KNUTH-MORRIS-PRATT AND AHO-CORASICK

24127094 – Trịnh Duy Nhân

24127035 – Võ Đình Cao Minh Hào

24127227 – Phan Thanh Trúc Quân

KNUTH – MORRIS – PRATT (KMP)

DEFINITION

- Knuth – Morris – Pratt (or KMP) algorithm is a algorithm that use to solve string matching problems, which is searching occurrences of “word” within the main “text – string”
- KMP consists of two following steps:

  - Preprocessing

  - Searching/Matching

PROBLEM

- Given a main string and a pattern string, check whether the pattern occurs in the main string?

- For example:
  - The main string: “ABXABABABAA”
  - The word: “ABABAA"
  - YES, the pattern occurs in the main string (from character 6^{th}).

MAIN IDEA OF KMP

- The idea behind KMP is to extend a suffix ending at position i to a suffix ending at position i+1 while still matching the corresponding prefix, which improves efficiency by eliminating expensive string comparisons.

NAIVE ALGORITHM

Main string:

A	B	X	A	B	A	B	A	B	A	A

Word:

A	B	A	B	A	A

NAIVE ALGORITHM

Main string:

A	B	X	A	B	A	B	A	B	A	A

Word:

A	B	A	B	A	A

NAIVE ALGORITHM

Main string:

A	B	X	A	B	A	B	A	B	A	A

Word:

A	B	A	B	A	A

NAIVE ALGORITHM

Main string:

A	B	X	A	B	A	B	A	B	A	A

Word:

A	B	A	B	A	A

NAIVE ALGORITHM

Main string:

A	B	X	A	B	A	B	A	B	A	A

Word:

A	B	A	B	A	A

NAIVE ALGORITHM

Main string:

A	B	X	A	B	A	B	A	B	A	A

Word:

A	B	A	B	A	A

NAIVE ALGORITHM

Main string:

A	B	X	A	B	A	B	A	B	A	A

Word:

A	B	A	B	A	A

NAIVE ALGORITHM

Main string:

A	B	X	A	B	A	B	A	B	A	A

Word:

A	B	A	B	A	A

NAIVE ALGORITHM

Main string:

A	B	X	A	B	A	B	A	B	A	A

Word:

A	B	A	B	A	A

NAIVE ALGORITHM

Main string:

A	B	X	A	B	A	B	A	B	A	A

Word:

A	B	A	B	A	A

NAIVE ALGORITHM

Main string:

A	B	X	A	B	A	B	A	B	A	A

Word:

A	B	A	B	A	A

NAIVE ALGORITHM

Main string:

A	B	X	A	B	A	B	A	B	A	A

Word:

A	B	A	B	A	A

NAIVE ALGORITHM

Main string:

A	B	X	A	B	A	B	A	B	A	A

Word:

A	B	A	B	A	A

NAIVE ALGORITHM

Main string:

A	B	X	A	B	A	B	A	B	A	A

Word:

A	B	A	B	A	A

NAIVE ALGORITHM

Main string:

A	B	X	A	B	A	B	A	B	A	A

Word:

A	B	A	B	A	A

NAIVE ALGORITHM

Main string:

A	B	X	A	B	A	B	A	B	A	A

Word:

A	B	A	B	A	A

NAIVE ALGORITHM

Main string:

A	B	X	A	B	A	B	A	B	A	A

Word:

A	B	A	B	A	A

NAIVE ALGORITHM

Main string:

A	B	X	A	B	A	B	A	B	A	A

Word:

A	B	A	B	A	A

NAIVE ALGORITHM

Main string:

A	B	X	A	B	A	B	A	B	A	A

Word:

A	B	A	B	A	A

NAIVE ALGORITHM

Main string:

A	B	X	A	B	A	B	A	B	A	A

Word:

A	B	A	B	A	A

NAIVE ALGORITHM

Main string:

A	B	X	A	B	A	B	A	B	A	A

Word:

A	B	A	B	A	A

NAIVE ALGORITHM

Main string:

A	B	X	A	B	A	B	A	B	A	A

Word:

A	B	A	B	A	A

NAIVE ALGORITHM

Main string:

A	B	X	A	B	A	B	A	B	A	A

Word:

A	B	A	B	A	A

NAIVE ALGORITHM

Main string:

A	B	X	A	B	A	B	A	B	A	A

Word:

A	B	A	B	A	A

NAIVE ALGORITHM

Main string:

A	B	X	A	B	A	B	A	B	A	A

Word:

A	B	A	B	A	A

NAIVE ALGORITHM

Main string:

A	B	X	A	B	A	B	A	B	A	A

Word:

A	B	A	B	A	A

NAIVE ALGORITHM

Main string:

A	B	X	A	B	A	B	A	B	A	A

Word:

A	B	A	B	A	A

Time Complexity? $O(n \times m)$

Space Complexity? $\Theta(1)$

PREPROCESSING

- Now we need to preprocessing the pattern to ehance the time complexity

→ Prefix function

- The prefix function for this string is defined as an array π of length m, where π[i] is the length of the longest proper prefix of the substring s[0..i] that is also a suffix of this substring.
- A proper prefix of a string is a prefix that is not equal to the string itself

$$\pi[i] = \max_{k=0\dots i} \{k : s[0\dots k-1] = s[i-(k-1)\dots i]\}$$

PREPROCESSING

J I

A	B	A	B	A	A

0					

PREPROCESSING

J I

A	B	A	B	A	A

0					

PREPROCESSING

J I

A	B	A	B	A	A
0	0				

PREPROCESSING

J I

A	B	A	B	A	A
0	0				

PREPROCESSING

J I

A	B	A	B	A	A
0	0				

PREPROCESSING

J		I			
A	B	A	B	A	A
0	0	1			

PREPROCESSING

	J		I		
A	B	A	B	A	A
0	0	1			

PREPROCESSING

	J		I		
A	B	A	B	A	A
0	0	1			

PREPROCESSING

	J		I		
A	B	A	B	A	A
0	0	1	2		

PREPROCESSING

		J		I	
A	B	A	B	A	A

0	0	1	2		

PREPROCESSING

		J		I	
A	B	A	B	A	A
0	0	1	2		

PREPROCESSING

J I

A	B	A	B	A	A
0	0	1	2	3	

PREPROCESSING

			J		I
A	B	A	B	A	A
0	0	1	2	3	

PREPROCESSING

			J		I
A	B	A	B	A	A
0	0	1	2	3	

PREPROCESSING

			J		I
A	B	A	B	A	A
0	0	1	2	3	

PREPROCESSING

			J		I
A	B	A	B	A	A
0	0	1	2	3	

PREPROCESSING

	J				I
A	B	A	B	A	A
0	0	1	2	3	

PREPROCESSING

j i

A	B	A	B	A	A
0	0	1	2	3	

PREPROCESSING

	J				I
A	B	A	B	A	A
0	0	1	2	3	

PREPROCESSING

	J				I
A	B	A	B	A	A
0	0	1	2	3	

PREPROCESSING

J I

A	B	A	B	A	A
0	0	1	2	3	

PREPROCESSING

J I

A	B	A	B	A	A
0	0	1	2	3	

PREPROCESSING

J					I
A	B	A	B	A	A
0	0	1	2	3	1

PREPROCESSING

A	B	A	B	A	A

0	0	1	2	3	1

Time Complexity? $\Theta(m)$

Space Complexity? $\Theta(m)$

STRING MATCHING

- After produce array π, we are ready to matching the main string now
- Array π helps us not to recheck all pattern again
- Array π helps us skip the overlap strings, hence, it will run linear time in the main string instead of go back to the next position of the current substring

→ Reduce time complexity

STRING MATCHING

I

A	B	X	A	B	A	B	A	B	A	A

J

A	B	A	B	A	A

0	0	1	2	3	1

STRING MATCHING

		I								
A	B	X	A	B	A	B	A	B	A	A

		J			
A	B	A	B	A	A

0	0	1	2	3	1

STRING MATCHING

I

A	B	X	A	B	A	B	A	B	A	A

J

A	B	A	B	A	A

0	0	1	2	3	1

STRING MATCHING

I

A	B	X	A	B	A	B	A	B	A	A

J

A	B	A	B	A	A

0	0	1	2	3	1

STRING MATCHING

I

A	B	X	A	B	A	B	A	B	A	A

J

A	B	A	B	A	A

0	0	1	2	3	1

STRING MATCHING

I

A	B	X	A	B	A	B	A	B	A	A

J

A	B	A	B	A	A

0	0	1	2	3	1

STRING MATCHING

			I							
A	B	X	A	B	A	B	A	B	A	A

J					
A	B	A	B	A	A

0	0	1	2	3	1

STRING MATCHING

								I		
A	B	X	A	B	A	B	A	B	A	A

					J
A	B	A	B	A	A

0	0	1	2	3	1

STRING MATCHING

I

A	B	X	A	B	A	B	A	B	A	A

J

A	B	A	B	A	A

0	0	1	2	3	1

STRING MATCHING

I

A	B	X	A	B	A	B	A	B	A	A

J

A	B	A	B	A	A

0	0	1	2	3	1

STRING MATCHING

I

A	B	X	A	B	A	B	A	B	A	A

J

A	B	A	B	A	A

0	0	1	2	3	1

STRING MATCHING

I

A	B	X	A	B	A	B	A	B	A	A

J

A	B	A	B	A	A

0	0	1	2	3	1

STRING MATCHING

I

A	B	X	A	B	A	B	A	B	A	A

J

A	B	A	B	A	A

0	0	1	2	3	1

STRING MATCHING

A	B	X	A	B	A	B	A	B	A	A

A	B	A	B	A	A

0	0	1	2	3	1

Time Complexity? $O(n + m)$

Space Complexity? $\Theta(m)$

AHO-CORASICK

DEFINITION

- The Aho-Corasick algorithm is a dictionary-matching algorithm that can find all the occurrences of all sets of patterns within a given text in linear time.
- Aho – Corasick consists of two following steps:

  - Preprocessing

  - Searching/Matching

PROBLEM

- Give a text and a set of multiple patterns, check how many pattern strings occur in the text?

- For example:
  - The Text: “DIDUDUADI”
  - The Pattern strings: “DI”, “DIDU”, “DIDI”, “DU”, “DUDUA”, “DUADI”
  - There are 5 pattern occur in the text: “DI”, “DIDU”, “DUDUA”, “DUADI”, “DU”.

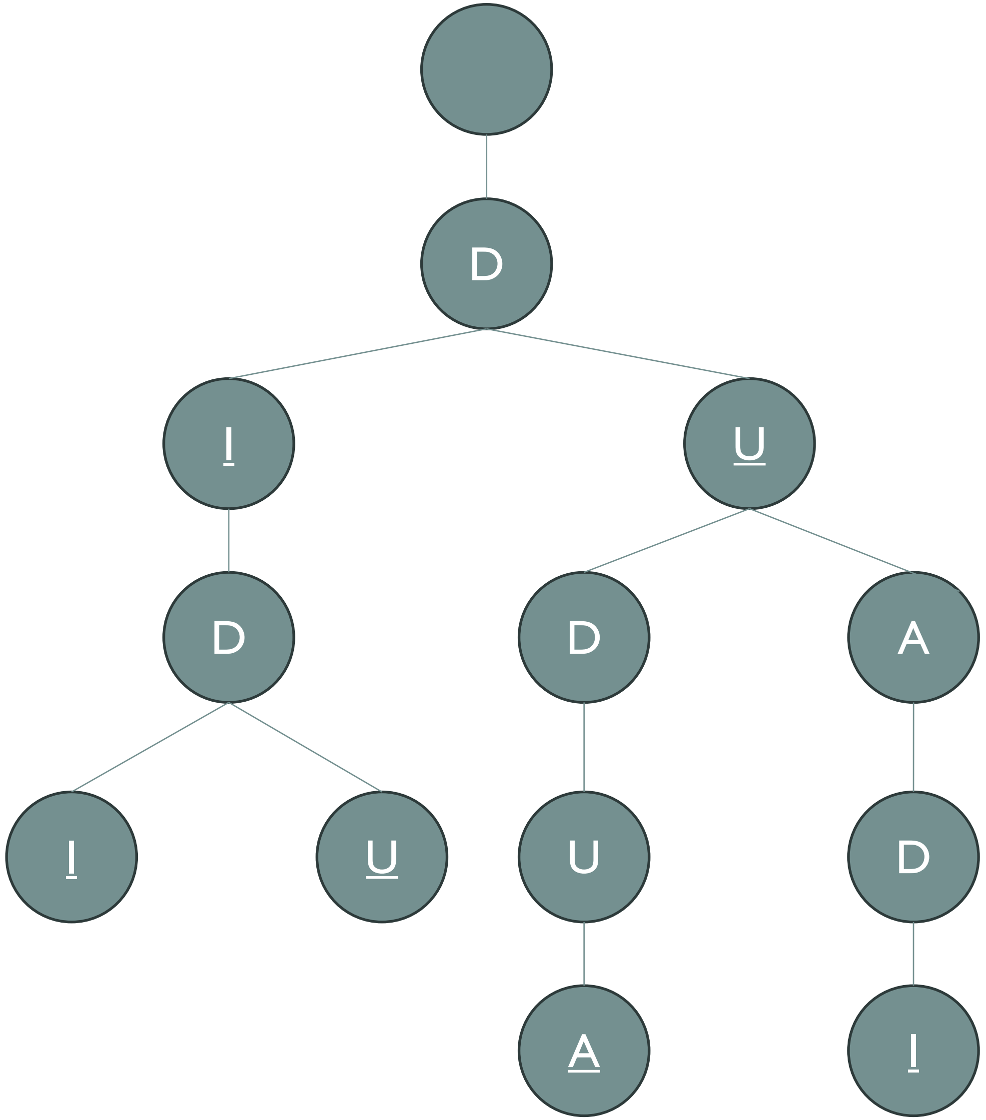
MAIN IDEA OF AHO-CORASICK

- Based on KMP algorithm's main idea, Aho-Corasick links the suffix ending at position i of the pattern to the prefix of another pattern string having the same as the suffix.
- So that, Aho-Corasick constructs trie structure to contain multiple pattern strings and each node of trie has a link pointing to the prefix that is similar to the suffix of current substring. If there aren't any prefix similar to the suffix, the link will point to the root of trie.
- The searching or matching process uses the link to link to the prefix when the mismatching occurs, so that we don't have to check the same substring many times.

ABOUT TRIE

It's a tree data structure used for storage multi string. Every string having the same prefix has the same parents. Each node contains a character of a string

For example, this is a trie structure storages the pattern strings in the first example



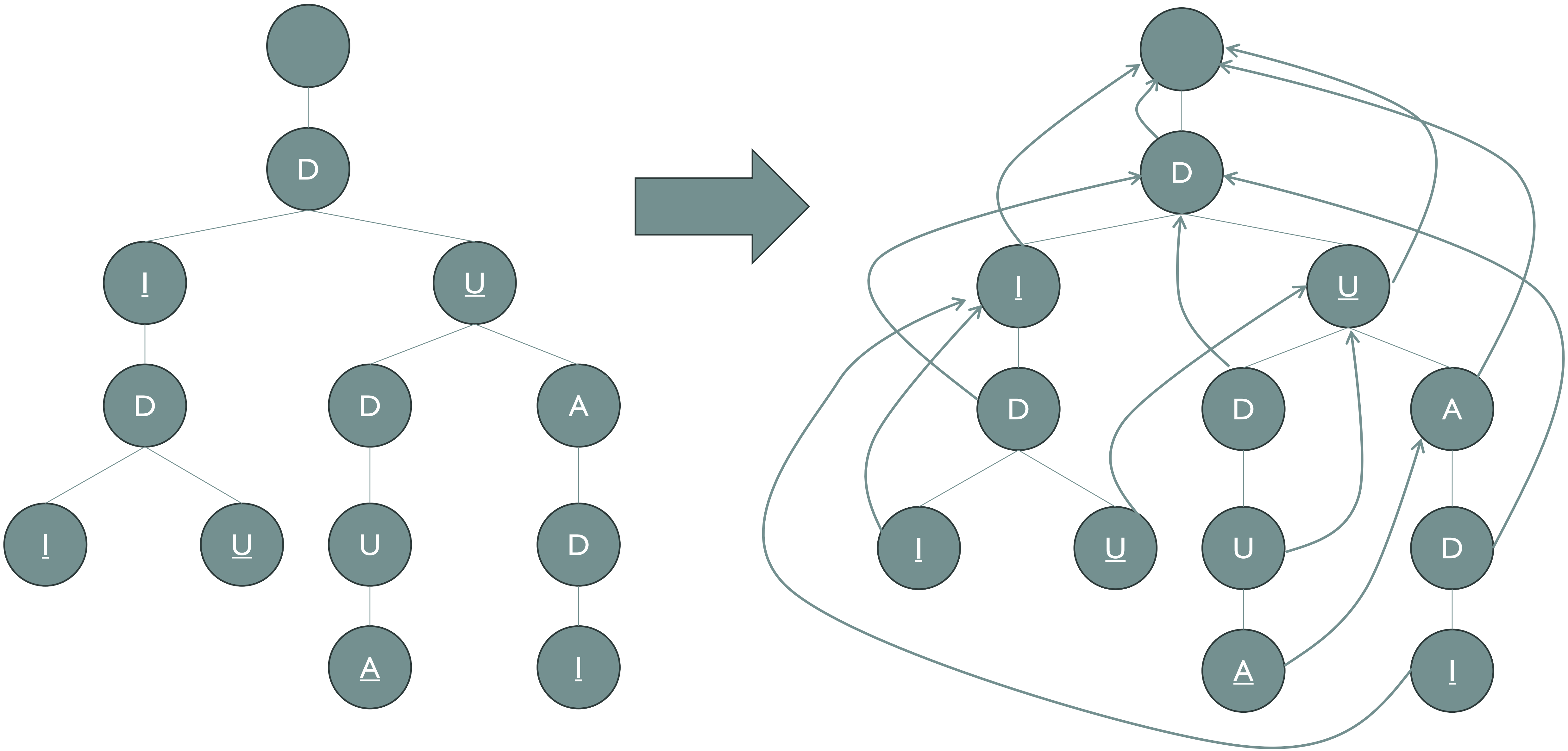
PREPROCESSING

PREPROCESSING

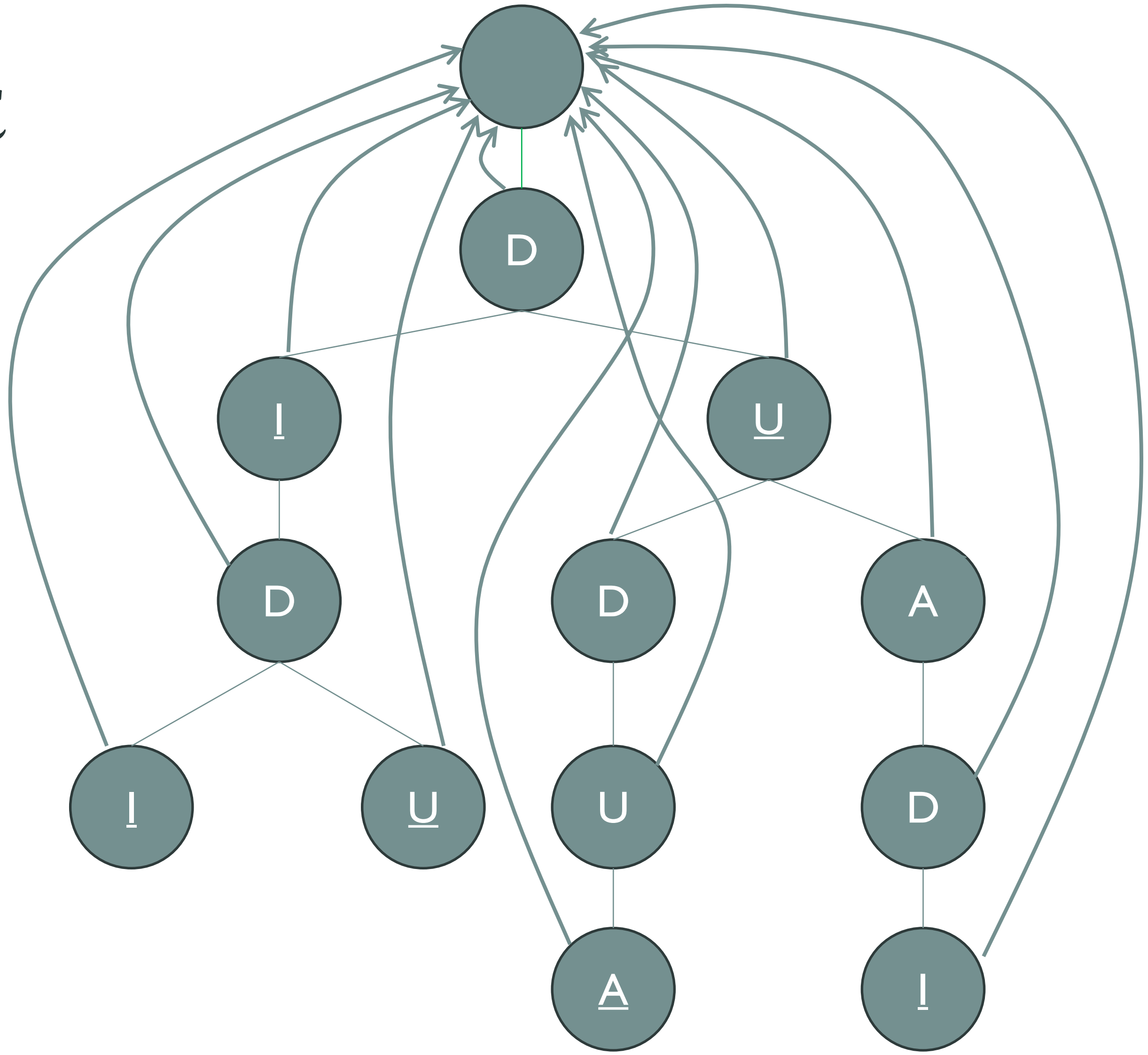
Step 1: Initialize a trie with suffix link pointing to the root to contain patterns.

Step 2: Linking the suffix link of each node to the node that satisfies the condition.

LINKING THE SUFFIX LINK

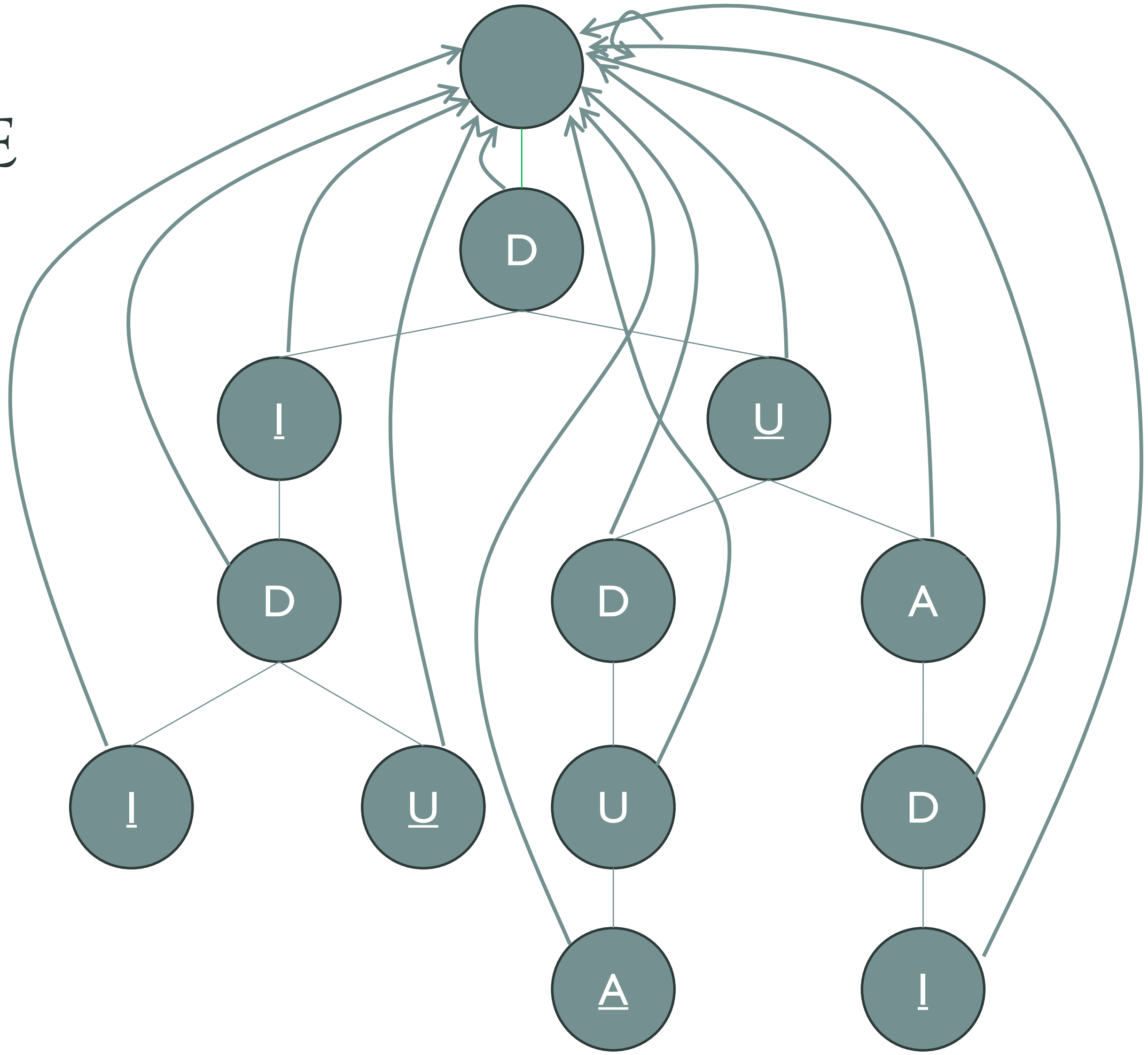


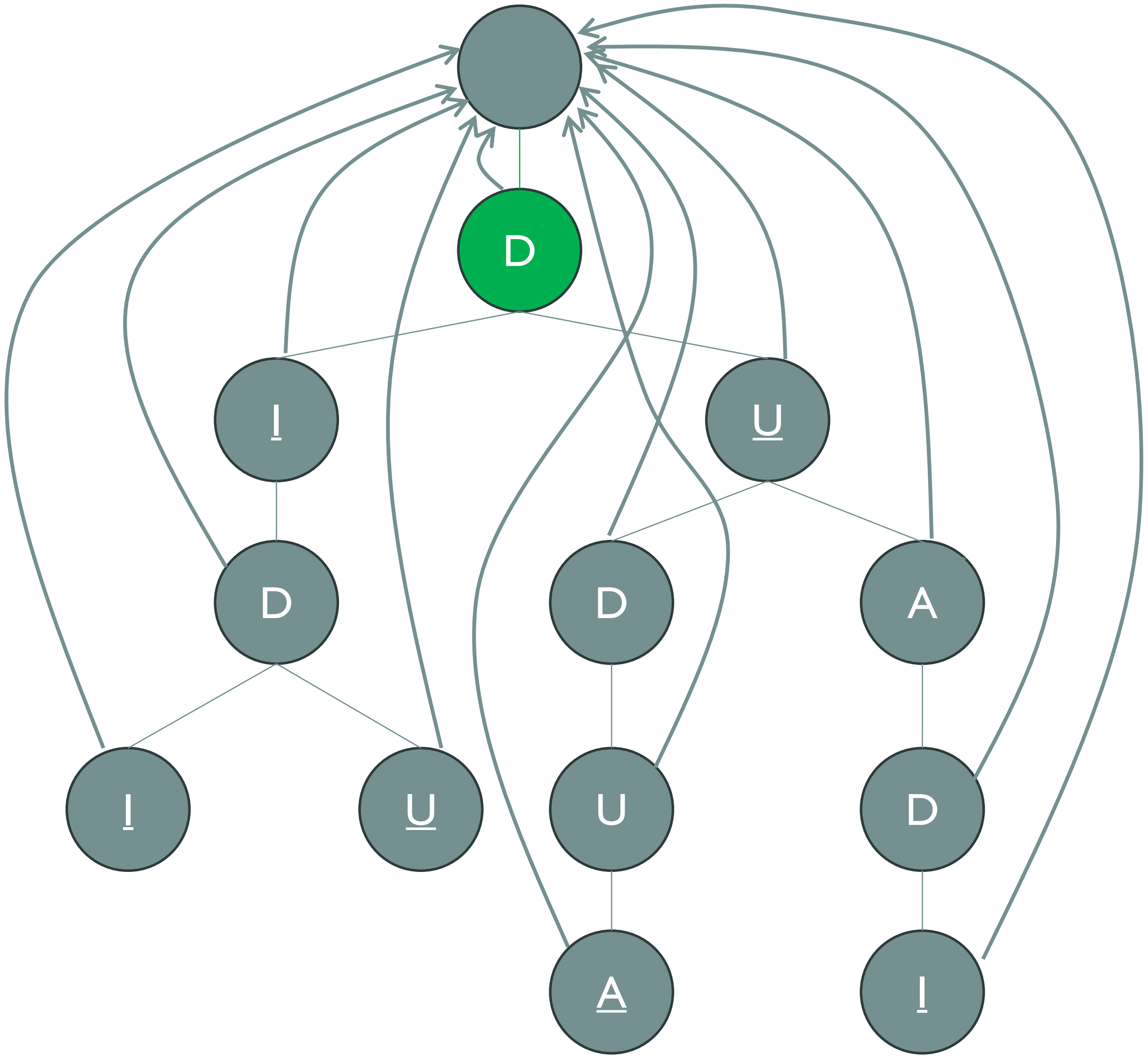
STEP 1: CREATE A TRIE AND LINK ALL THE NODE TO THE ROOT



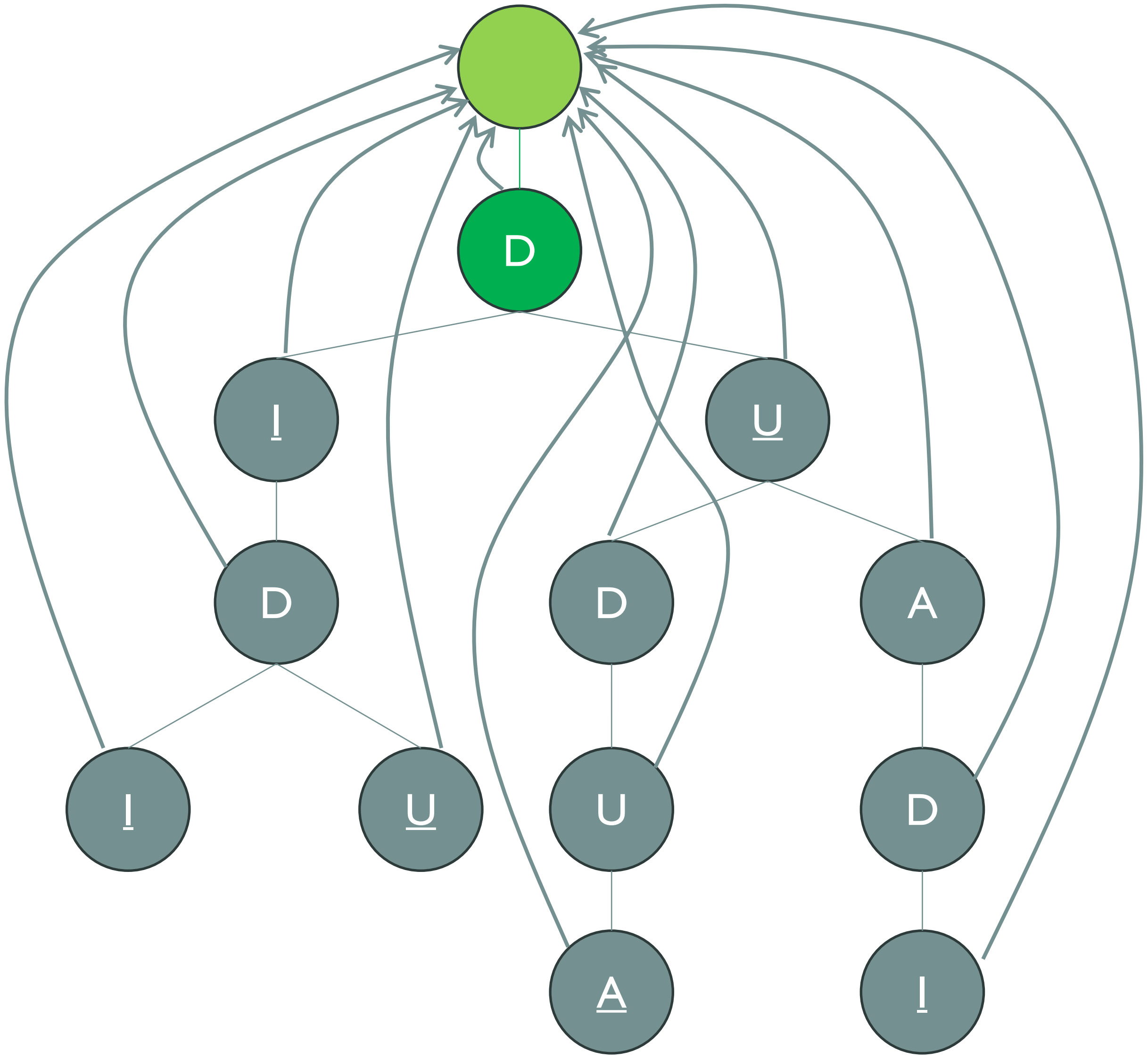
STEP 2:RELINKING THE SUFFIX LINK

At node i, we call node which the link of node i's parent points at is x.
If x is the root, we do nothing.
If the child of node x is as same as node i,
the link of node i will point to node x's child.
Otherwise, we do nothing.

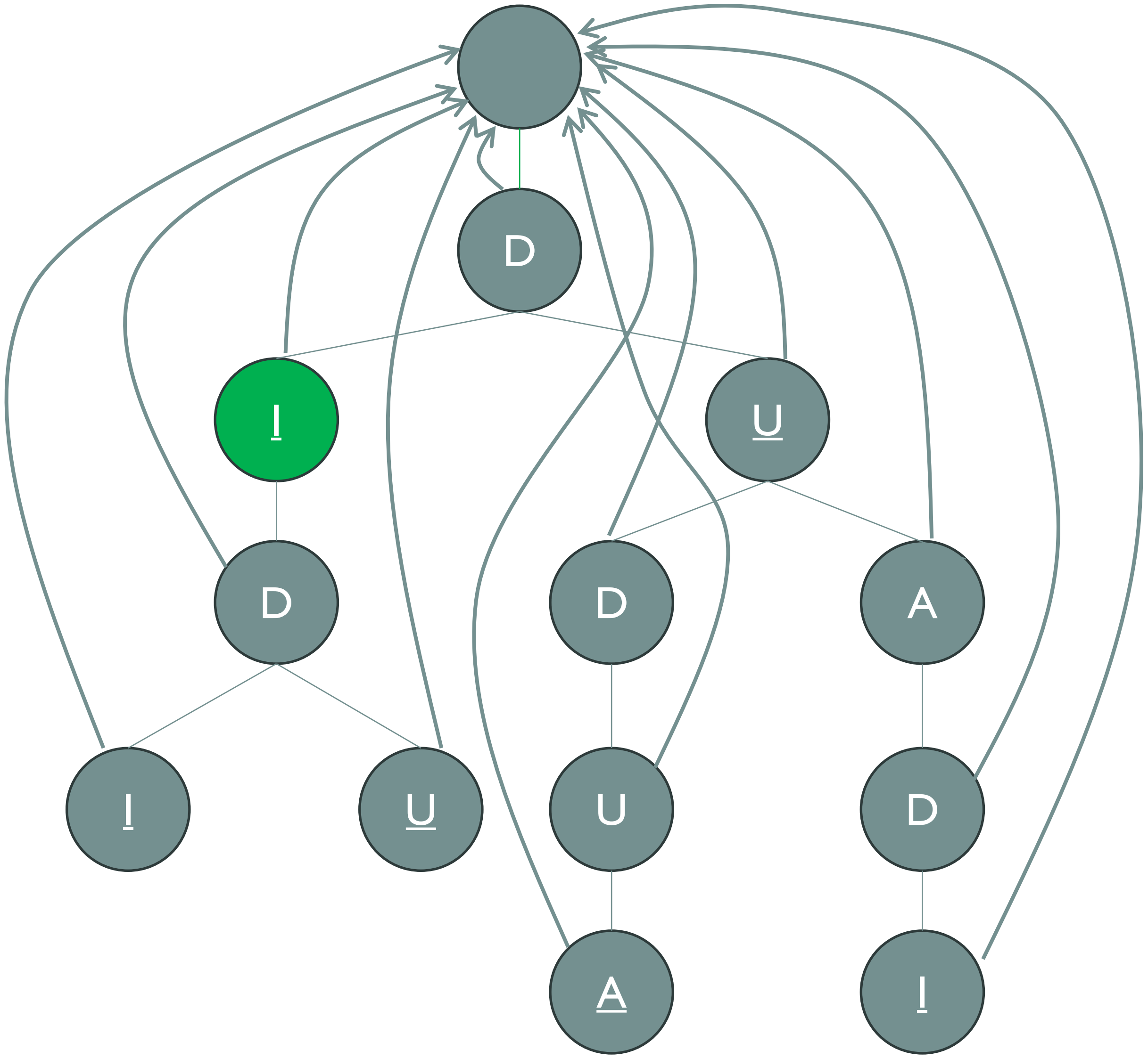



D
I
U
D
D
A
I
U
U
D
A
I

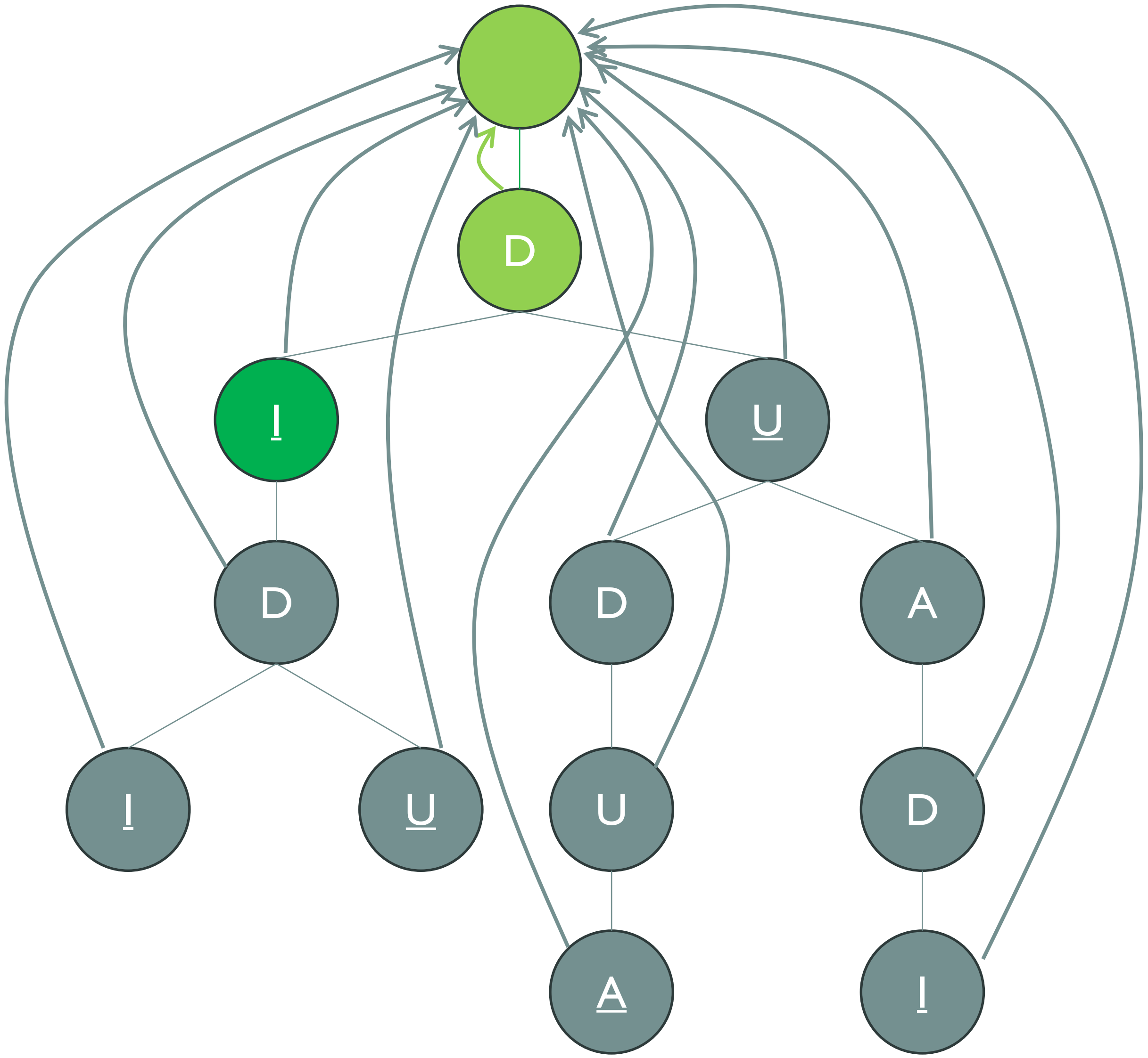
The parent of node
'D' is a root.
So we do nothing.



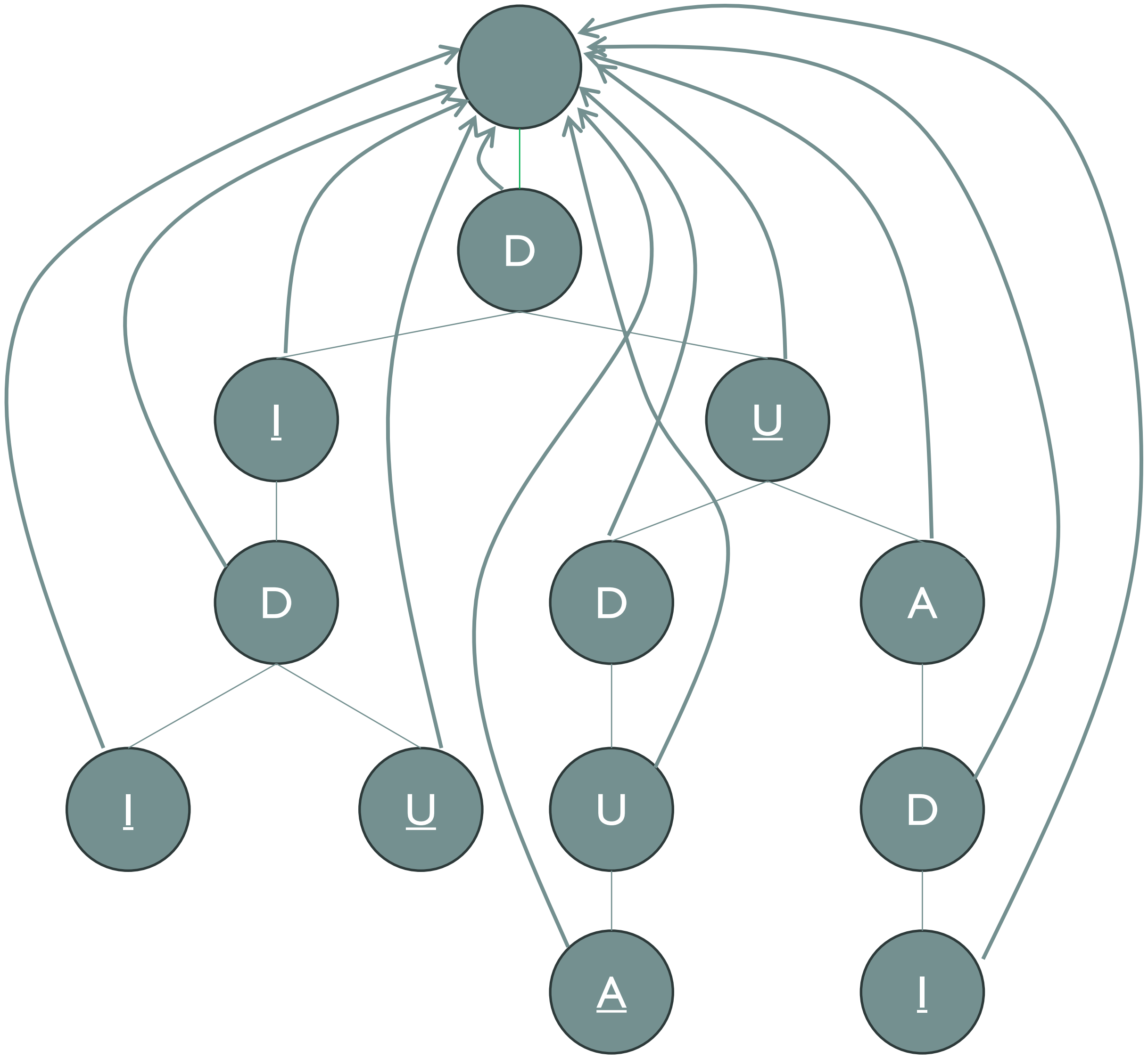
Move to Node 'I'



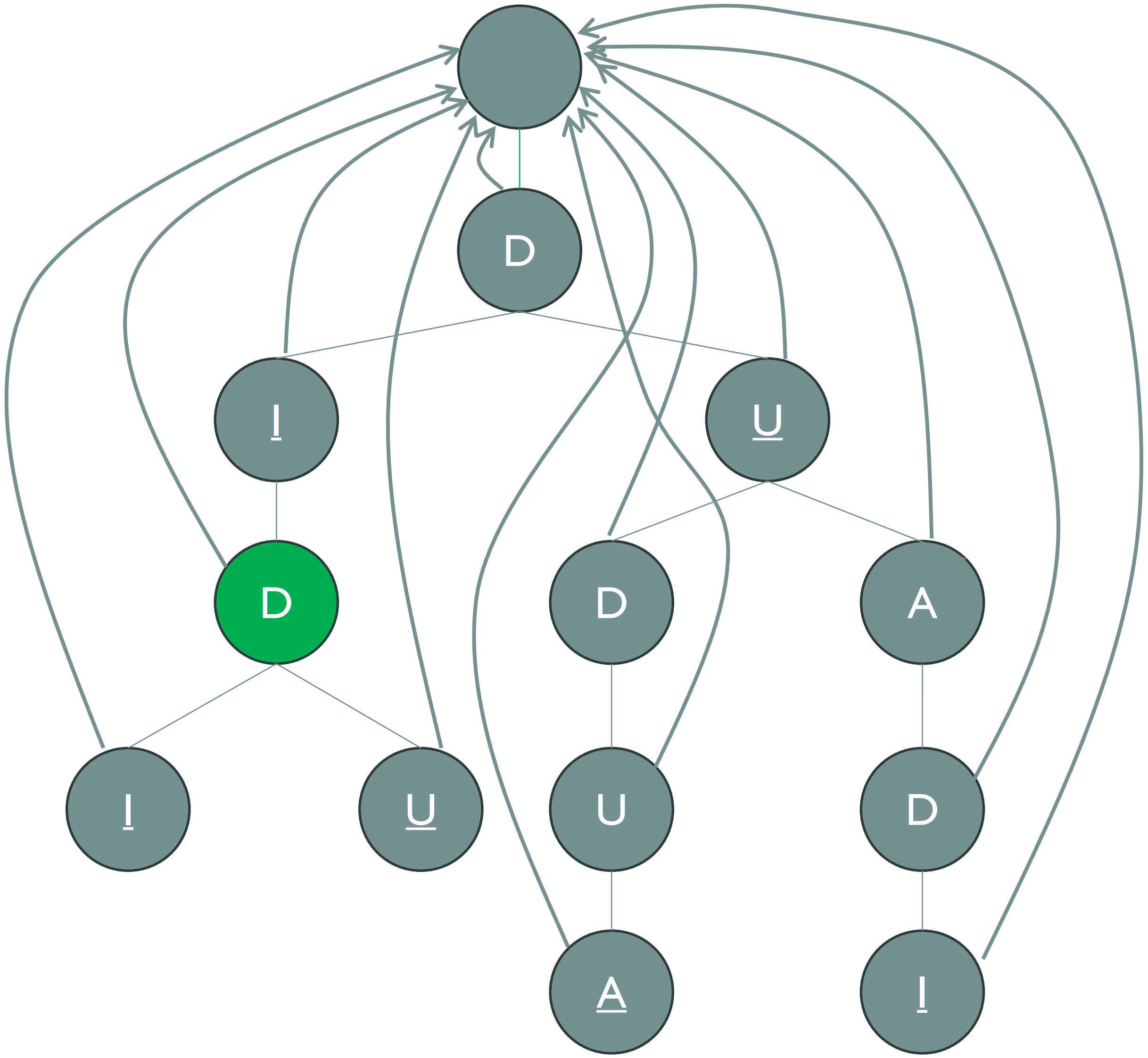
The parent of node ‘I’ is node ‘D’ and the link of that node point to the root. The root don’t have child as same node ‘I’ so we do nothing



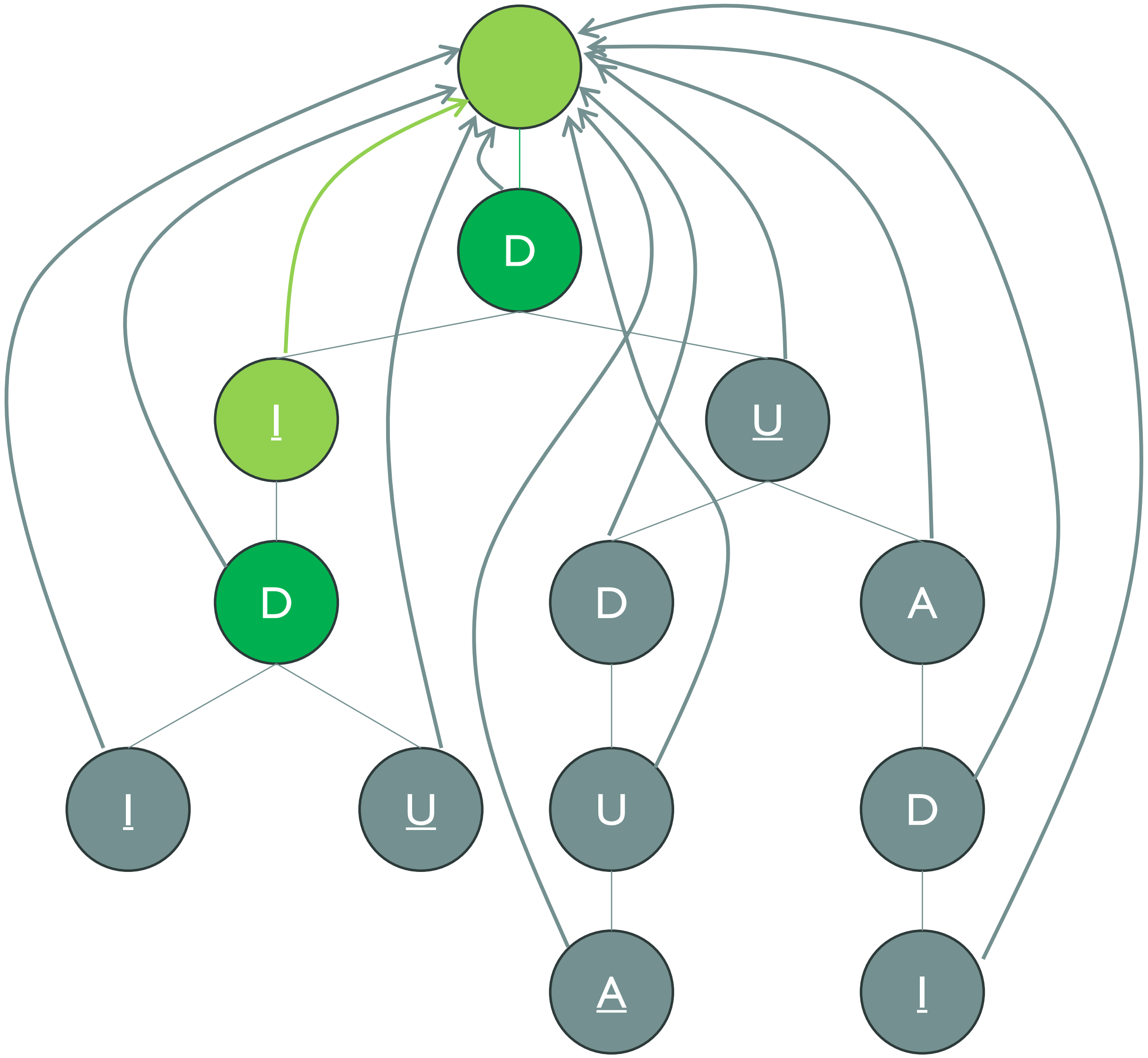
The parent of node ‘I’ is node ‘D’ and the link of that points to the root.
The root don’t have child as same node ‘I’ so we do nothing



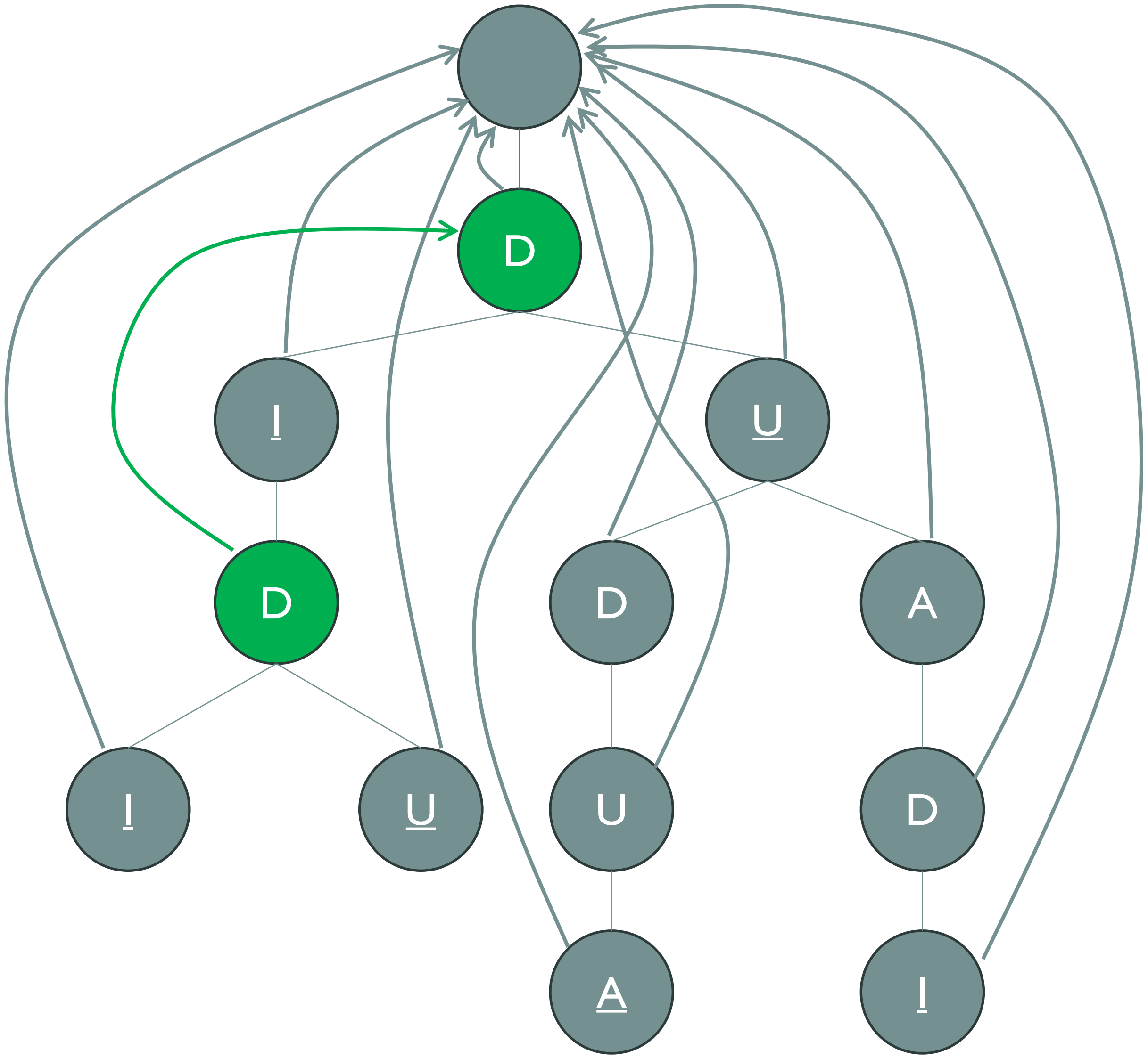
Move to node ‘D’



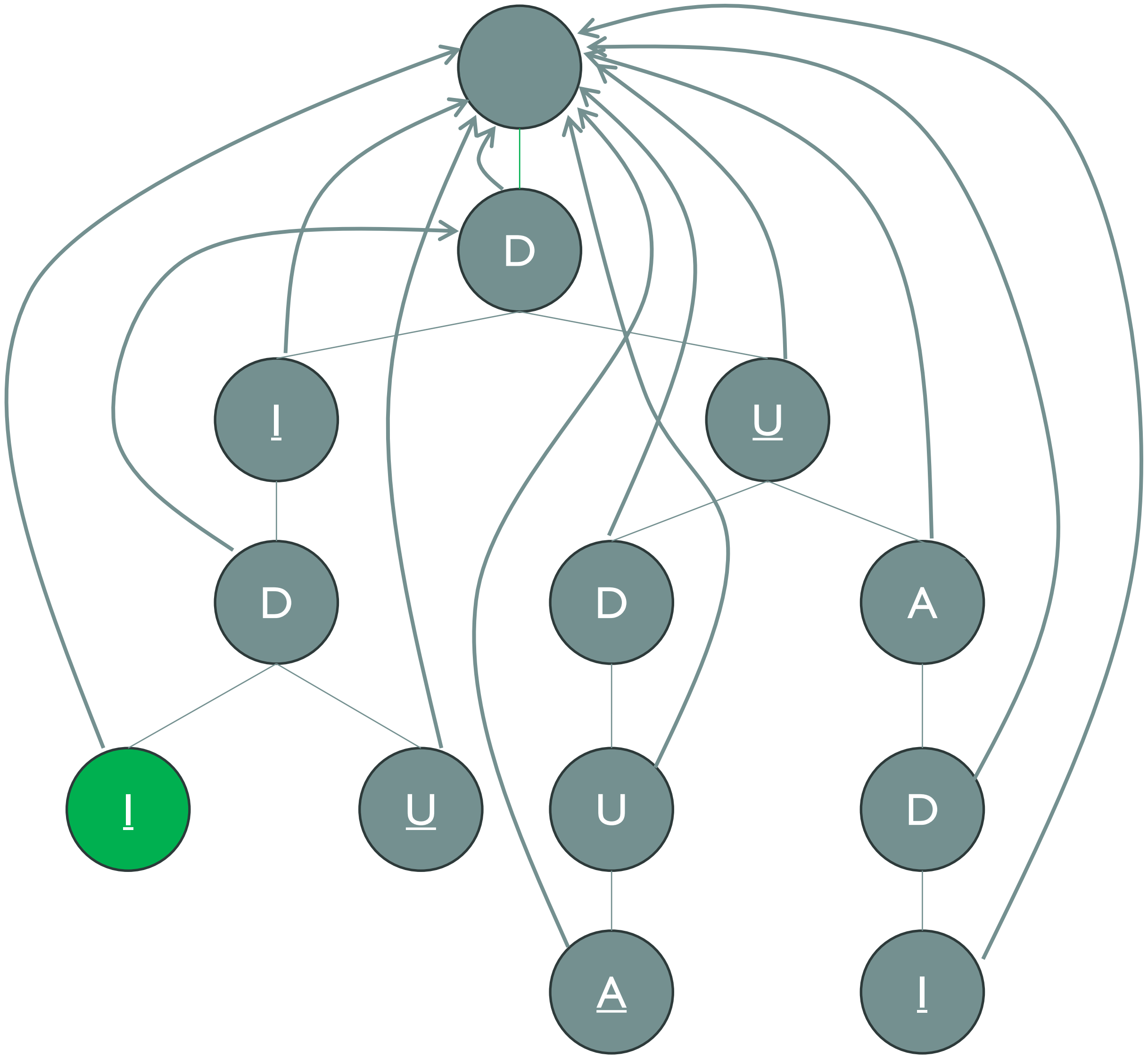
The parent of node 'D' is node 'I' and the link of that points to the root.
The root has node 'D' as its child node so the link of node 'D' will point to that child node.



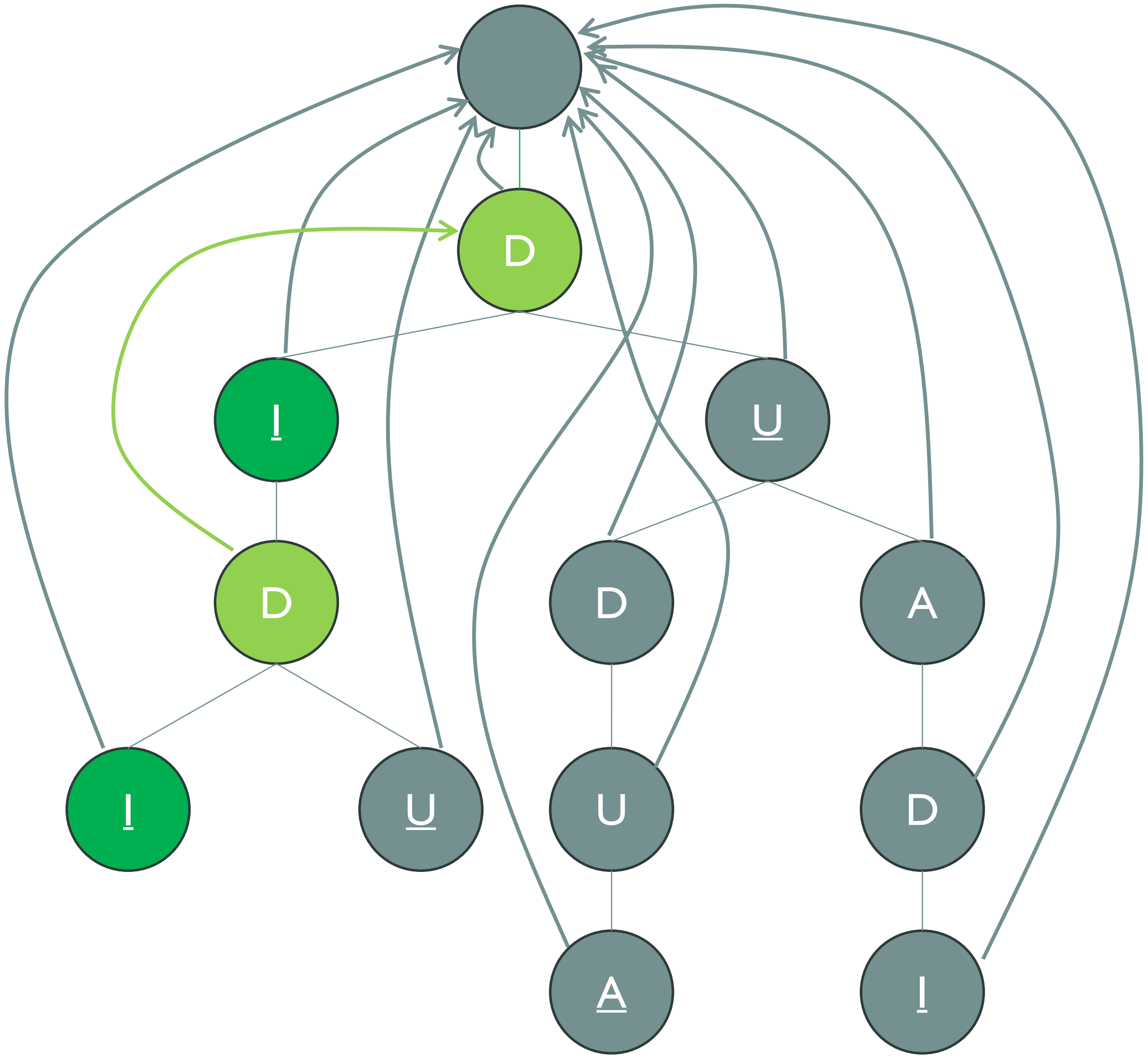
The parent of node ‘D’ is node ‘I’ and the link of that points to the root.
The root has node ‘D’ as its child node so the link of node ‘D’ will point to that child node.



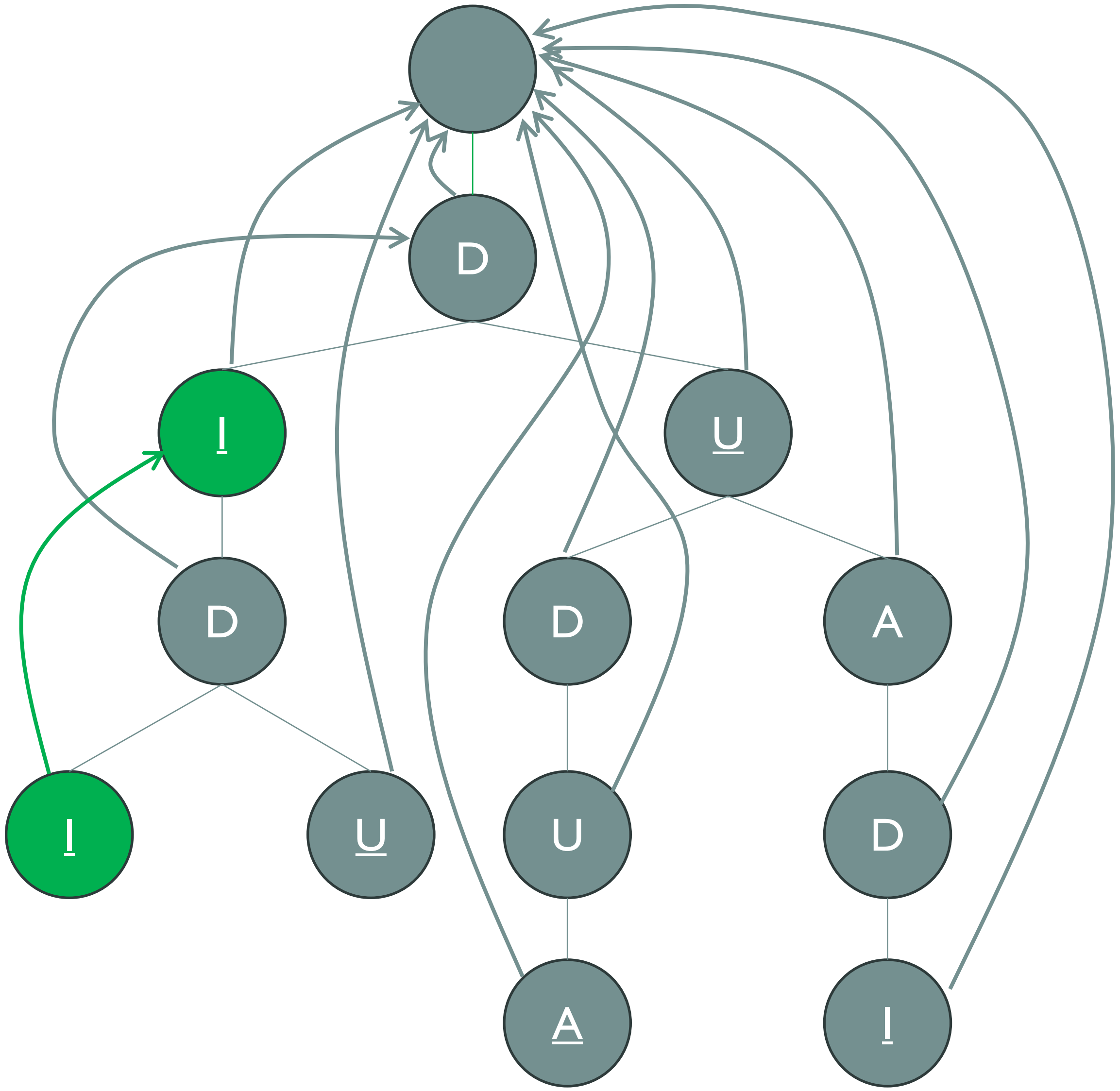
Move to node ‘I’



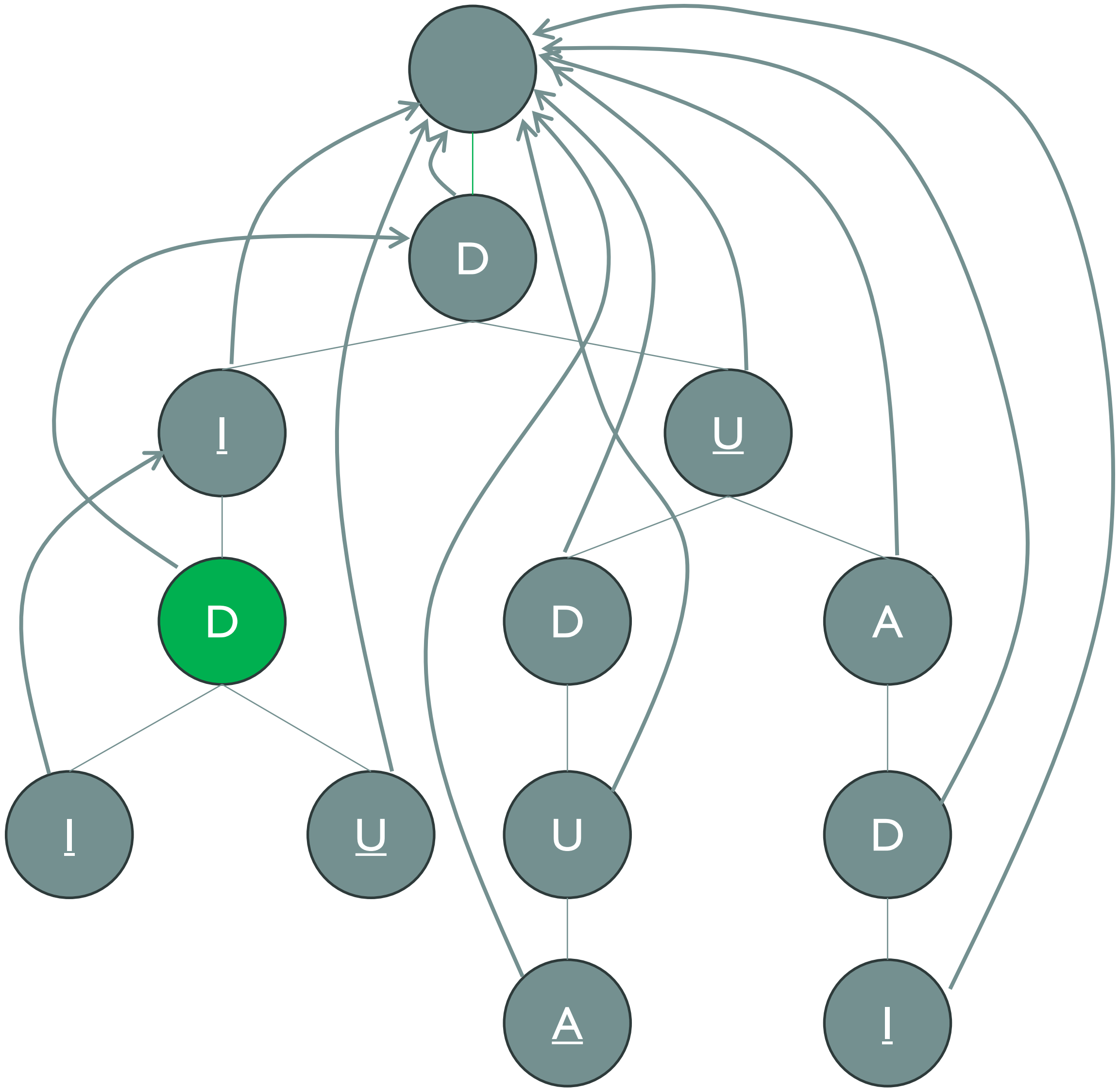
The parent of node ‘I’ is node ‘D’ and the link of that points to another node ‘D’. That node ‘D’ has node ‘I’ as its child node so the link of node ‘I’ will point to that child node.



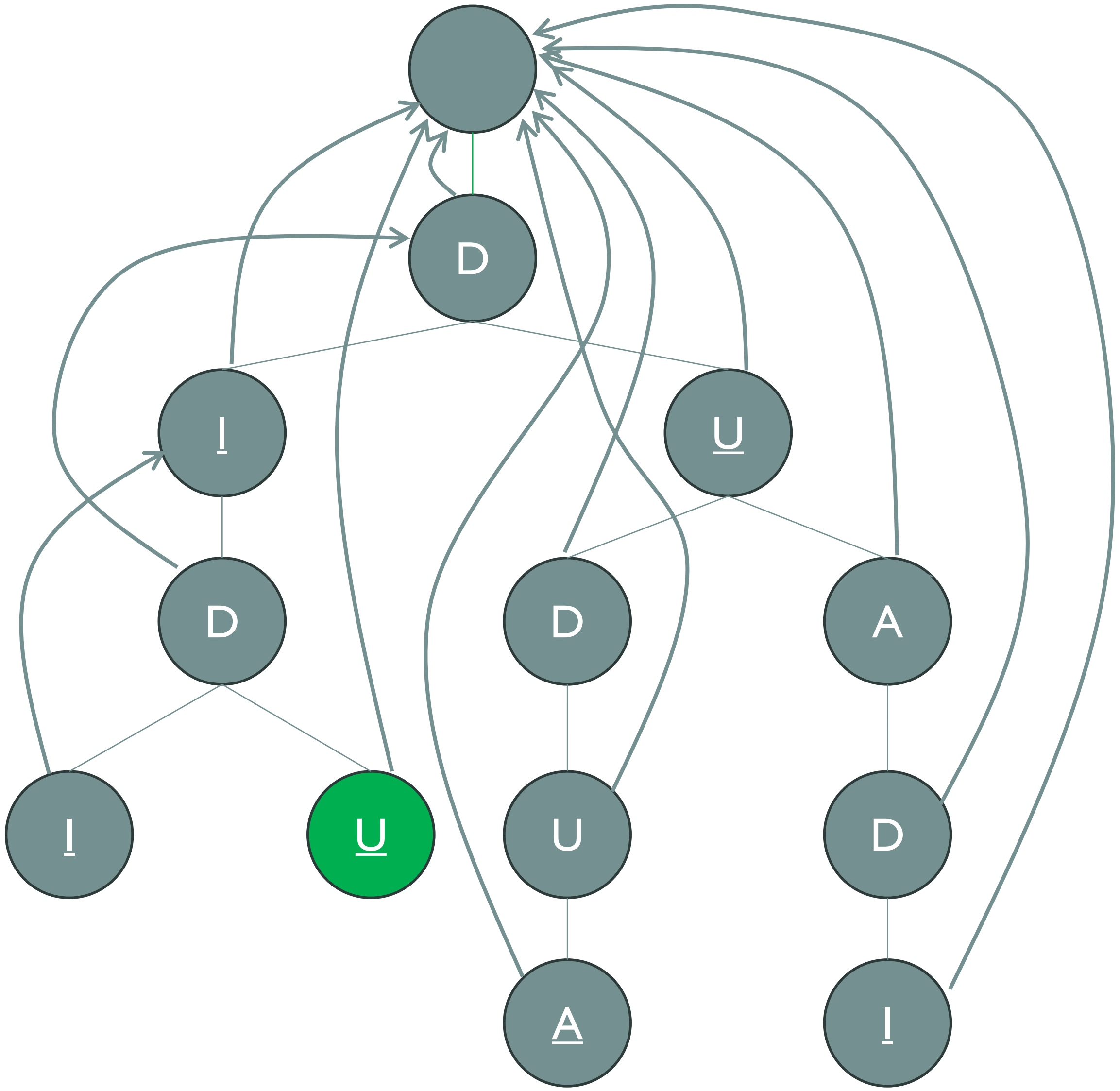
The parent of node ‘I’ is node ‘D’ and the link of that node points to another node ‘D’.
That node ‘D’ has node ‘I’ as its child node so the link of node ‘I’ will point to that child node.



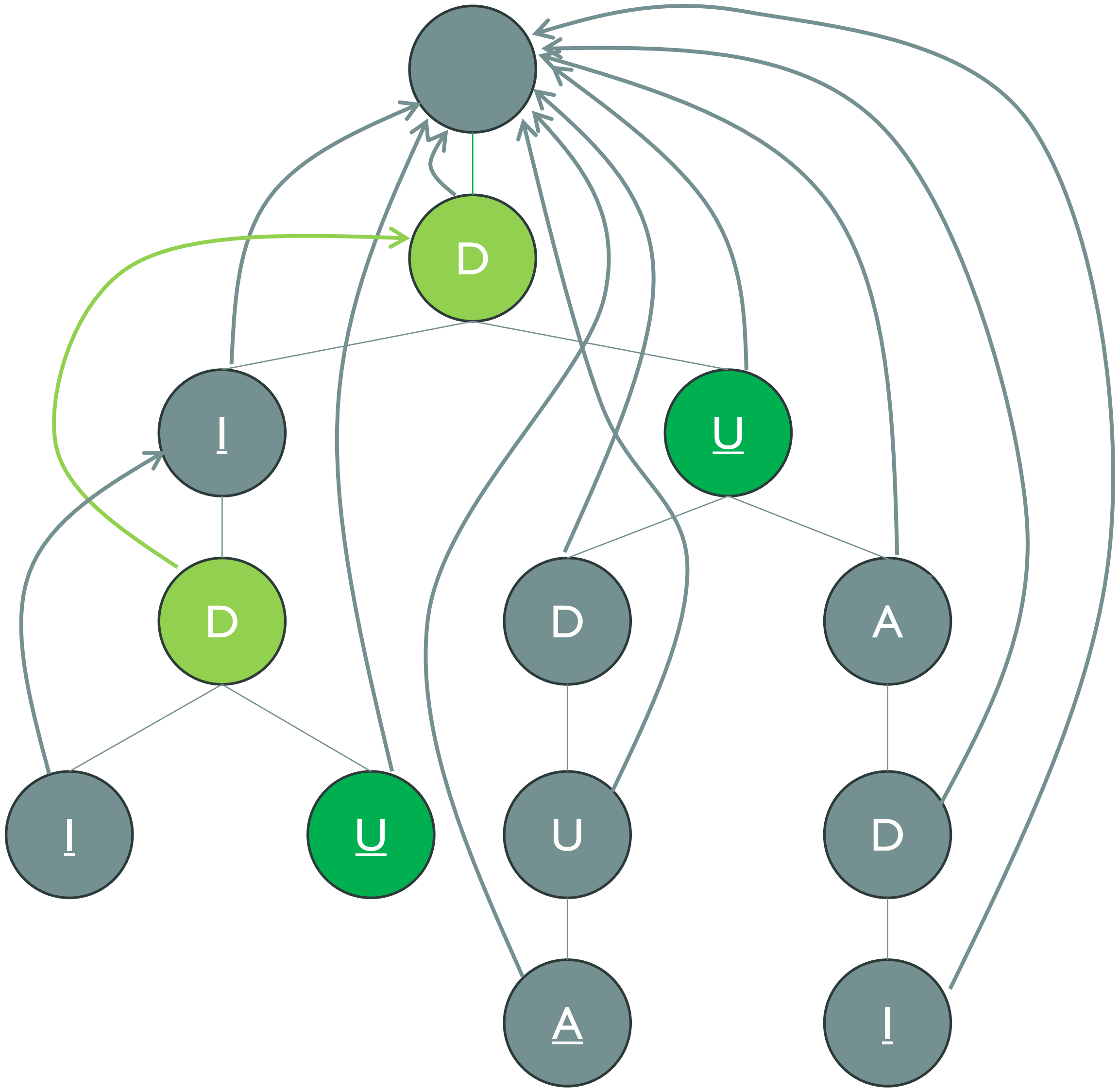
Node ‘I’ doesn’t have any child node so we move back to node ‘D’ and move to another ‘D’ ’s child node



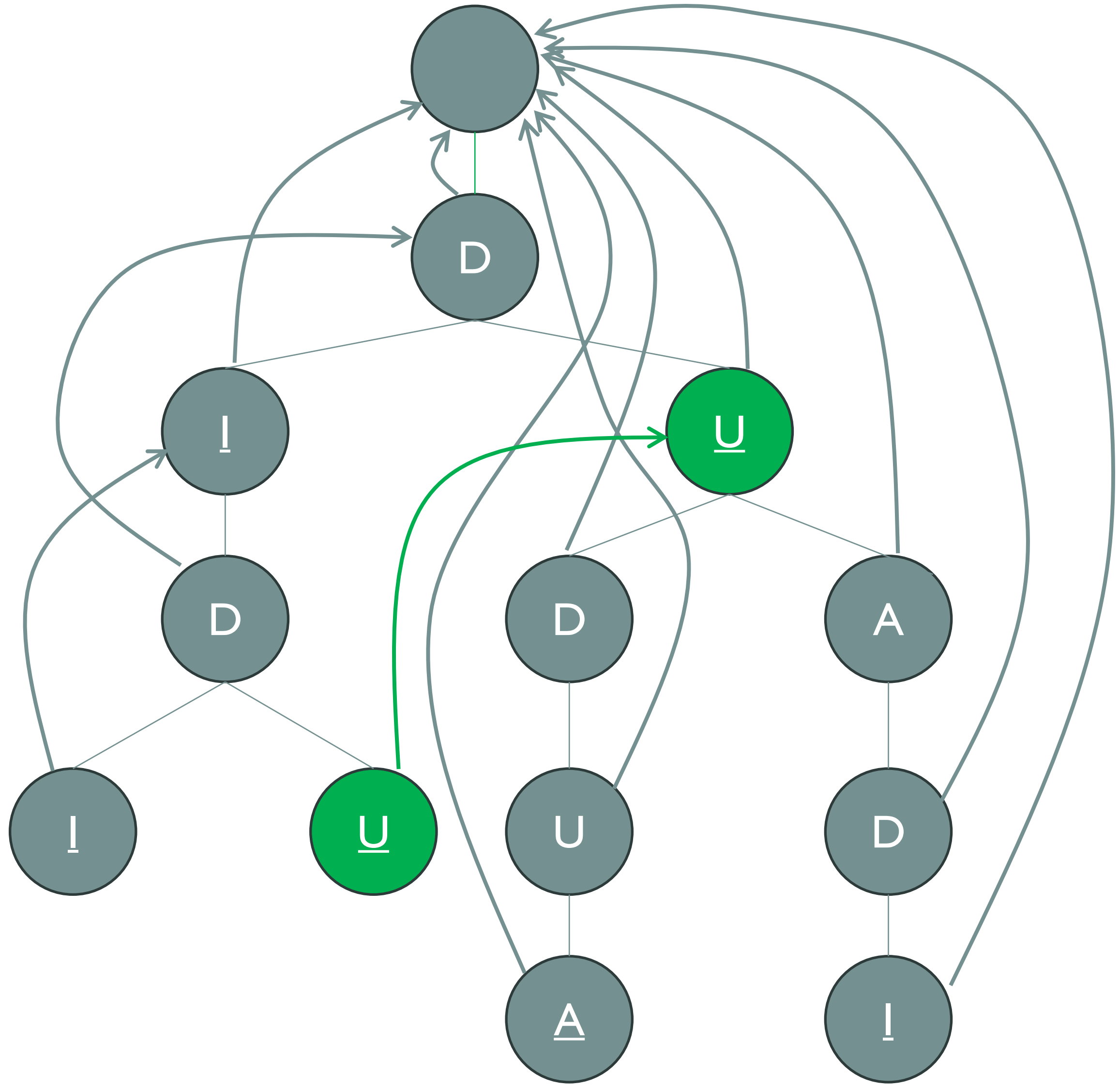
Node 'I' doesn't have any child node so we move back to node 'D' and move to another 'D' 's child node



The parent of node ‘U’ is node ‘D’ and the link of that node points to another node ‘D’.
That node ‘D’ has node ‘U’ as its child node so the link of node ‘U’ will point to that child node.



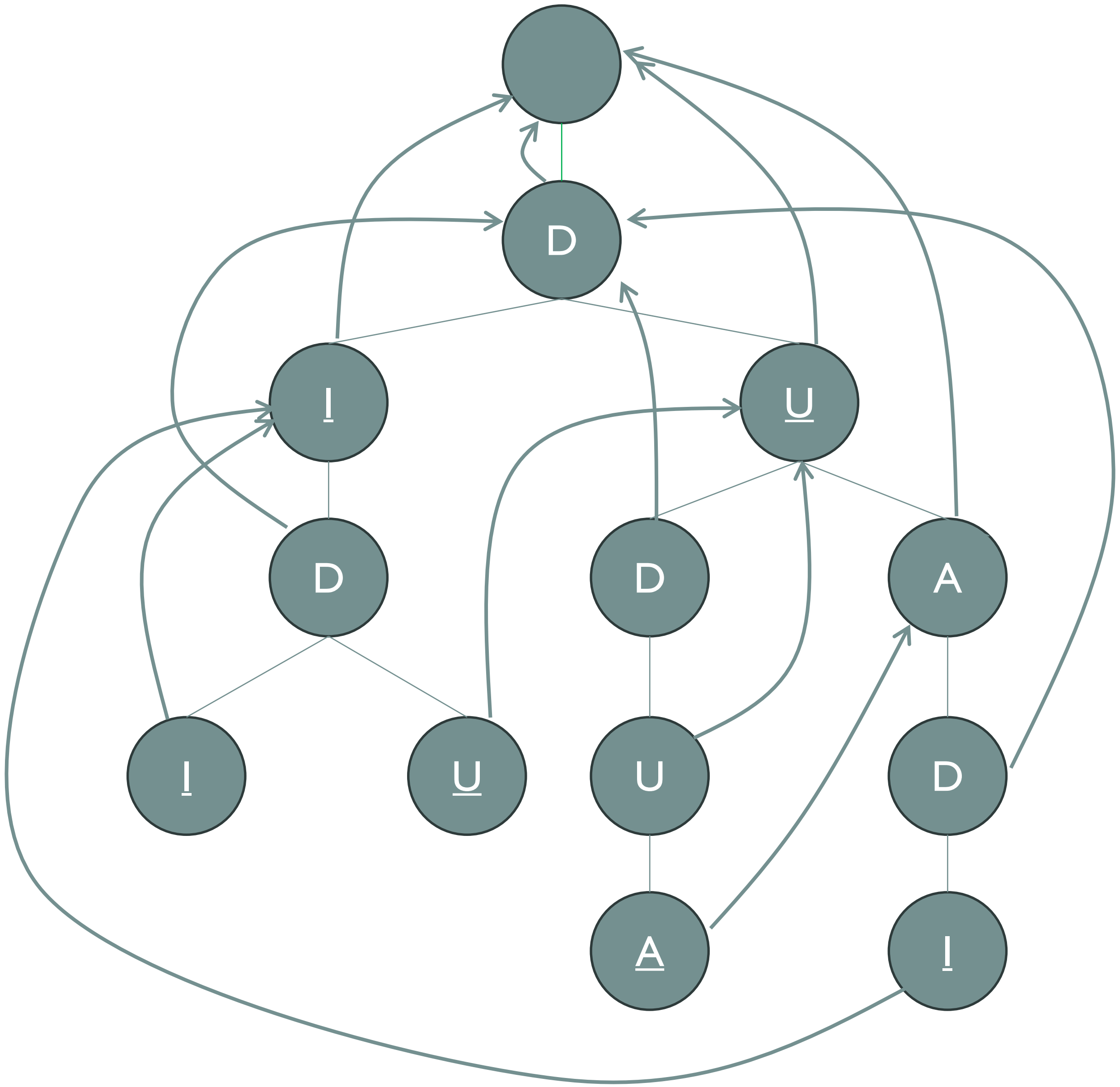
The parent of node ‘U’ is node ‘D’ and the link of that node point to another node ‘D’.
That node ‘D’ has node ‘U’ as its child node so the link of node ‘U’ will point to that node.



Continue with other nodes, we will have a trie tree with links like this picture

Time Complexity? $O(M)$

Space Complexity? $O(M)$



SEARCHING/ MATCHING PROCESS

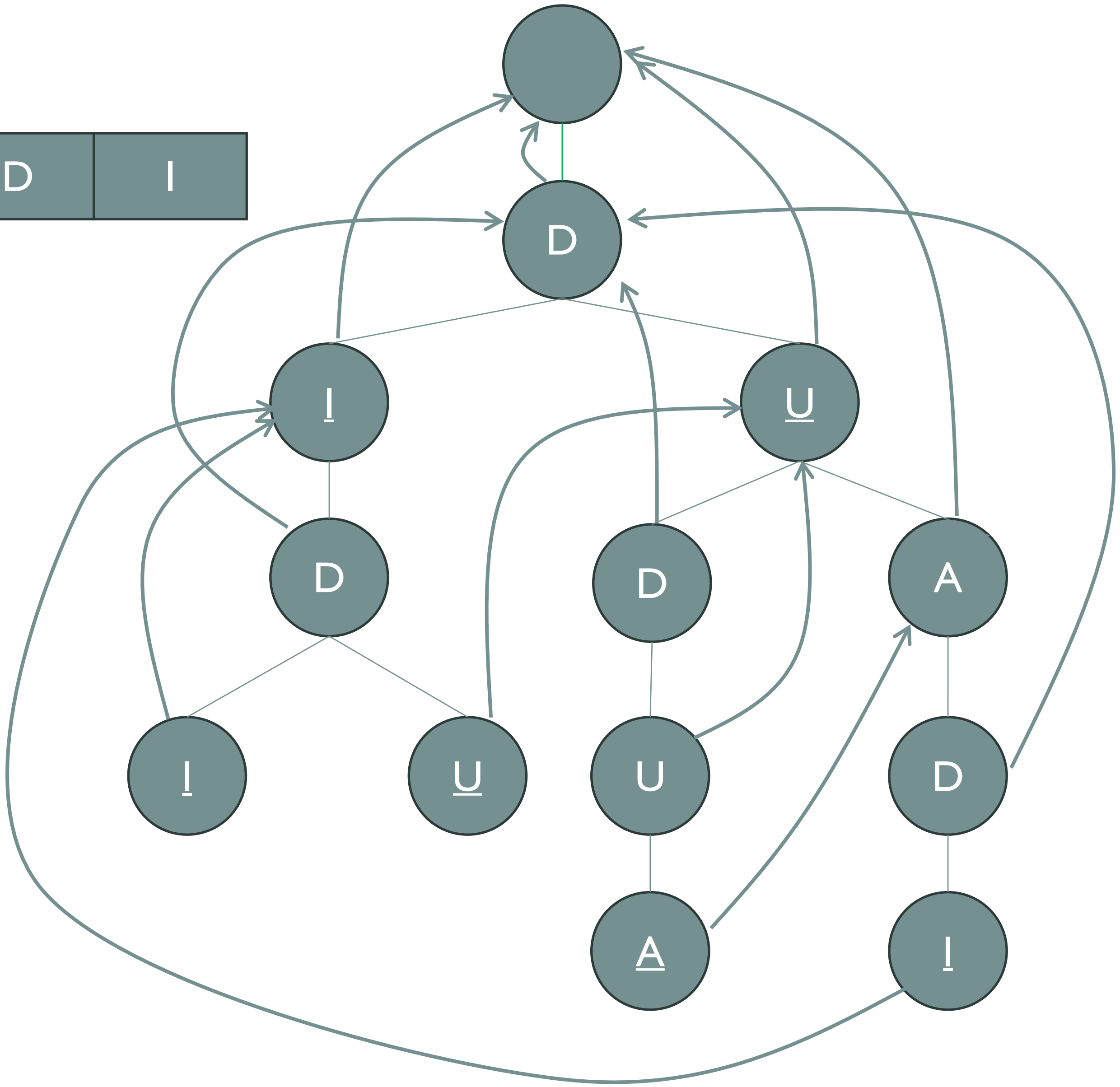
SEARCHING/ MATCHING PROCESS

At position i, if the current has the child text[i], we will travel to that child node. Otherwise, we use the suffix link to trace back to the other node to check whether that node has that child text[i]. This procedure repeats until there is a child node text[i] or the node is the root. If the node is the root, the searching process will stop. But there is a child node text[i], we will travel to that node and continue the procedure.

After travel to that node, if the node is the end of a pattern and we have not counted that pattern, so we count it. Then we trace back using the suffix link procedure until the node is the root to check that all the prefixes are the end of any pattern or not. If there is the end of the pattern and it has not been counted, we count it.

The text: “DIDUDUADI”

D	I	D	U	D	U	A	D	I

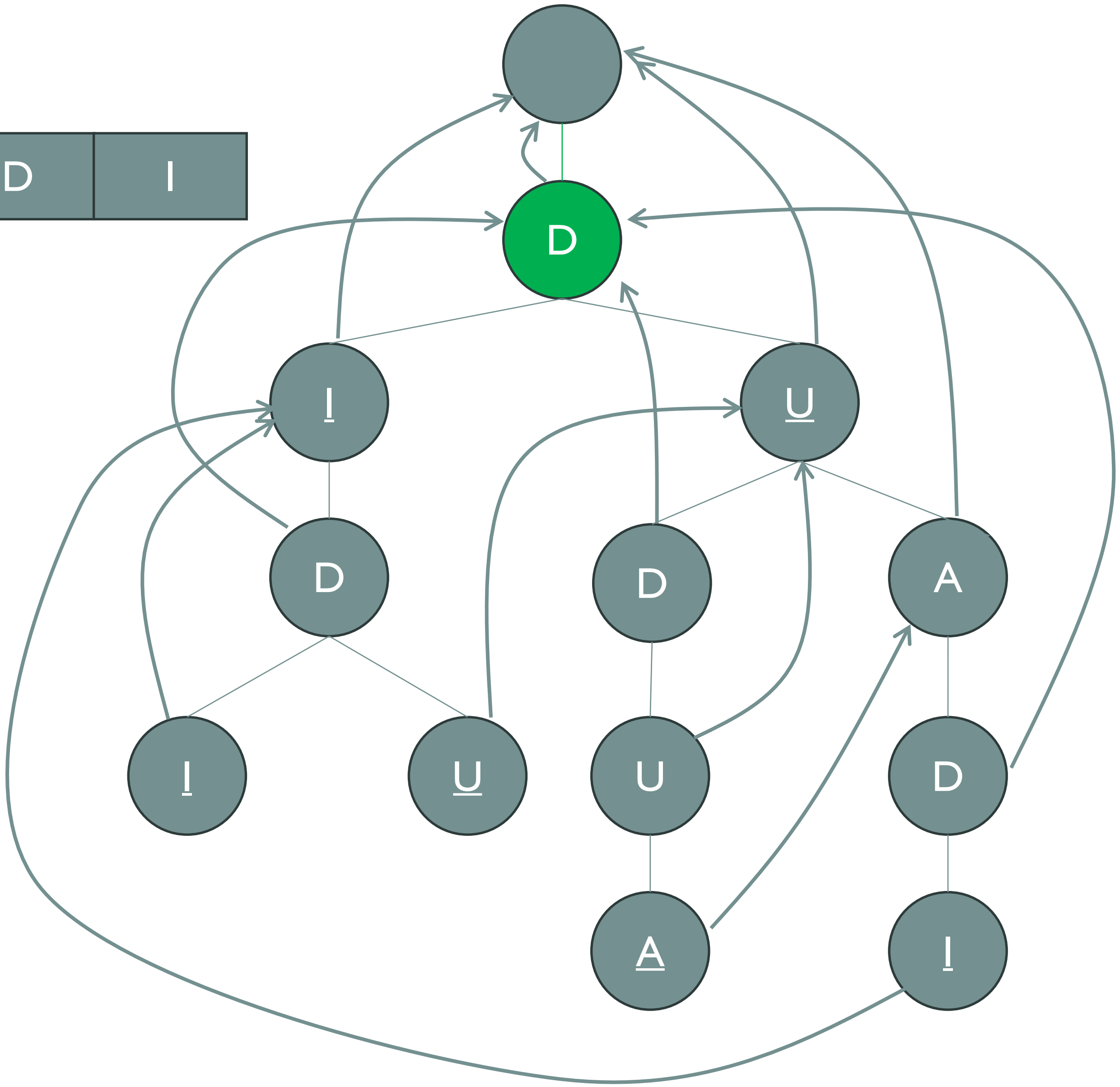


The pattern strings occur in the text:

The text: “DIDUDUADI”

D	I	D	U	D	U	A	D	I

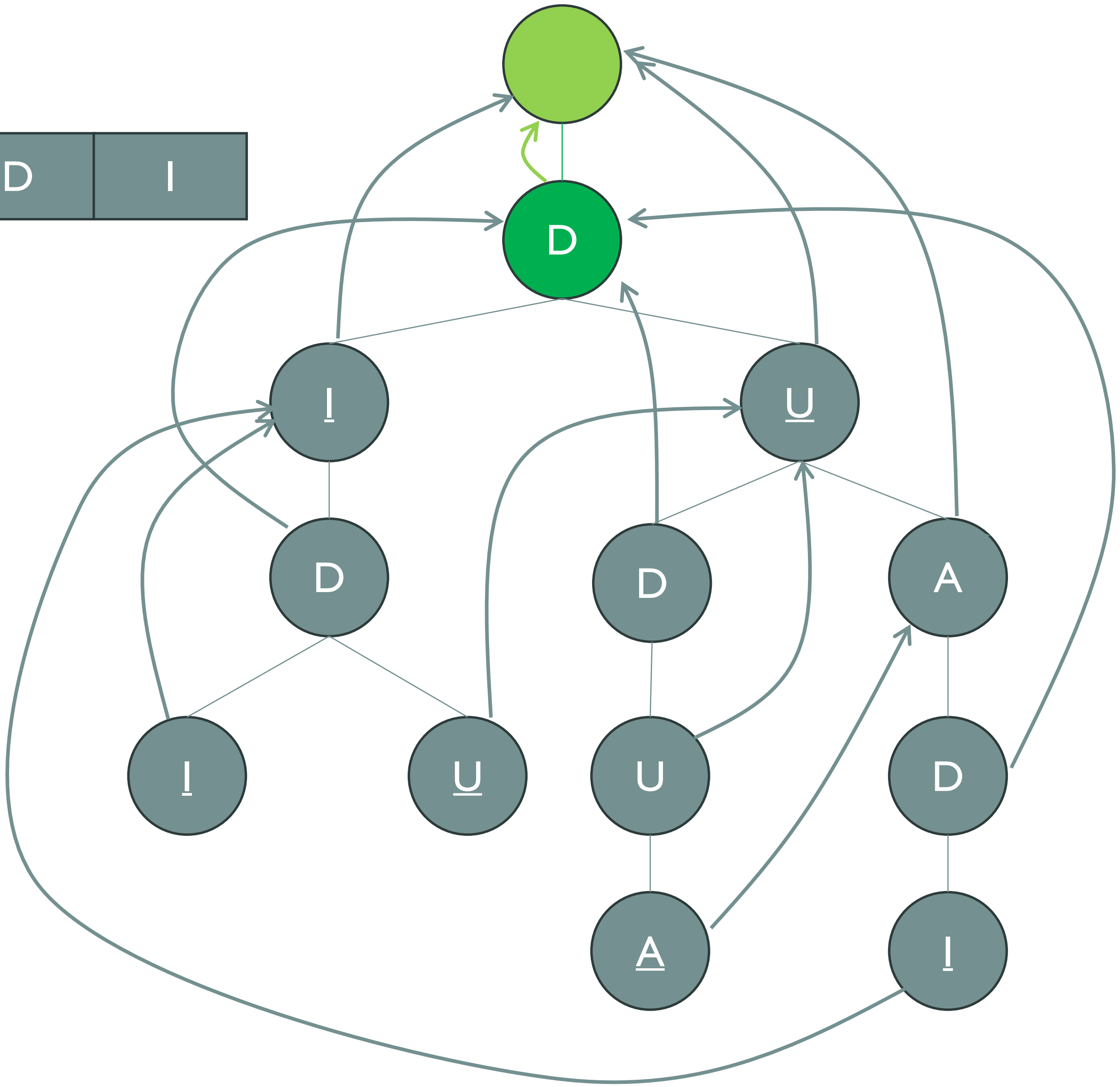
The pattern strings occur in the text:



The text: “DIDUDUADI”

D	I	D	U	D	U	A	D	I

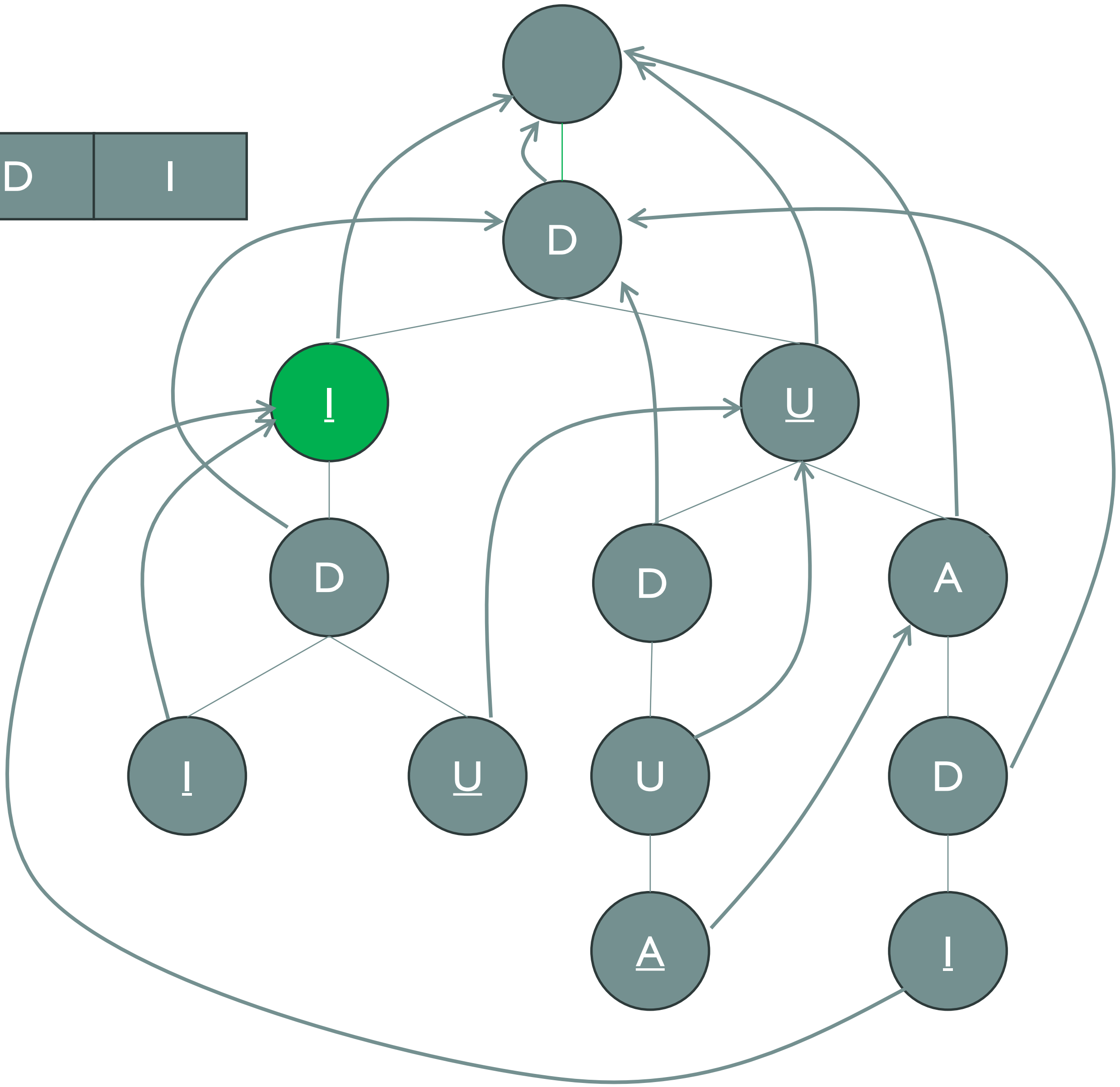
The pattern strings occur in the text:



The text: “DIDUDUADI”

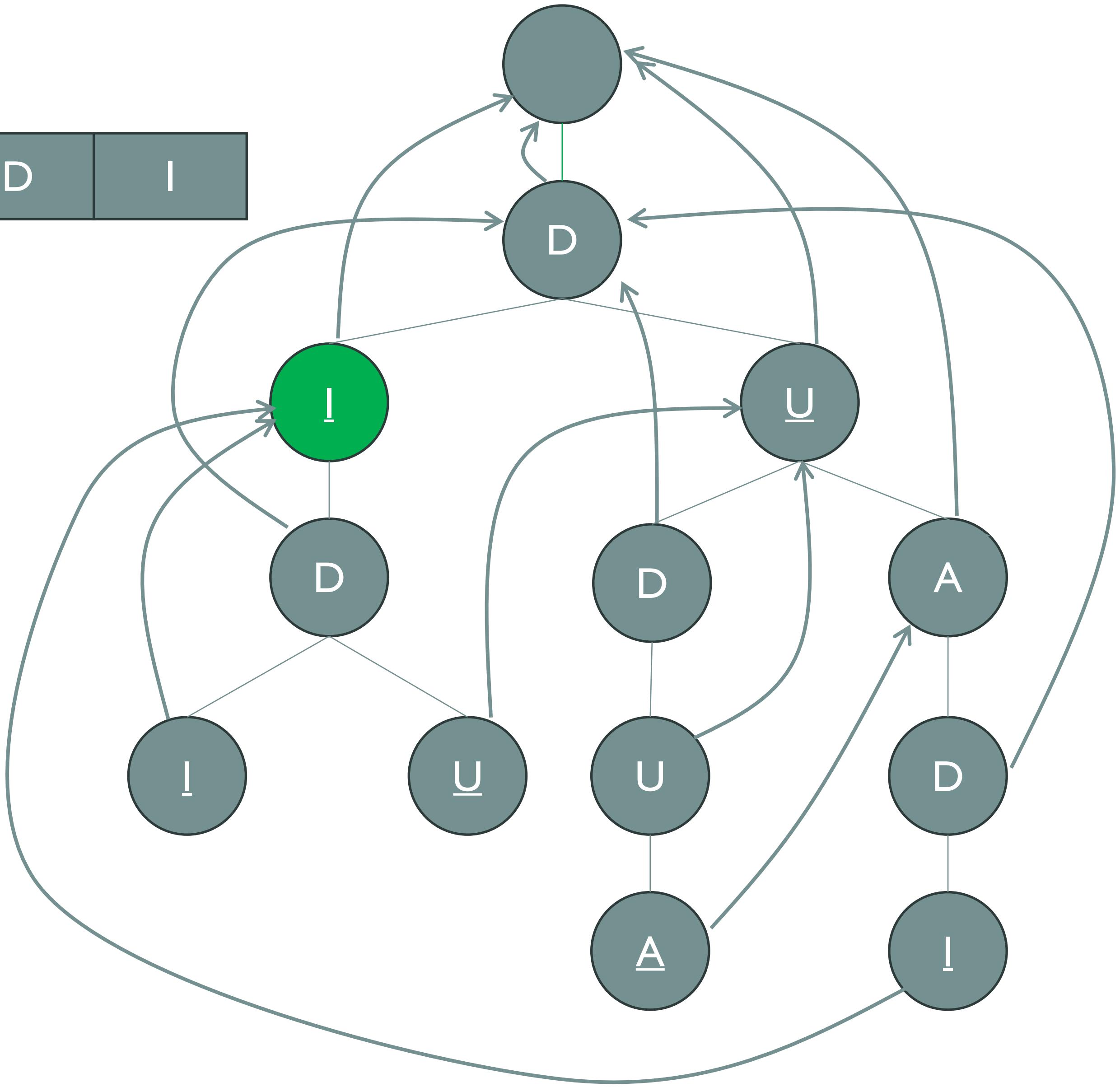
D	I	D	U	D	U	A	D	I

The pattern strings occur in the text:



The text: “DIDUDUADI”

D	I	D	U	D	U	A	D	I



The pattern strings occur in the text:

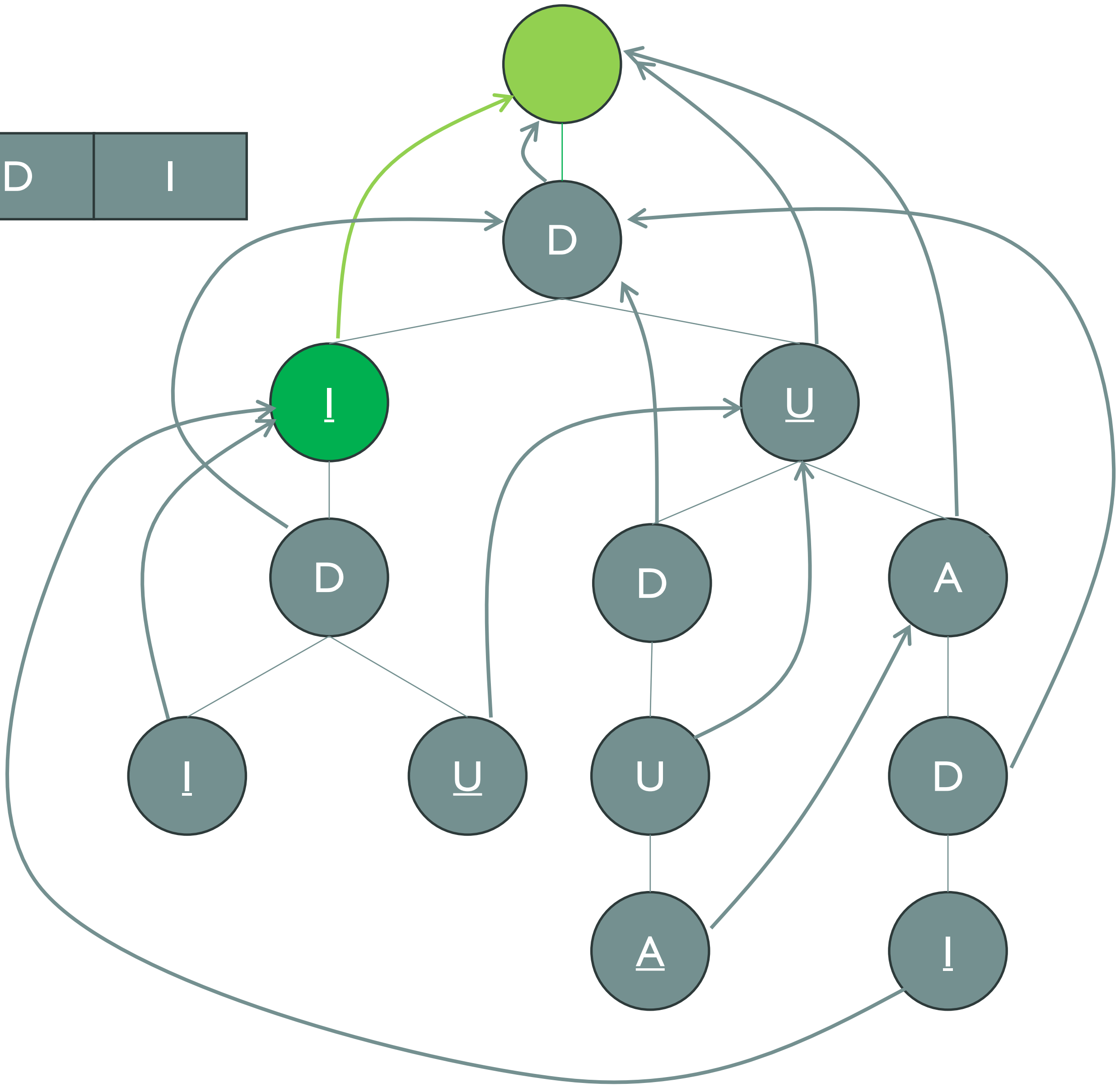
DI

The text: “DIDUDUADI”

D	I	D	U	D	U	A	D	I

The pattern strings occur in the text:

DI

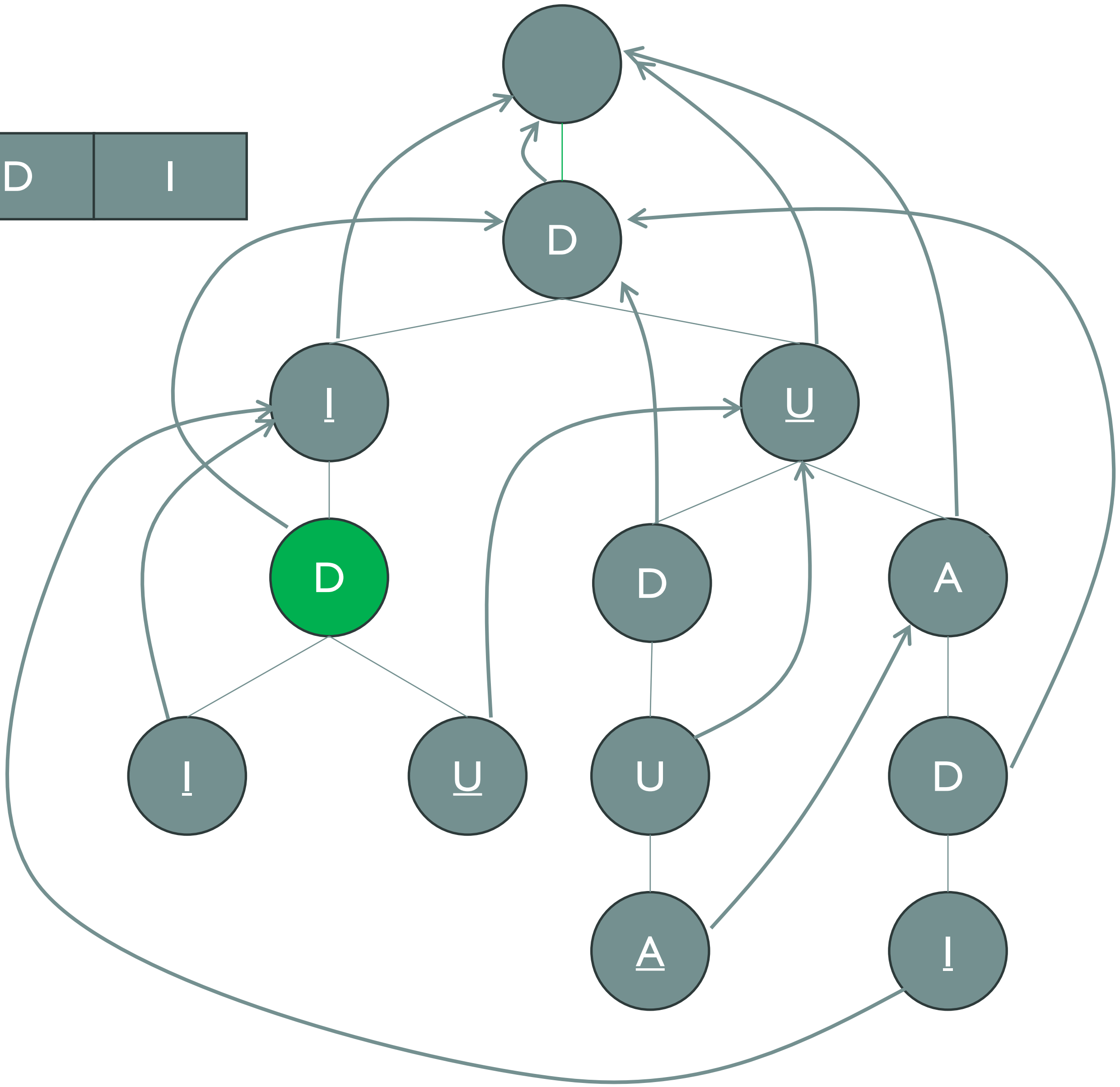


The text: “DIDUDUADI”

D	I	D	U	D	U	A	D	I

The pattern strings occur in the text:

DI

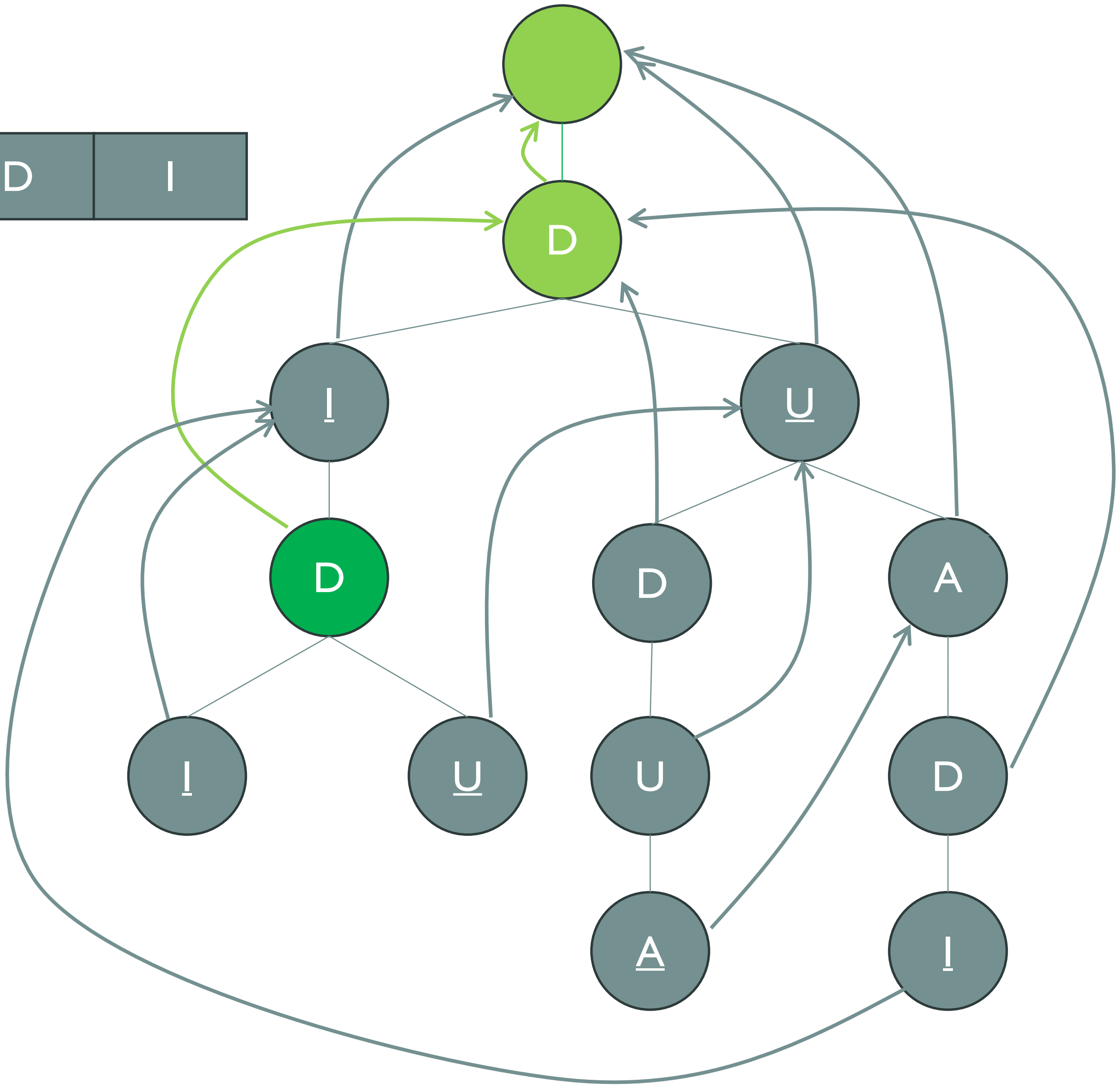


The text: “DIDUDUADI”

D	I	D	U	D	U	A	D	I

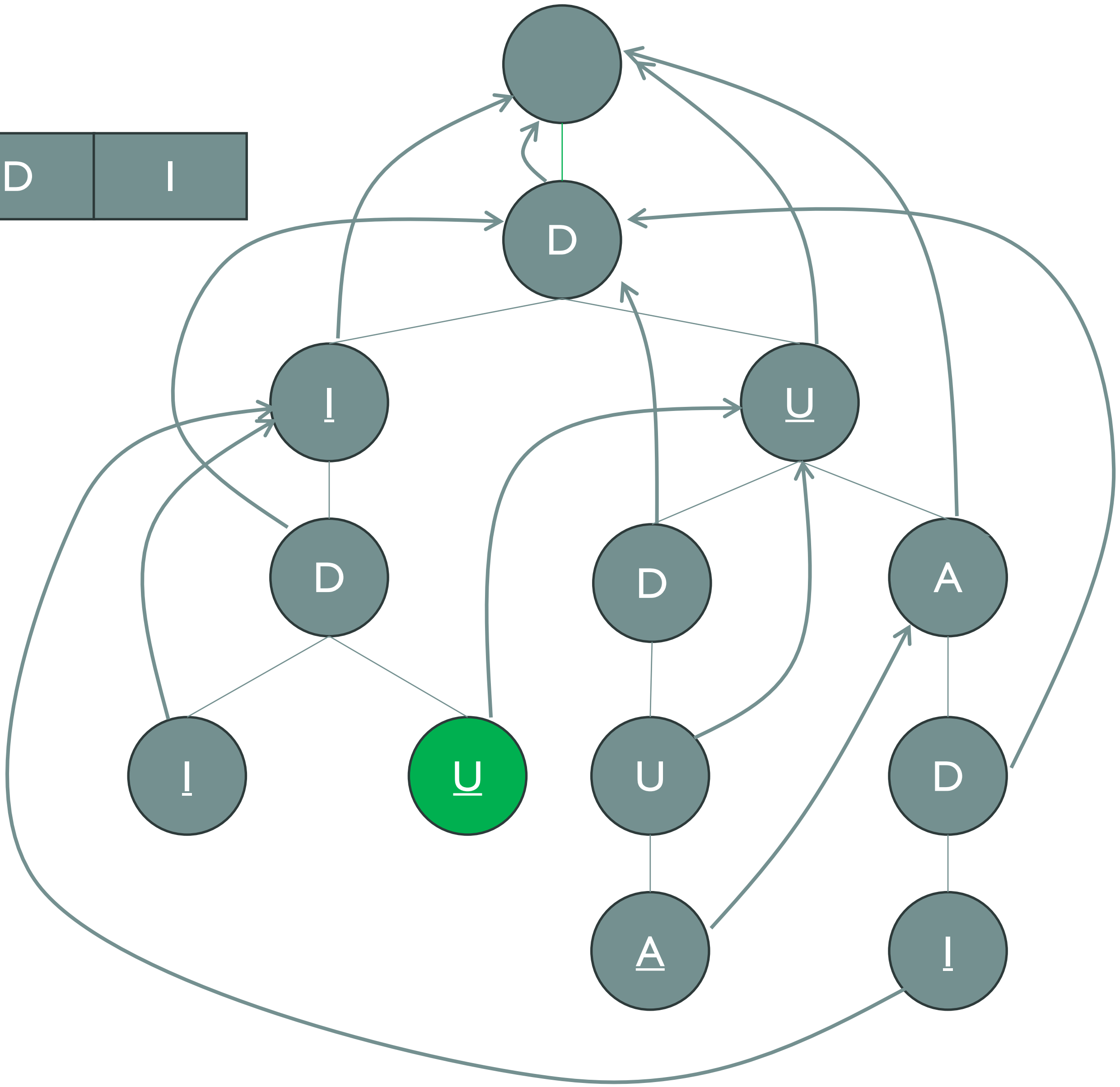
The pattern strings occur in the text:

DI



The text: “DIDUDUADI”

D	I	D	U	D	U	A	D	I



The pattern strings occur in the text:

DI

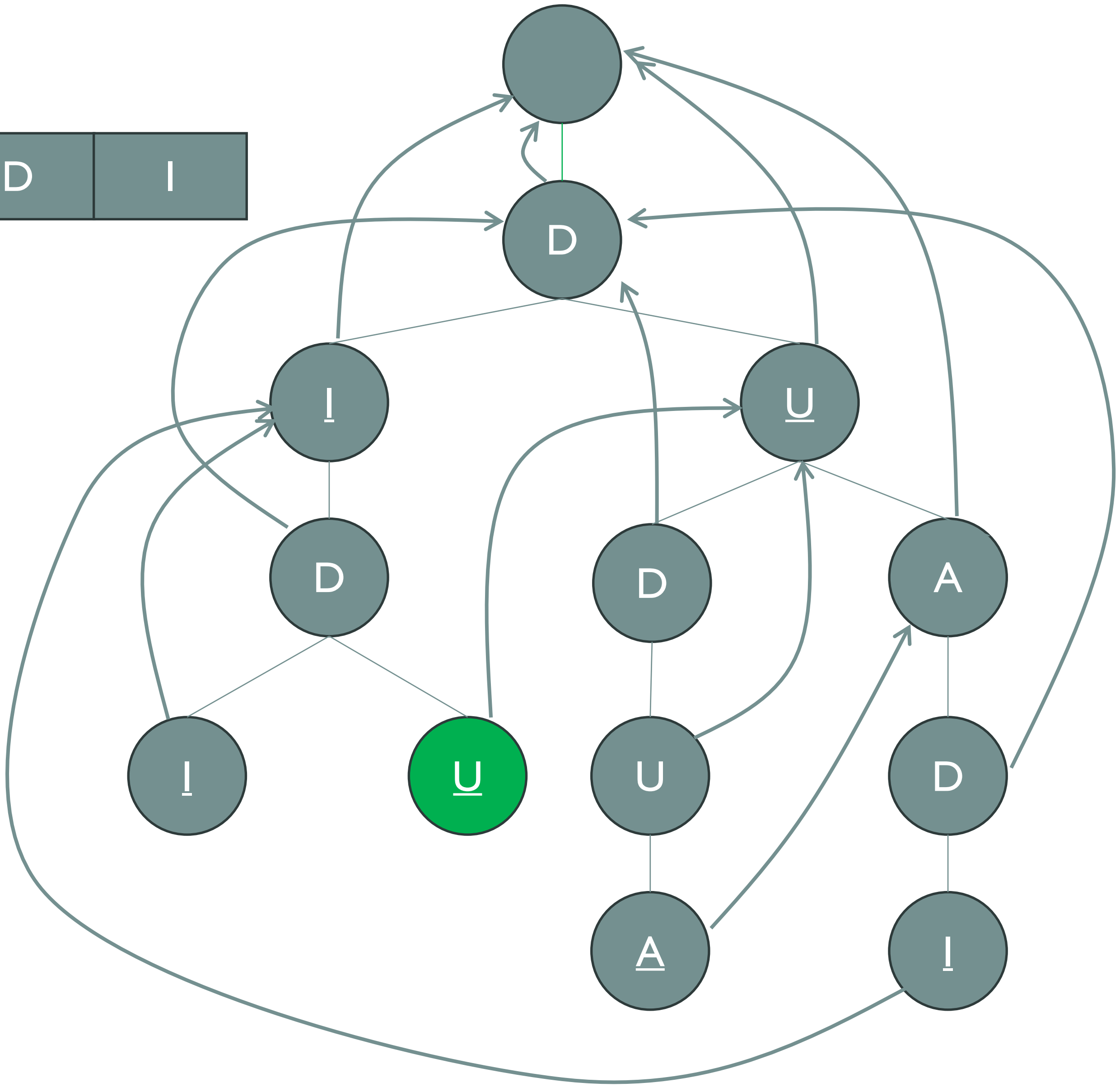
The text: “DIDUDUADI”

D	I	D	U	D	U	A	D	I

The pattern strings occur in the text:

DI

DIDU



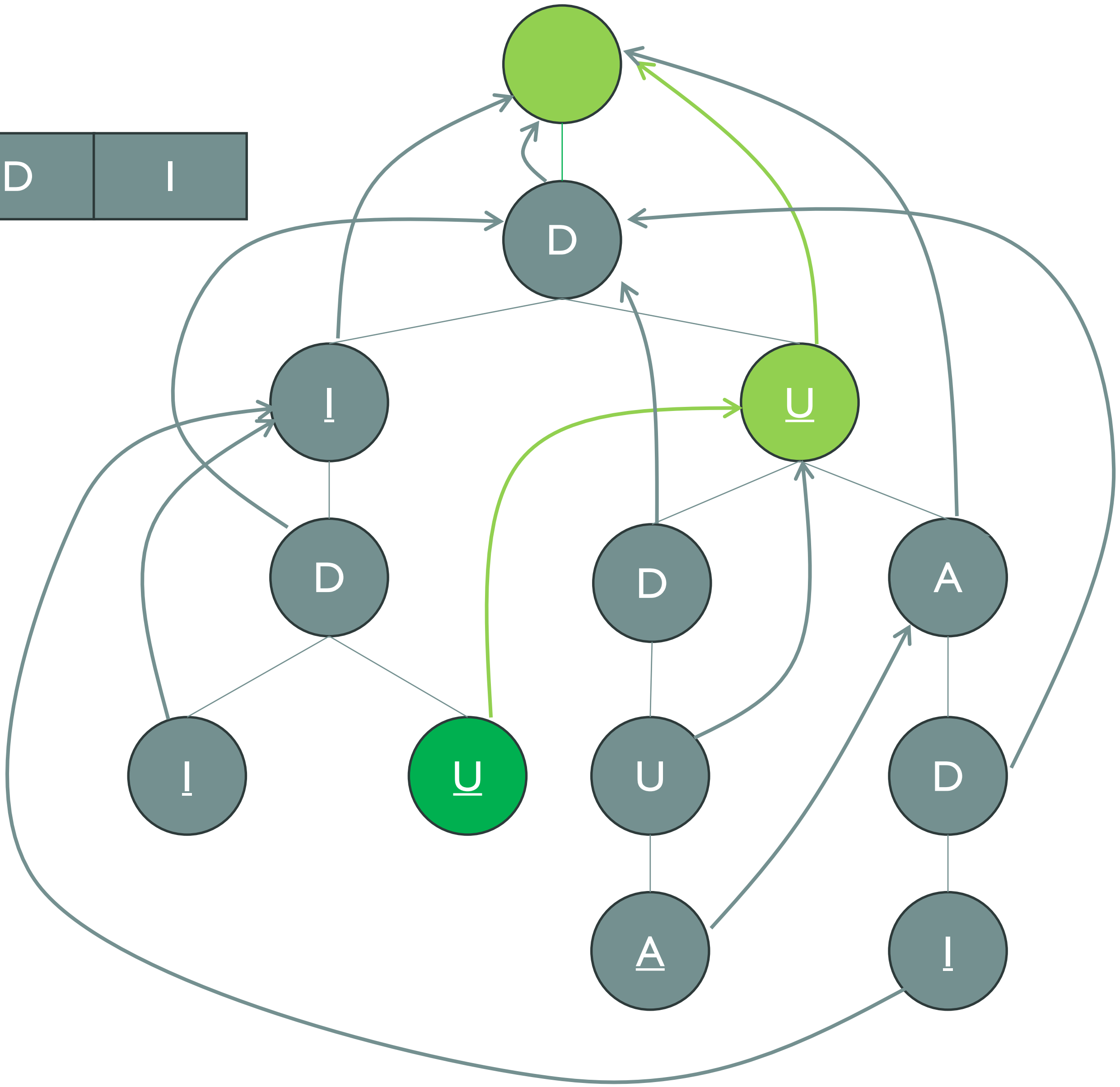
The text: “DIDUDUADI”

D	I	D	U	D	U	A	D	I

The pattern strings occur in the text:

DI

DIDU



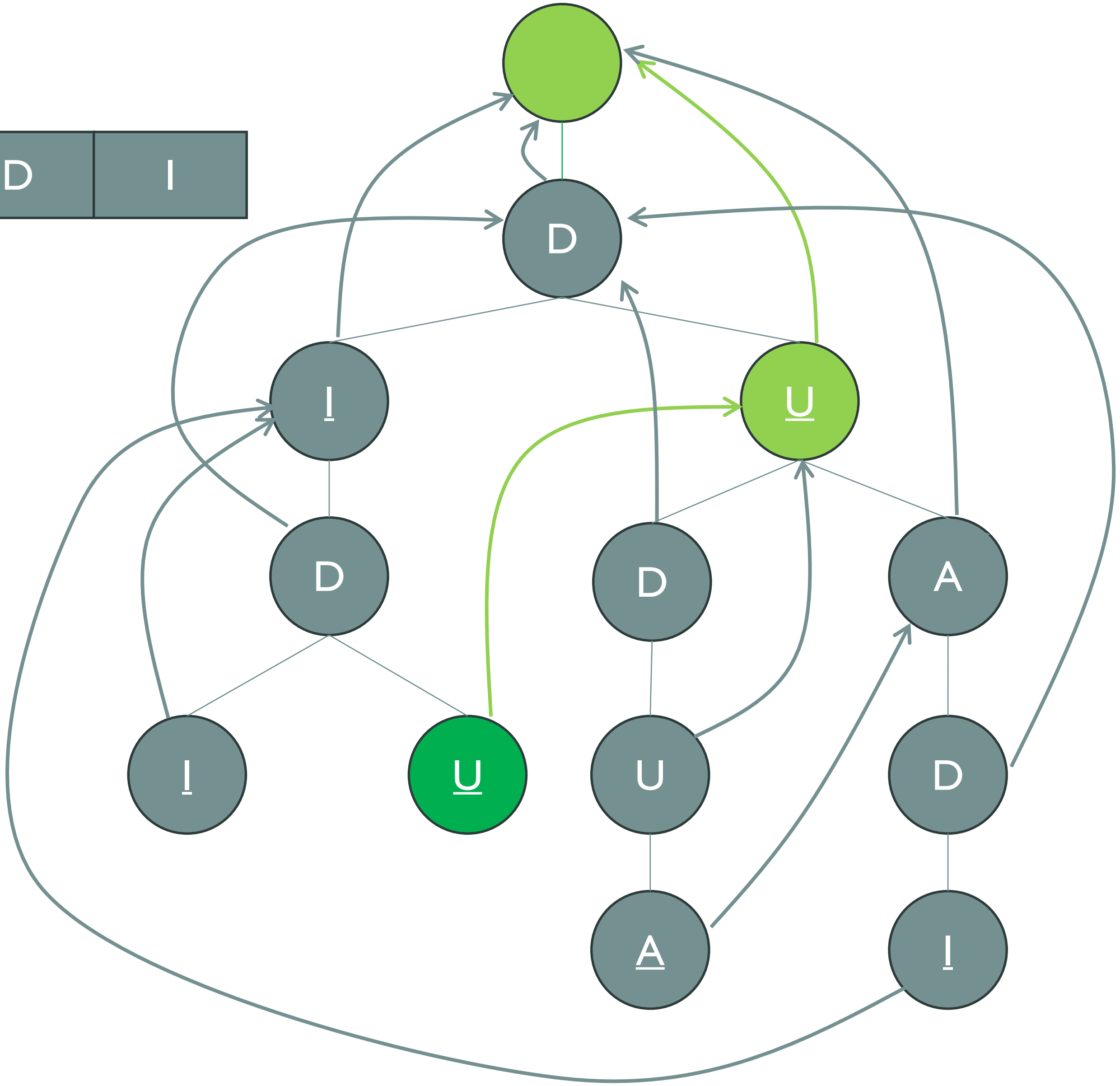
The text: “DIDUDUADI”

The pattern strings occur in the text:

DI

DIDU

DU



The text: “DIDUDUADI”

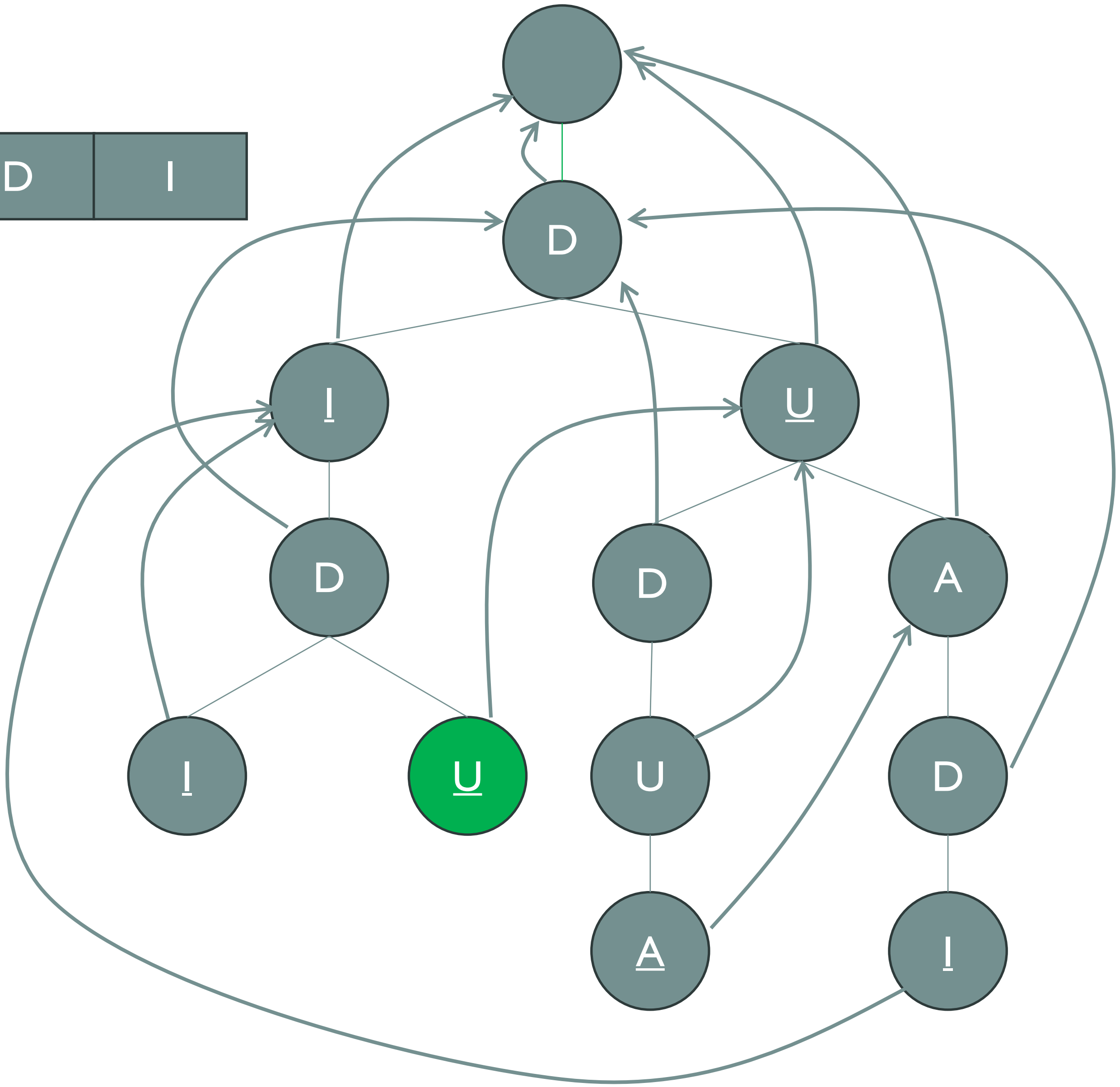
D	I	D	U	D	U	A	D	I

The pattern strings occur in the text:

DI

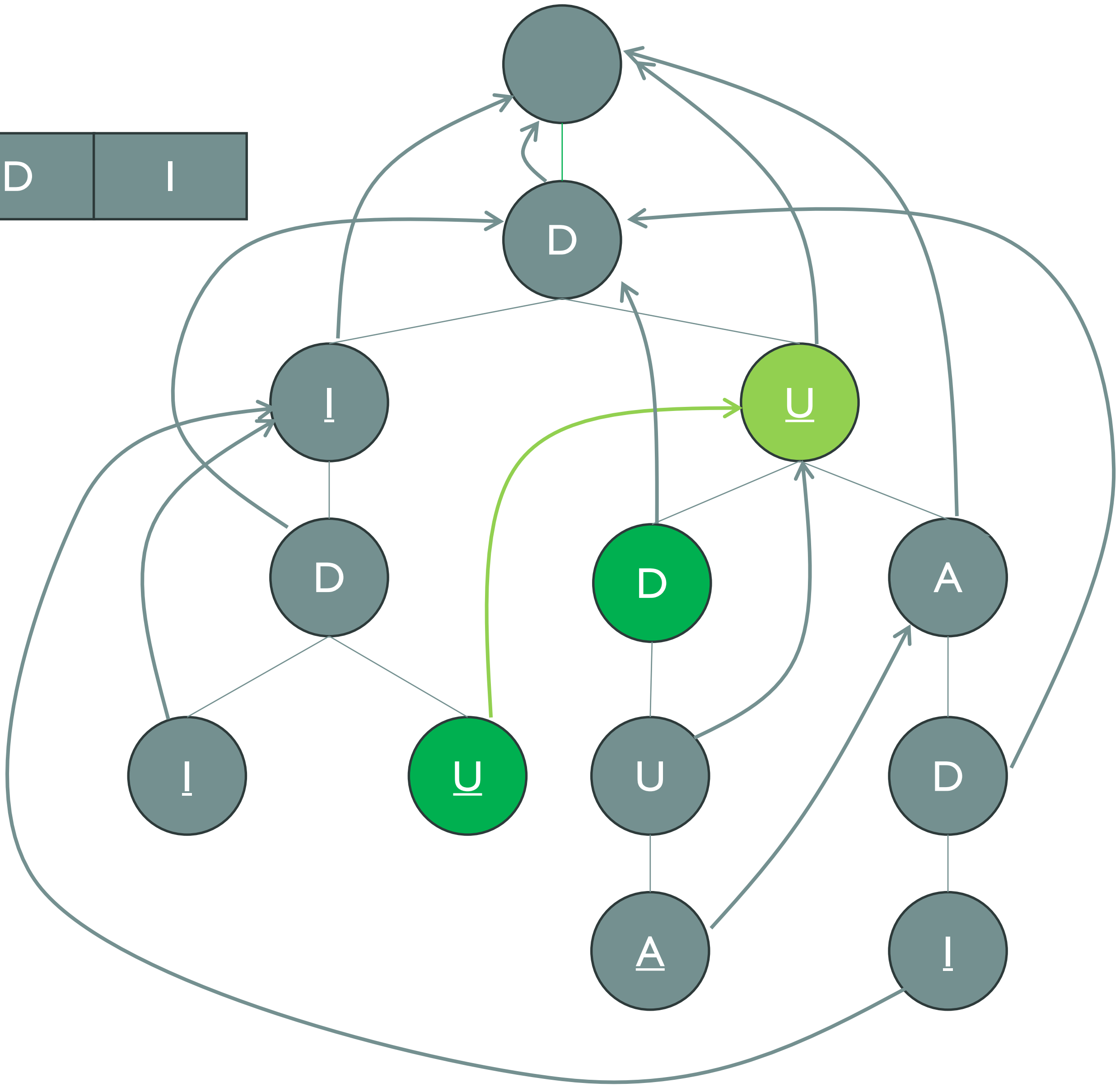
DIDU

DU



The text: "DIDUDUADI"

D	I	D	U	D	U	A	D	I



The pattern strings occur in the text:

DI

DIDU

DU

The text: "DIDUDUADI"

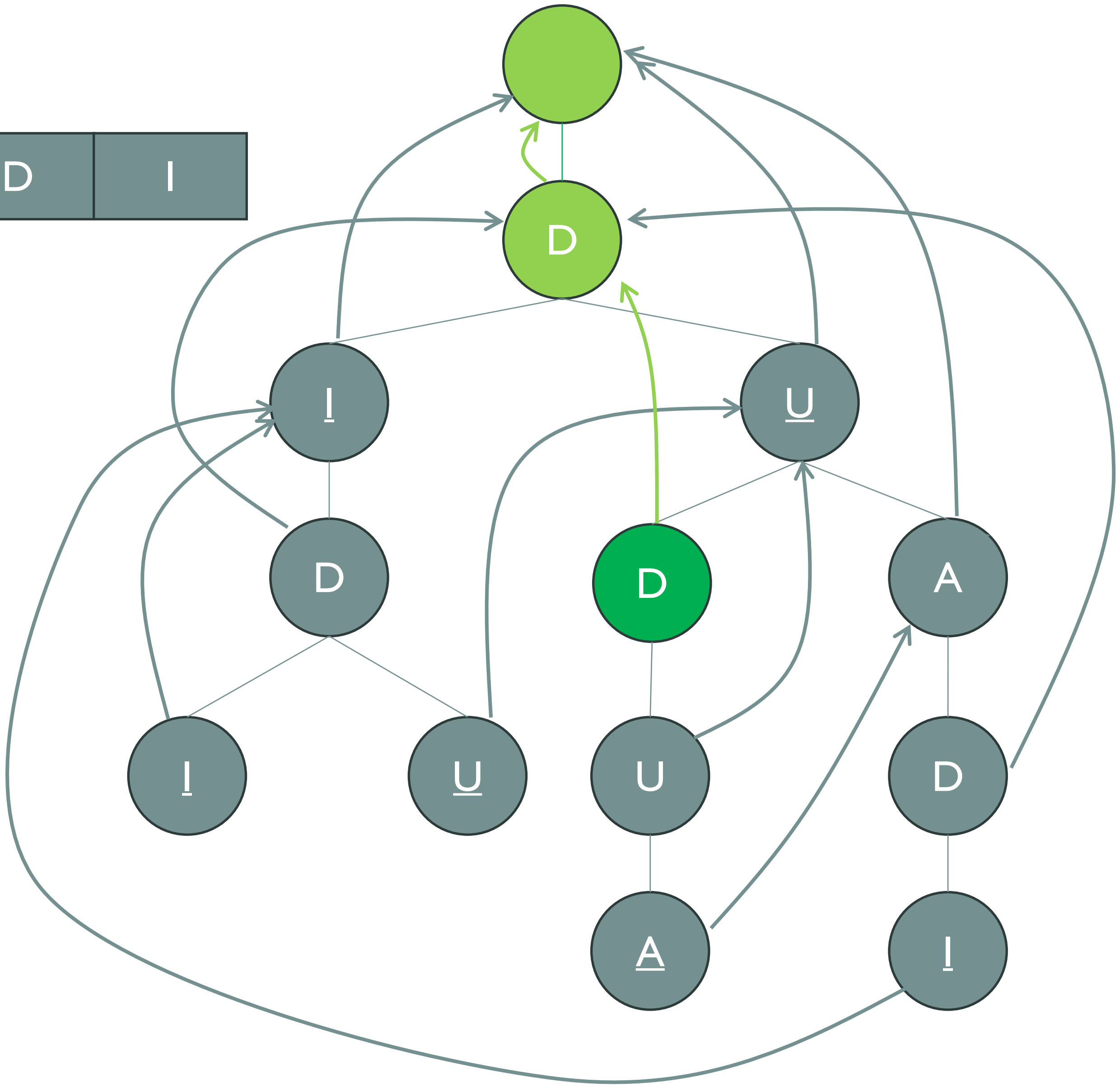
D	I	D	U	D	U	A	D	I

The pattern strings occur in the text:

DI

DIDU

DU



The text: “DIDUDUADI”

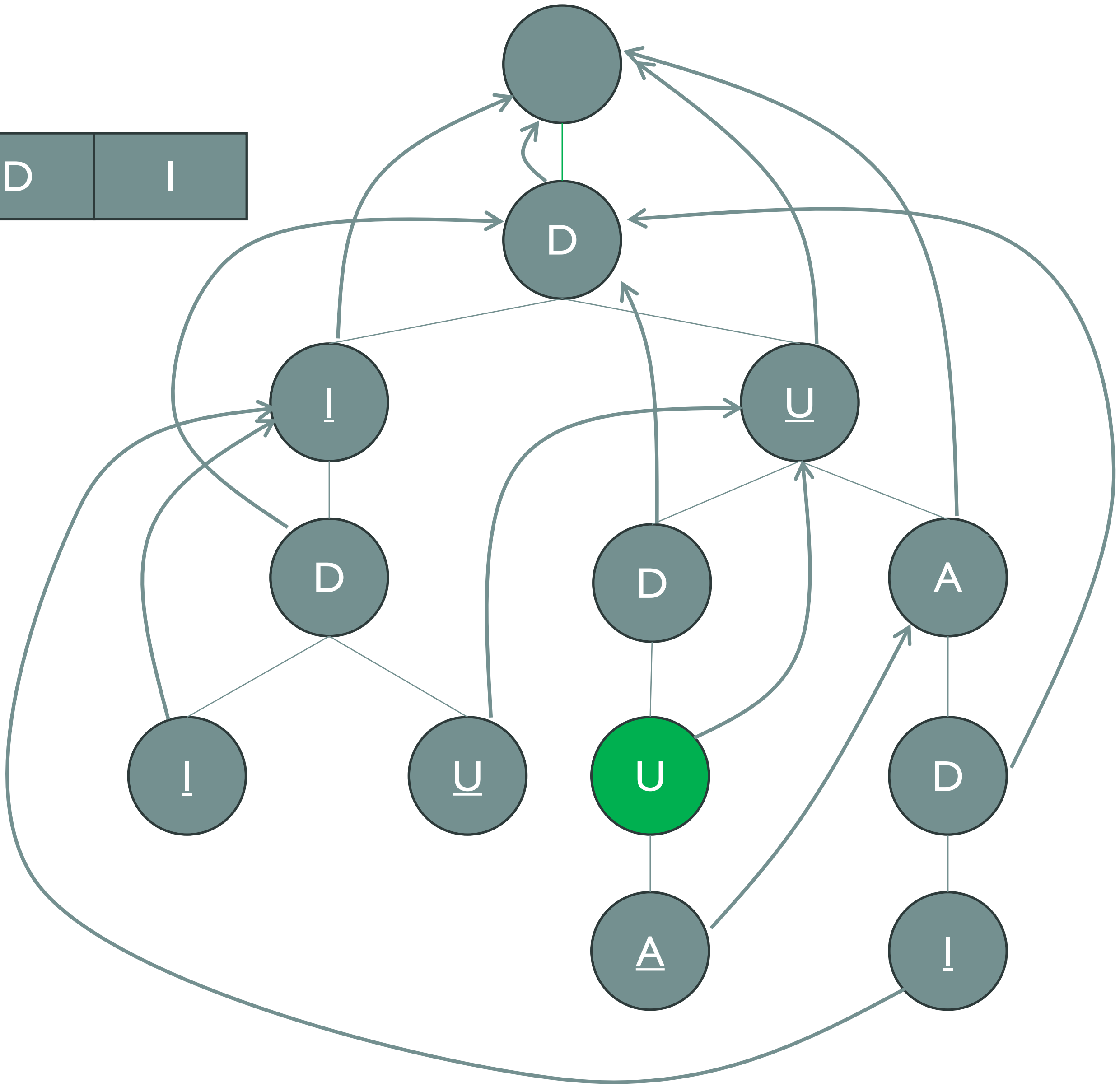
D	I	D	U	D	U	A	D	I

The pattern strings occur in the text:

DI

DIDU

DU



The text: "DIDUDUADI"

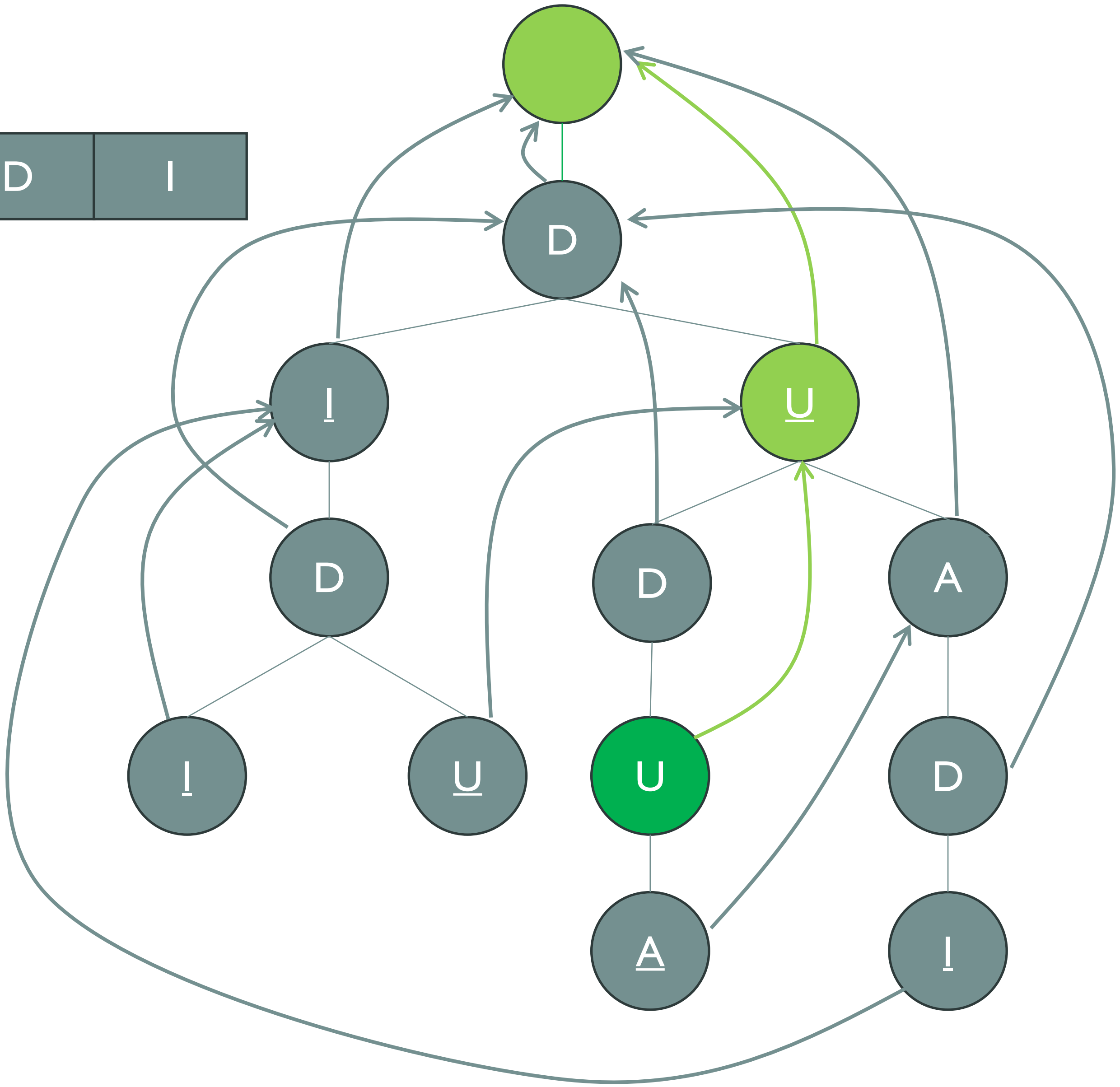
D	I	D	U	D	U	A	D	I

The pattern strings occur in the text:

DI

DIDU

DU

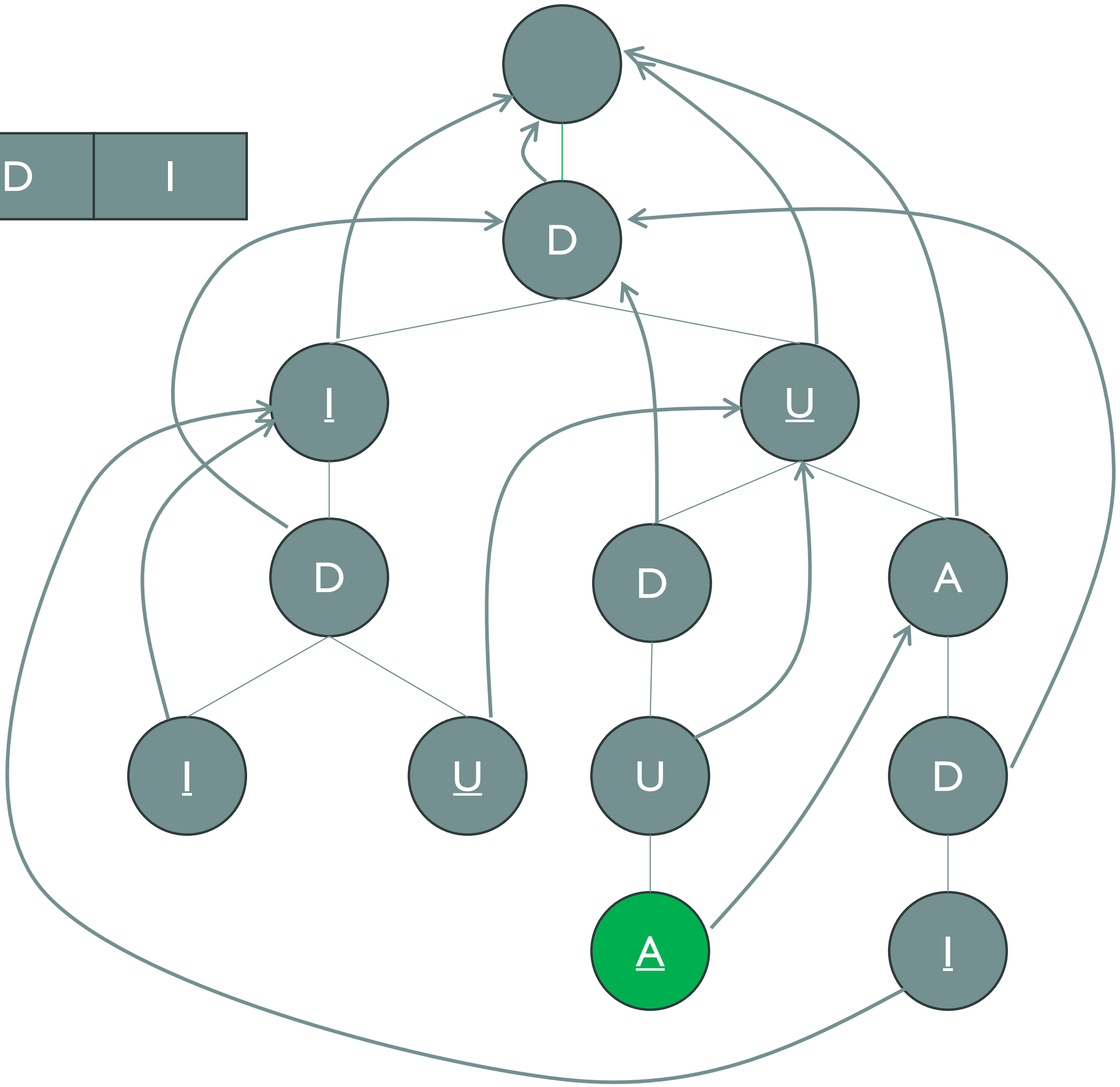


The text: “DIDUDUADI”

D	I	D	U	D	U	A	D	I

The pattern strings occur in the text:

- DI
- DIDU
- DU
- DUDUA



The text: “DIDUDUADI”

D	I	D	U	D	U	A	D	I

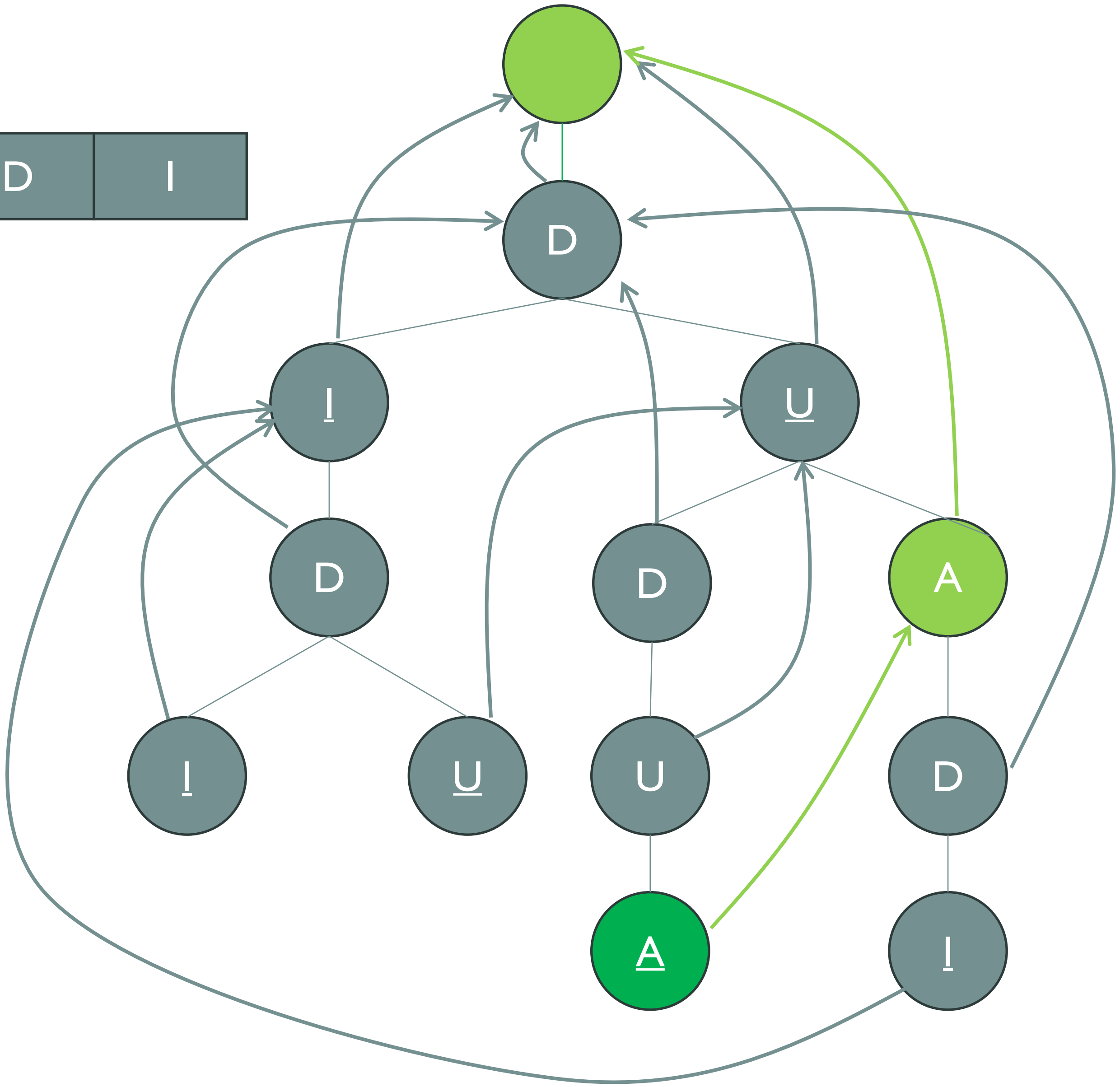
The pattern strings occur in the text:

DI

DIDU

DU

DUDUA

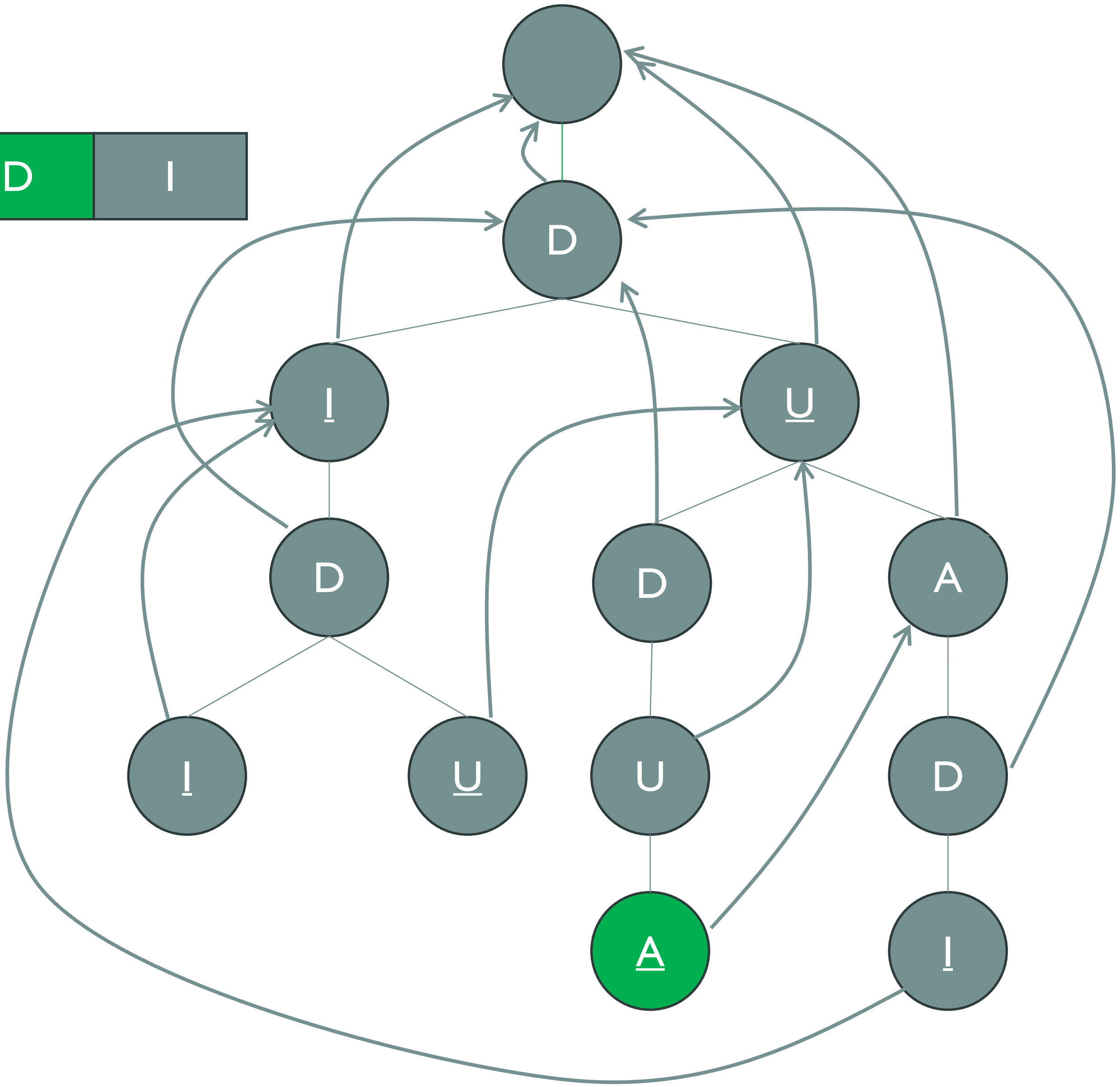


The text: “DIDUDUADI”

D	I	D	U	D	U	A	D	I

The pattern strings occur in the text:

- DI
- DIDU
- DU
- DUDUA

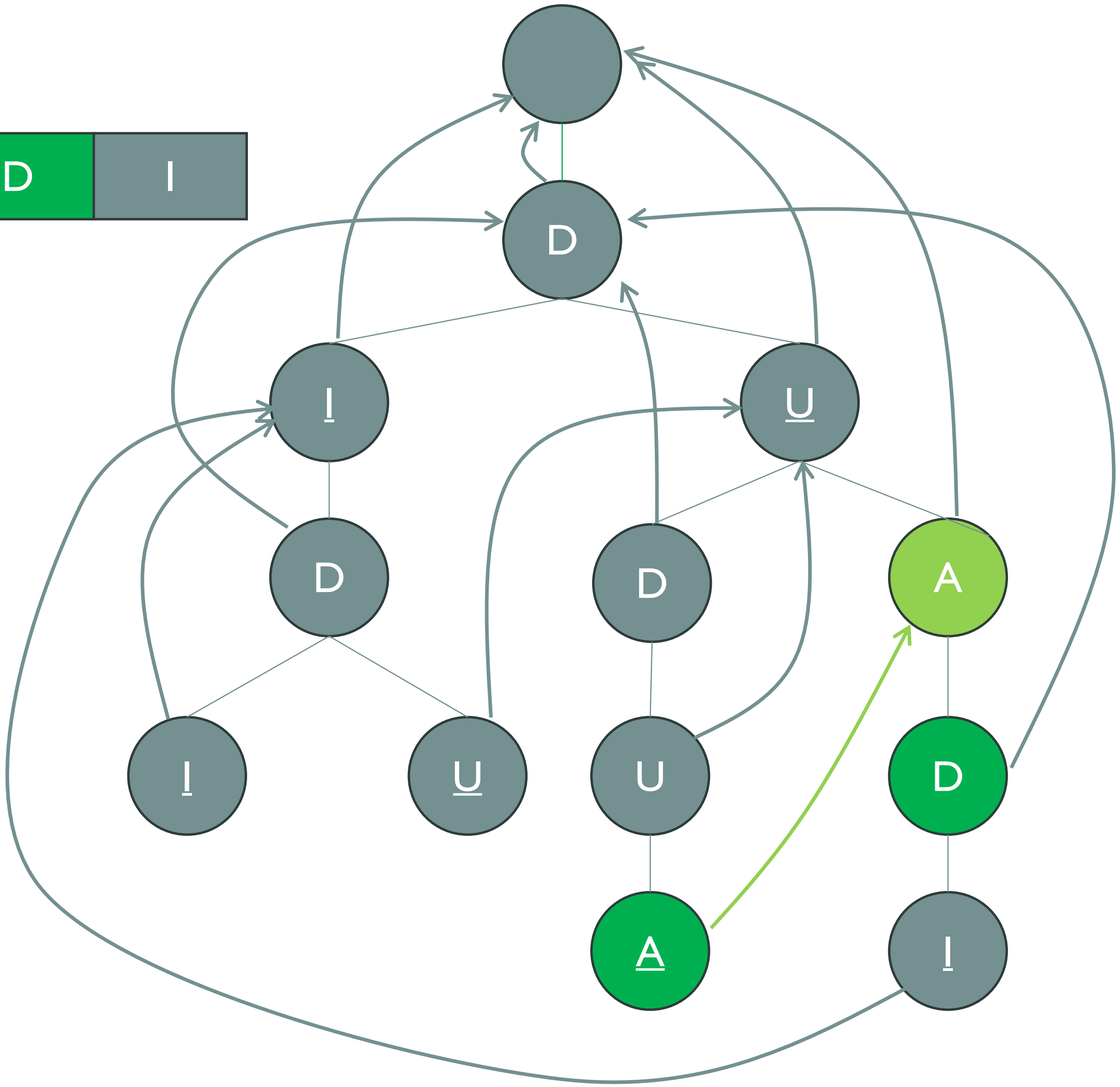


The text: “DIDUDUADI”

D	I	D	U	D	U	A	D	I

The pattern strings occur in the text:

- DI
- DIDU
- DU
- DUDUA



The text: “DIDUDUADI”

D	I	D	U	D	U	A	D	I

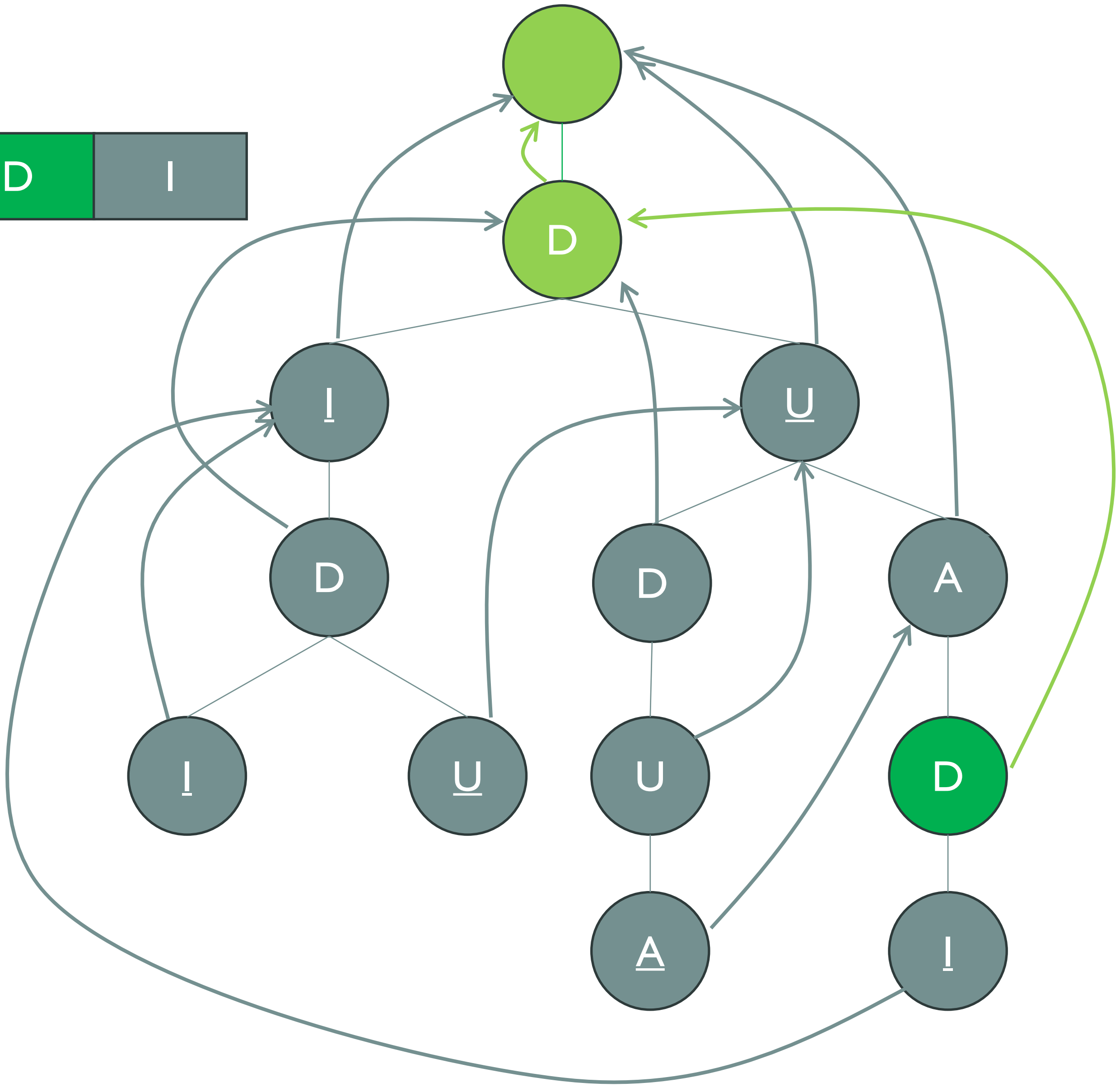
The pattern strings occur in the text:

DI

DIDU

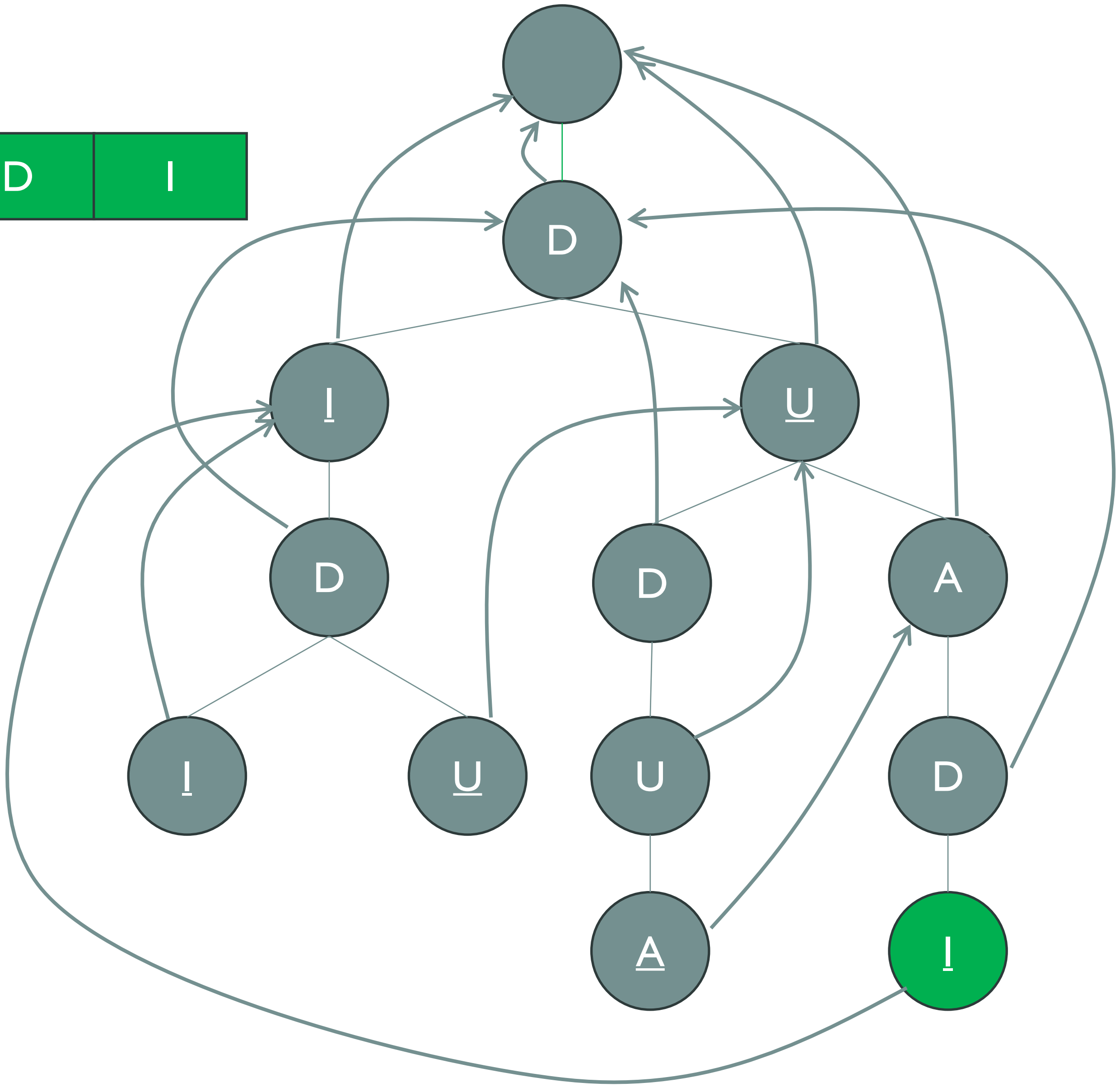
DU

DUDUA



The text: “DIDUDUADI”

D	I	D	U	D	U	A	D	I



The pattern strings occur in the text:

DI

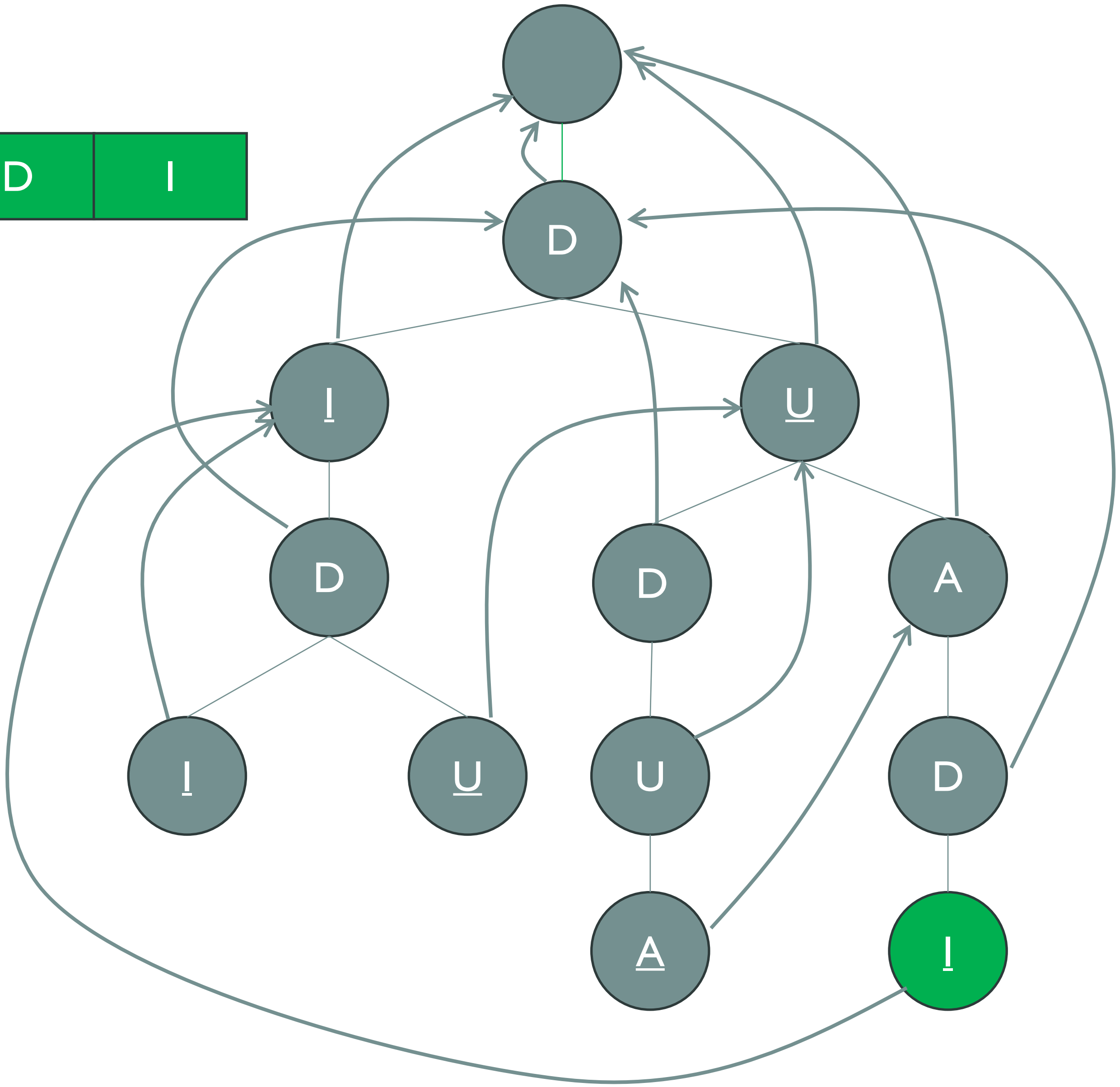
DIDU

DU

DUDUA

The text: “DIDUDUADI”

D	I	D	U	D	U	A	D	I



The pattern strings occur in the text:

- DI
- DIDU
- DU
- DUDUA
- DUADI

The text: “DIDUDUADI”

D	I	D	U	D	U	A	D	I

The pattern strings occur in the text:

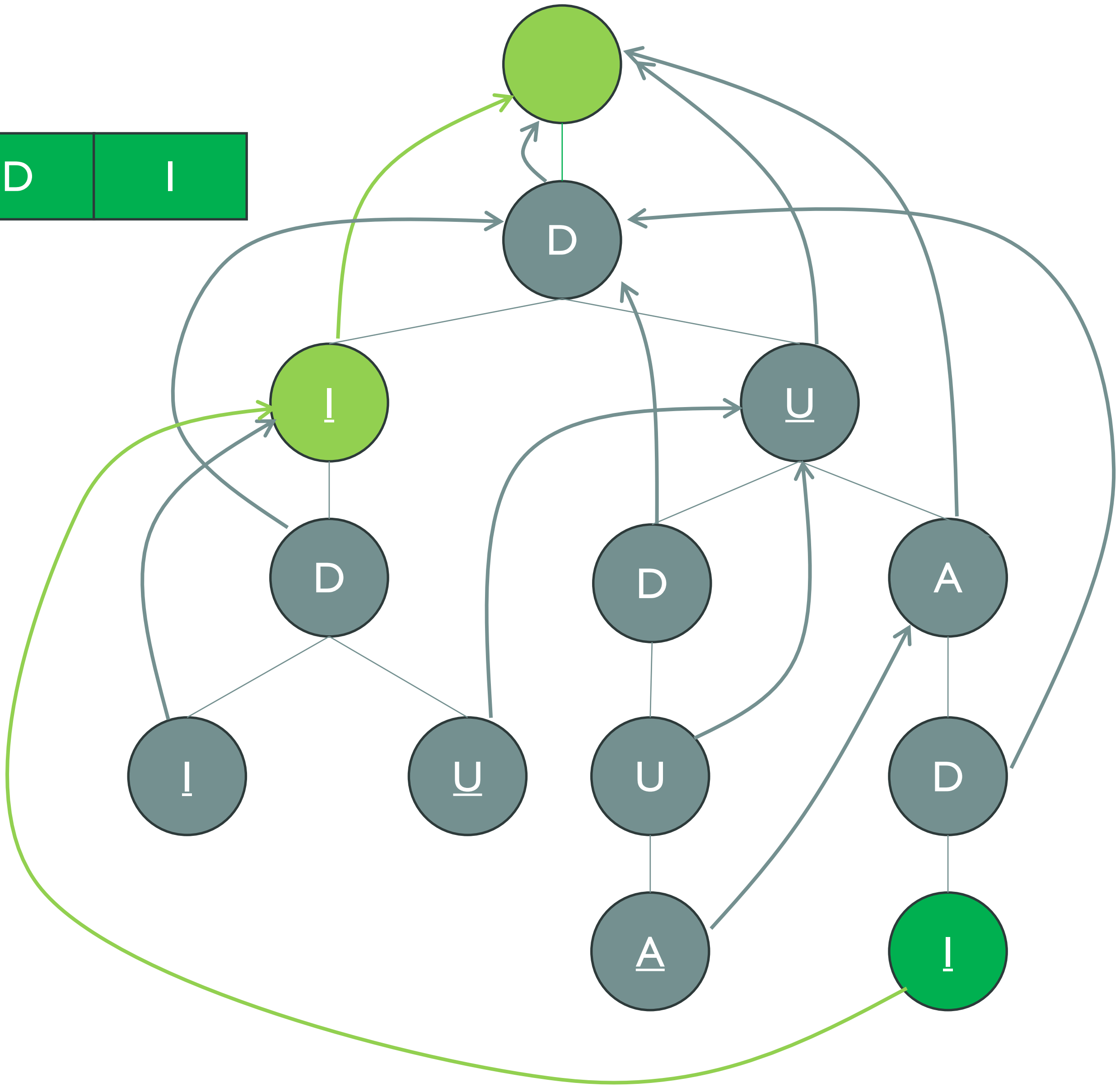
DI

DIDU

DU

DUDUA

DUADI



The text: “DIDUDUADI”

D	I	D	U	D	U	A	D	I

The pattern strings occur in the text:

DI

DIDU

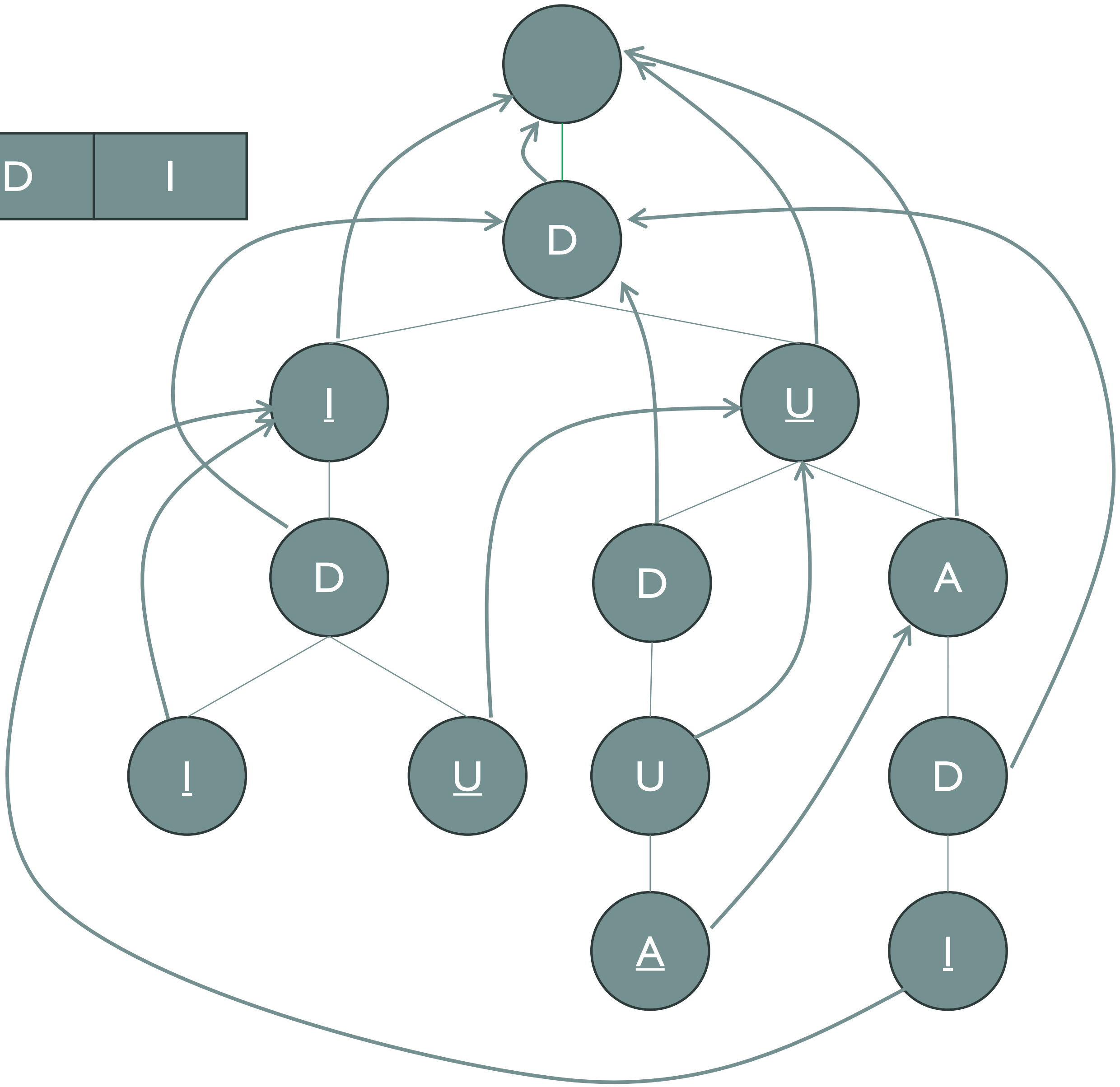
DU

DUDUA

DUADI

Time Complexity? $O(n + z)$

Space Complexity? $\Theta(M)$



SIMILARITIES AND DISSIMILARITIES BETWEEN KMP AND AHO-CORASICK

SIMILARITIES

1. Preprocessing for Efficient Searching: Both algorithms preprocess the pattern(s) beforesearching.

2. Failure Links for Efficient Backtracking: Both use suffix links to avoid unnecessary comparisons (KMP uses the Longest Prefix Suffix (LPS) array, while AC uses suffix links in a trie).

3. Linear Time Complexity in Searching: Both achieve an efficient search time of O(n), where n is the length of the text.

4. Pattern Matching Without Backtracking: Neither algorithm requires explicit backtracking in the text; instead, they use precomputed failure information to move efficiently.

DISSIMILARITIES

Feature	Aho-Corasick	Knuth-Morris-Pratt (KMP)
Purpose	Searches for multiple patterns simultaneously	Searches for a single pattern
Preprocessing Complexity	O(M), where M is the total length of all patterns	O(m), where m is the length of a pattern
Search Complexity	O(n + z), where n is the length of the text and z is the number of matches	O(n), where n is the length of the text
Data Structure Used	Trie with failure(or suffix) links	LPS (Longest Prefix Suffix) array
Space Complexity	O(M + s) in the worst case, where s is the size of alphabet	O(m)
Pattern Matching Approach	Simultaneously matches multiple patterns	Sequentially matches in a single pattern
Common use cases	Spam filtering, intrusion detection, dictionary matching	Substring search in text editors, string matching problems

THANK YOU