Arbre

Définitions

Terminologie

ADT Tree

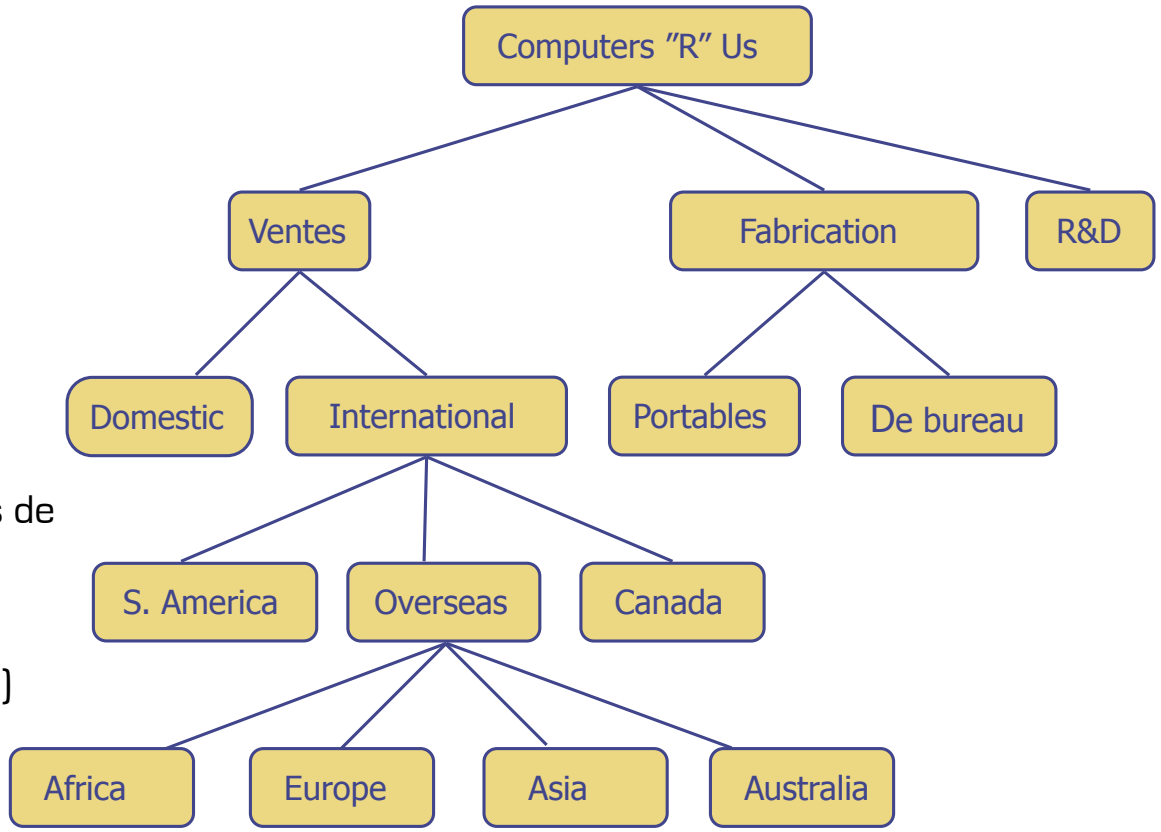
`interface` versus `abstract class`

Calcul de la profondeur et de la hauteur

Trie

Qu'est-ce qu'un arbre ?

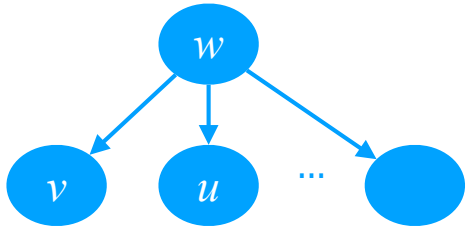
- En informatique, un **arbre** est une abstraction pour une structure hiérarchique
- Un **arbre** est constitué de **noeuds** ayant une relation de **parent-enfant**
- Les applications incluent :
  - systèmes de fichiers
  - hiérarchies organisationnelles
  - héritage en programmation objet
  - généalogies et phylogénies
  - syntaxe de langues naturels et de langages de programmation
  - expression arithmétiques
  - documents (avec sections et sous-sections)



Définitions formelles

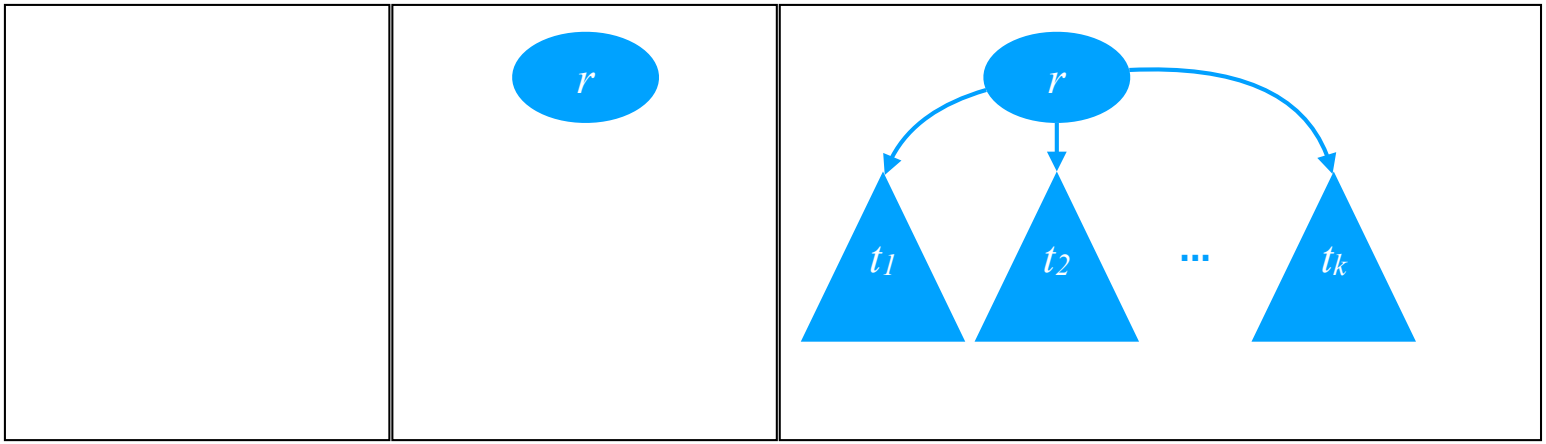
Formellement, un **arbre** T est un ensemble de **nœuds** stockant des éléments tels que les nœuds possèdent une relation **parent-enfant** qui satisfait les propriétés suivantes :

- Si T n'est pas vide, il a un nœud spécial, appelé **racine** de T, qui n'a pas de parent
- Chaque **nœud** v de T différent de la **racine** possède un unique **nœud parent** w ; tous les **nœuds** qui ont comme **parent** w sont des **enfants** de w.



Un **arbre** peut être vide, c'est-à-dire n'avoir aucun **nœud**.

Définition récursive : un **arbre** T est soit vide ou est constitué d'une **nœud** r, appelé **racine** de T, et d'un ensemble de **sous-arbres** (possiblement vide) dont les **racines** sont les **enfants** de r.



L'ADT arbre
stocke des éléments de manière hiérarchique

Racine: **nœud** sans **parent** (**A**)

Nœud interne: noeud avec au moins un enfant

(**A**, **B**, **C**, **F**)

Nœud externe, ou **feuille**: nœud sans enfants

(**E**, **I**, **J**, **K**, **G**, **H**, **D**)

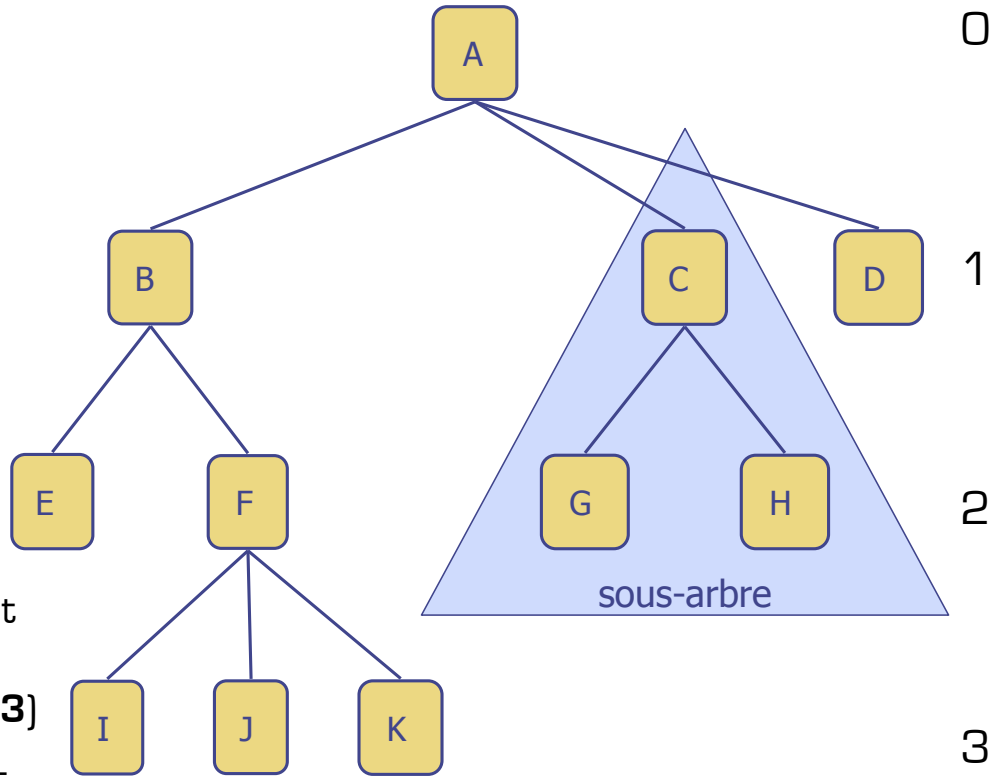
Ancêtre d'un nœud: parent, grand-parent, grand-grand-parent, etc (ancêtres de **F** : **B**, **A**)

Profondeur d'un nœud: nombre d'ancêtres (**F** possède 2 ancêtres, et donc est à **profondeur 2** ; **A** possède 0 ancêtre, et donc est à **profondeur 0**)

Hauteur d'un arbre: profondeur maximale d'un de ses nœuds (**I**, **J** et **K** possèdent les 3 mêmes ancêtres et sont donc à profondeur 3, qui se trouve être la profondeur maximale des noeuds de cet arbre, donc sa **hauteur** est **3**)

Descendant d'un nœud: enfant, petit-enfant, arrière-petit-enfant, etc (descendants de **B** : **E**, **F**, **I**, **J**, **K**)

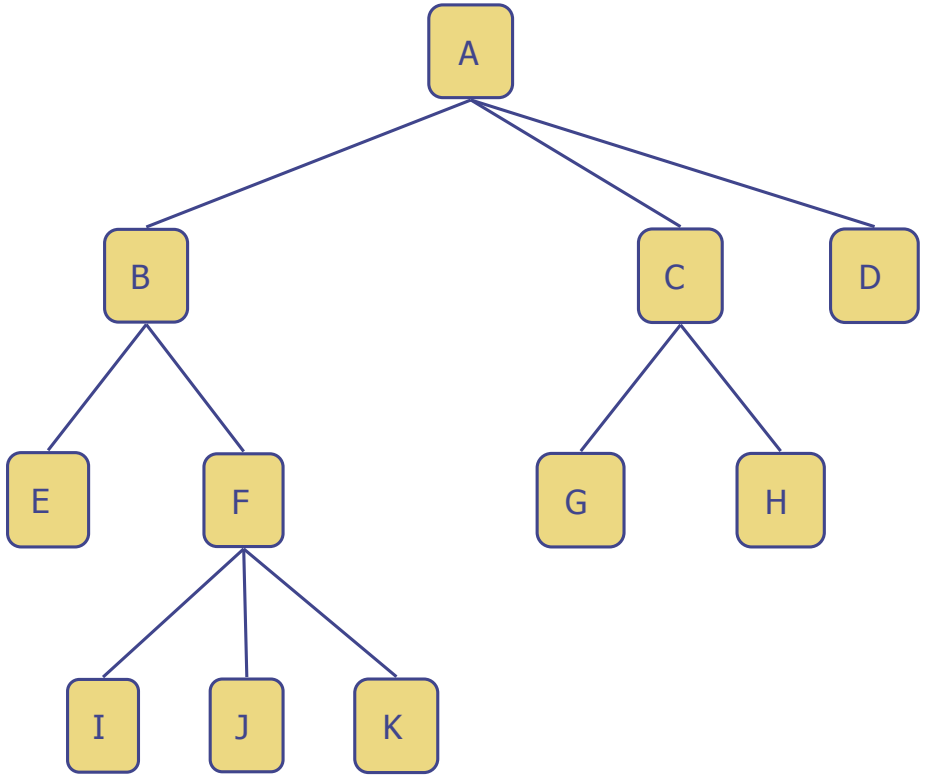
Sous-arbre d'un arbre: arbre constitué d'un de ses nœuds et de ses descendants (exemples : **C** et ses descendants **G** et **H** (en bleu) ; **B** et ses descendants **E**, **F**, **I**, **J** et **K**).



Arêtes et chemins

Arête: une paire de nœuds (u,v) tel que u est le parent de v, ou vice versa (**B**,**F**)

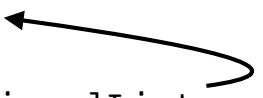
Chemin: une séquence de noeuds tel que deux noeuds consécutifs dans la séquence partage une arête (**B-F-J**; **A-C** ; **A-B-F-K**)



ADT `Tree`

(utilisant des positions pour chaque noeud de l'arbre)

- Rappel : L'interface `Position` (du package ca.umontreal.IFT2015.adt.list) supporte la méthode
  - `getElement()` : retourne l'élément stocké à cette position
- Méthodes d'accès :
  - `root()` - retourne la position de la racine de l'arbre, null si l'arbre est vide
  - `parent( p )` - retourne la position du parent de la position p, null si p est la racine
  - `children( p )` - retourne une collection itérable des enfants de la position p (s'ils existent)
  - `numChildren( p )` - retourne le nombre d'enfants de la position p
- Méthodes de requêtes :
  - `isInternal( p )` - retourne true si la position p possède au moins un enfant
  - `isExternal( p )` - retourne true si la position p ne possède pas d'enfant
  - `isRoot( p )` - retourne true si la position p est la racine de l'arbre
- Méthodes générales :
  - `size()` - retourne le nombre de d'éléments dans l'arbre
  - `isEmpty()` - retourne true si l'arbre contient aucune position (donc élément)
  - `iterator()` - retourne un itérateur des éléments de l'arbre (un arbre est itérable)
  - `positions()` - retourne une collection itérable des positions de l'arbre



👉 Comme on a fait pour `LinkedPositionalList`

Allons voir le code...

`Tree.java`

`interface` vs `abstract class`

En Java, le rôle d'une `interface` est de définir un type qui inclut des déclarations publiques de diverses méthodes.

👉 Une `interface` ne peut pas inclure de définitions pour aucune de ces méthodes.

En revanche, une `abstract class` peut définir des implémentations concrètes pour certaines de ses méthodes, tout en laissant d'autres méthodes abstraites sans définition. Cela peut servir de **classe de base** pour des implémentations ultérieures différentes.

Pourquoi une classe de base

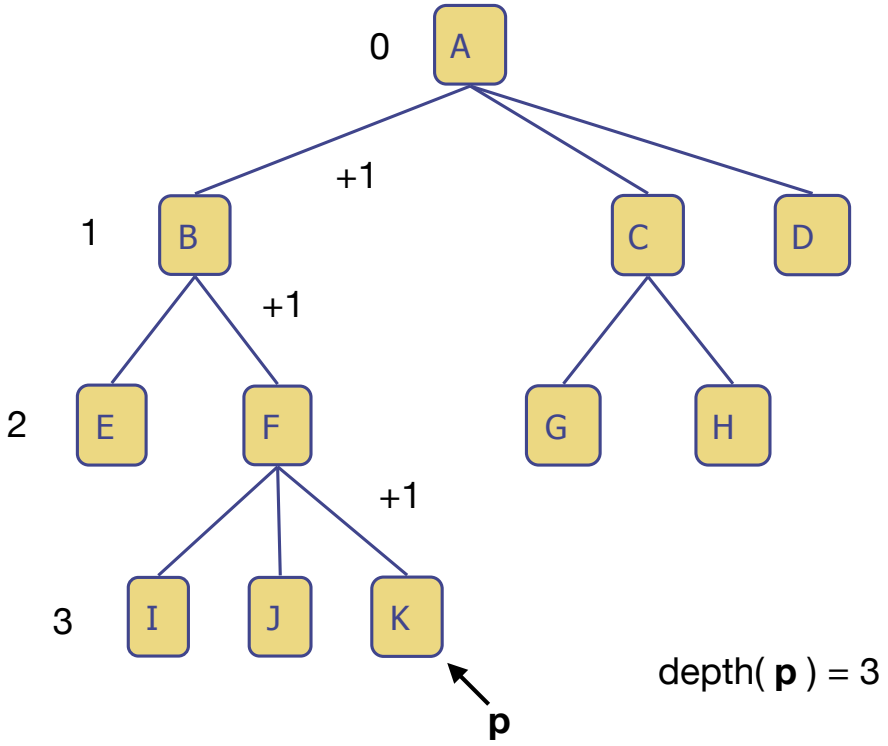
- Nous utiliserons une `abstract class` pour servir de **classe de base**, par héritage, pour une ou plusieurs implémentations ultérieures concrètes de l'interface `Tree`.
- Lorsqu'une partie des fonctionnalités d'une `interface` est implémentée dans une classe abstraite, il reste moins de travail pour achever une implémentation concrète.
- Les bibliothèques Java standard incluent de nombreuses classes abstraites, dont plusieurs au sein du Java Collections Framework. Pour clarifier leur objectif, ces classes sont classiquement nommées en commençant par le mot Abstract.
- Par exemple, il existe une classe `AbstractCollection` qui implémente certaines fonctionnalités de l'interface `Collection`, une classe `AbstractQueue` qui implémente certaines fonctionnalités de l'interface `Queue` et une classe `AbstractList` qui implémente certaines fonctionnalités de l'interface `List`.
- `AbstractTree` implémente certaines fonctionnalités de l'interface `Tree`.
- Les algorithmes d'arbres peuvent être décrits indépendamment de la représentation concrète. Si une implémentation concrète fournit les 3 méthodes : `root()`, `parent( p )` et `children( p )`, alors tous les autres comportements de l'interface `Tree` peuvent être dérivés dans la classe de base `AbstractTree`.

Allons voir le code...

`AbstractTree.java`

Les méthodes `root()`, `parent()`, `children()`, et `numChildren()` ne sont pas implémentées dans `AbstractTree` !

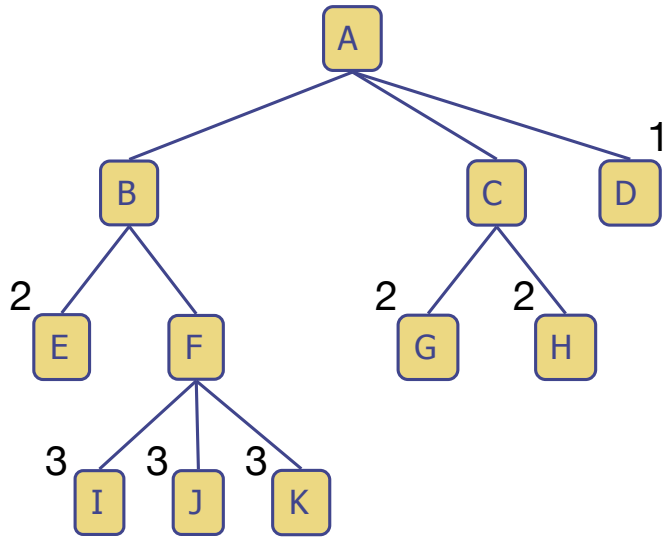
```
// return the number of levels separating Position p from the root (depth of p)
//    executes in O( depth of p + 1 ); O(n) in the worst case.
public int depth( Position<E> p ) {
    if( this.isRoot( p ) ) return 0;
    return 1 + this.depth( this.parent( p ) );
}
```



```java
// return the height of the tree
//    worst case execution time in O(n^2), bad!
public int heightBad() {
    int h = 0;
    for( Position<E> p : this.positions() )
        if( isExternal( p ) )
            h = Math.max( h, this.depth( p ) );
    return h;
}
```

```
depth( p = D ) (visite  2 noeuds)
depth( p = H ) (visite  3 noeuds)
depth( p = G ) (visite  3 noeuds)
depth( p = E ) (visite  3 noeuds)
depth( p = I ) (visite  4 noeuds)
depth( p = J ) (visite  4 noeuds)
depth( p = K ) (visite  4 noeuds)
               (visite 23 noeuds)
```

heightBad()

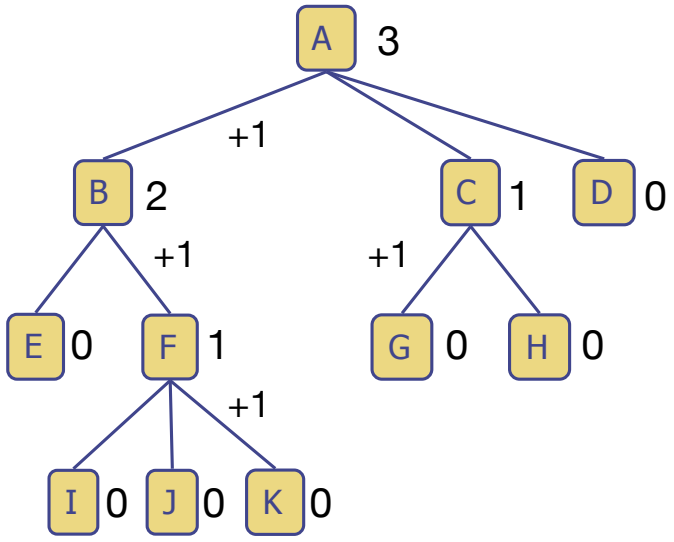


nb noeuds visités = 23

```java
// return the height of the subtree rooted at Position p
//    worst case execution time in O(n)
public int height( Position<E> p ) {
    int h = 0;
    for( Position<E> c : this.children( p ) )
        h = Math.max( h, 1 + this.height( c ) );
    return h;
}
```

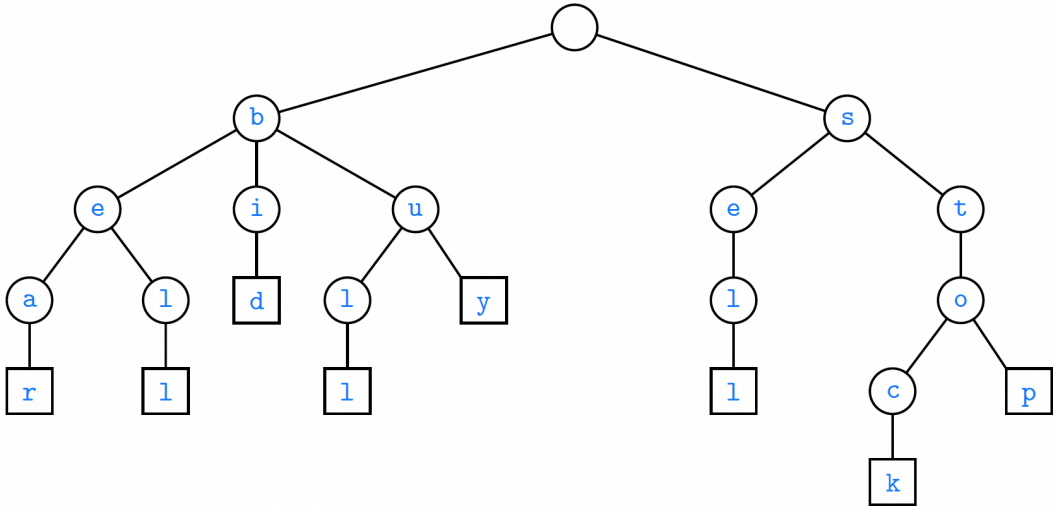
```
height( p = A )        (visite  1 noeud)
  height( p = B )      (visite  1 noeud)
    height( p = E )    (visite  1 noeud)
    height( p = F )    (visite  1 noeud)
      height( p = I )  (visite  1 noeud)
      height( p = J )  (visite  1 noeud)
      height( p = K )  (visite  1 noeud)
  height( p = C )      (visite  1 noeud)
    height( p = G )    (visite  1 noeud)
    height( p = H )    (visite  1 noeud)
  height( p = D )      (visite  1 noeud)
                       (visite 11 noeuds)
```

height(root())



nb noeuds visités = 11

Trie



Trie standard pour les Strings {bear, bell, bid, bull, buy, sell, stock, stop}

- Structure en arbre pour prétraiter un texte et stocker ses chaînes de caractères, S, pour des algorithmes de recherche.
  - Systèmes auto-complétion (trouver les mots qui commence par un préfixe)
  - Systèmes auto-correcteurs
  - Systèmes de "lookup" dans un dictionnaire

- Le nom vient de "re**trie**val". Prononcer à l'anglaise ("try").
- Largement utilisé, par exemple en bio-informatique, pour retrouver l'information d'une séquence ADN dans une base de données génomique.
- L'opération principale d'un `Trie` est le matching de préfixes: Étant donné un préfixe X, quelles sont les chaînes de S qui contiennent X (par exemple les mots dans l'exemple qui contiennent le préfixe "bu" : "bull" et "buy").

Tries standards

Étant donné S un ensemble de mots dans l'alphabet Σ tel que aucun mot dans S est un préfixe d'un autre mot. Un Trie standard pour S est un arbre ordonné T avec les propriétés suivantes :

- Chaque noeud de T, sauf la racine, est étiqueté avec un caractère de Σ.
- Les enfants d'un noeud interne de T ont des étiquettes distinctes.
- T possède s feuilles, chacune associée à un mot de S, tel que la concaténation des étiquettes des noeuds sur le chemin partant de la racine vers la feuille v de T révèle le mot de S associée à v.
- Un noeud interne de T peut avoir entre 1 et $|\Sigma|$ enfants.
- Un chemin de la racine à un noeud interne v à profondeur k correspond à un préfixe de k caractères X[0:k-1].

Un `Trie` standard qui stocke une collection S de s mots de longeur totale n d'un alphabet Σ a les propriétés suivantes :

- La hauteur de T est égale à la longueur du mot le plus long.
- Un noeud interne possède au plus $|\Sigma|$ enfants.
- T possède s feuilles.
- Le nombre de noeuds de T est au plus $n + 1$. Le **pire cas** survient lorsqu'aucun mot ne partage le même préfixe non-vide ; dans ce cas, sauf pour la racine, tous les noeuds internes possède un seul enfant.

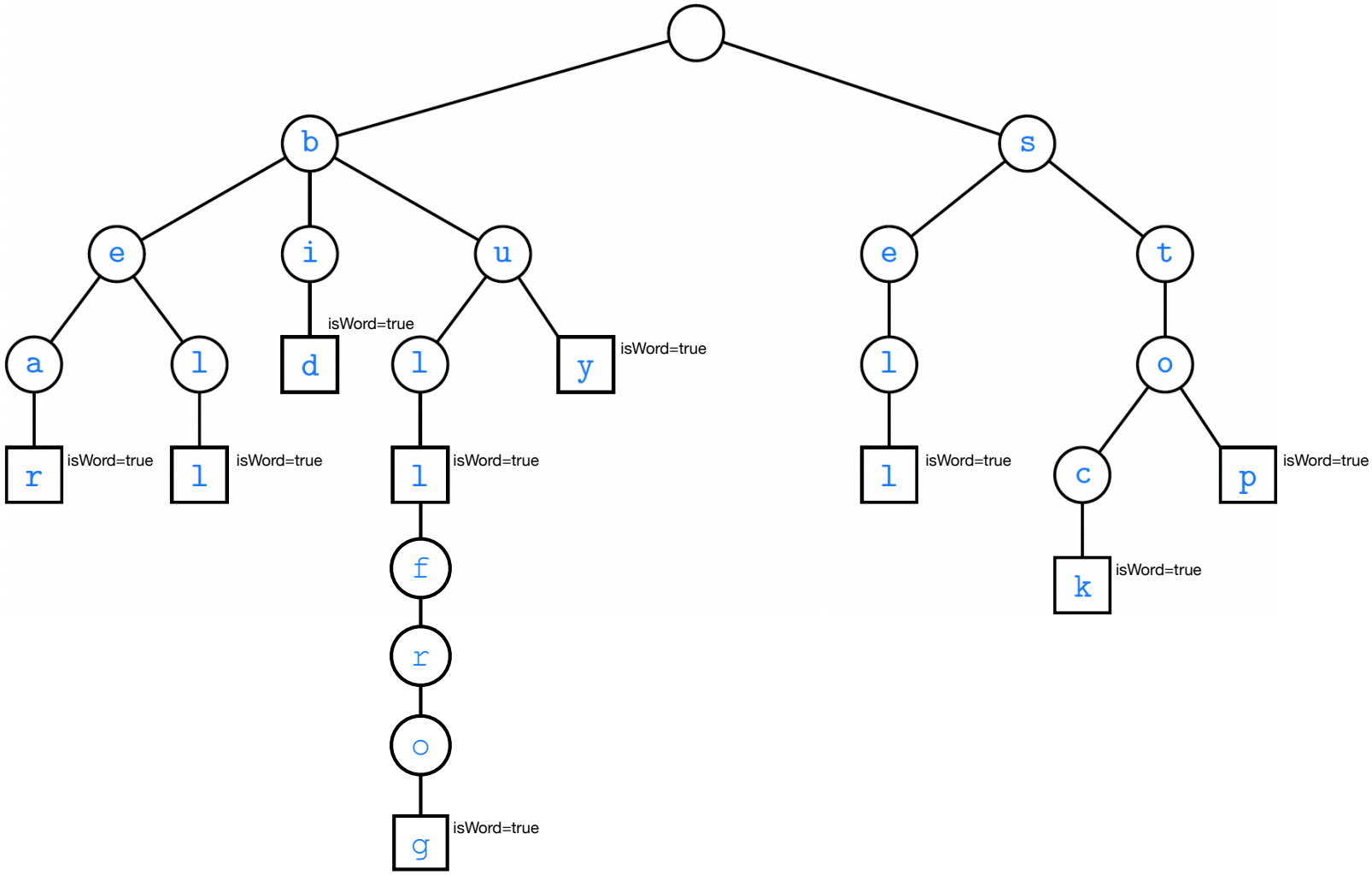
Autres propriétés et opérations

- La complexité pour **chercher** un mot de longueur m est entre $O(m \cdot |\Sigma|)$ et $O(m)$, en fonction de la structure de données utilisée pour stocker les enfants.
- Imaginez utiliser une liste non triée ! Dans ce cas, chercher la liste des possibles $|\Sigma|$ est dans $O(|\Sigma|)$. Imaginez maintenant une liste triée ! Dans ce cas, on peut chercher dans $O(\log |\Sigma|)$. Lorsqu'on utilise des listes, l'espace utilisé pour chaque noeud est optimal ; dans l'ordre du nombre de caractères nécessaires.
- Pour avoir une recherche dans $O(m)$ (l'optimal), notre implémentation utilise un tableau dont les indices correspondent aux caractères de Σ. Dans ce cas, on est dans la meilleure situation possible ; accéder au prochain caractère est dans $O(1)$. Le prix à payer, par contre, est l'utilisation en mémoire dans $O(|\Sigma|)$ par noeud.
  L'optimalité en temps et espace se fait en utilisant une `Map` (que nous verrons plus tard dans le cours) ; espace optimal et temps d'accès espéré dans $O(1)$.
- Pour **construire** un `Trie`, on insère les mots un à la fois, où chaque insertion possède la même complexité que chercher un mot.

Allons voir le code...

`ArrayTrie.java`
`TrieApp.java`

Notre implémentation permet les mots "préfixes"



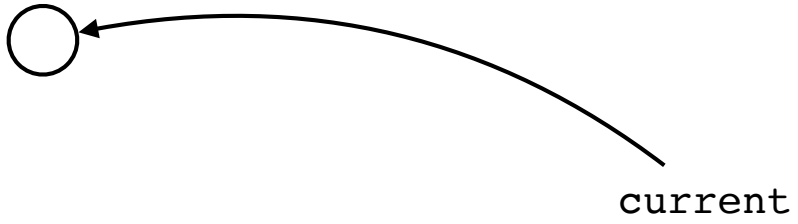
Trie étendu pour les Strings {bear, bell, bid, bull, bullfrog, buy, sell, stock, stop}

insert("bear")

```
root = new TrieNode();
```

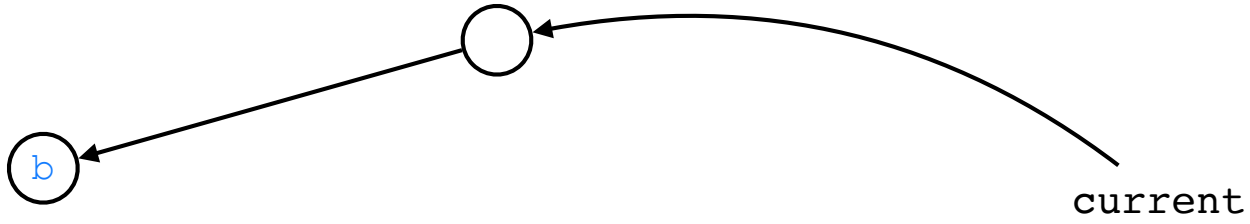
insert("bear")

```
root = new TrieNode();
insert( "bear" );
```



insert("bear")

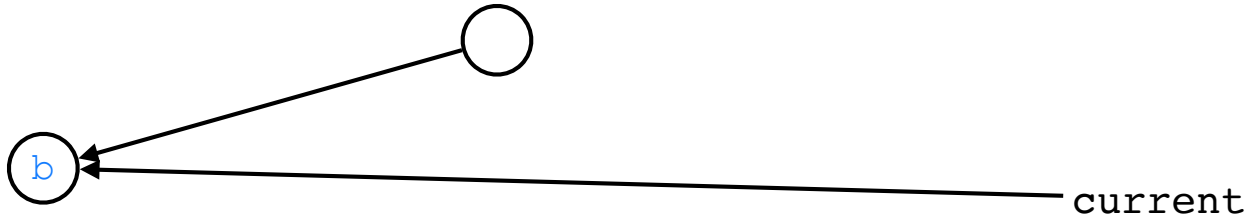
```
root = new TrieNode();
insert( "bear" );
```




```
for( char c : word.toCharArray() ) {
    if( current.children[c] == null ) {
        current.children[c] = new TrieNode();
    }
    current = current.children[c];
}
```

insert("bear")

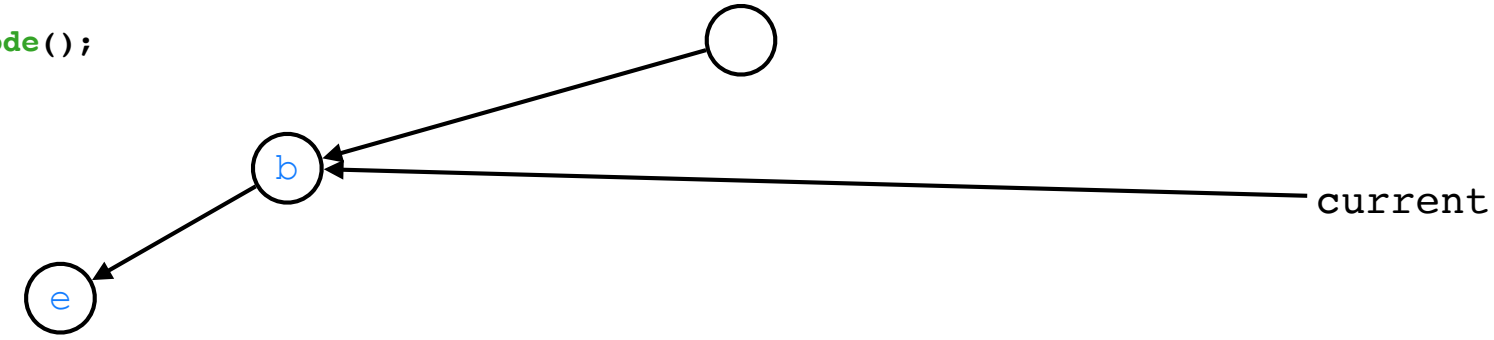
```
root = new TrieNode();
insert( "bear" );
```




```
for( char c : word.toCharArray() ) {
    if( current.children[c] == null ) {
        current.children[c] = new TrieNode();
    }
    current = current.children[c];
}
```

insert("bear")

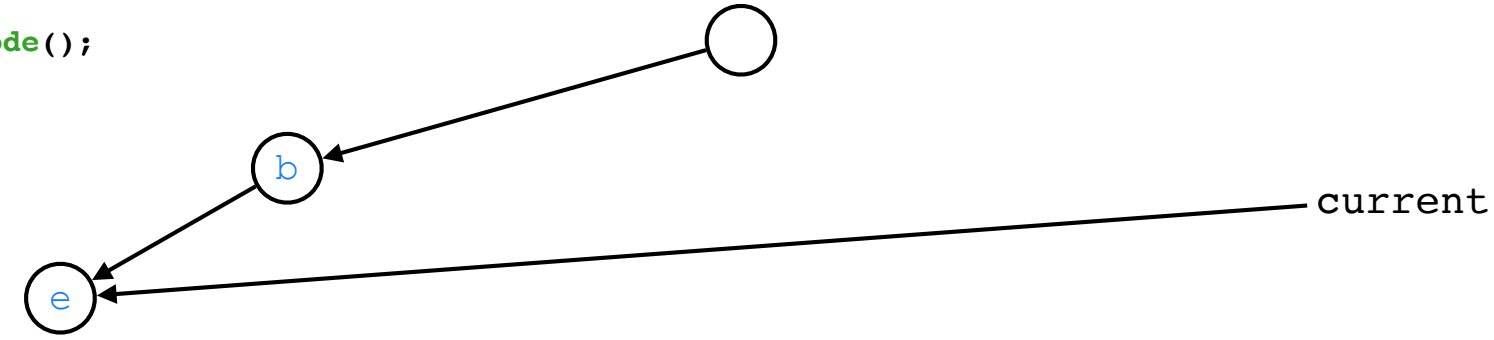
```
root = new TrieNode();
insert( "bear" );
```




```
for( char c : word.toCharArray() ) {
    if( current.children[c] == null ) {
        current.children[c] = new TrieNode();
    }
    current = current.children[c];
}
```

insert("bear")

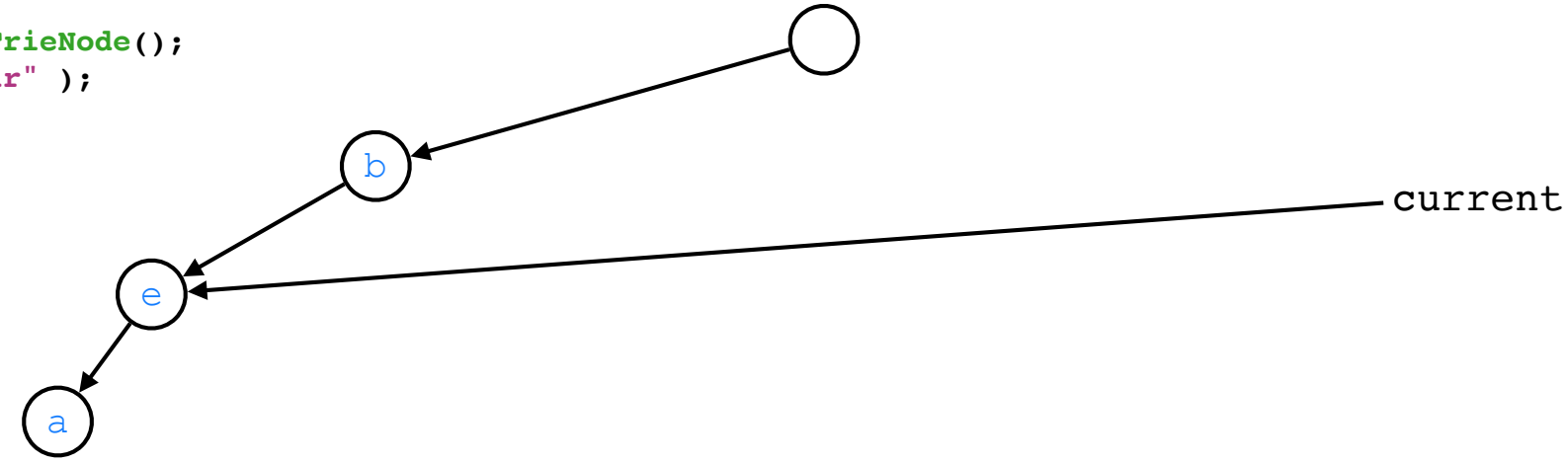
```
root = new TrieNode();
insert( "bear" );
```




```
for( char c : word.toCharArray() ) {
    if( current.children[c] == null ) {
        current.children[c] = new TrieNode();
    }
    current = current.children[c];
}
```

insert("bear")

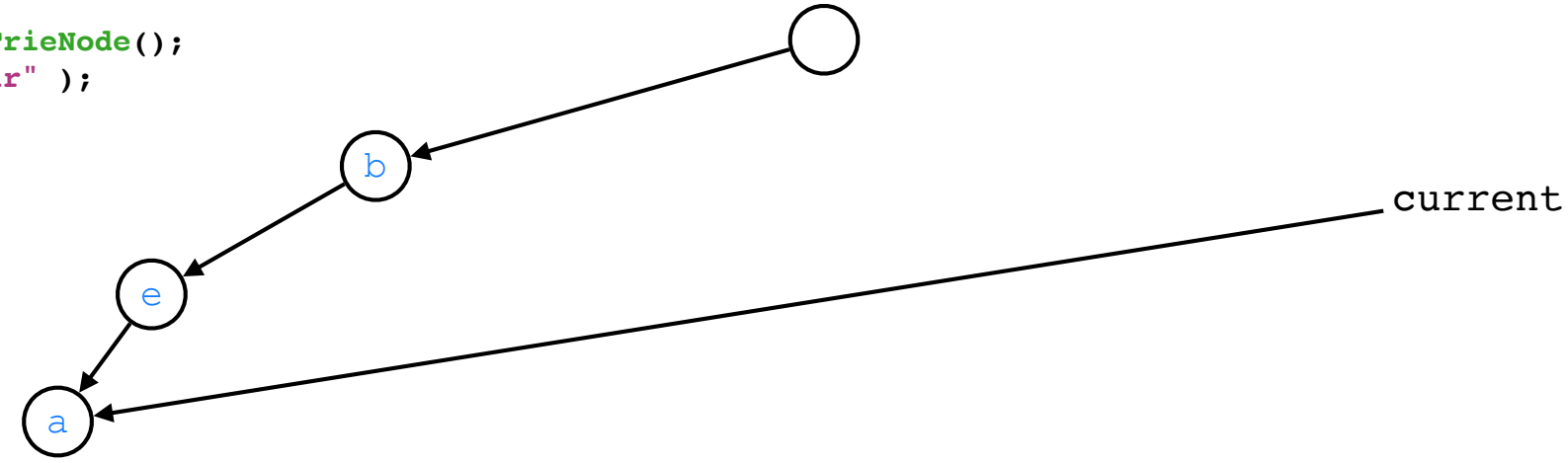
```
root = new TrieNode();
insert( "bear" );
```




```
for( char c : word.toCharArray() ) {
    if( current.children[c] == null ) {
        current.children[c] = new TrieNode();
    }
    current = current.children[c];
}
```

insert("bear")

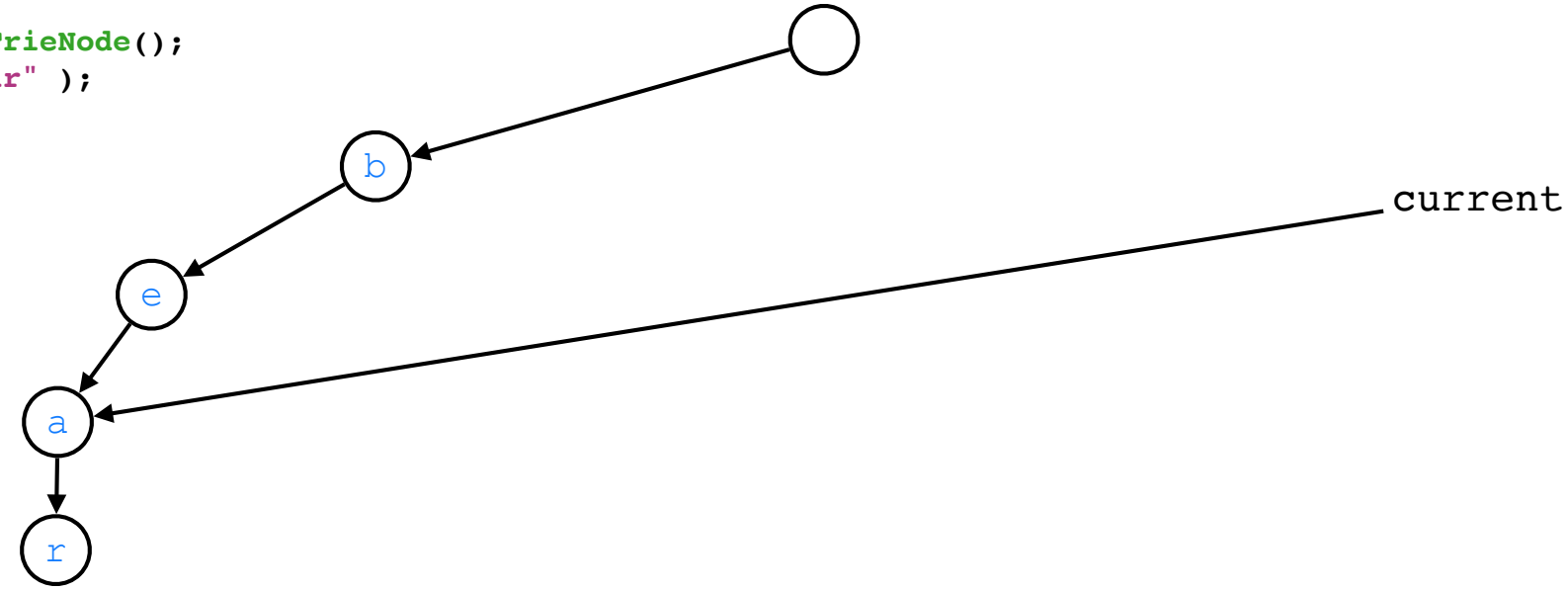
```
root = new TrieNode();
insert( "bear" );
```




```
for( char c : word.toCharArray() ) {
    if( current.children[c] == null ) {
        current.children[c] = new TrieNode();
    }
    current = current.children[c];
}
```

insert("bear")

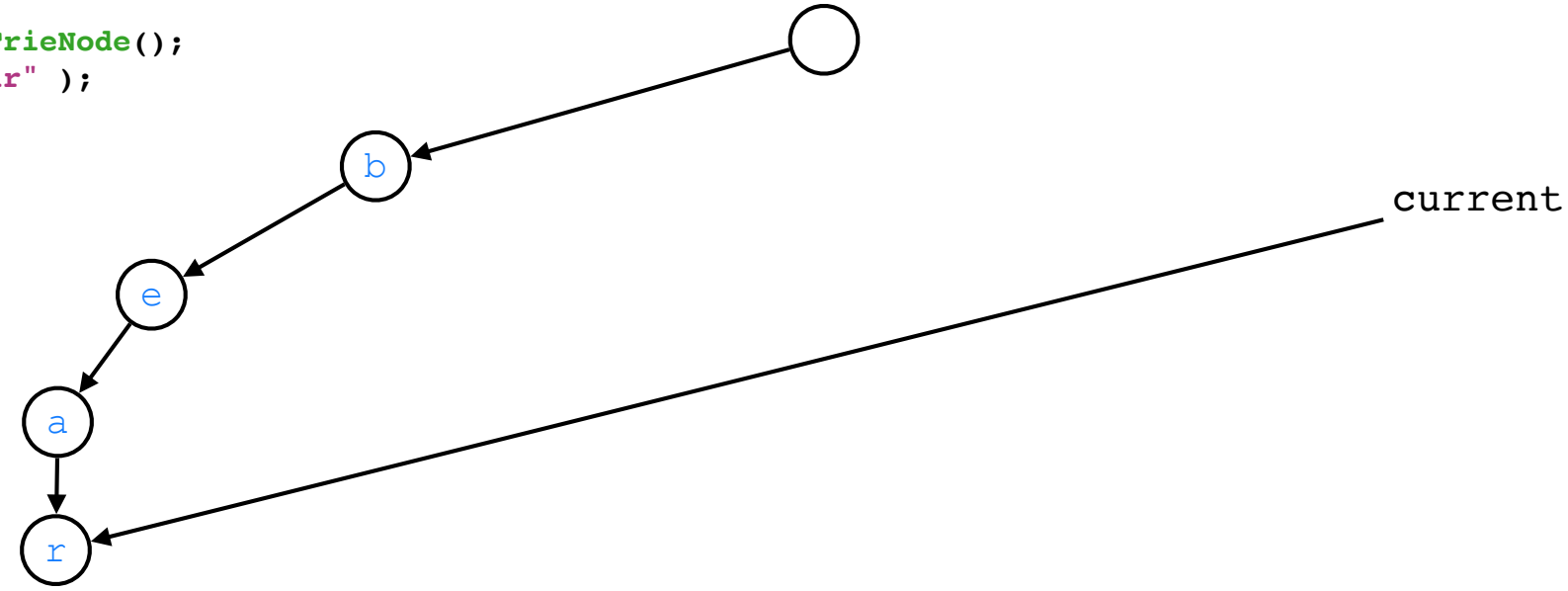
```
root = new TrieNode();
insert( "bear" );
```




```
for( char c : word.toCharArray() ) {
    if( current.children[c] == null ) {
        current.children[c] = new TrieNode();
    }
    current = current.children[c];
}
```

insert("bear")

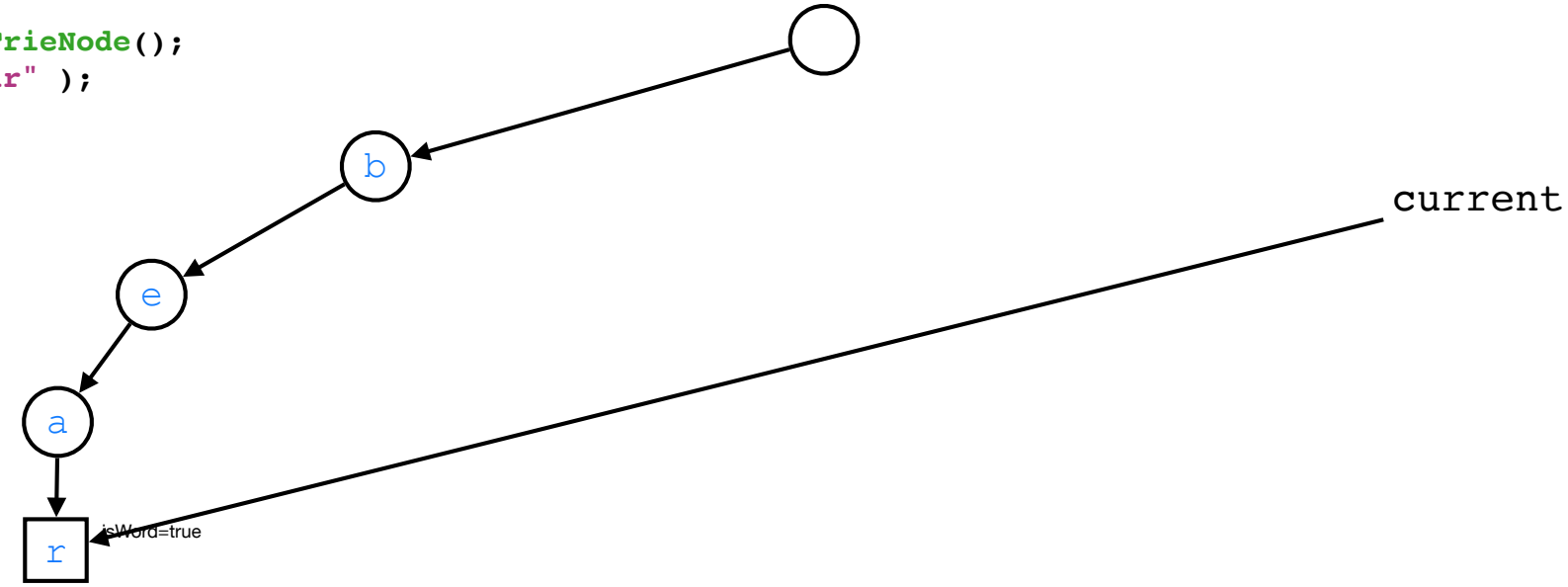
```
root = new TrieNode();
insert( "bear" );
```




```
for( char c : word.toCharArray() ) {
    if( current.children[c] == null ) {
        current.children[c] = new TrieNode();
    }
    current = current.children[c];
}
```

insert("bear")

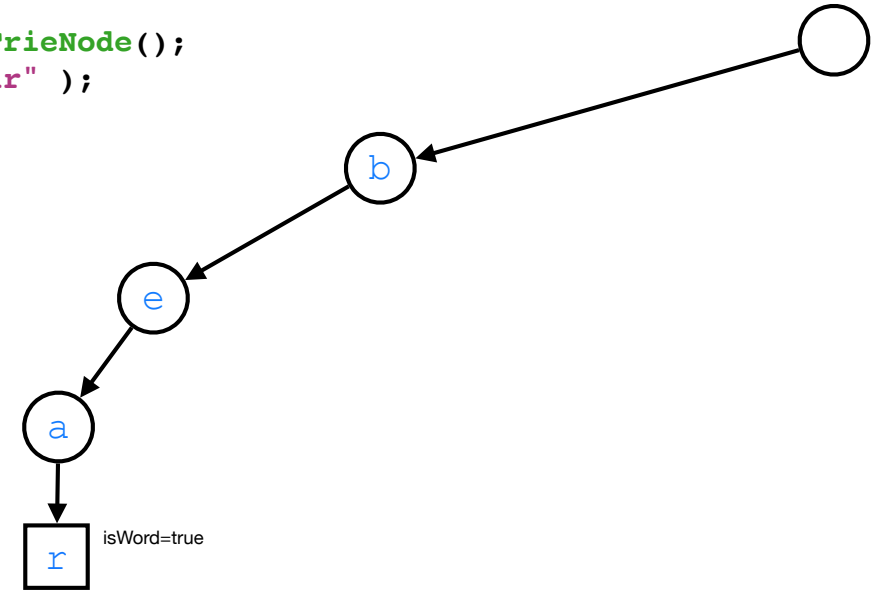
```
root = new TrieNode();
insert( "bear" );
```




```
current.isWord = true;
```

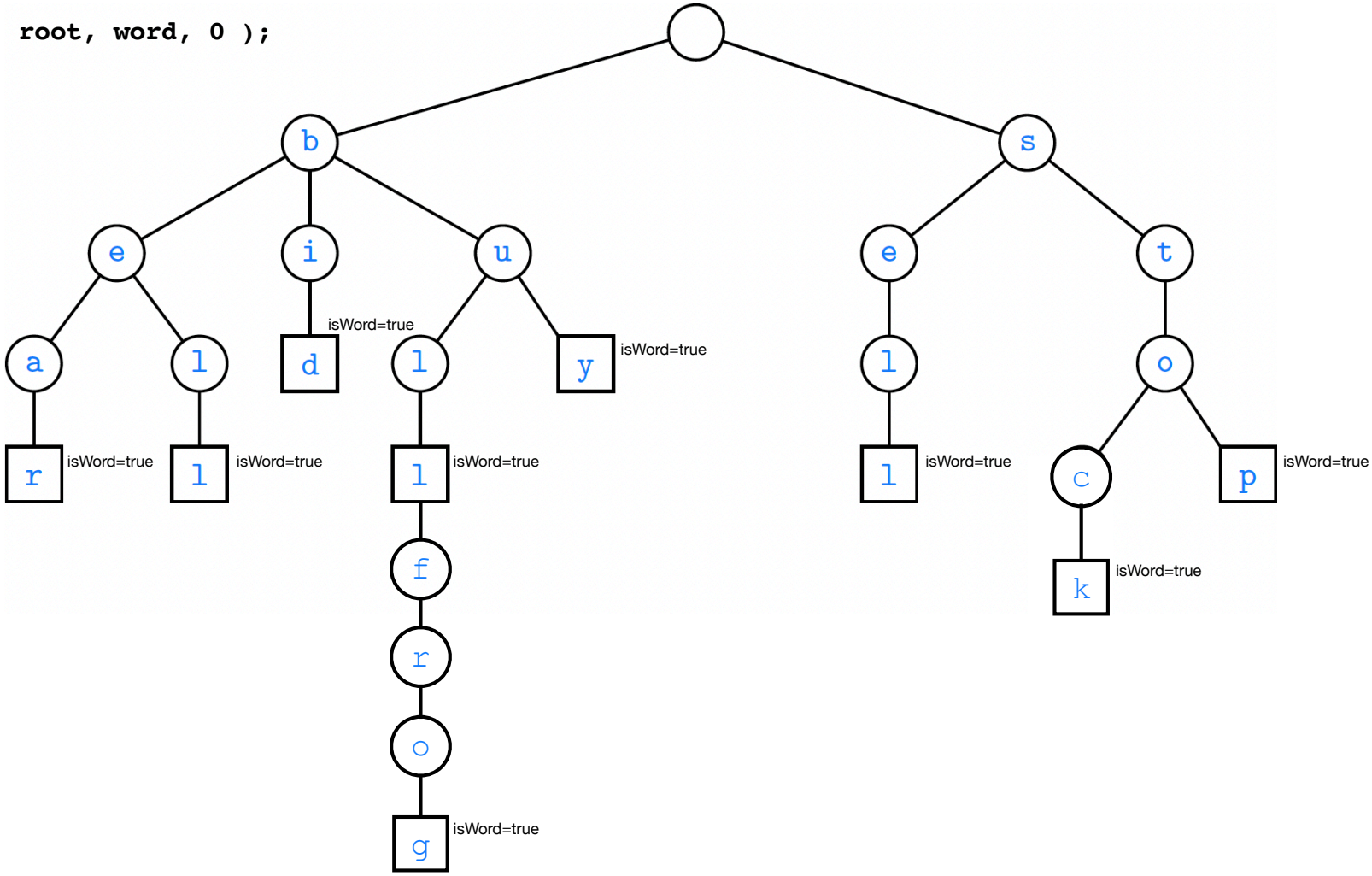
insert("bear")

```
root = new TrieNode();
insert( "bear" );
```

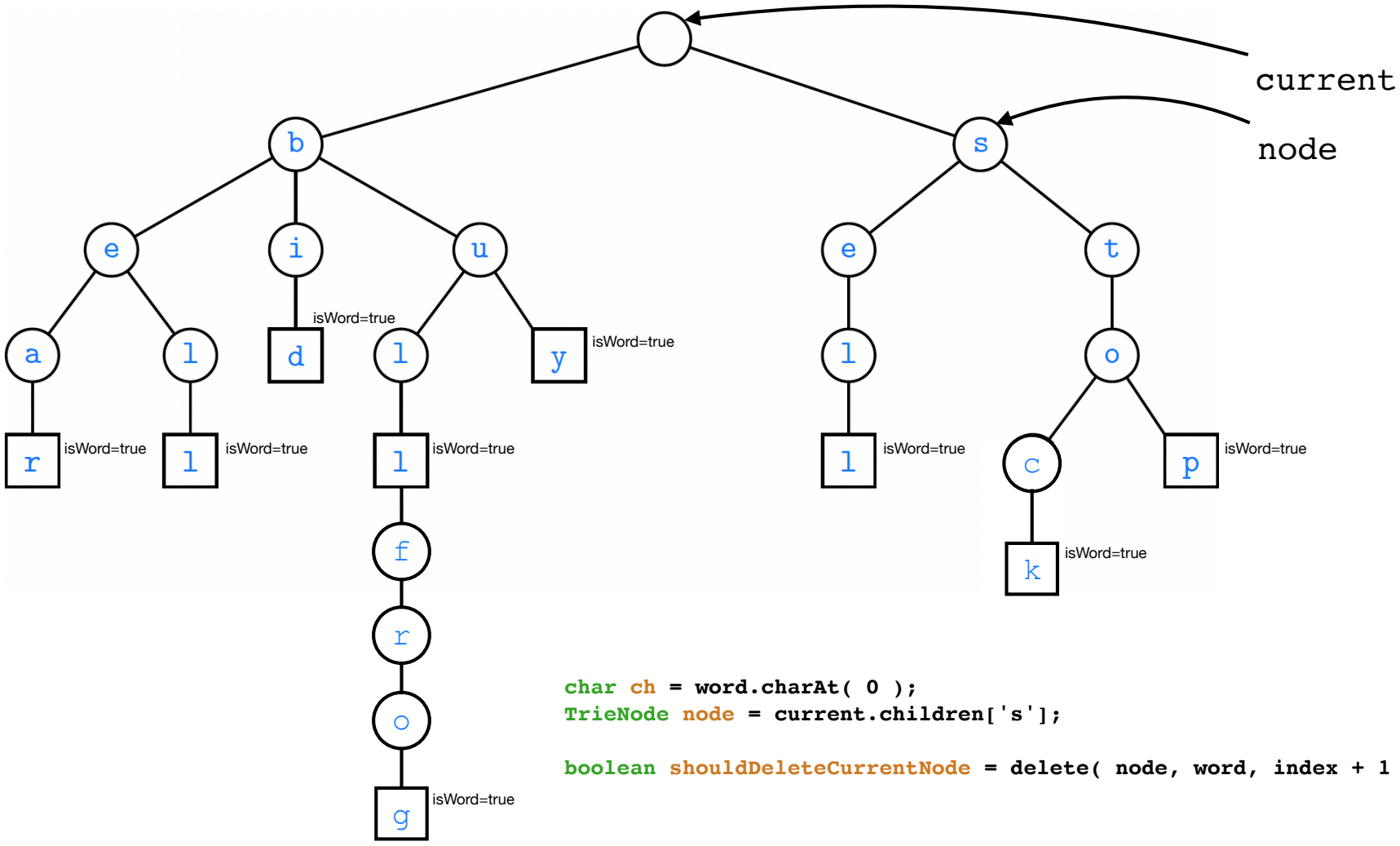


delete("stock")

```
return delete( root, word, 0 );
```

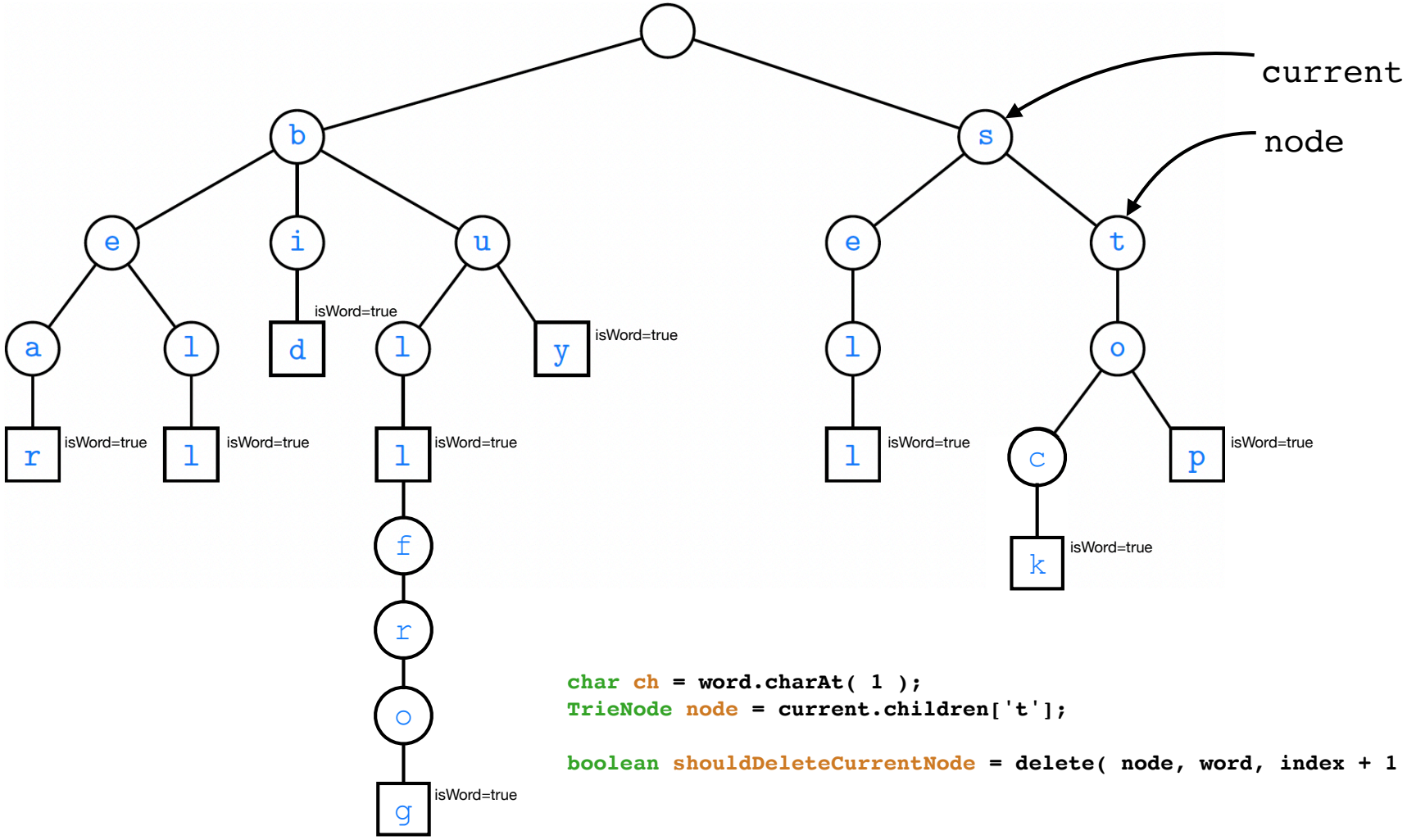


delete("stock")



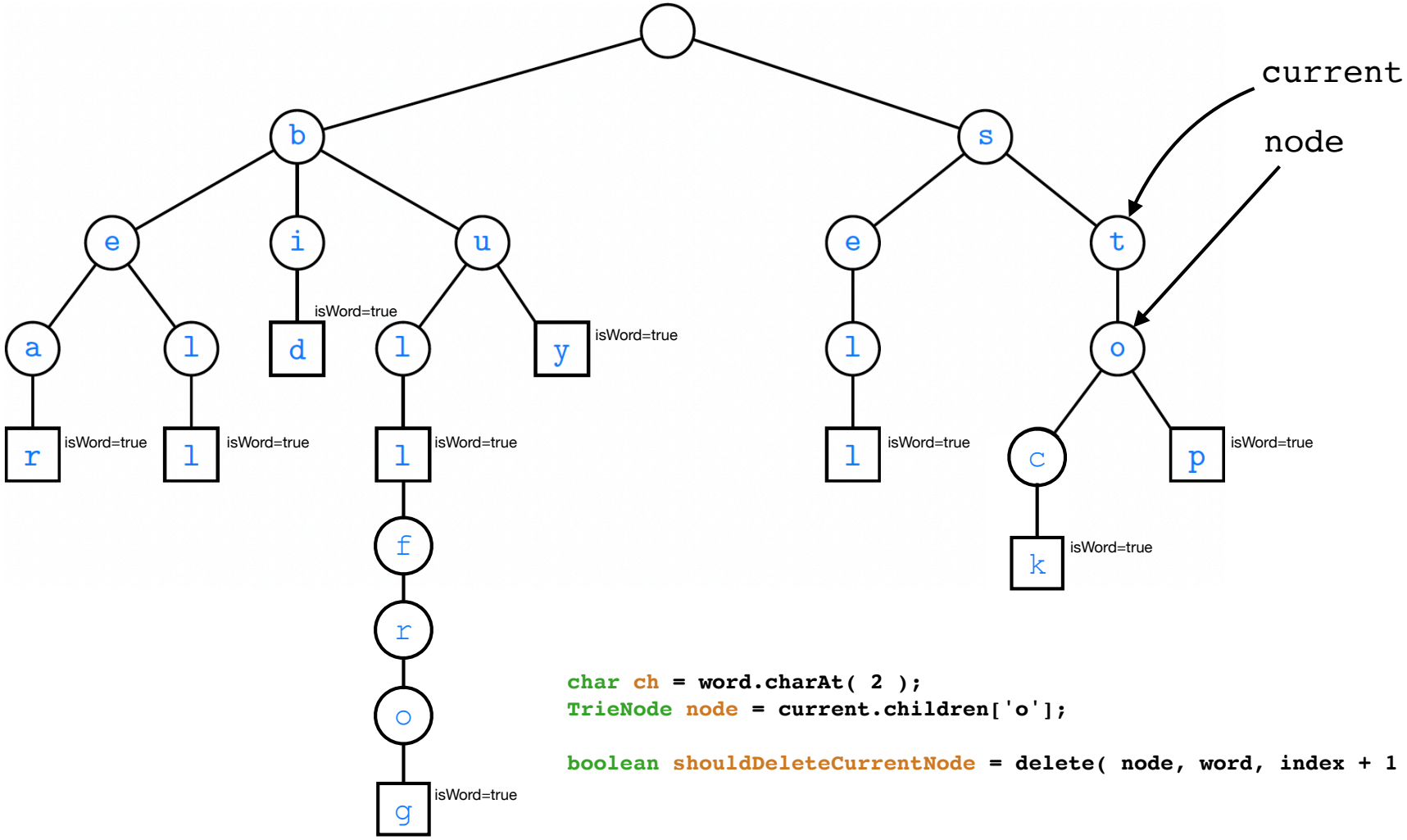

```
char ch = word.charAt( 0 );
TrieNode node = current.children['s'];

boolean shouldDeleteCurrentNode = delete( node, word, index + 1 );
```

delete("stock")



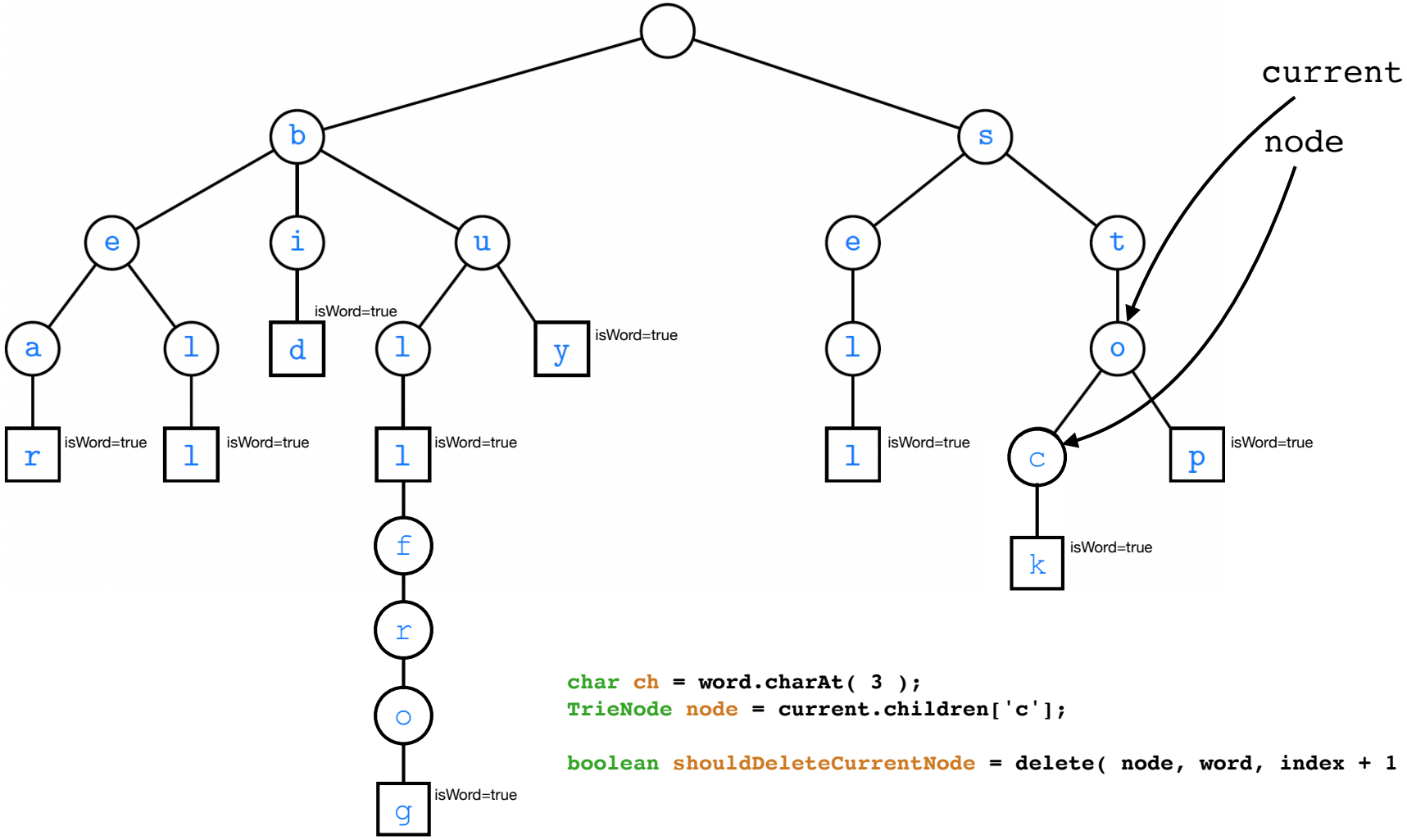

```
char ch = word.charAt( 1 );
TrieNode node = current.children['t'];

boolean shouldDeleteCurrentNode = delete( node, word, index + 1 );
```

delete("stock")



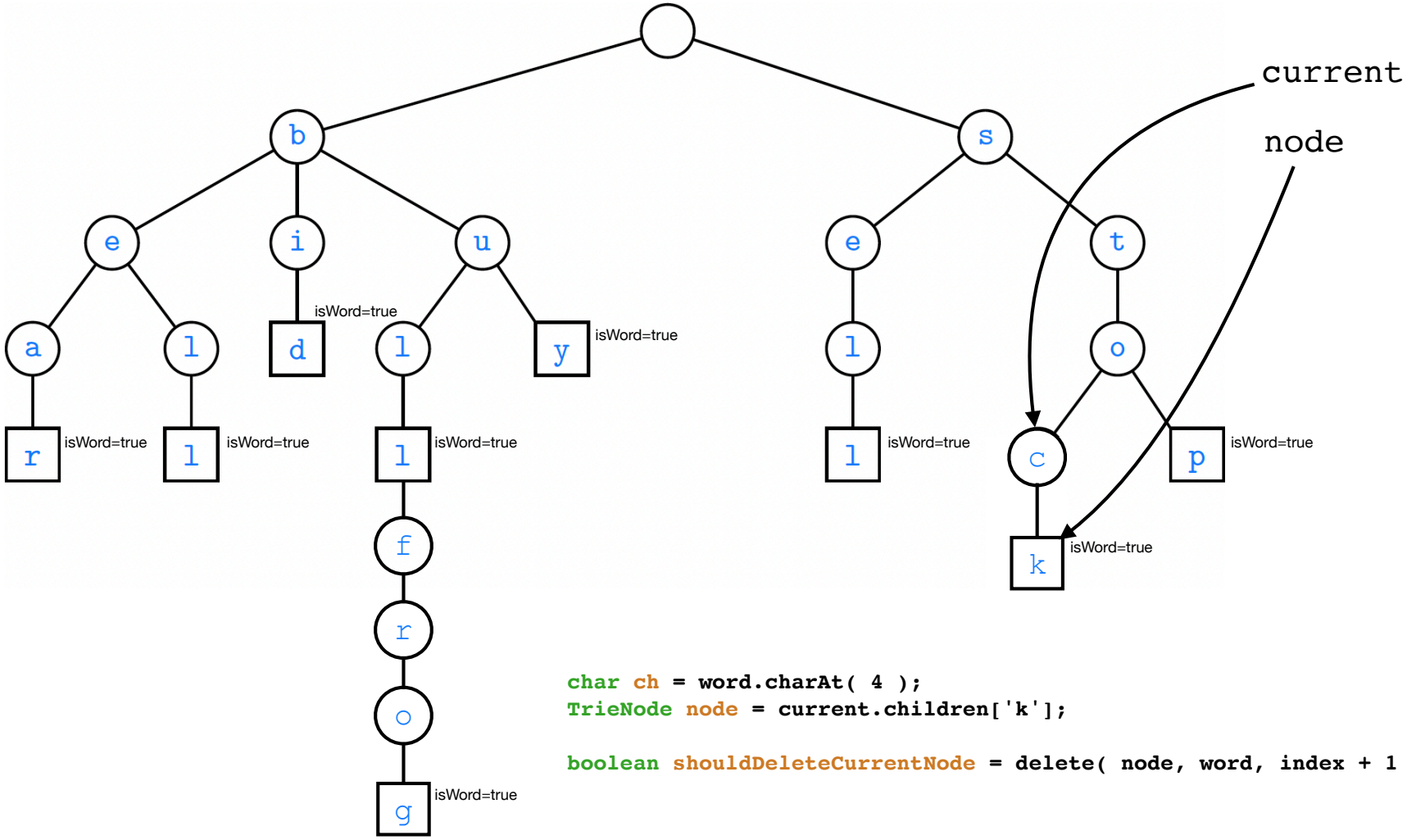

```
char ch = word.charAt( 2 );
TrieNode node = current.children['o'];

boolean shouldDeleteCurrentNode = delete( node, word, index + 1 );
```

delete("stock")



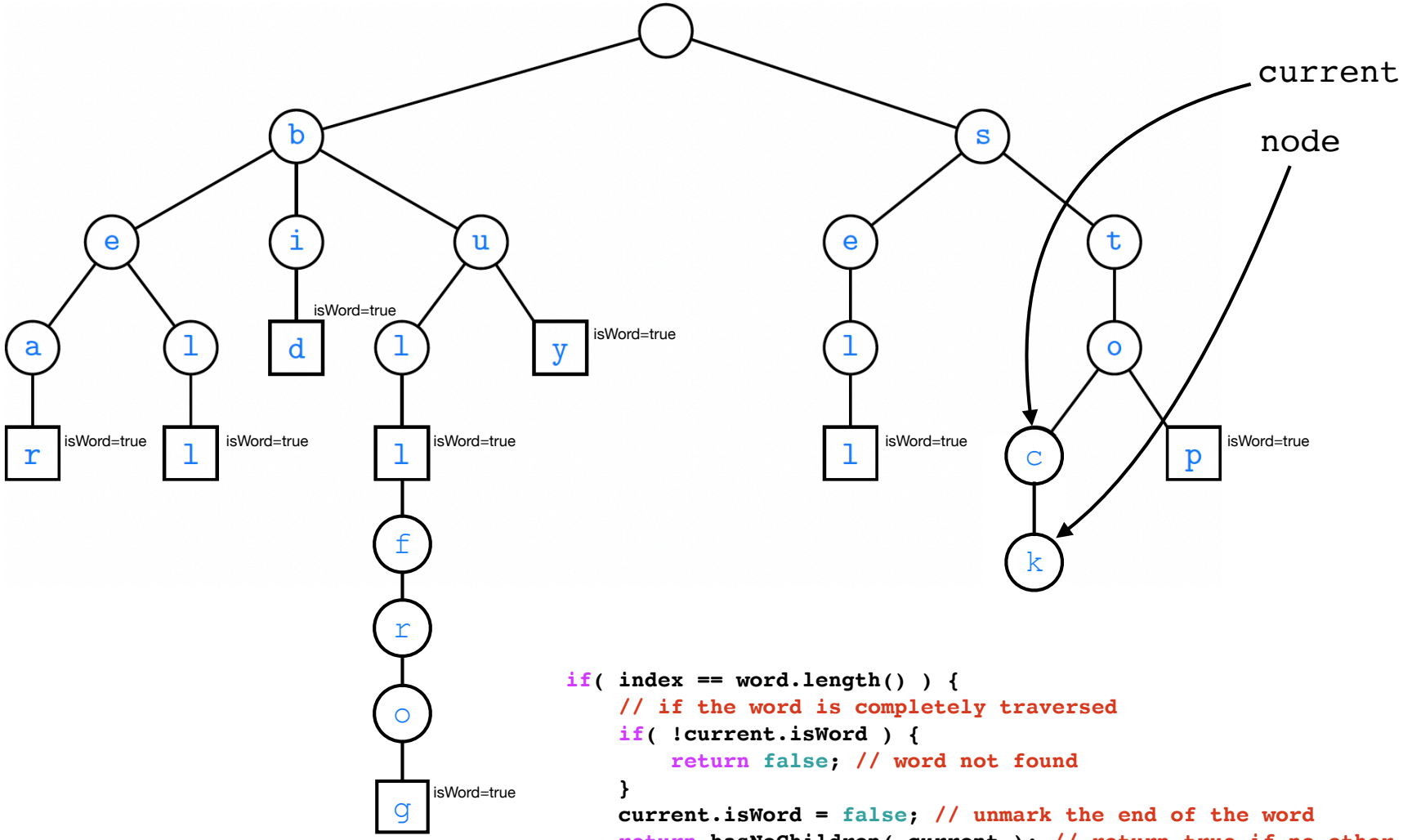

```
char ch = word.charAt( 3 );
TrieNode node = current.children['c'];

boolean shouldDeleteCurrentNode = delete( node, word, index + 1 );
```

delete("stock")



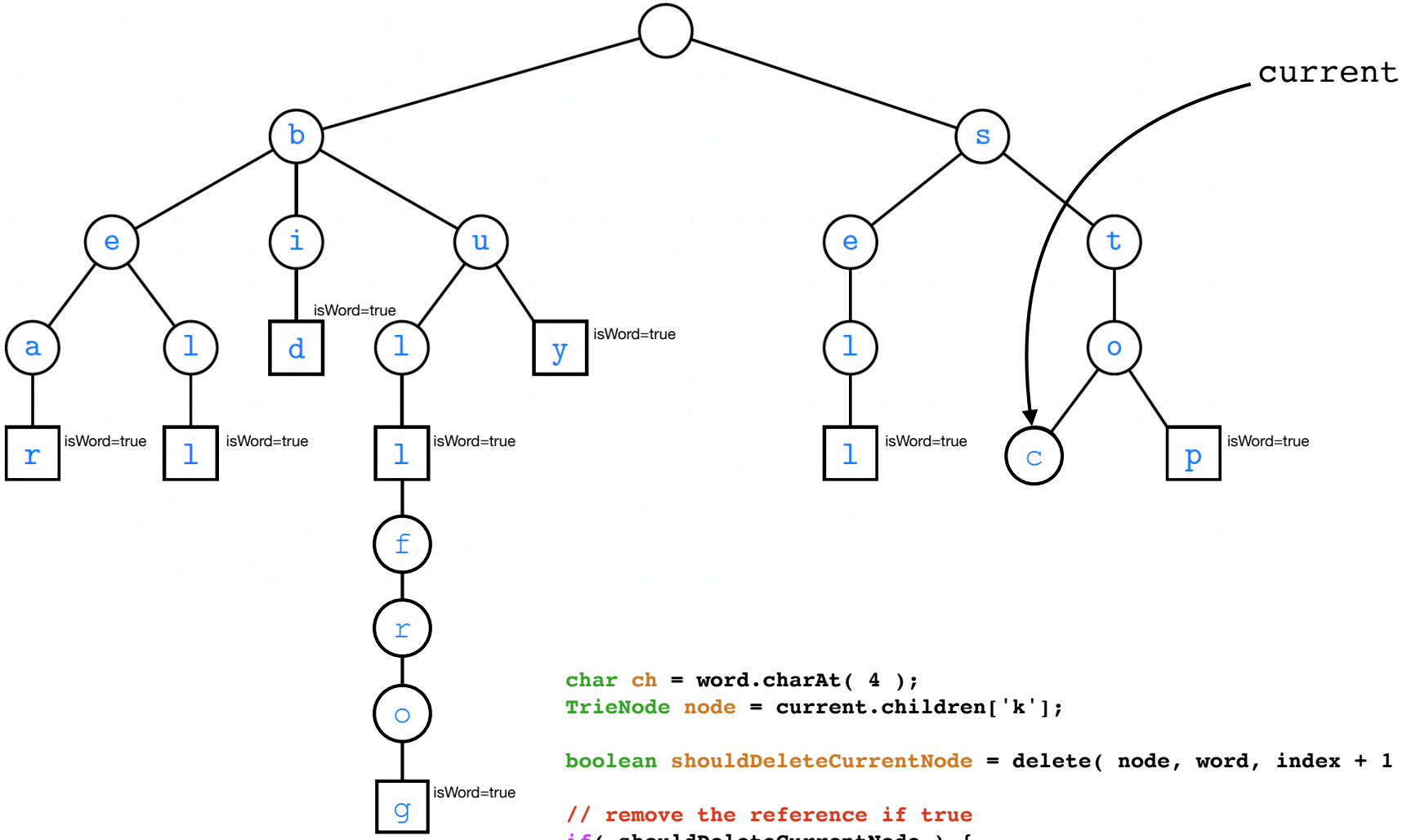

```
char ch = word.charAt( 4 );
TrieNode node = current.children['k'];

boolean shouldDeleteCurrentNode = delete( node, word, index + 1 );
```

delete("stock")



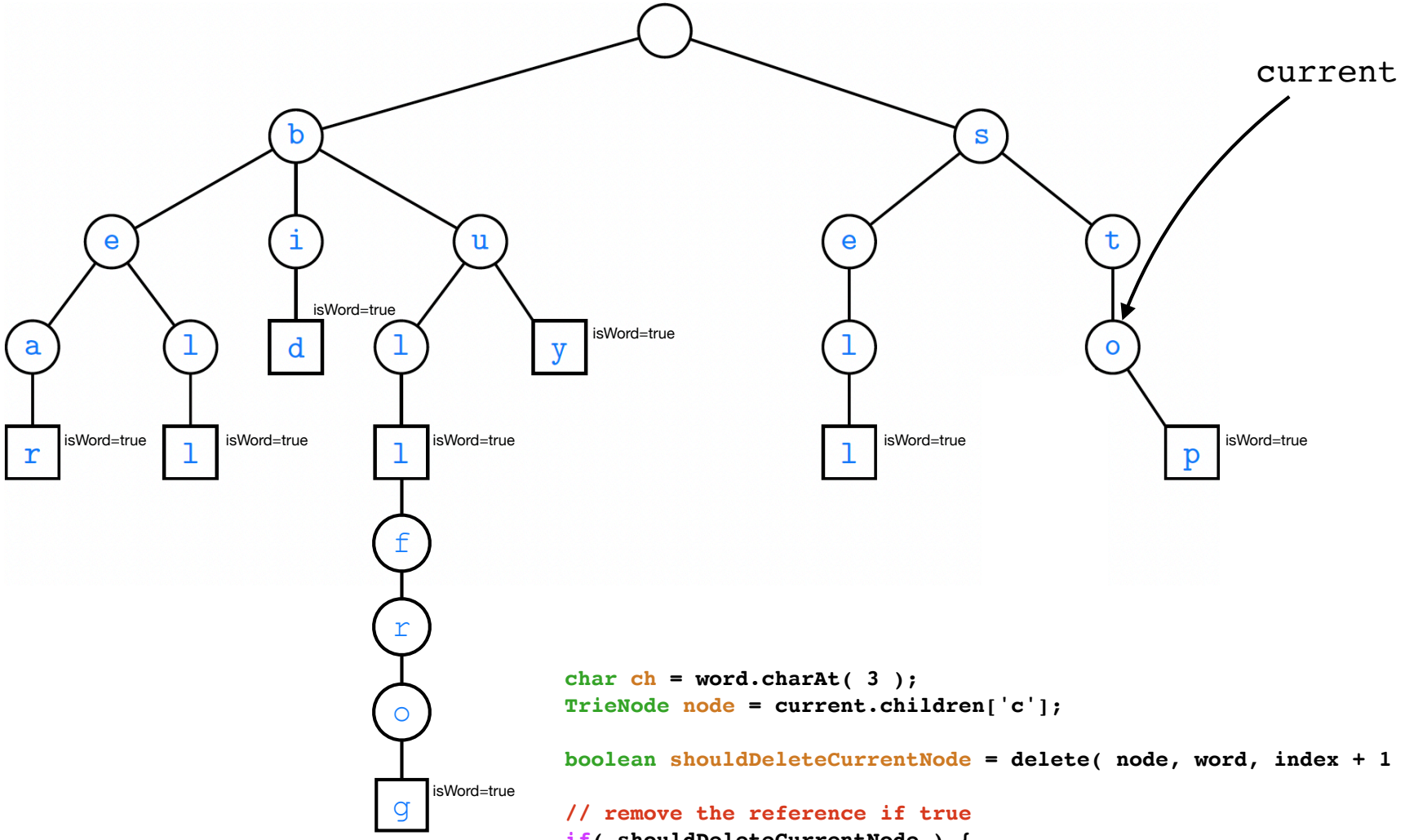

```
if( index == word.length() ) {
    // if the word is completely traversed
    if( !current.isWord ) {
        return false; // word not found
    }
    current.isWord = false; // unmark the end of the word
    return hasNoChildren( current ); // return true if no other
                                     // branches depend on this
}
```

delete("stock")



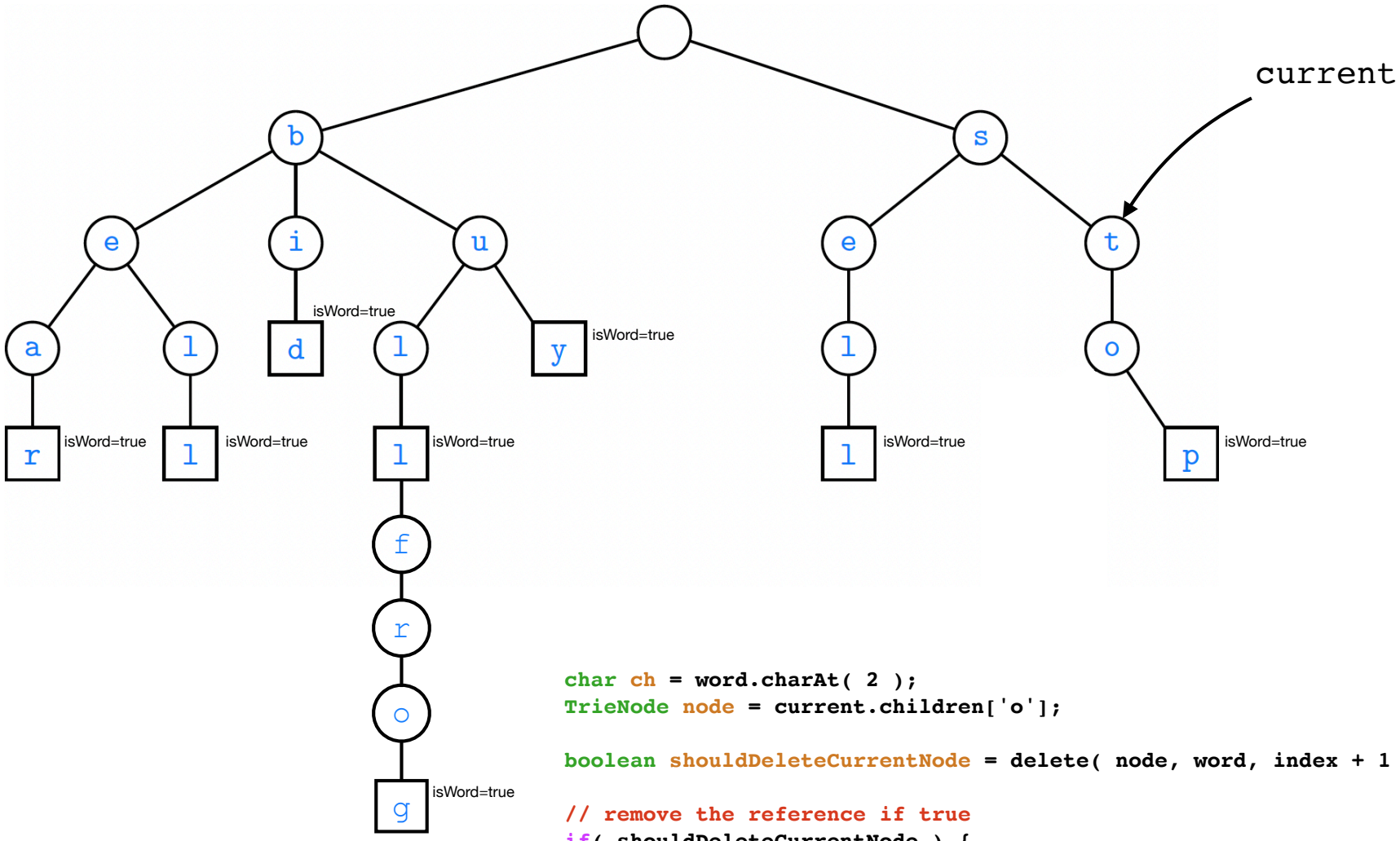

```
char ch = word.charAt( 4 );
TrieNode node = current.children['k'];

boolean shouldDeleteCurrentNode = delete( node, word, index + 1 );

// remove the reference if true
if( shouldDeleteCurrentNode ) {
    current.children[ch] = null;
    // return true if no other children are dependent on this node
    return hasNoChildren( current );
}
```

delete("stock")



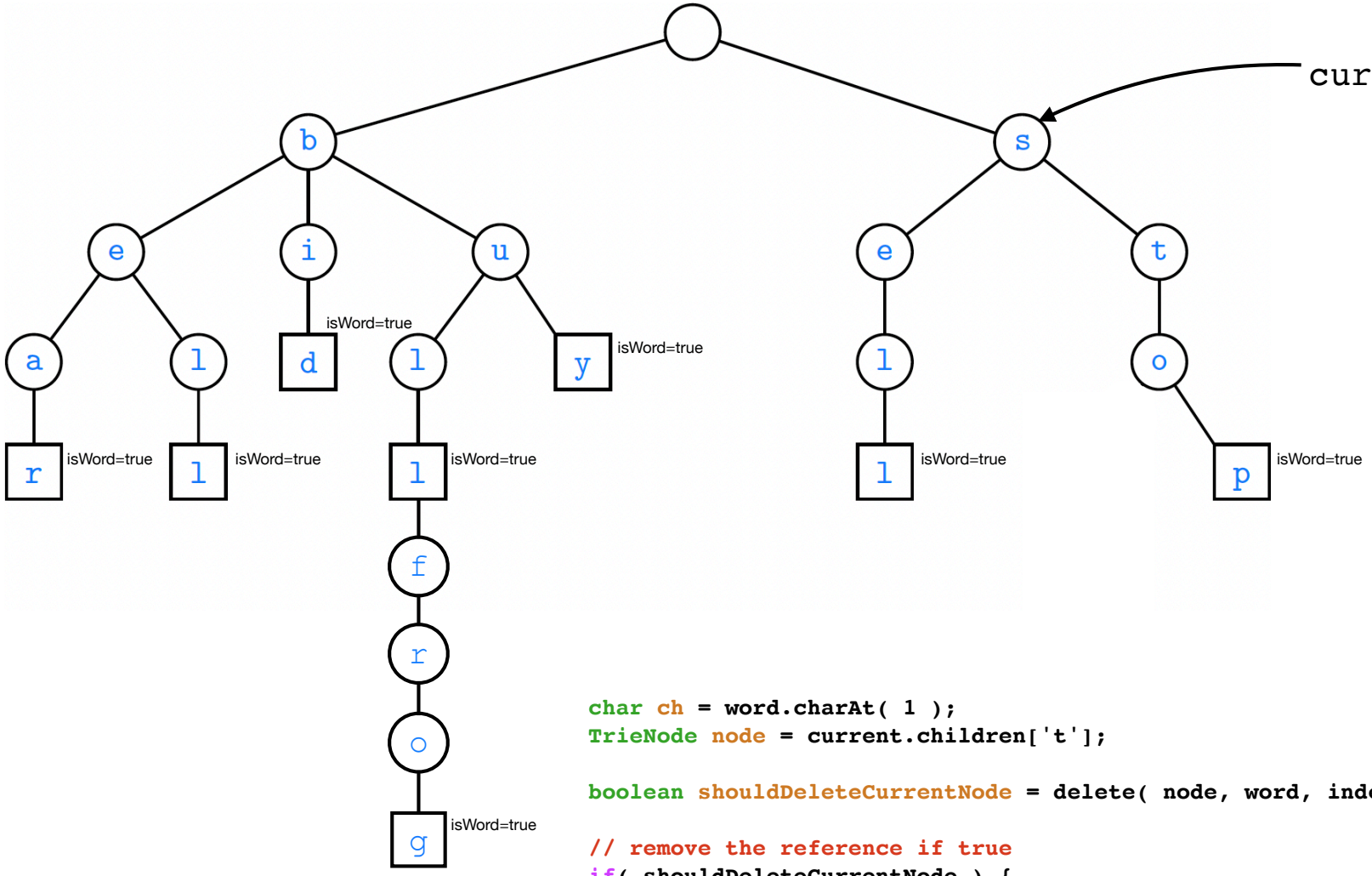

```
char ch = word.charAt( 3 );
TrieNode node = current.children['c'];

boolean shouldDeleteCurrentNode = delete( node, word, index + 1 );

// remove the reference if true
if( shouldDeleteCurrentNode ) {
    current.children[ch] = null;
    // return true if no other children are dependent on this node
    return hasNoChildren( current );
}
```

delete("stock")



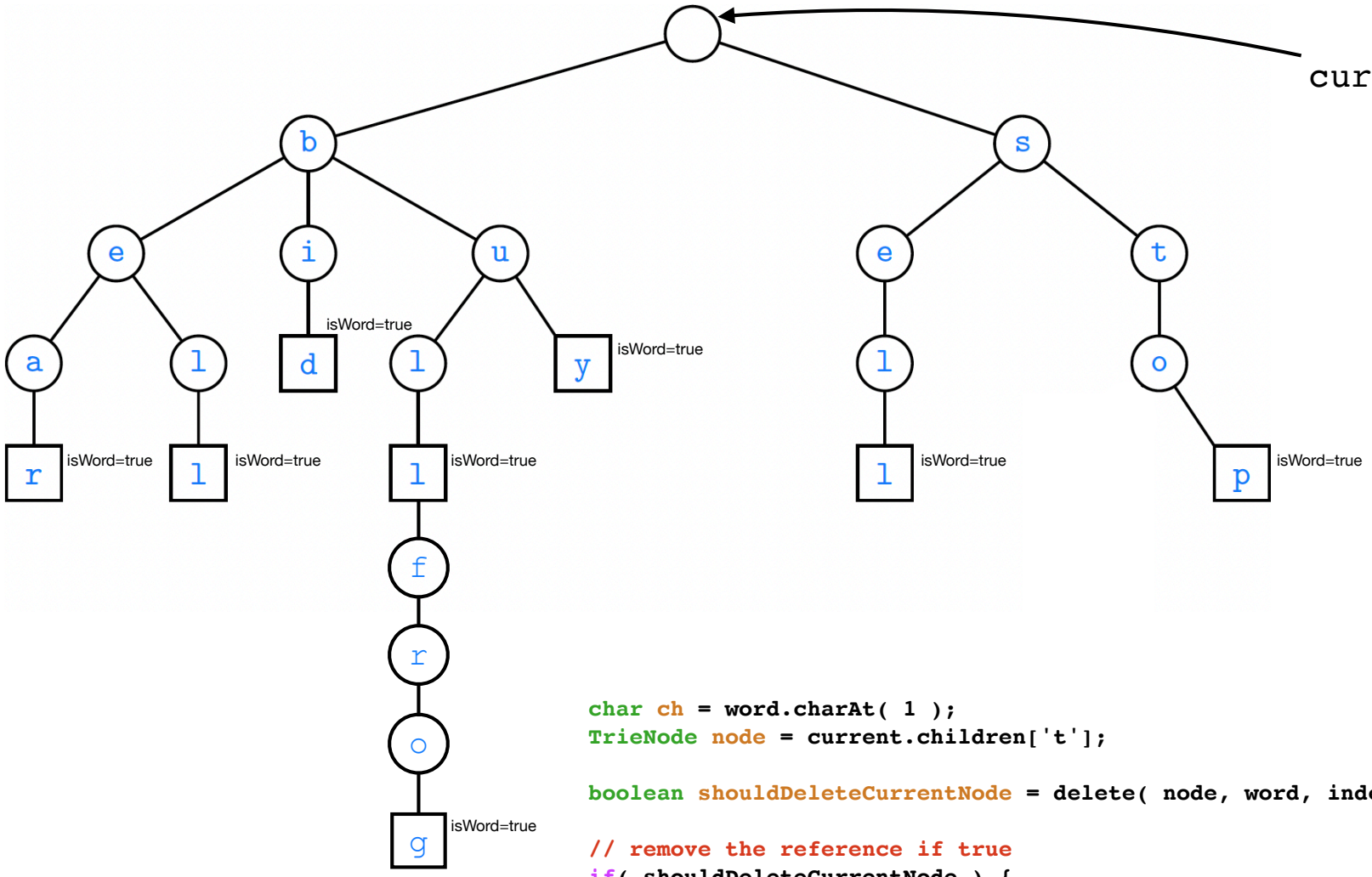

```
char ch = word.charAt( 2 );
TrieNode node = current.children['o'];

boolean shouldDeleteCurrentNode = delete( node, word, index + 1 );

// remove the reference if true
if( shouldDeleteCurrentNode ) {
    current.children[ch] = null;
    // return true if no other children are dependent on this node
    return hasNoChildren( current );
}
return false;
```

delete("stock")



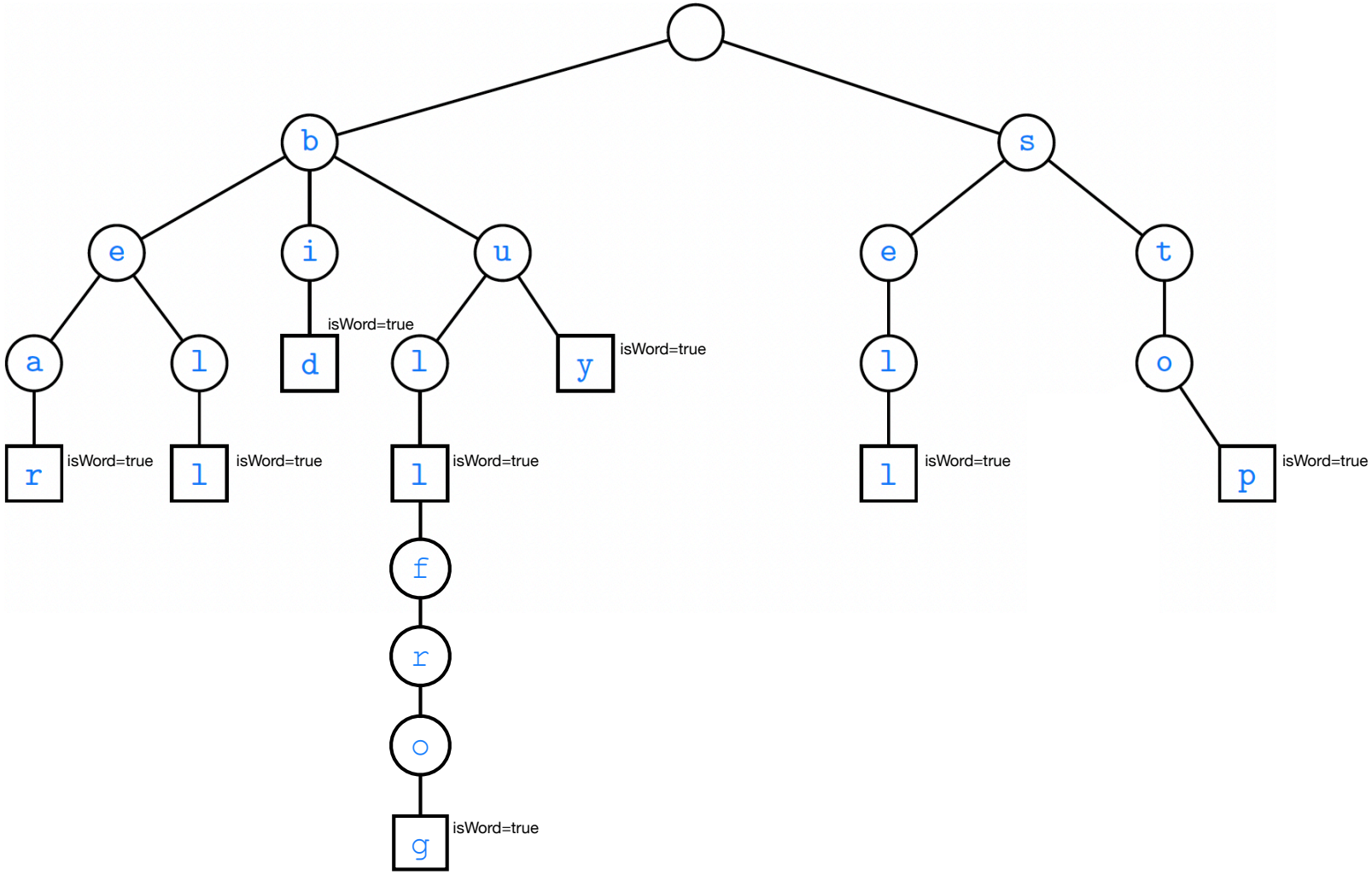

```
char ch = word.charAt( 1 );
TrieNode node = current.children['t'];

boolean shouldDeleteCurrentNode = delete( node, word, index + 1 );

// remove the reference if true
if( shouldDeleteCurrentNode ) {
    current.children[ch] = null;
    // return true if no other children are dependent on this node
    return hasNoChildren( current );
}
return false;
```

delete("stock")




```
char ch = word.charAt( 1 );
TrieNode node = current.children['t'];

boolean shouldDeleteCurrentNode = delete( node, word, index + 1 );

// remove the reference if true
if( shouldDeleteCurrentNode ) {
    current.children[ch] = null;
    // return true if no other children are dependent on this node
    return hasNoChildren( current );
}
return false;
```

delete("stock")



Conclusions du module

- Nous avons exploré la structure récursive d'arbre général
- Nous avons regardé la terminologie utilisée pour décrire ses noeuds, leurs relations et leurs propriétés
- Nous avons défini l'interface pour un arbre général (`Tree`) et une classe abstraite pour commencer son implémentation (`AbstractTree`)
- Nous avons implémenté une méthode pour déterminer la profondeur d'un noeud
- Nous avons implémenté 2 méthodes pour déterminer la hauteur d'un arbre (dont une très mauvaise !)
- Nous avons utilisé l'interface `Position`, comme nous l'avions fait pour la liste positionnelle
- Nous avons exploré le `Trie`, une structure qui utilise un arbre général pour préprocesser du texte