Beyond BSTs
(we can do better?!)

Ilan Goodman

Thanks to Keith Schwarz for teaching me multiple classes on this material and for letting me borrow gratuitously from your lecture examples—you rock! ☺

What We've Seen So Far

- Balanced BSTs are useful in implementing `Map`s and `Set`s and can be used for sorting
- In fact, the $\Theta(n \log n)$ time it takes to sort using a balanced BST is **optimal** for any comparison-based sorting algorithm
  - Take CS 161 for more details ☺
- What does it mean to do "better"?

“Better” than Balanced BSTs

- Additional functionality
  - `containsPrefix` in `Lexicon`
- Additional input guarantees
  - Only storing integers in a known range, e.g.
- Non-uniform access patterns
  - Weight-balanced BSTs and splay trees
- Not enough time to talk about these last two today, but take CS 166 for more details

Outline for Today

- Strings and tries
  - How do we implement our last ADT, the `Lexicon`?
  - What efficiency guarantees can we expect?
  - How does this relate to everything we've learned the rest of the quarter?

Ordered Dictionaries

- Key operations:
  - `insert/delete/lookup`
  - `containsPrefix` (for strings)
- Balanced BST does each of these in *O*(log *n*) time … assuming comparisons take constant time

String Comparisons

- How long, in the worst case, does it take to compare strings of lengths L_1 and L_2?
  - A) $O(1)$
  - B) $O(\min\{L_1, L_2\})$
  - C) $O(L_1 + L_2)$
  - D) $O(1 + (L_2 - L_1))$
  - E) Other/none of the above/multiple of the above/unknowable

String Comparisons

- How long, in the worst case, does it take to compare strings of lengths L_1 and L_2?
  - A) $O(1)$
  - **B) $O(\min\{L_1, L_2\})$**
  - C) $O(L_1 + L_2)$
  - D) $O(1 + (L_2 - L_1))$
  - E) Other/none of the above/multiple of the above/unknowable

Implementing `Lexicon` with BSTs

- If the longest string in our `Lexicon` has length *L*, the best bound we can get for our key operations (`insert`/`lookup`/`delete`/`containsPrefix`) is $O(L \log n)$
- We often use `Lexicon`s to represent English, for example, so *n* is large (*~1,000,000*) and *L* is also non-negligible
- Can we do better?

Implementing `Lexicon` with Hash Tables

- If we back our `Lexicon` with a hash table, we can knock the `insert`/`lookup`/`delete` operations to $O(L)$ time (expected, amortized)
- It now takes $O(Ln)$ time to check if a prefix exists, which is unacceptable in many applications
- Can we do better?

Rethinking Hashing

- Hashing does well except on `containsPrefix`, so could we just change it a little to improve our time bounds?
- In a hash table, we use the hash of the string to figure which bucket it goes in
- What if we had a really bad hash function, like the first letter of the string?

- 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

Source: http://images.mentalfloss.com/sites/default/files/styles/insert_main_wide_image/public/istock_000019851871_small.jpg and Keith Schwarz

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
“”	AR	AN	IKDIK
B	ARD	ANE	IKTAT
BOUT	ARN	AT	
D	E		
DAGE	ED		
DAGIO	ET		
	ETA		

A	B	C	D
B	AR	AN	IKDIK
BOUT	ARD	ANE	IKTAT
D	ARN	AT	
DAGE	E		
DAGIO	ED		
	ET		
	ETA		

Recursive Data Structures?

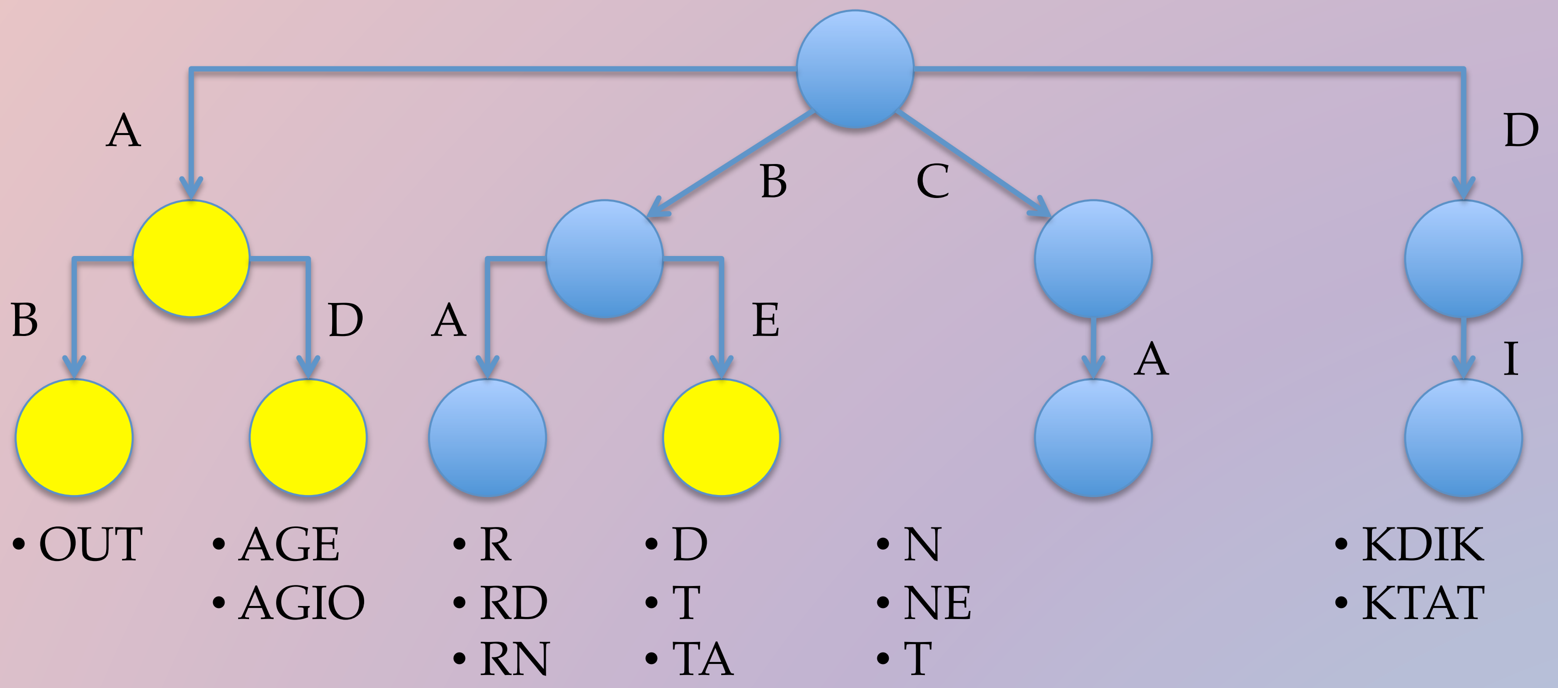
- That seems to look a bit nicer, but why stop here?
- Each bucket has a list of strings in it that could be sorted into smaller buckets based on their first letters
- We can do this until each bucket represents only one letter

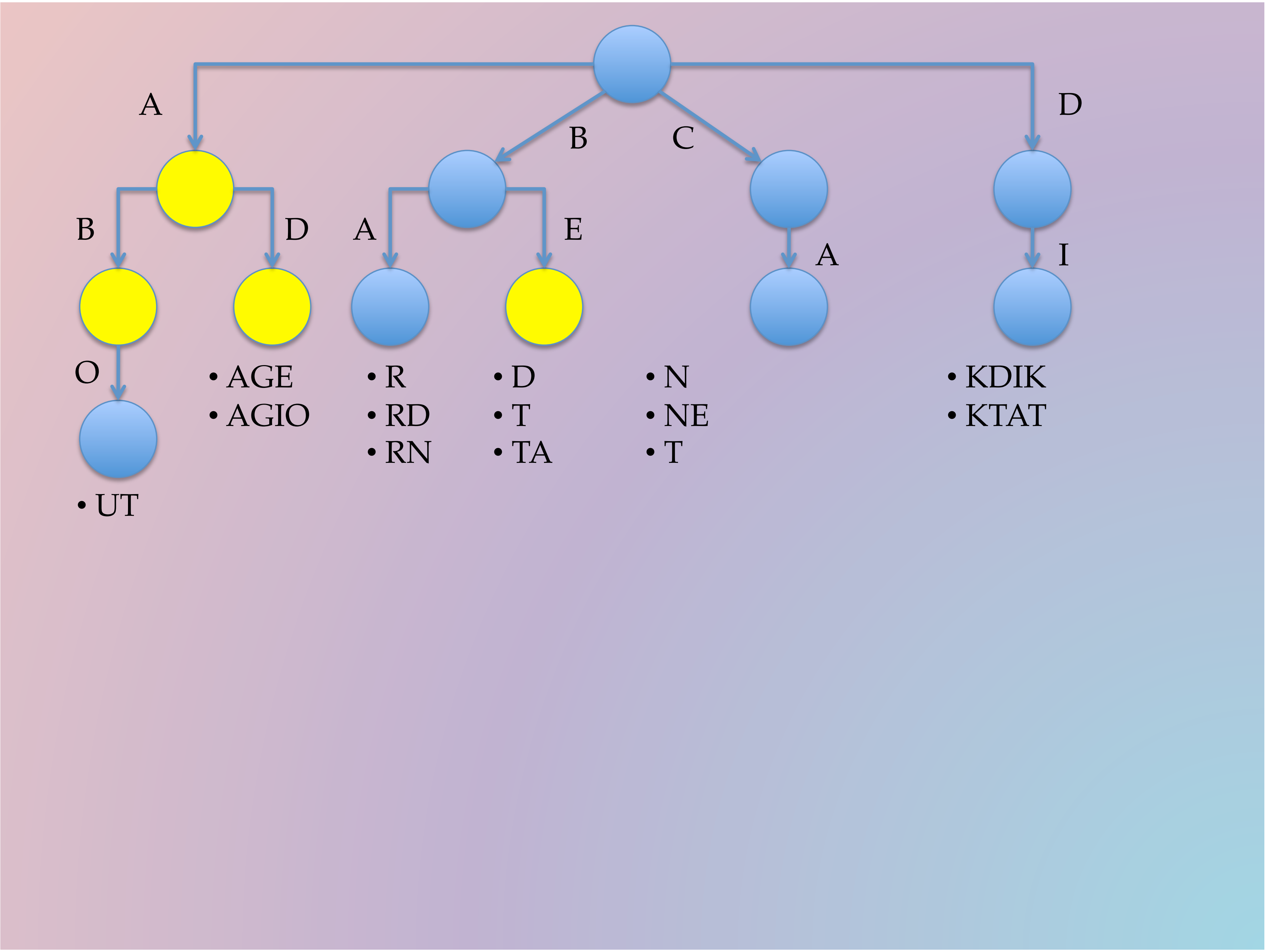
A	B	C	D
• B • BOUT • D • DAGE • DAGIO	• AR • ARD • ARN • E • ED • ET • ETA	• AN • ANE • AT	• IKDIK • IKTAT

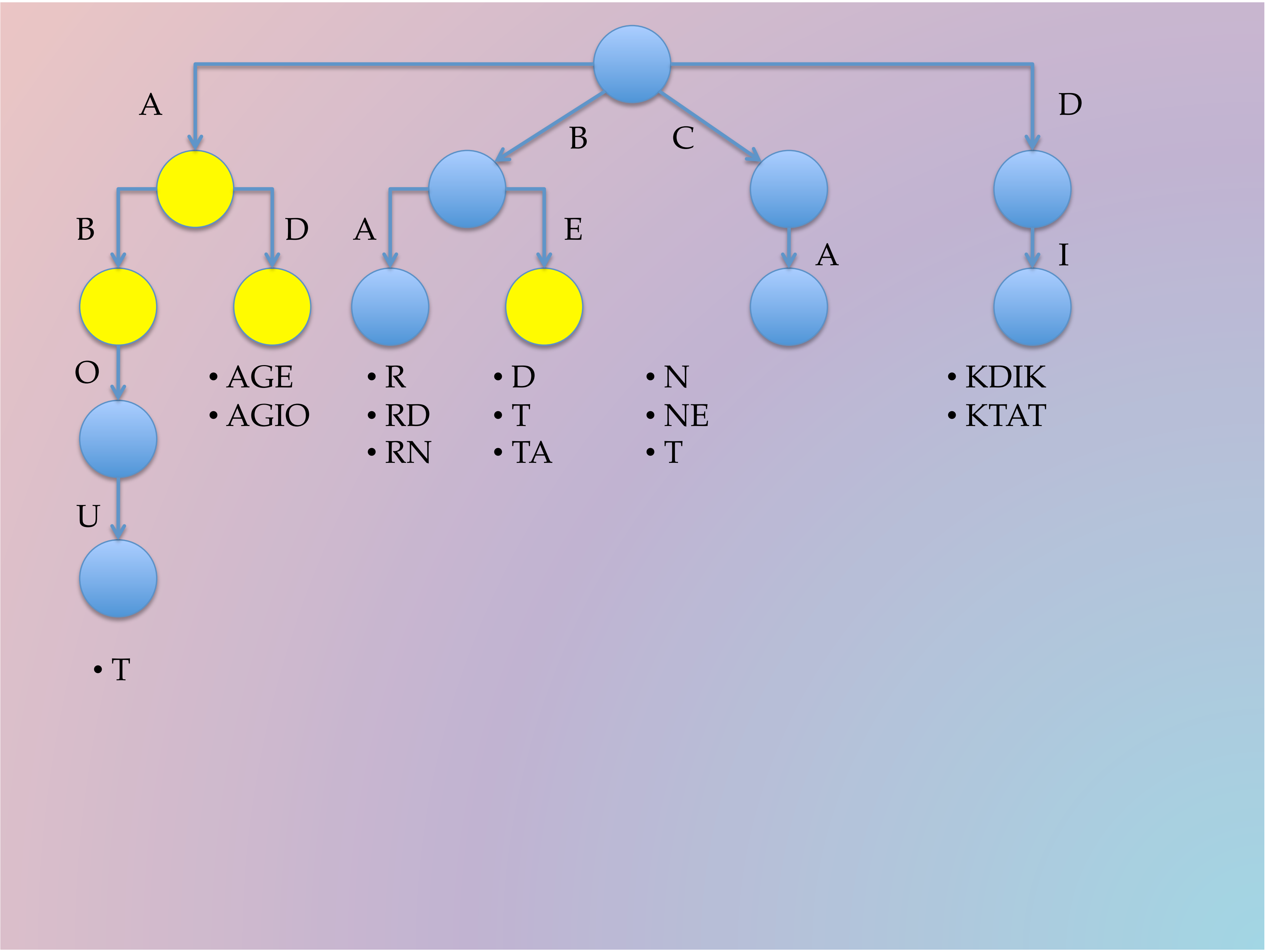
A		B		C	D
B	D	A	E	A	I
• B	• D	• AR	• E	• AN	• IKDIK
• BOUT	• DAGE	• ARD	• ED	• ANE	• IKTAT
	• DAGIO	• ARN	• ET	• AT	
			• ETA		

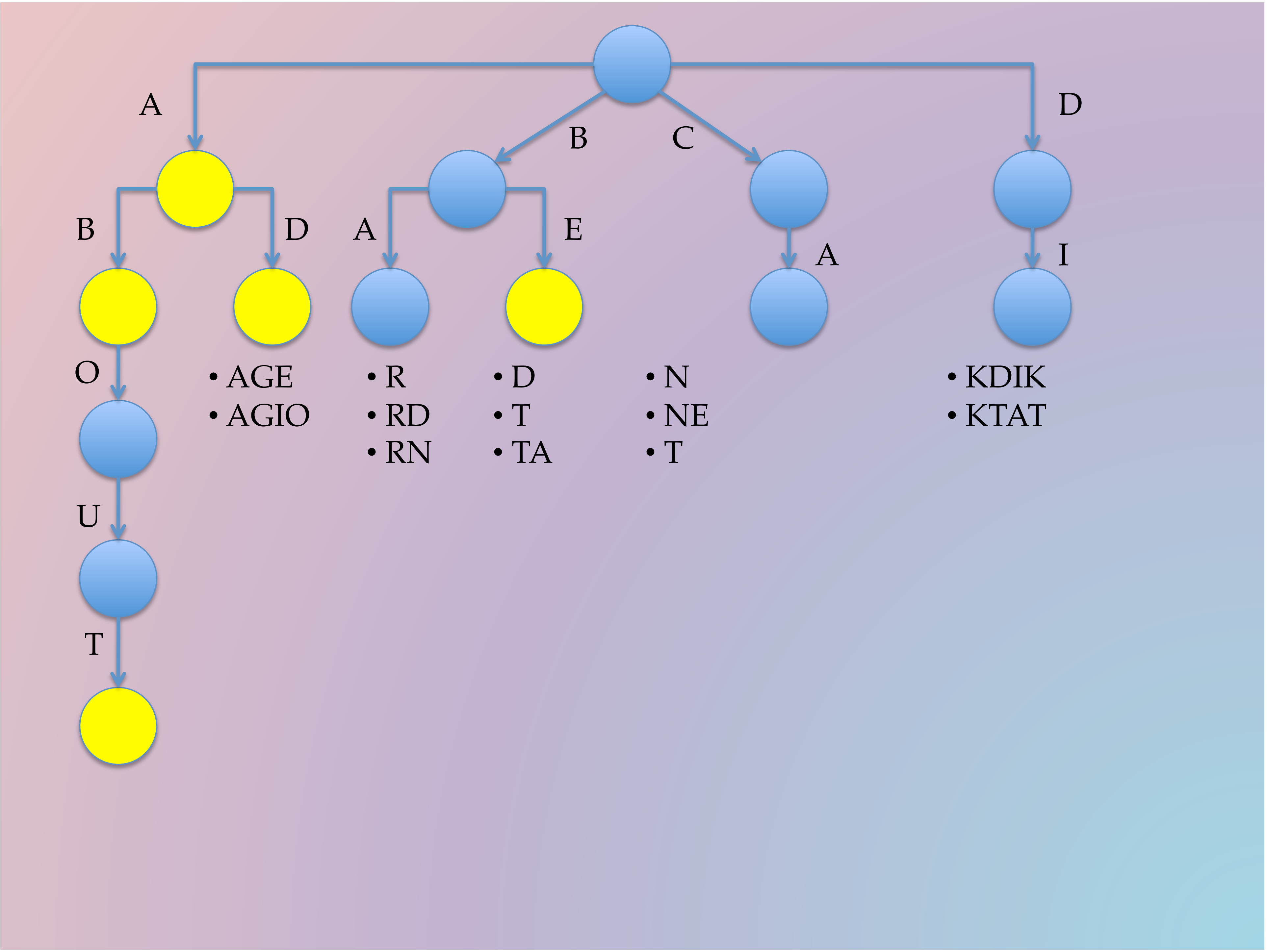
A		B		C	D
B	D	A	E	A	I
• “”	• “”	• R	• “”	• N	• KDIK
• OUT	• AGE	• RD	• D	• NE	• KTAT
	• AGIO	• RN	• T	• T	
			• TA		

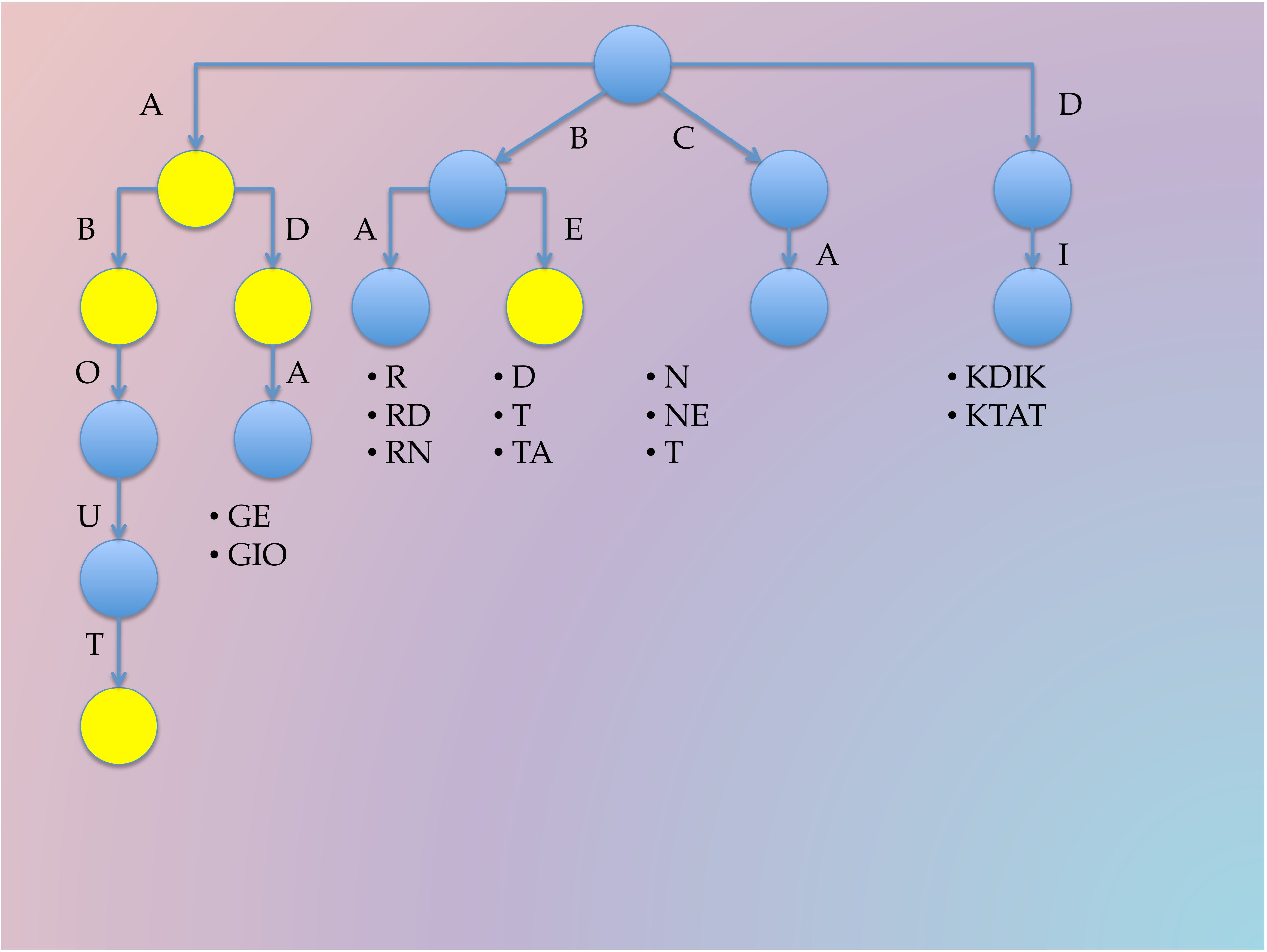
A		B		C	D
B	D	A	E	A	I
• OUT	• AGE • AGIO	• R • RD • RN	• D • T • TA	• N • NE • T	• KDIK • KTAT


A
B
C
D
B
D
A
E
A
I
• OUT
• AGE
• AGIO
• R
• RD
• RN
• D
• T
• TA
• N
• NE
• T
• KDIK
• KTAT

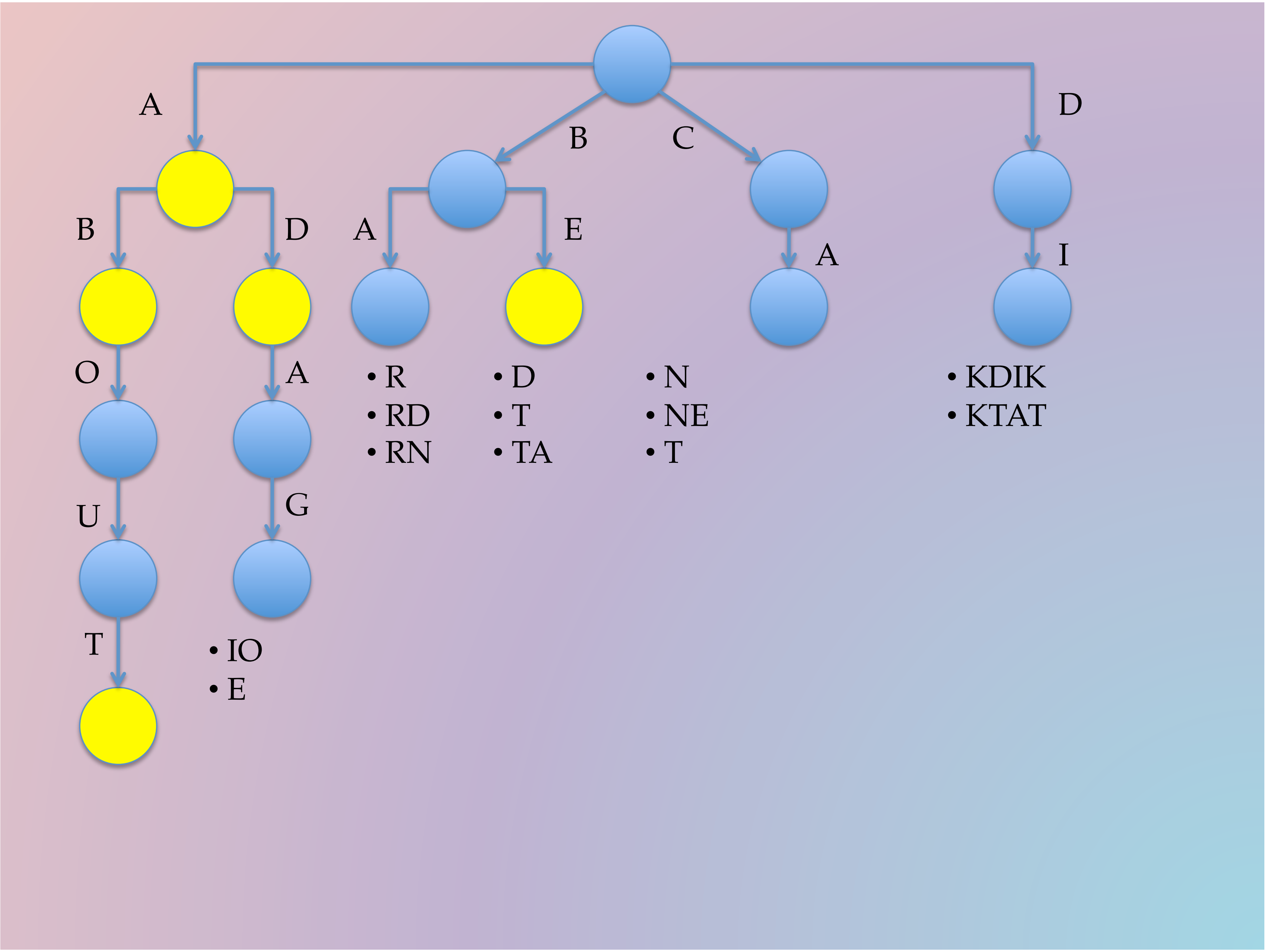

A
B
C
D
B
D
A
E
A
I
O
• AGE
• AGIO
• R
• RD
• RN
• D
• T
• TA
• N
• NE
• T
• KDIK
• KTAT
• UT

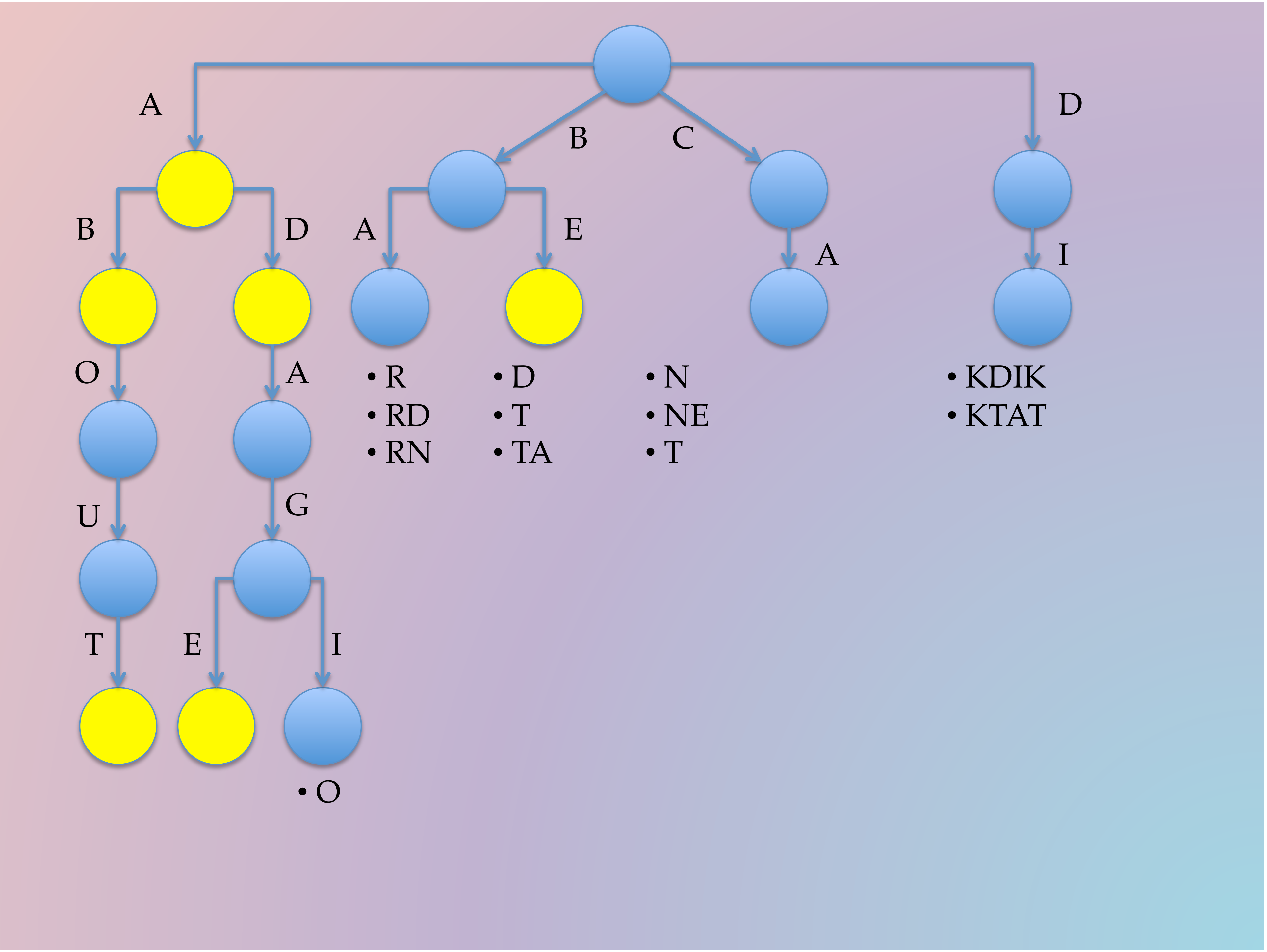

A
B
C
D
B
D
A
E
A
I
O
• AGE
• AGIO
• R
• RD
• RN
• D
• T
• TA
• N
• NE
• T
• KDIK
• KTAT
U
• T

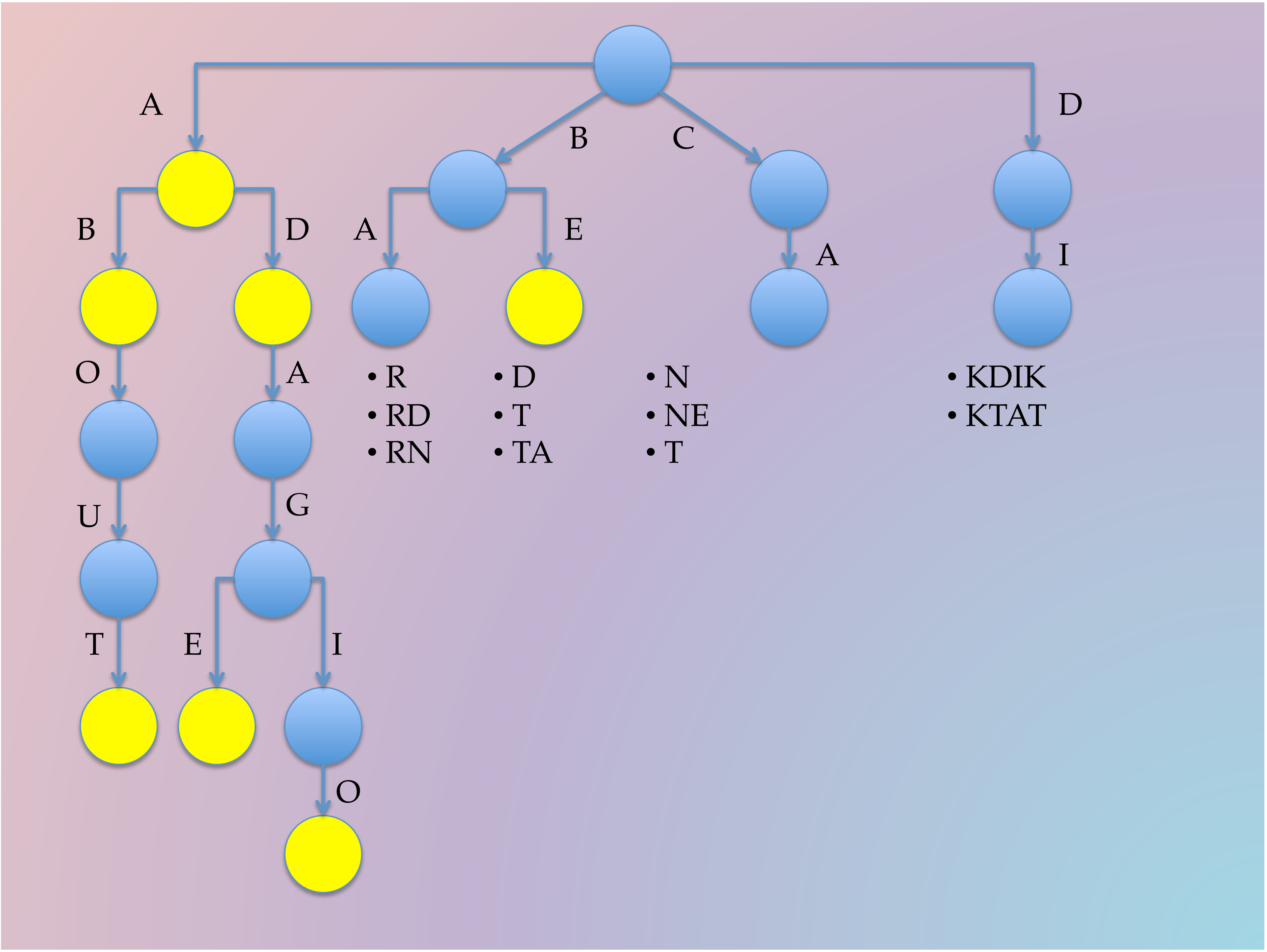

A
B
C
D
B
D
A
E
A
I
O
U
T
• AGE
• AGIO
• R
• RD
• RN
• D
• T
• TA
• N
• NE
• T
• KDIK
• KTAT

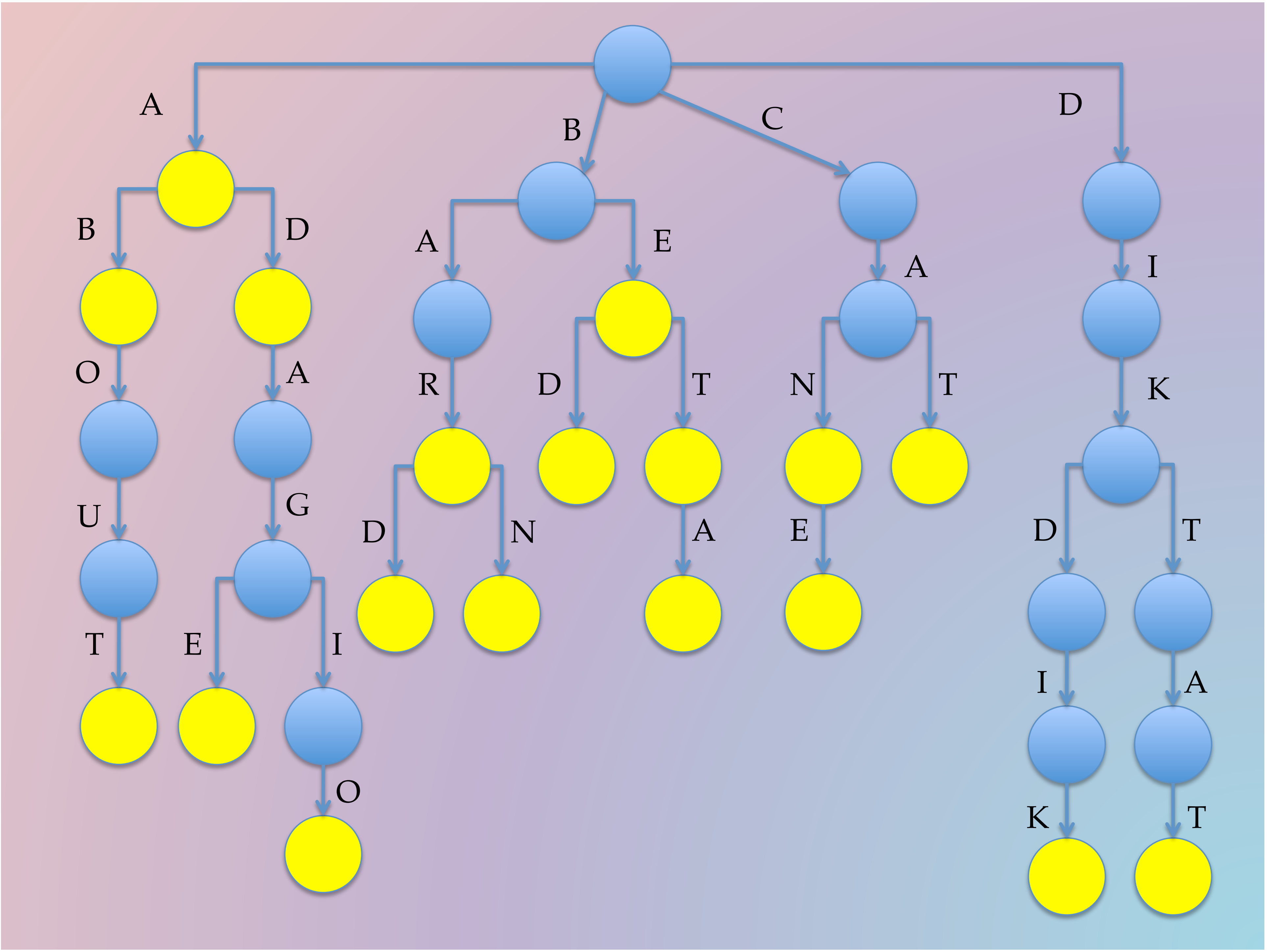


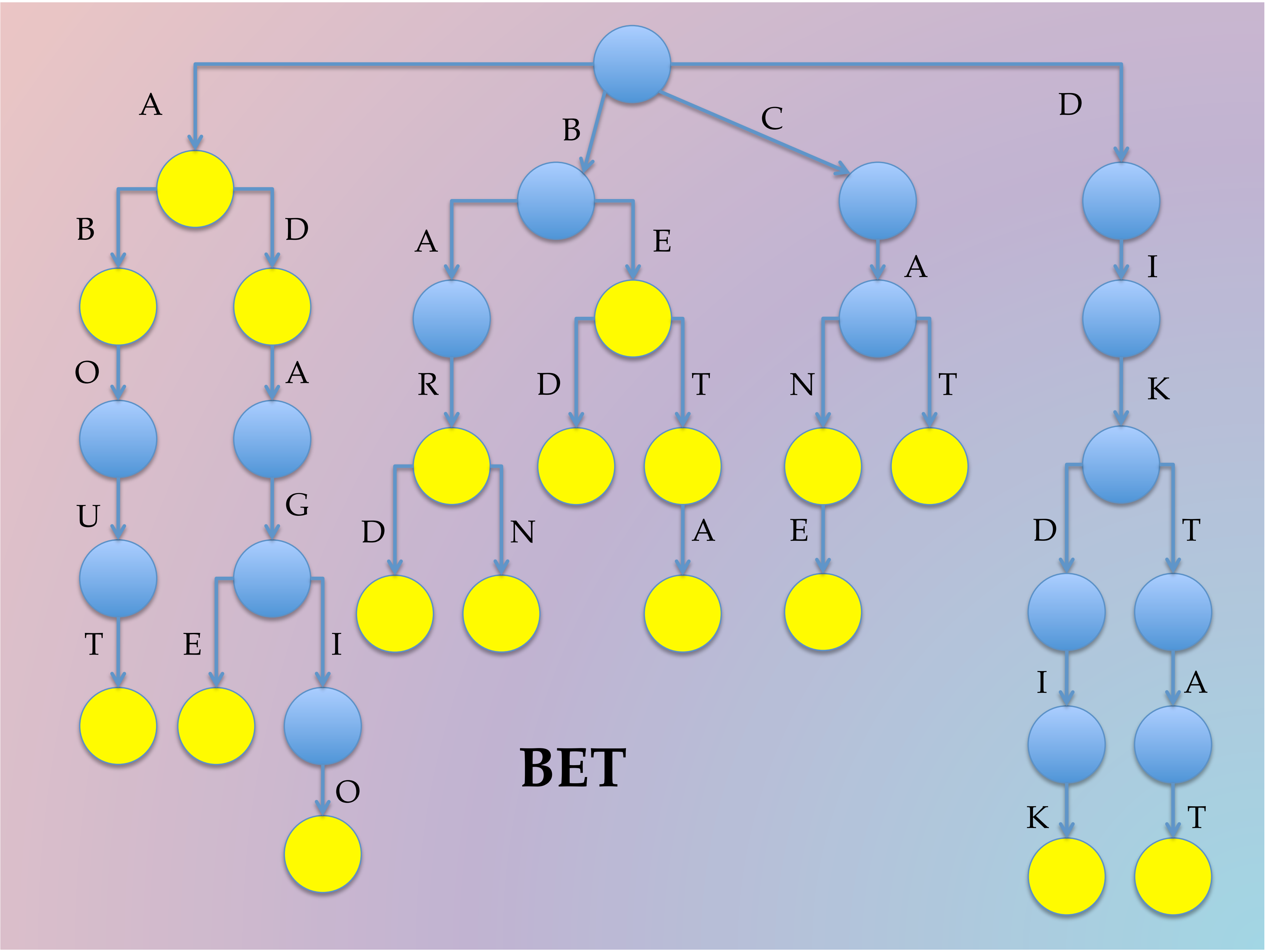
A
B
C
D
B
D
A
E
A
I
O
A
• R
• RD
• RN
• D
• T
• TA
• N
• NE
• T
• KDIK
• KTAT
U
• GE
• GIO
T

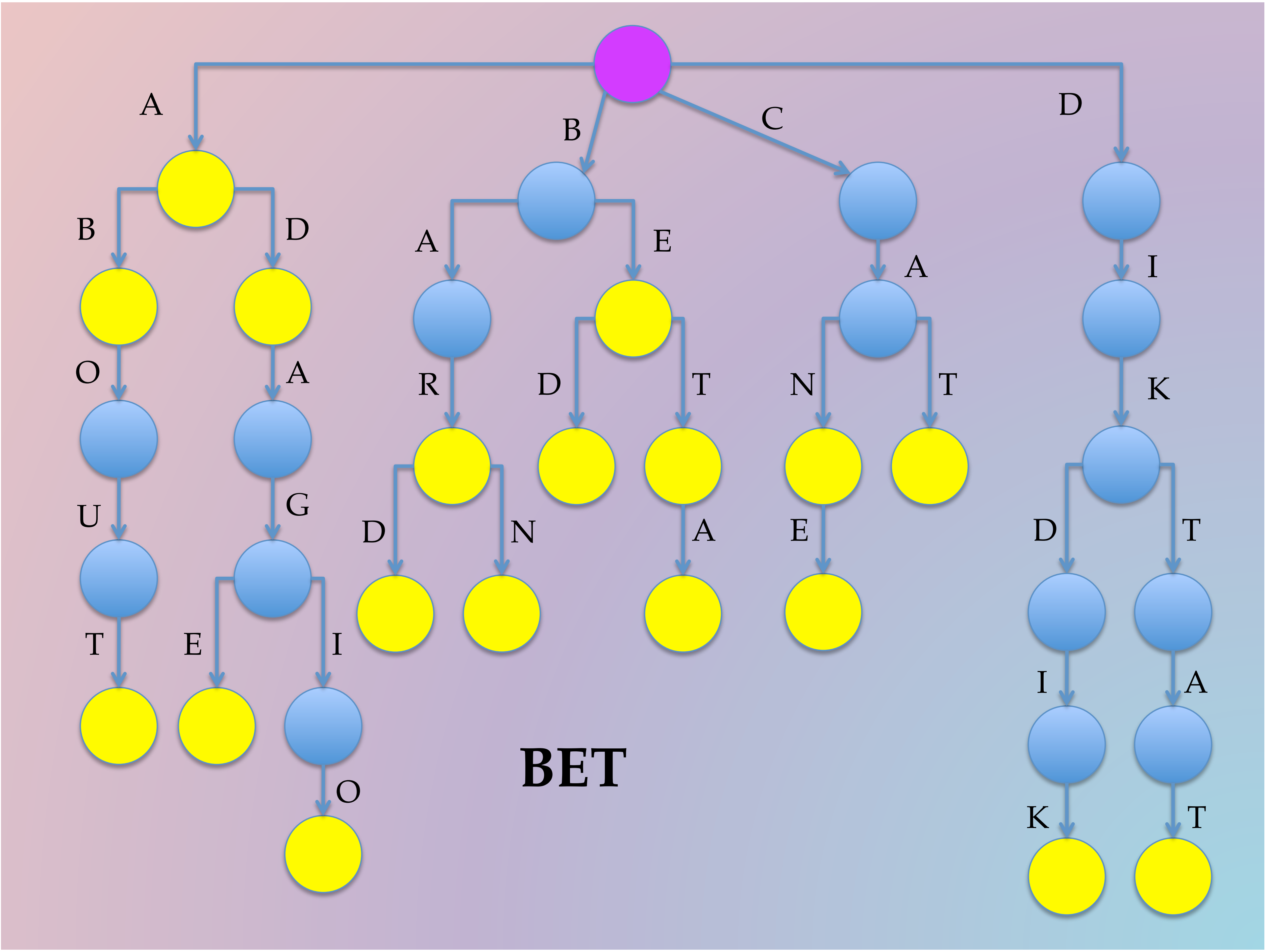

A
B
C
D
B
D
A
E
A
I
O
A
• R
• RD
• RN
• D
• T
• TA
• N
• NE
• T
• KDIK
• KTAT
G
U
T
• IO
• E

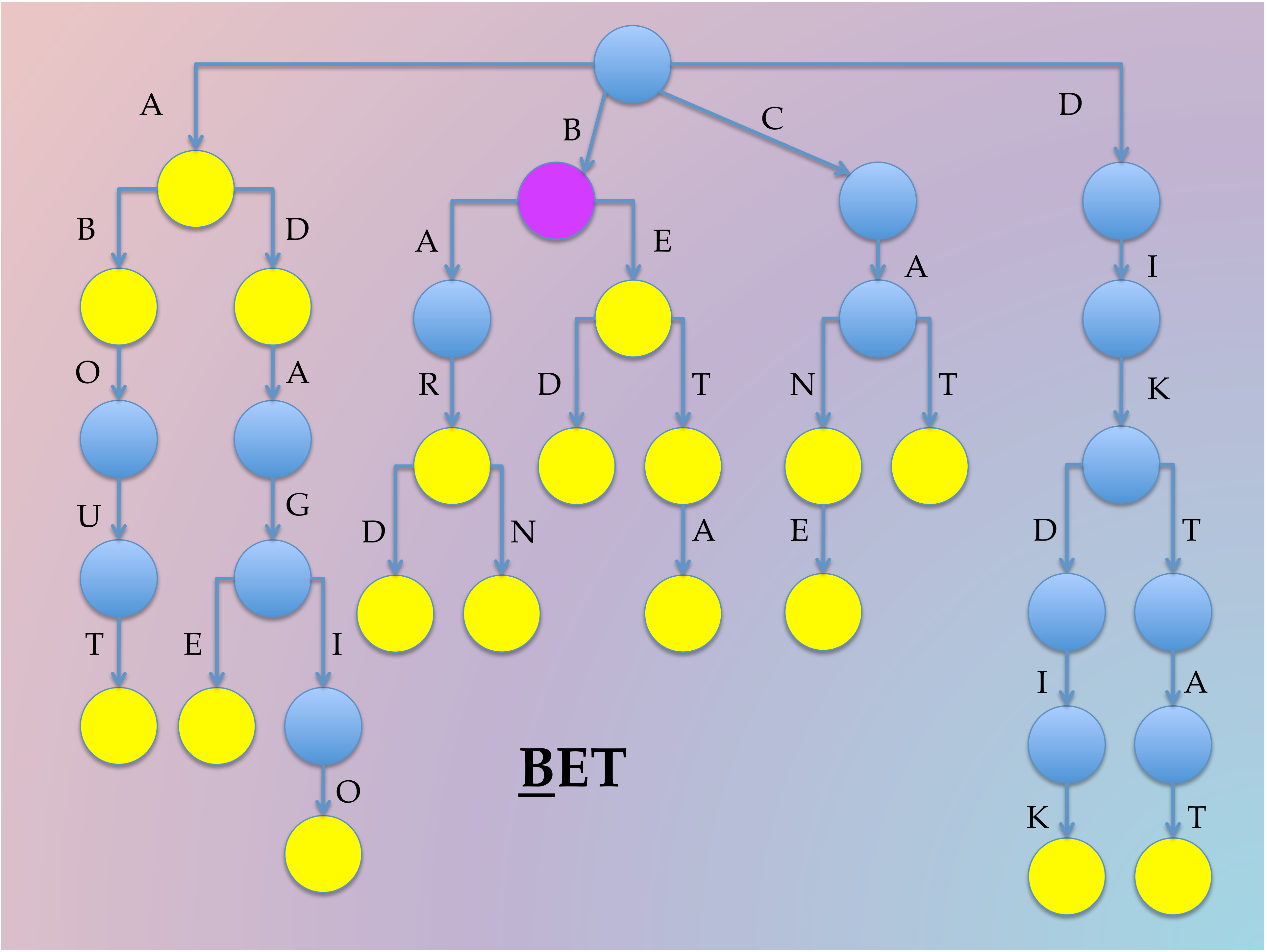

A
B
C
D
B
D
A
E
A
I
O
A
• R
• RD
• RN
• D
• T
• TA
• N
• NE
• T
• KDIK
• KTAT
U
G
T
E
I
• O

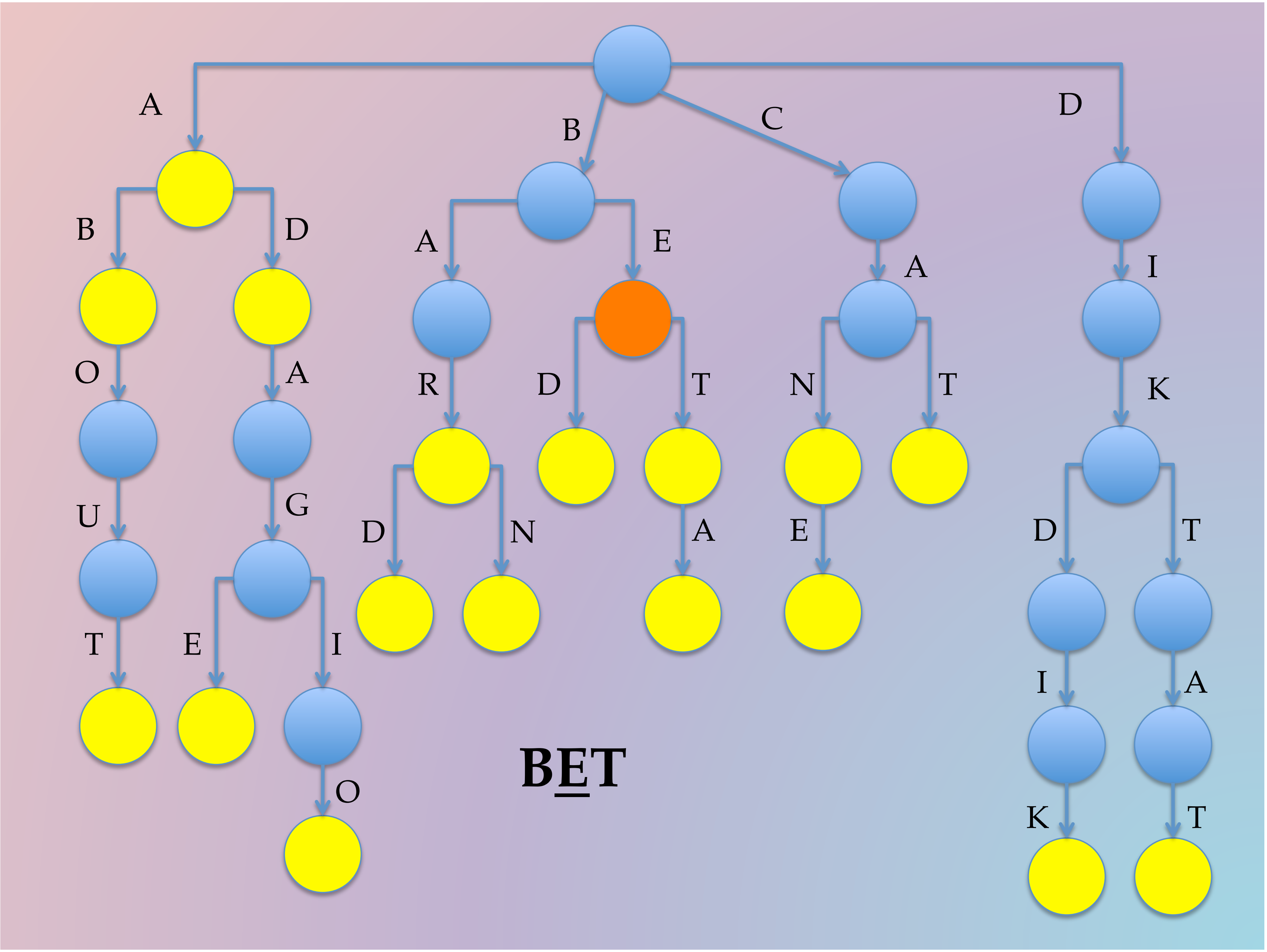

A
B
C
D
B
D
A
E
A
I
O
A
• R
• RD
• RN
• D
• T
• TA
• N
• NE
• T
• KDIK
• KTAT
U
G
T
E
I
O

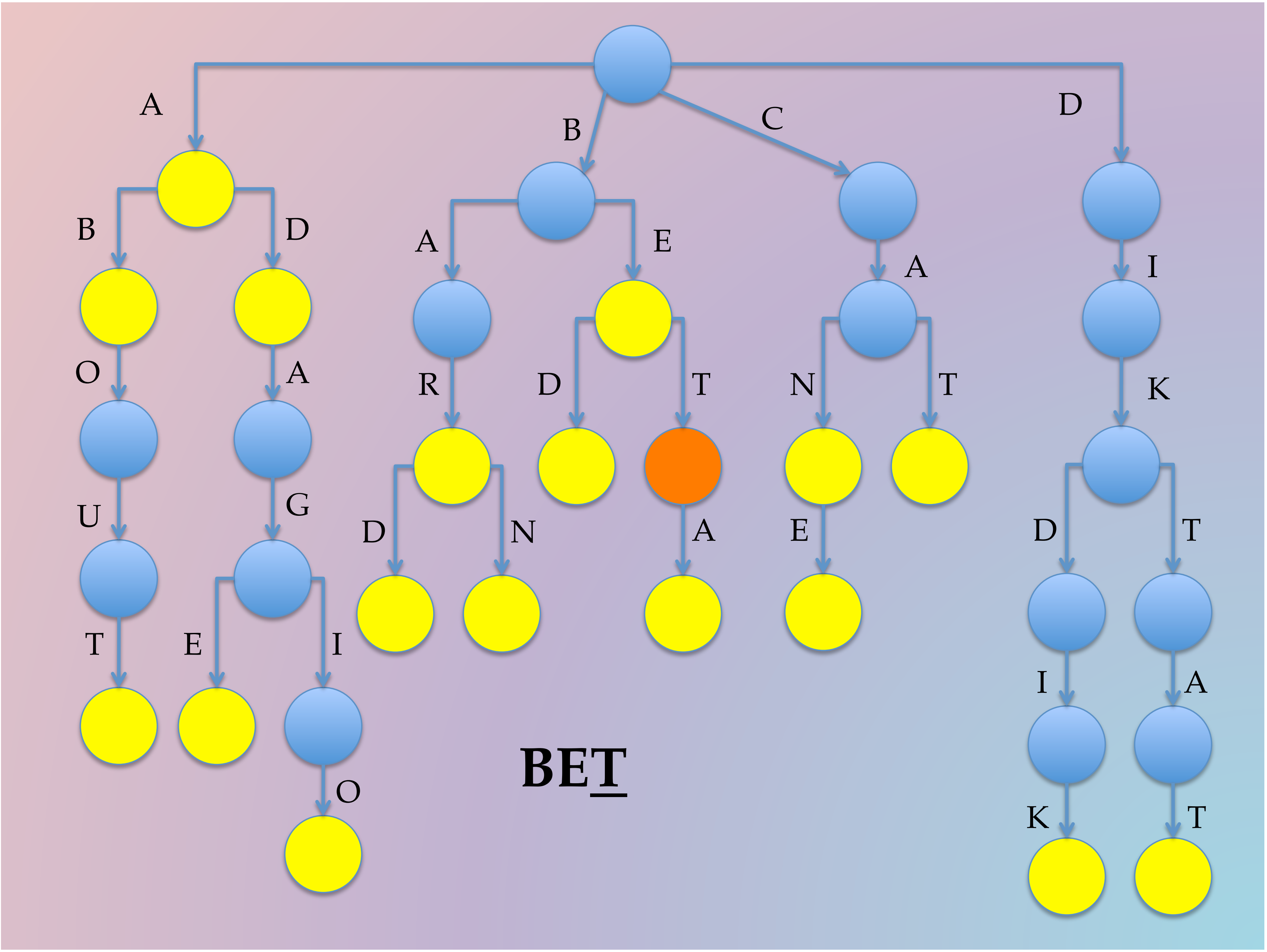

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T

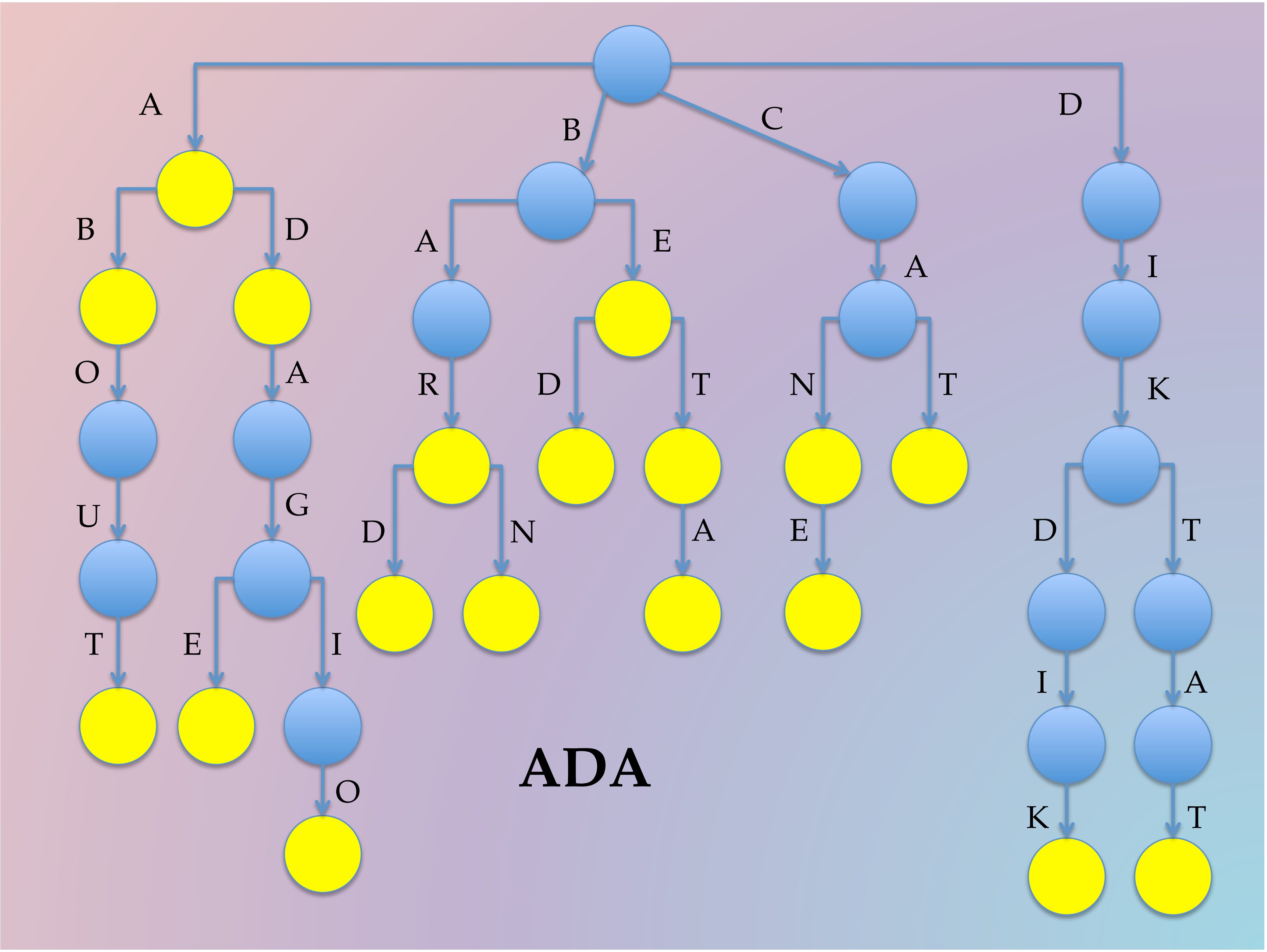

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
BET

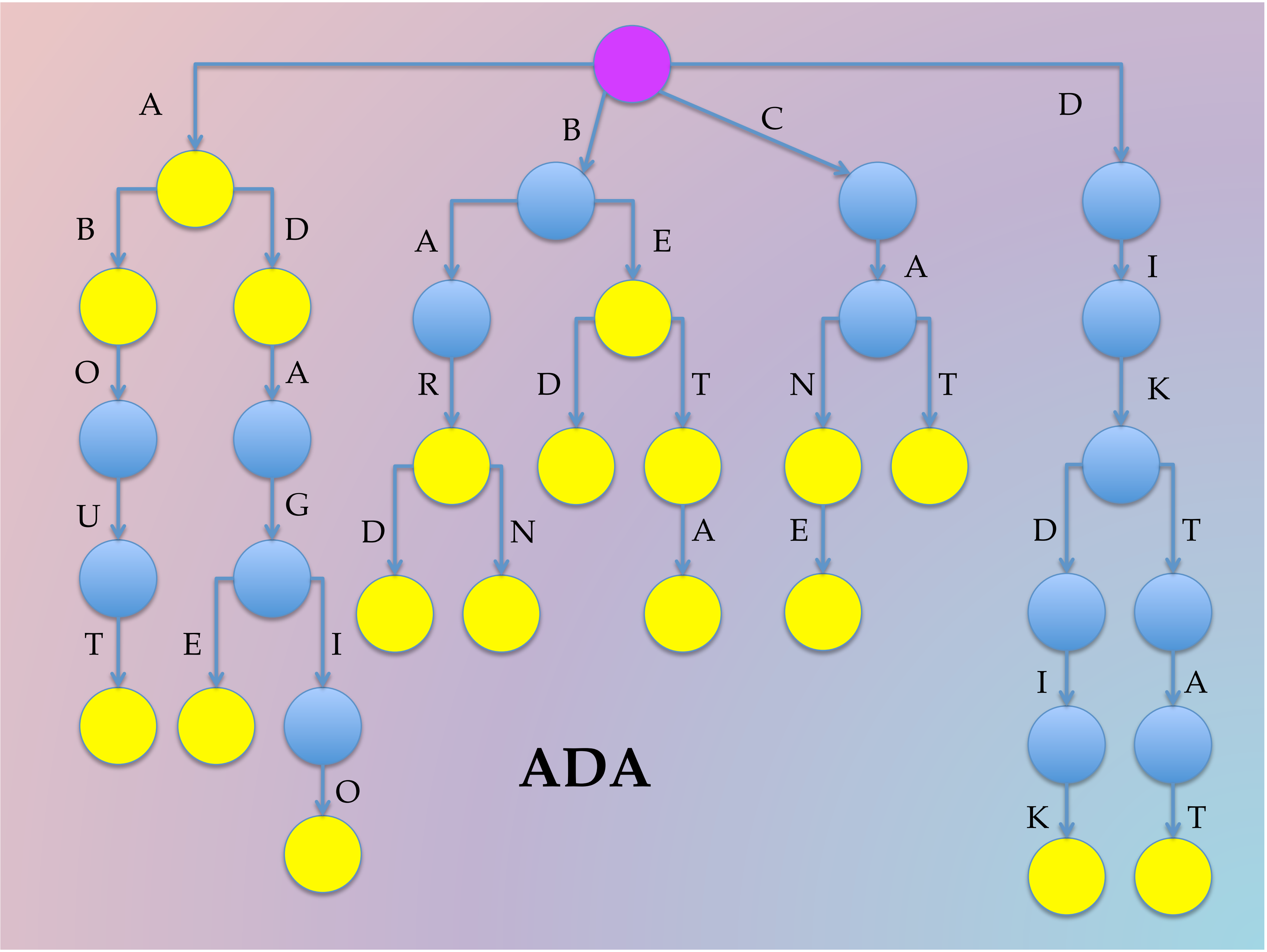

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
BET

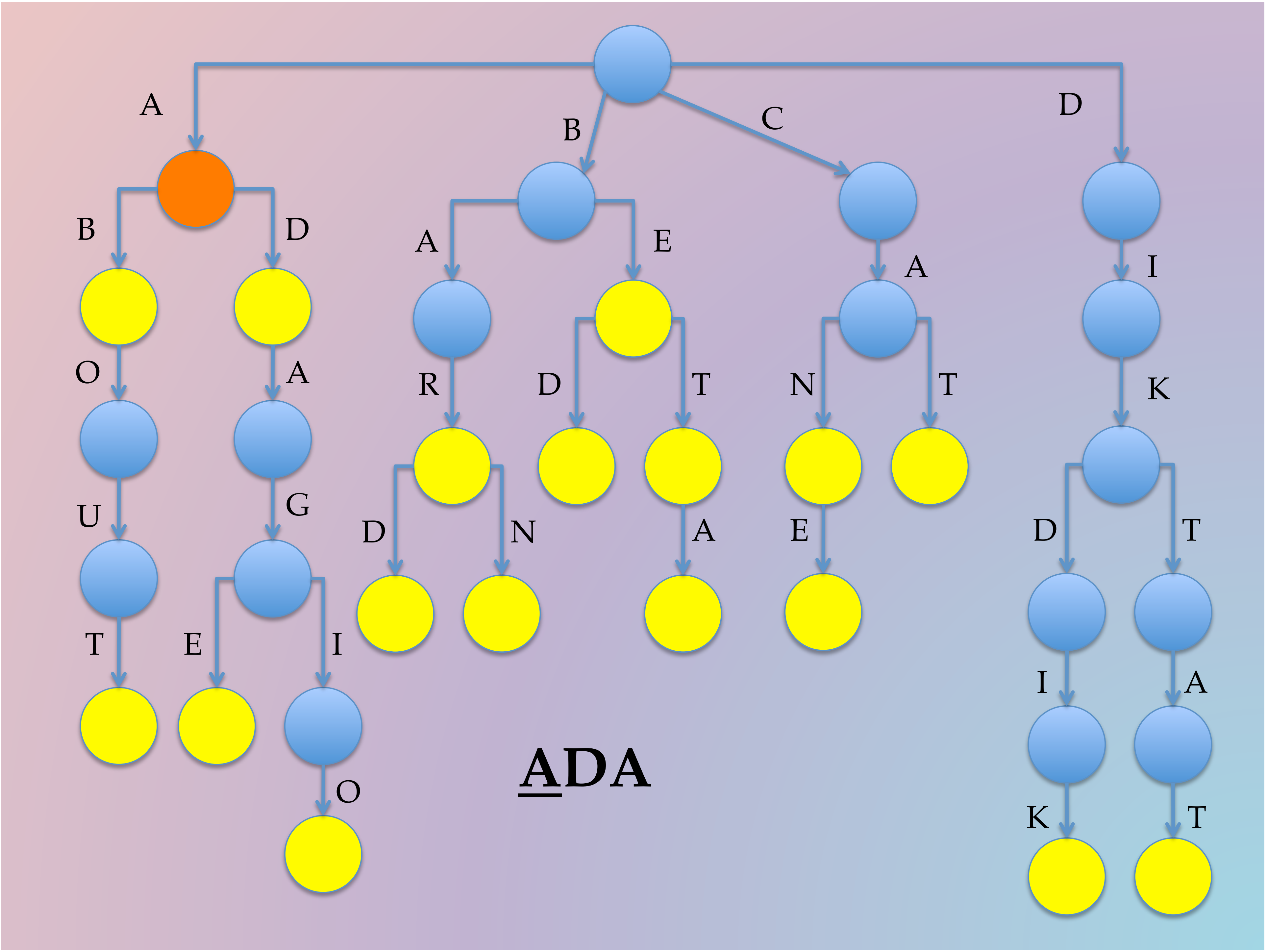

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
BET


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
BET

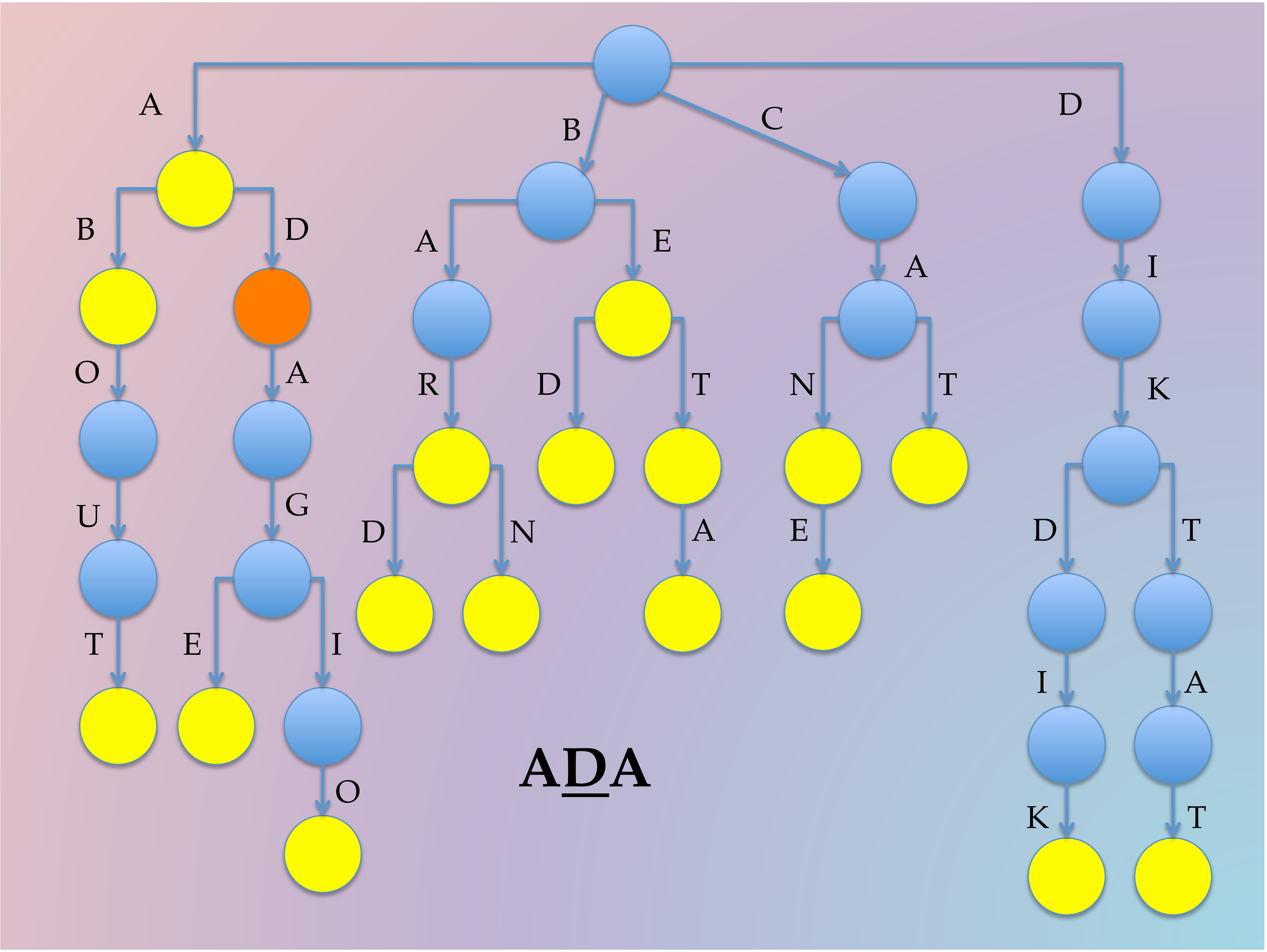

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
BET

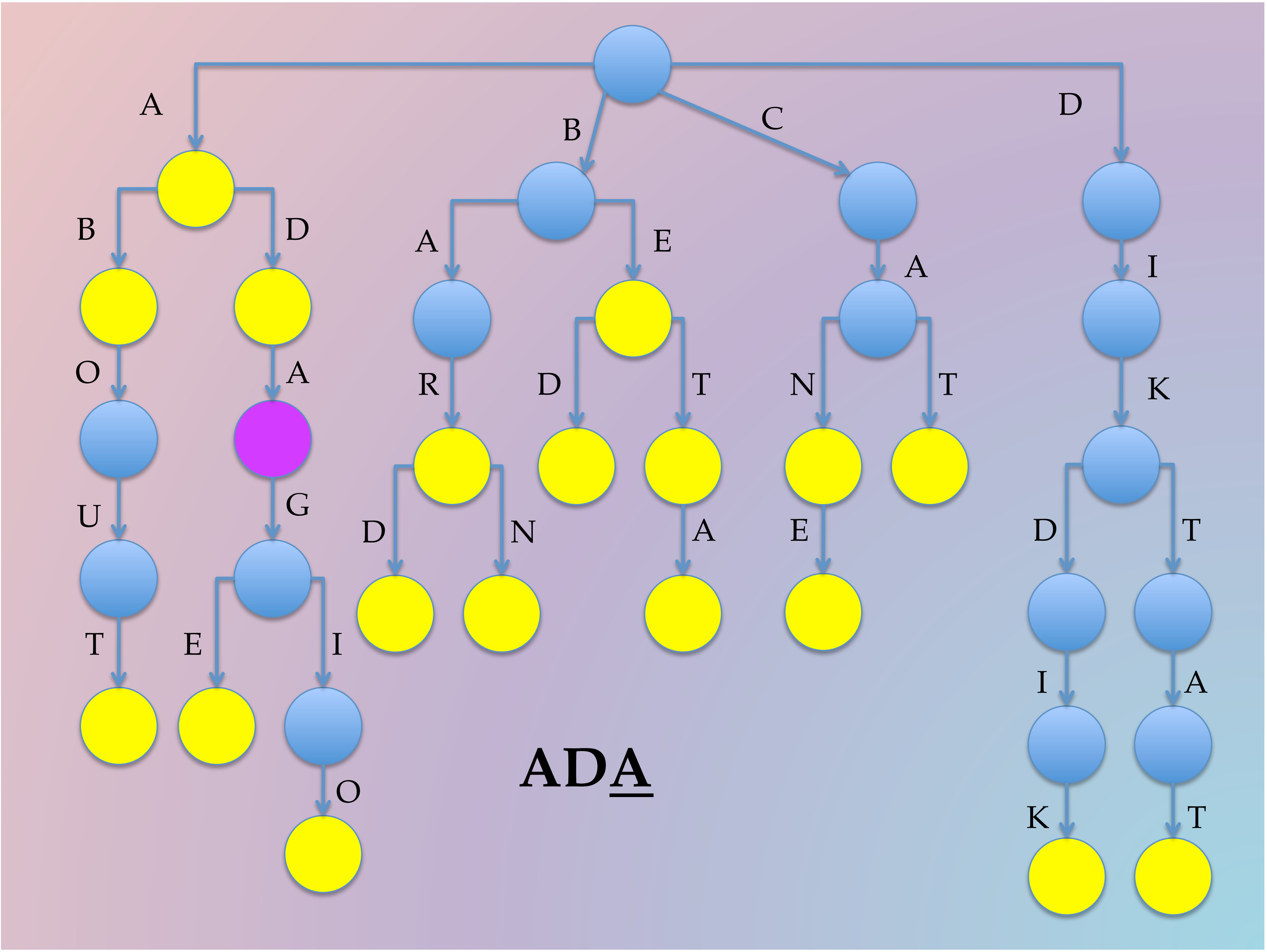

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
ADA
O
K
T

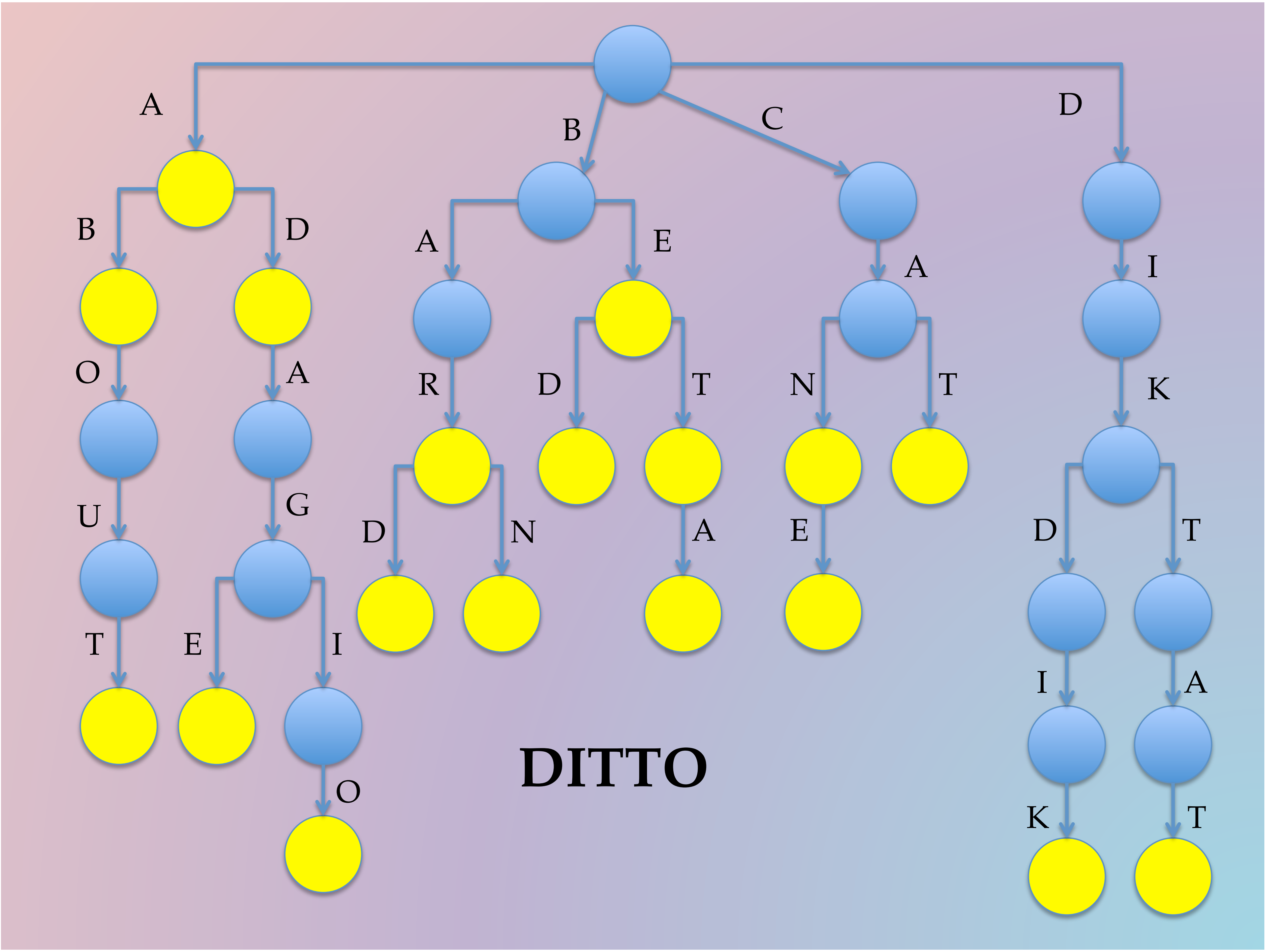

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
ADA

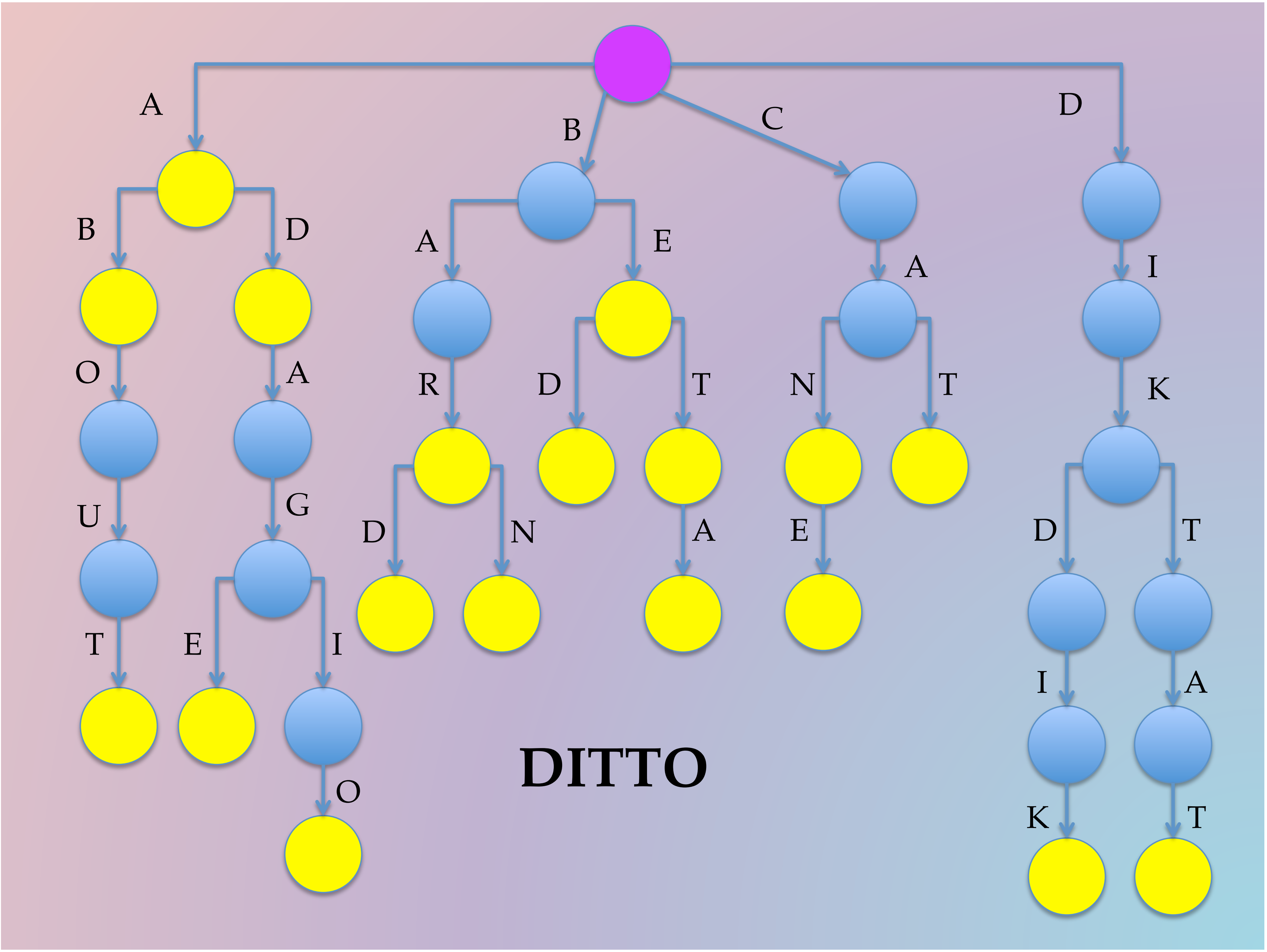


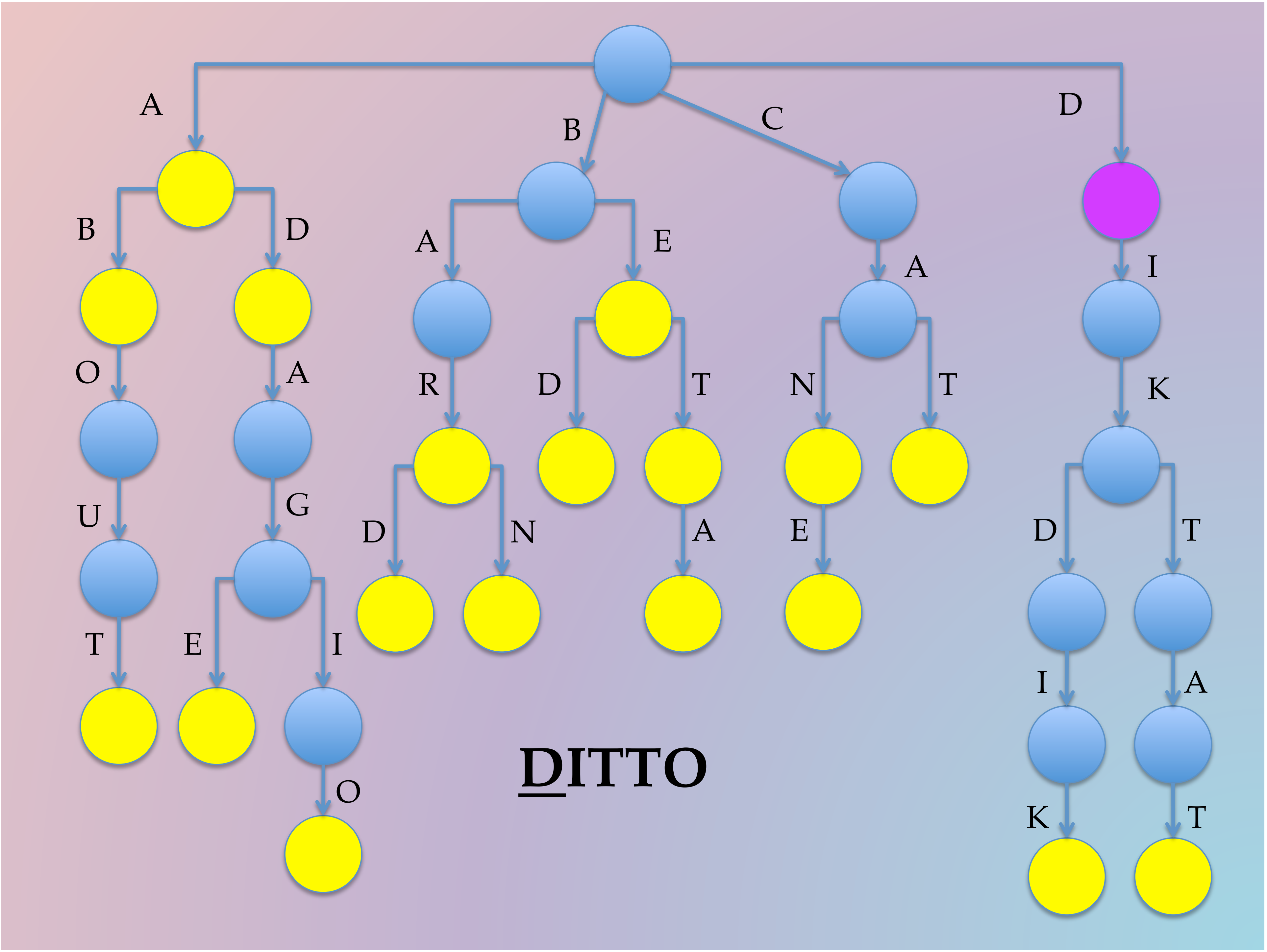
A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
ADA

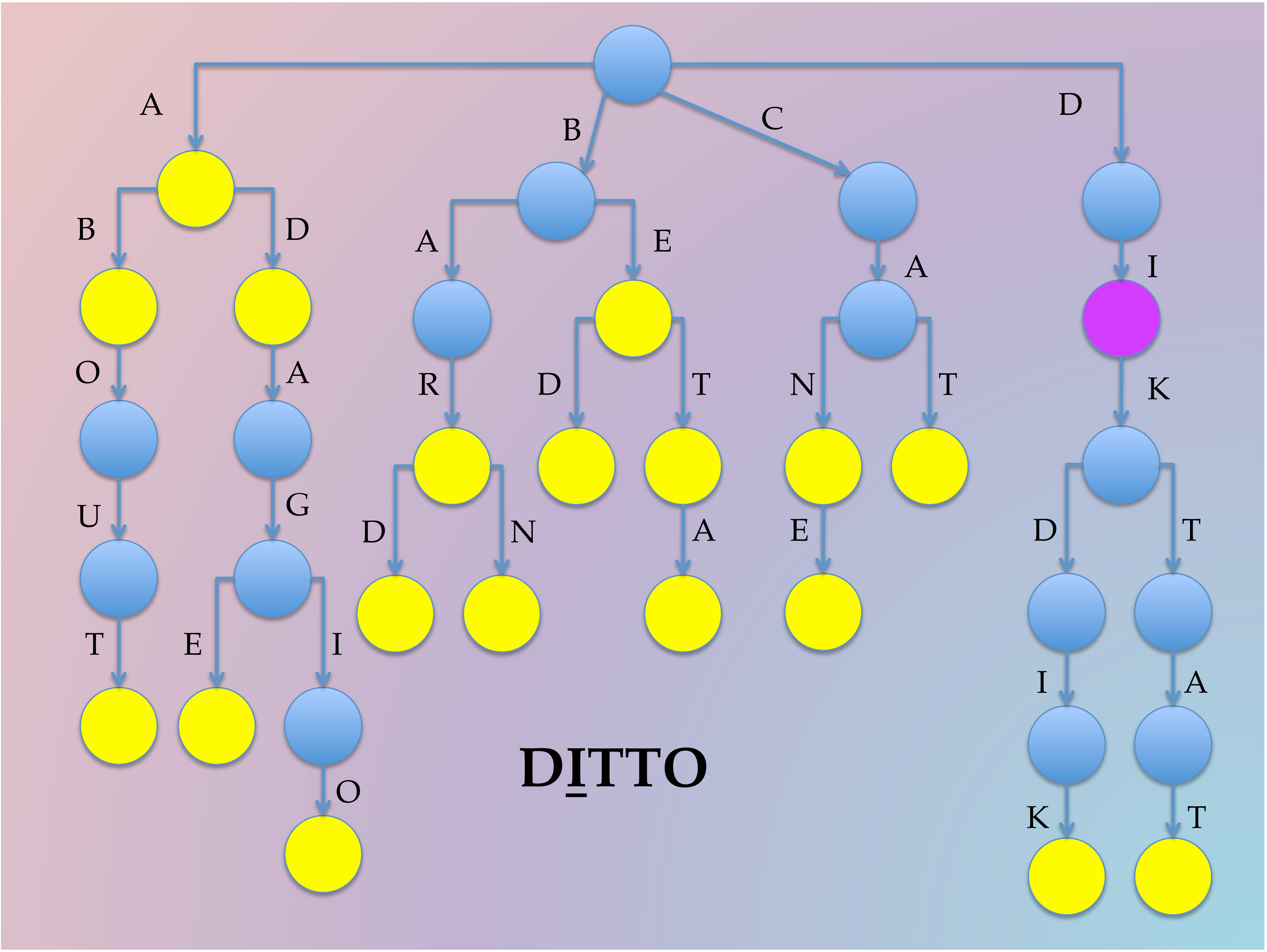

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
ADA

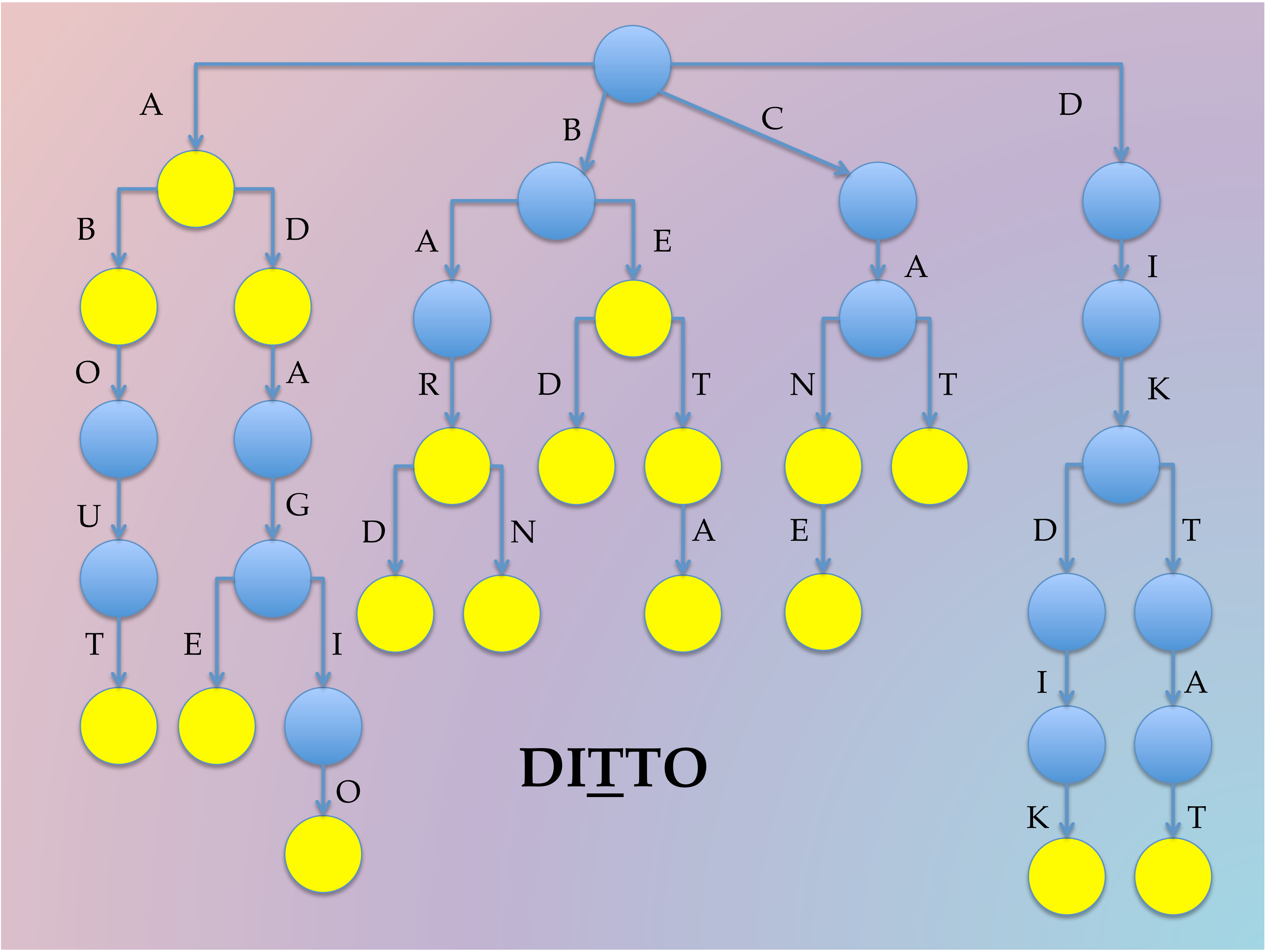

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
ADA


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
DITTO


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
DITTO


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
DITTO


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
DITTO


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
DITTO
O
K
T

Tries

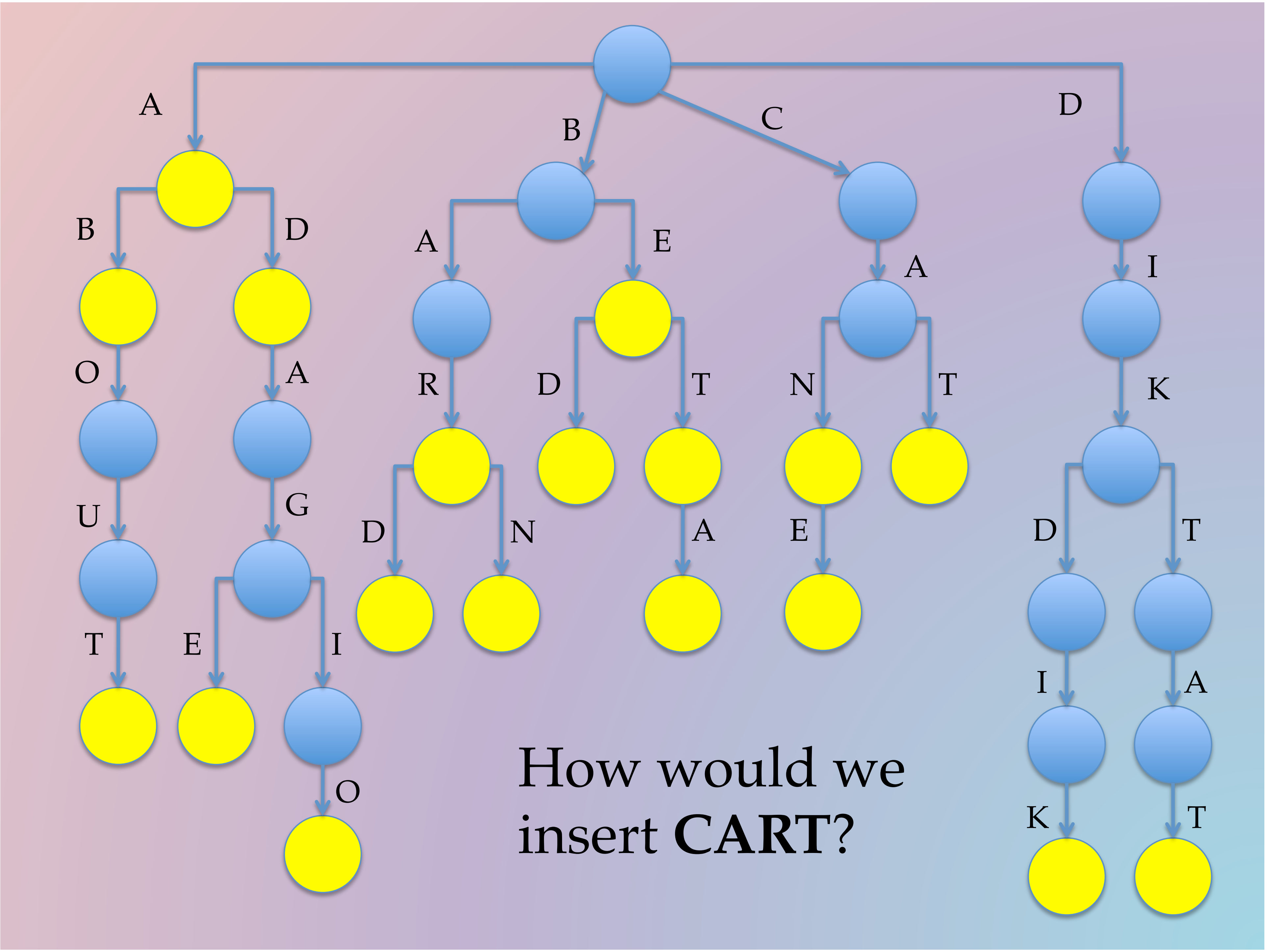
- Congratulations, we’ve just created a **trie**!
- Origin of the word: re**trie**val
- Pronounced “try” (not “tree”) because professors enjoy needlessly confusing students for sport
  - Edward Fredkin, who coined this term, pronounces it “tree”
- Also known as a “prefix tree”

Tries

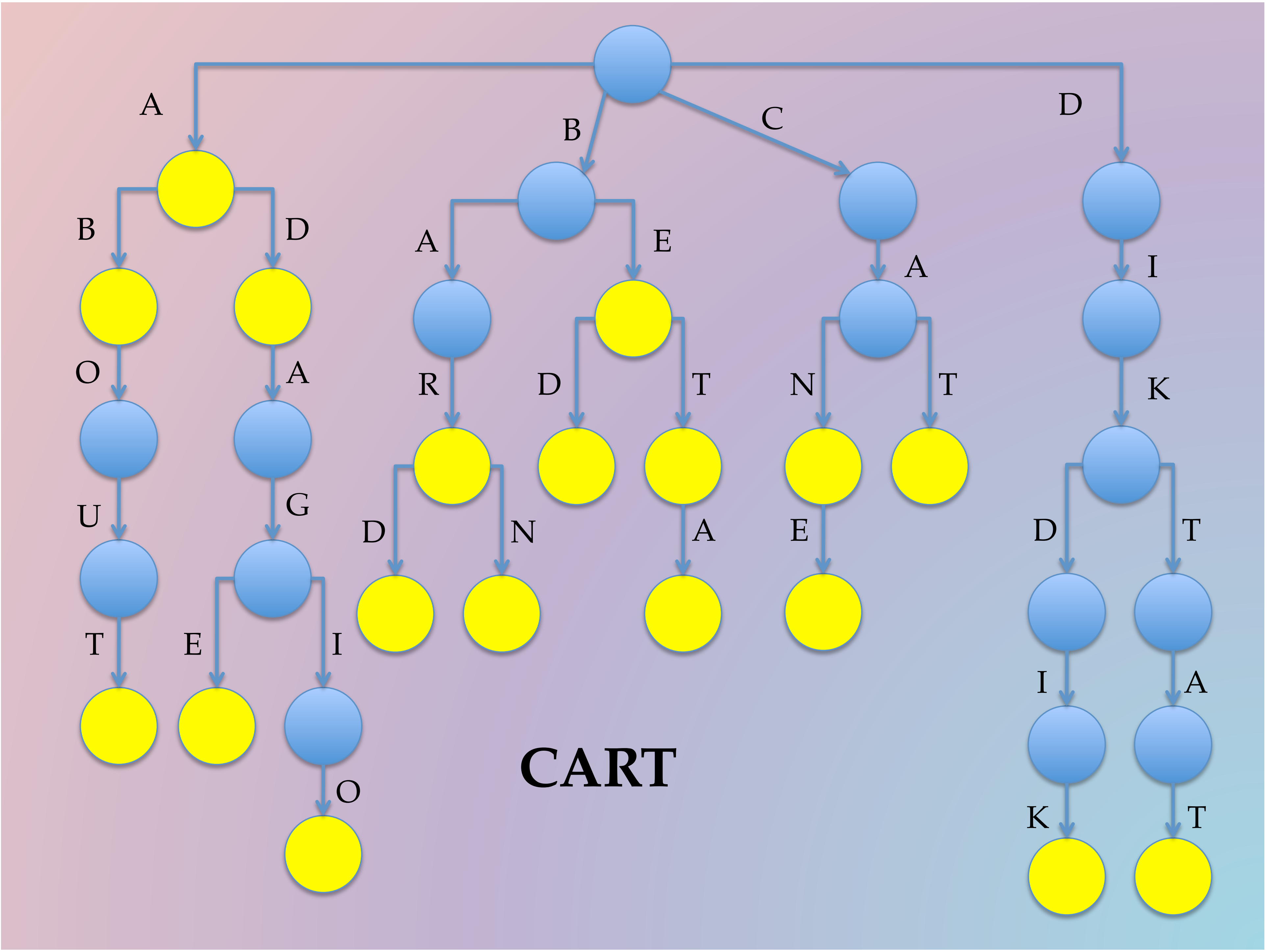
- A **trie** is a tree where each node stores:
  - A bit indicating whether the root-node path to this node represents a valid word
  - A map (could be an array or a tree) from characters to child pointers
- Each node corresponds to the string given by the path traced from the root to that node

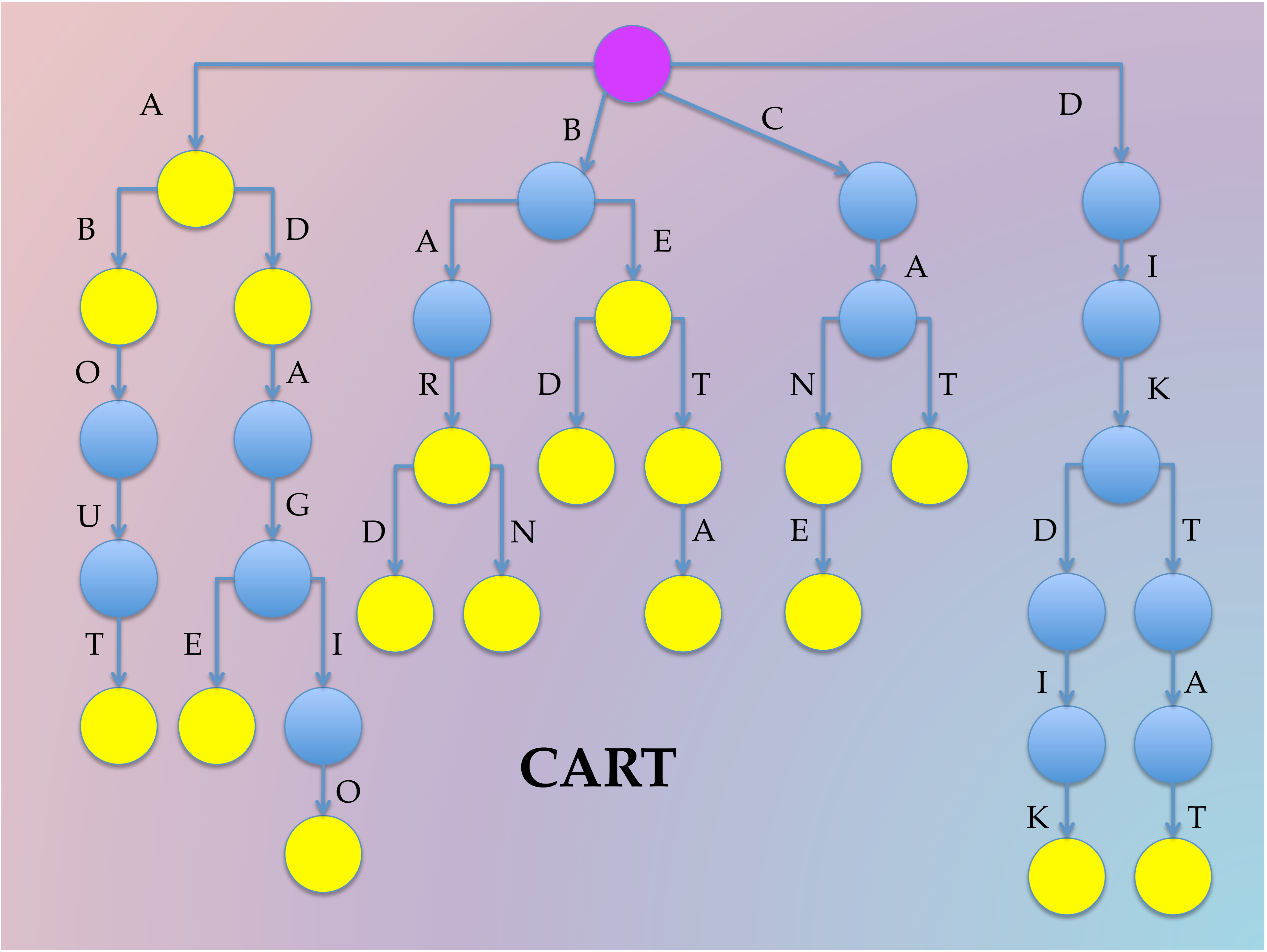
`lookup` and `containsPrefix` in Tries

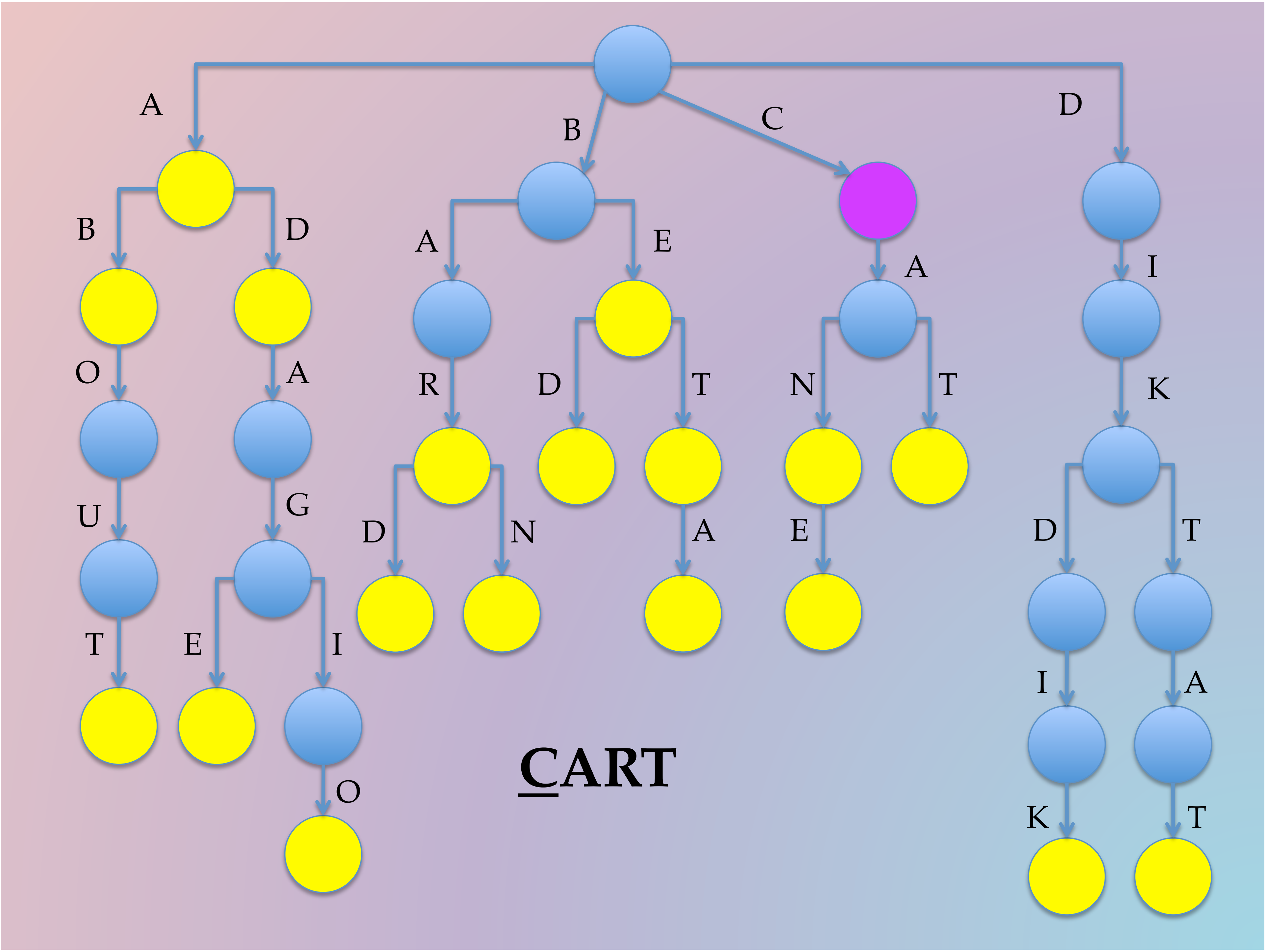
- As we already saw, if a word or prefix has length *L*, we need to follow *L* pointers to get to the node corresponding to that word/prefix
- Assuming each pointer can be accessed/traversed in *O(1)* time, this takes *O(L)* time
- **This is independent of *n*, the number of strings in our trie!**

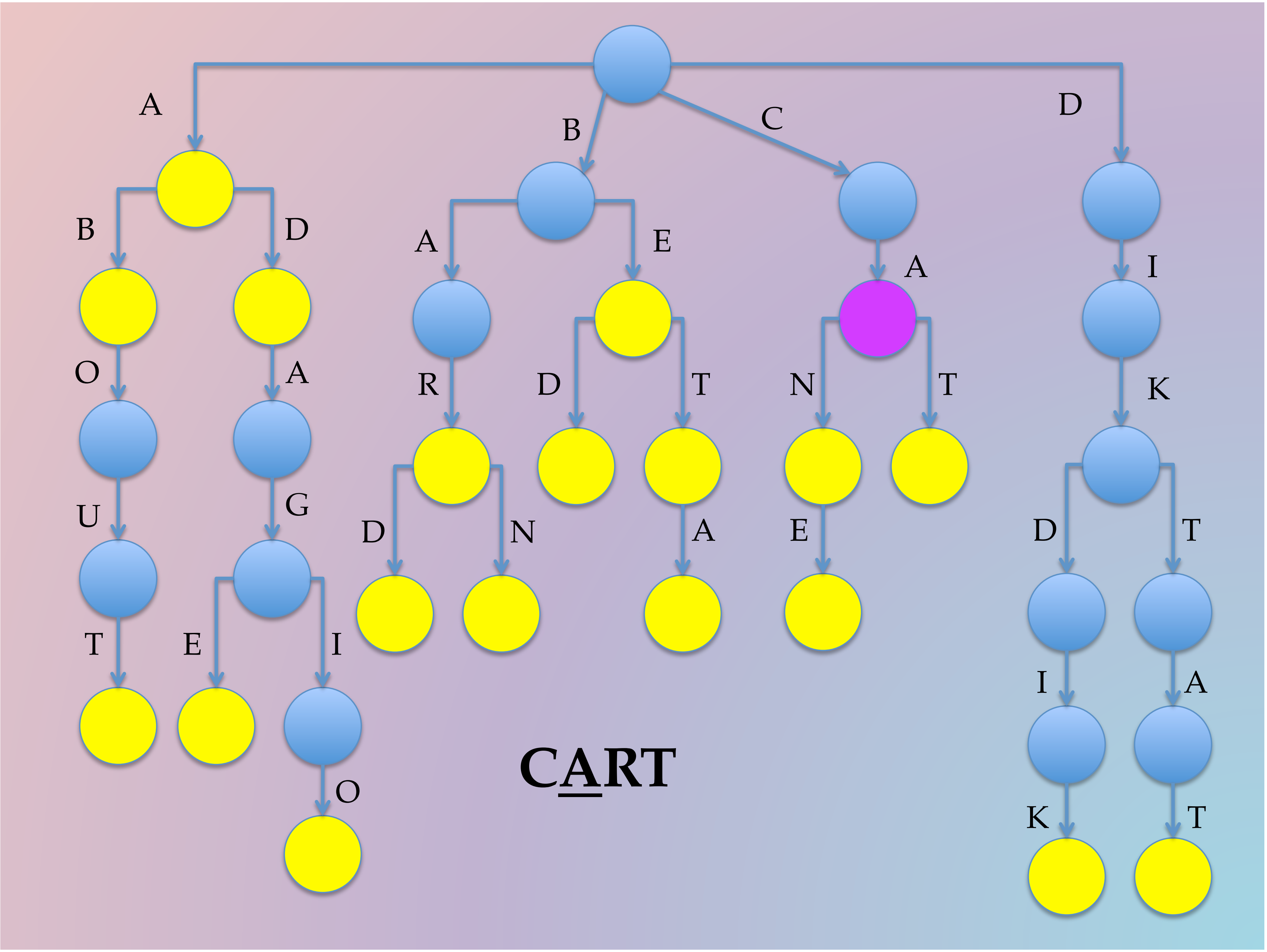


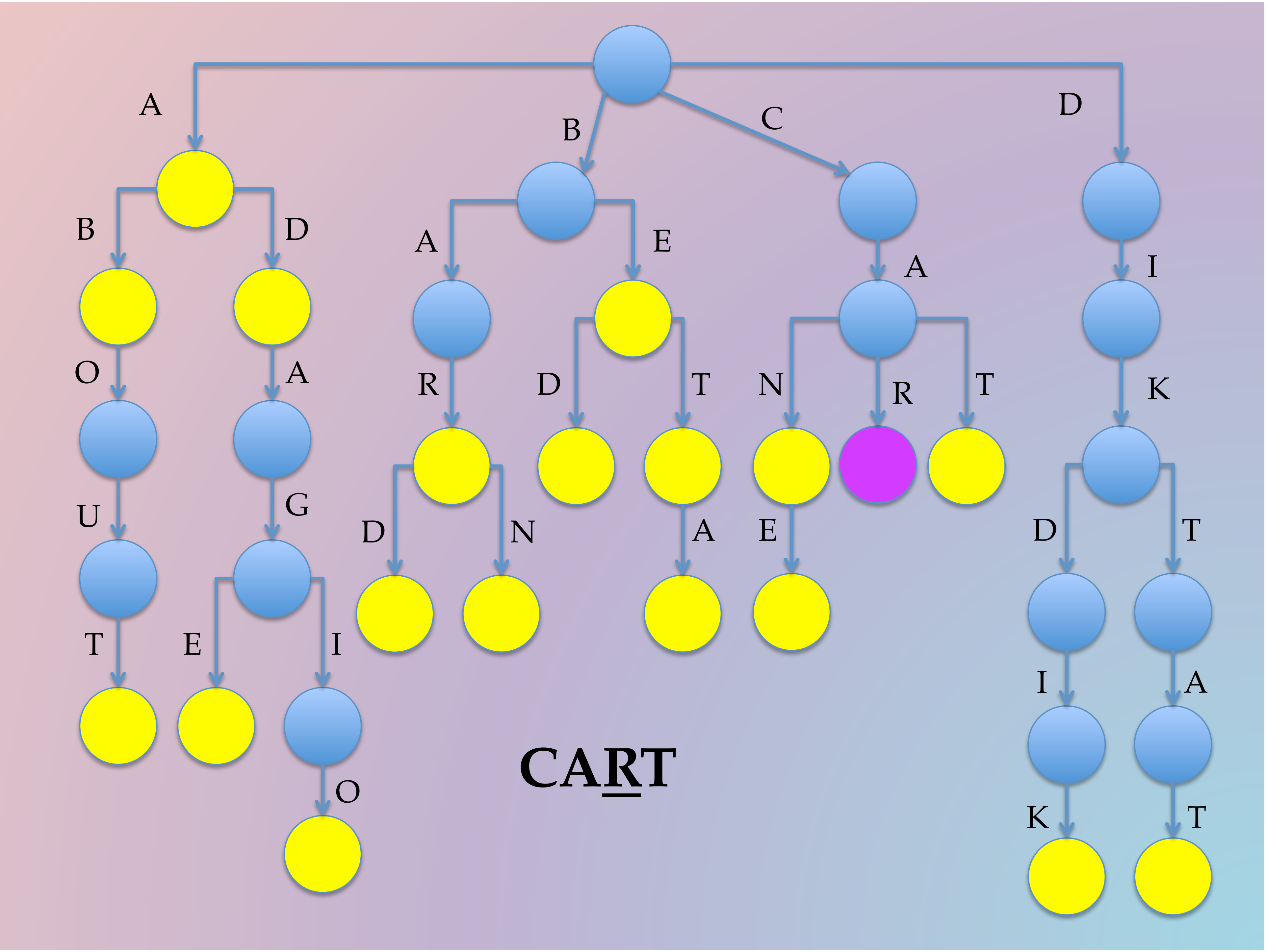
How would we insert **CART**?

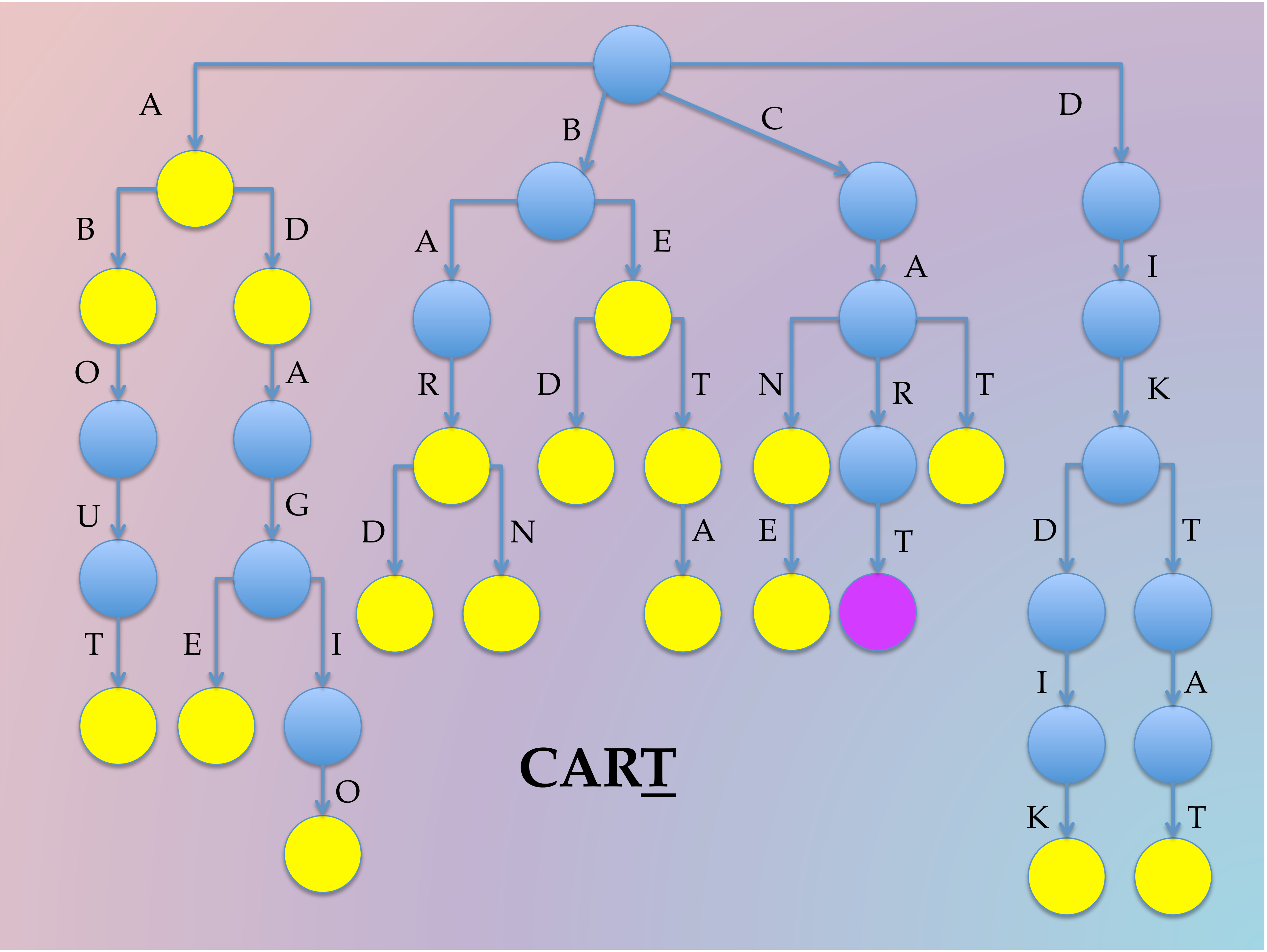

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
CART

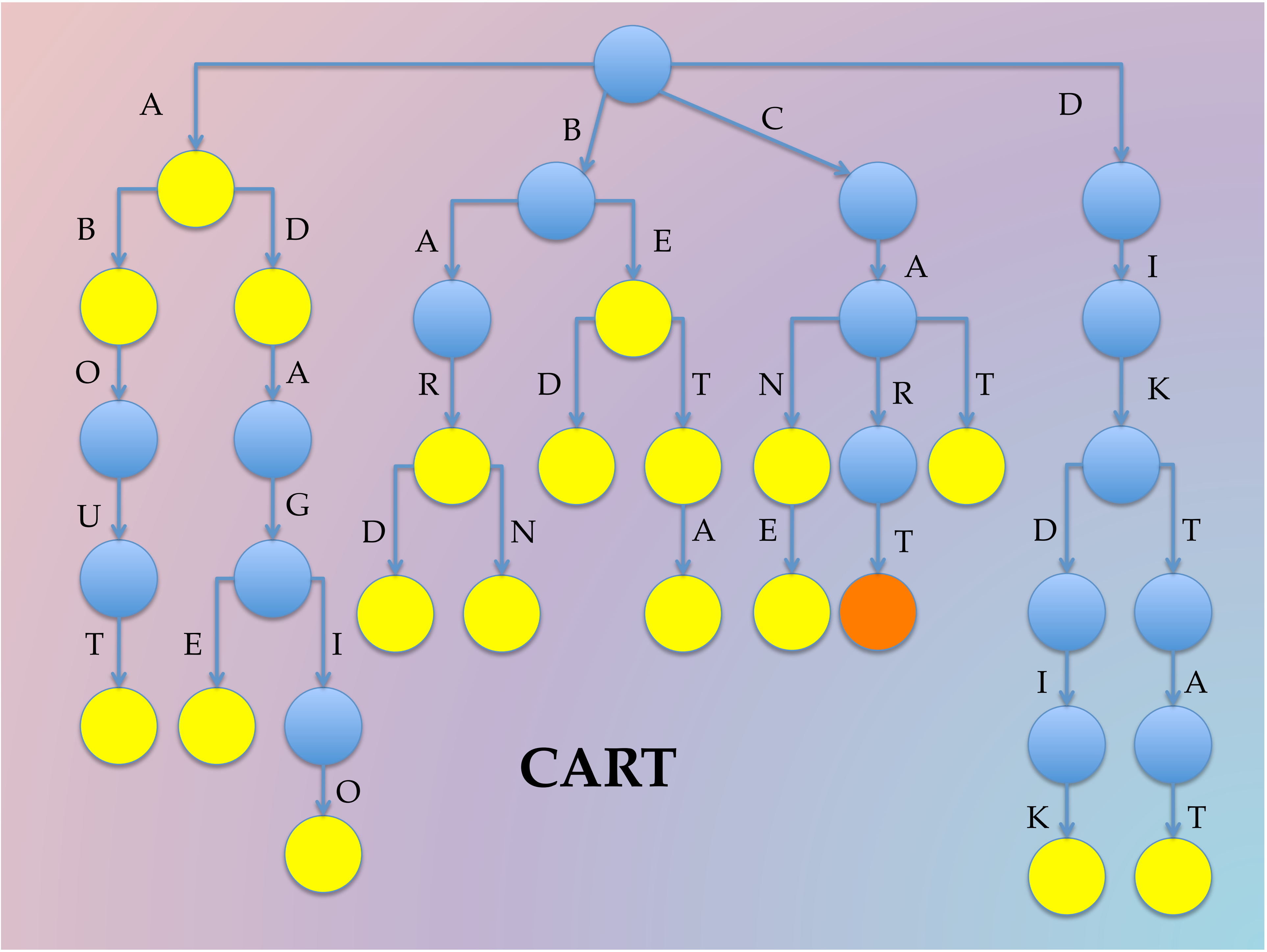

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
CART
K
T

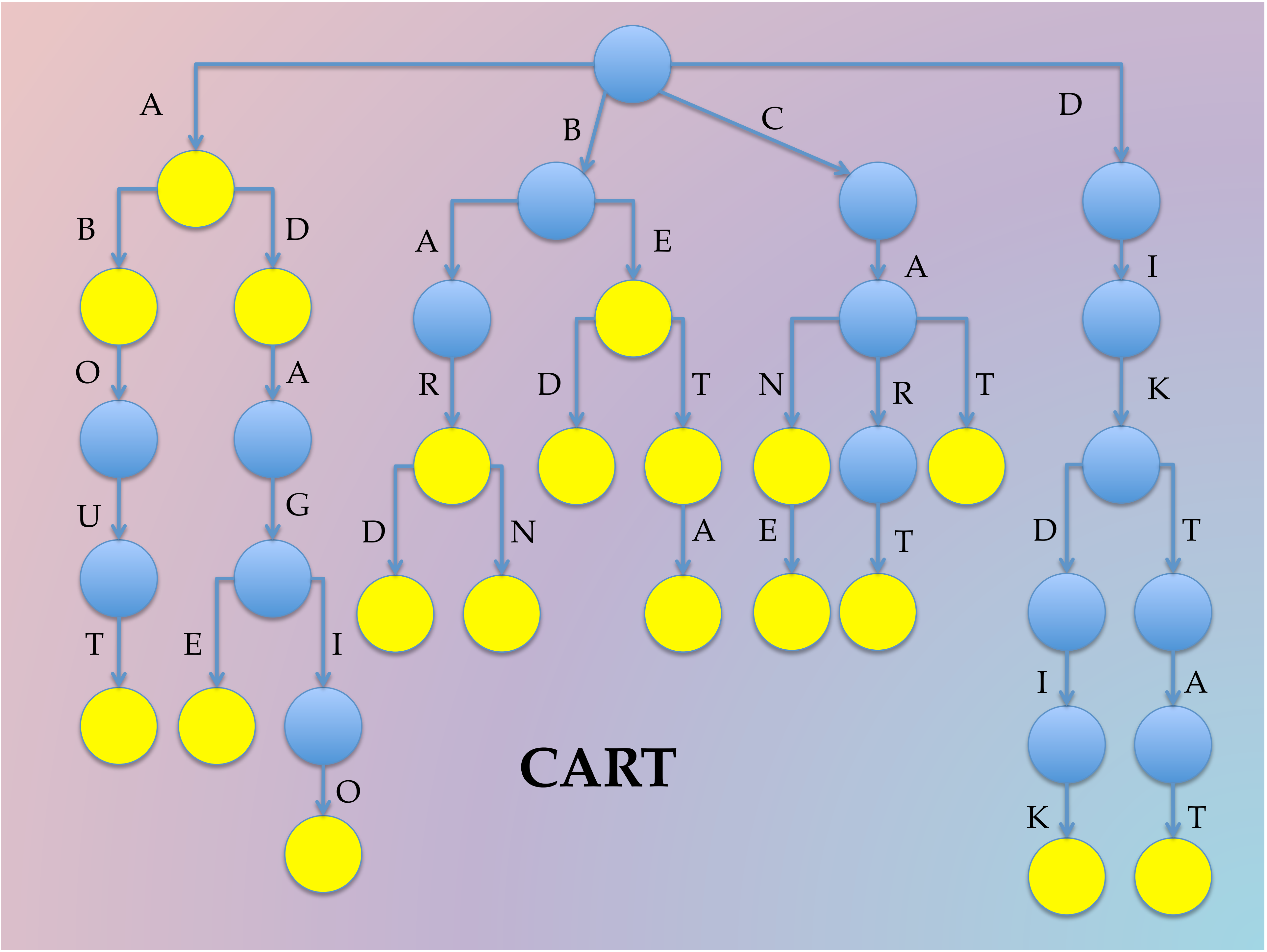

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
CART

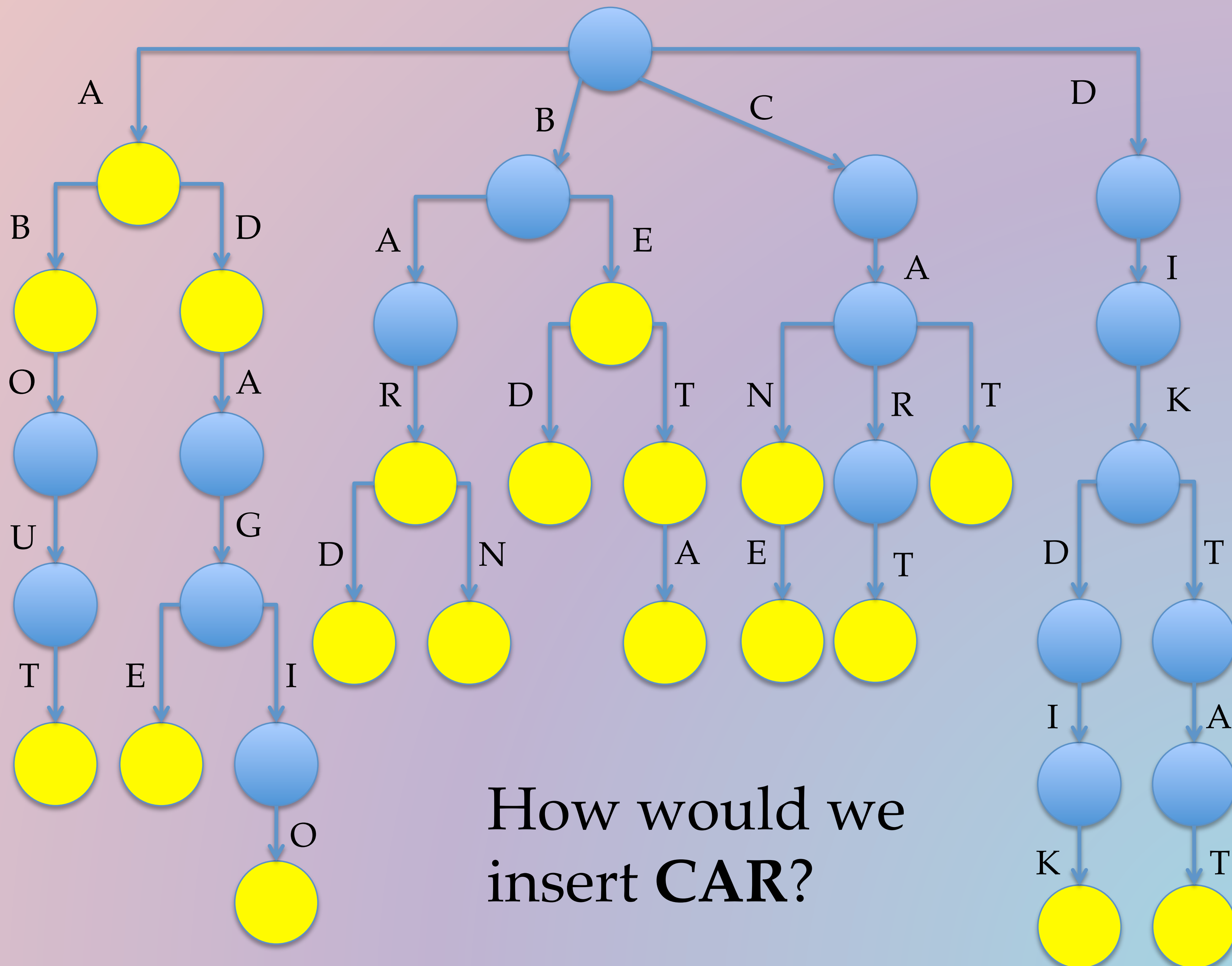

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
T
K
U
G
D
N
A
E
D
T
T
E
I
I
A
O
K
T
CART


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

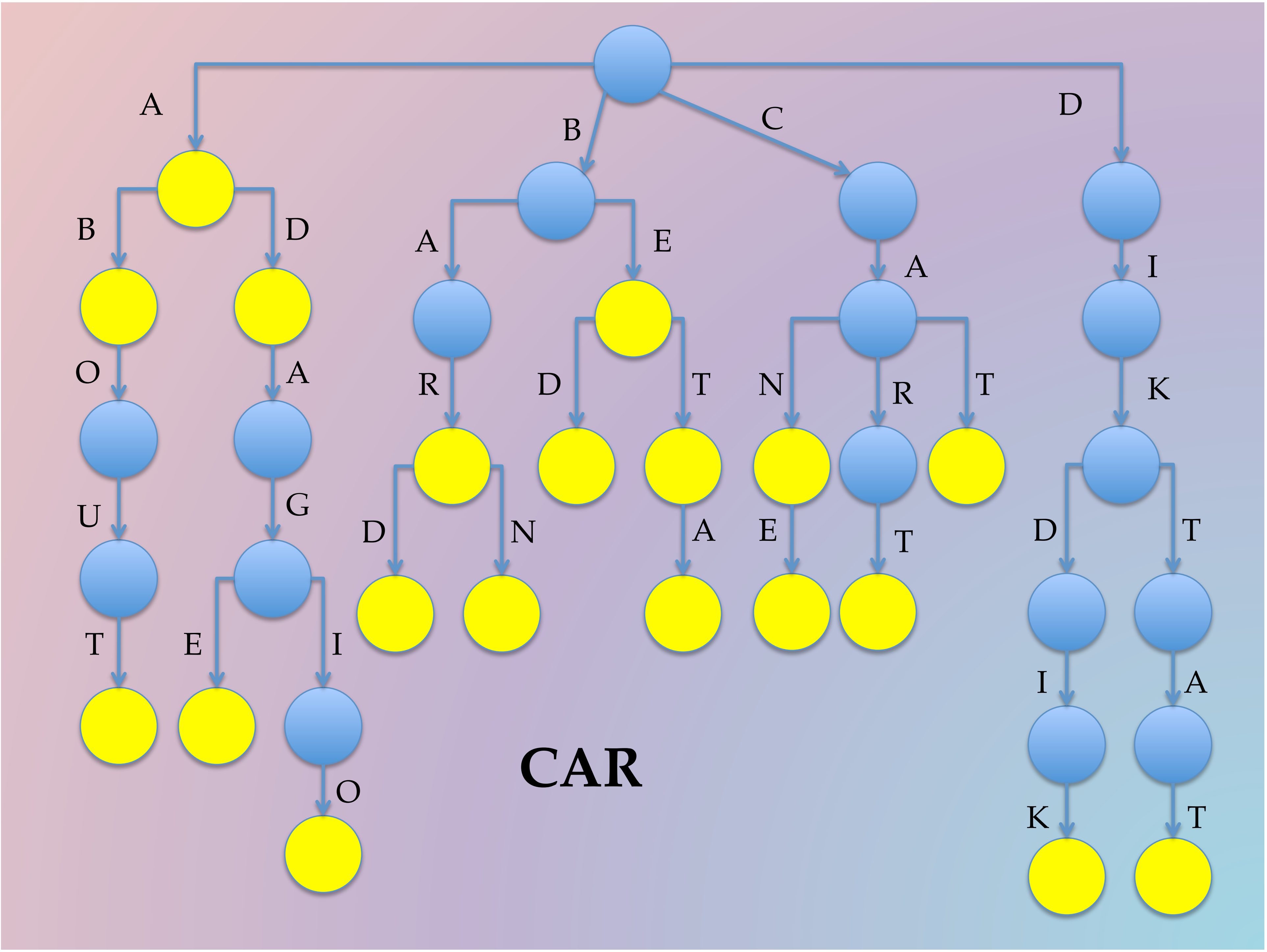

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
R
T
K
U
G
D
N
A
E
T
D
T
T
E
I
I
A
O
K
T
CART

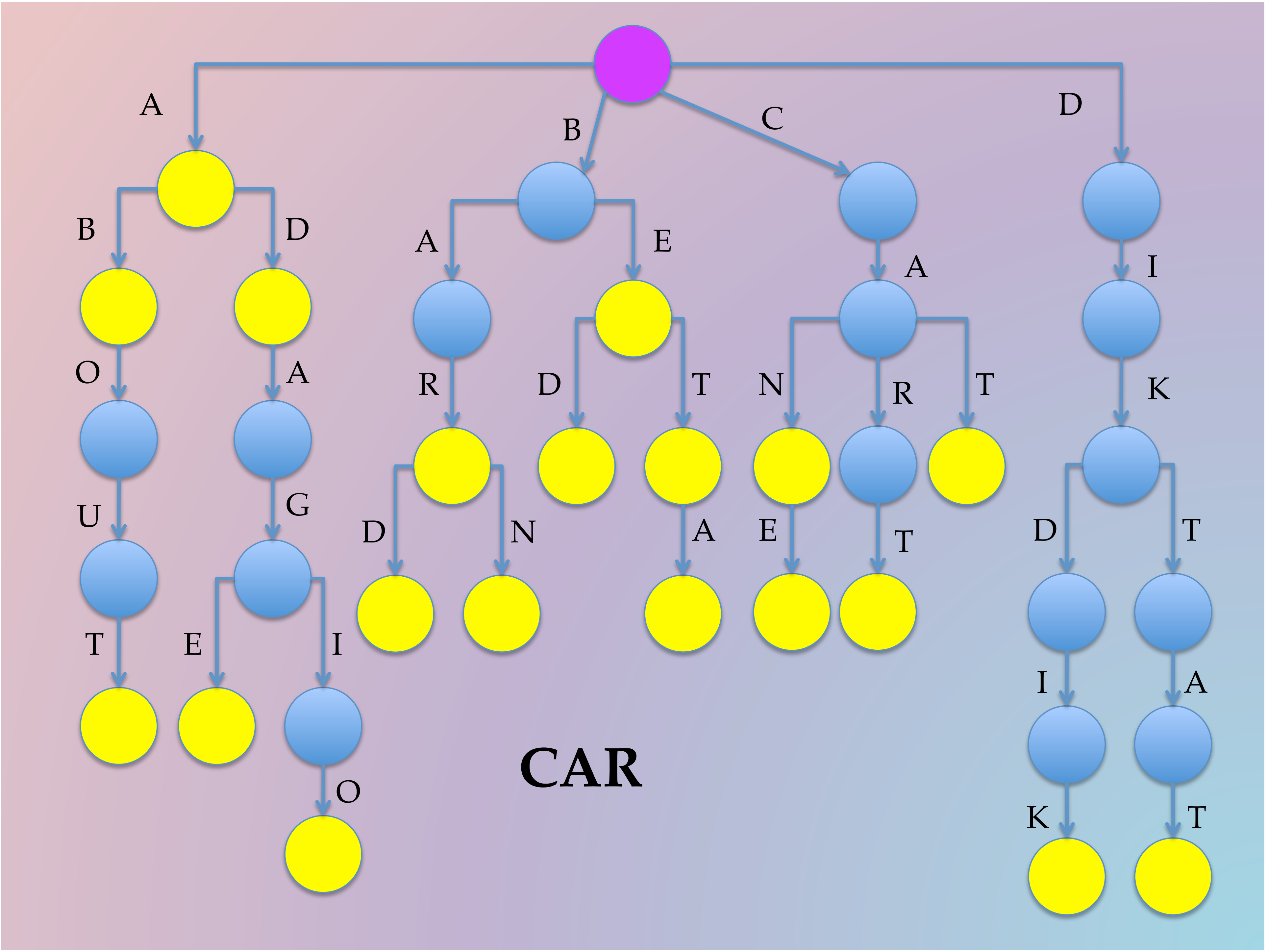

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
R
T
K
U
G
D
N
A
E
T
D
T
T
E
I
I
A
O
K
T
CART

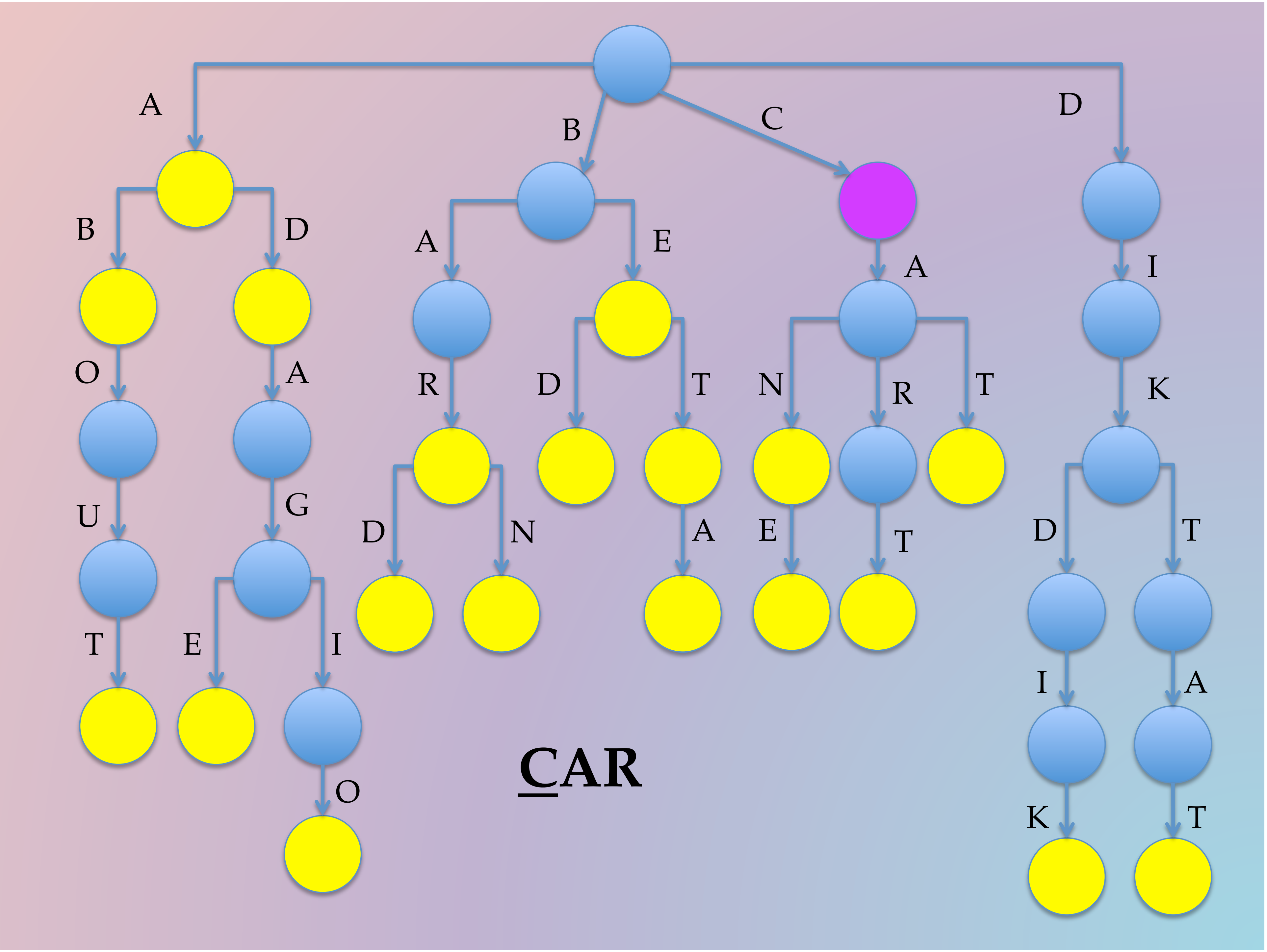

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
R
T
K
U
G
D
N
A
E
T
D
T
T
E
I
I
A
O
CART
K
T

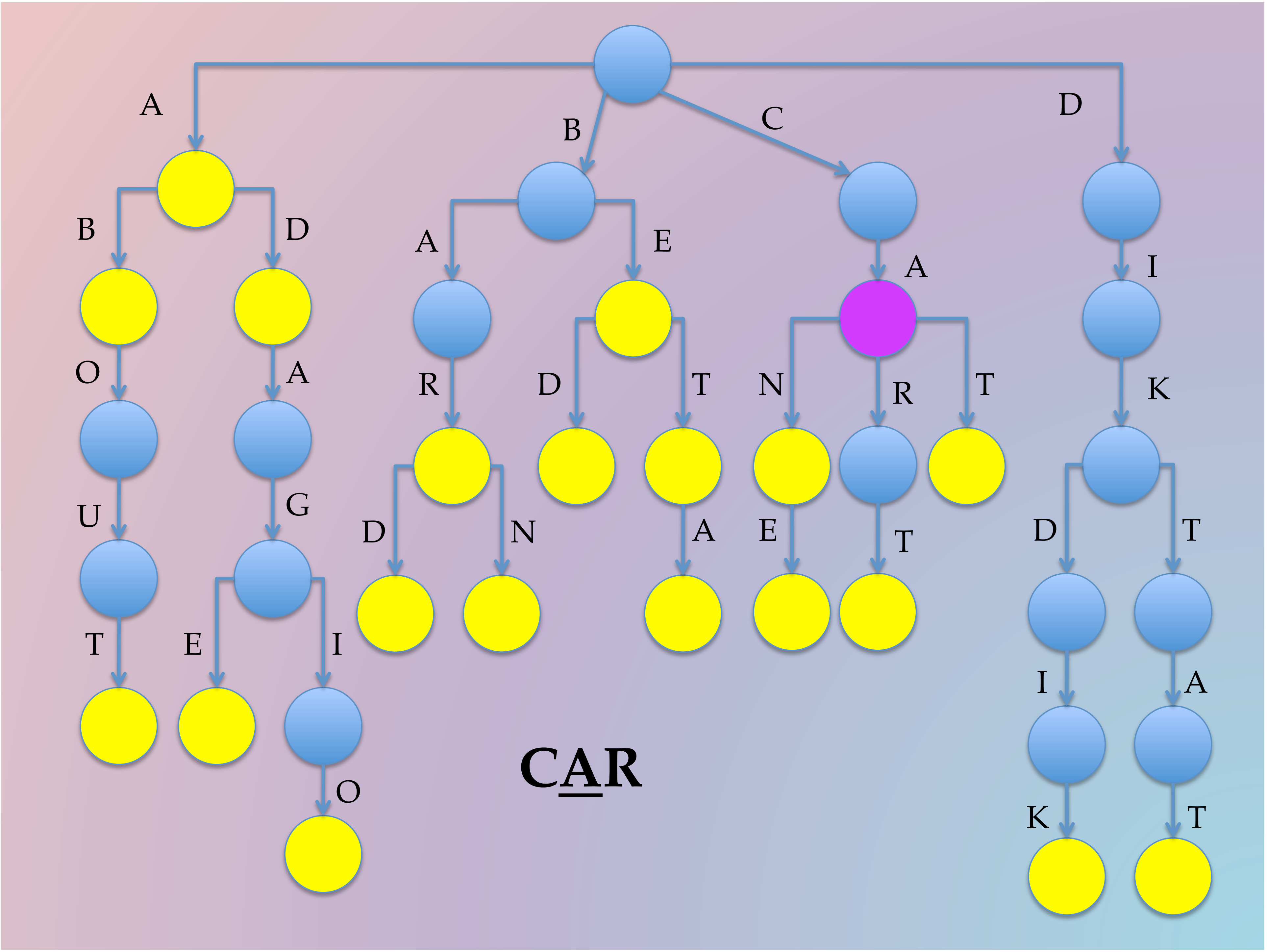


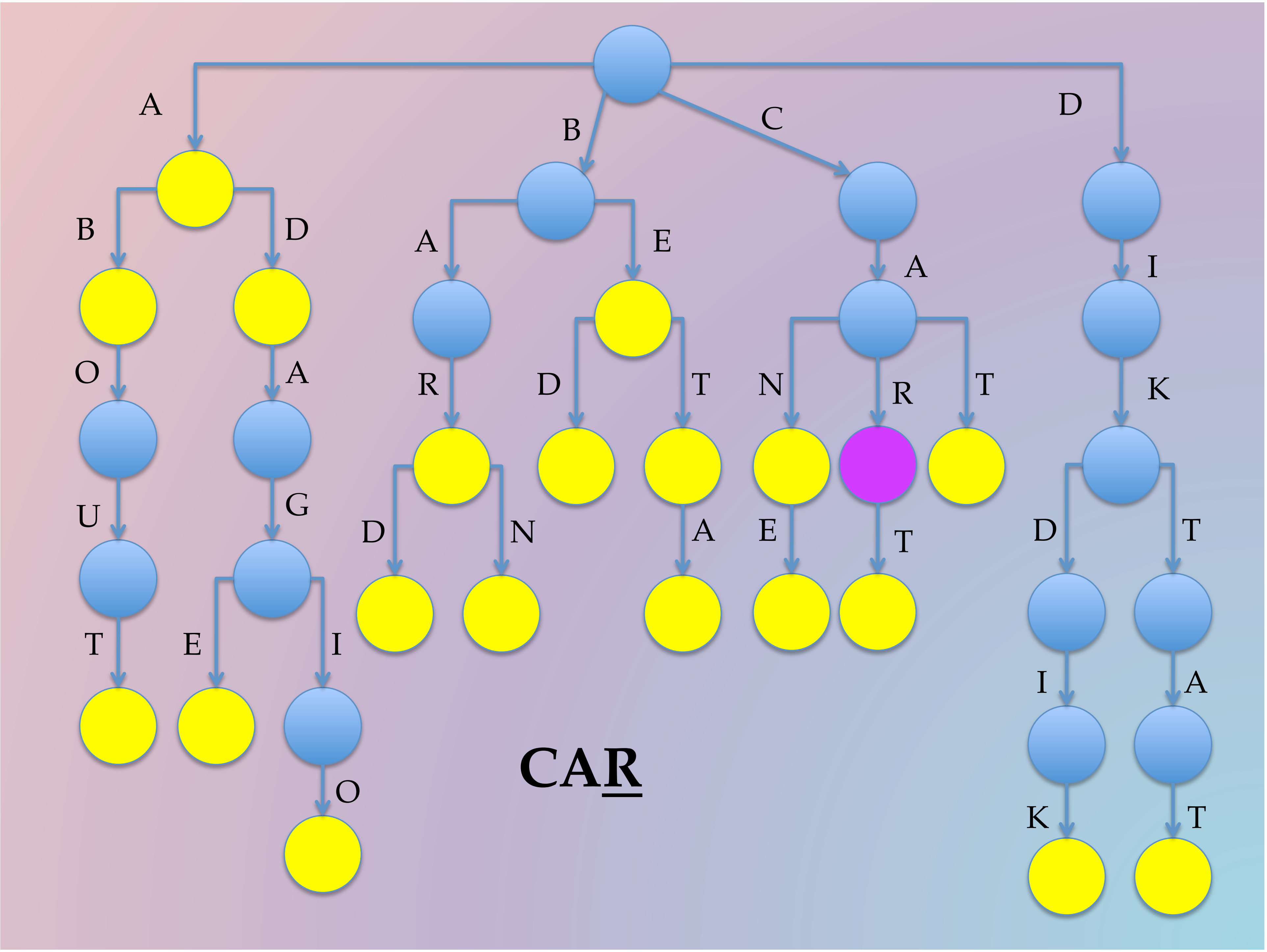
How would we insert **CAR**?

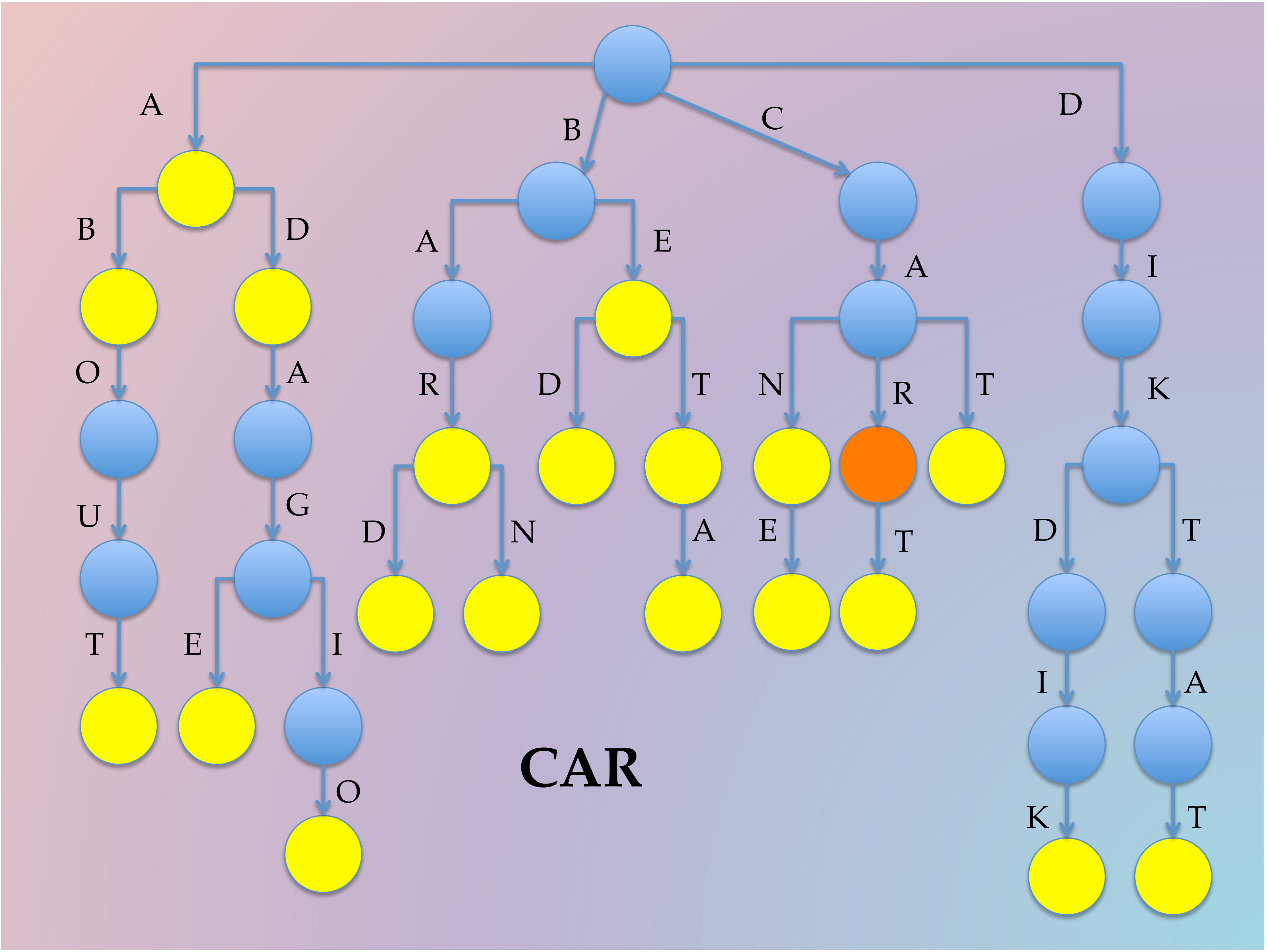

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
R
T
K
U
G
D
N
A
E
T
D
T
T
E
I
I
A
CAR
O
K
T

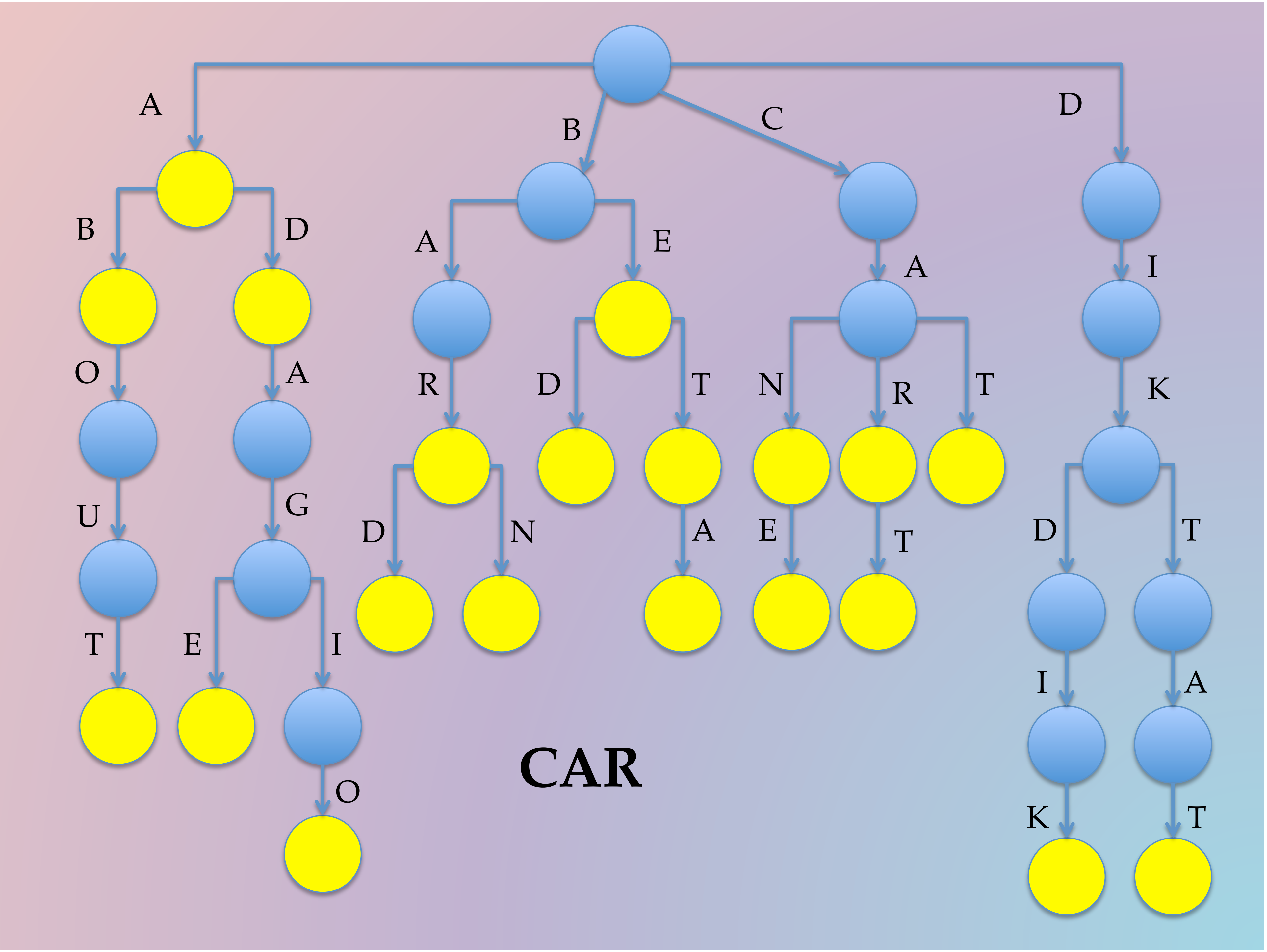

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
R
T
K
U
G
D
N
A
E
T
D
T
T
E
I
I
A
O
K
T
CAR


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


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


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


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

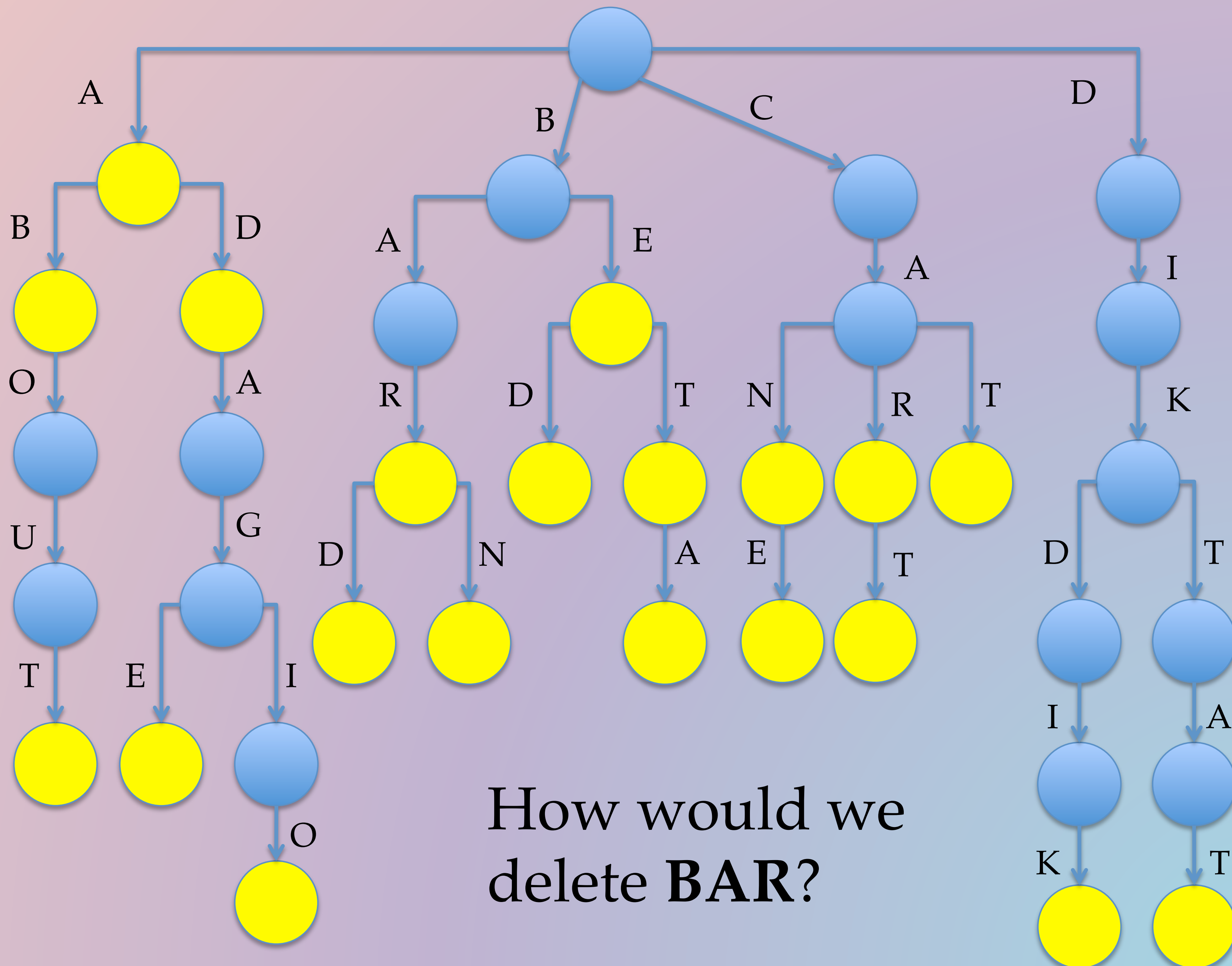

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
R
T
K
U
G
D
N
A
E
T
D
T
T
E
I
I
A
O
K
T
CAR

`insert` in Tries

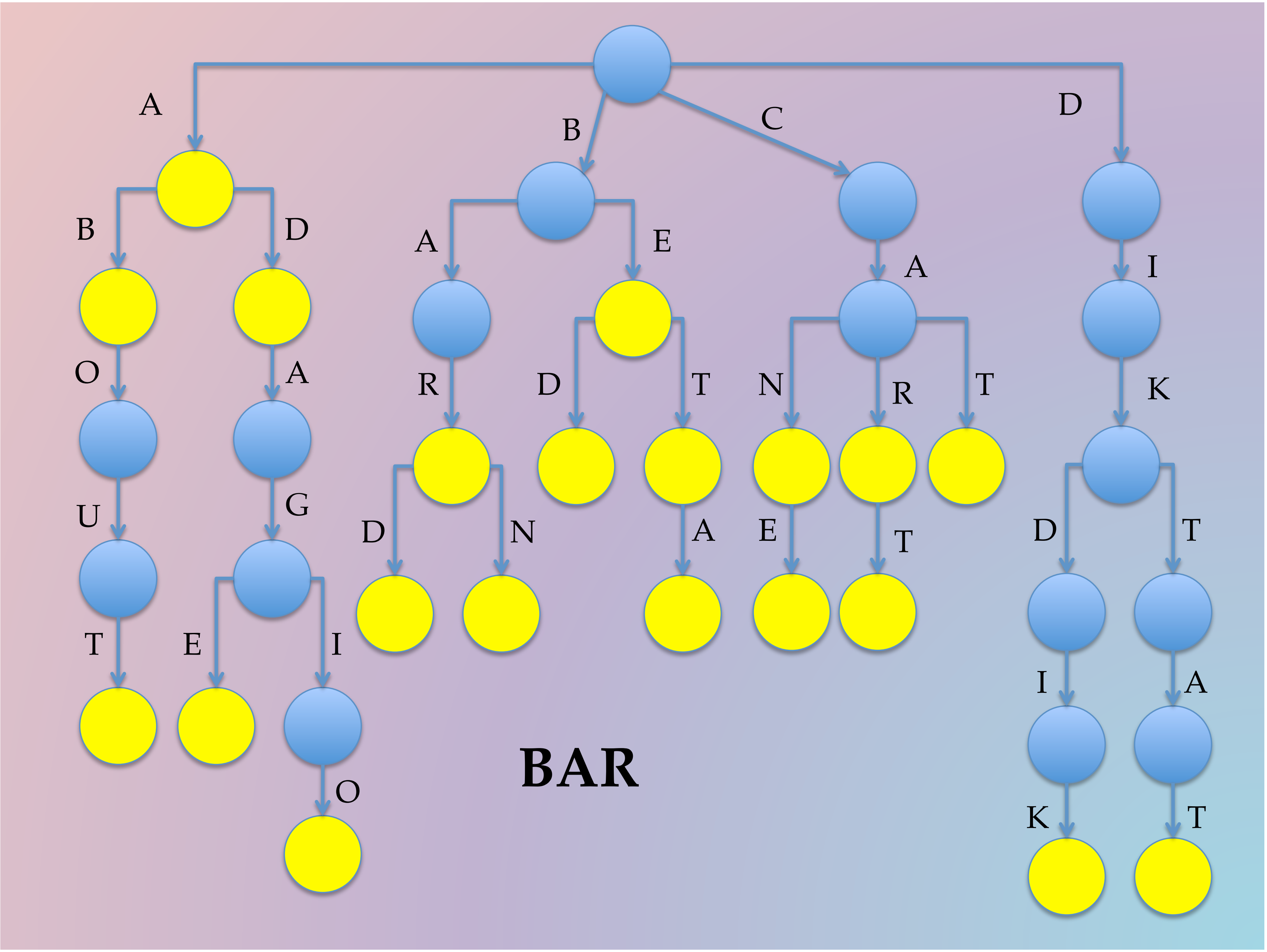
- How long does it take to `insert` a word of length L into a trie with n nodes?
  - A) $O(1)$
  - B) $O(L)$
  - C) $O(\log n)$
  - D) $O(L \log n)$
  - E) Other/none of the above/multiple of the above/unknowable

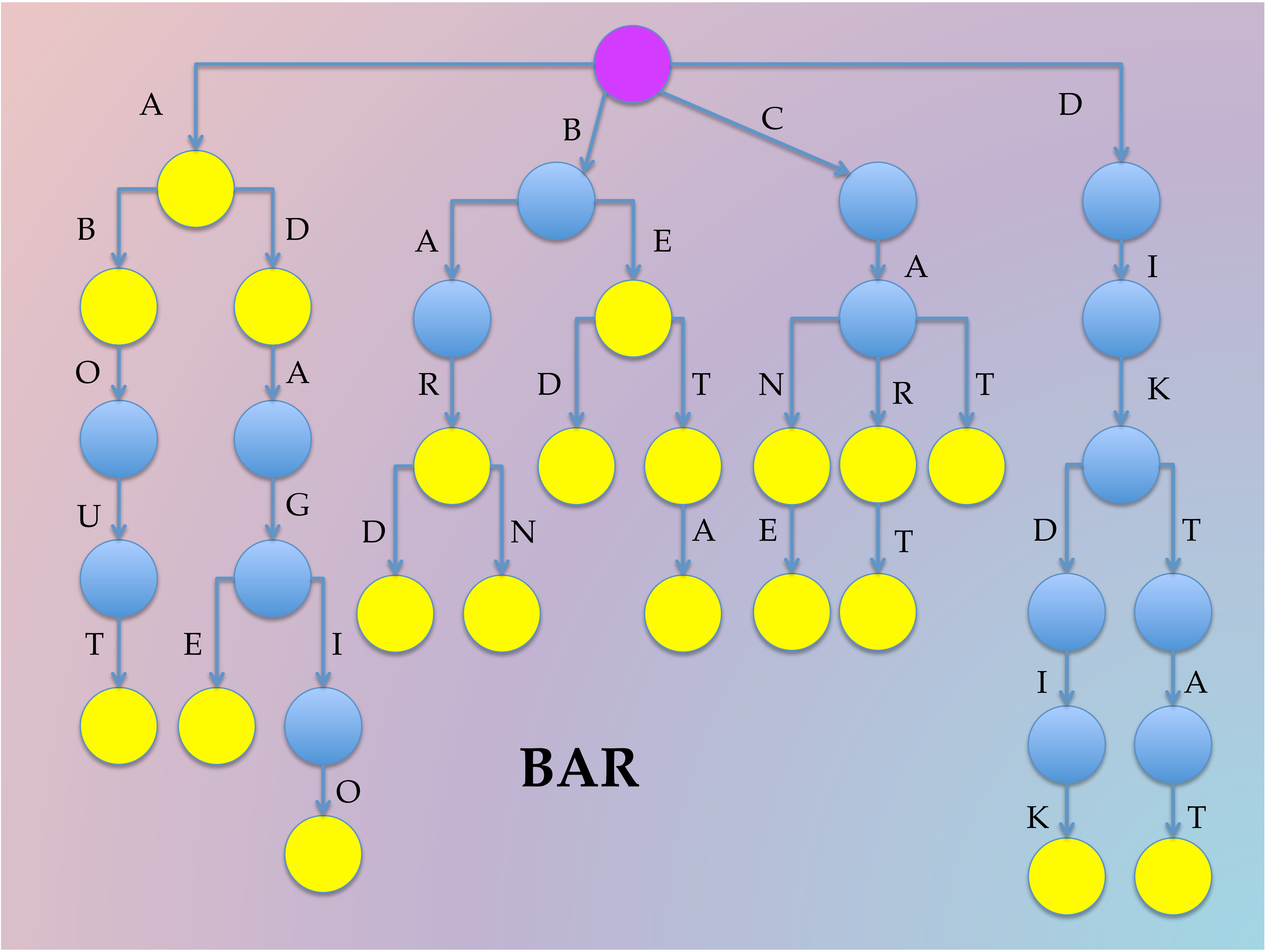
`insert` in Tries

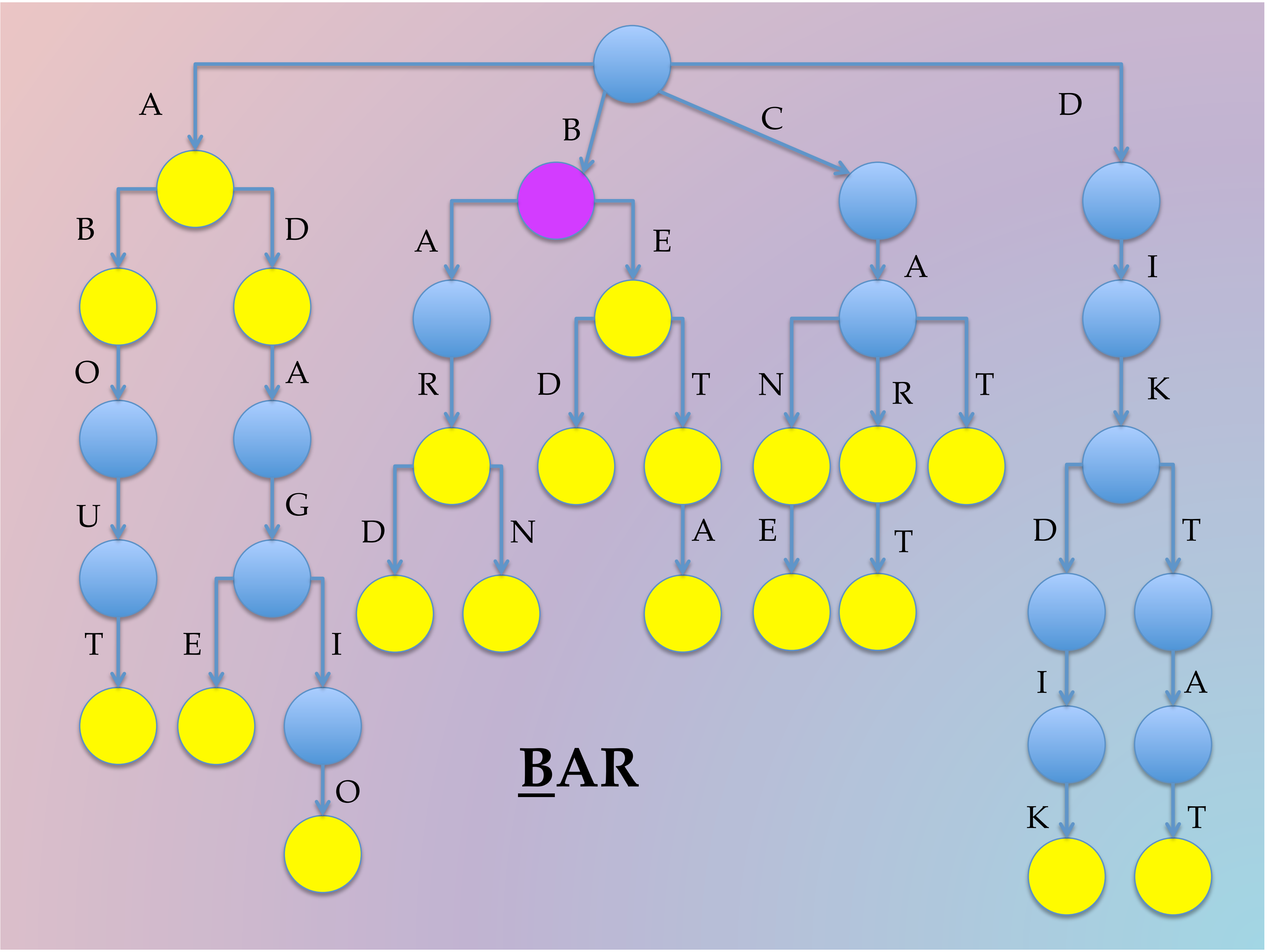
- How long does it take to `insert` a word of length L into a trie with n nodes?
  - A) $O(1)$
  - **B) $O(L)$**
  - C) $O(\log n)$
  - D) $O(L \log n)$
  - E) Other/none of the above/multiple of the above/unknowable

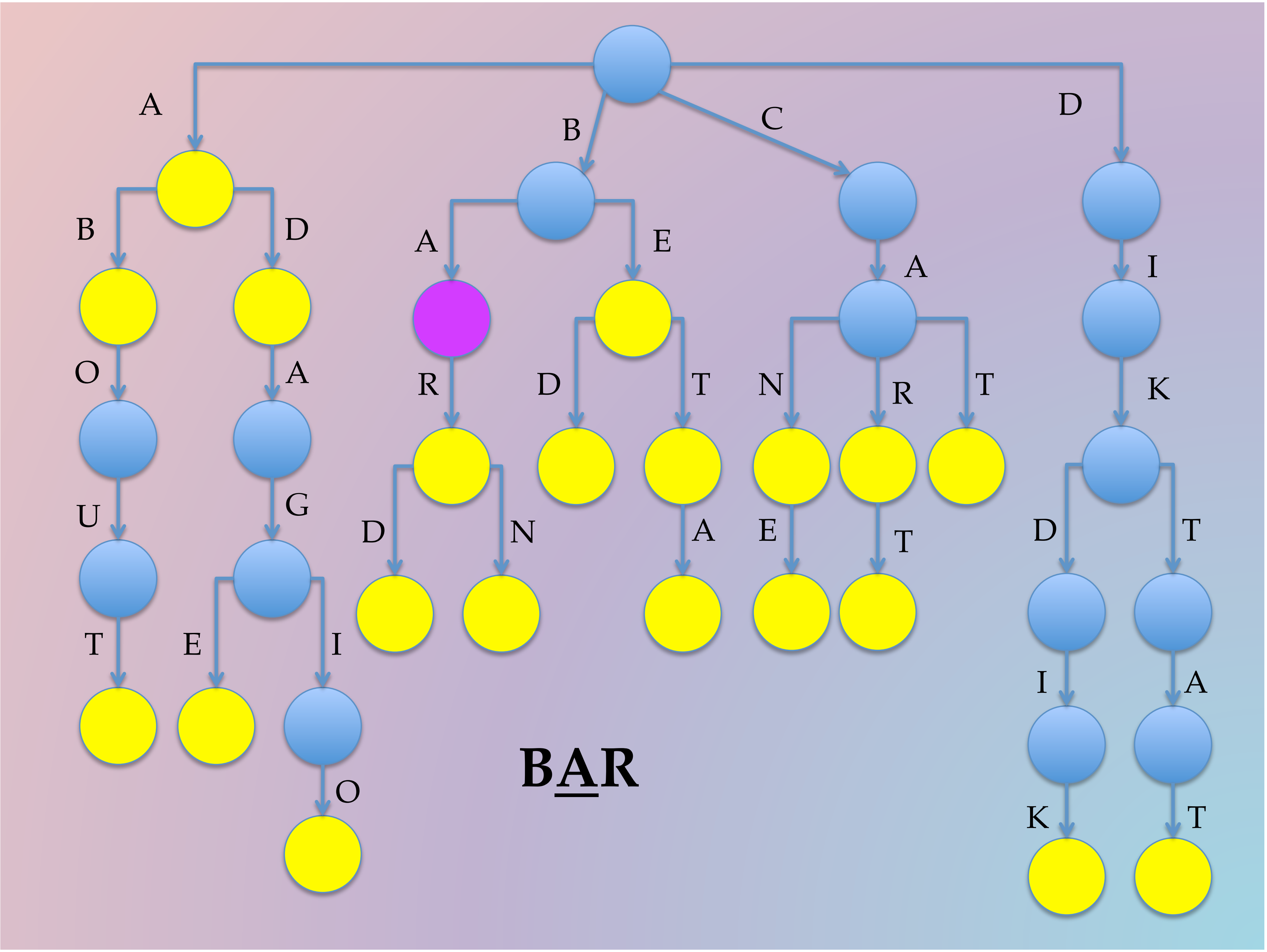


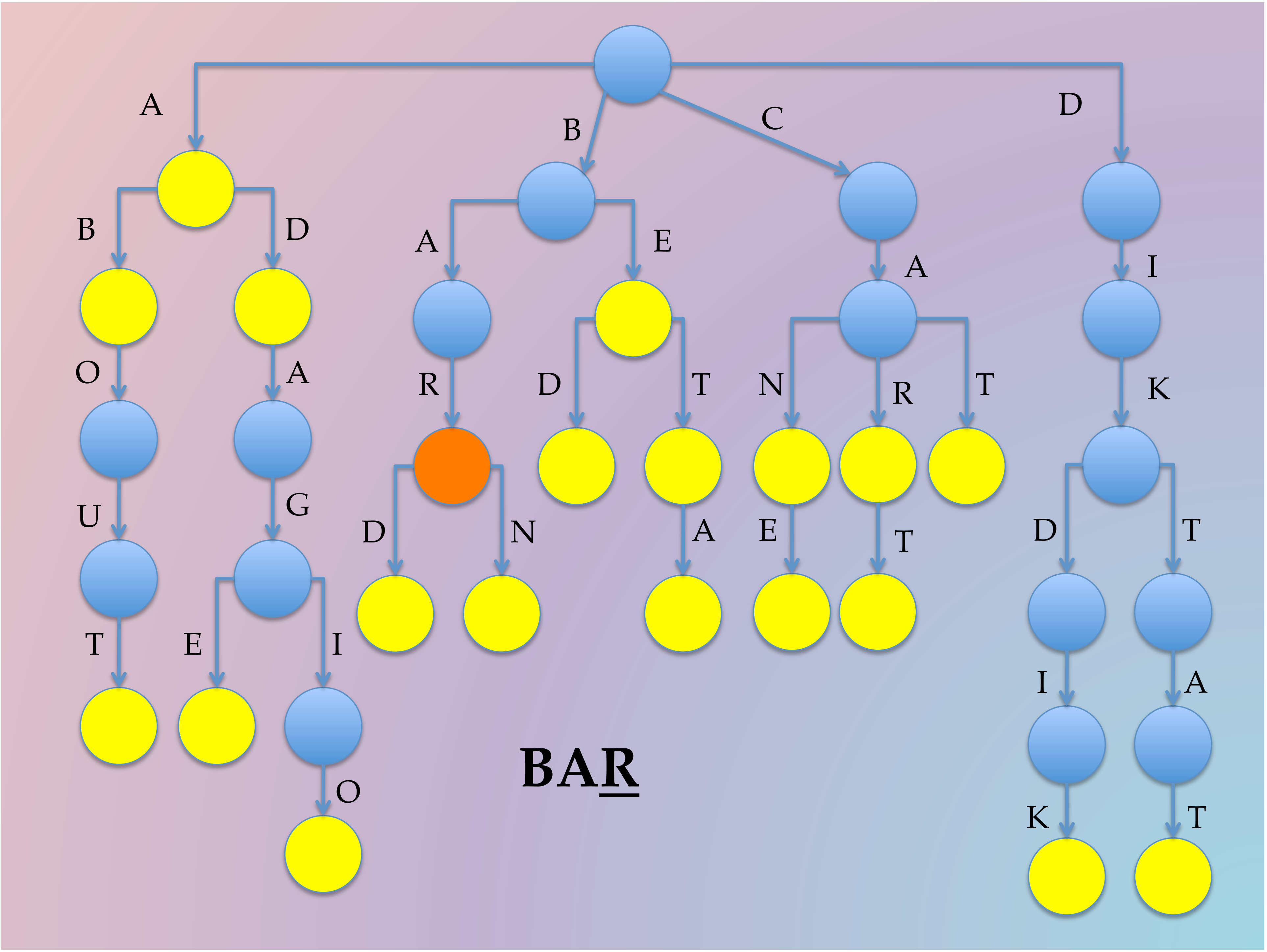
How would we delete **BAR**?

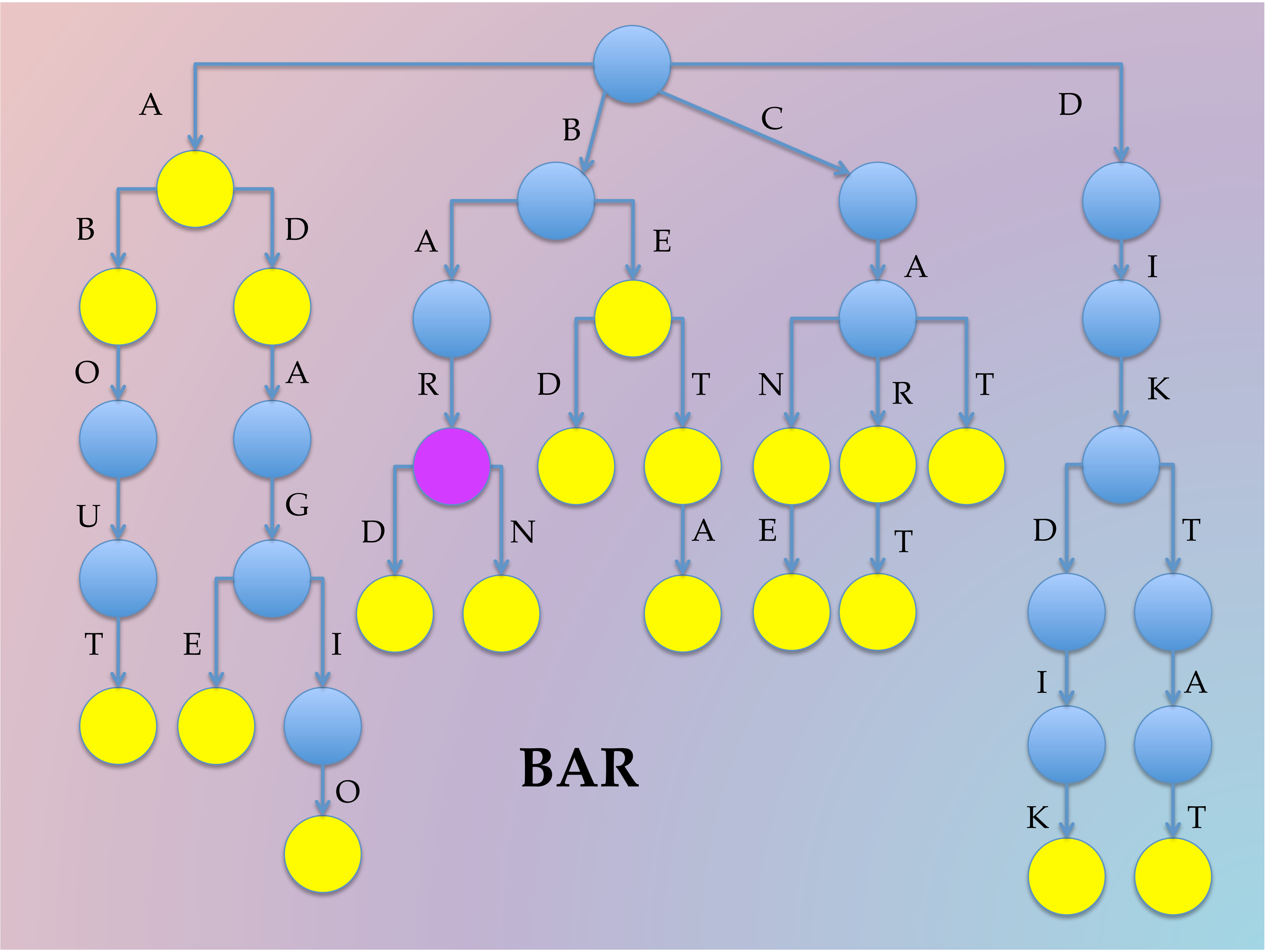

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
R
T
K
U
G
D
N
A
E
T
D
T
T
E
I
I
A
O
K
T
BAR

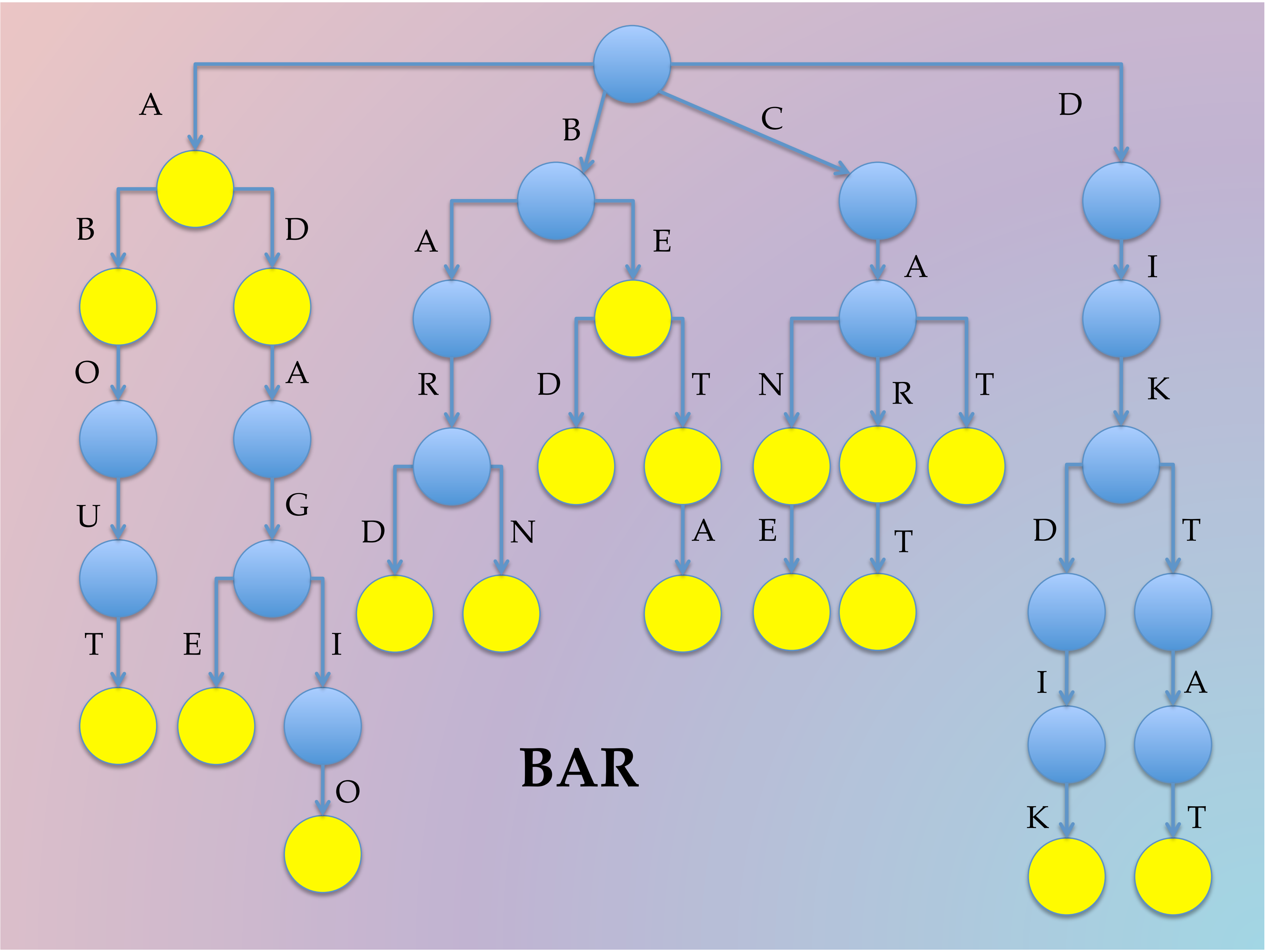

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
R
T
K
U
G
D
N
A
E
T
D
T
T
E
I
I
A
O
K
T
BAR

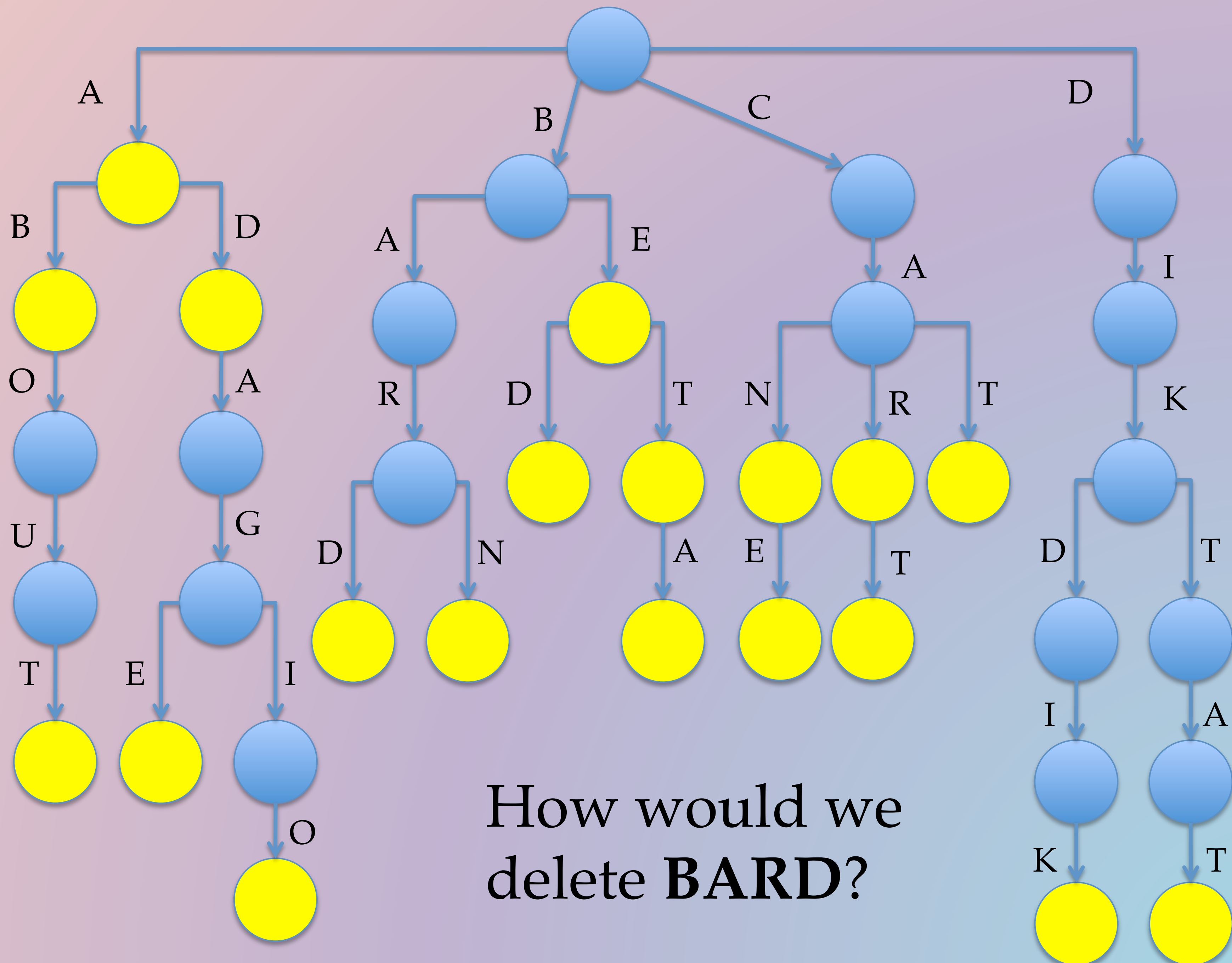

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
R
T
K
U
G
D
N
A
E
T
D
T
T
E
I
I
A
BAR
O
K
T


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

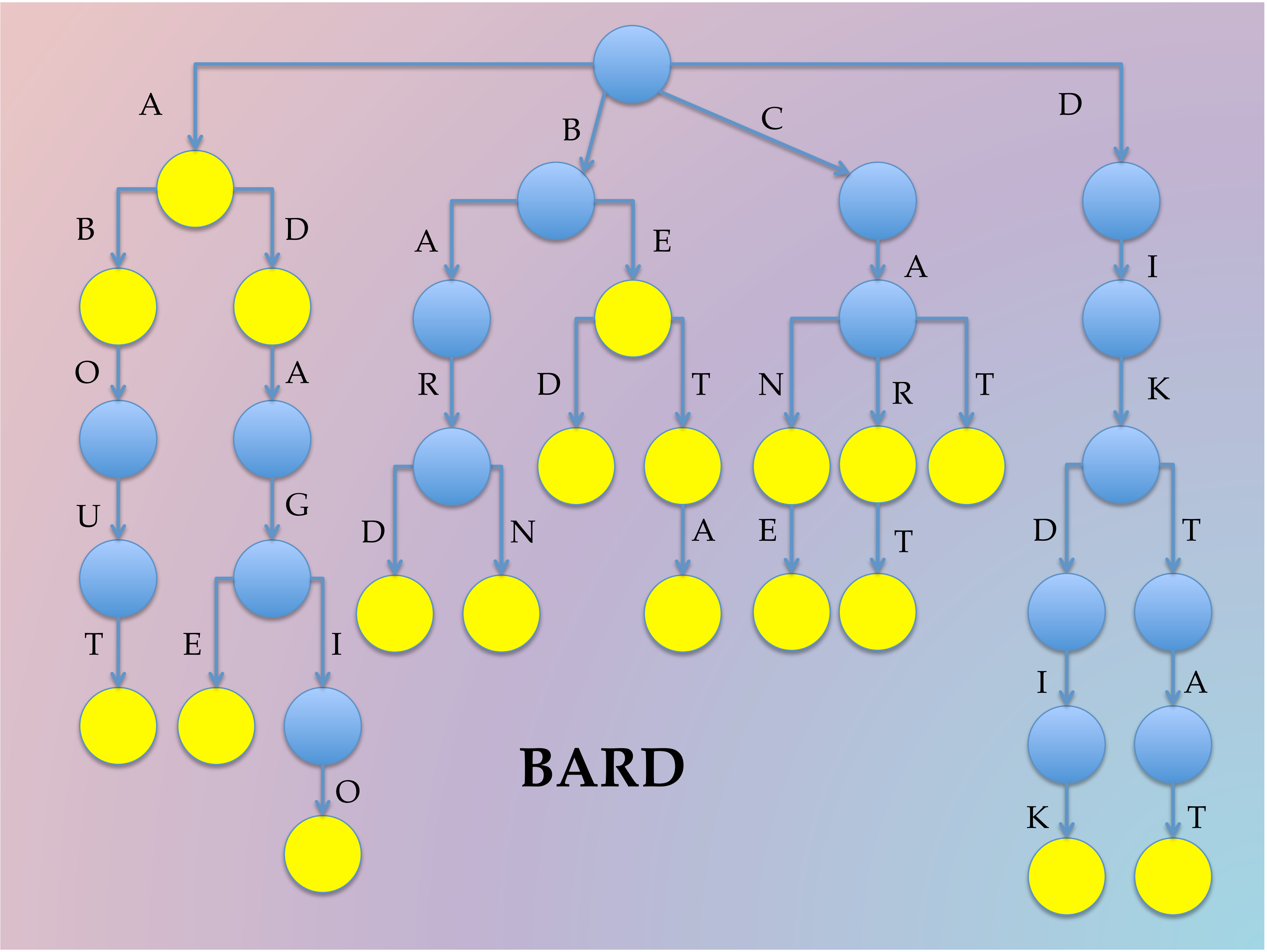

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
R
T
K
U
G
D
N
A
E
T
D
T
T
E
I
I
A
O
K
T
BAR

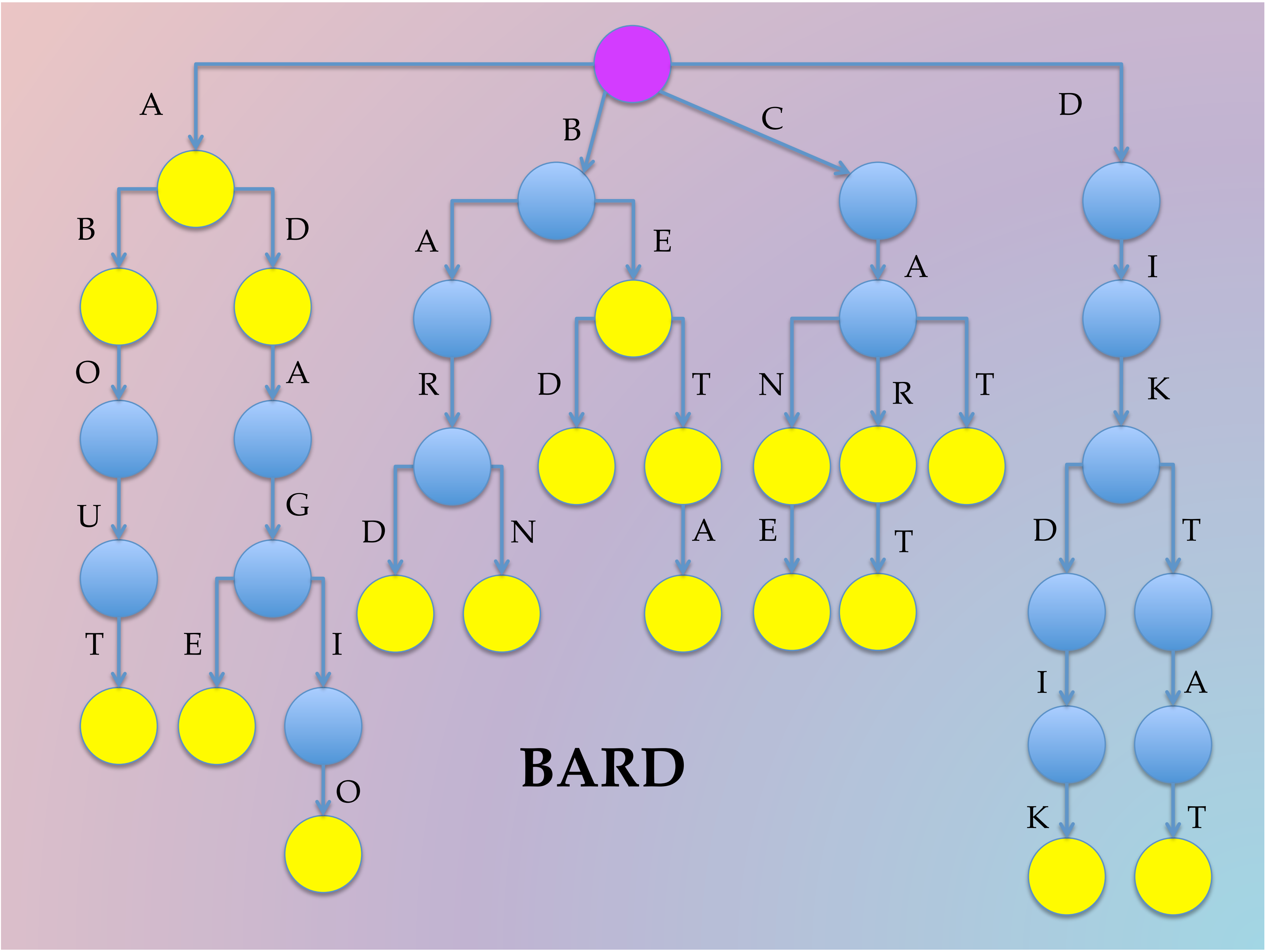

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
R
T
K
U
G
D
N
A
E
T
D
T
T
E
I
I
A
O
BAR
K
T

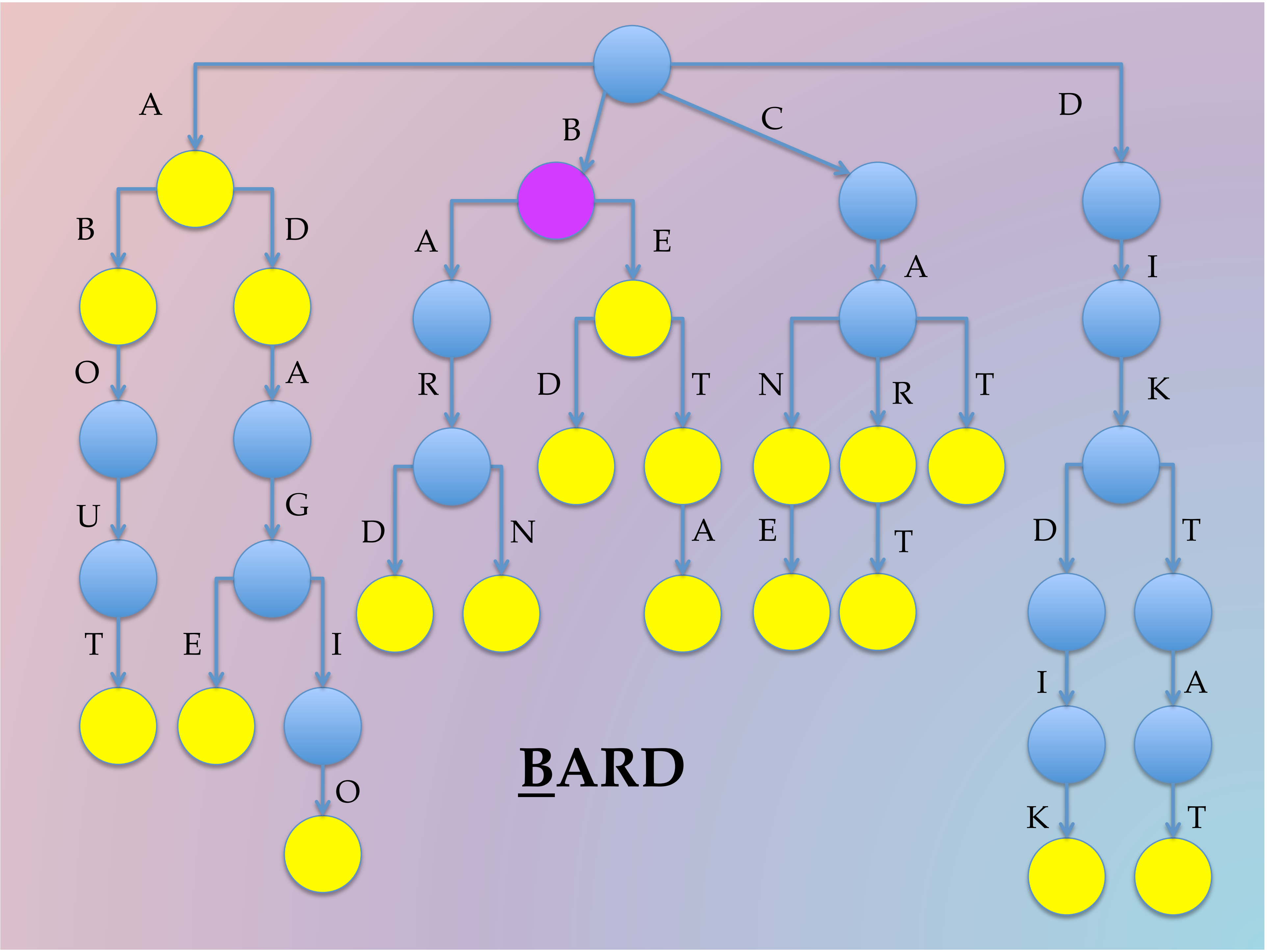

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
R
T
K
U
G
D
N
A
E
T
D
T
T
E
I
I
A
O
K
T
BAR

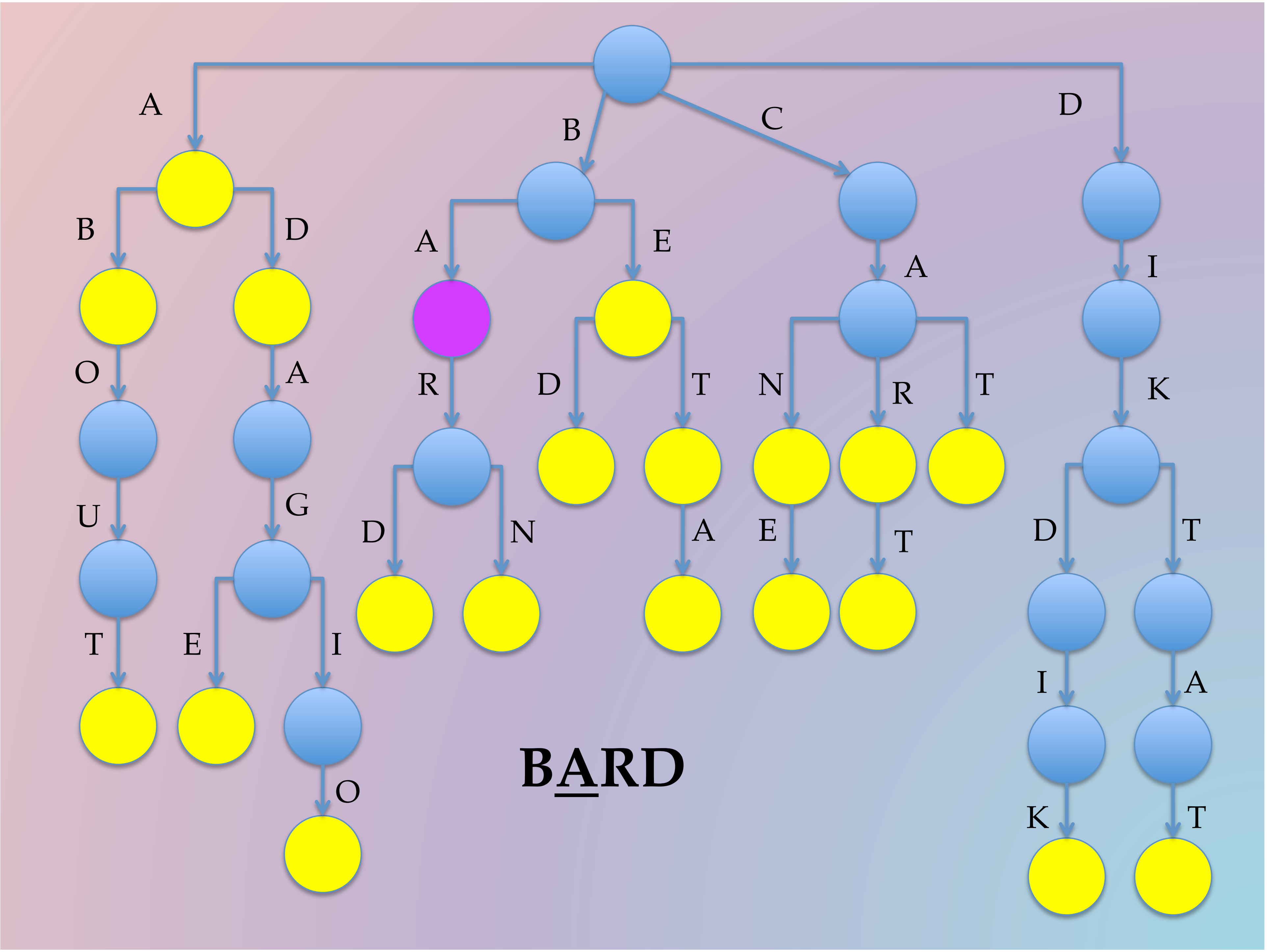


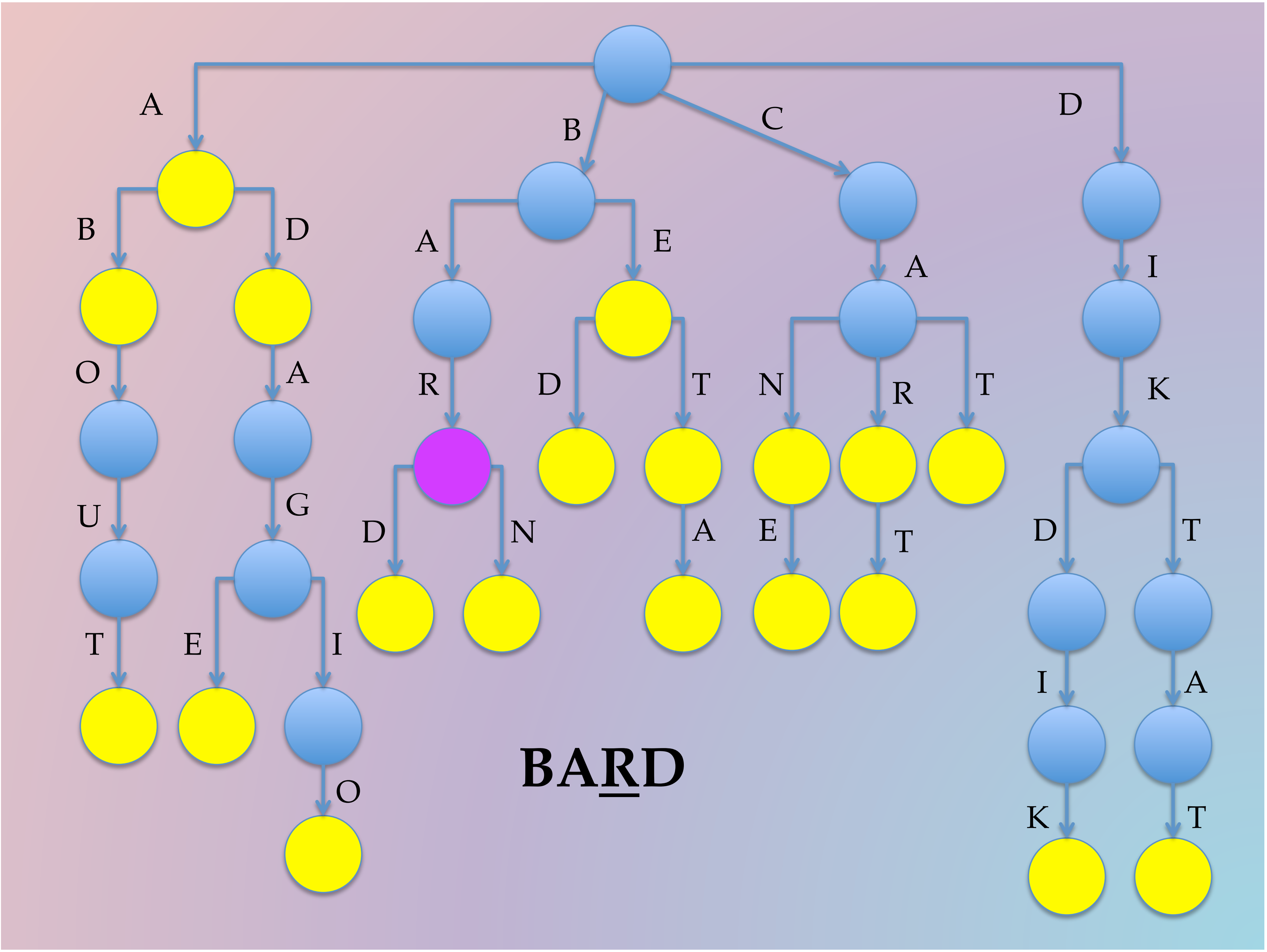
How would we delete **BARD**?

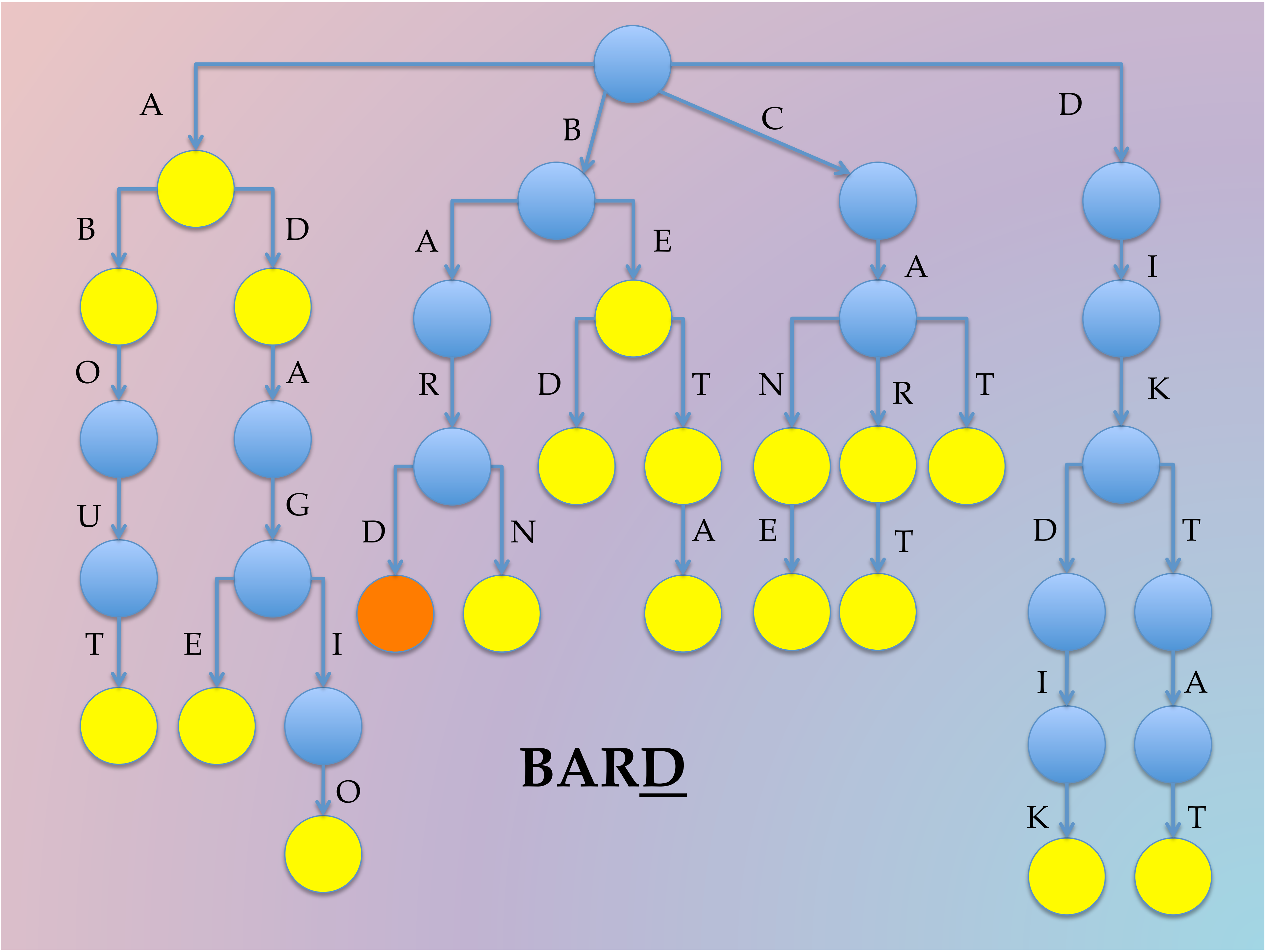

BARD

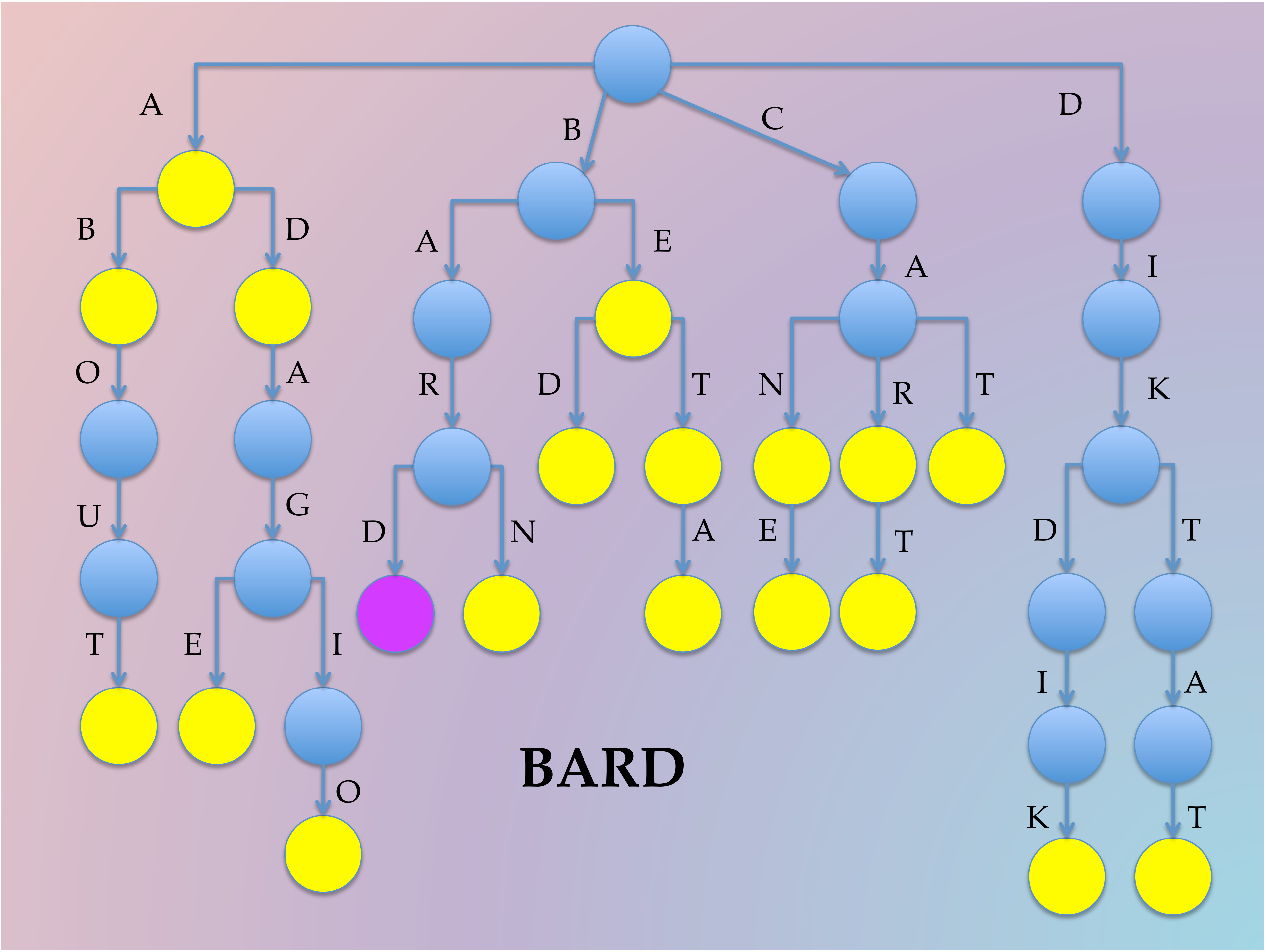

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
R
T
K
U
G
D
N
A
E
T
D
T
T
E
I
I
A
BARD
O
K
T

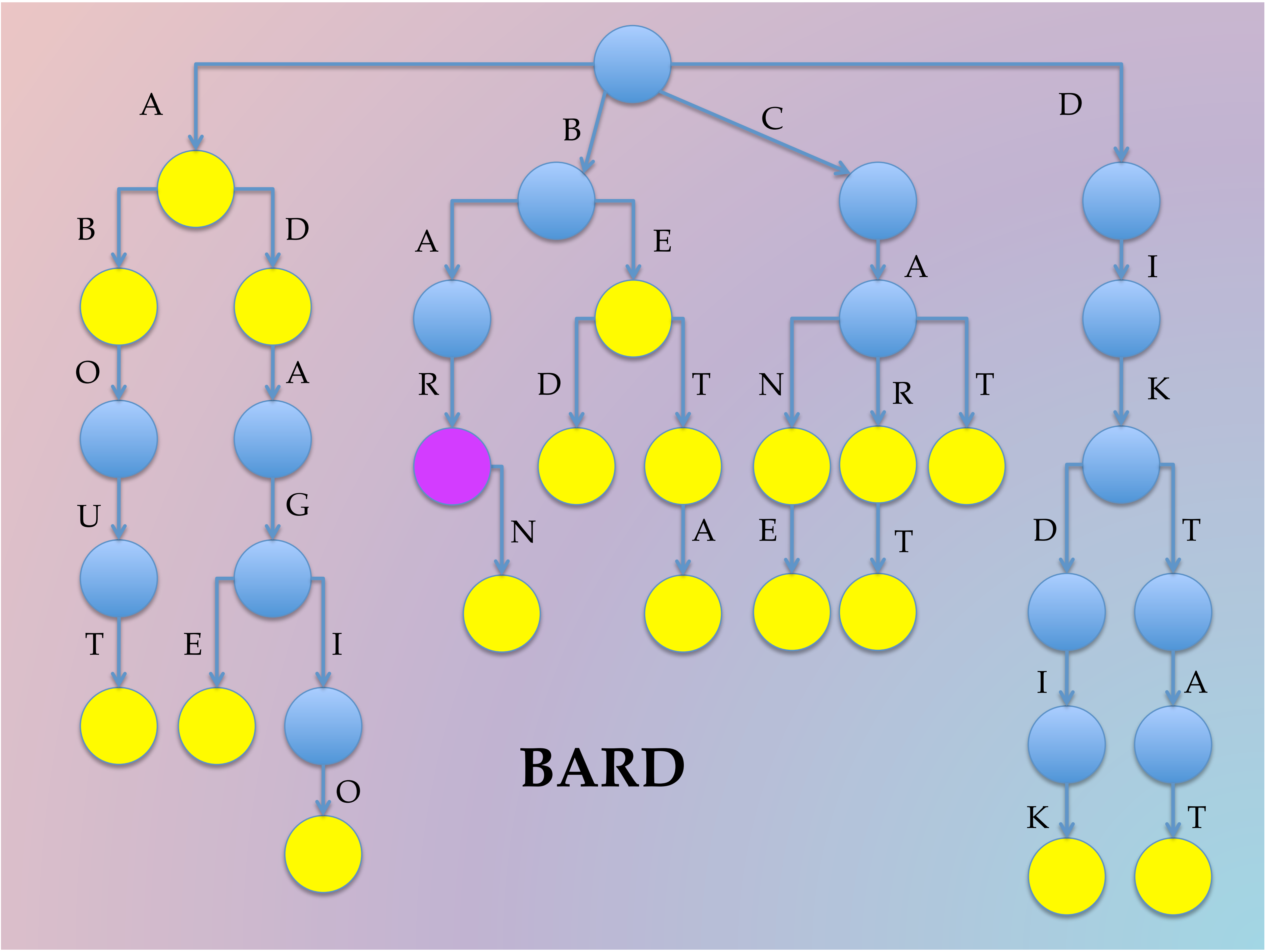

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
R
T
K
U
G
D
N
A
E
T
D
T
T
E
I
I
A
O
BARD
K
T

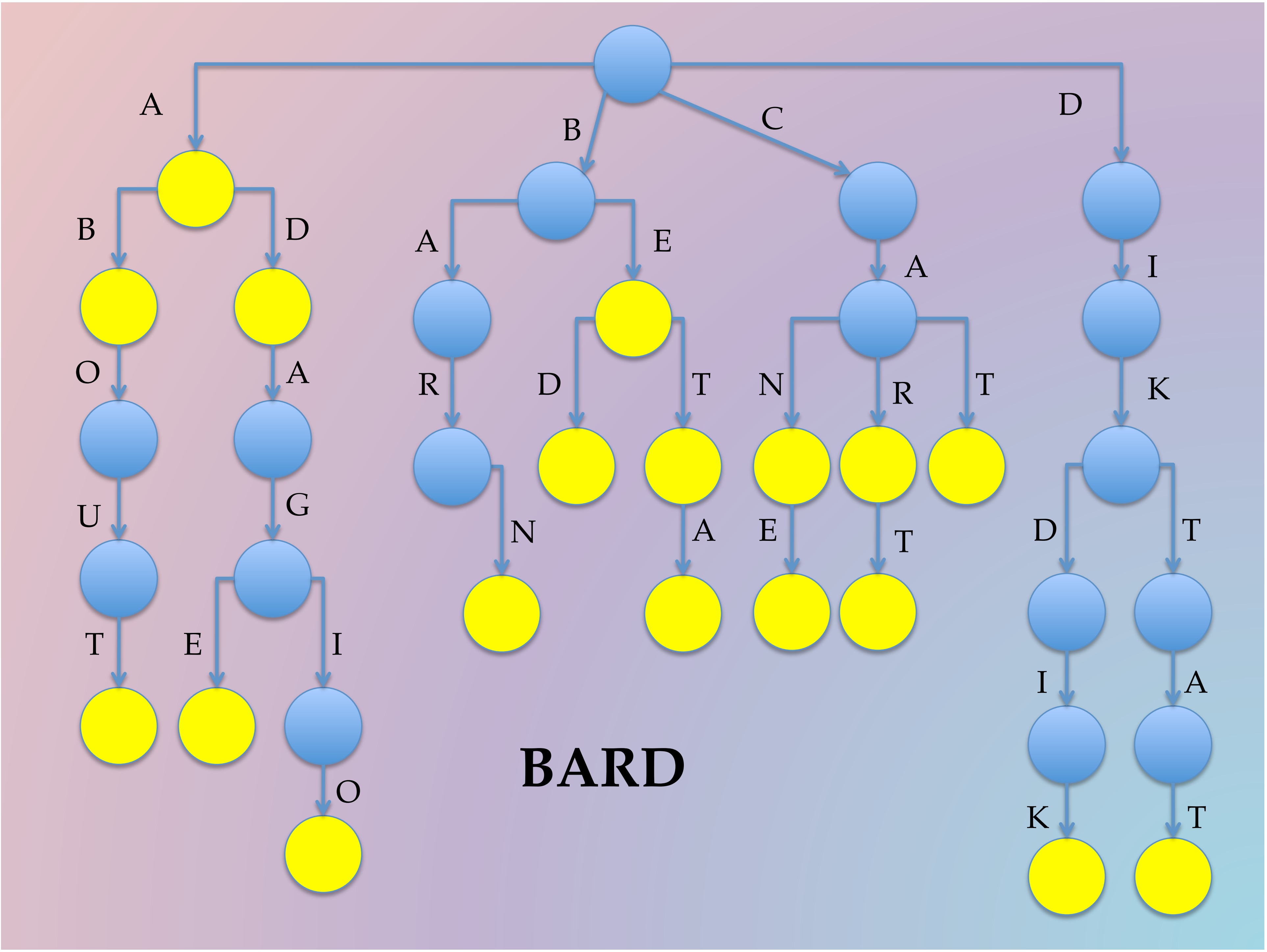

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
R
T
K
U
G
D
N
A
E
T
D
T
T
E
I
I
A
O
K
T
BARD

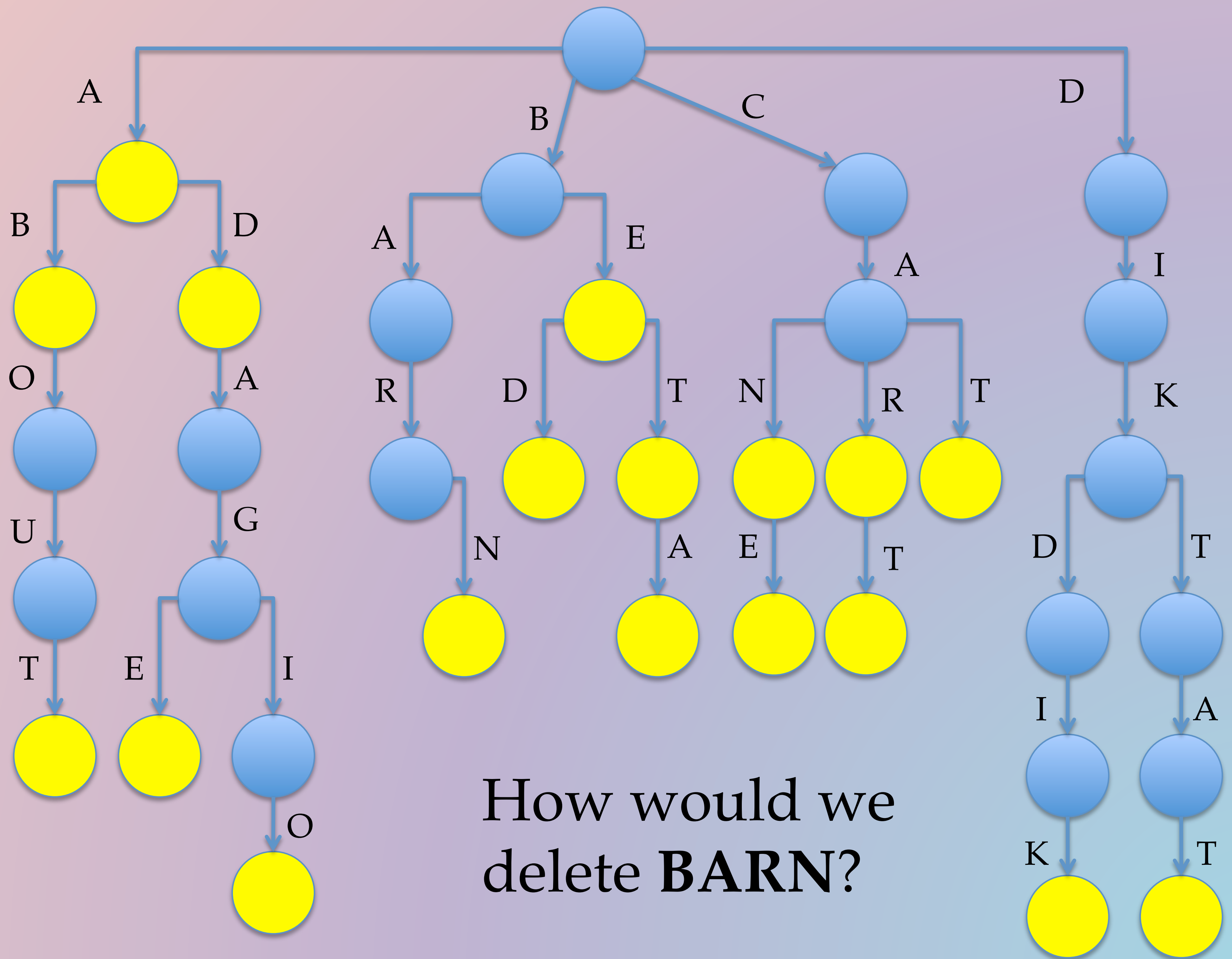

BARD


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

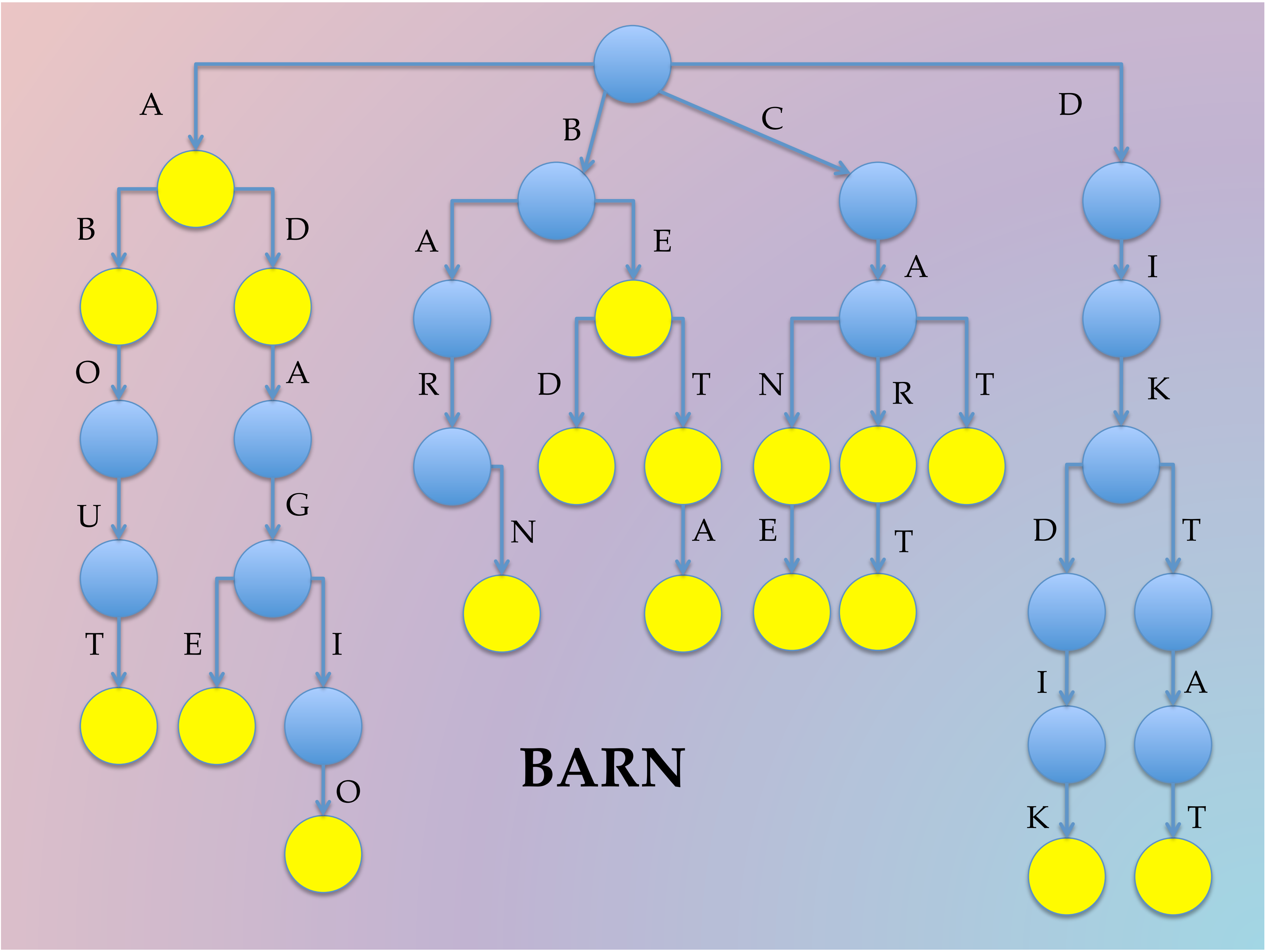

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
R
T
K
U
G
D
N
A
E
T
D
T
T
E
I
I
A
O
K
T
BARD

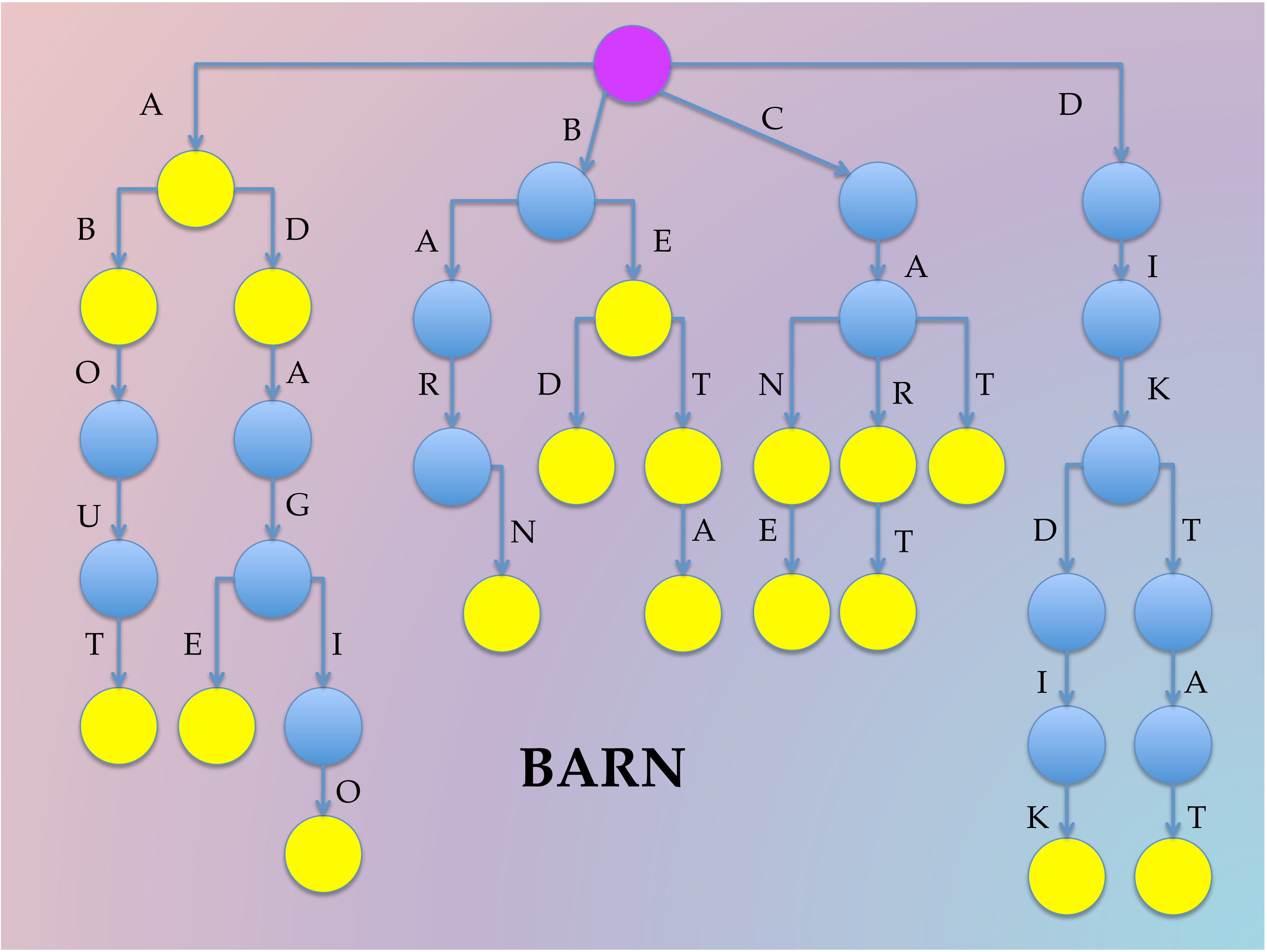

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
R
T
K
U
G
N
A
E
T
D
T
T
E
I
I
A
O
BARD
K
T

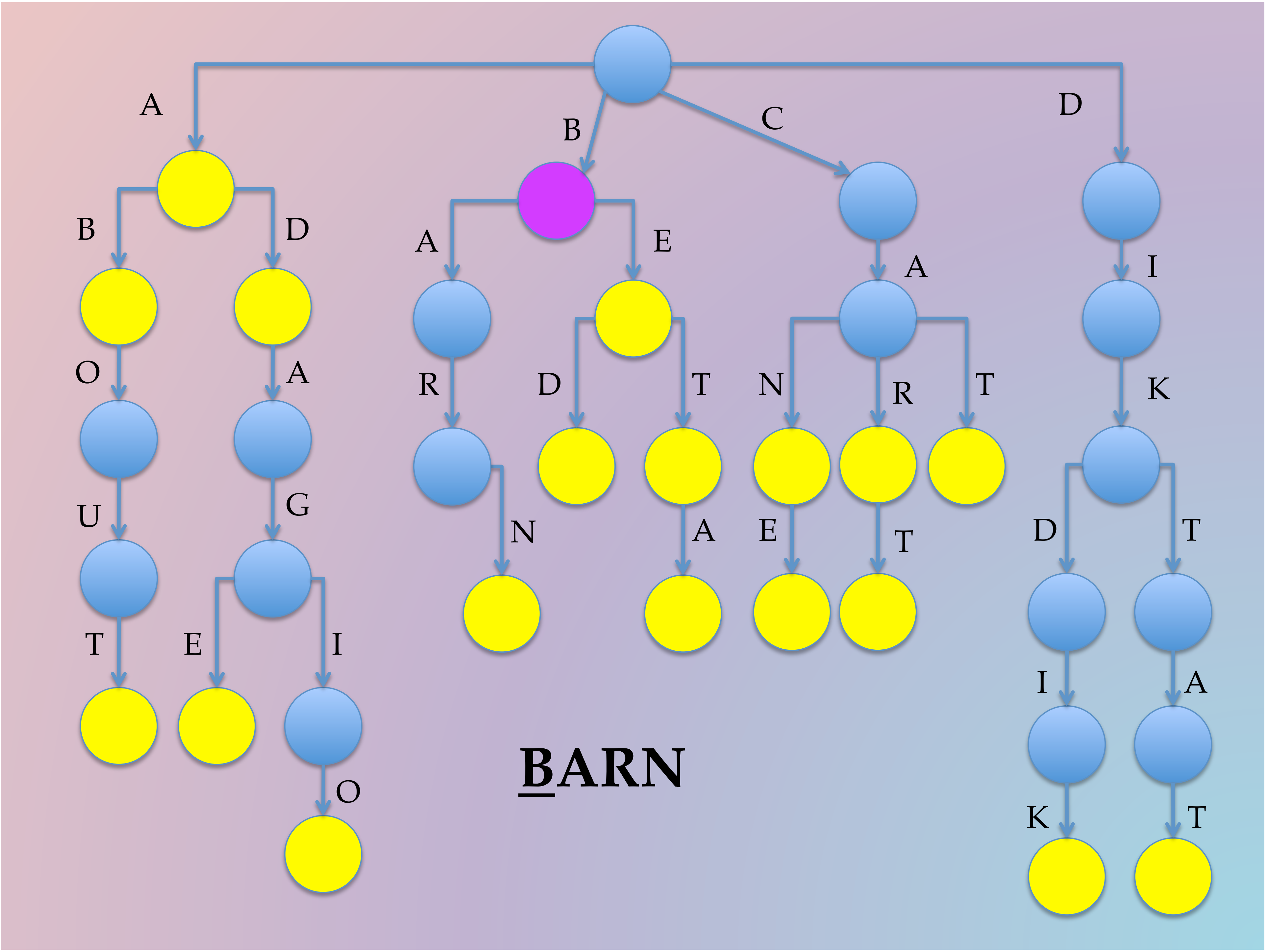

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
R
T
K
U
G
N
A
E
T
D
T
T
E
I
I
A
O
BARD
K
T

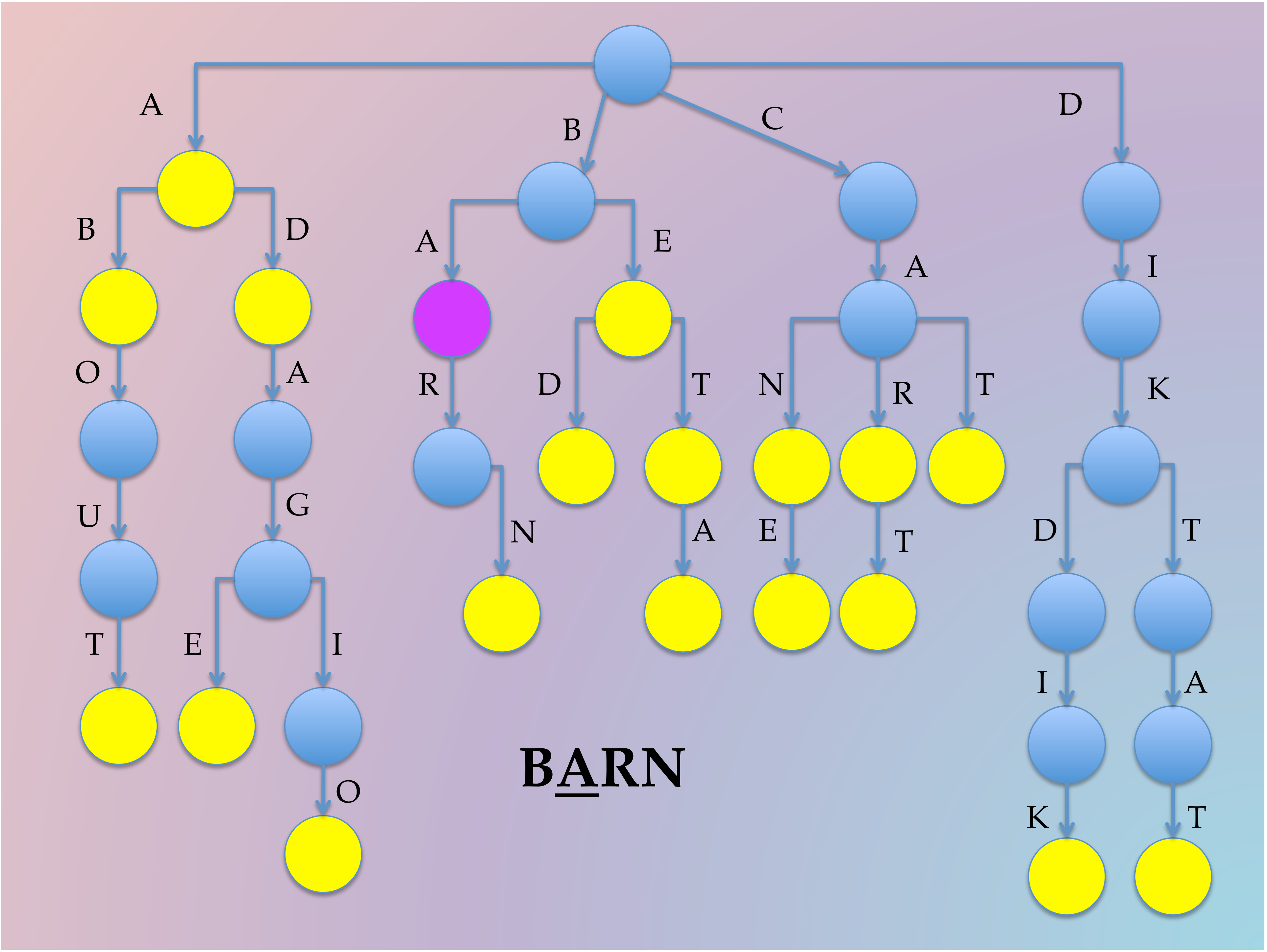


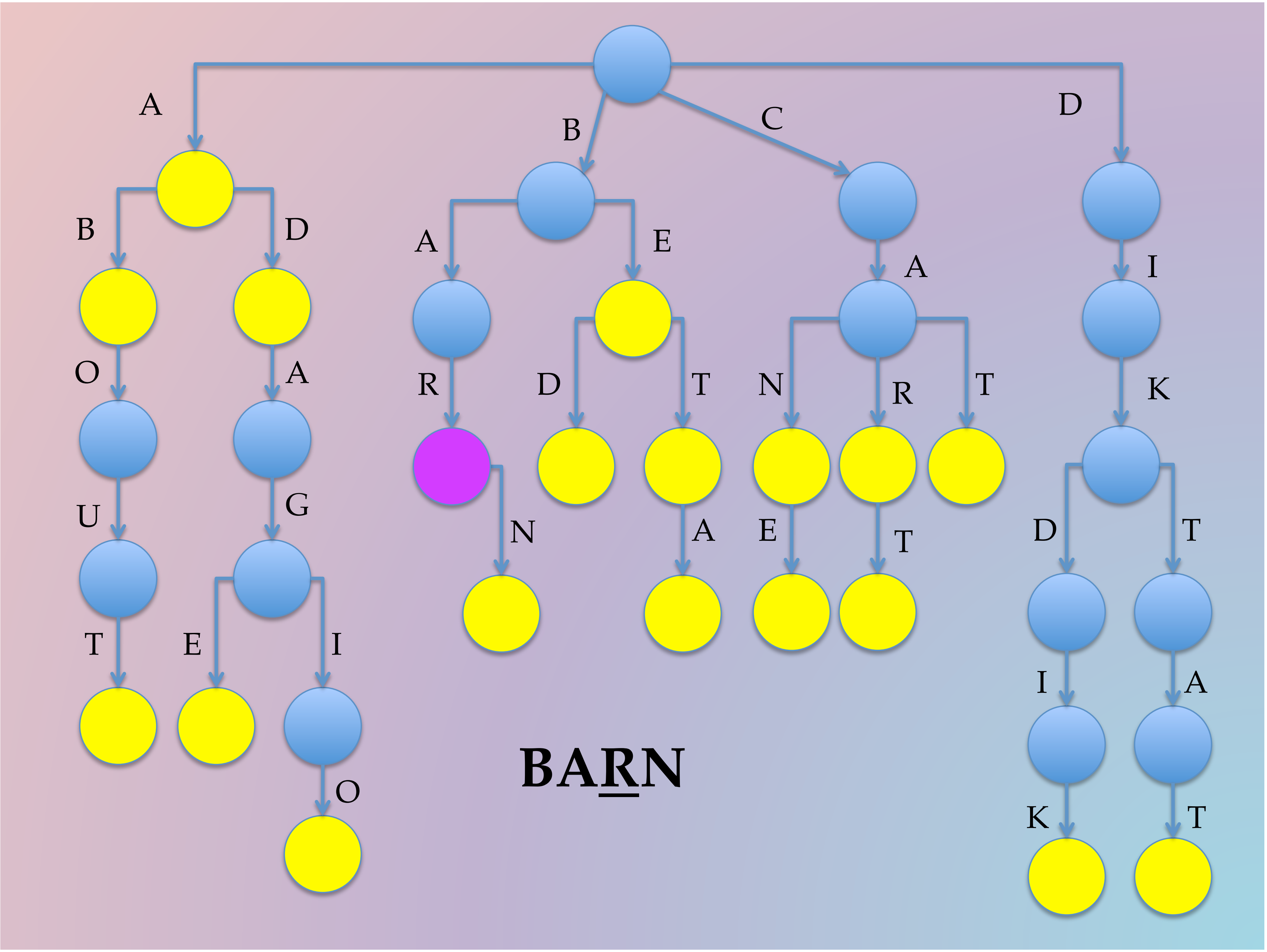
How would we delete **BARN**?

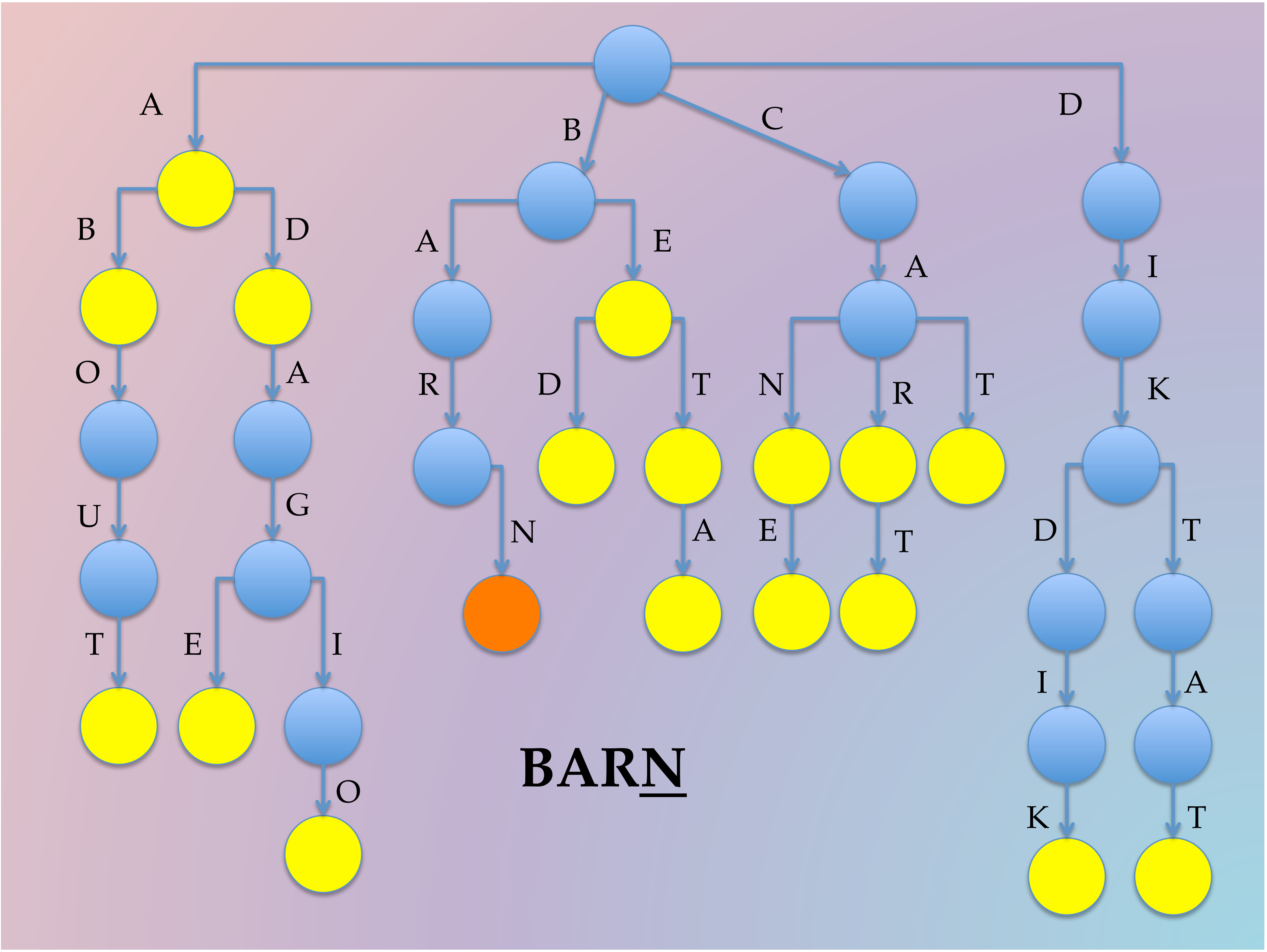

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
R
T
K
U
G
N
A
E
T
D
T
T
E
I
I
A
O
K
T
BARN

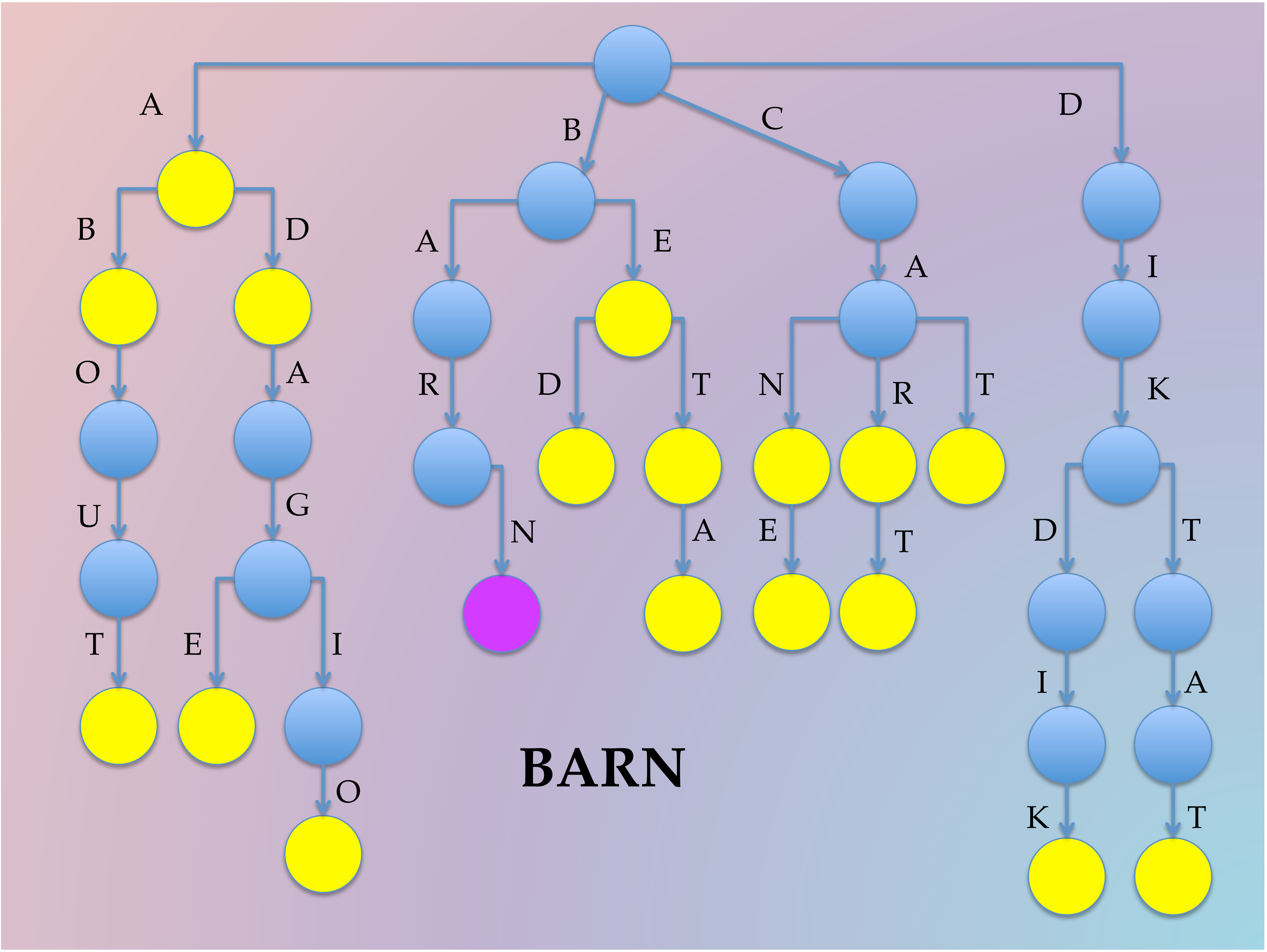

A
B
C
D
B
D
O
A
U
G
T
E
I
O
A
E
R
D
T
N
A
A
N
R
T
E
T
I
K
D
T
I
A
K
T
BARN

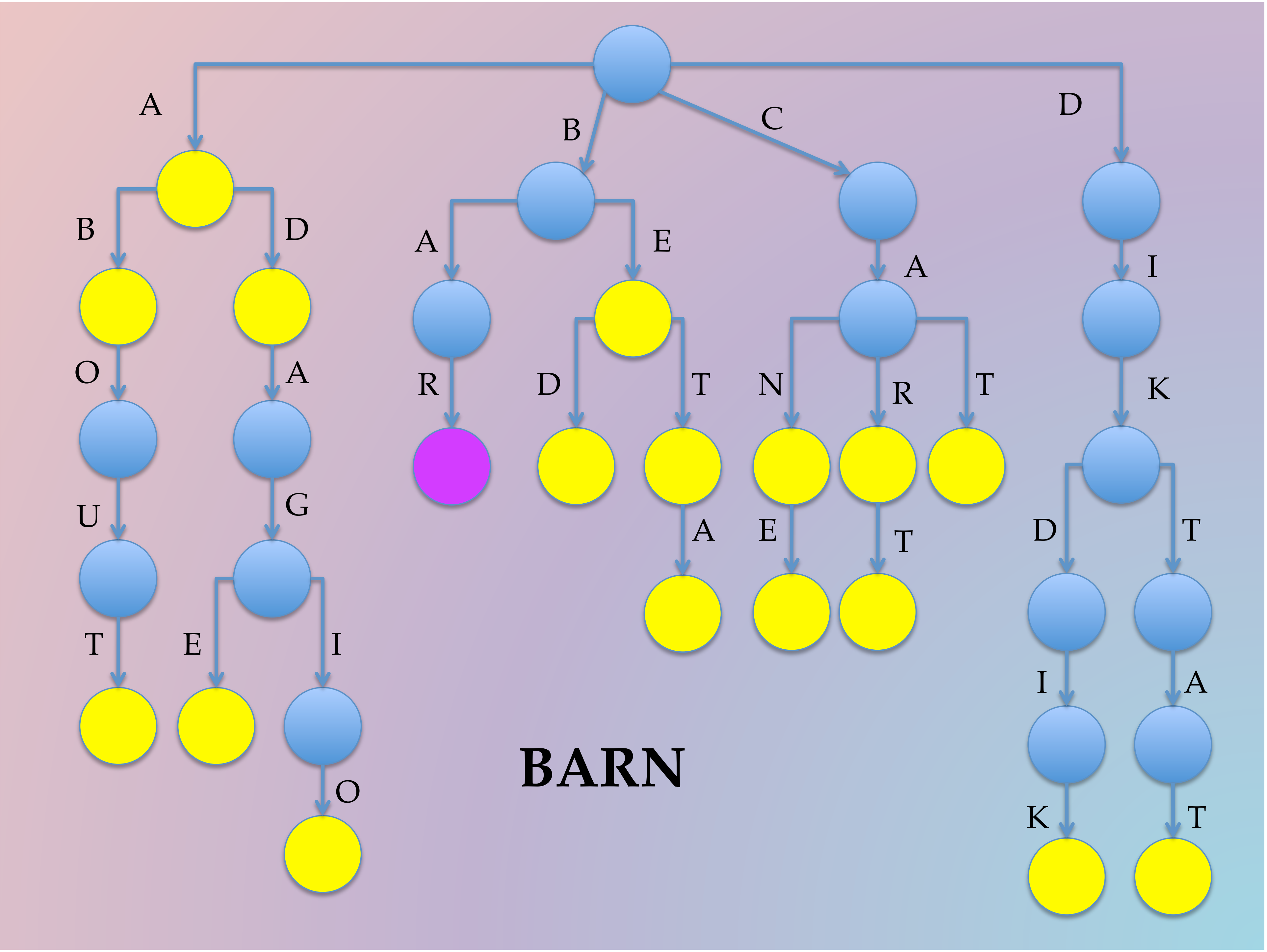

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
R
T
K
U
G
N
A
E
T
D
T
T
E
I
I
A
O
K
T
BARN

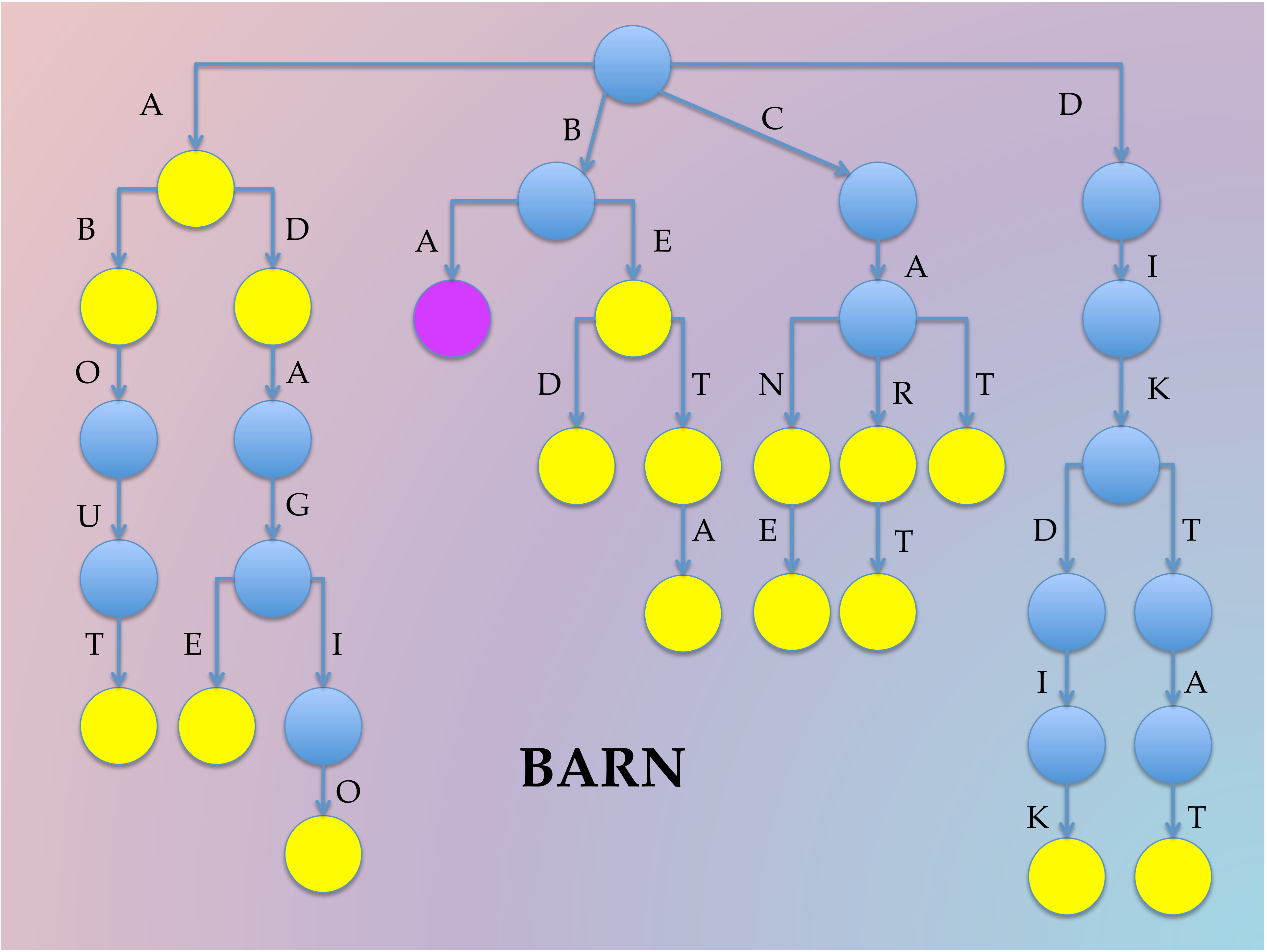

A
B
C
D
B
D
O
A
U
G
T
E
I
O
A
E
R
N
D
T
A
A
N
R
T
E
T
I
K
D
T
I
A
K
T
BARN

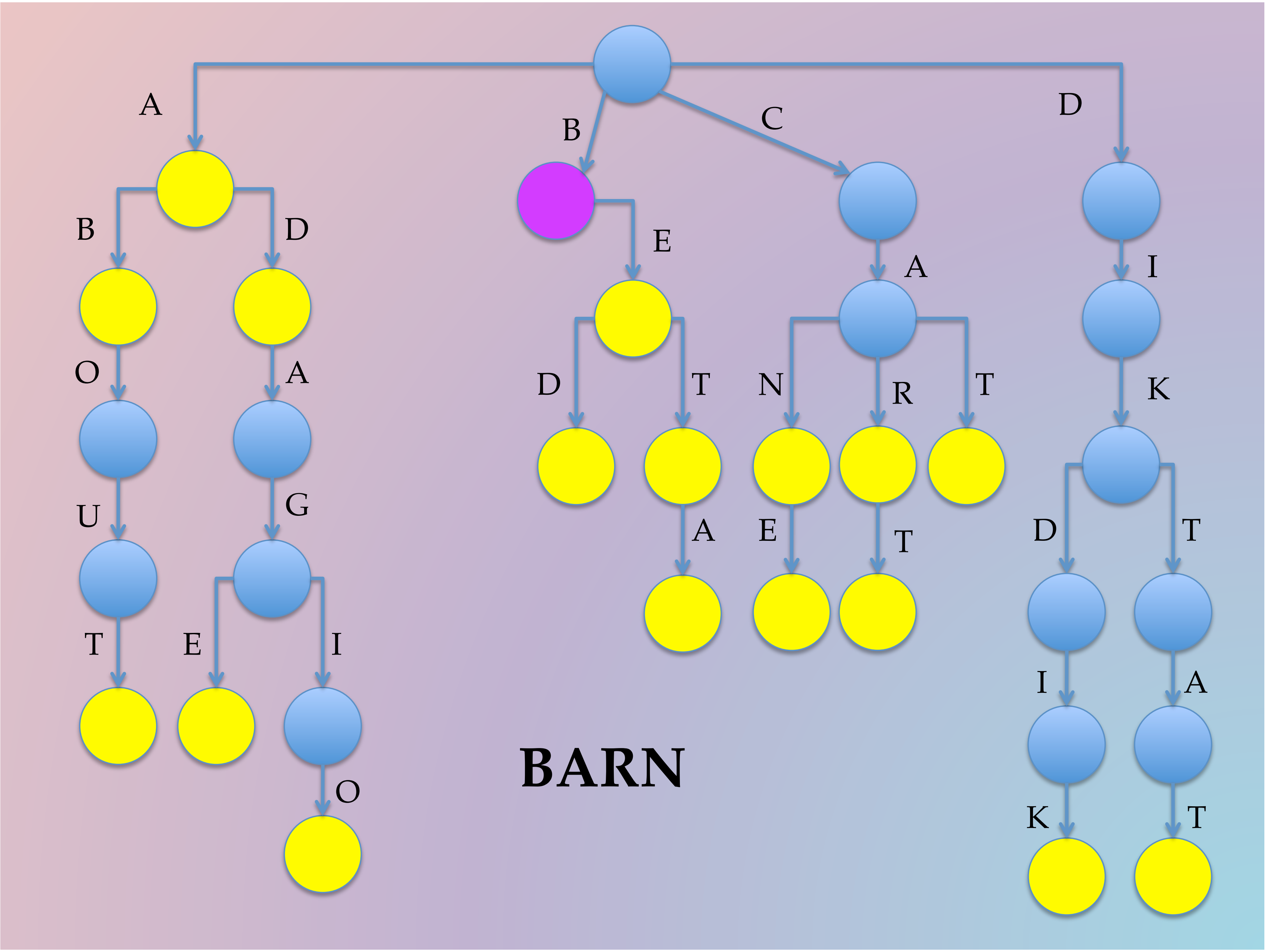

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
R
T
K
U
G
N
A
E
T
D
T
T
E
I
I
A
O
K
T
BARN

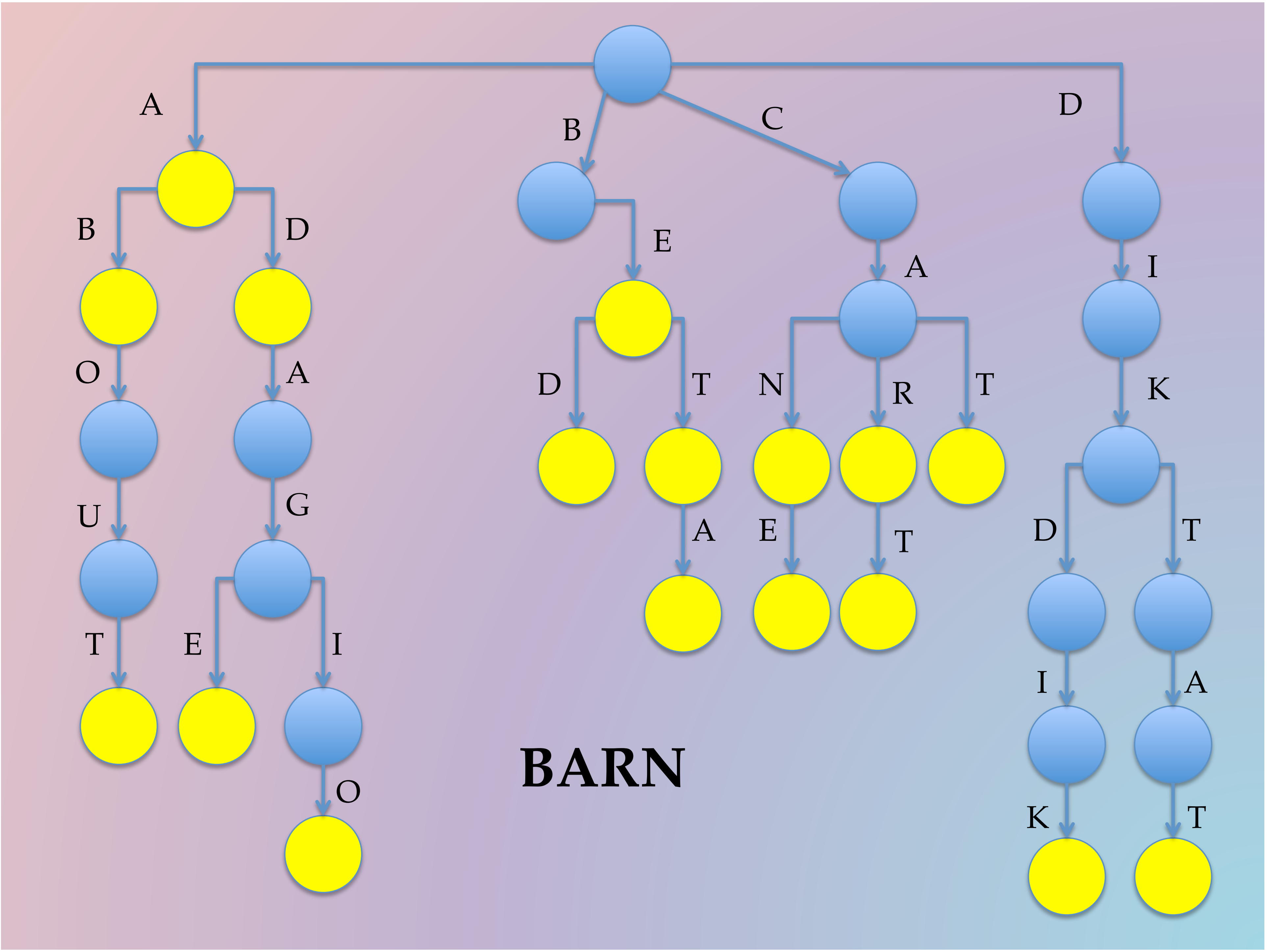

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
R
T
K
U
G
N
A
E
T
D
T
T
E
I
I
A
O
BARN
K
T

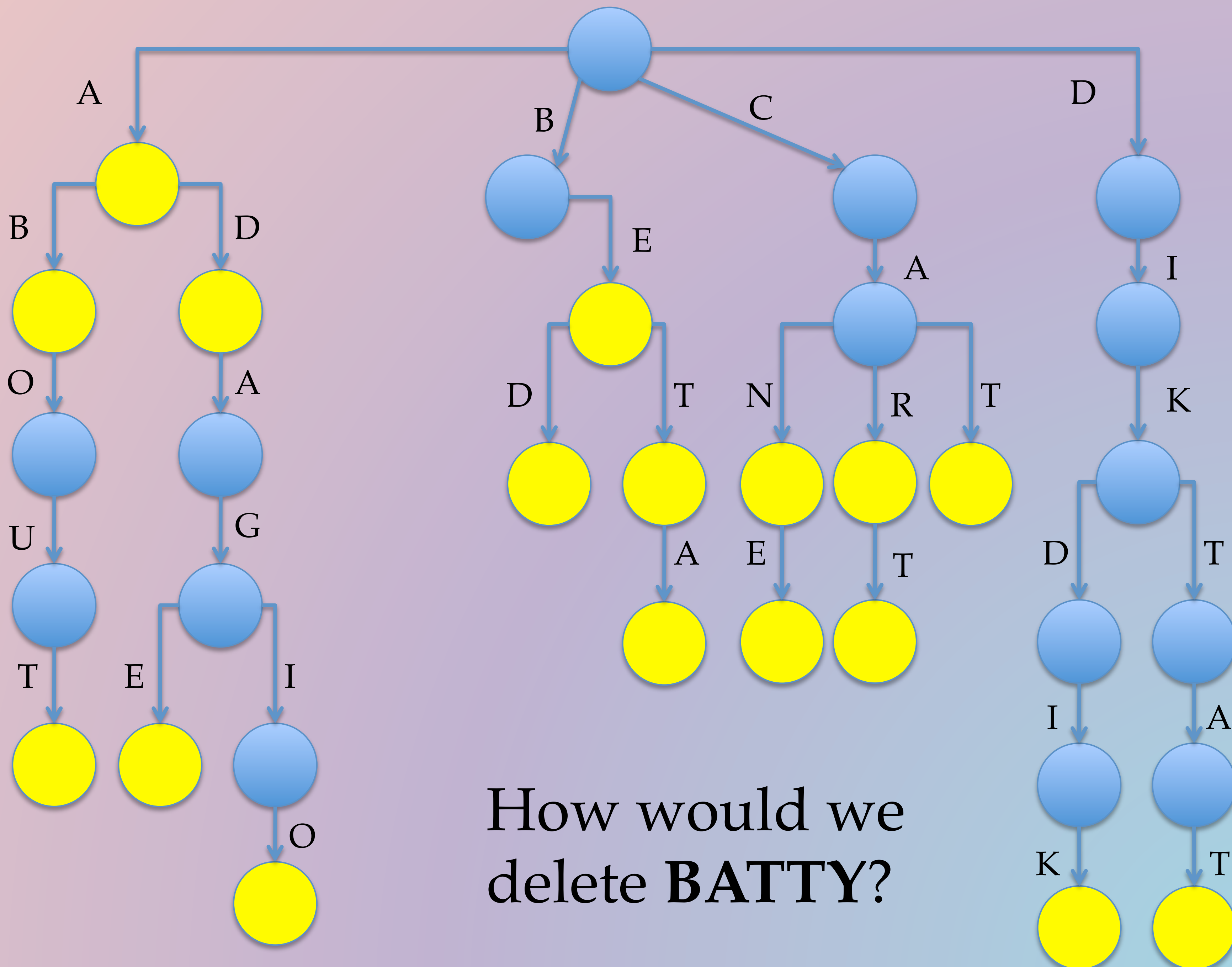

A
B
C
D
B
D
A
E
A
I
O
A
R
D
T
N
R
T
K
U
G
N
A
E
T
D
T
T
E
I
I
A
O
K
T
BARN


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

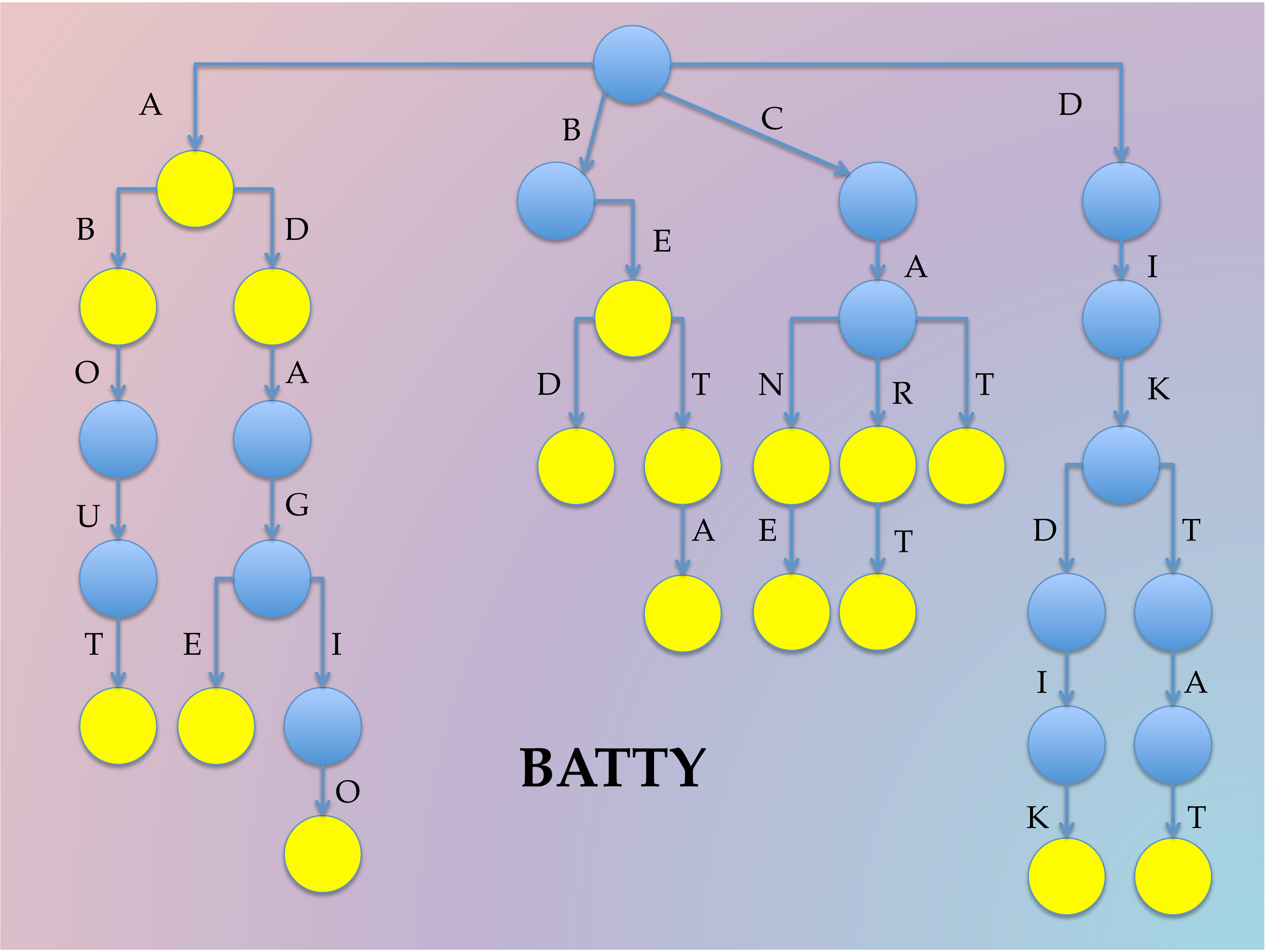

A
B
C
D
B
D
A
E
A
I
O
A
D
T
N
R
T
K
U
G
A
E
T
D
T
T
E
I
I
A
O
K
T
BARN

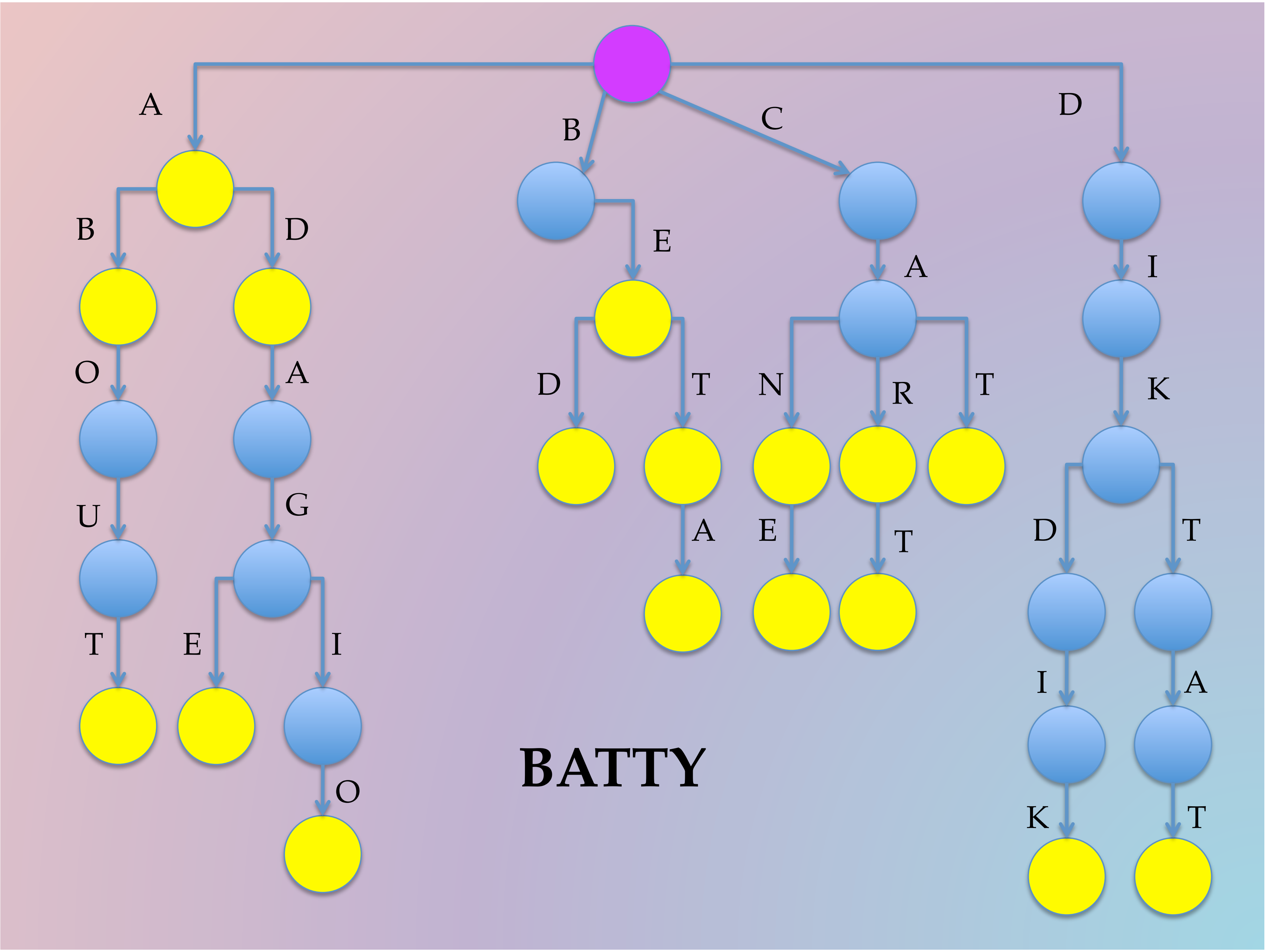

A
B
C
D
B
D
E
A
I
O
A
D
T
N
R
T
K
U
G
A
E
T
D
T
T
E
I
I
A
O
K
T
BARN

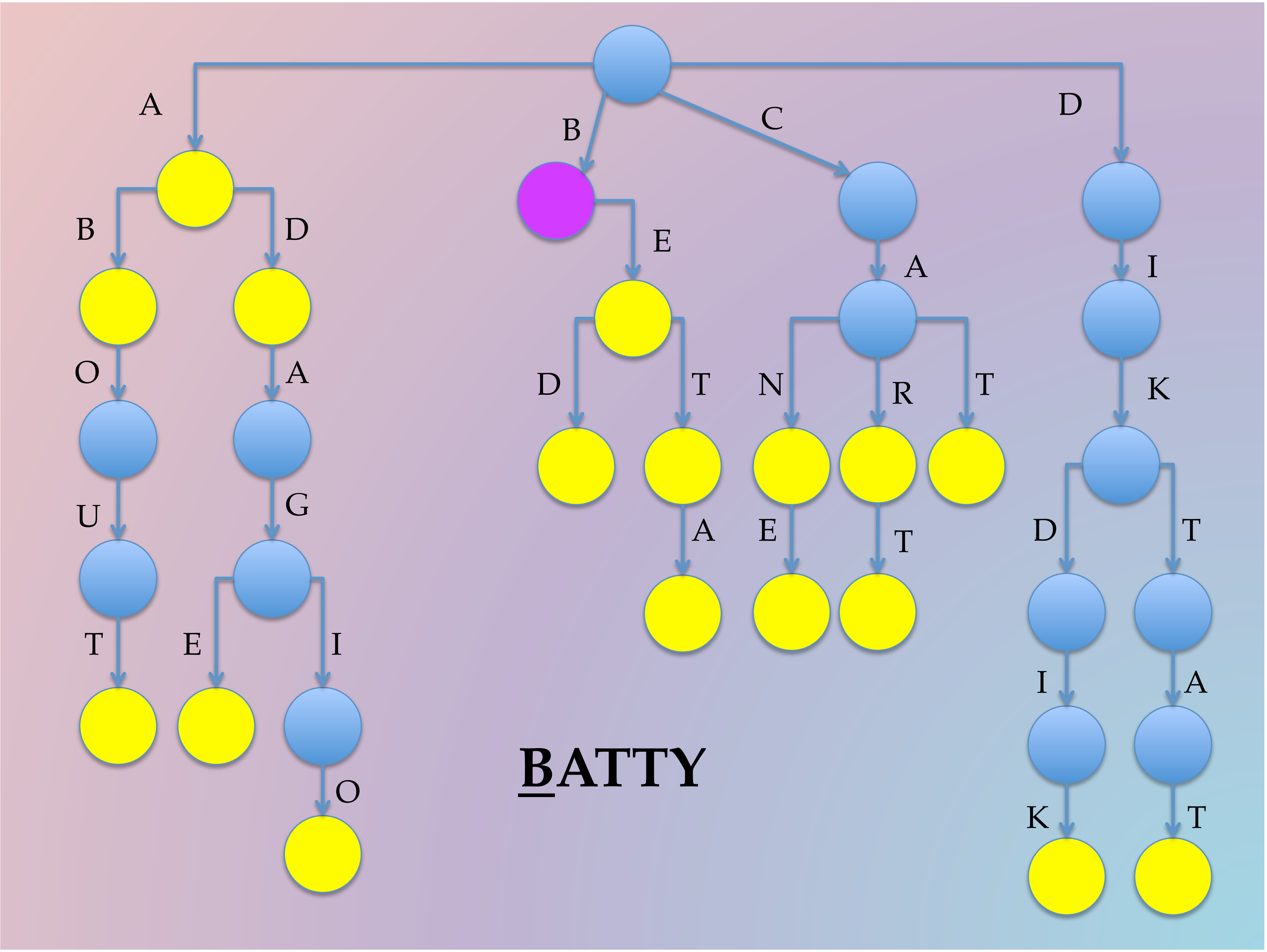

A
B
C
D
B
D
E
A
I
O
A
D
T
N
R
T
K
U
G
A
E
T
D
T
T
E
I
I
A
BARN
O
K
T

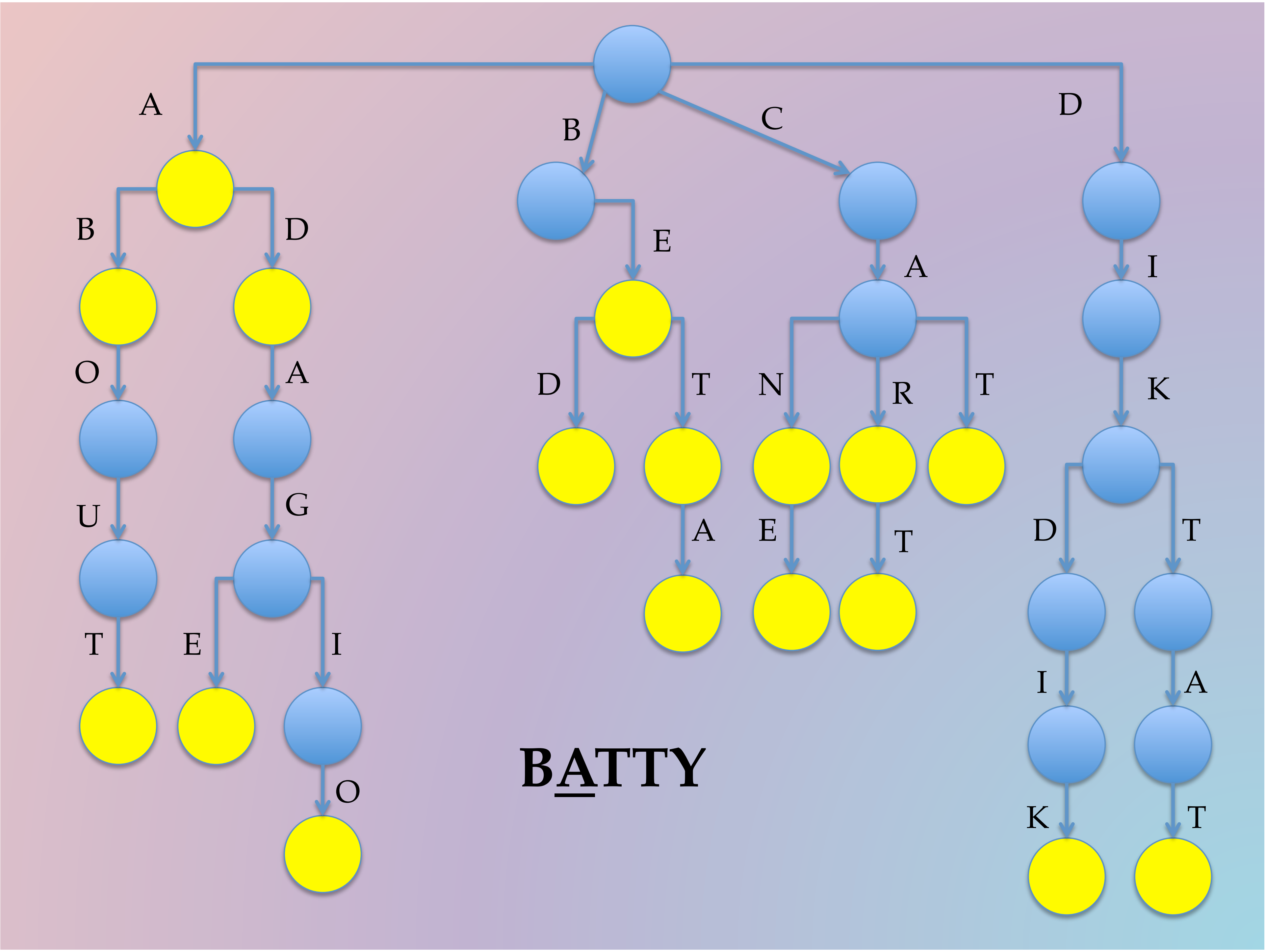


How would we delete **BATTY**?


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


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


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


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

`delete` in Tries

- How long does it take to `delete` a word of length L from a trie with n nodes?
  - A) $O(1)$
  - B) $O(L)$
  - C) $O(\log n)$
  - D) $O(L \log n)$
  - E) Other/none of the above/multiple of the above/unknowable

`delete` in Tries

- How long does it take to `delete` a word of length L from a trie with n nodes?
  - A) $O(1)$
  - **B) $O(L)$**
  - C) $O(\log n)$
  - D) $O(L \log n)$
  - E) Other/none of the above/multiple of the above/unknowable

Space Efficiency of Tries

- While the time efficiency for our trie is wonderful, it takes up a lot of space
- Each node needs to store a map/tree/array and we need a node for every letter
- Can improve the space usage a bit via Patricia tries, e.g. (outside the scope of this course, take 166 for details!)

Tradeoffs

- Is this really "better" than a BST?
- Data structures in particular, and CS/science/engineering/life in general, is all about making tradeoffs
  - Time vs. space efficiency
  - Worst-case vs. average-case
  - Which operations to optimize
  - Theory vs. practice
  - How to best allocate your time or resources
  - Etc.

Stanford Library ADTs Summary

- We've now seen how to implement virtually every Stanford library ADT!
  - `Vector`/`Stack`/`Queue`/`List`: Dynamically allocated array or linked list
  - `Map`/`Set`: BST or hash table
  - `Lexicon`: Trie (or DAWG)
  - `Grid`: Multidimensional array
  - `PriorityQueue`: Heap (in some variant)
  - `Graph`: Adjacency list, adjacency matrix, or incidence matrix

Other Cool Stuff to Research

- Read about it on your own or take CS 161 and 166 (or go to my OH)!
  - Suffix trees/arrays (related to tries)
  - Patricia tries
  - DAWGs and GADDAGs
  - Linear-time sorting algorithms
  - Weight-balanced trees and static optimality
  - Splay trees and dynamic optimality
  - van Emde Boas (vEB) trees and x- and y-fast tries
  - Augmented trees
  - Self-balancing BSTs (we've discussed a few already)
  - B-trees
  - Order-statistic trees
  - Other specialized trees
  - String matching
  - Formal analysis of all these data structures/algorithms
  - Etc.