Hoofdstuk 17

Indexering m.b.v. B^+-Trees

Index Files

Hoe vinden we het blok waar een gewenst tupel zit?

SELECT FROM table WHERE att=value

SELECT FROM table WHERE att>value

tabellen

Data files

Index files

wegwijzers

Een <u>index file</u> is een hulpfile die voldoende gegevens bevat om een tupel op basis van een sleutelwaarde (voor één van de velden) snel terug te vinden in een datafile.

Optimaal zoeken in een dynamische structuur ⇒ Bomen

Zoekbomen op Disk

"Gewone" AVL-bomen hebben een vreselijk slecht caching-gedrag: naburige nodes zitten potentieel in héél verschillende blokken.

slechte space locality

Mogelijke oplossing: naburige nodes in hetzelfde blok steken. Maar dan zijn de extra pointers overbodig en is een N-aire boom beter!

⇒ Basisidee: gebruik N-aire bomen met 1 node per blok

N hangt af van block-size (vast) en key-size (kan verschillen per boom)

Zoeken in een N-aire boom in centraal geheugen

Zoeksnelheid = #niveaus × #zoeken in een niveau

Voor n datawaarden en ariteit N, geeft dit:

hoogte gebalanceerde N-aire boom

binary search over N-1 keys

$$O(\log_N(n)\log_2(N))$$

$$\log_a(b) = \frac{\log_2(b)}{\log_2(a)}$$

$$= O\left(\frac{\log_2(n)}{\log_2(N)}\log_2(N)\right)$$

$$= O(\log_2(n))$$

Conclusie: N-aire bomen hebben geen nut. Tenzij...

N-aire bomen op disk

Zoeksnelheid = #niveaus × #zoeken in een niveau

Voor n datawaarden en ariteit N, geeft dit:

lees diskblok

binary search in N-1 keys

$$O(\log_N(n)(T + t\log_2(N)))$$

$\log_N(n)\log_2(N) = \log_2(n)$

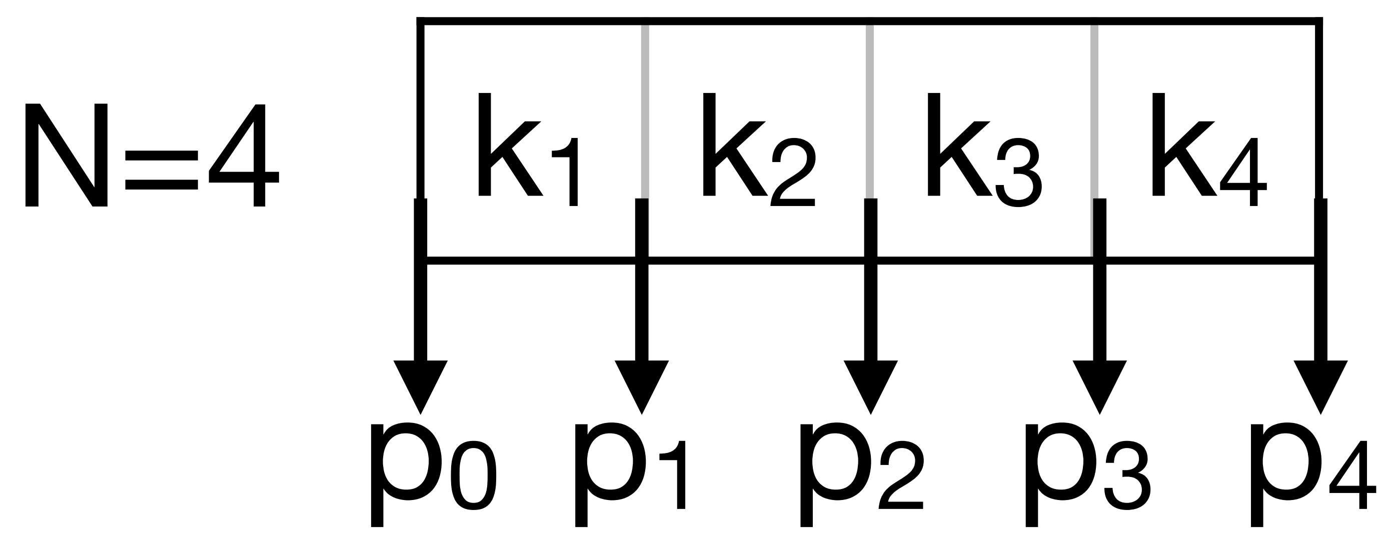
$$= O(T\log_N(n) + t\log_2(n))$$

$$= O(\log_N(n))$$ vermits $t \ll T$

$$T = T_{seek} + T_{latency} + T_{transfer}$$

Conclusie: N-aire bomen zijn nuttig op disk als een knoop overeenkomt met een blok.

B-Trees



Elke knoop bevat maximum N+1 pointers gescheiden door N keys

"Gebalanceerde N-aire bomen op disk"

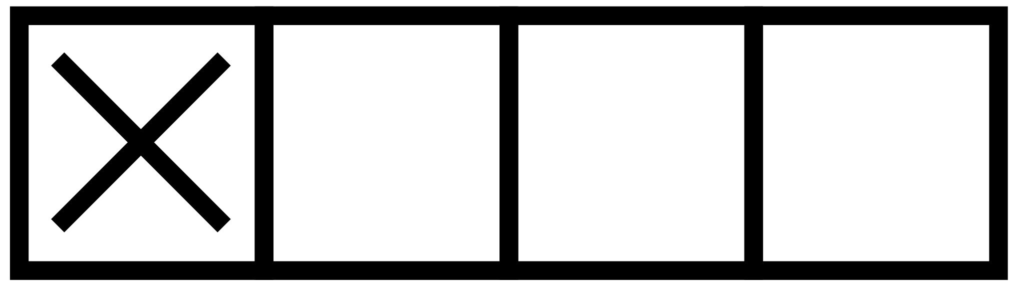
Variante: B^+-trees

Interne knopen worden maximaal gebruikt voor navigatie en bevatten dus enkel keys. Enkel de bladeren bevatten 'volledige' informatie (i.e., key + rcid)

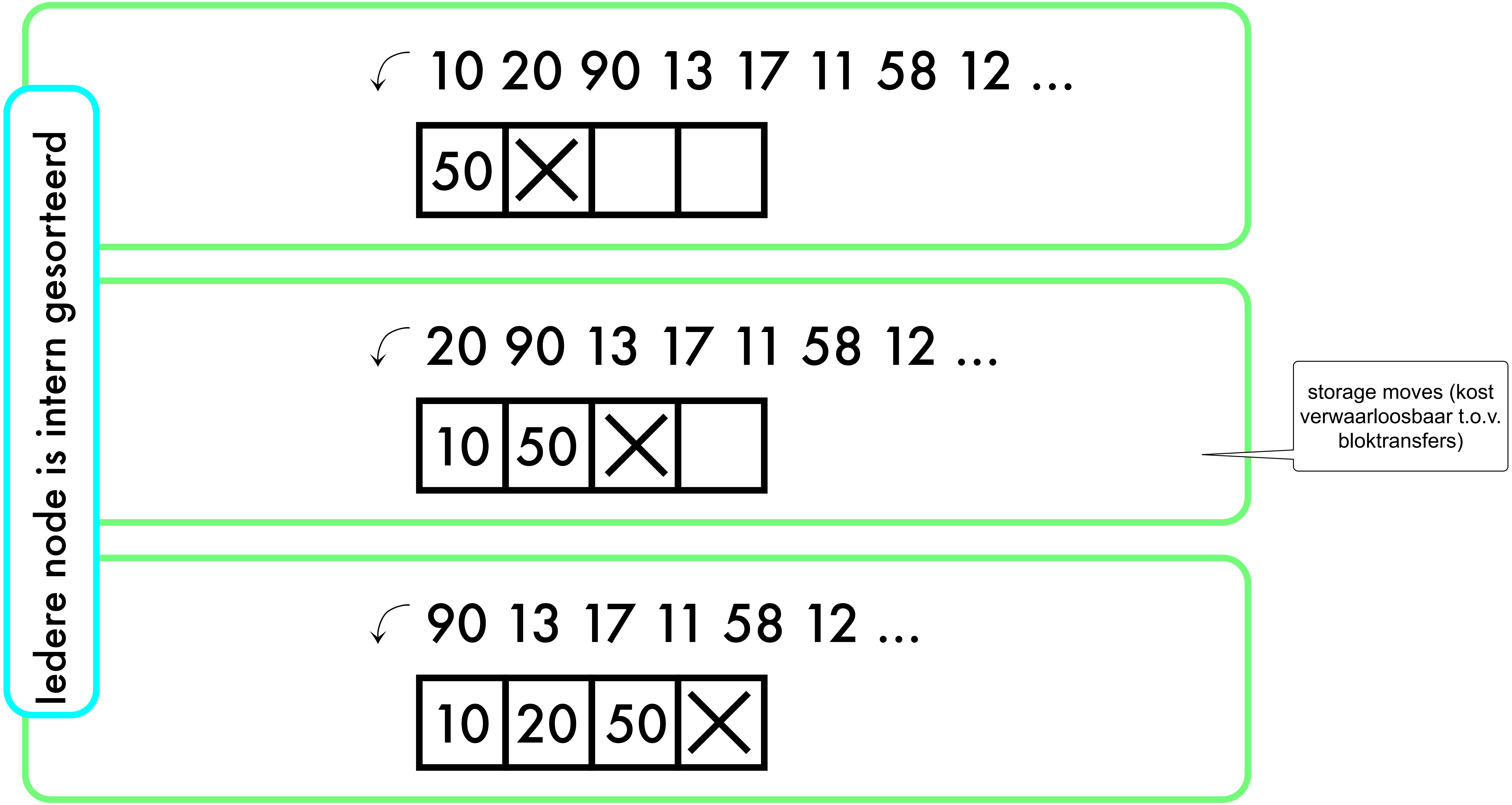
B-Trees: Voorbeeld (N=4)

We inserten achtereenvolgens:
50 10 20 90 13 17 11 58 12 ...

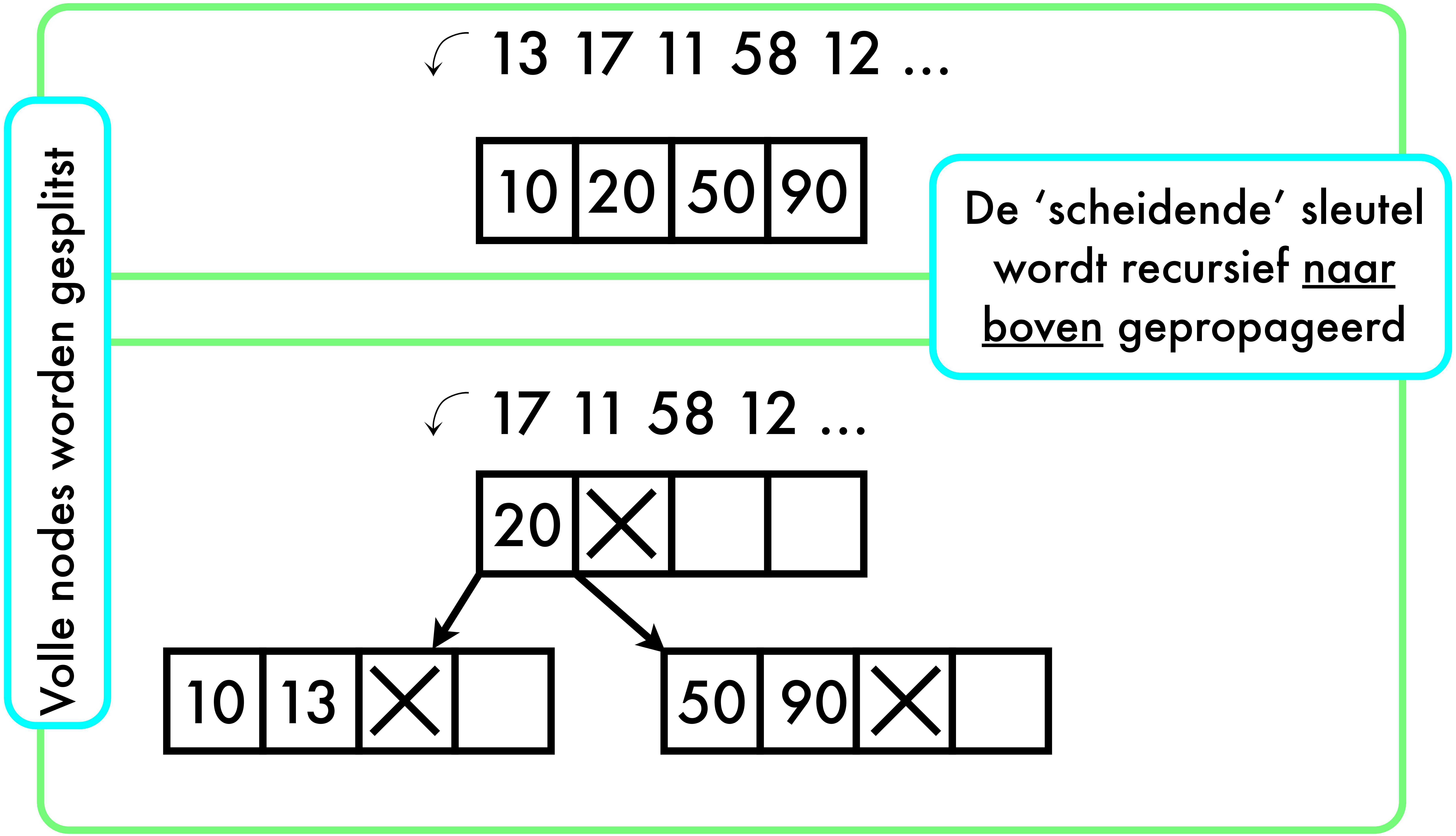
50 10 20 90 13 17 11 58 12 ...



Voorbeeld


Iedere node is intern gesorteerd
10 20 90 13 17 11 58 12 ...
50
20 90 13 17 11 58 12 ...
10 50
storage moves (kost verwaarloosbaar t.o.v. bloktransfers)
90 13 17 11 58 12 ...
10 20 50

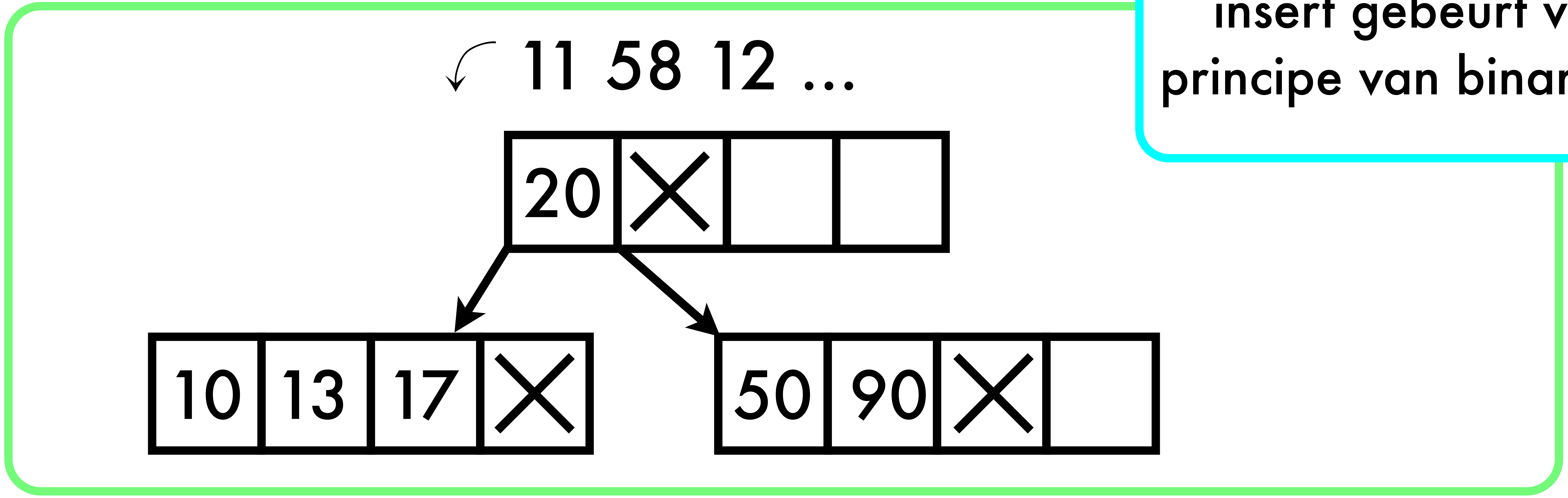
Voorbeeld

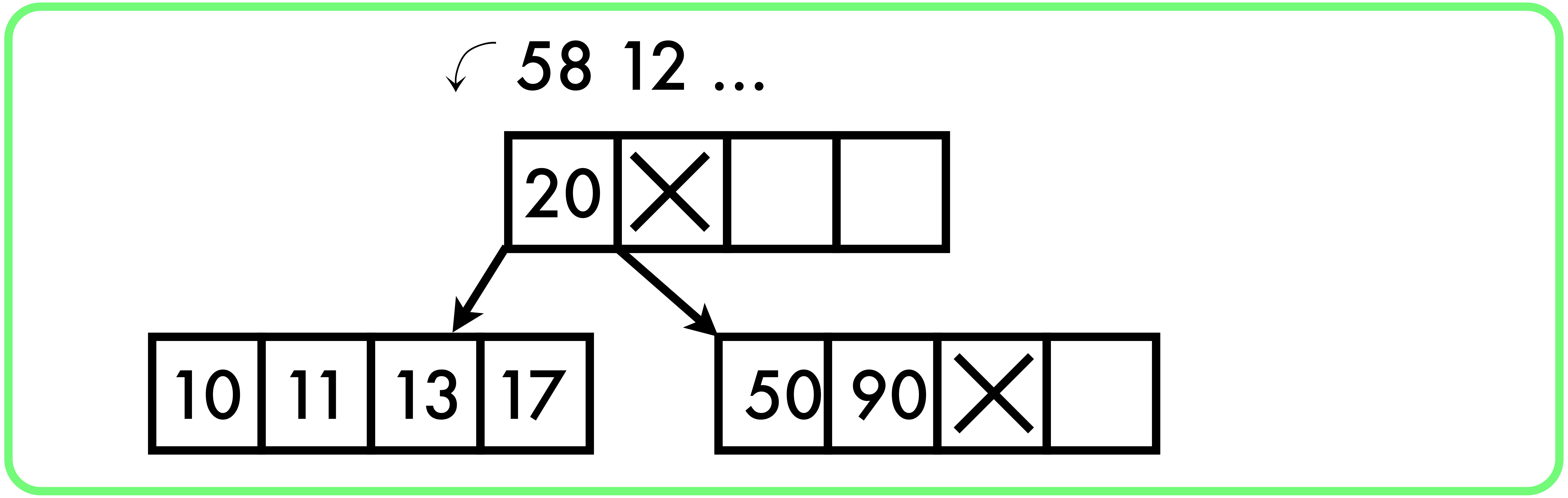


Volle nodes worden gesplitst
13 17 11 58 12 ...
10 20 50 90
De 'scheidende' sleutel wordt recursief naar boven gepropageerd
17 11 58 12 ...
20
10 13
50 90

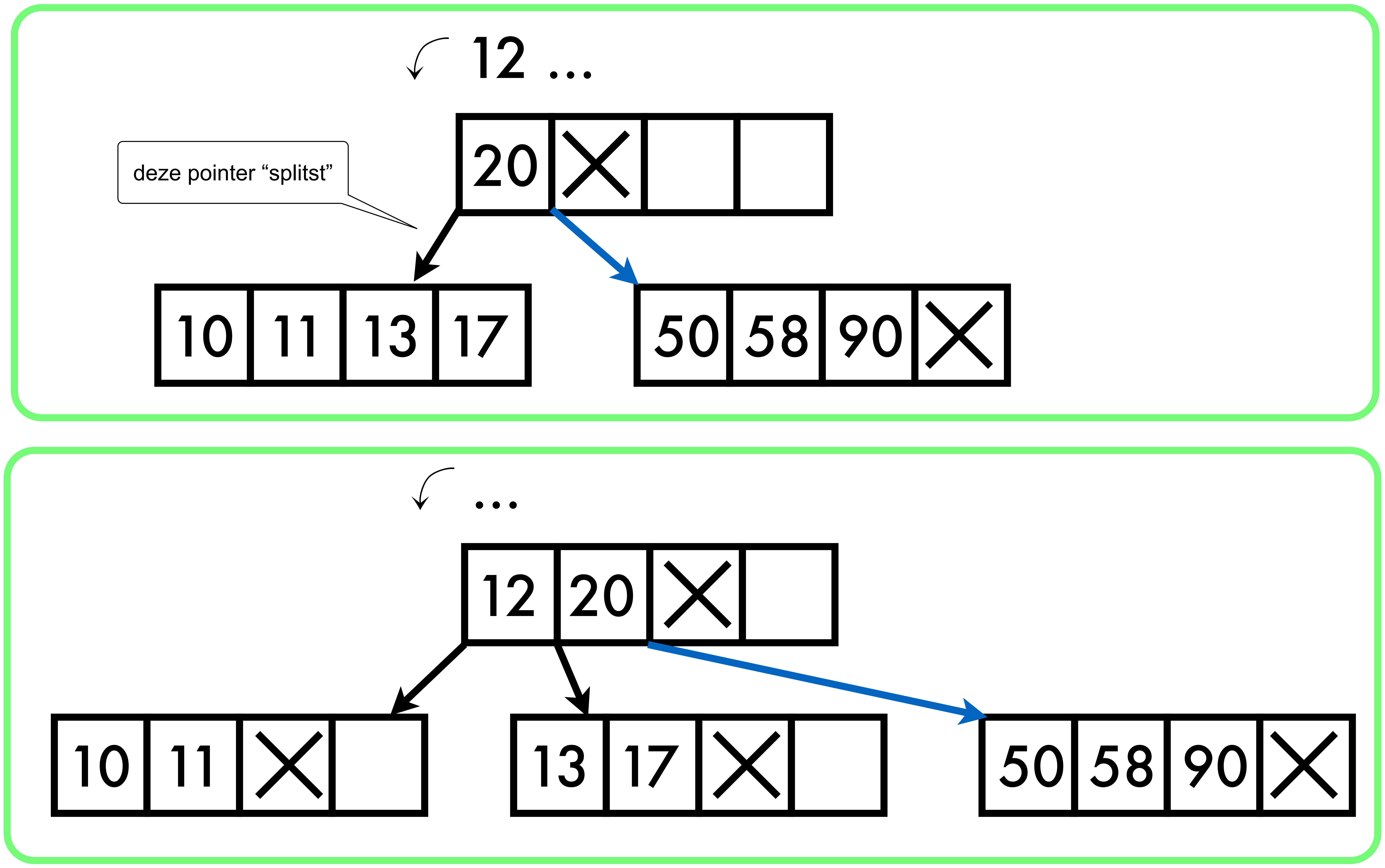
Voorbeeld

insert gebeurt volgens principe van binary search





Voorbeeld

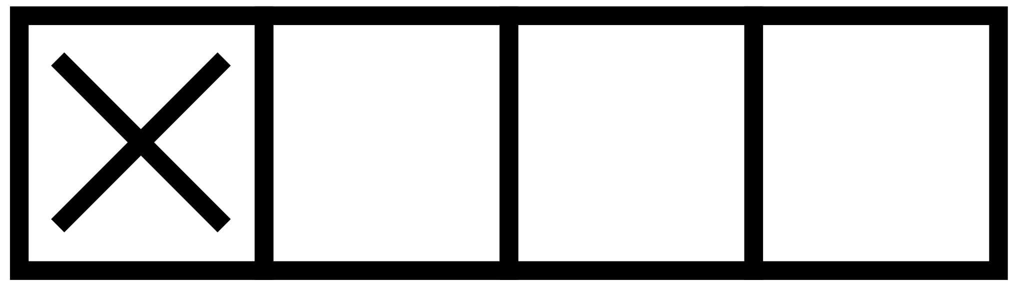


12 ...
deze pointer “splitst”
20
10
11
13
17
50
58
90
...
12
20
10
11
13
17
50
58
90

B+-Trees: Zelfde Voorbeeld

We inserten achtereenvolgens:
50 10 20 90 13 17 11 58 12 ...

50 10 20 90 13 17 11 58 12 ...



Voorbeeld

10 20 90 13 17 11 58 12 ...

50	X		

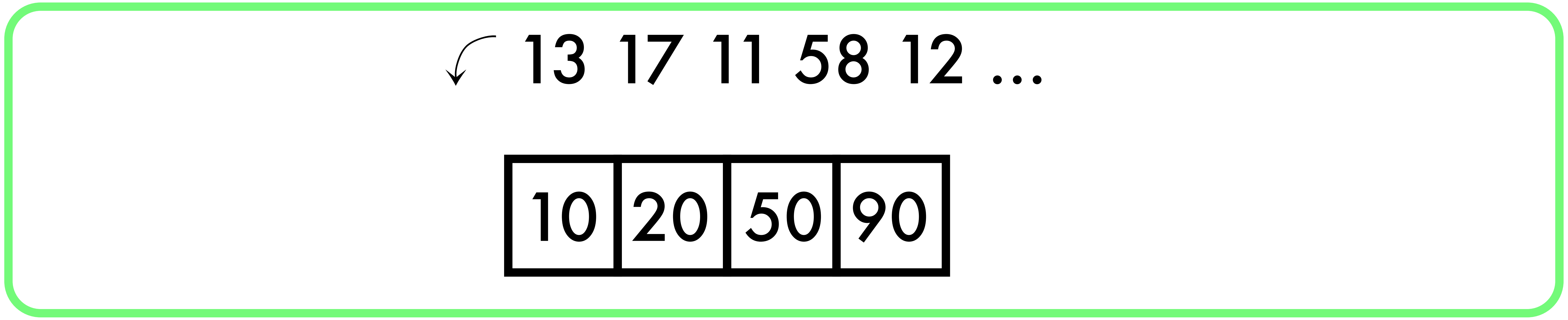
20 90 13 17 11 58 12 ...

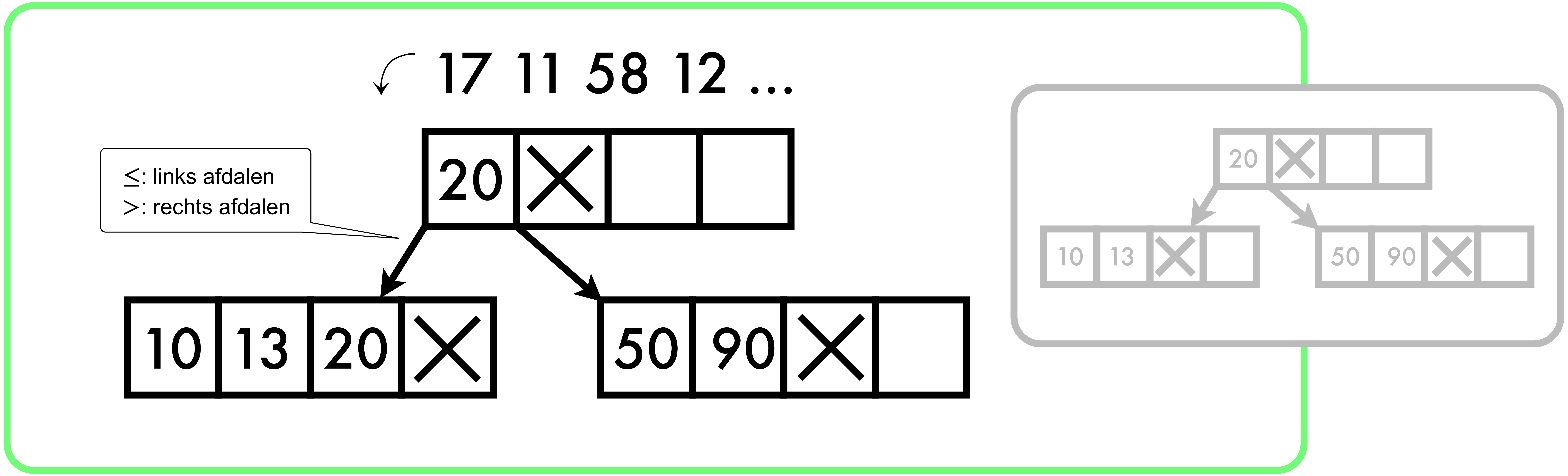
10	50	X	

90 13 17 11 58 12 ...

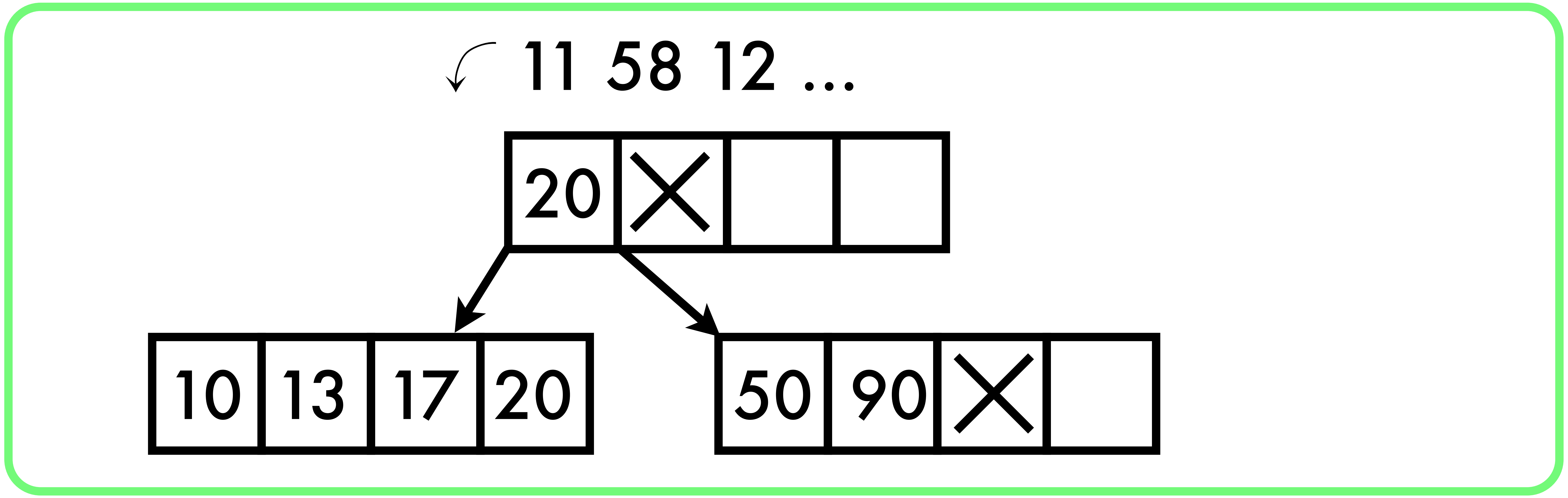
10	20	50	X

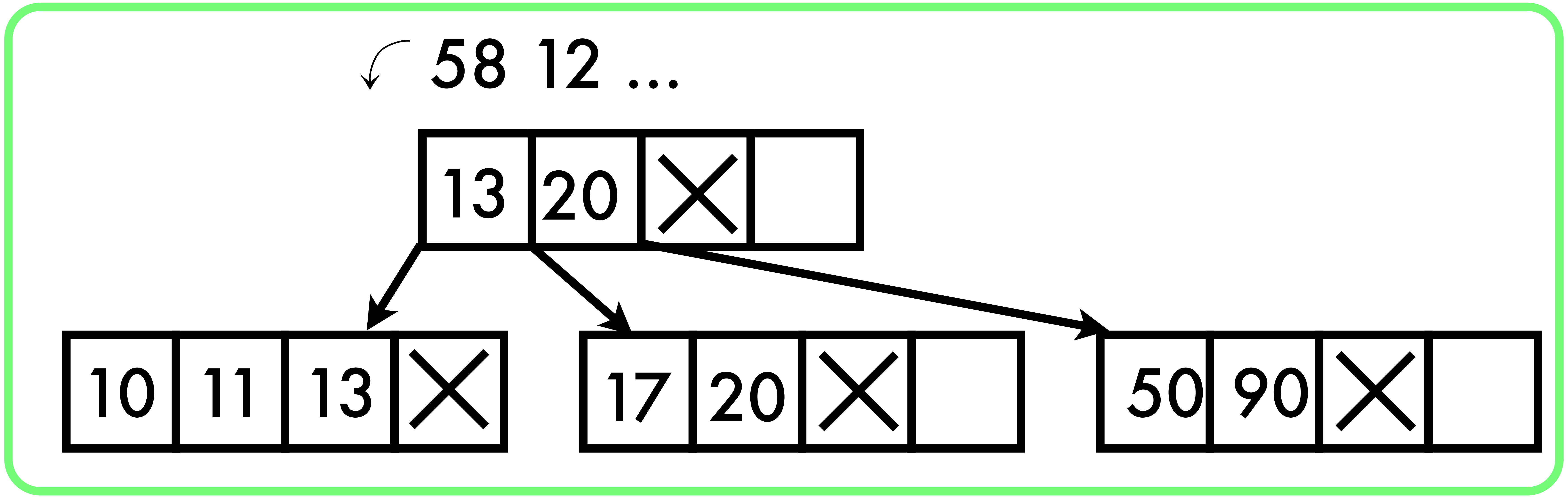
Voorbeeld


13 17 11 58 12 ...
10 20 50 90

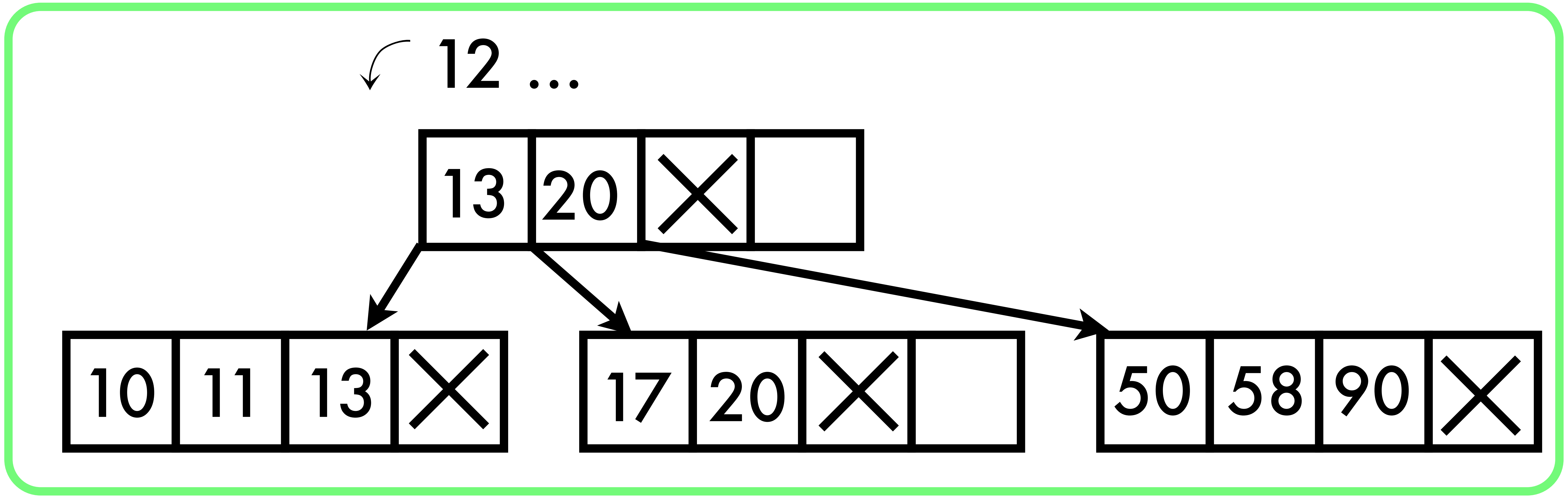

17 11 58 12 ...
≤: links afdalen
>: rechts afdalen
20
10 13 20
50 90
20
10 13
50 90

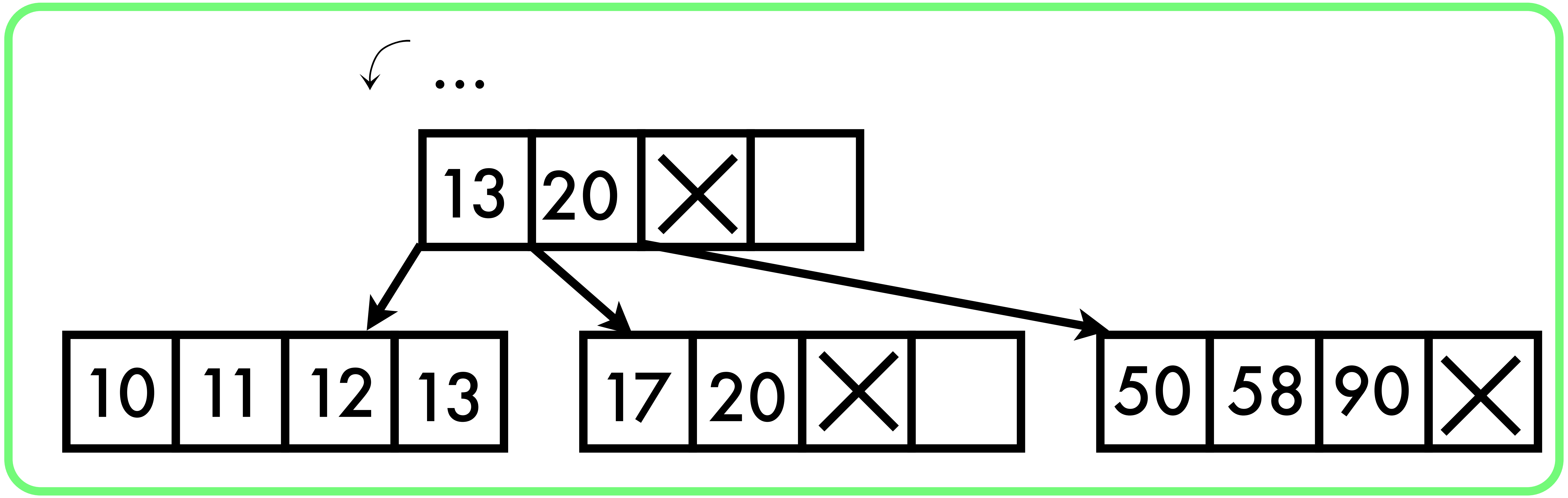
Voorbeeld



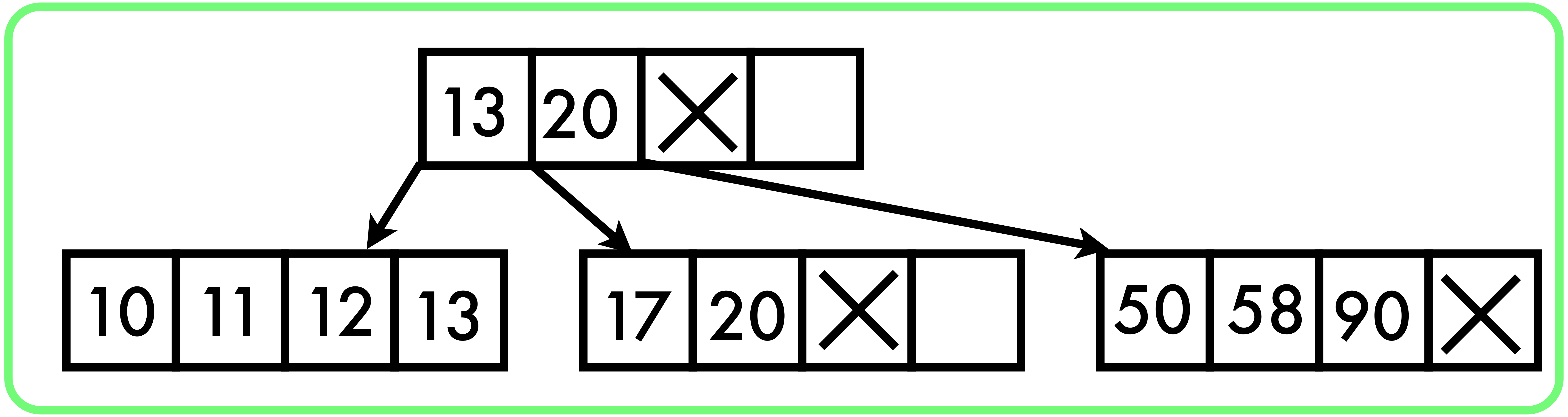


Voorbeeld

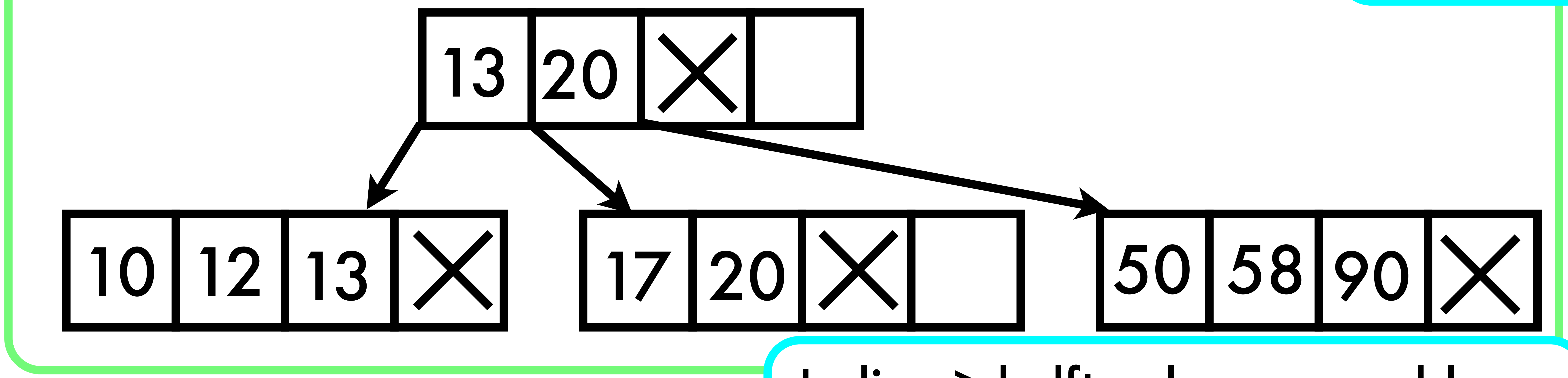




Delete uit B+-tree: Voorbeeld

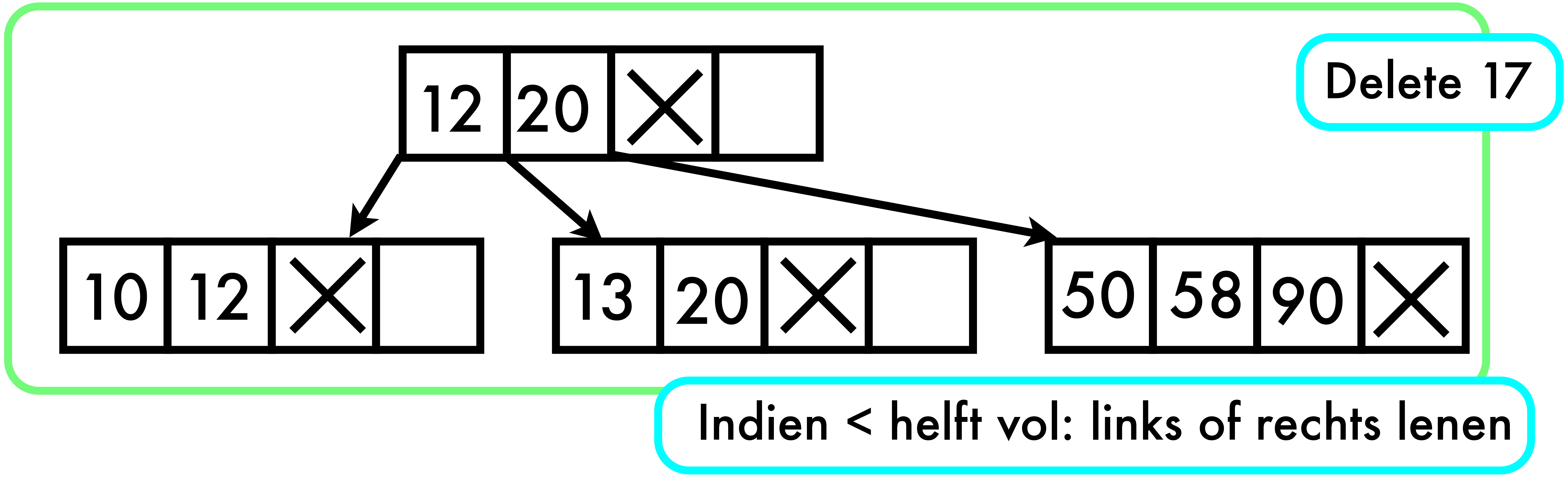


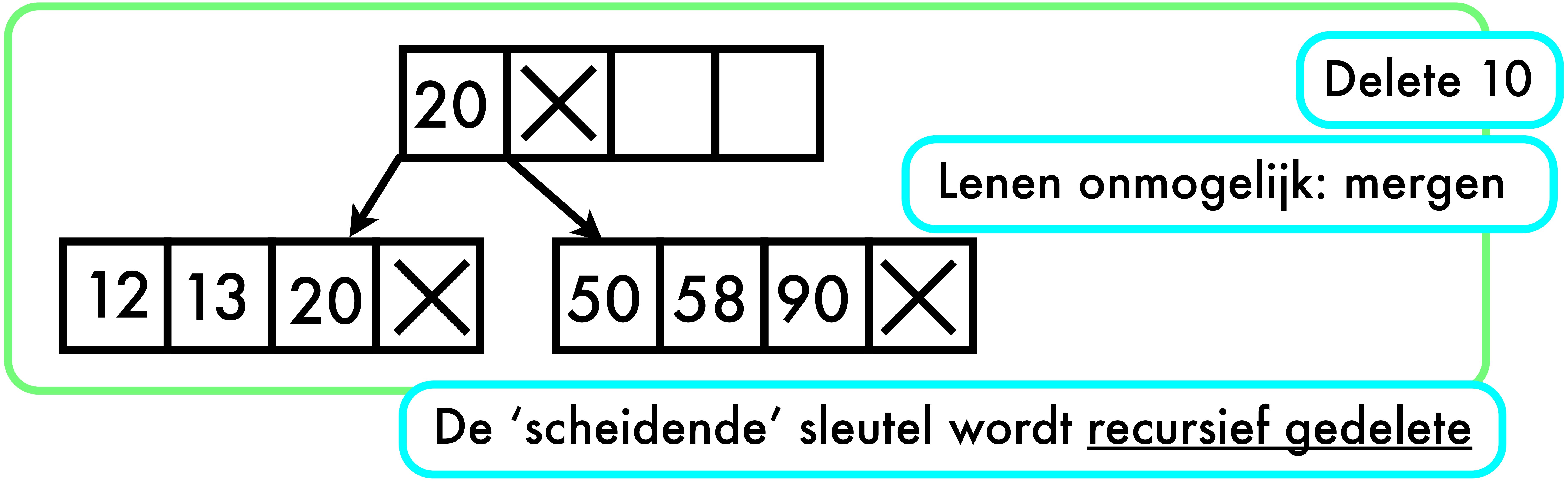
Delete 11



Indien ≥ helft vol: geen probleem

Voorbeeld


Delete 17
12 20
10 12
13 20
50 58 90
Indien < helft vol: links of rechts lenen


Delete 10
Lenen onmogelijk: mergen
20
12 13 20
50 58 90
De 'scheidende' sleutel wordt recursief gedelete

Samenvatting

Indien toevoeging in een blad een overflow oplevert, splitsen we de node in 2 en wordt de ‘scheidende’ sleutel naar boven toe gepropageerd en daar (recursief) toegevoegd.

Indien verwijdering uit een blad minder dan N/2 sleutels bevat, proberen we links of rechts te “lenen”; anders mergen we met links of rechts en wordt hun scheidende sleutel erboven (recursief) verwijderd.

B($^+$)-Trees

In een B($^+$)-Tree van orde N zitten in elke node (a) maximum N sleutels en (b) minimum N/2 sleutels. De enige uitzondering hierop is de root die tussen 1 en N sleutels kan bevatten.

Een B($^+$)-Tree is altijd gebalanceerd

Een B($^+$)-Tree groeit opwaarts en krimpt neerwaarts

Doel: Indexeren van Tabellen

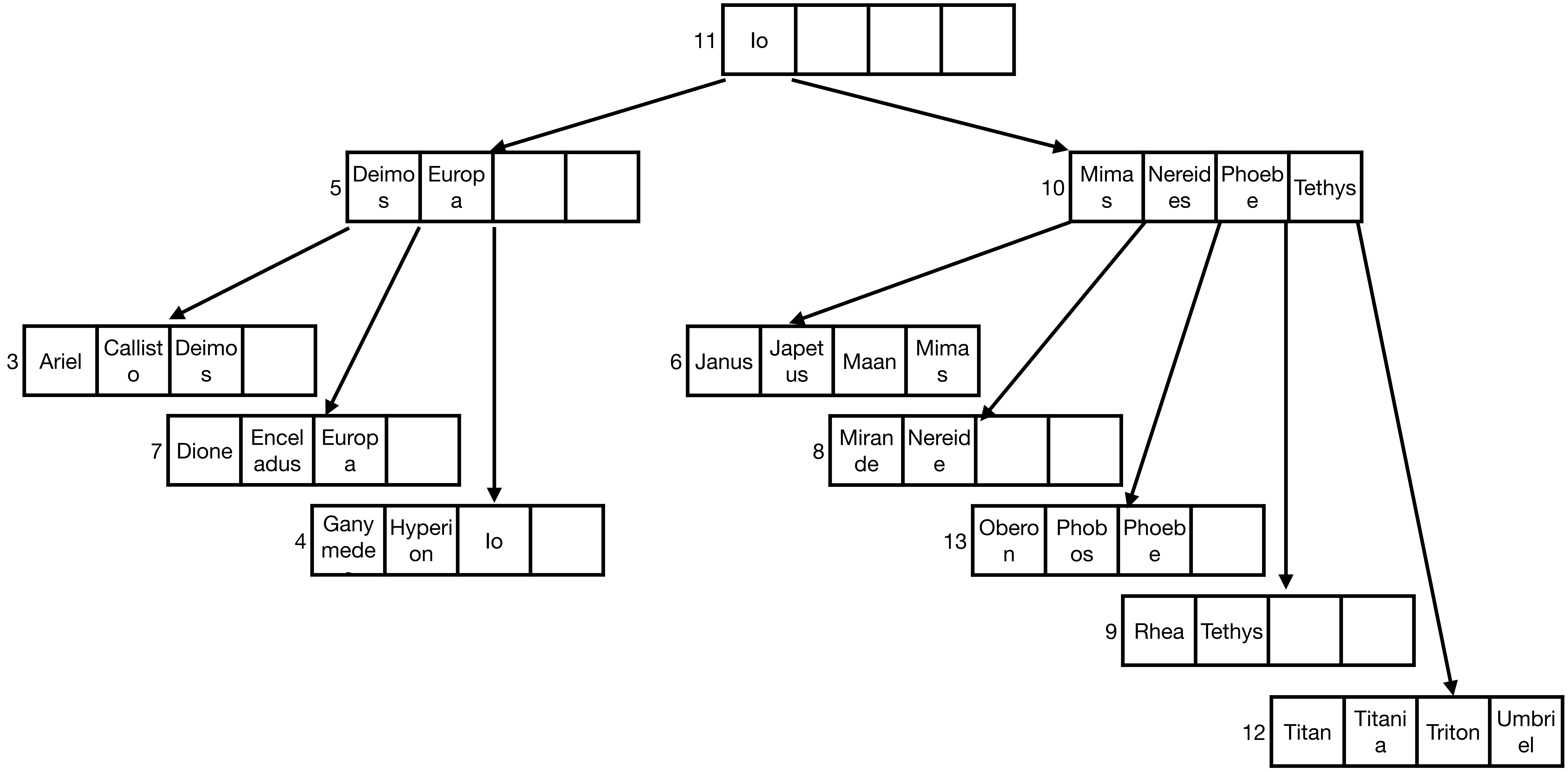
```
(define dsk (disk:new "treedisk"))
(fs:format! dsk)
(define d (b-tree:new dsk "Manen" string-tag 10))
```

```
(b-tree:insert! d "Maan"      rcid:null)
(b-tree:insert! d "Phobos"    rcid:null)
(b-tree:insert! d "Deimos"    rcid:null)
(b-tree:insert! d "Io"        rcid:null)
(b-tree:insert! d "Europa"    rcid:null)
(b-tree:insert! d "Ganymedes" rcid:null)
(b-tree:insert! d "Callisto"  rcid:null)
(b-tree:insert! d "Mimas"     rcid:null)
(b-tree:insert! d "Enceladus" rcid:null)
(b-tree:insert! d "Tethys"    rcid:null)
(b-tree:insert! d "Dione"     rcid:null)
(b-tree:insert! d "Rhea"      rcid:null)
(b-tree:insert! d "Titan"     rcid:null)
(b-tree:insert! d "Hyperion"  rcid:null)
(b-tree:insert! d "Japetus"   rcid:null)
(b-tree:insert! d "Phoebe"    rcid:null)
(b-tree:insert! d "Janus"     rcid:null)
(b-tree:insert! d "Ariel"     rcid:null)
(b-tree:insert! d "Umbriel"   rcid:null)
(b-tree:insert! d "Titania"   rcid:null)
(b-tree:insert! d "Oberon"    rcid:null)
(b-tree:insert! d "Miranda"   rcid:null)
(b-tree:insert! d "Triton"    rcid:null)
(b-tree:insert! d "Nereide"   rcid:null)
```

Test met `rcid:null`. Later met “echte” rcid’s.

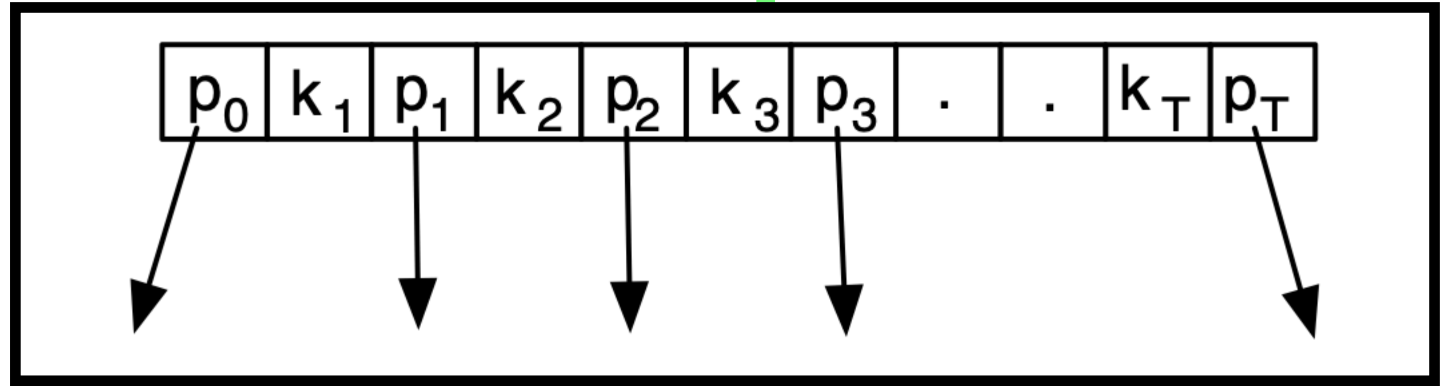
```
(define manenschema  '((string 9)   ; naam maan
                       (string 9)   ; naam planeet
                       (natural 2)  ; middellijn
                       (natural 2)  ; ontdekjaar
                       (string 10))); ontdekker
```

Corresponderende B+-tree



Corresponderende Disk Blokken

Een node = een blok




```
block[1] ADMIN: key-size= 10    key-type= 2    root    = 11

node[11] p0=5 k1=Io p1=10 k2=? p2=0 k3=? p3=0 k4=? p4=0

node[5]  p0=3 k1=Deimos p1=7 k2=Europa p2=4 k3=? p3=0 k4=? p4=0

node[3]  p0=0 k1=Ariel p1={0.0} k2=Callisto p2={0.0} k3=Deimos p3={0.0} k4=? p4={0.0}

node[7]  p0=0 k1=Dione p1={0.0} k2=Enceladus p2={0.0} k3=Europa p3={0.0} k4=? p4={0.0}

node[4]  p0=0 k1=Ganymedes p1={0.0} k2=Hyperion p2={0.0} k3=Io p3={0.0} k4=? p4={0.0}

node[10] p0=6 k1=Mimas p1=8 k2=Nereide p2=13 k3=Phoebe p3=9 k4=Tethys p4=12

node[6]  p0=0 k1=Janus p1={0.0} k2=Japetus p2={0.0} k3=Maan p3={0.0} k4=Mimas p4={0.0}

node[8]  p0=0 k1=Miranda p1={0.0} k2=Nereide p2={0.0} k3=? p3={0.0} k4=? p4={0.0}

node[13] p0=0 k1=Oberon p1={0.0} k2=Phobos p2={0.0} k3=Phoebe p3={0.0} k4=? p4={0.0}

node[9]  p0=0 k1=Rhea p1={0.0} k2=Tethys p2={0.0} k3=? p3={0.0} k4=? p4={0.0}

node[12] p0=0 k1=Titan p1={0.0} k2=Titania p2={0.0} k3=Triton p3={0.0} k4=Umbriel p4={0.0}
```

B+-tree Nodes: Layout

Conceptueel scheiden de keys de pointers.
Technisch zitten keys rechts en pointers links

Interne node:

b_0	b_1	...				...	k_2	k_1

pointer = block-pointer

Leaf-herkenning

Leaf node:

0	r_1	r_2				...	k_2	k_1

pointer = rcid

Hoe blok interpreteren als node?

```
(db:create-index! zonnestelsel planeten "Naam-IDX"   :planeet-naam:)
(db:create-index! zonnestelsel planeten "Omloop-IDX" :omlooptijd:)
```

node-type = dé sleutel om een blok (d.w.z. rij bytes) “correct” als node te lezen

```
ADT node-type

new
  ( disk byte byte → node-type )
node-type?
  ( any → boolean )
disk
  ( node-type → disk )
key-size
  ( node-type → byte )
key-type
  ( node-type → byte )
key-sent
  ( node-type → any )
leaf-capacity
  ( node-type → number )
internal-capacity
  ( node-type → number )
```

Type v/d sleutels

Grootte v/d sleutels

Aantal slots in één node

Essentie v/d Implementatie

pointer = block-pointer ∨ rcid

blok = block-pointer + cap×keys + cap×pointer

Leaf-herkenning

$\Longrightarrow$ cap = $\dfrac{\text{block-size - block-pointer-size}}{\text{key-size + pointer-size}}$

```
(define (new disk key-type key-size)
  (define leaf-node-cpty
    (quotient (- disk:block-size disk:block-ptr-size)
              (+ key-size rcid:size)))
  (define internal-node-cpty
    (quotient (- disk:block-size disk:block-ptr-size)
              (+ key-size disk:block-ptr-size)))
  (if (or (< leaf-node-cpty 2)
          (< internal-node-cpty 2))
      (error "key too large (new)" key-size)
      (make disk key-type key-size leaf-node-cpty internal-node-cpty)))
```

Node = { Key/Pntr }

```
ADT node

new
 ( node-type boolean → node )
read
 ( node-type number → node )
write!
 ( node → ∅ )
delete!
 ( node → ∅ )
position
 ( node → number )
type
 ( node → node-type )
capacity
 ( node → number )
size
 ( node → number )
meaningless?
 ( node number → boolean )
leaf?
 ( node → boolean )
locate-leftmost
 ( node any → number )
```

Huidige bezetting

pntr = rcid ∪ bptr

```
key
 ( node number → any )
key!
 ( node number any → ∅ )
pointer
 ( node number → pntr )
pointer!
 ( node number pntr → ∅ )
key-pointer!
 ( node number any pntr → ∅ )
key-pointer-insert!
 ( node number any pntr → ∅ )
key-pointer-delete!
 ( node number → ∅ )
key-pointer-insert-split!
 ( node node number any
   pntr boolean          → any )
borrow-from-left?
 ( node node any → any ∪ { #f } )
borrow-from-right?
 ( node node any → any ∪ { #f } )
merge
 ( node node any → ∅ )
```

Nodes Maken

```
(define-record-type node
  (make t b)
  node?
  (t type)
  (b block))
```

?	0	...				...	∞	∞

```
(define (sentinel-for ktyp ksiz)
  (cond ((= ktyp natural-tag)
         (- (expt 256 ksiz) 1))
        ((= ktyp integer-tag)
         (- (div (expt 256 ksiz) 2) 1))
        ((= ktyp decimal-tag)
         +inf.0)
        ((= ktyp string-tag)
         (utf8-sentinel-for ksiz))))
```

+∞

```
(define (new ntyp leaf)
  (define disk (ntype:disk ntyp))
  (define ktyp (ntype:key-type ntyp))
  (define ksiz (ntype:key-size ntyp))
  (define blck (fs:new-block disk))
  (define node (make ntyp blck))
  (define sent (sentinel-for ktyp ksiz))
  (pointer! node 0 (if (not leaf) 1 fs:null-block))
  (do ((null (null-ptr-for node))
       (slot 1 (+ slot 1 )))
    ((> slot (capacity node)))
    (key-pointer! node slot sent null))
  node)
```

```
(define (null-ptr-for node)
  (if (leaf? node)
      rcid:null
      fs:null-block))
```

```
(define (read ntyp bptr)
  (define disk (ntype:disk ntyp))
  (define blck (disk:read-block disk bptr))
  (make ntyp blck))
```

Nodes op de Disk

Eigenlijk gewoon de operaties doorvertalen naar blok-niveau

```
(define (delete! node)
  (fs:delete-block (block node)))

(define (position node)
  (disk:position (block node)))

(define (write! node)
  (disk:write-block! (block node)))
```

```
(define (key-pointer! node slot skey pntr)
  (key!     node slot skey)
  (pointer! node slot pntr))
```

Handig

Keys Lezen/Schrijven

```
(define (key node slot)
  (define ksiz (ntype:key-size (type node)))
  (define ktyp (ntype:key-type (type node)))
  (define blck (block node))
  (define offs (- disk:block-size (* ksiz slot)))
  (define decoder (vector-ref decoders ktyp))
  (decoder blck offs ksiz))

(define (key! node slot skey)
  (define ksiz (ntype:key-size (type node)))
  (define ktyp (ntype:key-type (type node)))
  (define blck (block node))
  (define offs (- disk:block-size (* ksiz slot)))
  (define encoder! (vector-ref encoders ktyp))
  (encoder! blck offs ksiz skey))
```

keys zitten rechts ⇒

block-size - ...

```
(define encoders  (vector disk:encode-fixed-natural!
                          disk:encode-arbitrary-integer!
                          disk:encode-real!
                          disk:encode-string!))
```

```
(define decoders  (vector disk:decode-fixed-natural
                          disk:decode-arbitrary-integer
                          disk:decode-real
                          disk:decode-string))
```

Zie H16

Pointers Lezen/Schrijven

```
(define (pointer node slot)
  (define blck (block node))
  (define rid? (and (not (= slot 0)) (leaf? node)))
  (define pntr-size (if rid?
                        rcid:size
                        disk:block-pointer-size))
  (define offs (* pntr-size slot))
  (define bptr (disk:decode-fixed-natural blck offs pntr-size))
  (if rid? (rcid:fixed->rcid bptr) bptr))

(define (pointer! node slot pntr)
  (define blck (block node))
  (define rid? (and (not (= slot 0)) (leaf? node)))
  (define pntr-size (if rid?
                        rcid:size
                        disk:block-pointer-size))
  (define offs (* pntr-size slot))
  (disk:encode-fixed-natural! blck offs pntr-size (if rid?
                                                      (rcid:rcid->fixed pntr)
                                                      pntr)))
```

pointers zitten links

⇒ 0 + ...

(rcids als fixed getal wegschrijven)

Zoeken in een Node

```
(define (locate-leftmost node skey)
  (define ntyp (type node))
  (define ktyp (ntype:key-type ntyp))
  (define <<<? (vector-ref smaller ktyp))
  (define >>>? (vector-ref greater ktyp))
  (define (search first last)
    (if (> first last)
        last
        (let*
            ((mid (div (+ first last) 2))
             (mid-key (key node mid)))
          (cond
            ((>>>? skey mid-key)
             (search (+ mid 1) last))
            ((<<<? skey mid-key)
             (search first (- mid 1)))
            (else
             (let ((try (search first (- mid 1))))
               (if (negative? try)
                   try
                   (- mid))))))))
  (search 1 (ntype:capacity ntyp)))
```

```
(define equals  (vector = = = string=?))
(define smaller (vector < < < string<?))
(define greater (vector > > > string>?))
```

Ga om met duplicaten

Gevonden? Zoek links naar identieke

gevonden ⇒ - slotnr

¬gevonden ⇒ verwacht slotnr

Node Informatie

```scheme
(define (complement slot)
  (if (negative? slot)
      (- -1 slot)
      slot))
```

```scheme
(define (size node)
  (define ncap (capacity node))
  (define sent (ntype:key-sent (type node)))
  (define indx (locate-leftmost node sent))
  (if (negative? indx)
      (complement indx)
      ncap))
```

Size van een node = meest linkse voorkomen van +∞

```scheme
(define (capacity node)
  (define ntyp (type node))
  (if (leaf? node)
      (ntype:leaf-capacity ntyp)
      (ntype:internal-capacity ntyp)))
```

```scheme
(define (meaningless? node slot)
  (define ntyp (type node))
  (define ktyp (ntype:key-type ntyp))
  (define sent (ntype:key-sent ntyp))
  (define ===? (vector-ref equals ktyp))
  (if (= slot (capacity node))
    #t                  ; 0 <= slot < cap
    (===? (key node (+ slot 1)) sent)))
```

Invoegen van Key-Pointer-paar

```
(define (key-pointer-insert! node slot skey pntr)
  (define nsiz (ntype:capacity (type node)))
  (define (move index)
    (if (> index slot)
        (let ((previous-index (- index 1)))
          (key-pointer! node index
                        (key node previous-index)
                        (pointer node previous-index))
          (move previous-index))))
  (move nsiz)
  (key-pointer! node slot skey pntr))
```

storage move

vanaf `slot`

naar rechts

paar invoegen

Uitvegen van Key-Pointer-paar

```
(define (key-pointer-delete! node slot)
  (define nsiz (ntype:capacity (type node)))
  (define ktyp (ntype:key-type (type node)))
  (define ksiz (ntype:key-size (type node)))
  (define sent (sentinel-for ktyp ksiz))
  (define nulp (null-ptr-for node))
  (define (move index)
    (if (< index nsiz)
        (let ((next-index (+ index 1)))
          (key-pointer! node index
                        (key node next-index)
                        (pointer node next-index))
          (move next-index))))
  (move slot)
  (key-pointer! node nsiz sent nulp))
```

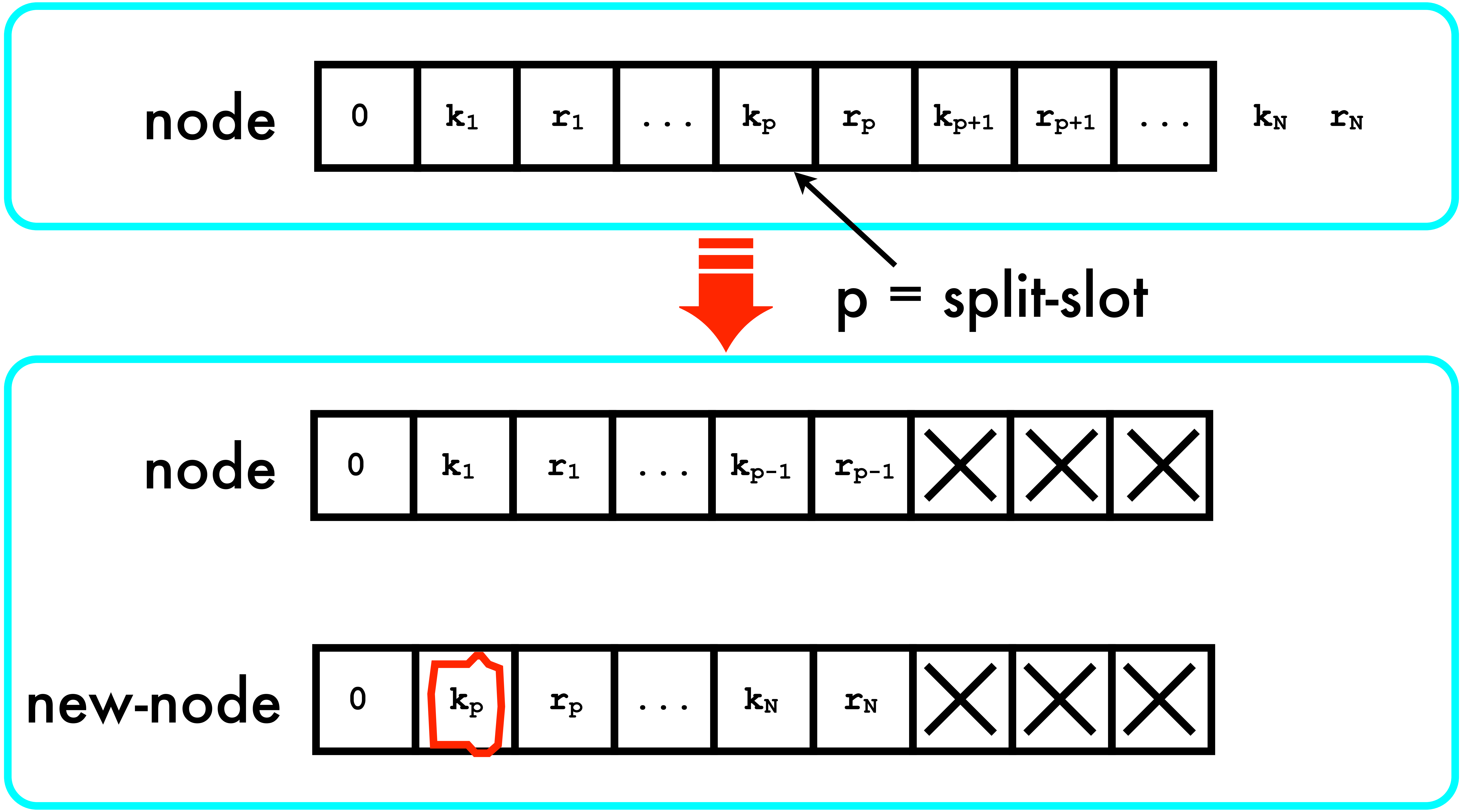
storage move

vanaf einde

naar links

laatste vakje betekenisvol maken

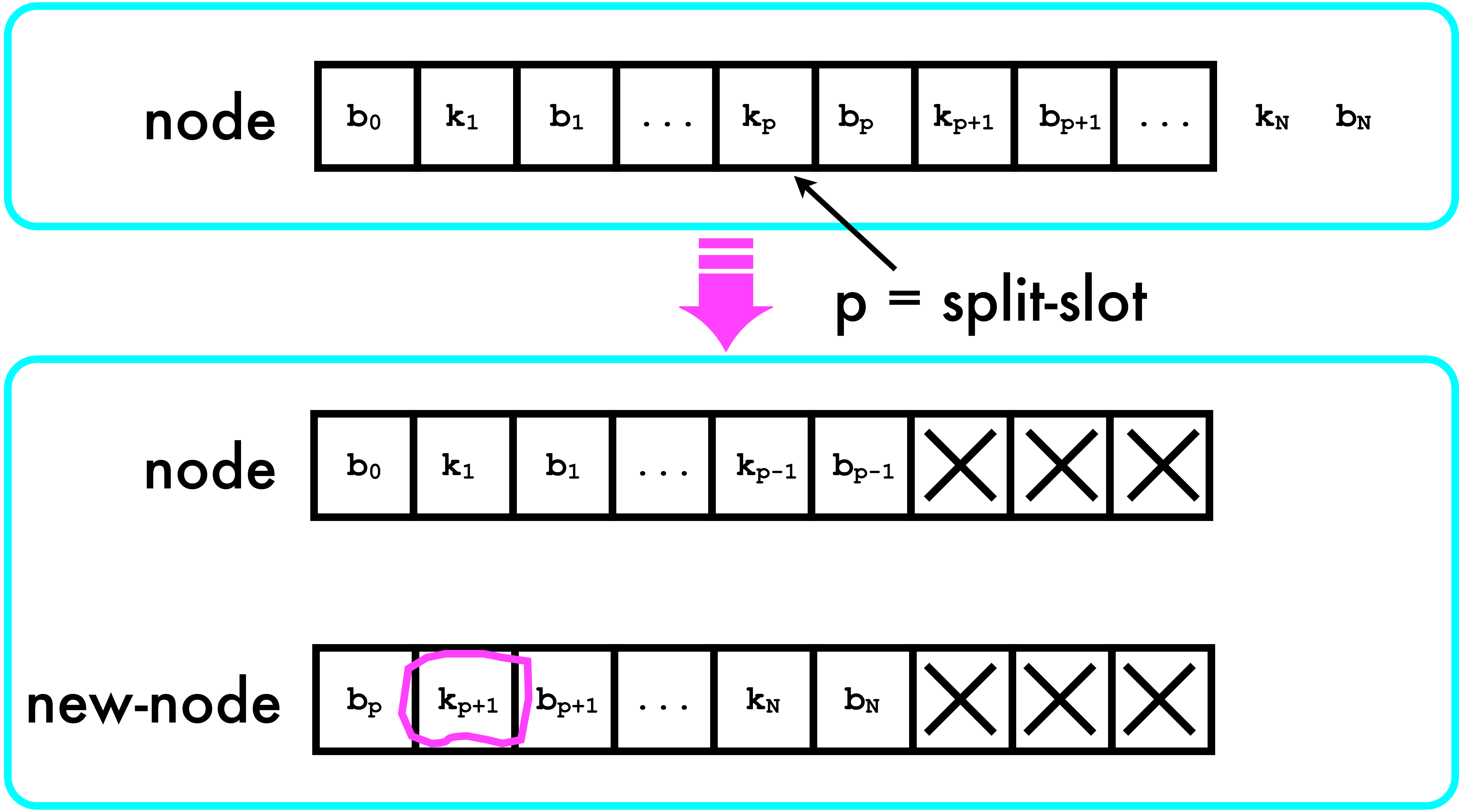
Leaf Nodes Splitsen



paren vanaf p kopieren

resultaat: k_{p-1}

Interne Nodes Splitsen



paren vanaf p+1 kopieren

resultaat: k_p

Nodes Splitsen

```
(define (key-pointer-insert-split! node new-node slot skey pntr leaf)
  (define nsiz (ntype:capacity (type node)))
  (define ktyp (ntype:key-type (type node)))
  (define ksiz (ntype:key-size (type node)))
  (define sent-skey (sentinel-for ktyp ksiz))
  (define nptr (null-ptr-for node))
  (define split-slot (+ (div (+ nsiz 1) 2) 1))
  (define at-end (> slot nsiz))
  (define hold-key (if at-end skey (key node nsiz)))
  (define hold-datum (if at-end pntr (pointer node nsiz)))
  (define (move slot new-slot)
    (cond
      ((<= slot nsiz)
       (key-pointer! new-node new-slot
                     (key node slot)
                     (pointer node slot))
       (move (+ slot 1) (+ new-slot 1)))
      (else
       new-slot)))
  (define (clear slot)
    (when (<= slot nsiz)
      (key-pointer! node slot sent-skey nptr)
      (clear (+ slot 1))))
  ...)
```

k_N

d_N

Kopieer (slot→...)
naar (new-slot→...)

Veeg de paren
uit van slot tot
vanachter

Nodes Splitsen (vervolg)

```
(define (key-pointer-insert-split! node new-node slot skey pntr leaf)
  ...
  (define split-slot (+ (div (+ nsiz 1) 2) 1))
  (define at-end (> slot nsiz))
  (define hold-key (if at-end skey (key node nsiz)))
  (define hold-datum (if at-end pntr (pointer node nsiz)))
  (define (move slot new-slot)
    ...)
  (define (clear slot)
    ...)
  (if (not at-end) (key-pointer-insert! node slot skey pntr))
  (let*
      ((prop-key (key node (if leaf (- split-slot 1) split-slot)))
       (insert-slot
        (cond
          (leaf
           (pointer! new-node 0 fs:null-block)
           (move split-slot 1))
          (else
           (pointer! new-node 0 (pointer node split-slot))
           (move (+ split-slot 1) 1)))))
    (key-pointer! new-node insert-slot hold-key hold-datum)
    (clear split-slot)
    prop-key))
```

∈ [1 ...]

k_N

d_N

k_{p-1} k_p

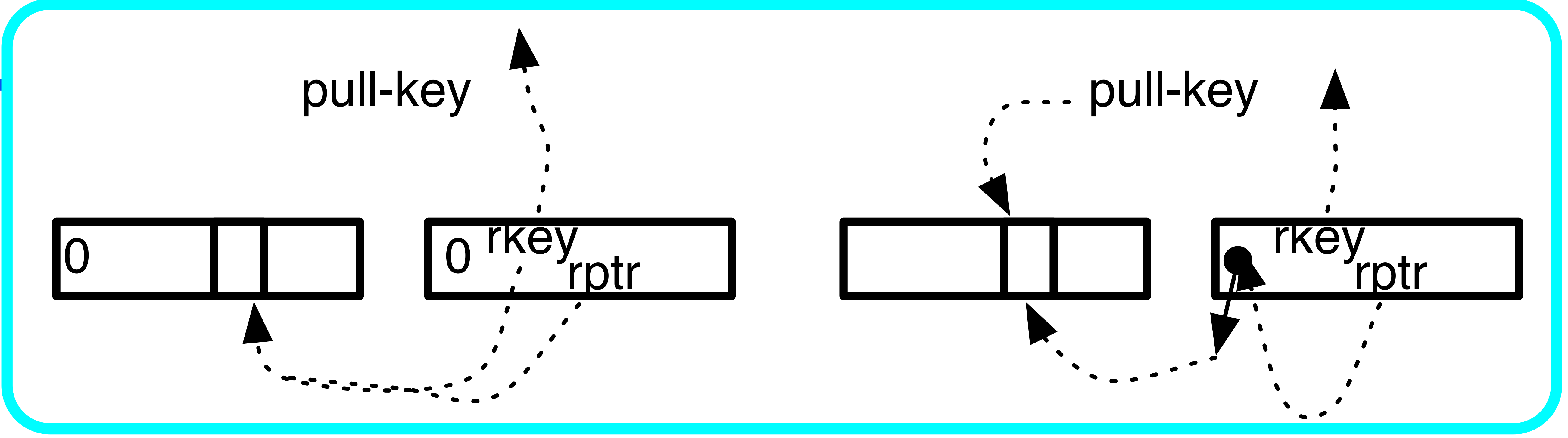
laatste slot

Lenen Rechts Mogelijk?

```
(define (borrow-from-right? node right pull-skey)
  (define rsiz (size right))
  (define cpty (ntype:capacity (type right)))
  (define nsiz (size node))
  (if (< (- rsiz 1) (quotient cpty 2))
      #f
      (let* ((rkey (key right 1))
             (rptr (pointer right 1))
             (lkey (if (leaf? node)
                       rkey
                       pull-skey))
             (lptr (if (leaf? node)
                       rptr
                       (pointer right 0))))
        (key-pointer-insert! node (+ nsiz 1) lkey lptr)
        (key-pointer-delete! right 1)
        (unless (leaf? node)
          (pointer! right 0 rptr))
        rkey)))
```

Enkel indien geen underflow

Pak de eerste



Lenen Links Mogelijk?

```
(define (borrow-from-left? left node pull-skey)
  (define lsiz (size left))
  (define cpty (capacity left))
  (if (< (- lsiz 1) (div cpty 2))
      #f
      (let* ((lkey (if (leaf? node)
                       (key left lsiz)
                       pull-skey))
             (lptr (pointer left lsiz))
             (prop-key (if (leaf? node)
                           (key left (- lsiz 1))
                           lkey))
             (rptr (if (leaf? node)
                       lptr
                       (pointer node 0))))
        (key-pointer-insert! node 1 lkey rptr)
        (key-pointer-delete! left lsiz)
        (unless (leaf? node)
          (pointer! node 0 lptr))
        prop-key)))
```

Enkel indien geen underflow

Pak de laatste

Oefening: maak een gelijkaardige tekening

Mergen van 2 Nodes

```
(define (merge accu-node node pull-skey)
  (define cpty (ntype:capacity (type node)))
  (define asiz (size accu-node))
  (define nsiz (size node))
  (define strt (if (leaf? node)
                   (+ asiz 1)
                   (+ asiz 2)))
  (if (not (leaf? node))
      (key-pointer! accu-node (+ asiz 1) pull-skey (pointer node 0)))
  (do ((indx strt (+ indx 1)))
    ((= (- indx strt -1) (+ nsiz 1)))
    (key-pointer! accu-node indx (key node 1) (pointer node 1))
    (key-pointer-delete! node 1)))
```

Alles wordt in accu-node gekopieerd

Het B+-Tree ADT

```
ADT b-tree

new
  ( disk string byte byte → b-tree )
open
  ( disk string → b-tree )
b-tree?
  ( any → boolean )
drop!
  ( b-tree → ∅ )
flush!
  ( b-tree → ∅ )
```

type tag

key size

```
status = { done,
           no-current,
           next-higher,
           not-found }
```

```
insert!
  ( b-tree any rcid → status )
set-current-to-first!
  ( b-tree → status )
find!
  ( b-tree any → status )
set-current-to-next!
  ( b-tree → status )
delete!
  ( b-tree → status )
peek
  ( b-tree → any × rcid )
update!
  ( b-tree rcid → status )
```

De operaties werken t.o.v. een current

Testvoorbeeld (herhaling)

```
(define dsk (disk:new "treedisk"))
(fs:format! dsk)
(define disk-size (fs:df dsk))
(define d (b-tree:new dsk "Manen" string-tag 10))

(b-tree:insert! d "Maan"      rcid:null)
(b-tree:insert! d "Phobos"    rcid:null)
(b-tree:insert! d "Deimos"    rcid:null)
(b-tree:insert! d "Io"        rcid:null)
(b-tree:insert! d "Europa"    rcid:null)
(b-tree:insert! d "Ganymedes" rcid:null)
(b-tree:insert! d "Callisto"  rcid:null)
(b-tree:insert! d "Mimas"     rcid:null)
(b-tree:insert! d "Enceladus" rcid:null)
(b-tree:insert! d "Tethys"    rcid:null)
(b-tree:insert! d "Dione"     rcid:null)
(b-tree:insert! d "Rhea"      rcid:null)
(b-tree:insert! d "Titan"     rcid:null)
(b-tree:insert! d "Hyperion"  rcid:null)
```

Test met `rcid:null`. Later met “echte” rcid’s.

```
(b-tree:insert! d "Japetus" rcid:null)
(b-tree:insert! d "Phoebe"  rcid:null)
(b-tree:insert! d "Janus"   rcid:null)
(b-tree:insert! d "Ariel"   rcid:null)
(b-tree:insert! d "Umbriel" rcid:null)
(b-tree:insert! d "Titania" rcid:null)
(b-tree:insert! d "Oberon"  rcid:null)
(b-tree:insert! d "Miranda" rcid:null)
(b-tree:insert! d "Triton"  rcid:null)
(b-tree:insert! d "Nereide" rcid:null)
```

Correponderende Disk Blocks

```
block[1] ADMIN: key-size= 10    key-type= 2    root    = 11

node[11] p0=5 k1=Io p1=10 k2=? p2=0 k3=? p3=0 k4=? p4=0

node[5]  p0=3 k1=Deimos p1=7 k2=Europa p2=4 k3=? p3=0 k4=? p4=0

node[3]  p0=0 k1=Ariel p1={0.0} k2=Callisto p2={0.0} k3=Deimos p3={0.0} k4=? p4={0.0}

node[7]  p0=0 k1=Dione p1={0.0} k2=Enceladus p2={0.0} k3=Europa p3={0.0} k4=? p4={0.0}

node[4]  p0=0 k1=Ganymedes p1={0.0} k2=Hyperion p2={0.0} k3=Io p3={0.0} k4=? p4={0.0}

node[10] p0=6 k1=Mimas p1=8 k2=Nereide p2=13 k3=Phoebe p3=9 k4=Tethys p4=12

node[6]  p0=0 k1=Janus p1={0.0} k2=Japetus p2={0.0} k3=Maan p3={0.0} k4=Mimas p4={0.0}

node[8]  p0=0 k1=Miranda p1={0.0} k2=Nereide p2={0.0} k3=? p3={0.0} k4=? p4={0.0}

node[13] p0=0 k1=Oberon p1={0.0} k2=Phobos p2={0.0} k3=Phoebe p3={0.0} k4=? p4={0.0}

node[9]  p0=0 k1=Rhea p1={0.0} k2=Tethys p2={0.0} k3=? p3={0.0} k4=? p4={0.0}

node[12] p0=0 k1=Titan p1={0.0} k2=Titania p2={0.0} k3=Triton p3={0.0} k4=Umbriel p4={0.0}
```

Een expliciet pad in de B+-Tree

Nodig voor set-current-to-next!

```
(define new    stck:new)

(define empty? stck:empty?)

(define pop!  stck:pop!)

(define (push! stck node slot)
  (stck:push! stck (cons node slot)))

(define (node stck)
  (car (stck:top stck)))

(define (slot stck)
  (cdr (stck:top stck)))

(define (clear! stck)
  (let loop
    ()
    (when (not (empty? stck))
      (pop! stck)
      (loop))))
```

(node,slot)-paren geven aan in welke node welke pointer gevolgd werd

Komt overeen met de recursiestapel

Duale B+-Tree Representatie

```
(define-record-type b-tree
  (make n h p t)
  b-tree?
  (n name name!)
  (h header header!)
  (p path path!)
  (t node-type node-type!))
```

Een B-tree houdt het pad naar "zijn current" bij

```
(define (key-type tree)
  ...)

(define (key-type! tree ktyp)
  ...)

(define (key-size tree)
  ...)

(define (key-size! tree ksiz)
  ...)

(define (tree-root tree)
  ...)

(define (tree-root! tree root)
  ...)
```

Een block met 3 slots

Header bevat: de key-size, key-type en het bloknummer van de root

Aanmaken/Openen van de B+-Tree

```
(define (new disk name ktyp ksiz)
  (define ntyp (ntype:new disk ktyp ksiz))
  (define stck (path:new))
  (define hder (fs:new-block disk))
  (define tree (make name hder stck ntyp))
  (key-size!  tree ksiz)
  (key-type!  tree ktyp)
  (tree-root! tree fs:null-block)
  (fs:mk disk name (disk:position hder))
  (disk:write-block! hder)
  tree)
```

```
(define (open disk name)
  (define hptr (fs:whereis disk name))
  (define stck (path:new))
  (define hder (disk:read-block disk hptr))
  (define tree (make name hder stck ()))
  (define ksiz (key-size tree))
  (define ktyp (key-type tree))
  (define ntyp (ntype:new disk ktyp ksiz))
  (node-type! tree ntyp)
  tree)
```

Een B+-Tree Sluiten/Droppen

```
(define (flush! indx)
  (disk:write-block! (header indx)))
```

Naieve Implementatie

```
(define (drop! tree)
  (define ntyp (node-type tree))
  (define root (tree-root tree))
  (define (rec-delete bptr)
    (define node (node:read ntyp bptr))
    (define head (node:pointer node 0))
    (cond ((node:leaf? node)
           (node:delete! node))
          (else
           (rec-delete head)
           (do ((slot 1 (+ slot 1)))
             ((fs:null-block? (node:pointer node slot))
              (node:delete! node))
             (rec-delete (node:pointer node slot))))))
  (if (not (fs:null-block? (tree-root tree)))
      (rec-delete (tree-root tree)))
  (fs:delete-block (header tree))
  (fs:rm (ntype:disk ntyp) (name tree)))
```

Oefening: verwijder recursie

Zoeken

```
(define (find! tree key)
  (define stck (path tree))
  (define (build-path bptr)
    (define node (node:read (node-type tree) bptr))
    (define slot (node:locate-leftmost node key))
    (define actual-slot (node:complement slot))
    (cond
      ((node:leaf? node)
       (cond
         ((negative? slot)
          (path:push! stck node (+ actual-slot 1))
          done)
         ((node:meaningless? node actual-slot)
          (path:clear! stck)
          not-found)
         (else
          (path:push! stck node (+ actual-slot 1))
          next-higher)))
      (else
       (path:push! stck node actual-slot)
       (build-path (node:pointer node actual-slot)))))
  (path:clear! stck)
  (if (fs:null-block? (tree-root tree))
      not-found
      (build-path (tree-root tree))))
```

gevonden!

leaf

¬leaf

Het eerste key/pointer paar

```
(define (set-current-to-first! tree)
  (define ntyp (node-type tree))
  (define stck (path tree))
  (define (build-path bptr)
    (define node (node:read ntyp bptr))
    (cond ((node:leaf? node)
           (path:push! stck node 1)
           done)
          (else
           (path:push! stck node 0)
           (build-path (node:pointer node 0)))))
  (path:clear! stck)
  (if (fs:null-block? (tree-root tree))
      no-current
      (build-path (tree-root tree))))
```

Wandel "meestlinks" naar beneden

Het volgende key/pointer paar

```
(define (set-current-to-next! tree)
  (define stck (path tree))
  (define ntyp (node-type tree))
  (define (backtrack level)
    (define node (path:node stck))
    (define slot (path:slot stck))
    (path:pop! stck)
    (if (node:meaningless? node slot)
        (if (path:empty? stck)
            not-found
            (backtrack (+ level 1)))
        (advance node (+ slot 1) level)))
  (define (advance node slot level)
    (path:push! stck node slot)
    (if (= 0 level)
        done
        (let*
            ((bptr (node:pointer node slot))
             (next-node (node:read ntyp bptr)))
          (if (= level 1)
              (advance next-node 1 0)
              (advance next-node 0 (- level 1))))))
  (if (path:empty? stck)
      no-current
      (backtrack 0)))
```

naar boven

naar rechts

log de gevolgde weg

leaf: gedaan (net gelogd)

vanaf 1/0 zoeken
in intern/leaf

Bewerkingen t.o.v. current

```
(define (peek tree)
  (define stck (path tree))
  (if (path:empty? stck)
      no-current
      (let ((node (path:node stck))
            (slot (path:slot stck)))
        (cons (node:key node slot) (node:pointer node slot)))))

(define (update! tree rcid)
  (define stck (path tree))
  (if (path:empty? stck)
      no-current
      (let ((node (path:node stck))
            (slot (path:slot stck)))
        (node:pointer! node slot rcid)
        (node:write! node)
        done)))
```

gewoon lezen

schrijven + write-back

Herinnering (slide 19)

Indien <u>toevoeging</u> in een blad een overflow oplevert, splitsen we de node in 2 en wordt de ‘scheidende’ sleutel naar boven toe gepropageerd en daar (recursief) toegevoegd.

Indien <u>verwijdering</u> uit een blad minder dan N/2 sleutels bevat, proberen we (a) links of rechts te “lenen” of (b) met links of rechts te mergen indien lenen onmogelijk is.

Toevoegen

```
(define (insert! tree key rcid)
  (define ntyp (node-type tree))
  (define ktyp (ntype:key-type ntyp))
  (define sent (ntype:key-sent ntyp))
  (define stck (path tree))
  (define root (tree-root tree))
  (define (build-path bptr)
    (define node (node:read ntyp bptr))
    (define slot (node:locate-leftmost node key))
    (path:push! stck node (node:complement slot))
    (if (not (node:leaf? node))
        (build-path (node:pointer node (node:complement slot)))))
  (define (traverse-path key pointer leaf?)
    ...)
  (path:clear! stck)
  (if (not (fs:null-block? root))
    (build-path root))
  (traverse-path key rcid #t)
  (path:clear! stck)
  done)
```

Zoek waar het zou moeten zitten

Toevoegen (vervolg)

```
(define (traverse-path key pointer leaf?)
  (if (path:empty? stck)
    (let
      ((new-root (node:new ntyp leaf?)))
     (node:key-pointer! new-root 1 key pointer)
     (node:pointer! new-root 0 root)
     (node:write! new-root)
     (tree-root! tree (node:position new-root)))
    (let*
      ((node (path:node stck))
       (slot (path:slot stck))
       (boundary (node:locate-leftmost node sent)))
     (path:pop! stck)
     (cond
       ((negative? boundary)
        (node:key-pointer-insert! node (+ slot 1) key pointer)
        (node:write! node))
       (else
        (let*
            ((new-node (node:new ntyp leaf?))
             (prop-key
              (node:key-pointer-insert-split!
               node new-node (+ slot 1) key pointer leaf?)))
          (node:write! node)
          (node:write! new-node)
          (traverse-path prop-key (node:position new-node) #f)))))))
```

De tree groeit

Zoek +∞

Gevonden? Er is nog plaats!

Anders splitsen en recursief inserten

Verwijdering

```
(define (delete! tree)
  (define ntyp (node-type tree))
  (define stck (path tree))
  (define (traverse-path node)
    ...)
  (define (do-delete-key-pointer! node slot)
    (node:key-pointer-delete! node slot)
    (if (< (node:size node) (div (node:capacity node) 2))
      (traverse-path node)
      (node:write! node)))
  (if (path:empty? stck)
      no-current
      (let ((node (path:node stck))
            (slot (path:slot stck)))
        (path:pop! stck)
        (do-delete-key-pointer! node slot)
        (path:clear! stck)
        done)))
```

verwijder het slot

indien underflow opkuisen

Verwijdering (vervolg)

```
(define (traverse-path node)
  (if (path:empty? stck)
      (cond ((= (node:size node) 0)
             (tree-root! tree (node:pointer node 0))
             (node:delete! node))
            (else
             (node:write! node)))
      (let* ((prnt-node (path:node stck))
             (prnt-slot (path:slot stck))
             (left-sibl (and (< 0 prnt-slot)
                             (node:read ntyp
                                        (node:pointer prnt-node (- prnt-slot 1)))))
             (push-lkey (and left-sibl
                             (node:borrow-from-left?
                              left-sibl node (node:key prnt-node prnt-slot))))
             (rght-sibl (and (not push-lkey)
                             (< prnt-slot (node:size prnt-node))
                             (node:read ntyp
                                        (node:pointer prnt-node (+ prnt-slot 1)))))
             (push-rkey (and rght-sibl
                             (node:borrow-from-right?
                              node rght-sibl (node:key prnt-node (+ prnt-slot 1))))))
        (path:pop! stck)
        ...)
```

Heeft node linker-sibling?

Heeft node rechter-sibling?

Lenen?

Verwijdering (vervolg)

```
     ...
(path:pop! stck)
(cond ((or push-lkey push-rkey)
       (cond (push-lkey
              (node:write! left-sibl)
              (node:key! prnt-node prnt-slot push-lkey))
             (else
              (node:write! rght-sibl)
              (node:key! prnt-node (+ prnt-slot 1) push-rkey)))
       (node:write! node)
       (node:write! prnt-node))
      (else
       (cond (left-sibl
              (node:merge!
               left-sibl node (node:key prnt-node prnt-slot))
              (node:write! left-sibl)
              (node:delete! node)
              (do-delete-key-pointer! prnt-node prnt-slot))
             (rght-sibl
              (node:merge!
               node rght-sibl (node:key prnt-node (+ prnt-slot 1)))
              (node:write! node)
              (node:delete! rght-sibl)
              (do-delete-key-pointer! prnt-node (+ prnt-slot 1))))))))
```

lenen gelukt => afwerken

anders mergen

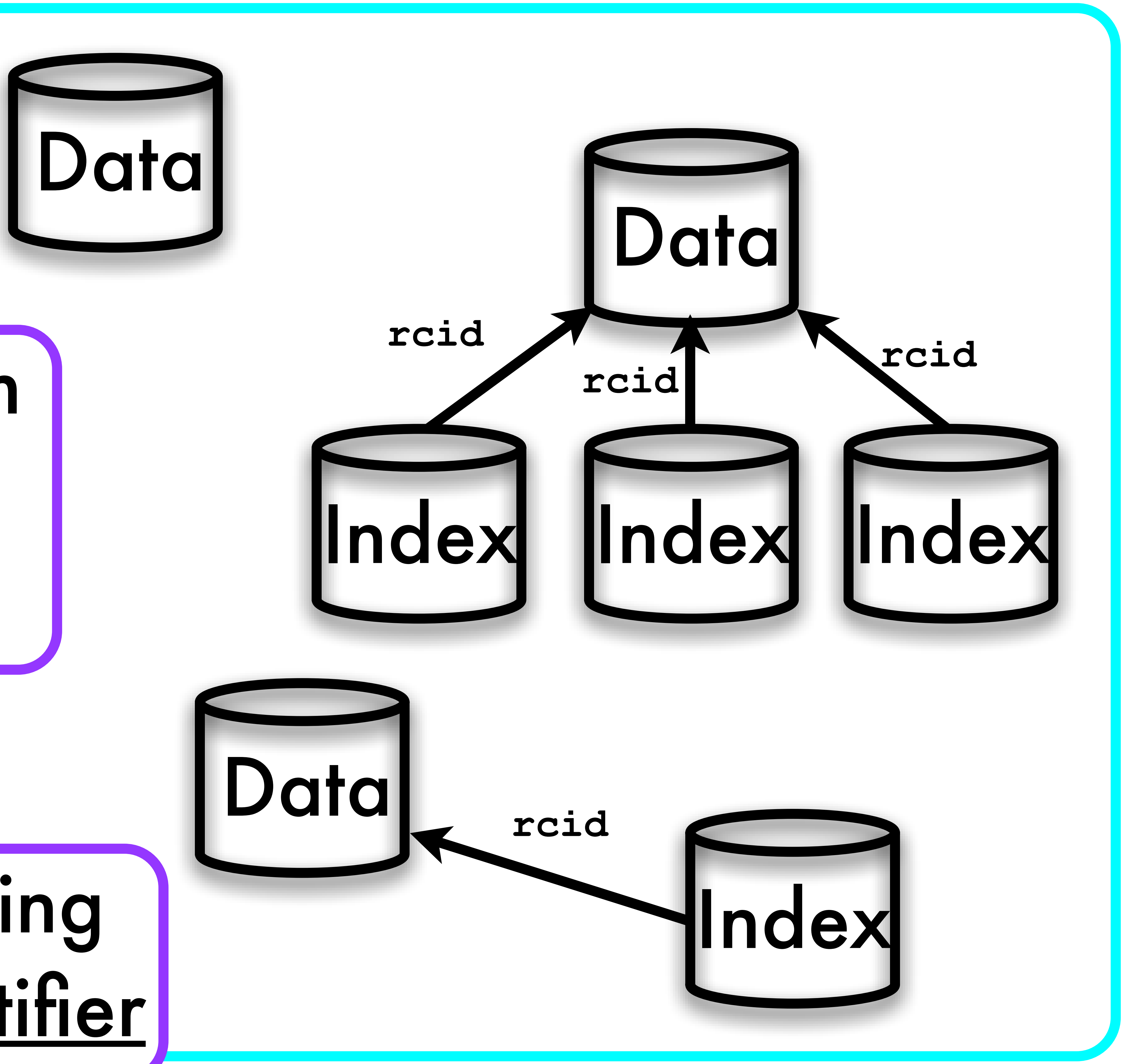
links of rechts + recursief deleten

Databank = { Files }

<u>Datafiles</u> bevatten de eigenlijke tupels

<u>Indexfiles</u> dienen om tupels uit datafiles snel terug te vinden

Tupelverwijzing = <u>record identifier</u>



Het Database ADT

Enkel de hoofdlijnen

Definitie

```
ADT database

new
   ( disk string → database )
delete!
   ( database → ∅ )
create-table
   ( database string pair → table )
create-index!
   ( database table string number → ∅ )
drop-table!
   ( database table → ∅ )
insert-into-table!
   ( database table pair → ∅ )
delete-where!
   ( database table number any → ∅ )
select-from/eq
   ( database table number any → pair )
```

We bespreken de creates, insert en select

Het Tabel ADT (Herinnering)

```
ADT table

new
  ( disk string pair → table )
open
  ( disk string → table )
close!
  ( table → ∅ )
table?
  ( any → boolean )
drop!
  ( table → ∅ )
schema
  ( table → schema )
disk
  ( table → disk )
set-current-to-first!
  ( table → status )
set-current-to-next!
  ( table → status )
current
  ( table → rcid U {no-current})
current!
  ( table rcid → ∅ )
```

```
status = { done,
           no-current,
           next-higher,
           duplicate,
           not-found }
```

```
insert!
  ( table pair → rcid )
delete!
  ( table rcid → ∅ )
peek
  ( table → pair )
```

Het B+-Tree ADT (Herinnering)

```
ADT b-tree

new
  ( disk string byte byte → b-tree )
open
  ( disk string → b-tree )
b-tree?
  ( any → boolean )
drop!
  ( b-tree → ∅ )
flush!
  ( b-tree → ∅ )
```

```
status = { done,
           no-current,
           next-higher,
           not-found }
```

```
insert!
  ( b-tree any rcid → status )
set-current-to-first!
  ( b-tree → status )
find!
  ( b-tree any → status )
set-current-to-next!
  ( b-tree → status )
delete!
  ( b-tree → status )
peek
  ( b-tree → any × rcid )
update!
  ( b-tree rcid → status )
```

Het Meta-Schema

De table "TBL" bevat voor elke tabelnaam een (gegenereerde) ID

```
(define meta-schema:table `((string  ,fs:filename-size)
                            (natural 2)))
(define table:table-name    0)
(define table:table-id      1)


(define meta-schema:indexes `((natural 2)                  ; table identity
                              (string  ,fs:filename-size) ; index name
                              (natural 2)))               ; attribute-number
(define indexes:tble-idty  0)
(define indexes:index-name 1)
(define indexes:key-att    2)
```

De table "IDX" bevat voor elke tabel-ID de namen van de index-files en het attribuutnummer van die indices

Creatie

```
(define-record-type database
  (make t i)
  database?
  (t tables)
  (i indexes))
```

Creëer beide meta-tabellen

```
(define (new disk name)
  (define tbls (tbl:new disk (string-append "TBL" name) meta-schema:table))
  (define idxs (tbl:new disk (string-append "IDX" name) meta-schema:indexes))
  (make tbls idxs))

(define (open disk name)
  (define tbls (tbl:open disk name))
  (define idxs (tbl:open disk name))
  (make tbls idxs))
```

Voorbeeldgebruik Tabelcurrent

```
(define (find-id-in-meta-table dbse tabl)
  (define name (tbl:name tabl))
  (define tbls (tables dbse))
  (tbl:set-current-to-first! tbls)
  (let loop
    ((tuple (tbl:peek tbls)))
    (let ((tble-name (car tuple))
          (tble-idty (cadr tuple)))
      (cond ((string=? tble-name name)
             tble-idty)
            ((not (eq? (tbl:set-current-to-next! tbls) no-current))
             (loop (tbl:peek tbls)))
            (else
             not-found)))))
```

Zoek de ID van een tabel op in TBL

```
(define (for-all-tuples table proc)
  (if (not (eq? (tbl:set-current-to-first! table) no-current))
      (let loop
        ((tuple (tbl:peek table)))
        (let ((curr (tbl:current table)))
          (if (and (proc tuple curr)
                   (not (eq? (tbl:set-current-to-next! table) no-current)))
              (loop (tbl:peek table)))))))
```

Doe proc met alle tupels in een tabel

Creëer Tabel + Creëer Index

```
(define (create-table dbse name scma)
  (define tbls (tables dbse))
  (define disk (tbl:disk tbls))
  (define tble (tbl:new disk name scma))
  (define idty (gennum))
  (tbl:insert! tbls (list name idty))
  tble)
```

Creëer de tabel en voeg hem aan de TBL-tabel toe

```
(define (create-index! dbse tabl name attribute)
  (define disk (tbl:disk tabl))
  (define tbls (tables dbse))
  (define idxs (indexes dbse))
  (define idty (find-id-in-meta-table dbse tabl))
  (define scma (tbl:schema tabl))
  (define indx (btree:new disk name
                          (scma:type scma attribute)
                          (scma:size scma attribute)))
  (tbl:insert! idxs (list idty name attribute))
  (for-all-tuples
   tabl
   (lambda (tuple rcid)
     (btree:insert! indx (list-ref tuple attribute) rcid)))
  (tbl:close! idxs)
  (btree:flush! indx))
```

Creëer de index en voeg hem aan de IDX-tabel toe

Indexeer reeds bestaande tupels

Meer Hulpstukken

Apply proc op alle tabellen in TBL

```
(define (for-all-tables dbse proc)
  (define tbls (tables dbse))
  (define disk (tbl:disk tbls))
  (when (not (eq? (tbl:set-current-to-first! tbls) no-current))
    (le
     (
```

```
(define (for-all-indices dbse tble proc)
  (define idxs (indexes dbse))
  (define disk (tbl:disk idxs))
  (define idty (find-id-in-meta-table dbse tble))
  (when (not (eq? (tbl:set-current-to-first! idxs) no-current))
    (let all-index-tuples
      ((tuple (tbl:peek idxs)))
      (if ((= (list-ref tuple indexes:tble-idty) idty) ; the index belongs to the tble-indx
          (let ((indx (btree:open disk (list-ref tuple indexes:index-name))))
            (if (and (proc indx (list-ref tuple indexes:key-att))
                     (not (eq? (tbl:set-current-to-next! idxs) no-current)))
                (all-index-tuples (tbl:peek idxs)))))
          (if (not (eq? (tbl:set-current-to-next! idxs) no-current))
              (all-index-tuples (tbl:peek idxs))
```

Manuele Implementatie m.b.v. tabel ADT

Apply proc op alle indices in IDX van een tabel

Tupel Toevoegen

```
(define (insert-into-table! dbse tble tuple)
  (define rcid  (tbl:insert! tble tuple))
  (tbl:close! tble)
  (for-all-indices dbse tble
                 (lambda (indx att)
                   (btree:insert! indx (list-ref tuple att) rcid)
                   (btree:flush! indx))))
```

B+-tree

kolomnummer

Tupel aan de tabel toevoegen

En aan alle indices

Equality-Queries

kolomnummer

```
(define (select-from/eq dbse tble attr valu)
  (define scma (tbl:schema tble))
  (define type (scma:type scma attr))
  (define eqls (vector-ref equals type))
  (define indx ())
  (define rslt ())
  (for-all-indices dbse tble (lambda (idx att)
                               (when (= att attr)
                                 (set! indx idx)
                                 #f)))
  (if (null? indx)
    (for-all-tuples tble (lambda (tple rcid)
                          (if (eqls (list-ref tple attr) valu)
                            (set! rslt (cons (tbl:peek tble) rslt)))))
    (for-all-identical-keys indx eqls valu
                           (lambda (rcid)
                             (tbl:current! tble rcid)
                             (set! rslt (cons (tbl:peek tble) rslt)))))
  rslt)
```

Zie eerst of er een index bestaat op het gebruikte attribuutnummer

Zonee: loop hele tabel af

```
(define (for-all-identical-keys indx eqls valu proc)
 (let loop
   ((cur? (eq? (btree:find! indx valu) done)))
   (if cur?
     (loop (and (proc (cdr (btree:peek indx)))
                (eq? (btree:set-current-to-next! indx) done)
                (eqls (car (btree:peek indx)) valu))))))
```

Zoja: loop in de index

Hoofdstuk 17