KU LEUVEN

Inleiding tot databanken

5. Opslag & Indexering

Prof. dr. Paolo Pilozzi

Overzicht

Hoe databanken hun data opslagen, zoeken en aanpassen.
Hoofdstuk 13 & 14

5.1 Data Opslag
5.2 Indexering

Overzicht

Hoe databanken data opslagen.
Hoofdstuk 13

5.1 Data opslag

- Inleiding
- Opslag
- Organisatie
- Hashing

Inleiding

Doel: Performantie (= snelheid & ruimte optimaliseren)

1. Werkgeheugen of primaire opslag: RAM & Cache
   - CPU toegang, Volatiel, heel snel, beperkt in ruimte, duur
2. Secundaire opslag (Eerst naar werkgeheugen voor CPU toegang)
   * SSD (persistent, snel, schrijf-operatie limieten)
   * HDD (persistent, redelijk snel, goedkoper)
   * Tape (persistent, traag, archivering)
3. Toegang via interfaces: FSB (moederbord), USB, SATA, ..., NVMe
4. Bestandsstructuren persistent geheugen: FAT, NTFS, ..., EXT2/3/4
5. Blokgebaseerde data-transfer persistent -en werkgeheugen (DMA)

=> Technologie/configuratie is toepassingsafhankelijk: snelheid en ruimte.

Opslag

Verschillende technieken om blokken van een bestand (file system, OS afhankelijk) te alloceren op een schijf:

* Aaneensluitend: bestandsblokken opeenvolgend
  - Problematisch om bestand uit te breiden
  - Snel om hele bestand te lezen
* Gelinkt: bestandsblok met pointer naar volgend blok
  - Makkelijk om bestand uit te breiden
  - Traag om hele bestand te lezen
* Clusters: Combinatie aaneensluitend en gelinkt
  = opeenvolgende blokken in gelinkte clusters

Opslag

Een databank wordt opgeslagen in de vorm van records.

Records (= rijen in tabel) beschrijven
* entiteiten en hun attributen (waardes)
* als bytestrings via standaard datatypes
* en zijn van vaste of variabele lengte.

Grote hoeveelheden ongestructureerde data (e.g., BLOB) wordt apart opgeslagen met pointer in record naar de data.

Opslag

Bestand = sequentie van records.

Meestal bestaat bestand uit records van hetzelfde type:

* Vaste-lengte records of Variabele-lengte records
* Bij Variabele-lengte: "separator" of met lengte indicatie

Records in een bestand gealloceerd in disk/schijf blokken:

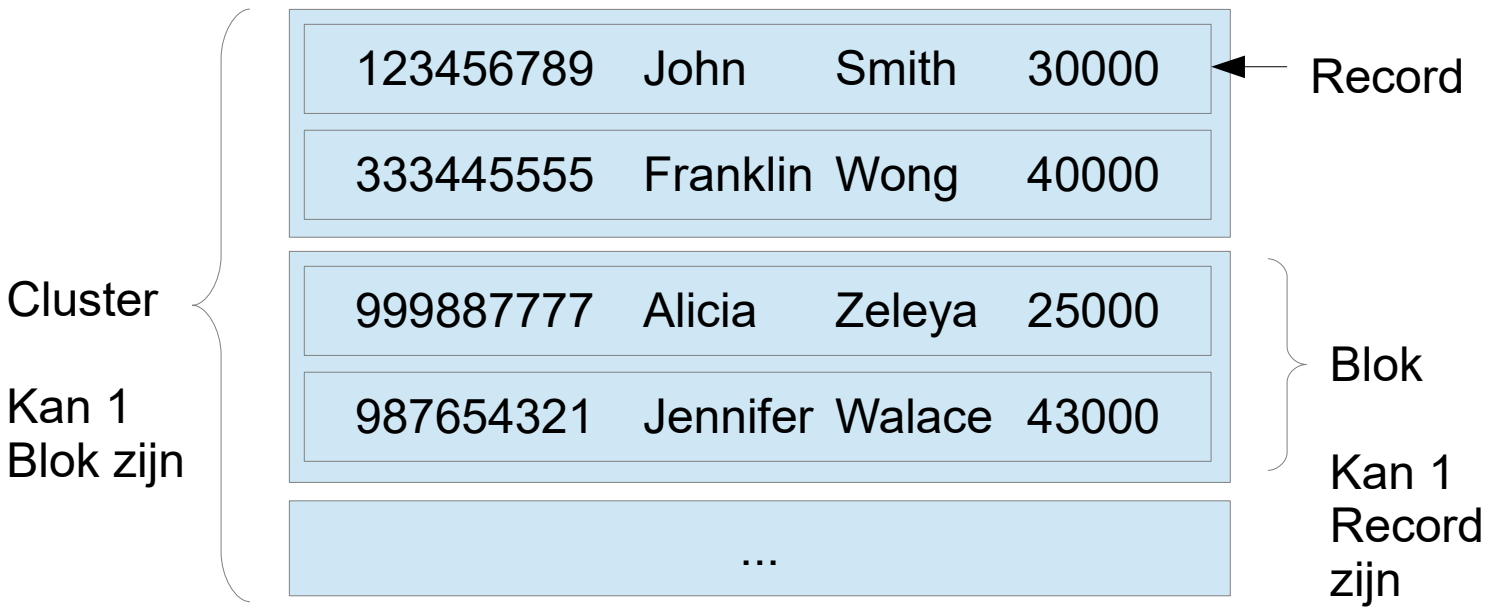
= Transfereenheid tussen schijf en werkgeheugen

* "Spanned" records (= over blokken heen met pointer)
* "Unspanned" records (= alle records volledig in blok)

Opslag

De bestandsheader (hoofding): informatie over bestand

* Bestandsstructuur: Schijfadressen (clustered?)
* Recordstructuur:
  - Vaste-lengte?
  - Spanned?
  - Veldtypes
  - Recordtypes
  - Separators



Opslag

Merk op!

1. Je leest/schrijft nooit 1 record, steeds een heel blok!
   * Met records werken => van/naar werkgeheugen

2. Snelheid wordt geoptimaliseerd als:
   a. Zo weinig mogelijk records opgezocht moeten worden
   b. De opgezochte records zo klein mogelijk zijn
   c. Records binnen blokken (geordend) naast elkaar zitten

Bestandsorganisatie

Belangrijke operaties op bestanden:

* Opvraging = Zoeken in records (op attributen)
* Wijziging = Toevoegen, verwijderen en aanpassen
  - Wijzigingen kunnen pas na opvraging

Types bestanden:

* Ongeordend (Heap files)
* Geordend (Sorted files) op een attribuut
* Hashed

Hashing

Hash bestand organiseert snelle toegang tot records.

* Hash functie: beeldt record af op geheugenadres
  - Op basis van Hash veld (als sleutel: hash key)

Kan intern (volatiel) of extern (persistent) gebruikt worden.

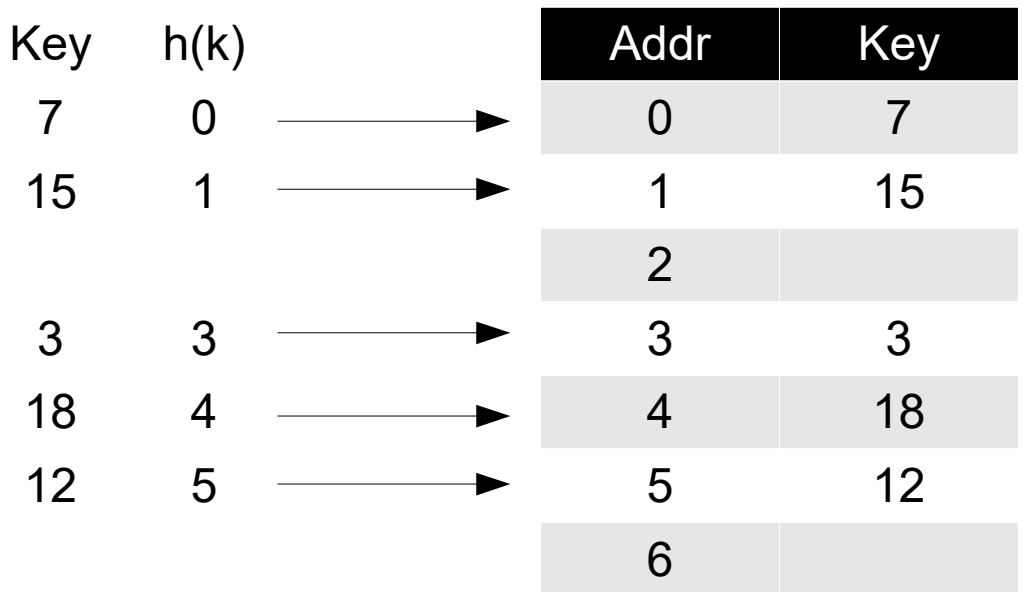
* Intern in werkgeheugen: hash levert geheugenadres
  => Zoeken in constante tijd!
* Extern op schijf: hash levert blokadres
  => Doel: Meeste records slechts 1 blok opvragen

Hashing – Intern

Voorbeeld hashfuncties:

* Hash key: Integer als record identifier
  - h(k) = k mod 7
* Hash key: String
  - Product ASCII waarden
  - h(k) = k mod 7

Opgelet voor Collisions (Botsing)
als h(k1) = h(k2) met k1 ≠ k2
Vb. Record 7 en 14



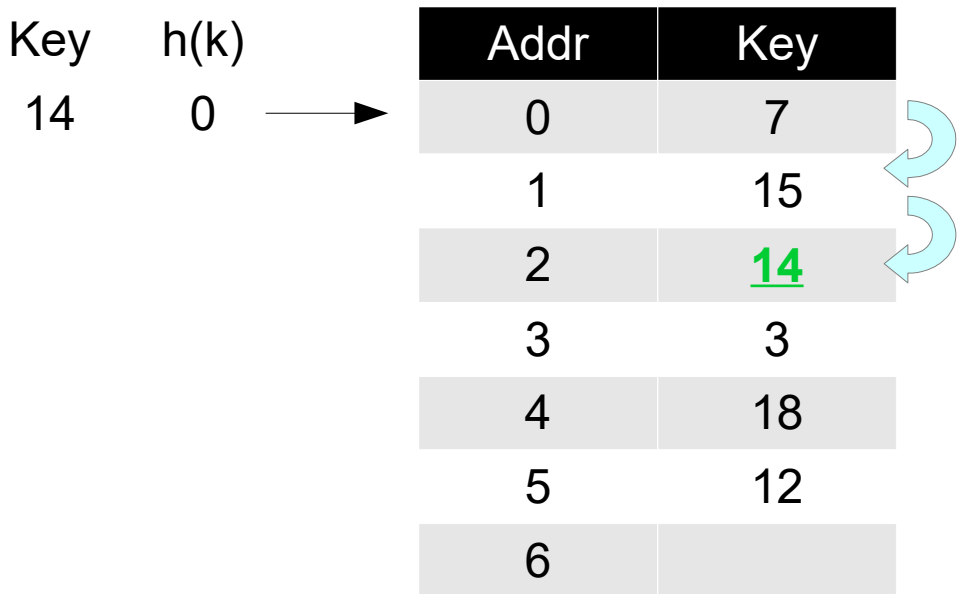
Hashing – Intern

Collision: Meerdere records verwijzen naar zelfde adres
    => komen terecht in zelfde cel

Meestal: sleutelruimte >> adresruimte
    => Botsingen zijn onvermijdelijk
=> Collision resolution:
    * Meerdere records in 1 cel gegeven celdiepte
    * Indien te veel records: overloop (overflow):
        - Open of gesloten (ketening) adressering
        - Multi-hashing (+ open adressering)

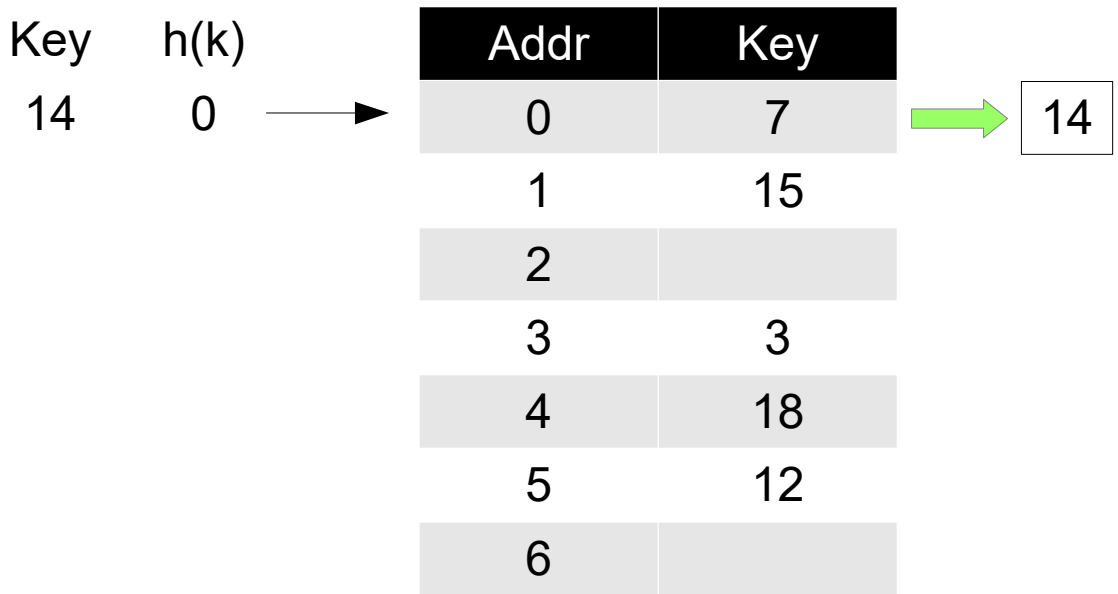
Hashing – Intern

Open adressering



Eerstvolgende vrije plaats

Gesloten adressering



Via pointer als reeds bezet

Hashing – Extern

= Hashing voor bestanden op schijf

Addresruimte: Buckets (bevatten meerdere records)
Een bucket is een tussenstap:

* Bucket is 1 blok of cluster van opeenvolgende blokken
* Hashfunctie beeldt record af op bucket nummer.
* Bestandsheader beschrijft bucketnummer-blokadres

Collisions minder problematisch want buckets zijn groot:

* Kan toch niet vermeden worden => overflows.
* Overflowbehandeling (bij interesse, zie boek).

Hashing

Een goede hashfunctie sprijdt zo goed als mogelijk de verwachtte records (=> verwachtte hashwaarden) over een gegeven adresruimte.

De gegeven adresruimte is typisch veel kleiner dan de hashwaarderuimte.

Hashing dient om snel toegang te krijgen tot specifieke records: Het laat zoeken in constante tijd toe!

Bestandsorganisatie

Structuur	Zoekstrategie	Complexiteit
Heap file *(Ongeordend)*	Lineair	O(n)
Sorted file *(Geordend)*	Lineair Binair	O(n) $O(\log_2(n))$
Hash file *(Hashfunctie)*	Lineair Direct	O(n) O(1)

Overzicht

Hoe databanken data opslagen.
Hoofdstuk 14

5.2 Indexering

- Inleiding
- Soorten indexen
- Hash-indexen
- Multiniveau-indexen
- B+ bomen

Wat is een index?

Een gegevensstructuur die de toegang op een bestand via een bepaald veld (of groep velden) efficiënter maakt.

=> Voor efficiënt te zoeken naar waardes van velden.

* Cfr. De woordenlijst achteraan een handboek.

Een index kan bijgehouden worden (als een tabel):

* In werkgeheugen (volatiel), maar relatief klein.
* Op schijf (persistent).

Wat indexeren?

Applicatieafhankelijk! => Wat zal er gezocht moeten worden

Bv. EMPLOYEE tabel:

* Op Ssn nummer zoeken? => Index voor Ssn nummers
  => Ook nuttig bij het maken van organigrammen
* Op naam zoeken? => Index voor -en achternaam
* Zoeken op:
  - departement?
  - geb.datum?
  - etc.

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

Soorten indexen

* Op veld dat de **ordening** van het bestand **bepaalt**
    * Primaire index: records zijn **uniek** geïdentificeerd
      => op sleutelveld
    * Cluster index: records **niet uniek** geïdentificeerd
      => op clusterveld
* Op veld dat de **ordening** van het bestand **niet bepaalt**
    * Secundaire index: zowel uniek als niet uniek
      => op sleutel -of clusterveld
      => samen zoekveld genoemd

Soorten – Primaire index

* Gegeven een bestand met vaste-lengte records
  - Geordend naar sleutelwaarde

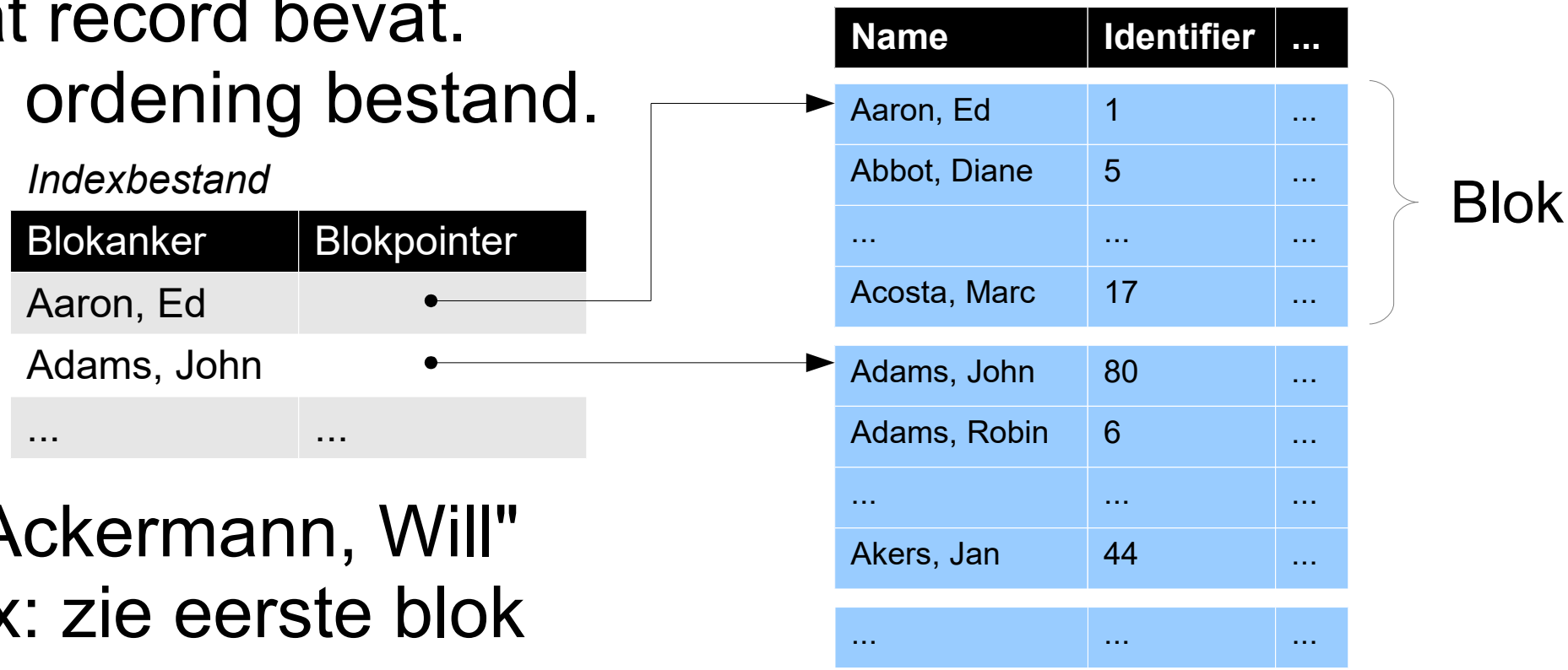
* Dan is een primaire index voor dat bestand er één met
  - Een (index)record per blok in het bestand
  - met sleutelwaarde van ankerrecord
    = eerste of laatste record in blok
  - en blokadres.

=> Via sleutelwaarde zoeken naar blokadres van record

Soorten – Primaire index

=> Gegeven sleutelwaarde zoeken in index naar blokadres van blok dat record bevat.

* Dankzij ordening bestand.



* Zoek "Ackermann, Will"
  - Index: zie eerste blok

Soorten – Primaire index

Een primaire index bevat (meestal)

* Kleinere records dan bestand
* Minder records dan bestand

= Niet-dichte/ijle (non-dense/sparse) index

=> Index is kleiner dan bestand

=> Doorlopen van index gaat sneller dan bestand

=> Minder toegang tot schijf nodig

Soorten – Clusterindex

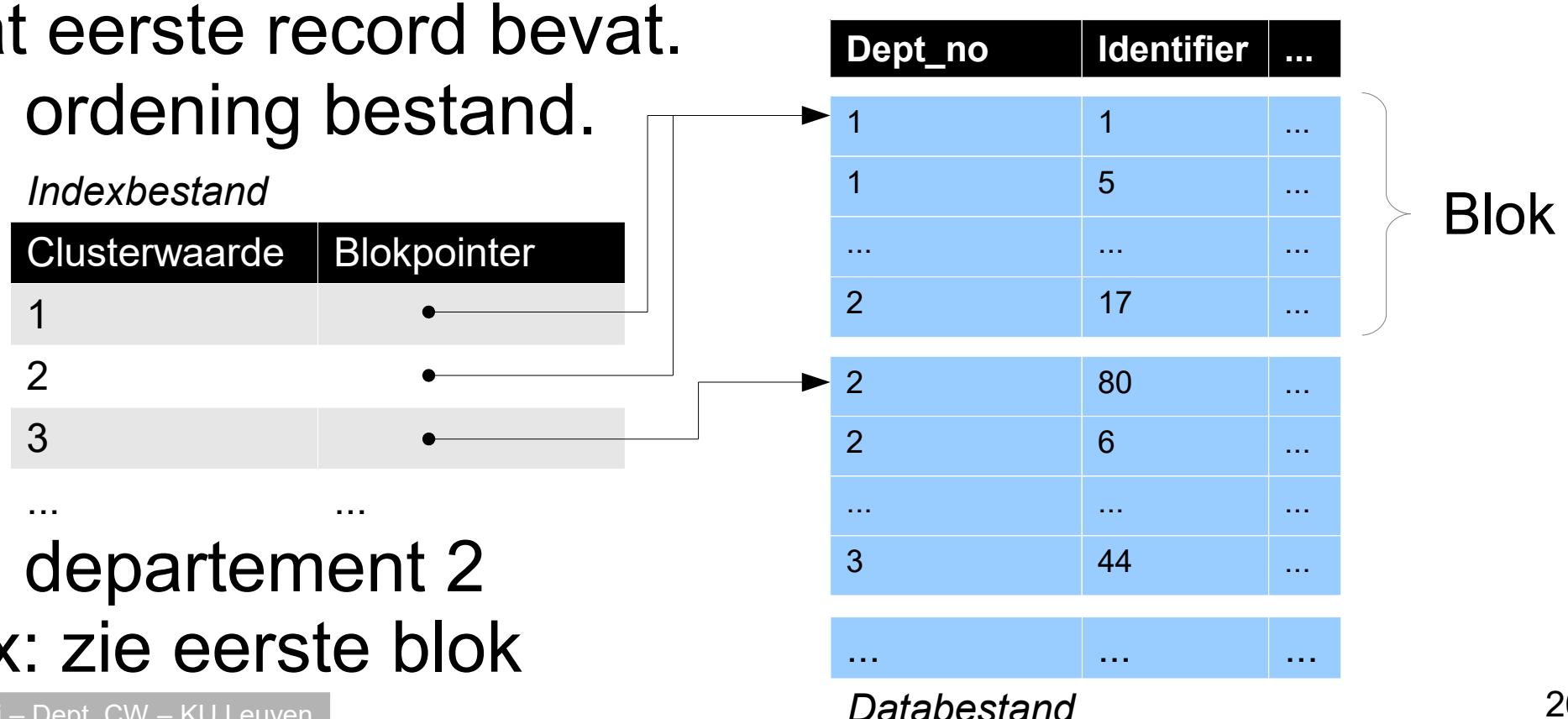
* Gegeven een bestand dat fysisch geordend is volgens een niet-uniek veld
    => Dit is geen sleutelveld maar een clusterveld

* Dan is een clusterindex voor dat bestand er één met
    - een (index)record per waarde clusterveld
    - met een veld dat de clusterwaarde bevat
    - en blokadres waar de clusterwaarde eerst voorkomt.
* Dankzij ordering ook hier een ijle index.

Soorten – Clusterindex

=> Gegeven clusterwaarde zoeken in index naar blokadres van blok dat eerste record bevat.

* Dankzij ordening bestand.



* Zoek in departement 2
  - Index: zie eerste blok

Soorten – Secundaire index

= Index op veld dat ordening bestandsrecords niet bepaalt
- Let op: Index is wel geordend volgens die veldwaarde!

* Secundaire index op sleutelveld:
  - Eén record in index per sleutelwaarde
  - Bevat sleutelwaarde + blok-/recordadres
  - Geen ordening in bestand => Geen ankerrecords

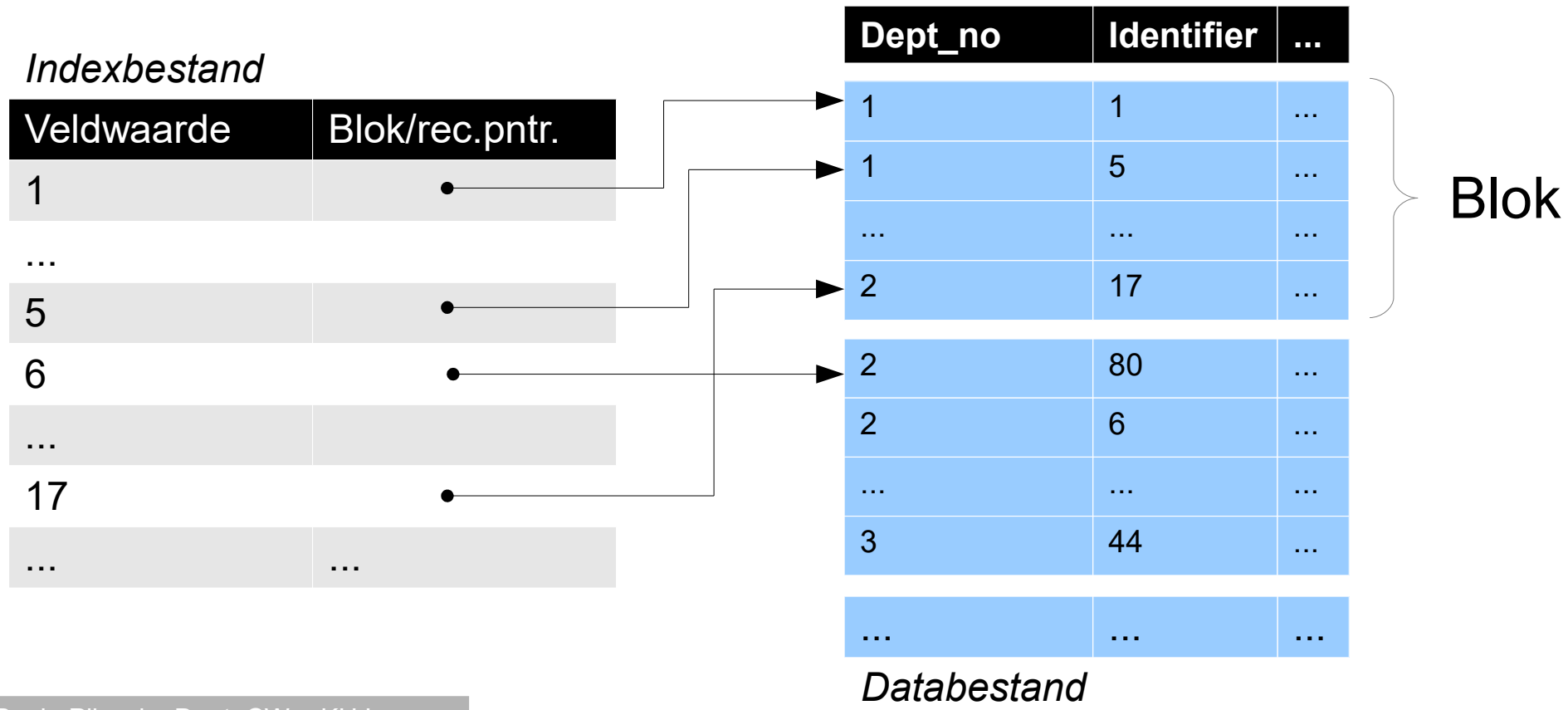
  => Dichte index: #indexrecords = #records in bestand

* Secundaire index op niet-sleutelveld (niet-uniek)
  - Variabele-lengte: meerdere blok-/recordadressen
  - Vaste-lengte: blokadres naar blok met daarin blok-/recordadressen

  => Mogelijks dichte index (Meestal ijl omwille van niet-uniek)

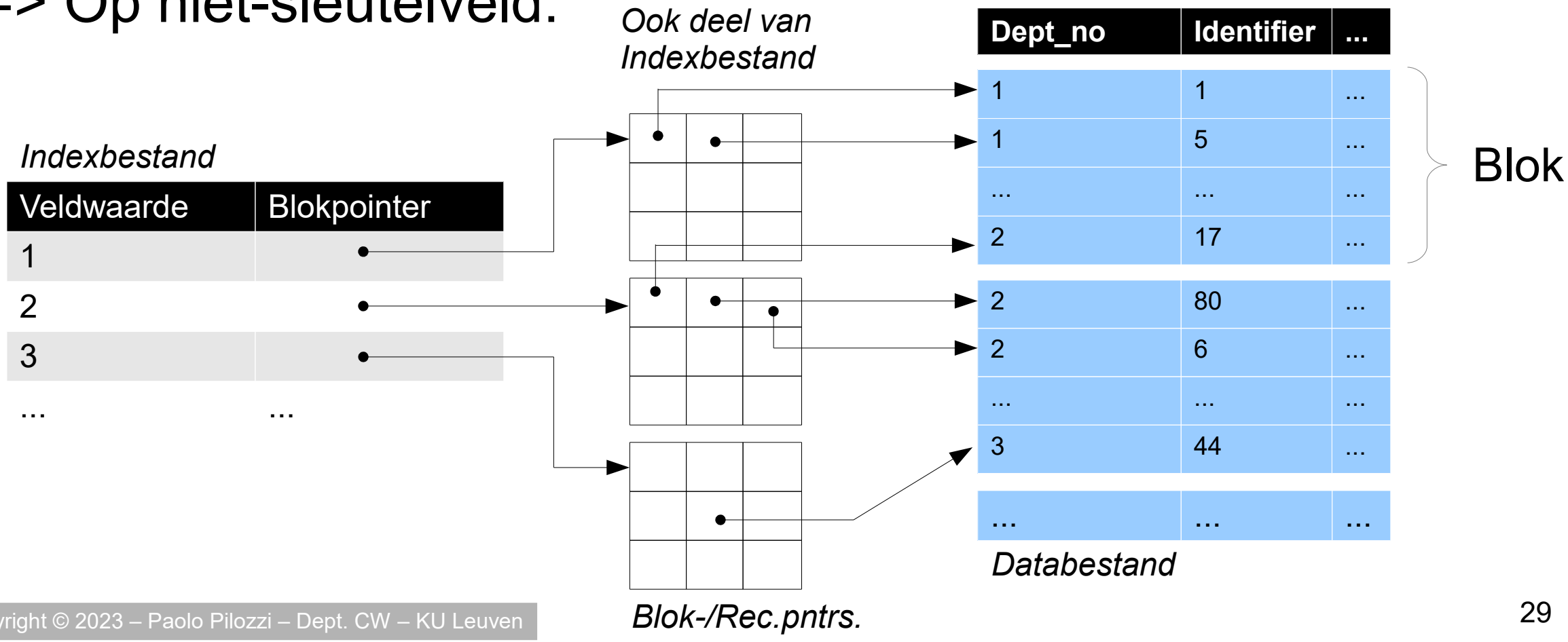
Soorten – Secundaire index

=> Op sleutelveld:



Soorten – Secundaire index

=> Op niet-sleutelveld:



Soorten – Secundaire index

Index kan zeer groot zijn!

=> Dichte index: #indexrecords = #records in bestand

Maar index is geordend!

=> Binair zoeken mogelijk in index.

=> In bestand kan enkel lineair gezocht worden.

Eens blok gevonden is, lineair zoeken in blok zelf:

* Blok is klein & zoeken in werkgeheugen, dus snel.

=> Verwaarloosbaar t.o.v. blok in werkgeheugen laden.

Hash-indexen

= Secundaire index, zonder indexordening

* In het algemeen op sleutel waarop niet geordend wordt
  - Hash functie bewaart de verbanden (orde) typisch niet
  - Als sleutel (=> uniek) dan triviaal minder collisions
* Hashfunctie: Beeldt sleutel op Bucket nummer af.
* Buckets bevatten Indexrecords die bestaan uit
  - veldwaarde en blokpointer (of recordpointer)

Hash-indexen

Zoeken in een hash-index:

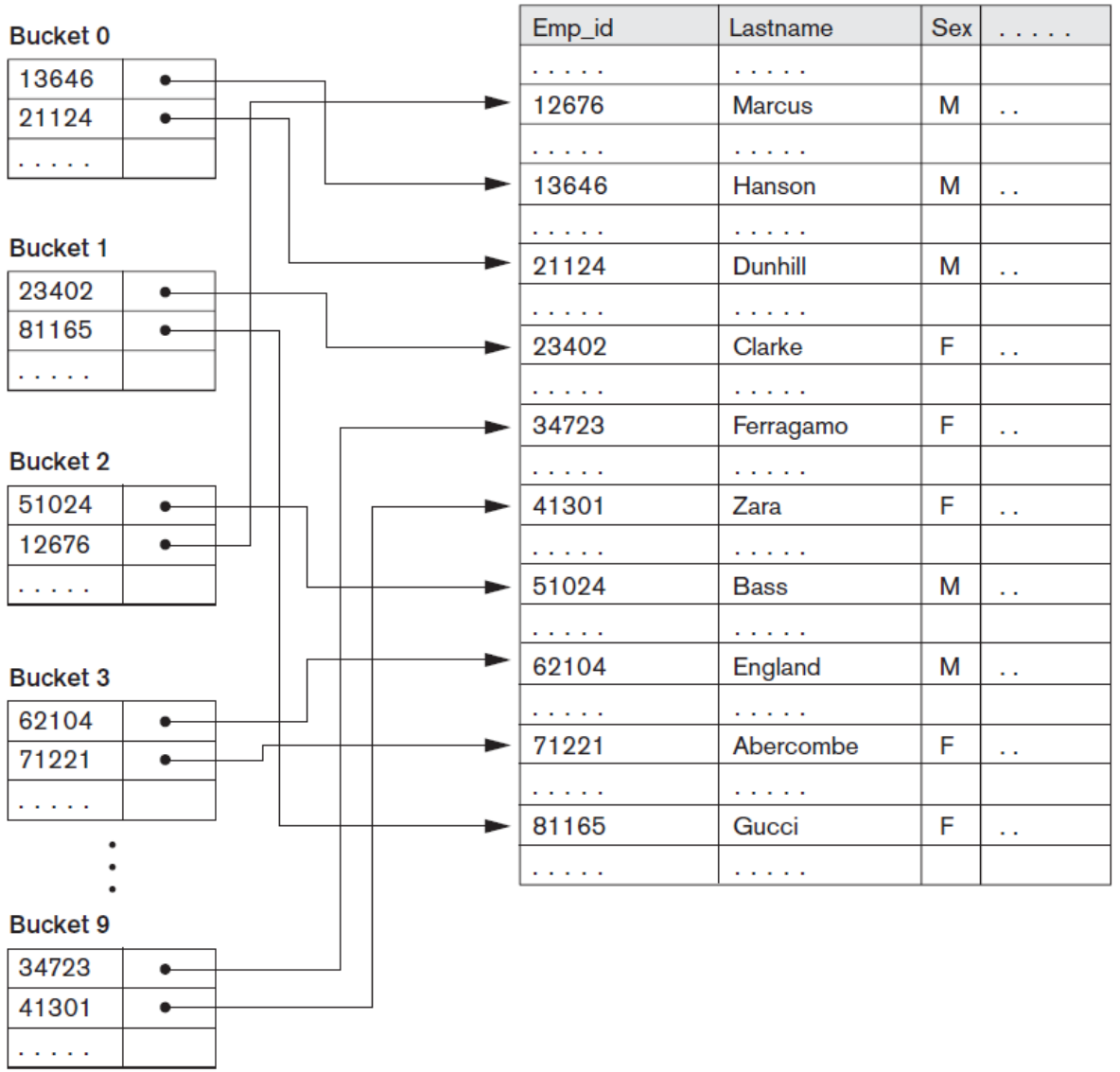
1. Bereken hash van sleutelwaarde
2. Bekom Bucketnummer
3. Zoek in Bucket naar sleutelwaarde
4. Bekom blokpointer of recordpointer
5. Met blokpointer, laadt blok in werkgeheugen
   - Zoek sleutelwaarde in blok en bekom recordpointer
6. Met recordpointer, vraag record op uit werkgeheugen

Hash-indexen

Voorbeeld Hashfunctie:
"Cijfersom mod 10"

Vb. h(81165)
= (8+1+1+6+5) mod 10
= 21 mod 10 = 1

Zoek 81165 in
Bucket h(81165) = 1
Zoek 62104 in
Bucket h(62104) = 3



Multiniveau-indexen

= Het indexeren van indexen.

Elke index van een index is een primaire index:

* Immers, een index is een geordende lijst van records bestaande uit unieke indexwaarden en hun pointers

=> Eerste niveau: primair/cluster/secundair

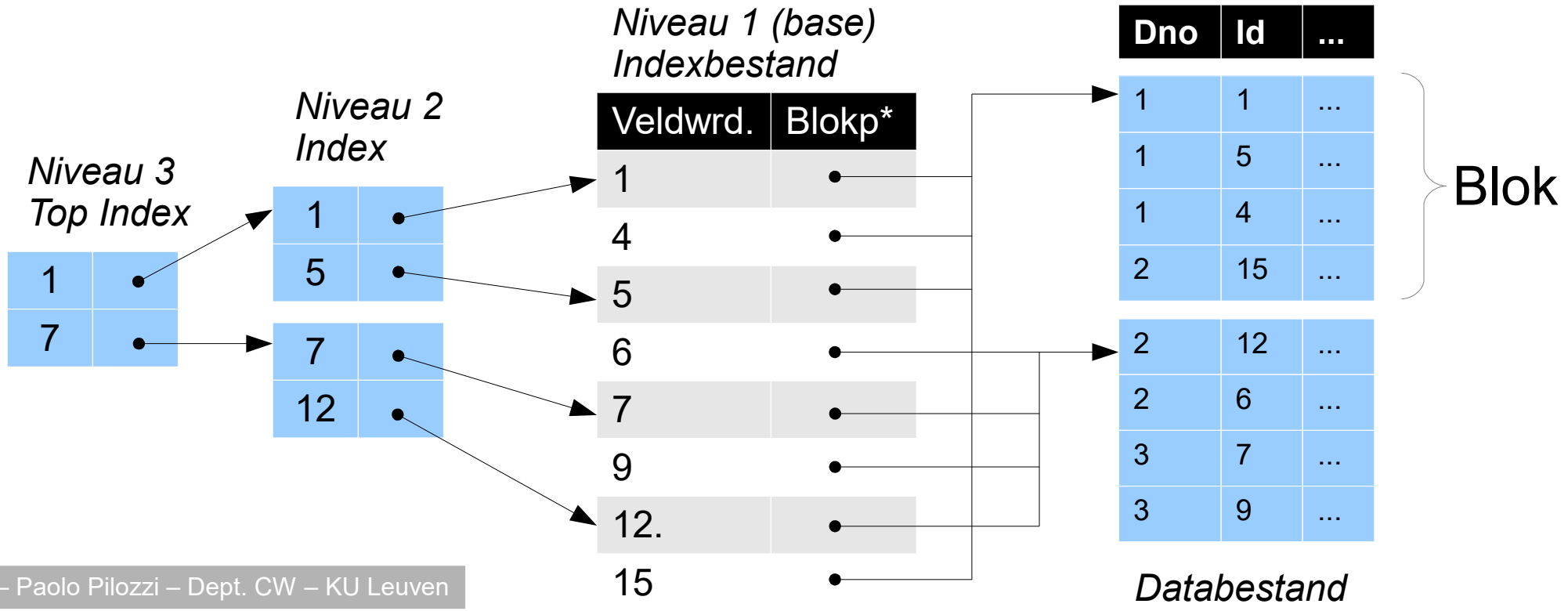
=> 2e tot n-de niveau per definite primair => ankers

=> Sneller waarden vinden in indexen zelf

Multiniveau fan-out => vast aantal vertakkingen per niveau = blokgrootte basisindex

Multiniveau-indexen

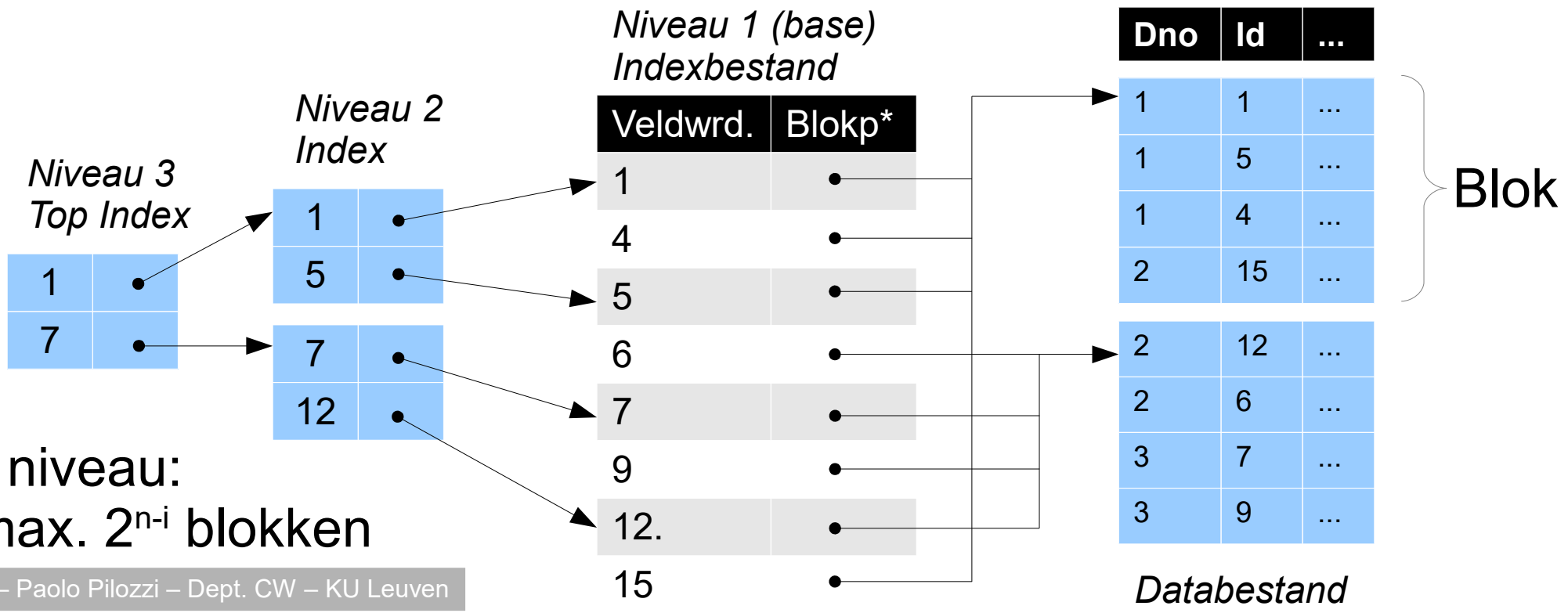
Vb. Blokgrootte basisindex: 2 (= blocking factor);
=> fo (fan-out) = 2 en basisindex bevat m = 4 blokken



Multiniveau-indexen

Zoeken? #niveaus, n = roundup($\log_{fo}(m)$)+1 => n= 3

=> roundup($\log_2(m)$)+1 blokken lezen



=> i-de niveau:
bevat max. 2^{n-i} blokken

Zoekcomplexiteit indexen

Zoeken waarde in mogelijks ongeordend uniek veld voor n blokken/records:

Geen index	O(n)	*Alles nakijken*
Lineaire index	$O(\log_2(n))$	*Geordend => binair zoeken*
Multiniveau	$O(\log_{fo}(n))$	*[fo = 2] ≈ binair zoeken*
Hash	O(1)	*Beste geval: constante tijd*

Statische vs. Dynamische Index

Indexen zijn gemaakt om te zoeken!
=> Trade-off: Toevoegen/Verwijderen is problematisch:

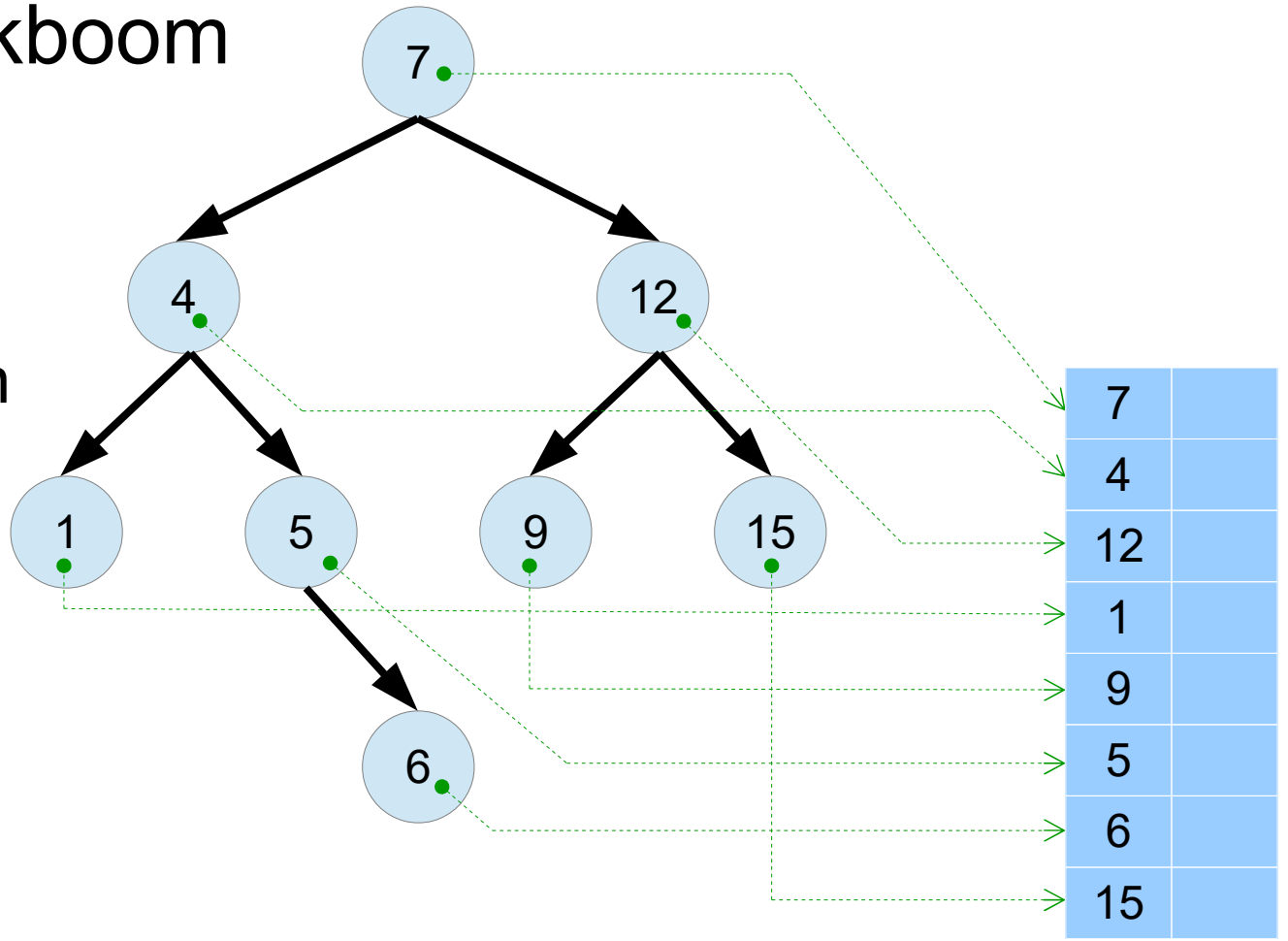
* Reorganisatie onvermijdelijk: indexen zijn geordend!
  - Hele boom indexen dienen aangepast te worden
* Batch/tussentijdse reorganisatie als mitigatie
  - Toevoegen altijd problematisch => laat plaats
  - Weglaten soms problematisch => maakt plaats

Oplossing: Dynamische indexstructuren zoals B+ bomen
* Merk op: Bestaat ook voor hashing (niet te kennen).

Indexeren met boomstructuren

Voorbeeld: Binaire zoekboom

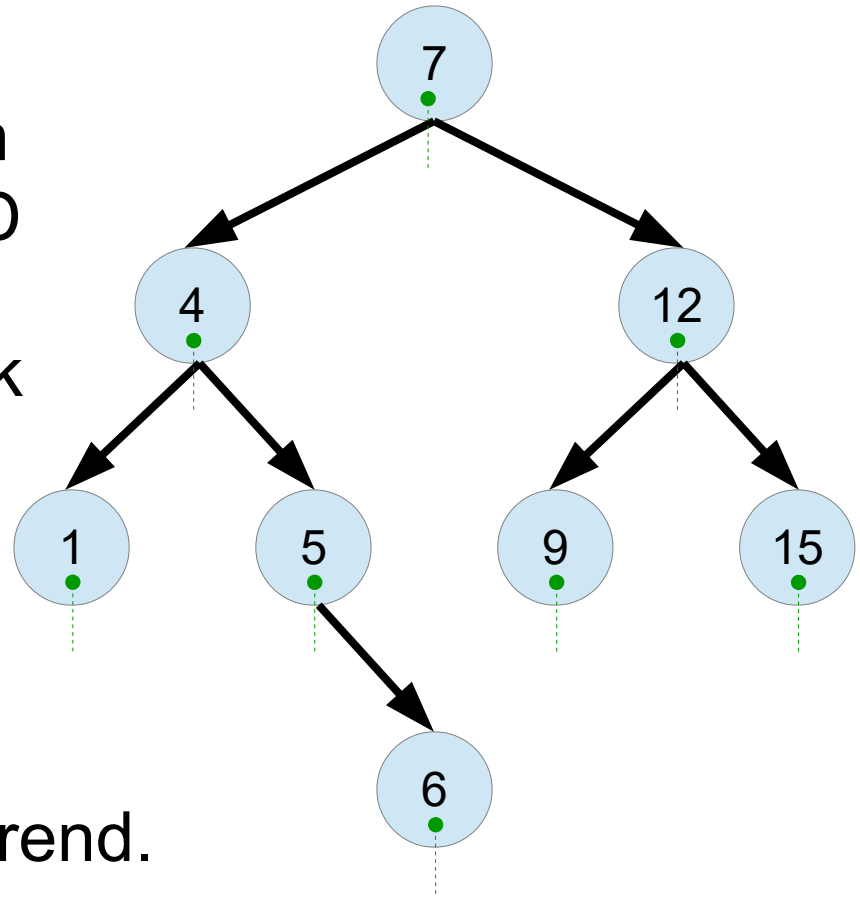
* 1 waarde per node
* Geordend:
  - => linkerdeelboom: strikt kleinere waarden
  - => rechterdeelboom: strikt grotere waarden
* Opzoeken evenredig met Diepte boom, D ≈ $\log_2(n)$
* Zoeken is meestal efficiënt



Indexeren met boomstructuren

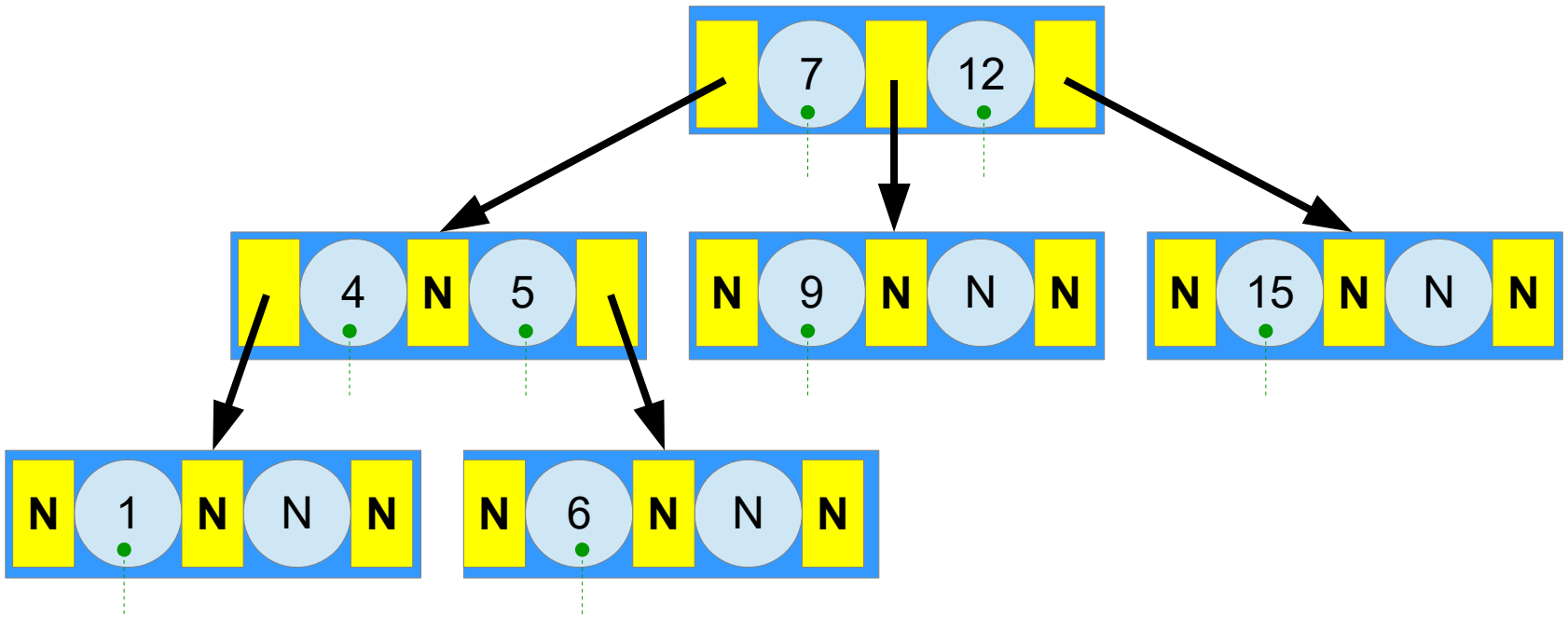
Voorbeeld: Binaire zoekboom

* Groeit en krimpt dynamisch met wijzigingen
  - Complexiteit wijzigingen evenredig met D
* Aanname evenwichtige boom!
  - Complexiteit O(n) ipv. O(log2(n)) mogelijk
  - Vb. Bij naïeve aanpak:
    => Voeg in stijgende volgorde toe:
    => 1,4,5,6,7,9,12,15
    => Diepte boom 8 ipv. 4
* Algoritmes voor evenwichtige zoekbomen:
  Implementeer met oog op evenwichtbewarend.



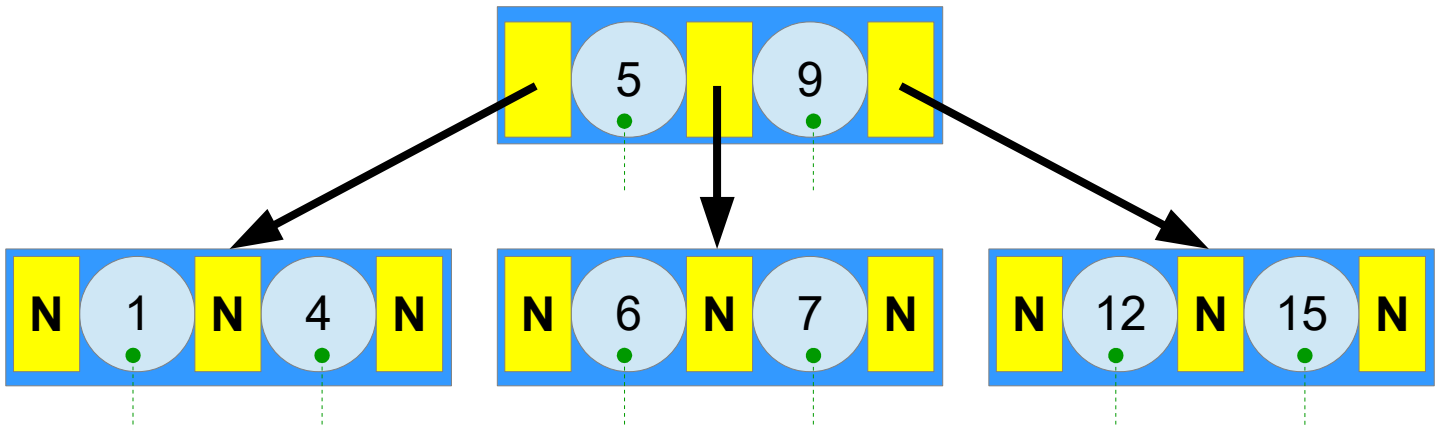
Indexeren met boomstructuren

Voorbeeld hogere-orde: Ternaire zoekboom
Naïef toevoegen: 7,4,12,1,9,5,6,15
N: NULL



Indexeren met boomstructuren

Voorbeeld hogere-orde: Ternaire zoekboom
Slim toevoegen: Best case



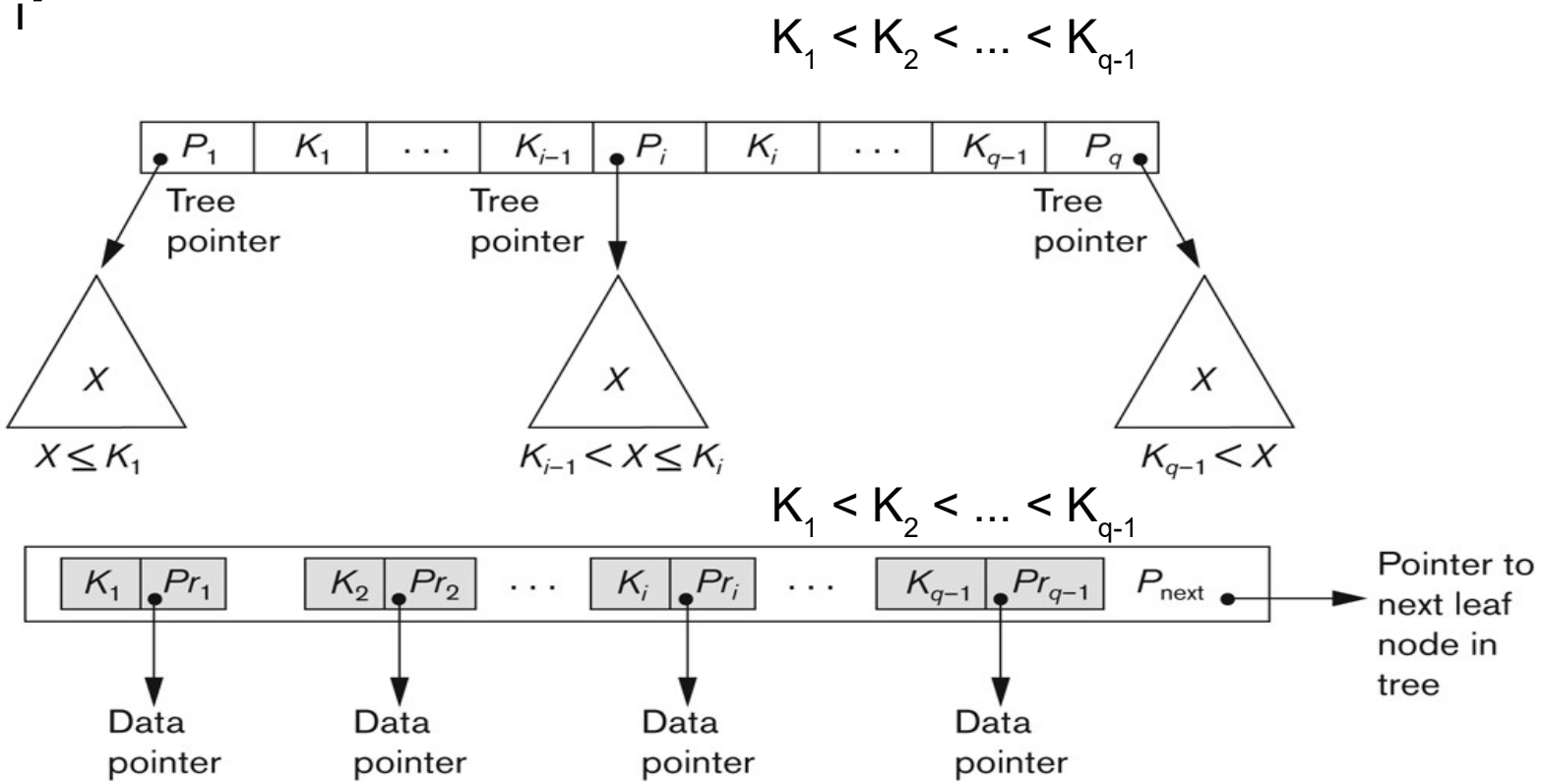
B+ bomen

Interne node, orde p_i:

q-1 zoekwaarden
q deelboompointers
$q \leq p_i \leq 2*q$

Bladnode, orde p_r:

q-1 zoekwaarden
q-1 recordpointers
1 next pointer
$q \leq p_r \leq 2*q$



Alle bladnodes op zelfde niveau!

Berekening orde B+ bomen

Voorbeeld met als gegeven:

* Grootte veld waarop gezocht, V = 9 bytes
* Grootte blok, B = 512 bytes
* Recordadres P_r = 7 bytes, blokadres P_b = 6 bytes

* Orde interne nodes, r_i (= #deelboompointers)

  => $r_i.P_b + (r_i-1).V \leq B$ <=> $r_i \leq (B+V)/(P_b+V)$

  <=> $r_i \leq (512+9)/(5+9) = 34.733$ => $r_i = 34$

* Orde bladnodes, r_b (= #recordpointers)

  => $r_b.(P_r+V)+P_b \leq B$ <=> $r_b \leq (B-P_b)/(P_r+V)$

  <=> $r_b \leq (512-6)/(7+9) = 31.625$ => $r_b = 31$

Berekening capaciteit B+ bomen

Voorbeeld vorige slide met Nodes voor 69% bezet
* Bezettingsgraad per niveau?

1. Gemiddeld #deelboompointers in interne nodes: r_i*0.69 = 23
2. Gemiddeld #recordpointers in bladnodes: r_b*0.69 = 21
3. Per niveau (voor N niveaus):

Niveau 0 (Root)	1 node	22 waardes	23 pointers
Niveau 1	23 nodes	506 waardes	529 pointers
Niveau 2	529 nodes	11638 waardes	12167 pointers
...			
Niveau N-1	r_i^{N-1} nodes	$(r_i-1)*r_i^{N-1}$ waardes	r_i^N pointers
Niveau N (Leaf)	r_i^N nodes	$r_b*r_i^N$ waardes & recordpntrs	r_i^N-1 pointers

Algoritme zoeken B+ bomen

K := zoekwaarde;
nBlok := nodeblok op blokadres rootnode;
Zolang nBlok geen bladnode is doe:
    q := #deelbomen in nBlok;
    v_0 := -∞; v_q := +∞;
    Voor i=1,2,...,q-1 doe v_i := zoekwaarde op de i-de plaats node nBlok;
    Kies i zodat $v_i < K \leq v_{i+1}$;
    nBlok := blok op blokadres b_i;
Zoek in bladnode nBlok een koppel (K,P_r);
Als gevonden, lees record op adres P_r anders meldt "niet gevonden";

Algoritme toevoegen B+ bomen

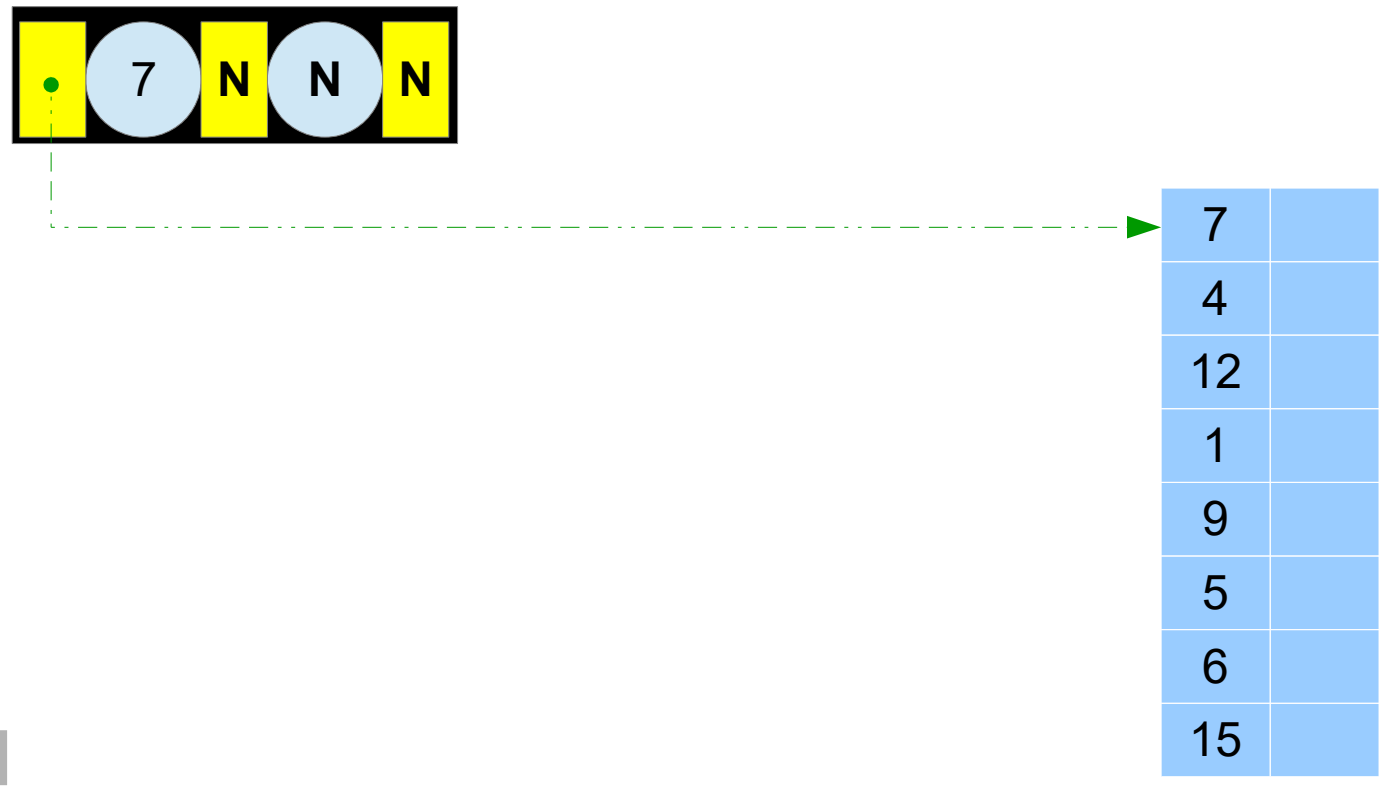
1. Zoek naar bladnode dat mogelijks K al bevat.
    1.1. Bevat de bladnode K reeds, dan kan Pr niet toegevoegd worden.
    1.2. Anders, als bladnode niet vol is: voeg (K,Pr) toe (zoals het hoort)
    1.3. Als bladnode n vol: maak nieuwe bladnode m
        * Pnext van m wordt die van n en Pnext van n wordt Pm
        * Voeg (K,Pr) toe (zoals hoort) en doe de lijst doormidden rond K'
        * Eerste helft (en K') blijft in n, tweede helft gaat naar m
        1.3.1. Ga naar parent van n
        1.3.2. Als parent onbestaande, nieuwe root met <Pn,K',Pm>
        1.3.3. Als parent bestaat en niet vol dan
                <...,Pn,Kn,...> wordt <...,Pn,K',Pm,Kn,...>
        1.3.4. Als parent vol zie volgende slide

Algoritme toevoegen B+ bomen

1.3.4. Als parent x vol en dus overvol gegeven <...,Pn,K',Pm,Kn,...>
Maak nieuwe interne node y
Splits <...,Pn,K',Pm,Kn,...> middendoor rond K"
Alles strikt links van K" blijft in x, rest wordt verwijderd
Alles strikt rechts van K" gaat naar y
Ga naar parent van x
* Als parent onbestaande, nieuwe root met <Px,K",Py>
* Als niet vol: <...,Px,...> wordt <...,Px,K",Py,...>
* Als parent vol dan herhaal 1.3.4 met <...,Px,K",Py,...>

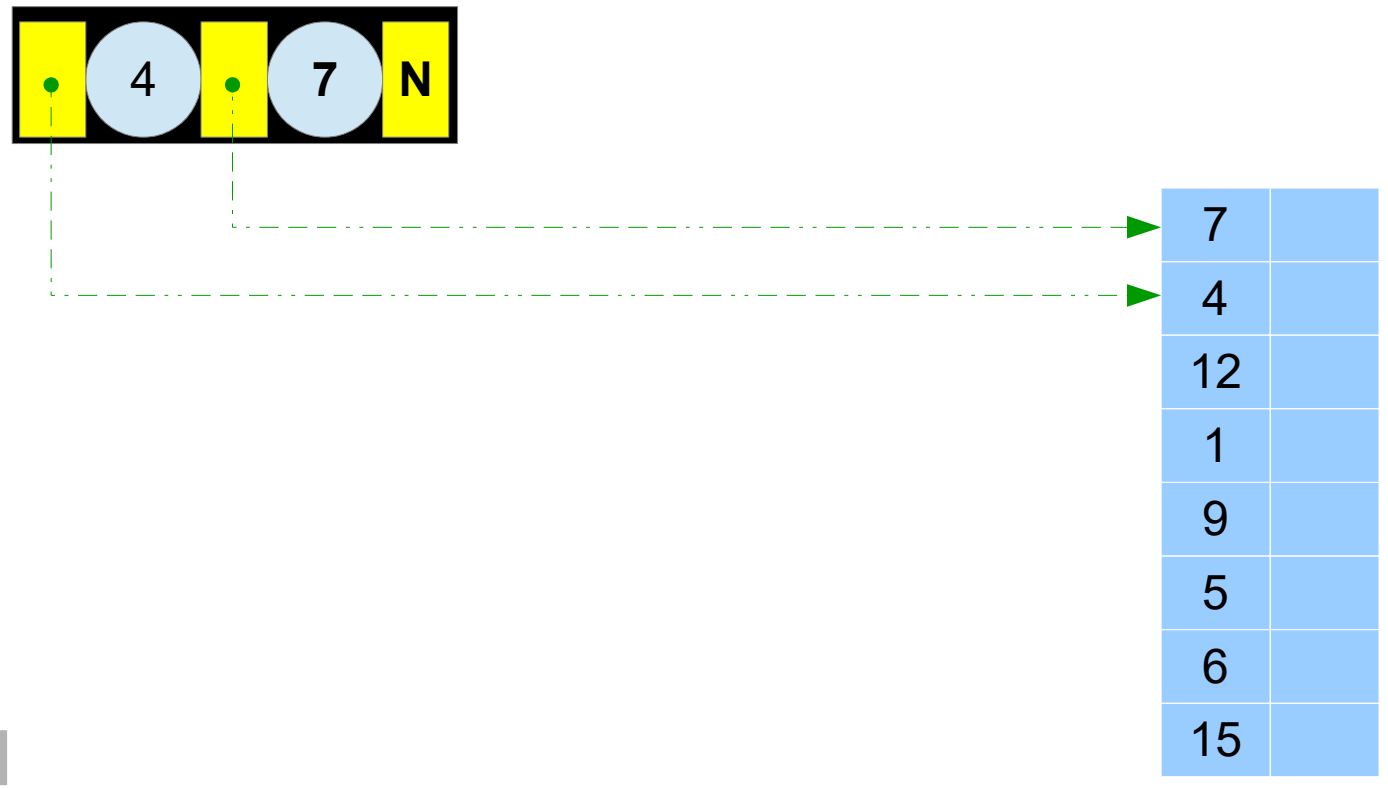
B+ bomen – Voorbeeld

Toevoegen 7,4,12,1,9,5,6,15 voor orde B+ boom: $p_i = p_r = 3$



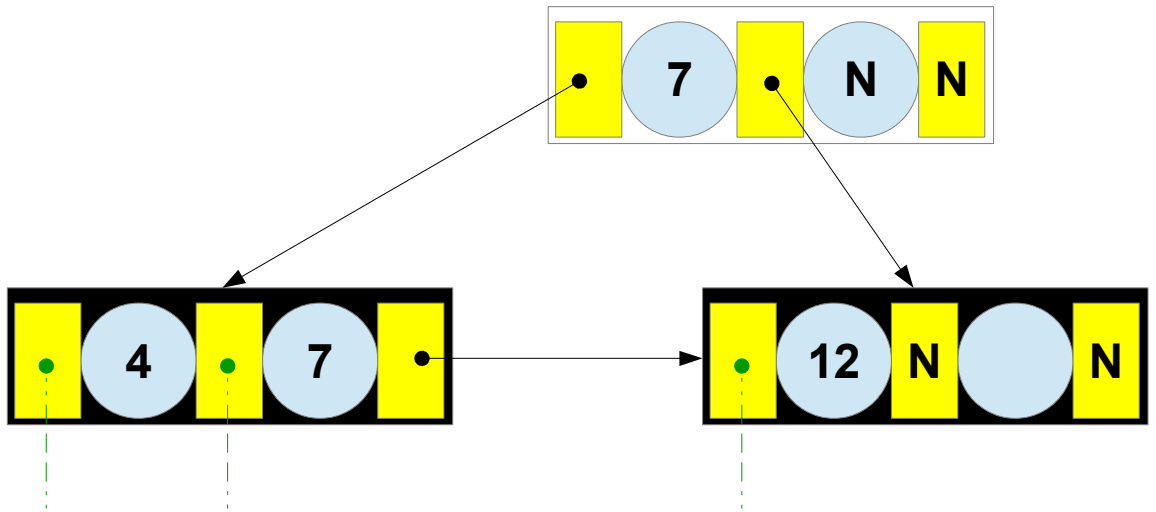
B+ bomen – Voorbeeld

Toevoegen 7,4,12,1,9,5,6,15 voor orde B+ boom: $p_i = p_r = 3$



B+ bomen – Voorbeeld

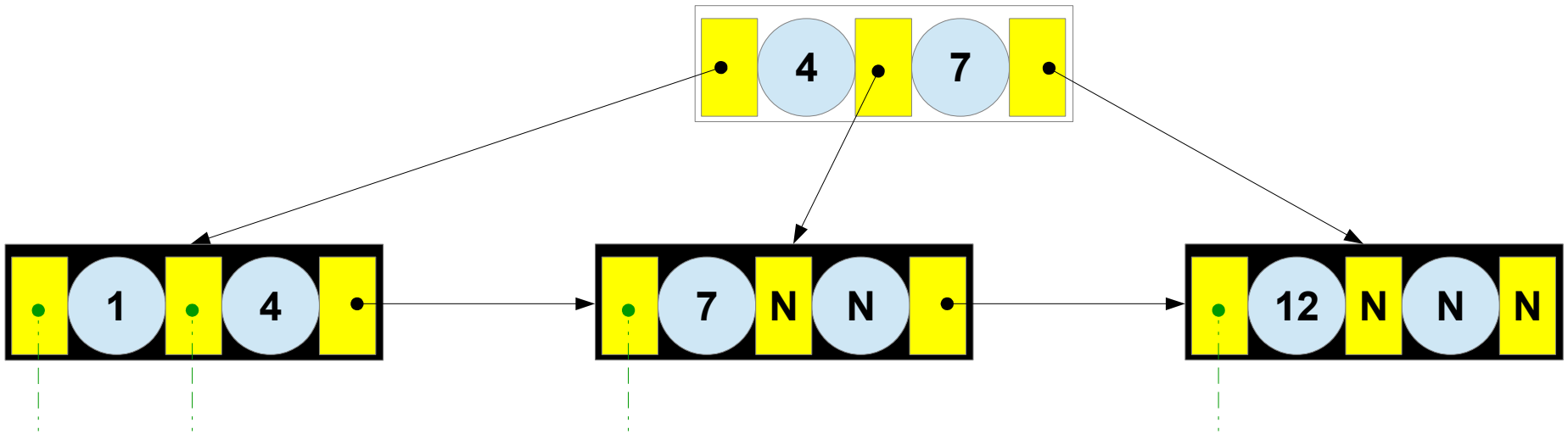
Toevoegen 7,4,12,1,9,5,6,15 voor orde B+ boom: $p_i = p_r = 3$



7	
4	
12	
1	
9	
5	
6	
15	

B+ bomen – Voorbeeld

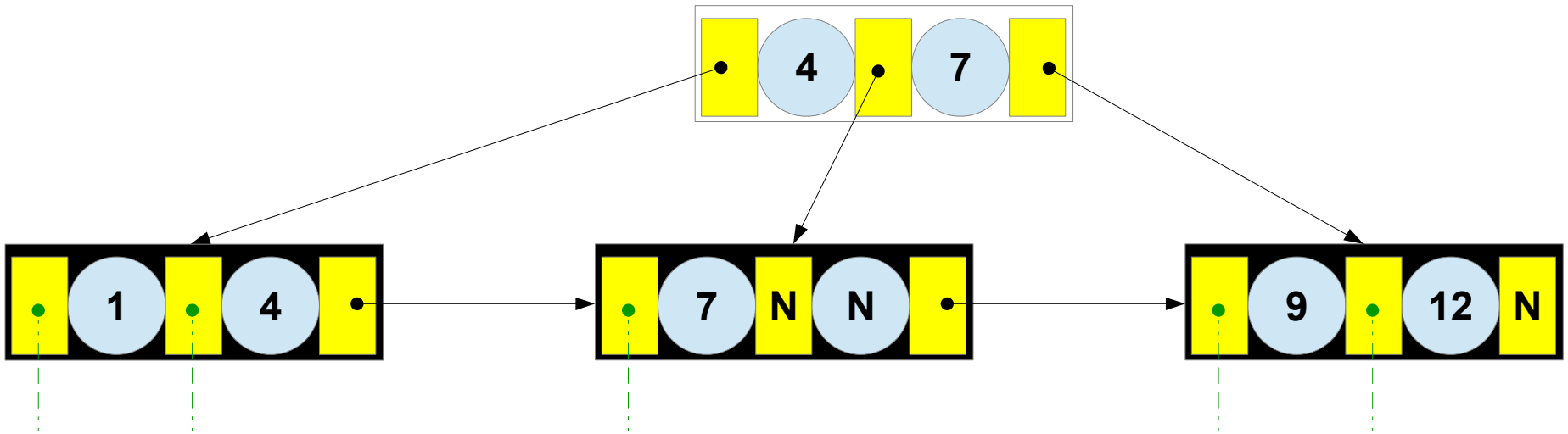
Toevoegen 7,4,12,1,9,5,6,15 voor orde B+ boom: $p_i = p_r = 3$



7	
4	
12	
1	
9	
5	
6	
15	

B+ bomen – Voorbeeld

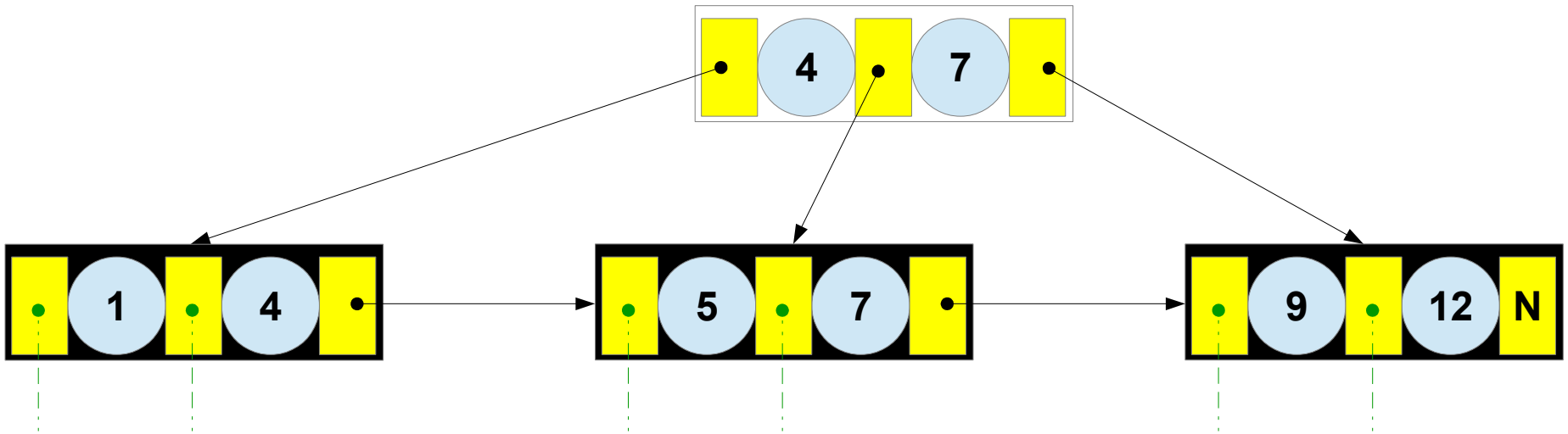
Toevoegen 7,4,12,1,9,5,6,15 voor orde B+ boom: $p_i = p_r = 3$



7
4
12
1
9
5
6
15

B+ bomen – Voorbeeld

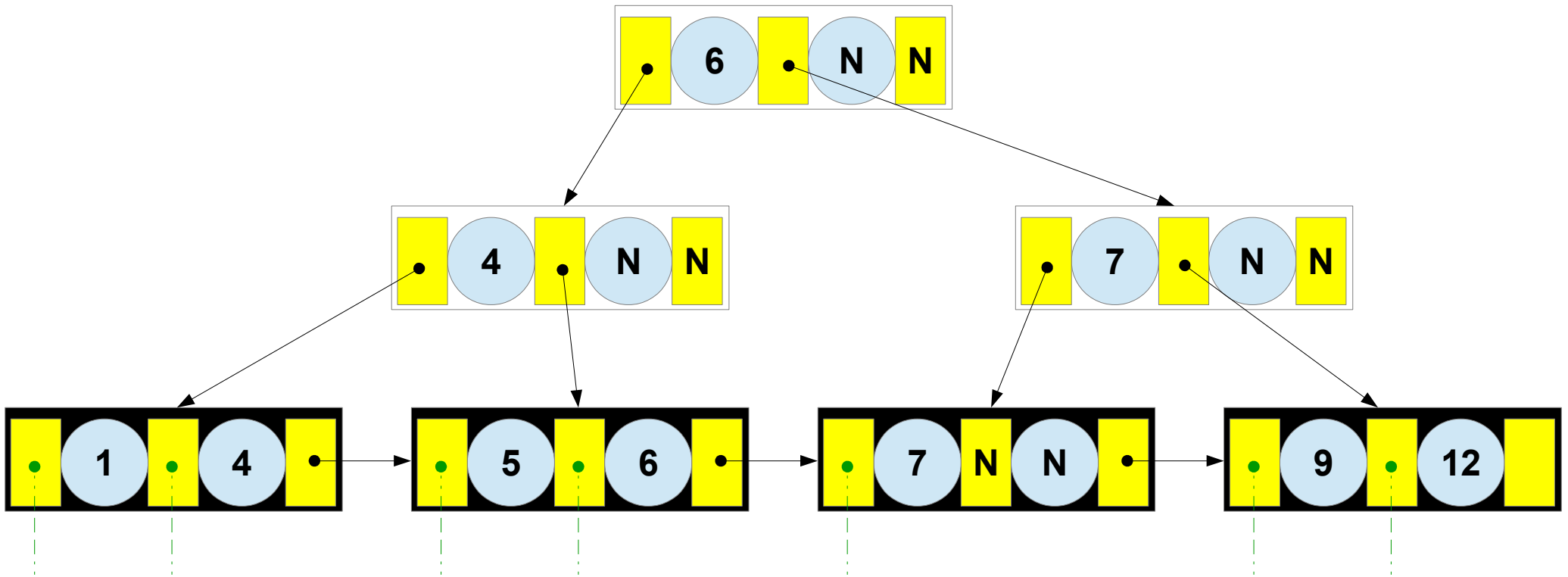
Toevoegen 7,4,12,1,9,5,6,15 voor orde B+ boom: $p_i = p_r = 3$



7
4
12
1
9
5
6
15

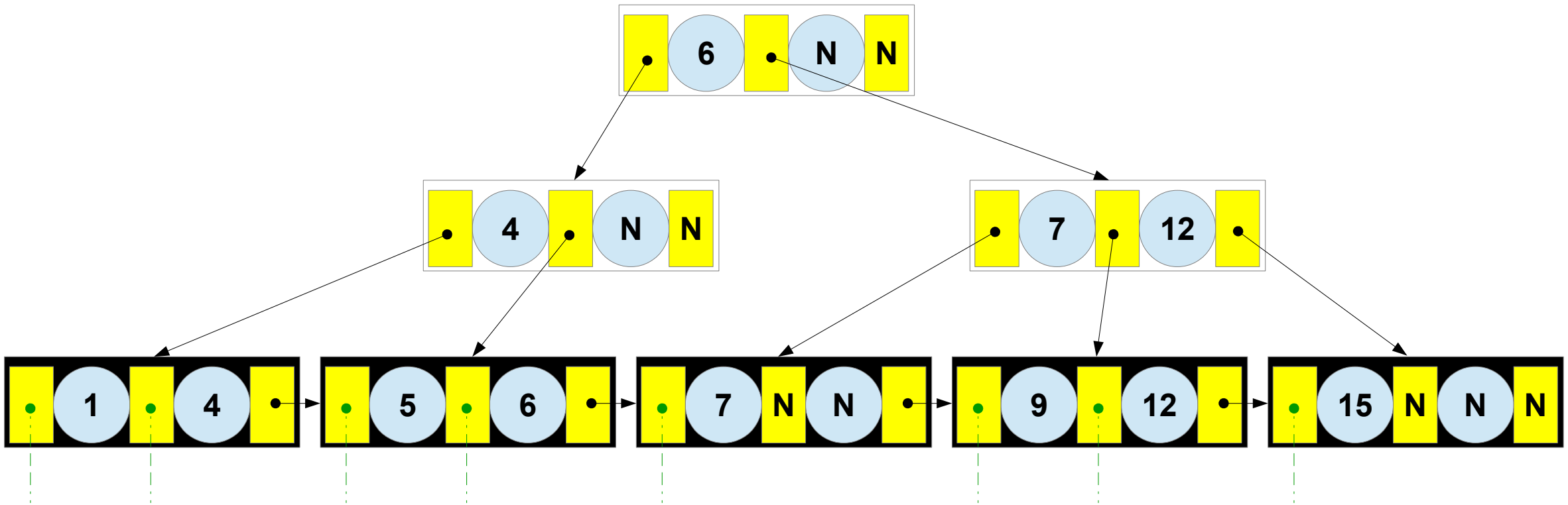
B+ bomen – Voorbeeld

Toevoegen 7,4,12,1,9,5,6,15 voor orde B+ boom: $p_i = p_r = 3$



B+ bomen – Voorbeeld

Toevoegen 7,4,12,1,9,5,6,15 voor orde B+ boom: $p_i = p_r = 3$

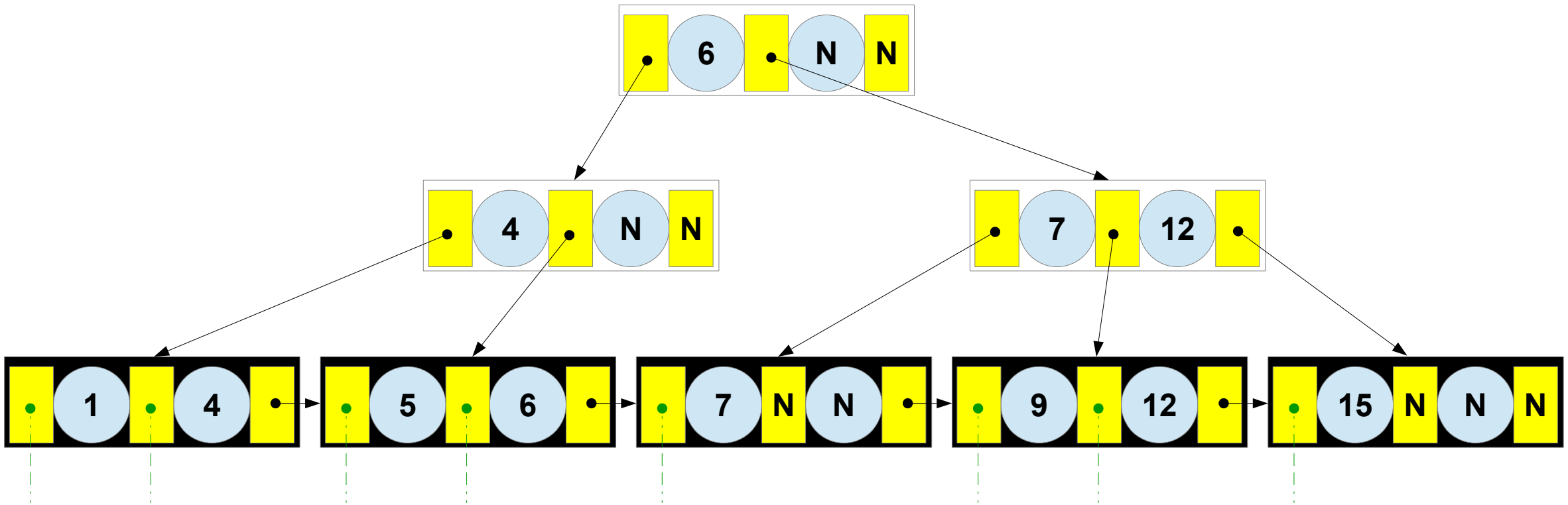


Algoritme verwijderen B+ bomen

1. Zoek naar bladnode n dat mogelijks K bevat.
    1.1. Bevat de bladnode K, Verwijder (K,Pr).
    1.2. Als bladnode n minder dan halfvol
        1.2.1 merge met bladnodes in zelfde parent, eerst links dan rechts
            - Adjacente bladnodes samenvoegen in de linkse bladnode
            => merge dan ook op het parentniveau
            - Verwijder in parent maxwaarde links en pointer rechts blad
        1.2.2 Als nog steeds bekomen bladnode minder dan halfvol
            => merge over verschillende parentniveaus heen
    1.3. Bevatten de parents van bladnode ook K
            => vervang K door nieuwe max van de bladnode n
    1.4. Als interne nodes minder dan halfvol dan analoog aan 1.2

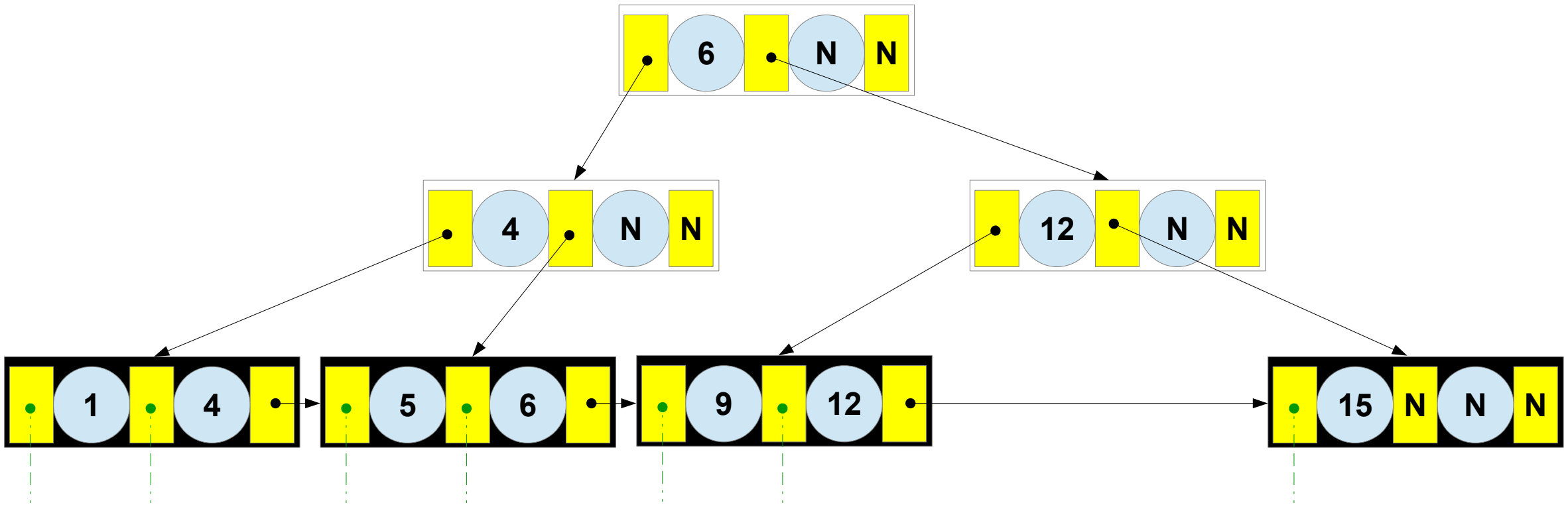
B+ bomen – Voorbeeld

Verwijderen 7,12,9,15 voor orde B+ boom: $p_i = p_r = 3$



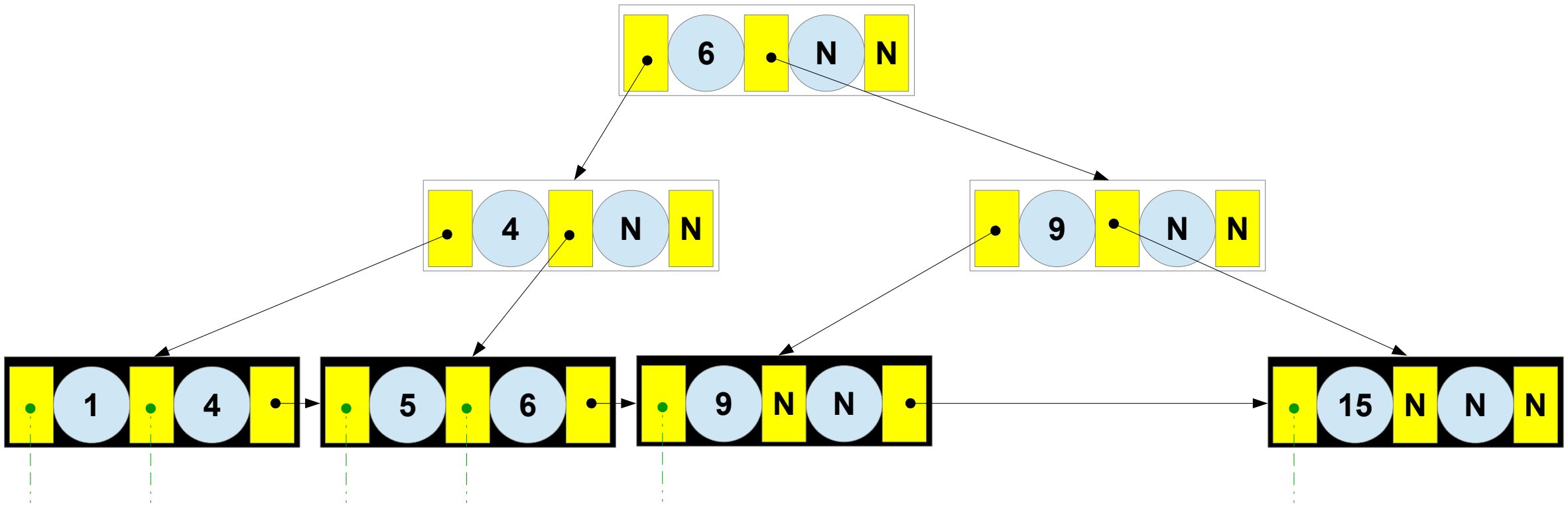
B+ bomen – Voorbeeld

Verwijderen 7,12,9,15 voor orde B+ boom: $p_i = p_r = 3$



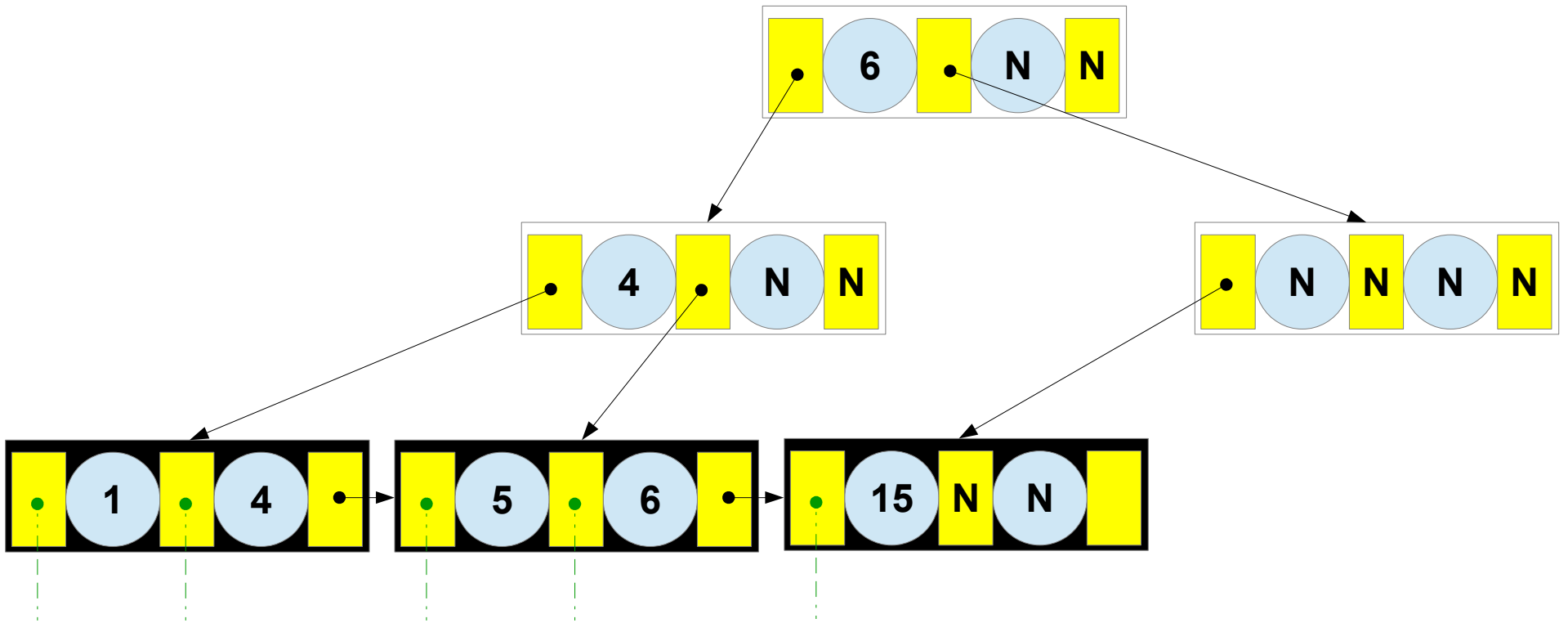
B+ bomen – Voorbeeld

Verwijderen 7,12,9,15 voor orde B+ boom: $p_i = p_r = 3$



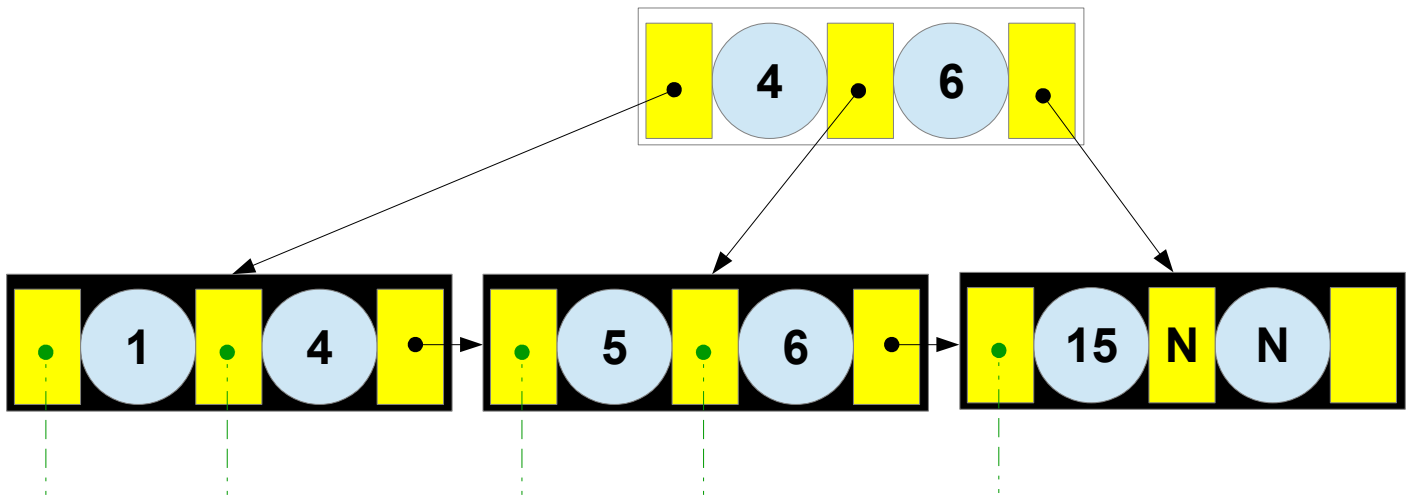
B+ bomen – Voorbeeld

Verwijderen 7,12,9,15 voor orde B+ boom: $p_i = p_r = 3$



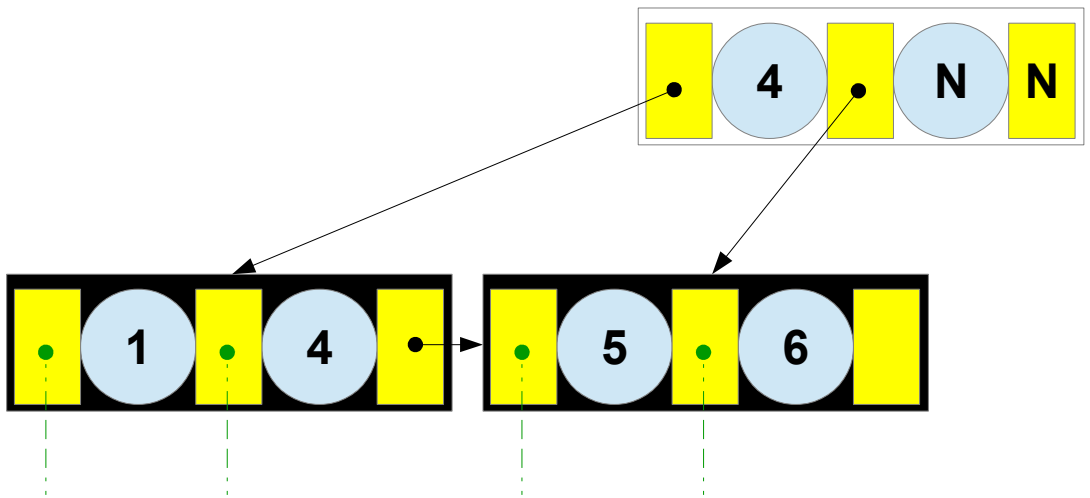
B+ bomen – Voorbeeld

Verwijderen 7,12,9,15 voor orde B+ boom: $p_i = p_r = 3$



B+ bomen – Voorbeeld

Verwijderen 7,12,9,15 voor orde B+ boom: $p_i = p_r = 3$



Algoritmes B+ bomen

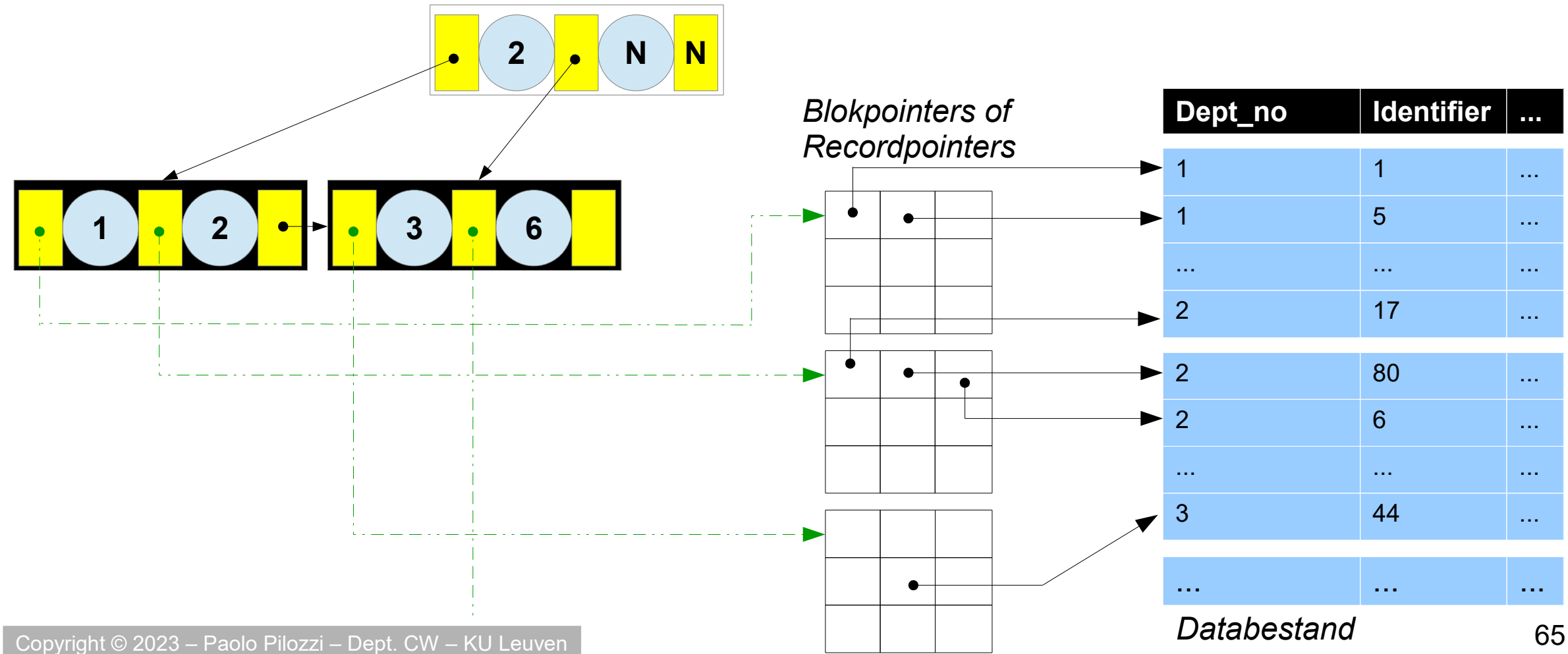
Volgende visualisatie werkt met duaal algoritme:

https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html

Dit duaal algoritme werkt averechts t.a.v. algoritme in slides (Knuth):
=> Intervallen in interne nodes half-open naar rechts ipv. links
=> Splitsing bladnodes rond K': K' naar rechtse bladnode ipv. links

Toepassing in oefenzittingen en op examens: algoritme van de slides!

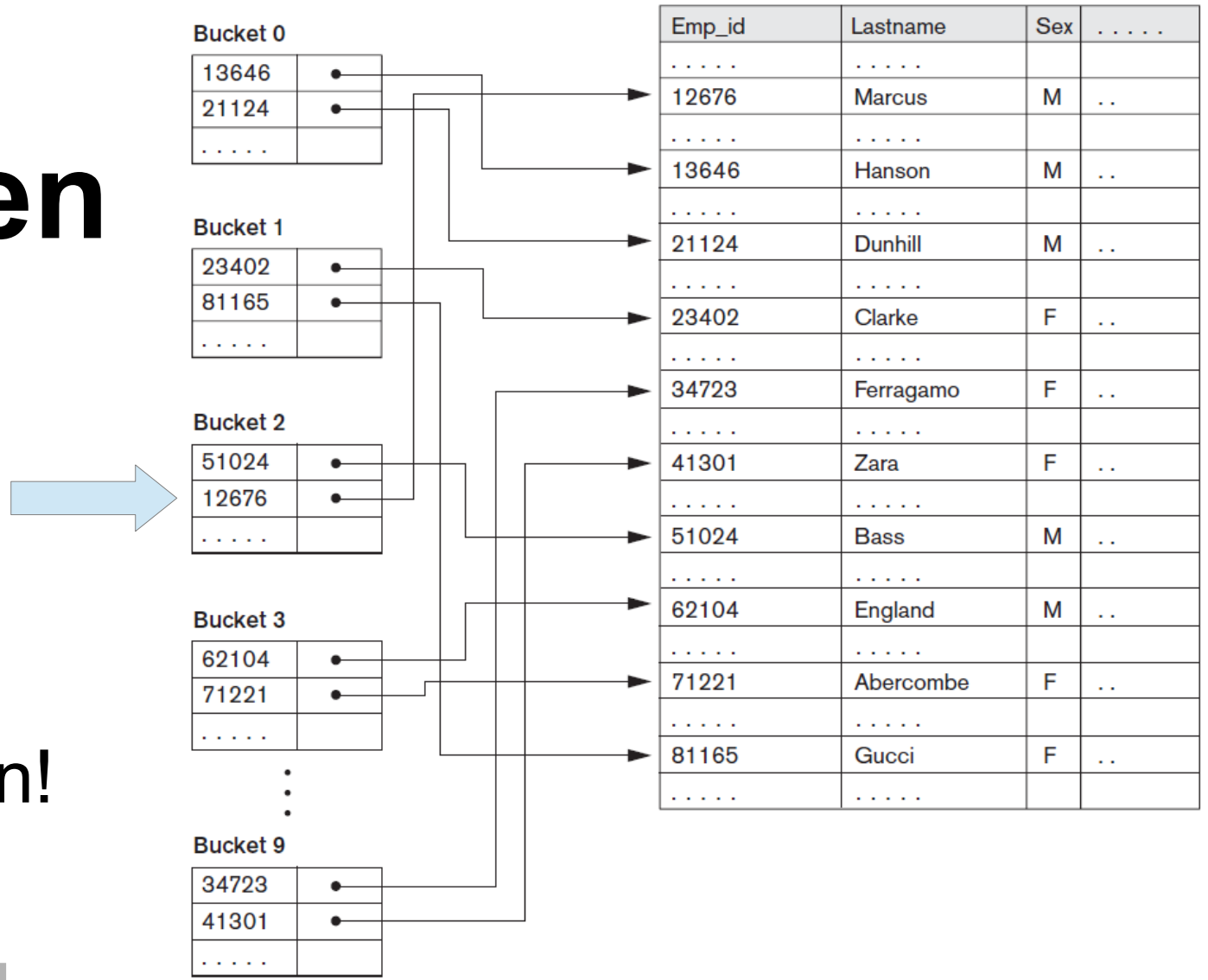
B+ bomen: Sec. niet-unieke idx.



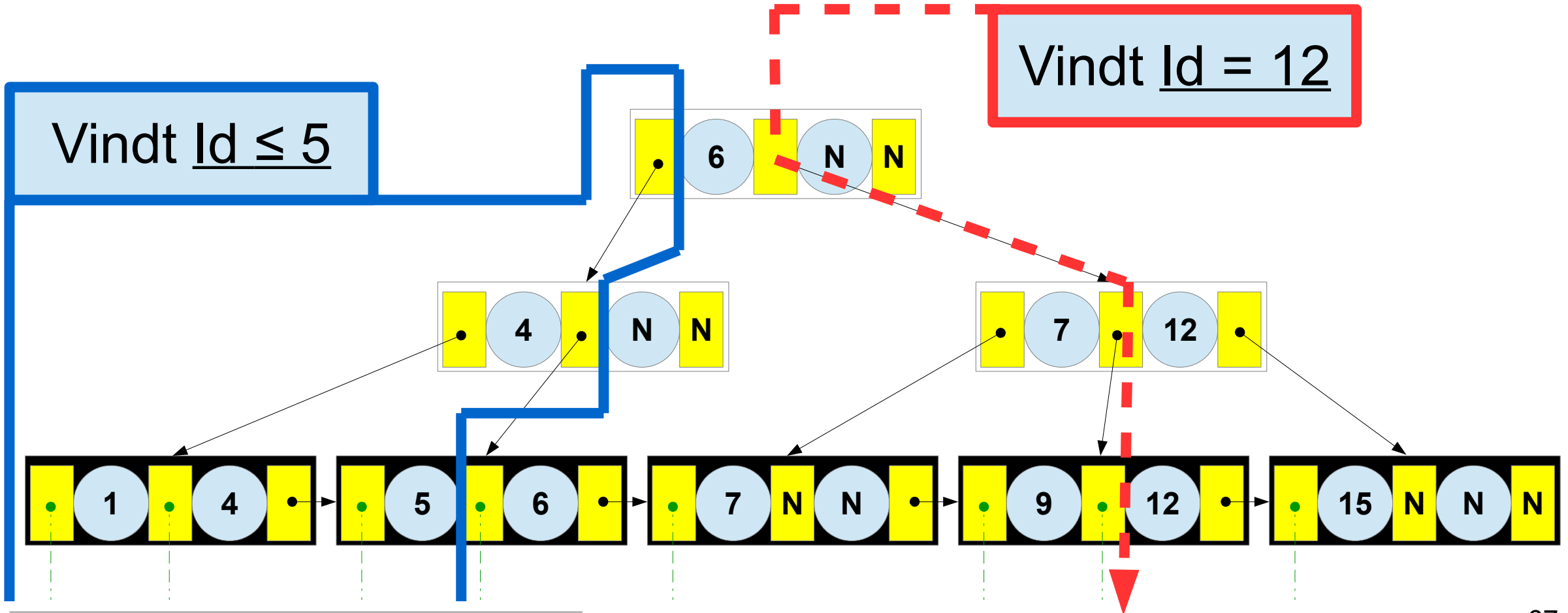
Hash-index vs. B+ bomen

Vindt Id = 12876
=> h(12867) = 2
=> gevonden

Vindt Id ≤ 12876
=> Door hele lijst gaan!



Hash Index vs B+ Bomen



Hash-index vs. B+ bomen

Elke multiniveau-index kan orde gebaseerde opvragingen snel, niet enkel "het speciale geval" bomen.
Merk bovendien op: Een multiniveau-index met fan-out = 2 is vergelijkbaar aan binair zoeken.

Hash-indexen dienen voor constante tijd opzoekingen met gelijkheden. Echter, ze bewaren geen verbanden. Dus opzoekingen met andere condities dan gelijkheden: lineair.

Hash-indexen kunnen als top-level index gebruikt worden.