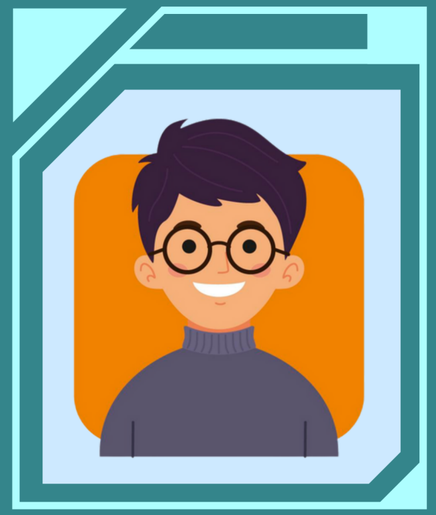
TRIES

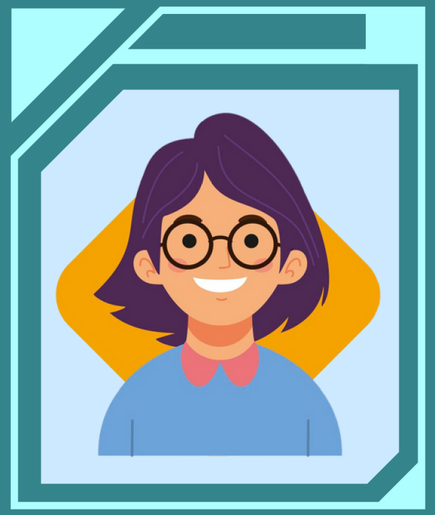
A SPECIAL TREE

Meet Our Team

Ryan



Angelo



Iza

Jay

Table of Contents

Introduction

- The idea of a trie for representing a set of strings was first abstractly described by Axel Thue in 1912.
- Tries were first described in a computer context by René de la Briandais in 1959.
- The idea was independently described in 1960 by Edward Fredkin, who coined the term trie, pronouncing it /ˈtriː/ (as "tree"), after the middle syllable of retrieval. However, other authors pronounce it /ˈtraɪ/ (as "try"), in an attempt to distinguish it verbally from "tree".

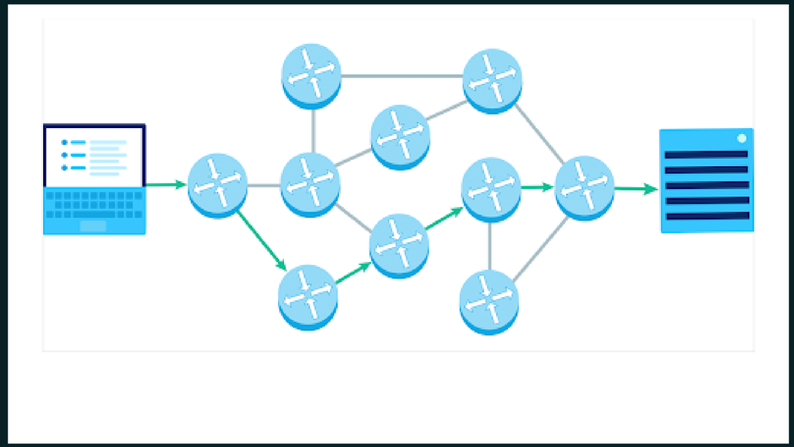
Introduction

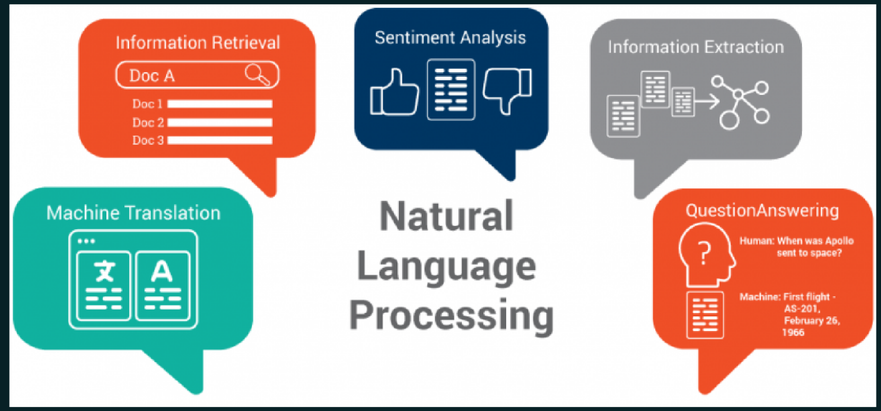
- **Tries** (also known as radix trees or prefix trees) are tree-based data structures that are typically used to store associative arrays where the keys are usually strings. It is totally dependent on the contents of the tree. *(Brilliant.org)*

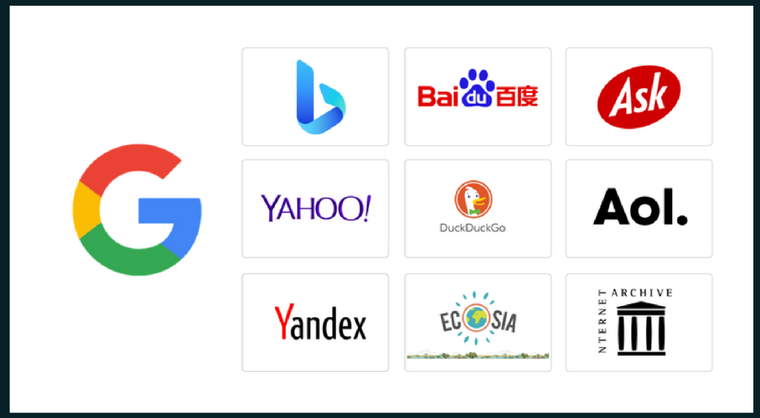
- It can be visualized as a graph consisting of nodes and edges. Another name for Trie is the digital tree. Each node of a trie can have as many as 26 pointers/references. These 26 pointers represent the 26 characters of the English language. *(geeksforgeeks.org)*

REFERENCES: https://brilliant.org/wiki/tries/ | https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-trie/

Applications







Time and Space Complexity

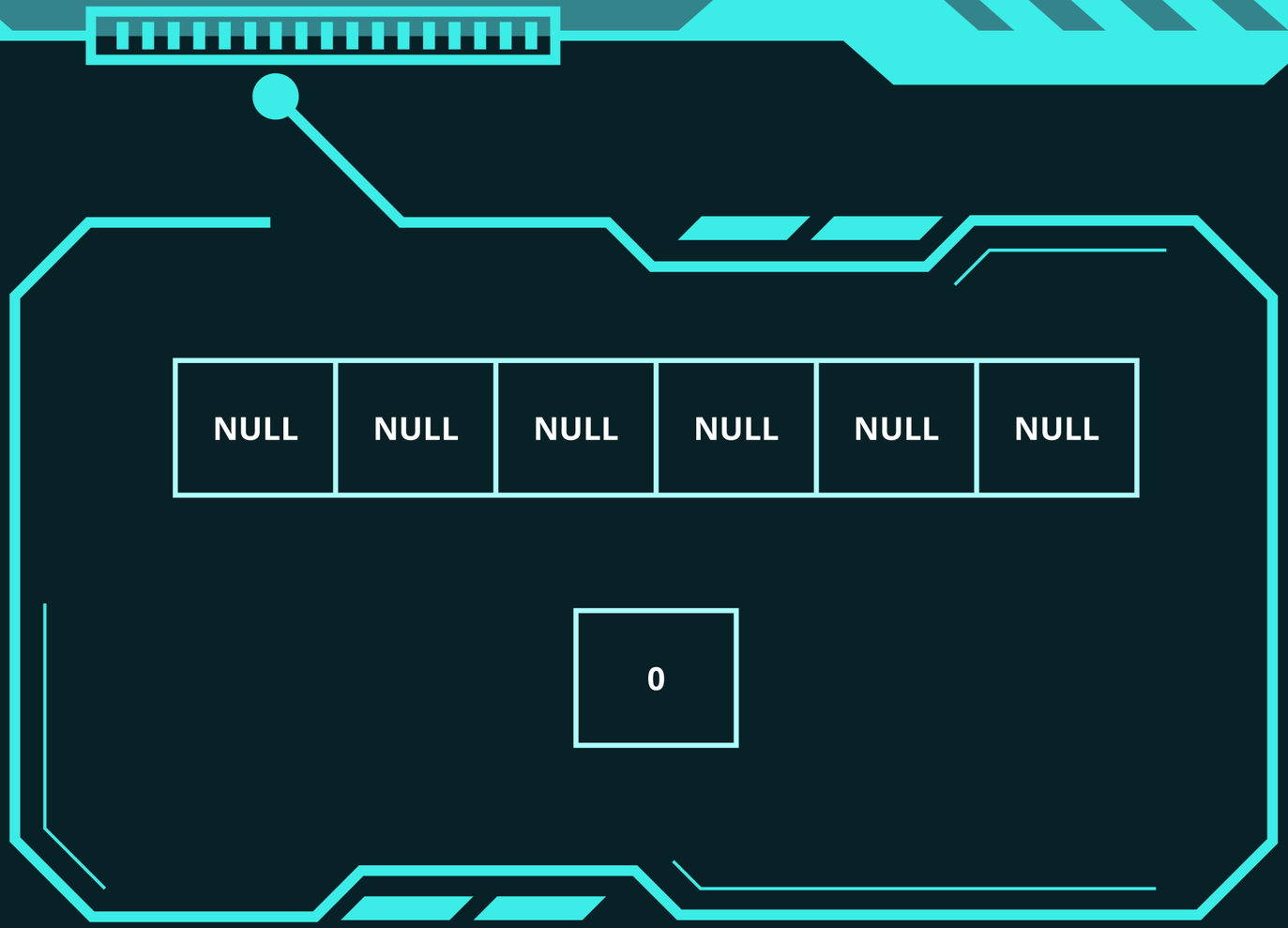
Time Complexity:
O(L)

Space Complexity:
O(1)

Where: L = Length of string

Data Structure

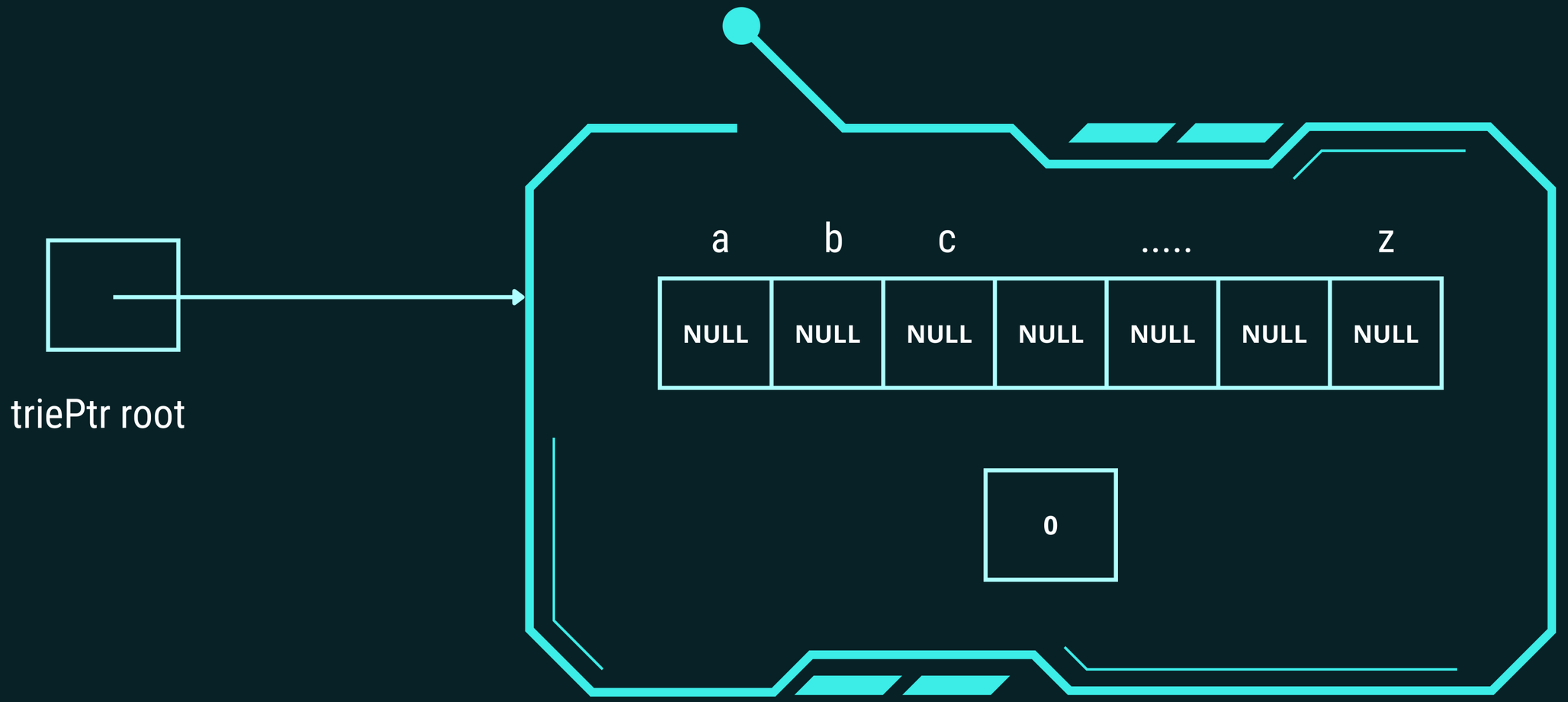
```
typedef struct trieNode{
    struct trieNode *children[CHILDREN_SIZE];
    int isEndofWord; //0 if not end and 1 if end
}*triePtr;
```



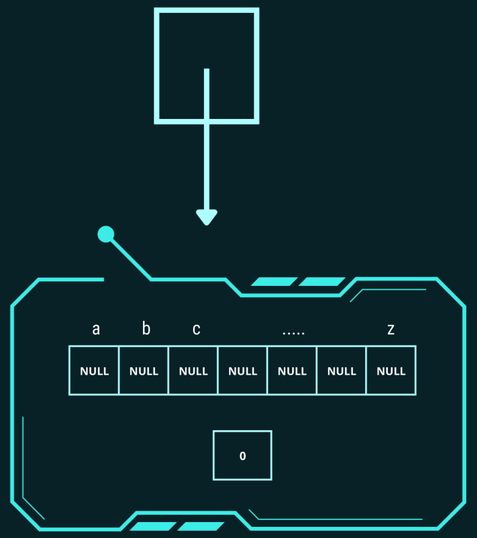
Initialize a Trie Node

triePtr root

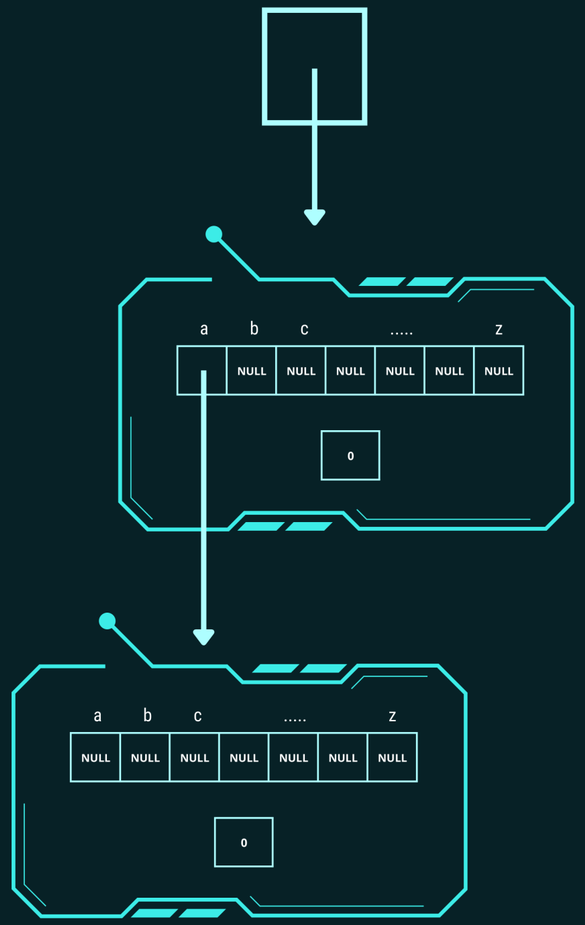
Initialize a Trie Node



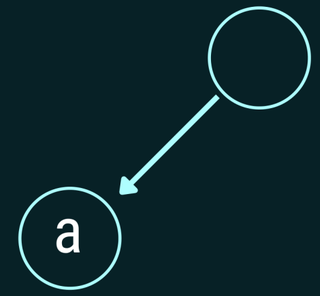
Inserting a Word



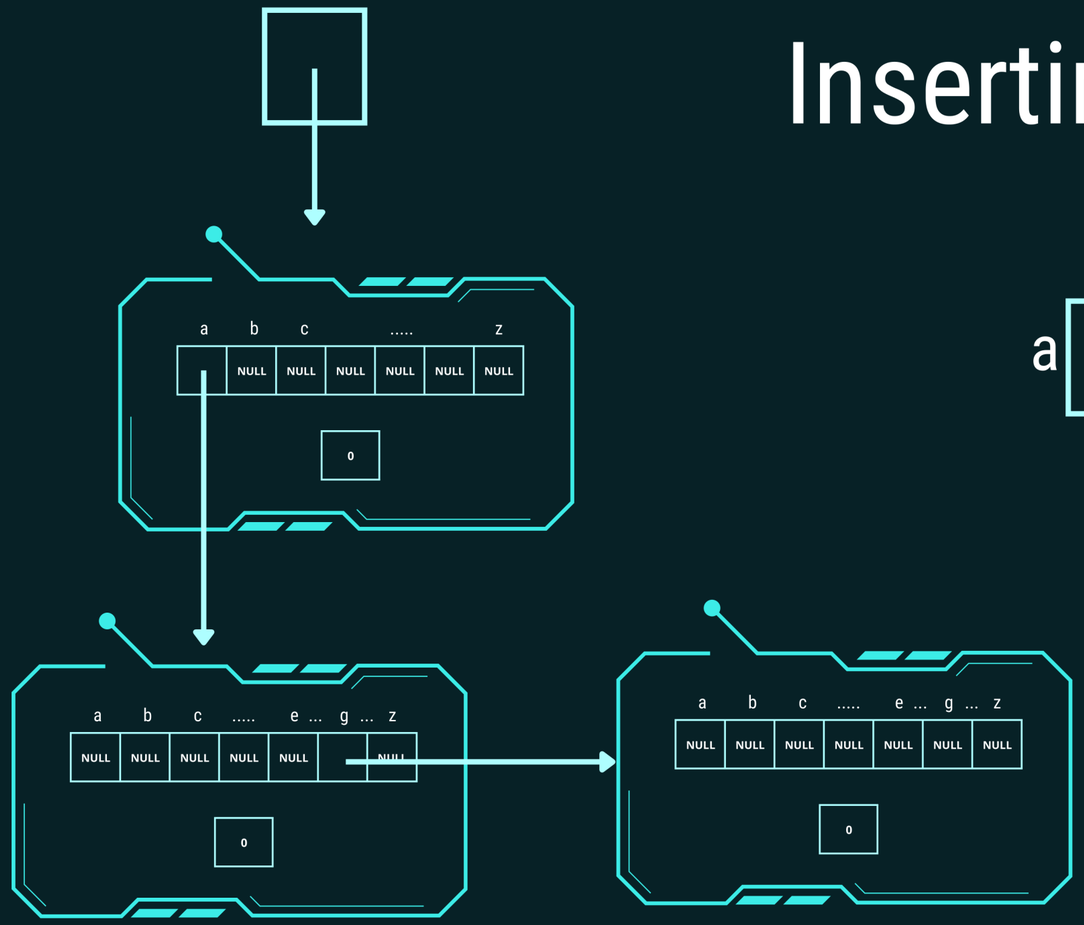
Inserting a Word

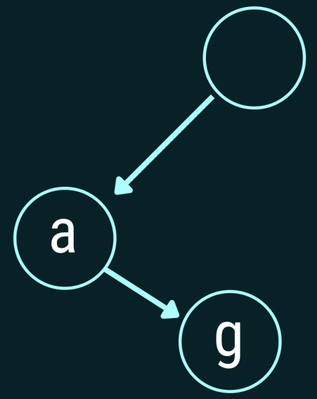


a g e

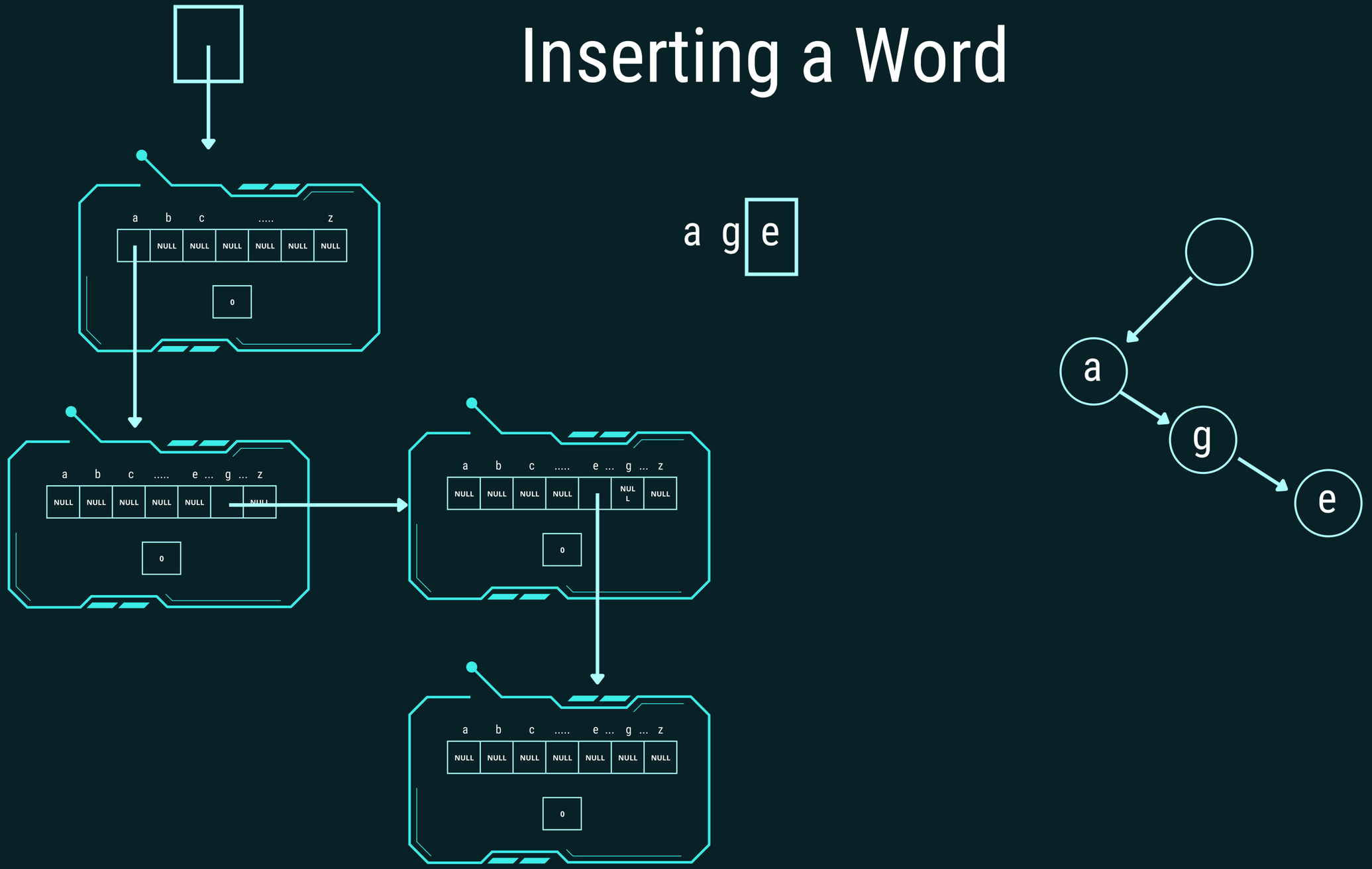


Inserting a Word

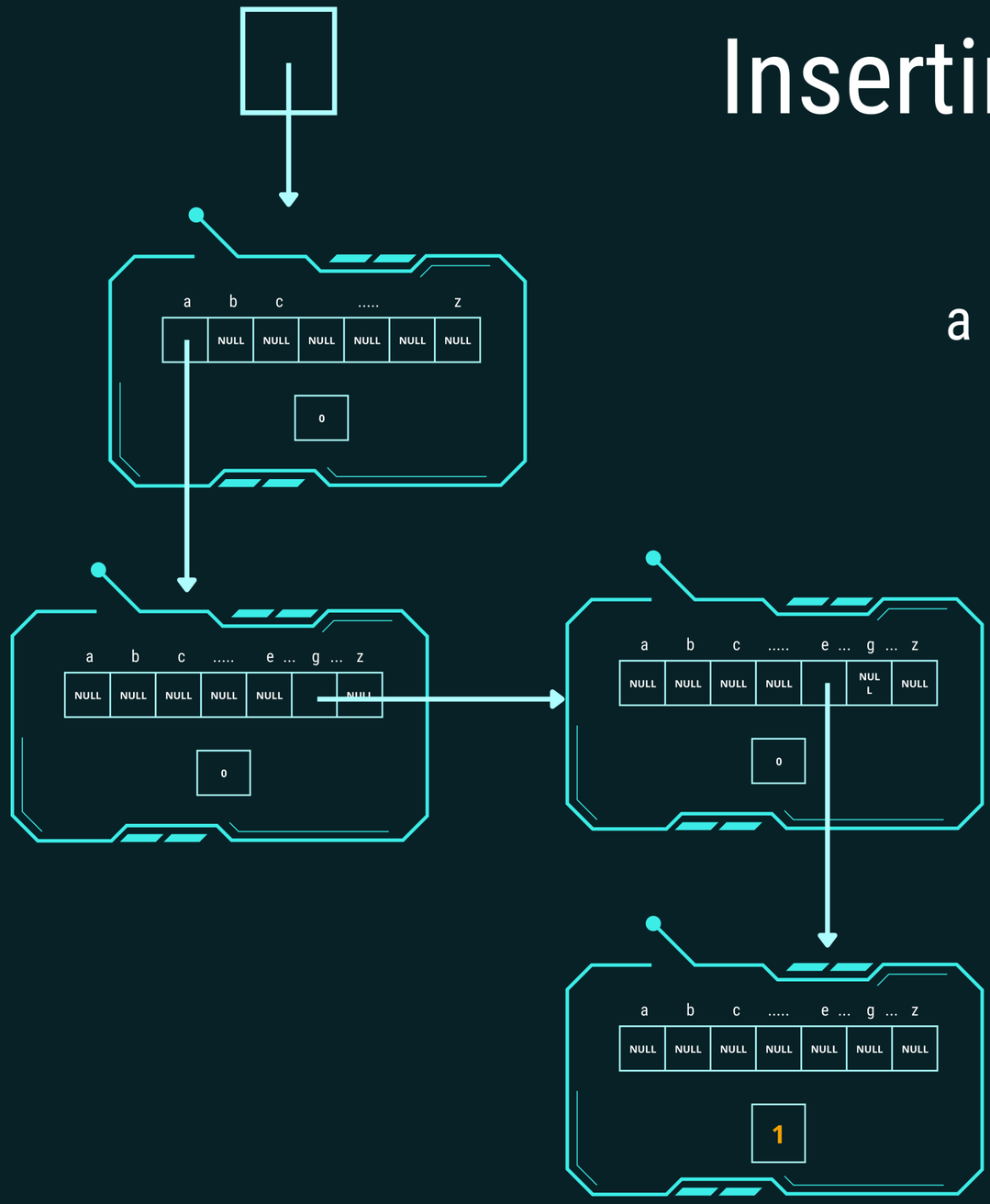




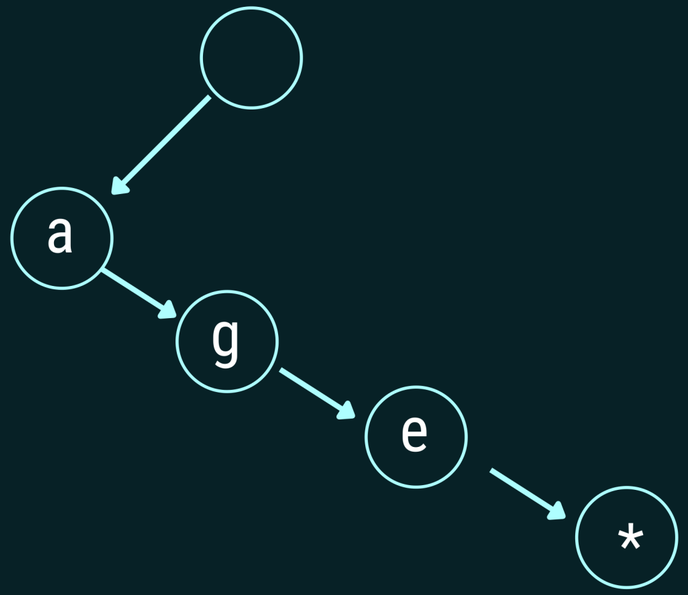
Inserting a Word



Inserting a Word

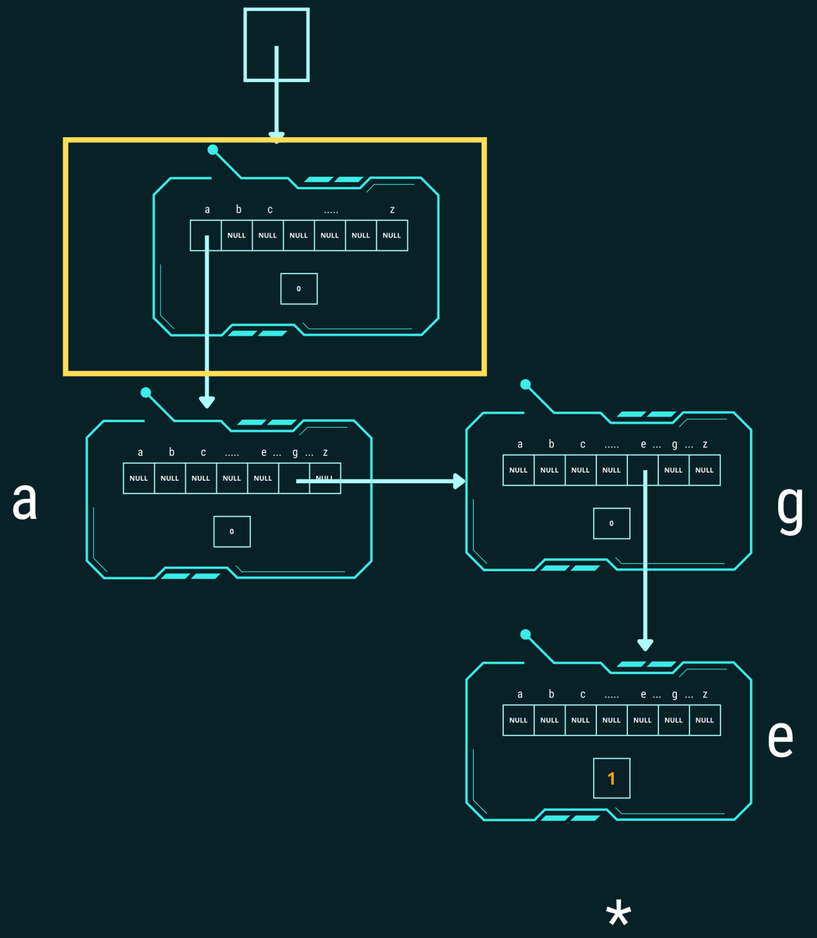


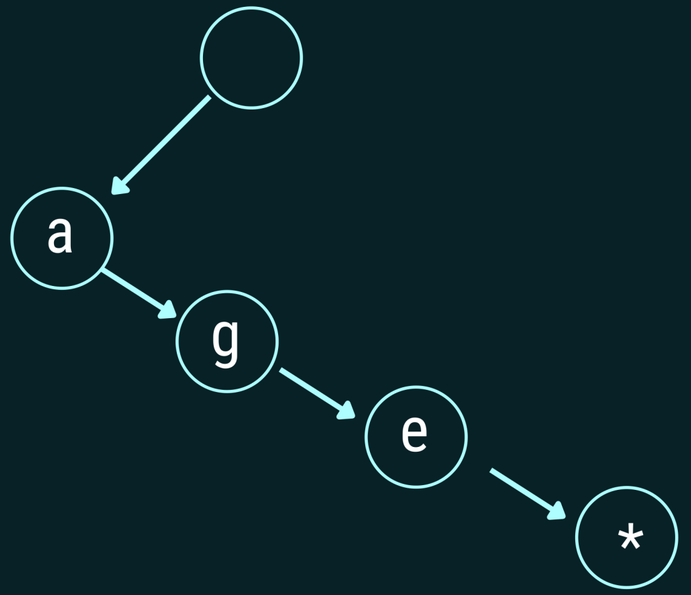
a g e



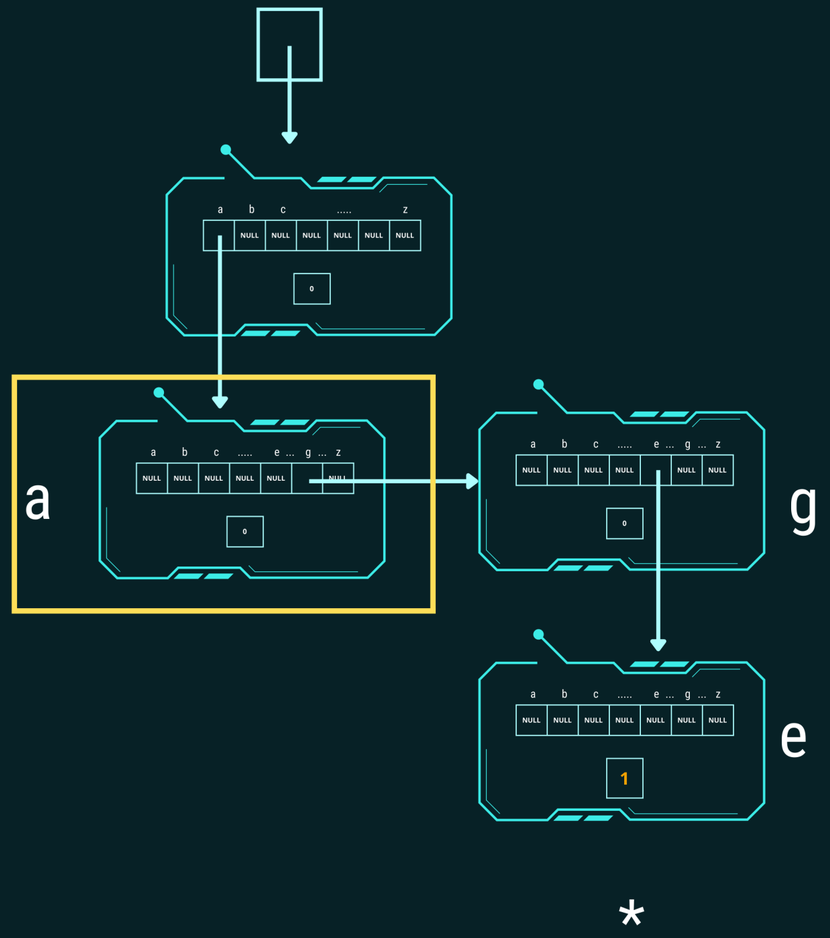
Inserting a Word

a g e d

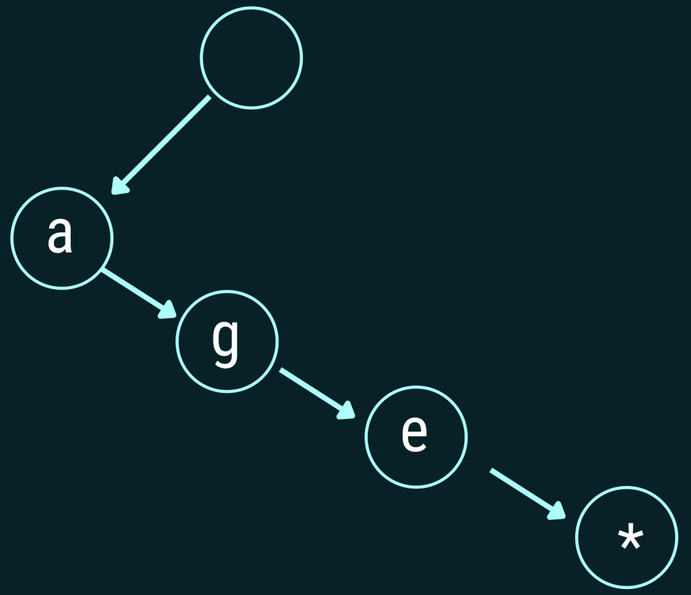




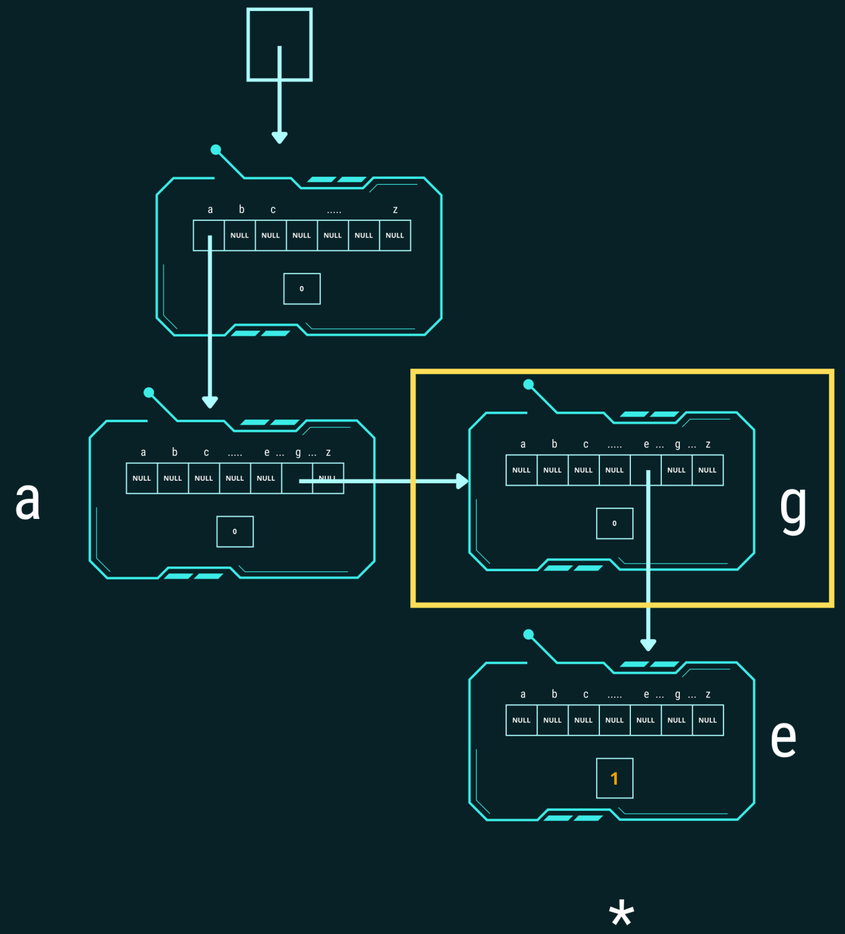
Inserting a Word



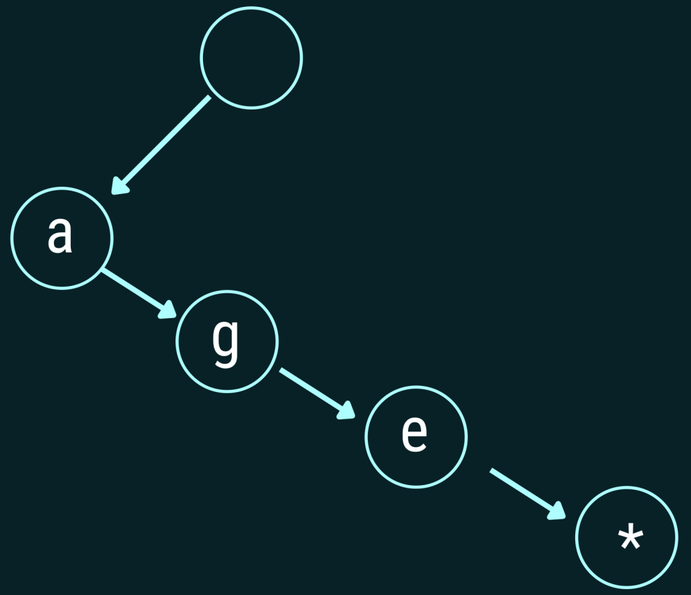
a g e d



Inserting a Word

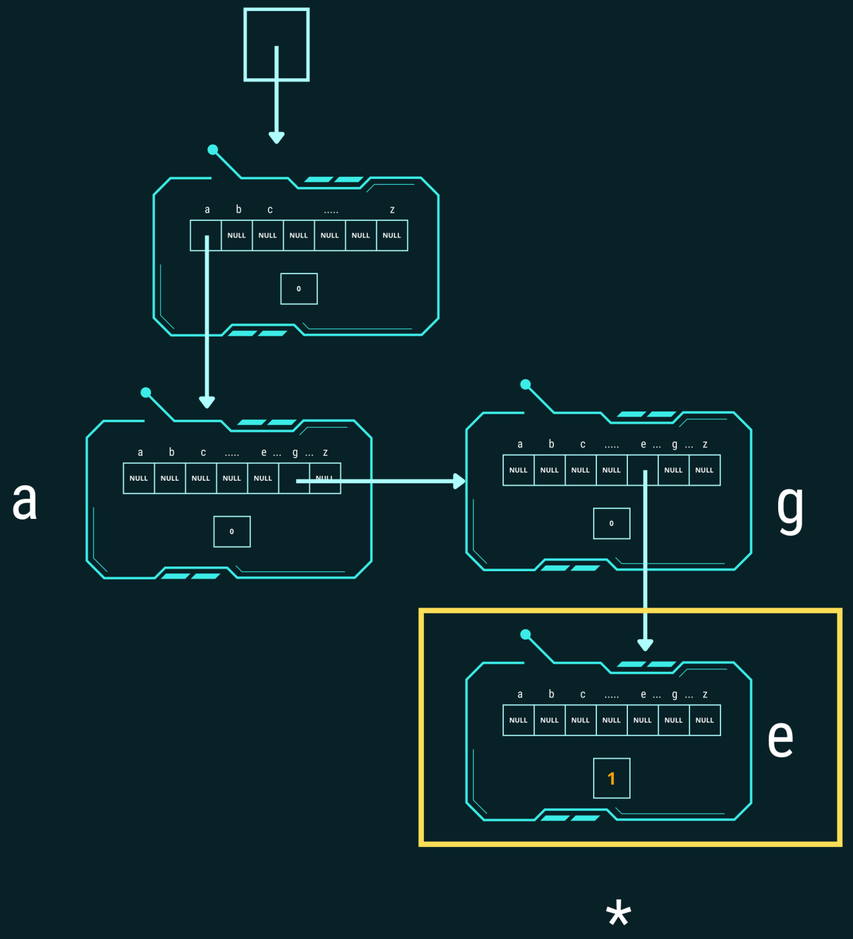


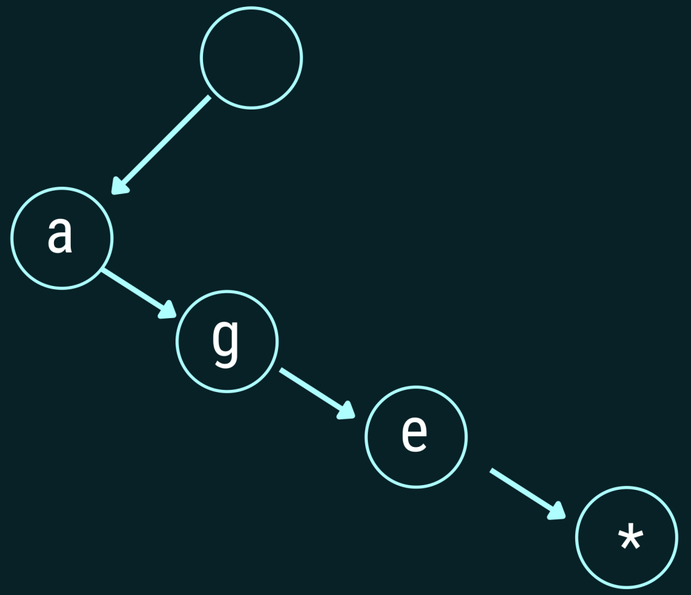
a g e d



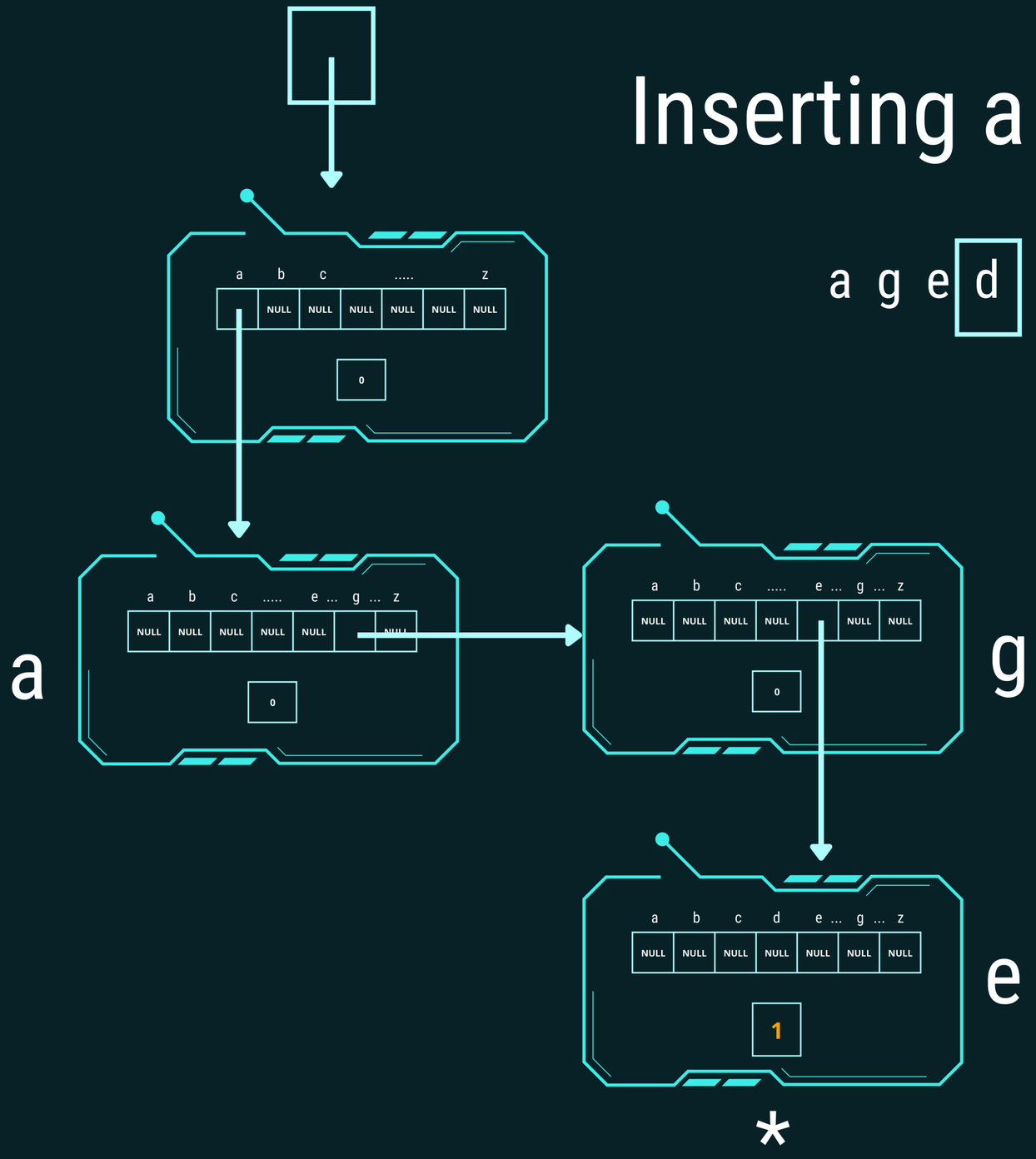
Inserting a Word

a g e d



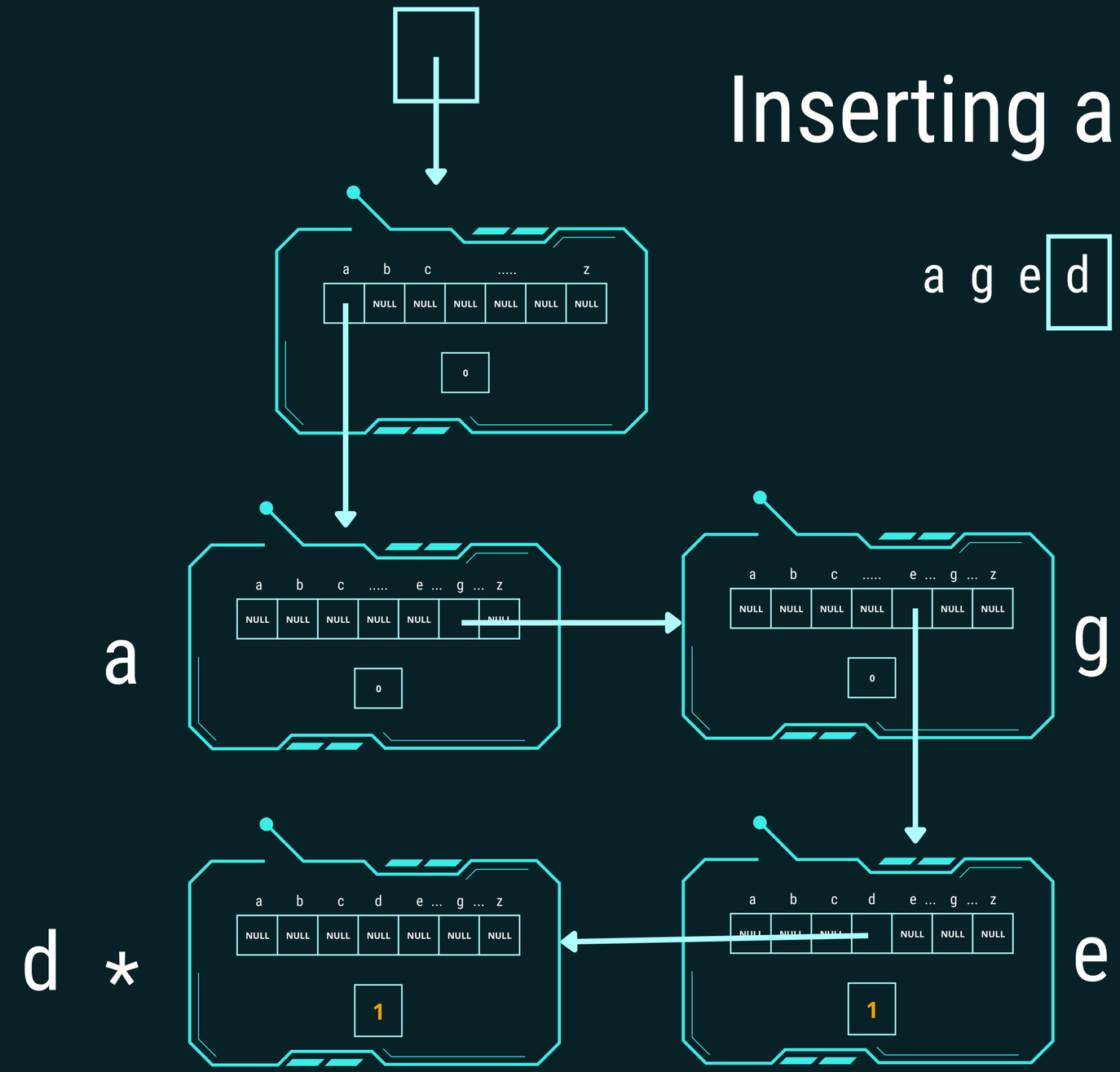


Inserting a Word

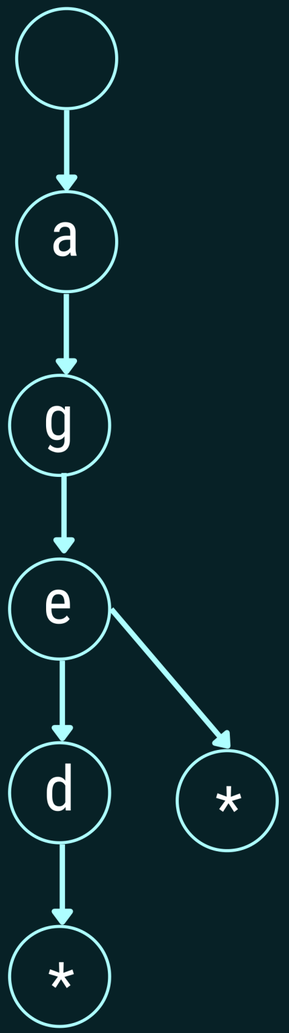


a g e d

Inserting a Word

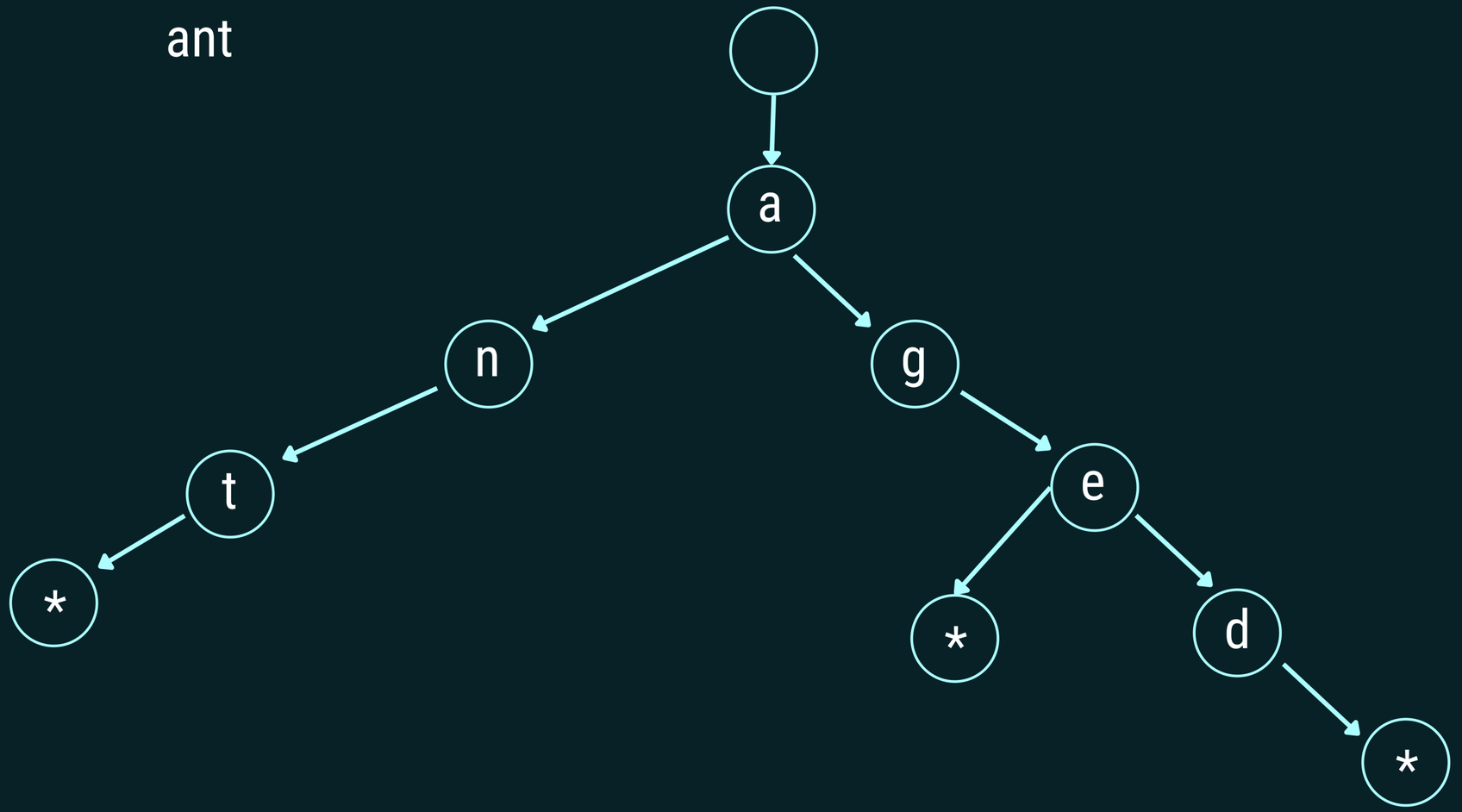


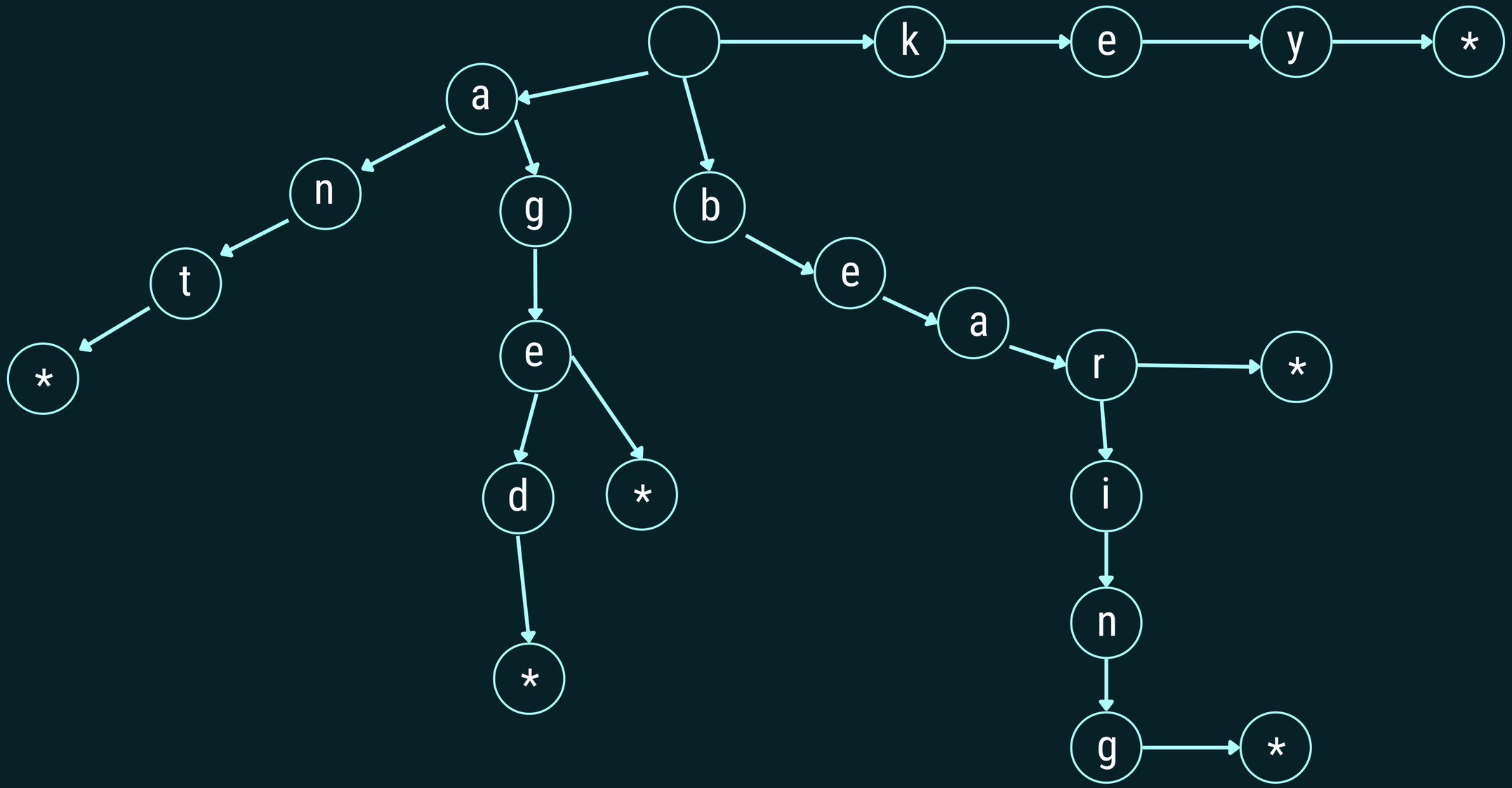
a g e d



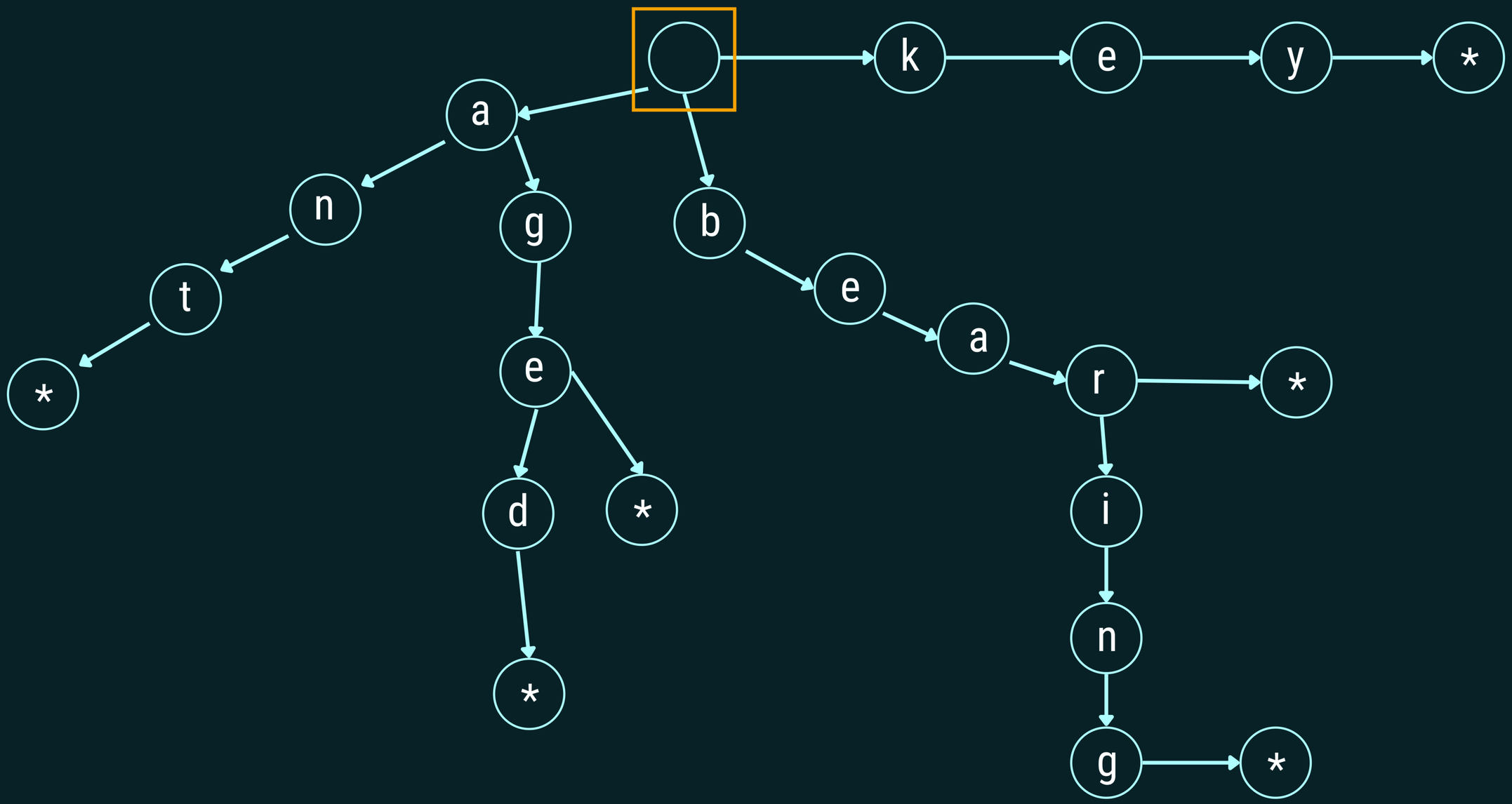
Inserting a Word

ant

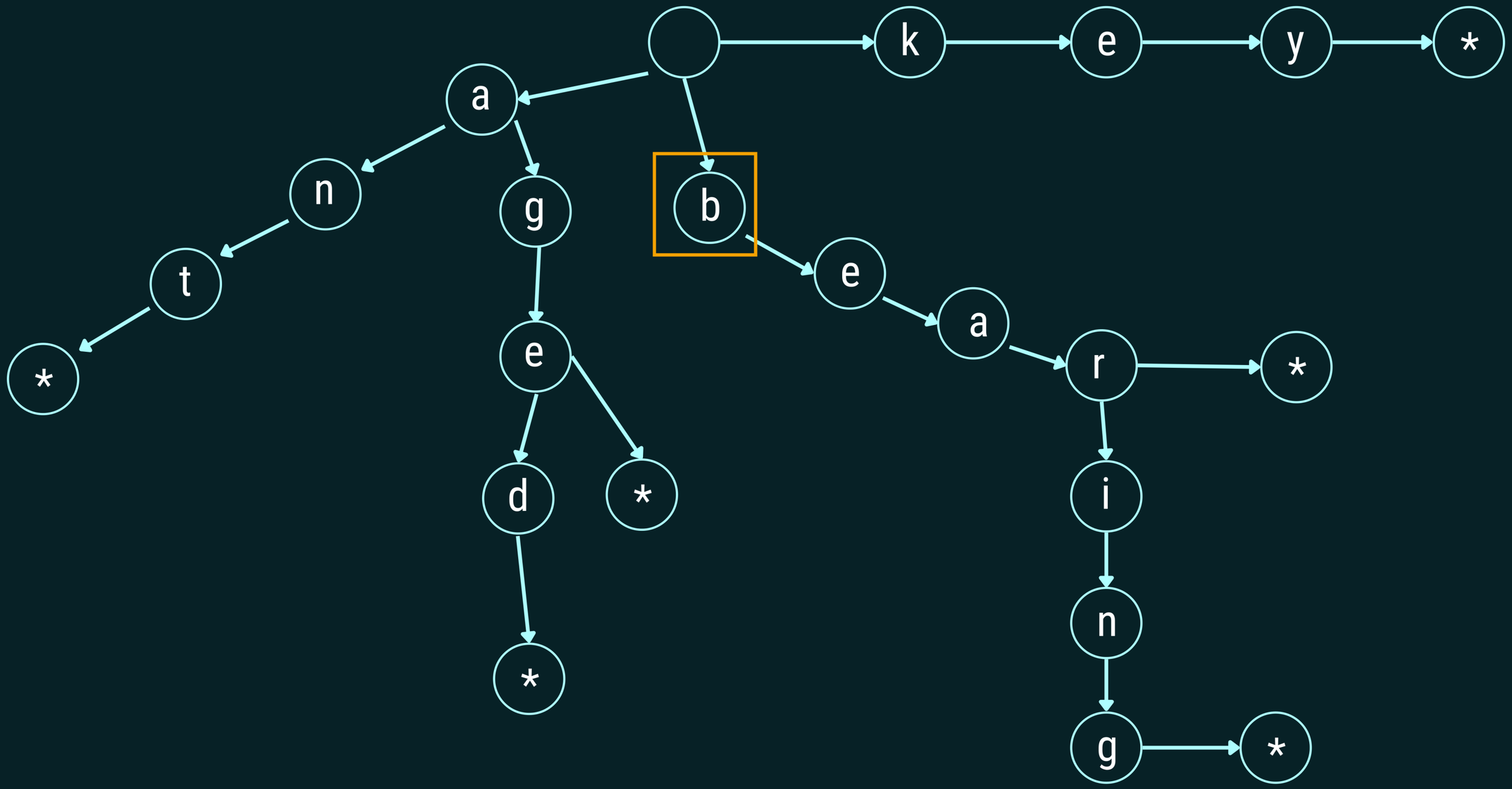



k
e
y
*
a
n
g
b
t
e
a
*
e
r
*
d
*
i
*
n
g
*

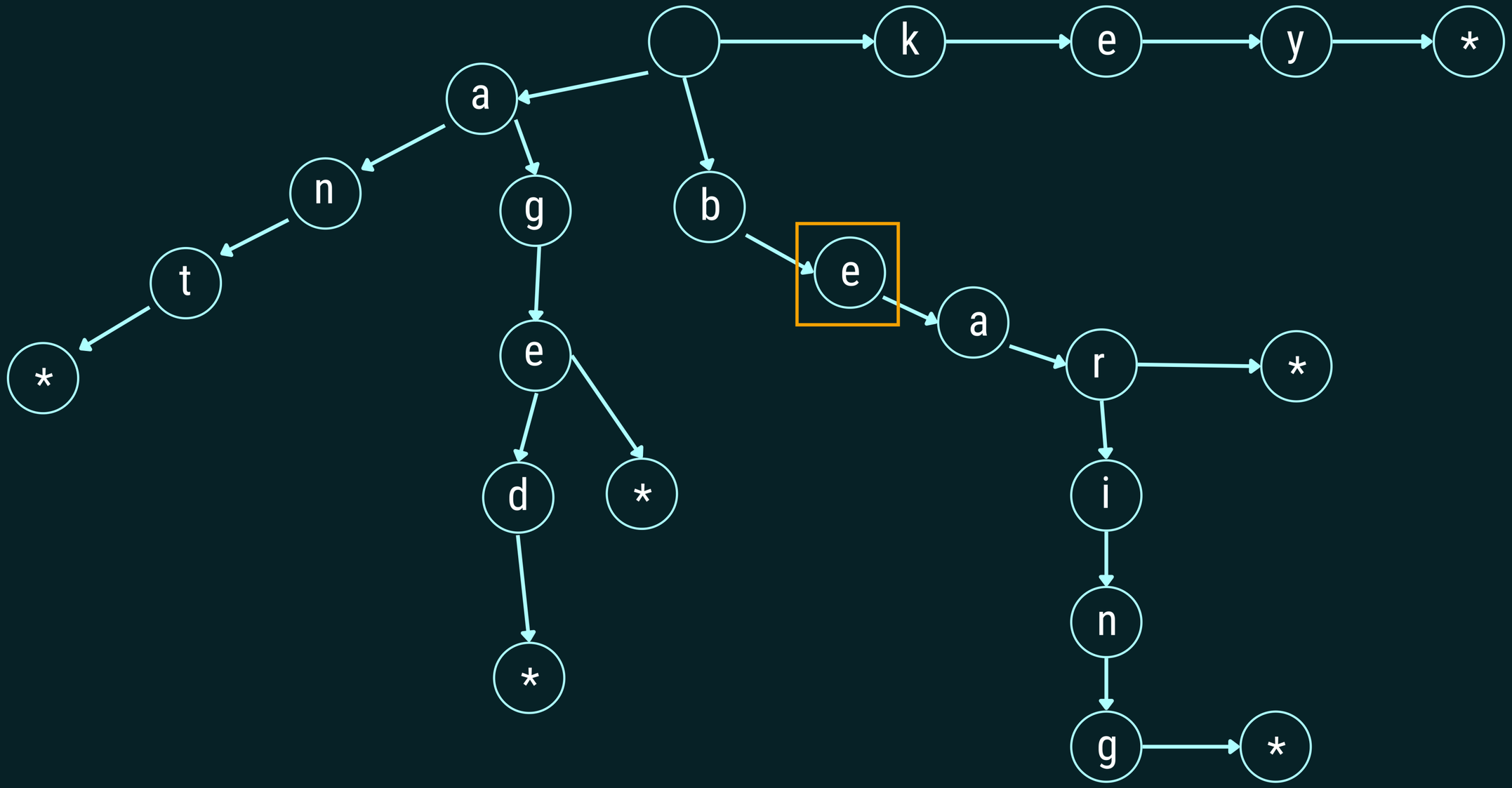
Search - bear



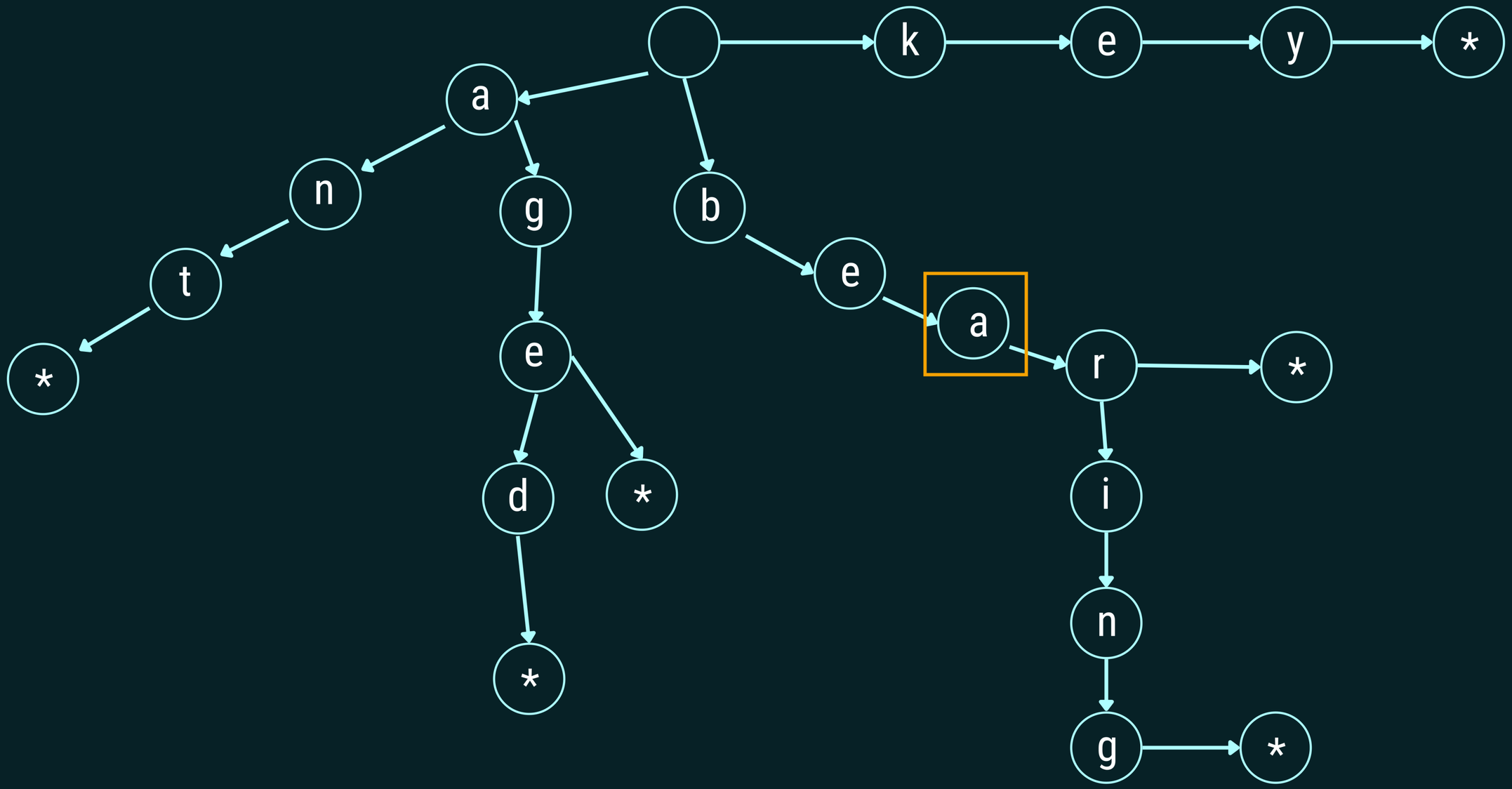
Search - bear



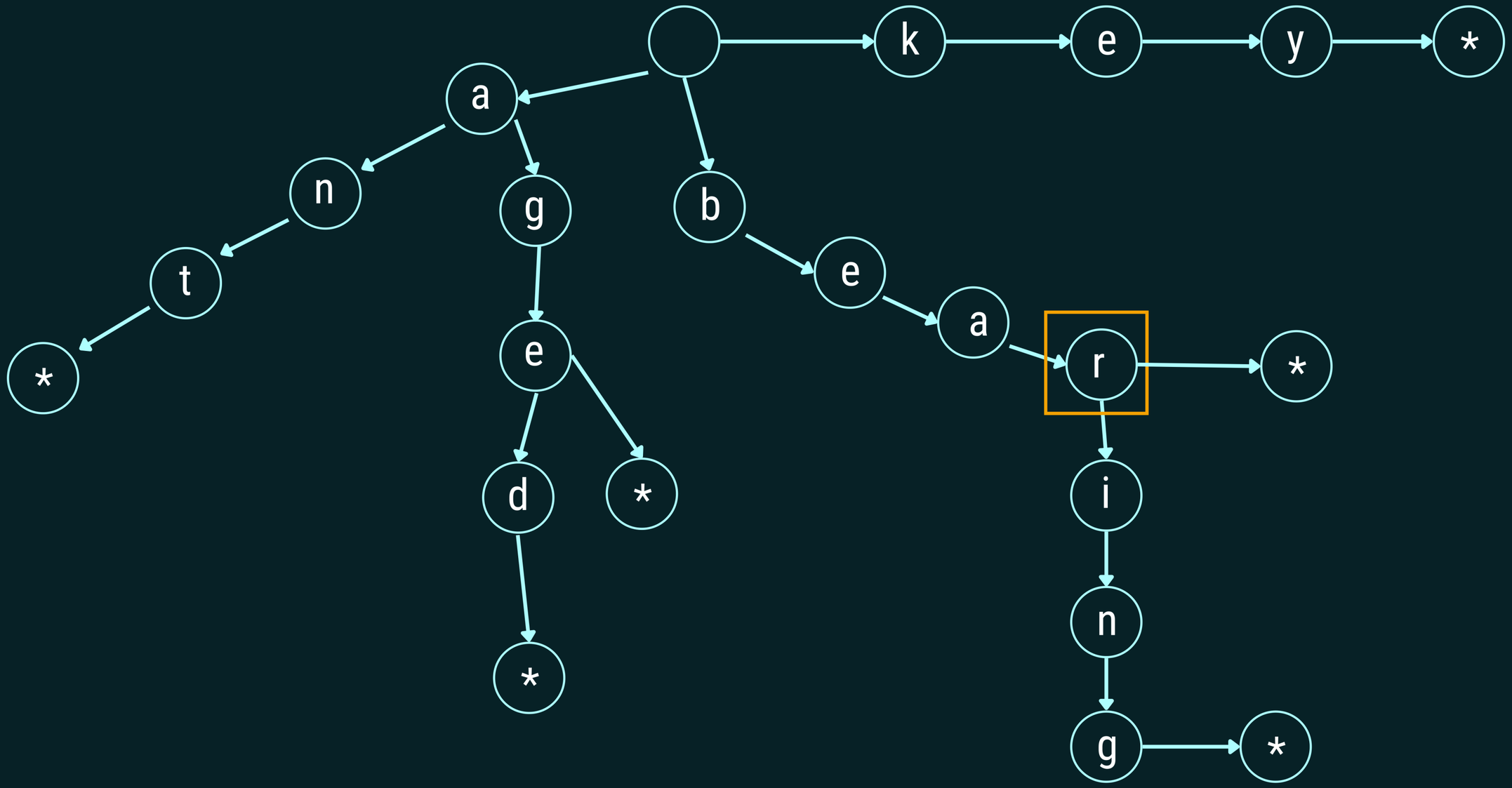
Search - bear



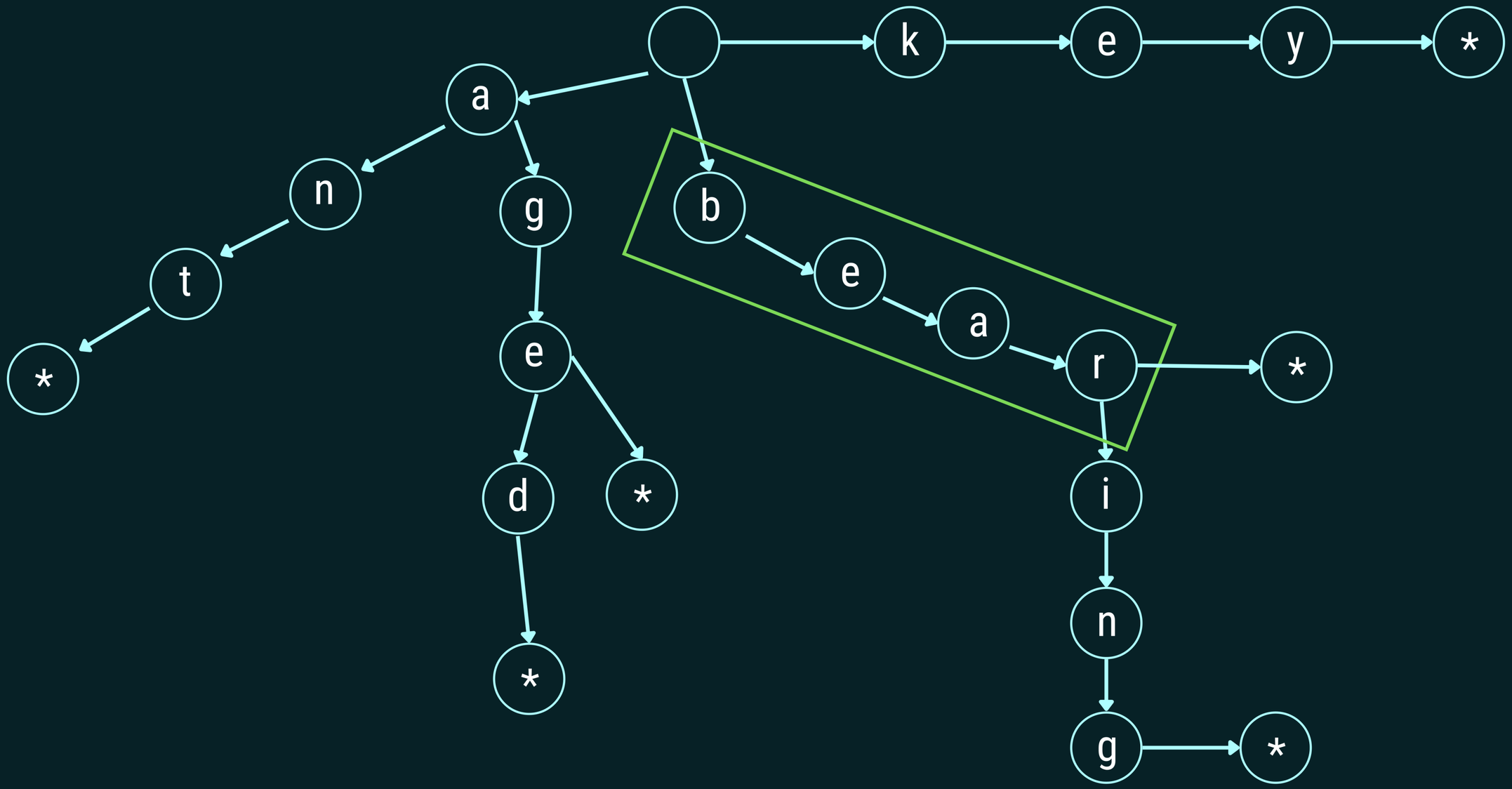
Search - bear



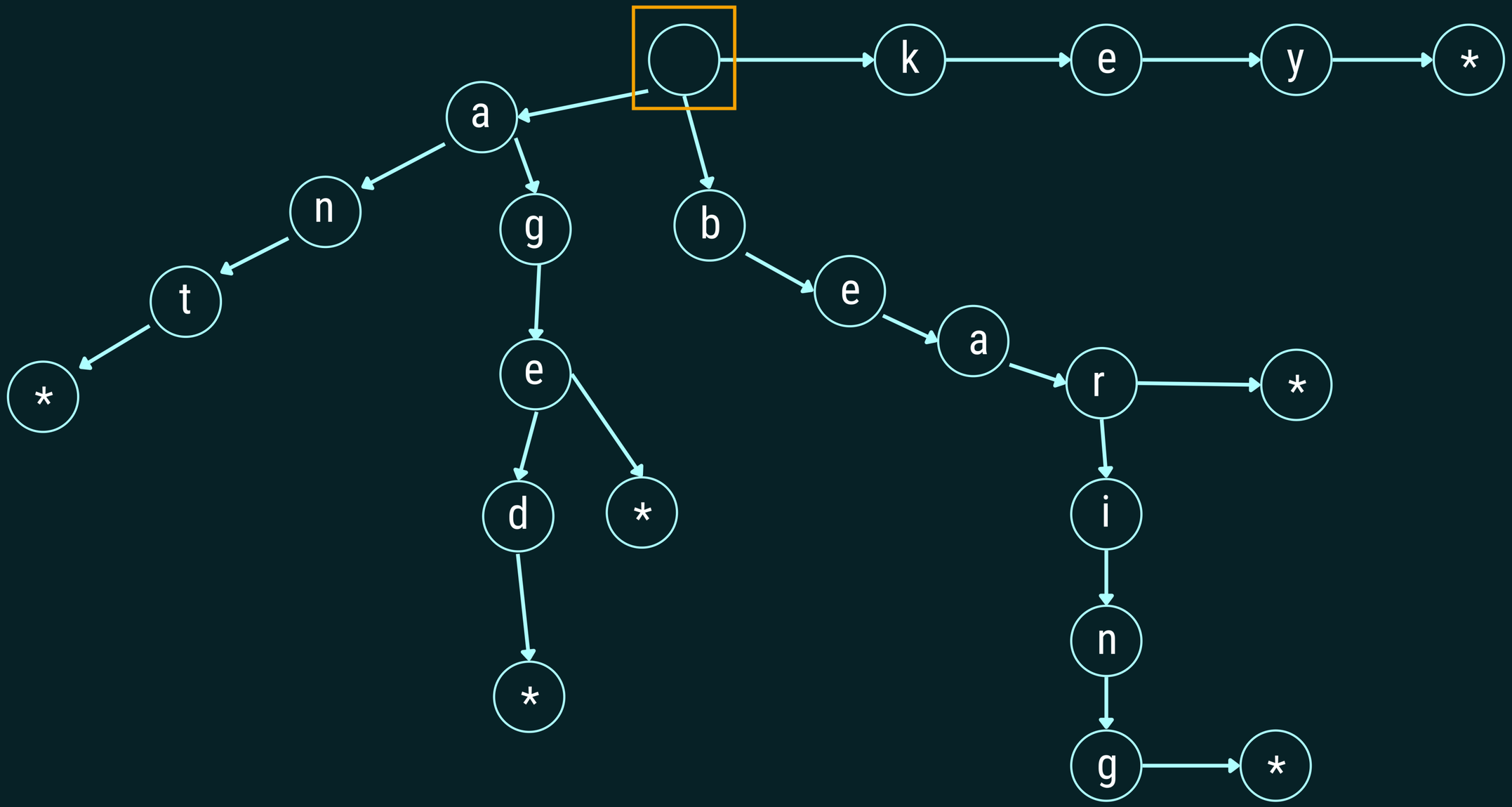
Search - bear



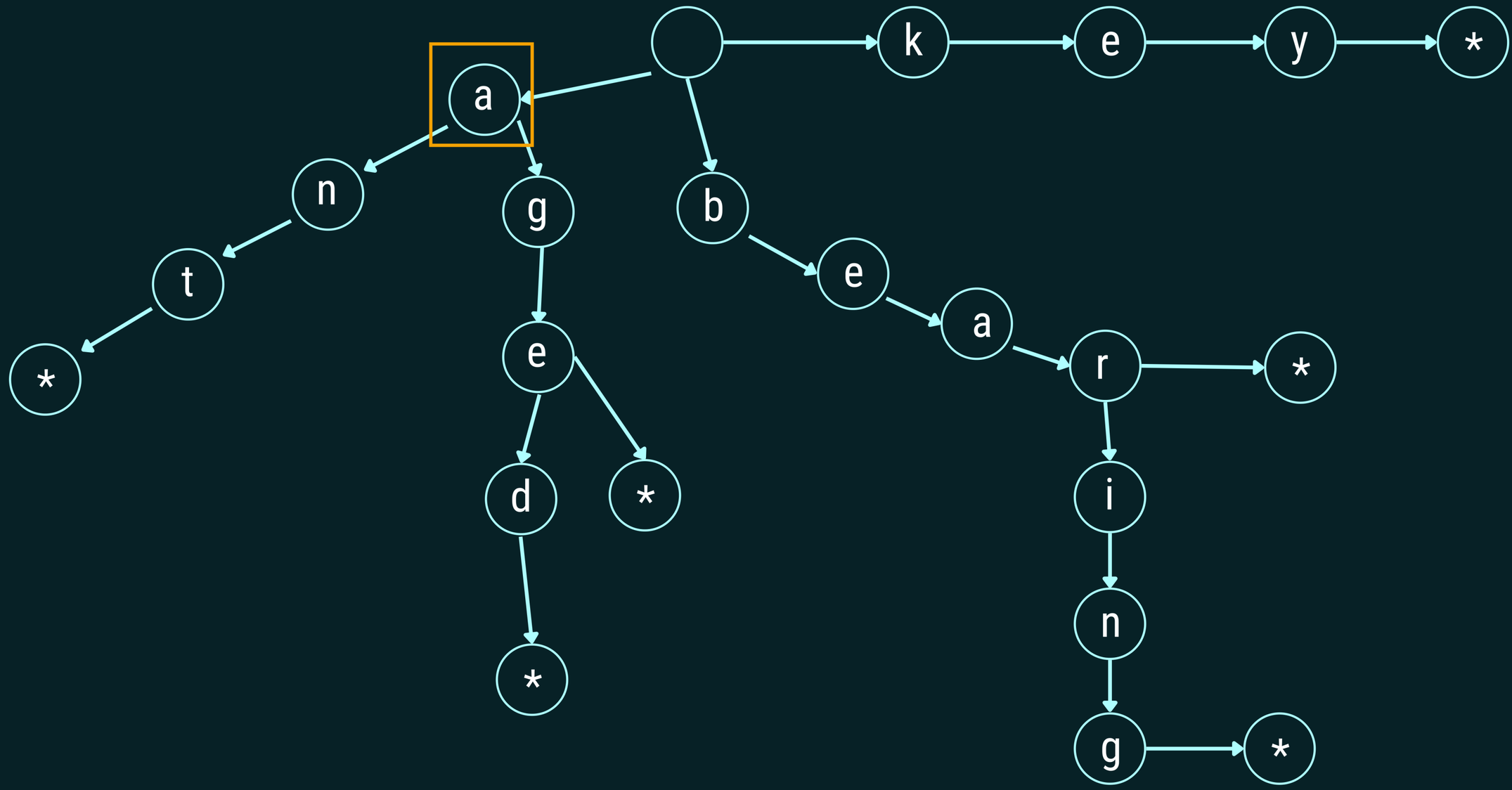
Search - bear



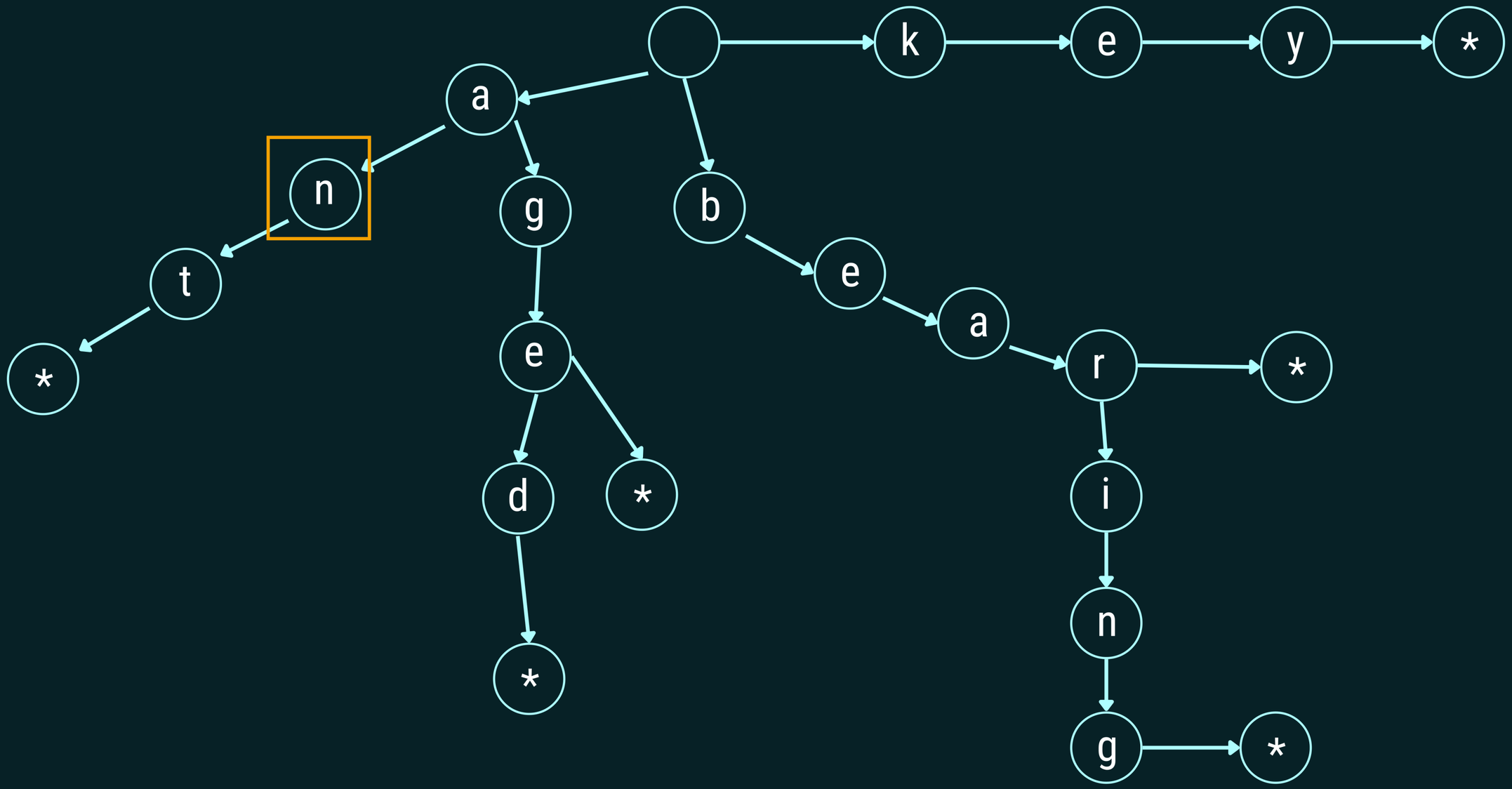
Search - an



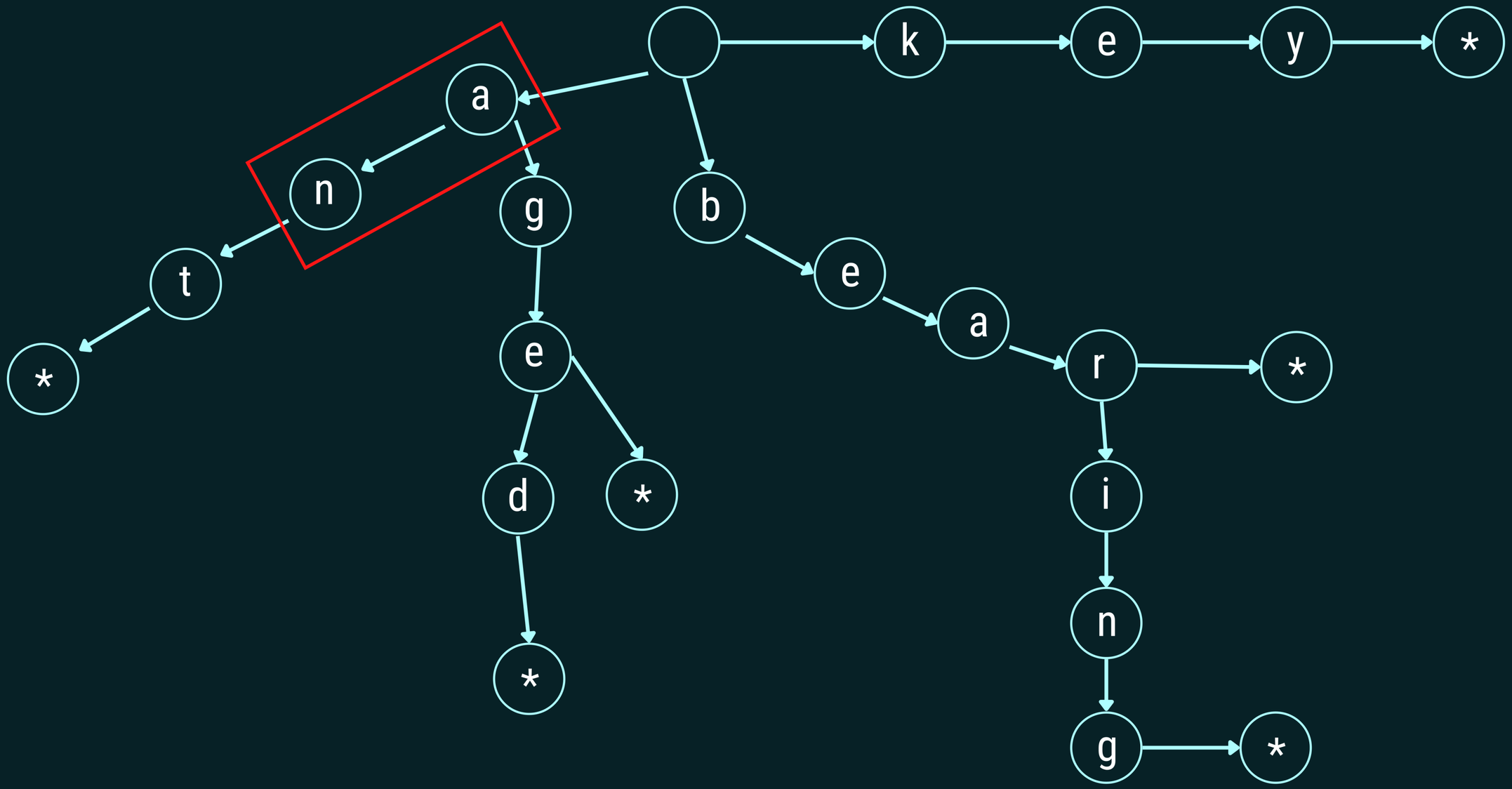
Search - an

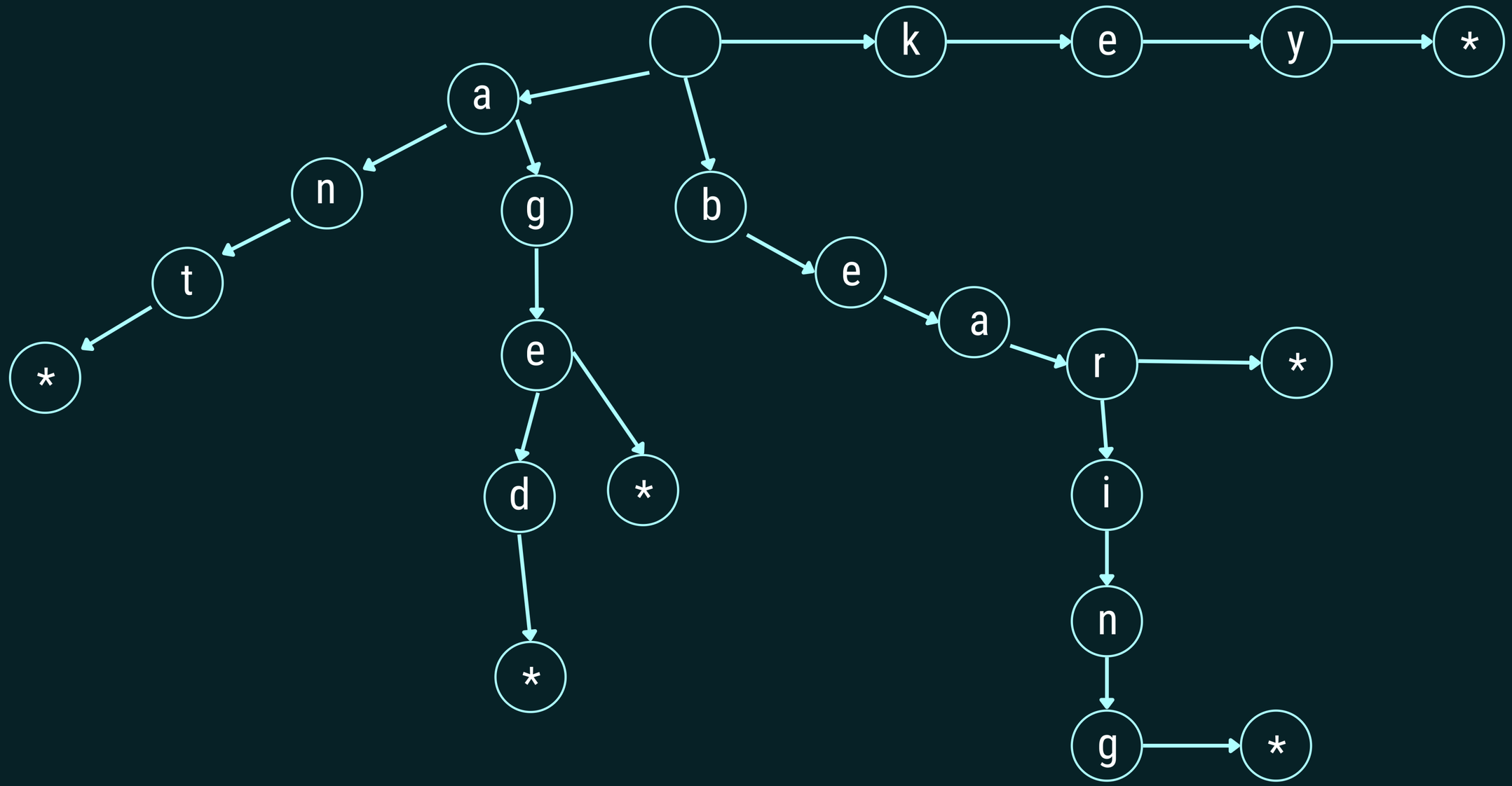


Search - an



Search - an





Bottom-up Approach

Delete a Word

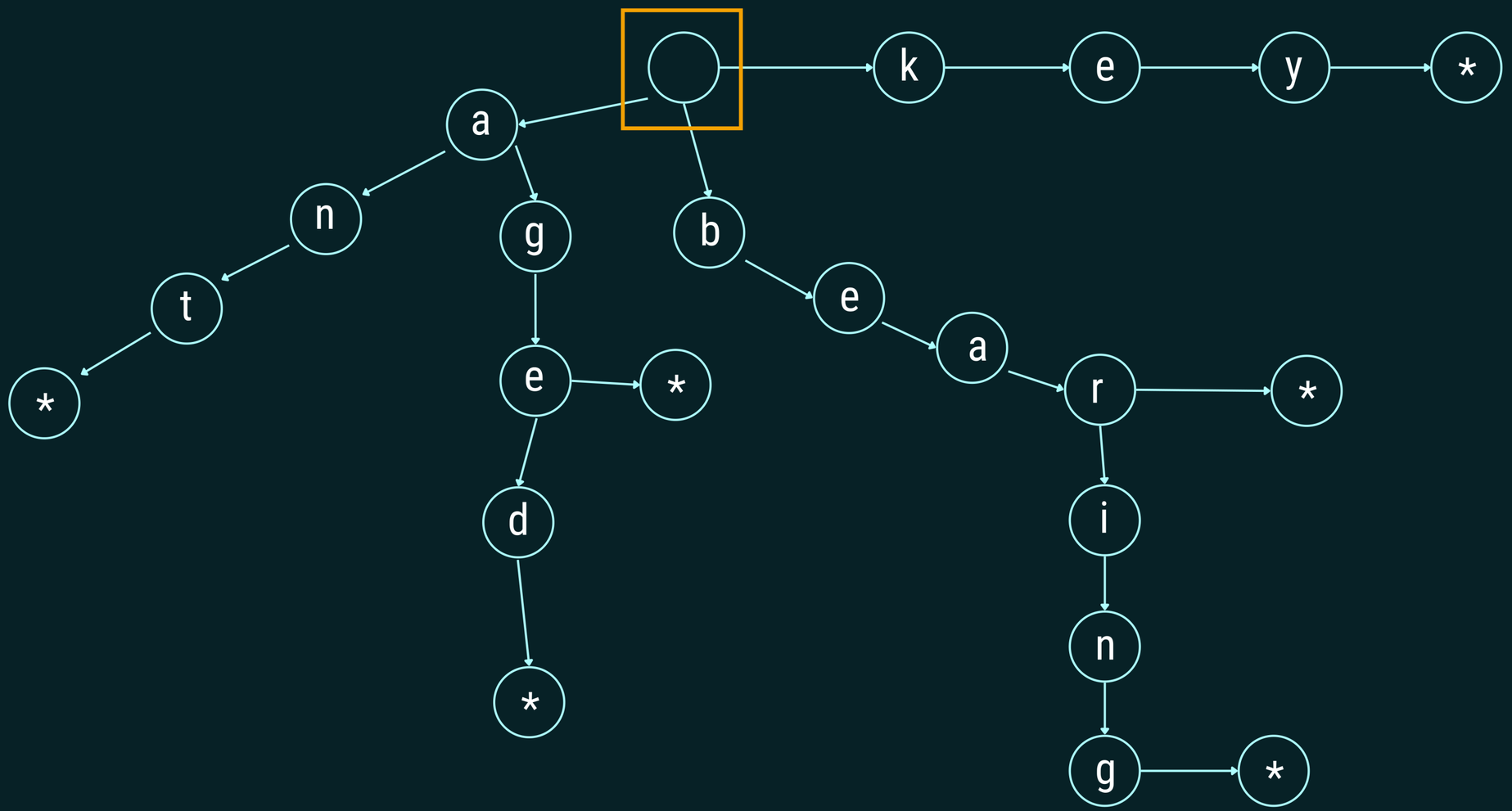
```
delete(root, word, letter)
        // go to the last character
        if(letter == last character)
                return
        delete(root->next character, word, letter + 1)
        check if letter is end of word
                remove end of word
                if letter has no children
                        delete node
        check if letter has no children and
        not end of word
                delete node
```

Delete a Word

1. word does not exist in trie
2. word is not a prefix to other words
3. word is a prefix to other words
4. word is a tail to another word
5. word is a base case

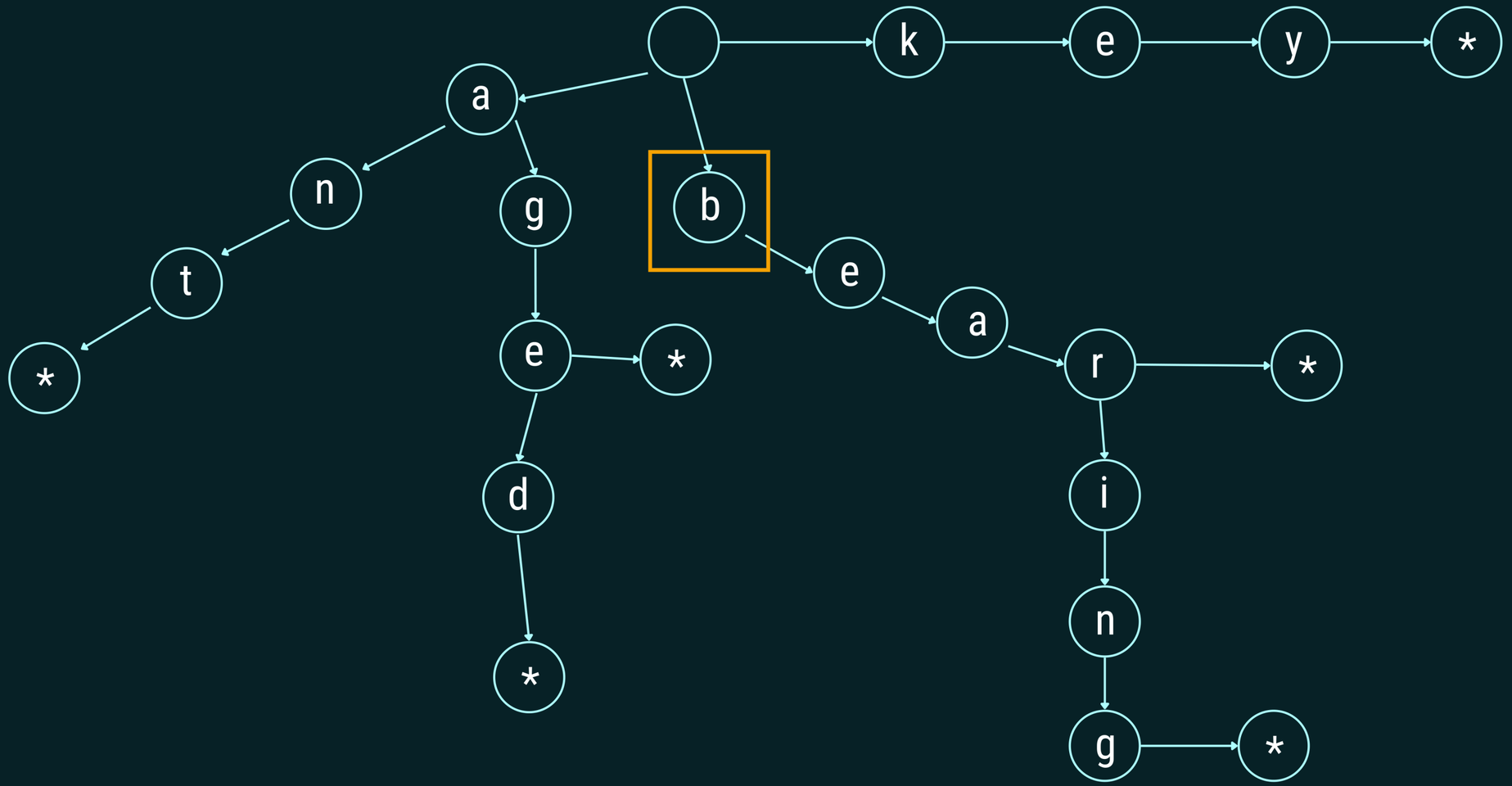
word does not exist in trie

b a g

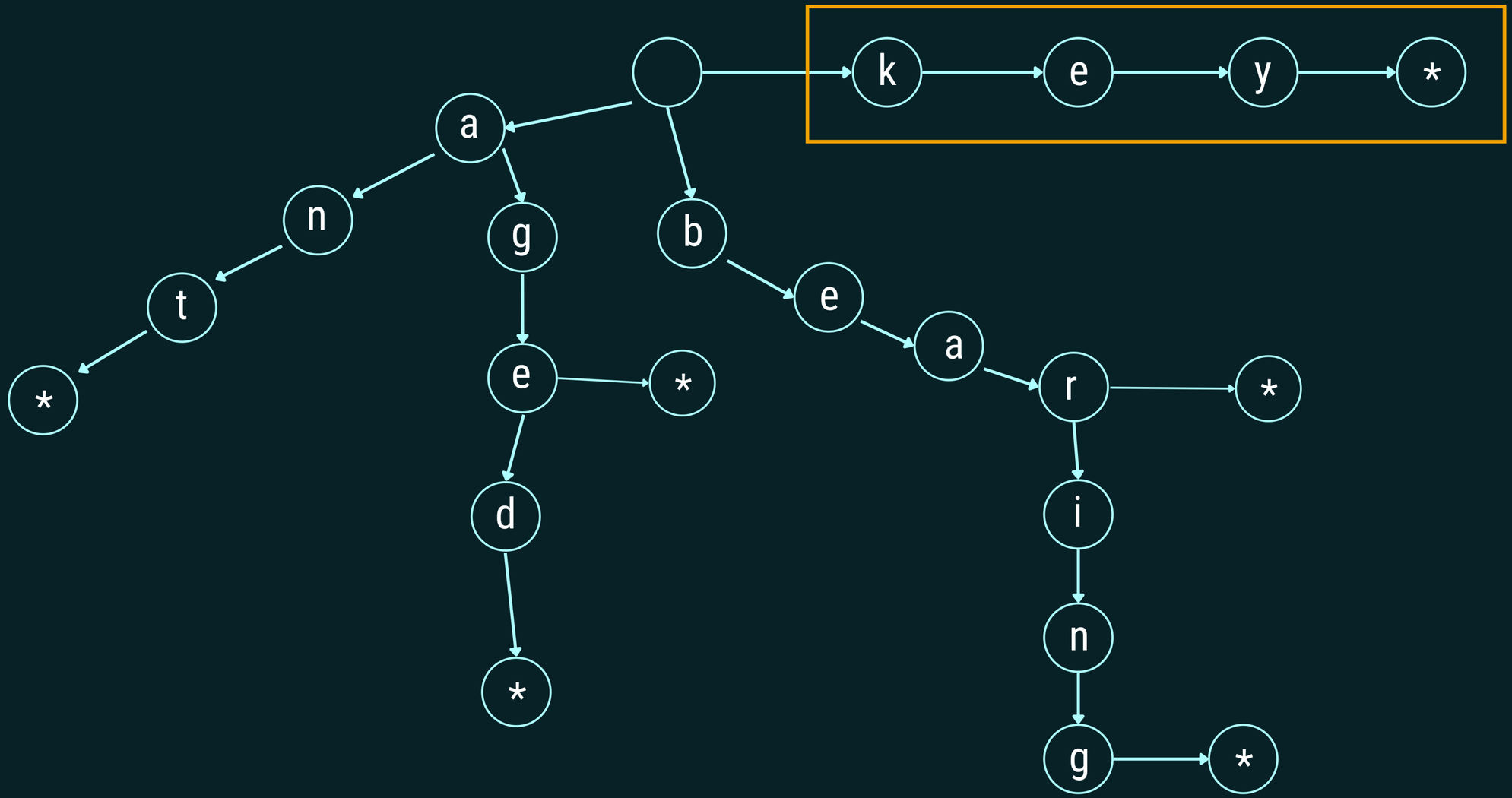


word does not exist in trie

b a g

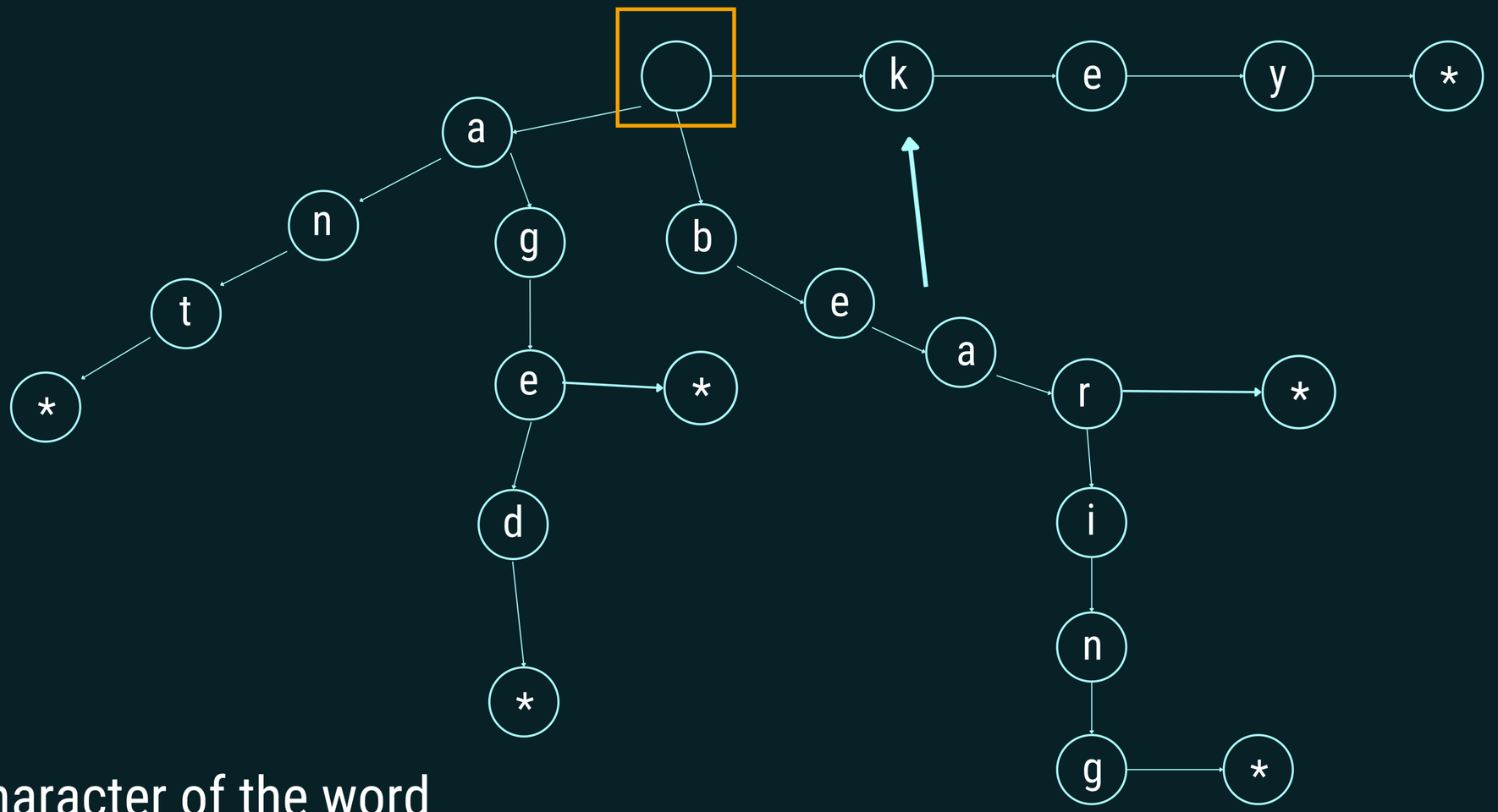


word is not a prefix to other words



word is not a prefix to other words

k e y

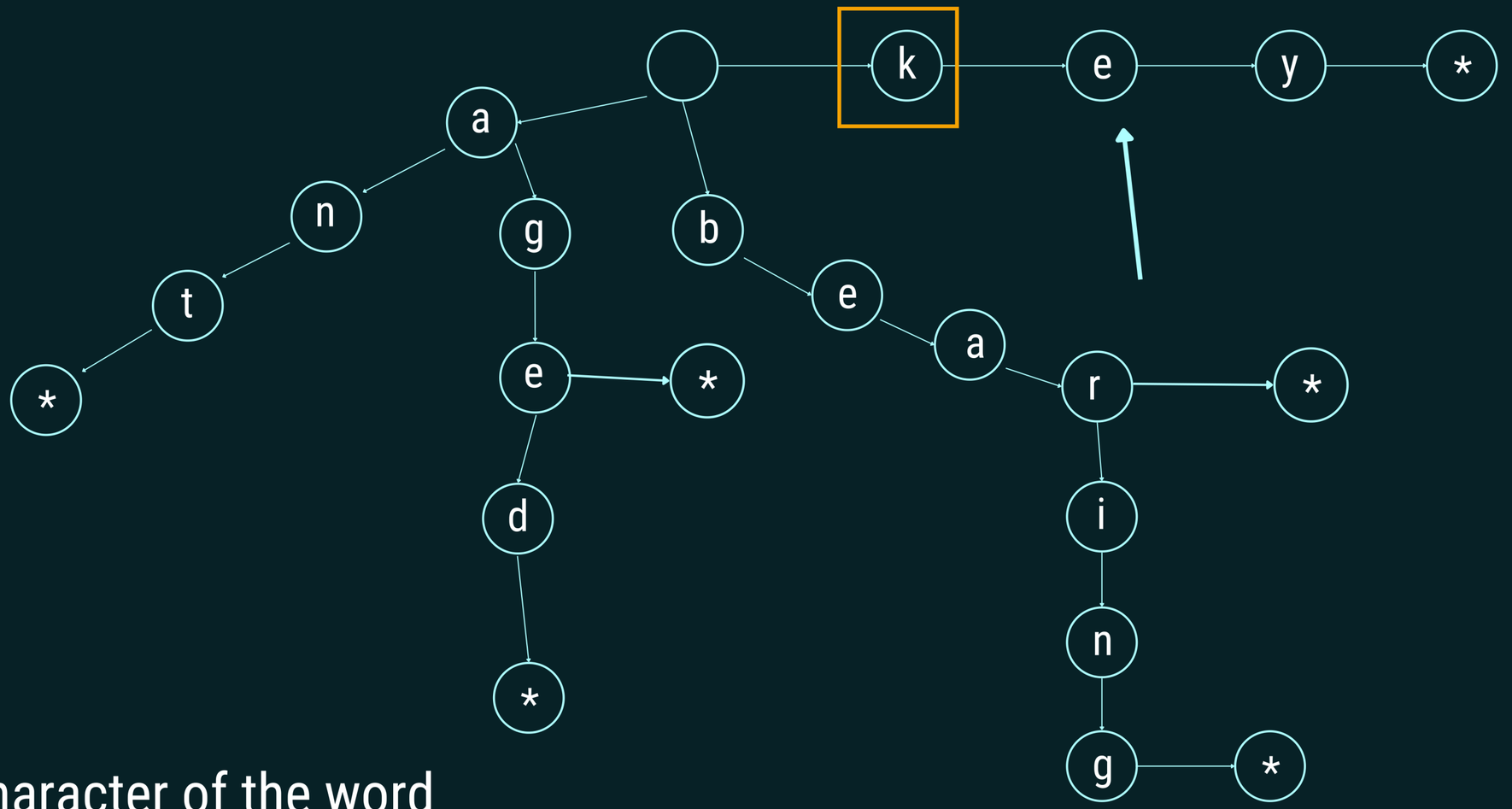


go to the last character of the word

using recursion

word is not a prefix to other words

k e y

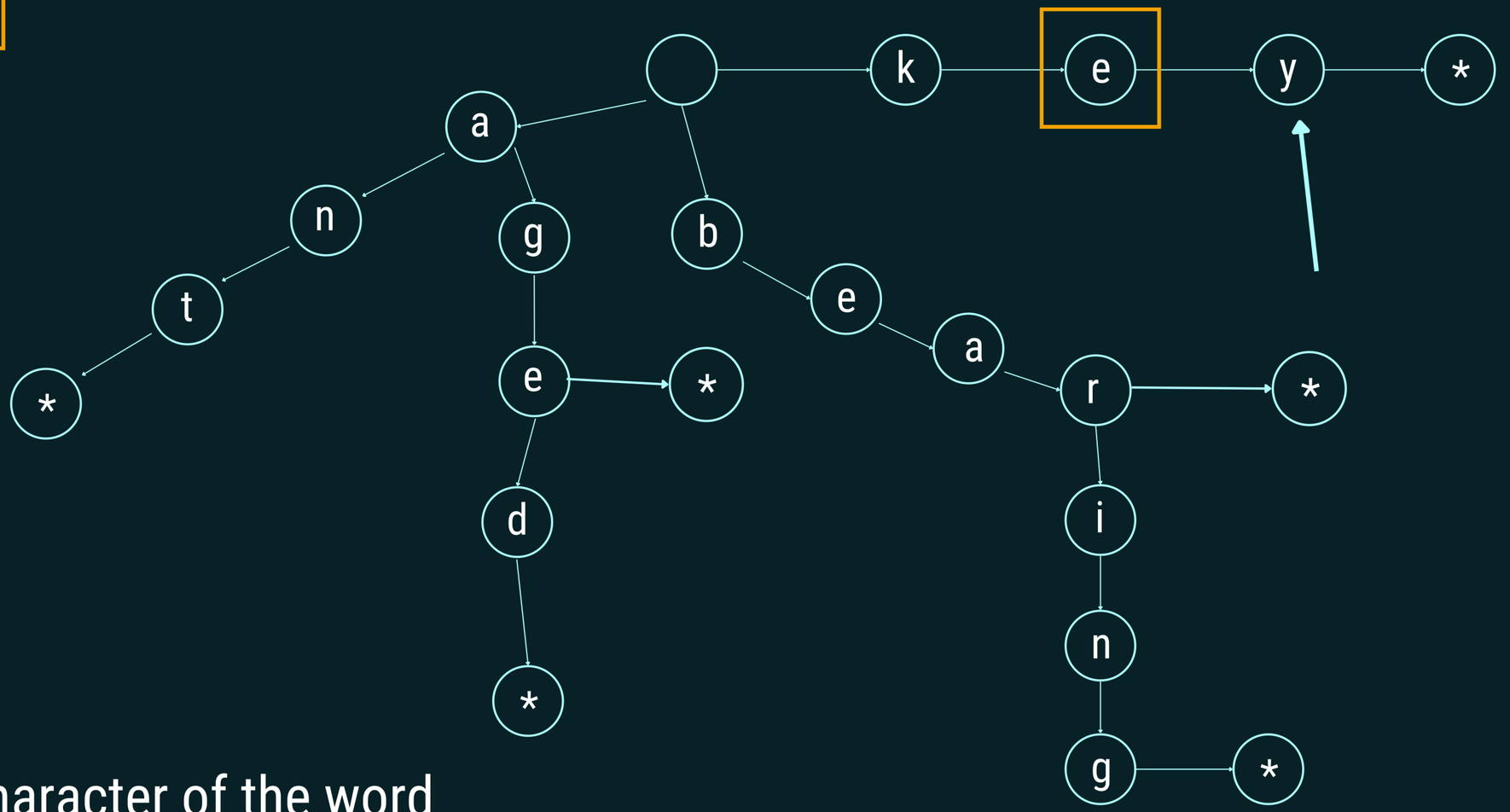


go to the last character of the word

using recursion

word is not a prefix to other words

k e y

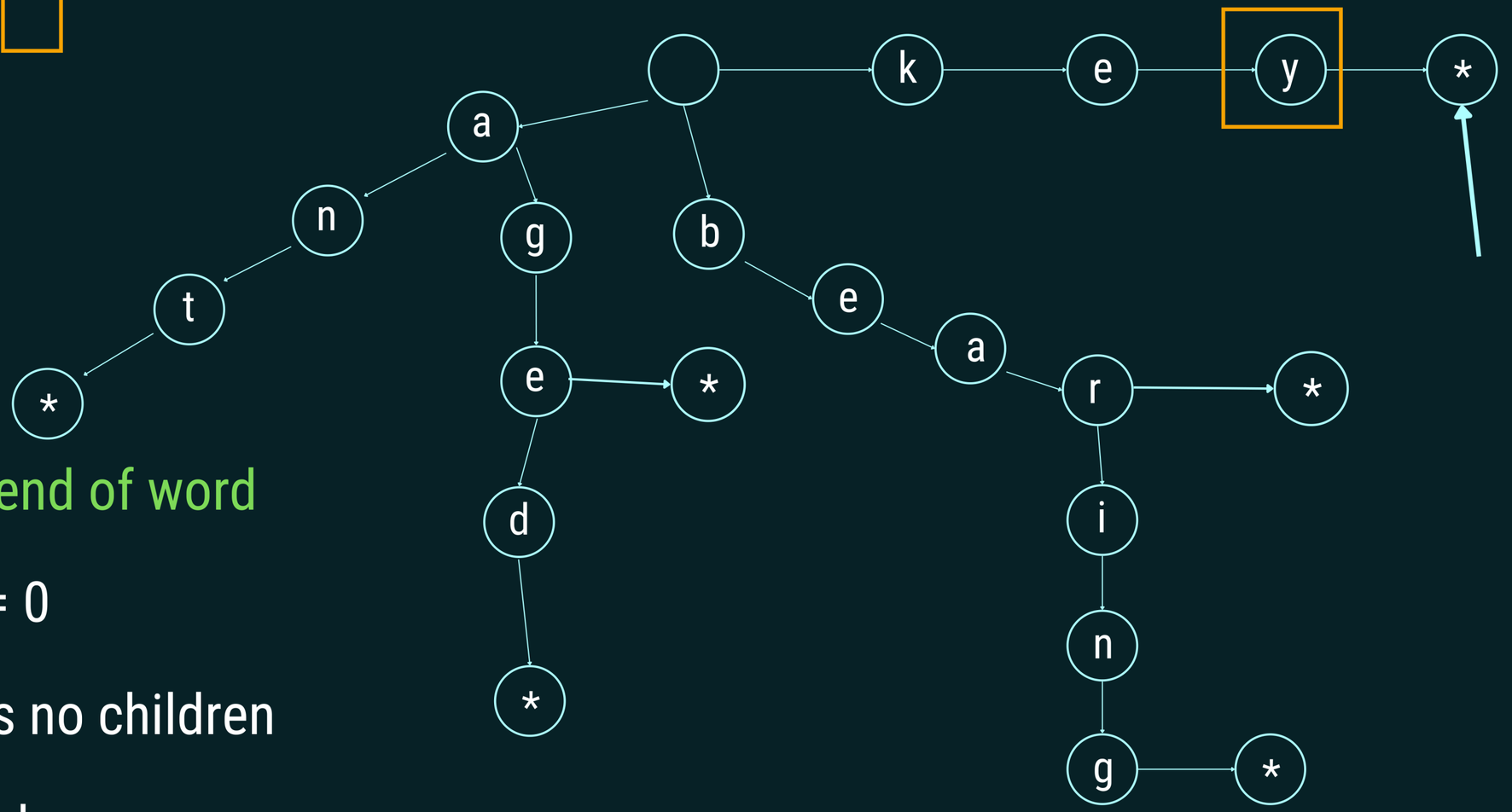


go to the last character of the word

using recursion

word is not a prefix to other words

k e y



check if it is the end of word

end of word = 0

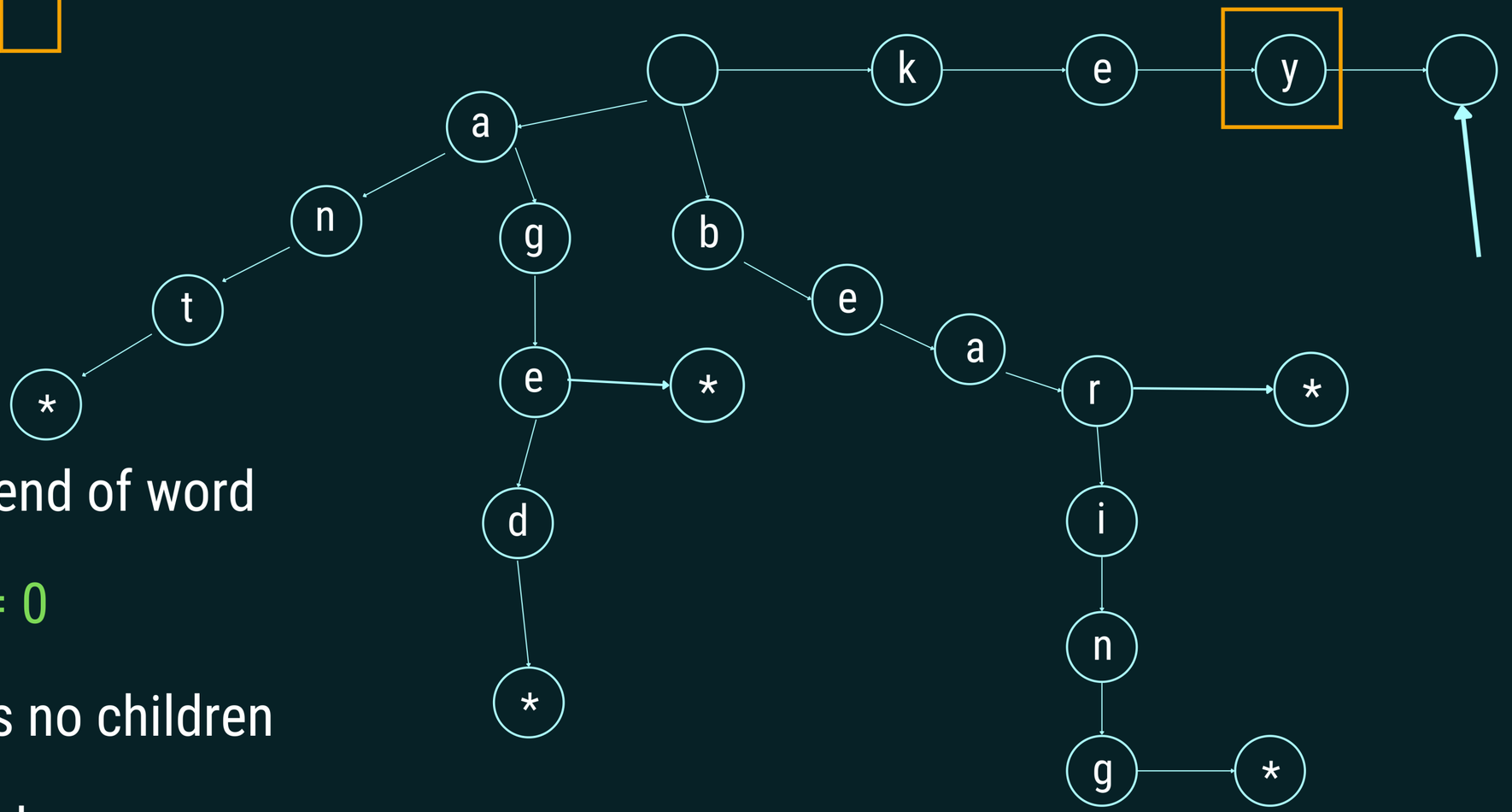
check if it has no children

delete node

exit

word is not a prefix to other words

k e y



check if it is the end of word

end of word = 0

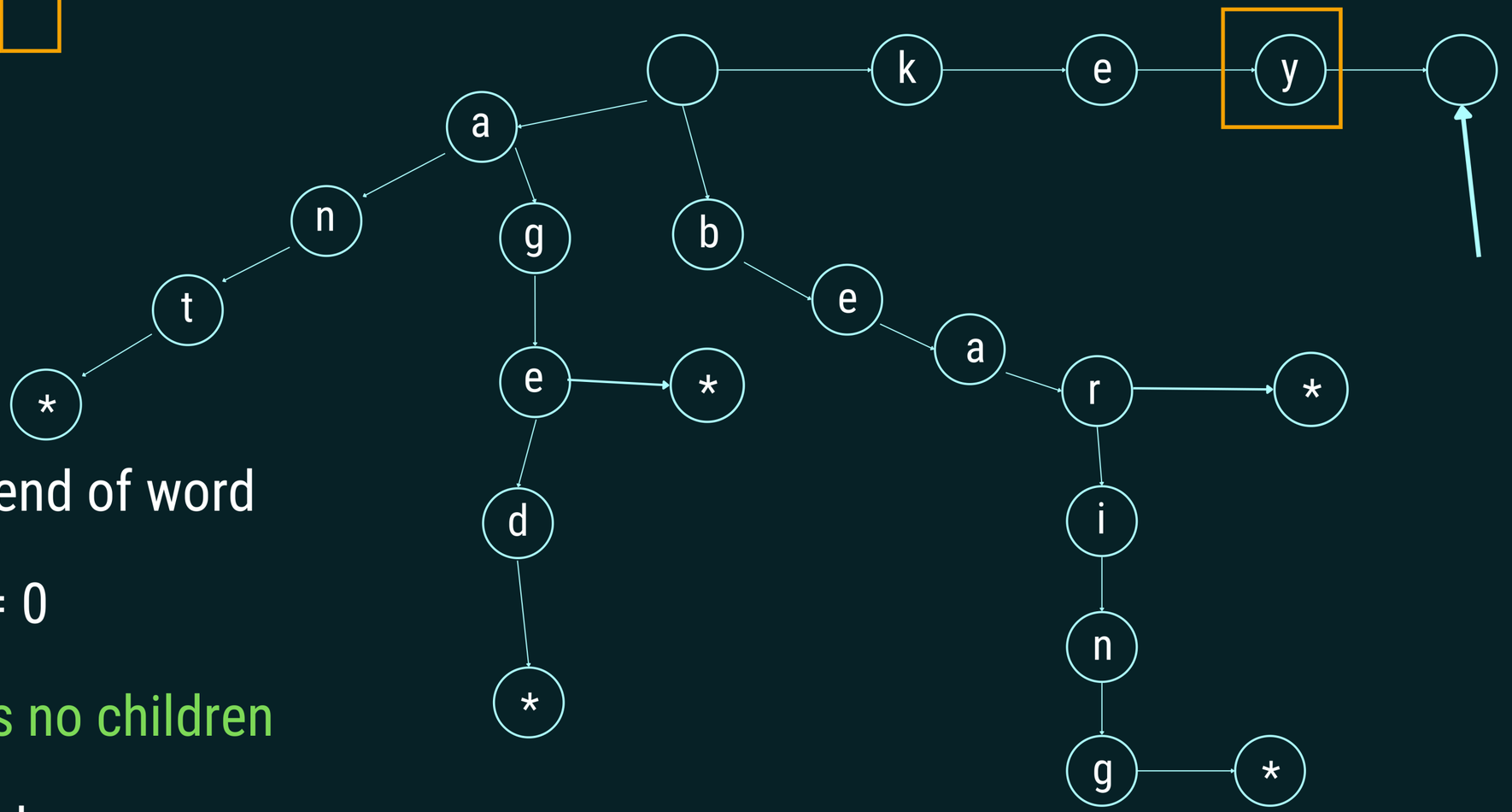
check if it has no children

delete node

exit

word is not a prefix to other words

k e y



check if it is the end of word

end of word = 0

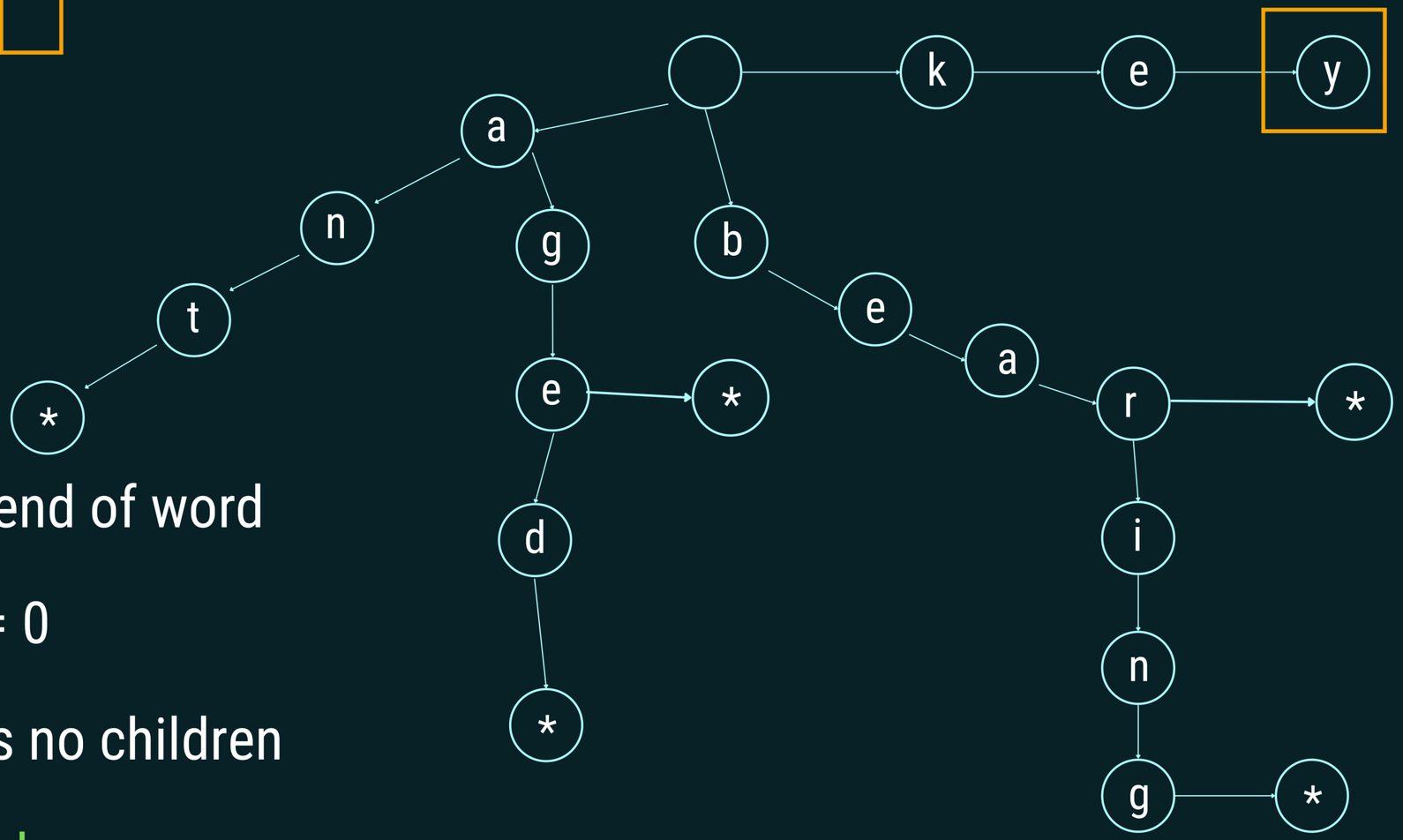
check if it has no children

delete node

exit

word is not a prefix to other words

k e y



check if it is the end of word

end of word = 0

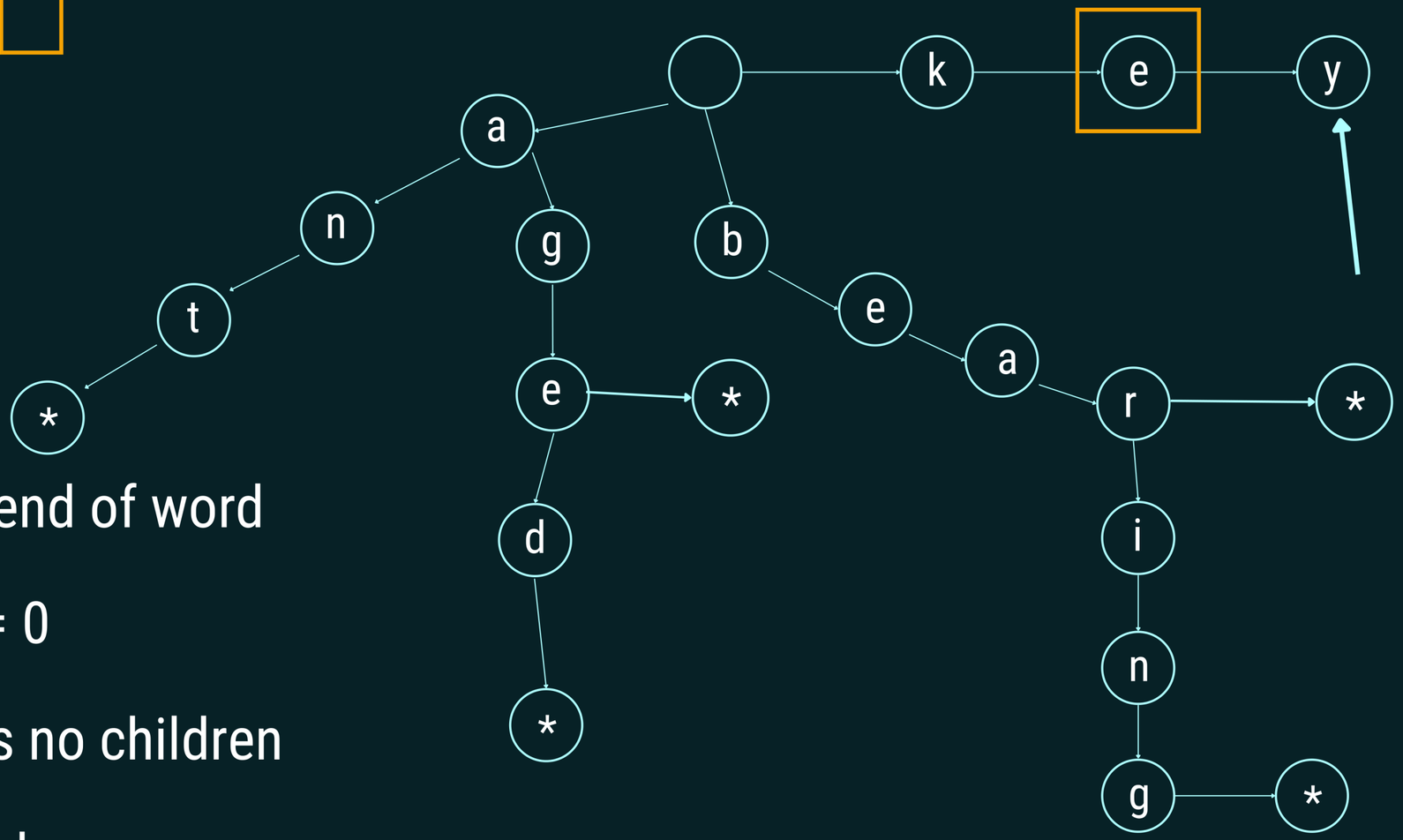
check if it has no children

delete node

exit

word is not a prefix to other words

k e y



check if it is the end of word

end of word = 0

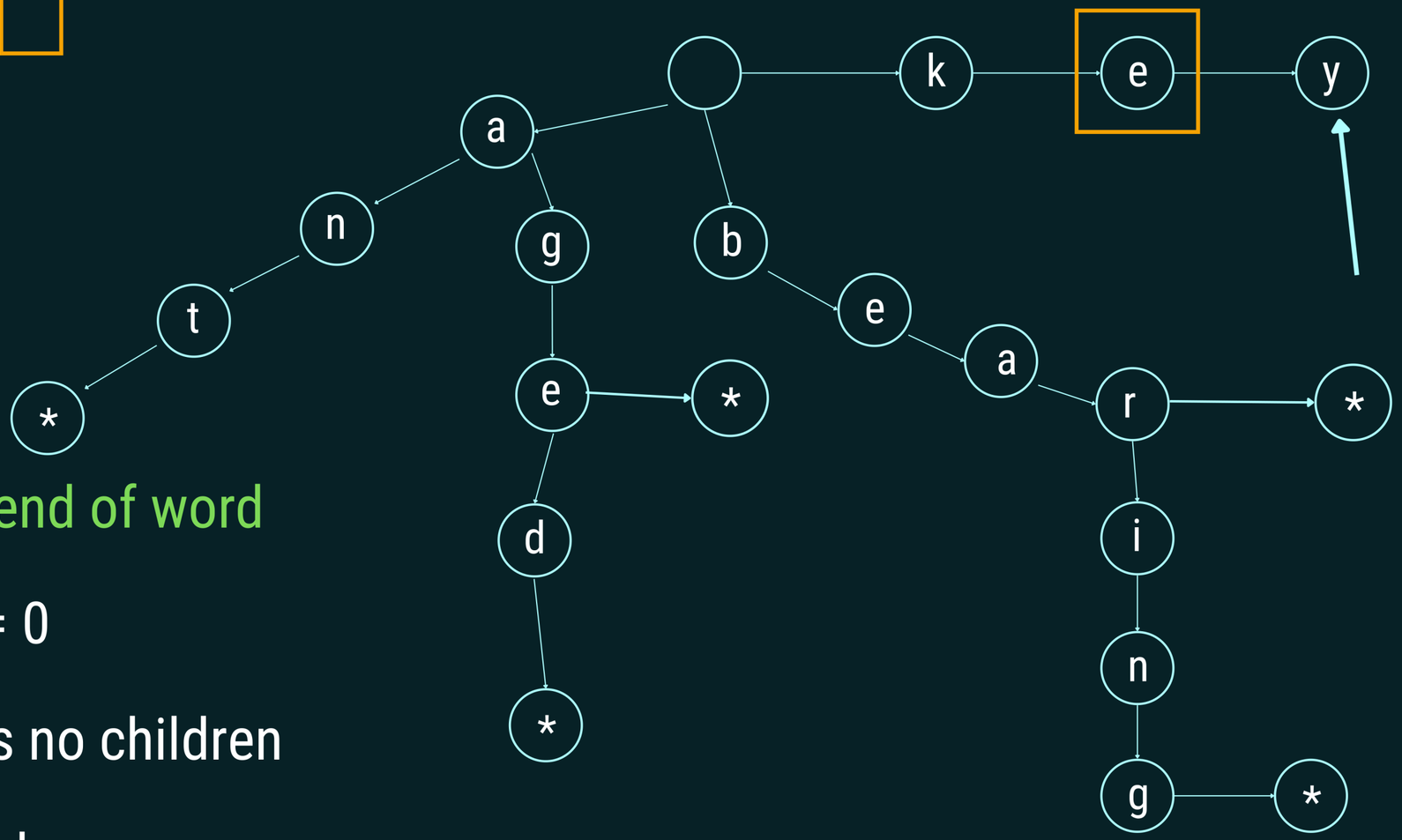
check if it has no children

delete node

exit

word is not a prefix to other words

k e y



check if it is the end of word

end of word = 0

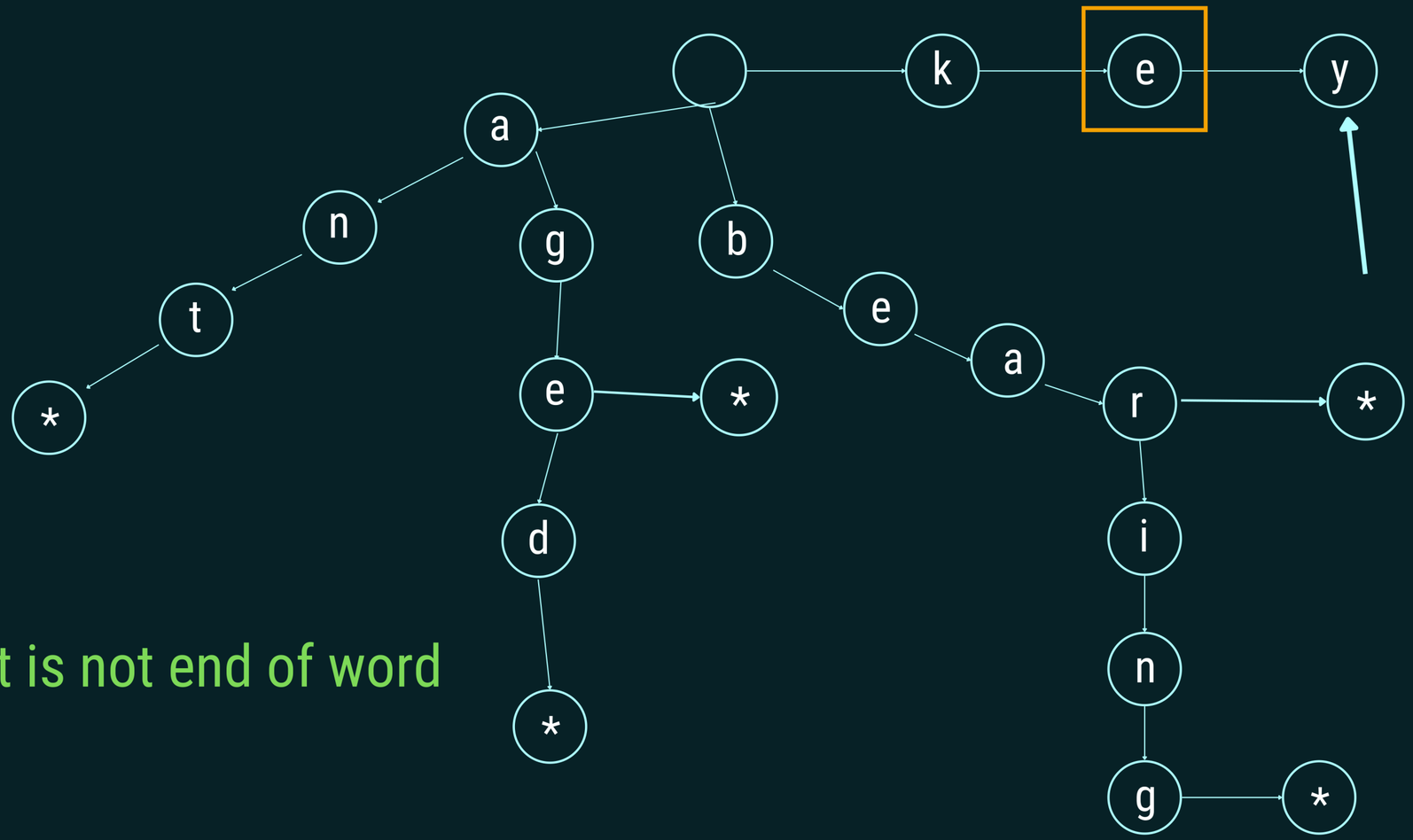
check if it has no children

delete node

exit

word is not a prefix to other words

k e y



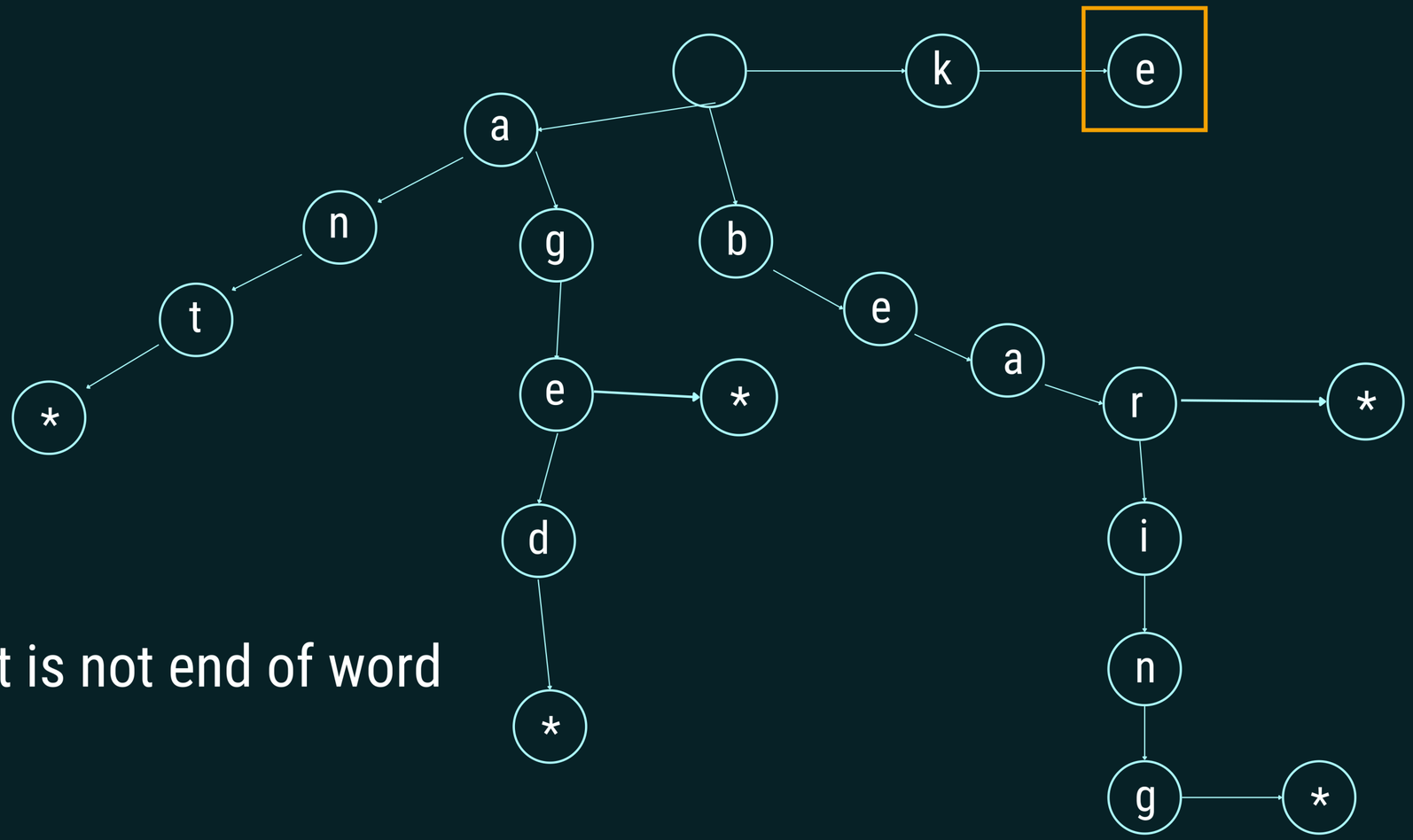
check if it has no children AND it is not end of word

delete the node

exit

word is not a prefix to other words

k e y



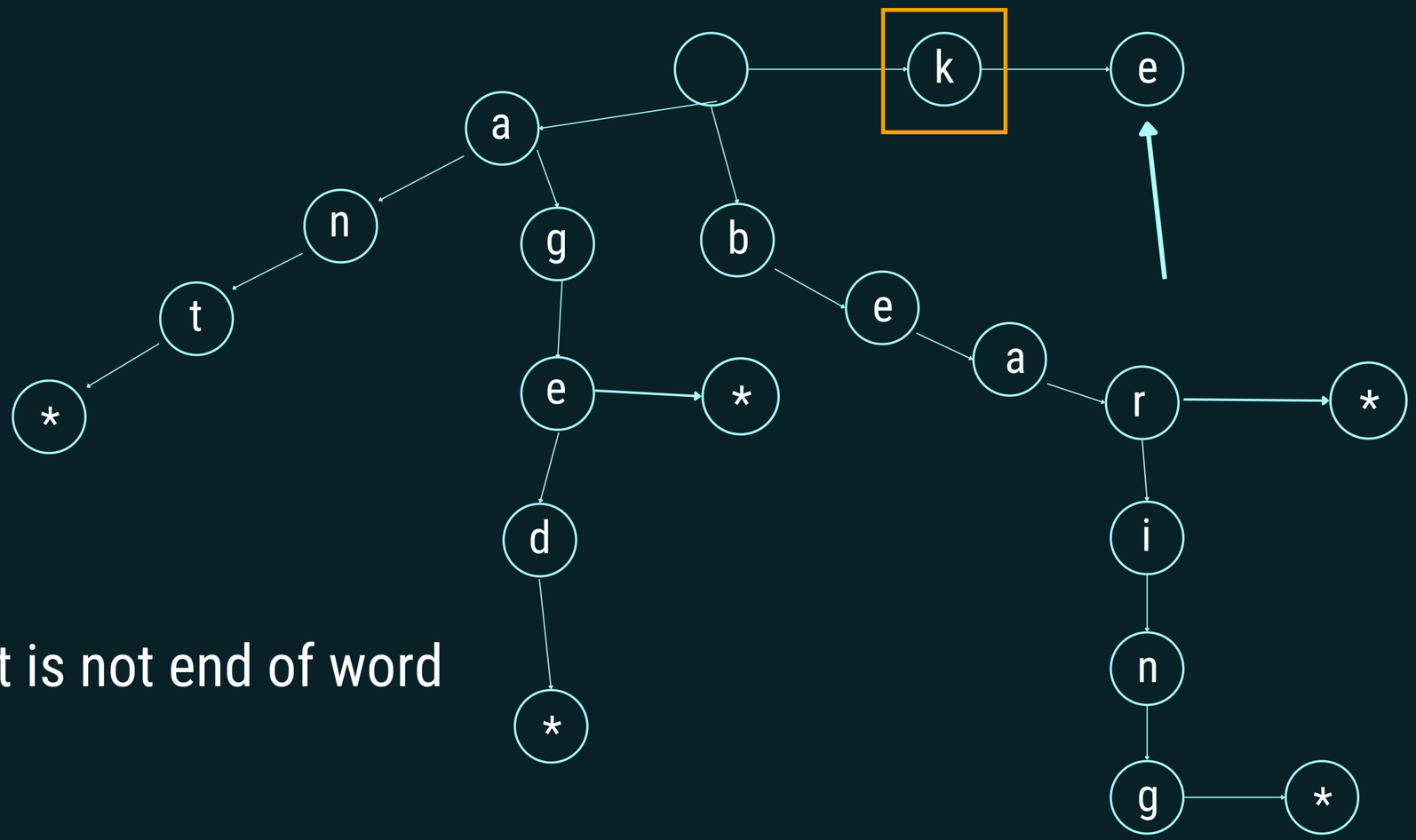
check if it has no children AND it is not end of word

delete the node

exit

word is not a prefix to other words

k e y



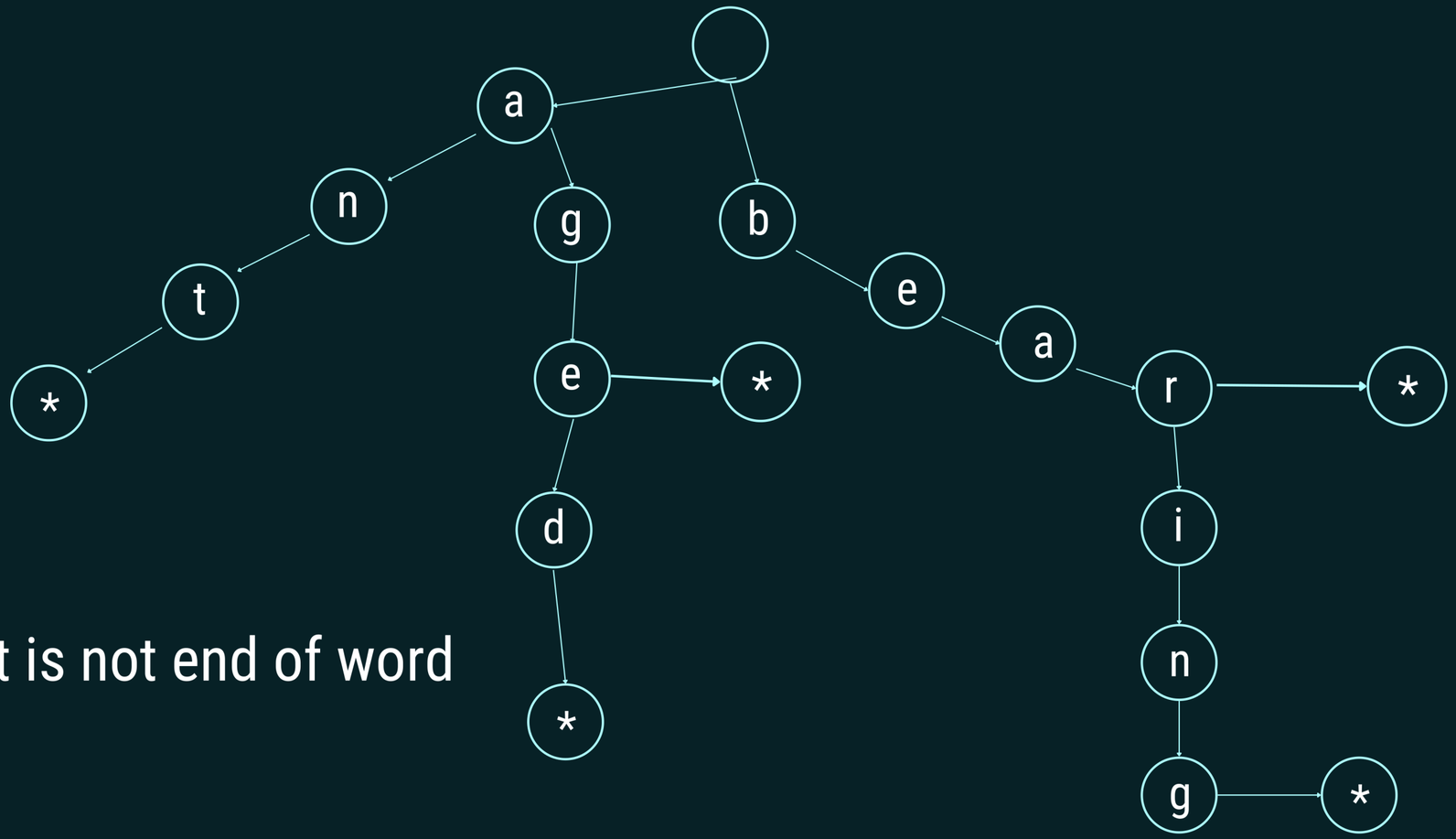
check if it has no children AND it is not end of word

delete the node

exit

word is not a prefix to other words

k e y



check if it has no children AND it is not end of word

delete the node

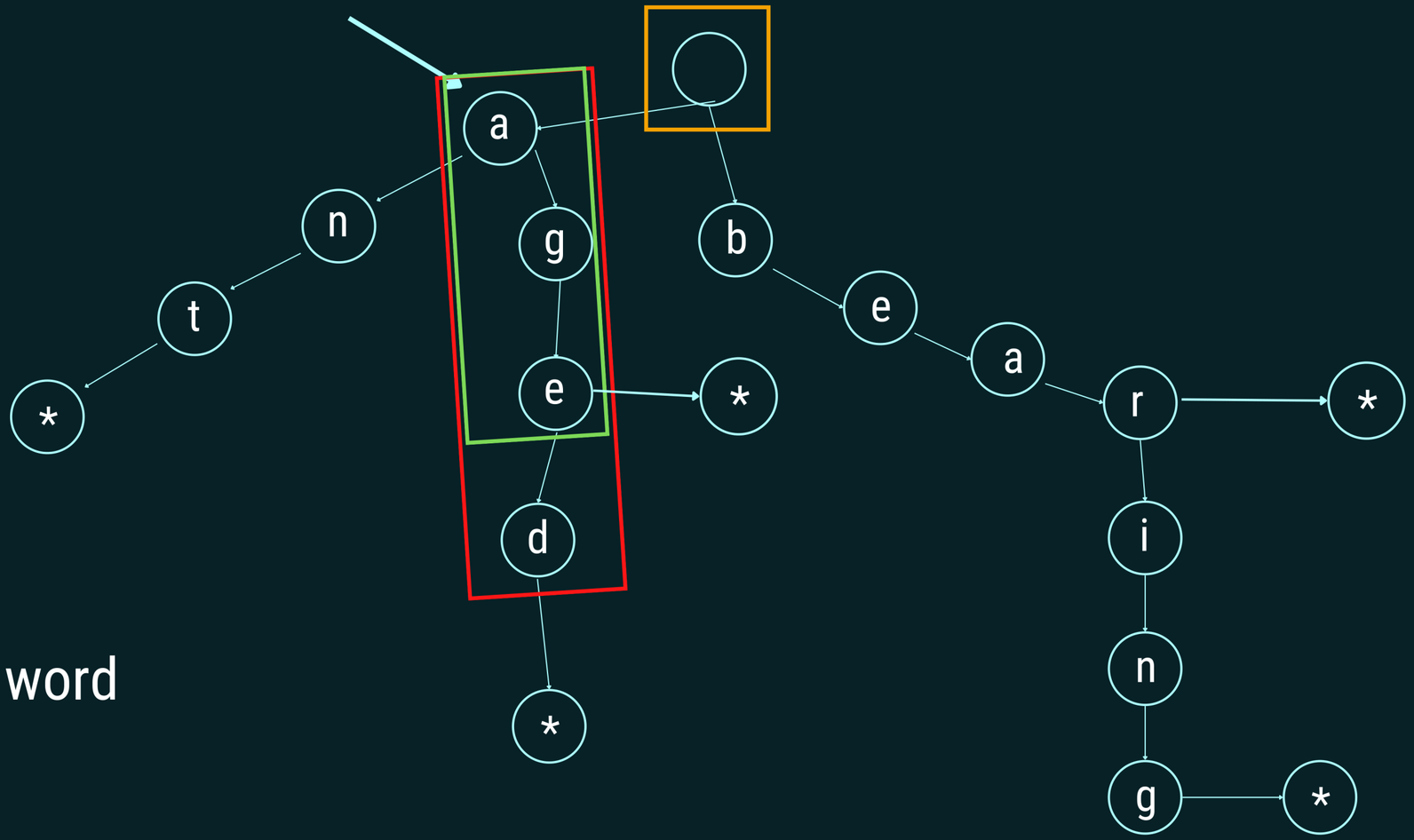
exit

word is a prefix to another word

a g e

aged is dependent to age

tail prefix

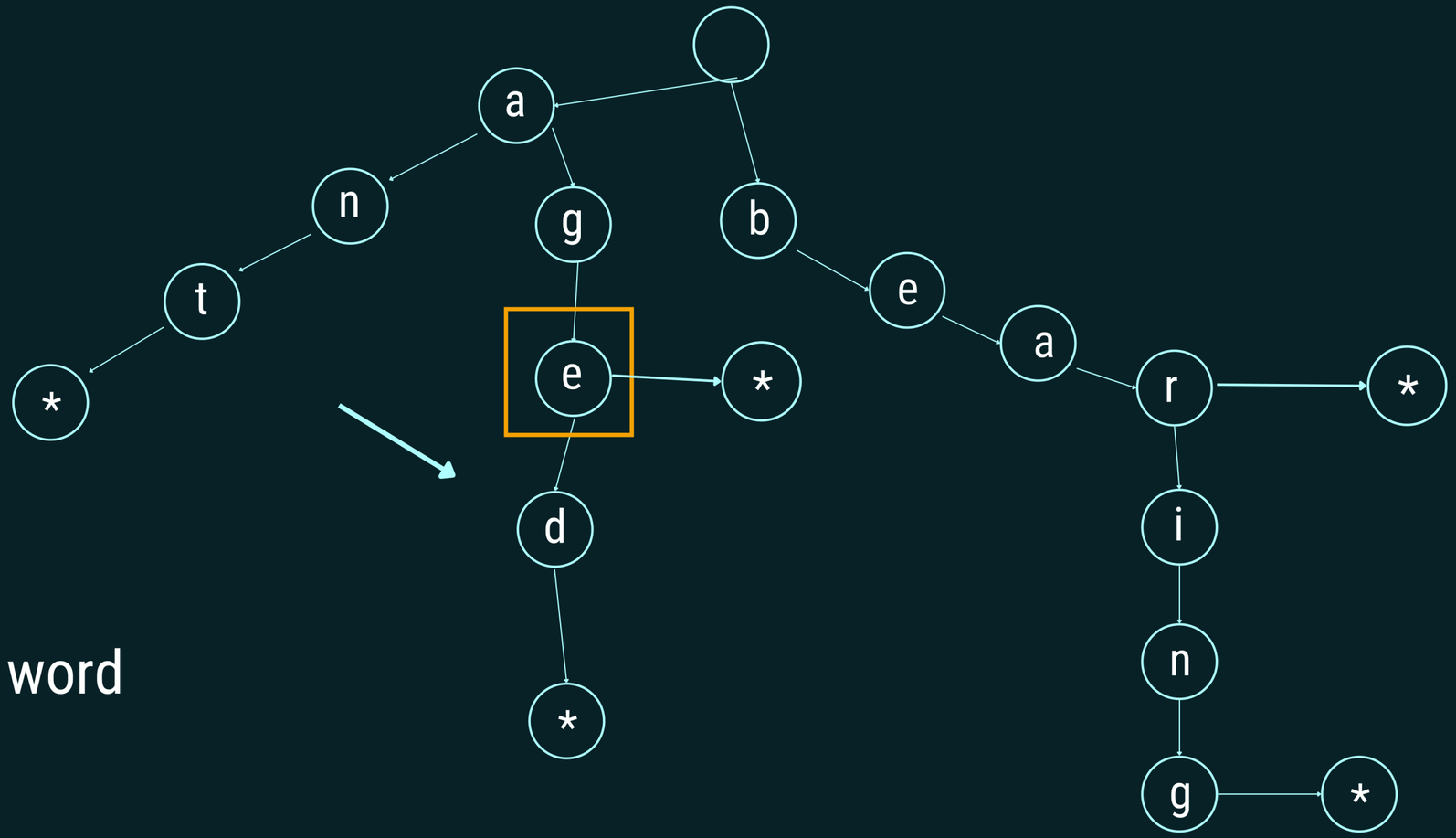


go to the last character of the word

using recursion

word is a prefix to another word

a g e

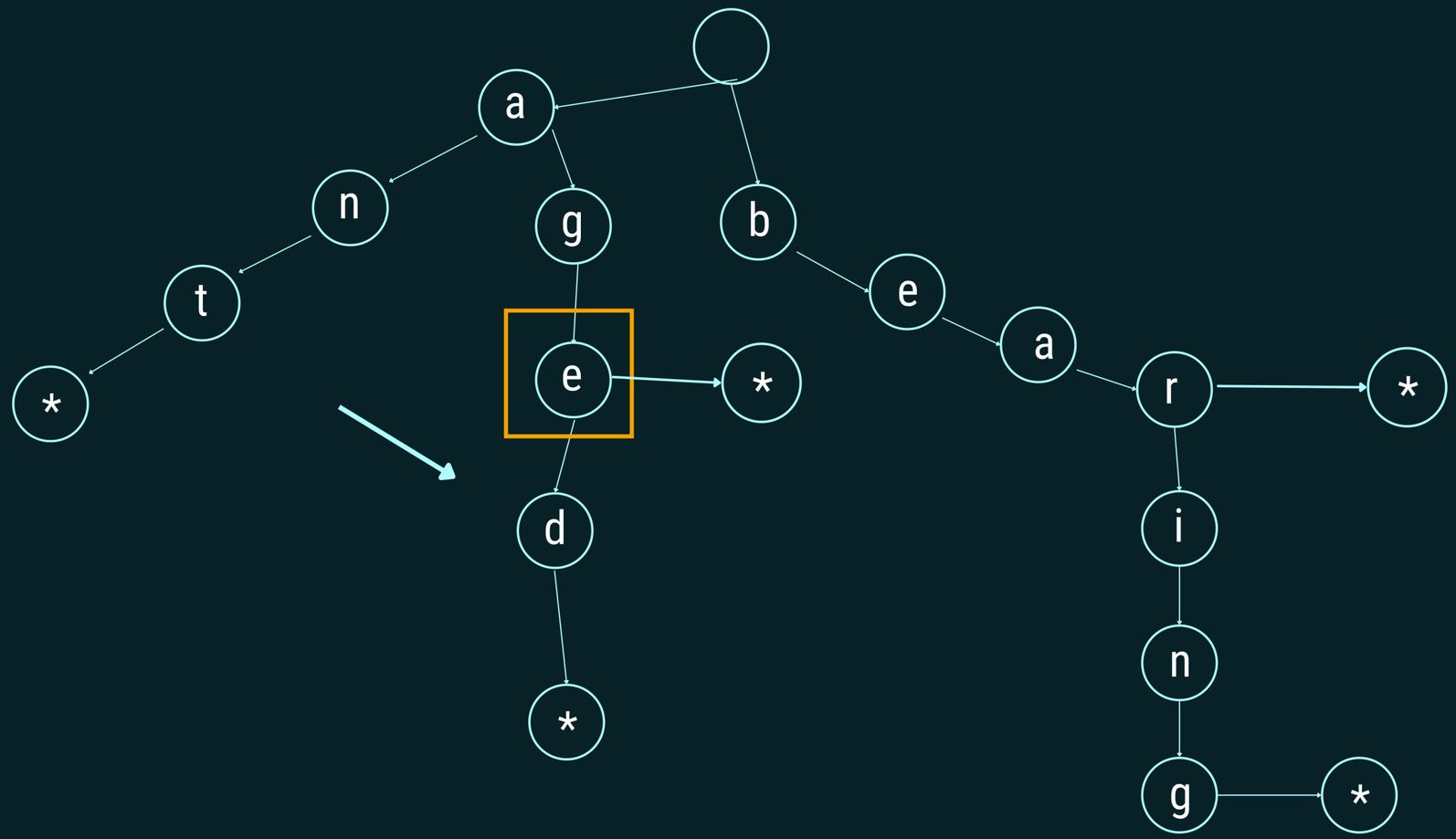


go to the last character of the word

using recursion

word is a prefix to another word

a g e



check if it is the end of word

end of word = 0

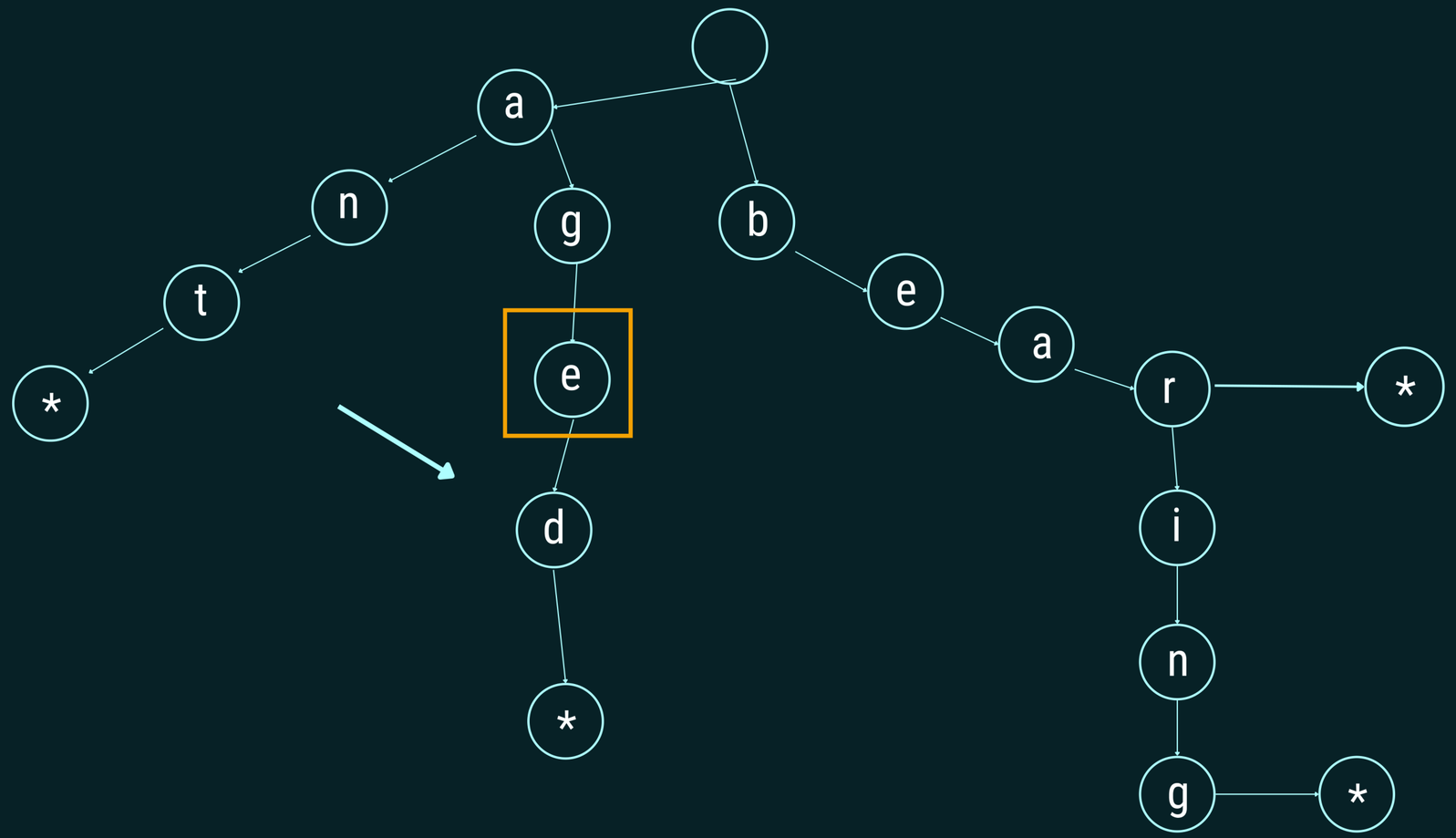
check if it has no children

delete node

exit

word is a prefix to another word

a g e



check if it is the end of word

end of word = 0

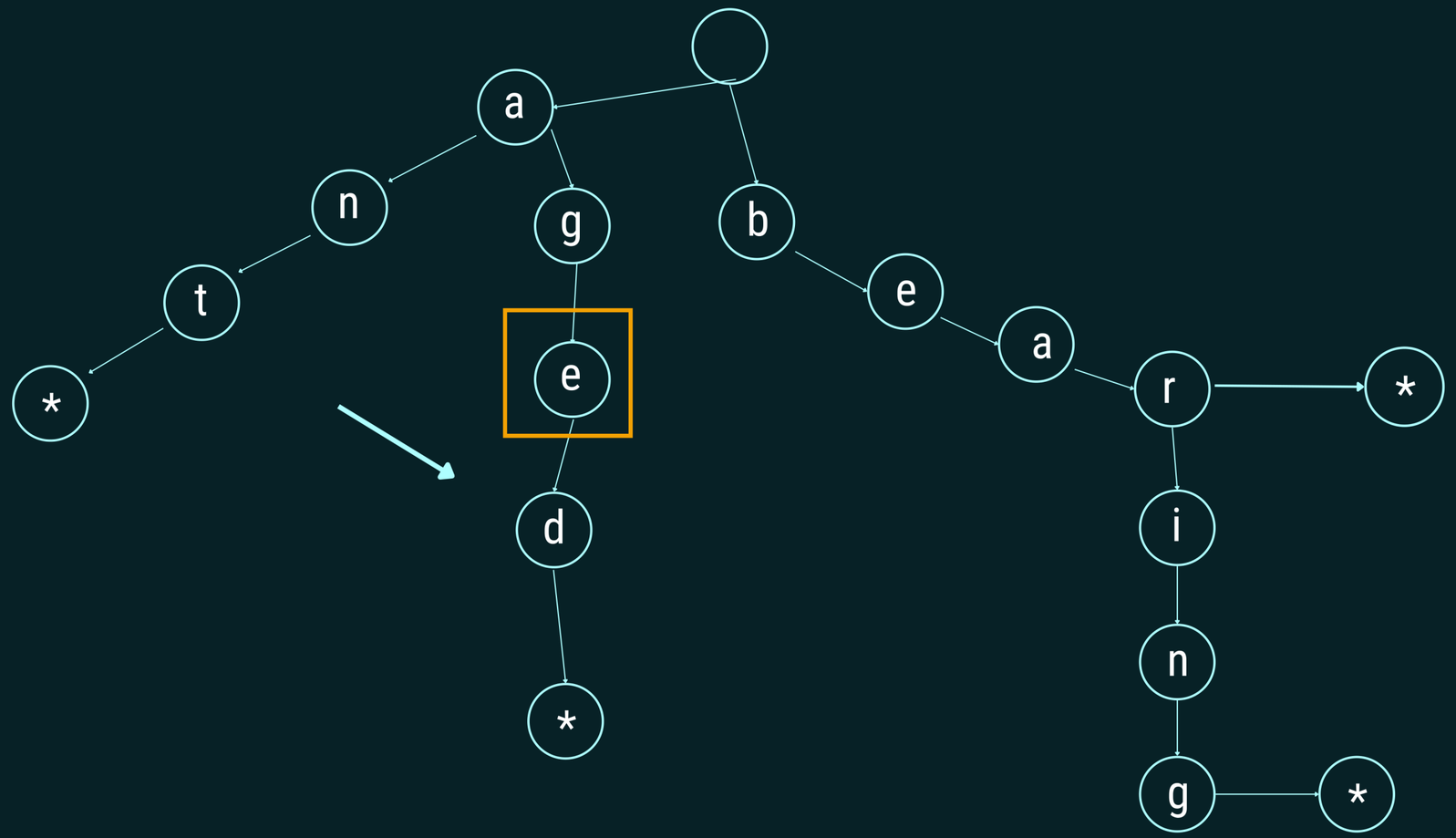
check if it has no children

delete node

exit

word is a prefix to another word

a g e



check if it is the end of word

end of word = 0

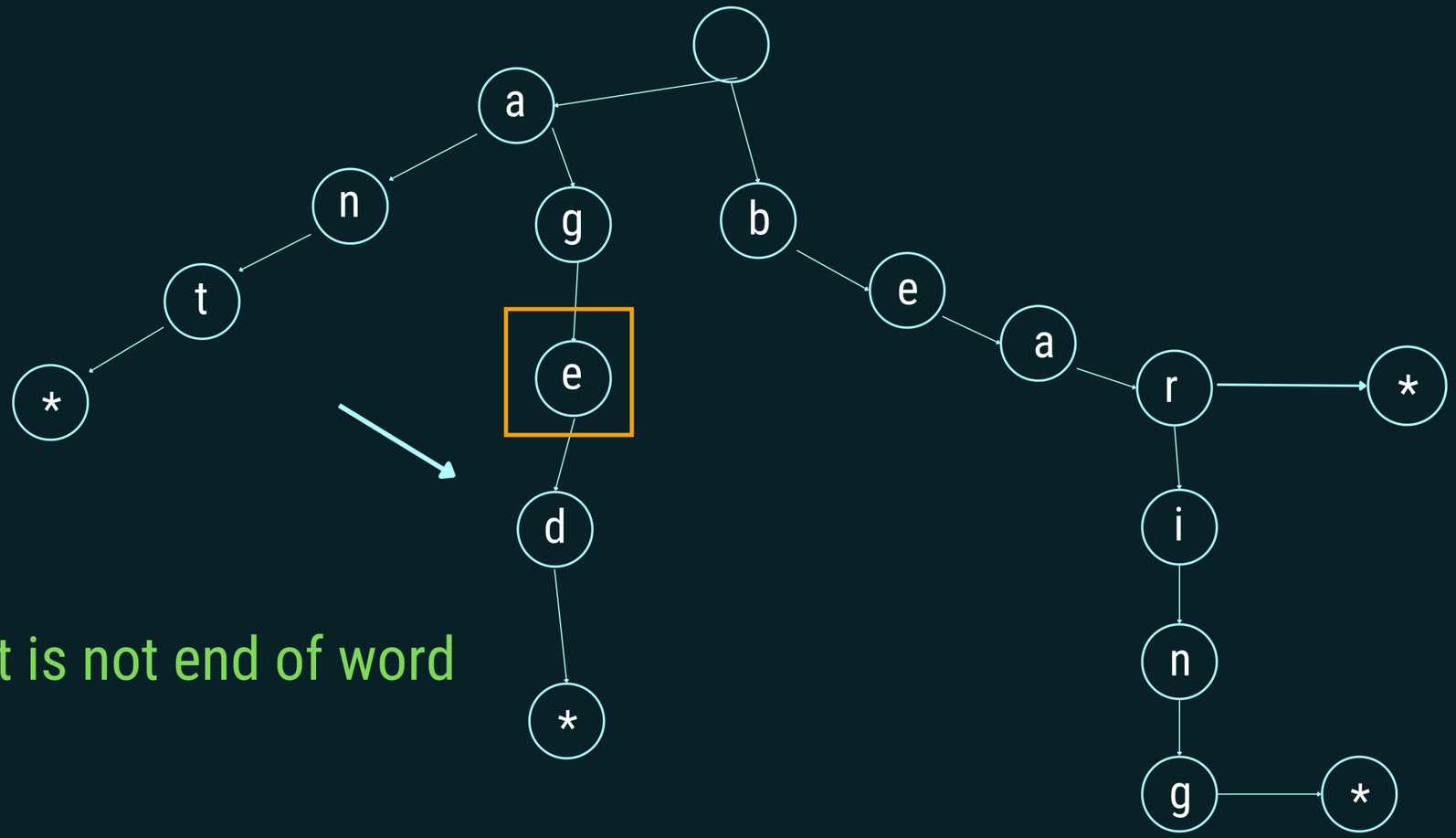
check if it has no children

delete node

exit

word is a prefix to another word

a g e



check if it has no children AND it is not end of word

delete the node

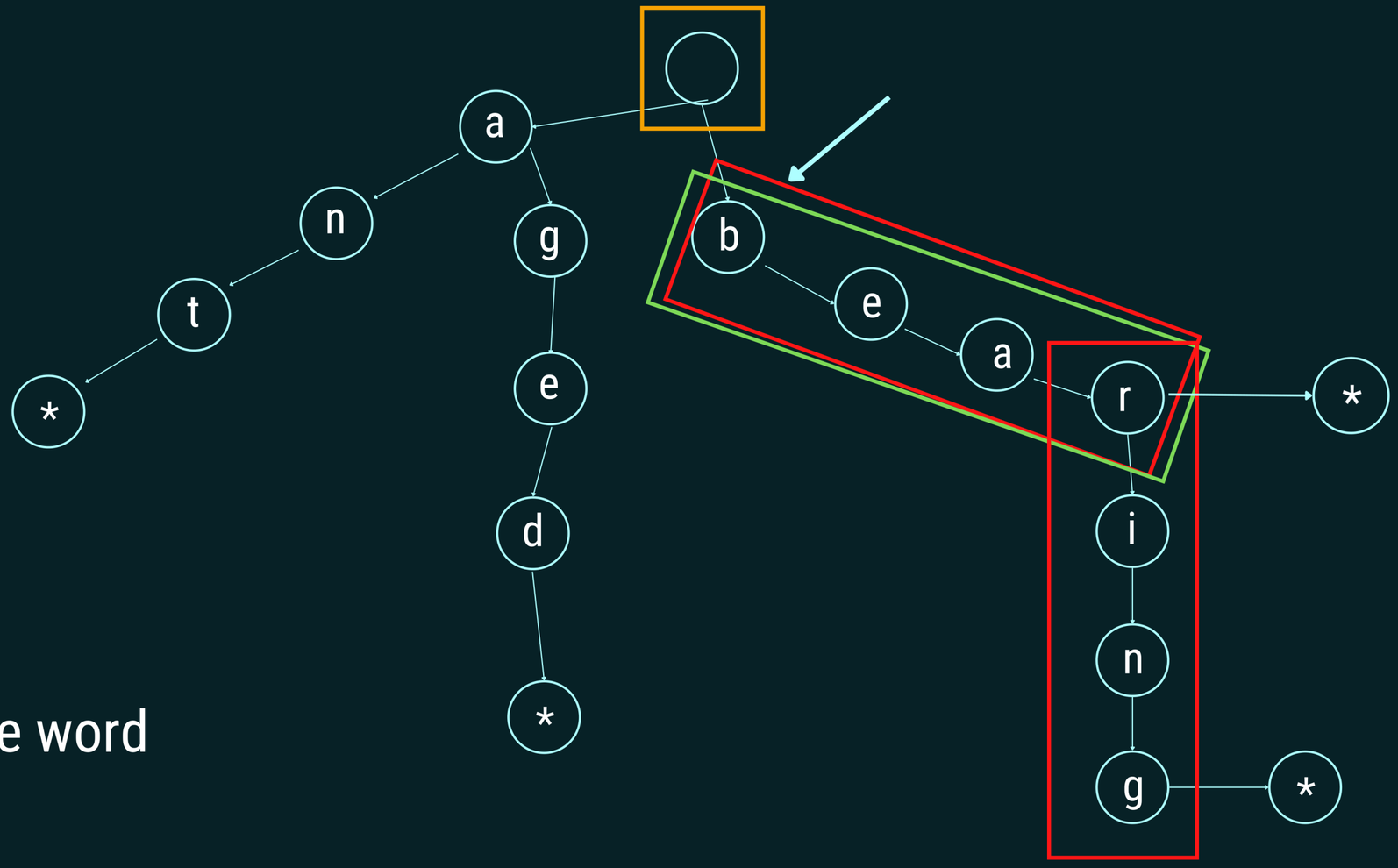
exit

word is a tail

b e a r i n g

bearing is dependent to bear

tail prefix

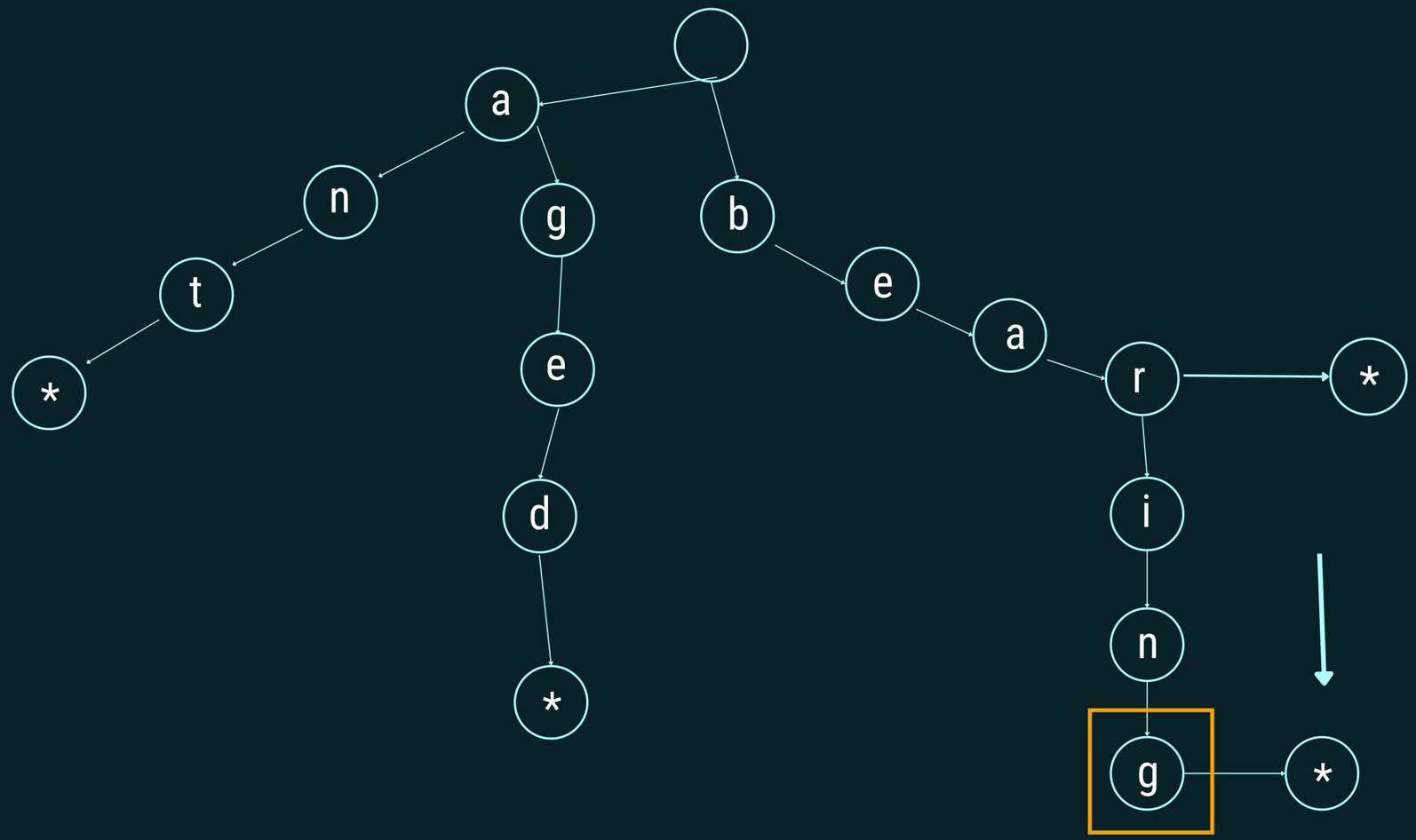


go to the last character of the word

using recursion

word is a tail

b e a r i n g



check if it is the end of word

end of word = 0

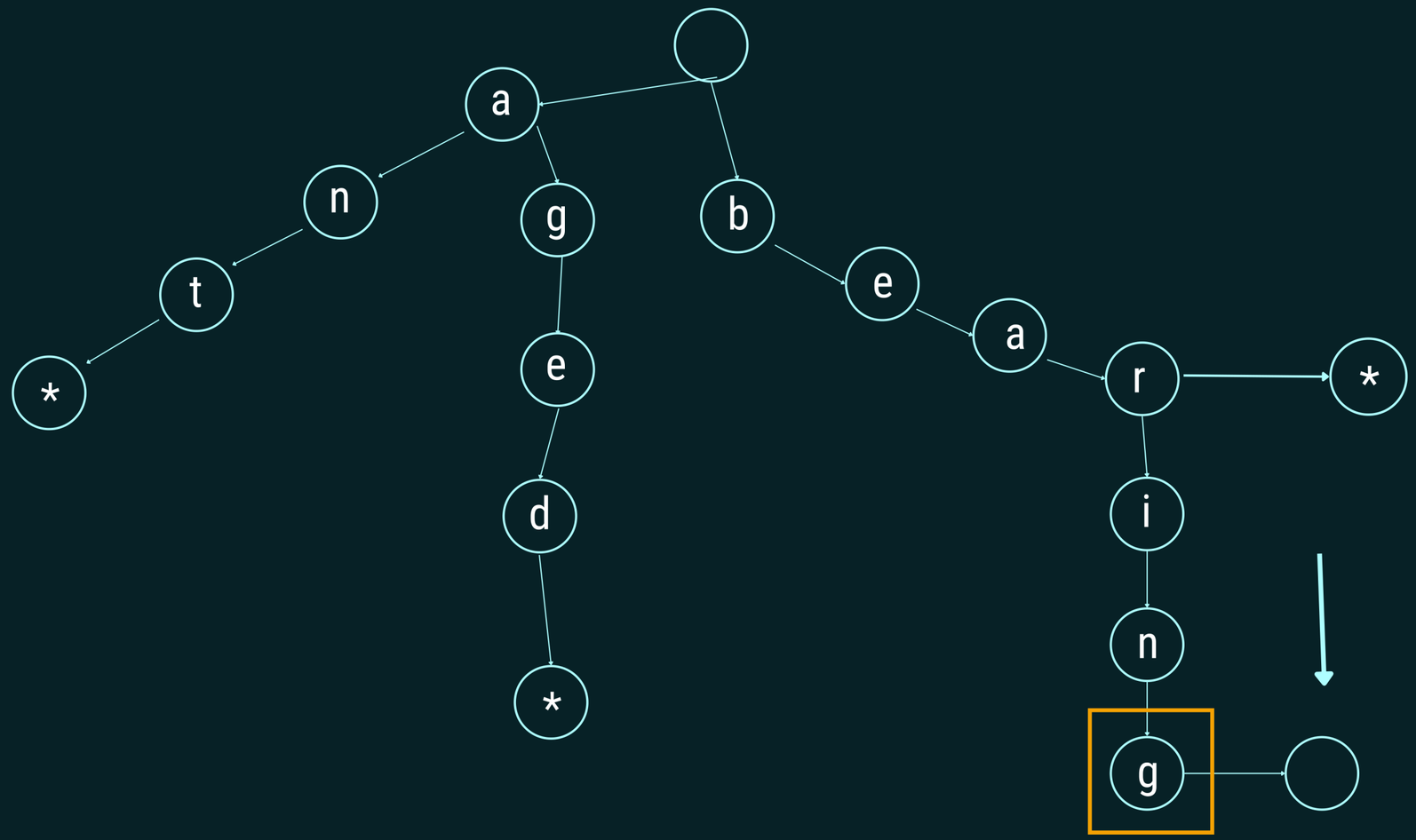
check if it has no children

delete node

exit

word is a tail

b e a r i n g



check if it is the end of word

end of word = 0

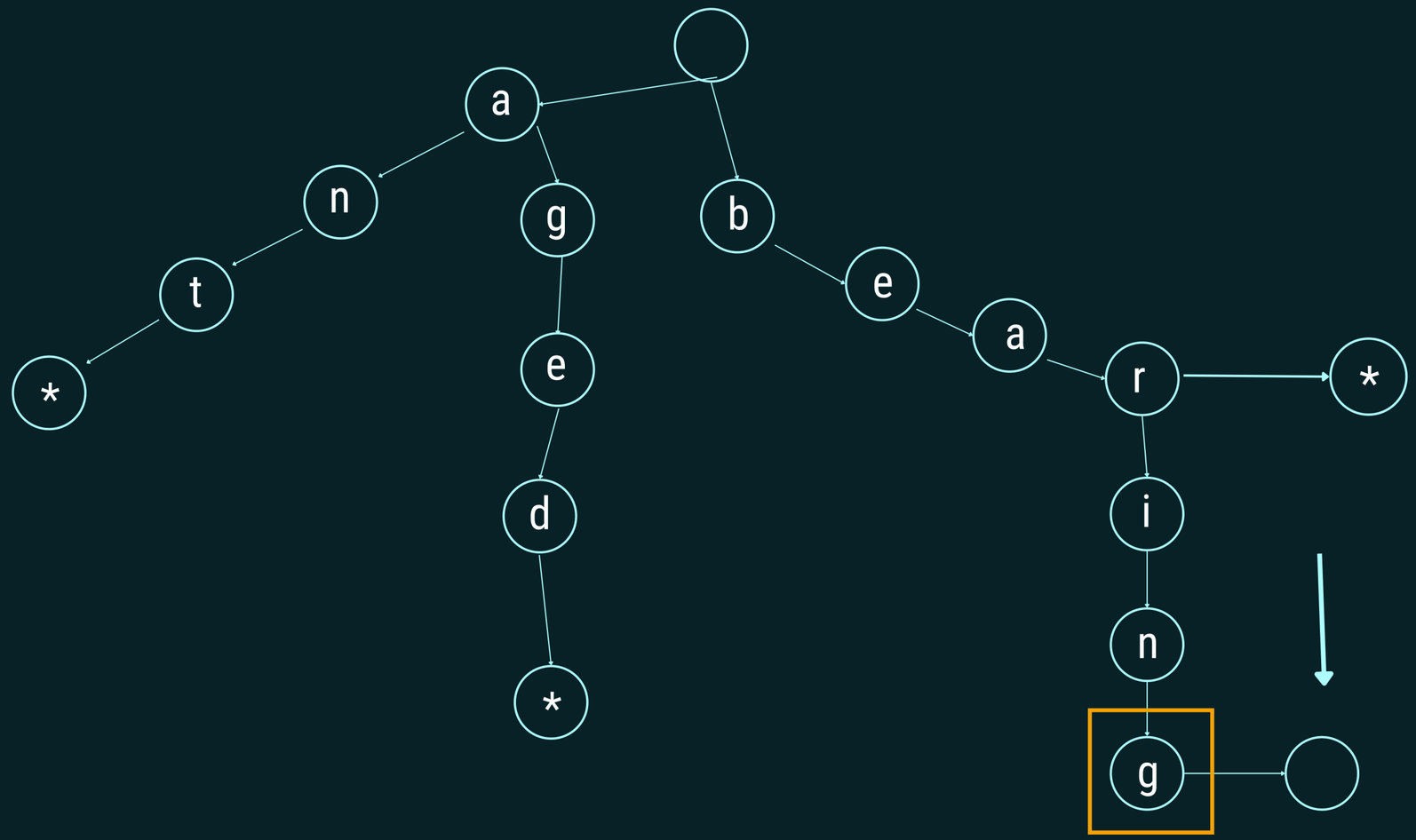
check if it has no children

delete node

exit

word is a tail

b e a r i n g



check if it is the end of word

end of word = 0

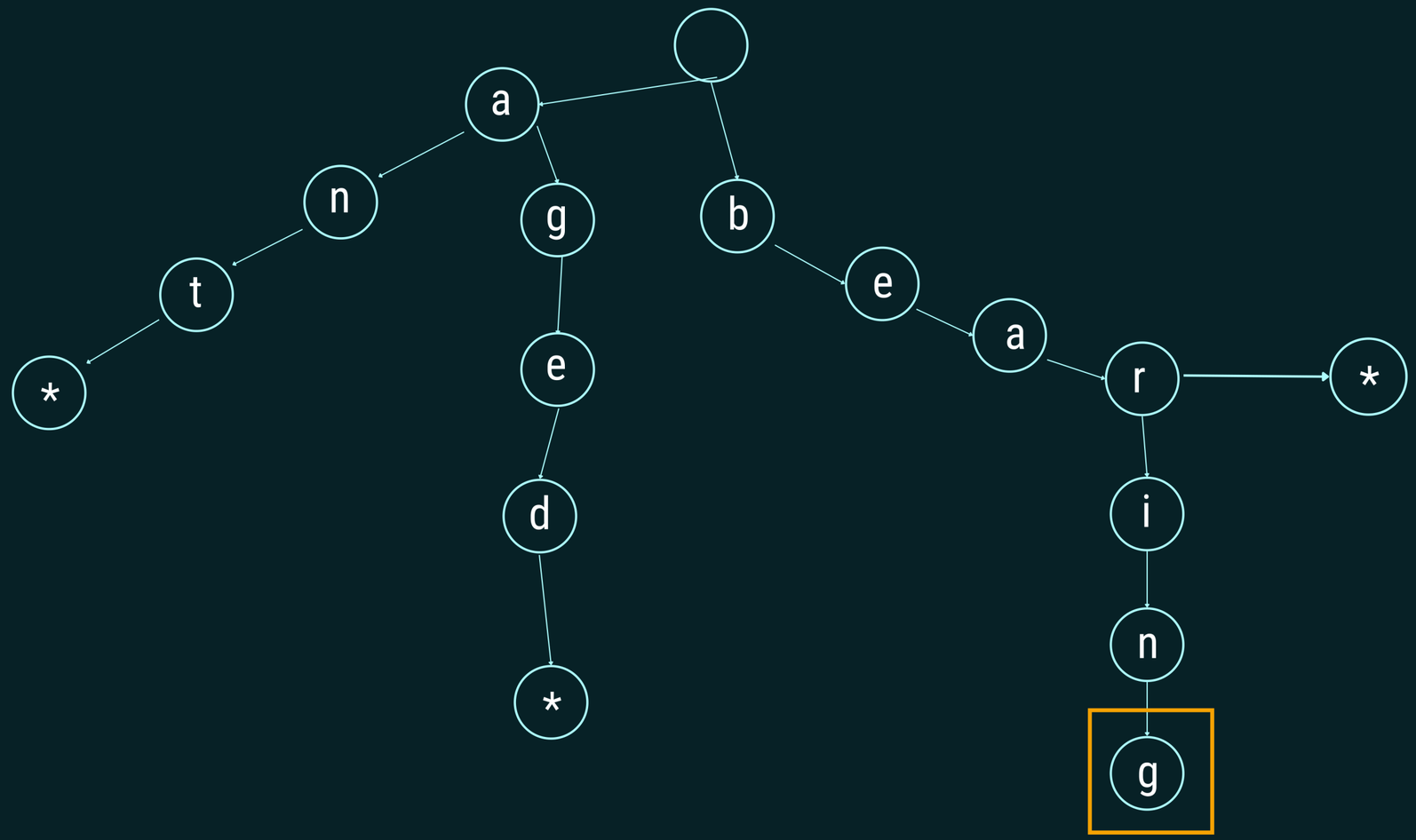
check if it has no children

delete node

exit

word is a tail

b e a r i n g



check if it is the end of word

end of word = 0

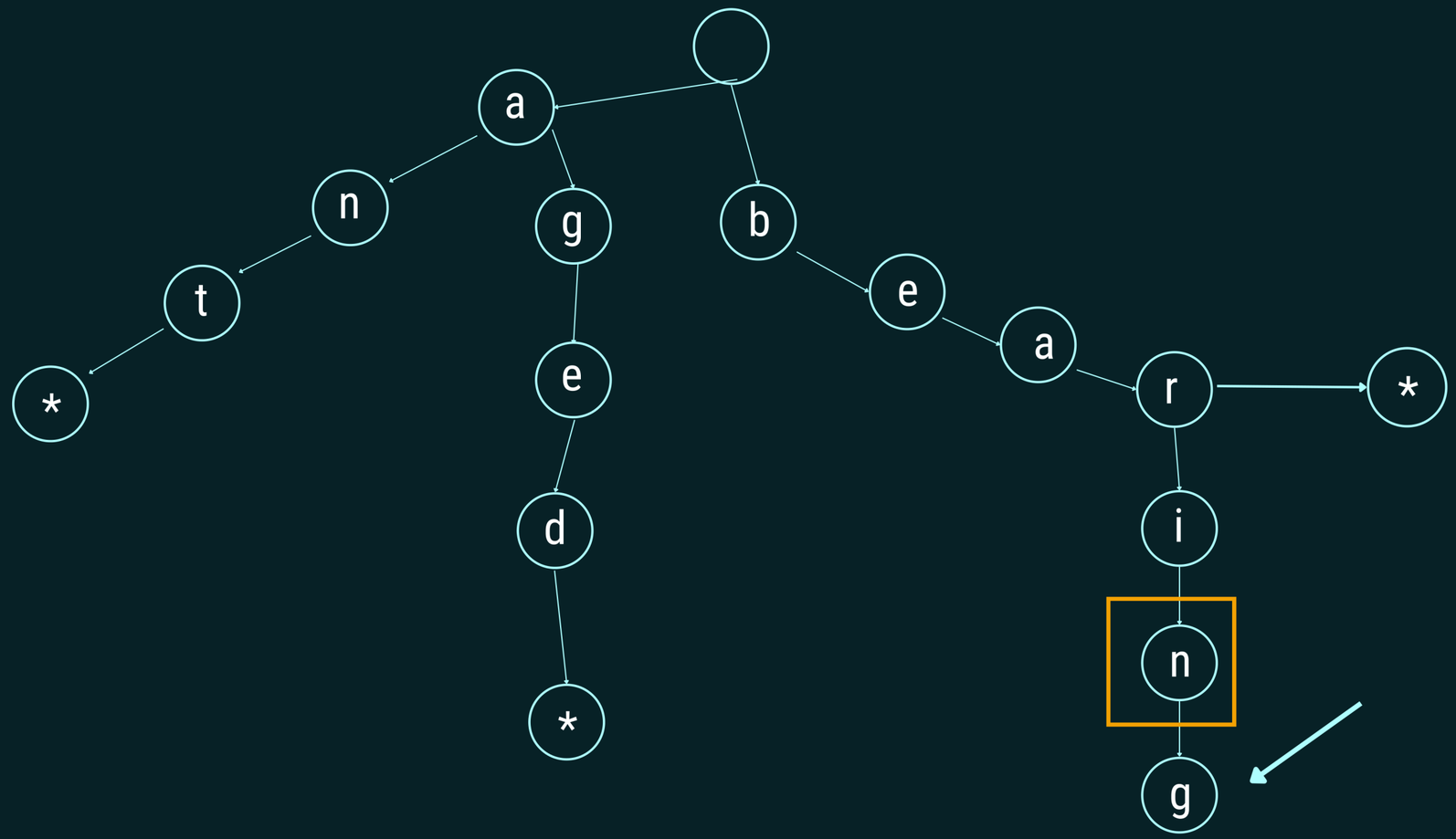
check if it has no children

delete node

exit

word is a tail

b e a r i n g



check if it is the end of word

end of word = 0

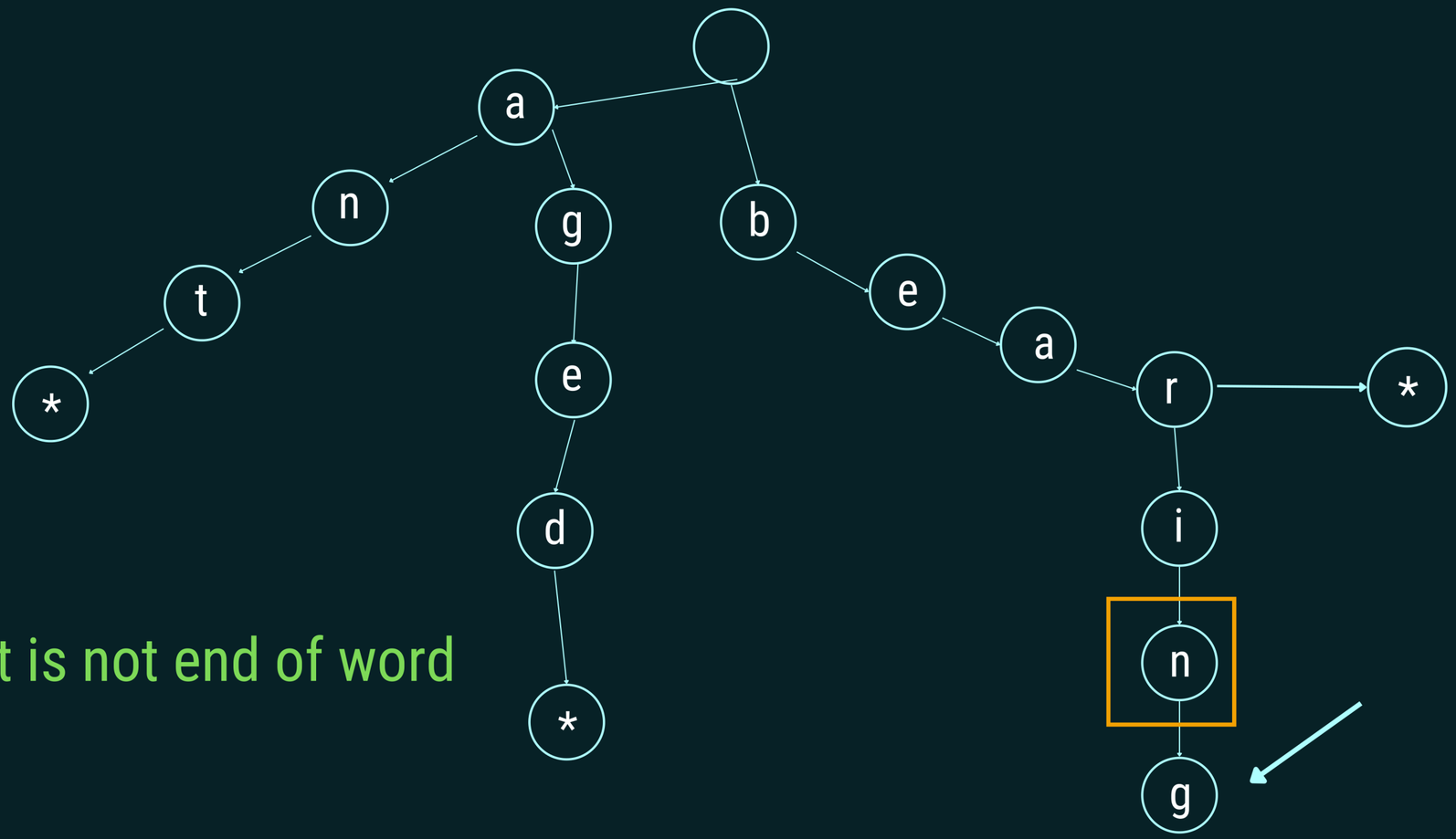
check if it has no children

delete node

exit

word is a tail

b e a r i n g



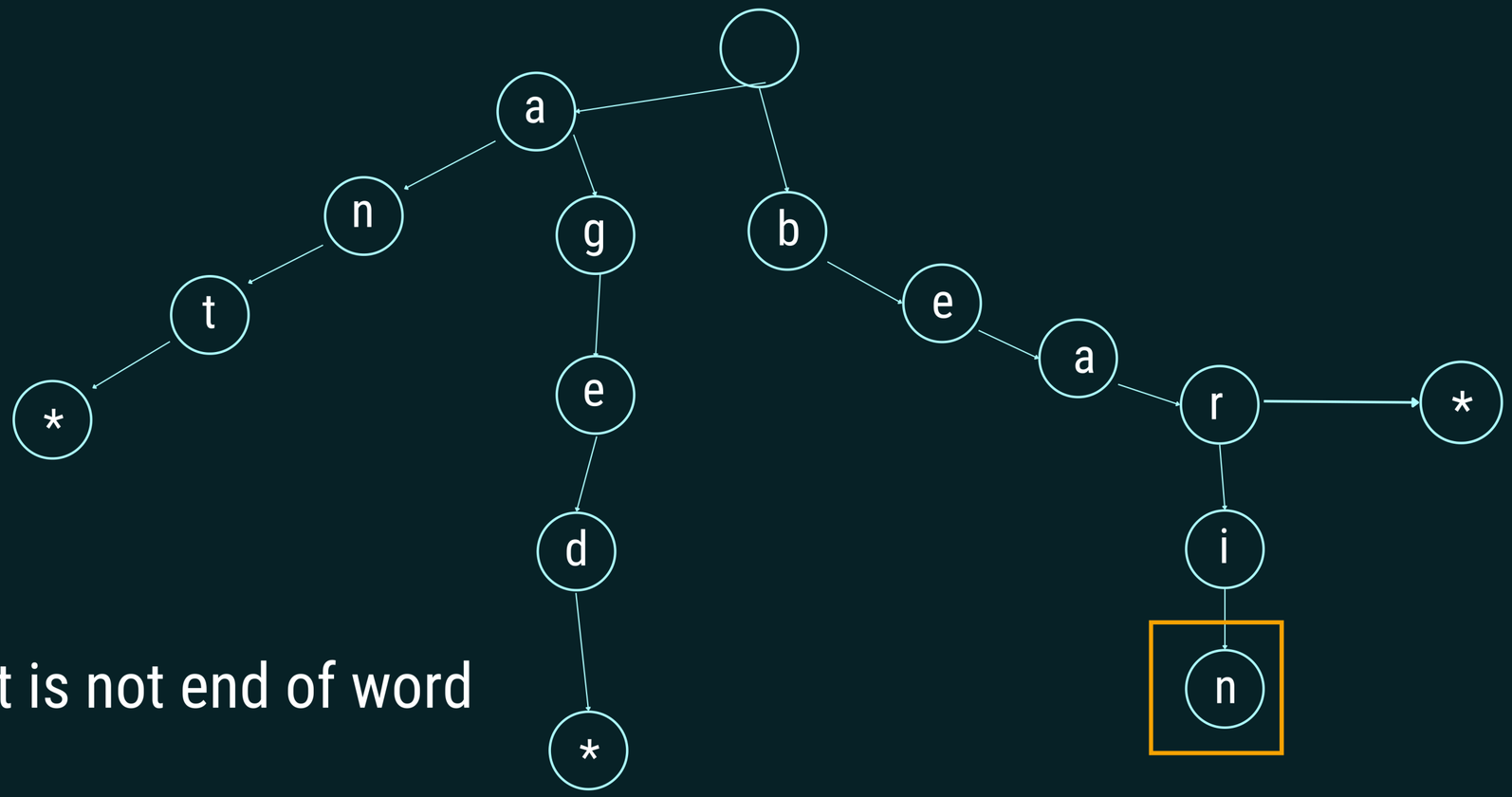
check if it has no children AND it is not end of word

delete the node

exit

word is a tail

b e a r i n g



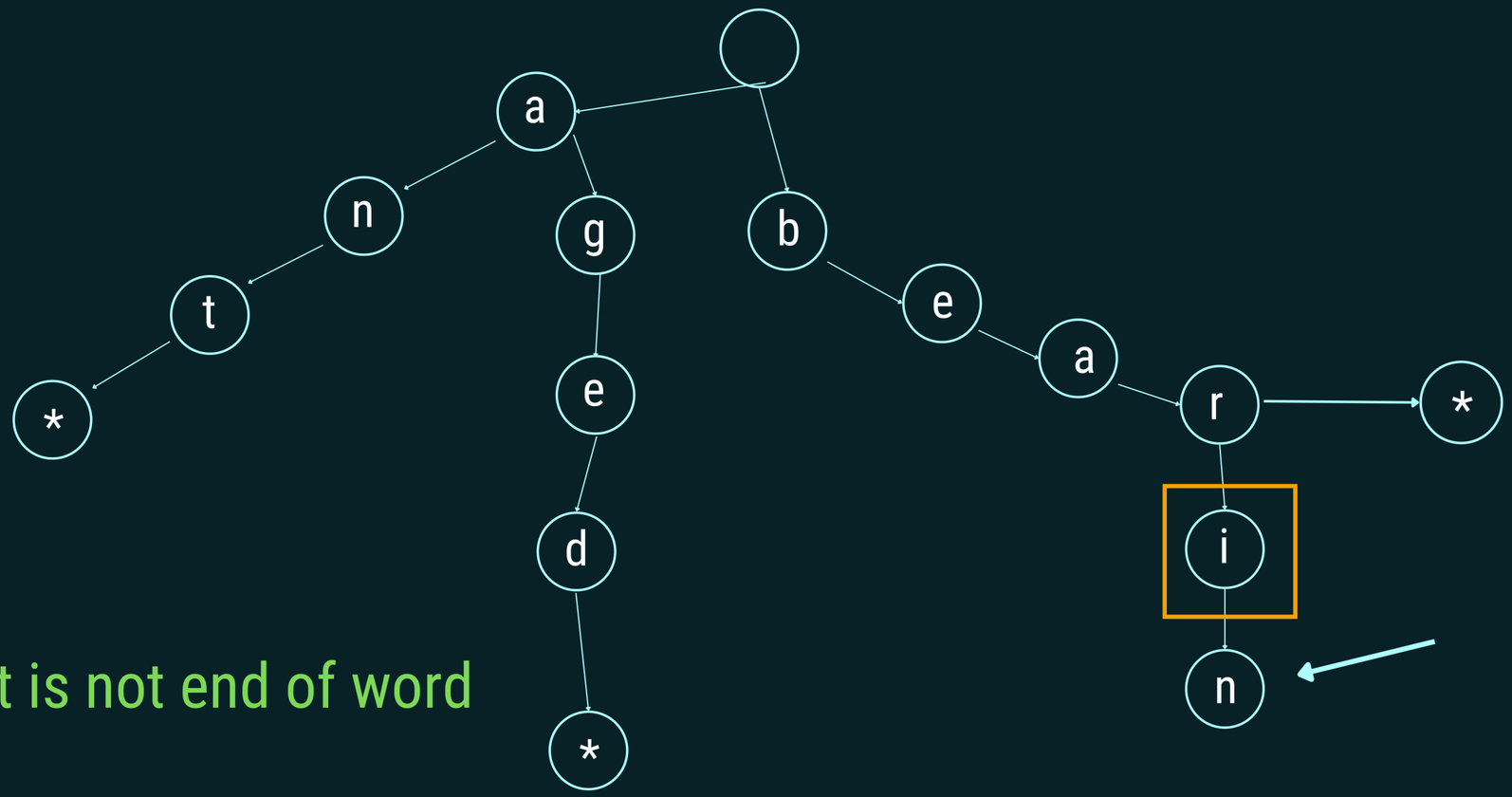
check if it has no children AND it is not end of word

delete the node

exit

word is a tail

b e a r i n g



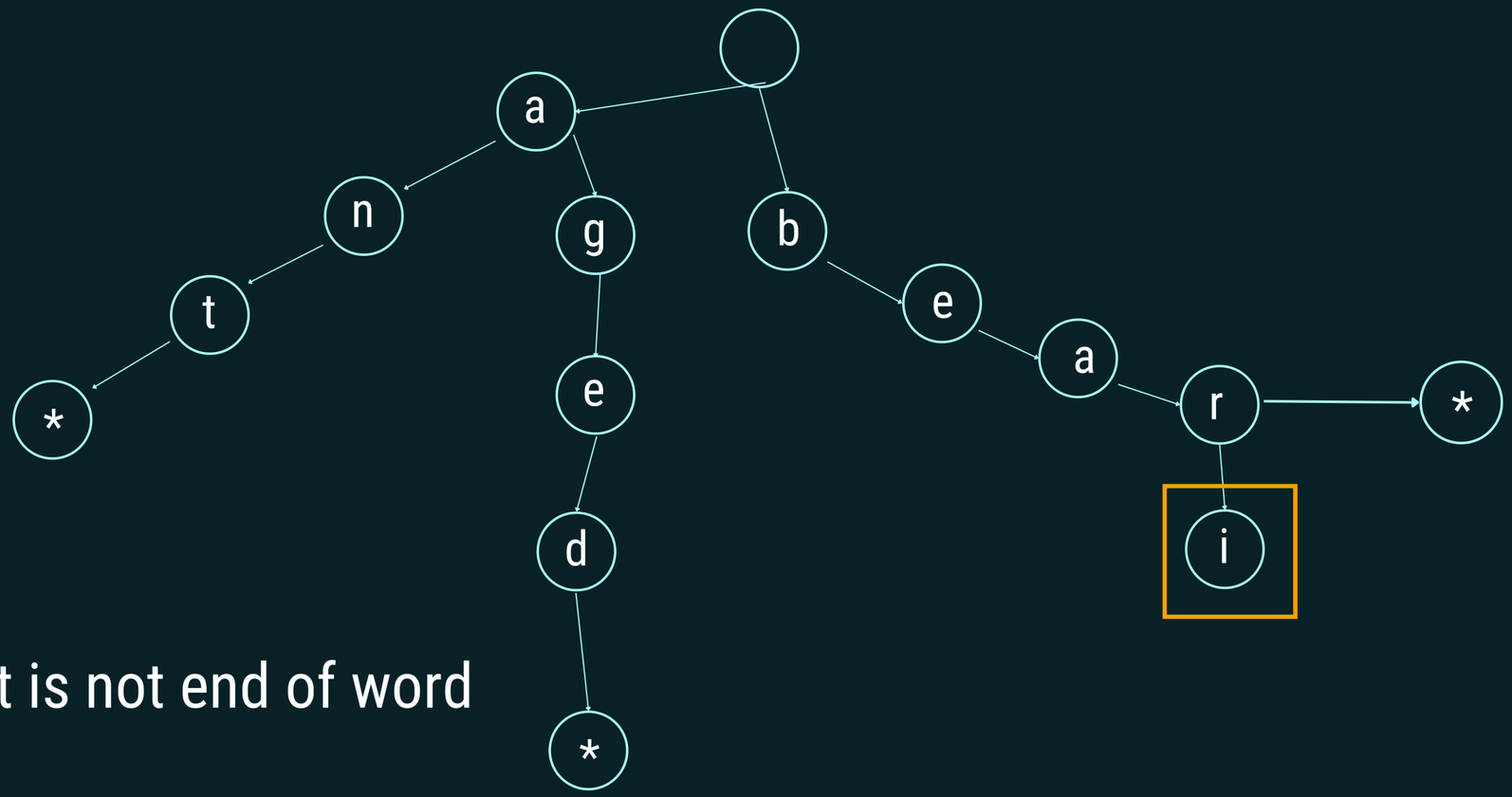
check if it has no children AND it is not end of word

delete the node

exit

word is a tail

b e a r i n g



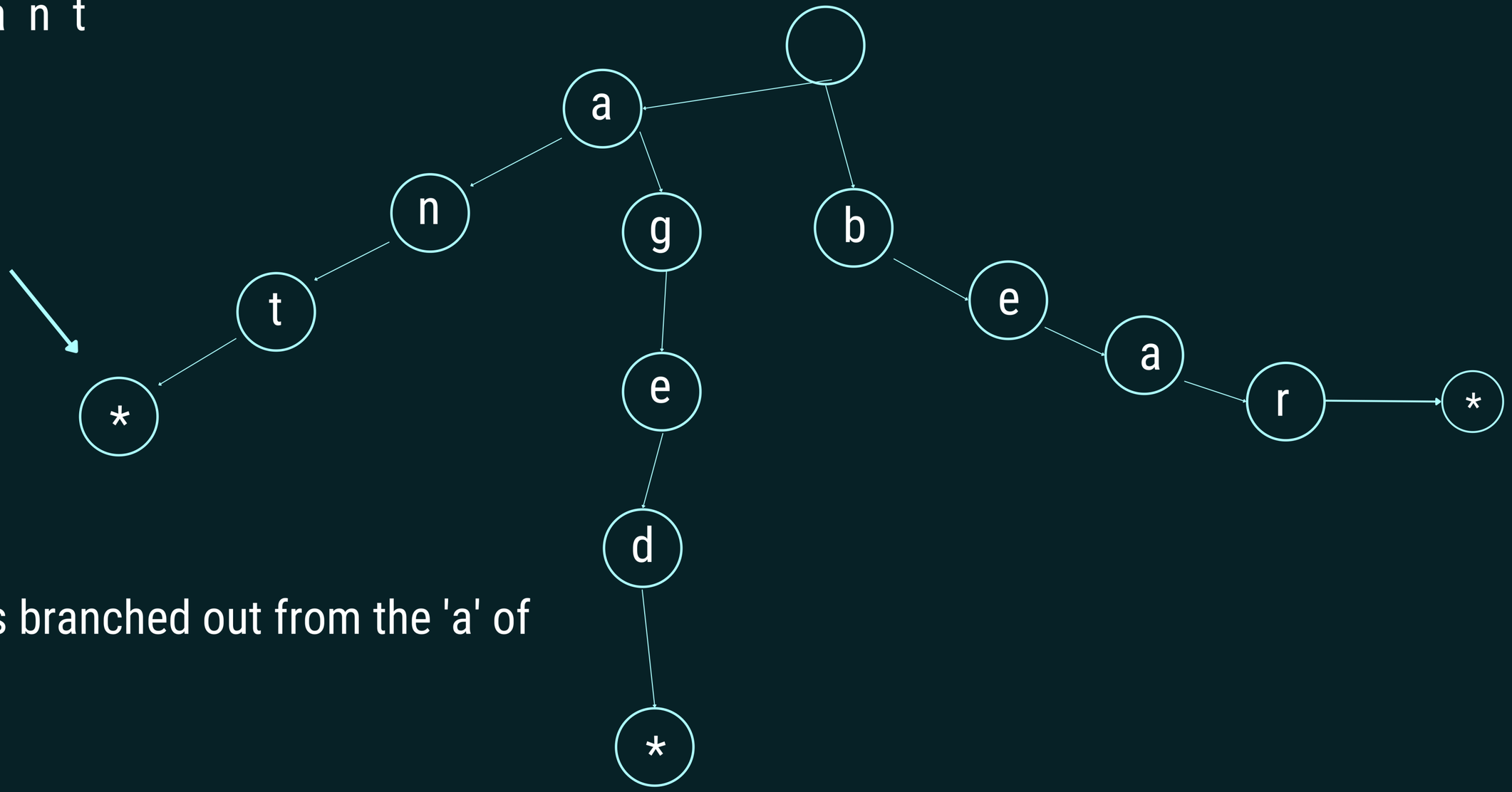
check if it has no children AND it is not end of word

delete the node

exit

word is a base case

a n t

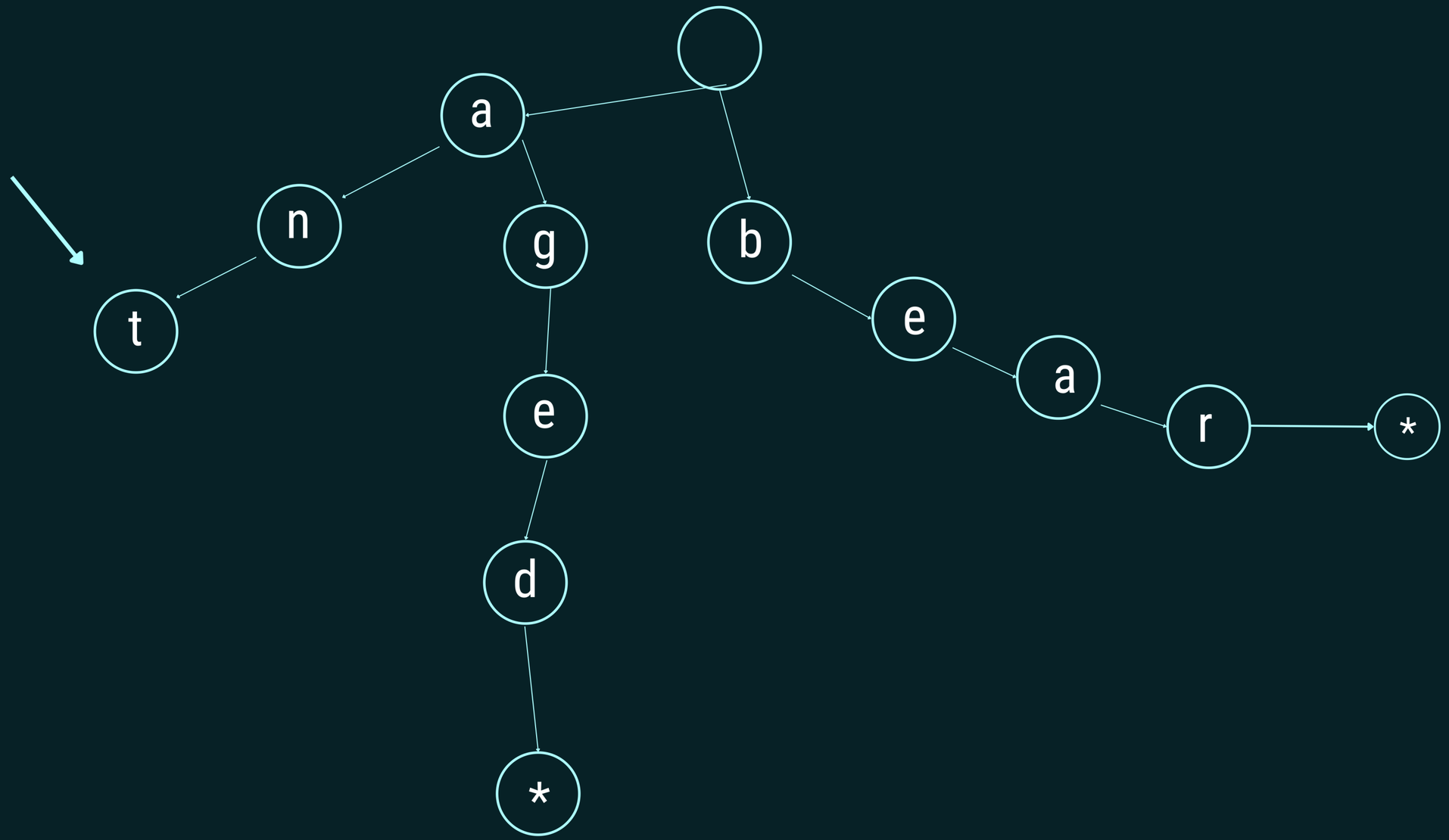


ant is branched out from the 'a' of

aged

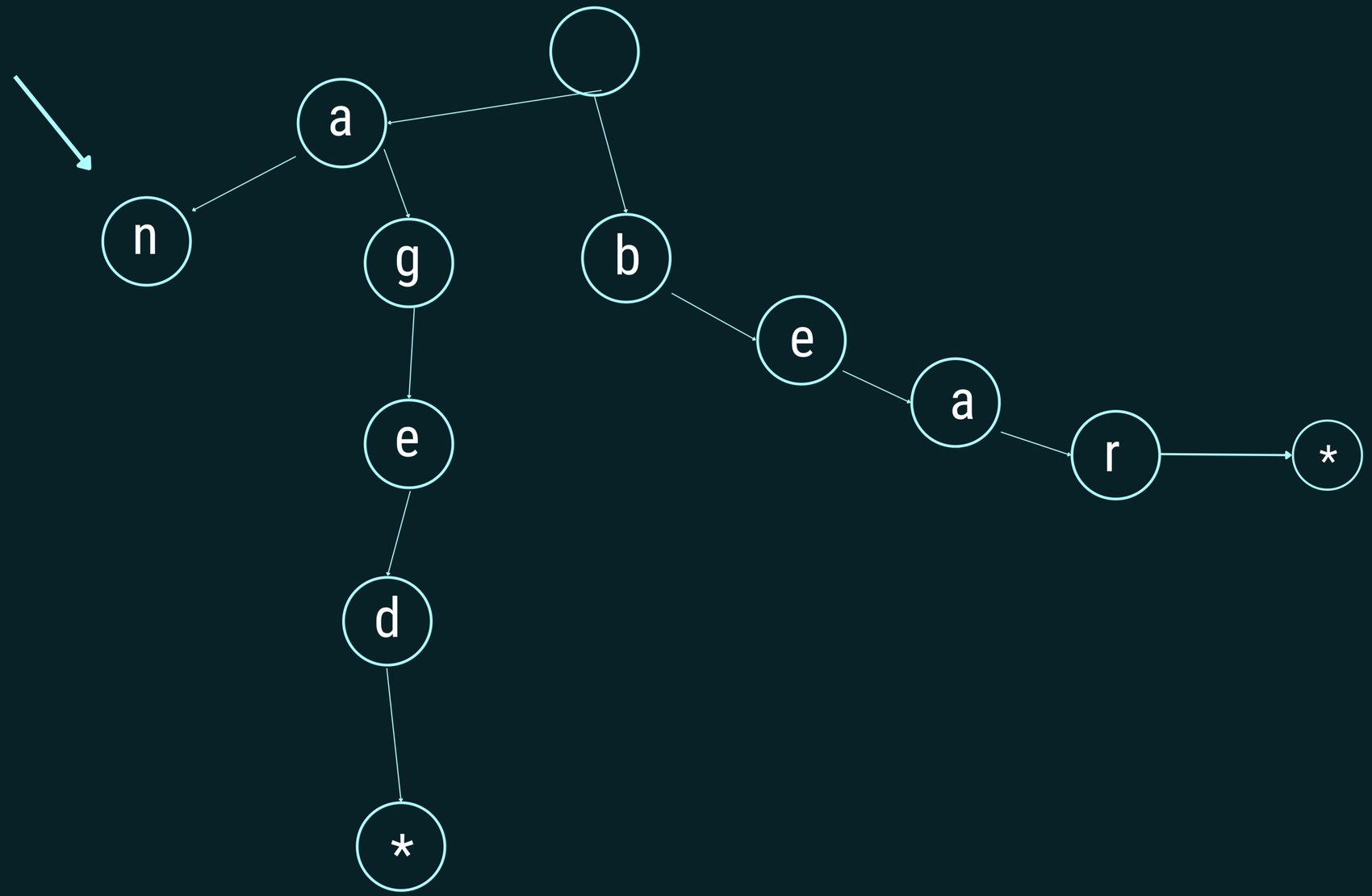
word is a base case

a n t



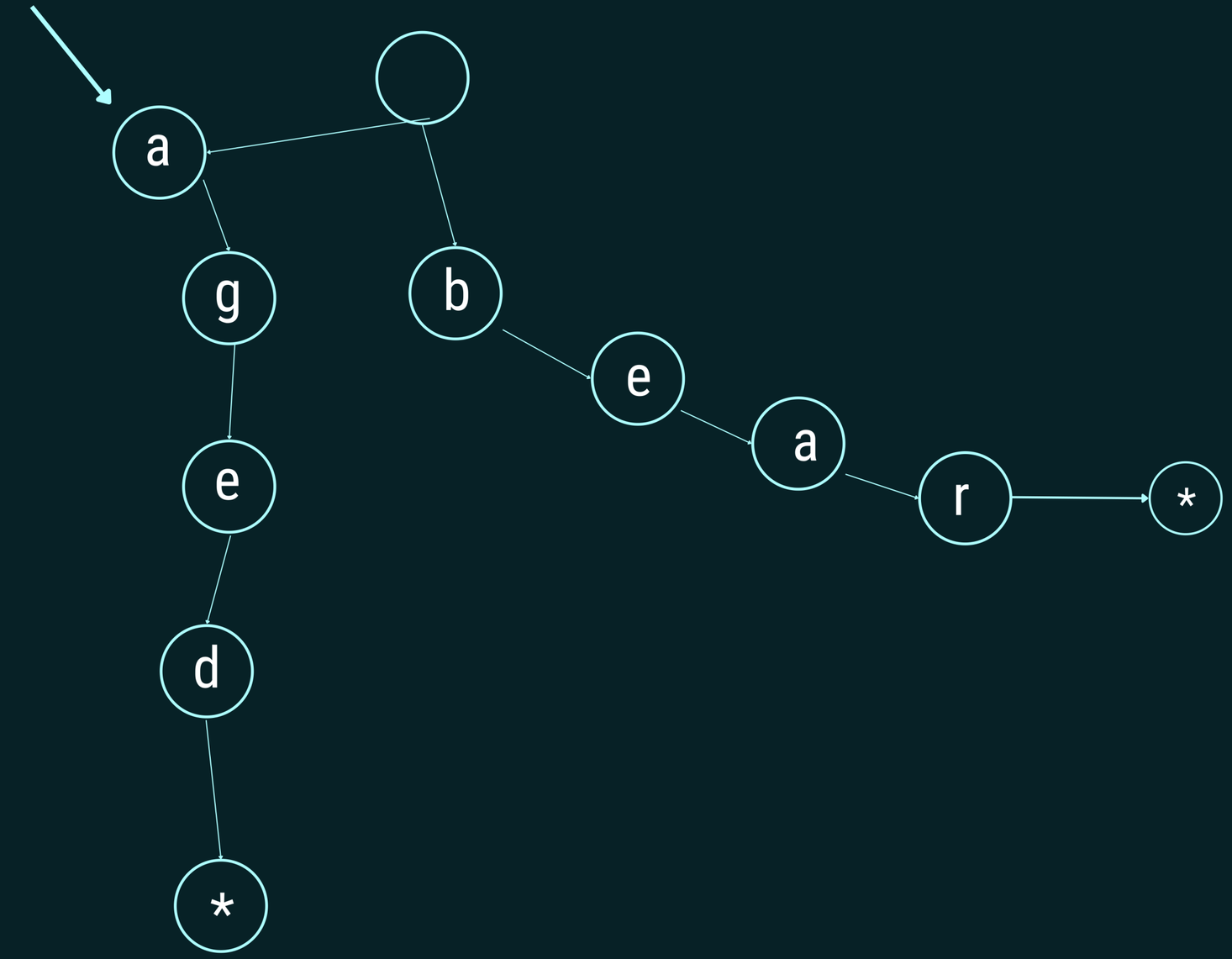
word is a base case

a n t



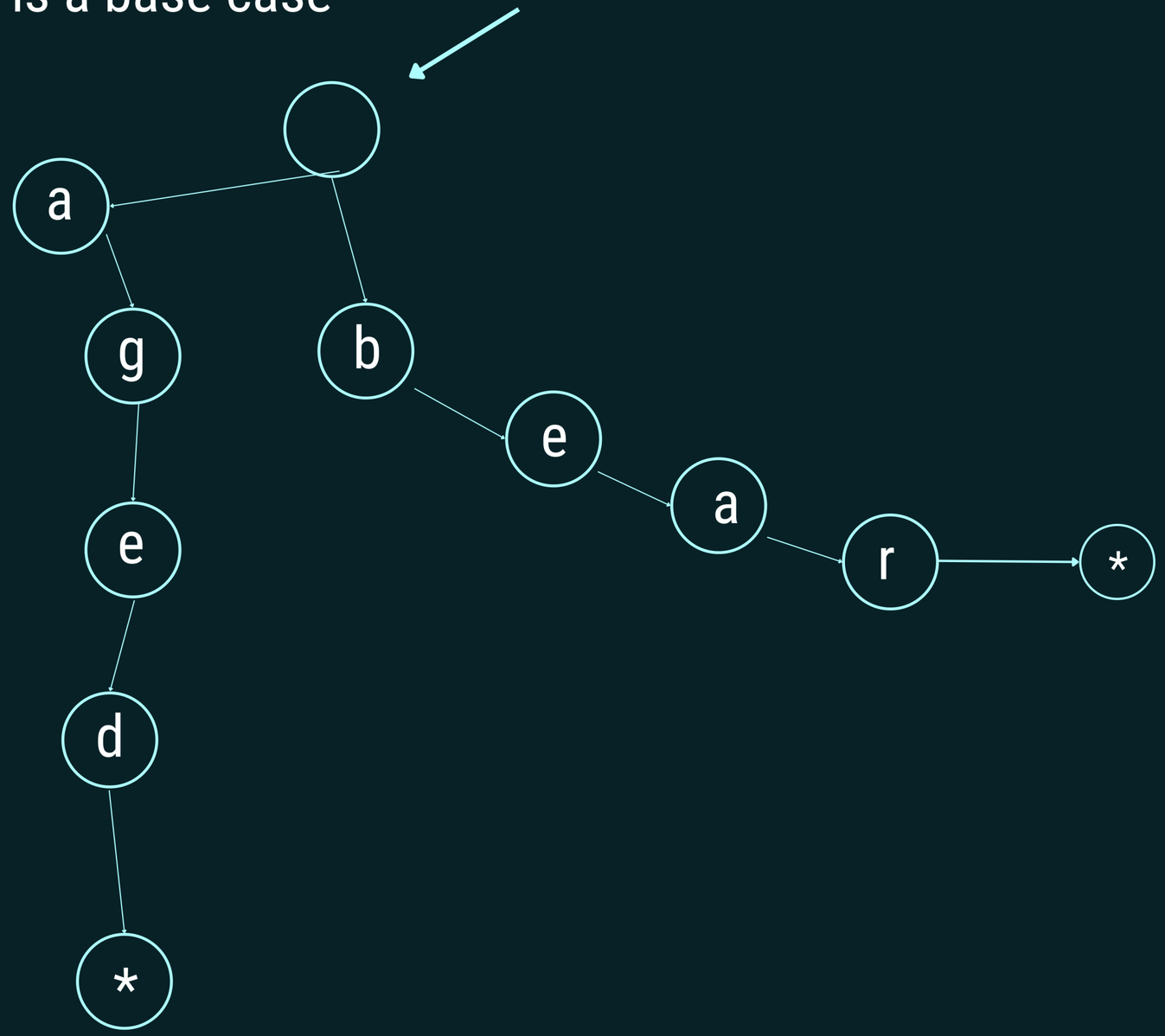
word is a base case

a n t



word is a base case

a n t



Radix Tree

Radix tree, also known as a compressed trie, is a space-optimized variant of a trie in which nodes with only one child get merged with its parents; elimination of branches of the nodes with a single child results in better in both space and time metrics.

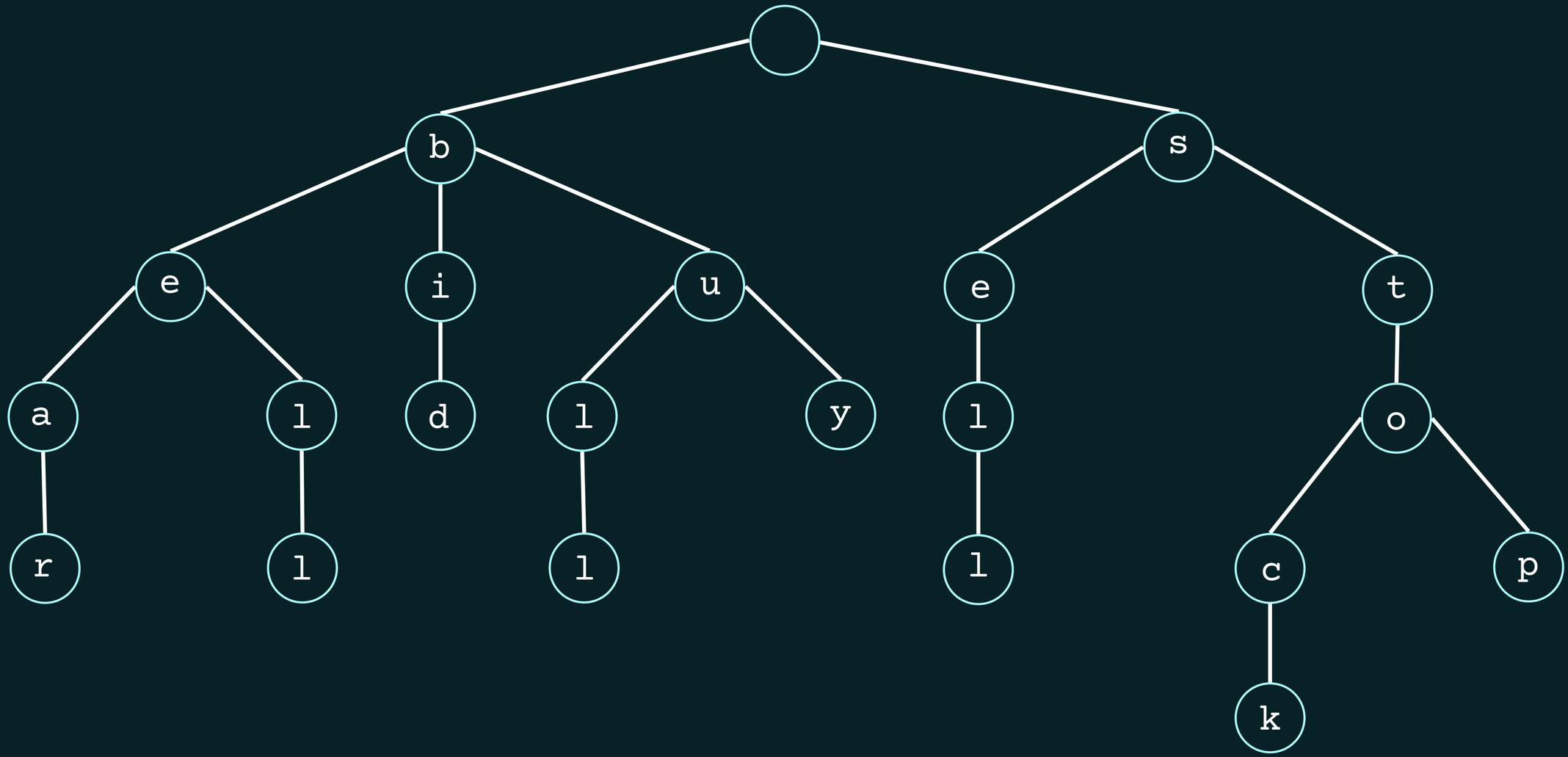
Why Compressed Tries?

A tree in which every node has at least 2 children has at most L-1 internal nodes, where L is the number of leaves.

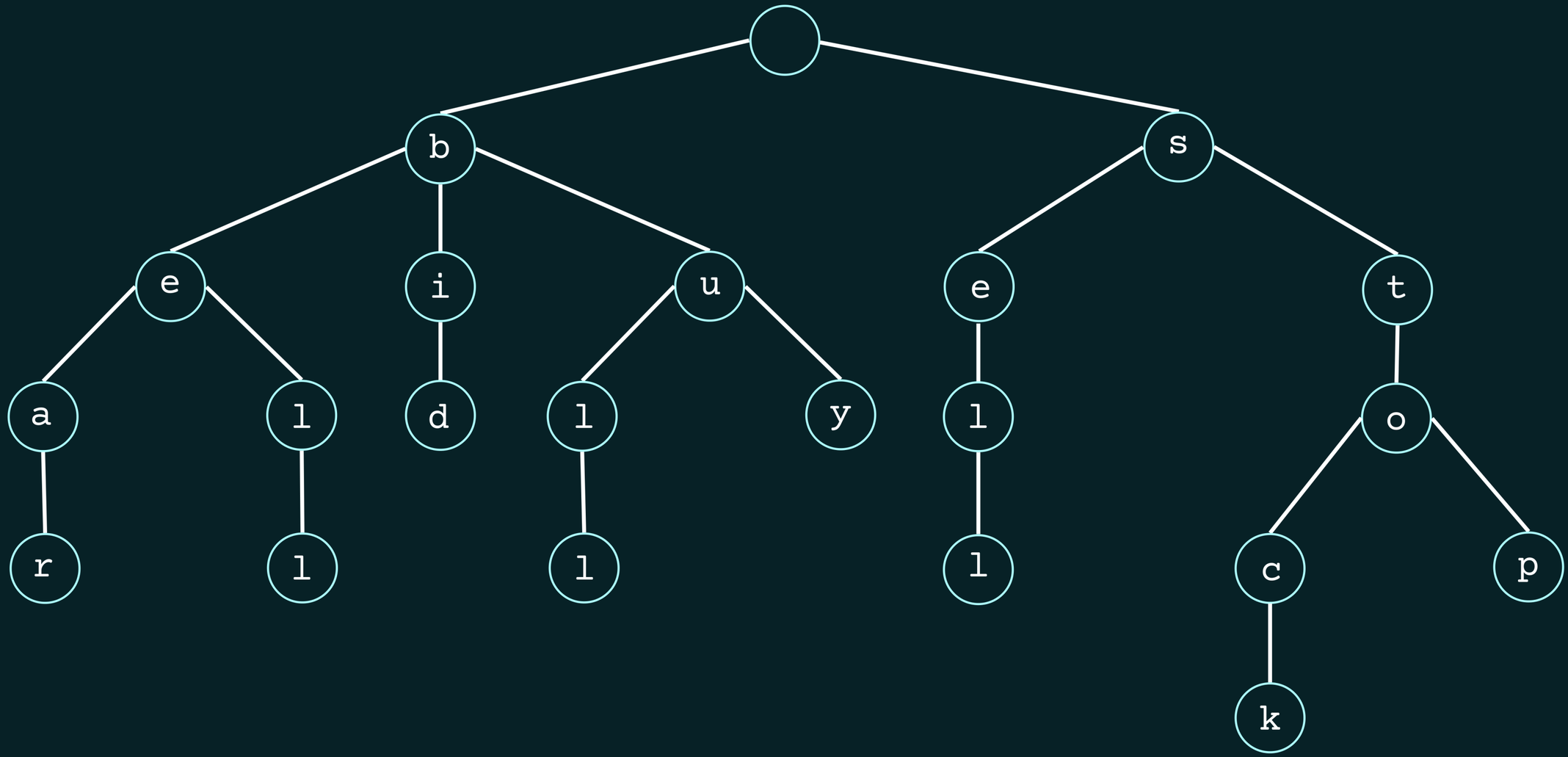
- A node may now have a longer label inside

{ bear, bell, bid, bull, buy, sell, stock, stop}

{ bear, bell, bid, bull, buy, sell, stock, stop}

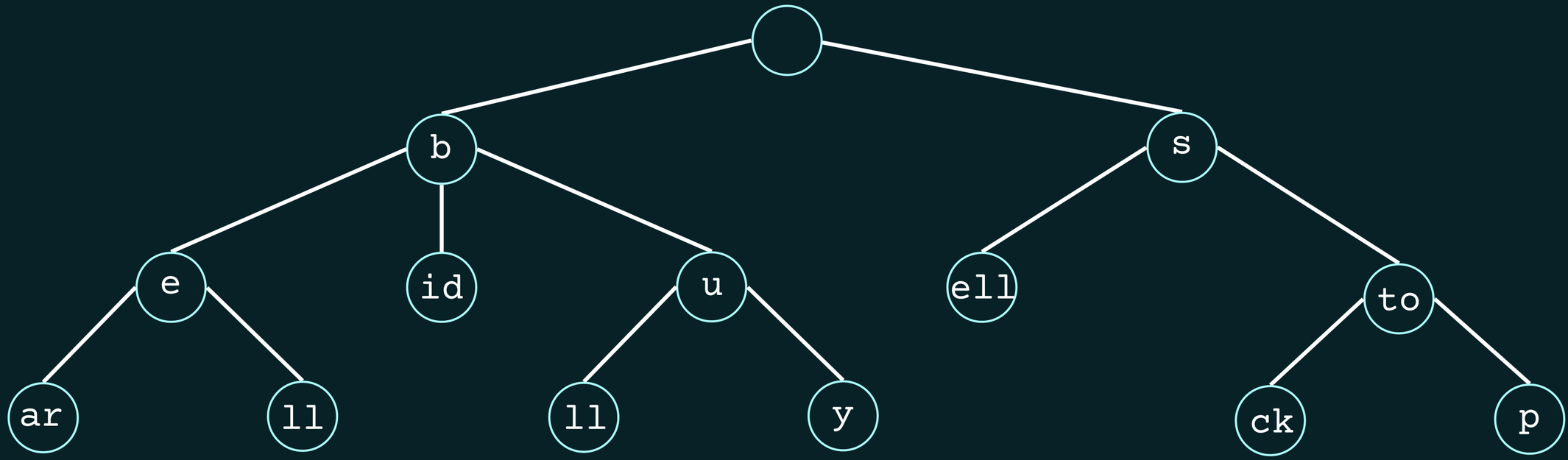


{ bear, bell, bid, bull, buy, sell, stock, stop}



NO. OF NODES: 22

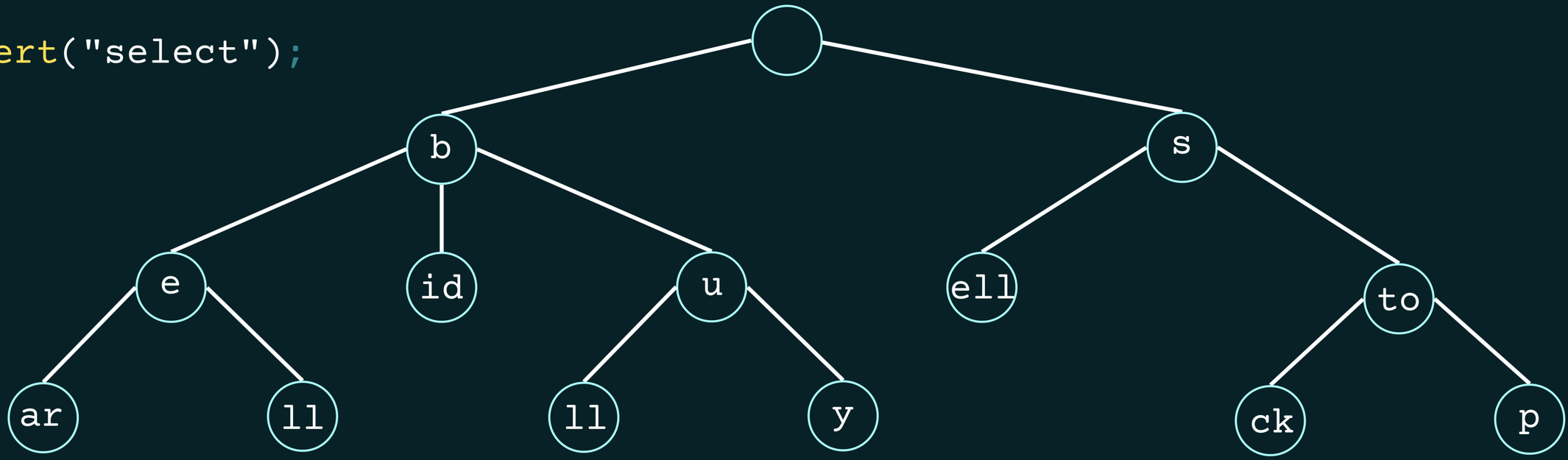
{ bear, bell, bid, bull, buy, sell, stock, stop}



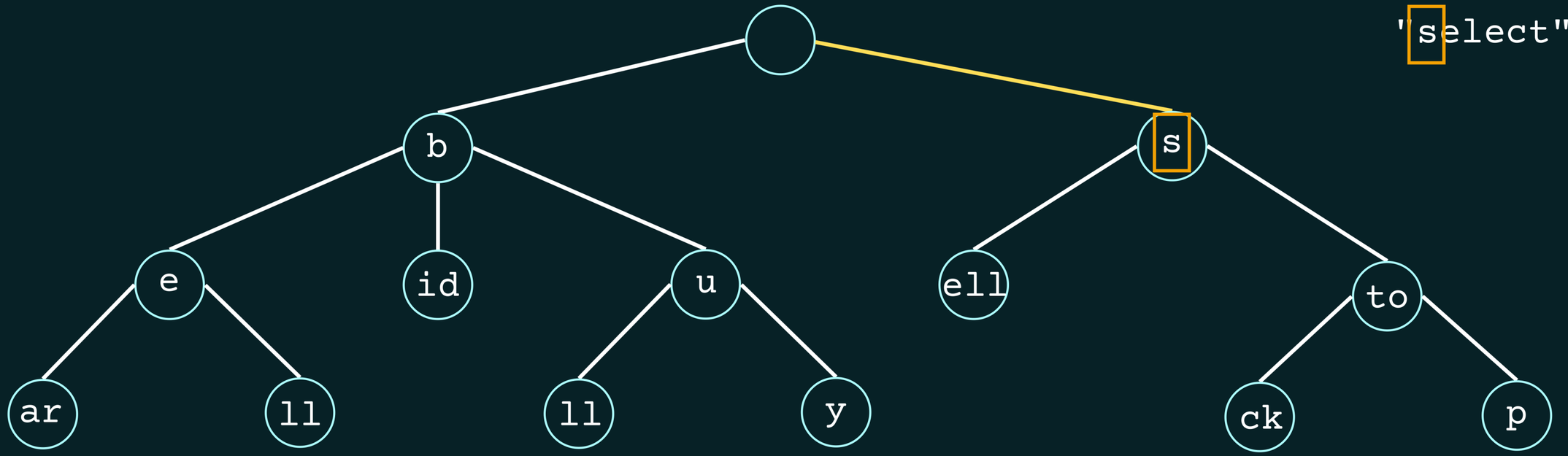
NO. OF NODES: 14

Inserting in Compressed Tries

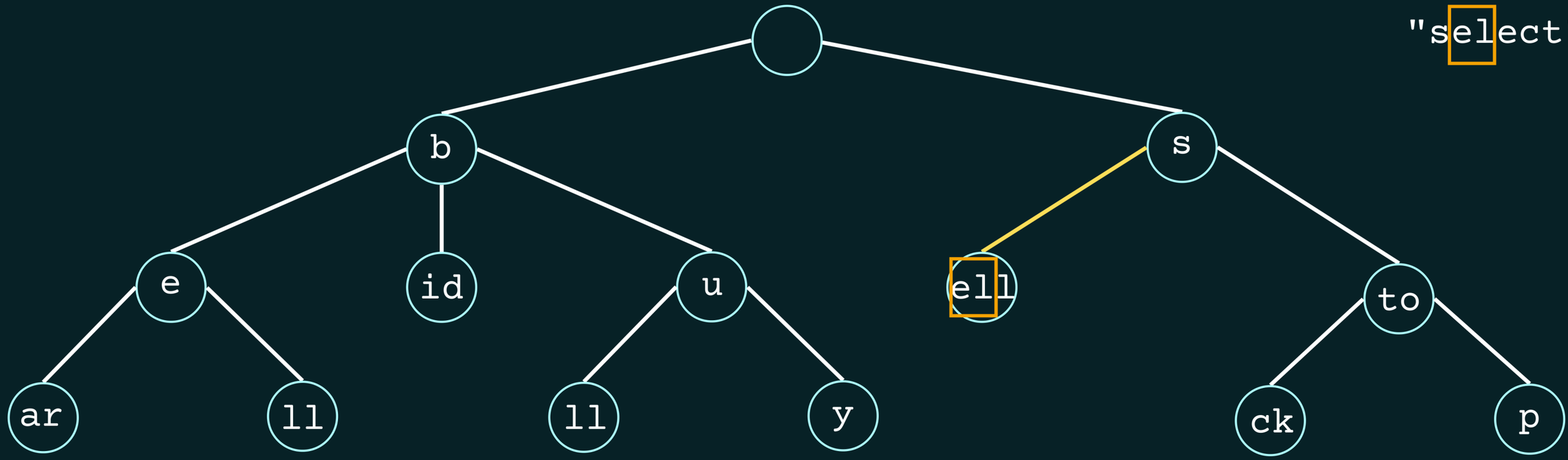
```
insert("select");
```



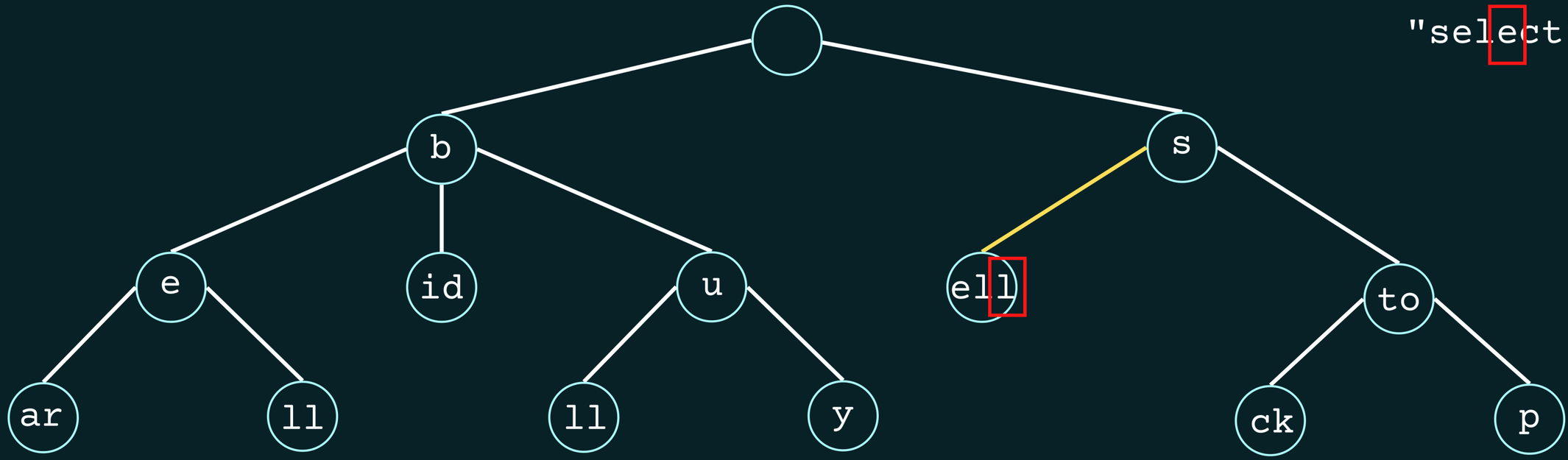
Inserting in Compressed Tries



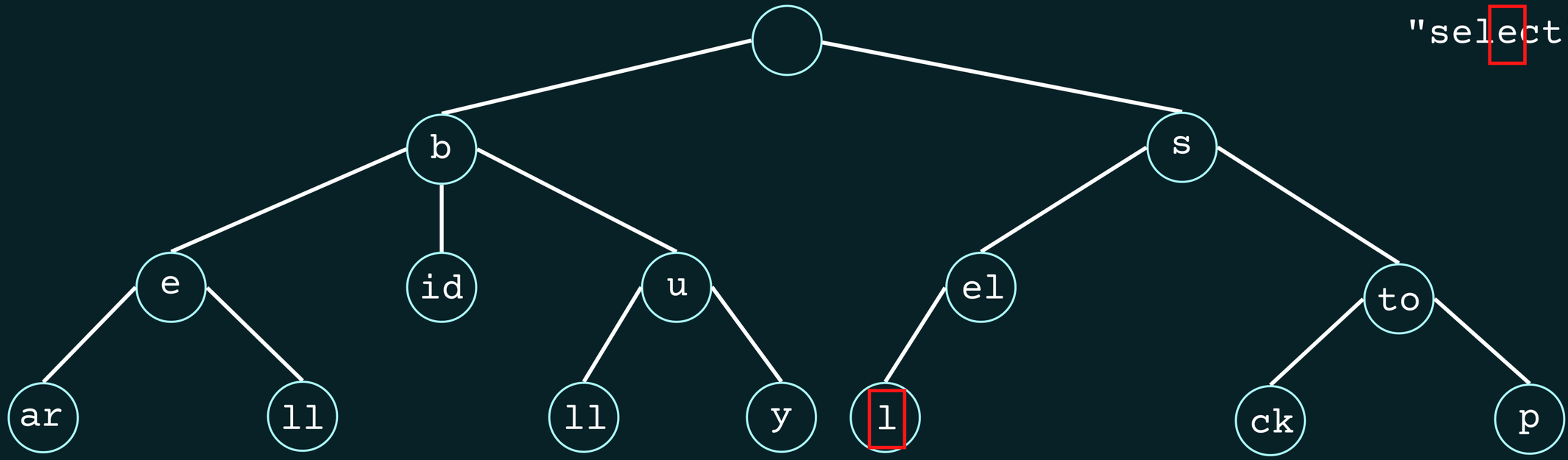
Inserting in Compressed Tries



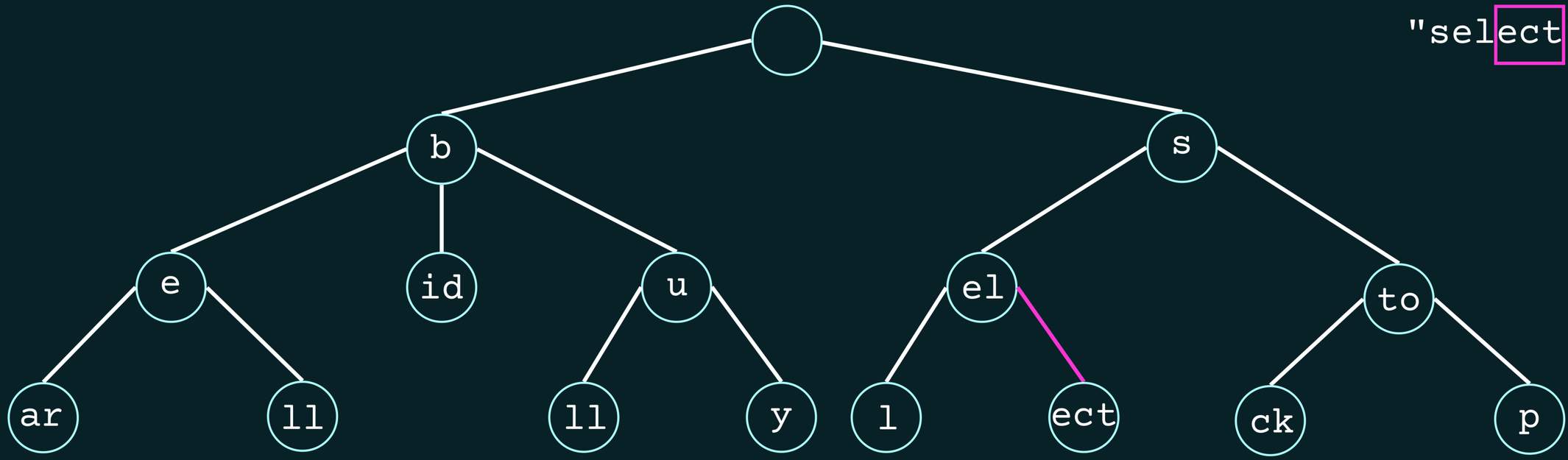
Inserting in Compressed Tries



Inserting in Compressed Tries

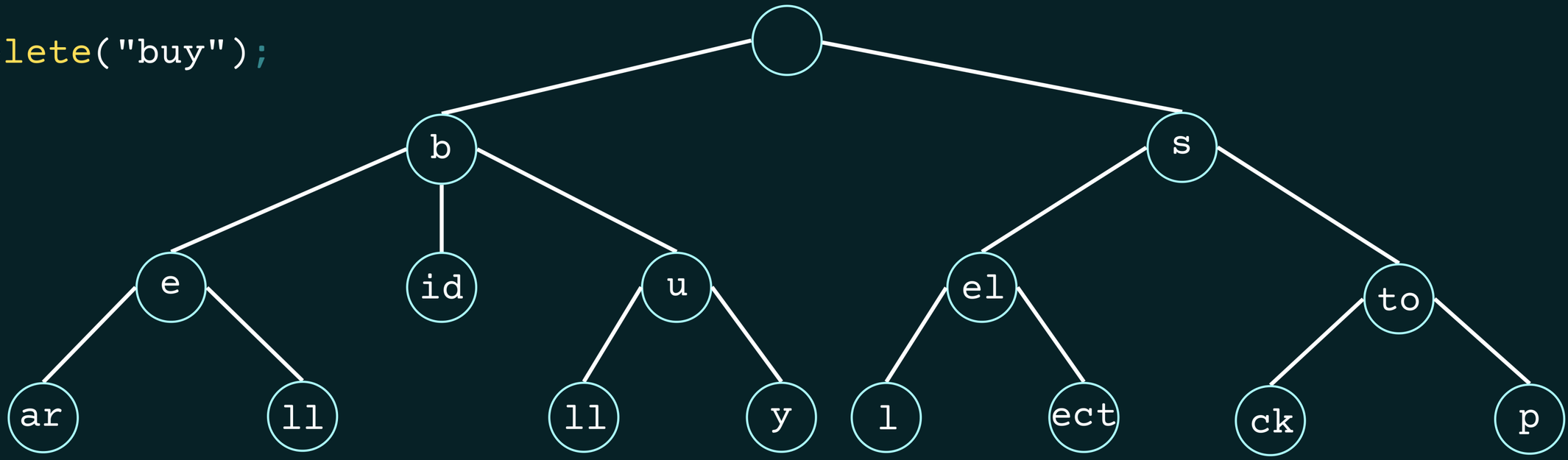


Inserting in Compressed Tries

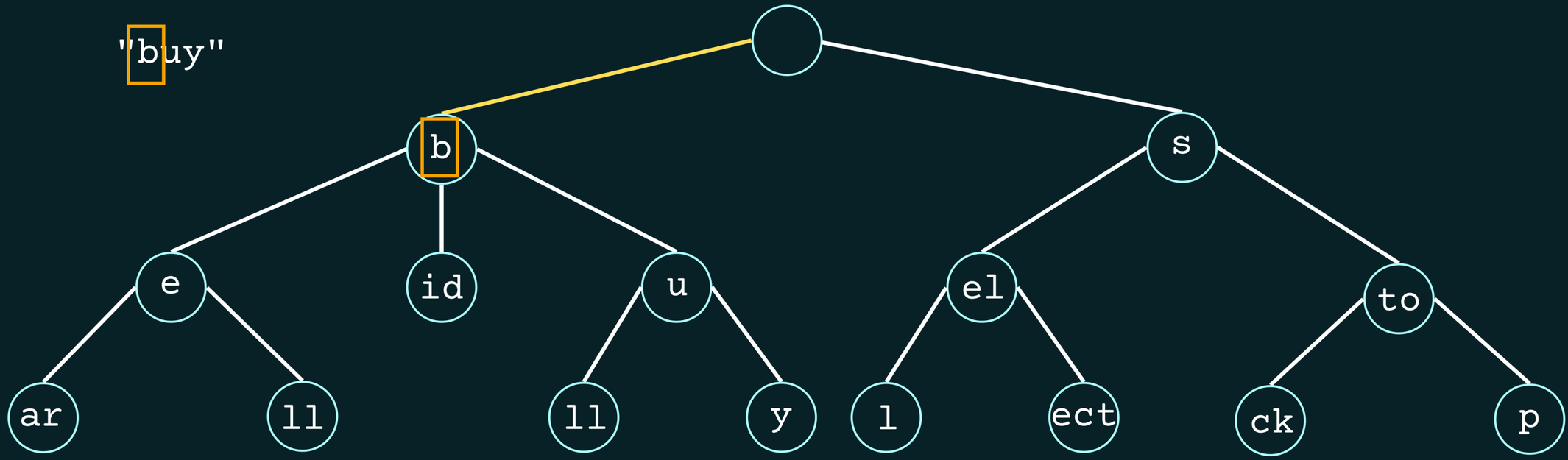


Deleting in Compressed Tries

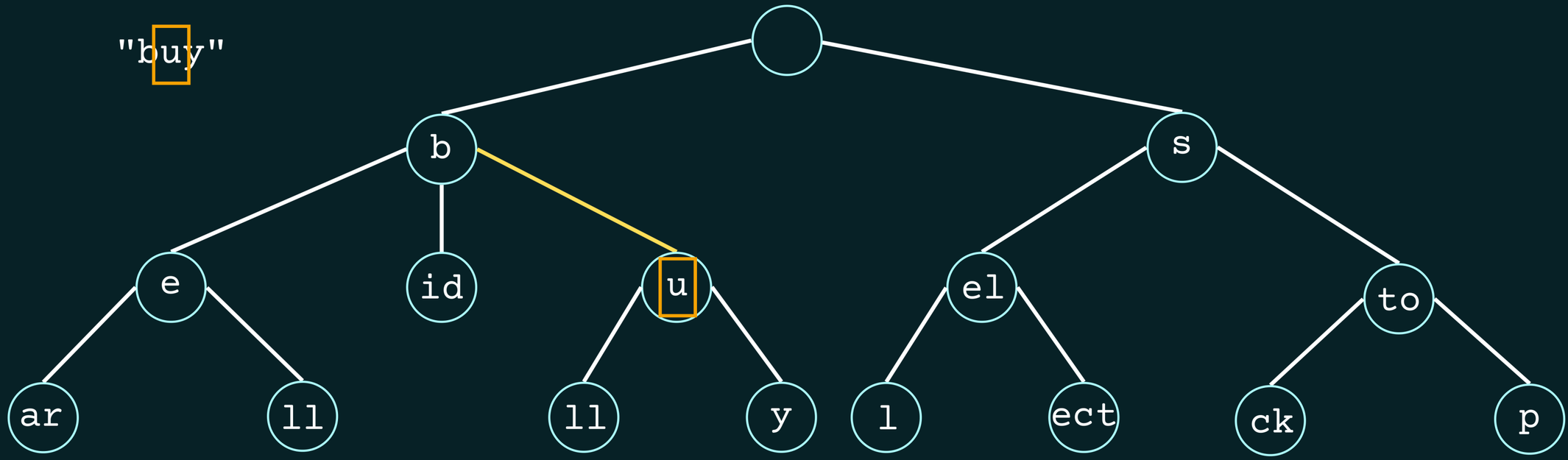
```
delete("buy");
```



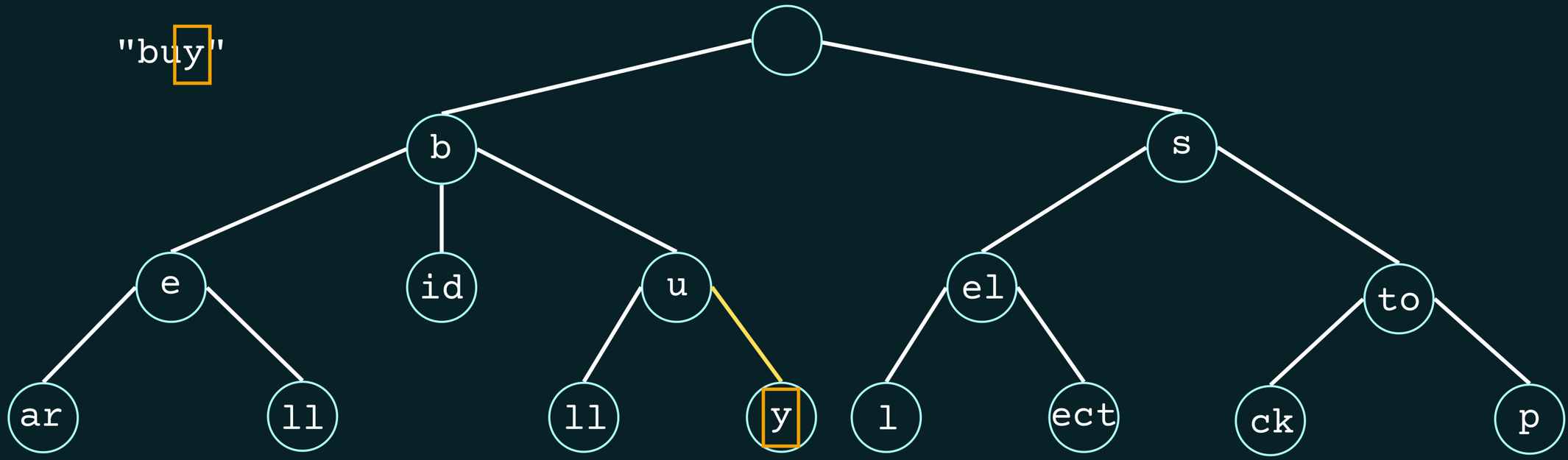
Deleting in Compressed Tries



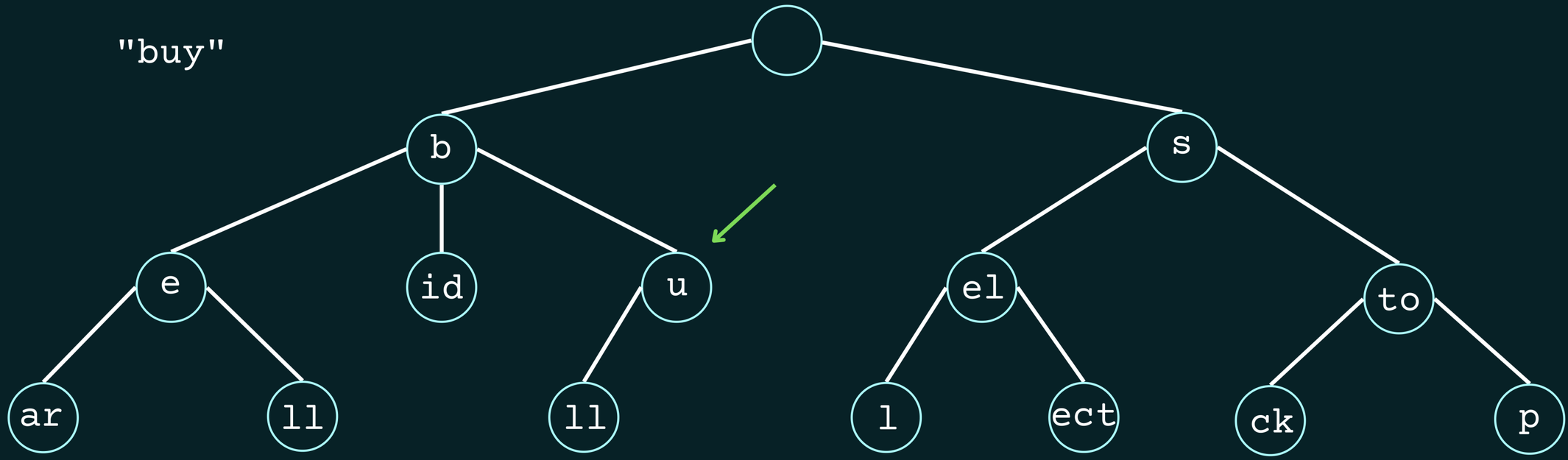
Deleting in Compressed Tries



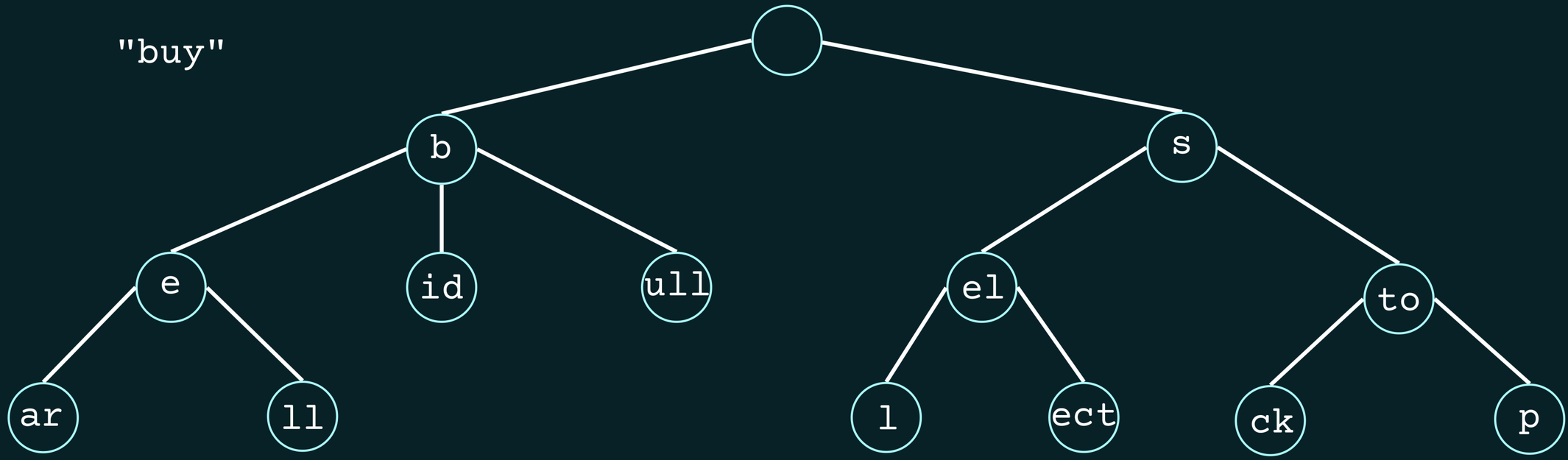
Deleting in Compressed Tries



Deleting in Compressed Tries



Deleting in Compressed Tries



Internet vs Streamline Code

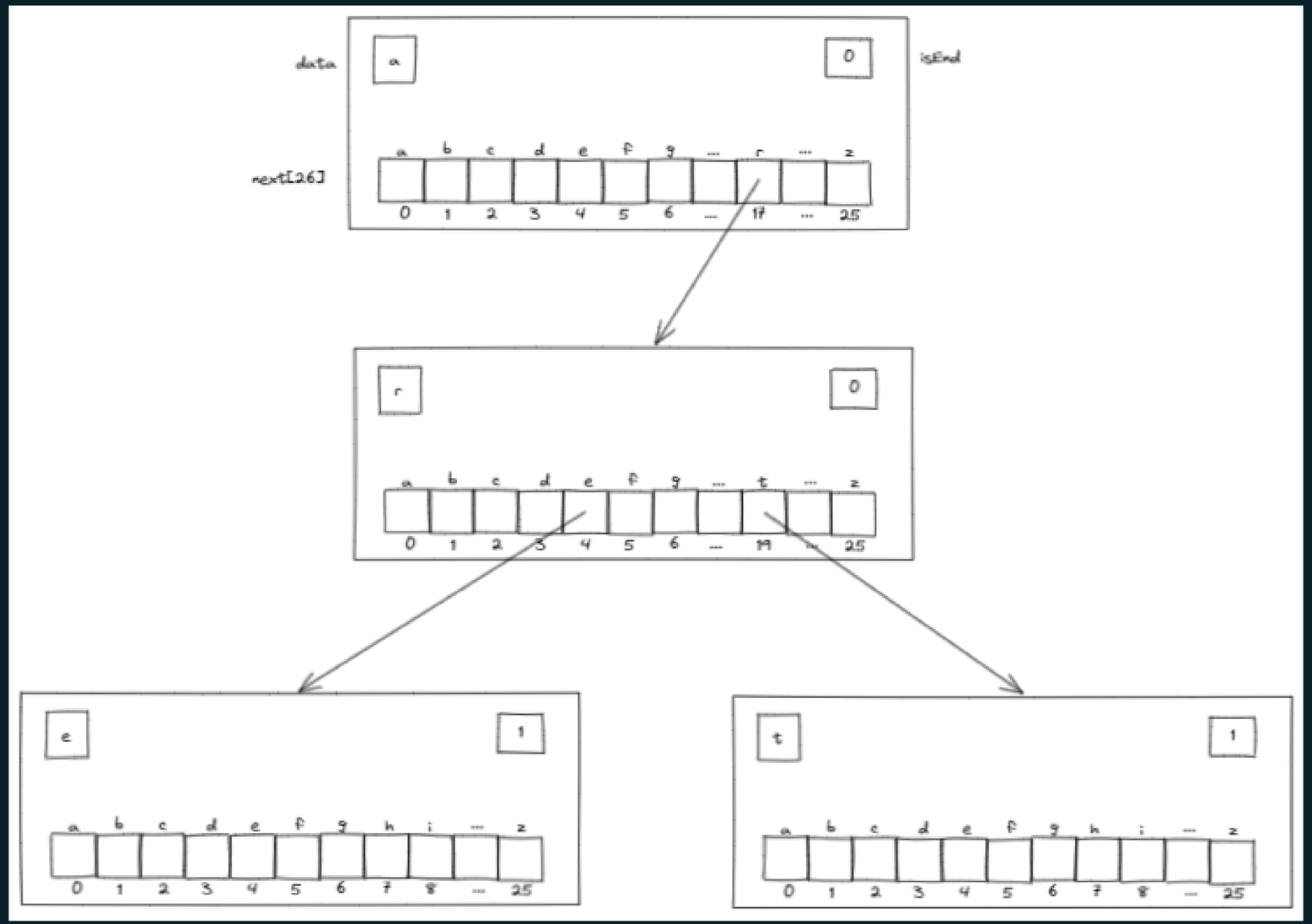
1. Data Structure
2. Insert Function
3. Search Function
4. Delete Function
5. Starts-With Function

```
typedef struct trieNode{
    char data; //character data of the node
    struct trieNode *children[26]; //array of trie pointer
    int isEndofWord; //0 if not end and 1 if end
}*triePtr; //INTERNET CODE, SIZE = 224 BYTES
```

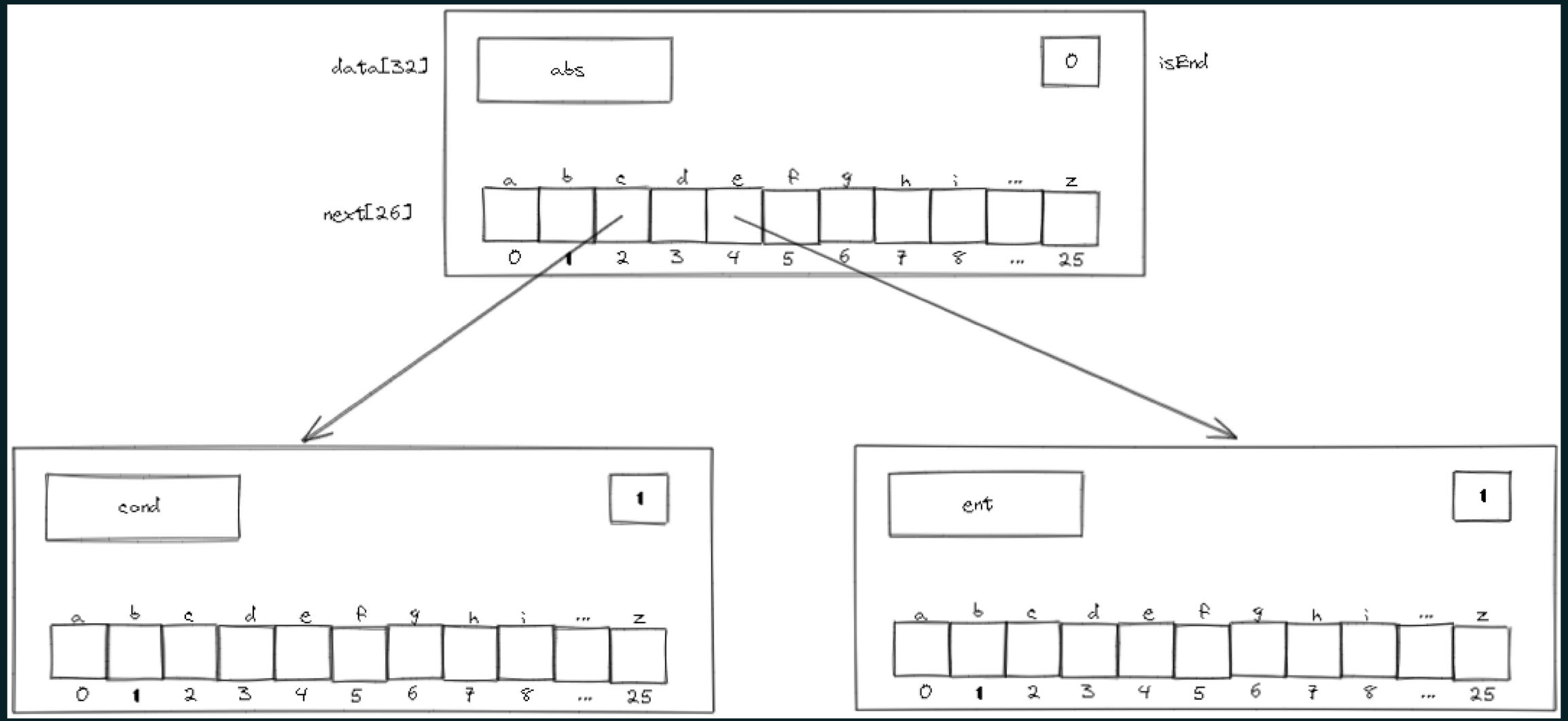
DATA STRUCTURE

```
typedef char String[32];
typedef struct trieNode{
    String data; //string data of the node
    struct trieNode* next[ALPHABET_SIZE]; //array of trie pointer
    int isEnd; //0 if not end and 1 if end
}*triePtr; //STREAMLINED CODE, SIZE  = 248 BYTES
```

DATA STRUCTURE (ORIGINAL TRIE)



DATA STRUCTURE (COMPRESSED TRIE)



Insertion Function

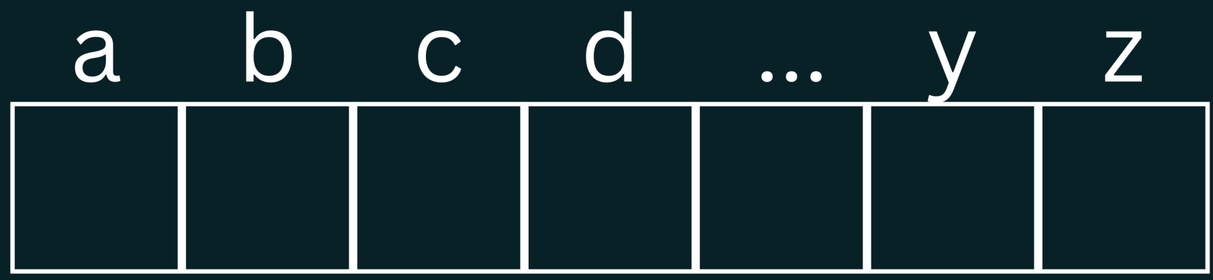
INSERT FUNCTION (Internet Code)

```
void insert(triePtr root, char word[]){
    triePtr trav = NULL;
    int x, index = 0;

    for(x = 0, trav = root; word[x] != '\0'; x++, trav = trav->children[index]){
        // get the index of the current character
        index = tolower(word[x]) - 'a';

        // check if the current character exists in the trie
        // if not, initialize character
        if(trav->children[index] == NULL){
            trav->children[index] = (triePtr) calloc(1, sizeof(struct trieNode));
            trav->data = tolower(word[x]);
        }
    }

    // set the last character as the end of the word
    trav->isEndofWord = 1;
}//INTERNET CODE
```

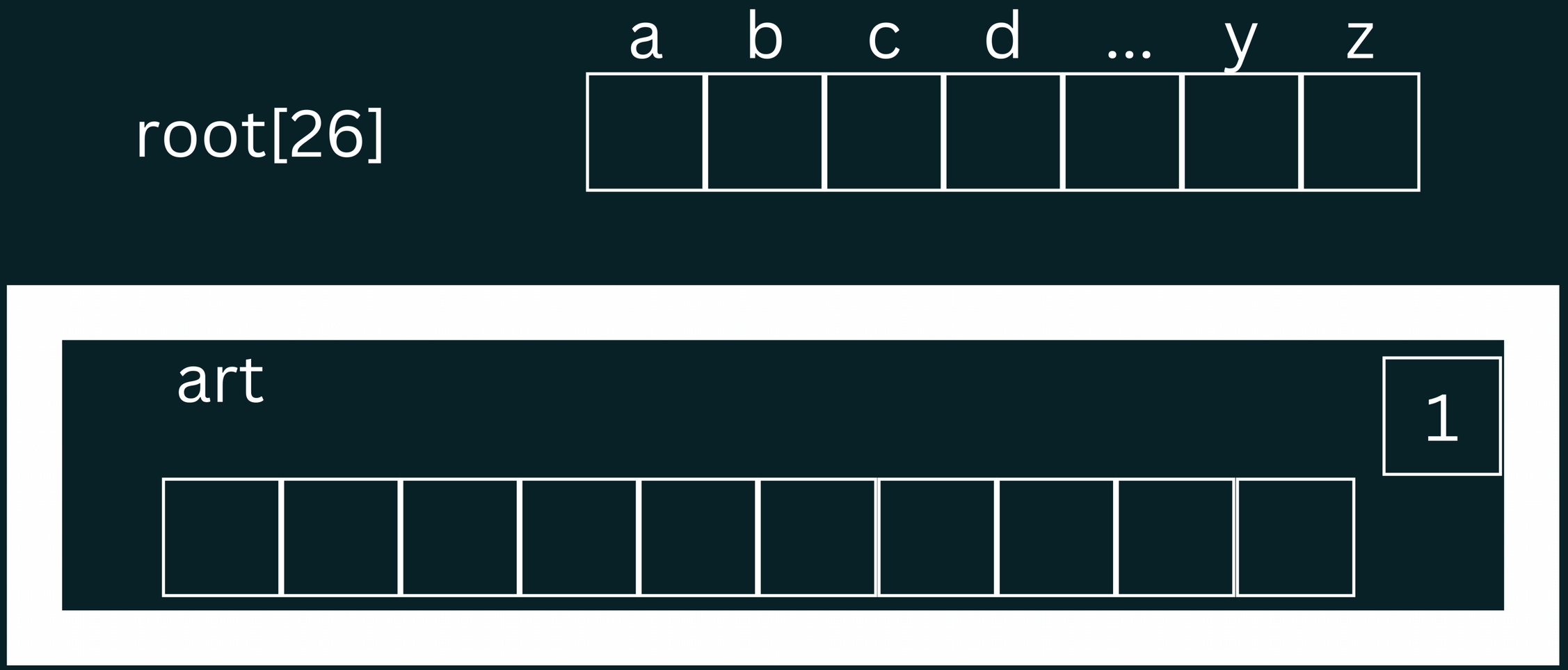
INSERT FUNCTION (Streamlined)

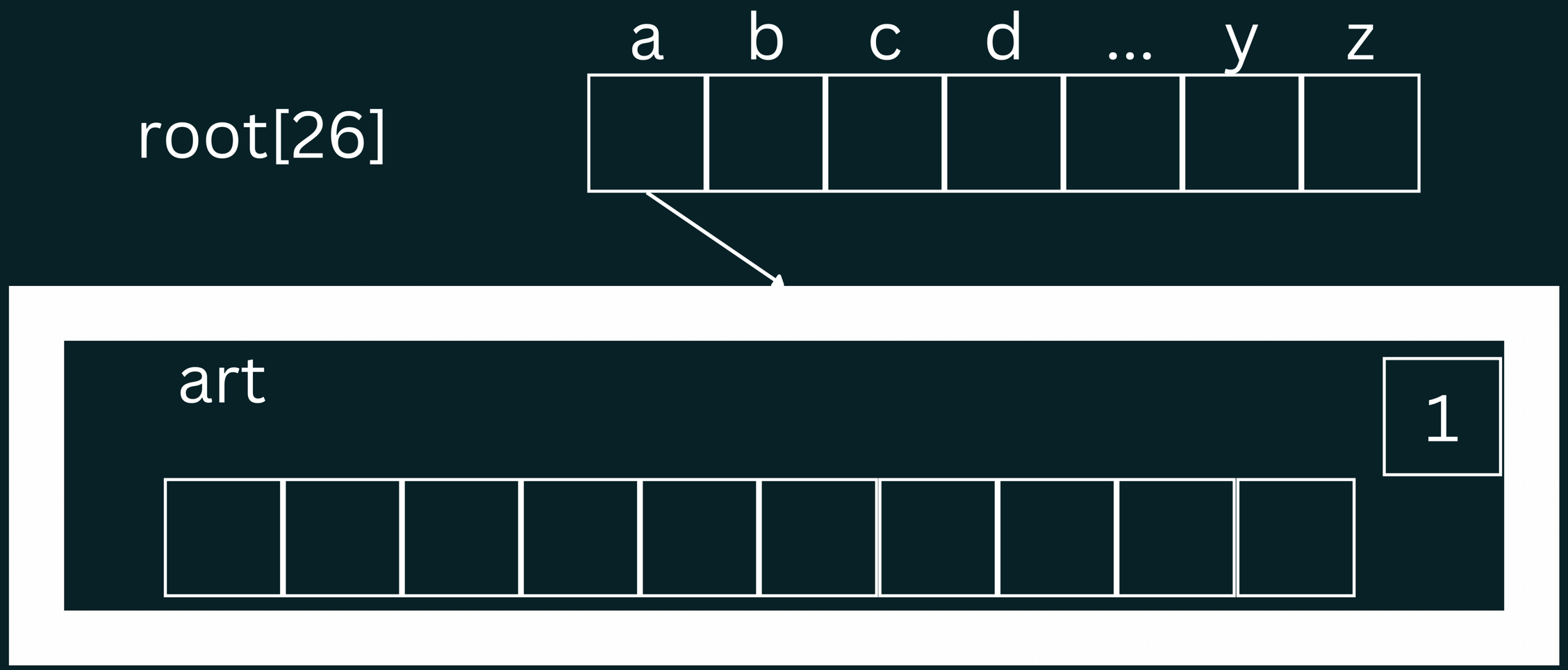
```
triePtr root[ALPHABET_SIZE];
void insertWord(triePtr root[], String s){
   triePtr *head = &root[tolower(s[0]) - 'a'];

   if(*head != NULL){
      insertNode(*head, s);
   }else{
      insertNewNode(head, s);
   }
}

void insertNewNode(triePtr *head, String s){
   *head = (triePtr) calloc(1, sizeof(struct trieNode));
   strcpy((*head)->data, s);
   (*head)->isEnd = 1;
}
```

root[26]



string to be inserted


a b c d ... y z
root[26]
art
1
string to be inserted
a r t *



string to be inserted | a | r | t | *

INSERT FUNCTION (Streamlined)

```
void insertNode(triePtr head, String s){
   int currCtr;

   //checking same characters and stops where similarities stop
   for(currCtr = 0; s[currCtr] == head->data[currCtr] && head->data[currCtr] != '\0'; currCtr++);

   if(currCtr < strlen(head->data) && currCtr < strlen(s)){
      insertCase1(currCtr, head, s, tempArray);
   }else if(strlen(head->data) < strlen(s)){
      insertCase2(currCtr, head, s);
   }else if(strlen(s) < strlen(head->data)){
      insertCase3(currCtr, head, s, tempArray);
   }else{
      head->isEnd = 1;
   }
}
```

Cases for Compressed Trie Insertion

Case 1

This is the case when the string to be inserted and the data string of the current node are not a substring of one or the other. The portion of both strings that are similar are retained on the current node and the differing portions are put on 2 different nodes.

Case 2

This is the case when the data of the current node is a substring of the string to be inserted. If the correct position is initialized, the excess portion of the string is send to the node on the approprite position otherwise the string will be put on a new node at position.

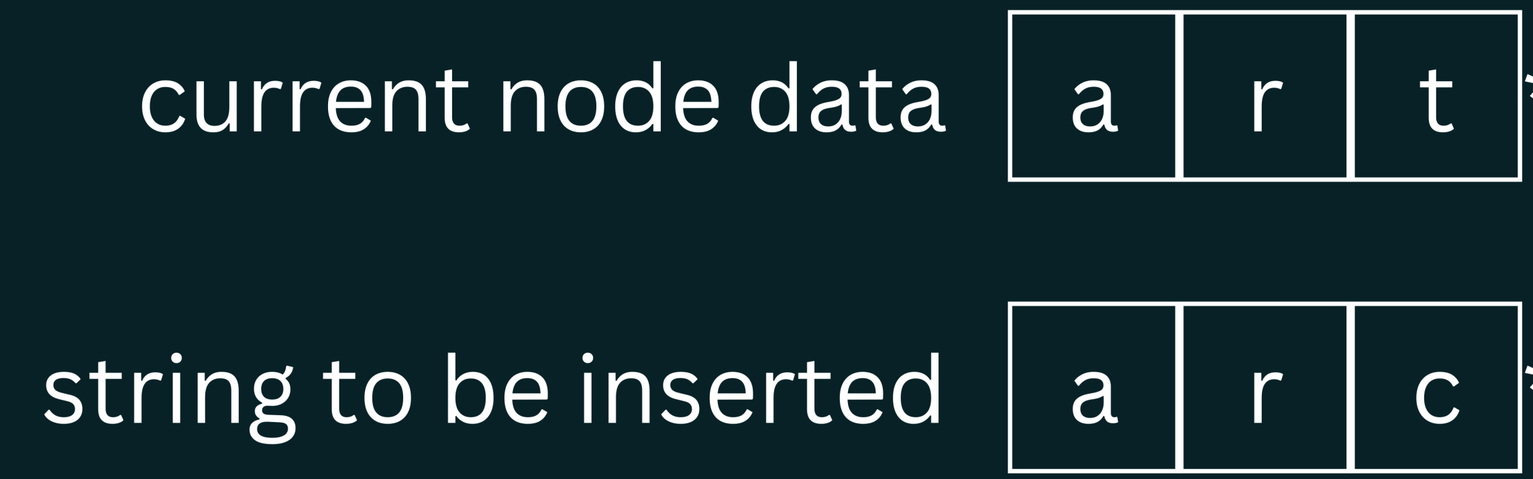
Case 3

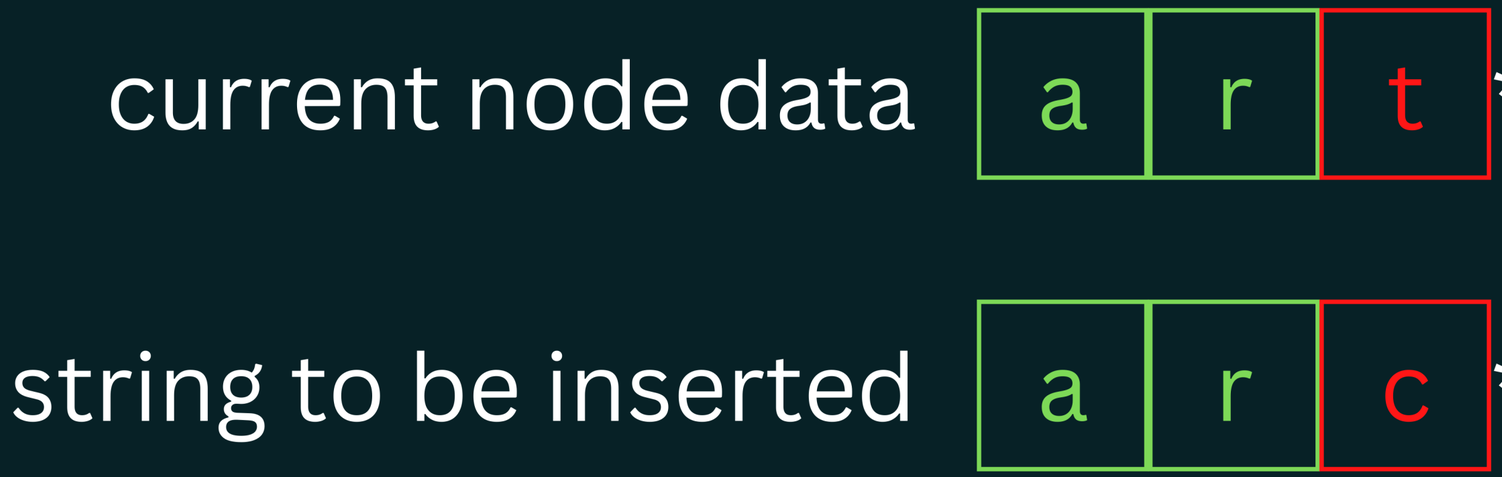
This is the case when the string to be inserted is a substring of the data of the current node. The excess portion of the data string is put on a new node.

Case 4

This is the case when the string to be inserted is identical to the data string of the current node. The current node's isEnd flag is set to 1.

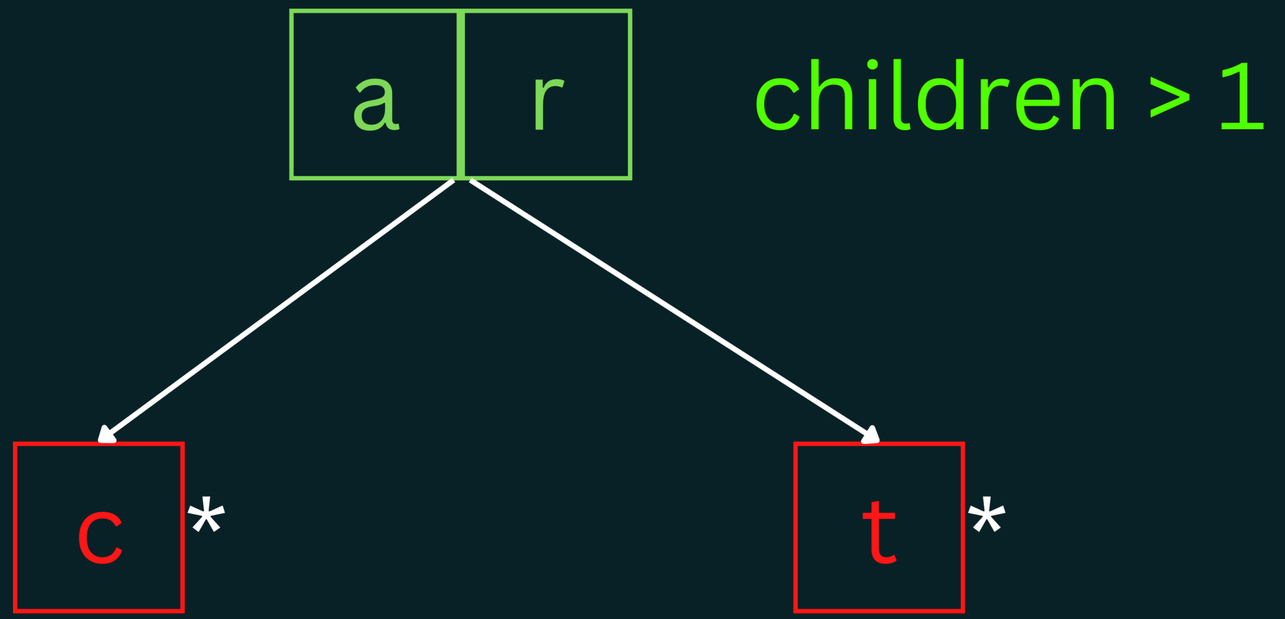
```
void insertCase1(int currCtr, triePtr head, String s, triePtr tempArray[]){
   triePtr tempArray[ALPHABET_SIZE];
   memcpy(tempArray, head->next, sizeof(tempArray));

   int x, nextCtr;
   for(x = 0; x < ALPHABET_SIZE; x++)
      head->next[x] = NULL;

   String s1; memset(s1, '\0', sizeof(String));
   String s2; memset(s2, '\0', sizeof(String));

   for(nextCtr = 0; s[currCtr] != '\0' || head->data[currCtr] != '\0'; nextCtr++, currCtr++){
      if(s[currCtr] != '\0')
         s1[nextCtr] = s[currCtr];
      if(head->data[currCtr] != '\0'){
         s2[nextCtr] = head->data[currCtr];
         head->data[currCtr] = '\0';
      }
   }
   insertNewNode(&head->next[s1[0] - 'a'], s1);
   insertNewNode(&head->next[s2[0] - 'a'], s2);
   triePtr temp = head->next[s2[0] - 'a'];
   temp->isEnd = head->isEnd;
   memcpy(temp->next, tempArray, sizeof(tempArray));
   head->isEnd = 0;
}
```


current node data
a
r
t
*
string to be inserted
a
r
c
*


current node data
a
r
t
*
string to be inserted
a
r
c
*

current node data

newly inserted
node data



Cases for Compressed Trie Insertion

Case 1

This is the case when the string to be inserted and the data string of the current node are not a substring of one or the other. The portion of both strings that are similar are retained on the current node and the differing portions are put on 2 different nodes.

Case 2

This is the case when the data of the current node is a substring of the string to be inserted. If the correct position is initialized, the excess portion of the string is sent to the node on the approprite position otherwise the string will be put on a new node at position.

Case 3

This is the case when the string to be inserted is a substring of the data of the current node. The excess portion of the data string is put on a new node.

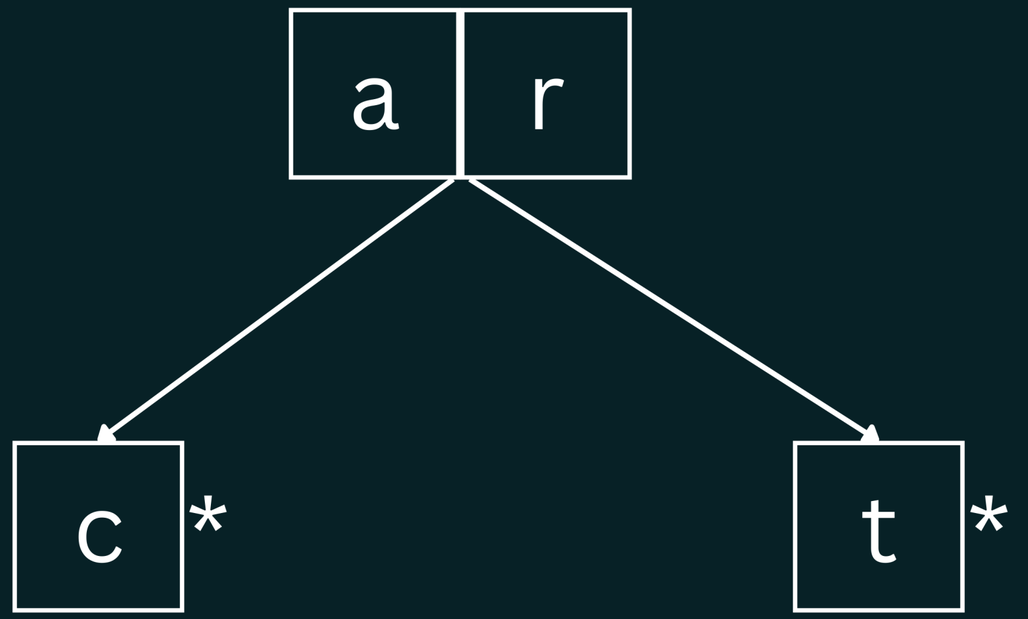
Case 4

This is the case when the string to be inserted is identical to the data string of the current node. The current node's isEnd flag is set to 1.

INSERT FUNCTION

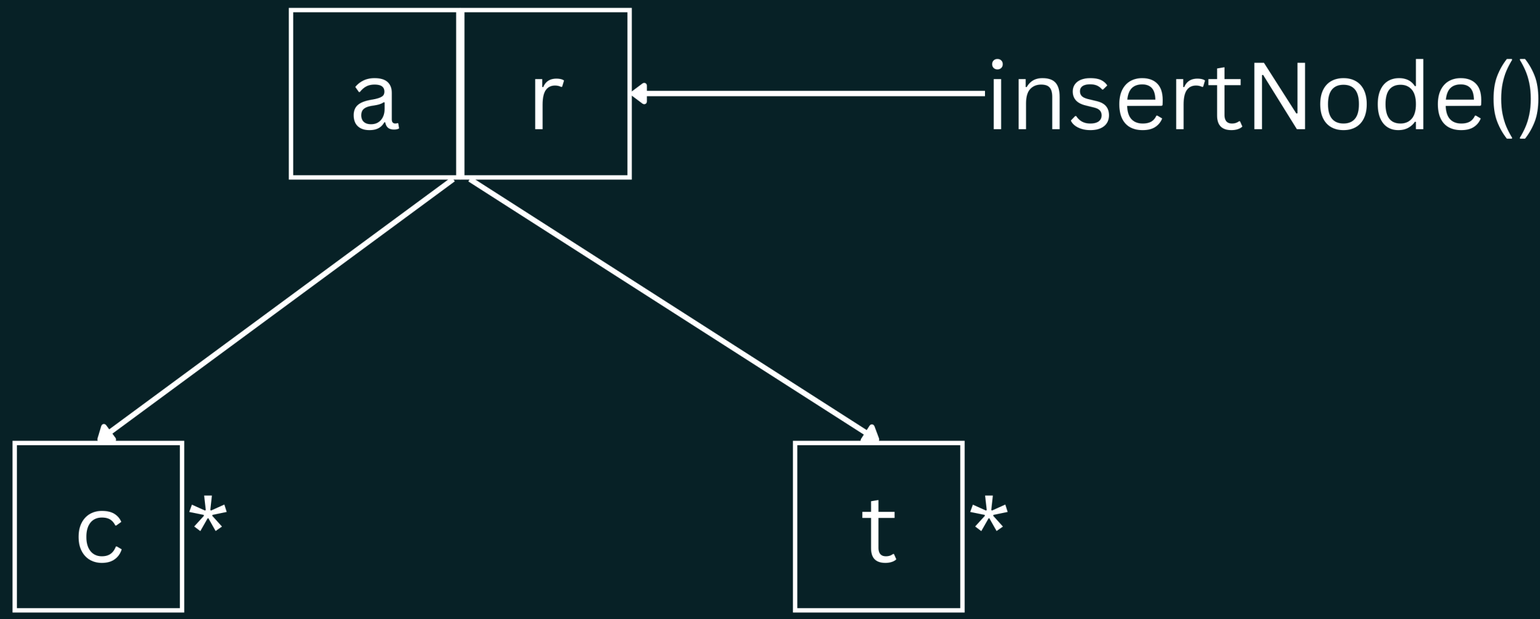
```
void insertCase2(int currCtr, triePtr head, String s){
   String ret;
   memset(ret, '\0', sizeof(String));

   int nextCtr;
   for(nextCtr = 0; s[currCtr] != '\0'; nextCtr++, currCtr++){
      ret[nextCtr] = s[currCtr];
   }

   if(head->next[ret[0] - 'a'] != NULL){
      insertNode(head->next[ret[0] - 'a'], ret);
   }else{
      insertNewNode(&head->next[ret[0] - 'a'], ret);
   }
}
```

current node data



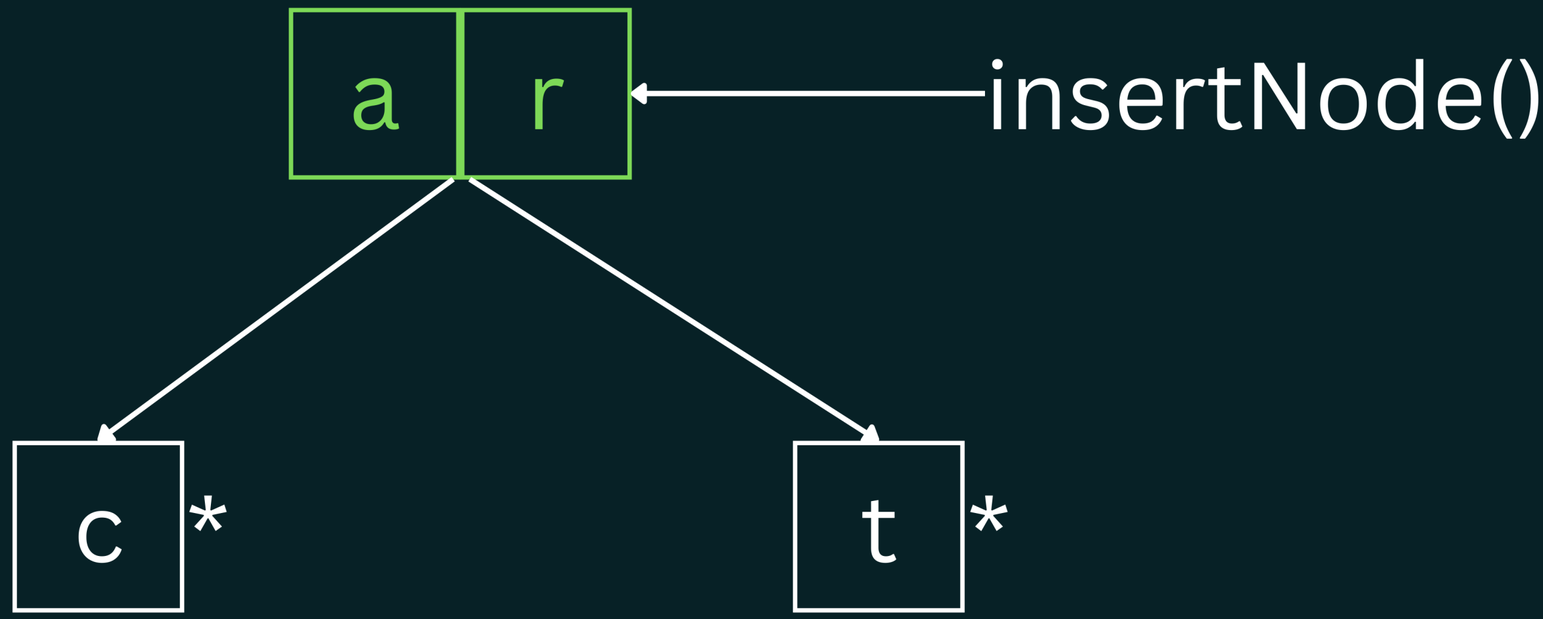
string to be inserted

current node data



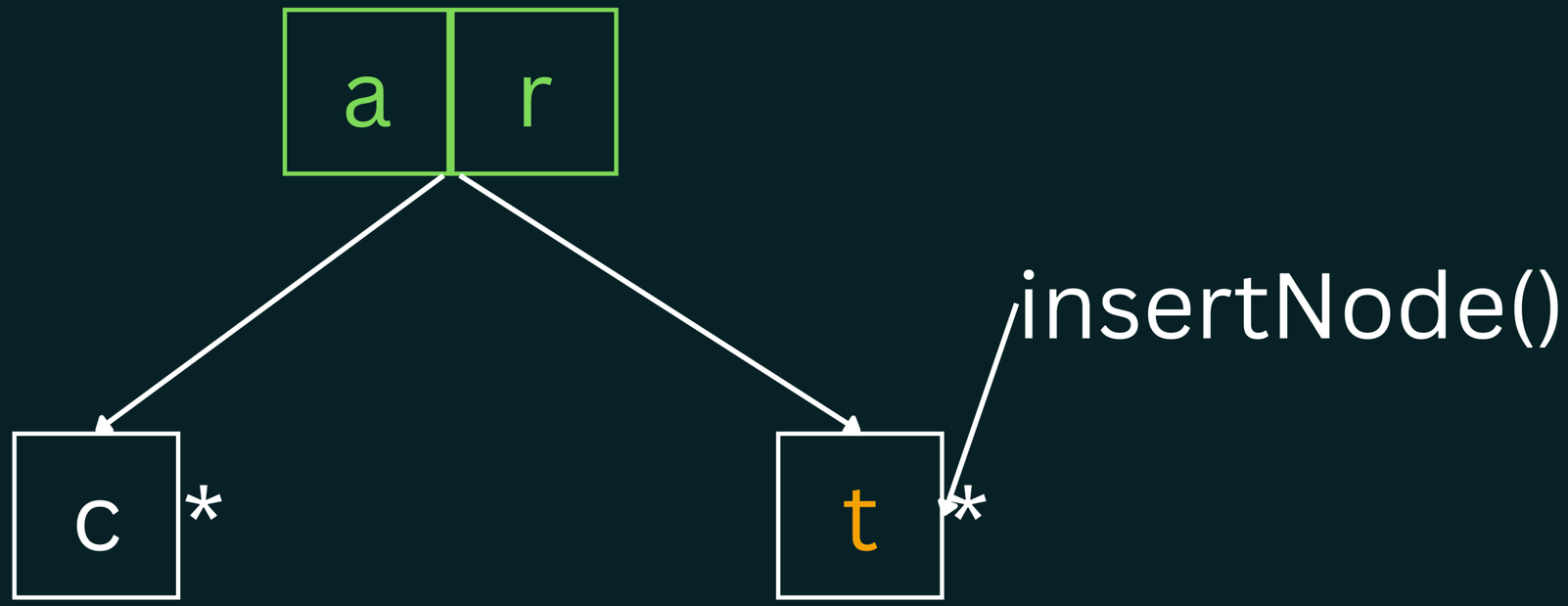
string to be inserted

current node data



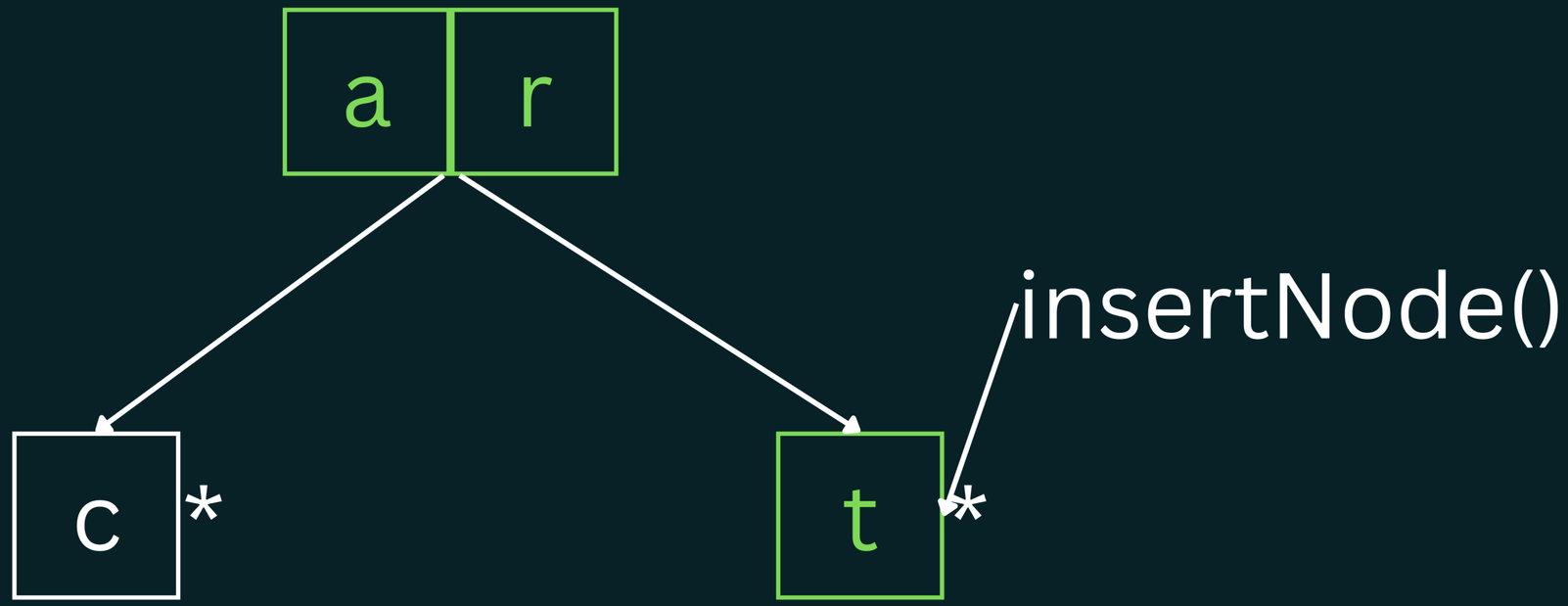
string to be inserted

current node data



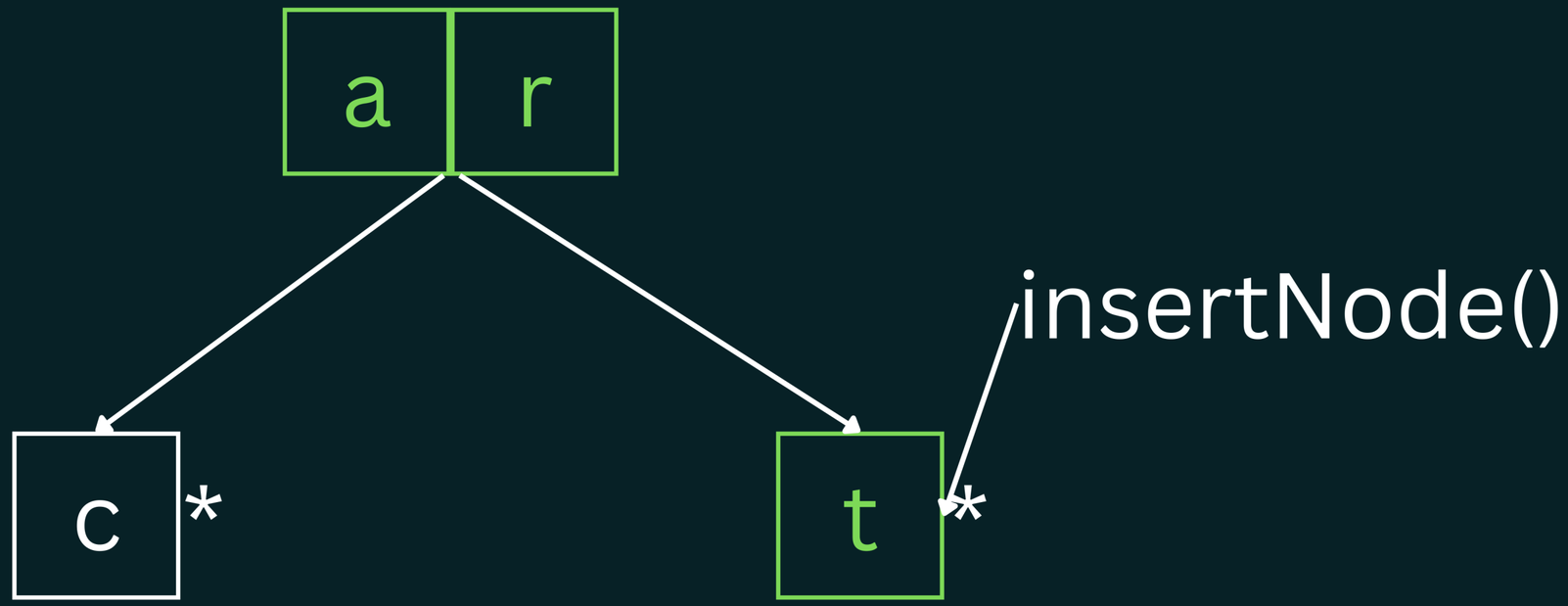
string to be inserted

current node data



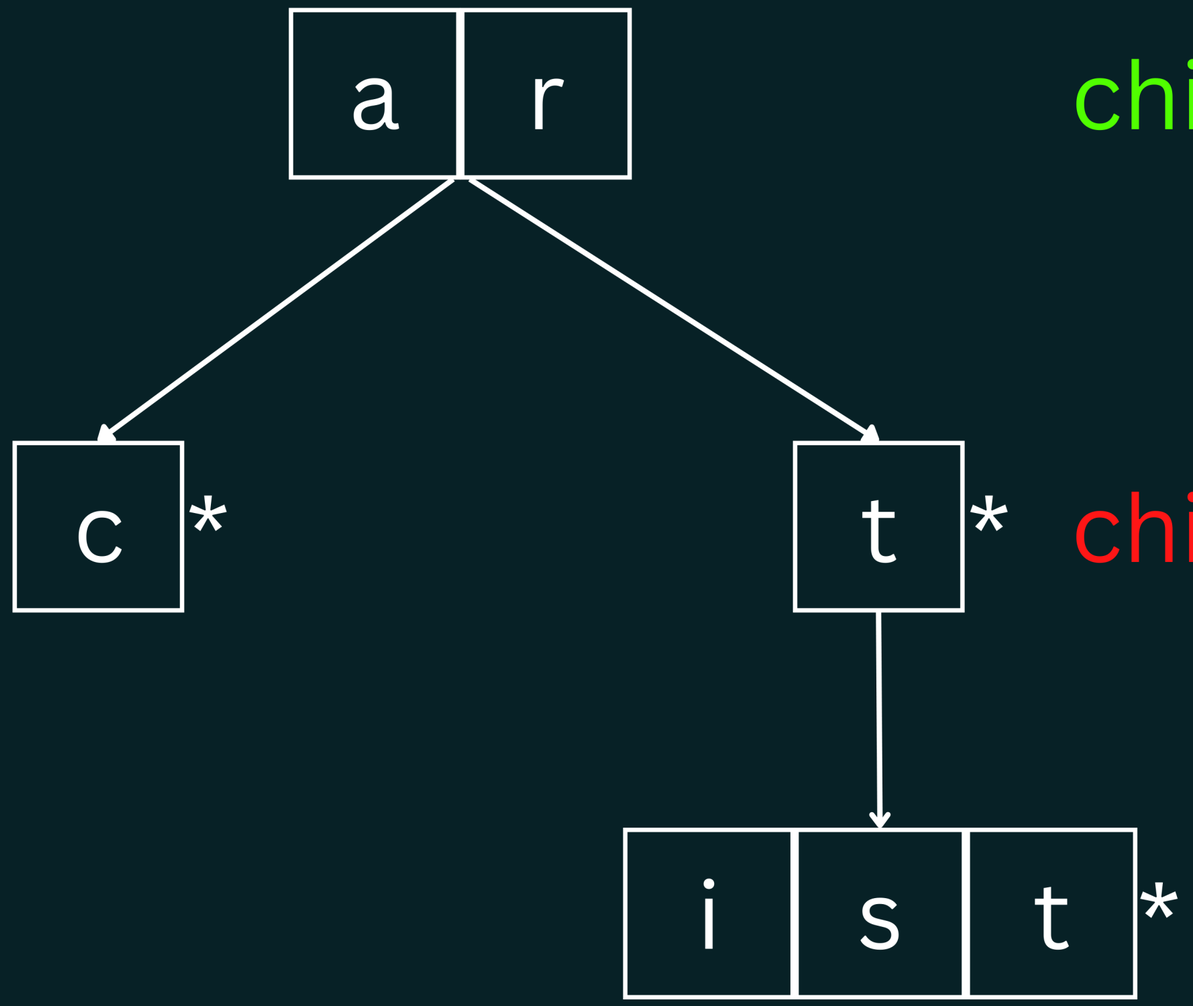
string to be inserted

current node data



string to be inserted

current node data



children > 1

children > 1

newly inserted
node data

Team Exception:

endNodes can have 1 child

Cases for Compressed Trie Insertion

Case 1

This is the case when the string to be inserted and the data string of the current node are not a substring of one or the other. The portion of both strings that are similar are retained on the current node and the differing portions are put on 2 different nodes.

Case 2

This is the case when the data of the current node is a substring of the string to be inserted. If the correct position is initialized, the excess portion of the string is send to the node on the approprite position otherwise the string will be put on a new node at position.

Case 3

This is the case when the string to be inserted is a substring of the data of the current node. The excess portion of the data string is put on a new node and the array of children is carried over to the new node.

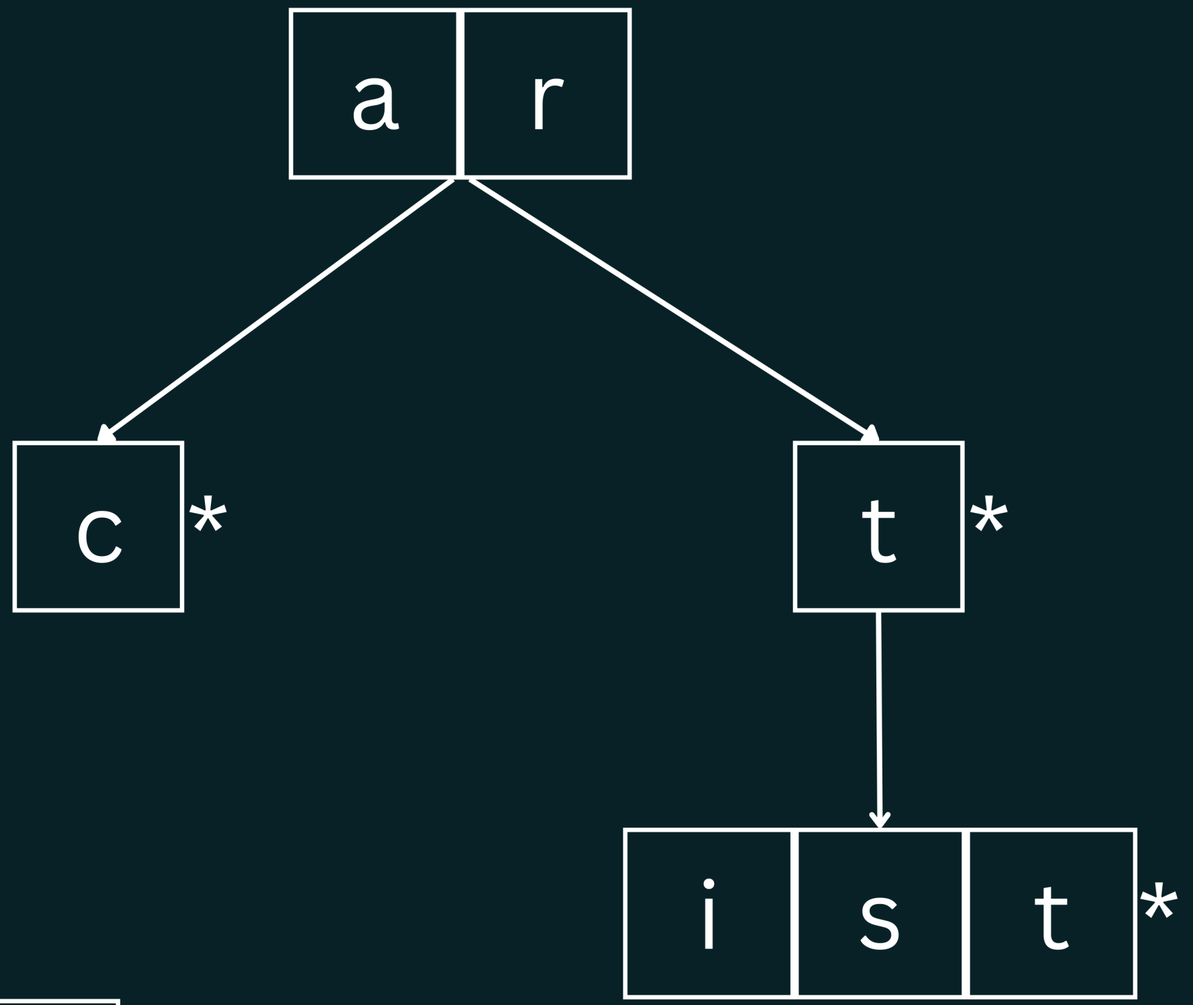
Case 4

This is the case when the string to be inserted is identical to the data string of the current node. The current node's isEnd flag is set to 1.

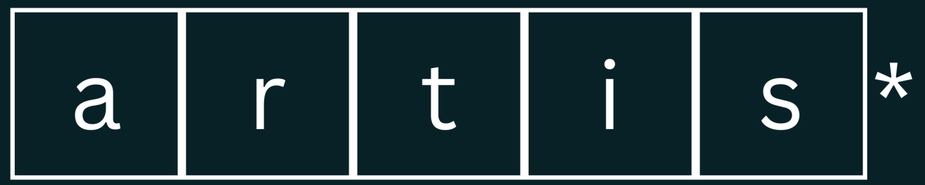
INSERT FUNCTION

```
void insertCase3(int currCtr, triePtr head, String s, triePtr tempArray[]){
   triePtr tempArray[ALPHABET_SIZE];
   memcpy(tempArray, head->next, sizeof(tempArray));

   String ret;
   memset(ret, '\0', sizeof(String));

   int nextCtr;
   for(nextCtr = 0; head->data[currCtr] != '\0'; nextCtr++, currCtr++){
      ret[nextCtr] = head->data[currCtr];
      head->data[currCtr] = '\0';
   }

   insertNewNode(&head->next[ret[0] - 'a'], ret);
   triePtr temp = head->next[ret[0] - 'a'];
   temp->isEnd = head->isEnd;
   memcpy(temp->next, tempArray, sizeof(tempArray));

    head->isEnd = 1;
}
```

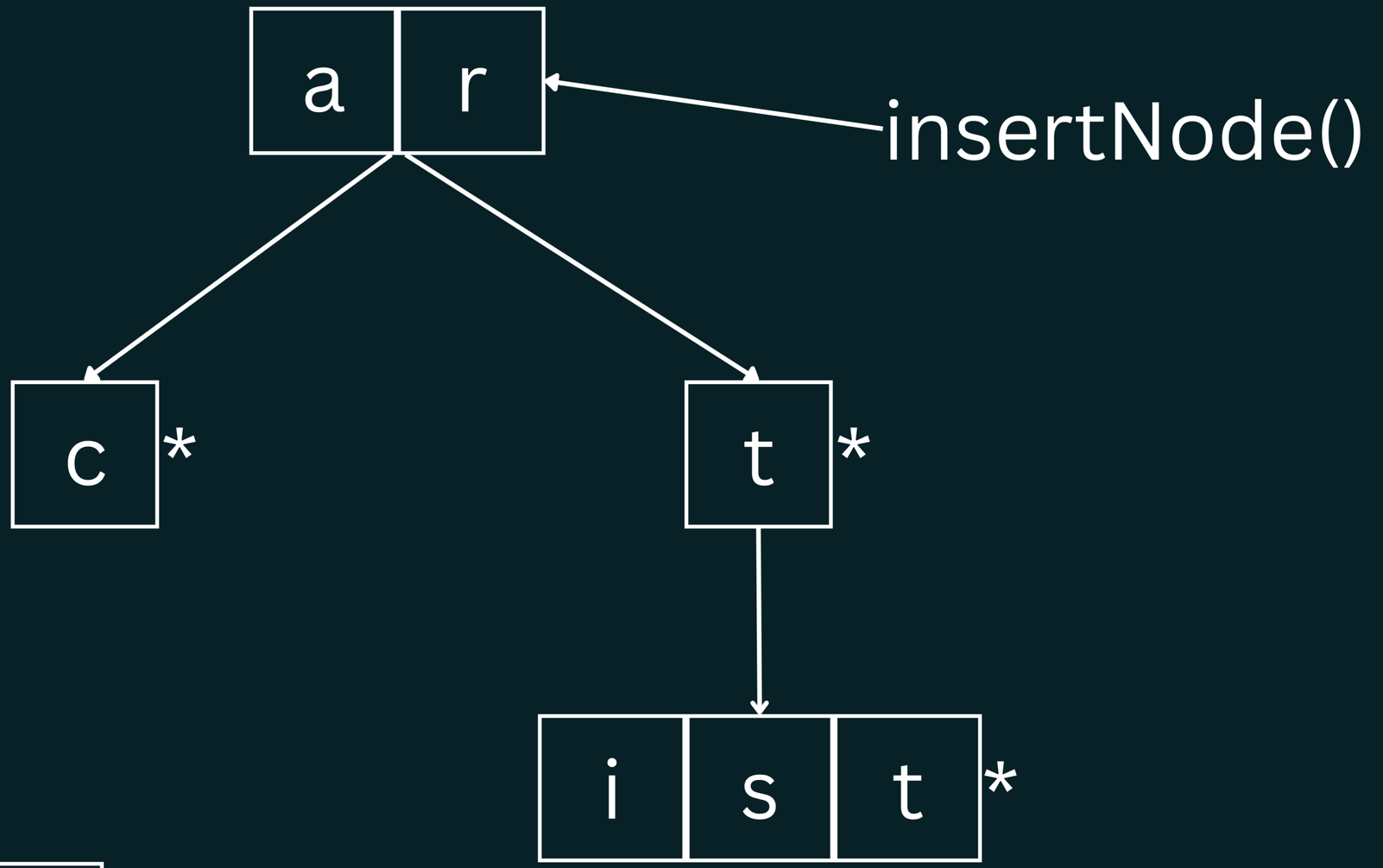
current node data



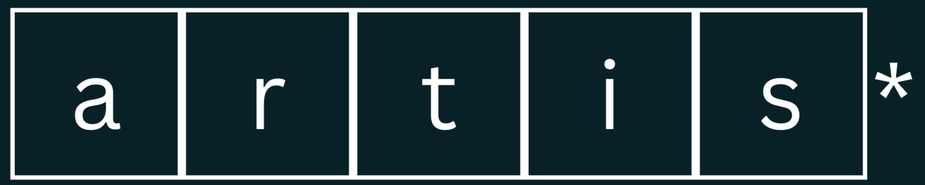
string to be inserted

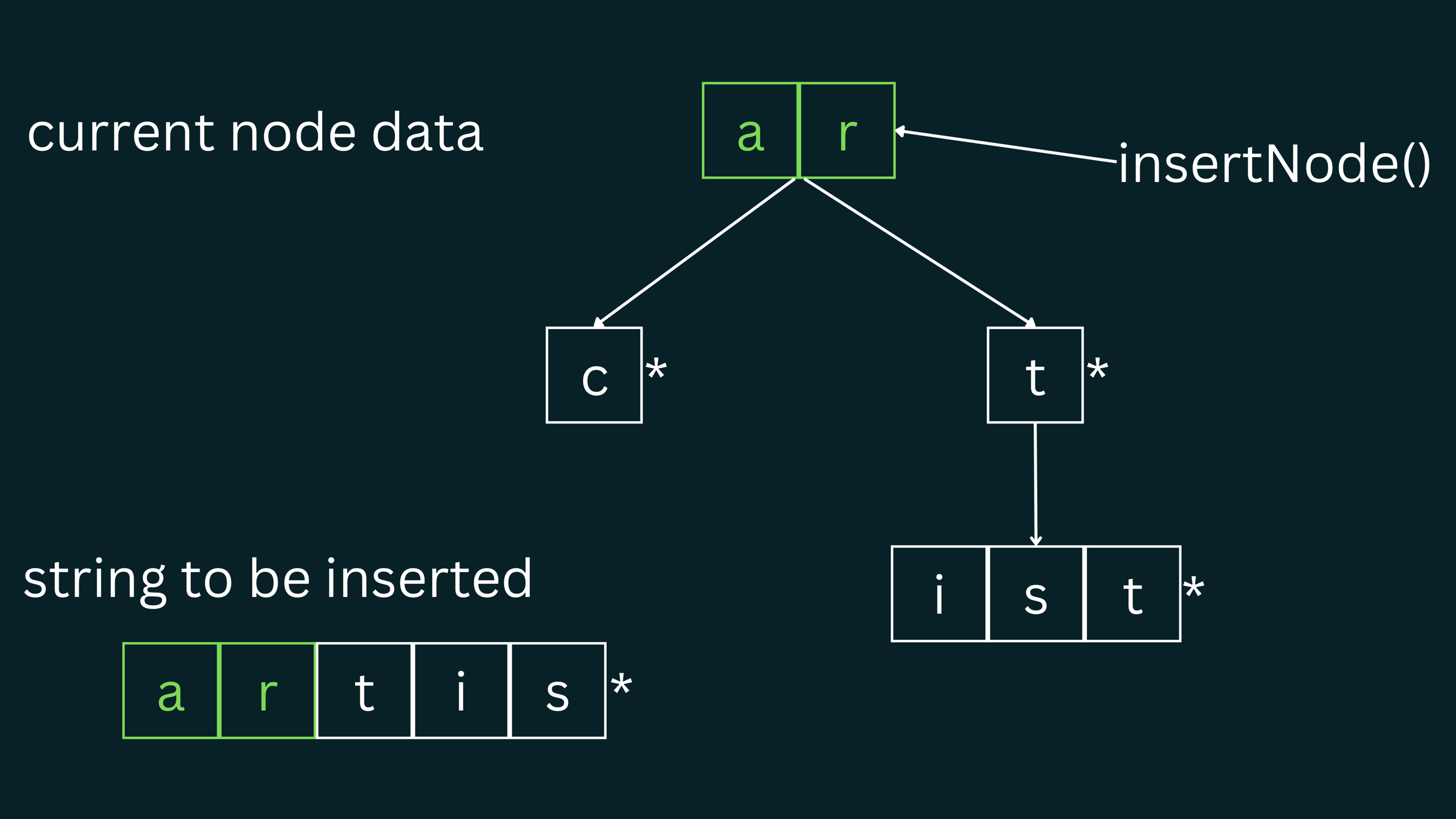


current node data

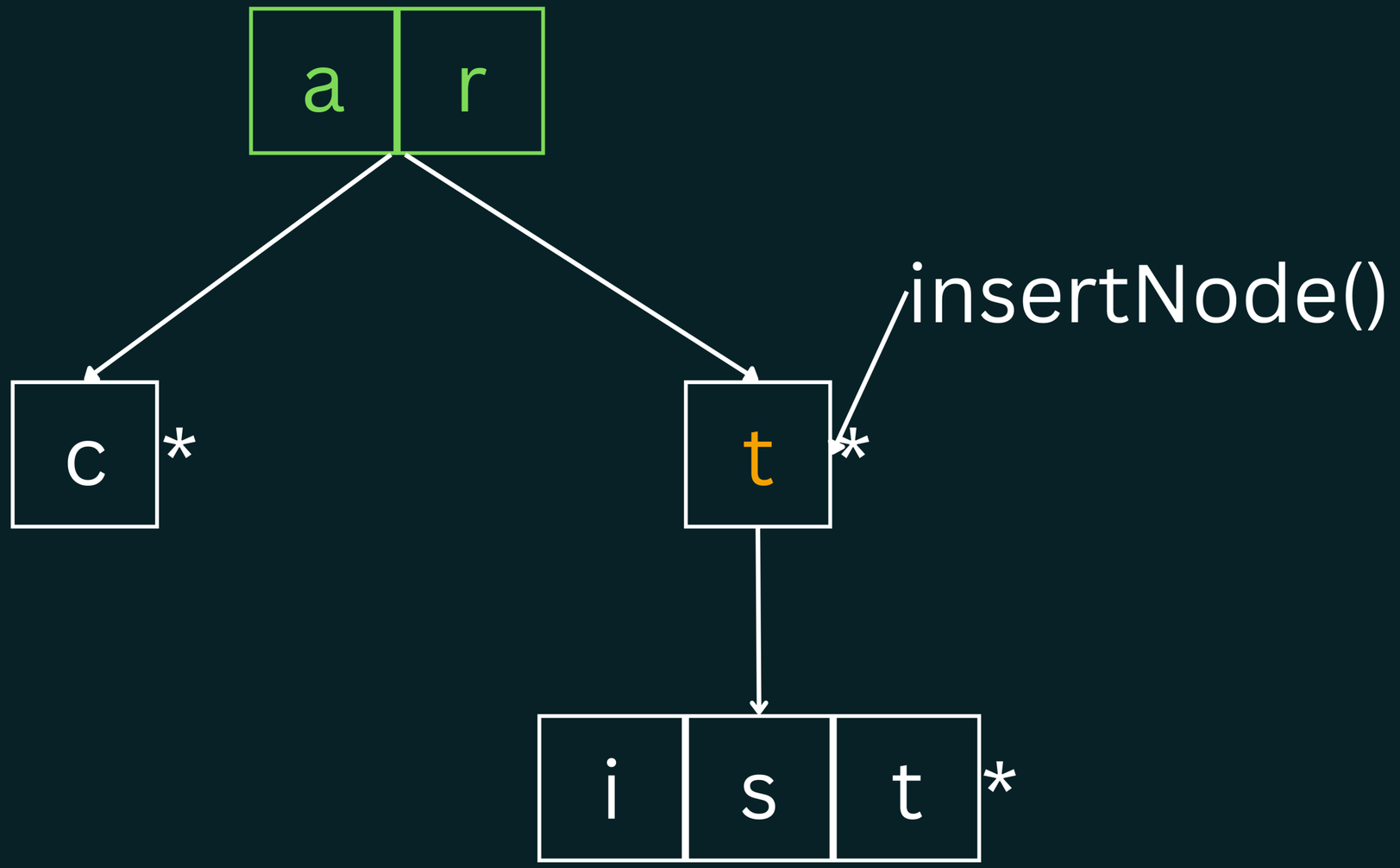


string to be inserted



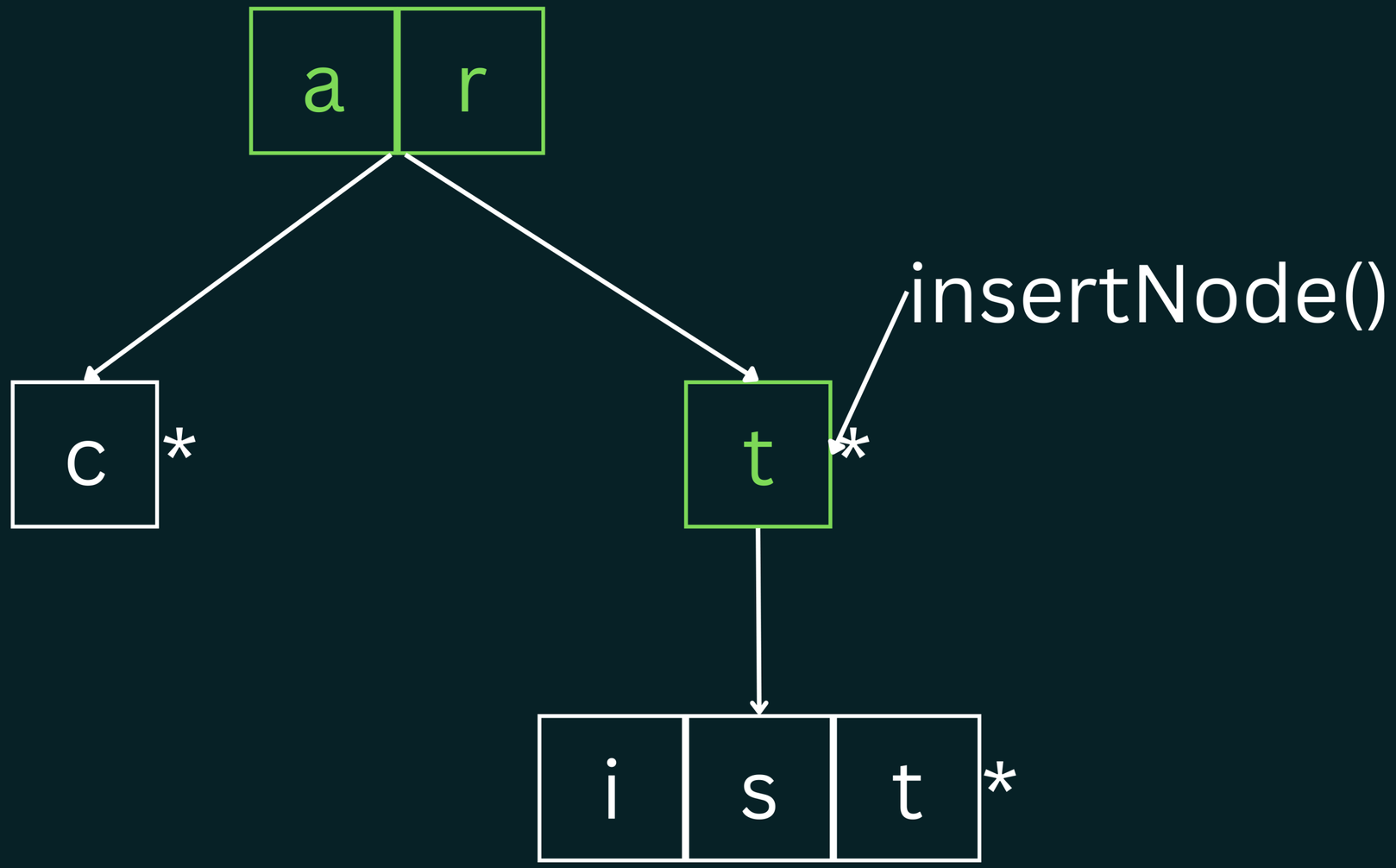

current node data
a
r
insertNode()
c
*
t
*
i
s
t
*
string to be inserted
a
r
t
i
s
*

current node data

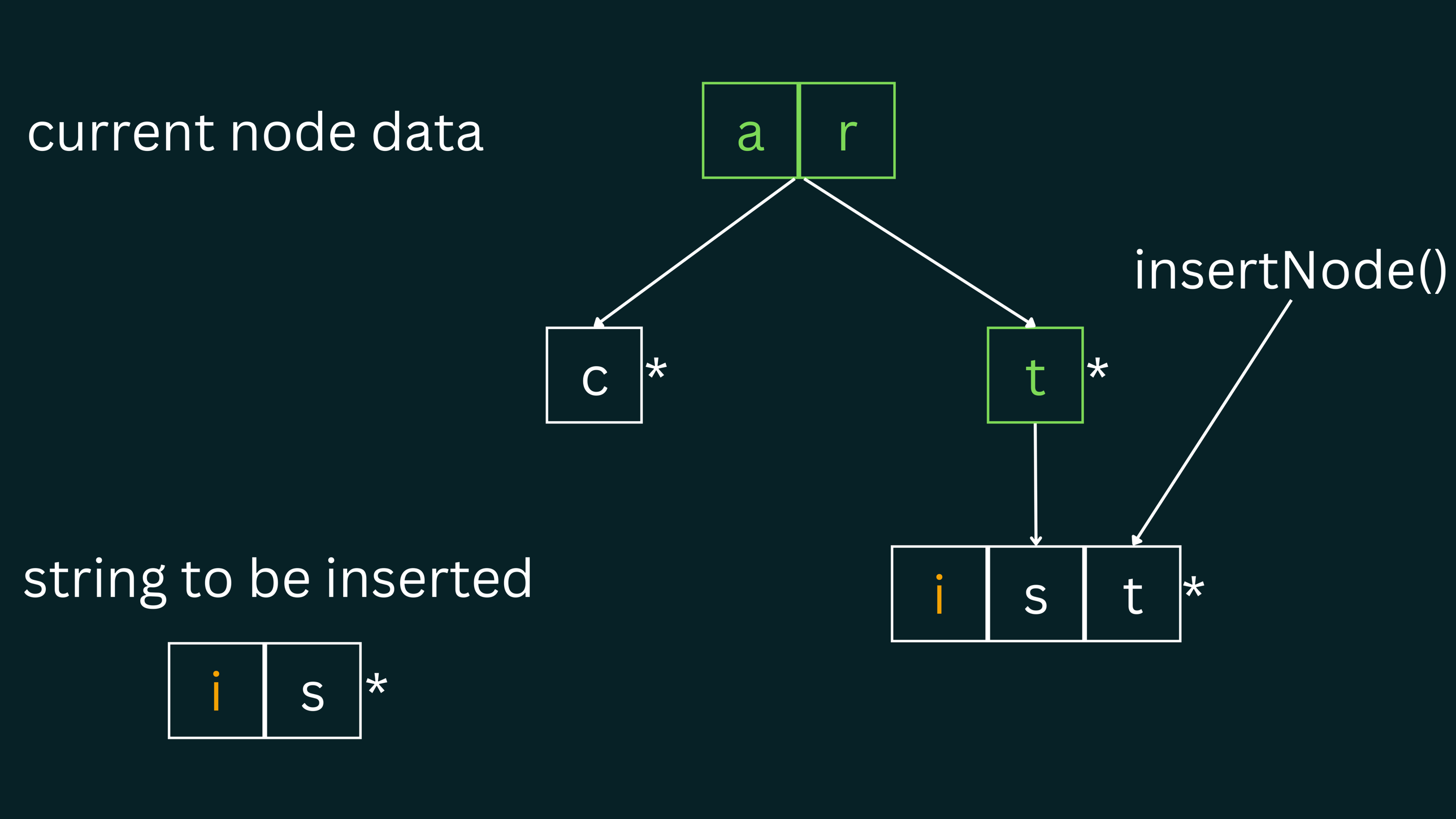


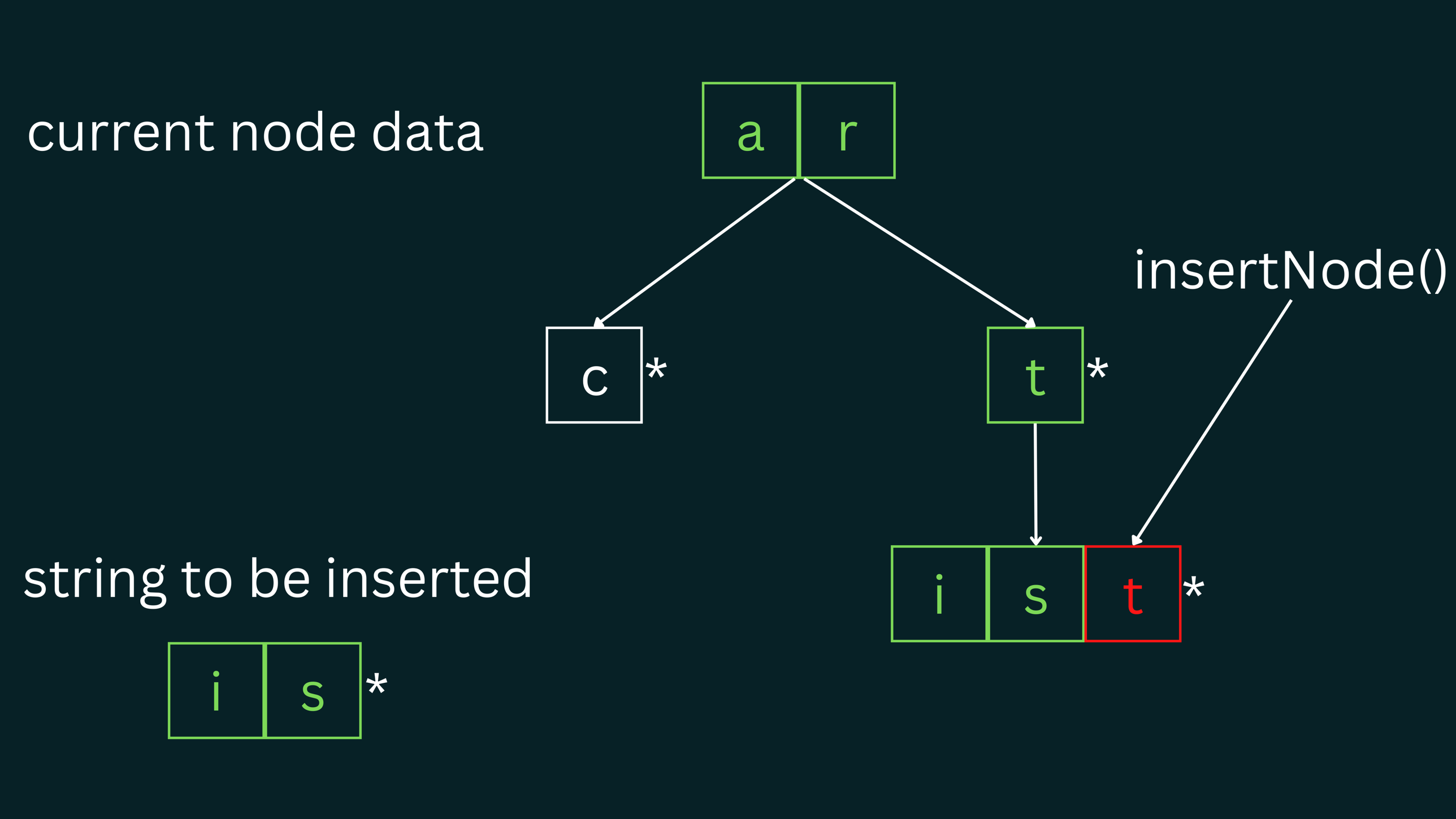
string to be inserted

current node data

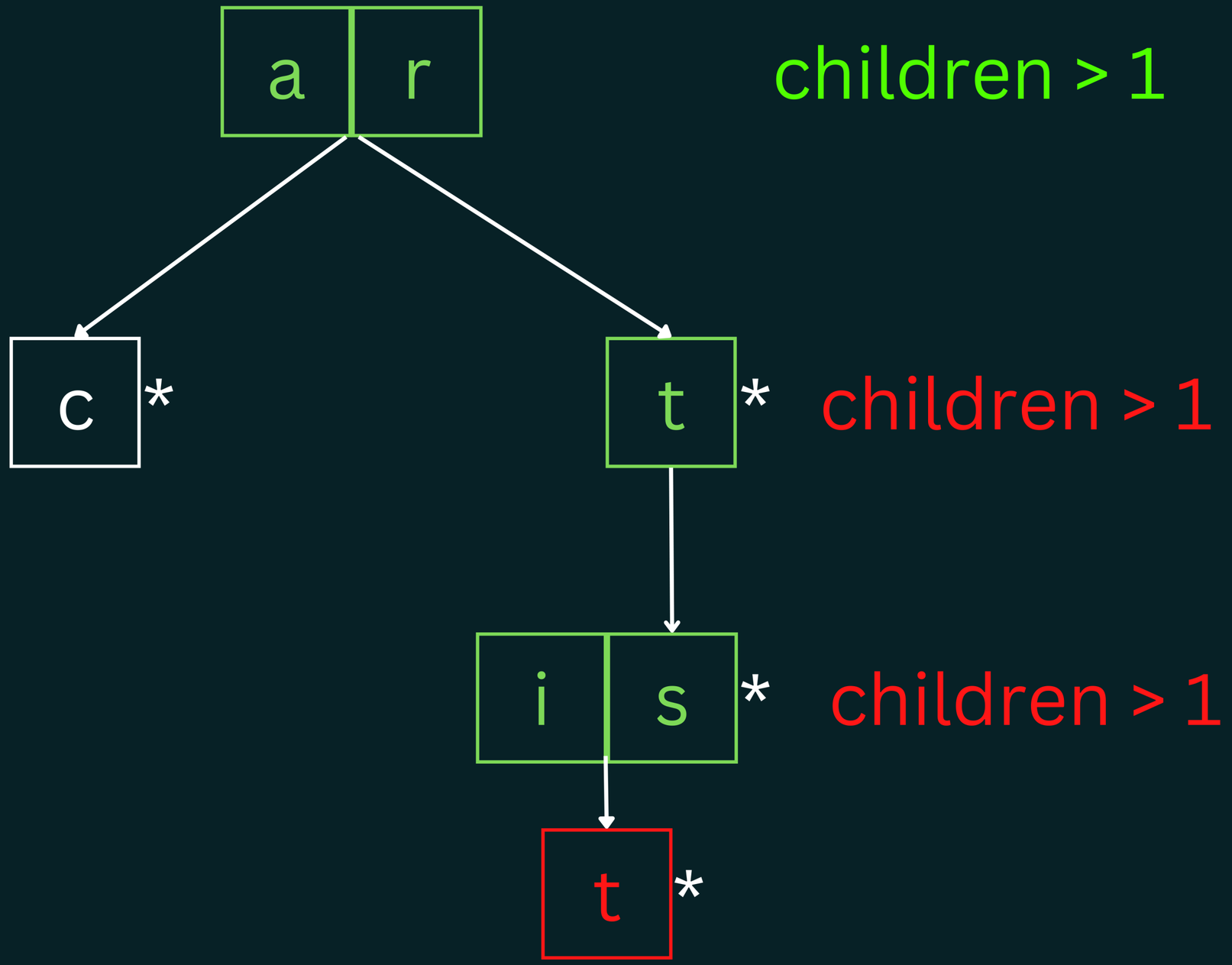


string to be inserted


current node data
a
r
c
*
t
*
insertNode()
string to be inserted
i
s
*
i
s
t
*


current node data
a
r
c
*
t
*
insertNode()
string to be inserted
i
s
*
i
s
t
*

current node data



newly inserted
node data

Cases for Compressed Trie Insertion

Case 1

This is the case when the string to be inserted and the data string of the current node are not a substring of one or the other. The portion of both strings that are similar are retained on the current node and the differing portions are put on 2 different nodes.

Case 2

This is the case when the data of the current node is a substring of the string to be inserted. If the correct position is initialized, the excess portion of the string is send to the node on the approprite position otherwise the string will be put on a new node at position.

Case 3

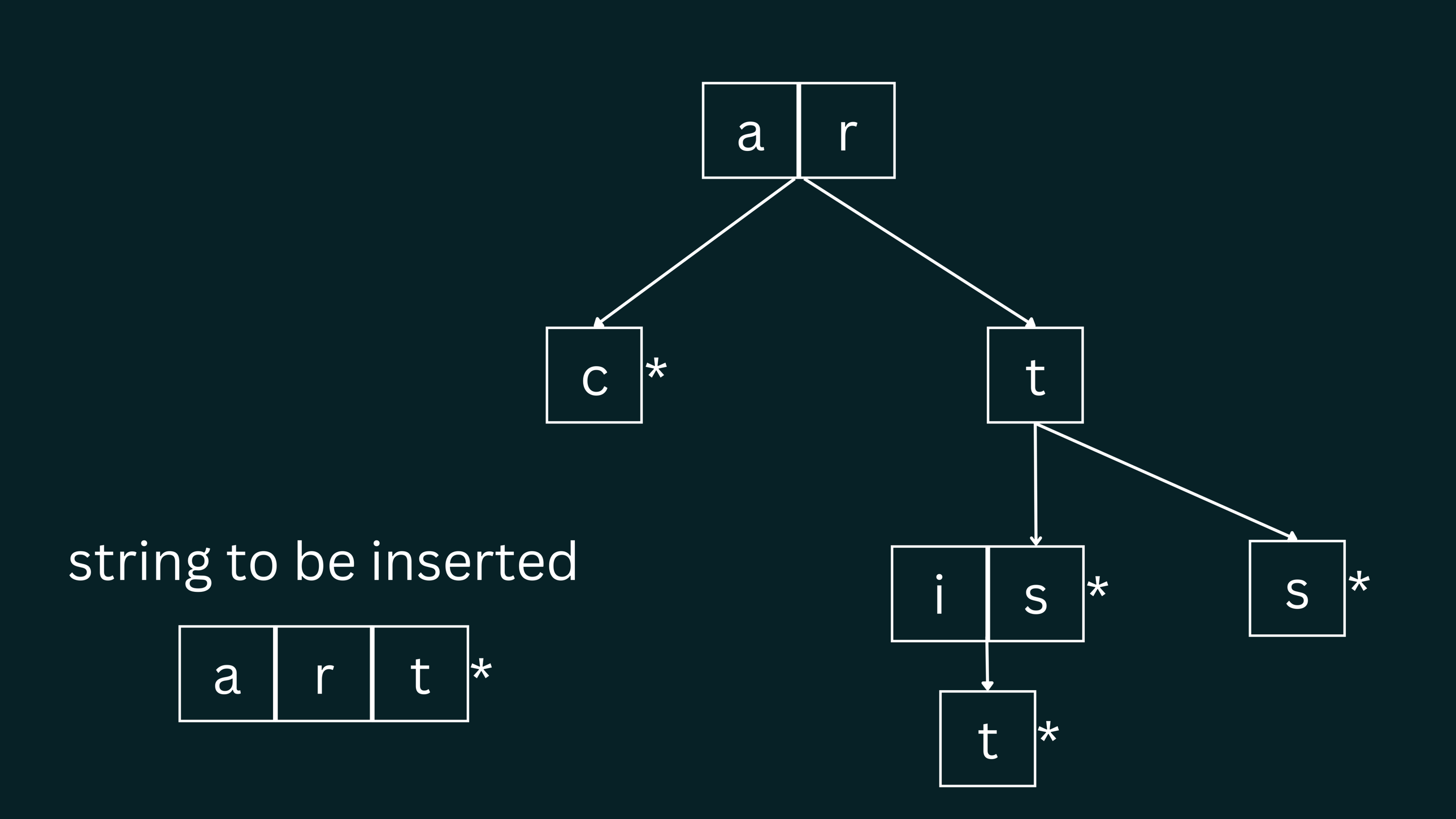
This is the case when the string to be inserted is a substring of the data of the current node. The excess portion of the data string is put on a new node.

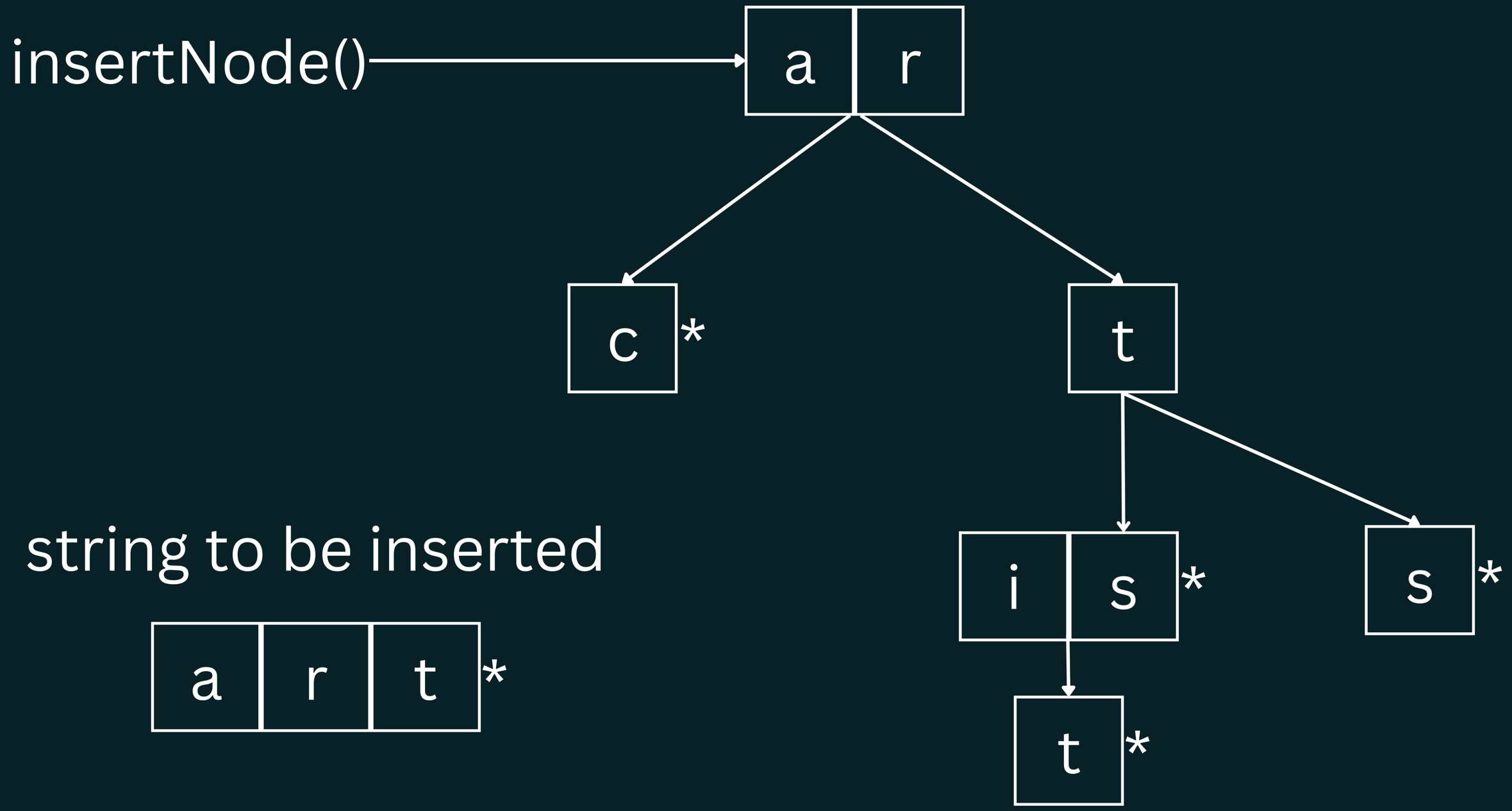
Case 4

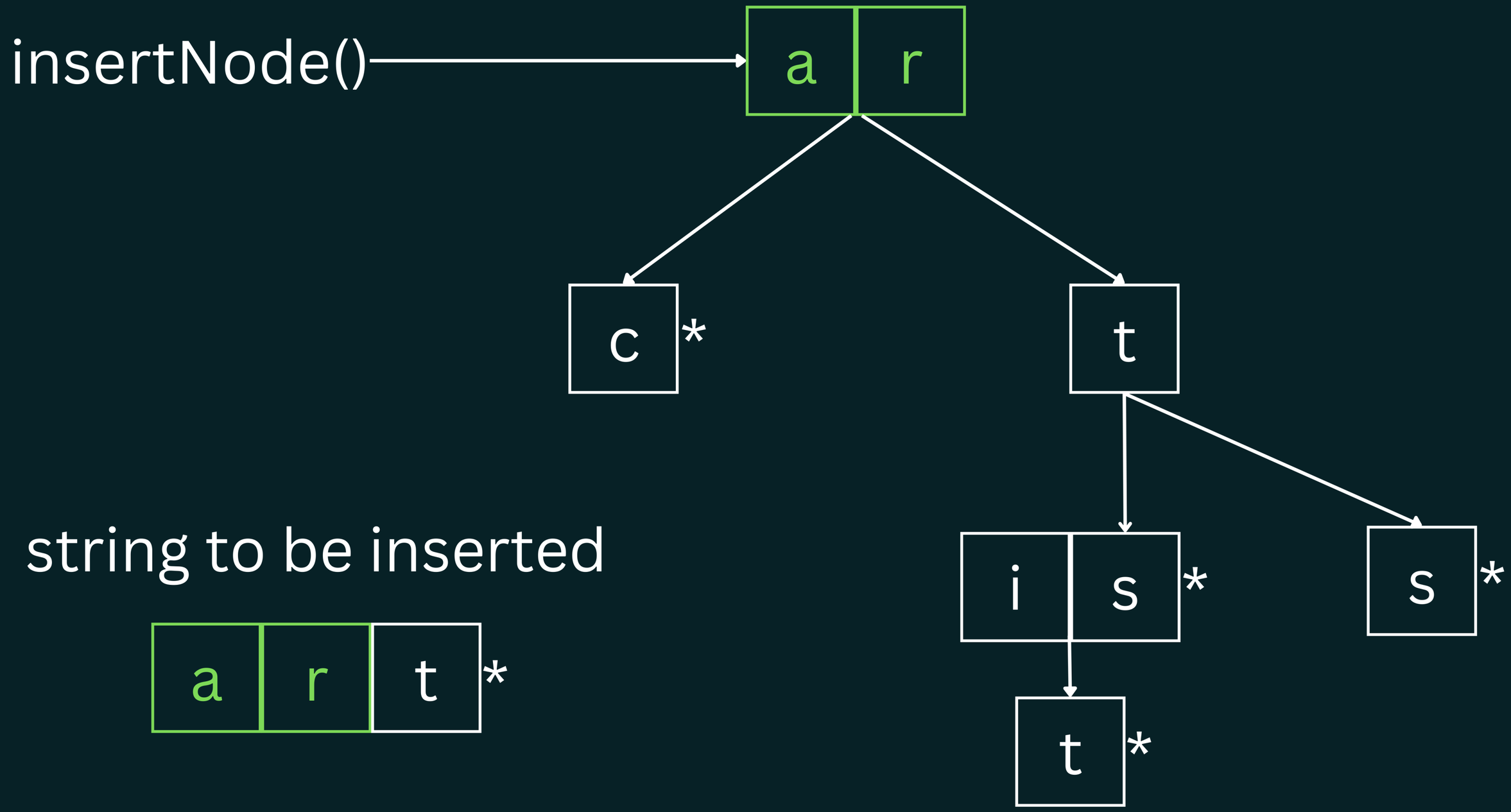
This is the case when the string to be inserted is identical to the data string of the current node. The current node's isEnd flag is set to 1.

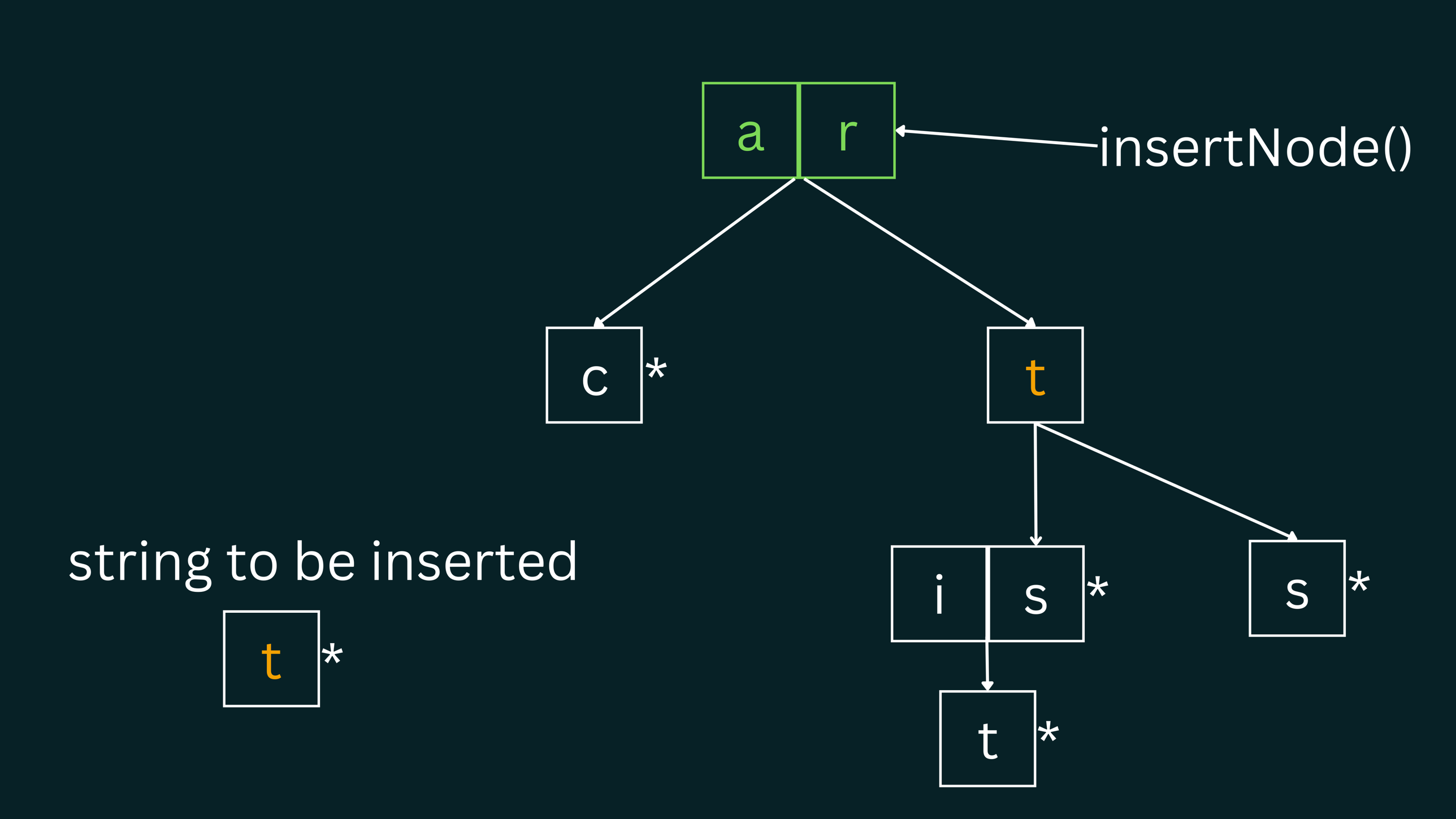
INSERT FUNCTION (Streamlined)

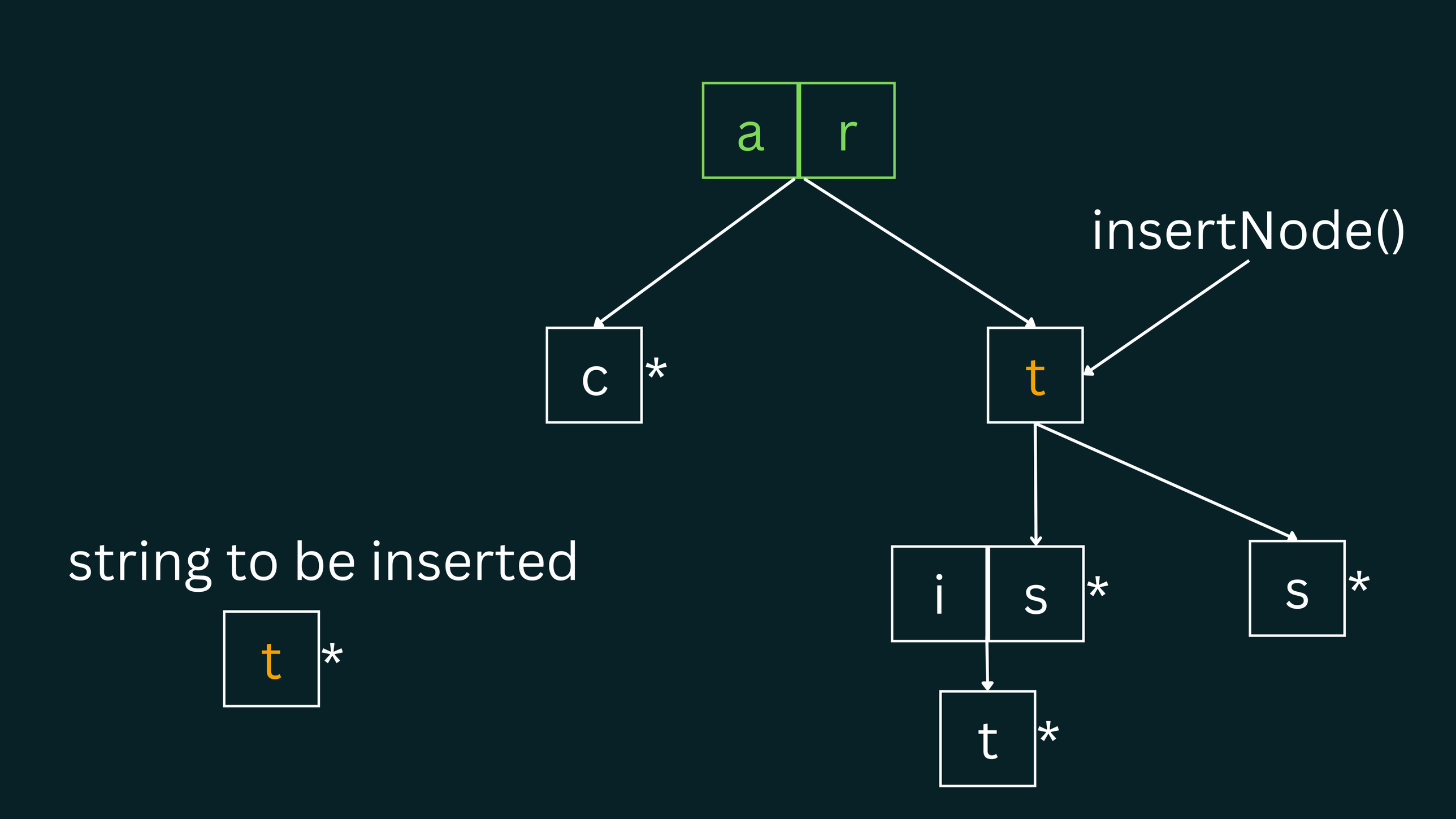
```
void insertNode(triePtr head, String s){
   int currCtr;

   //checking same characters and stops where similarities stop
   for(currCtr = 0; s[currCtr] == head->data[currCtr] && head->data[currCtr] != '\0'; currCtr++);

   if(currCtr < strlen(head->data) && currCtr < strlen(s)){
      insertCase1(currCtr, head, s, tempArray);
   }else if(strlen(head->data) < strlen(s)){
      insertCase2(currCtr, head, s);
   }else if(strlen(s) < strlen(head->data)){
      insertCase3(currCtr, head, s, tempArray);
   }else if(currChildren > 1 && (s[strCtr] == (*head)->data[dataCtr])){
      head->isEnd = 1;
   }
}
```

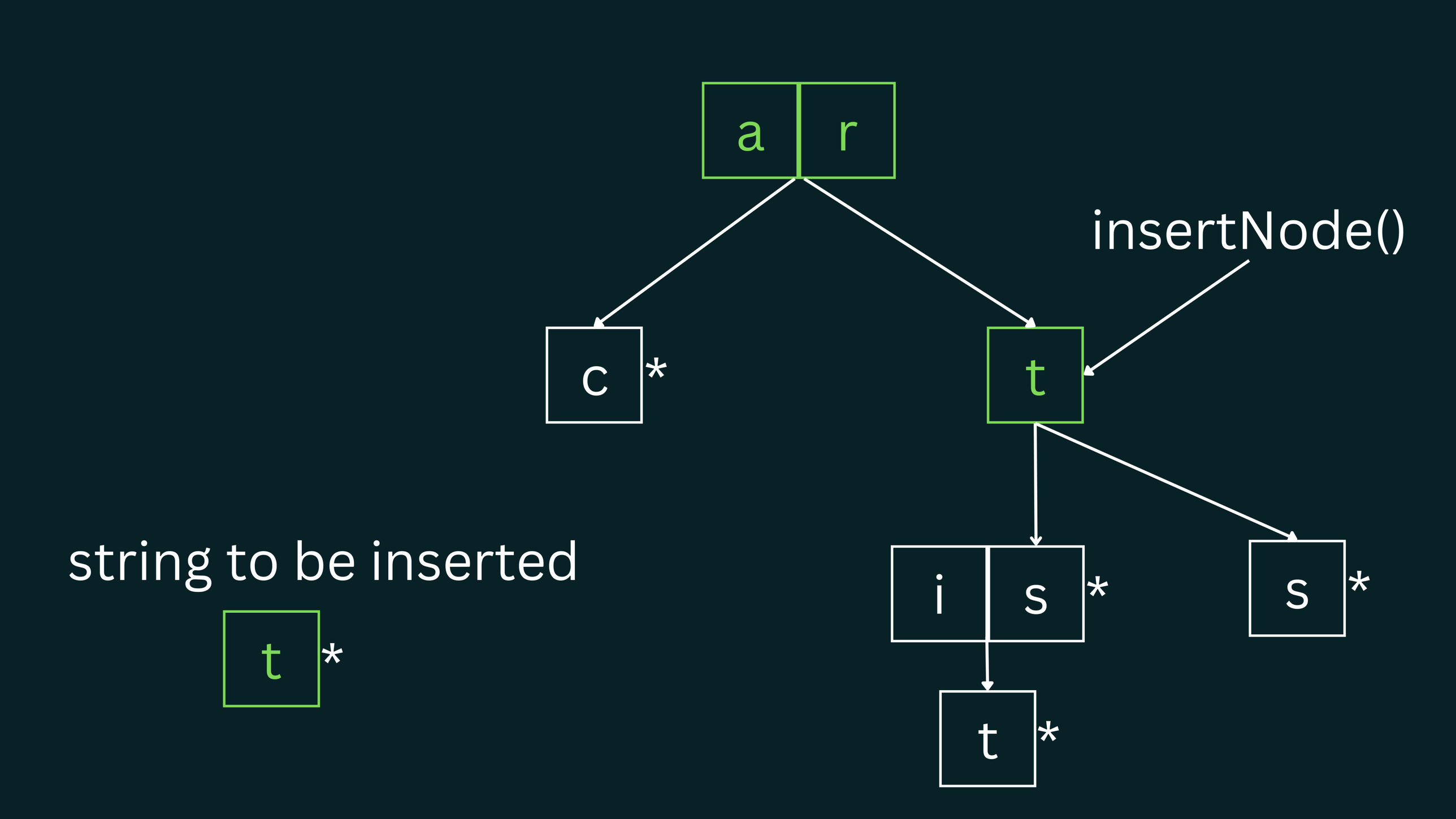

a
r
c
*
t
i
s
*
s
*
t
*
string to be inserted
a
r
t
*

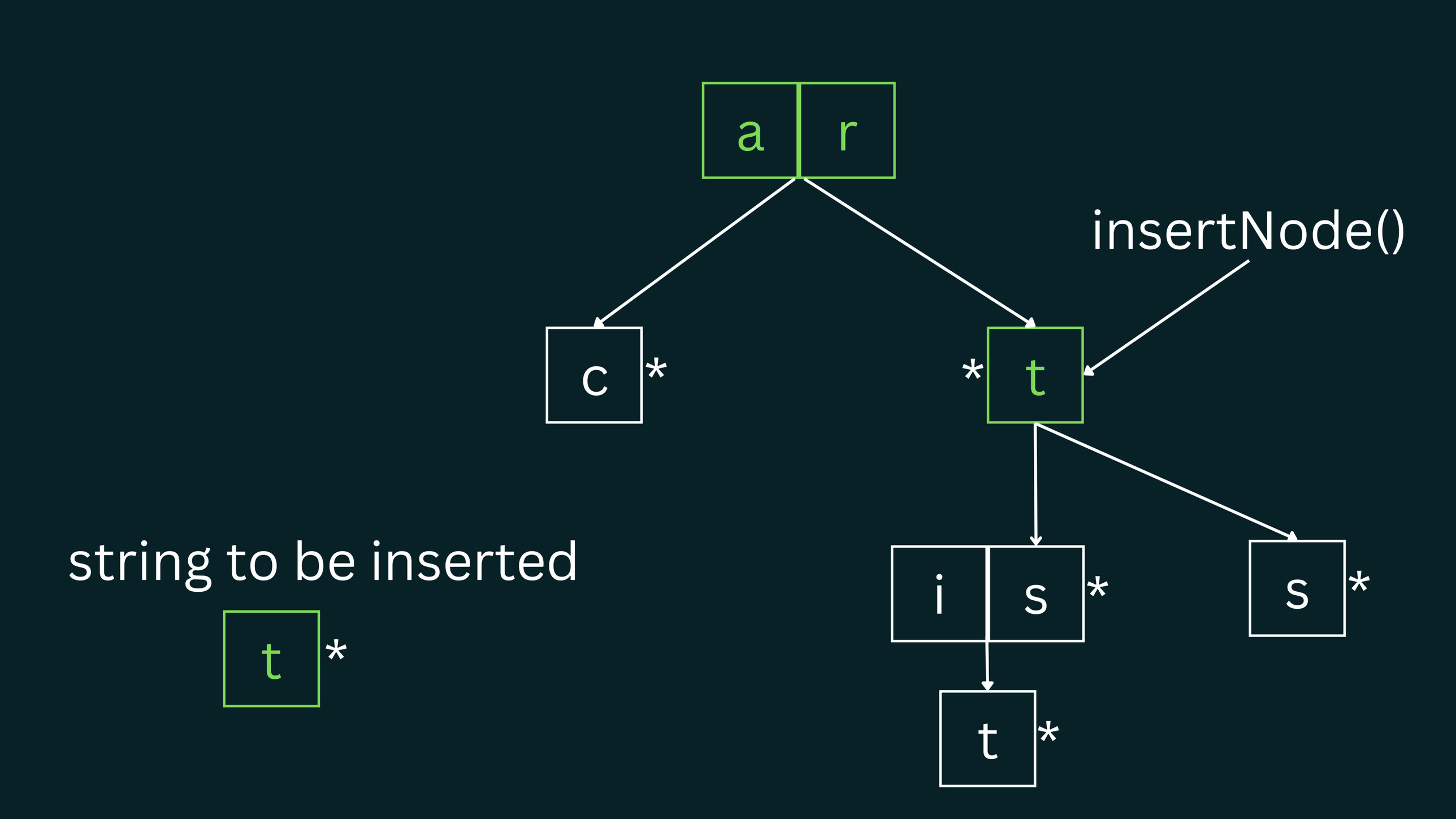

insertNode()
a
r
c
*
t
string to be inserted
a
r
t
*
i
s
*
s
*
t
*


insertNode()
a
r
c
*
t
string to be inserted
a
r
t
*
i
s
*
s
*
t
*

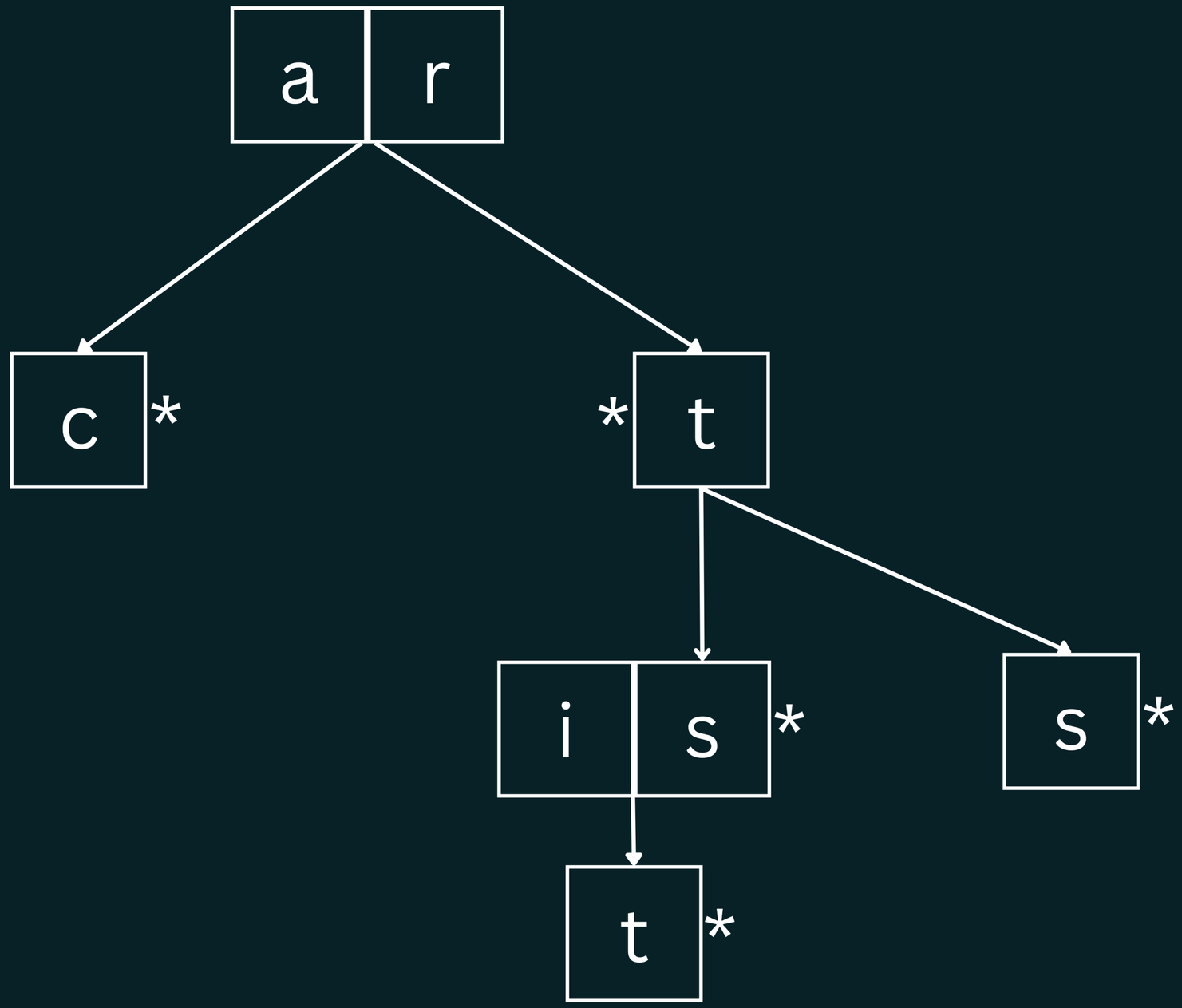

a
r
insertNode()
c
*
t
string to be inserted
t
*
i
s
*
s
*
t
*


a
r
insertNode()
c
*
t
string to be inserted
t
*
i
s
*
s
*
t
*


a
r
c
*
t
insertNode()
string to be inserted
t
*
i
s
*
s
*
t
*


a
r
c
*
*
t
insertNode()
string to be inserted
t
*
i
s
*
s
*
t
*

new trie



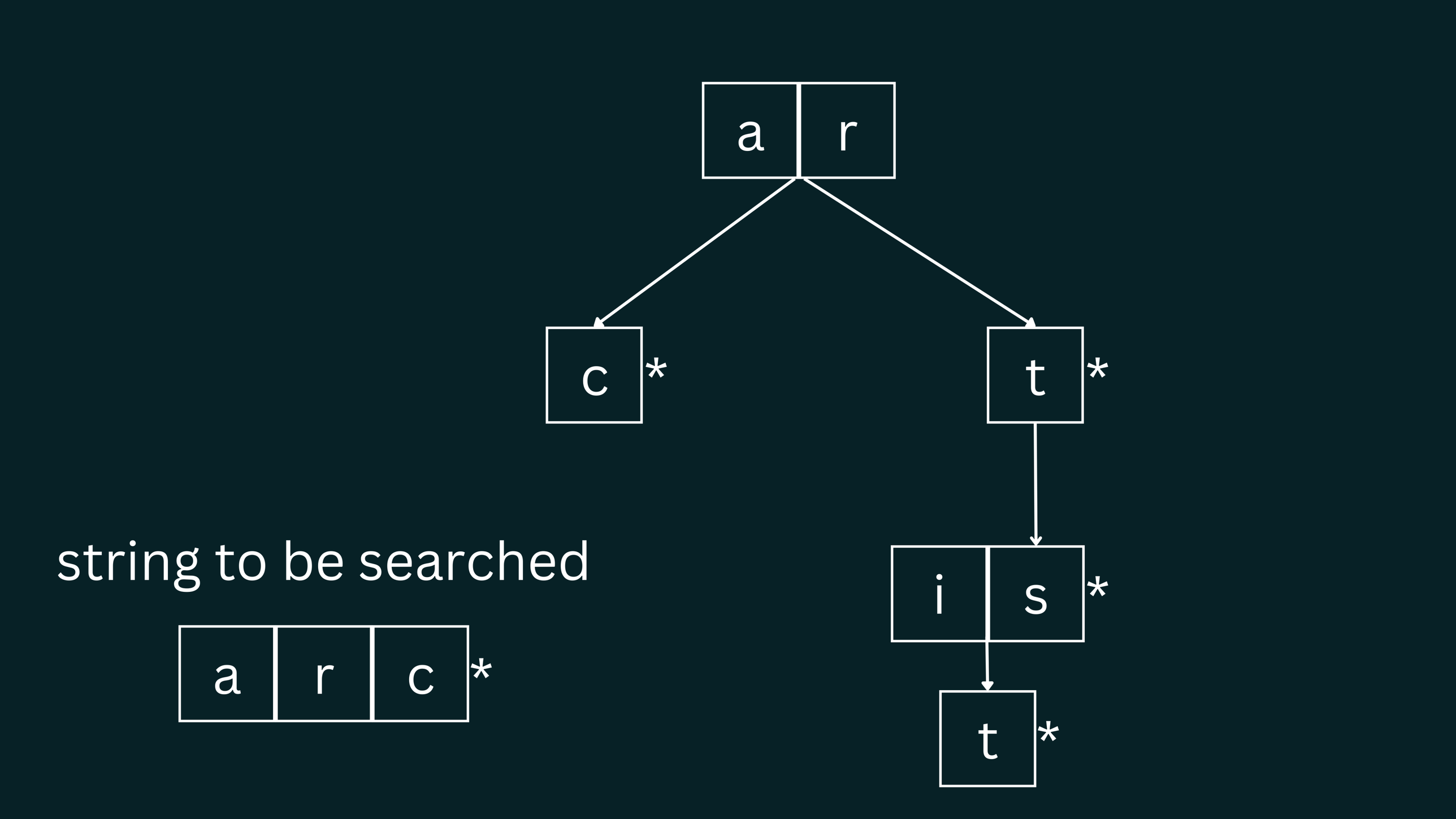
Search Function

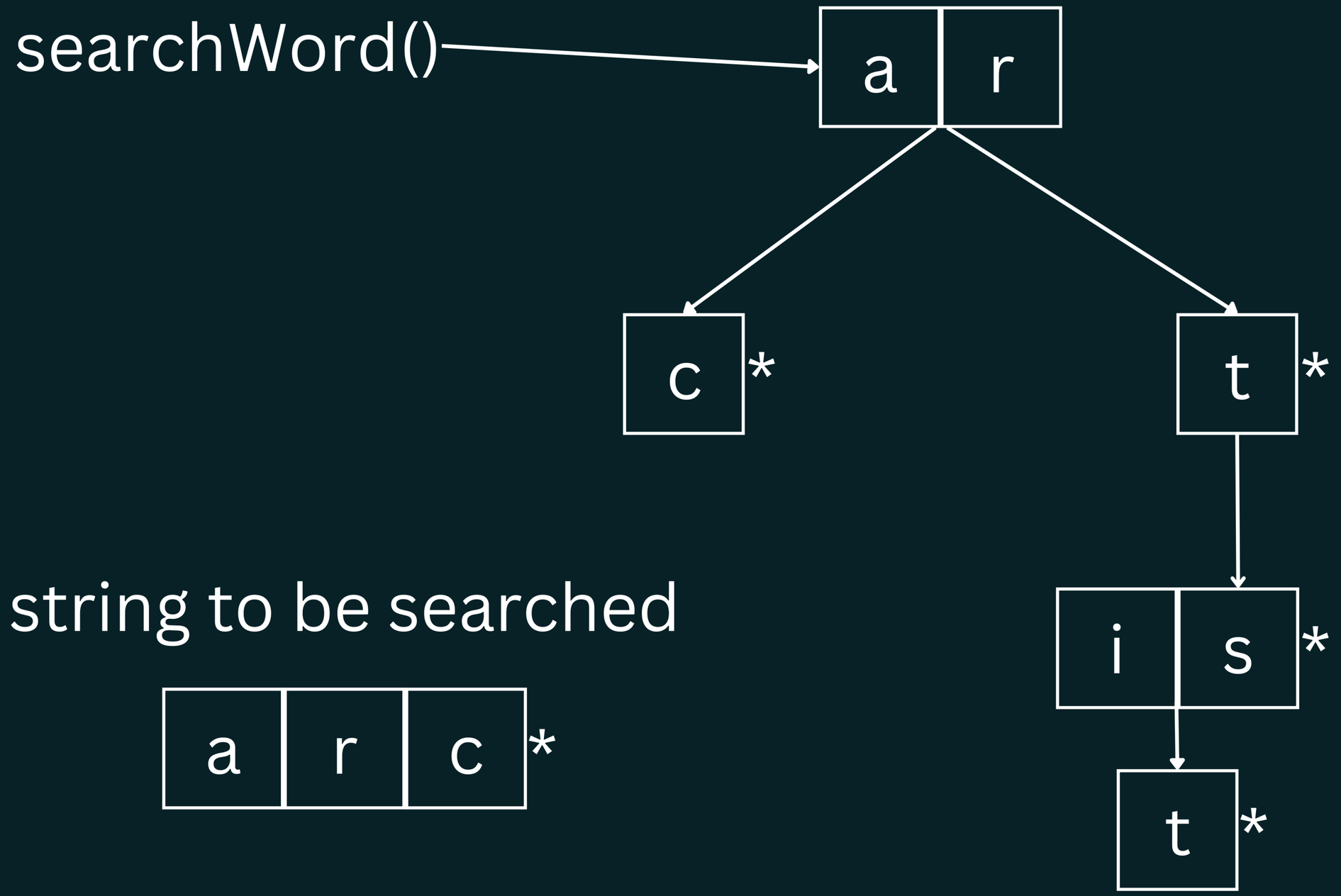
SEARCH FUNCTION

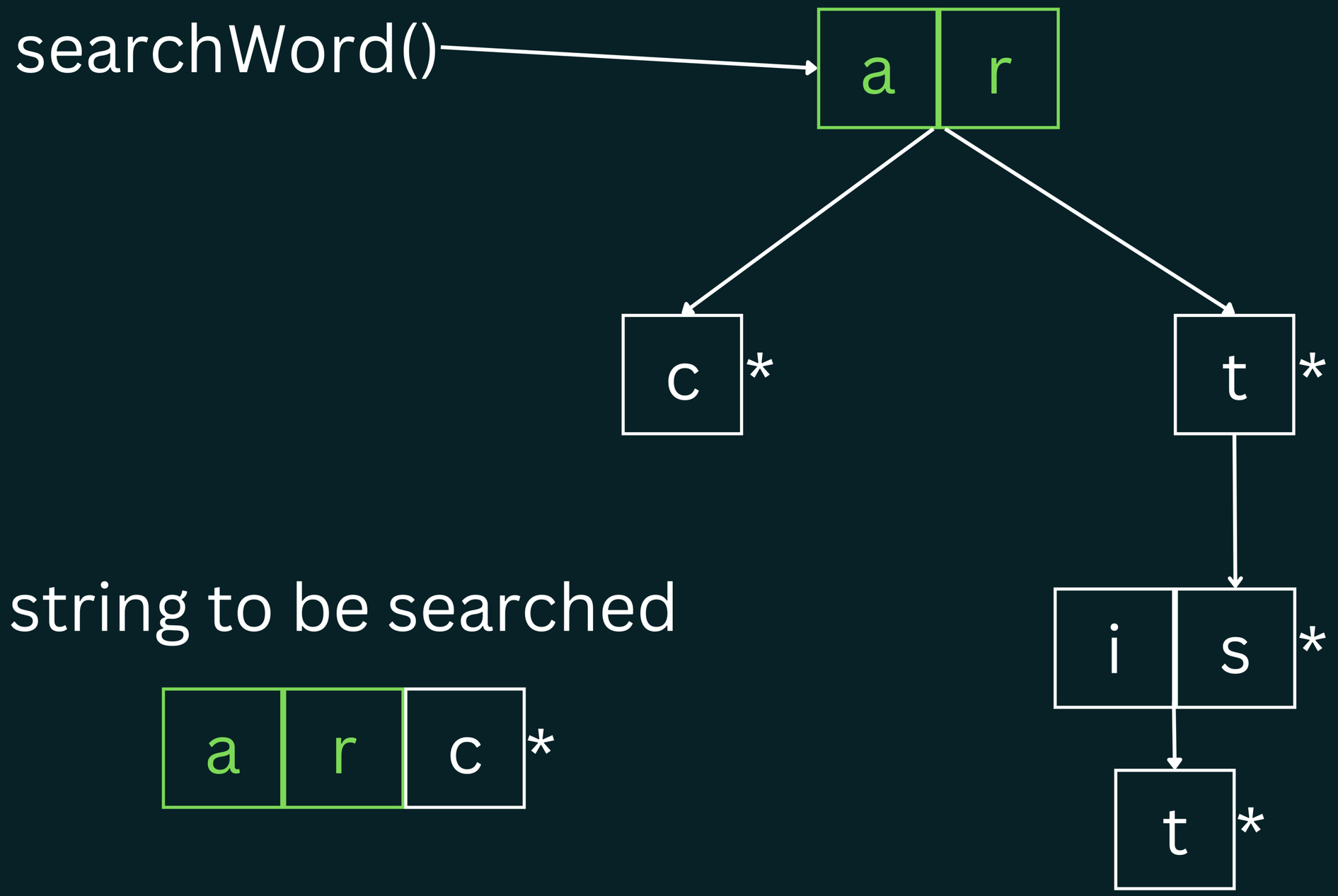
```
int search(triePtr root, char word[]){
    triePtr trav = NULL;
    int x, index = 0;

    // traverse the given word
    for(x = 0, trav = root; word[x] != '\0'; x++, trav = trav->children[index]){
        index = getIndex(word[x]);

        // check if the current letter exists
        if(trav->children[index] == NULL){
            return 0;
        }
    }

    return trav->isEndofWord; //returns 1 if found and 0 if not found
}//INTERNET CODE
```

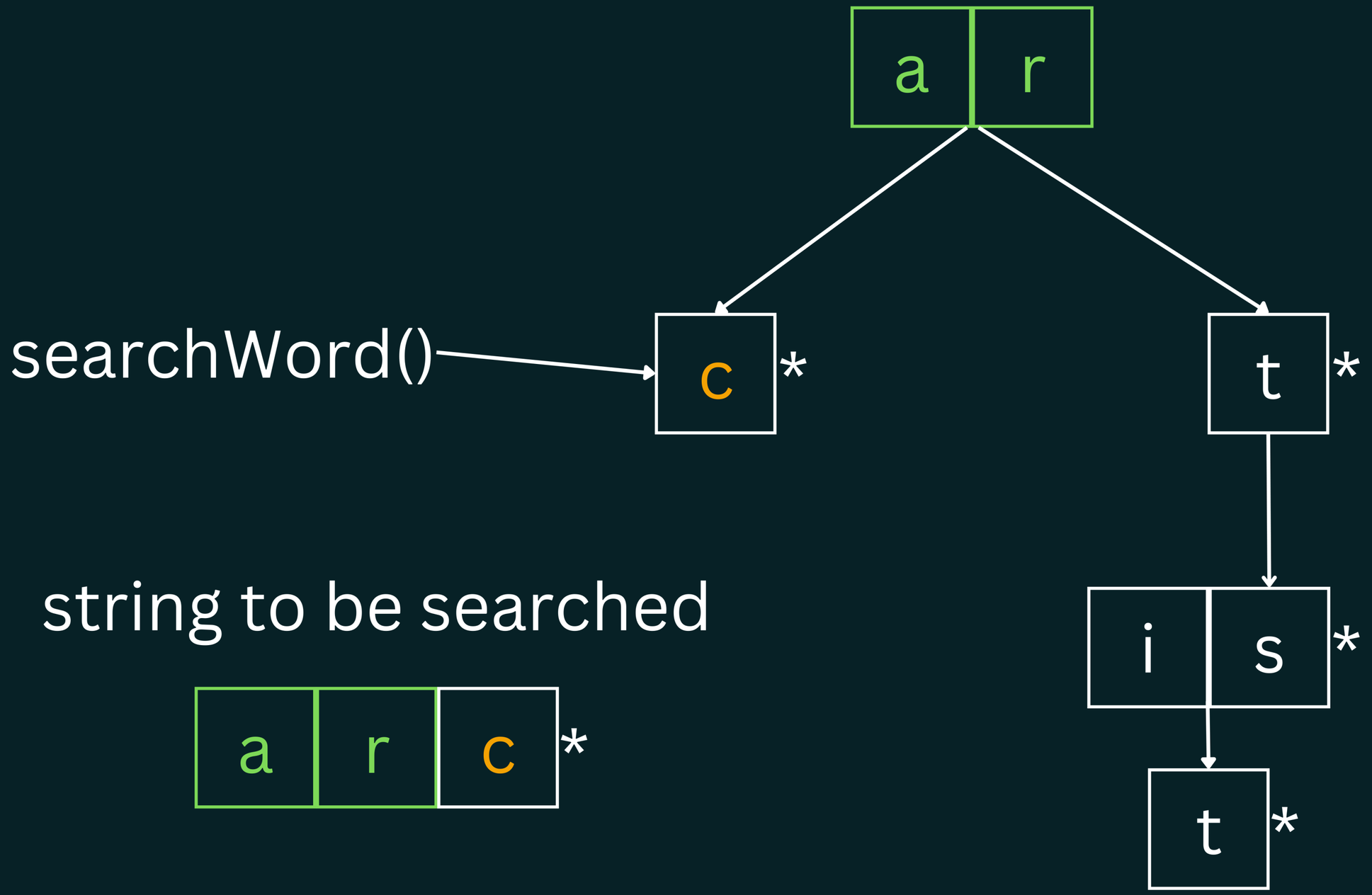
SEARCH FUNCTION

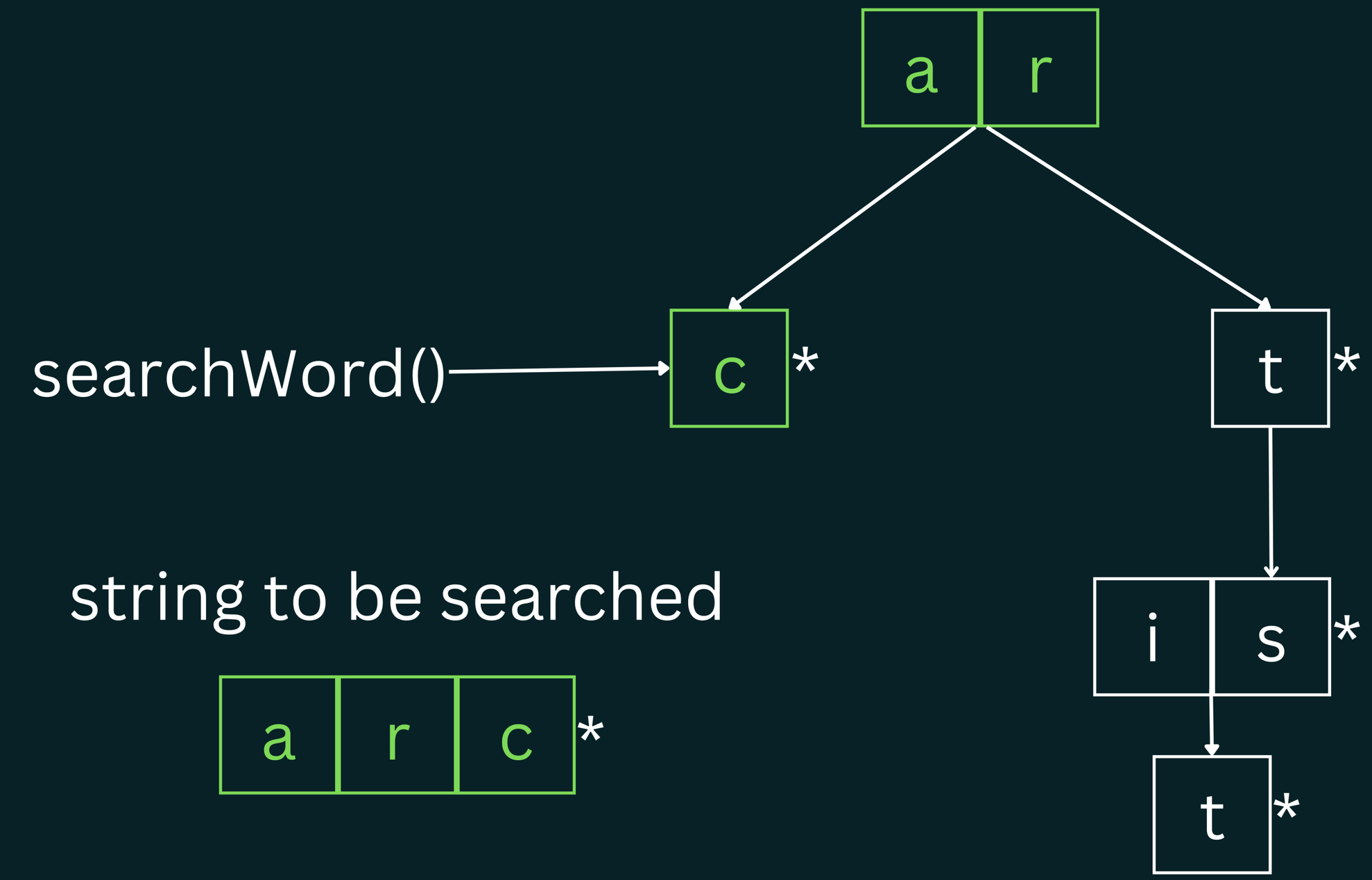
```
int searchWord(triePtr root[], String s){
   printf("\nSearching %s...", s);

   triePtr trav = root[s[0] - 'a'];
   int x = 0, y = 0, check = 1;

   while(check == 1 && s[x] != '\0'){
      x++;
      y++;

      if(trav->data[y] == '\0' && trav->next[s[x] - 'a'] != NULL && s[x] != '\0'){
         y = 0;
         trav = trav->next[s[x] - 'a'];
      }
      check = s[x] == trav->data[y] ? 1 : 0;
   }

   check = (s[x] == '\0' && trav->data[y] == '\0') ? trav->isEnd : 0;

   return check;
}
```

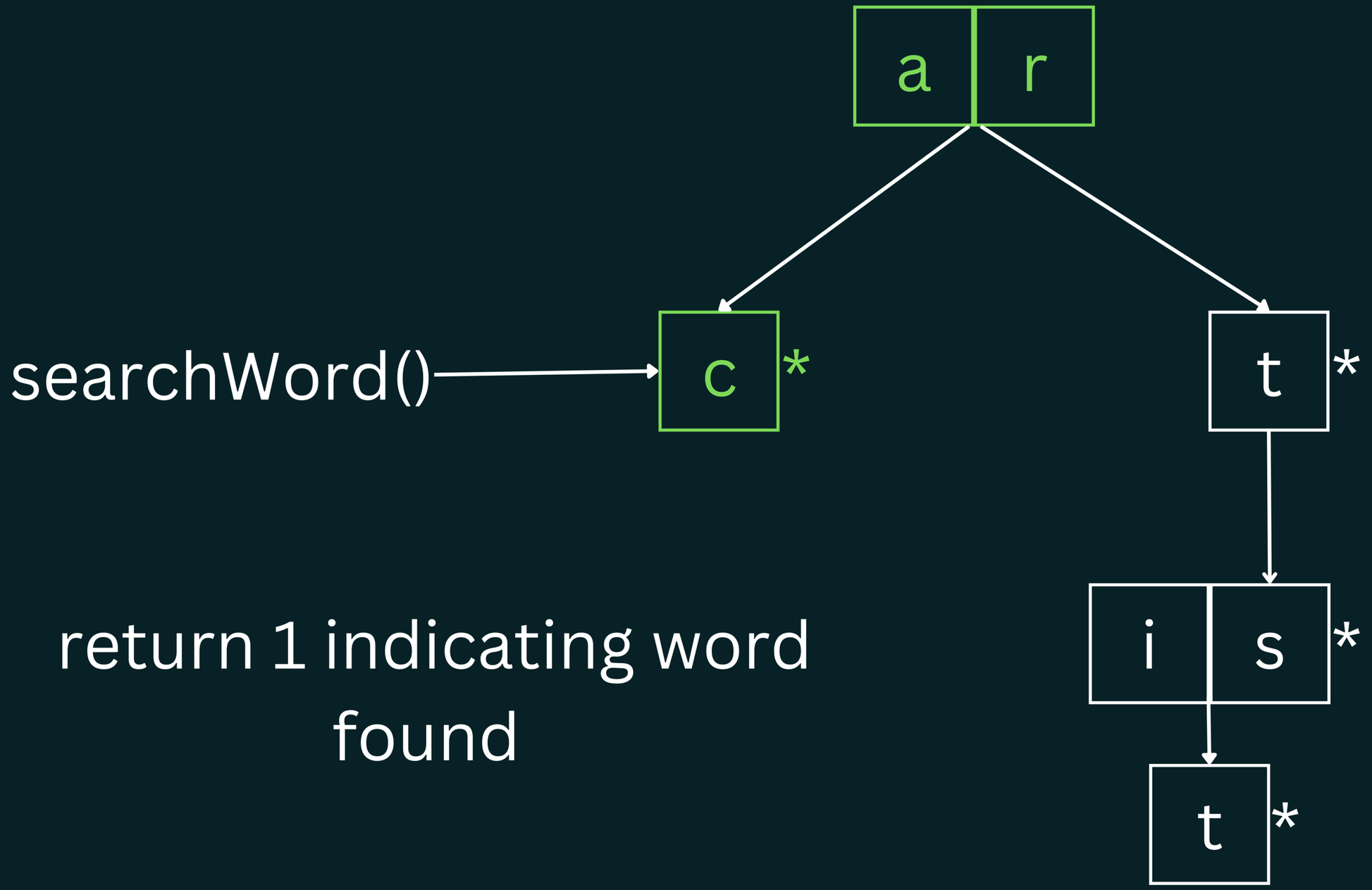

a
r
c
*
t
*
i
s
*
t
*
string to be searched
a
r
c
*


searchWord()
a
r
c
*
t
*
i
s
*
t
*
string to be searched
a
r
c
*


searchWord()
a
r
c
*
t
*
i
s
*
t
*
string to be searched
a
r
c
*


a
r
searchWord()
c
*
t
*
string to be searched
a
r
c
*
i
s
*
t
*


a
r
searchWord()
c
*
t
*
string to be searched
a
r
c
*
i
s
*
t
*



return 1 indicating word found

Delete Function

DELETE FUNCTION

```
void deleteWord(triePtr *root, char word[], int letter){
    if(letter == strlen(word) + 1){
        return;
    }

    int index = getIndex(word[letter]);
    deleteWord(&(*root)->children[index], word, letter + 1);

    triePtr *trav = root, temp = *trav;
    if(letter == strlen(word) && temp->isEndofWord == 1){
        temp->isEndofWord = 0;

        if(isPrefix(temp) == 0){
            *trav = NULL;
            free(temp);
            temp = NULL;
        }
    }

    if(isPrefix(temp) == 0 && temp->isEndofWord == 0){
        *trav = NULL;
        free(temp);
        temp = NULL;
    }
}//INTERNET CODE
```

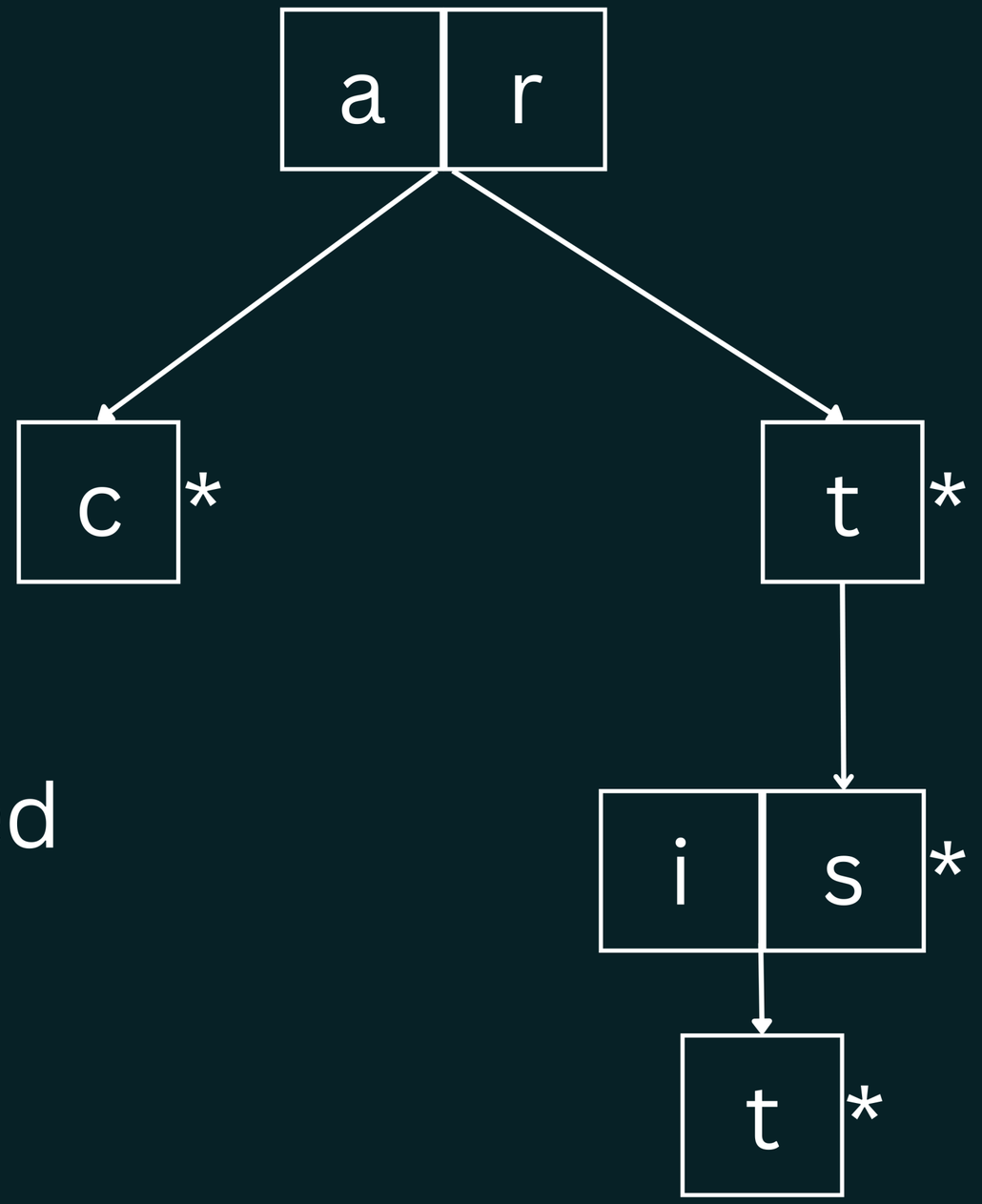
DELETE FUNCTION

```
void deleteWord(triePtr root[], String s){
   if(searchWord(root, s) > 0)
      deleteNode(root[s[0] - 'a'], s, 0);
}
```

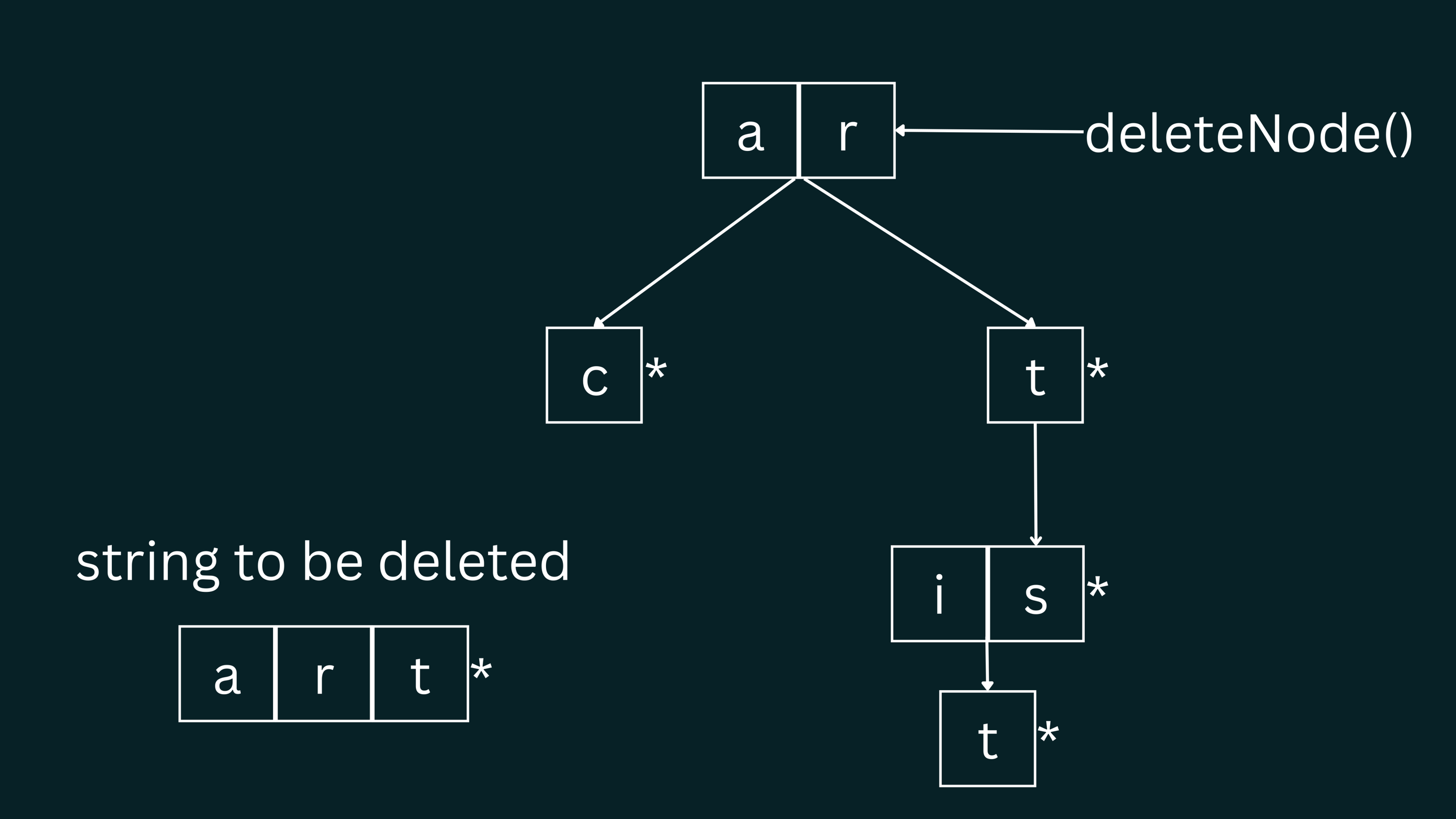
```
void deleteNode(triePtr *head, String s, int curr){
   int dataCtr, strCtr;

   //checking same characters
   for(dataCtr = 0, strCtr = curr;
         s[strCtr] == (*head)->data[dataCtr] && (*head)->data[dataCtr] != '\0'; dataCtr++, strCtr++);

   if(s[strCtr] != (*head)->data[dataCtr])
      deleteNode(&(*head)->next[tolower(s[strCtr]) - 'a'], s, strCtr);

   int currChildren, x;
   triePtr *child = NULL;
   for(x = currChildren = 0; x < ALPHABET_SIZE; x++)
      if((*head)->next[x] != NULL){
         currChildren++;
         child = &(*head)->next[x];
      }

   if(currChildren == 1){
      mergeChild(head, child);
   }else if(currChildren == 0 && (*head)->isEnd != 1){
      freeNode(head);
   }else if(currChildren > 1 && (s[strCtr] == (*head)->data[dataCtr])){
      (*head)->isEnd = 0;
   }
}
```

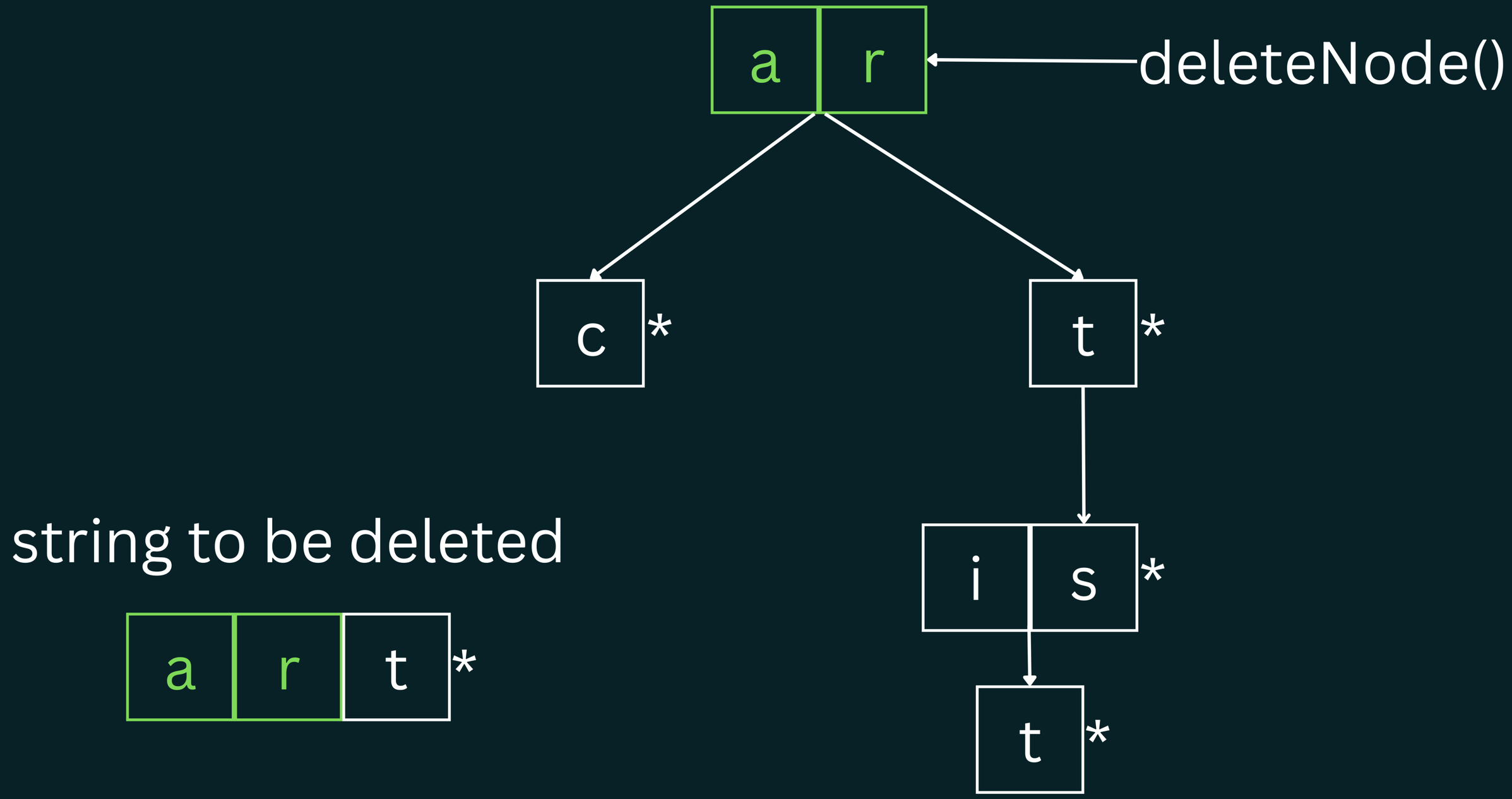
DELETE FUNCTION

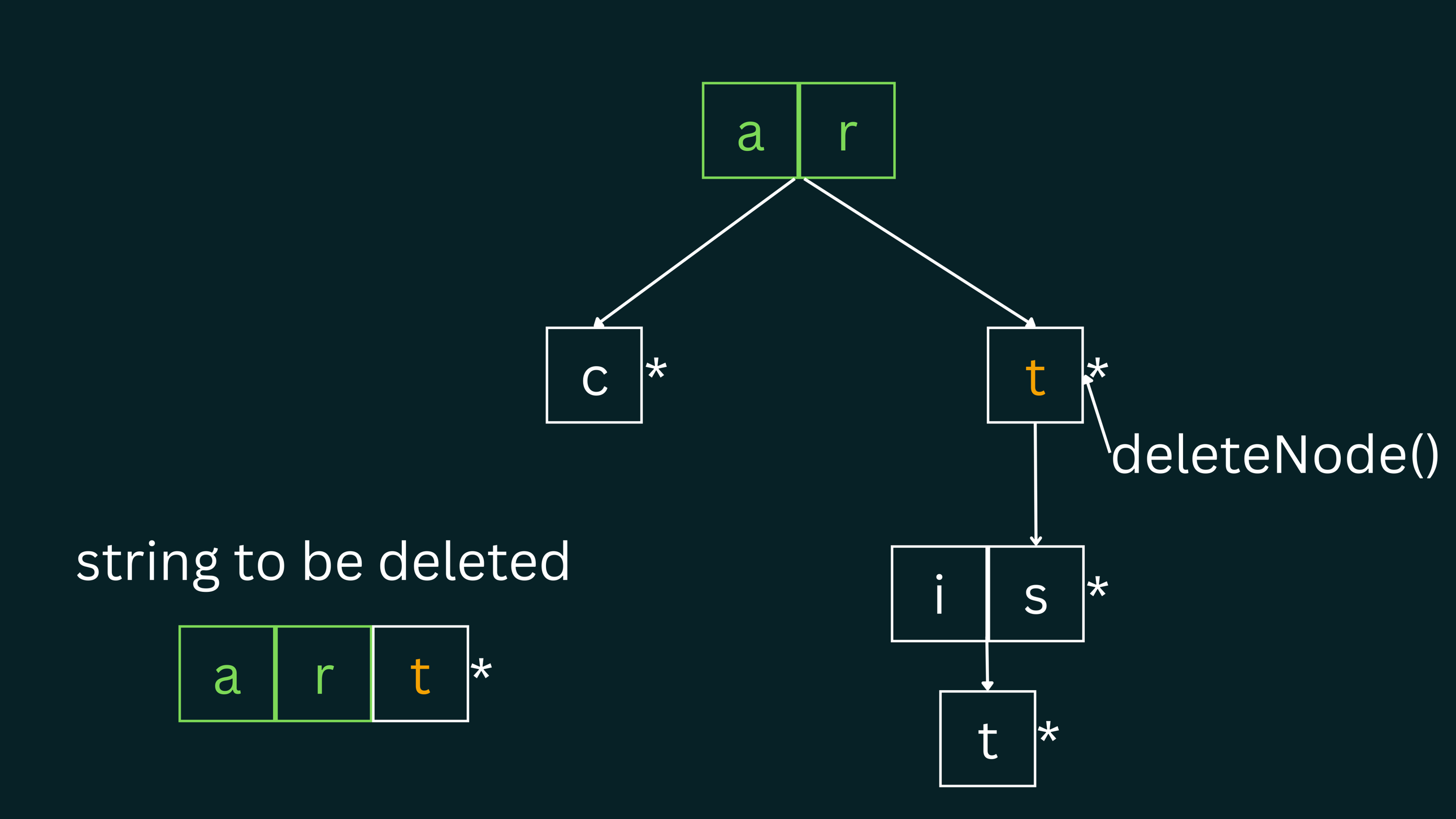
```
void mergeChild(triePtr *head, triePtr *child){
    triePtr tempArray[ALPHABET_SIZE];

    /*Transfer child data to head*/
    memcpy(tempArray, (*child)->next, sizeof(tempArray));
    strcat((*head)->data, (*child)->data);
    (*head)->isEnd = (*child)->isEnd;

    free(*child);
    *child = NULL;

    memcpy((*head)->next, tempArray, sizeof(tempArray));
}

void freeNode(triePtr *head){
    free(*head);
    *head = NULL;
}
```

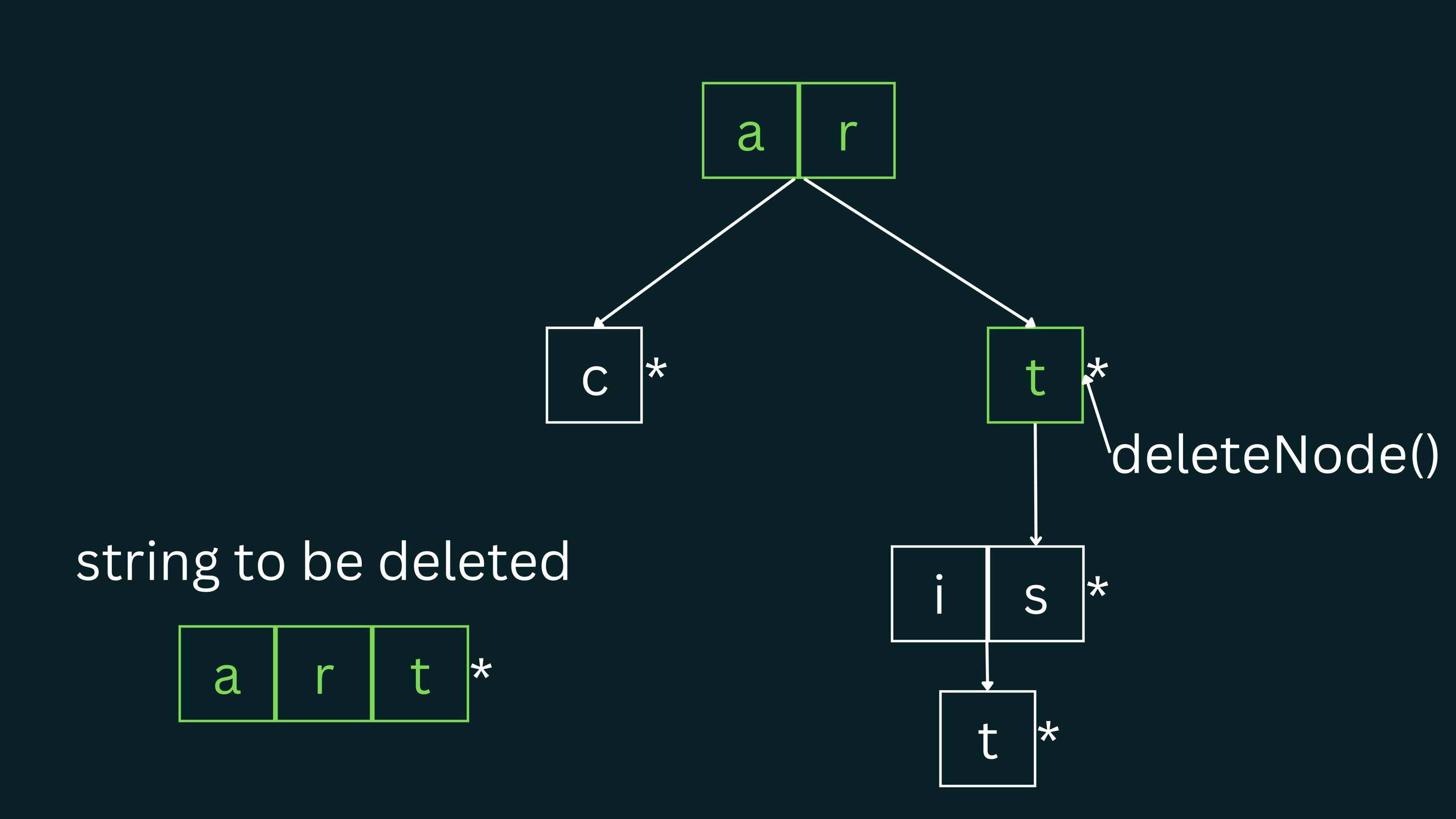


string to be deleted


a
r
deleteNode()
c
*
t
*
i
s
*
t
*
string to be deleted
a
r
t
*


a
r
deleteNode()
c
*
t
*
string to be deleted
a
r
t
*
i
s
*
t
*


a
r
c
*
t
*
deleteNode()
string to be deleted
a
r
t
*
i
s
*
t
*

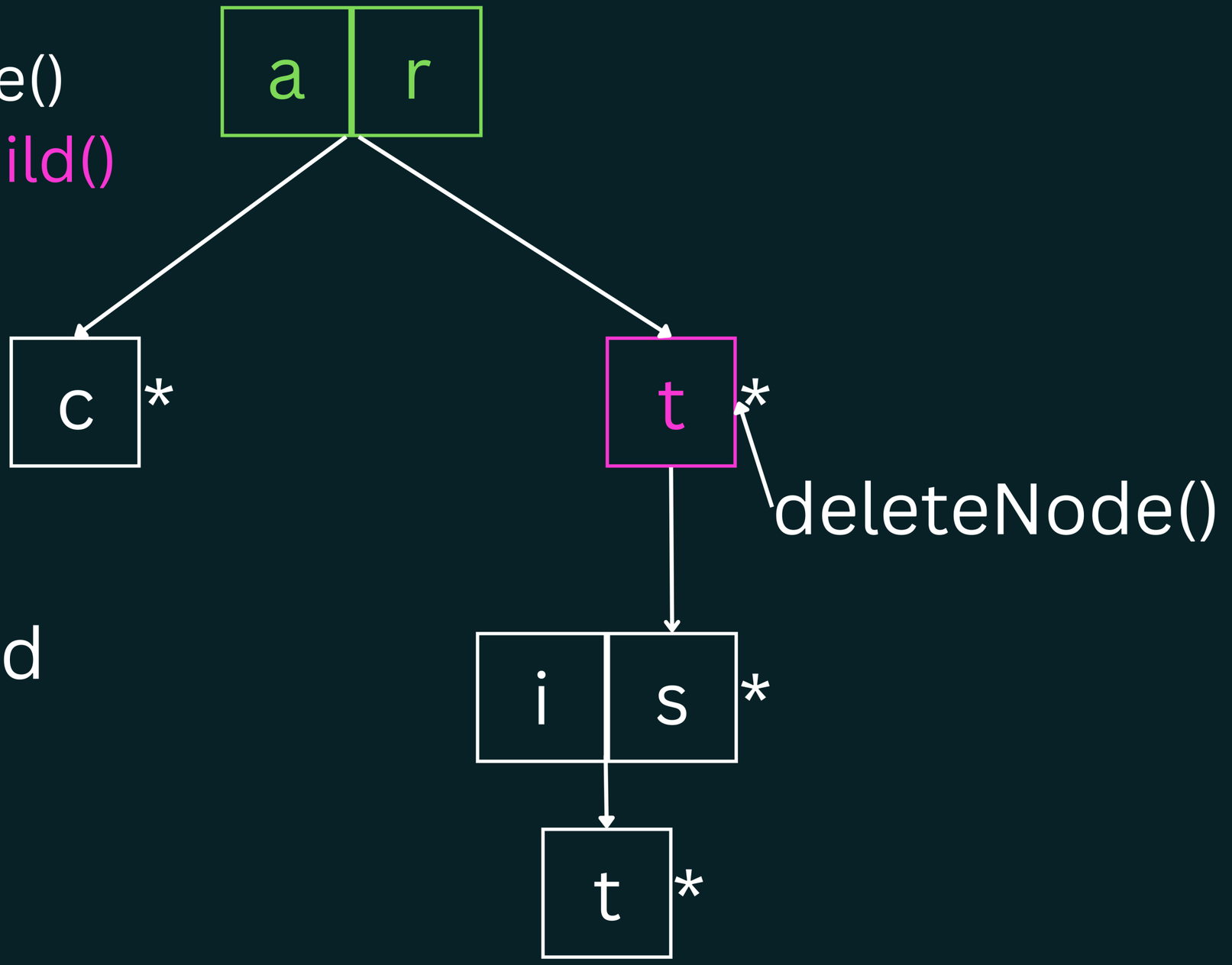

a
r
c
*
t
*
deleteNode()
string to be deleted
a
r
t
*
i
s
*
t
*

Outcome Deletion

child = 0 => freeNode()

child = 1 => mergeChild()

child > 1 => isEnd = 0



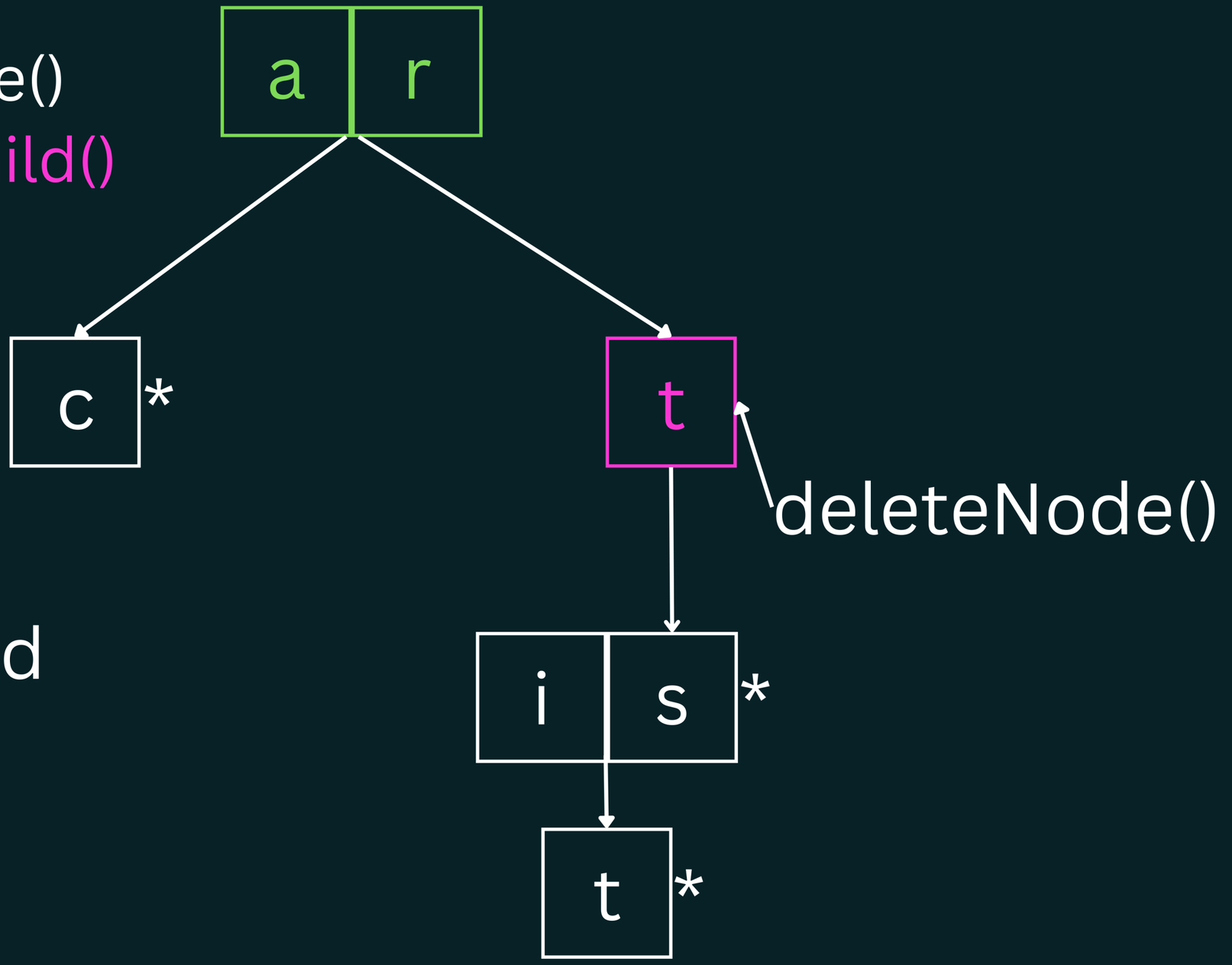
string to be deleted

Outcome Deletion

child = 0 => freeNode()

child = 1 => mergeChild()

child > 1 => isEnd = 0



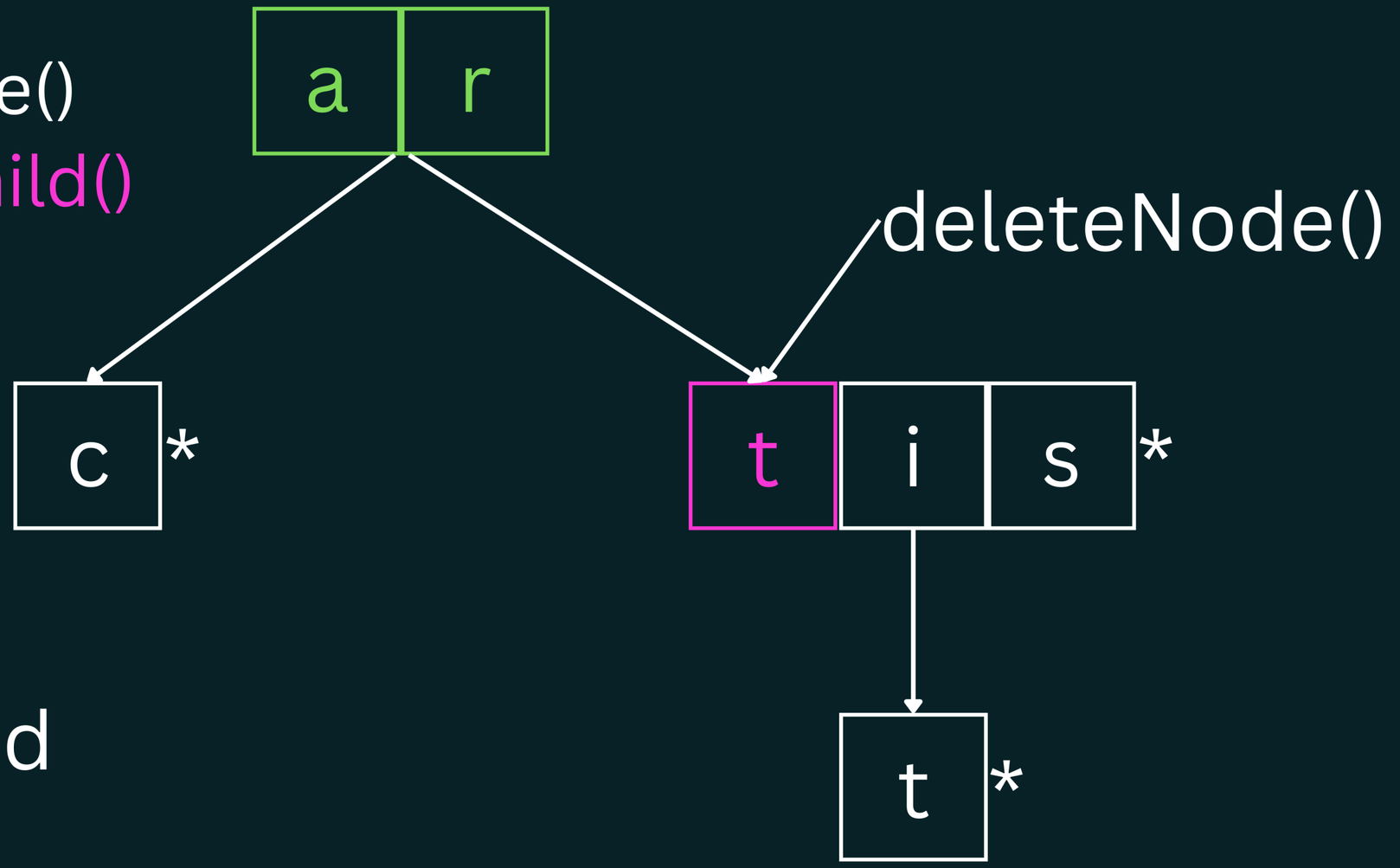
string to be deleted

Outcome Deletion

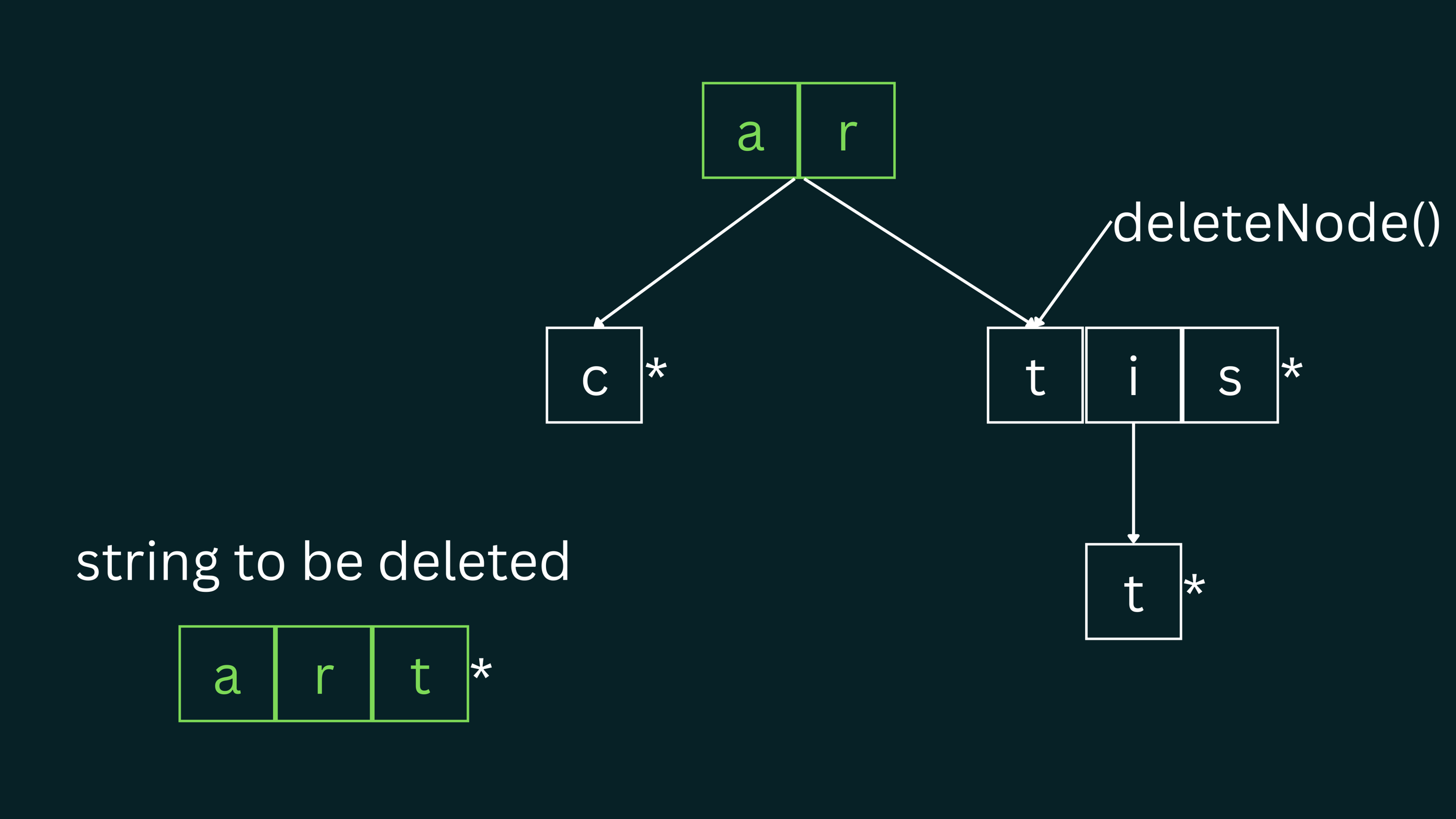
child = 0 => freeNode()

child = 1 => mergeChild()

child > 1 => isEnd = 0



string to be deleted

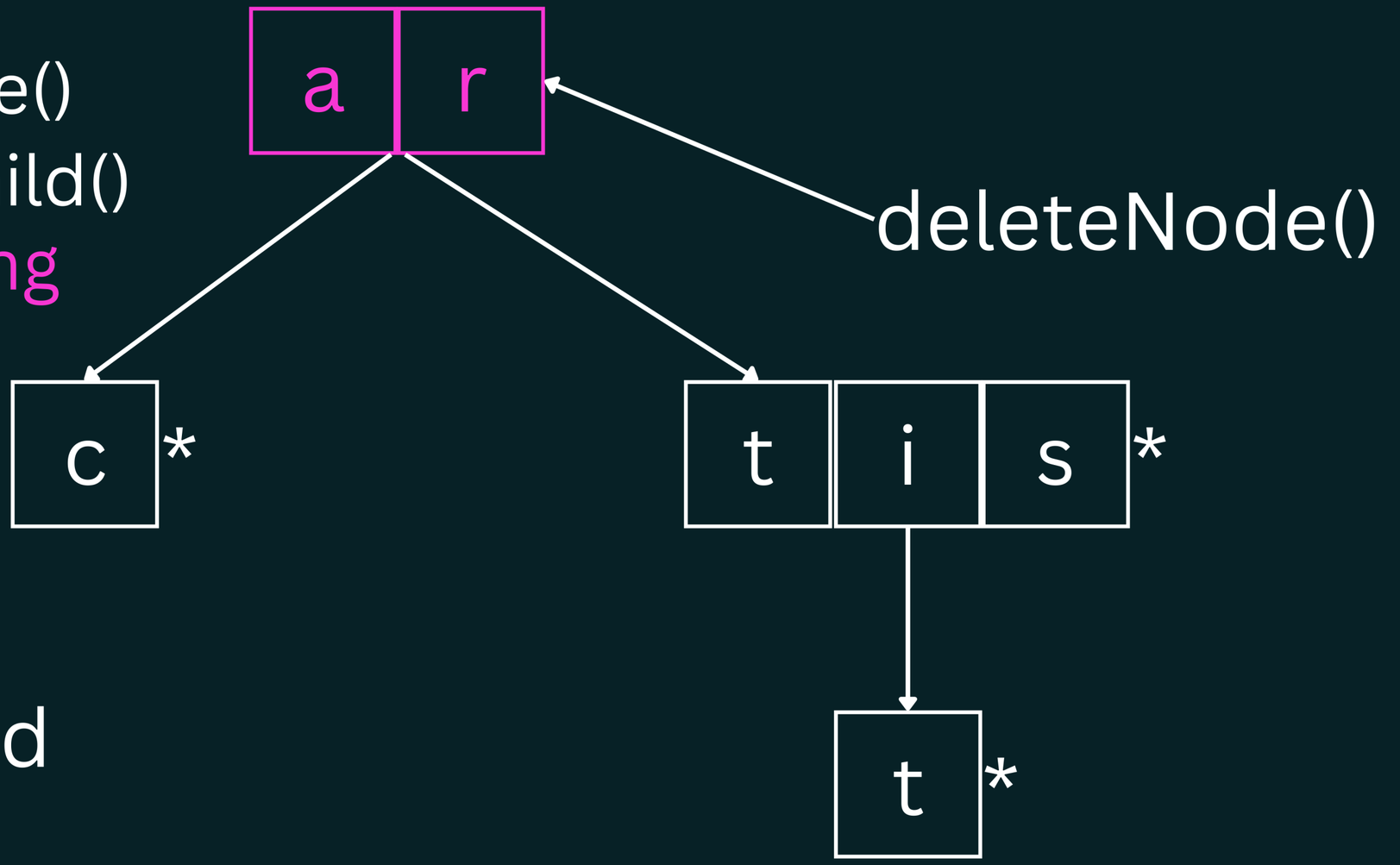

a
r
deleteNode()
c
*
t
i
s
*
t
*
string to be deleted
a
r
t
*

Outcome Deletion

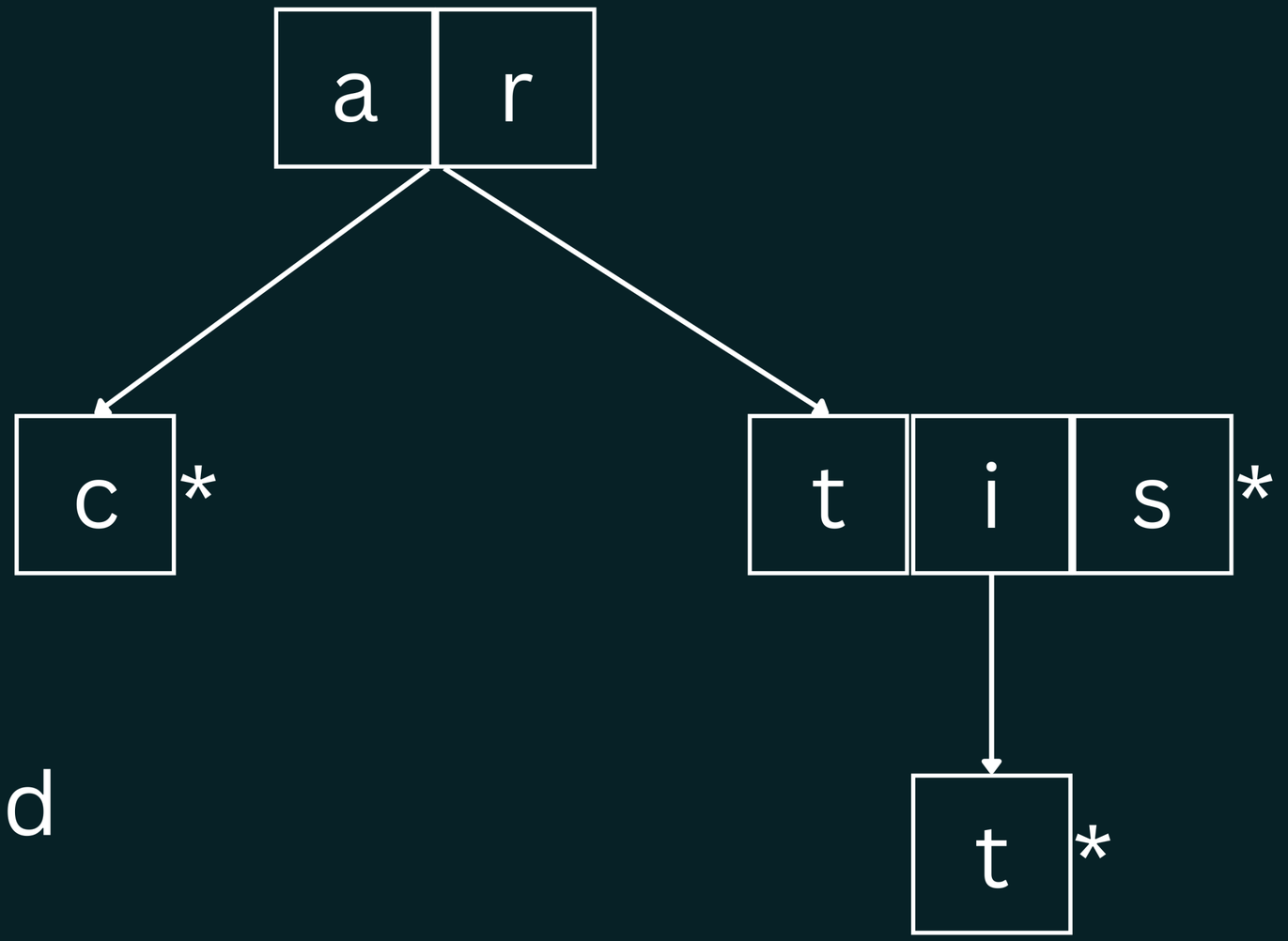
child = 0 => freeNode()

child = 1 => mergeChild()

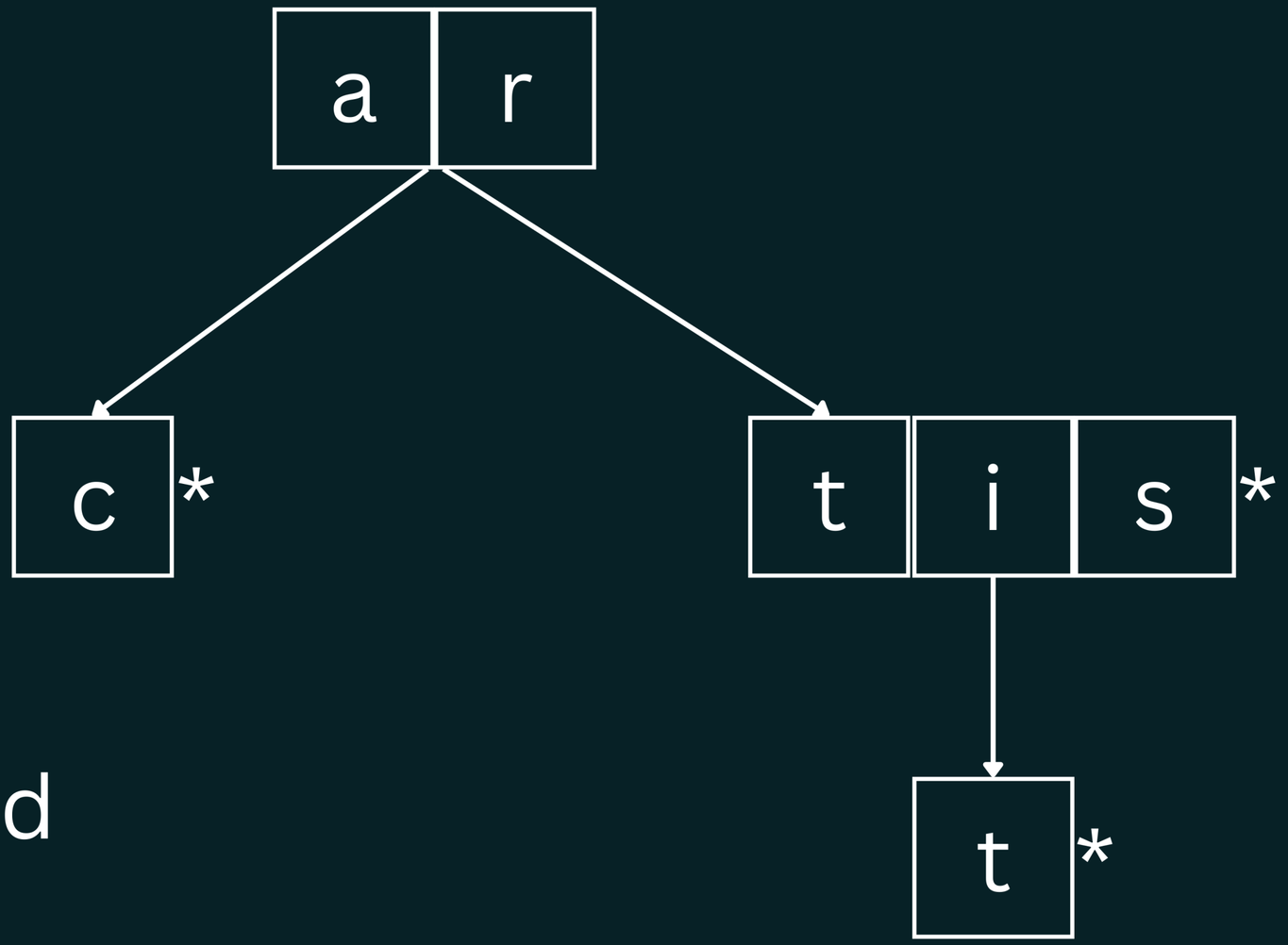
child > 1 => do nothing



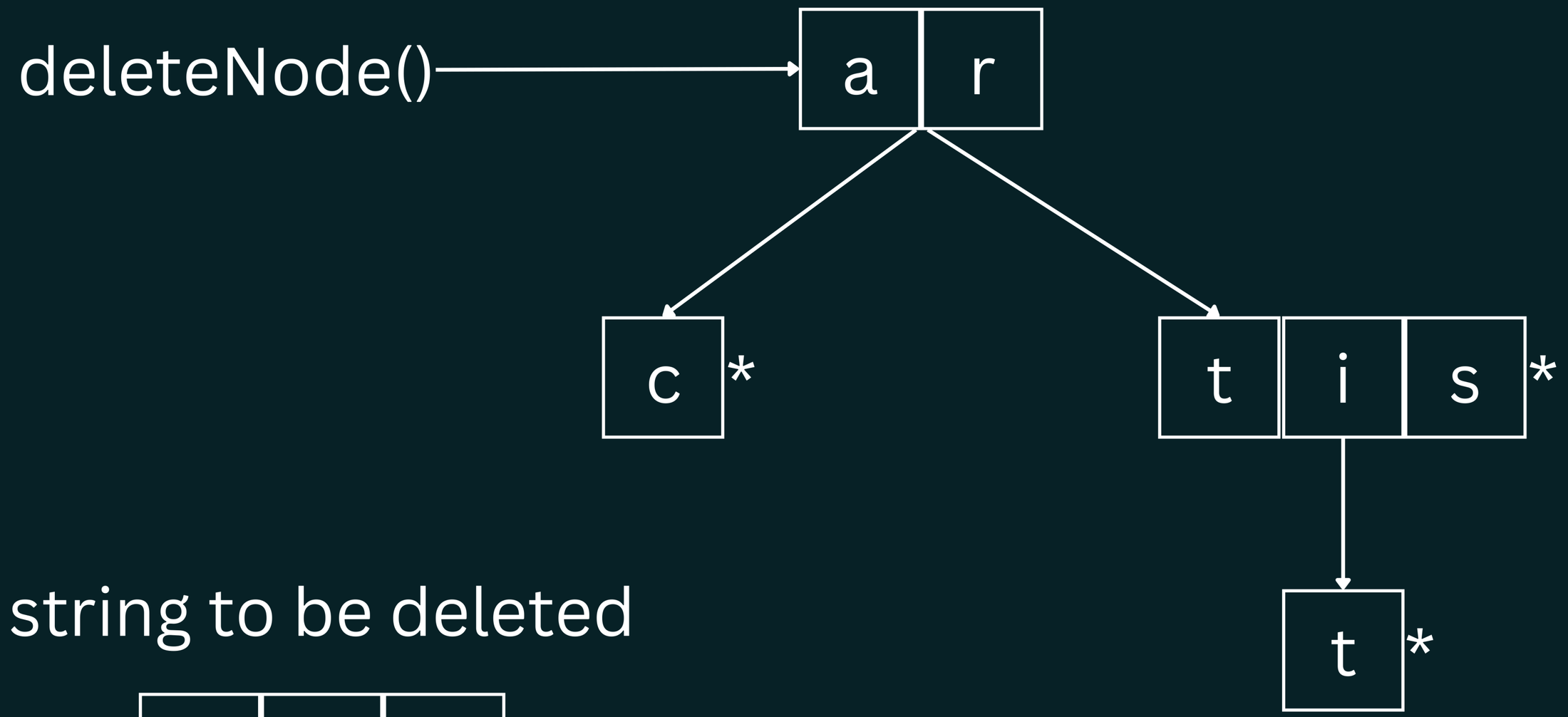
string to be deleted

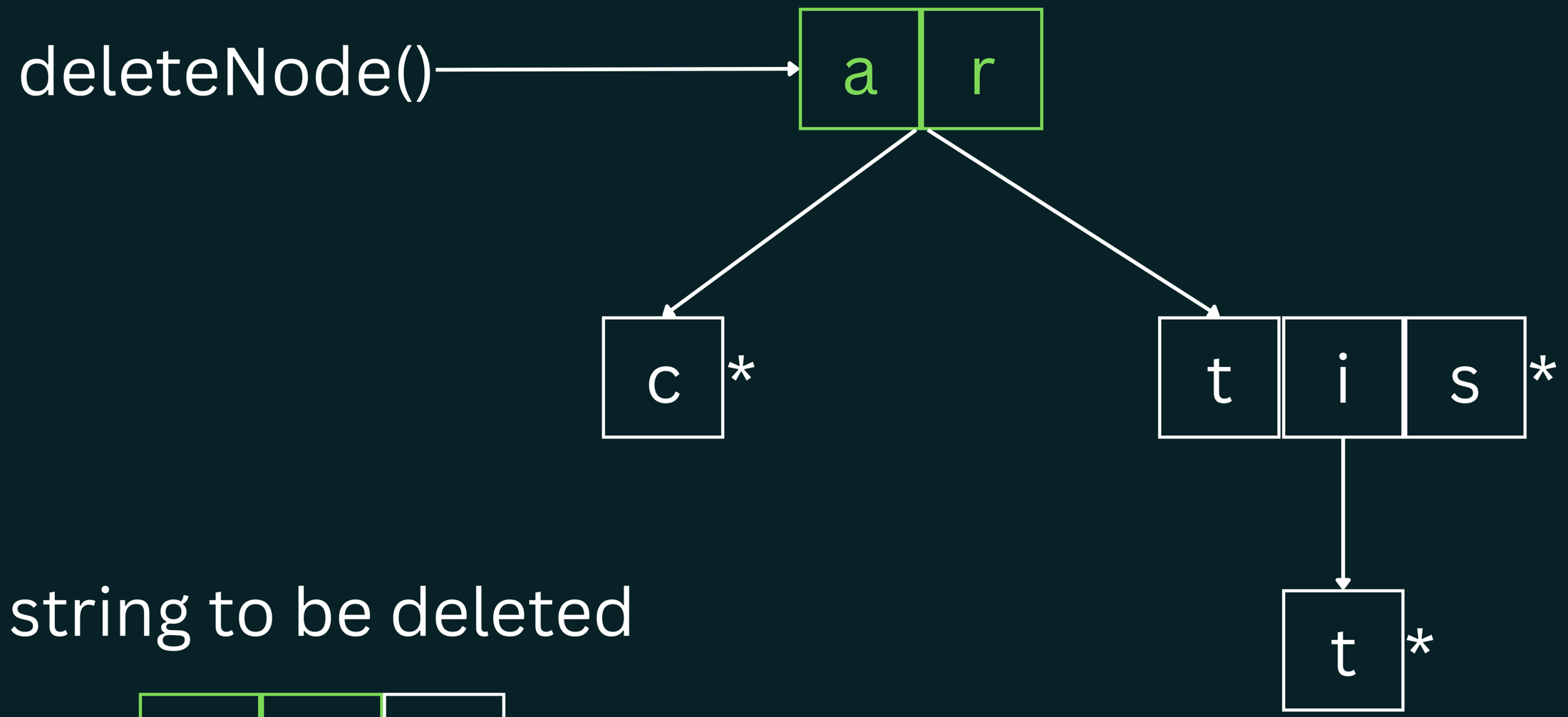


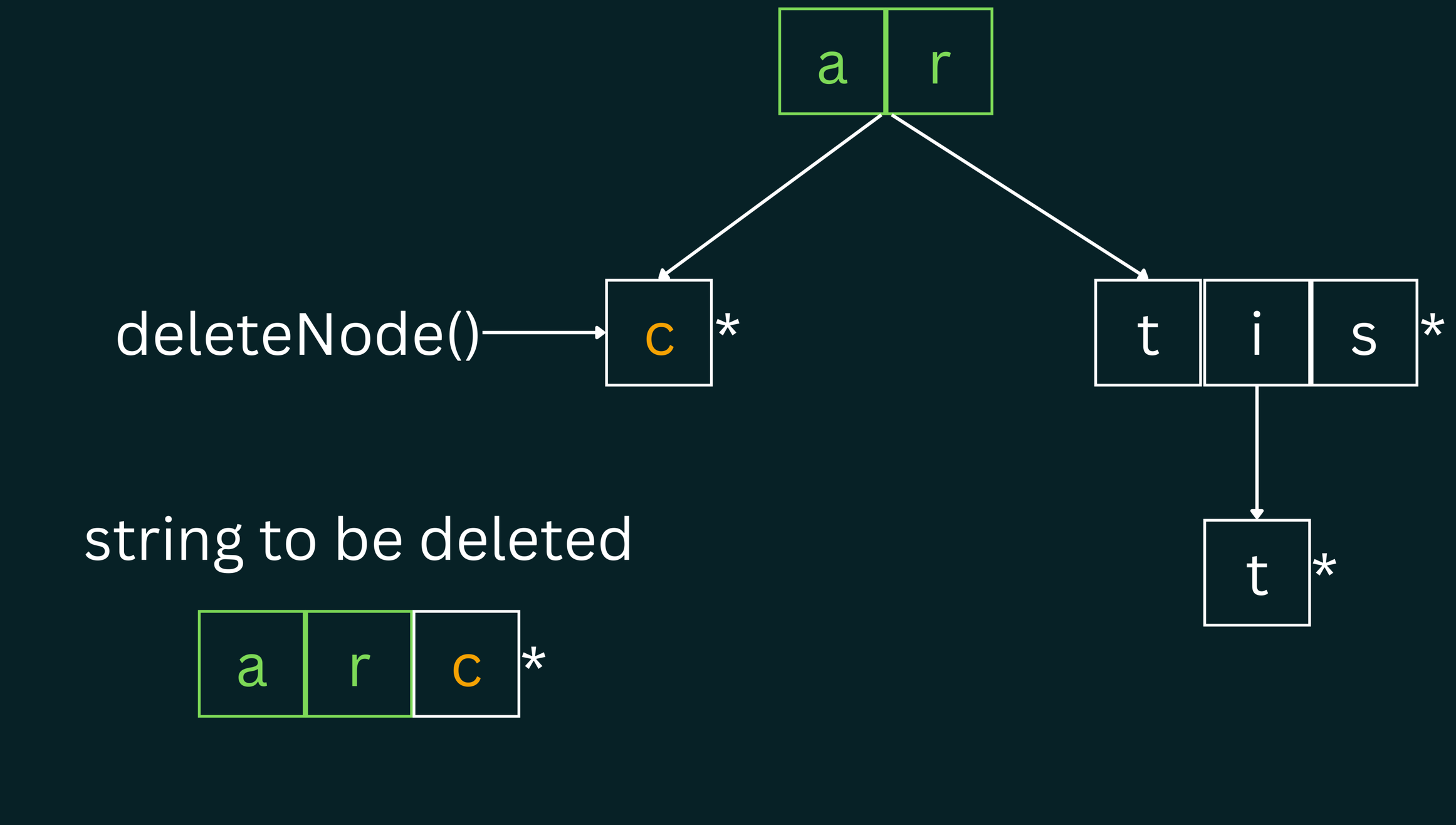
string to be deleted

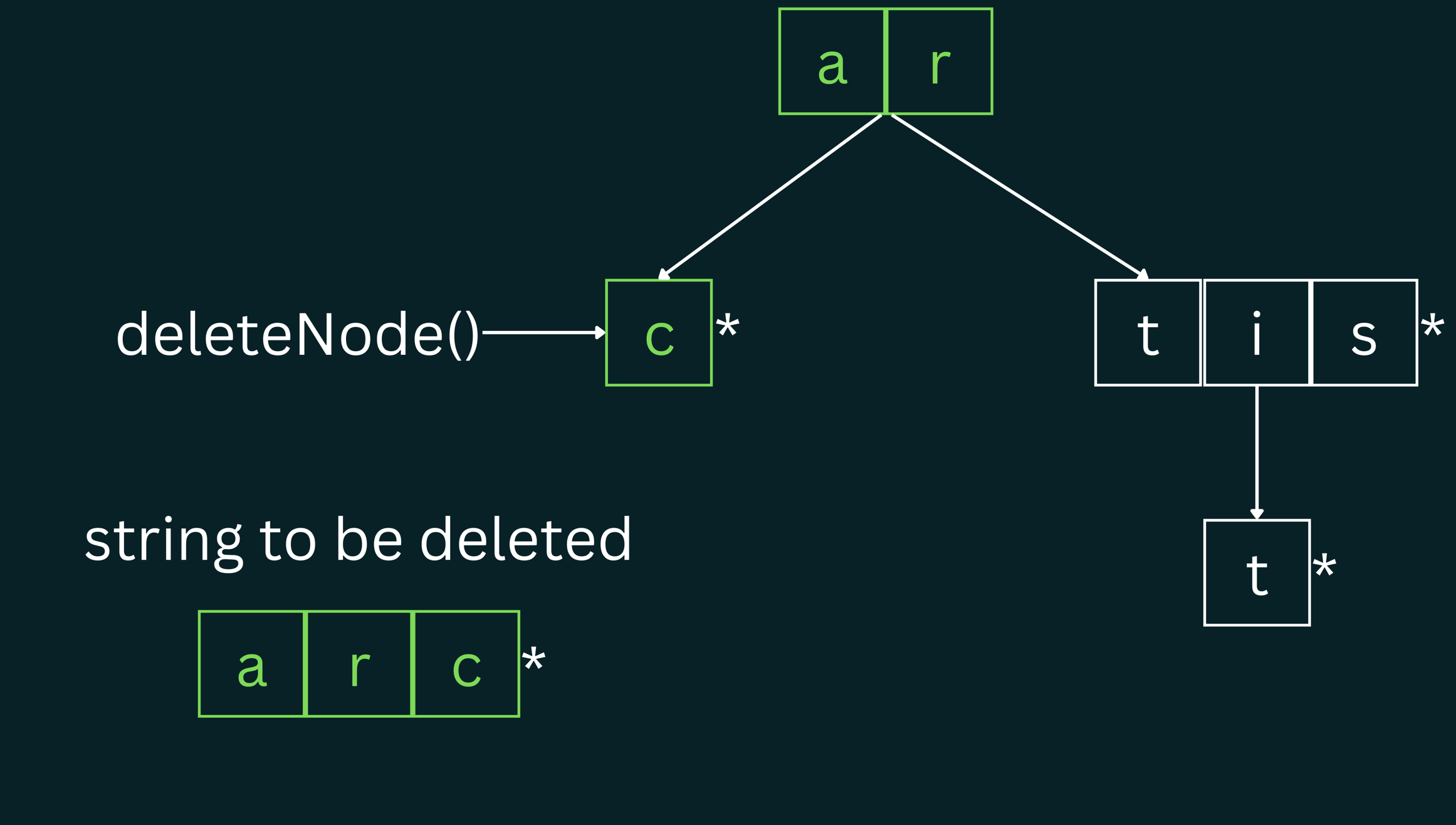


string to be deleted


deleteNode()
a
r
c
*
t
i
s
*
t
*
string to be deleted
a
r
c
*


deleteNode()
a
r
c
*
t
i
s
*
t
*
string to be deleted
a
r
c
*


a
r
deleteNode()
c
*
t
i
s
*
t
*
string to be deleted
a
r
c
*


a
r
deleteNode()
c
*
t
i
s
*
t
*
string to be deleted
a
r
c
*

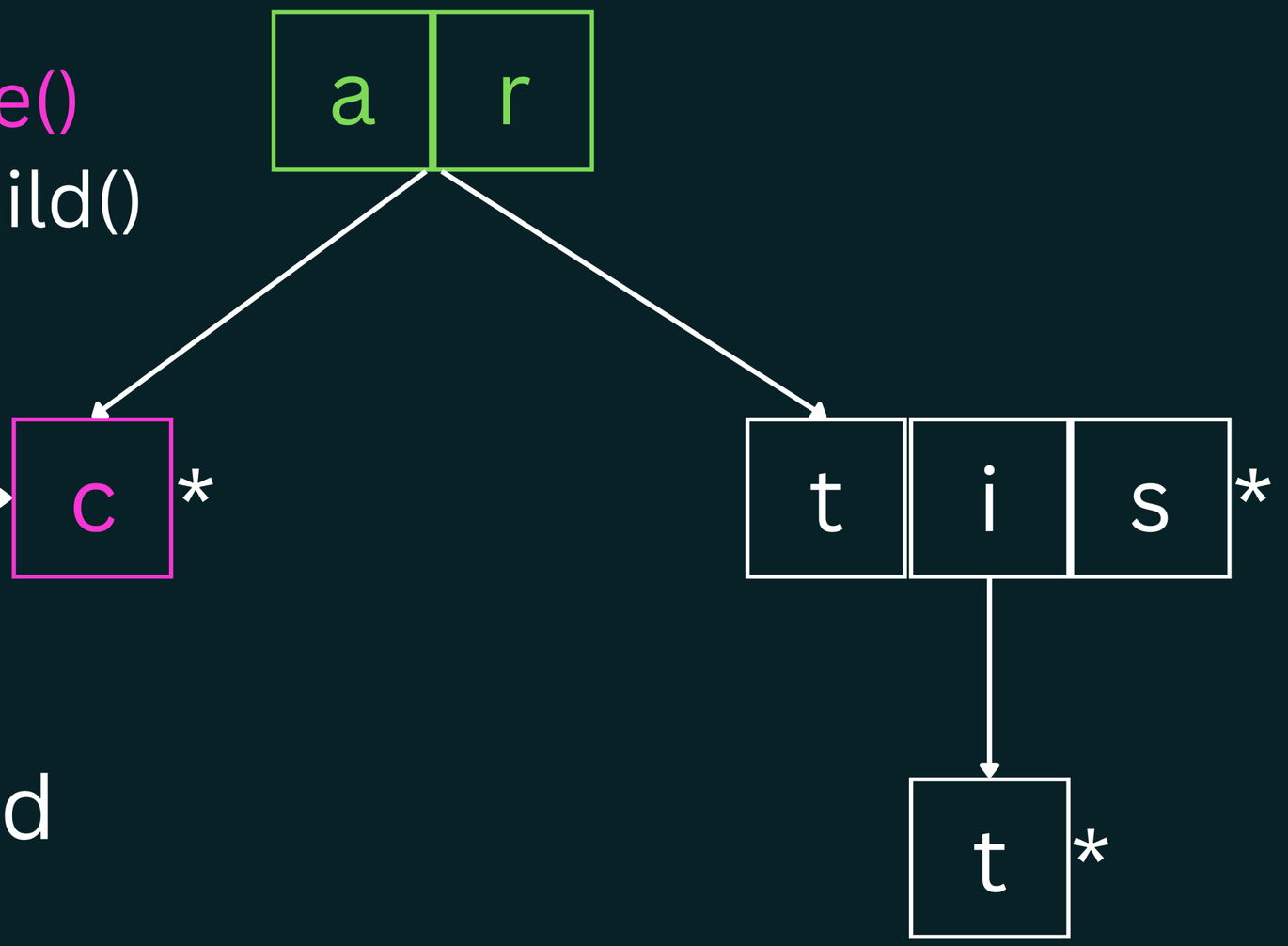
Outcome Deletion

child = 0 => freeNode()

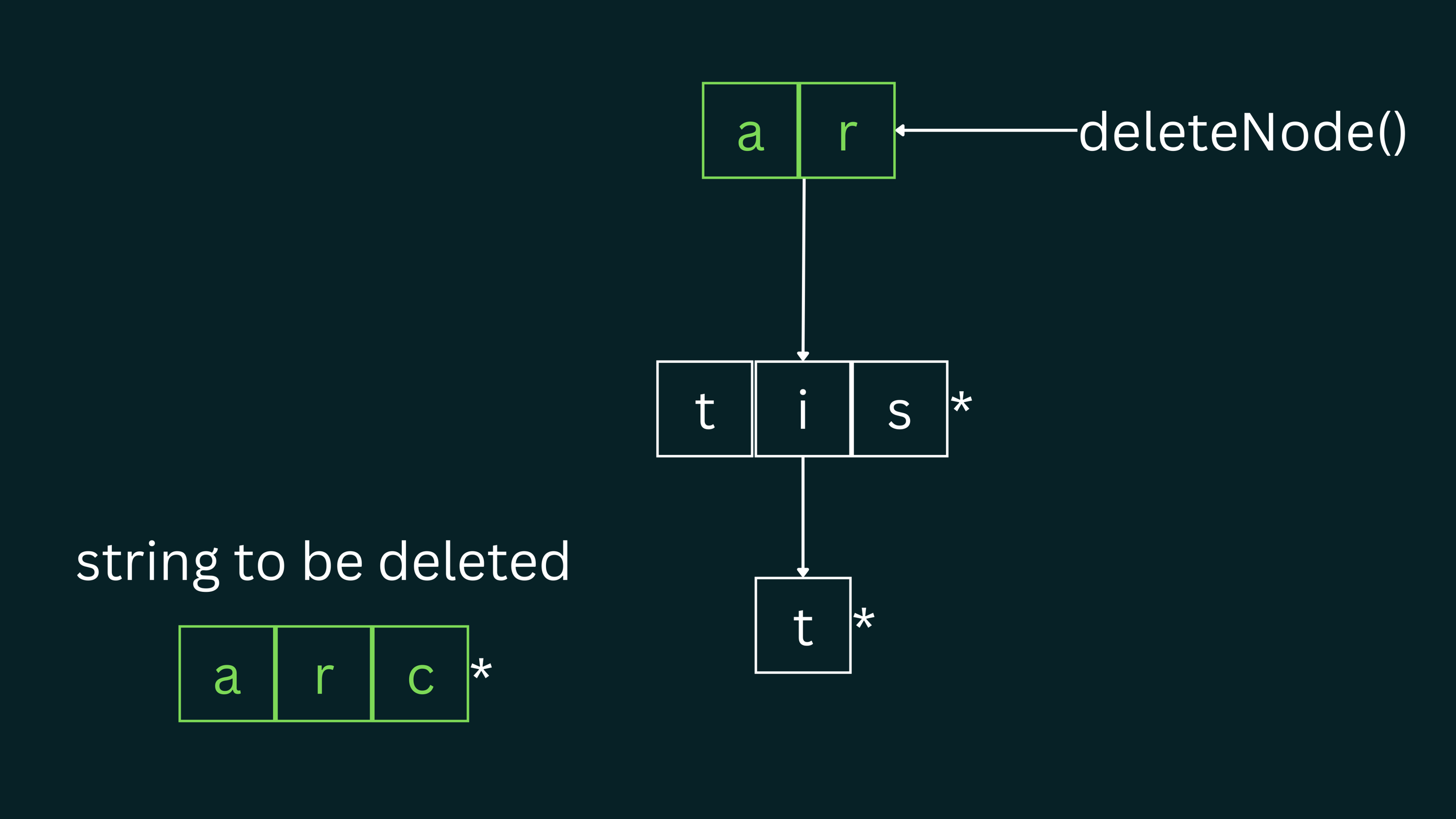
child = 1 => mergeChild()

child > 1 => isEnd = 0

deleteNode()



string to be deleted

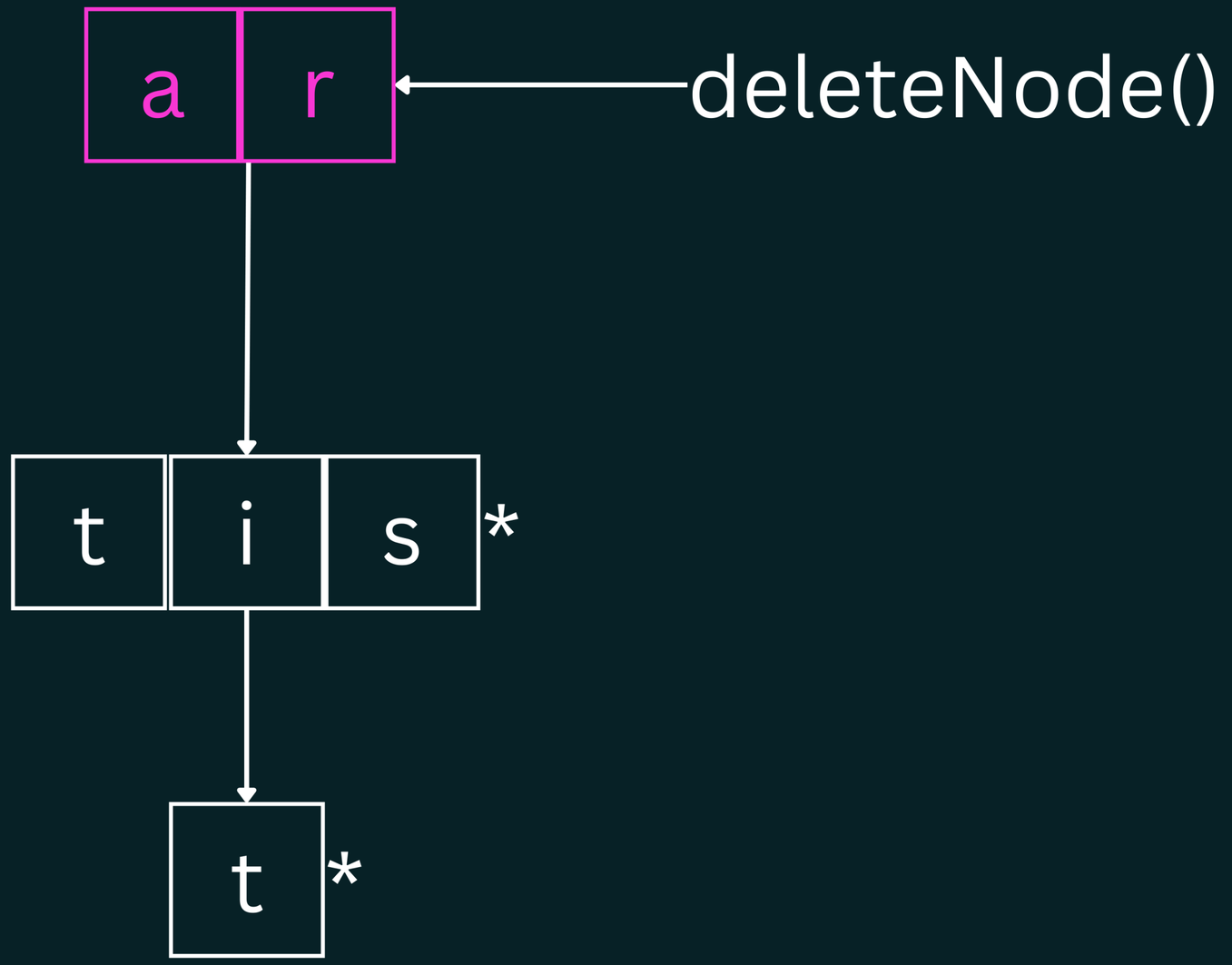

a
r
deleteNode()
t
i
s
*
t
*
string to be deleted
a
r
c
*

Outcome Deletion

child = 0 => freeNode()

child = 1 => mergeChild()

child > 1 => do nothing



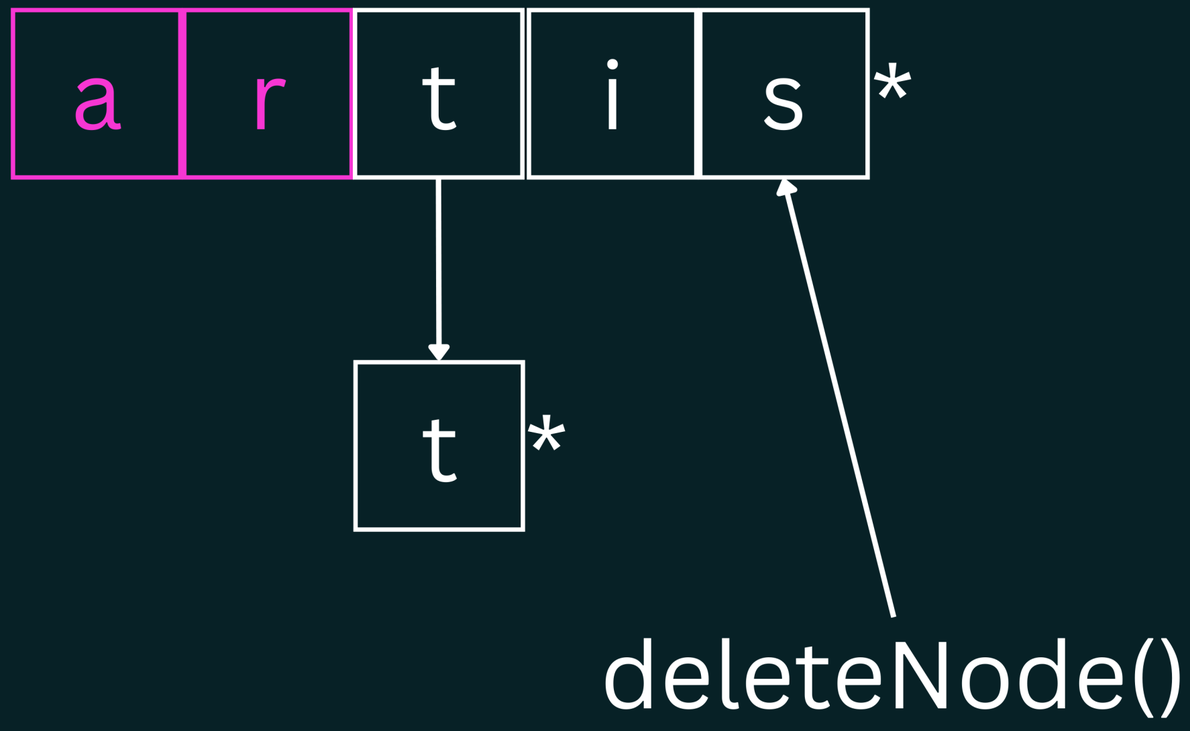
string to be deleted

Outcome Deletion

child = 0 => freeNode()

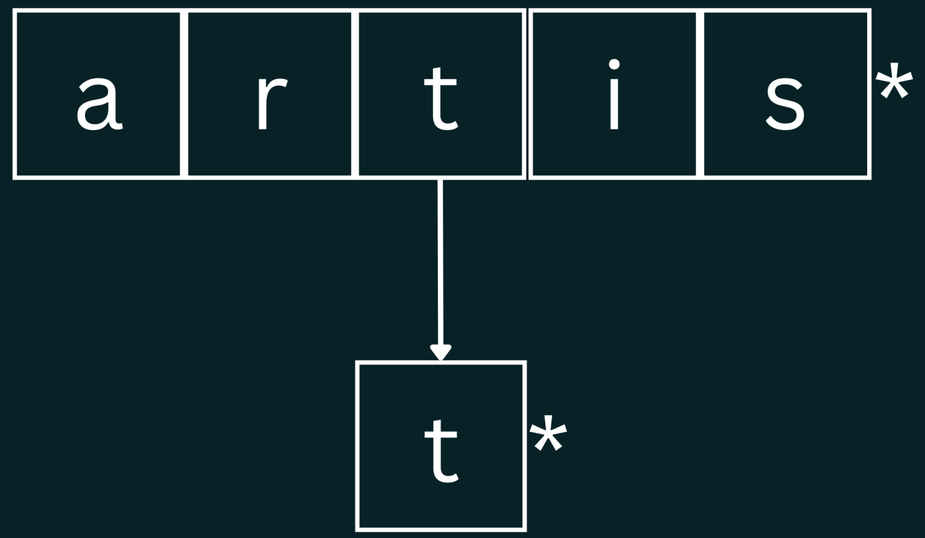
child = 1 => mergeChild()

child > 1 => do nothing



string to be deleted

new trie



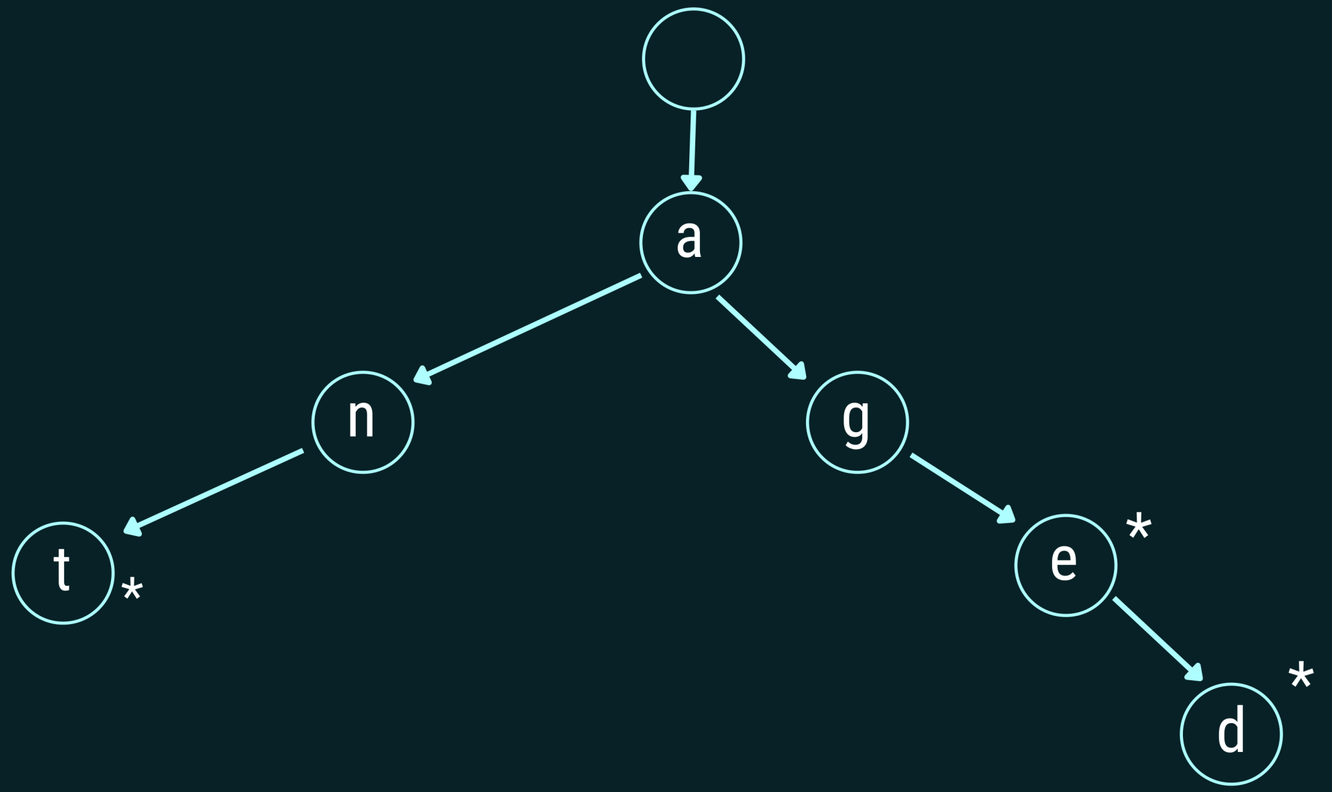
Starts With Function

GET WORDS FUNCTION

```
void startsWith(triePtr root, char string[]){
   printf("\n\nSearching for words starting in '%s'...", string);
   int curr;
   for(curr = 0; string[curr] != '\0'; root = root->children[string[curr] - 'a'], curr++);
   printWords(root, string);
}//INTERNET CODE

void printWords(triePtr root, char string[]){
   if(root->isEndofWord == 1){
      doSomething(string);
   }

   int x, y;
   triePtr next;
   char newString[32];
   for(x = 0; x < CHILDREN_SIZE; x++){
      next = root->children[x];
      if(next != NULL){
         memset(newString, '\0', sizeof(newString));
         strcpy(newString, string);
         for(y = 0; newString[y] != '\0'; y++);
         newString[y] = x + 'a';
         printWords(next, newString);
      }
   }
}
```

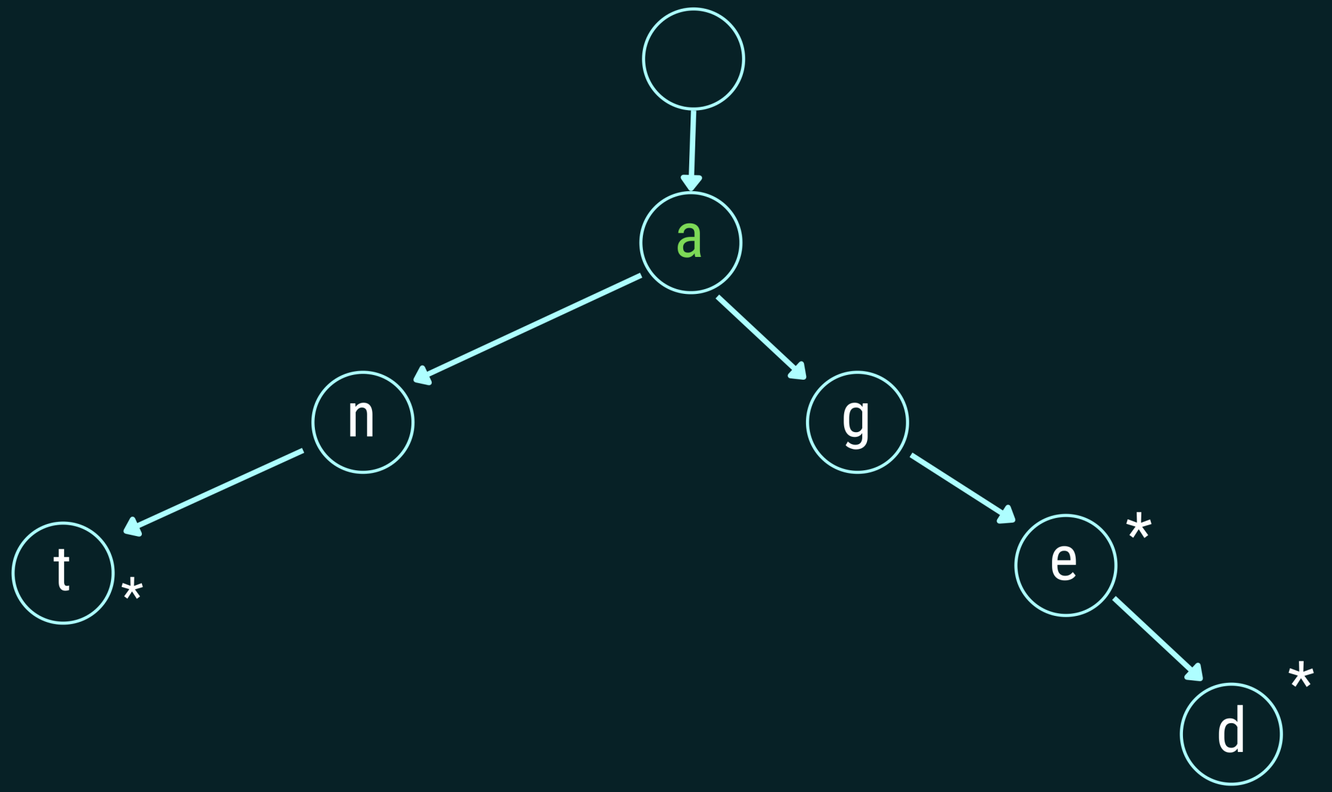
get words starting with...

a



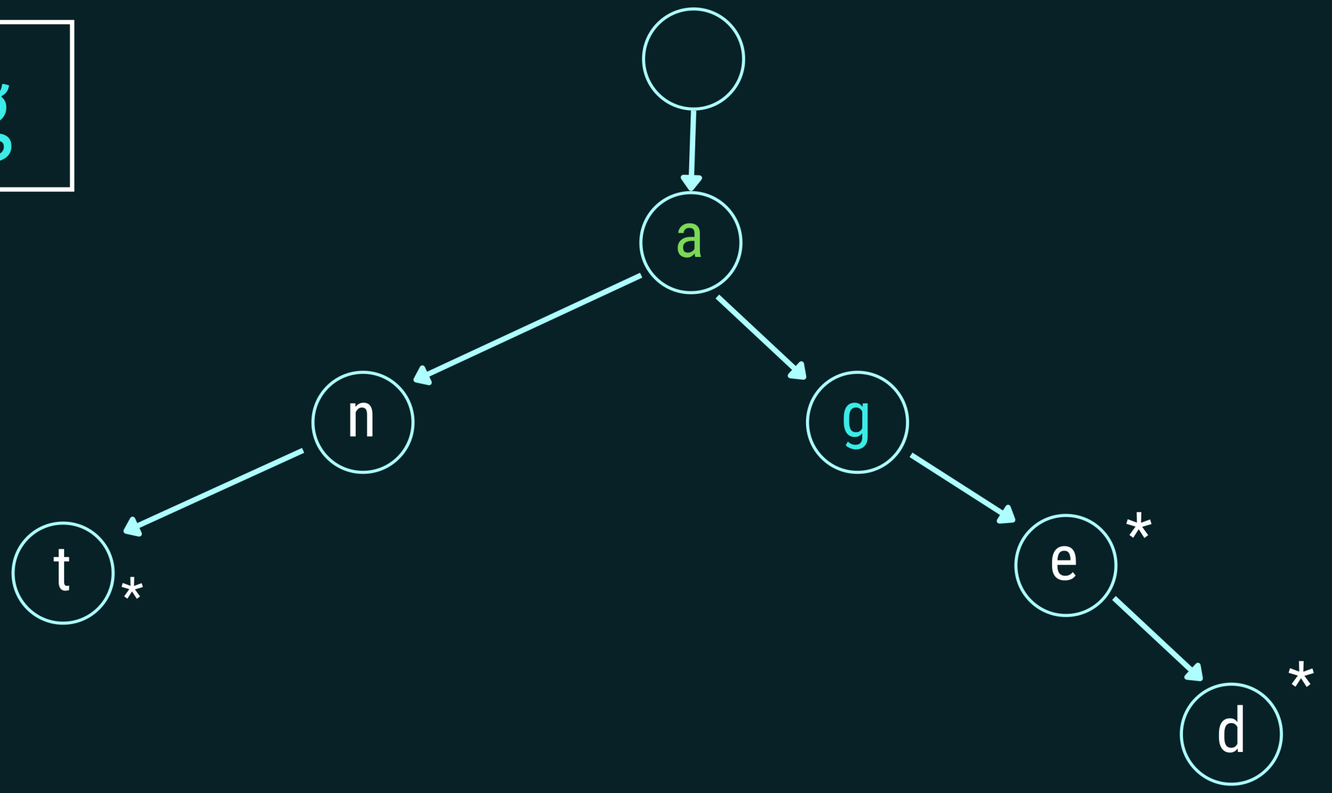
get words starting with...

a

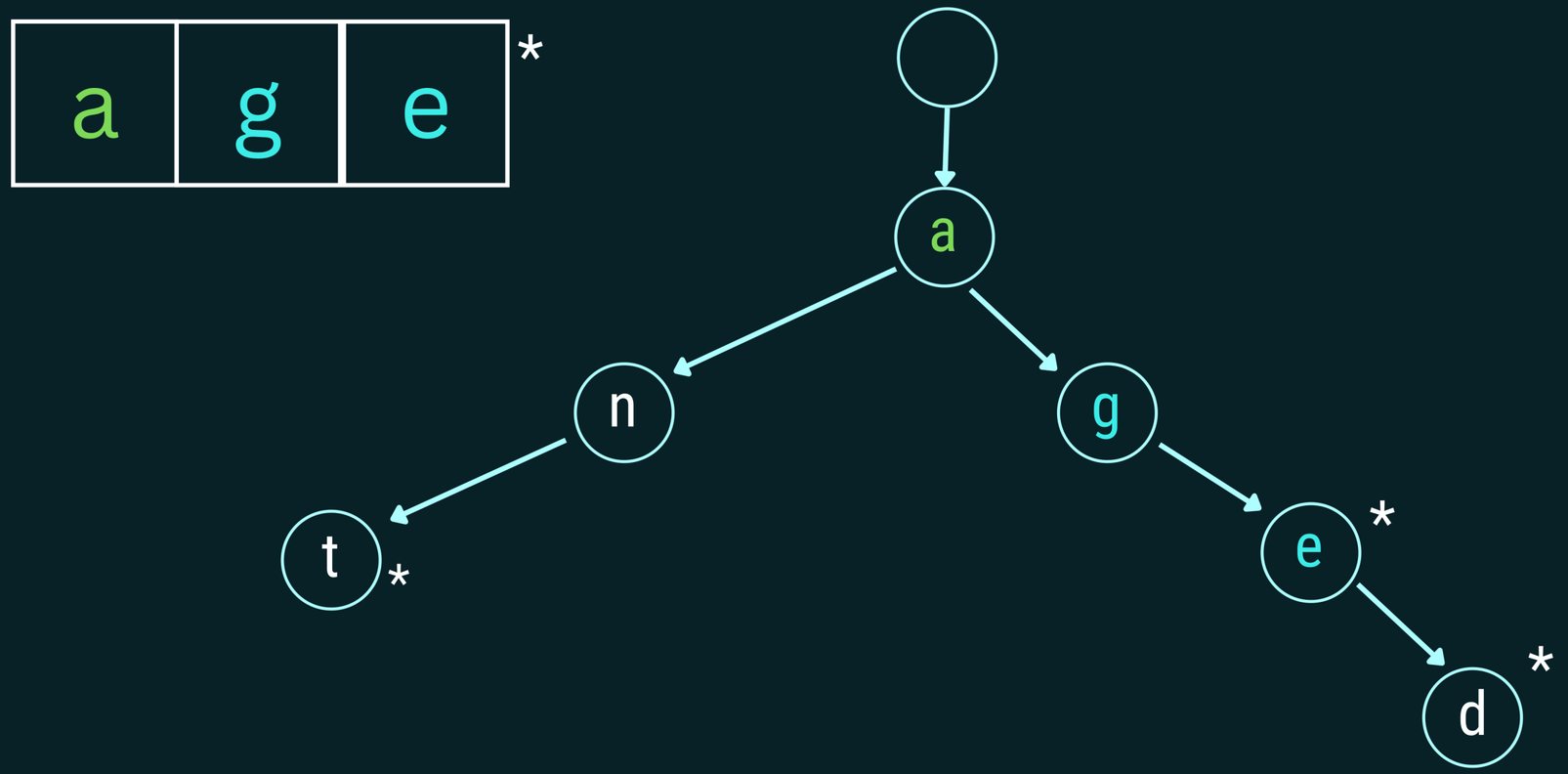


get words starting with...

a	g



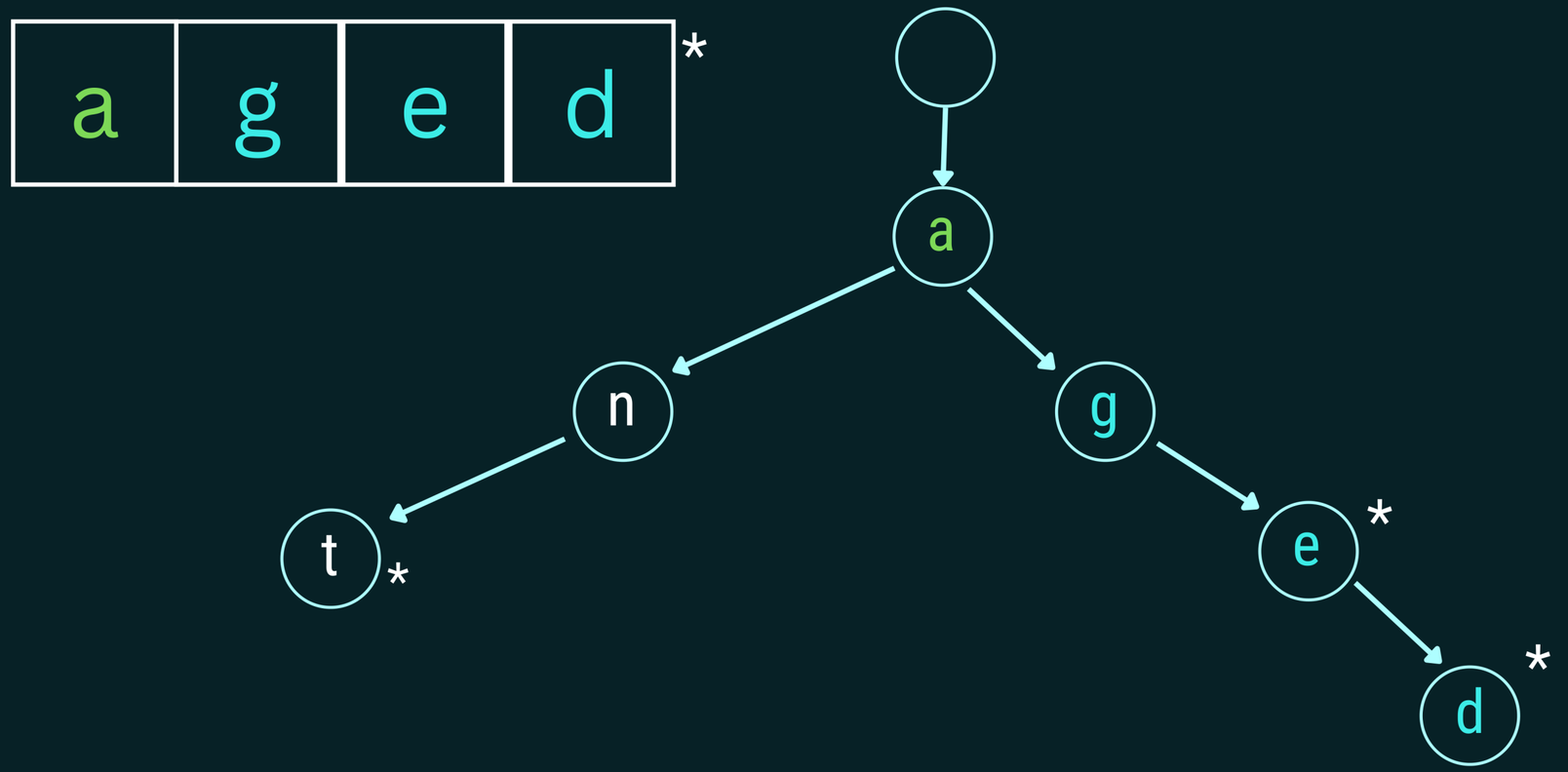
get words starting with...



get words starting with...

get word

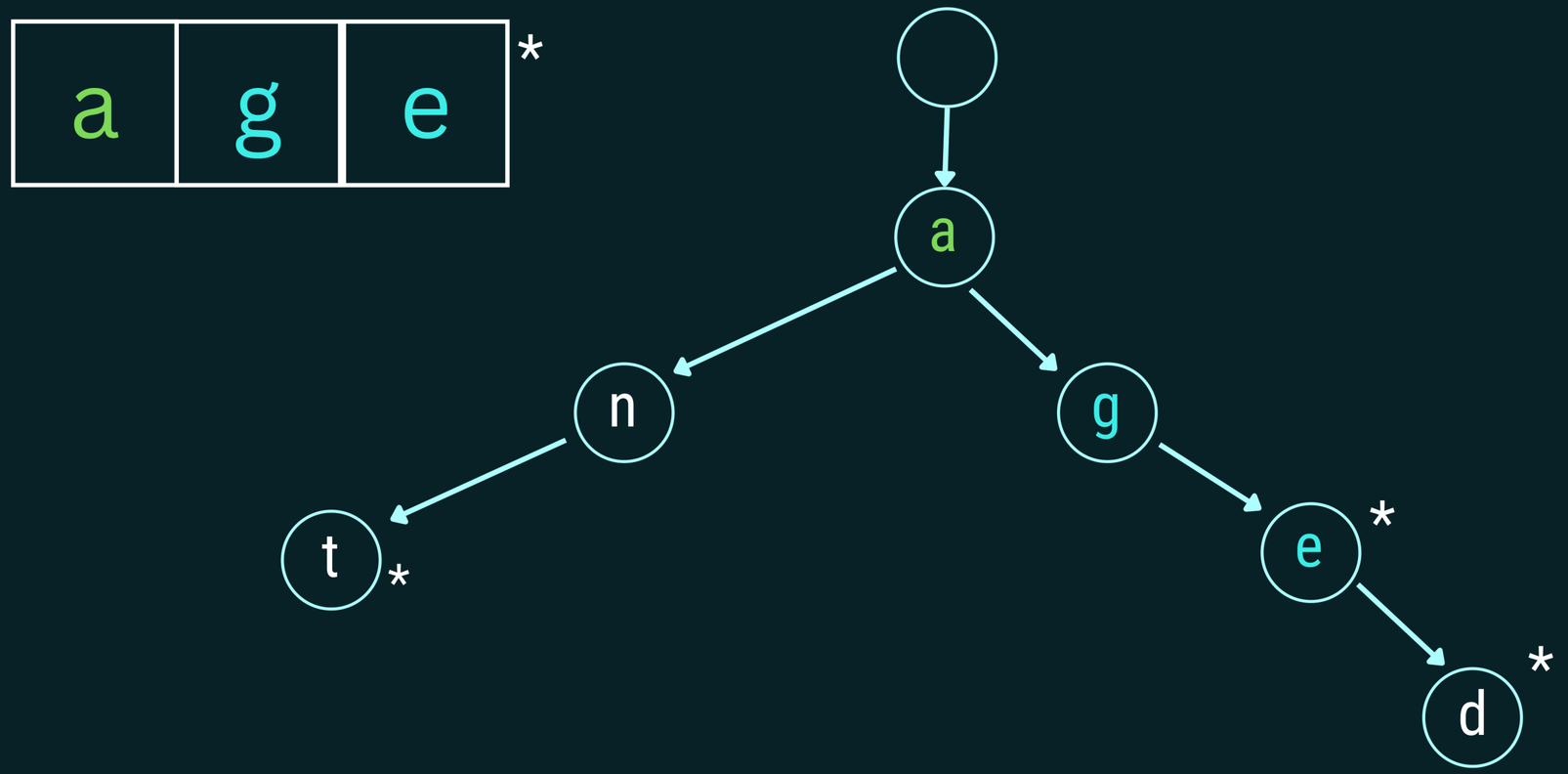
get words starting with...



get words starting with...

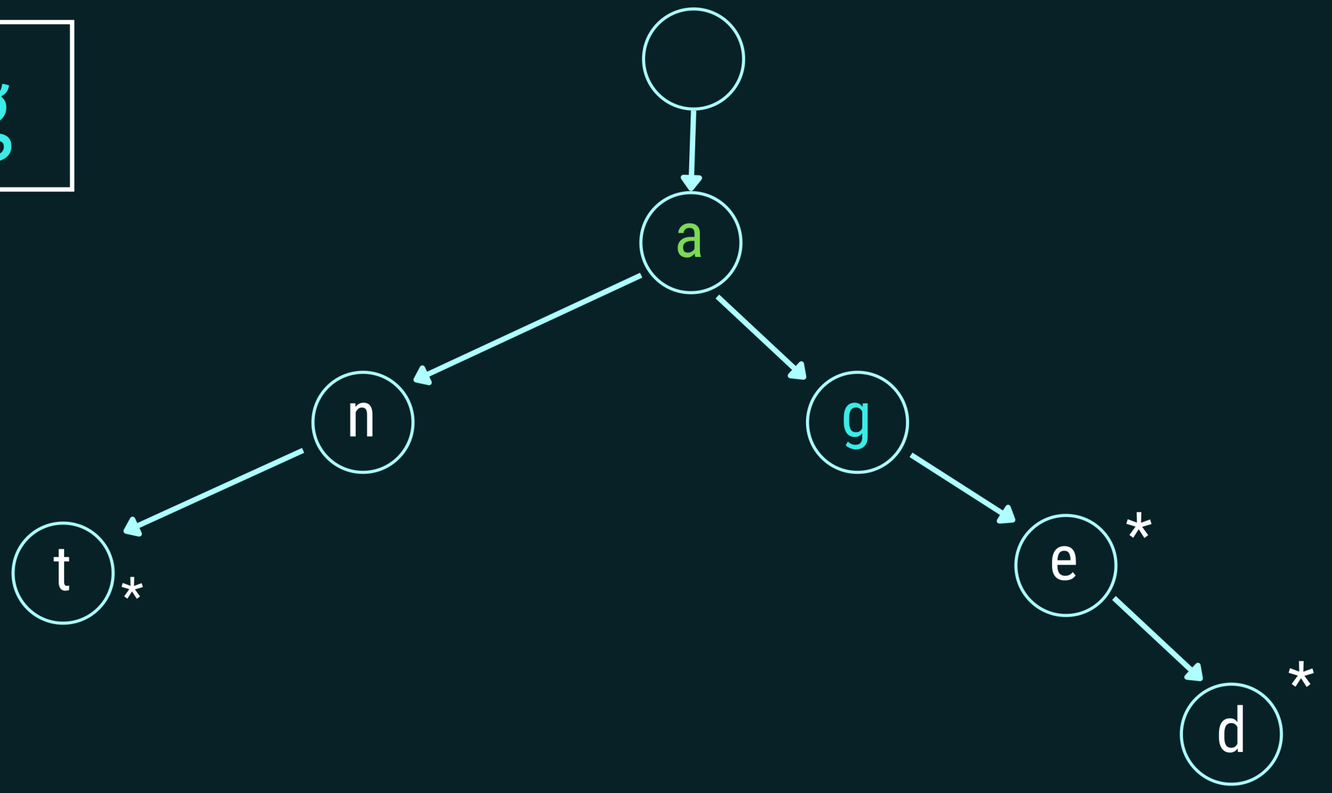
get word

get words starting with...



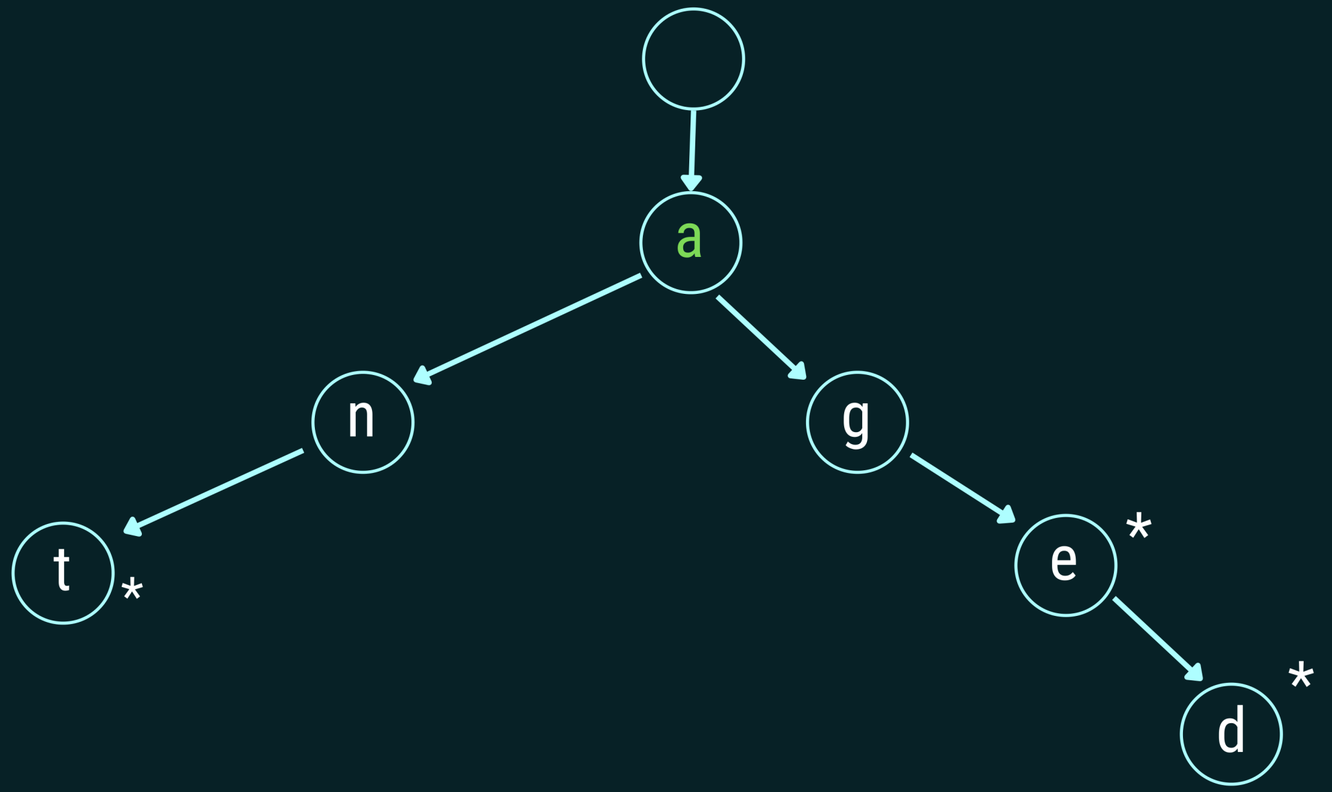
get words starting with...

a	g



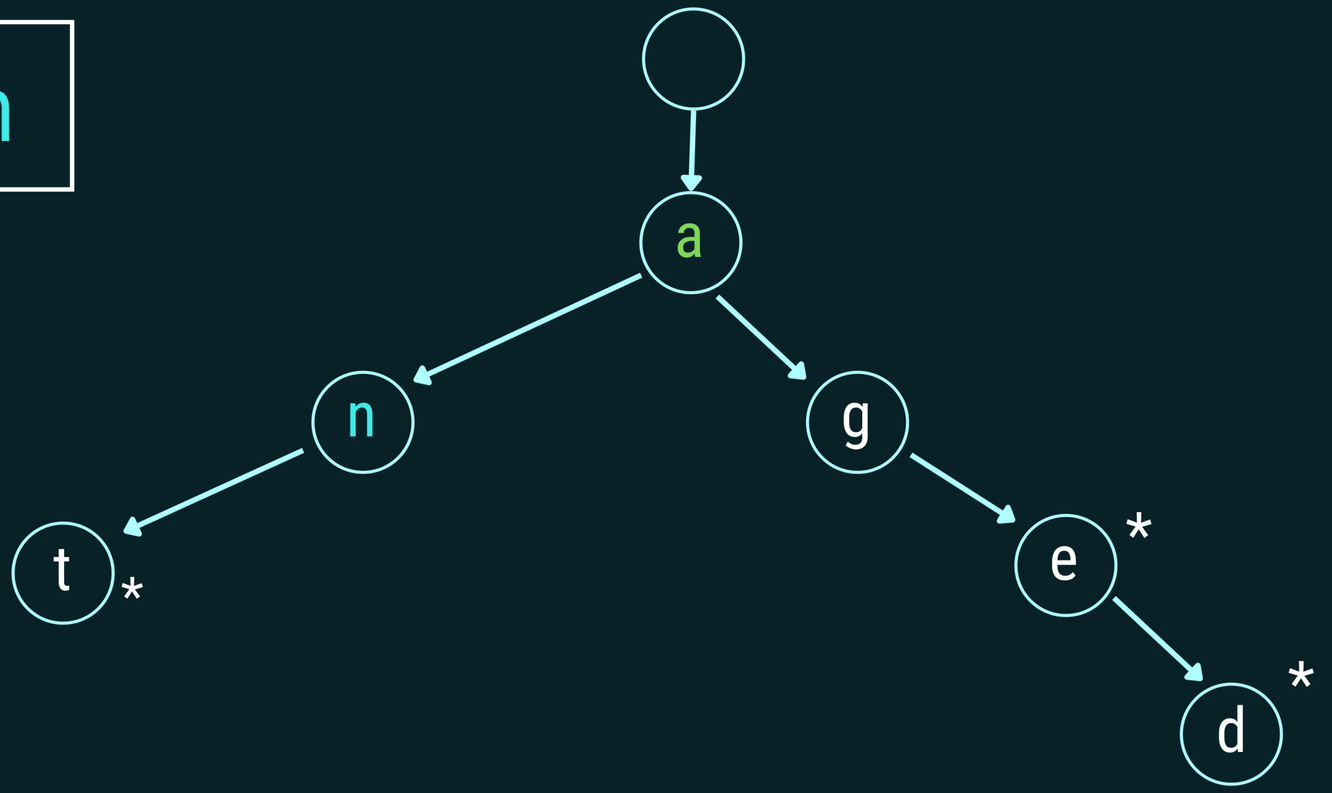
get words starting with...

a

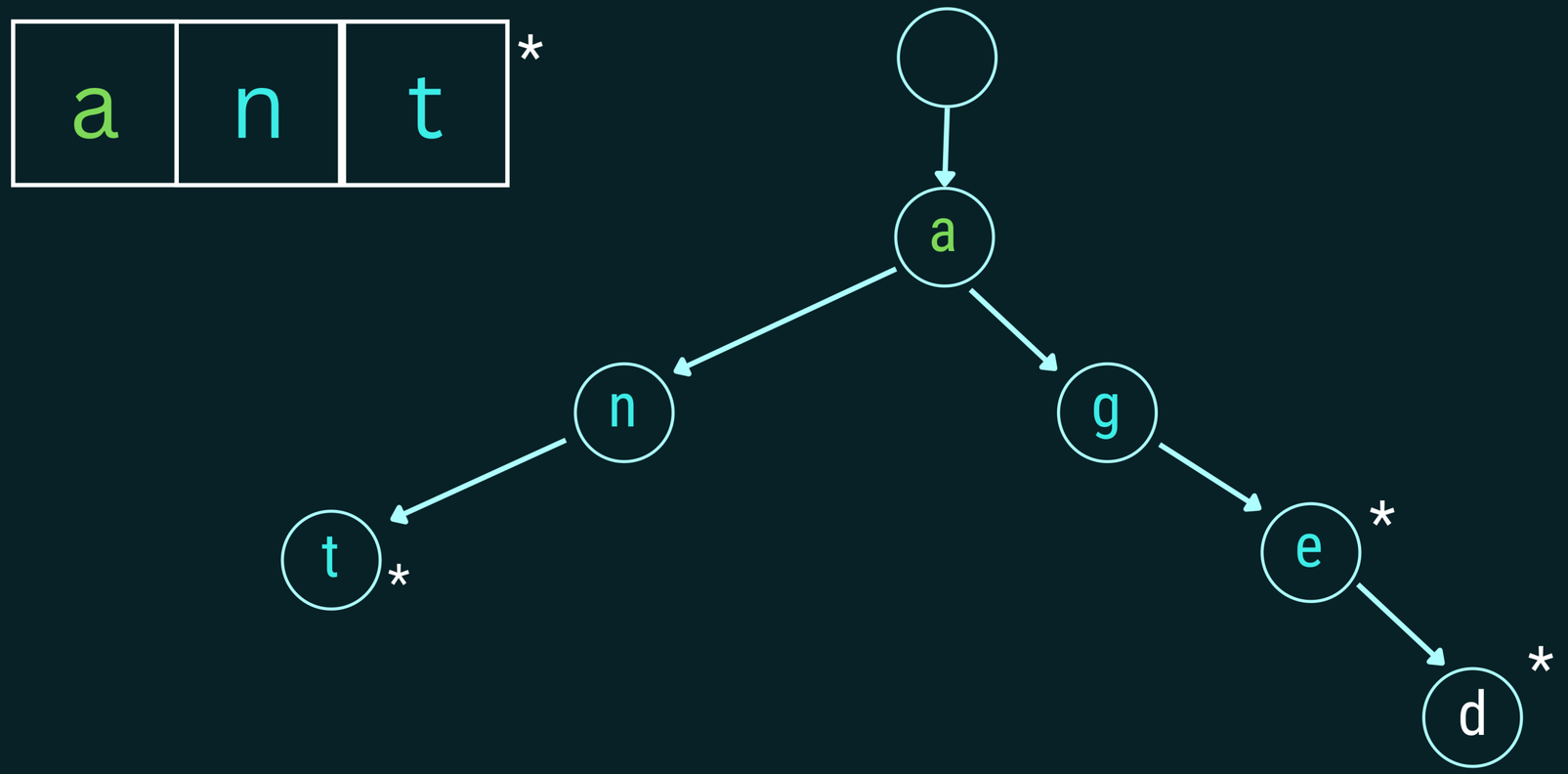


get words starting with...

a	n



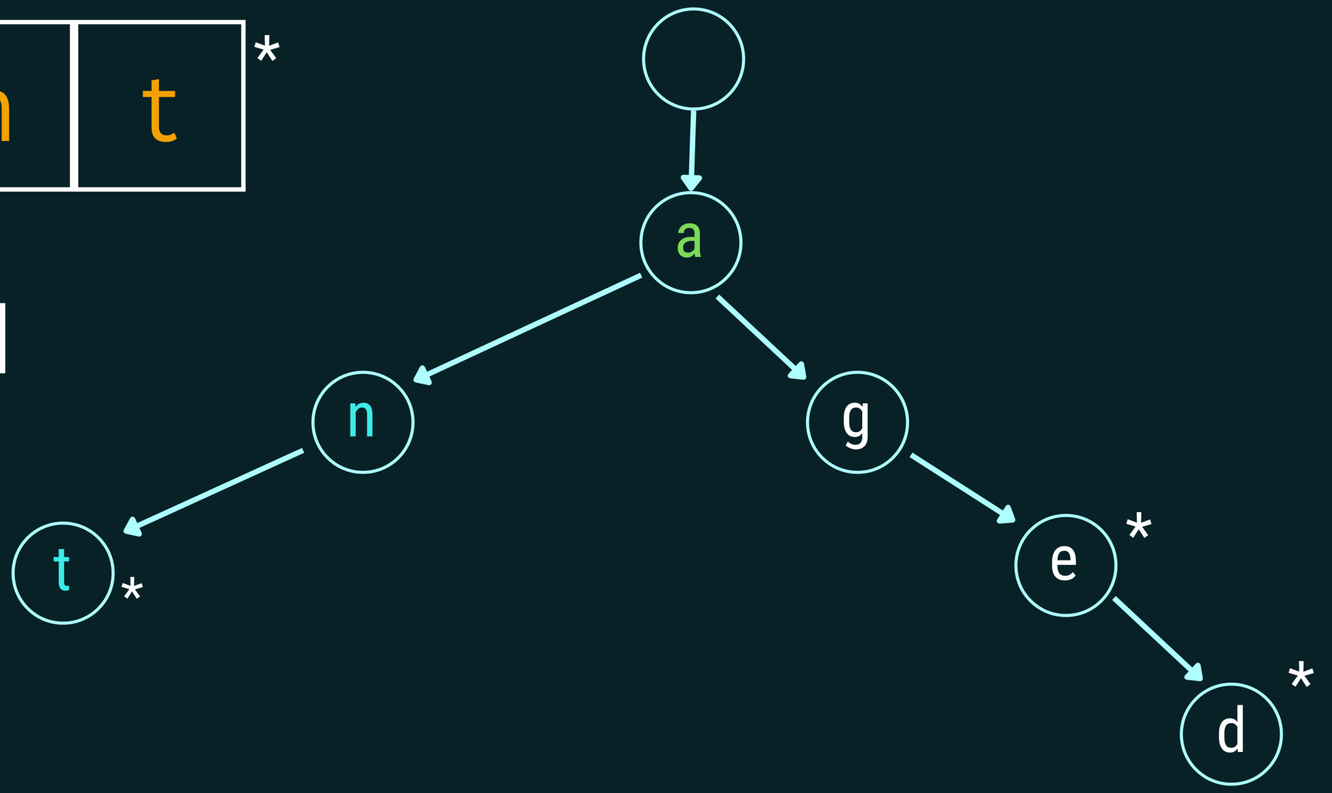
get words starting with...



get words starting with...

a	n	t	*

get word

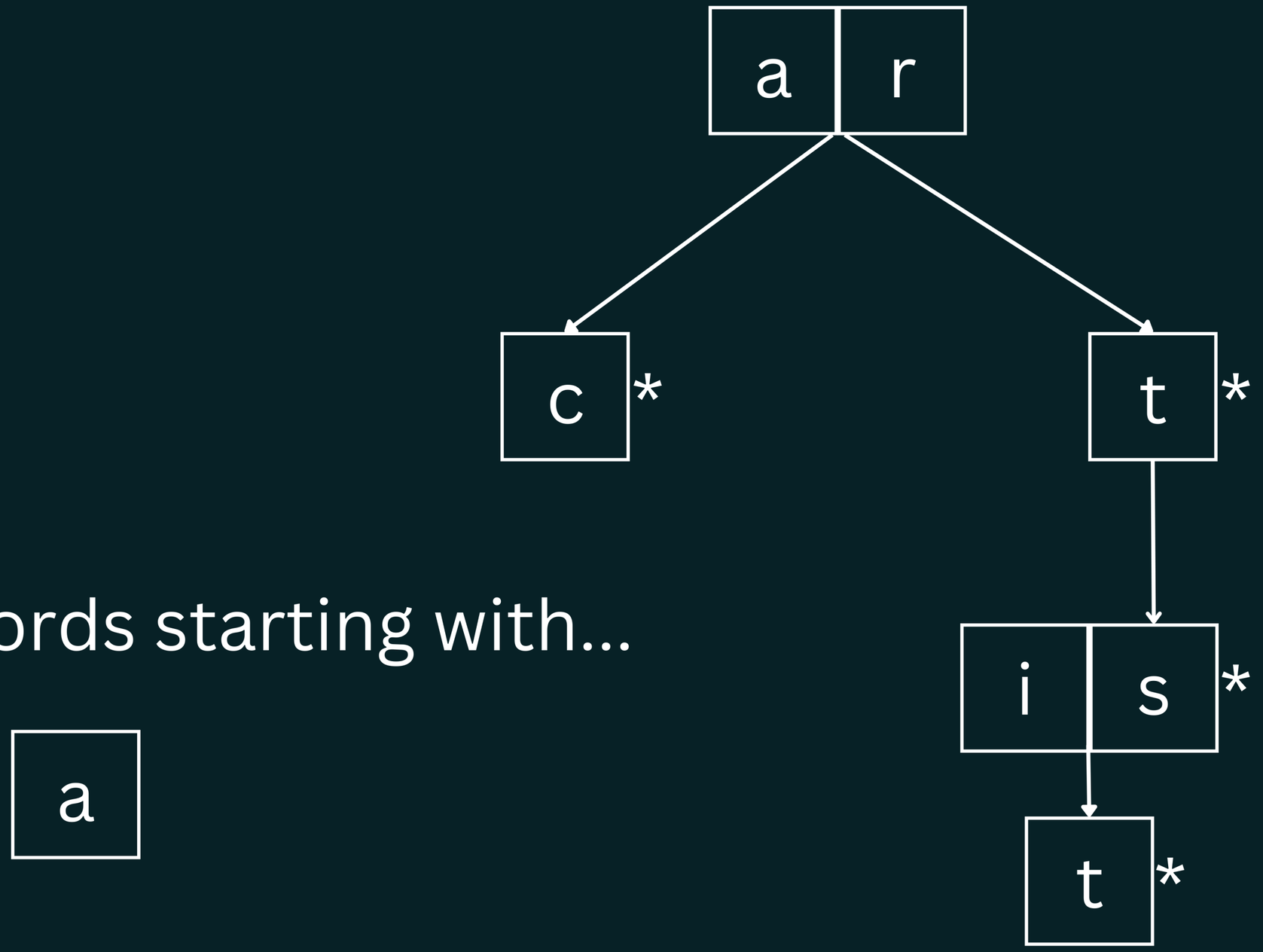


GET WORDS FUNCTION

```
void getWordsStarting(triePtr root[], String s){
   triePtr trav = root[s[0] - 'a'];
   int x = 0, y = 0, check = 1;
   while(check == 1 && s[x] != '\0'){
      x++;
      y++;
      if(trav->data[y] == '\0' && trav->next[s[x] - 'a'] != NULL && s[x] != '\0'){
         y = 0;
         trav = trav->next[s[x] - 'a'];
      }
      check = s[x] == trav->data[y] ? 1 : 0;
   }
   if(check == 1){
      String next;
      memset(next, '\0', sizeof(String));
      strcpy(next, s);
      int nextCtr;
      for(nextCtr = strlen(next); trav->data[y] != '\0'; nextCtr++, y++)
         next[nextCtr] = trav->data[y];
      for(nextCtr = 0; nextCtr < ALPHABET_SIZE; nextCtr++)
         if(trav->next[nextCtr] != NULL)
            getWord(trav->next[nextCtr], next);
  }else{
     printf("\nThe prefix you specified is not inside the trie.");
  }
}
```

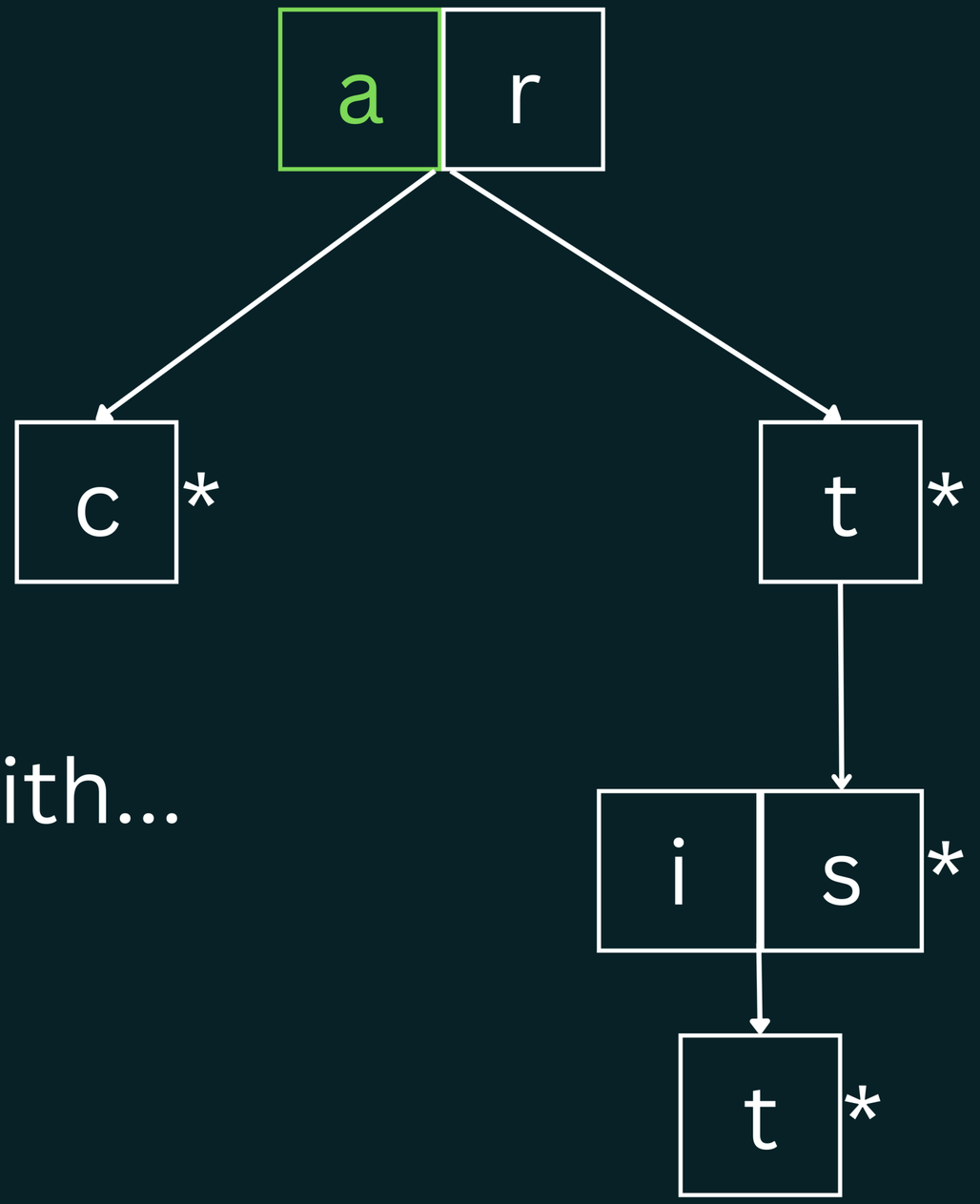
GET WORDS FUNCTION

```
void getWord(triePtr head, String s){
   if(head->isEnd > 0)
      doSomething(ret);

   String ret;
   memset(ret, '\0', sizeof(String));

   strcat(ret, s);
   strcat(ret, head->data);

   int x;
   for(x = 0; x < ALPHABET_SIZE; x++)
      if(head->next[x] != NULL)
         getWord(head->next[x], ret);
}
```



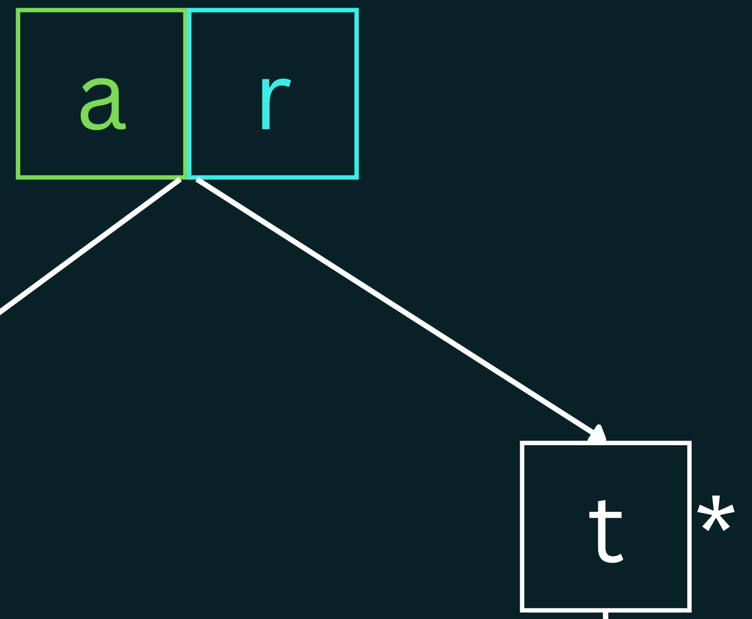
get words starting with...

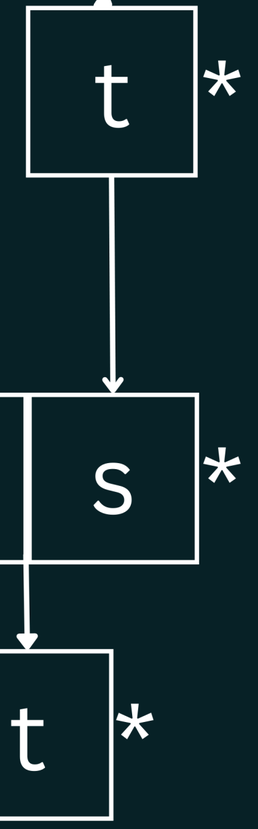
a



get words starting with...

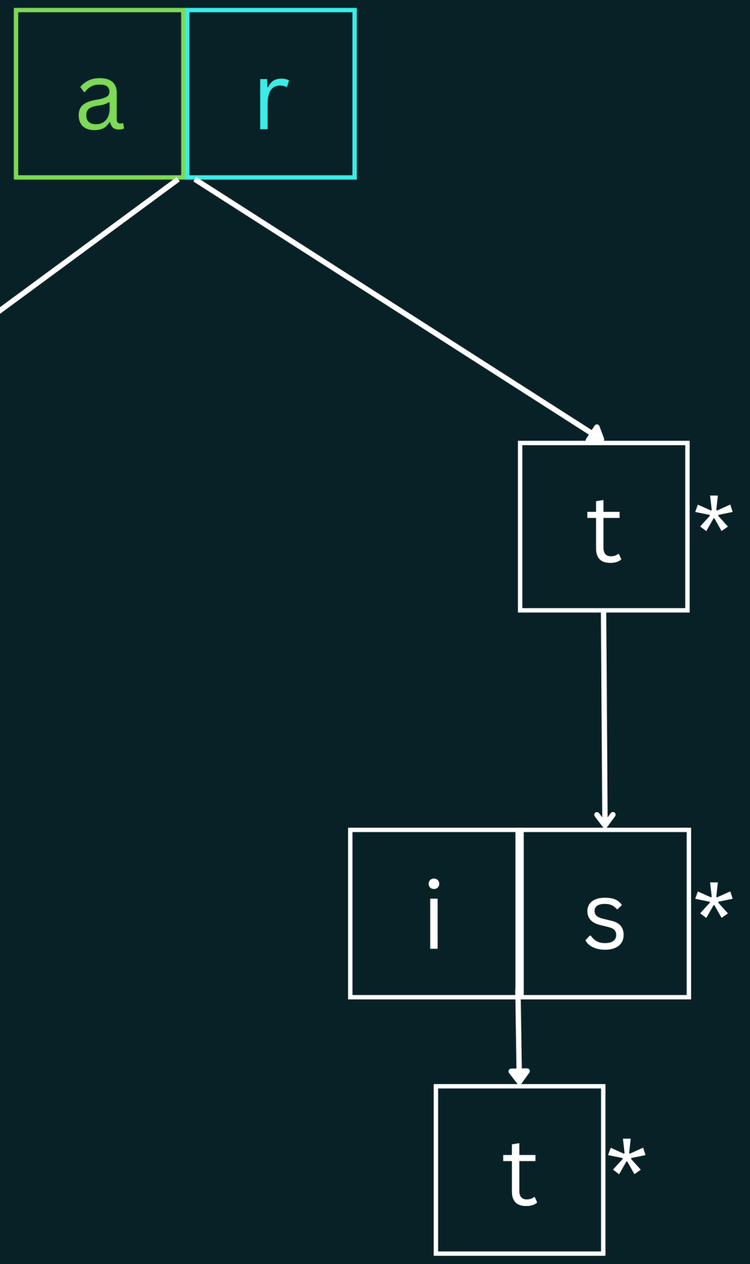
a

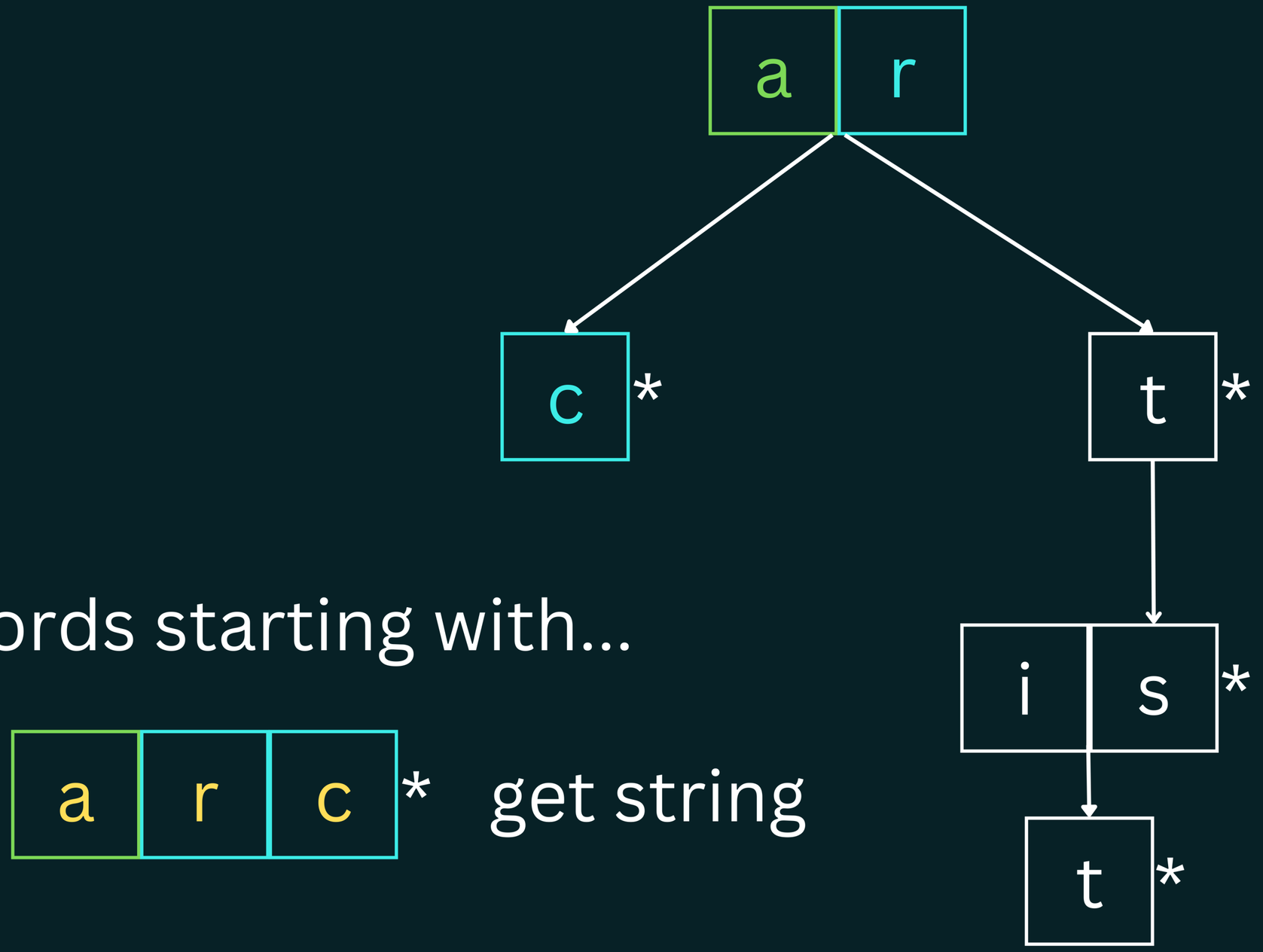




get words starting with...

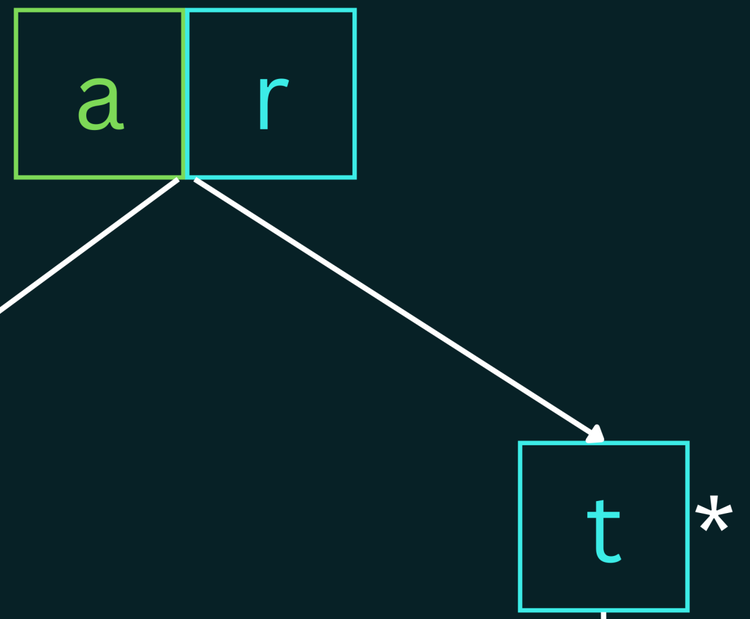
get words starting with...



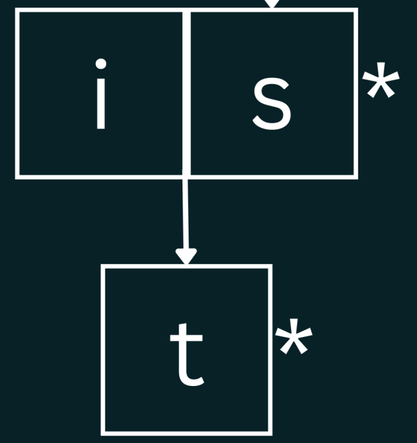


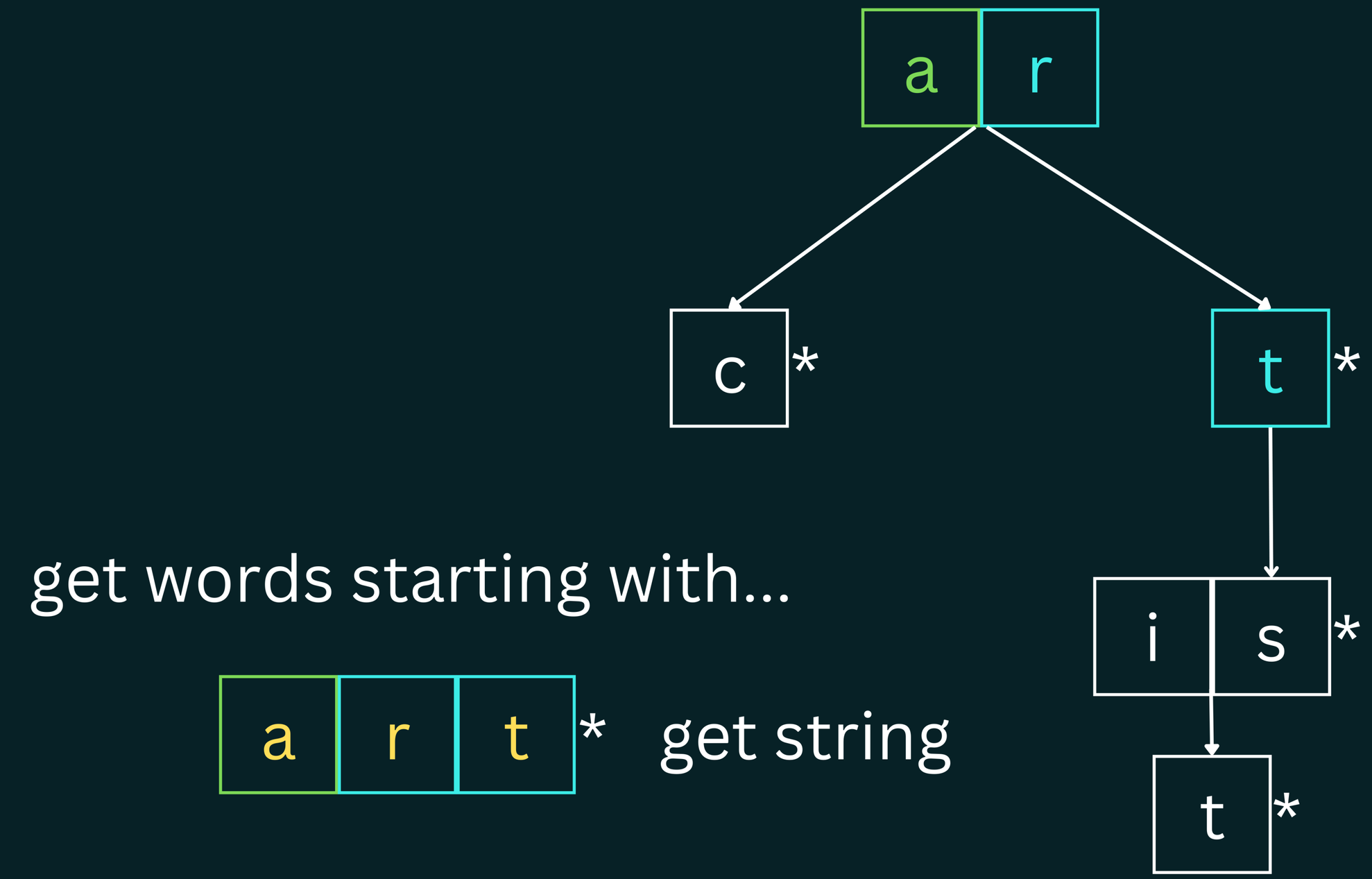
get words starting with...

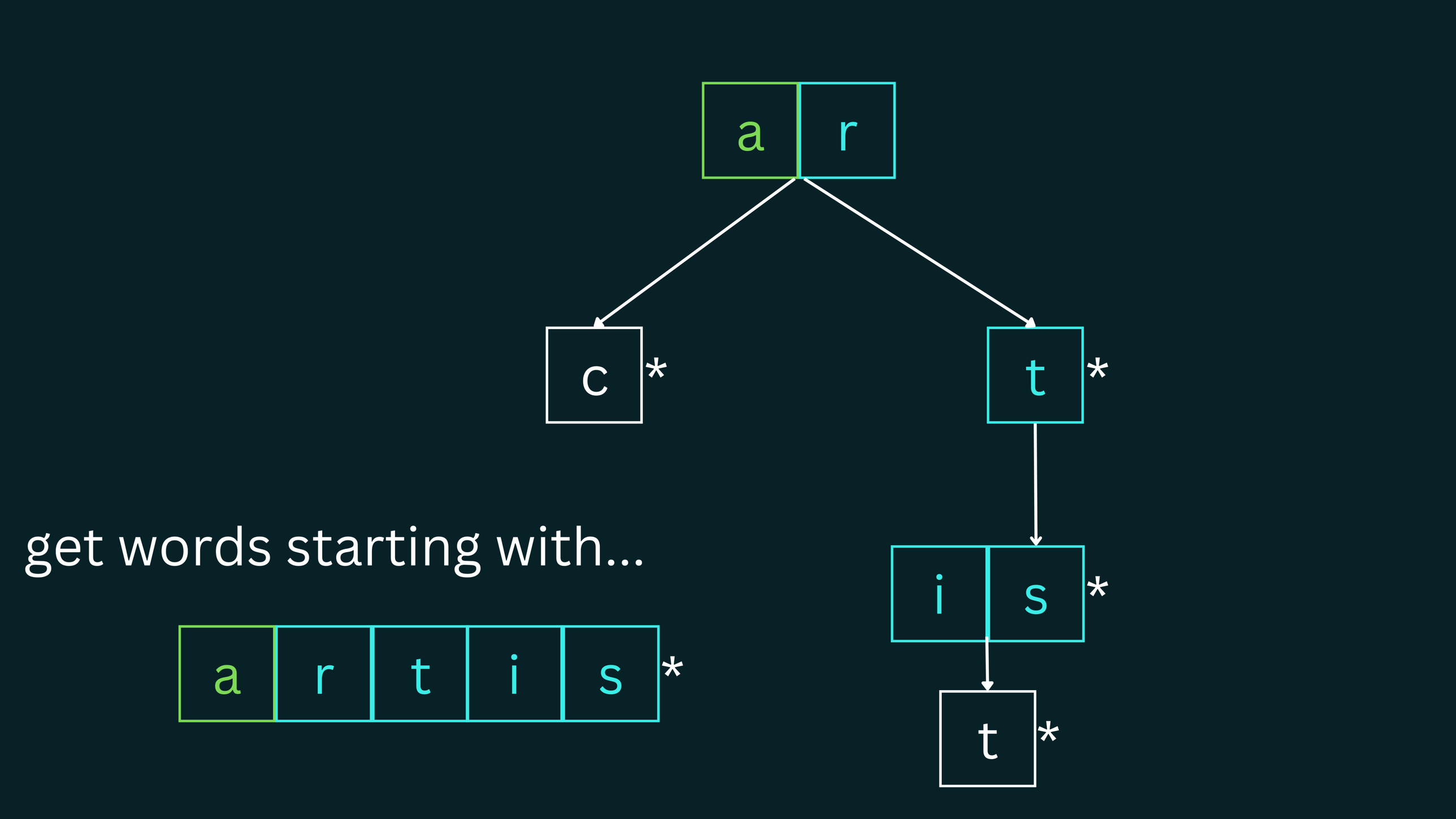
a r c * get string

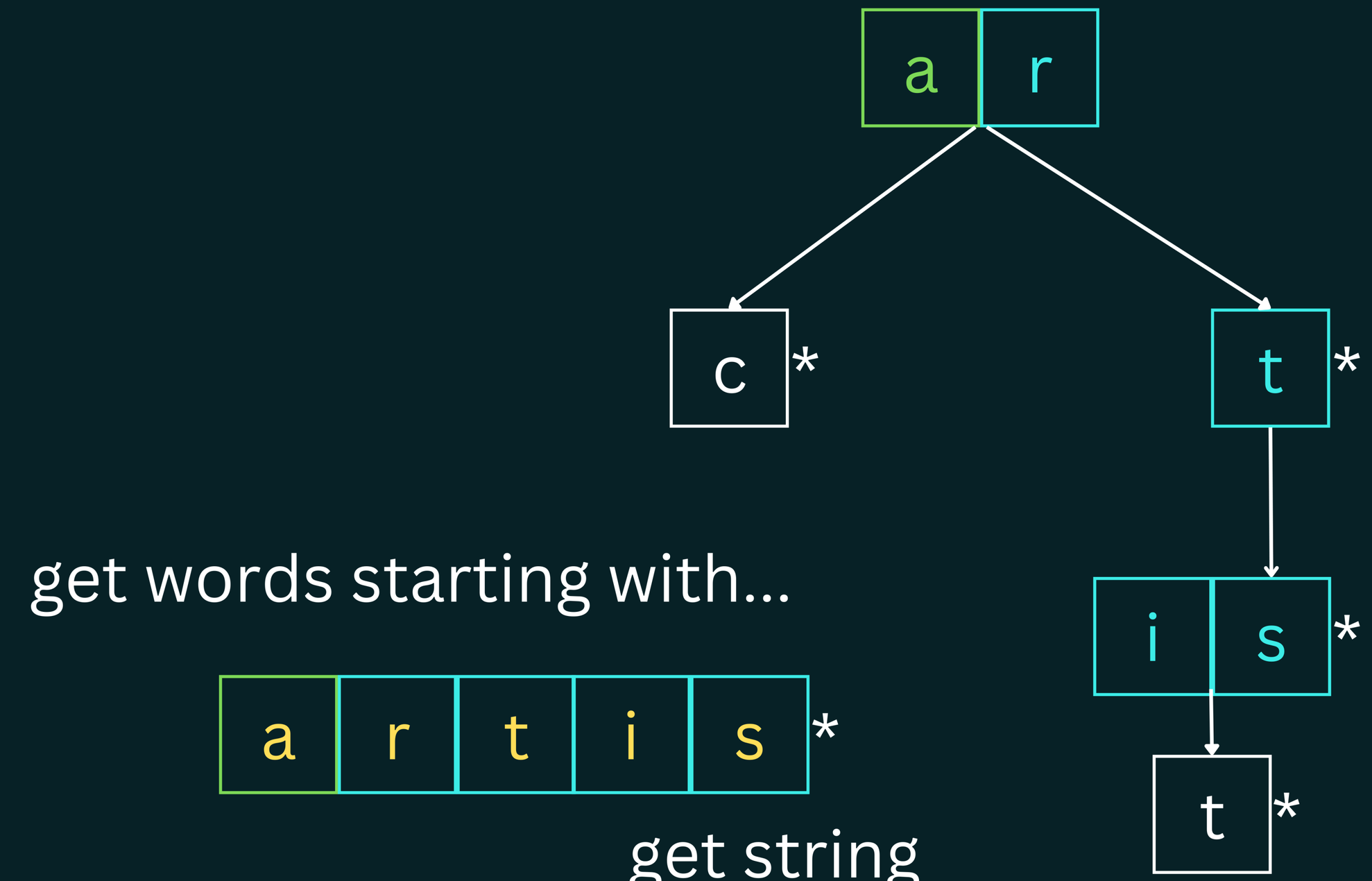


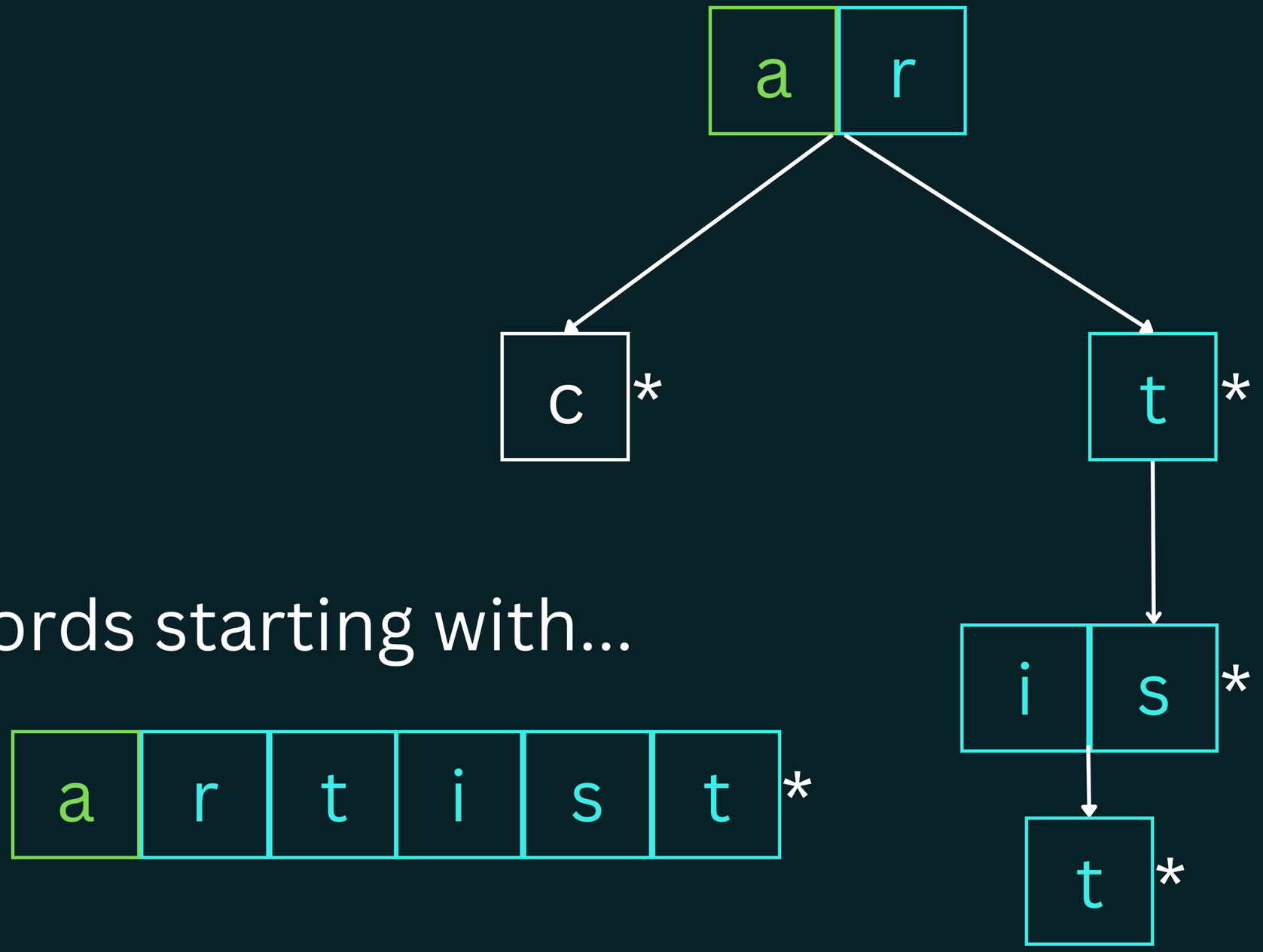
get words starting with...



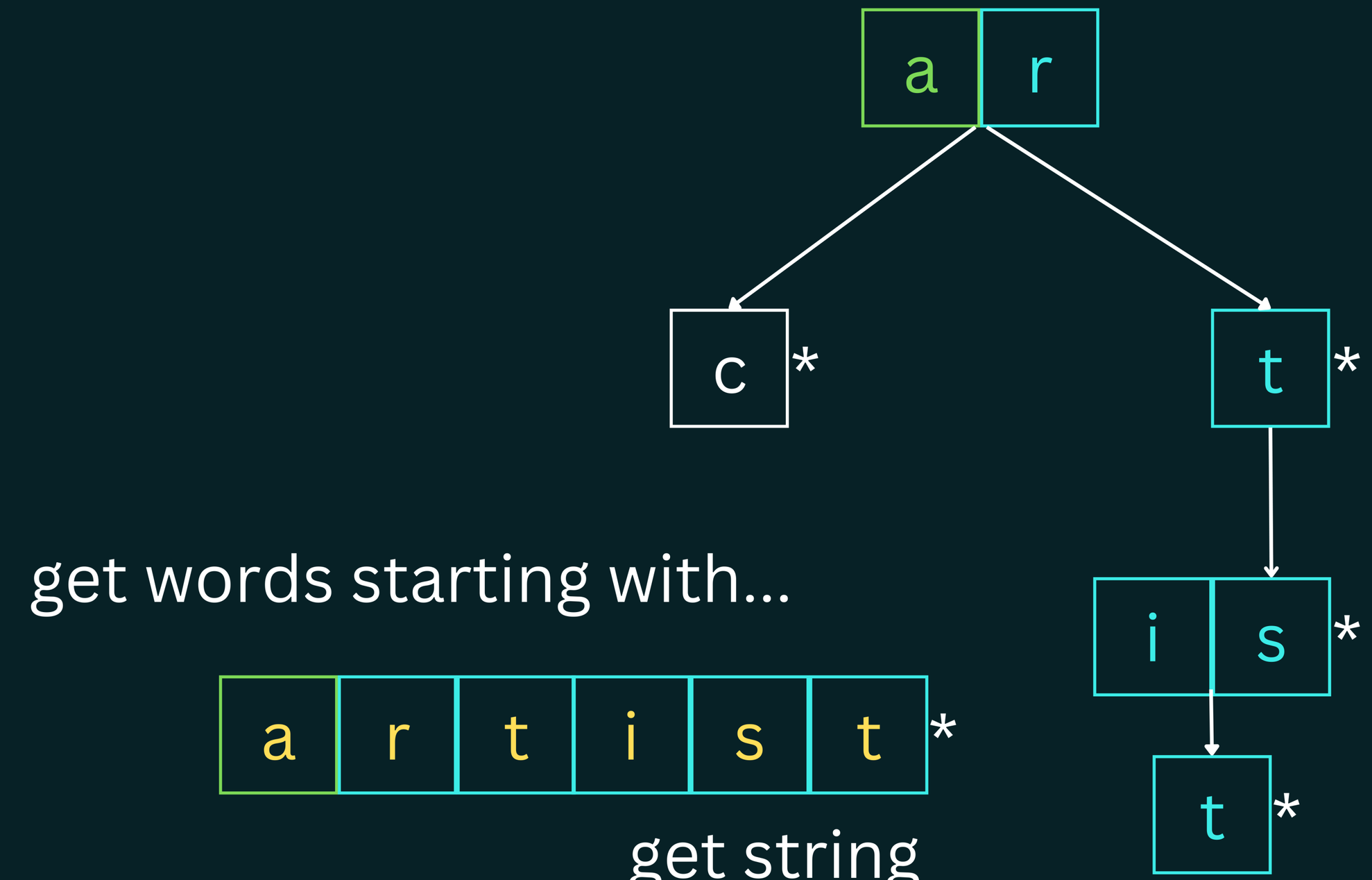

a
r
c
*
t
*
i
s
*
t
*
get words starting with...
a
r
t
*
get string


a
r
c
*
t
*
i
s
*
t
*
get words starting with...
a
r
t
i
s
*


a
r
c
*
t
*
i
s
*
t
*
get words starting with...
a
r
t
i
s
*
get string



get words starting with...


a
r
c
*
t
*
i
s
*
t
*
get words starting with...
a
r
t
i
s
t
*
get string


THANK
YOU