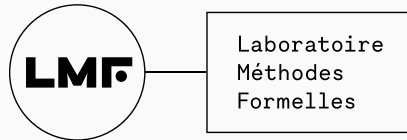


# Boucler la boucle du parcours de Morris

---

Arnaud Golfouse, Paul Patault



# Partage XOR mutation

```
unsafe fn f(x: *mut i32, y: *mut i32) {  
    *x = 1; *y = 2;  
  
}
```

# Partage XOR mutation

```
unsafe fn f(x: *mut i32, y: *mut i32) {  
    *x = 1; *y = 2;  
    assert!(*x == 1);  
}
```

# Partage XOR mutation

```
unsafe fn f(x: *mut i32, y: *mut i32) {  
    *x = 1; *y = 2;  
    assert!(*x == 1);  
}  
let x: *mut i32 = &mut 0;  
unsafe { f(x, x); }
```

# Partage XOR mutation

```
unsafe fn f(x: *mut i32, y: *mut i32) {  
    *x = 1; *y = 2;  
    assert!(*x == 1);  
}  
let x: *mut i32 = &mut 0;  
unsafe { f(x, x); }
```

---

```
fn f(x: &mut i32, y: &mut i32) {  
    *x = 1; *y = 2;  
    assert!(*x == 1);  
}  
let x: &mut i32 = &mut 0;  
f(x, x); // x erreur de compilation
```

# Partage XOR mutation

```
unsafe fn f(x: *mut i32, y: *mut i32) {  
    *x = 1; *y = 2;  
    assert!(*x == 1);  
}  
let x: *mut i32 = &mut 0;  
unsafe { f(x, x); }
```

Plus expressif

---

```
fn f(x: &mut i32, y: &mut i32) {  
    *x = 1; *y = 2;  
    assert!(*x == 1);  
}  
let x: &mut i32 = &mut 0;  
f(x, x); // x erreur de compilation
```

Vérification facile

Creusot  utilise « partage XOR mutation » :

```
#[requires(true)]
#[ensures(^x == 1 && ^y == 2)]
fn f(x: &mut i32, y: &mut i32) {
    *x = 1; *y = 2;
    assert!(*x == 1);
}
```

L'algorithme de Morris :

- Un parcours d'arbre en espace **constant**.
- Utilise des **pointeurs**.

⇒ Vérifiable avec Creusot ?

# Algorithme de Morris

Arbre binaire avec pointeurs :

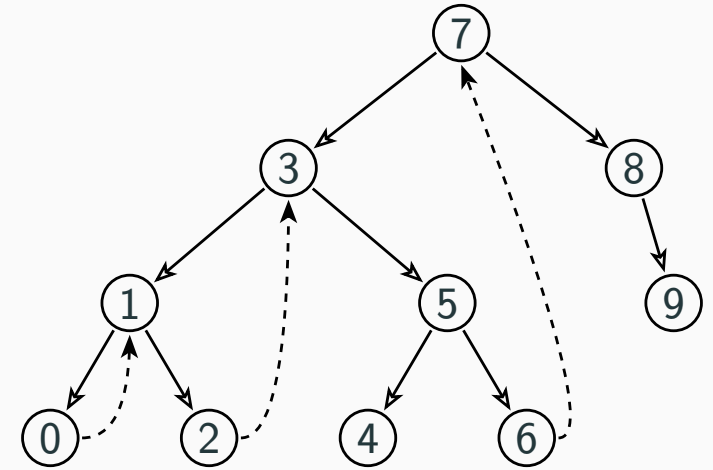
```
struct Tree<T> {  
    root: *mut Node<T>,  
}
```

```
struct Node<T> {  
    left: *mut Node<T>,  
    right: *mut Node<T>,  
    value: T,  
}
```

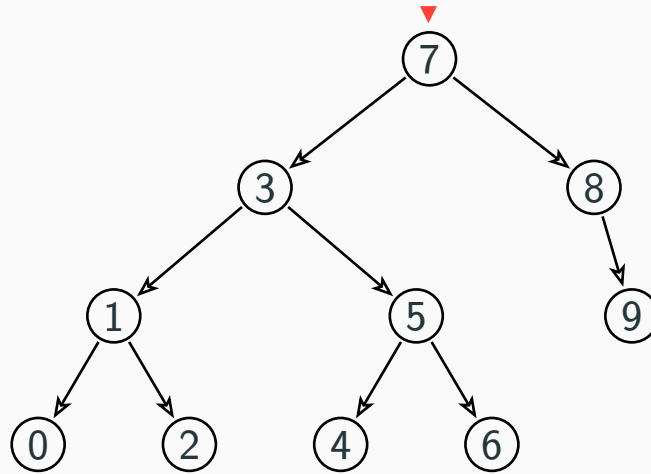
# Algorithme de Morris

Arbre binaire avec pointeurs :

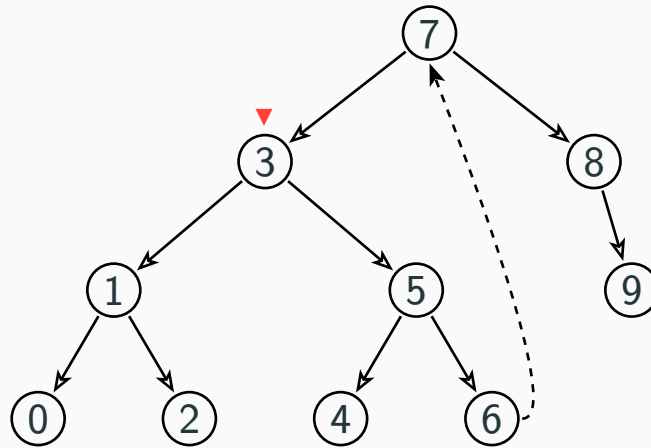
```
struct Tree<T> {  
    root: *mut Node<T>,  
}  
  
struct Node<T> {  
    left: *mut Node<T>,  
    right: *mut Node<T>,  
    value: T,  
}
```



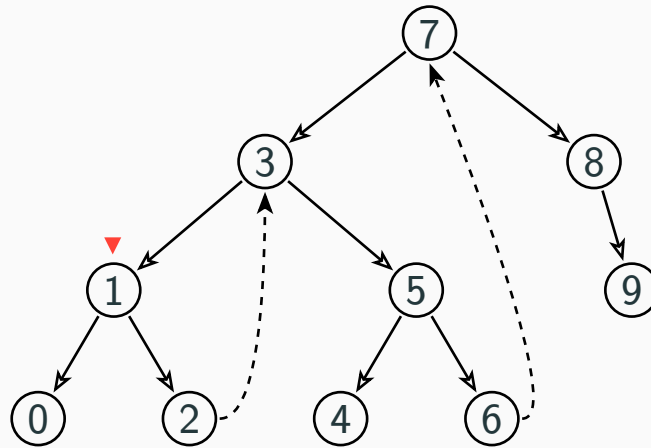
# Algorithme de Morris



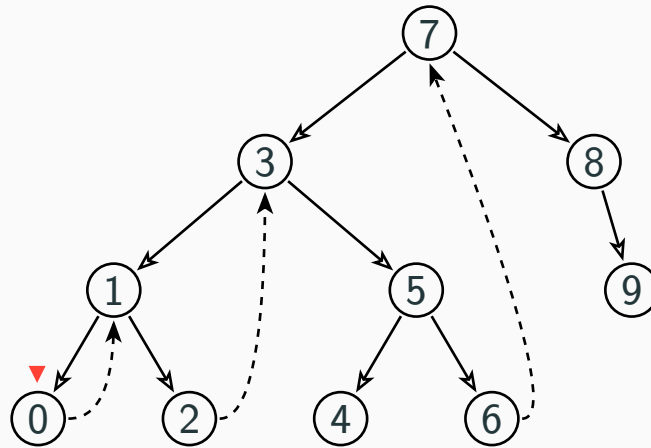
# Algorithme de Morris



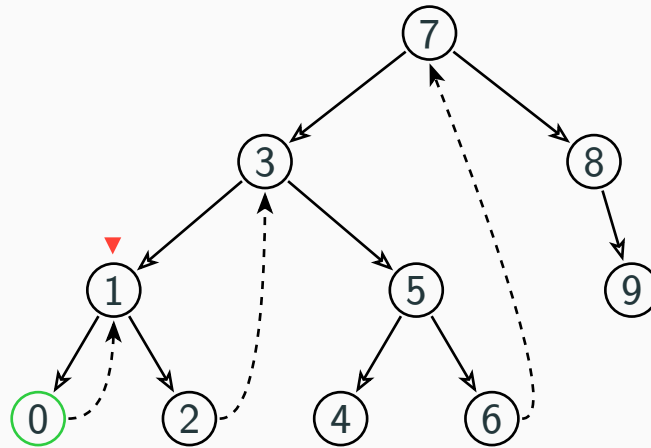
# Algorithme de Morris



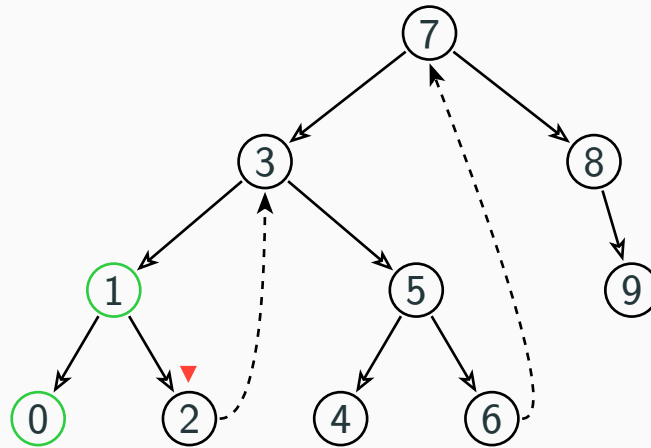
# Algorithme de Morris



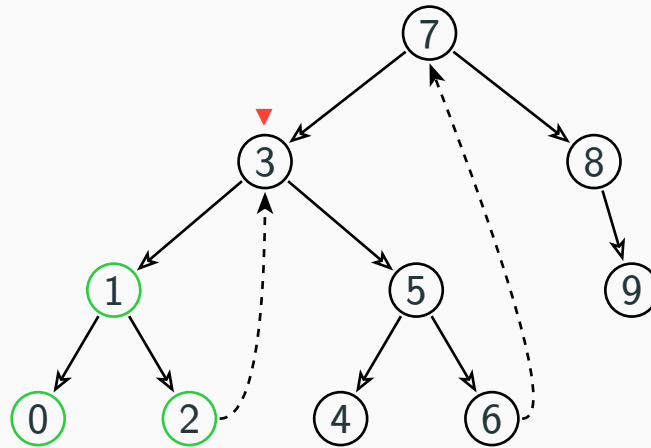
# Algorithme de Morris



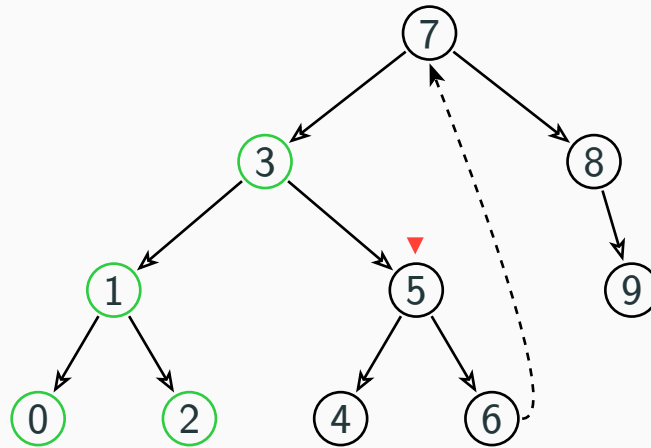
# Algorithme de Morris



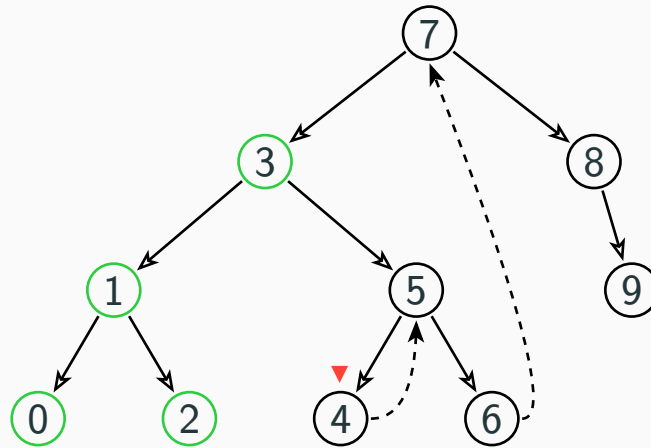
# Algorithme de Morris



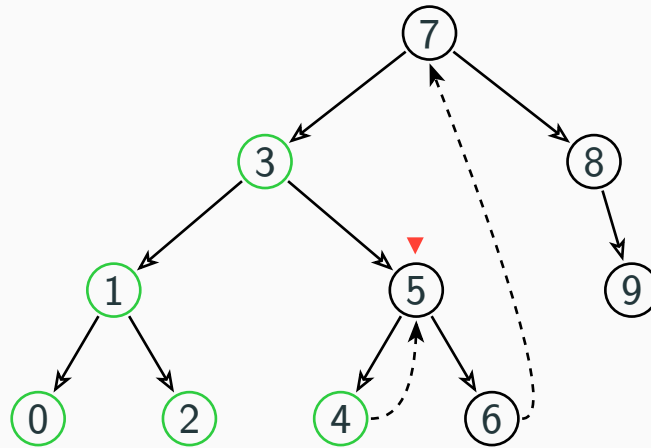
# Algorithme de Morris



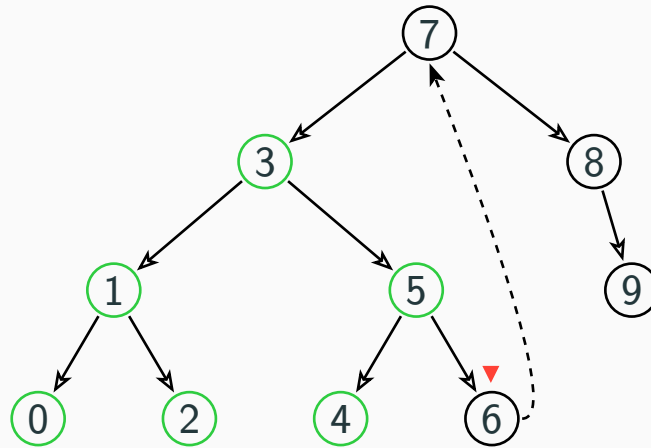
# Algorithme de Morris



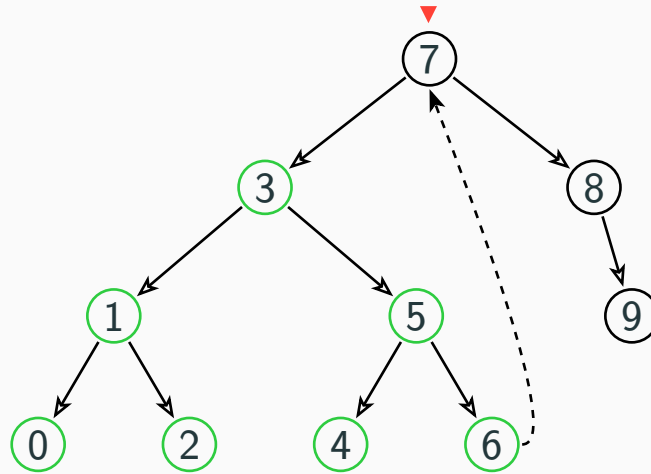
# Algorithme de Morris



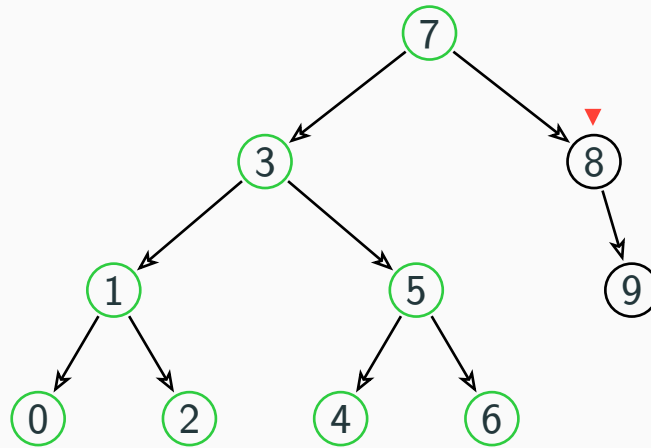
# Algorithme de Morris



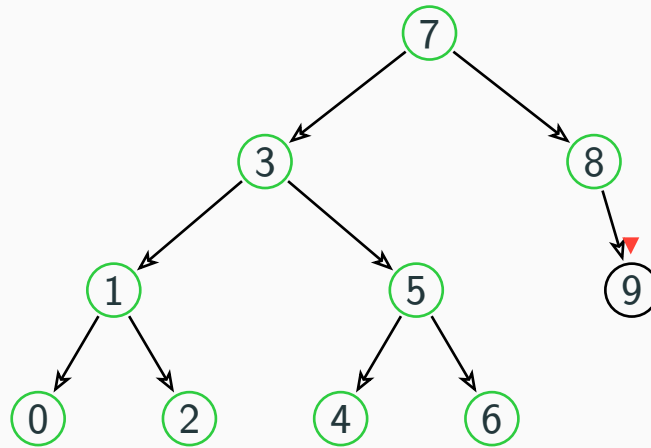
# Algorithme de Morris



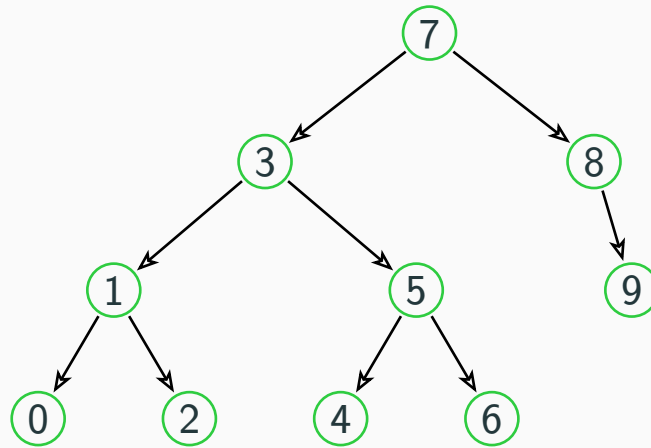
# Algorithme de Morris



# Algorithme de Morris



# Algorithme de Morris

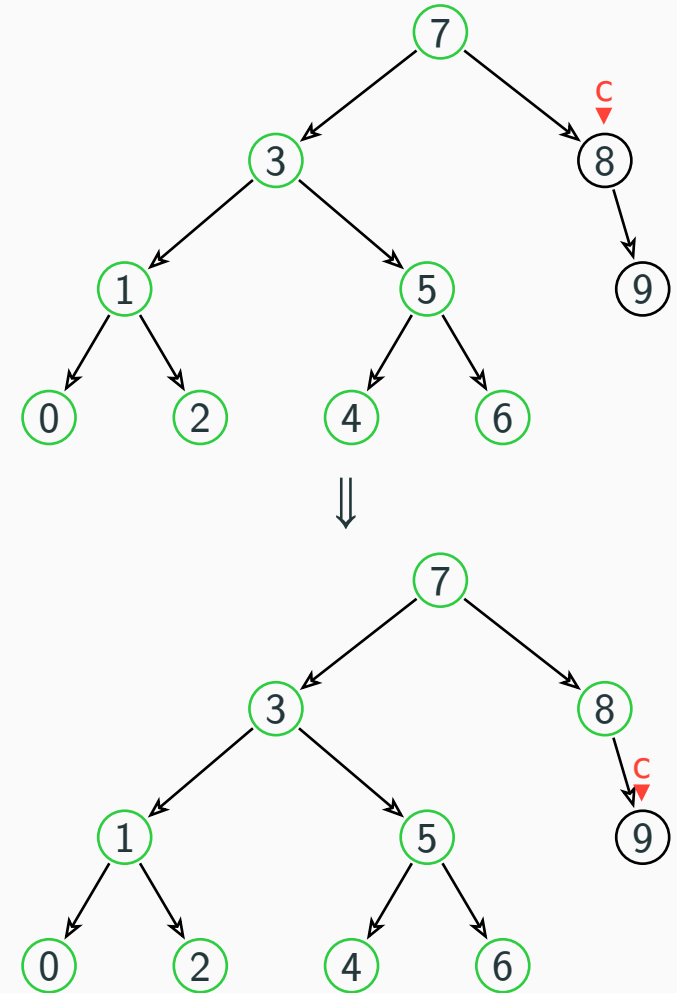


# Algorithme de Morris

```
unsafe fn morris<T>(t: Tree<T>, visit: fn(&mut T)) {
    let mut c = t.root;
'L: while !c.is_null() {
    if !(*c).left.is_null() {
        let mut p = (*c).left;
        loop {
            if (*p).right.is_null() {
                (*p).right = c; c = (*c).left;
                continue 'L;
            } else if (*p).right == c {
                (*p).right = null_mut();
                break;
            }
            p = (*p).right;
        }
    }
    visit(&mut (*c).value); c = (*c).right;
}
}
```

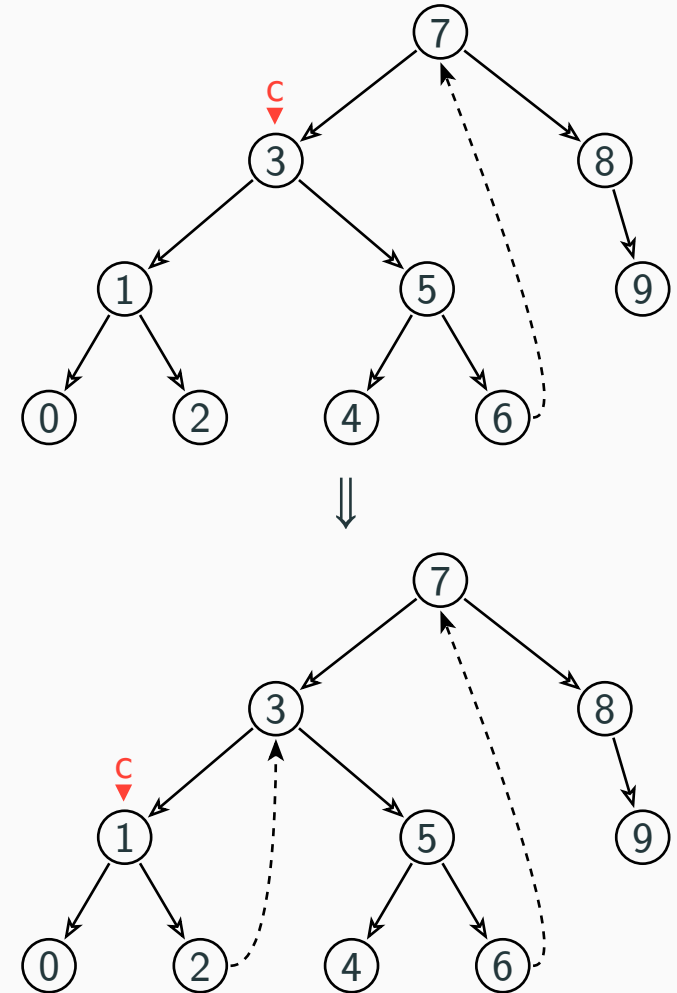
# Algorithme de Morris

```
unsafe fn morris<T>(t: Tree<T>, visit: fn(&mut T)) {  
    let mut c = t.root;  
    'L: while !c.is_null() {  
        if !(*c).left.is_null() {  
            let mut p = (*c).left;  
            loop {  
                if (*p).right.is_null() {  
                    (*p).right = c; c = (*c).left;  
                    continue 'L;  
                } else if (*p).right == c {  
                    (*p).right = null_mut();  
                    break;  
                }  
                p = (*p).right;  
            }  
        }  
        visit(&mut (*c).value); c = (*c).right;  
    }  
}
```



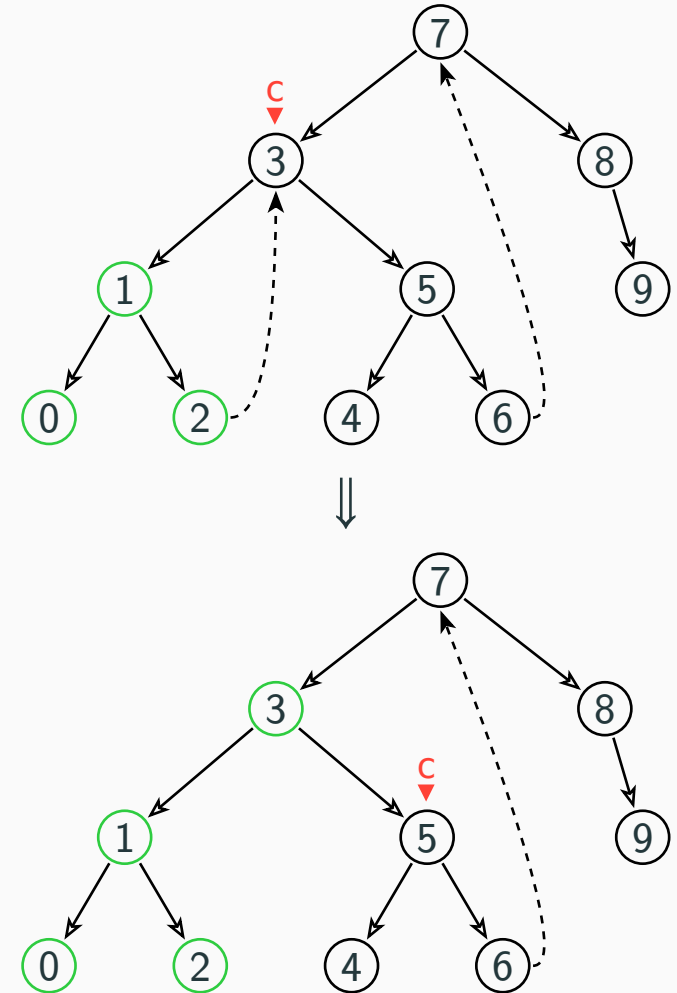
# Algorithme de Morris

```
unsafe fn morris<T>(t: Tree<T>, visit: fn(&mut T)) {  
    let mut c = t.root;  
    'L: while !c.is_null() {  
        if !(*c).left.is_null() {  
            let mut p = (*c).left;  
            loop {  
                if (*p).right.is_null() {  
                    (*p).right = c; c = (*c).left;  
                    continue 'L;  
                } else if (*p).right == c {  
                    (*p).right = null_mut();  
                    break;  
                }  
                p = (*p).right;  
            }  
        }  
        visit(&mut (*c).value); c = (*c).right;  
    }  
}
```



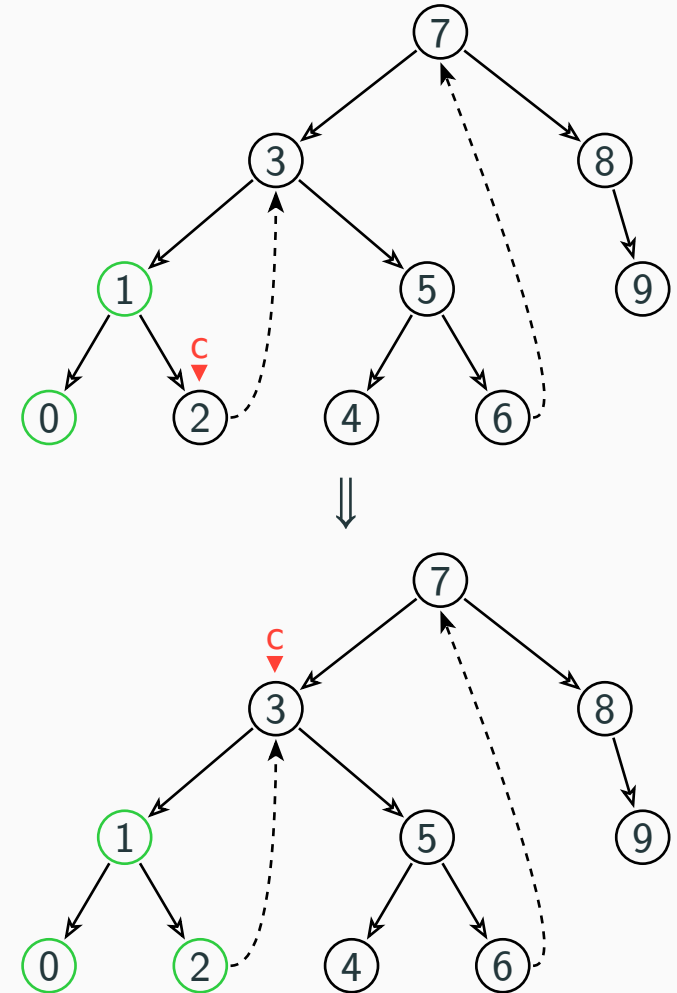
# Algorithme de Morris

```
unsafe fn morris<T>(t: Tree<T>, visit: fn(&mut T)) {  
    let mut c = t.root;  
    'L: while !c.is_null() {  
        if !(*c).left.is_null() {  
            let mut p = (*c).left;  
            loop {  
                if (*p).right.is_null() {  
                    (*p).right = c; c = (*c).left;  
                    continue 'L;  
                } else if (*p).right == c {  
                    (*p).right = null_mut();  
                    break;  
                }  
                p = (*p).right;  
            }  
        }  
        visit(&mut (*c).value); c = (*c).right;  
    }  
}
```



# Algorithme de Morris

```
unsafe fn morris<T>(t: Tree<T>, visit: fn(&mut T)) {  
    let mut c = t.root;  
    'L: while !c.is_null() {  
        if !(*c).left.is_null() {  
            let mut p = (*c).left;  
            loop {  
                if (*p).right.is_null() {  
                    (*p).right = c; c = (*c).left;  
                    continue 'L;  
                } else if (*p).right == c {  
                    (*p).right = null_mut();  
                    break;  
                }  
                p = (*p).right;  
            }  
        }  
        visit(&mut (*c).value); c = (*c).right;  
    }  
}
```



Pointeurs partagés  $\implies$  preuve compliquée !

Pointeurs partagés  $\implies$  preuve compliquée !

Idée :

- Le code *exécuté* est inchangé ;
- La vérification raisonne sur un tableau (arbre « à plat ») ;
- Du **code fantôme** relie les deux.

Creusot interdit de lire/écrire un pointeur directement.

Creusot interdit de lire/écrire un pointeur directement.

À la place : `Ghost<Seq<PtrOwn<T>>>`

- `Ghost<X>` : effacé, typé comme `X`
- `Seq` : type des séquences fantômes

Creusot interdit de lire/écrire un pointeur directement.

À la place : `Ghost<Seq<PtrOwn<T>>>`

- `Ghost<X>` : effacé, typé comme `X`
- `Seq` : type des séquences fantômes

Et `PtrOwn` ?

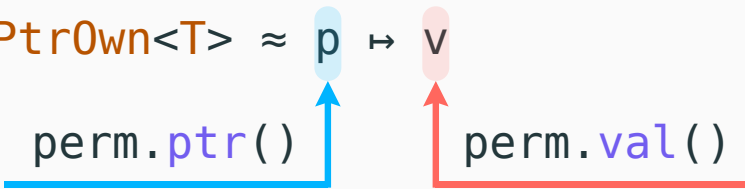
perm: `PtrOwn<T>`  $\approx$  `p`  $\mapsto$  `v`

perm: `PtrOwn<T>`  $\approx$  `p`  $\mapsto$  `v`  
spécifications : `perm.ptr()` `perm.val()`

The diagram illustrates the relationship between a `PtrOwn<T>` object and its internal state. The text `perm: PtrOwn<T>  $\approx$  p  $\mapsto$  v` shows that the object is approximately equal to a pointer `p` and maps to a value `v`. Below this, the specifications `perm.ptr()` and `perm.val()` are shown. A blue arrow points from `perm.ptr()` to `p`, and a red arrow points from `perm.val()` to `v`.

# Gestion des pointeurs

perm: `PtrOwn<T>`  $\approx$  `p`  $\mapsto$  `v`  
spécifications : `perm.ptr()` `perm.val()`



code :

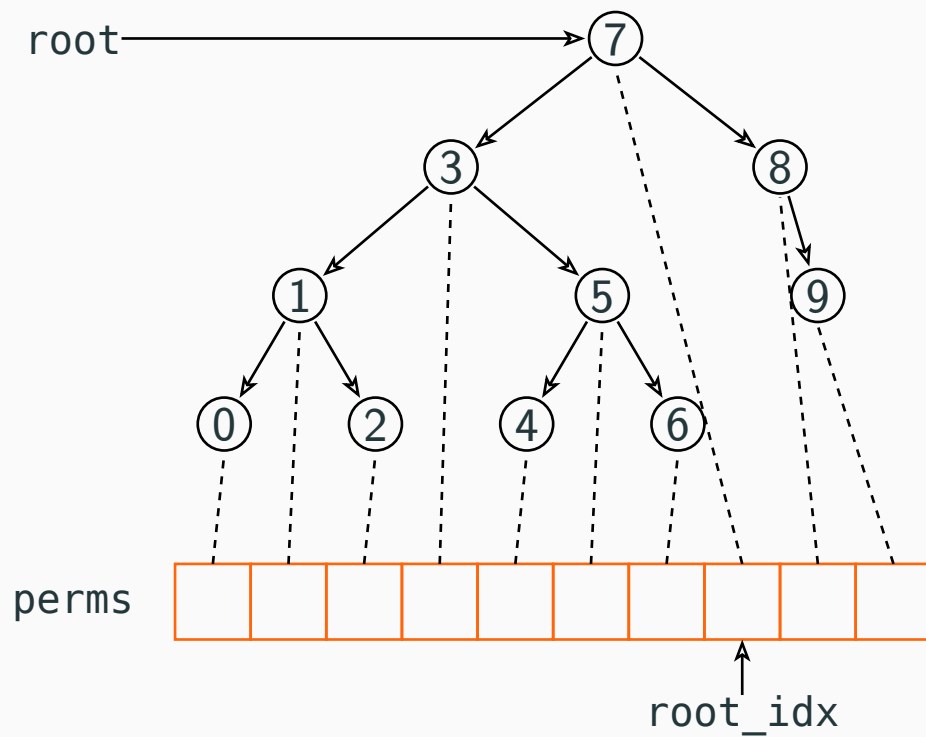
Création	<code>let (p, perm): (*mut T, Ghost&lt;PtrOwn&lt;T&gt;&gt;) = PtrOwn::new(v);</code>
Lecture	<code>let x: &amp;T = PtrOwn::as_ref(p, perm.borrow());</code>
Écriture	<code>let x: &amp;mut T = PtrOwn::as_mut(p, perm.borrow_mut());</code>

# Preuve

```
struct Tree<T> {  
  root: *mut Node<T>,  
  root_idx: Ghost<Int>,  
  perms: Ghost<Seq<PtrOwn<Node<T>>>>,  
  tree_info: Ghost<TreeInfo>,  
}
```

```
struct Node<T> {  
  left: *mut Node<T>,  
  right: *mut Node<T>,  
  value: T,  
}
```

```
fn left(tree_info: &TreeInfo, idx: Int) -> Int { /* ... */ }  
fn right(tree_info: &TreeInfo, idx: Int) -> Int { /* ... */ }
```



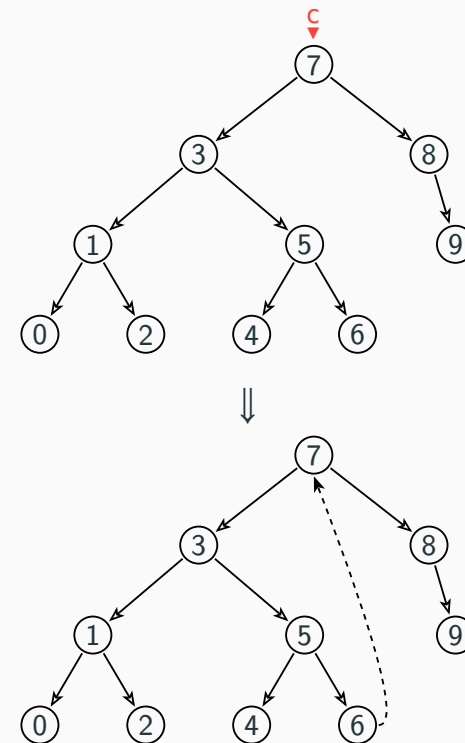
# Preuve

```
c: *mut Node<T>
```

```
// ...
```

```
let mut p = (*c).left;
```

```
loop {  
    if ... { }  
    p = (*p).right;  
}
```

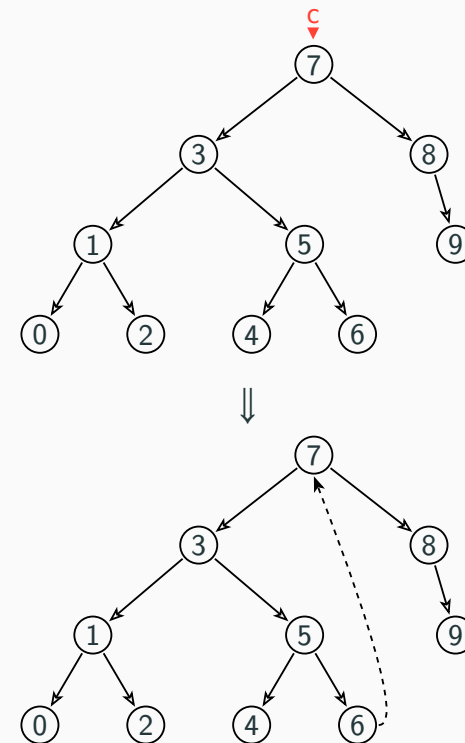


# Preuve

```
c: *mut Node<T>, c_idx: Ghost<Int>
// ...

let mut p = (*c).left;
let mut p_idx = ghost!(left(&t.tree_info, *c_idx));

loop {
  if ... { }
  p = (*p).right;
  p_idx = ghost!(right(&t.tree_info, *p_idx));
}
```



# Preuve

```
c: *mut Node<T>, c_idx: Ghost<Int>
```

```
// ...
```

```
let mut p = PtrOwn::as_ref(c, ghost!(&t.perms[c_idx])).left;
```

```
let mut p_idx = ghost!(left(&t.tree_info, *c_idx));
```

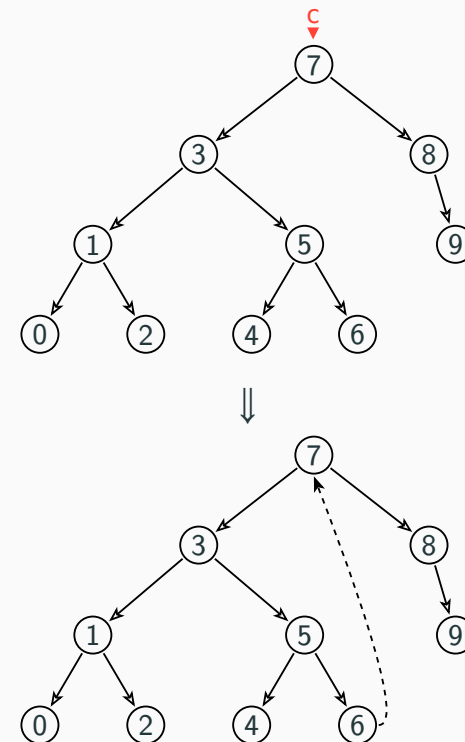
```
loop {
```

```
  if ... { }
```

```
  p = PtrOwn::as_ref(p, ghost!(&t.perms[*p_idx])).right;
```

```
  p_idx = ghost!(right(&t.tree_info, *p_idx));
```

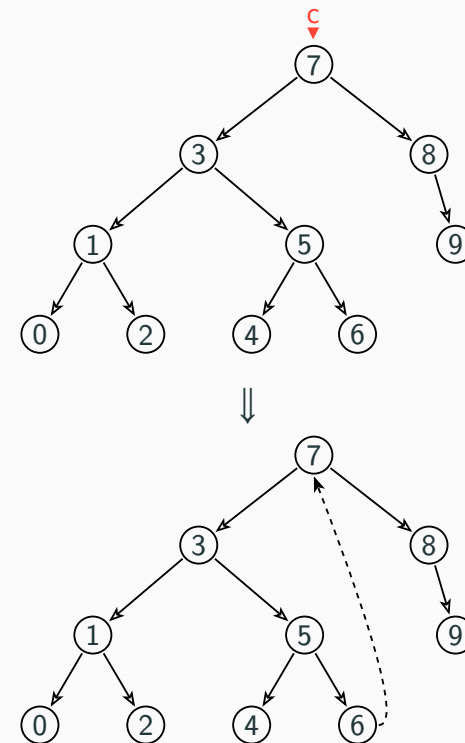
```
}
```



# Preuve

```
c: *mut Node<T>, c_idx: Ghost<Int>
// ...

let mut p = PtrOwn::as_ref(c, ghost!(&t.perms[c_idx])).left;
let mut p_idx = ghost!(left(&t.tree_info, *c_idx));
#[invariant(t.perms[*p_idx].ptr() == p)]
loop {
    if ... { }
    p = PtrOwn::as_ref(p, ghost!(&t.perms[*p_idx])).right;
    p_idx = ghost!(right(&t.tree_info, *p_idx));
}
```



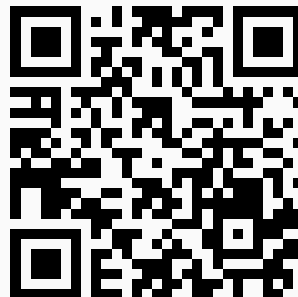
# Conclusion

On a une preuve de l'algorithme de Morris, **sans modification du code exécuté**.

Permi par :

- Le code fantôme
- Le support des pointeurs

Preuve disponible en suivant le pointeur : <https://zenodo.org/records/17914344>



# Spécification de morris

On inverse le contrôle :

```
struct TreeIter<T> {
    c: *mut Node<T>,
    c_idx: Ghost<Int>,
    perms: Ghost<Seq<PtrOwn<Node<T>>>>,
    info: Ghost<TreeInfo>,
}

#[ensures(match result {
    None => (*iter)@ == Seq::empty() && *iter == ^iter,
    Some(x) => (*iter)@ == (^iter)@.push_front(*x),
})]
fn next<T>(iter: &mut TreeIter<T>) -> Option<&mut T>;
```

# Spécification de `as_mut`

```
#[requires(ptr == perm.ptr())]  
#[ensures((^perm).ptr() == perm.ptr())]  
#[ensures(*result == perm.val())]  
#[ensures((^perm).val() == ^result)]  
unsafe fn as_mut(ptr: *mut T, perm: Ghost<&mut PtrOwn<T>>) -> &mut T;
```

```
// perm ≈ p ↦ v  
let bor = PtrOwn::as_mut(ptr, perm.borrow_mut());  
*bor = *bor + 1;  
// perm ≈ p' ↦ v'
```

# Spécification de `as_mut`

```
#[requires(ptr == perm.ptr())]  
#[ensures((^perm).ptr() == perm.ptr())]  
#[ensures(*result == perm.val())]  
#[ensures((^perm).val() == ^result)]  
unsafe fn as_mut(ptr: *mut T, perm: Ghost<&mut PtrOwn<T>>) -> &mut T;
```

```
// perm ≈ p ↦ v  
let bor = PtrOwn::as_mut(ptr, perm.borrow_mut());  
*bor = *bor + 1;  
// perm ≈ p' ↦ v'
```

