Προχωρημένες Δομές Δεδομένων

Απρίλιος 2011

Γιάννης Χατζημίχος
feedward@gmail.com

Προκατασκευαστικό Camp 23ου ΠΔΠ

Προχωρημένες Δομές Δεδομένων

1) **Union-Find (Disjoint sets)**
2) **Hash tables** -- Βαλκανιάδες
3) **Tries**
4) Δέντρα
   - Interval Trees
   - **Binary Indexed Trees**
   - Quad Trees -- Βαλκανιάδες
5) Suffix Arrays -- Βαλκανιάδες

Tries

- Είναι μια δενρική δομή δεδομένων που μας επιτρέπει να αποθηκεύουμε σύνολα συμβολοσειρών και να εκτελούμε πράξεις σε αυτά.
- Μπορούμε να εκτελούμε πράξεις on-line (δεν χρειάζεται να έχουμε όλα τα δεδομένα από πριν).
- Χρησιμοποιούνται συχνά ως αναπαράσταση λεξικών.

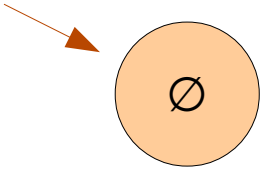
Tries: Πράξεις

- Προσθήκη συμβολοσειράς
- Αφαίρεση συμβολοσειράς
- Έλεγχος για ύπαρξη συμβολοσειράς
- Αναζήτηση/καταμέτριση συμβολοσειρών με το ίδιο πρόθεμα
- Αλφαβητική ταξινόμηση συμβολοσειρών
  - Εύρεση κ-οστης λέξης
- ...

Tries: Κατασκευή

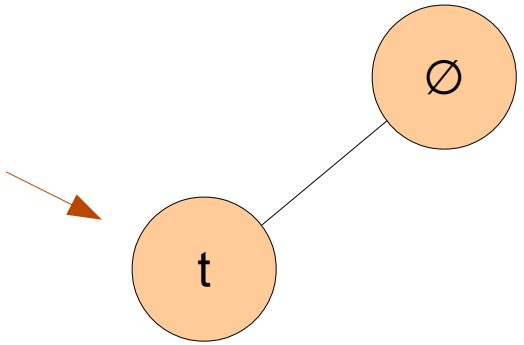
- Αρχικά έχουμε μια “ψεύτικη” ρίζα που αντιστοιχεί στην κενή συμβολοσειρά.
- Καθένας από τους υπόλοιπους κόμβους αντιστοιχεί σε κάποιο γράμμα της αλφαβήτου.
- Για να προσθέσουμε μια λέξη ξεκινάμε από την ρίζα και ακολουθούμε το μονοπάτι που δείχνουν τα γράμματά της, προσθέτοντας όσους κόμβους δεν υπάρχουν.

Tries: Παράδειγμα κατασκευής



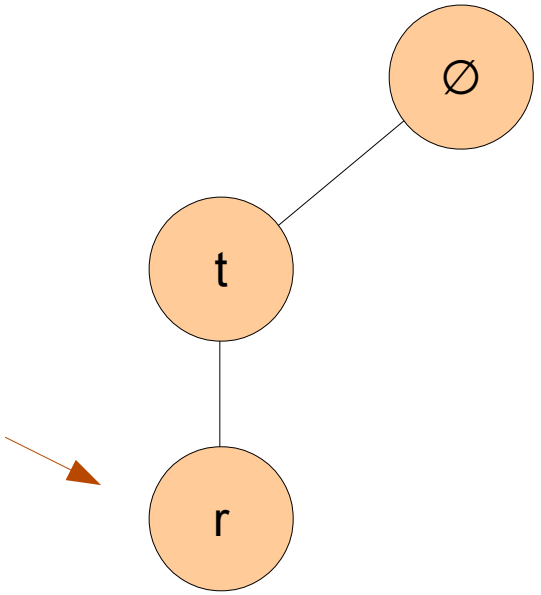
add “tree”

Tries: Παράδειγμα κατασκευής



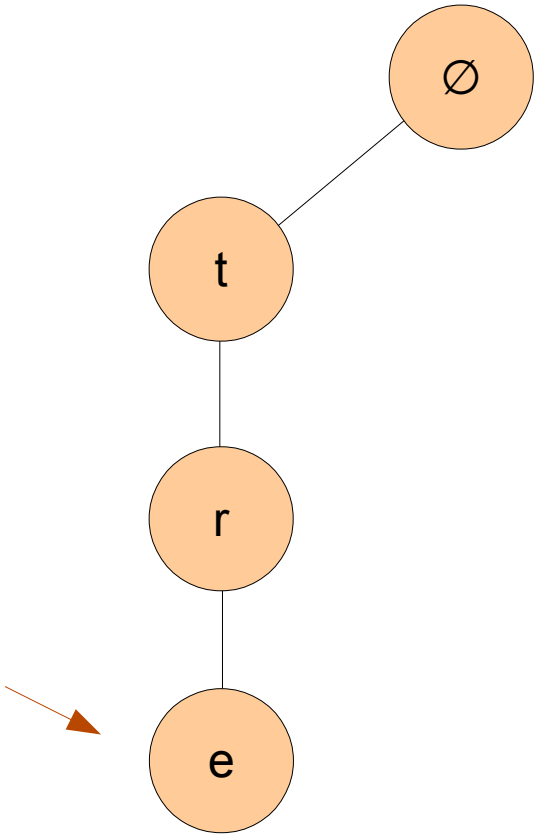
add “tree”

Tries: Παράδειγμα κατασκευής



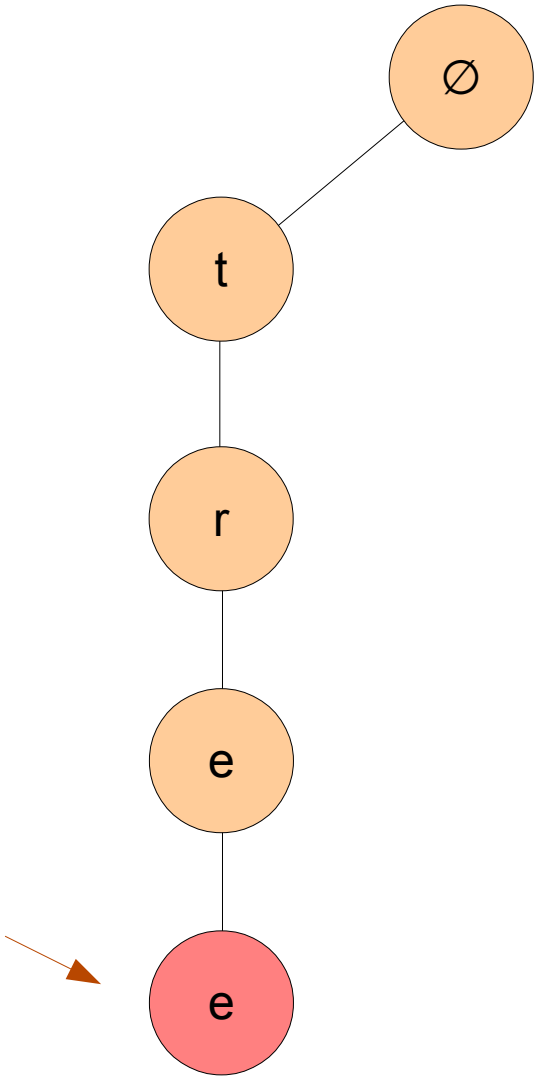
add “tree”

Tries: Παράδειγμα κατασκευής



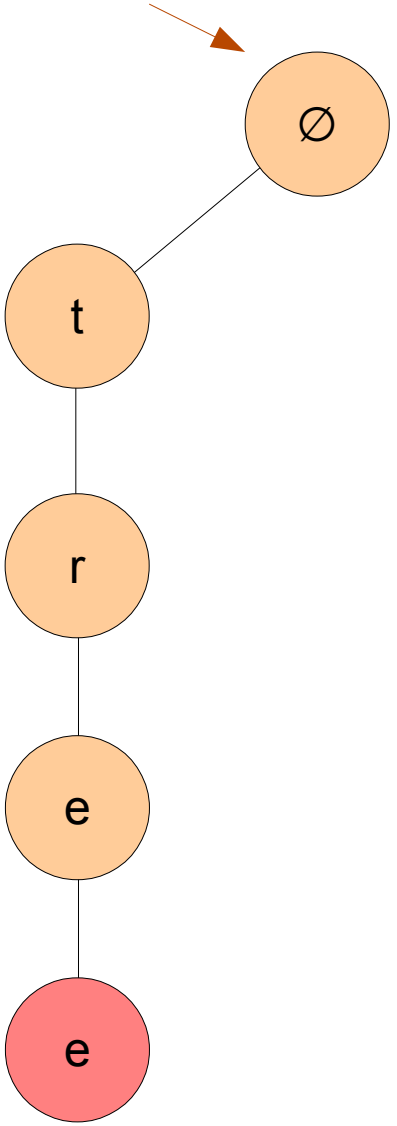
add “tree”

Tries: Παράδειγμα κατασκευής



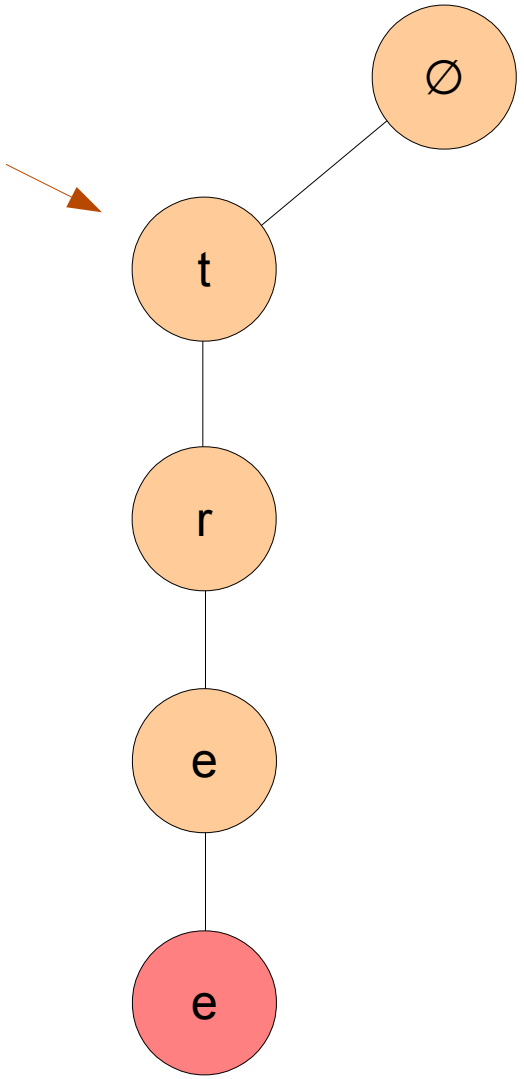
add “tree”

Tries: Παράδειγμα κατασκευής



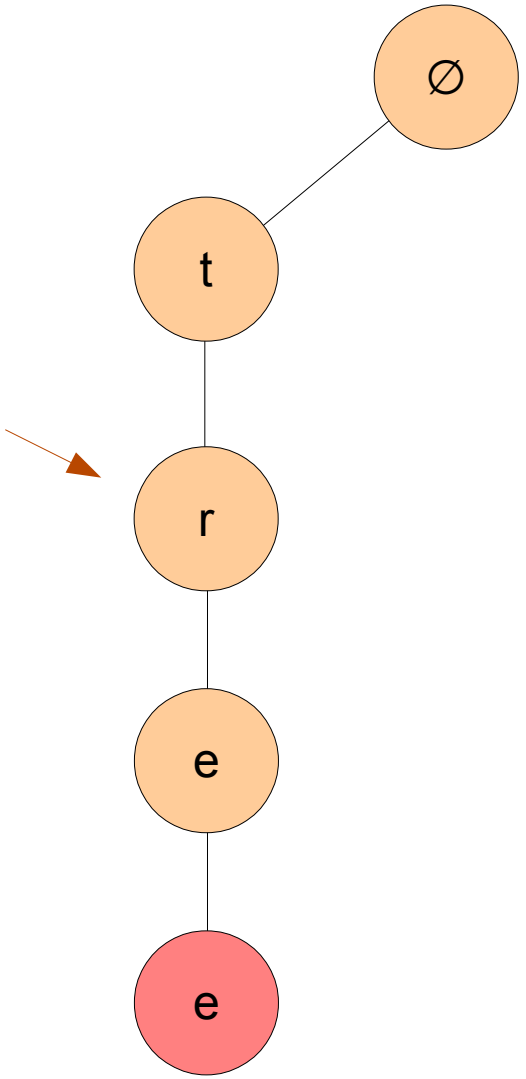
add “tree”
add “trie”

Tries: Παράδειγμα κατασκευής



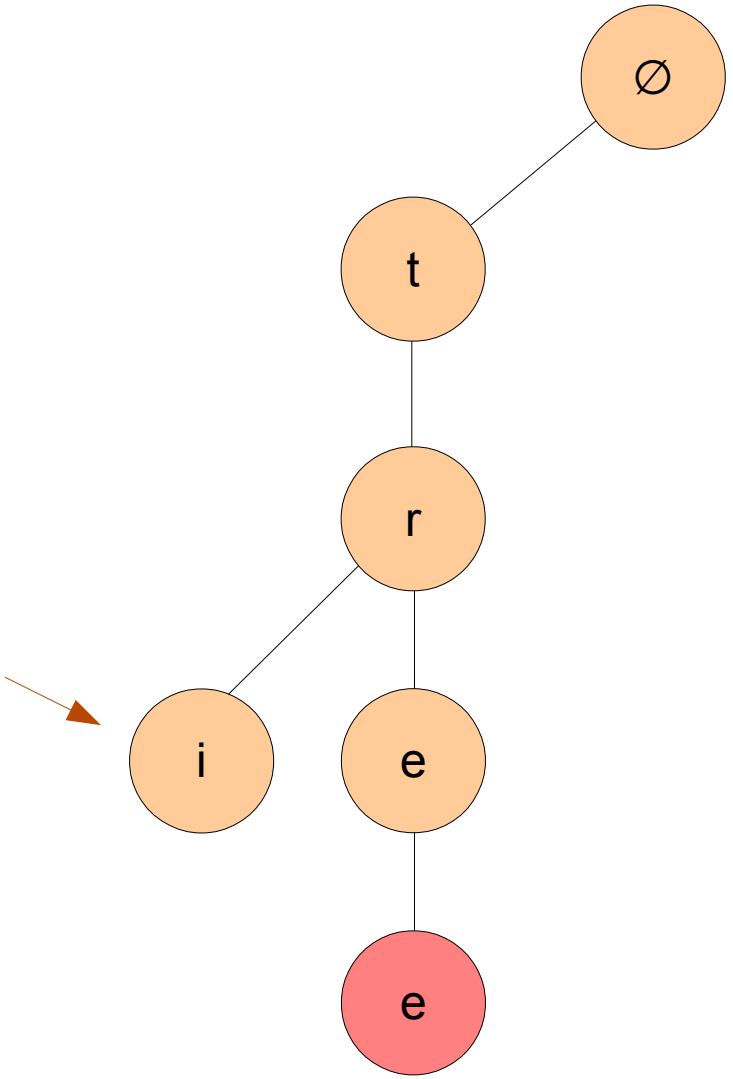

```
add “tree”
add “trie”
```

Tries: Παράδειγμα κατασκευής



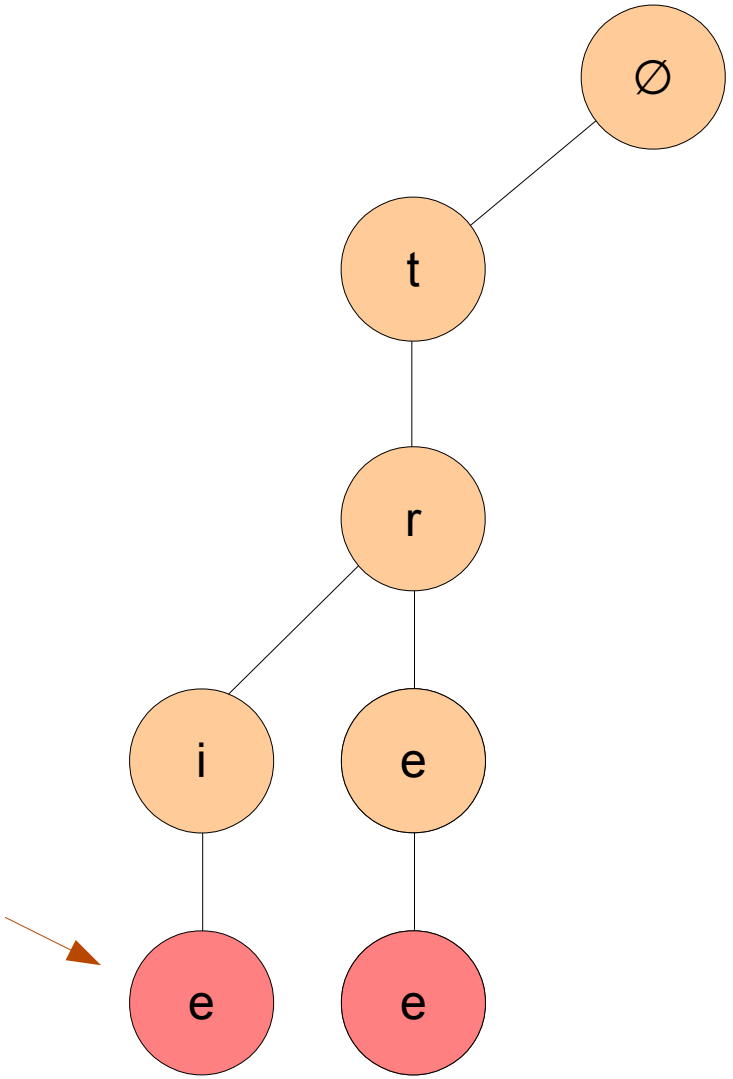
add “tree”
add “trie”

Tries: Παράδειγμα κατασκευής



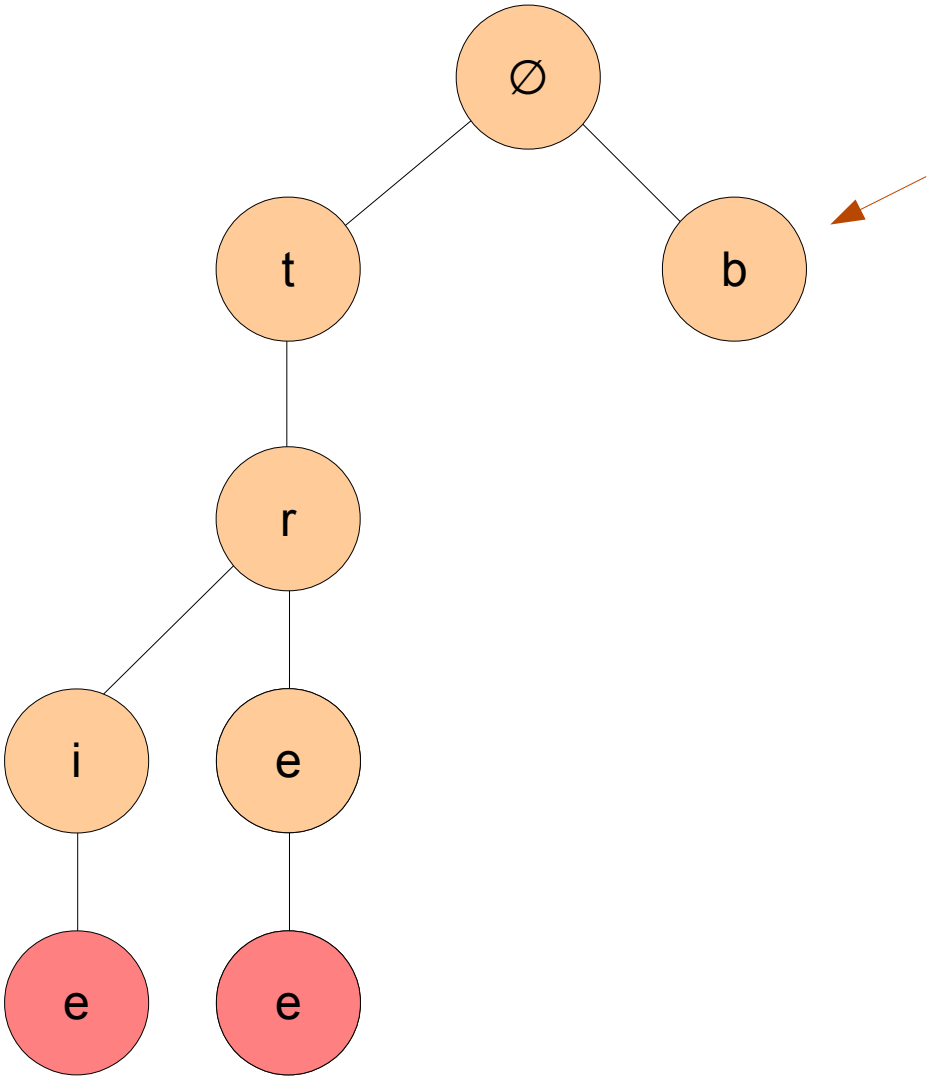
add “tree”
add “trie”

Tries: Παράδειγμα κατασκευής



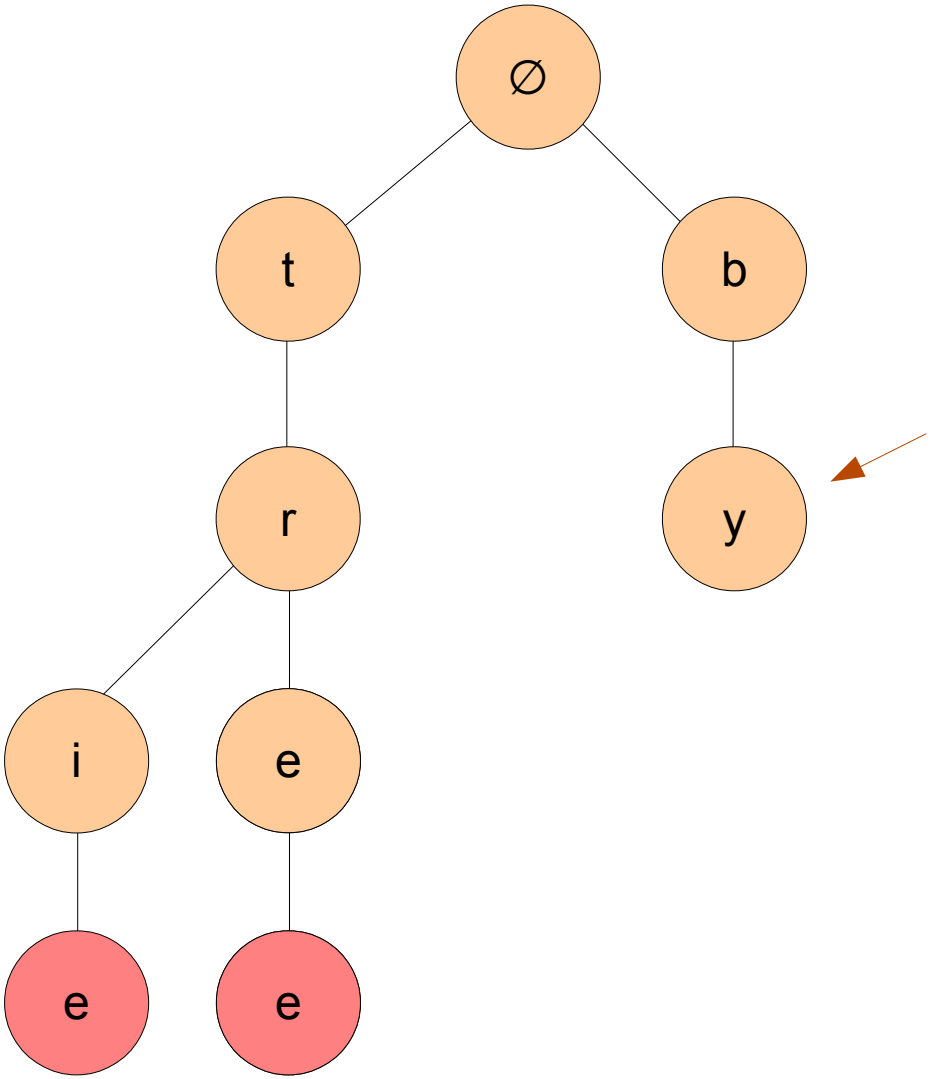
add “tree”
add “trie”

Tries: Παράδειγμα κατασκευής



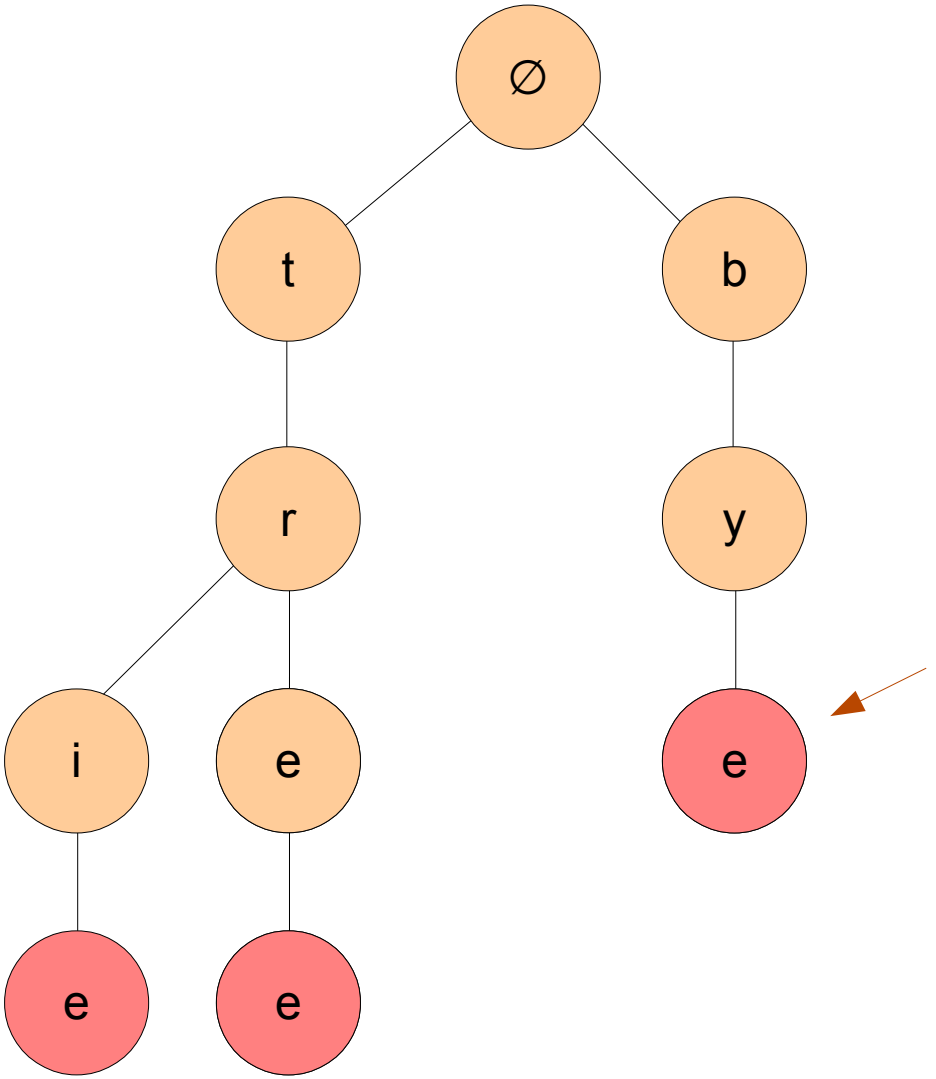

```
add “tree”
add “trie”
add “bye”
```

Tries: Παράδειγμα κατασκευής



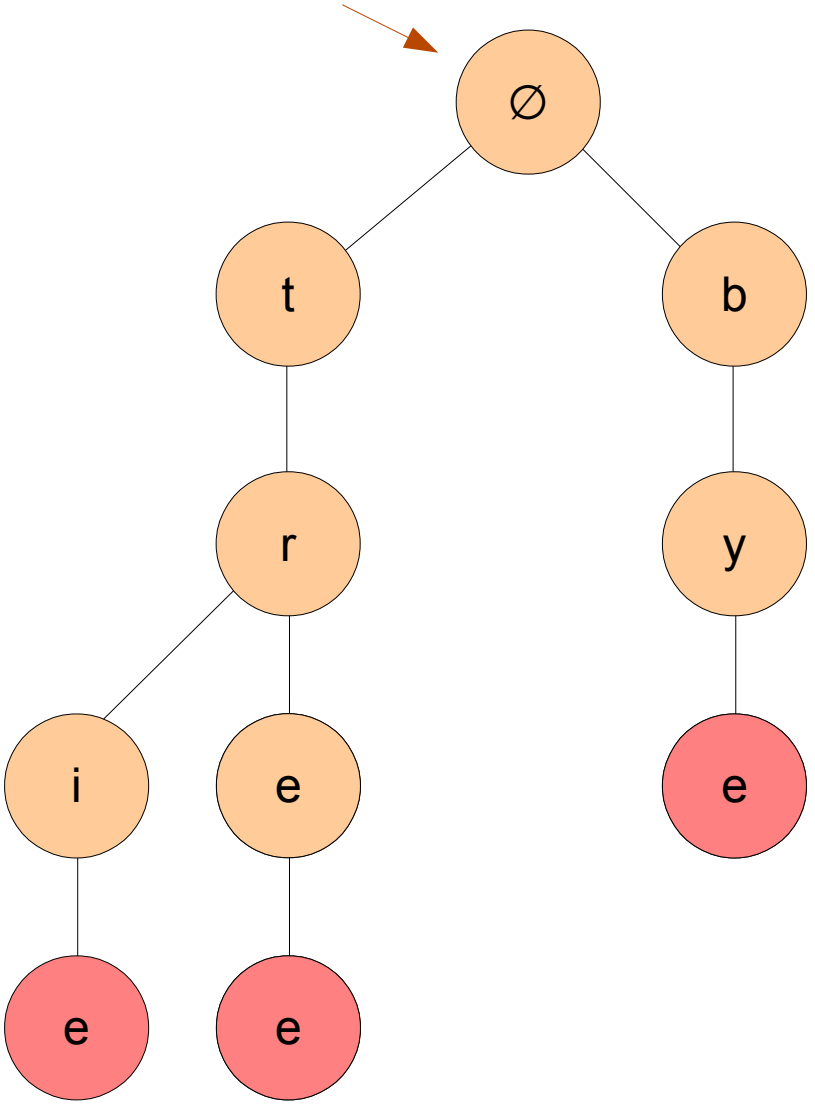

```
add “tree”
add “trie”
add “bye”
```

Tries: Παράδειγμα κατασκευής



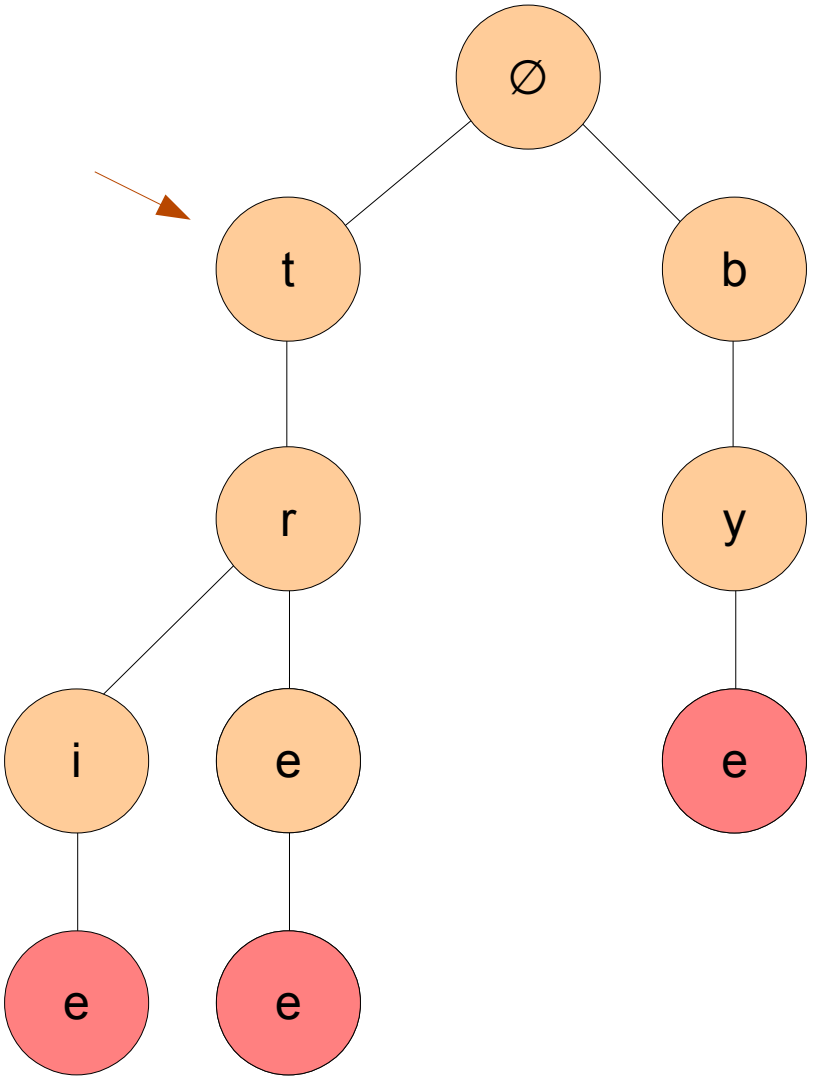

```
add “tree”
add “trie”
add “bye”
```

Tries: Παράδειγμα κατασκευής



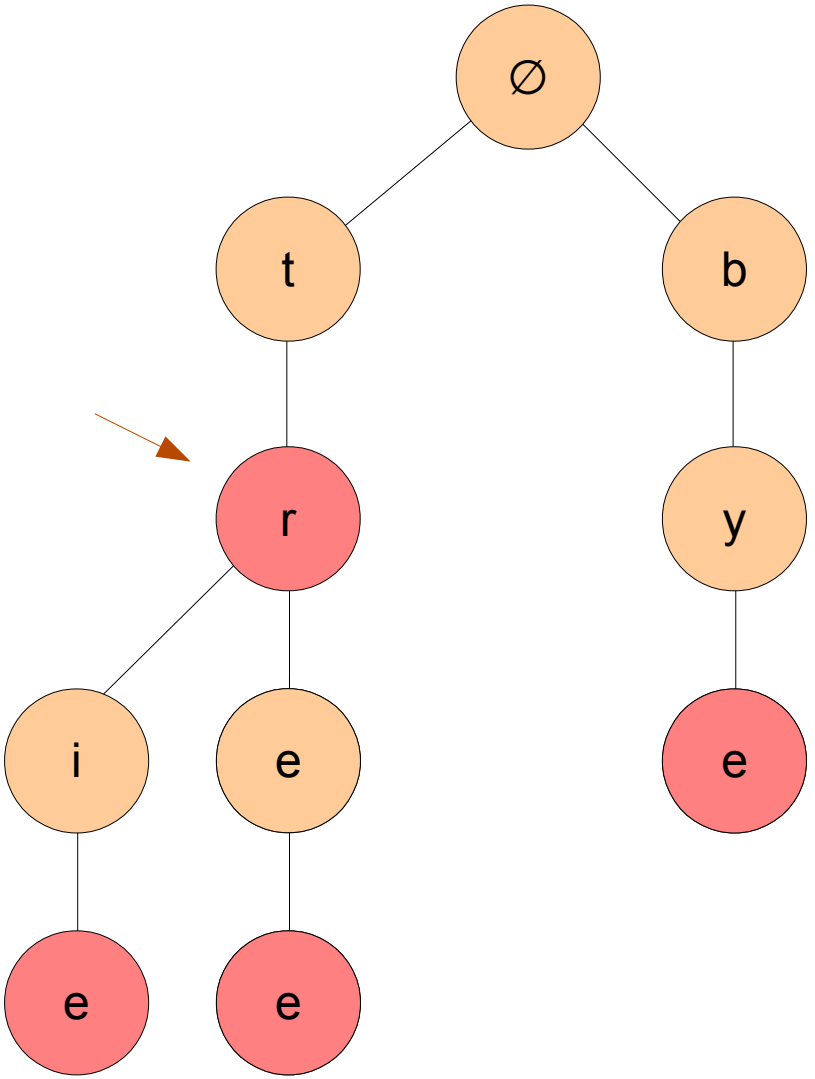

```
add “tree”
add “trie”
add “bye”
add “tr”
```

Tries: Παράδειγμα κατασκευής




```
add “tree”
add “trie”
add “bye”
add “tr”
```

Tries: Παράδειγμα κατασκευής




```
add "tree"
add "trie"
add "bye"
add "tr"
```

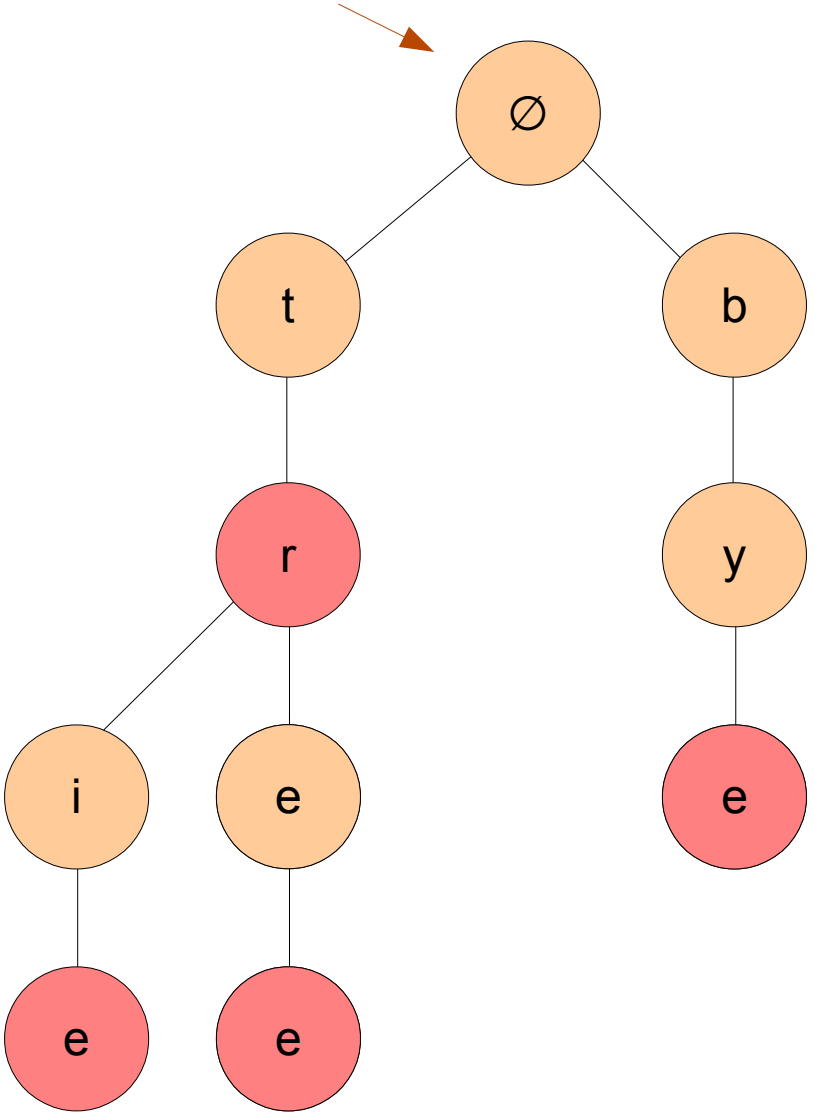
Tries: Υλοποίηση (the IOI way)

```
#define N 50000
struct node {
    int children[26];
    char isWord;
};
struct node Trie[N];
int trieNodeCount;

int initialize() {
    trieNodeCount = 1; /* προσθέτουμε την “ψεύτικη” ρίζα */
}
```

Tries: Υλοποίηση της add

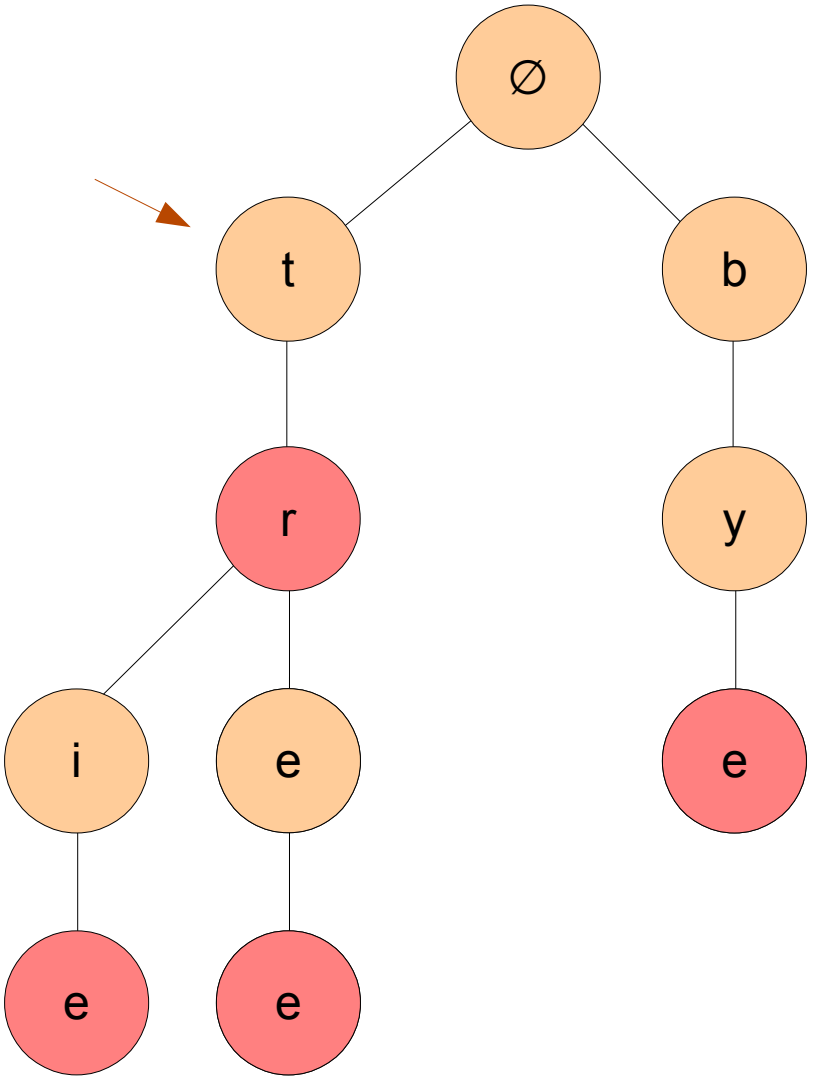
```
int add(char *word, int length) {
   int i, nextNode, curNode = 1; /* τοποθετούμε το βέλος στην ρίζα */

   for (i=0; i<length; i++) {
      nextNode = Trie[curNode].children[ word[i] - 'a' ];

      if ( nextNode == 0 ) { /* αν δεν υπάρχει ο κόμβος στο
                                trie, τότε τον δημιουργούμε */

         trieNodeCount++;
         Trie[curNode].children[ word[i] - 'a' ] = trieNodeCount;
         curNode = trieNodeCount;
      }
      else {
         curNode = nextNode;
      }
   }
   Trie[curNode].isWord = 1;
}
```

Tries: Παραδείγματα



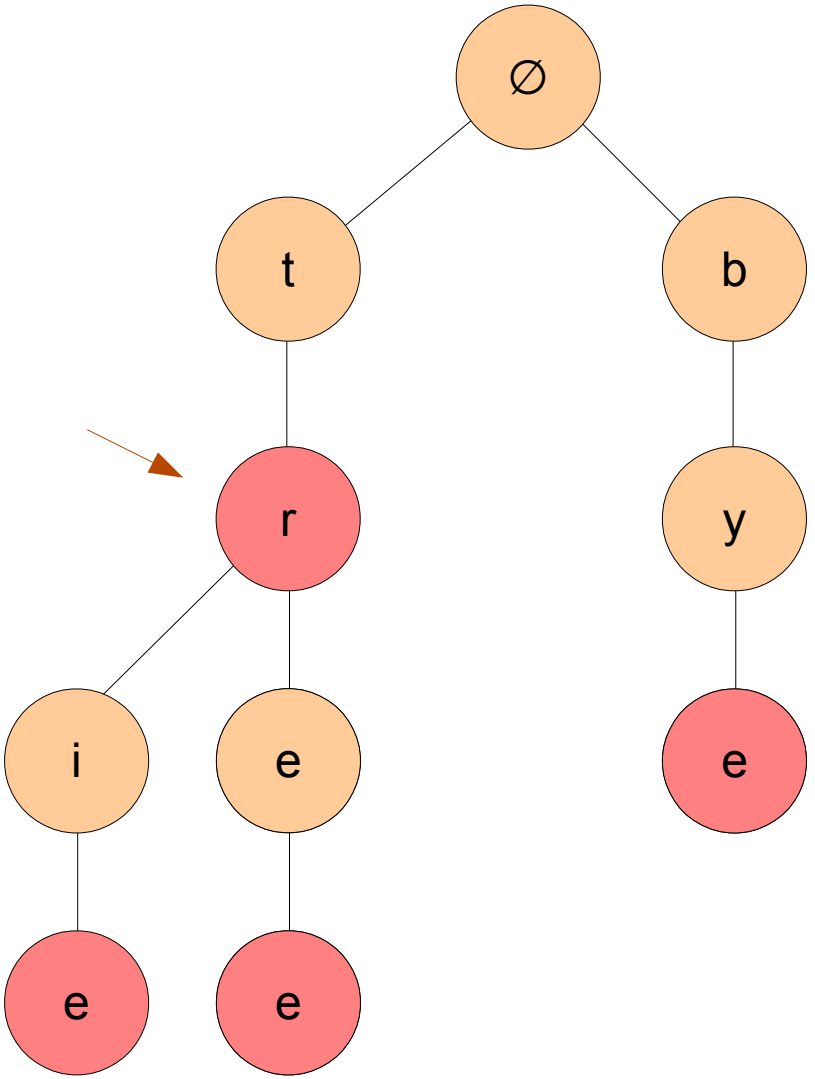

```
add “tree”
add “trie”
add “bye”
add “tr”
remove “trie”
```

Tries: Παραδείγματα



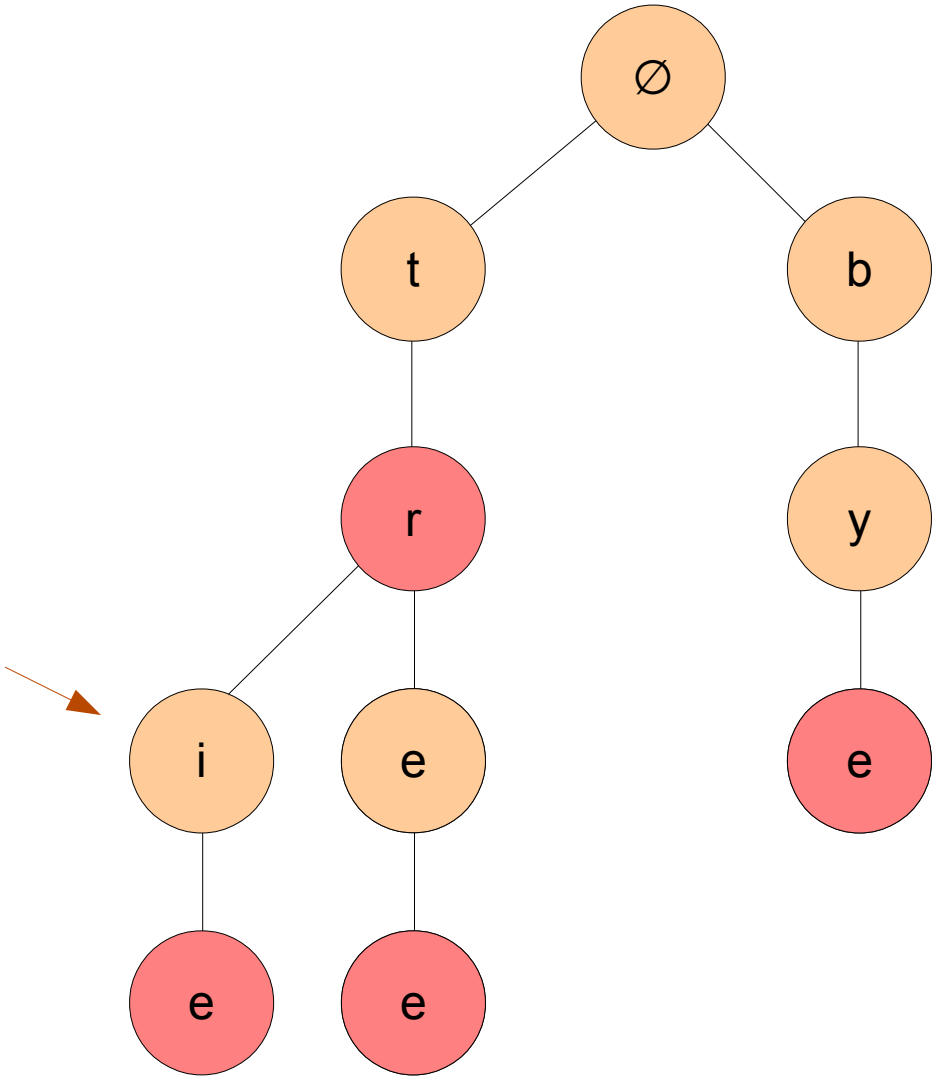

```
add “tree”
add “trie”
add “bye”
add “tr”
remove “trie”
```

Tries: Παραδείγματα



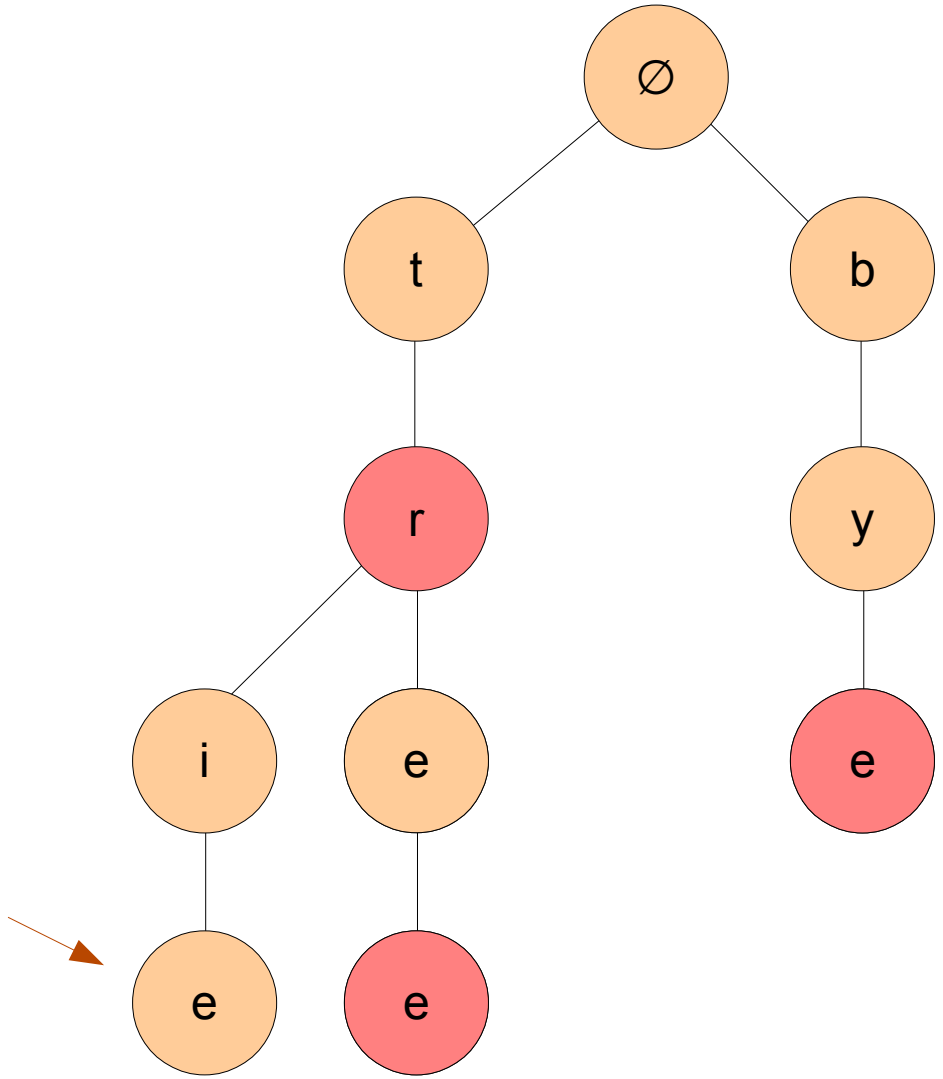

```
add “tree”
add “trie”
add “bye”
add “tr”
remove “trie”
```

Tries: Παραδείγματα




```
add “tree”
add “trie”
add “bye”
add “tr”
remove “trie”
```

Tries: Παραδείγματα



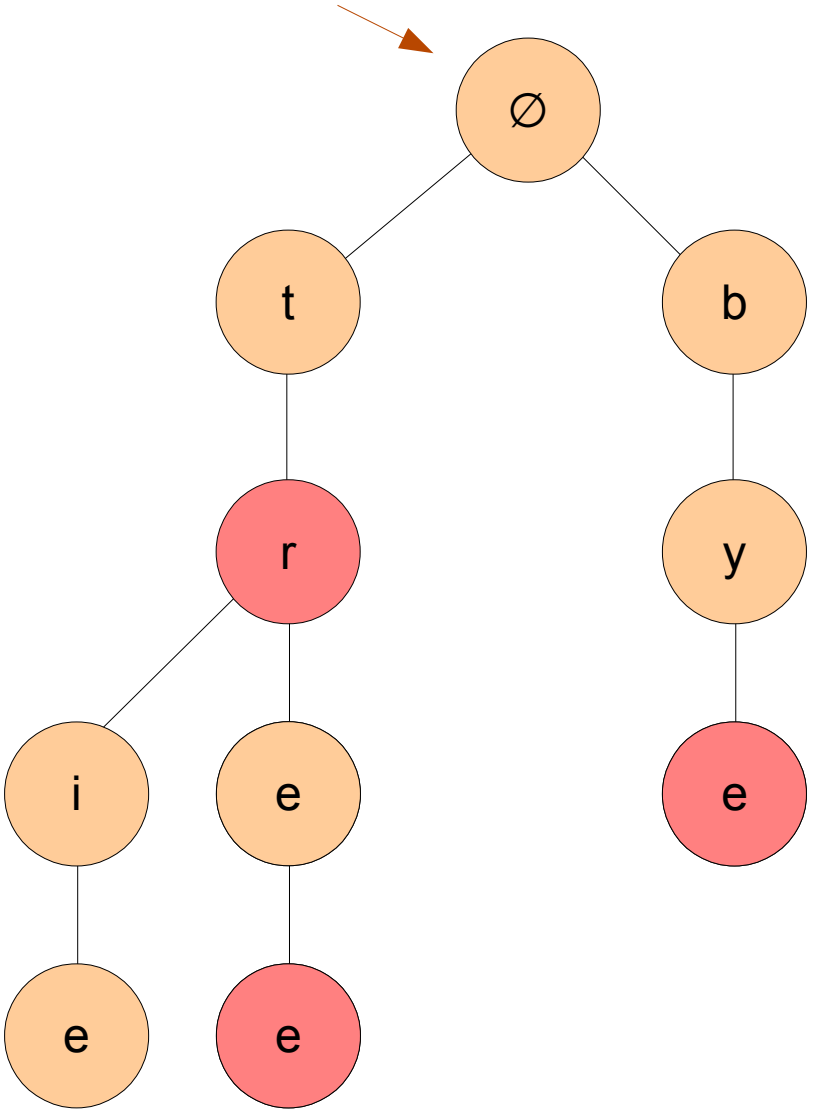

```
add “tree”
add “trie”
add “bye”
add “tr”
remove “trie”
```

αλλάζουμε το isWord σε 0

Tries: Υλοποίηση της remove

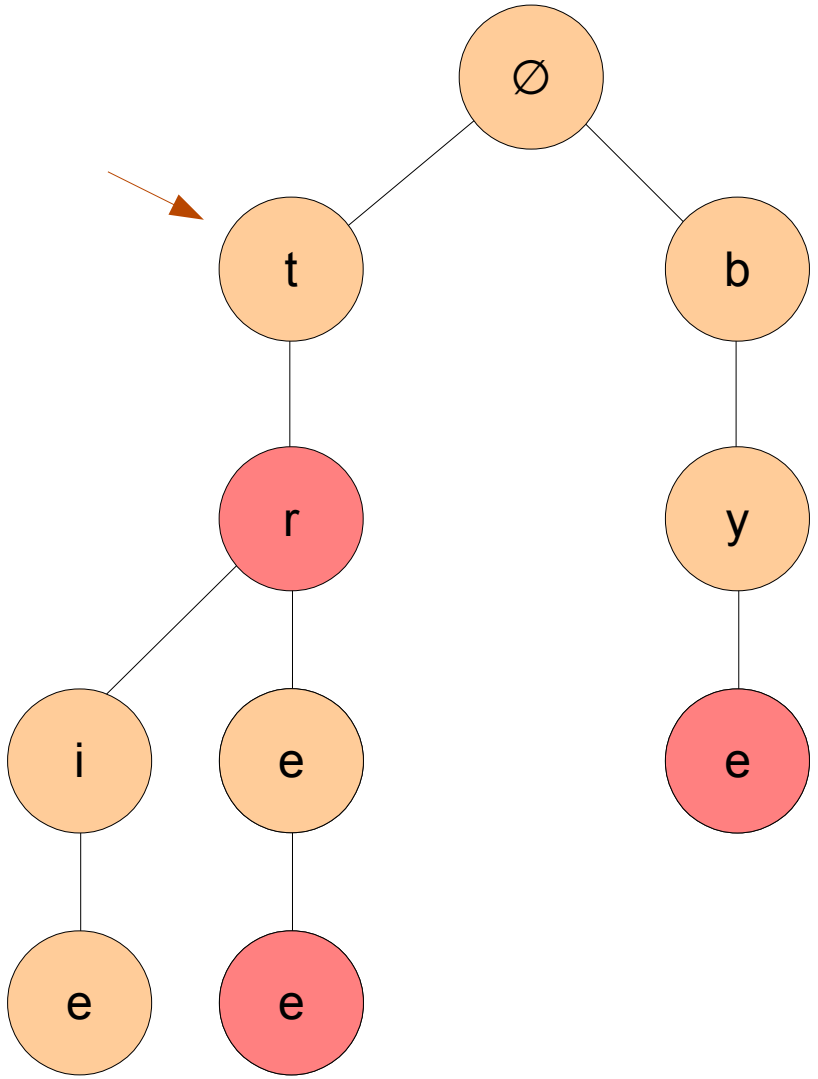
```
int remove(char word, int length) { /* ακολουθούμε το μονοπάτι μέχρι τον
                                       κόμβο του τελευταίου γράμματος και στη
                                       συνέχεια αλλάζουμε το isWord σε 0 */
    int i, curNode = 1; /* τοποθετούμε το βέλος στην ρίζα του trie */
    for (i=0; i<length; i++) {
            curNode = Trie[curNode].children[ word[i] - 'a' ];

            assert(curNode != 0); /* υποθέτουμε ότι η λέξη που θέλουμε να
                                     διαγράψουμε υπάρχει πάντα */
    }
    Trie[curNode].isWord = 0;
}
```

Tries: Παραδείγματα



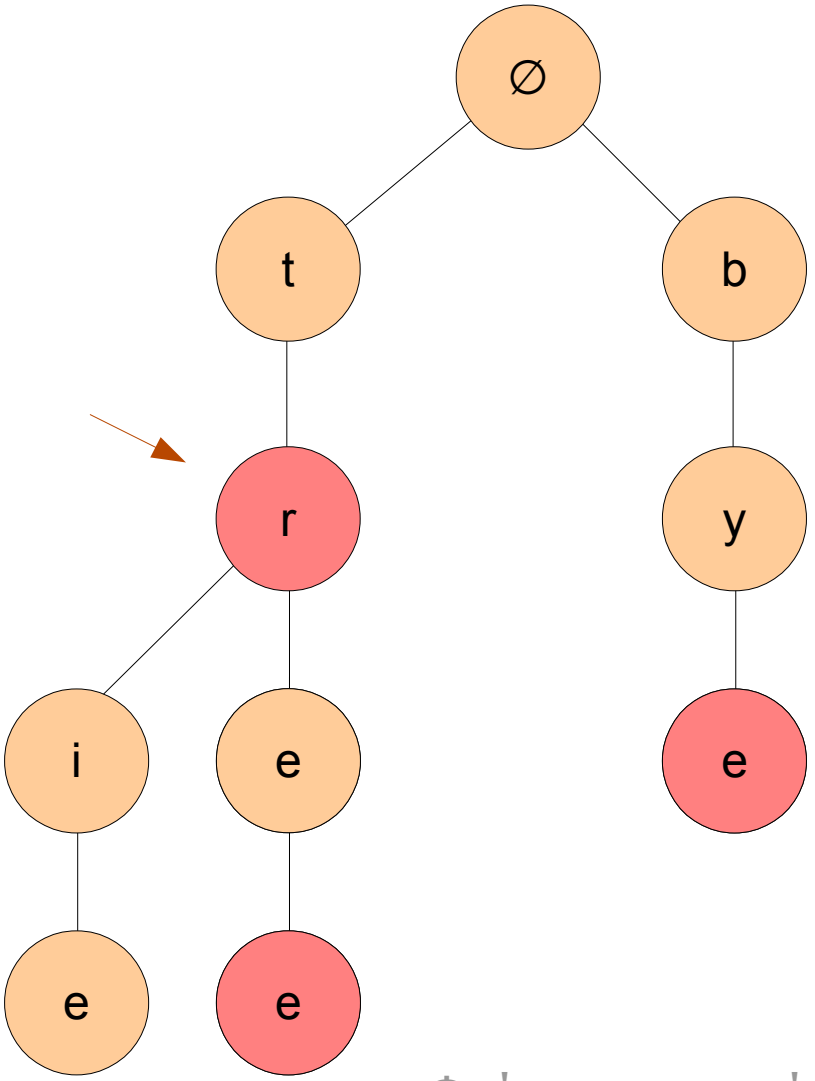

```
add “tree”
add “trie”
add “bye”
add “tr”
remove “trie”
check “tr”
```

Tries: Παραδείγματα




```
add “tree”
add “trie”
add “bye”
add “tr”
remove “trie”
check “tr”
```

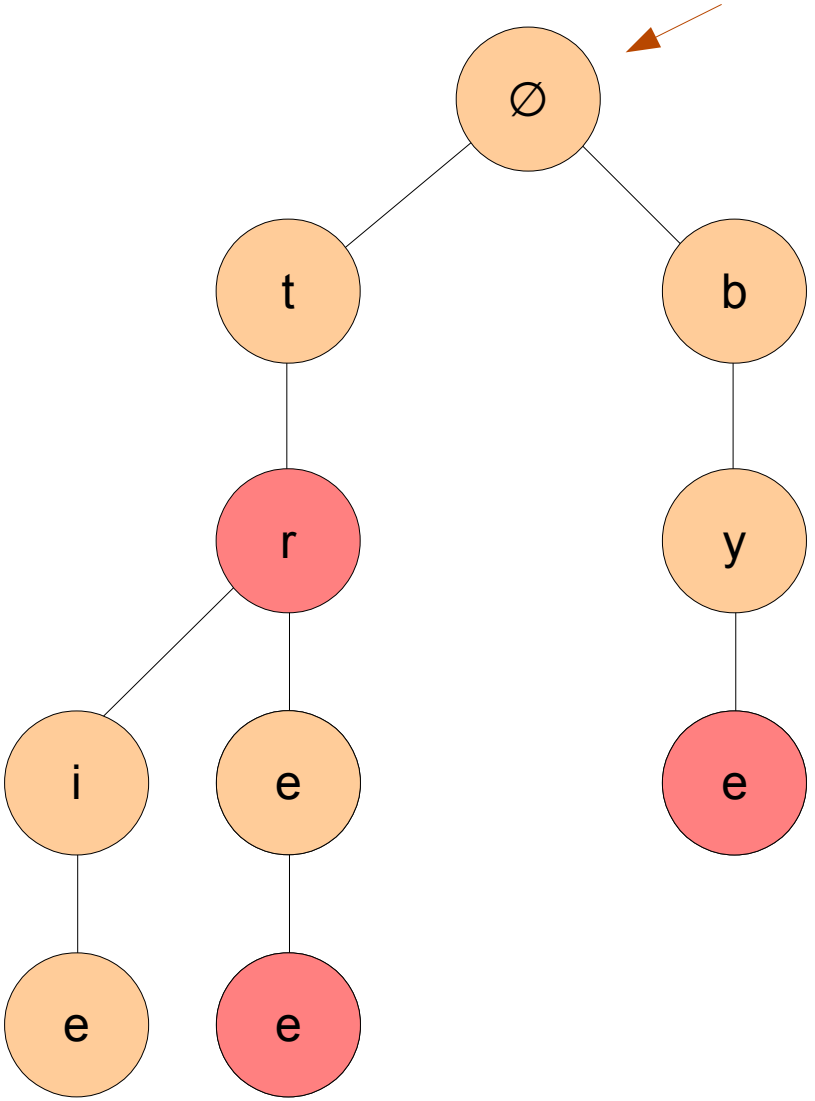
Tries: Παραδείγματα




```
add “tree”
add “trie”
add “bye”
add “tr”
remove “trie”
check “tr” true
```

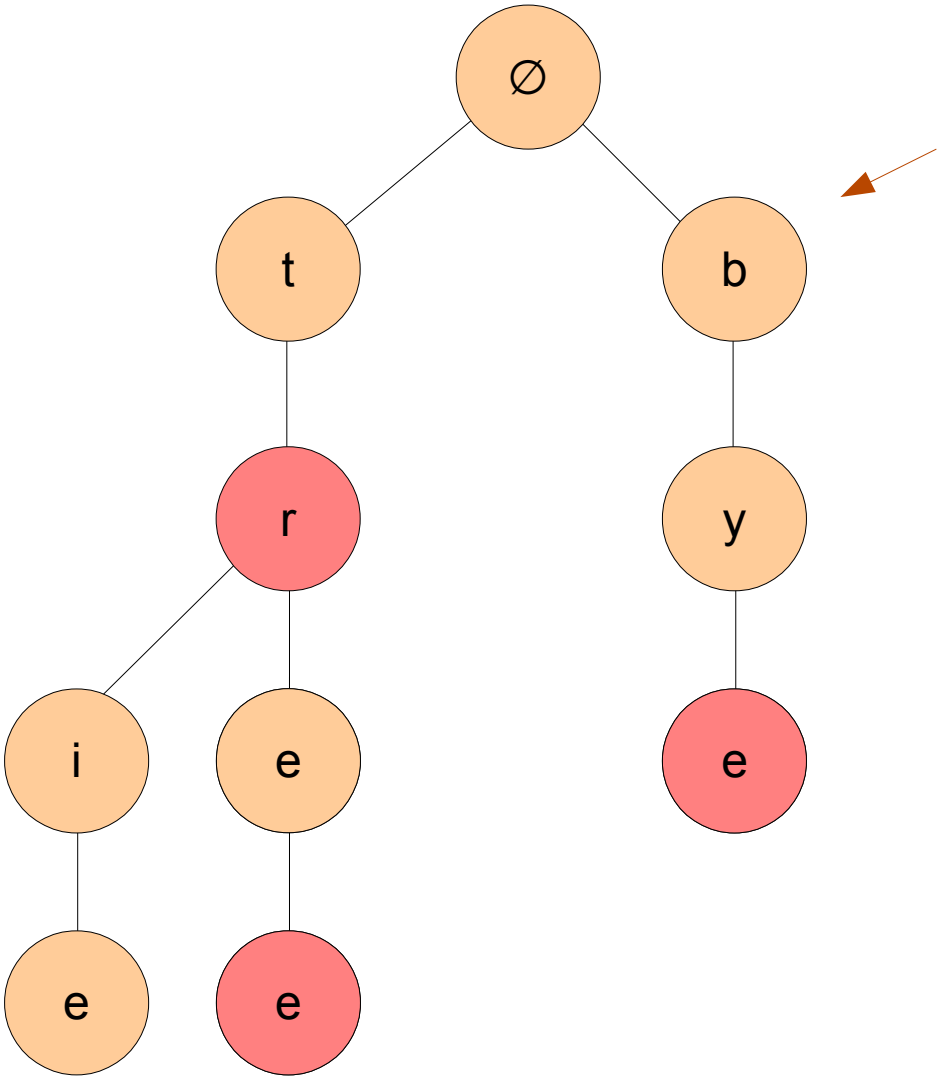
Φτάσαμε στο κόμβο του τελεύταιου γράμματος, ελέγχουμε αν το isWord αυτού του κόμβου είναι 1

Tries: Παραδείγματα



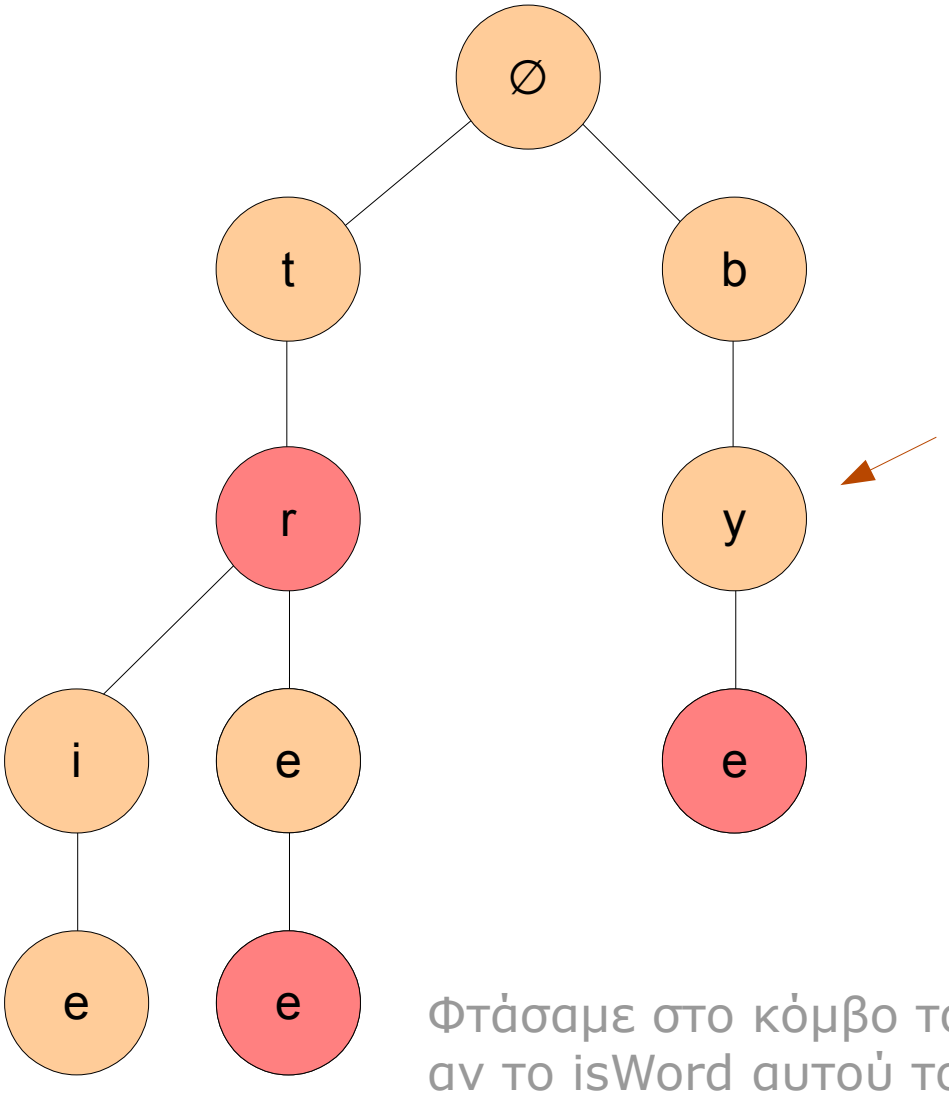

```
add “tree”
add “trie”
add “bye”
add “tr”
remove “trie”
check “tr” true
check “by”
```

Tries: Παραδείγματα




```
add “tree”
add “trie”
add “bye”
add “tr”
remove “trie”
check “tr” true
check “by”
```

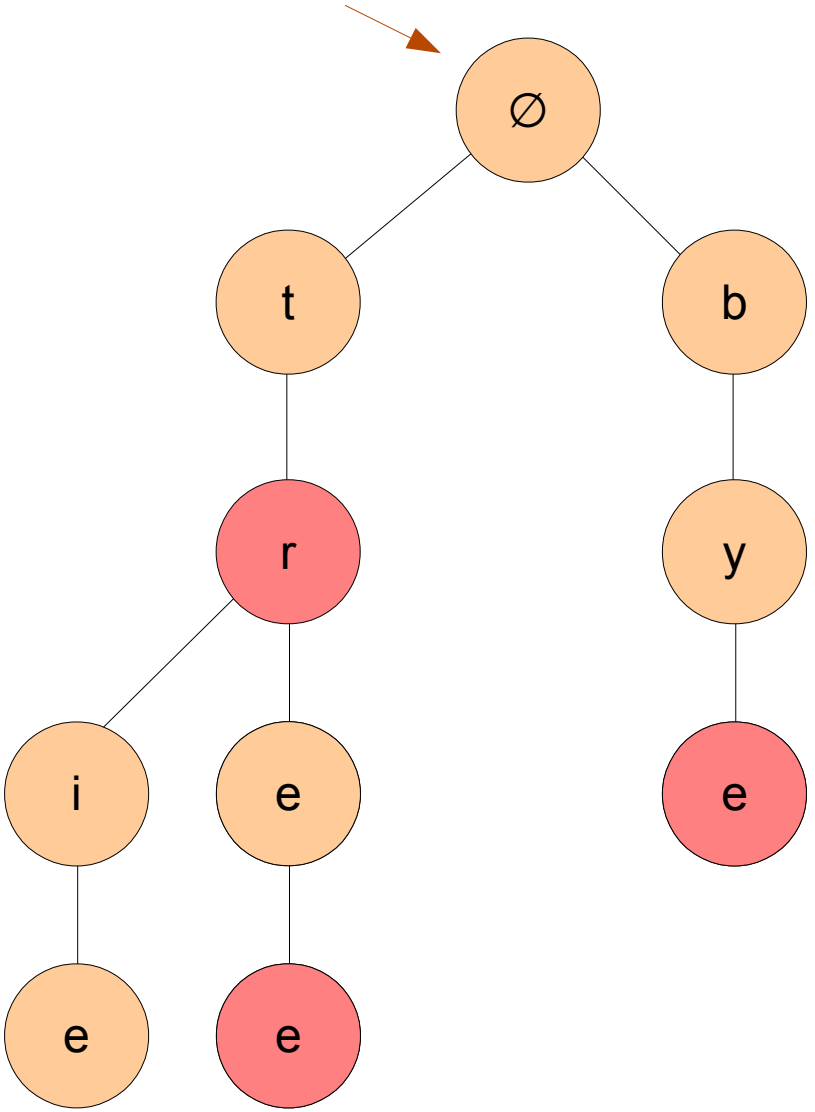
Tries: Παραδείγματα




```
add “tree”
add “trie”
add “bye”
add “tr”
remove “trie”
check “tr” true
check “by” false
```

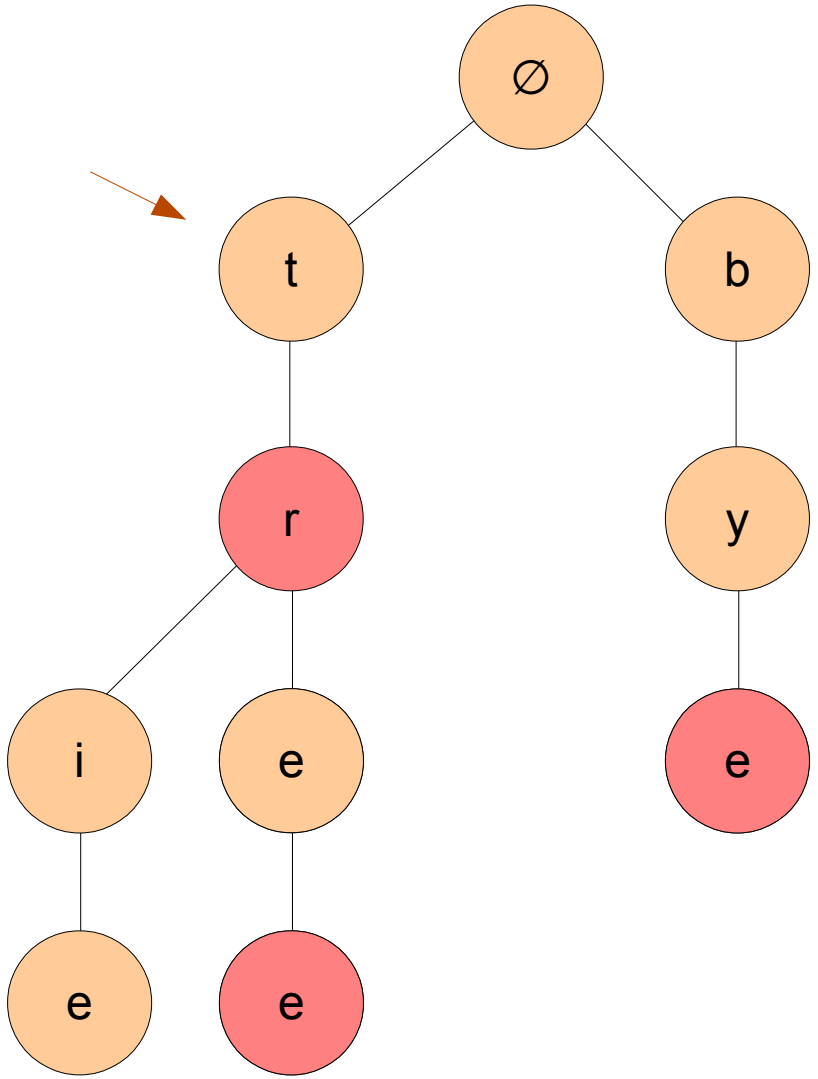
Φτάσαμε στο κόμβο του τελεύταιου γράμματος, ελέγχουμε αν το isWord αυτού του κόμβου είναι 1

Tries: Παραδείγματα



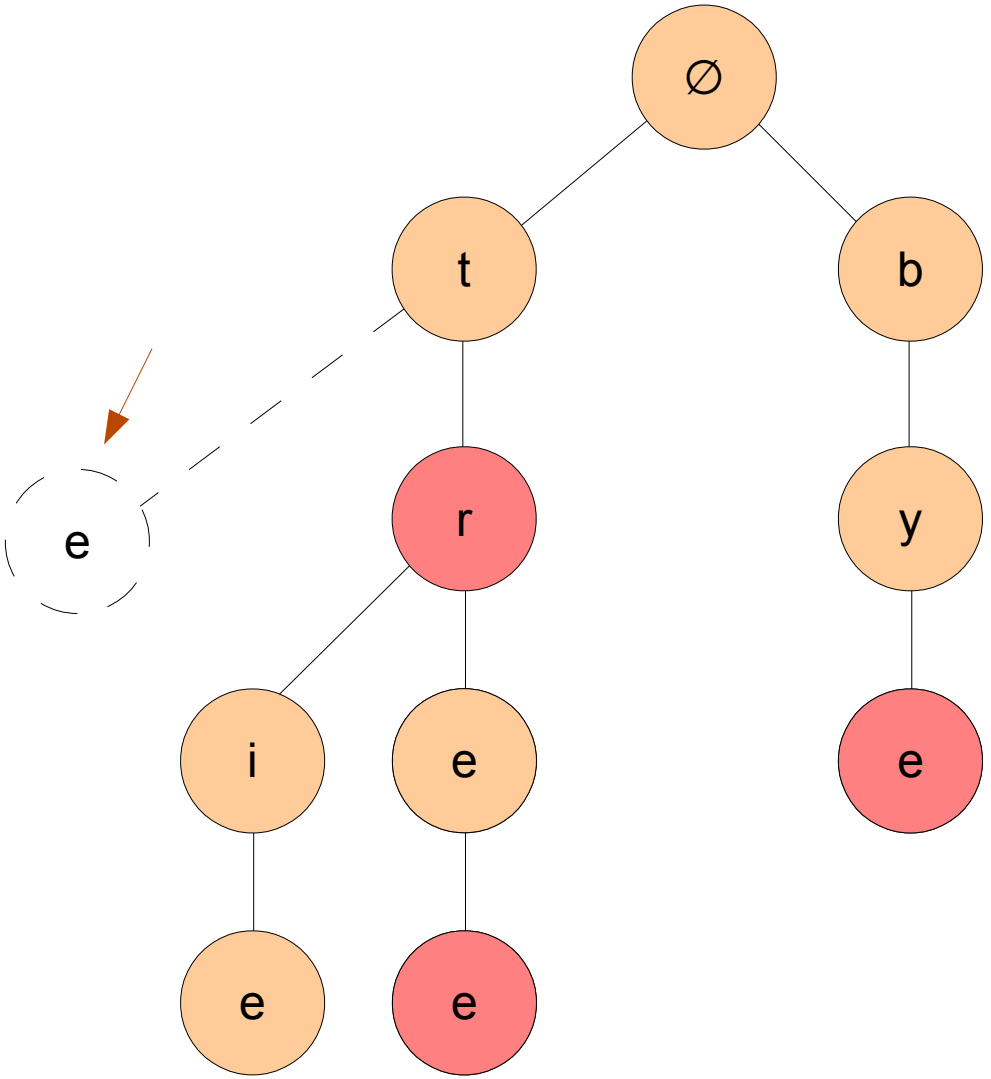

```
add “tree”
add “trie”
add “bye”
add “tr”
remove “trie”
check “tr” true
check “by” false
check “ten”
```

Tries: Παραδείγματα



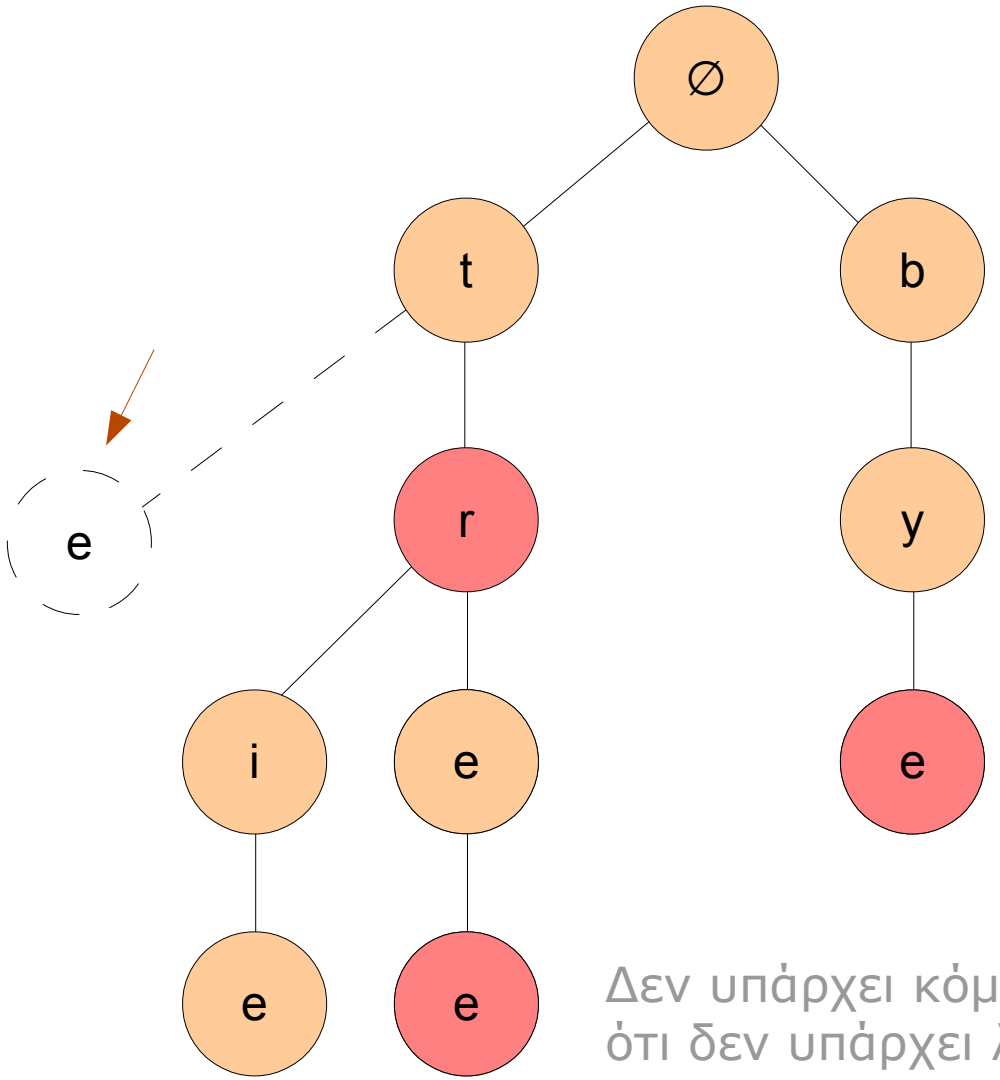

```
add “tree”
add “trie”
add “bye”
add “tr”
remove “trie”
check “tr” true
check “by” false
check “ten”
```

Tries: Παραδείγματα




```
add “tree”
add “trie”
add “bye”
add “tr”
remove “trie”
check “tr” true
check “by” false
check “ten”
```

Tries: Παραδείγματα




```
add “tree”
add “trie”
add “bye”
add “tr”
remove “trie”
check “tr” true
check “by” false
check “ten” false
```

Δεν υπάρχει κόμβος προς το γράμμα “e”. Αυτό σημαίνει ότι δεν υπάρχει λέξη που να αρχίζει από “te”, άρα δεν υπάρχει και η συμβολοσειρά που ψάχνουμε. **Πρέπει να λάβουμε υπόψιν μας και αυτή την περίπτωση.**

Tries: Υλοποίηση της check

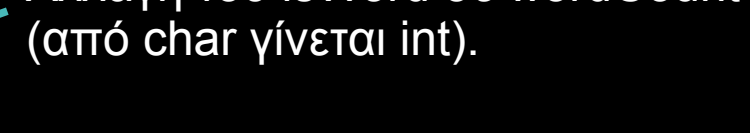
```
bool check(char word, int length) {
    int i, curNode = 1; /* τοποθετούμε το βέλος στην ρίζα του trie */
    for (i=0; i<length; i++) {
            curNode = Trie[curNode].children[ word[i] - 'a' ];

            if ( curNode == 0 ) { /* αν δεν υπάρχει ο κόμβος στο trie
                    τότε δεν υπάρχει και η λέξη που ψάχνουμε */
                return false;
            }
    }
    return (Trie[curNode].isWord == 1 ? true : false);
}
```

Tries: Παράδειγμα Επέκτασης (1)

- Δυνατότητα προσθήκης της ίδιας λέξης περισσότερες από μια φορές και
- αλλαγή της check ωστέ να επιστρέφει πόσες φορές υπάρχει η λέξη στο trie (0 αν δεν υπάρχει).

Tries: Υλοποίηση με wordCount

```
#define N 50000

struct node {
    int children[26];
    int wordCount;
};

struct node Trie[N];
int trieNodeCount;


int initialize() {
    trieNodeCount = 1; /* προσθέτουμε την “ψεύτικη” ρίζα */
}
```



Tries: Υλοποίηση της *add*

```
int add(char *word, int length) {
   int i, nextNode, curNode = 1; /* τοποθετούμε το βέλος στην ρίζα */

   for (i=0; i<length; i++) {
      nextNode = Trie[curNode].children[ word[i] - 'a' ];

      if ( nextNode == 0 ) { /* αν δεν υπάρχει ο κόμβος στο
                                trie, τότε τον δημιουργούμε */

         trieNodeCount++;
         Trie[curNode].children[ word[i] - 'a' ] = trieNodeCount;
         curNode = trieNodeCount;
      }
      else {
         curNode = nextNode;
      }
   }
   Trie[curNode].wordCount++;
}
```

Αυξάνουμε το πλήθος των συμβολοσειρών που καταλήγουν σε αυτόν τον κόμβο κατά 1.

Tries: Υλοποίηση της remove

```
int remove(char word, int length) { /* ακολουθούμε το μονοπάτι μέχρι τον
                                       κόμβο του τελευταίου γράμματος και στη
                                       συνέχεια αλλάζουμε το isWord σε 0 */

    int i, curNode = 1; /* τοποθετούμε το βέλος στην ρίζα του trie */

    for (i=0; i<length; i++) {

            curNode = Trie[curNode].children[ word[i] - 'a' ];


            assert(curNode != 0); /* υποθέτουμε ότι η λέξη που θέλουμε να
                                     διαγράψουμε υπάρχει πάντα */

    }

    Trie[curNode].wordCount--;

}
```

Μειώνουμε το πλήθος των συμβολοσειρών που καταλήγουν σε αυτόν τον κόμβο κατά 1.

Tries: Υλοποίηση της check

Η συνάρτηση επιστρέφει πλέον αριθμό.

```
int check(char word, int length) {

    int i, curNode = 1; /* τοποθετούμε το βέλος στην ρίζα του trie */

    for (i=0; i<length; i++) {

            curNode = Trie[curNode].children[ word[i] - 'a' ];


            if ( curNode == 0 ) { /* αν δεν υπάρχει ο κόμβος στο trie
                    τότε δεν υπάρχει και η λέξη που ψάχνουμε */

                return 0;
            }

    }

    return Trie[curNode].wordCount;

}
```

Δεν υπάρχει μονοπάτι, άρα 0 λέξεις

Επιστρέφουμε το πλήθος των λέξεων

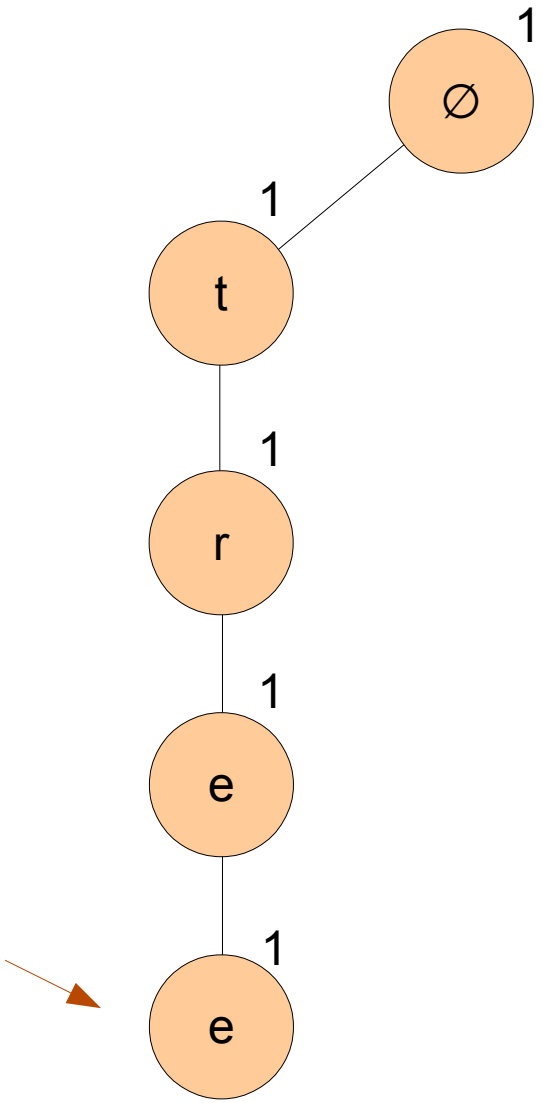
Tries: Παράδειγμα Επέκτασης (2)

- Δυνατότητα απάντησης σε ερωτήματα “πόσες λέξεις αρχίζουν από ....;”
- Δημιουργία συνάρτησης `prefixCount` για αυτό τον σκοπό.

Υλοποίηση:

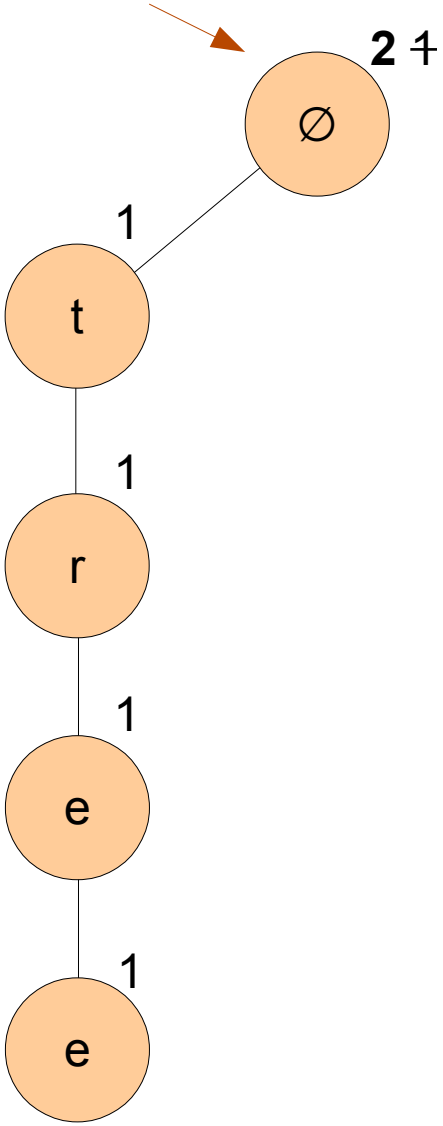
- Σε κάθε κόμβο του δέντρου θα αποθηκεύουμε έναν επιπλέον αριθμό: το πλήθος των λέξεων στις οποίες “ανήκει”.
- Θα πρέπει να συντηρούμε αυτά τα αθροίσματα σε κάθε πράξη `add` και `delete`.

Tries: Παράδειγμα κατασκευής



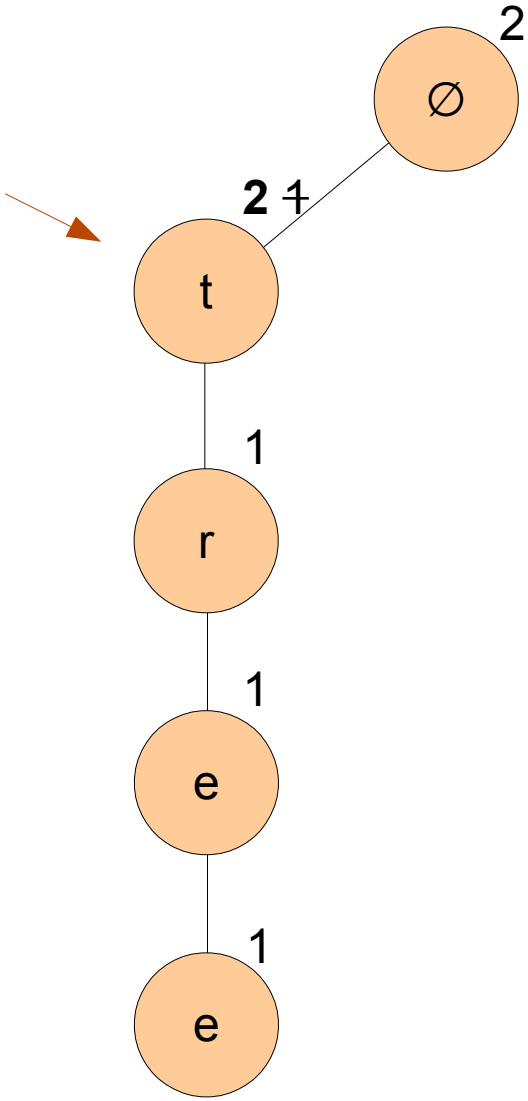
add “tree”

Tries: Παράδειγμα κατασκευής



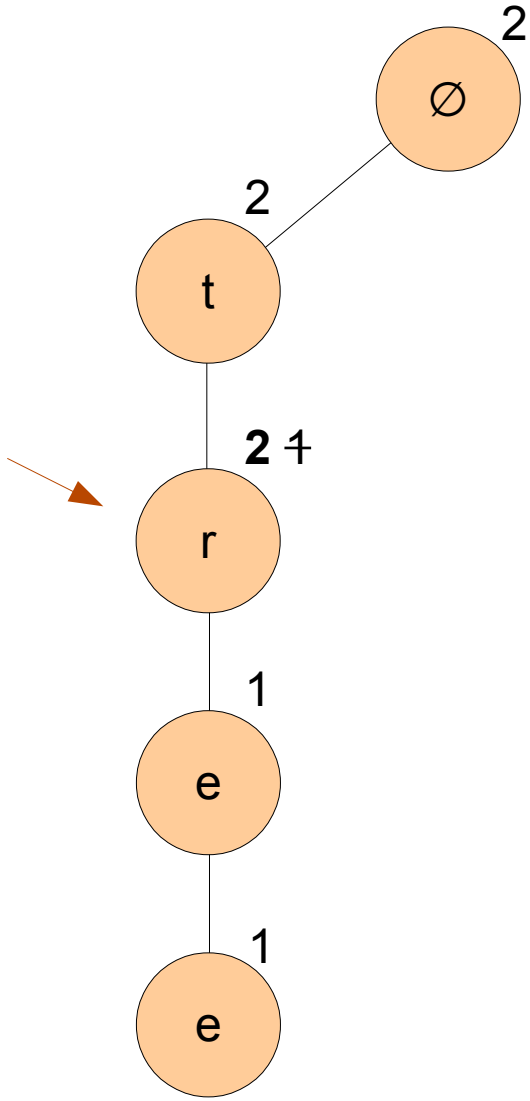

```
    add "tree"
→   add "trie"
```

Tries: Παράδειγμα κατασκευής



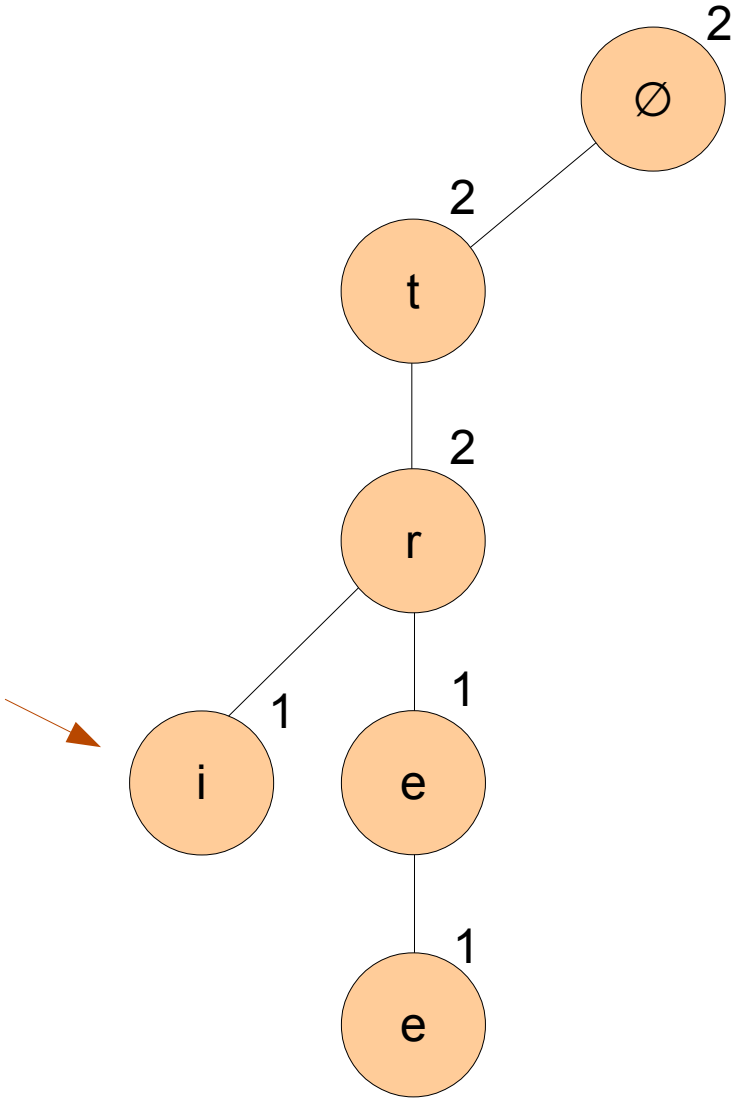

```
add “tree”
add “trie”
```

Tries: Παράδειγμα κατασκευής



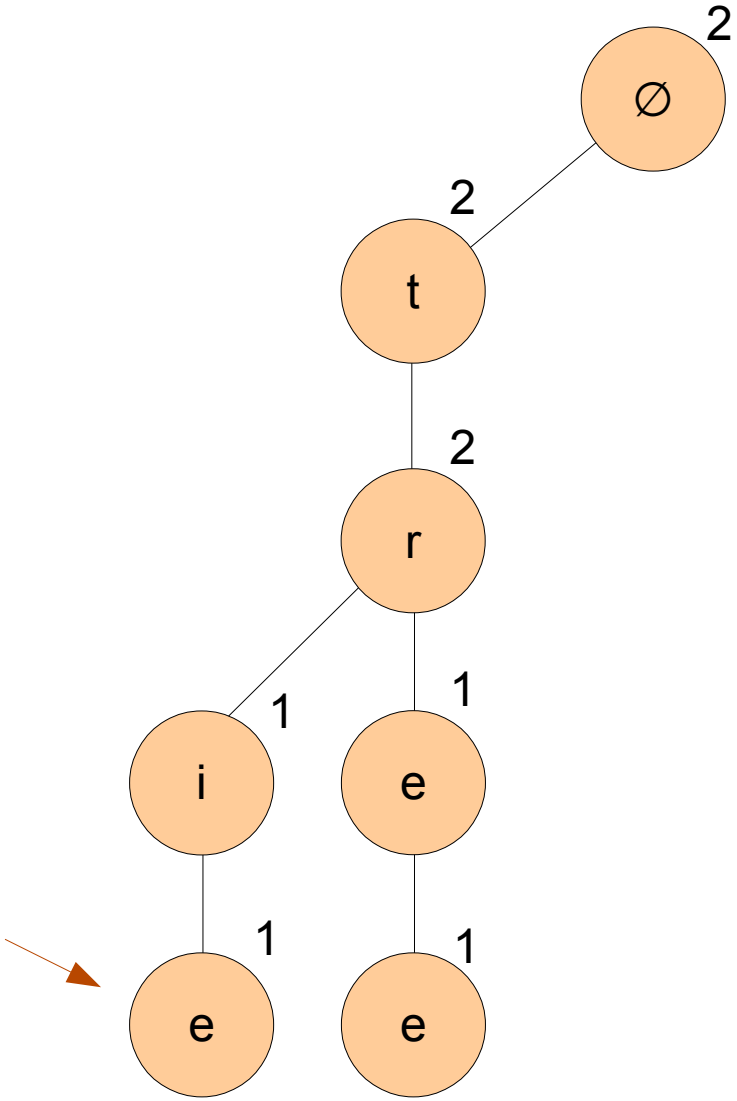

```
    add "tree"
→   add "trie"
```

Tries: Παράδειγμα κατασκευής



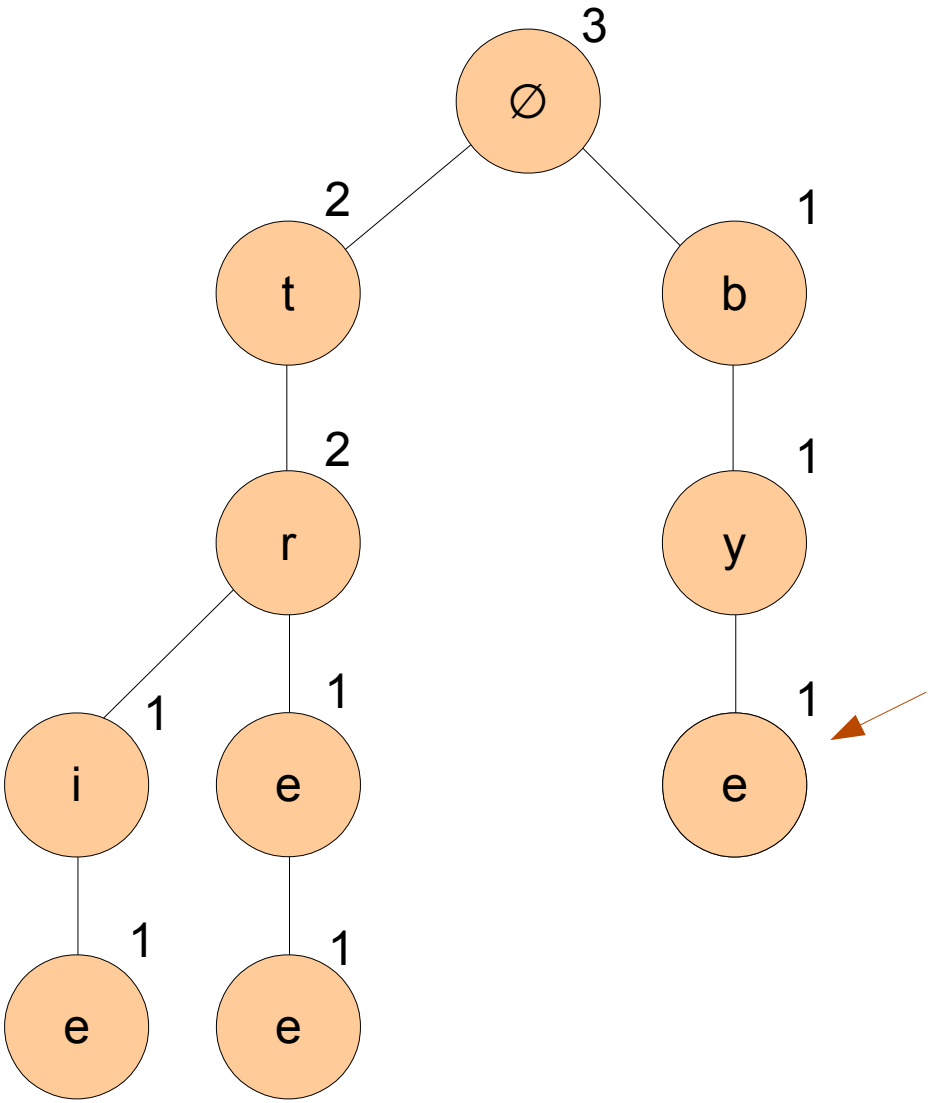
add “tree”
add “trie”

Tries: Παράδειγμα κατασκευής



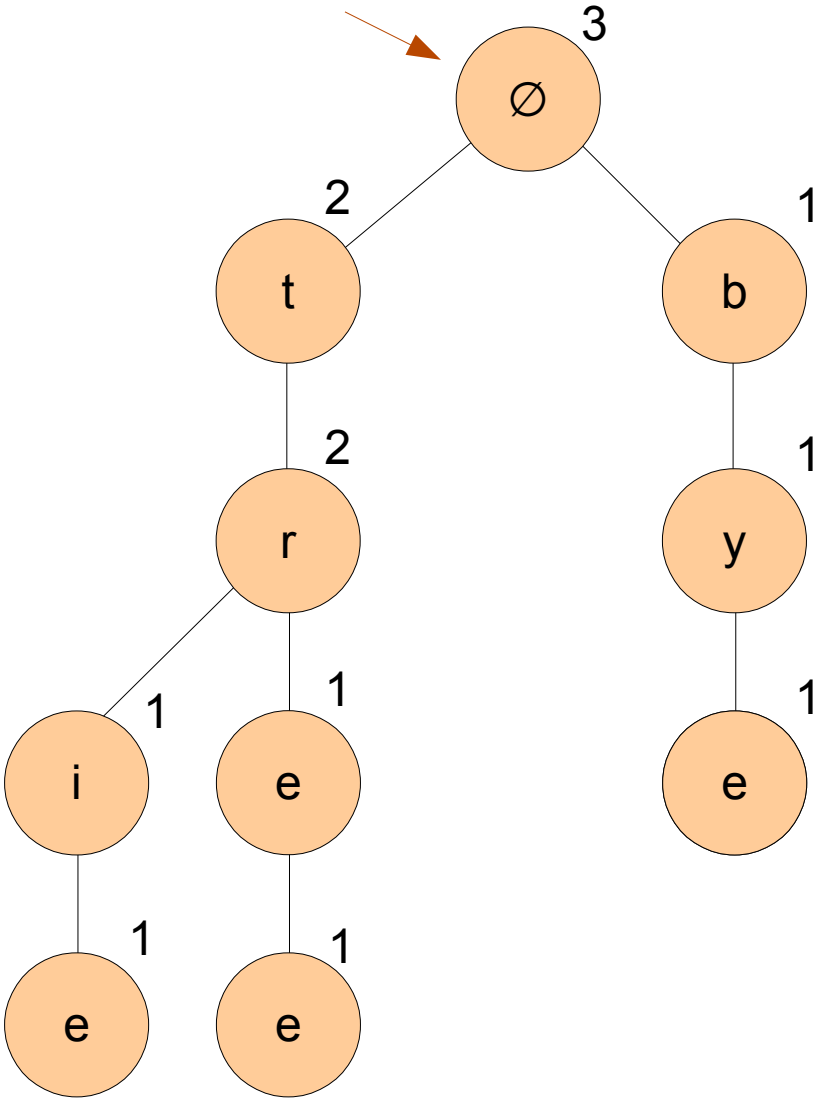

```
    add “tree”
→   add “trie”
```

Tries: Παράδειγμα κατασκευής



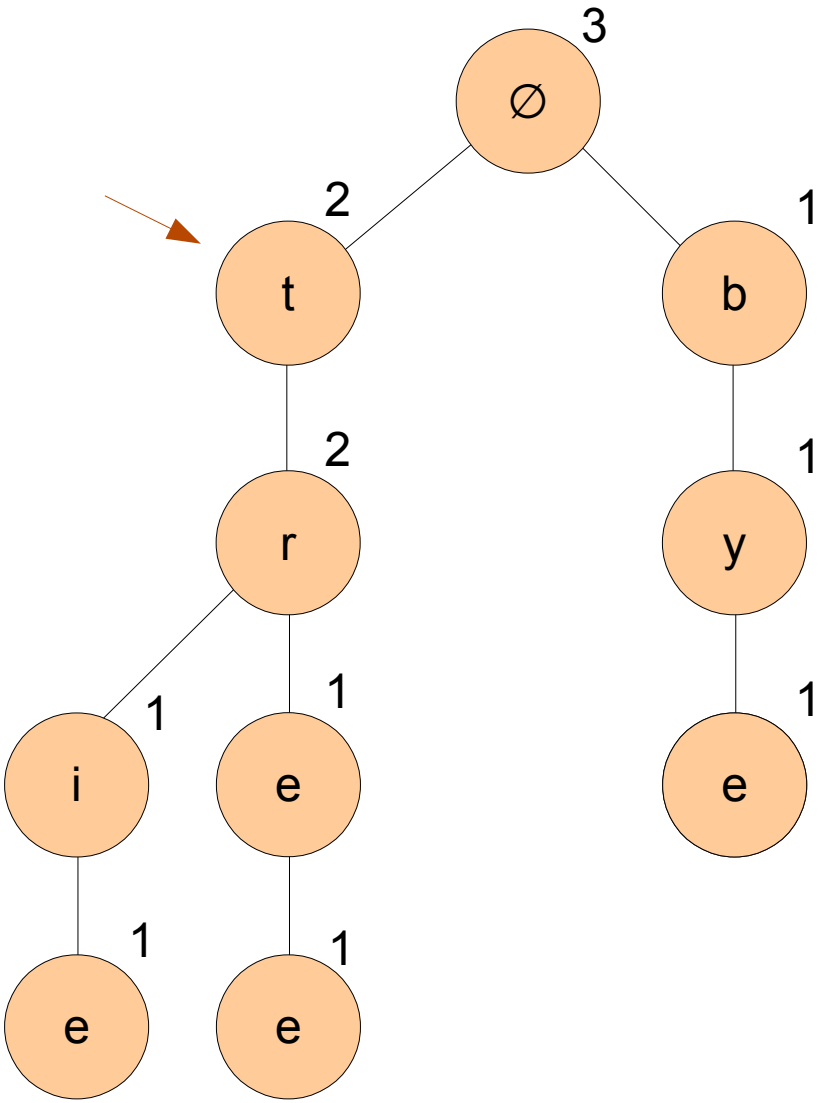

```
add “tree”
add “trie”
add “bye”
```

Tries: Παράδειγμα κατασκευής



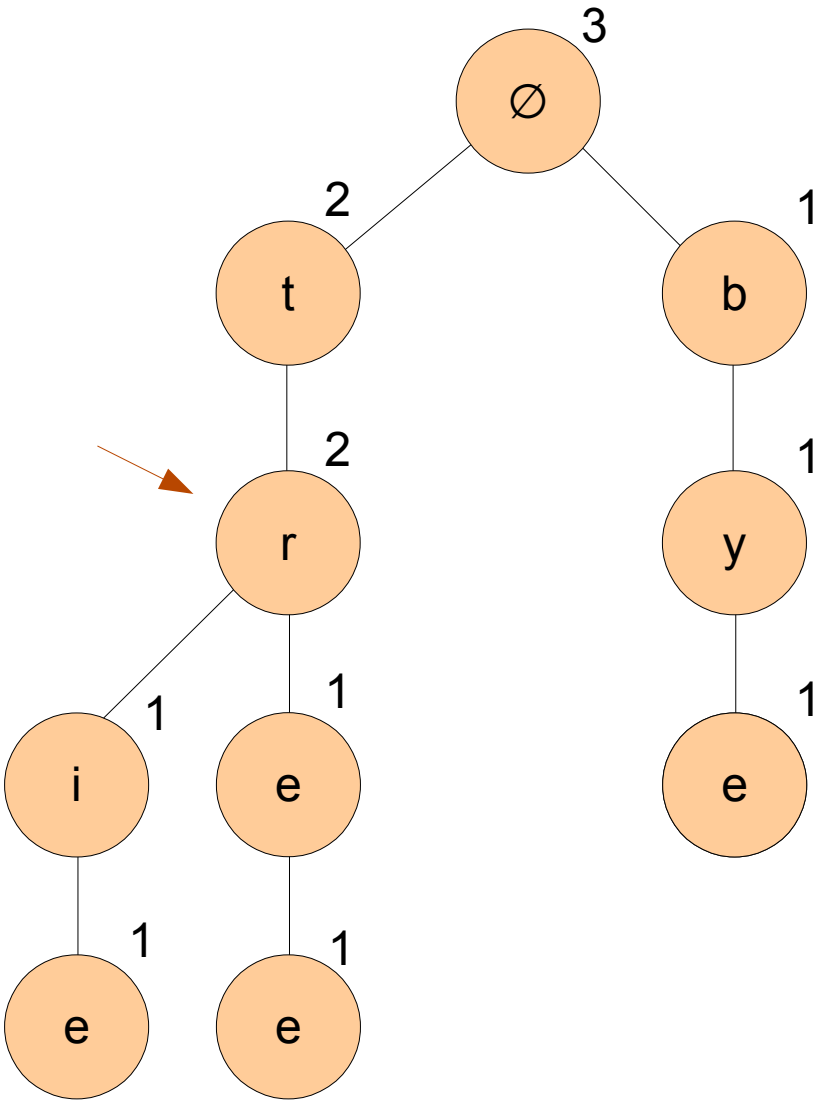

```
add “tree”
add “trie”
add “bye”
pcount “tr”
```

Tries: Παράδειγμα κατασκευής



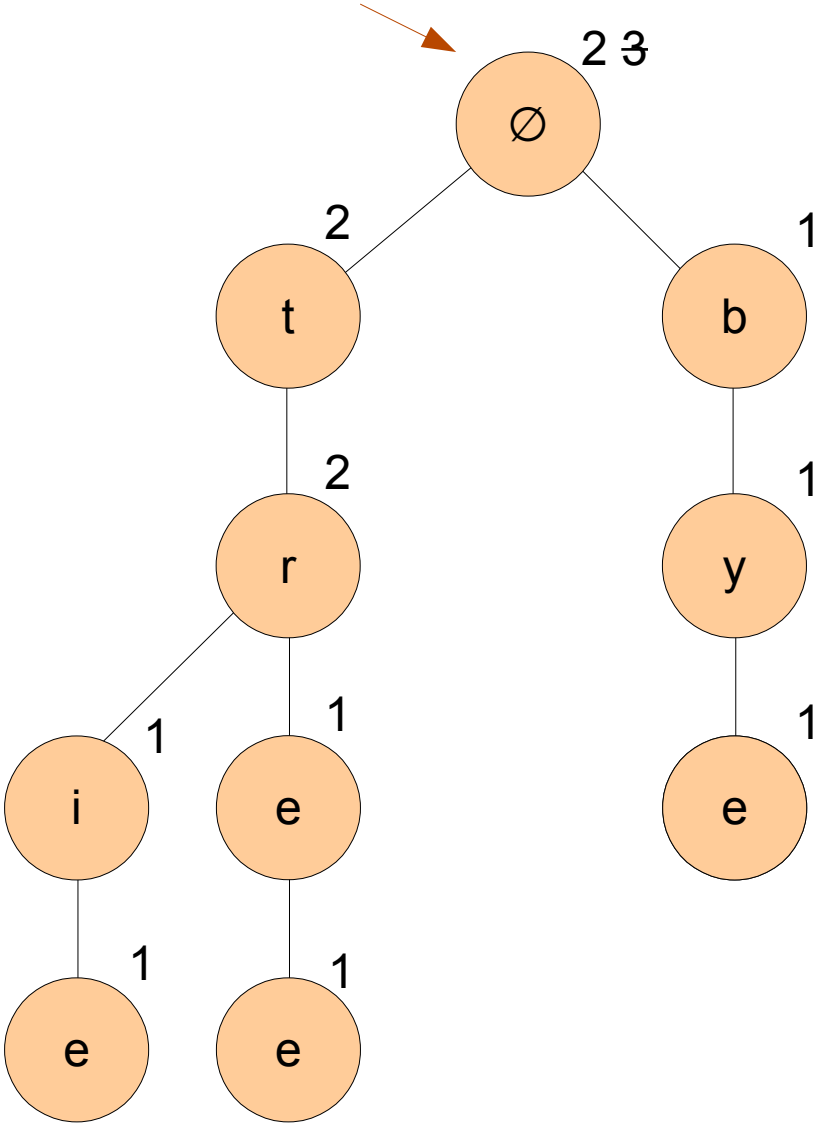

```
add “tree”
add “trie”
add “bye”
pcount “tr”
```

Tries: Παράδειγμα κατασκευής



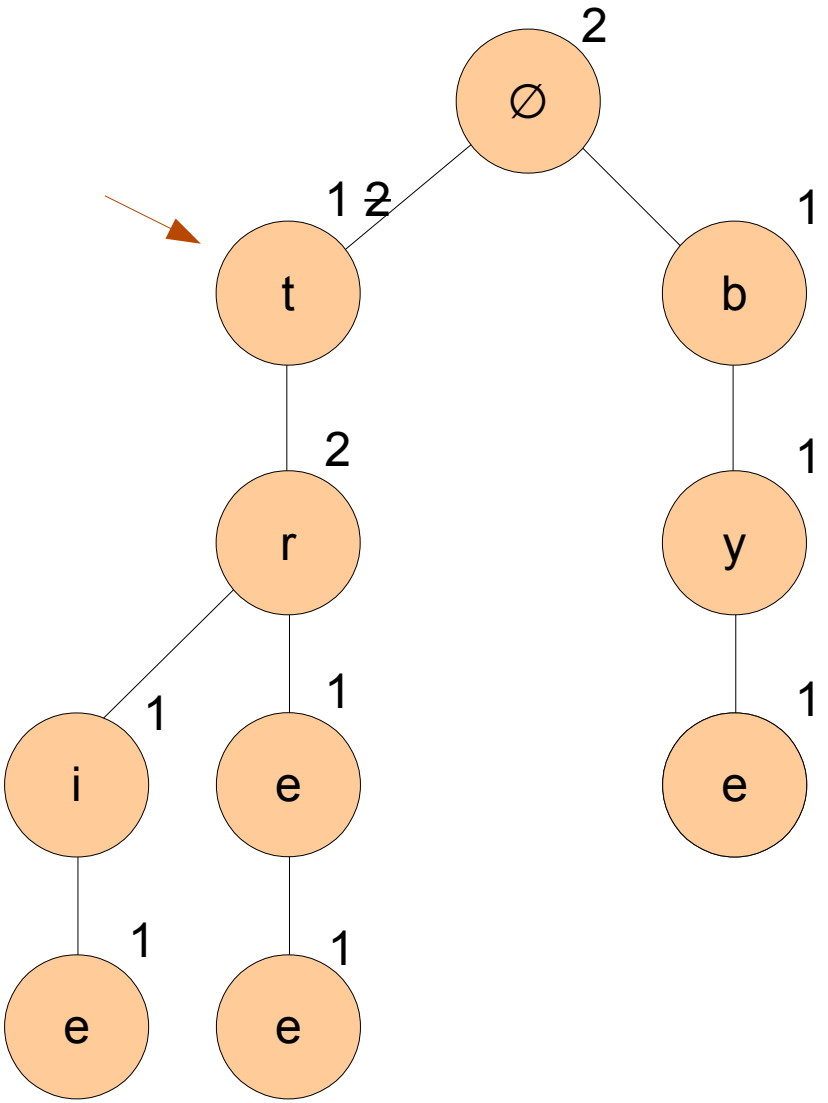

```
add “tree”
add “trie”
add “bye”
pcount “tr” 2
```

Tries: Παράδειγμα κατασκευής



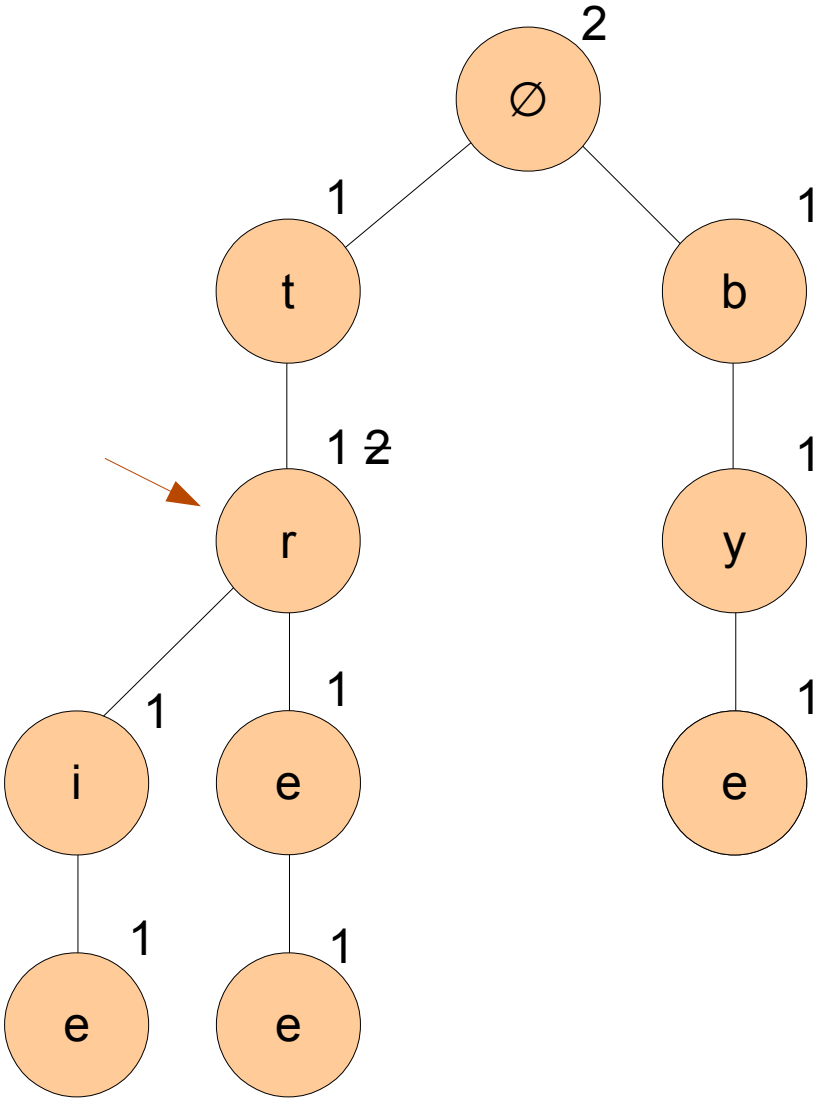

```
add “tree”
add “trie”
add “bye”
pcount “tr” 2
remove “trie”
```

Tries: Παράδειγμα κατασκευής



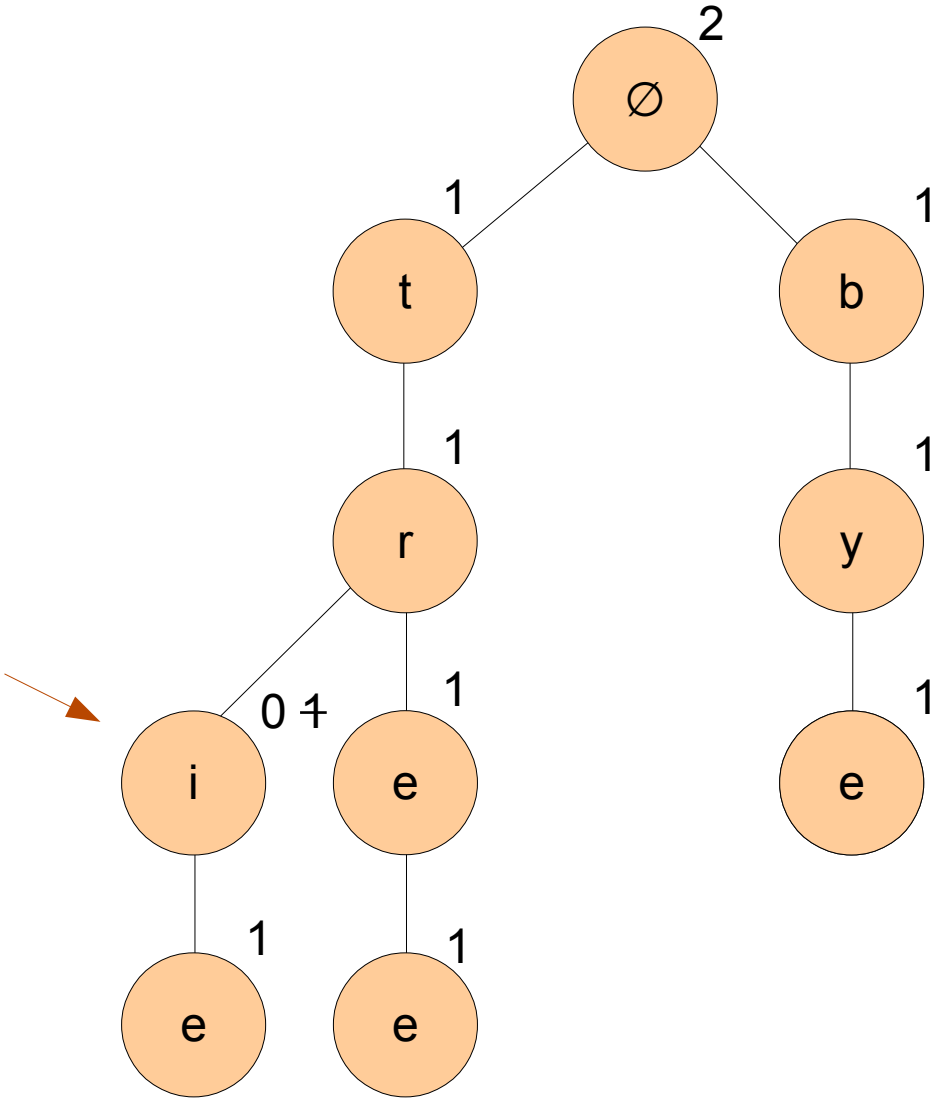

```
add “tree”
add “trie”
add “bye”
pcount “tr” 2
remove “trie”
```

Tries: Παράδειγμα κατασκευής



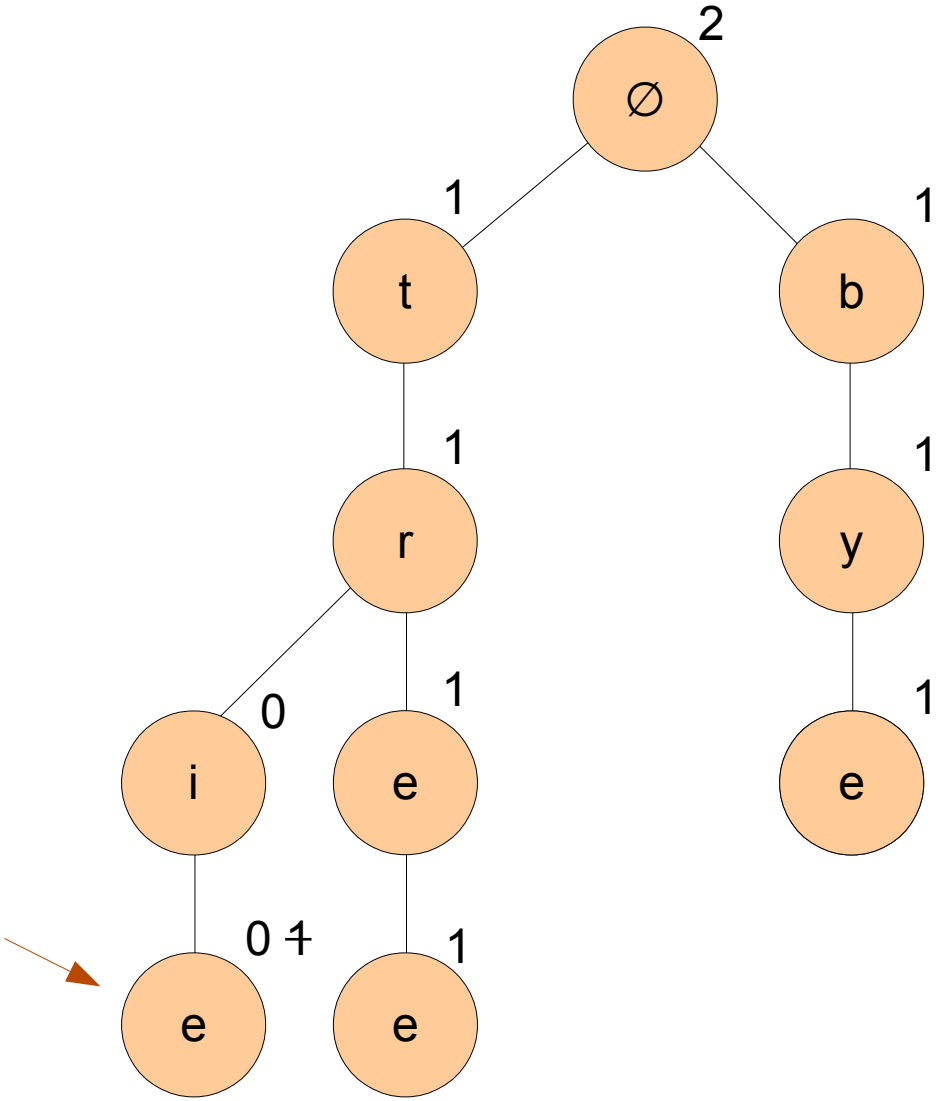

```
add “tree”
add “trie”
add “bye”
pcount “tr” 2
remove “trie”
```

Tries: Παράδειγμα κατασκευής




```
add “tree”
add “trie”
add “bye”
pcount “tr” 2
remove “trie”
```

Tries: Παράδειγμα κατασκευής




```
add “tree”
add “trie”
add “bye”
pcount “tr” 2
remove “trie”
```

Tries: Υλοποίηση για μέτρηση προθεμάτων

```
#define N 50000

struct node {
    int children[26];
    int prefixes;
};

struct node Trie[N];
int trieNodeCount;


int initialize() {
    trieNodeCount = 1; /* προσθέτουμε την “ψεύτικη” ρίζα */
}
```

Μεταβλητή που αποθηκεύει το πλήθος των προθεμάτων που αντιστοιχούν σε αυτόν τον κόμβο

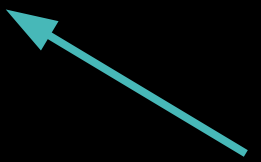
Tries: Υλοποίηση της add

```
int add(char *word, int length) {
   int i, nextNode, curNode = 1; /* τοποθετούμε το βέλος στην ρίζα */

   for (i=0; i<length; i++) {
      nextNode = Trie[curNode].children[ word[i] - 'a' ];

      if ( nextNode == 0 ) { /* αν δεν υπάρχει ο κόμβος στο
                                trie, τότε τον δημιουργούμε */
         trieNodeCount++;
         Trie[curNode].children[ word[i] - 'a' ] = trieNodeCount;
         curNode = trieNodeCount;
      }
      else {
         curNode = nextNode;
      }

      Trie[curNode].prefixes++;
   }
}
```

Αυξάνουμε το prefixes για κάθε κόμβο στο μονοπάτι.

Tries: Υλοποίηση της remove

```
int remove(char word, int length) { /* ακολουθούμε το μονοπάτι μέχρι τον
                                       κόμβο του τελευταίου γράμματος και στη
                                       συνέχεια αλλάζουμε το isWord σε 0 */
    int i, curNode = 1; /* τοποθετούμε το βέλος στην ρίζα του trie */
    for (i=0; i<length; i++) {
        curNode = Trie[curNode].children[ word[i] - 'a' ];

        assert(curNode != 0); /* υποθέτουμε ότι η λέξη που θέλουμε να
                                 διαγράψουμε υπάρχει πάντα */
        Trie[curNode].prefixes--;
    }
}
```



Αντίστοιχα εδώ μειώνουμε το πλήθος των προθεμάτων για κάθε κόμβο του μονοπατιού.

Tries: Υλοποίηση της prefixCount

```
int prefixCount(char word, int length) {
    int i, curNode = 1; /* τοποθετούμε το βέλος στην ρίζα του trie */
    for (i=0; i<length; i++) {
            curNode = Trie[curNode].children[ word[i] - 'a' ];

            if ( curNode == 0 ) { /* αν δεν υπάρχει ο κόμβος στο trie
            τότε σίγουρα δεν υπάρχουν λέξεις με το prefix που ψάχνουμε */
                return 0;
            }
    }
    return Trie[curNode].prefixes;
}
```

Tries: Παραλλαγές

- Αν δεν επαρκεί η μνήμη για να αποθηκεύσουμε έναν πίνακα children μήκους |Σ| σε κάθε κόμβο, τότε μπορούμε να έχουμε έναν δυναμικό πίνακα που να περιέχει μόνο τα παιδιά που πράγματι υπάρχουν.
- Μπορούμε να αποθηκεύουμε μεγαλύτερα κομμάτια συμβολοσειρών σε κάθε κόμβο (αντί για ένα μόνο γράμμα).

Binary Indexed Trees

Πρόβλημα:

Έχουμε N κουτιά αριθμημένα από το 1 ως το N. Θέλουμε να φτιάξουμε ένα πρόγραμμα που να υποστηρίζει τις πράξεις:

- ADD S X: Πρόσθεσε X σπίρτα στο κουτί S.
- SUM X Y: Βρες το άθροισμα των σπίρτων που βρίσκονται από το κουτί X έως το κουτί Y. (X <= Y)

Binary Indexed Trees

Προφανής Λύση:

Έχουμε έναν πίνακα N ακεραίων. Στην i-οστη θέση του πίνακα είναι αποθηκευμένο το πλήθος των σπίρτων που βρίσκονται στο i-οστο κουτί.

- ADD S X: Απλά προσθέτουμε τον αριθμό X στην θέση S του πίνακα. Αυτό φυσικά απαιτεί σταθερό χρόνο.
- SUM X Y: Πρέπει να προσθέσουμε όλες τις θέσεις από την X έως και την Y. Στην χειρότερη περίπτωση θα πρέπει να εκτελέσουμε N προσθέσεις. Επομένως η πολυπλοκότητα της SUM είναι O(N).

Binary Indexed Trees

Λύση με μερικά αθροίσματα:

Έχουμε έναν πίνακα Π από Ν ακεραίους. Στην i-οστη θέση του πίνακα είναι αποθηκευμένο το άθροισμα των σπίρτων που βρίσκονται από το 1ο μέχρι και το i-οστο κουτί.

- ADD S X: Θα πρέπει να προσθέσουμε το Χ σε όλες τις θέσεις από την S έως και την Ν.

Binary Indexed Trees

Λύση με μερικά αθροίσματα:

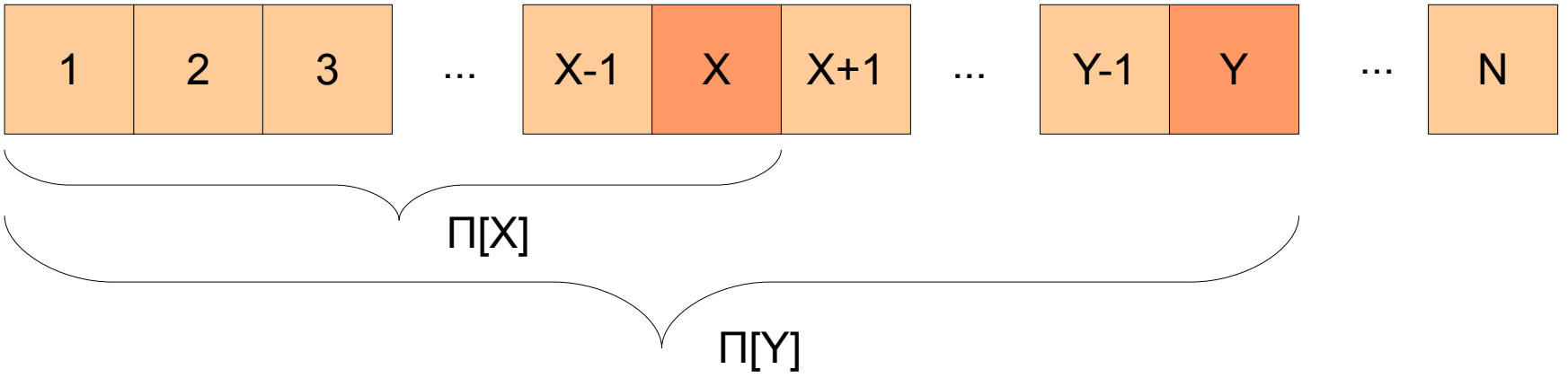
- SUM X Y: Το στοιχείο της θέσης Y (δηλαδή το Π[Y]) περιέχει το άθροισμα:

  κουτί[1] + κουτί[2] + κουτι[3] + κουτι[X-1] + κουτι[X] + ... + κουτι[Y]

  Από αυτό θέλουμε να αφαιρέσουμε τα:

  κουτί[1] + κουτί[2] + κουτι[3] + κουτι[X-1]

  Που αντιστοιχούν στο στοιχείο Π[X-1]



Binary Indexed Trees

Λύση με μερικά αθροίσματα:

- Πιο προχωρημένη ιδέα, αλλά:
- ADD: O(N)
- SUM: O(1)
- Δεν κερδίσαμε τίποτα

Binary Indexed Trees

Λύση με buckets:

- Χωρίζουμε τον πίνακα σε O(N) buckets.
- ADD: O(1)
- SUM: O(sqrt(N))

Binary Indexed Trees

Λύσεις:

	ADD	**SUM**
Προφανής Λύση	O(1)	O(N)
Μερικά Αθροίσματα	O(N)	O(1)
Buckets	O(1)	O(sqrt(N))
Binary Indexed Trees	**O(logN)**	**O(logN)**

Binary Indexed Trees

- Είναι μια δομή δεδομένων που μας μας επιτρέπει να διαχειριζόμαστε μερικά αθροίσματα. Συγκεκριμένα υποστηρίζει τις πράξεις:
  - Πρόσθεση ενός αριθμού σε μια συγκεκριμένη θέση.
  - Άθροισμα των αριθμών που βρίσκονται σε ένα συγκεκριμένο διάστημα.

Binary Indexed Trees

- Ουσιαστικά είναι ένας πίνακας αριθμών. Σε κάθε θέση του πίνακα αποθηκεύεται το άθροισμα των σπίρτων που βρίσκονται στο κουτί αυτής της θέσης καθώς και σε κάποιες από τις αμέσως προηγούμενες.

  π.χ.

  - Στην 12η θέση του πίνακα αποθηκεύονται τα σπίρτα που βρίσκονται από το 9ο έως και το 12ο κουτί.
  - Αντίστοιχα στην 14η θέση του πίνακα αποθηκεύεται το πλήθος των σπίρτων που βρίσκονται από το 13ο έως και το 14ο κουτί.

Binary Indexed Trees

- Πως αποφασίζουμε τα σπίρτα πόσων κουτιων θα αποθηκεύουμε σε κάθε θέση;
- Εξαρτάται αποκλειστικά από τον αριθμό της θέσης.
- Μετατροπή στο δυαδικό σύστημα.
- Κάθε θέση αποθηκεύει τόσα κουτιά όσα δείχνει ο μικρότερος άσσος στην δυαδική αναπαράσταση.

Binary Indexed Trees

- $12_{10} = 1\mathbf{1}00_{2}$. Αν κρατήσουμε μόνο τα ψηφία από τον τελεύταιο άσσο και δεξιά, τότε έχουμε $\mathbf{1}00_{2} = 4_{10}$. Άρα στην θέση 12 αποθηκεύουμε τα κουτιά των 4 τελευταίων θέσεων.

  Δηλαδή από το 9ο έως το 12ο κουτί.

Binary Indexed Trees

- $40_{10} = 101000_2$. Αν κρατήσουμε μόνο τα ψηφία από τον τελεύταιο άσσο και δεξιά, τότε έχουμε $1000_2 = 8_{10}$. Άρα στην θέση 40 αποθηκεύουμε τα κουτιά των 8 τελευταίων θέσεων.

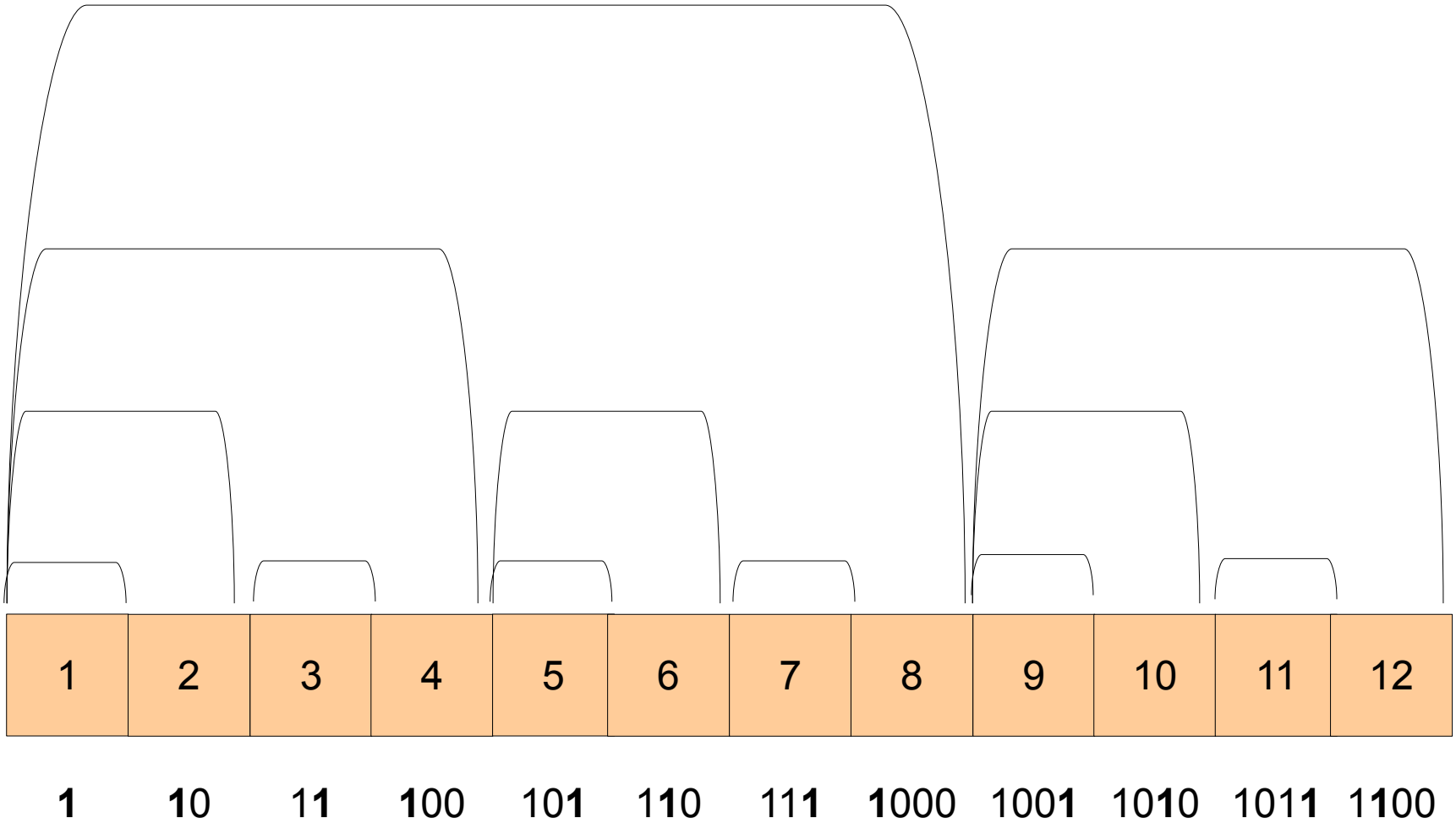
  Δηλαδή από το 33ο έως το 40ο κουτί.

Binary Indexed Trees

- $41_{10} = 101001_{2}$. Αν κρατήσουμε μόνο τα ψηφία από τον τελεύταιο άσσο και δεξιά, τότε έχουμε $\mathbf{1}_{2} = 1_{10}$. Άρα στην θέση 40 αποθηκεύουμε το κουτί μόνο της τελευταίας θέσης.

  Δηλαδή μόνο το 41ο κουτί.

Binary Indexed Trees



Binary Indexed Trees

Θέση	Περιέχει
1	1
2	1 ... 2
3	3
4	1 ... 4
5	5
6	5 ... 6
7	7
8	1 ... 8
9	9
10	9 ... 10
11	11
12	9 ... 12
13	13
14	13 ... 14
15	15
16	1 ... 16

Binary Indexed Trees

- Πώς απαντάμε στα ερωτήματα SUM 1 Y;
  - **Βήμα 1**

    Προσθέτουμε την τιμή της θέσης Y.
  - **Βήμα 2**

    Αφαιρούμε τον μικρότερο (δεξιότερο) άσσο από την δυαδική αναπαράσταση του Y. (π.χ. 10**1**0 → 10**0**0)
  - **Βήμα 3**

    Επαναλαμβάνουμε μέχρι το Y να γίνει 0.

Binary Indexed Trees

Θέση:	1	2	3	4	5	6	7	8	9	10	11	12
(Σπίρτα) Σ:	7	0	3	2	0	0	0	4	6	3	2	8
BIT:	7	7	3	12	0	0	0	16	6	9	2	10

Έστω ότι θέλουμε να υπολογίσουμε το άθροισμα:
Σ[1] + Σ[2] + Σ[3] + ... + Σ[11]

Θέλουμε, δηλαδή, να απαντήσουμε στο ερώτημα:
SUM 1 11

Binary Indexed Trees

Θέση:	1	2	3	4	5	6	7	8	9	10	11	12
(Σπίρτα) Σ:	7	0	3	2	0	0	0	4	6	3	2	8
BIT:	7	7	3	12	0	0	0	16	6	9	2	10

Αρχίζουμε από την θέση 11 και προσθέτουμε το BIT[11] = 2 στο άθροισμα.

SUM = 2

Binary Indexed Trees

Θέση:	1	2	3	4	5	6	7	8	9	10	11	12
(Σπίρτα) Σ:	7	0	3	2	0	0	0	4	6	3	2	8
BIT:	7	7	3	12	0	0	0	16	6	9	2	10

Η δυαδική αναπαράσταση του 11 είναι 101**1.** Αφαιρούμε τον πρώτο άσσο, ενώ τα υπόλοιπα ψηφία παραμένουν ίδια

$$
\begin{array}{r}
1011 \\
-\ 0001 \\
\hline
1010 \quad = 10_{10}
\end{array}
$$

SUM = 2

Binary Indexed Trees

Θέση:	1	2	3	4	5	6	7	8	9	10	11	12
(Σπίρτα) Σ:	7	0	3	2	0	0	0	4	6	3	2	8
BIT:	7	7	3	12	0	0	0	16	6	9	2	10

Πλέον βρισκόμαστε στην θέση 10, οπότε προσθέτουμε την τιμή του BIT[10] στο άθροισμα.

SUM = 2 **+ 9** = 11

Binary Indexed Trees

Θέση:	1	2	3	4	5	6	7	8	9	10	11	12
(Σπίρτα) Σ:	7	0	3	2	0	0	0	4	6	3	2	8
BIT:	7	7	3	12	0	0	0	16	6	9	2	10

Η δυαδική αναπαράσταση του 10 είναι 1010. Αφαιρούμε τον πρώτο άσσο, ενώ τα υπόλοιπα ψηφία παραμένουν ίδια

```
  1010
- 0010
------
  1000   = 8_10
```

SUM = 11

Binary Indexed Trees

Θέση:	1	2	3	4	5	6	7	8	9	10	11	12
(Σπίρτα) Σ:	7	0	3	2	0	0	0	4	6	3	2	8
BIT:	7	7	3	12	0	0	0	16	6	9	2	10

Τώρα είμαστε στην θέση 8, οπότε προσθέτουμε την τιμή του BIT[8] στο άθροισμα.

SUM =11 **+ 16** = 27

Binary Indexed Trees

Θέση:	1	2	3	4	5	6	7	8	9	10	11	12
(Σπίρτα) Σ:	7	0	3	2	0	0	0	4	6	3	2	8
BIT:	7	7	3	12	0	0	0	16	6	9	2	10

Η δυαδική αναπαράσταση του 8 είναι 1000. Αφαιρούμε τον πρώτο άσσο, ενώ τα υπόλοιπα ψηφία παραμένουν ίδια

```
  1000
- 1000
------
  0000   = 0
```

Άρα η τελική απάντηση είναι SUM = 27.

Binary Indexed Trees

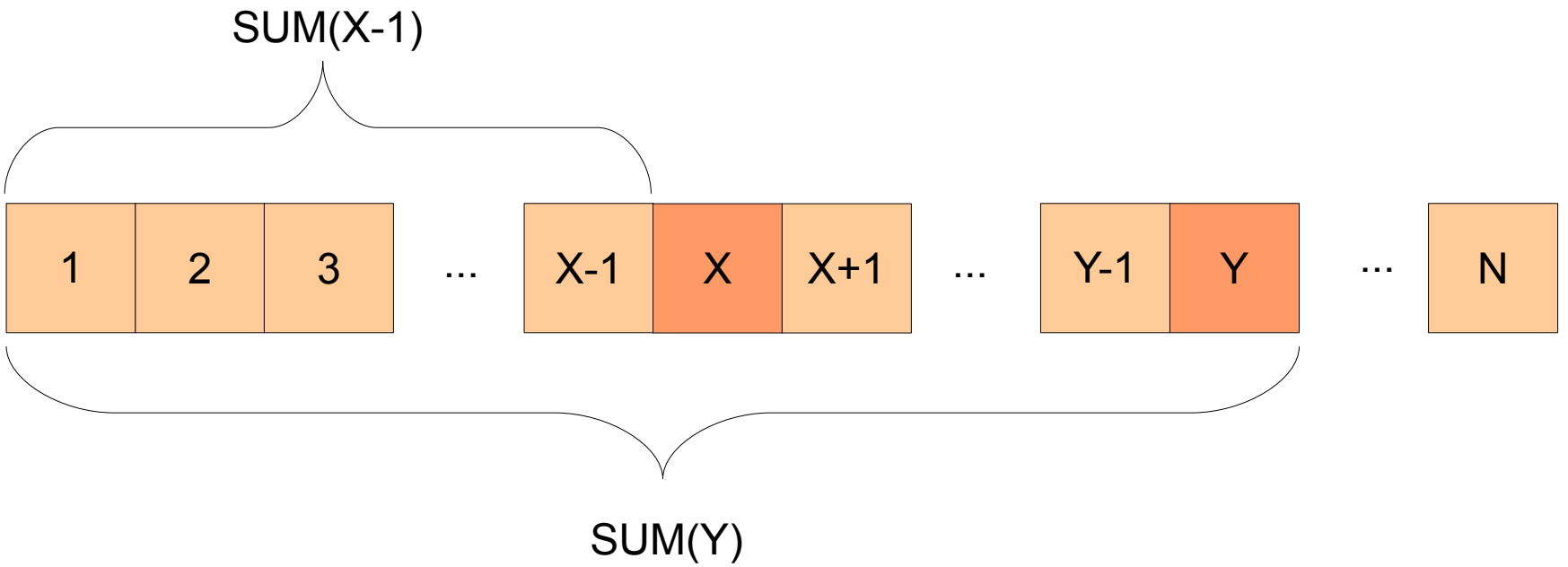
- **Ερώτηση:** Γιατί είναι O(logN);
  - Ξεκινώντας από το Y, θα κάνουμε τόσες προσθέσεις όσοι είναι οι άσσοι του Y (αφού σε κάθε βήμα θα κάνουμε μια πρόσθεση και θα σβήνουμε έναν άσσο από την δυαδική αναπαράστασή του).
  - Η δυαδική αναπαράσταση κάθε αριθμού N αποτελείται από $\log_2 N + 1$ ψηφία.
  - Θα εκτελέστουν το πολύ $\log_2 N + 1 = O(\log N)$ προσθέσεις.

Binary Indexed Trees

- Πως απαντάμε στο ερώτημα SUM X Y;

Το σπάμε σε δύο:

SUM X Y = SUM 1 Y – SUM 1 X-1

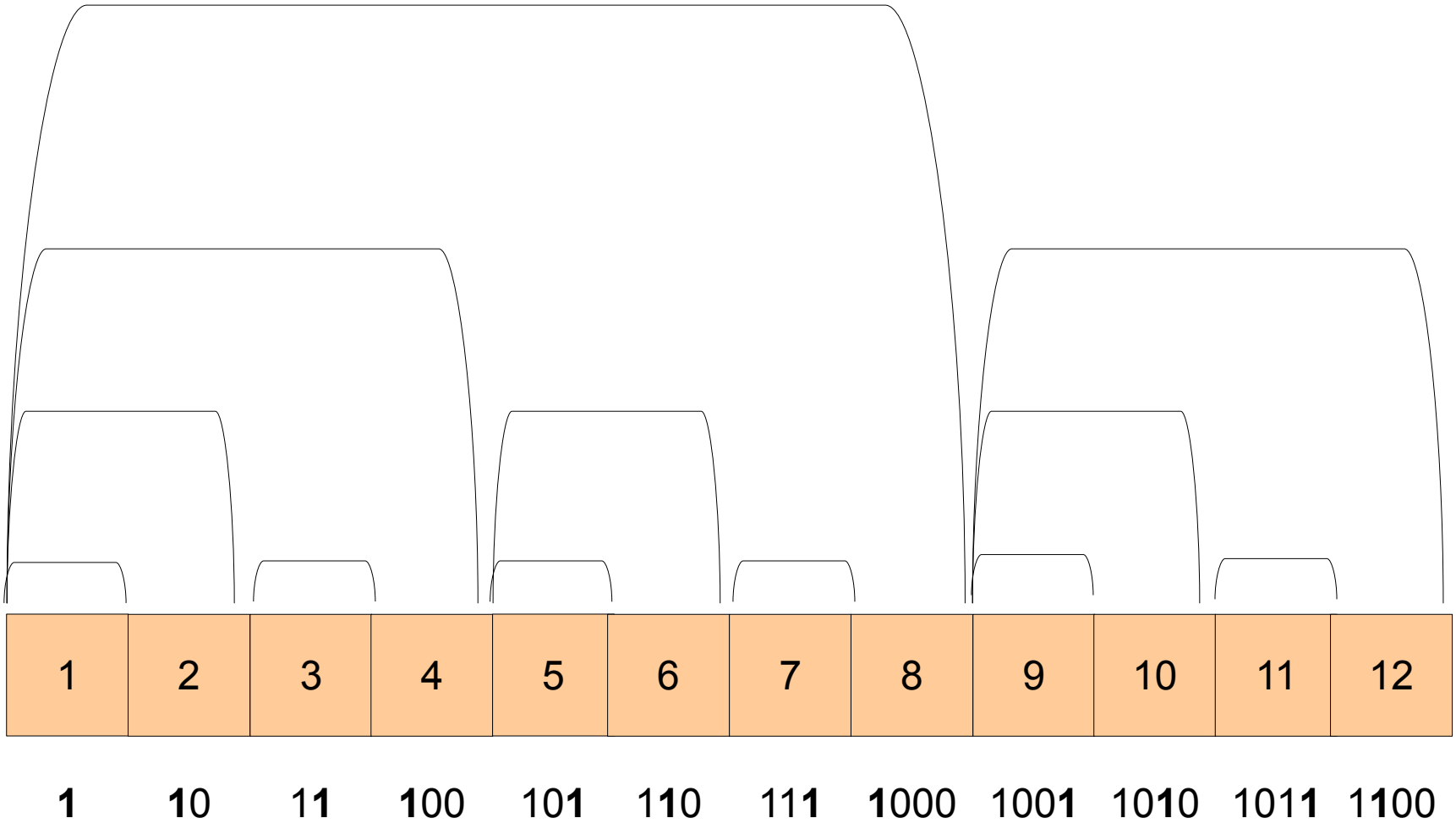


Binary Indexed Trees

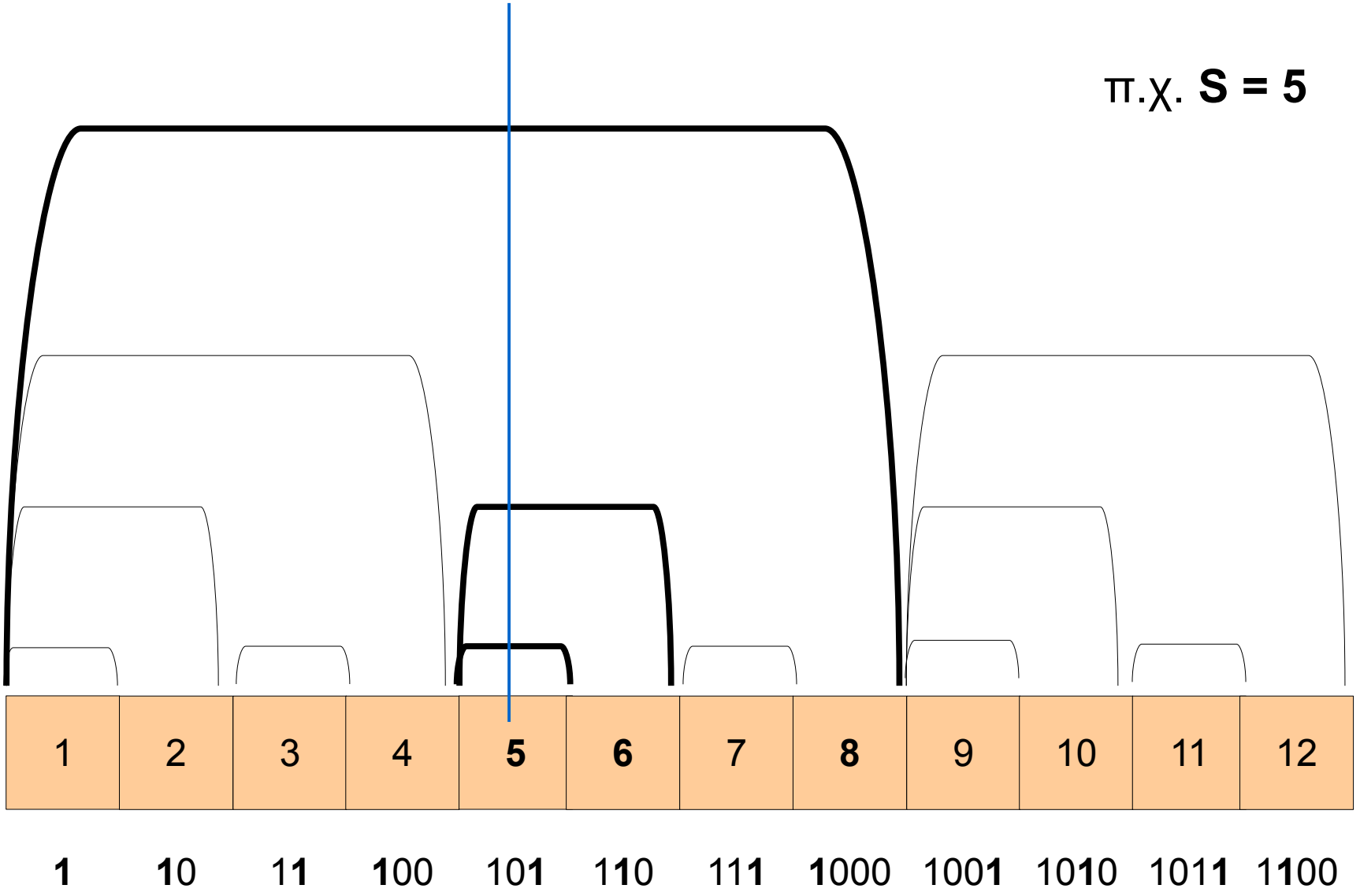
- Πώς απαντάμε στα ερωτήματα ADD S X;
  - Πρέπει να αλλάξουμε την τιμή όλων των θέσεων που περιέχουν την S.
  - Αντίστροφη Διαδικασία

Binary Indexed Trees

π.χ. **S = 5**

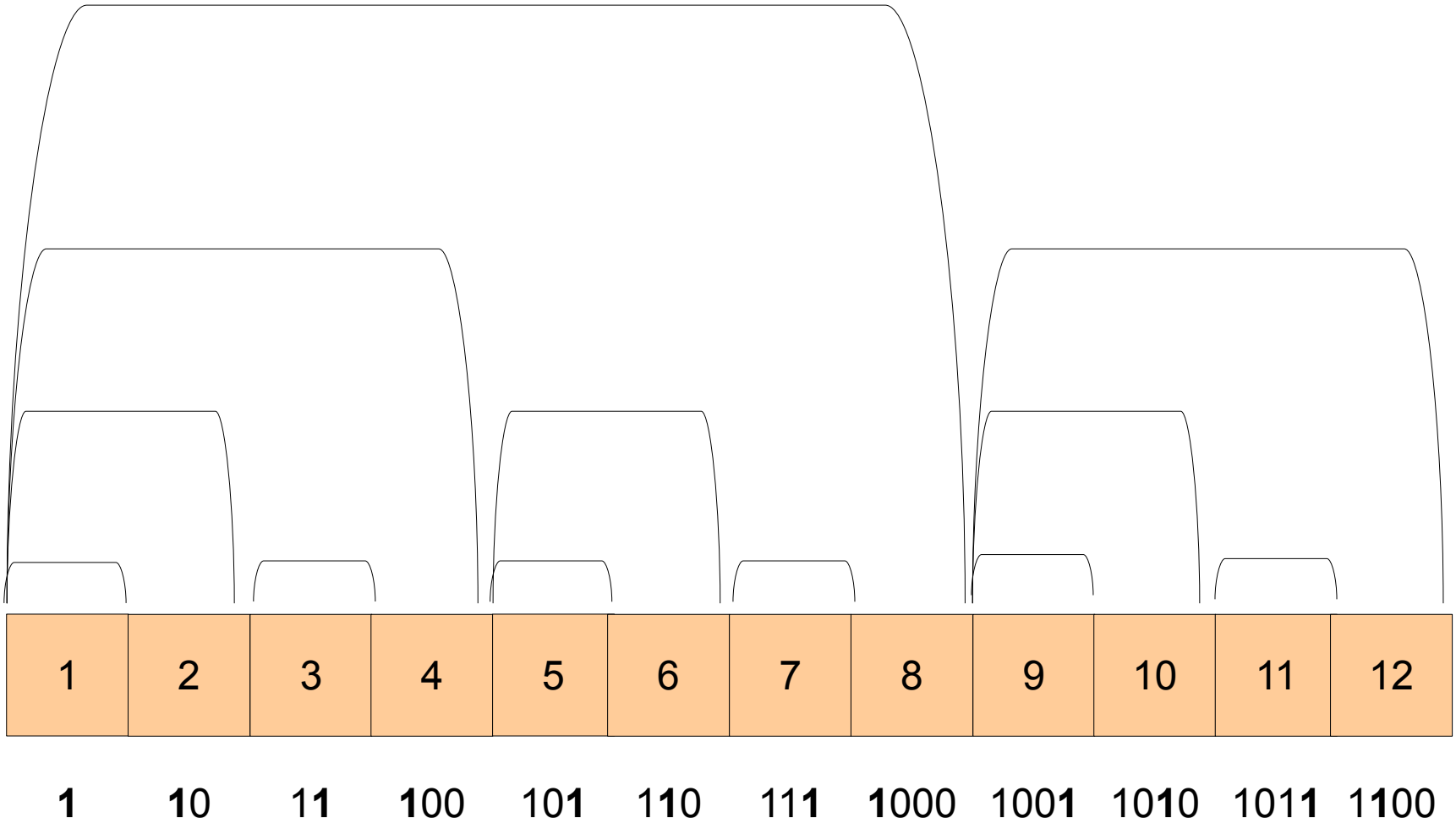


Binary Indexed Trees



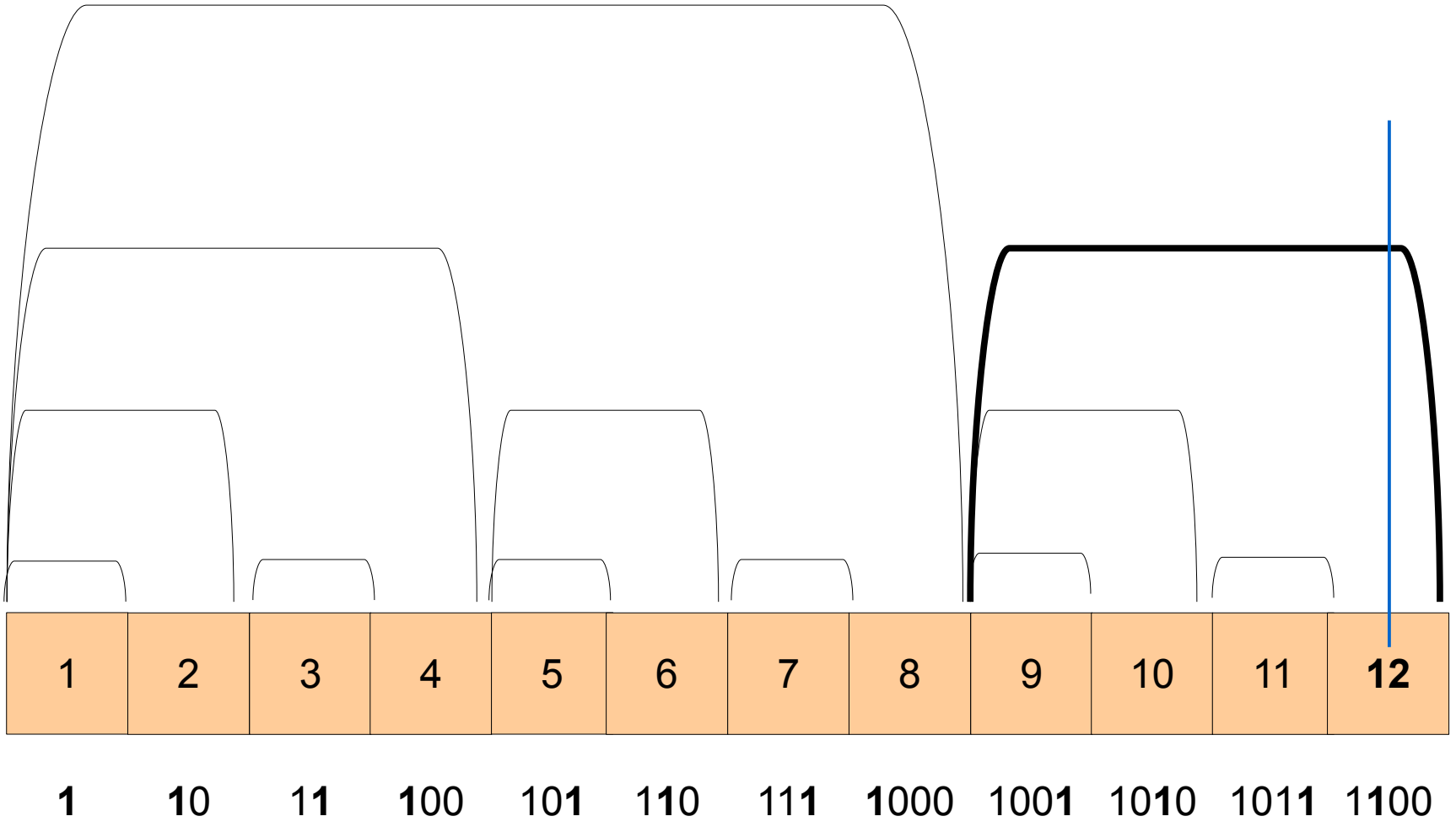
Binary Indexed Trees

π.χ. **S = 12**



Binary Indexed Trees

π.χ. **S = 12**



Binary Indexed Trees

- **Βήμα 1**

  Προσθέτουμε τον αριθμό X στην θέση S.

- **Βήμα 2**

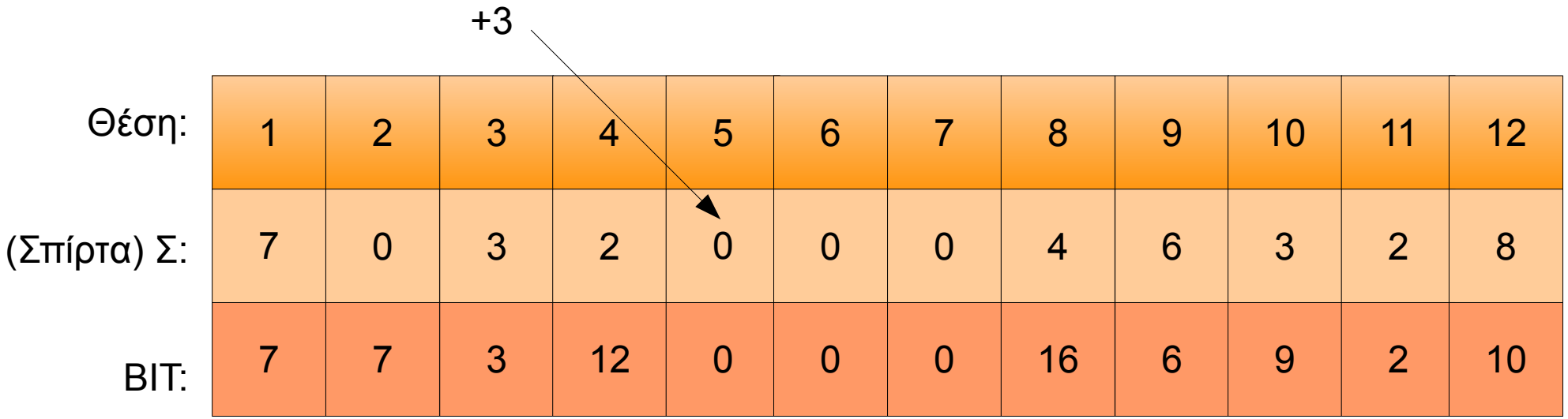
  Προσθέτουμε τον μικρότερο (δεξιότερο) άσσο στην την δυαδική αναπαράσταση του S.

  (π.χ. 10**1**0 → 1100)

  Βήμα 3

  Επαναλαμβάνουμε μέχρι το S να ξεπεράσει το μέγεθος του πίνακα.

Binary Indexed Trees



Έστω ότι θέλουμε να προσθέσουμε 3 σπίρτα στην θέση 5:

Θέλουμε, δηλαδή, να εκτελέσουμε την πράξη:

ADD 5 3

Binary Indexed Trees

Θέση:	1	2	3	4	5	6	7	8	9	10	11	12
(Σπίρτα) Σ:	7	0	3	2	3	0	0	4	6	3	2	8
BIT:	7	7	3	12	**3**	0	0	16	6	9	2	10

Αρχικά προσθέτουμε το 3 στην θέση 5 του πίνακα BIT.

Binary Indexed Trees

Θέση:	1	2	3	4	5	6	7	8	9	10	11	12
(Σπίρτα) Σ:	7	0	3	2	3	0	0	4	6	3	2	8
BIT:	7	7	3	12	**3**	0	0	16	6	9	2	10

Η δυαδική αναπαράσταση του 5 είναι 10**1.** Προσθέτουμε, Λοιπόν, πρώτο άσσο:

```
  101
+ 001
-----
  110   = 6₁₀
```

Binary Indexed Trees

Θέση:	1	2	3	4	5	6	7	8	9	10	11	12
(Σπίρτα) Σ:	7	0	3	2	3	0	0	4	6	3	2	8
BIT:	7	7	3	12	**3**	**3**	0	16	6	9	2	10

Άρα η επόμενη θέση του πίνακα BIT στην οποία πρέπει να προστεθεί το 3 είναι η θέση 6.

Binary Indexed Trees

Θέση:	1	2	3	4	5	6	7	8	9	10	11	12
(Σπίρτα) Σ:	7	0	3	2	3	0	0	4	6	3	2	8
BIT:	7	7	3	12	**3**	**3**	0	16	6	9	2	10

Η δυαδική αναπαράσταση του 6 είναι 1**1**0**.** Προσθέτουμε, ξανά πρώτο άσσο:

```
  110
+ 010
-----
 1000   = 8_10
```

Binary Indexed Trees

Θέση:	1	2	3	4	5	6	7	8	9	10	11	12
(Σπίρτα) Σ:	7	0	3	2	3	0	0	4	6	3	2	8
BIT:	7	7	3	12	**3**	**3**	0	**19**	6	9	2	10

Έτσι το προσθέτουμε το 3 στο BIT[8] και γίνεται:

BIT[8] = 16 **+ 3 = 19**

Binary Indexed Trees

Θέση:	1	2	3	4	5	6	7	8	9	10	11	12
(Σπίρτα) Σ:	7	0	3	2	3	0	0	4	6	3	2	8
BIT:	7	7	3	12	**3**	**3**	0	**19**	6	9	2	10

Η δυαδική αναπαράσταση του 8 είναι **1**000**.** Προσθέτοντας τον πρώτο (και μοναδικό) άσσο παίρνουμε:

$$
\begin{array}{r}
1000 \\
+\ 1000 \\
\hline
10000
\end{array}
\quad = 16_{10}
$$

Binary Indexed Trees

Θέση:	1	2	3	4	5	6	7	8	9	10	11	12
(Σπίρτα) Σ:	7	0	3	2	3	0	0	4	6	3	2	8
BIT:	7	7	3	12	**3**	**3**	0	**19**	6	9	2	10

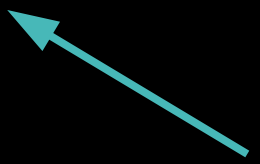
Το 16 όμως είναι μεγαλύτερο από το μέγεθος του πίνακα BIT, άρα η διαδικασία τερματίζει εδώ.

Binary Indexed Trees: Υλοποίηση της sum

```
int sum(int Y) {
    int answer = 0; /* μεταβλητή που κρατάει το ζητούμενο άθροισμα */
    while (Y > 0) {
        answer += BIT[Y];
        Y -= (Y & -Y);
    }
    return answer;
}
```

Binary Indexed Trees: Υλοποίηση της sum

```
int sum(int Y) {
    int answer = 0; /* μεταβλητή που κρατάει το ζητούμενο άθροισμα */
    while (Y > 0) {
        answer += BIT[Y];
        Y -= (Y & -Y);
    }
    return answer;
}
```

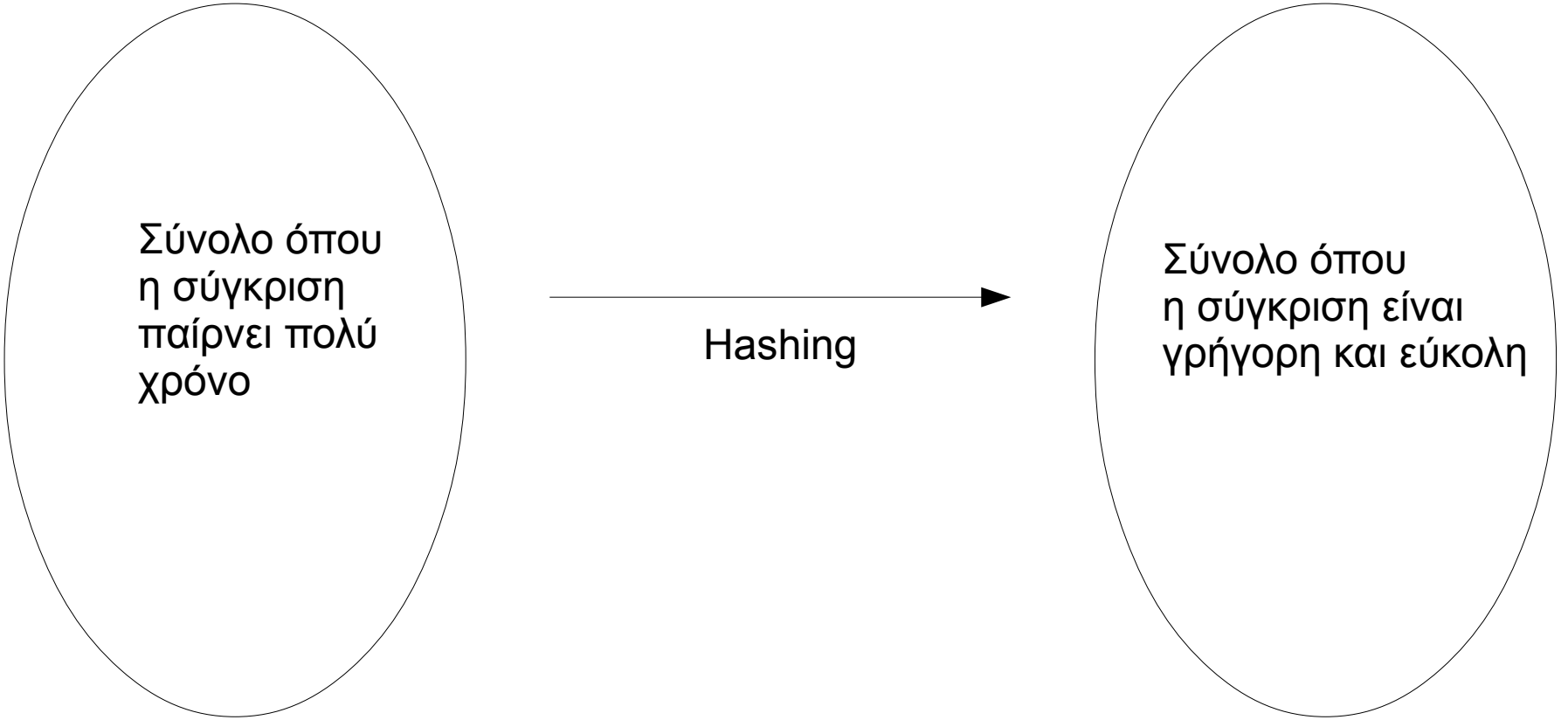


Τρικ

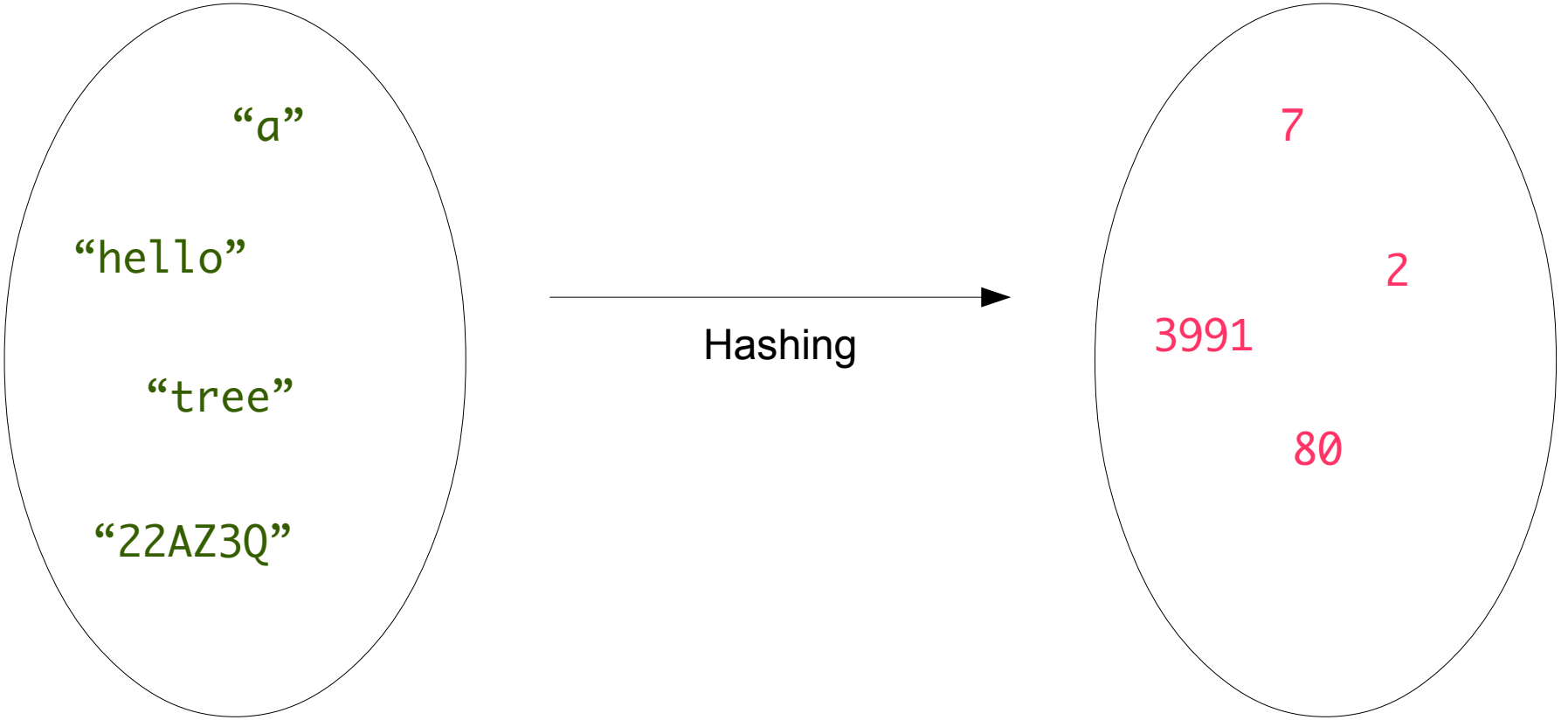
Binary Indexed Trees: Υλοποίηση της add

```
void add(int pos, int num) {
    while (pos <= N) {
        BIT[pos] += num;
        pos += (pos & -pos);
    }
}
```

Hashing


Σύνολο όπου
η σύγκριση
παίρνει πολύ
χρόνο
Hashing
Σύνολο όπου
η σύγκριση είναι
γρήγορη και εύκολη

Hashing



Hash function

- Μετατρέπει ένα στοιχείο του πρώτου συνόλου σε ένα στοιχείο του δεύτερου.
- Λειτουργεί πάντα με τον ίδιο τρόπο (όσες φορές και να δώσουμε ένα στοιχείο του πρώτου συνόλου θα μας επιστρέφει το ίδιο στοιχείο του δεύτερου)
- π.χ.

```
H(“hello”) = 33
H(“test”) = 87
H( [1, 9, 4, 3] ) = 11
```

Collisions

- Δύο στοιχεία του πρώτου συνόλου δείχνουν στο ίδιο δεύτερο σύνολο!
- π.χ.
  - `H("hello") = 33`
  - `H("bye") = 33`
- Θέλουμε να έχουμε όσο το δυνατόν λιγότερα collisions. Αν είναι δυνατόν, να μην έχουμε κανένα.
- Πρέπει να επιλέξουμε ένα καλό hash function.

Perfect Hashing

- Κάθε στοιχείο του δεύτερου συνόλου αντιστοιχεί σε μοναδικό στοιχείο του πρώτου συνόλου.
- Είναι 1-1.
- Έχει χρόνο εκτέλεσης O(1).

Παράδειγμα

- Χαζό hash function που μετατρέπει μια συμβολοσειρά σε έναν αριθμό:

```
int hash(char *word, int length) {
    int i, H = 0;
    for (i = 0; i < length; i++) {
        H += word[i];
    }
    return H;
}
```

- Έχει πολλά collisions:
  - hash(“AB”) = 65 + 66 = 131
  - hash(“BA”) = 66 + 65 = 131

Παράδειγμα (linear hashing)

- Γραμμικό hash function: θεωρεί μια συμβολοσειρά ως αριθμό εκφρασμένο στο |Σ|-ικο σύστημα και τον μετατρέπει στο δεκαδικό

```
int hash(char *word, int length) {
    int i, H = 0, base = 1;
    for (i = 0; i < H; i++) {
        H = H[i] + s[i]*base;
        base *= 256;
    }
    return H;
}
```

Το πλήθος των διαφορετικών συμβόλων

Παράδειγμα (linear hashing)

- **Πλεονέκτημα:**

  Δεν έχει καθόλου collisions

- **Μειονεκτήματα:**

  Μπορεί να δώσει πολύ μεγάλα αποτελέσματα που δεν χωράνε ούτε σε ακεραίους 64bit. Έτσι πρέπει να χρησιμοποιήσουμε αριθμητική υπολοίπων για να περιορίσουμε το αποτέλεσμα. Αυτό οδηγεί σε:

  - Μερικά collisions
  - Αύξηση του χρόνου εκτέλεσης

Double hashing

- Για να μειώσουμε ακόμα περισσότερο τα collisions μπορούμε να συνδυάσουμε δύο μεθόδους hashing:

$h_1(\text{“hello”}) = 32$

$h_2(\text{“hello”}) = 991$

$H(\text{“hello”}) = < h_1(\text{“hello”}), h_2(\text{“hello”}) > = <32, 991>$