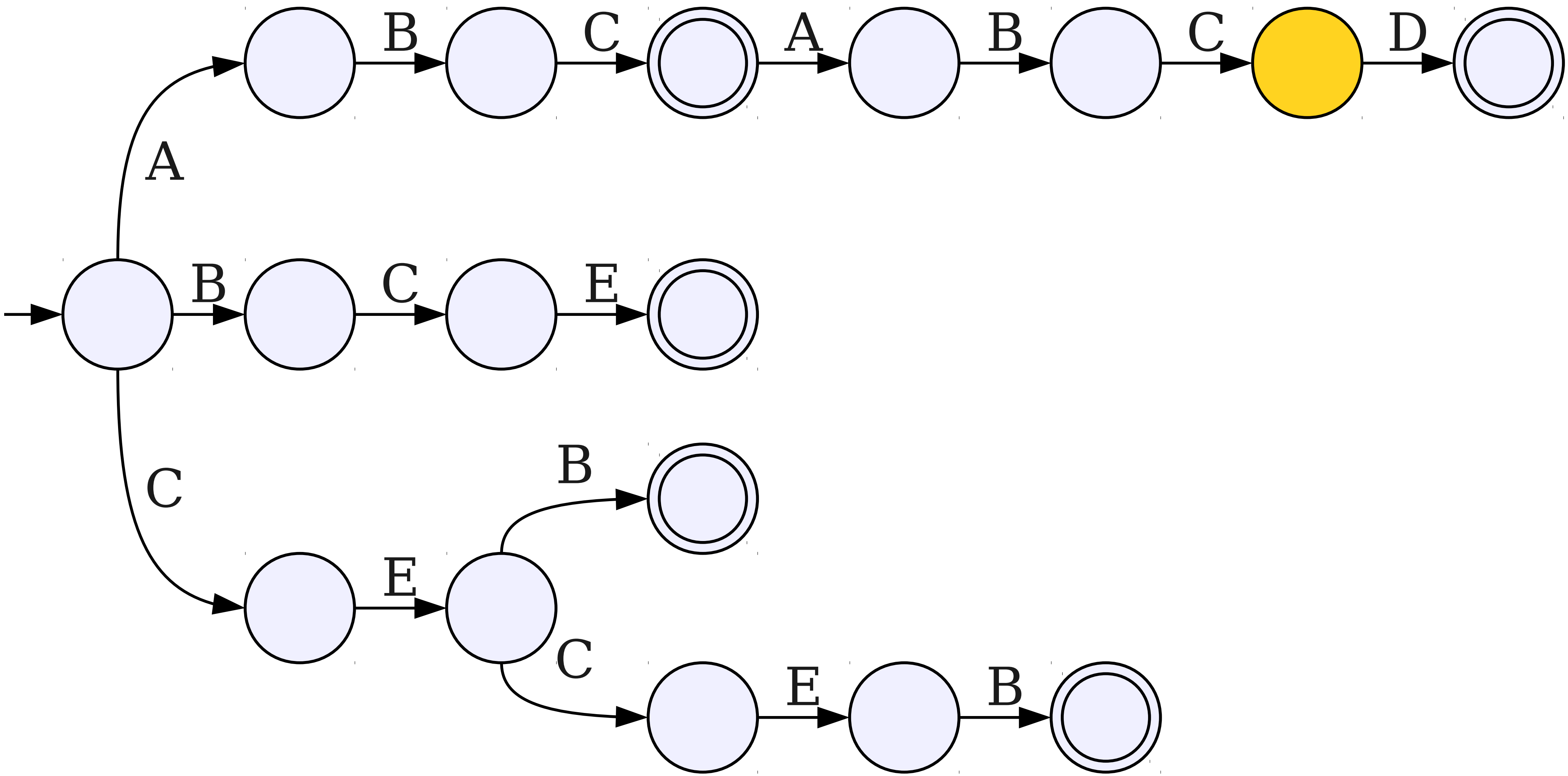
<u>A B C</u> A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



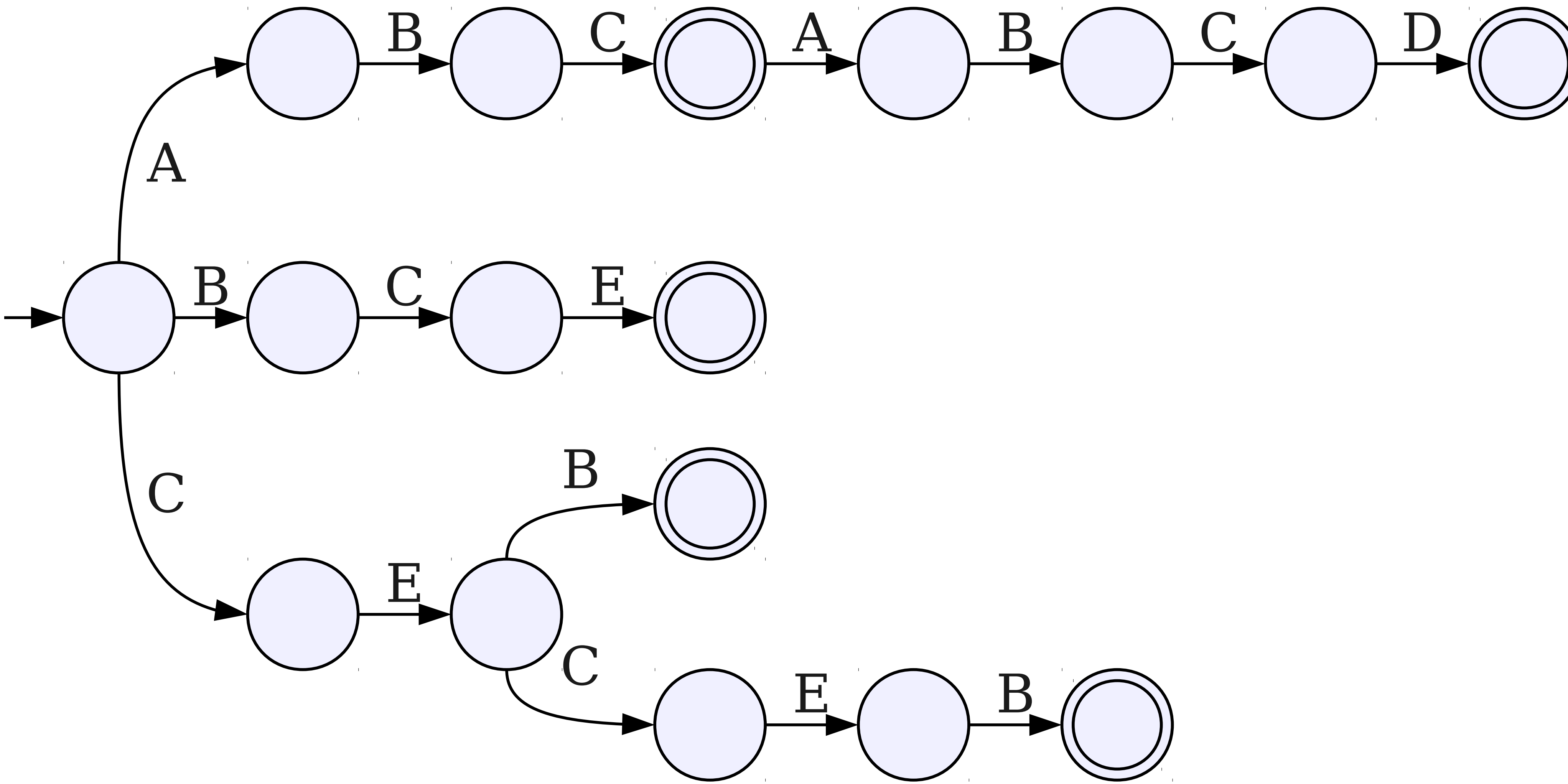
A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



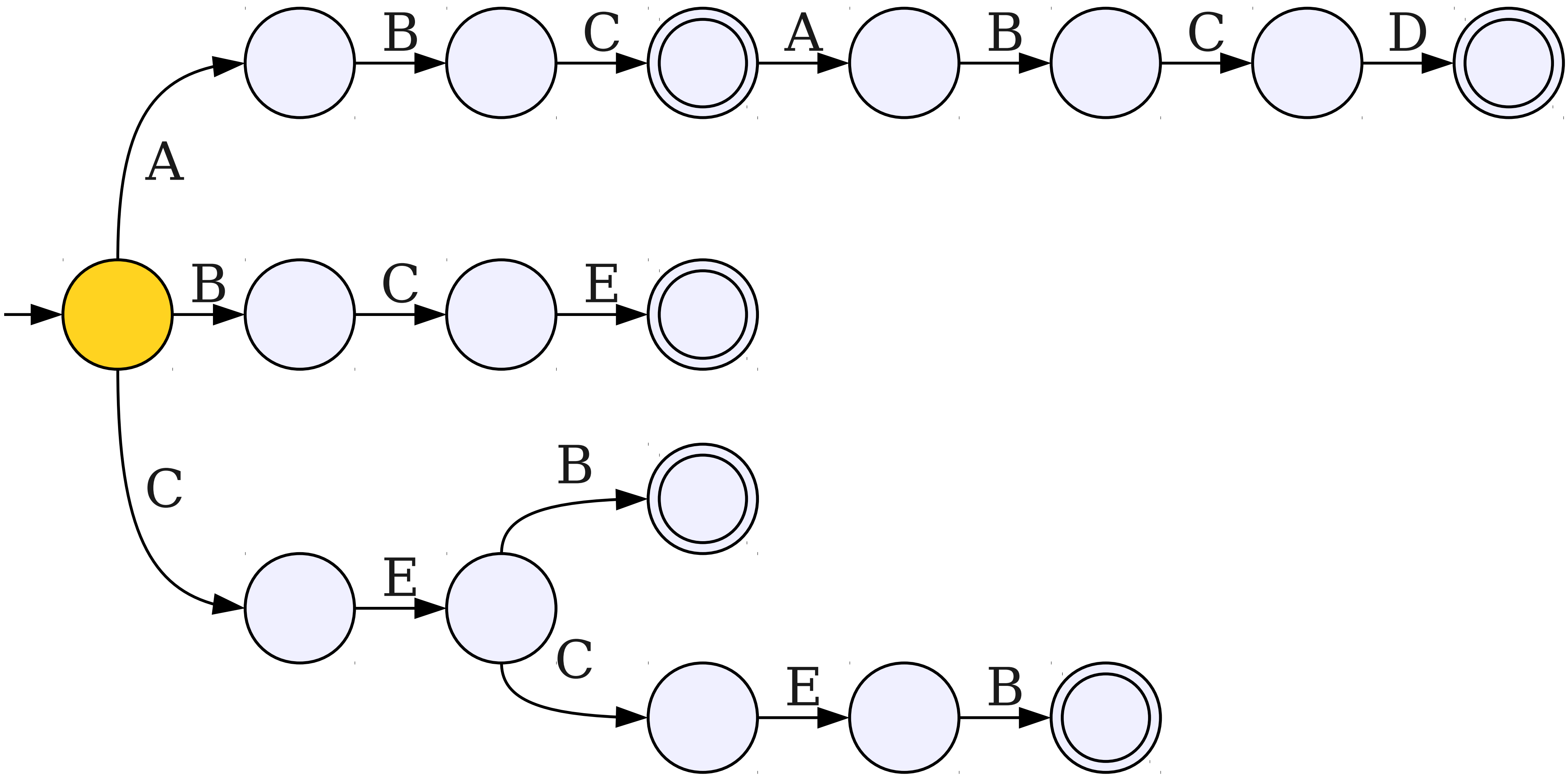
A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



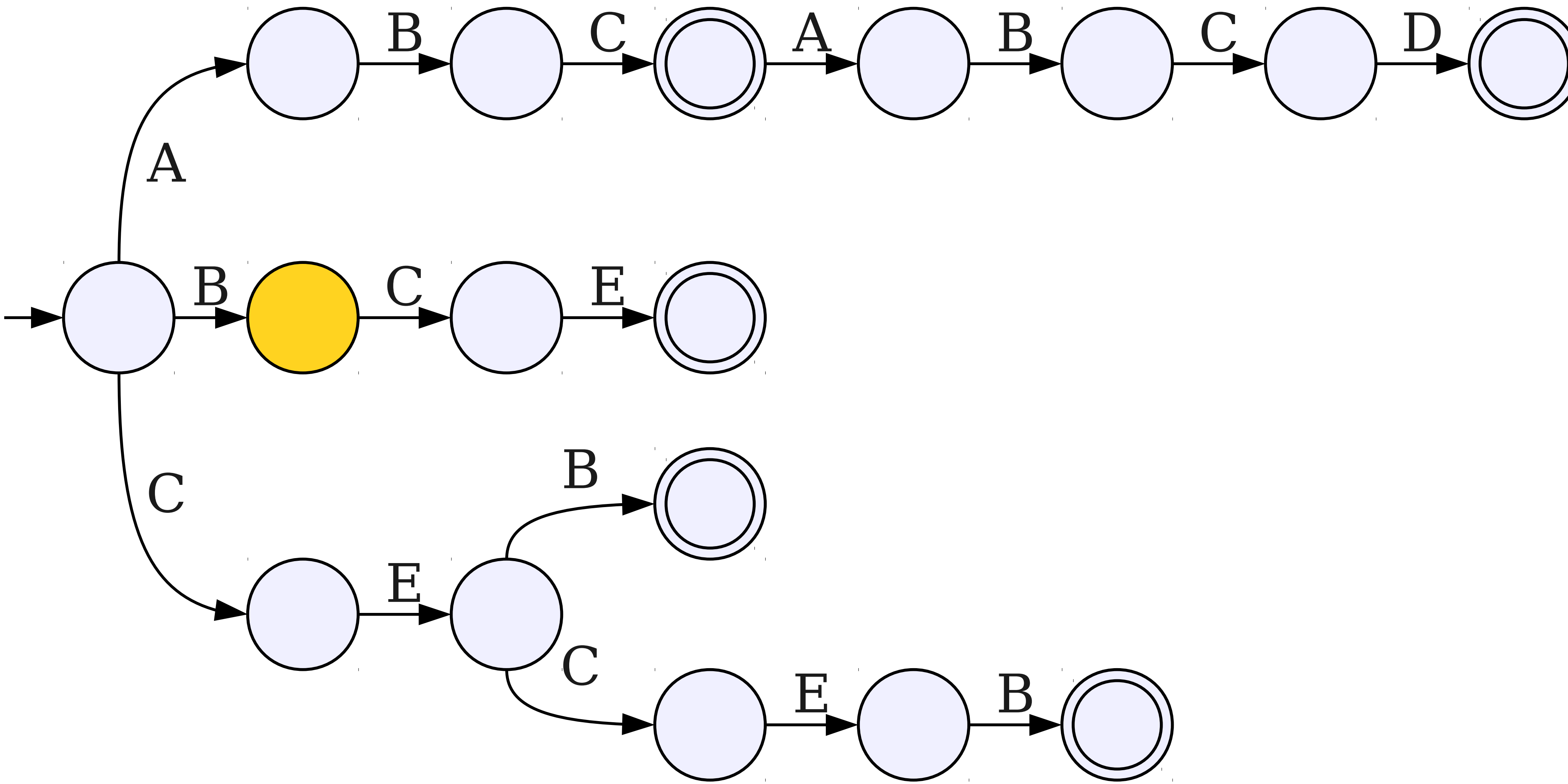
<u>A B C</u> A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



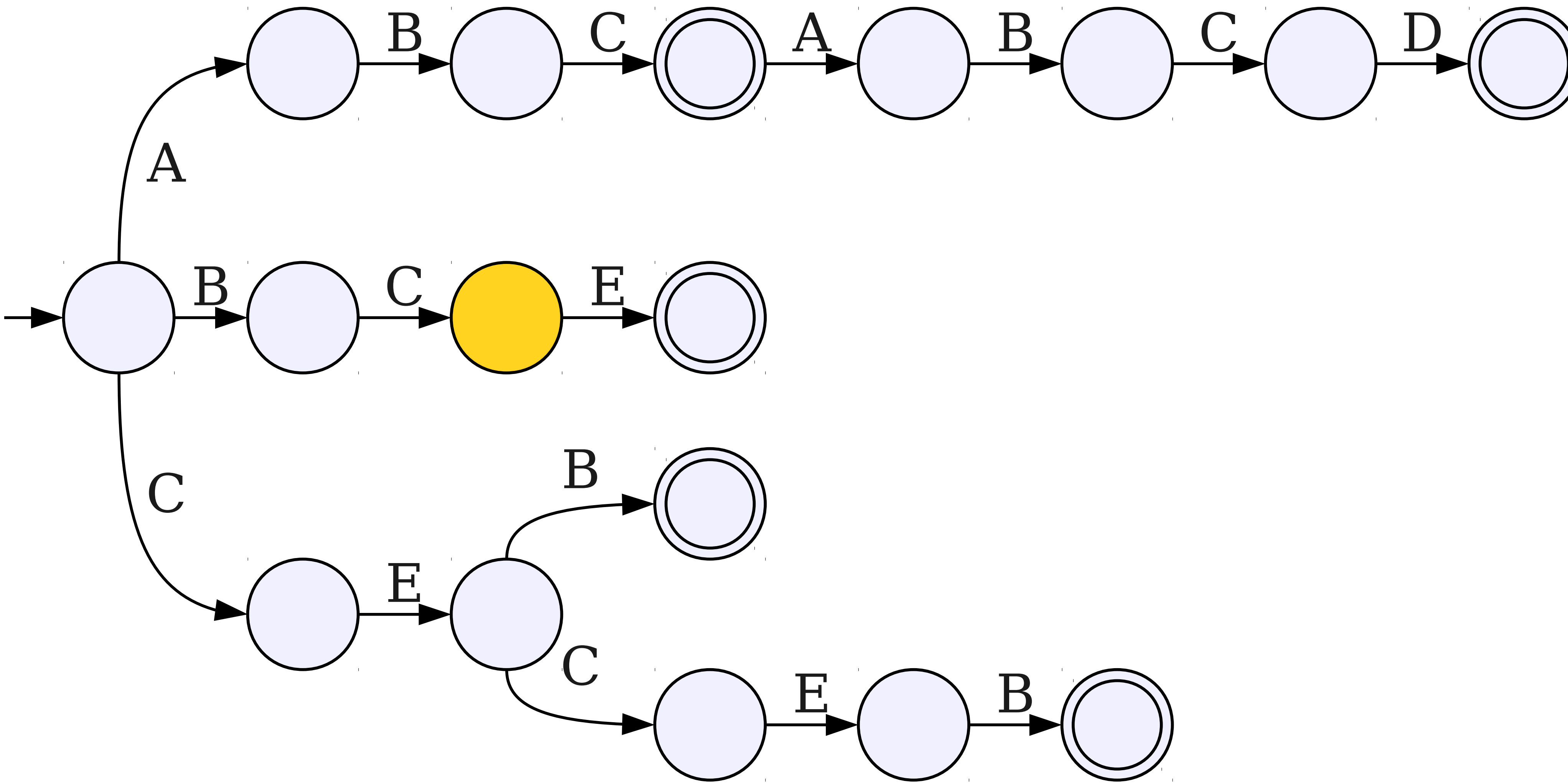
A B C A B C E B E B

$P_1 = \mathrm{ABCABCD}$ $P_2 = \mathrm{BCE}$ $P_3 = \mathrm{CEB}$ $P_4 = \mathrm{CECEB}$ $P_5 = \mathrm{ABC}$



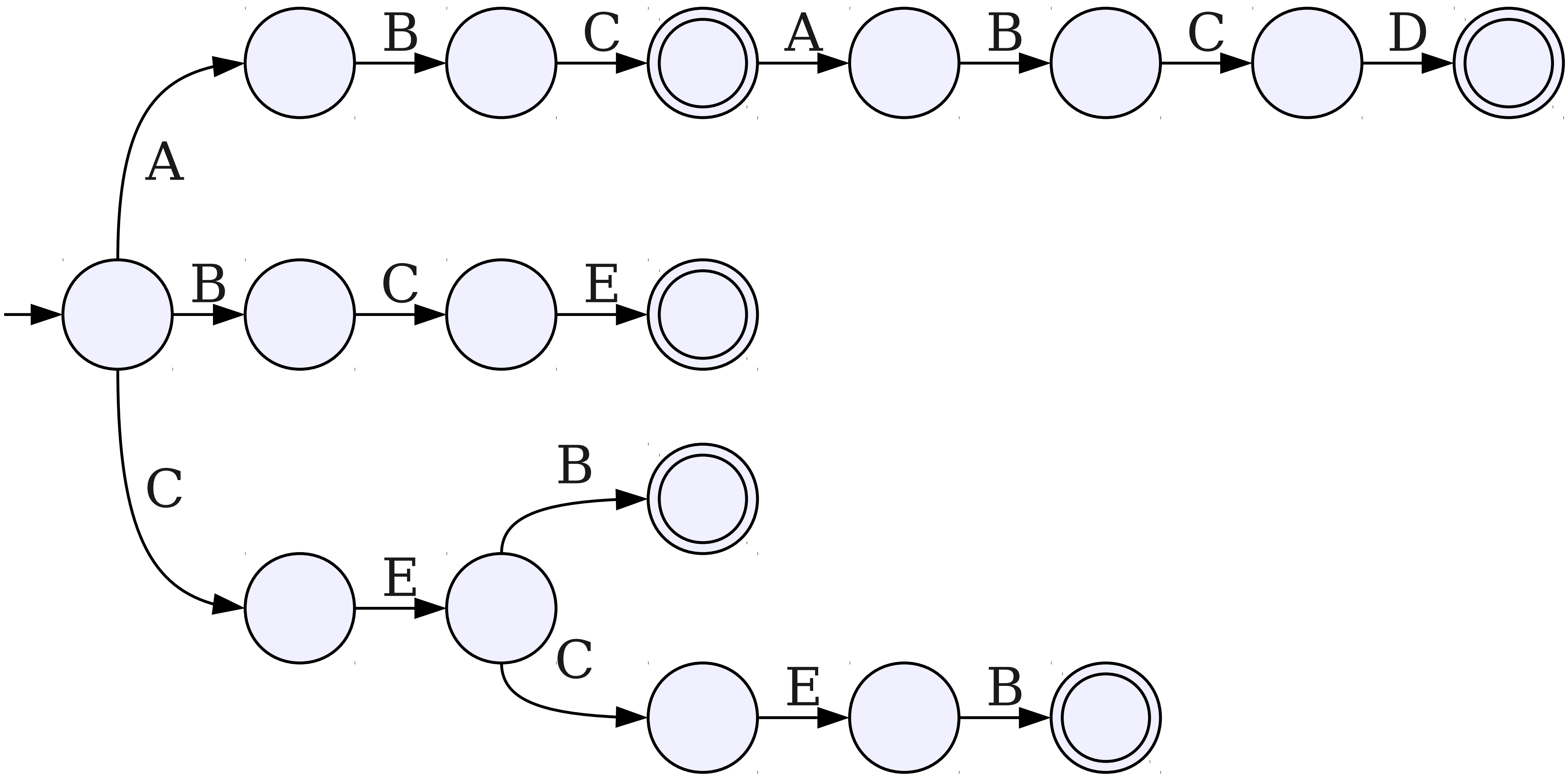
<u>A B C</u> A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



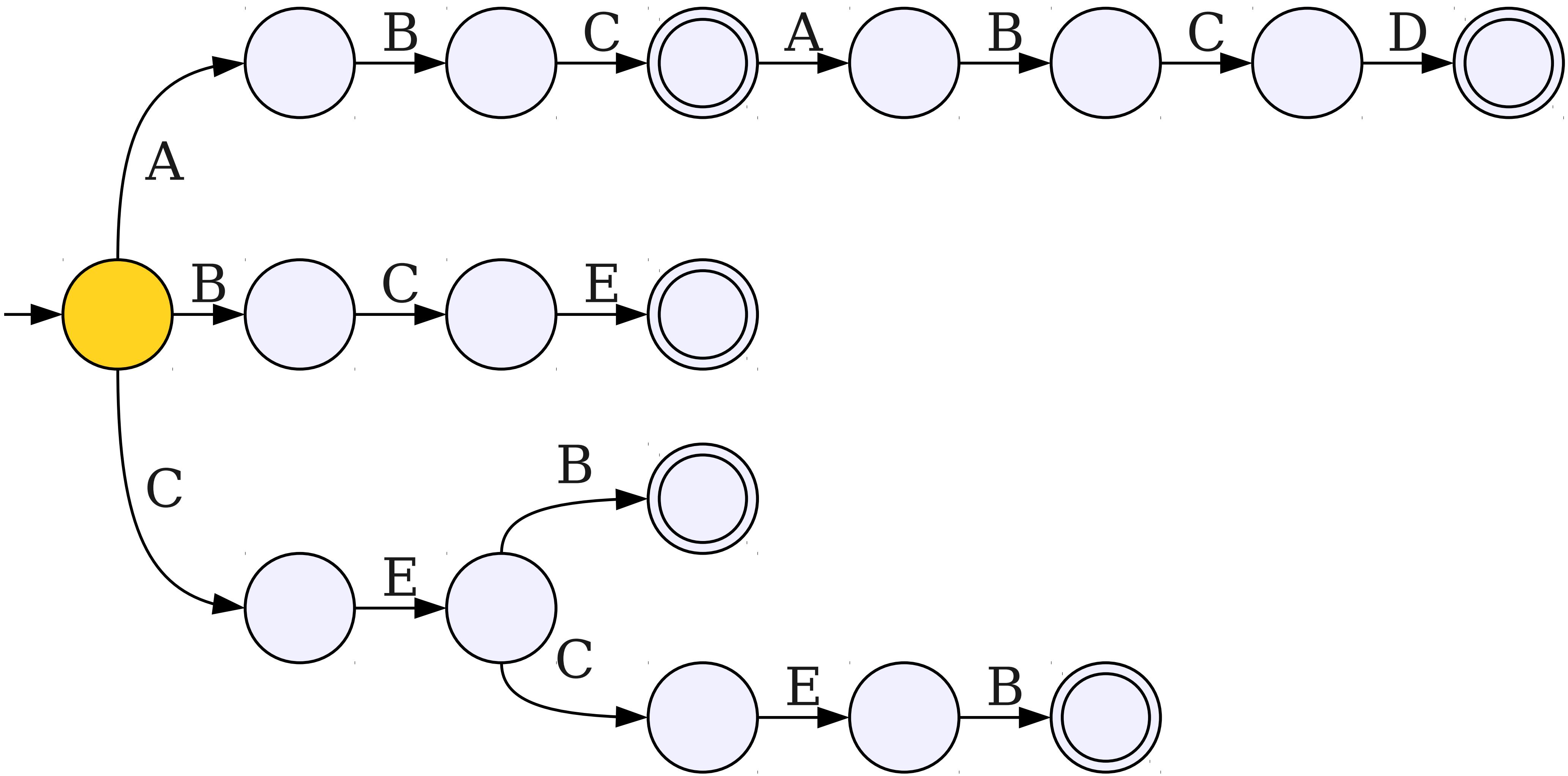
<u>A B C</u> A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



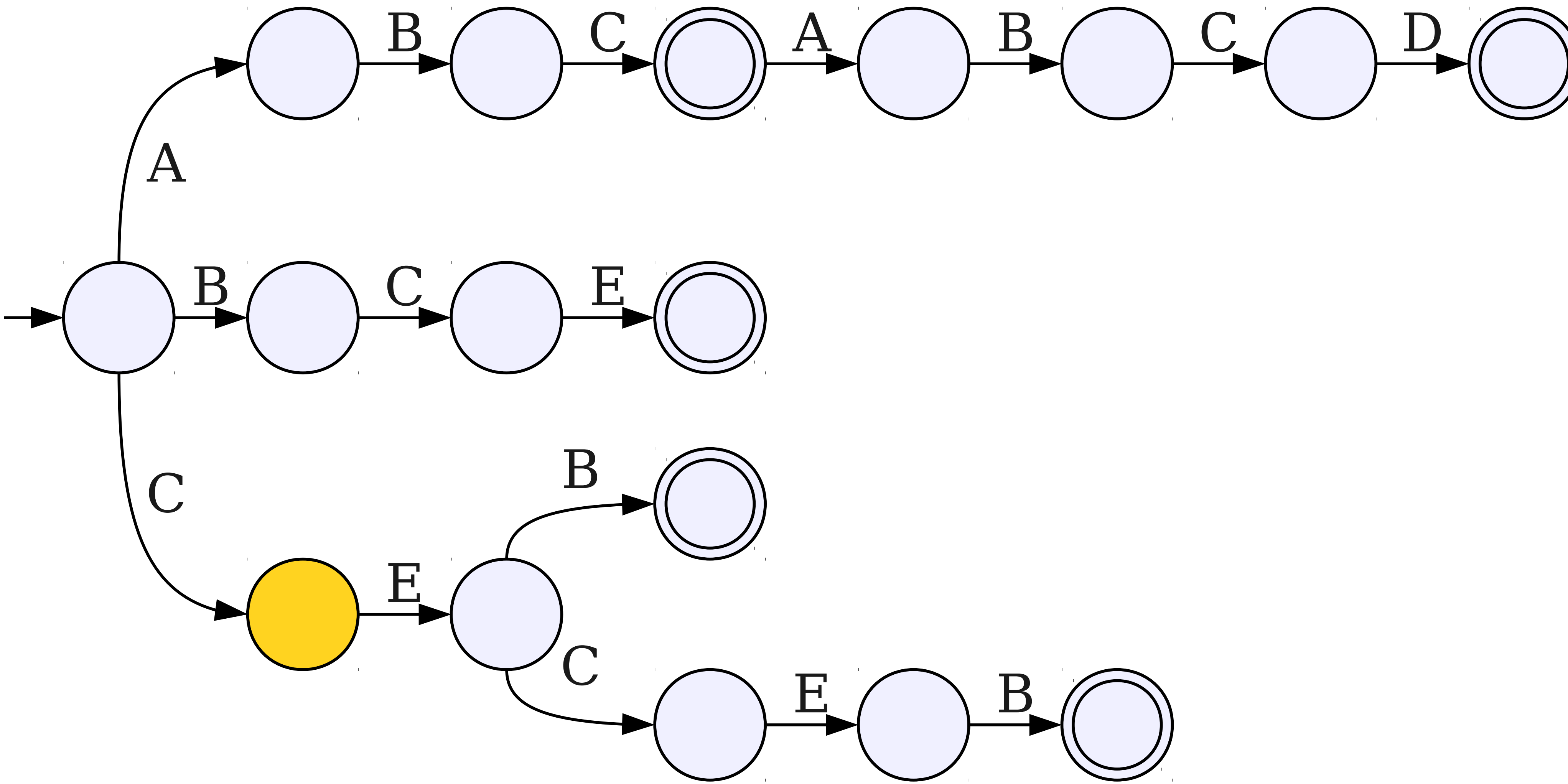
<u>A B C</u> A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



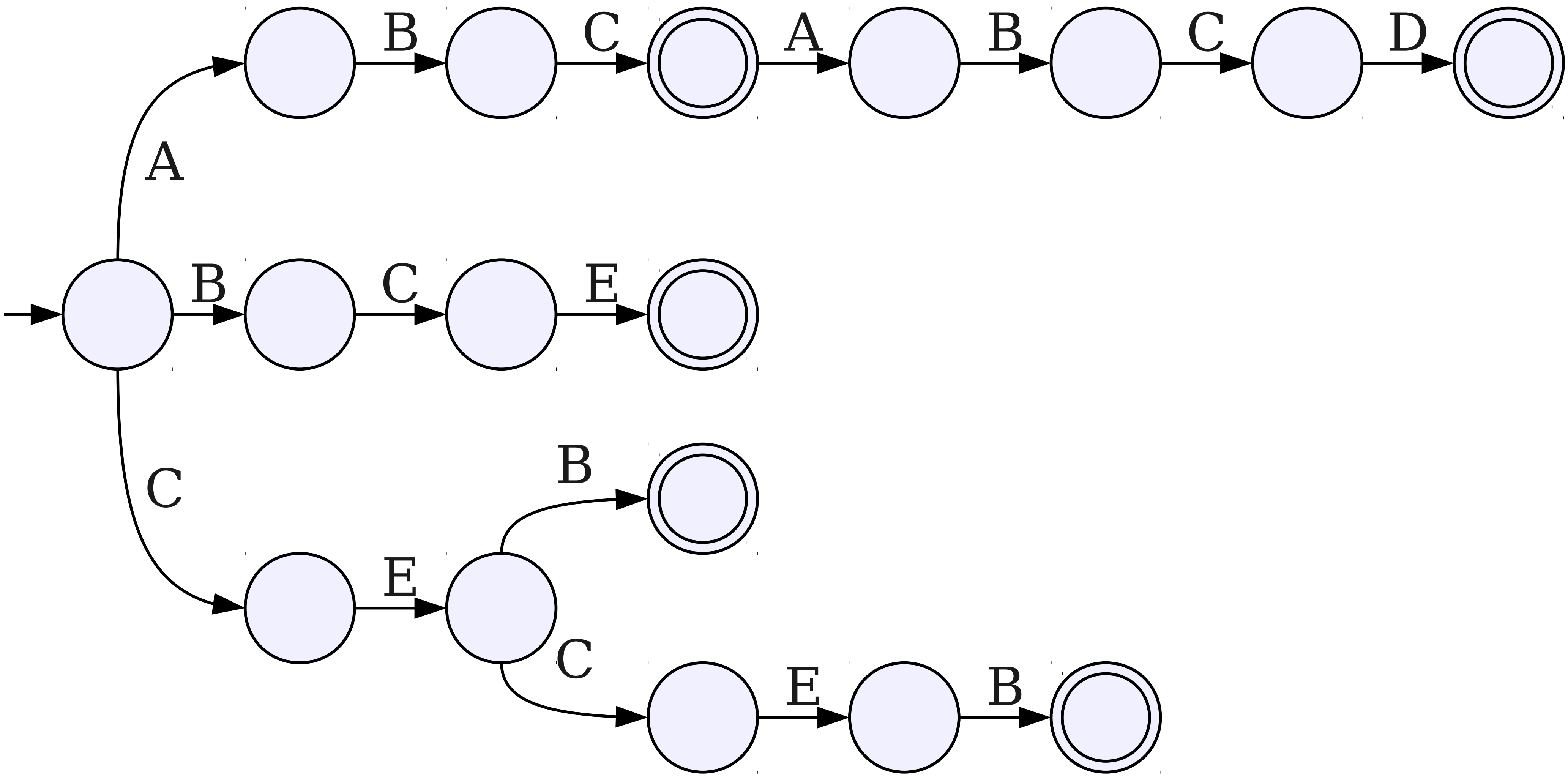
A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



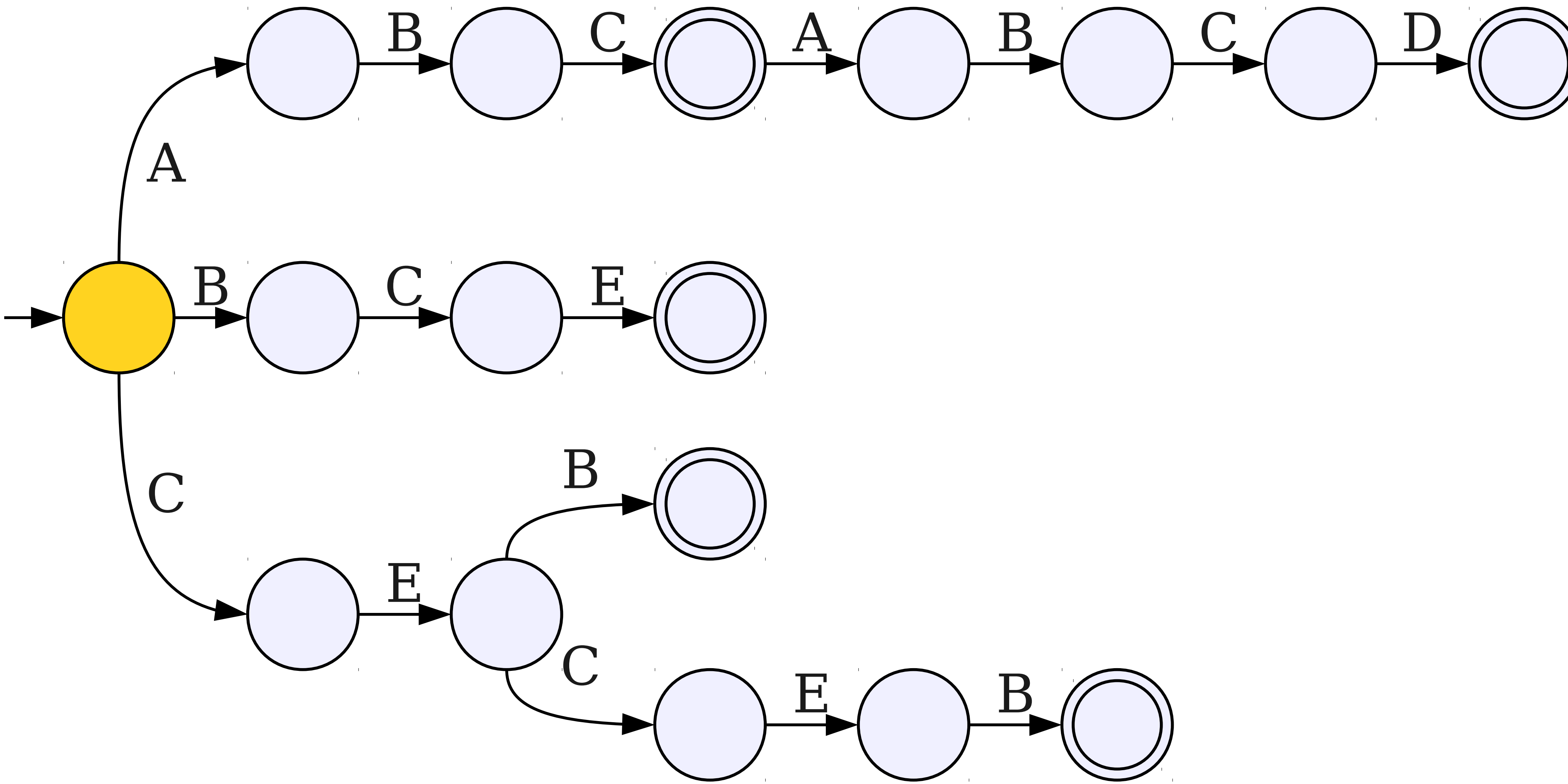
A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



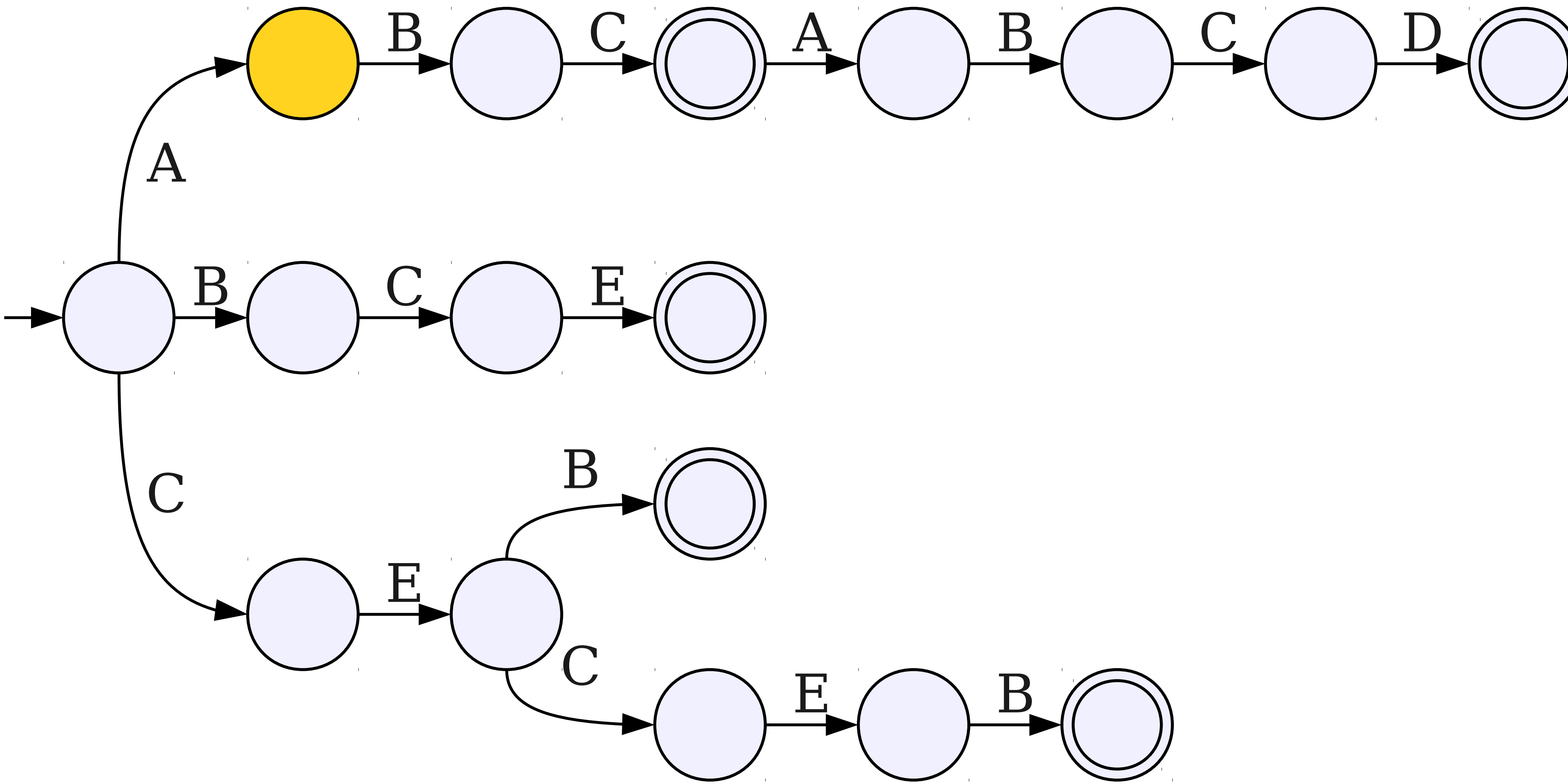
A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



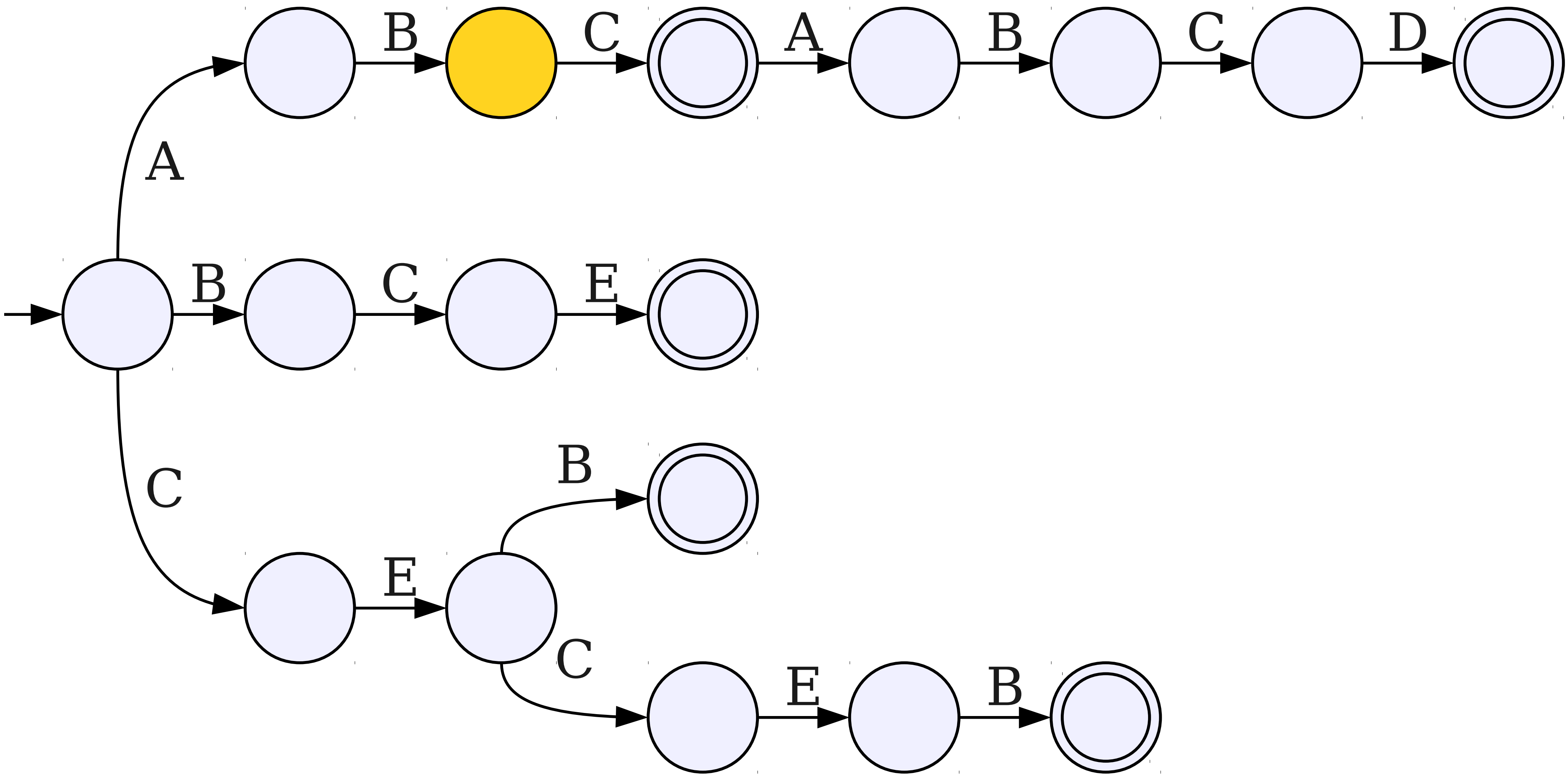
A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



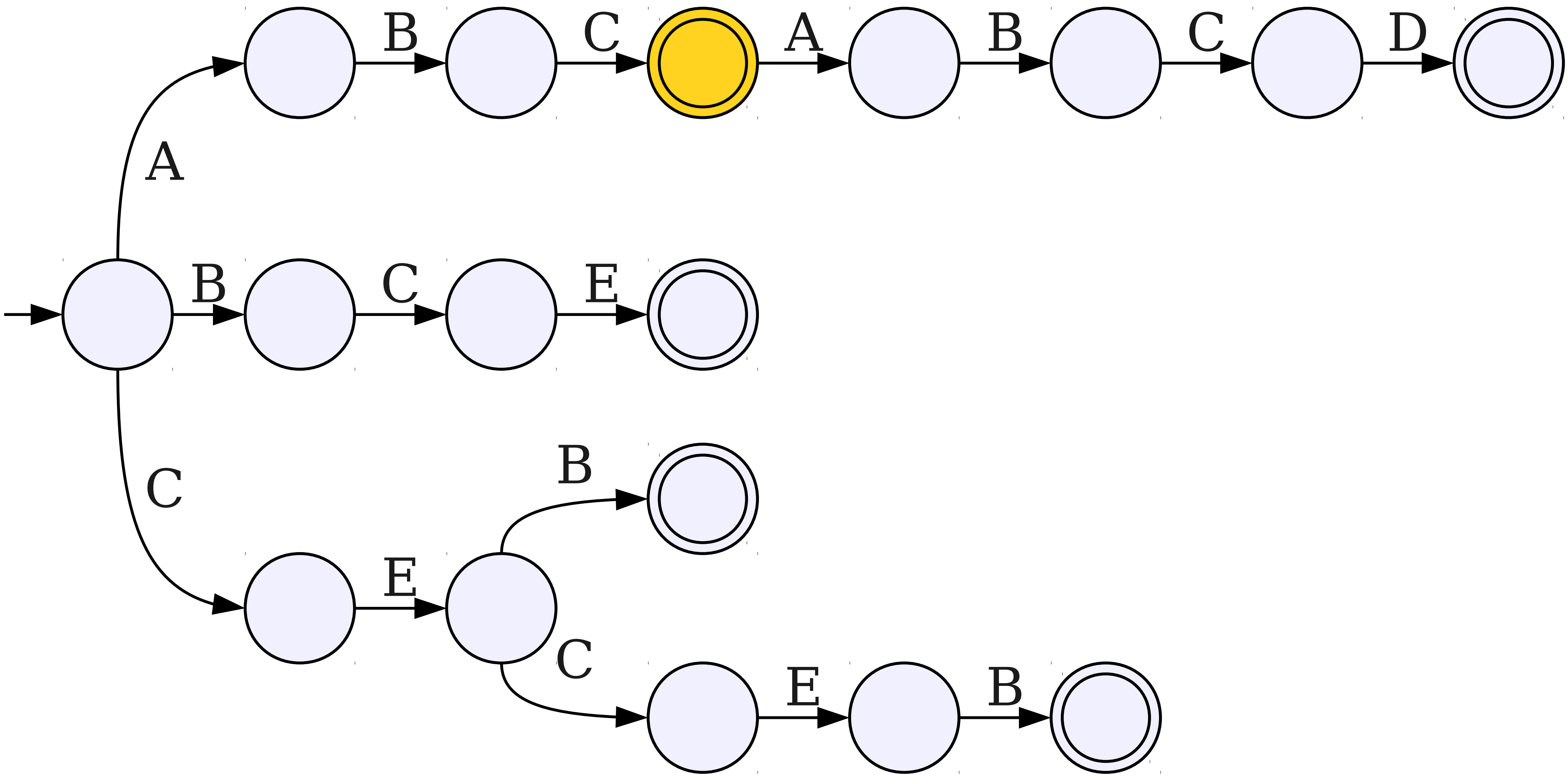
<u>A B C</u> A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



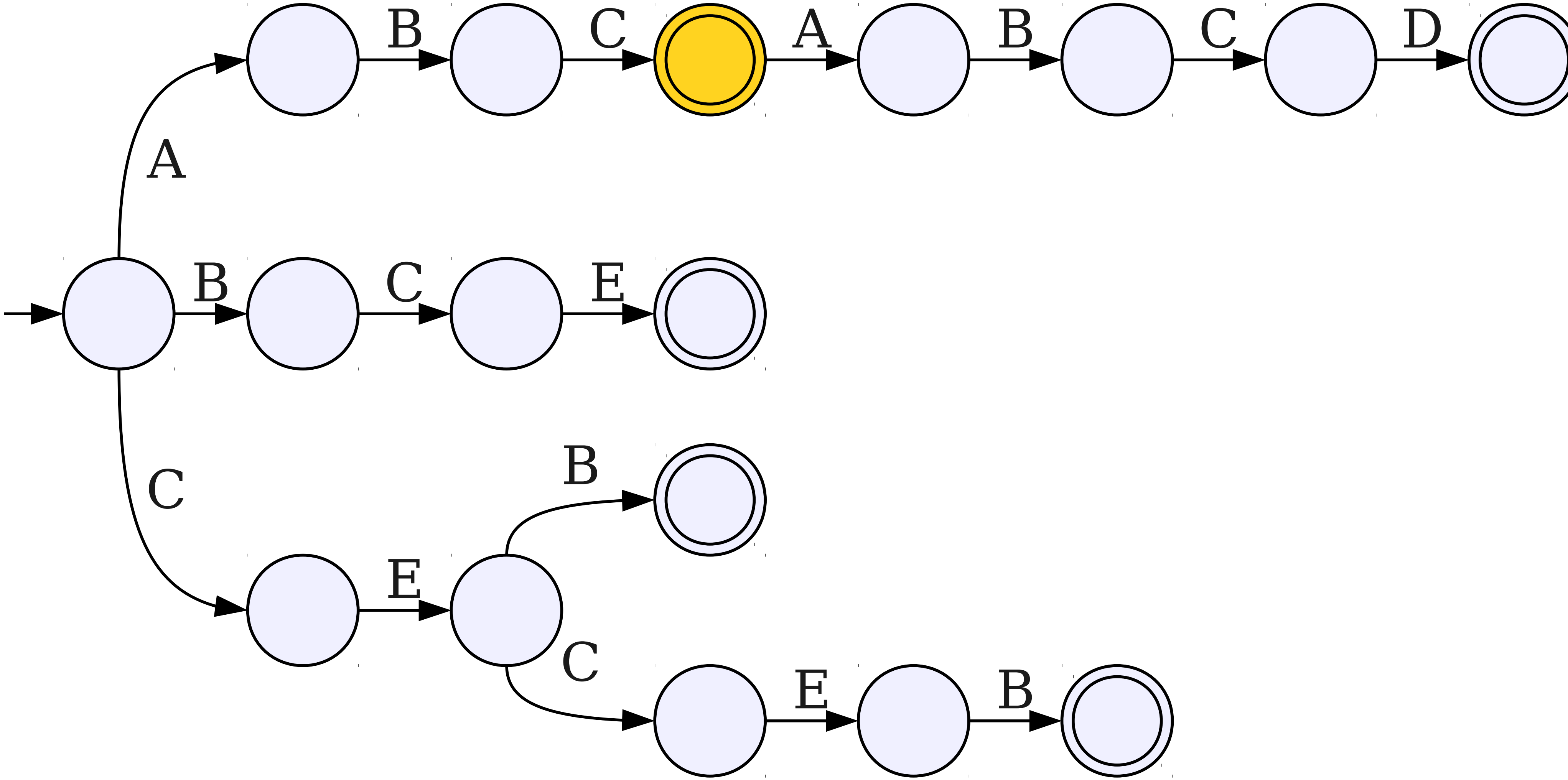
A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



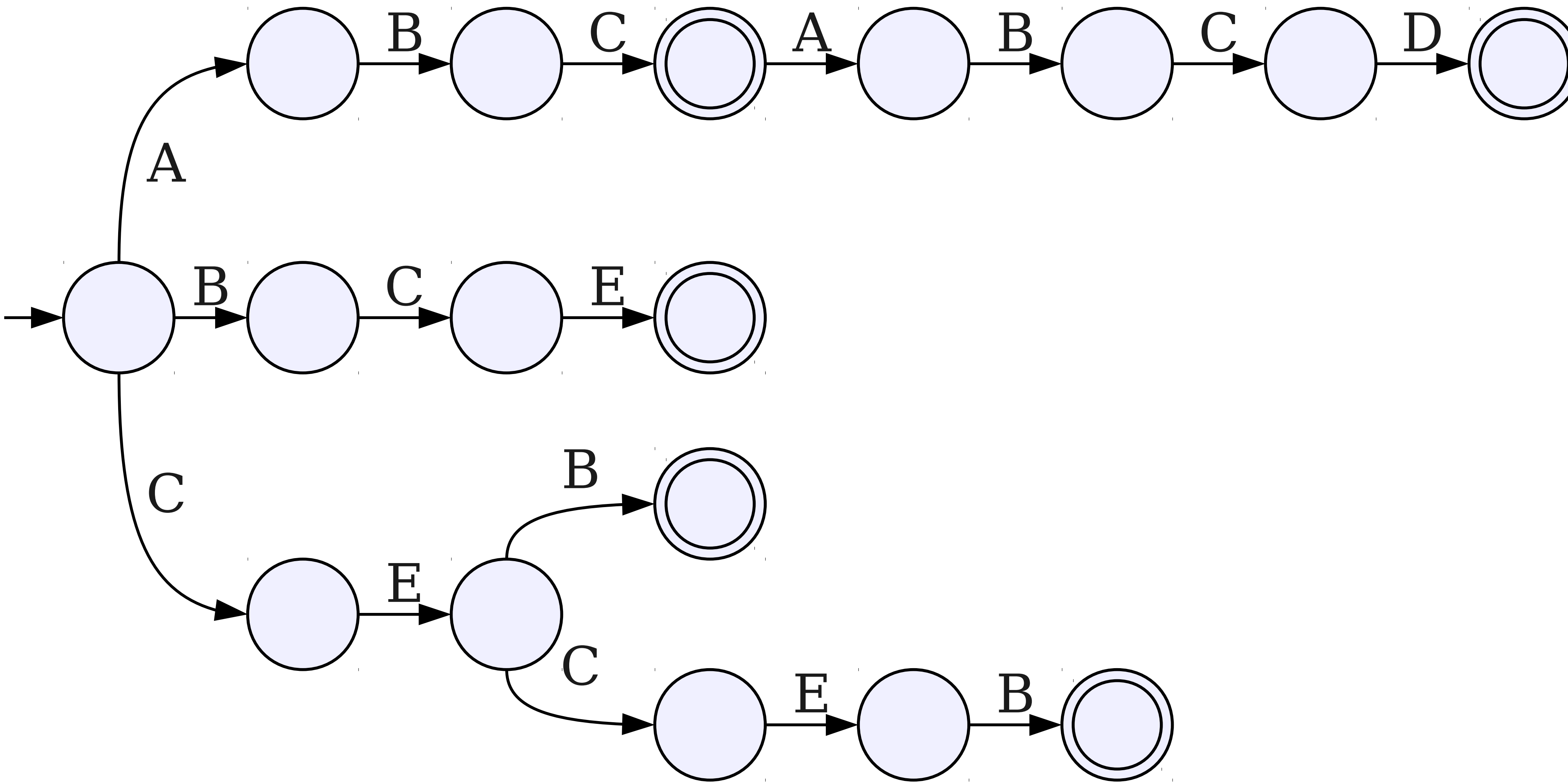
<u>A B C</u> A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



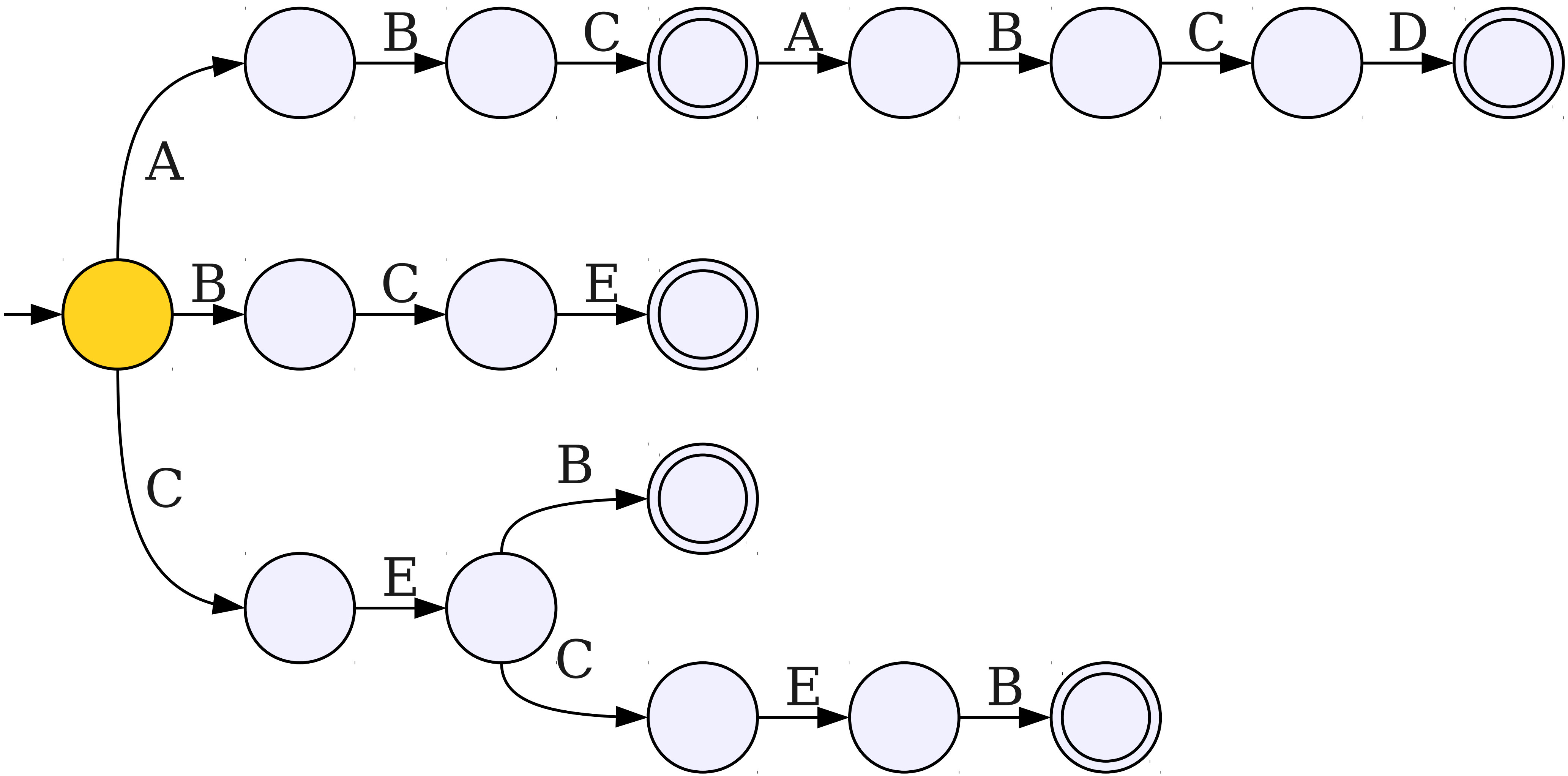
<u>A B C</u> <u>A B C</u> E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



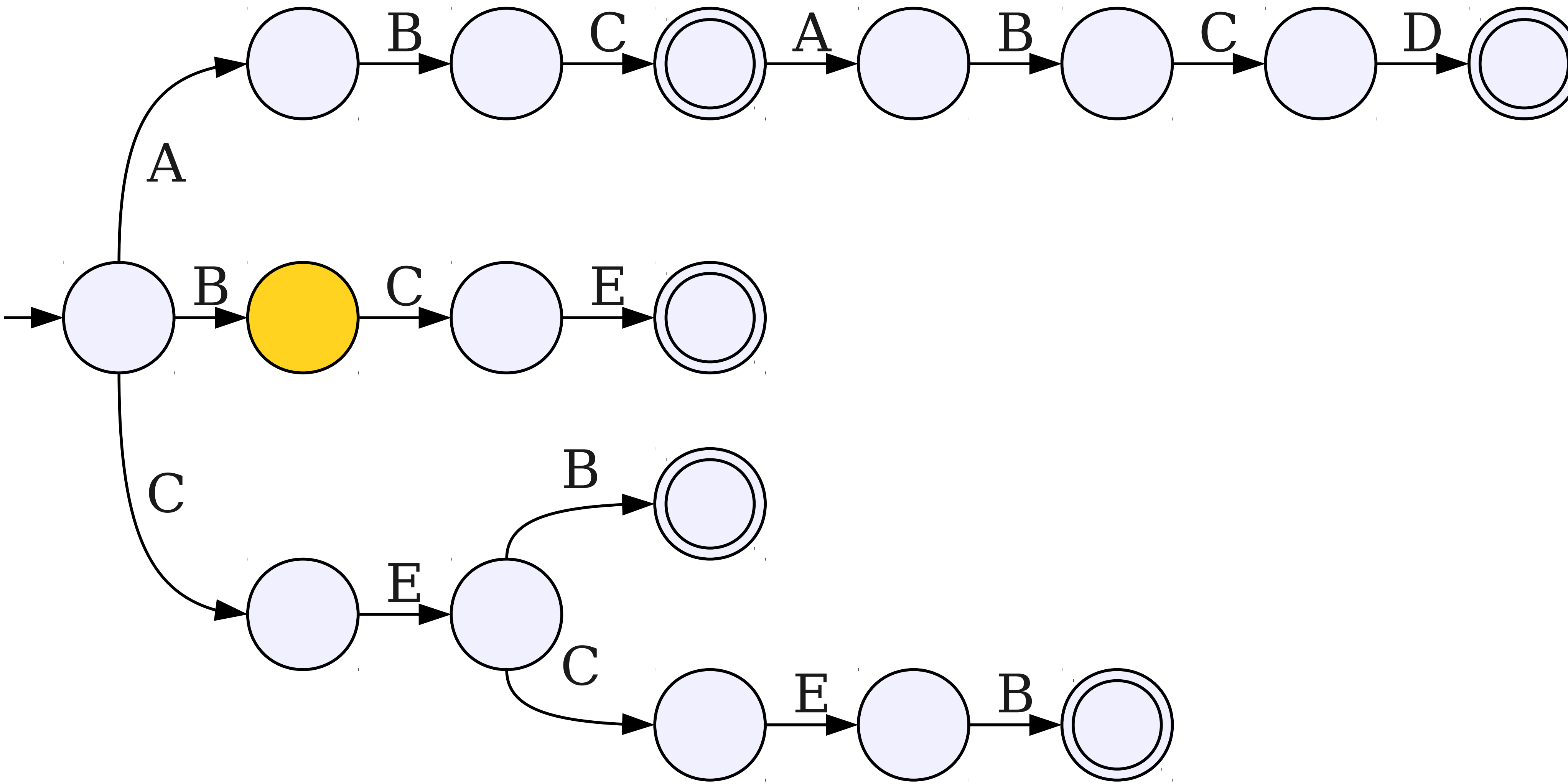
A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



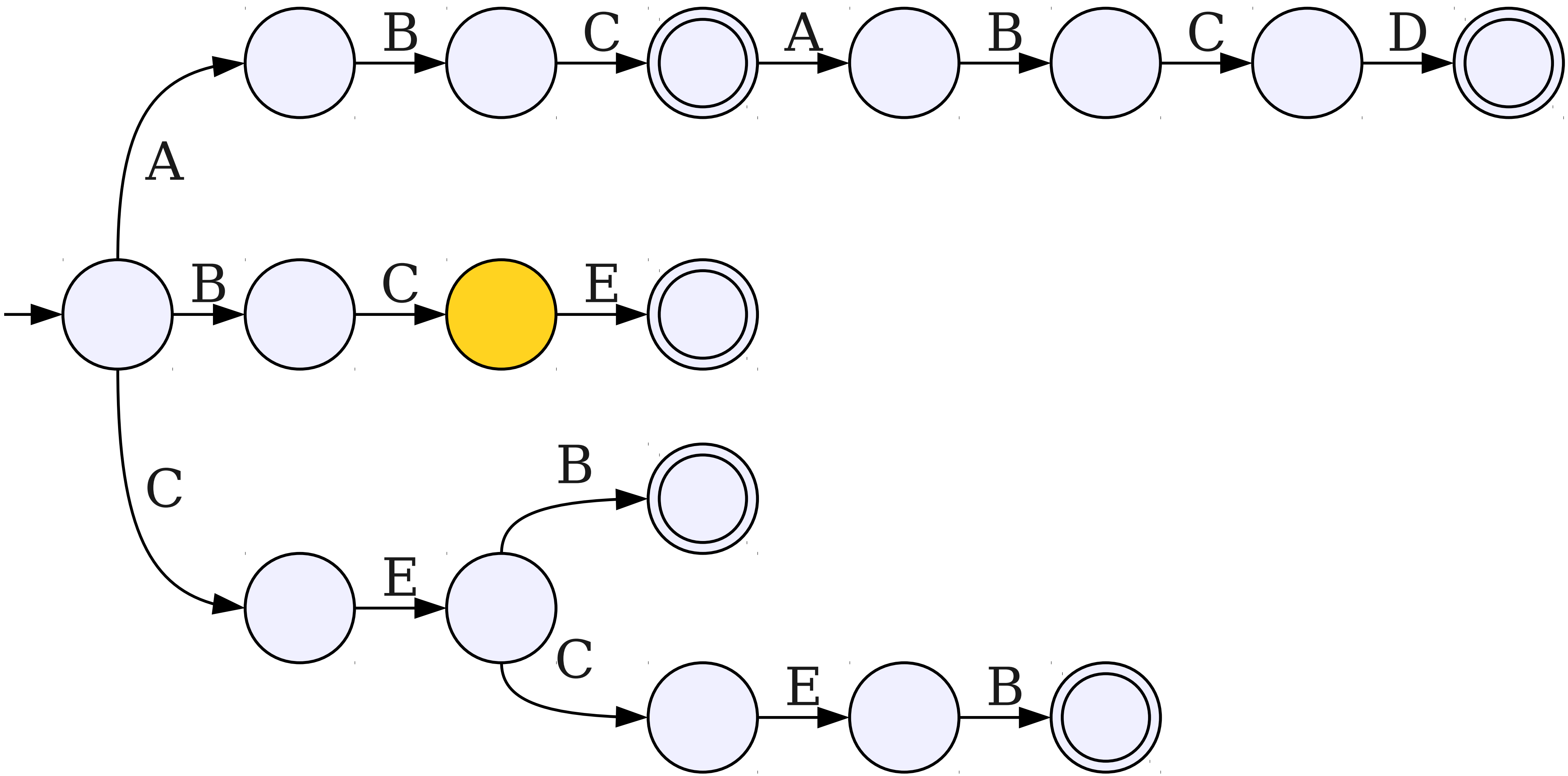
A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



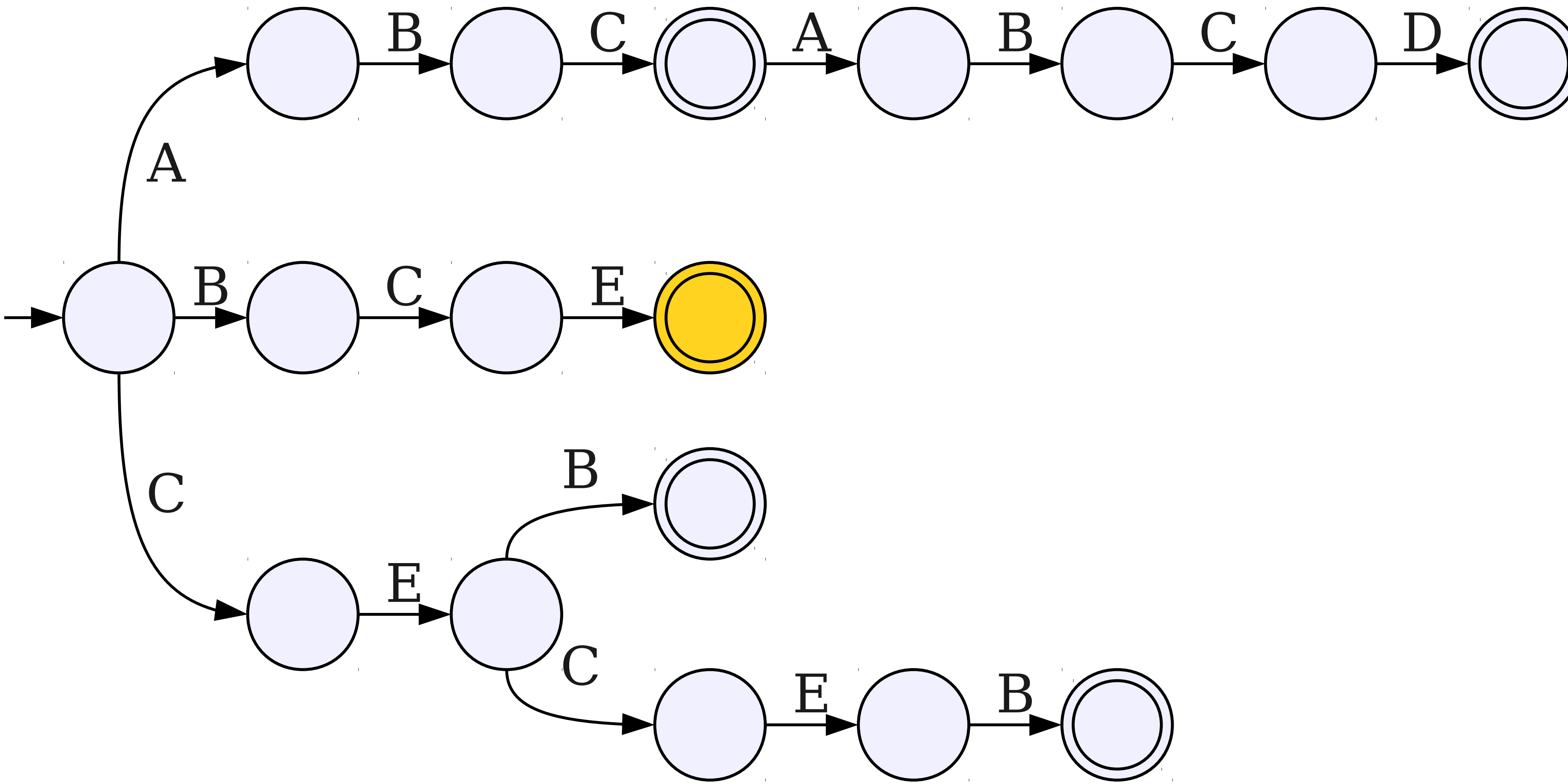
A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



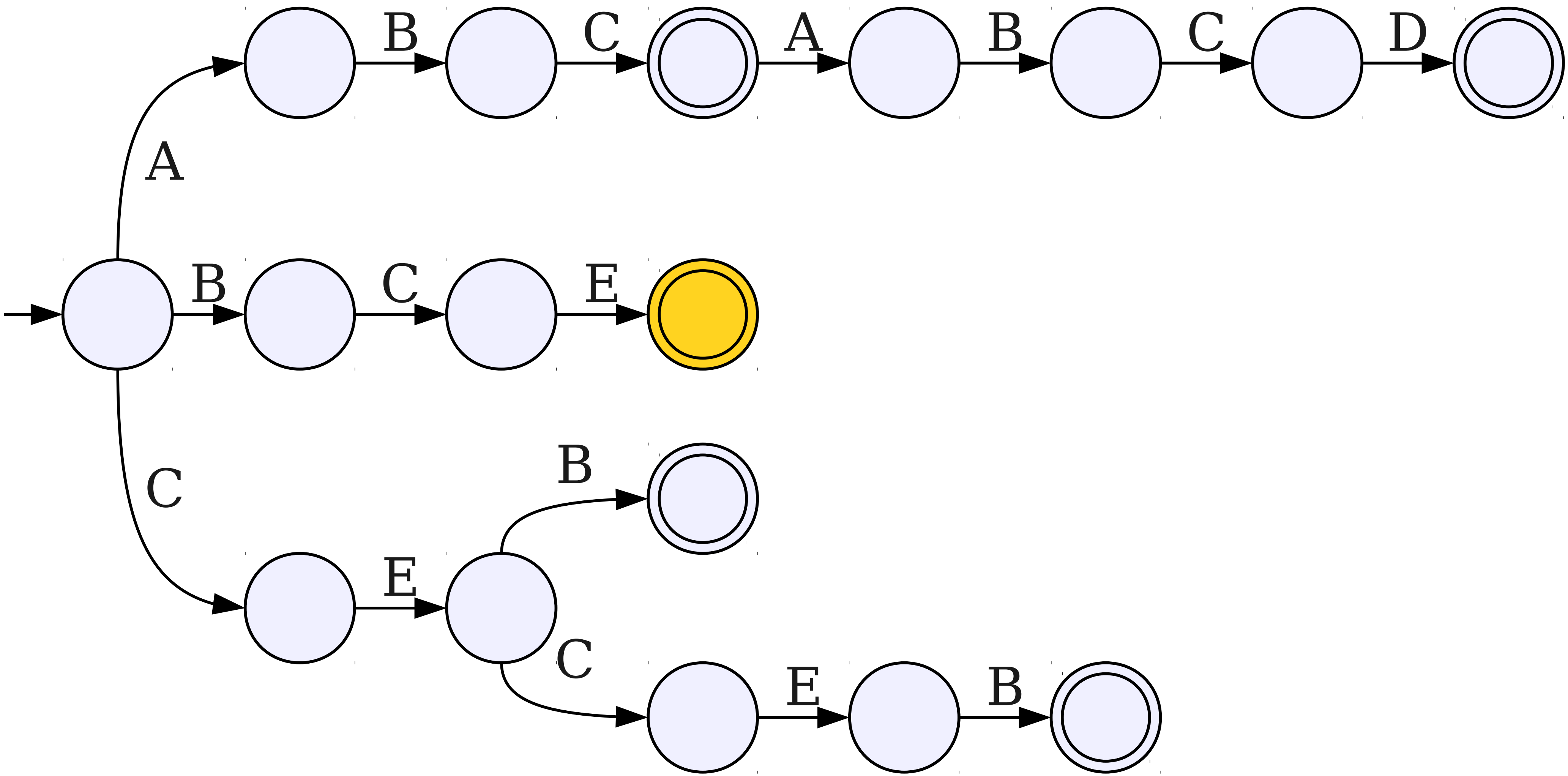
A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



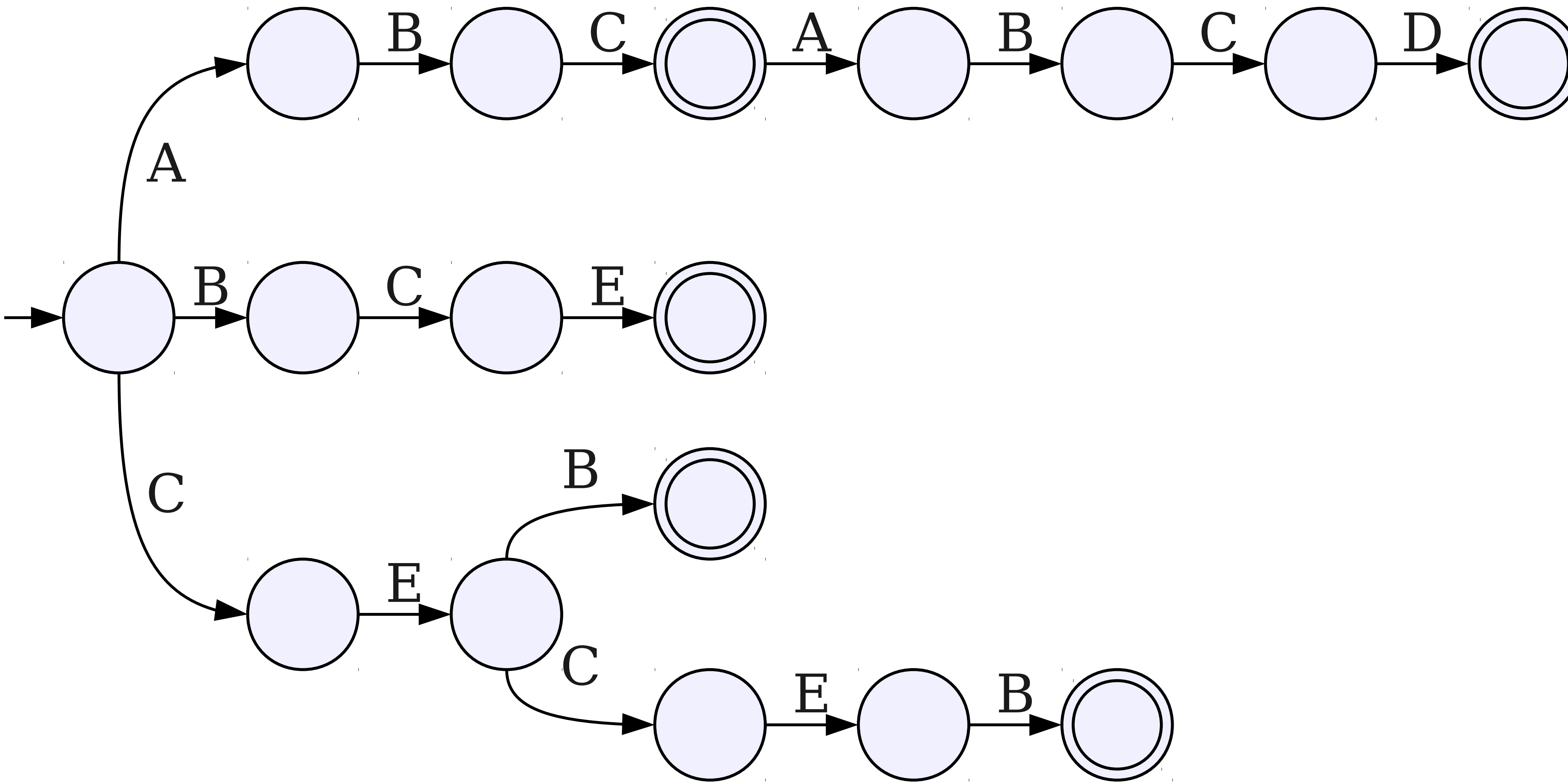
<u>A B C</u> <u>A B C</u> E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



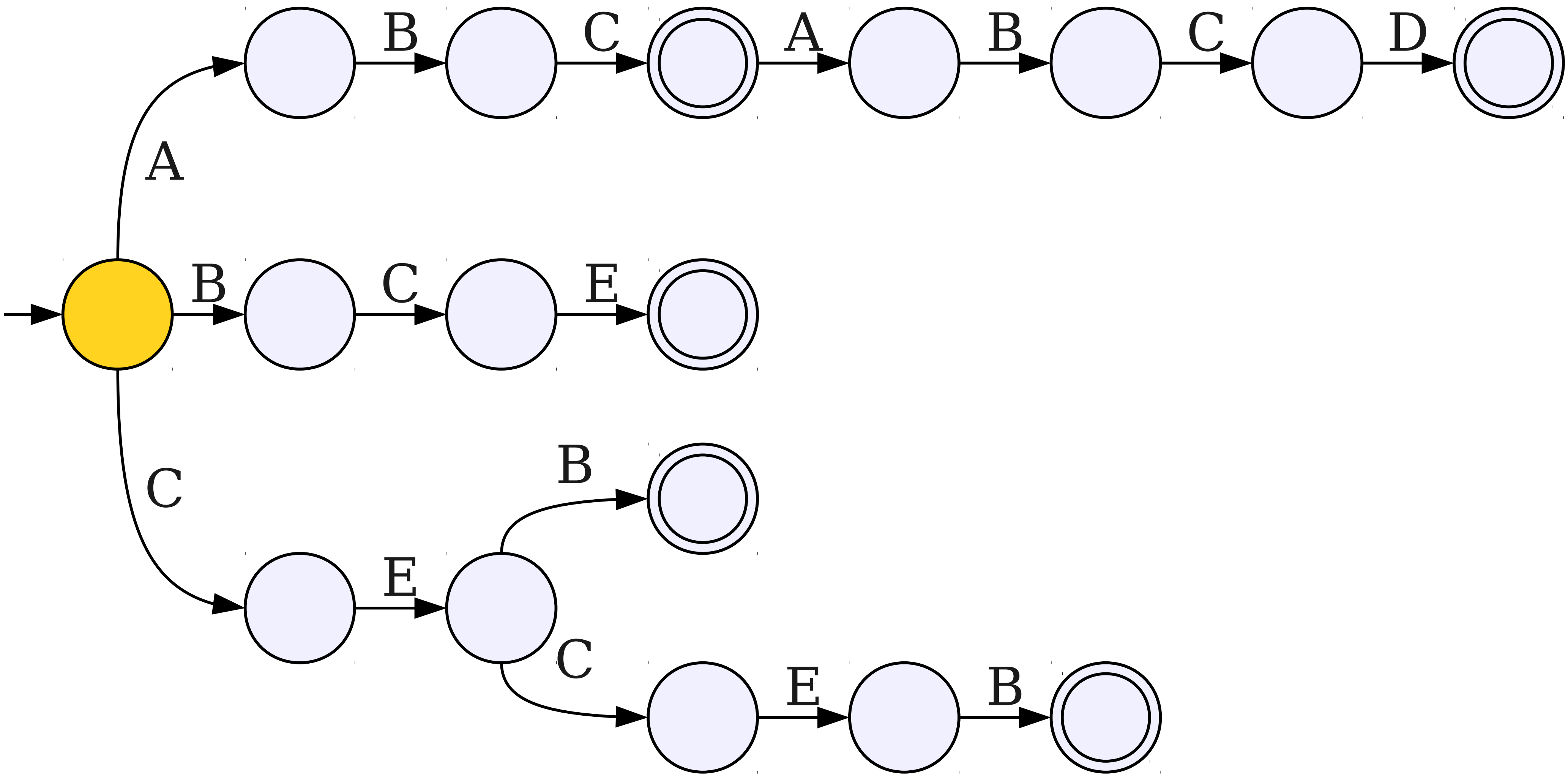
A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



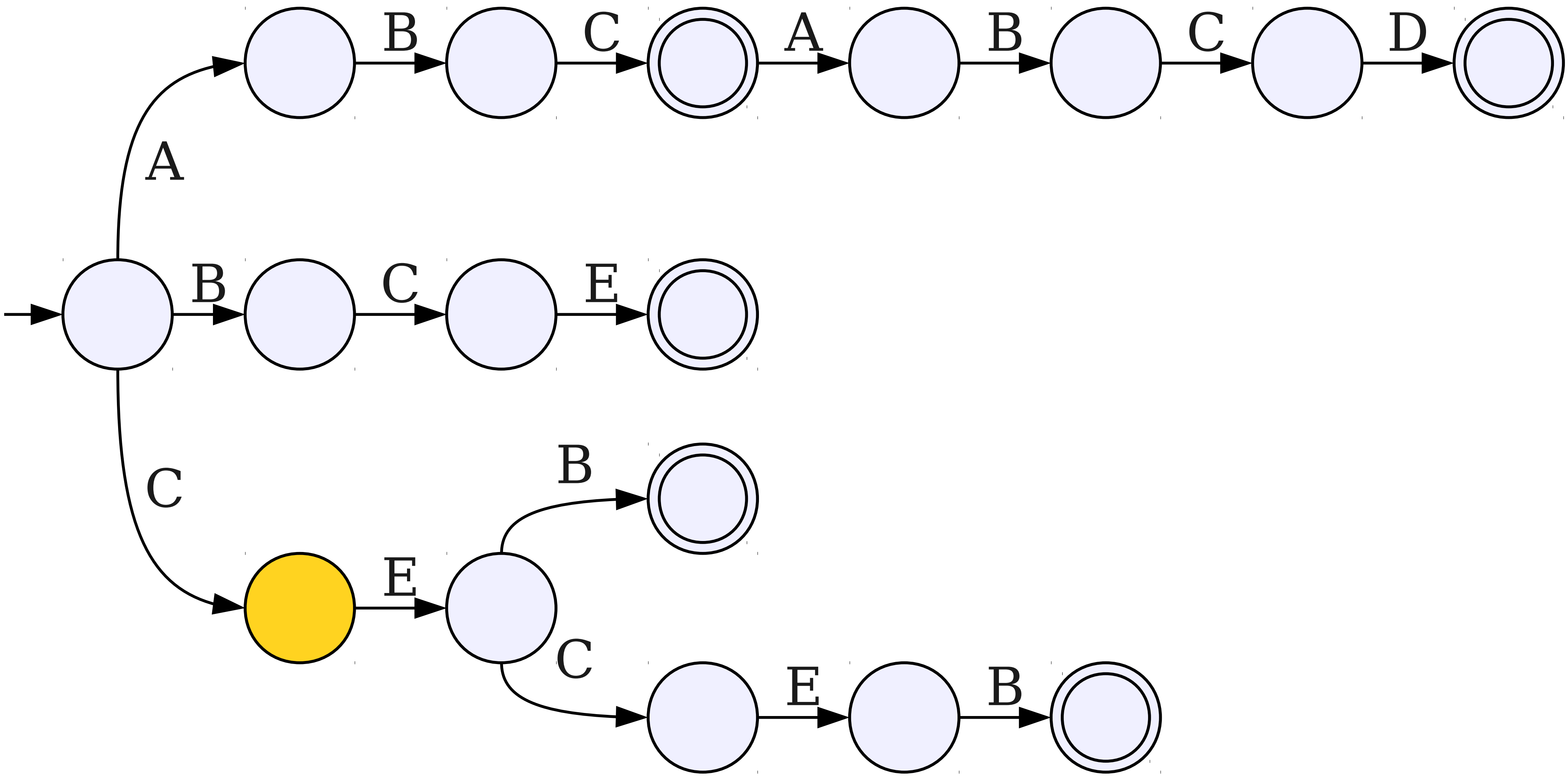
A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



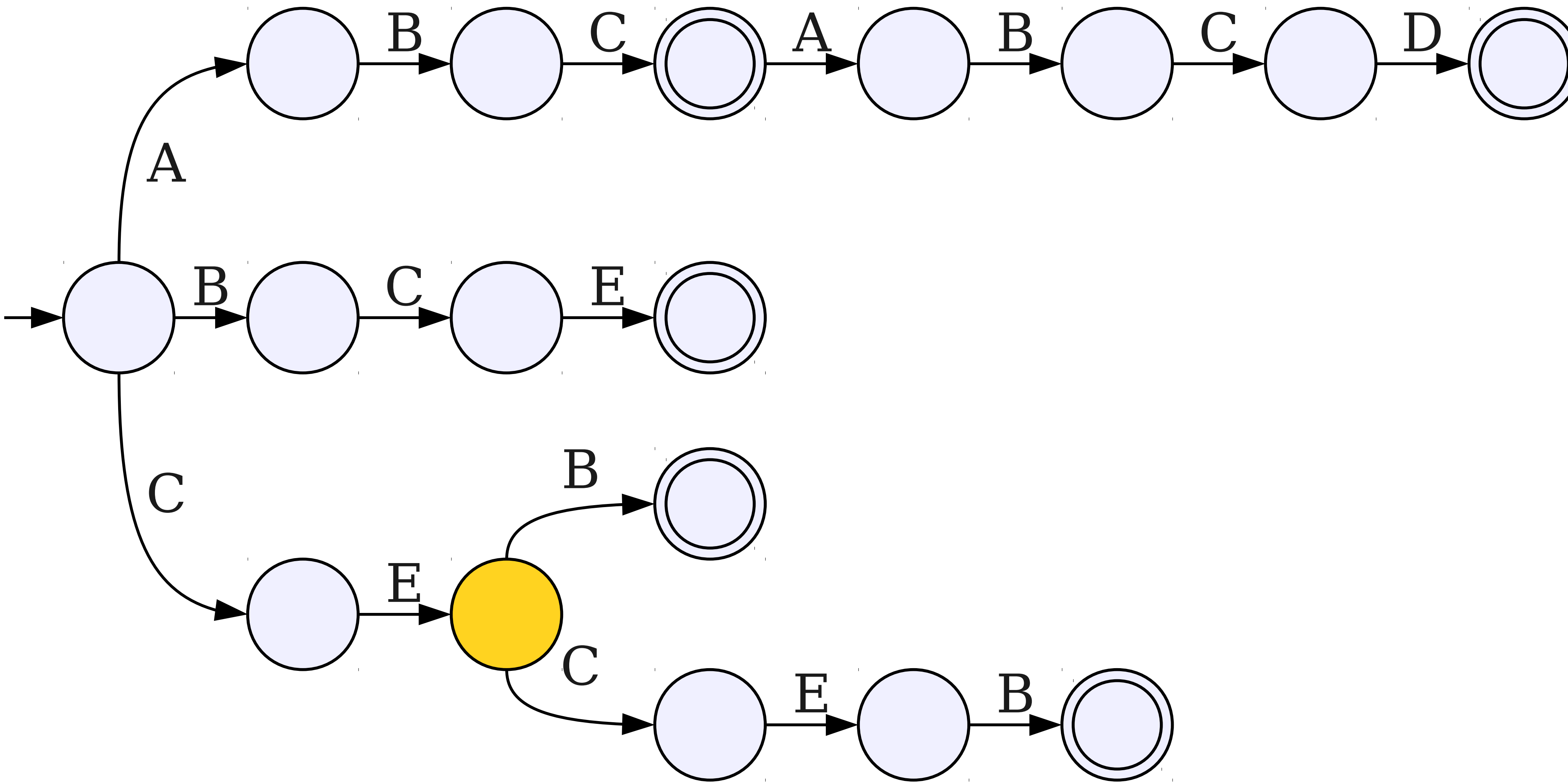
A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



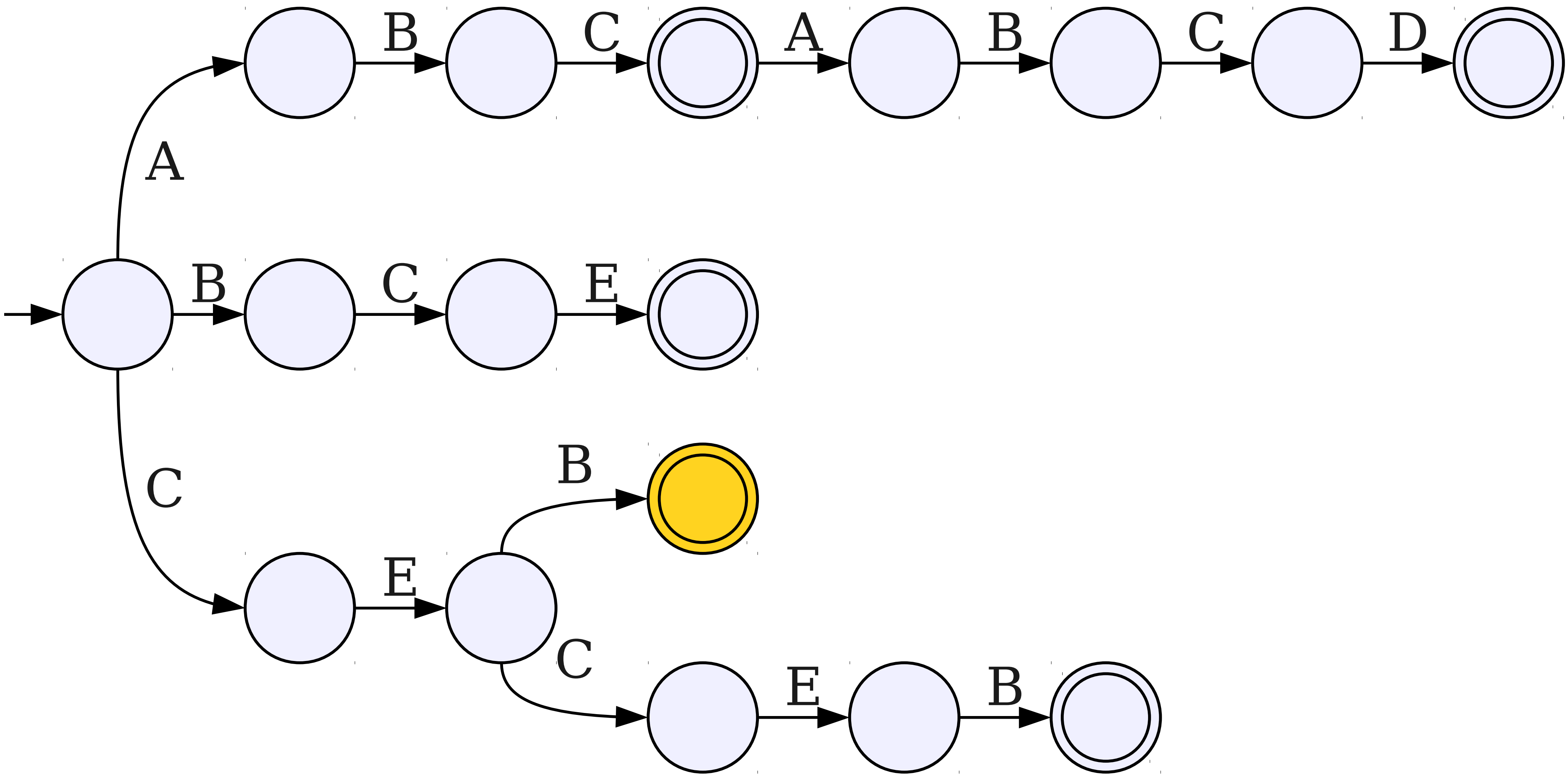
A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



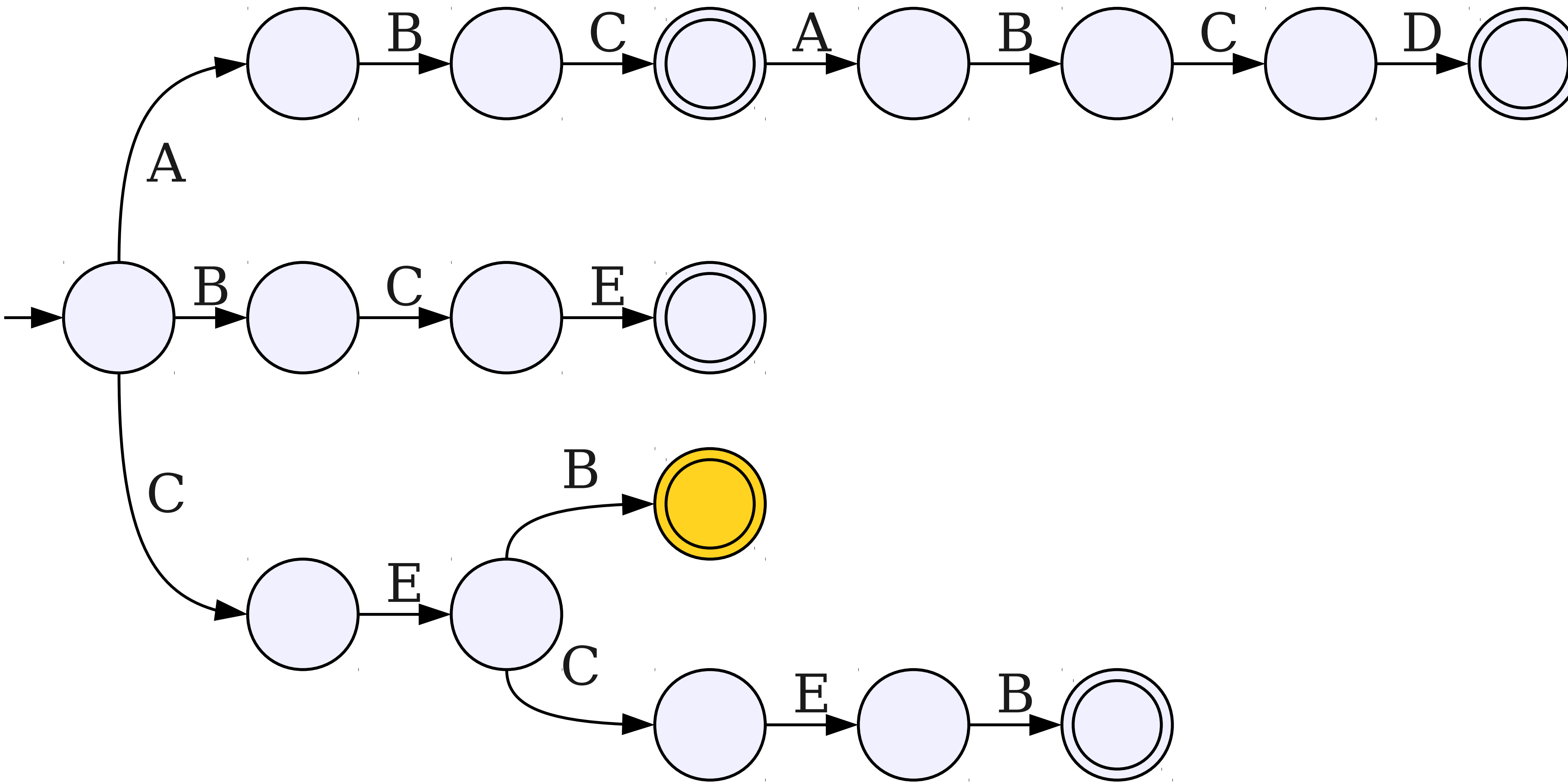
A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



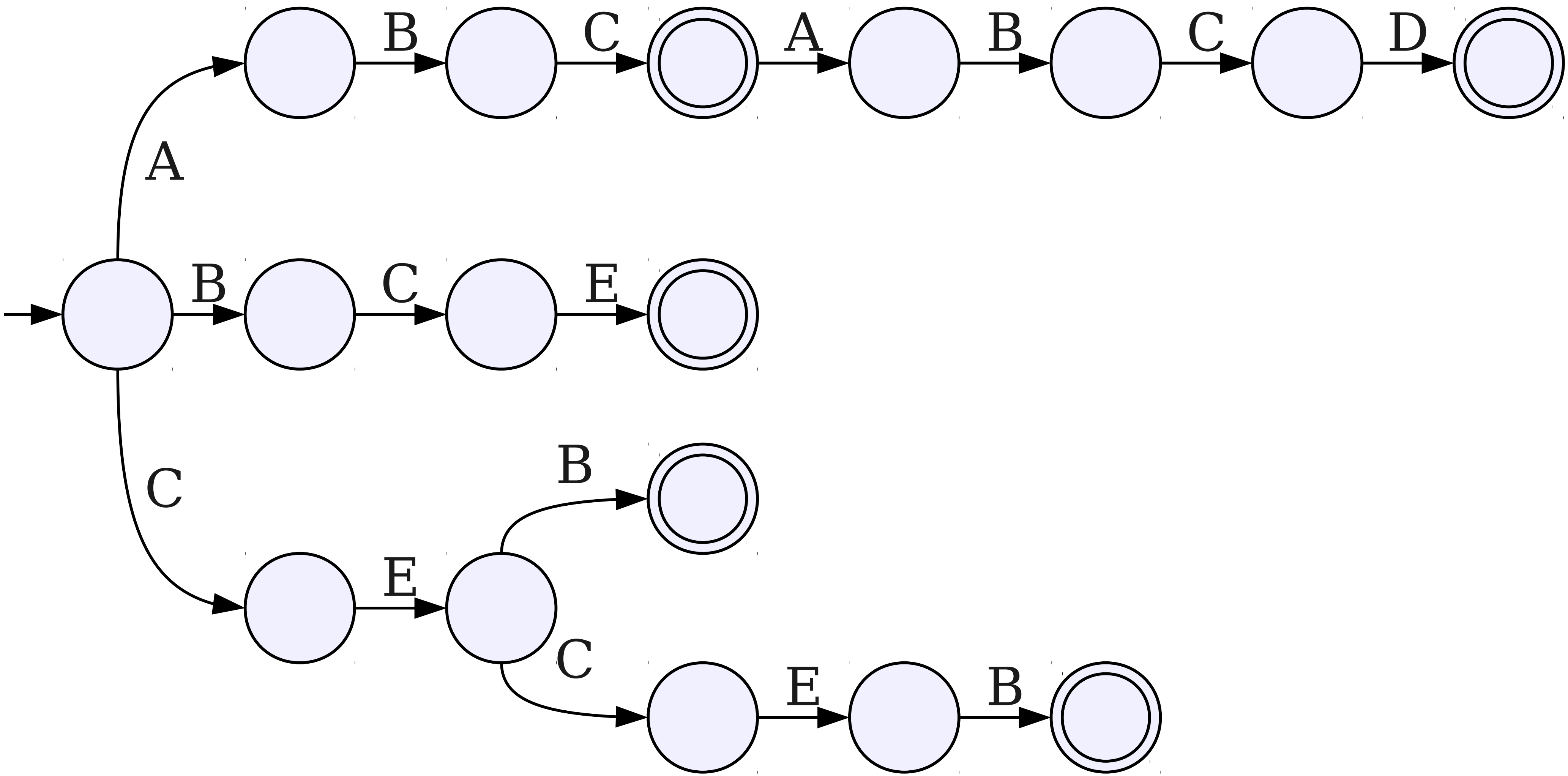
A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



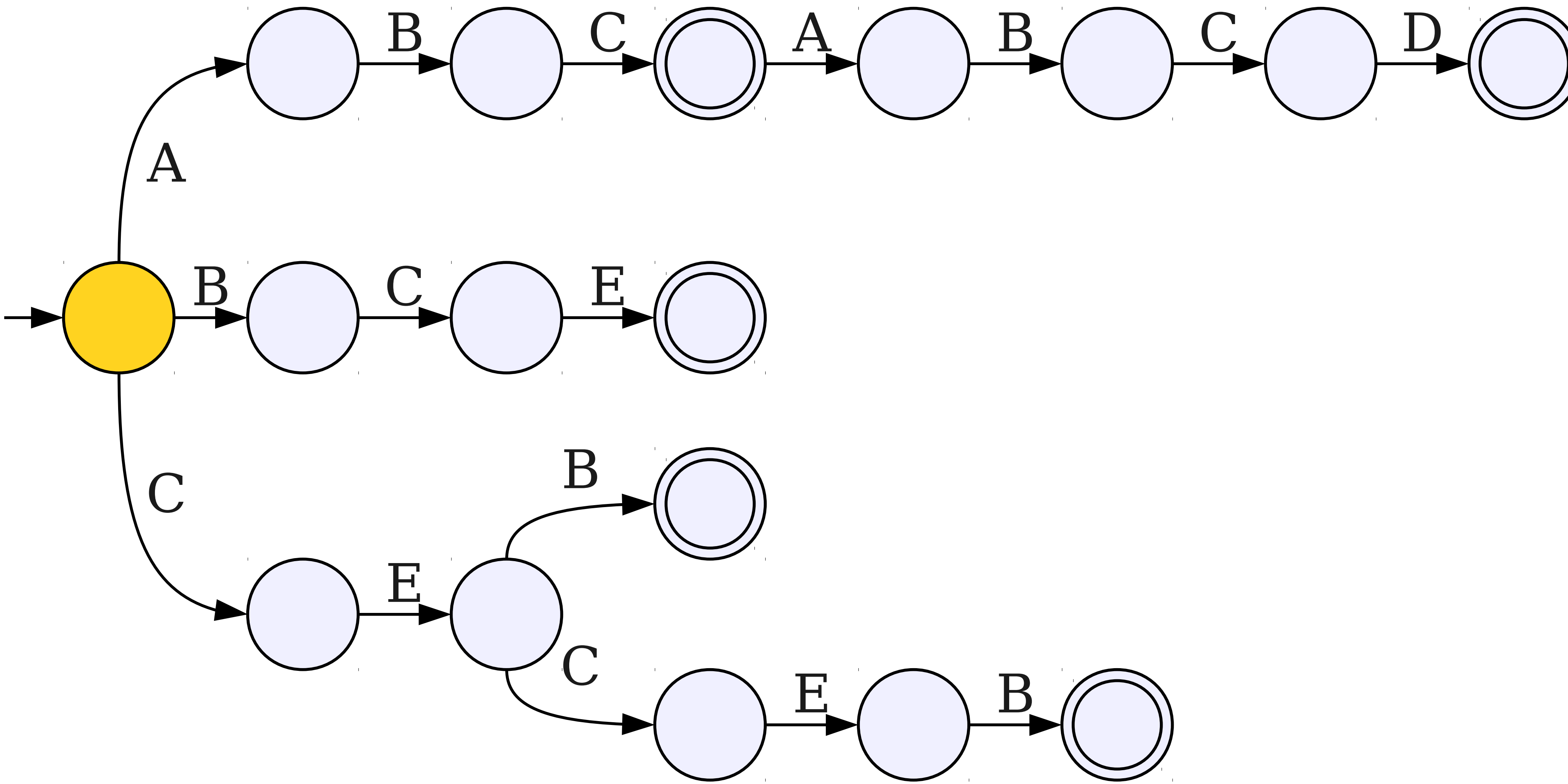
A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



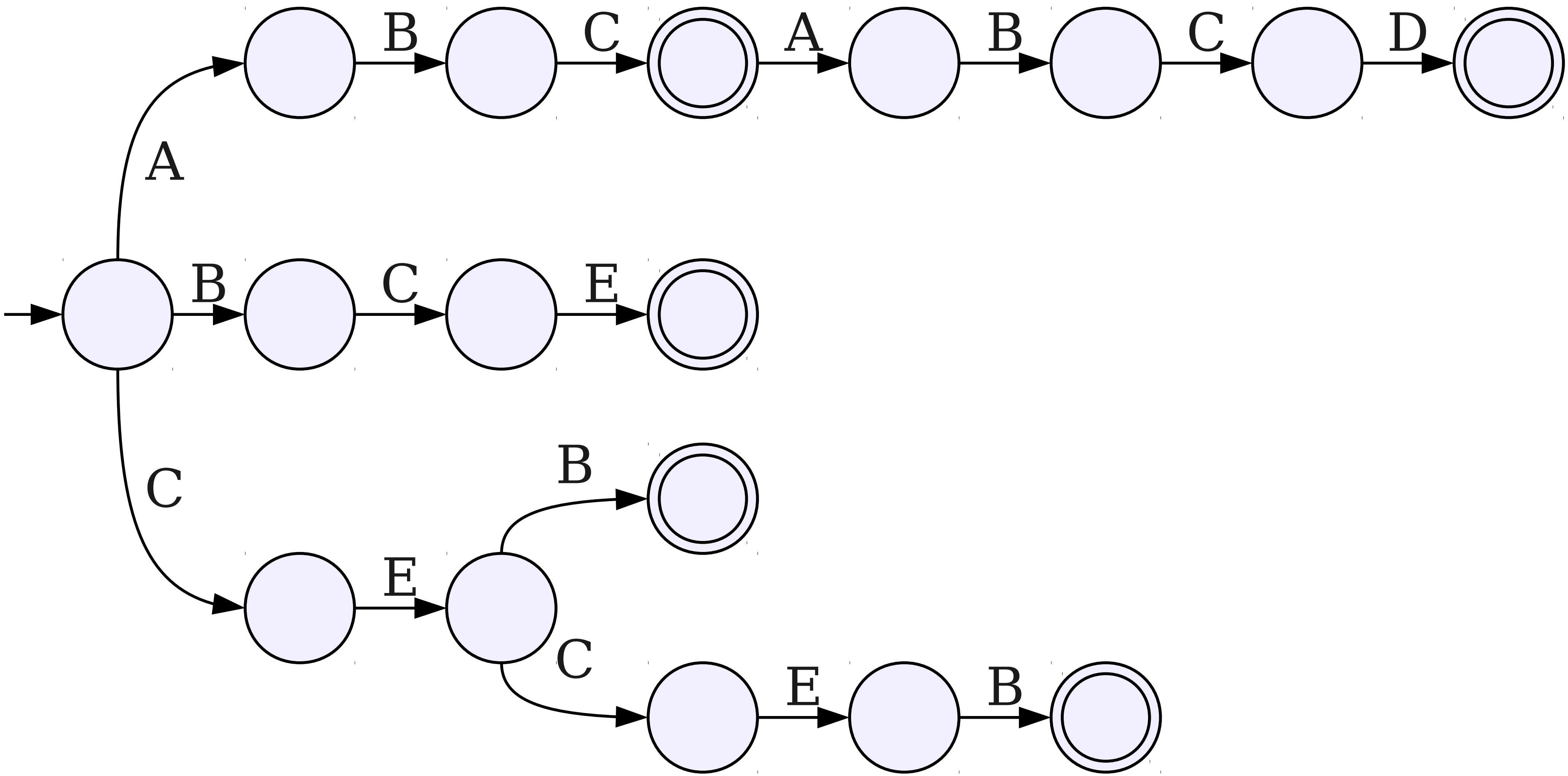
A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



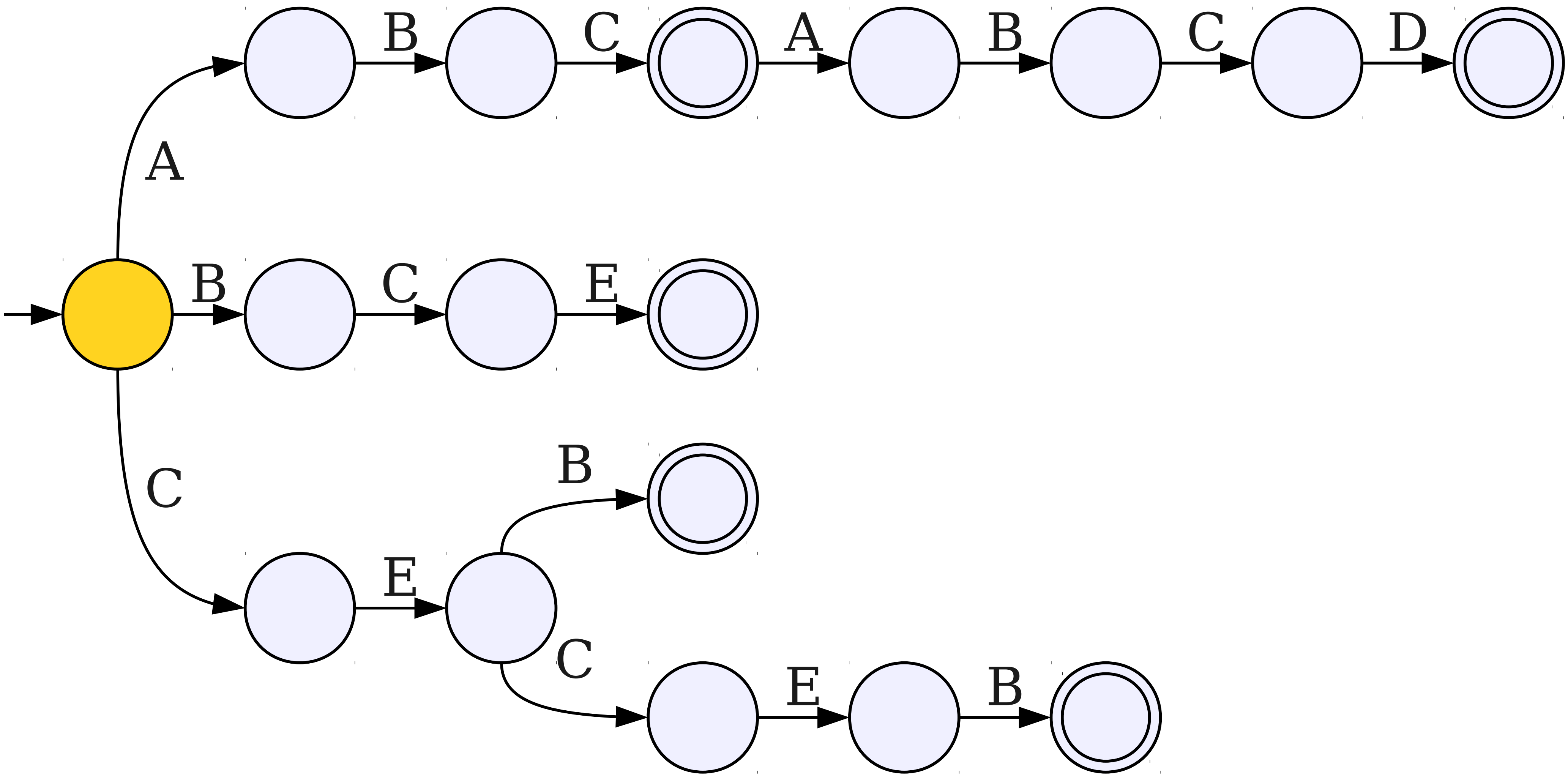
A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



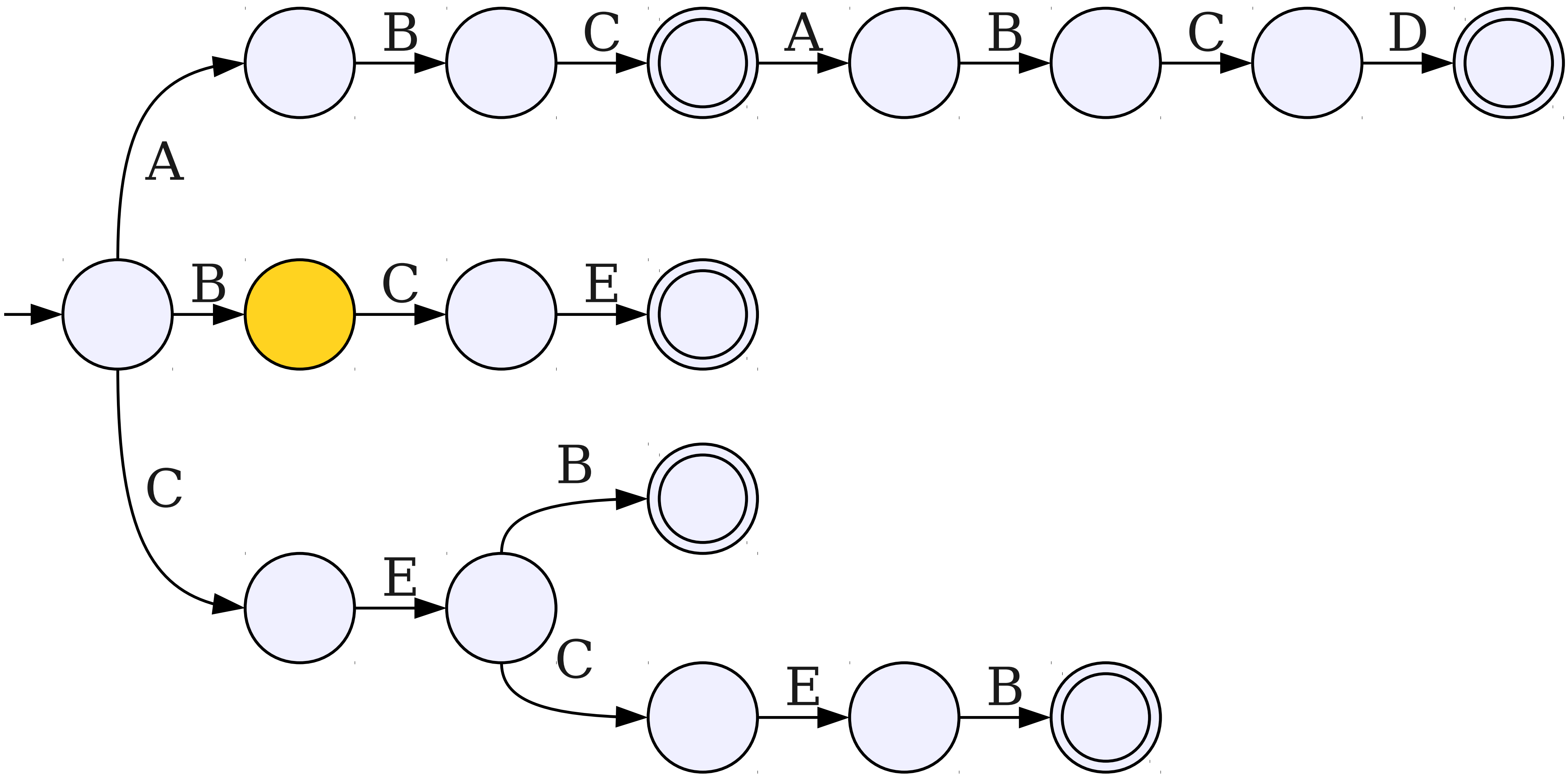
A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



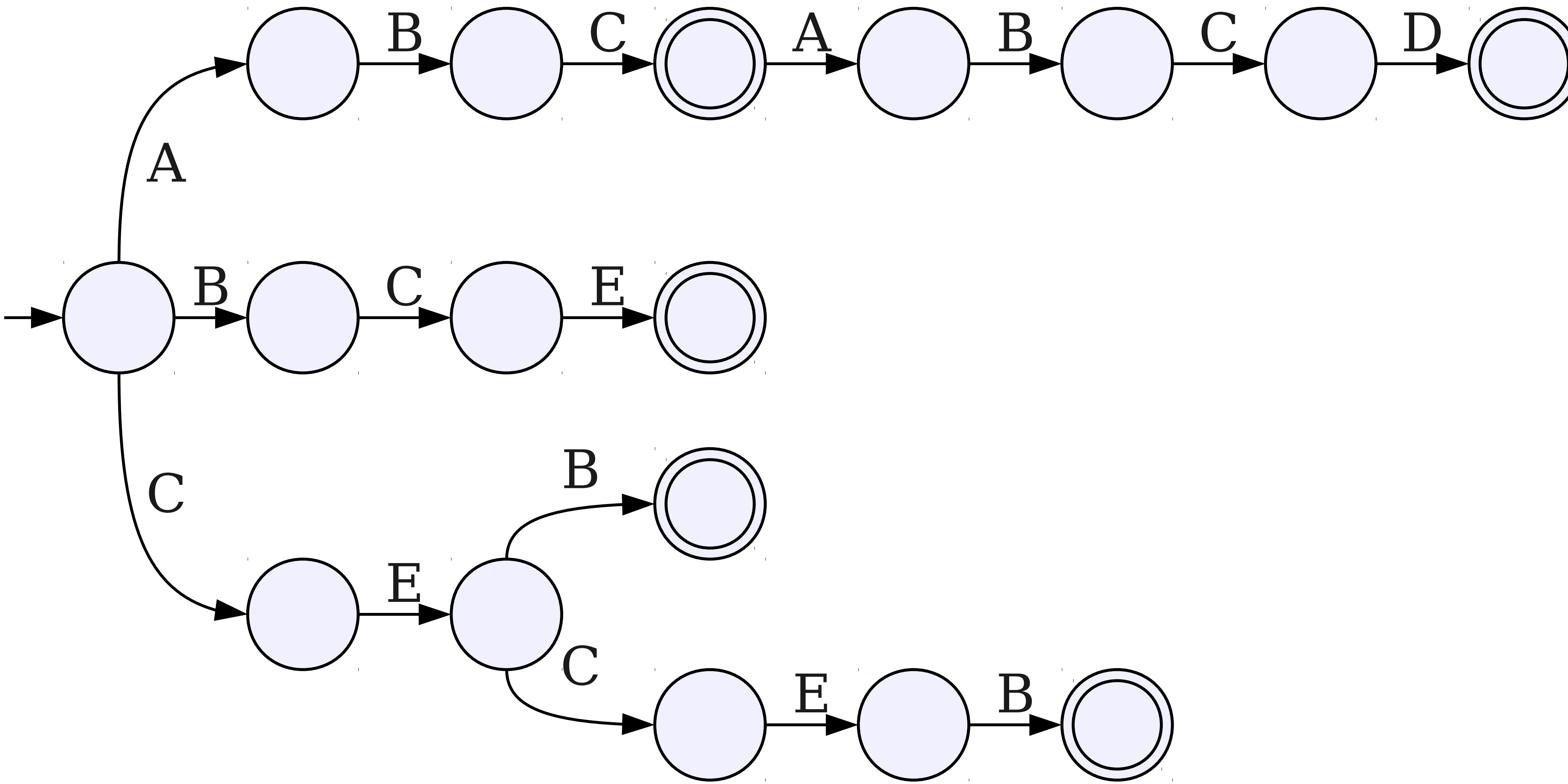
A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



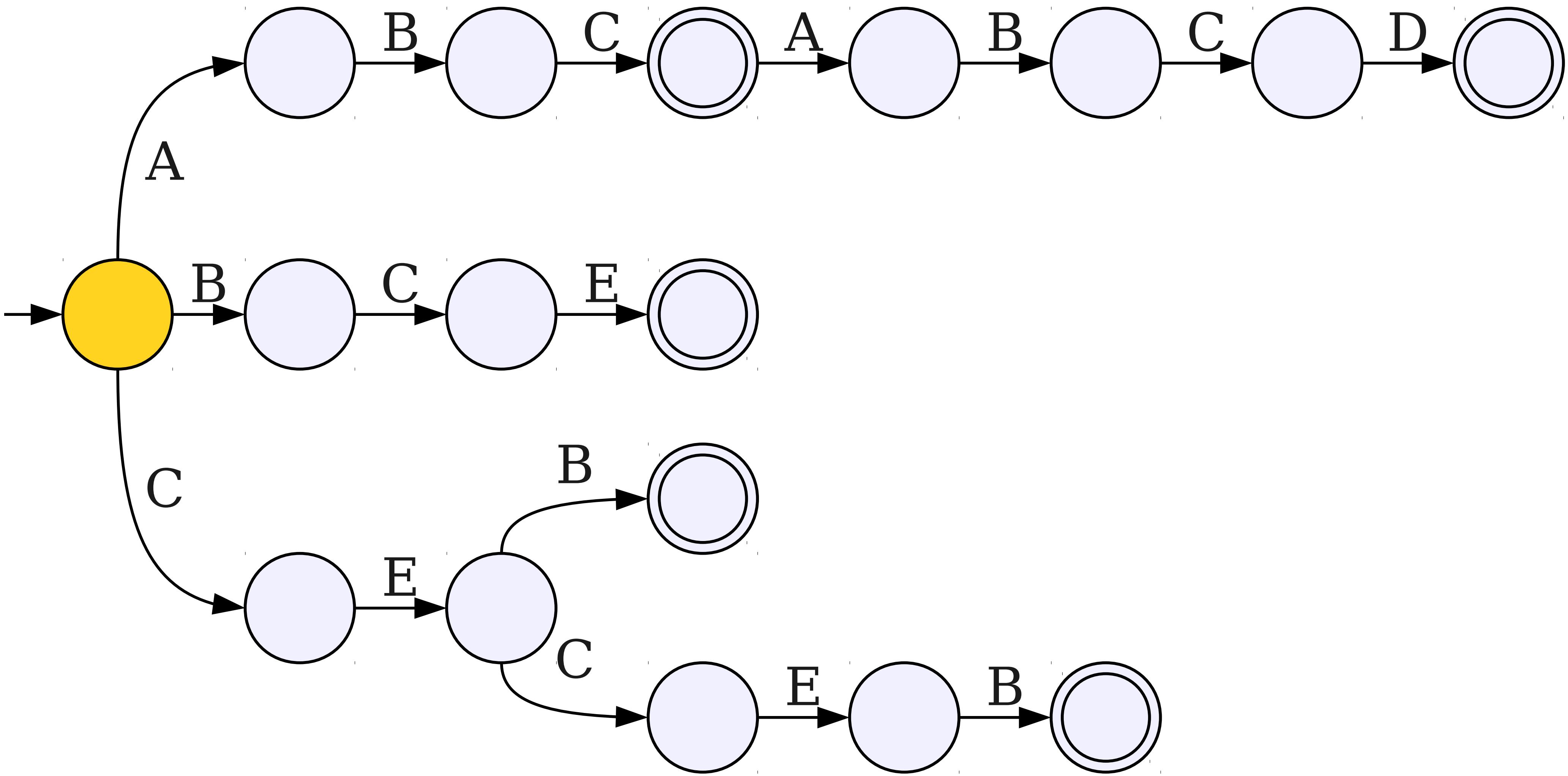
A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



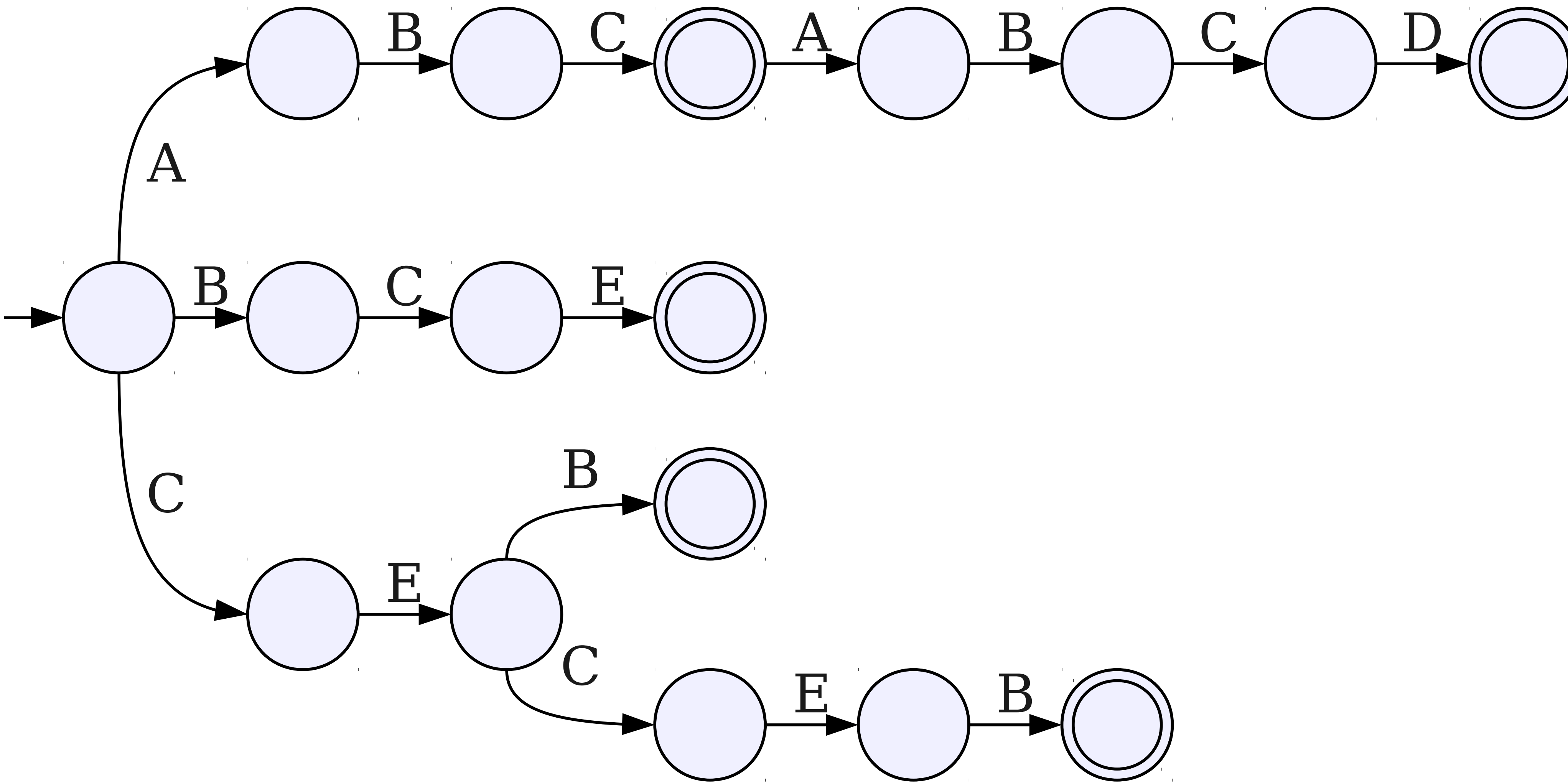
A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



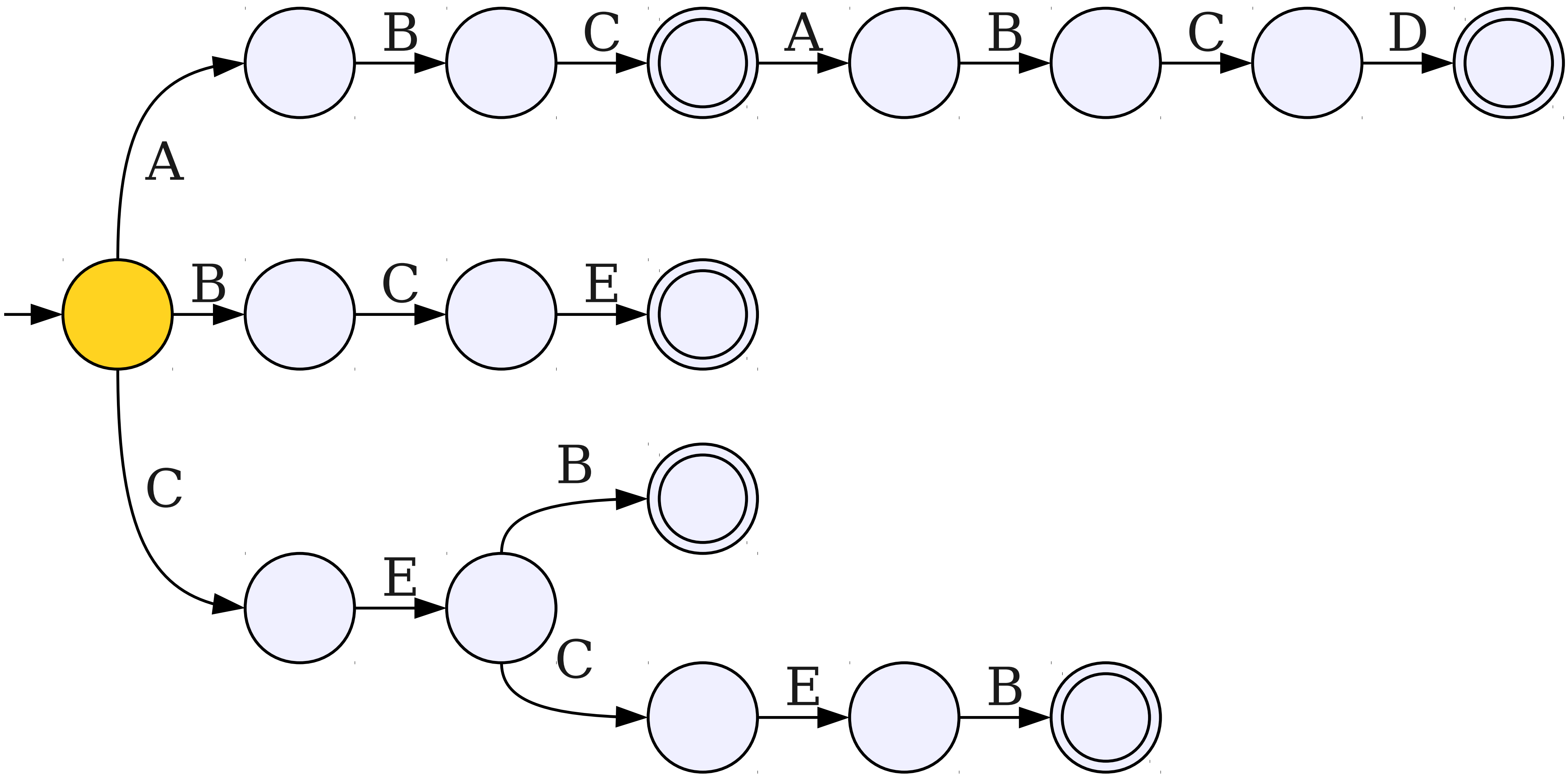
A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



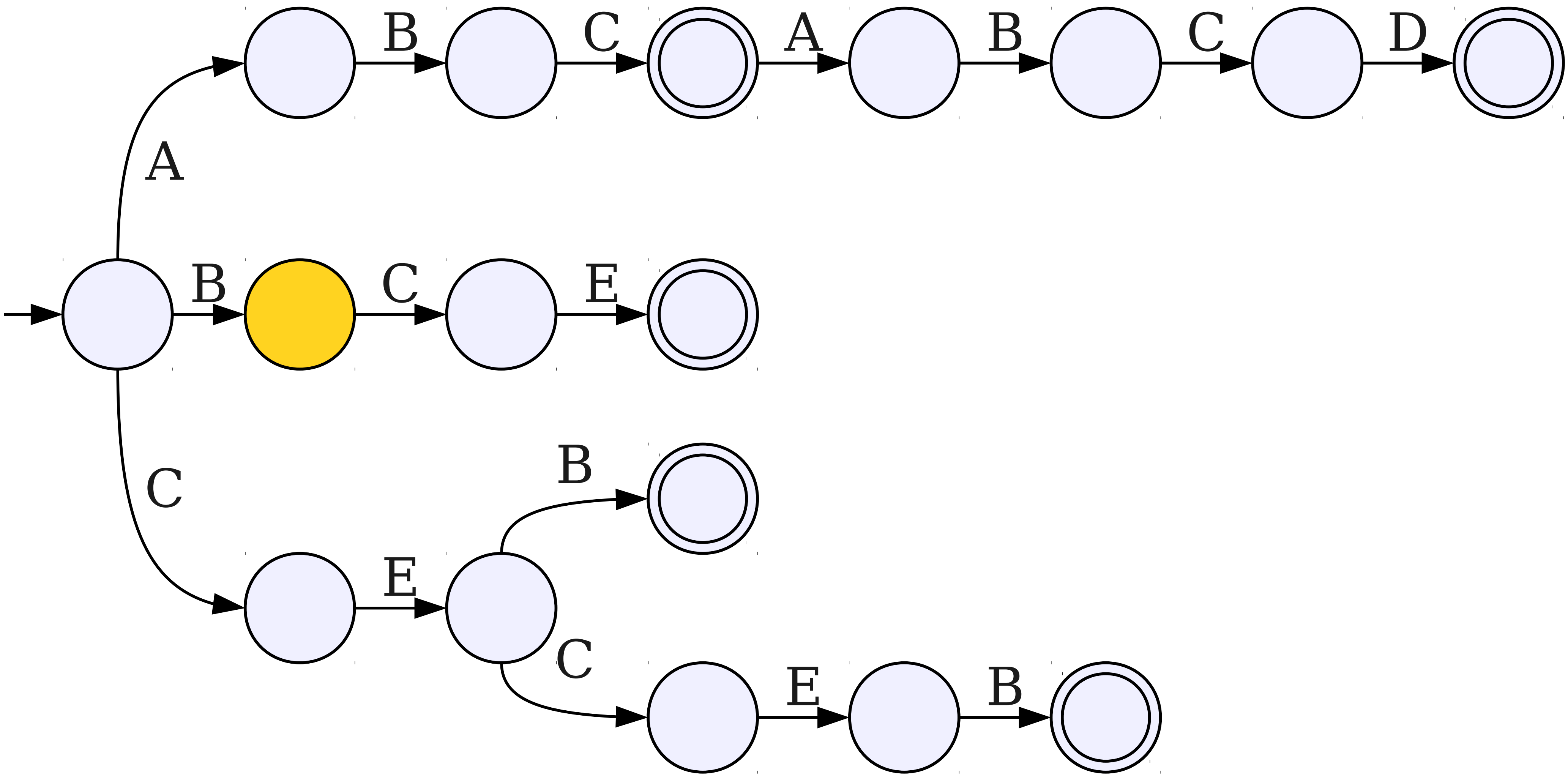
A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



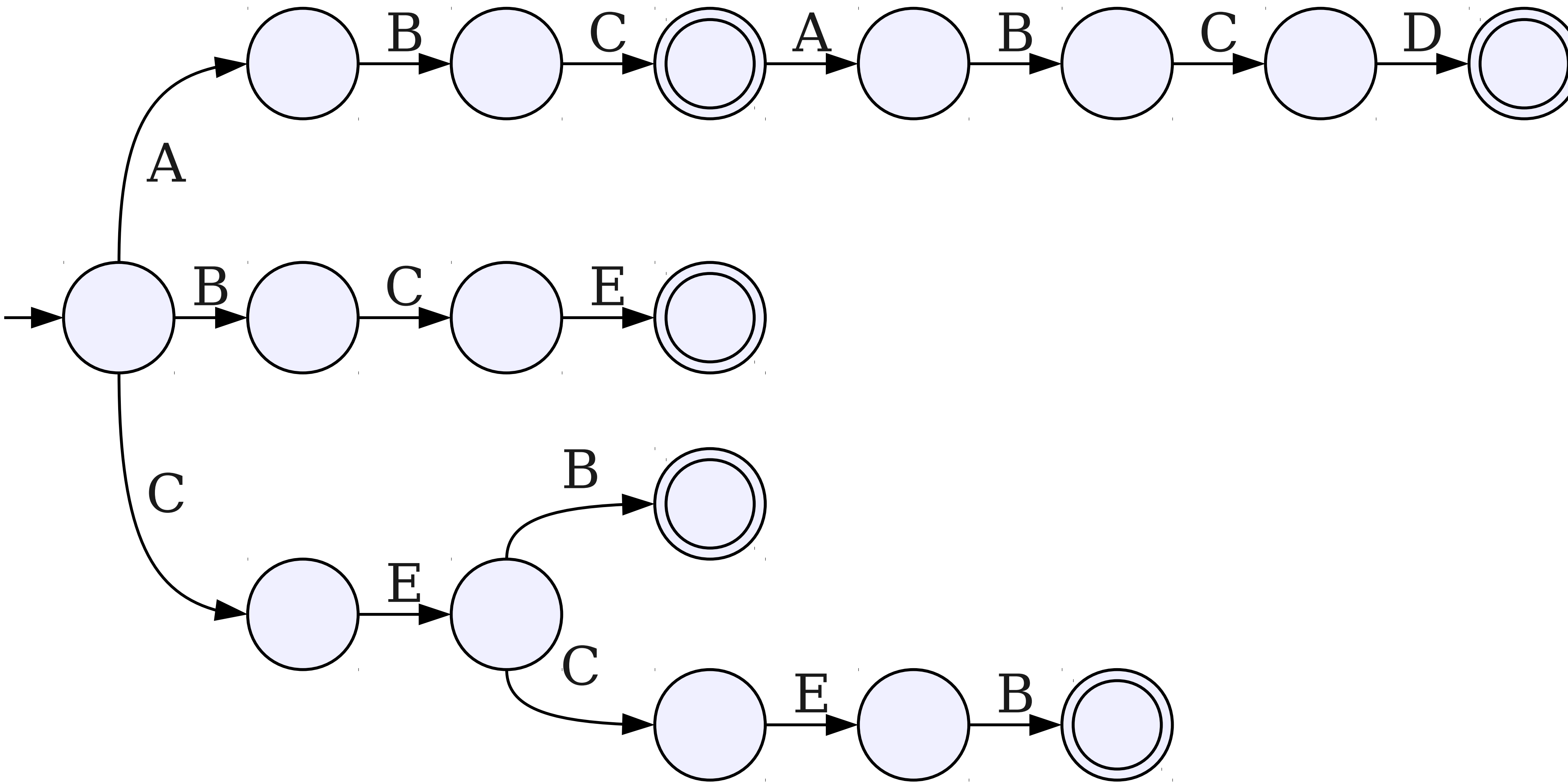
A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC

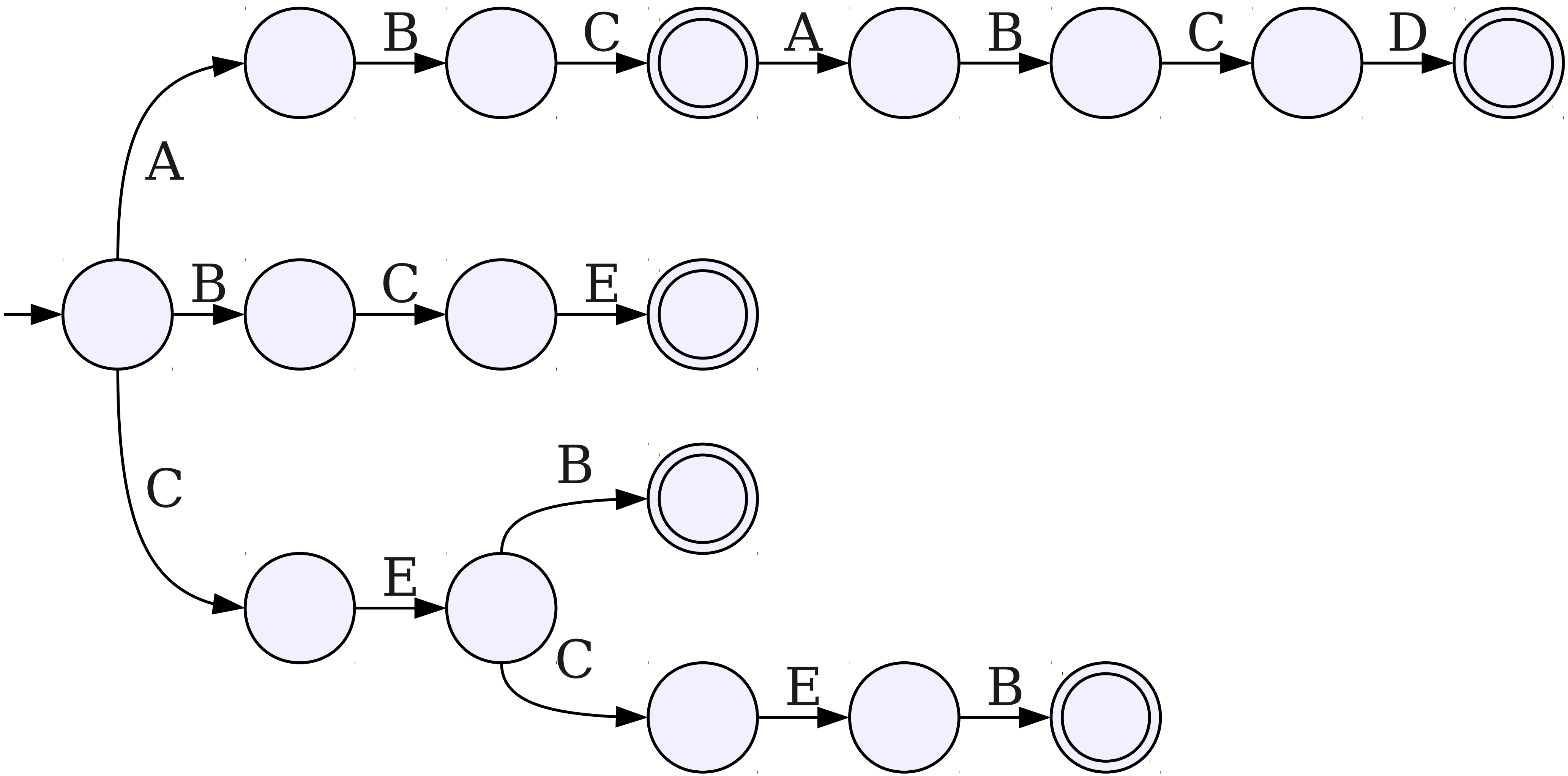


A B C A B C E B E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC

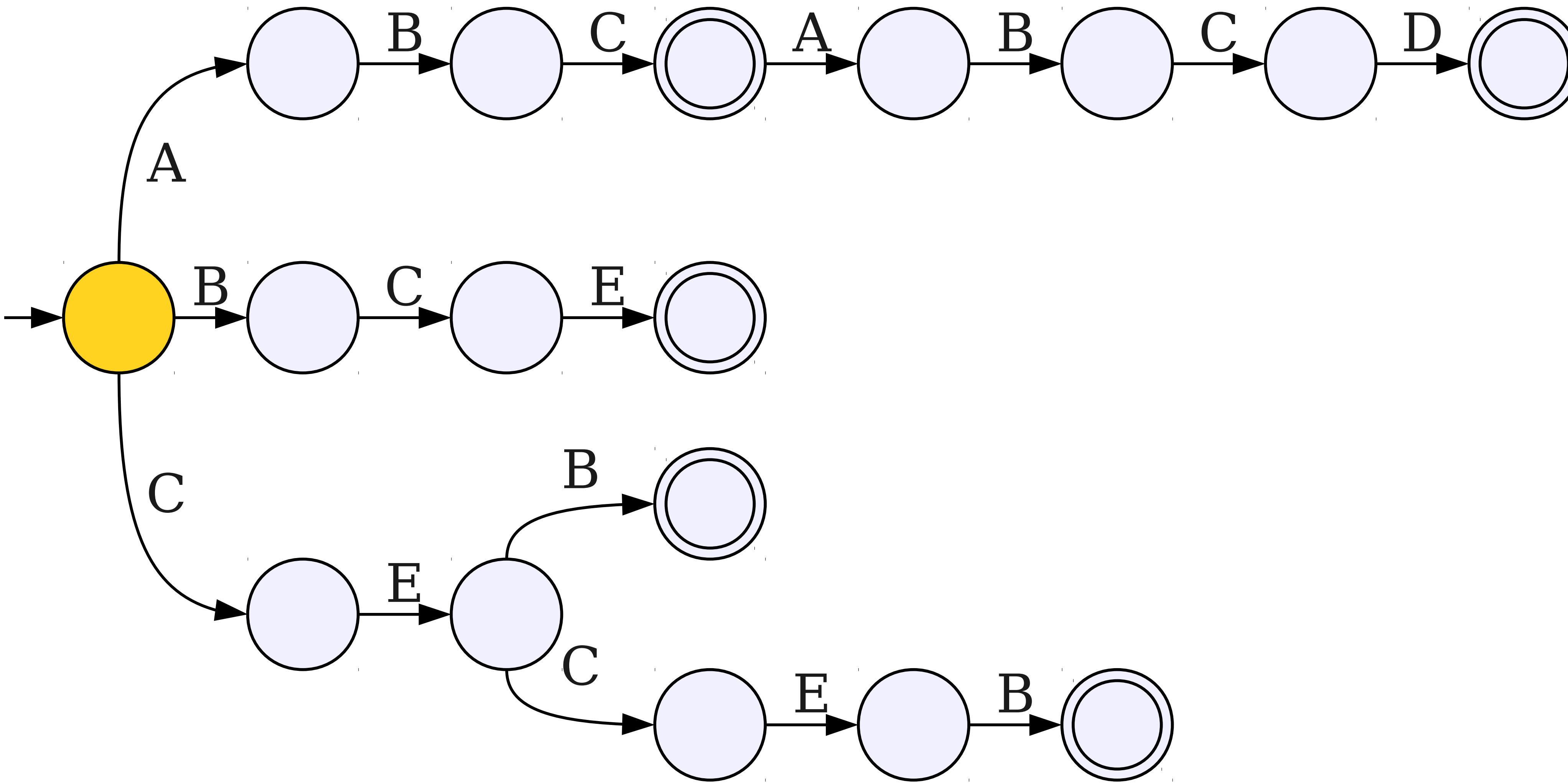
Why So Slow?

- This algorithm is slow because we repeatedly descend into the trie starting at the root.
- This means that each character of T is processed multiple times.
- **Question:** Can we avoid restarting our search at the tree root, which will avoid revisiting characters in T?



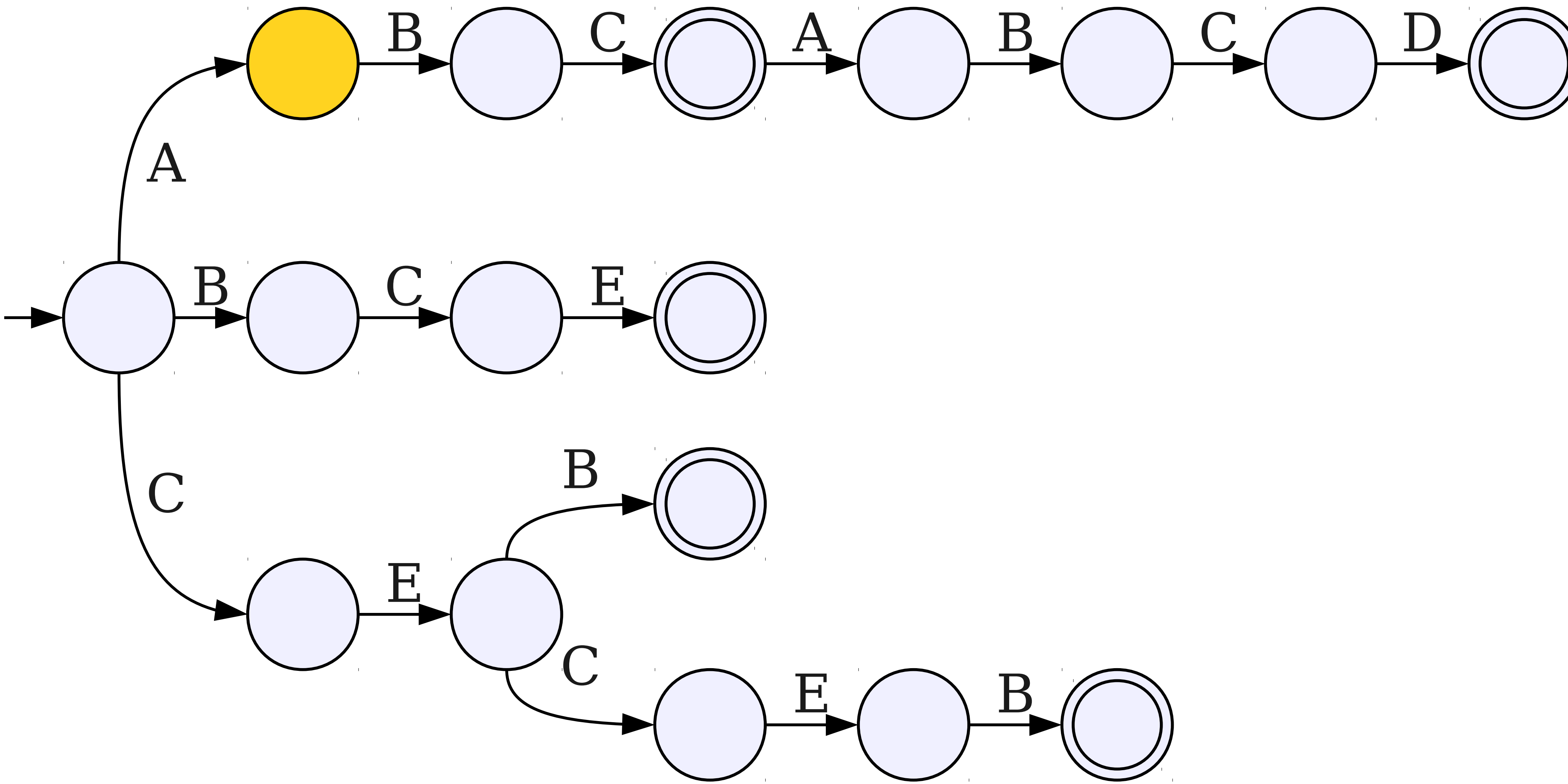
A B C E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



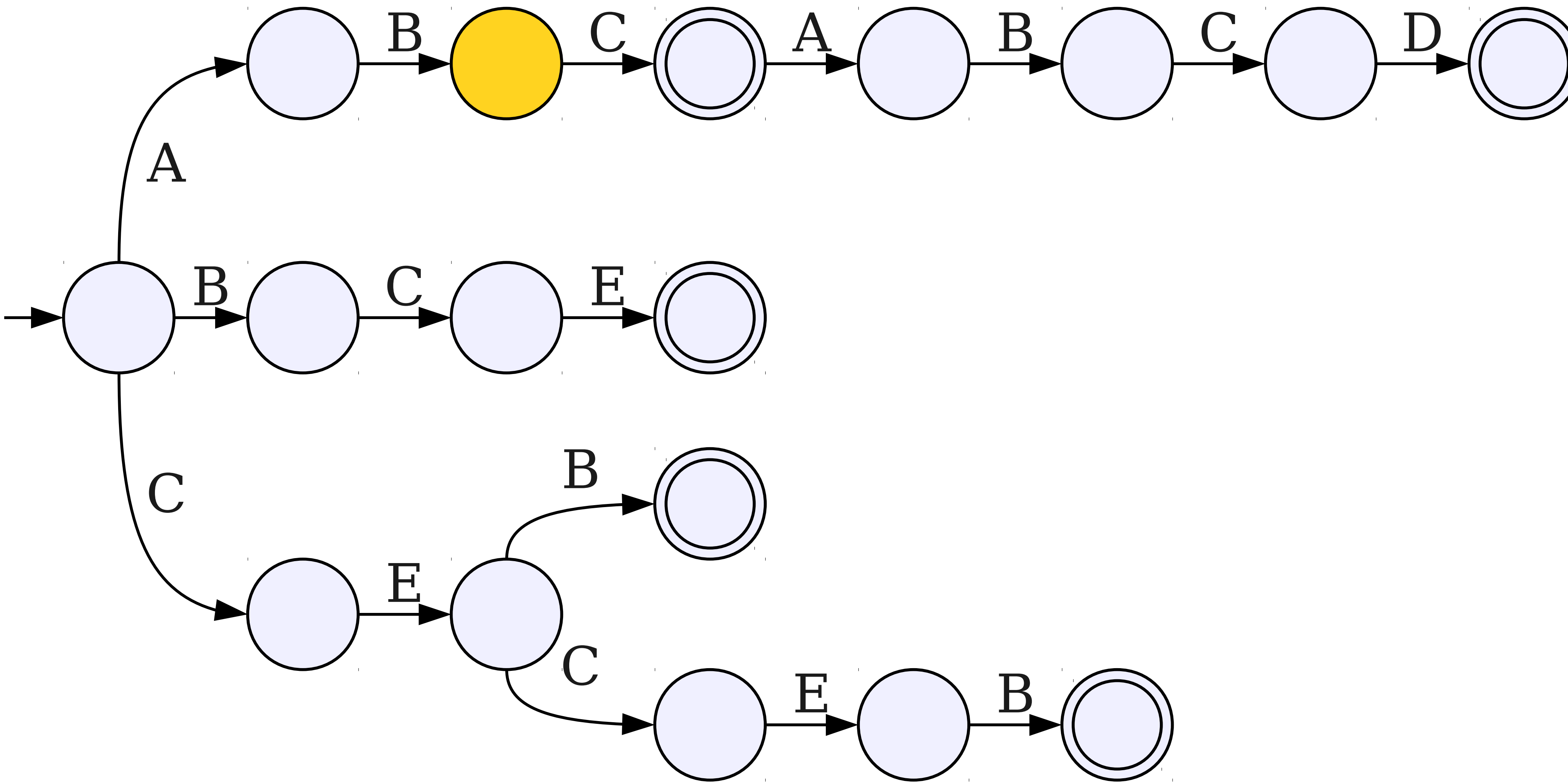
A B C E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



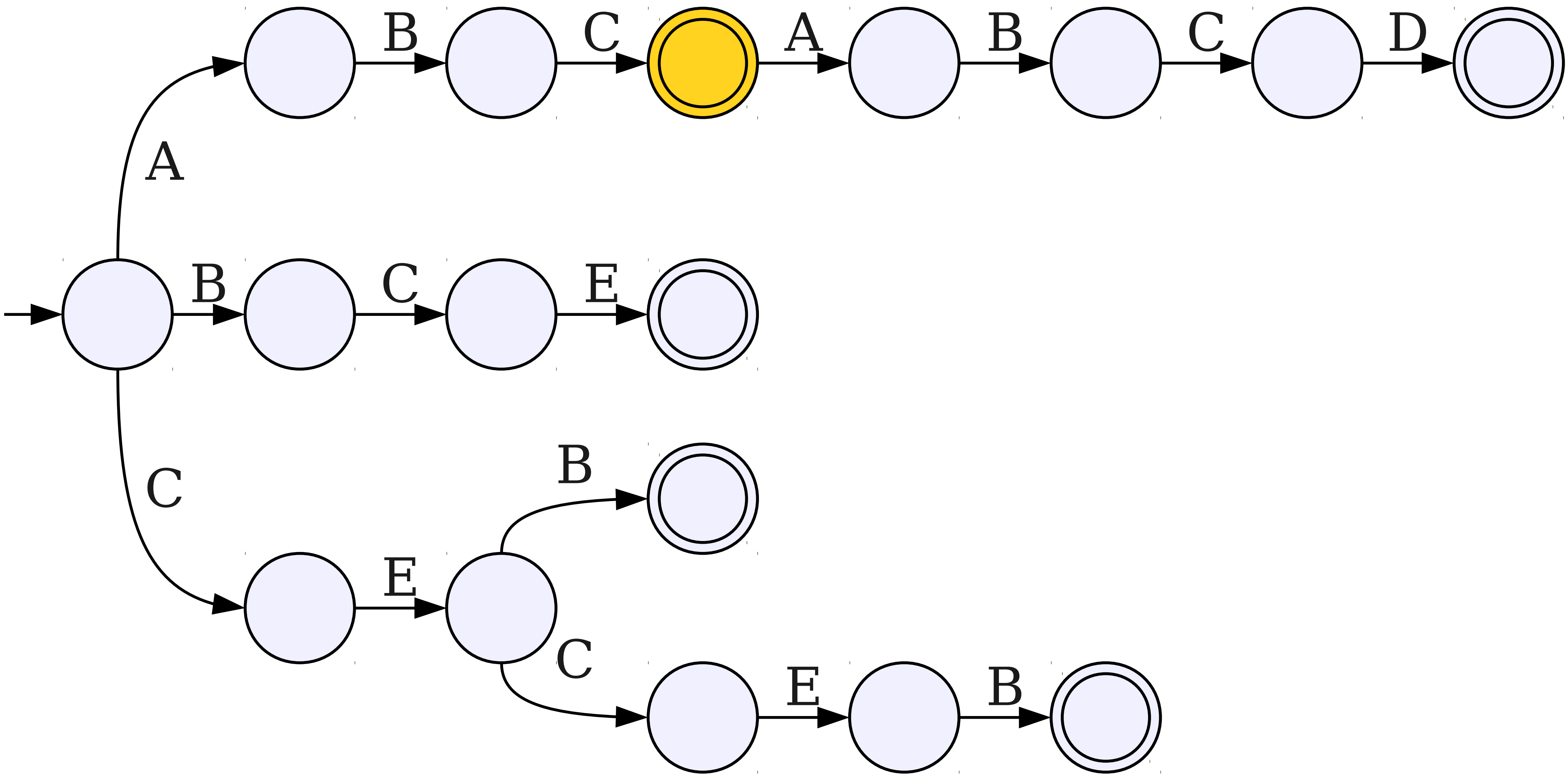
A B C E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



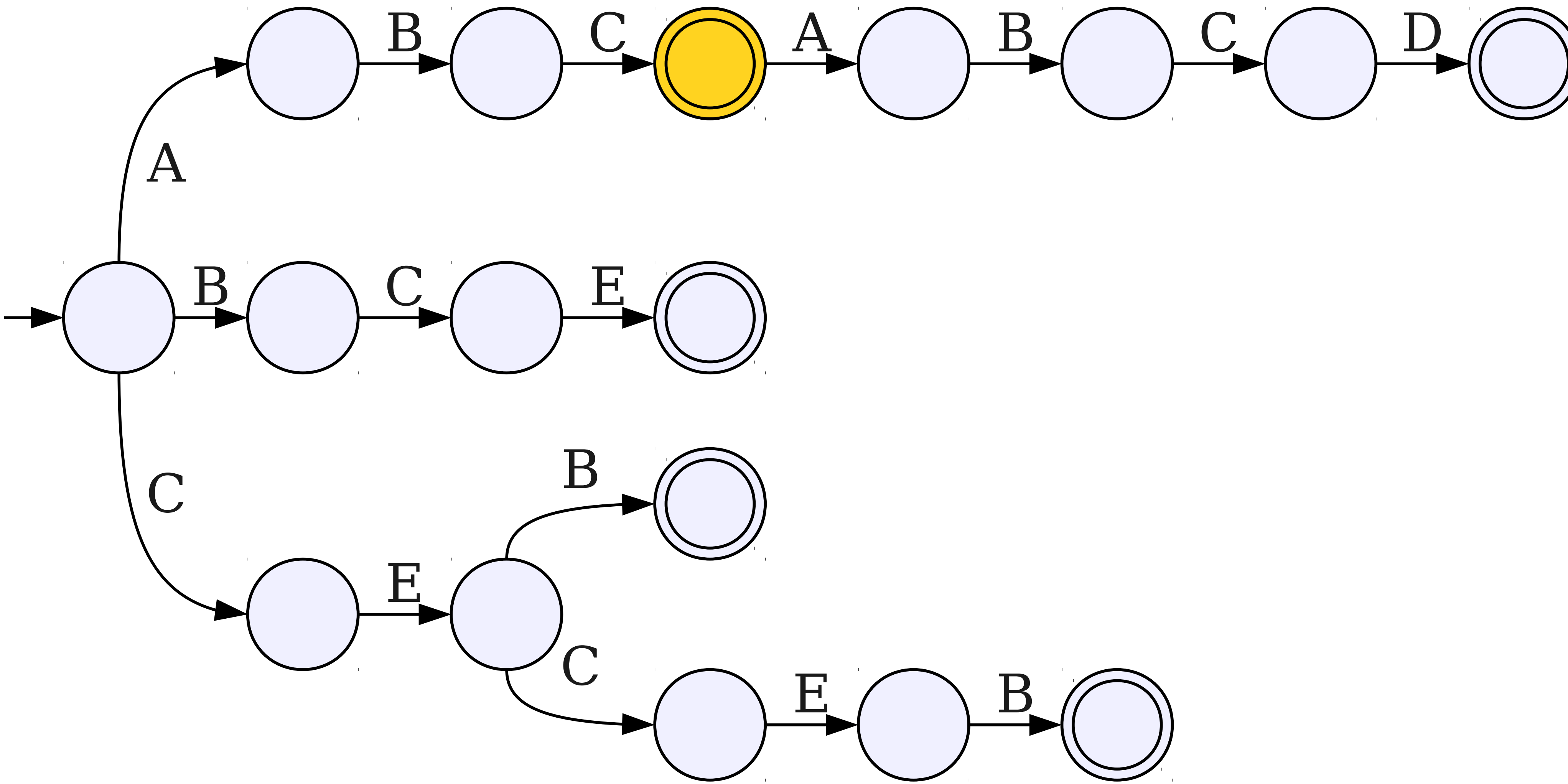
A B C E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



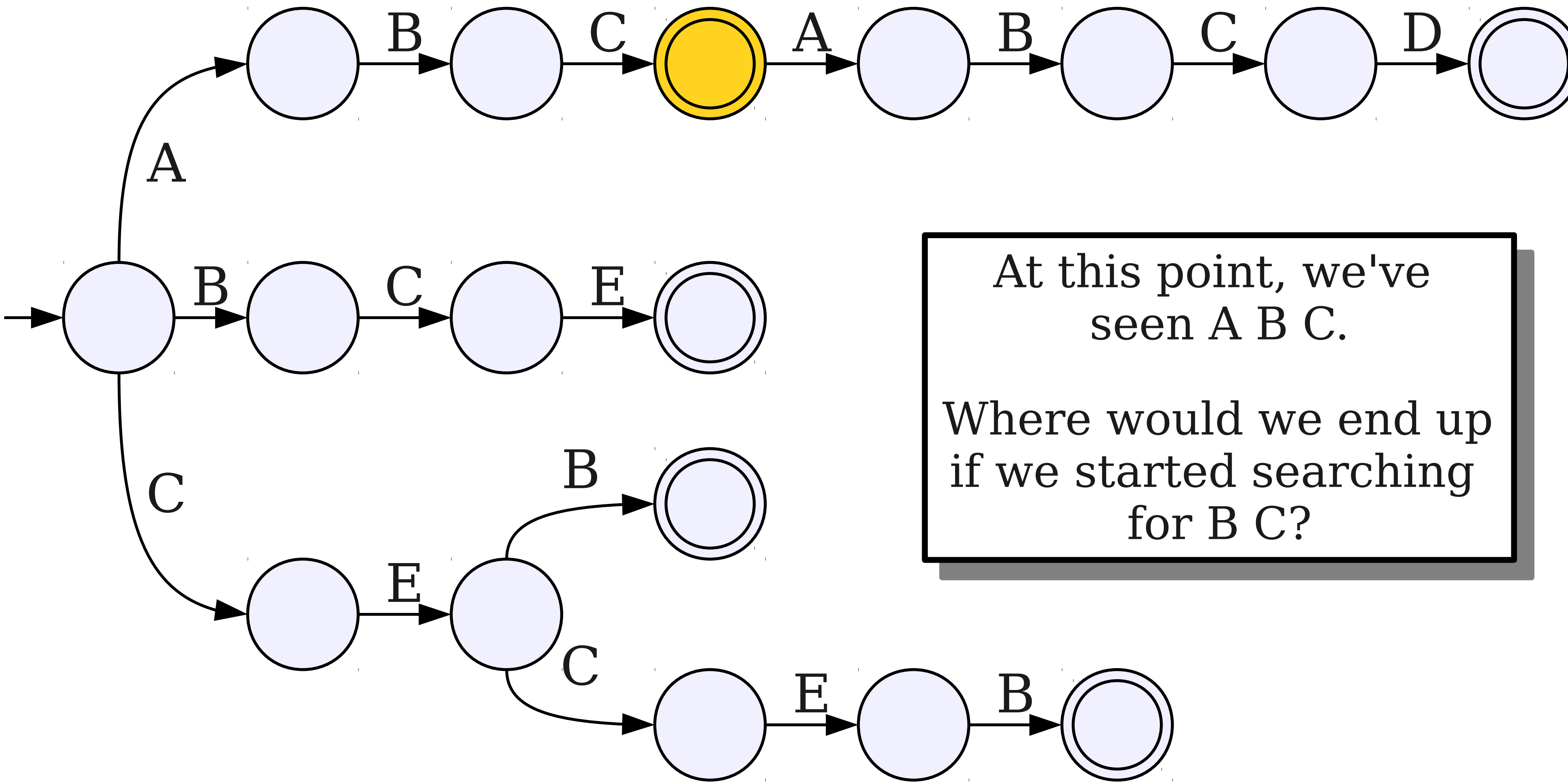
A B C E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



A B C E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC

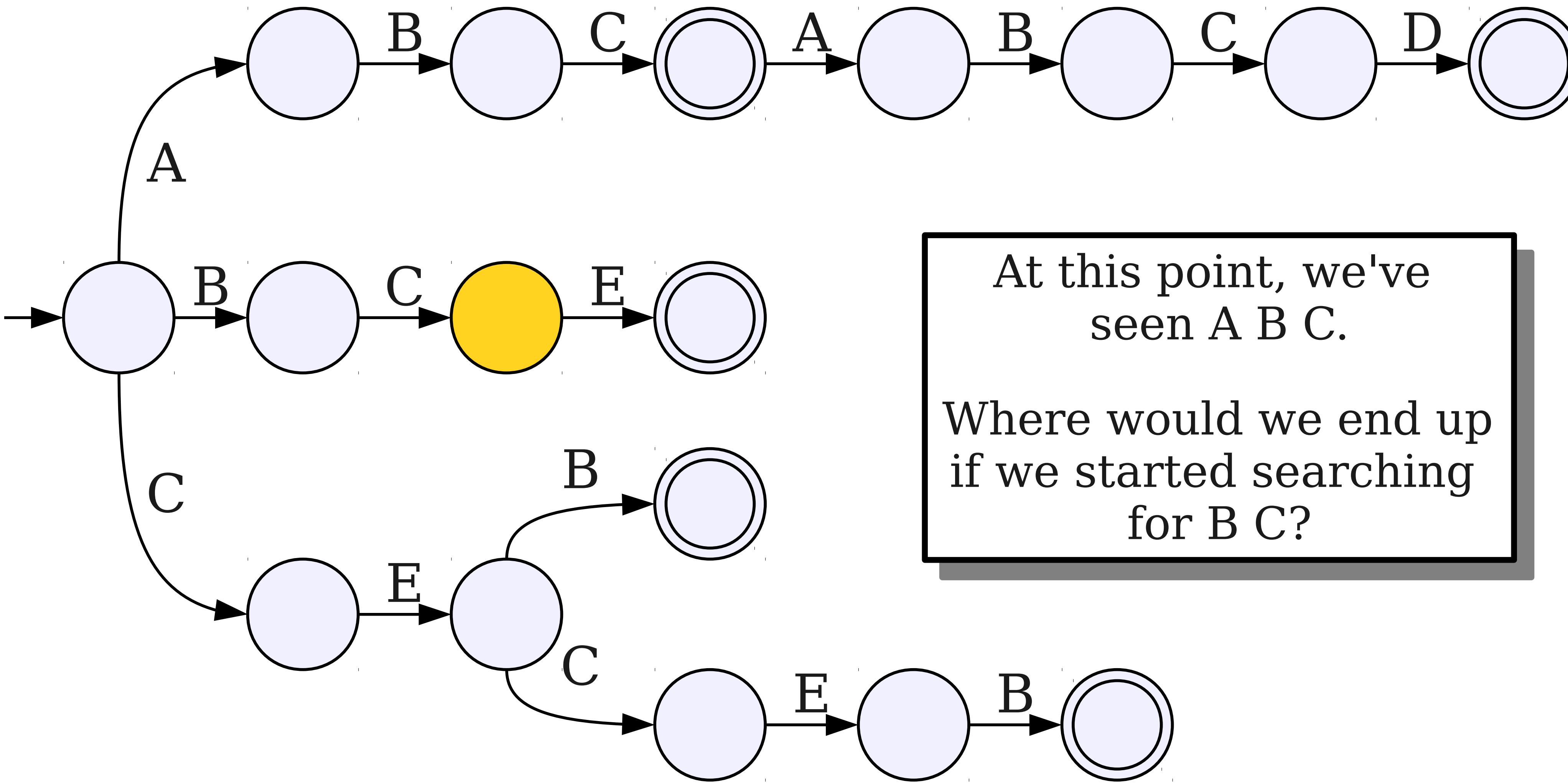


At this point, we've seen A B C.

Where would we end up if we started searching for B C?

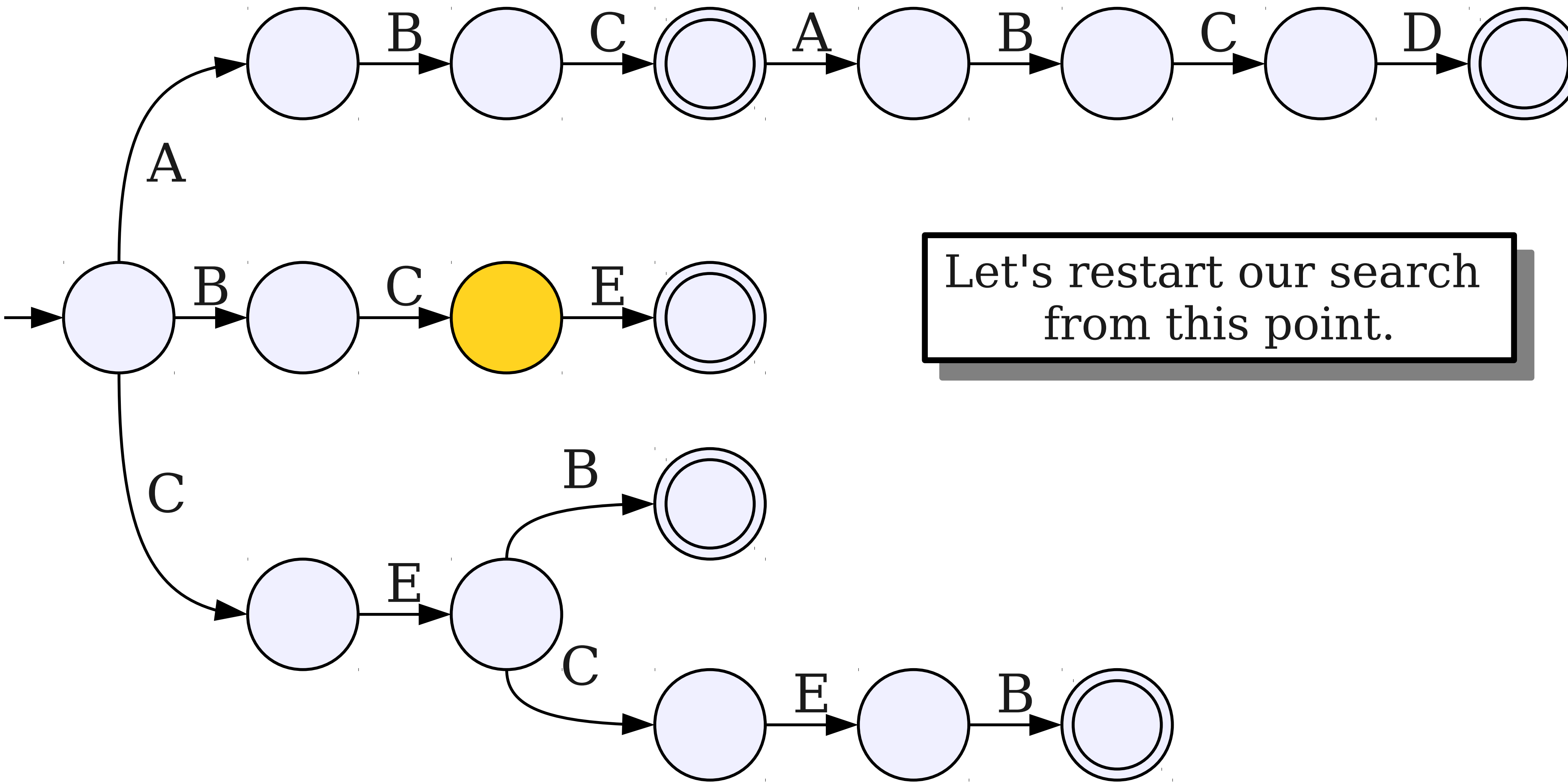
<u>A B C</u> E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



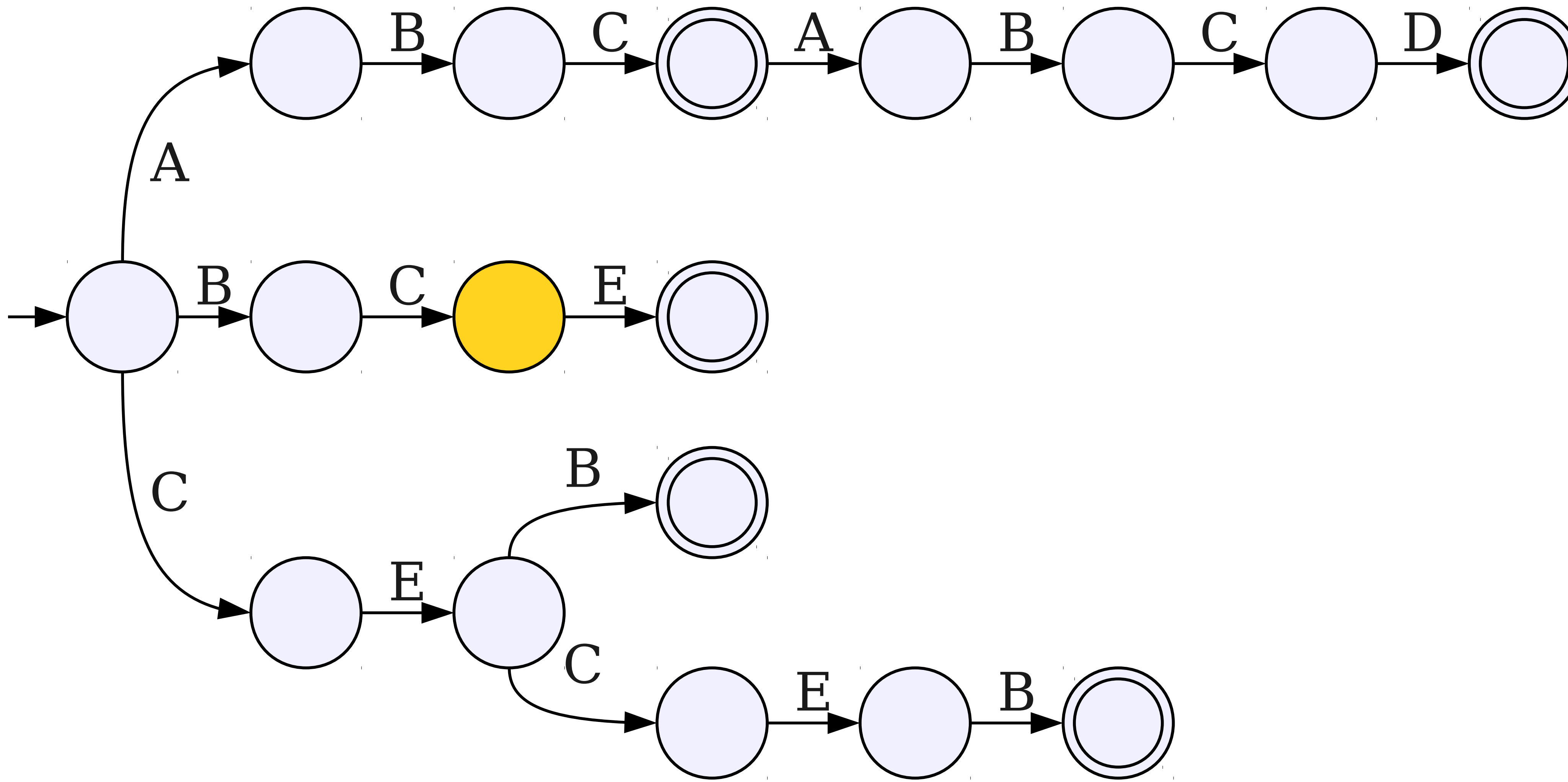
<u>A B C</u> E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



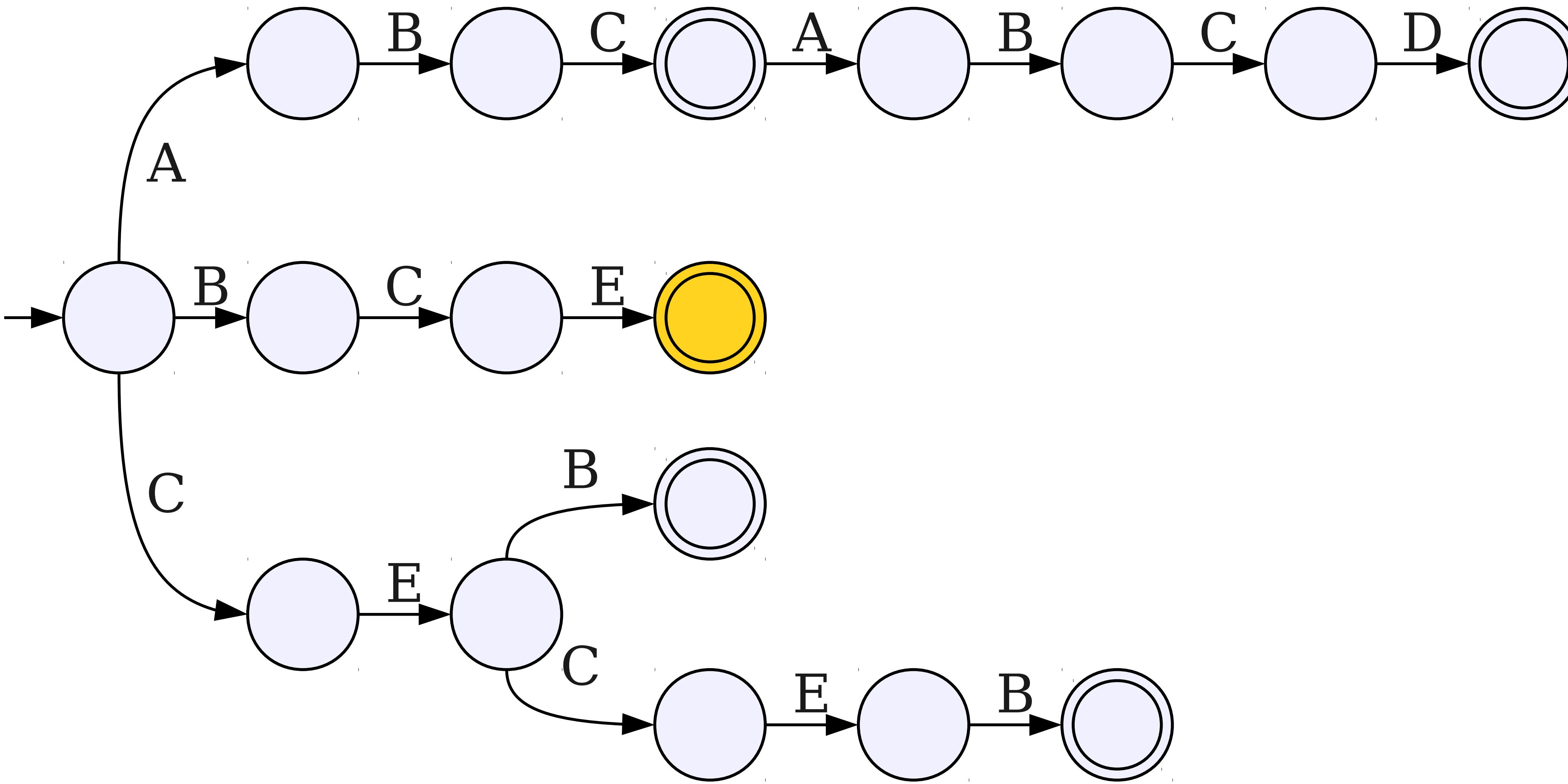
<u>A B C</u> E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



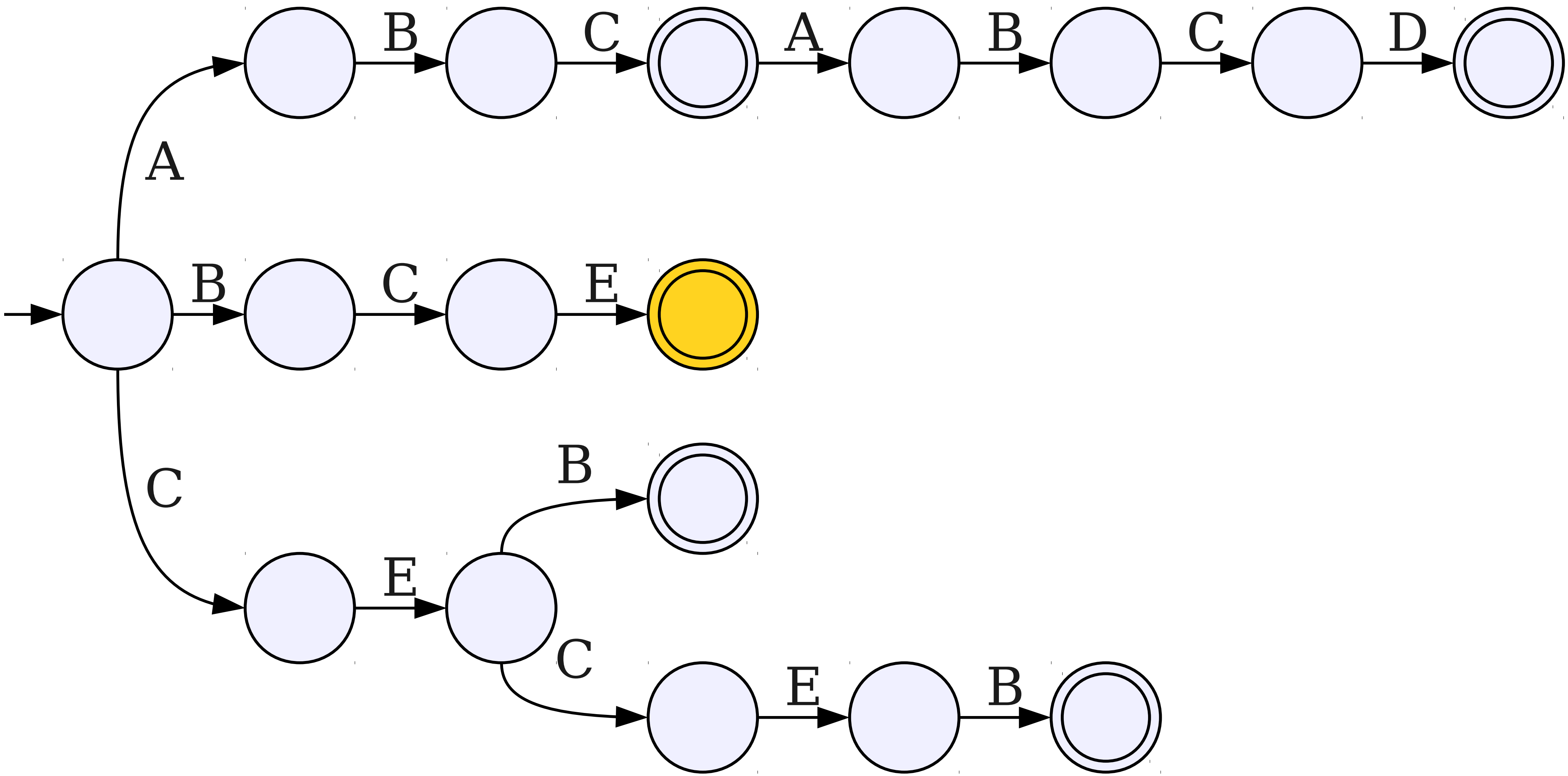
A B C E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



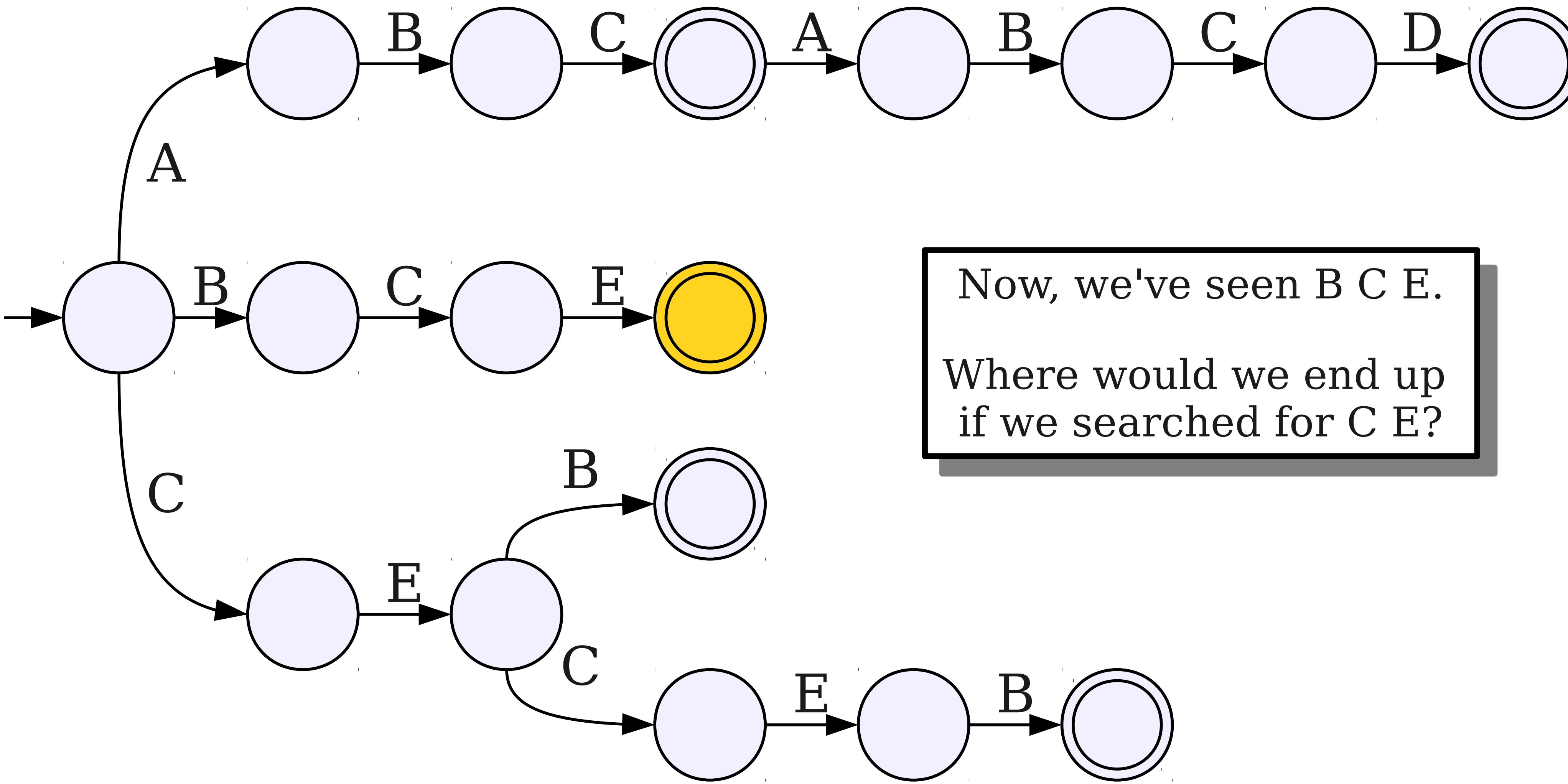
<u>A B C</u> E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



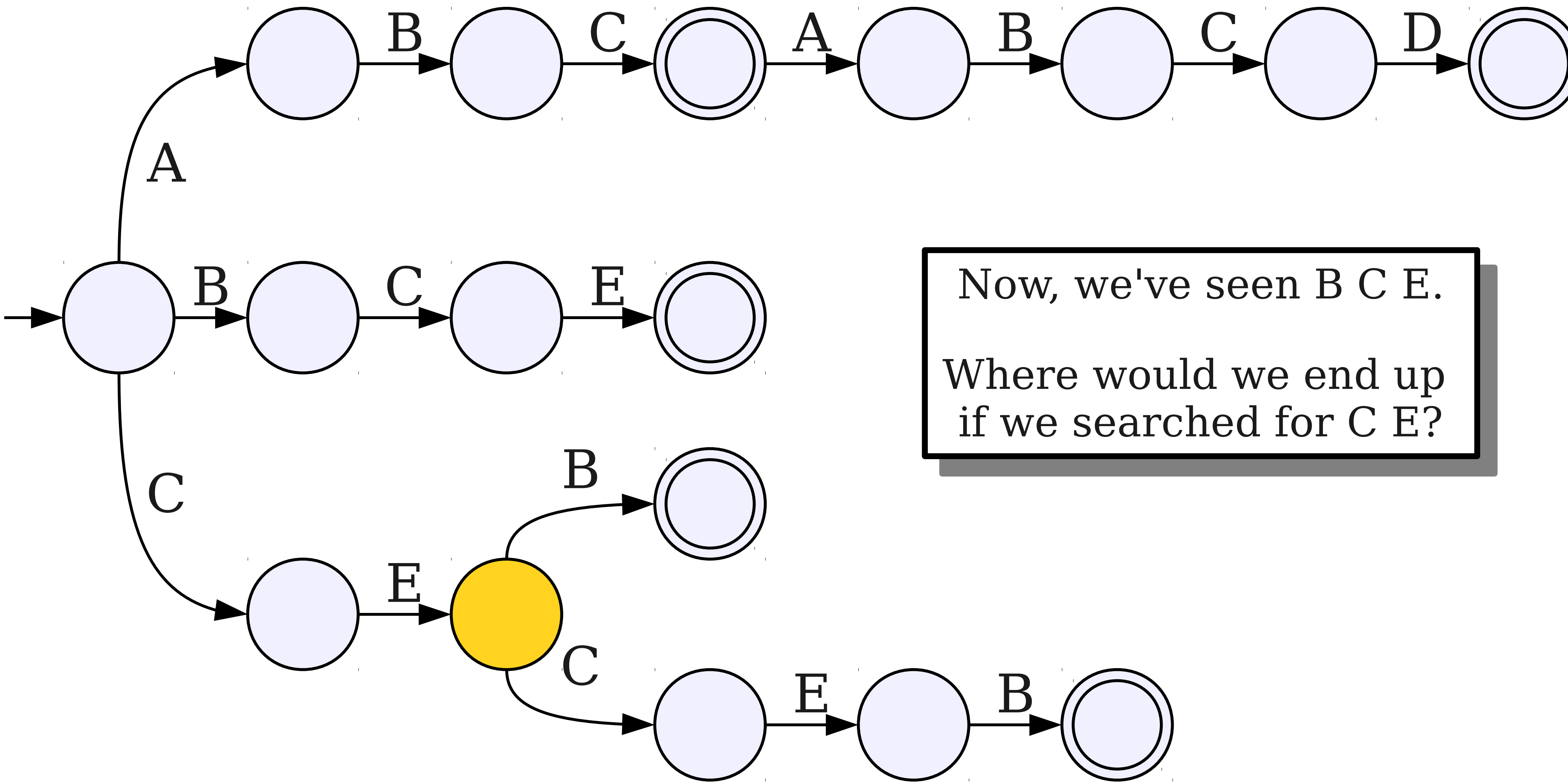
A B C E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



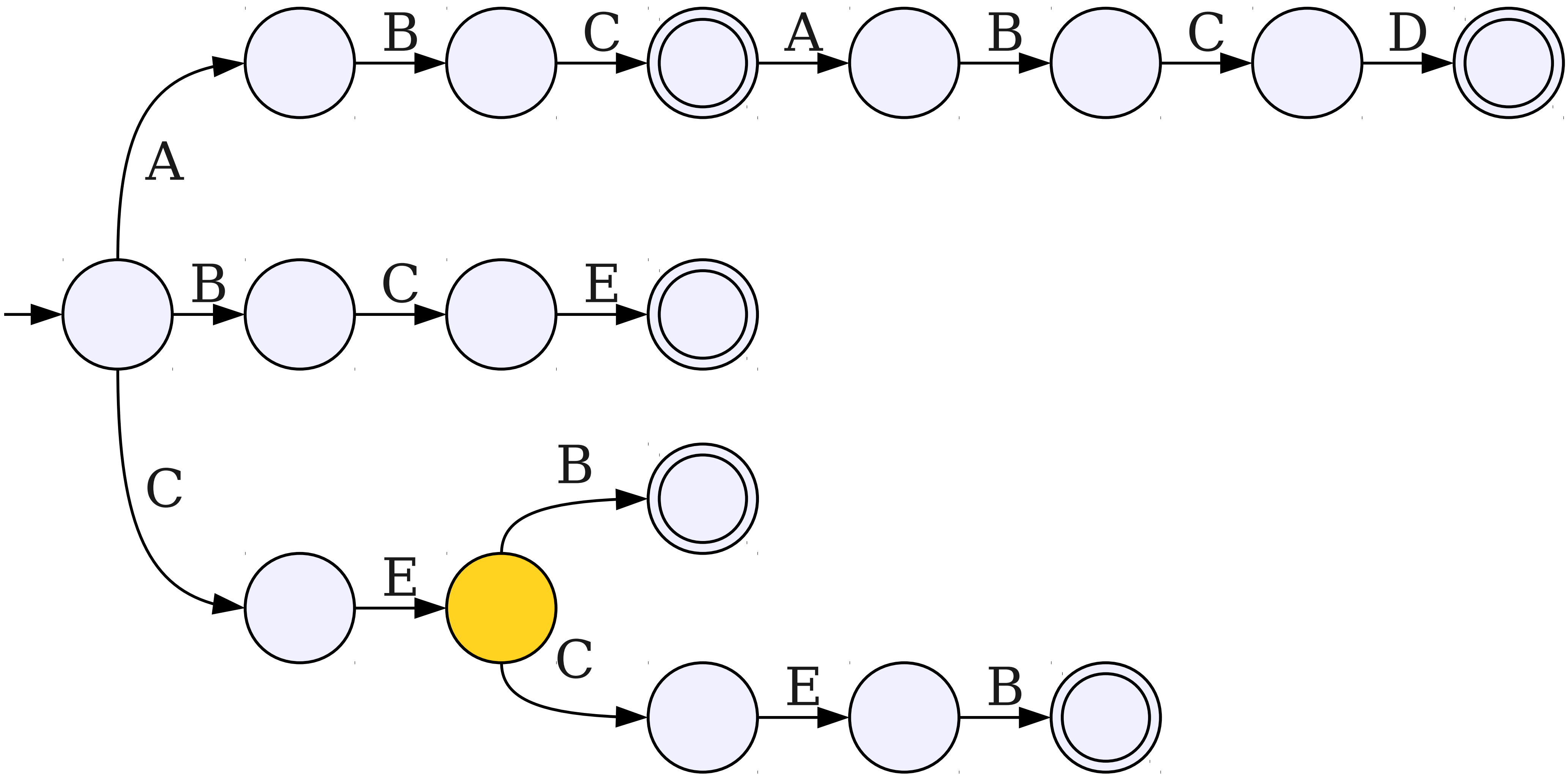
<u>A B C</u> E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



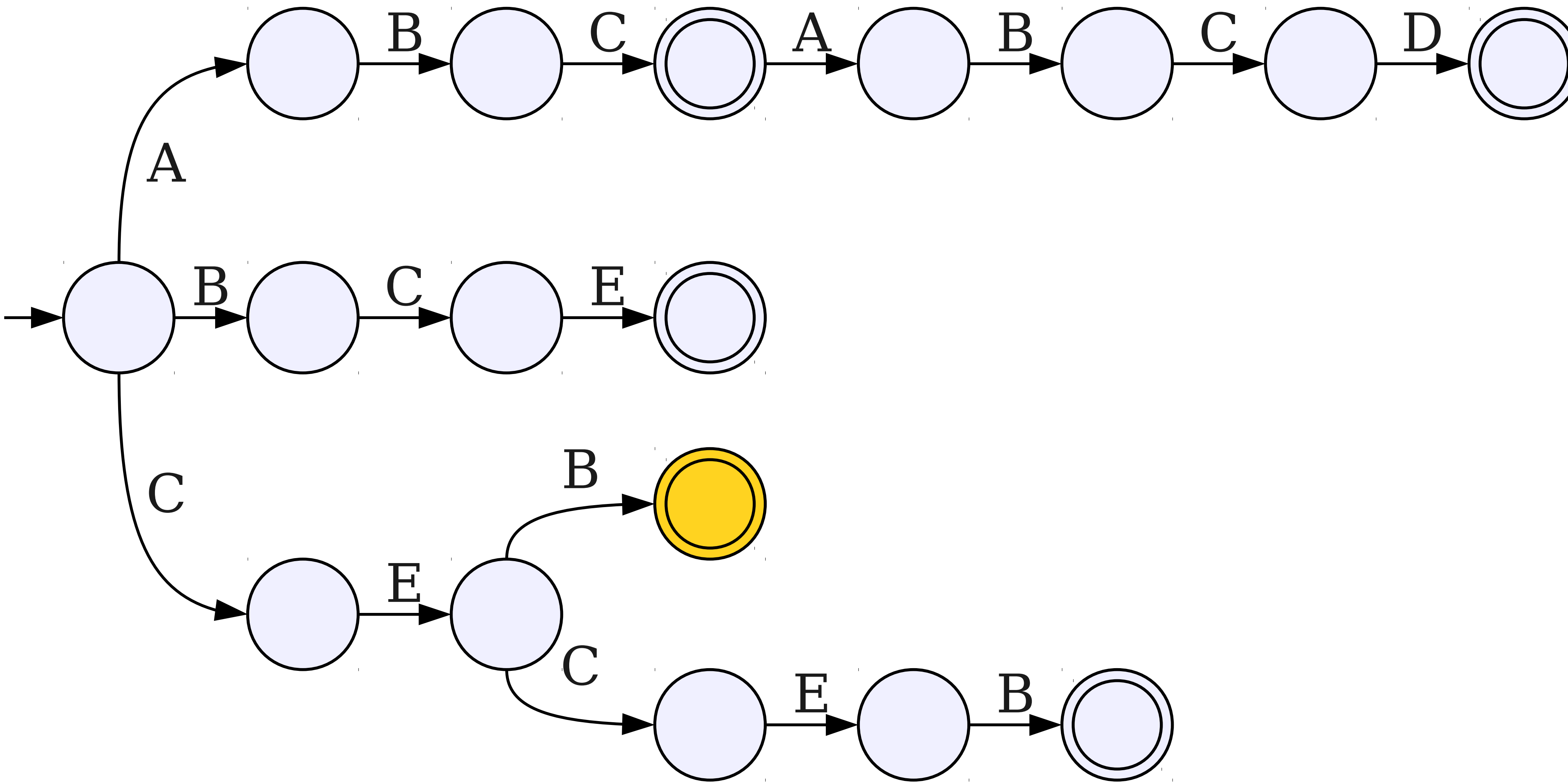
A B C E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



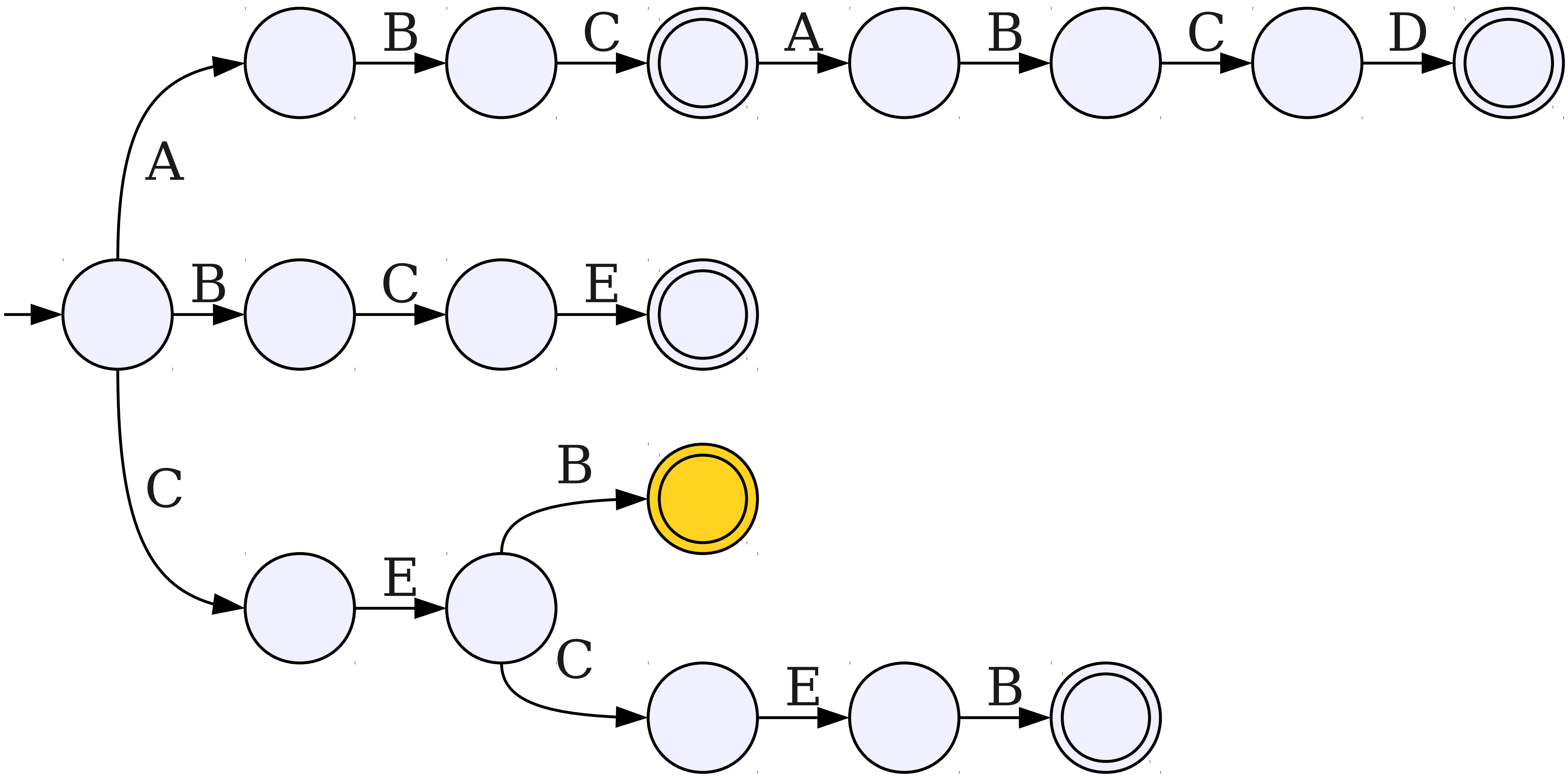
A B C E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



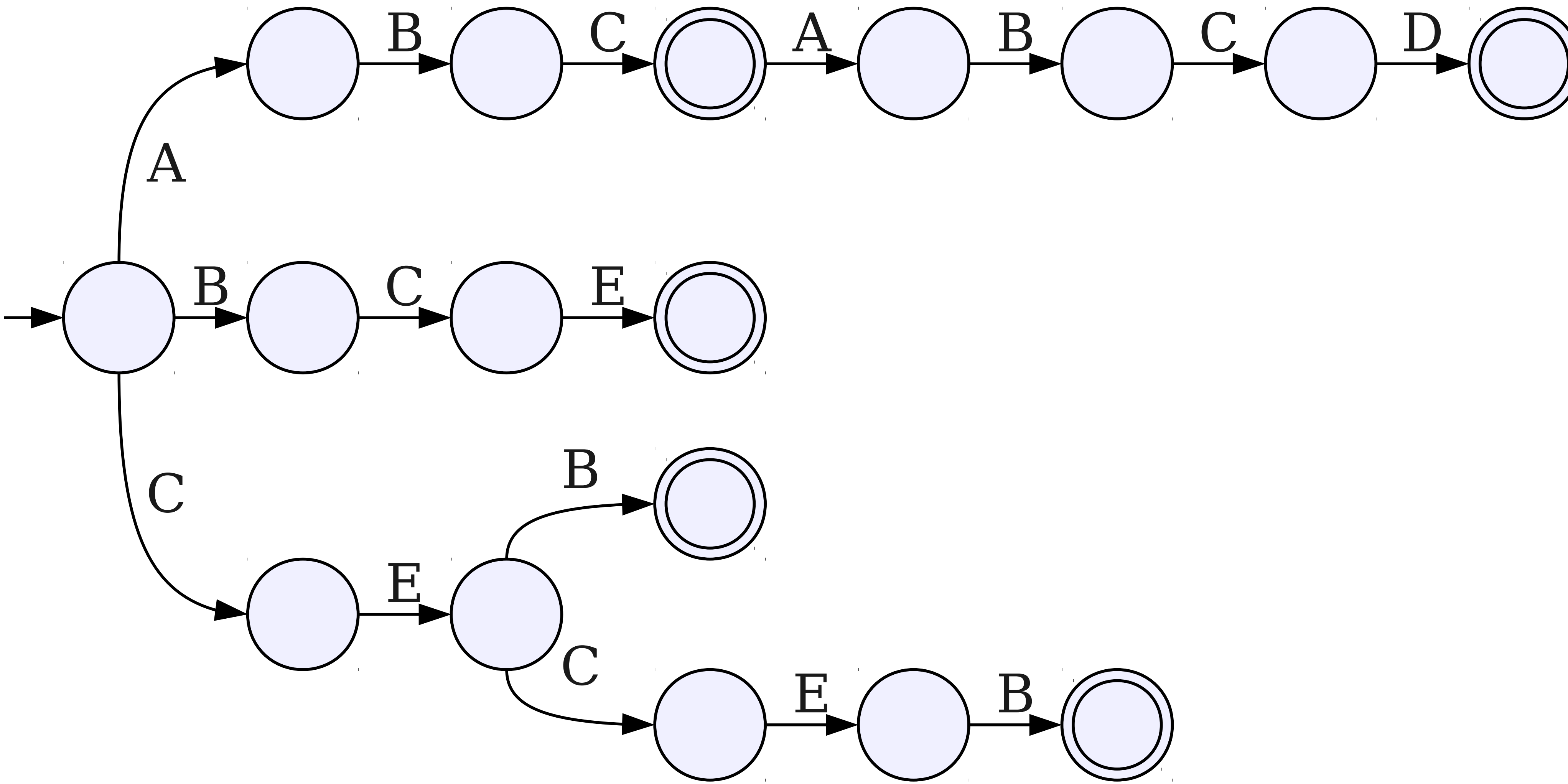
A B C E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



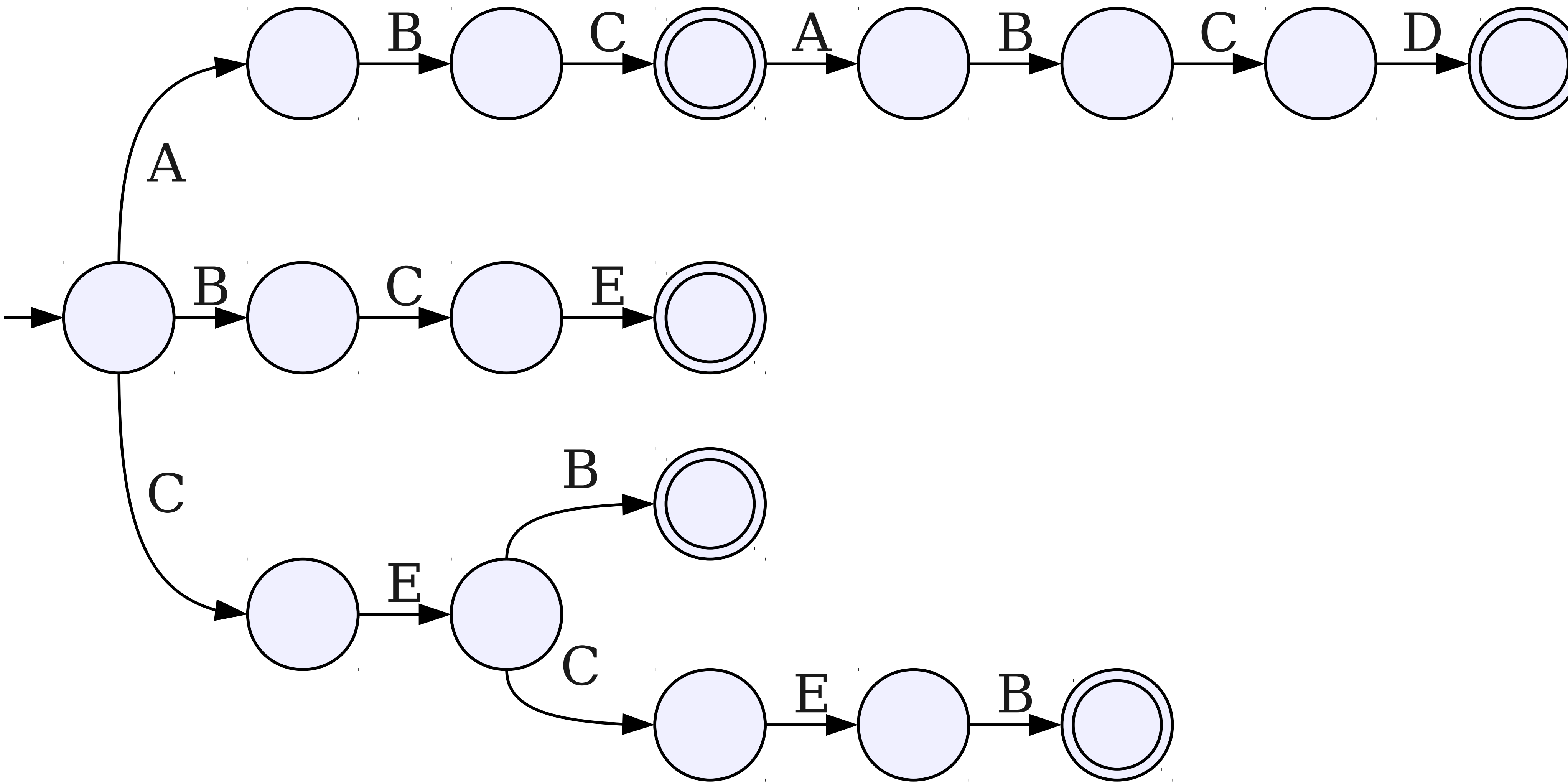
A B C E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



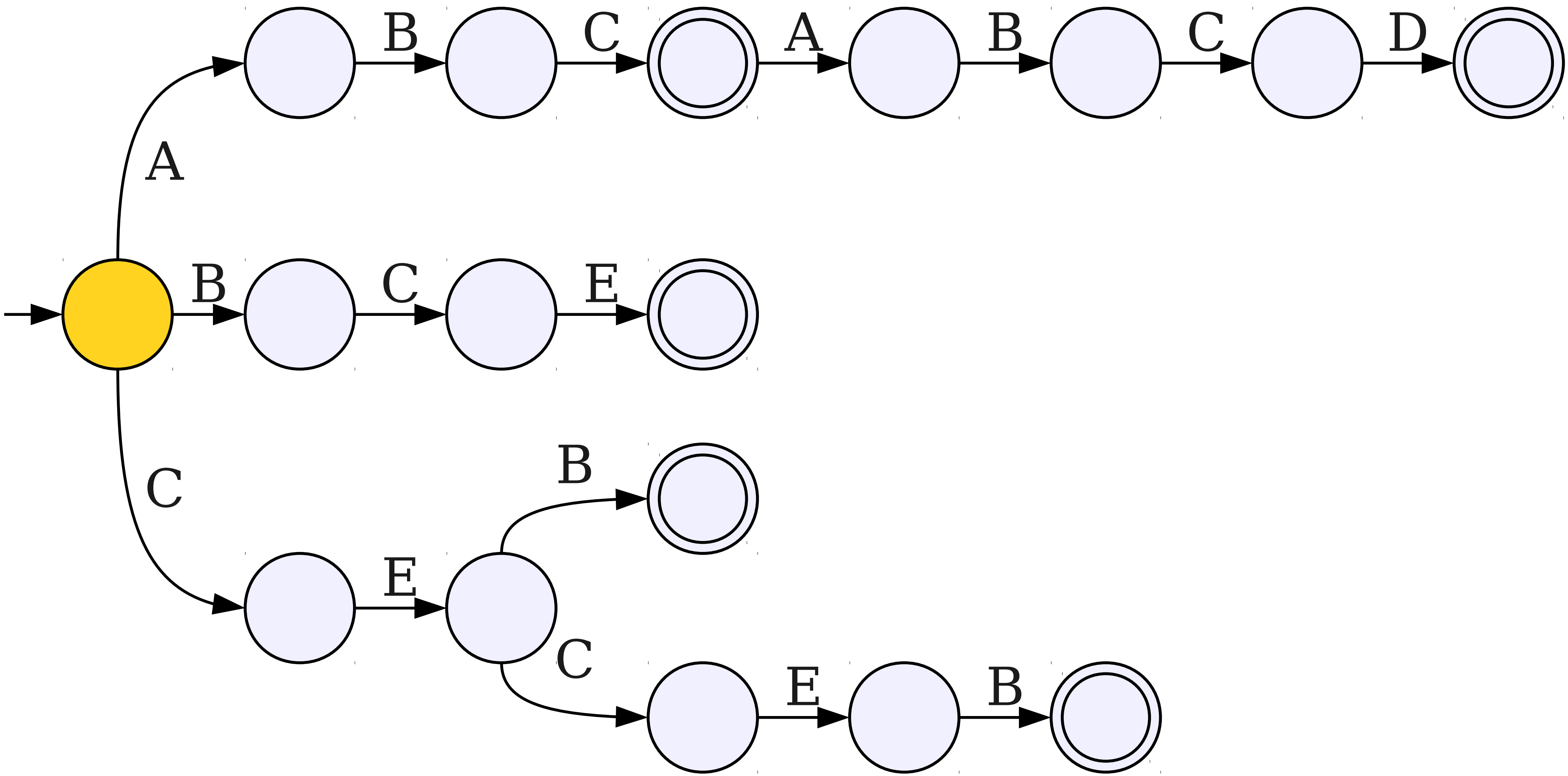
A B C E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



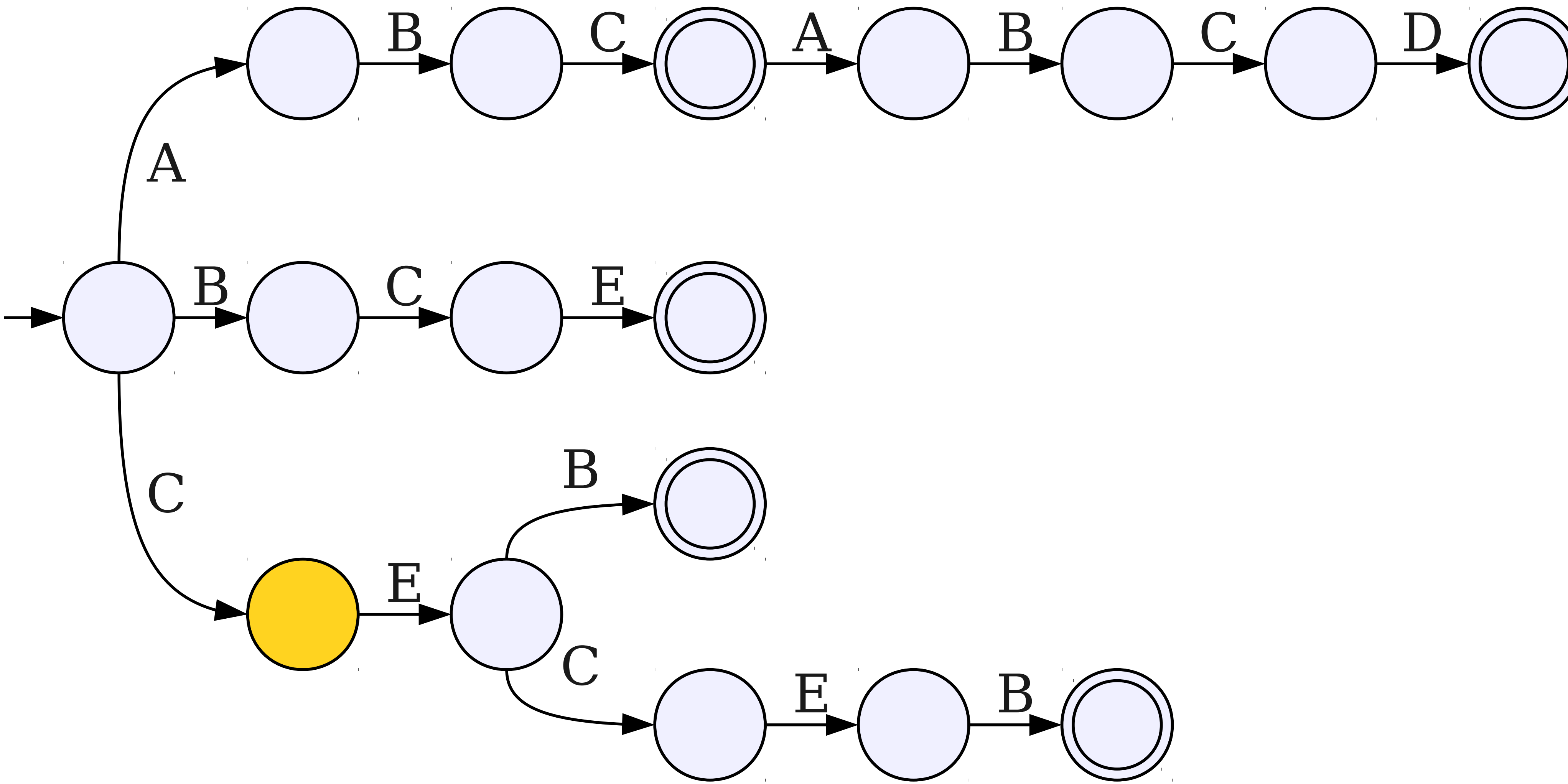
C E B C

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



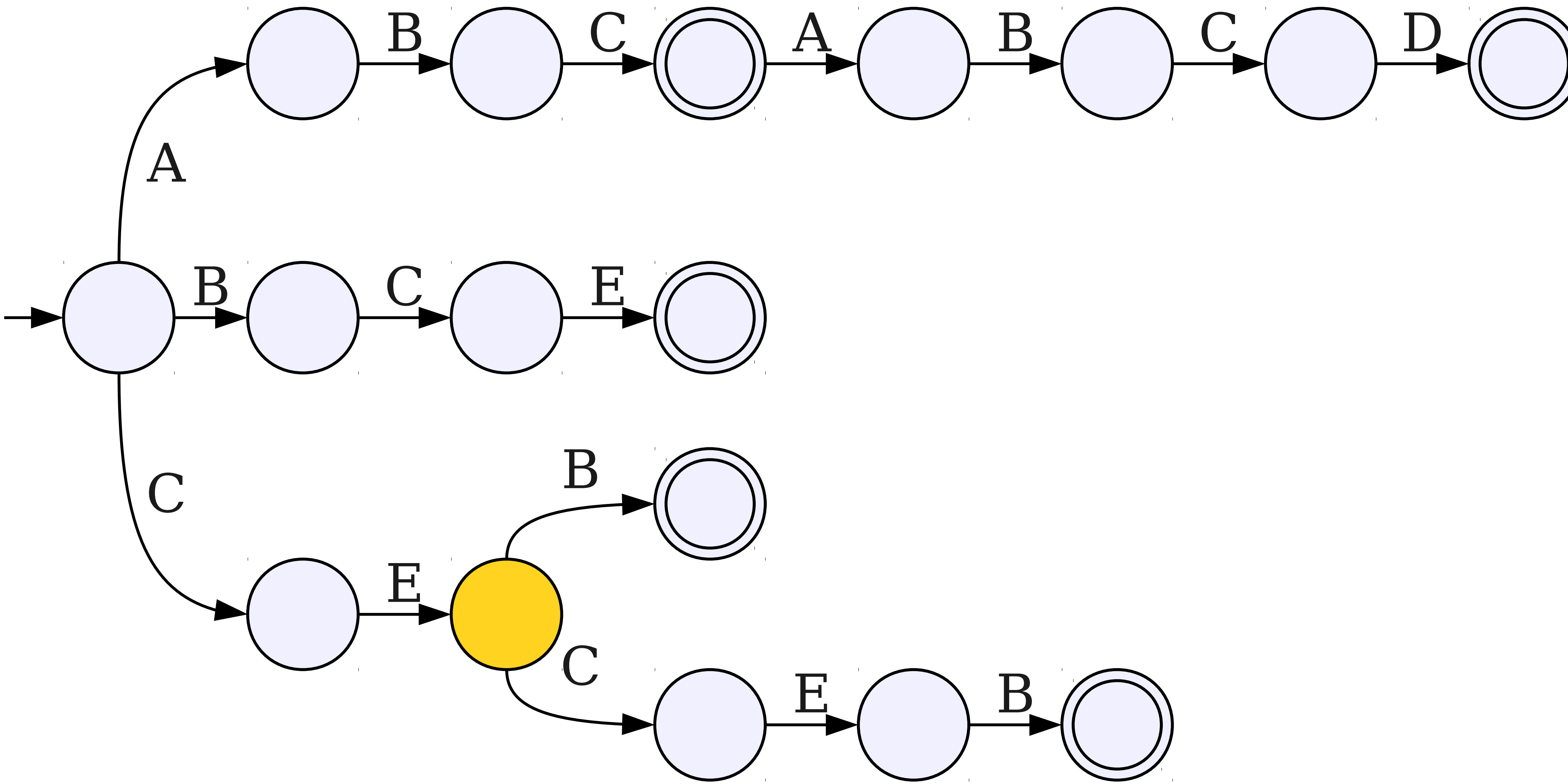
C E B C

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



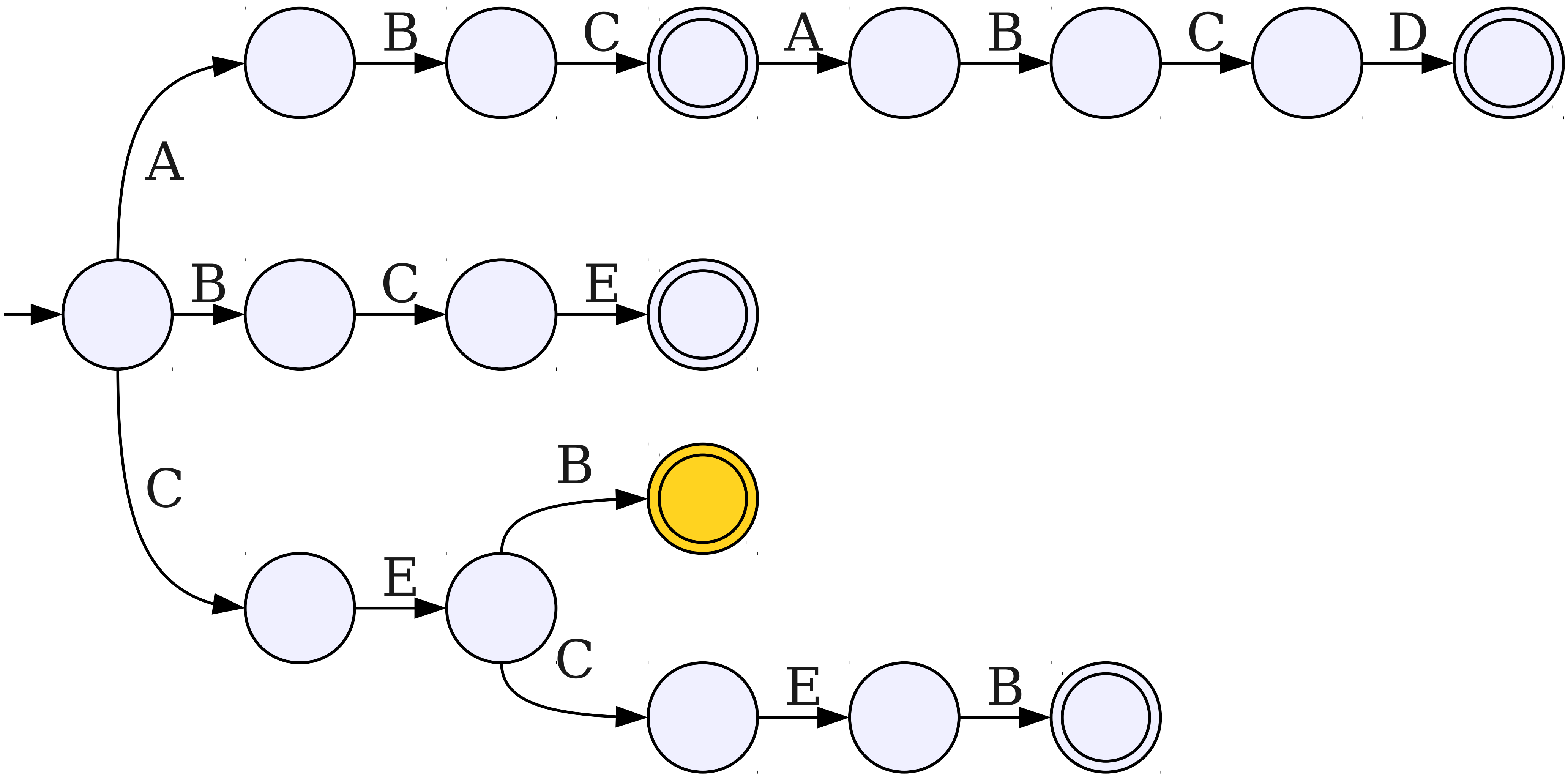
C E B C

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



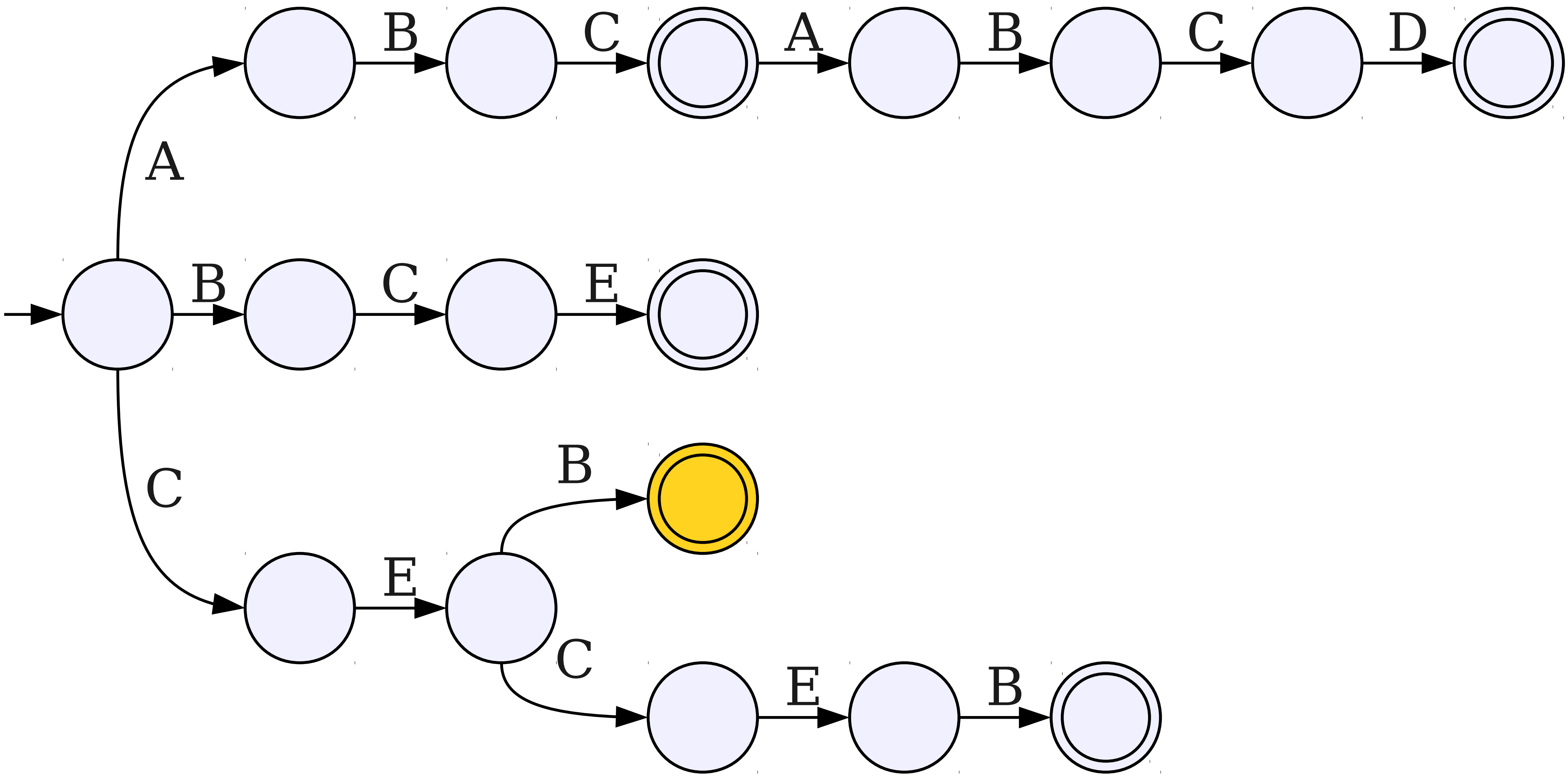
C E B C

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



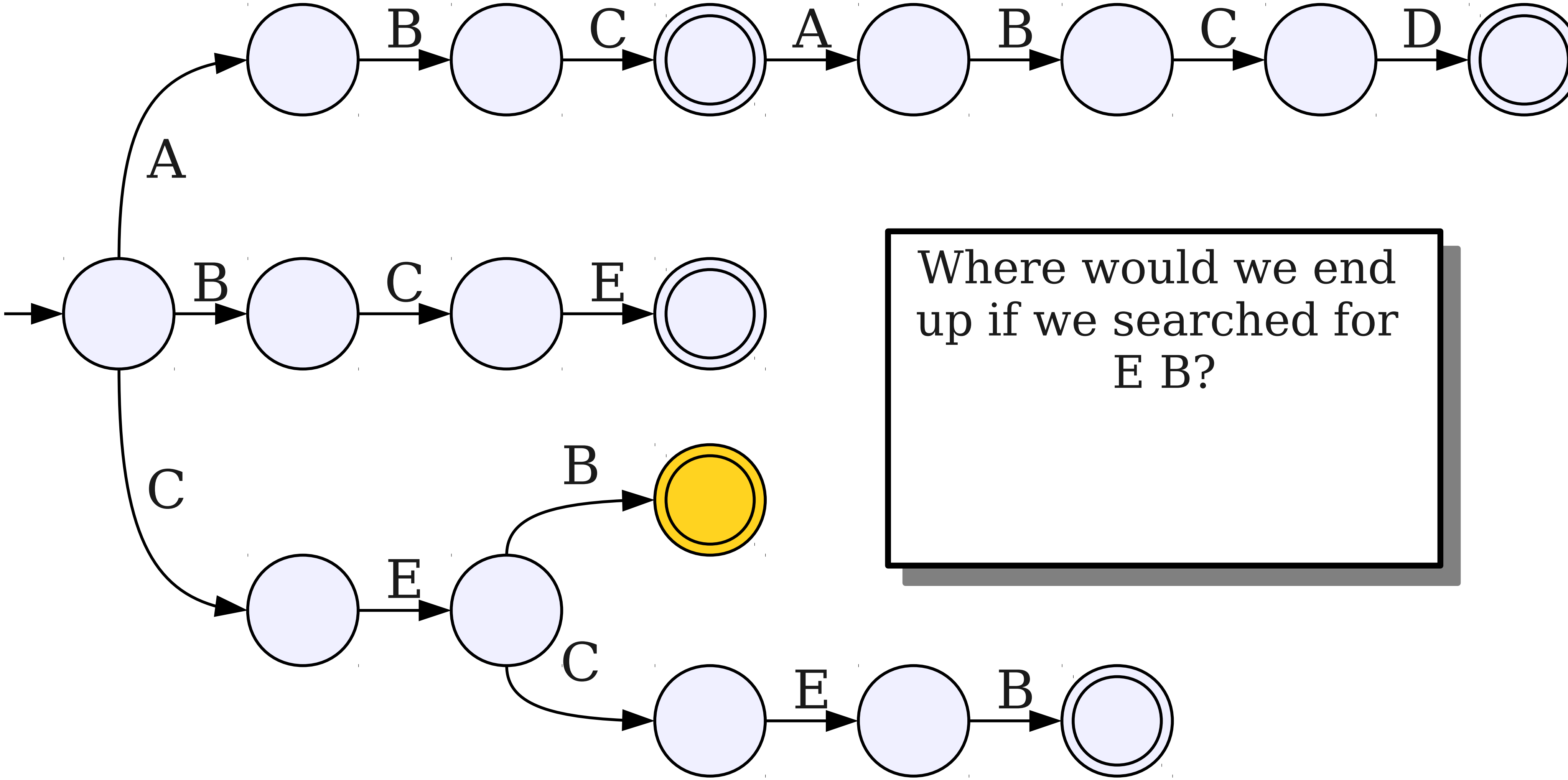
C E B C

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



C E B C

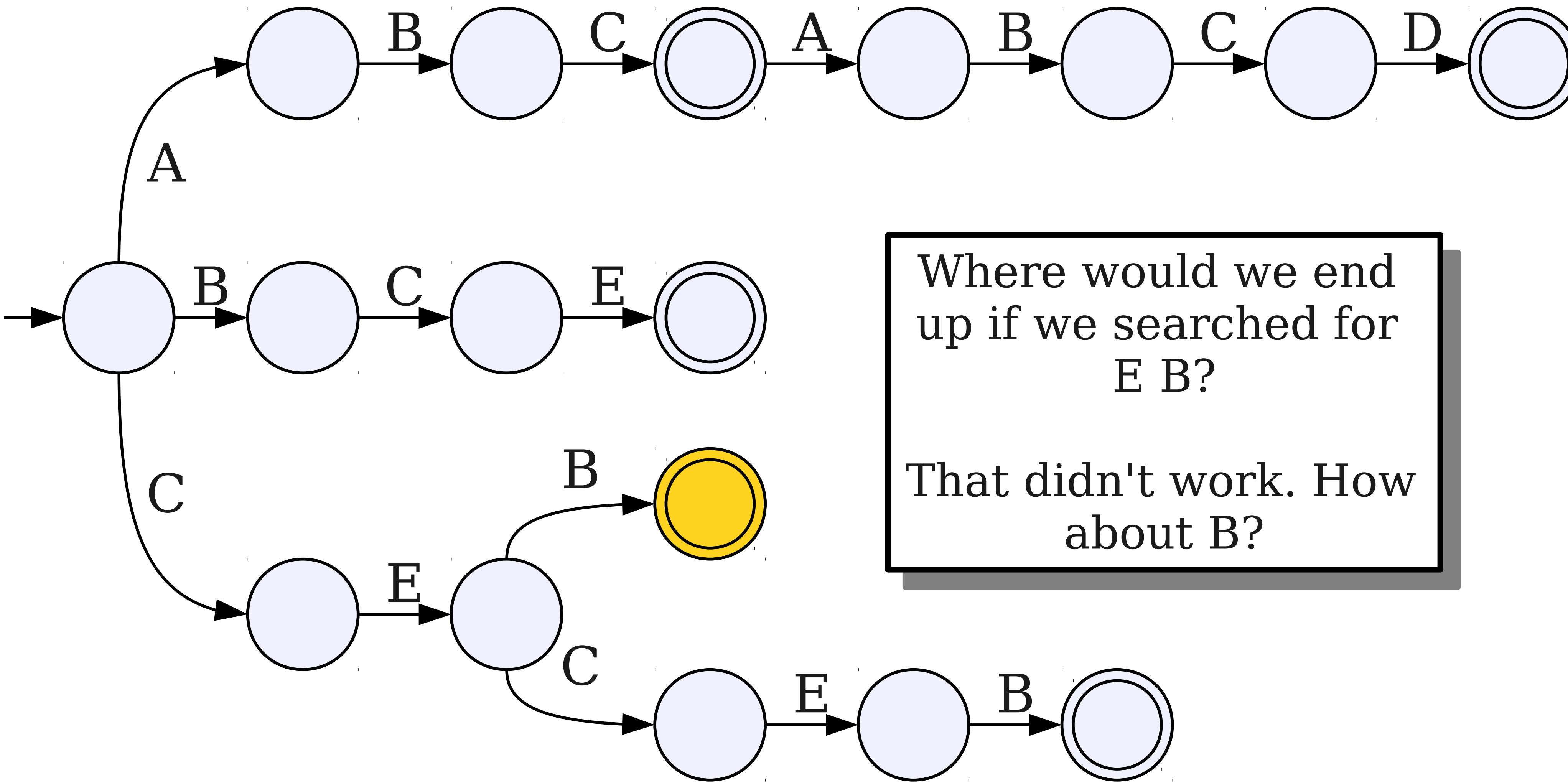
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



Where would we end up if we searched for E B?

C E B C

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC

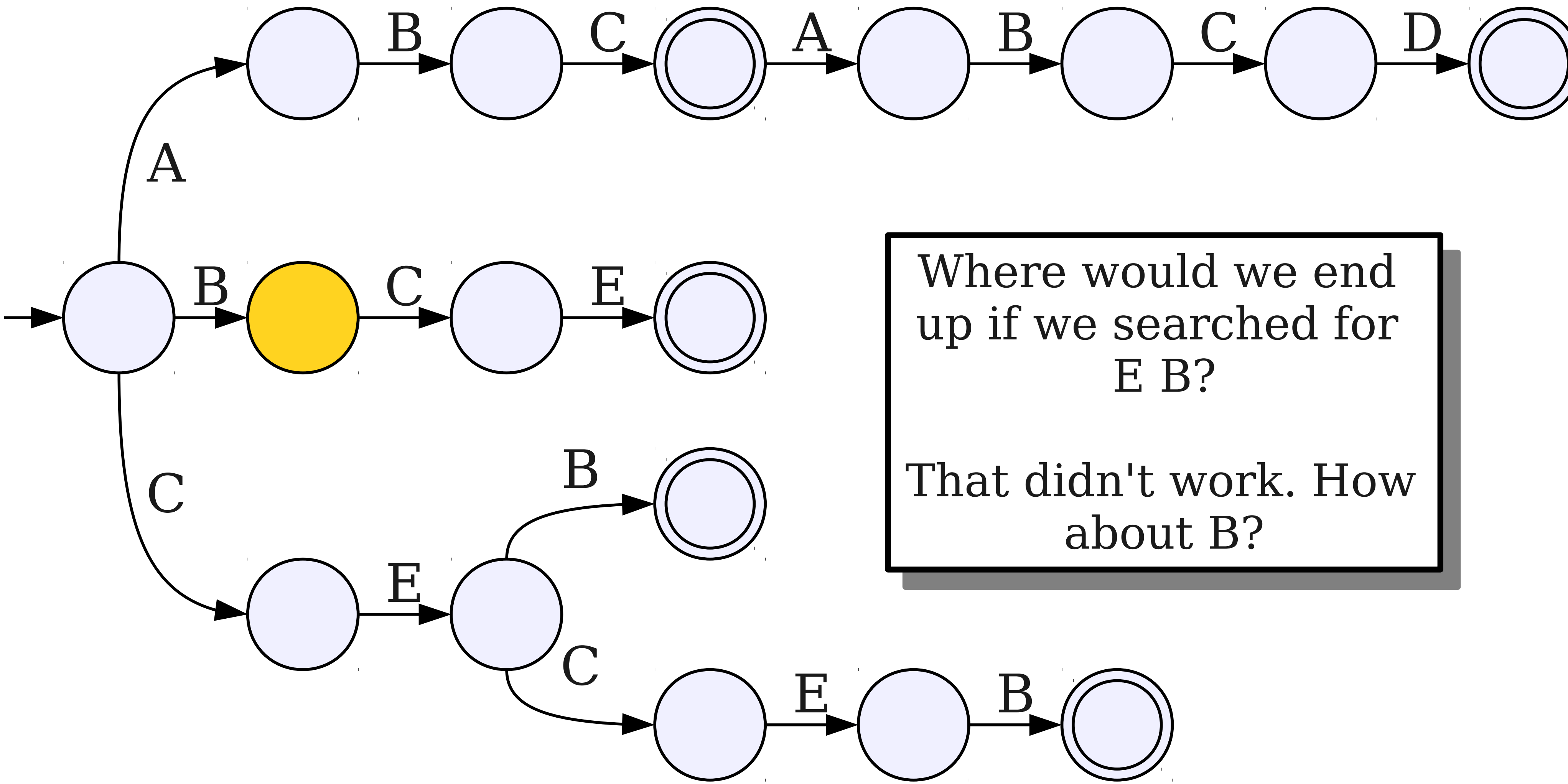


Where would we end up if we searched for E B?

That didn't work. How about B?

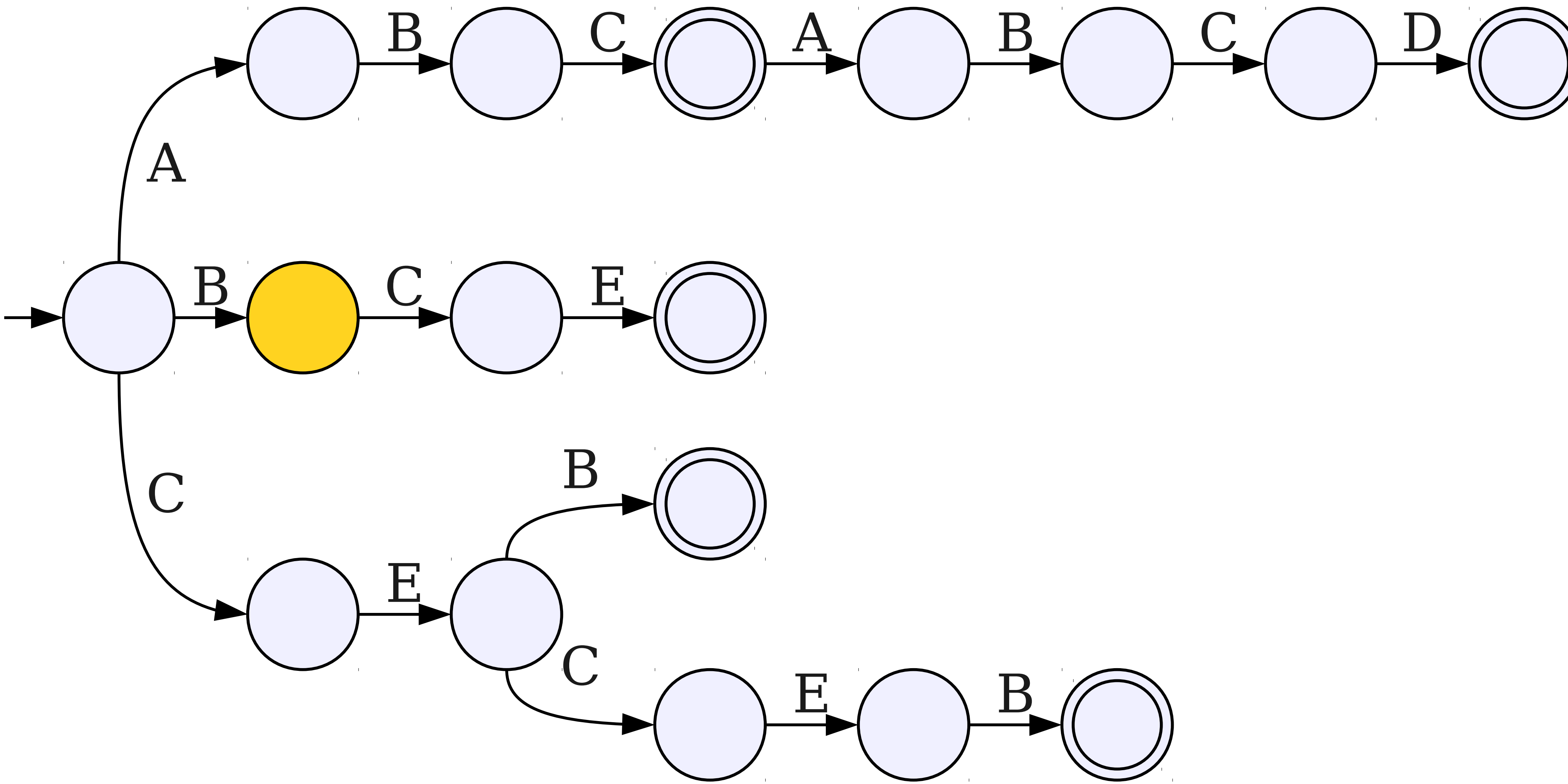
C E B C

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



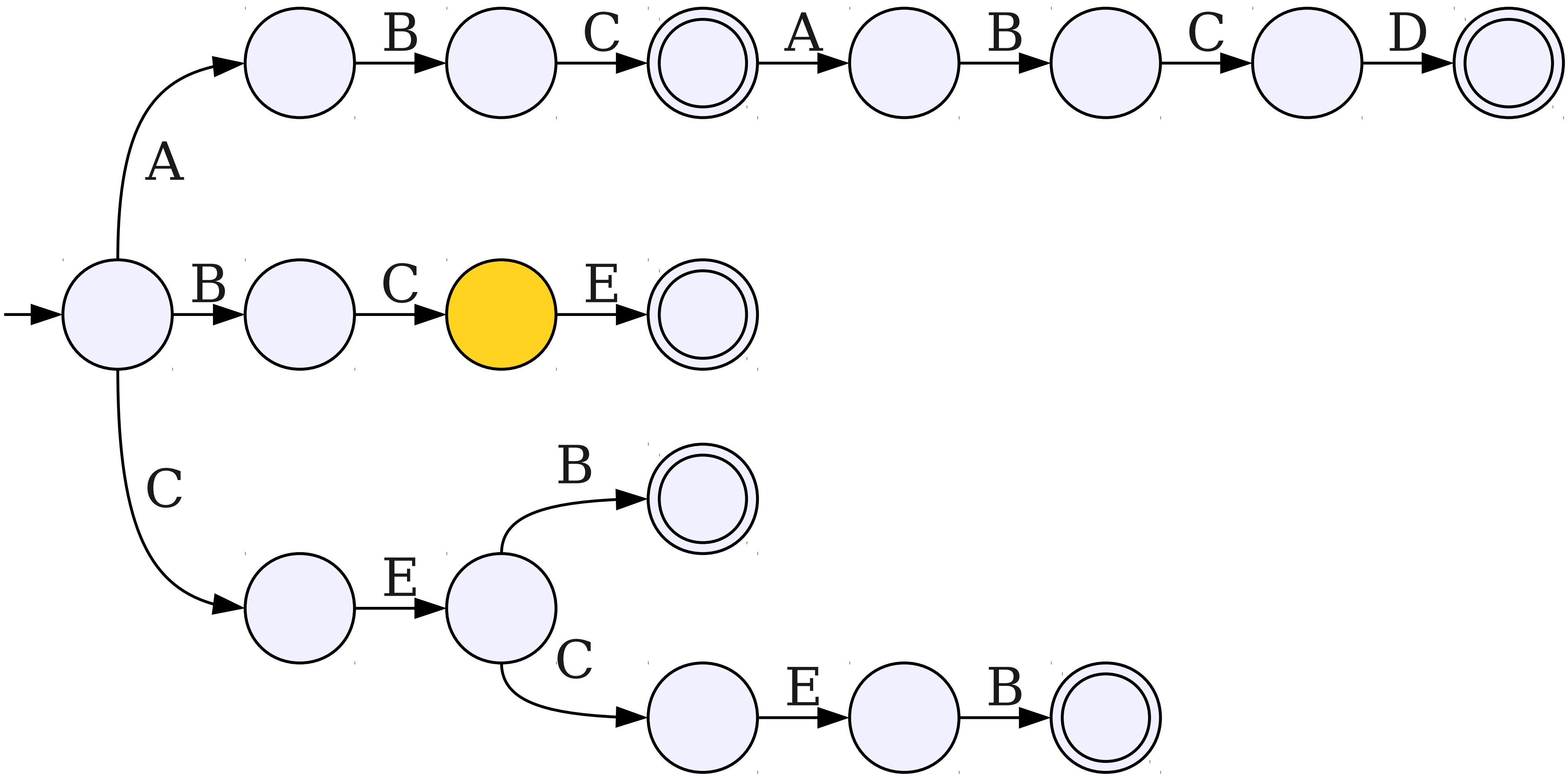
C E B C

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



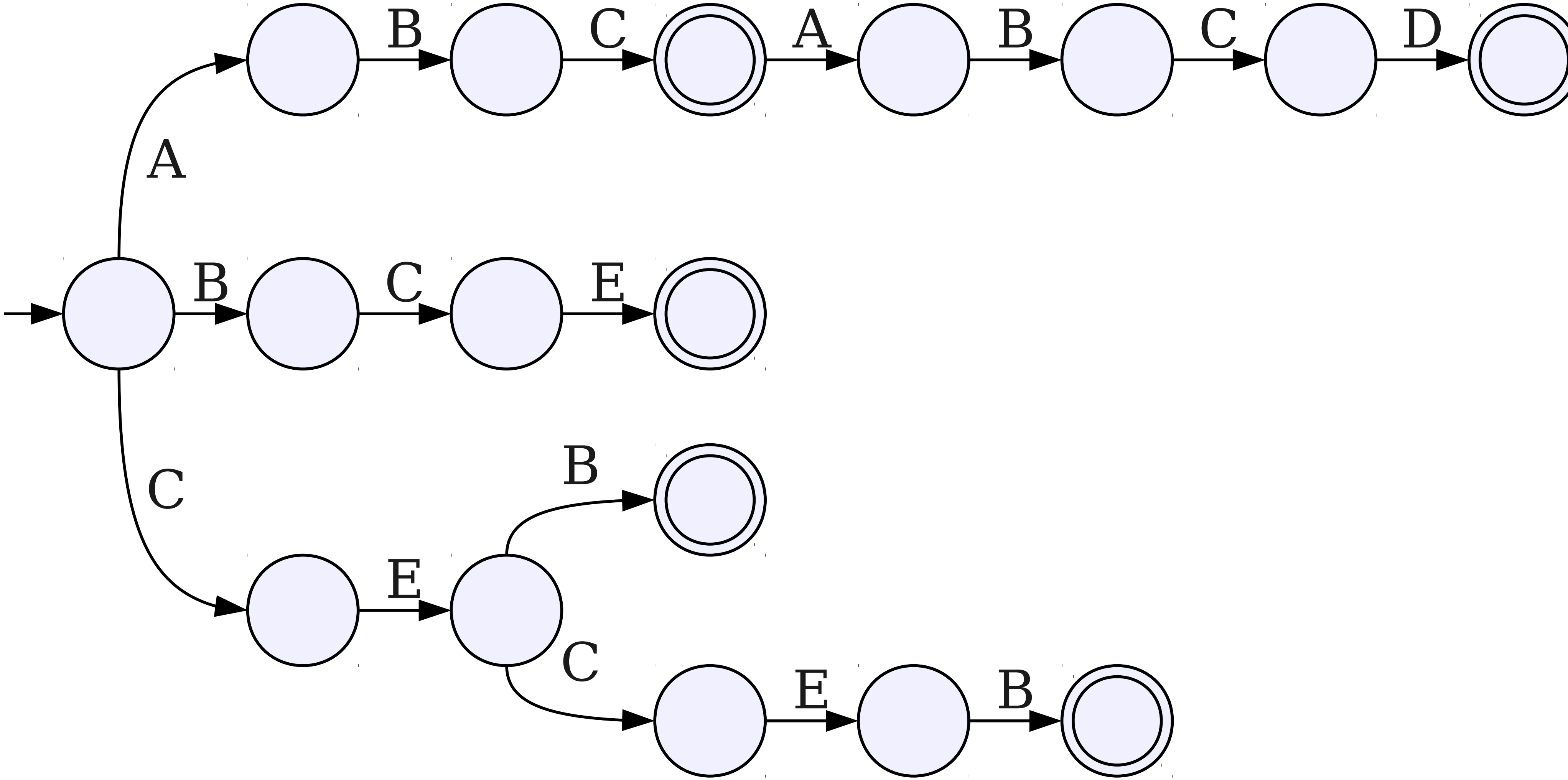
C E B C

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



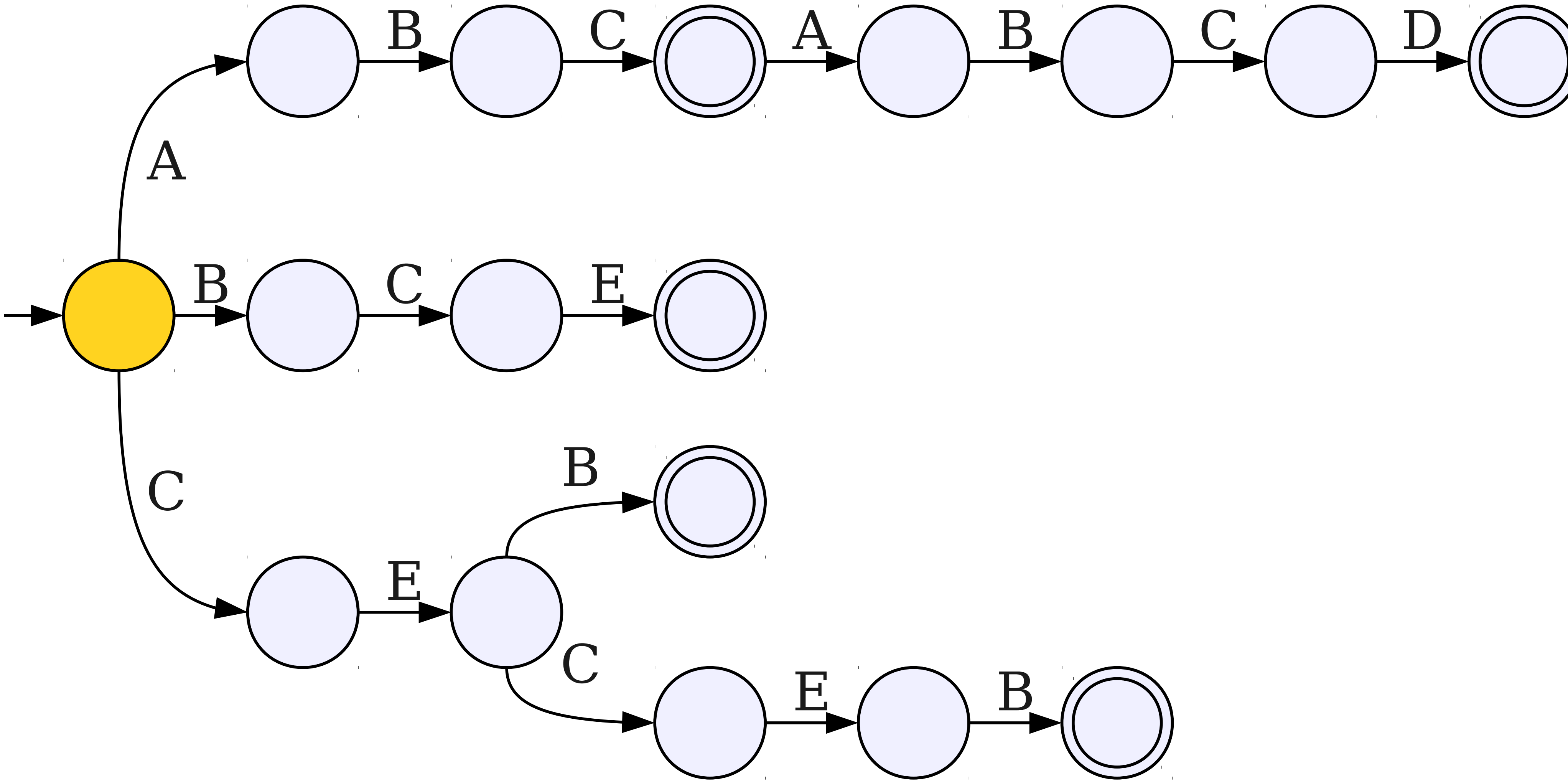
C E B C

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



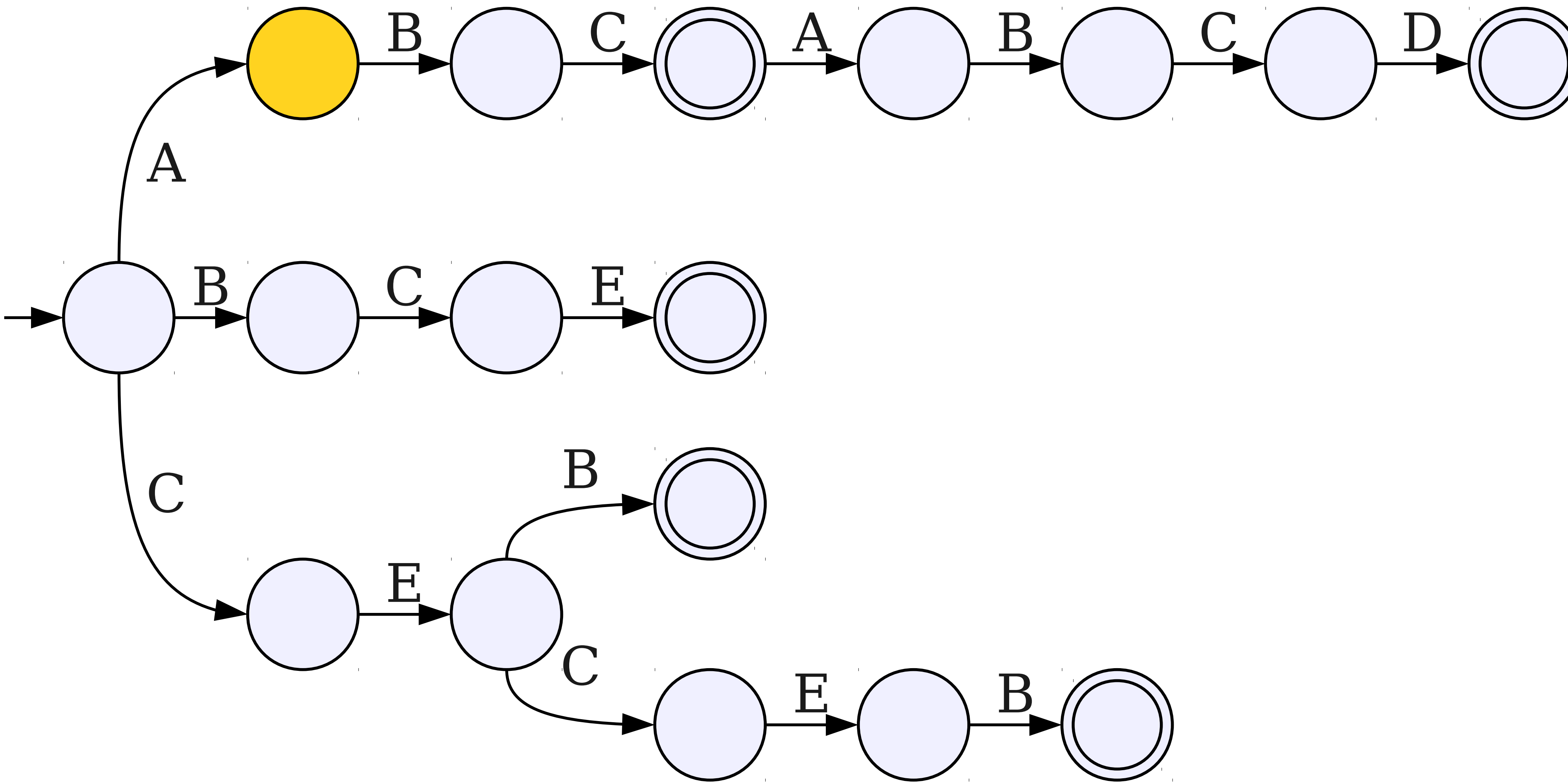
A B C A B C A

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



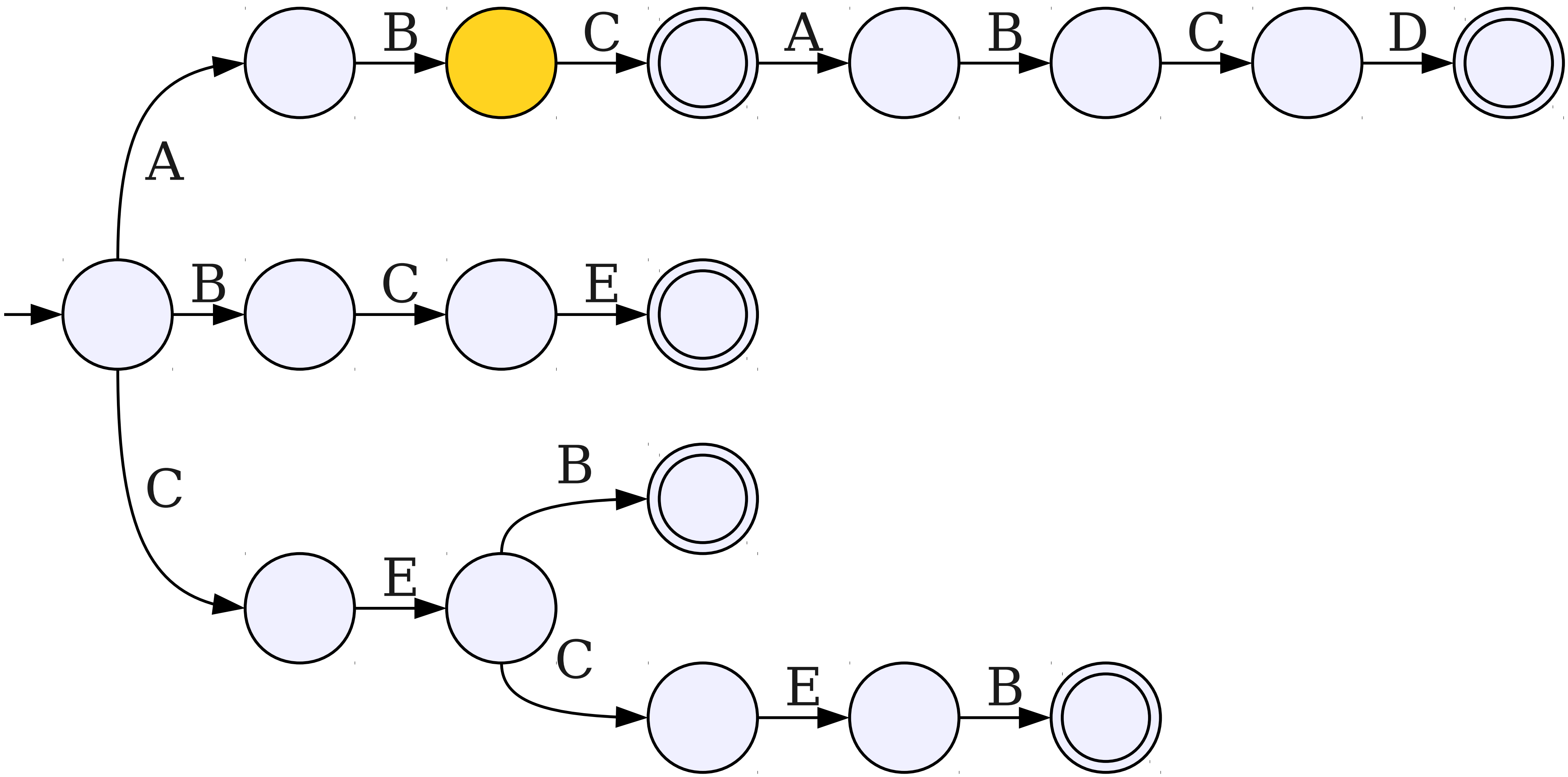
A B C A B C A

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



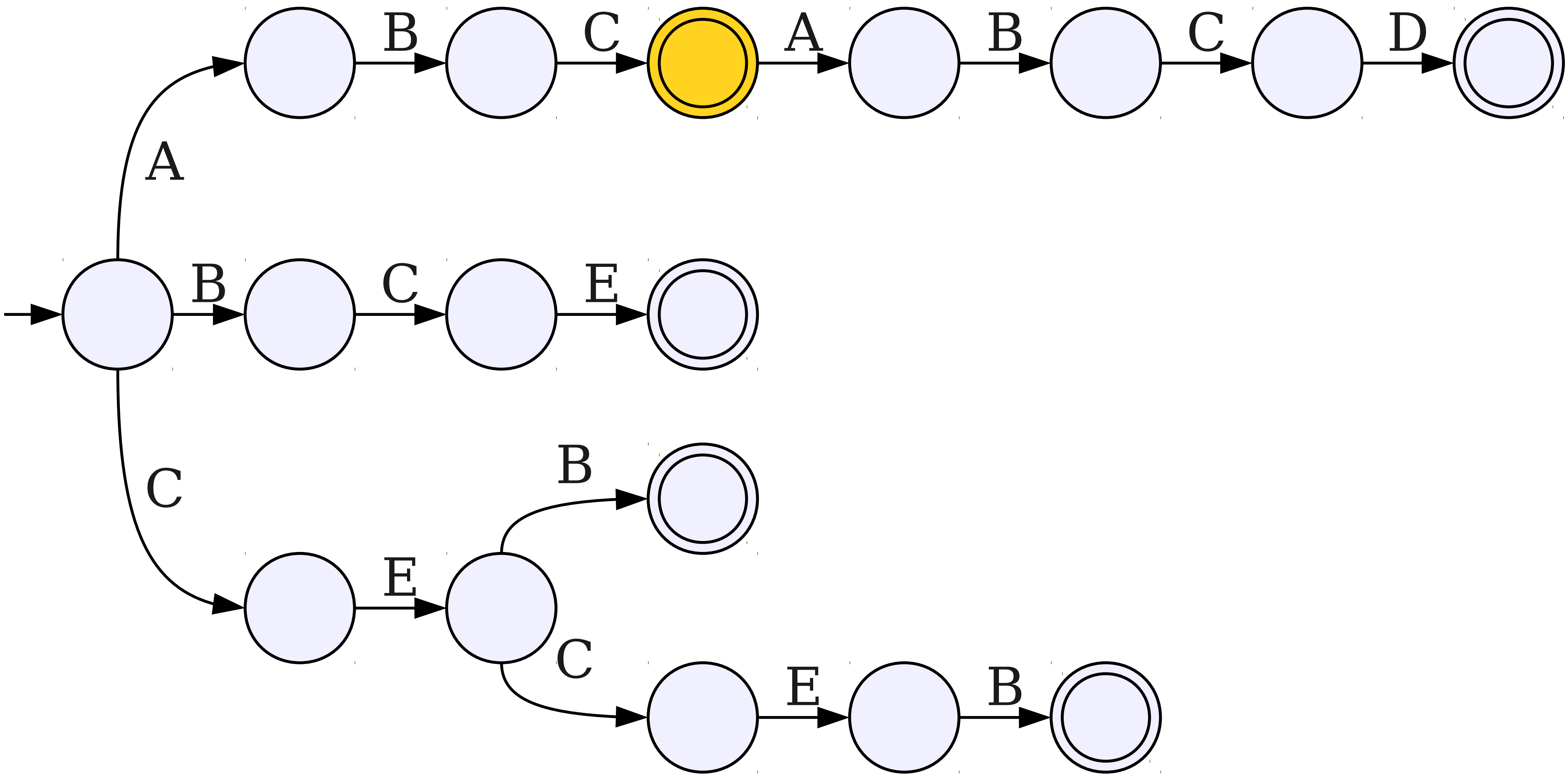
A B C A B C A

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



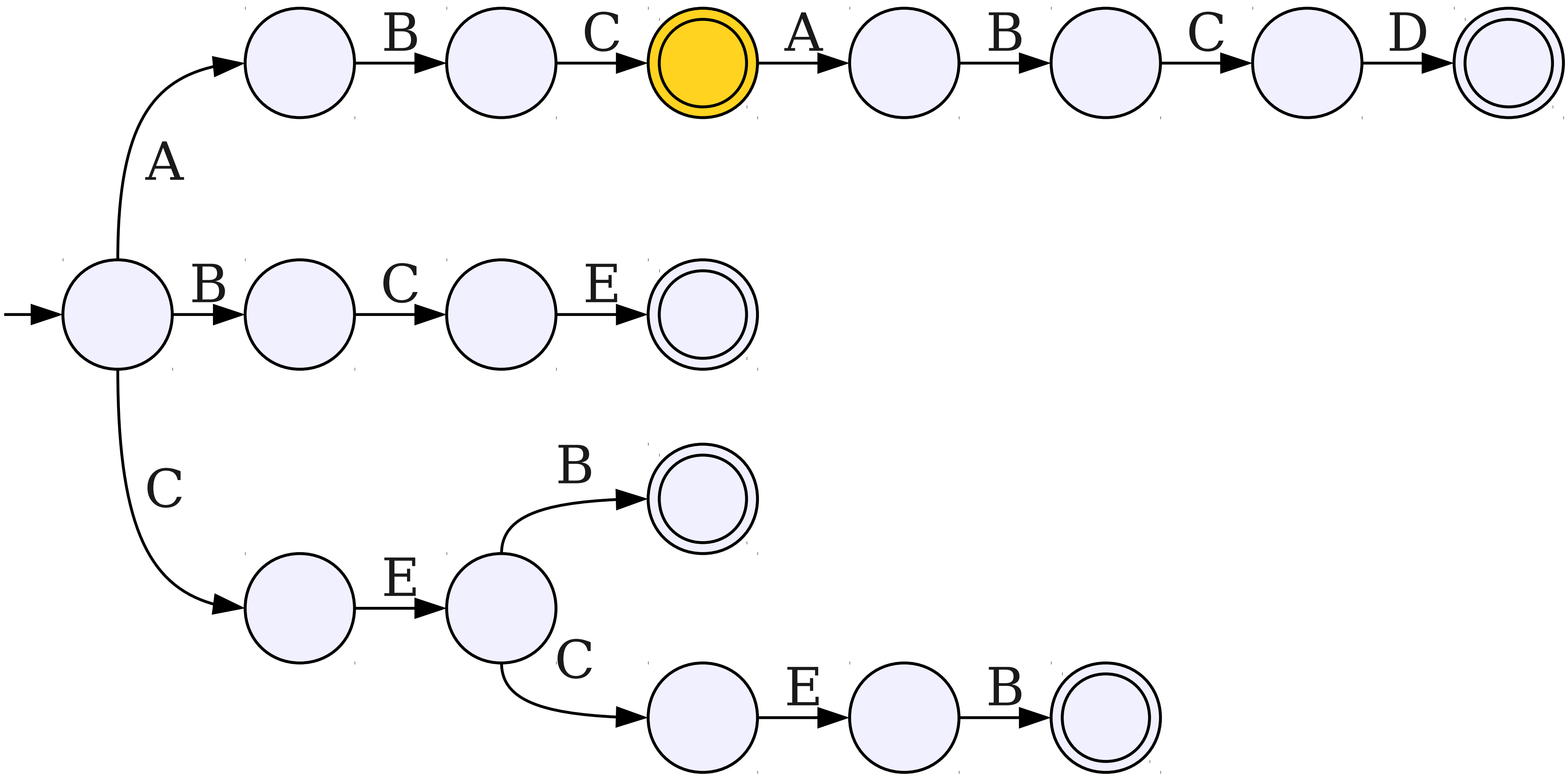
A B C A B C A

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



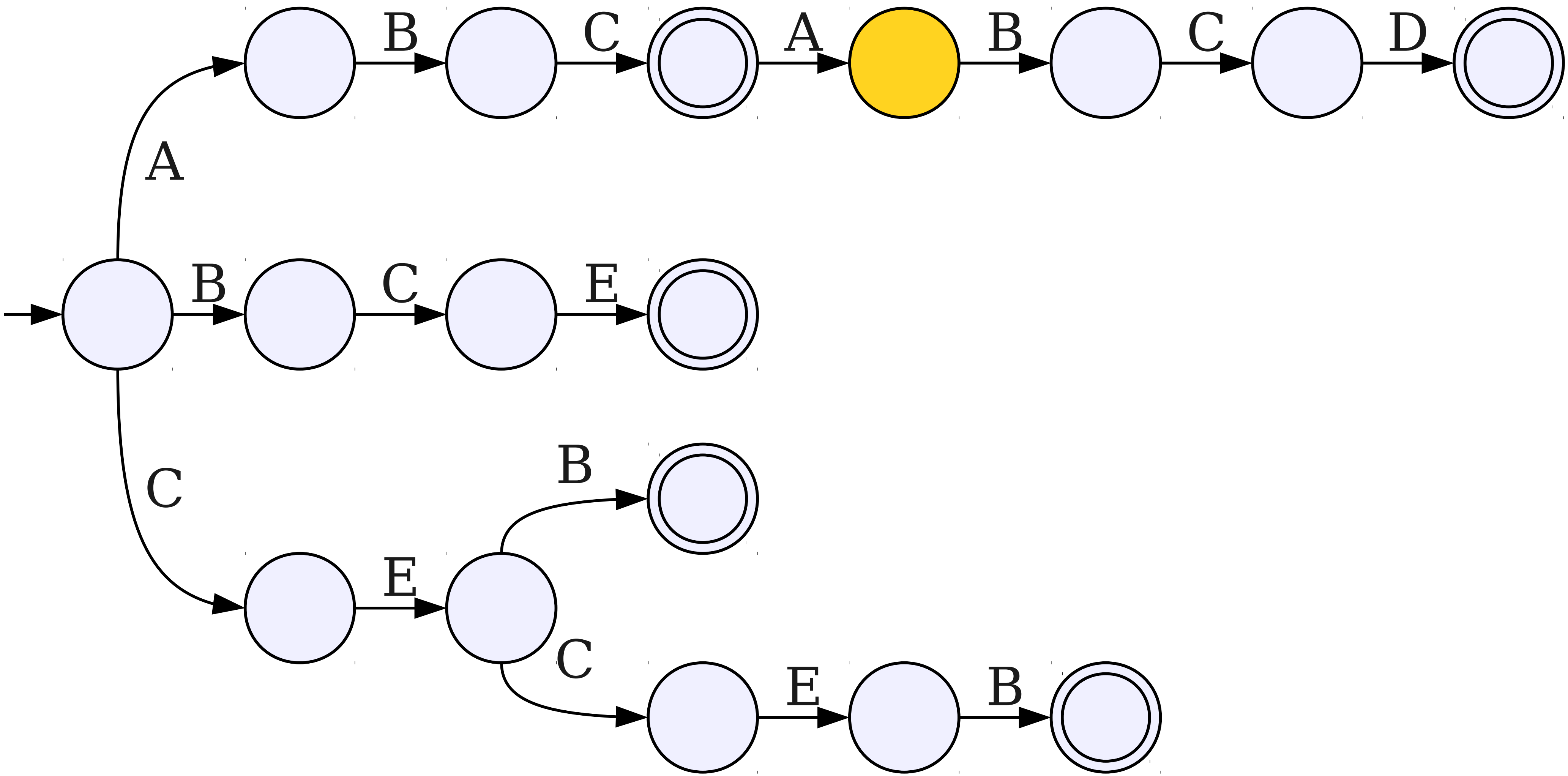
A B C A B C A

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



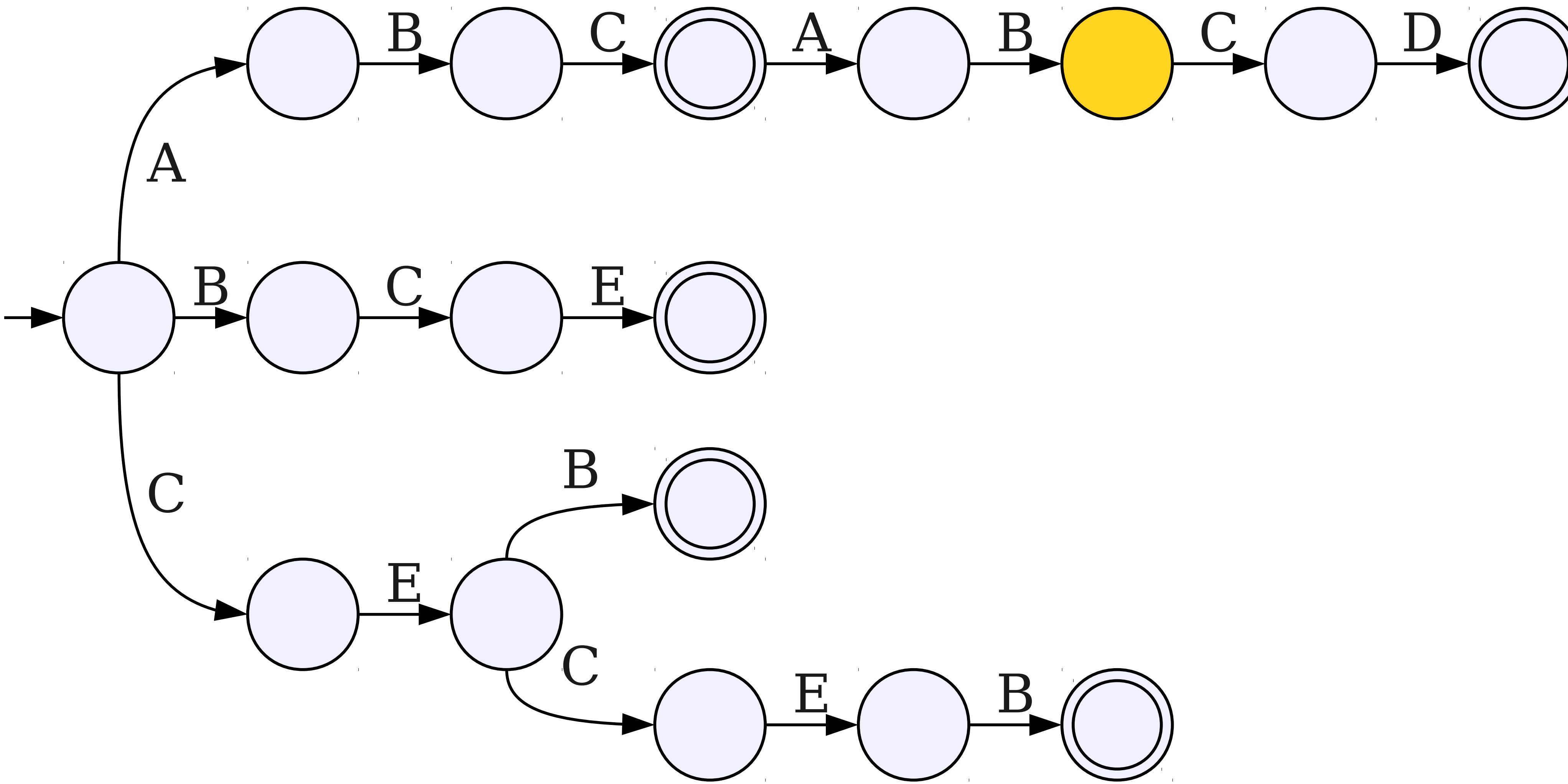
<u>A B C</u> A B C A

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



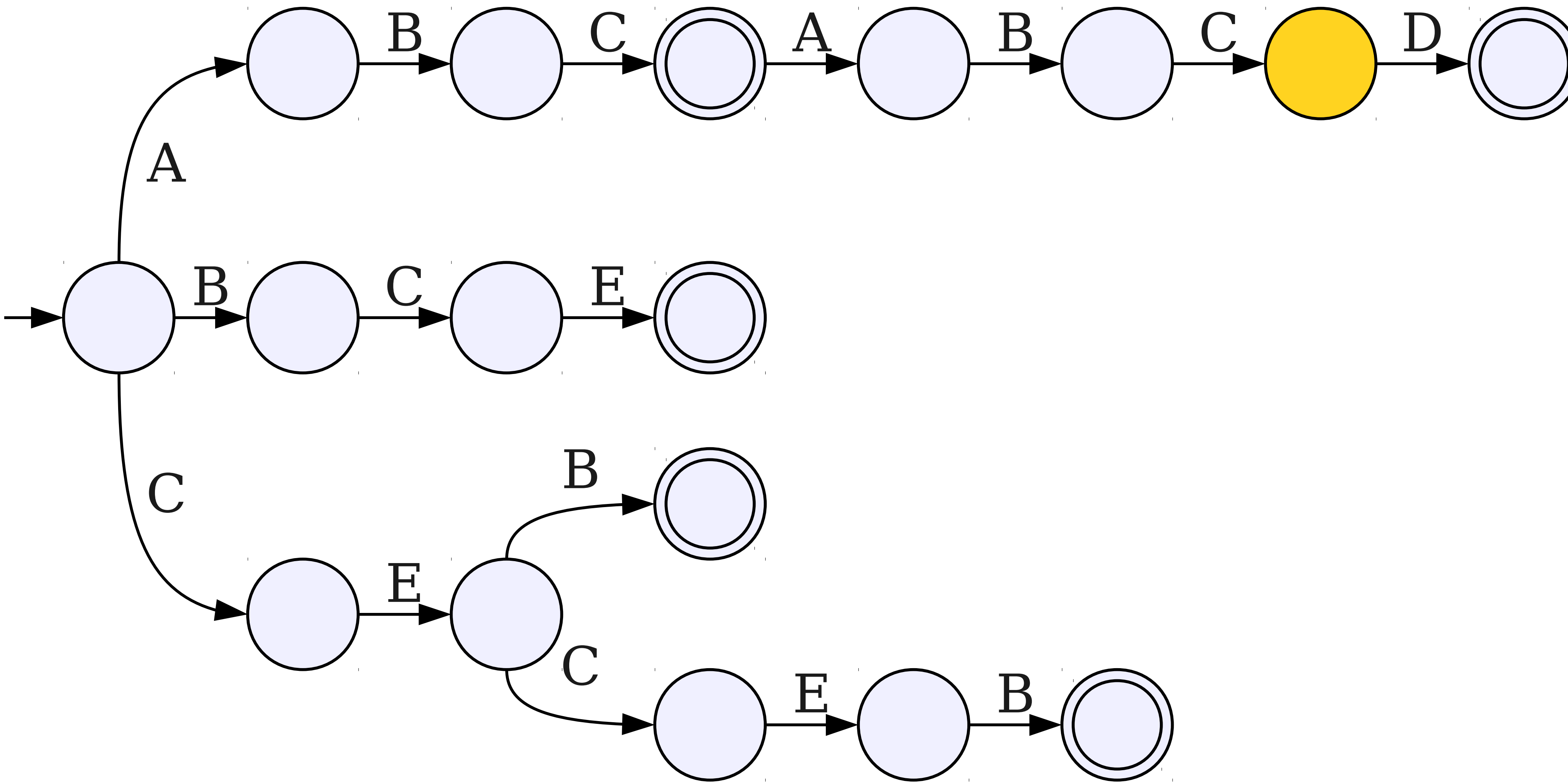
A B C A B C A

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



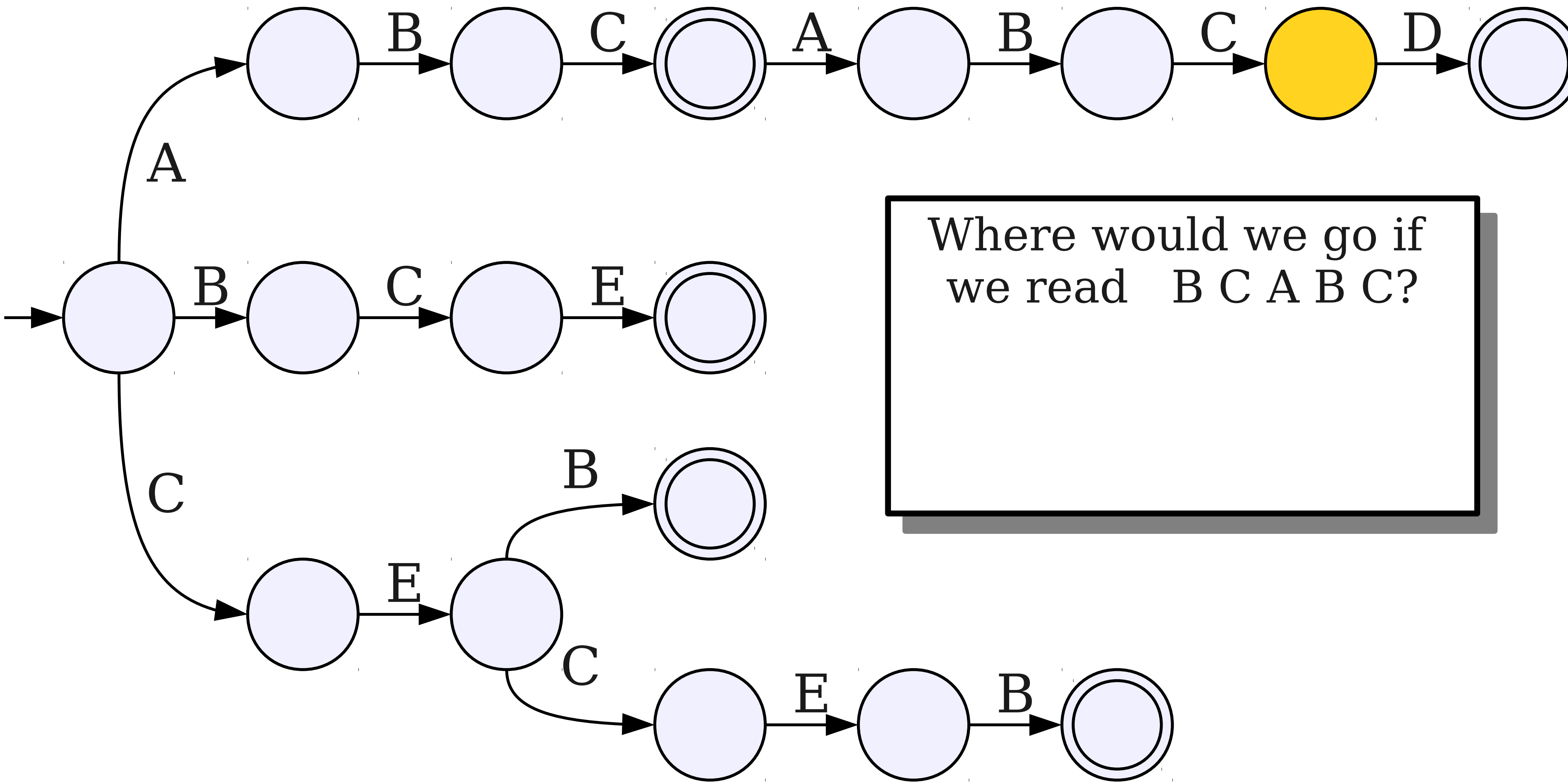
<u>A B C</u> A B C A

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



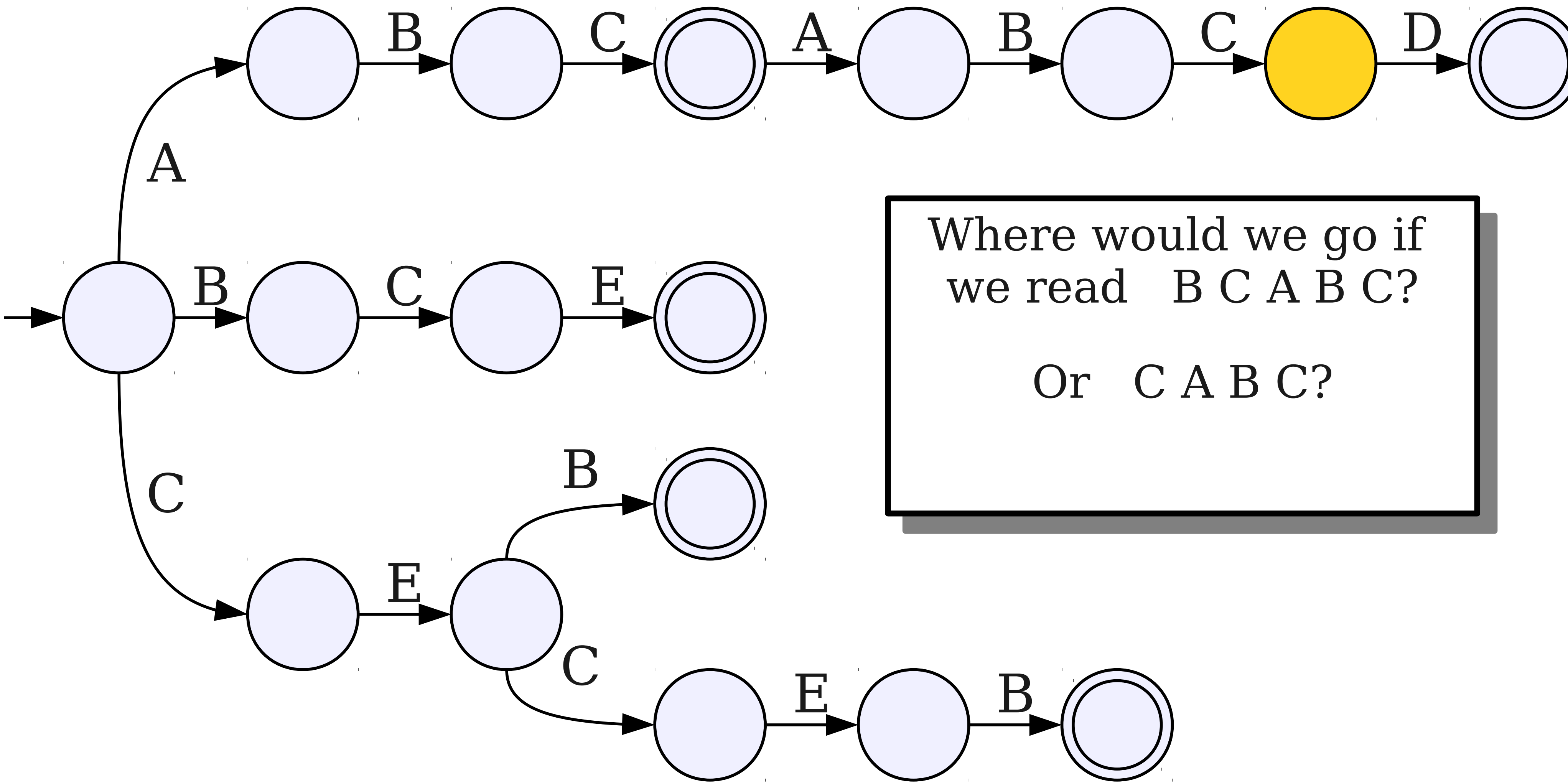
A B C A B C A

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



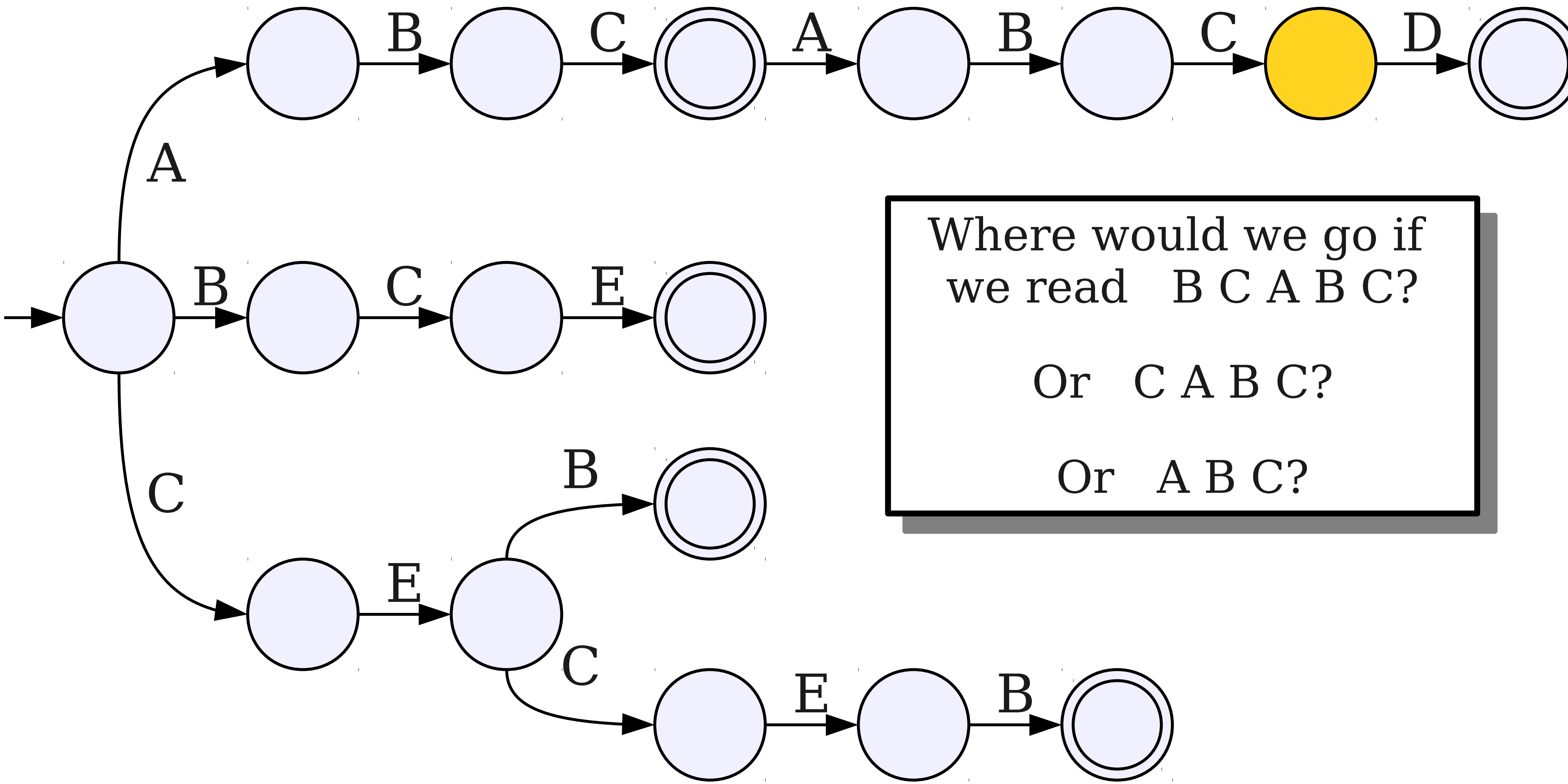
A B C A B C A

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



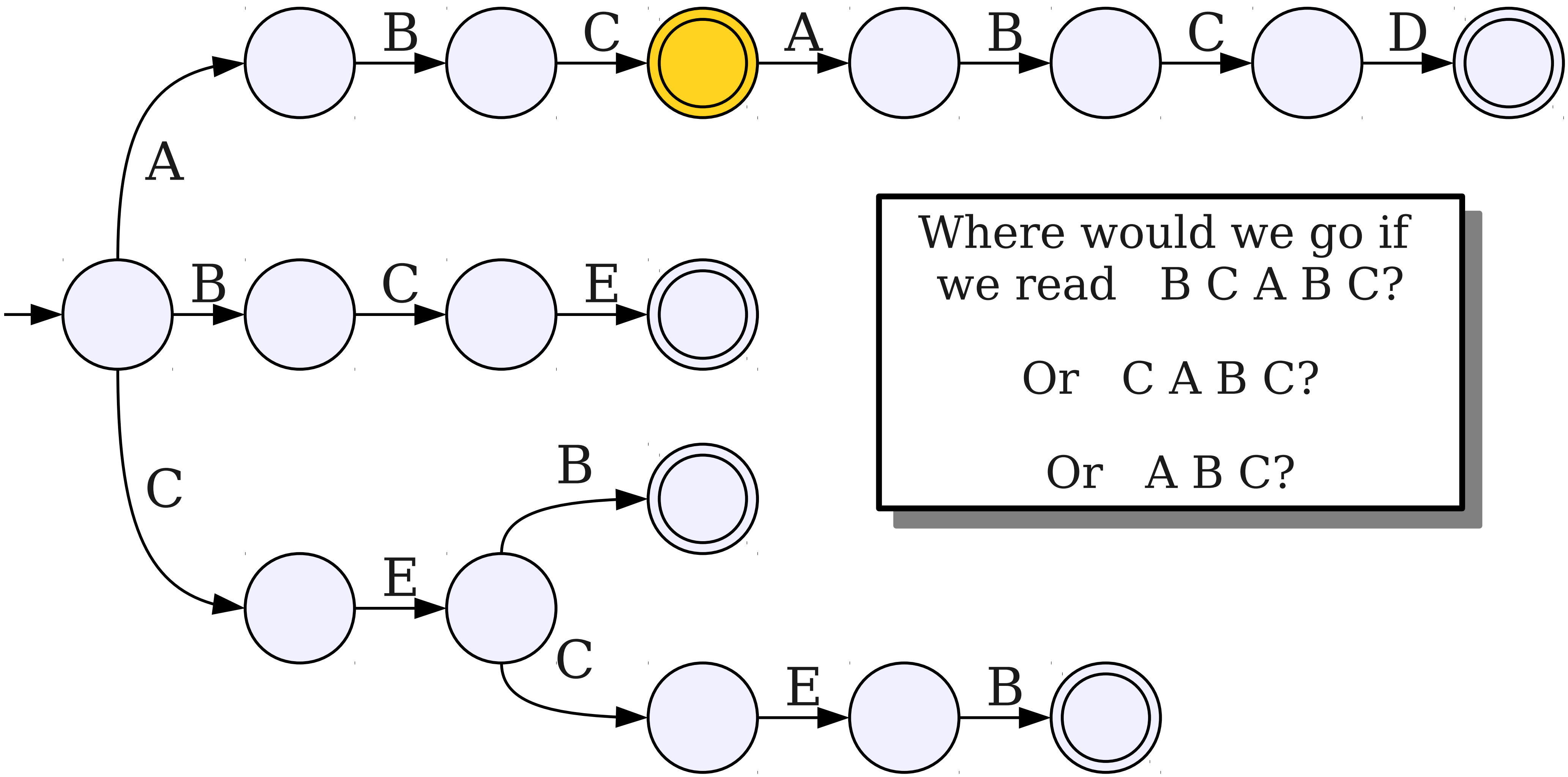
A B C A B C A

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



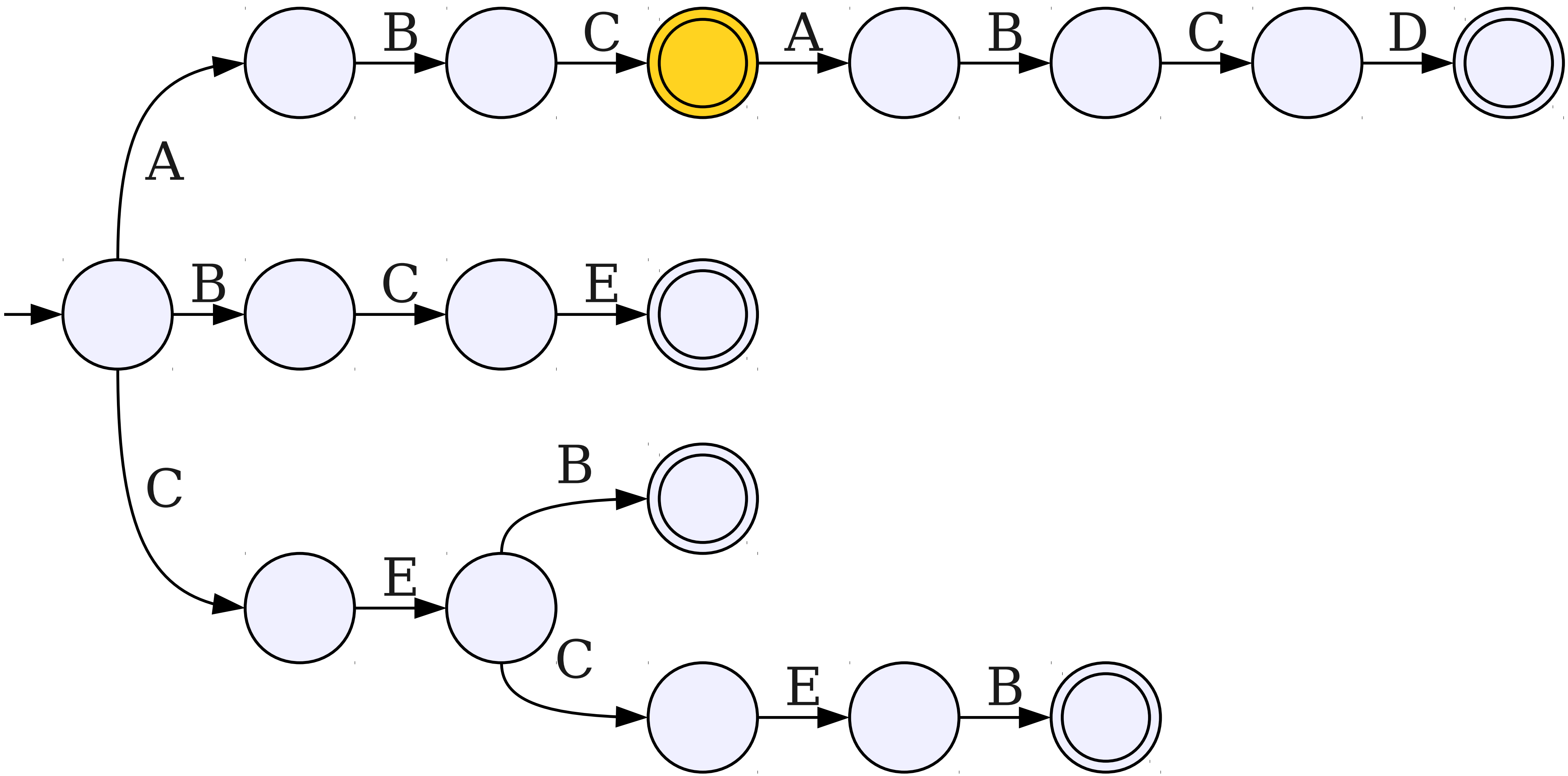
<u>A B C</u> A B C A

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



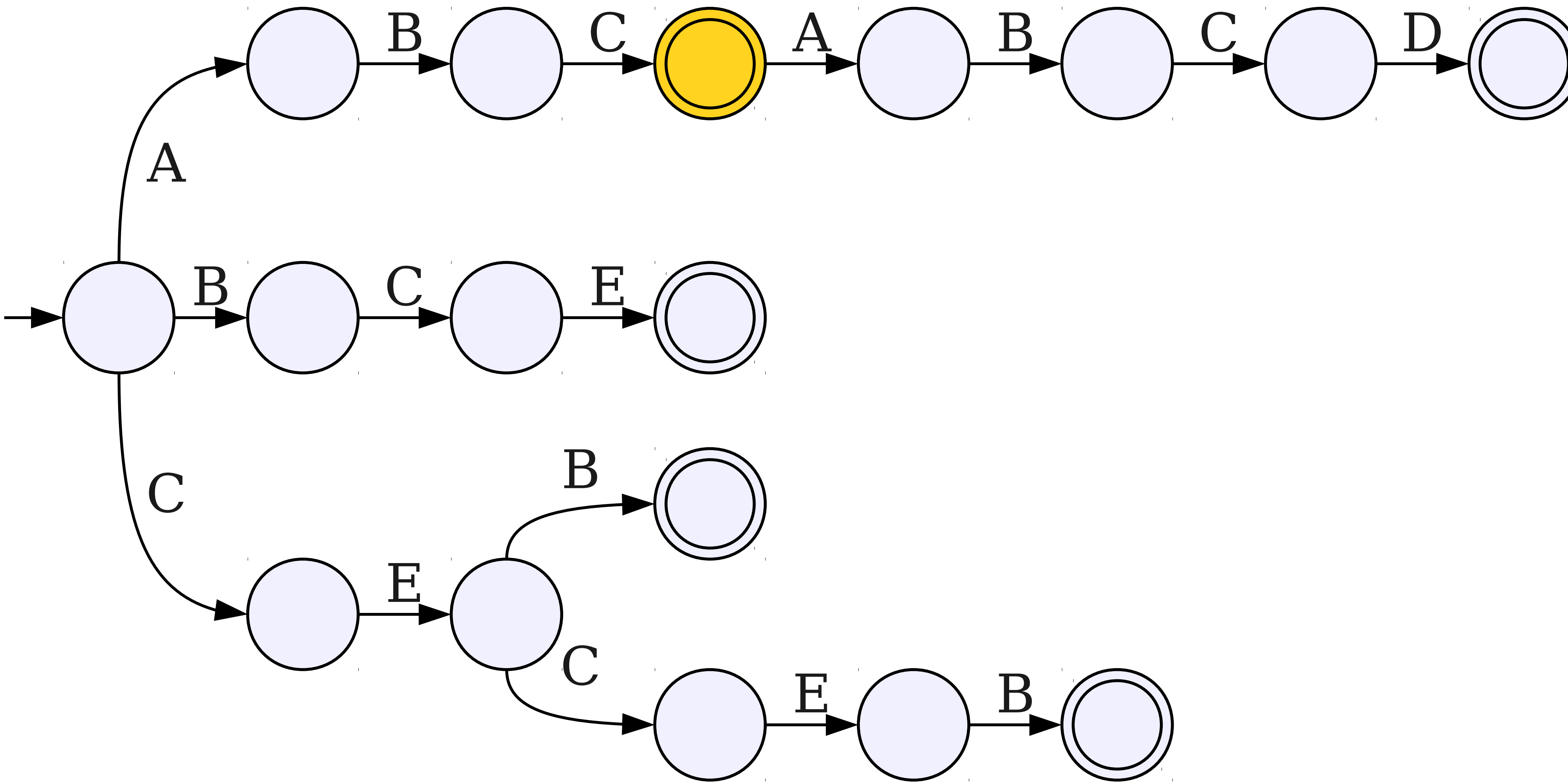
A B C A B C A

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



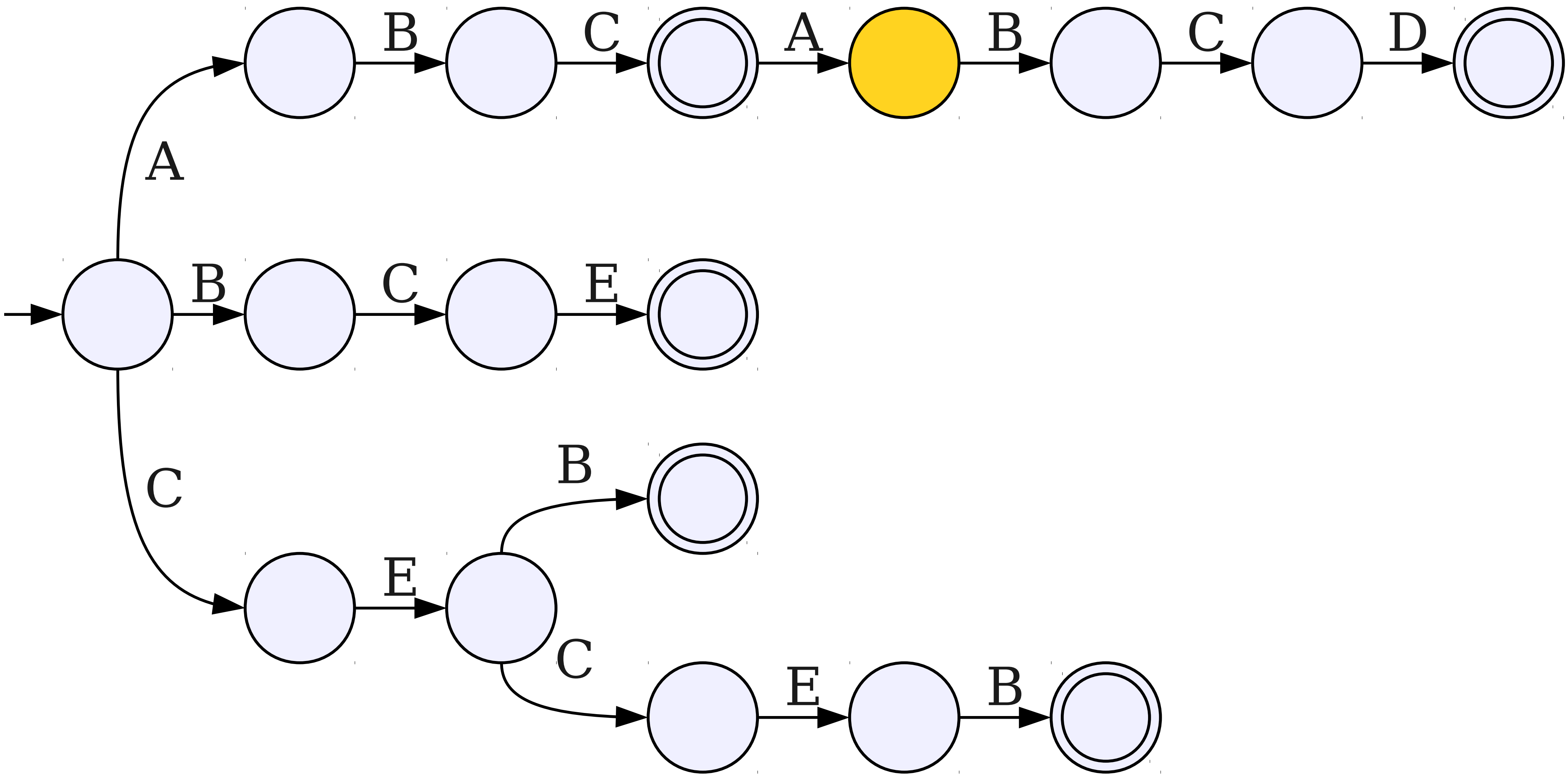
<u>A B C</u> A B C A

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



$\underline{\mathrm{A\ B\ C}}\ \underline{\mathrm{A\ B\ C}}\ \mathrm{A}$

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



A B C A B C A

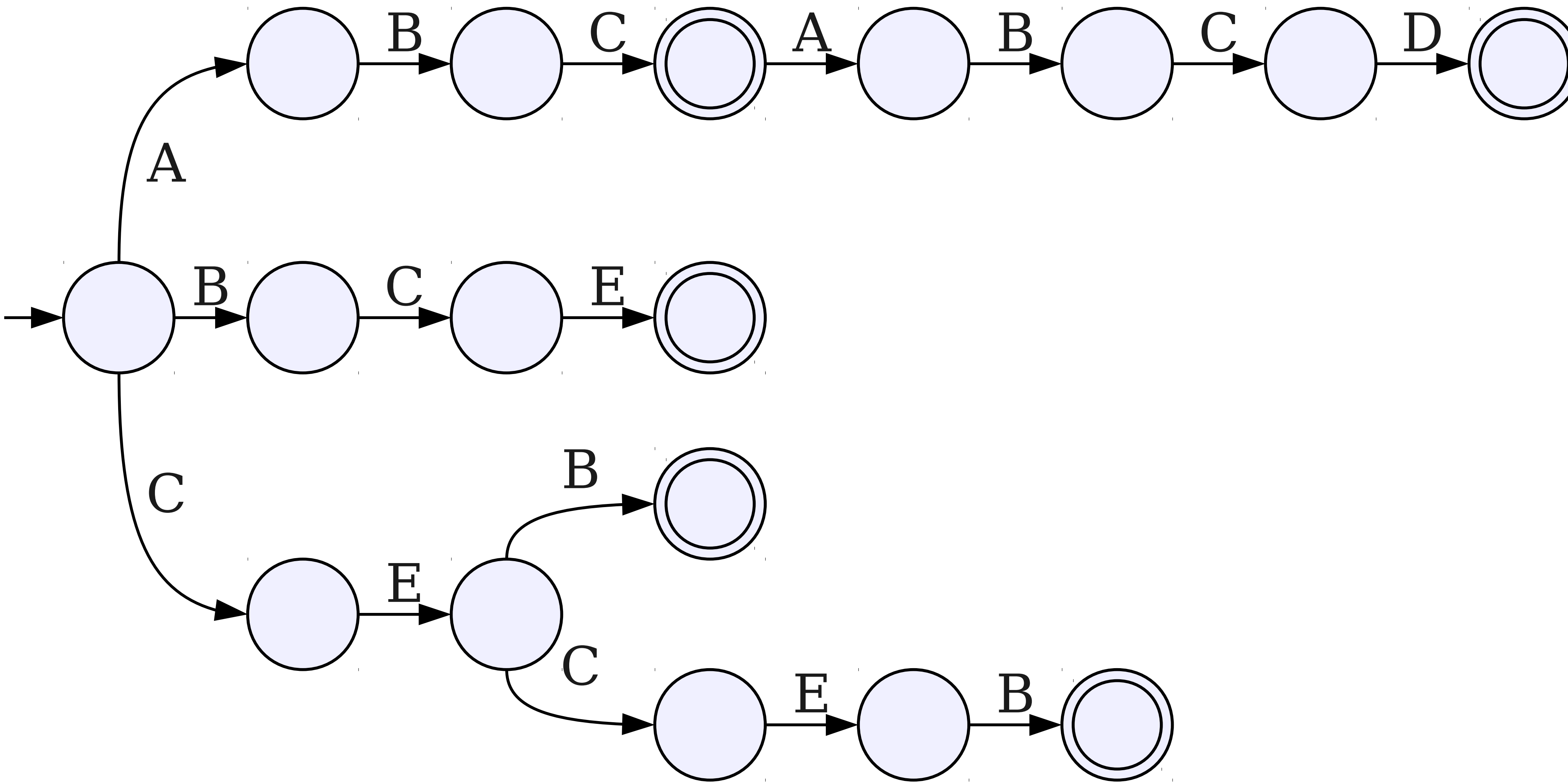
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC

The Idea

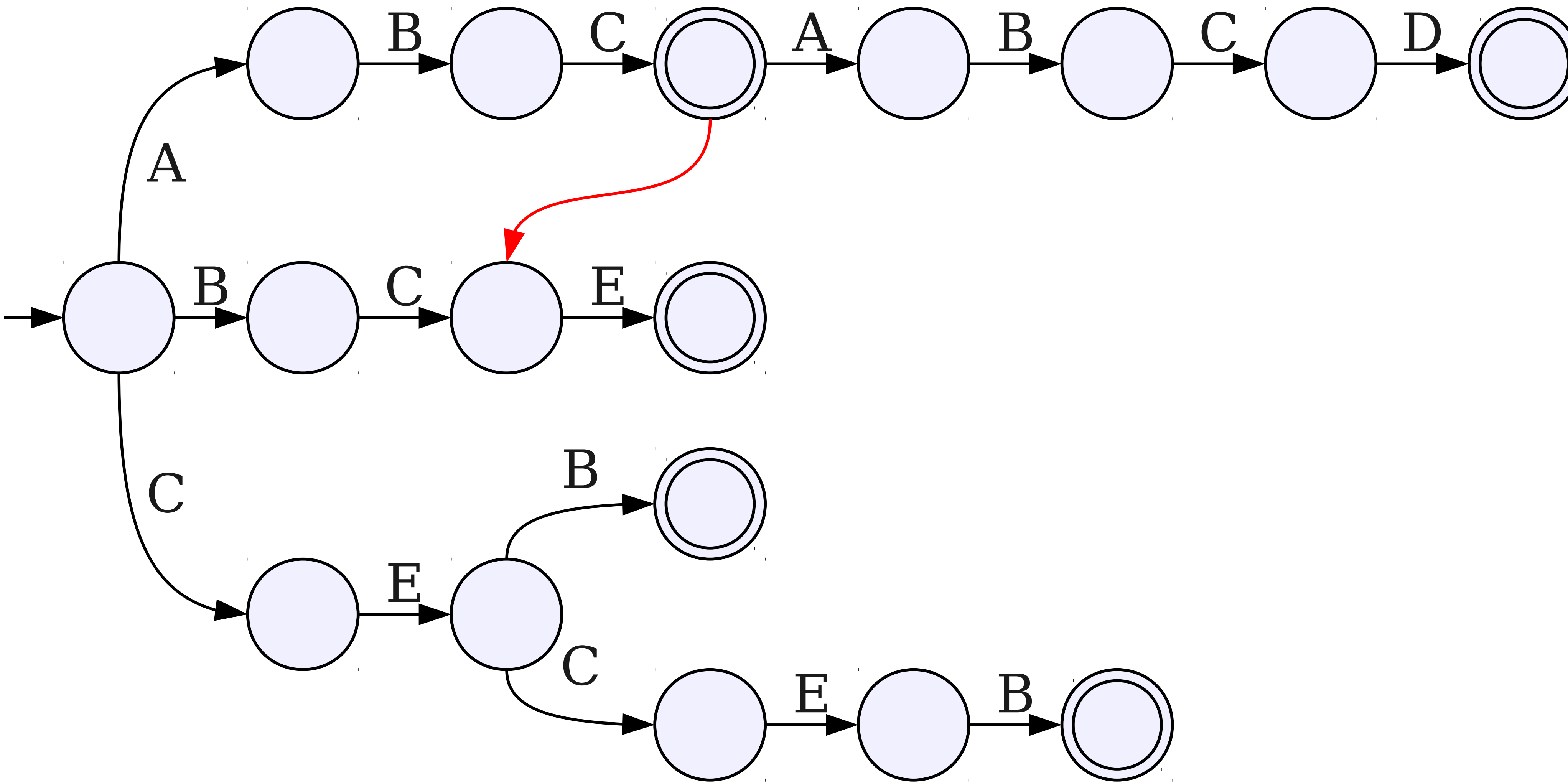
- Suppose we have descended into the trie via string w.
- When we cannot proceed, we want to jump to the node corresponding to the longest proper suffix of w.
- **Claim:** The nodes to jump to can be precomputed efficiently.

Suffix Links

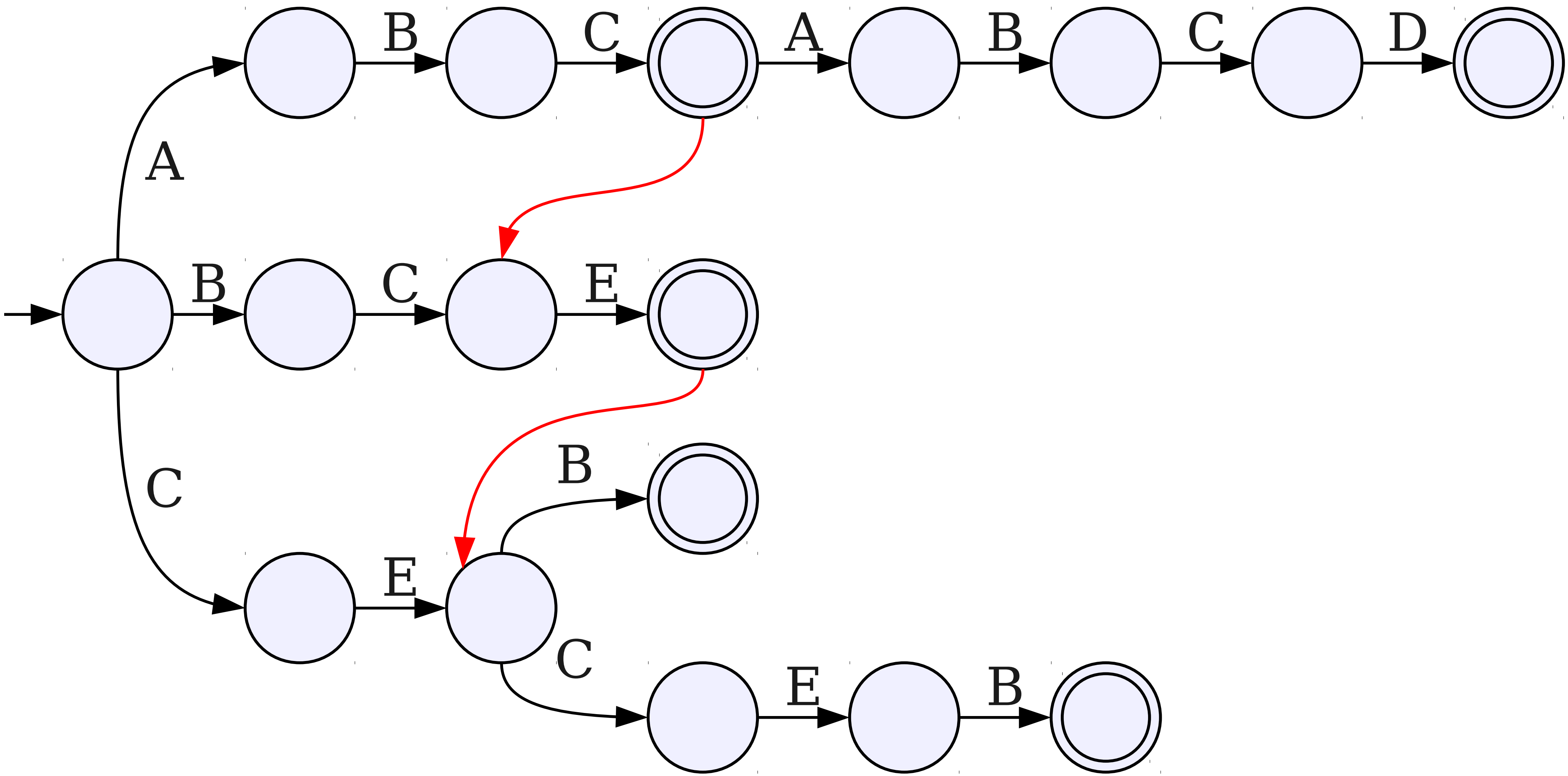
- A **suffix link** in a trie is a pointer from a node for string *w* to the node corresponding to the longest proper suffix of *w*.
- All nodes other than the root node will have a suffix link.



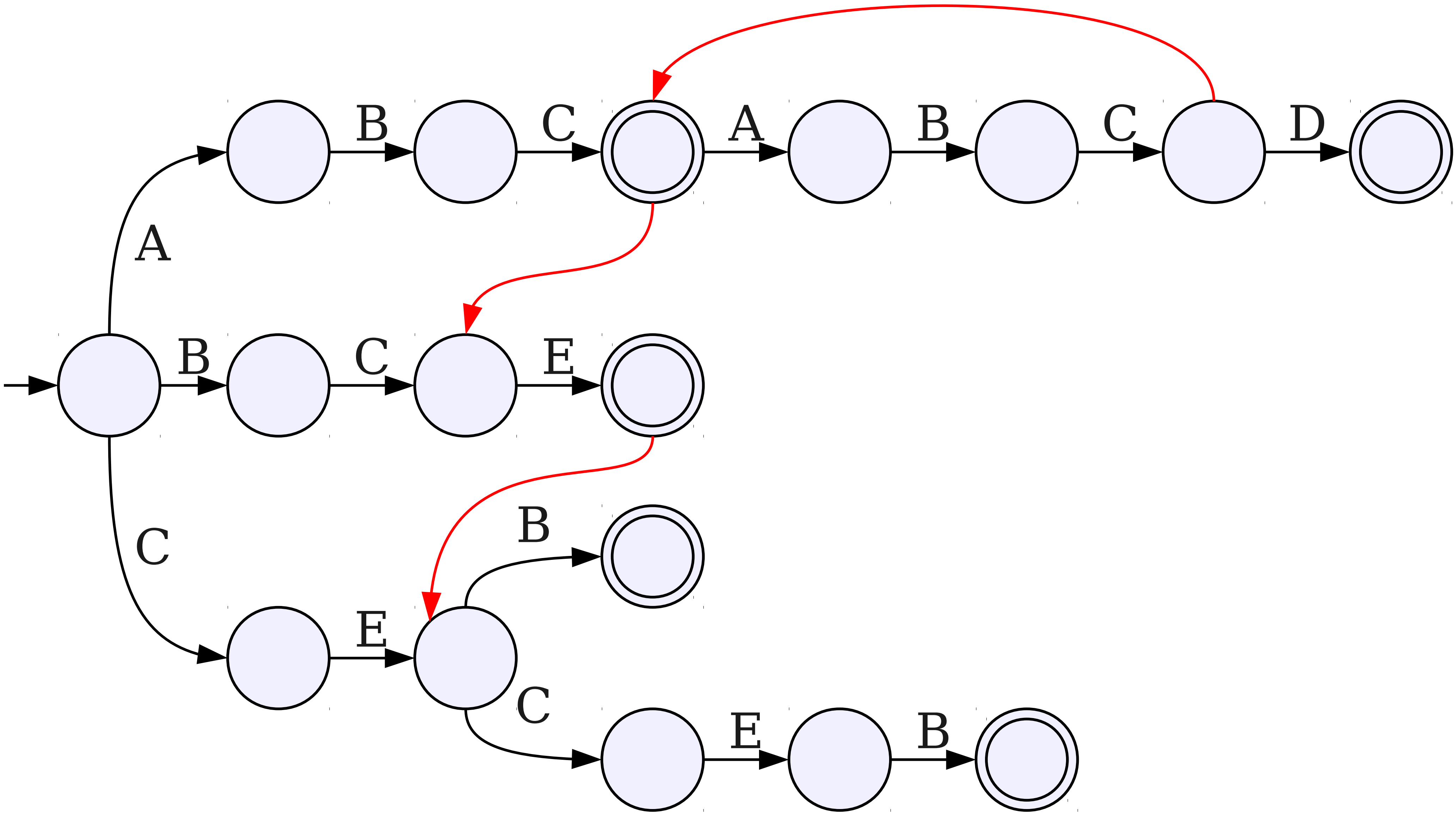
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



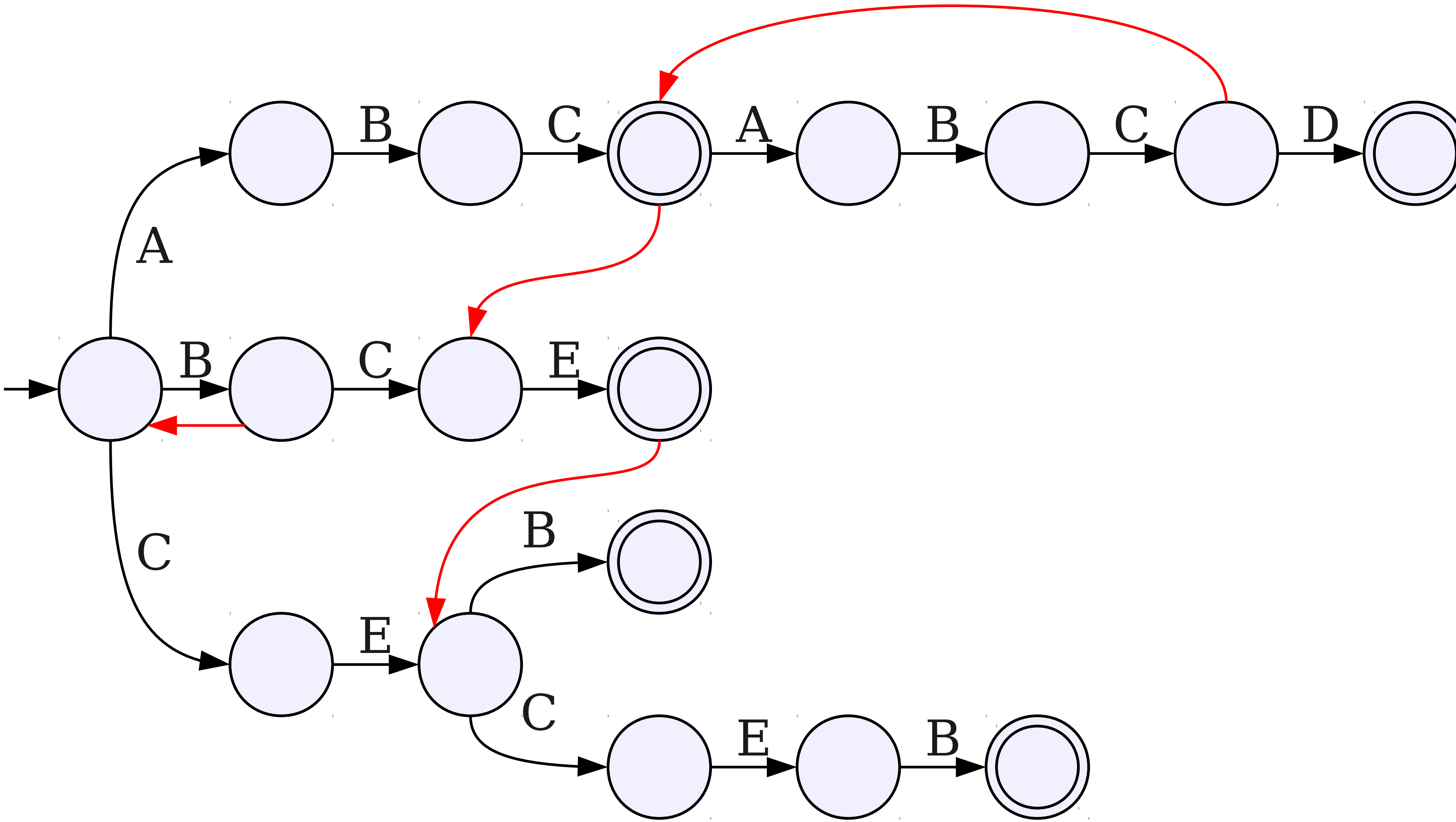
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



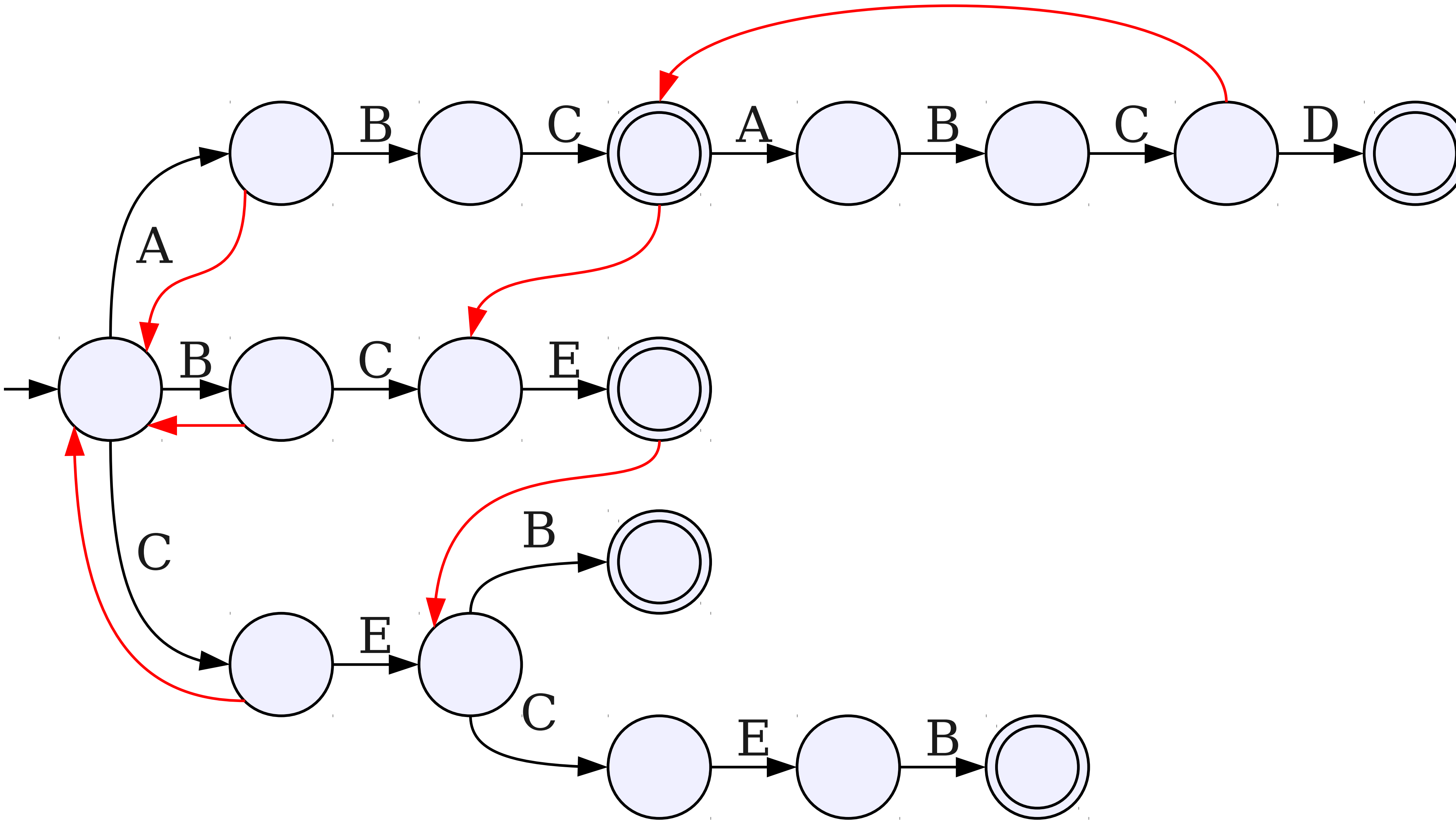
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



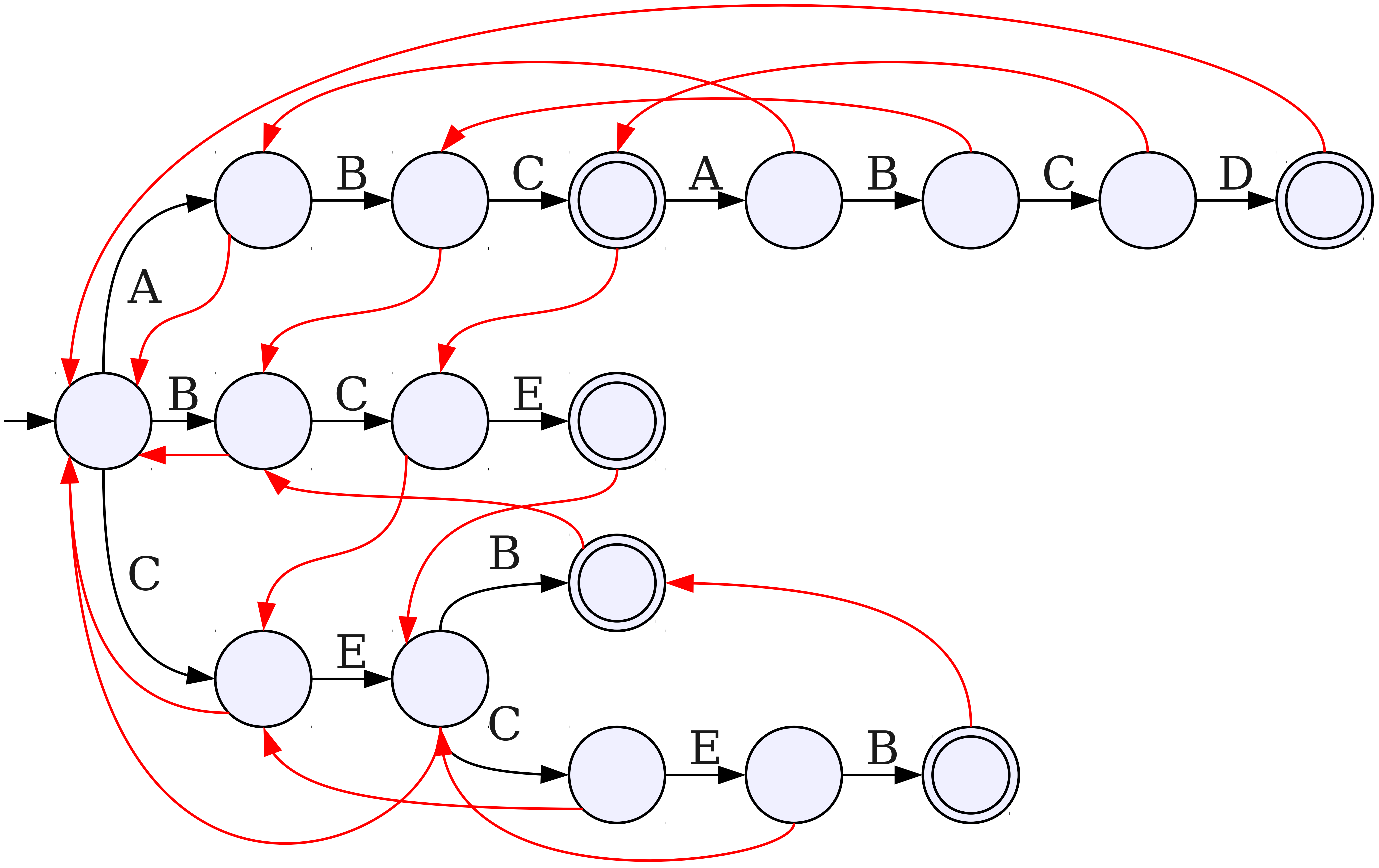
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



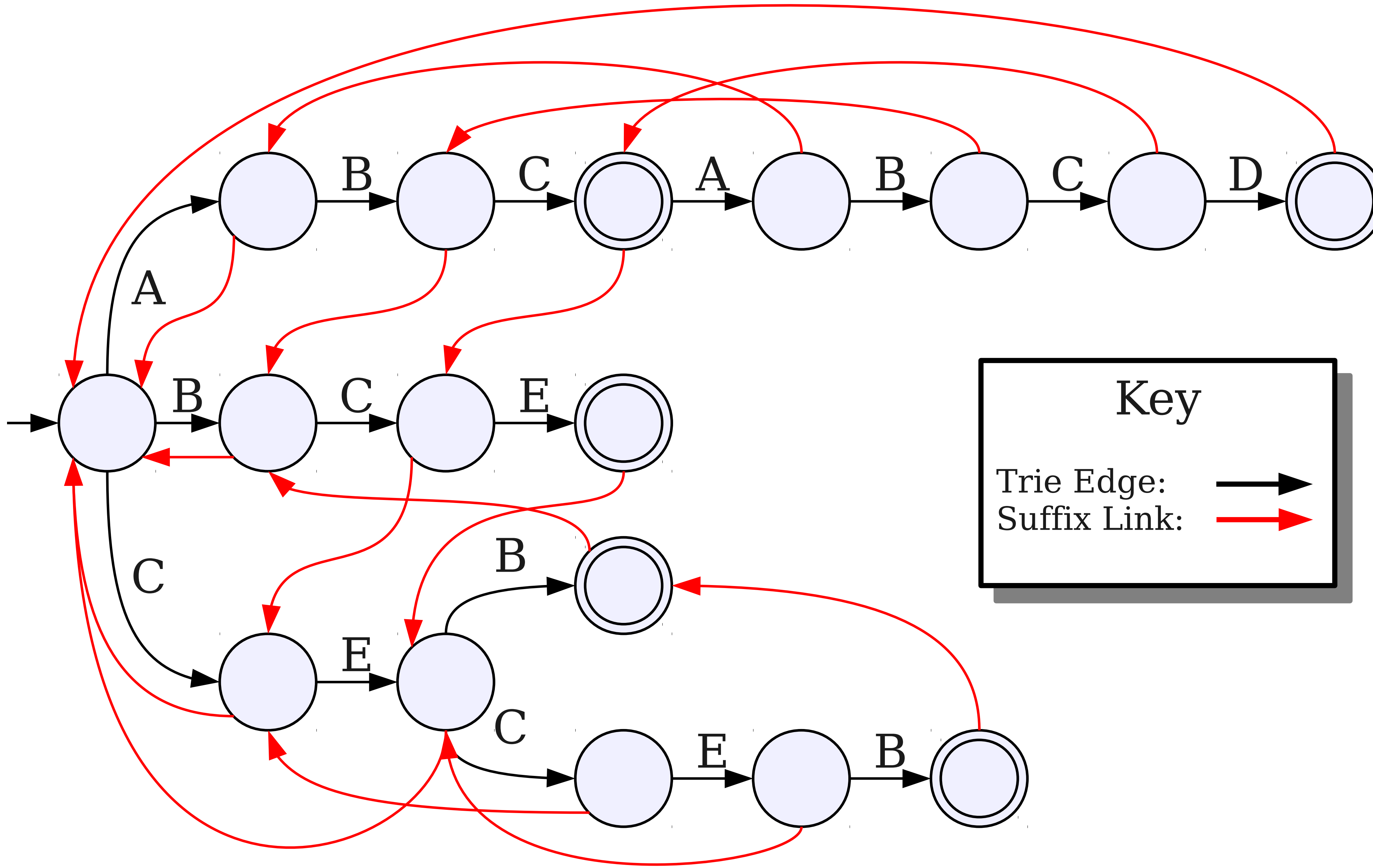
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



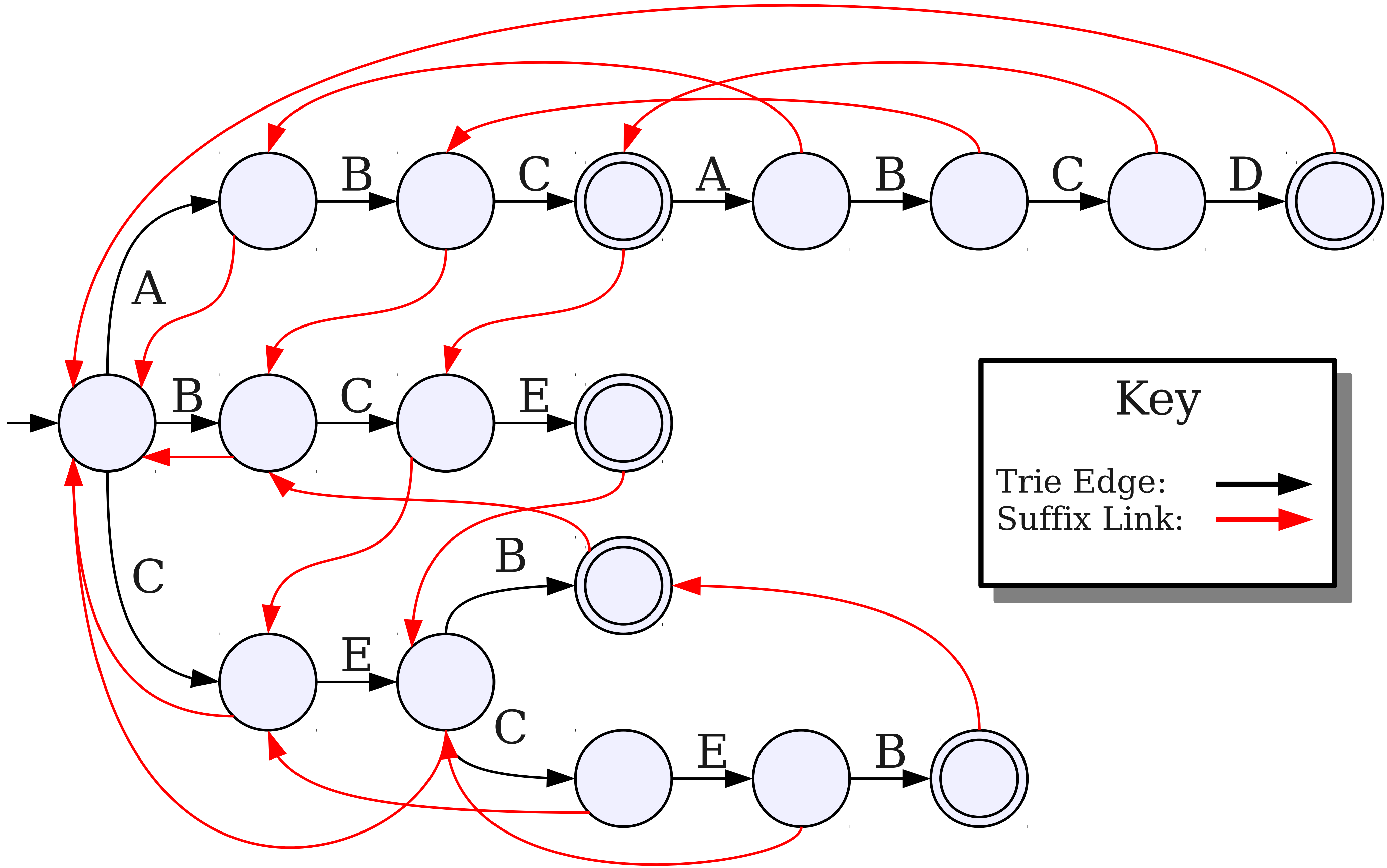
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC

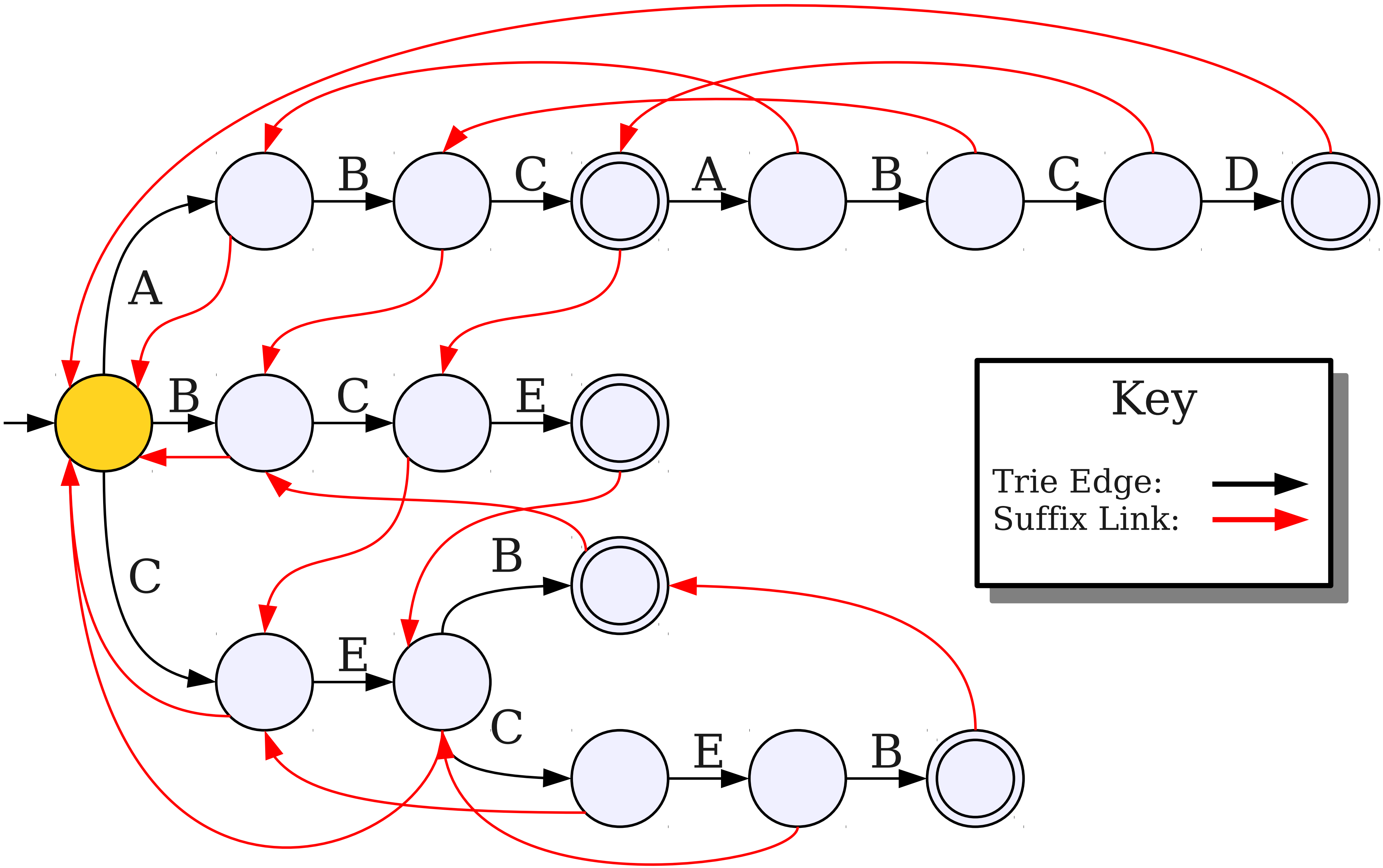


P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



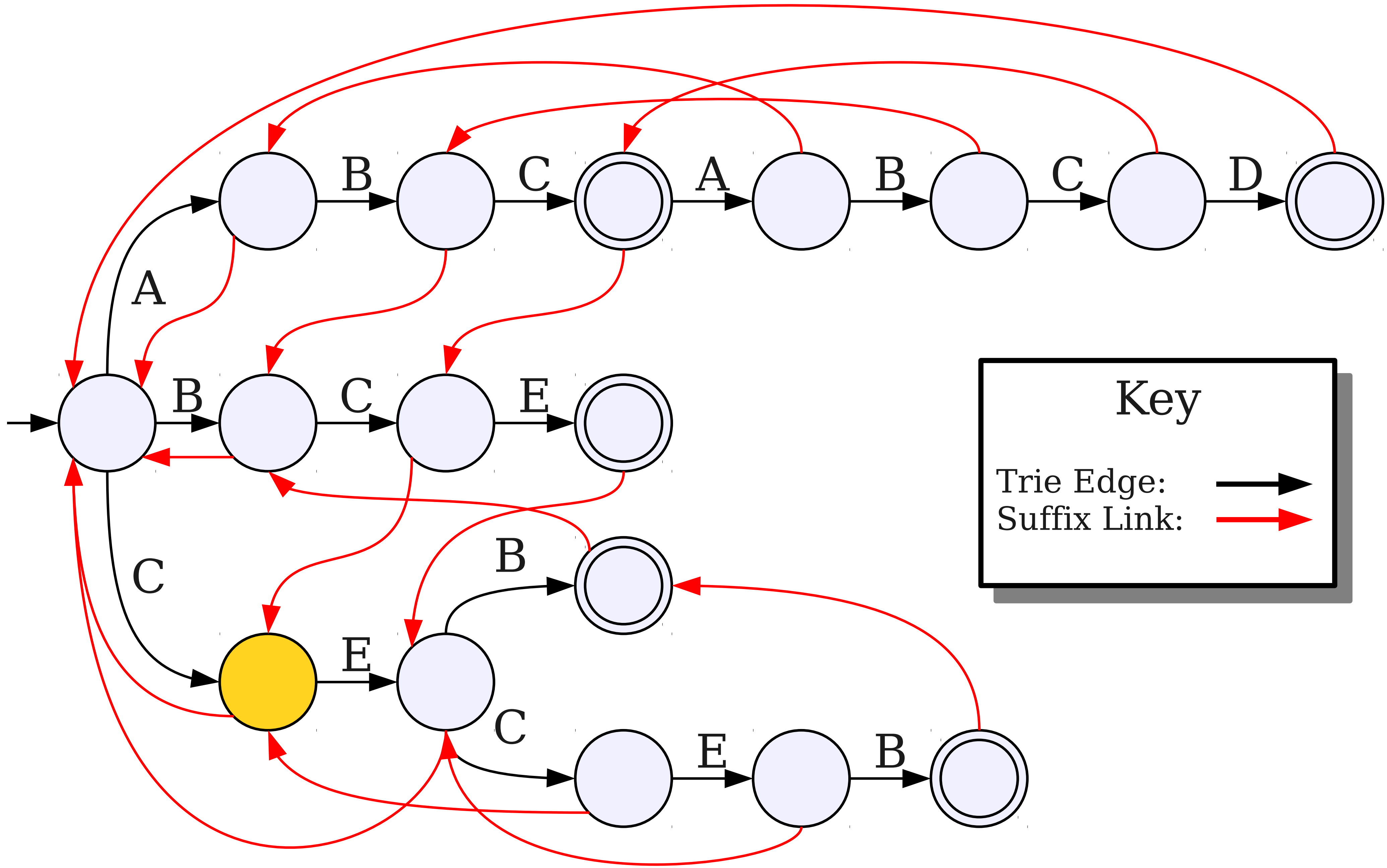
C E C E C E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



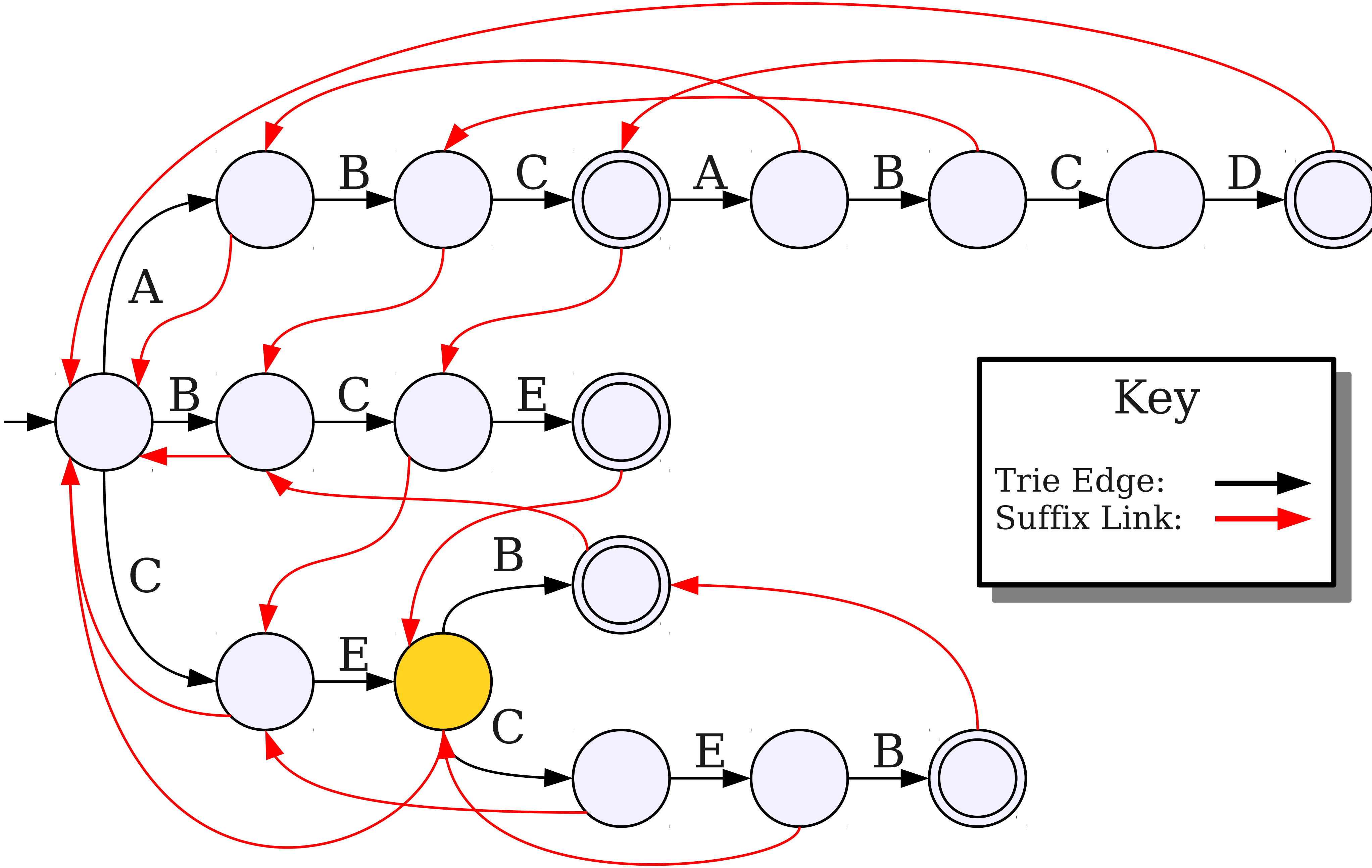
C E C E C E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



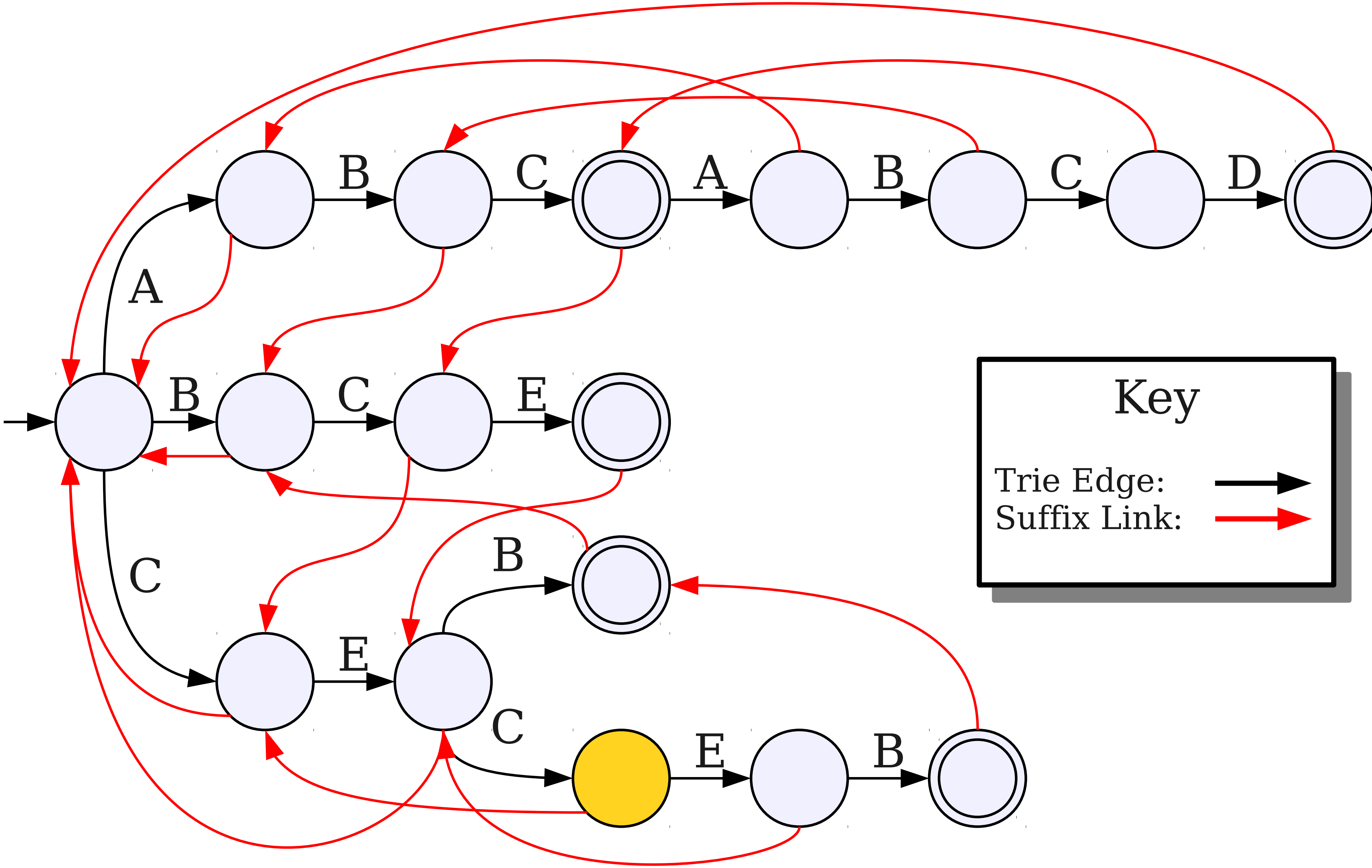
C E C E C E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



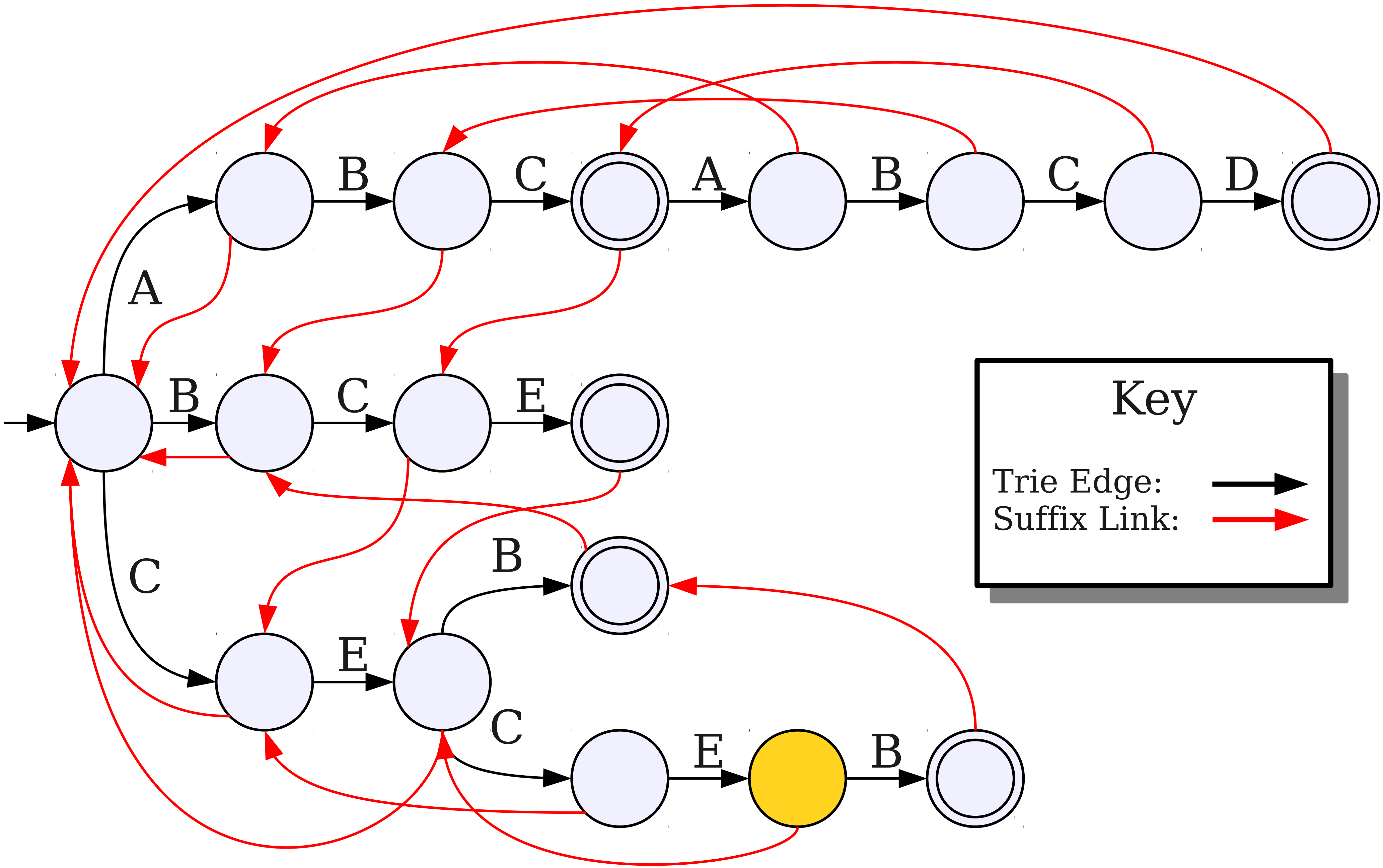
C E C E C E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



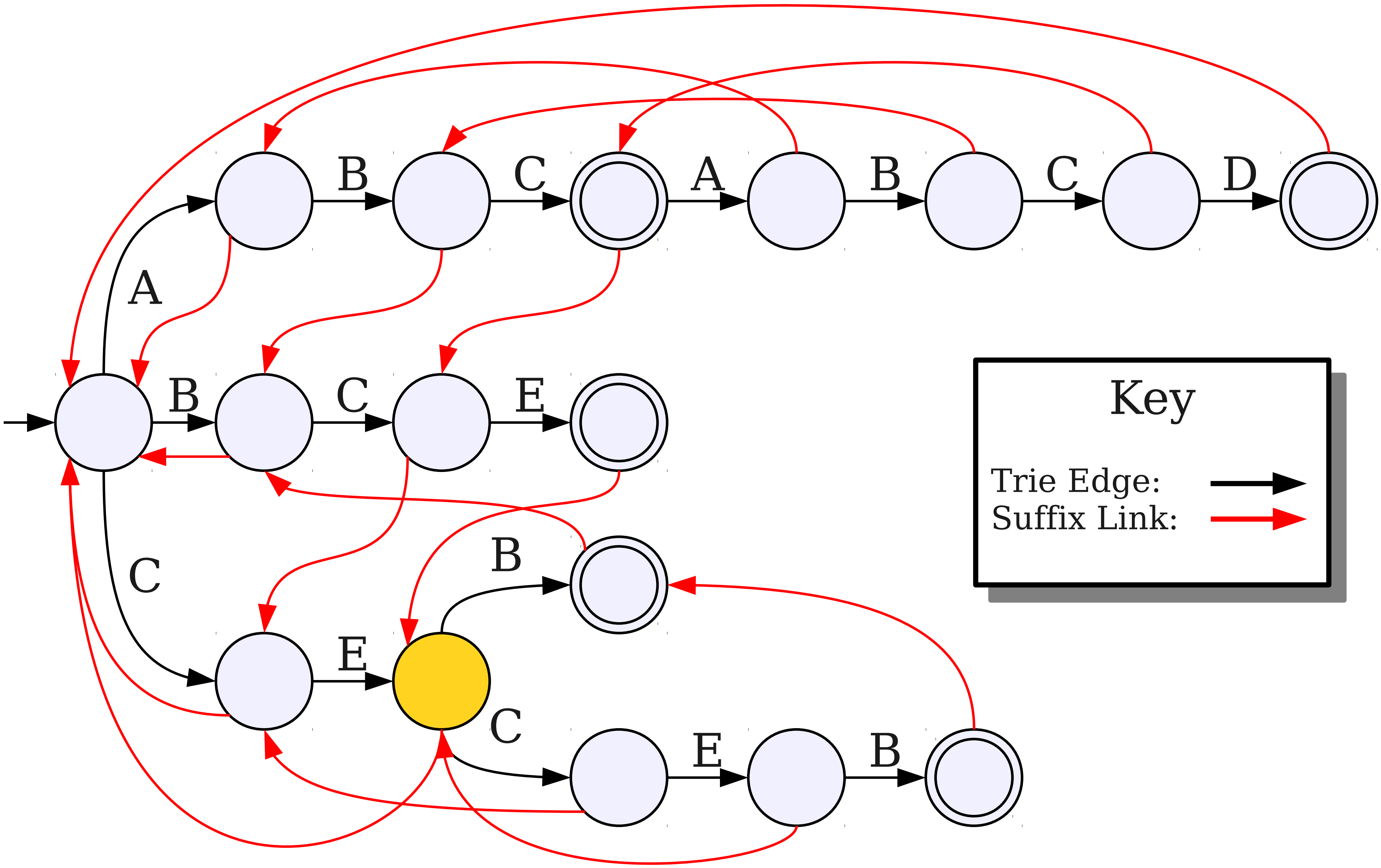
C E C E C E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



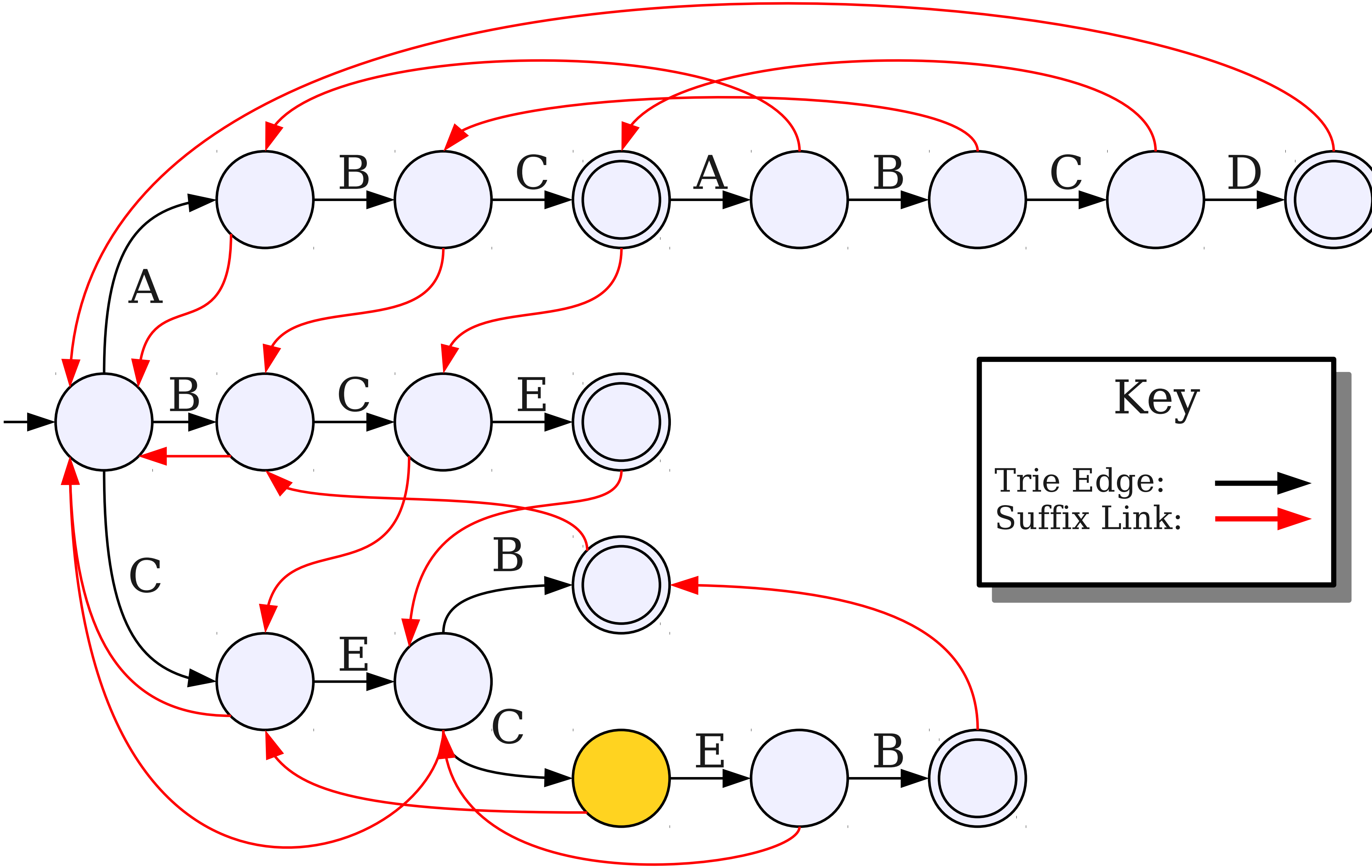
C E C E C E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



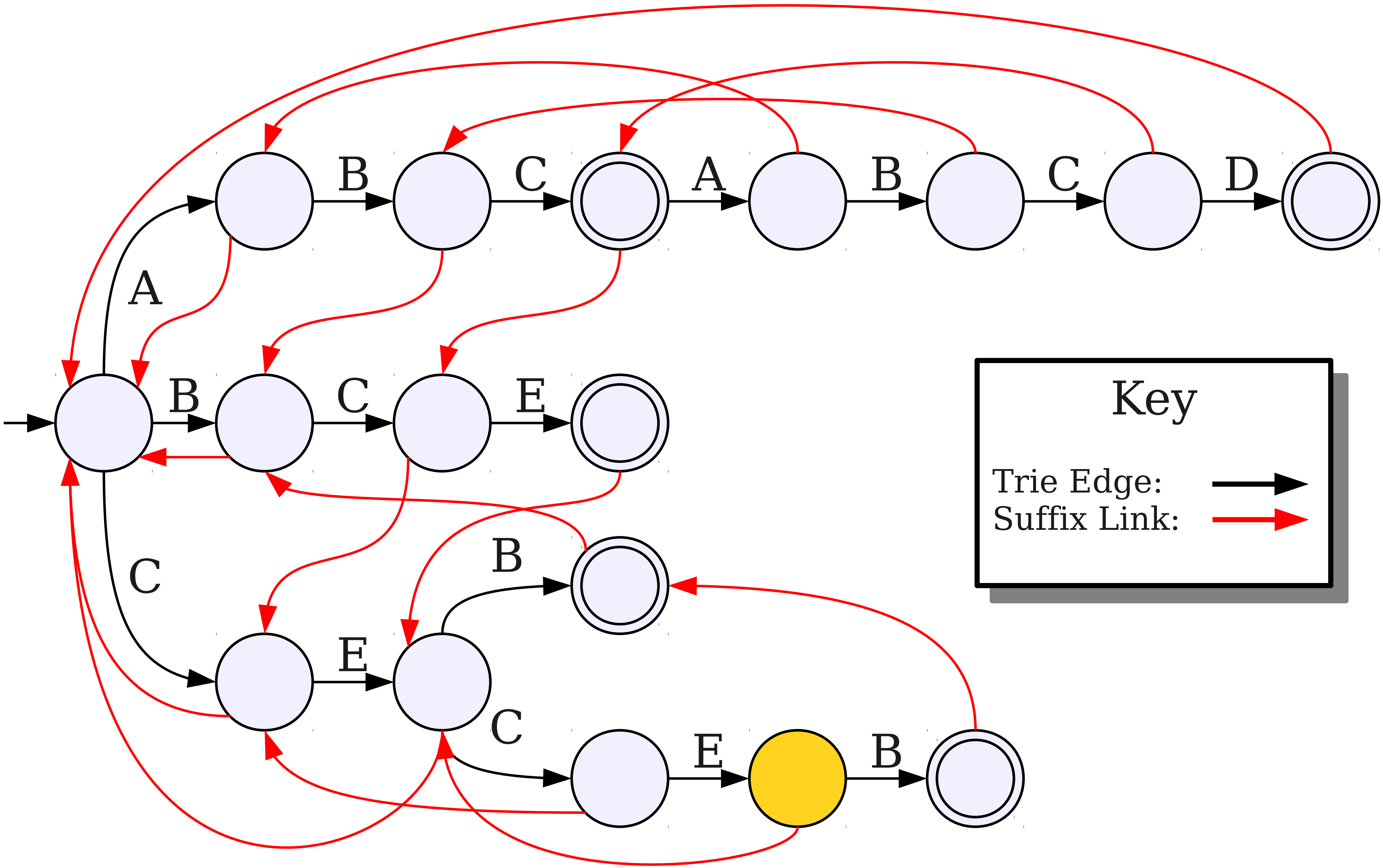
C E C E C E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



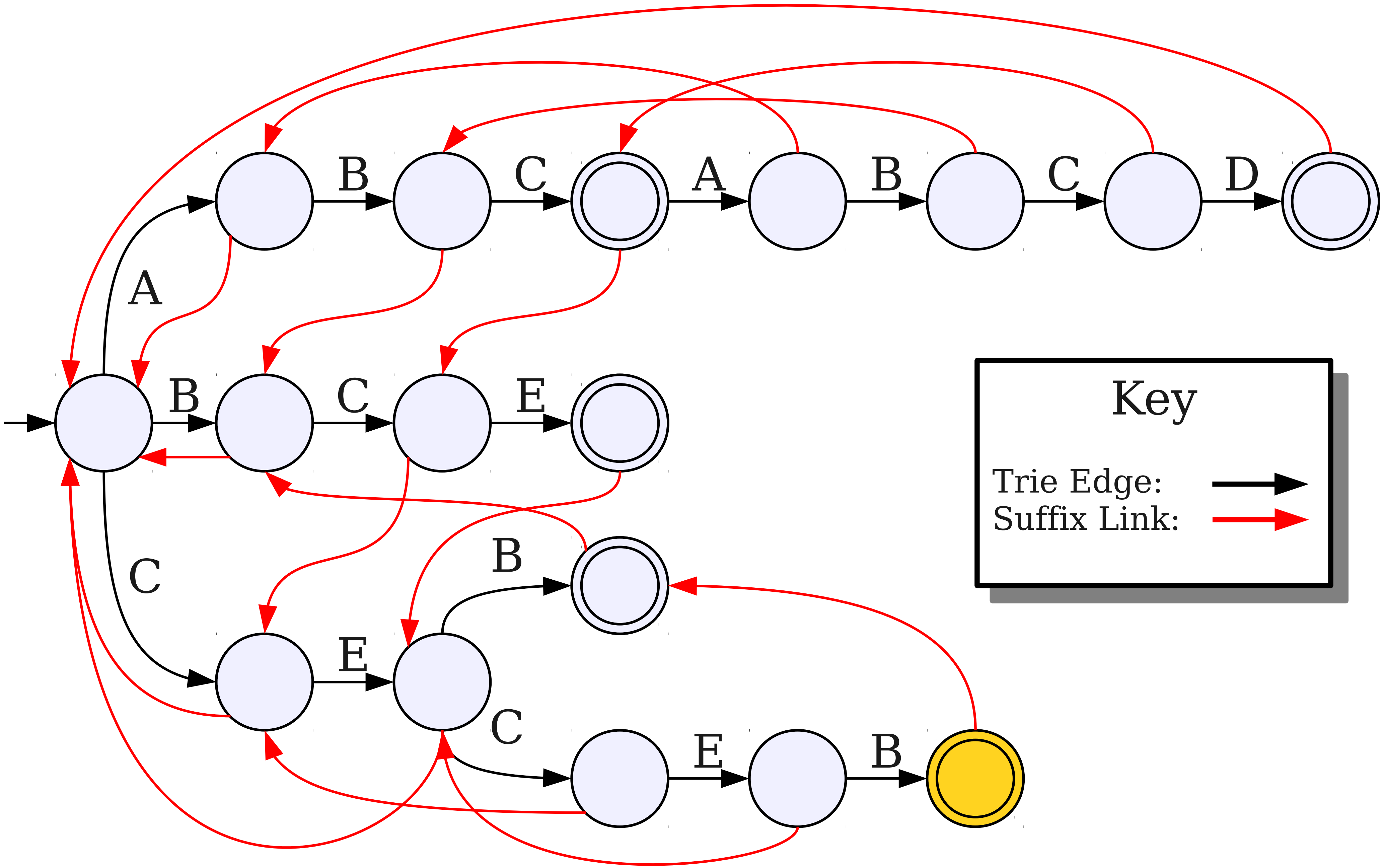
C E C E C E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



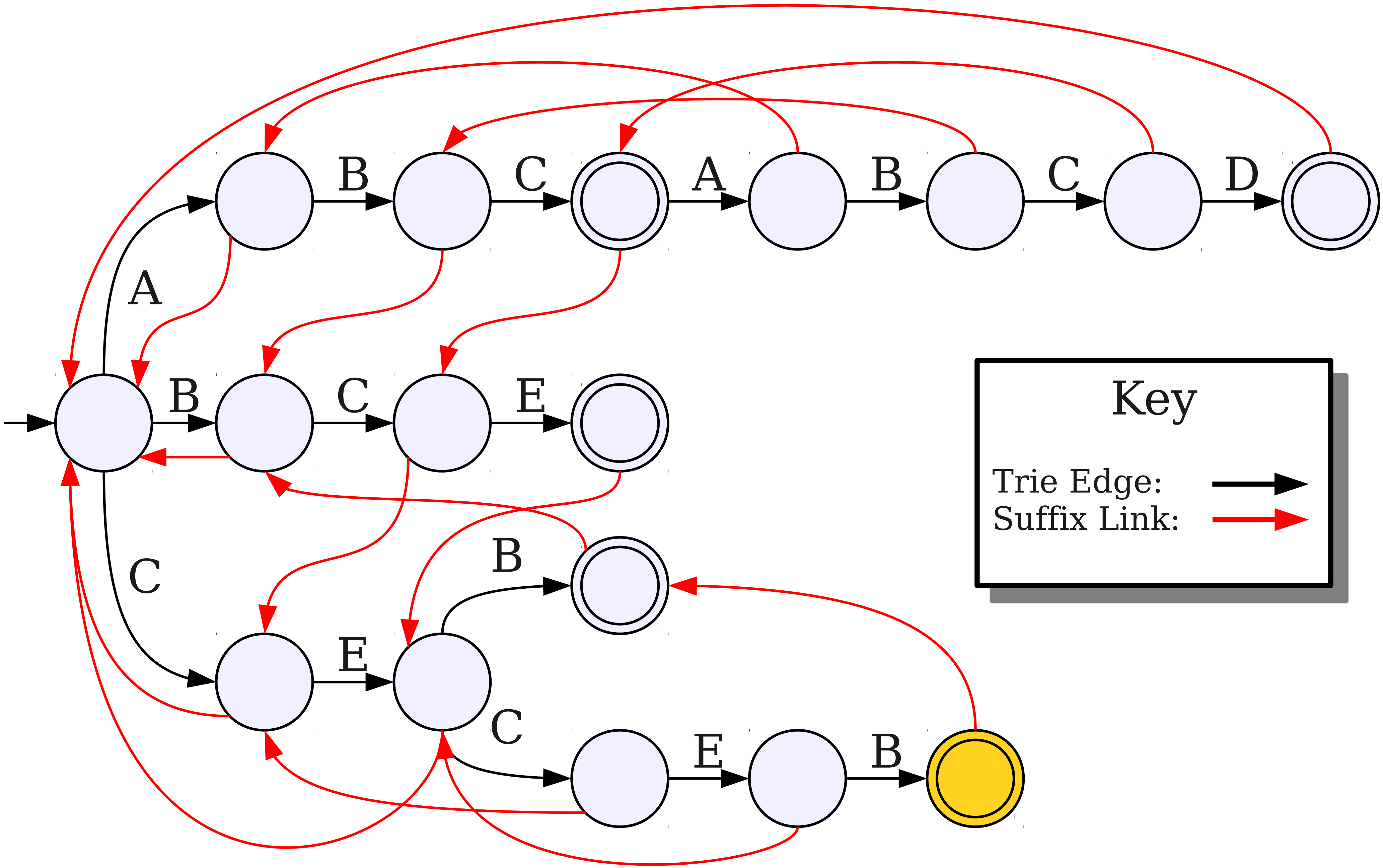
C E C E C E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



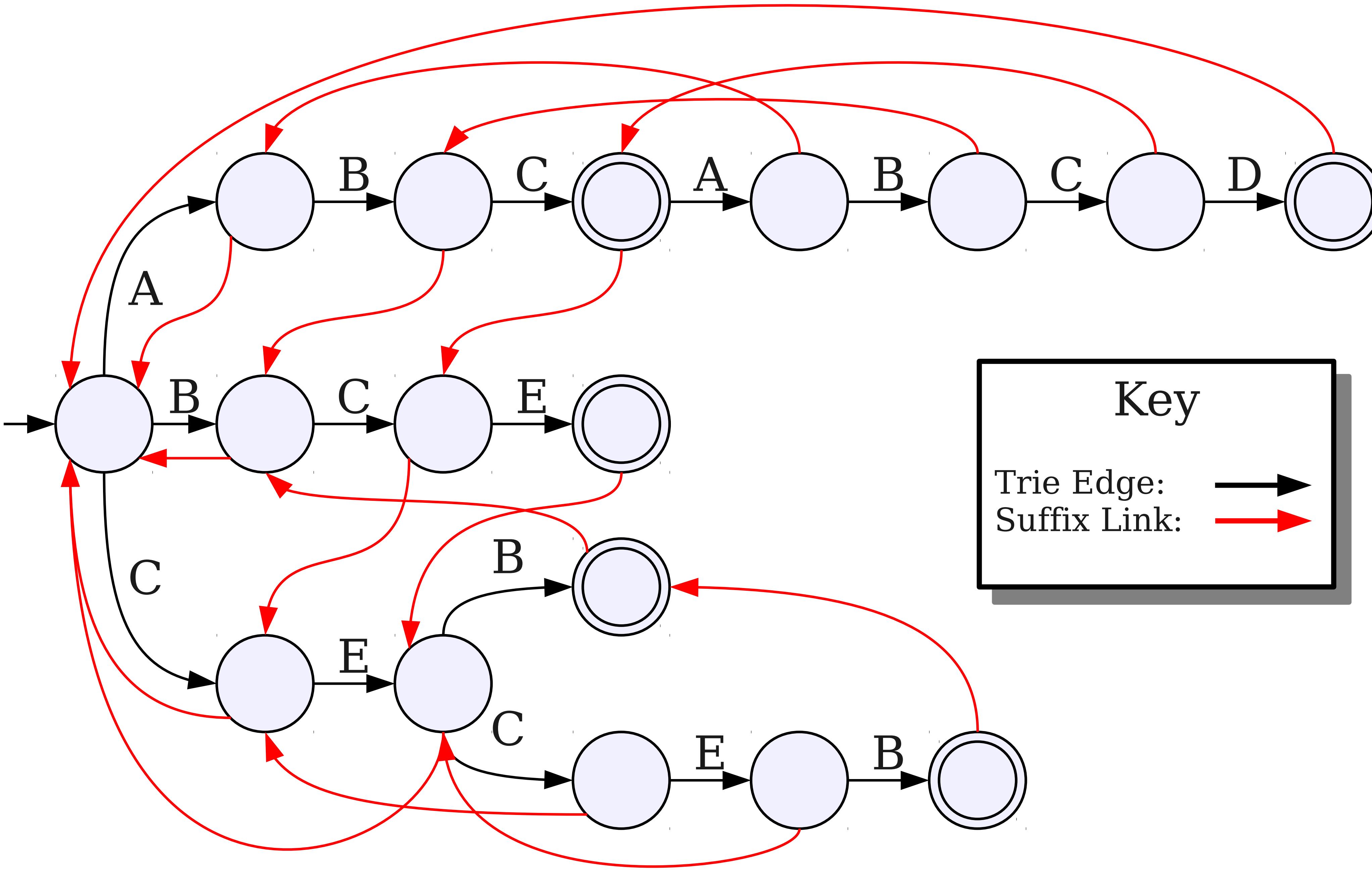
C E C E C E B

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC

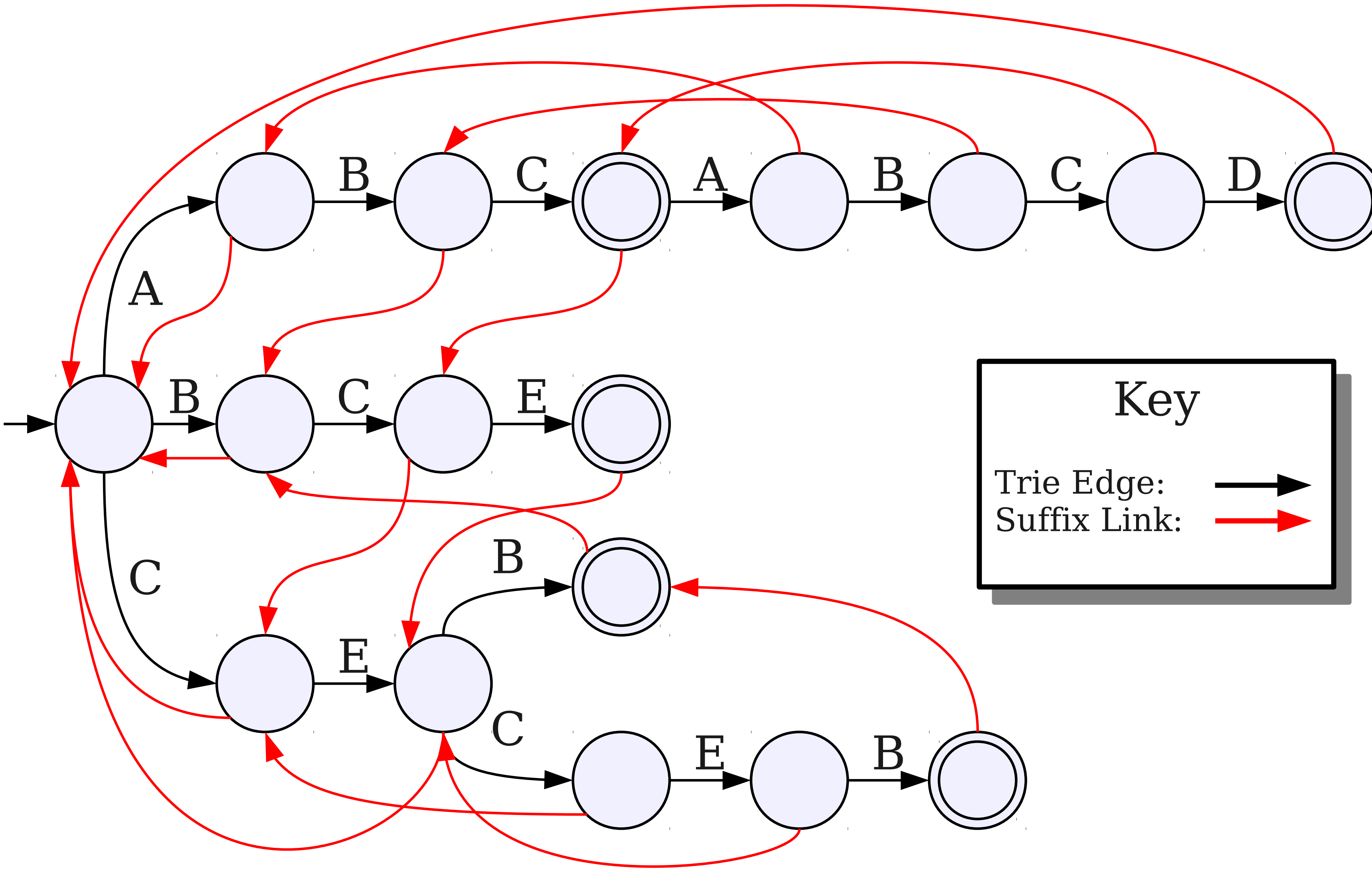


C E <u>C E C E B</u>

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC

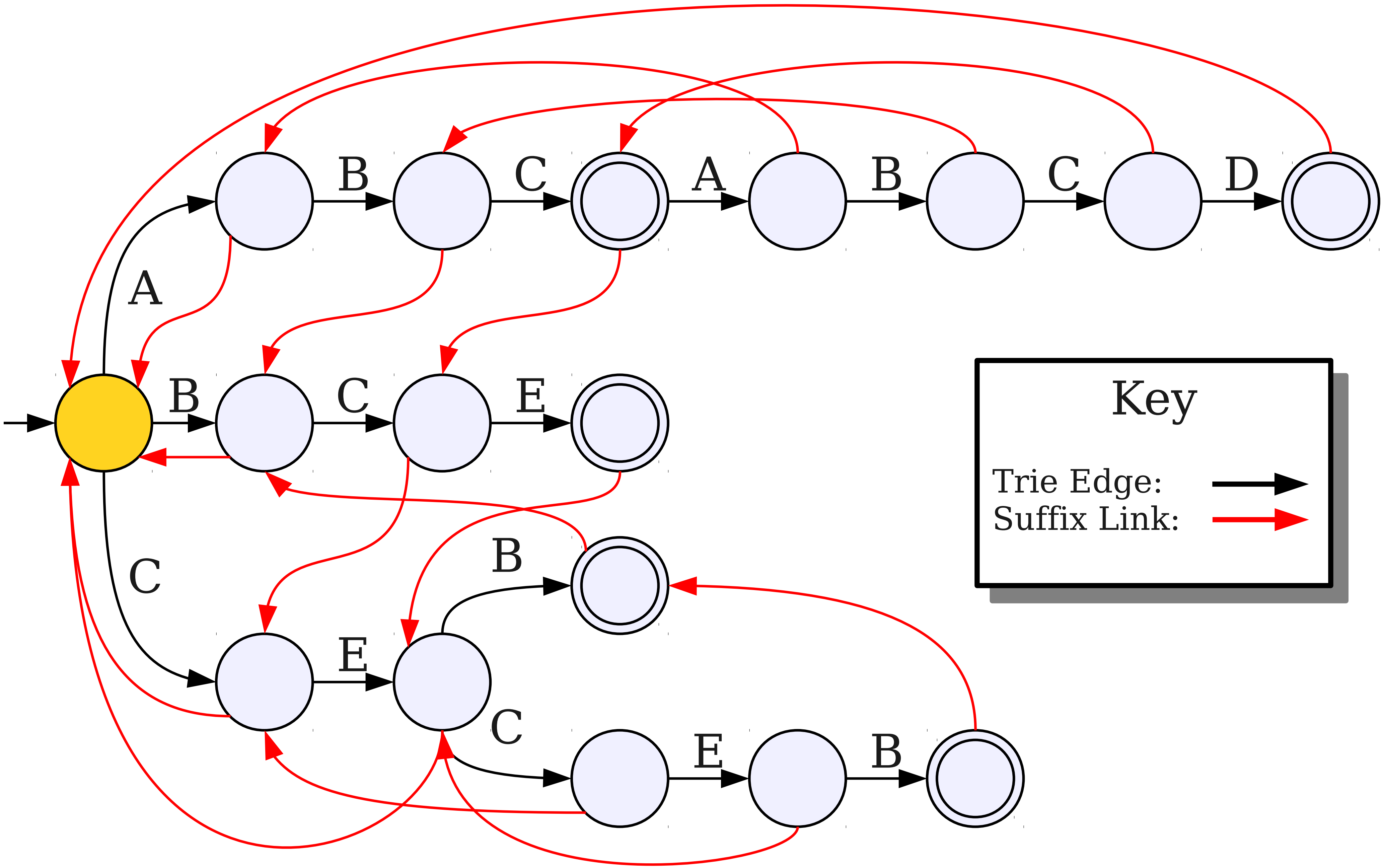


P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



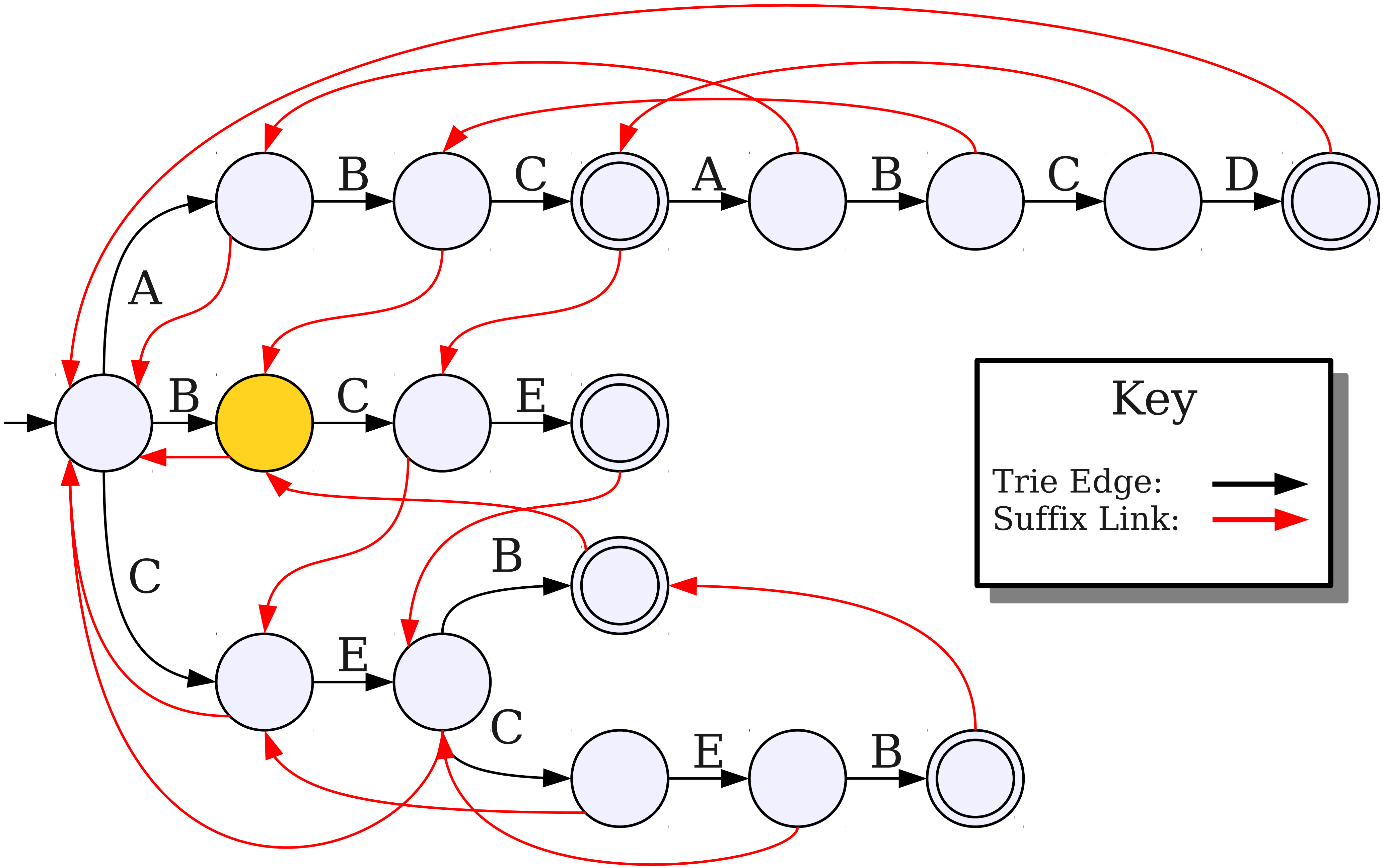
B C E A

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



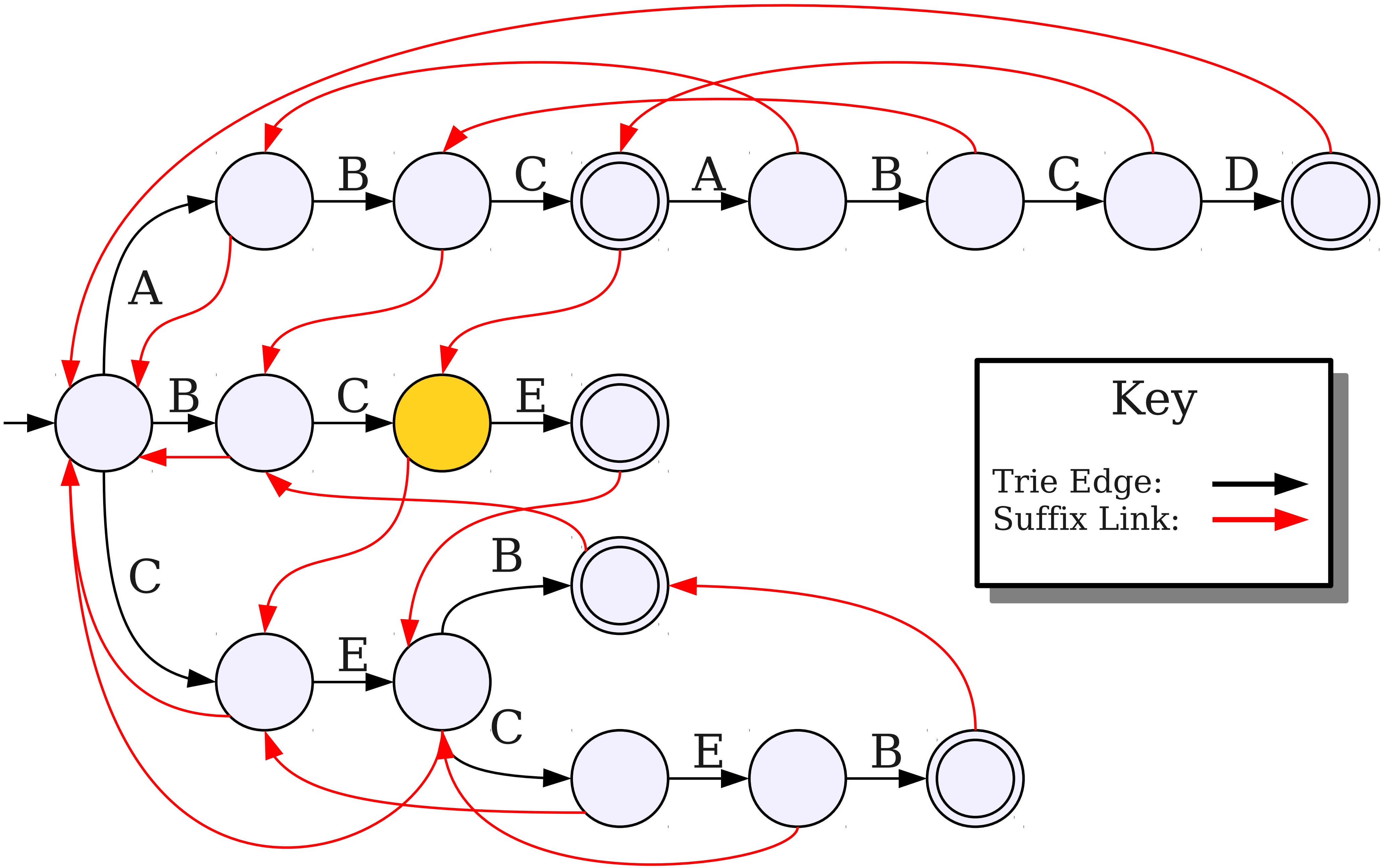
B C E A

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



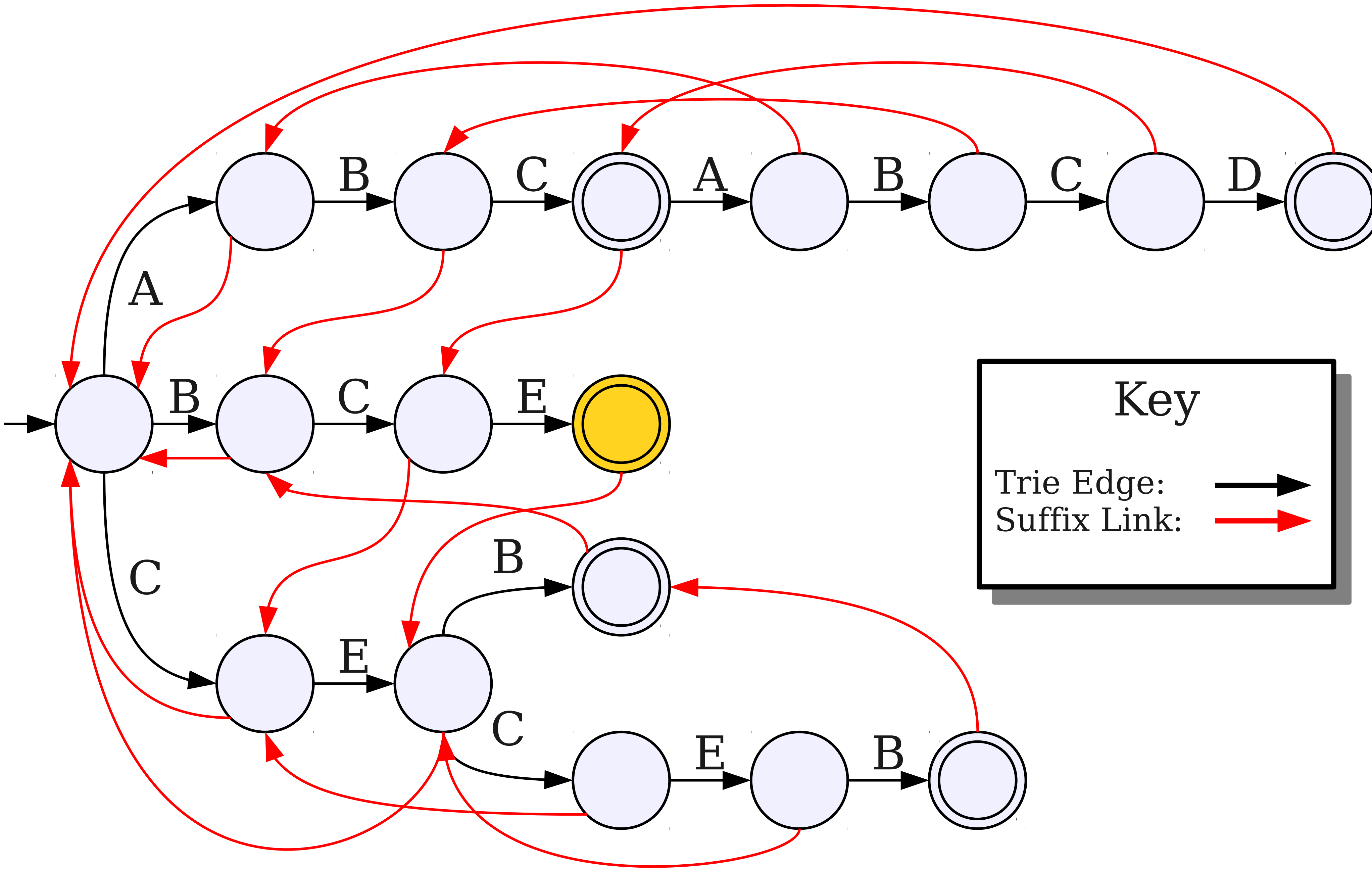
B C E A

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



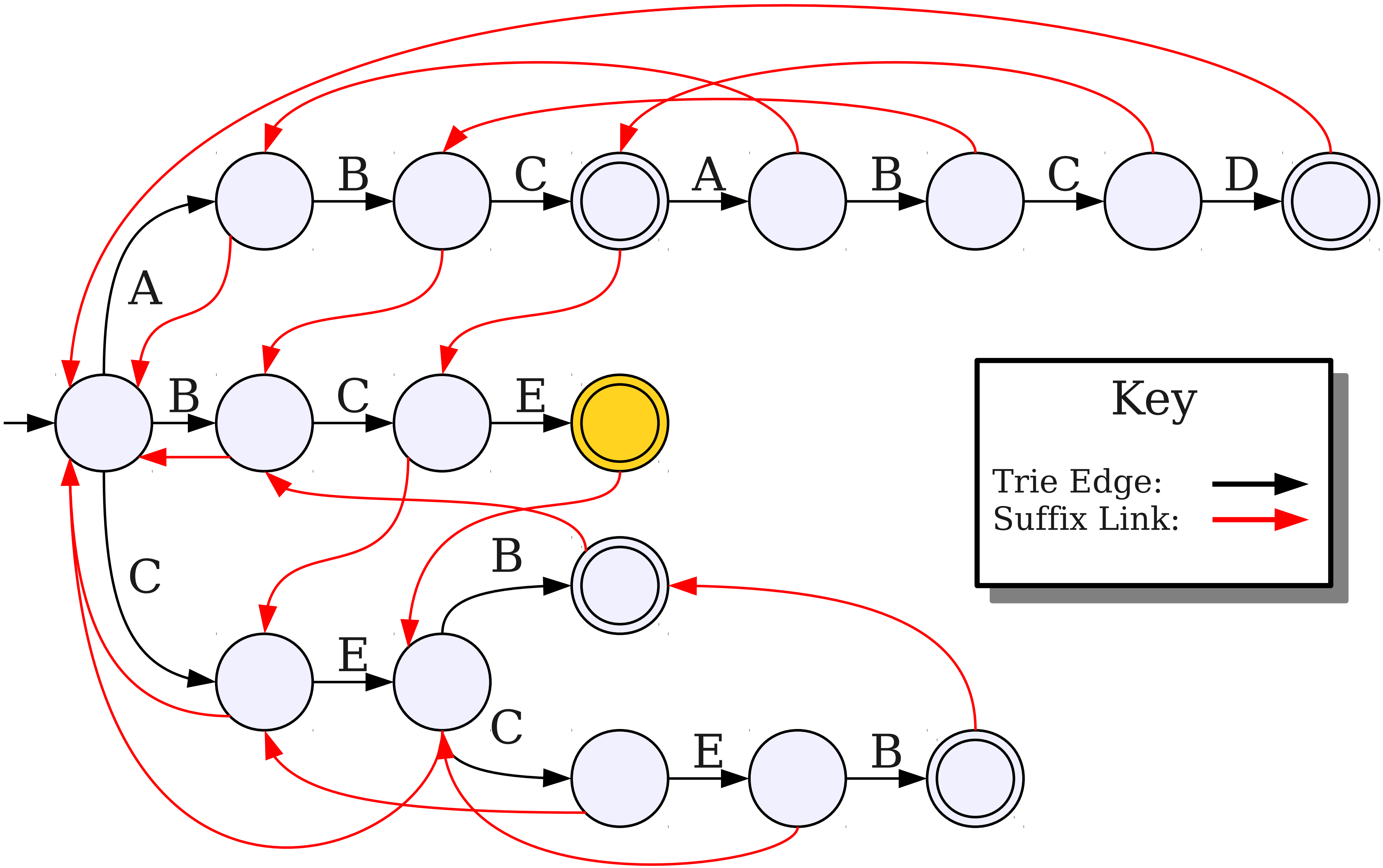
B C E A

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



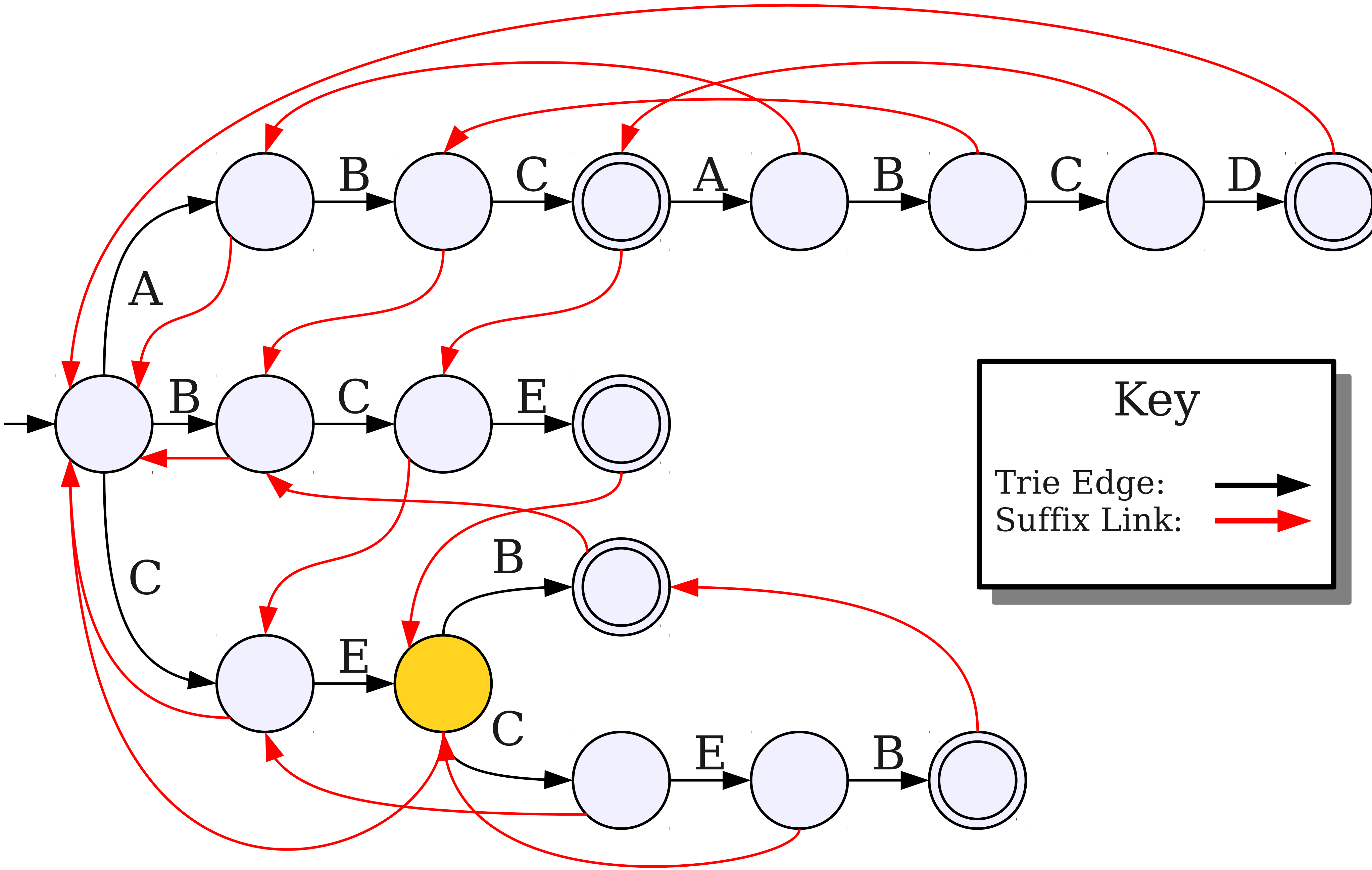
B C E A

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



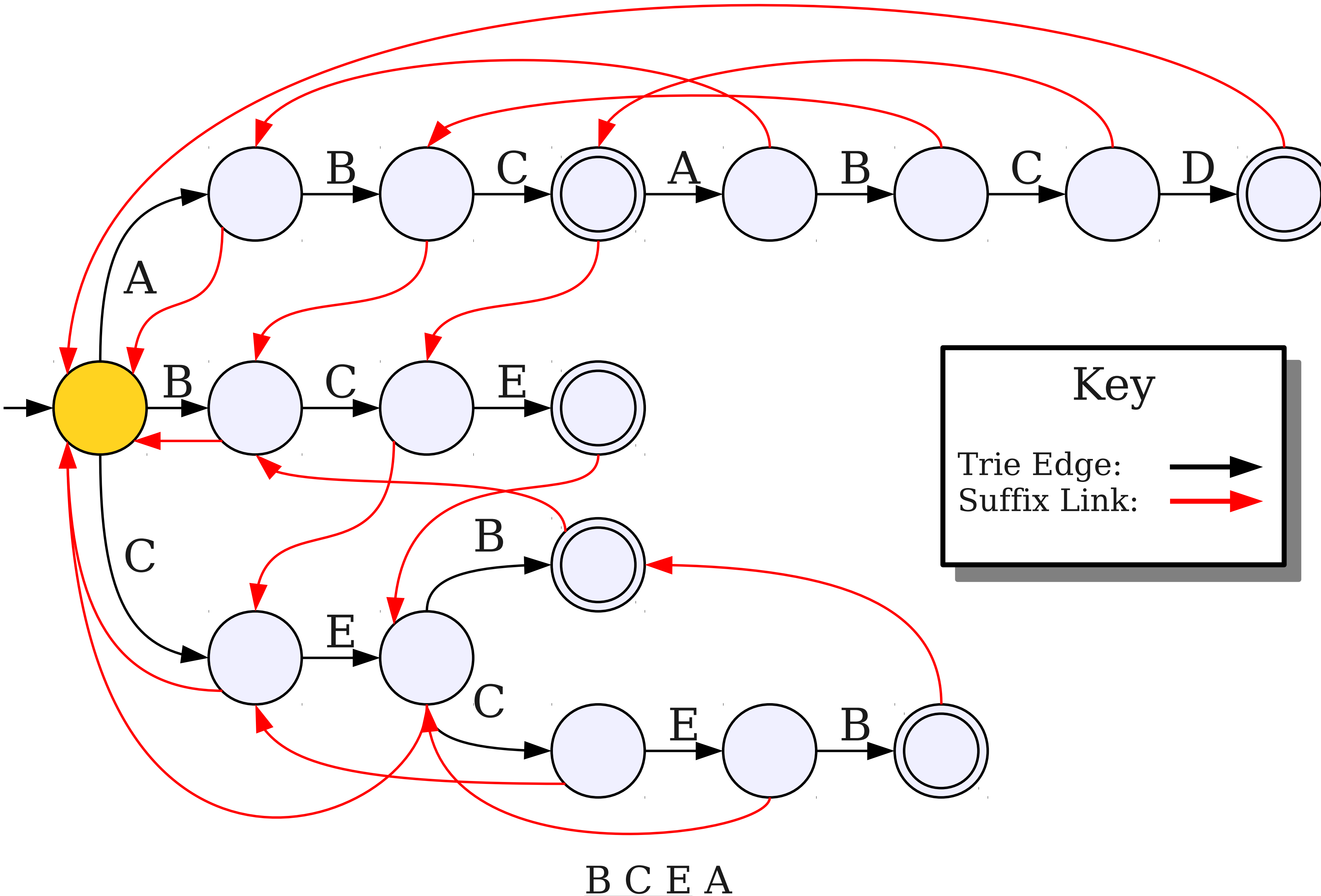
<u>B C E</u> A

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC

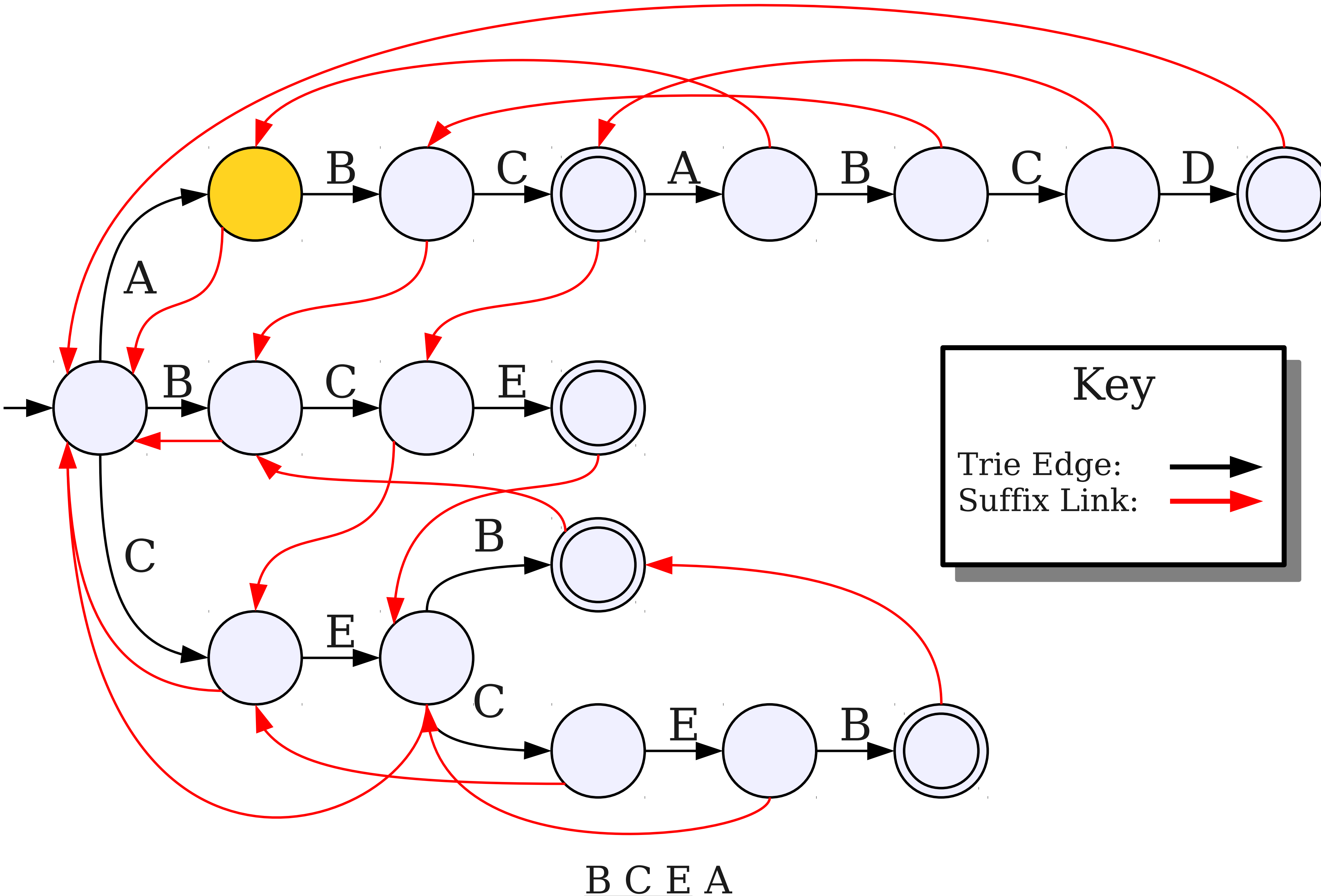


<u>B C E</u> A

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC

The (Basic) Algorithm

- Let *state* be the start state.
- For $i = 0$ to $m - 1$
  - While *state* is not start and there is no trie edge labeled $T[i]$:
    - Follow the suffix link.
  - If there is a trie edge labeled $T[i]$, follow that edge.

This algorithm won't actually mark all of the strings that appear in the text. We'll handle that later.

Runtime Analysis

- **Claim:** Once the trie is constructed and suffix links added, the runtime of searching through string P is O(m).
- **Proof:** Total number of steps forward is O(m), and we cannot follow suffix links backwards more times than we go forwards. Therefore, time complexity is O(m).

Will our heroes ever build
suffix links efficiently?

And will they be able to match
pattern strings quickly?

Stay tuned!

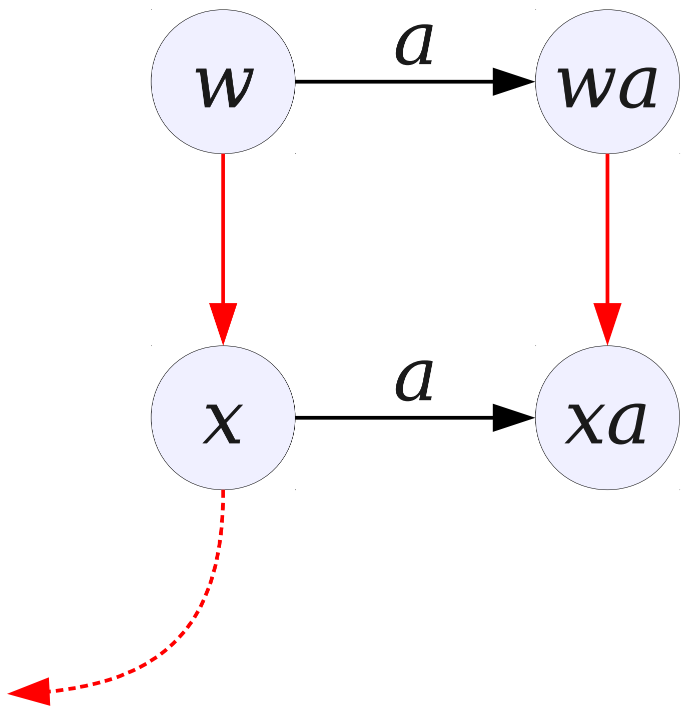
Your Questions

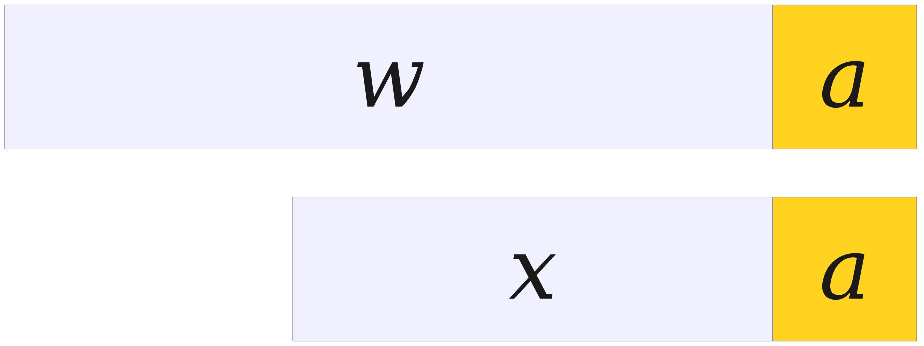
The Story So Far

- Start with a trie.
- Add suffix links to allow for failure recovery and fast searching.
- Unresolved questions:
  - How do you build suffix links efficiently?
  - How do you do searches efficiently?

Constructing Suffix Links

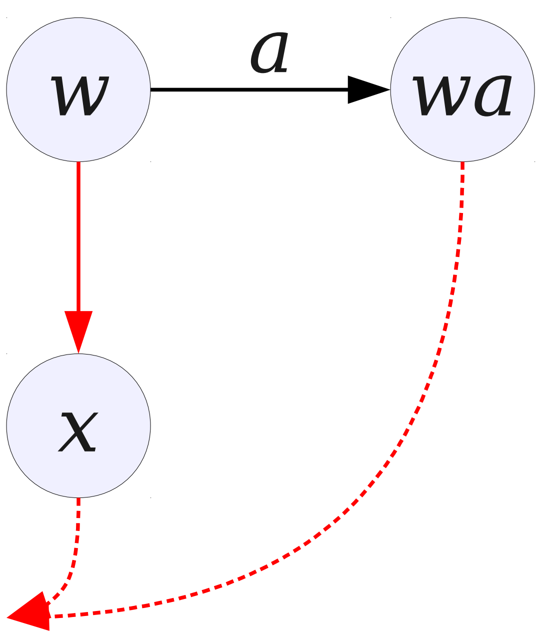
- **Key insight**: Suppose we know the suffix link for a node labeled w. After following a trie edge labeled a, there are two possibilities.
- **Case 1:** xa exists.

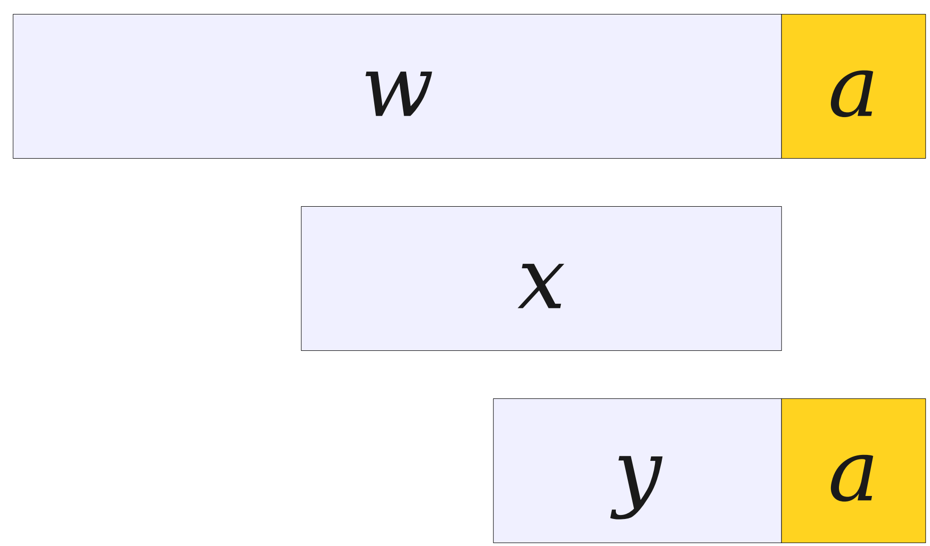




Constructing Suffix Links

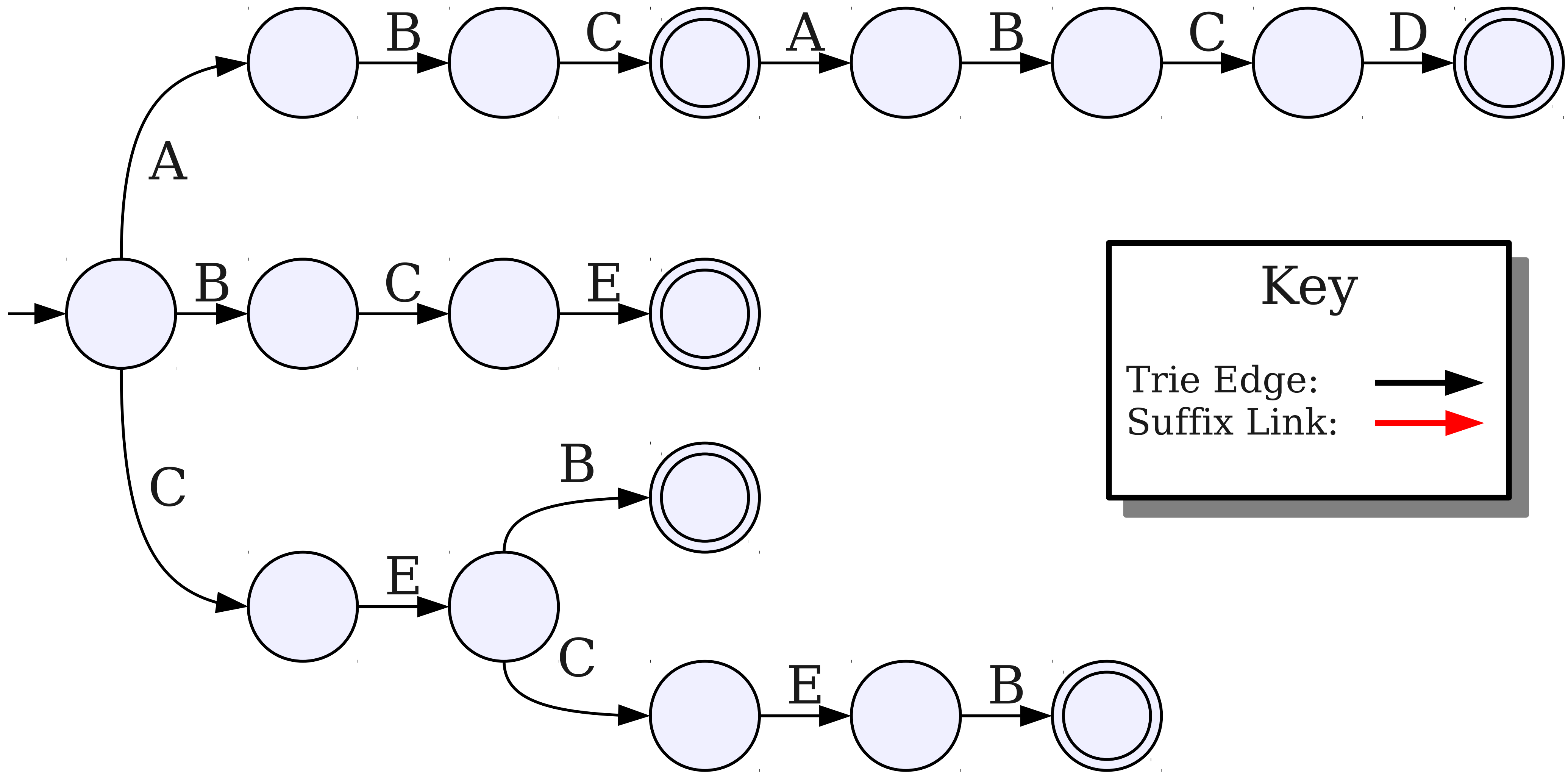
- **Key insight**: Suppose we know the suffix link for a node labeled *w*. After following a trie edge labeled *a*, there are two possibilities.
- **Case 2:** *xa* does not exist.



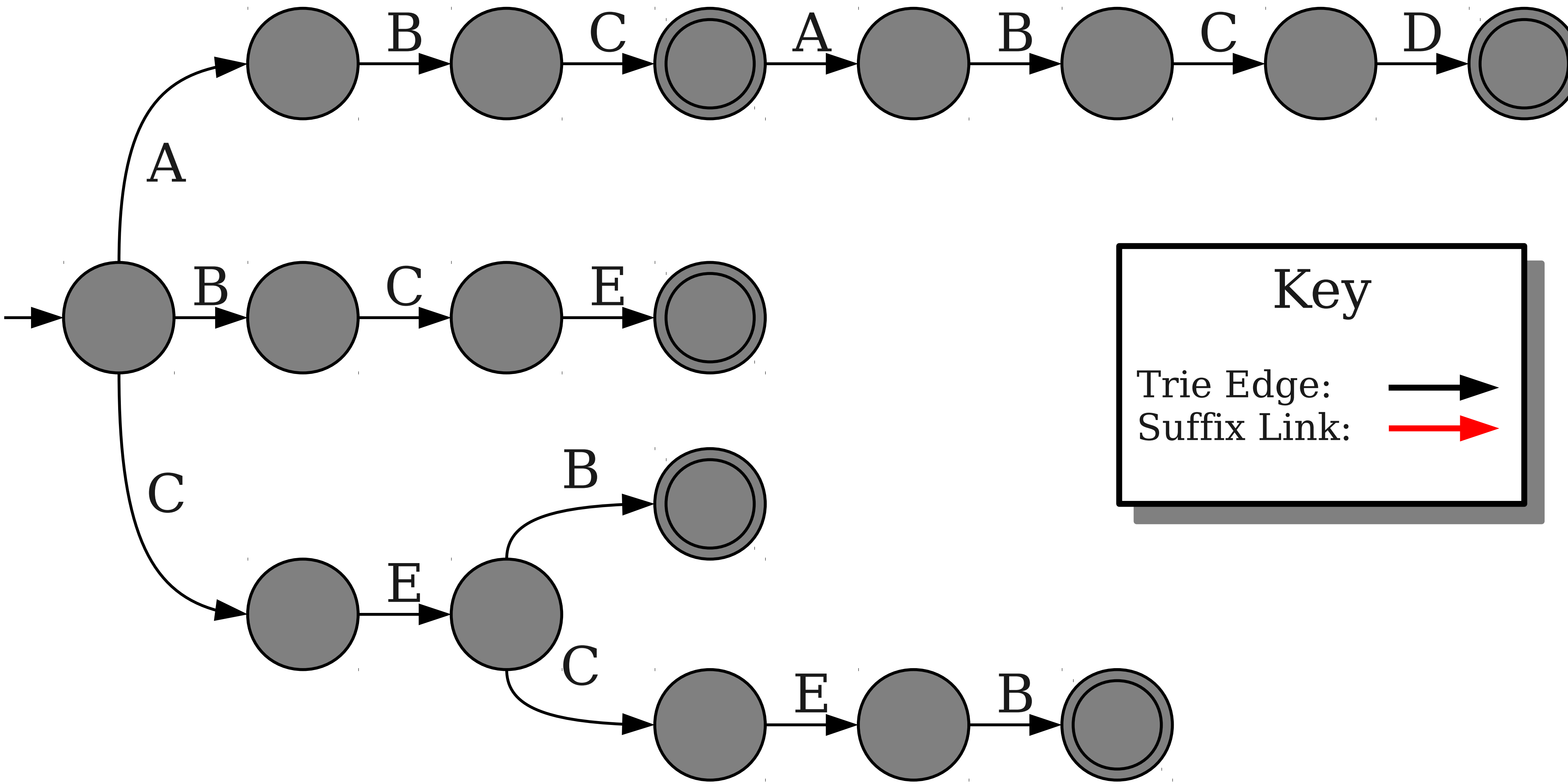


Constructing Suffix Links

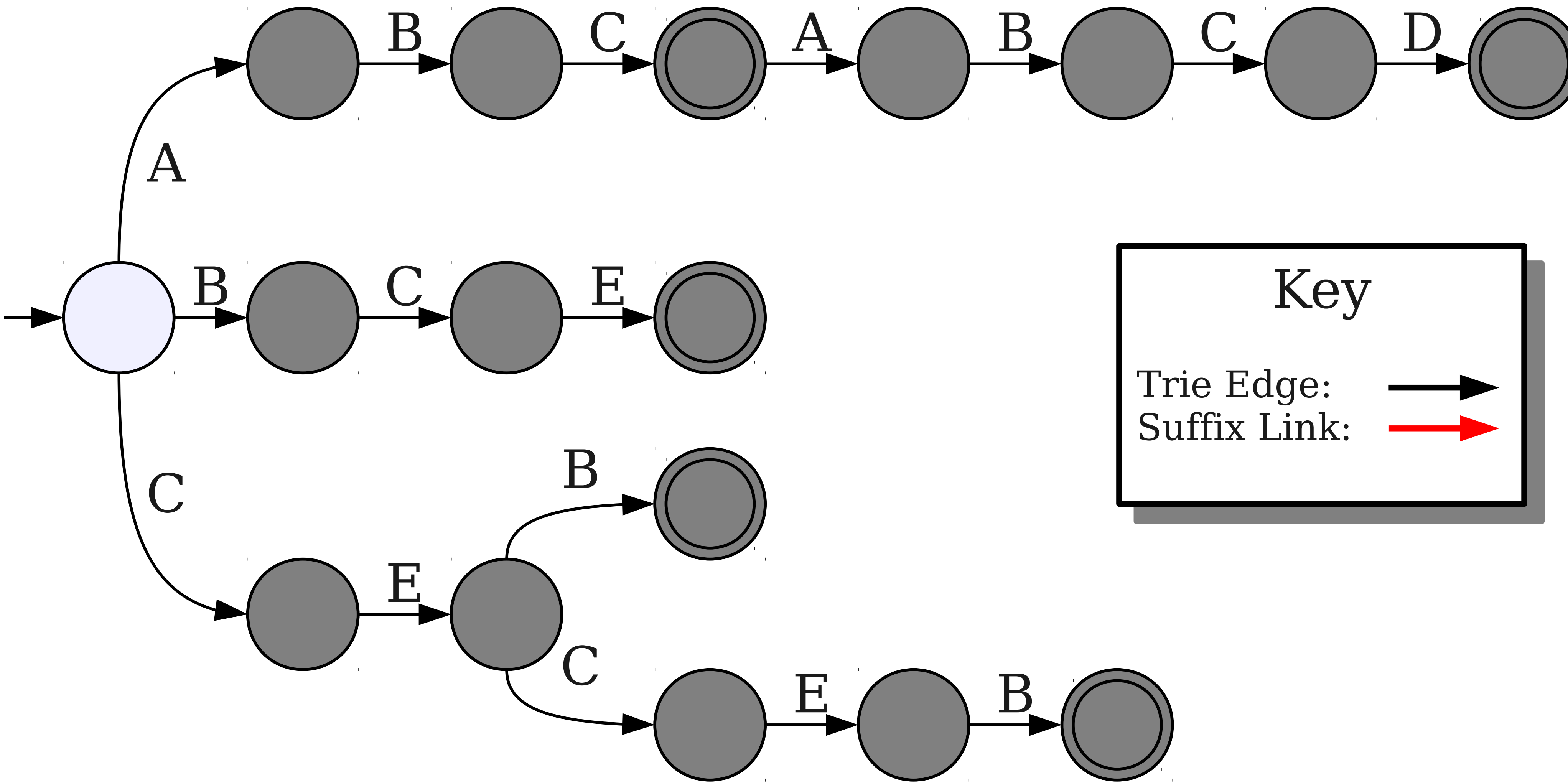
- To construct the suffix link for a node *wa*:
  - Follow *w*'s suffix link to node *x*.
  - If node *xa* exists, *wa* has a suffix link to *xa*.
  - Otherwise, follow *x*'s suffix link and repeat.
  - If you need to follow backwards from the root, then *wa*'s suffix link points to the root.
- **Idea:** Construct suffix links for trie nodes ascending order of length using BFS.



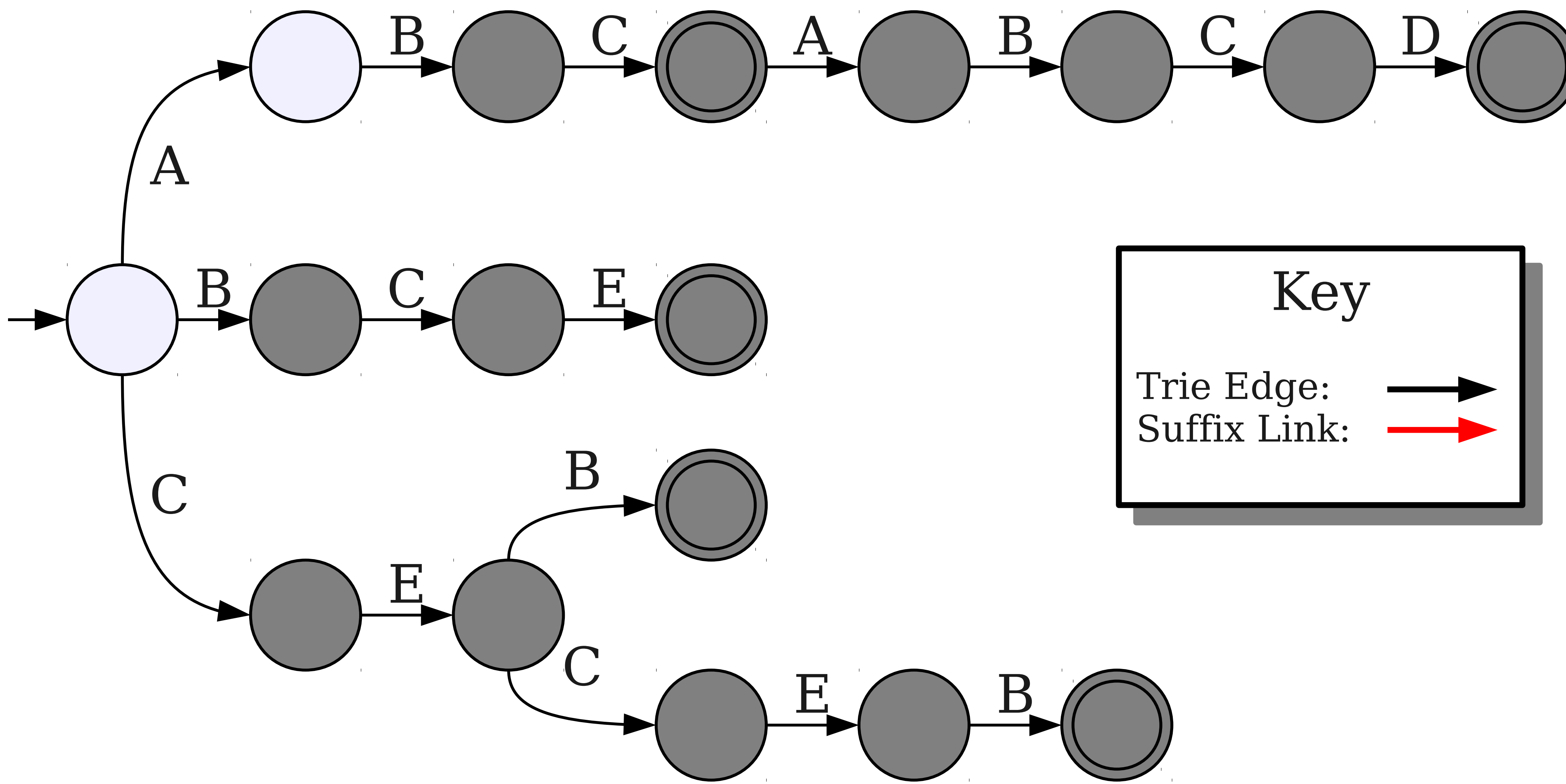
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



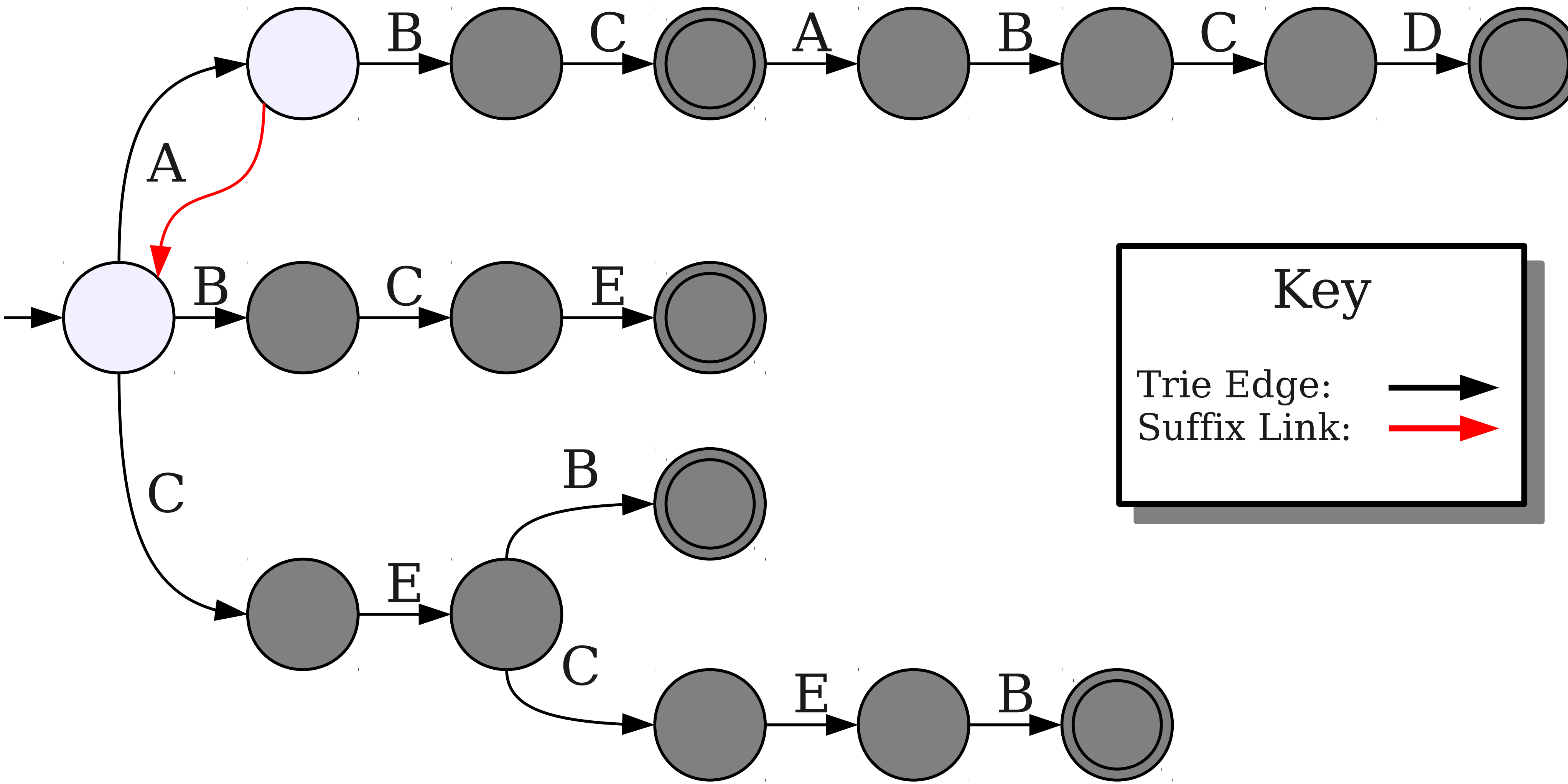
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



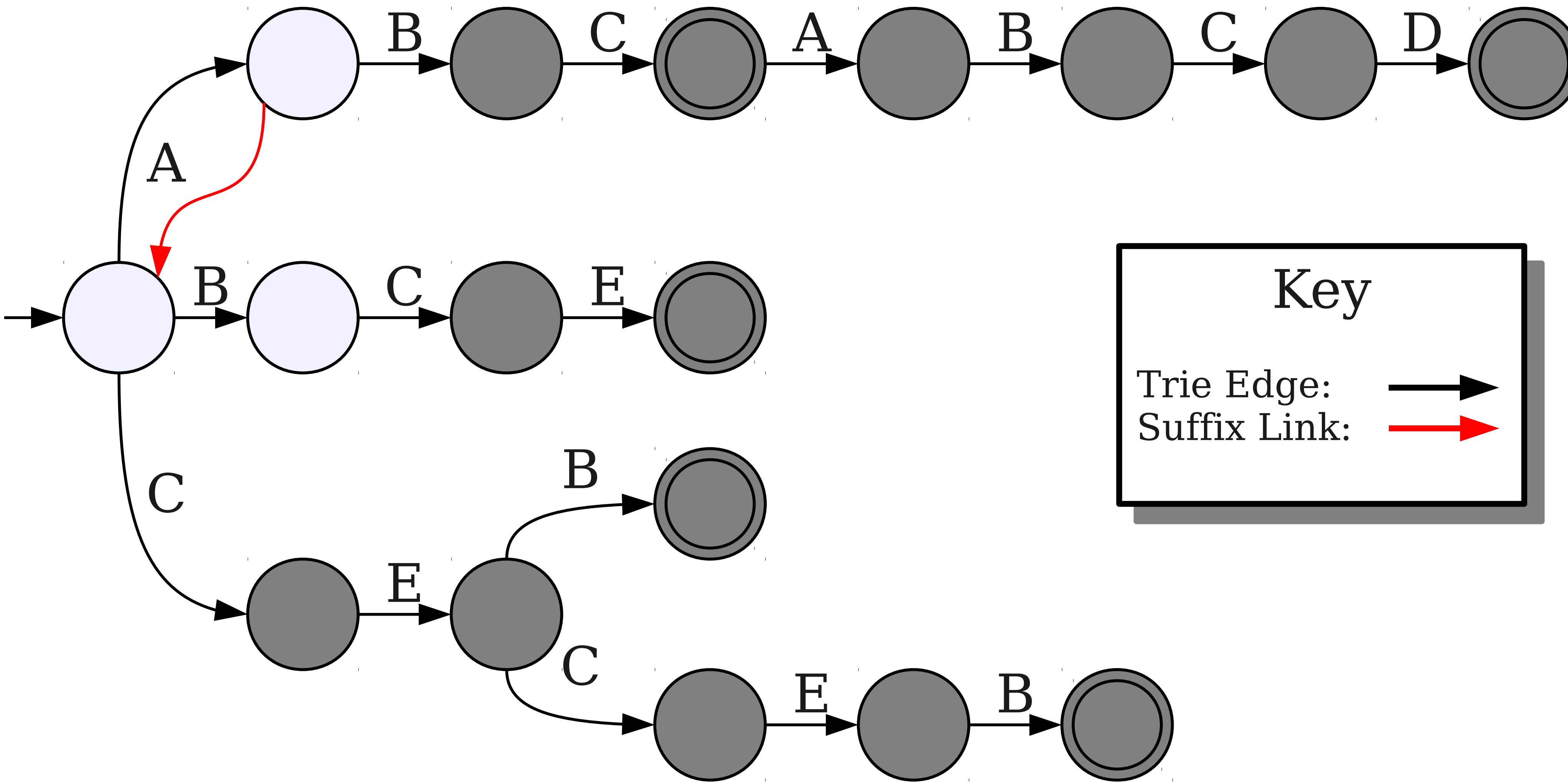
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



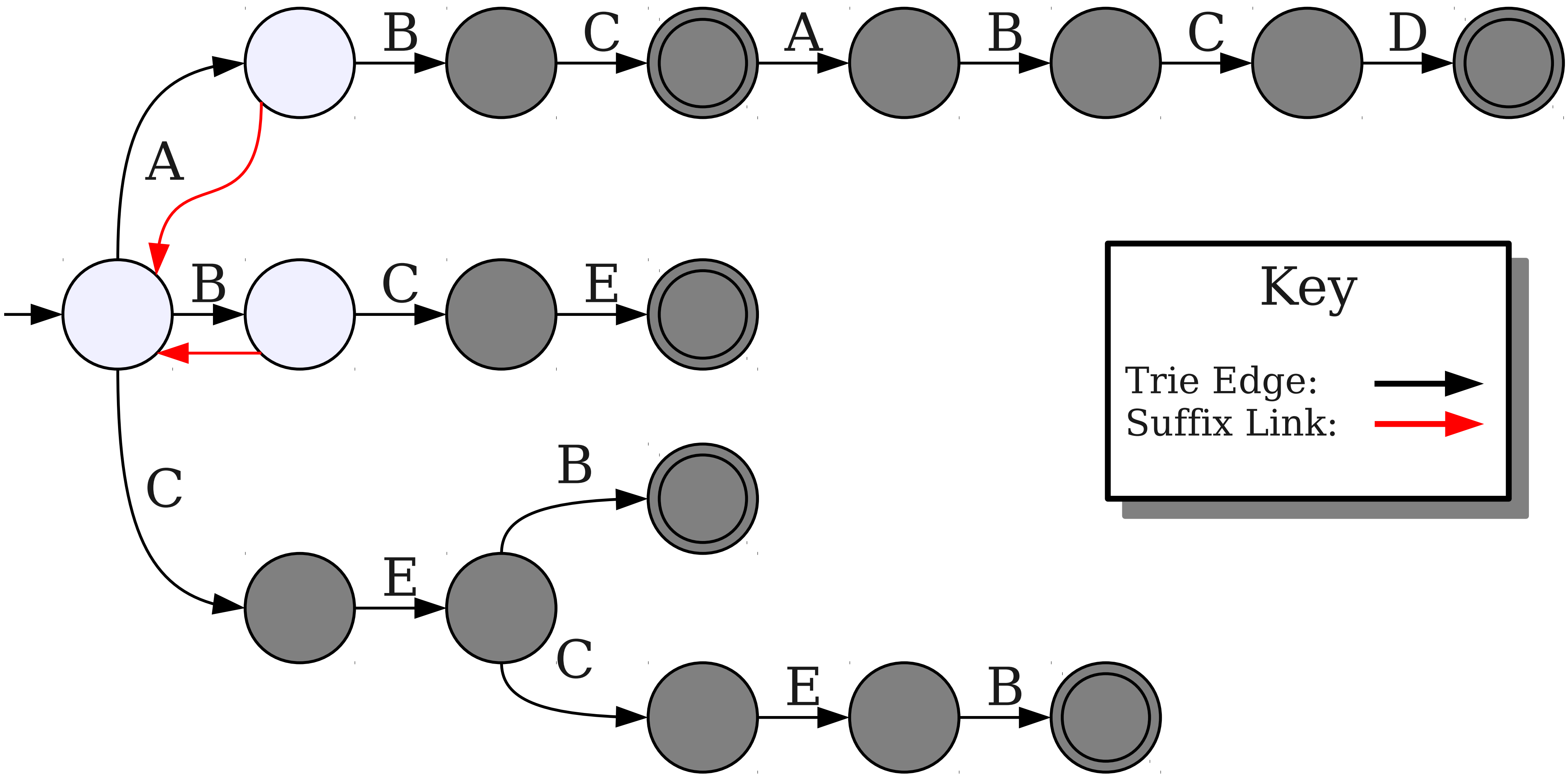
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



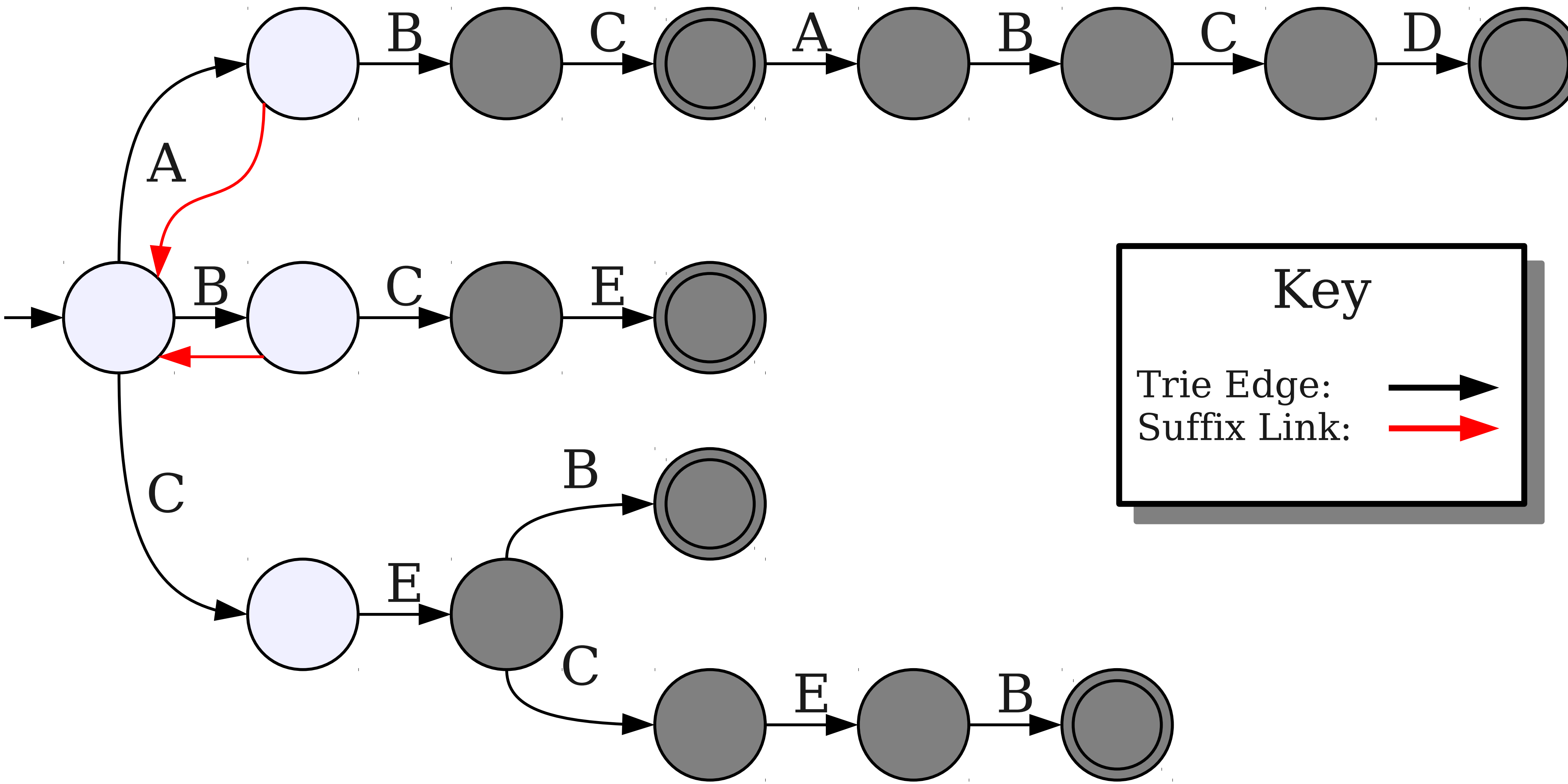
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



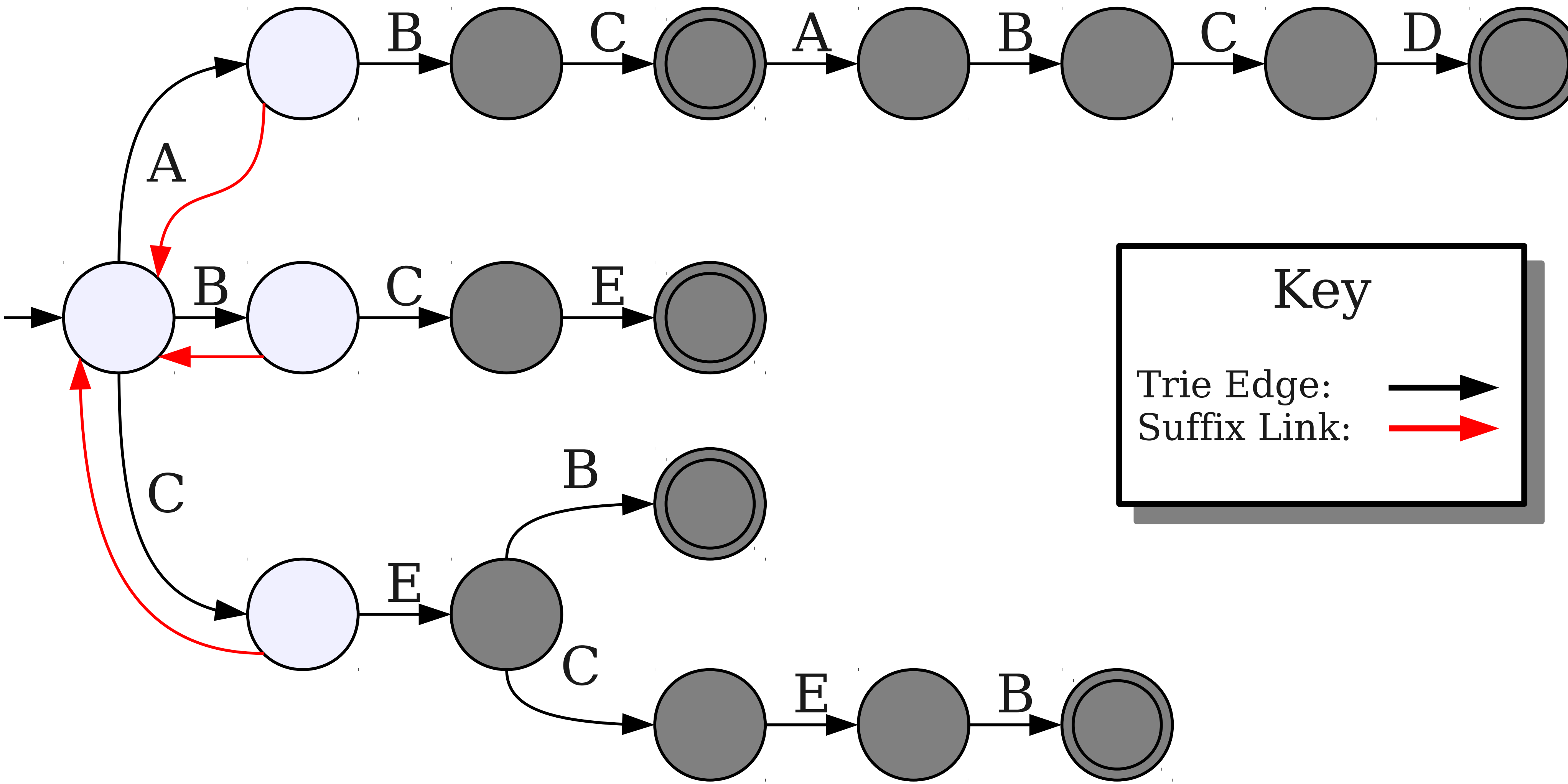
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



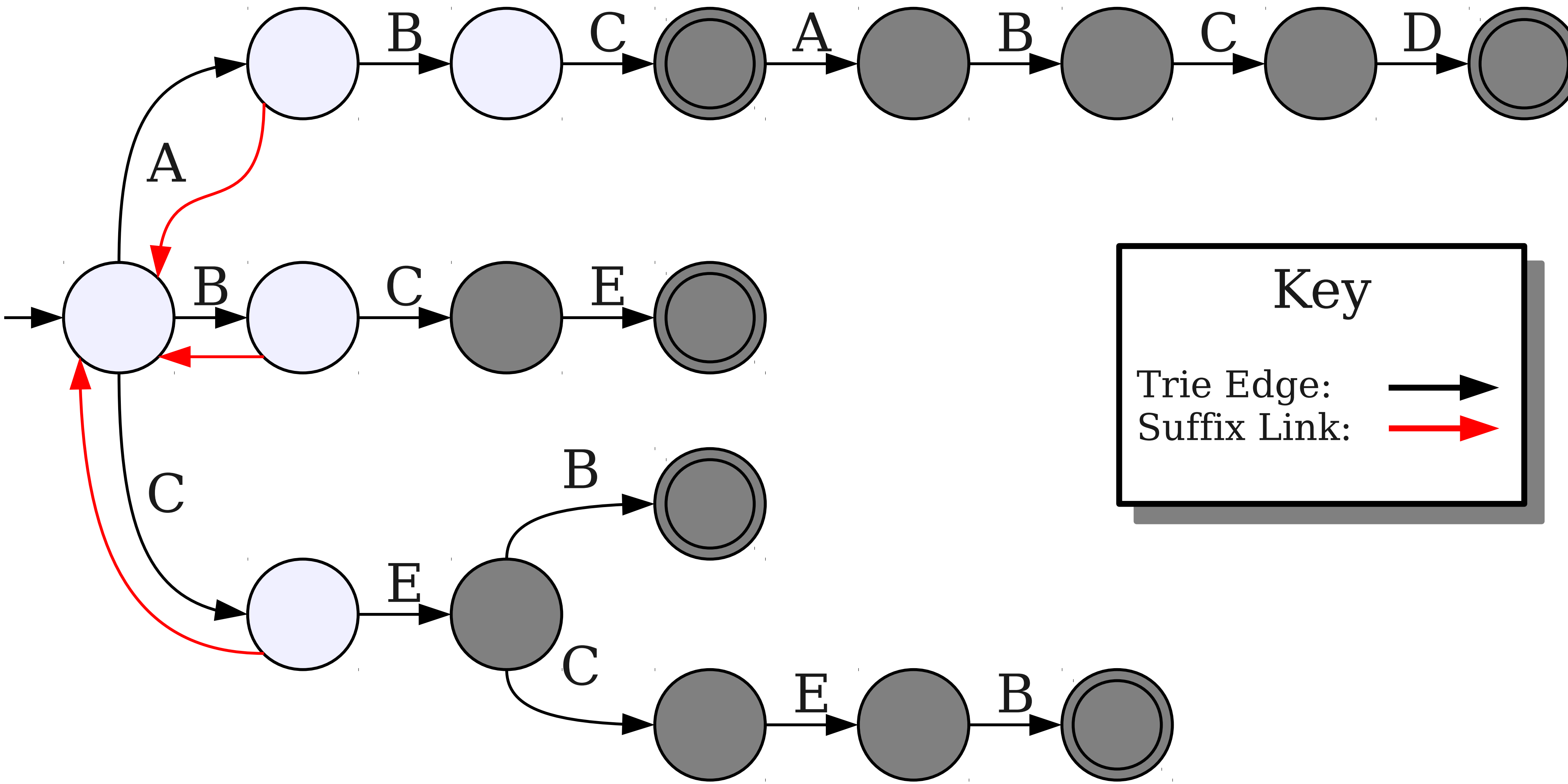
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



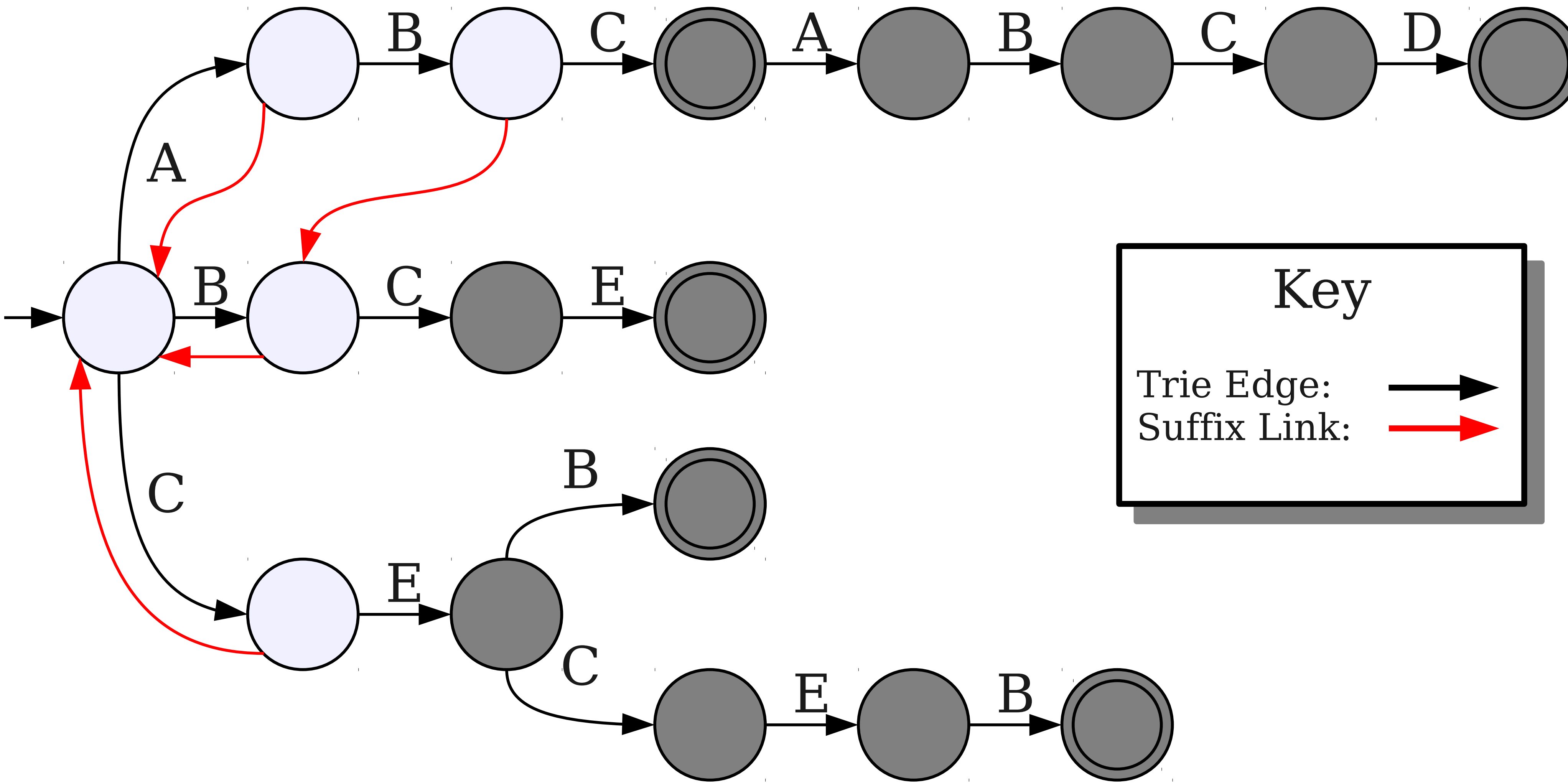
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



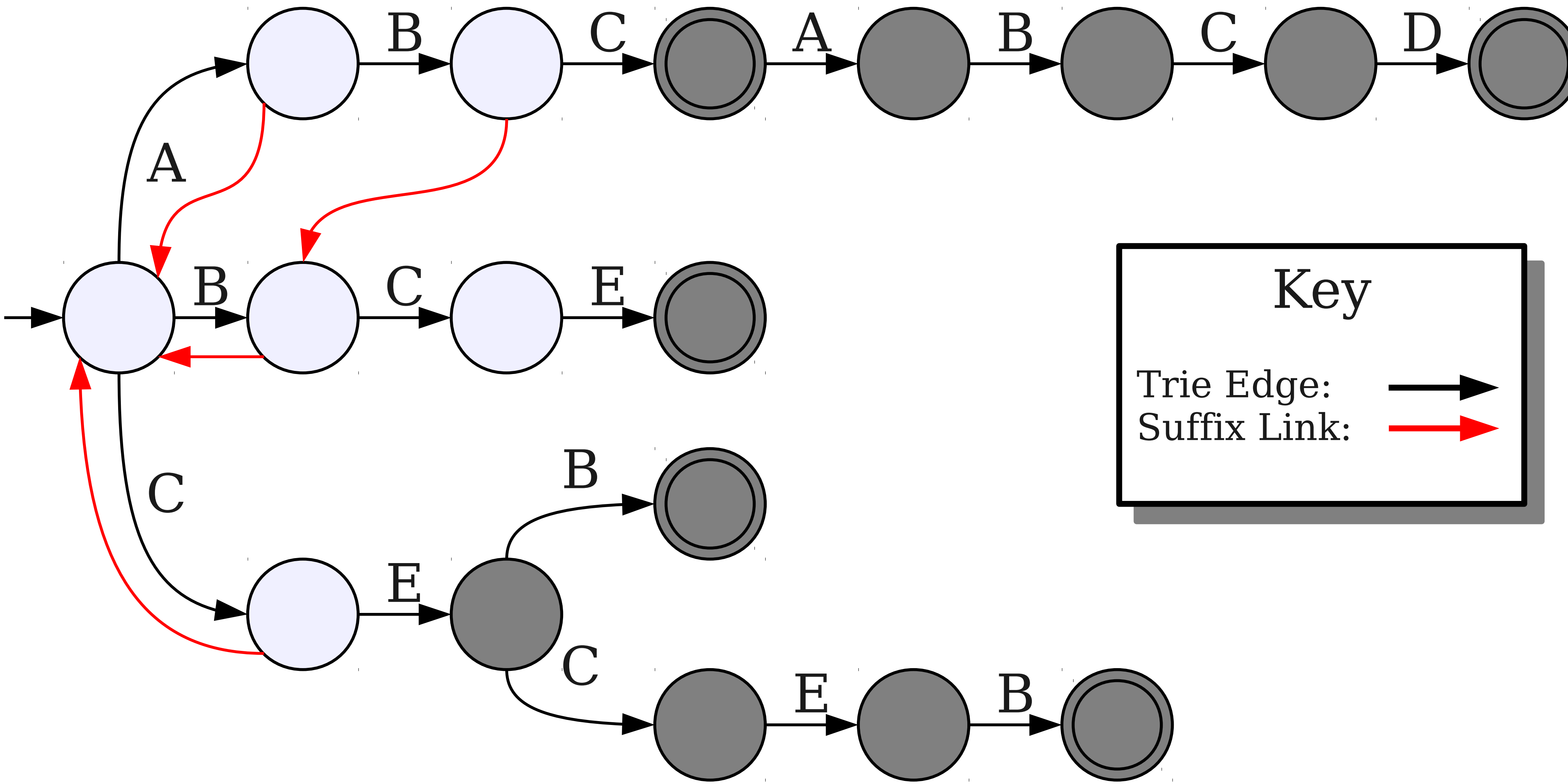
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



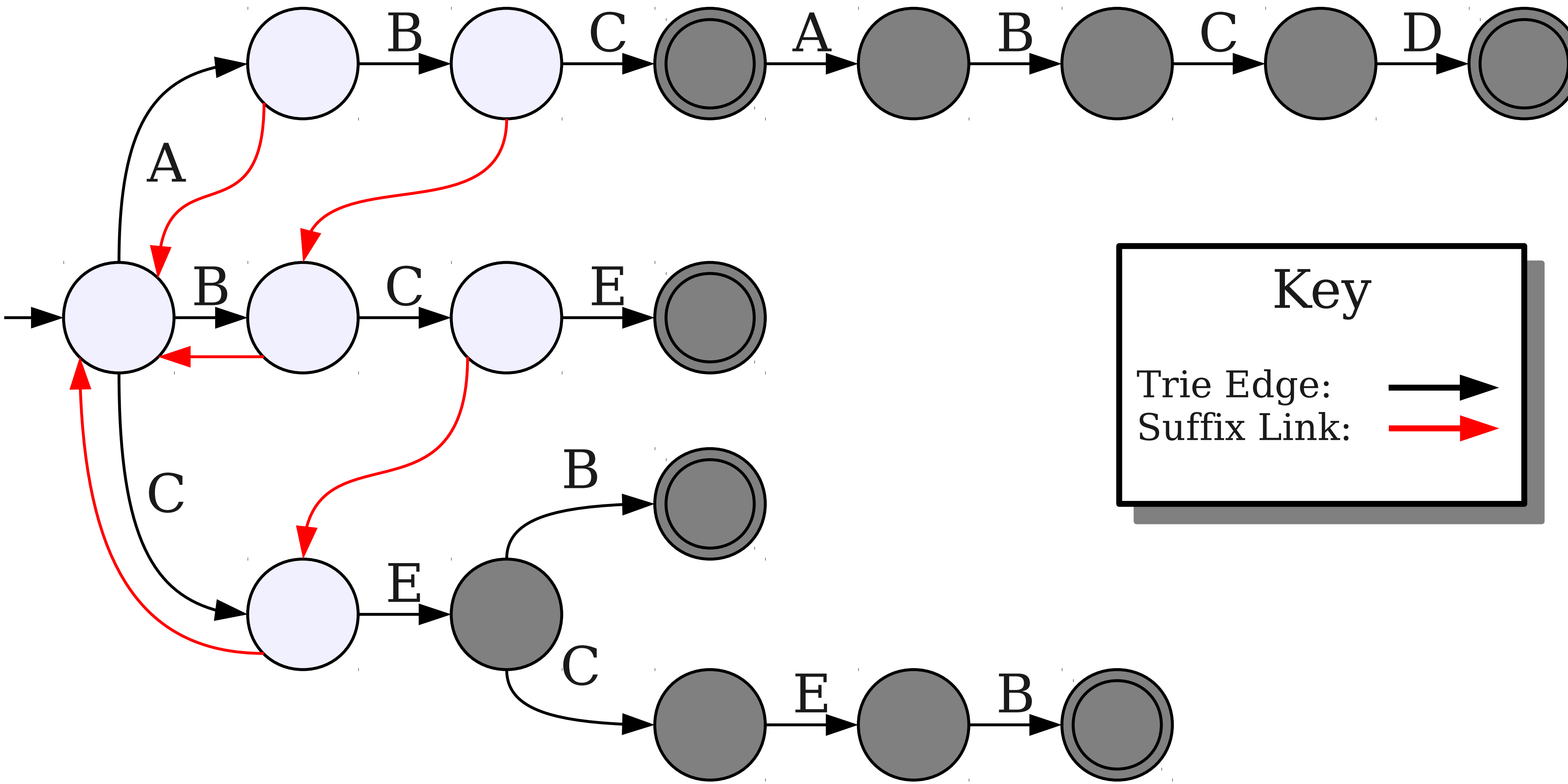
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



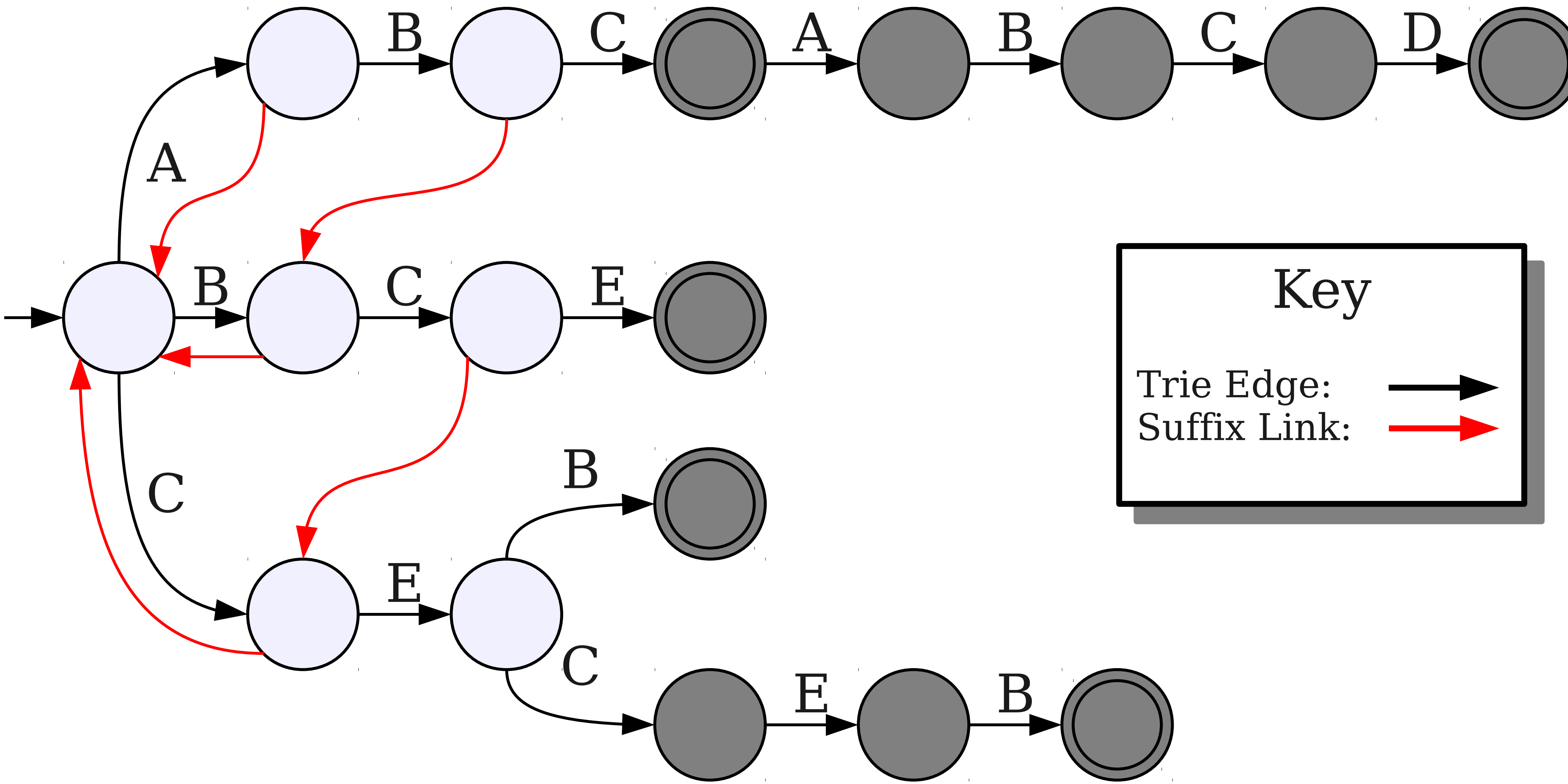
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



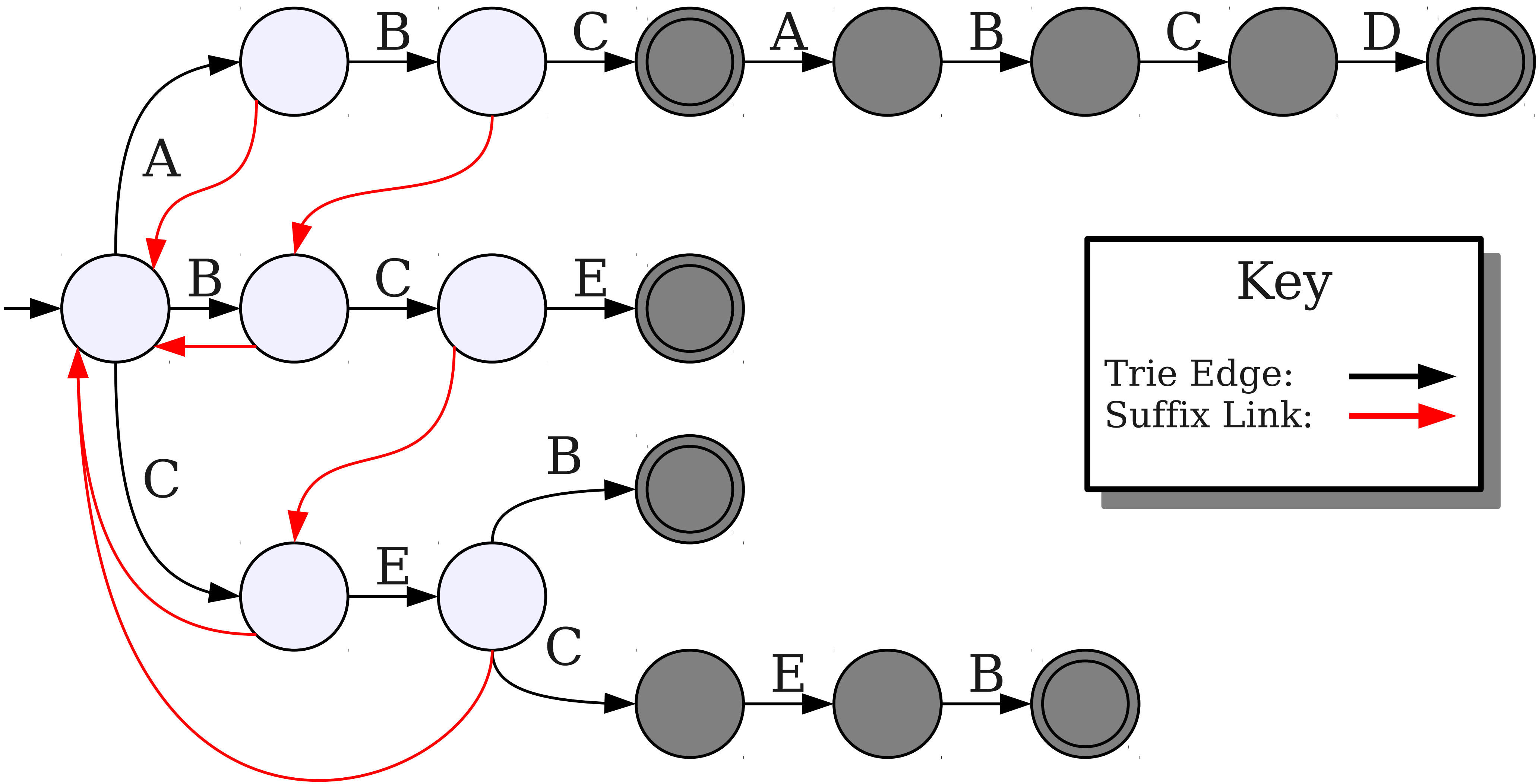
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



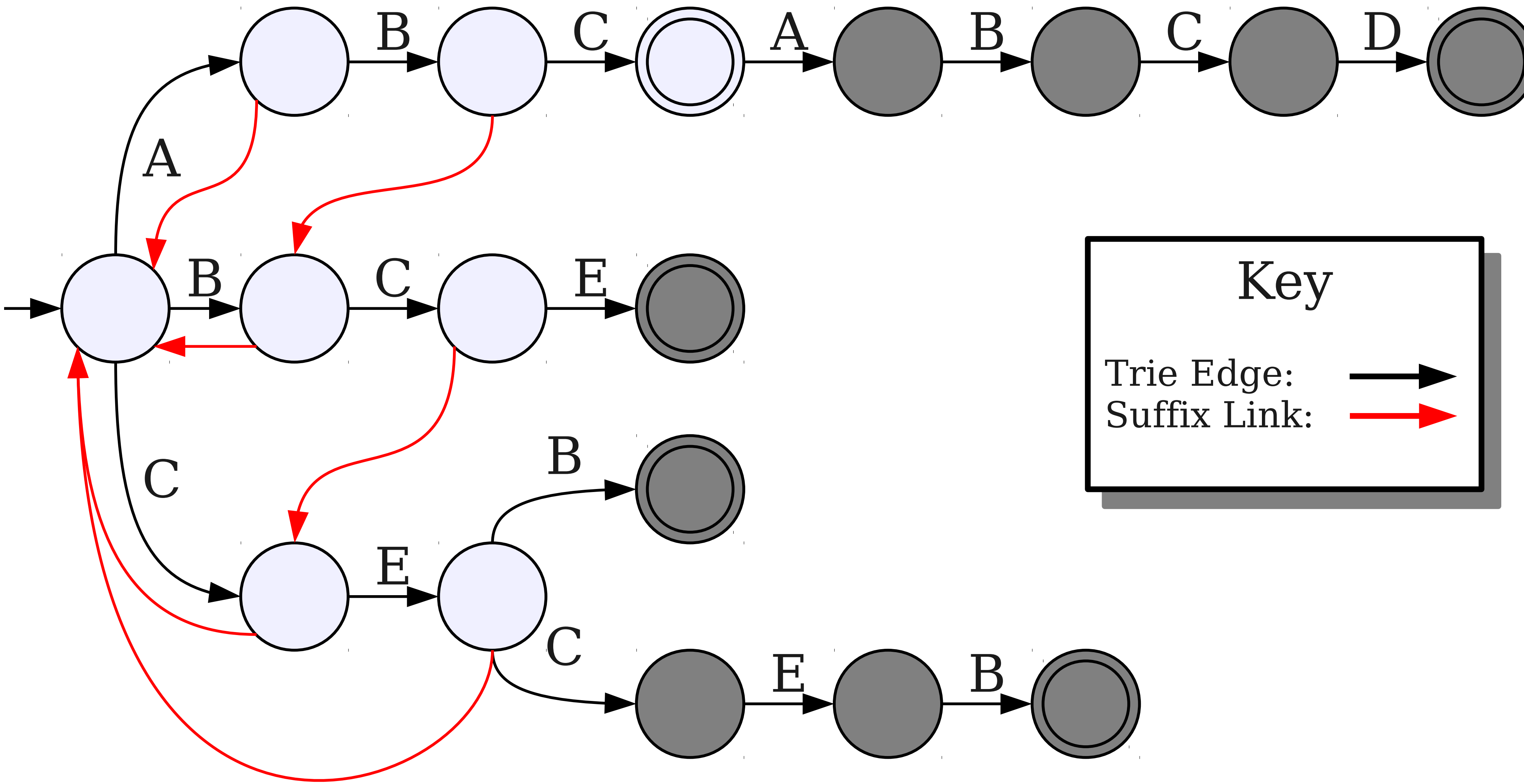
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



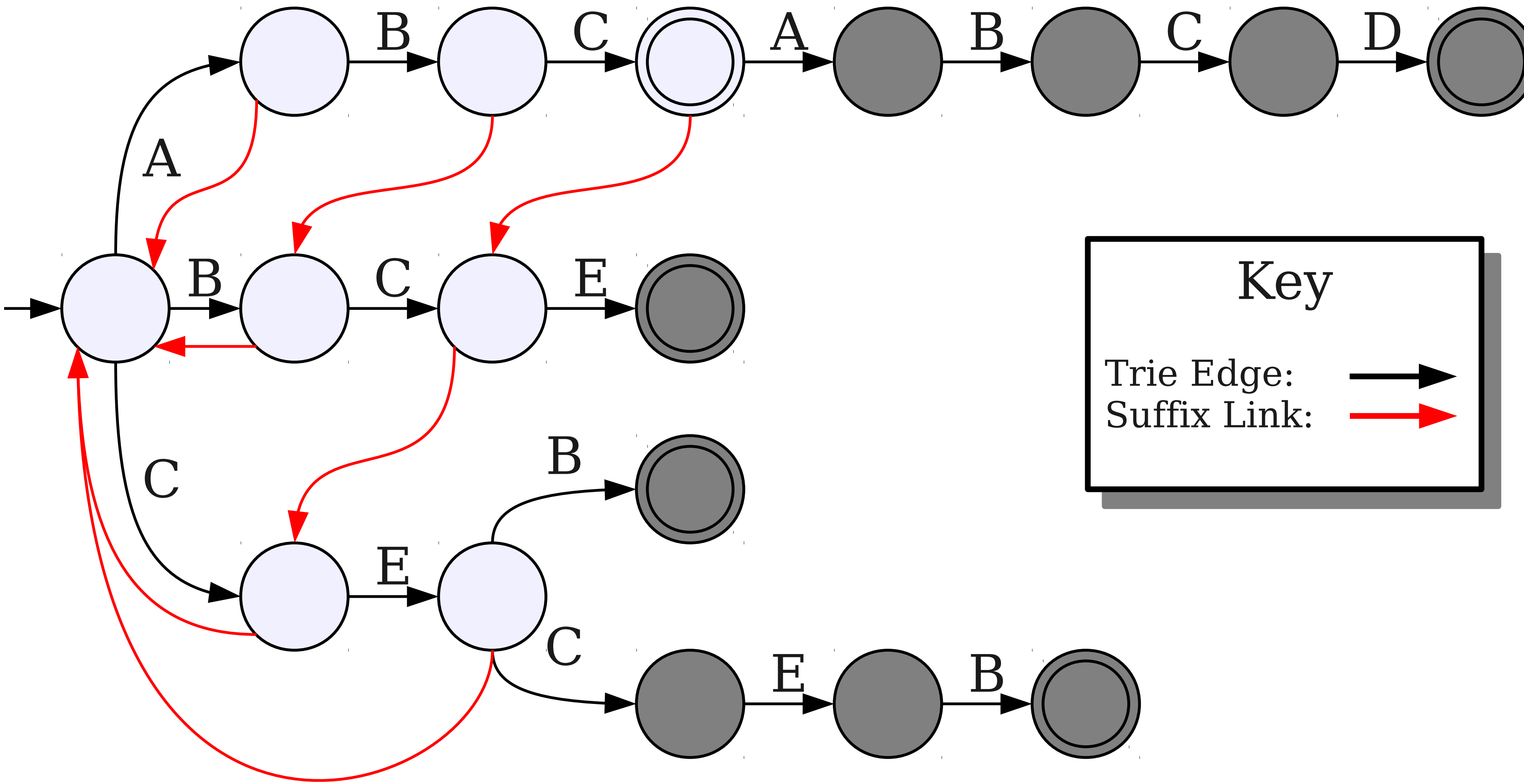
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



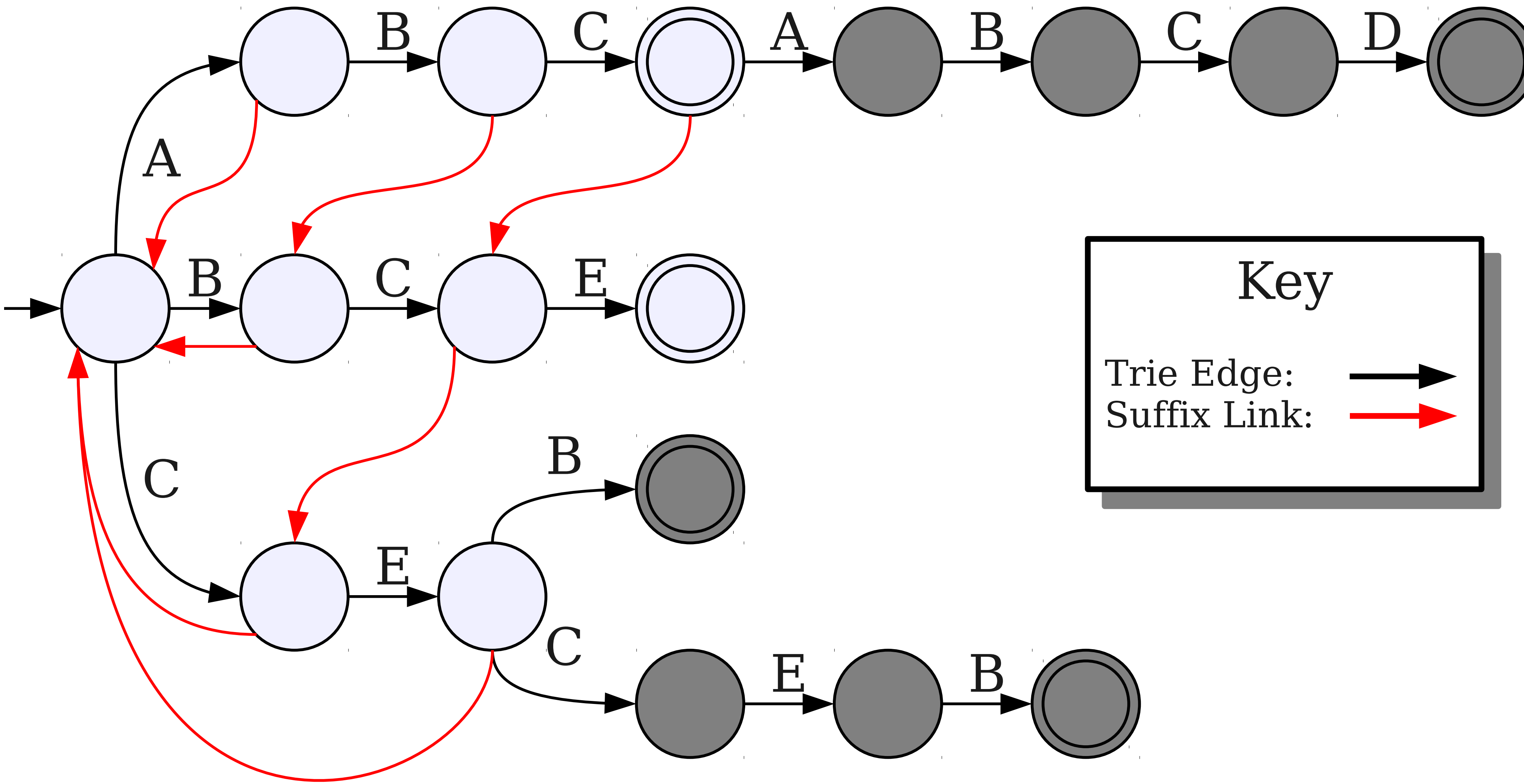
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



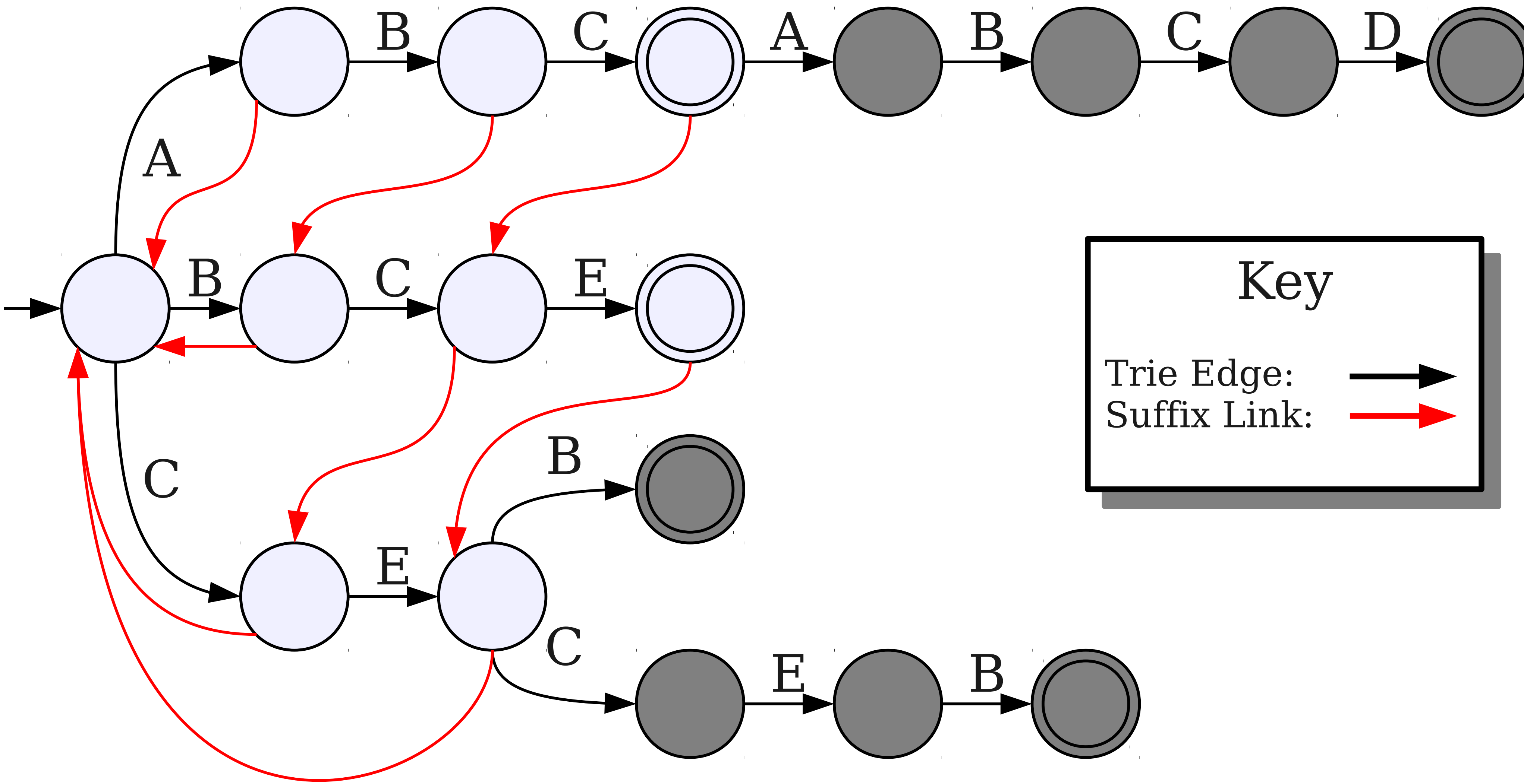
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



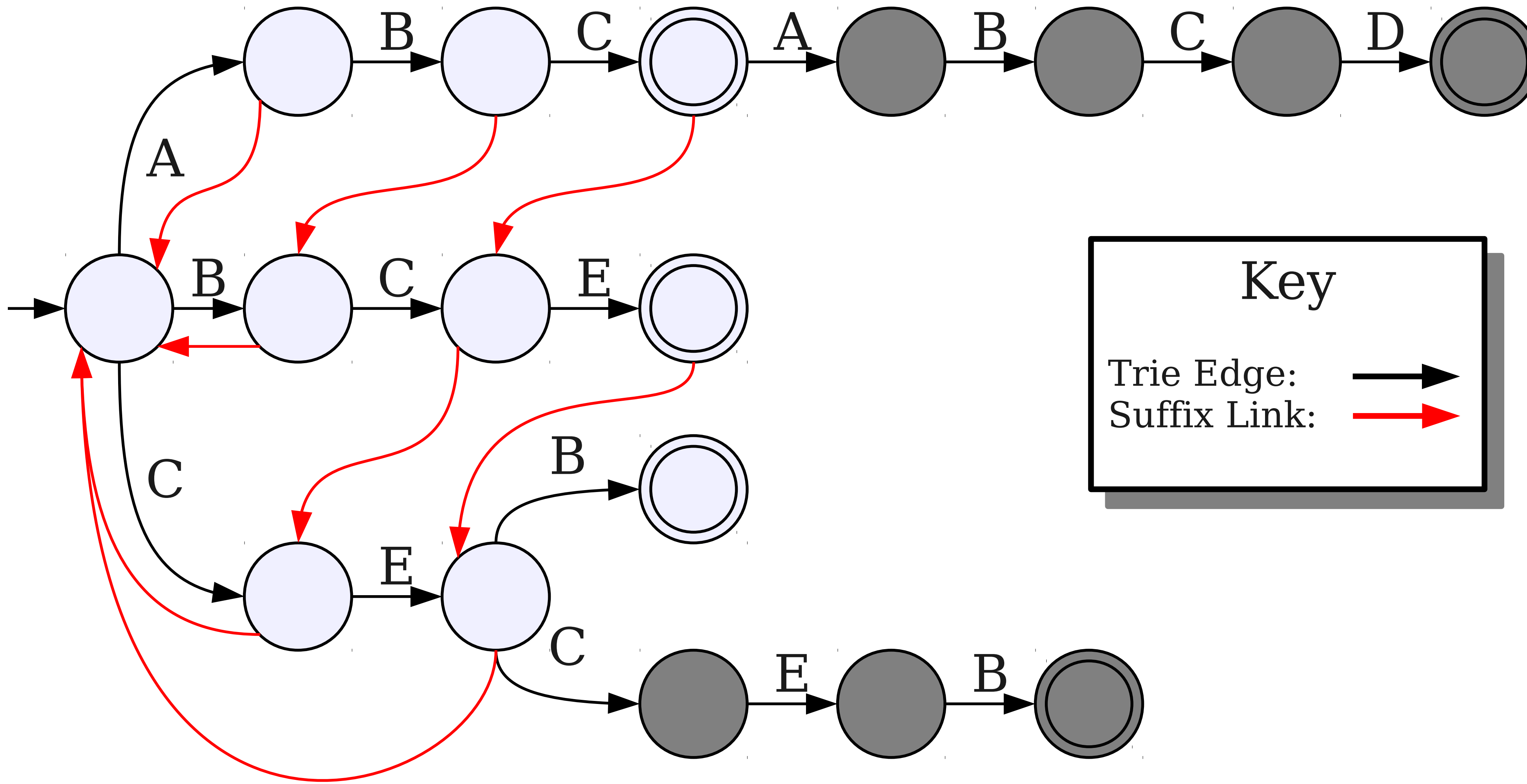
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



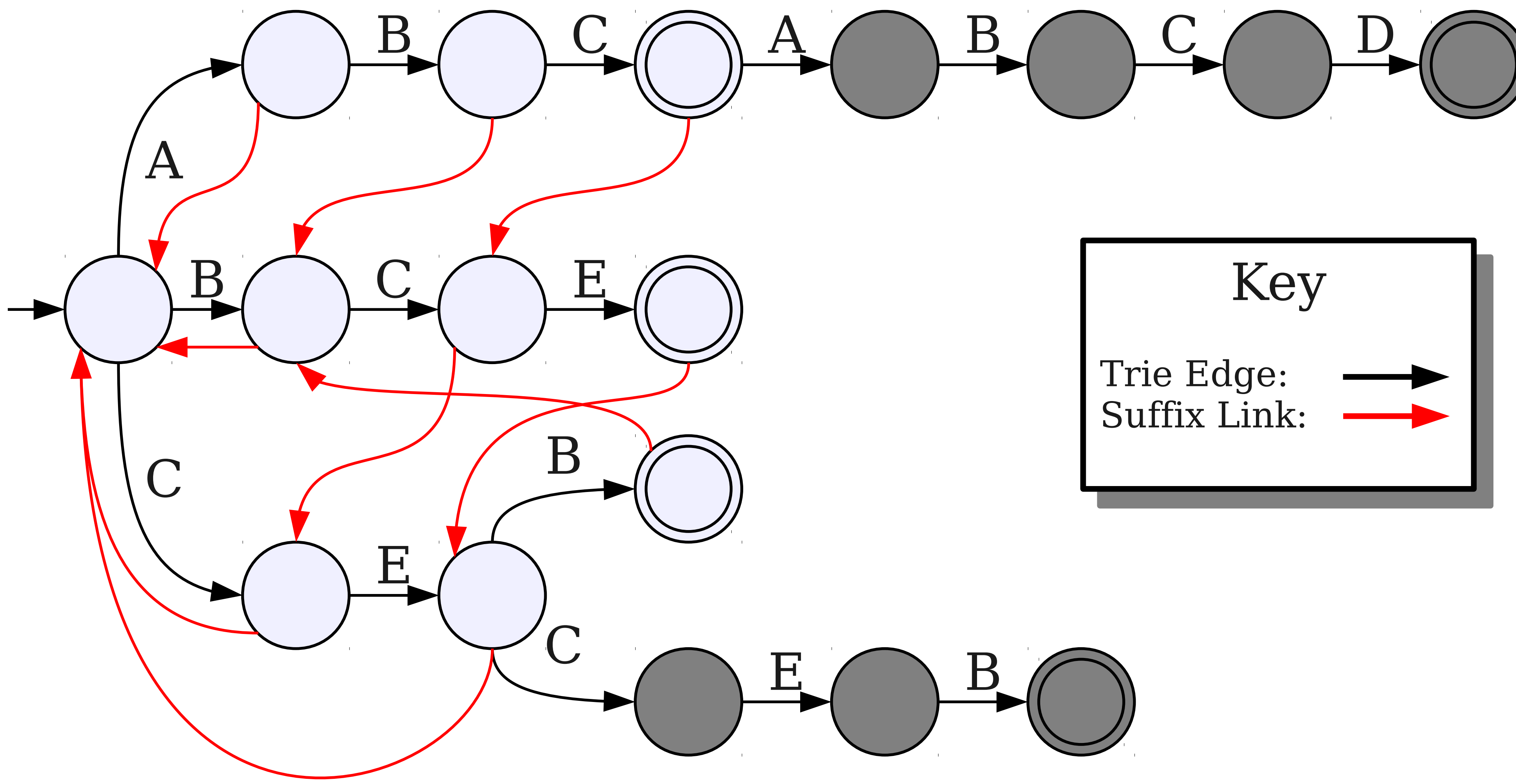
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



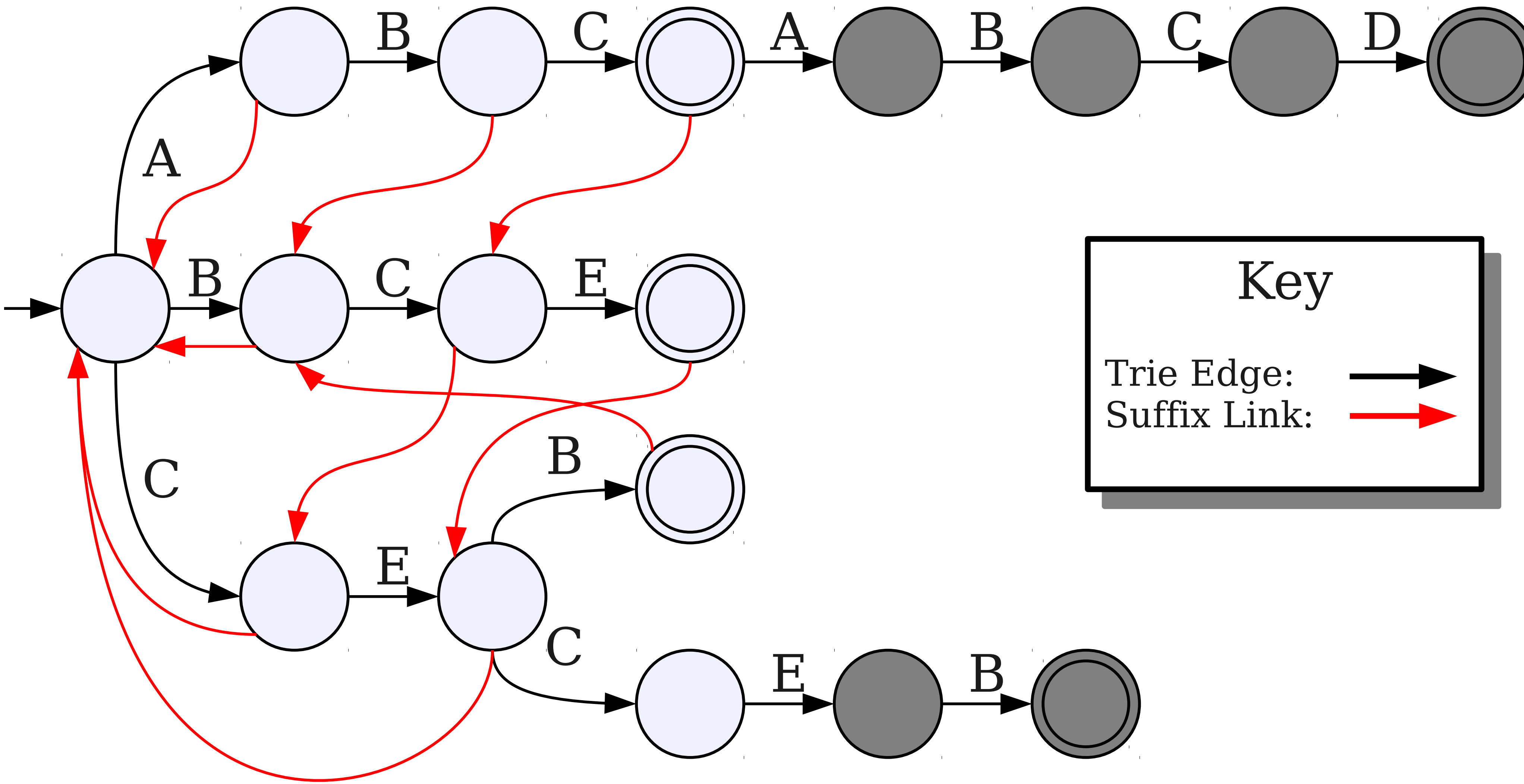
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



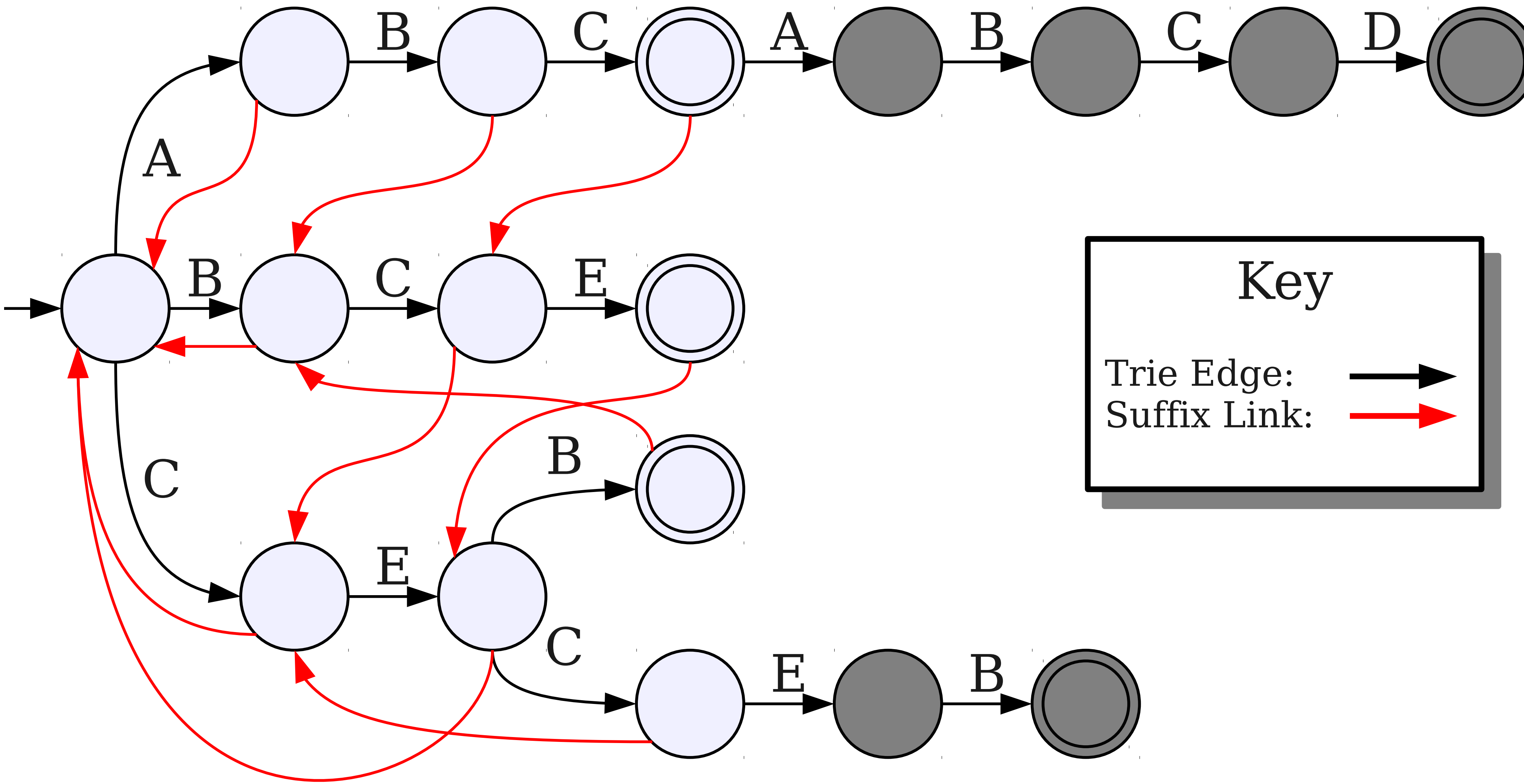
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



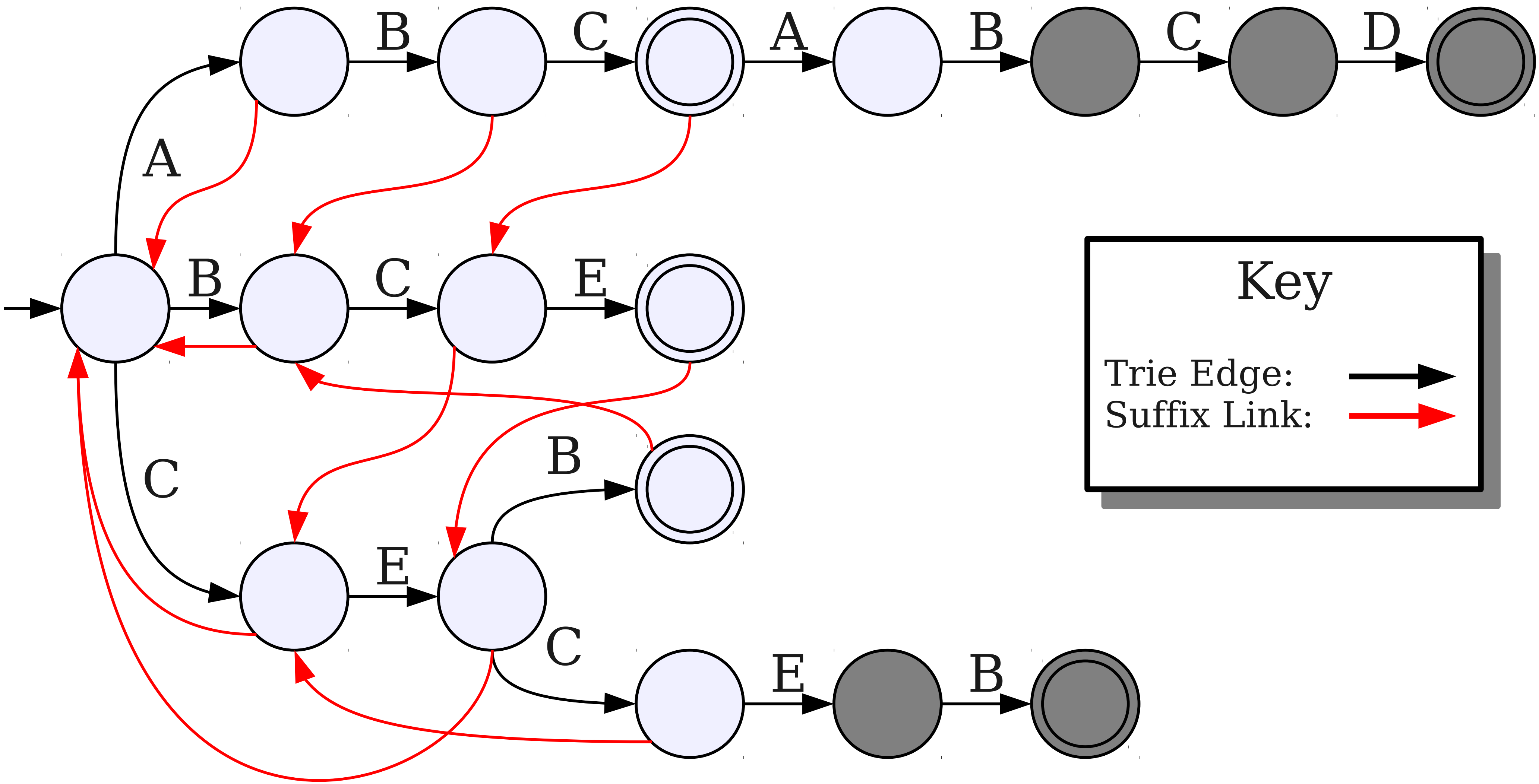
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



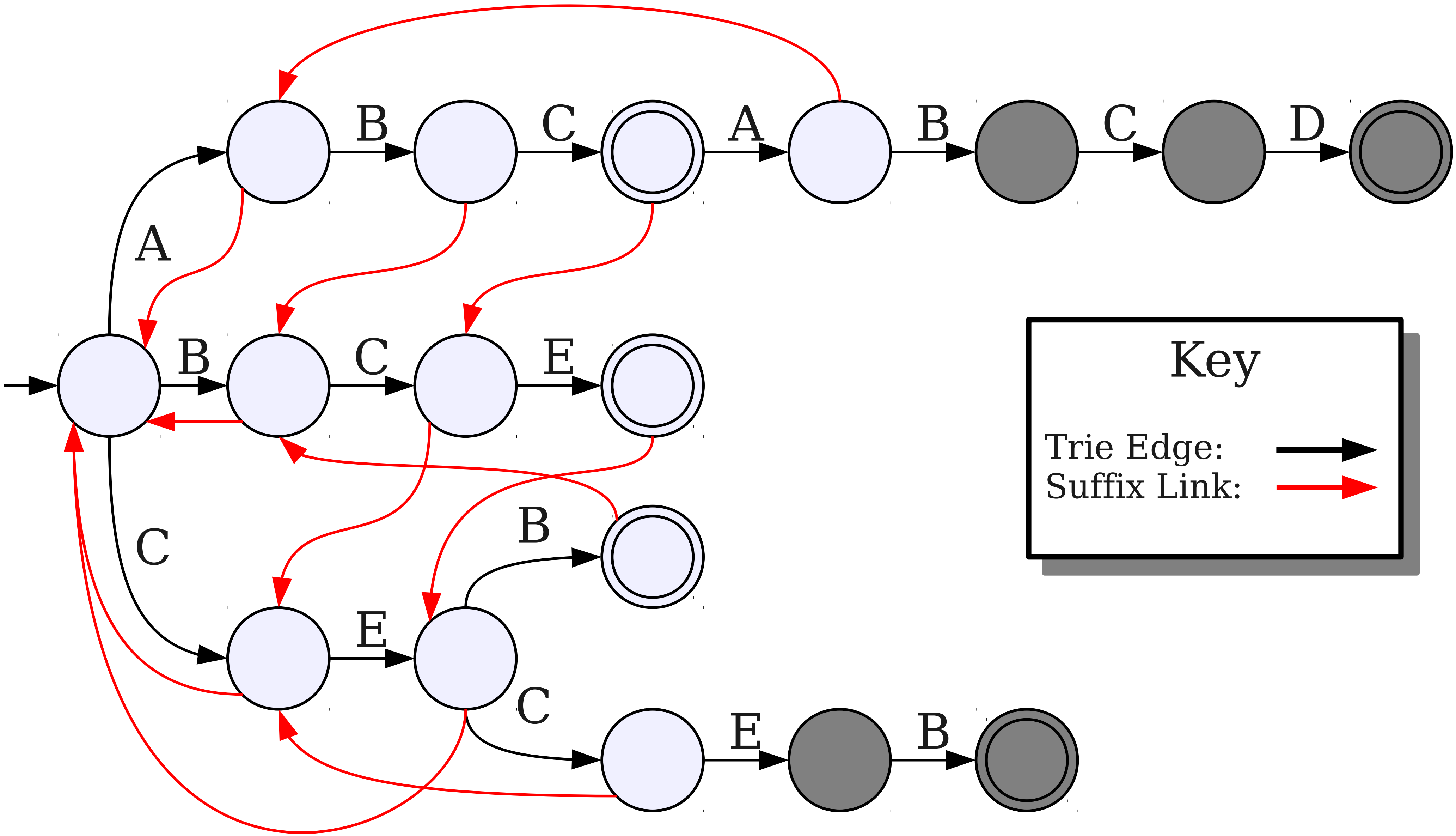
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



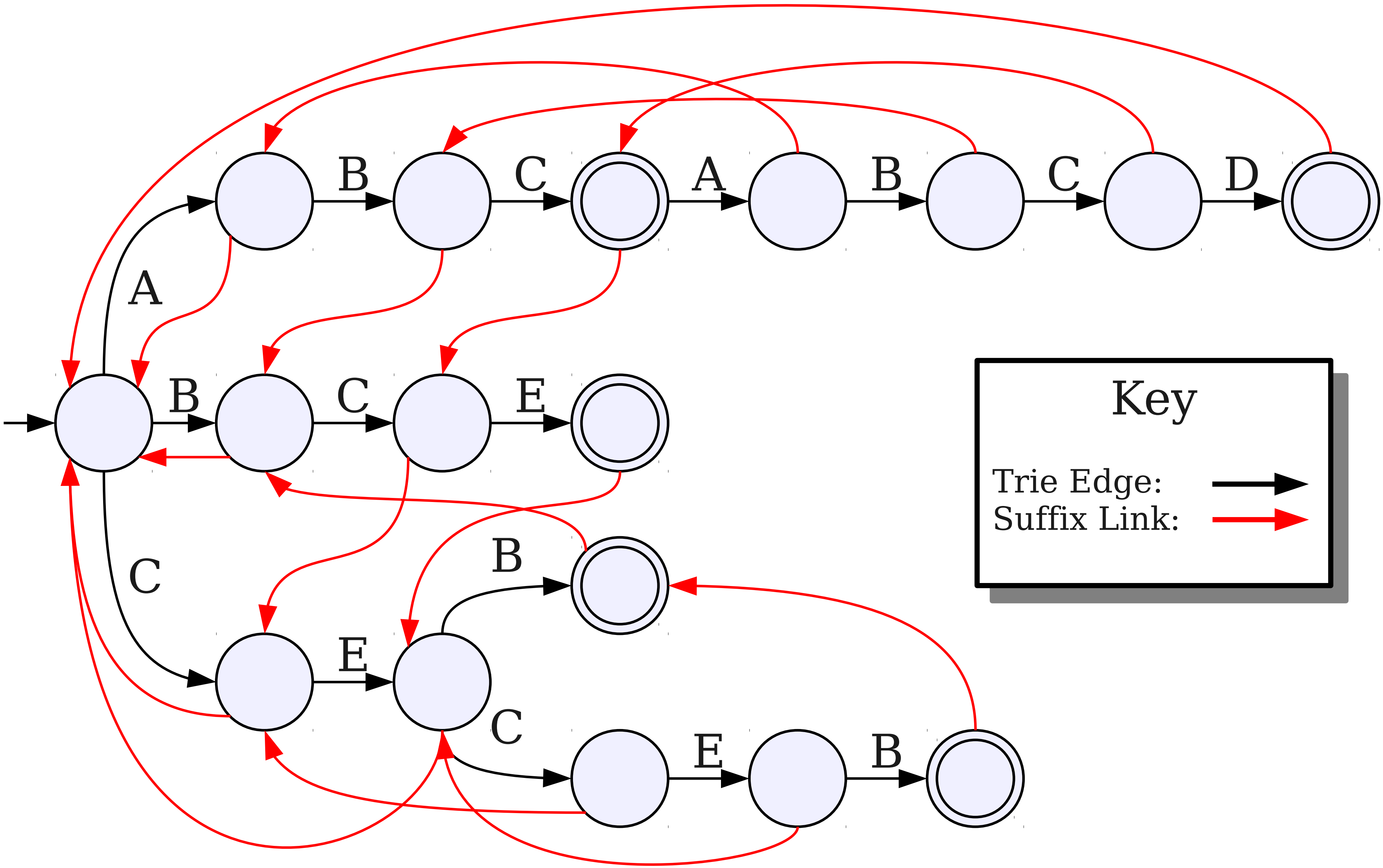
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC

Analyzing the Runtime

Claim: This algorithm constructs suffix links in the trie in time $O(n)$.

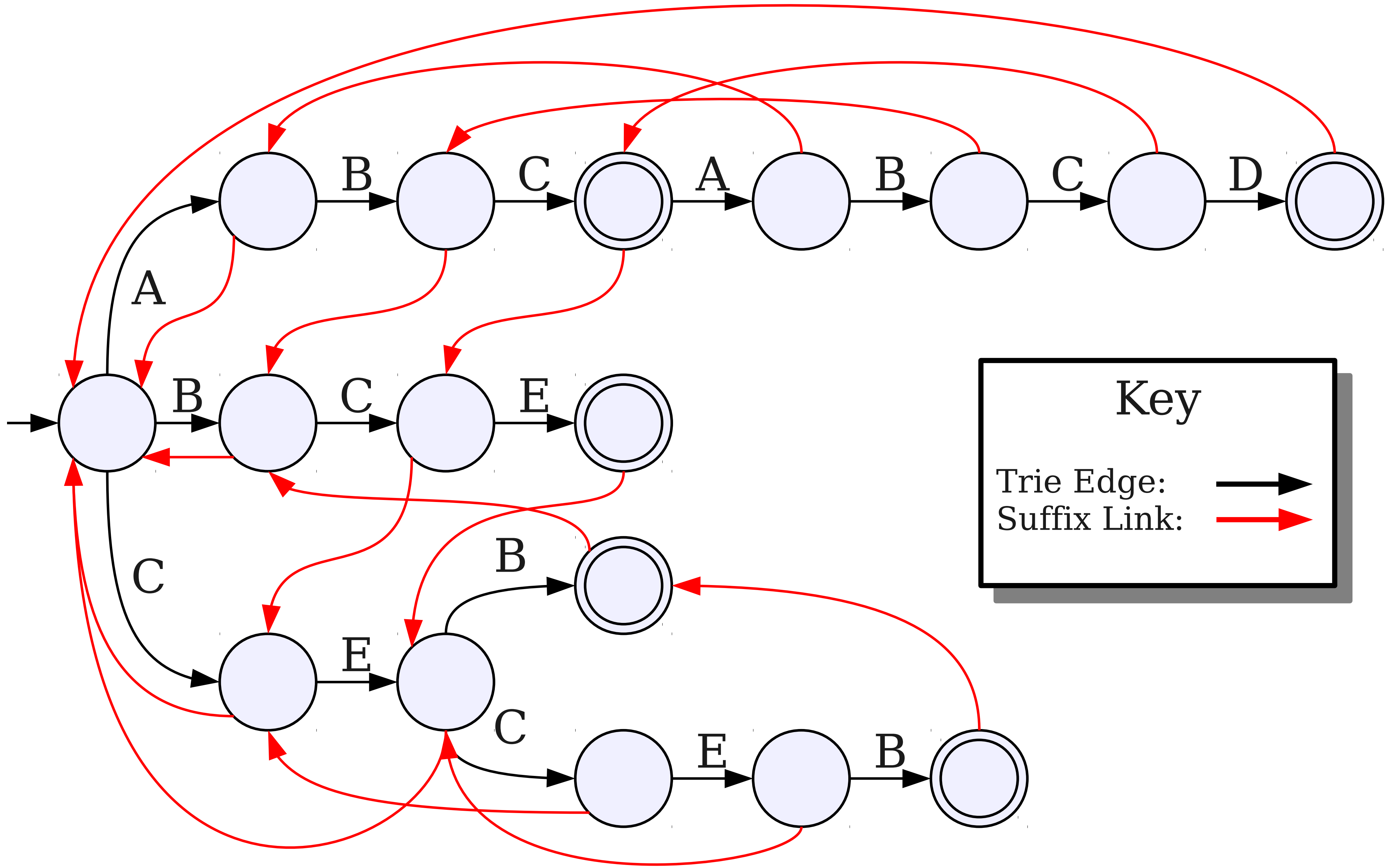
Proof: There are at most $O(n)$ nodes in the trie, so the breadth-first search will take time at most $O(n)$. Therefore, we have to bound the work done stepping backwards.

Focus on any individual word P_i. When processing nodes that make up the letters of P_i, the number of backward steps taken cannot exceed the number of forward steps taken, which is $O(|P_i|)$.

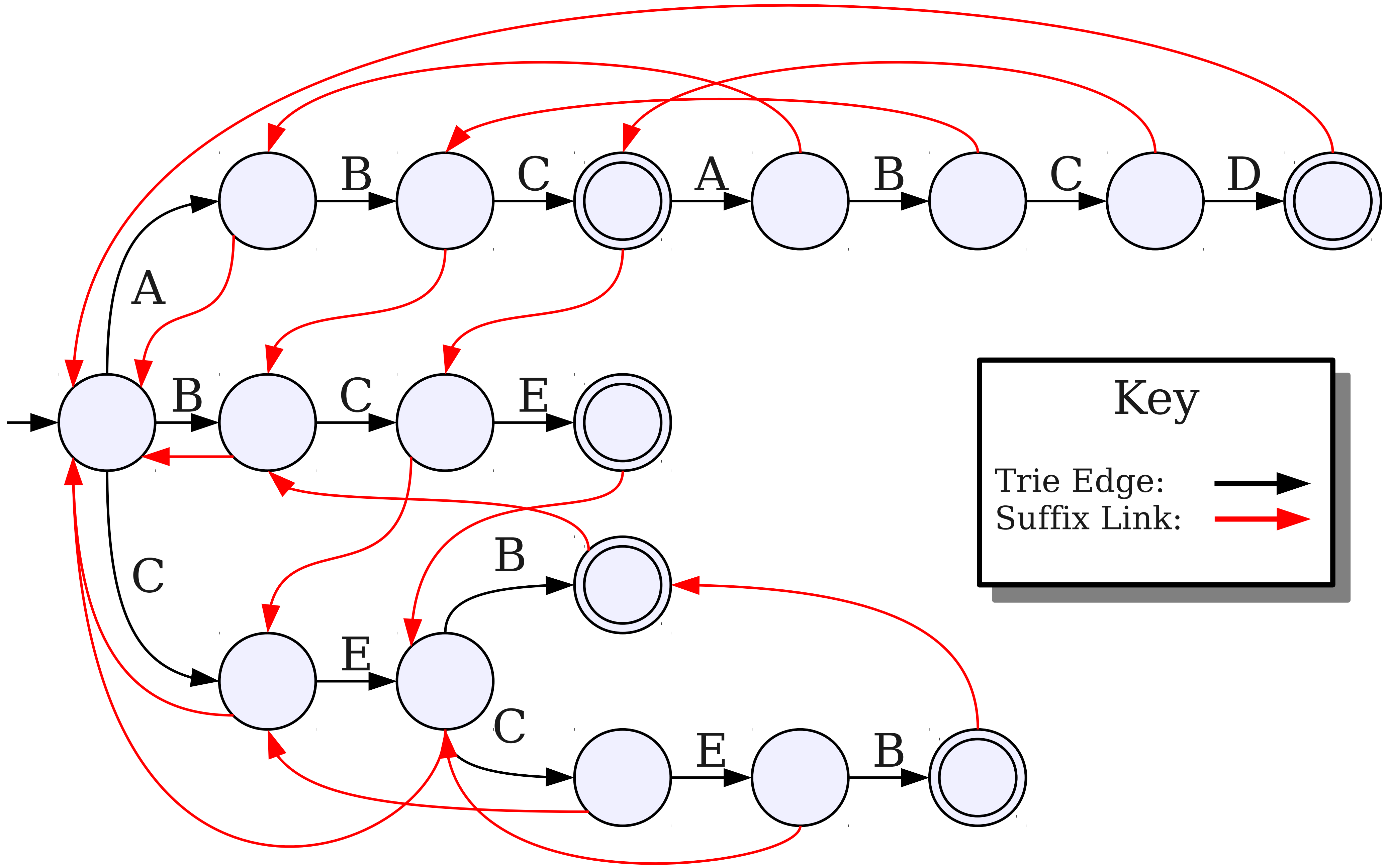
Summing across all words, the total number of backward steps is therefore $O(n)$. ■

The Story So Far

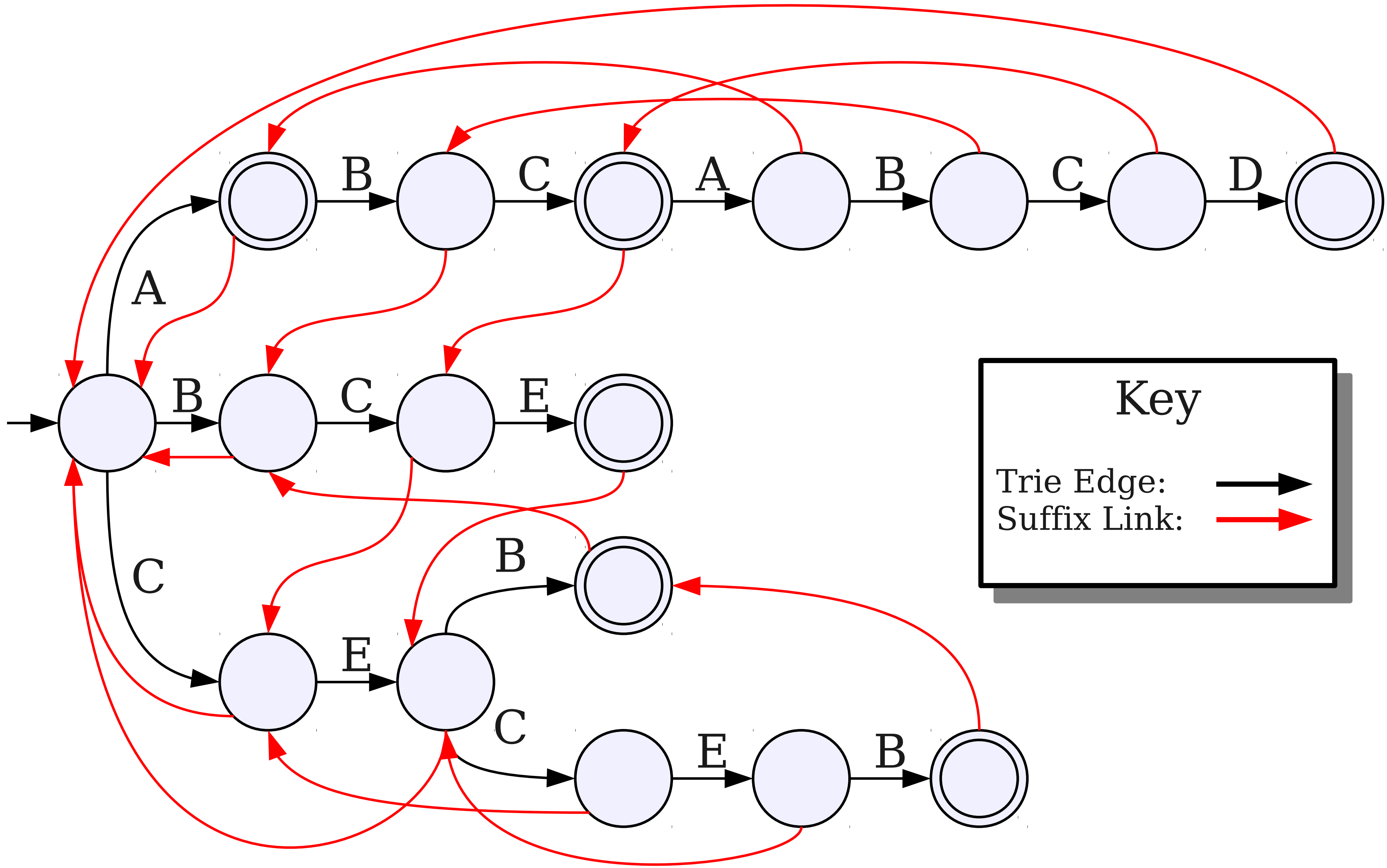
- We can construct our trie, augmented with suffix links, in time O(n).
- Once we have the trie, we can scan over a string in time O(m).
- **Catch from before:** We still don't have a way to identify all the substrings we find.
- Let's go fix that!



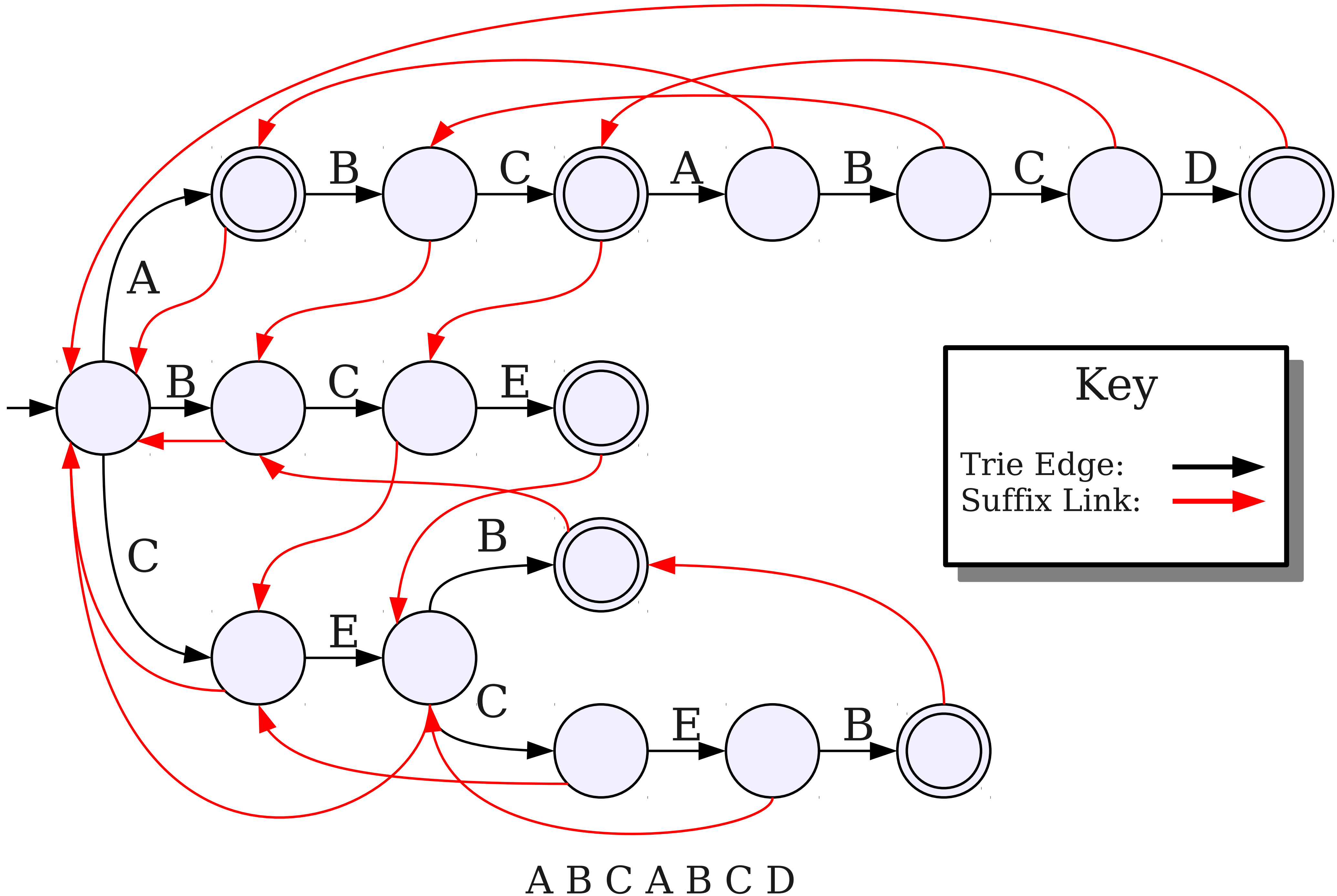
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC



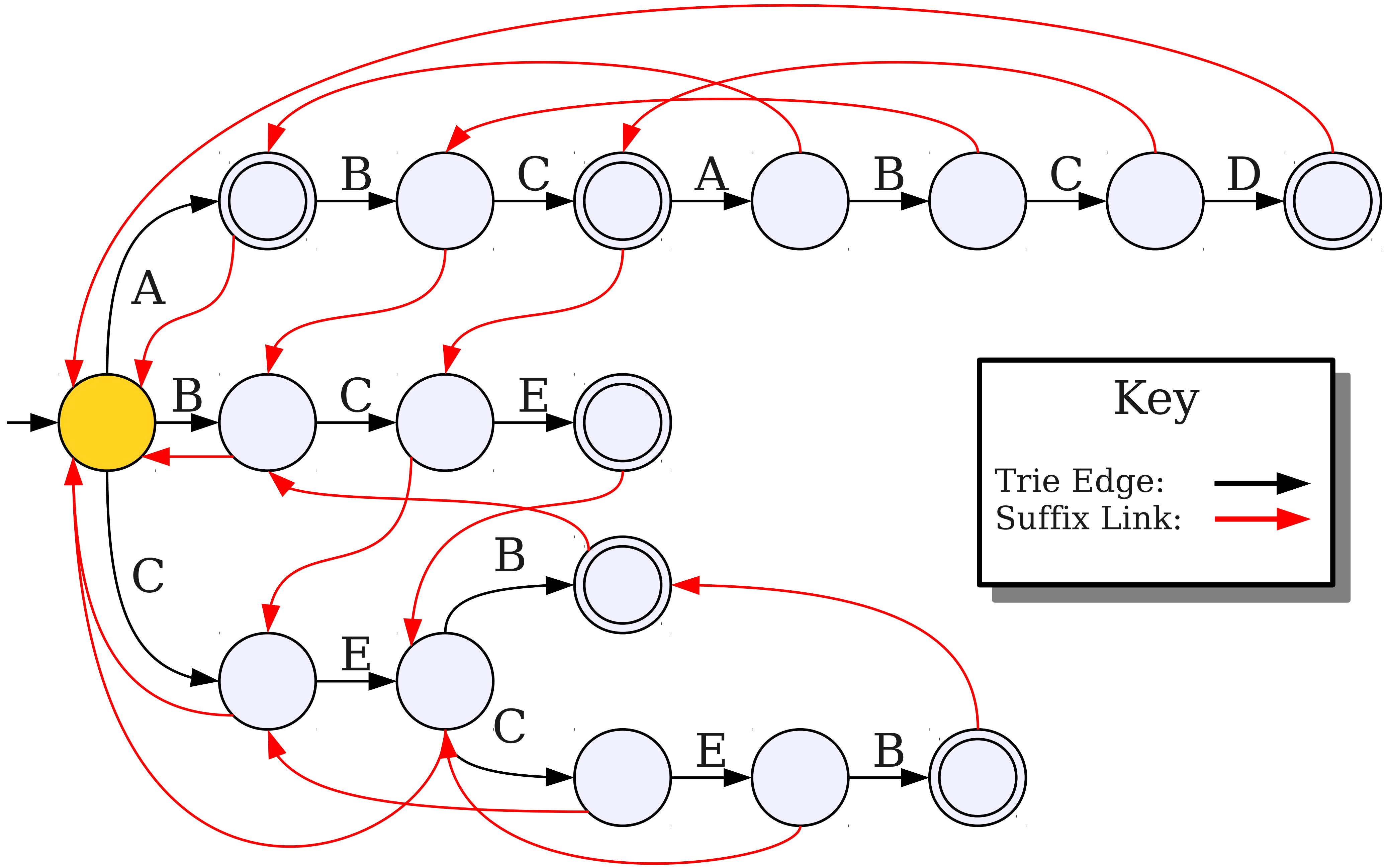
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC P_6 = A



P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC P_6 = *A*

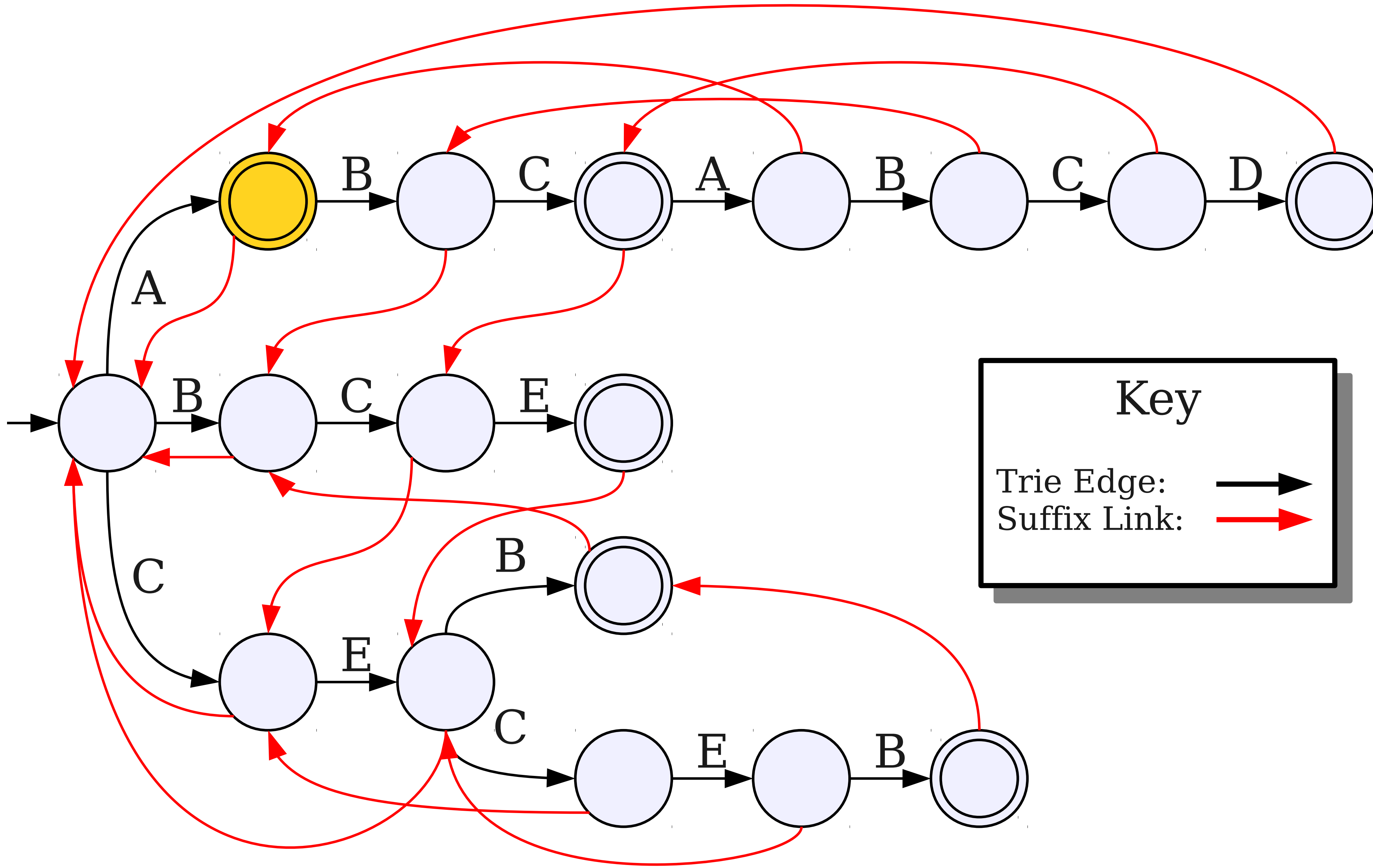


P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC $P_6 = A$



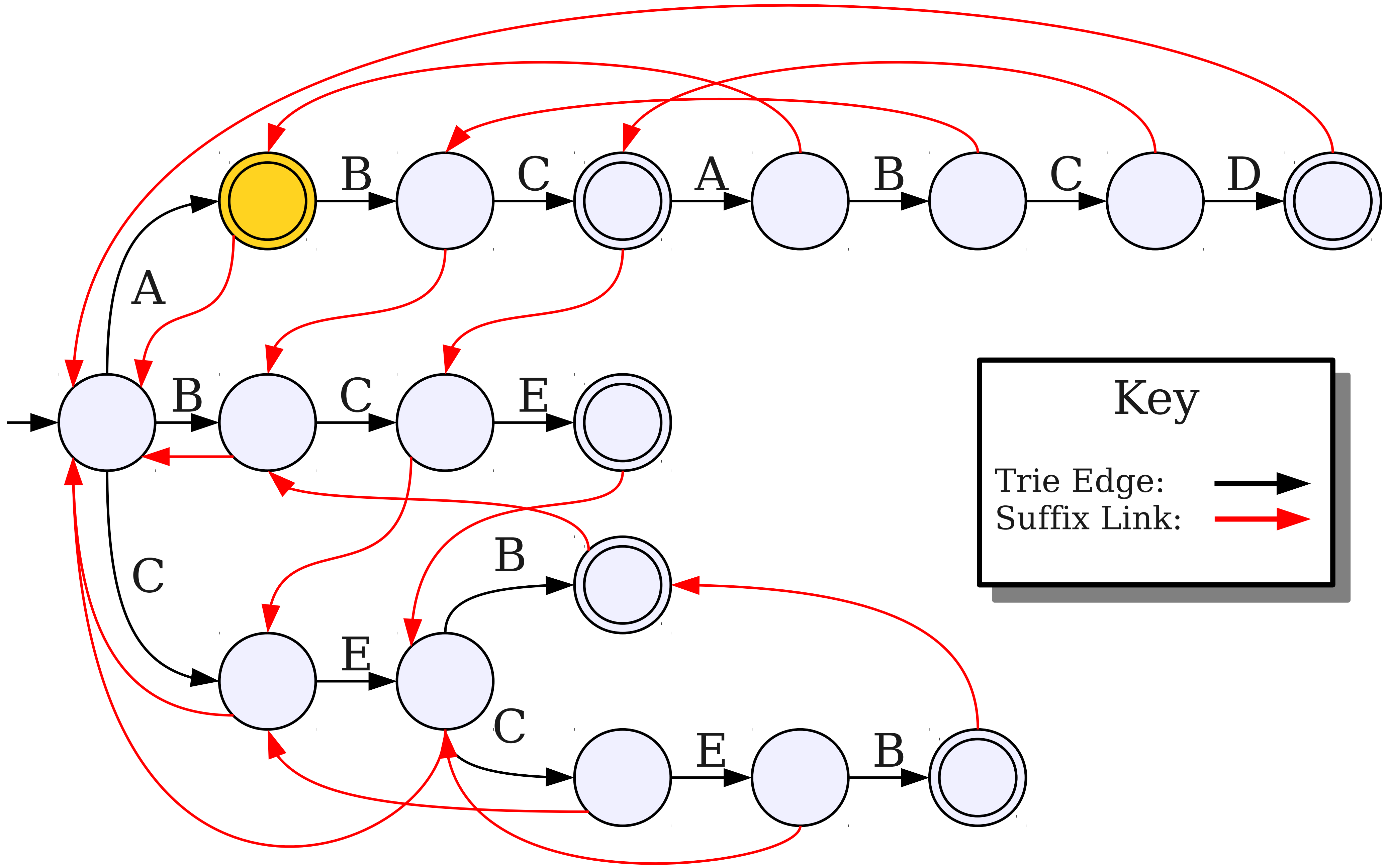
A B C A B C D

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC P_6 = A



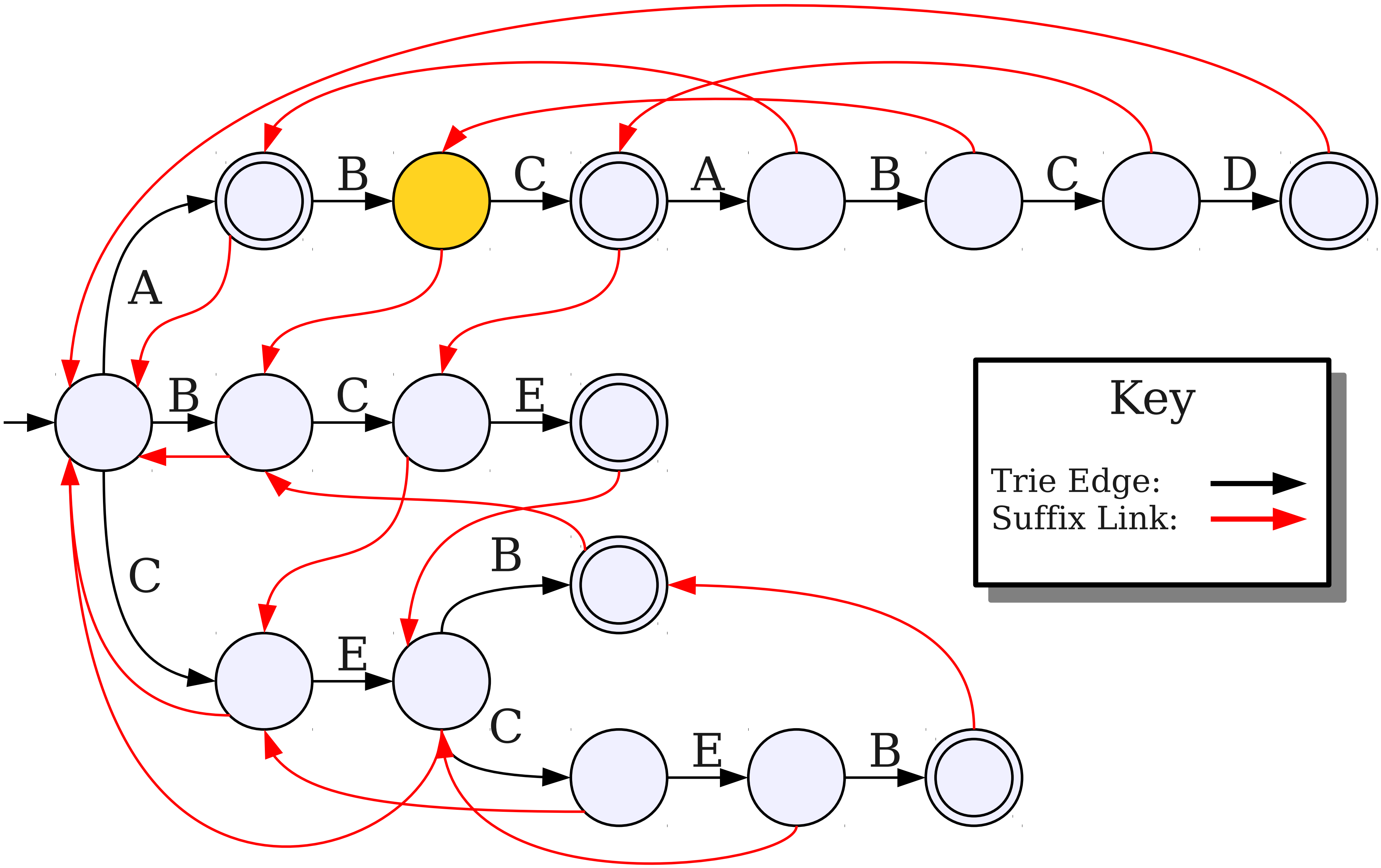
A B C A B C D

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC P_6 = A



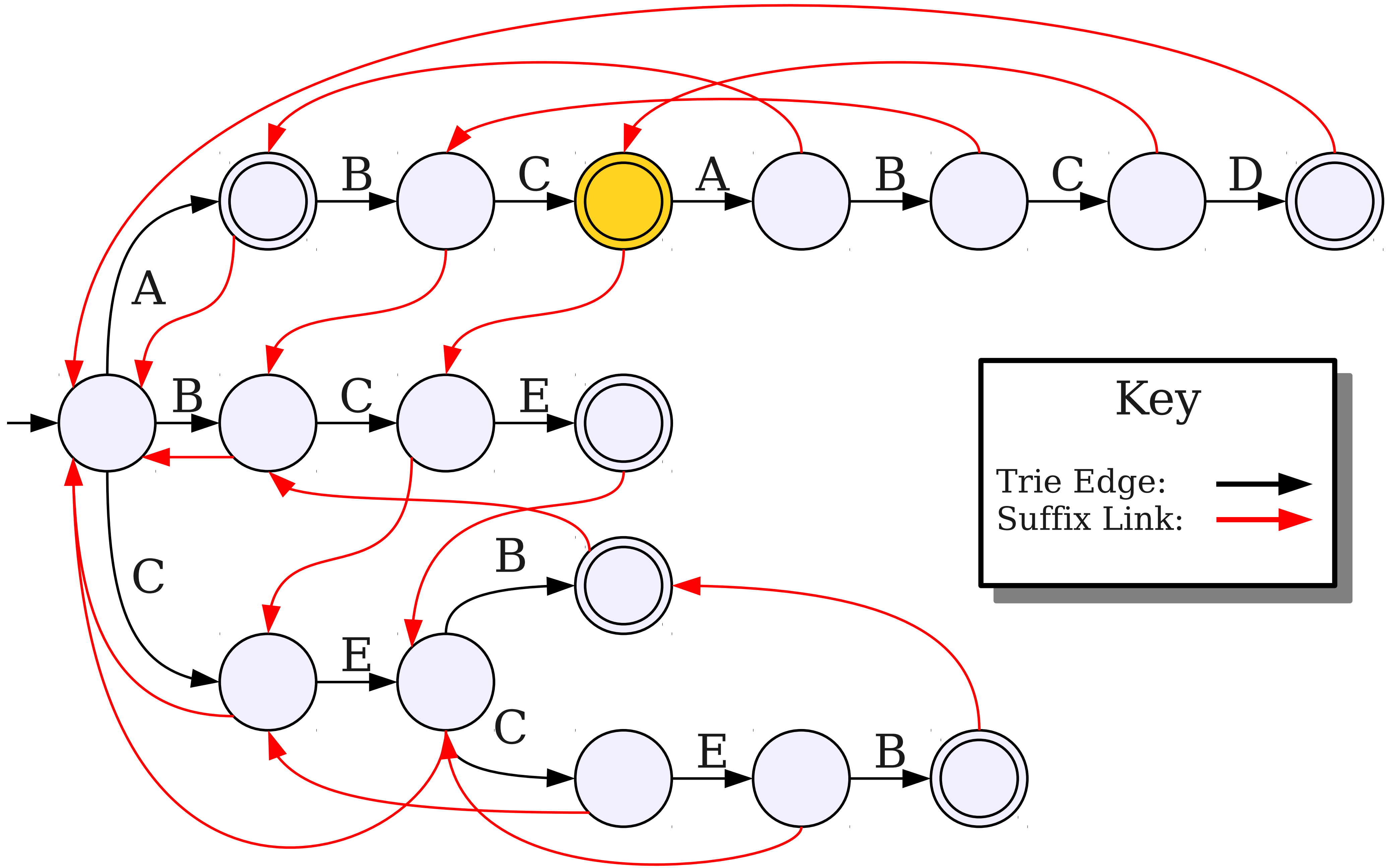
A B C A B C D

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC P_6 = *A*



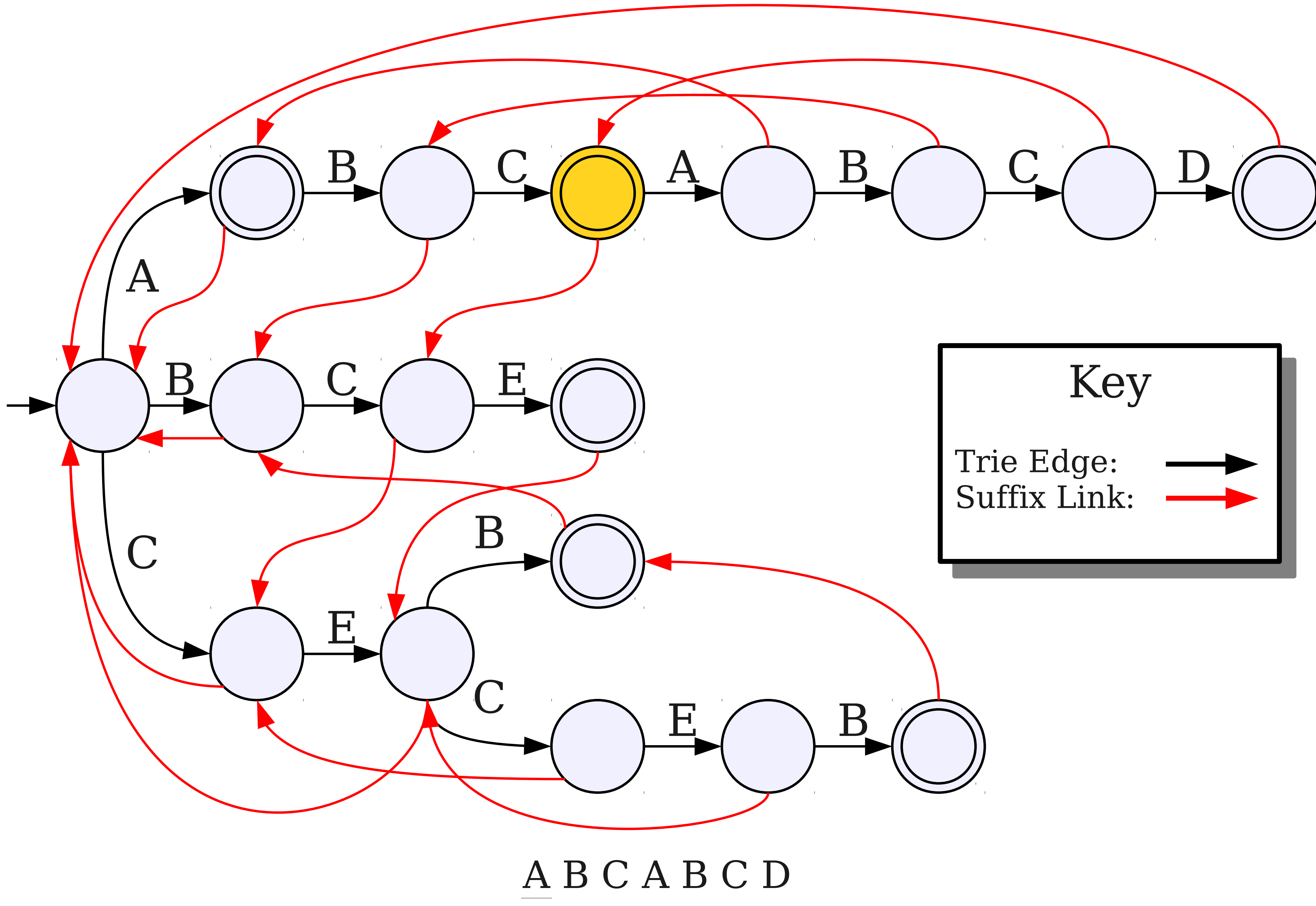
A B C A B C D

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC P_6 = A

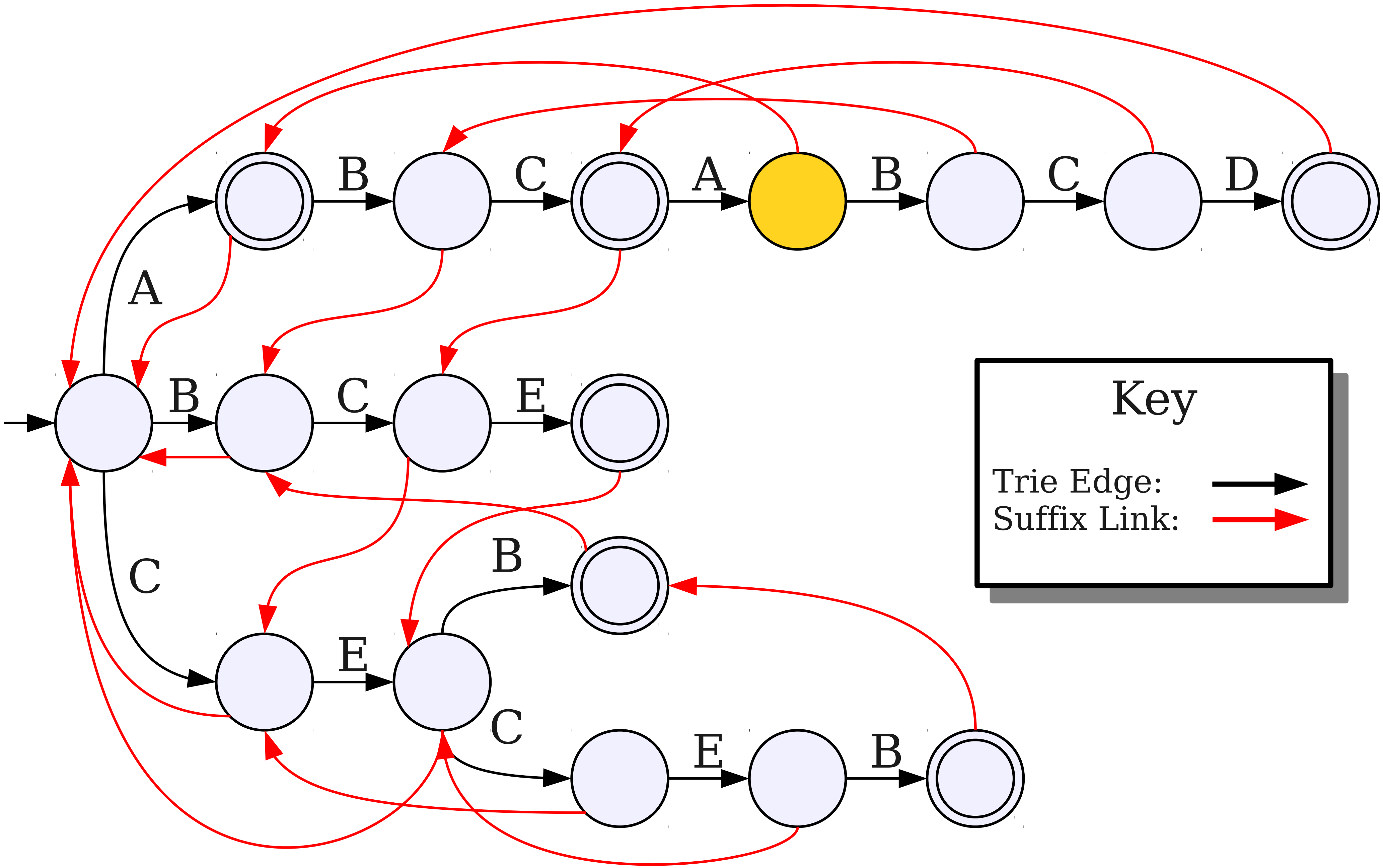


A B C A B C D

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC $P_6 = A$

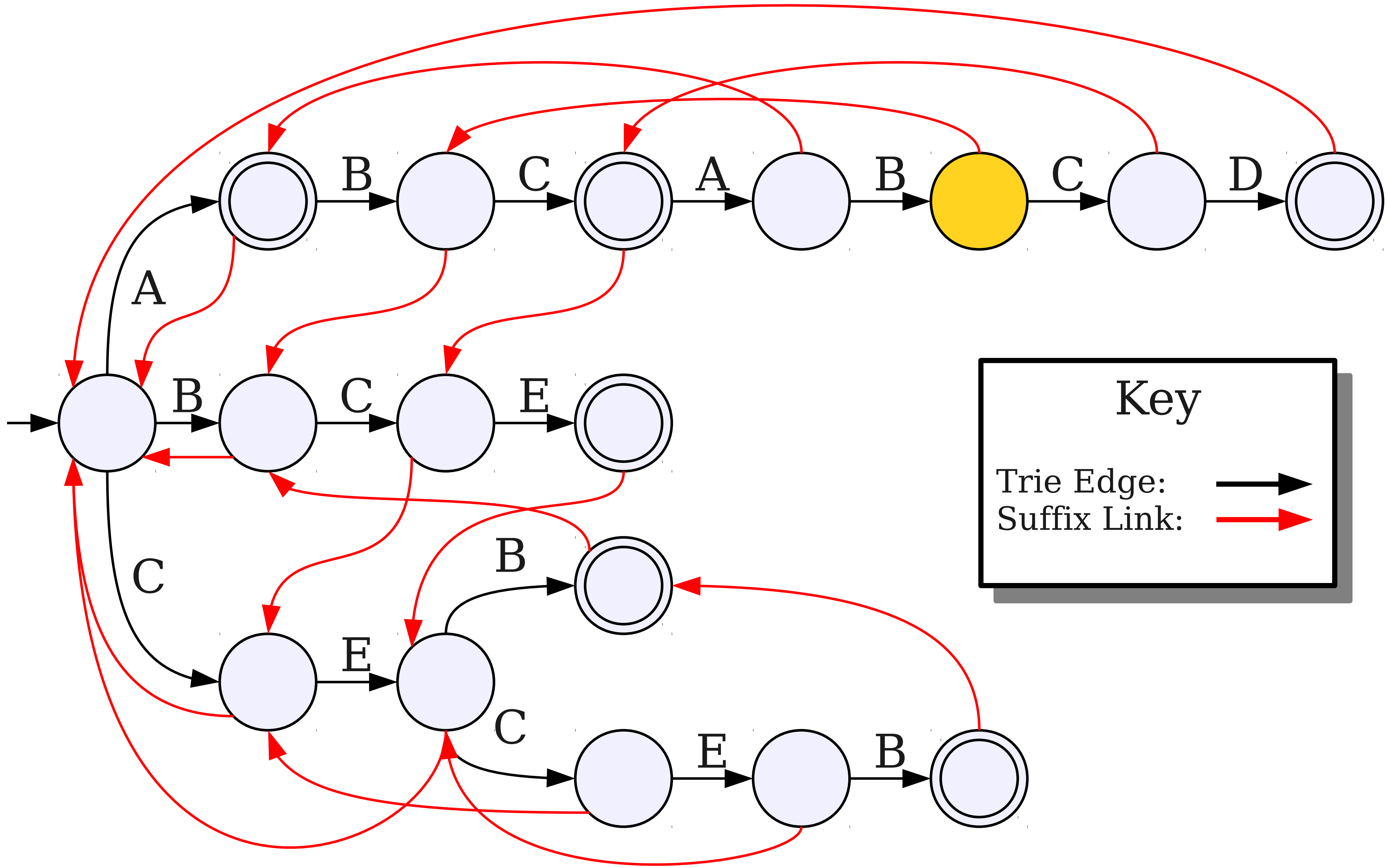


P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC P_6 = A



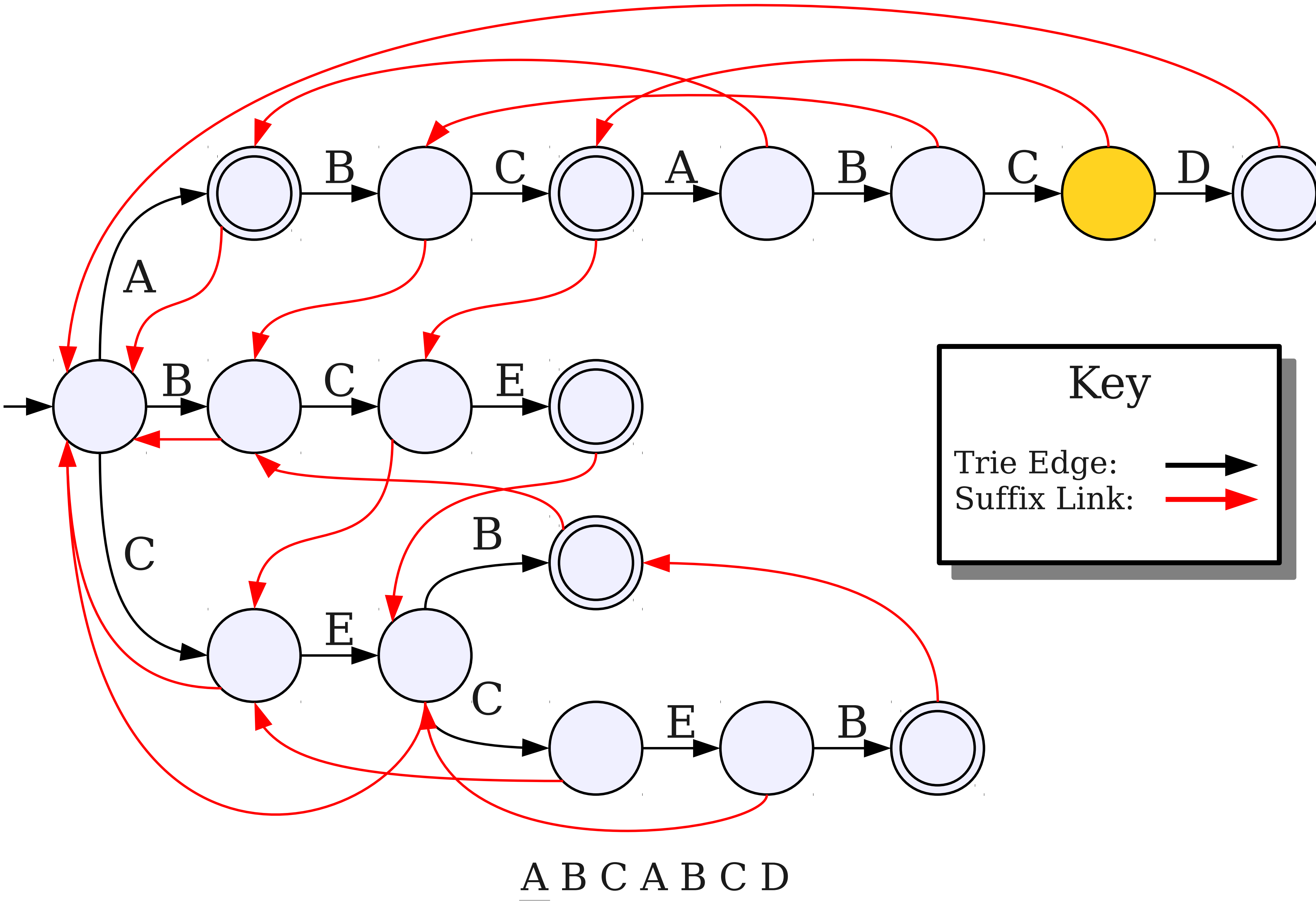
A B C A B C D

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC P_6 = *A*

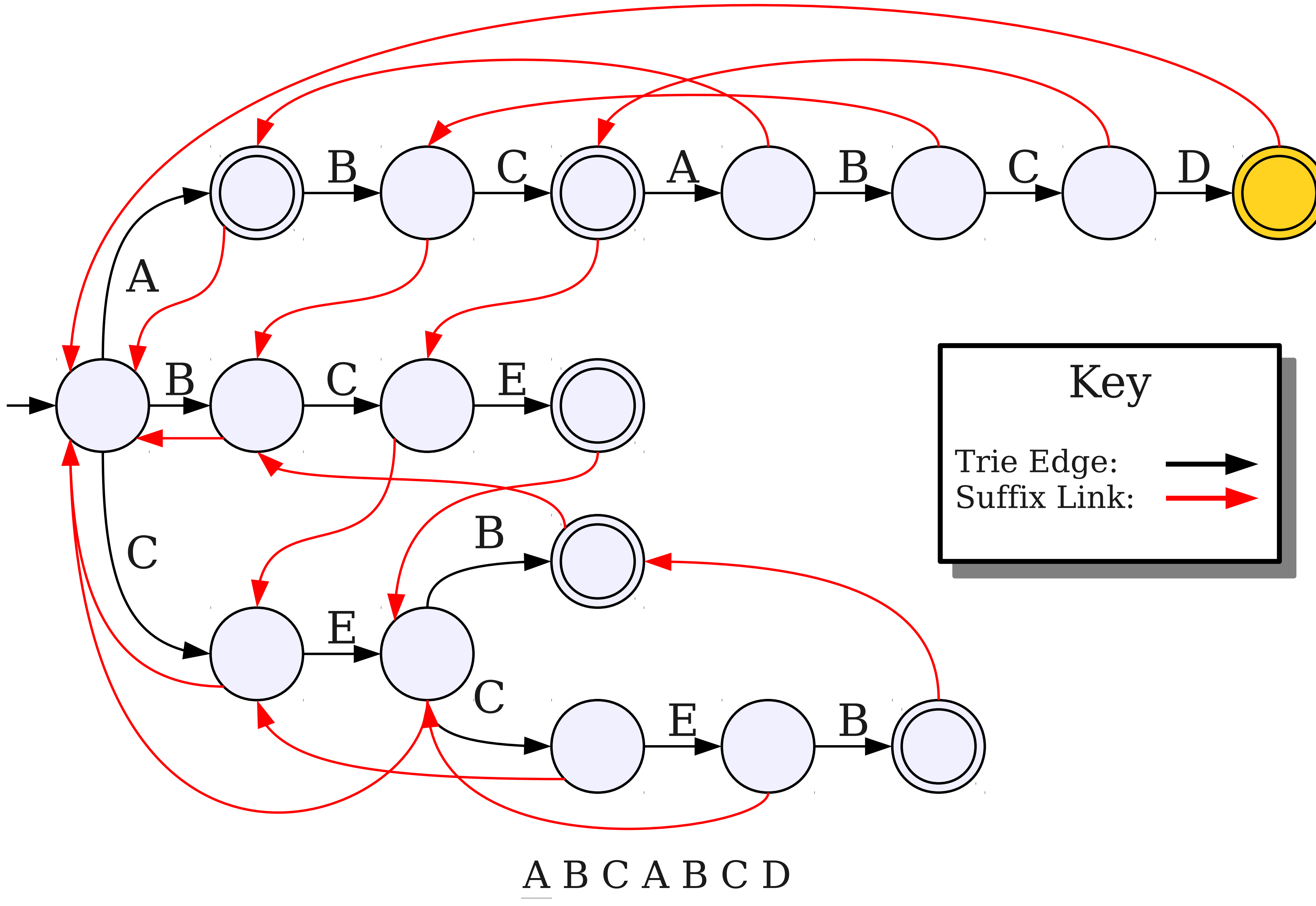


A B C A B C D

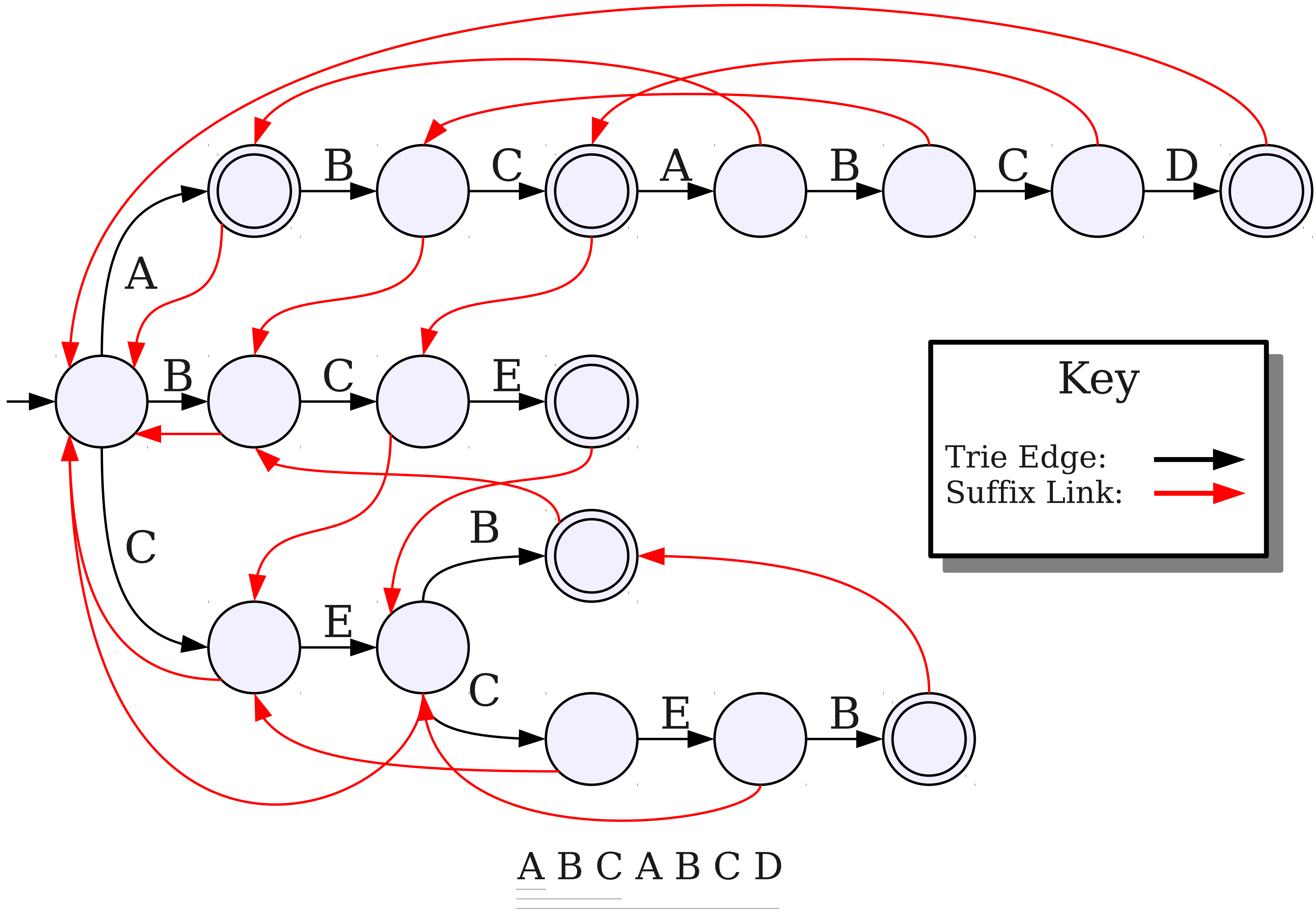
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC P_6 = A



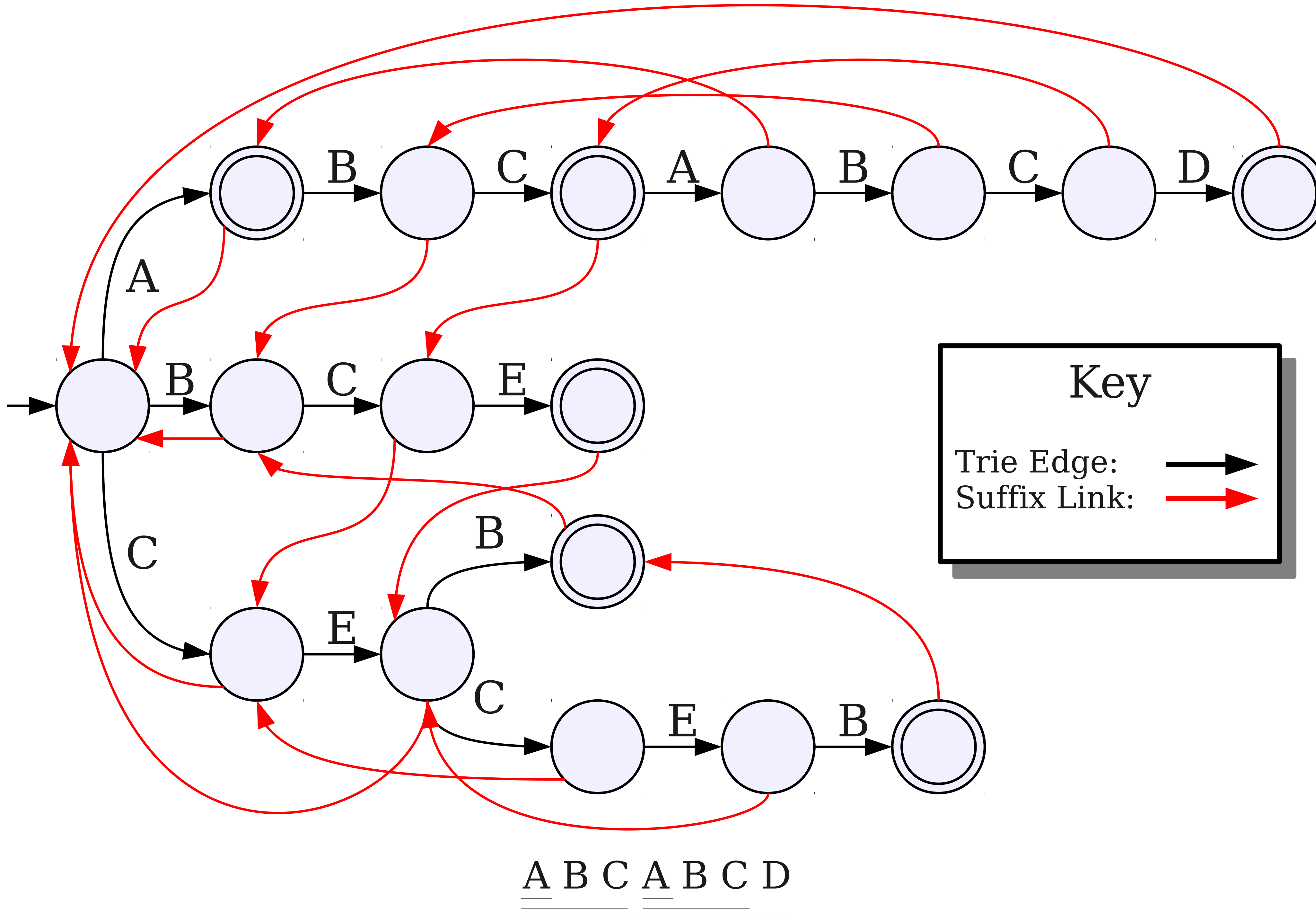
P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC P_6 = *A*



P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC P_6 = A



P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC P_6 = A



P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC P_6 = A

The Problem

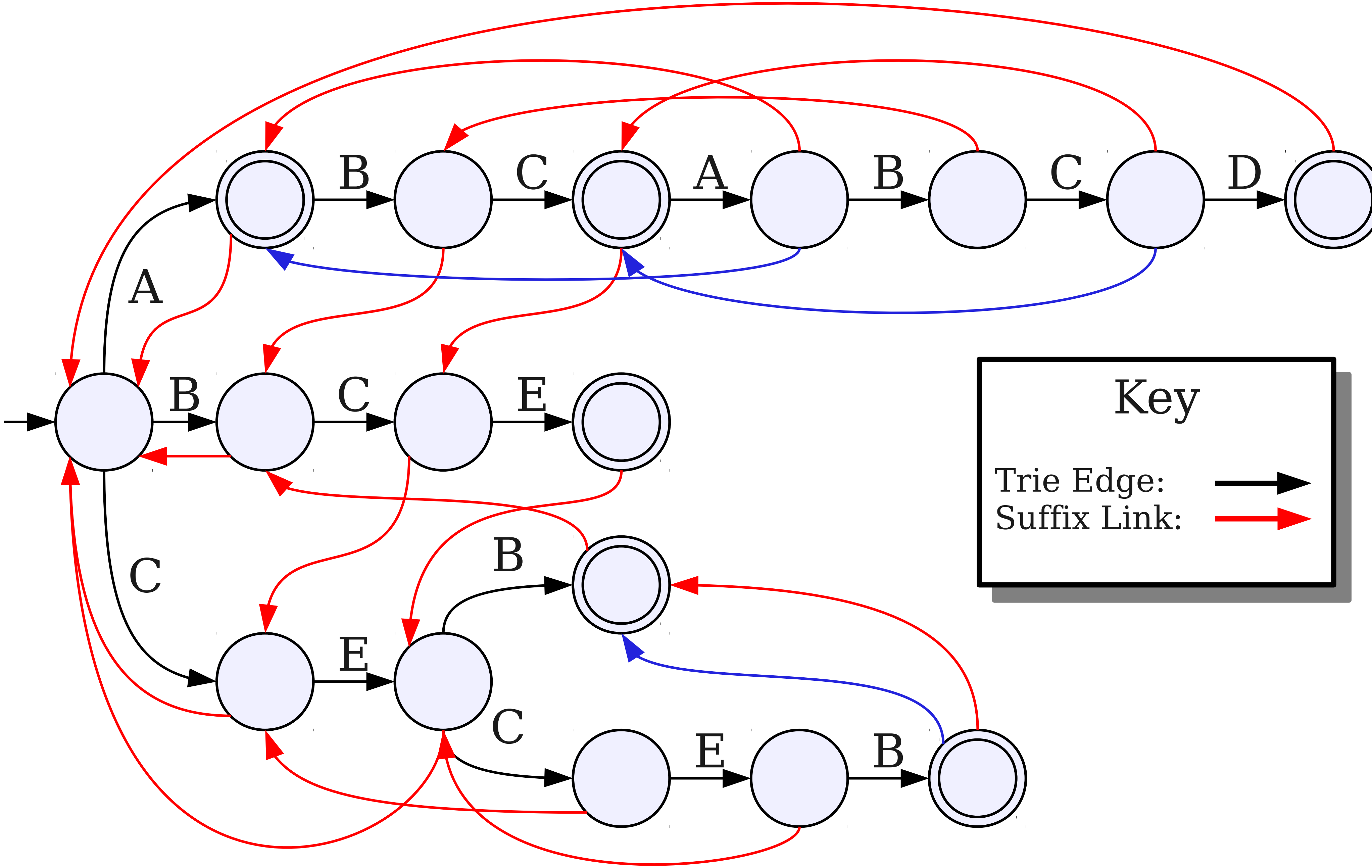
- Some pattern strings might be substrings of other pattern strings.
- Without taking this into account, our trie traversal will not find all matching substrings.
- Can we fix this?

A Useful Observation

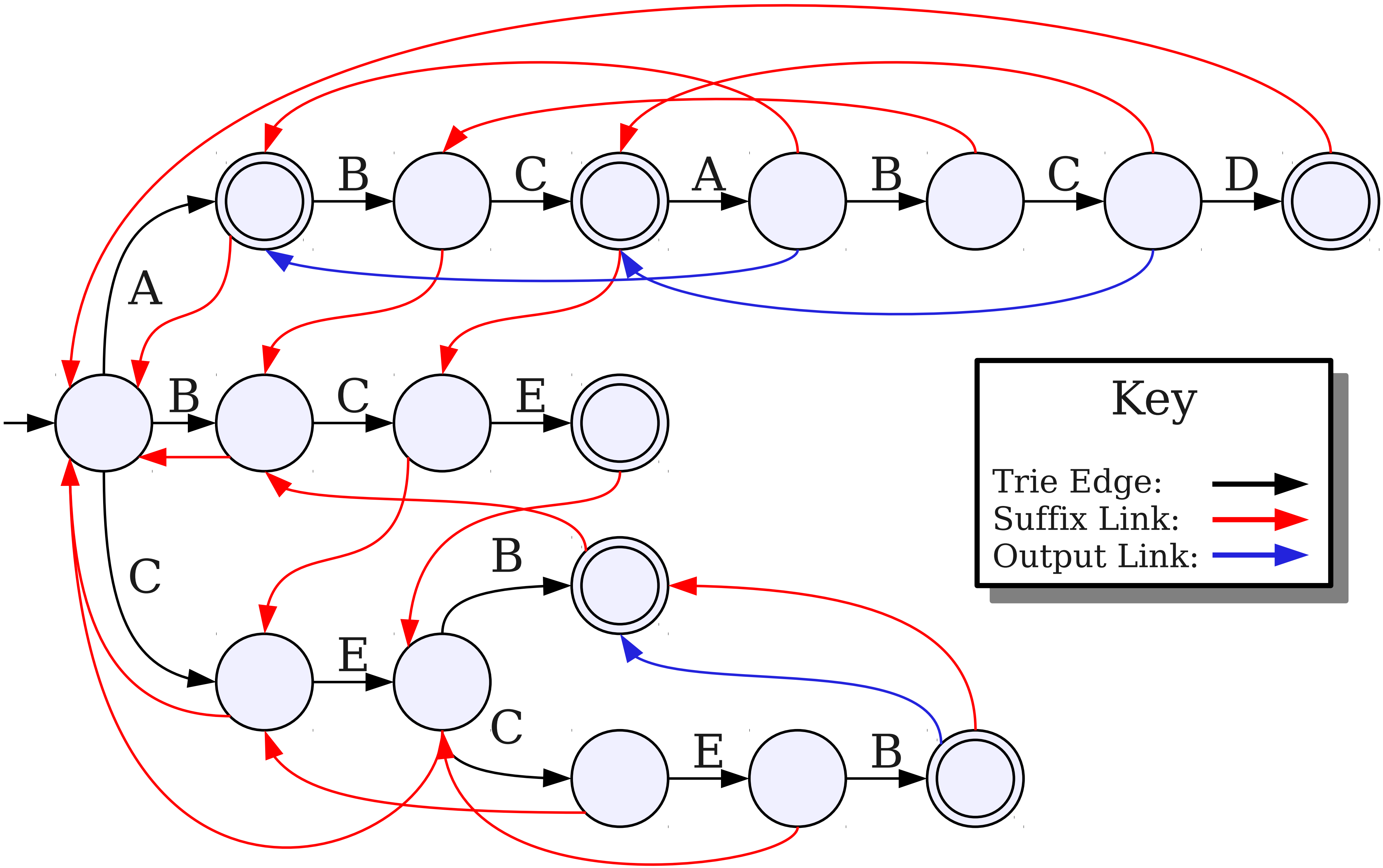
- **Fact:** If x is a substring of w, then x is a suffix of a prefix of w.
- **Proof:** Let $w = \alpha x \omega$. Then x is a suffix of the prefix αx.
- Each node in the trie corresponds to a prefix of some pattern string.
- Suffix links give us information about the suffixes of those strings.

Another Useful Observation

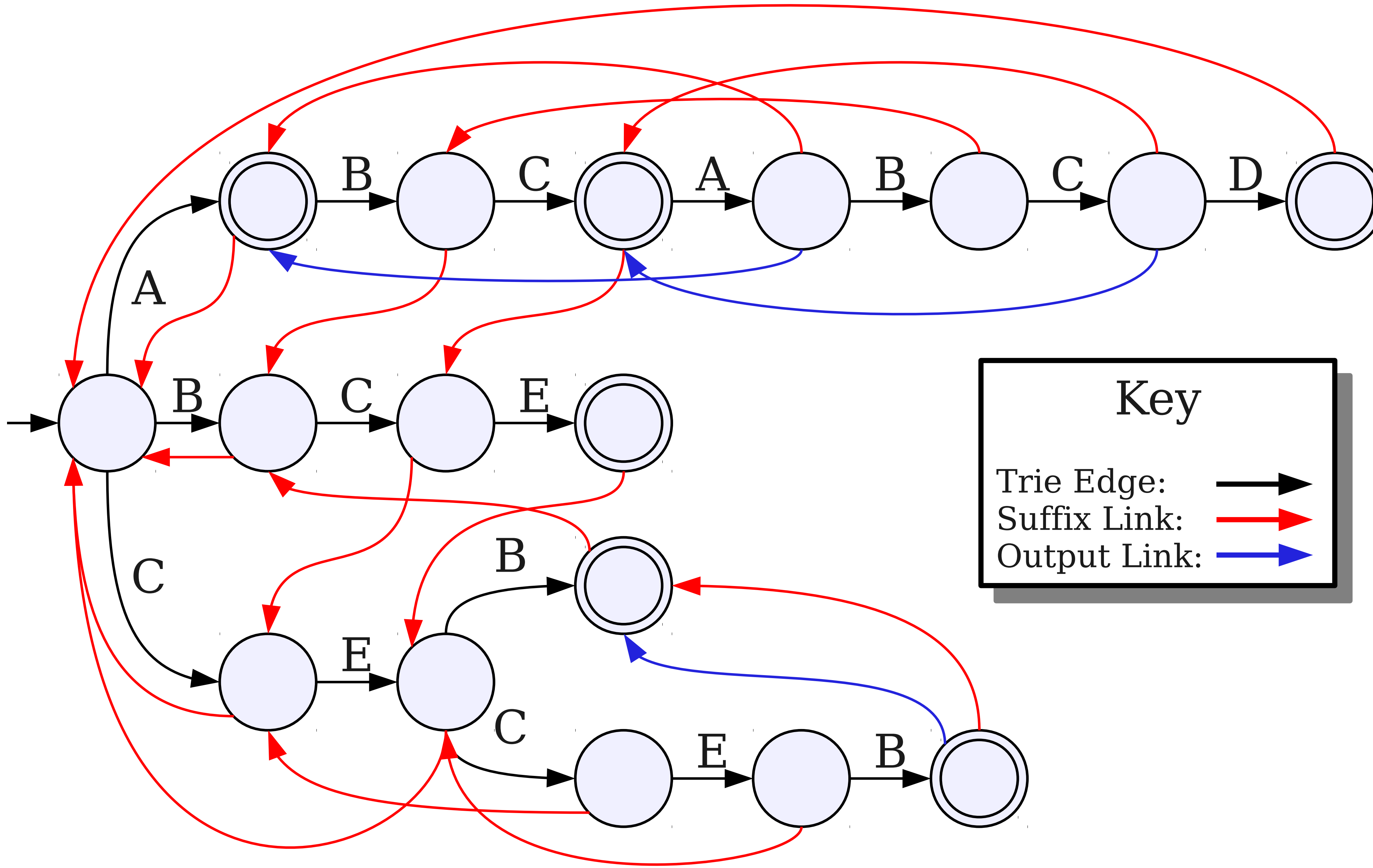
- **Fact:** Suppose that P_s and P_t are where $|P_s| > |P_t|$ and P_t is a suffix of P_s. Then any time P_s occurs, P_t occurs as well.
- This motivates the following idea:
  - Each node w in the trie may store an *output link* pointing to the longest pattern string that is a proper suffix of w.
  - Whenever we visit a node, we traverse backwards through the output links to find all matches.



P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC $P_6 = A$

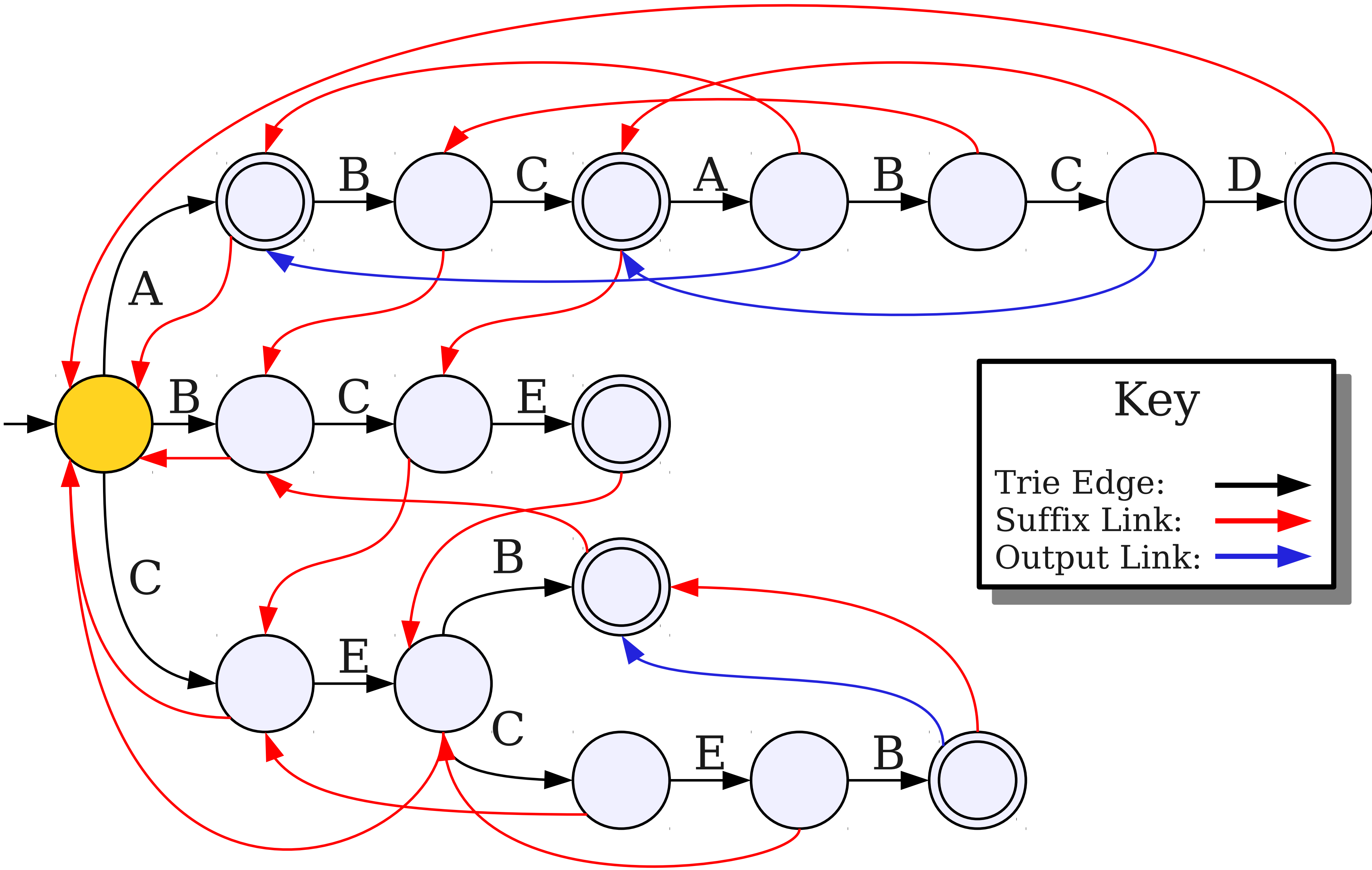


P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC P_6 = *A*



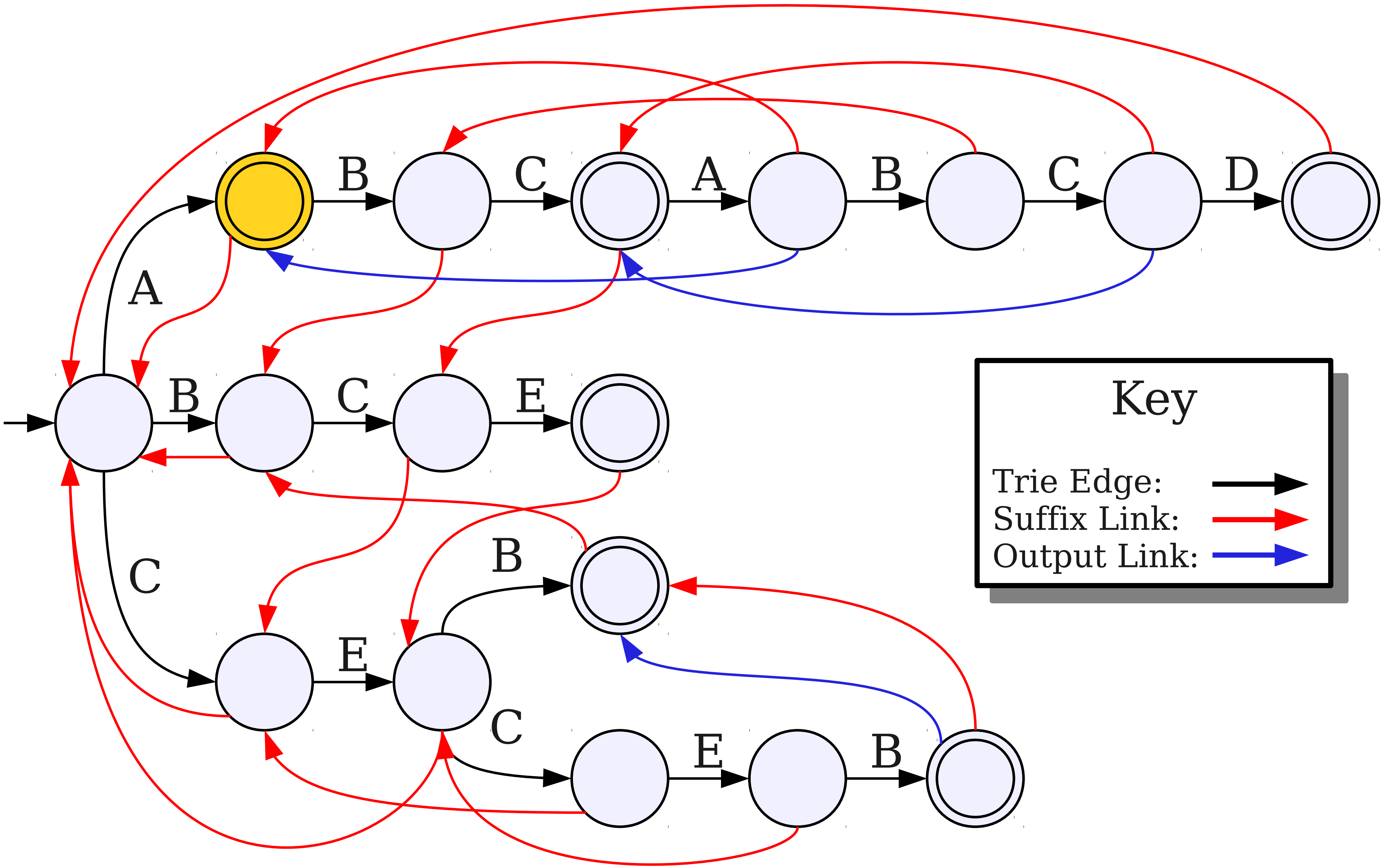
A B C A B C D

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC P_6 = A



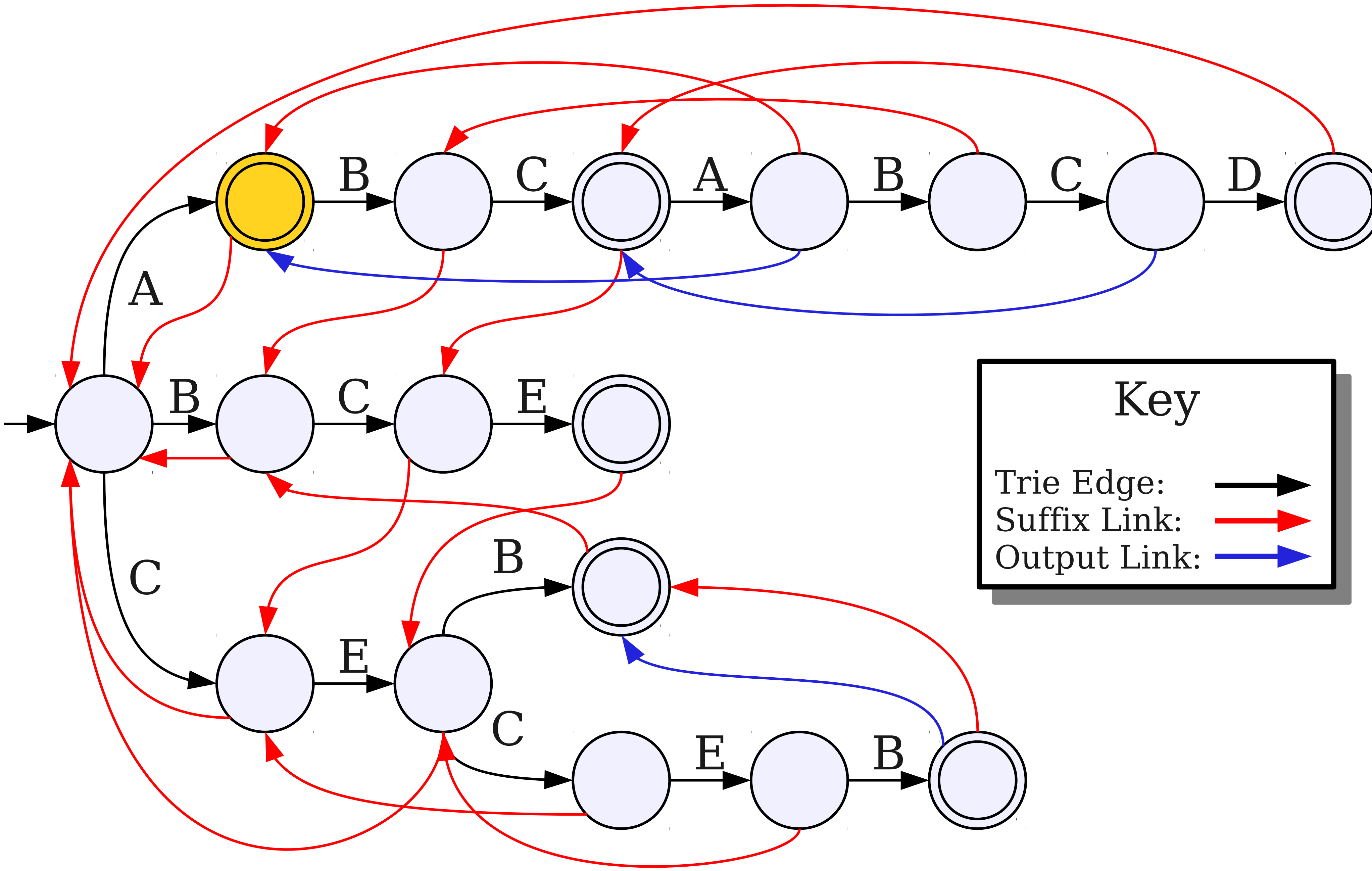
A B C A B C D

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC P_6 = A



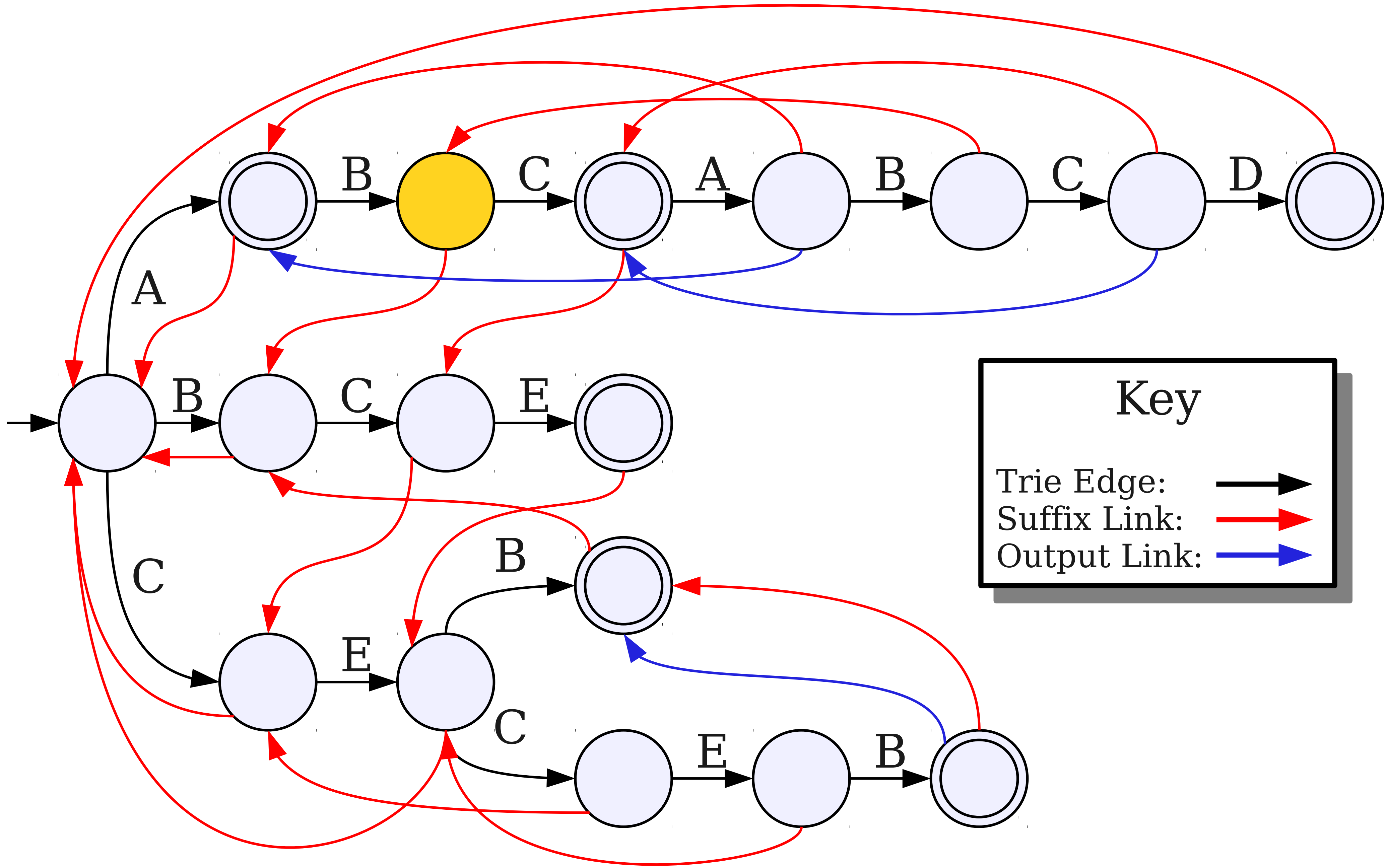
A B C A B C D

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC P_6 = A



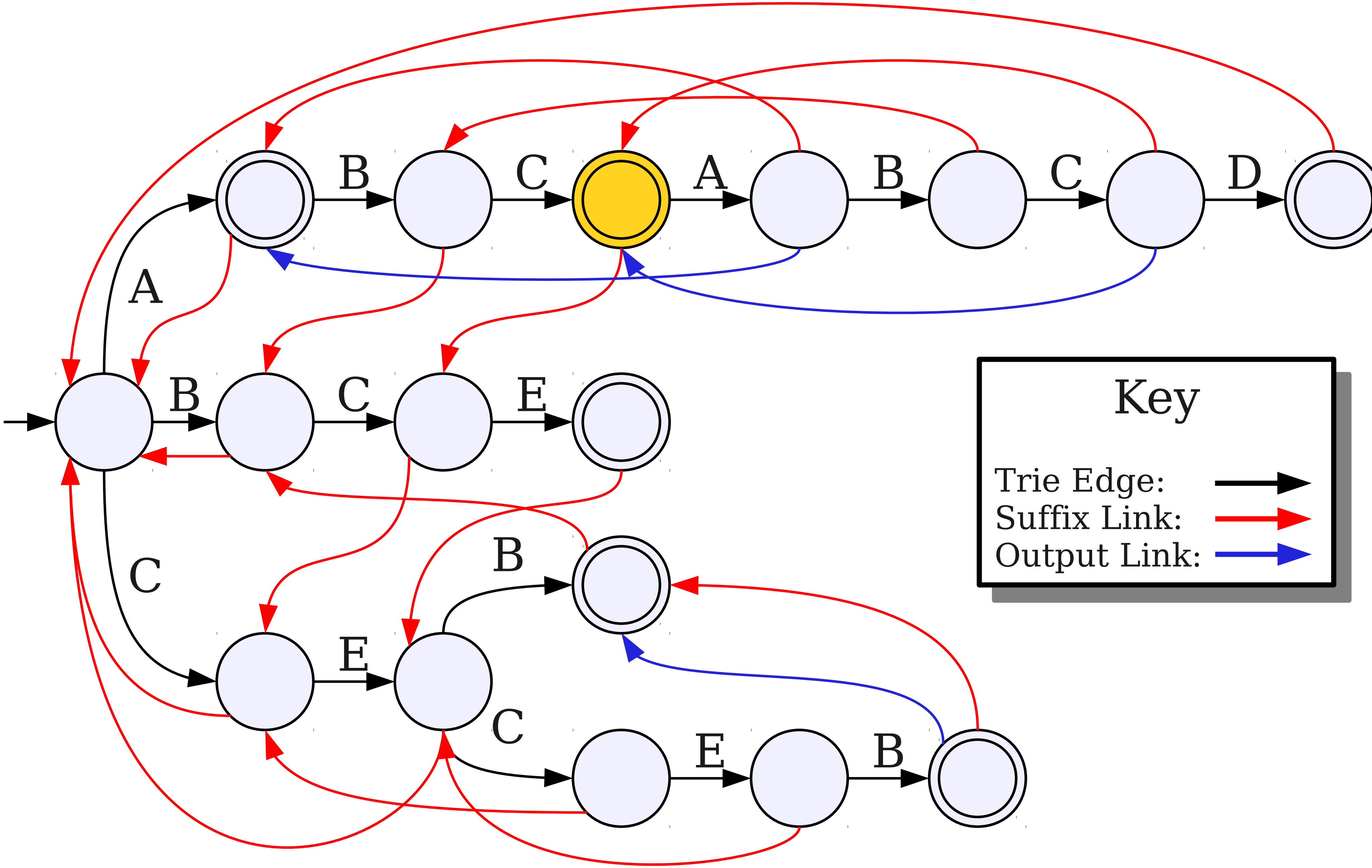
A B C A B C D

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC $P_6 = A$



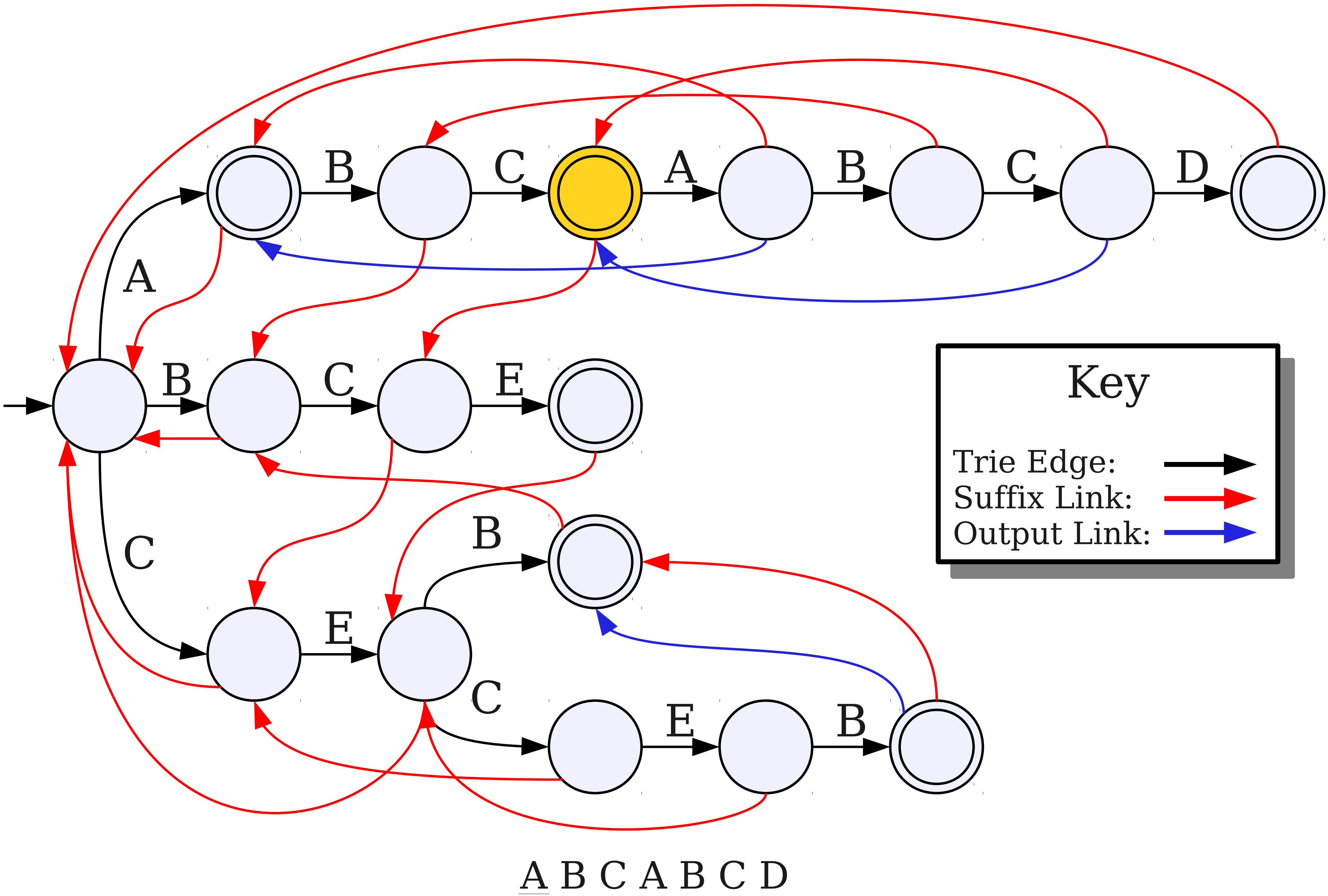
A B C A B C D

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC $P_6 = A$

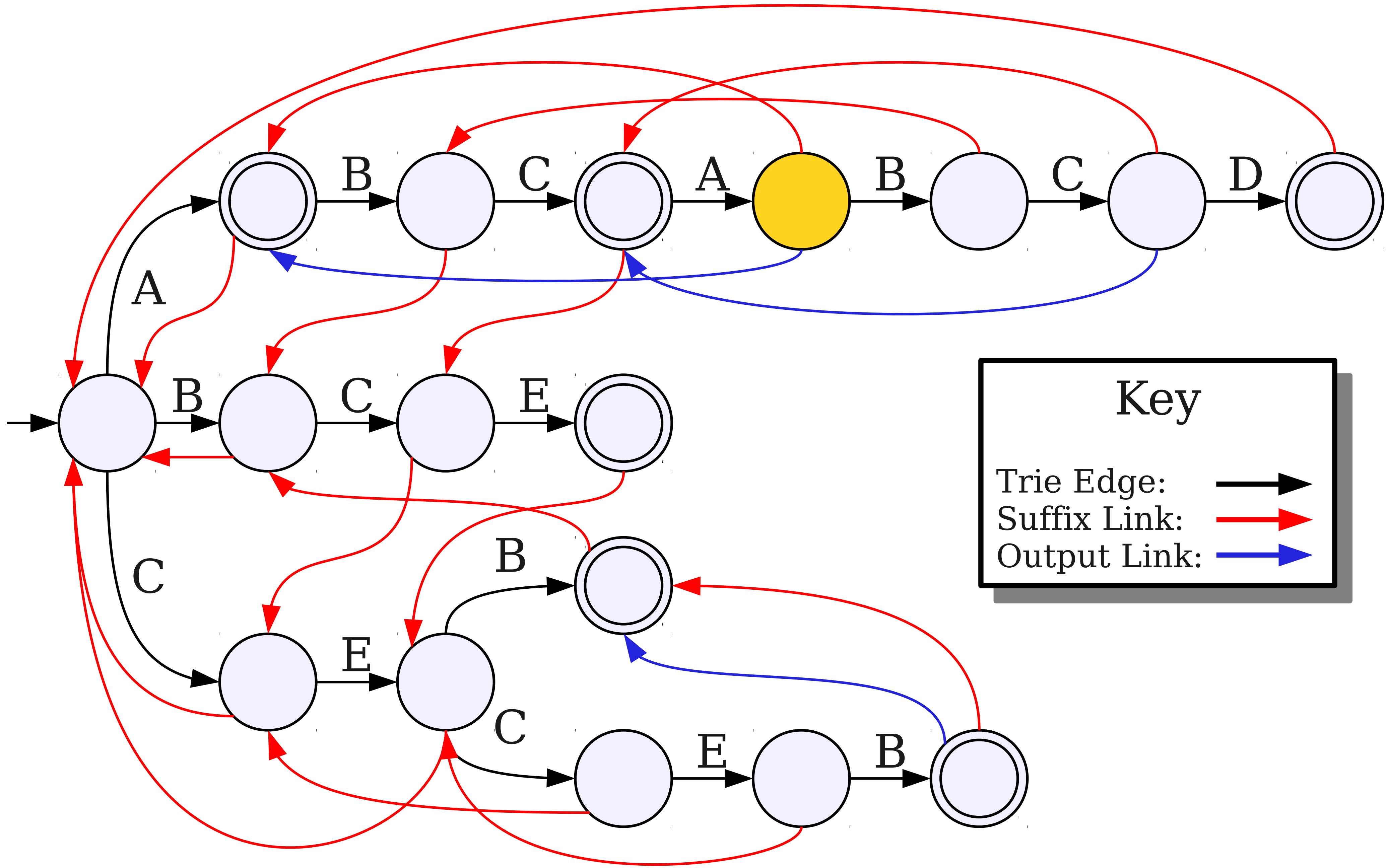


A B C A B C D

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC P_6 = A

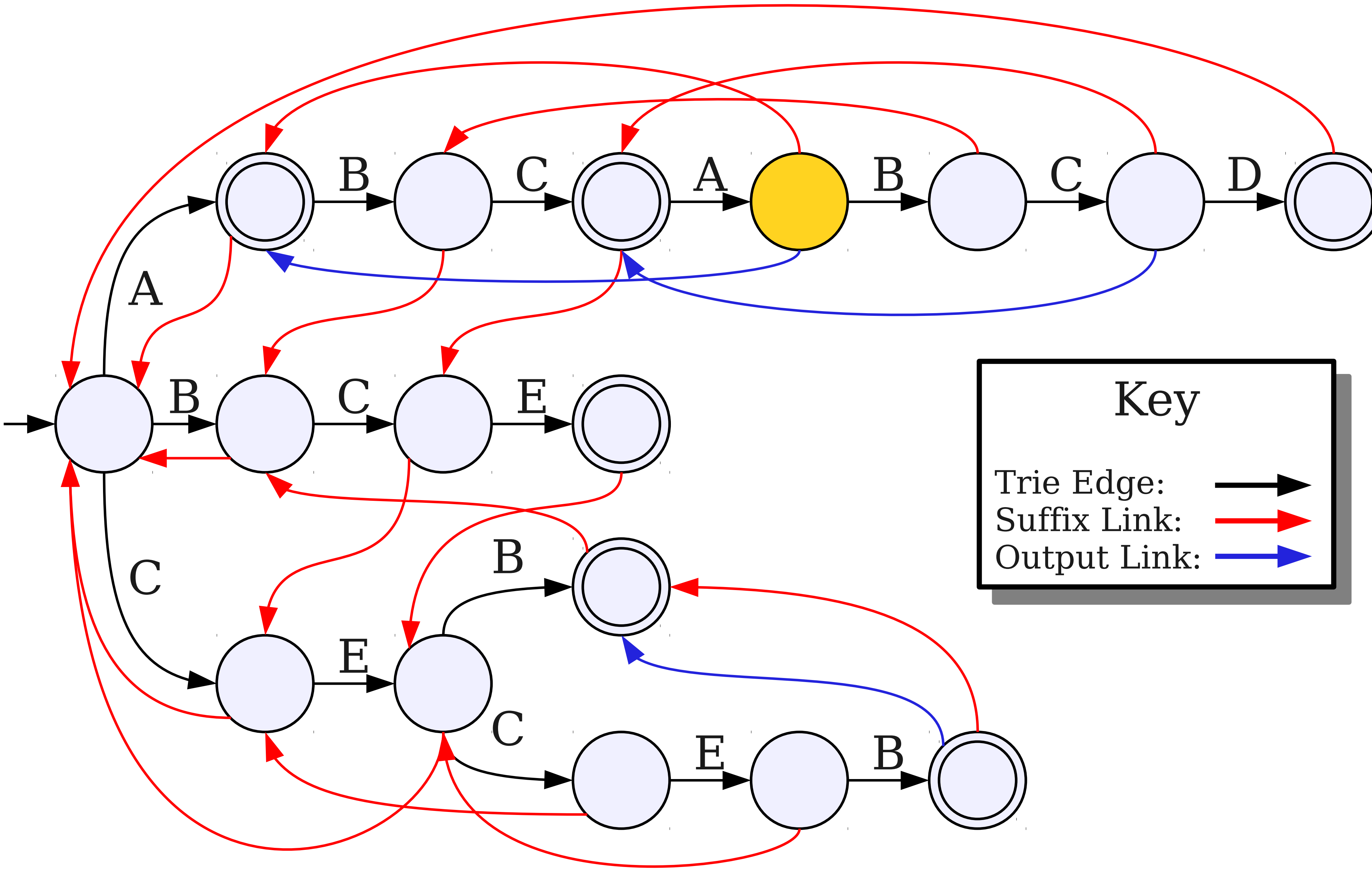


P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC P_6 = A



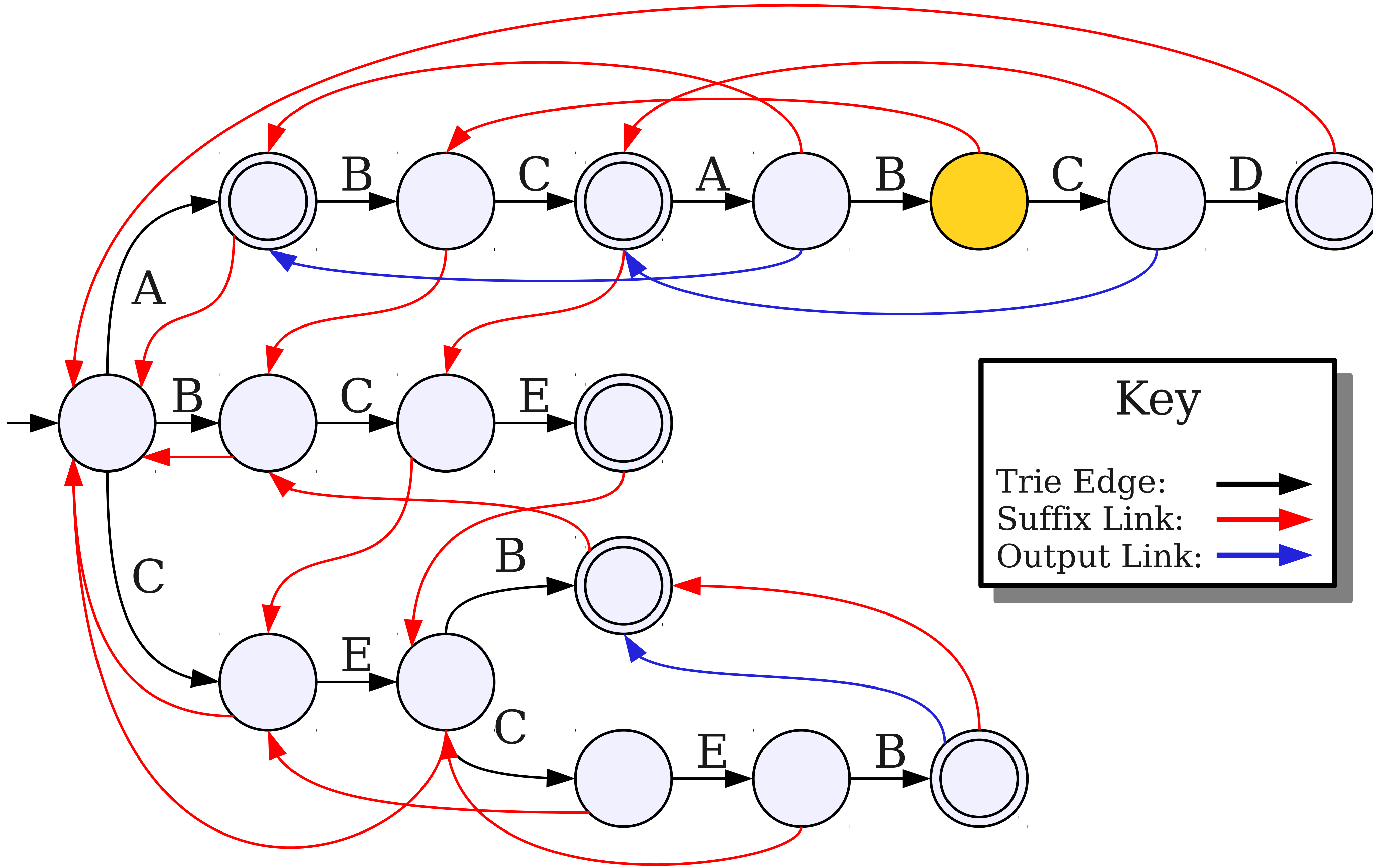
A B C A B C D

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC P_6 = A



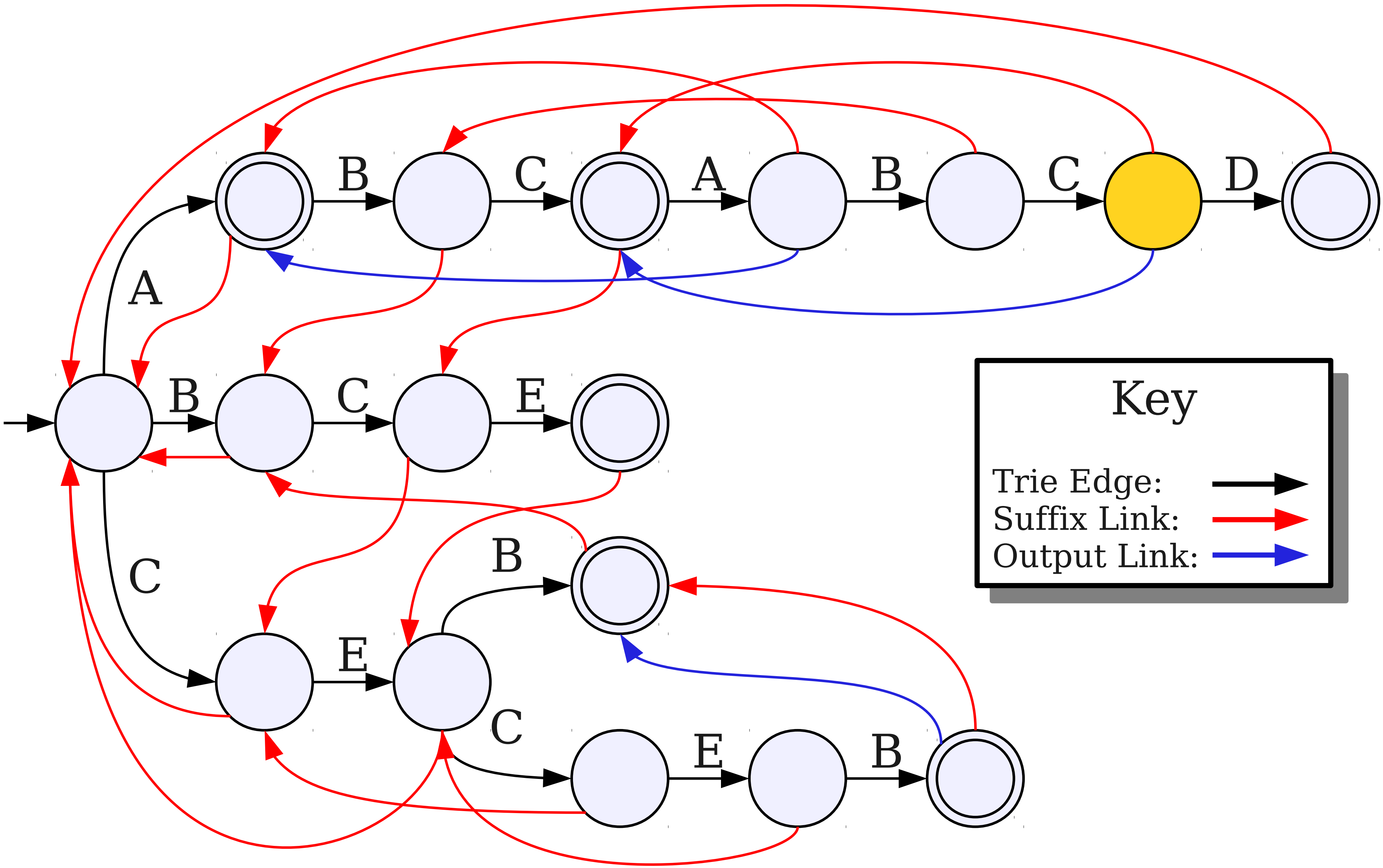
A B C A B C D

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC $P_6 = A$



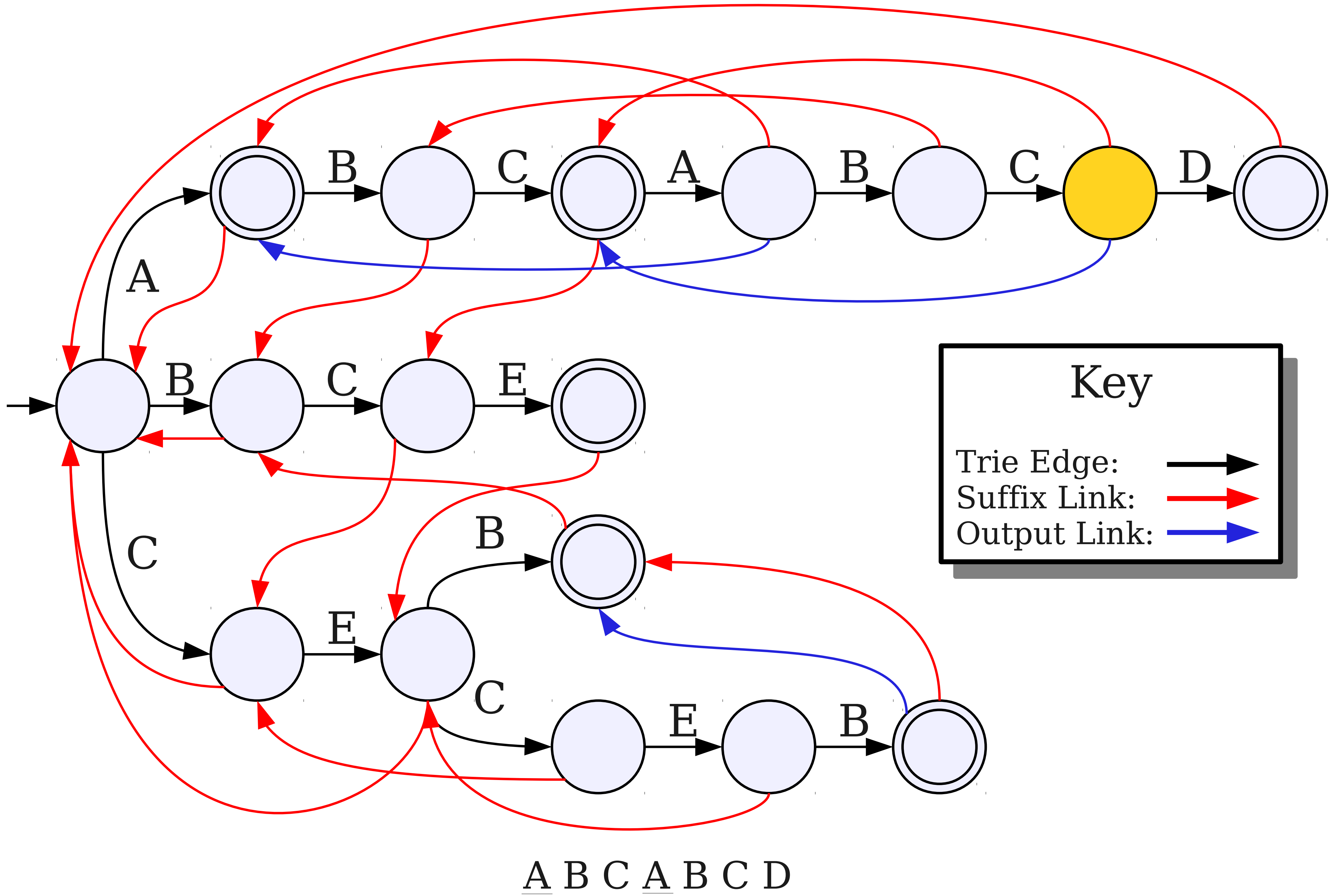
A B C A B C D

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC P_6 = *A*

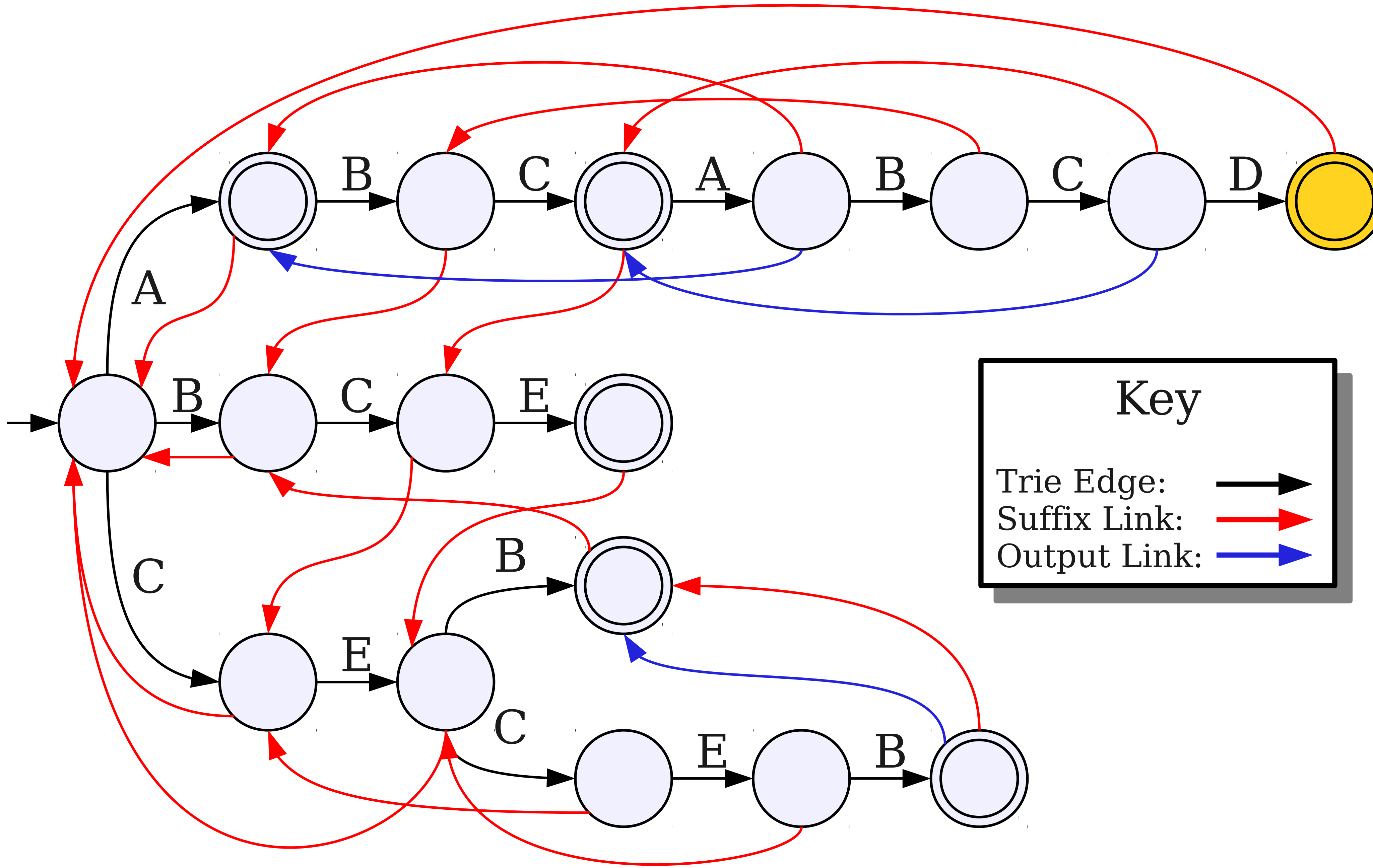


A B C A B C D

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC P_6 = A

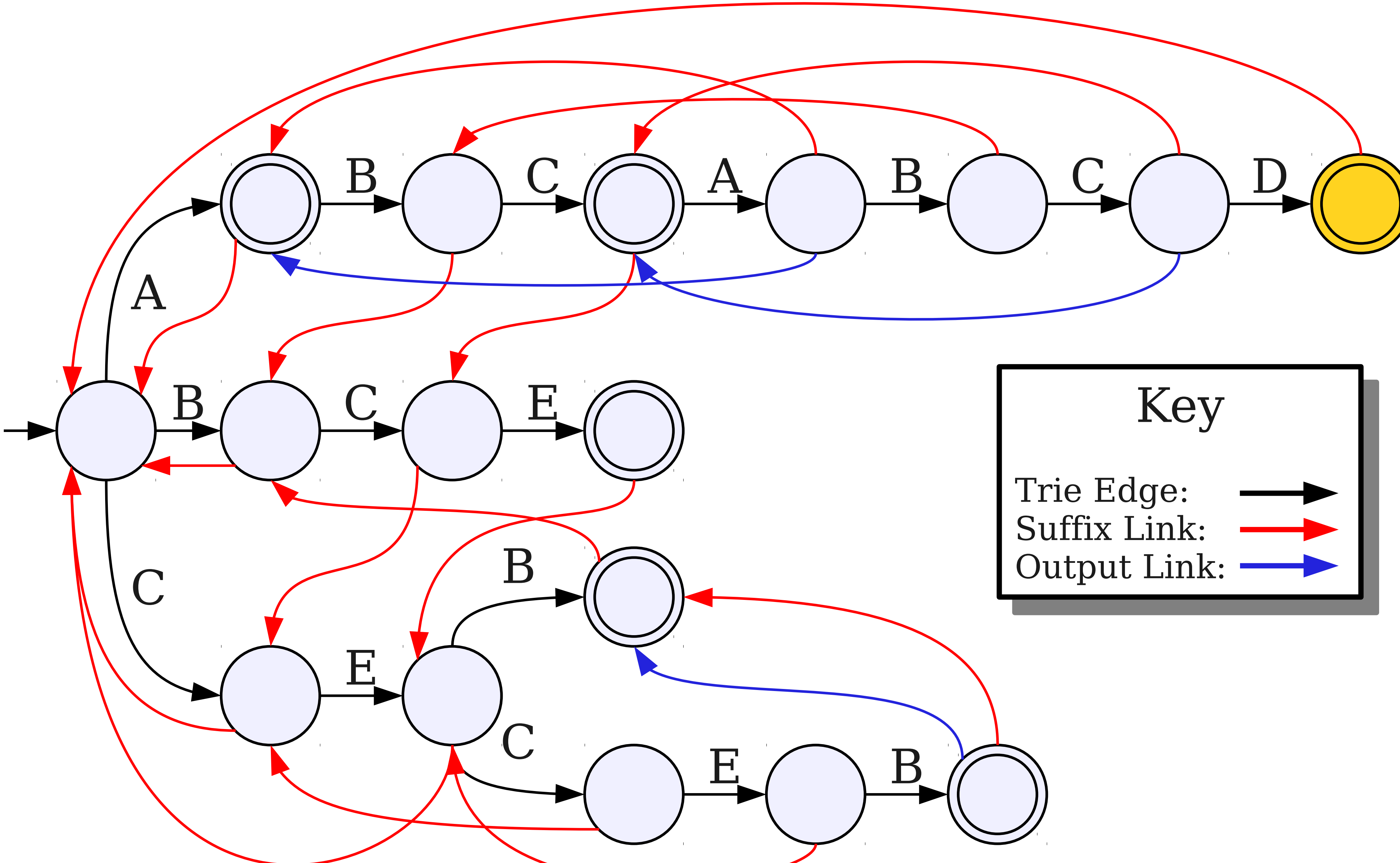


P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC P_6 = A



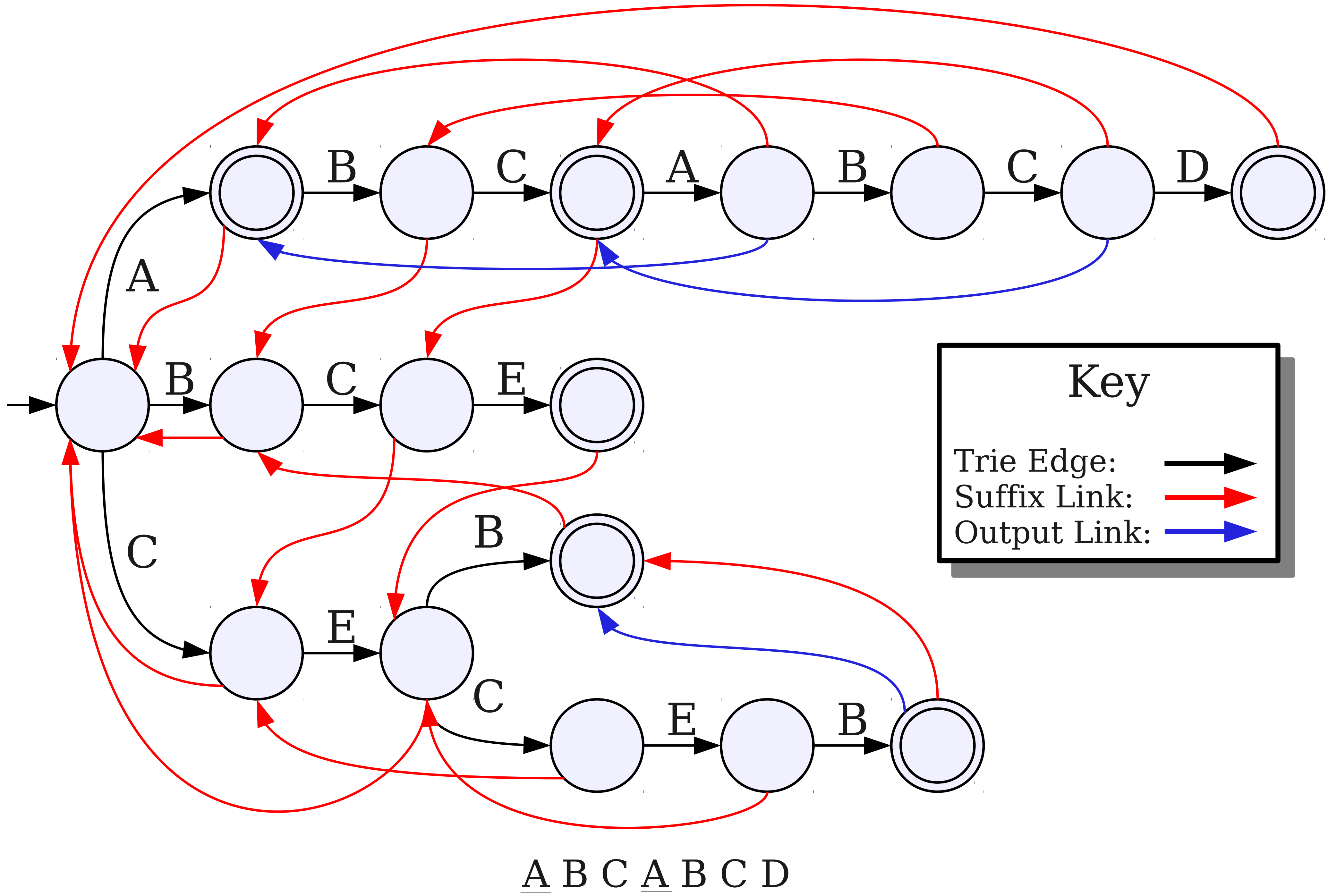
A B C A B C D

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC P_6 = *A*



A B C A B C D

P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC $P_6 = A$



P_1 = ABCABCD P_2 = BCE P_3 = CEB P_4 = CECEB P_5 = ABC P_6 = A

The Algorithm

- Let *state* be the start state.
- For $i = 0$ to $m - 1$
  - While *state* is not start and there is no trie edge labeled $T[i]$:
    - Follow the suffix link.
  - If there is a trie edge labeled $T[i]$, follow that edge.
  - If *state* is a word, output that word.
  - If *state* has an output link, repeatedly follow that link and output the words discovered.

The Runtime

- **Fact:** If $n = O(m)$, the number of occurrences of the substrings can be $\Theta(m^2)$.
- Consider patterns $a^1, a^2, \ldots, a^{\sqrt{m}}$ and search inside the string a^m.
- Total length of pattern strings: $O(m)$
- Total number of matches:

$$
\begin{aligned}
& m + (m - 1) + (m - 2) + \ldots + (m - \sqrt{m}) \\
= {} & m + (m - 1) + \ldots + 1 - (1 + 2 + 3 + \ldots + \sqrt{m}) \\
= {} & \Theta(m^2) - \Theta(m) \\
= {} & \boldsymbol{\Theta(m^2)}
\end{aligned}
$$

The Runtime

- The quadratic worst-case is not due to any inefficiencies; it's a fundamental limitation due to the number of matches that have to be generated.
- Let z be the total number of matches reported.
- Runtime of a search operation $\Theta(m + z)$.
- This is an **output-sensitive algorithm**; the runtime depends on how much data is generated.

Constructing Output Links

- Focus on a node w.
- **Claim:** Any pattern P_i that is a proper suffix of w is also a suffix of the string represented by w's suffix link.
- **Rationale:** w's suffix link points to the longest proper suffix of w in the trie.
- That suffix must be at least as long as P_i.

Constructing Output Links

- Initialize the root node's output link to be null.
- Run a breadth-first search over the trie.
- For each node w encountered, follow its suffix link to get to node x.
- If x is a pattern, set w's output link to be x.
- If x is not a pattern, set w's output link to be x's output link.
- Time required: **$O(n)$**.

The Complete Construction

- The algorithm we've explored is called the **Aho-Corasick string matching algorithm**.
- Given the patterns P_1, …, P_k, do the following:
  - Construct a trie holding the patterns in time $O(n)$.
  - Add suffix links to the trie in time $O(n)$.
  - Add output links to the trie in time $O(n)$.
  - Total time required: $O(n)$.
- To search a text T, run the previous algorithm to find all matches in time $\Theta(m + z)$.
- Total time required: $\mathbf{O(m + n + z)}$.

A Data-Structural View

- We've presented Aho-Corasick string matching as an algorithm, but you can really think of it as a data structure.
- Given a set of patterns, you only need to do the O(n) preprocessing once.
- From there, you can match in time O(m + z) on any input string you'd like.
- In fact, this is frequently done in practice!

Summary

- Tries are a simple and flexible data structure for storing strings.
- Suffix links point from trie nodes to the nodes corresponding to their longest proper suffixes. (*suffices?*) They can be filled in in time linear in the length of the strings.
- A string x is a substring of a string w precisely when x is a suffix of a prefix of w.
- Aho-Corasick string matching requires $O(n)$ preprocessing and can do matching in time $O(m + z)$.

Next Time

- **Suffix Trees**
  - A powerful, flexible data structure for solving just about every string problem ever.
- **Suffix Arrays**
  - A simpler and more compact representation of suffix trees.