Algoritmi sugli alberi binari

Liceo G.B. Brocchi
Classi quarte Scientifico – opzione scienze applicate
Bassano del Grappa, Gennaio 2023
Prof. Giovanni Mazzocchin

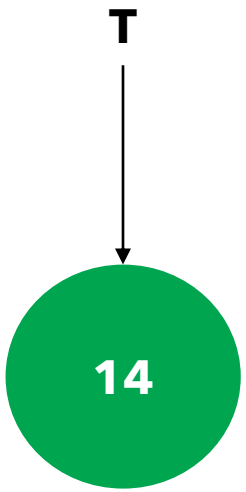
Inserimento di un nodo in un *BST*

bst_insert(T, key):

- se l'albero T è vuoto, allora allora un nuovo nodo e restituiscine il puntatore
- altrimenti, se key è minore o uguale a T.key, inserisci key nel sottoalbero sinistro di T, fallo puntare da T.left, e restituisci T
- altrimenti, se key è maggiore di T.key, inserisci key nel sottoalbero destro di T, fallo puntare da T.right, e restituisci T

Inserimento di un nodo in un *BST*

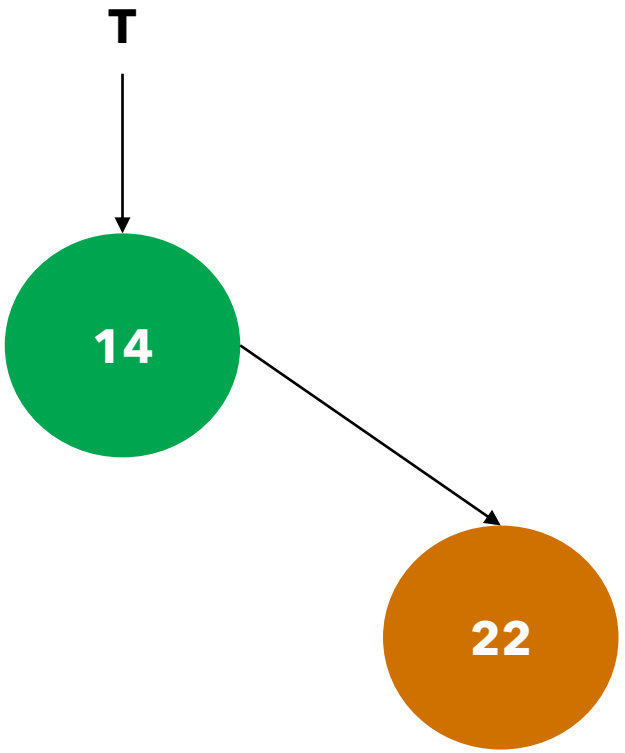
```
T = nil
bst_insert(T, 14)

Caso base, si crea un nuovo
nodo e lo si fa puntare da T
```



Inserimento di un nodo in un *BST*

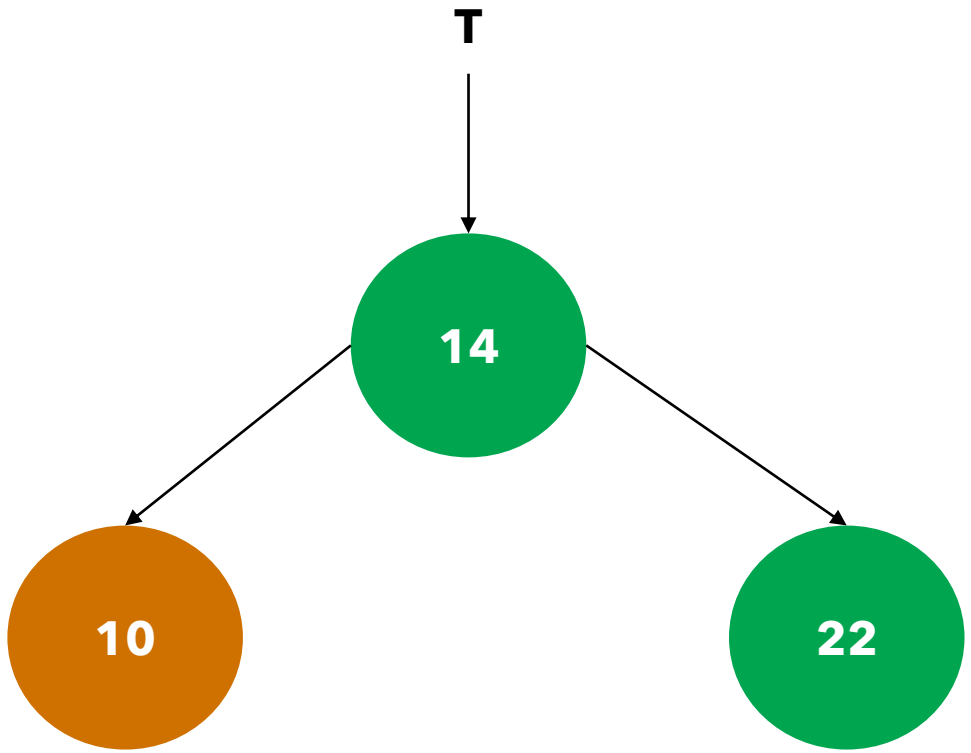
```
bst_insert(T, 22)
```

Caso ricorsivo: 22 > 14, quindi si inserisce un nuovo nodo con chiave 22 nel sottoalbero destro di T. Per mantenere i collegamenti, T.right deve puntare al nuovo nodo



Inserimento di un nodo in un *BST*

```
bst_insert(T, 10)

Caso ricorsivo: 10 < 14,
quindi si inserisce un nuovo nodo
con chiave 10 nel sottoalbero
sinistro. Per mantenere i
collegamenti, T.left deve puntare
al nuovo nodo
```



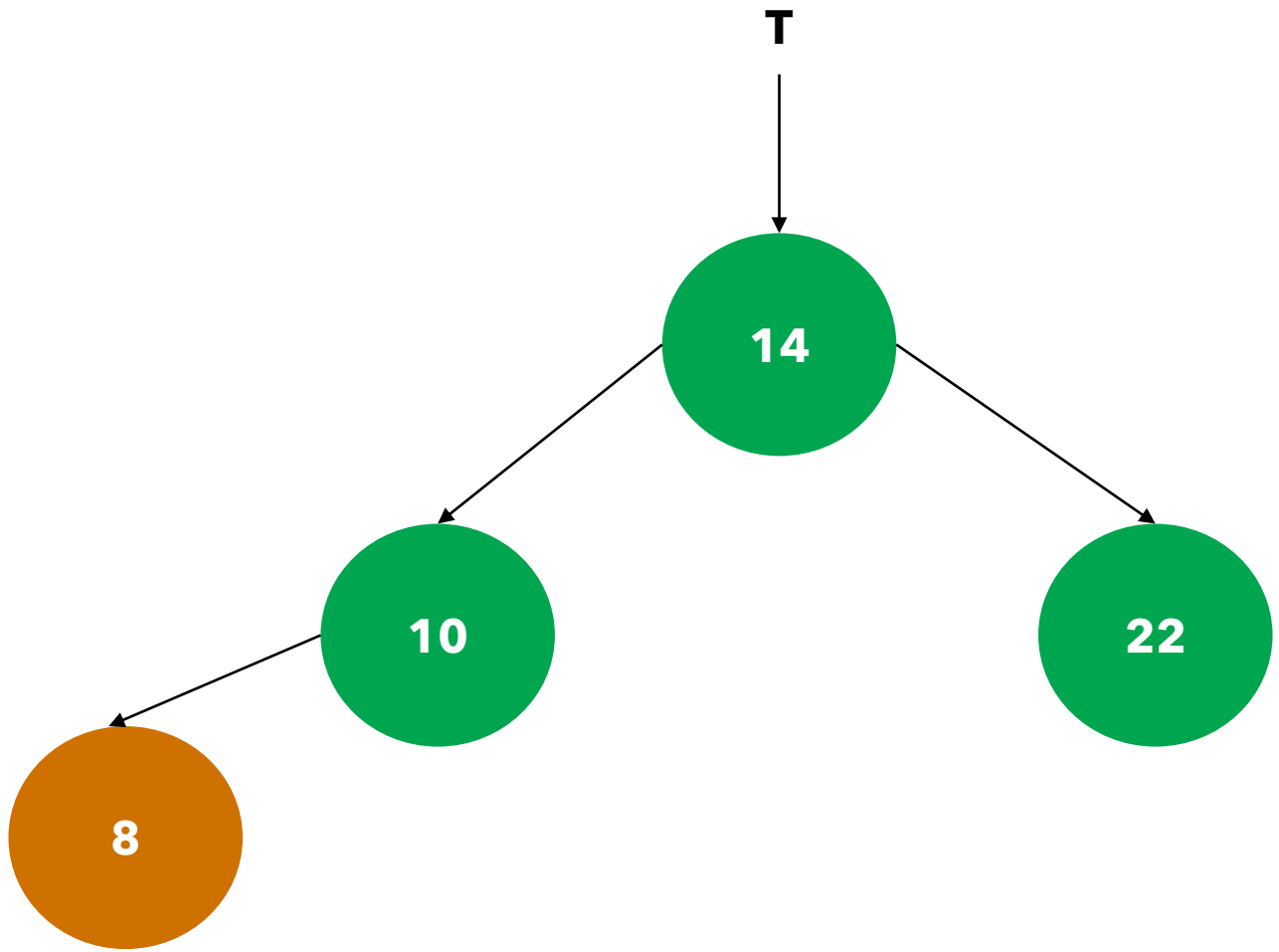
Inserimento di un nodo in un *BST*

```
bst_insert(T, 8
```

Caso ricorsivo: 8 < 14,
quindi si inserisce un nuovo nodo con chiave 8 nel sottoalbero sinistro di T;

Caso ricorsivo: 8 < 10: si inserisce un nuovo nodo con chiave 8 nel sottoalbero sinistro di T.left;

Caso base: si crea un nodo con chiave 8 e lo si fa puntare da T.left.left

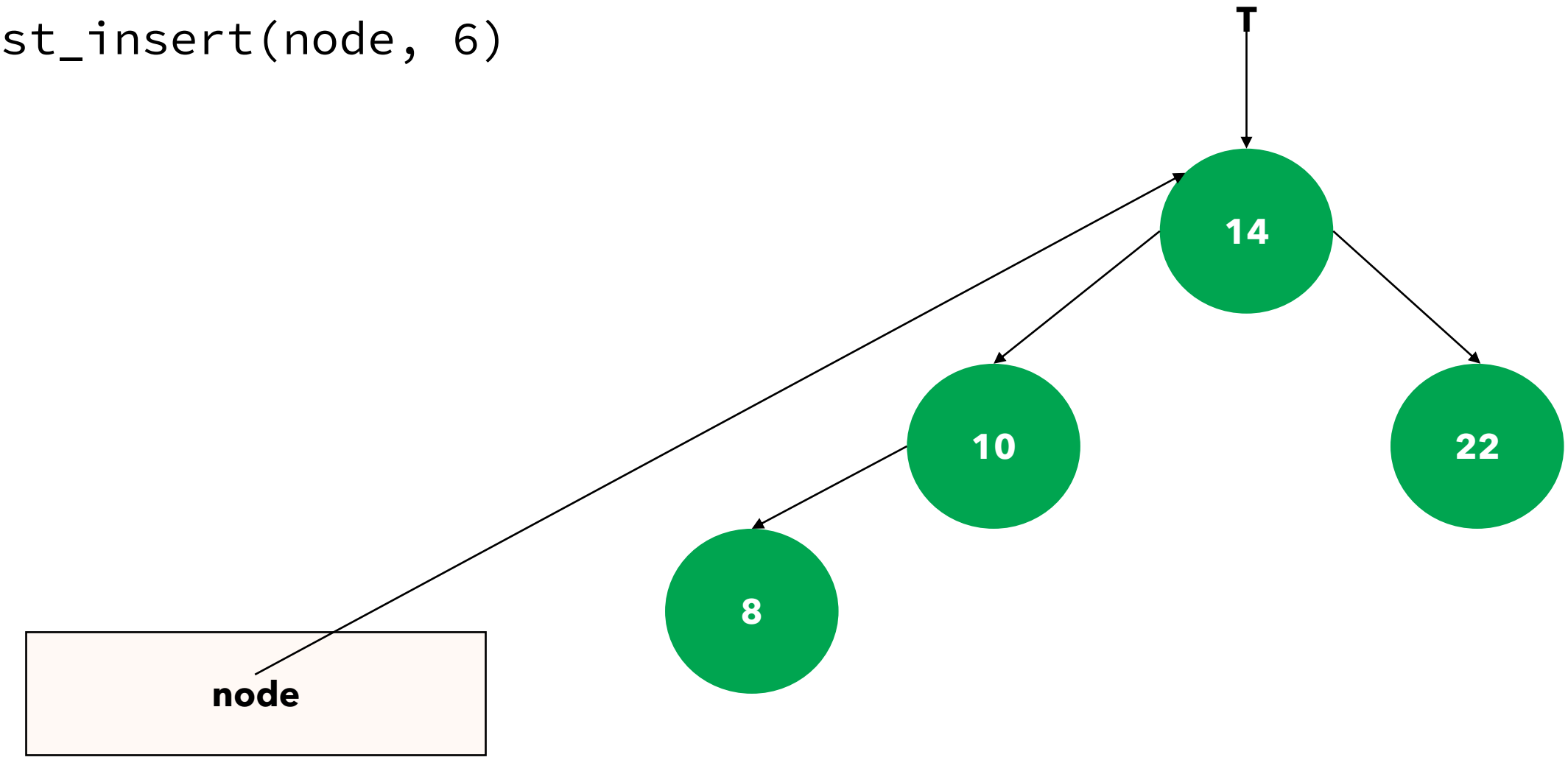


Inserimento di un nodo in un *BST*

```
TREE_NODE *bst_insert(TREE_NODE *node, int k) {
  if (!node) {
    TREE_NODE *new_node = (TREE_NODE*) malloc(sizeof(TREE_NODE));
    new_node->key = k;
    new_node->left = NULL;
    new_node->right = NULL;
    return new_node;
  }
  else if (k <= node->key) {
    node->left = bst_insert(node->left, k);
  }
  else {
    node->right = bst_insert(node->right, k);
  }

  return node;
}
```

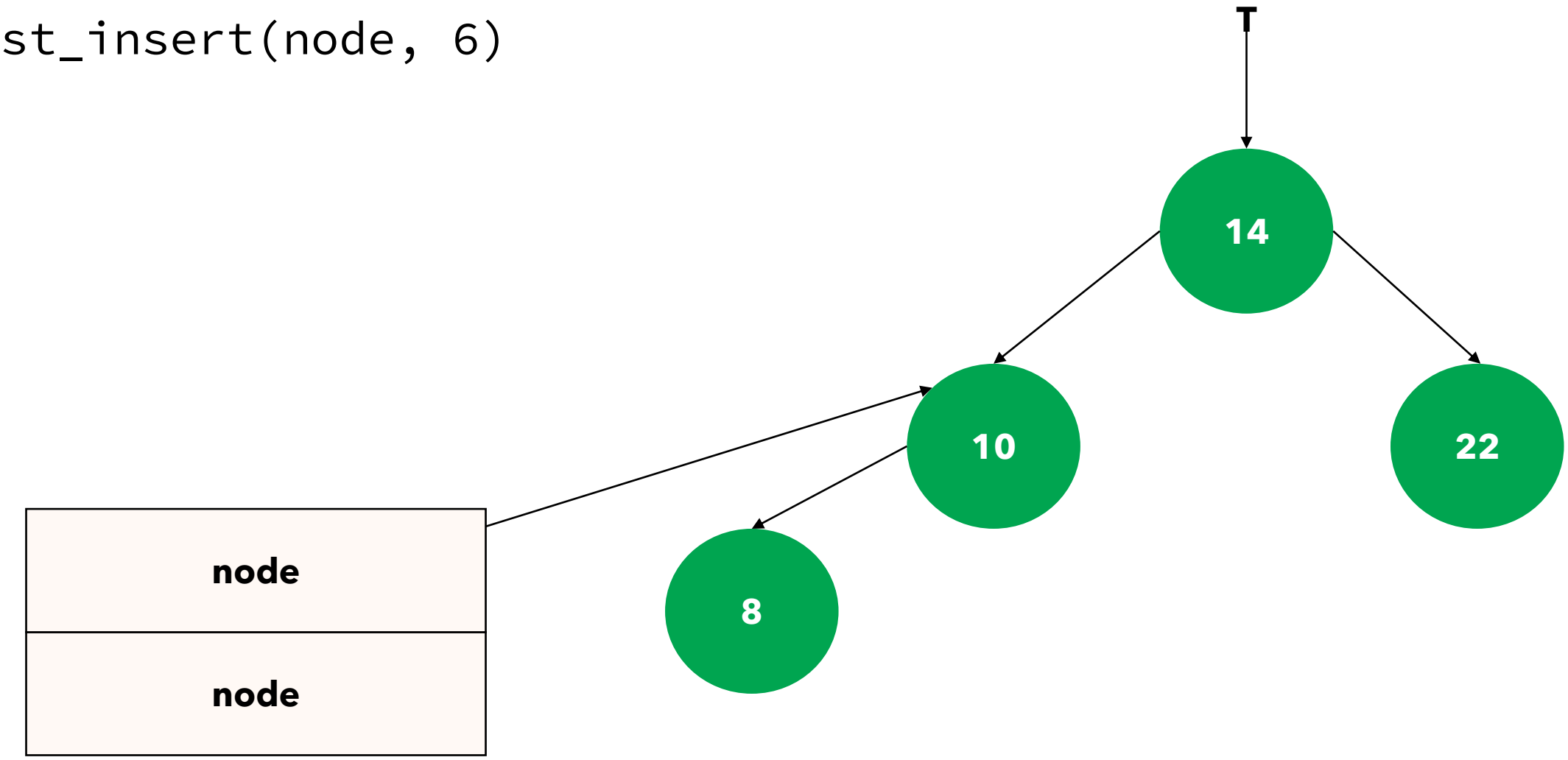
Inserimento di un nodo in un *BST*

```
bst_insert(node, 6)
```



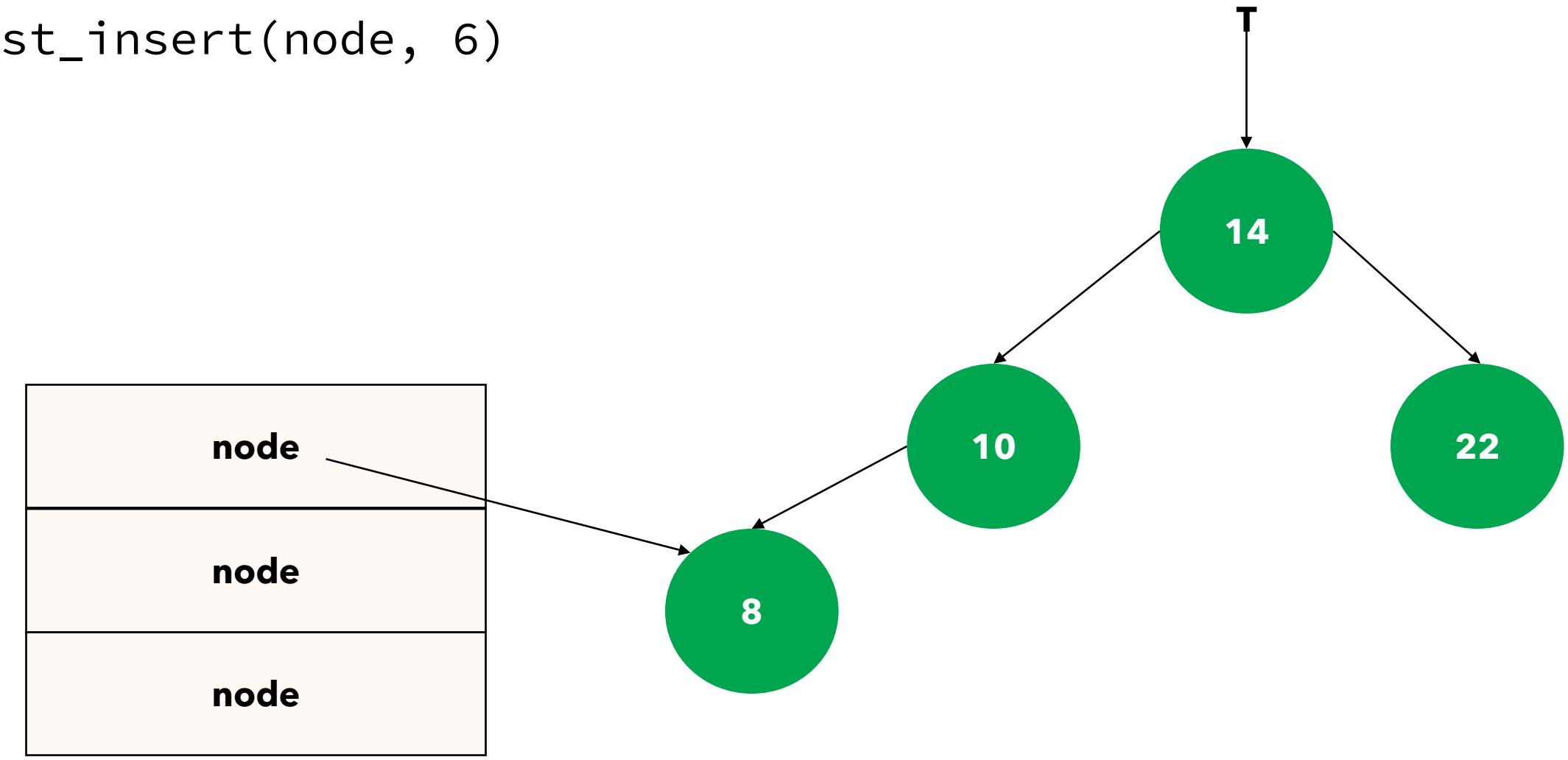
Inserimento di un nodo in un *BST*

```
bst_insert(node, 6)
```



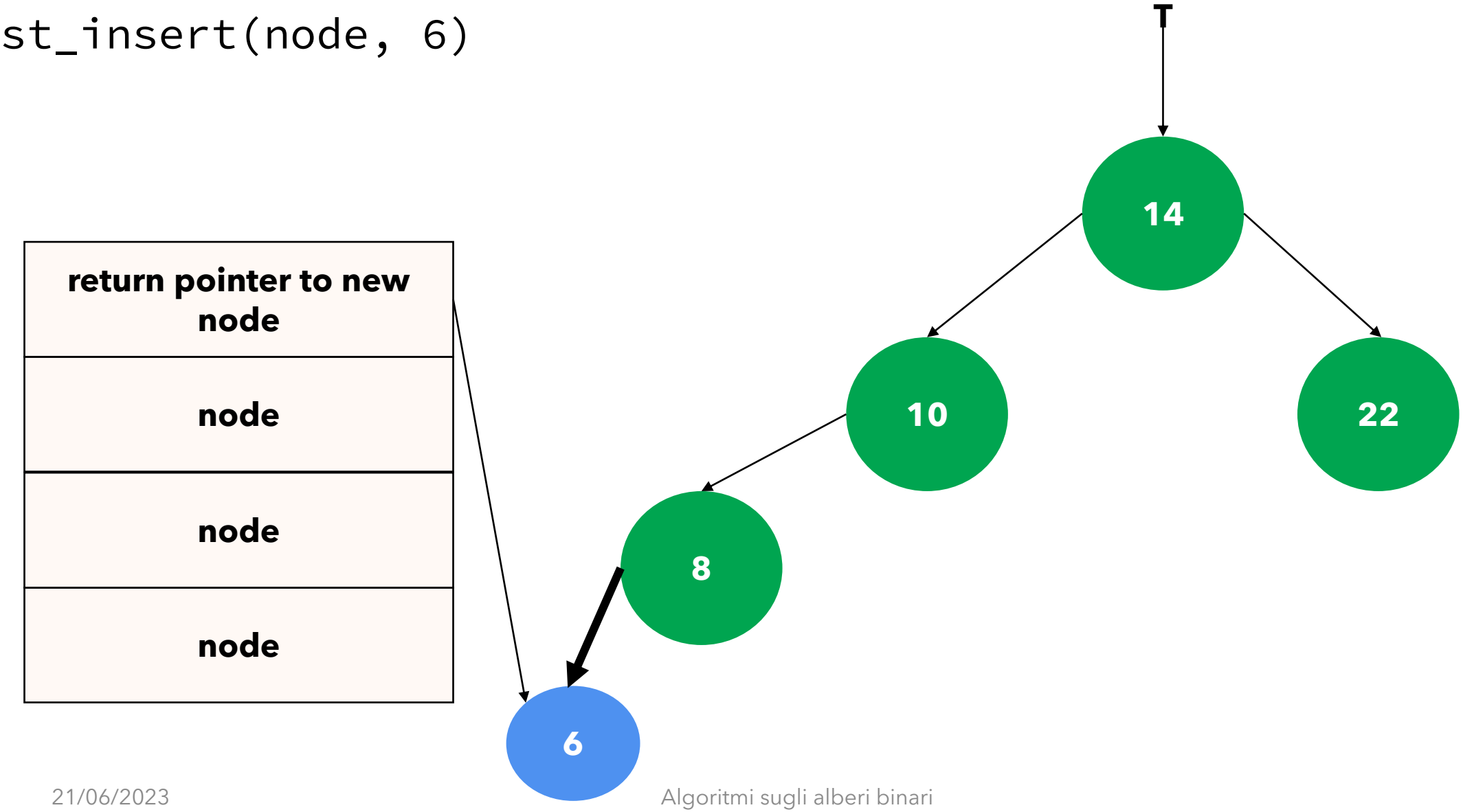
Inserimento di un nodo in un *BST*

```
bst_insert(node, 6)
```



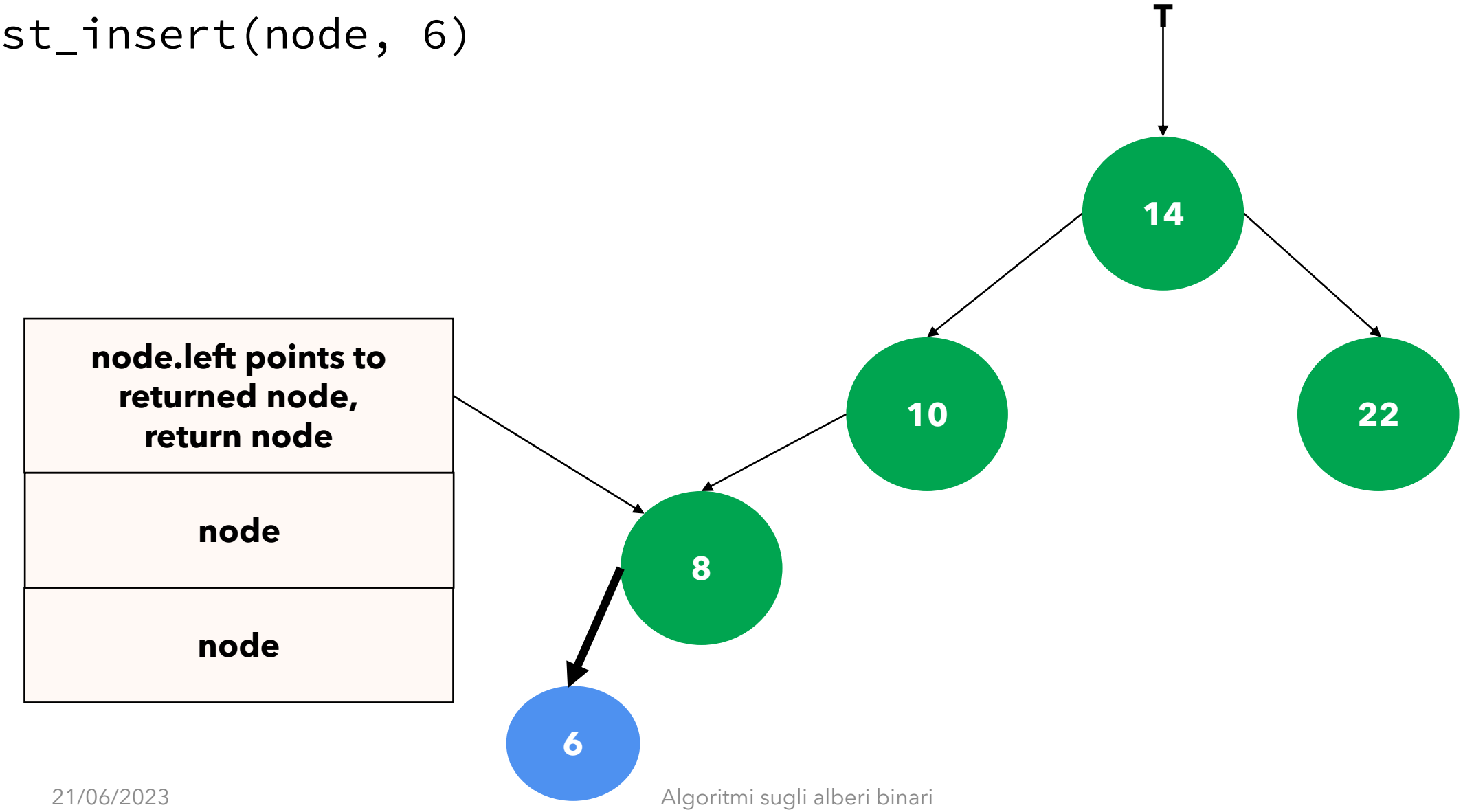
Inserimento di un nodo in un *BST*

```
bst_insert(node, 6)
```



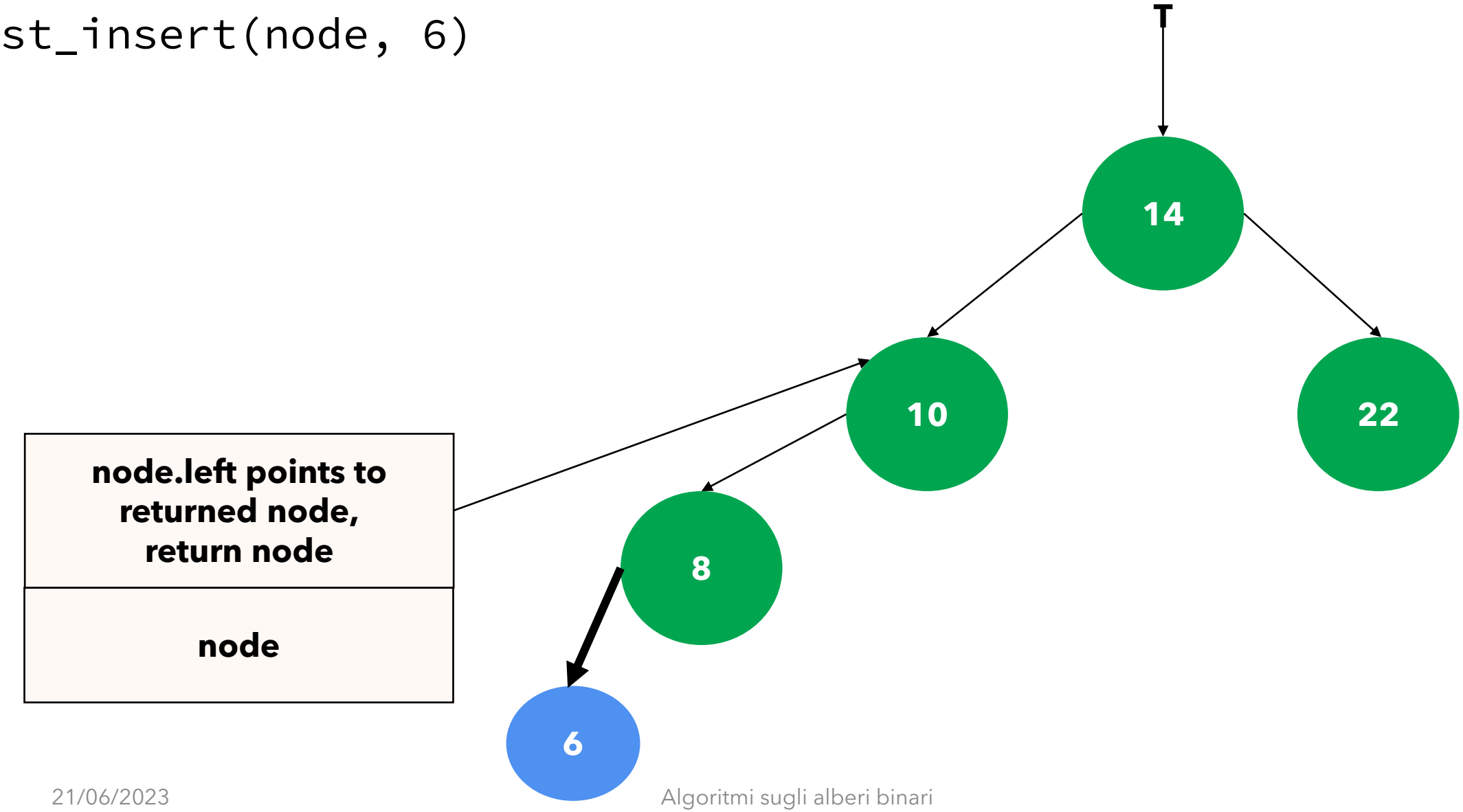
Inserimento di un nodo in un *BST*

```
bst_insert(node, 6)
```



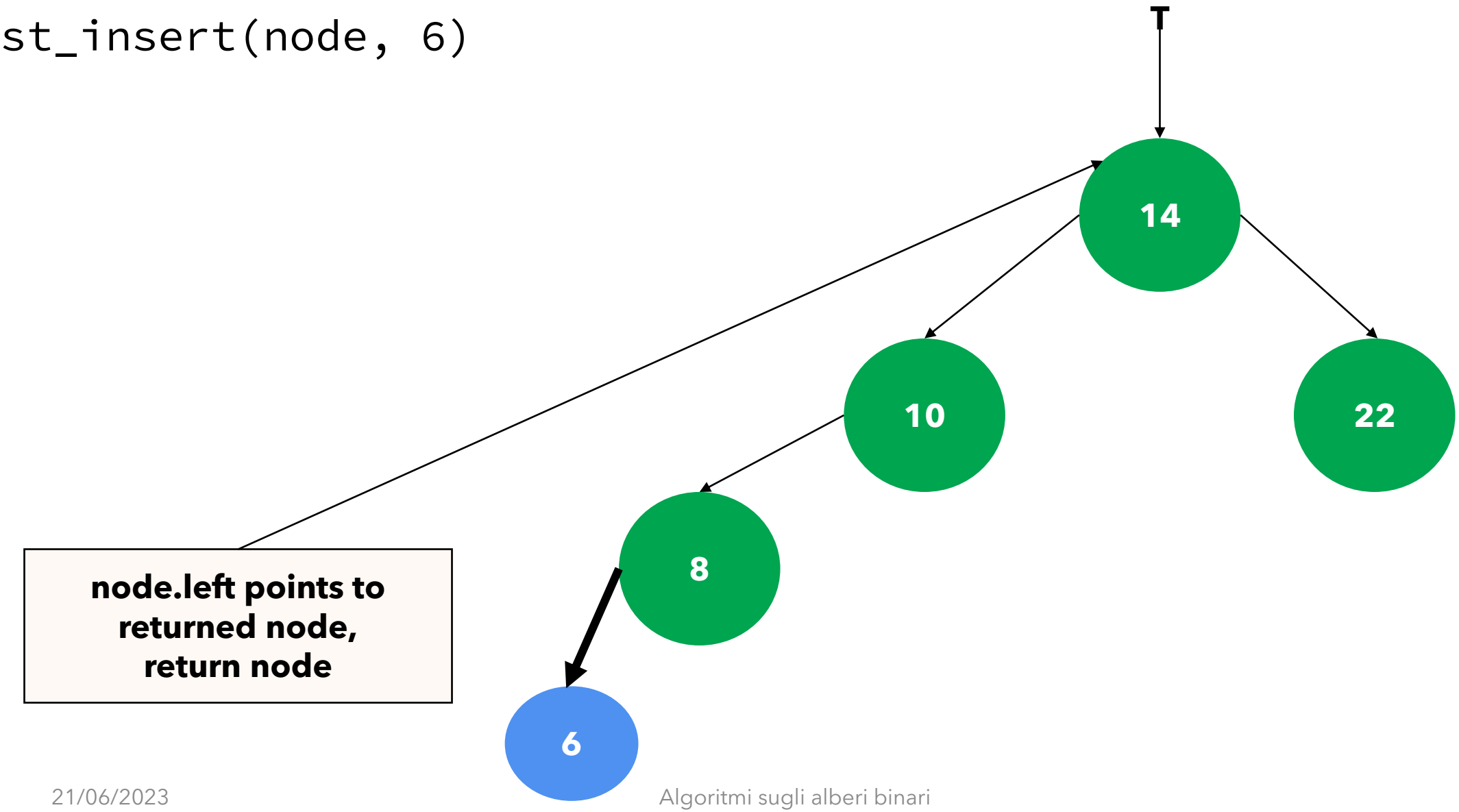
Inserimento di un nodo in un *BST*

```
bst_insert(node, 6)
```



Inserimento di un nodo in un *BST*

```
bst_insert(node, 6)
```

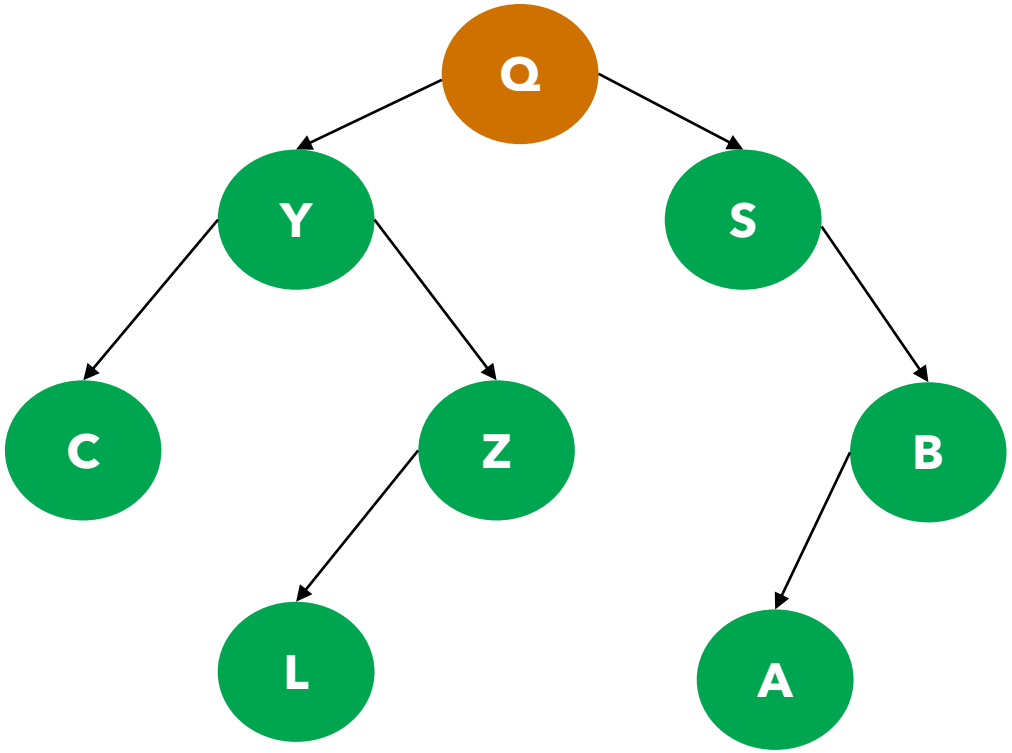


Depth-first search (*DFS*) su alberi binari

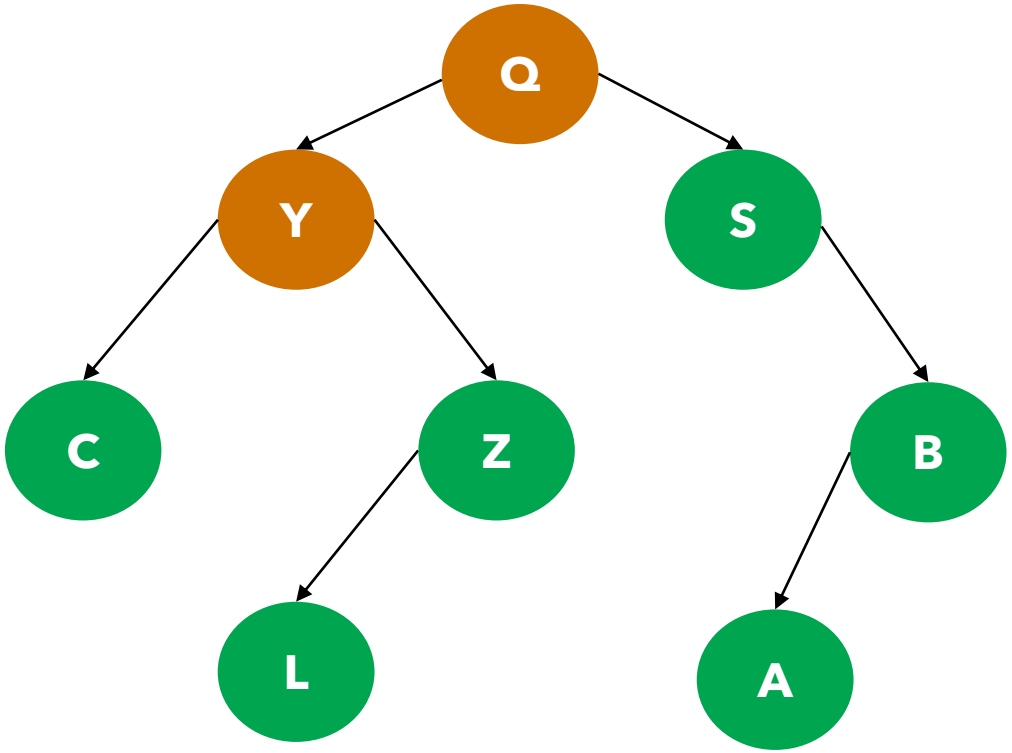
- Nella ricerca in profondità (**depth-first**), per ogni nodo vengono visitati ricorsivamente in profondità prima il figlio sinistro e poi il figlio destro
- Questo significa che prima di *tornare indietro* ad un nodo *T* (*backtracking*), uno dei due sottoalberi di *T* deve essere visitato completamente
- Per tornare indietro fino al punto giusto e non ripetere gli stessi percorsi serve uno **stack**. Ovviamente la ricorsione qui ci aiuta molto, in quanto si basa già sul *call stack* del programma

- Esistono diverse versioni della ricerca in profondità (***depth-first***)
  - visita **preorder**
  - visita **inorder**
  - visita **postorder**

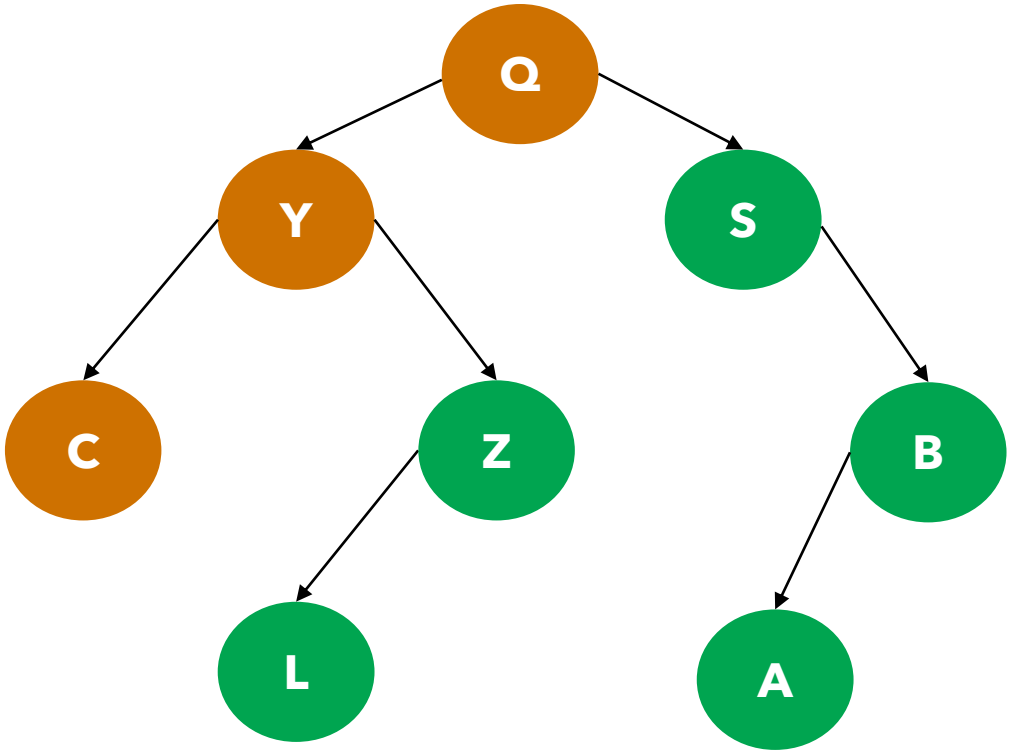
Depth-first search (*DFS*) su alberi binari



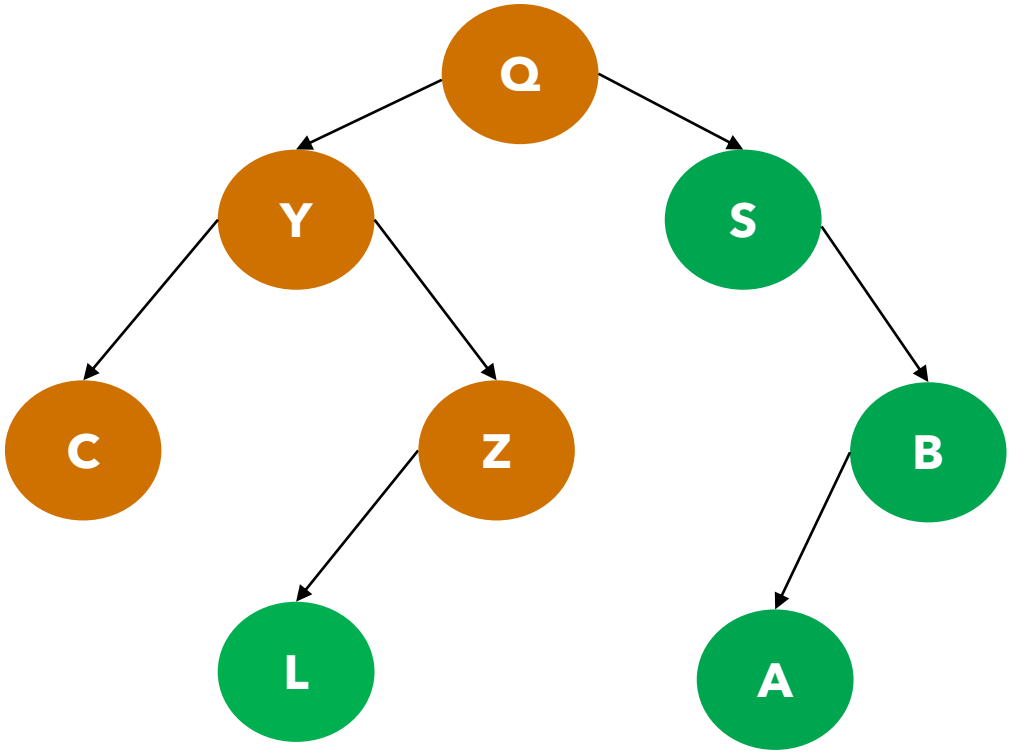
Depth-first search (*DFS*) su alberi binari



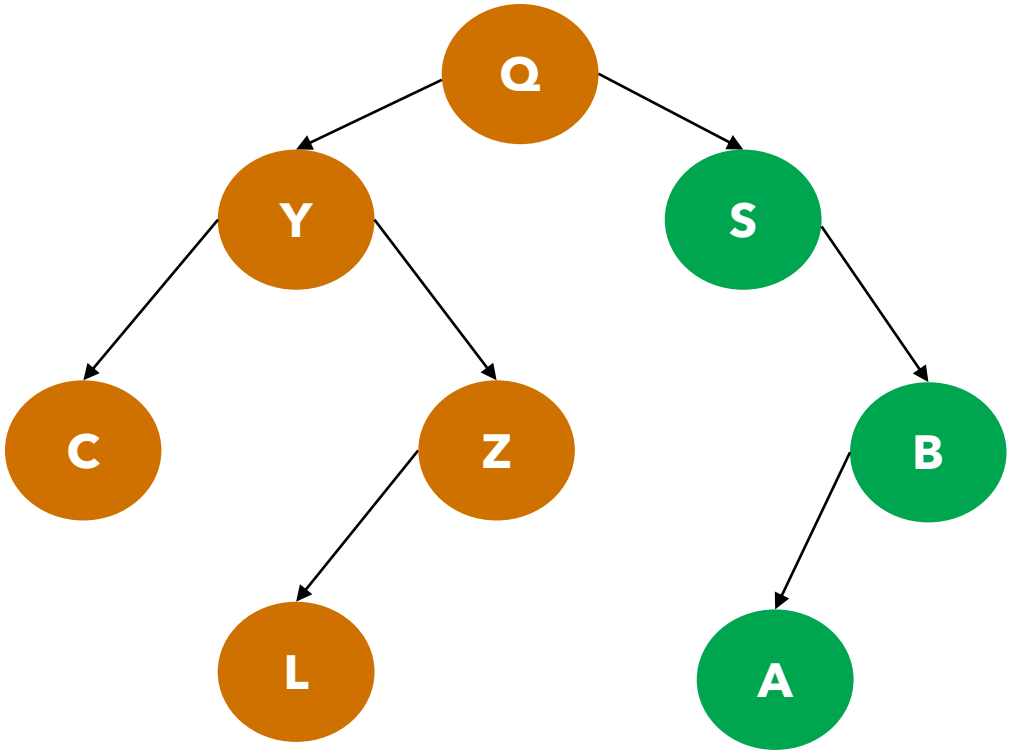
Depth-first search (*DFS*) su alberi binari



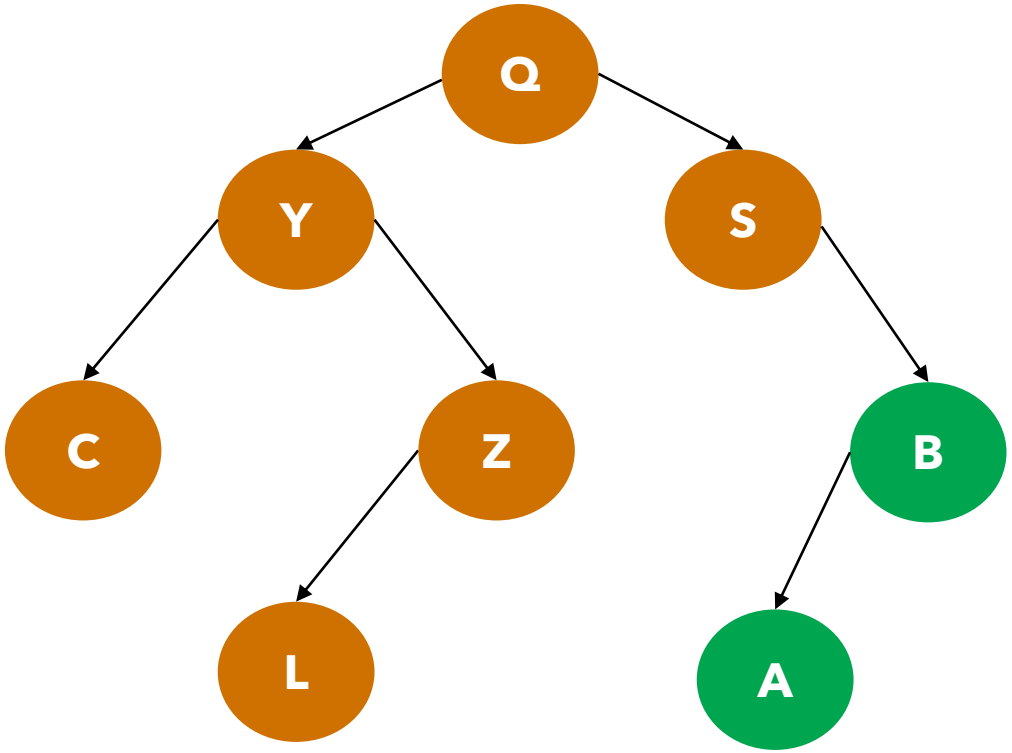
Depth-first search (*DFS*) su alberi binari



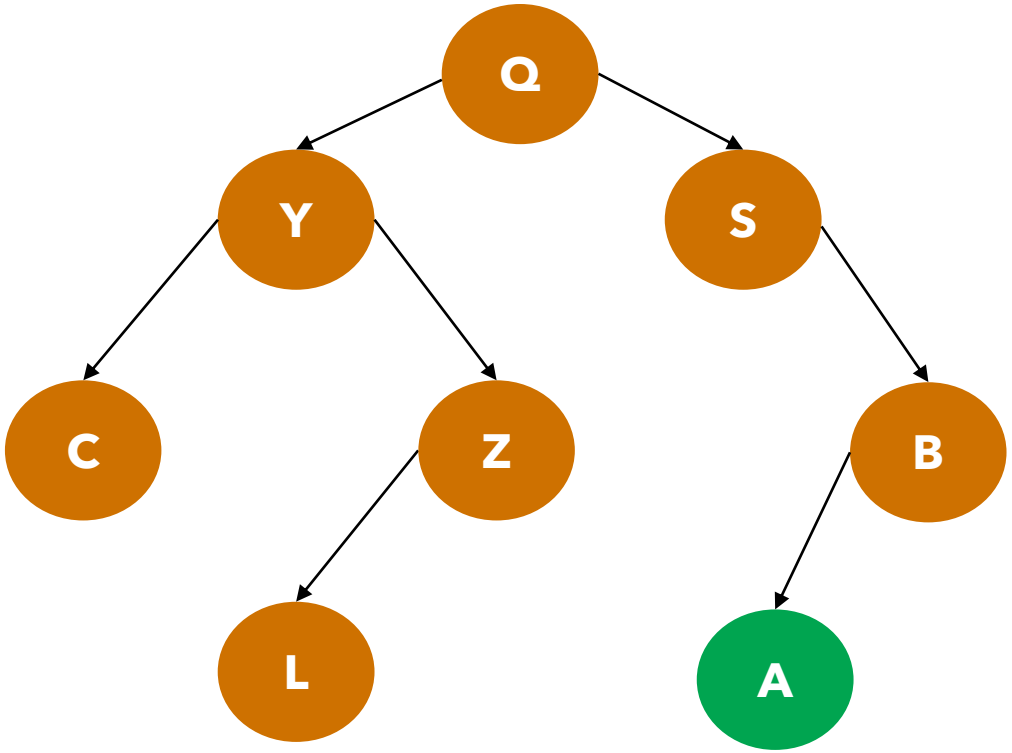
Depth-first search (*DFS*) su alberi binari



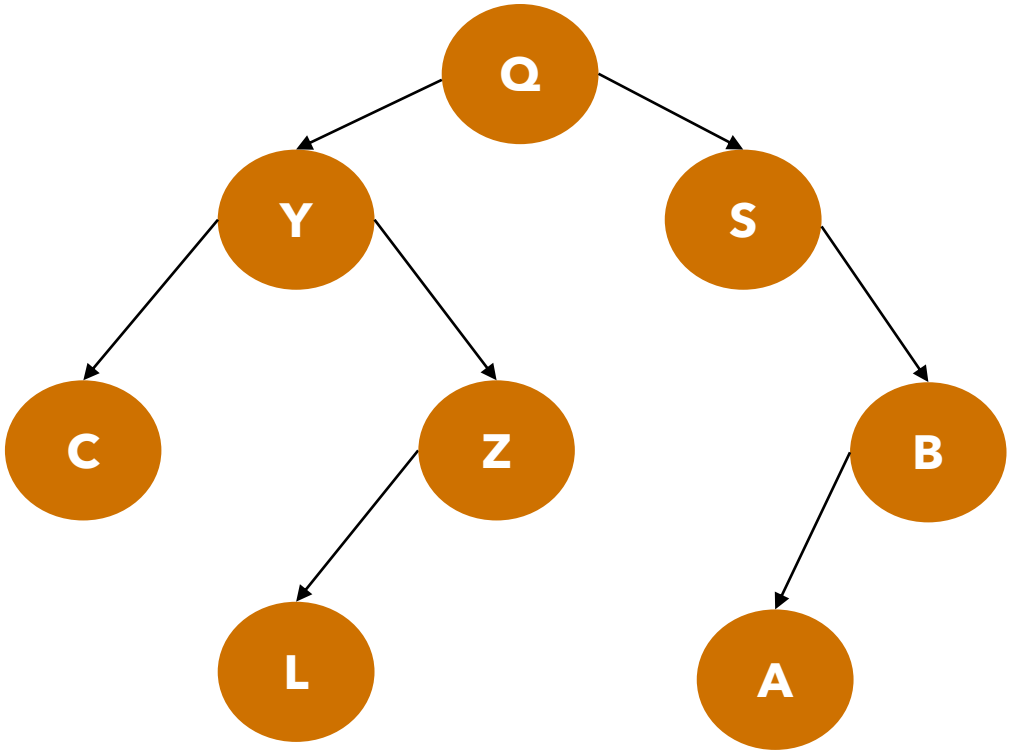
Depth-first search (*DFS*) su alberi binari



Depth-first search (*DFS*) su alberi binari



Depth-first search (DFS) su alberi binari



una ricerca in profondità su questo albero visita i nodi in questo ordine:
Q -> Y -> C -> Z -> L -> S -> B -> A

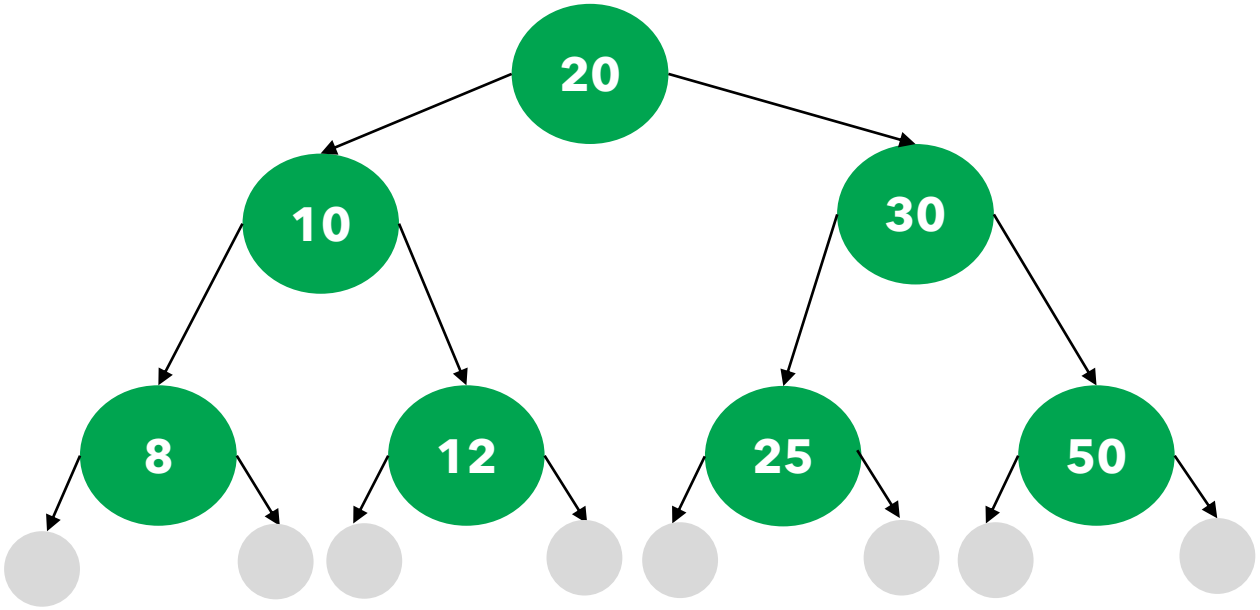
Esempio di *DFS*: visita `inorder` di un *BST*

`inorder_tree_walk(T):`

- se l'albero T è vuoto, return
- altrimenti
  - esegui inorder_tree_walk su T.left
  - 'apri' il nodo T (ad esempio: stampa T.key, o in generale *esegui un'operazione su T*)
  - esegui inorder_tree_walk su T.right

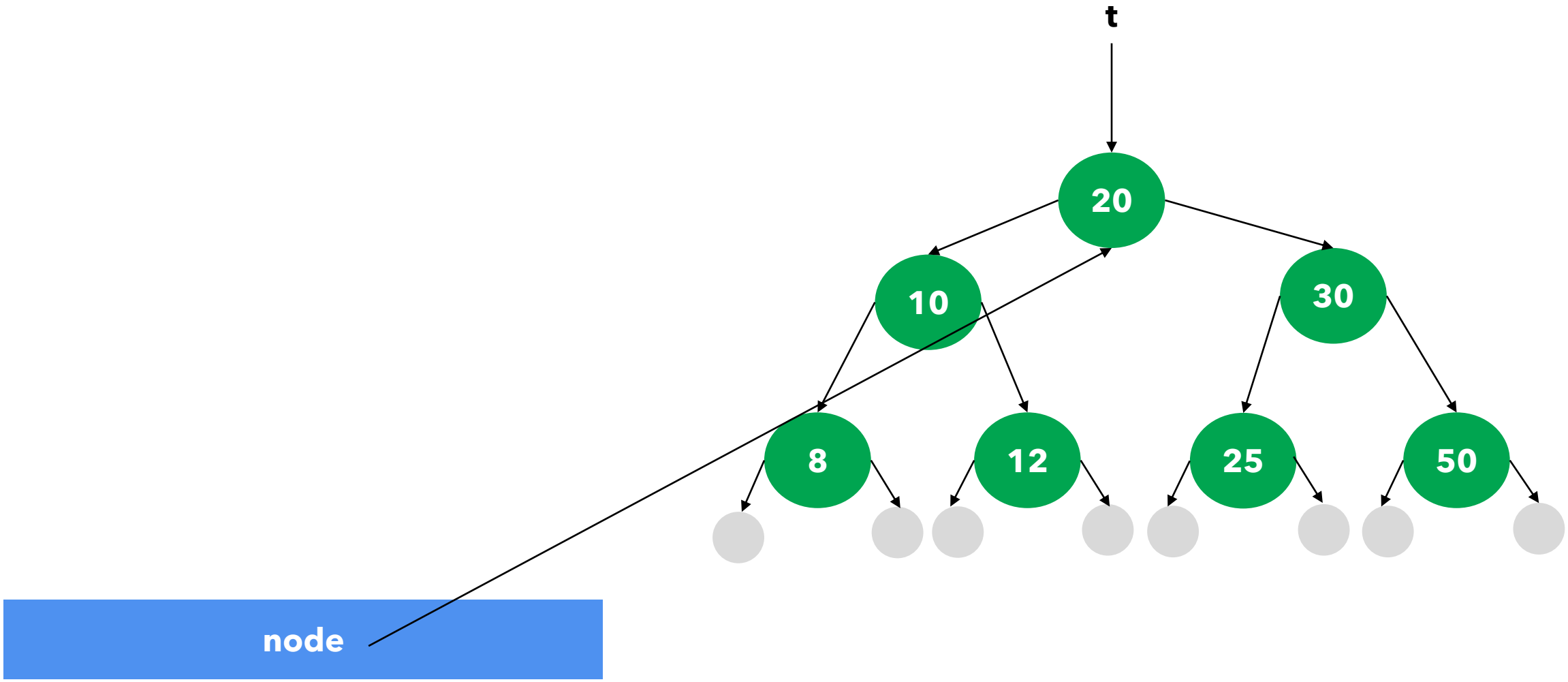
Esempio di *DFS*: visita `inorder` di un *BST*

```
void in_order(TREE_NODE *node) {
  if (node == NULL) {
    return;
  }
  in_order(node->left);
  printf("%d ", node->key);
  in_order(node->right);
}
```

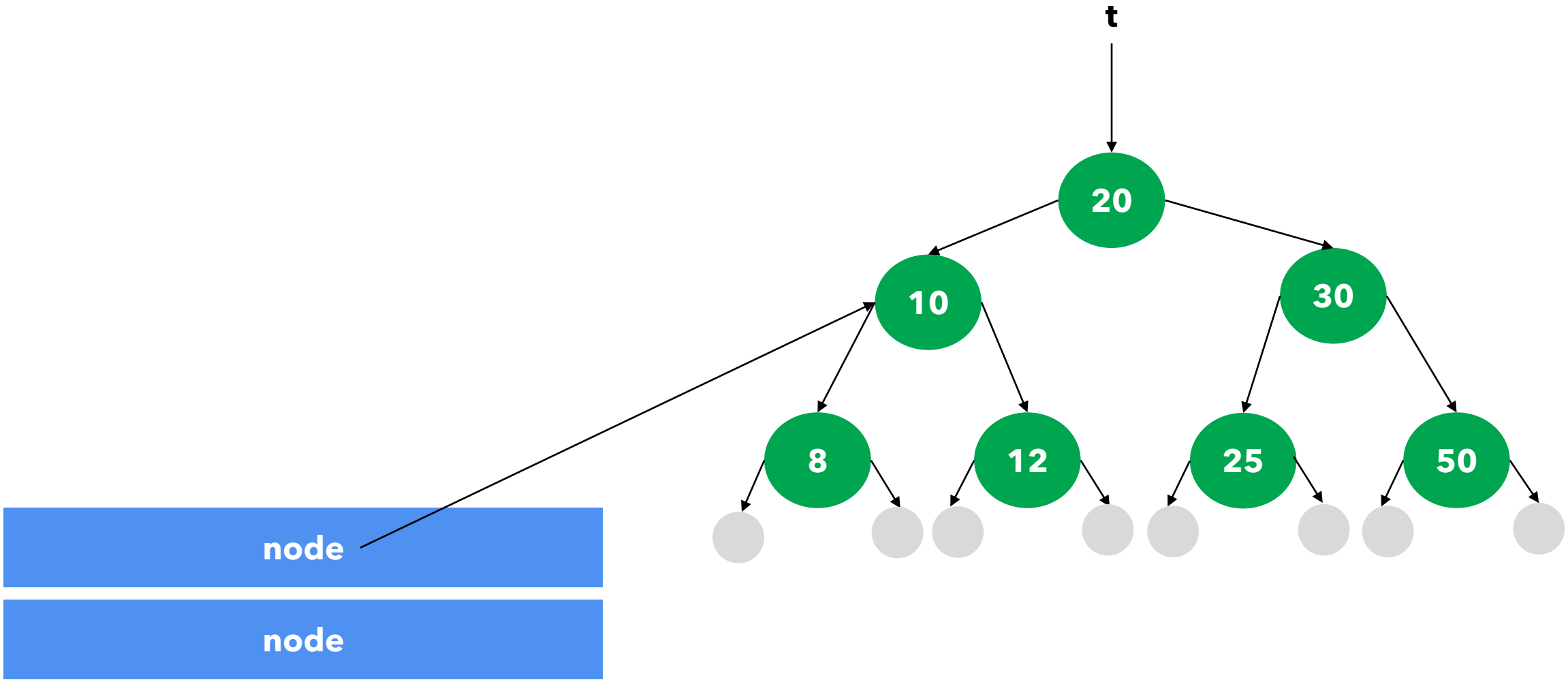


La vista *inorder* di un albero binario di ricerca stampa le chiavi dei nodi in ordine crescente

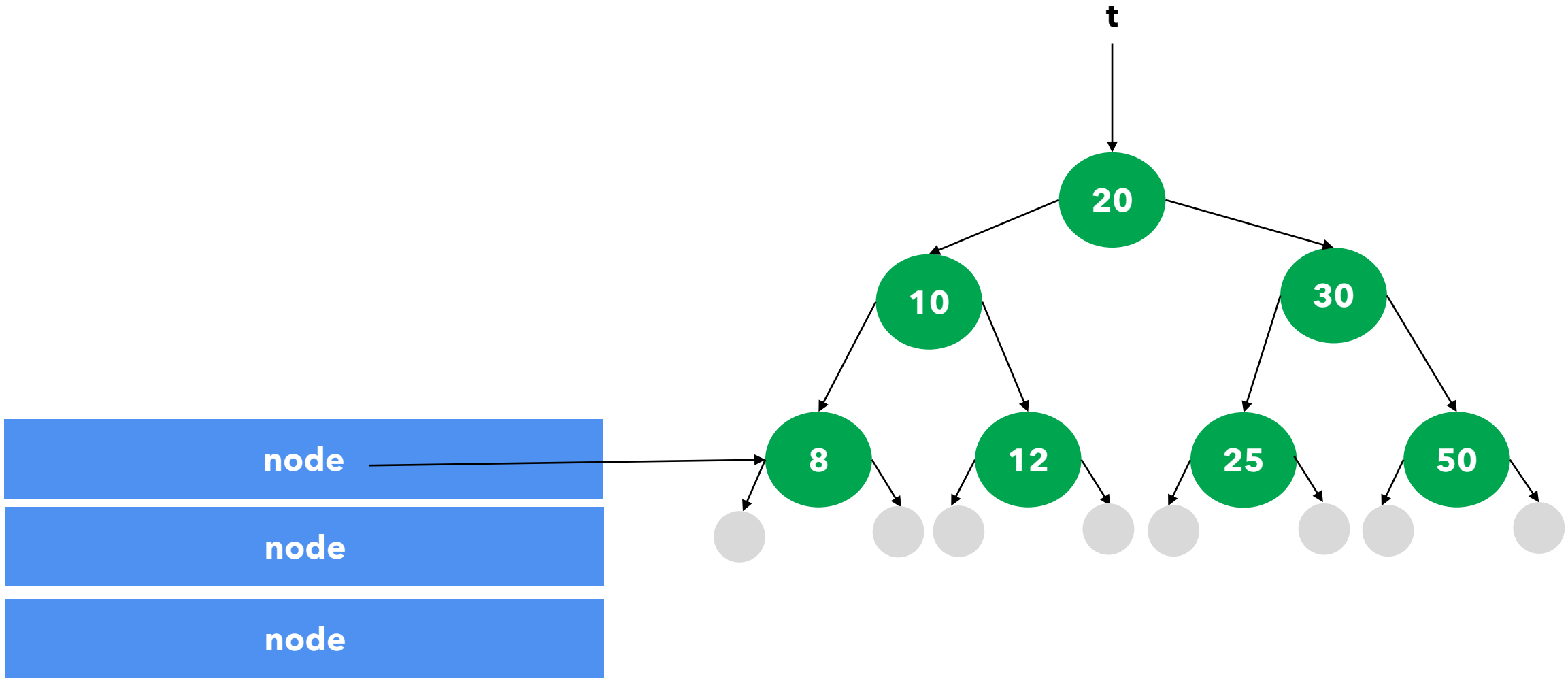
Esempio di *DFS*: visita `inorder` di un *BST*



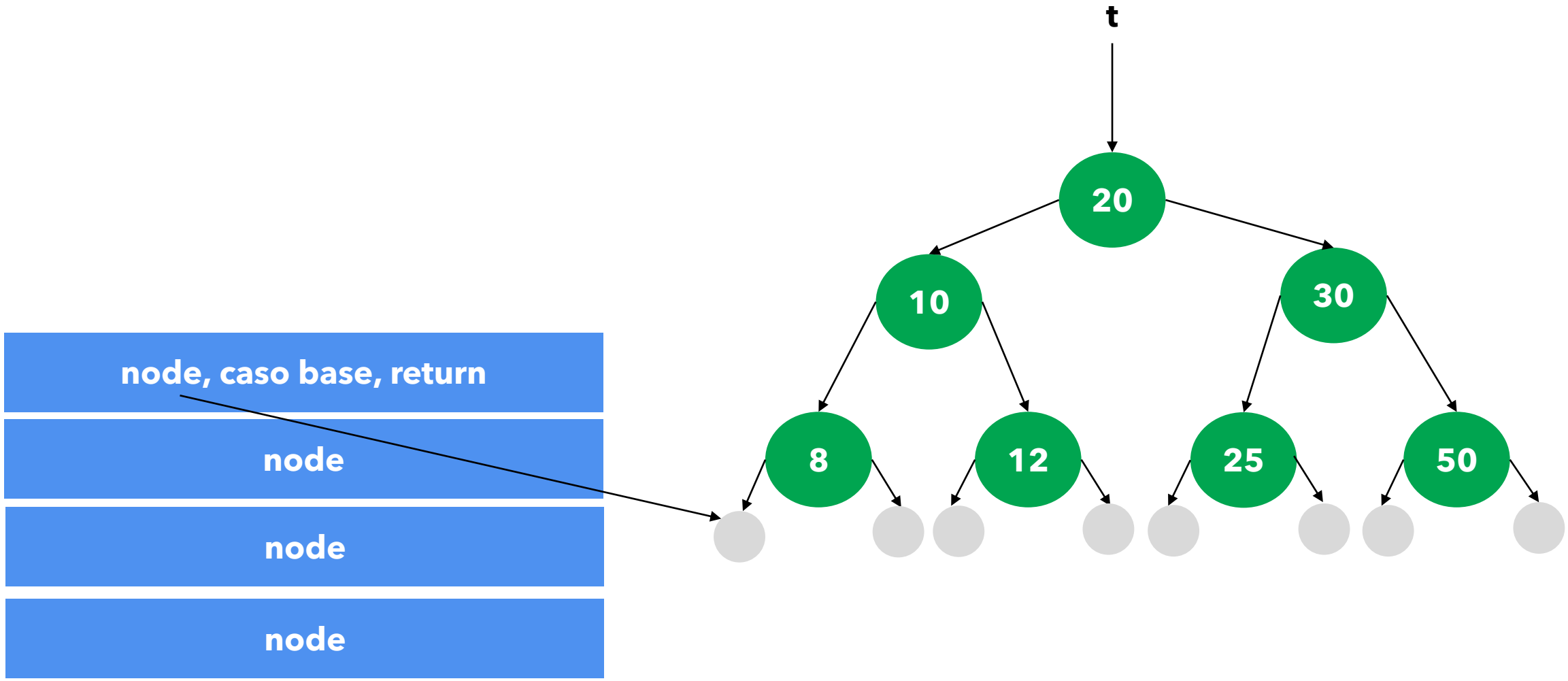
Esempio di *DFS*: visita `inorder` di un *BST*



Esempio di *DFS*: visita `inorder` di un *BST*

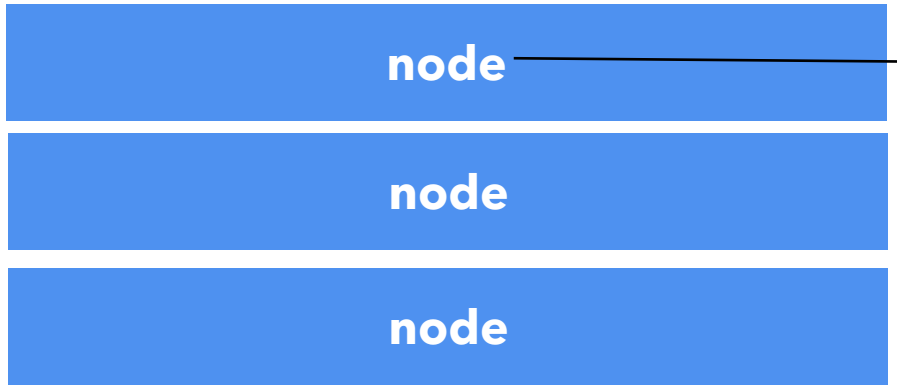


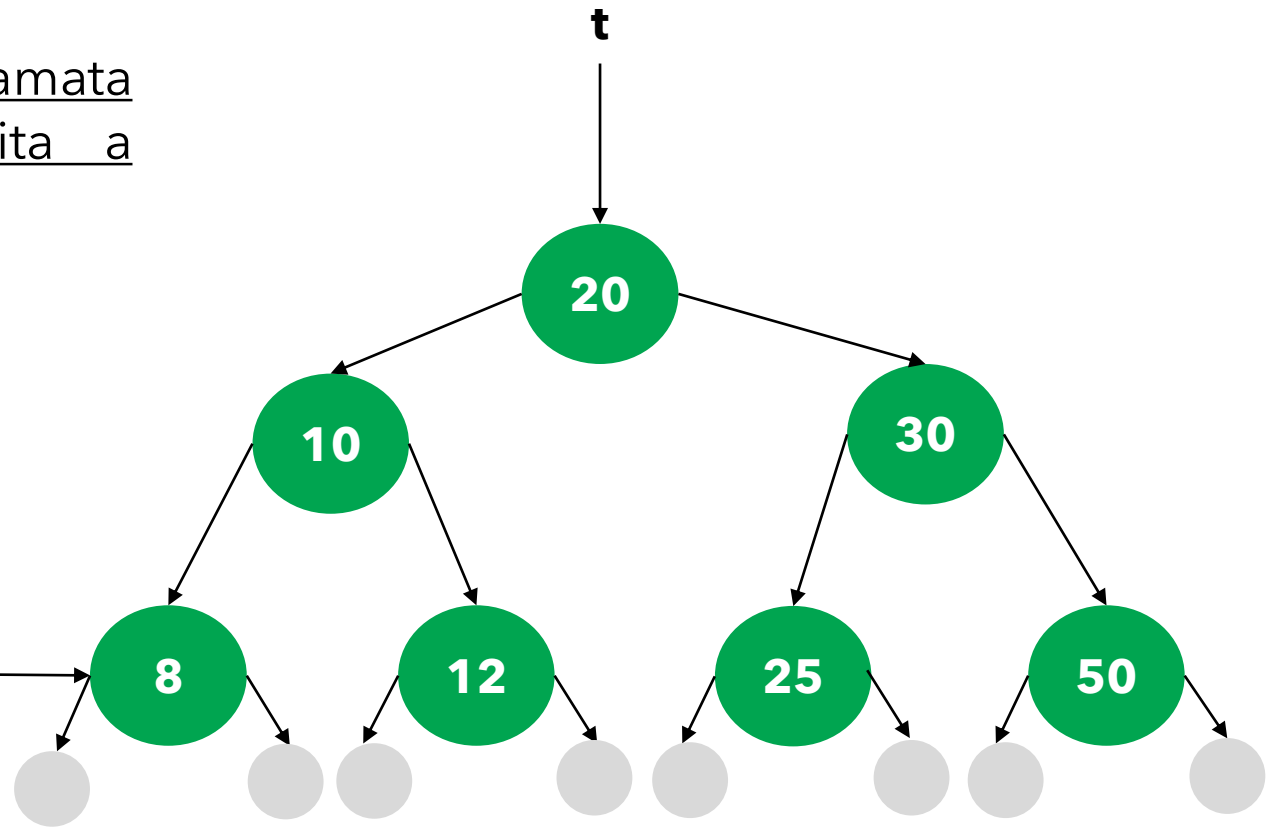
Esempio di *DFS*: visita `inorder` di un *BST*



Esempio di *DFS*: visita `inorder` di un *BST*

l'istruzione di stampa è posta dopo la chiamata ricorsiva sinistra, quindi viene eseguita a questo punto!

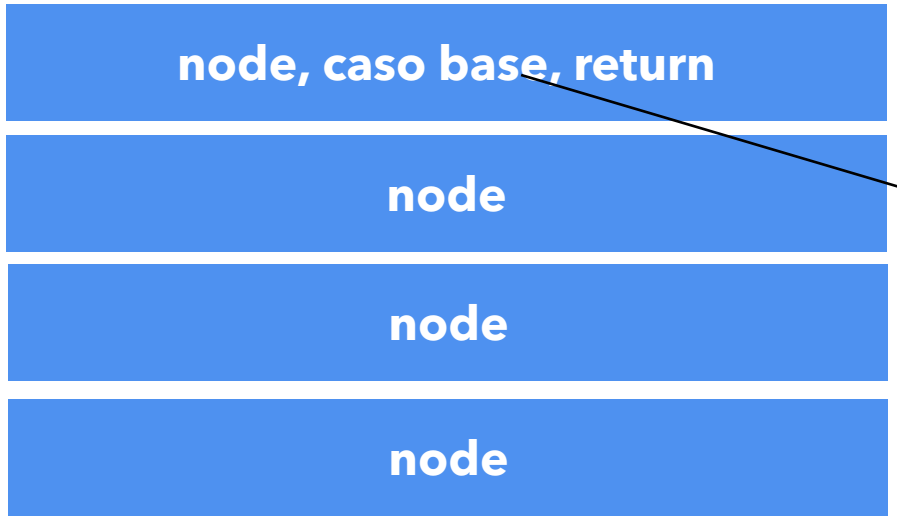


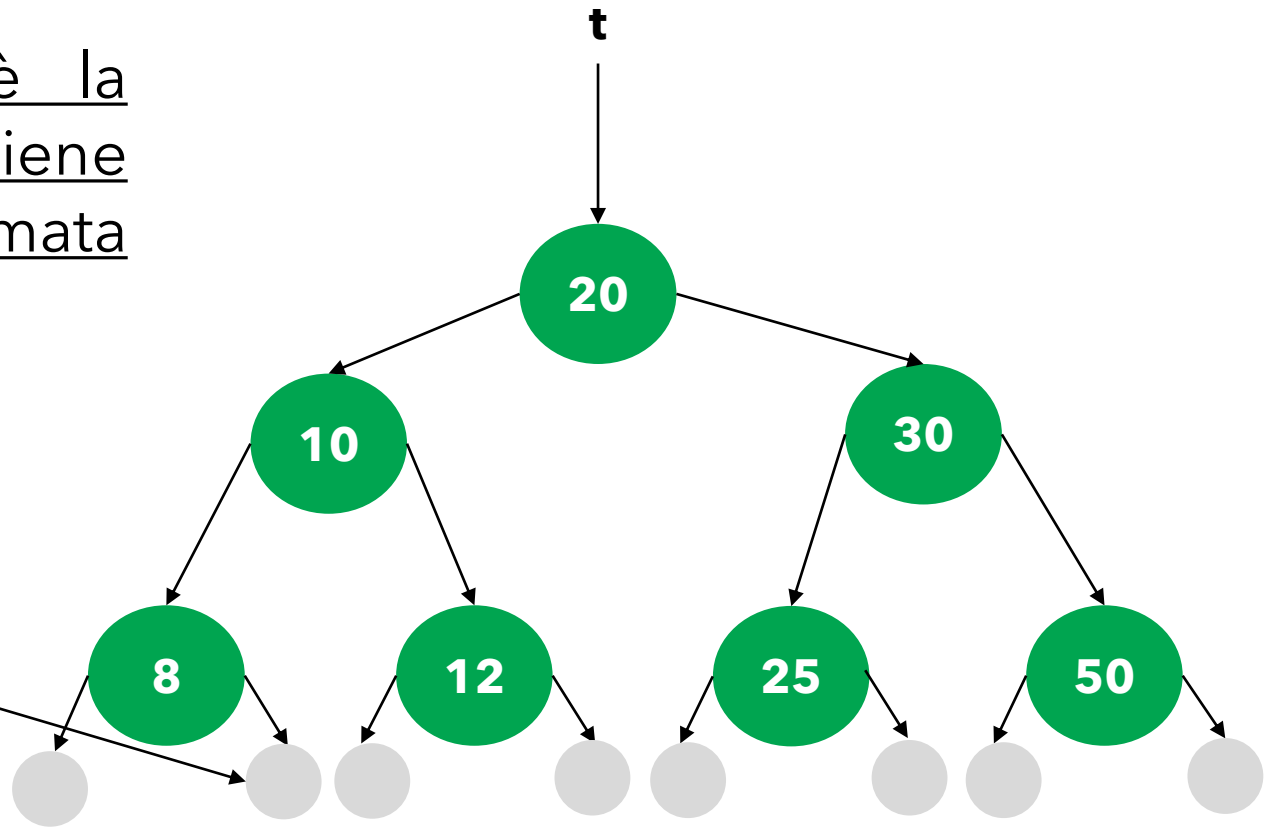


`prints 8`

Esempio di *DFS*: visita `inorder` di un *BST*

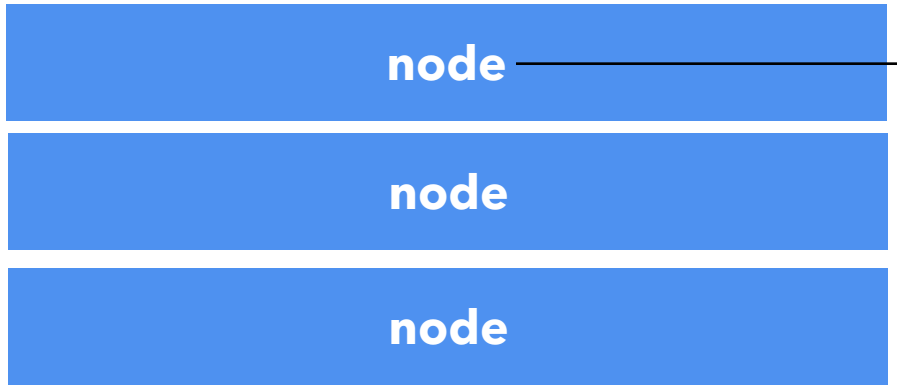
dopo l'istruzione di stampa c'è la chiamata ricorsiva destra, quindi viene eseguita dopo il return della chiamata ricorsiva sinistra

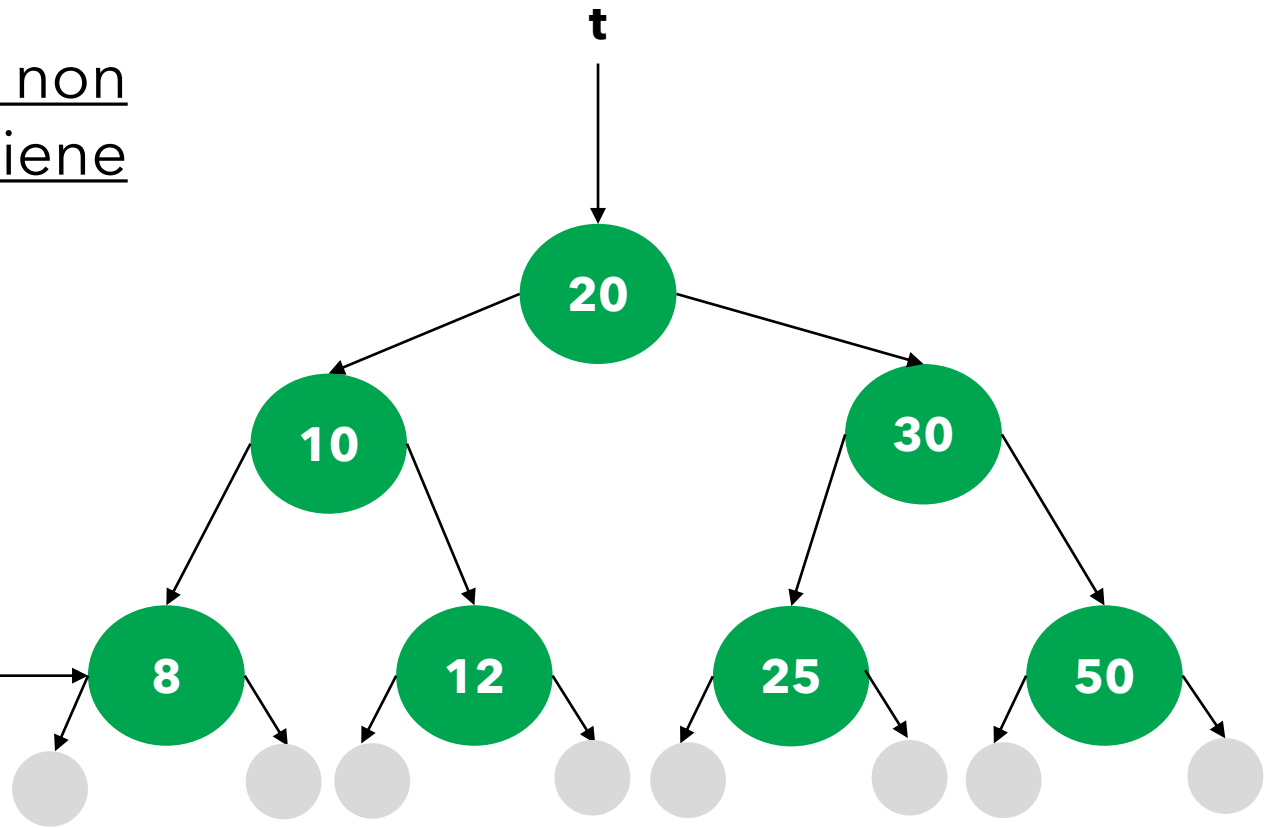




Esempio di *DFS*: visita `inorder` di un *BST*

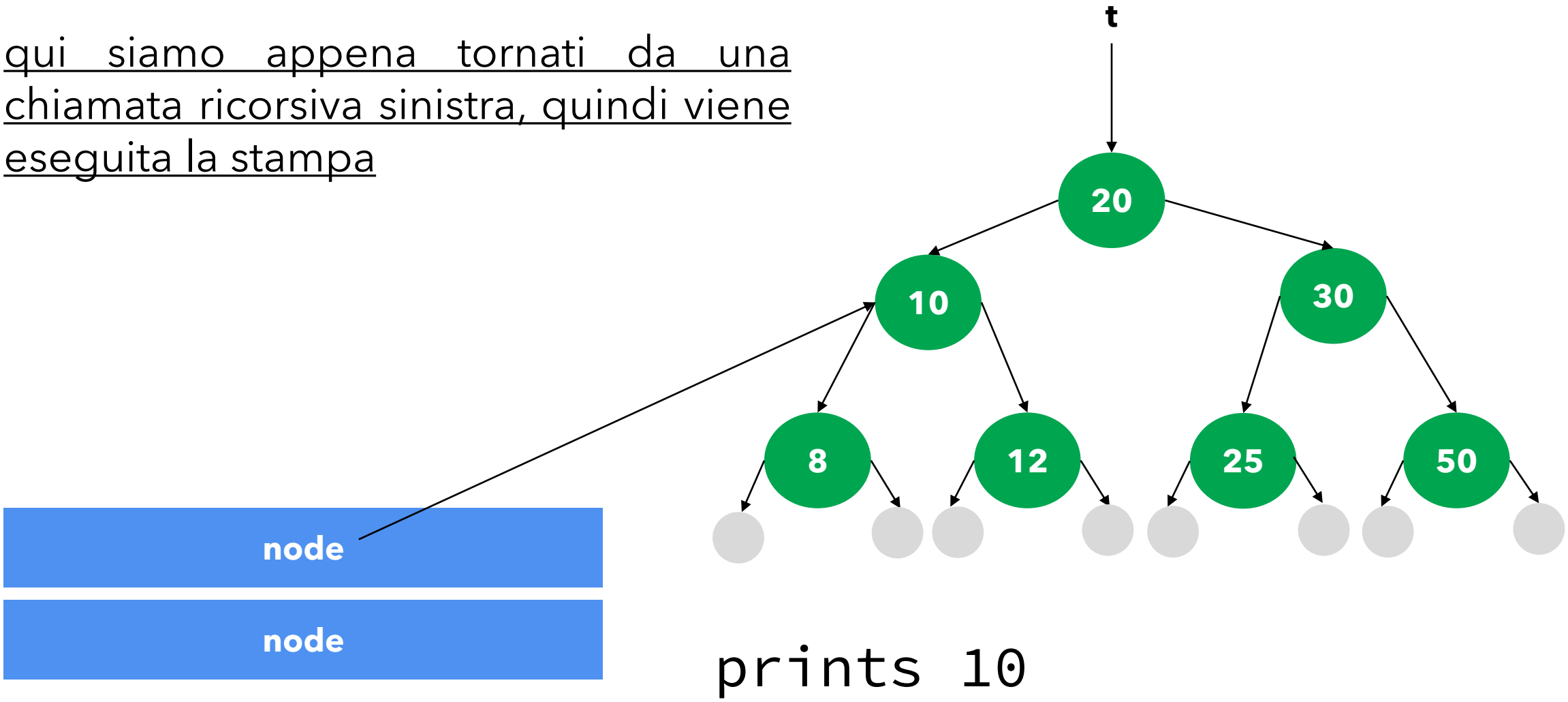
dopo la chiamata ricorsiva destra non c'è niente, quindi il frame viene poppato e non viene fatto altro





Esempio di *DFS*: visita `inorder` di un *BST*

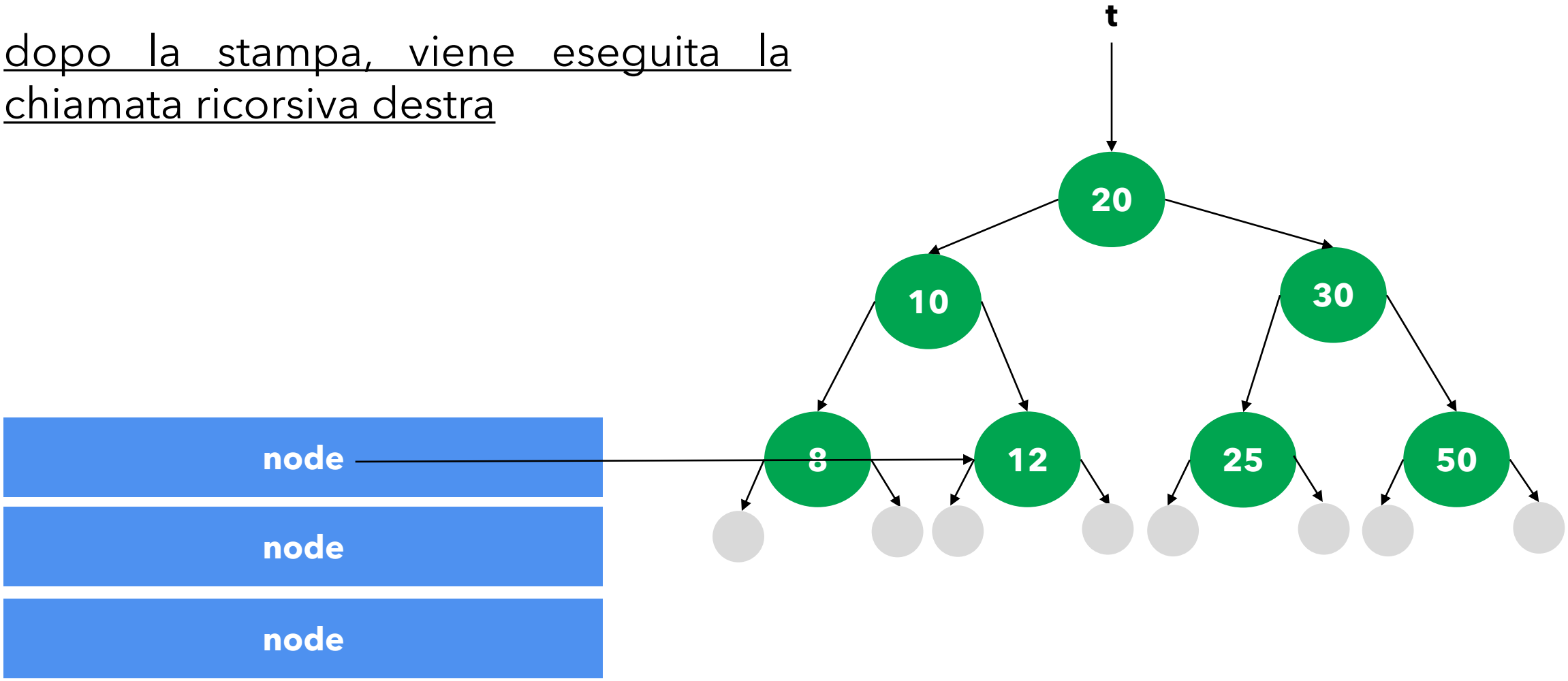
qui siamo appena tornati da una chiamata ricorsiva sinistra, quindi viene eseguita la stampa



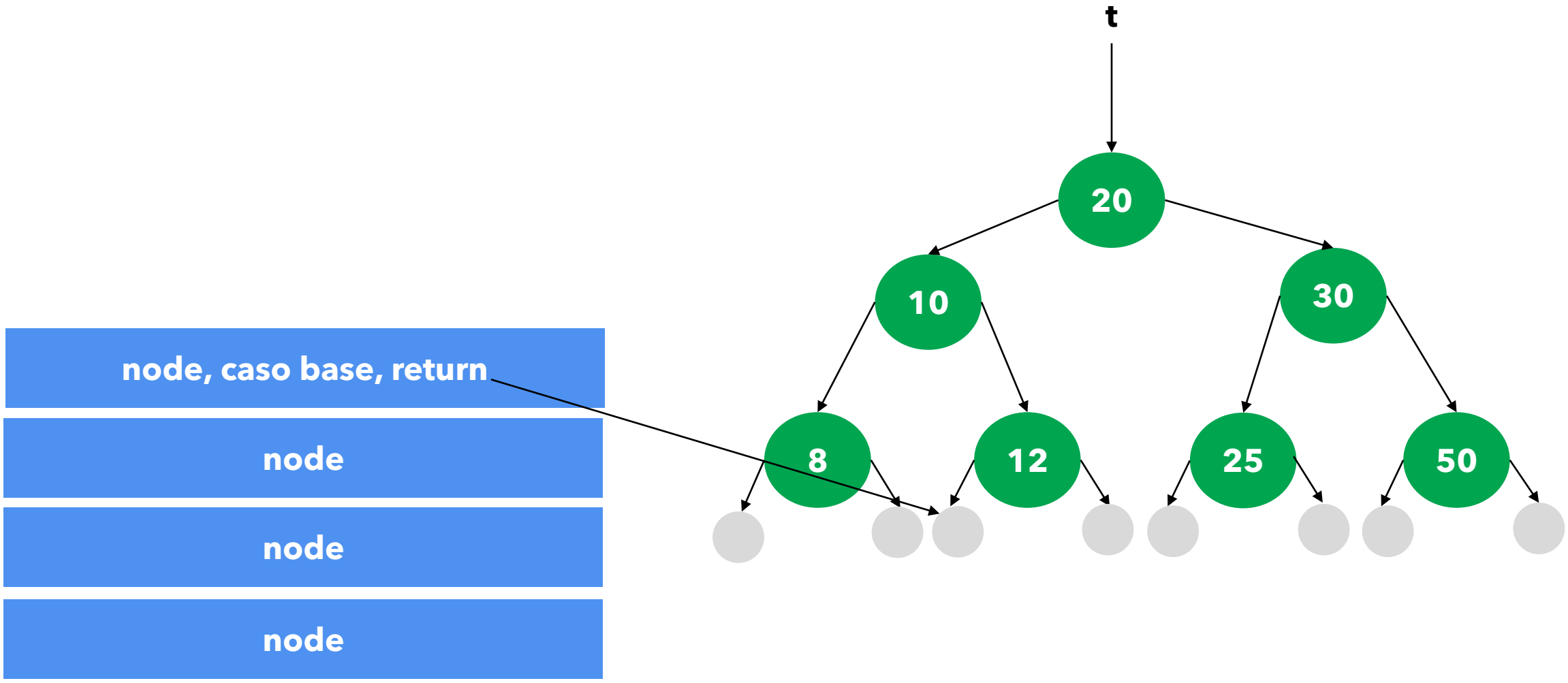
`prints 10`

Esempio di *DFS*: visita `inorder` di un *BST*

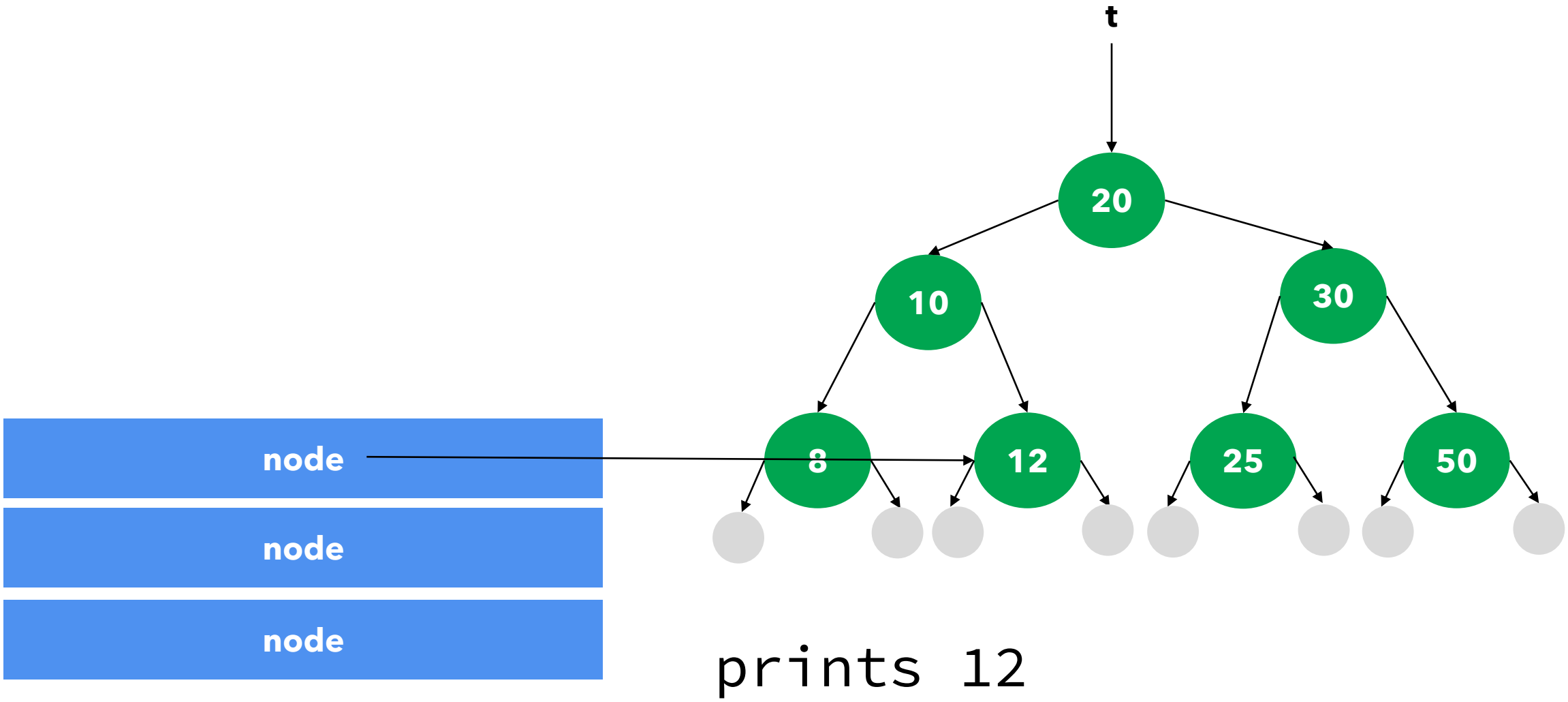
dopo la stampa, viene eseguita la chiamata ricorsiva destra



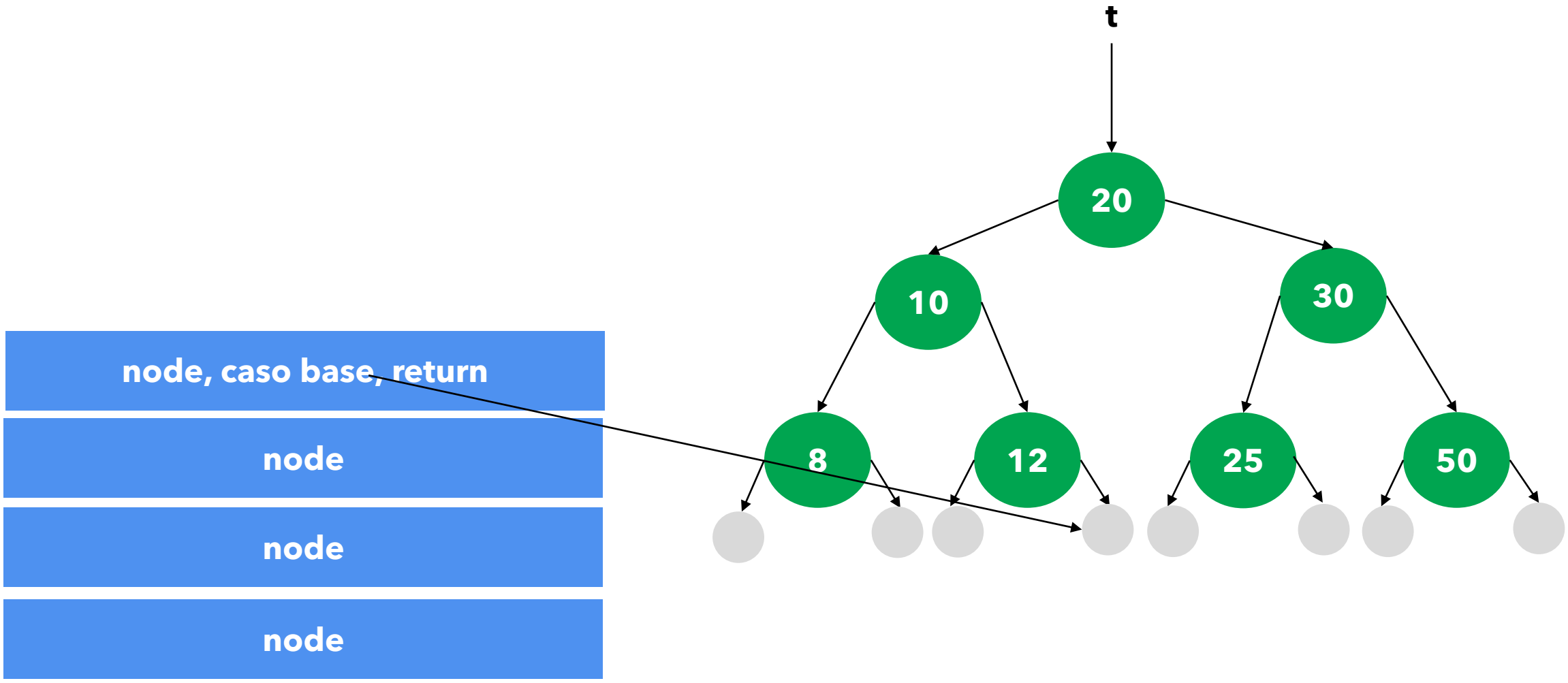
Esempio di *DFS*: visita `inorder` di un *BST*



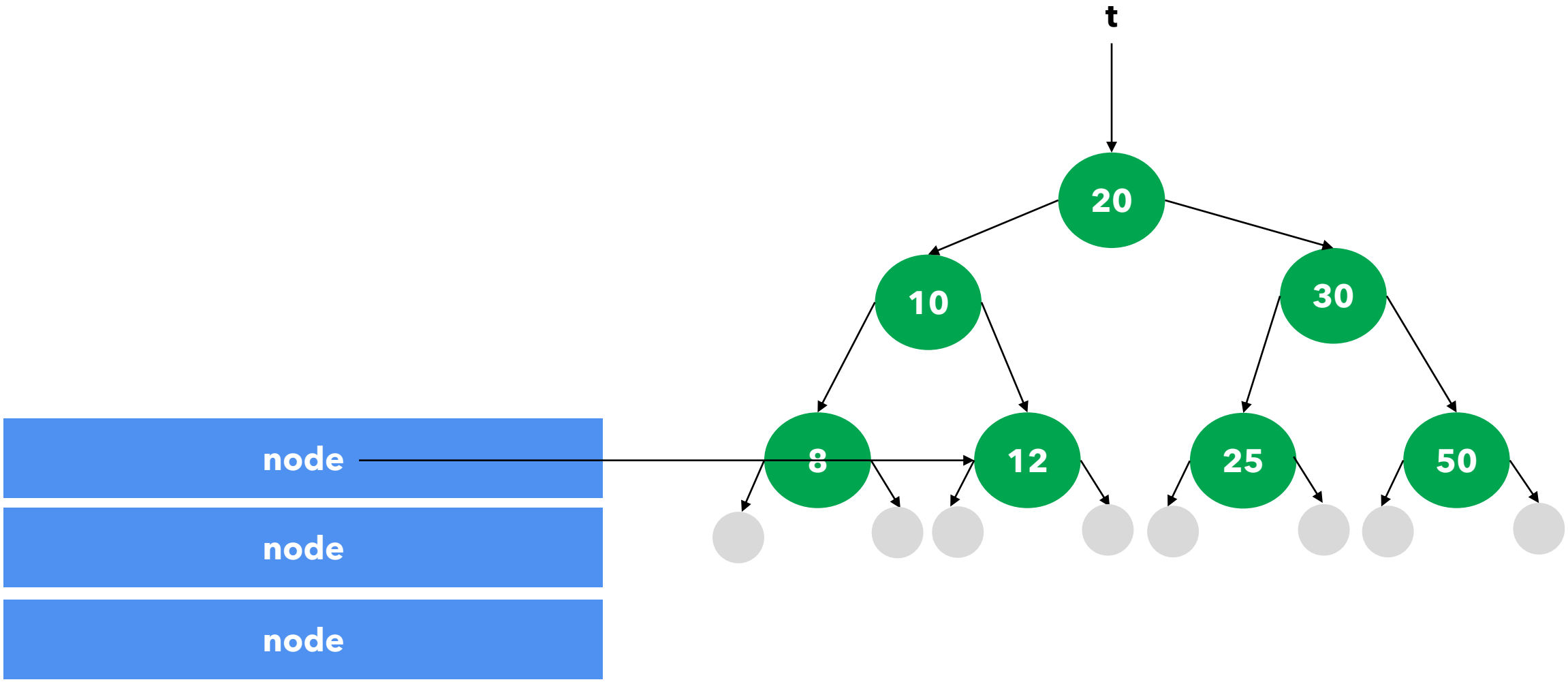
Esempio di *DFS*: visita `inorder` di un *BST*



Esempio di *DFS*: visita `inorder` di un *BST*

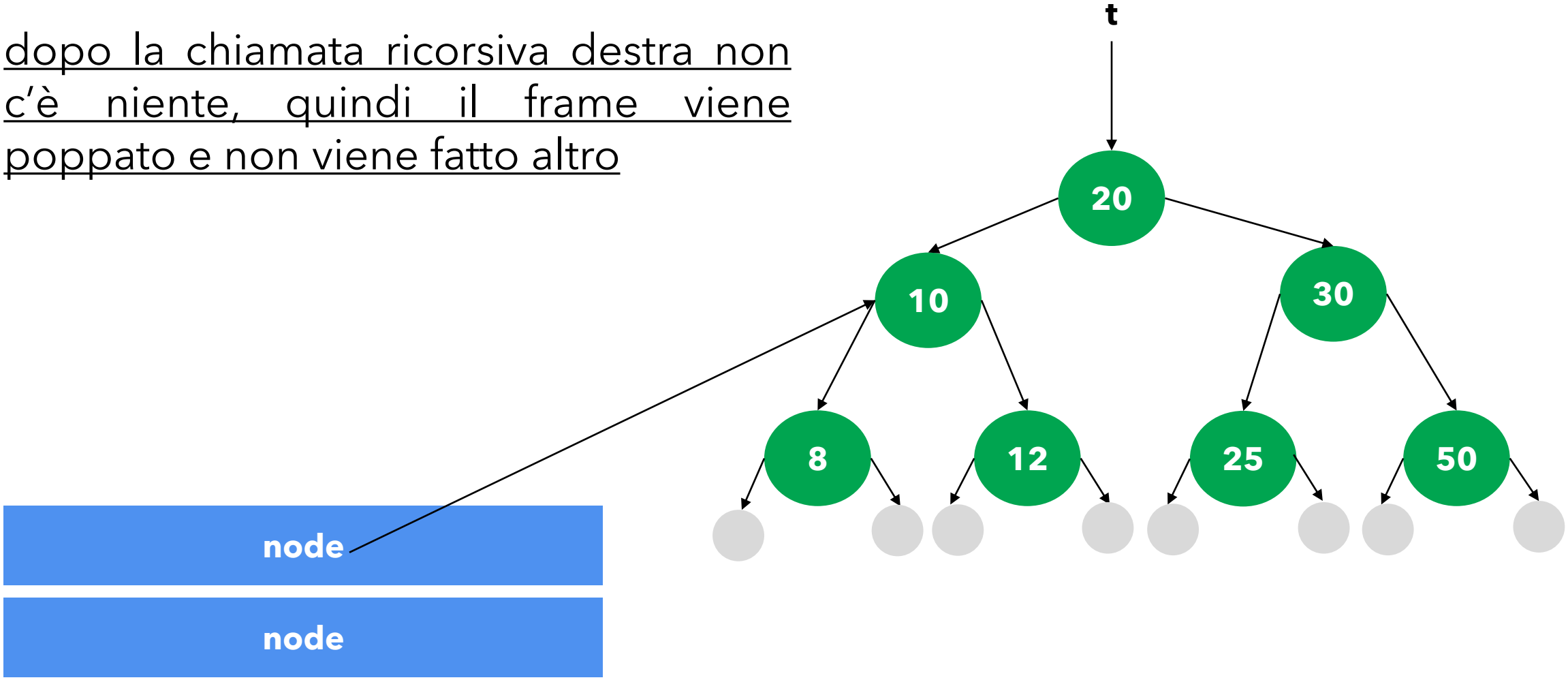


Esempio di *DFS*: visita `inorder` di un *BST*

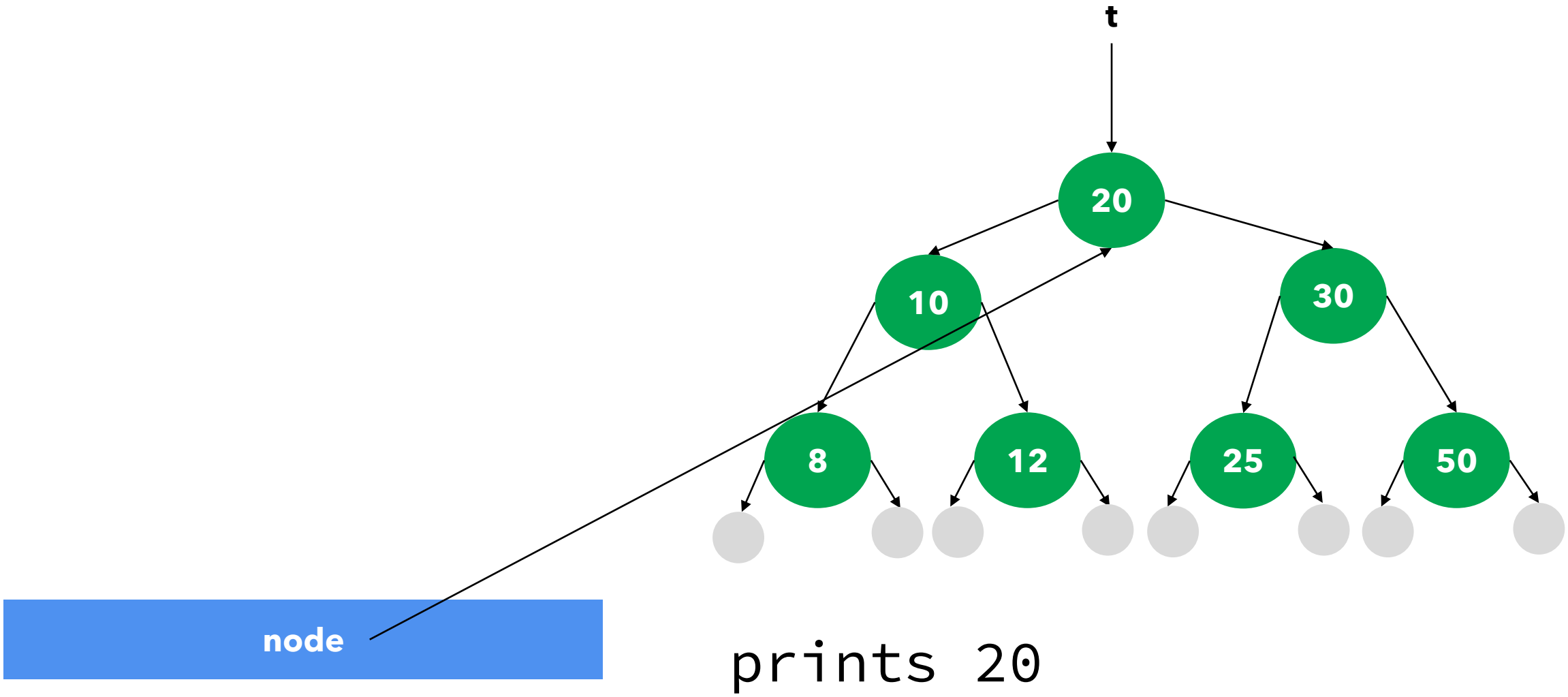


Esempio di *DFS*: visita `inorder` di un *BST*

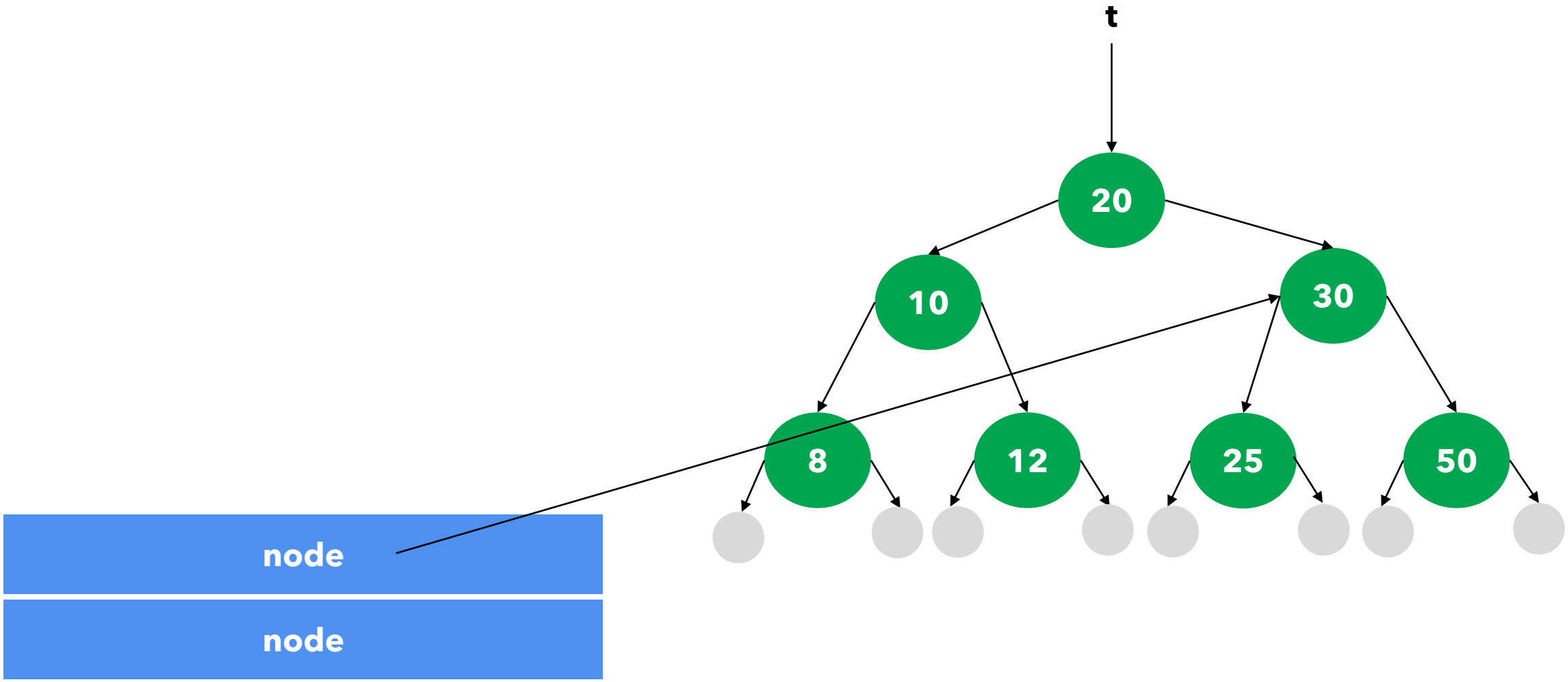
dopo la chiamata ricorsiva destra non c'è niente, quindi il frame viene poppato e non viene fatto altro



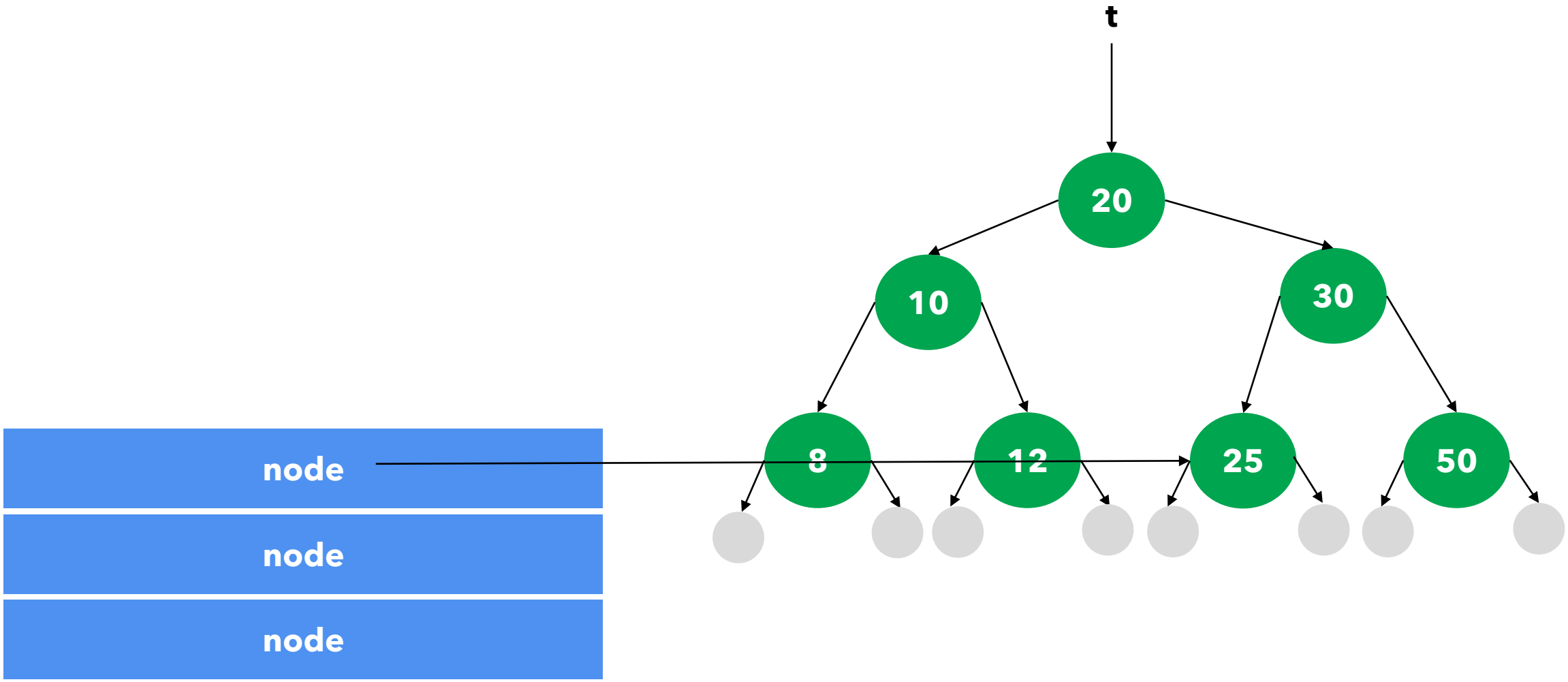
Esempio di *DFS*: visita `inorder` di un *BST*



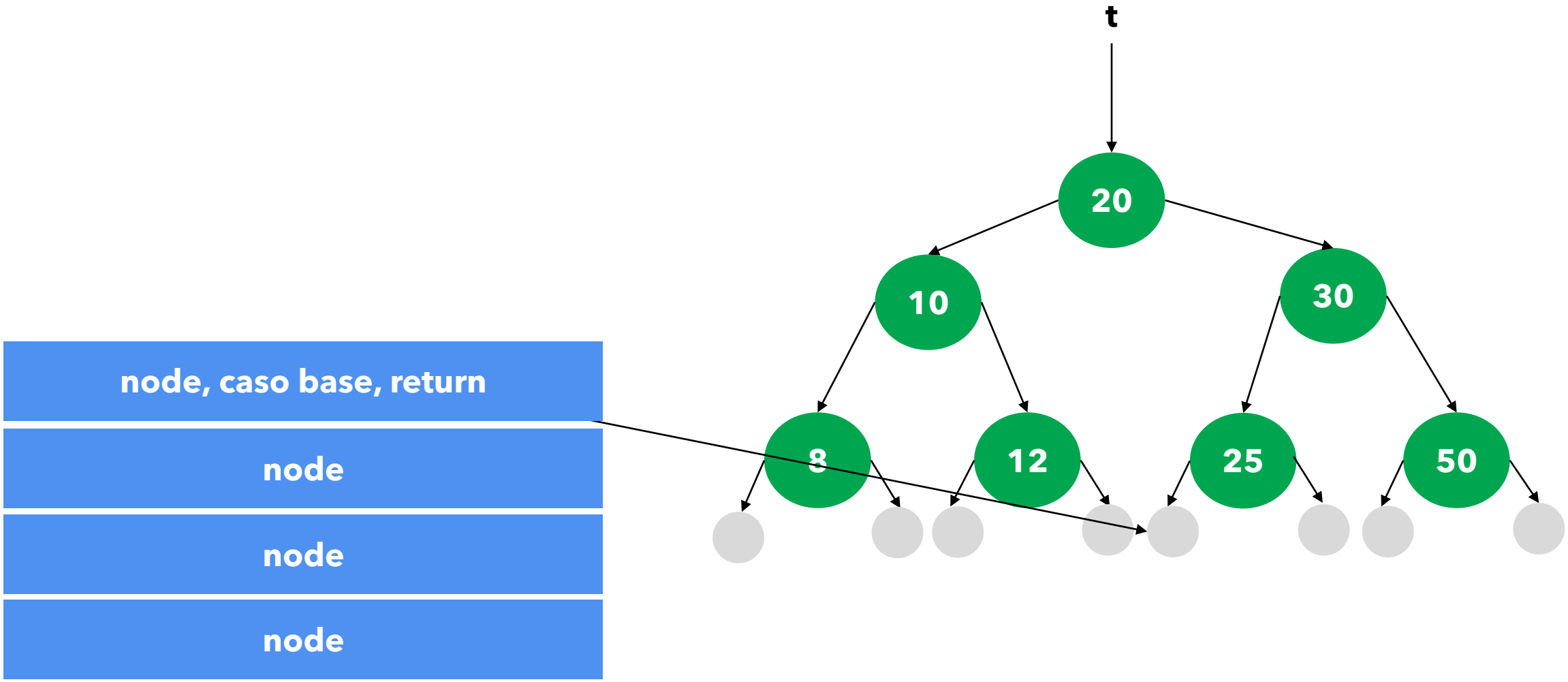
Esempio di *DFS*: visita `inorder` di un *BST*



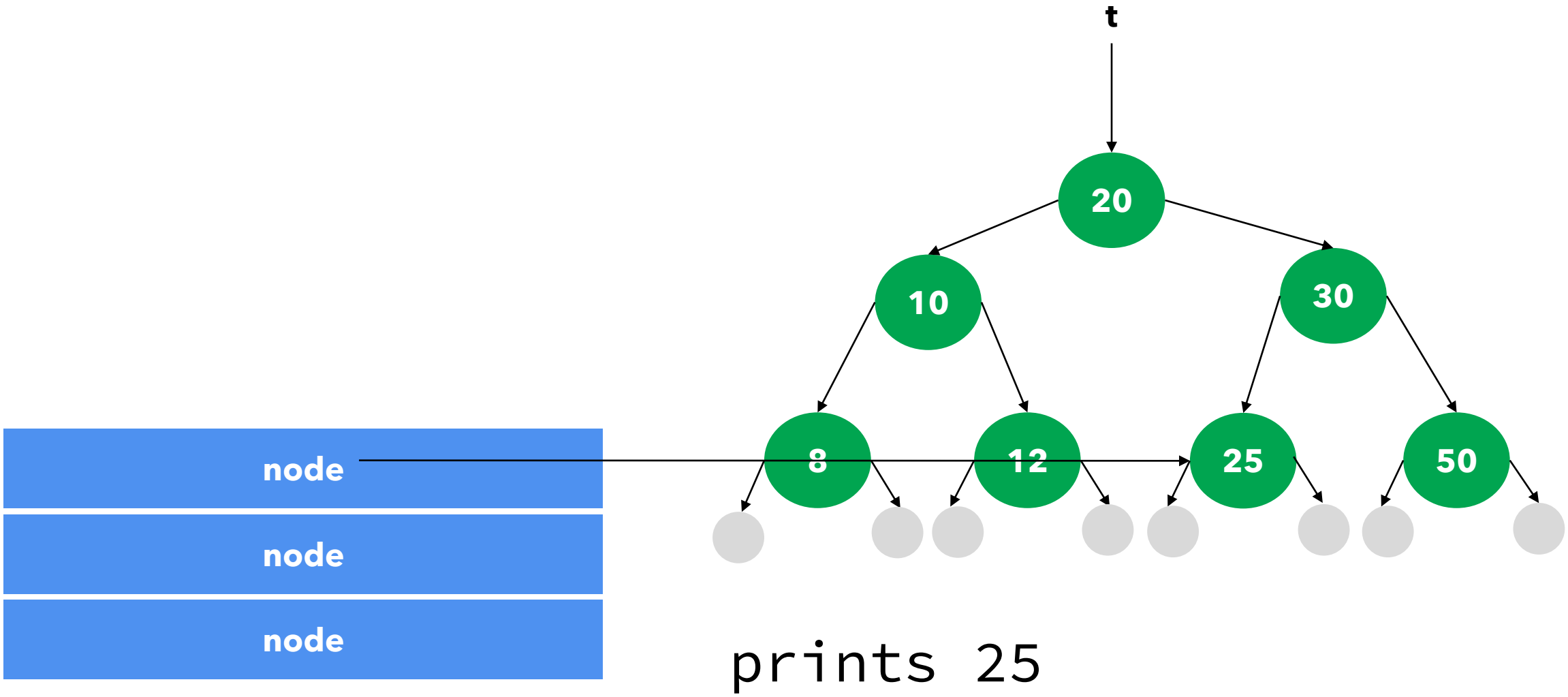
Esempio di *DFS*: visita `inorder` di un *BST*



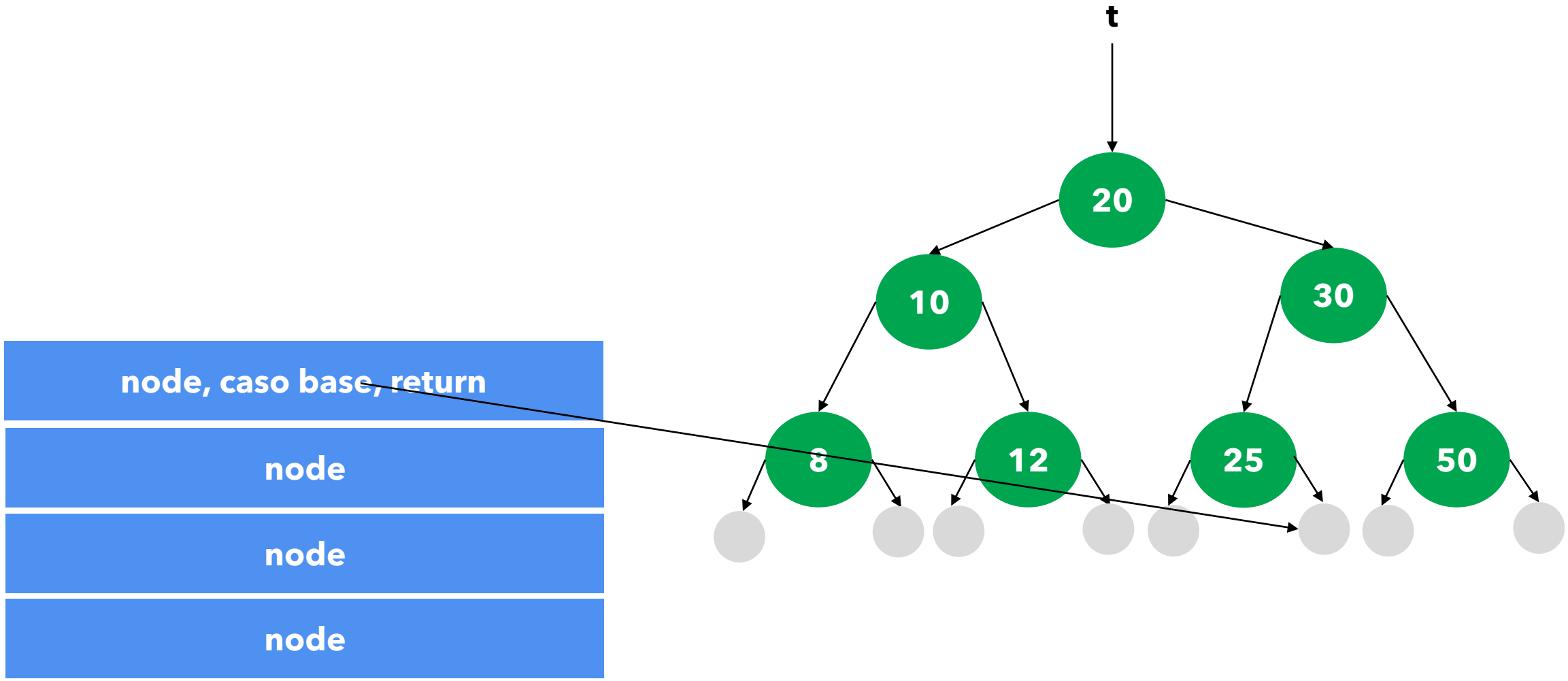
Esempio di *DFS*: visita `inorder` di un *BST*



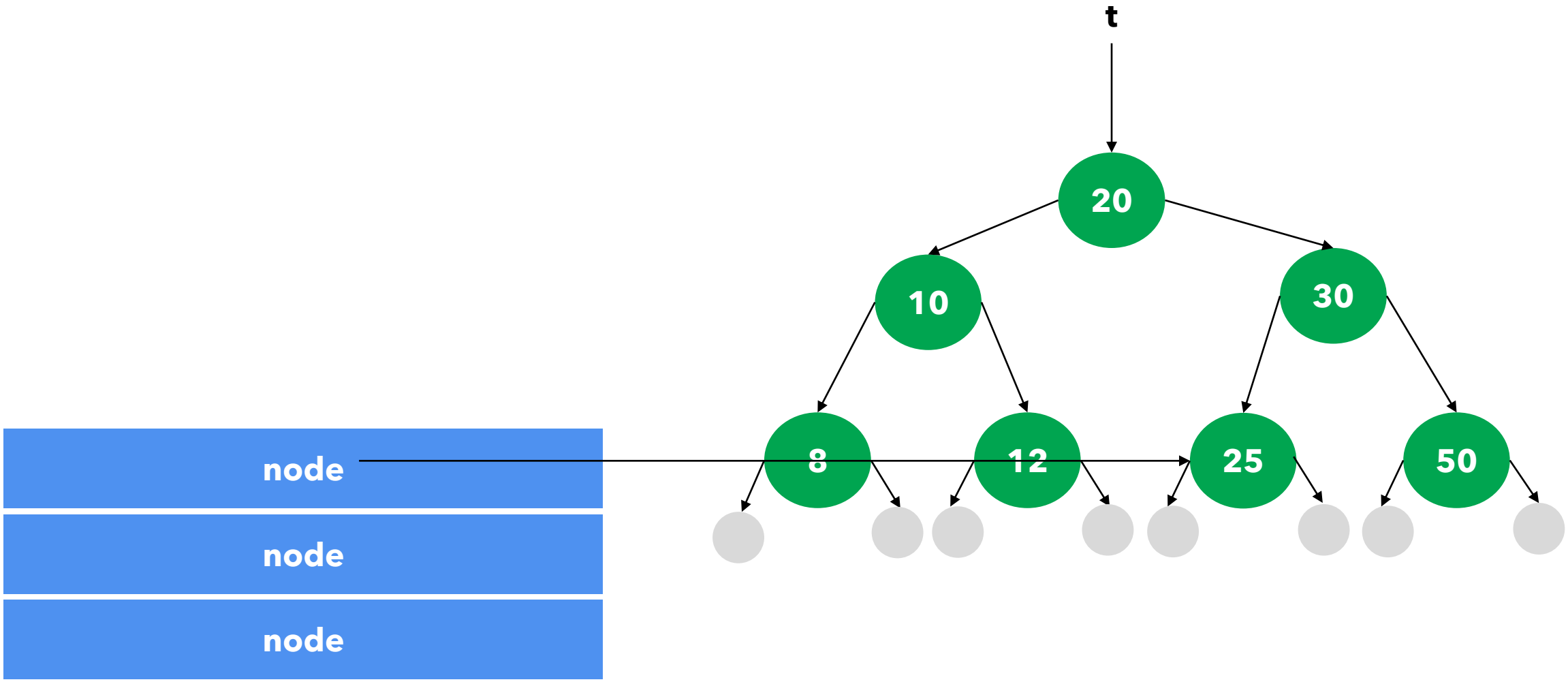
Esempio di *DFS*: visita `inorder` di un *BST*



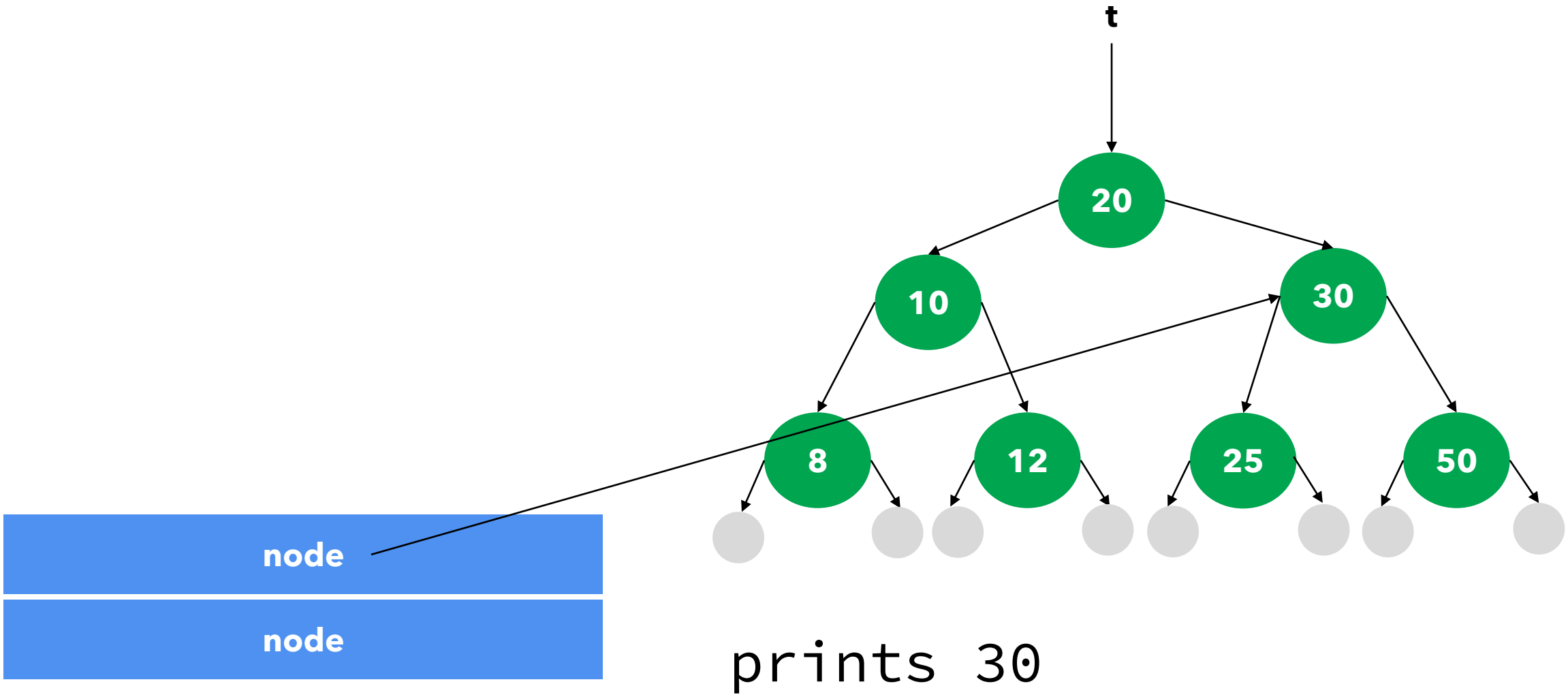
Esempio di *DFS*: visita `inorder` di un *BST*



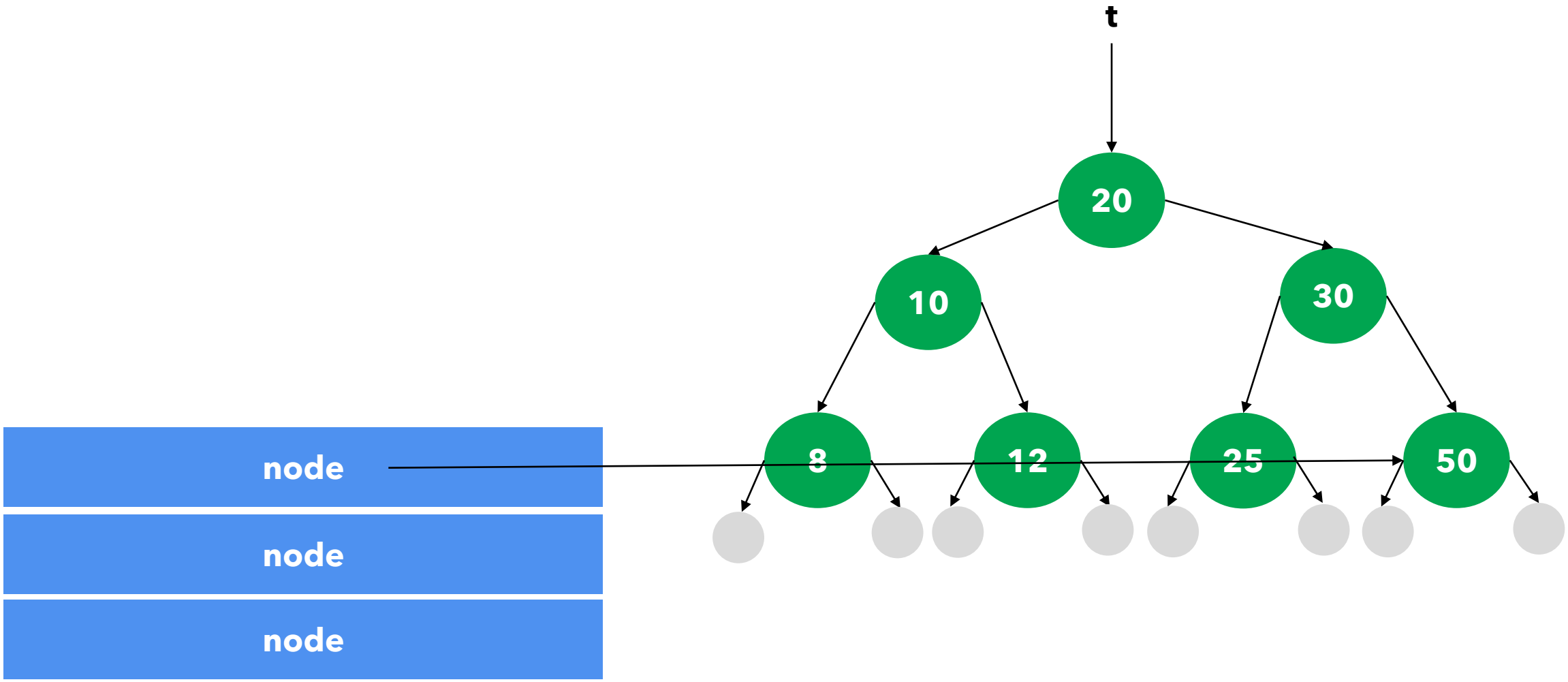
Esempio di *DFS*: visita `inorder` di un *BST*



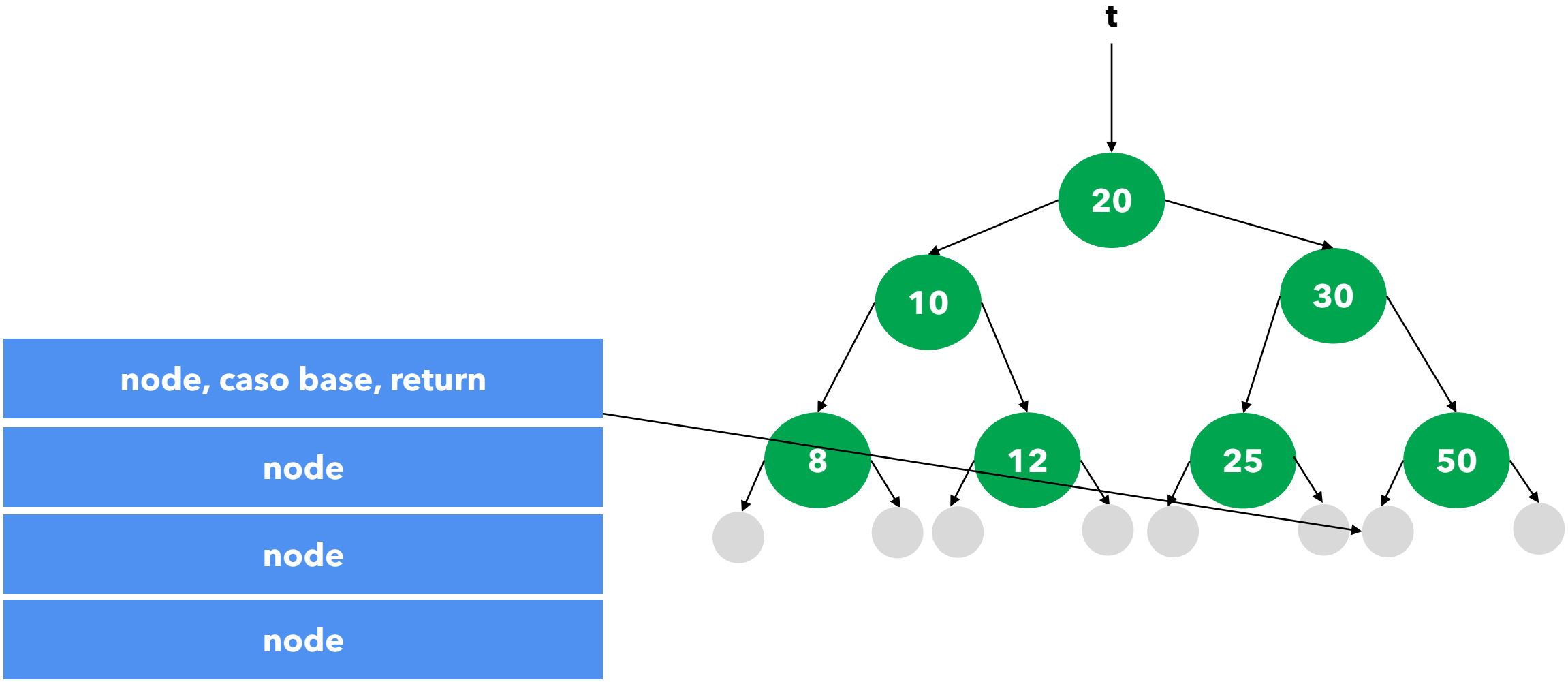
Esempio di *DFS*: visita `inorder` di un *BST*



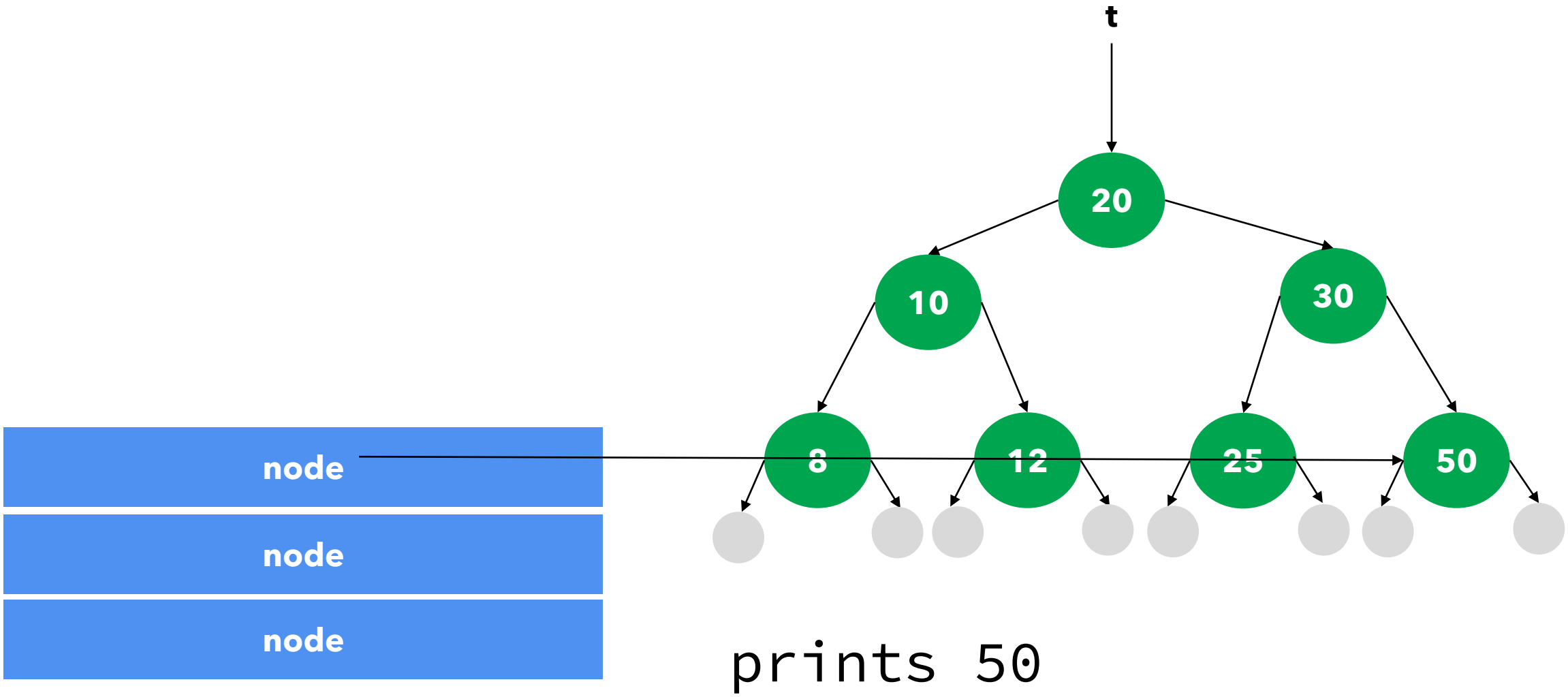
Esempio di *DFS*: visita `inorder` di un *BST*



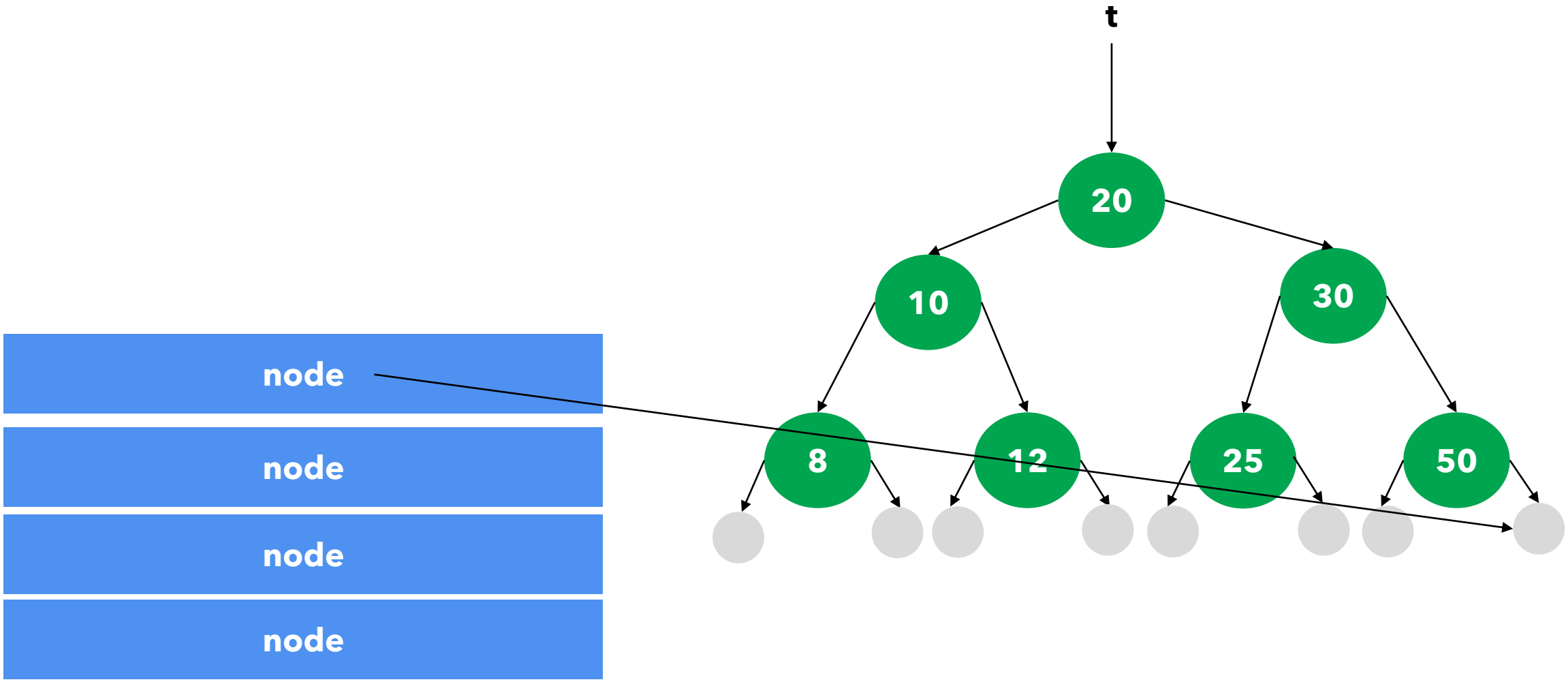
Esempio di *DFS*: visita `inorder` di un *BST*



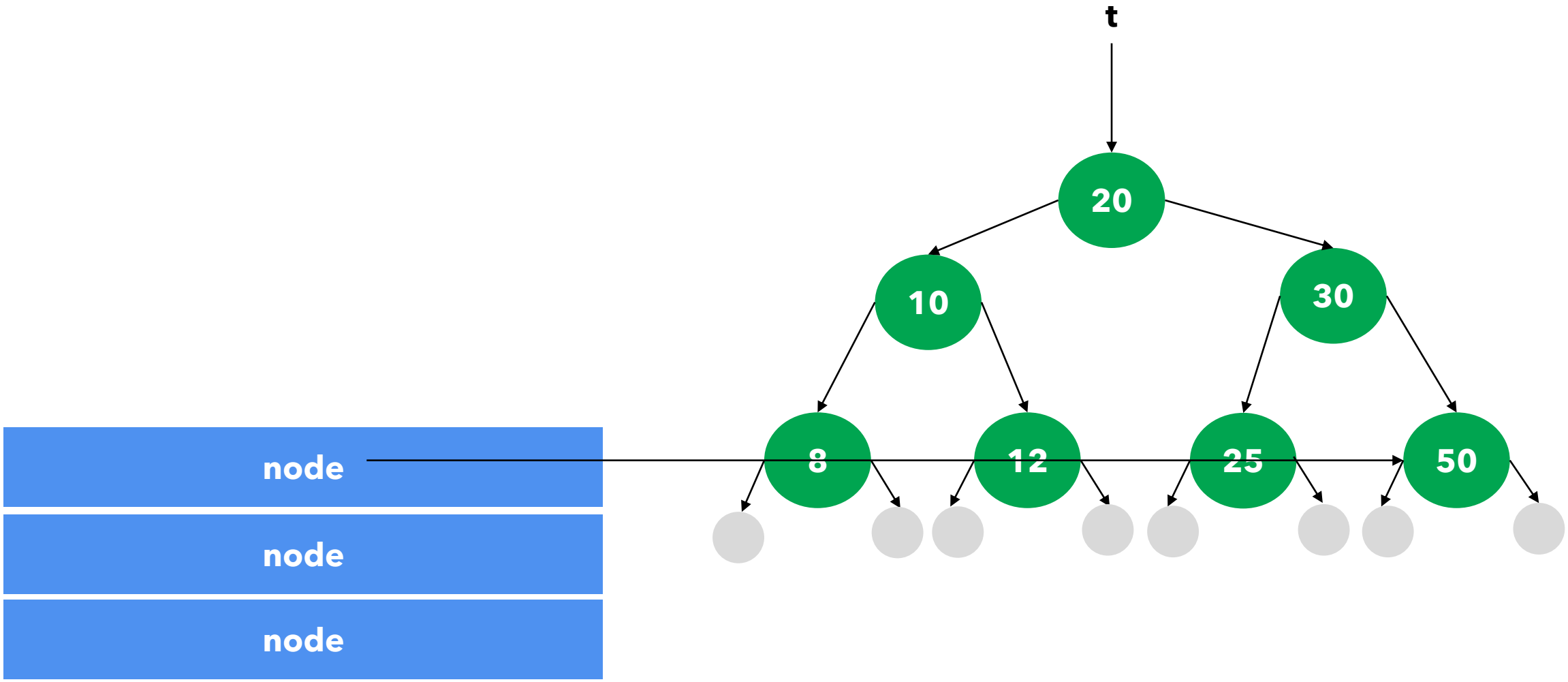
Esempio di *DFS*: visita `inorder` di un *BST*



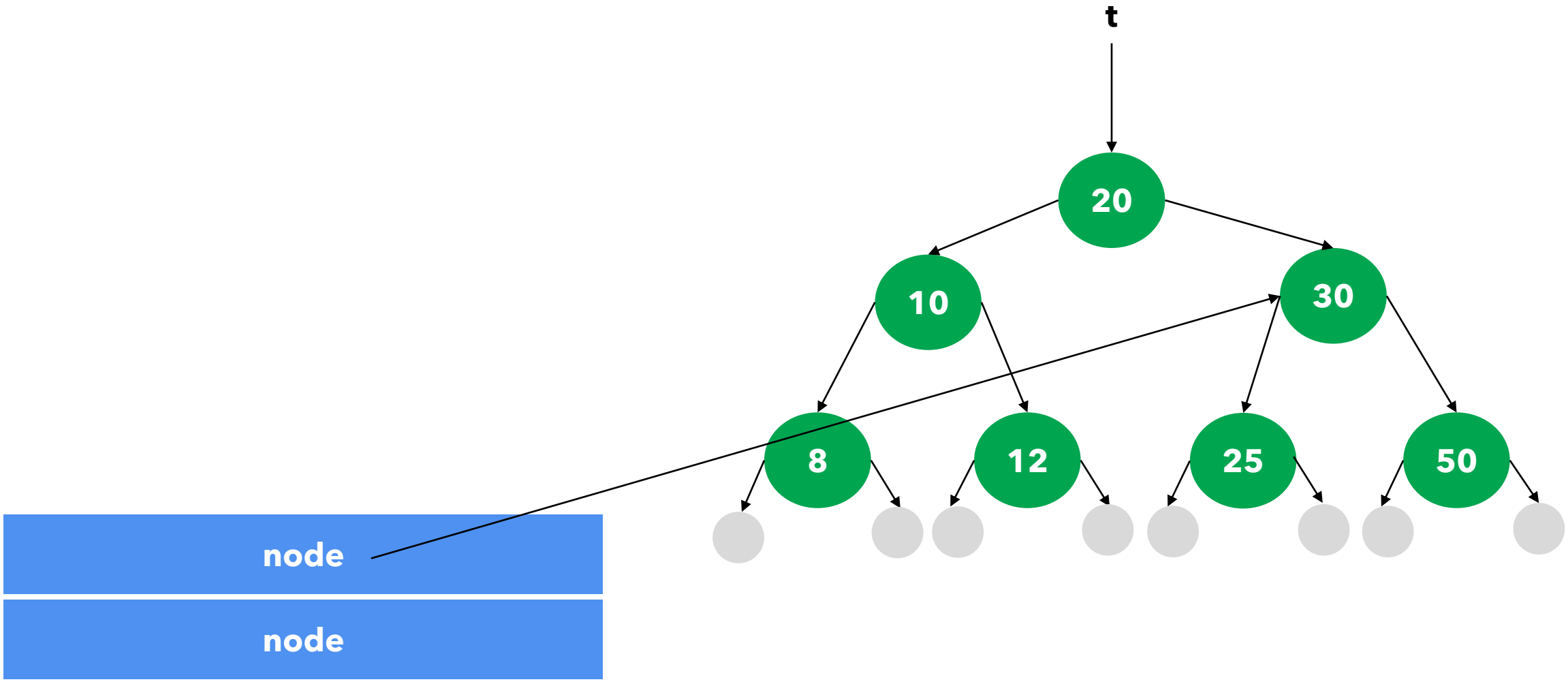
Esempio di *DFS*: visita `inorder` di un *BST*



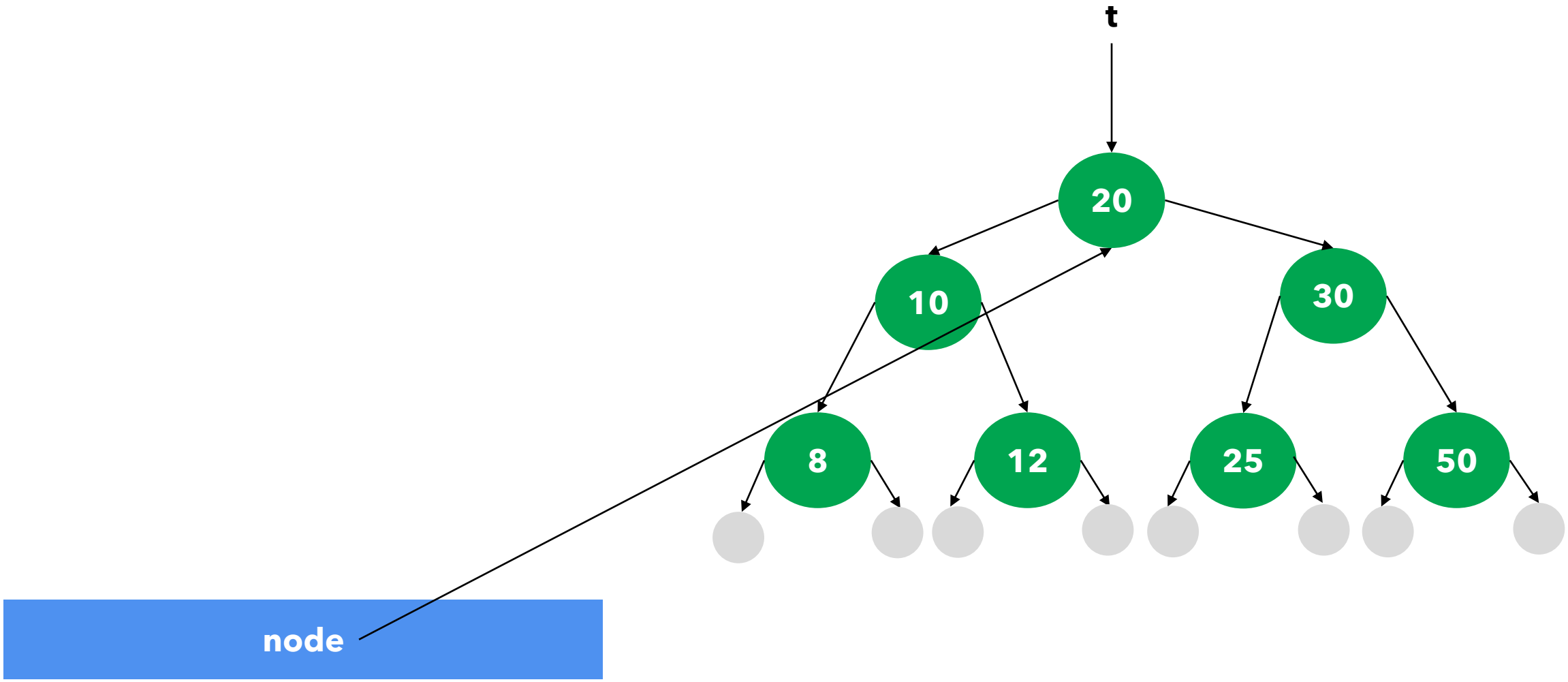
Esempio di *DFS*: visita `inorder` di un *BST*



Esempio di *DFS*: visita `inorder` di un *BST*



Esempio di *DFS*: visita `inorder` di un *BST*



Esempio di *DFS*: visita preorder

preorder_tree_walk(T):

- se l'albero T è vuoto, return
- altrimenti
  - 'apri' il nodo T (ad esempio: stampa T.key, o in generale *esegui un'operazione su T*)
  - esegui preorder_tree_walk su T.left
  - esegui preorder_tree_walk su T.right

Esempi di *DFS*: visite `preorder` e `postorder`

```
void pre_order(TREE_NODE *node) {
  if (!node) {
    return;
  }
  printf("%d ", node->key);
  pre_order(node->left);
  pre_order(node->right);
}
```

```
void post_order(TREE_NODE *node) {
  if (!node) {
    return;
  }
  post_order(node->left);
  post_order(node->right);
  printf("%d ", node->key);
}
```

Breadth-first search (*BFS*) su un albero binario

- La ricerca in larghezza/ampiezza è una ricerca per livelli
- Prima si visitano tutti i nodi al livello 0, poi tutti quelli al livello 1, poi tutti quelli al livello 2 etc… Per comodità, visitiamo ogni livello da sinistra a destra
- Per tenere traccia dei livelli nell'ordine corretto serve una **queue** (coda)
- La *BFS* diventerà chiarissima con dei disegni

Breadth-first search (*BFS*) su un albero binario

```
BFS(Tree):
    Queue = EMPTY
    Enqueue(Queue, Tree)
    while Queue is not empty:
        if Queue.Head.TreeNode.leftchild is not nil:
            Enqueue(Queue, Tree.leftchild)
        if Queue.Head.TreeNode.rightchild is not nil:
            Enqueue(Queue, Tree.rightchild)
        dequeue(Queue)
```

In italiano: incoda la radice dell'albero. Poi, fintantoché la coda non è vuota, incoda i 2 figli della testa alla coda (se non sono nulli), poi, rimuovi la testa della coda

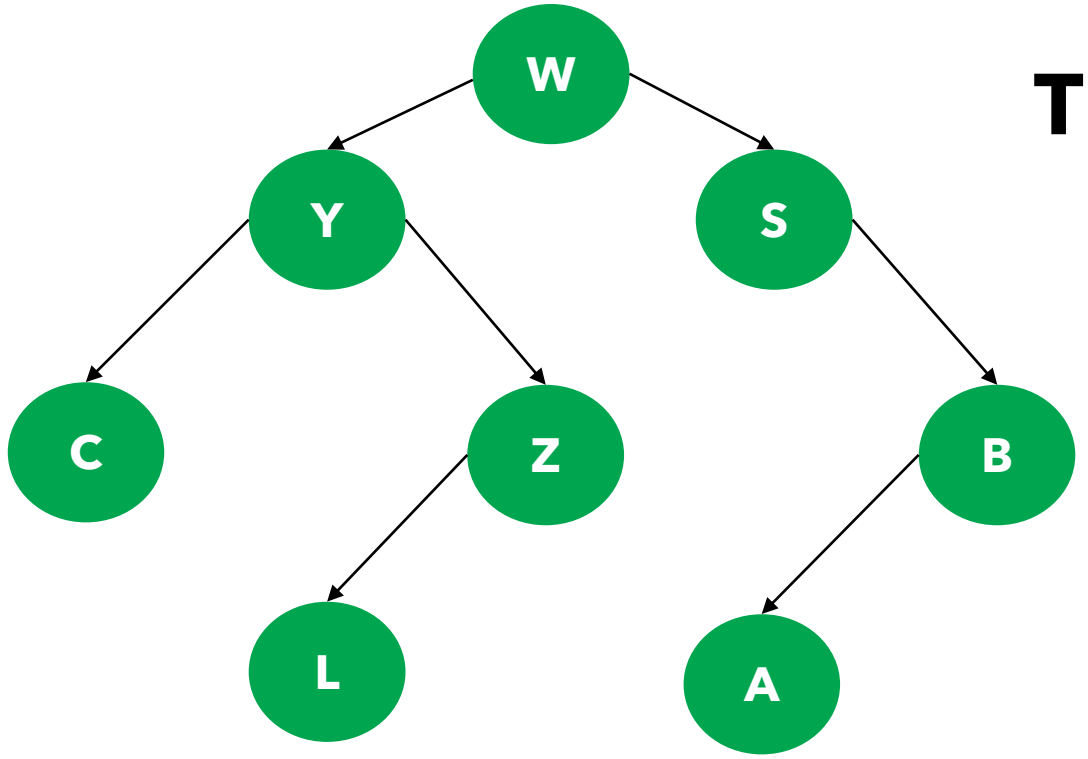
Breadth-first search (*BFS*) su un albero binario

livello 0

livello 1

livello 2

livello 3



TREE *T*

QUEUE Q

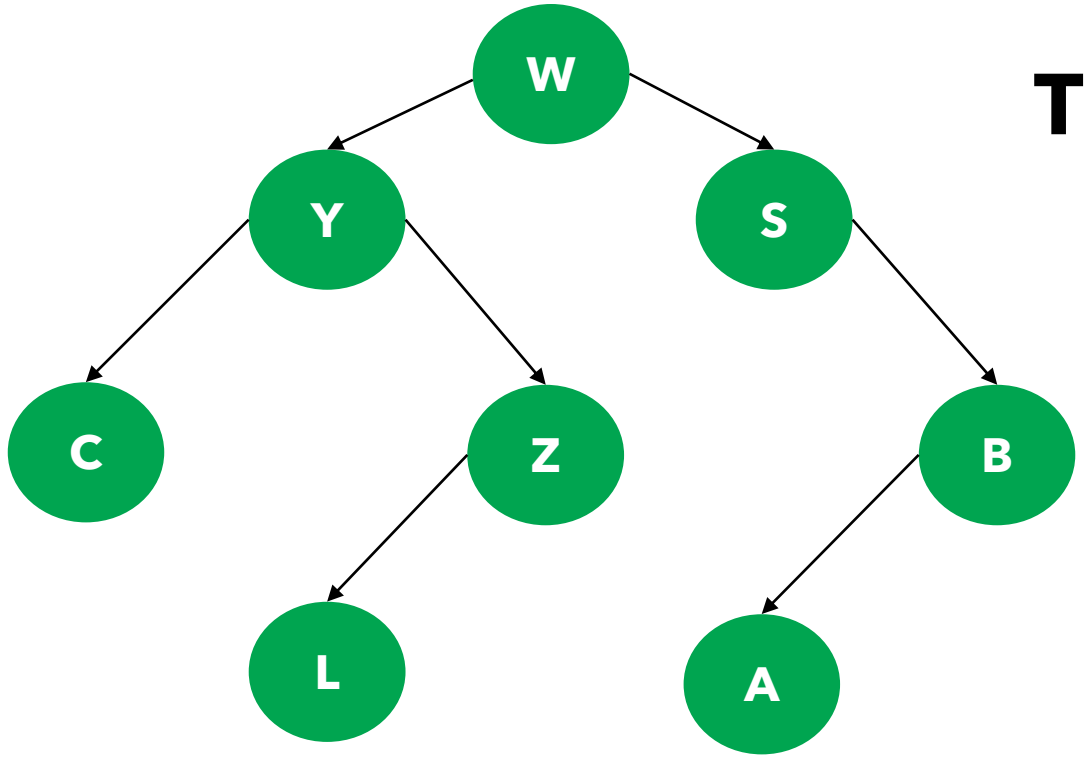
Breadth-first search (BFS) su un albero binario

livello 0

livello 1

livello 2

livello 3

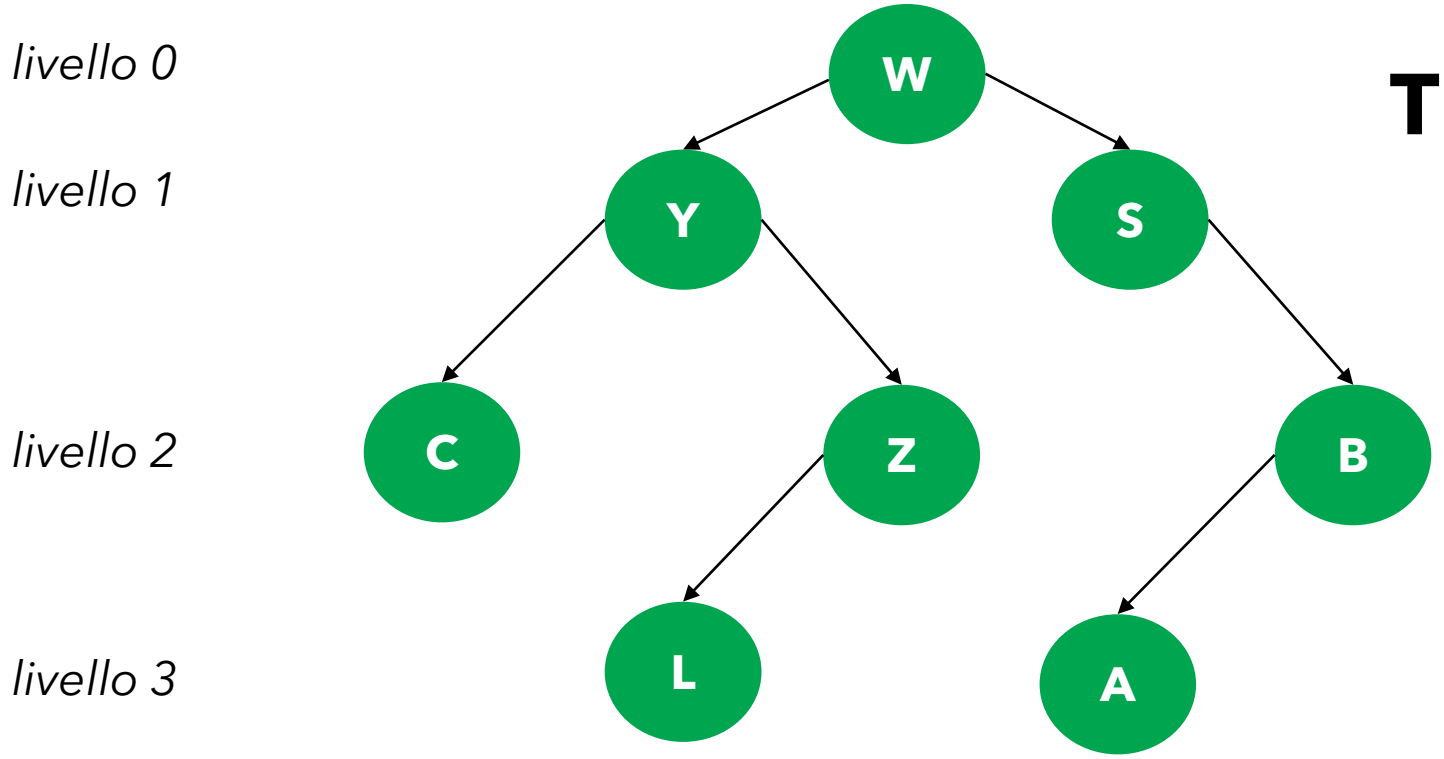


TREE *T*

QUEUE Q

```
enqueue(W);
```

Breadth-first search (BFS) su un albero binario



TREE *T*

QUEUE Q

```
enqueue(W.left); enqueue(W.right) -> enqueue(Y); enqueue(S);
```

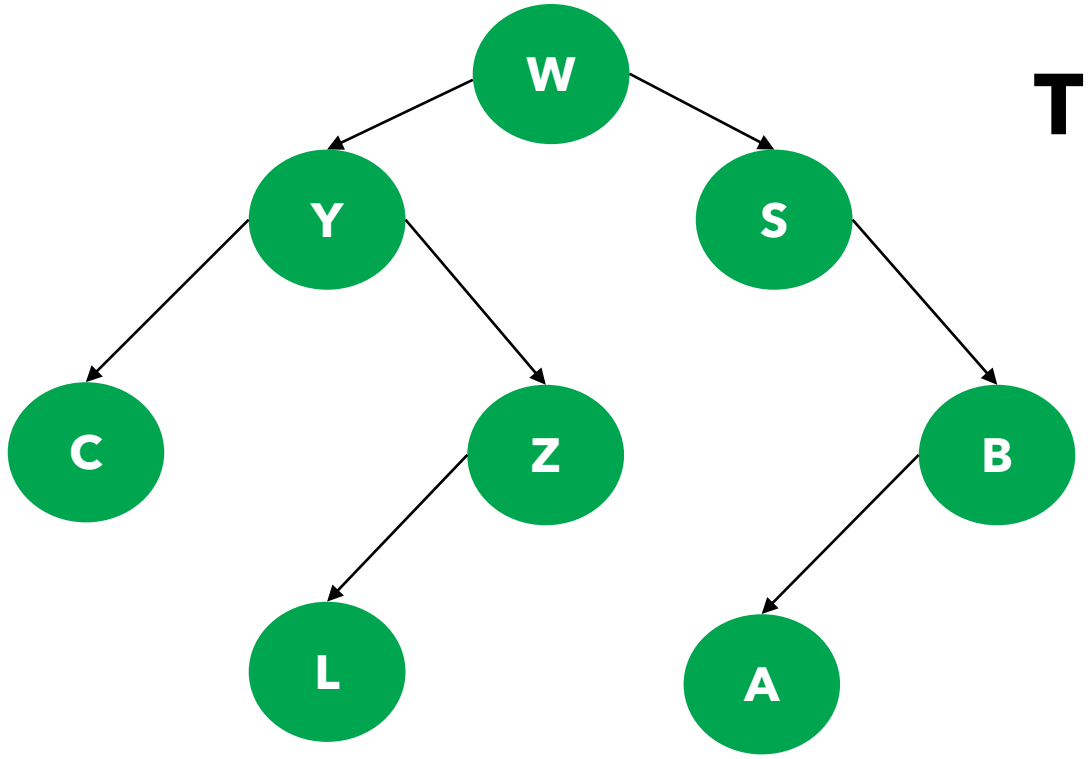
Breadth-first search (BFS) su un albero binario

livello 0

livello 1

livello 2

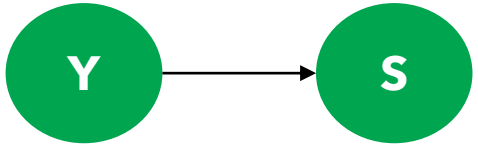
livello 3



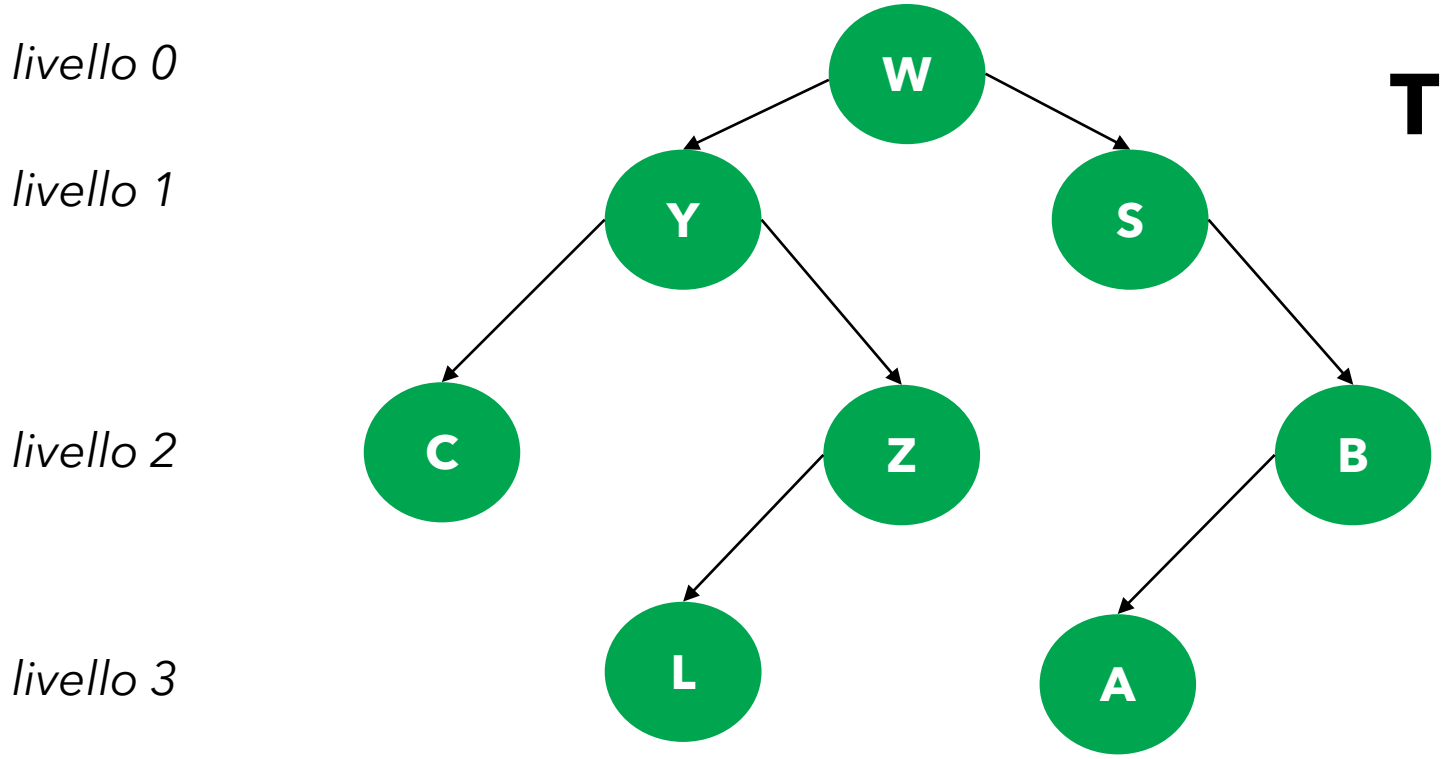
TREE *T*

QUEUE Q

```
dequeue();
```



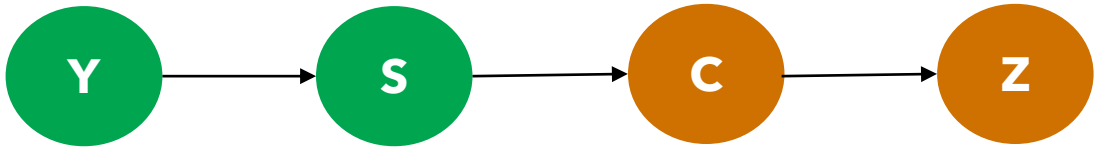
Breadth-first search (BFS) su un albero binario



TREE *T*

QUEUE Q

```
enqueue(Y.left); enqueue(Y.right) -> enqueue(C); enqueue(Z);
```



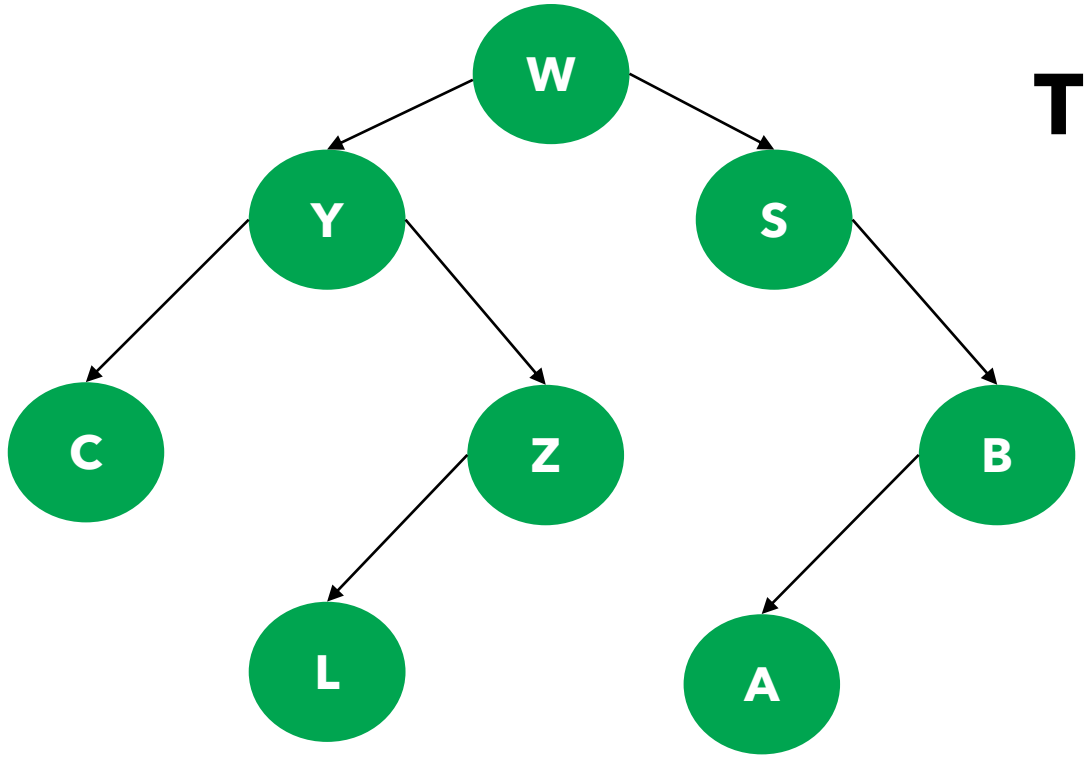
Breadth-first search (BFS) su un albero binario

livello 0

livello 1

livello 2

livello 3



TREE *T*

QUEUE Q

```
dequeue();
```

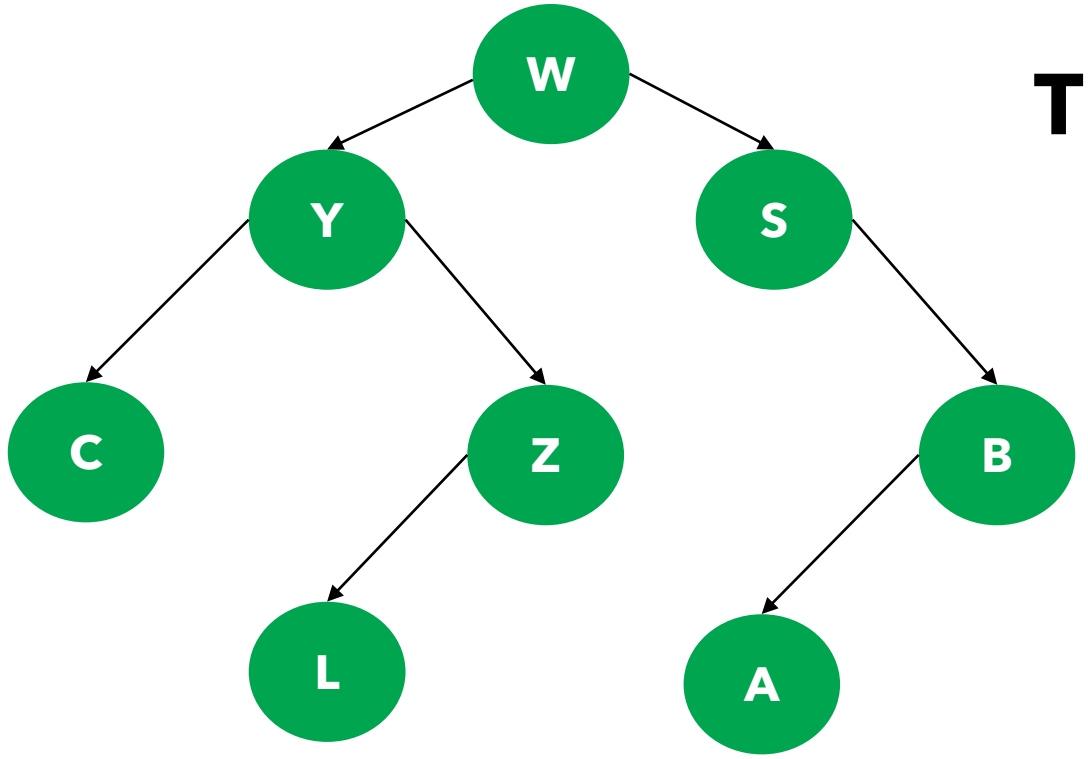
Breadth-first search (BFS) su un albero binario

livello 0

livello 1

livello 2

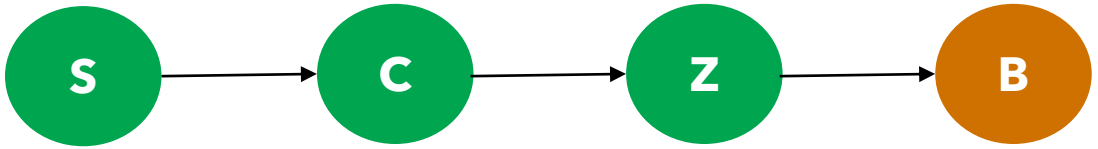
livello 3



TREE *T*

QUEUE Q

```
enqueue(S.right) -> enqueue(B);
```



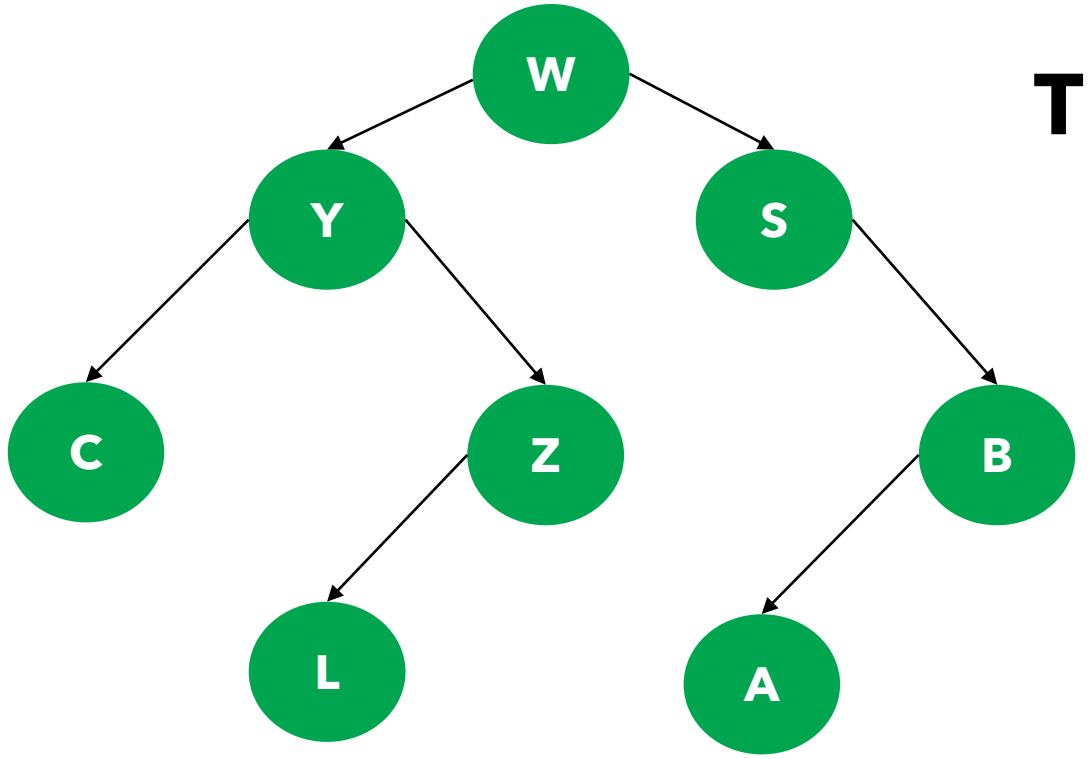
Breadth-first search (BFS) su un albero binario

livello 0

livello 1

livello 2

livello 3



TREE *T*

QUEUE Q

```
dequeue();
```

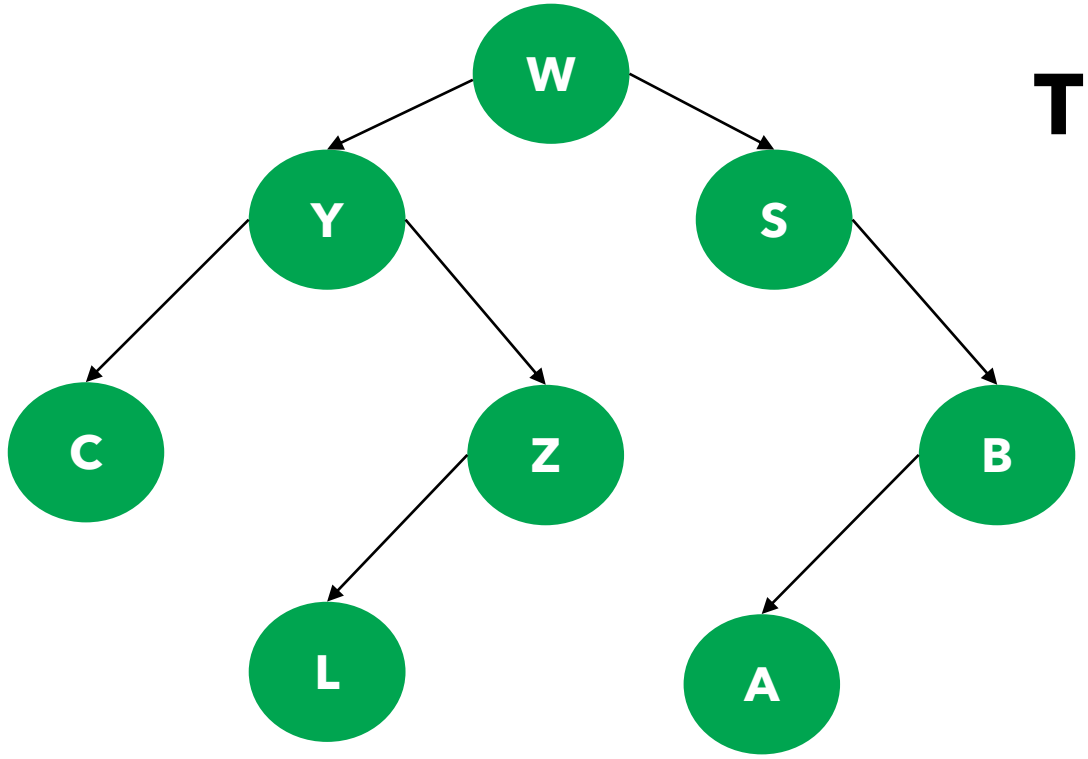
Breadth-first search (BFS) su un albero binario

livello 0

livello 1

livello 2

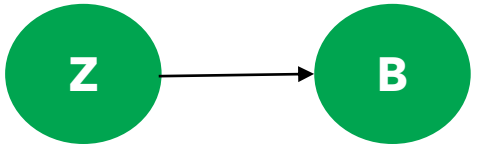
livello 3



TREE *T*

QUEUE Q

```
dequeue();
```



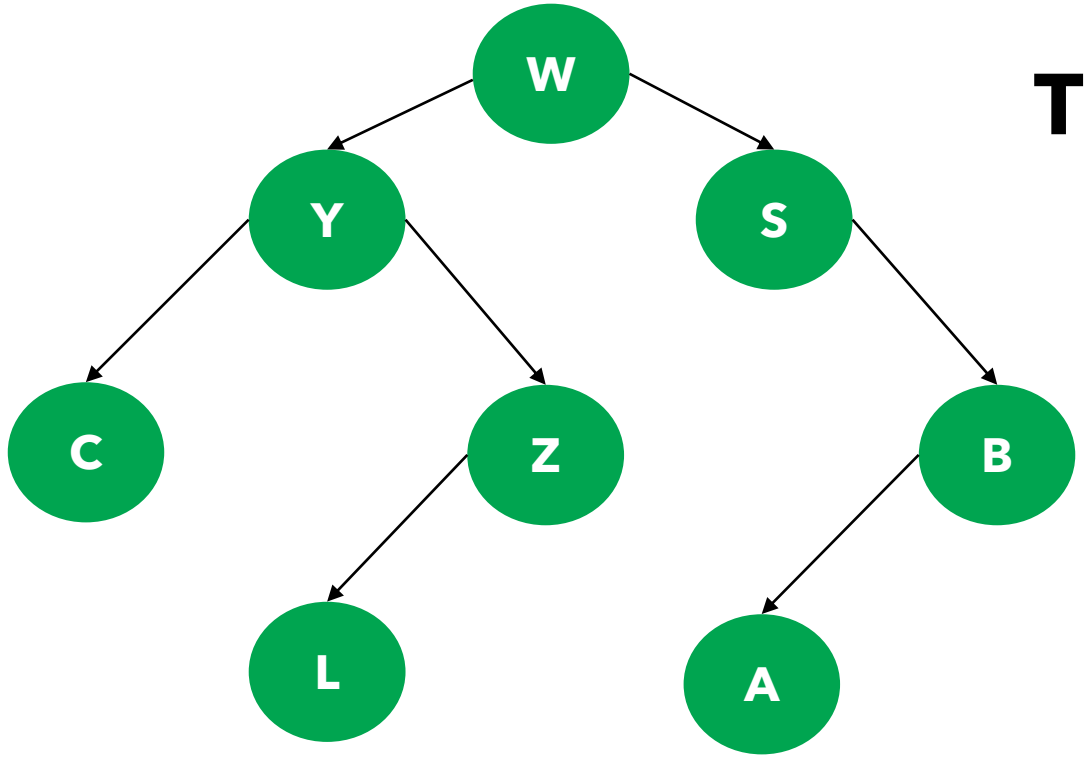
Breadth-first search (BFS) su un albero binario

livello 0

livello 1

livello 2

livello 3



TREE *T*

QUEUE Q

```
enqueue(Z.left) -> enqueue(L);
```

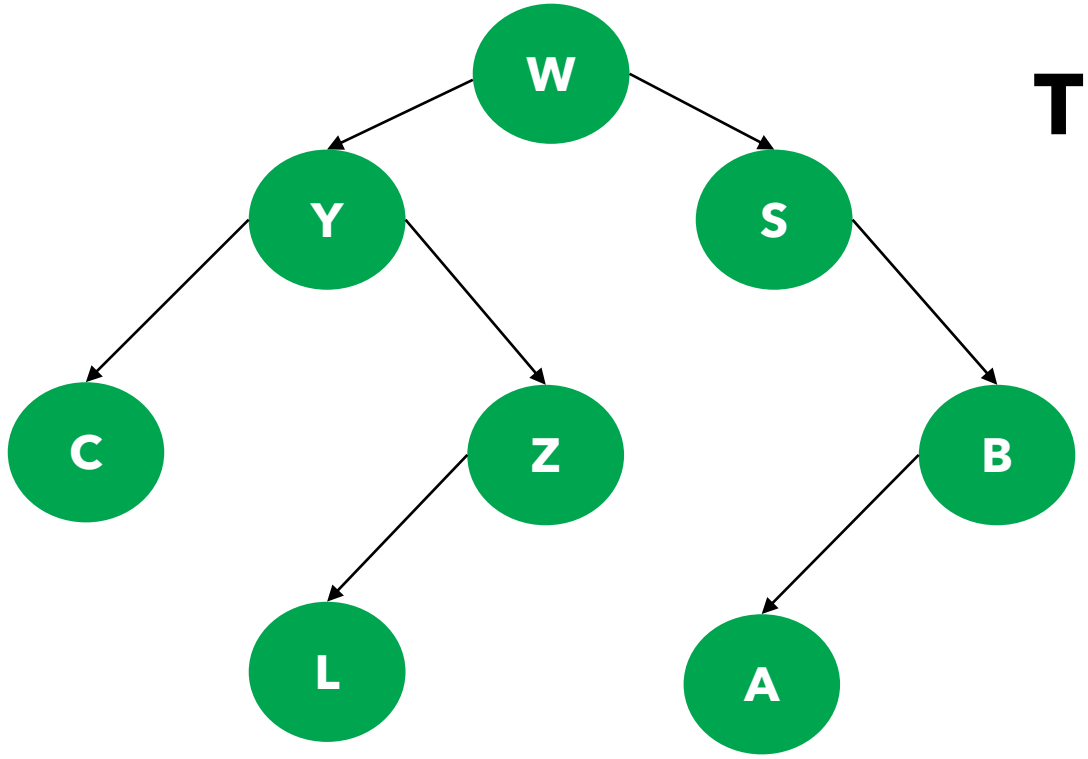
Breadth-first search (BFS) su un albero binario

livello 0

livello 1

livello 2

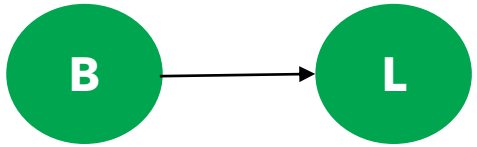
livello 3



TREE *T*

QUEUE Q

```
dequeue();
```



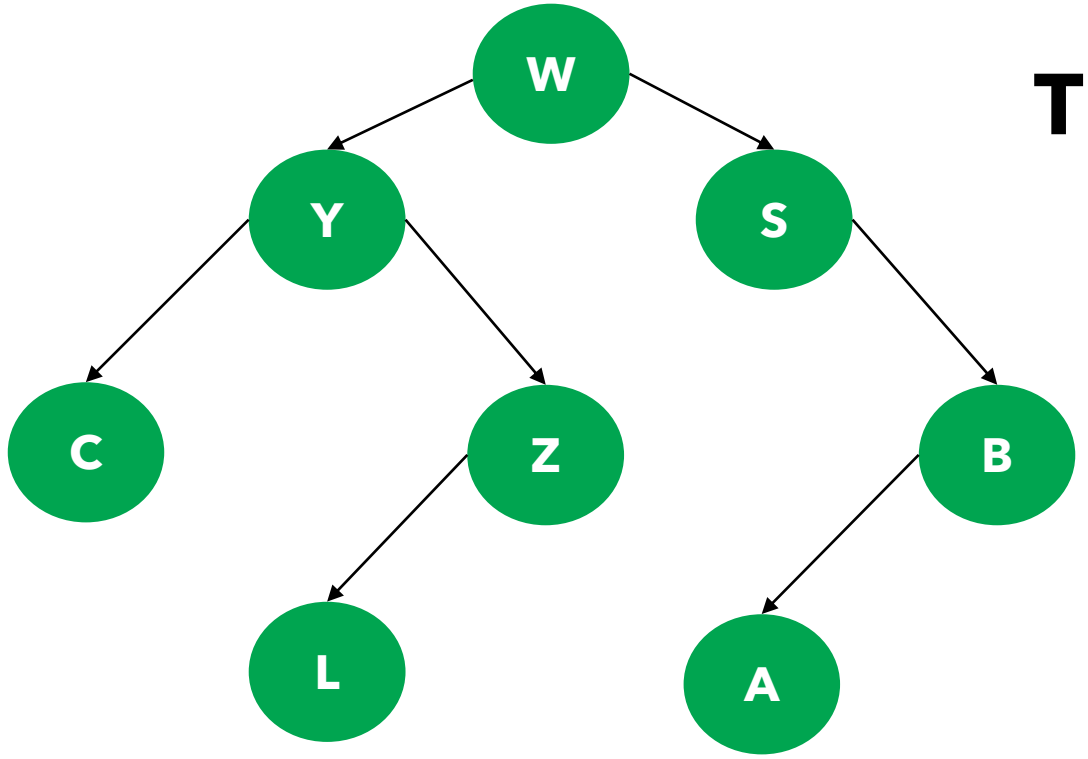
Breadth-first search (BFS) su un albero binario

livello 0

livello 1

livello 2

livello 3



TREE *T*

QUEUE Q

```
enqueue(B.left) -> enqueue(A);
```

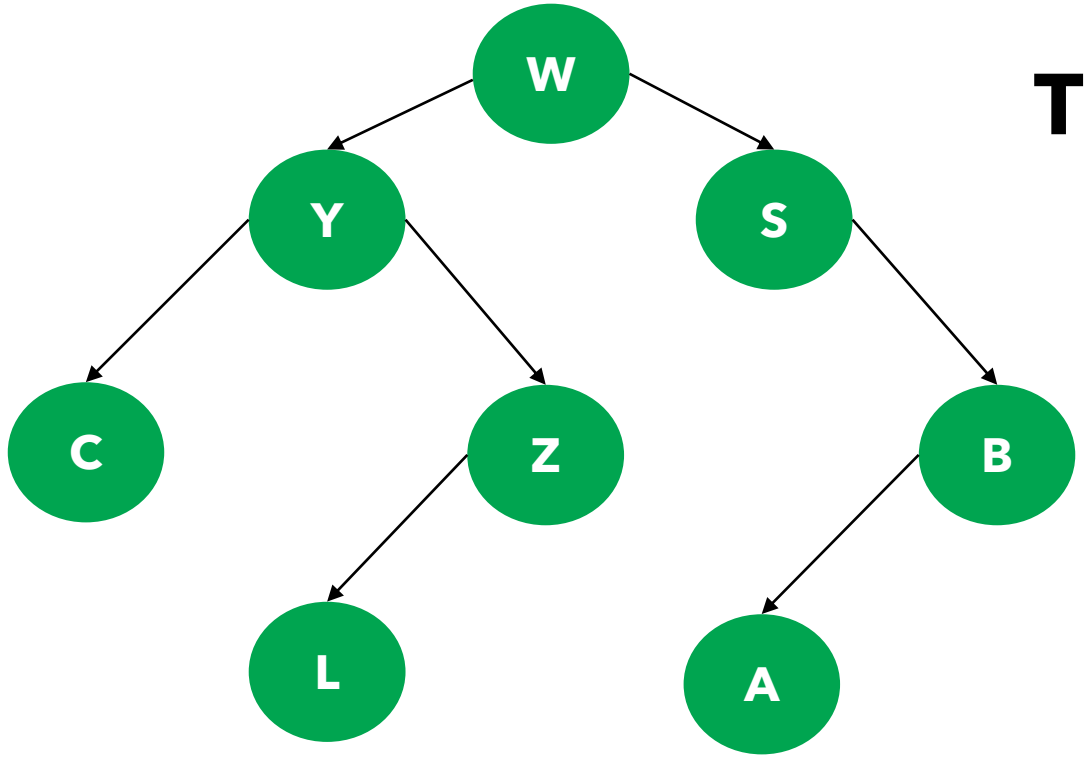
Breadth-first search (BFS) su un albero binario

livello 0

livello 1

livello 2

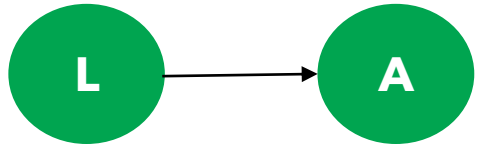
livello 3



TREE *T*

QUEUE Q

```
dequeue();
```



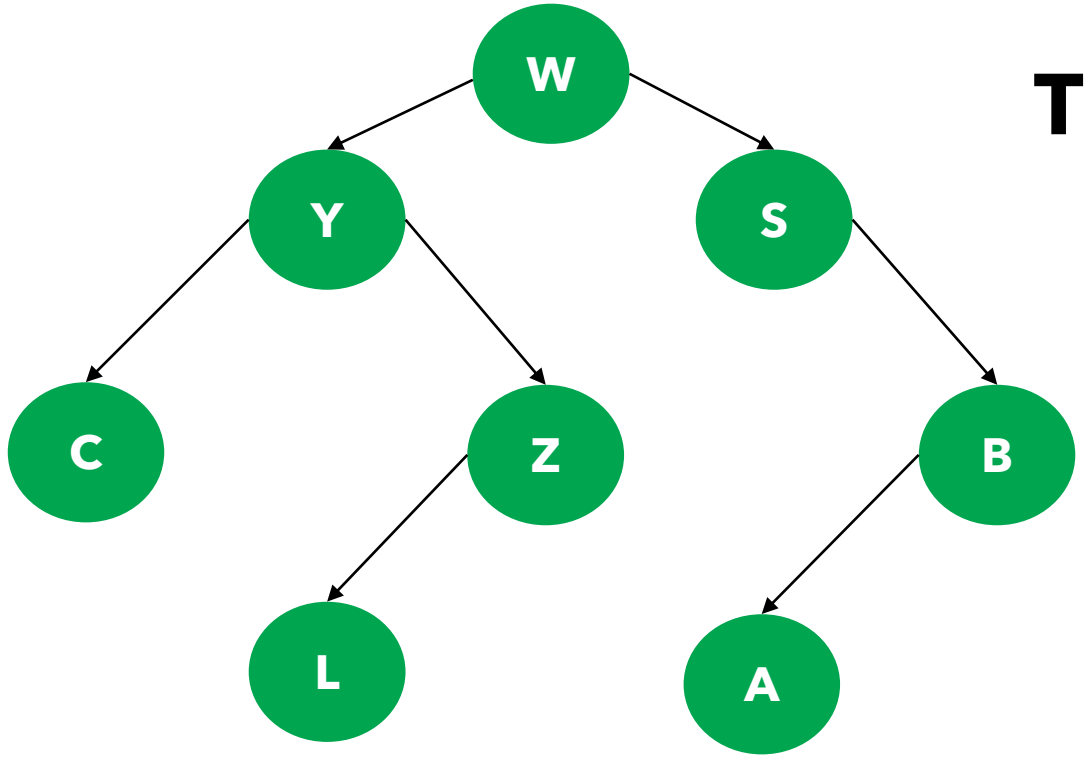
Breadth-first search (BFS) su un albero binario

livello 0

livello 1

livello 2

livello 3



TREE *T*

QUEUE Q

```
dequeue();
```

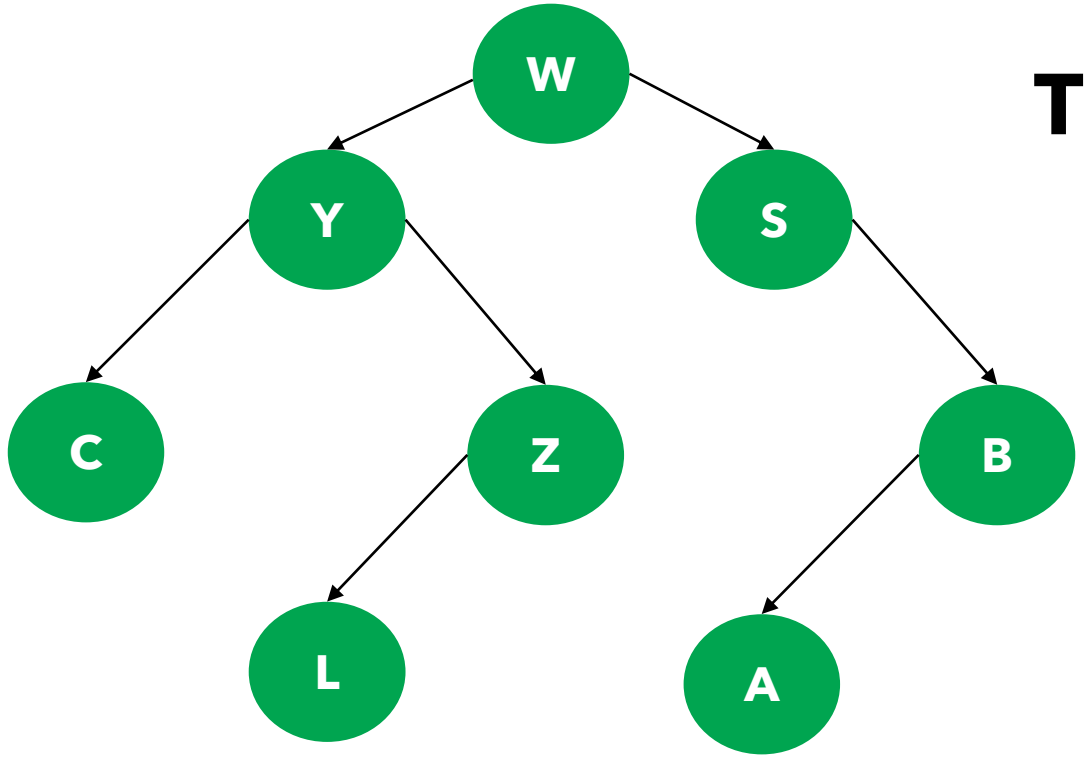
Breadth-first search (*BFS*) su un albero binario

livello 0

livello 1

livello 2

livello 3



TREE *T*

QUEUE Q

```
dequeue();
```

Ricerca ricorsiva di una chiave in un *BST*

```
TREE_NODE *search_bst_recursive(TREE_NODE *t, int k) {
 if (!t){
  return NULL;
 }
 if (k == t->key) {
  return t;
 }
 if (k < t->key) {
  return search_bst_recursive(t->left, k);
 }
 return search_bst_recursive(t->right, k);
}
```

Sappiamo quando andare a sinistra/destra perché è un BST. Gli alberi binari di ricerca si chiamano così proprio perché permettono la ricerca efficiente di una chiave (se sono bilanciati)

Ricerca iterativa di una chiave in un *BST*

```
TREE_NODE *search_bst_iterative(TREE_NODE *t, int k) {
  TREE_NODE *p = t;
  BOOL found_key = false;
  while (p && !found_key) {
    if (k == p->key) {
      found_key = true;
    }
    else if (k < p->key) {
      p = p->left;
    }
    else {
      p = p->right;
    }
  }

  return p;
}
```

Facile da scrivere, a differenza della *DFS* iterativa.
Perché è facile? Perché non serve fare nessun *backtrack*, ad ogni iterazione scendiamo di livello andando o a destra o a sinistra, non torniamo mai indietro

Ricerca della chiave minima in un *BST*

```
TREE_NODE
*bst_minimum_recursive(TREE_NODE *t)
{
  if (t) {
    if (!t->left) {
      return t;
    }
    return bst_minimum_recursive(t->left);
  }
  else {
    return NULL;
  }
}
```

```
TREE_NODE
*bst_minimum_iterative(TREE_NODE *t) {
  TREE_NODE *p = t;
  while (p && p->left) {
    p = p->left;
  }

  return p;
}
```

BST con puntatore al nodo genitore

- Per i prossimi algoritmi ci servirà un BST in cui i ogni nodo «conosce» il proprio genitore
- Nei nodi servirà un puntatore al genitore inizializzato correttamente all'inserimento di un nuovo nodo

- Il tipo dei nodi cambierà, con l'aggiunta del puntatore parent, nel modo seguente:

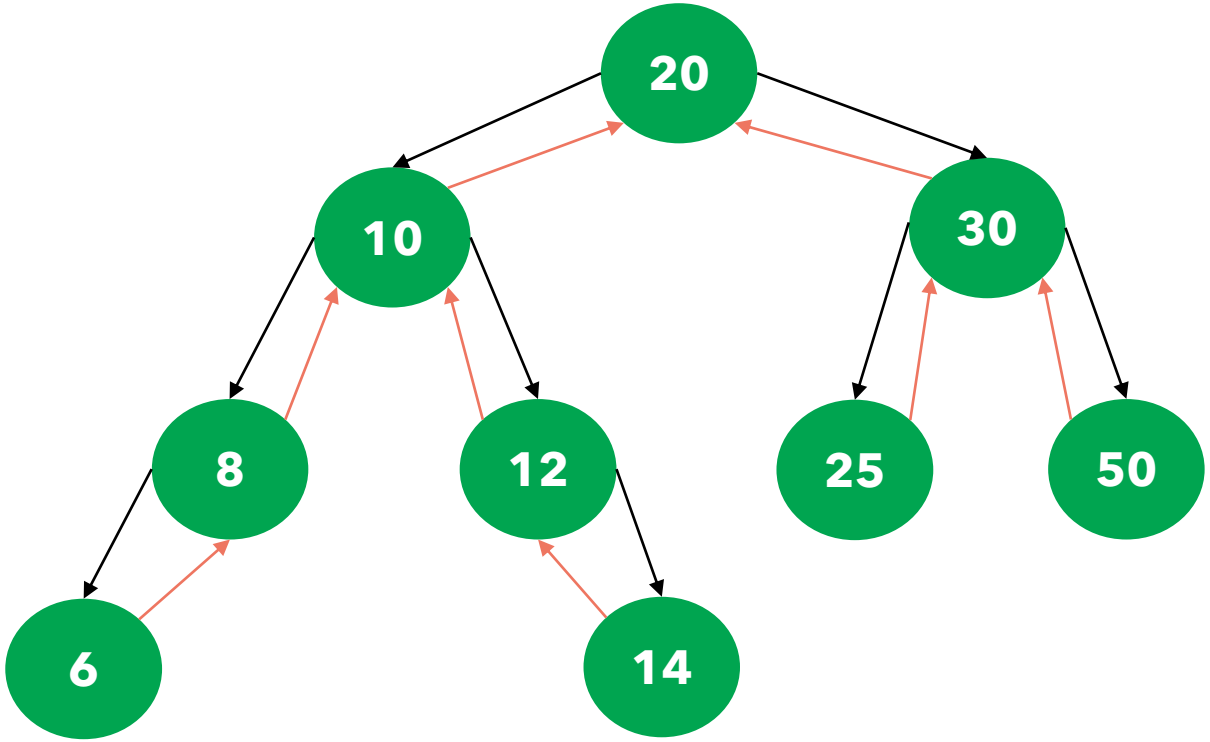
```
typedef struct tree_node {
  int key;
  struct tree_node *left, *right, *parent;
} TREE_NODE;
```

Inserimento iterativo di un nodo in un *BST* con puntatore al genitore

```
bst_insert_iter(T, k):
    current_parent = nil
    last_left = false
    while (T):
        current_parent = T
        if k <= T.key:
            T = T.left
            last_left = true
        else
            T = T.right
            last_left = false

    create new node N
    N.key = k
    N.left = nil
    N.right = nil
    N.parent = current_parent
```

```
if current_parent is not nil:
   if last_left is true:
        current_parent.left = N
   else current_parent.right = N
else
   return N
```

Ricerca del nodo successore in un *BST*



Il successore di un nodo *p* è il nodo *s* tale che *s.key* è la chiave minima tra le chiavi più grandi di *p.key* nell'albero. Se un nodo non possiede un successore, allora l'algoritmo restituirà un puntatore nullo.
Per cercare il successore, in un caso, dovremmo percorrere l'albero verso l'alto, quindi ci tornerà utile il puntatore 'parent'.
Vedrete che non dovremmo neanche considerare i valori delle chiavi per scrivere la procedura corretta.

Ricerca del nodo successore in un *BST*

```
Successor(T): returns TreeNode
    if T.right is not nil:
        return minimum(T.right)
```

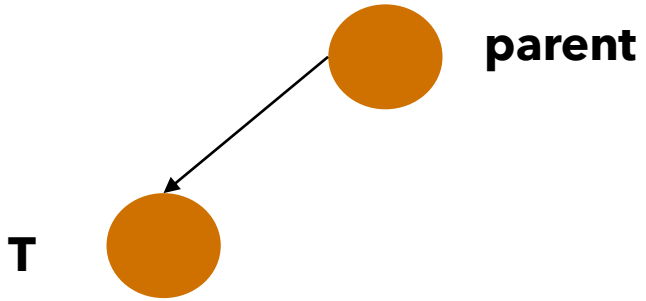
Il successore di un nodo *T* che ha il sottoalbero destro (*T.right != nil*) è il nodo minimo del sottoalbero destro di *T*. Effettivamente, in una sequenza ordinata *p->q->r->s->t->w*, *q* è il successore di *p* perché *q* è l'elemento minimo della sequenza *q->r->s->t->w*

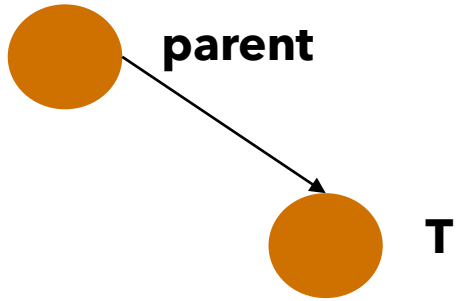
Ricerca del nodo successore in un *BST*

Nel caso in cui il sottoalbero destro di *T* non esista, sicuramente non avrebbe senso cercare il successore di *T* nel sottoalbero *T.left*, in quanto le chiavi dell'albero radicato in *T.left* sono tutte minori o uguali di *T.key*

Quindi, dove cerchiamo il successore? I casi sono 2:

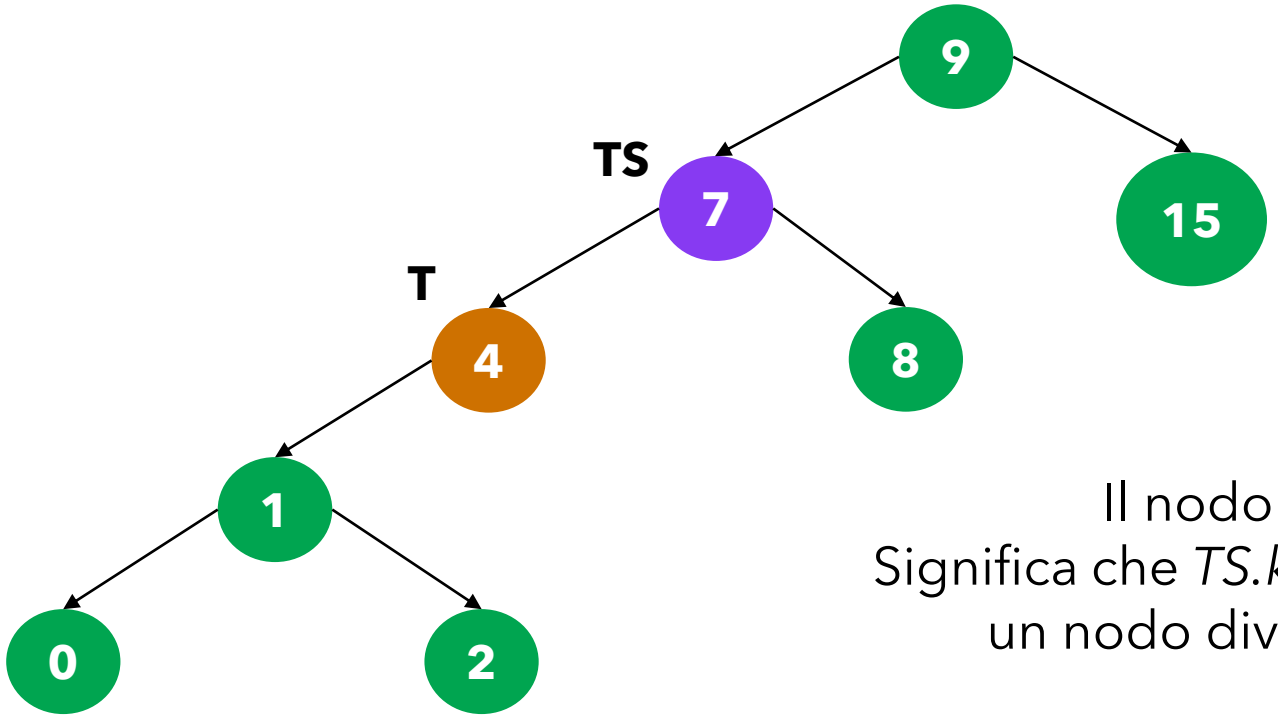
- *T* è il figlio sinistro di *T.parent*
- *T* è il figlio destro di *T.parent*





Ricerca del nodo successore in un *BST*

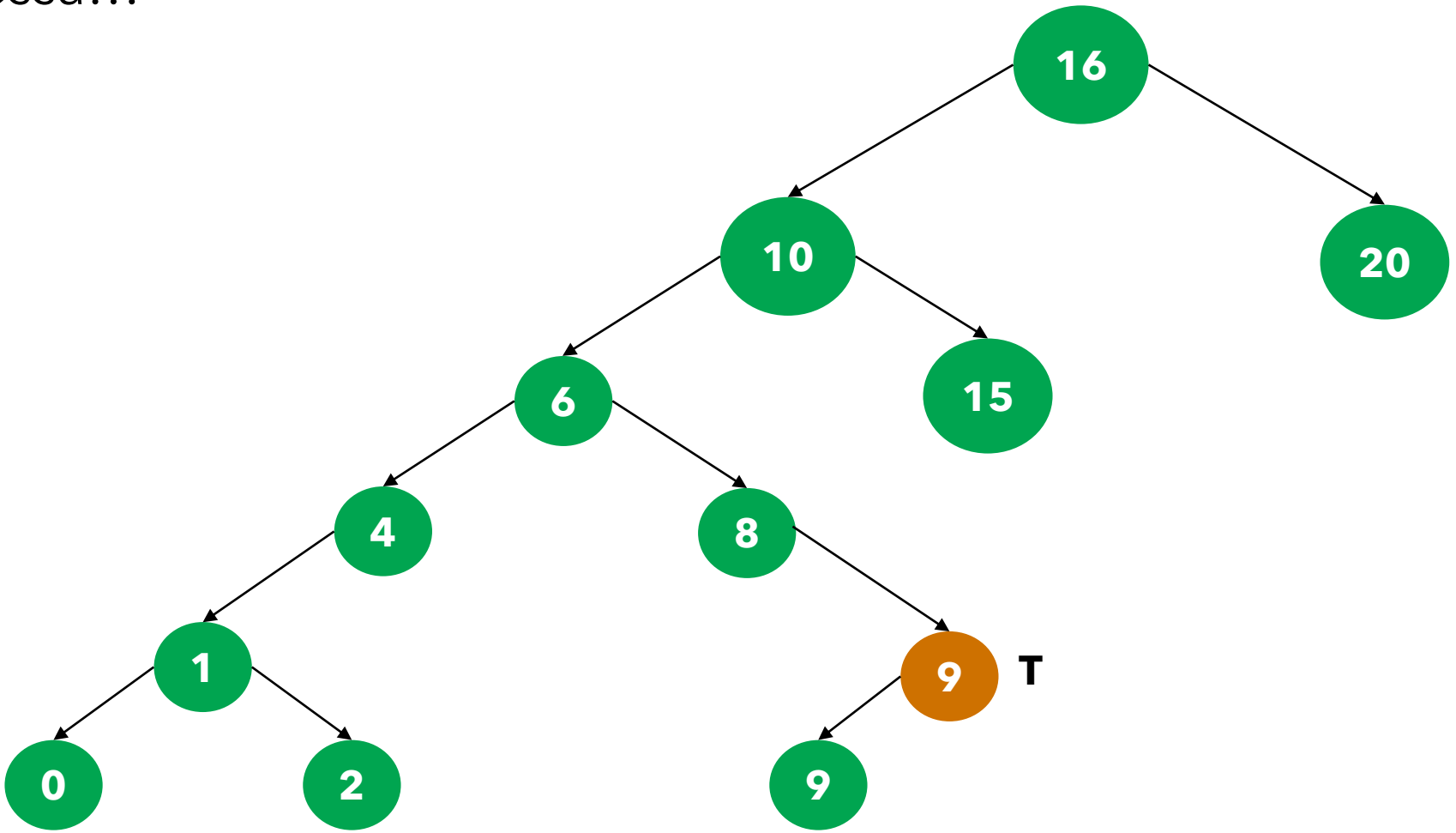
Se *T* è figlio sinistro di *T.parent*, e *T.right* è nullo, allora sicuramente il successore è proprio *T.parent*
Considerare il disegno:



Il nodo *TS* è il successore del nodo *T*.
Significa che *TS.key* è la più piccola chiave nell'albero, in un nodo diverso da *T*, tale che *TS.key* >= *T.key*

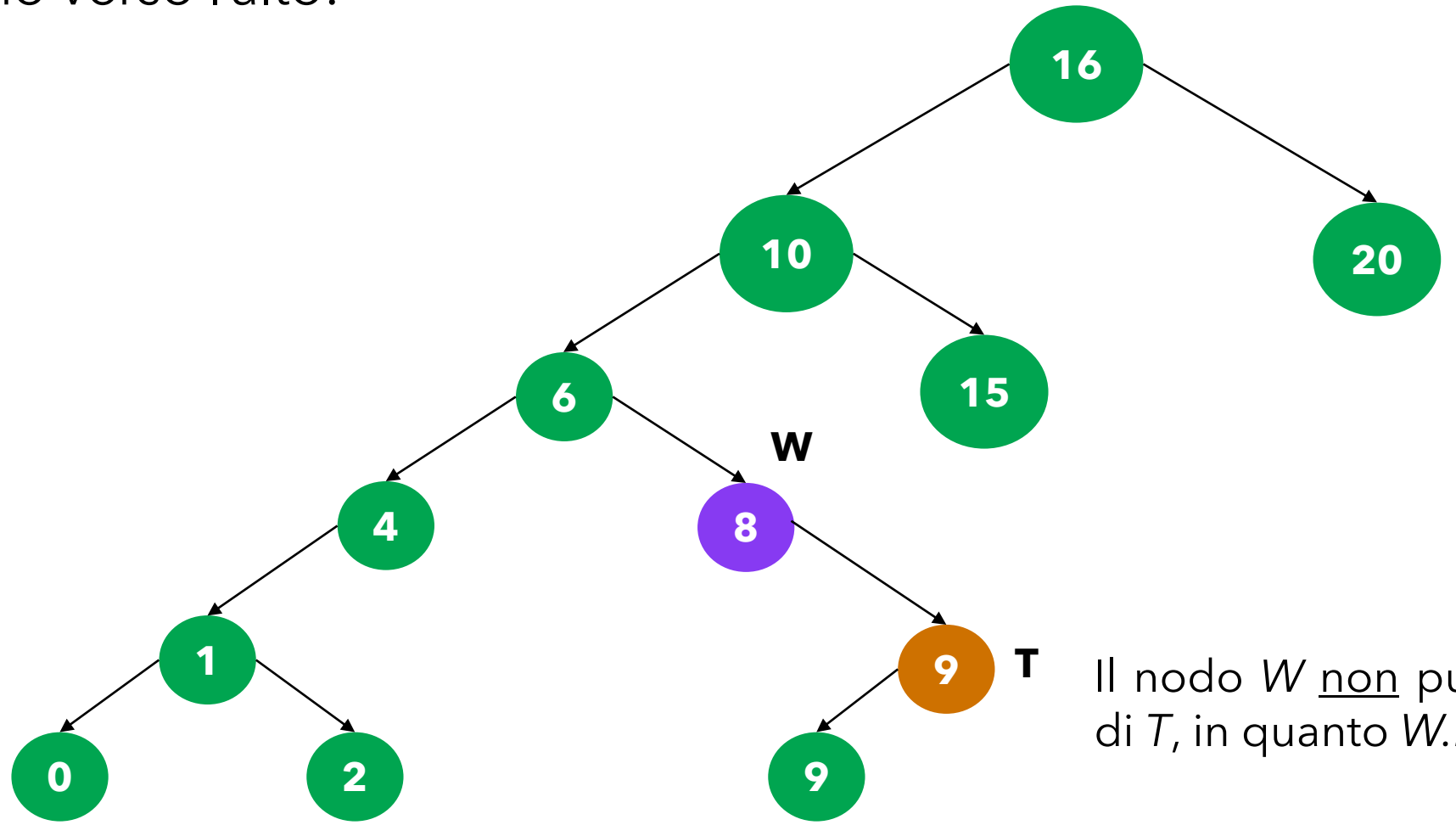
Ricerca del nodo successore in un *BST*

Se *T* è figlio destro di *T.parent* e *T.right* è nullo, allora la ricerca del successore è più complessa…



Ricerca del nodo successore in un *BST*

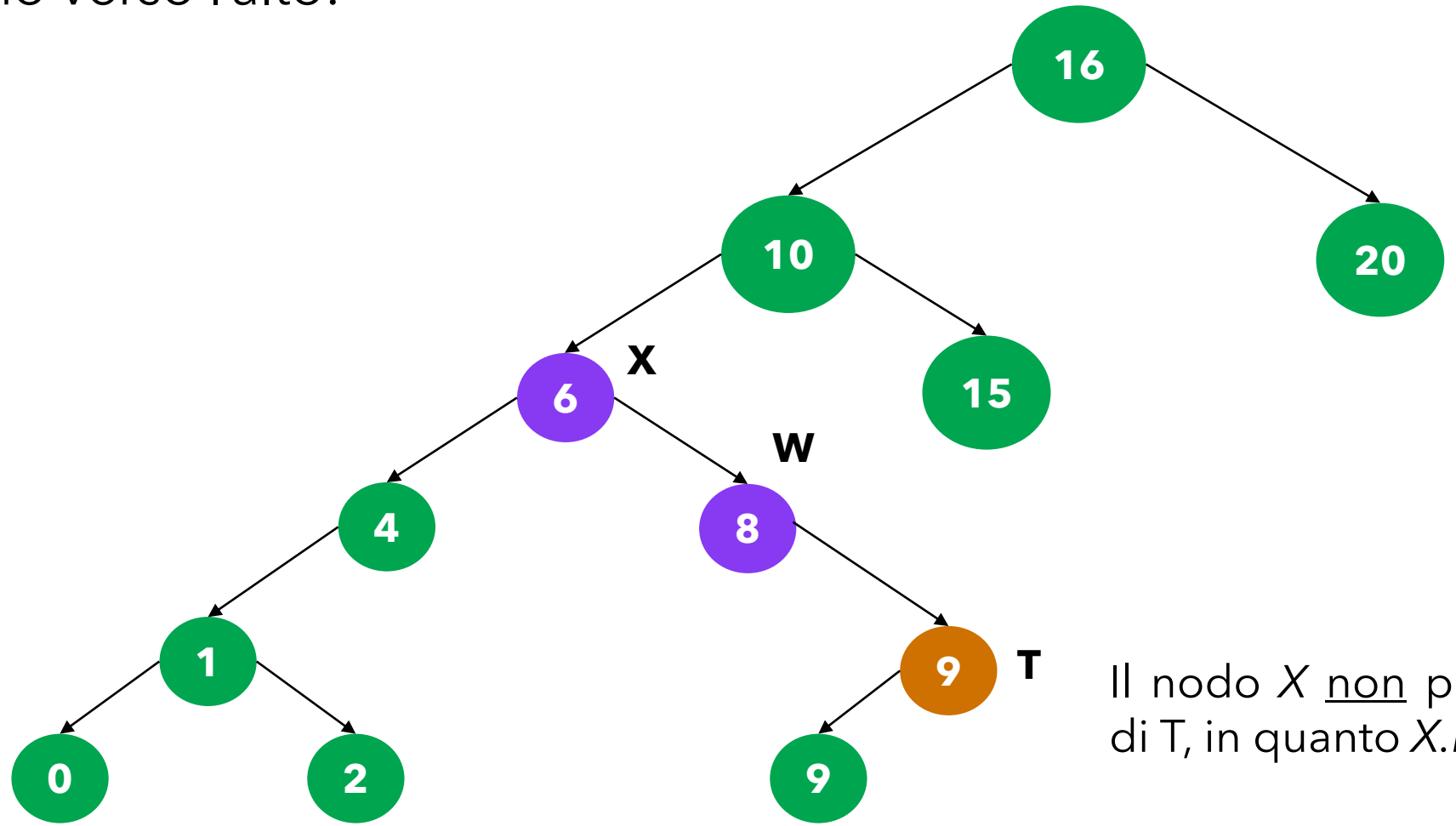
A sinistra non ha senso andare, a destra non possiamo nemmeno provarci… andiamo verso l'alto!



Il nodo *W* non può essere il successore di *T*, in quanto *W.key* < *T.key*

Ricerca del nodo successore in un *BST*

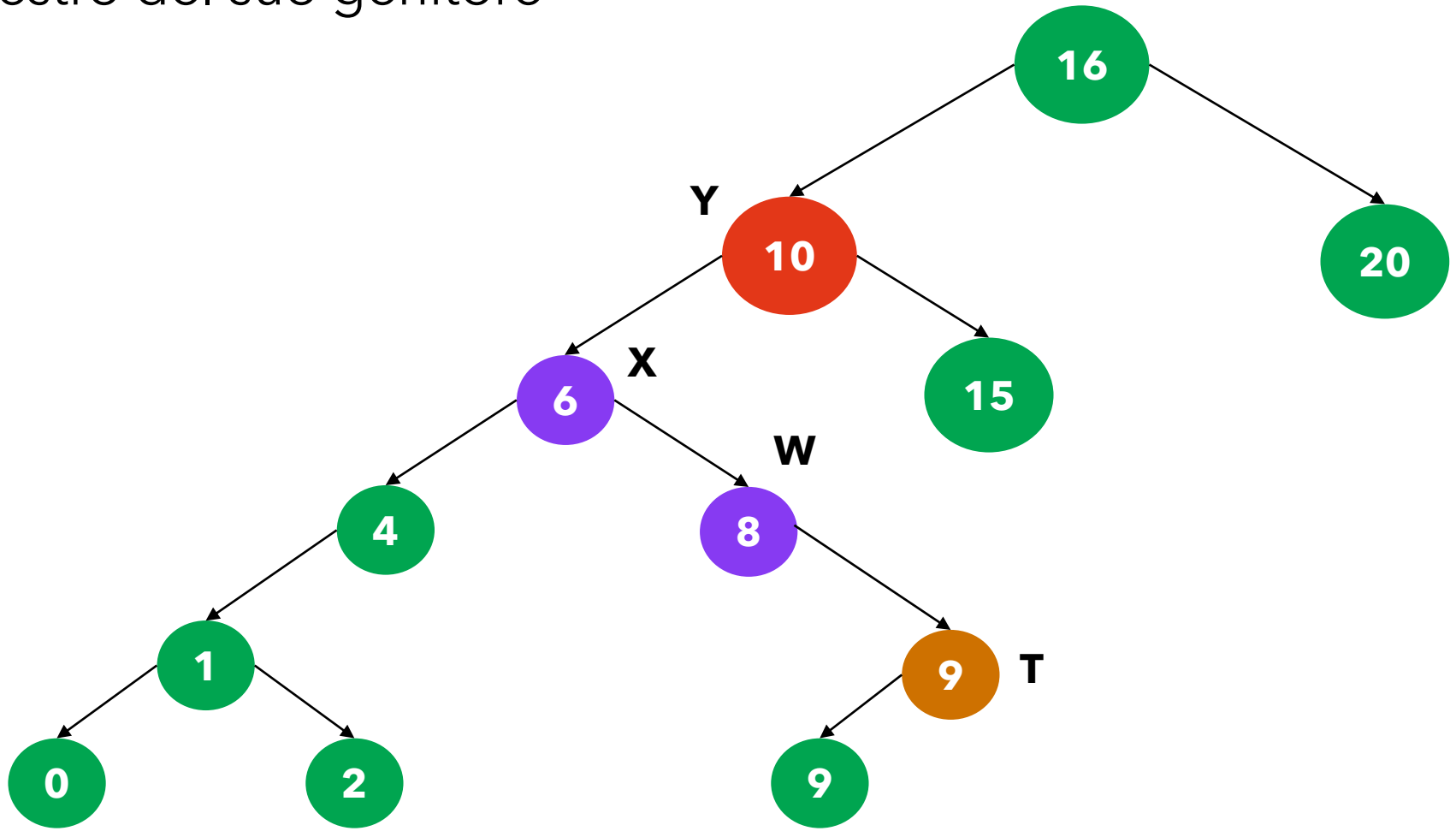
A sinistra non ha senso andare, a destra non possiamo nemmeno provarci… andiamo verso l'alto!



Il nodo *X* <u>non</u> può essere il successore di T, in quanto *X.key* < *T.key*

Ricerca del nodo successore in un *BST*

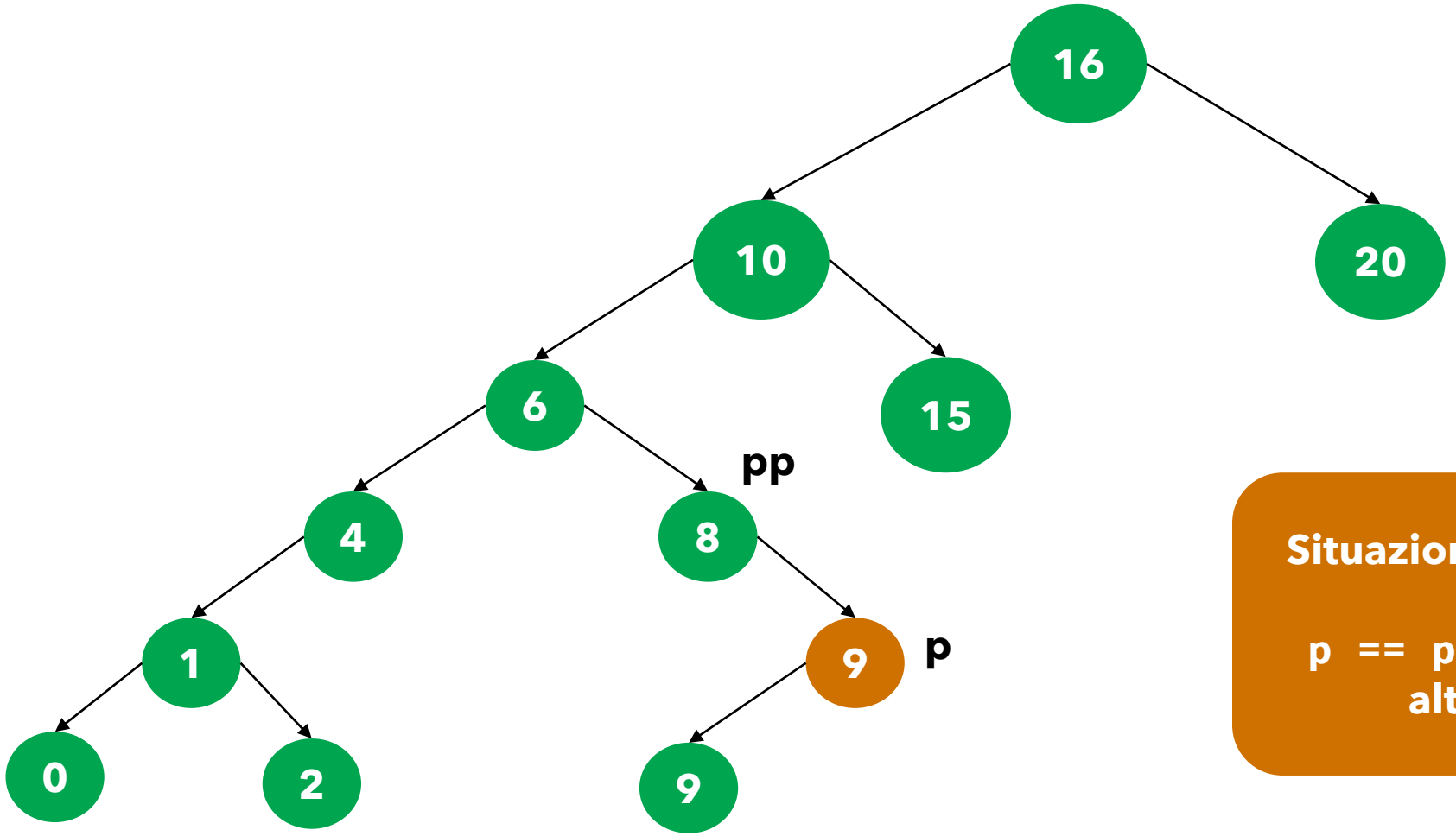
Y è il successore di *T*. Ci siamo «arrampicati» sull'albero fintantoché l'iteratore era figlio destro del suo genitore



Ricerca del nodo successore in un *BST*

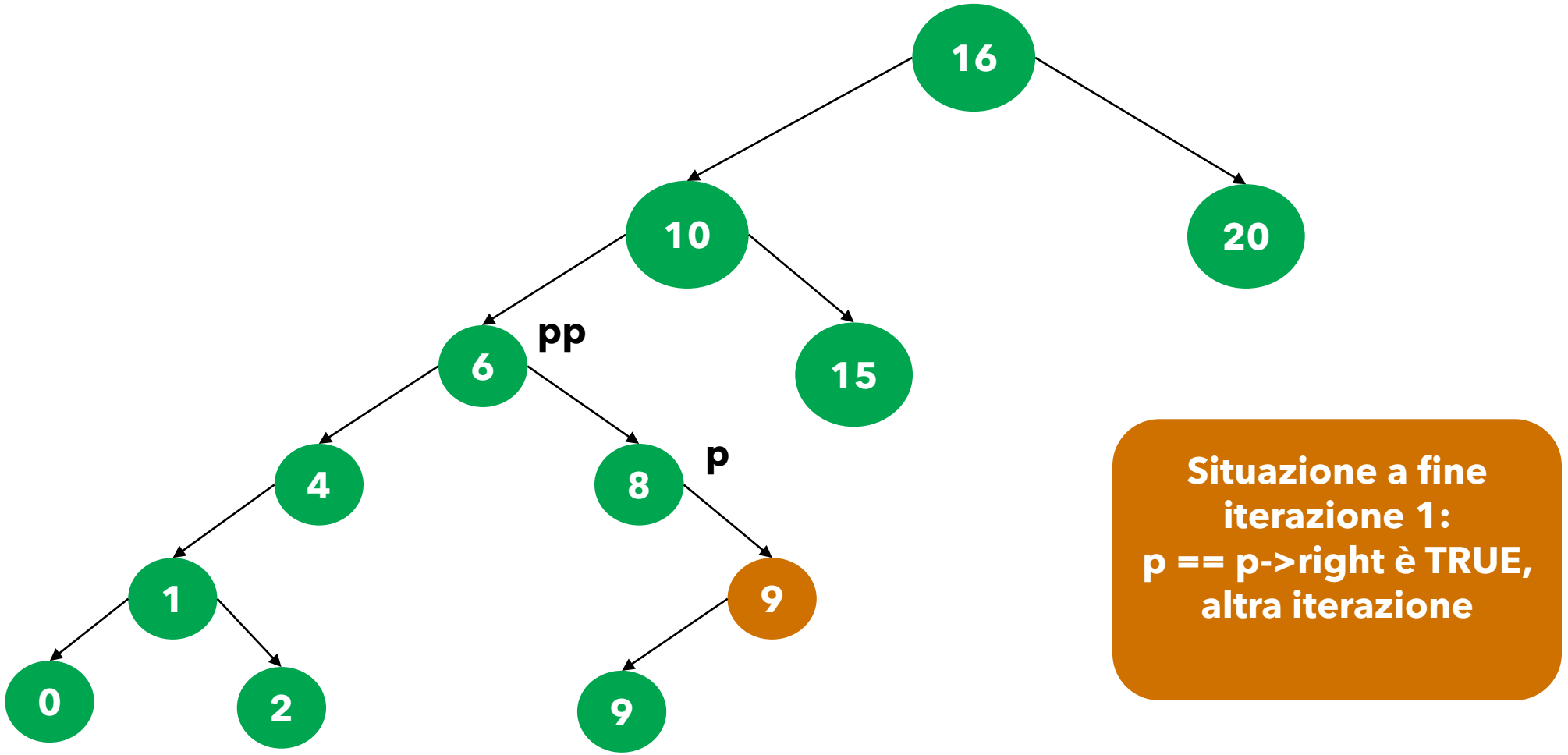
```
TREE_NODE *bst_successor(TREE_NODE *p) {
 if (!p) {
  return NULL;
 }
 if (p->right) {
  return bst_minimum_iterative(p->right);
 }

 TREE_NODE *pp = p->parent;
 /*
  while pp exists and p is its right child
 */
 while (pp && p == pp->right) {
  p = pp;
  pp = pp->parent;
 }

 return pp;
}
```

Ricerca del nodo successore in un *BST*



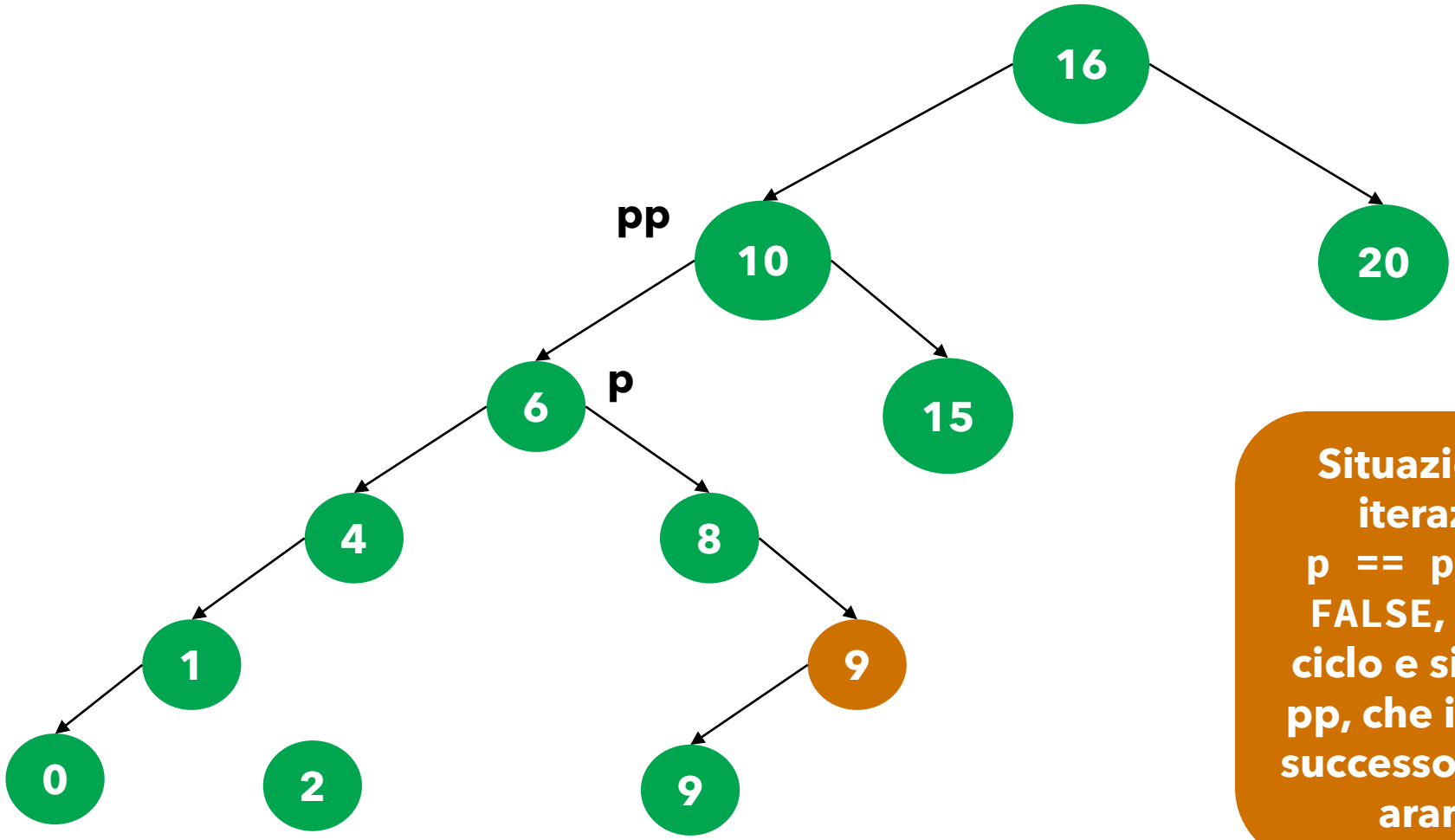
Situazione a fine iterazione 0:
`p == pp->right` è TRUE, altra iterazione

Ricerca del nodo successore in un *BST*



Situazione a fine iterazione 1:
p == p->right è TRUE, altra iterazione

Ricerca del nodo successore in un *BST*



Situazione a fine iterazione 2:
`p == p->right` **è FALSE, si esce dal ciclo e si restituisce pp, che in effetti è il successore del nodo arancione!**

Ricerca del nodo predecessore in un *BST*

```
Predecessor(T): returns TreeNode
    if T.left is not nil:
        return maximum(T.left)

etc…
```

Per realizzare questo algoritmo basta «invertire» quello del successore

Verificare se un albero è un *BST*

- Scriviamo un algoritmo ricorsivo per verificare se un albero radicato su *T* è un *binary-search tree*
- Cerchiamo di scriverlo sfruttando la struttura ricorsiva/autosomigliante di un albero binario, senza pensare a cosa succede sullo *stack*
- Partire dai casi base: se *T* è vuoto… allora è un BST
- Gli altri casi base non sono altro che la negazione della proprietà dei *BST*!
- E i casi ricorsivi? A cosa servono?

Verificare se un albero è un *BST*

```
verifyBST(T): returns true or false
      if T is nil:
            return true
      if T.left and T.left.key > T.key:
            return false
      if T.right and T.right.key <= T.key:
            return false
      isLeftBST = verifyBST(T.left)
      if isLeftBST is FALSE:
            return false
      else:
            return verifyBST(T.right)
```

Analogia sulle liste

- L'algoritmo precedente assomiglia ad una procedura ricorsiva che verifica se un array, oppure una lista concatenata è ordinata

- Pensateci:
  - un array/lista vuoto è sicuramente ordinato
  - un array/lista con un solo elemento è sicuramente ordinato
  - … c'è un altro caso base
  - e la ricorsione a cosa serve?

Analogia sulle liste

```
isSorted(node): returns true or false
      if node is nil:
          return TRUE
      if node.next is nil:
          return TRUE
      if node.key > node.next.key:
          return FALSE
      return isSorted(node.next)
```