

Voorbeeldvragen Algoritmen & Datastructuren I

Bomen

Titularis: Prof. Dr. Wolfgang De Meuter

Assistenten: Nathalie Oostvogels, Simon Van de Water

16 december 2015

Vraag 1

In de cursus definieerden we het `binary-tree` ADT, dat toelaat om bomen te beschrijven waarvan elke knoop één waarde, en maximum twee kinderen heeft. Analooq definiëren we hieronder het `ternary-tree` ADT, dat toelaat om bomen te beschrijven waarvan elke knoop maximum twee waarden, en maximum drie kinderen heeft:

ADT ternary-tree

`null-tree`

`ternary-tree`

`new`

`( any any ternary-tree ternary-tree ternary-tree → ternary-tree )`

`null-tree?`

`( ternary-tree → boolean )`

`left!`

`( ternary-tree ternary-tree → ternary-tree )`

`left`

`( ternary-tree → ternary-tree )`

`middle!`

`( ternary-tree ternary-tree → ternary-tree )`

`middle`

`( ternary-tree → ternary-tree )`

`right!`

`( ternary-tree ternary-tree → ternary-tree )`

`right`

`( ternary-tree → ternary-tree )`

`left-value!`

`( ternary-tree any → ternary-tree )`

`left-value`

`( ternary-tree → any )`

`right-value!`

`( ternary-tree any → ternary-tree )`

`right-value`

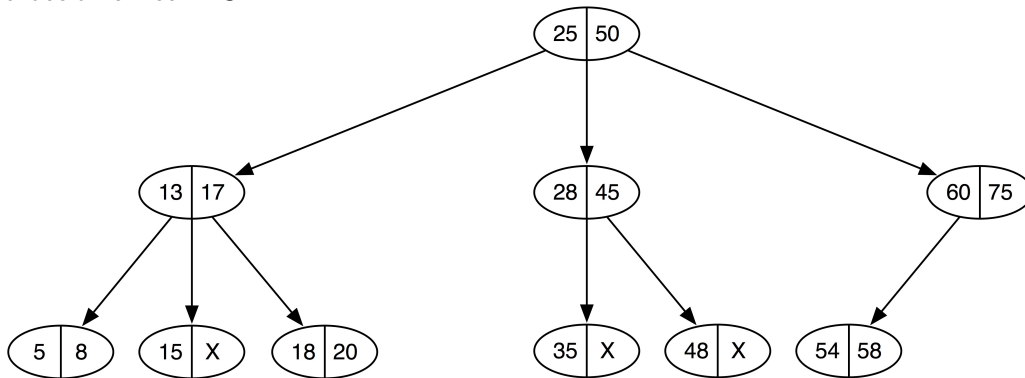
`( ternary-tree → any )`

Een *ternary search tree* of TST is een ternary tree die aan de volgende *TST voorwaarde* voldoet. Voor elke knoop k geldt:

- Als k een linkerkind heeft, dan zijn de waarden van alle knopen van dat linkerkind kleiner dan de linkerwaarde van k .
- Als k een middelste kind heeft, dan zijn de waarden van alle knopen van dat middelste kind groter dan de linkerwaarde van k , en kleiner dan de rechterwaarde van k .

- Als k een rechterkind heeft, dan zijn de waarden van alle knopen van dat rechterkind groter dan de rechterwaarde van k .

Ziehier een voorbeeld van een TST:



Door middel van het ternary-tree ADT zouden we deze boom als volgt kunnen aanmaken:

```
(define my-tst
  (new 25 50
    (new 13 17
      (new 5 8 null-tree null-tree null-tree)
      (new 15 '() null-tree null-tree null-tree)
      (new 18 20 null-tree null-tree null-tree))
    (new 28 45
      null-tree
      (new 35 '() null-tree null-tree null-tree)
      (new 48 '() null-tree null-tree null-tree))
    (new 60 75
      (new 54 58 null-tree null-tree null-tree)
      null-tree
      null-tree)))
```

- Implementeer de functie (`find tst key`), die de gegeven sleutel **key** opzoekt in de gegeven ternary search tree **tst**.
- Implementeer de functie (`insert! tst val`), die de gegeven waarde **val** aan de gegeven ternary search tree **tst** toevoegt. Als hulp vind je in bijlage de gelijknamige functie voor binary search trees.
- Beschrijf in woorden (maximum 1/2 bladzijde) hoe je de functie (`delete! tst val`) zou implementeren. Deze functie verwijdert de gegeven waarde **val** uit de gegeven ternary search tree **tst**.
- Geef de performantie $O()$ van de functie (`find tst key`). Beschrijf duidelijk je redenering.

Vraag 2

In bijlage vind je de implementatie van het binary-tree ADT.

- a) Schrijf een functie (`bst? my-tree`) die bepaalt of de gegeven binaire boom `my-tree` een **binaire zoek-boom** is.
- b) Schrijf een functie (`avl? my-tree`) die bepaalt of de gegeven binaire boom een **AVL-boom** is.
- c) Schrijf een functie (`heapish? my-tree`) die bepaalt of de gegeven binaire boom **“heap-achtig”** is, m.a.w., of de boom voldoet aan de heap-voorwaarde.
- d) Waarom werd er in vraag c) voor gekozen om deze functie `heapish?` te noemen, en niet `heap`?

```
#!/r6rs
```

```
;*****  
;*****  
;*-*                                     *-*  
;*-*                               Binary Trees          *-*  
;*-*                                     *-*  
;*-*                               Wolfgang De Meuter    *-*  
;*-*                        2009 Software Languages Lab  *-*  
;*-*                        Vrije Universiteit Brussel   *-*  
;*-*                                     *-*  
;*****  
;*****
```

```
(library  
  (binary-tree)  
  (export null-tree new null-tree? value value! left left! right right!)  
  (import (rnrs base)  
    (srfi :9)  
    (rnrs mutable-pairs))  
  
  (define-record-type tree  
    (new v l r)  
    tree?  
    (v value value!)  
    (l left left!)  
    (r right right!))  
  
  (define null-tree ())  
  
  (define (null-tree? node)  
    (eq? node null-tree)))
```

Vraag 3

- a) Beschouw onderstaande procedure uit de cursus. Leg uit (m.b.v. de terminologie van de cursus) *wat* deze procedure precies doet. Gebruik maximaal 1 bladzijde voor je antwoord.

```
(define (mystery tree proc)
  (define stack (stack:new))
  (define (do-it-2)
    (let ((node (stack:pop! stack)))
      (proc (value node))
      (if (not (null-tree? (right node)))
          (begin (stack:push! stack (right node))
                  (do-it-1))
          (if (not (stack:empty? stack))
              (do-it-2))))))
  (define (do-it-1)
    (let ((node (stack:top stack)))
      (if (not (null-tree? (left node)))
          (begin (stack:push! stack (left node))
                  (do-it-1))
          (do-it-2))))
  (stack:push! stack tree)
  (do-it-1))
```

- b) Bewijs dat het zoeken van een datawaarde in een k -aire zoekboom die n elementen bevat in totaal $O(\log_2(n))$ computationele stappen vereist.

Vraag 4

Deelvraag 4.1

- Voeg de elementen met de volgende sleutels – in de volgorde waarin ze hieronder opgelijst staan – toe aan een AVL-boom die initieel leeg is: 30, 40, 24, 58, 48, 26, 28, 29, 45. Teken na elke toevoeging de resulterende boom.
- Verwijder vervolgens het element met sleutel 26 en teken de resulterende boom.
- Verwijder vervolgens het element met sleutel 48 en teken de resulterende boom.

Deelvraag 4.2

- De procedure om een element uit een AVL-boom te verwijderen kan op twee manieren geschreven worden die volledig symmetrisch zijn. Schrijf de symmetrische tegenhanger van de delete!-procedure in bijlage.

```

(define (delete! avl val)
  (define ==? (equality avl))
  (define <=? (lesser avl))

  (define (find-leftmost deleted parent child! child)
    (if (null-tree? (AVL-node-left child))
        (begin
          (AVL-node-value! deleted (AVL-node-value child))
          (child! parent (AVL-node-right child))
          #t)
        (if (find-leftmost deleted child AVL-node-left! (AVL-node-left child))
            (check-after-delete-left parent child! child)
            #f)))

  (define (delete-node parent child! child)
    (cond
      ((null-tree? (AVL-node-left child))
       (child! parent (AVL-node-right child))
       #t)
      ((null-tree? (AVL-node-right child))
       (child! parent (AVL-node-left child))
       #t)
      (else
       (if (find-leftmost child child AVL-node-right! (AVL-node-right child))
           (check-after-delete-right parent child! child)
           #f)))))

  (let find-node
    ((parent null-tree)
     (child! (lambda (ignore child) (root! avl child)))
     (child (root avl)))
    (cond
      ((null-tree? child)
       #f)
      ((==? (AVL-node-value child) val)
       (delete-node parent child! child))
      ((<=? (AVL-node-value child) val)
       (if (find-node child AVL-node-right! (AVL-node-right child))
           (check-after-delete-right parent child! child)
           #f))
      ((<=? val (AVL-node-value child))
       (if (find-node child AVL-node-left! (AVL-node-left child))
           (check-after-delete-left parent child! child)
           #f)))))

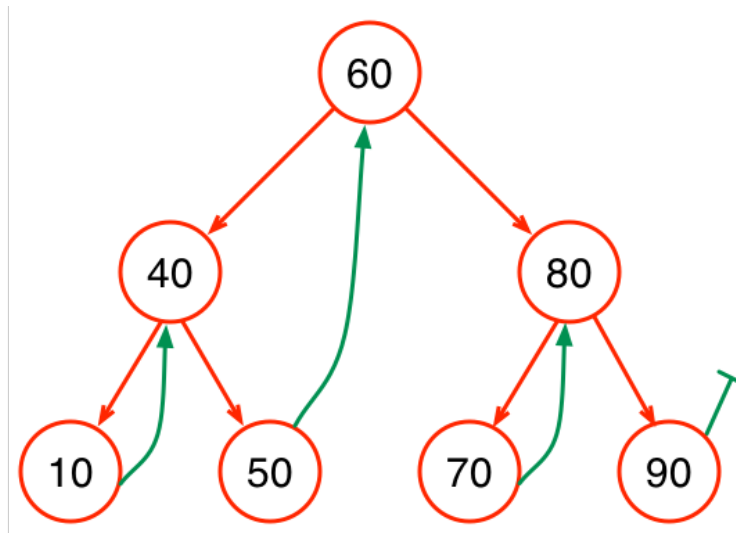
  avl)

```

Vraag 5

Threaded bomen zijn binaire bomen waarbij elke node dat geen rechterkind heeft, deze ongebruikte pointer opvult door te wijzen naar de eerstvolgende node nodig in een in-order traversal. Dit zorgt ervoor dat we op een iteratieve manier een in-order traversal van de boom kunnen doen zonder gebruik te maken van een stack.

Figuur 1 geeft een voorbeeld van zo een threaded binaire boom die bovendien ook een zoekboom is. De rode nodes en pijlen vormen de standaard binaire zoekboom. De groene pijlen zijn telkens de lege rechterkinderen die opgevuld zijn met een pointer naar de eerstvolgende node voor de in-order traversal.



Figuur 1: Een threaded binaire boom

Om threaded zoekbomen te implementeren, hebben we naast het tree ADT (voor de standaard binaire boom), ook een nieuw type nodig. Met dit type kunnen we dan de groene pijlen van de threaded zoekbomen voorstellen. In de volgende code wordt dit nieuw type aangemaakt:

```
(define-record-type groenepijl
  (new-groenepijl rode)
  groenepijl?
  (rode opvolger opvolger!))
```

- a) Herschrijf de volgende procedures voor threaded zoekbomen, zodat er ook rekening wordt gehouden met de groene pijlen. Als geheugensteuntje staan op de volgende pagina het tree ADT, alsook de `find` en `insert!` procedures voor de standaard binaire zoekboom, zoals gezien in de cursus.
 - (a) Implementeer `find`
 - (b) Implementeer `insert!`
- b) Schrijf de in-order traversal voor deze threaded zoekbomen op een iteratieve manier, zonder een stack te gebruiken. Zoals steeds, verwacht deze procedure twee argumenten: de boom en de procedure die uitgevoerd moet worden voor elke node.

```

(define-record-type tree
  (new v l r)
  tree?
  (v value value!)
  (l left left!)
  (r right right!))

(define null-tree ())
(define (null-tree? node)
  (eq? node null-tree))

```

```

(define-record-type bst
  (make r e l)
  bst?
  (r root root!)
  (e equality)
  (l lesser))

(define (find bst key)
  (define <<? (lesser bst))
  (define ==? (equality bst))
  (let find-key
    ((node (root bst)))
    (if (tree:null-tree? node)
        #f
        (let
          ((node-value (tree:value node)))
          (cond
            ((==? node-value key)
             node-value)
            (<<? node-value key)
            (find-key (tree:right node)))
            (<<? key node-value)
            (find-key (tree:left node)))))))

(define (insert! bst val)
  (define <<? (lesser bst))
  (let insert-iter
    ((parent tree:null-tree)
     (child! (lambda (ignore child) (root! bst child)))
     (child (root bst)))
    (cond
      ((tree:null-tree? child)
       (child! parent
        (tree:new val
         tree:null-tree
         tree:null-tree)))
      (<<? (tree:value child) val)
      (insert-iter child tree:right!
        (tree:right child)))
      (<<? val (tree:value child)
      (insert-iter child tree:left!
        (tree:left child)))
      (else
       (tree:value! child val)))))

```

