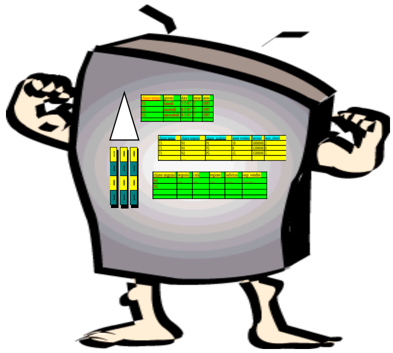
Organizzazioni dei dati – parte II



Dario Maio

http://bias.csr.unibo.it/maio/

Organizzazioni primarie e secondarie

- **Considerazioni sull'efficienza della ricerca dicotomica**:
  - Per accedere velocemente a uno o più record, facendo uso di una chiave di ricerca, è possibile mantenere ordinato il file secondo i valori della chiave. In questo caso, tuttavia, si hanno i seguenti due problemi:
  1. il costo di ricerca è proporzionale a $\log_2 NP$, e quindi elevato per file di grandi dimensioni (cercare un record in un file di 2^{10} blocchi richiede circa 0.25 sec, se $T_{I/O}$ = 25 msec).
  2. la ricerca su altri campi è estremamente inefficiente.

- **Soluzioni:**
  - **Organizzazioni primarie**

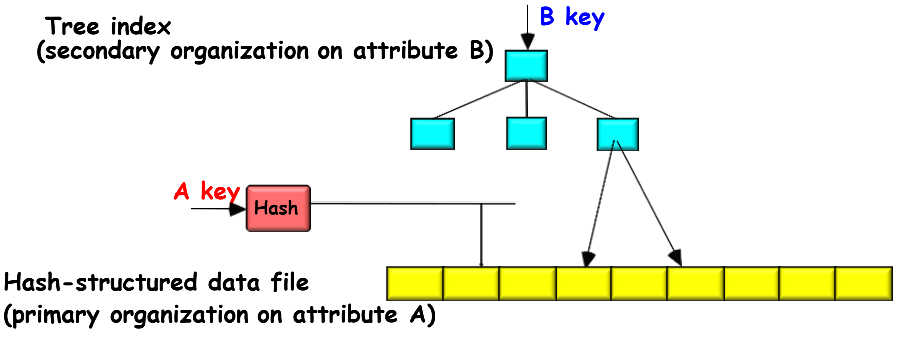
    I dati sono organizzati in modo tale da ridurre drasticamente i costi, ricorrendo a organizzazioni (tipicamente per chiave primaria) ad **albero** o basate su tecniche **hash**.

  - **Organizzazioni secondarie**

    Si fa ricorso a **indici** (separati dal file dati) che sono normalmente organizzati **ad albero** (ma sono anche possibili **indici hash**).

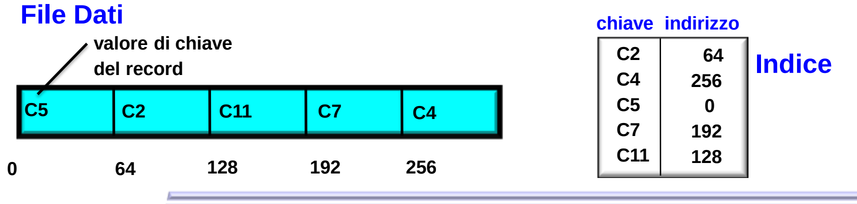
Cammini di accesso

- In generale si dispone di più cammini di accesso ai dati.



Indici: idea di base

- 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** del campo 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.
- I diversi indici differiscono essenzialmente nel modo con cui organizzano l'insieme {(k_i,p_i)} di coppie.

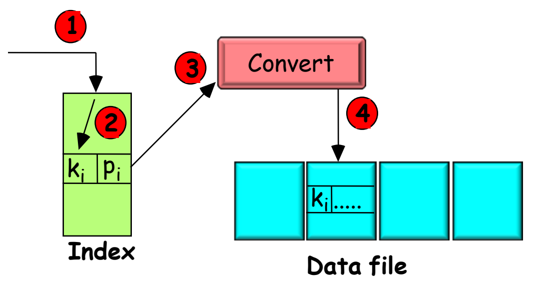


Accesso con indice

- Si consideri un **indice su chiave primaria**

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



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.

Ricerca binaria su indice ordinato

- 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.
- Il limite di questa soluzione, che ne prova l'inadeguatezza per file di grandi dimensioni, deriva dal fatto che, rispetto al caso di ricerca binaria sul file dati,

il numero di accessi che si risparmiano è una costante che non dipende dalla dimensione del file dati, ma solo dalla lunghezza dei record e della chiave.

Costo ricerca binaria

- Il file dati è memorizzato in NP blocchi di capacità C record, e l'indice è memorizzato in un file relative di IP blocchi di capacità IC (> C). Vale la relazione:

$$C \times NP = IC \times IP$$

in quanto $C \times NP$ è il numero di record, che coincide con il numero di coppie (k_i,p_i) nell'indice.

- Poiché il costo di reperimento di un record, facendo uso di ricerca binaria sull'indice è (*si omette per semplicità l'arrotondamento*):

  $\log_2 IP$ (accessi a blocchi indice) + 1 (accesso a blocco dati)

  il risparmio rispetto a una ricerca dicotomica su file ordinato risulta essere pari a:

$$\log_2 NP - (\log_2 IP+1) = \log_2 NP - \log_2((NP \times C)/IC) - 1 = \log_2 (IC/C) - 1$$

Tipi di indice: una classificazione

unicità dei valori di chiave	Primary index	indice su un attributo che assume valori unici
	Secondary index	indice su un attributo che può assumere valori ripetuti
ordinamento del file dati	Clustered index	indice su un attributo secondo cui il file dati è mantenuto ordinato
	Unclustered index	indice su un attributo secondo cui il file dati non è mantenuto 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)

Possibili combinazioni (1)

- Quasi tutte le $2^4 = 16$ combinazioni di tipi di indice sono possibili. In principio, l'unica incompatibilità è data da **sparse & unclustered** che non permetterebbe di reperire i record i cui relativi riferimenti non sono nell'indice.

primary clustered sparse single-level index

valore di chiave	puntatore a blocco
160229323	
160239980	
160240576	
……	

160229323	BNCGRG78L21A944K	Bianchi	Giorgio
160235467	RSSNNA78A53A944V	Rossi	Anna
160239654	VRDMRC79H20F839X	Verdi	Marco
160239980	VRDMRT78L66A944K	Verdoni	Marta
160240467	…………………	……….	……….
160240532	…………………	……….	……….
160240576	…………………	……….	……….
160240600	…………………	……….	……….
160240623	…………………	……….	……….

blocchi dati

Possibili combinazioni (2)

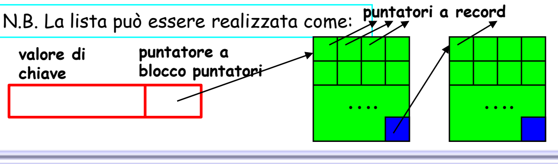
secondary unclustered dense single-level index

valore di chiave	lista puntatori a record
Alessandri	
Bianchi	
Bianchini	
… … …	
Rossi	
……	
Verdi	
Verdoni	
…… …	

blocchi dati

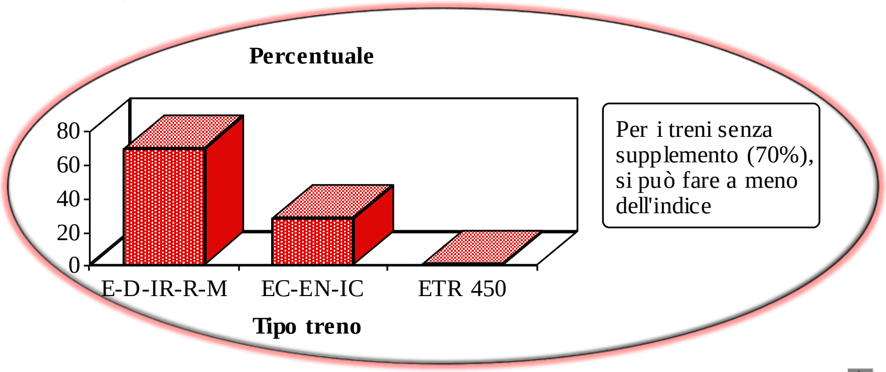
160229323	BNCGRG78L21A944K	Bianchi	Giorgio
160235467	RSSNNA78A53A944V	Rossi	Anna
160239654	VRDMRC79H20F839X	Verdi	Marco
160239980	VRDMRT78L66A944K	Verdoni	Marta
160240467	…………………	Alessandri	Maria
160240532	…………………	Bianchini	Carlo
…………………	…………………	…………………	…………………
160240576	…………………	Rossi	Demetrio
160240600	…………………	Bianchi	Arrigo
160240623	…………………	Verdi	Remo

N.B. La lista può essere realizzata come:



Possibili combinazioni (3)

- Una soluzione **sparse & unclustered** particolare, nota col nome di **partial index**, **per un indice secondario**, 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.



Record Pointer vs Block Pointers

SUPPNO	PTNO	PTYPE	SHELF	
S2	P4	SCREW	A1	1,1
S6	P7	NUT	A1	1,2
S3	P1	SCREW	A1	1,3
S2	P1	SCREW	A1	1,4
S6	P2	NUT	A1	2,1
S1	P3	BOLT	A2	2,2
S5	P11	CAM	A2	2,3
S3	P3	NUT	A2	2,4
S5	P12	CAM	A2	3,1
S5	P3	BOLT	A3	3,2
S3	P5	BOLT	A3	3,3
S4	P11	CAM	A4	3,4

N° blocco

N° record nel blocco

indice su PTYPE con puntatori 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)

Alcune organizzazioni notevoli

- Si ottengono nel caso **primary clustered** **sparse** **multi-level**

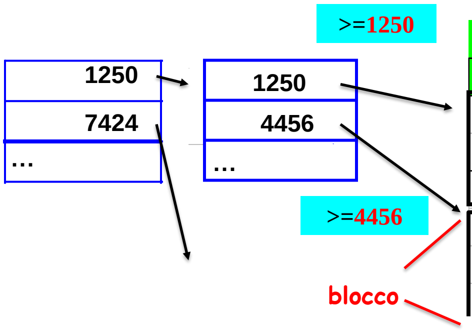
	Clustered Index	Unclustered Index
Dense Index	Possible	PISM
Sparse Index	ISAM VSAM UFAS	Not possible

PISM = Pure Indexed Sequential Method (heap + indice)
ISAM = Indexed Sequential Access Method (IBM)
VSAM = Virtual Storage Access Method (IBM) (evoluzione di ISAM)
UFAS = Regular Indexed Sequential (Bull)

- In queste organizzazioni l'indice può essere parte integrante del file dati, nel senso che i blocchi dell'indice sono allocati in modo non indipendente dai corrispondenti blocchi del file dati.

ISAM

- **ISAM** (**Indexed Sequential Access Method**): struttura usata negli ambienti DOS e OS/VS IBM, comprende un file dati ordinato sul valore della chiave primaria e un indice non denso a più livelli.

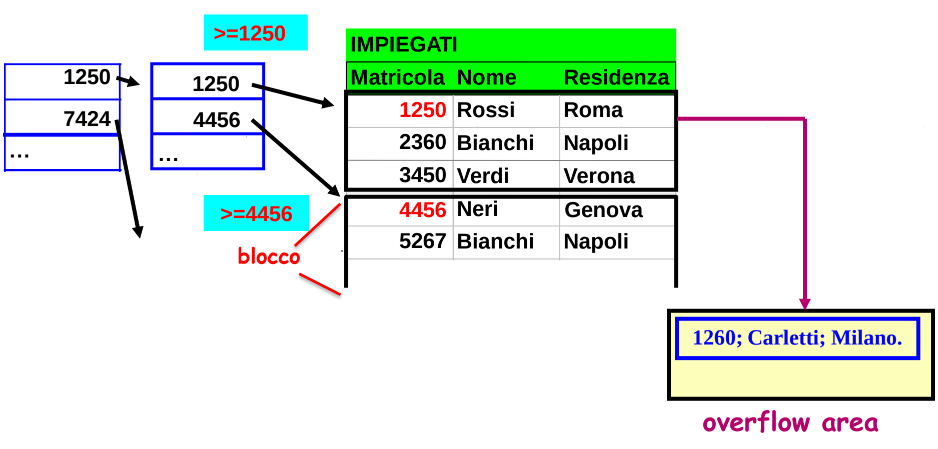


IMPIEGATI		
Matricola	Nome	Residenza
1250	Rossi	Roma
2360	Bianchi	Napoli
3450	Verdi	Verona
4456	Neri	Genova
5267	Bianchi	Napoli

ISAM: considerazioni

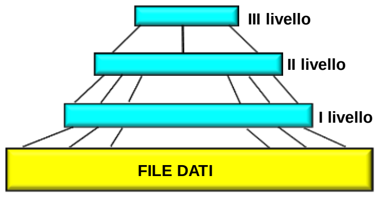
- ISAM è un'organizzazione statica, soggetta quindi a (costose) riorganizzazioni periodiche.
- Per mantenere l'ordinamento dei dati, all'atto del caricamento dei record (in ordine crescente di valore di chiave) sono lasciati spazi liberi per ulteriori inserimenti.
- Nel caso gli spazi liberi non siano sufficienti, si fa uso di un'**area di overflow**.
- Ogni coppia (k_i,p_i) dell'indice è tale per cui k_i è il più basso valore di chiave nel sottoalbero individuato da p_i.
- In alcuni sistemi di vecchia generazione le strutture ISAM erano ottimizzate a livello di disco: foglie vicine risiedono su una stessa traccia o su tracce adiacenti; i livelli alti di una struttura ISAM non erano mai modificati, e quindi assenza di lock sull'albero.

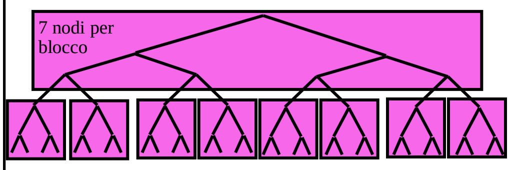
ISAM: esempio di inserimento



Indici multilivello

- Un indice in memoria secondaria è, per ragioni di efficienza, di solito organizzato in più livelli.



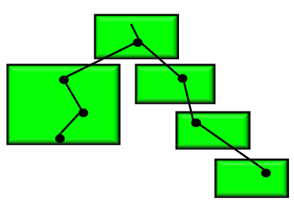


- In memoria centrale la soluzione più comune per un indice si basa su **AVL** (**alberi binari bilanciati**). La "**paginazione**" di alberi non è adeguata dal punto di vista dinamico, in quanto:

1. 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;
2. 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 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

- Esistono molte varianti, tra cui il **B-tree** "vero e proprio", il B*- tree e il B⁺-tree.

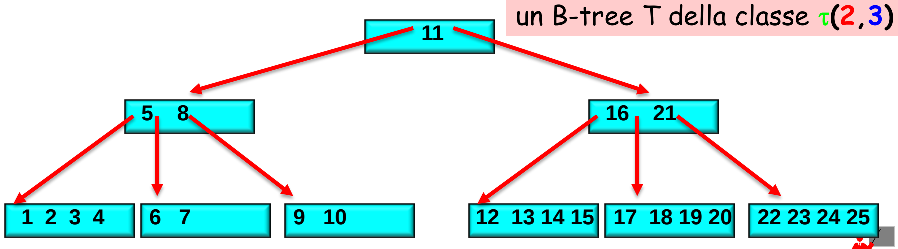
- Un B-tree è un albero (**direzionato**) a più vie **perfettamente bilanciato organizzato a nodi**, che corrispondono ora a **blocchi di disco** (il caso in cui un nodo corrisponde a più blocchi sarà trattato in seguito).

B-tree: definizione

- Siano g,h > 0 due numeri naturali, detti rispettivamente **ordine** e **altezza** del B-tree. Un B-tree T della classe τ(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 = 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.
3. Ogni nodo ha al più 2g+1 nodi figli.

un B-tree T della classe τ**(2,3)**



B-tree: organizzazione di un nodo

- **Un B-tree è organizzato a nodi (o pagine logiche):**
  1. Ogni nodo memorizza tra g e $2g$ chiavi, eccetto la radice che può avere da 1 a $2g$ chiavi.
  2. Un nodo interno (non foglia) con l chiavi ($g \le l \le 2g$) ha $l+1$ puntatori ad altrettanti nodi figli.
  3. In ogni nodo le chiavi sono memorizzate in ordine crescente.

q_0	k_1	p_1	q_1	k_2	p_2	q_2	……	k_i	p_i	q_i	……	spazio non usato

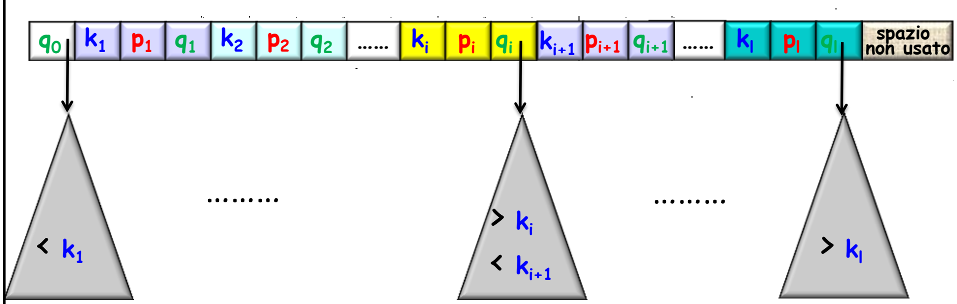
k_i: valore di chiave
p_i: puntatore al record con valore di chiave k_i
q_i: puntatore a un nodo figlio

- *N.B. Nel caso di indice su valori ripetuti, il formato del nodo deve essere adattato per consentire di memorizzare il grado di molteplicità di k_i e la lista di puntatori a record con valore di chiave k_i.*

B-tree: insieme dei valori

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

$$\forall y \in K(q_0): \quad (y < k_1)$$
$$\forall y \in K(q_i): \quad (k_i < y < k_{i+1}) \quad i = 1,2,\dots,l-1$$
$$\forall y \in K(q_l): \quad (k_l < y)$$



Ricerca in un B-tree

$P(q)$	il nodo puntato da q
$k_1,\ldots,k_l$	le chiavi in $P(q)$
$q_0,\ldots,q_l$	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:

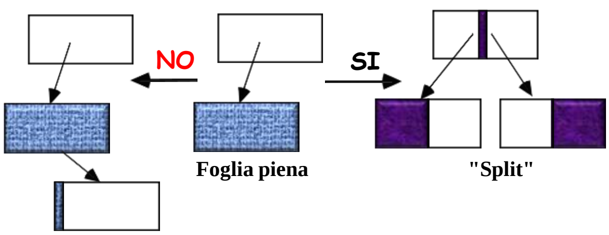
$$1 \le C(search) \le h$$

```
{q:=root;
 s:=nil;
 trovata:=false;
 while (q≠nil) and (not trovata) do
      { s:=q;
        if y<k1 then q:=q0
             else if  ∀ i (y=ki)
                    then trovata:=true
                    else if  ∀ i (ki<y<ki+1)
                           then q:=qi
                           else q:=ql
      }
}
```


Evoluzione di un B-tree

- 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**.



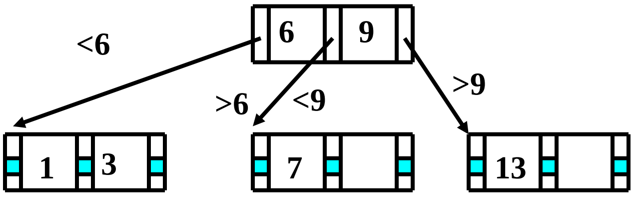
Inserimento di chiavi nel B-tree

- 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 (**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 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.**

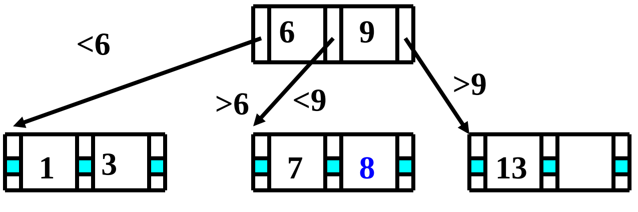
```
{ ricerca la chiave y;
     if (not trovata)
     then { if s=nil
                  then crea la radice con y
                  else if  P(s) è pieno
                           then attiva splitting
                           else inserisci (y,p,nil) in P(s)
            }
}
```


Caso 1: foglia non piena

- Prima dell'inserimento della chiave **8**

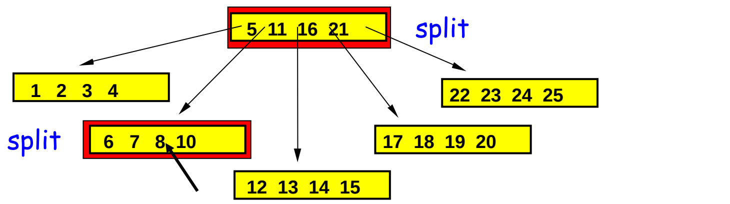


- Dopo l'inserimento:

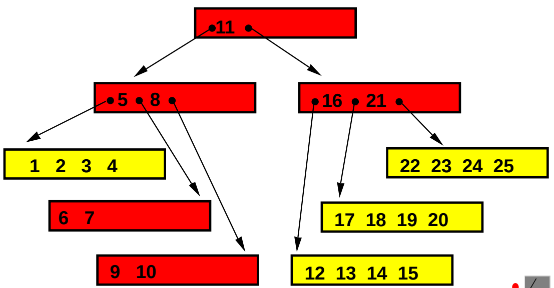


Caso 2: foglia piena (splitting)

- Inserimento della chiave **9** nel B-tree $T \in \tau(2,2)$

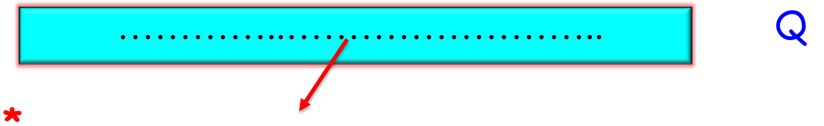


- B-tree $T \in \tau(2,3)$ dopo l'inserimento della chiave **9**



Splitting di un nodo (1)

- 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 (si omettono i puntatori p_i)

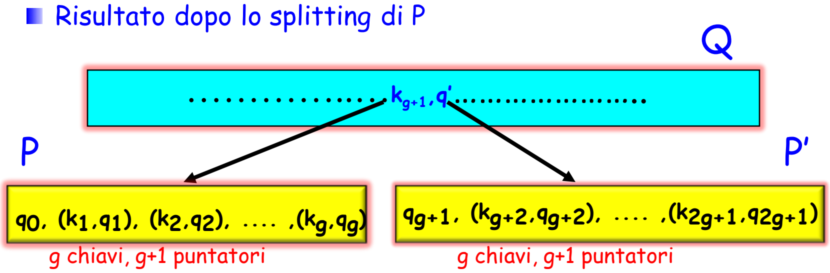


* $q_0,(k_1,q_1),(k_2,q_2),\ldots,(k_g,q_g),(k_{g+1},q_{g+1}),(k_{g+2},q_{g+2}),\ldots,(k_{2g+1},q_{2g+1})$ P

- 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 di un nodo (2)

- Risultato dopo lo splitting di P

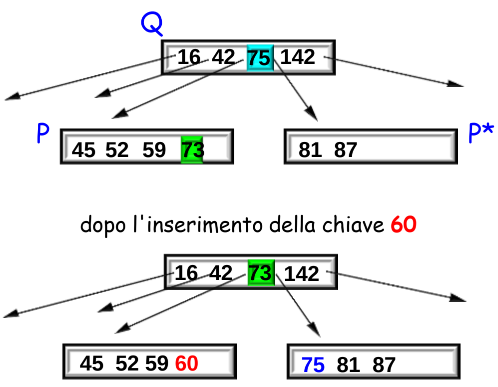


- Se anche Q risulta pieno e non riesce a ospitare l'aggiunta di un'entrata, il processo di splitting si propaga.
- Se il nodo da sdoppiare è la radice del B-tree, allora la nuova radice Q conterrà (q,k_{g+1},q') dove q punta a P e q' punta a P'.
- L'algoritmo di splitting preserva l'ordine g del B-tree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

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.

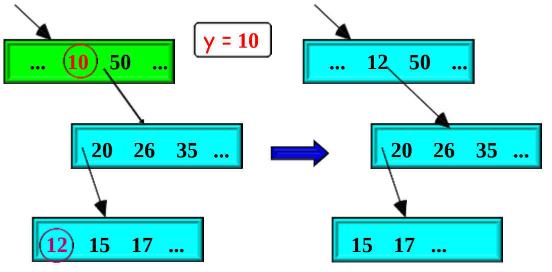
✓Un B-tree che adotta questa strategia è detto un B-tree che **gestisce l'overflow**.
✓La gestione dell'overflow tende a generare alberi con nodi più pieni, ma comporta, in generale, maggiori costi di inserimento.
✓La gestione dell'overflow può essere generalizzata, infatti si possono considerare per la ridistribuzione 3 o più nodi fratelli anziché 2.



Eliminazione di una chiave (1)

- 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**.

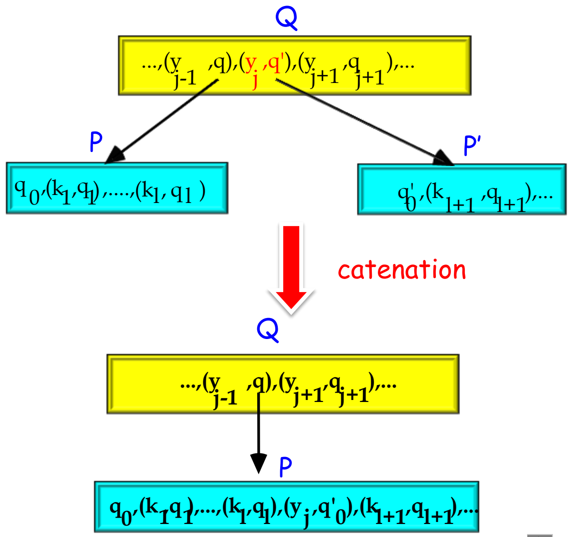


Eliminazione di una chiave (2)

```
{ 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;
                sostituisci a y la prima chiave di L;
                cancella la prima chiave di L
              };
        if  L contiene meno di g chiavi
          then attiva catenation e underflow
      }
}
```

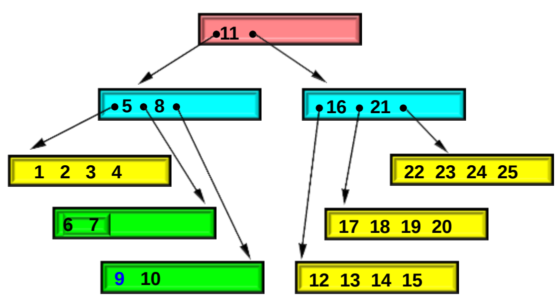
Catenation

- 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_j,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.

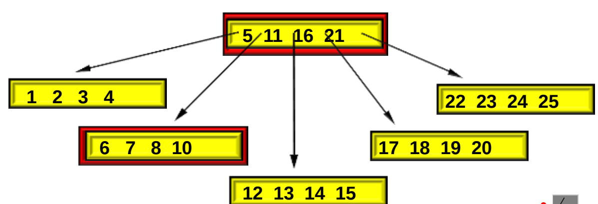


Esempio di catenation

- T∈τ(2,3) prima della cancellazione della chiave 9.



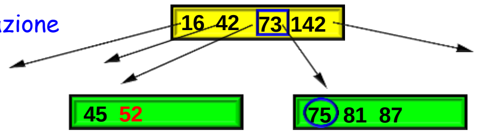
- T∈τ(2,2) dopo la cancellazione della chiave 9.



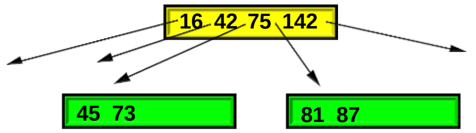
Underflow

- 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**



Prestazioni di un B-tree

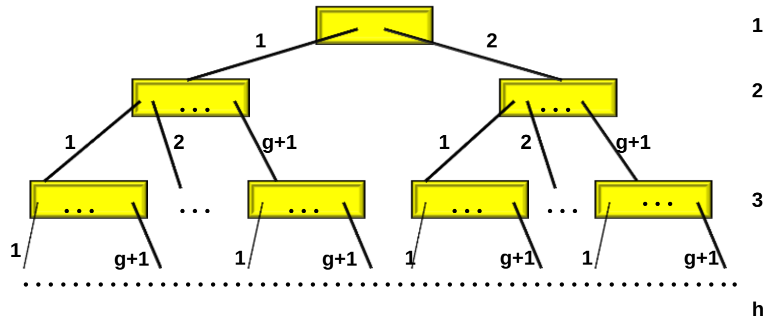
- Dipendono da vari fattori, tra loro correlati, quali:
  - altezza,
  - ordine,
  - gestione della cache,
  - caratteristiche del dispositivo,
  - gestione della concorrenza,...
- Per quanto riguarda la gestione della cache è desiderabile che un qualunque nodo esaminato durante una singola operazione di ricerca, inserimento o cancellazione di una chiave, venga rispettivamente letto e/o scritto una sola volta. A tale scopo è sufficiente prevedere in memoria centrale un buffer per h+1 nodi. Una tecnica comune consiste nel mantenere comunque la radice sempre in memoria centrale

notazione

NR	numero di record
NP	numero di blocchi del file dati
NK	numero di valori distinti di chiave dell'attributo indiciato = NR se l'attributo è chiave primaria e indice denso
IP	numero di nodi dell'indice
NL	numero di nodi foglia

Numero nodi di un B-tree (1)

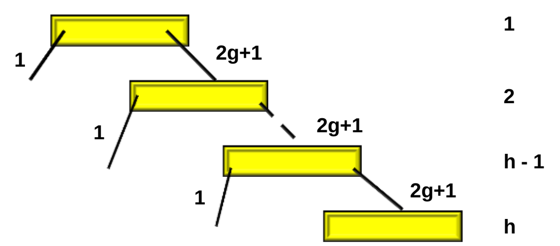
- numero minimo di nodi IP_{min} di un albero $T \in \tau(g,h)$



$$IP_{min} = 1 + 2 + 2(g+1) + 2(g+1)(g+1) + \ldots + 2(g+1)^{h-2} =$$
$$= 1 + 2((g+1)^0 + (g+1)^1 + \ldots + (g+1)^{h-2}) =$$
$$= 1 + 2\sum_{i=0}^{h-2}(g+1)^i = 1 + 2\frac{(g+1)^{h-1} - 1}{(g+1) - 1} = 1 + \frac{2}{g}((g+1)^{h-1} - 1)$$

Numero nodi di un B-tree (2)

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



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

Altezza di un B-tree (2)

- caso peggiore: il minimo numero di chiavi presenti in un B-tree $T \in \tau(g,h)$ 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 $T \in \tau(g,h)$ 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:

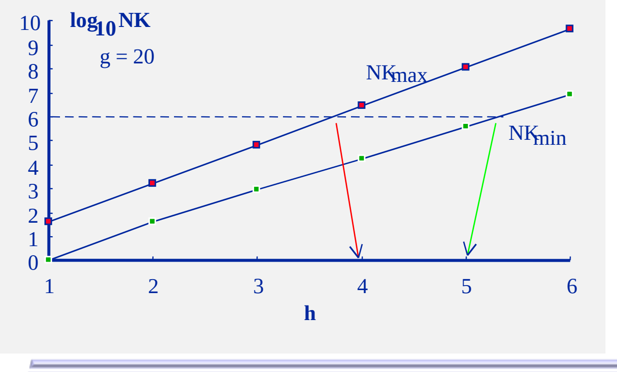
$$\left\lceil \log_{2g+1}(NK+1) \right\rceil \le h \le \left\lfloor 1 + \log_{g+1}\left(\frac{NK+1}{2}\right) \right\rfloor$$

Altezza di un B-tree (3)

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

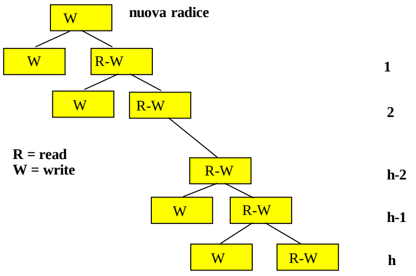
$$NK_{max} = 2g \times IP_{max} = (2g+1)^{h} - 1$$

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



Costo di un inserimento

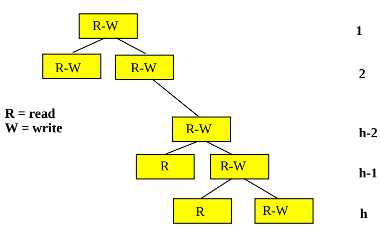
- caso migliore: in assenza di splitting; si leggono h nodi e si riscrive una foglia: $C_{min}(\text{insert}) = h+1$
- caso peggiore: quando lo splitting si propaga fino alla radice; si leggono h nodi e si riscrivono $2h+1$ nodi: $C_{max}(\text{insert}) = 3h+1$



- caso medio: per un puro processo di creazione, senza gestione dell'overflow, si deriva un limite superiore: $C_{avg}(\text{insert}) < h + 1 + \frac{2}{g}$

Costo di un'eliminazione

- caso migliore: cancellazione in una foglia con numero di chiavi residuo maggiore o uguale a g. $C_{min}(\text{delete}) = h+1$
- caso peggiore: si verifica quando tutte le pagine nel percorso di ricerca devono essere concatenate a eccezione delle prime due, il figlio della radice nel percorso subisce un underflow e la radice viene modificata. Deve essere eseguito in una foglia con g chiavi. Ciò richiede $2h-1$ letture e $h+1$ scritture: $C_{max}(\text{delete}) = 3h$



$$C_{avg}(\text{delete}) < h + 5 + \frac{3}{g}$$

- caso medio: per un puro processo di eliminazione, senza gestione dell'overflow, si deriva un limite superiore:

Gestione dell'overflow

- In un B-tree senza gestione dell'overflow, l'utilizzazione minima della memoria secondaria è pari al 50%. Per un puro processo di inserimento la gestione dell'overflow comporta nel caso peggiore una utilizzazione pari al 66%.
- Il limite teorico inferiore è ancora del 50% nel caso di cancellazioni e inserimenti, ma sperimentalmente l'utilizzazione media è comunque più elevata di quella di un B-tree che non gestisce l'overflow. Nel caso di gestione dell'overflow, Bayer e McCreight dimostrano che, per un puro processo di inserimento, in media si ha:

$$C_{avg}(\text{insert}) < h + 5 + \frac{4}{g}$$

- e sperimentalmente si osserva che il numero medio di operazioni di I/O nel caso di gestione dell'overflow è maggiore rispetto al caso in cui si esegue comunque lo split.
- Le prestazioni di un B-tree dipendono dall'interferenza tra inserimenti e cancellazioni, la cui analisi richiede la costruzione di un modello probabilistico. Bayer e McCreight mostrano che questa interferenza peggiora le prestazioni al più di un fattore 3.

Scelta dell'ordine (1)

- Nel caso in cui sia possibile accedere a più blocchi con una singola operazione di lettura, si pone il problema di determinare un valore appropriato per l'ordine del B-tree.
- Al crescere di g il numero di nodi e l'altezza dell'albero tendono a diminuire, e così i costi delle varie operazioni. Aumenta, viceversa, il costo di trasferimento di un nodo.
- Si considera un semplice modello che valuta il solo tempo di I/O. Il tempo speso per ogni nodo scritto o letto è circa pari a:

$$T_{I/O} \approx (t_s + t_r) + 2g \times t_b / (2g_1)$$

dove:

$t_s + t_r$:latenza

t_b :tempo di trasferimento di un blocco

g_1 :ordine nel caso nodo = blocco

$2g / (2g_1)$:n. blocchi/nodo in un B-tree di ordine g

Scelta dell'ordine (2)

- Il numero medio di pagine lette o scritte per ogni singola operazione è approssimativamente proporzionale a h. Pertanto il tempo totale per una operazione, T(op), può essere espresso come:

$$T(op) \propto h \times T_{I/O} \approx h \times [(t_s + t_r) + t_b \times 2g / (2g_1)]$$

- Approssimando **h** con $\log_{2\beta g+1}(NK+1)$, essendo β l'utilizzazione di un nodo ($0.5 \le \beta \le 1$), si ha:

$$T(op) \propto \log_{2\beta g+1}(NK+1) \times [(t_s + t_r) + t_b \times 2g / (2g_1)]$$

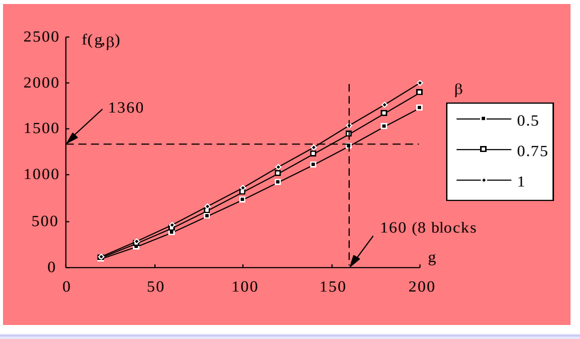
- L'ordine ottimale rispetto al modello dato dipende dalle caratteristiche del dispositivo di memoria secondaria e delle chiavi (a causa di g_1); il minimo di **T(op)** si ottiene scegliendo **g** in modo che sia:

$$\frac{t_s + t_r}{t_b} 2g_1 = \frac{(2\beta g + 1) \times \ln(2\beta g + 1)}{\beta} - 2g = f(g, \beta)$$

Esempio di scelta dell'ordine g

- Dato un disco con t_s = 11 msec
  t_r = 6 msec
  t_b = 0.5 msec (blocchi da 1 KB)

  e supponendo che sia $g_1 = 20$, si ottiene **f(g,β) = 1360**.
- Una scelta adeguata è pertanto avere nodi di dimensione pari a 8 blocchi (**g=160**).

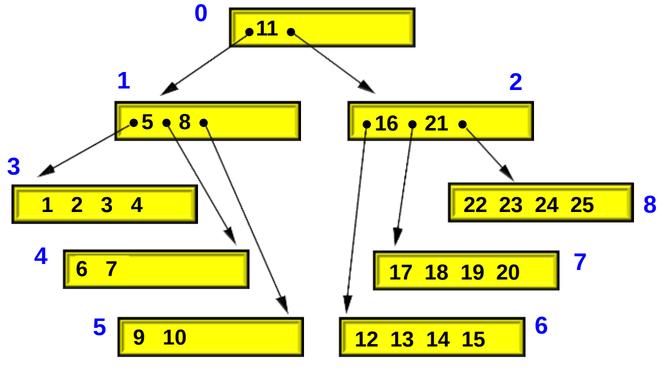


Pregi e difetti dei B-tree

- Un B-tree è molto efficiente per la ricerca e la modifica di singoli record. Ad esempio, con un B-tree di ordine $g=99$ e $NK=1999999$ chiavi, si ha: $h \leq 1+\log_{100} 10^6 = 4$. Pertanto la ricerca di una chiave comporta al massimo 4 accessi a disco.
- 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.
- 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.

Esempio ricerca in un intervallo

- Ricercare tutti i record con valori di chiave $6 \leq k \leq 19$ implica l'accesso ai nodi 0, 1, 4, 1, 5, 1, 0, 2, 6, 2, 7.



B* - tree

- 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 entrambi 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.
- Il termine B*-tree a volte è stato usato per indicare un'altra famosa variante del B-tree suggerita da Knuth, che in questo contesto è invece chiamata B+ -tree.