Структура от данни
Trie

Дефиниция

Структура от данни, базирана на дърво, която се използва за съхраняване на някакво множество от низове и извършване на ефективни операции за търсене върху тях.

Думата Trie произлиза от **reTRIEval**, което означава намиране, извличане или получаване на нещо.

Trie следва правилото:

Ако два низа имат общ префикс, тогава те ще имат един и същ предшественик в trie.

Това може да се използва за сортиране на множество от низове по азбучен ред, както и за търсене дали низ с даден префикс присъства в множеството или не.

Необходимост от структурата от данни Trie

Използва се за ***съхраняване*** и ***извличане*** на данни.

Тези операции могат да се извършват с помощта на хеш-таблица, но Trie може да изпълнява тези операции по-ефективно от хеш-таблица.

Предимство на Trie пред хеш-таблицата е, че може да се използва за **търсене, базирано на префикс**, докато хеш-таблицата не може да се използва по същия начин.

Предимства на Trie пред Хеш-таблицата:

- Можем ефективно да извършваме префиксно търсене (или автоматично попълване) с Trie.
- Можем лесно да отпечатаме всички думи по азбучен ред, което по-трудно се реализира с хеширане.
- Няма допълнителни разходи за хеш функции.

Предимства на Trie пред Хеш-таблицата:

- Търсенето на низ дори в голямо множество от низове в Trie може да се извърши за O(L) времева сложност, където L е броят на думите в низа на заявката. Това време за търсене може да бъде дори по-малко от O(L), ако низът в заявката не съществува в множеството.

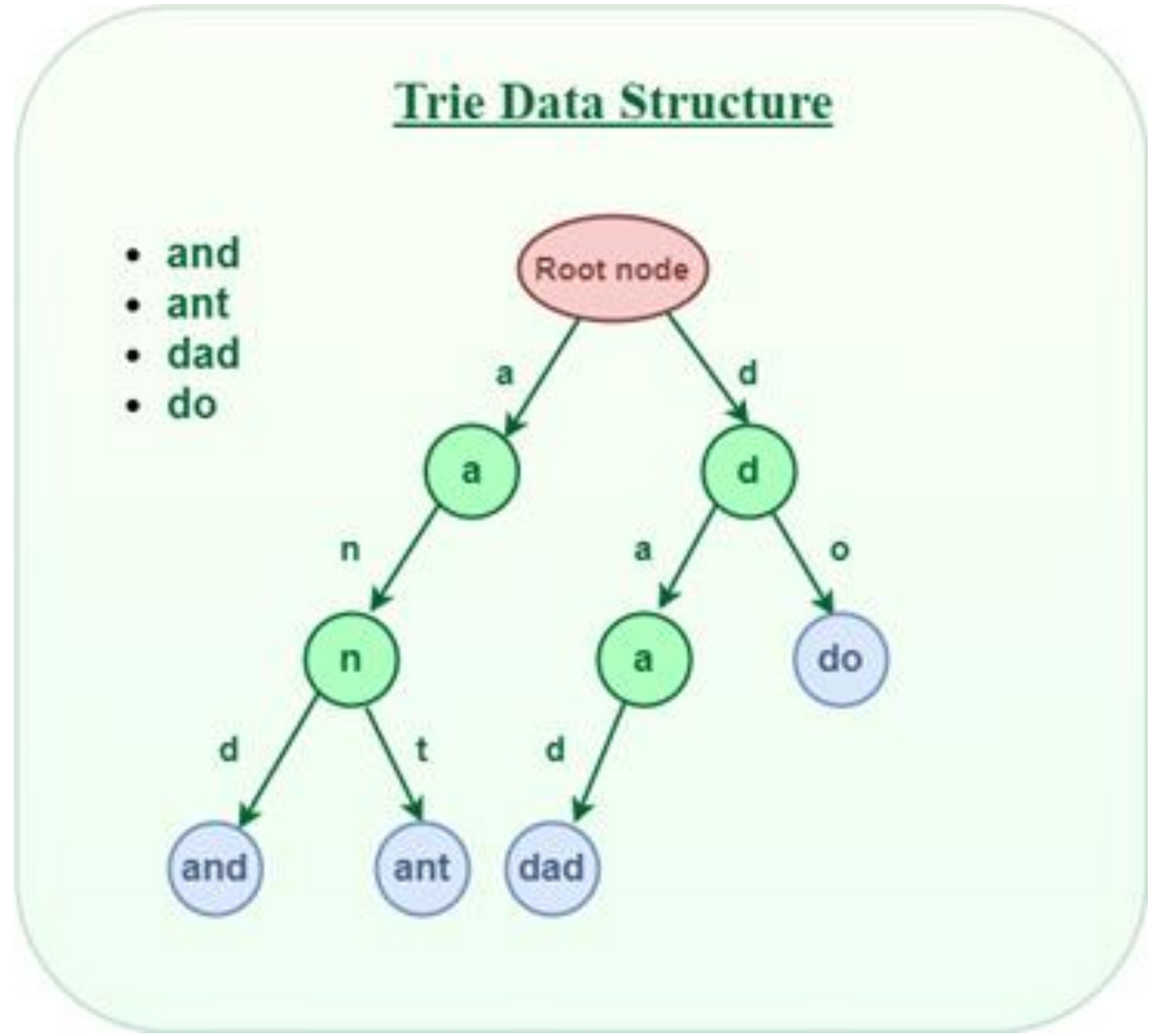
Свойства на Trie

- Всяка структура Trie има един коренов възел.
- Всеки възел представлява низ, а всеки лист представлява символ.
- Всеки възел се състои от хеш-карти или масив от указатели, като всеки индекс на масива представлява символ и флаг, който показва дали някой низ завършва в текущия възел.

Свойства на Trie

- Може да съдържа произволен брой знаци, включително азбуки, числа и специални знаци. Ще използваме низове, включващи знаците от a до z. Следователно са необходими само 26 указателя за всеки възел, където 0-ият индекс представлява „a“, а 25-ият индекс представлява символа „z“.
- Всеки път от корена до всеки възел представлява дума или низ.

Пример за Trie



Как работи Trie

Всяка английска дума, изписана с малки букви, може да започва с буква измежду a-z, след това следващата буква от думата може да бъде a-z, третата буква от думата отново може да бъде a-z и т.н.

Така че, за да съхраним дума, трябва да вземем масив (контейнер) с размер 26 и първоначално всички знаци са празни, тъй като няма думи и ще изглежда както е показано на следващия слайд.

Масив от указатели във всеки Trie възел

a b c d e f g h i j k l m n o p q r s t u v w x y z

Да покажем как думите “and” и “ant” се съхраняват в Trie:

1 Съхраняване на “and”

- Започва с “a“, така че маркираме “a” както е попълнено във възела Trie, което представлява използването на „a“.
- За втория символ отново има 26 възможности, така че от „a“, отново има масив с размер 26, за съхраняване на 2-ри знак.

Да покажем как думите “and” и “ant” се съхраняват в Trie:

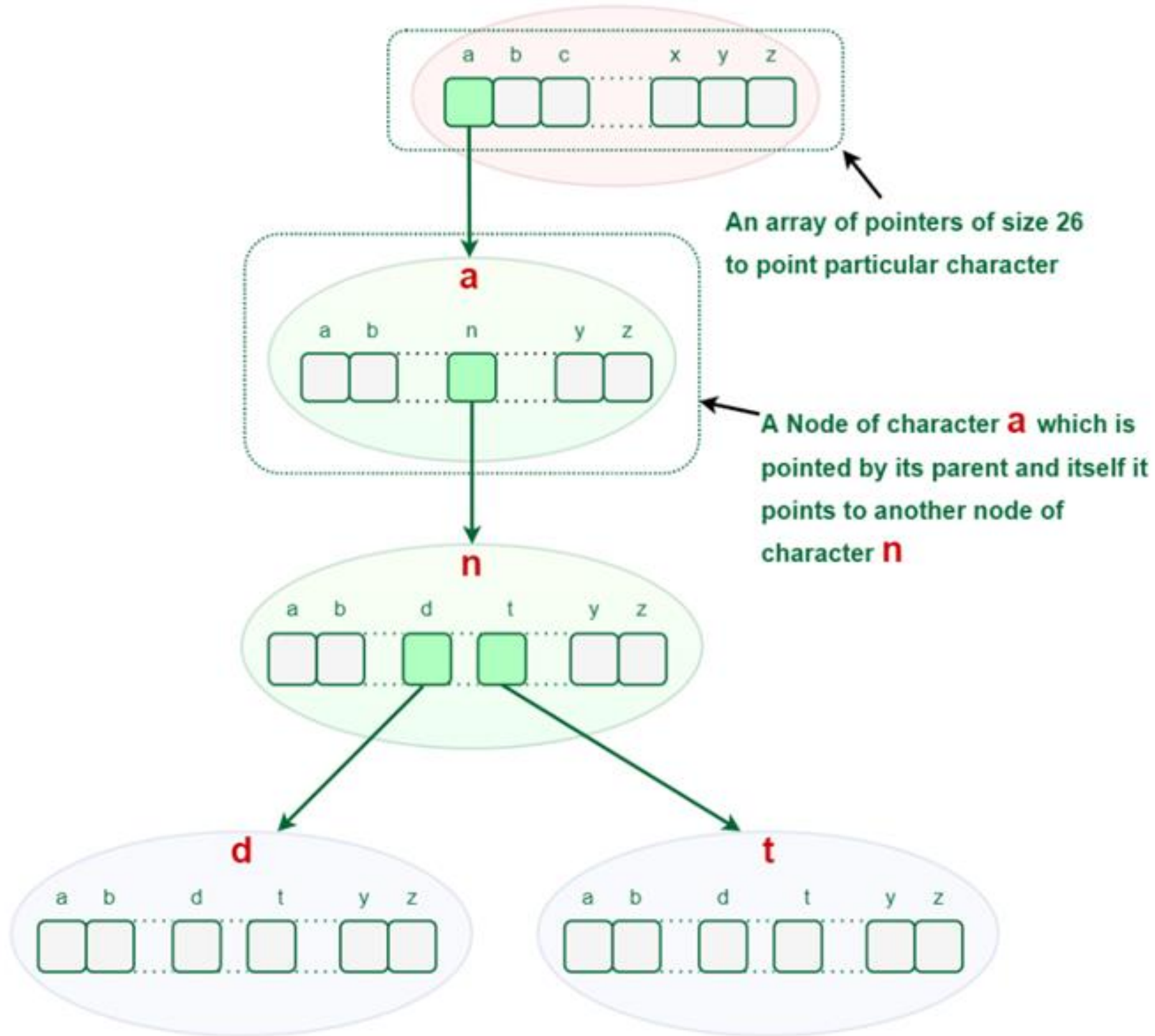
- Вторият символ е “n“, така от “a“, ние се преместваме в “n” и маркираме “n” във втория масив като използван.
- След “n“, третият символ е “d“, маркираме позицията “d” като използвана в съответния масив.

2 Съхраняване на “ant”

- Започва с “a” и позицията на “a” в корена вече е попълнена. Така че не е необходимо да я попълваме отново. Само се преместваме във връх ‘a‘ на Trie.
- За втория символ ‘n‘ констатираме, че ‘n’ във възела ‘a’ също е запълнен. Така че не е необходимо да я попълваме отново. Само се преместваме във връх ‘n‘ на Trie.

- За последния символ ‘t‘ на думата, позицията на ‘t‘ във възела ‘n‘ не е запълнена. Така че маркираме позицията на ‘t‘ във възела ‘n‘ и се придвижваме във възел ‘t‘.

Така получаваме структурата Trie:



Представяне на Trie в С++:

```
#define N 26  //Брой букви
typedef struct TrieNode TrieNode;
struct TrieNode {
// Всеки възел има N деца
// и флаг за проверка дали е листо
char data;
TrieNode* children[N];
int is_leaf; //Дали е край на дума
};
```

Дефиниране на структурата

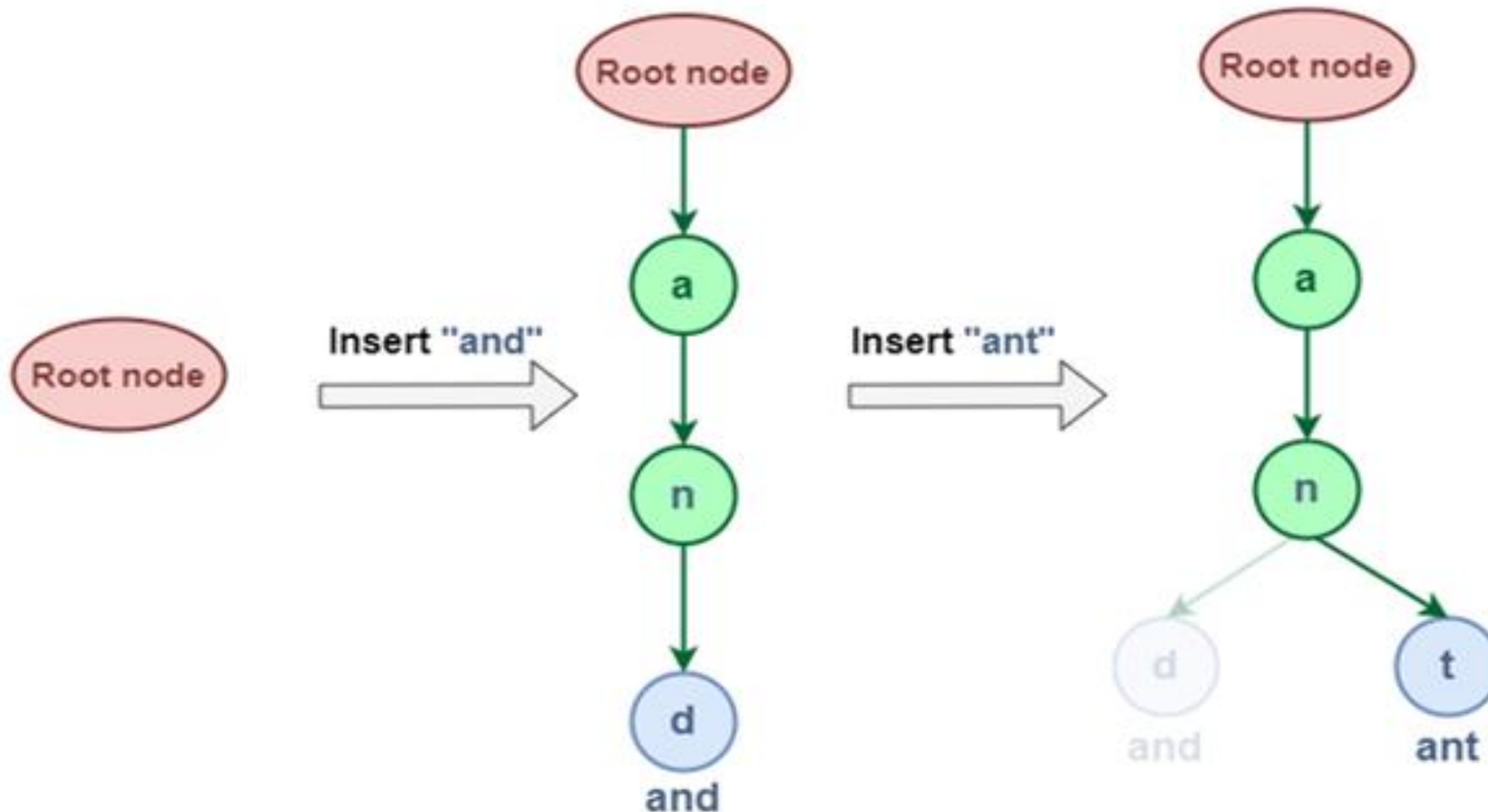
```
TrieNode* make_trienode(char data) {
TrieNode* node = (TrieNode*) calloc (1,
sizeof(TrieNode));
  for (int i=0; i<N; i++)
        node->children[i] = NULL;
  node->is_leaf = 0;
  node->data = data;
return node;}
```

Освобождаване на паметта

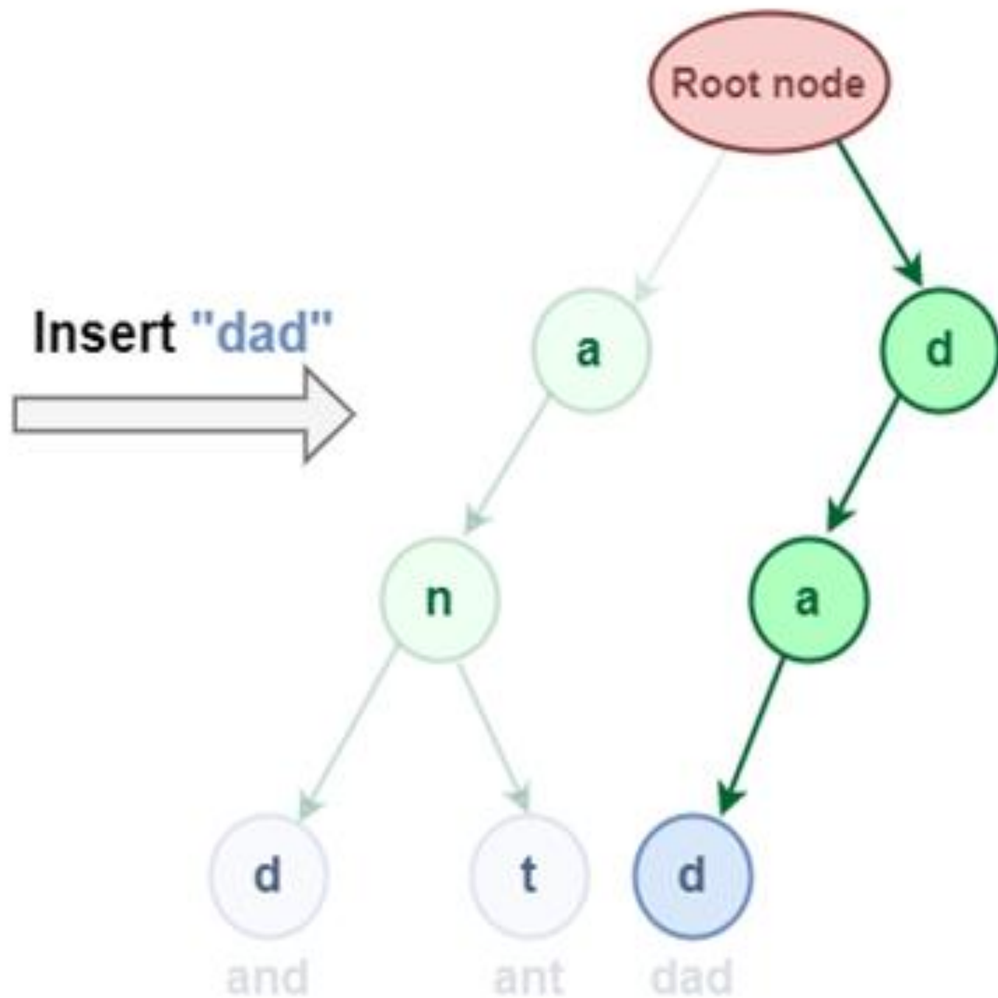
```
void free_trienode(TrieNode* node) {
  for(int i=0; i<N; i++) {
    if (node->children[i] != NULL)
        free_trienode(node->children[i]);
     else continue;
  }
    free(node);
}
```

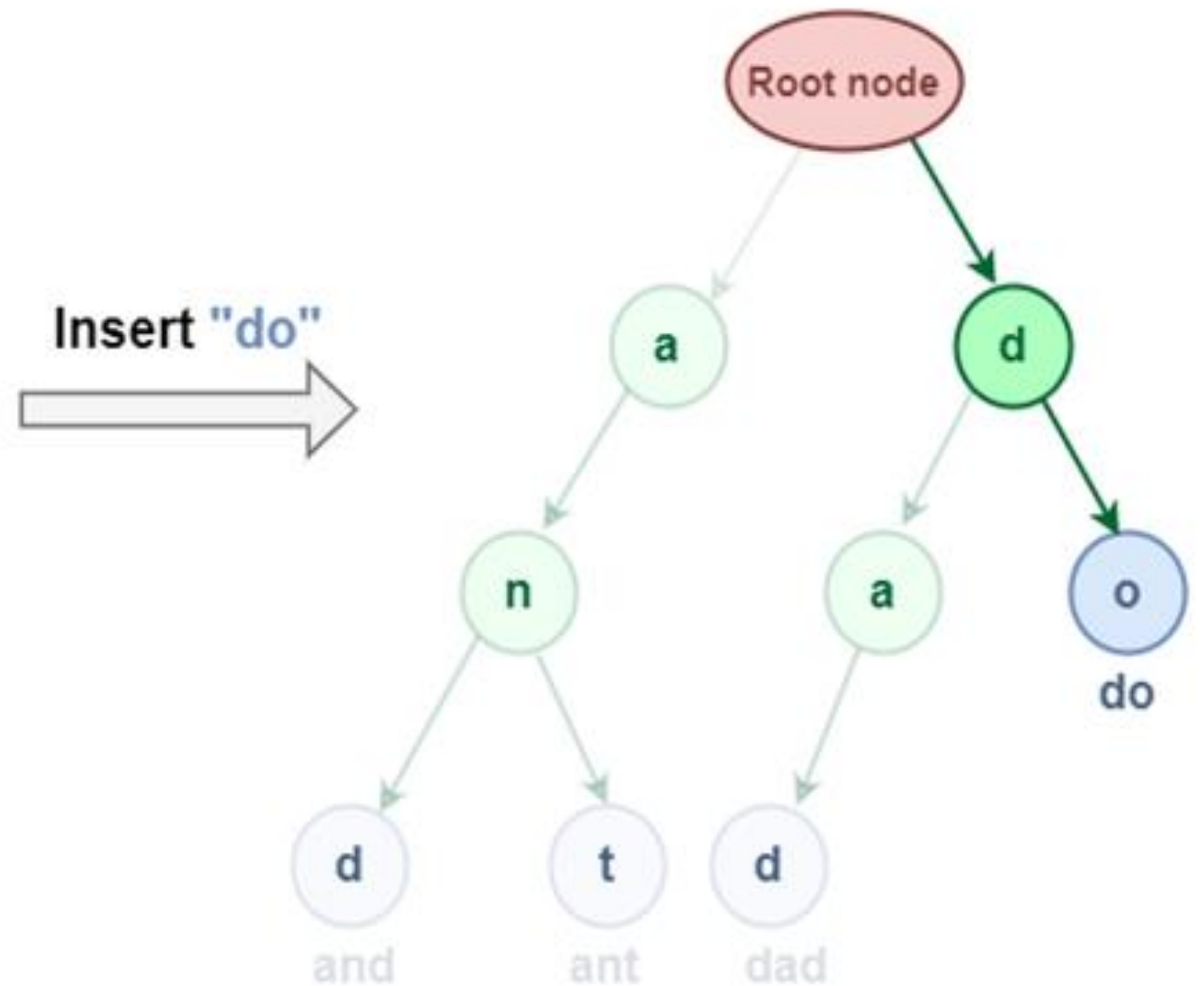
Операции със структурата Trie:

1. ***Вмъкване:*** Нека да вмъкнем “and” & “ant”:



Нека сега да вмъкнем “dad” & “do”:





Реализация на вмъкване в Trie:

Функцията insert_trie() получава указател към корена на дървото и вмъква дума в Trie.

Функцията итерира през символите на думата и намира позицията на всеки символ.

```
TrieNode* insert_trie(TrieNode* root, char*
word) {
 TrieNode* temp = root;
 for (int i=0; word[i] != '\0'; i++) {
  int idx = (int) word[i] - 'a';
  if (temp->children[idx] == NULL)
  temp->children[idx]= make_trienode(word[i]);
  temp = temp->children[idx];
 }
```

```
// В края на думата, маркираме
// върха като краен
    temp->is_leaf = 1;
 return root;
}
```

2. Търсене в Trie

Търсене в Trie се извършва по подобен начин като операцията за вмъкване. Единствената разлика е, че всеки път, когато открием, че масивът от указатели във възел curr не сочи към текущия знак на думата, тогава се връща *false* вместо създаване на нов възел за този символ на думата.

Използва се за търсене дали даден низ присъства в структурата на данни Trie или не.

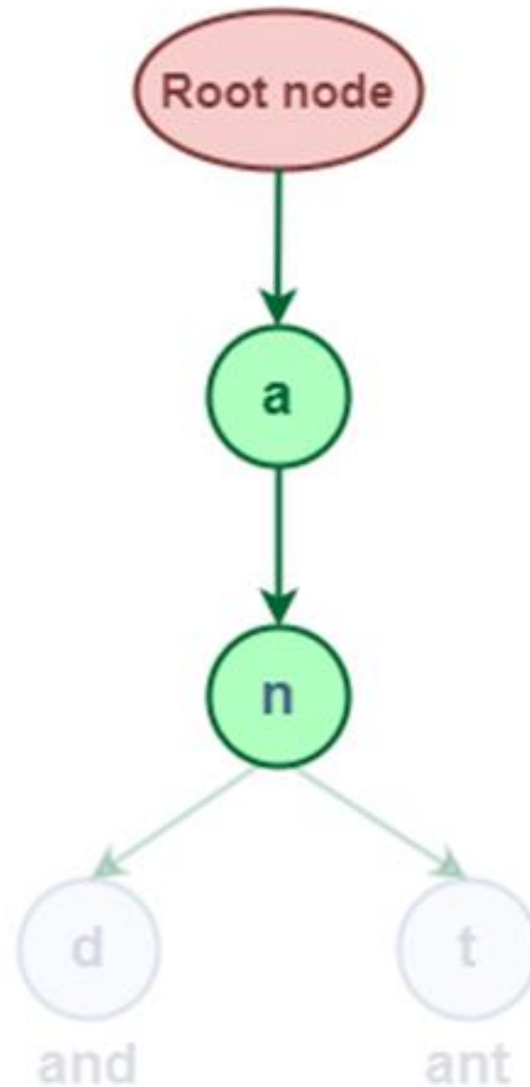
Има два подхода за търсене в Trie:

1. Да се търси дали в Trie съществува дума, която започва с даден префикс.
2. Да се търси дали дадена дума съществува в Trie.

И в двата подхода има подобен модел на търсене.

Първата стъпка при търсене на дадена дума в Trie е да преобразуваме думата в знаци и след това да сравним всеки знак с trie възела от основния възел. Ако текущият символ присъства във възела, преминаваме напред към неговите деца. Повторяме този процес, докато бъдат намерени всички знаци.

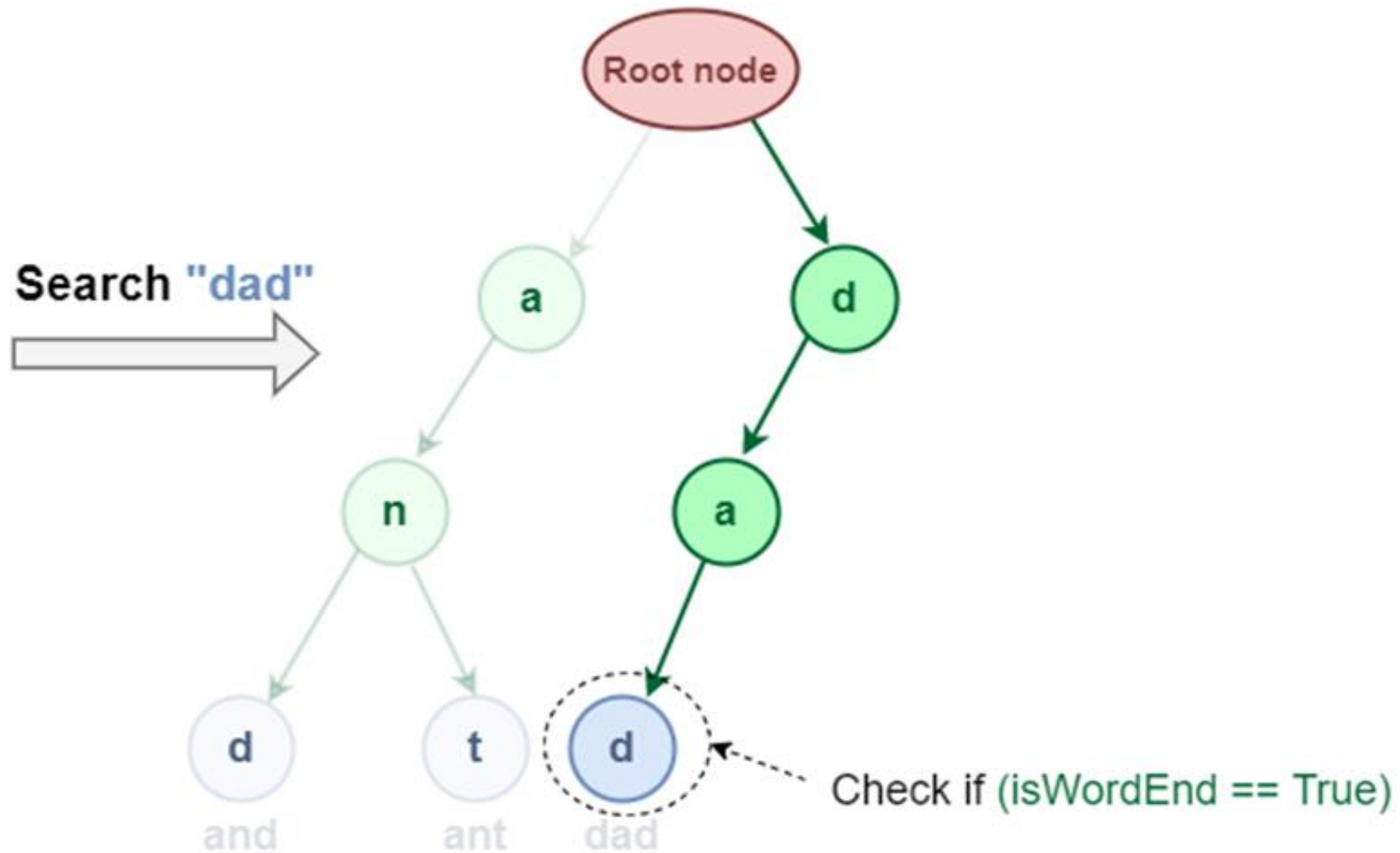
Търсене на префикс в Trie



Примерна реализация

```
bool isPrefixExist(TrieNode* root, string& key){
  TrieNode* currentNode = root; //коренът е текущ
връх
  for (auto c : key) { //движим се по стринга
     if (currentNode->childNode == NULL)
         return false;//Префиксът не съществува
     currentNode = currentNode->childNode;
  }
  // Префиксът съществува в структурата
    return true;
}
```

Търсене на дума в Trie



Примерна реализация

```
bool search_key(TrieNode* root, string& key){
  TrieNode* currentNode = root;
  for (auto c : key) {
    if (currentNode->childNode == NULL)return false;
    currentNode = currentNode->childNode;
    }
//Проверяваме дали сме стигнали до край на дума
    return (currentNode->wordCount > 0);
}
```

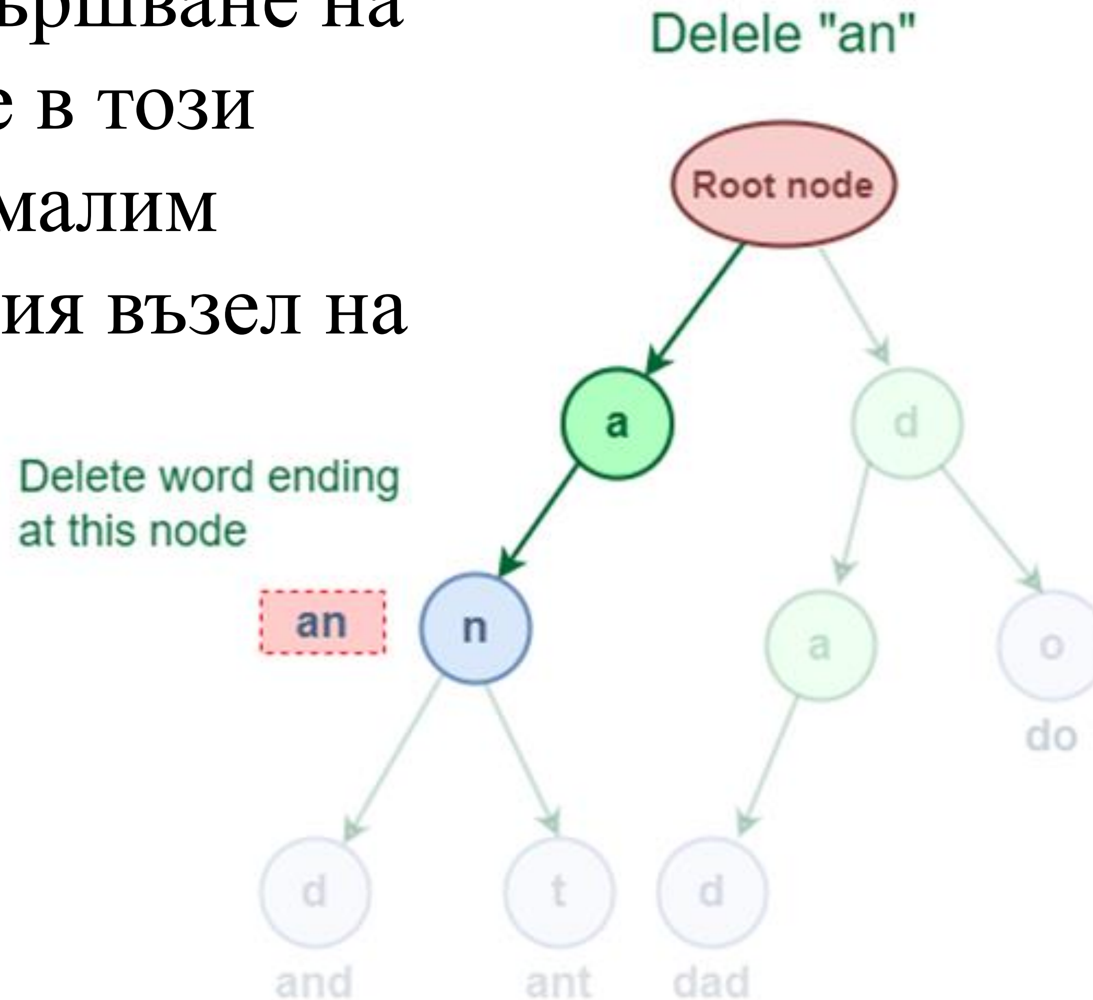
Изтриване в Trie

Възможни са три случая при изтриване на дума от Trie:

1. Изтритата дума да е префикс на други думи в Trie.
2. Изтритата дума да споделя общ префикс с други думи в Trie.
3. Изтритата дума да не споделя общ префикс с други думи в Trie.

Изтритата дума да е префикс на други думи в Trie

Лесно решение за извършване на операция за изтриване в този случай е просто да намалим wordCount с 1 в крайния възел на думата.



Изтритата дума да споделя общ префикс с други думи в Trie.

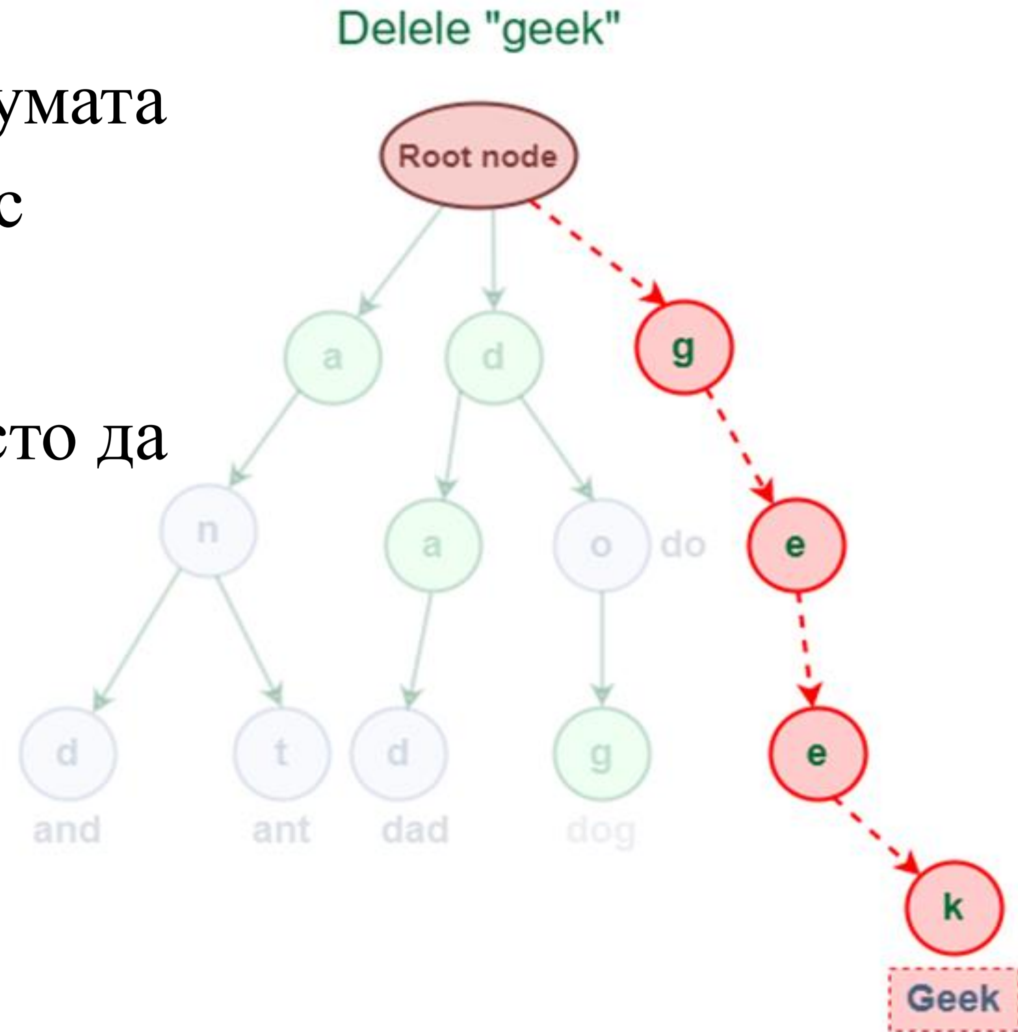
Решението за този случай е да изтрием всички възли, започвайки от края на префикса до последния знак на дадената дума.

Case 2: The deleted word share common prefix with other words

Изтритата дума да не споделя общ префикс с други думи в Trie

Както е показано на фигурата, думата „geek“ не споделя общ префикс с други думи.

Решението за този случай е просто да изтрием всички възли.



Case 3: The deleted word does not share any common prefix with other words

Примерна реализация

```
bool delete_key(TrieNode* root, string& word){
    TrieNode* currentNode = root;
    TrieNode* lastBranchNode = NULL;
    char lastBrachChar = 'a';
    for (auto c : word) {
        if (currentNode->childNode == NULL) return
false;
        int count = 0;
        for (int i = 0; i < 26; i++) {
          if (currentNode->childNode[i] != NULL) count++;
            }
```

```
        if (count > 1) {
            lastBranchNode = currentNode;
            lastBrachChar = c;
        }
        currentNode = currentNode->childNode;
    }
    int count = 0;
    for (int i = 0; i < 26; i++)
        if (currentNode->childNode[i] != NULL) count++;
// Case 1: The deleted word is a prefix of other words in Trie.
     if (count > 0) {currentNode->wordCount--;
          return true; }
```

```
// Case 2: The deleted word shares a common prefix with other words
// in Trie.
    if (lastBranchNode != NULL) {
        lastBranchNode->childNode[lastBrachChar] = NULL;
        return true;
    }
// Case 3: The deleted word does not share any common prefix with
// other words in Trie.
    else {
        root->childNode[word[0]] = NULL;
        return true;
    }
}
```

https://www.hackerearth.com/practice/notes/lalitkundu95/tutorial-on-trie-and-example-problems/

https://www.geeksforgeeks.org/advantages-trie-data-structure/

https://leetcode.com/problems/implement-trie-prefix-tree/description/