BASI DI DATI

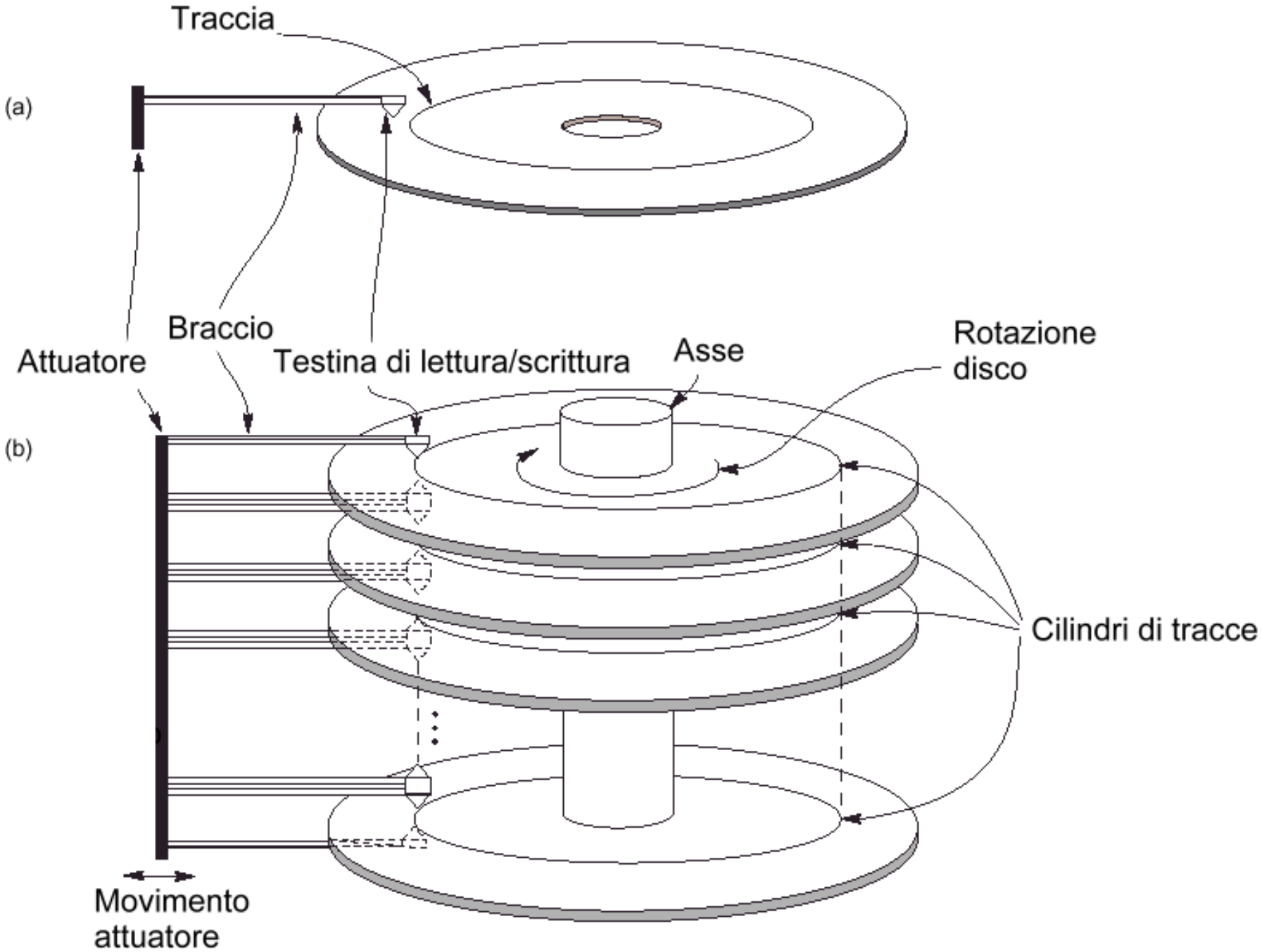
PROGETTAZIONE FISICA

Polese G. Caruccio L. Breve B.

a.a. 2023/2024

Memoria principale e secondaria

- I programmi possono fare riferimento solo a dati in memoria principale
- Le basi di dati debbono essere (sostanzialmente) in memoria secondaria per due motivi:
  - Dimensioni
  - Persistenza
- I dati in memoria secondaria possono essere utilizzati solo se prima trasferiti in memoria principale (questo spiega i termini "principale" e "secondaria")


Traccia
(a)
Attuatore
Braccio
Testina di lettura/scrittura
Asse
Rotazione
disco
(b)
Cilindri di tracce
Movimento
attuatore

Memoria principale e secondaria, 2

- I dispositivi di memoria secondaria sono organizzati in **blocchi** di lunghezza (di solito) **fissa** (ordine di grandezza: alcuni KB)
- Le uniche operazioni sui dispositivi sono la lettura e la scrittura di una **pagina**, cioè dei dati di un blocco (cioè di una stringa di byte);
- Per comodità consideriamo **blocco** e **pagina** sinonimi

Memoria principale e secondaria, 3

- Accesso a memoria secondaria (dati dal sito della Seagate e da Wikipedia, 2009):
  - tempo di **posizionamento della testina** (**seek time**): in media 2-15ms (a seconda del tipo di disco)
  - tempo di **latenza** (**rotational delay**): 2-6ms (conseguenza della velocità di rotazione, 4-15K giri al minuto)
  - tempo di **trasferimento** di un blocco: frazioni di ms (conseguenza della velocità di trasferimento, 100-300MB al secondo)

  In media non meno di qualche ms

Memoria principale e secondaria, 3

- Commenti:
  - Il costo di un accesso a memoria secondaria è quattro o più ordini di grandezza maggiore di quello per operazioni in memoria centrale
  - Nelle applicazioni "**I/O bound**" (cioè con molti accessi a memoria secondaria e relativamente poche operazioni) il costo dipende esclusivamente dal numero di accessi a memoria secondaria
  - Accessi a blocchi “vicini” costano meno (**contiguità**)

Strutture

- Sequenziali
- Calcolate ("Hash")
- Ad albero (di solito, indici)

Primarie

Primarie (e talvolta secondarie)

Secondarie e primarie

Strutture sequenziali

- Esiste un ordinamento fra le ennuple, che può essere rilevante ai fini della gestione
  - **seriale**: ordinamento fisico ma non logico
  - **ordinata**: ordinamento fisico coerente con quello di un campo

Struttura seriale

- Chiamata anche:
  - "entry sequenced"
  - file heap ("mucchio")
  - file disordinato
- Molto diffusa nelle basi di dati relazionali, associata a indici secondari
- Gli inserimenti vengono effettuati (varianti)
  - in coda (con riorganizzazioni periodiche)
  - al posto di record cancellati
- La gestione è molto semplice, ma spesso inefficiente

Struttura sequenziale ordinata

- Ogni tupla ha una posizione basata sul valore di un campo “Chiave” (o “pseudo-chiave”)
- Storicamente, le strutture sequenziali ordinate venivano usate da processi batch su dispositivi sequenziali (nastri). I dati venivano riposti in un *file principale*, le modifiche venivano raccolte in *file differenziali*. Tali file venivano *periodicamente fusi* (*merge*). Tale pratica non è più attuale.

Struttura sequenziale ordinata

- Principali problemi: inserimenti o modifiche aumentano lo spazio fisico utilizzato – richiedono riorganizzazioni periodiche
- Opzioni per evitare riordinamenti globali:
  - Lasciare un certo numero di slot liberi durante il primo caricamento. Successivamente, si applicano operazioni di riordinamento 'locale'.
  - Integrare i file ordinati sequenzialmente con un *overflow file*, dove le nuove tuple vengono inserite in blocchi collegati, formando una *catena di overflow*.

Strutture ordinate

- Permettono ricerche binarie, ma solo fino ad un certo punto (come troviamo la "metà" del file?)
- Nelle basi di dati relazionali si utilizzano quasi solo in combinazione con indici (file ISAM o file ordinati con indice primario)

File hash

- Permettono un accesso diretto molto efficiente (da alcuni punti di vista)
- La tecnica si basa su quella utilizzata per le tavole hash in memoria centrale

Tavola hash

- Obiettivo: accesso diretto ad un insieme di record sulla base del valore di un campo (detto **chiave**, che per semplicità supponiamo identificante, ma non è necessario)
- Se i possibili valori della chiave sono in numero paragonabile al numero di record (e corrispondono ad un "tipo indice") allora usiamo un array; ad esempio: università con 1000 studenti e numeri di matricola compresi fra 1 e 1000 o poco più e file con tutti gli studenti

Tavola hash

- Se i possibili valori della chiave sono molti di più di quelli effettivamente utilizzati, non possiamo usare l'array (spreco); ad esempio:
  - 40 studenti e numero di matricola di 6 cifre (un milione di possibili chiavi)

Tavola hash, 2

- Volendo continuare ad usare qualcosa di simile ad un array, ma senza sprecare spazio, possiamo pensare di trasformare i valori della chiave in possibili indici di un array:
  - **funzione hash**:
    - ✓ associa ad ogni valore della chiave un "indirizzo", in uno spazio di dimensione paragonabile (leggermente superiore) rispetto a quello strettamente necessario
    - ✓ poiché il numero di possibili chiavi è molto maggiore del numero di possibili indirizzi ("lo spazio delle chiavi è più grande dello spazio degli indirizzi"), la funzione non può essere iniettiva e quindi esiste la possibilità di collisioni (chiavi diverse che corrispondono allo stesso indirizzo)
    - ✓ le buone funzioni hash distribuiscono in modo casuale e uniforme, riducendo le probabilità di collisione (che si riduce aumentando lo spazio ridondante)

Un esempio

- 40 record
- tavola hash con 50 posizioni:
  - 1 collisione a 4
  - 2 collisioni a 3
  - 5 collisioni a 2
  - 20 record senza collisione

M	M mod 50
60600	0
66301	1
205751	1
205802	2
200902	2
116202	2
200604	4
66005	5
116455	5
200205	5
201159	9
205610	10
201260	10
102360	10
205460	10
205912	12
205762	12
200464	14
205617	17
205667	17

M	M mod 50
200268	18
205619	19
210522	22
205724	24
205977	27
205478	28
200430	30
210533	33
205887	37
200138	38
102338	38
102690	40
115541	41
206092	42
205693	43
205845	45
200296	46
205796	46
200498	48
206049	49

Tavola hash

0	60600
1	66301
2	205802
4	200604
5	66005
9	201159
10	205610
12	205912
14	200464
17	205617
18	200268
19	205619

22	210522
24	205724
27	205977
28	205478
30	200430
33	210533
37	205887
38	102338

40	102690
41	115541
42	206092
43	205693
45	205845
46	205796
48	200498
49	206049

1	205751
2	200902
2	116202
5	116455
5	200205
10	201260
10	102360
10	205460
12	205762
17	205667
38	200138
46	200296

Un esempio

- 40 record
- tavola hash con 50 posizioni:
  - 1 collisione a 4
  - 2 collisioni a 3
  - 5 collisioni a 2
  - 20 record senza collisione
- numero medio di accessi:

  (28x1+8x2+3x3+1x4)/40 = 1,425

M	M mod 50
60600	0
66301	1
205751	1
205802	2
200902	2
116202	2
200604	4
66005	5
116455	5
200205	5
201159	9
205610	10
201260	10
102360	10
205460	10
205912	12
205762	12
200464	14
205617	17
205667	17

M	M mod 50
200268	18
205619	19
210522	22
205724	24
205977	27
205478	28
200430	30
210533	33
205887	37
200138	38
102338	38
102690	40
115541	41
206092	42
205693	43
205845	45
200296	46
205796	46
200498	48
206049	49

Tavola hash, collisioni

- Varie tecniche:
  - posizioni successive disponibili
  - tabella di overflow (gestita in forma collegata)
  - funzioni hash "alternative"
- Nota:
  - le collisioni ci sono (quasi) sempre
  - le collisioni multiple hanno probabilità che decresce al crescere della molteplicità
  - la molteplicità media delle collisioni è molto bassa

Un esempio

- 40 record
- tavola hash con 50 posizioni:
  - 1 collisione a 4
  - 2 collisioni a 3
  - 5 collisioni a 2
  - 20 record senza collisione
- numero medio di accessi:

  (28x1+8x2+3x3+1x4)/40 = 1,425

M	M mod 50
60600	0
66301	1
205751	1
205802	2
200902	2
116202	2
200604	4
66005	5
116455	5
200205	5
201159	9
205610	10
201260	10
102360	10
205460	10
205912	12
205762	12
200464	14
205617	17
205667	17

M	M mod 50
200268	18
205619	19
210522	22
205724	24
205977	27
205478	28
200430	30
210533	33
205887	37
200138	38
102338	38
102690	40
115541	41
206092	42
205693	43
205845	45
200296	46
205796	46
200498	48
206049	49

File hash

- L'idea è la stessa della tavola hash, ma si basa sull'organizzazione in blocchi:
  - ogni blocco contiene più record
  - lo spazio degli indirizzi è più piccolo
    - ✓ Nell'esempio, con fattore di blocco pari a 10, possiamo usare "mod 5" invece di "mod 50"

Un file hash

0	1	2	3	4
60600	66301	205802	200268	200604
66005	205751	200902	205478	201159
116455	115541	116202	210533	200464
200205	200296	205912	200138	205619
205610	205796	205762	102338	205724
201260		205617	205693	206049
102360		205667	200498	
205460		210522		
200430		205977		
102690		205887		
205845		206092		

Nell’esempio

- 40 record
- tavola hash con 50 posizioni:
  - 1 collisione a 4
  - 2 collisioni a 3
  - 5 collisioni a 2
  - in totale, 12 record in overflow

  numero medio di accessi: 1,425
- file hash con fattore di blocco 10; 5 blocchi con 10 posizioni ciascuno:
  - due soli record in overflow

  numero medio di accessi: (42/40) = 1,05
- perché?

Collisioni, stima

- Lunghezza media delle catene di overflow, al variare di
  - Numero di record esistenti: T
  - Numero di blocchi: B
  - Fattore di blocco: F
  - Coefficiente di riempimento: T/(FxB)

	1	2	3	5	10	(F)
.5	0.5	0.177	0.087	0.031	0.005	
.6	0.75	0.293	0.158	0.066	0.015	
.7	1.167	0.494	0.286	0.136	0.042	
.8	2.0	0.903	0.554	0.289	0.110	
.9	4.495	2.146	1.377	0.777	0.345	

T/(FxB)

File hash, osservazioni

- Le collisioni (overflow) sono di solito gestite con blocchi collegati
- È l'organizzazione più efficiente per l'accesso diretto basato su valori della chiave con condizioni di uguaglianza (accesso puntuale):
  - costo medio di poco superiore all'unità (il caso peggiore è molto costoso ma talmente improbabile da poter essere ignorato)
- Non è efficiente per ricerche basate su intervalli (né per ricerche basate su altri attributi)
- I file hash "degenerano" se si riduce lo spazio sovrabbondante: funzionano solo con file la cui dimensione non varia molto nel tempo

Indici di file

- Indice:
  - struttura ausiliaria per l'accesso (efficiente) ai record di un file sulla base dei valori di un campo (o di una "concatenazione di campi") detto chiave (o, meglio, pseudochiave, perché non è necessariamente identificante);
- Idea fondamentale: l'indice analitico di un libro: lista di coppie (termine, pagina), ordinata alfabeticamente sui termini, posta in fondo al libro e separabile da esso
- Un indice ***I*** di un file ***F*** è un altro file, con record a due campi: chiave e indirizzo (dei record di ***F*** o dei relativi blocchi), ordinato secondo i valori della chiave

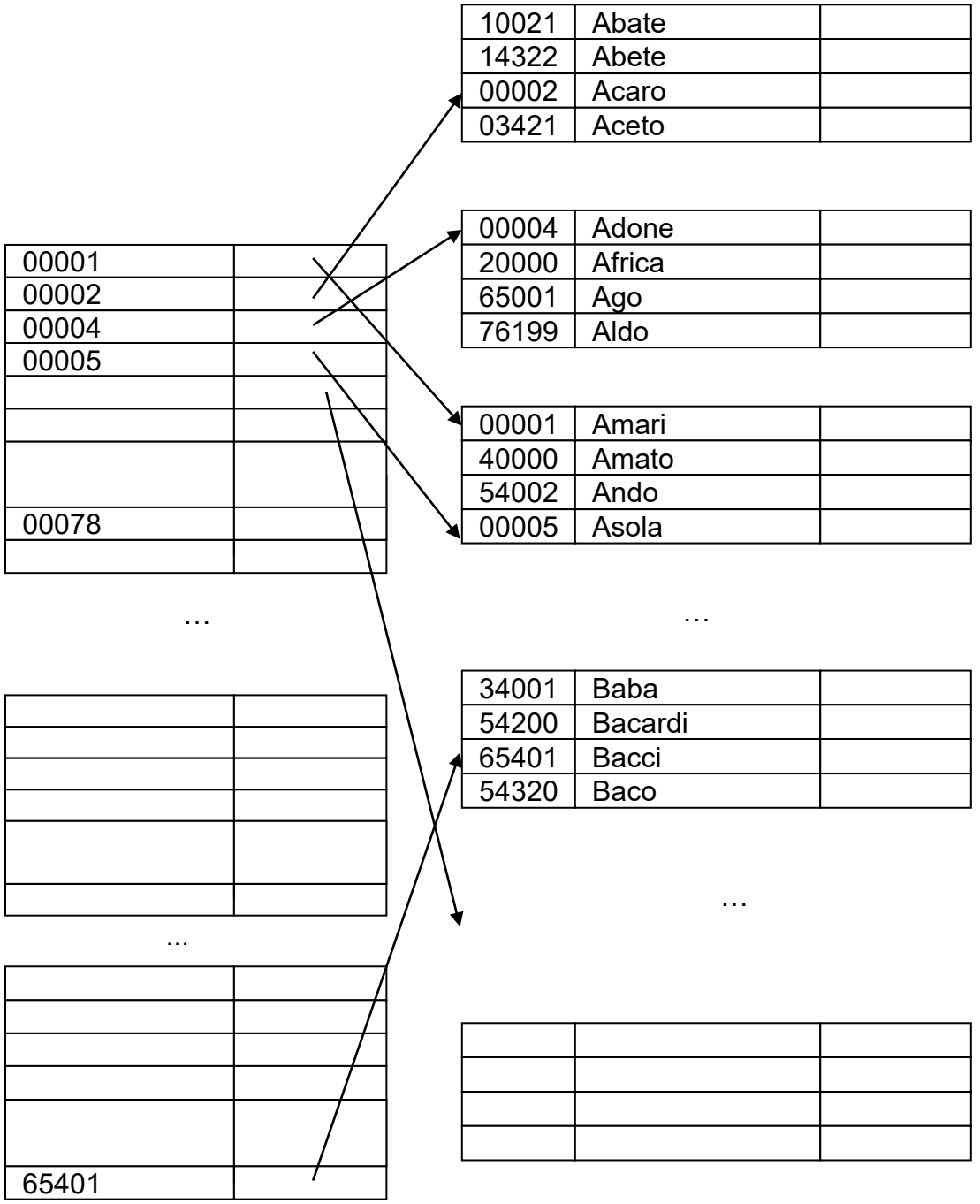
Tipi di indice

- Indice primario:
  - su un campo sul cui ordinamento è basata la memorizzazione (detti anche indici di cluster, anche se talvolta si chiamano primari quelli su una chiave identificante e di cluster quelli su una pseudochiave non identificante)
- Indice secondario
  - su un campo con ordinamento diverso da quello di memorizzazione

Tipi di indice, commenti

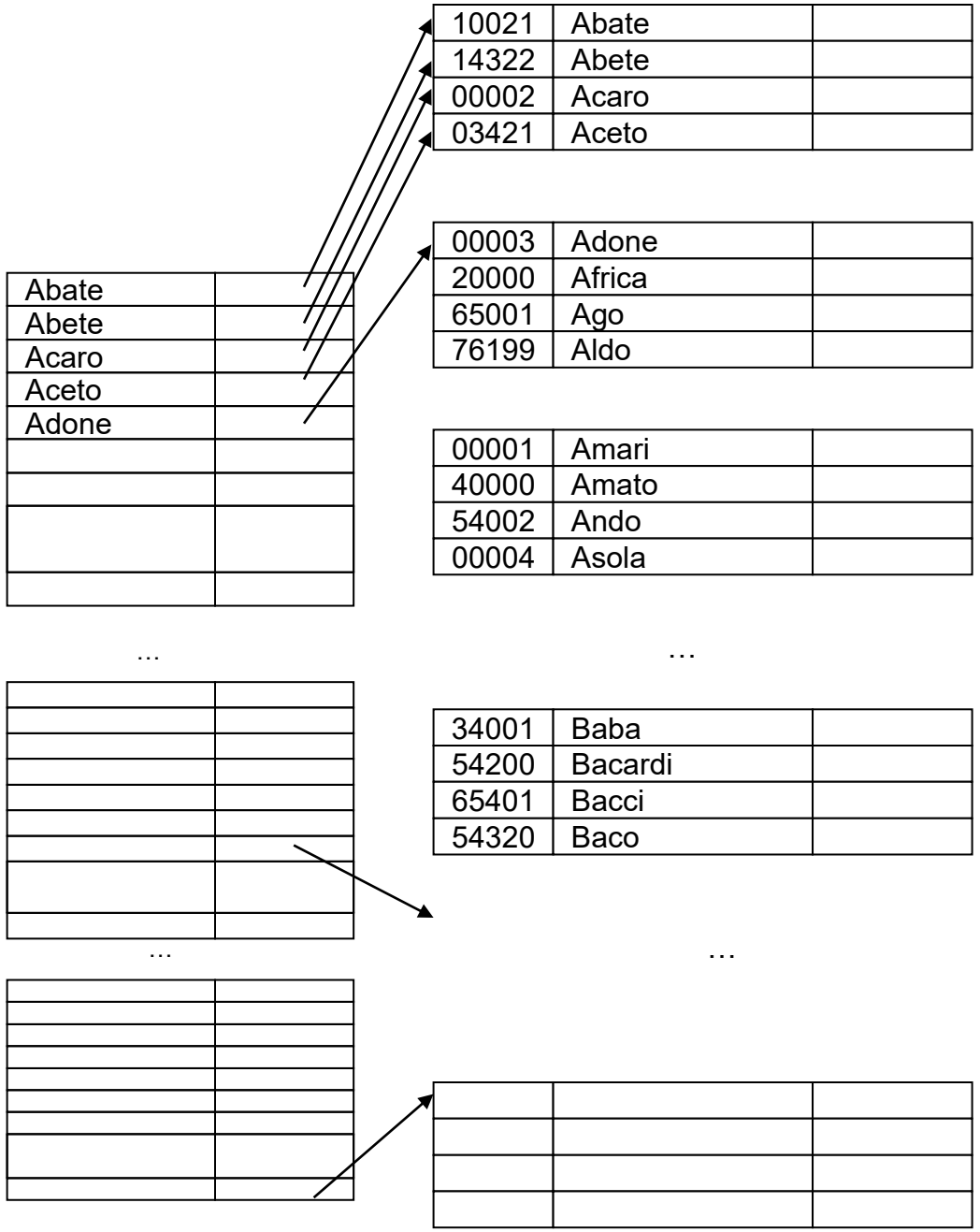
- Esempio, sempre rispetto ad un libro
  - indice generale
  - indice analitico
- I benefici legati alla presenza di indici secondari sono molto più sensibili
- Ogni file può avere al più un indice primario e un numero qualunque di indici secondari (su campi diversi). Esempio:
  - una guida turistica può avere l'indice dei luoghi e quello degli artisti
- Un file hash non può avere un indice primario

Indice secondario

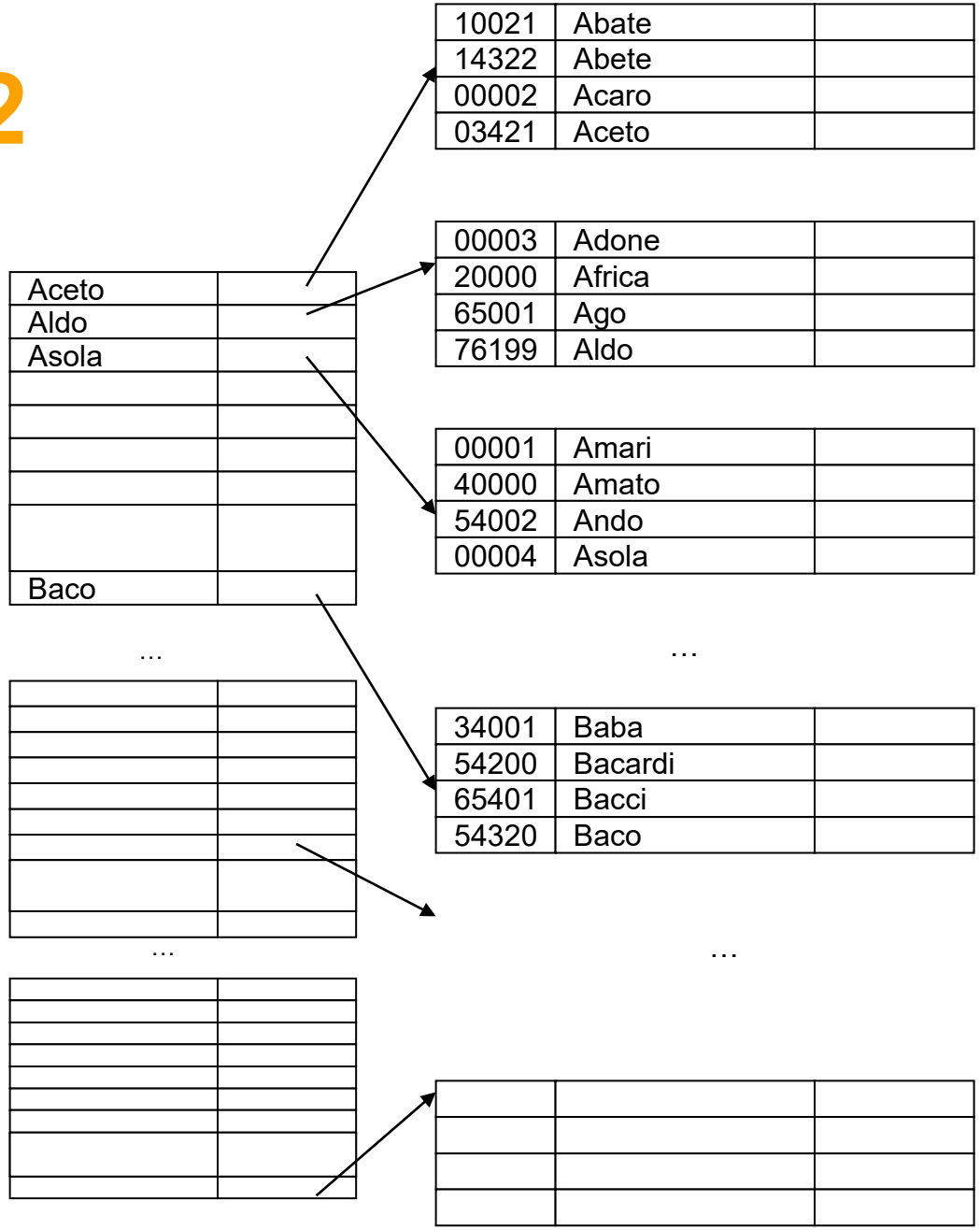


Indice primario, 1

Servono tutti i riferimenti?



Indice primario, 2



Tipi di indice, ancora

- Indice denso:
  - contiene tutti i valori della chiave (e quindi, per indici su campi identificanti, un riferimento per ciascun record del file)
- Indice sparso:
  - contiene solo alcuni valori della chiave e quindi (anche per indici su campi identificanti) un numero di riferimenti inferiore rispetto ai record del file
- Un indice primario
  - di solito è sparso
  - denso permette di eseguire operazioni sugli indirizzi, senza accedere ai record
- Un indice secondario deve essere denso

Indici densi, un’osservazione

- Si possono usare, come detto, puntatori ai blocchi oppure puntatori ai record
  - I puntatori ai blocchi sono più compatti
  - I puntatori ai record permettono di
    - ✓ semplificare alcune operazioni (effettuate solo sull'indice, senza accedere al file se non quando indispensabile)

Dimensioni dell'indice

- L numero di record nel file
- B dimensione dei blocchi
- R lunghezza dei record (fissa)
- K lunghezza del campo pseudochiave
- P lunghezza degli indirizzi (ai blocchi)

N. di blocchi per il file (circa): $NF = L / (B/R)$

N. di blocchi per un indice denso: $ND = L / (B/(K+P))$

N. di blocchi per un indice sparso: $NS = NF / (B/(K+P))$

Dimensioni dell'indice, esempio

- L numero di record nel file 1.000.000
- B dimensione dei blocchi 4KB
- R lunghezza dei record (fissa per semplicità) 100B
- K lunghezza del campo chiave 4B
- P lunghezza degli indirizzi (ai blocchi) 4B

NF = L / (B/R) =~ 1.000.000/(4.000/100) = 25.000

ND = L / (B/(K+P)) =~ 1.000.000/(4.000/8) = 2.000

NS = NF / (B/(K+P)) =~ 25.000/(4.000/8) = 50

Caratteristiche degli indici

- Accesso diretto (sulla chiave) efficiente, sia puntuale sia per intervalli
- Scansione sequenziale ordinata efficiente:
  - Tutti gli indici (in particolare quelli secondari) forniscono un **ordinamento logico** sui record del file; con numero di accessi pari al numero di record del file (a parte qualche beneficio dovuto alla bufferizzazione)

Caratteristiche degli indici

- Modifiche della chiave, inserimenti, eliminazioni inefficienti (come nei file ordinati)
  - tecniche per alleviare i problemi:
    - ✓ file o blocchi di overflow
    - ✓ marcatura per le eliminazioni
    - ✓ riempimento parziale
    - ✓ blocchi collegati (non contigui)
    - ✓ riorganizzazioni periodiche
    - ✓ … (vedremo più avanti)

Indici su campi non chiave

- Ci sono (in generale) più record per un valore della (pseudo)chiave
  - primario sparso, possibili semplificazioni:
    - ✓ puntatori solo a blocchi con valori "nuovi"
  - primario denso:
    - ✓ una coppia con valore della chiave e riferimento per ogni record (quindi i valori della chiave si ripetono)
    - ✓ valore della chiave seguito dalla lista di riferimenti ai record con quel valore
    - ✓ valore della chiave seguito dal riferimento al primo record con quel valore (perde i benefici dell'indice primario denso legati alla possibilità di lavorare sui puntatori)

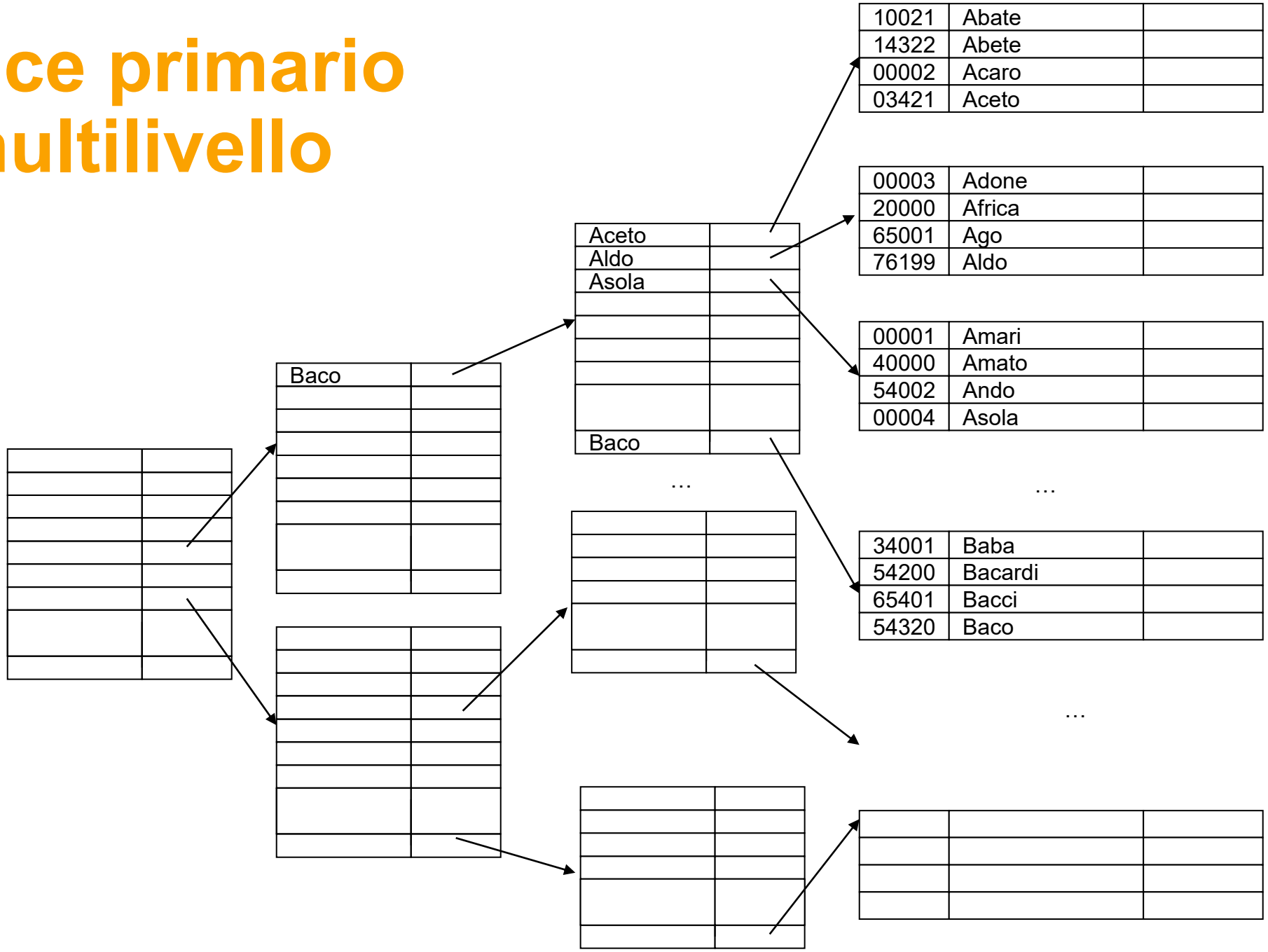
Indici su campi non chiave

- secondario (denso):
  - ✓ una coppia con valore della chiave e riferimento per ogni record (quindi i valori della chiave si ripetono)
  - ✓ un livello (di "indirezione") in più: per ogni valore della chiave l'indice contiene un record con riferimento al blocco di una struttura intermedia che contiene riferimenti ai record

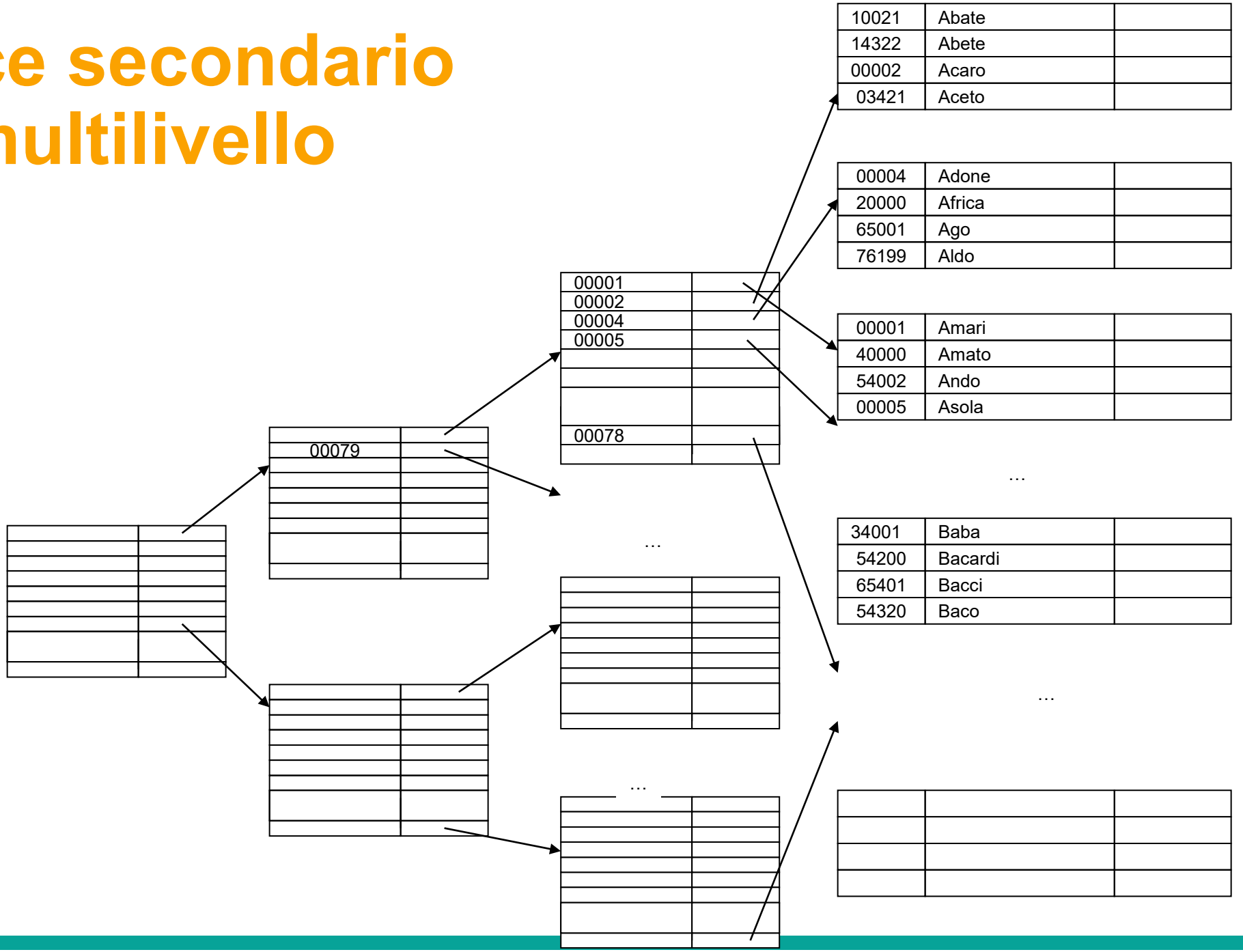
Indici multilivello

- Gli indici sono file essi stessi e quindi ha senso costruire indici sugli indici, per evitare di fare ricerche fra blocchi diversi (che potrebbero richiedere scansioni sequenziali)
- L'indice è ordinato e quindi l'indice sull'indice è primario (e sparso)
- Il tutto a più livelli, fino ad avere un livello con un solo blocco

Indice primario multilivello



Indice secondario multilivello



Indici multilivello

- I livelli sono di solito abbastanza pochi, perché
  - l'indice è ordinato, quindi l'indice sull'indice è sparso
  - i record dell'indice sono piccoli
- N_j numero di blocchi al livello j dell'indice (circa):
  - $N_j = N_{j-1} / (B/(K+P))$

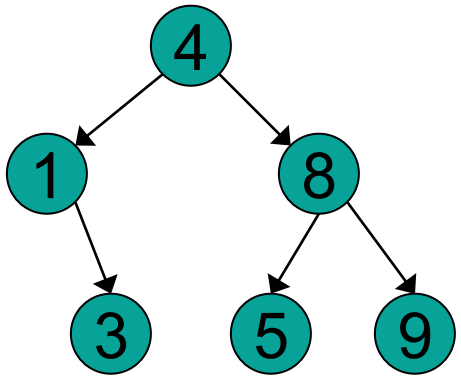
- Negli esempi numerici (B/(K+P) = 4.000/8=500)
  - Denso: $N_1 = 2.000$, $N_2 = 4$, $N_3 = 1$
  - Sparso: $N_1 = 50$, $N_2 = 1$

Indici, problemi

- Tutte le strutture di indice viste finora sono basate su strutture ordinate e quindi sono poco flessibili in presenza di elevata dinamicità
- Gli indici utilizzati dai DBMS sono più sofisticati:
  - indici dinamici multilivello: B-tree (intuitivamente: alberi di ricerca bilanciati)
    - ✓ Arriviamo ai B-tree per gradi
      - ➢ Alberi binari di ricerca
      - ➢ Alberi n-ari di ricerca
      - ➢ Alberi n-ari di ricerca bilanciati

Albero binario di ricerca

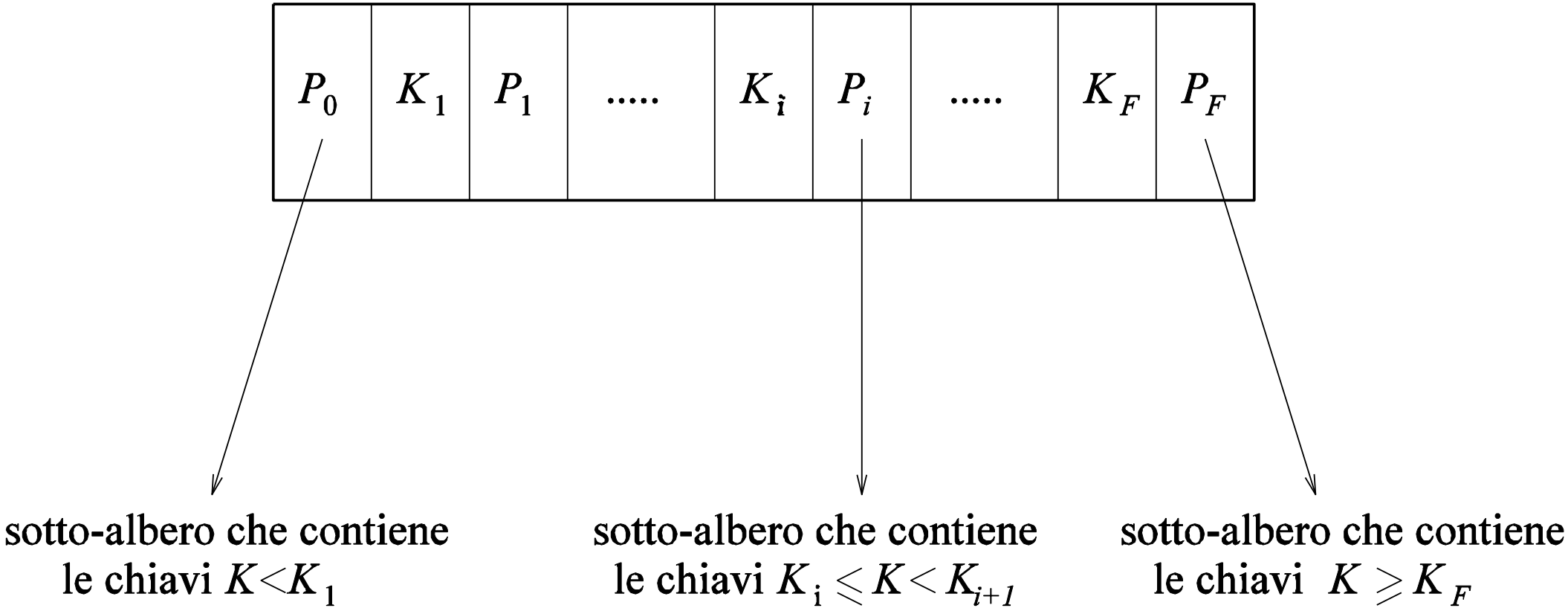
- Albero binario etichettato in cui per ogni nodo il sottoalbero sinistro contiene solo etichette minori di quella del nodo e il sottoalbero destro etichette maggiori
- tempo di ricerca (e inserimento), pari alla profondità:
  - logaritmico nel caso "medio" (assumendo un ordine di inserimento casuale)



Albero di ricerca di ordine P

- Ogni nodo ha (fino a) P figli e (fino a) P-1 etichette, ordinate
- Nell'i-esimo sottoalbero abbiamo tutte etichette maggiori della (i-1)-esima etichetta e minori della i-esima
- Ogni ricerca o modifica comporta la visita di un cammino radice foglia
- In strutture fisiche, un nodo corrisponde di solito ad un blocco e quindi ogni nodo intermedio ha molti figli (un “fan-out” molto grande, pari al fattore di blocco dell’indice)
- All’interno di un nodo, la ricerca è sequenziale (ma in memoria centrale!)
- La struttura è ancora (potenzialmente) rigida

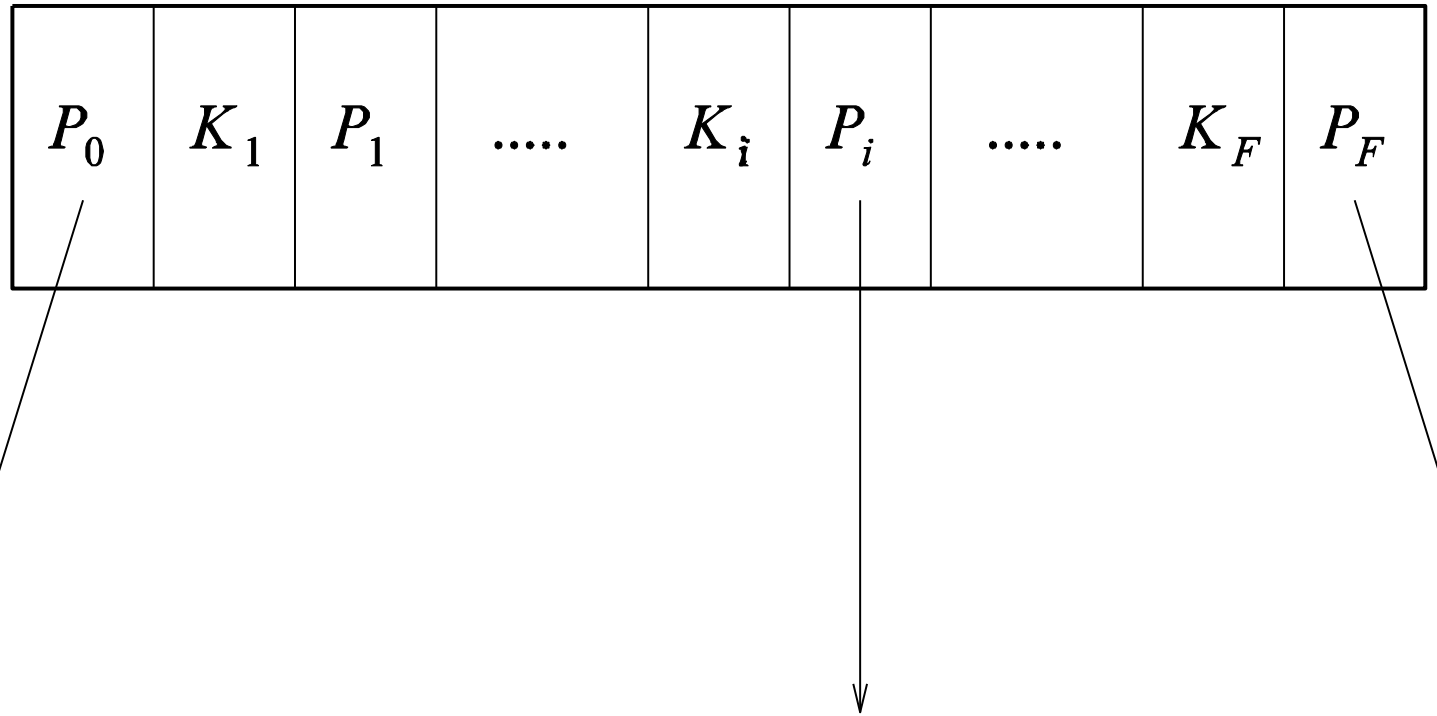
Nodi in un albero di ricerca di ordine F+1



B-tree

- Albero di ricerca in cui ogni nodo corrisponde ad un blocco,
  - viene mantenuto perfettamente bilanciato (tutte le foglie sono allo stesso livello), grazie a:
    - ✓ riempimento parziale (mediamente 70%)
    - ✓ riorganizzazioni (locali) in caso di sbilanciamento

Organizzazione dei nodi del B-tree

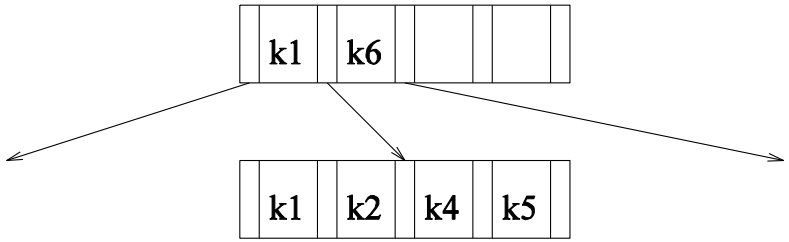


Split e merge

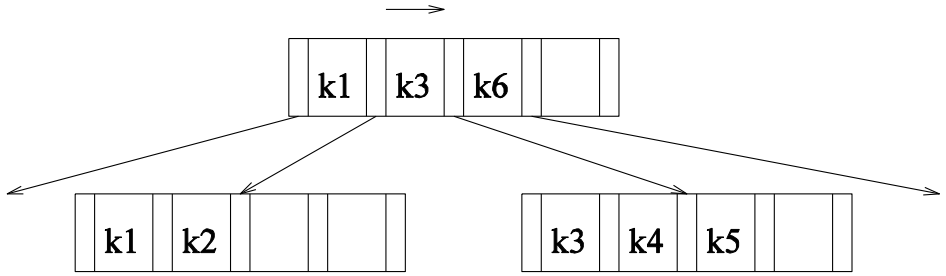
- Inserimenti ed eliminazioni sono precedute da una ricerca fino ad una foglia
- Per gli inserimenti, se c'è posto nella foglia, ok, altrimenti il nodo va suddiviso, con necessità di un puntatore in più per il nodo genitore; se non c'è posto, si sale ancora, eventualmente fino alla radice. Il riempimento rimane sempre superiore al 50%
- Dualmente, le eliminazioni possono portare a riduzioni di nodi
- Modifiche del campo chiave vanno trattate come eliminazioni seguite da inserimenti
- Vedi applet

Split and merge operations

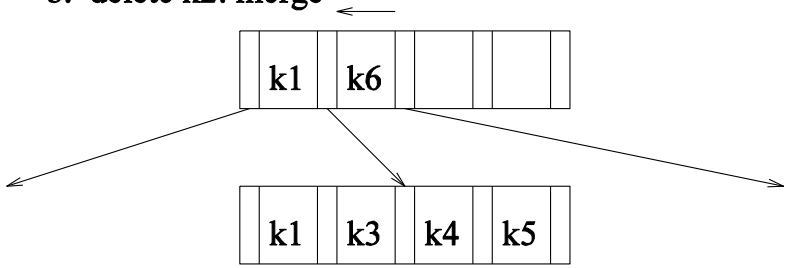
situazione iniziale



a. insert k3: split



b. delete k2: merge



Esempio 1

- Calcolo dell'ordine p di un B-Tree su disco:
  - ✓ Campo di ricerca di V = 9 byte
  - ✓ Dimensione blocchi su disco B = 512 byte
  - ✓ Puntatore a record di Pr = 7byte
  - ✓ Puntatore a blocco di P = 6byte
- Ogni nodo del B-Tree, contenuto in un blocco del disco, può avere al più p puntatori ad albero, p-1 puntatori a record e p-1 valori del campo chiave di ricerca.

Esempio 1 (2)

- Quindi deve essere:

  $(p*P)+((p-1)*(Pr+V)) \leq B \rightarrow (p*6)+((p-1)*(7+9)) \leq 512 \rightarrow (22 * p) \leq 528 \rightarrow p \leq 24$

- Scegliamo p =23, poiché in ogni nodo ci potrebbe essere la necessità di memorizzare informazioni addizionali, quali il numero di entrate q o un puntatore al nodo padre

Esempio 2

- Calcolo del numero di blocchi in un B-Tree
  - Supponiamo di costruire un B-Tree sul campo di ricerca dell'esempio precedente, non-ordering e chiave. Supponiamo che ogni nodo sia pieno per il 69%.
  - Ogni nodo avrà in media p *0.69 = 23 * 0.69 = circa 16 puntatori (fo medio) e quindi 15 valori del campo chiave di ricerca. Partendo dalla radice vediamo quanti valori e puntatori esistono in media a ogni livello:
  - Radice: 1 nodo 15 entry 16 ptr
  - livello 1: 16 nodi 240(16*15) entry 256 (16*16) ptr
  - livello 2: 256 nodi 3840(256*15) entry 4096 ptr.

Esempio 2 (2)

- livello 3: 4096 nodi 61440 (4096 *15) entry

- Per ogni livello:
  - # entrate livello i-mo = [# puntatori livello (i-1)-mo]*15
- Per le date dimensioni di blocco, puntatore e campo chiave di ricerca, un B-tree a 3 livelli contiene fino a 3840 + 240 + 15 = 4095 entrate
- Un B-tree a 4 livelli contiene fino a 4095 + 61440 = 65535 entrate

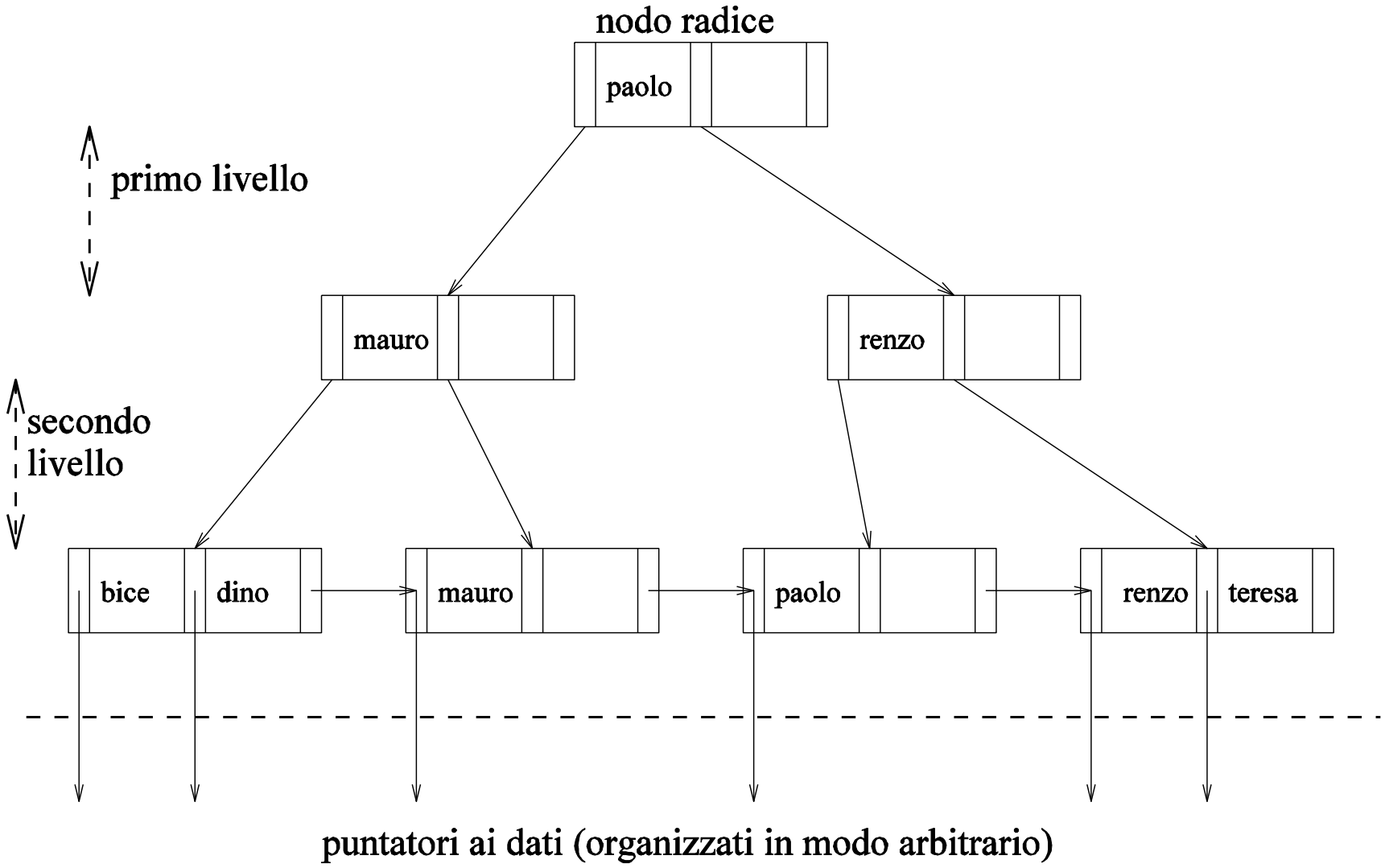
B-tree e B^{+-}tree

- B^{+-}tree:
  - le chiavi compaiono tutte nelle foglie (e quindi quelle nei nodi intermedi sono comunque ripetute nelle foglie)
  - le foglie sono collegate in una lista
  - ottimi per le ricerche su intervalli
  - molto usati nei DBMS
- B-tree:
  - Le chiavi che compaiono nei nodi intermedi non sono ripetute nelle foglie

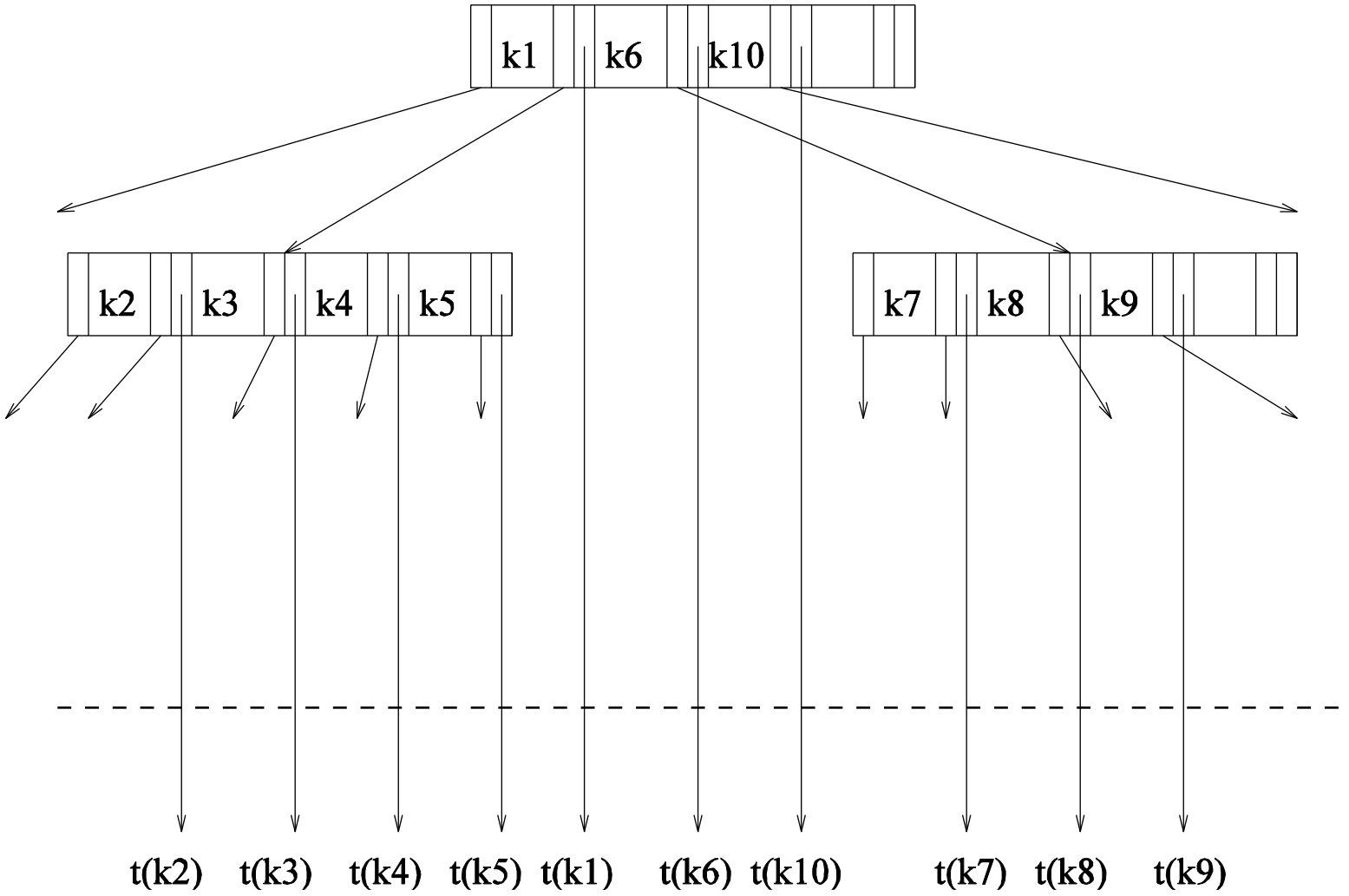
B-tree e B$^+$-tree, primari e secondari

- In un B$^+$-tree
  - primario, le ennuple possono essere contenute nelle foglie
  - secondario, le foglie contengono puntatori alle ennuple
- In un B-tree
  - anche i nodi intermedi contengono ennuple (se primari) o puntatori (se secondari)

Un B$^+$-tree


nodo radice
paolo
primo livello
mauro
renzo
secondo livello
bice
dino
mauro
paolo
renzo
teresa
puntatori ai dati (organizzati in modo arbitrario)

Un B-tree


k1
k6
k10
k2
k3
k4
k5
k7
k8
k9
t(k2)
t(k3)
t(k4)
t(k5)
t(k1)
t(k6)
t(k10)
t(k7)
t(k8)
t(k9)

Esempio

- Calcolo dell'ordine p di un B^+-Tree su disco:
- Campo di ricerca di V = 9 byte
- Dimensione blocchi su disco B=512 byte
- Puntatore a record di Pr = 7 byte
- Puntatore a blocco di P = 6 byte
- Un nodo interno di un B^+-Tree, contenuto in un singolo blocco, può avere fino a p puntatori ad albero e p-1 valori del campo di ricerca. Quindi deve essere:

  $(p*P)+((p-1)*V) \leq B \rightarrow (p*6)+((p-1)*9) \leq 512 \rightarrow (15 * p) \leq 521$

Esempio (2)

- Il massimo intero che soddisfa la disuguaglianza è p=34, che è maggiore di 23 (ordine del B-Tree corrispondente). Quindi ne risulta un fan-out maggiore e più entrate in ogni nodo interno.
- L'ordine P_{leaf} dei nodi foglia è:

  $(P_{leaf} * (Pr + V)) + P$ (per memorizzare P_{next}) $\leq B \rightarrow$

  $(P_{leaf} * (7+9)) + 6 \leq 512 \rightarrow$

  $(P_{leaf} * 16) \leq 506$
- Quindi ogni nodo foglia può tenere fino a $P_{leaf} = 31$ combinazioni valore chiave/data pointer, supponendo che i data pointer siano puntatori a record.

Esempio 2

- Calcolo del numero di blocchi e di livelli in un B^+-Tree
  - Supponiamo che ogni nodo del B^+-Tree sia pieno per il 69%.
  - Ogni nodo interno avrà in media p * 0.69 = 34 * 0.69 = circa 23 puntatori (fan-out medio) e quindi 22 valori del campo di ricerca. Ogni nodo foglia avrà in media P_{leaf}*0.69 = 31*0.69 = circa 21 puntatori a record dati.

Esempio 2 (2)

- Partendo dalla radice vediamo quanti valori e puntatori esistono in media a ogni livello:

Radice : 1 nodo 22 entry 23 ptr

Livello 1: 23 nodi 506 (22*23) entry 529 (23*23) ptr

Livello 2: 529 nodi 11638 (22*529) entry 12167 ptr

Livello foglie: 12167 nodi 255507 (21*12167) puntatori a record.

- Quindi un B^+-Tree di quattro livelli tiene fino a 255.507 puntatori a record, contro i 65.535 calcolati per il B-Tree corrispondente.

Inserimento in un B^+-Tree

- Inizialmente la radice è l'unico nodo dell'albero: essendo quindi un nodo foglia conterrà anche i puntatori ai dati
- Se si cerca di inserire un'entry in un nodo foglia pieno, il nodo va in overflow e deve essere scisso:
  - le prime $j = \lceil (p_{leaf} + 1)/2 \rceil$ entry sono mantenute nel nodo, mentre le rimanenti sono spostate in un nuovo nodo foglia.
  - Il j-mo valore di ricerca (quello mediano) è replicato nel nodo padre
  - Nel padre viene creato un puntatore al nuovo nodo.

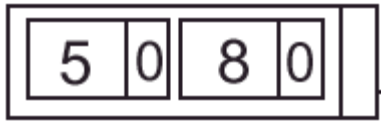
Inserimento in un B^+-Tree (2)

- Se il nodo padre (interno) è pieno si ha un altro overflow:
  - Il nodo viene diviso, creando un nuovo nodo interno.
  - Le entrate nel nodo interno fino a P_j (con $j = \lfloor((p+1)/2)\rfloor$) sono mantenute nel nodo originario, mentre il j-mo valore di ricerca è spostato al padre, non replicato.
  - Il nuovo nodo interno conterrà le entrate dalla P_{j+1} alla fine delle entrate del nodo originario.
- Tale scissione si può propagare verso l'alto fino a creare un nuovo livello per il B^+-Tree.

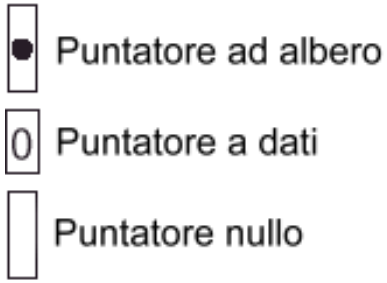
Esempio di inserimento

- Vogliamo inserire dei record in un

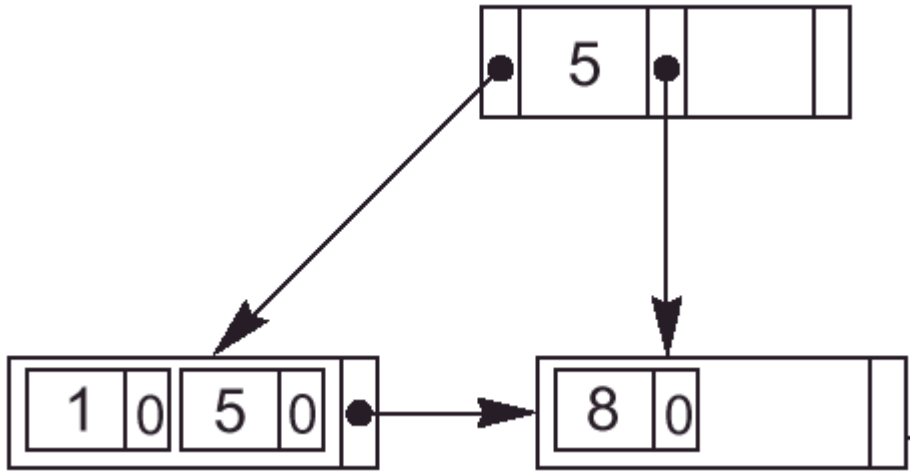
  B^+-Tree di ordine p = 3 e P_{leaf} = 2, nella sequenza 8, 5, 1, 7, 3, 12.



L'inserimento dei valori 8 e 5 non provoca overflow: sono entrambi inseriti nella radice



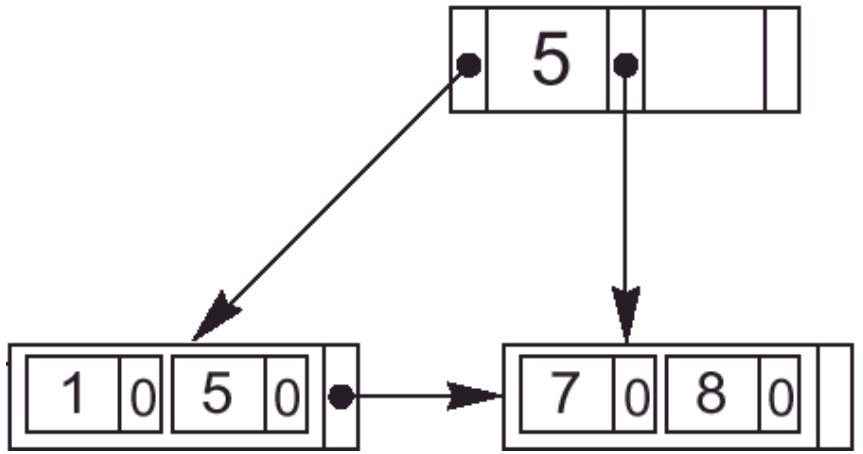
Esempio di inserimento (2)



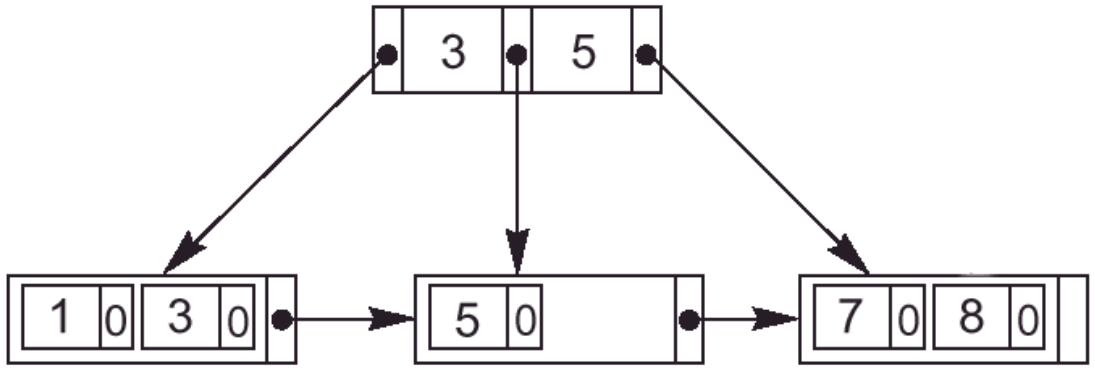
L'inserimento di 1 provoca overflow. Il nodo è scisso ed il valore mediano è ripetuto in un nuovo nodo radice

L'inserimento di 7 non provoca overflow.

Si noti che tutti i valori sono a livello foglia, perché i data pointer sono tutti a quel livello. Alcuni sono replicati nei nodi interni per guidare la ricerca.



Esempio di inserimento (3)

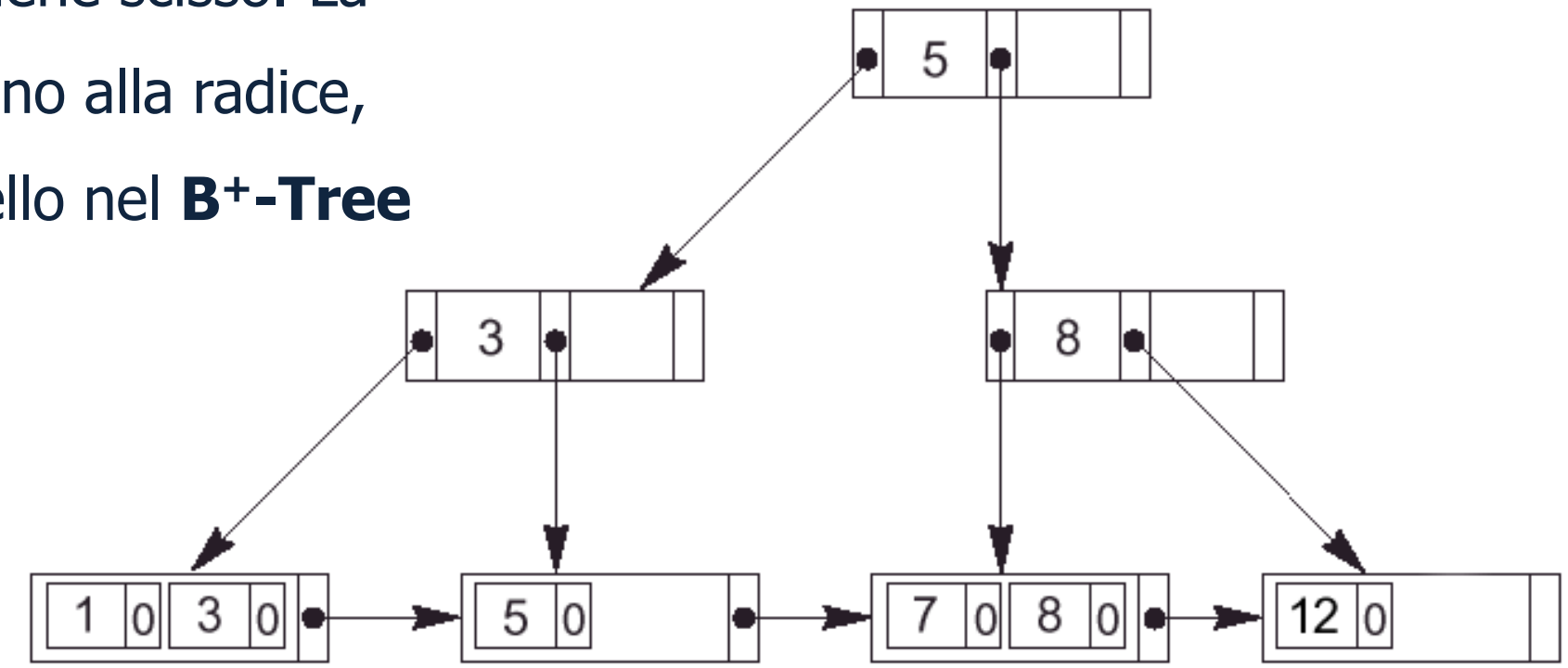


L'inserimento di 3 provoca un overflow nel nodo (1, 5), che viene scisso.

Si noti che un valore che compare nel nodo interno, compare anche come valore più a destra del sottoalbero referenziato dal puntatore alla sinistra di tale valore

Esempio di inserimento (4)

L'inserimento di 12 provoca un overflow nel nodo (7, 8), che viene scisso. La scissione si propaga fino alla radice, creando un nuovo livello nel **B$^+$-Tree**



Cancellazione in un B^+-Tree

- Un'entry è cancellata sempre a livello foglie. Se essa ricorre anche in un nodo interno, allora è rimossa anche da quello e al suo posto si inserisce il valore immediatamente alla sinistra del valore rimosso nel nodo foglia.
- La cancellazione di valori può causare l'**underflow** di un nodo, riducendo il numero di entry in un nodo foglia per meno del minimo consentito.
  - In tal caso si cerca un fratello del nodo foglia in modo da ridistribuire le entrate, cosicché entrambi i nodi siano pieni almeno per meta;

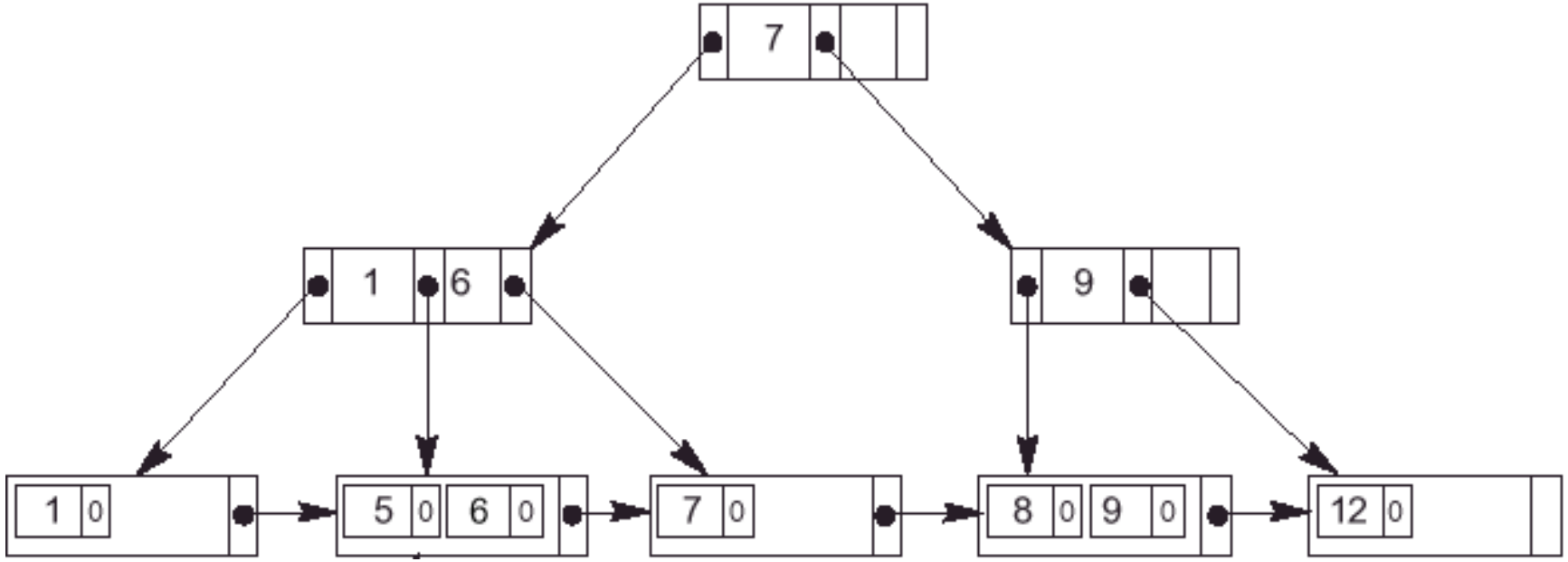
Cancellazione in un B^+-Tree (2)

- Se ciò non è possibile si effettua il merge del nodo con i suoi fratelli, riducendo di uno il numero di nodi foglia.

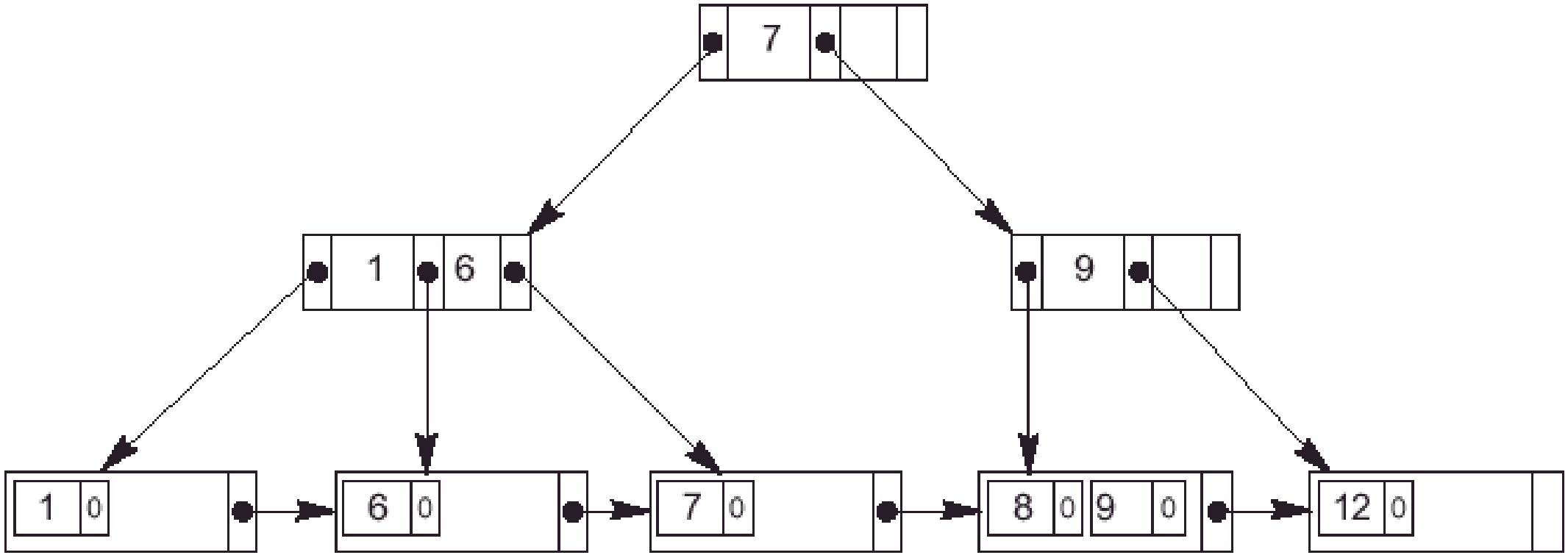
- Approccio comune:
  - Si tenta di ridistribuire le entrate con il fratello sinistro;
  - Se non è possibile, si tenta di ridistribuire con il fratello destro
  - Altrimenti si fondono i tre nodi in due nodi foglia. L'underflow in questo caso si può trasmettere ai nodi interni, poiché sono necessari un puntatore ad albero ed un valore di ricerca in meno. Se la propagazione arriva alla radice, si riduce il numero di livelli dell'albero.

Esempio di cancellazione

- Dato il seguente B^+-Tree, vogliamo cancellare i record 5, 12 e 9.

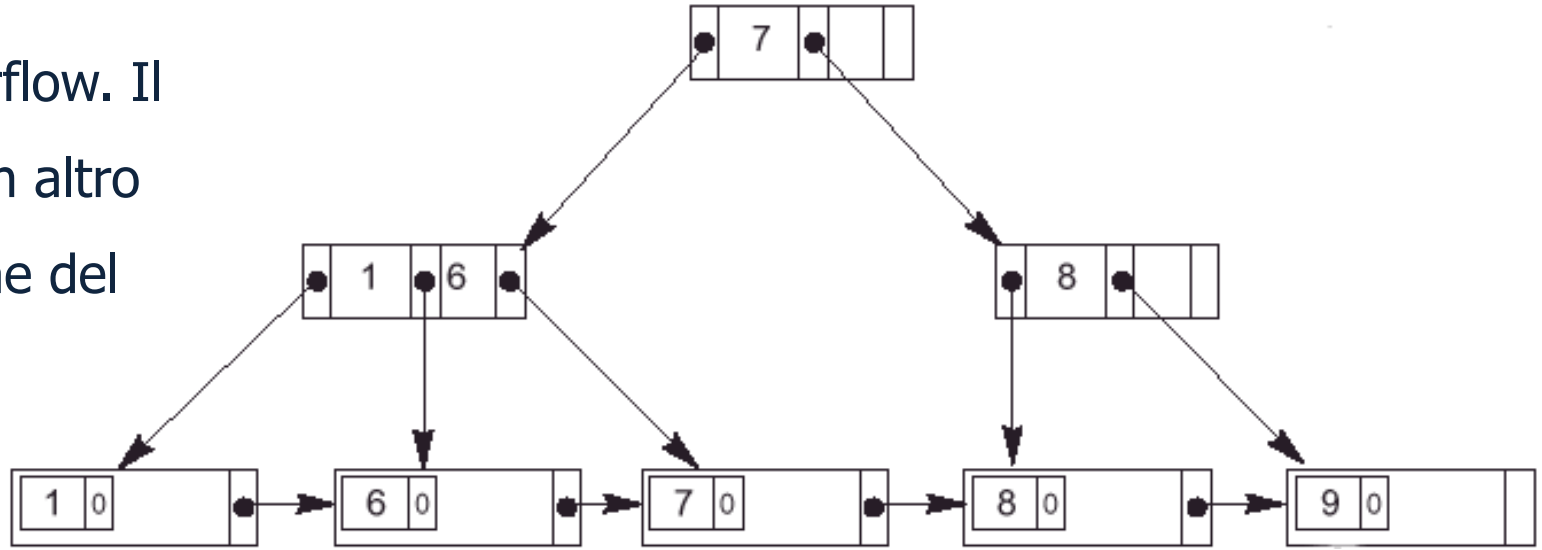


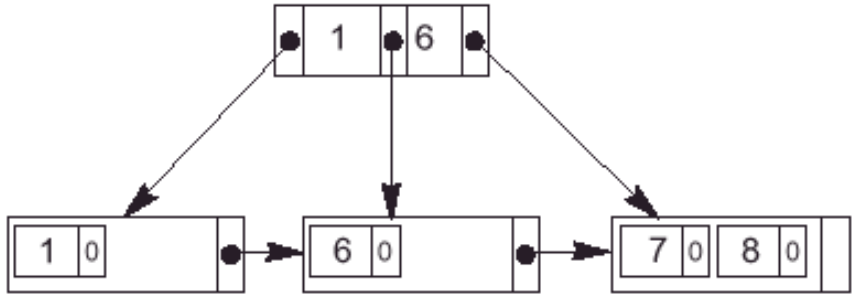
Esempio di cancellazione (2)



Esempio di cancellazione (3)

La cancellazione di 9 causa un underflow. Il merge con il fratello sinistro causa un altro underflow, che porta ad una riduzione del numero di livelli





Il B^+-Tree risultante

Progettazione fisica

- La fase finale del processo di progettazione di basi di dati
- input
  - lo schema logico e informazioni sul carico applicativo
- output
  - schema fisico, costituito dalle definizione delle relazioni con le relative strutture fisiche (e molti parametri, spesso legati allo specifico DBMS)

Strutture fisiche nei DBMS relazionali

- Struttura primaria:
  - disordinata (heap, "unclustered")
  - ordinata ("clustered"), anche su una pseudochiave
  - hash ("clustered"), anche su una pseudochiave, senza ordinamento
  - clustering di più relazioni
- Indici (densi/sparsi, semplici/composti):
  - ISAM (statico), di solito su struttura ordinata
  - B-tree (dinamico)
  - Indici hash (secondario, poco dinamico)

Strutture fisiche in alcuni DBMS

- Oracle:
  - struttura primaria
    - ✓ file heap
    - ✓ "hash cluster" (cioè struttura hash)
    - ✓ cluster (anche plurirelazionali) anche ordinati (con B-tree denso)
- indici secondari di vario tipo (B-tree, bit-map, funzioni)
- DB2:
  - primaria: heap o ordinata con B-tree denso
  - indice sulla chiave primaria (automaticamente)
  - indici secondari B-tree densi
- SQL Server:
  - primaria: heap o ordinata con indice B-tree sparso
  - indici secondari B-tree densi

Strutture fisiche in alcuni DBMS, 2

- Ingres (anni fa):
  - file heap, hash, ISAM (ciascuno anche compresso)
  - indici secondari
- Informix (per DOS, 1994):
  - file heap
  - indici secondari (e primari [cluster] ma non mantenuti)

Definizione degli indici SQL

- Non è standard, ma presente in forma simile nei vari DBMS
  - `create [unique] index` *IndexName* `on` *TableName* (*AttributeList*)
  - `drop index` *IndexName*

Progettazione fisica nel modello relazionale

- La caratteristica comune dei DBMS relazionali è la disponibilità degli indici:
  - la progettazione fisica spesso coincide con la scelta degli indici (oltre ai parametri strettamente dipendenti dal DBMS)
- Le chiavi (primarie) delle relazioni sono di solito coinvolte in selezioni e join: molti sistemi prevedono (oppure suggeriscono) di definire indici sulle chiavi primarie
- Altri indici vengono definiti con riferimento ad altre selezioni o join "importanti"
- Se le prestazioni sono insoddisfacenti, si "tara" il sistema aggiungendo o eliminando indici
- È utile verificare se e come gli indici sono utilizzati con il comando SQL `show plan` oppure `explain`