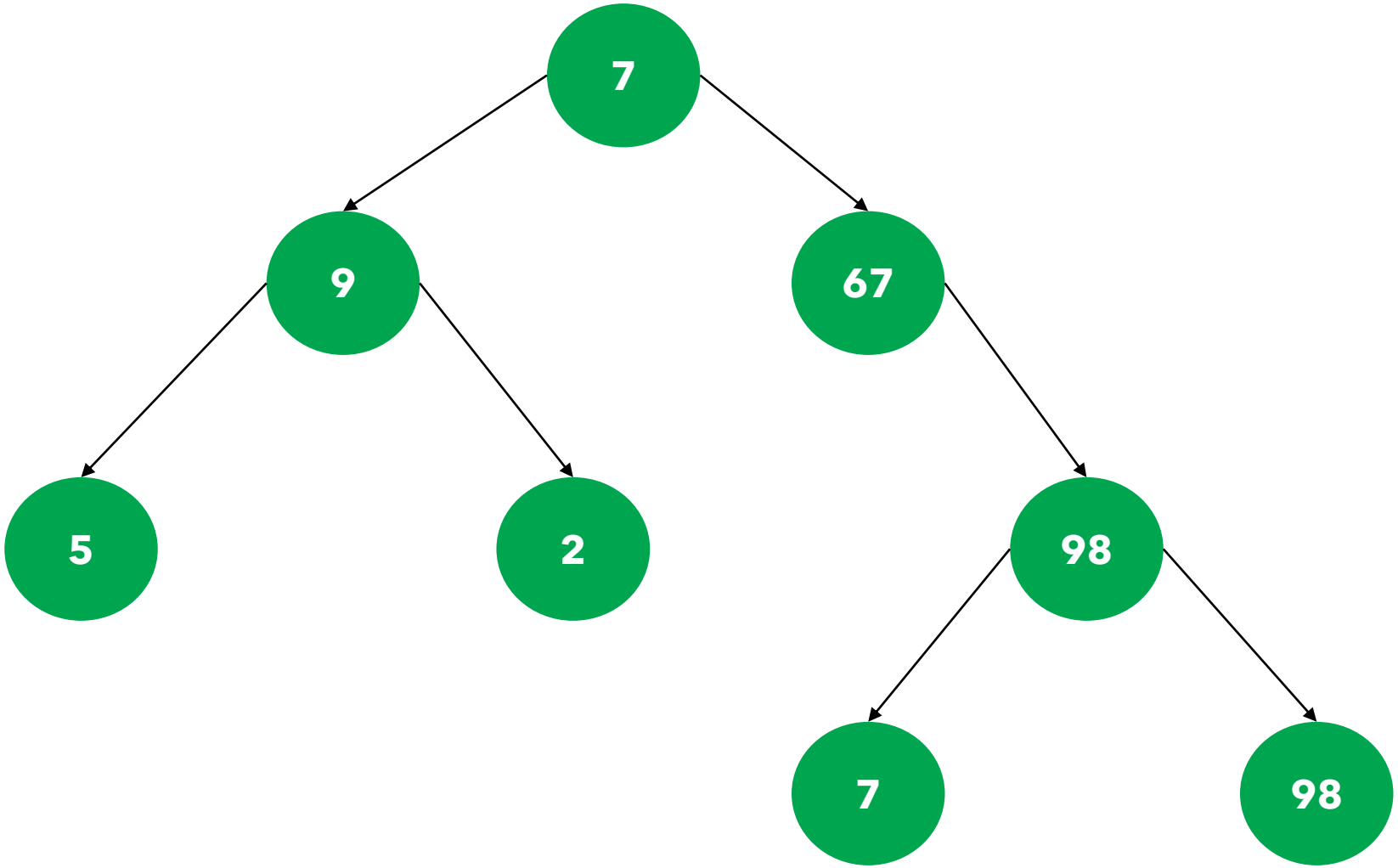
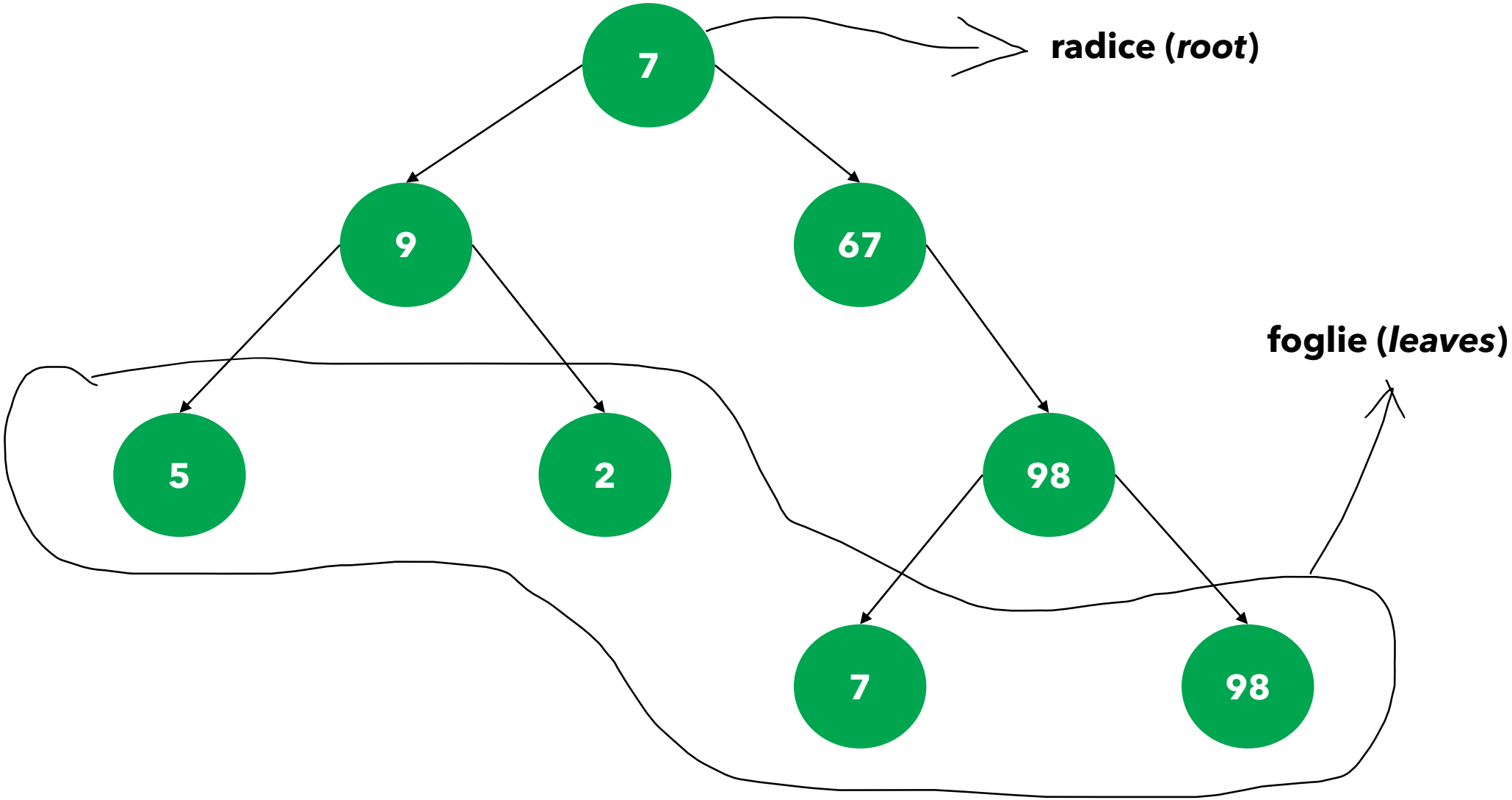
Alberi binari (*binary trees*)

Liceo G.B. Brocchi
Classi quarte Scientifico – opzione scienze applicate
Bassano del Grappa, Dicembre 2022
Prof. Giovanni Mazzocchin

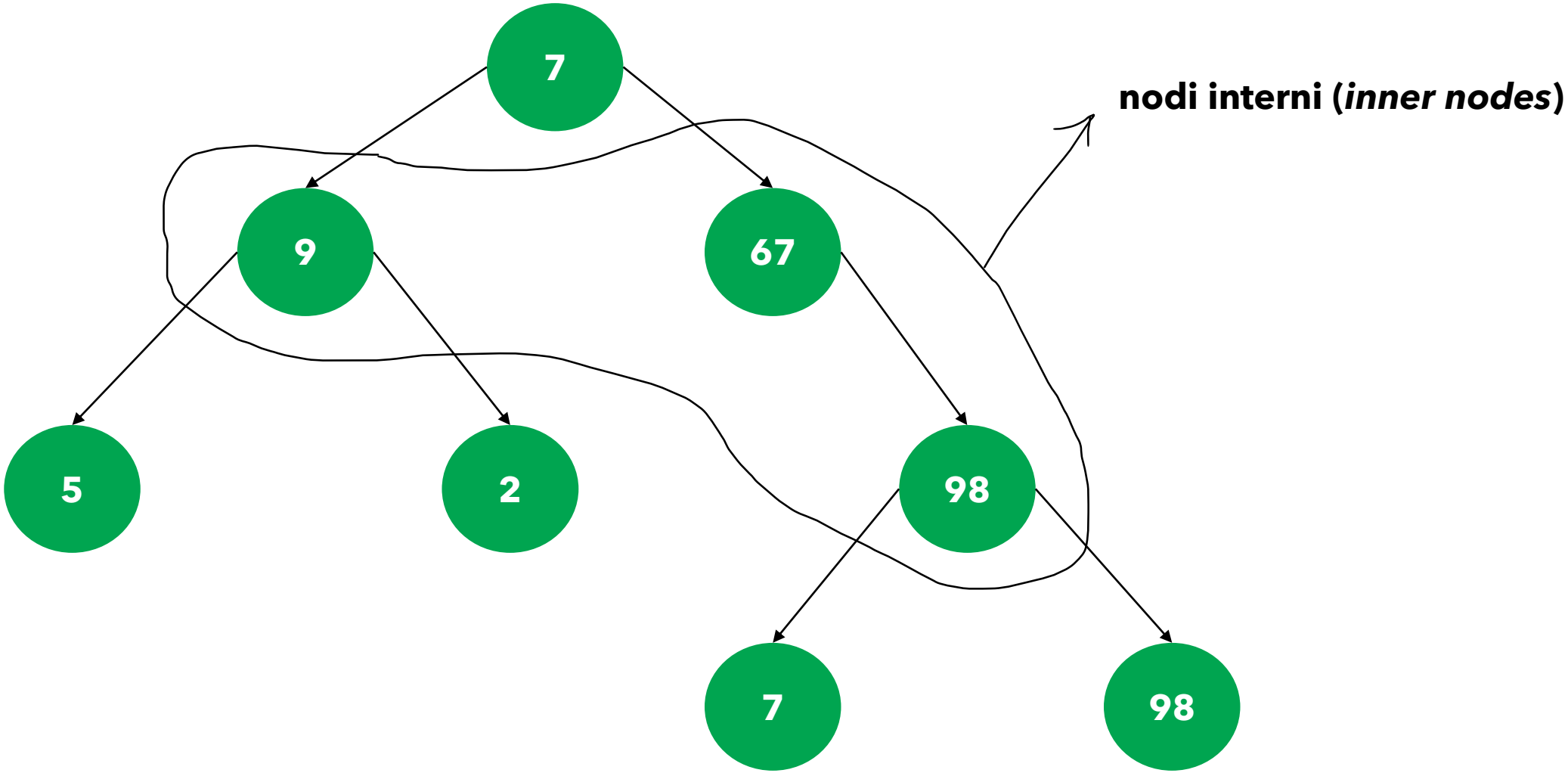
La struttura che vogliamo realizzare (*rooted tree*)



La struttura che vogliamo realizzare (*rooted tree*)



La struttura che vogliamo realizzare (*rooted tree*)



Definizione ricorsiva

Un **albero binario** è:

- un albero senza alcun nodo

oppure

- un nodo che punta a due **alberi binari**

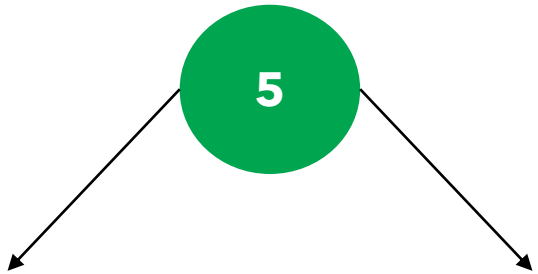
Recuperare la definizione ricorsiva di lista concatenata
Il caso «vuoto» servirà come caso base per le funzioni ricorsive, come per le liste

Implementazione tramite puntatori ai figli

```
typedef struct tree_node {
  int key;
  struct tree_node* left;
  struct tree_node* right;
} T_NODE;
```

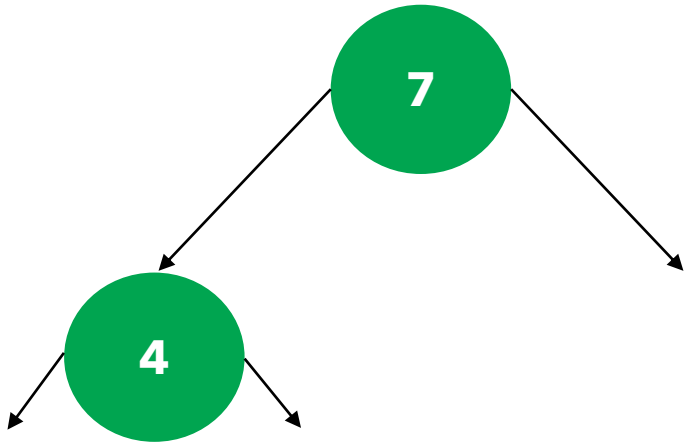
Per rappresentare alberi n-ari (in cui ogni nodo ha fino a n figli) il principio è lo stesso. Chi studierà informatica vedrà anche altre rappresentazioni più efficienti in termini di memoria occupata

Allocazione in memoria di un albero



leftChild e *rightChild* sono riferimenti con valore *null*. Significa che non puntano ad alcun oggetto in memoria

Allocazione in memoria di un albero



leftChild e *rightChild* sono riferimenti con valore *null*. Significa che non puntano ad alcun oggetto in memoria

Alberi binari di ricerca (*binary search trees*)

- **Binary-search-tree property**

  se ***n*** è un nodo di un albero binario di ricerca, ***n.key*** è la chiave di *n*, ***n.left*** è la radice del sottoalbero sinistro di *n*, e ***n.right*** è la radice del sottoalbero destro di *n*, allora:

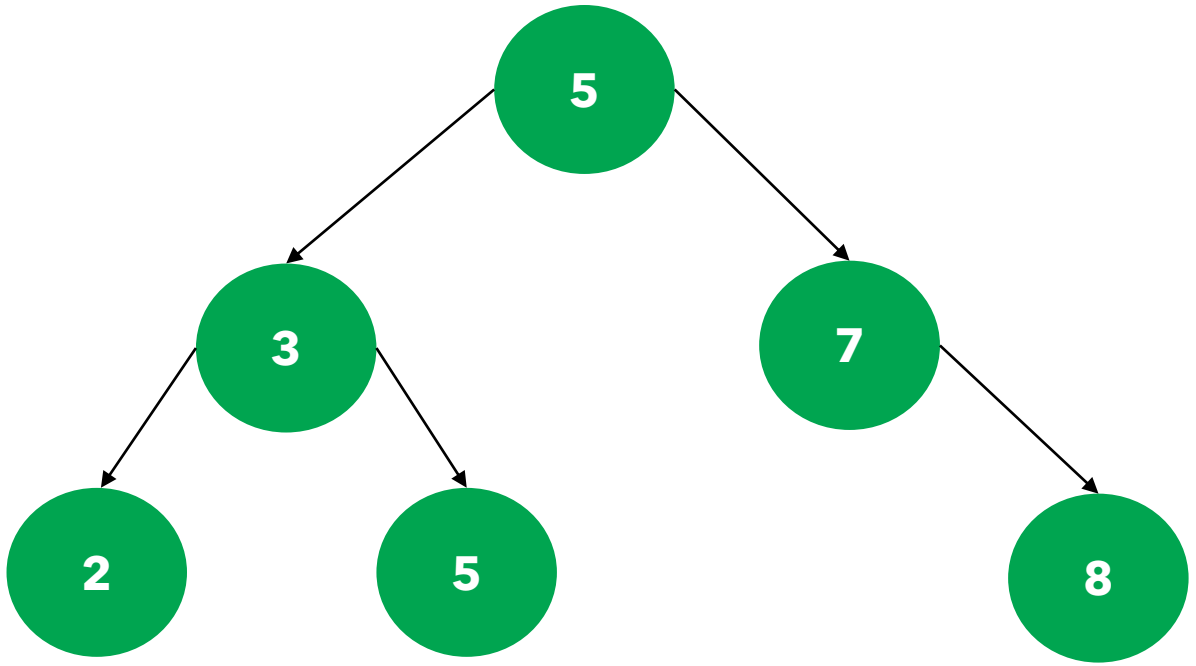
  per ogni nodo ***nl*** *del sottoalbero radicato in* ***n.left*** *è vero che* ***nl.key <= n.key***

  per ogni nodo ***nr*** *del sottoalbero radicato in* ***n.right*** *è vero che* ***n.key < nr.key***

  Una struttura del genere è molto utile per ricercare informazioni.

  L'albero che salta fuori quando si analizza la ricerca binaria è un albero binario di ricerca, ma è solo logico, non viene veramente allocato in memoria

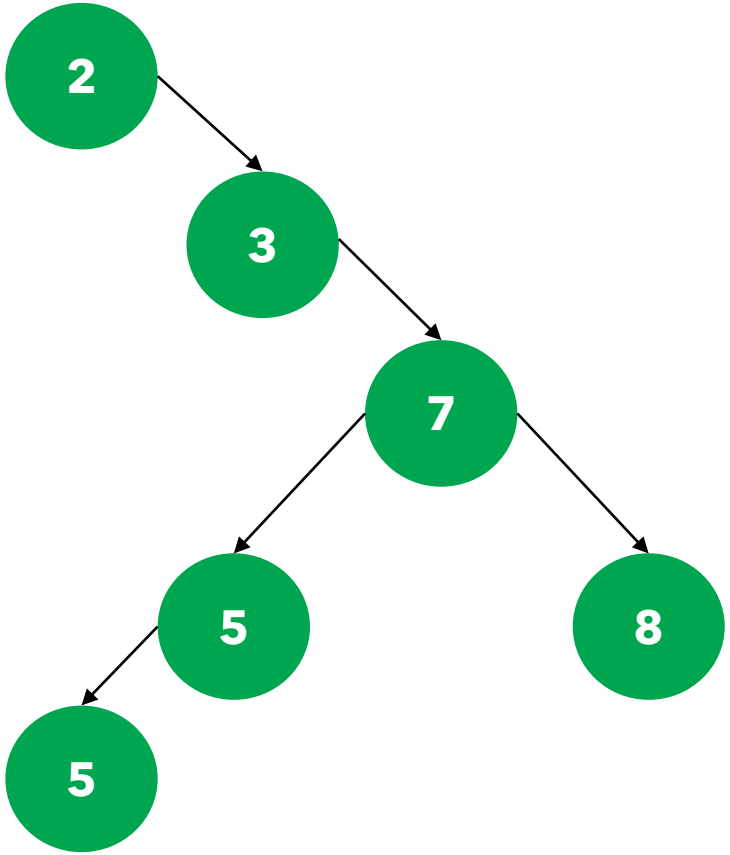
Alberi binari di ricerca (*binary search trees*)



Questo albero ha **altezza 2**: l'altezza di un albero binario è la distanza del percorso più lungo dalla radice ad una foglia. In questo caso abbiamo 3 percorsi radice foglia di lunghezza 2:

- *5 -> 3 -> 2 (lunghezza 2: la lunghezza del percorso è il numero delle frecce)*
- *5 -> 3 -> 5 (lunghezza 2: la lunghezza del percorso è il numero delle frecce)*
- *5 -> 7 -> 8 (lunghezza 2: la lunghezza del percorso è il numero delle frecce)*

Alberi binari di ricerca (*binary search trees*)



Questo albero ha le stesse chiavi del precedente, ma ha **altezza 4**:

- *2 -> 3 -> 7 -> 8: percorso di lunghezza 3*
- *2 -> 3 -> 7 -> 5 -> 5: percorso di lunghezza 4*

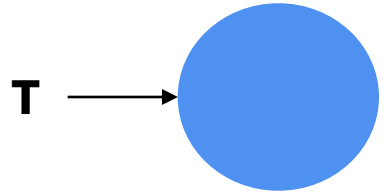
Calcolo ricorsivo dell'altezza

Chiamiamo l'albero **T**

Se *T* è un albero vuoto, ossia un puntatore **null**, allora:

`height(T) = 0`

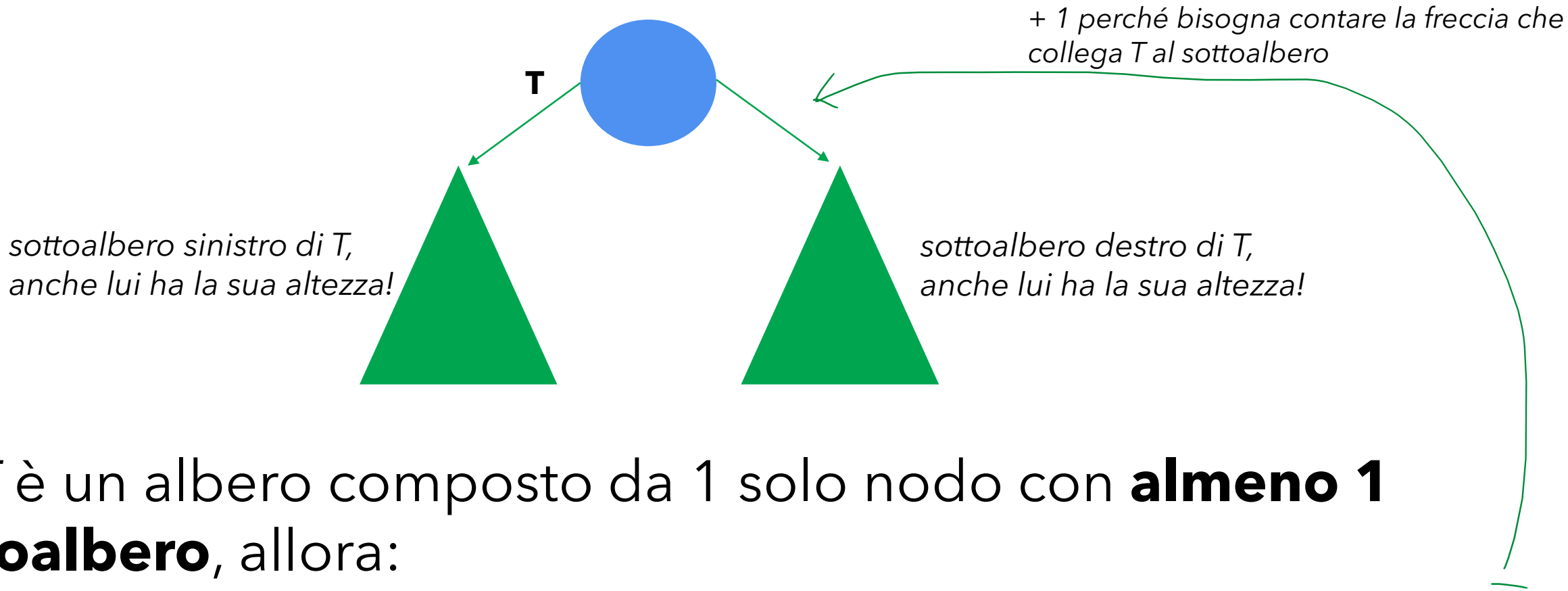
Calcolo ricorsivo dell'altezza



Se *T* è un albero composto da 1 solo nodo **senza sottoalberi**, allora:

`height(T) = 0`

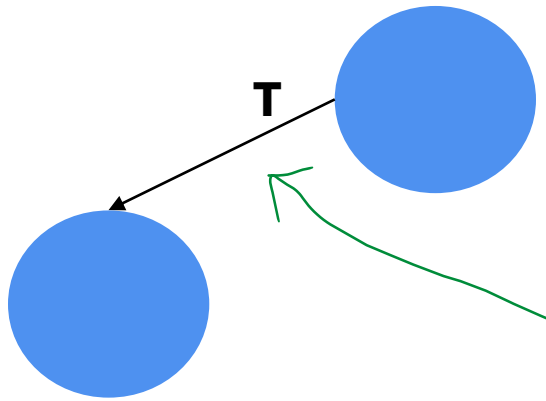
Calcolo ricorsivo dell'altezza



Se *T* è un albero composto da 1 solo nodo con **almeno 1 sottoalbero**, allora:

```
height(T) = max(height(T.left),height(T.right)) + 1
```

Calcolo ricorsivo dell'altezza



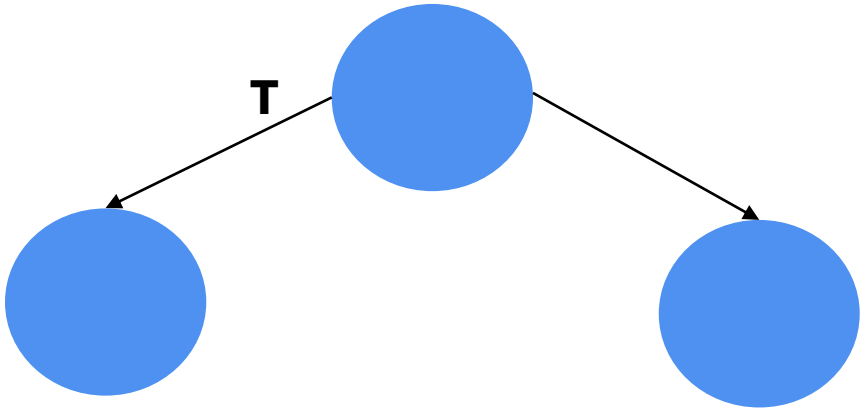
bisogna aggiungere 1:
sicuramente un albero composto da 1 nodo con almeno un sottoalbero ha altezza almeno 1.
Come in questo esempio.
È proprio questo +1 che permette di effettuare il calcolo completo.

```
height(T) = max(height(T.left), height(T.right)) + 1 =
max(0, 0) + 1 =
0 + 1 = 1
```

albero di 1 nodo

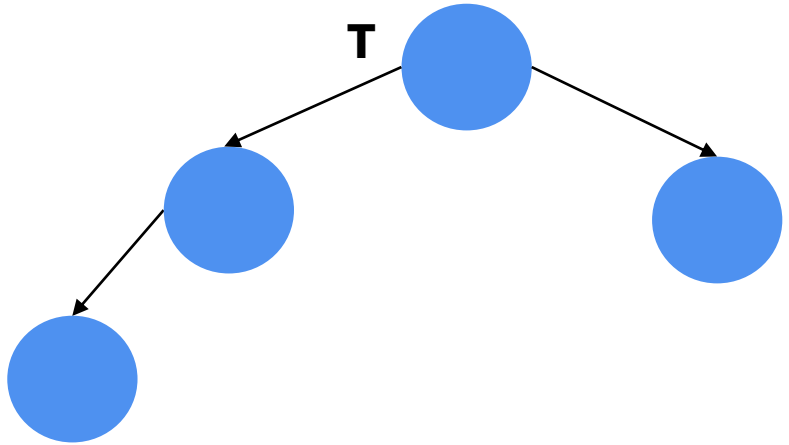
albero vuoto

Calcolo ricorsivo dell'altezza




```
height(T) = max(height(T.left), height(T.right)) + 1 =
max(0, 0) + 1 =
0 + 1 = 1
```

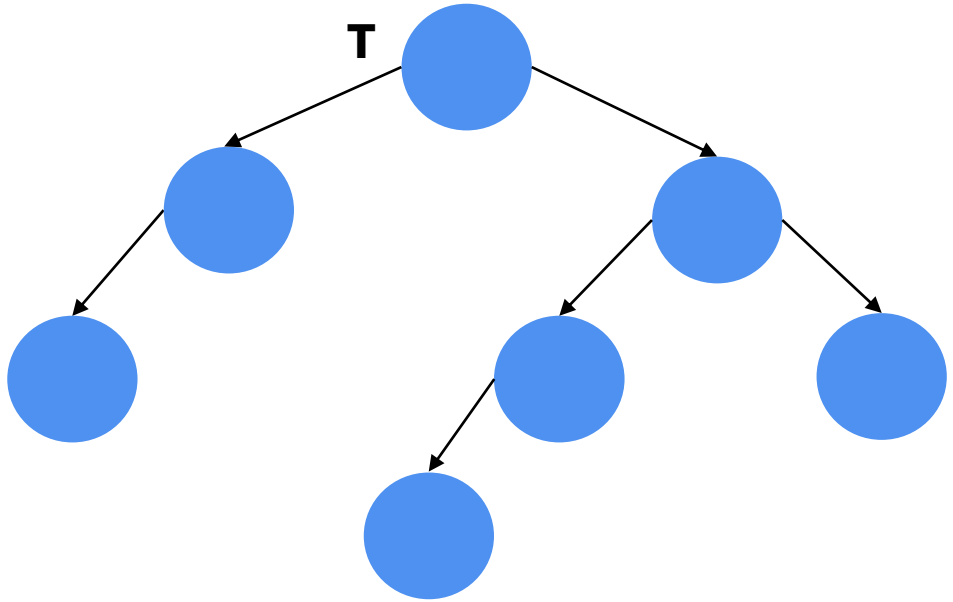
Calcolo ricorsivo dell'altezza



Bisogna espandere la funzione height ricorsivamente fino ai casi base, così:

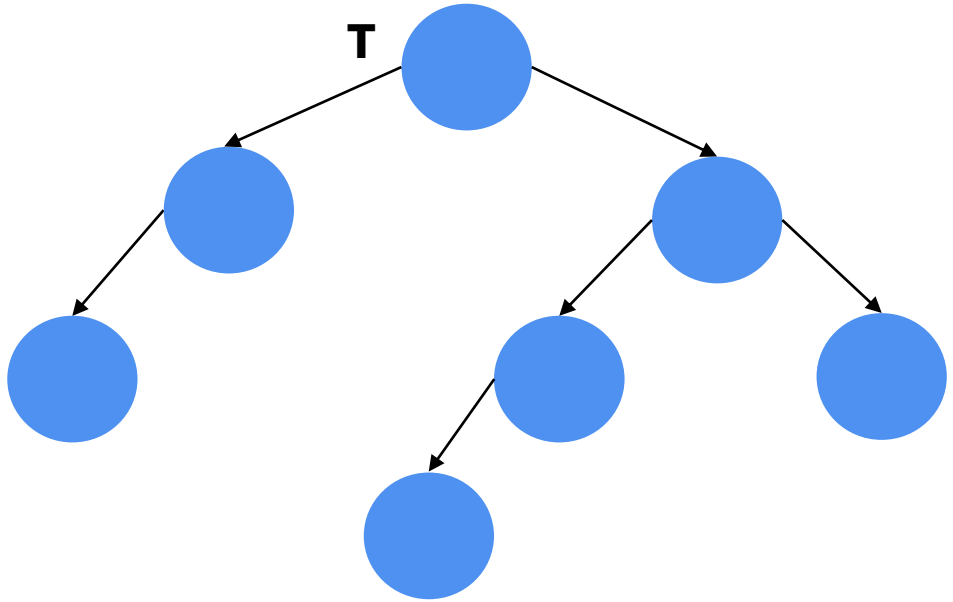
```
height(T) = max(height(T.left), height(T.right)) + 1 =
max(max(height(T.left.left), height(T.left.right)) + 1, 0) + 1 =
max(max(0, 0) + 1, 0) + 1 = max(1, 0) + 1 = 2
```

Calcolo ricorsivo dell'altezza



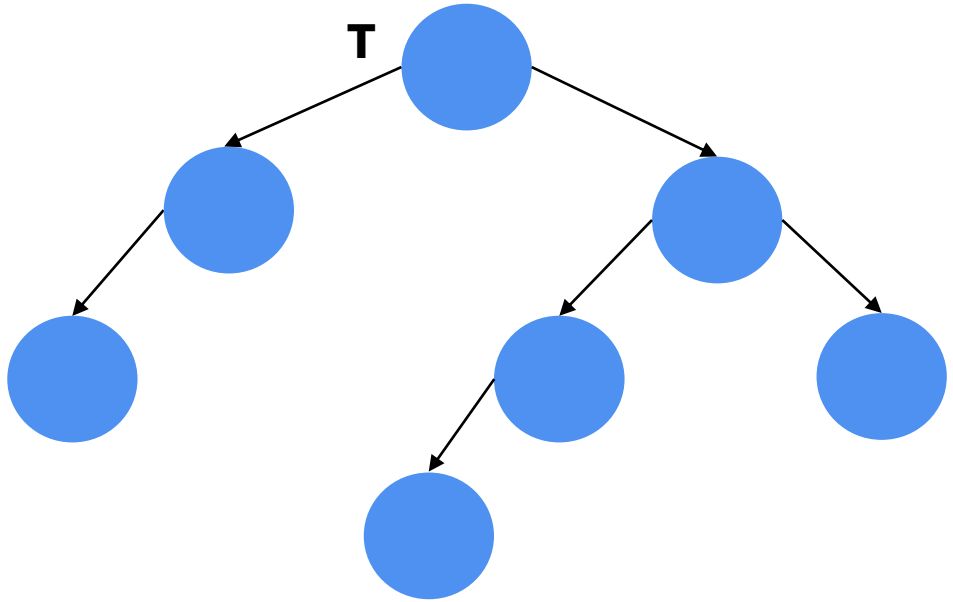

```
height(T) = max(height(T.left), height(T.right)) + 1
=
```

Calcolo ricorsivo dell'altezza



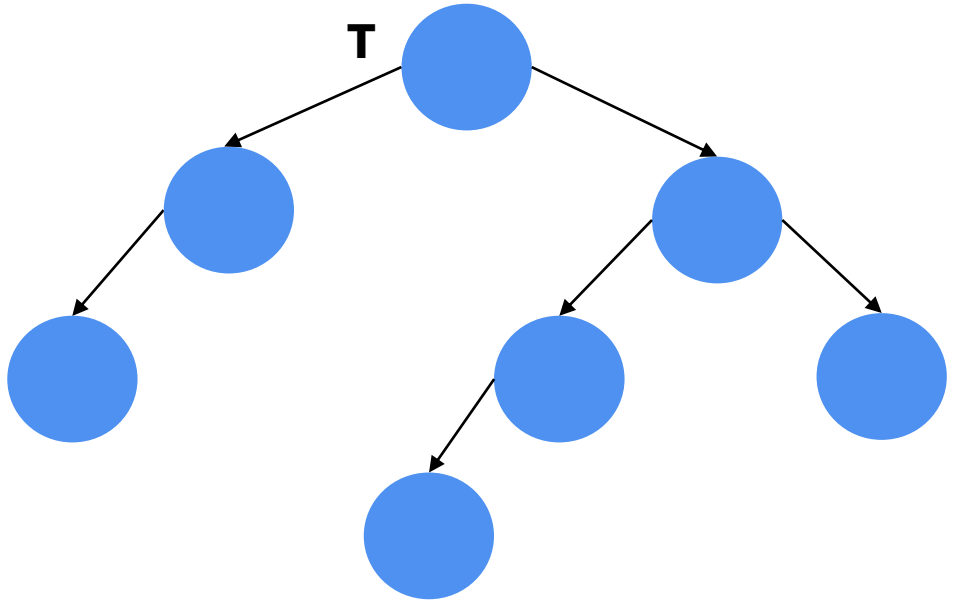

```
=
max(max(height(T.left.left),height(T.left.right))+1,max(height(T.right.left,
height(T.right.right))+1)+1 =
```

Calcolo ricorsivo dell'altezza



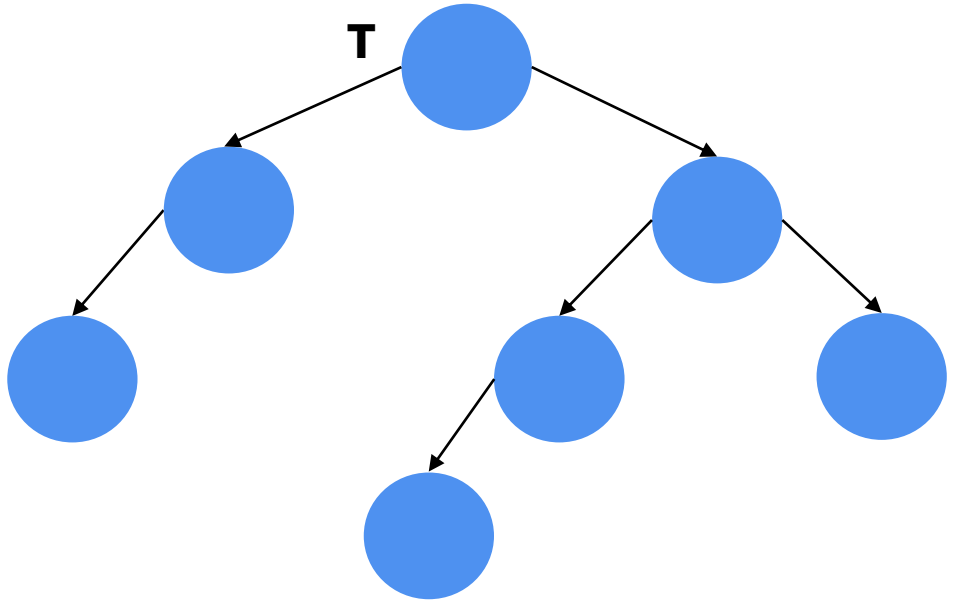

```
= max(max(0,0) + 1, max(height(T.right.left),
0) + 1) + 1 =
```

Calcolo ricorsivo dell'altezza



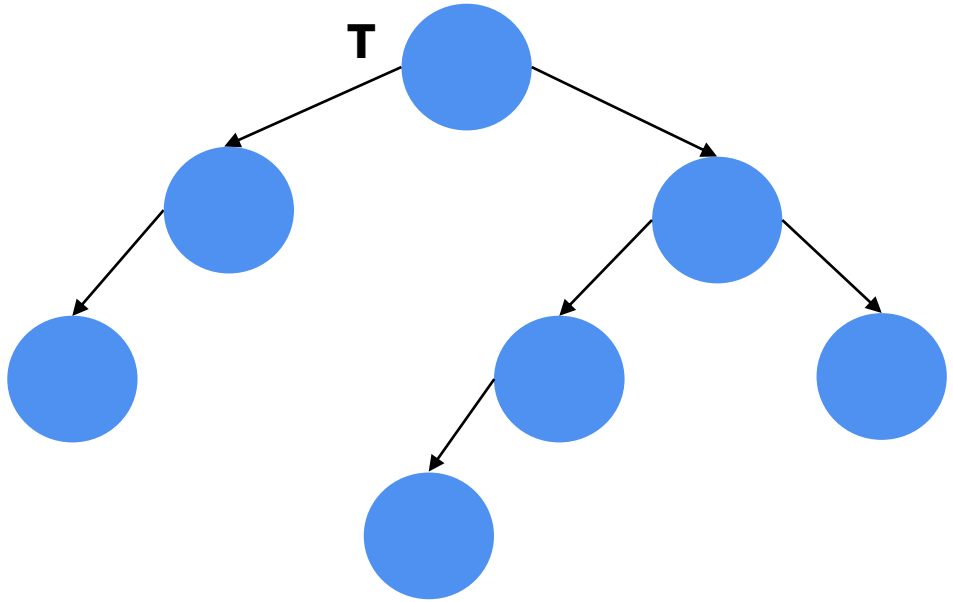

```
=
max(max(0,0)+1,max(max(height(T.right.left.left),height(T.right.left.right))
+1,0)+1)+1 =
```

Calcolo ricorsivo dell'altezza



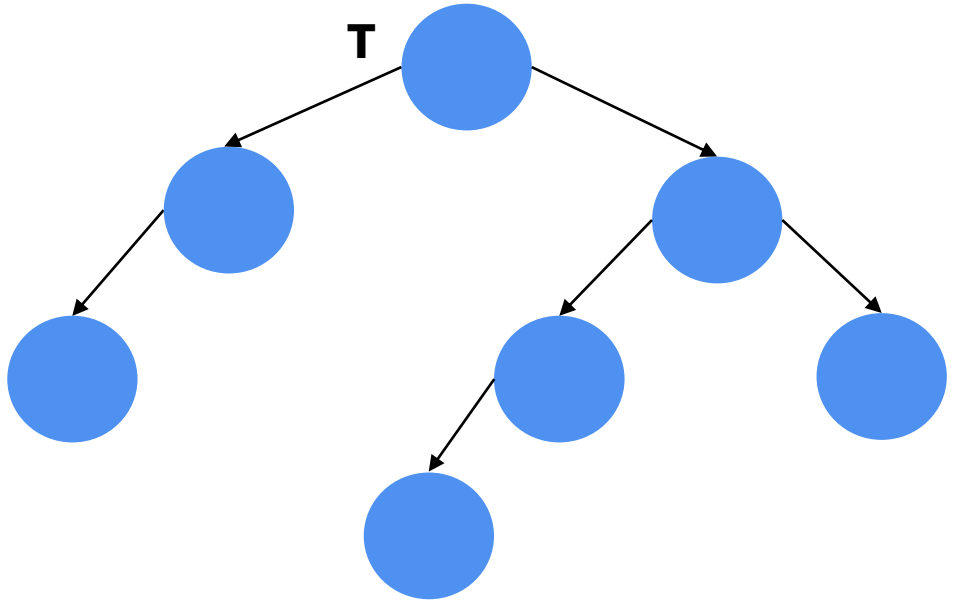

```
=
max(max(0,0)+1,max(max(height(T.right.left.left),height(T.right.left.right))
+1,0)+1)+1 =
```

Calcolo ricorsivo dell'altezza



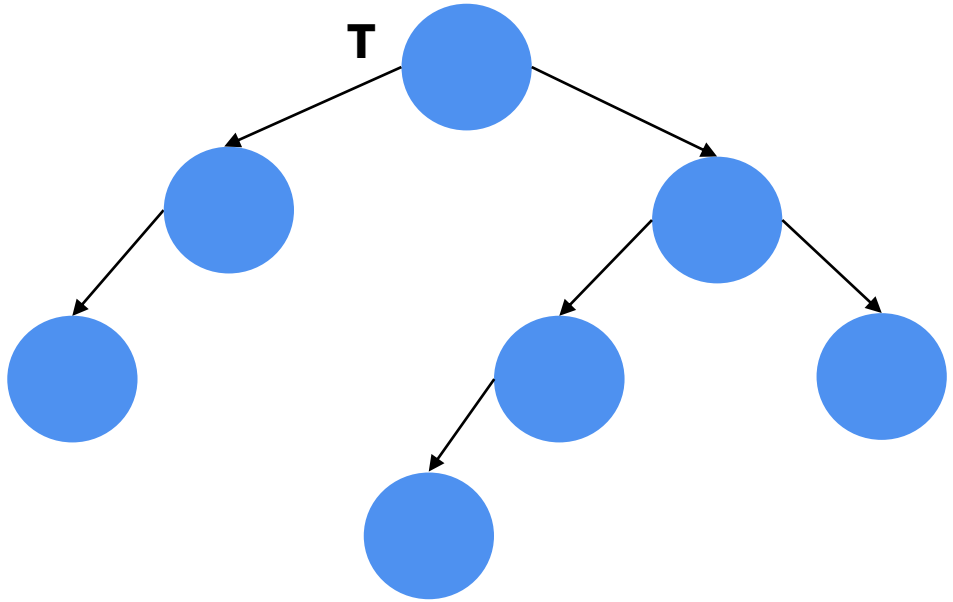

```
= max(max(0,0)+1,max(max(0,0)+1,0)+1)+1 =
```

Calcolo ricorsivo dell'altezza



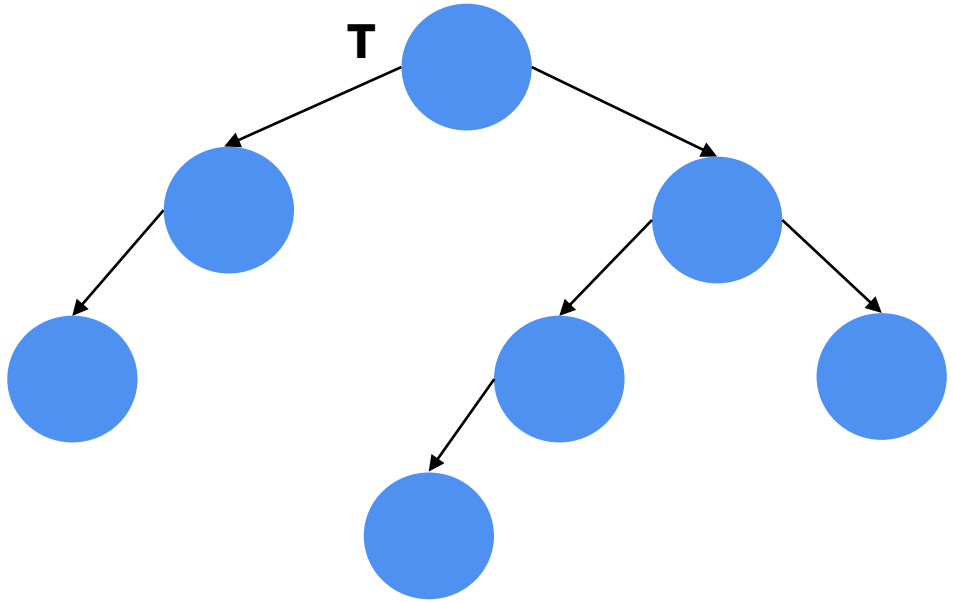

```
= max(0 + 1, max(0 + 1, 0) + 1) + 1 =
```

Calcolo ricorsivo dell'altezza



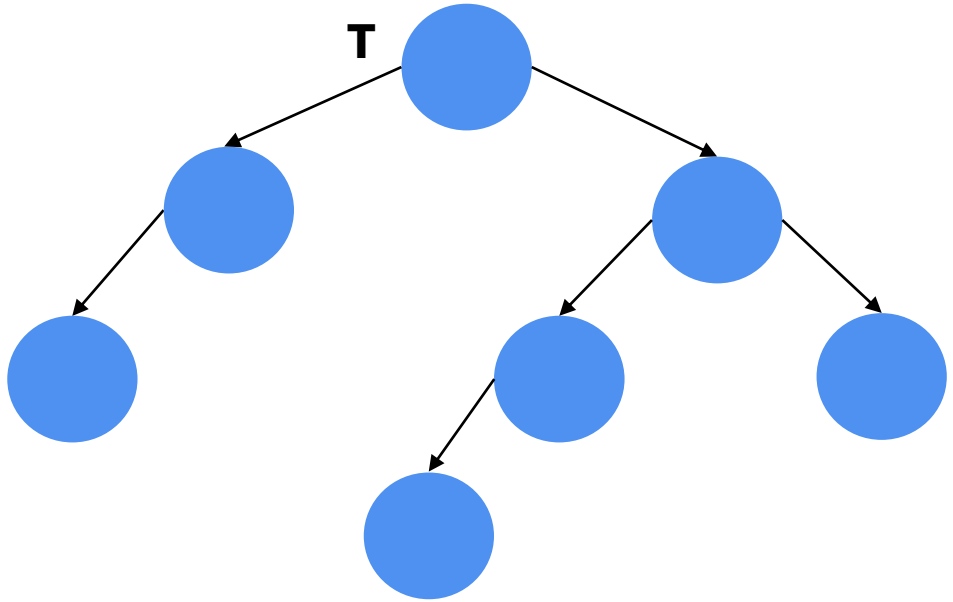

```
= max(1, max(1, 0) + 1) + 1 =
```

Calcolo ricorsivo dell'altezza



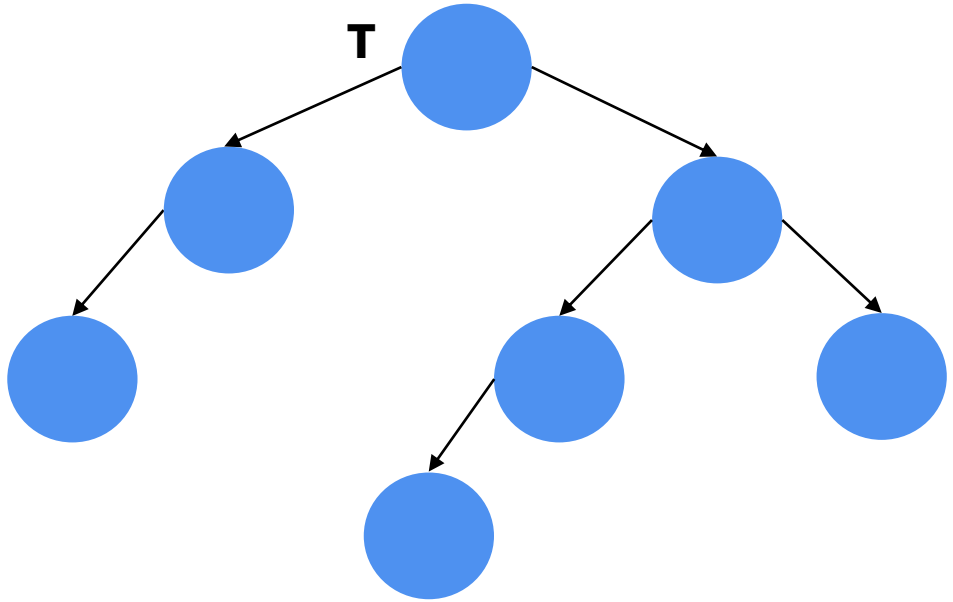

```
= max(1, 1 + 1) + 1 =
```

Calcolo ricorsivo dell'altezza




```
= max(1, 2) + 1 =
```

Calcolo ricorsivo dell'altezza




```
= 2 + 1 = 3
```

La funzione **height** in pseudocodice

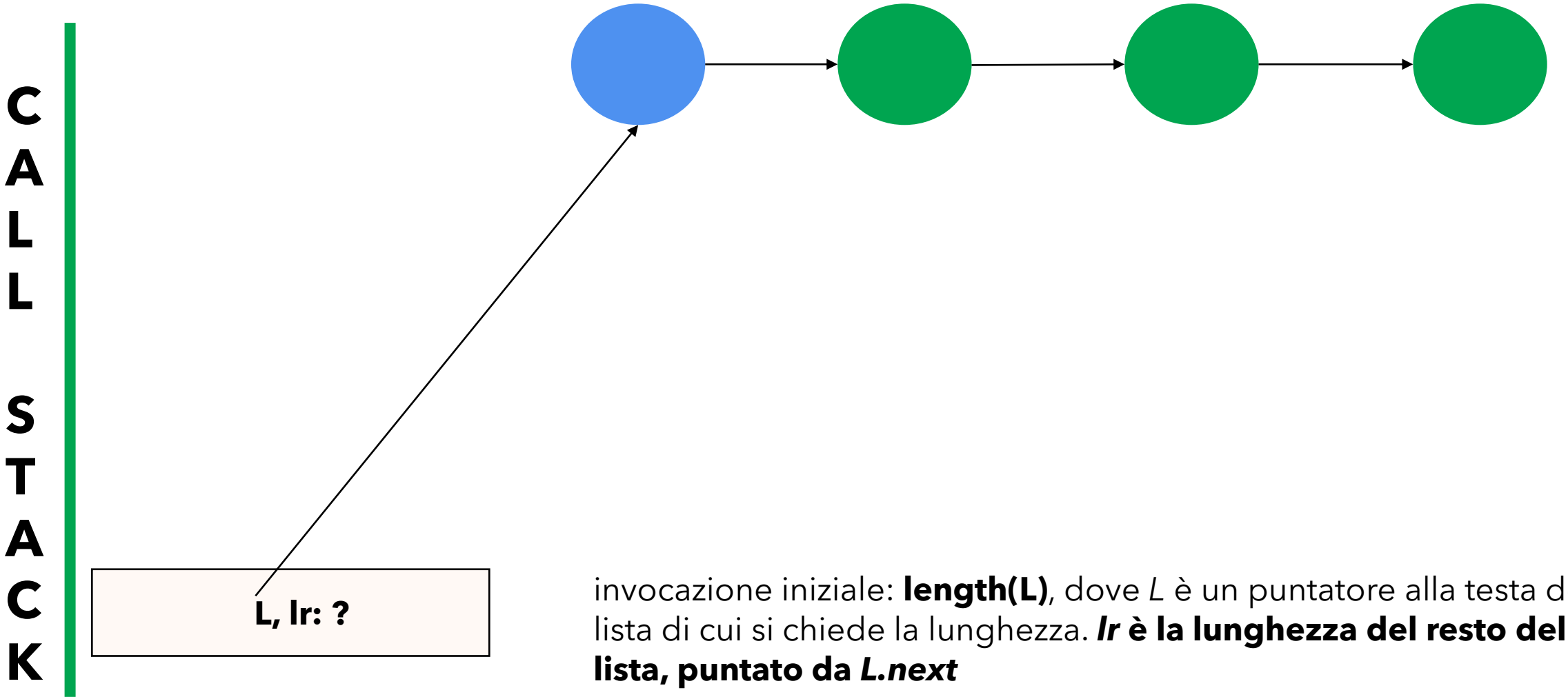
```
height(T): returns int
        if T is nil:
                return 0
        if T.left is nil and T.right is nil:
                return 0
        int heightLeftSubTree = height(T.left)
        int heightRightSubTree = height(T.right)

        return max(heightLeftSubTree, heightRightSubTree) + 1
```

Analogia: calcolo della lunghezza di una *linked list*

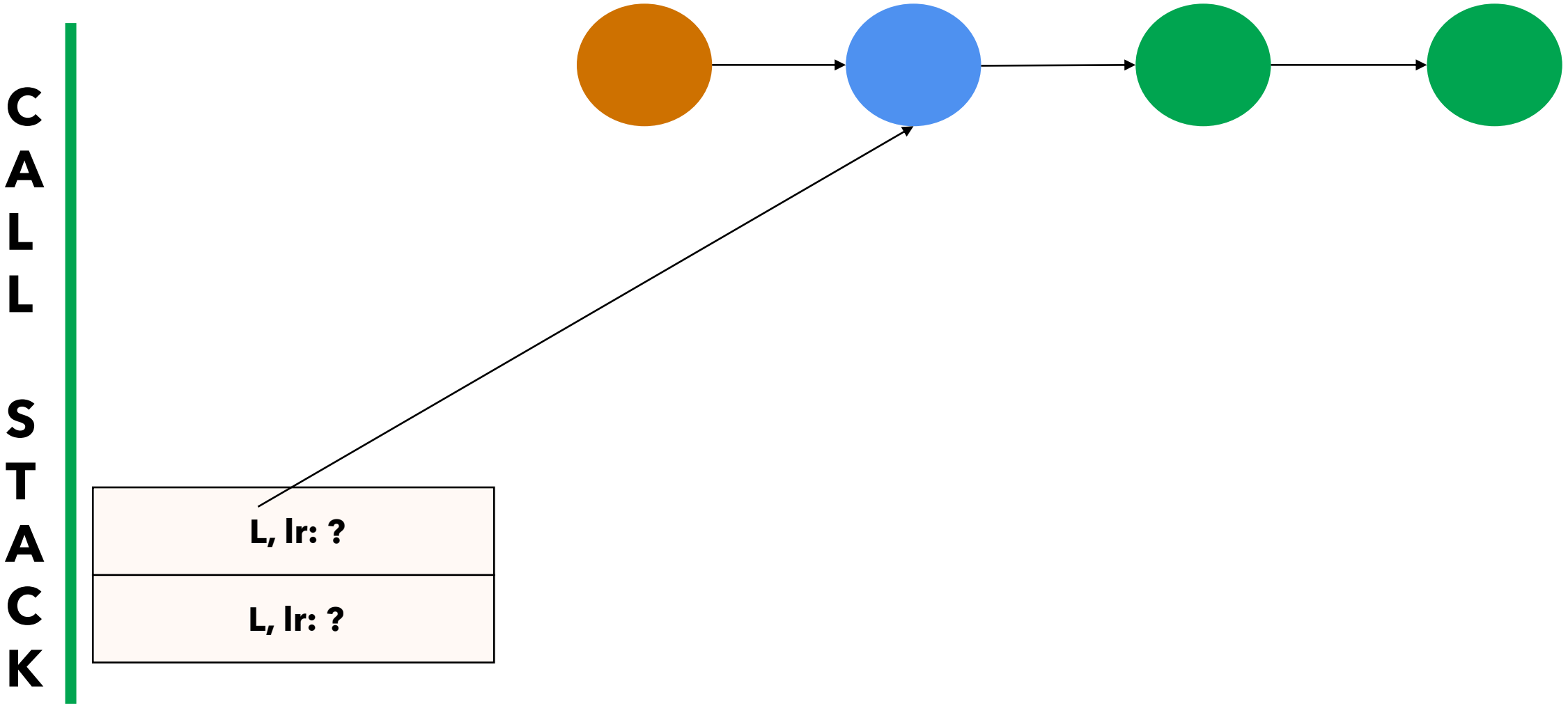
```
length(L): returns an integer
          if L is nil:
               return 0
          int length_remainder = length(L.next)
          return length_remainder + 1
```

Analogia: calcolo della lunghezza di una *linked list*

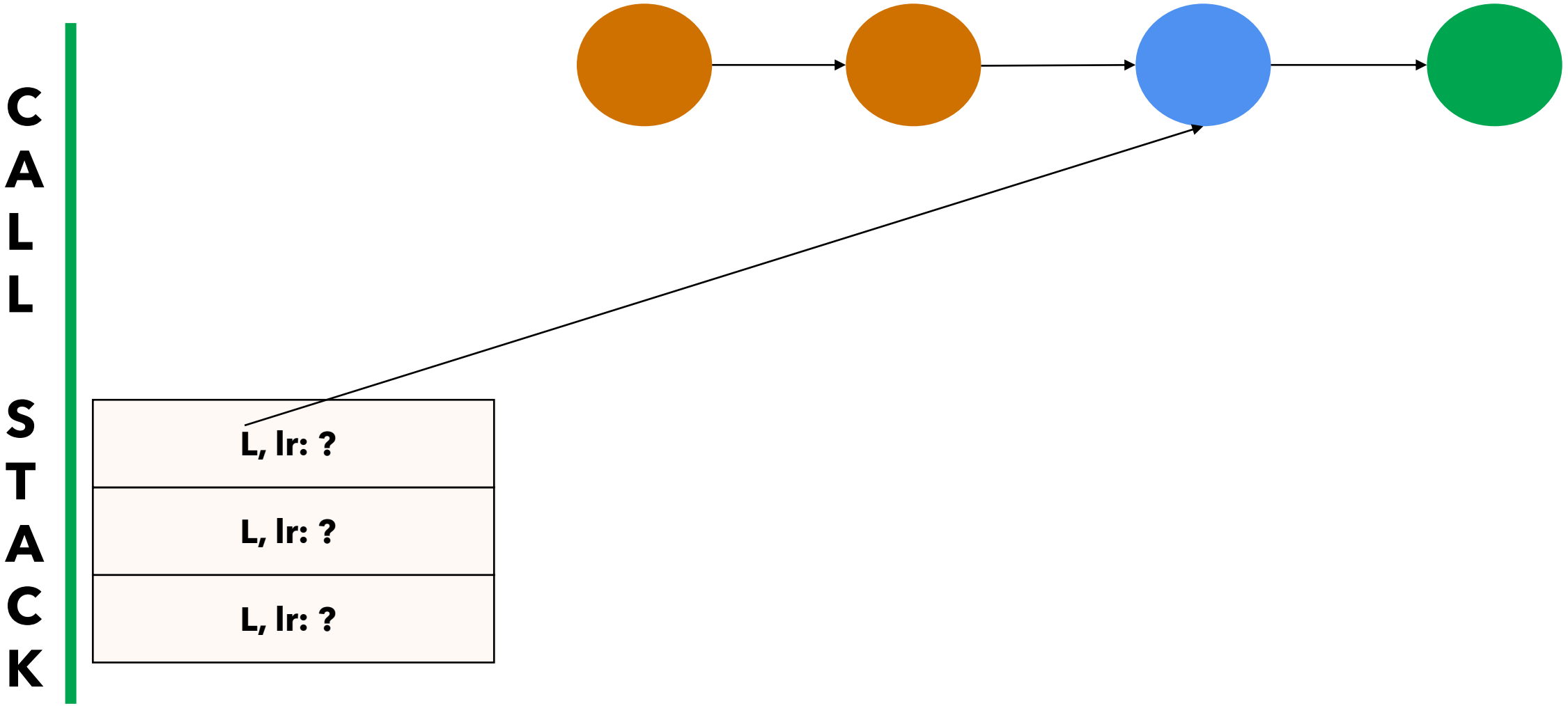


invocazione iniziale: **length(L)**, dove *L* è un puntatore alla testa della lista di cui si chiede la lunghezza. ***lr* è la lunghezza del resto della lista, puntato da *L.next***

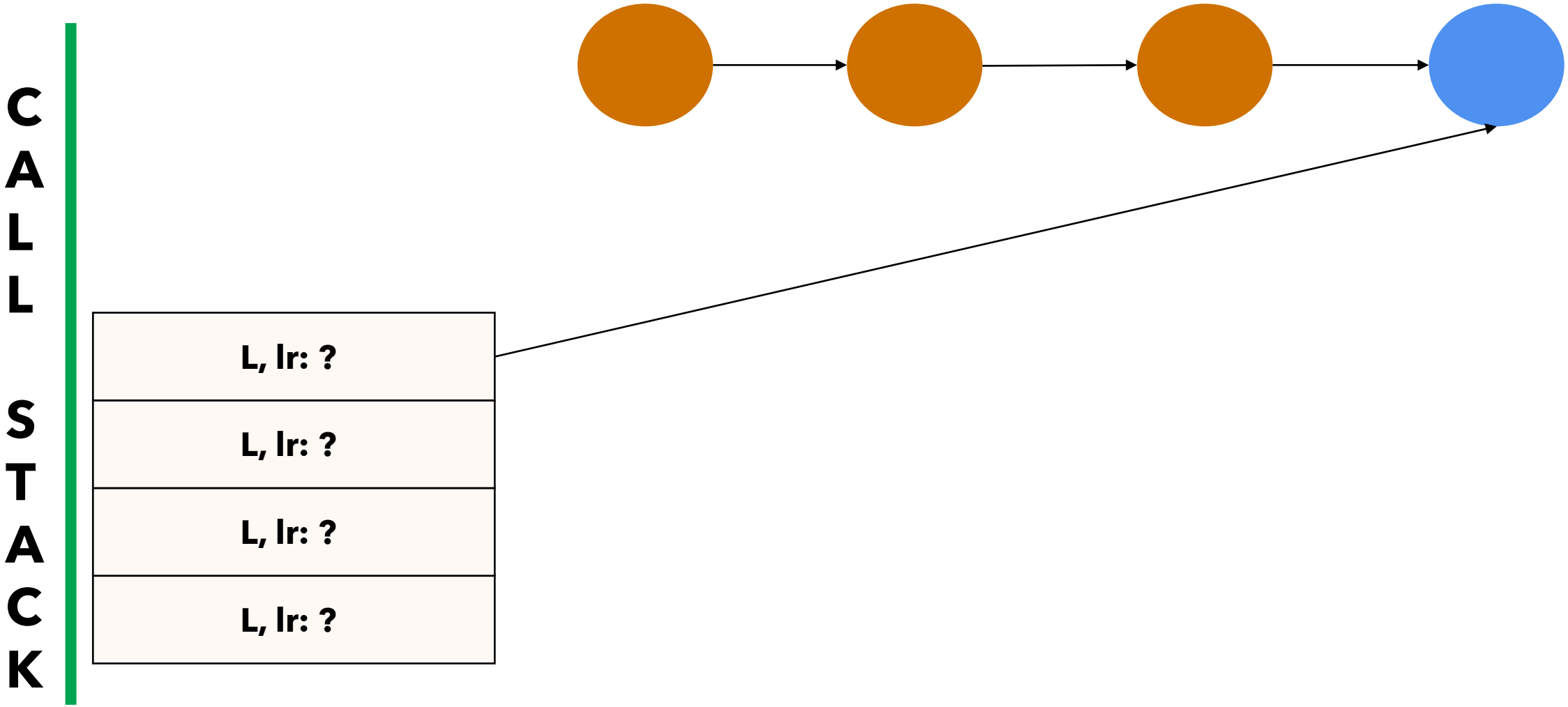
Analogia: calcolo della lunghezza di una *linked list*



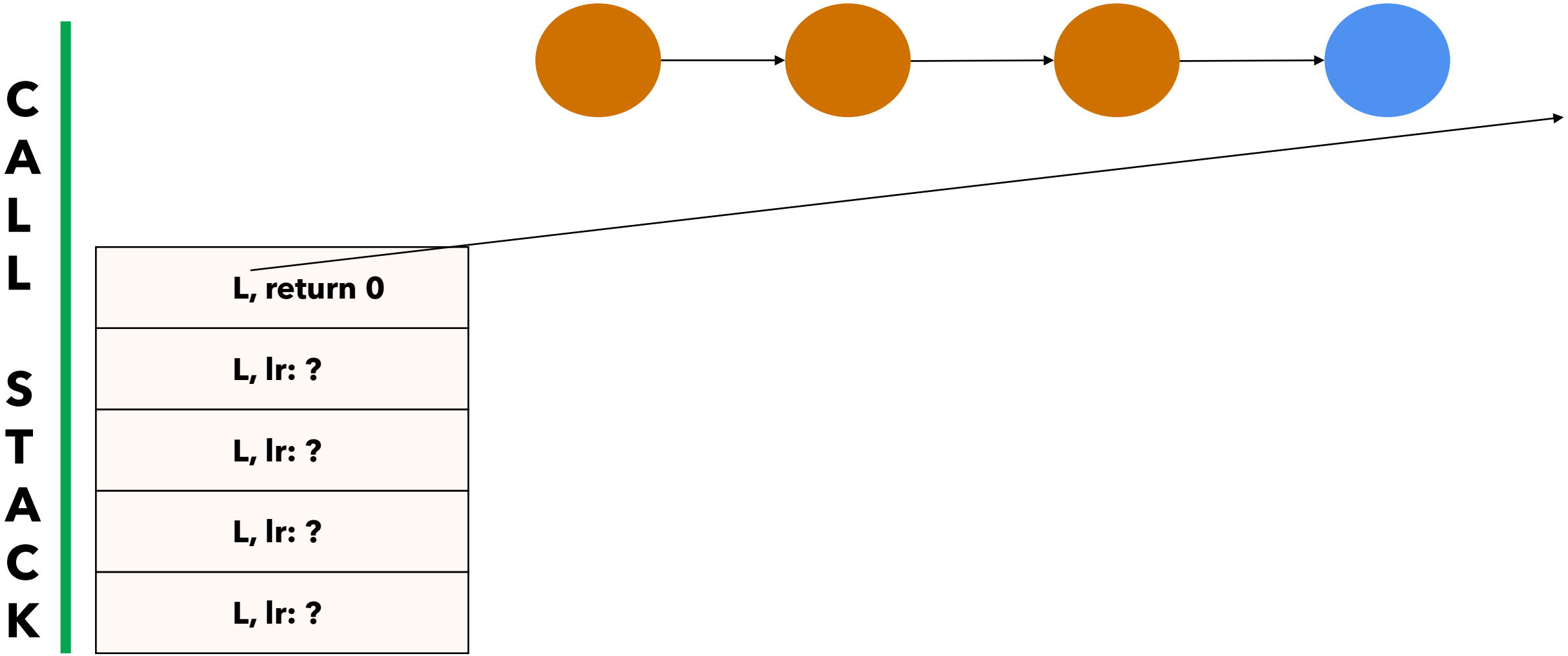
Analogia: calcolo della lunghezza di una *linked list*



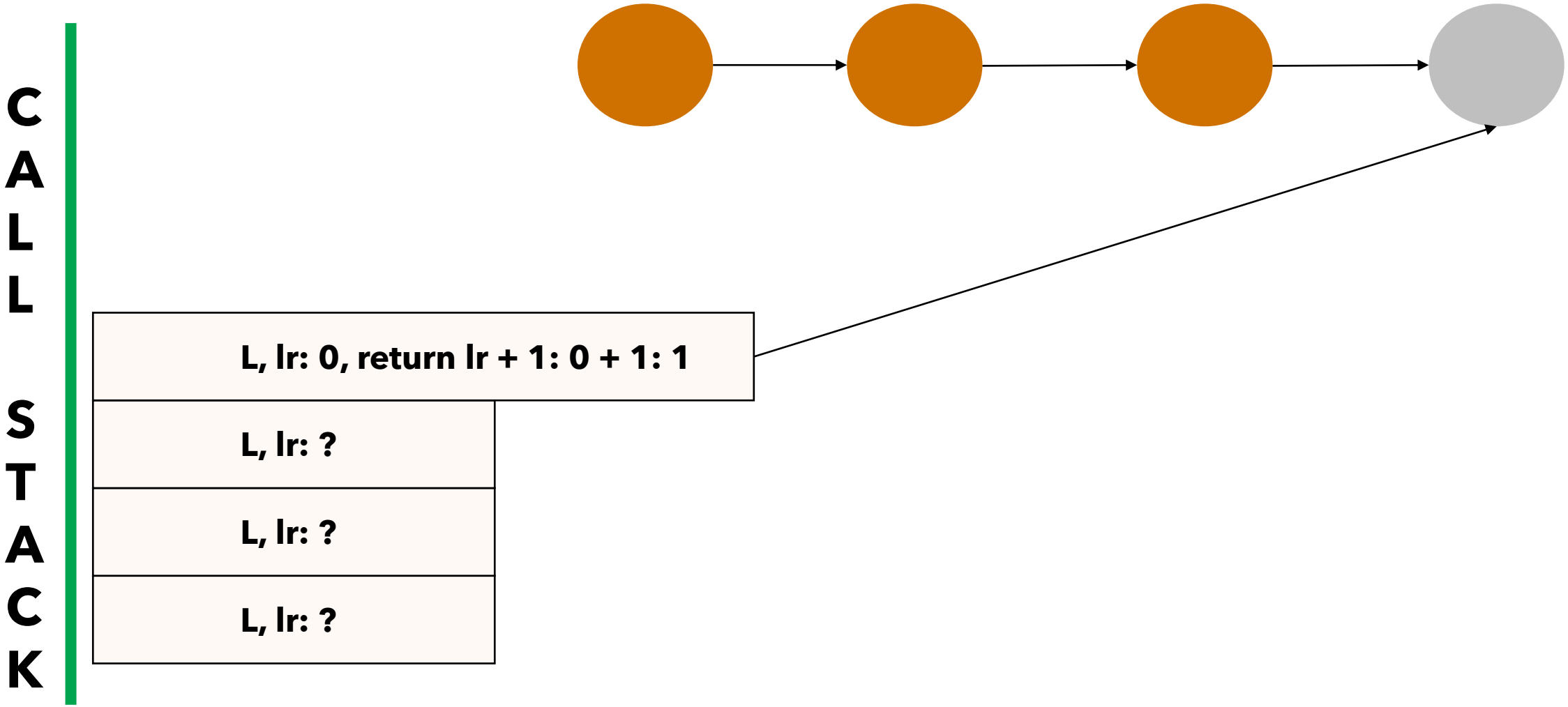
Analogia: calcolo della lunghezza di una *linked list*



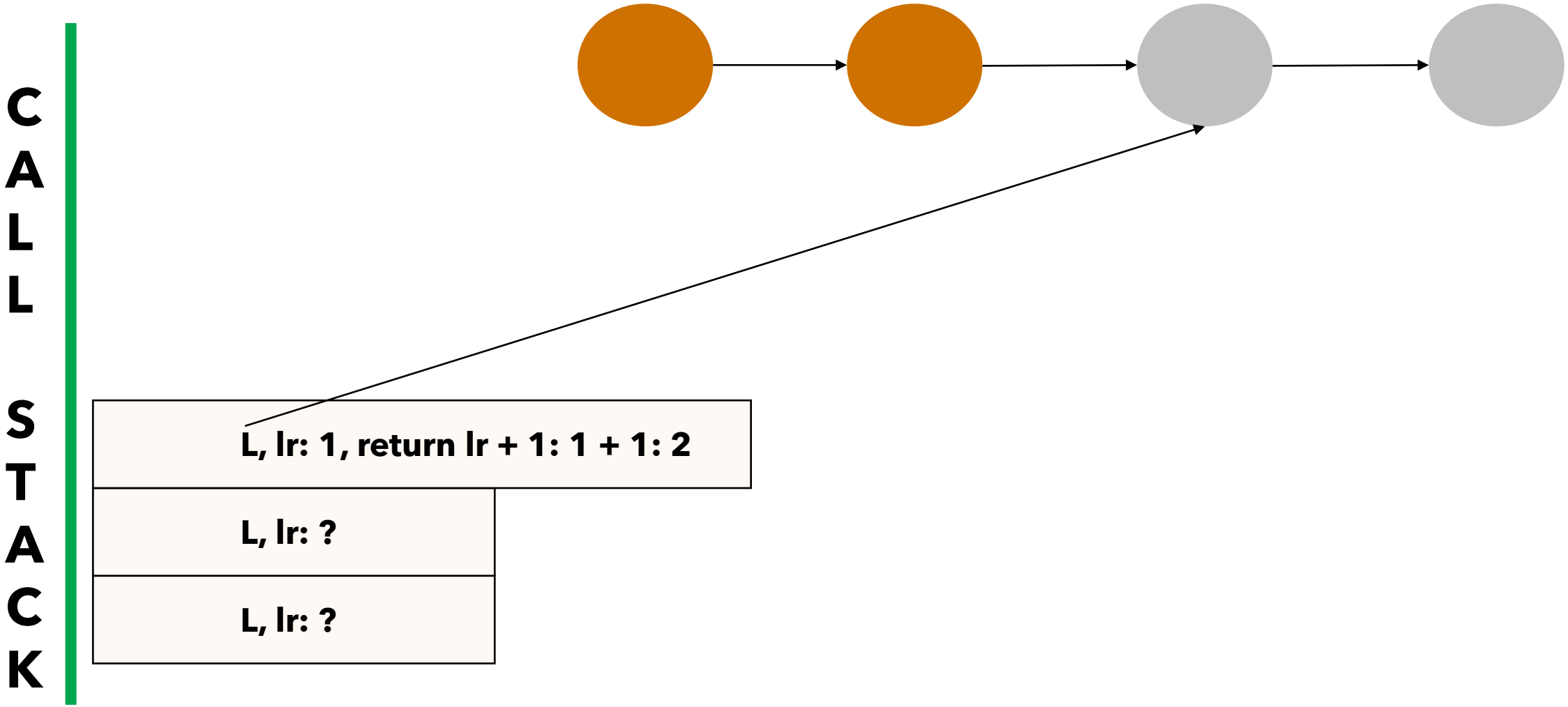
Analogia: calcolo della lunghezza di una *linked list*



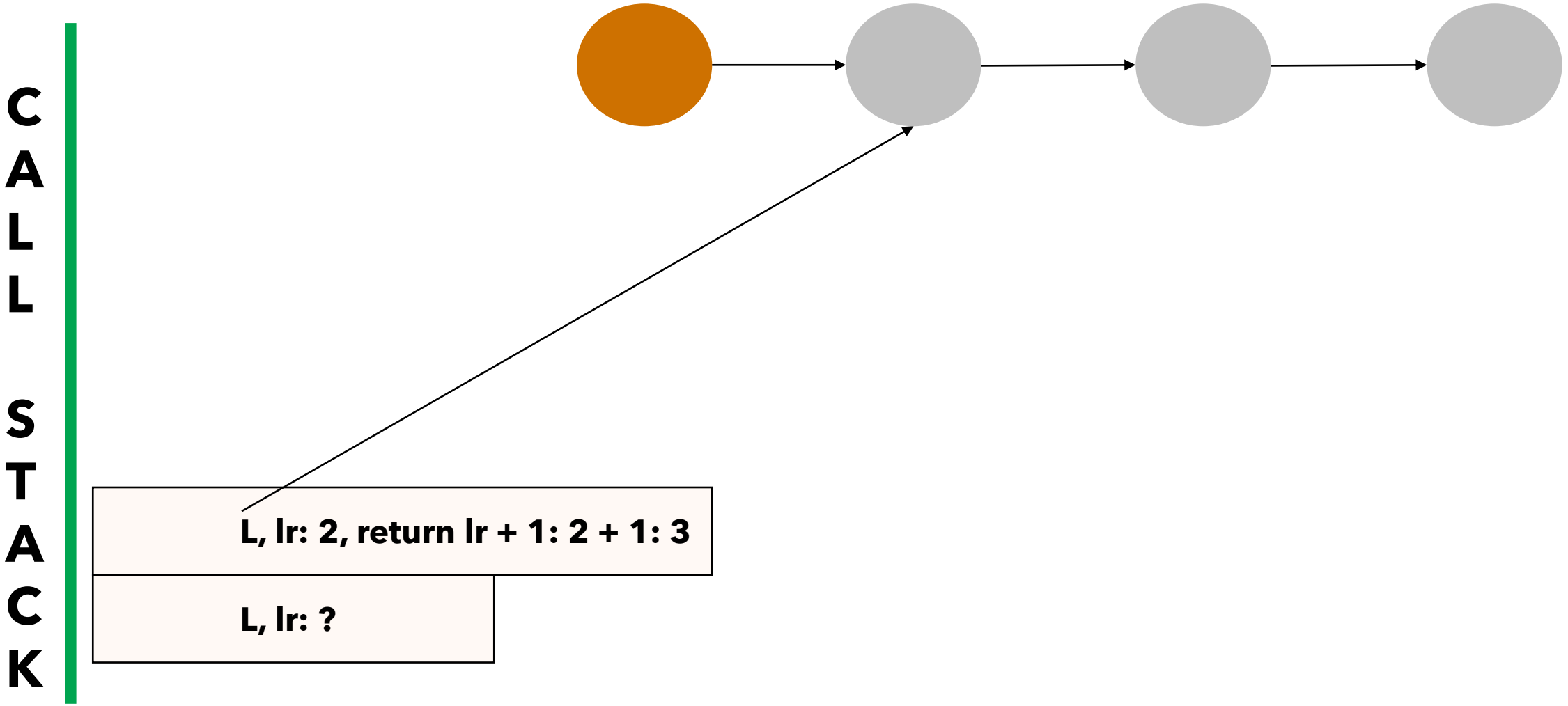
Analogia: calcolo della lunghezza di una *linked list*



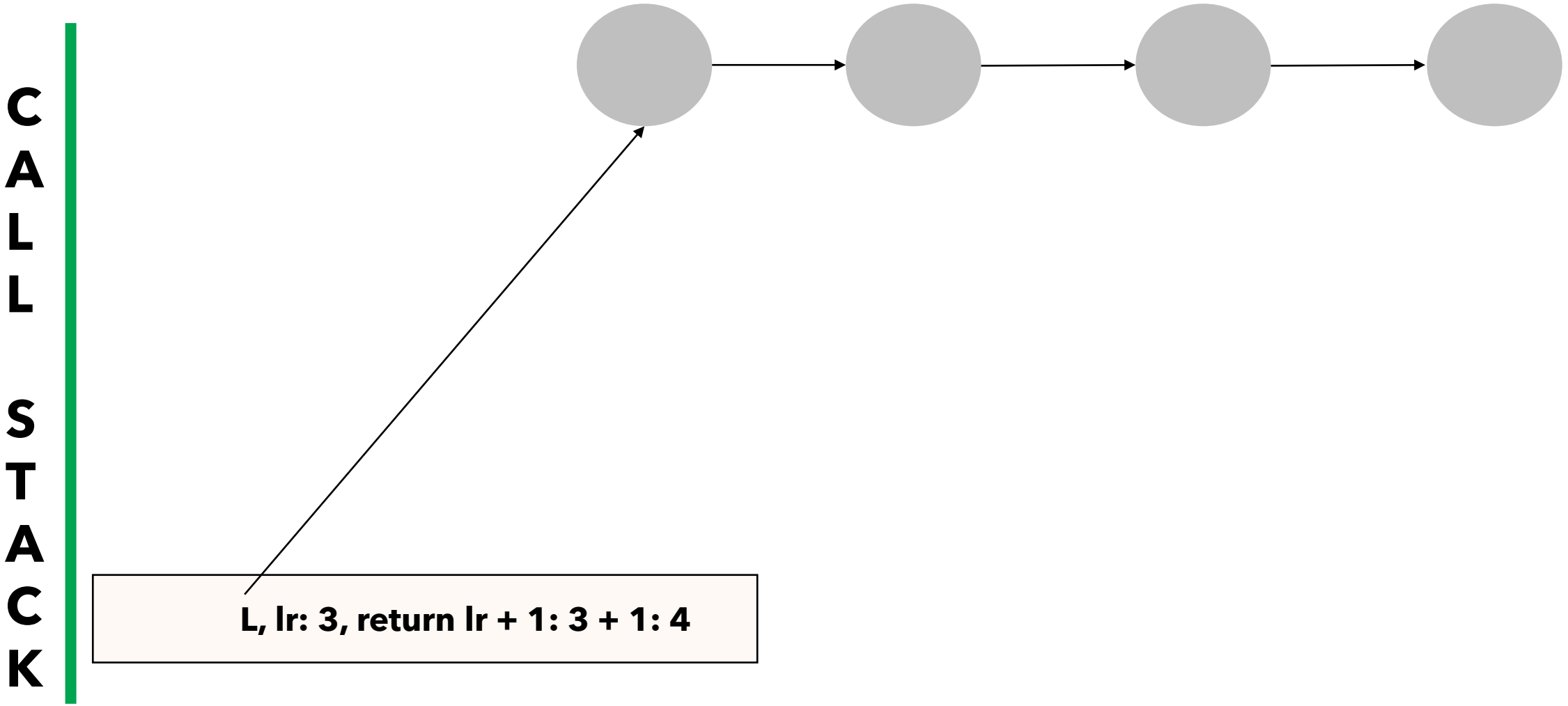
Analogia: calcolo della lunghezza di una *linked list*



Analogia: calcolo della lunghezza di una *linked list*



Analogia: calcolo della lunghezza di una *linked list*

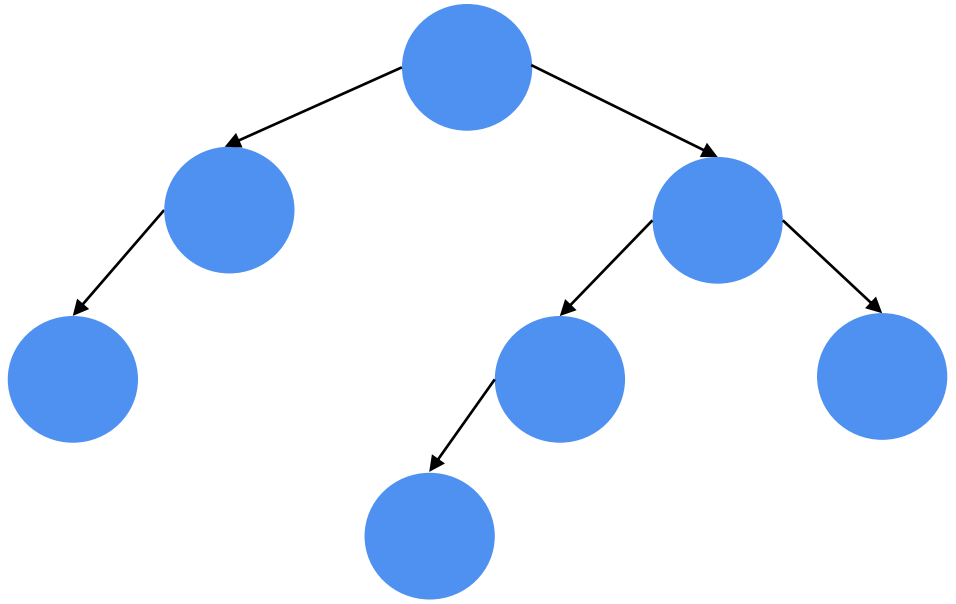


‘Srotoliamo’ la ricorsione

Nodo radice dell’invocazione corrente

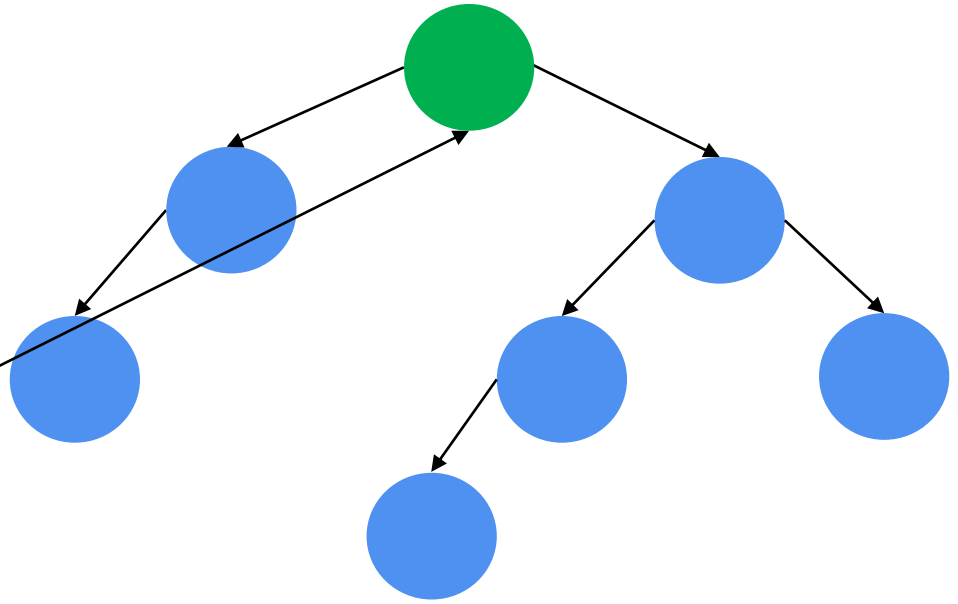
Nodo per cui l ’altezza è stata calcolata completamente

Nodo per cui l’altezza non è stata calcolata completamente



Il chiamante (ad esempio il metodo *main*) invoca **height(T)**. Vediamo cosa succede sullo stack delle chiamate di funzione. Indicheremo in *verde la radice dell’invocazione corrente*, in ***grigio quelli per cui l’altezza è stata calcolata***, *in marrone i nodi visitati la cui altezza non è stata calcolata*

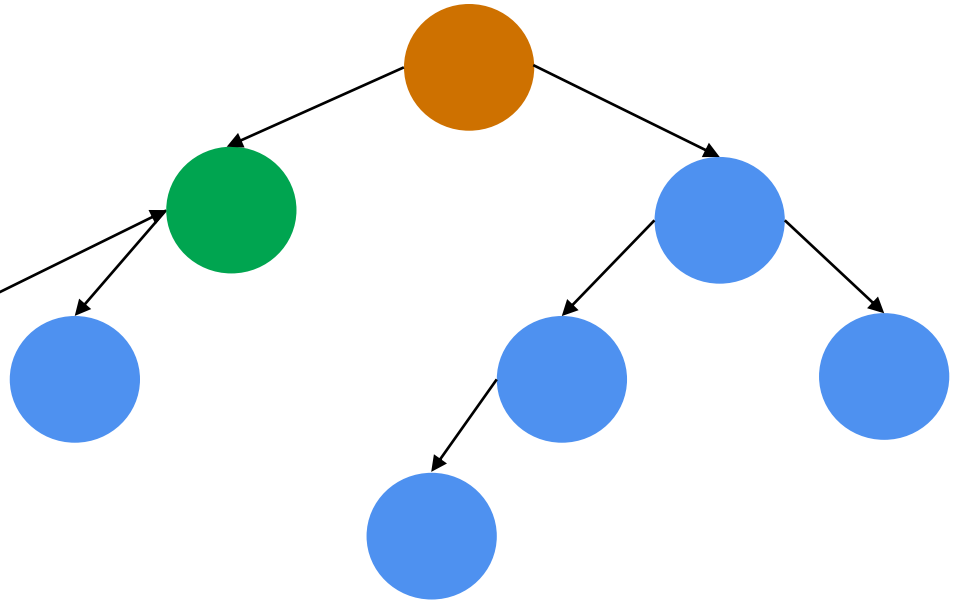
‘Srotoliamo’ la ricorsione



T, hl: ?, hr: ?

invocazione corrente: **height(T)**, **hl**: altezza del sottoalbero sinistro, **hr**: altezza del sottoalbero destro. Sono variabili locali diverse ad ogni invocazione ricorsiva

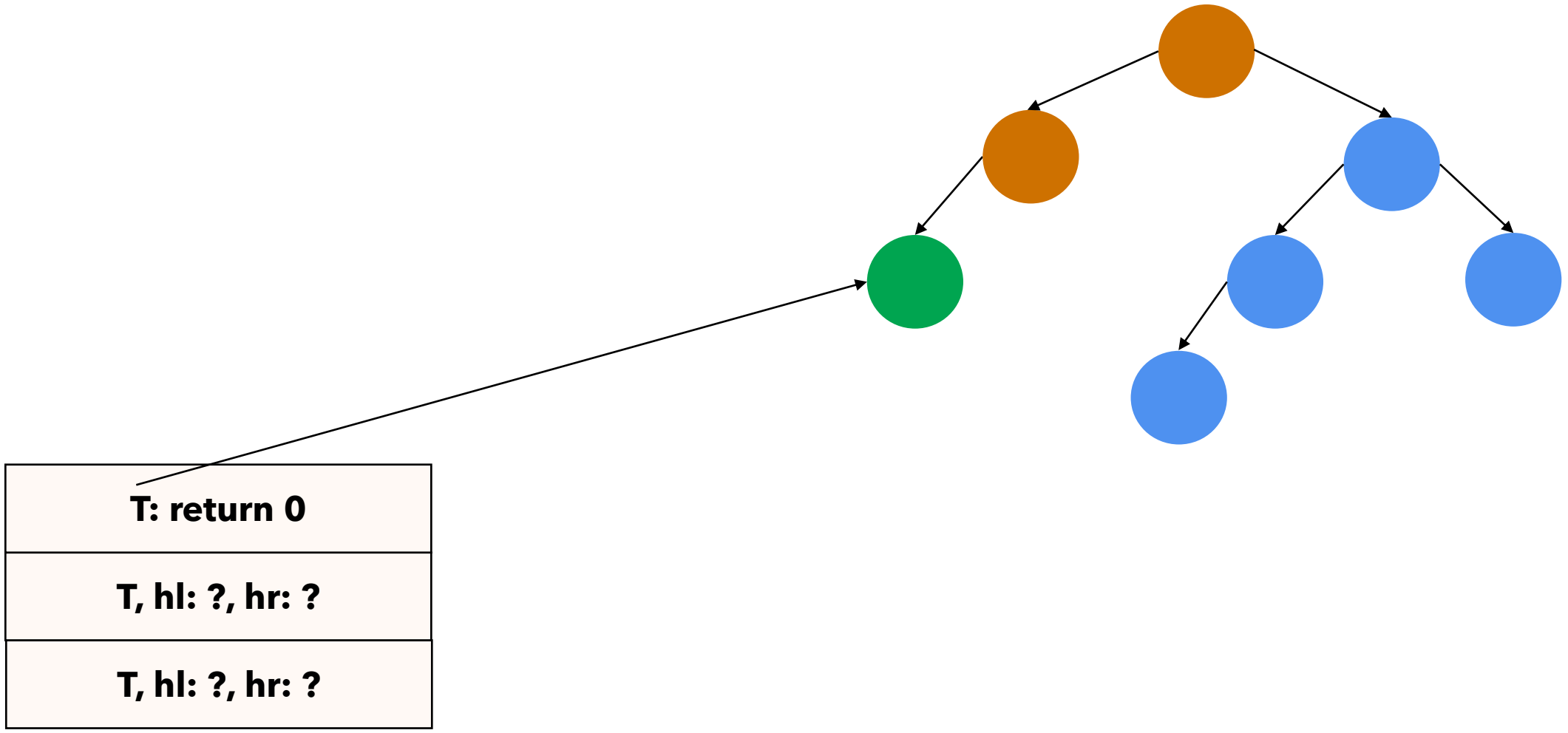
‘Srotoliamo’ la ricorsione



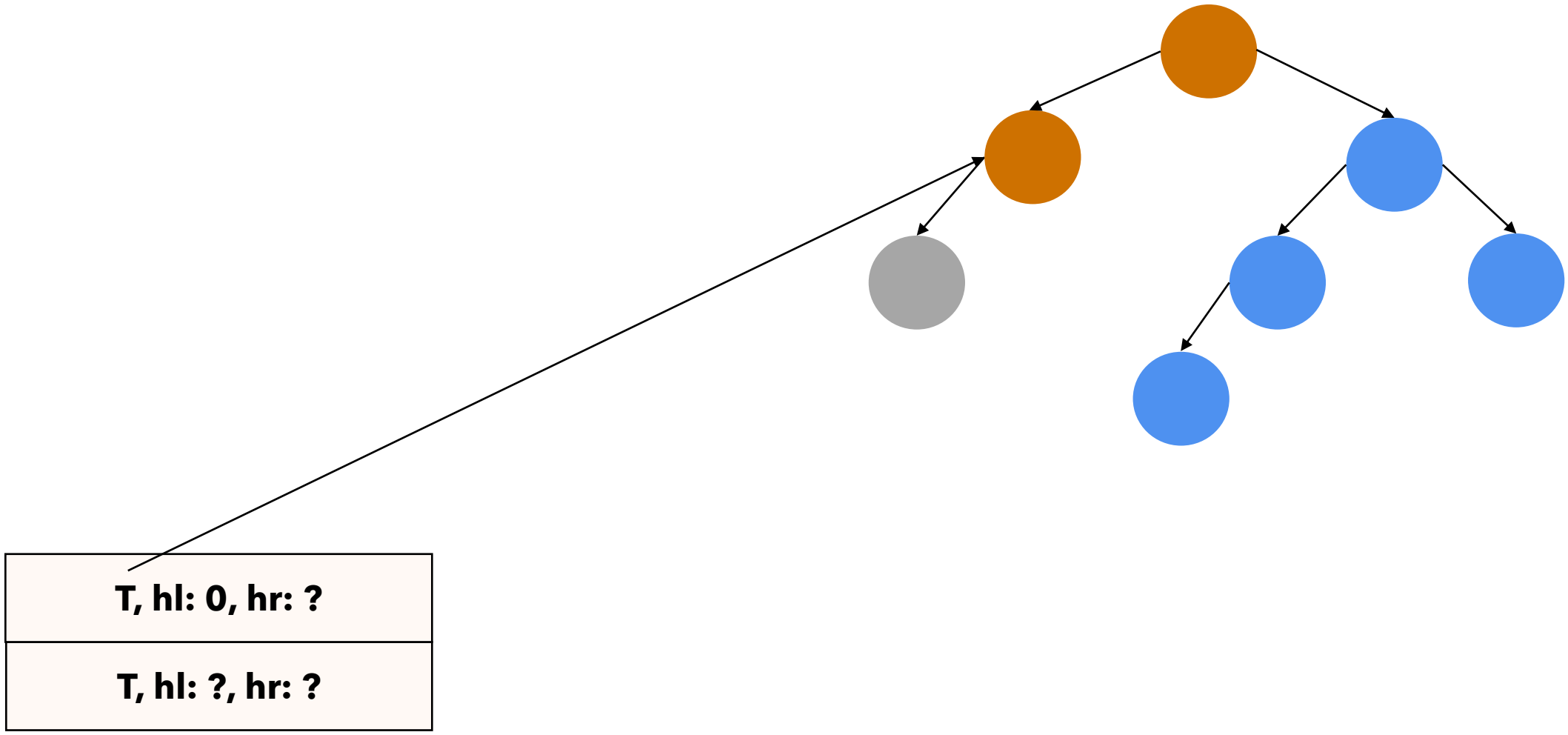
T, hl: ?, hr: ?
T, hl: ?, hr: ?

NB: *per come abbiamo scritto il codice, visitiamo prima il sottoalbero sinistro, e poi quello destro. Chiaramente li visitiamo solo se esistono.*

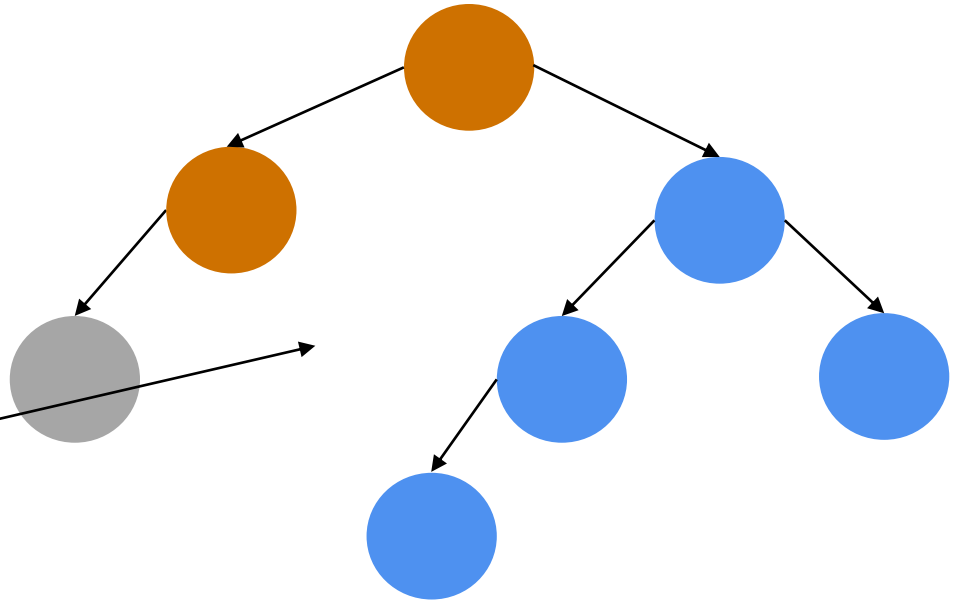
‘Srotoliamo’ la ricorsione



‘Srotoliamo’ la ricorsione

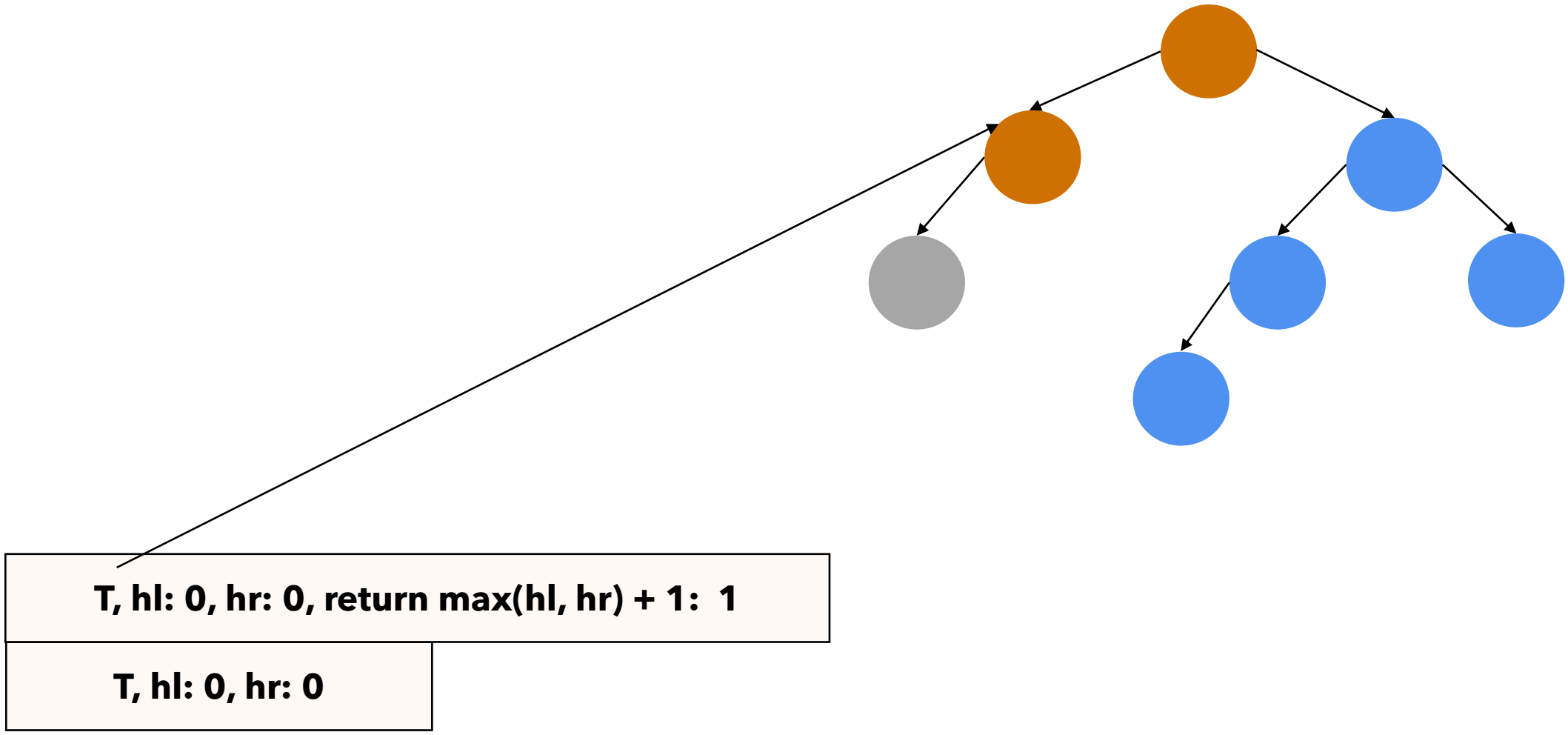


‘Srotoliamo’ la ricorsione

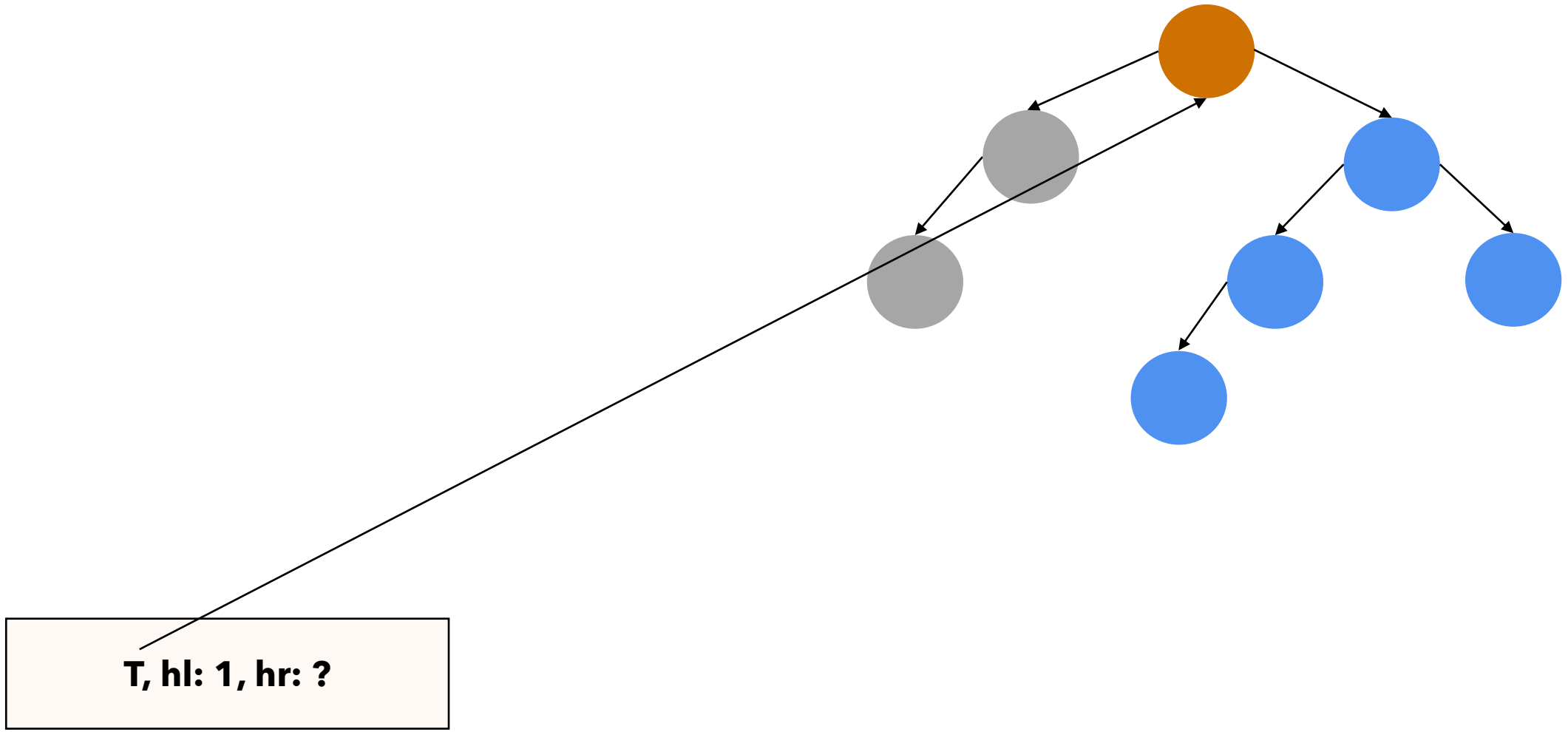


T, return 0
T, hl: 0, hr: ?
T, hl: 0, hr: 0

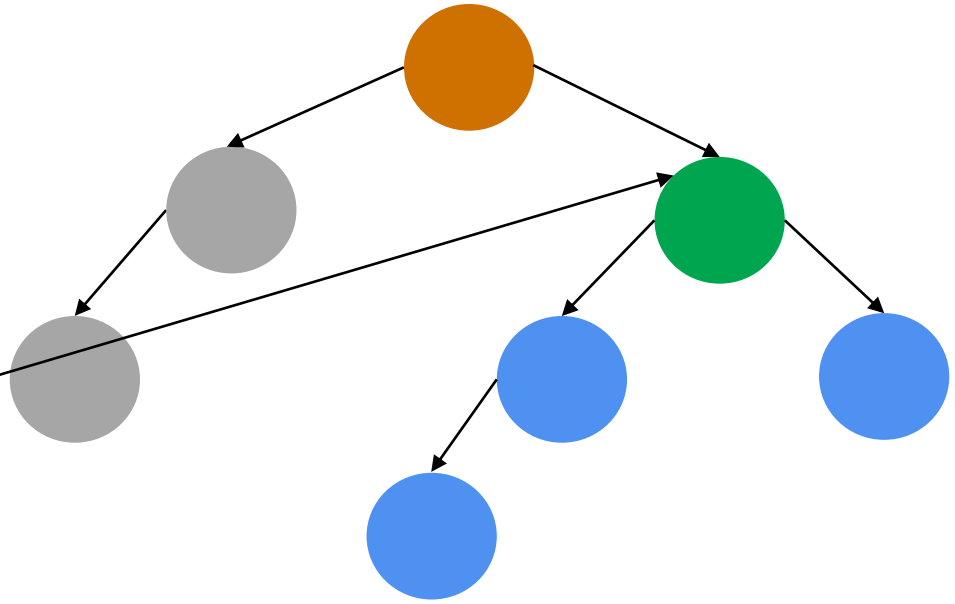
‘Srotoliamo’ la ricorsione

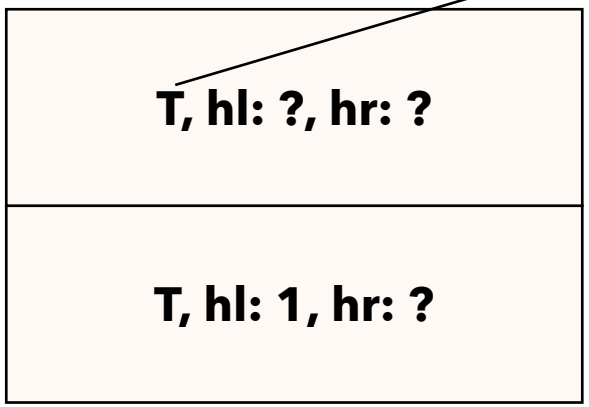


‘Srotoliamo’ la ricorsione

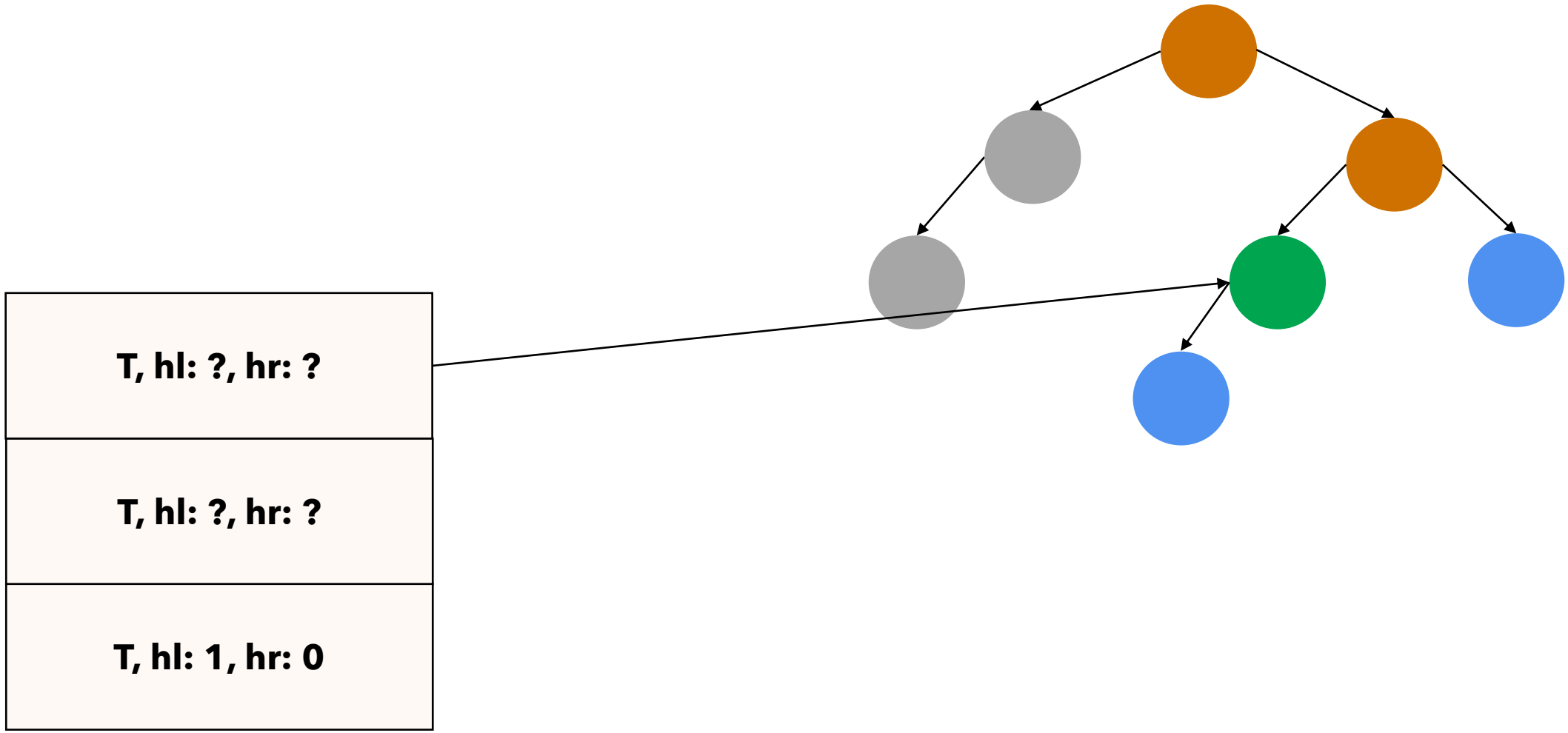


‘Srotoliamo’ la ricorsione

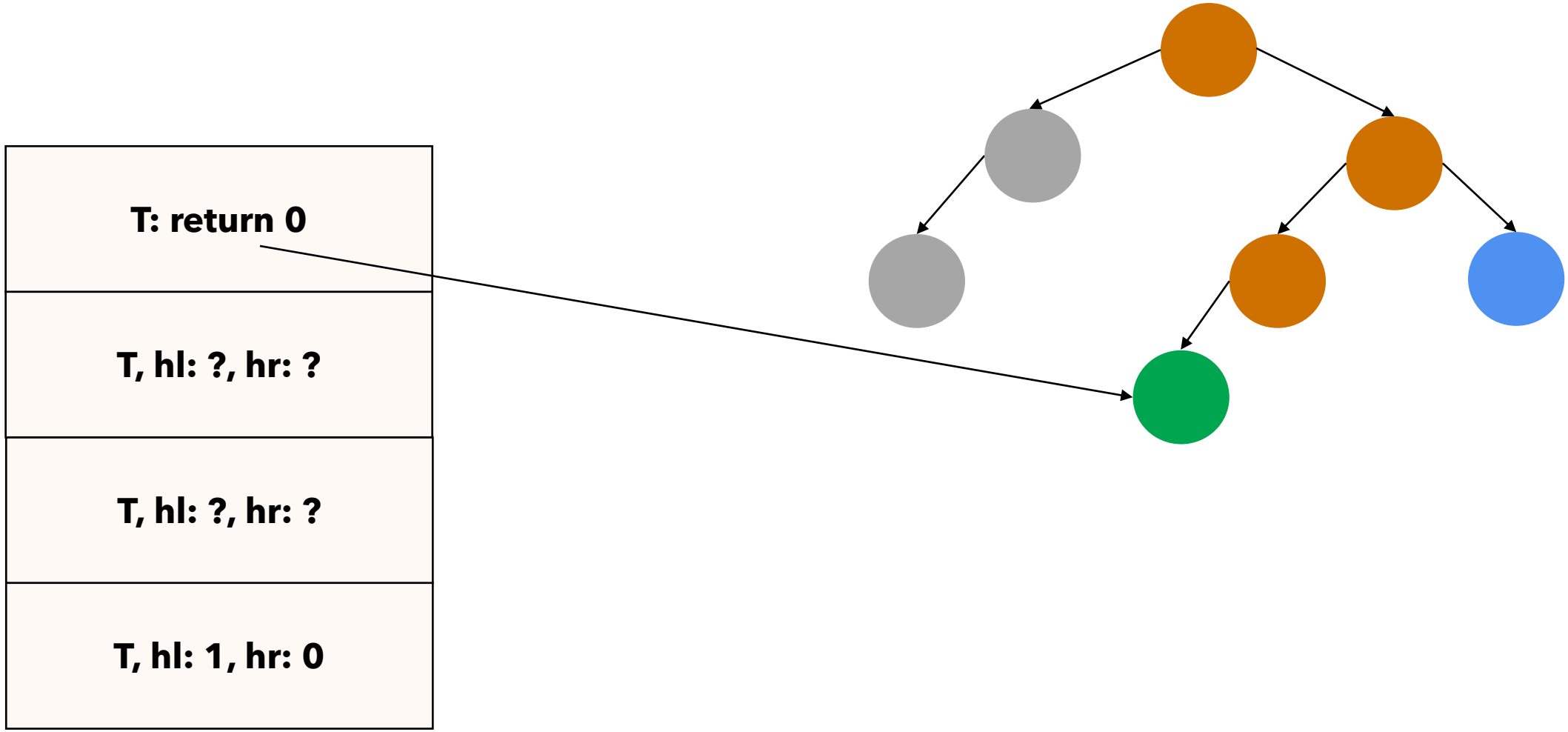




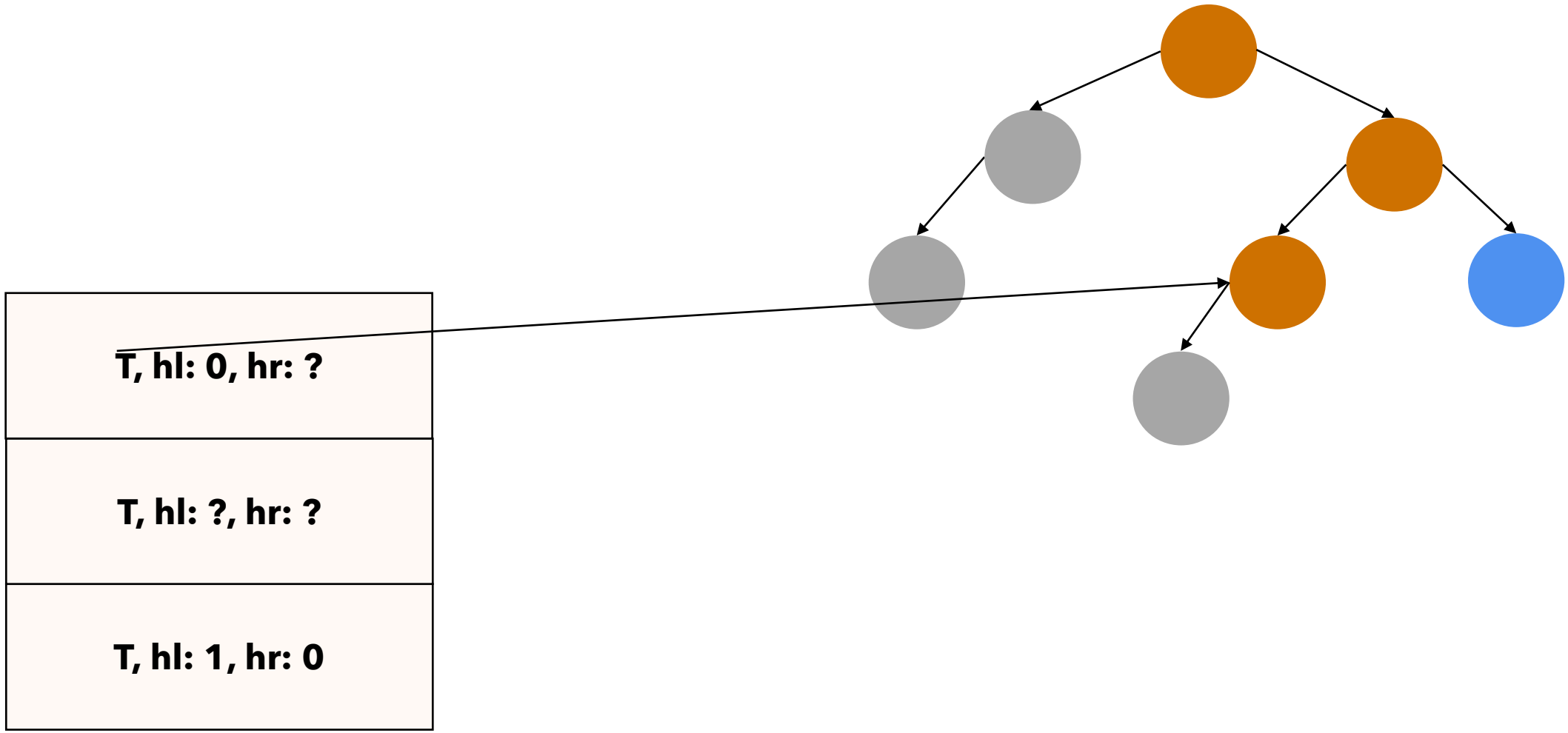
‘Srotoliamo’ la ricorsione



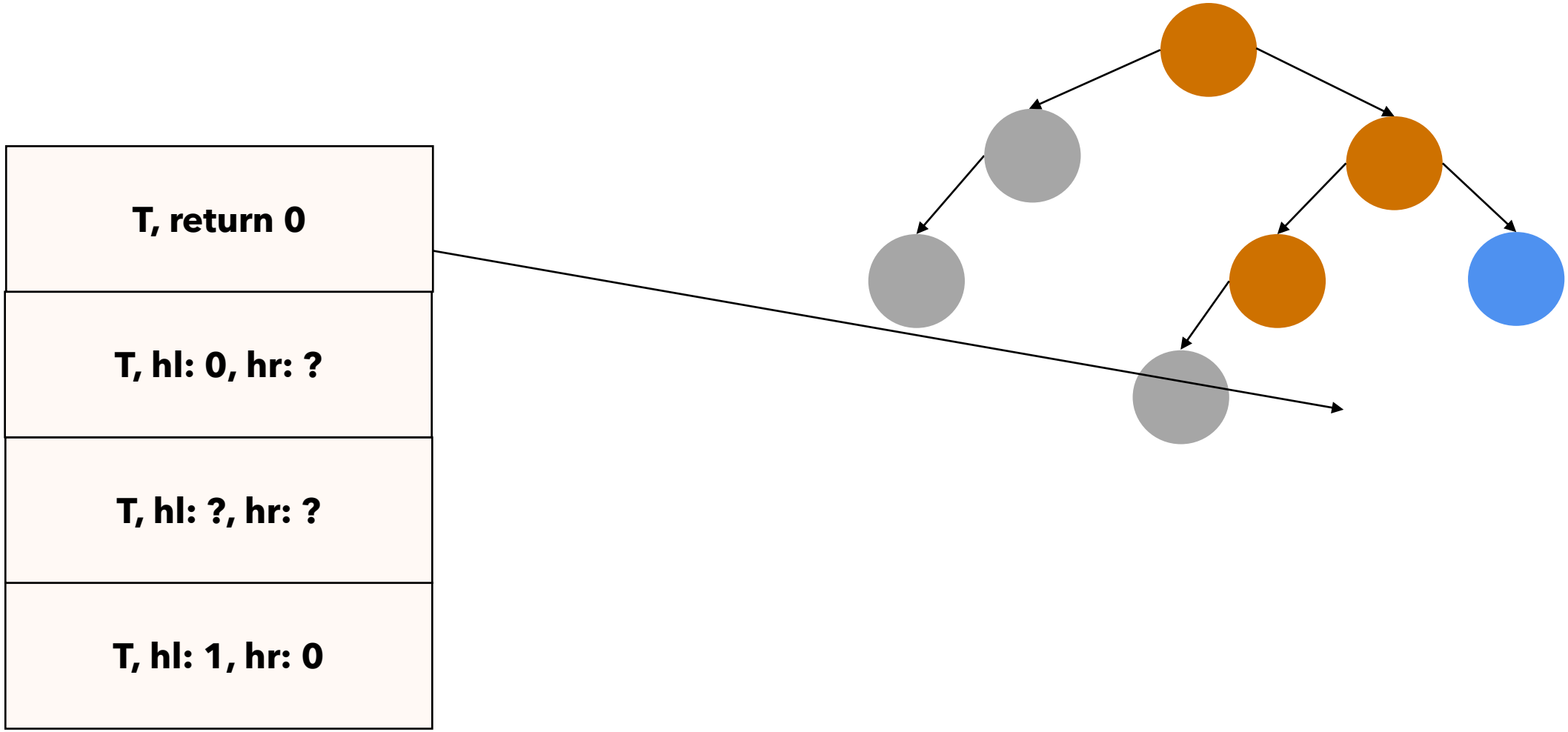
‘Srotoliamo’ la ricorsione



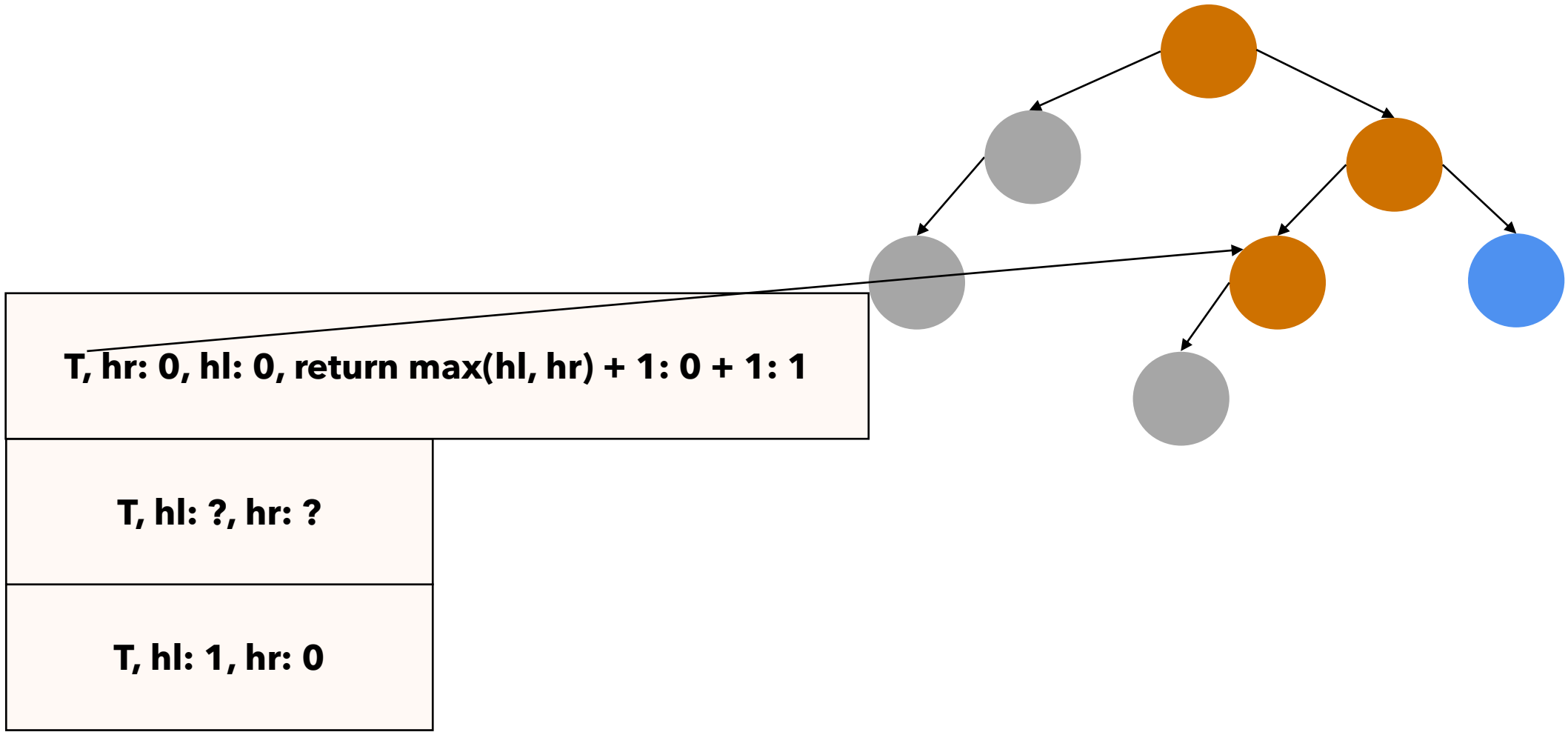
‘Srotoliamo’ la ricorsione



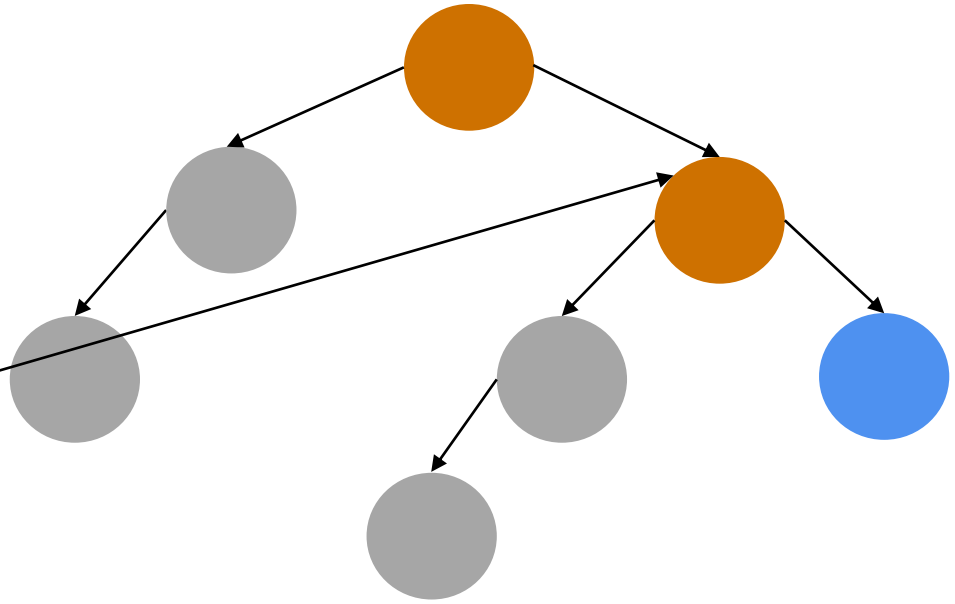
‘Srotoliamo’ la ricorsione

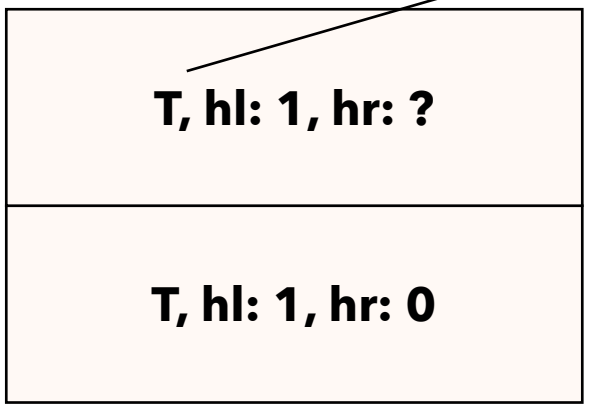


‘Srotoliamo’ la ricorsione

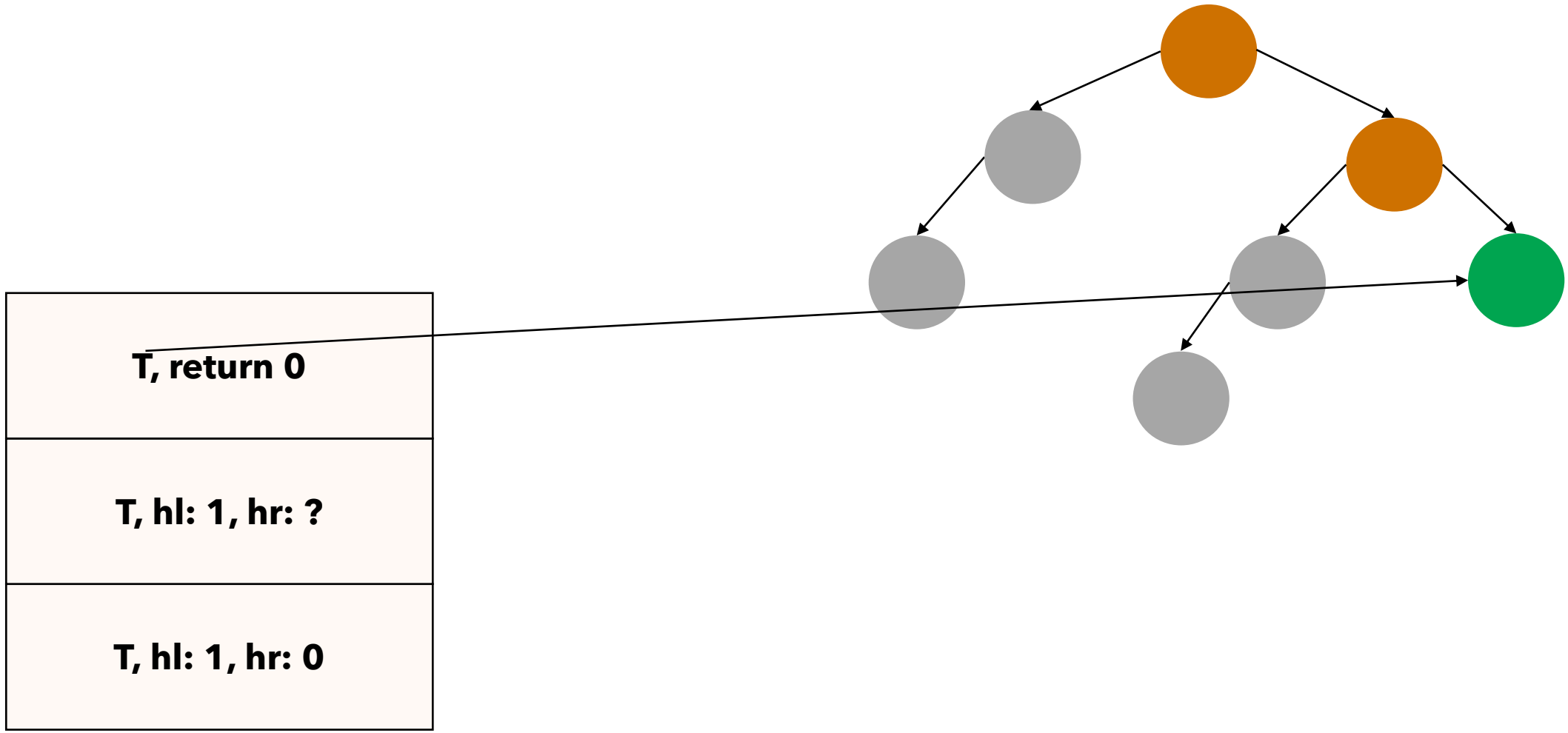


‘Srotoliamo’ la ricorsione

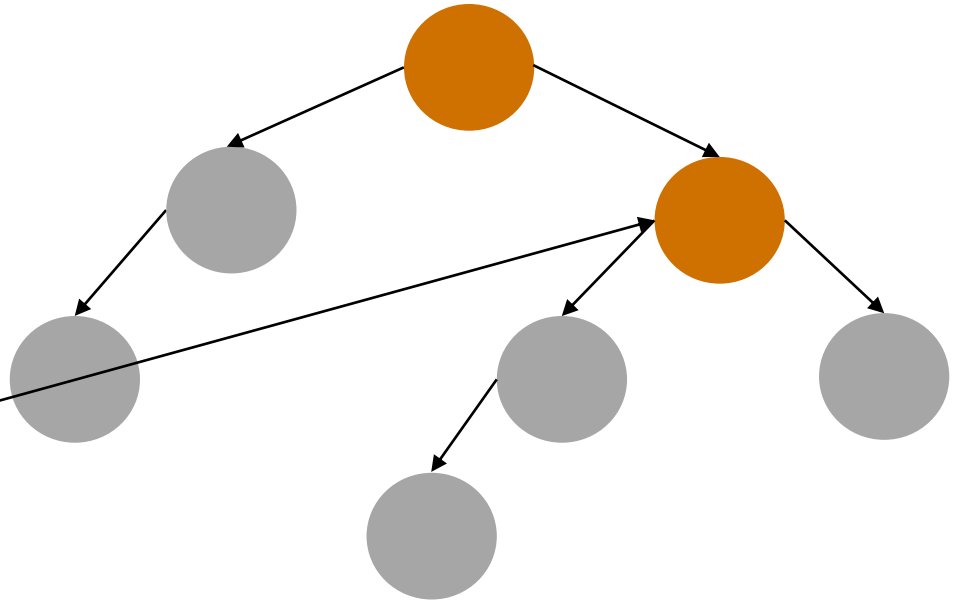




‘Srotoliamo’ la ricorsione



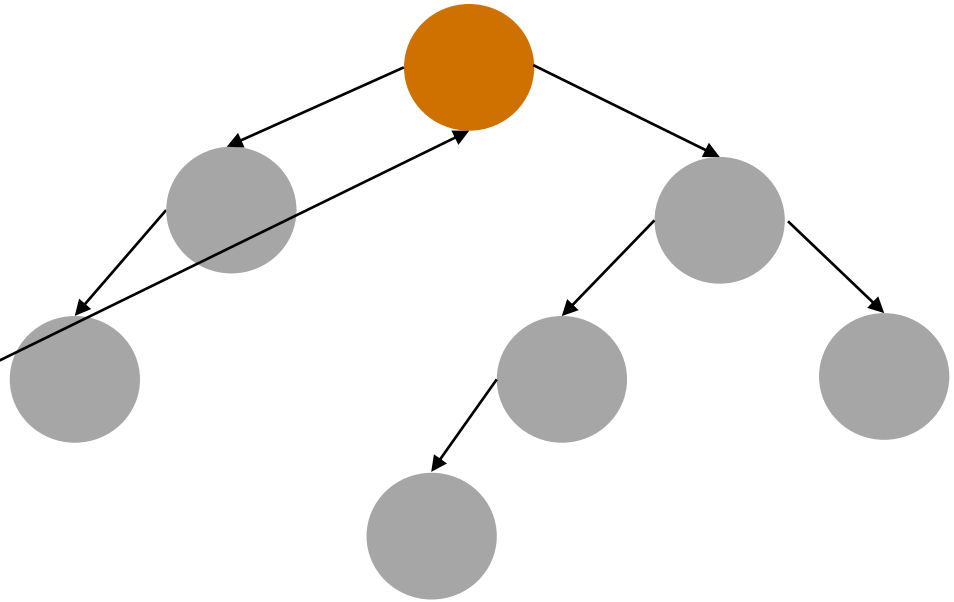
‘Srotoliamo’ la ricorsione



T, hl: 1, hr: 0, return max(hl, hr) + 1: 1 + 1: 2

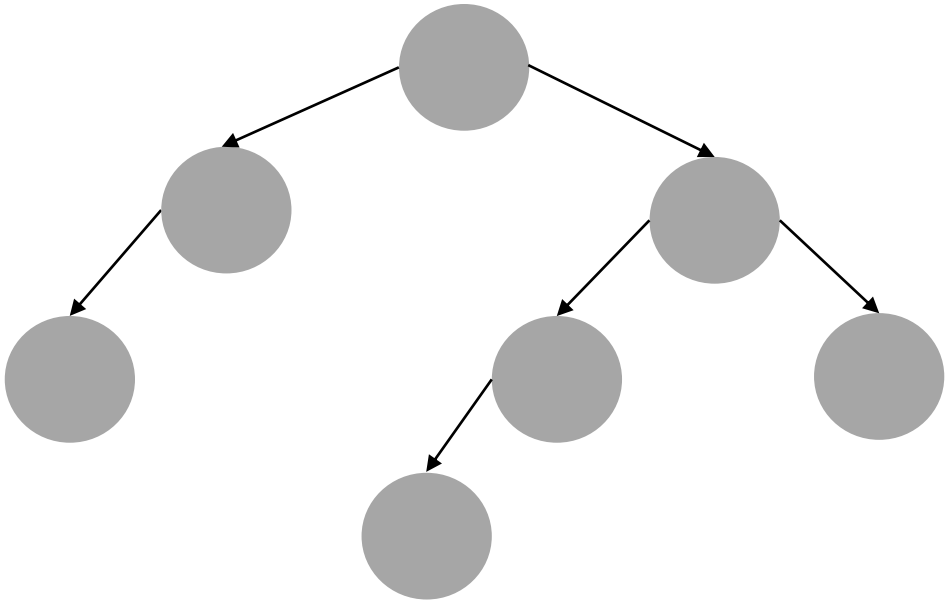
T, hl: 1, hr: 0

‘Srotoliamo’ la ricorsione



T, hl: 1, hr: 2, return max(hl, lr) + 1: 2 + 1: 3

‘Srotoliamo’ la ricorsione



La funzione `preOrderWalkAndPrint`

- Scriviamo un metodo utile per stampare su standard output un albero, in modo chiaro
- Per ora potrebbe risultare misteriosa e magica, più avanti la spieghiamo meglio

- Vogliamo stampare un albero in questa forma:

```
T.key(T.leftSubtree, T.rightSubTree)
```

T.leftSubTree vanno stampati ricorsivamente nello stesso modo
Se *T.leftSubTree* o *T.rightSubTree* sono alberi vuoti, stamperemo:

```
nil
```

La funzione **preOrderWalkAndPrint**

```
preOrderWalkAndPrint(T)
        if T is nil:
            print 'nil'
            return
        print T.key
        print '('
        preOrderWalkAndPrint(T.left)
        print ', '
        preOrderWalkAndPrint(T.right)
        print ')'
```