Mesurer la hauteur d'un arbre

Jean-Christophe Filliâtre
CNRS

JFLA

Gruissan, 31 janvier 2020

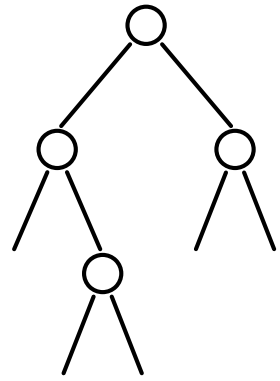
H

comme

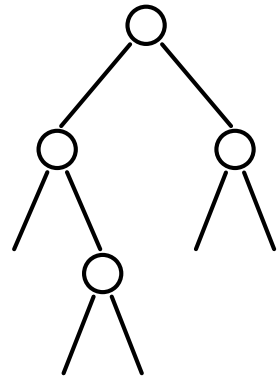
hauteur d'un arbre

```
type tree = E | N of tree * tree
```

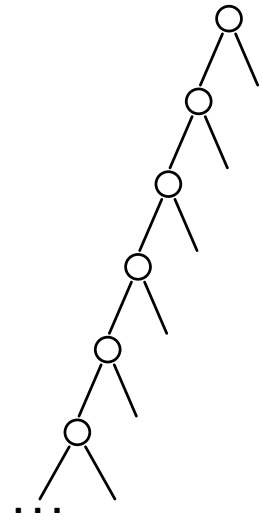
```
type tree = E | N of tree * tree
```



```
type tree = E | N of tree * tree
```



```
let rec height = function
  | E          -> 0
  | N (l, r) -> 1 + max (height l) (height r)
```



```
Stack overflow during evaluation (looping recursion?).
```

« il suffit d'utiliser une pile »

des solutions ad hoc

```
let rec bfs m next = function
  | [] when next = [] -> m
  | []                -> bfs (m+1) []           next
  | E         :: curr -> bfs m     next         curr
  | N (l, r) :: curr  -> bfs m     (l::r::next) curr
```

```
let rec bfs m next = function
  | [] when next = [] -> m
  | []                -> bfs (m+1) []           next
  | E         :: curr -> bfs m     next         curr
  | N (l, r) :: curr  -> bfs m     (l::r::next) curr
```

```
let height t = bfs 0 [] [t]
```

parcours en profondeur

```
let rec dfs m = function
  | []                    -> m
  | (n, E) :: s           -> dfs (max n m) s
  | (n, N (l,r)) :: s -> dfs m ((n+1,l)::(n+1,r)::s)
```

```
let rec dfs m = function
  | []                    -> m
  | (n, E) :: s           -> dfs (max n m) s
  | (n, N (l,r)) :: s -> dfs m ((n+1,l)::(n+1,r)::s)
```

```
let height t = dfs 0 [0, t]
```

des solutions génériques

idée : on calcule `k (height t)` plutôt que `height t`

idée : on calcule `k (height t)` plutôt que `height t`

```
let rec hcps t k = match t with
  | E          -> k 0
  | N (l, r) -> hcps l (fun hl ->
                hcps r (fun hr ->
                k (1 + max hl hr)))
```

idée : on calcule `k (height t)` plutôt que `height t`

```
let rec hcps t k = match t with
  | E          -> k 0
  | N (l, r) -> hcps l (fun hl ->
                hcps r (fun hr ->
                k (1 + max hl hr)))
```

```
let height t = hcps t (fun h -> h)
```

idée : on calcule `k (height t)` plutôt que `height t`

```
let rec hcps t k = match t with
  | E          -> k 0
  | N (l, r) -> hcps l (fun hl ->
                hcps r (fun hr ->
                k (1 + max hl hr)))
```

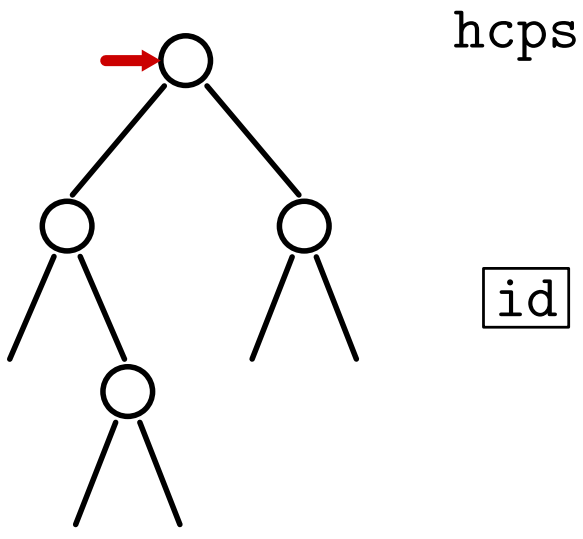
```
let height t = hcps t (fun h -> h)
```

```
...
... jmp *%rdi ...
... jmp hcps  ...
... jmp hcps  ...
... jmp *%rdi ...
...
... jmp hcps  ...
```

une « pile » sur le tas

```
let rec hcps t k = match t with
  | E          -> k 0
  | N (l, r) -> hcps l (fun hl ->
                hcps r (fun hr ->
                k (1 + max hl hr)))
```

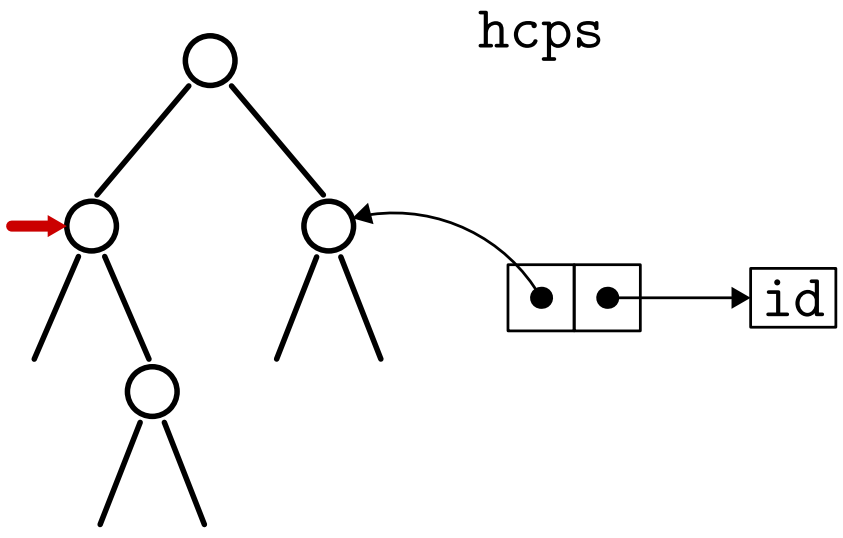
```
let height t = hcps t (fun h -> h)
```



une « pile » sur le tas

```
let rec hcps t k = match t with
  | E          -> k 0
  | N (l, r) -> hcps l (fun hl ->
                hcps r (fun hr ->
                k (1 + max hl hr)))
```

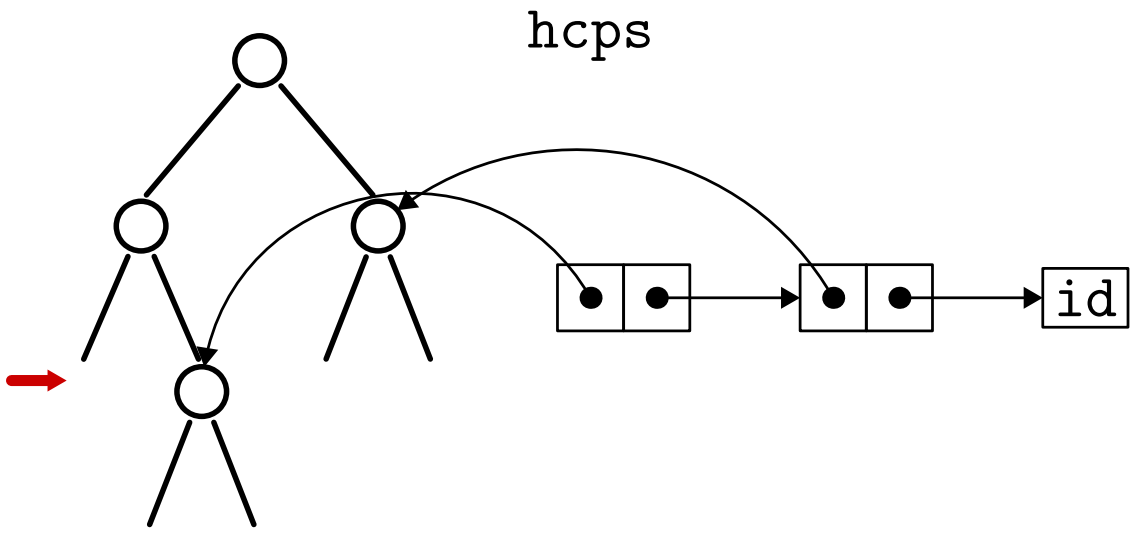
```
let height t = hcps t (fun h -> h)
```



une « pile » sur le tas

```
let rec hcps t k = match t with
  | E          -> k 0
  | N (l, r) -> hcps l (fun hl ->
                hcps r (fun hr ->
                k (1 + max hl hr)))
```

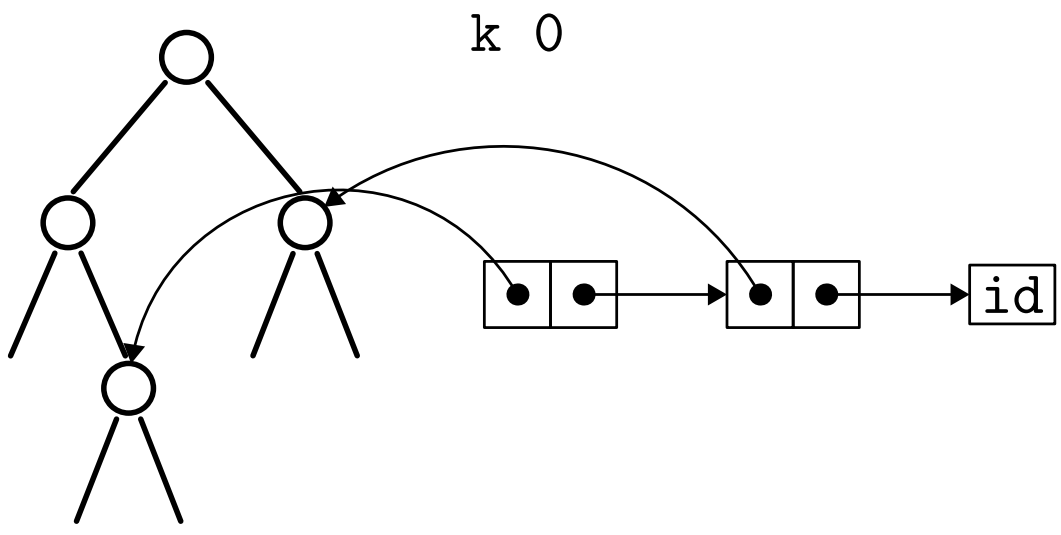
```
let height t = hcps t (fun h -> h)
```



une « pile » sur le tas

```
let rec hcps t k = match t with
  | E          -> k 0
  | N (l, r) -> hcps l (fun hl ->
                  hcps r (fun hr ->
                  k (1 + max hl hr)))
```

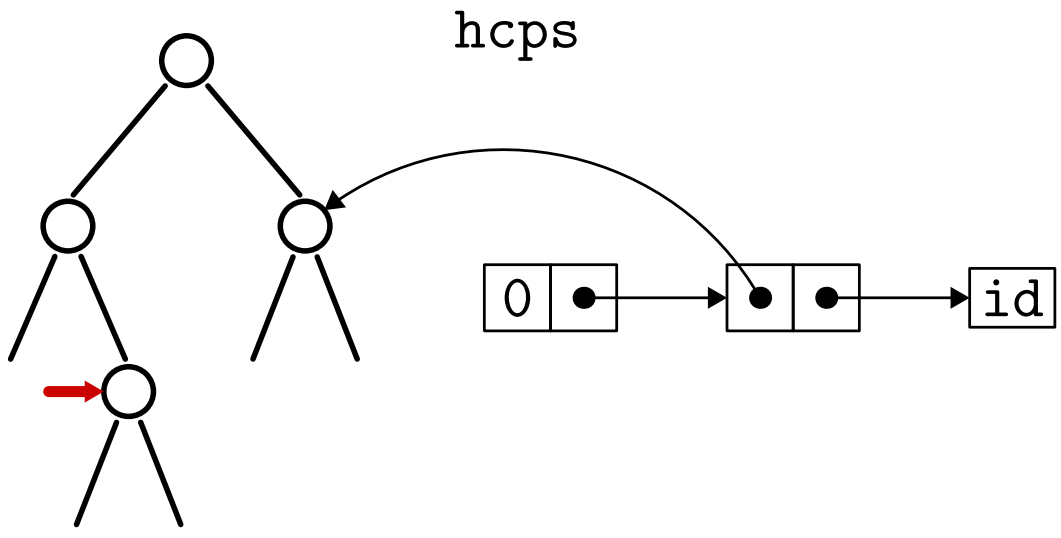
```
let height t = hcps t (fun h -> h)
```



une « pile » sur le tas

```
let rec hcps t k = match t with
  | E          -> k 0
  | N (l, r) -> hcps l (fun hl ->
                hcps r (fun hr ->
                k (1 + max hl hr)))
```

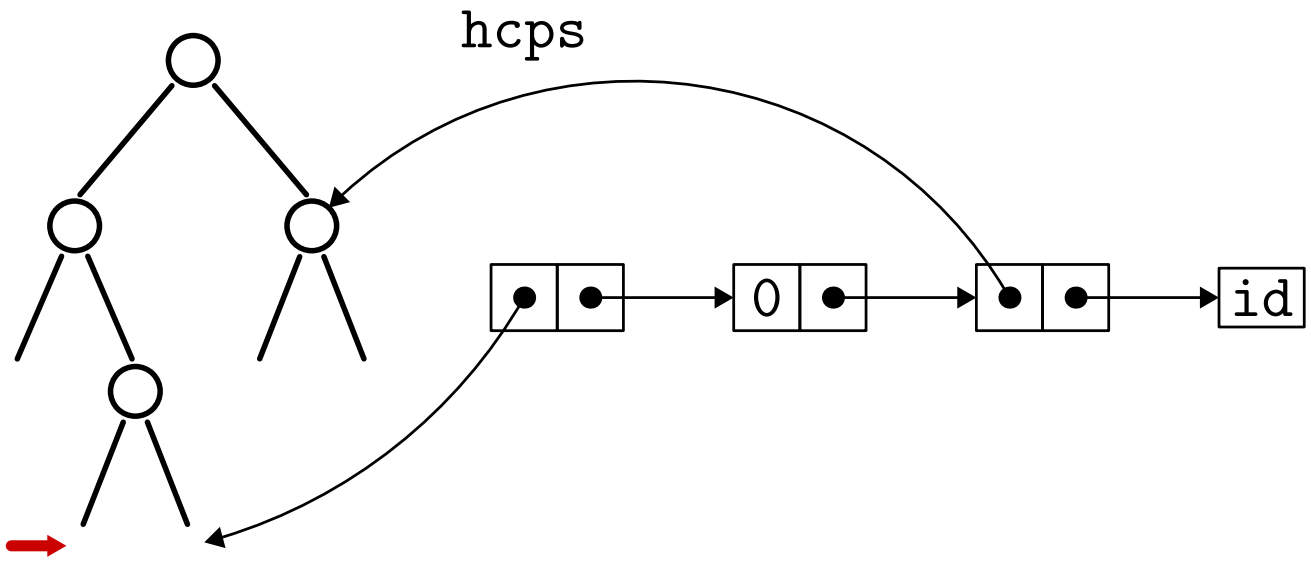
```
let height t = hcps t (fun h -> h)
```



une « pile » sur le tas

```
let rec hcps t k = match t with
  | E          -> k 0
  | N (l, r) -> hcps l (fun hl ->
                hcps r (fun hr ->
                k (1 + max hl hr)))
```

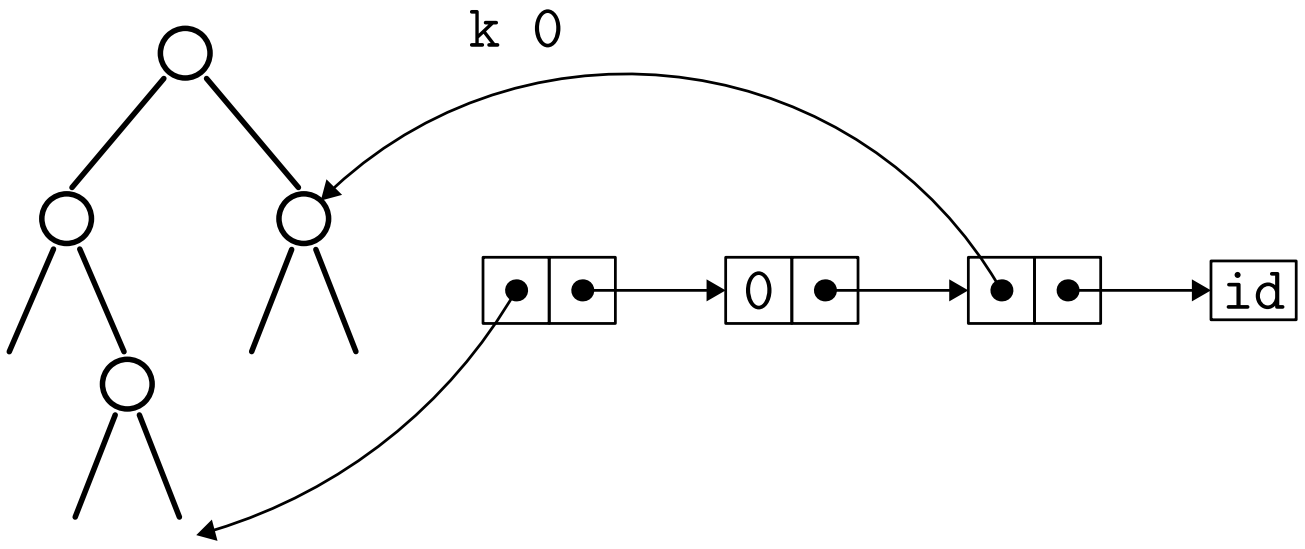
```
let height t = hcps t (fun h -> h)
```



une « pile » sur le tas

```
let rec hcps t k = match t with
  | E          -> k 0
  | N (l, r) -> hcps l (fun hl ->
                hcps r (fun hr ->
                k (1 + max hl hr)))
```

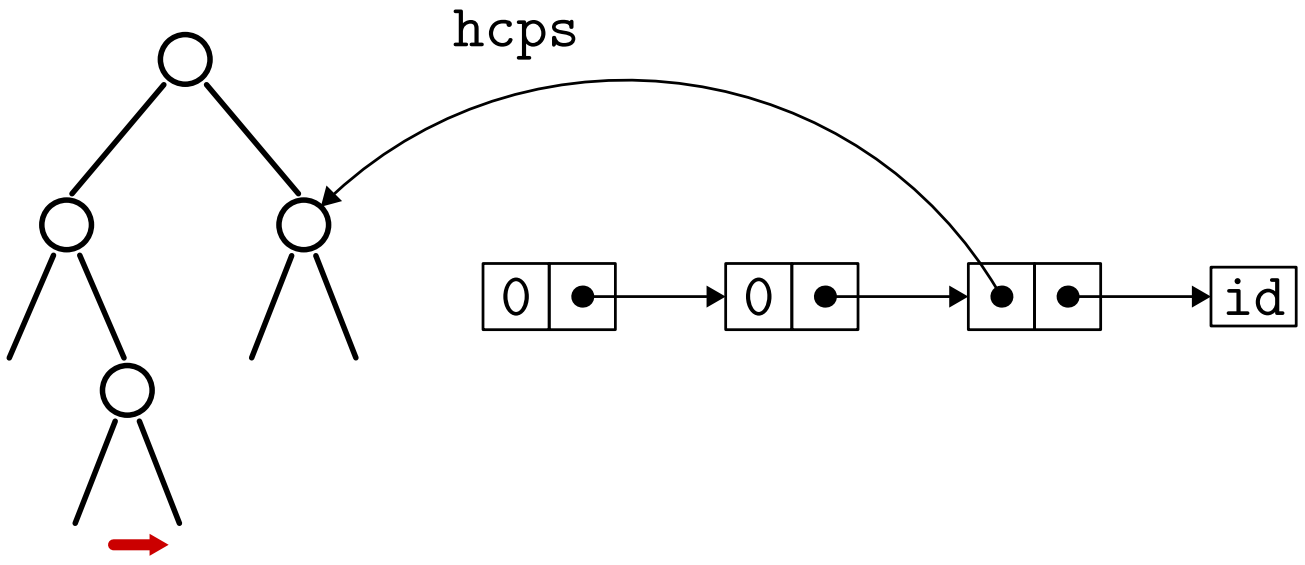
```
let height t = hcps t (fun h -> h)
```



une « pile » sur le tas

```
let rec hcps t k = match t with
  | E          -> k 0
  | N (l, r) -> hcps l (fun hl ->
                hcps r (fun hr ->
                k (1 + max hl hr)))
```

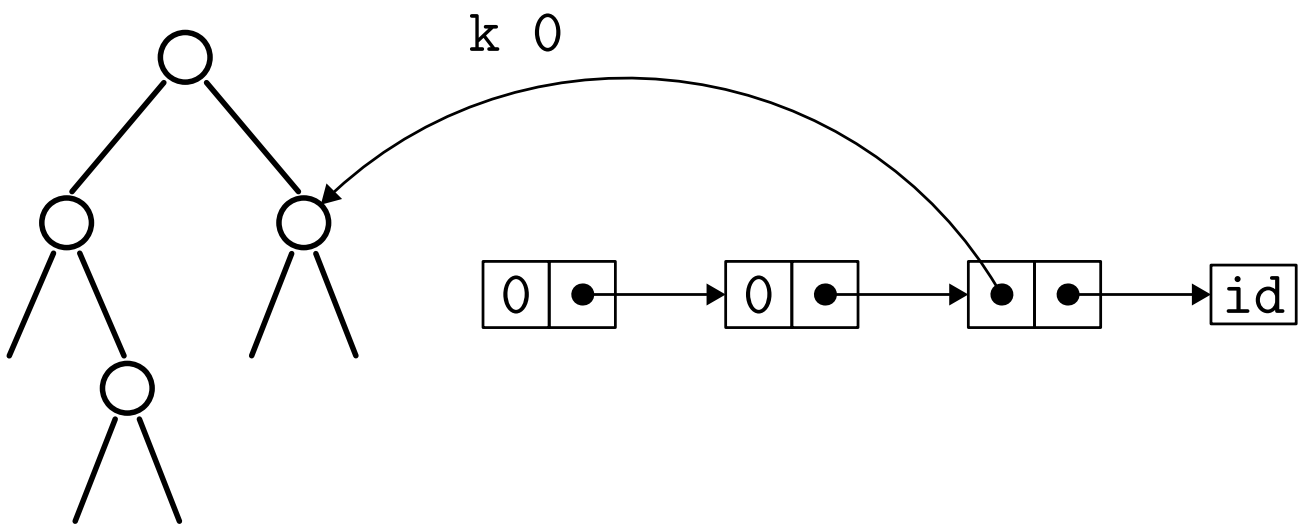
```
let height t = hcps t (fun h -> h)
```



une « pile » sur le tas

```
let rec hcps t k = match t with
  | E          -> k 0
  | N (l, r) -> hcps l (fun hl ->
                hcps r (fun hr ->
                k (1 + max hl hr)))
```

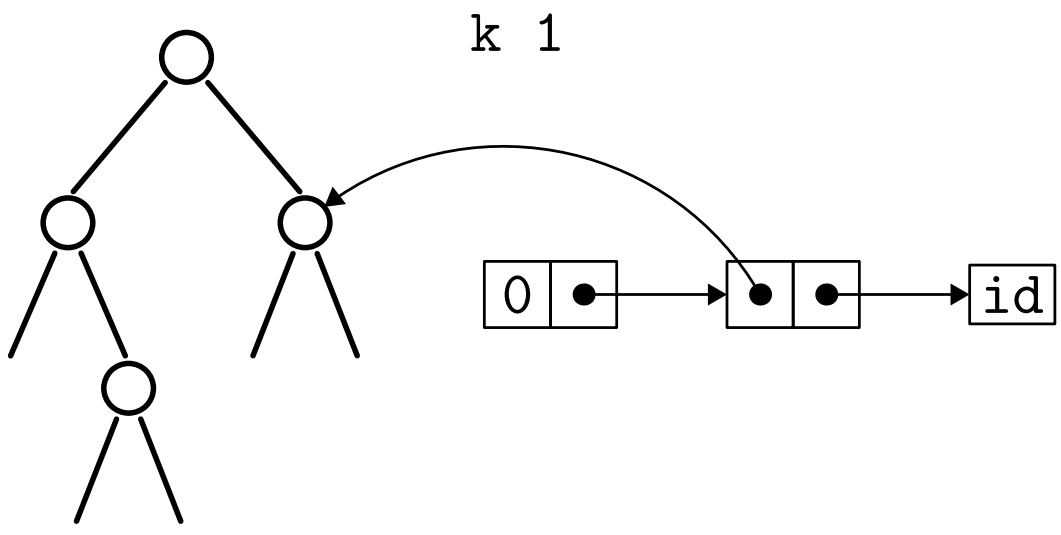
```
let height t = hcps t (fun h -> h)
```



une « pile » sur le tas

```
let rec hcps t k = match t with
  | E          -> k 0
  | N (l, r) -> hcps l (fun hl ->
                hcps r (fun hr ->
                k (1 + max hl hr)))
```

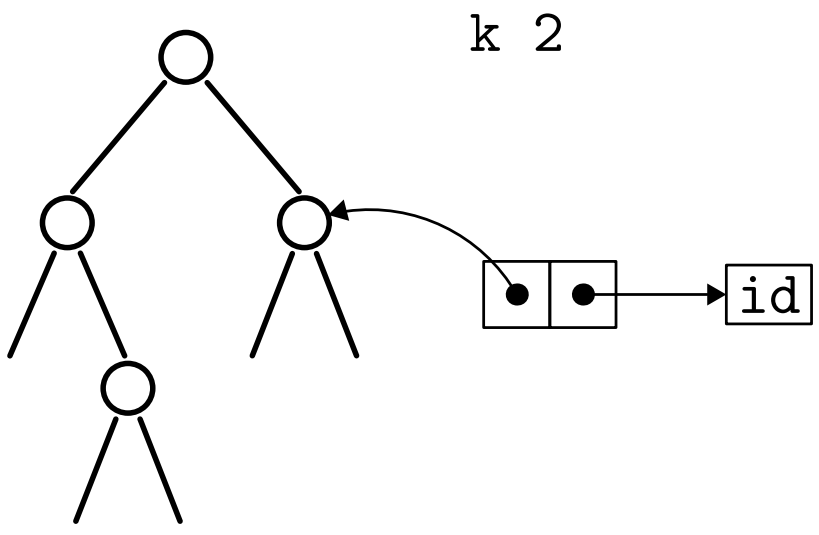
```
let height t = hcps t (fun h -> h)
```



une « pile » sur le tas

```
let rec hcps t k = match t with
  | E          -> k 0
  | N (l, r) -> hcps l (fun hl ->
                hcps r (fun hr ->
                k (1 + max hl hr)))
```

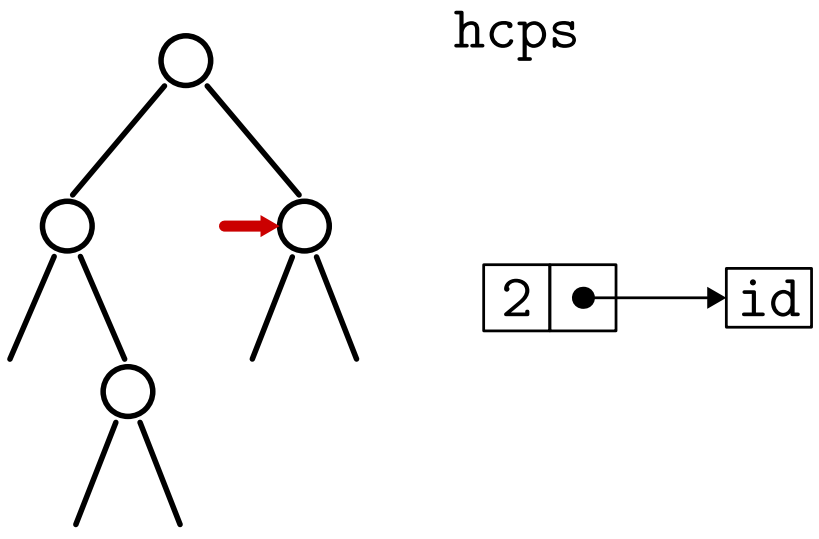
```
let height t = hcps t (fun h -> h)
```



une « pile » sur le tas

```
let rec hcps t k = match t with
  | E          -> k 0
  | N (l, r) -> hcps l (fun hl ->
                hcps r (fun hr ->
                k (1 + max hl hr)))
```

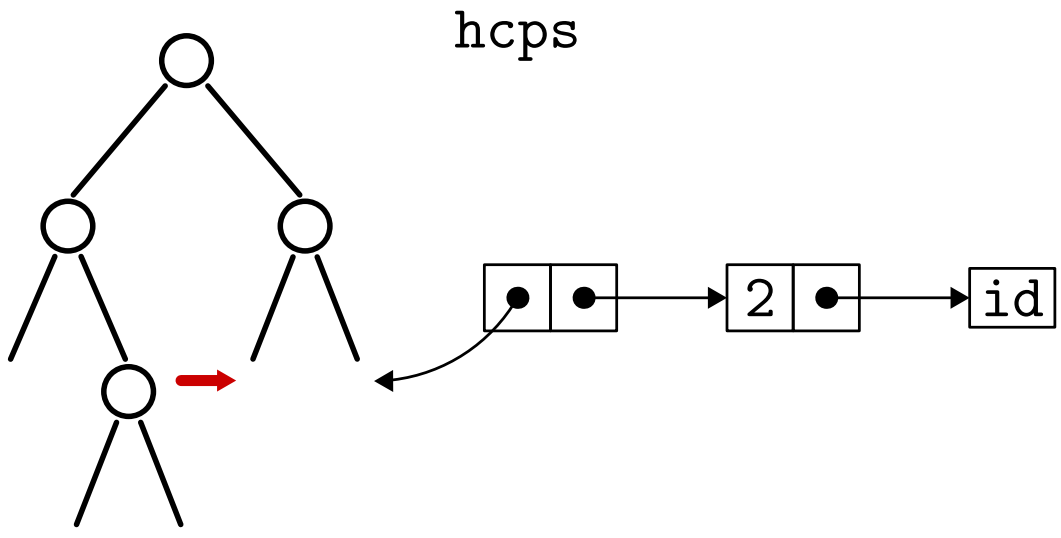
```
let height t = hcps t (fun h -> h)
```



une « pile » sur le tas

```
let rec hcps t k = match t with
  | E          -> k 0
  | N (l, r) -> hcps l (fun hl ->
                hcps r (fun hr ->
                k (1 + max hl hr)))
```

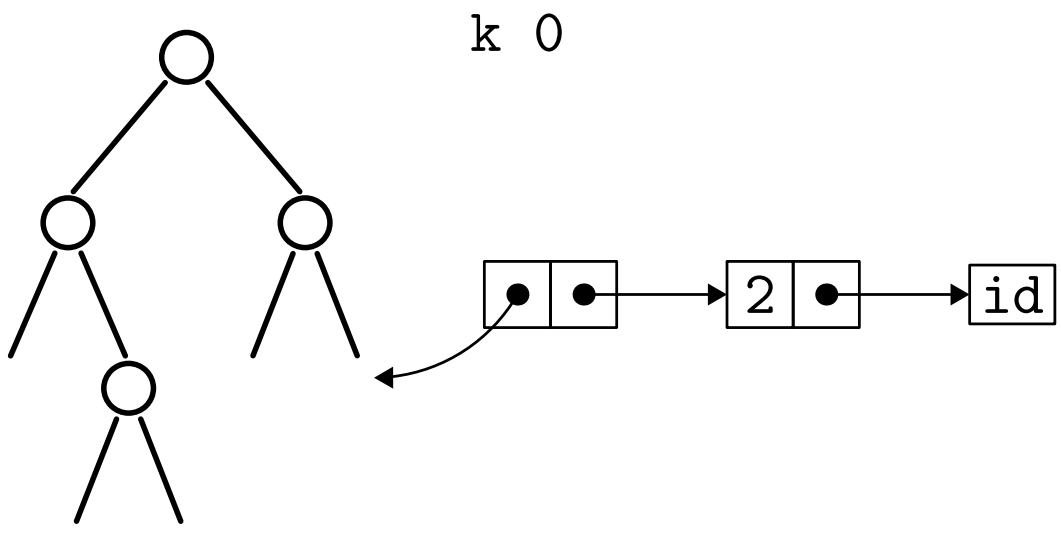
```
let height t = hcps t (fun h -> h)
```



une « pile » sur le tas

```
let rec hcps t k = match t with
  | E          -> k 0
  | N (l, r) -> hcps l (fun hl ->
                hcps r (fun hr ->
                k (1 + max hl hr)))
```

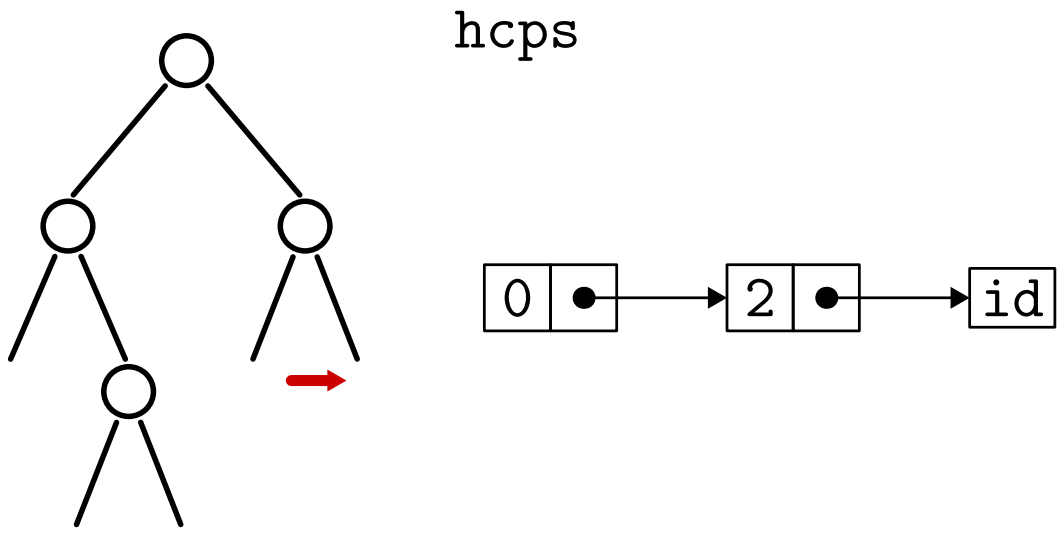
```
let height t = hcps t (fun h -> h)
```



une « pile » sur le tas

```
let rec hcps t k = match t with
  | E          -> k 0
  | N (l, r) -> hcps l (fun hl ->
                hcps r (fun hr ->
                k (1 + max hl hr)))
```

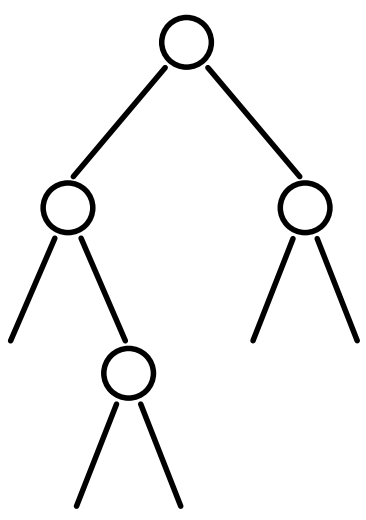
```
let height t = hcps t (fun h -> h)
```



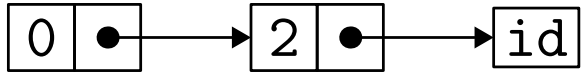
une « pile » sur le tas

```
let rec hcps t k = match t with
  | E          -> k 0
  | N (l, r) -> hcps l (fun hl ->
                hcps r (fun hr ->
                k (1 + max hl hr)))
```

```
let height t = hcps t (fun h -> h)
```



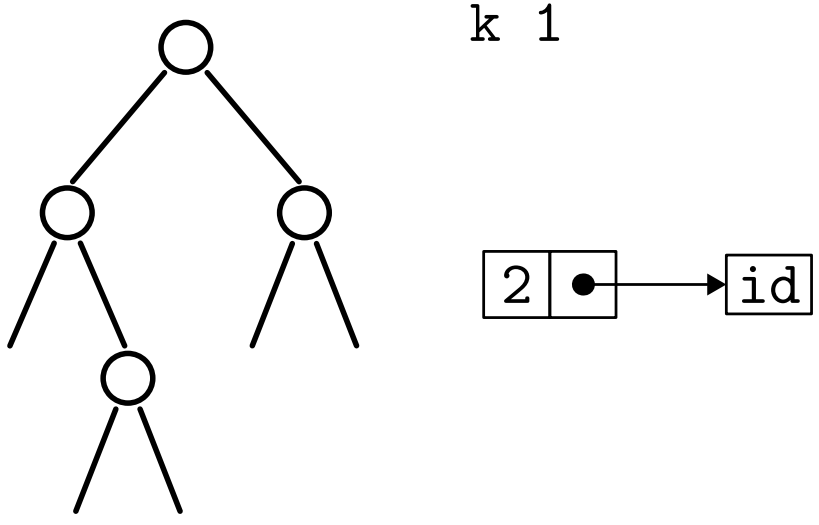
k 0



une « pile » sur le tas

```
let rec hcps t k = match t with
  | E          -> k 0
  | N (l, r) -> hcps l (fun hl ->
                hcps r (fun hr ->
                k (1 + max hl hr)))
```

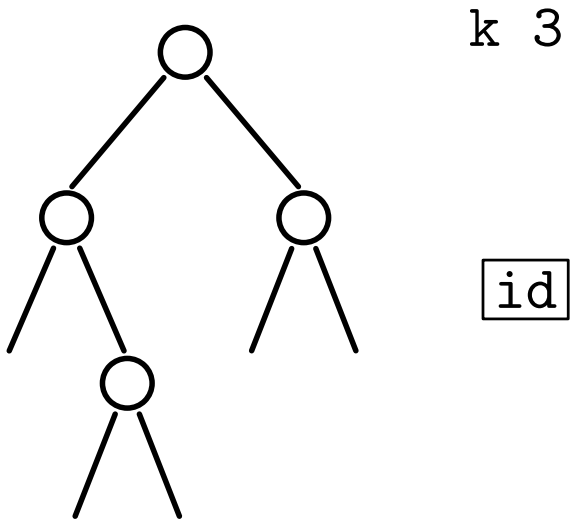
```
let height t = hcps t (fun h -> h)
```



une « pile » sur le tas

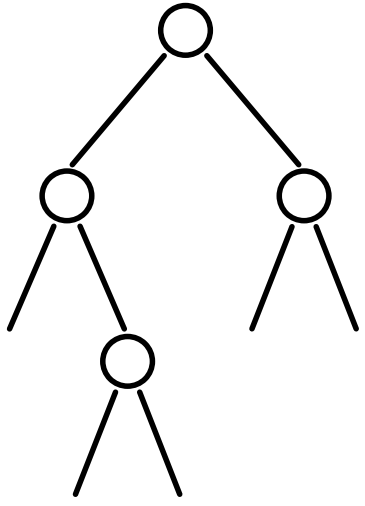
```
let rec hcps t k = match t with
  | E          -> k 0
  | N (l, r) -> hcps l (fun hl ->
                hcps r (fun hr ->
                k (1 + max hl hr)))
```

```
let height t = hcps t (fun h -> h)
```



```
let rec hcps t k = match t with
  | E          -> k 0
  | N (l, r) -> hcps l (fun hl ->
                hcps r (fun hr ->
                k (1 + max hl hr)))
```

```
let height t = hcps t (fun h -> h)
```



3

et dans un autre langage ?

après tout, on a des fonctions anonymes en Java, C++, etc.

après tout, on a des fonctions anonymes en Java, C++, etc.

```
int hcps(tree t, function<int(int)> const&k) {
  if (t == NULL) return k(0);
  return hcps(t->left,  [t, &k](int hl) {
  return hcps(t->right, [hl,&k](int hr) {
  return k(1 + (hl > hr ? hl : hr)); }); });
}
```

malheureusement, les appels terminaux ne sont pas optimisés

défonctionnalisation (Reynolds, 1972)

```
type cont =
  | Kid
  | Kleft  of tree * cont
  | Kright of int  * cont
```

défonctionnalisation (Reynolds, 1972)

```
type cont =
  | Kid
  | Kleft  of tree * cont
  | Kright of int  * cont
```

```
let rec hcps t k = match t with
  | E         -> apply k 0
  | N (l, r) -> hcps l (Kleft (r, k))
```

défonctionnalisation (Reynolds, 1972)

```
type cont =
  | Kid
  | Kleft  of tree * cont
  | Kright of int  * cont
```

```
let rec hcps t k = match t with
  | E         -> apply k 0
  | N (l, r) -> hcps l (Kleft (r, k))
```

```
and apply k v = match k with
  | Kid              -> v
  | Kleft  (r,  k) -> hcps r (Kright (v, k))
  | Kright (hl, k) -> apply k (1 + max hl v)
```

```
let hdefun t = hcps t Kid
```

s'adapte plus facilement à d'autres langages

s'adapte plus facilement à d'autres langages

```
struct Kont {
  enum   { Kid, Kleft, Kright } kind;
  union  { struct Node *r; int hl; };
  struct Kont *kont;
};

int hcps(struct Node *t, struct Kont *k) { ... }
int apply(struct Kont *k, int v) { ... }
```

gcc/clang optimisent ici tous les appels terminaux

encore une autre solution

parcourir l'arbre avec un zipper (Huet, 1997)

version	
`height`	1,28

version	
`bfs`	0,94
opt. `bfs`	0,58
`dfs`	2,32
opt. `dfs`	0,52

version	
`cps`	2,55
`defun`	1,80
`zipper`	3,75

(ici avec OCaml uniquement)

si l'arbre est mutable, on peut faire un parcours infixe en place :

> *Joseph M. Morris.*
> *Traversing binary trees simply and cheaply.*
> *Information Processing Letters, 9(5) :197–200, 1979.*

(modifie et restaure la structure de l'arbre)

s'adapte facilement pour calculer la hauteur

éviter le débordement de pile est rapidement compliqué

avec OCaml (ou Haskell), on est plutôt bien loti !

```
   type tree = E |
  N of tree * tree
let rec aux t k =match
t with | E -> k 0 |N(l,
 r) -> aux l(fun hl ->
aux r (fun hr ->k (1 +
   max hl hr)))let
        height
        (t)=
         aux
         (t)(
         fun
         (h)
        ->h )
```