

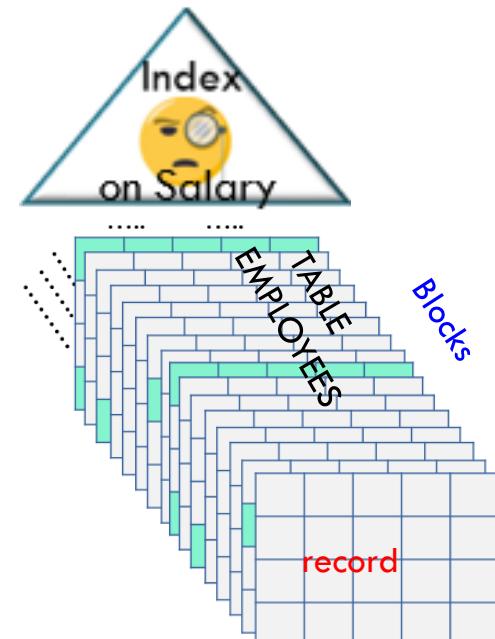
Indici

Annalisa Franco, Dario Maio  
Università di Bologna

# Premessa

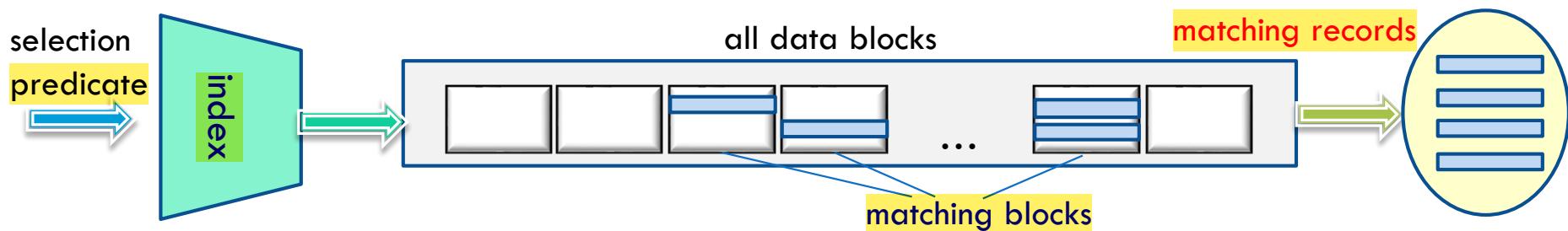
- Un indice, nel contesto dei database, è definito come una struttura dati che rende disponibile un “**cammino d’accesso**” atto a localizzare efficientemente i record d’interesse nel rispetto di un criterio di selezione.

```
SELECT E.EmpNo, E.FirstName, E.LastName, E.Job  
FROM EMPLOYEES E  
WHERE E.Salary BETWEEN 30000 AND 40000  
ORDER BY E.Salary
```



# Introduzione agli indici

- Un indice può essere definito come una struttura progettata per ottimizzare le ricerche di record che soddisfano un certo predicato di selezione.



- Un indice concettualmente può essere visto come una mappa che memorizza entrate del tipo: [valore di chiave di ricerca, riferimento/i al/ai record].
- Sono di rilevante interesse due grandi famiglie di indici:
  - **ordered index**: i valori di chiave sono mantenuti ordinati; questi indici possono essere a uno o più livelli (secondo una struttura ad albero a più vie, es.  $B^+$ -tree); Valori di Chiave hanno i riferimenti ai record
  - **hash index**: i valori di chiave, con i relativi riferimenti ai record, sono memorizzati in bucket i cui indirizzi sono generati da una funzione hash.

Un hash index utilizza una funzione hash per calcolare l'indirizzo del bucket (dando come input la chiave) in cui memorizzare o cercare i valori di chiave e i relativi riferimenti ai record. Questo tipo di indice è particolarmente efficiente per le ricerche di uguaglianza.

# Ordered index e hash index

- Un indice ordinato, se contiene al proprio interno i record, funge da organizzazione primaria; un esempio classico è rappresentato da un B<sup>+</sup>-tree che memorizza nelle foglie i record ordinati per valore della chiave primaria. In un certo senso si può affermare che, in questi casi, l'indice stesso costituisce l'organizzazione primaria.
- Un indice ordinato rappresenta un'organizzazione secondaria (cioè aggiuntiva) se è, invece, un ulteriore cammino d'accesso per accedere all'organizzazione primaria del file dati (che può essere realizzato, ad esempio, come heap, sorted sequential file, struttura hash o anche ad albero) per facilitare ricerche su singolo attributo o combinazione di attributi (in genere diverso/a dalla chiave primaria) con eventuali valori ripetuti.
- Si ricorda che anche un hash index è una struttura secondaria mentre un hash file, ovvero un file ove i dati sono allocati tramite hashing, è una organizzazione primaria.

# Hash index

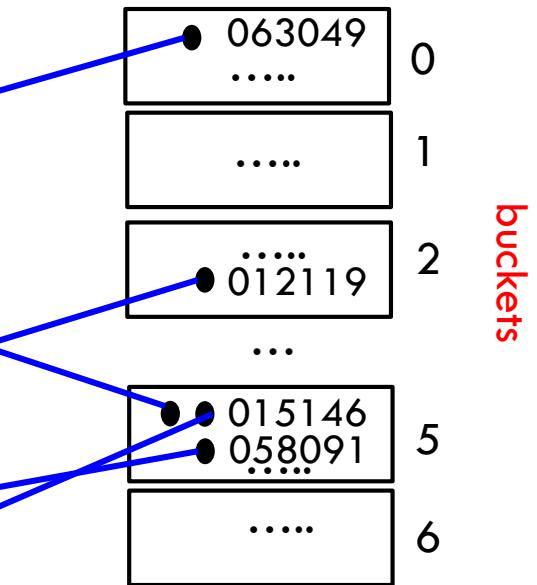
Supponiamo di voler trovare il cliente con l'email luigi@example.com. Il processo sarebbe il seguente:  
Applicazione della Funzione Hash: Il database applica la funzione hash a luigi@example.com, ottenendo il valore hash 5.  
Accesso al Bucket: Il database accede al bucket 5 nella struttura hash.  
Recupero del Record: All'interno del bucket, il database trova il riferimento al record corrispondente e recupera i dati effettivi dal file dati.

- Le entry [valore di chiave di ricerca, riferimento/i al/ai record] sono organizzate in una struttura hash.
- È una organizzazione secondaria molto efficiente per ricerche su singolo valore.

$$H(k) = k \bmod Z \quad \text{Numero di Bucket}$$

CodImp	Cognome	Nome	<u>CodComune</u>
...	...	...	
A017	Grigi	Marco	063049
...	...	...	
A221	Verdi	Giorgio	015146
...	...	...	
C100	Rossi	Anna	012119
...	...	...	
H022	Rossi	Maria	058091
H023	Violetti	Franco	015146
...	...	...	

Hash index su CodComune



# Ordered index: idea di base

- L'idea alla base di un indice ordinato consiste nell'associare al file dati una "tabella" in cui l'entrata i-esima memorizza una coppia del tipo  $(k_i, p_i)$  dove:
  - $k_i$  : è un **valore di chiave** dell'attributo su cui l'indice è costruito;
  - $p_i$  : è un "riferimento" al record (eventualmente il solo) con valore di chiave  $k_i$ .
- Il riferimento (puntatore) può essere un indirizzo (logico o fisico) di record o di blocco.
- Il vantaggio di usare un indice nasce dal fatto che la chiave è solo parte dell'informazione contenuta in un record. Pertanto, l'indice occupa un spazio minore rispetto al file dati (questa considerazione vale anche per gli indici hash).
- □ I diversi indici ordinati differiscono essenzialmente nel modo con cui organizzano l'insieme  $\{(k_i, p_i)\}$  di coppie, cioè differisce la struttura dati dove vengono memorizzate le coppie  $k_i, p_i$

File Dati      **valore di chiave  
nel record**      (caso di valori non ripetuti)

C5 .....	C2 .....	C11 .....	C7 .....	C4 .....
0	64	128	192	256

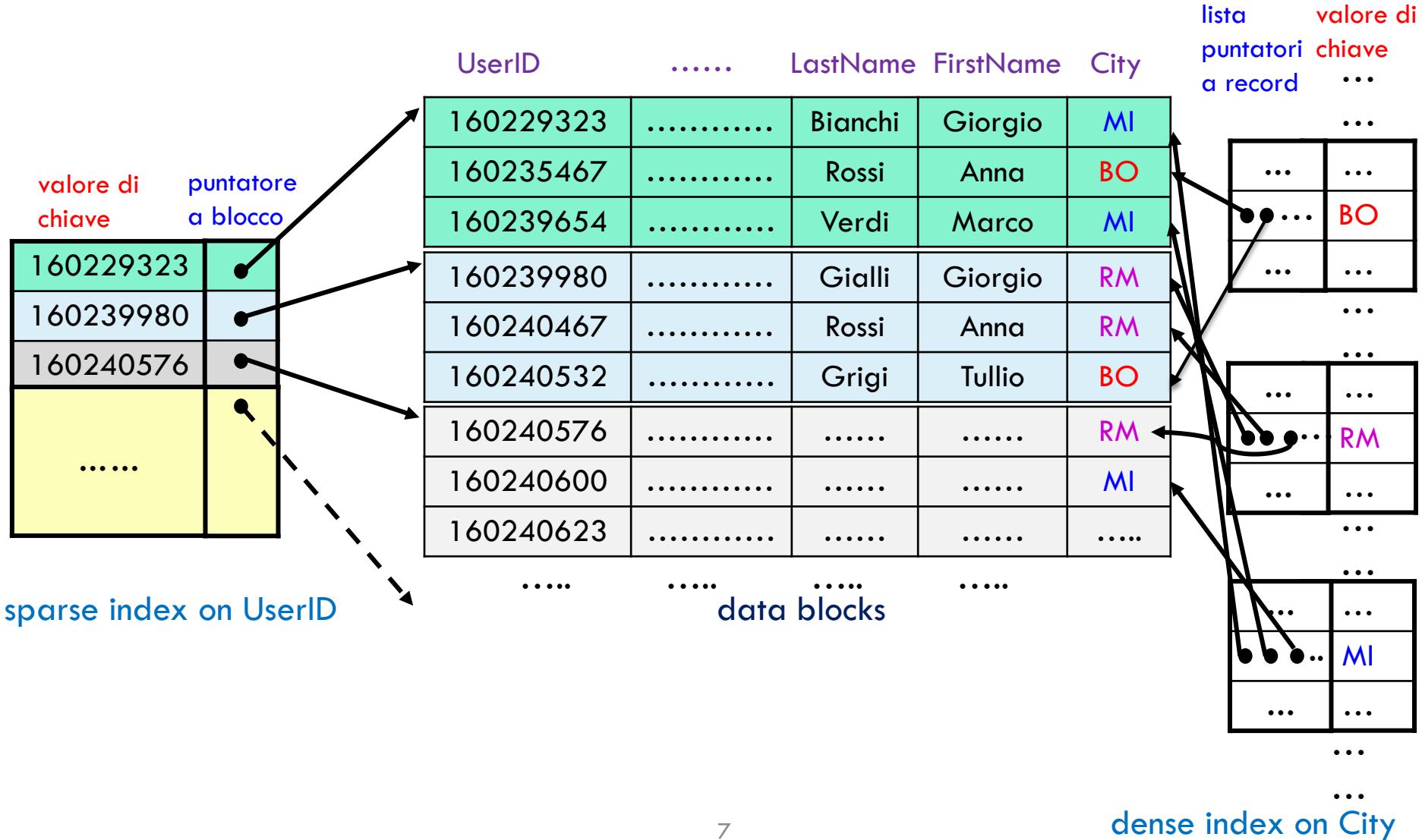
Attributo/i su cui l'indice viene costruito

chiave      indirizzo      Riferimento al record

C2	64	Indice ordinato
C4	256	
C5	0	
C7	192	
C11	128	

# Indici mono-livello: esempio

Un indice mono-livello è un tipo di indice in cui tutti i puntatori ai record sono contenuti in un'unica struttura d'indice. Non ci sono livelli intermedi di indici, e l'intero indice è organizzato in un'unica lista o tabella che mappa i valori delle chiavi agli indirizzi dei record nel file dati.



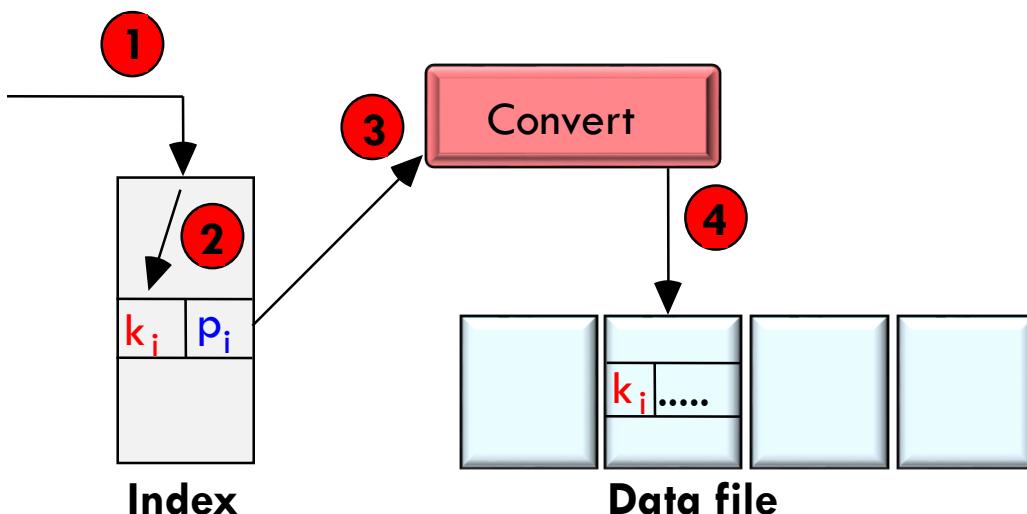
# Accesso con indice mono-livello

Questo indice mono-livello sarebbe considerato un'organizzazione secondaria.

Si consideri un indice mono-livello su chiave primaria (o altro attributo a valori unici)

ricerca del record con chiave  $k_i$

1. accesso all'indice
2. ricerca della coppia  $(k_i, p_i)$
3. conversione di  $p_i$  in indirizzo assoluto
4. accesso al blocco dati relativo



Organizzazione primaria: Se l'indice su chiave primaria contiene tutti i record o se l'ordine dei record nel file dati è determinato dalla chiave primaria (cioè, i dati stessi sono ordinati e organizzati in base alla chiave primaria), allora l'indice è parte dell'organizzazione primaria.

Nonostante occupi un minor spazio rispetto al file dati, un indice può anche raggiungere notevoli dimensioni che determinano problemi di gestione simili a quelli del file dati.

Esempio:

un indice per un file di 50K record, in cui i valori di chiave sono stringhe di 20 byte e i puntatori sono di 4 byte, richiede circa 1.2 MB.

Organizzazione secondaria: Se l'indice su chiave primaria contiene solo la chiave primaria e i riferimenti ai record senza determinare l'ordine fisico dei dati, allora si tratta di un'organizzazione secondaria. In questo caso, l'indice non è l'unica struttura per accedere ai dati, ma solo un metodo di accesso aggiuntivo.

# Ricerca binaria su indice mono-livello

- Poiché l'indice contiene un insieme di valori di chiave, le coppie  $(k_i, p_i)$  possono essere mantenute ordinate in base ai valori  $k_i$ , al fine di poter applicare la ricerca binaria.
- In generale, questa tecnica permette risparmi tanto più marcati quanto minore è la dimensione (in byte) del campo chiave rispetto a quella del record intero.

# Tipi di indice ordinato: una classificazione

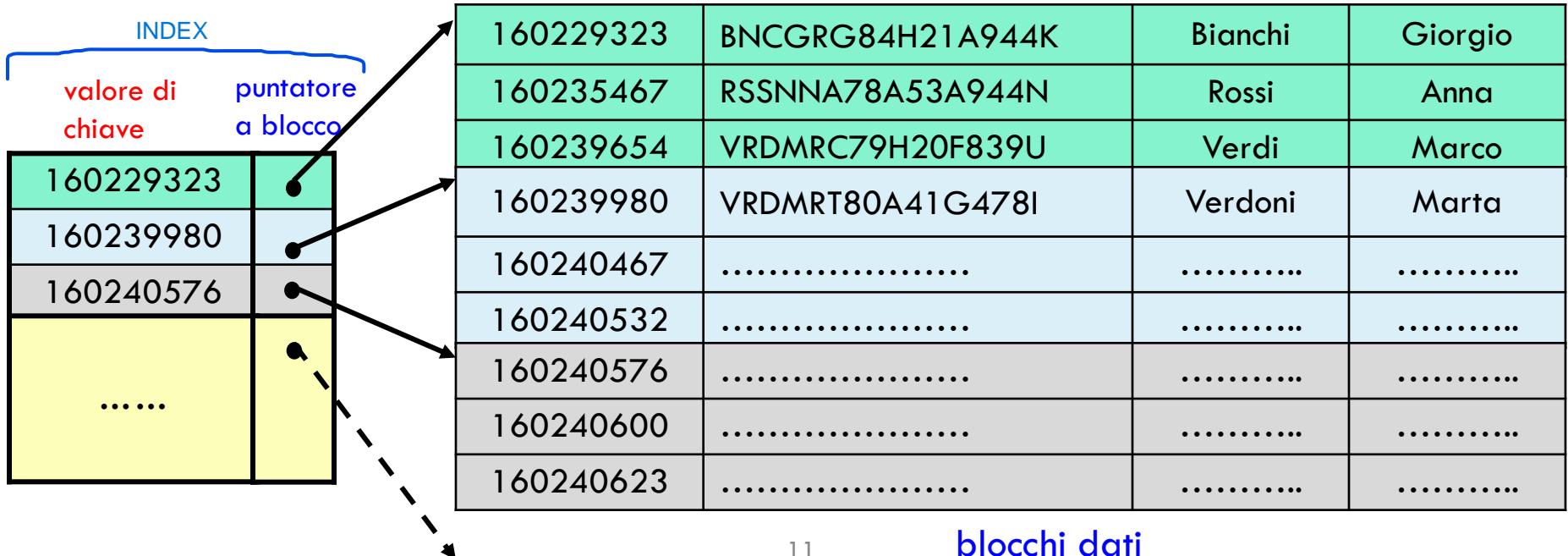
Caratteristica	Denominazione	Significato
Unicità dei valori di chiave	Primary (unique) index	Indice su un attributo (o combinazione di attributi) che assume valori unici (non ripetuti)
	Secondary index	Indice su un attributo (o combinazione di attributi) che può assumere valori ripetuti
Ordinamento del file dati	Clustered index	Indice su un attributo (o combinazione di attributi) secondo cui il file dati è ordinato
	Unclustered index	Indice su un attributo (o combinazione di attributi) secondo cui il file dati non è ordinato
Numero di coppie nell'indice	Dense index	Indice in cui il numero di coppie ( $k_i, p_i$ ) è pari al numero di record dati
	Sparse index	Indice in cui il numero di coppie ( $k_i, p_i$ ) è minore del numero di record dati
Numero di livelli dell'indice	Single-level index	Indice organizzato in modo “flat”
	Multi-level index	Indice organizzato in più livelli (albero)

# Primary clustered sparse single-level index

Quasi tutte le combinazioni di tipi di indice sono possibili.

In principio, l'unica incompatibilità è data dalla combinazione **sparse & unclustered** che non permetterebbe di reperire i record i cui relativi riferimenti non sono nell'indice.

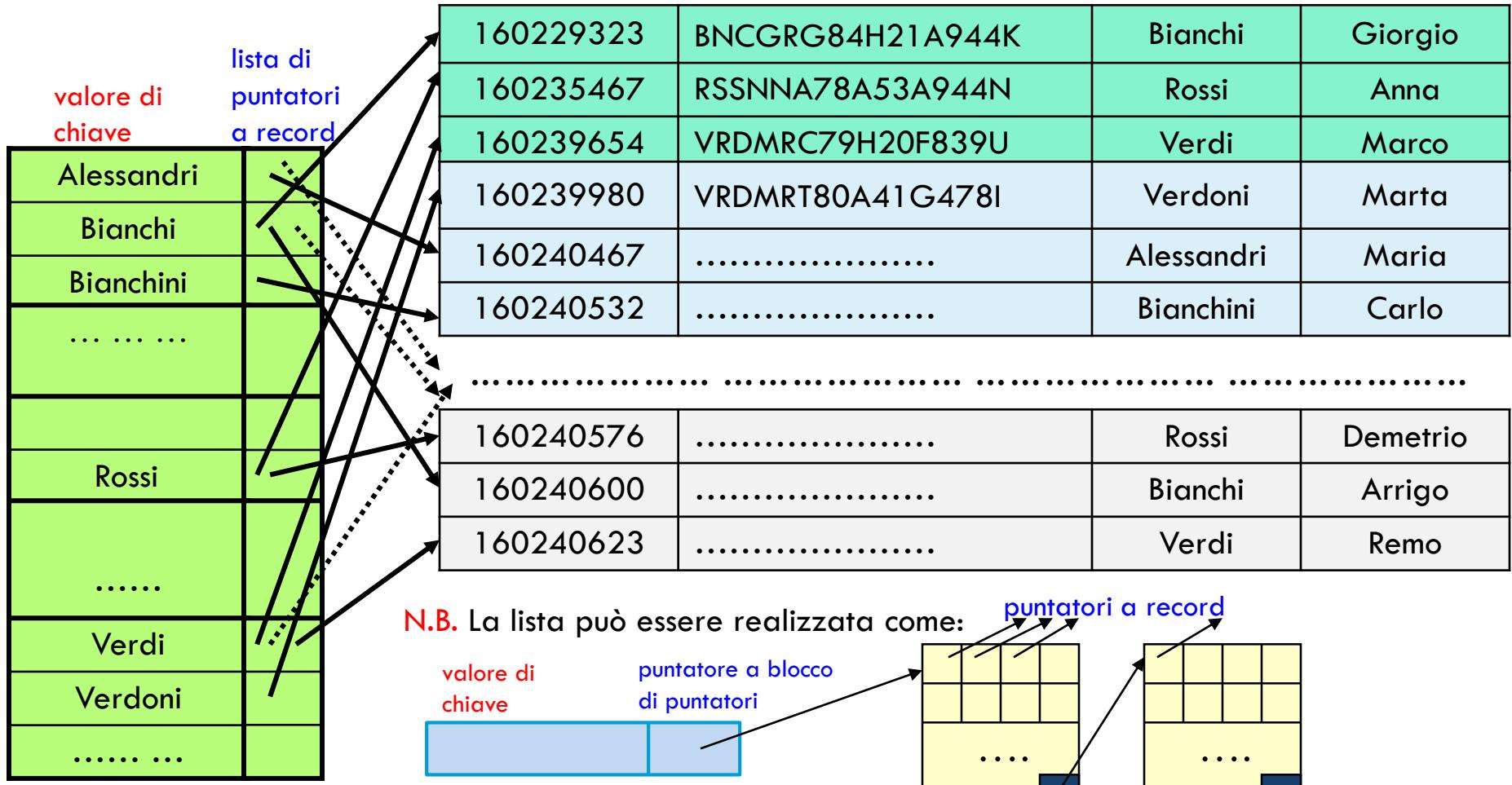
primary clustered sparse single-level index



# Secondary unclustered dense single-level index

secondary unclustered dense single-level index

blocchi dati



# Sparse & unclustered

- Una soluzione sparse & unclustered particolare, nota col nome di **partial index**, per un **secondary index**, può essere interessante se si escludono dall'indice i valori di chiave molto ripetuti, per i quali l'uso dell'indice stesso può rivelarsi inutile se non controproducente.
- Si pensi ad esempio al log degli accessi al web server di una grande organizzazione.
- Esempio in PostgreSQL:

ES

```
CREATE TABLE ACCESS_LOG ( Url varchar, ClientIP inet, ... )
```

- Se si è interessati principalmente a monitorare gli accessi da IP al di fuori dell'organizzazione, poiché la stragrande maggioranza degli accessi è originata da IP interni all'organizzazione stessa, non è conveniente costruire un indice sull'attributo ClientIP che riporti tutti i suoi possibili valori.

```
CREATE INDEX AccessLogClientIP_IX ON ACCESS_LOG (ClientIP) WHERE NOT  
(ClientIP > inet '192.168.100.0' AND ClientIP < inet '192.168.100.255');
```

- Un altro esempio riguarda un unclustered secondary index su un attributo A che ammette NULL, se il DMBS non indicizza valori di chiave NULL; in tal caso di fatto è uno sparse & unclustered index.

# Record Pointer vs Block Pointer

SUPPNO	PTNO	PTYPE	SHELF
S2	P4	SCREW	A1
S6	P7	NUT	A1
S3	P1	SCREW	A1
S2	P1	SCREW	A1
S6	P2	NUT	A1
S1	P3	BOLT	A2
S5	P11	CAM	A2
S3	P3	NUT	A2
S5	P12	CAM	A2
S5	P3	BOLT	A3
S3	P5	BOLT	A3
S4	P11	CAM	A4

1,1  
 1,2  
 1,3  
 1,4 → N° blocco  
 2,1 → N° record  
 nel blocco  
 2,2  
 2,3  
 2,4  
 3,1  
 3,2  
 3,3  
 3,4

indice su PTYPE con puntatori (RID) a record

BOLT (2,2) (3,2) (3,3) CAM (2,3) (3,1) (3,4) NUT (1,2) (2,1) (2,4) SCREW (1,1) (1,3) (1,4)

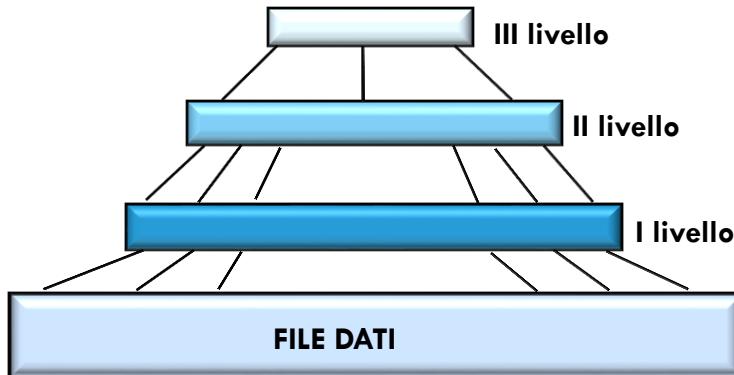
indice su PTYPE con puntatori a blocchi

BOLT (2) (3) CAM (2) (3) NUT (1) (2) SCREW (1)

# Indici multilivello

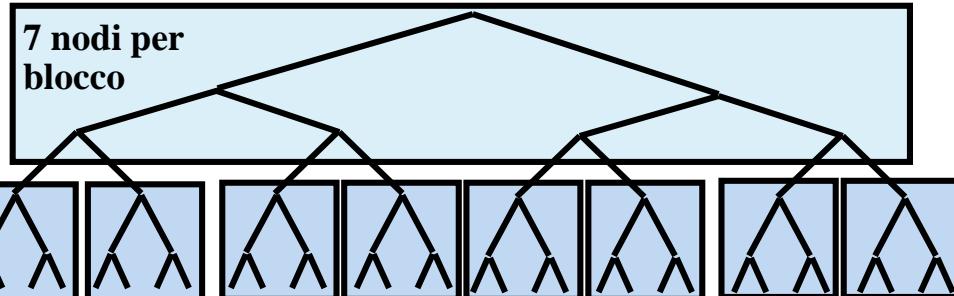
Gli indici multi-livello sono progettati per migliorare l'efficienza delle operazioni di ricerca, inserimento e cancellazione su memoria secondaria, dove i dati non possono essere completamente caricati in memoria centrale

Per ragioni di efficienza un indice in memoria secondaria è, di solito, organizzato in più livelli.



In memoria centrale la soluzione più comune per un indice si basa su AVL (alberi binari bilanciati). La “paginazione” di alberi non è adeguata dal punto di vista dinamico, in quanto:

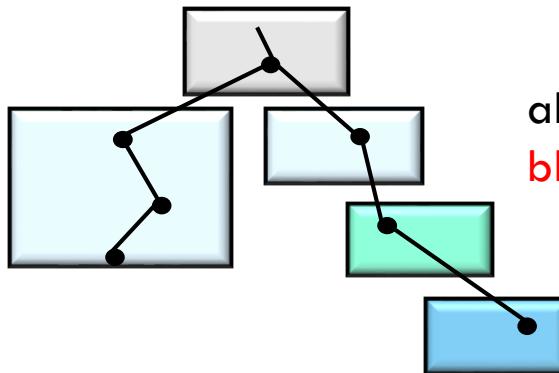
- le operazioni di bilanciamento degli AVL non tengono conto di un'organizzazione a blocchi; pertanto i costi di I/O dovuti a inserimenti e cancellazioni possono risultare elevati;
- volendo ridurre i costi, non si hanno garanzie sull'utilizzazione minima dei blocchi allocati.



# Indici multilivello a blocchi

Un indice multilivello per memoria secondaria deve soddisfare i seguenti requisiti:

- ❑ **Bilanciamento:** l'indice deve essere sì bilanciato, ma considerando i blocchi anziché i singoli nodi, in quanto è il numero di blocchi a cui bisogna accedere che determina il costo di I/O di una ricerca.



albero sbilanciato rispetto ai blocchi, bilanciato rispetto ai nodi

- ❑ **Occupazione minima:** è importante che si possa stabilire un limite inferiore all'utilizzazione dei blocchi, onde evitare eccessivo spreco di memoria.
- ❑ **Efficienza di aggiornamento:** i due requisiti espressi devono essere soddisfatti garantendo al tempo stesso che le operazioni di aggiornamento abbiano un costo limitato.

# B-tree (Bayer, McCreight 1972)

- Una famiglia di indici multilivello che soddisfa i tre requisiti (**bilanciamento a blocchi**, **occupazione minima**, **efficienza di aggiornamento**) è collettivamente nota con il nome di **B-tree**, dove la **B** sta per (?):

- **Balanced tree**,
- **Bayer**, inventore insieme a McCreight
- **Boeing**, la compagnia per la quale gli autori lavoravano
- **Broad** Rudolf Bayer, Edward Meyers McCreight  
Organization and Maintenance of Large Ordered Indexes  
Acta Informatica 1, 173-189 0972)
- **Bushy**

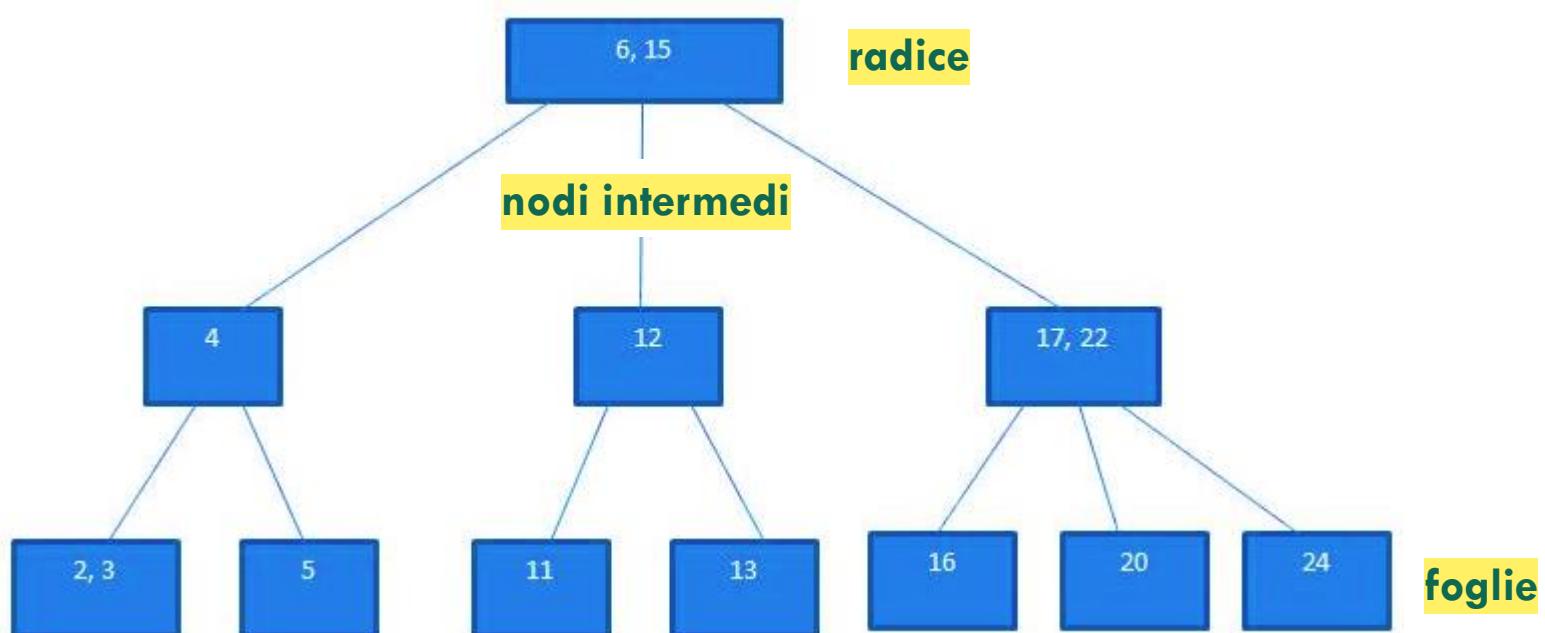
*“What really lives to say is: the more you think about what the B in B-trees means, the better you understand B-trees.”* (E. McCreight, 2013)

- Esistono molte varianti, tra cui il **B-tree** “vero e proprio”, il **B\*-tree** e il **B<sup>+</sup>-tree**.

# B-tree

- 1) Un B-tree è un albero bilanciato, il che significa che tutte le foglie dell'albero si trovano allo stesso livello.
- 2) In un B-tree, ogni nodo dell'albero corrisponde a un blocco di memoria (o pagina) nello storage. Questi nodi contengono chiavi e puntatori. Ogni blocco di memoria può contenere più chiavi e puntatori a figli.
- 3) I nodi interni dell'albero contengono chiavi che servono come separatori per determinare a quale sotto-albero (o blocco) devono essere indirizzati i dati. Ogni nodo interno ha tra  $m/2$  e  $m$  figli (eccetto la radice).
- 4) Le foglie contengono le chiavi e i puntatori ai record effettivi memorizzati nel file di dati. Le foglie sono collegate tra loro in una lista ordinata, facilitando le operazioni di scansione e intervallo.

Un B-tree è un albero (direzionato) a più vie perfettamente bilanciato organizzato a nodi, che corrispondono a blocchi dati (pagine) di uno storage device.



# B-tree: definizione

- Siano  $g, h > 0$  due numeri naturali, detti rispettivamente **ordine** e **altezza** del B-tree. L'ordine  $g$  corrisponde al numero minimo di chiavi in un nodo non radice.
- Un B-tree  $T$  della classe  $\tau(g,h)$  ha le seguenti proprietà:
  1. ogni percorso dalla radice a una foglia ha sempre la stessa lunghezza  $h$ , chiamata altezza del B-tree ( $h = \text{numero nodi nel percorso}$ );
  2. ogni nodo, a eccezione della radice e delle foglie, ha almeno  $g+1$  figli. La radice o è una foglia ( $h = 1$ ) o ha almeno  $2$  figli; le foglie non hanno figli.
  3. la radice e ogni nodo intermedio hanno al più  $2g+1$  nodi figli.

MAX

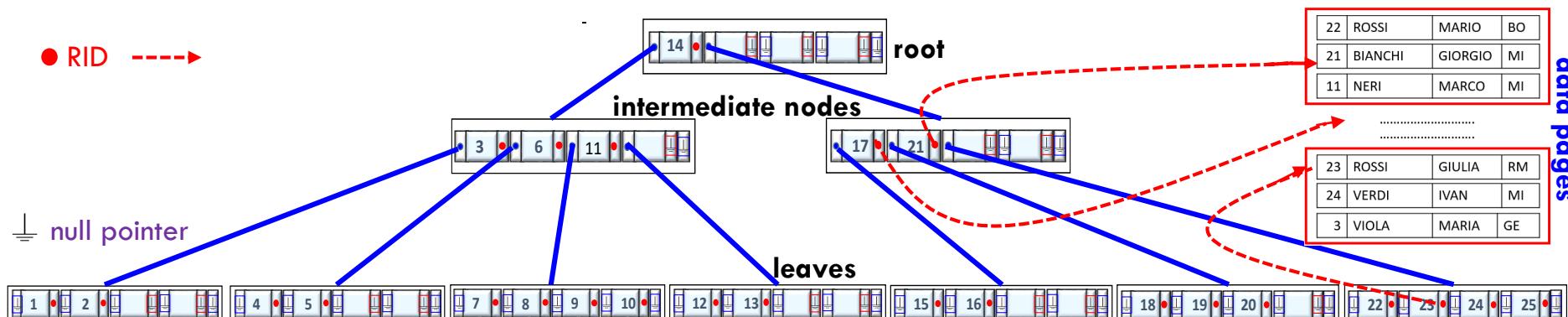
# B-tree: ipotesi semplificative ed esempio

- Per semplicità di spiegazione, assumiamo:

- indice su singolo attributo che assume valori non ripetuti;
- ordinamento crescente dei valori di chiave all'interno di ogni nodo;
- uguale lunghezza per ogni valore di chiave;
- stesso formato del nodo sia per le foglie sia per gli altri nodi;
- record memorizzati in area separata rispetto all'indice B-tree.

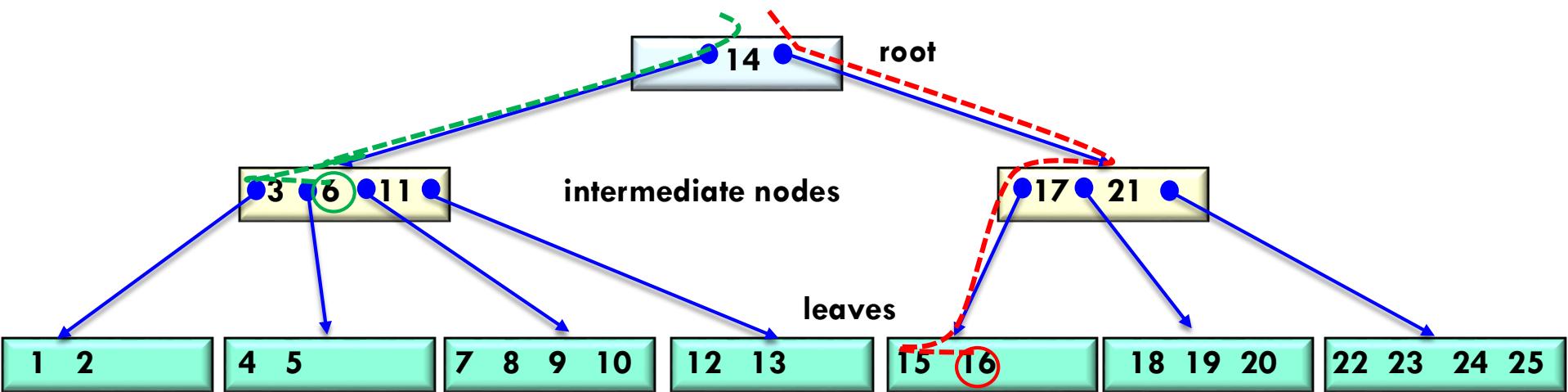
Nell'esempio,  
4 slot per  
chiavi dove  
ognuna due  
figli possibili

Un B-tree  $T$  della classe  $\tau(2,3)$  – esempio di indice su chiave primaria.



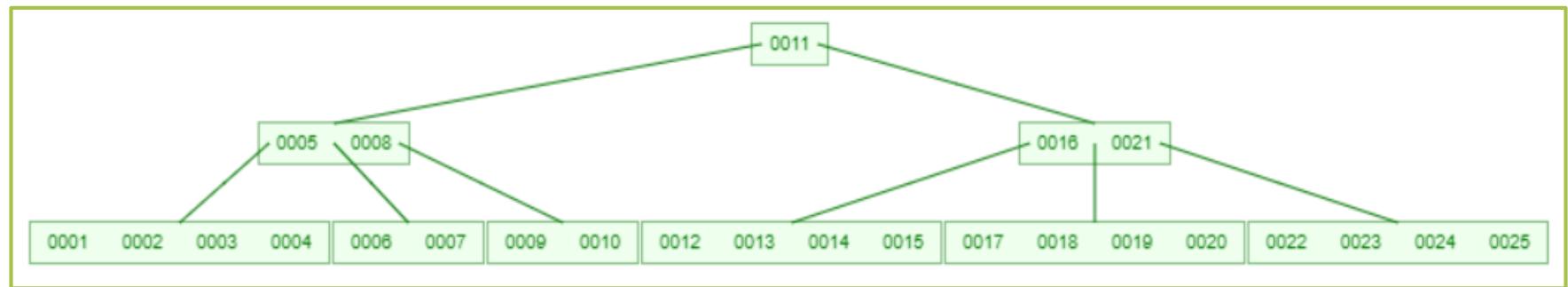
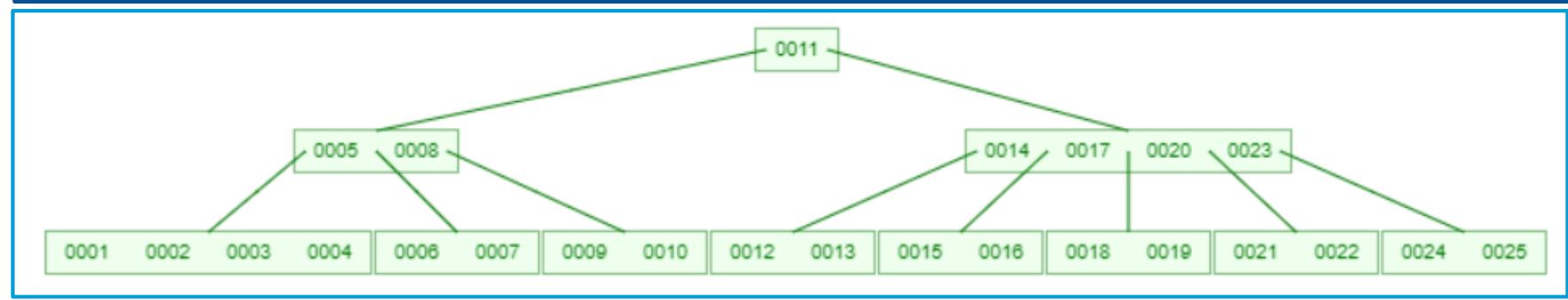
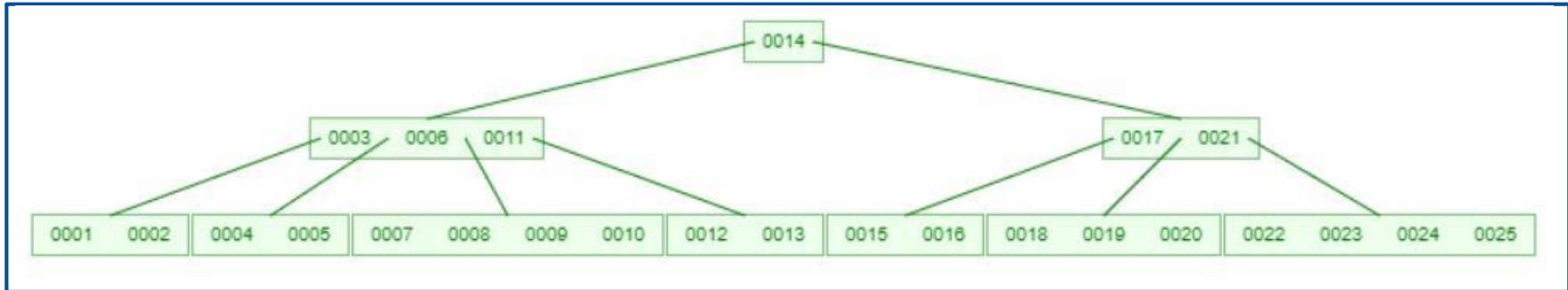
# Ricerca in un B-tree: un esempio

- a) Percorso per la ricerca del record con valore di chiave uguale a 16
- b) Percorso per la ricerca del record con valore di chiave uguale a 6



# Diversi possibili B-tree fissati g e h

È possibile ottenere diversi B-tree  $T$  della classe  $\tau(2,3)$  per i medesimi valori di chiave, a seconda della sequenza d'inserimento e degli algoritmi utilizzati. Di seguito alcune configurazioni per l'esempio introdotto:

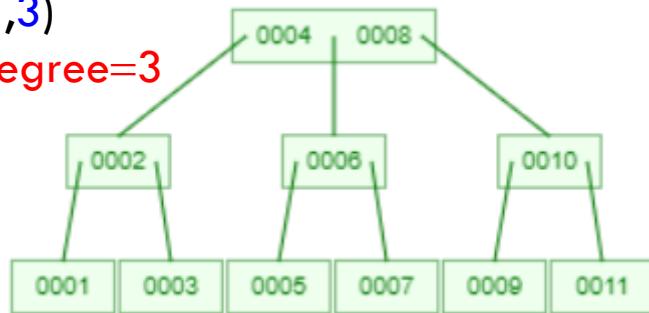


# Diversi possibili B-tree variando g e h

È possibile ottenere diversi B-tree  $\mathcal{T}$  di diverse classi  $\tau(g,h)$  per i medesimi valori di chiave e per la medesima sequenza d'inserimento:

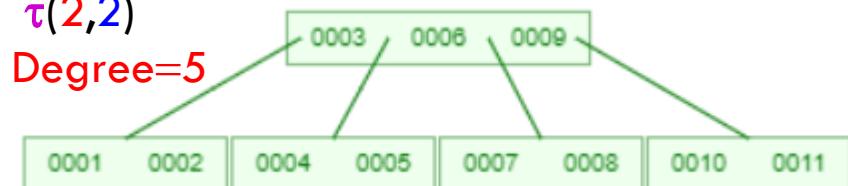
$\tau(1,3)$

Max. Degree=3



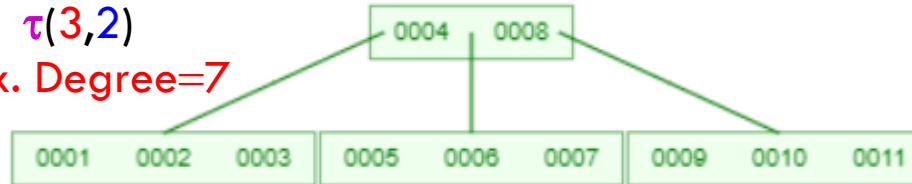
$\tau(2,2)$

Max. Degree=5



$\tau(3,2)$

Max. Degree=7



N.B. Nel tool l'ordine è  $m = \text{Max. Degree}$ , essendo  $m$  il massimo numero di figli che può avere un nodo non foglia ( $m-1$  è il massimo numero di chiavi in un nodo): per  $m$  dispari vale la corrispondenza:  $g = (m-1)/2$ . La definizione di Bayer e McCreight non consente un numero pari per il massimo numero di figli di un nodo non foglia.

# B-tree: organizzazione di un nodo

Un B-tree è organizzato a nodi (detti anche pagine logiche o index page):

- 1. ogni nodo intermedio o nodo foglia memorizza tra  $g$  e  $2g$  chiavi (intese come valori della chiave di ricerca); la radice può contenere da 1 a  $2g$  chiavi e può avere di conseguenza da 0 a  $2g+1$  puntatori a figli;
- 2. un nodo intermedio con  $1$  chiavi ( $g \leq 1 \leq 2g$ ) ha  $1+1$  puntatori ad altrettanti nodi figli;
- 3. in ogni nodo le chiavi sono memorizzate in ordine crescente.



$k_i$ : generico valore di chiave;

$<5$   $(5)$   $(m)$   $>11$   
 $>5$  e  $<11$

P: index page

□ per  $i = 1, \dots, l-1$

$p_i$ : puntatore (Row Identifier – RID) al record con valore di chiave  $k_i$ ;

$q_{i-1}$ : puntatore (Page Identifier – PID, null se P è un nodo foglia) al nodo figlio che contiene valori di chiave  $k < k_i$ ;

$q_i$ : puntatore (Page Identifier – PID, null se P è un nodo foglia) al nodo figlio che contiene valori di chiave  $k_i < k < k_{i+1}$ ;

□ per  $i = 1$

$q_l$  è il puntatore (null se P è un nodo foglia) al nodo figlio che contiene valori  $k > k_l$ .

# B-tree: insieme dei valori

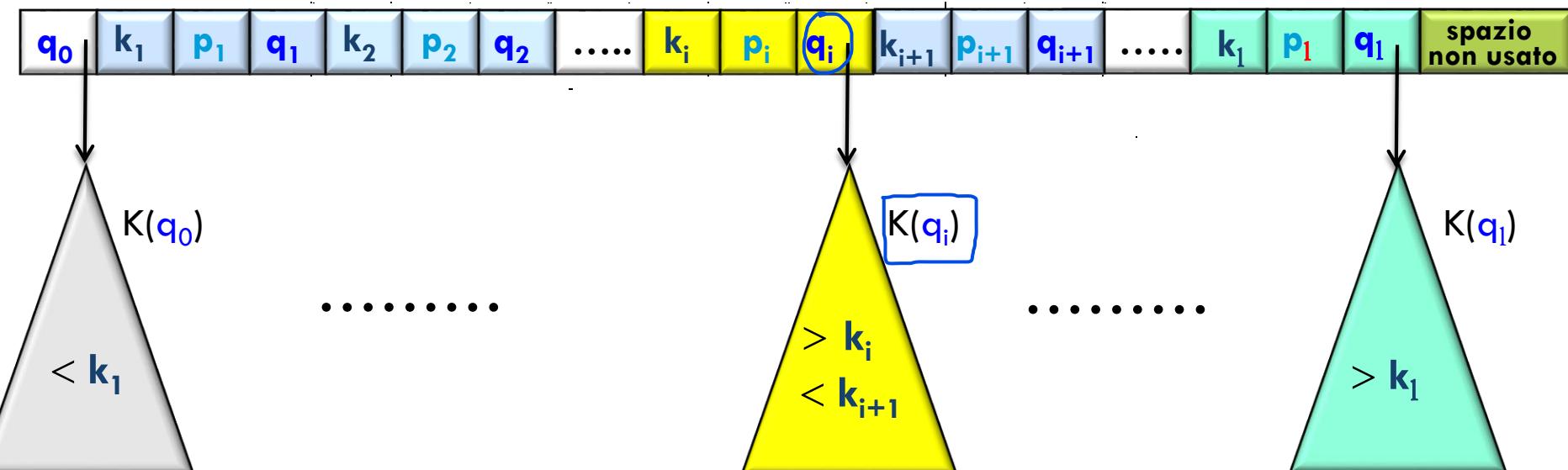
Sia  $K(q_i)$  l'insieme dei valori di chiave del sottoalbero la cui radice è il nodo  $P$  di indirizzo  $q_i$ . Si ha:

$$\forall y \in K(q_0): (y < k_1)$$

Ipotesi: ordinamento crescente  
e valori di chiavi non ripetuti.

$$\forall y \in K(q_i): (k_i < y < k_{i+1}) \quad i=1,2,\dots,\ell-1$$

$$\forall y \in K(q_\ell): (k_\ell < y)$$



# Ricerca in un B-tree

P(q)	il nodo puntato da q
k <sub>1</sub> ,...,k <sub>l</sub>	le chiavi in P(q)
q <sub>0</sub> ,...,q <sub>l</sub>	i puntatori in P(q)
y	il valore di chiave da cercare
root	il puntatore alla radice
s	puntatore per inserimento

Il costo di ricerca di un valore di chiave è pari al numero di nodi letti:

$$1 \leq C(\text{search}) \leq h$$

```
{q ← root;  
s ← null;  
trovata ← false;  
while (q≠null) and (not trovata) do  
    { s ← q;  
        if y< k1 then q ← q0      Se y < k1, q viene aggiornato per puntare al sottoalbero sinistro.  
        else if ∃ i (y=ki)  
            then trovata ← true  
        else if ∃ i (ki< y < ki+1)  
            then q ← qi  
        else q ← ql  
    }  
}
```

Se esiste un i tale che k<sub>i</sub> < y < k<sub>i+1</sub>, q viene aggiornato per puntare al sottoalbero giusto, cioè q<-q<sub>i</sub>.

Al termine della ricerca:

se trovata = true  
allora q punta al nodo dove risiede il valore cercato y  
altrimenti  
q = null e s punta al nodo dove andrebbe inserito il valore y

# Evoluzione di un B-tree

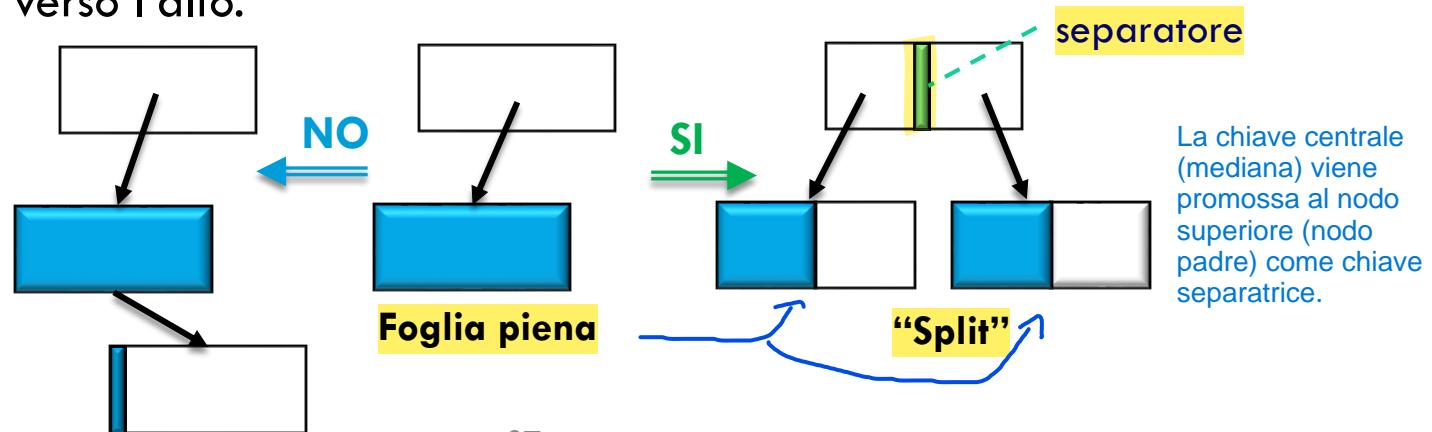
Se la divisione e la propagazione raggiungono la radice e anche la radice è piena, la radice stessa viene divisa in due, e una nuova radice viene creata, aumentando l'altezza dell'albero. In questo modo, l'albero "cresce verso l'alto".

- L'idea chiave su cui si basano gli algoritmi per l'inserimento e la cancellazione in un B-tree è la seguente:

le modifiche partono sempre dalle foglie e l'albero cresce o si accorcia **verso l'alto** ovvero, nel caso di inserimenti, non si "appendono" nuovi nodi alle foglie ma, se necessario, si crea una nuova foglia allo stesso livello delle altre e si propaga un valore di chiave (**separatore**) verso l'alto.

- Questo modo di procedere è reso possibile dal fatto che i nodi ai livelli superiori non sono necessariamente pieni, e quindi possono "assorbire" le informazioni che si propagano a partire dalle foglie. La propagazione degli effetti sino alla radice può provocare l'aumento dell'altezza dell'albero. In questo senso si dice che i B-tree crescono verso l'alto.

Se il nodo superiore (nodo padre) ha spazio, la chiave separatrice viene inserita. Tuttavia, se anche il nodo superiore è pieno, anch'esso viene diviso, e la chiave centrale viene propagata ulteriormente verso l'alto, ripetendo il processo. Questo può continuare fino alla radice.



# Inserimento di una chiave nel B-tree

L'inserimento di una nuova chiave in un B-tree comporta una ricerca per verificare se essa è già presente nell'albero. Nell'ipotesi di non consentire duplicati (es. **primary index**), si procede all'inserimento solo in caso di insuccesso della ricerca. L'inserimento avviene sempre in una foglia.

Si distinguono due casi:

- se la foglia non è piena, si inserisce la chiave e il puntatore al record, quindi si riscrive la foglia così aggiornata;
- se la foglia è piena, si attiva un processo di “splitting” che può essere ricorsivo e, nel caso peggiore, propagarsi fino alla radice.

```
{ ricerca la chiave y;  
  if (not trovata)  
    then { if s = null
```

then crea la radice con y

else if P(s) è pieno

then attiva splitting

else inserisci (y,p,null) in P(s)

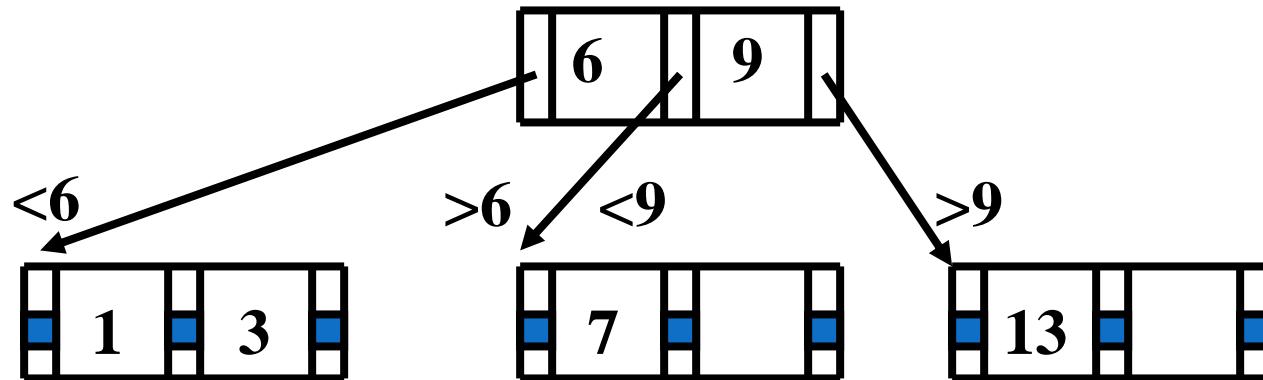
```
}
```

N.B. p è il puntatore al record con chiave y.

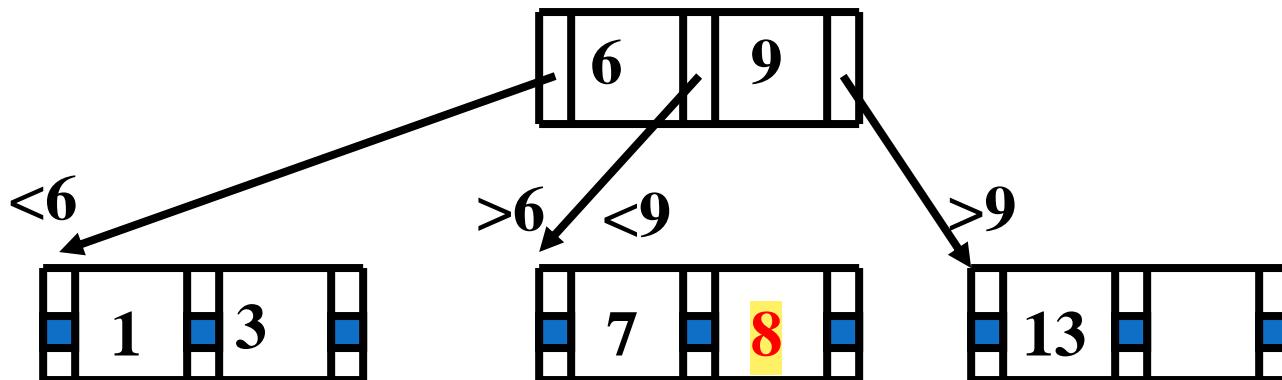
```
}
```

# Caso 1: foglia non piena

Prima dell'inserimento del valore di chiave 8:



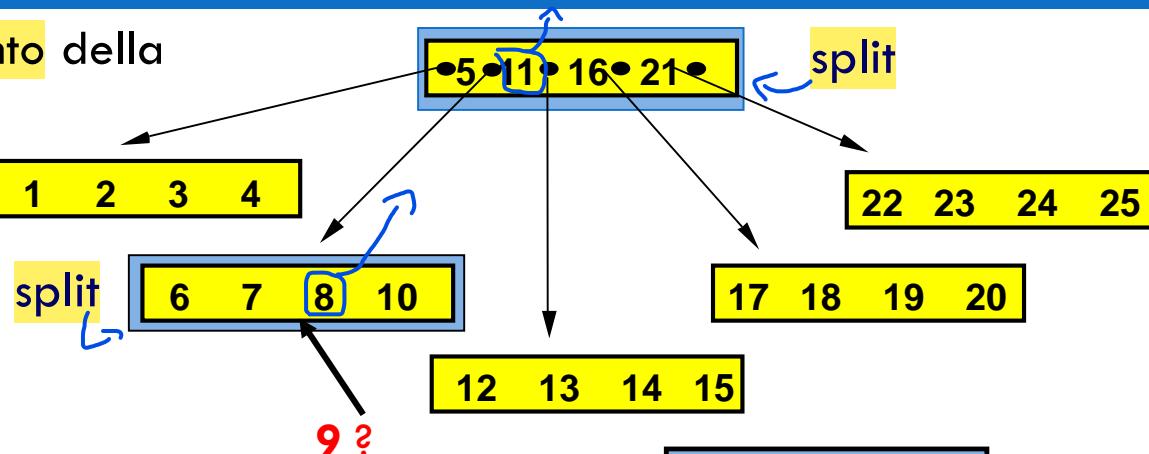
Dopo l'inserimento:



# Caso 2: foglia piena (splitting)

Prima dell'inserimento della chiave 9 nel B-tree

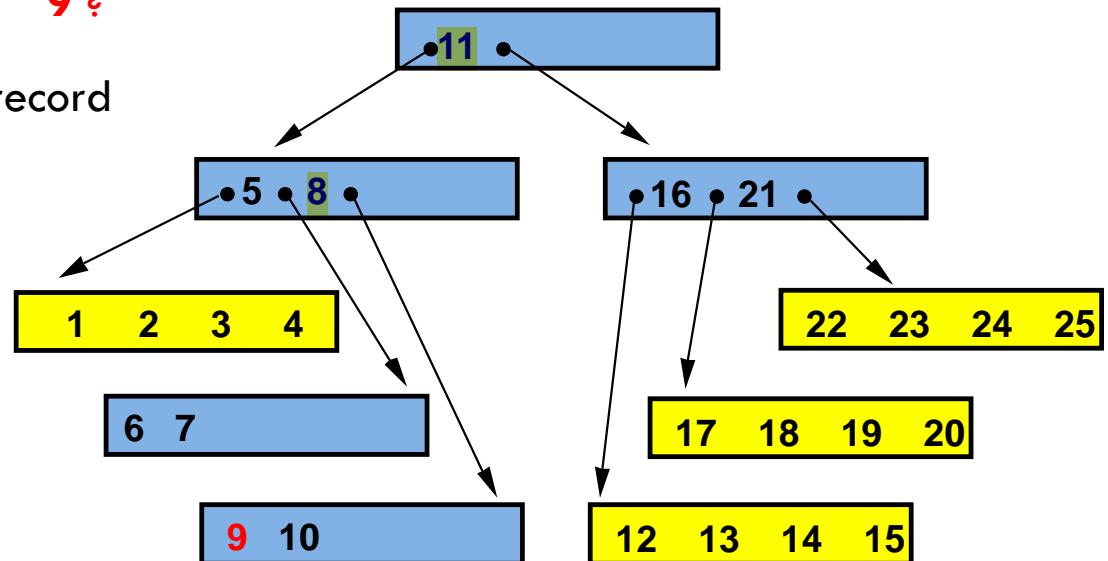
$\tau$  della classe  $\tau(2,2)$   
 $2 \leq g \leq 4$



N.B. Non sono mostrati i puntatori ai record

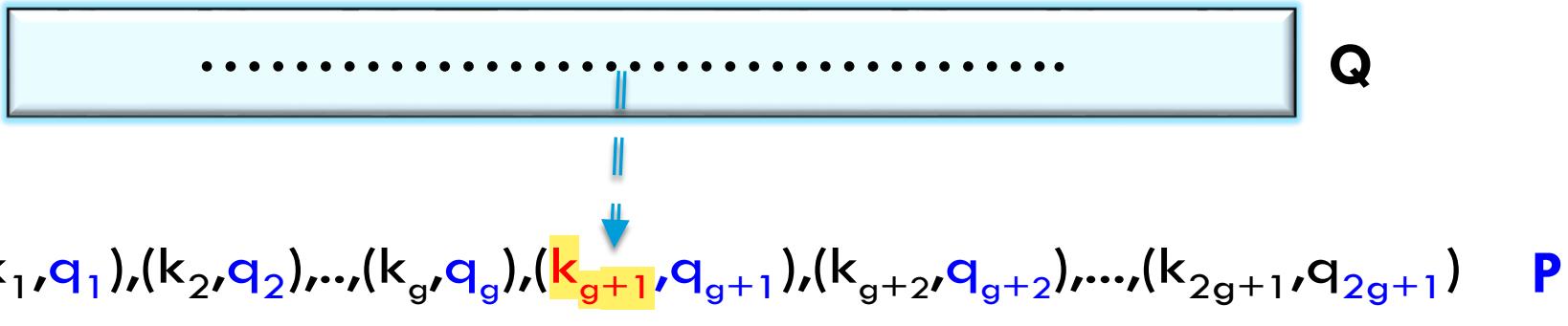
Dopo l'inserimento del valore 9:

$\tau$  della classe  $\tau(2,3)$



# Splitting di un nodo: valore mediano

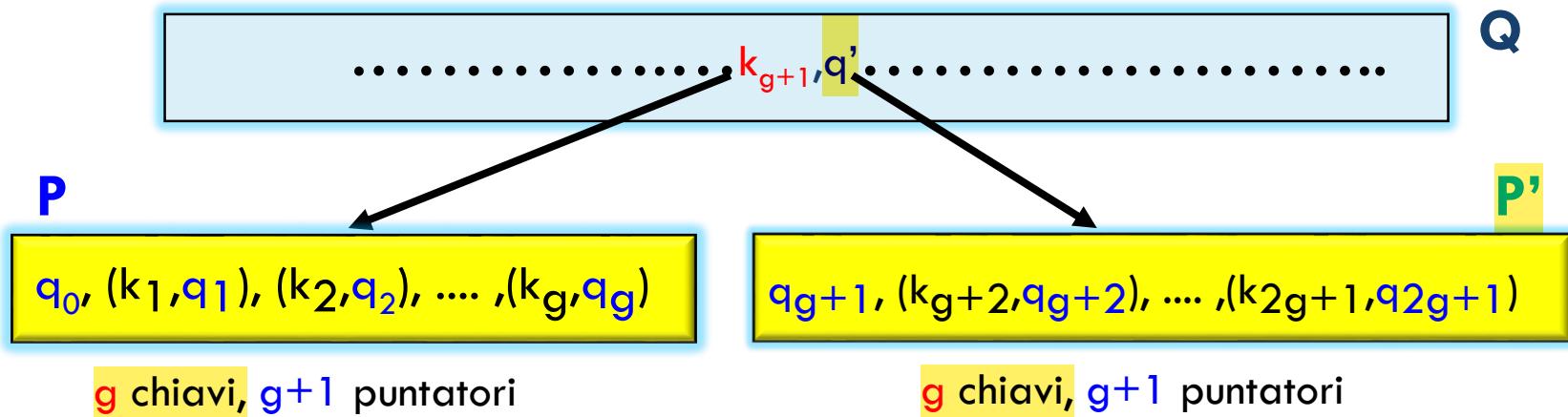
- **P**: nodo pieno dove inserire una chiave
- **Q**: nodo padre di **P**
- Si consideri la sequenza \* ordinata di  $2g+1$  entrate che si verrebbe a creare in **P** (si omettono i puntatori  $p_i$ )



- \*  $q_0, (k_1, q_1), (k_2, q_2), \dots, (k_g, q_g), (k_{g+1}, q_{g+1}), (k_{g+2}, q_{g+2}), \dots, (k_{2g+1}, q_{2g+1})$  **P**
- Valore mediano di chiave:  $k_{g+1}$ .
- Si alloca un nuovo nodo **P'** e si partizionano le restanti chiavi tra **P** e **P'**.
- Si inserisce in **Q** l'entrata  $(k_{g+1}, q')$ .

# Splitting e modifica del nodo padre

- Risultato dopo lo splitting di  $P$

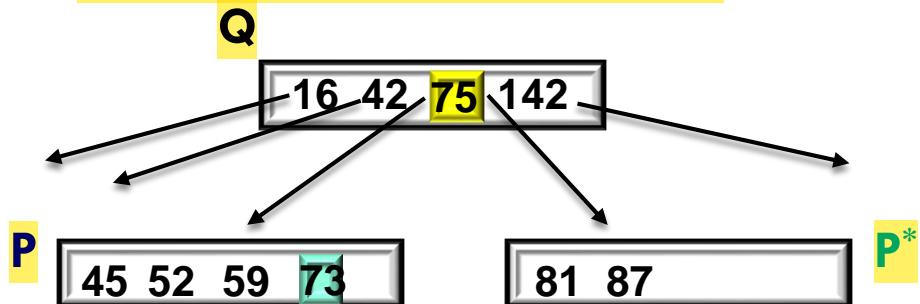


- Se anche  $Q$  risulta pieno e non riesce a ospitare l'aggiunta di un'entrata, il processo di splitting si propaga.
- Se il nodo da sdoppiare è la radice del B-tree, allora la nuova radice  $Q$  conterrà  $(q, k_{g+1}, q')$  dove  $q$  punta a  $P$  e  $q'$  punta a  $P'$ .
- L'algoritmo di splitting preserva l'ordine  $g$  del B-tree e le proprietà 1,2,3 definite in precedenza.

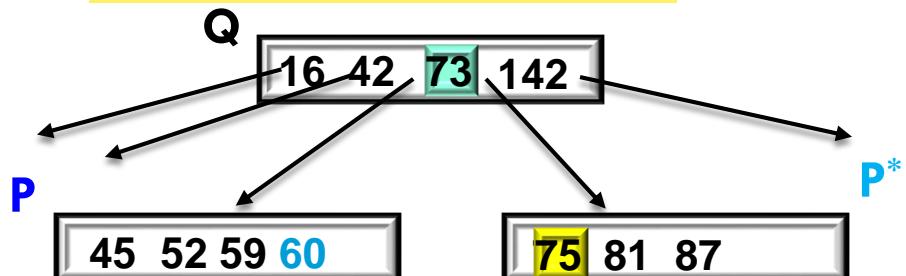
# Gestione dell'overflow

- Due nodi **P** e **P\*** sono adiacenti se figli dello stesso padre **Q** e indirizzati da puntatori adiacenti in **Q**.
  - Una strategia per evitare eccessivi split: si accede a un nodo adiacente **P\*** al nodo **P** pieno e si ridistribuiscono le chiavi tra **P**, **P\*** e **Q**.
- Un B-tree che adotta questa strategia è detto un B-tree che **gestisce l'overflow**.
- La gestione dell'overflow tende a generare alberi con nodi più pieni, ma comporta, in generale, maggiori costi di inserimento.
- La gestione dell'overflow può essere generalizzata, infatti si possono considerare per la ridistribuzione 3 o più nodi fratelli anziché 2.

Prima dell'inserimento della chiave **60**:



Dopo l'inserimento della chiave **60**:

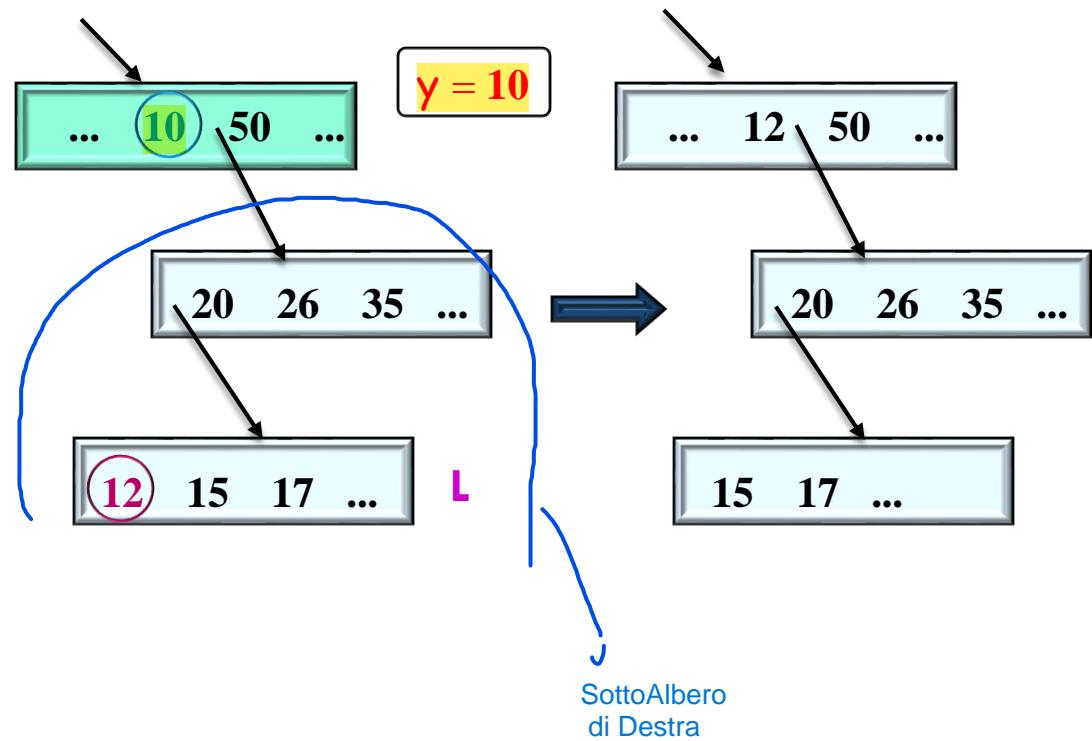


# Eliminazione di una chiave

- Se la chiave  $y$  da cancellare si trova in una foglia  $L$ , si rimuove.
- In caso contrario,  $y$  è rimpiazzata dal valore di chiave più piccolo del suo sottoalbero di destra.

In entrambi i casi un valore che risiede in una certa foglia  $L$  deve essere cancellato.

Se, in conseguenza di ciò,  $L$  viene a contenere un numero di chiavi inferiore a  $g$ , si rende necessario modificare la struttura dell'albero tramite processi di **catenation** e **underflow**.

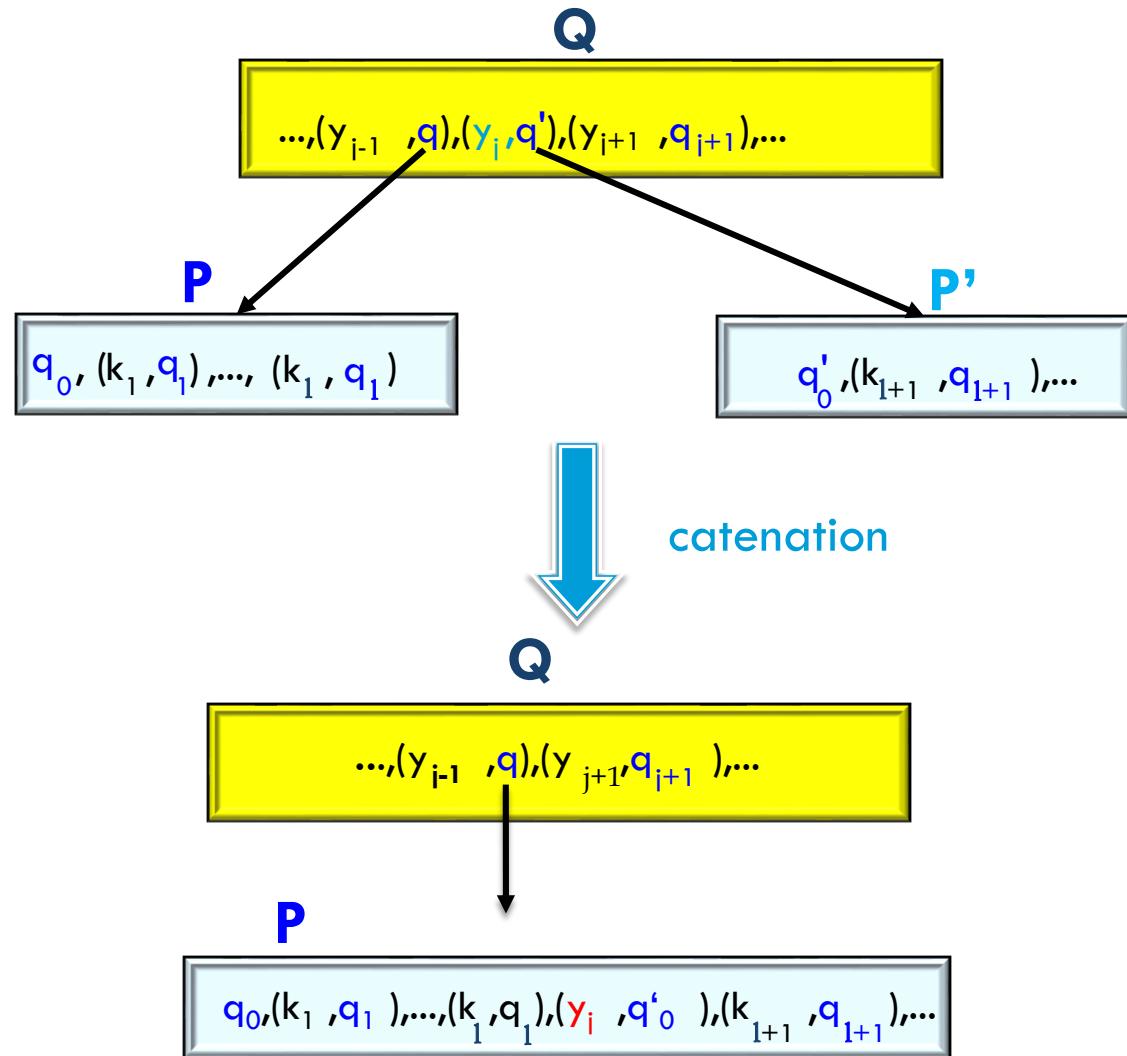


# Algoritmo per l'eliminazione di una chiave

```
{ ricerca la chiave y;  
  if trovata  
    then { if y in una foglia L  
           then cancella y da L  
           else { ricerca nel sottoalbero di destra fino a una foglia  
                  L seguendo la catena dei puntatori q0;  
                  sostituisci a y la prima chiave k di L;  
                  cancella la prima chiave k di L;  
           };  
           if L contiene meno di g chiavi  
             then attiva catenation e underflow  
     }  
 }
```

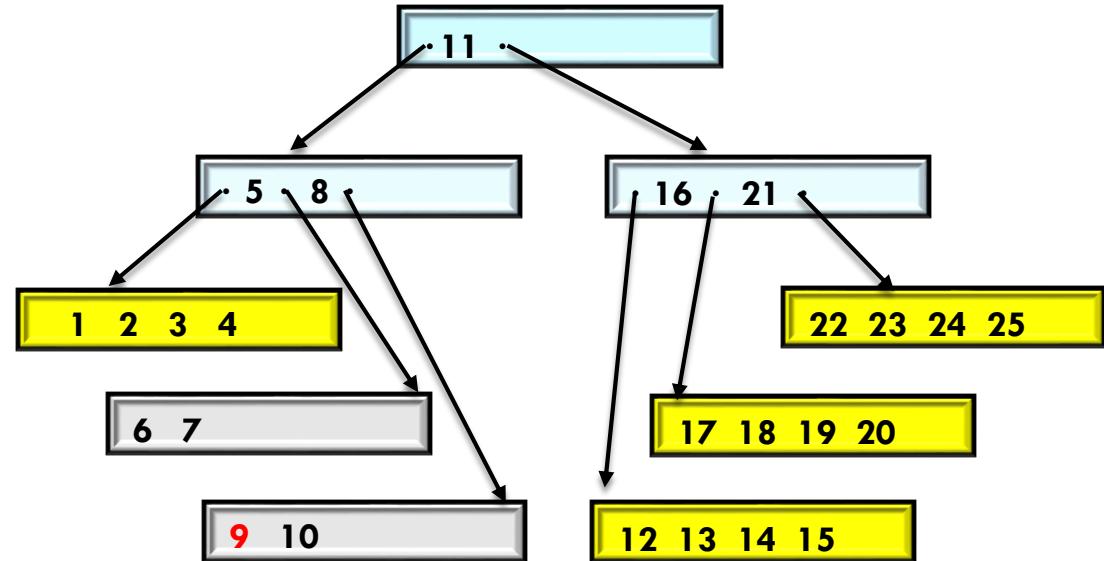
# Catenation

- La catenation di due nodi adiacenti  $P$  e  $P'$  è possibile se, complessivamente, i due nodi contengono meno di **2g chiavi**.
- Poiché ogni nodo contiene almeno **g** chiavi, la catenation si attiva quando  $P$  ha  **$g-1$**  chiavi e  $P'$  esattamente **g**.
- Come conseguenza della cancellazione dell'entrata  $(y_i, q')$  nel nodo  $Q$ , è possibile che **anche  $Q$**  venga a contenere meno di **g chiavi** e pertanto il processo si può propagare fino alla radice.

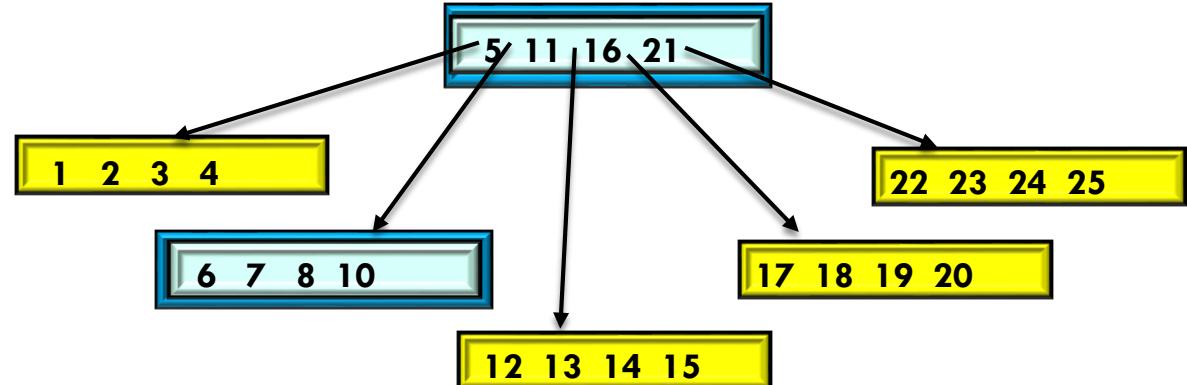


# Esempio di catenation

- +  $T \in \tau(2,3)$  prima della cancellazione della chiave 9.



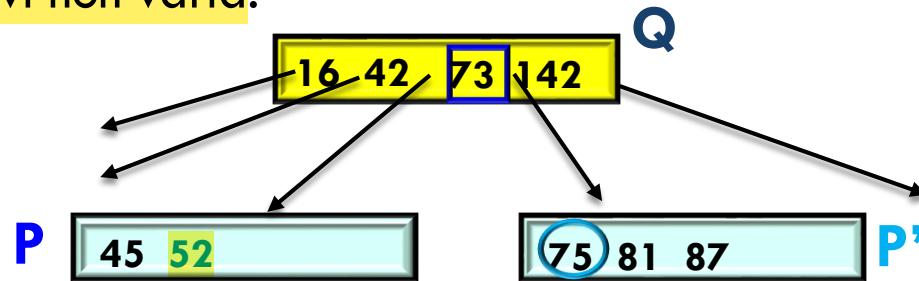
- +  $T \in \tau(2,2)$  dopo la cancellazione della chiave 9.



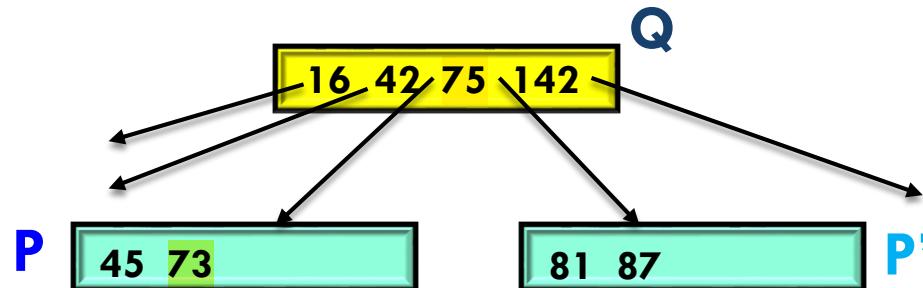
# Underflow

- Se, prima della cancellazione, la somma del numero delle chiavi in **P** e **P'** è maggiore di **2g**, allora le chiavi devono essere ridistribuite tra i due nodi e il nodo padre **Q**.
- Il processo di underflow non si propaga in quanto **Q** viene modificato ma il numero delle sue chiavi non varia.

Prima della cancellazione  
della chiave **52** :

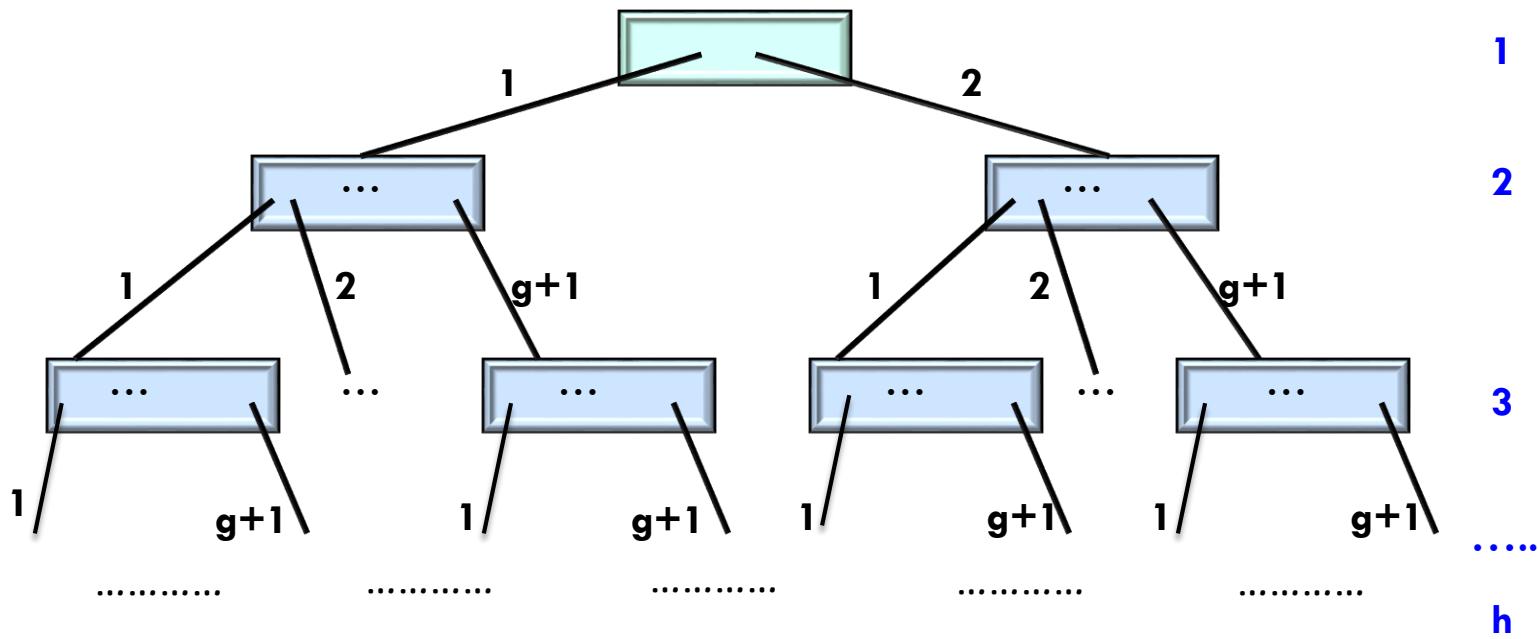


Dopo la cancellazione della chiave **52** :



# Numero minimo di nodi di un B-tree

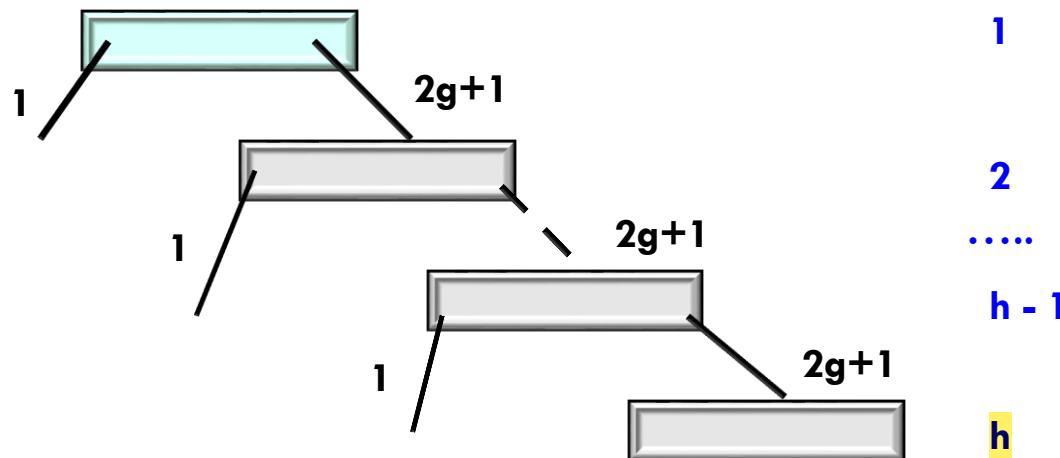
- Numero minimo di nodi  $IP_{\min}$  di un albero  $T \in \tau(g, h)$ :



$$\begin{aligned} IP_{\min} &= 1 + 2 + 2(g + 1) + 2(g + 1)(g + 1) + \dots + 2(g + 1)^{h-2} = \\ &= 1 + 2 \left( (g + 1)^0 + (g + 1)^1 + (g + 1)^2 + \dots + (g + 1)^{h-2} \right) = \\ &= 1 + 2 \sum_{i=0}^{h-2} (g + 1)^i = 1 + 2 \frac{(g+1)^{h-1}-1}{(g+1)-1} = 1 + \frac{2}{g} \left( (g + 1)^{h-1} - 1 \right) \end{aligned}$$

# Numero massimo di nodi di un B-tree

- Numero massimo di nodi  $IP_{\max}$  di un albero  $\mathcal{T} \in \tau(g, h)$ :



$$IP_{\max} = \sum_{i=0}^{h-1} (2g + 1)^i = \frac{(2g + 1)^h - 1}{(2g + 1) - 1} = \frac{1}{2g} \left( (2g + 1)^h - 1 \right)$$

# Altezza di un B-tree

- **Caso peggiore:** il minimo numero di chiavi presenti in un B-tree  $\mathcal{T} \in \tau(g,h)$  si ha quando il numero di nodi è pari a  $IP_{\min}$  e quindi ogni nodo, eccetto la radice, contiene  $g$  chiavi, e la radice una sola chiave.

$$NK_{\min} = 1 + g \times (IP_{\min} - 1) = 2 \times (g + 1)^{h-1} - 1$$

- **Caso migliore:** il massimo numero di chiavi presenti in un B-tree  $\mathcal{T} \in \tau(g,h)$  si ha quando il numero di nodi è pari a  $IP_{\max}$  e quindi ogni nodo, compresa la radice, contiene  $2g$  chiavi:

$$NK_{\max} = 2g \times IP_{\max} = (2g + 1)^h - 1$$

- pertanto se il B-tree ha  $NK$  chiavi si ha:

$$\left\lceil \log_{2g+1}(NK + 1) \right\rceil \leq h \leq \left\lceil 1 + \log_{g+1} \left( \frac{NK + 1}{2} \right) \right\rceil$$

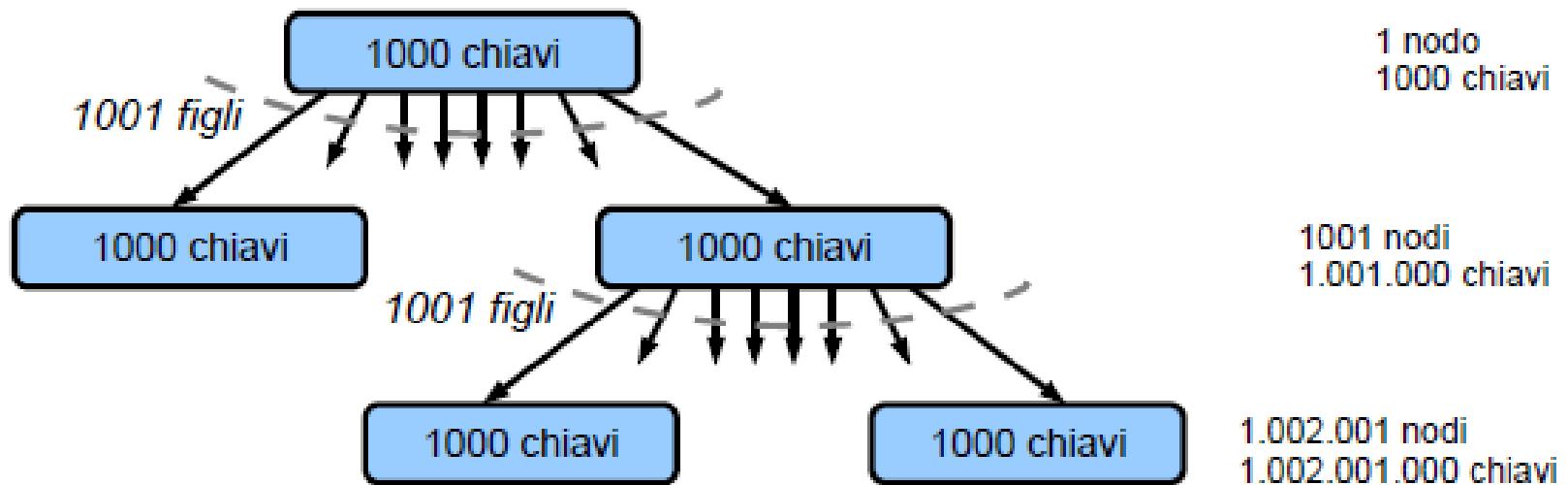
# Pregi e difetti dei B-tree

- Un B-tree è molto efficiente per la ricerca e la modifica di singoli record.
- Esiste un limite inferiore all'utilizzazione della memoria (50%), ma l'utilizzazione media è solo del 69%.
- Un B-tree non è particolarmente adatto per elaborazioni di tipo sequenziale nell'ordine dei valori di chiave, e nel reperimento di valori di chiave in un intervallo dato. L'inefficienza deriva dal fatto che i RID sono memorizzati all'interno dei nodi.
- La ricerca del successore di un valore di chiave può comportare la scansione di molti nodi.
- La ricerca del valore di chiave più piccolo, che si trova nella foglia più a sinistra, implica l'accesso a tutti i nodi del percorso tra la radice e la foglia.

# B-tree: note

- Poiché un nodo può contenere in genere molti valori di chiave, un B-tree può indicizzare file con un numero molto elevato di record, offrendo benefici in termini di operazioni di I/O per ricerche su singolo valore di chiave.

Esempio: B-tree di ordine  $m = 1001$ .

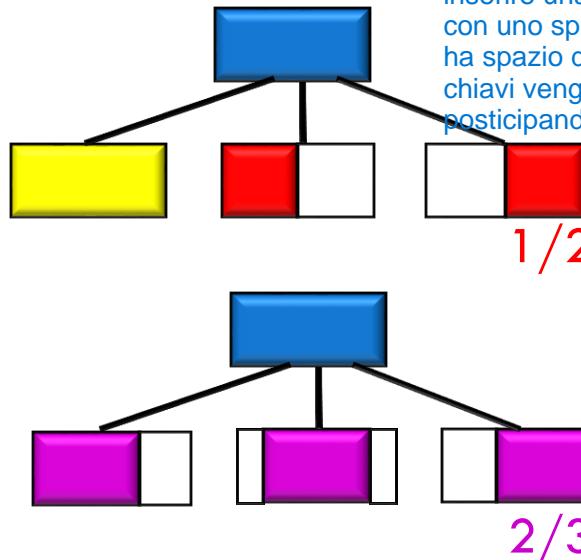
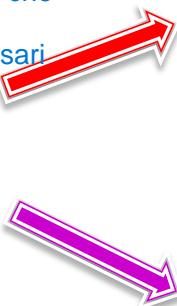
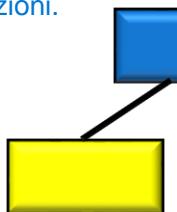


# B\* - tree

Nel B\*-tree, prima di effettuare uno split, si tenta una ridistribuzione locale delle chiavi tra i nodi adiacenti per evitare la divisione immediata. Lo splitting avviene solo quando entrambi i nodi fratelli sono anch'essi pieni. In questo caso, invece di dividere un nodo in due, due nodi vengono ridistribuiti in tre nodi, ciascuno dei quali sarà riempito per almeno 2/3.

- Il B\*-tree (adottando il termine da Knuth) è una variante del B-tree in cui l'utilizzazione dei nodi è almeno pari a 2/3 anziché 1/2.
- L'inserimento in un B\*-tree implica l'adozione di uno schema di ridistribuzione locale così da ritardare lo splitting al caso in cui due fratelli adiacenti siano completamente pieni.
- In questo caso da 2 nodi se ne derivano 3 ciascuno riempito per 2/3. La figura illustra la differenza con i B-tree che gestiscono l'overflow.

In un B-tree, i nodi sono riempiti per almeno il 50% (cioè, ogni nodo deve contenere almeno la metà del numero massimo di chiavi che può contenere). Tuttavia, nel B\*-tree, i nodi devono essere riempiti almeno per 2/3. Questo significa che i nodi del B\*-tree sono più densamente popolati rispetto a quelli del B-tree, il che riduce il numero di nodi totali nell'albero e, di conseguenza, il numero di accessi al disco necessari durante le operazioni.



B\*-tree

44

Il B-tree\* migliora l'efficienza dell'uso dello spazio e riduce il numero di split dei nodi durante le operazioni di inserimento.

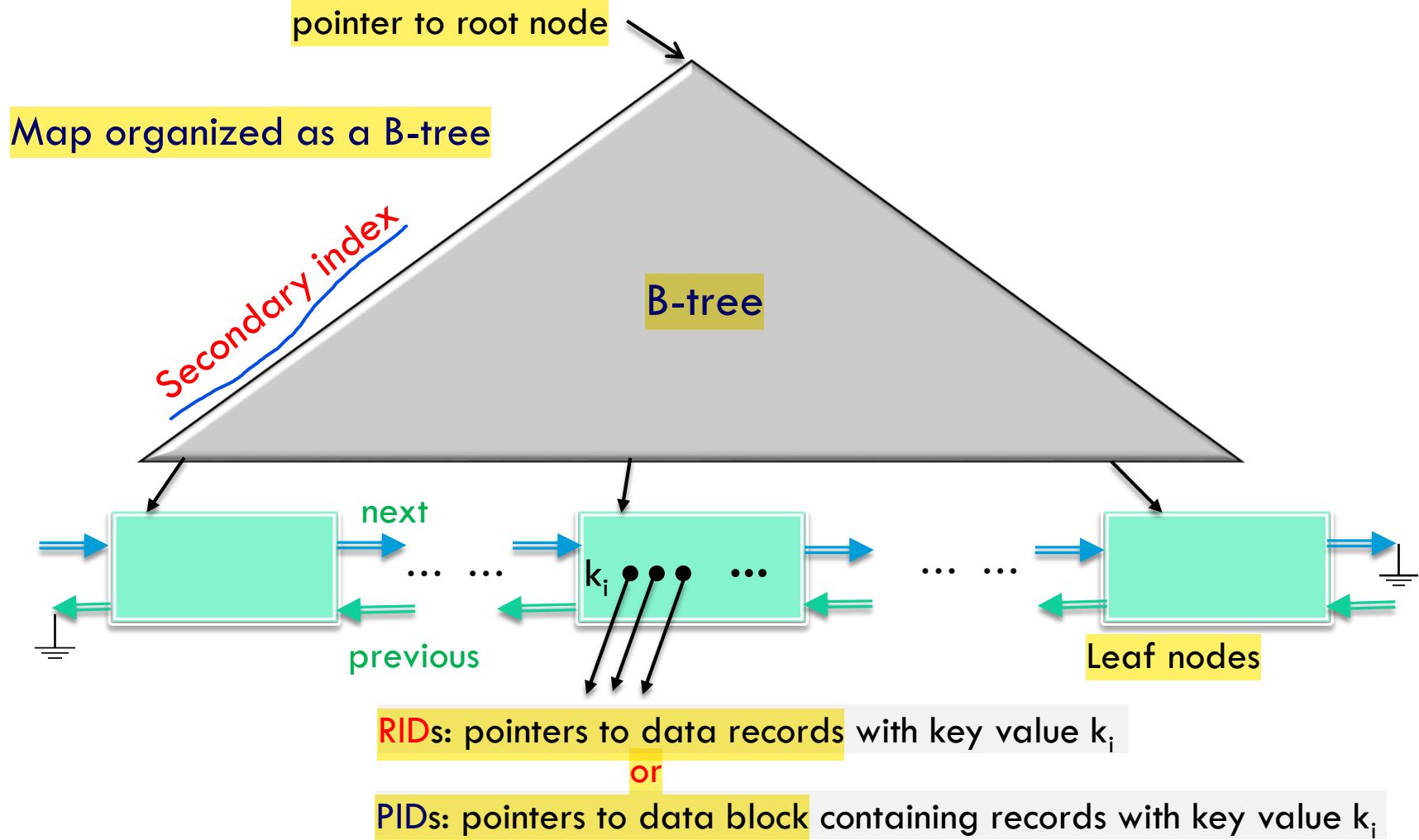
Schema di ridistribuzione locale:  
Quando un nodo del B\*-tree è pieno e si deve inserire una nuova chiave, prima di procedere con uno split, l'albero verifica se il nodo adiacente ha spazio disponibile. Se il fratello ha spazio, le chiavi vengono ridistribuite tra quei nodi, posticipando la necessità di uno split.

B-tree

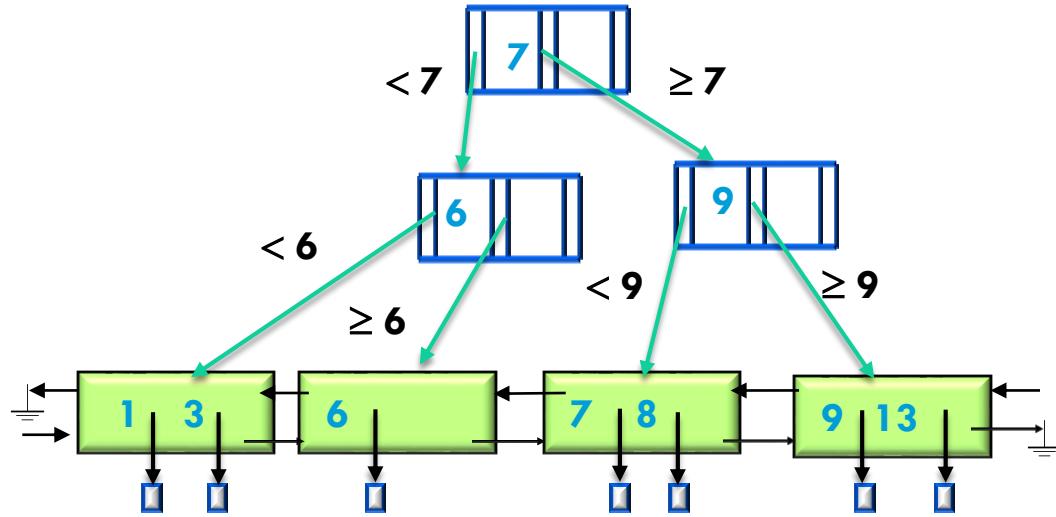
# B<sup>+</sup>-tree

- In un **B-tree**, i valori di chiave svolgono una duplice funzione:
  - ▣ come **separatori** permettono di determinare il cammino da seguire in fase di ricerca;
  - ▣ come **valori di chiave** permettono di accedere all'informazione a essi associata.
- Nei B<sup>+</sup>-tree queste funzioni sono, invece, **mantenute separate**:
  - ▣ le foglie contengono tutti i valori di chiave;
  - ▣ la radice e i nodi interni, organizzati come un **B-tree**, costituiscono solo una “**mappa**” per consentire una rapida localizzazione delle chiavi, e memorizzano “**separatori**” di cammino;
  - ▣ a parità di dimensione del nodo, l’ordine del **B-tree** che funge da mappa è più grande rispetto a quello di un solo **B-tree**, non dovendo né la radice né i nodi interni memorizzare puntatori a record.
- Al fine di facilitare elaborazioni sequenziali e su intervalli, **le foglie sono tra loro concatenate in una lista**. È inoltre presente un puntatore alla testa della lista. La lista, nella maggior parte delle attuali implementazioni, è doppiamente concatenata (**doubly-linked list**).

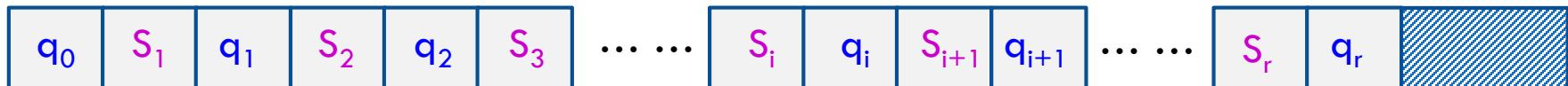
# Struttura di un B<sup>+</sup>-tree



# B<sup>+</sup>-tree : formato dei nodi



**Format of the root and internal nodes**



$r \leq 2g$

**Format of a leaf node**



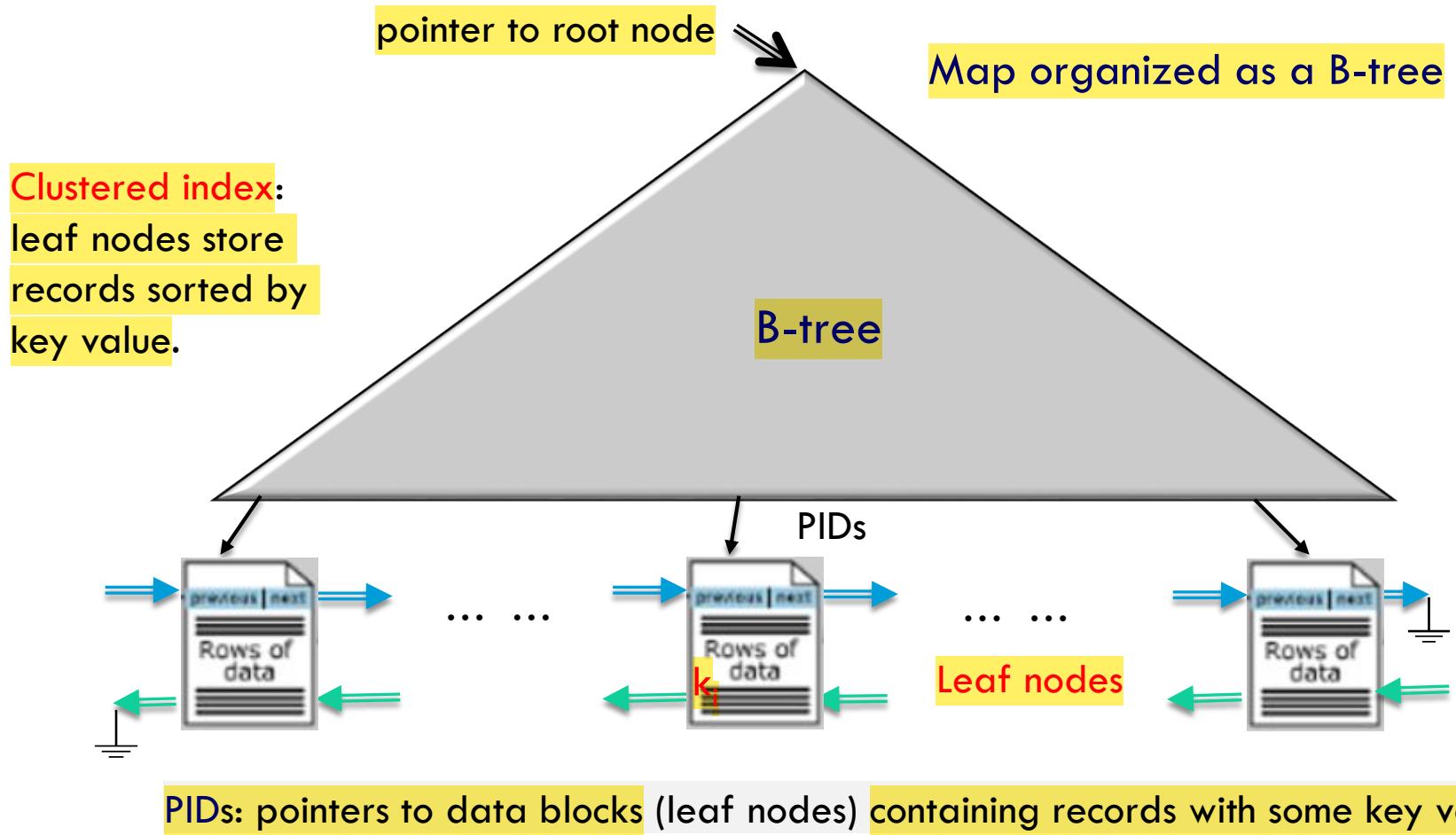
$t \leq 2g$

**q** : pointer to subtree (PID)

**p** : pointer to record (RID); **next, prev**: pointer to leaf

**S** : separator; **k** : key value

# Struttura di un B<sup>+</sup>-tree come data file

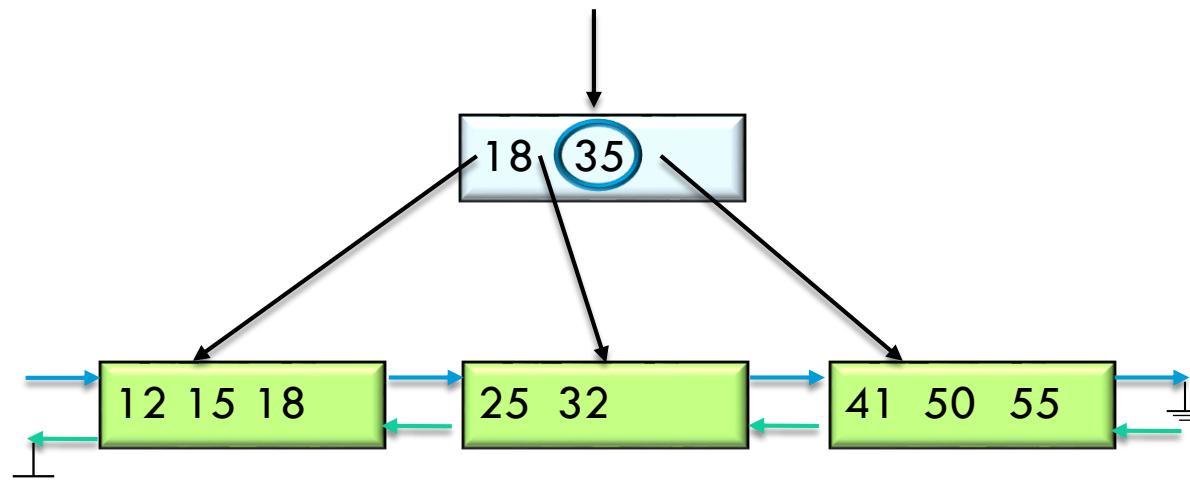


# Separatori

- In un B<sup>+</sup>-tree, la sola funzione dei separatori è determinare il giusto cammino quando si ricerca un valore di chiave. Dunque è anche possibile che un separatore non sia un valore di chiave presente nel file dati.

Nell'esempio si considera ordinamento crescente:

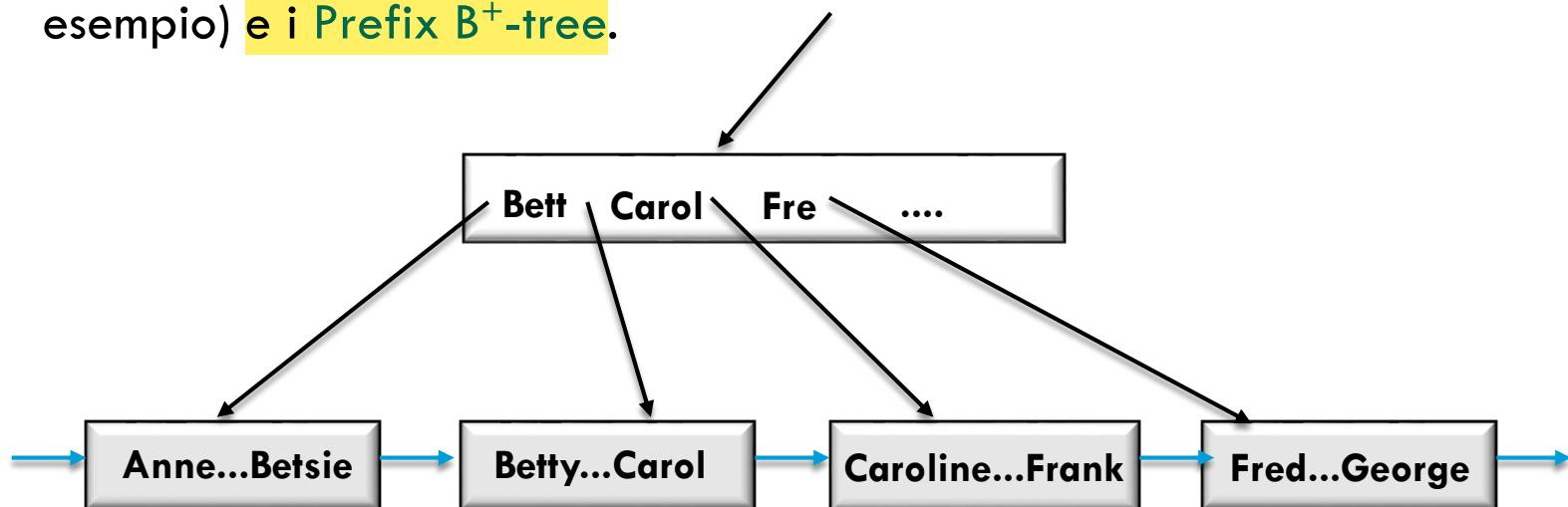
- il sottoalbero sinistro di un separatore contiene valori di chiave minori o uguali al separatore;
- il sottoalbero destro contiene valori di chiave strettamente maggiori del separatore.



# Separatori come prefissi

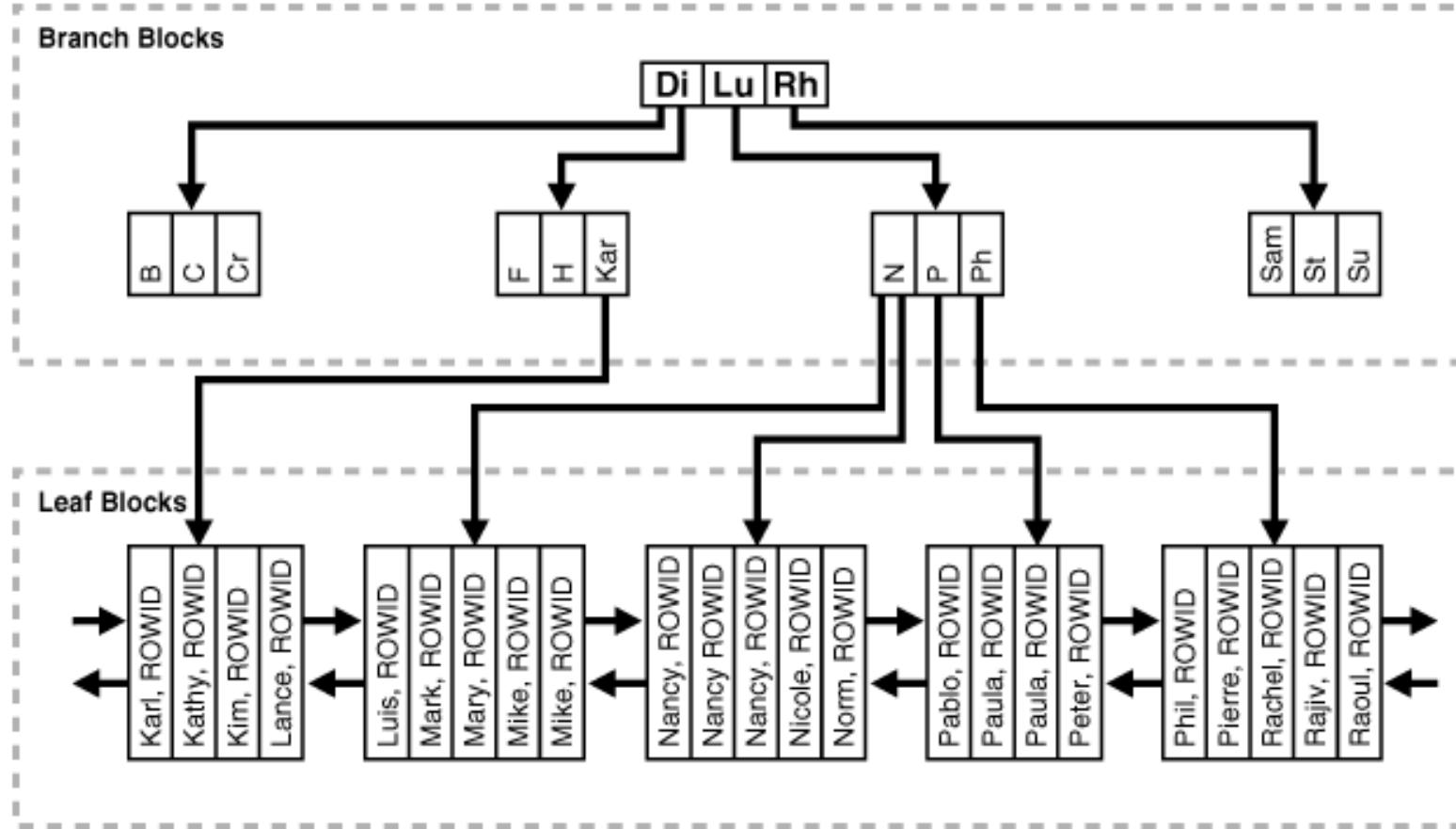
Il Prefix B+-tree è un'altra variante che adotta la compressione dei separatori, ma in maniera più sofisticata. In questo caso, si cerca di utilizzare il prefisso comune più lungo possibile tra chiavi adiacenti come separatore. L'obiettivo è minimizzare ulteriormente lo spazio occupato e migliorare l'efficienza delle operazioni

- In un B<sup>+</sup>-tree, nel caso di chiavi alfanumeriche la scelta dei separatori è particolarmente importante, in quanto, facendo uso di separatori di lunghezza ridotta si risparmia spazio e si riduce, eventualmente, l'altezza dell'albero.
- Esistono molte varianti in cui si esegue una “compressione” dei separatori; tra queste si ricordano i Simple Prefix B<sup>+</sup>-tree (di cui la figura è un esempio) e i Prefix B<sup>+</sup>-tree.



Un esempio di B+-tree che utilizza questa tecnica di compressione è il Simple Prefix B+-tree. In questo approccio, invece di memorizzare l'intera chiave come separatore, si memorizza solo il prefisso più corto sufficiente a distinguere tra i sottoalberi. Questo riduce significativamente la quantità di spazio necessaria per i nodi interni.

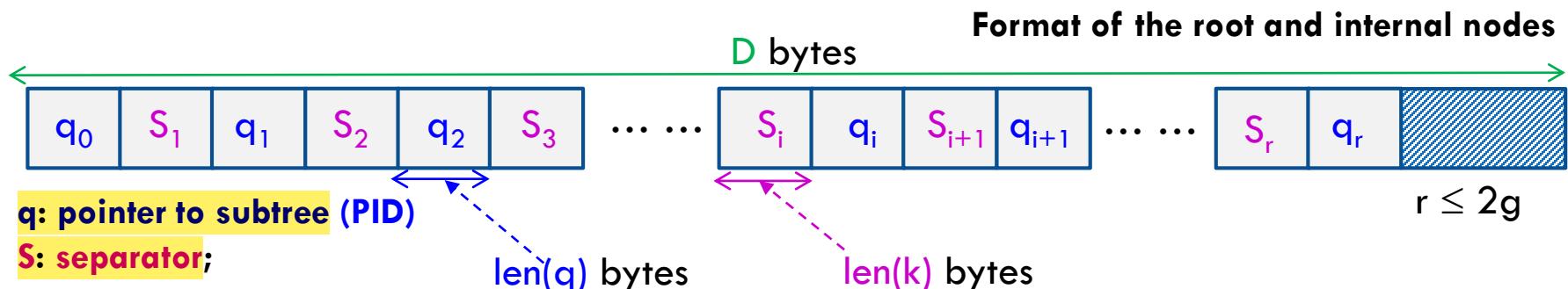
# Esempio B<sup>+</sup>-tree con prefissi



Fonte: [Oracle](#)

# Ordine di un B<sup>+</sup>-tree

- In un B<sup>+</sup>-tree l'ordine è un concetto ancora significativo solo se si fa uso di separatori di lunghezza fissa.
- Negli altri casi si può considerare, con un certo grado di approssimazione, la lunghezza media dei separatori stessi.
- Si supponga che la dimensione di un nodo sia fissata a  $D$  byte (es. 4096 byte) e che ogni puntatore,  $q$ , a nodi del B<sup>+</sup>-tree richieda  $\text{len}(q)$  byte. Si supponga inoltre che i separatori siano gli stessi valori di chiave, di lunghezza  $\text{len}(k)$  byte. Assumiamo per semplicità il formato di un nodo non foglia rappresentato in figura:



# Ordine di un B<sup>+</sup>-tree: derivazione

- Con le ipotesi fatte, poiché ogni nodo non foglia può ospitare al massimo **2g** separatori e **(2g+1)** puntatori q a nodi figli, si ha:

Calcola il Numero Tot  
di Byte per Separatori

$$2g \times \text{len}(k) + (2g + 1) \times \text{len}(q) \leq D$$

Calcola il Numero Tot  
di Byte per Puntatori

- Allora si deriva, trascurando l'header della pagina, che l'ordine di un B<sup>+</sup>-tree è:

$$g = \left\lfloor \frac{D - \text{len}(q)}{2(\text{len}(k) + \text{len}(q))} \right\rfloor$$

- Con pagine di **D** = 4096 byte, **len(q)** = 4 byte, e chiavi con **len(k)** = 10 caratteri, si ha un B<sup>+</sup>-tree di ordine **g** = 146, con 4 byte inutilizzati in ogni nodo. Con chiavi di 40 caratteri l'ordine si riduce a 46, con 42 byte inutilizzati in ogni nodo.

# Primary B<sup>+</sup>-tree: numero foglie

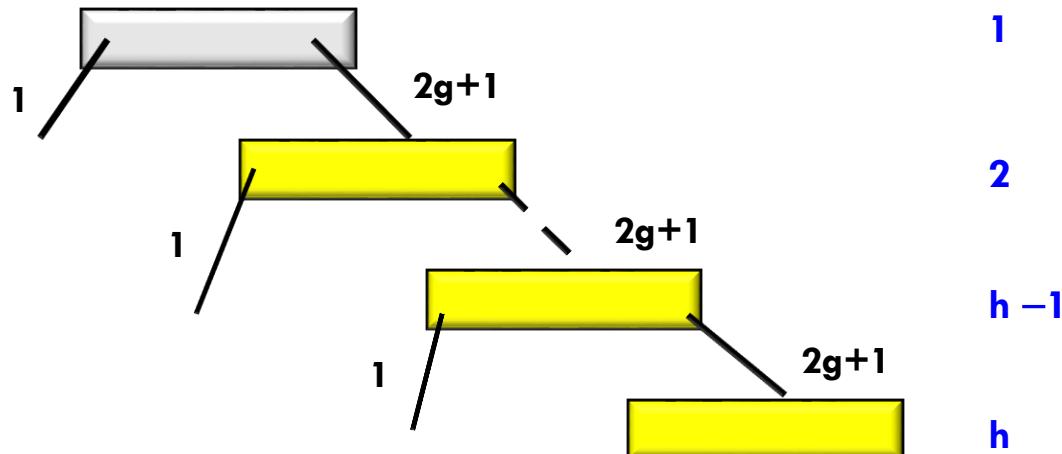
- In un B<sup>+</sup>-tree per chiave primaria (o chiave alternativa in assenza di NULL) il numero delle foglie dipende dal numero dei record, NR, presenti nel file, dalla dimensione dei nodi, D, e dall'utilizzazione delle foglie stesse.
- Si può dimostrare (e si verifica sperimentalmente) che l'utilizzazione media delle foglie, u, è pari a circa  $\ln 2 \approx 0.69$ .
- Trascurando, per semplicità, il puntatore alla foglia successiva (e, se presente, il puntatore alla foglia precedente), e considerando chiavi di lunghezza len(k) e puntatori ai record dati (RID) di lunghezza len(p), il numero di foglie, NL, si può approssimare come:

$$NL = \left\lceil \frac{NR \times (len(k) + len(p))}{D \times u} \right\rceil$$

# Primary B<sup>+</sup>-tree: altezza minima

- Si ottiene quando tutti i nodi dei livelli intermedi (radice compresa) sono pieni. Poiché al penultimo livello sono necessari almeno NL puntatori alle foglie, si ottiene:

$$(2g+1)^{h-1} \geq NL$$



da cui si ricava

$$h \geq 1 + \lceil \log_{2g+1} NL \rceil$$

# Primary B<sup>+</sup>-tree: altezza massima

- Nel caso peggiore la radice contiene 2 soli puntatori e ogni nodo dei livelli intermedi ne contiene  $g+1$ , da cui segue:

$$2 \times (g + 1)^{h-2} \leq NL$$

h-2 perché togli  
dal calcolo sia le  
foglie sia la radice

e quindi

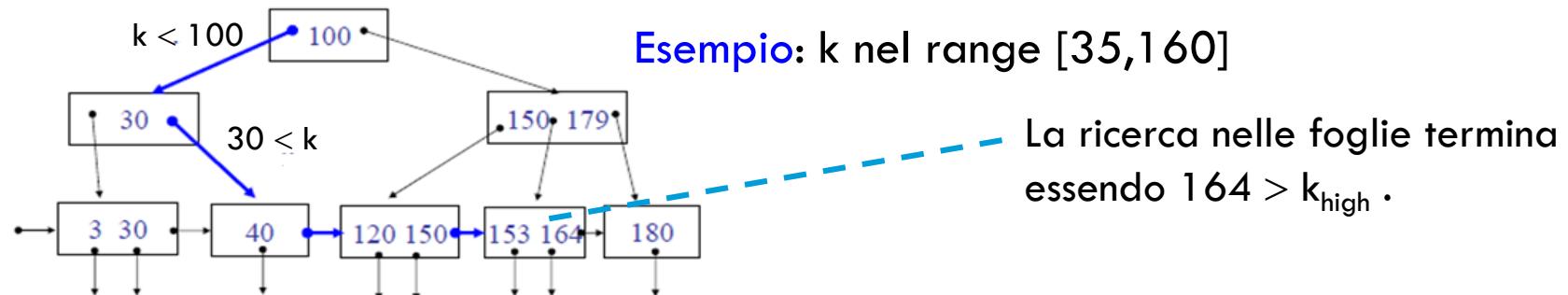
$$h \leq 2 + \left\lceil \log_{g+1} \frac{NL}{2} \right\rceil$$

- Ne consegue che l'altezza  $h$  di un primary B<sup>+</sup>-tree di ordine  $g$  è:

$$1 + \left\lceil \log_{2g+1} NL \right\rceil \leq h \leq 2 + \left\lceil \log_{g+1} \frac{NL}{2} \right\rceil$$

# Ricerca di valori in un primary B+-tree

- Il costo della ricerca di un singolo valore di chiave, misurato in numero di nodi visitati, è sempre pari all'altezza  $h$ . A questo costo si deve aggiungere 1 per accedere alla pagina ove risiede il record nel caso in cui i record non sono memorizzati direttamente nelle foglie.
- Dunque, per il caso medio, il costo di ricerca nell'indice è superiore a quello che si ha con un B-tree, e ciò è dovuto al fatto che i valori di chiave sono nelle foglie e i livelli superiori contengono solo separatori.
- Sia  $NK$  il numero di valori distinti di chiave; nel caso di una **query di range**  $[k_{low}, k_{high}]$  la ricerca si avvale delle sequenze memorizzate nelle foglie che sono tra loro concatenate.



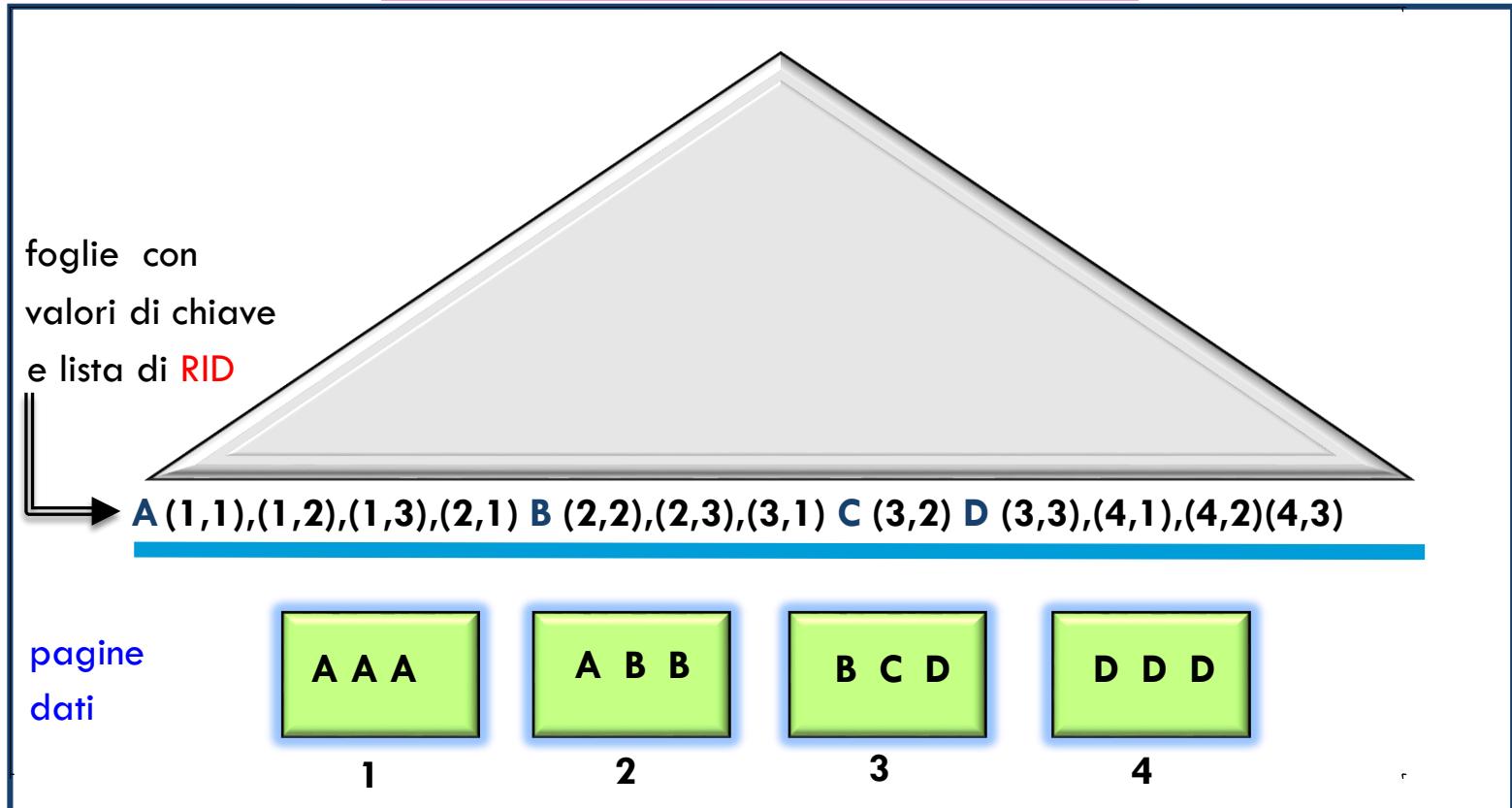
# Secondary B<sup>+</sup>-tree

- B<sup>+</sup>-tree a RID (TID)
  - In un B<sup>+</sup>-tree per chiavi secondarie, per ogni valore di chiave “si gestisce” nelle foglie una lista di puntatori (RID: Record Identifier detto anche TID: Tuple Identifier) ai record con quel valore (questa soluzione è anche nota come inverted index).
  - La lista di RID, per ogni singolo valore di chiave, è normalmente mantenuta ordinata per valori crescenti, sia nel caso di indice clustered sia nel caso di indice unclustered. Ciò consente di minimizzare i costi d’accesso alle pagine dati che contengono record con quel valore di chiave.
- B<sup>+</sup>-tree a PID
  - Anziché fare uso di RID, si impiegano identificatori di pagina (PID: Page Identifier).
  - Per un valore di chiave, la lista consiste di tanti PID quante sono le pagine del file dati che contengono almeno un record con quel valore di chiave.
  - È pertanto una soluzione particolarmente conveniente nel caso in cui molti record con lo stesso valore di chiave si trovino nella stessa pagina (es. indice clustered), o comunque quando vi sono molte pagine che contengono addensamenti di record con lo stesso valore di chiave.

# Secondary clustered B<sup>+</sup>-tree (RID)

B<sup>+</sup>-tree a RID

Indice clustered su chiave secondaria

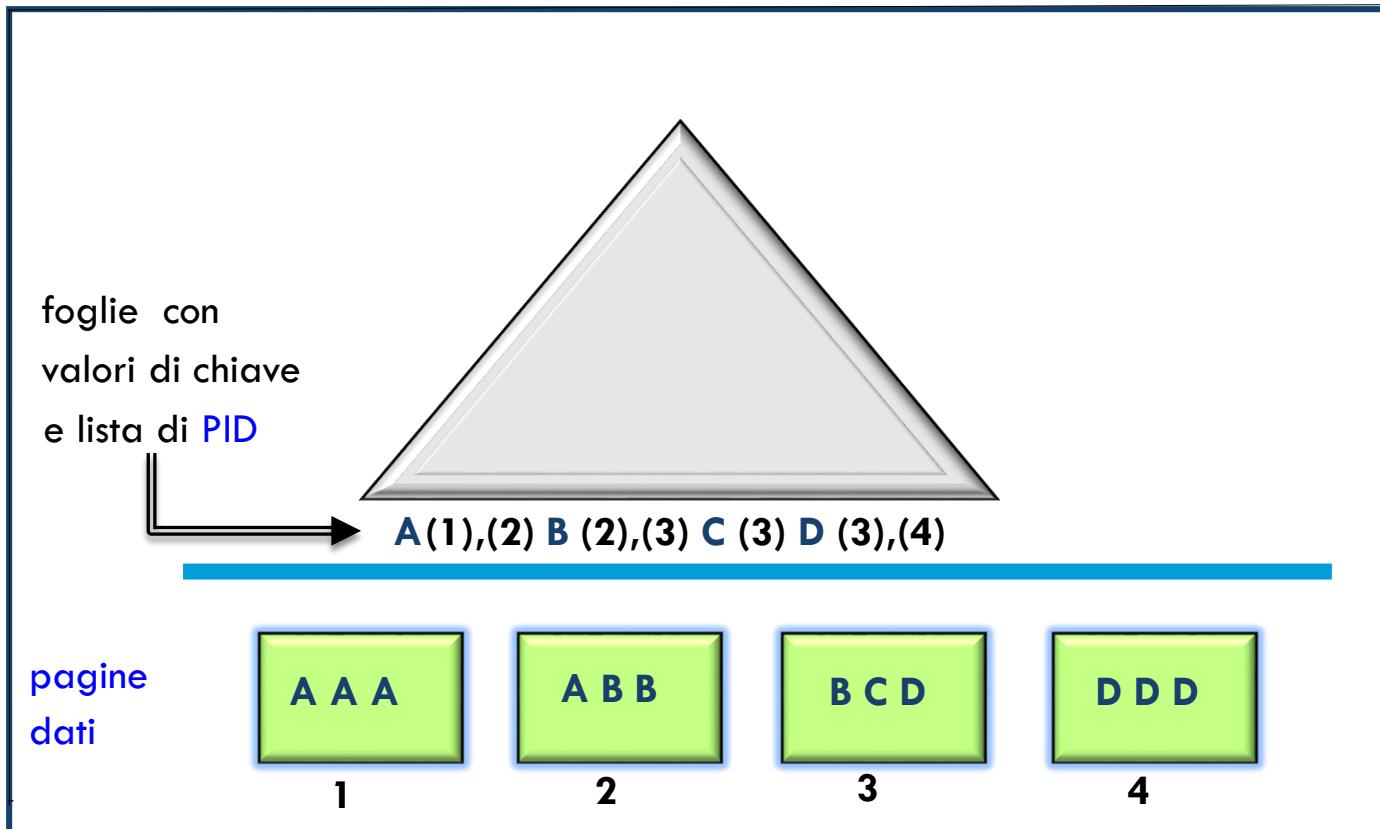


N.B. In questo schema semplificato e nei successivi un RID è visualizzato come coppia (numero di pagina logica - PID, numero del record all'interno della pagina).

# Secondary clustered B<sup>+</sup>-tree (PID)

B<sup>+</sup>-tree a PID

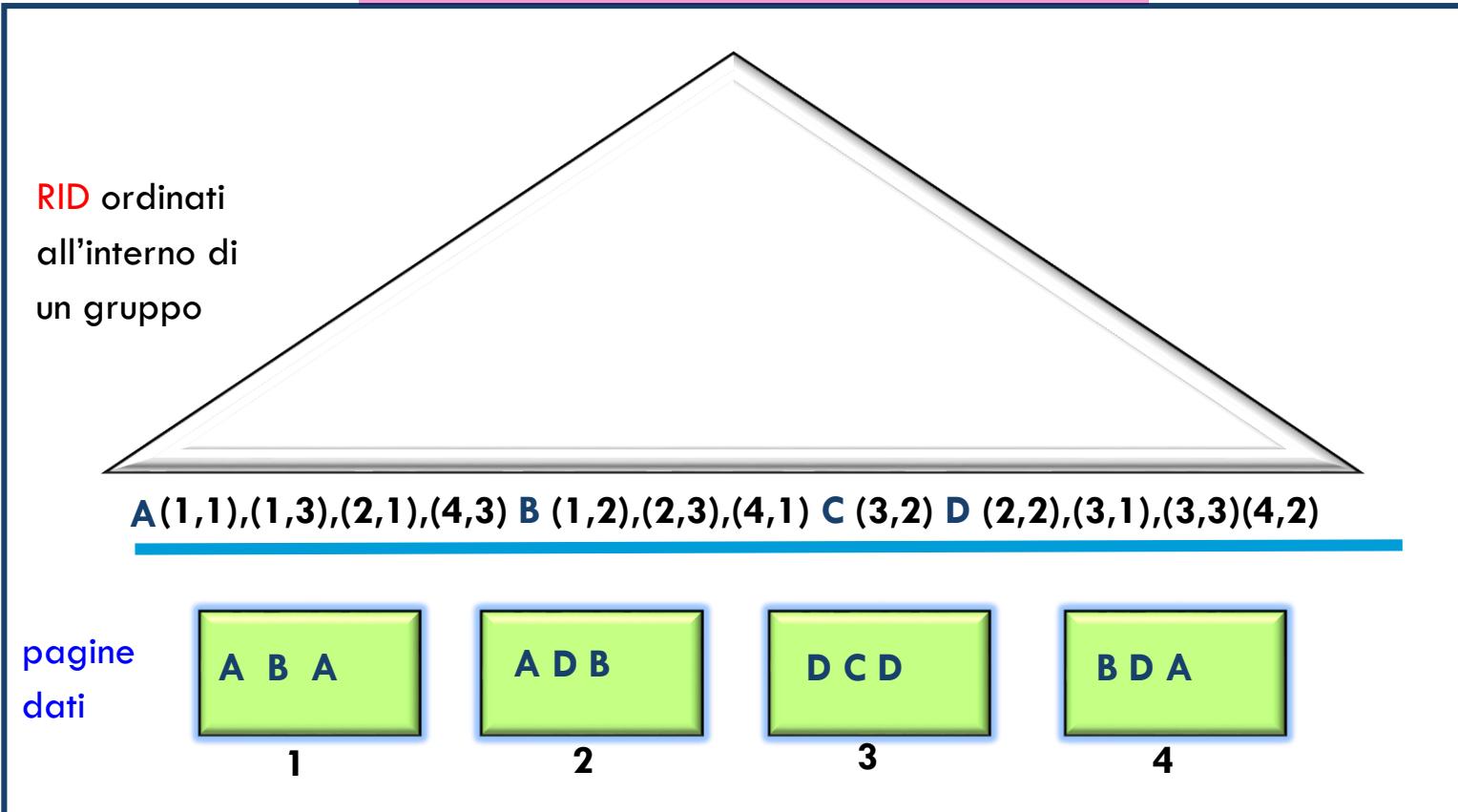
Indice clustered su chiave secondaria



# Secondary unclustered B<sup>+</sup>-tree (RID)

## B<sup>+</sup>-tree a RID

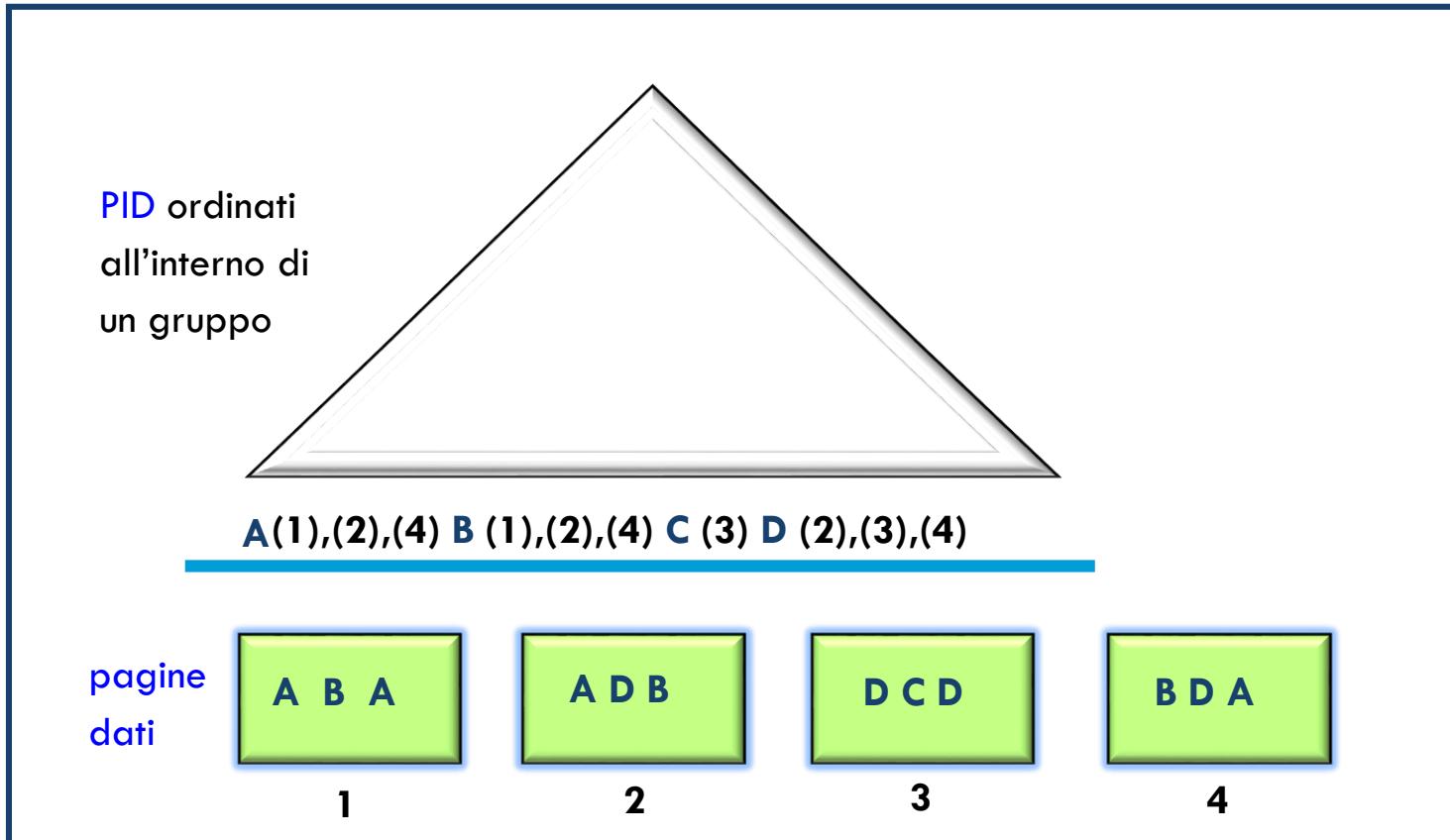
Indice unclustered su chiave secondaria



# Secondary unclustered B<sup>+</sup>-tree (PID)

## B<sup>+</sup>-tree a PID

Indice unclustered su chiave secondaria



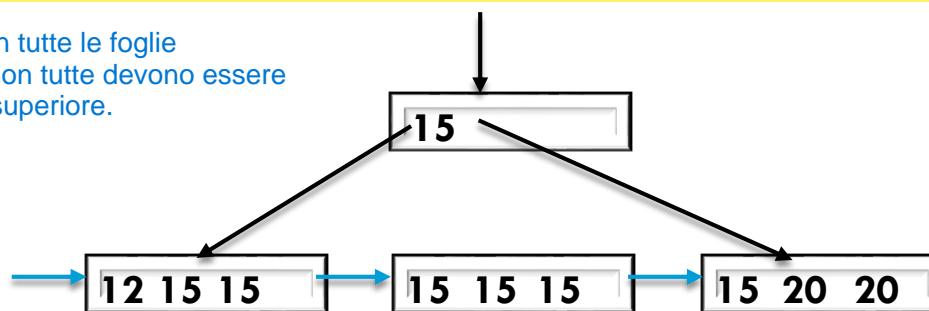
# Secondary B<sup>+</sup>-tree: numero di foglie

- (B<sup>+</sup>-tree a RID): il numero delle foglie dipende ora anche dal numero di valori di chiave distinti, NK, in quanto (sempre trascurando i puntatori alla foglia precedente e successiva) si può stimare come:

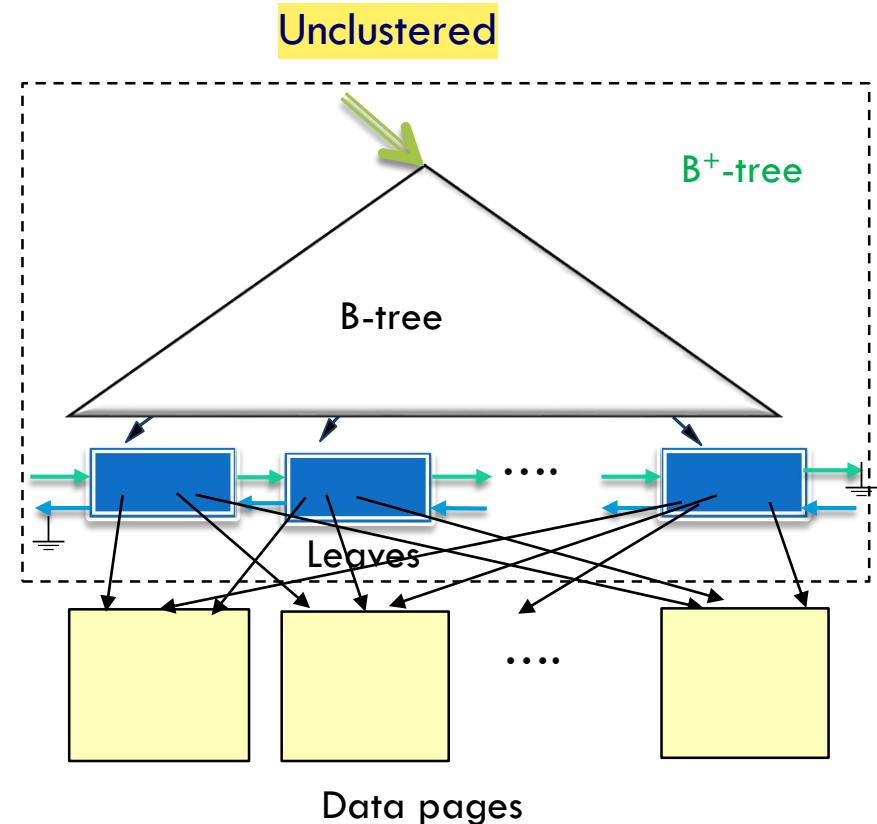
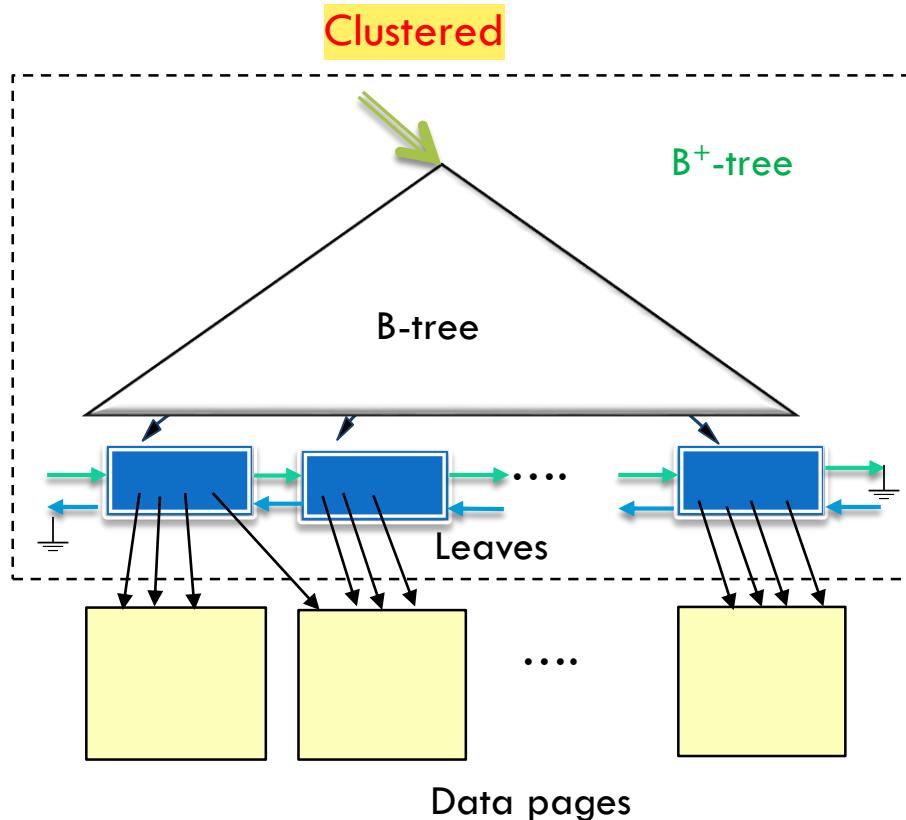
$$NL = \left\lceil \frac{NK \times \text{len}(k) + NR \times \text{len}(p)}{D \times u} \right\rceil$$

- Considerando  $D^* < D$  per tener conto dello spazio per l'header e per i puntatori alla foglia successiva e alla precedente si ha una stima migliore.
- È importante considerare che una foglia deve essere indirizzata dal livello superiore solo se contiene RID di un “nuovo” valore di chiave. Per calcolare l’altezza è quindi necessario usare  $\min\{NK, NL\}$  in luogo di NL.

L’altezza dipende dal fatto che non tutte le foglie contengono nuove chiavi, quindi non tutte devono essere raggiunte direttamente dal livello superiore.



# Clustered vs Unclustered B<sup>+</sup>-tree



# Domande?

