Alberi binari in C
Pensiero algoritmico avanzato

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

Binary search trees in C

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

int main() {
  TREE_NODE *t = NULL;
}
```

salvare il programma in un file con nome bst.c

Compilare con:
gcc -o bst bst.c

Eseguirlo con:
./bst

Inserimento di un nodo in un albero binario di ricerca

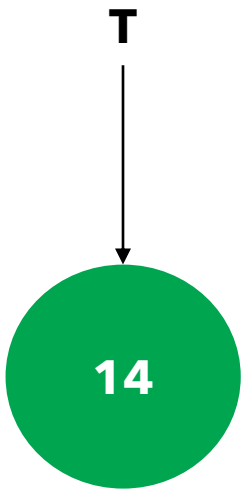
bst_insert(T, key):

- se l'albero T è vuoto, allora restituisci un nuovo nodo con chiave key, puntato da T
- 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 albero binario di ricerca

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

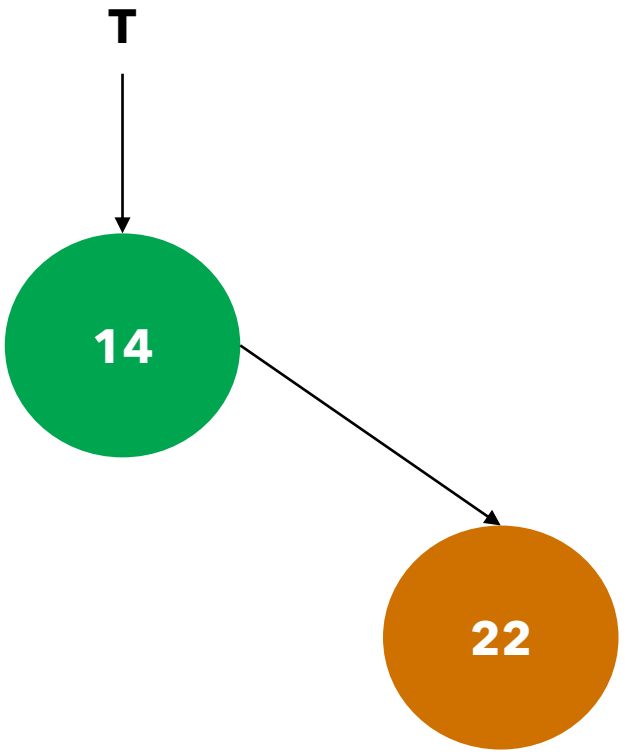
Caso base, si crea un nuovo nodo e lo si fa puntare da T



Inserimento di un nodo in un albero binario di ricerca

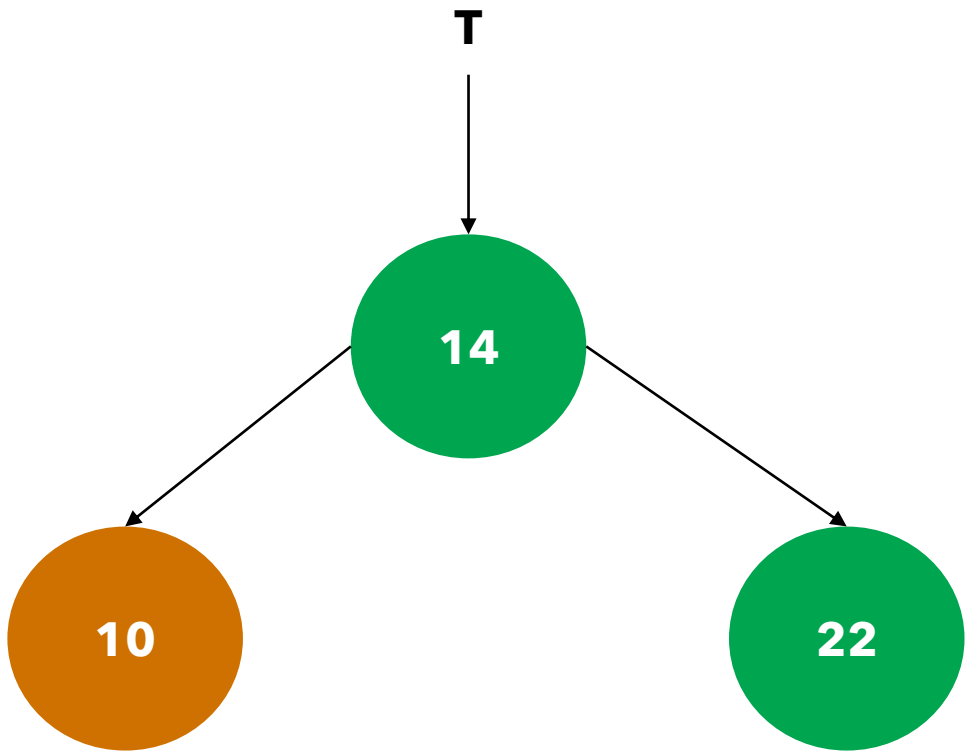
```
bst_insert(T, 22)
```

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 albero binario di ricerca

```
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
```



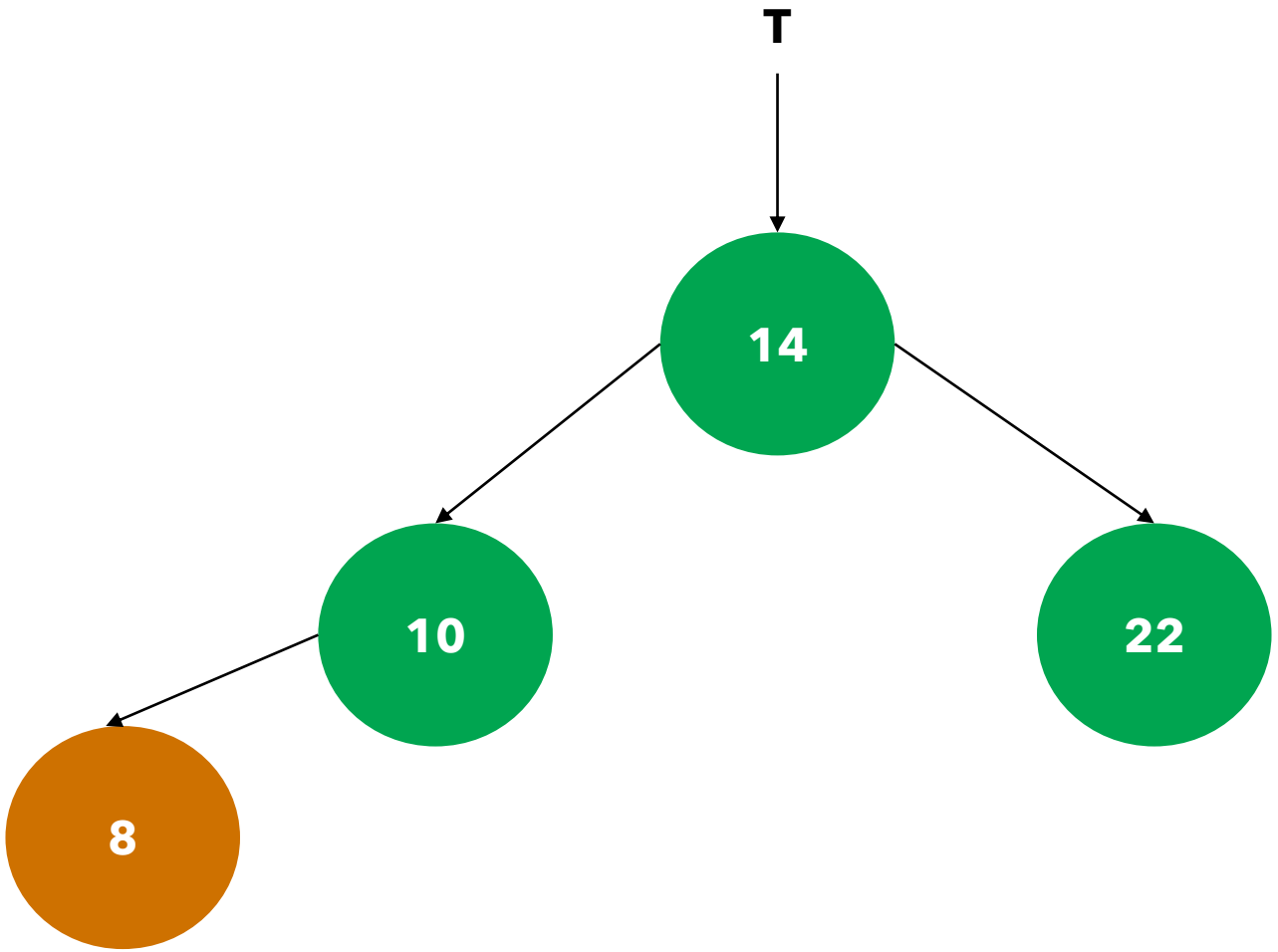
Inserimento di un nodo in un albero binario di ricerca

```
bst_insert(T, 8)
```

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

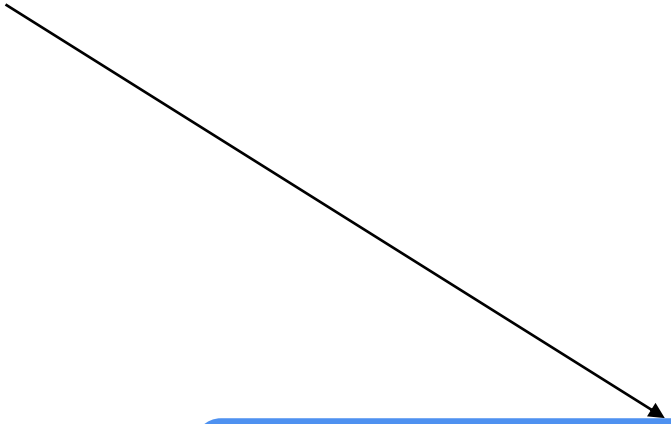
Caso ricorsivo: 8 < 10: si inserisce un nuovo nodo con chiave 8 nel sottoalbero sinistro di T.left;

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



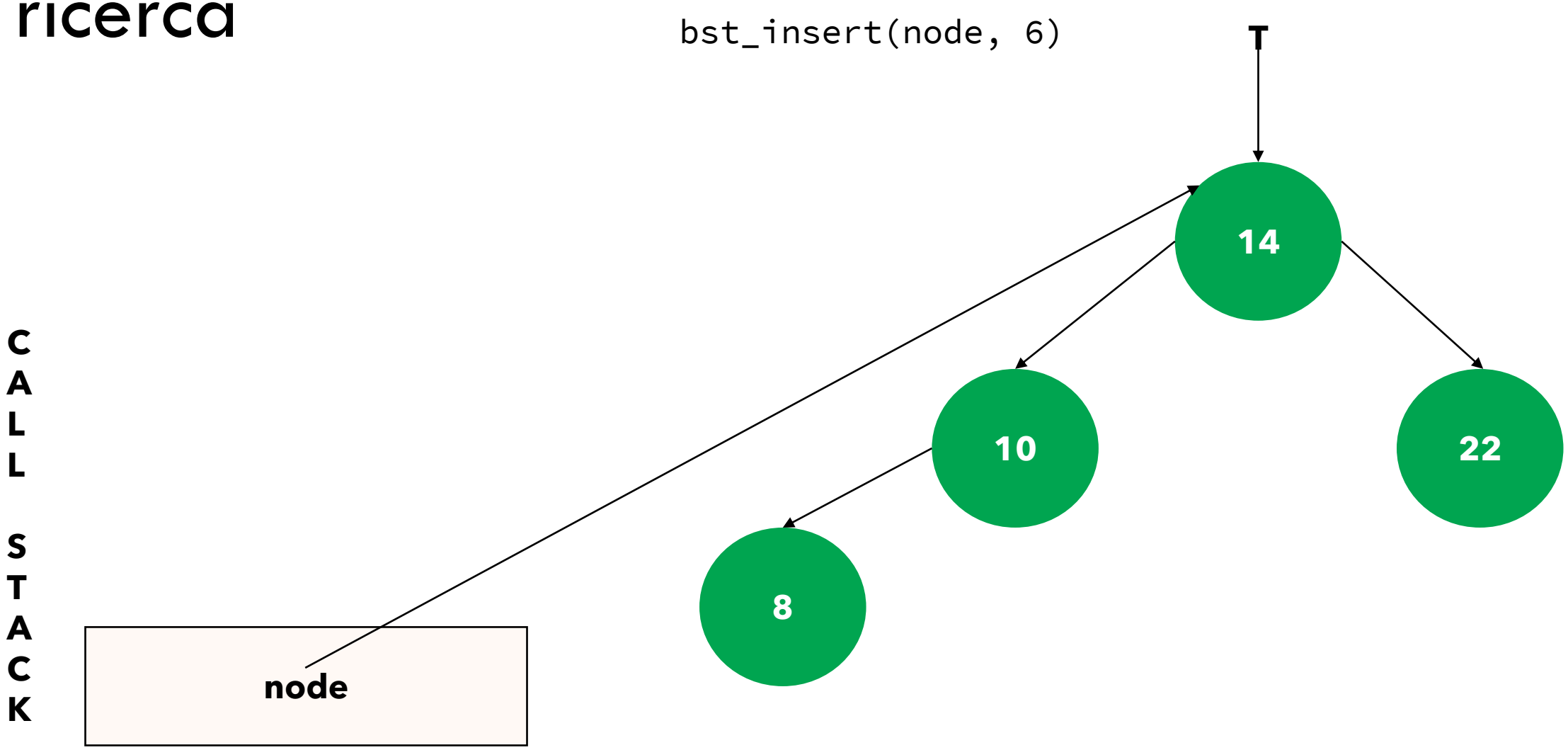
Inserimento di un nodo in un albero binario di ricerca

```
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;
}
```

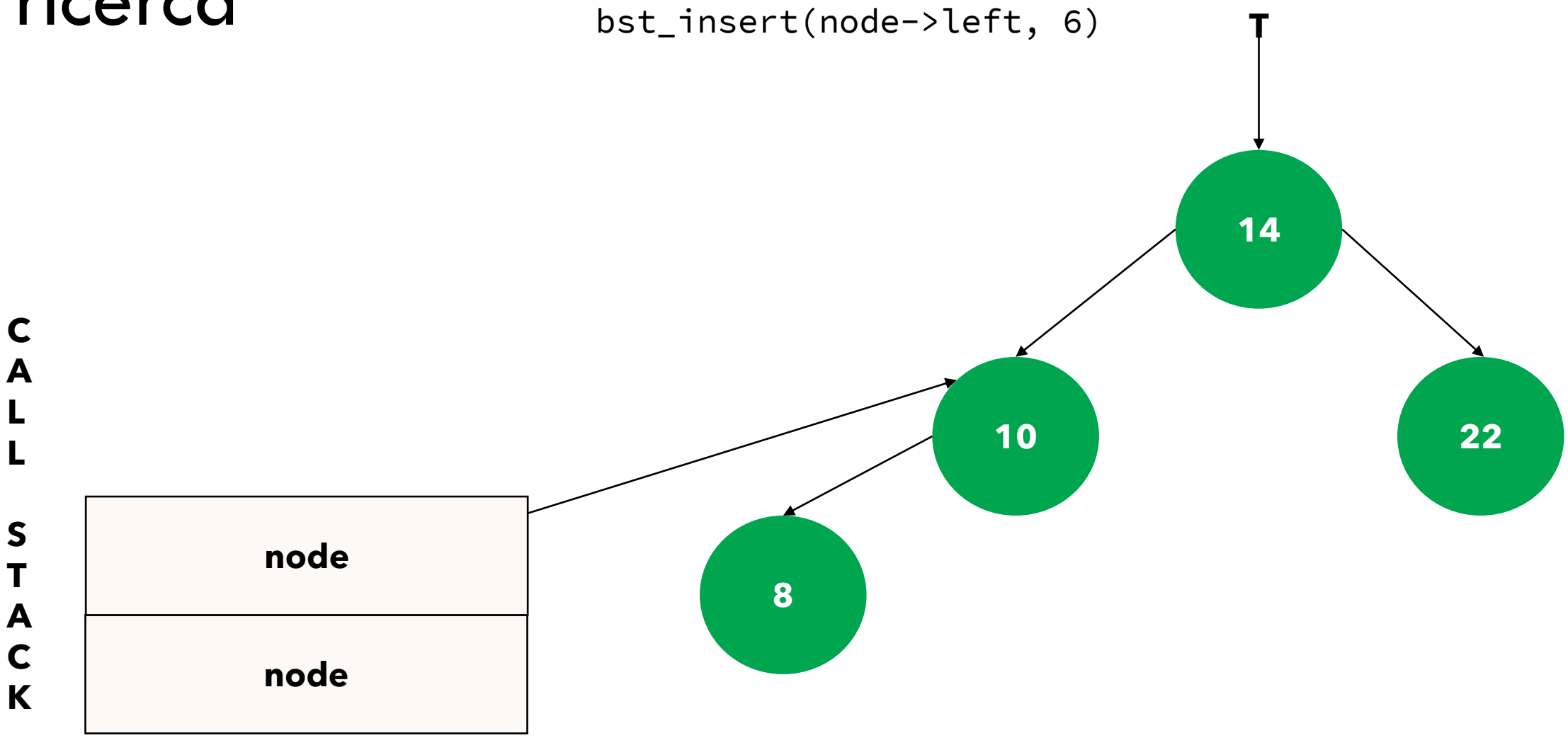


equivale a new in Java. Alloca un oggetto nella memoria heap

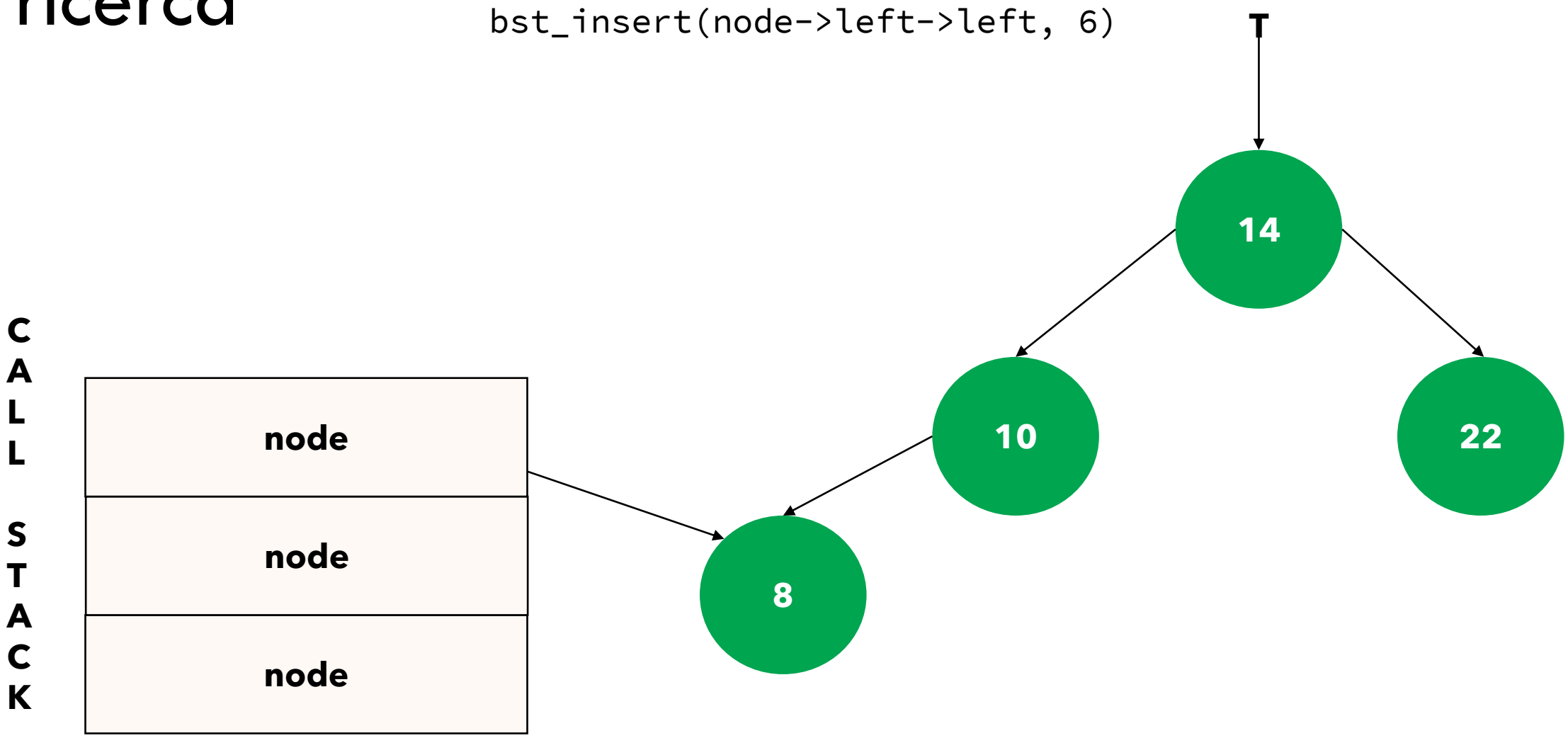
Inserimento di un nodo in un albero binario di ricerca



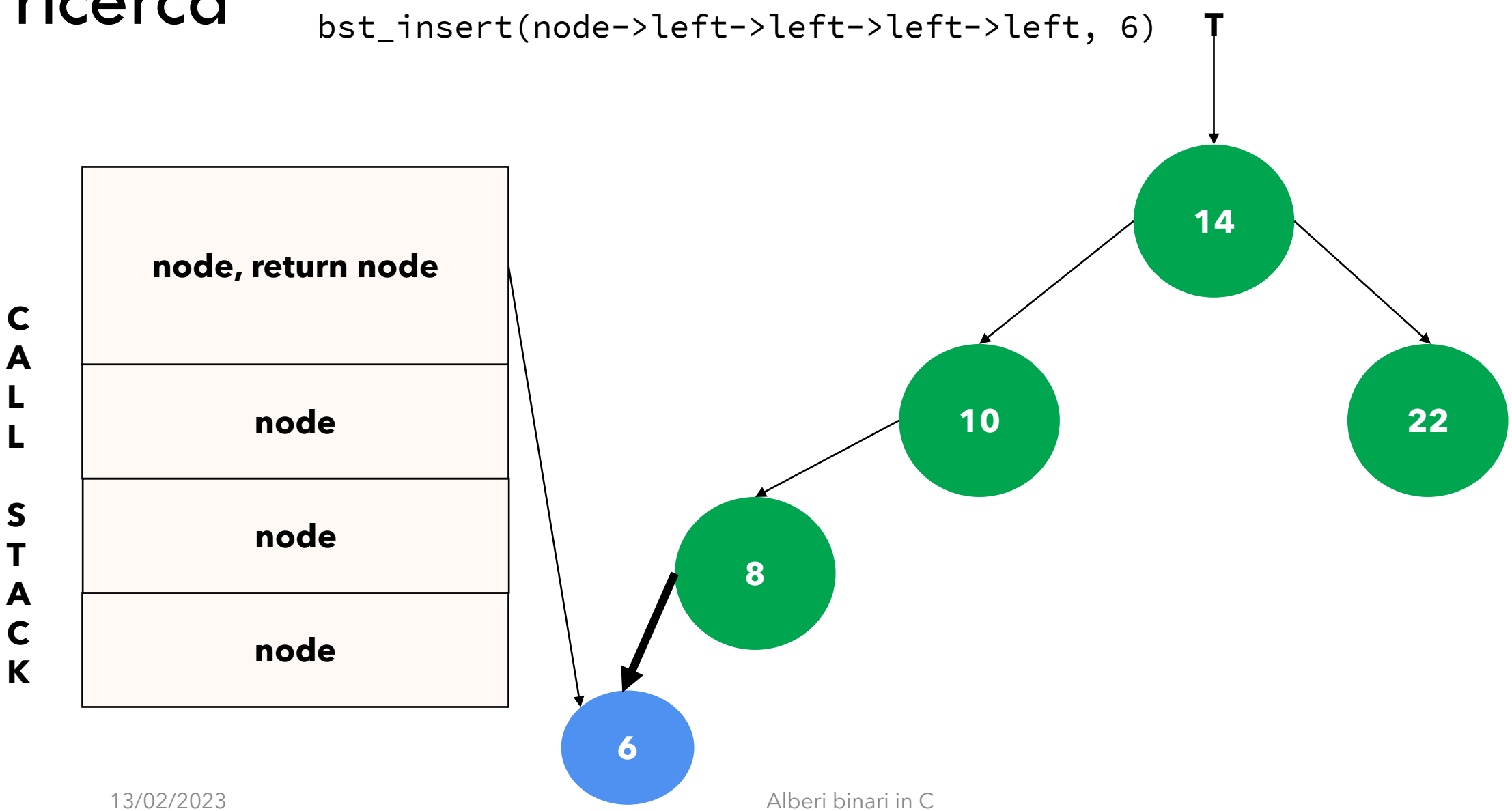
Inserimento di un nodo in un albero binario di ricerca



Inserimento di un nodo in un albero binario di ricerca

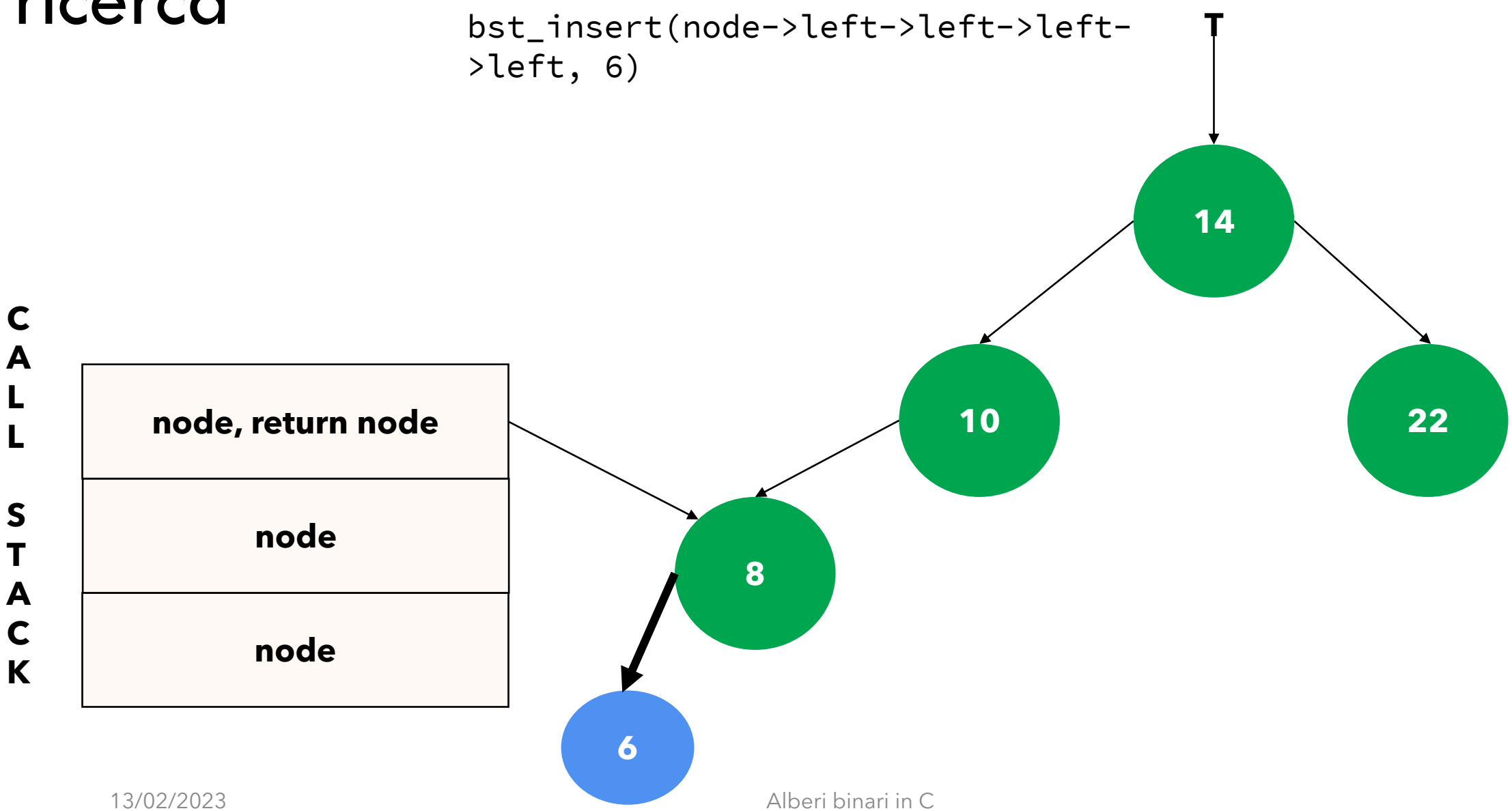


Inserimento di un nodo in un albero binario di ricerca

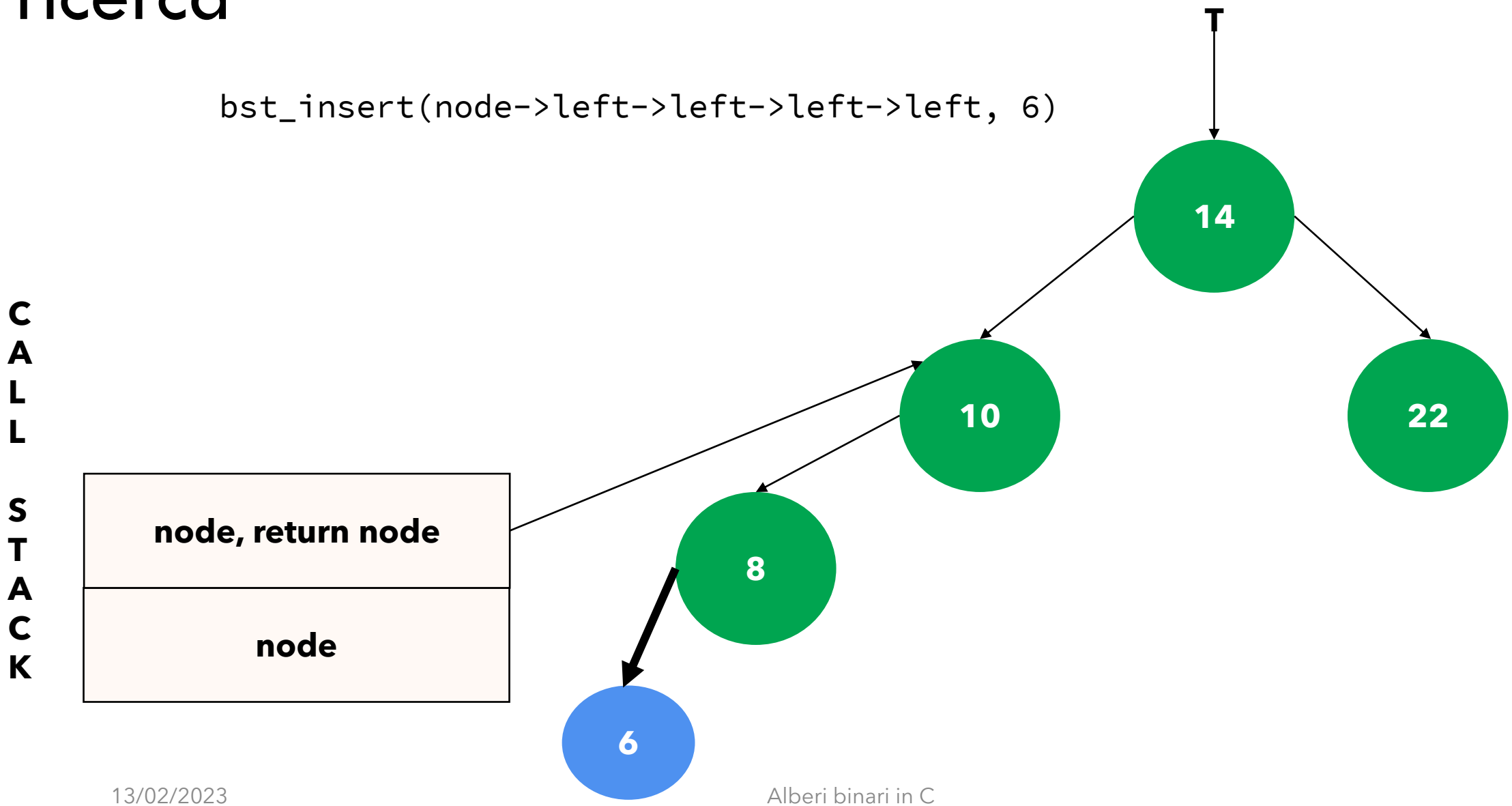


Inserimento di un nodo in un albero binario di ricerca

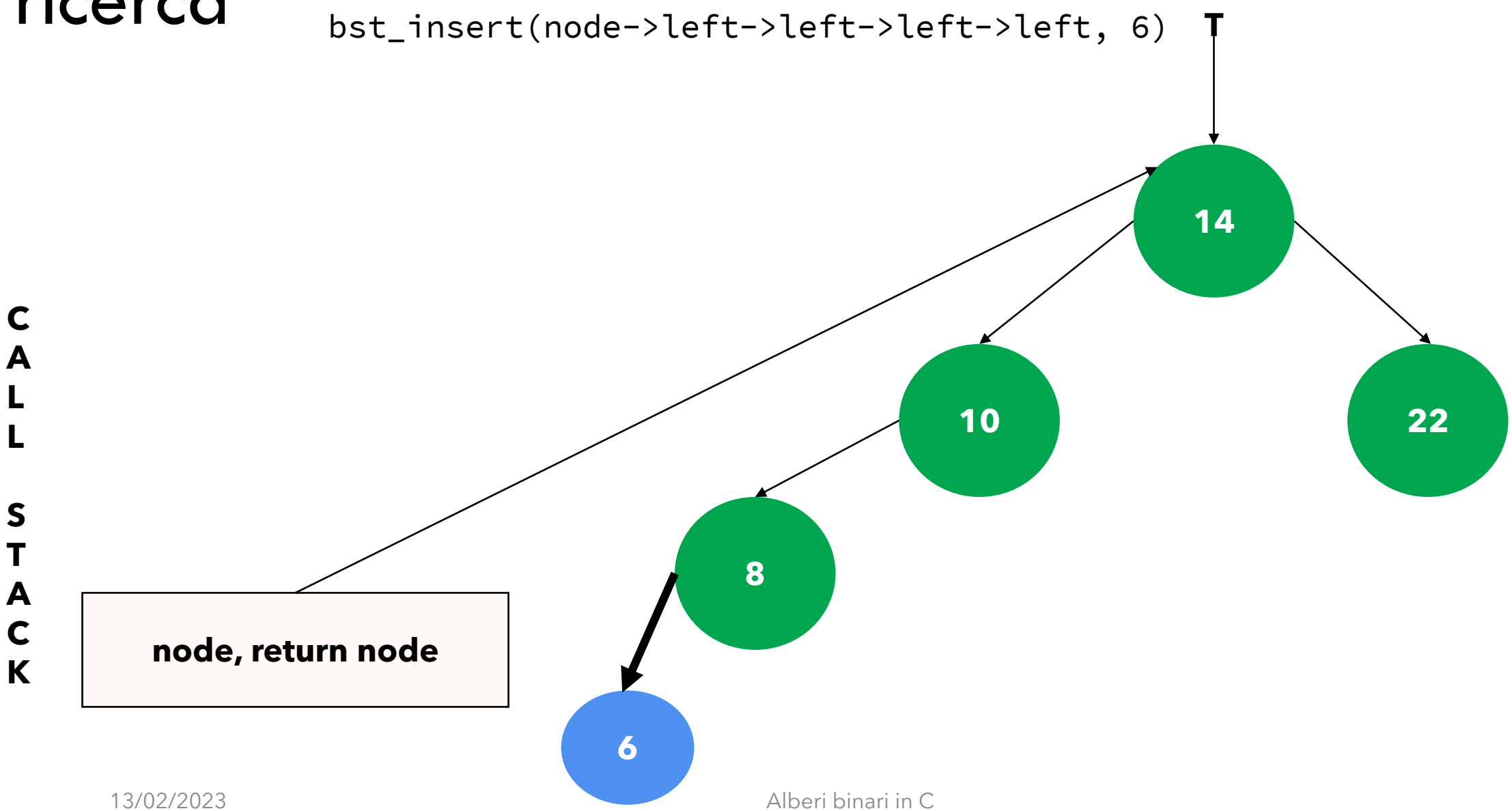
```
bst_insert(node->left->left->left->left, 6)
```



Inserimento di un nodo in un albero binario di ricerca



Inserimento di un nodo in un albero binario di ricerca

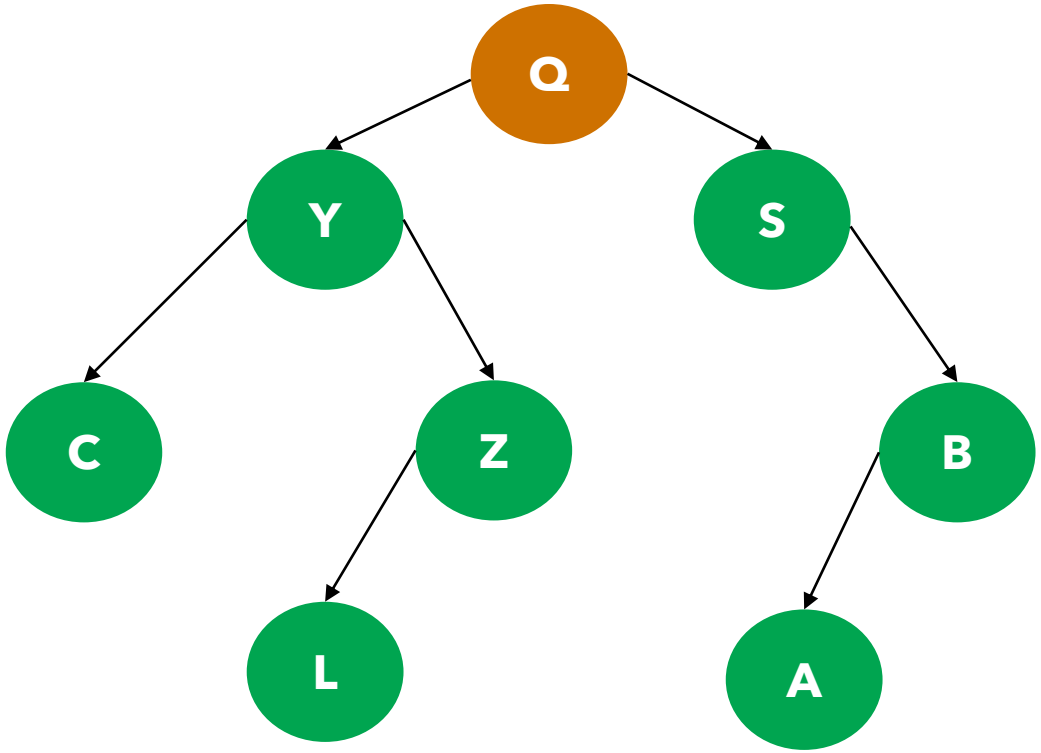


Depth-first search (DFS) su alberi binari

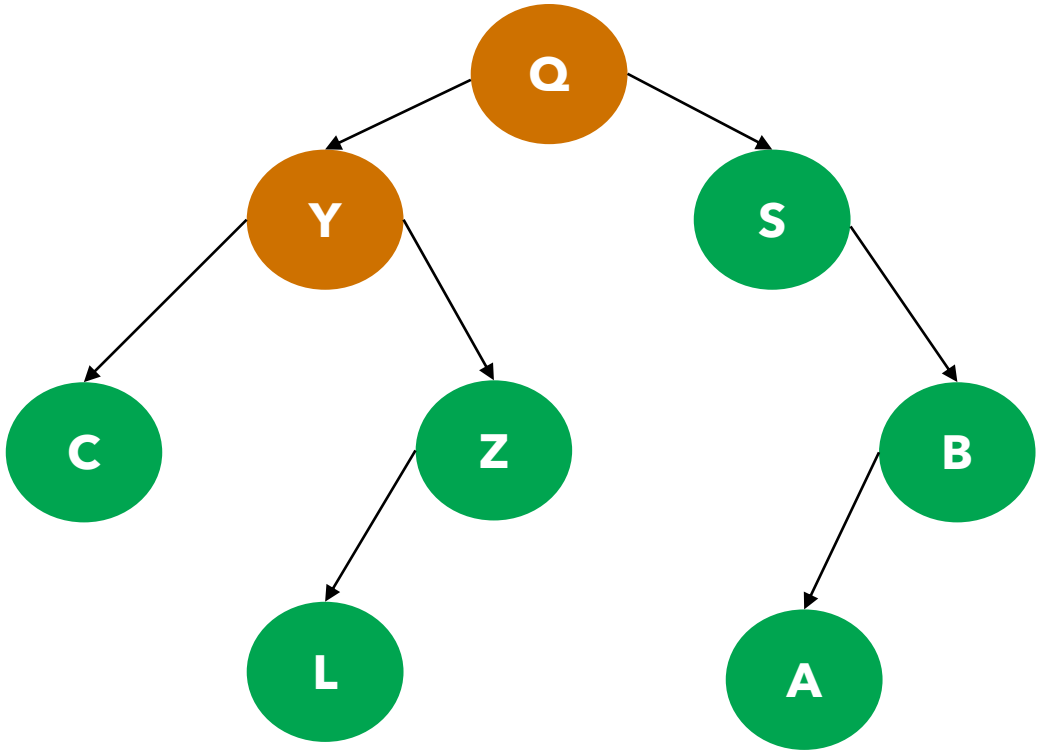
- Nella ricerca in profondità (**depth-first**), per ogni nodo vengono visitati ricorsivamente in profondità prima il figlio sinistro e poi il figlio destro
- Questo significa che prima di *tornare indietro* ad un nodo T (*backtracking*), uno dei due sottoalberi di T deve essere visitato completamente
- 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**

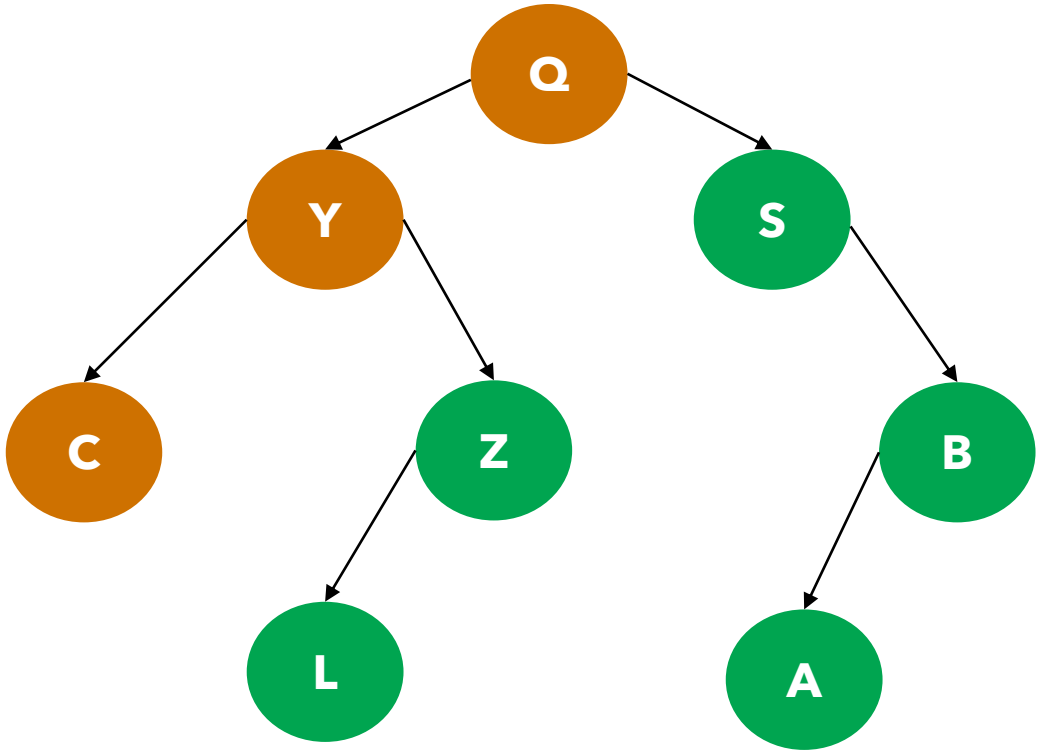
Depth-first search (DFS) su alberi binari



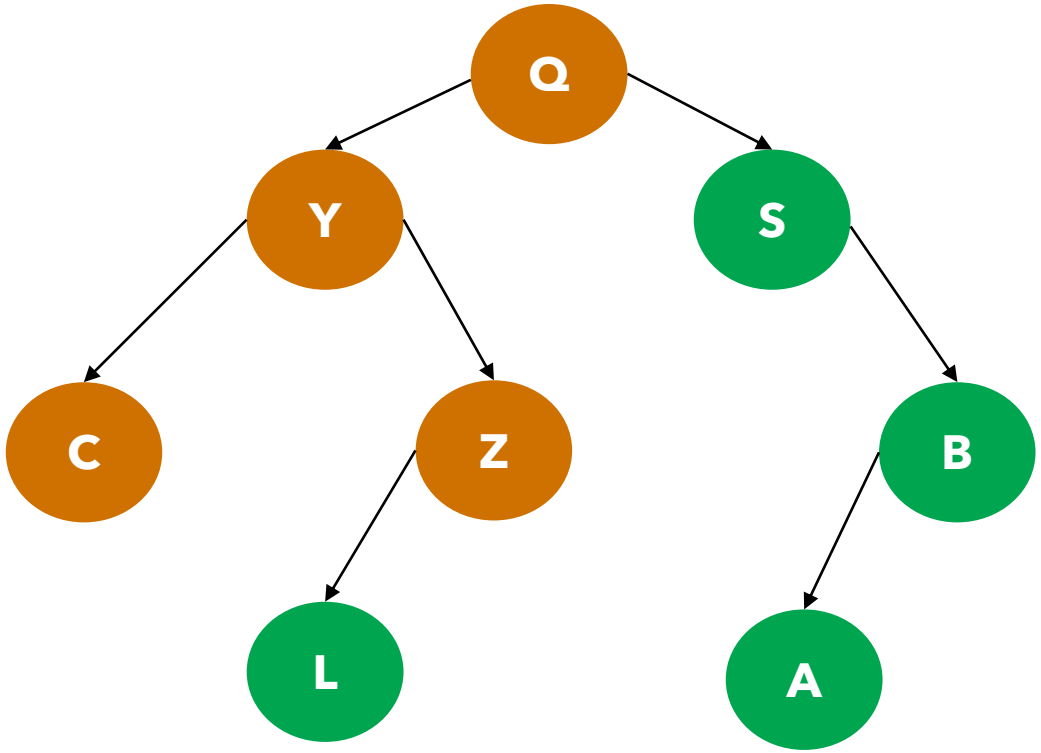
Depth-first search (DFS) su alberi binari



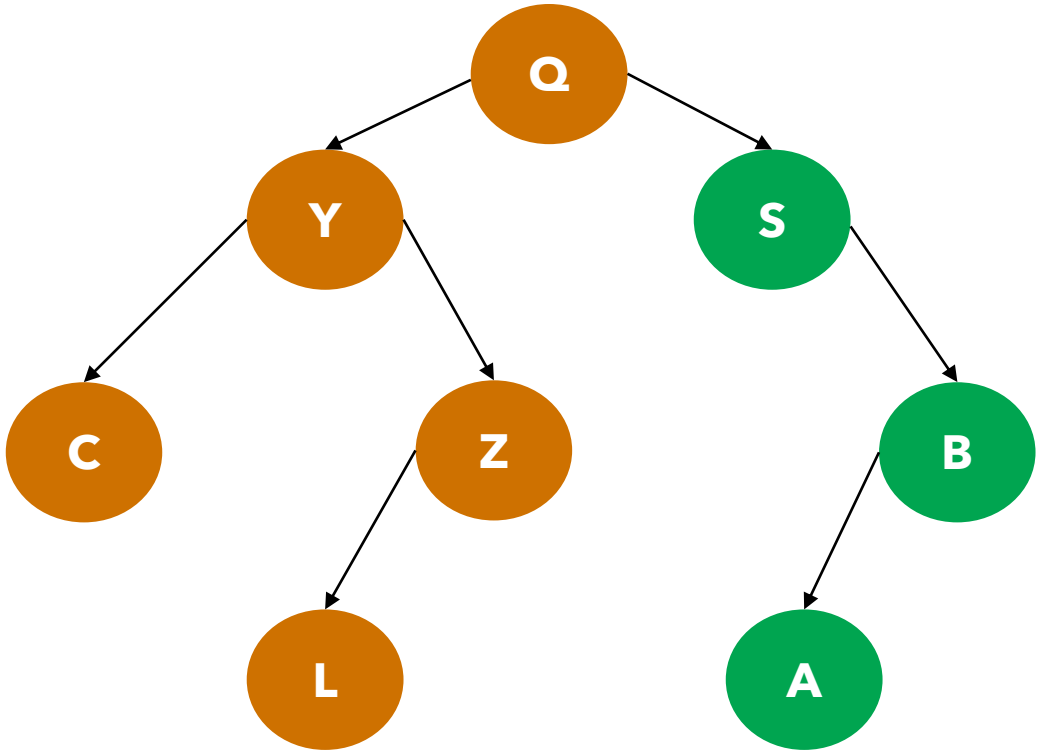
Depth-first search (DFS) su su alberi binari



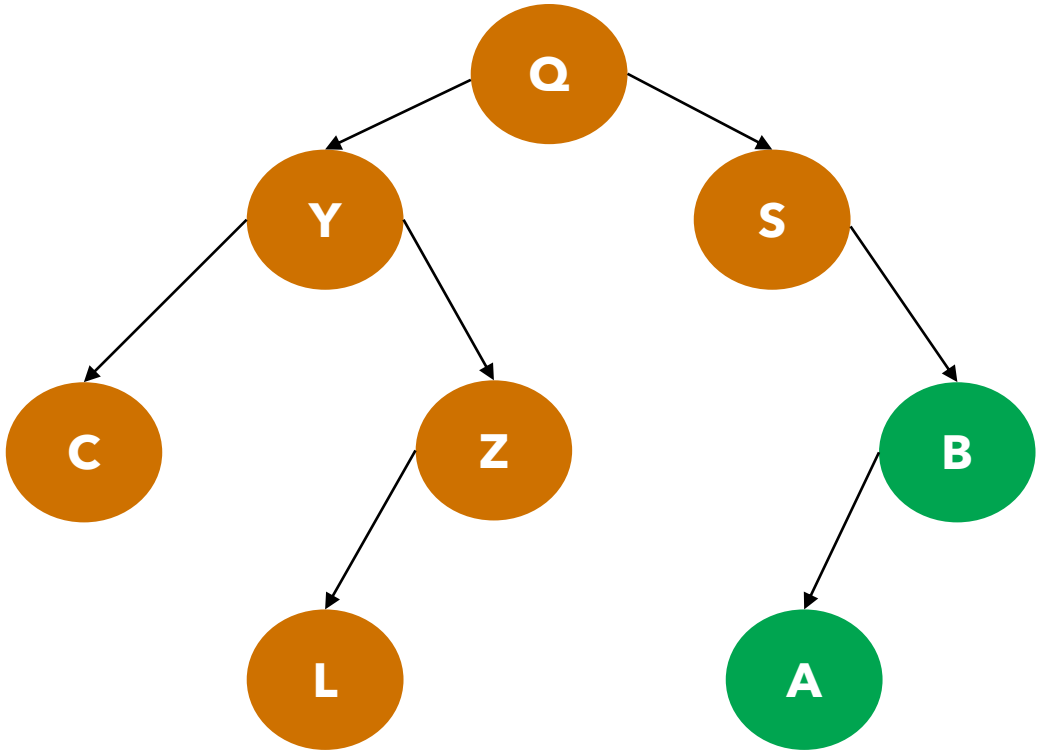
Depth-first search (DFS) su alberi binari



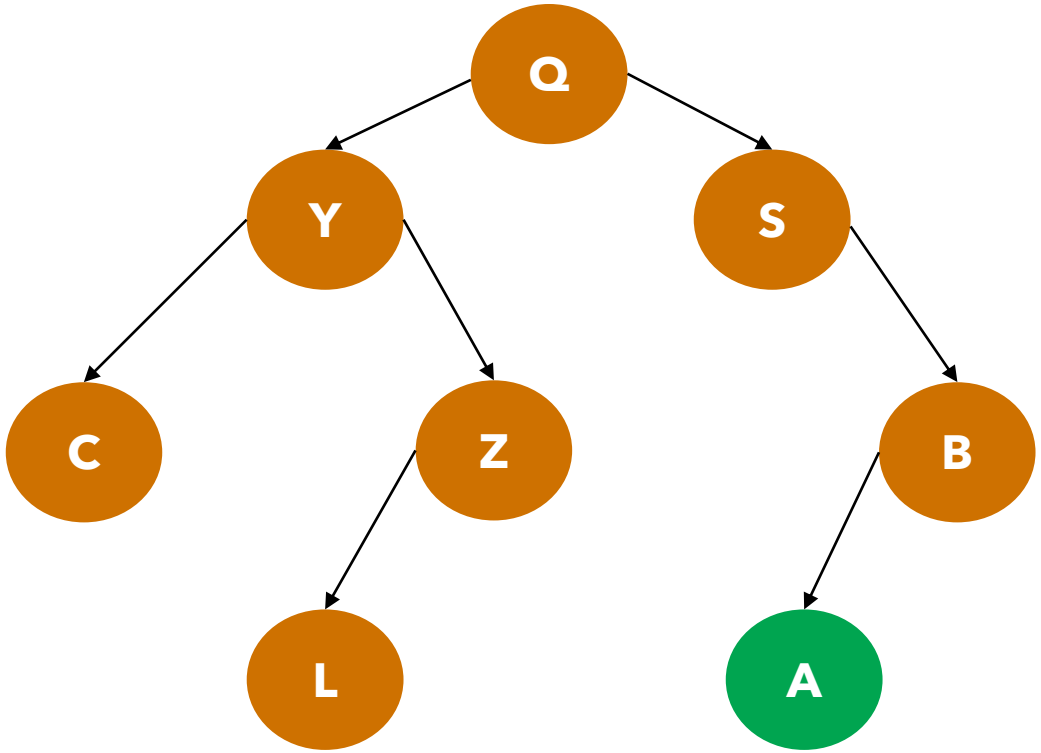
Depth-first search (DFS) su alberi binari



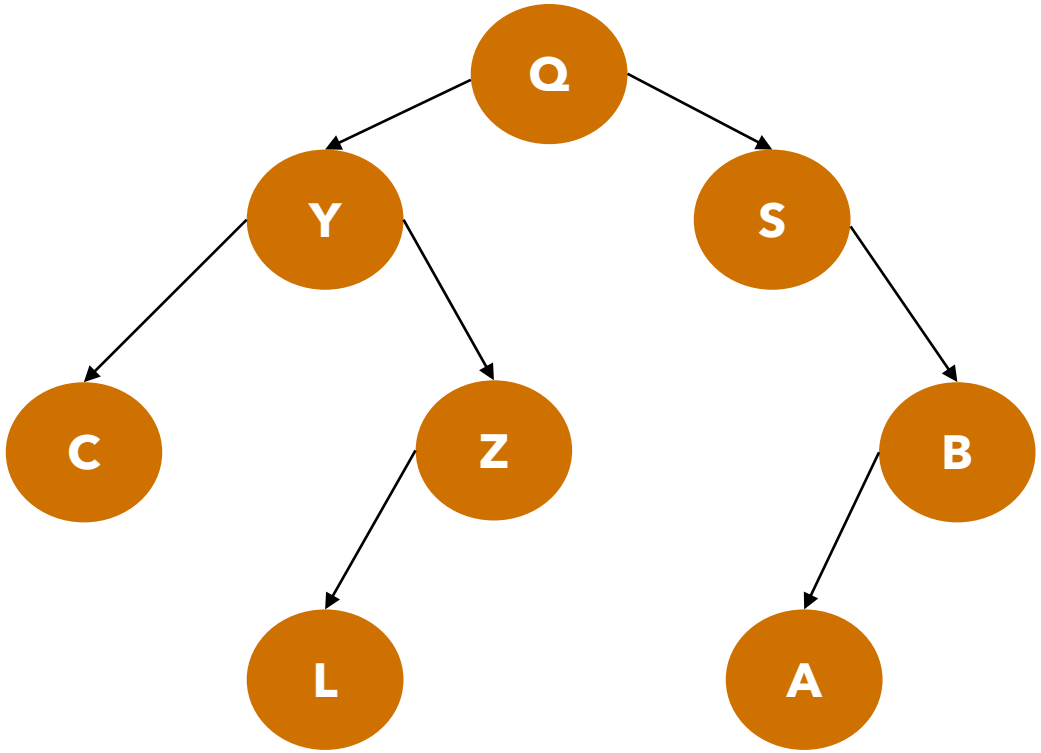
Depth-first search (DFS) su alberi binari



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

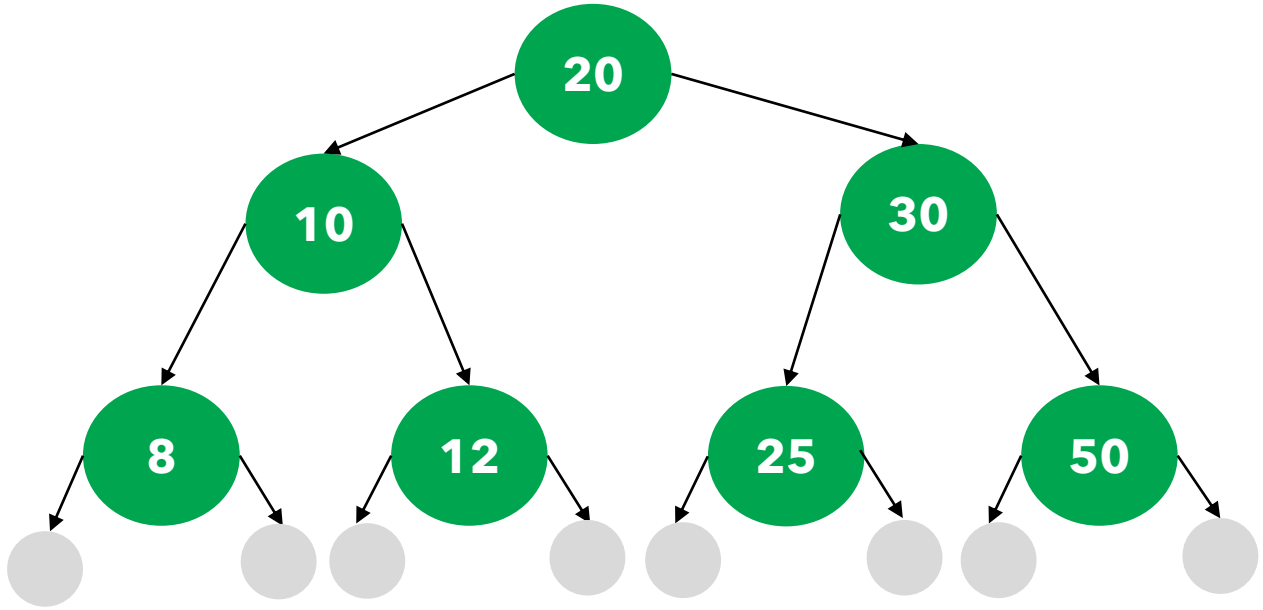
Esempio di DFS: visita inorder di un BST (inorder_tree_walk)

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

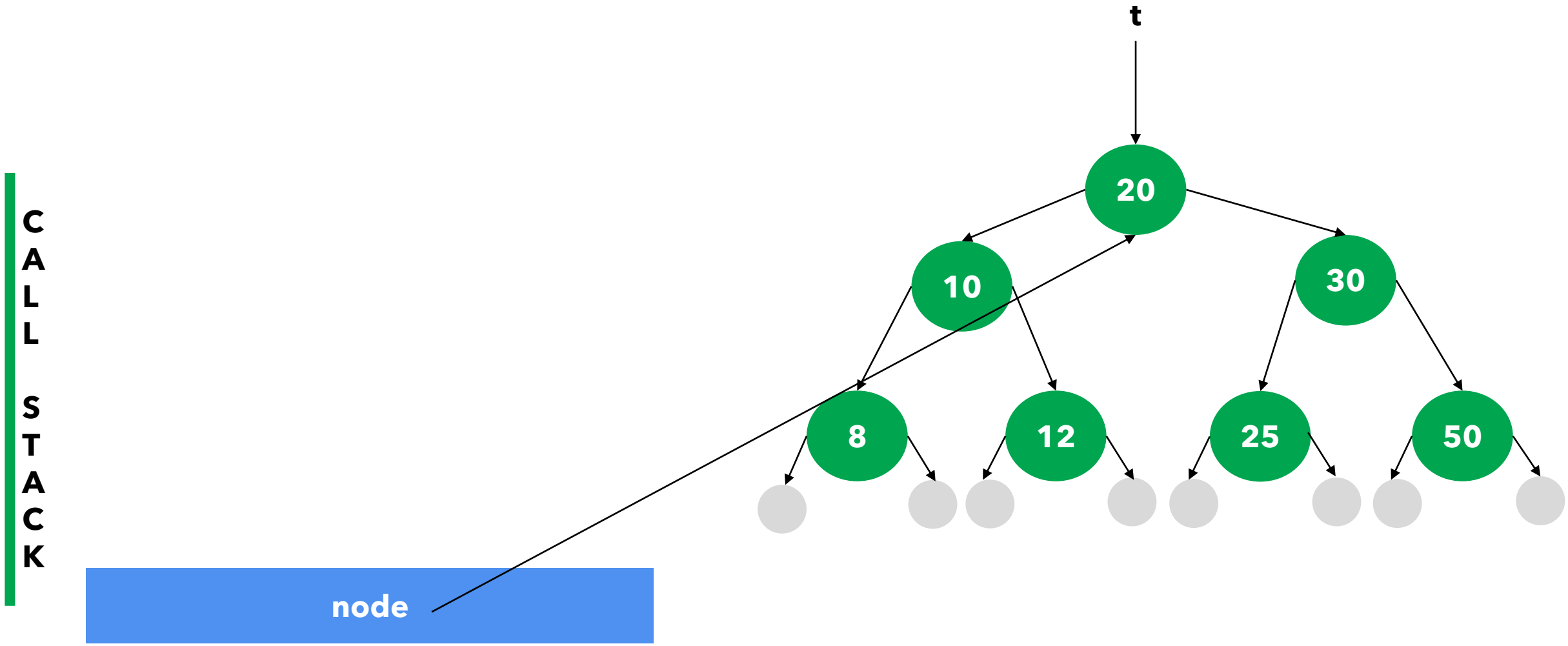
Esempio di DFS: visita inorder di un BST (inorder_tree_walk)

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

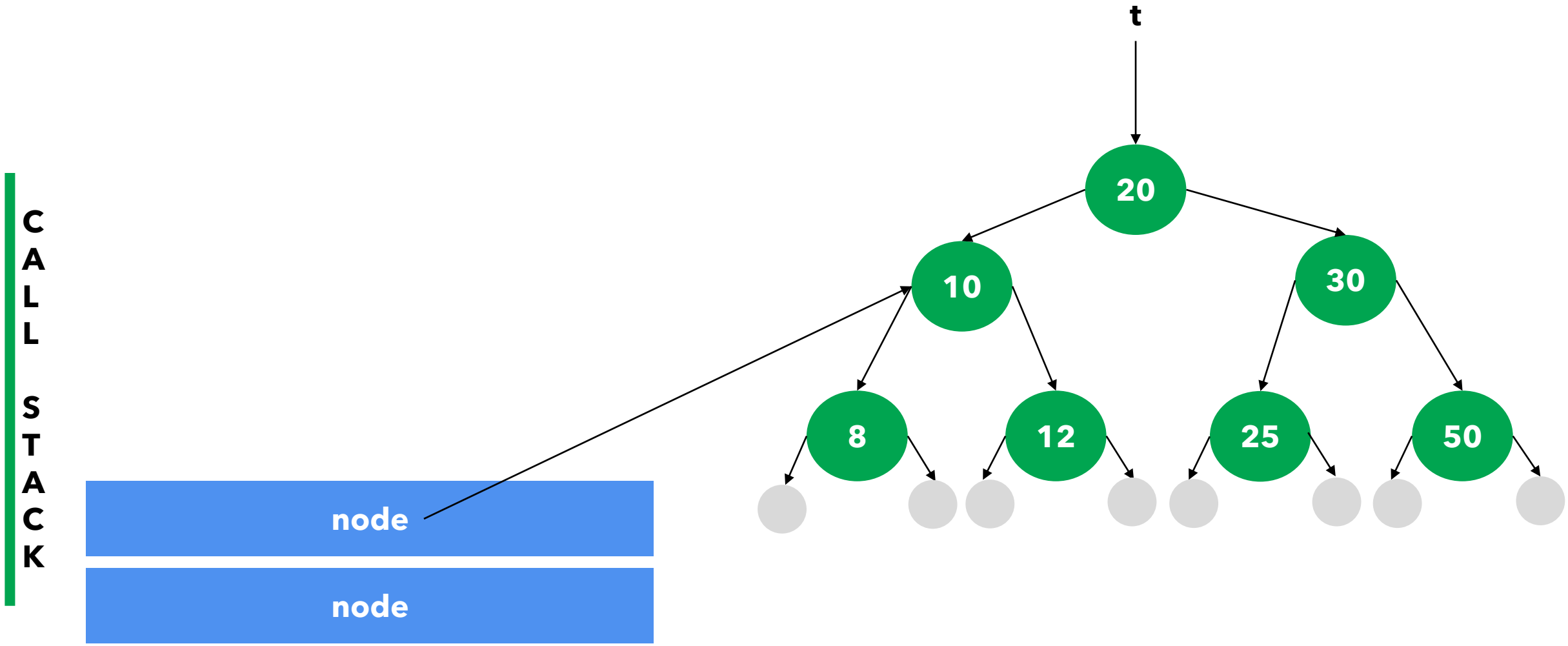


La vista *inorder* di un albero binario di ricerca stampa le chiavi dei nodi in ordine crescente

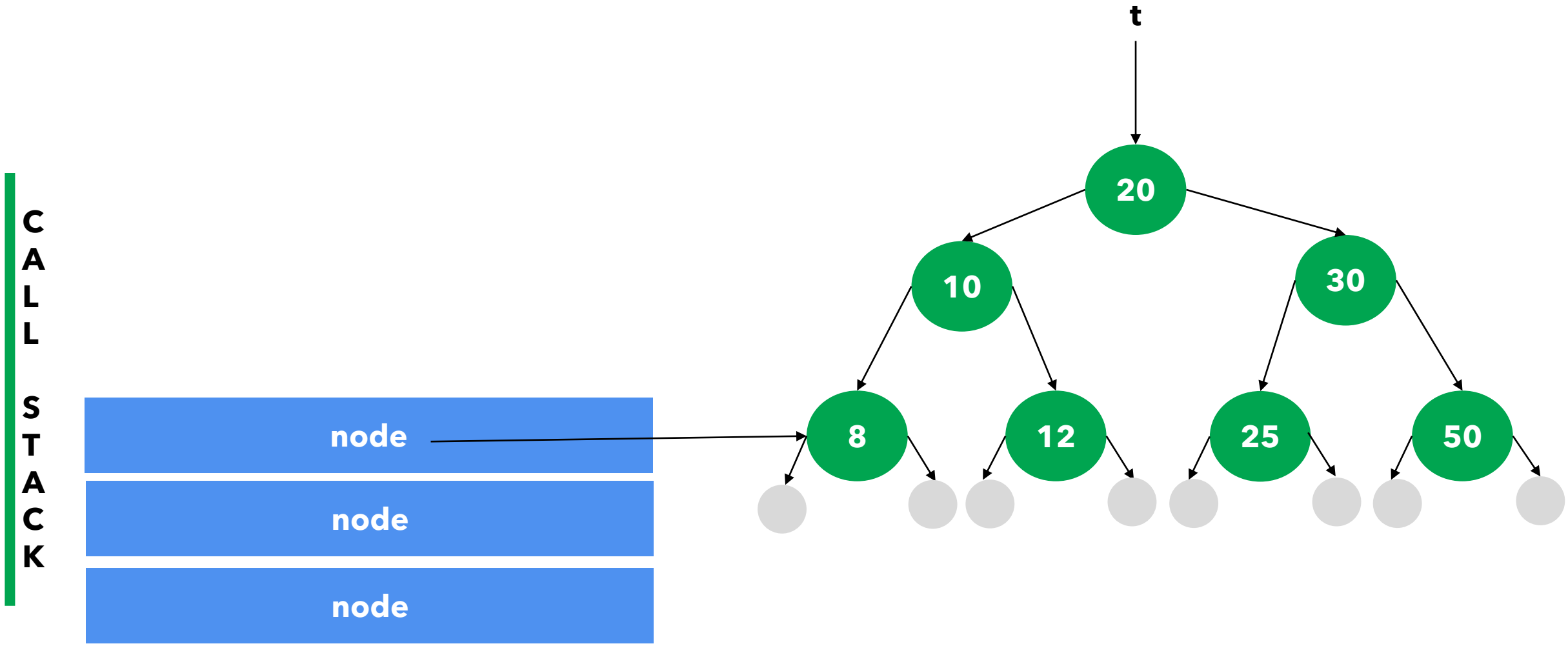
Esempio di DFS: visita inorder di un BST (inorder_tree_walk)



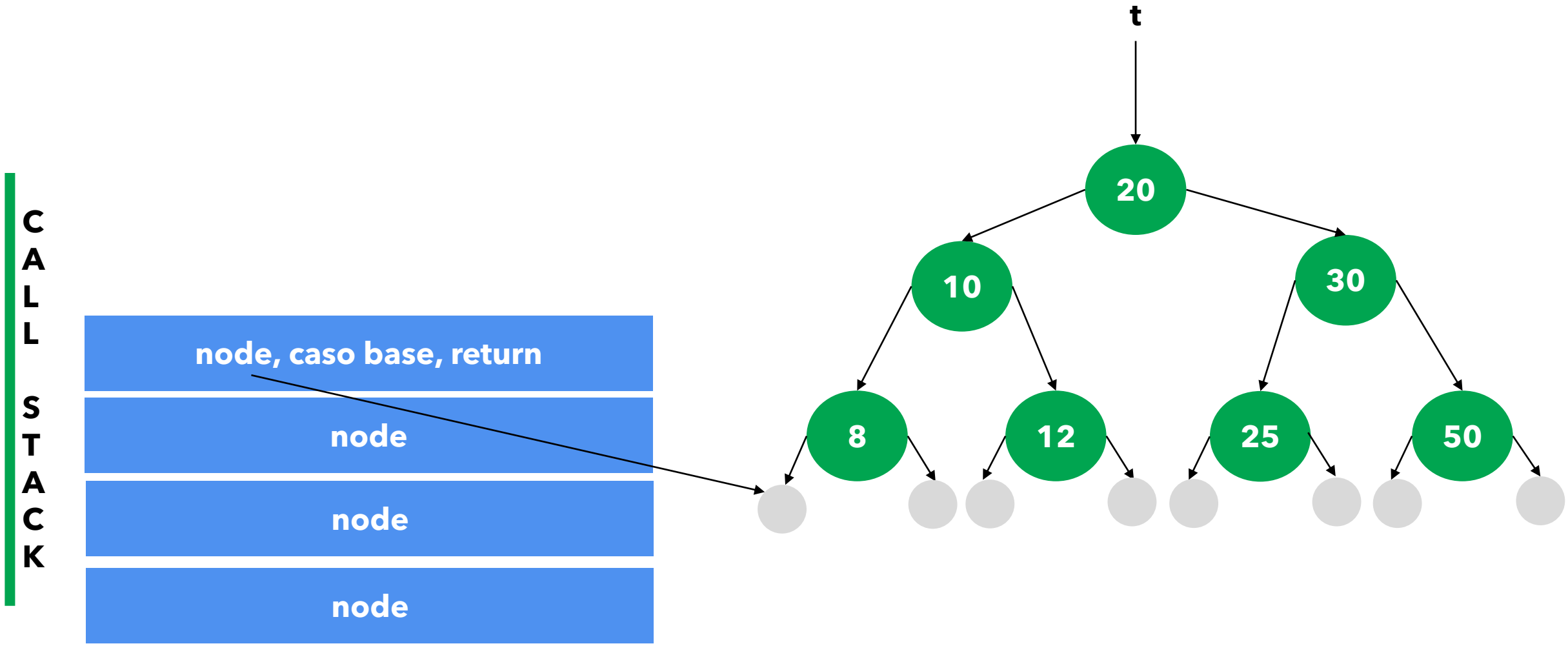
Esempio di DFS: visita inorder di un BST (inorder_tree_walk)



Esempio di DFS: visita inorder di un BST (inorder_tree_walk)

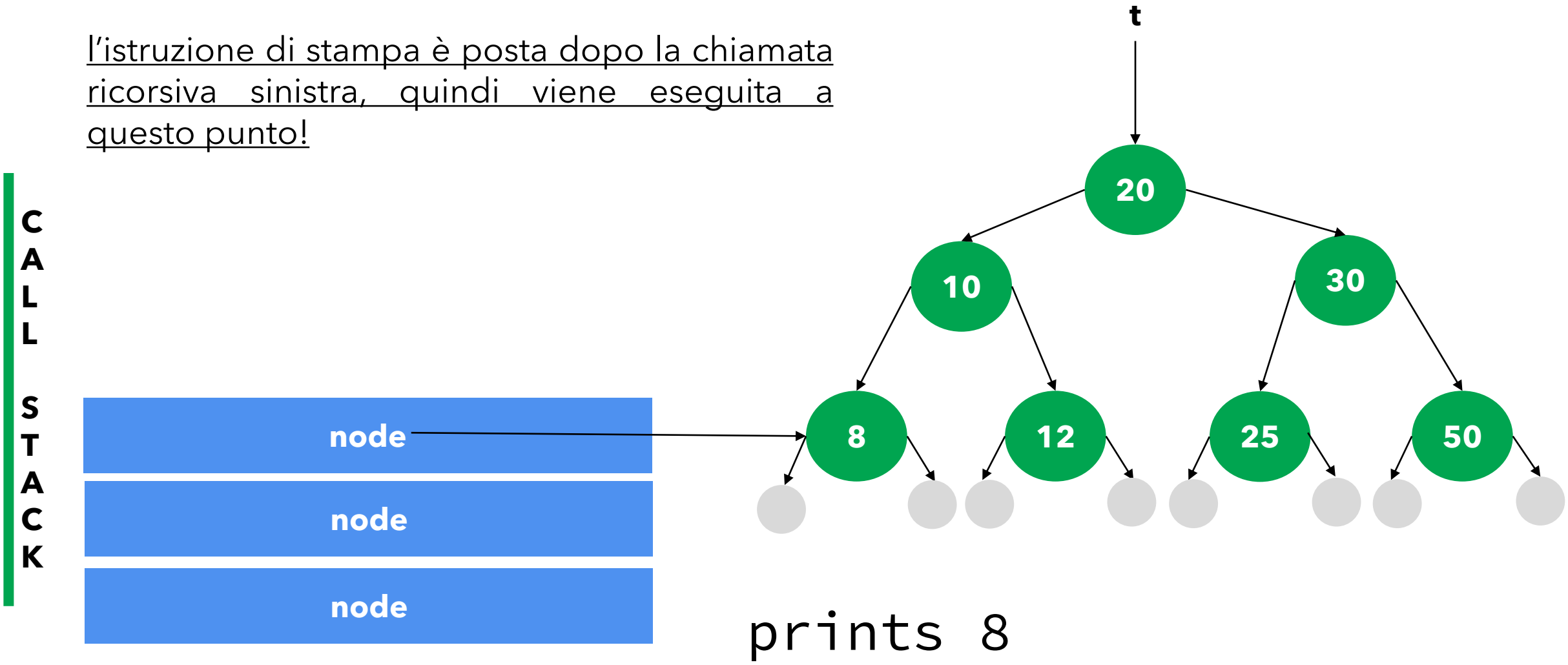


Esempio di DFS: visita inorder di un BST (inorder_tree_walk)



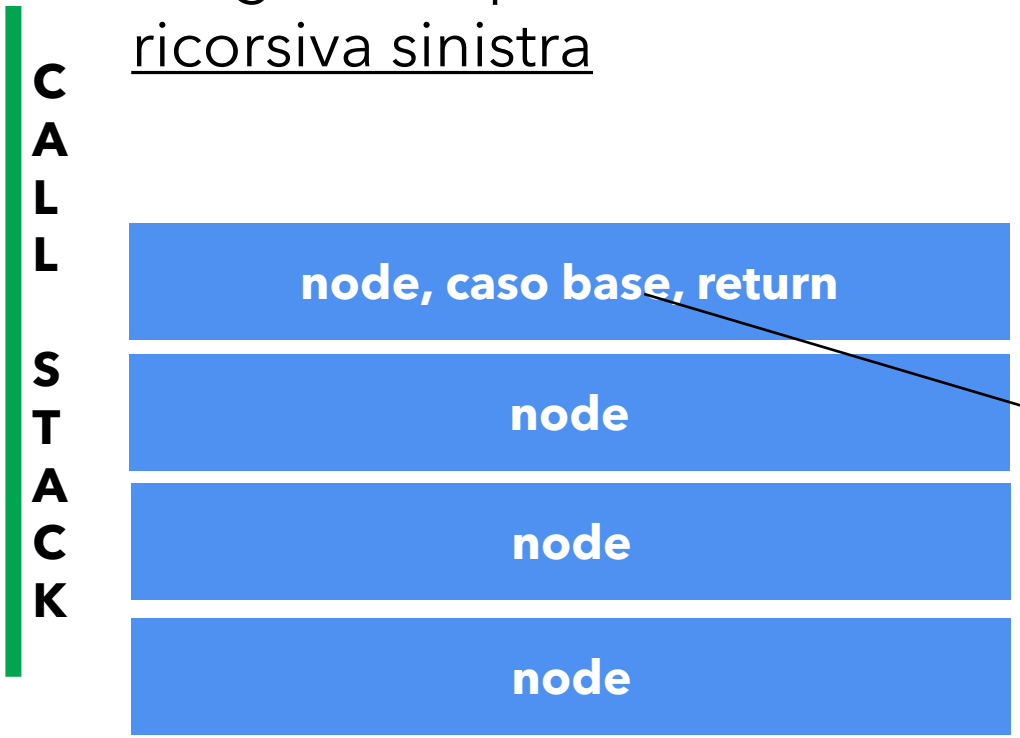
Esempio di DFS: visita inorder di un BST (inorder_tree_walk)

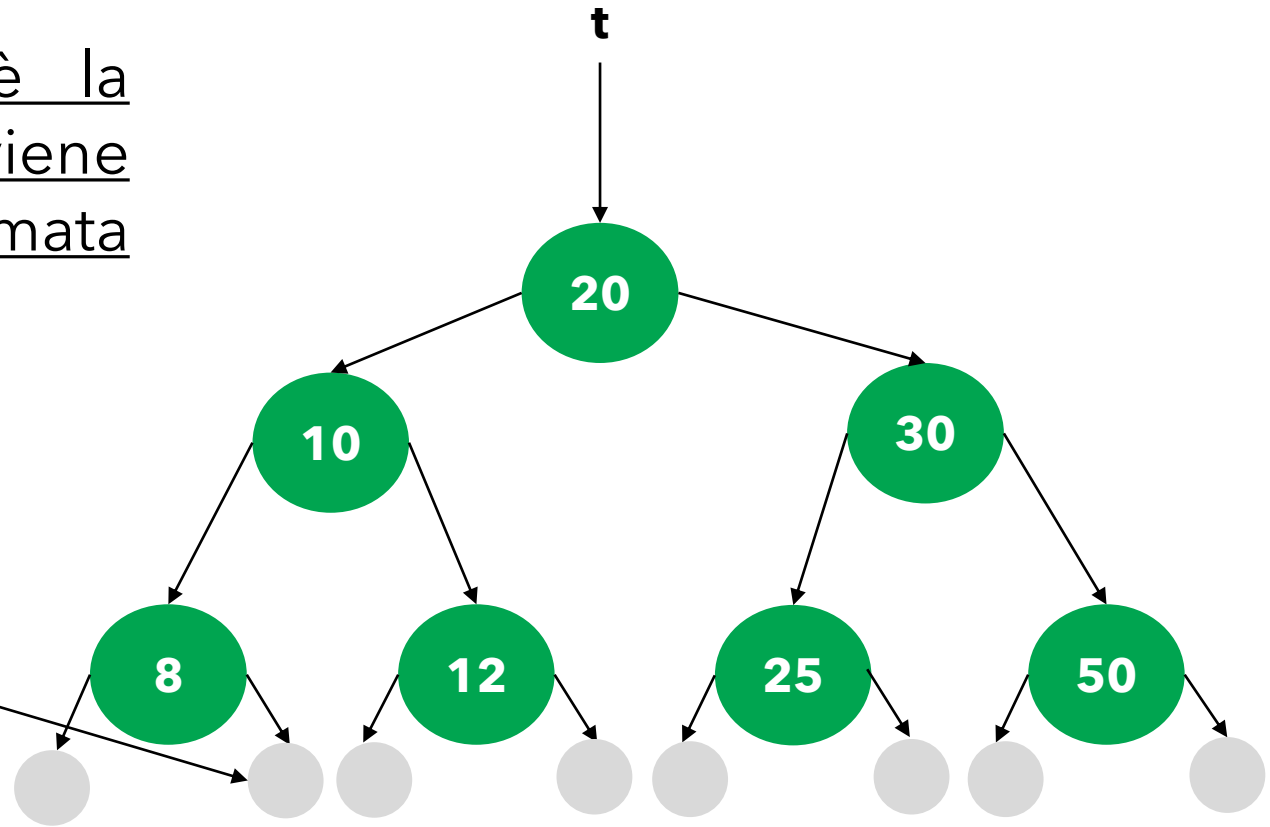
l'istruzione di stampa è posta dopo la chiamata ricorsiva sinistra, quindi viene eseguita a questo punto!



Esempio di DFS: visita `inorder` di un BST (`inorder_tree_walk`)

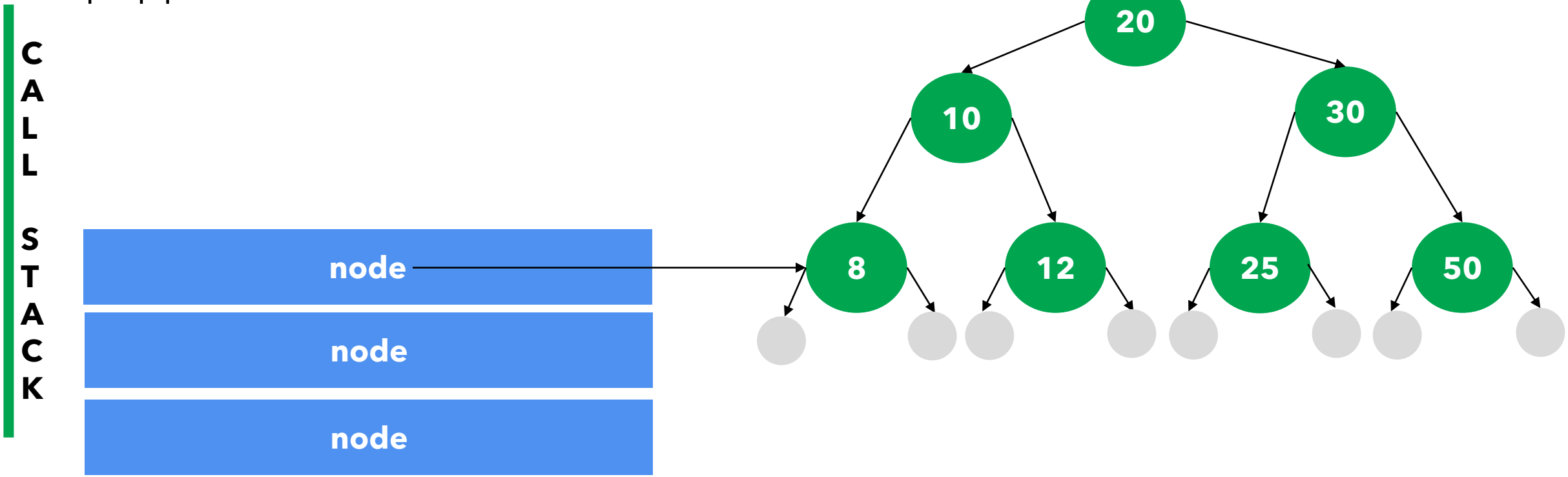
dopo l'istruzione di stampa c'è la chiamata ricorsiva destra, quindi viene eseguita dopo il return della chiamata ricorsiva sinistra





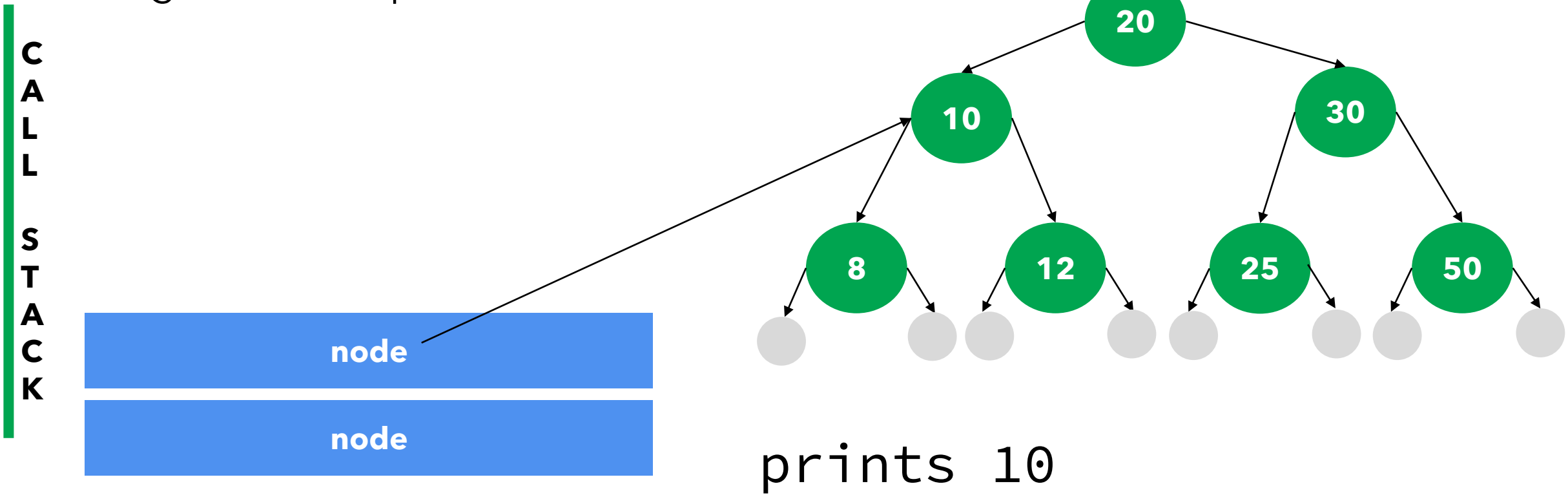
Esempio di DFS: visita `inorder` di un BST (`inorder_tree_walk`)

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 (inorder_tree_walk)

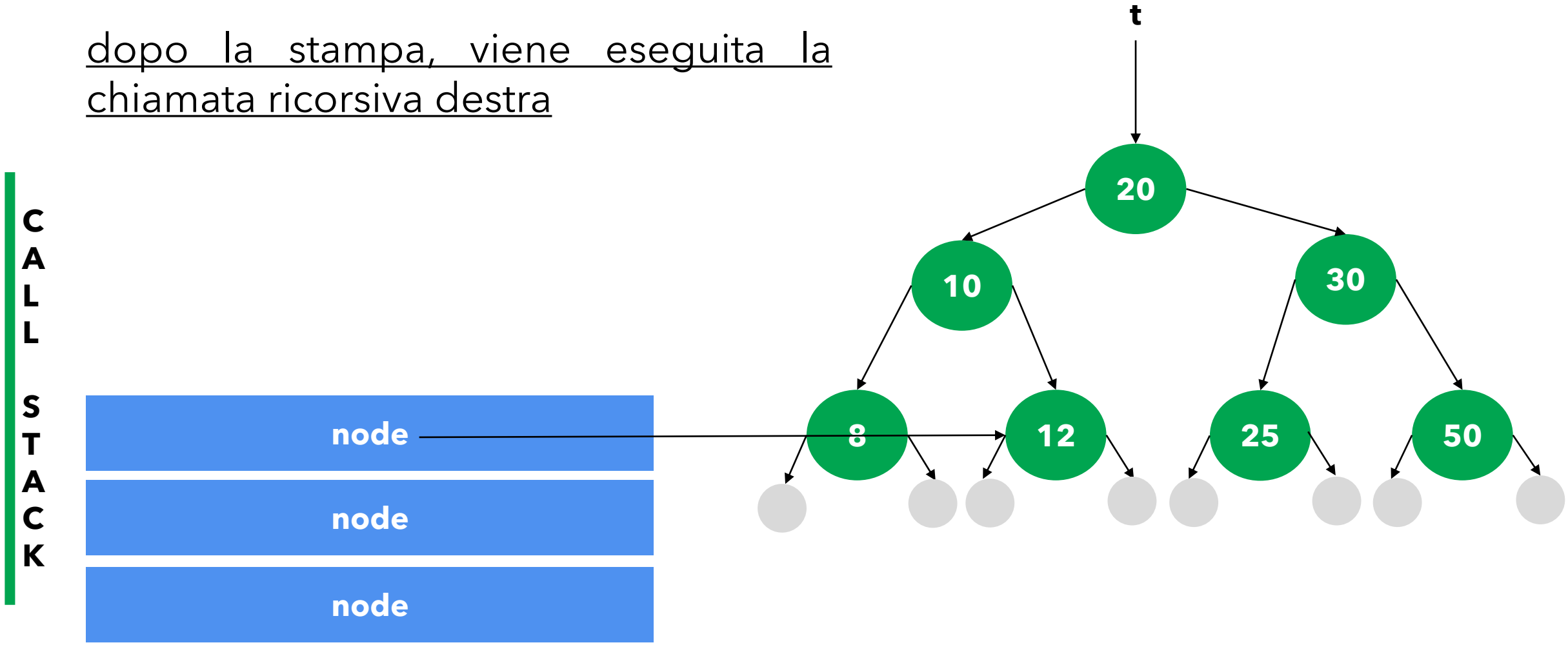
qui siamo appena tornati da una chiamata ricorsiva sinistra, quindi viene eseguita la stampa



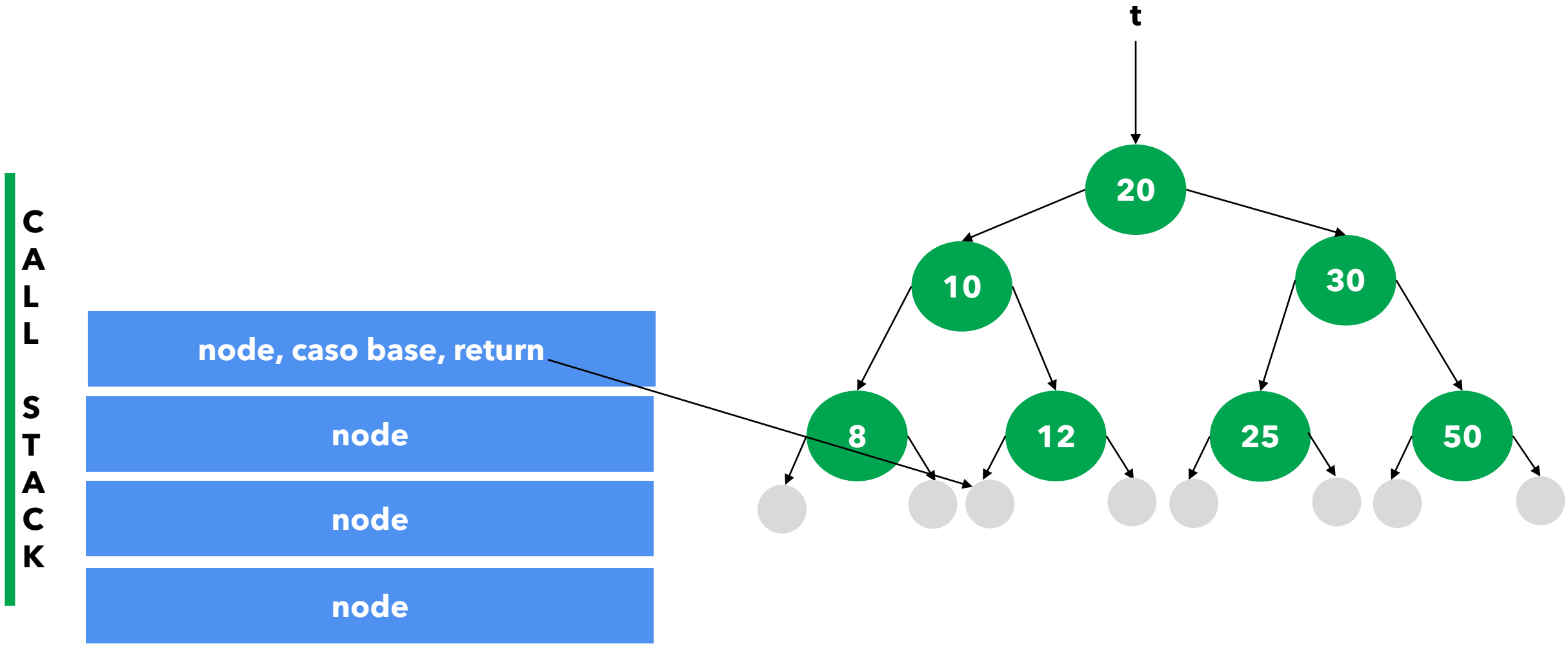
prints 10

Esempio di DFS: visita `inorder` di un BST (`inorder_tree_walk`)

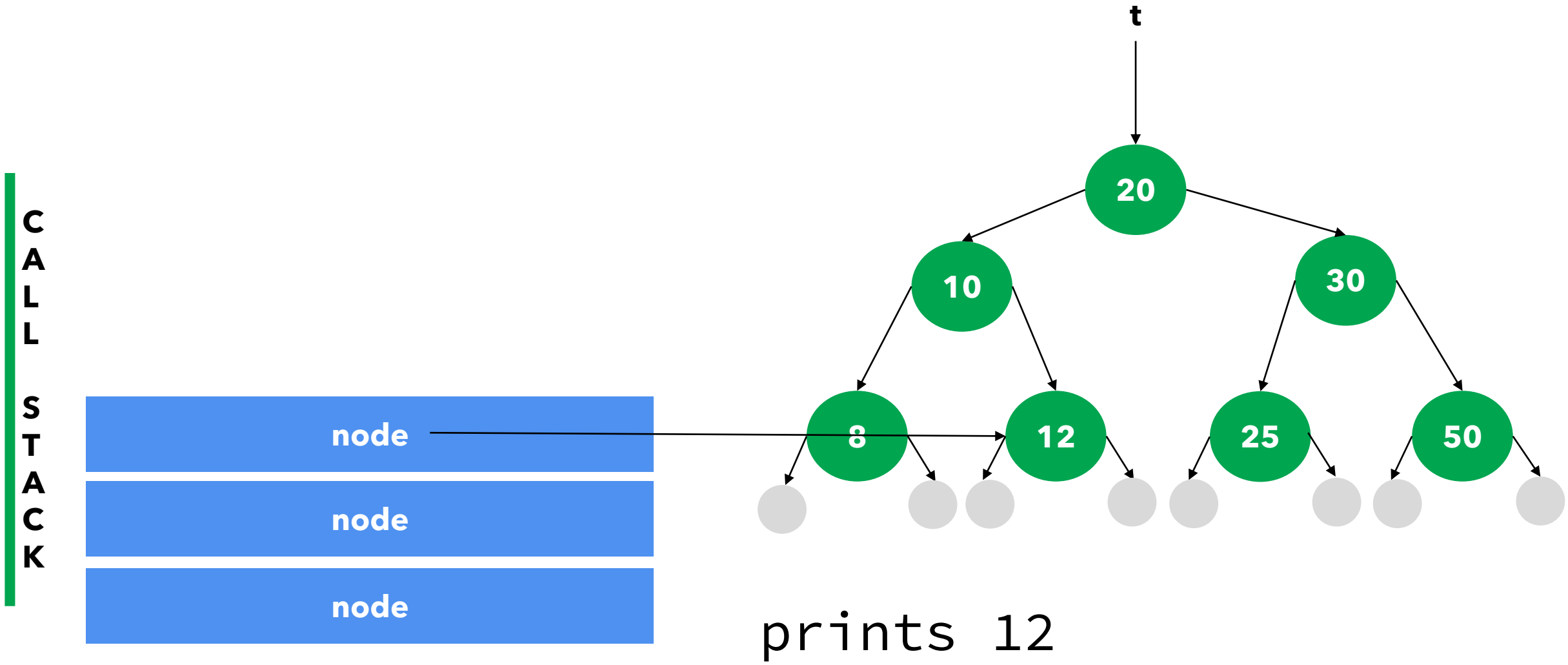
dopo la stampa, viene eseguita la chiamata ricorsiva destra



Esempio di DFS: visita inorder di un BST (inorder_tree_walk)

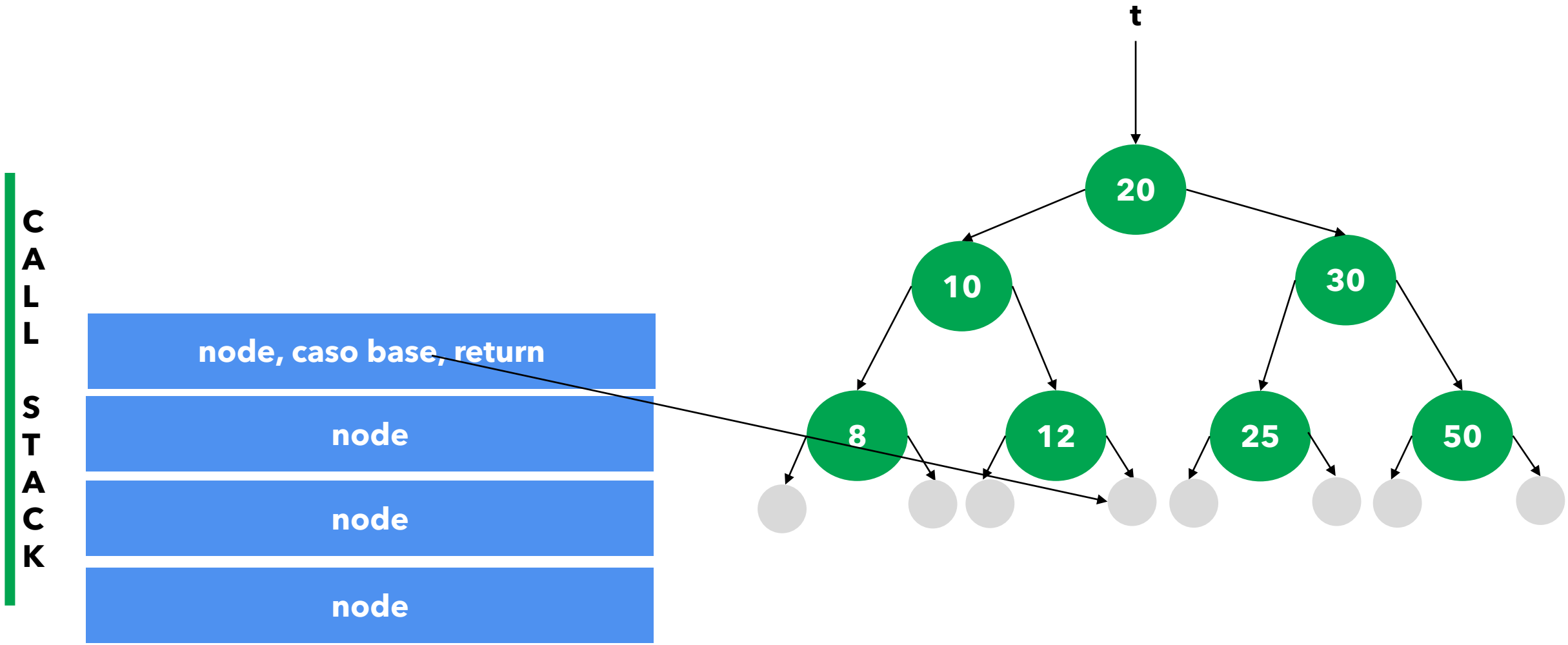


Esempio di DFS: visita inorder di un BST (inorder_tree_walk)

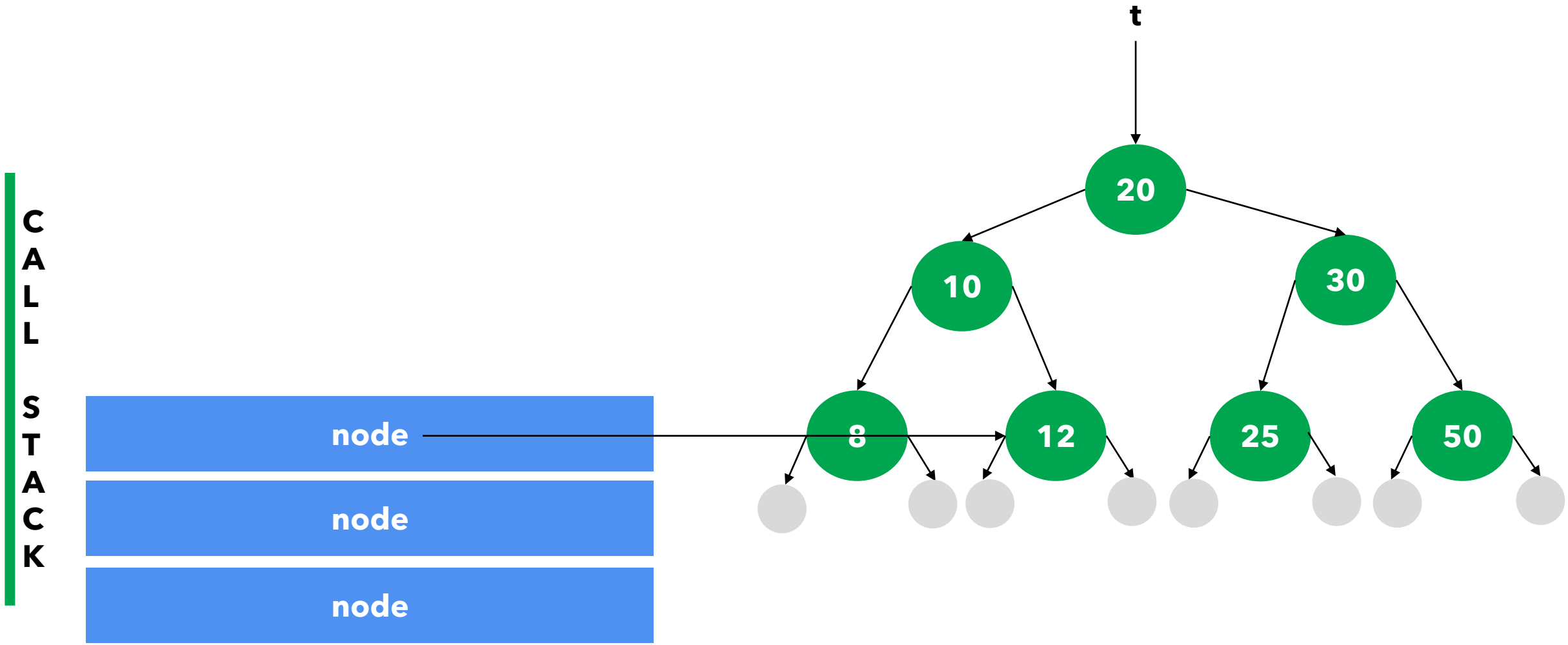


prints 12

Esempio di DFS: visita inorder di un BST (inorder_tree_walk)

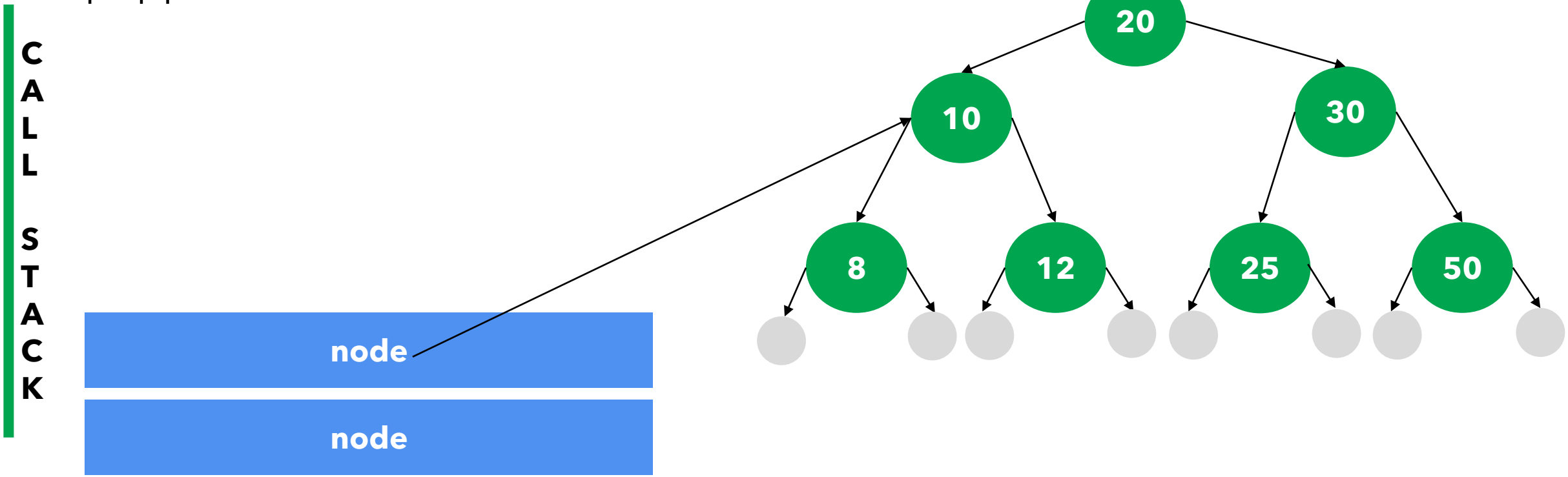


Esempio di DFS: visita inorder di un BST (inorder_tree_walk)

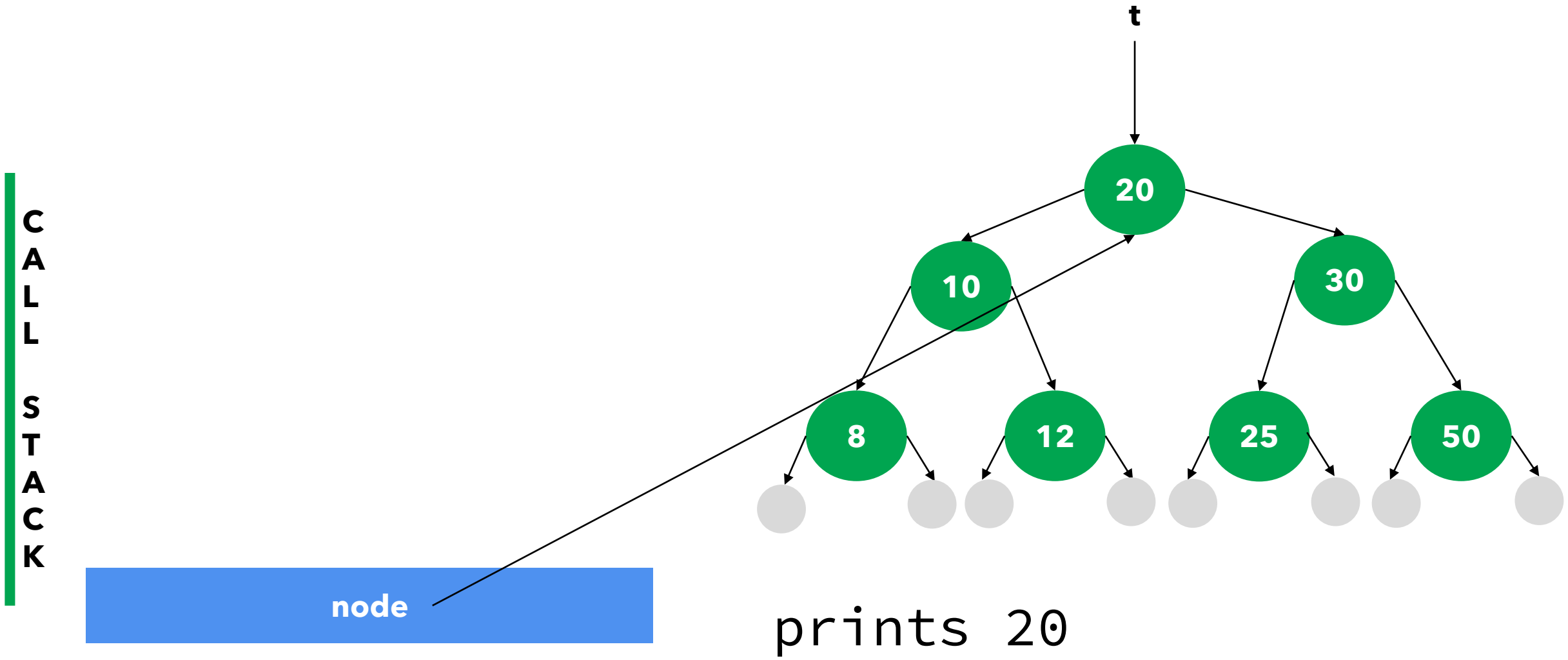


Esempio di DFS: visita `inorder` di un BST (`inorder_tree_walk`)

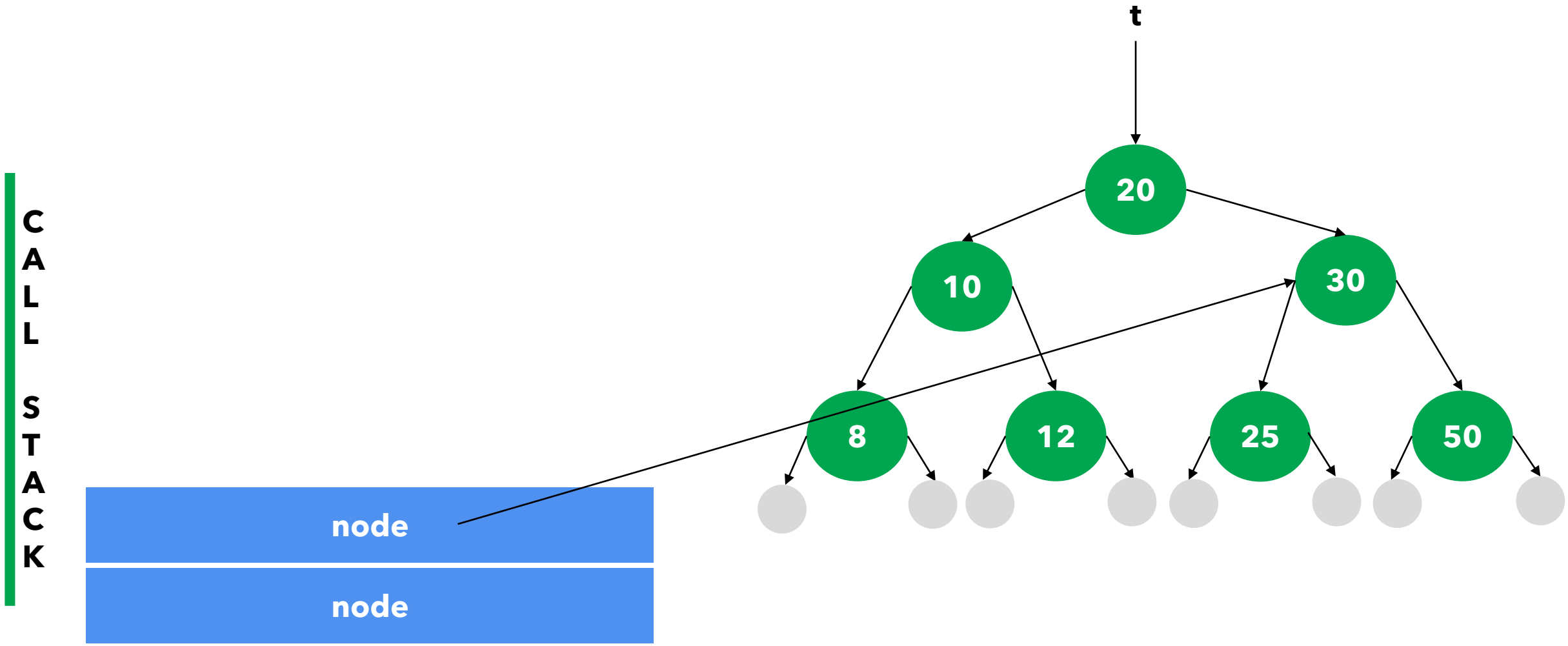
dopo la chiamata ricorsiva destra non c'è niente, quindi il frame viene poppato e non viene fatto altro



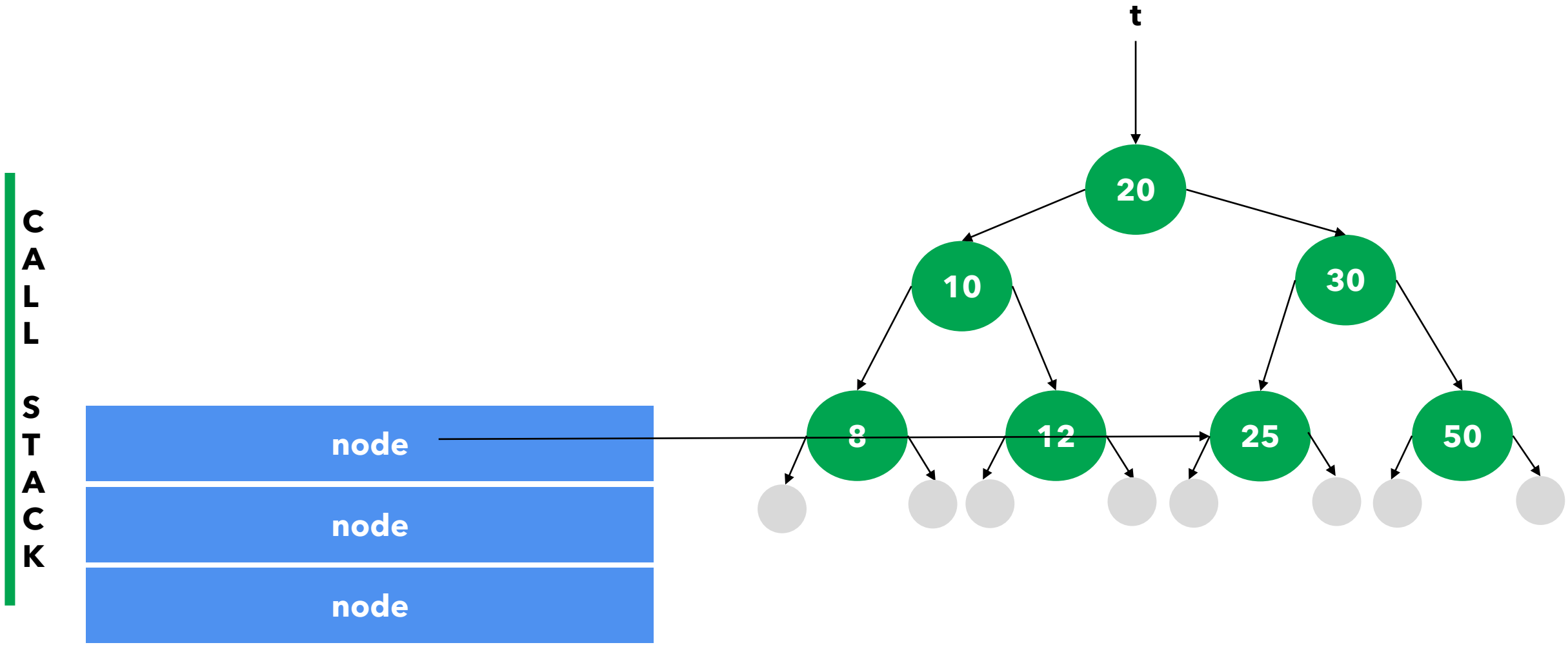
Esempio di DFS: visita inorder di un BST (inorder_tree_walk)



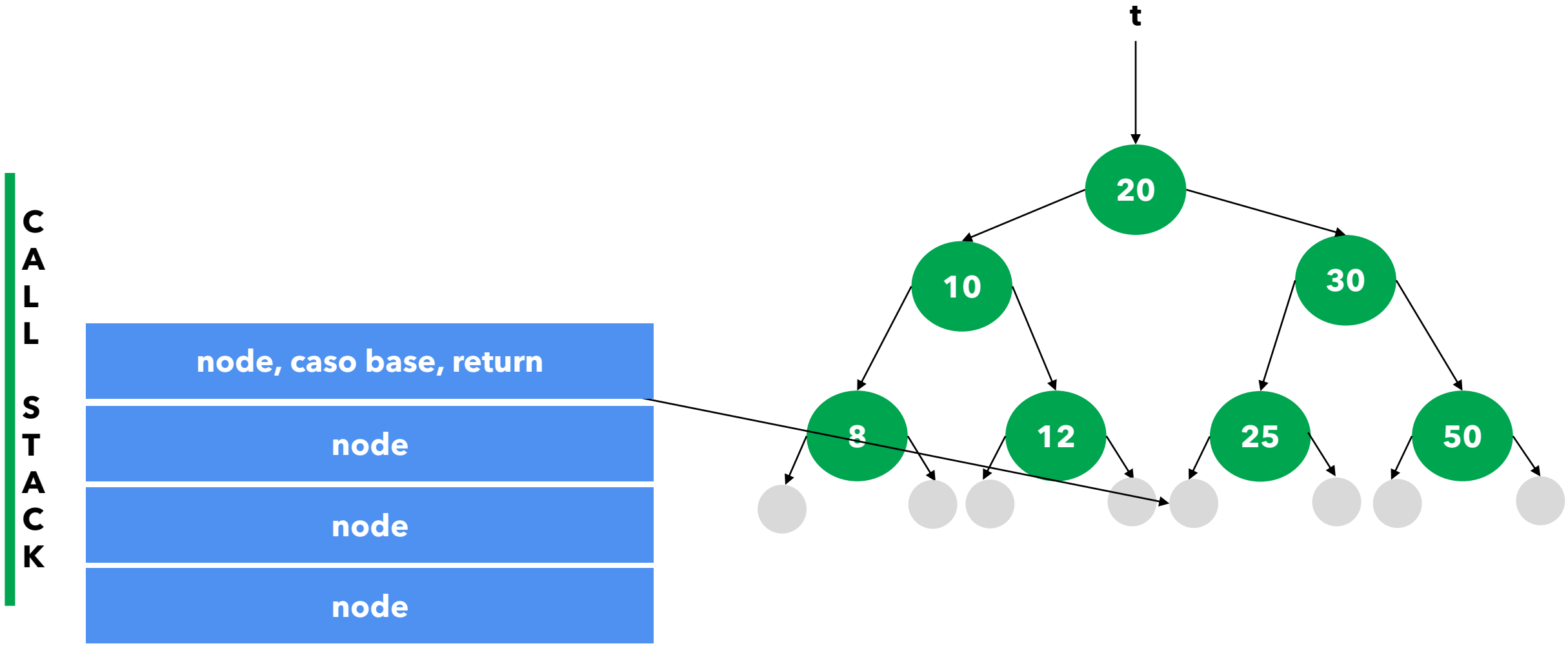
Esempio di DFS: visita inorder di un BST (inorder_tree_walk)



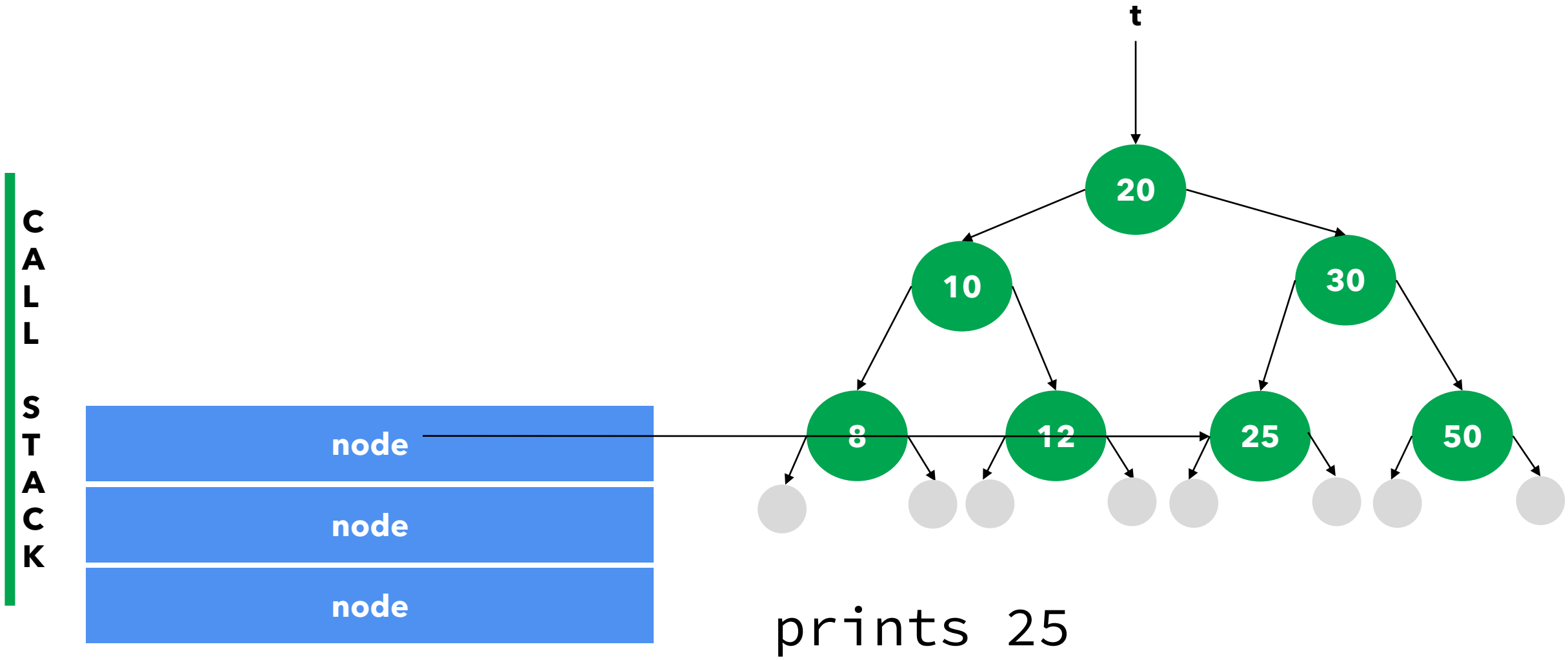
Esempio di DFS: visita inorder di un BST (inorder_tree_walk)



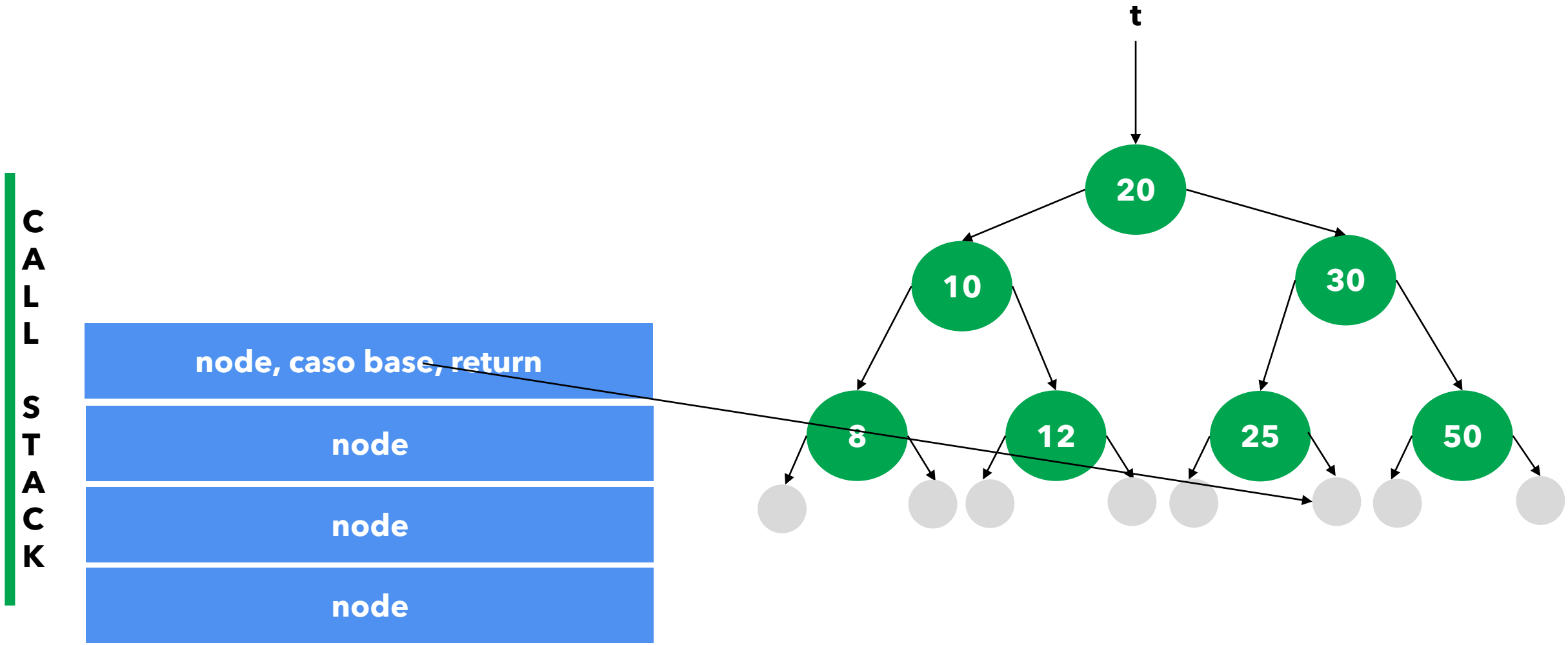
Esempio di DFS: visita `inorder` di un BST (`inorder_tree_walk`)



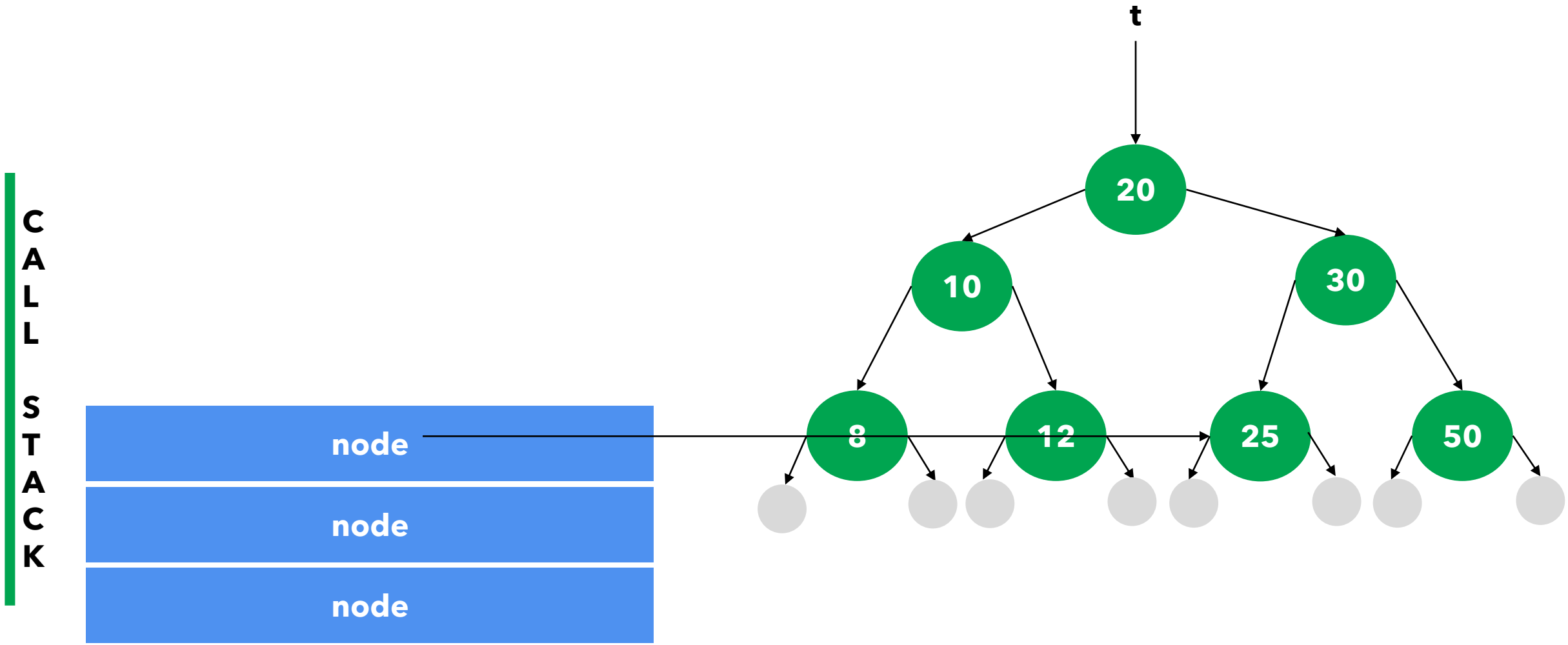
Esempio di DFS: visita inorder di un BST (inorder_tree_walk)



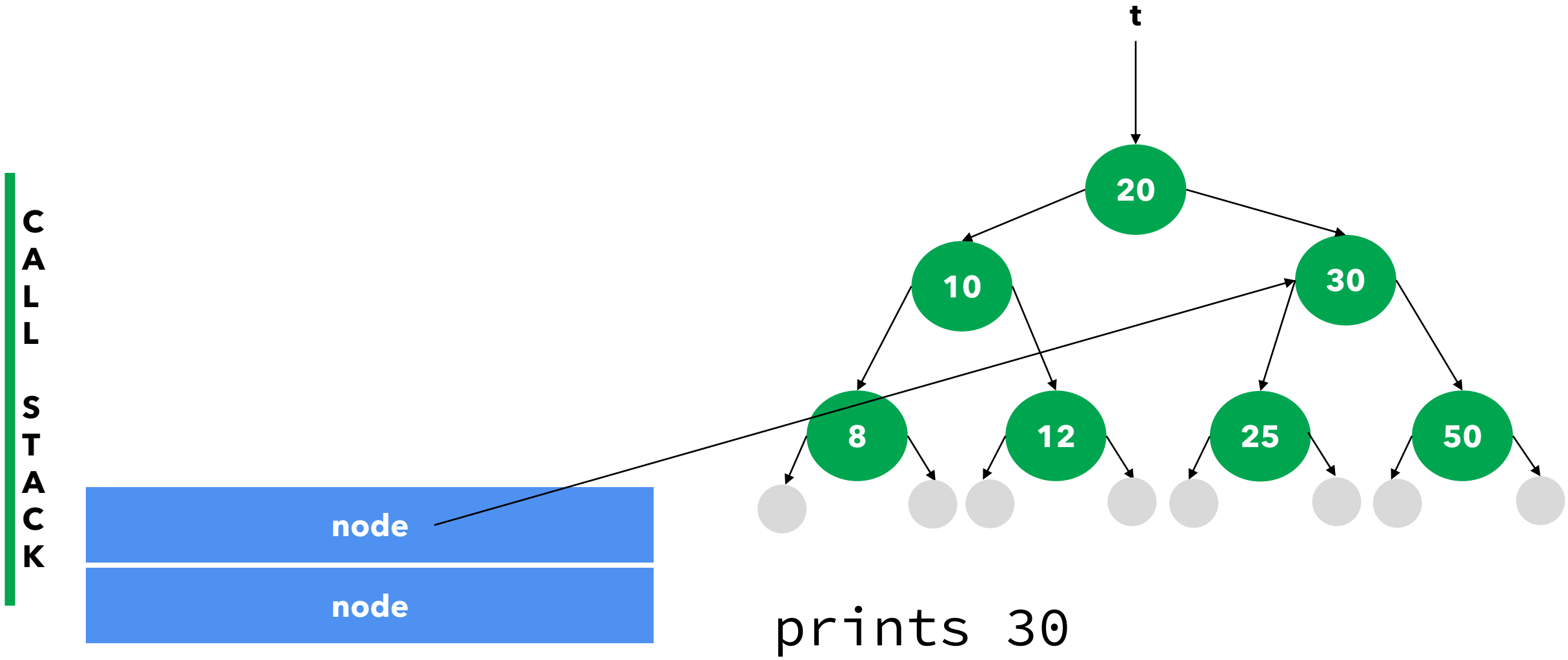
Esempio di DFS: visita inorder di un BST (inorder_tree_walk)



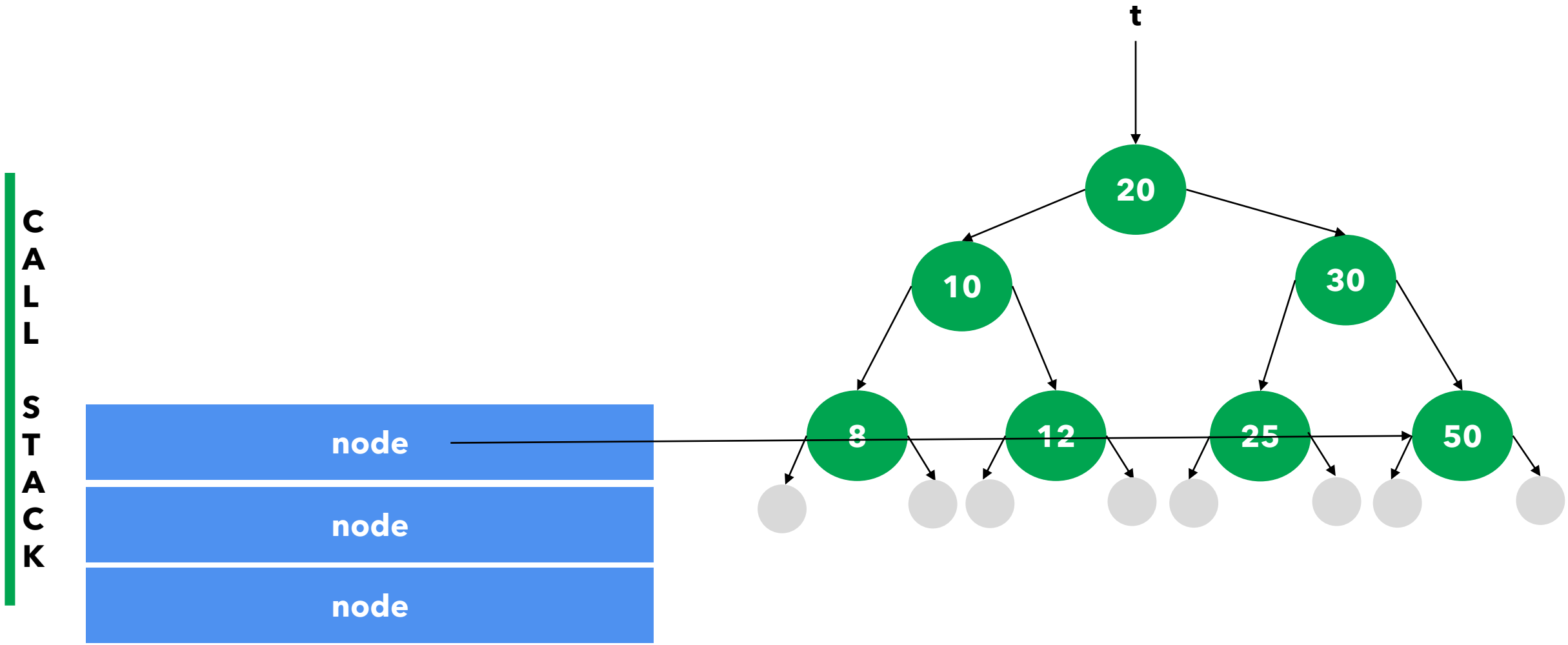
Esempio di DFS: visita inorder di un BST (inorder_tree_walk)



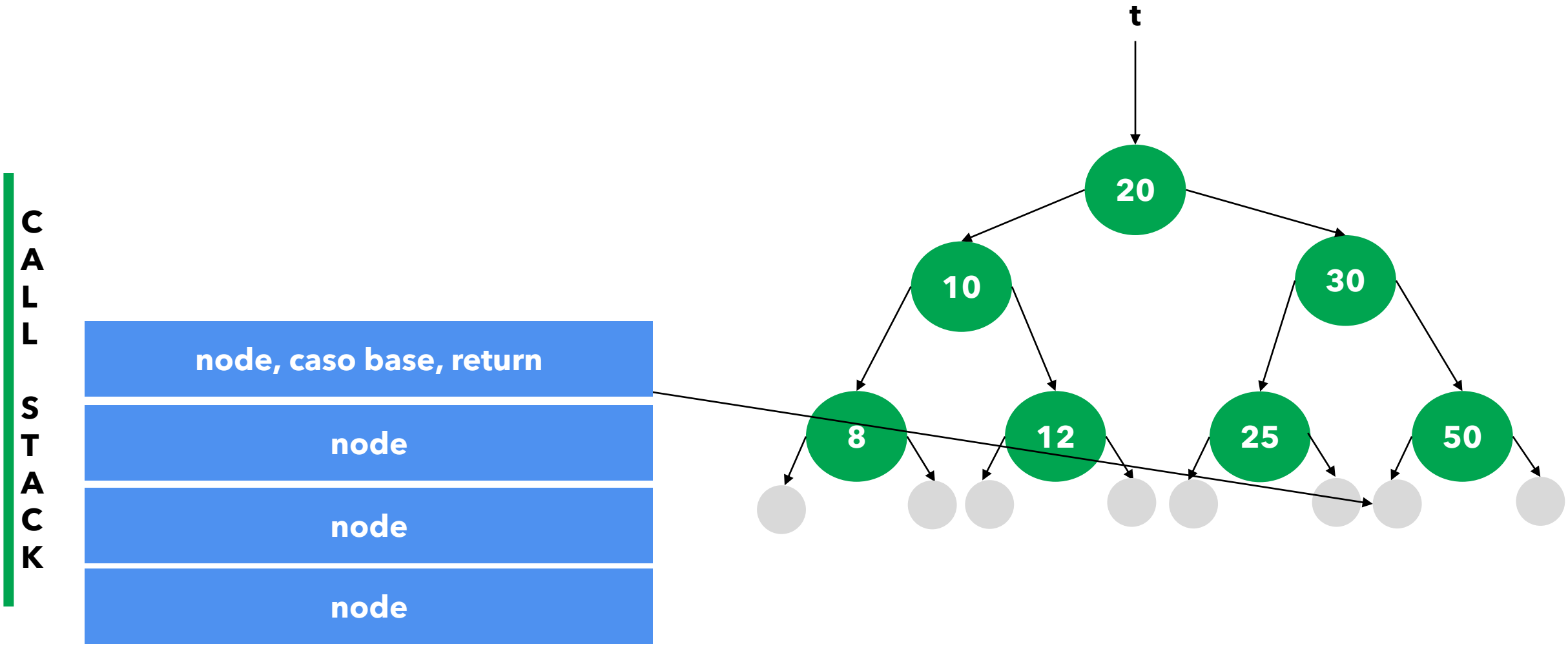
Esempio di DFS: visita inorder di un BST (inorder_tree_walk)



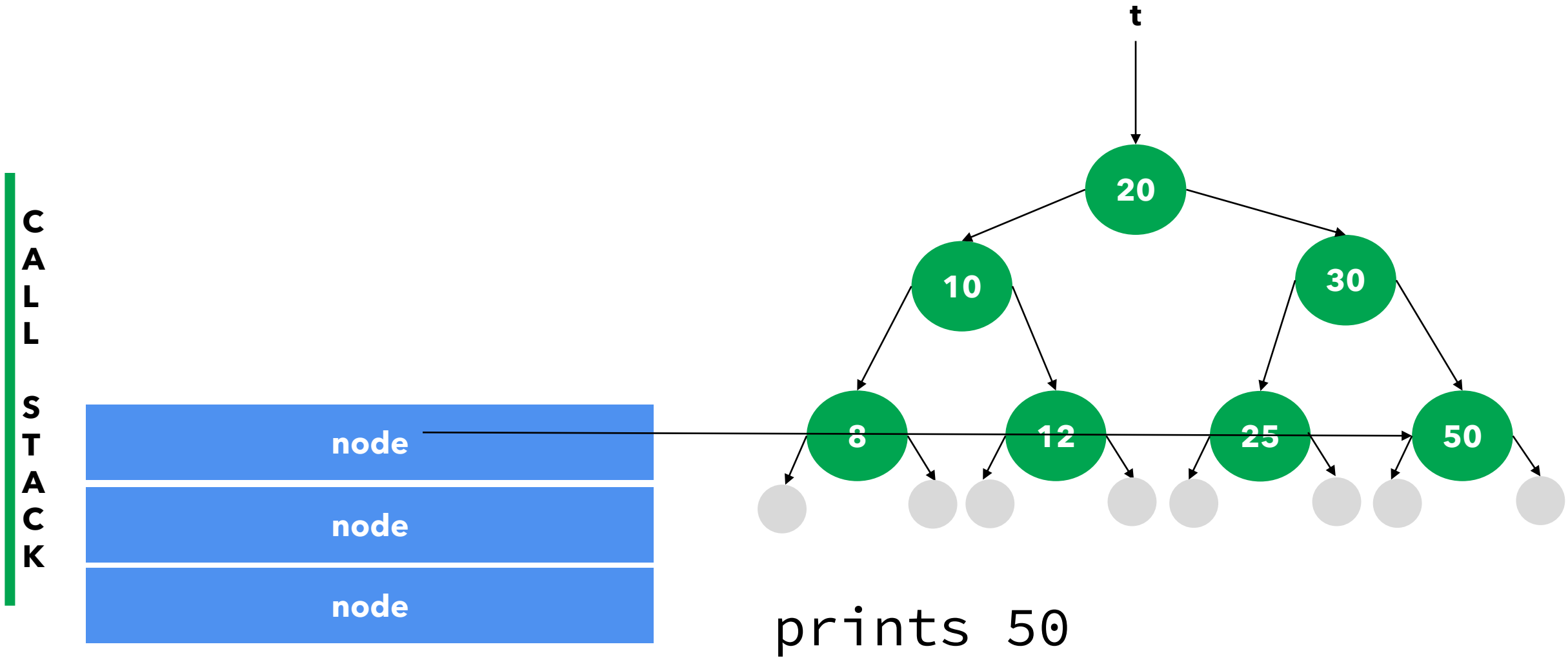
Esempio di DFS: visita inorder di un BST (inorder_tree_walk)



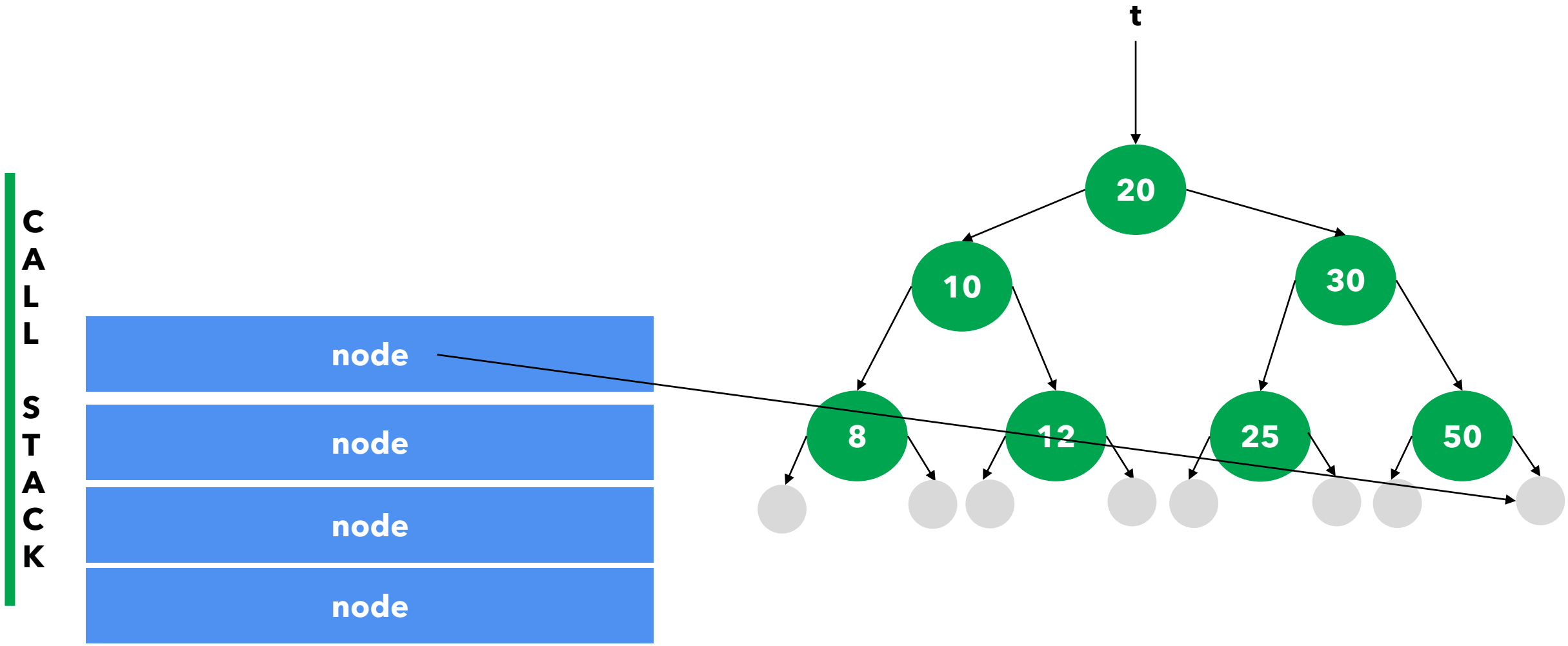
Esempio di DFS: visita inorder di un BST (inorder_tree_walk)



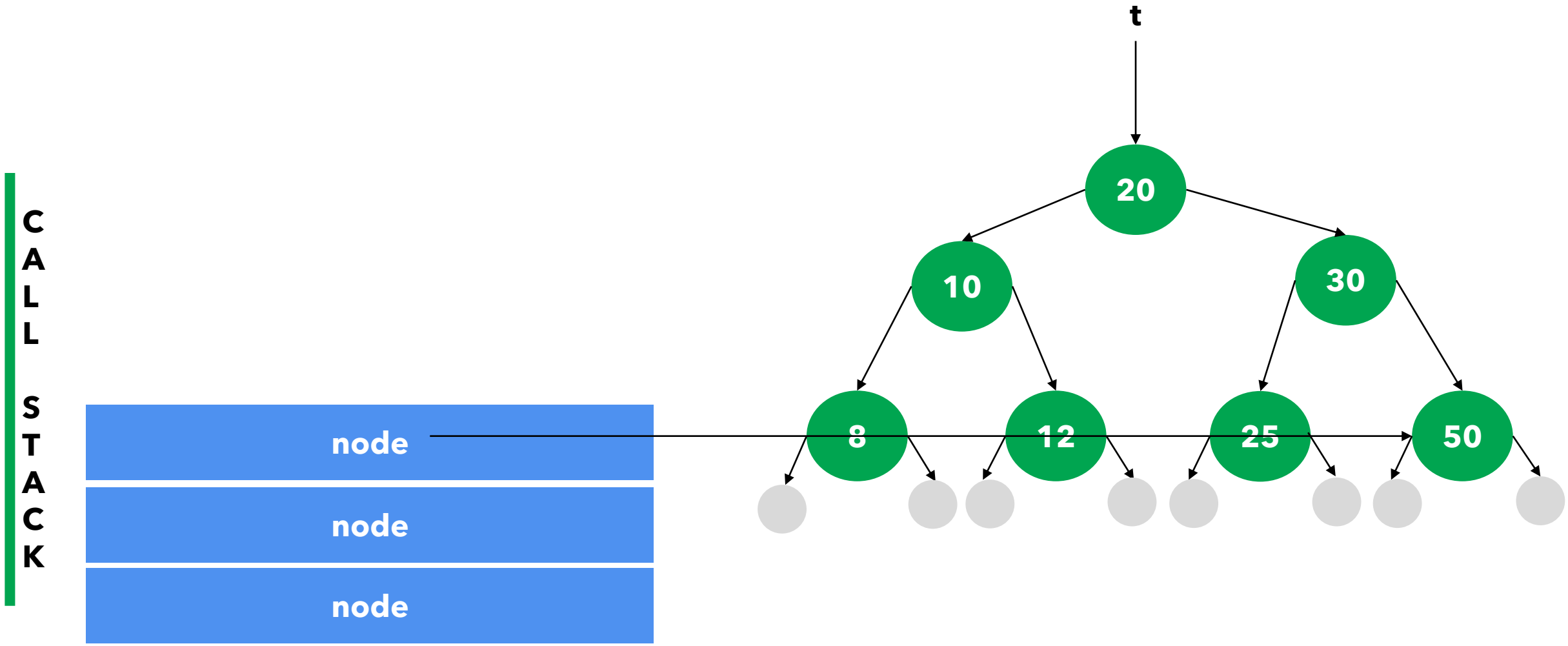
Esempio di DFS: visita inorder di un BST (inorder_tree_walk)



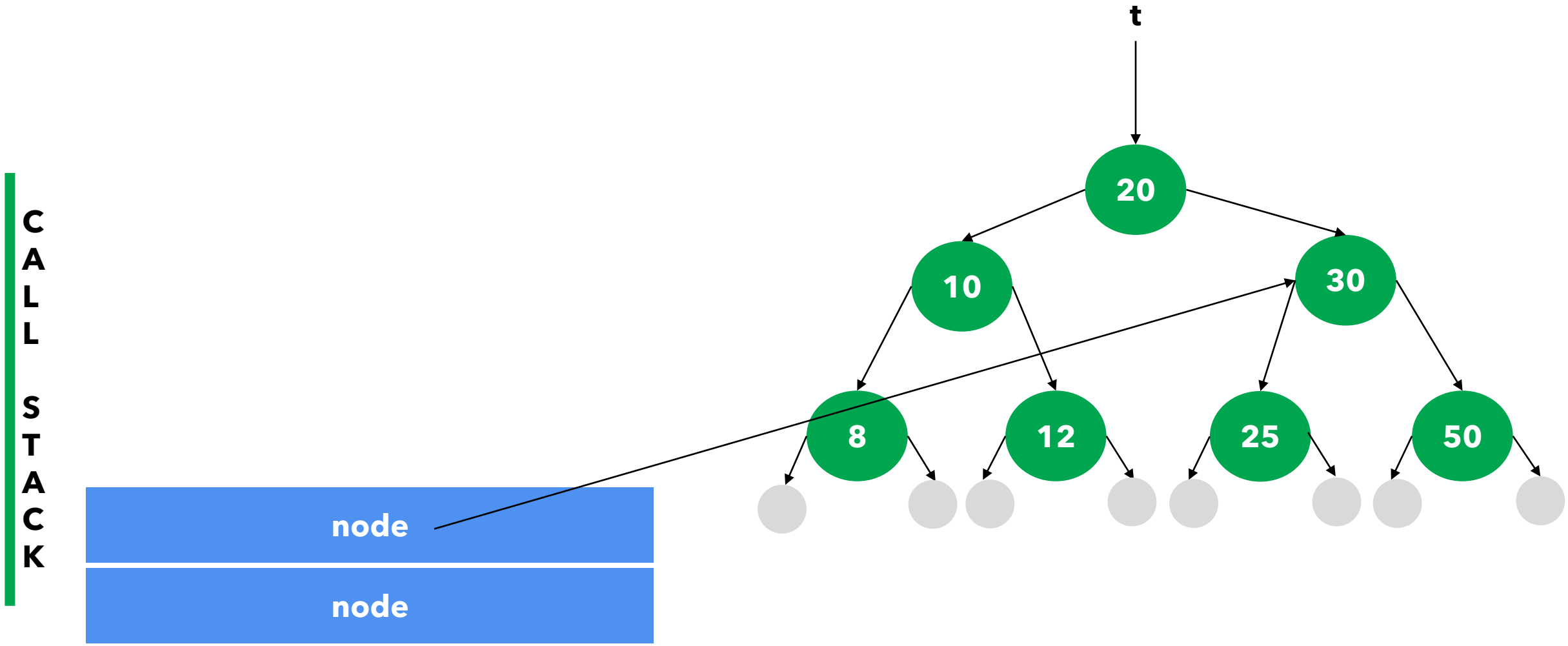
Esempio di DFS: visita inorder di un BST (inorder_tree_walk)



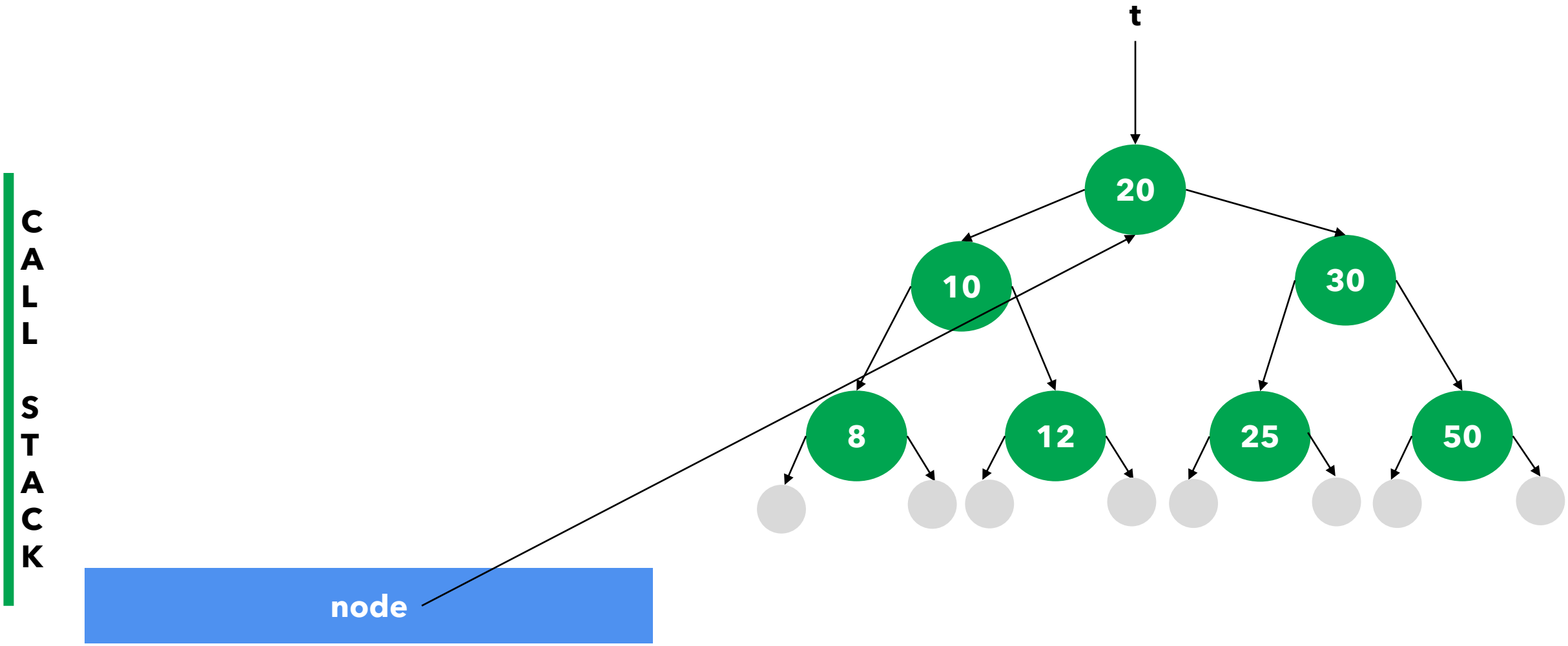
Esempio di DFS: visita inorder di un BST (inorder_tree_walk)



Esempio di DFS: visita inorder di un BST (inorder_tree_walk)

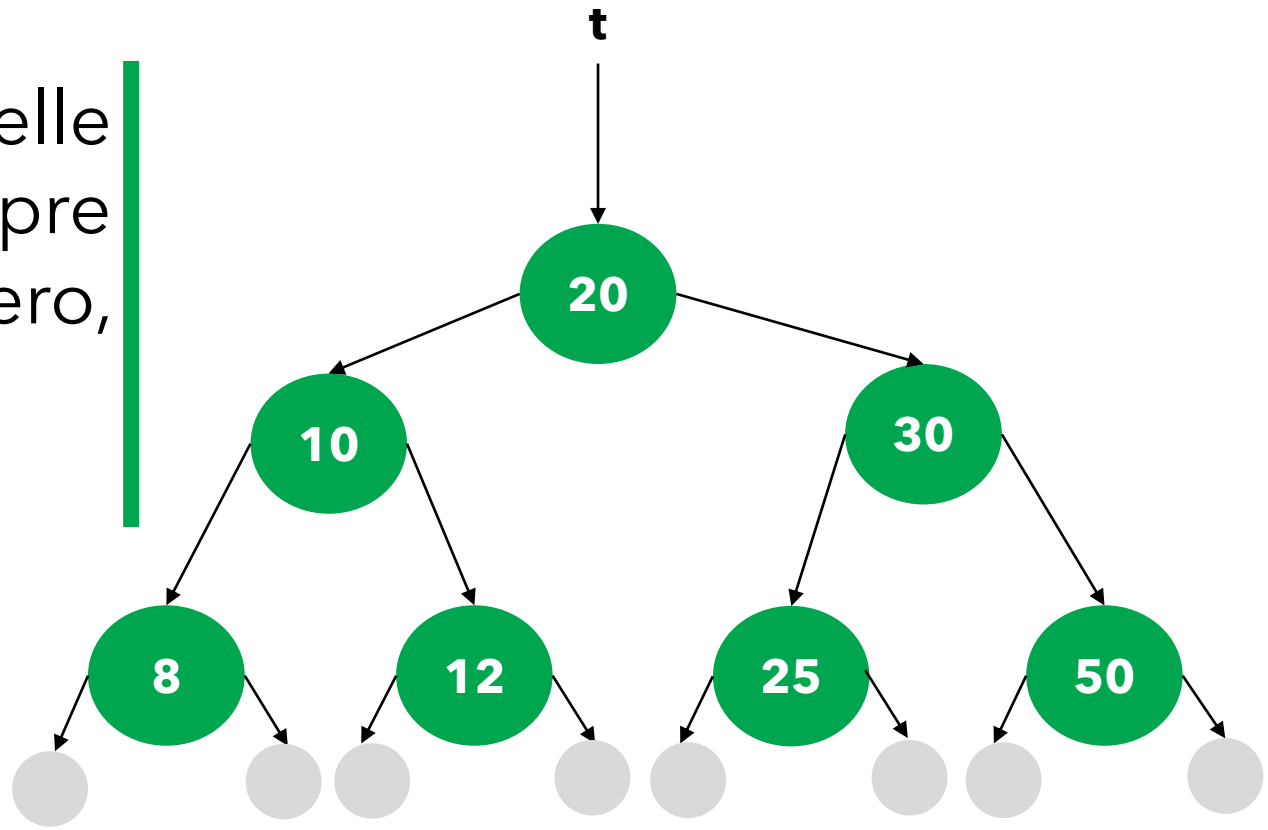


Esempio di DFS: visita inorder di un BST (inorder_tree_walk)



Esempio di DFS: visita `inorder` di un BST (`inorder_tree_walk`)

NB: l'altezza dello stack delle chiamate di `in_order` è sempre uguale al livello corrente dell'albero, se si contano i livelli da 1



Visita inorder, iterativa (per niente facile)

- Proviamo ad implementare la visita inorder iterativamente
- Non abbiamo più lo stack delle chiamate che mantiene in memoria i nodi visitati
- Possiamo però realizzare la stessa logica con un ADT stack
- Il campo dati dei nodi dello stack non sarà altro che un puntatore ad un nodo dell'albero che vogliamo visitare
- Dobbiamo inoltre «aumentare» con 2 campi booleani il nodo dell'albero, per ricordarci se abbiamo già visitato i sottoalberi

```
typedef struct stack_node {
   TREE_NODE *t;
   struct stack_node *prev, *next;
} STACK_NODE;
```

```
typedef struct node {
   int key;
   struct node *left, *right;
   BOOL visited_left, visited_right;
} TREE_NODE;
```

Visita inorder, iterativa (per niente facile)

```
STACK_NODE *init_stack_node(STACK_NODE *sn,
STACK_NODE *psn, TREE_NODE *t) {
   sn = (STACK_NODE*) malloc(sizeof(STACK_NODE));
   sn->t = t;
   sn->prev = psn;
   sn->next = NULL;

   return sn;
}
```

```
STACK_NODE *pop(STACK_NODE *st_ptr) {
   STACK_NODE *new_stack_pointer = NULL;
   if (st_ptr != NULL) {
      new_stack_pointer = st_ptr->prev;
      free(st_ptr);
   }
   return new_stack_pointer;
}
```

```
typedef enum bool {FALSE, TRUE} BOOL;
```

```
STACK_NODE *push(STACK_NODE *st_ptr, TREE_NODE *t) {
  STACK_NODE *new_stack_pointer = NULL;
  if (!st_ptr) {
    st_ptr = init_stack_node(st_ptr, NULL, t);
    new_stack_pointer = st_ptr;
  }
  else {
    st_ptr->next = init_stack_node(st_ptr->next, st_ptr, t);
    new_stack_pointer = st_ptr->next;
  }

  return new_stack_pointer;
}
```

Visita inorder, iterativa (per niente facile)

```
void in_order_iter(TREE_NODE *t_node) {
  STACK_NODE *stack_pointer = NULL;
  stack_pointer = push(stack_pointer, t_node);

  while (stack_pointer != NULL) {
    if (stack_pointer->t == NULL || (stack_pointer->t->visited_left
                                     && stack_pointer->t->visited_right)) {
          stack_pointer = pop(stack_pointer);
    }
    else if (!stack_pointer->t->visited_left) {
          stack_pointer = push(stack_pointer, stack_pointer->t->left);
          stack_pointer->prev->t->visited_left = TRUE;
    }
    else if (!stack_pointer->t->visited_right) {
          printf("%d ", stack_pointer->t->key);
          stack_pointer = push(stack_pointer, stack_pointer->t->right);
          stack_pointer->prev->t->visited_right = TRUE;
    }
  }
}
```

Esempio di DFS: visita preorder di un BST (inorder_tree_walk)

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` di un BST (`inorder_tree_walk`)

```
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);
}
```

Esempi di DFS: visita postorder iterativa

```
void post_order_iter(TREE_NODE *t_node) {
  STACK_NODE *stack_pointer = NULL;
  stack_pointer = push(stack_pointer, t_node);

  while (stack_pointer != NULL) {
    if (stack_pointer->t == NULL) {
      stack_pointer = pop(stack_pointer);
    }
    else if (!stack_pointer->t->visited_left) {
      stack_pointer = push(stack_pointer, stack_pointer->t->left);
      stack_pointer->prev->t->visited_left = TRUE;
    }
    else if (!stack_pointer->t->visited_right) {
      stack_pointer = push(stack_pointer, stack_pointer->t->right);
      stack_pointer->prev->t->visited_right = TRUE;
    }
    else {
      printf("%d ", stack_pointer->t->key);
      stack_pointer = pop(stack_pointer);
    }
  }
}
```

Esempi di DFS: visita preorder iterativa

```
void pre_order_iter(TREE_NODE *t_node) {
  STACK_NODE *stack_pointer = NULL;
  stack_pointer = push(stack_pointer, t_node);

  while (stack_pointer != NULL) {
    if (stack_pointer->t == NULL || (stack_pointer->t->visited_left
                                          && stack_pointer->t->visited_right)) {
          stack_pointer = pop(stack_pointer);
    }
    else {
       if (!stack_pointer->t->visited_left && !stack_pointer->t->visited_right) {
           printf("%d ", stack_pointer->t->key);
           stack_pointer = push(stack_pointer, stack_pointer->t->left);
           stack_pointer->prev->t->visited_left = TRUE;
       }
       else if (!stack_pointer->t->visited_right) {
          stack_pointer = push(stack_pointer, stack_pointer->t->right);
          stack_pointer->prev->t->visited_right = TRUE;
       }
    }
  }
}
```

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**
- La BFS diventerà chiarissima con dei disegni
- In questo caso sarà più semplice una versione iterativa

```
typedef struct queue_node {
  TREE_NODE *t;
  struct queue_node *next;
} QUEUE_NODE;
```

Breadth-first search (BFS) su un albero binario

```
BFS(Tree, Queue):
    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 del nodo dell'albero in testa alla coda (se non sono nulli), poi rimuovi la testa della coda.

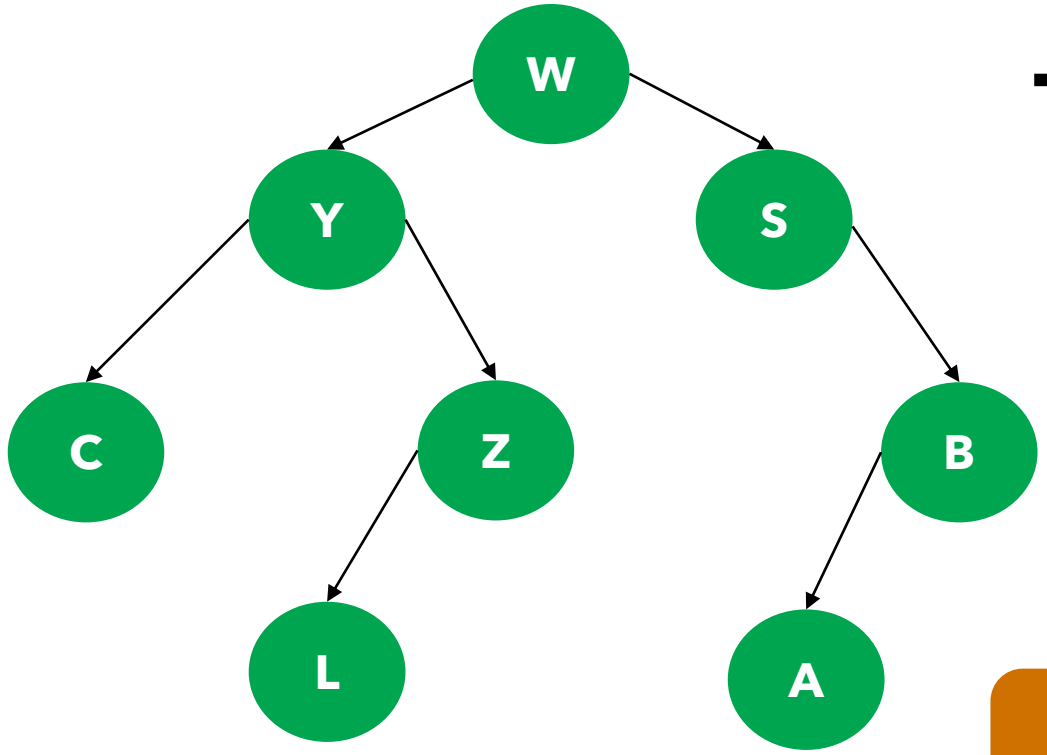
Breadth-first search (BFS) su alberi binari

livello 0

livello 1

livello 2

livello 3

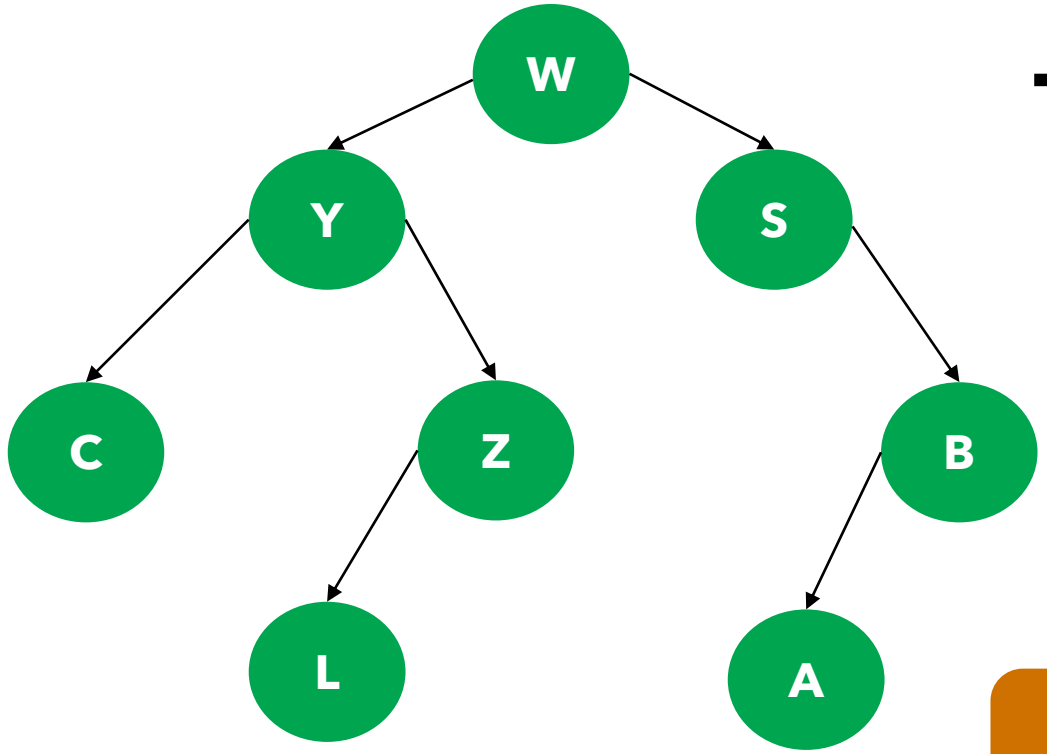


TREE T

QUEUE Q

Situazione a fine iterazione -1:
coda vuota, livello raggiunto: -1

Breadth-first search (BFS) su alberi binari



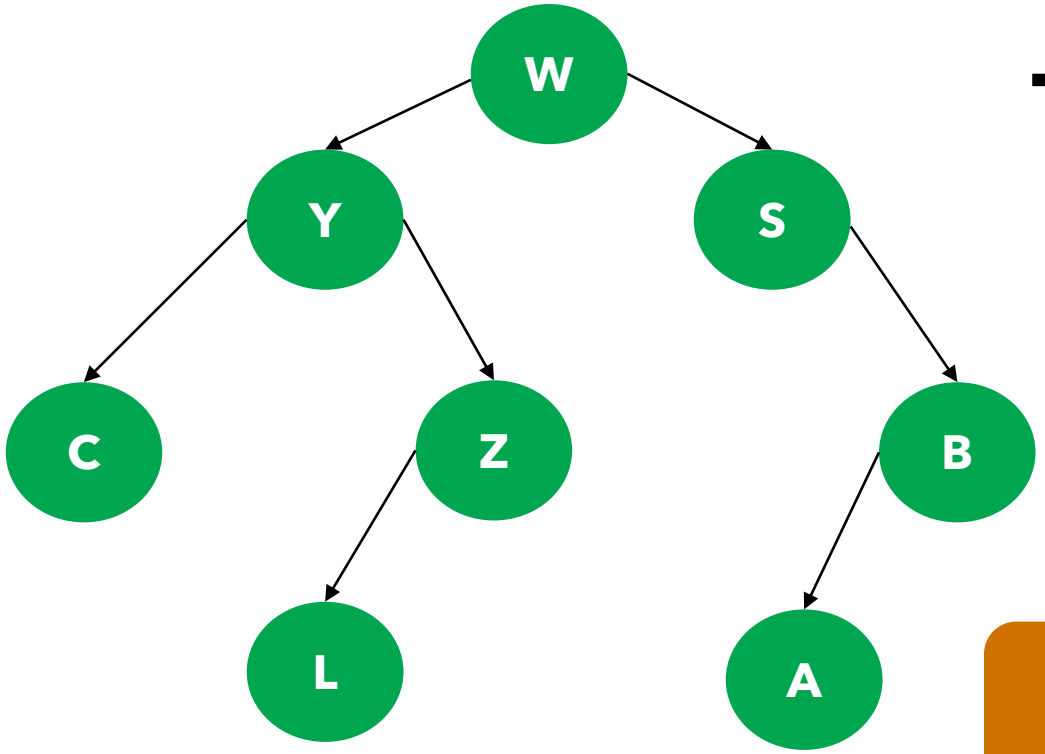
TREE T

Situazione a fine iterazione 0:
livello raggiunto: 0

QUEUE Q
enqueue(W);

W

Breadth-first search (BFS) su alberi binari



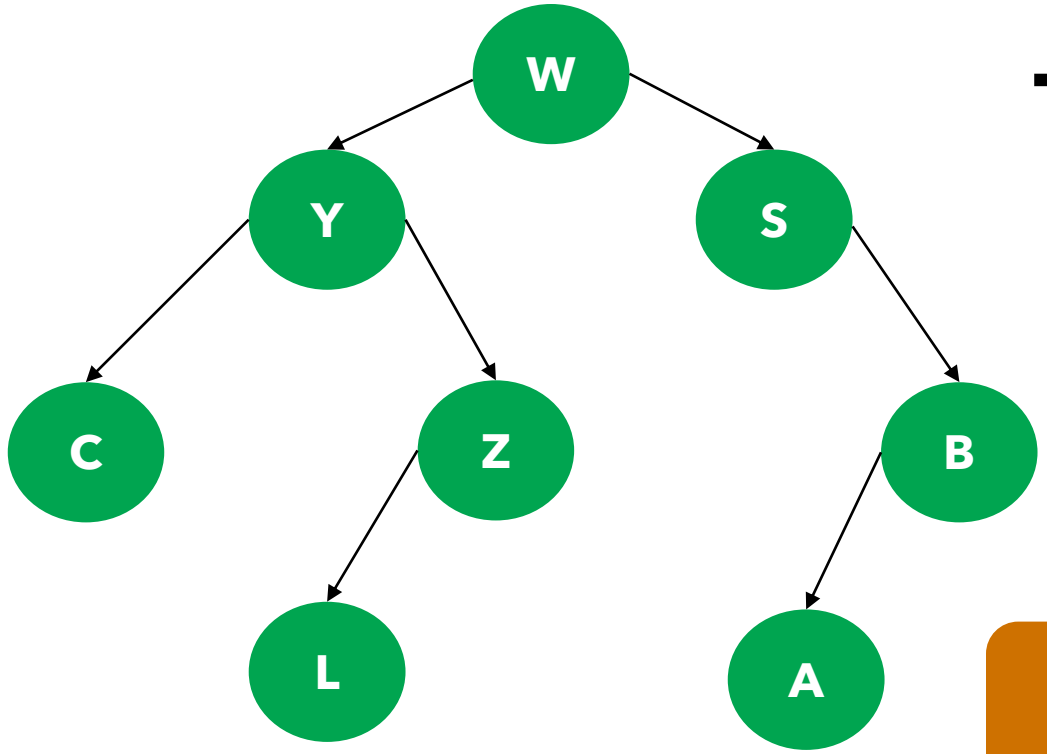
TREE T

Situazione a iterazione 1:
livello raggiunto: 1

QUEUE Q

enqueue(W.left); enqueue(W.right); ossia **enqueue(Y); enqueue(S);**

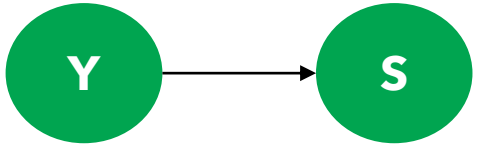
Breadth-first search (BFS) su alberi binari



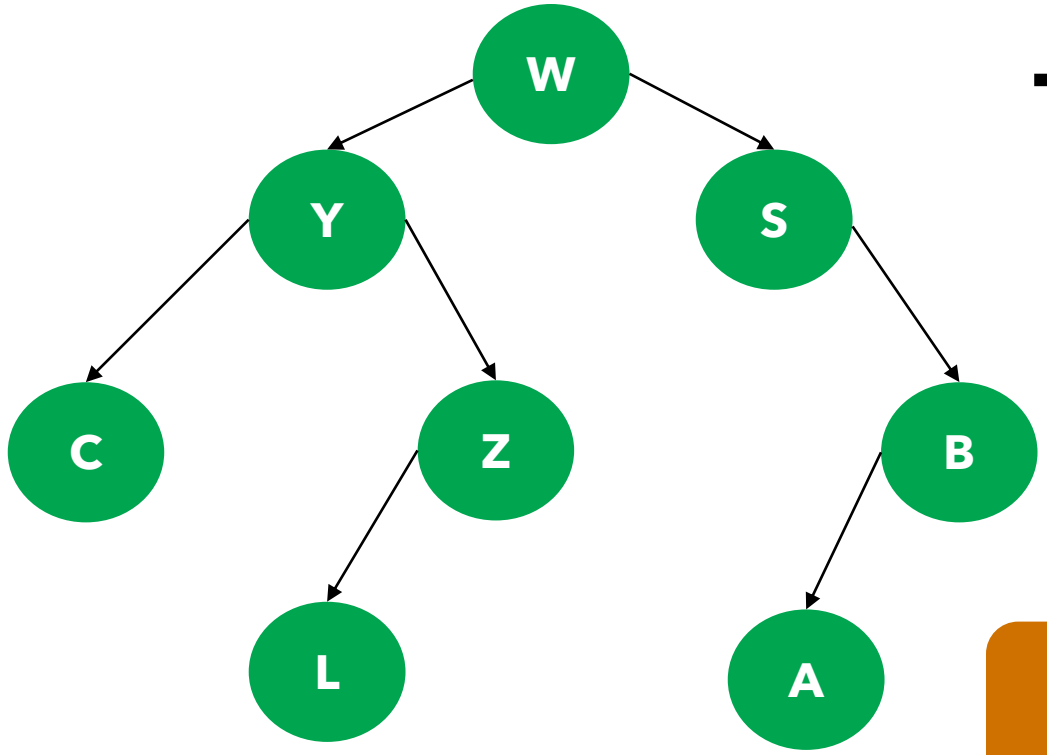
TREE T

**Situazione a fine iterazione 1:
livello raggiunto: 1**

QUEUE Q
dequeue();



Breadth-first search (BFS) su alberi binari

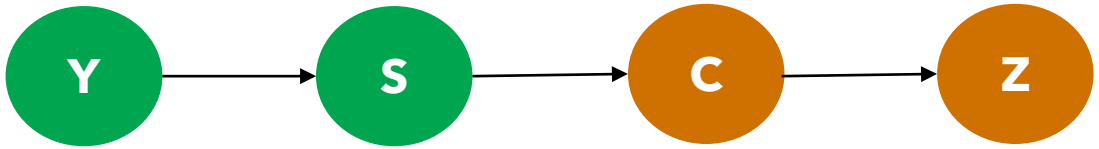


TREE T

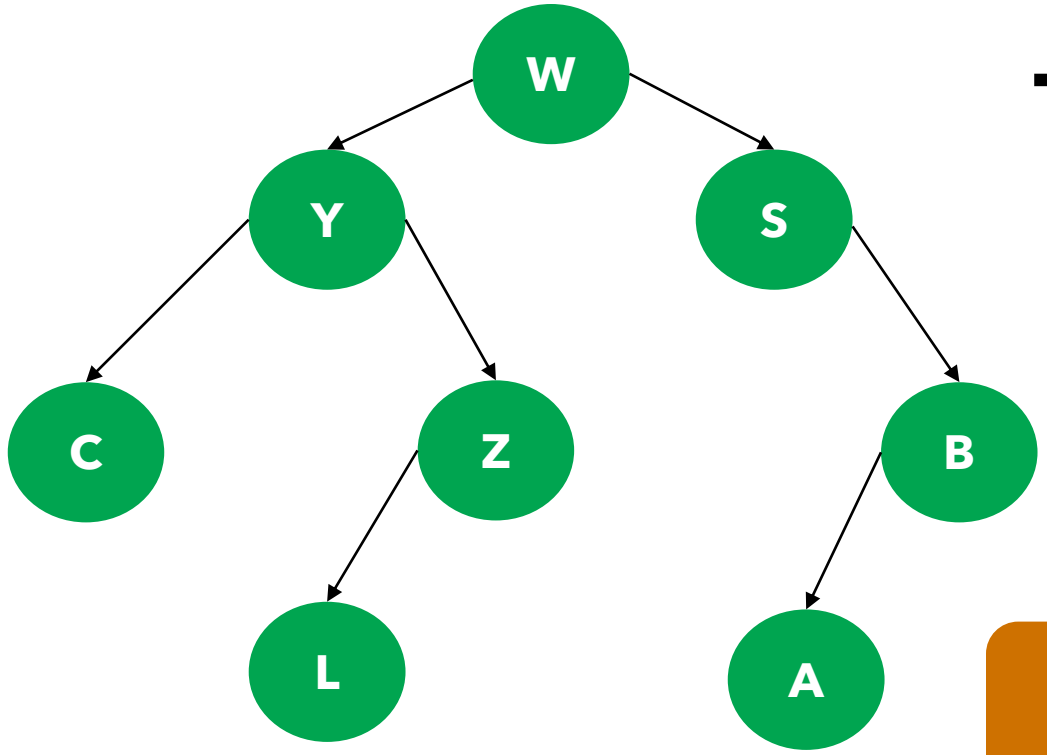
Situazione a iterazione 2:
livello raggiunto: 2

QUEUE Q

enqueue(Y.left); enqueue(Y.right); ossia **enqueue(C); enqueue(Z);**



Breadth-first search (BFS) su alberi binari

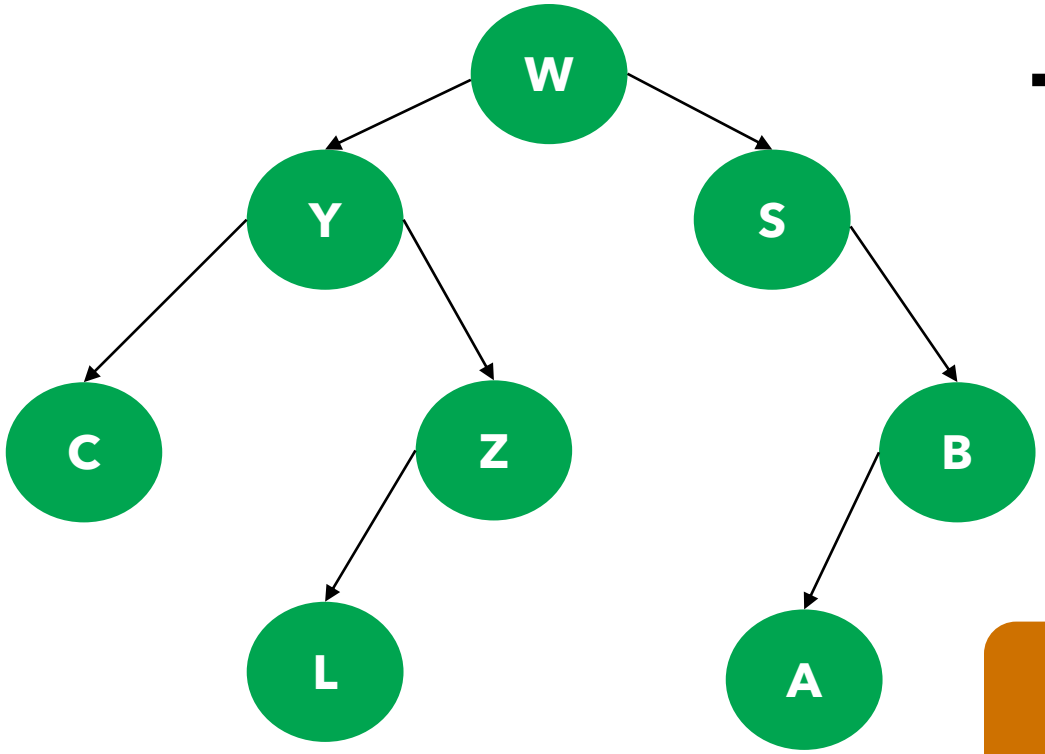


TREE T

Situazione a fine iterazione 2:
livello raggiunto: 2

QUEUE Q
dequeue();

Breadth-first search (BFS) su alberi binari

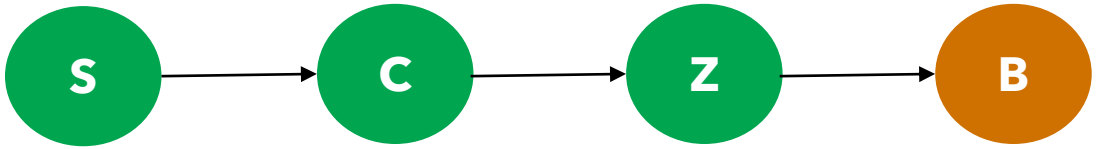


TREE T

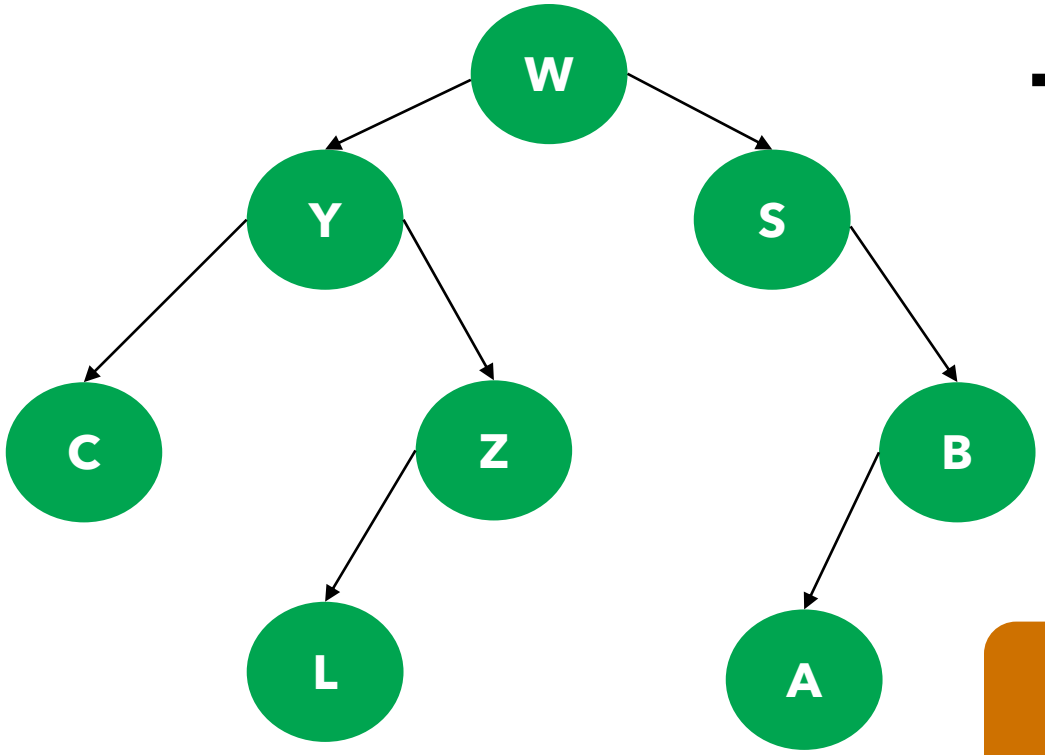
Situazione iterazione 3:
livello raggiunto: 2

QUEUE Q

enqueue(S.right); ossia **enqueue(B);**



Breadth-first search (BFS) su alberi binari

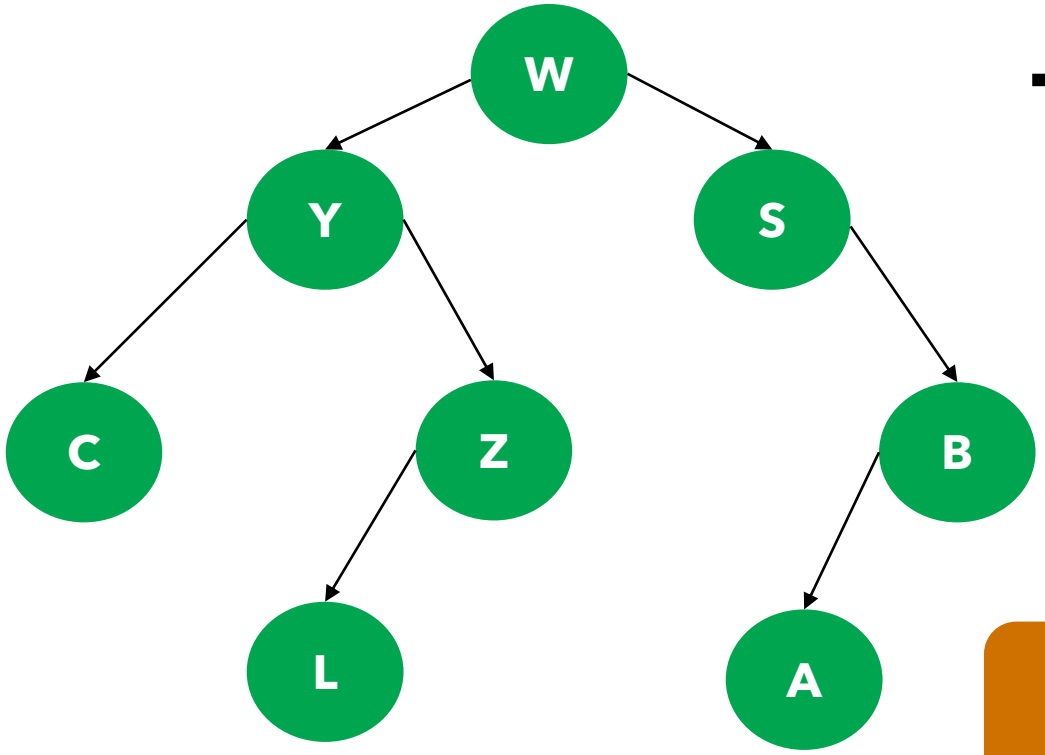


TREE T

Situazione a fine iterazione 3:
livello raggiunto: 2

QUEUE Q
dequeue();

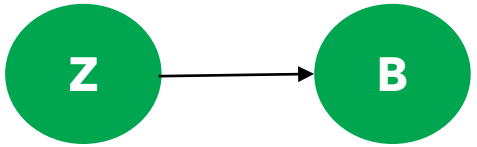
Breadth-first search (BFS) su alberi binari



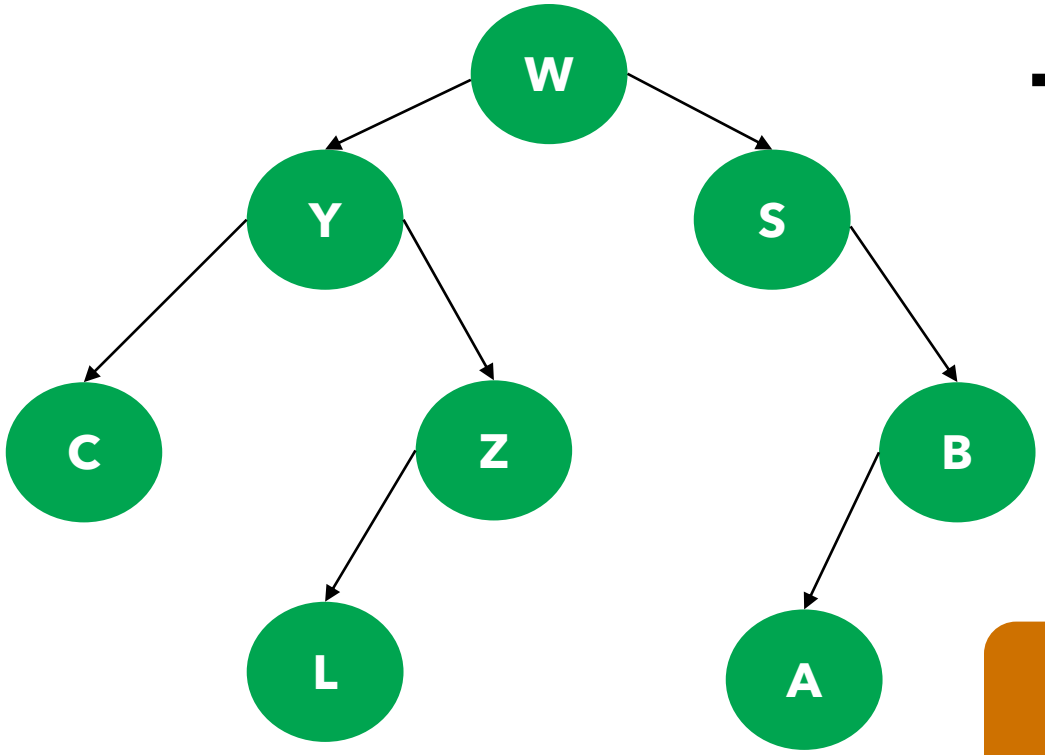
TREE T

Situazione a fine iterazione 4:
livello raggiunto: 2

QUEUE Q
dequeue();



Breadth-first search (BFS) su alberi binari



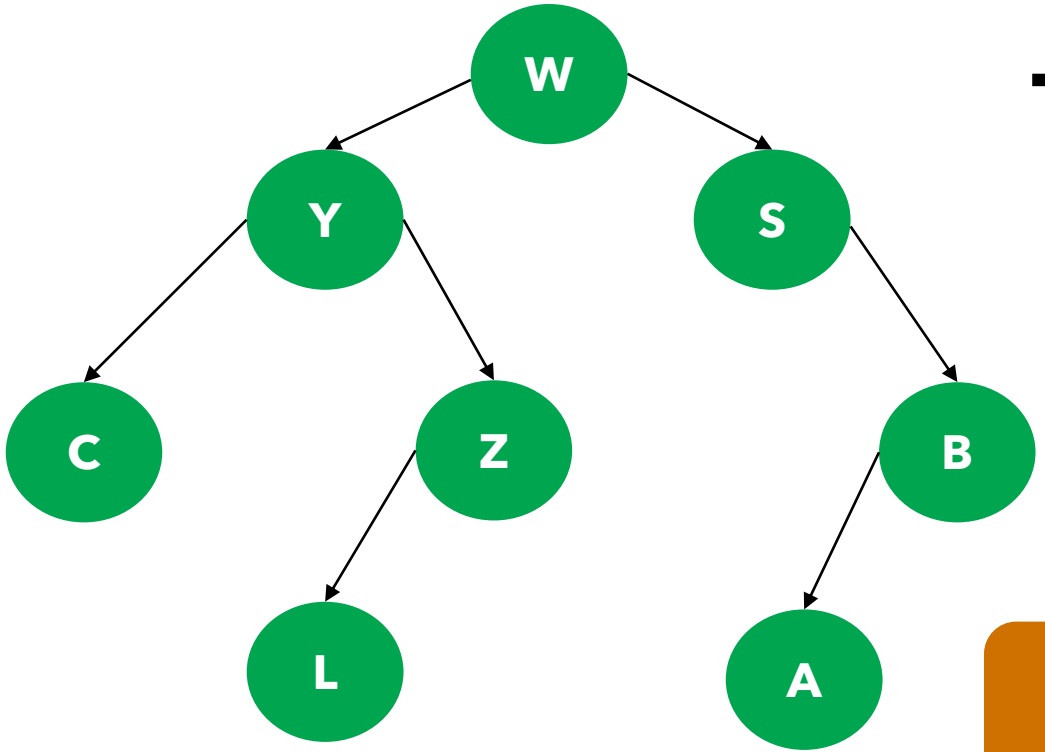
TREE T

Situazione a iterazione 5:
livello raggiunto: 3

QUEUE Q

enqueue(Z.left); ossia **enqueue(L);**

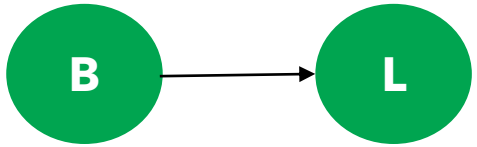
Breadth-first search (BFS) su alberi binari



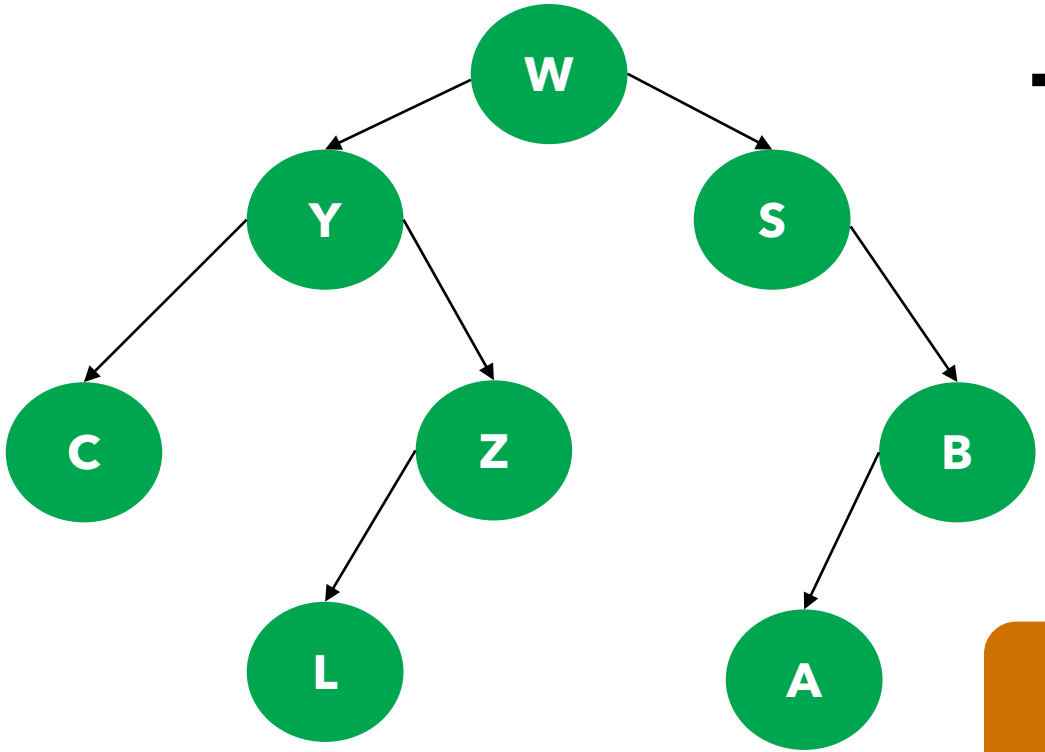
TREE T

Situazione a fine iterazione 5:
livello raggiunto: 3

QUEUE Q
dequeue();



Breadth-first search (BFS) su alberi binari



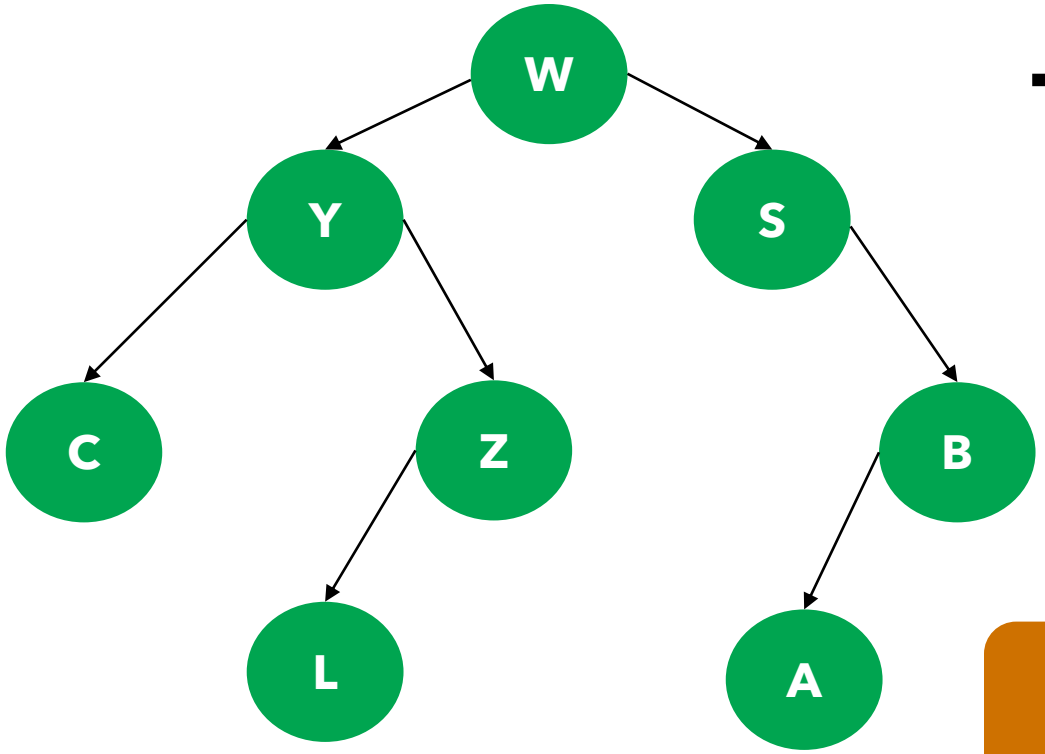
TREE T

Situazione a iterazione 6:
livello raggiunto: 3

QUEUE Q

enqueue(B.left); ossia **enqueue(A);**

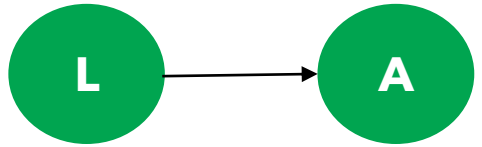
Breadth-first search (BFS) su alberi binari



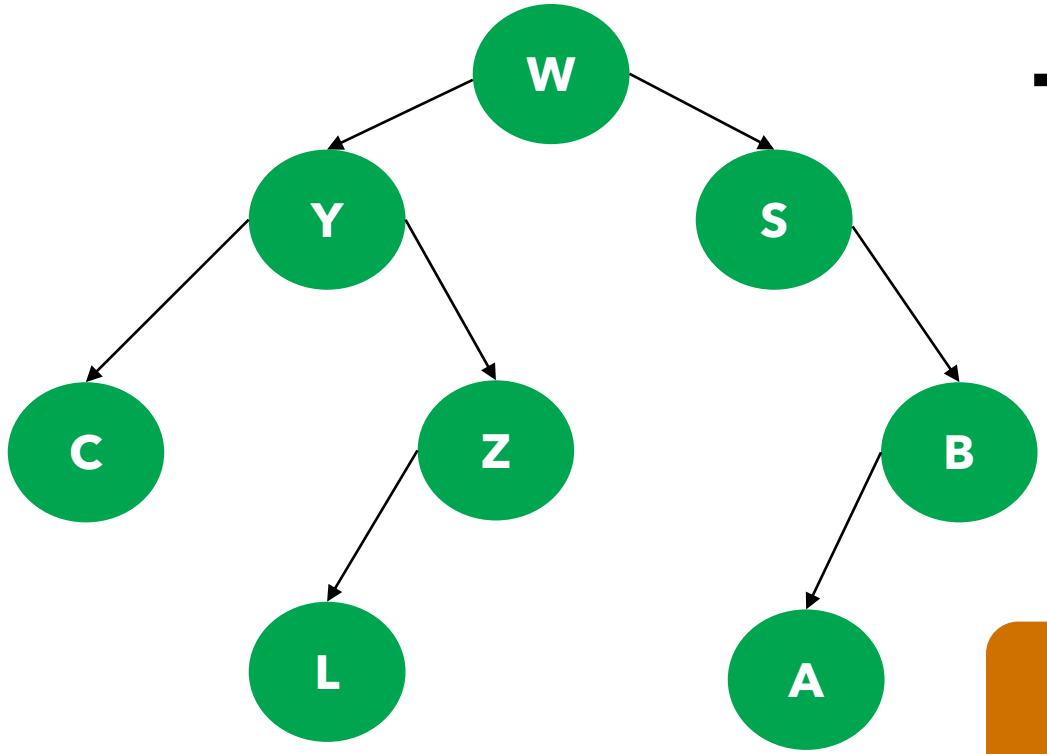
TREE T

Situazione a fine iterazione 6:
livello raggiunto: 3

QUEUE Q
dequeue();



Breadth-first search (BFS) su alberi binari

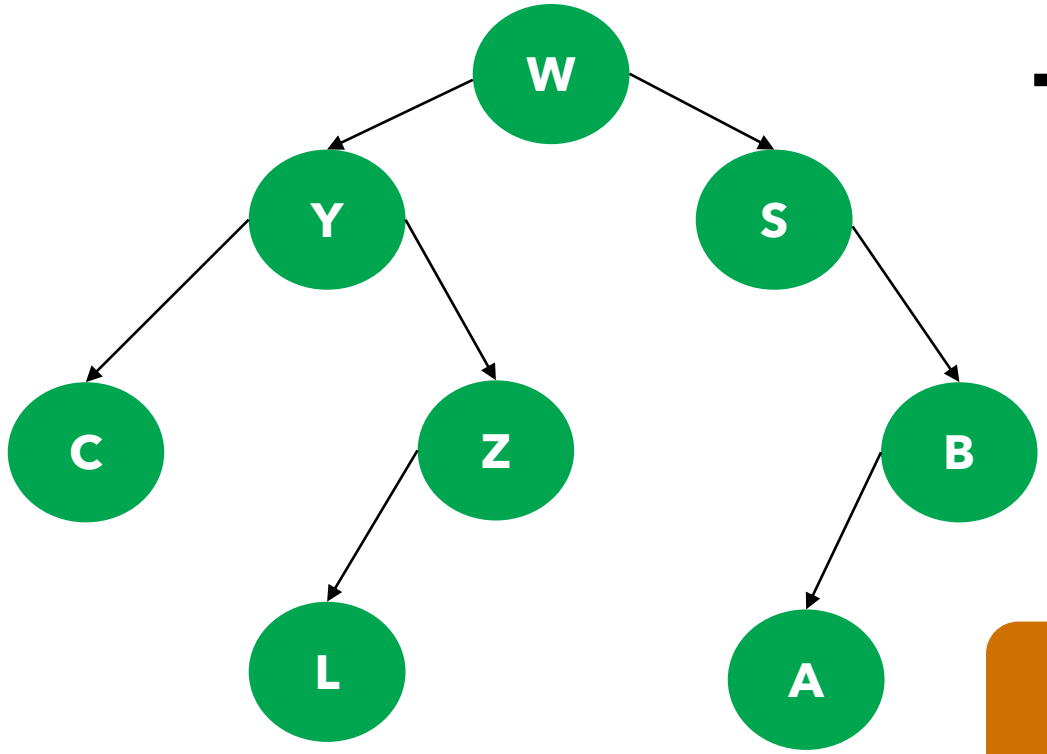


TREE T

Situazione a fine iterazione 7:
livello raggiunto: 3

QUEUE Q
dequeue();

Breadth-first search (BFS) su alberi binari



TREE T

Situazione a fine iterazione 8:
livello raggiunto: 3

QUEUE Q
dequeue();

Breadth-first search (BFS) su un albero binario

```
void breadth_first_search(TREE_NODE *t) {
 if (t) {
  QUEUE_NODE *q_head = NULL, *q_tail = NULL;
  q_tail = enqueue(q_tail, t);
  q_head = q_tail;
  printf("%d\n", q_head->t->key);
  unsigned current_level = q_head->t->level;
  unsigned old_level;

  while (q_head) {
   unsigned old_level = current_level;
   unsigned star_counter;
   current_level = q_head->t->level;
   //if level changed, print newline
   if (current_level != old_level) {
    putchar('\n');
   }
   if (q_head->t->left) {
    //n stars means level n
    print_n_chars(q_head->t->left->level, '*');
    printf("%d ", q_head->t->left->key);
    q_tail = enqueue(q_tail, q_head->t->left);
   }
   if (q_head->t->right) {
    //n stars means level n
    print_n_chars(q_head->t->right->level, '*');
    printf("%d ", q_head->t->right->key);
    q_tail = enqueue(q_tail, q_head->t->right);
   }
   q_head = dequeue(q_head);
  }
 }
}
```

Breadth-first search (BFS) su un albero binario

```
breadth first search:
12
*8 *16
**4 **10 **15 **20

in order, recursive and iterative:
4 8 10 12 15 16 20
4 8 10 12 15 16 20

pre order, recursive and iterative:
12 8 4 10 16 15 20
12 8 4 10 16 15 20

post order, recursive and iterative:
4 10 8 15 20 16 12
4 10 8 15 20 16 12
```

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. 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(T, k):
    current_parent = nil
    while (T):
        if k <= T.key:
            T = T.left
        else
            T = T.right

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

Mancano un bel po' di pezzi in questo codice. Sicuramente, scritto così, fa le cose a metà.
Quali sono i problemi?

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

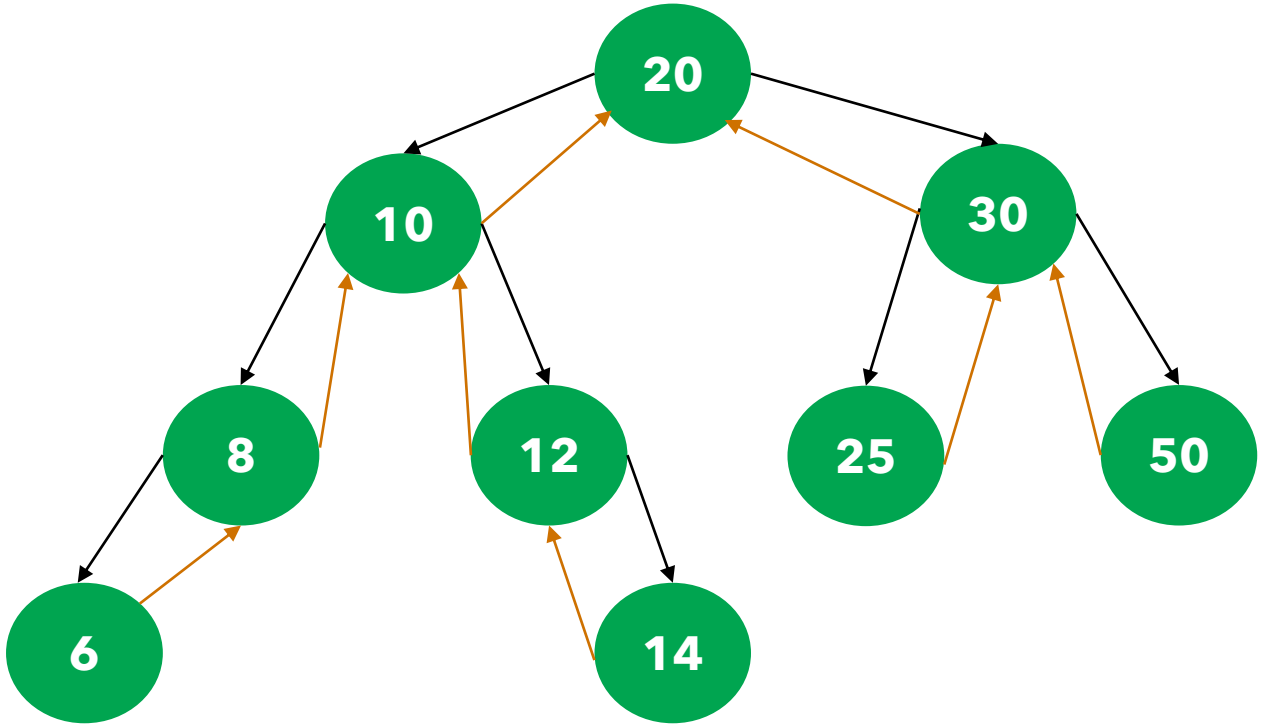
```
bst_insert(T, k):
    current_parent = nil
    last_left = false
    while (T):
        current_parent = T
        if k <= T.key:
            T = T.left
            last_left = true
        else
            T = T.right
            last_left = false

    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
```

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

```
TREE_NODE *bst_insert_iterative(TREE_NODE *t, int key, unsigned int level) {
 TREE_NODE *parent_pointer = NULL;
 BOOL last_left = FALSE;  //TRUE if went left in last iteration
 while (t) {
  parent_pointer = t;
  if (key <= t->key) {
   t = t->left;
   last_left = TRUE;
  }
  else {
   t = t->right;
   last_left = FALSE;
  }
  level++;
 }
 TREE_NODE *new_node = (TREE_NODE*) malloc(sizeof(TREE_NODE));
 new_node->key = key;
 new_node->left = NULL;
 new_node->right = NULL;
 new_node->parent = parent_pointer;
 if (parent_pointer) {
  if (last_left) {
   parent_pointer->left = new_node;
  }
  else {
   parent_pointer->right = new_node;
  }
  return t;
 }
 return new_node;
}
```

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 una procedura corretta.

Ricerca del nodo successore in un BST

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

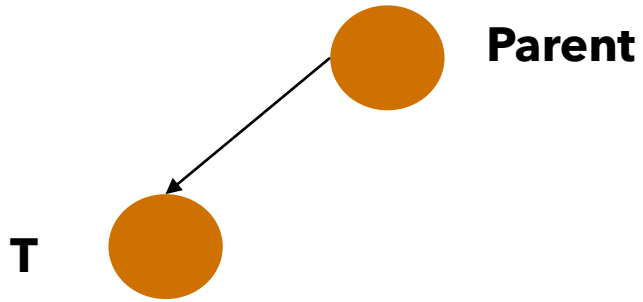
Il successore di un nodo T che ha il sottoalbero destro (T.right != nil) è il nodo minimo del sottoalbero destro di T. Effettivamente, in una sequenza ordinata p->q->r->s->t->w, q è il successore di p perché q è l'elemento minimo della sequenza q->r->s->t->w.

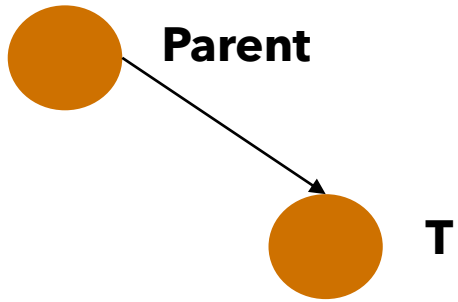
Ricerca del nodo successore in un BST

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 di T.left sono tutte minori o uguali di T.key.

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

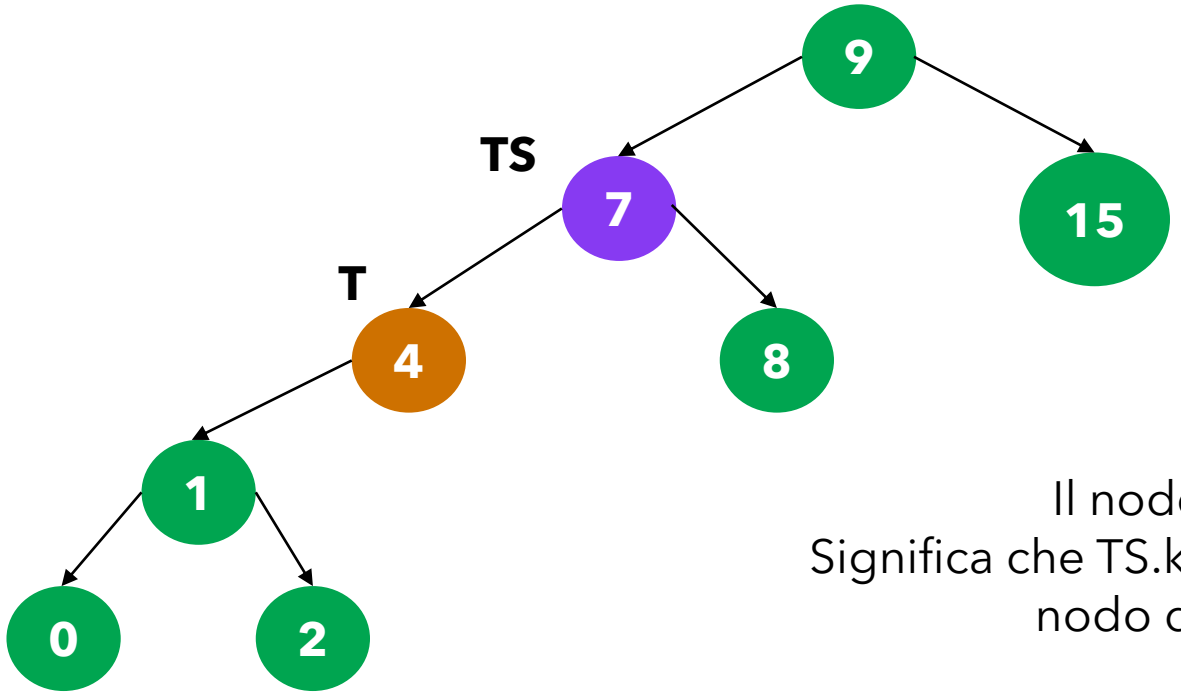
- T è il figlio sinistro di T.parent
- T è il figlio destro di T.parent





Ricerca del nodo successore in un BST

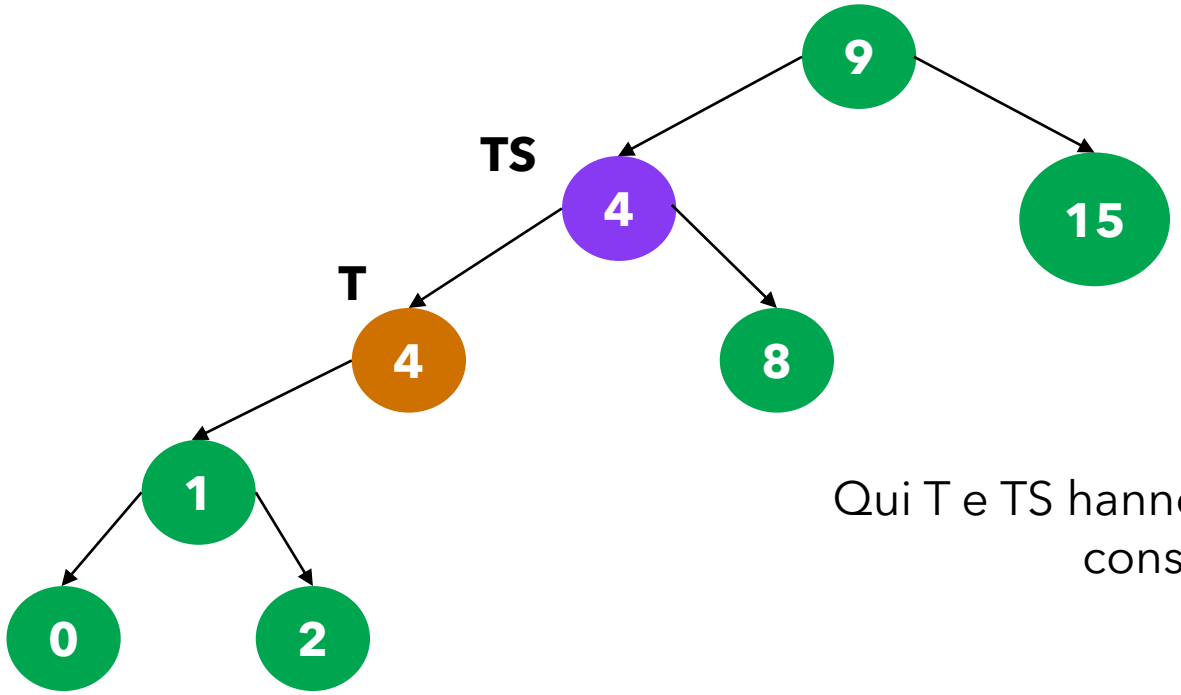
Se T è figlio sinistro di T.parent e T.right è nullo, allora sicuramente il successore è proprio T.parent.
Considera il disegno:



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, t.c. TS.key >= T.key

Ricerca del nodo successore in un BST

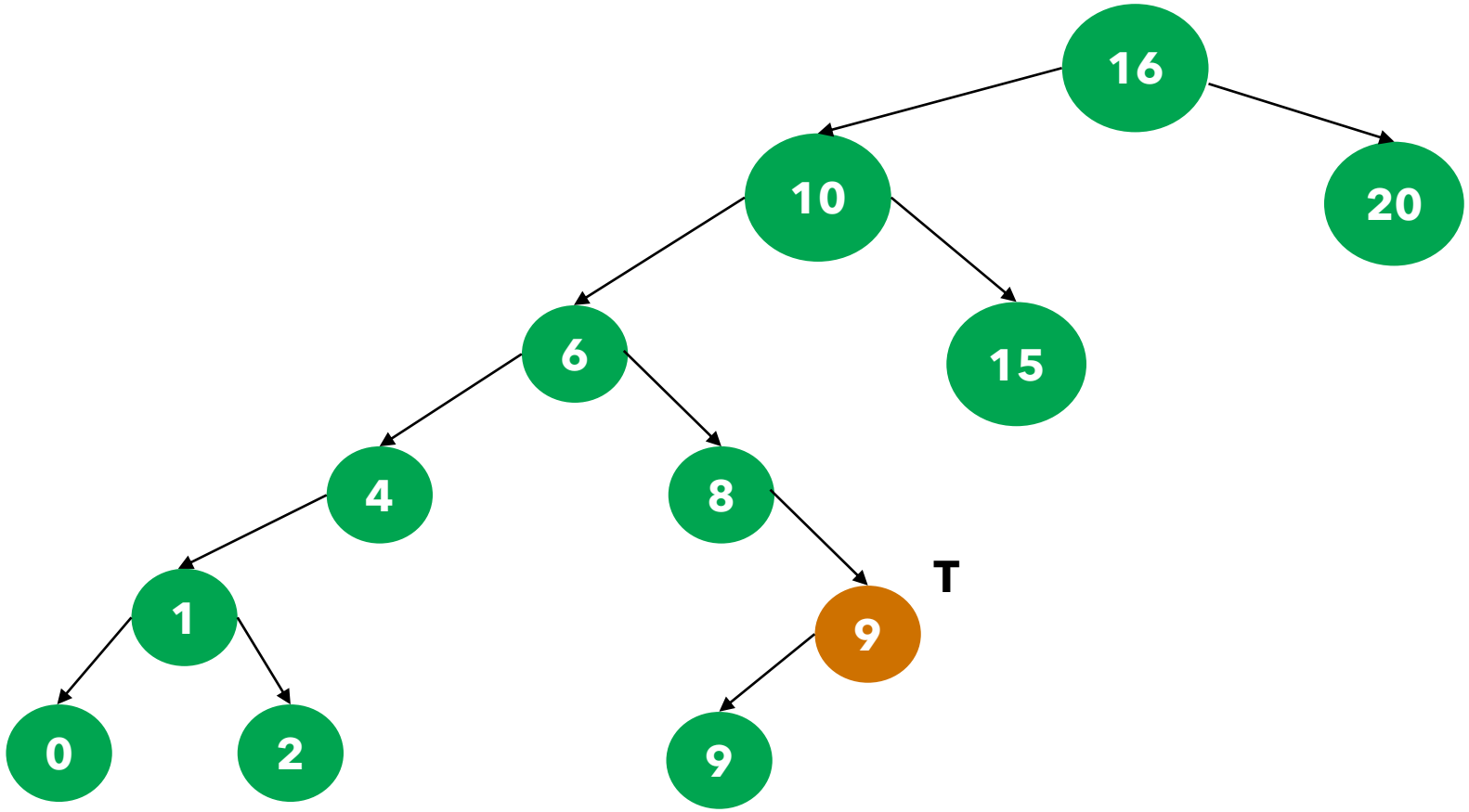
Se T è figlio sinistro di T.parent e T.right è nullo, allora sicuramente il successore è proprio T.parent.
Considera il disegno:



Qui T e TS hanno la stessa chiave. Il discorso non cambia, consideriamo TS il successore di T

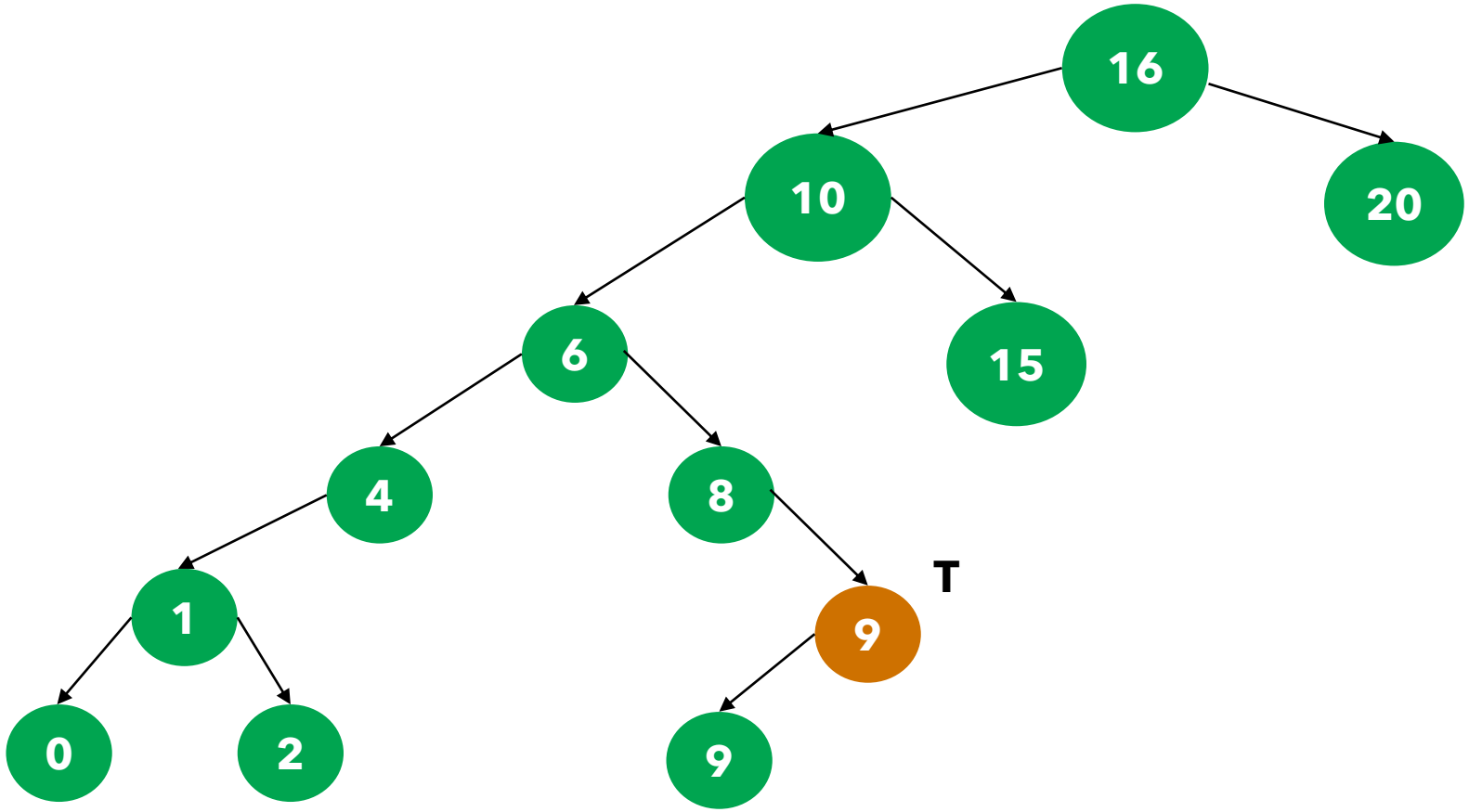
Ricerca del nodo successore in un BST

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



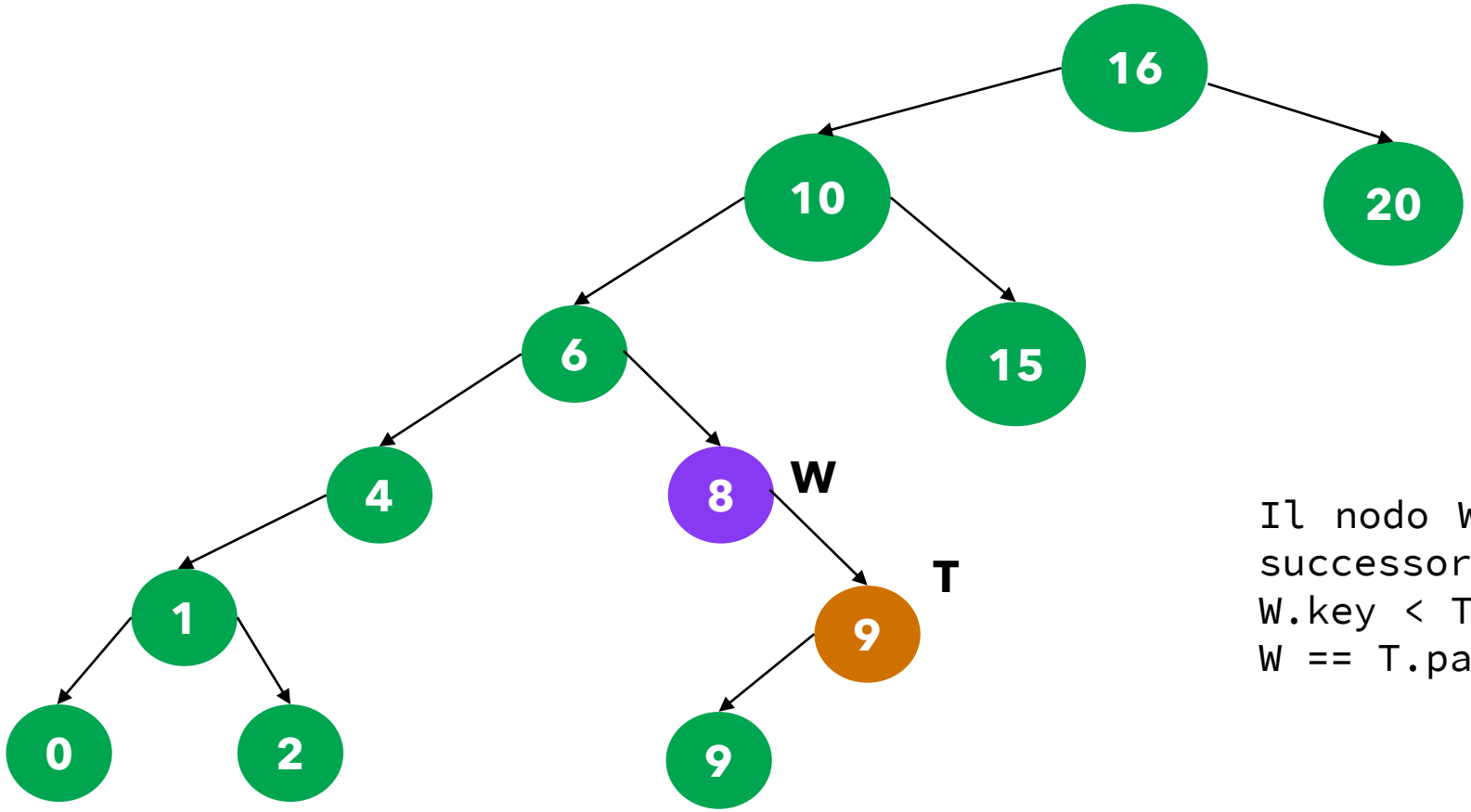
Ricerca del nodo successore in un BST

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



Ricerca del nodo successore in un BST

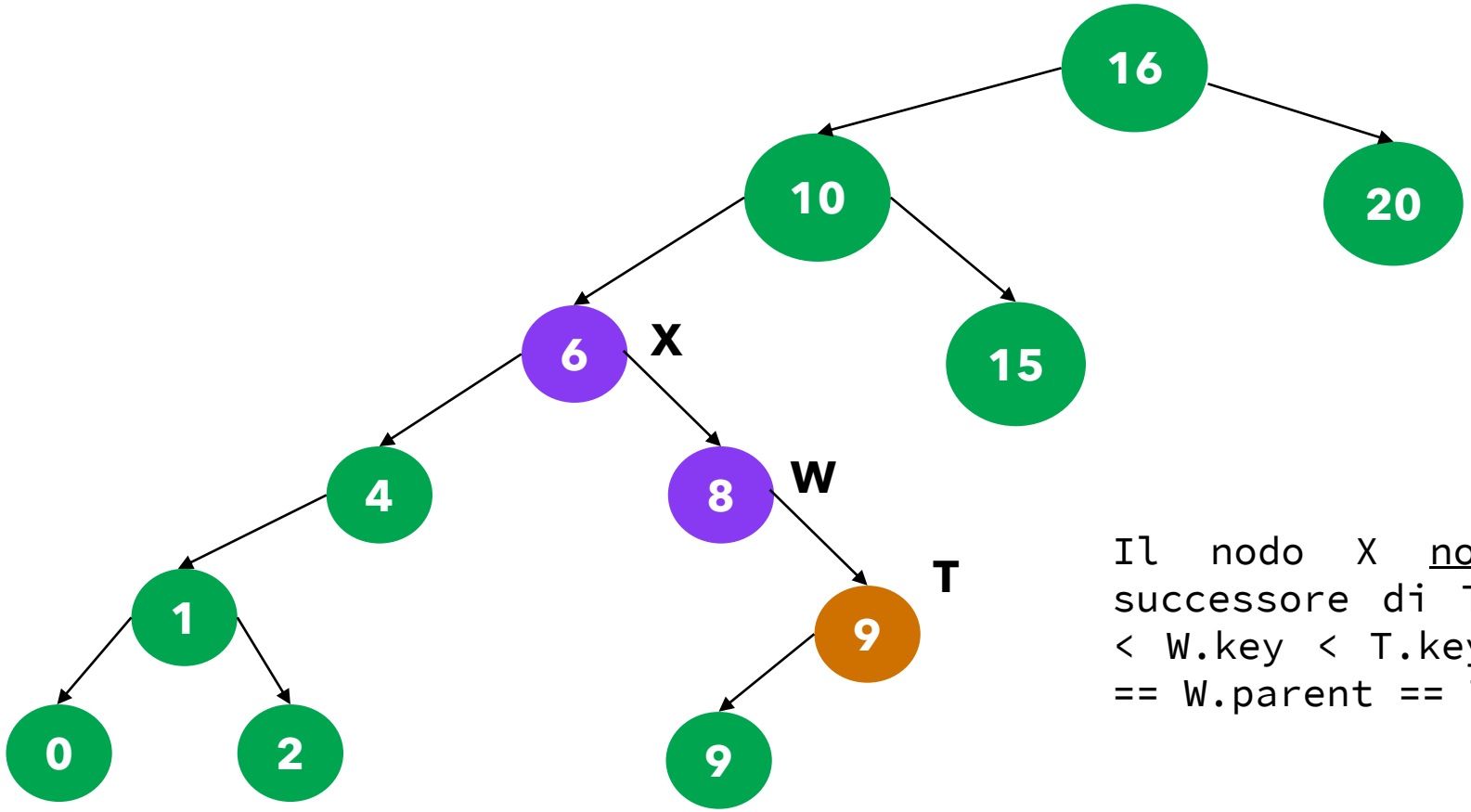
A sinistra non ha senso andare, a destra non possiamo nemmeno provarci… andiamo verso l'alto!



Il nodo W non può essere il successore di T, in quando W.key < T.key. Questo perché W == T.parent

Ricerca del nodo successore in un BST

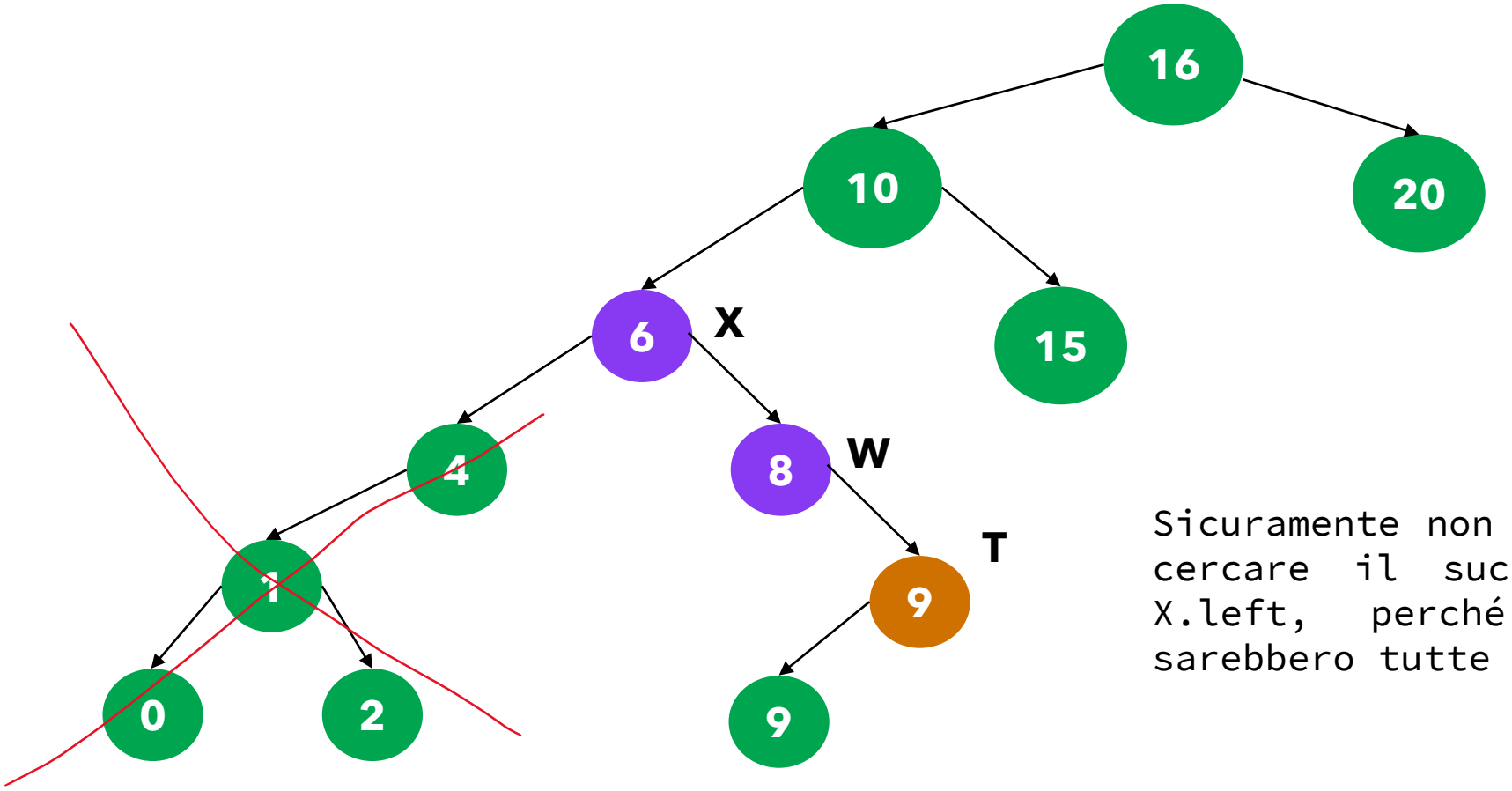
A sinistra non ha senso andare, a destra non possiamo nemmeno provarci… andiamo verso l'alto!



Il nodo X non può essere il successore di T, in quando X.key < W.key < T.key. Questo perché X == W.parent == T.parent.parent

Ricerca del nodo successore in un BST

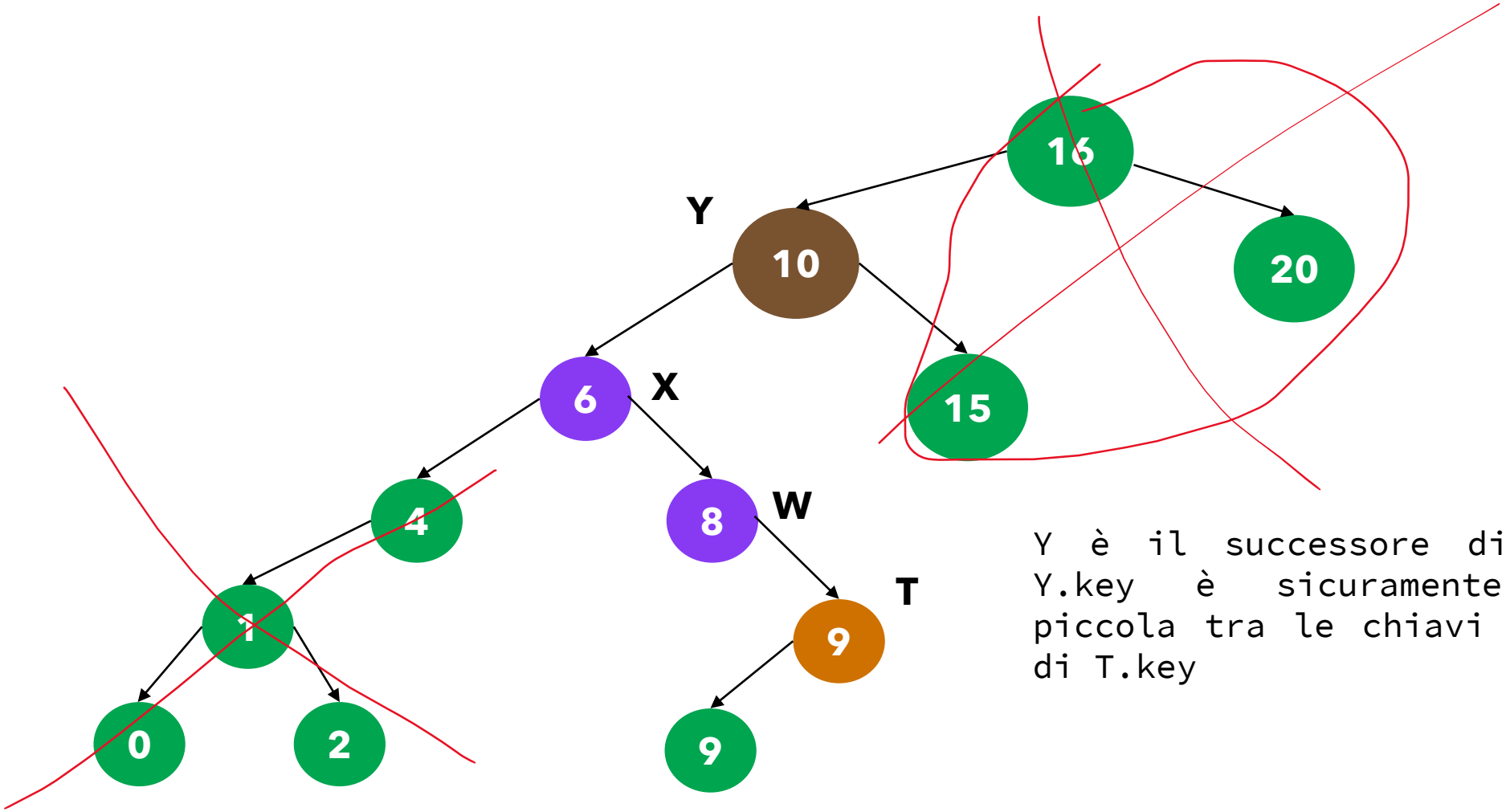
A sinistra non ha senso andare, a destra non possiamo nemmeno provarci… andiamo verso l'alto!



Sicuramente non ha senso andare a cercare il successore di T in X.left, perché lì le chiavi sarebbero tutte < T.key

Ricerca del nodo successore in un BST

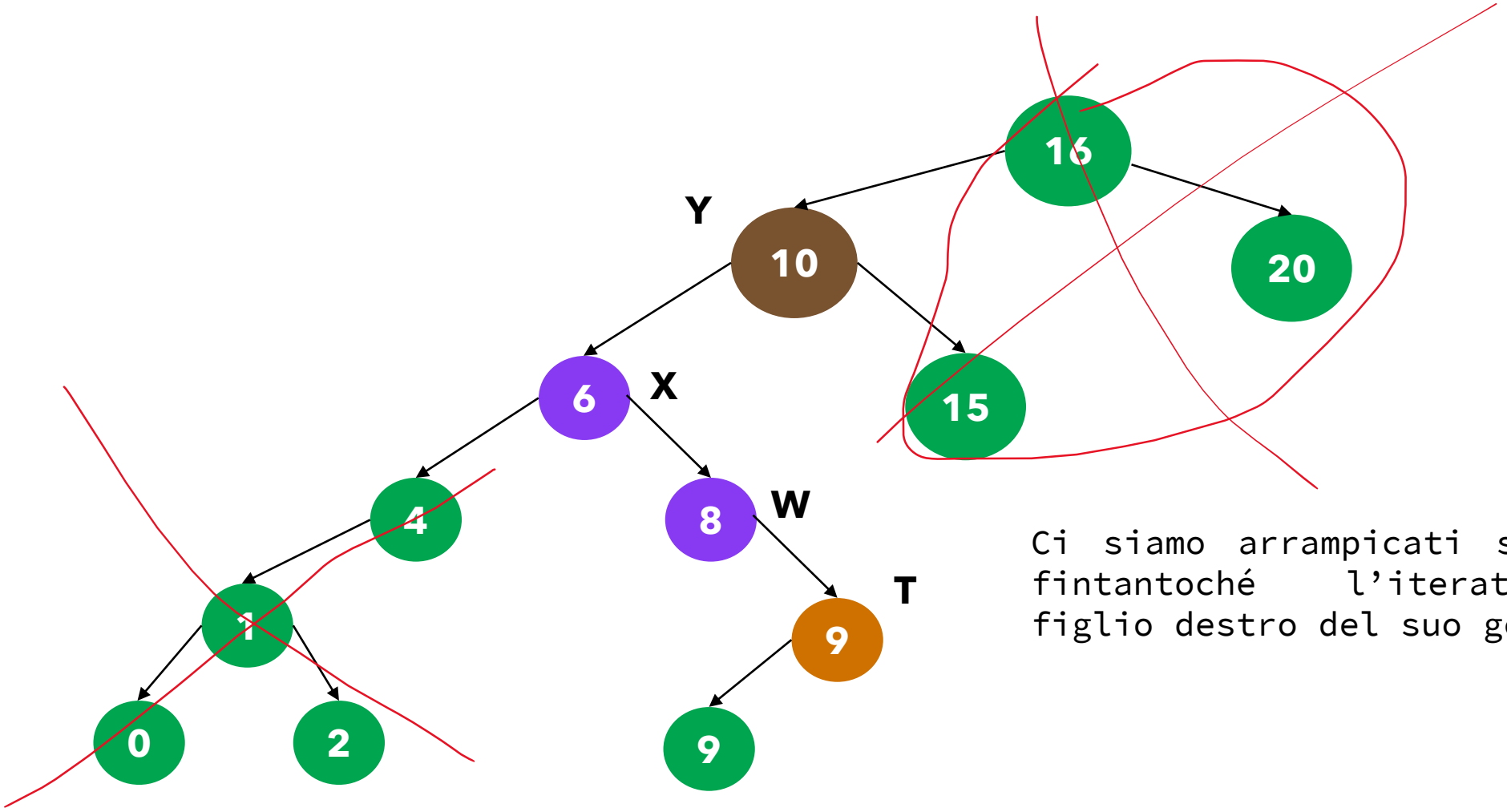
Il nodo successore di T è Y!



Y è il successore di T perché Y.key è sicuramente la più piccola tra le chiavi più grandi di T.key

Ricerca del nodo successore in un BST

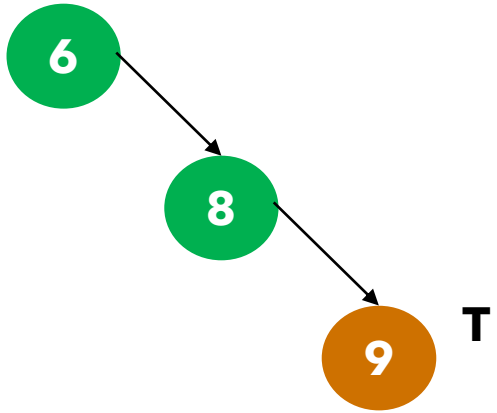
Il nodo successore di T è Y!



Ci siamo arrampicati sull'albero fintantoché l'iteratore era figlio destro del suo genitore

Ricerca del nodo successore in un BST

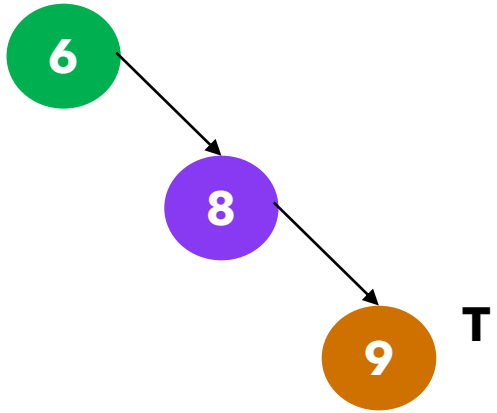
Caso limite:



Secondo il ragionamento precedente, dovremmo arrampicarci sull'albero fintantoché l'iteratore è il figlio destro di suo padre …

Ricerca del nodo successore in un BST

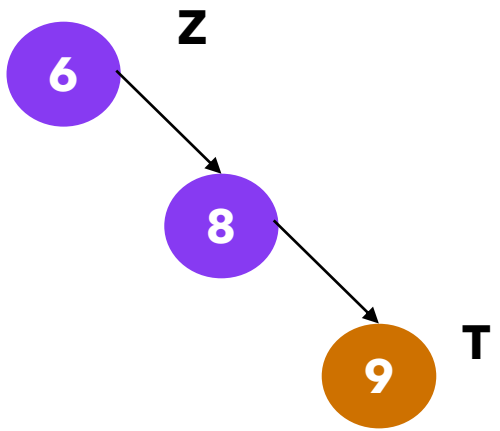
Caso limite:



Secondo il ragionamento precedente, dovremmo arrampicarci sull'albero fintantoché l'iteratore è il figlio destro di suo padre …

Ricerca del nodo successore in un BST

Caso limite:

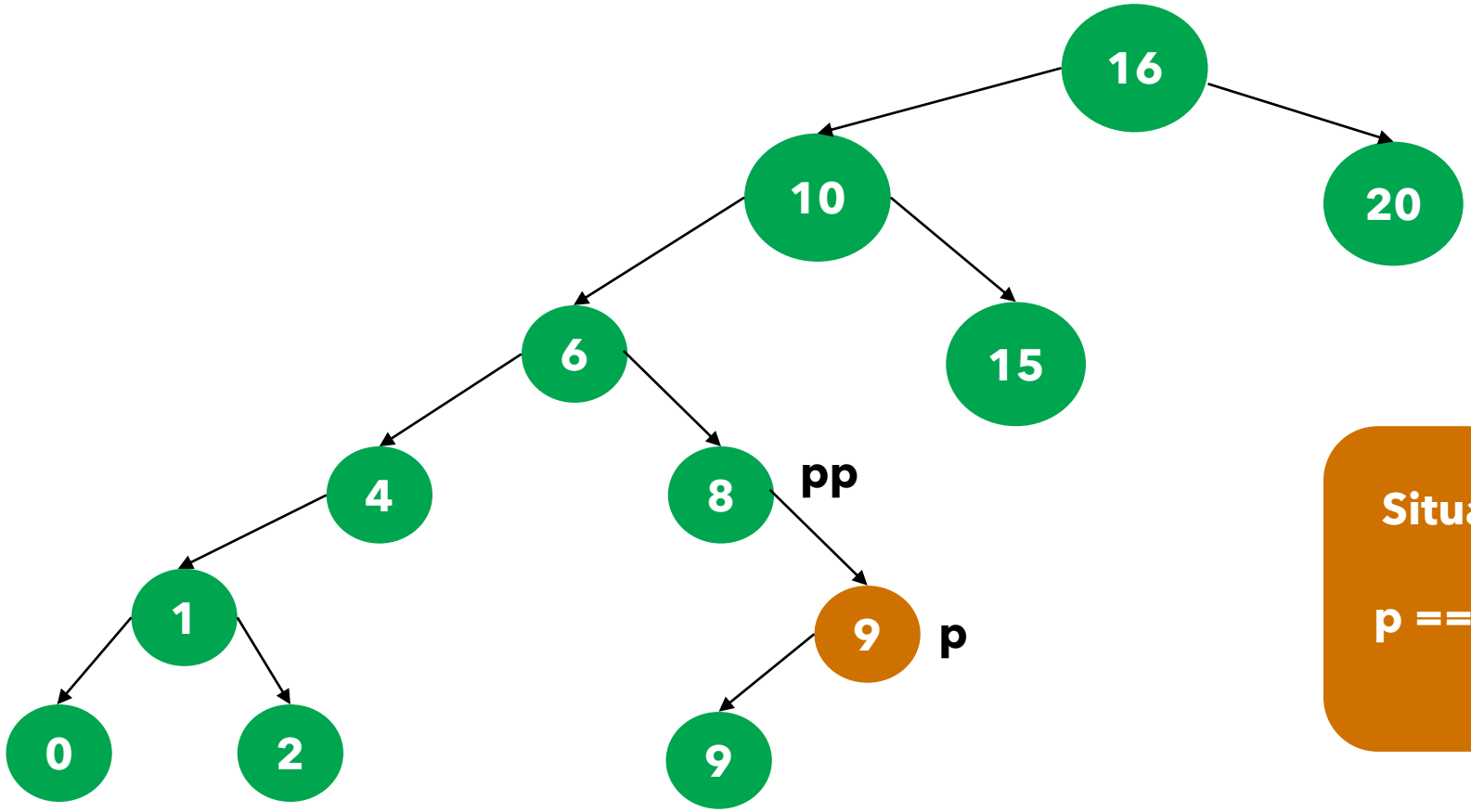


Il nodo Z non ha genitori (è la radice), quindi possiamo concludere che T non ha successori

Ricerca del nodo successore in un BST

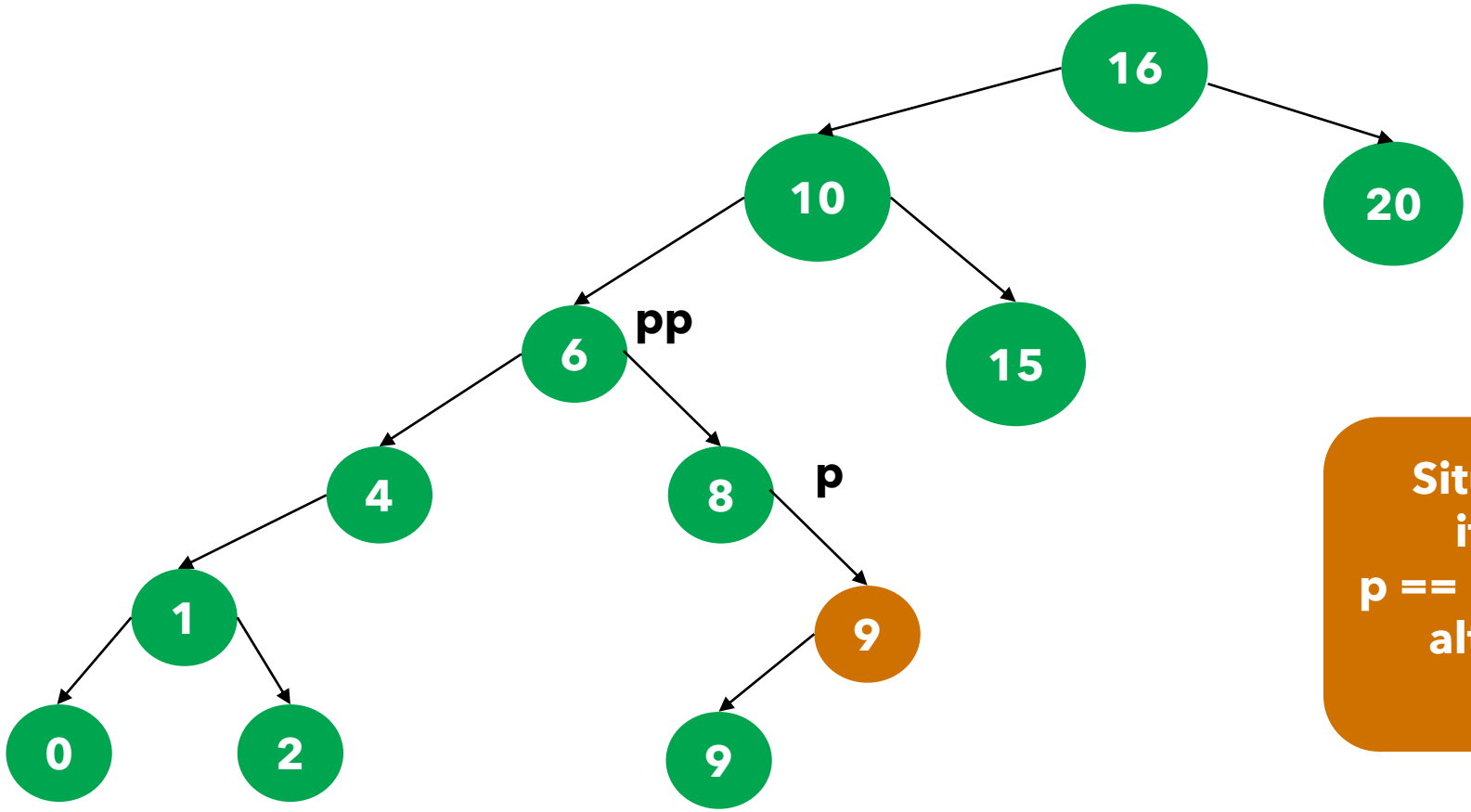
```
TREE_NODE *bst_successor(TREE_NODE *p) {
 if (!p) {
  return NULL;
 }
 if (p->right) {
  return bst_minimum_iterative(p->right);
 }

 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



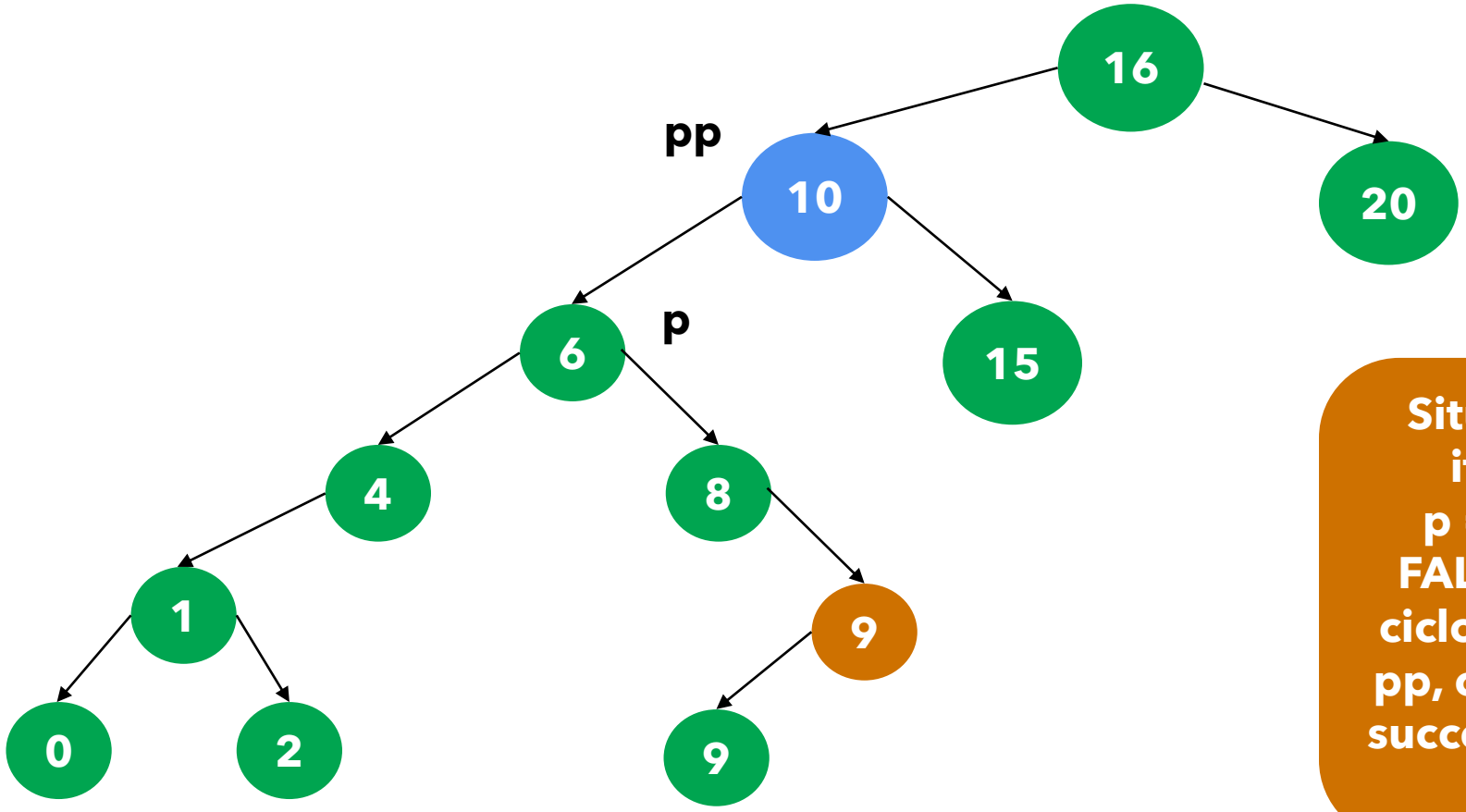
Situazione a fine iterazione 0;
p == pp->right è TRUE, altra iterazione

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 binario 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

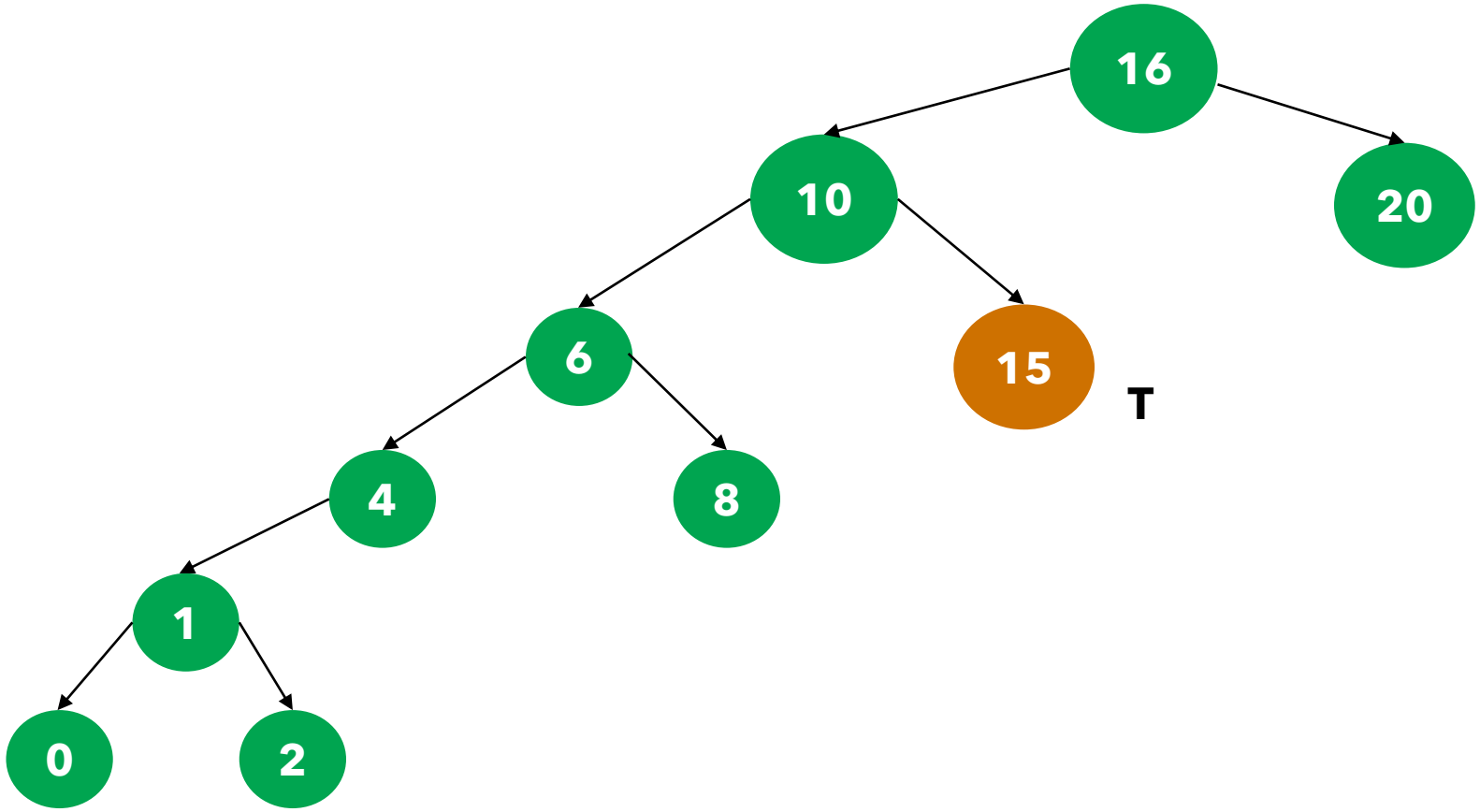
```
isSorted(node): returns true or false
    if node is nil:
        return TRUE
    if node.next is nil:
        return TRUE
    if node.key > node.next.key:
        return FALSE
    return isSorted(node.next)
```

Cancellazione di un nodo da un BST

- Cancellare un nodo da un BST non è proprio banale
- Ipotizziamo di lavorare con un linguaggio che prevede la deallocazione manuale della memoria (come C e C++)
- Nel nostro pseudocodice, per deallocare la memoria puntata da un puntatore T, scriveremo l'istruzione `delete T`
- Il problema consiste nel mantenere la proprietà dei BST anche dopo la cancellazione di un nodo
- Abbiamo a che fare con diversi casi
  - cancellazione di un nodo foglia
  - cancellazione di un nodo con un solo figlio
  - cancellazione di un nodo con 2 figli

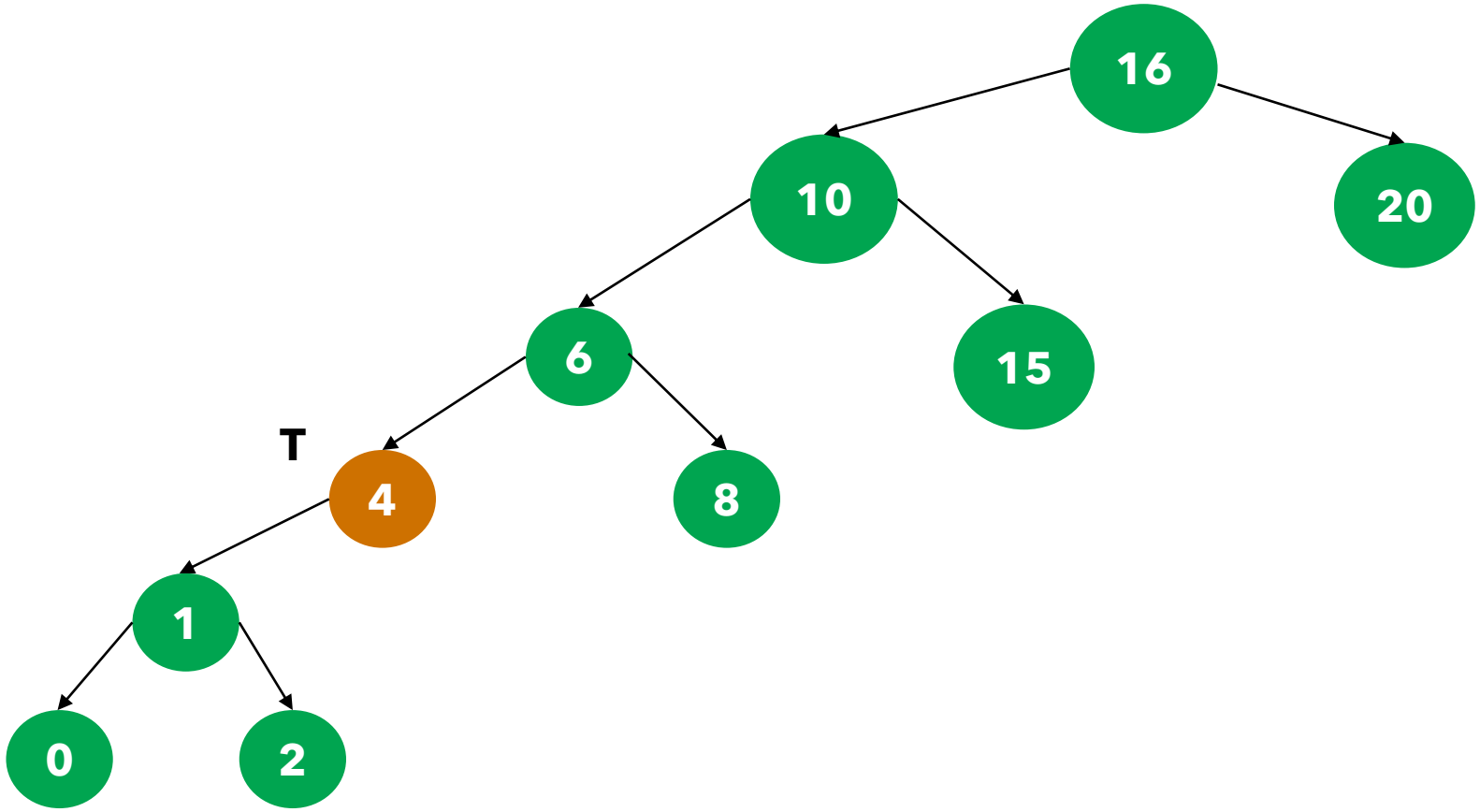
Cancellazione di un nodo da un BST

Dobbiamo cancellare un nodo T foglia. E' complicato? NO, basta deallocarlo con l'istruzione *delete T*



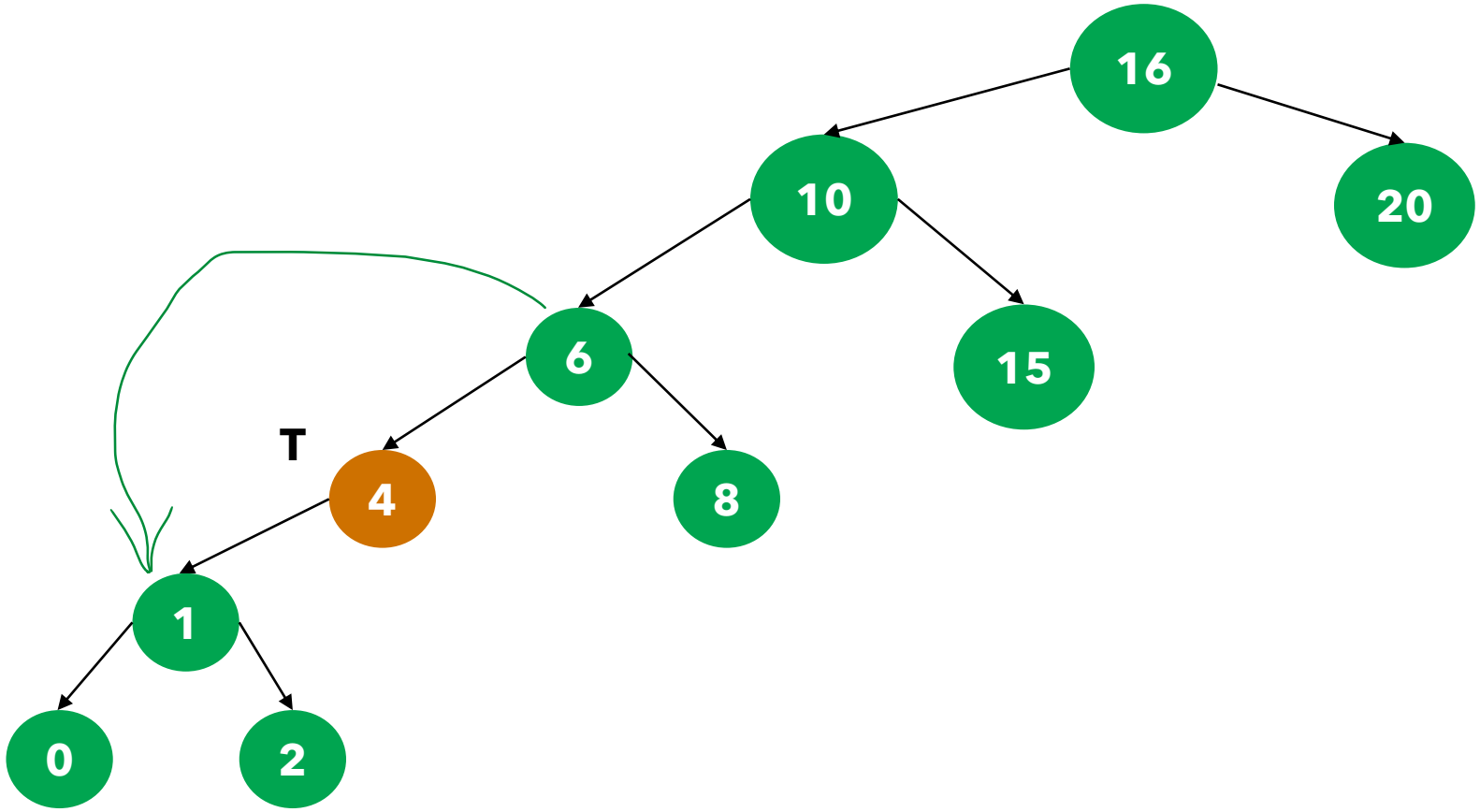
Cancellazione di un nodo da un BST

Dobbiamo cancellare un nodo T con un solo figlio. La questione si fa più complicata



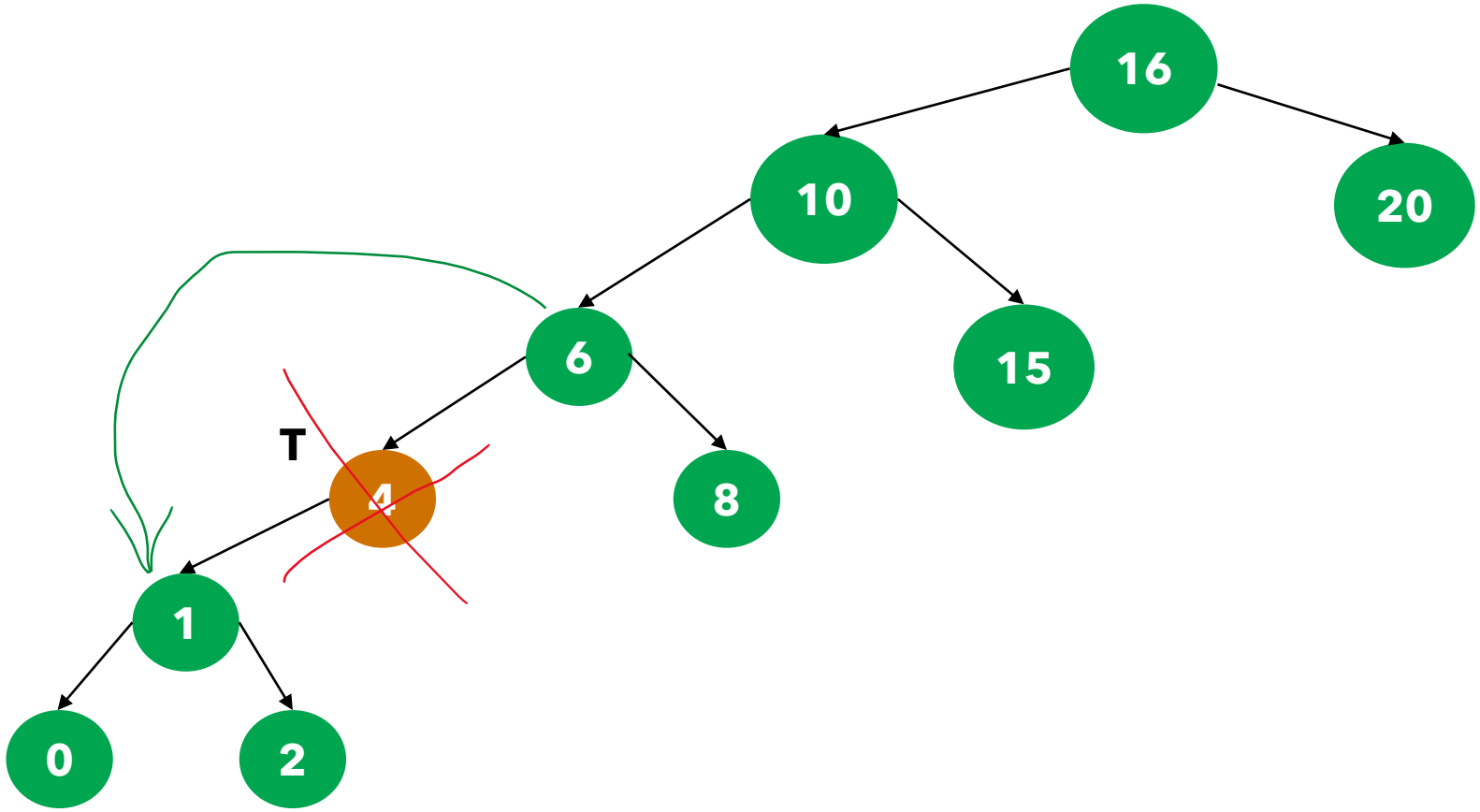
Cancellazione di un nodo da un BST

Dobbiamo far puntare T.parent all'unico figlio di T



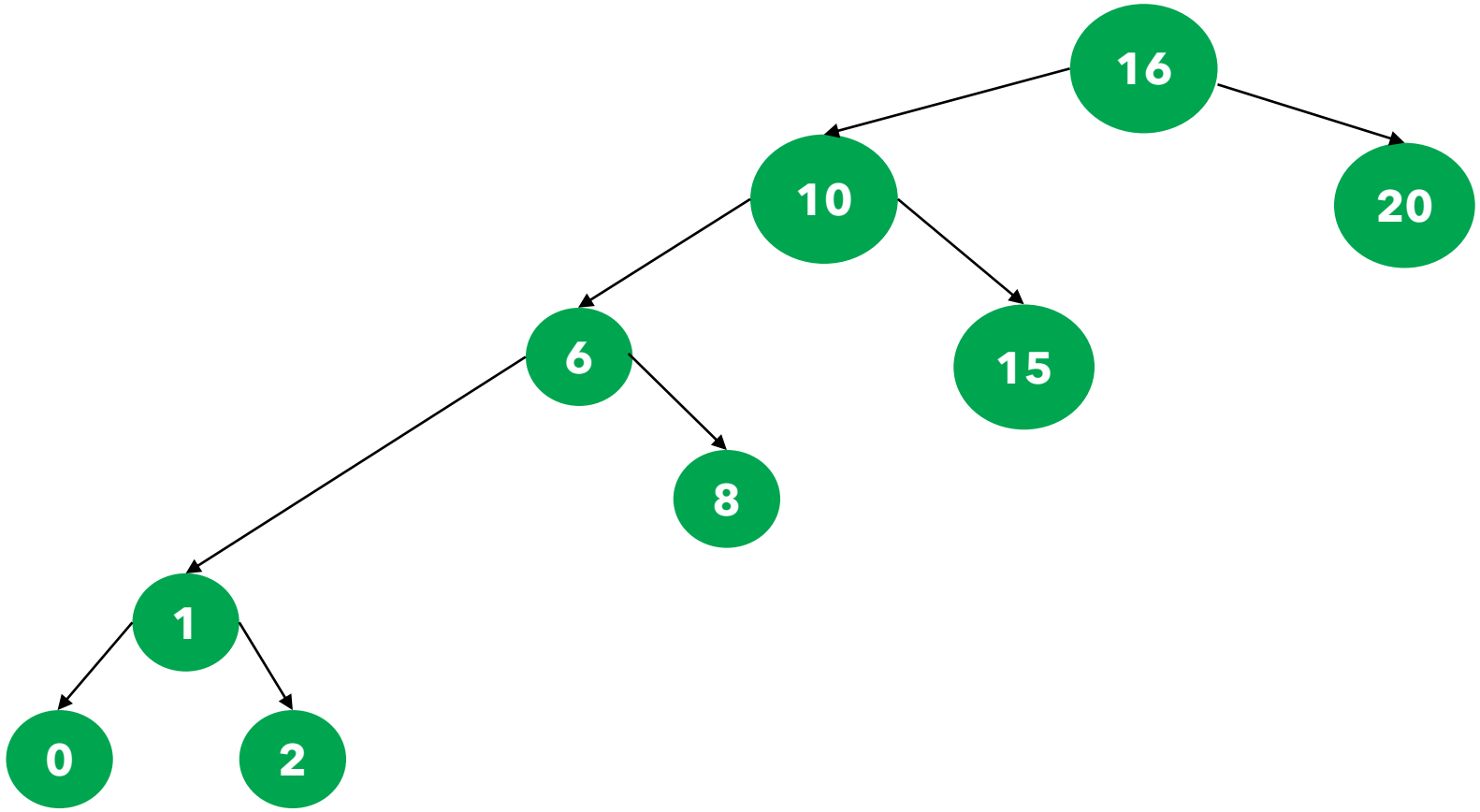
Cancellazione di un nodo da un BST

Poi dobbiamo ricordarci di deallocare T



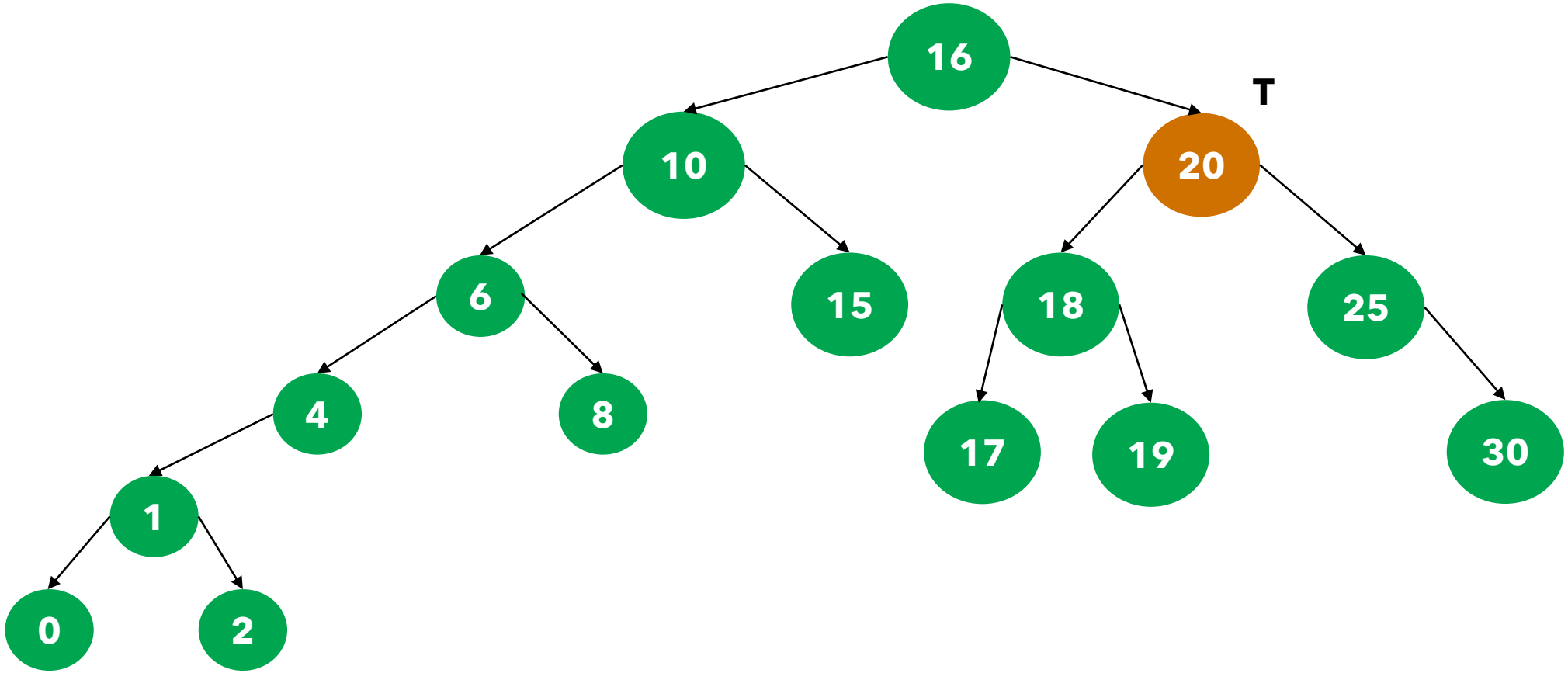
Cancellazione di un nodo da un BST

Il risultato è ancora un BST



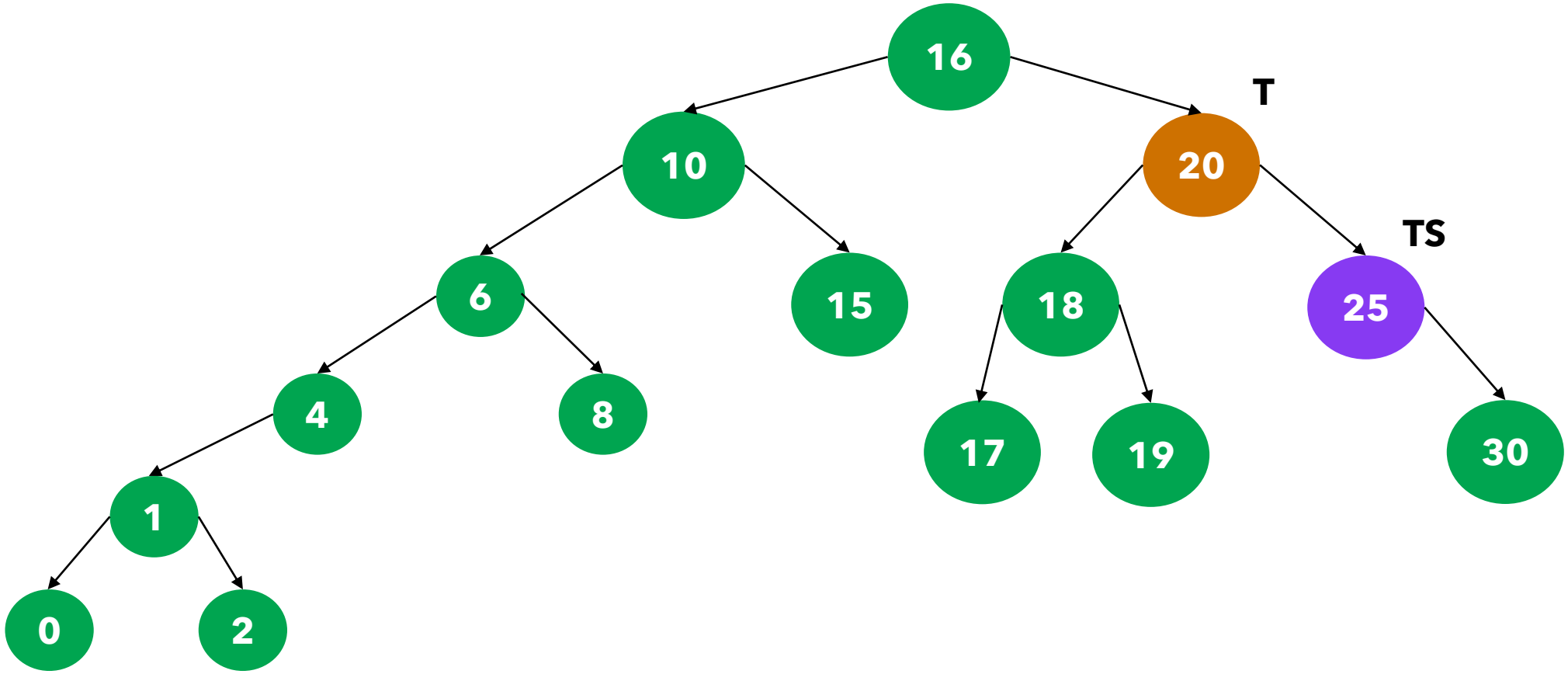
Cancellazione di un nodo da un BST

Consideriamo il nodo T: ha 2 figli



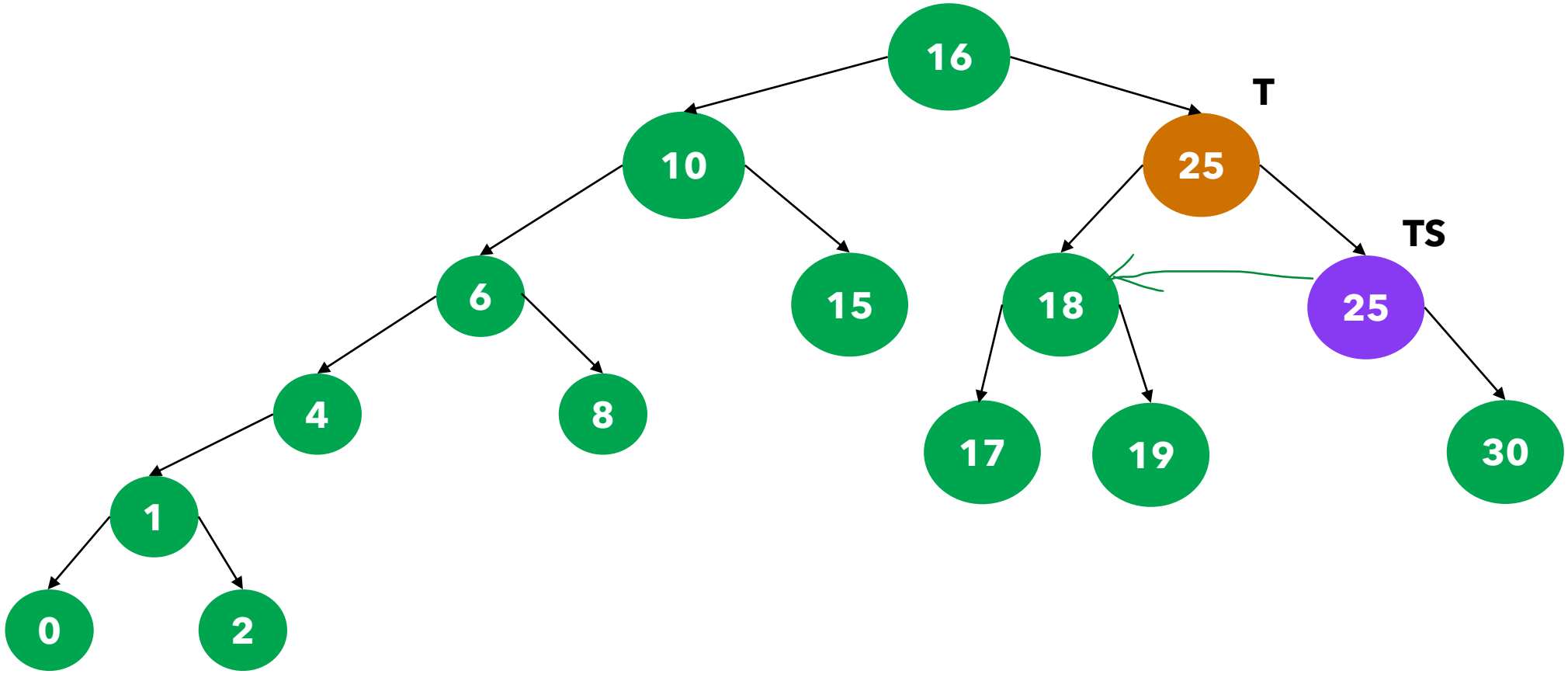
Cancellazione di un nodo da un BST

Cerchiamo il nodo successore di T. Se al posto di T mettiamo il nodo TS, la proprietà dei BST sarà ancora verificata



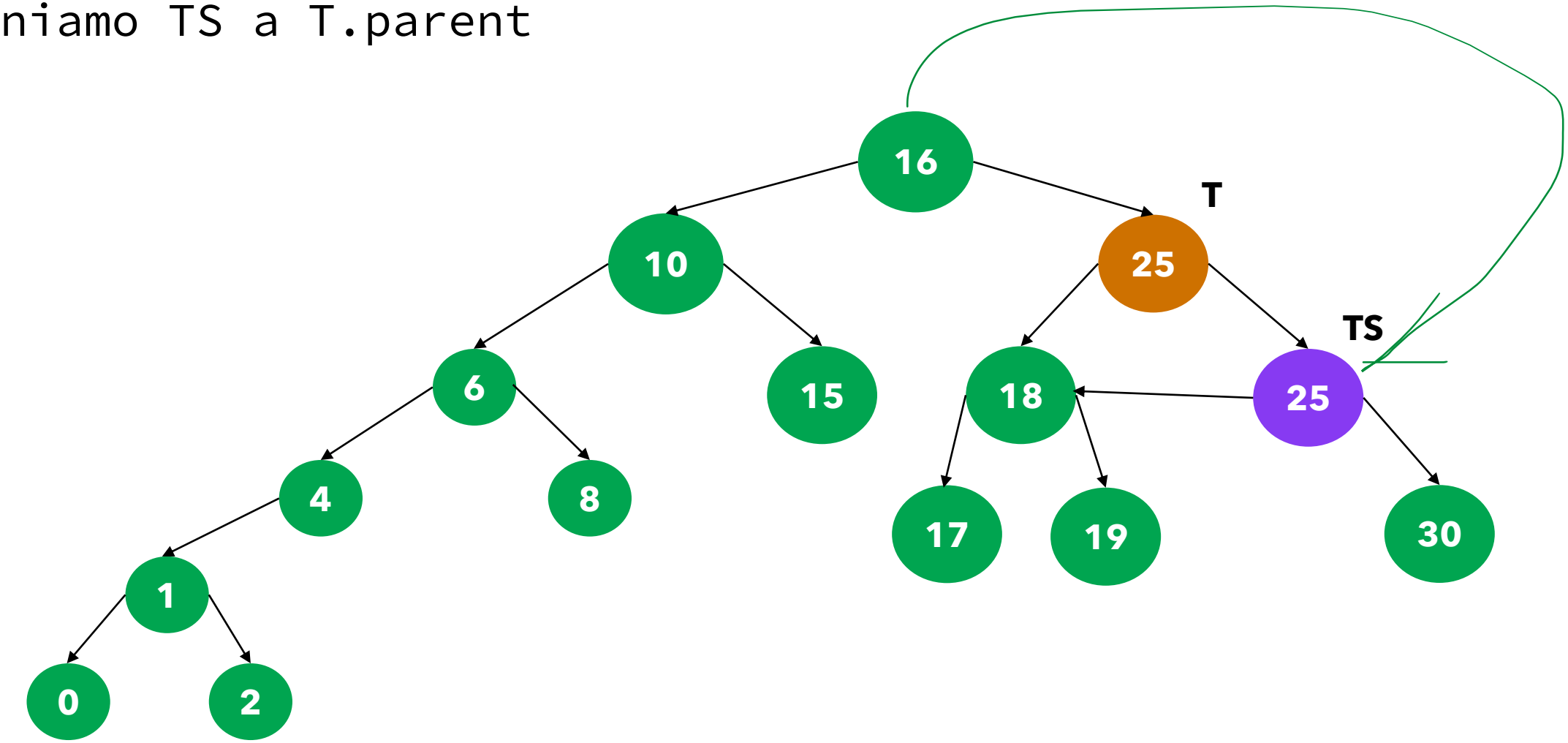
Cancellazione di un nodo da un BST

`Assegniamo T.left a TS.left`



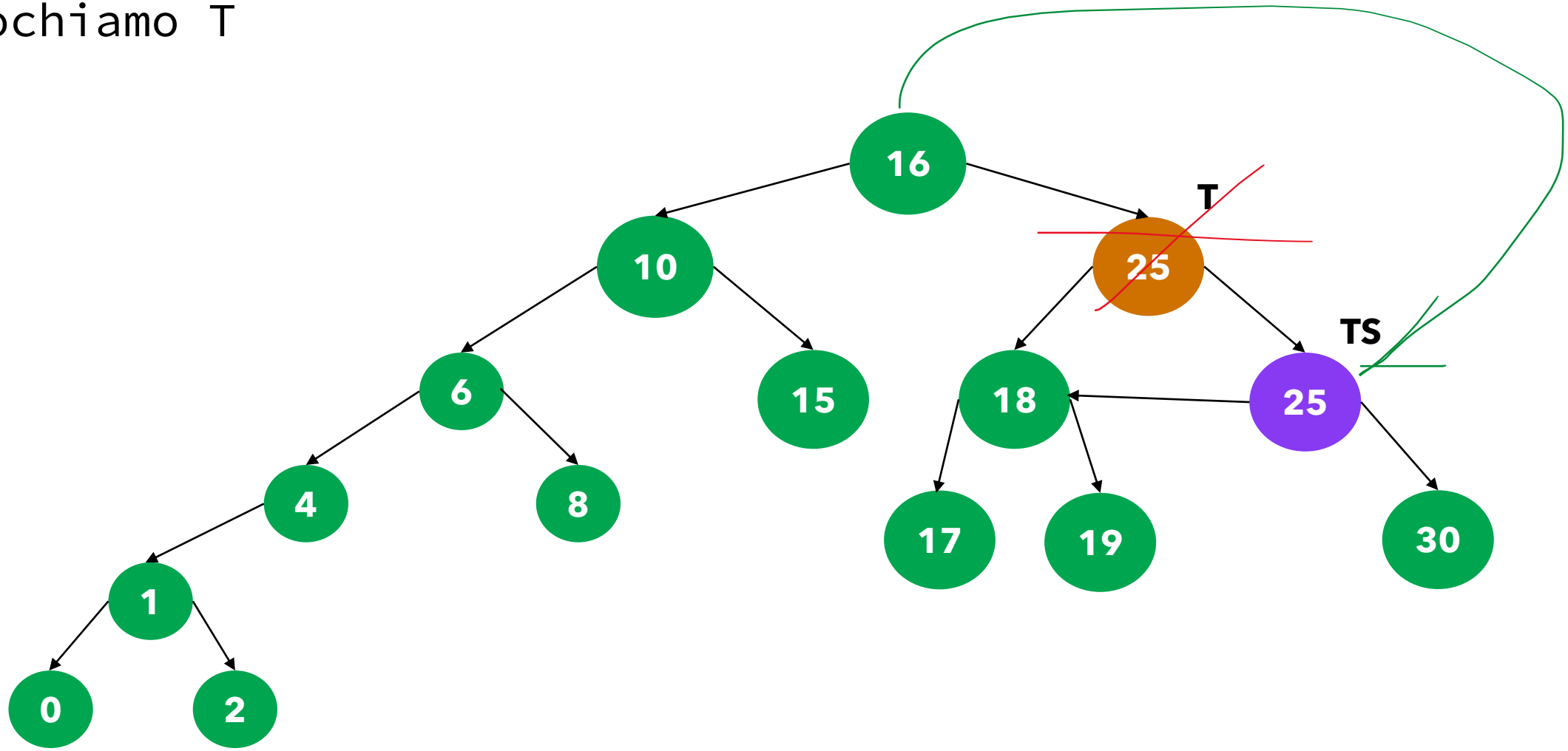
Cancellazione di un nodo da un BST

Assegniamo TS a T.parent



Cancellazione di un nodo da un BST

Deallochiamo T



Cancellazione di un nodo da un BST

Sistemiamo il disegno. Si vede subito che è ancora un BST!