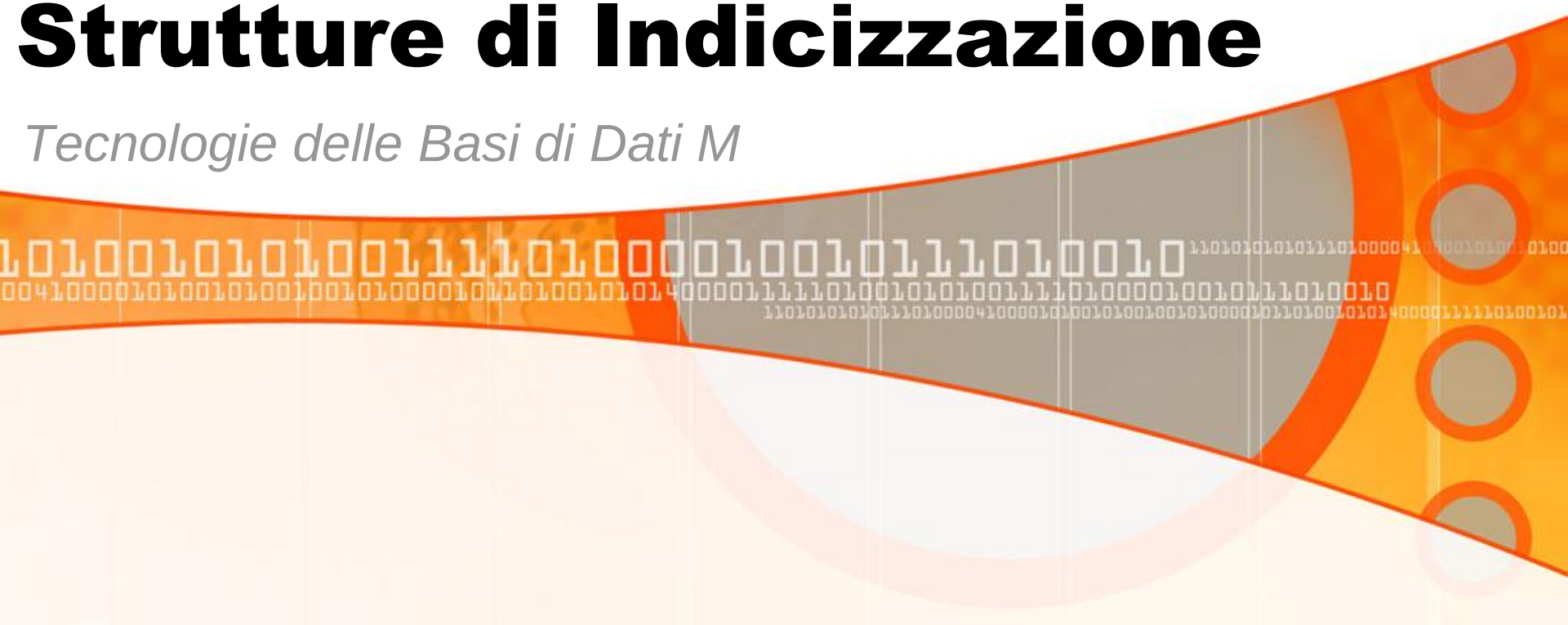
Strutture di Indicizzazione

Tecnologie delle Basi di Dati M



Svantaggi delle organizzazioni dei file

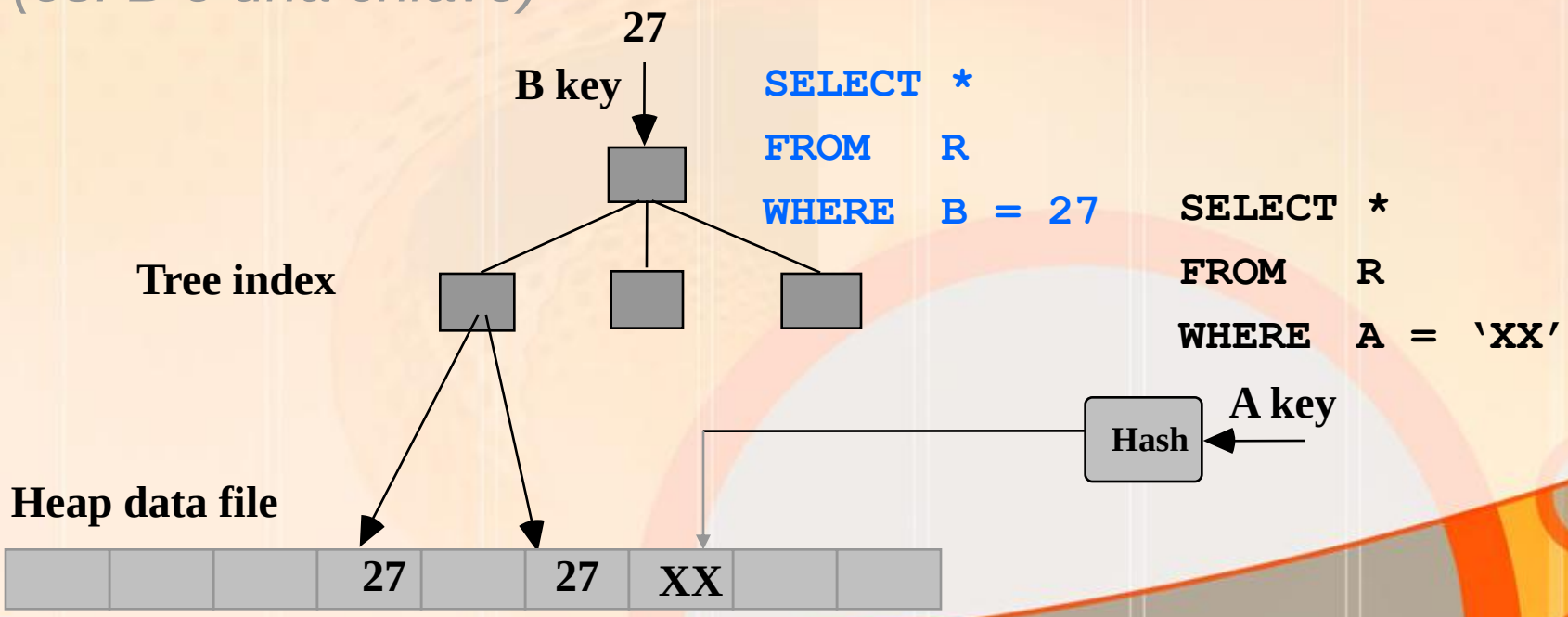
- *Le organizzazioni a heap o sequenziale hanno indubbi vantaggi:*
  - *Per gli heap l'inserimento è molto veloce*
  - *Per i sequential tutte le operazioni sono abbastanza rapide*
- *Entrambe hanno però alcuni svantaggi:*
  - *Negli heap la ricerca è molto lenta*
  - *Per i sequential la ricerca è efficiente (+ o -) solo se effettuata sul campo di ordinamento (inoltre richiedono periodiche riorganizzazioni)*

Strutture di indicizzazione

- *Sono strutture ausiliarie che permettono di recuperare velocemente i RID dei record che soddisfano una certa condizione*
  - *I dati sono comunque memorizzati come già visto*
  - *Ogni indice facilita la ricerca di una diversa condizione (chiave di ricerca)*
  - *È (sostanzialmente) una collezione di coppie <chiave, RID> (entry)*
  - *Lo scopo dell'indice è quello di velocizzare il recupero delle entry il cui valore della chiave soddisfa la condizione*
  - *Il vantaggio è che le entry sono più piccole dei record*

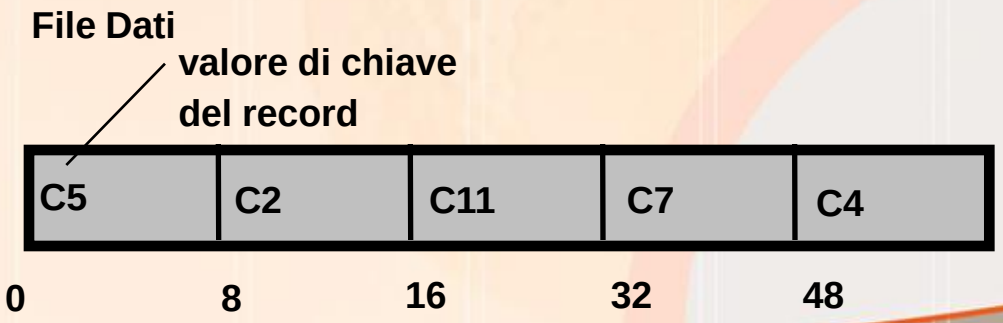
Cammini di accesso

- *La costruzione di indici su una relazione mette a disposizione modalità alternative (cammini di accesso) per localizzare velocemente i dati di interesse*
- *Si usa comunemente il termine (valore di) chiave (di ricerca) per indicare il valore di un campo usato per selezionare i record (es. B è una chiave)*



Indici: principio di base

- *Logicamente, un indice può essere visto come un insieme di coppie (entry) 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 puntatore ai record (eventualmente il solo) con valore di chiave* k_i*. Nei DBMS è quindi un RID o, al limite, un PID*
- *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 uno spazio minore rispetto al file dati*
- *I diversi indici differiscono essenzialmente nel modo con cui organizzano l'insieme di coppie* (k_i,p_i)

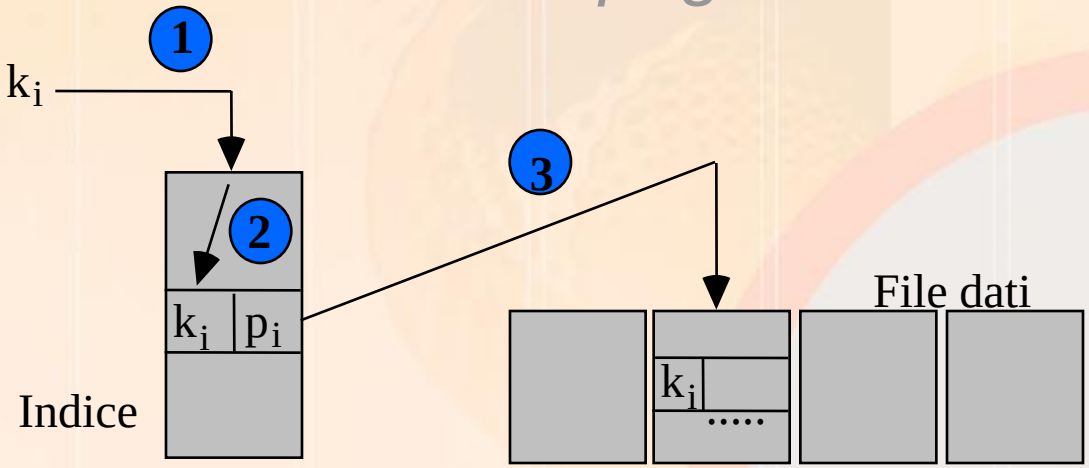


chiave	indirizzo
C2	8
C4	48
C5	0
C7	32
C11	16

Indice

Accesso con indice: schema generale

- *Si consideri un indice su chiave primaria usato per ricercare il record con chiave* k_i
- *I passi da compiere sono:*
  1. *Accedere all'indice*
  2. *Ricercare la coppia (ki, pi)*
  3. *Accedere alla pagina dati relativa*



Tipi di indici

- *Esistono diverse tipologie di indici; la prima distinzione è tra*
  - *Indici ordinati: i valori di chiave k_i vengono mantenuti ordinati, in modo da poter essere reperiti più efficientemente*
  - *Indici hash: si usa una funzione hash per determinare la posizione dei valori di chiave k_i*
    - *Questi indici tuttavia non forniscono prestazioni soddisfacenti per ricerche di intervallo*

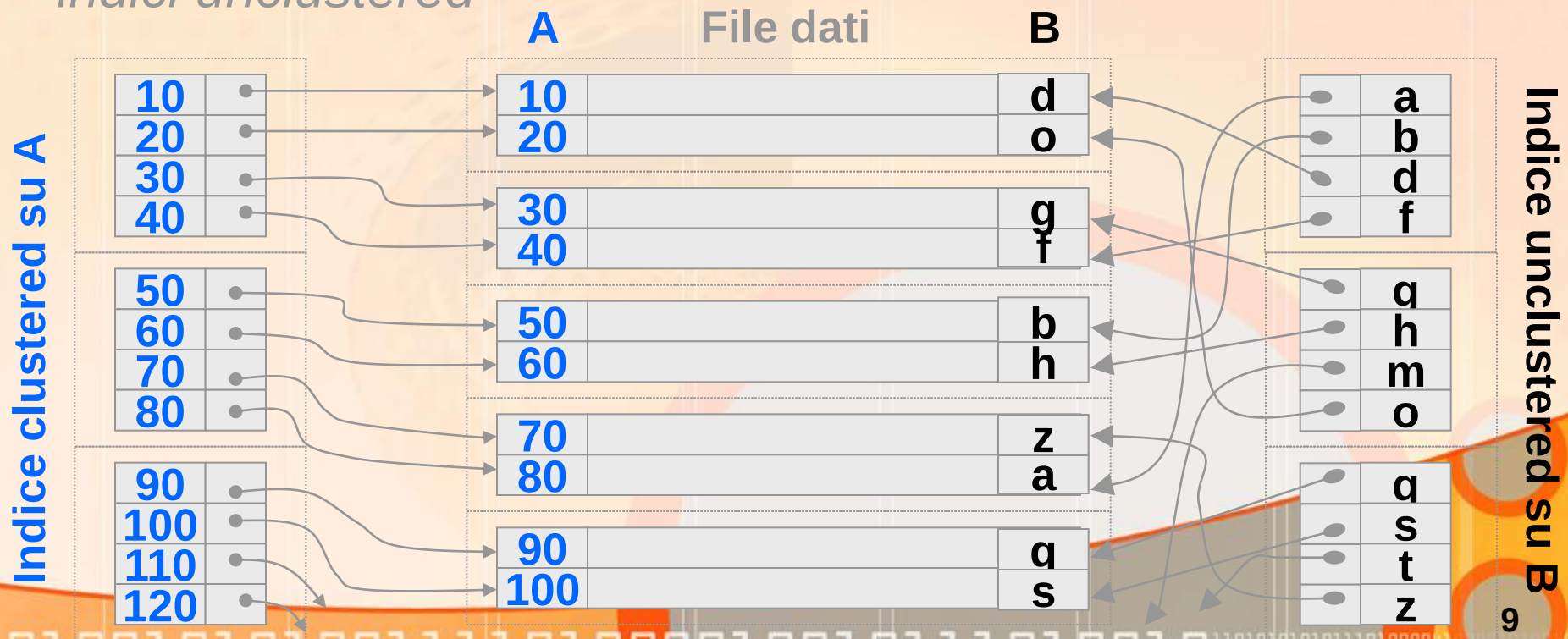
Indici: nomenclatura

- *Clustered vs. unclustered*
- *Primary vs. secondary*
- *Single-level vs. multi-level*
- *Dense vs. sparse*

- *La terminologia non è standard,*
  ad esempio qualcuno chiama primary gli indici che noi chiamiamo clustered
  e secondary gli unclustered

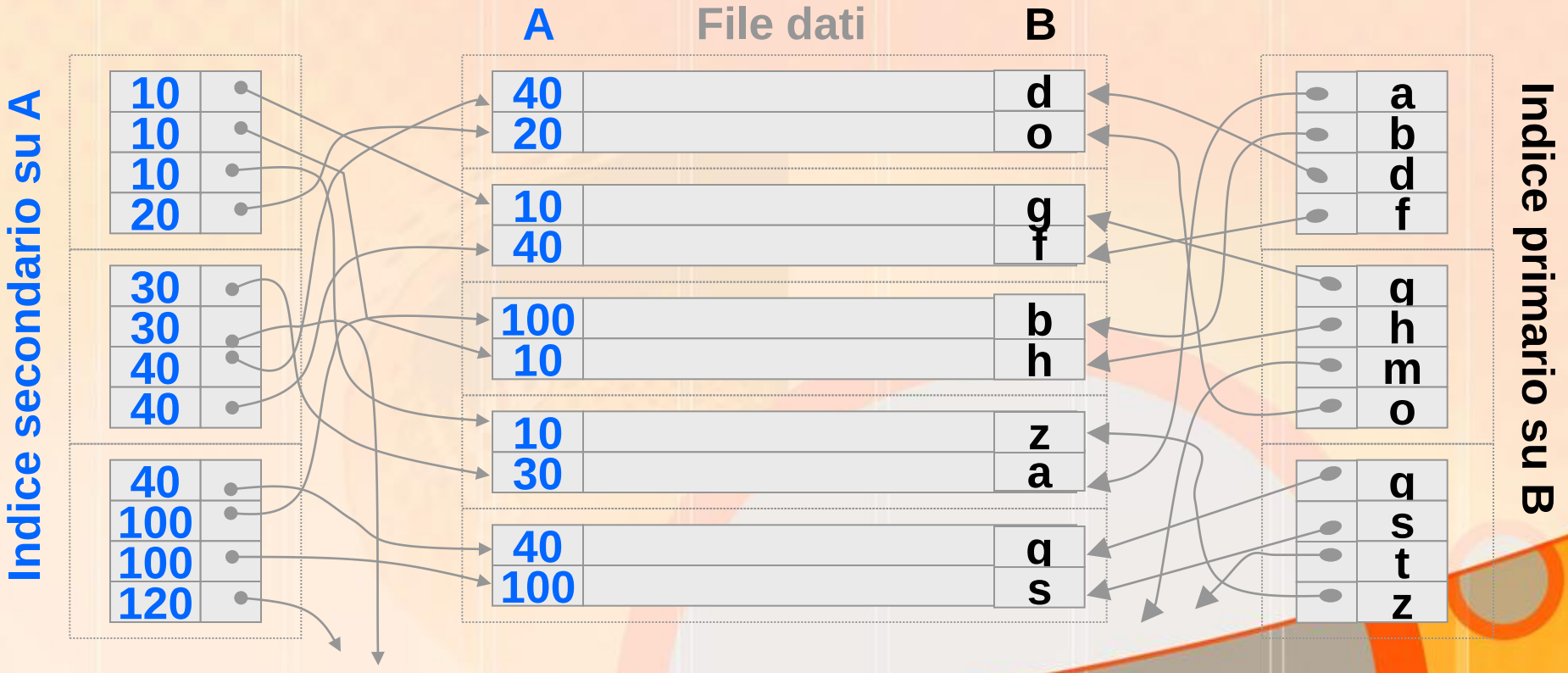
Indici clustered e unclustered

- *Un indice è detto clustered se è costruito sul campo su cui i record nel file dati sono mantenuti ordinati, altrimenti è detto unclustered*
- *Ovviamente si può costruire al massimo un indice clustered per relazione, mentre si può costruire un arbitrario numero di indici unclustered*



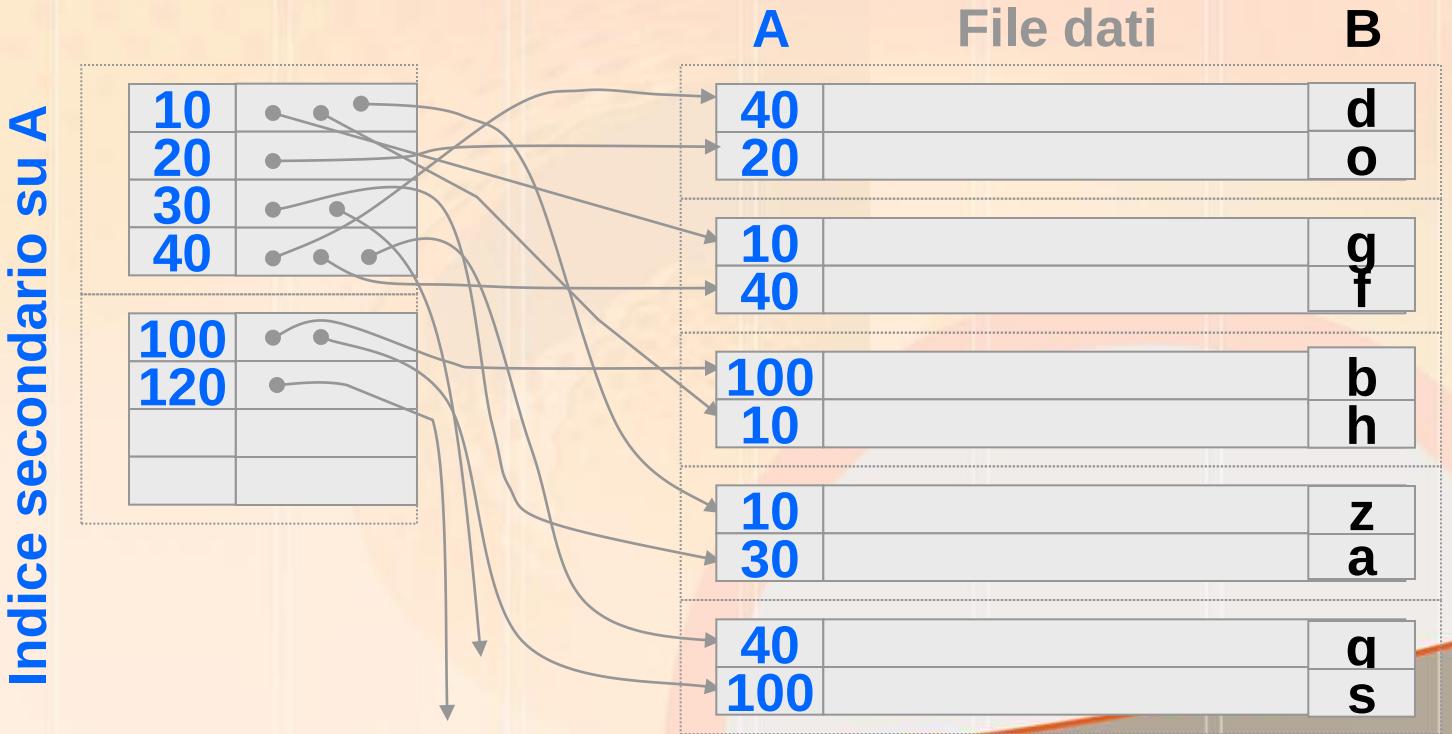
Indici primary e secondary

- *Un indice è detto primary (primario) se è costruito su un campo a valori non ripetuti (chiave relazionale), altrimenti è detto secondary (secondario)*



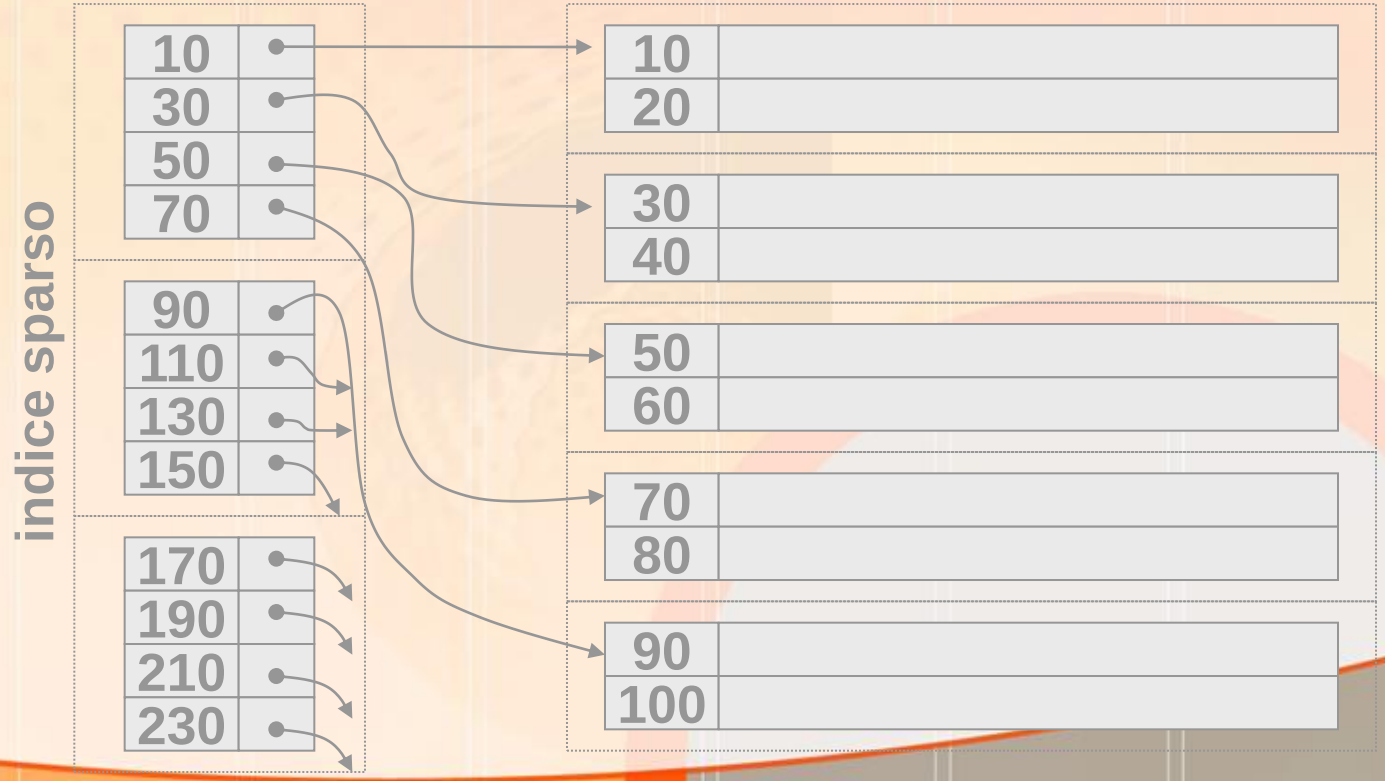
Indici secondari a liste di puntatori

- *Per evitare di replicare inutilmente i valori di chiave, la soluzione più comunemente adottata per gli indici secondari consiste nel raggruppare tutte le coppie con lo stesso valore di chiave in una lista di puntatori*



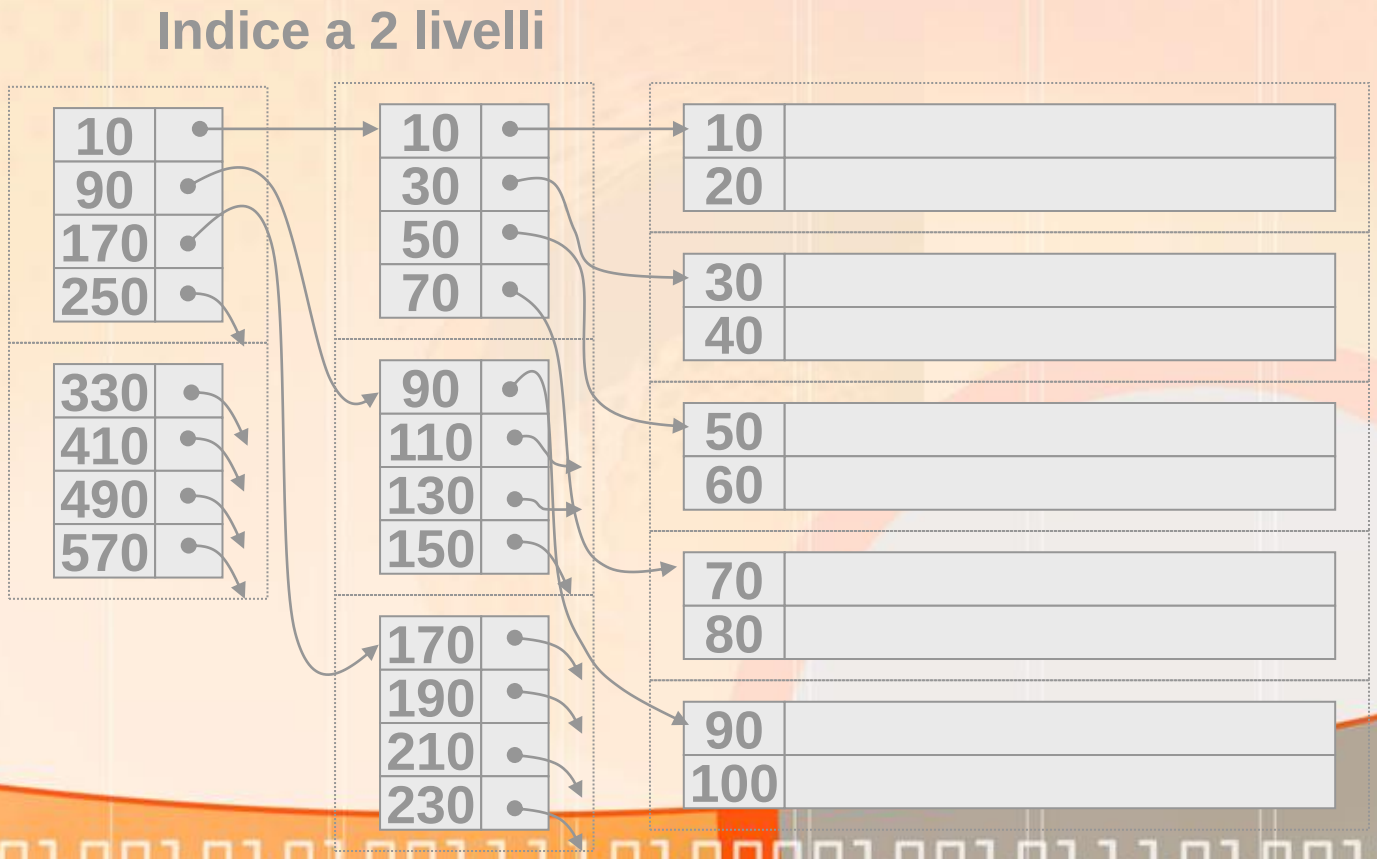
Indici dense e sparse

- *Negli indici dense (densi) il numero di puntatori è pari al numero di record del file dati, negli indici sparse (sparsi) è minore (tipicamente uno per pagina dati)*
  - *È una soluzione applicabile con indici clustered*



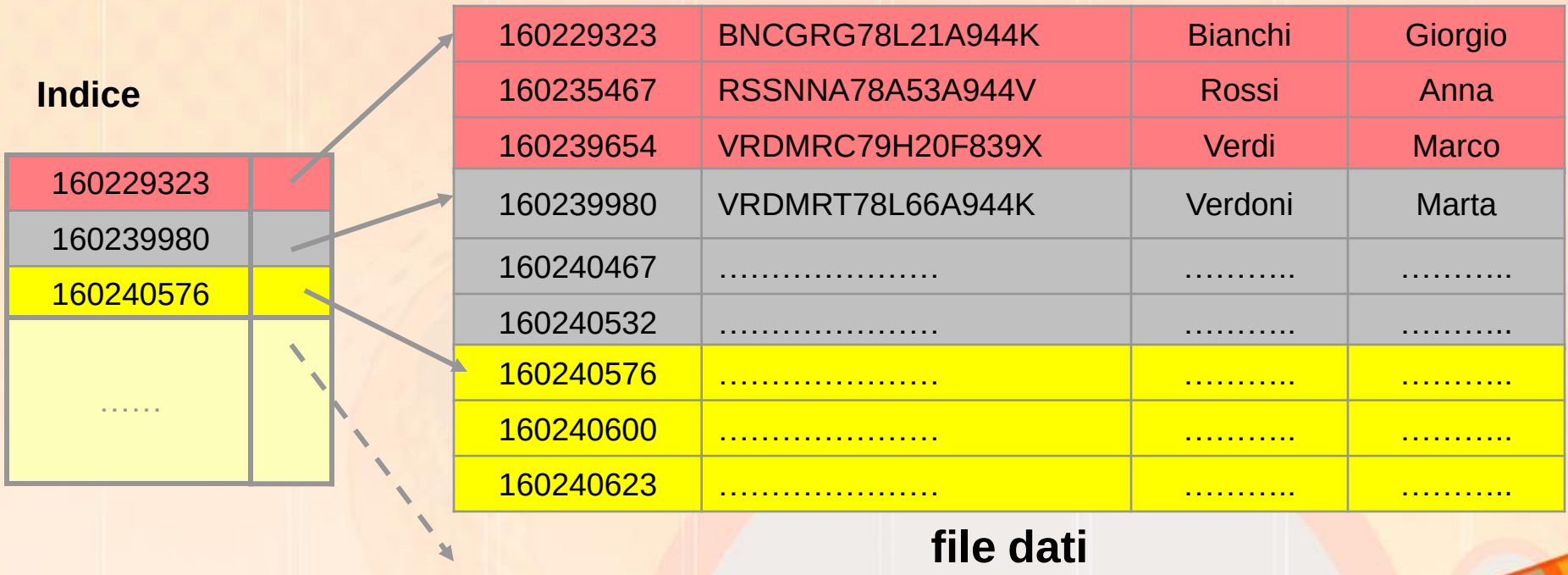
Indici single-level e multi-level

- *Gli esempi visti sono tutti relativi a indici single-level (o "flat")*
- *È tuttavia possibile "indicizzare l'indice" usando un indice (sparso!), e così via ricorsivamente, in modo da creare una struttura multi-livello (albero)*



Esempio (1)

primary clustered sparse single-level index



Esempio (2)

secondary unclustered dense single-level index

pagine dati

Lista di RID

Alessandri	
Bianchi	
Bianchini	
… … …	
Rossi	
……	
Verdi	
Verdone	
…… …	

160229323	BNCGRG78L21A944K	Bianchi	Giorgio
160235467	RSSNNA78A53A944V	Rossi	Anna
160239654	VRDMRC79H20F839X	Verdi	Marco
160239980	VRDMRT78L66A944K	Verdone	Marta
160240467	……	Alessandri	Maria
160240532	……	Bianchini	Carlo

…… …… …… ……

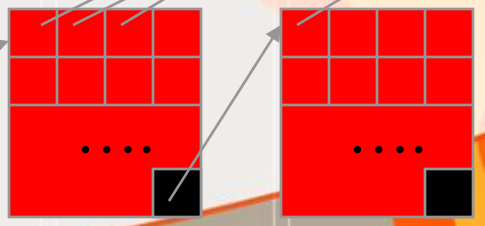
160240576	……	Rossi	Demetrio
160240600	……	Bianchi	Arrigo
160240623	……	Verdi	Remo

N.B. La lista può essere realizzata come:

valore di chiave

puntatore a pagina di puntatori

puntatori a record



Indici in SQL

- *In SQL la definizione di indici avviene mediante lo statement CREATE INDEX (ma non è standardizzato!)*
- *In DB2:*

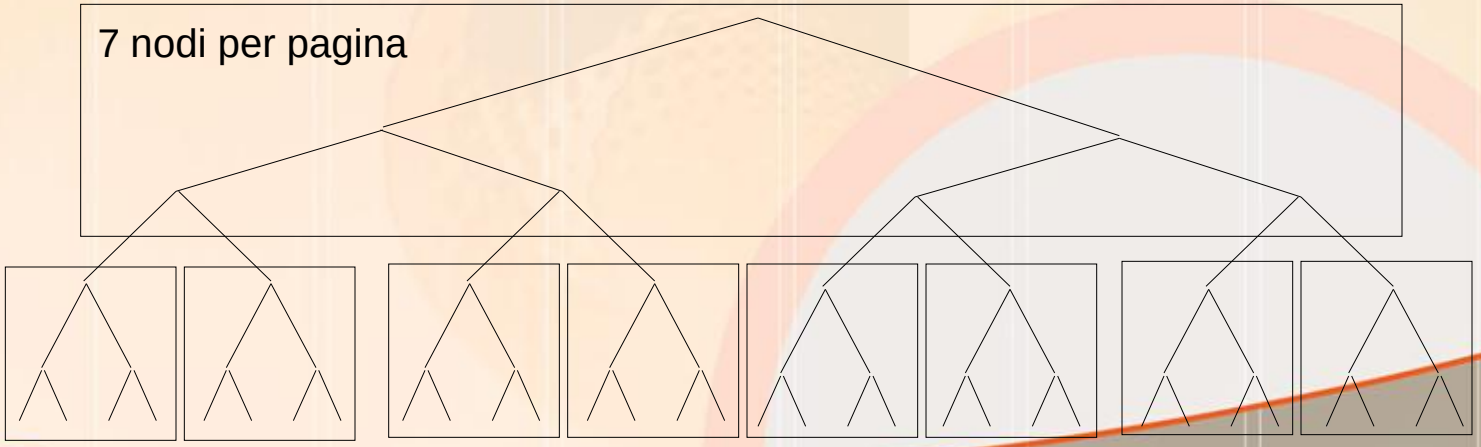
```
CREATE INDEX VotoIDX            -- secondary unclustered index
ON Esami(Voto DESC)             -- ASC è il default

CREATE UNIQUE INDEX MatrIDX     -- primary unclustered index
ON Studenti(Matricola)

CREATE INDEX VotoIDX             -- clustered index
ON Esami(Voto DESC) CLUSTER

CREATE INDEX Anagrafica         -- indice multi-attributo
ON Persone(Cognome,Nome)
```

Indici multi-livello

- *Per ragioni di efficienza, normalmente gli indici sono multi-livello (alberi)*
- *Possiamo "adattare" alla memoria secondaria gli alberi di ricerca pensati per la memoria centrale?*
- *Requisiti:*
  - *Bilanciamento (prestazioni nel caso peggiore)*
  - *Paginazione (lettura da disco)*
  - *Utilizzazione minima delle pagine (dimensioni)*
  - *Efficienza in aggiornamento*

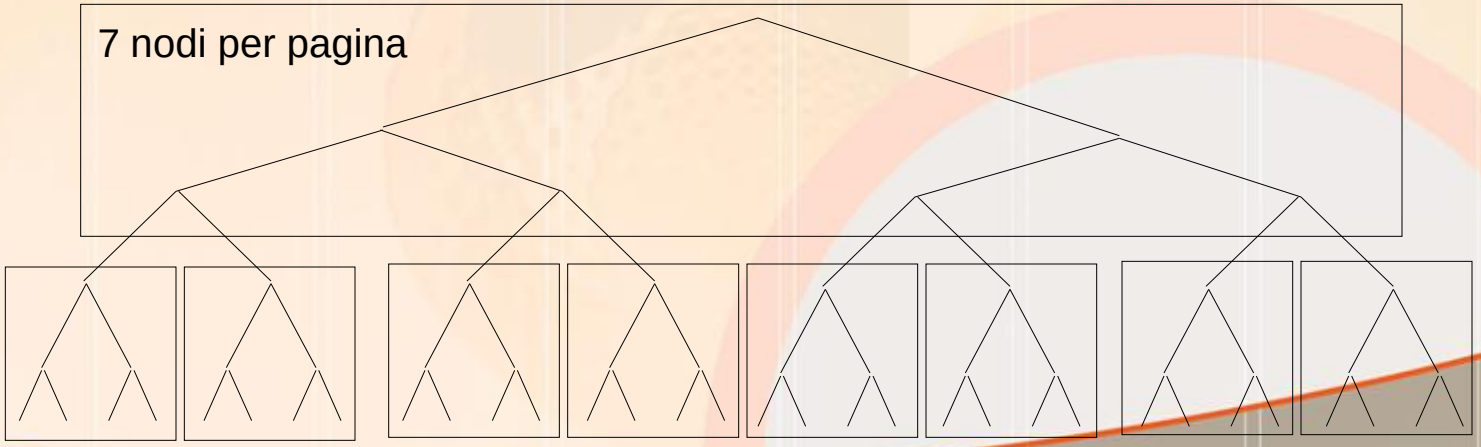
Paginazione di alberi (i)

- *Gli alberi in memoria centrale (AVL tree, red-black tree) sono tipicamente binari*
- *Numero di nodi molto alto*
  - *La visita richiede molti accessi*
- *Inclusione di più nodi all'interno della stessa pagina*



Paginazione di alberi (ii)

- *Svantaggi:*
  - *Complicazione dell'algoritmo di bilanciamento (inefficiente in aggiornamento)*
  - *Nessuna garanzia di occupazione minima delle pagine*
- *Occorre trovare una soluzione specifica!*



B-tree (R. Bayer & E. McCreight, 1972)

- *Struttura dati ad albero che mantiene i dati ordinati e i nodi bilanciati permettendo operazioni di inserimento, cancellazione e ricerca in tempi ammortizzati logaritmicamente*
- *Etimologia (D. Comer):*
  - *Balanced?*
  - *Broad?*
  - *Bushy?*
  - *Boeing?*
  - *Bayer?*

B-tree: terminologia

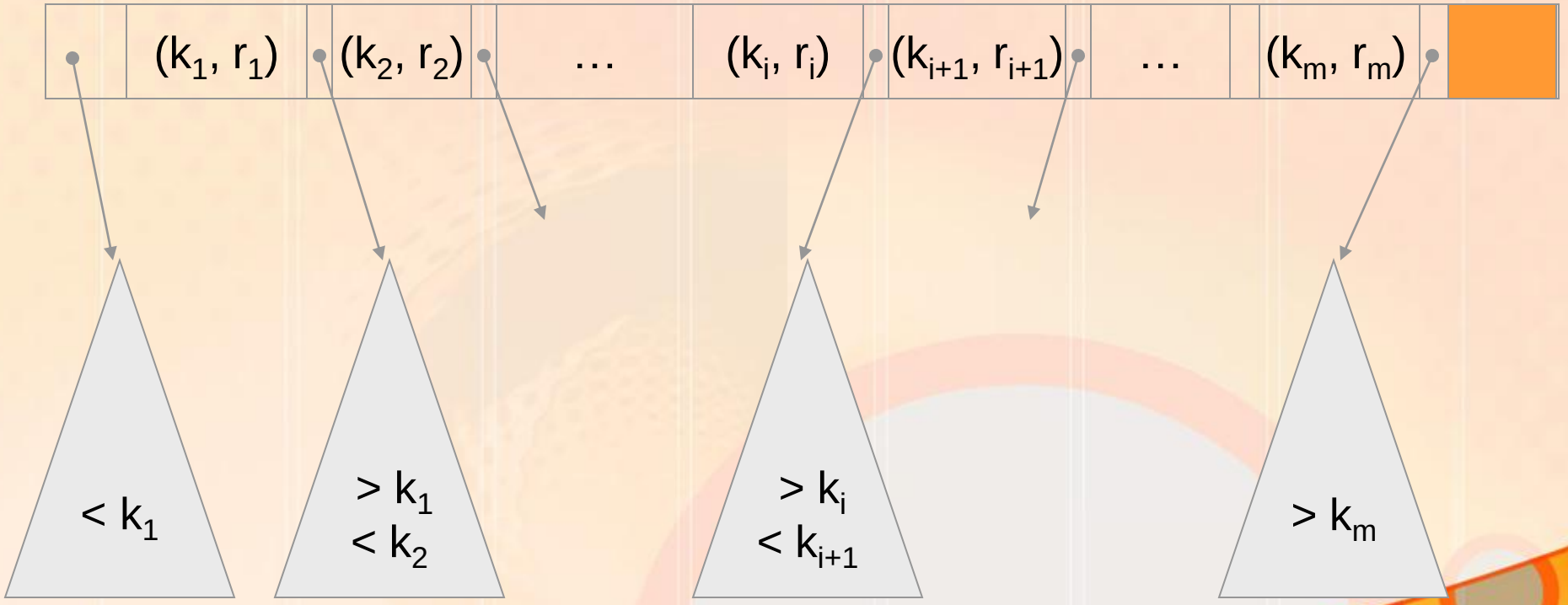
- *Un B-tree è un albero a più vie perfettamente bilanciato organizzato a nodi, corrispondenti a pagine*
- *Ogni nodo contiene un numero di entry* ***m*** *che può variare tra* ***d*** *e* ***2d*** *(****d*** *= ordine dell'albero)*
- *Il numero di nodi figli di un nodo è pari a* ***m+1*** *(può quindi variare tra* ***d+1*** *e* ***2d+1****)*
  - *Fan-out elevato, quindi altezza limitata*
  - *Costo di ricerca (molto) basso*
  - *Dimensioni ridotte*
- *La radice può violare il vincolo di minima utilizzazione ed avere anche una sola entry*

B-tree: caratteristiche

- *Il perfetto bilanciamento indica che il percorso dalla radice ad ogni foglia ha la stessa lunghezza (altezza dell'albero)*
- *Ogni ricerca segue un unico percorso dalla radice ad una foglia (costo ≤ altezza)*
- *Il bilanciamento è garantito dalle operazioni di inserimento e cancellazione dei record*
  - *Per garantire il bilanciamento, le operazioni di diramazione di un nodo avvengono verso l'alto (e non verso il basso)*
- *Occupazione minima del 50% (tranne per la radice)*
- *Occupazione tipica > 66%*

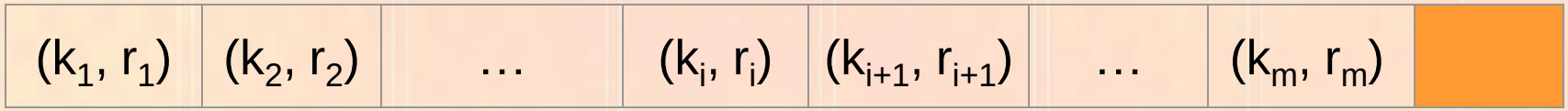
B-tree: Formato dei nodi interni

- *I nodi interni hanno il formato mostrato in figura, in cui è:* $k_1 < k_2 < \ldots < k_m$



B-tree: Formato dei nodi foglia

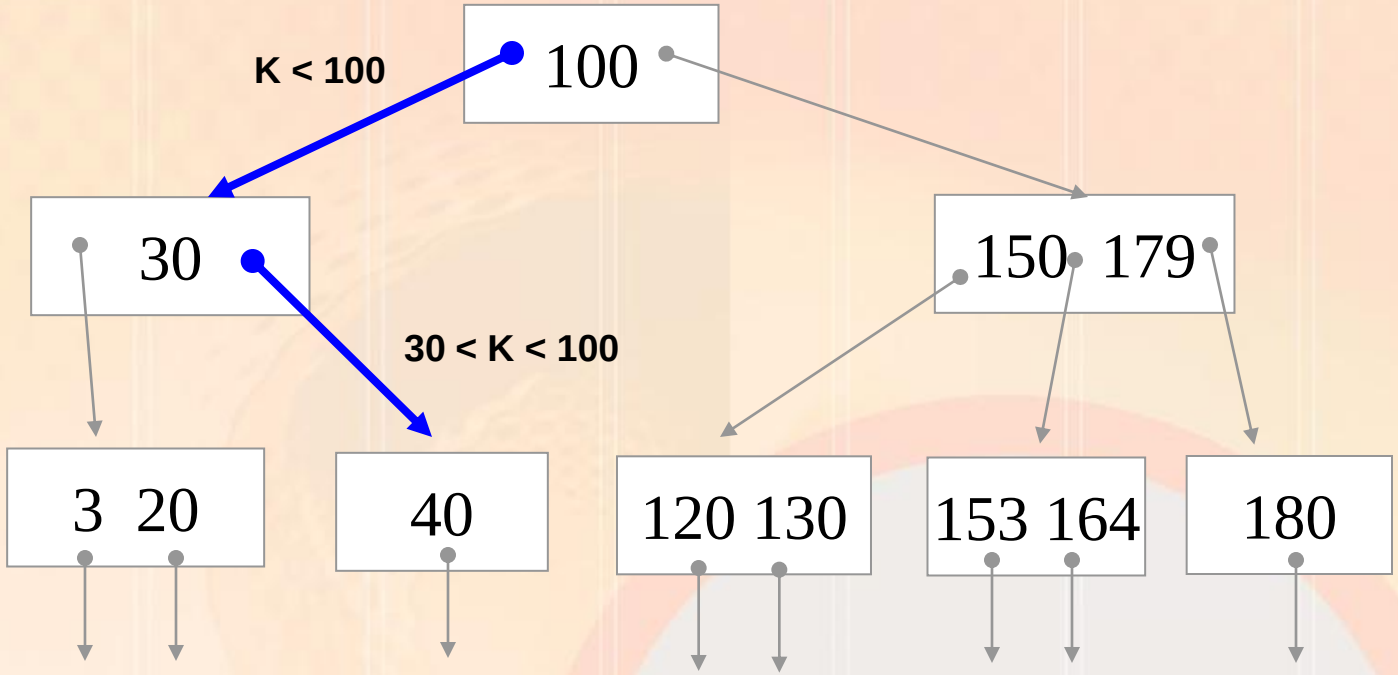
- *I nodi foglia hanno il formato mostrato in figura, in cui è:* $k_1 < k_2 < \ldots < k_m$



- *Vista la mancanza dei puntatori ai sotto-alberi, tipicamente le foglie contengono più entry dei nodi interni*

Esempio di B-tree

- *La figura mostra un esempio di B-tree di ordine 1*

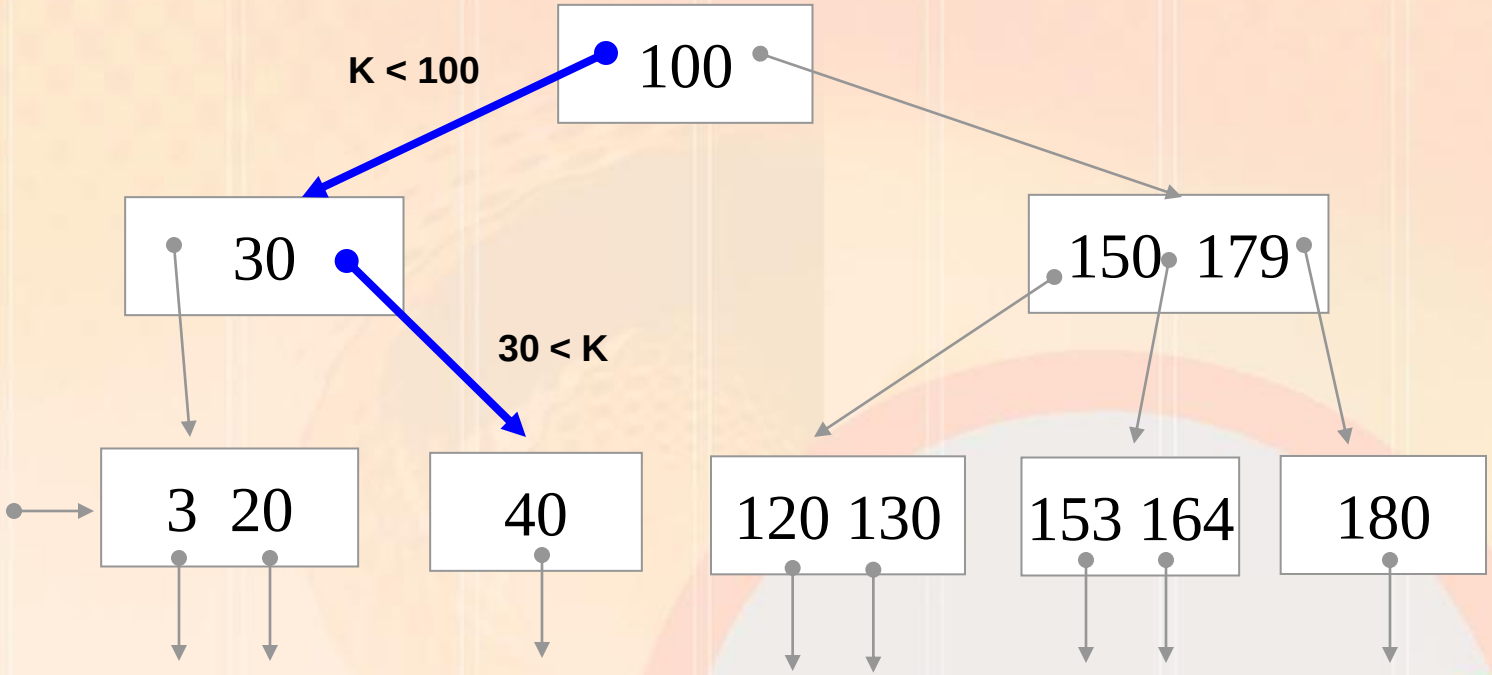


B-tree: Ricerca

- *Si parte dalla radice dell'albero*
  - *In genere la radice viene mantenuta in RAM*
- *Si cerca la chiave di ricerca* k *tra quelle del nodo corrente*
  - *Se è presente, trovato*
  - *Se non è presente e siamo in una foglia, non trovato*
  - *Se non è presente e siamo in un nodo interno sostituisci il nodo corrente coll'*i*-esimo nodo figlio, dove:* $k_{i-1} < k < k_i$

B-tree: Esempio di ricerca

- *Cerchiamo la chiave 40*
  - *40<100 => seguiamo il figlio sinistro*
  - *40>30 => seguiamo il figlio destro*



- *E per cercare 30? E 90?*

Costo della ricerca

- *Ogni nodo viene sostituito da un unico figlio*
- *Nel caso peggiore si arriva ad una foglia*
- *Costo ≤ altezza dell'albero – 1 + 1*
  - *– 1 perché la radice è già in RAM*
  - *+ 1 per l'accesso al file dati*
    - *Non necessario se l'indice contiene il file dati*
- *Ne consegue che occorre saper calcolare l'altezza di un B-tree di ordine d*

Numero massimo di nodi in un B-tree di altezza h

- *Il numero massimo di nodi si ha quando tutti i nodi sono pieni, ovvero contengono* $2d$ *entry*
  - *Ogni nodo interno ha quindi* $2d+1$ *figli*

$$b_{\max} = \sum_{l=0}^{h-1} (2d+1)^l = \frac{(2d+1)^h - 1}{2d}$$

- *Il corrispondente numero di entry è quindi:*

$$N_{\max} = 2d \cdot b_{\max} = (2d+1)^h - 1$$

Numero minimo di nodi in un B-tree di altezza h

- *Il numero minimo di nodi si ha quando tutti i nodi (tranne la radice) sono pieni a metà, ovvero contengono* d *entry, mentre la radice contiene una sola entry*
  - *Ogni nodo interno ha quindi* $d+1$ *figli*

$$b_{\min} = 1 + 2\sum_{l=0}^{h-2}(d+1)^l = 1 + 2\frac{(d+1)^{h-1}-1}{d}$$

- *Il corrispondente numero di entry è quindi:*

$$N_{\min} = 1 + d(b_{\min} - 1) = 2(d+1)^{h-1} - 1$$

Altezza di un B-tree

- *Ora è possibile, conoscendo N, calcolare l'altezza del B-tree:*

$$N_{\min} \le N \le N_{\max}$$

$$2(d+1)^{h-1} - 1 \le N \le (2d+1)^h - 1$$

- *Passando ai logaritmi si ha:*

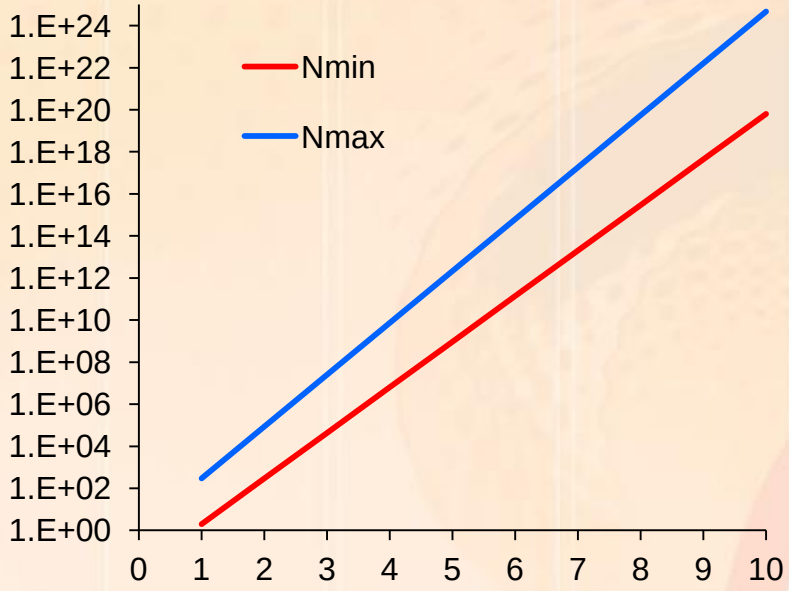
$$\lceil \log_{2d+1}(N+1) \rceil \le h \le \left\lfloor \log_{d+1}\left(\frac{N+1}{2}\right) \right\rfloor + 1$$

Altezza di un B-tree: esempio

- *Supponiamo di avere le seguenti dimensioni:*
  - *chiave di 8 byte, RID di 4 byte, PID di 2 byte*
  - *pagine di 4096 byte*
- *Si ottiene:*
  - $(8+4)2d + 2(2d+1) = 4096$
  - $d = \lfloor (4096-2)/(24+4) \rfloor = 146$
- *Se $N = 10^9$, la ricerca di un valore di chiave richiede al massimo* $\lfloor \log_{147}(10^9/2) \rfloor + 1 = 5$ *operazioni di I/O!*
  - *Una ricerca binaria richiederebbe 22 accessi, supponendo di avere le pagine piene*

Altezza di un B-tree: considerazioni

- *La variabilità di h è, fissati N e d, molto limitata (differenza di 1 tra minima e massima)*
- *Con questi valori di d, con* $h = 3$ *si gestiscono fino a* $(2*146+1)^3 - 1$ = *circa 25 milioni di chiavi*



P	d	N					
		1000		1000000		1000000000	
		h_{min}	h_{max}	h_{min}	h_{max}	h_{min}	h_{max}
512	18	3	3	5	5	7	8
1024	36	2	2	4	4	6	6
2048	73	2	2	4	4	5	5
4096	146	2	2	3	3	5	5
8192	292	2	2	3	3	4	4
16384	585	1	2	3	3	4	4

B-tree: Ricerca di un intervallo

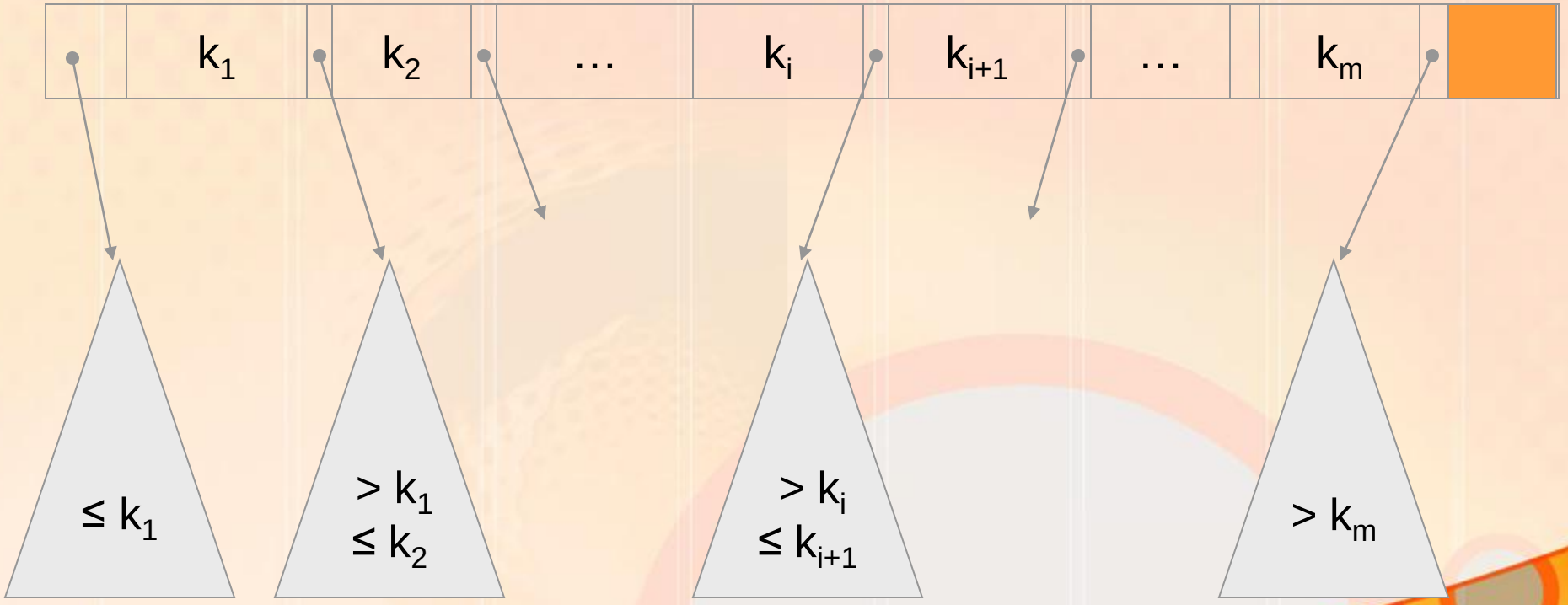
- *Si parte dalla radice dell'albero*
- *Occorre visitare l'albero in ordine*
- *Supponendo d=0:*
  - *Visita del sotto-albero sinistro*
  - *Visita della chiave*
  - *Visita del sotto-albero destro*
- *L'inefficienza è dovuta al fatto che le RID non sono memorizzate solo nelle foglie*

B$^+$-tree

- *Le principali caratteristiche di un B$^+$-tree sono:*
  - *Le coppie (k_i,r_i) sono tutte contenute nelle foglie*
    - *Aumenta l'altezza (rispetto al B-tree)*
  - *Le foglie sono collegate a lista (eventualmente doppia) mediante puntatori (PID) per favorire la risoluzione di query di intervallo*
  - *I nodi interni contengono solamente valori delle chiavi (non necessariamente corrispondenti a dati esistenti)*
    - *Aumenta la capacità dei nodi interni*
    - *Diminuisce l'altezza (rispetto al B-tree)*

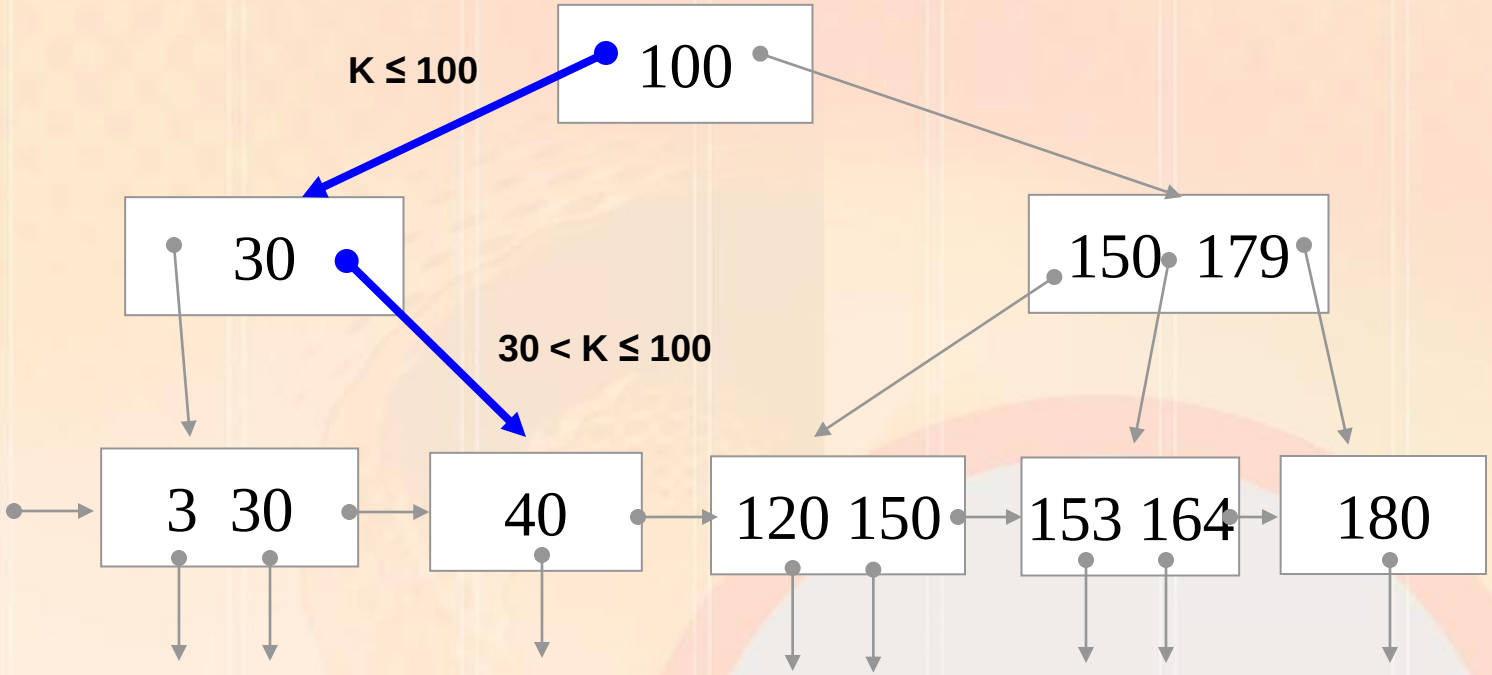
B^+-tree: Formato dei nodi interni

- *I nodi interni hanno il formato mostrato in figura, in cui è:* $k_1 < k_2 < \ldots < k_m$



Esempio di B⁺-tree

- *La figura mostra un esempio di B⁺-tree di ordine 1*

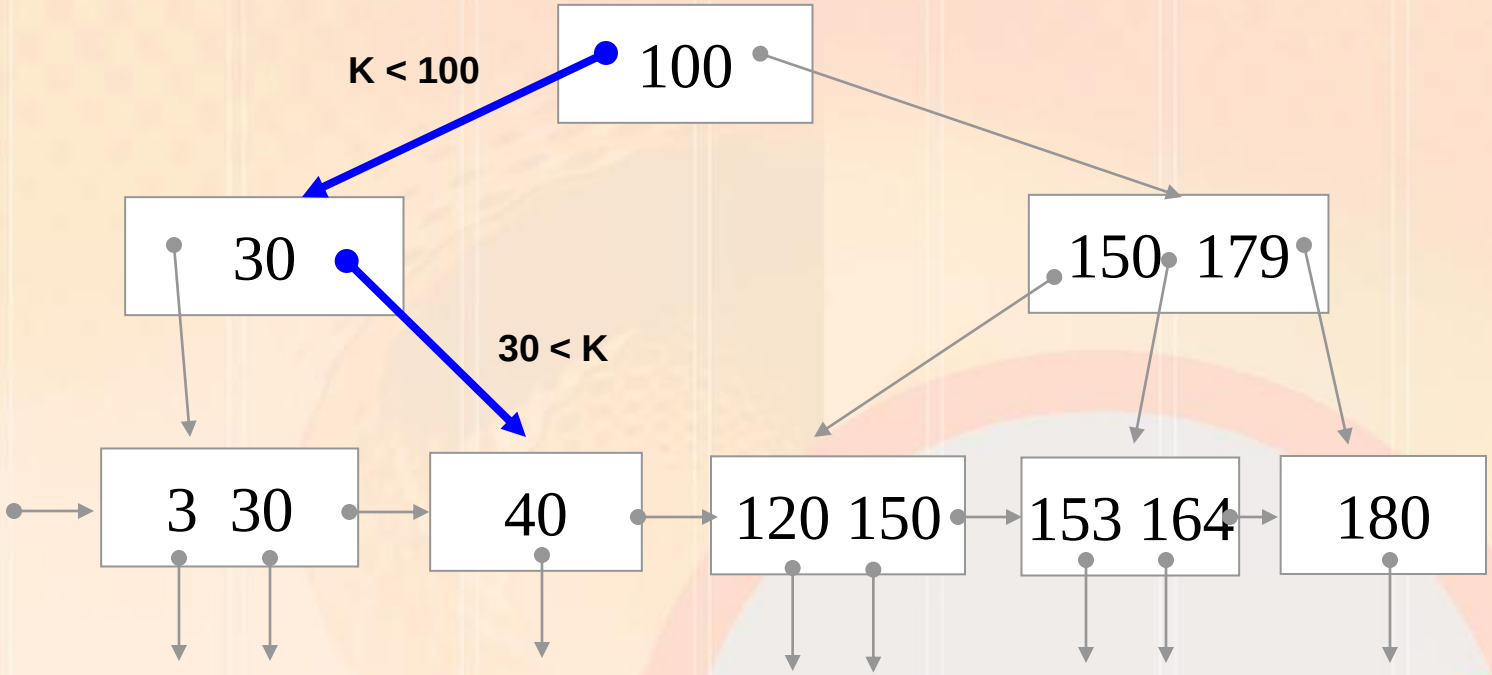


B^+-tree: Ricerca

- *Si parte dalla radice dell'albero*
  - *In genere la radice viene mantenuta in RAM*
- *Si cerca la chiave di ricerca* k *tra quelle del nodo corrente*
  - *Se siamo in un nodo interno sostituisci il nodo corrente coll'*i*-esimo nodo figlio, dove:* $k_{i-1} < k \leq k_i$
  - *Se siamo in una foglia ed è presente, trovato*
  - *Se siamo in una foglia e non è presente, non trovato*
- *Costo = altezza dell'albero*

B+-tree: Esempio di ricerca

- *Cerchiamo la chiave 40*
  - *40<100 => seguiamo il figlio sinistro*
  - *40>30 => seguiamo il figlio destro*



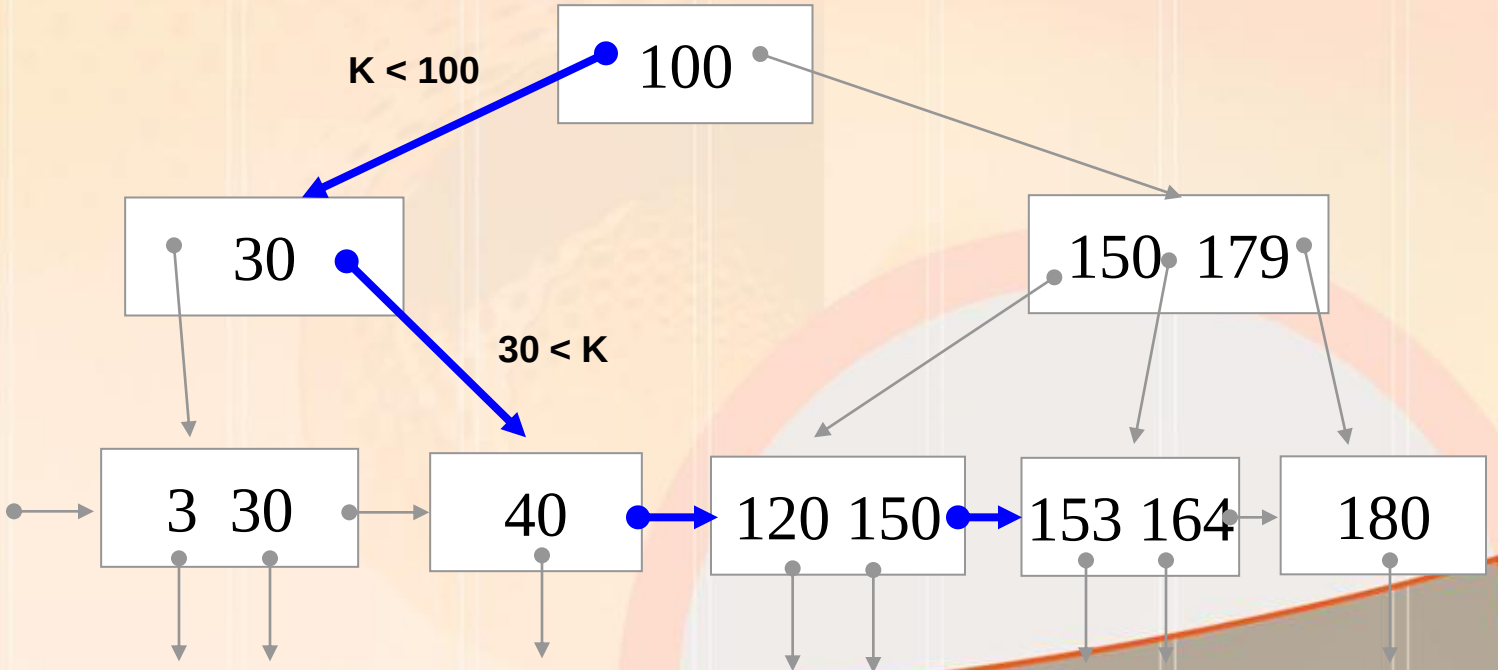
- *E per cercare 30? E 90?*

B+-tree: Ricerca di un intervallo

- *Supponiamo di cercare nell'intervallo* $[k_{low}, k_{high}]$
- *Cerchiamo il primo valore di chiave* $k \geq k_{low}$
- *Siccome le foglie sono collegate a lista possiamo evitare di navigare l'albero e scandire sequenzialmente le foglie fino a quando non troviamo un valore* $k > k_{high}$
- *I RID incontrati sono il risultato della query*
  - *Se l'indice è unclustered, può essere necessario ordinare i RID, in modo da evitare di leggere più volte una pagina dati*
  - *E se l'indice è sparse?*

B+-tree: Esempio di ricerca di un intervallo

- *Cerchiamo le chiavi nell'intervallo* [*35,160*]
  - *35<100 => seguiamo il figlio sinistro*
  - *35>30 => seguiamo il figlio destro*
  - *Da qui scorriamo le foglie fino a 164>160*



B+-tree: Inserimento

- *Supponiamo di voler inserire una nuova entry (k,r)*
- *La procedura di inserimento procede innanzitutto cercando la foglia in cui inserire il nuovo valore di chiave k*
- *Se c'è posto (la foglia contiene meno di 2d entry) la nuova coppia (k,r) viene inserita nella foglia e la procedura termina*
- *Se non c'è più posto (foglia in overflow)?*

B$^+$-tree: Split di una foglia

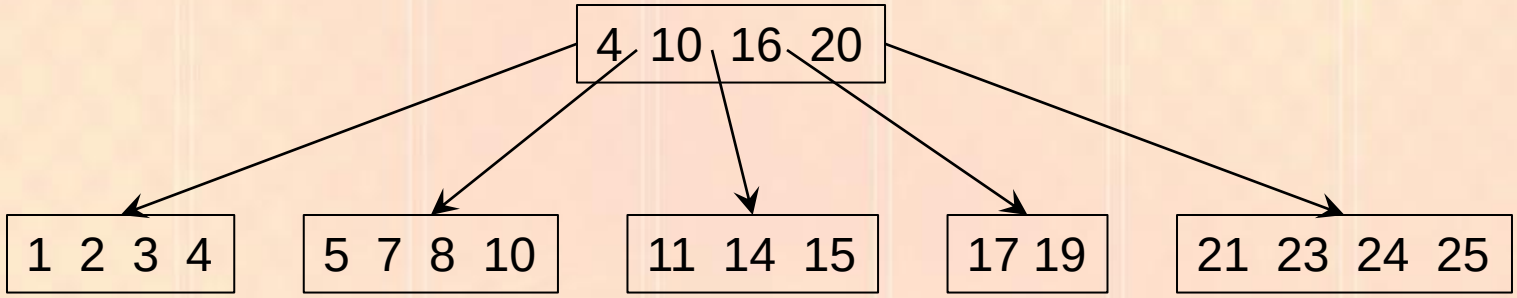
- *La foglia F in overflow viene divisa in due foglie (F_L e F_R)*
- *Ciascuna foglia conterrà la metà delle entry di F*
- *Viene individuato il valore mediano k_c delle entry di F (normalmente $c=d$)*
- *In F_L vanno tutte le entry con chiave $k \leq k_c$*
- *In F_R vanno tutte le entry con chiave $k > k_c$*
- *Nel nodo padre di F il puntatore a F viene sostituito dai puntatori a F_L e F_R e dal valore di k_c*

B+-tree: Propagazione degli split

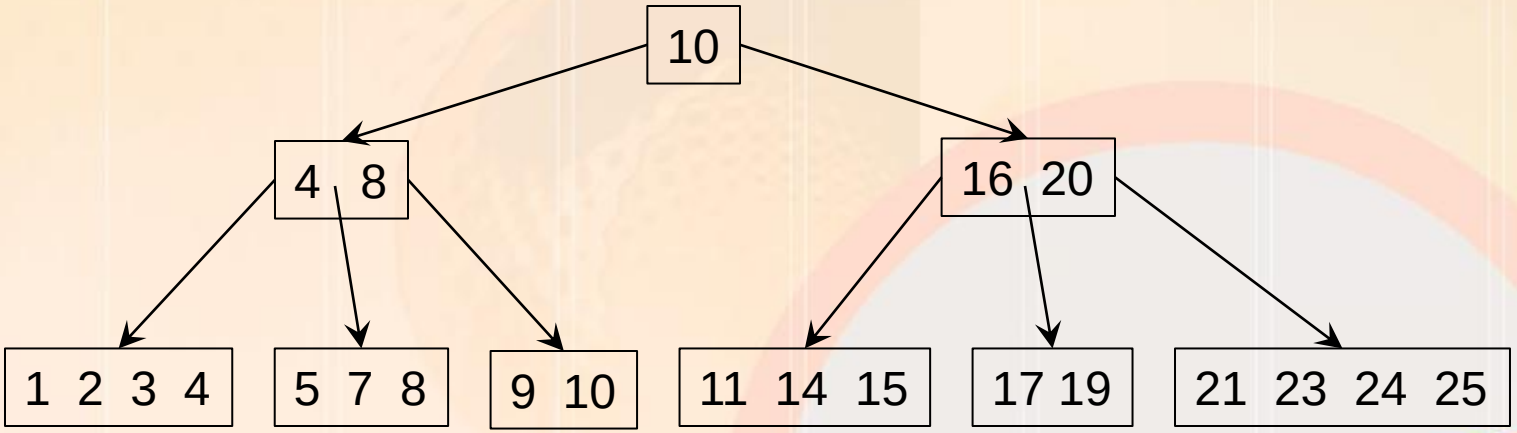
- *Cosa succede se il nodo padre di F è pieno?*
- *È un nodo che va in overflow:*
  si agisce come in precedenza, ovvero split
- *Ne consegue che lo splitting si propaga ricorsivamente verso l'alto*
- *Al limite, se anche la radice è piena,*
  si divide in due e si crea un nuovo nodo radice
  - *L'albero cresce in altezza*
  - *Motivo per cui la radice non può avere un'occupazione minima superiore a 2*

B+-tree: Esempio di split

- *B+-tree di ordine 2*



- *Inseriamo la chiave* 9



B+-tree: Costo di inserimento

- *Senza split: h letture + 1 scrittura*
- *Con split:*
  - *Nel caso peggiore si effettua la ricorsione fino alla radice: h letture + 2h+1 scritture*
  - *Per il calcolo del caso medio occorre ricordare che, per un albero con b nodi, ci sono stati b-1 split*
  - *Siccome* $N_{min}=1+d(b-1)\leq N$
  - *Otteniamo che il numero medio di split è (b-1)/N, ovvero circa 1/d*
  - *Costo medio: ≤ h letture + 1+2/d scritture*

B+-tree: Alternativa allo split

- *Come visto lo split di un nodo può essere molto costoso*
- *Un'alternativa può essere quella di inserire alcune entry in un nodo vicino con lo stesso padre (fratello) non in overflow (re-distribution)*
  - *Si riduce il numero di split*
  - *Aumenta il costo (si leggono e riscrivono più nodi)*
  - *Si ottiene un albero mediamente "più pieno"*

B+-tree: Cancellazione

- *Supponiamo di voler cancellare una entry (k,r)*
- *La procedura di inserimento procede innanzitutto cercando la foglia in cui si trova il valore di chiave k*
- *Si cancella la entry dalla foglia*
- *Se la foglia contiene non meno di d entry la procedura termina*
- *Se la foglia contiene d–1 entry (underflow)?*

B+-tree: Gestione dell'underflow

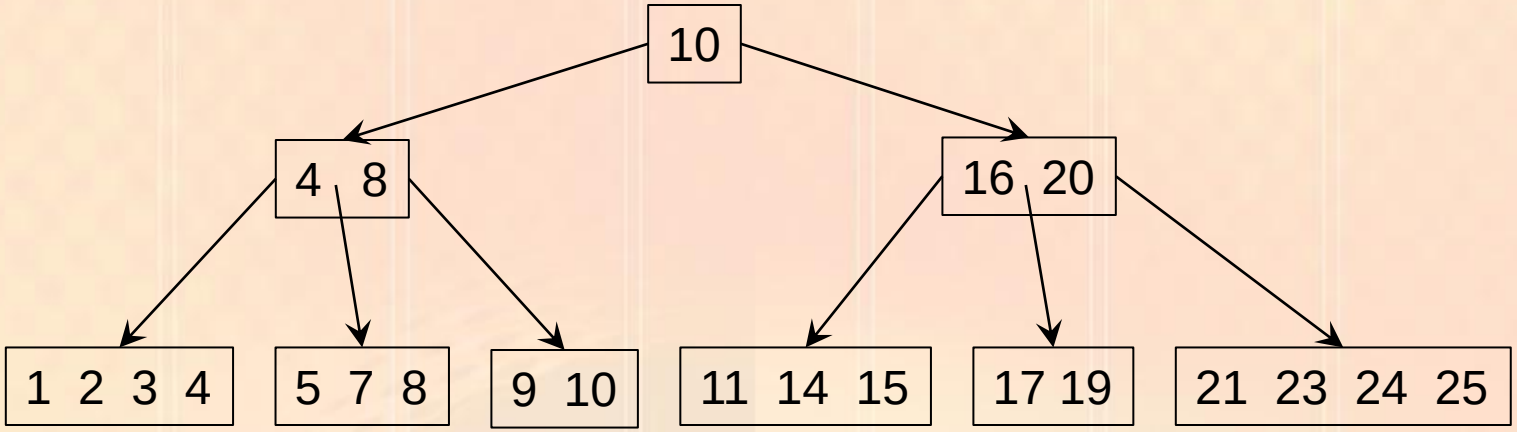
- *Si hanno due possibilità*
  - *Ridistribuire le entry della foglia in underflow e di una sorella (vicina con lo stesso padre)*
  - *Cancellare la foglia in underflow, inserendo le sue entry in una sorella*
- *La cancellazione è possibile solo se la foglia sorella ha d o d+1 entry (altrimenti?)*
- *La cancellazione può essere propagata verso l'alto*
  - *Al limite la radice va in underflow*
  - *L'albero si abbassa*

B+-tree: Ridistribuzione

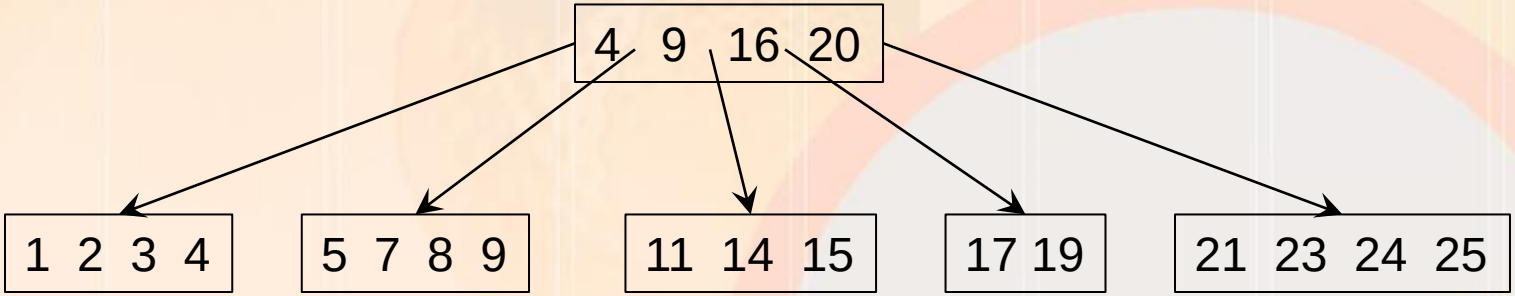
- *Se il nodo fratello ha più di d+1 entry occorre ridistribuire*
- *Le entry vengono distribuite in modo bilanciato tra i due nodi*
- *Occorre aggiornare il valore di separatore nel nodo padre*
  - *Il numero di entry nel padre non cambia*
  - *Il fenomeno non viene propagato verso l'alto*

B+-tree: Esempio di concatenazione

- *B+-tree di ordine 2*

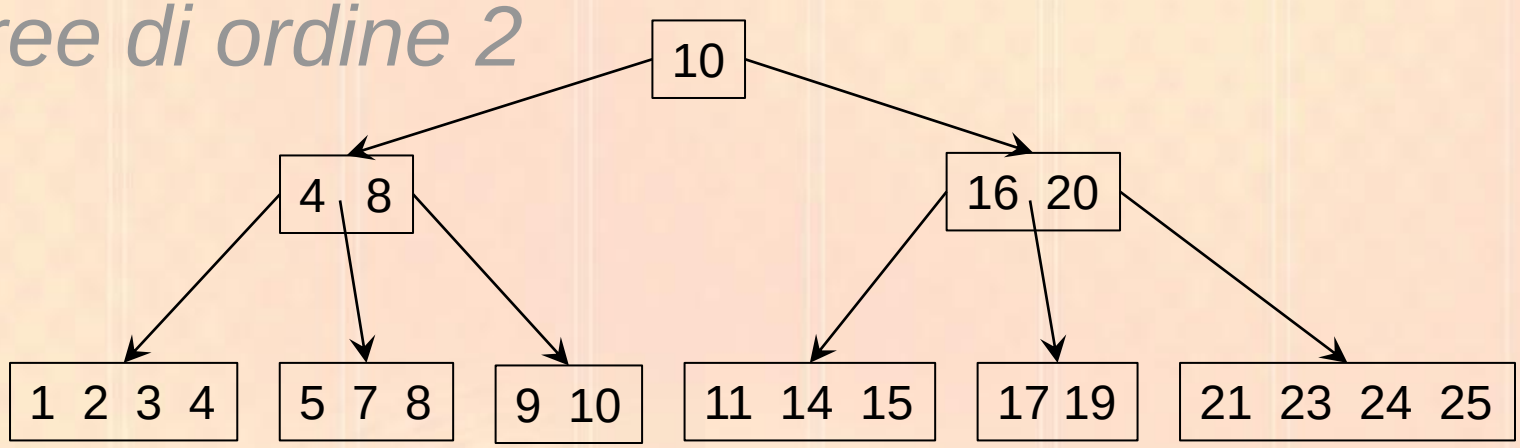


- *Cancelliamo la chiave* 10

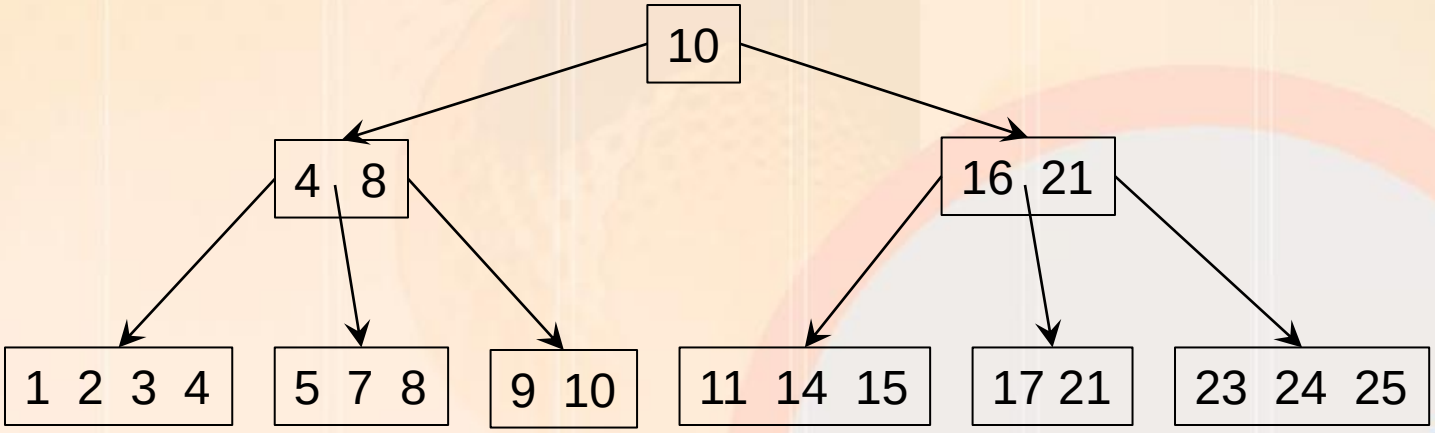


B^+-tree: Esempio di ridistribuzione

- *B^+-tree di ordine 2*



- *Cancelliamo la chiave* 19



B+-tree: Costo di cancellazione

- *Senza underflow:* h *letture +* 1 *scrittura*
- *Con underflow:*
  - *Ogni concatenazione costa* 1 *lettura +* 2 *scritture*
  - *Nel caso peggiore si effettua la ricorsione fino alla radice*
    - *Concatenazione per tutti i livelli tranne i primi due*
    - *Ridistribuzione nel figlio della radice*
  - *Costo massimo:* $2h–1$ *letture +* $h+1$ *scritture*
  - *Numero medio di concatenazioni:* $1/d$
  - *Costo medio:* $\leq h+1+1/d$ *letture + *$1+2+2/d$ *scritture*

B+-tree: Occupazione di memoria

- *Ogni nodo interno contiene al più* $2d$ *valori di chiave e* $2d+1$ *puntatori*
- *L'ordine di un B+-tree è quindi dato da:*

$$d = \left\lfloor \frac{pagesize - PIDsize}{2(keysize + PIDsize)} \right\rfloor$$

- *Per le foglie occorre vedere se l'indice è primario o secondario*
  - *In taluni casi, il livello delle foglie può coincidere con il file dati*

B+-tree: Numero di foglie (primary index)

- *In ogni foglia troviamo al più* $2d$ *entry* (k,r) *e* 1 *o* 2 *puntatori ai nodi vicini*
- *Ne consegue che l'ordine delle foglie è*

$$d_{leaf} = \left\lfloor \frac{pagesize - 2PIDsize}{2(keysize + RIDsize)} \right\rfloor$$

- *Quindi il numero di foglie è*

$$NL = \left\lceil \frac{N}{d_{leaf} \cdot u} \right\rceil$$

- *Il fattore medio di occupazione* u *normalmente vale* $log2$

Altezza di un B$^+$-tree

- *Conoscendo NL, è possibile calcolare l'altezza del B$^+$-tree:*

$$NL_{\min} \leq NL \leq NL_{\max}$$

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

- *Passando ai logaritmi si ha:*

$$\lceil \log_{2d+1}(NL) \rceil + 1 \leq h \leq \left\lfloor \log_{d+1}\left(\frac{NL}{2}\right) \right\rfloor + 2$$

B^+-tree in pratica

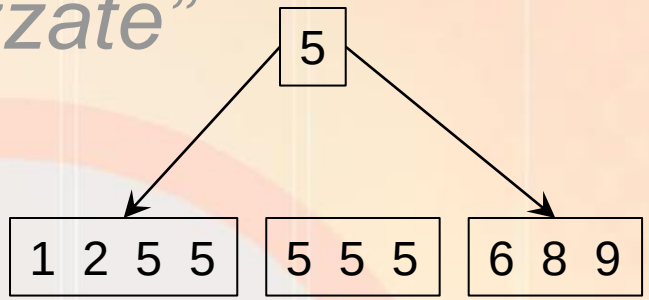
- *Indici secondari*
- *B^+-tree come organizzazione principale dei dati*
- *Chiavi di lunghezza variabile*
- *Compressione delle chiavi*
- *B^+-tree multi-attributo*
- *Bulk-loading*
- *Implementazioni B^+-tree: GiST*

B+-tree come indice secondario

- *In caso di valori duplicati, occorre che ogni valore di chiave sia associato non più ad un RID ma ad una lista di RID*
  - *Tipicamente la lista è ordinata per valori di PID (perché?)*
- *Cosa succede se la lista di RID è molto lunga?*
  - *Può capitare che una singola entry abbia dimensioni maggiori di una singola pagina*
- *Soluzioni possibili:*
  - *Pagine di overflow per la foglia in questione*
  - *Duplicazione delle chiavi nell'indice*
  - *Uso di PID in luogo di RID*
  - *Posting file*

B+-tree: duplicazione delle chiavi

- *Significa che ci sono diverse entry con lo stesso valore della chiave (ma diverso RID)*
- *Occorre modificare leggermente l'algoritmo di ricerca (cercare il primo valore della chiave, quindi proseguire con la lista delle foglie)*
- *Non tutte le foglie sono "indirizzate"*
- *Inefficiente in cancellazione*
  - *Si inserisce il RID come "parte" della chiave*
  - *L'indice diventa "primario"*



B+-tree: uso di PID

- *Invece che mantenere un elenco di RID, si mantiene un elenco di PID, includendo solo le pagine che contengono almeno un record con quel valore della chiave*
  - *PIDsize ≤ RIDsize*
  - *numero di PID ≤ numero di RID*
- *È efficiente se in una pagina ci sono molti record con un valore di chiave*
  - *Indice clustered*

B+-tree: posting file

- *L'elenco dei RID viene mantenuto in un file separato*
- *Ogni entry del posting file ha la forma* (k,l) *dove*
  - k *valore di chiave*
  - l *lista di RID aventi* k *come valore di chiave*
- *Le entry del B+-tree contengono riferimenti alle entry del posting file*
- *Introduce un livello aggiuntivo di indirezione*
  - *Aumenta di 1 il costo di ogni operazione*

B^+-tree come organizzazione principale

- *Le foglie dell'albero contengono direttamente i record dati*
  - *Originariamente, il B-tree fu introdotto proprio come organizzazione principale*
- *Vantaggi:*
  - *Il file dati è ordinato automaticamente*
  - *L'ordinamento viene mantenuto a fronte di inserimenti/cancellazioni*
- *Svantaggi:*
  - *Gli aggiornamenti spostano i record, quindi il RID cambia*

B+-tree: chiavi di lunghezza variabile

- *Tutto quello che abbiamo visto in precedenza si applica se la lunghezza delle entry è fissa*
- *Questo può non essere vero:*
  - *Chiavi di lunghezza variabile (es. varchar)*
  - *Indice secondario*
  - *Indice come organizzazione principale dei dati*
- *In questi casi il concetto di ordine perde di validità e si applicano considerazioni sull'utilizzazione minima dei nodi*

B+-tree: compressione delle chiavi

- *È evidente che, per minimizzare i tempi di accesso all'albero, conviene avere valori elevati di d*
- *Si può pensare di ridurre (comprimere) la lunghezza dei valori della chiave nei nodi*
  - *Nei B+-tree non è necessario che i nodi interni contengano valori di chiave esistenti nei dati*
  - *Devono differenziare il contenuto di nodi adiacenti*
  - *Esempio:*

Semenzara … Serbelloni Mazzanti Vien Dal Mare	Silvani…

"Ser" *è sufficiente…*

B+-tree: ricerche multi-attributo

- *Supponiamo di avere un predicato di ricerca multi-attributo*

```
SELECT * FROM persone
WHERE cognome="Rossi"
AND anno>1990
```

- *Come possiamo utilizzare un indice per risolvere efficientemente la query?*

B+-tree: soluzione a più indici

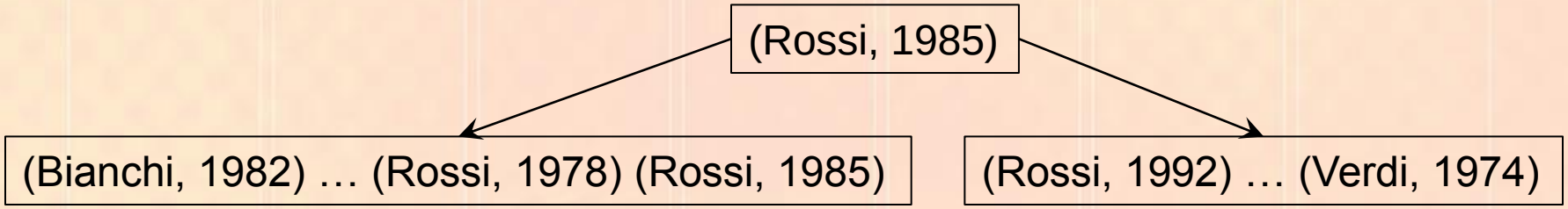
- *Una prima soluzione è quella di usare un solo indice:*
  - *Si recuperano i record che soddisfano il primo (o il secondo) predicato e si verifica l'altro predicato*
- *Una seconda soluzione prevede di usare entrambi gli indici*
  - *Si recuperano separatamente i RID che soddisfano i due predicati*
  - *Si effettua l'intersezione dei RID*
    - *Efficiente se i RID sono mantenuti ordinati*

B$^+$-tree multi-attributo

- *In entrambi i casi il lavoro aggiuntivo può vanificare il vantaggio di usare l'indice*
- *Si può costruire un indice multi-attributo*
- *La chiave è composta dalla concatenazione degli attributi coinvolti*
- *L'ordinamento è quello lessicografico*
  - *Solo a parità di un attributo si considera l'attributo successivo*

B+-tree multi-attributo: esempio

- *Indice su (cognome, anno)*



- *È in grado di risolvere efficientemente interrogazioni su cognome e (cognome, anno)*
  - *Anche interrogazioni di tipo intervallo?*
- *Non su anno (perché?)*
- *Con n attributi sono possibili n! indici*

B+-tree: Bulk-loading

- *Come visto, l'inserimento di elementi in un B+-tree provoca uno split ogni 1/d record inseriti*
- *Spesso la decisione di costruire un indice non avviene in fase di creazione del DB ma in seguito*
  - *Ad esempio, ci accorgiamo che una query è lenta*
- *È conveniente creare un indice su una tabella numerosa effettuando l'inserimento uno a uno?*
  - *Evidentemente no…*

B+-tree: Caricamento

- *Si può creare una lista di coppie (chiave,RID) e ordinarla per valori della chiave*
- *Questa lista (opportunamente paginata) corrisponde al livello delle foglie*
  - *Eventualmente è possibile non usare le pagine al 100%*
- *A partire dalle chiavi contenute in ogni foglia si crea una lista di coppie (chiave,PID)*
- *Questa lista (opportunamente paginata) corrisponde al livello superiore alle foglie*
- *E così via, fino ad ottenere la radice*

Implementare B$^+$-tree: GiST

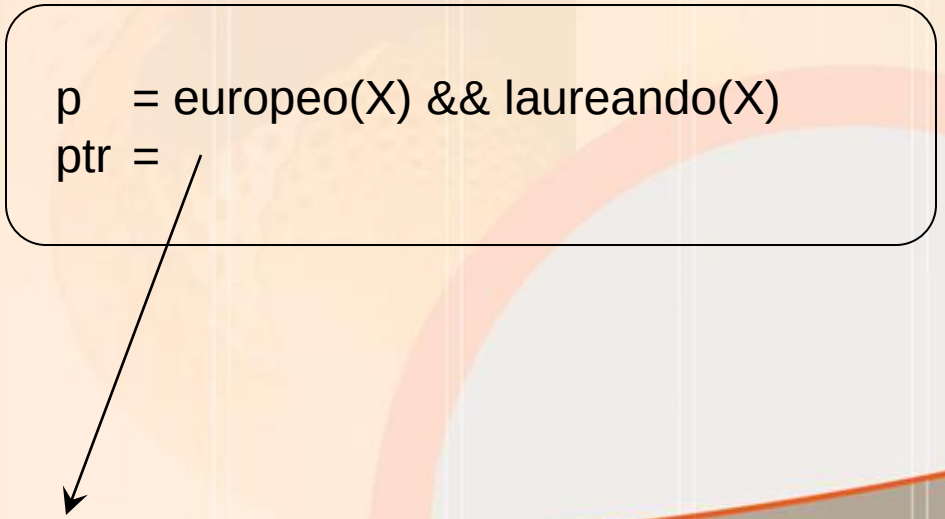
- *GiST (Generalized Search Tree) (Hellerstein, Naughton, Pfeffer, ’95) non è uno specifico metodo di accesso, bensì una struttura generalizzata che, opportunamente istanziata, può comportarsi come un B$^+$-tree, un R-tree, ecc.*
- *L’obiettivo principale non è la definizione di un nuovo tipo di indice, bensì la semplificazione dello sviluppo di diversi metodi di accesso*
  - ***Esempio:** B$^+$-tree nel sistema Postgres consiste di circa 3000 linee di codice C.*
  - *Lo stesso B$^+$-tree implementato come istanza del GiST, richiede circa 500 linee di codice.*

Concetti alla base di GiST

- *Anziché considerare specifiche query, si vede la query come un generico predicato (q)*
- *Ogni nodo del GiST contiene una serie di entry (p,ptr) dove p è un predicato (chiave) e ptr è un puntatore*
- *L'accesso al sottoalbero individuato dal puntatore associato a una chiave avviene solo se la chiave è consistente con il predicato q, ovvero solo se la chiave non esclude la possibilità che il sottoalbero relativo possa contenere dati che soddisfano q*
- *Il vincolo per un GiST è la monotonicità di p, che deve valere per ogni dato accessibile dal relativo sottoalbero*

Monotonicità del predicato

- *Ci chiediamo se, data una query* ***q****, dobbiamo accedere al sotto-albero puntato da* ***ptr***
- *Es.:* ***q=italiano(X)&&studente(X)***
- *Es.:* ***q=messicano(X)&&matricola(X)***
- *Es.:* ***q=francese(X)&&lavoratore(X)***



Proprietà di un GiST

- *Indipendentemente dalla specifica istanziazione, ogni GiST gode delle seguenti proprietà:*
  - *Un GiST è un albero paginato perfettamente bilanciato*
  - *Una entry di un nodo intermedio è una coppia (p,ptr), con*
    - *p predicato usato come chiave di ricerca*
    - *ptr puntatore a un altro nodo del GiST*
  - *Una entry di un nodo foglia è una coppia (p,ptr), con:*
    - *p valore di chiave*
    - *ptr puntatore a una tupla (oggetto del DB) che soddisfa p*
  - *Ogni nodo, ad eccezione della radice, contiene un massimo di M entry e un minimo di f M entry, con 2/M ≤ f ≤ 1/2 (f = minimum fill factor)*
    - *In caso di entry di lunghezza variabile si usano fattori non legati al numero delle entry ma alla loro dimensione*
  - *La radice, se non è una foglia, ha almeno due entry*
  - *Per ogni entry (p,ptr) di un nodo intermedio, p vale per ogni tupla raggiungibile via ptr*

Realizzazione concreta

- *Alla base del GiST vi è la definizione di una serie di metodi, relativi alla gestione:*
  - *dei valori di chiave (Key methods)*
  - *dell'albero (Tree methods)*
- *La definizione del GiST specifica solo i Tree methods*
- *La specifica dei Key methods è eseguita quando si istanzia il GiST per gestire uno specifico tipo di chiavi*
  - *Es.: valori interi (B^+-tree)*
  - *Es.: intervalli multi-dimensionali (R-tree)*
- *Poiché i Key methods sono invocati dai Tree methods, per i primi è comunque necessario standardizzare l'interfaccia*

Key methods: ricerca

- *Consistent(E,q)*
  - ***Input**: Entry E=(p,ptr) e predicato di ricerca q*
  - ***Output**: **if** p & q == false **then** false **else** true*

- *Scopo di Consistent è eseguire un "pruning" dello spazio di ricerca (ovvero eliminare sotto-alberi)*
- *Se il predicato di un sotto-albero non è consistente con la query, si evita di accedere all'intero sotto-albero*

Consistent: commenti

- *Nel caso il test p & q risulti oneroso, è comunque possibile lavorare con approssimazioni conservative, ovvero rispondere "true" anche se p & q = false*
  - *Ciò incide a livello di prestazioni, ma non di correttezza (si accede a un sottoalbero, anche se i suoi dati non contribuiscono al risultato della query)*
- *Consistent (così come gli altri metodi) è specificato per lavorare con predicati di complessità arbitraria*
  - *In pratica, i predicati possono essere "ristretti" per migliorare l'efficienza degli algoritmi*

Consistent nel B$^+$-tree

- *Ogni predicato è un intervallo* $[x,y[$
- *Se la query è un valore v*
  - *Consistent restituisce true se e solo se* $x \leq v < y$
- *Se la query è un intervallo* $[v,w[$
  - *Consistent restituisce true se e solo se* $x < w$ *o* $y > v$

Key methods: creazione del predicato

- *Union(P)*
  - ***Input***: *Insieme di entry* $P = \{(p_1,ptr_1),\ldots,(p_n,ptr_n)\}$
  - ***Output***: *Un predicato* r *che vale per tutte le tuple accessibili tramite uno dei puntatori delle entry*
- *Scopo di* Union *è fornire l'informazione necessaria a caratterizzare il predicato di un nodo padre a partire dai predicati dei nodi figli*
- *In generale r può essere logicamente derivato come un predicato per cui vale*
  $(p_1 \mid \ldots \mid p_n) \Rightarrow r$

Union nel B$^+$-tree

- *Dato* $P = \{([v_1,w_1[,ptr_1),\ldots,([v_n,w_n[,ptr_n)\}$
- *Restituisce* $[\min\{v_1,\ldots,v_n\}, \max\{w_1,\ldots,w_n\}[$

Key methods: compressione delle chiavi

- *Compress(E)*
  - ***Input**: Entry E = (p,ptr)*
  - ***Output**: Entry E'= (p',ptr), con p' rappresentazione compressa di p*
- *Scopo di Compress è fornire una rappresentazione più efficiente del predicato p*
  - *Es.: separatori in luogo di intervalli totalmente ordinati*
  - *Es.: prefissi da stringhe (prefix-B$^+$-tree)*

Key methods: decompressione delle chiavi

- *Decompress(E)*
  - ***Input***: *Entry* $E'=(p',ptr)$, *con* $p' = Compress(p)$
  - ***Output***: *Entry* $E = (r,ptr)$, *con* $p \Rightarrow r$
- *La compressione non è, in generale, lossless*
  - *Es.: prefix-B$^+$-tree*
- *La condizione* $p \Rightarrow r$ *richiede che, se c'è perdita, ciò che si ottiene decomprimendo sia un predicato che vale se vale* p
- *Il caso più semplice è che la decompressione sia la funzione identità*

Key methods: inserimento

- *Penalty(E_1,E_2)*
  - ***Input**: Entries $E_1 = (p_1,ptr_1)$ e $E_2 = (p_2,ptr_2)$*
  - ***Output**: Un valore di "penalità" che risulta dall'inserire E_2 nel sotto-albero con radice E_1*
- *Penalty viene usata dai Tree methods Insert e Split, e serve a confrontare tra loro diverse alternative di modifica dell'albero*

Penalty nel B$^+$-tree

- $E_1 = ([x_1,y_1[,ptr_1)$ e $E_2 = ([x_2,y_2[,ptr_2)$
- Se E_1 è la prima entry del suo nodo
  - *Penalty* restituisce $\max\{y_2 - y_1, 0\}$
- Se E_1 è l'ultima entry del suo nodo
  - *Penalty* restituisce $\max\{x_1 - x_2, 0\}$
- Altrimenti
  - *Penalty* restituisce $\max\{y_2 - y_1, 0\} + \max\{x_1 - x_2, 0\}$

Key methods: split

- *PickSplit(P)*
  - ***Input***: *Insieme di M+1 entry*
  - ***Output***: *Due insiemi di entry, P_1 e P_2, di cardinalità* $\geq f\,M$
- *PickSplit implementa la vera e propria strategia di split, che non viene specificata a questo livello*
- *Tipicamente, si cerca di minimizzare una metrica simile alla Penalty*

Picksplit nel B$^+$-tree

- *P_1 contiene le prime (M+1)/2 entry*
- *P_2 contiene le rimanenti entry*
- *Nel caso di entry di lunghezza variabile si usano criteri legati alla dimensione delle entry e non al loro numero*
  - *Questo può portare a violare il vincolo di utilizzazione minima*

Tree methods

- *I Tree methods si richiamano tra loro e fanno uso dei Key methods definiti*
- *Si assume implicitamente che le chiavi vengano compresse in fase di scrittura, e decompresse in fase di lettura*

Tree methods: architettura

- *Search: usa Consistent*
- *Insert: usa ChooseSubtree, Split e AdjustKeys*
- *ChooseSubtree: usa Penalty*
- *Split: usa PickSplit e Union*
- *AdjustKeys: usa Union*
- *Delete: usa Search e CondenseTree*
- *CondenseTree: usa AdjustKeys e Insert*

Search

- *L'algoritmo di ricerca discende ricorsivamente l'albero, e usa Consistent per eliminare i rami inutili*

```
Search(R,q)
Input:    (sotto-)albero con radice R e query q
Output:  Tutti i record che soddisfano q
if R non è una foglia
    for each E in R
        if Consistent(E,q) Search(*(E.ptr),q)
else for each E in R
        if Consistent(E,q)
            aggiungi *(E.ptr) al risultato
```

Search in domini lineari

- *Per domini lineari (totalmente ordinati), come nel caso dei B+-tree, la specifica del GiST prevede un'estensione più efficiente che sfrutta la contiguità delle foglie per risolvere interrogazioni di range*
- *In particolare, la Search raggiunge la prima foglia "consistente" con la query q*
- *Dopodiché si sfrutta il collegamento a lista delle foglie fino a che non si raggiunge una foglia "inconsistente" con la query q*

Insert

- *L'algoritmo di inserimento viene usato sia per inserire nuove entry, che per reinserire entry "orfane", risultanti da underflow*
- *Per tale motivo viene passato in input anche il livello dell'albero in cui inserire la entry, con la convenzione che le foglie sono a livello 0*
- *In caso di overflow, si attiva la procedura di Split e si propagano i cambiamenti verso l'alto*

Insert

Insert(R,E,l)

***Input**: Albero con radice R, entry E, livello l*

***Output**: Albero con E inserita a livello l*

N = *ChooseSubtree(R,E,l)*

***if** E può essere inserita in N*
    inserisci E in N

***else** Split(R,N,E)*

AdjustKeys(R,N)

ChooseSubtree

- *ChooseSubtree usa Penalty per determinare ricorsivamente il sotto-albero in cui inserire E*

  ChooseSubtree(R,E,l)

  ***Input**: Albero con radice R, entry E, livello l*

  ***Output**: nodo N a livello l in cui inserire E*

  ***if** R è a livello l **return** R*

  ***else** scegli tra tutte le entry F = (p',ptr') in R*
  quella per cui Penalty(F,E) è minimo

  ***return** ChooseSubtree(*(F.ptr'),E,l)*

Split

- *Split usa PickSplit per dividere le entry di un nodo in overflow*
- *Il nodo padre del nodo in overflow si trova sullo stack*

Split(R,N,E)

***Input**: Albero con radice R, nodo N, entry E*

***Output**: Albero con N suddiviso ed E inserita*

P_1,P_2 = *PickSplit({entries di N} ∪ {E})*

inserisci P_1 *in N e* P_2 *in un nuovo nodo N'*

p' = Union(P_2*), ptr' = &N', E' = (p',ptr')*

***if** E' può essere inserita in Parent(N)*

***then** inserisci E' in Parent(N)*

***else** Split(R,Parent(N),E')*

F = entry in Parent(N) con F.ptr = &N

F.p = Union(P_1*)*

AdjustKeys

- *AdjustKeys ricalcola i valori di chiave (predicati) a seguito di modifiche*
- *L'algoritmo risale ricorsivamente l'albero e termina quando si raggiunge la radice o un valore di chiave già accurato*

AdjustKeys(R,N)

***Input**: Albero con radice R, nodo N*

***Output**: Albero con gli antenati di N con valori di chiave corretti e accurati*

***if** N = R o la entry E = (p,ptr), con ptr = &N, è tale che E.p = Union({entry di N})*

return

***else** E.p = Union({entry di N})*

AdjustKeys(R,Parent(N))

Delete

- *Delete mantiene l'albero bilanciato, e ne riduce l'altezza se la radice, al termine di CondenseTree, ha un solo figlio*

Delete(R,E)

***Input**: Albero con radice R, entry E = (p,ptr)*

***Output**: Albero con E rimossa*

Search(R,E.p)

if** E non trovata **return

L = nodo che contiene E, rimuovi E da L

CondenseTree(R,L)

***if** R ha una sola entry*

rimuovi R

rendi il figlio di R la nuova radice del GiST

CondenseTree

- *CondenseTree gestisce il reinserimento al livello originario delle entry orfane di nodi in underflow, mantenute in un insieme*

CondenseTree(R,L)

***Input**: Albero con radice R e una foglia L*

***Output**: Nuovo albero*

N = L, Q = {}

if** N = R **goto end

***else** P = Parent(N), E = entry in P: E.ptr = &N*

***if** #{entry di N} < k M*

Q = Q ∪ {entry di N}, *rimuovi E da P, AdjustKeys(R,P)*

***if** E non è stata rimossa da P AdjustKeys(R,N)*

else** N = P, **restart

***for each** E in Q Insert(R,E,level(E))*

Valutazione delle prestazioni

- *Finora abbiamo calcolato le prestazioni in ricerca per un B^+-tree usato come indice primario*
- *Nel caso di indice secondario dobbiamo capire*
  - *Quante siano le foglie contenenti entry nel risultato*
  - *Quante siano le pagine dati contenenti i record*
- *Supporremo che*
  - *Le liste dei riferimenti nelle foglie siano ordinate*
    - *Non accediamo più di una volta ad ogni pagina dati*
  - *I valori degli attributi siano distribuiti uniformemente nel file dati*
    - *Ogni valore è ripetuto (in media) N/K volte*
  - *I record siano distribuiti uniformemente nelle pagine del file dati*

Stima del numero di pagine

- *Se R sono i record da reperire in P pagine*
  - *$1/P$ = probabilità che uno dei record si trovi in una data pagina*
  - *$1–1/P$ = probabilità che la pagina non contenga il record*
  - *$(1–1/P)^R$ = probabilità che la pagina non contenga alcun record*
  - *$1–(1–1/P)^R$ = probabilità che la pagina contenga almeno un record*
- *Moltiplicando per il numero delle pagine otteniamo il numero medio di pagine cui accedere*

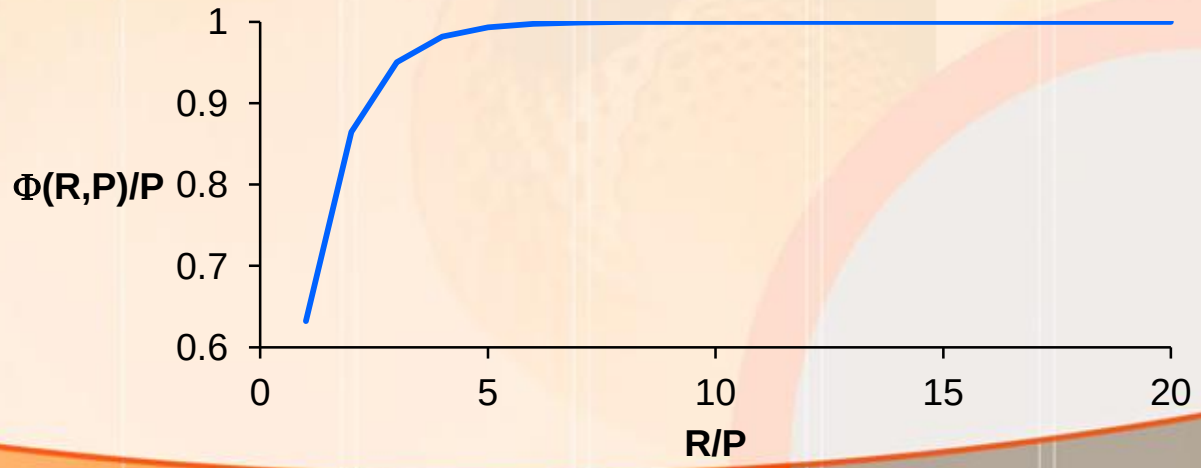
$$\Phi(R,P) = P\,(1–(1–1/P)^R) \leq \min\{R,P\}$$

- *Dato un cassetto contenente calzini di P colori, ogni colore ripetuto un infinito numero di volte, quanti colori distinti risultano, in media, dall'estrazione di R calzini?*

Modello di Cardenas

- *Il modello appena visto (Cardenas) assume pagine di capacità infinita (si noti che N non figura come argomento)*
  - *Porta a sottostimare apprezzabilmente il valore corretto nel caso di pagine con meno di circa 10 record*
- *Se R = N, la formula restituisce un valore minore di P*

$$\Phi(N,P) = P\,(1-(1-1/P)^N) \approx P\,(1-e^{-N/P})$$



Modello di Yao

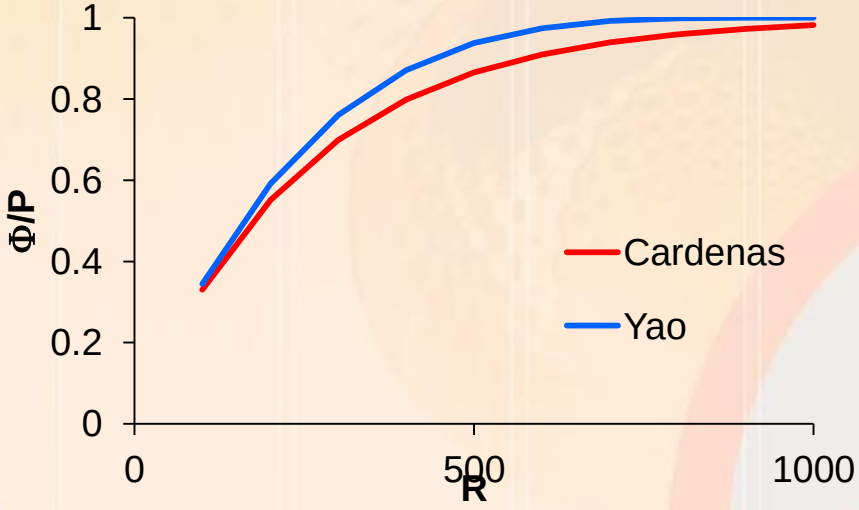
- *Il modello di Yao tiene conto della capacità effettiva* $C = N/P$ *delle pagine*
- *La sua derivazione considera tutti i modi possibili in cui si possono trovare allocati gli* R *record richiesti sulle* P *pagine*
  - $\binom{N}{R}$ *= numero di combinazioni*
  - $\binom{N-C}{R}$ *= numero di combinazioni escludendo una pagina*
  - $\binom{N}{R} - \binom{N-C}{R}$ *= combinazioni che interessano una pagina*
  - $1 - \frac{\binom{N-C}{R}}{\binom{N}{R}}$ *= probabilità che la pagina contenga almeno un record*

Formula di Yao

- *Moltiplicando per il numero di pagine otteniamo il valor medio del numero di pagine accedute*

$$\Phi(R,N,C) = NP \times \left(1 - \frac{\binom{N-C}{R}}{\binom{N}{R}}\right)$$

- *Es.: N=1000, P=250*



Confronto tra i modelli

- *Nel caso di pagine con numero variabile di record si può dimostrare che la formula di Yao sovrastima*
- *Se l'allocazione dei record non è casuale entrambi i modelli sovrastimano.*
- *Se R è grande il calcolo della formula di Yao può richiedere tempi elevati*

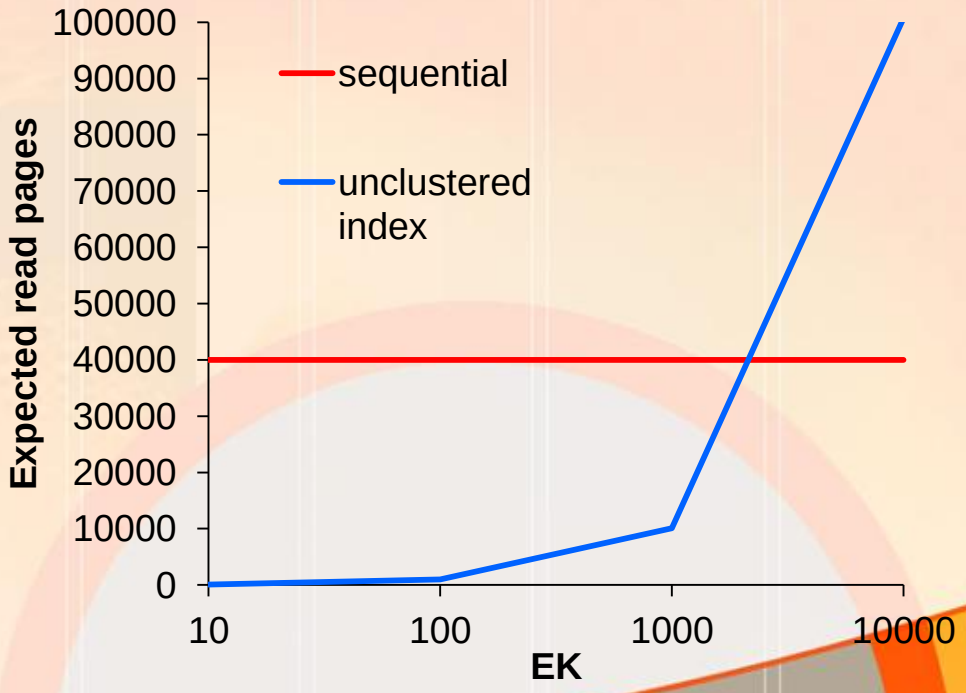
$$\Phi(R,N,C) = NP \times \left(1 - \prod_{i=1}^{R} \frac{N-C-i+1}{N-i+1}\right)$$

Costo dell'accesso con indice

- *Costo totale = costo indice + costo pagine dati*
- *Costo indice = costo per la prima foglia + costo per leggere tutte le foglie*
  - *Costo per la prima foglia = h-1*
  - *Numero di foglie = $\lceil L*EK/K \rceil$*
- *Costo pagine dati: EK volte la formula di Cardenas (o Yao)*
  - *= EK x $\Phi(N/K,P)$*
- *Questo va confrontato col costo sequenziale*
  - *= P*

Esempio

- *File con $N = 10^6$ record su $P = 40000$ pagine*
- *Indice unclustered su campo con $K = 10^5$ valori con $L = 7045$ foglie e altezza $h = 4$*
- *Si nota come l'altezza dell'albero sia del tutto ininfluente*



Costo per ricerca con intervallo A∈[x,y]

- *Costo totale = costo indice + costo pagine dati*
- *Costo indice = costo per la prima foglia + costo per leggere "sequenzialmente" le foglie*
  - *Costo per la prima foglia =* $h-1$
  - *Numero di foglie =* $\lceil fs*L \rceil$
    - fs *= fattore di selettività del predicato =* $(y-x)/(maxA-minA)$
- *Costo pagine dati:* $\lceil fs*K \rceil$ *volte la formula di Cardenas (o Yao)*
  - $= \lceil fs*K \rceil \Phi(N/K,P)$ *(attributo non di ordinamento)*
  - $= \lceil fs*P \rceil$ *(attributo di ordinamento)*

Organizzazioni hash

- *A differenza delle tecniche di tipo "tabellare", in cui l'associazione <chiave, RID> è mantenuta in forma esplicita, un'organizzazione hash utilizza una funzione hash, H, che trasforma ogni valore di chiave in un indirizzo.*

Organizzazioni hash: collisioni

- *Salvo casi particolari, le funzioni hash non sono iniettive:* $k_1 \neq k_2 \not\Rightarrow H(k_1) \neq H(k_2)$

  quindi possono verificarsi collisioni
  - *Se* k_1 *e* $k_2 \neq k_1$ *collidono,* $H(k_1) = H(k_2)$
  - *Una funzione hash che non genera collisioni si dice perfetta*
- *Ogni indirizzo generato dalla funzione hash individua una pagina logica, o bucket*
- *Il numero di "elementi" (valori di chiave se l'organizzazione è un indice, record dati se l'organizzazione è primaria) che possono essere allocati nello stesso bucket determina la capacità,* C*, dei bucket*

Organizzazioni hash: overflow

- *L'area di memoria costituita dai bucket indirizzabili dalla funzione hash è detta area primaria*
- *Se una chiave viene assegnata a un bucket che già contiene C chiavi, si ha un overflow (o trabocco)*
- *La presenza di overflow può richiedere, a seconda della specifica organizzazione, l'uso di un'area di memoria separata, detta appunto area di overflow*

Organizzazioni hash statiche e dinamiche

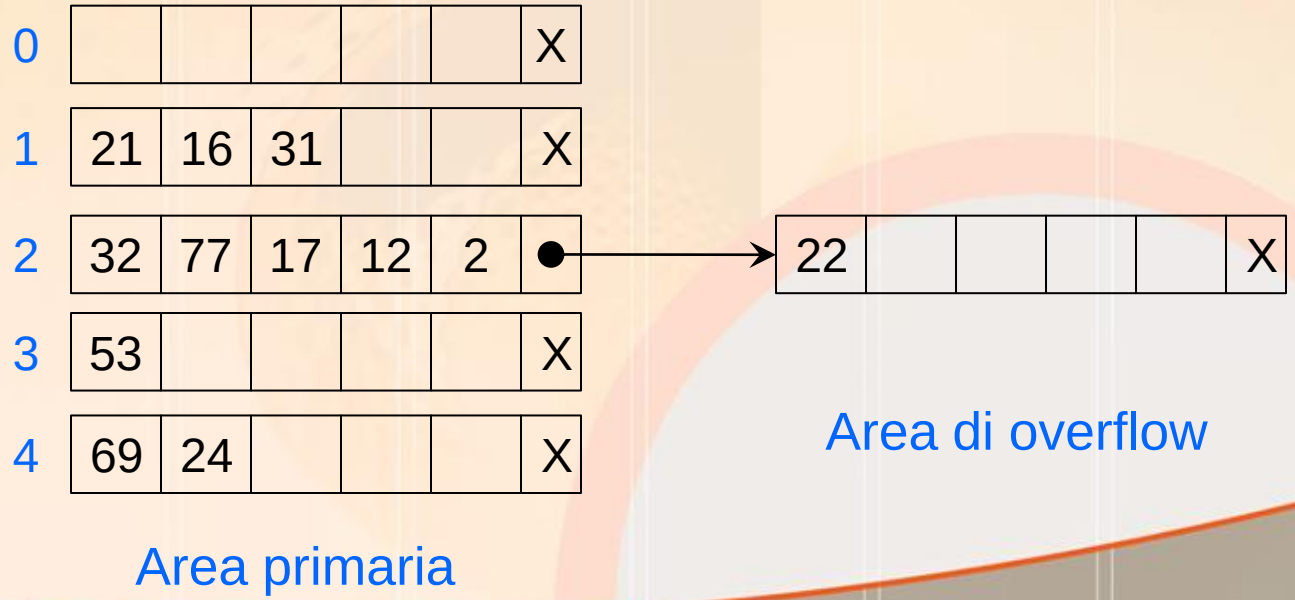
- *Una funzione hash deve essere suriettiva, e quindi generare P indirizzi, tanti quanti sono i bucket dell'area primaria*
- *Se il valore di P è, per una data organizzazione, costante, l'organizzazione è detta statica*
  - *In questo caso, il dimensionamento dell'area primaria è parte integrante del progetto dell'organizzazione*
- *Viceversa, se l'area primaria può espandersi e contrarsi, per meglio adattarsi al volume effettivo dei dati da gestire, allora l'organizzazione è detta dinamica*
  - *In questo caso si rendono necessarie più funzioni hash*
- *Le prime organizzazioni hash dinamiche nate intorno alla fine degli anni '70, mentre quelle statiche sono state sviluppate a partire dagli anni '50*

Caratteristiche delle organizzazioni hash

- *Per entrambi i tipi di organizzazione, vi sono aspetti comuni che meritano considerazione:*
  - *Scelta della funzione hash* H
  - *Politica di gestione degli overflow*
  - *Capacità* C *dei bucket dell'area primaria*
  - *Capacità* C_{ov} *dei bucket dell'eventuale area di overflow (non necessariamente uguale a* C*)*
  - *Utilizzazione della memoria allocata*
- *Le organizzazioni hash sono generalmente primarie (ma anche secondarie)*
- *Le funzioni hash normalmente non preservano l'ordine, ovvero non sono funzioni monotone*
  - *L'uso di indici hash è quindi fortemente sconsigliato nei casi in cui sono possibili interrogazioni di intervallo*

Static hashing

- *Un semplice esempio di organizzazione hash statica è mostrato in figura, in cui:*
  - *Le chiavi sono numeri naturali*
  - *L'area primaria consiste di $P = 5$ bucket di capacità $C = 5$*
  - *La funzione hash è: $H(k_i) = k_i \% 5$*
  - *Gli overflow sono gestiti allocando, per ogni bucket dell'area primaria, uno o più bucket di overflow, di capacità $C_{ov} = 5$, collegati a lista*



Static hashing: analisi dei costi

- *Un file hash con* N *record in bucket di capacità* $C = C_{ov}$, *e la cui area primaria consiste di* P *bucket, comporta, nell'ipotesi di perfetta ripartizione dei record sullo spazio dei* P *indirizzi, che:*
  - *ogni indirizzo è generato* N/P *volte*
  - *ogni catena consiste di* $N/(P*C)$ *bucket*
- *Il costo di ricerca di un record risulta pertanto proporzionale a* $N/(P*C)$
- *Esempio:* $N = 10^6$, $C = 10$, $P = 25000$
  - *Una ricerca (con successo) accede in media a* 2 *bucket*
  - *Questo è anche il numero di operazioni di I/O, se ogni bucket può essere letto con un solo I/O*

Funzioni hash

- *Una funzione hash è una trasformazione (suriettiva) dallo spazio, K, delle chiavi allo spazio, {0,...,P - 1}, degli indirizzi*
- *L'ipotesi che un arbitrario sottoinsieme di K si ripartisca sui P indirizzi in maniera perfettamente omogenea è una pura astrazione, di scarsa utilità per analizzare le prestazioni ottenibili dalle diverse organizzazioni hash*
- *Il caso ideale rispetto al quale è ragionevole confrontare una specifica funzione hash H è quello di distribuzione uniforme sullo spazio degli indirizzi, in cui, per ogni sottoinsieme di K, ognuno dei P indirizzi ha la stessa probabilità, 1/P, di essere generato*

Funzioni hash: distribuzione uniforme

- *Nel caso ideale il numero di chiavi, X_j, assegnate al bucket j-esimo segue una distribuzione binomiale*

$$\Pr\{X_j = x_j\} = \binom{N}{x_j}\left(\frac{1}{P}\right)^{x_j}\left(1-\frac{1}{P}\right)^{N-x_j}$$

con valor medio μ e varianza σ^2 dati da:

$$\mu = \frac{N}{P} \qquad \sigma^2 = \frac{N}{P}\left(1-\frac{1}{P}\right)$$

in cui né μ né σ^2 dipendono dallo specifico bucket

- *Per $P >> 1$, il rapporto $\sigma/\sqrt{\mu}$ vale circa 1*

Qualità di una funzione hash

- *Nel caso di funzioni hash "reali" le prestazioni variano al variare dello specifico set di chiavi*
- *Esempio: La funzione* $H(k_i) = k_i \% P$ *è una "buona" funzione, ma nel caso del set di chiavi* $\{0, P, 2P, 3P, \ldots, N \cdot P\}$ *alloca tutte le chiavi nel bucket 0*
- *Ogni funzione hash, scelta indipendentemente dallo specifico set di chiavi, può dar luogo a prestazioni disastrose, nel caso peggiore.*
- *Nel caso "medio", tuttavia, considerando arbitrari sottoinsiemi di* K *e file dati reali, si osserva che le diverse funzioni hash si comportano effettivamente in modo diverso*

Degenerazione

- *Un criterio adeguato di valutazione di una funzione H, riferito a un particolare insieme di chiavi, è dato dall'analisi della sua degenerazione* $\sigma/\sqrt{\mu}$*, dove:*

$$\mu = \sum_{j=0}^{P-1} \frac{x_j}{P} = \frac{N}{P} \qquad \sigma^2 = \sum_{j=0}^{P-1} \frac{(x_j - \mu)^2}{P}$$

sono calcolati su tutti i P bucket e x_j *è il numero di record osservato nel* j*-esimo bucket*

- *Quanto più bassa è la degenerazione, tanto migliore è il comportamento della funzione hash*

Funzioni hash: mid square

- *La chiave è moltiplicata per se stessa*
- *Viene estratto un numero di cifre centrali pari a quelle di* $P - 1$
- *Il numero ottenuto è normalizzato a* P

$$145142^2 = 21066200164$$

- *Se, ad esempio,* $P = 8000$*, la normalizzazione produce l'indirizzo:* $\lfloor 6620 \times 0.8 \rfloor = 5296$

Funzioni hash: shifting

- *La chiave è suddivisa in un certo numero di parti, ognuna costituita da un numero di cifre pari a quelle di* $P - 1$
- *Si sommano le parti e si normalizza il risultato*
- *Es.:* $P = 800$, $k = 14514387$

$$387+514+14=915$$

- *La normalizzazione produce* $\lfloor 915 \times 0.8 \rfloor = 732$

Funzioni hash: folding

- *La chiave è suddivisa come per lo shifting*
- *Le parti vengono "ripiegate" e sommate, prima di normalizzare il risultato*
- *Es.: P = 800, k =14514387*

  783+514+41=1338
- *La normalizzazione produce:*

  1338%800 = 538

  $\lfloor 538 \times 0.8 \rfloor = 430$

Funzioni hash: divisione

- *La chiave numerica viene divisa per un numero Q e l'indirizzo è ottenuto considerando il resto:*

$$H(k) = k \% Q$$

- *Per la scelta di Q si hanno le seguenti indicazioni pratiche:*
  - *Q è il più grande numero primo minore o uguale a P*
  - *Q è non primo, minore o uguale a P, con nessun fattore primo minore di 20*
  - *Se $Q < P$, si deve porre $P = Q$ per non perdere la suriettività della funzione hash*

Funzioni hash: divisione (esempio)

- *P = 6997*
- *k = 172146* *H(k) = 4218*
- *172147* *4219*
- *172148* *4220*
- *172149* *4221*
- *…* *…*
- *174924* *6996*
- *174925* *0*

Chiavi alfanumeriche

- *Il trattamento di stringhe alfanumeriche richiede una fase preliminare di conversione.*
- *Uno dei metodi più comuni è quello di stabilire*
  - *Un alfabeto* A*, cui appartengono i caratteri delle stringhe*
  - *Una funzione biiettiva* $ord(\ )$*, che associa a ogni elemento dell'alfabeto un intero nel range* $[1,|A|]$
  - *Una base di conversione* b
- *Una stringa* $S = s_{n-1}, \ldots, s_i, \ldots, s_0$ *viene quindi convertita in una chiave numerica*

$$k(S) = \sum_{i=0}^{n-1} ord(s_i) \times b^i$$

Chiavi alfanumeriche: esempio

- *Posto* $A = \{a,b,\ldots,z\}$, *ord() a valori in* $[1,26]$, *e* $b = 32$, *la stringa "indice" produce il valore*

$$k(\text{"indice"}) = 9x32^5+14x32^4+4x32^3+9x32^2+3x32^1+5x32^0 = 316810341$$

- *Metodi più semplici, che non fanno uso di una base, quali ad esempio*

$$k(S) = \sum_{i=0}^{n-1} ord(s_i)$$

funzionano meno bene, in quanto possono generare la stessa chiave numerica a partire da stringhe distinte, una l'anagramma dell'altra

Scelta della base

- *Con il metodo della divisione si possono avere seri problemi se la base b ha fattori primi in comune con P*
- *Es.: Sia* $A = \{a,b,\ldots,z\}$, $b = 32$ *e* $P = 512$.
- *Il valore di* $H(k(S))$ *è determinato solo dagli ultimi due caratteri:*
  - *"folder": i valori ordinali sono 6, 15, 12, 4, 5, 18,* $k(\text{"folder"})=217,452,722$, $H(217452722)=\mathbf{178}$
  - *"primer": gli ordinali sono 16, 18, 9, 13, 5, 18,* $k(\text{"primer"})=556,053,682$, $H(556053682)=\mathbf{178}$

Scelta della base (cont.)

- *Per capire perché insorge questo tipo di problema, è necessario rifarsi a una serie di proprietà dell'operatore %*
- *Il caso più semplice da considerare è quello in cui* P *è un multiplo di* b*, ovvero* $P = \alpha \times b$
- *Esisterà un valore* y *tale per cui* $b^y \% (\alpha \times b) = 0$
- *Poiché per l'operatore % vale la proprietà (utile per il calcolo di* $k(S)$*):*

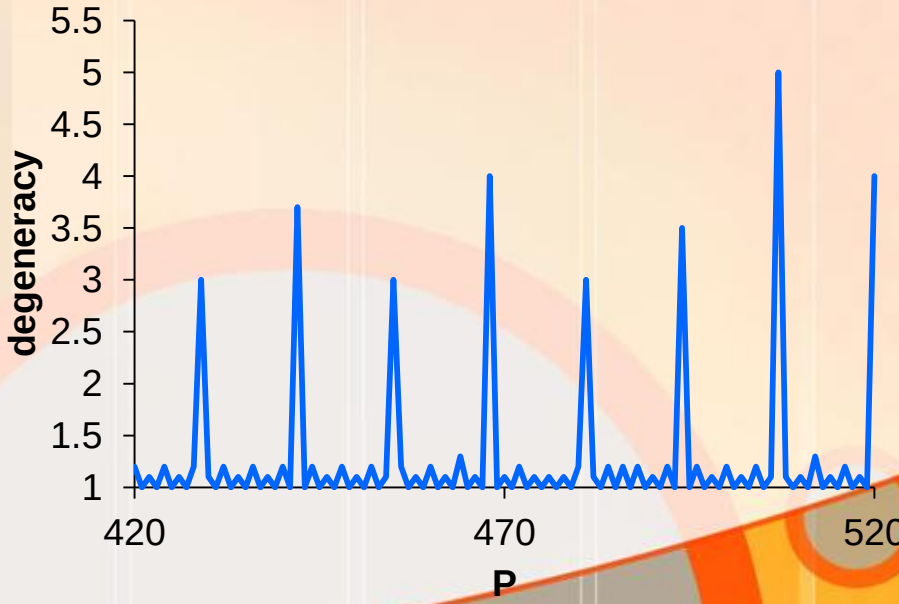
$$H(k(S)) = \left[\sum_{i=0}^{n-1} ord(s_i) \times b^i\right] \% (\alpha \times b) = \left(\sum_{i=0}^{n-1} \left[\left(ord(s_i) \times b^i\right) \% (\alpha \times b)\right]\right) \% (\alpha \times b)$$

i caratteri s_{n-1} *a* s_y *danno contributo = 0 al valore di* $H(k(S))$ *(nell'es.* $y = 2$*), e quindi la stringa "utile" è lunga solo* y *caratteri*

- *Con base* 26 *si hanno problemi quando* P *ha come fattori* 13 *e* 2

Analisi di un caso

- *L'esperimento in esame è stato eseguito da Mullin nel 1991 facendo uso, come set di chiavi, di tutte le parole di 6 caratteri, costruite sull'alfabeto* $A = \{a,b,\ldots,z\}$*, usate dallo spelling checker di Unix*
- *La figura si riferisce al caso* $b = 26$ *con* P *che varia tra* 420 *e* 520
  - *I picchi della distribuzione si hanno per valori di* P *multipli di* 13
  - *In particolare, il massimo si ottiene per* $P = 507 = 13 \times 13 \times 3$



Fattore di caricamento

- *Data una stima del numero N di record da gestire, e fissata la capacità C dei bucket, la scelta di un determinato fattore di caricamento d determina il numero di bucket, P, in area primaria*
- *Va tenuto presente che, al diminuire di d, diminuisce la percentuale di record in overflow.*
  - *Quindi non è consigliabile utilizzare fattori di caricamento elevati*
- *Valori tipici, che rappresentano un buon compromesso tra utilizzazione della memoria e costi di esecuzione delle operazioni, si hanno nell'intervallo* [0.7,0.8]

Capacità dei bucket

- *È evidente che avere una capacità tale da richiedere più operazioni di I/O per il trasferimento di un bucket non comporta alcun vantaggio*
- *Il motivo per cui è invece conveniente avere bucket di capacità $C > 1$ è essenzialmente legato alla relazione esistente tra C e la percentuale di record in overflow*
- *All'aumentare di C, e a parità di fattore di caricamento d, la percentuale di record in overflow diminuisce, sotto le ipotesi di una funzione hash ideale e di gestione degli overflow in area separata*
  - *Sperimentalmente, il risultato si dimostra valido anche nel caso di funzioni hash non ideali*

Capacità dei bucket ottimale

- *Poiché un aumento del numero di record in overflow tende a deteriorare le prestazioni, è consigliabile lavorare con bucket di capacità C massima, soggetta ai vincoli:*
  - *la lettura di un bucket deve comportare una singola richiesta di I/O (blocchi contigui)*
  - *il trasferimento di un bucket di capacità C deve poter avvenire in un tempo minore rispetto al trasferimento di due bucket di capacità minore di C*

Calcolo del numero medio di overflow

- *Il numero di volte che viene generato l'indirizzo* j *è una variabile aleatoria* X_j *con distribuzione binomiale, il numero medio di overflow nel* j*-esimo bucket è:*

$$OV_j(C) = \sum_{x_j = C+1}^{N} (x_j - C) \times \Pr\{X_j = x_j\}$$

che non dipende dal bucket j *specifico*

  - *Per brevità scriveremo* $\Pr(x)$ *in luogo di* $\Pr\{X_j = x_j\}$

Distribuzione degli overflow

- *Il numero totale di overflow si ottiene come:*

$$OV(C) = \sum_{j=0}^{P-1} OV_j(C) = P \times \sum_{x_j=C+1}^{N} (x_j - C) \times \Pr(x)$$

- *Per valori elevati di N e P, è possibile approssimare la distribuzione binomiale con quella di Poisson:*

$$\Pr(x) = \binom{N}{x} \left(\frac{1}{P}\right)^x \left(1 - \frac{1}{P}\right)^{N-x} \approx \left(\frac{N}{P}\right)^x \frac{e^{-\frac{N}{P}}}{x!}$$

- *Essendo P = N/(C x d):*

$$\Pr(x) \approx (C \times d)^x \frac{e^{-C \times d}}{x!}$$

Numero totale di overflow

- *Il numero totale di overflow si ottiene quindi come:*

$$OV(C) \approx P \times \sum_{x=C+1}^{N} (x - C) \times \frac{(C \times d)^x e^{-(C \times d)}}{x!}$$

- *Con il cambio di variabile* $i = x - C$*:*

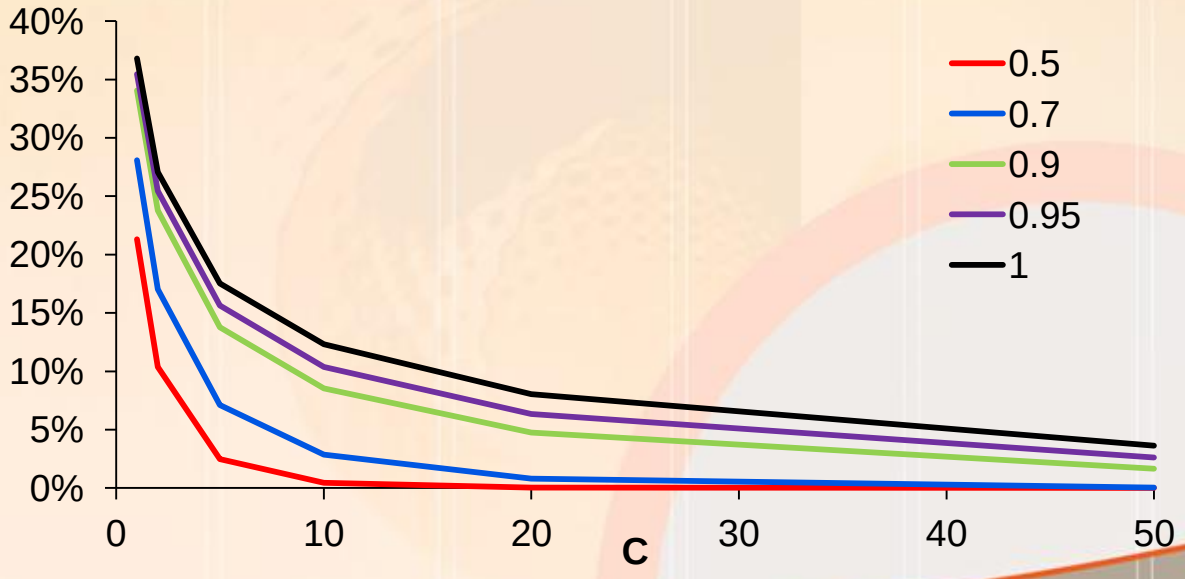
$$OV(C) \approx P \times \frac{(C \times d)^{C+1} e^{-(C \times d)}}{C!} \sum_{i=1}^{N-C} \frac{i \times C^{i-1} \times d^{i-1}}{(C+1)(C+2)\ldots(C+i)}$$

- *Ricordando nuovamente* $P = N/(C \times d)$*:*

$$OV(C) \approx N \times \frac{(C \times d)^{C} e^{-(C \times d)}}{C!} \times f(C,d)$$

Esempio

C\d	0.5	0.7	0.9	0.95	1
1	0.213061	0.280836	0.340633	0.354464	0.367879
2	0.103638	0.170307	0.237853	0.254377	0.270669
5	0.02478	0.071143	0.137768	0.156336	0.17531
10	0.004437	0.028736	0.085344	0.103778	0.123244
20	0.000278	0.008014	0.047641	0.063497	0.080492
50	2.51E-07	0.000496	0.016561	0.026191	0.03622



Gestione degli overflow

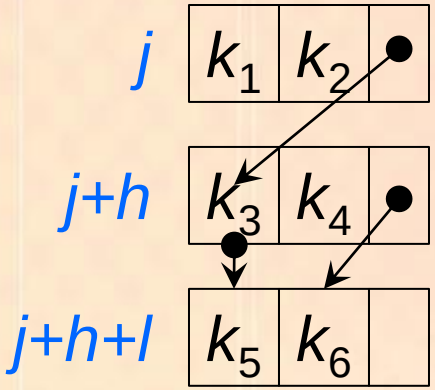
- *I metodi studiati per la gestione degli overflow mirano a ridurre al minimo gli accessi a bucket necessari per reperire il record cercato*
- *Esistono due tipi di strategie*
  - *Metodi di concatenamento (chaining)*
    - *Usano puntatori*
    - *Possono usare o meno l'area di overflow*
  - *Metodi di indirizzamento aperto (open addressing)*
    - *Non usano puntatori*
    - *Usano bucket in area primaria per memorizzare i record in overflow*

Concatenamento in area primaria

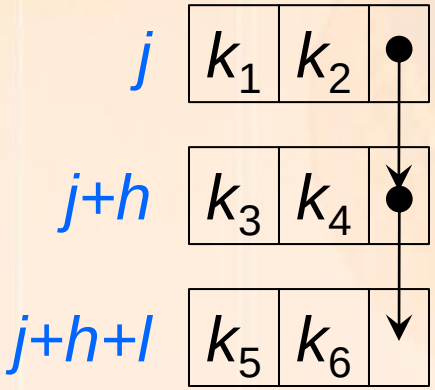
- *Liste separate*
  - *Se il bucket j è in overflow, si inserisce il nuovo record nel primo bucket non pieno successivo a j*
  - *I record in overflow sono collegati a lista*
  - *Anche i record non in overflow vanno collegati*
- *Liste confluenti (coalesced chaining)*
  - *Si usa un solo puntatore per ogni bucket (non per ogni record)*
  - *Le liste dei bucket possono fondersi*
    - *Es.: j va in overflow in j+h e j+h va in overflow, entrambe continuano in j+h+l*

Esempi

- *Liste separate*



- *Coalesced chaining*



k	*H(k)*
k_1	*j*
k_2	*j*
k_3	*j*
k_4	*j+h*
k_5	*j*
k_6	*j+h*

- *Il metodo a liste confluenti semplifica la gestione dei puntatori ma peggiora le prestazioni*

Concatenamento in area di overflow

- *Come detto, la capacità dei blocchi nell'area di overflow* C_{ov} *può essere diversa dalla capacità dell'area primaria* C
- *In genere* $C_{ov} < C$ *per evitare sprechi quando gli overflow sono ridotti*
- *Per lo stesso motivo si possono avere liste confluenti*
- *Ovviamente, in caso di overflow di un bucket di overflow, si otterrà una lista di bucket di overflow*

Open addressing

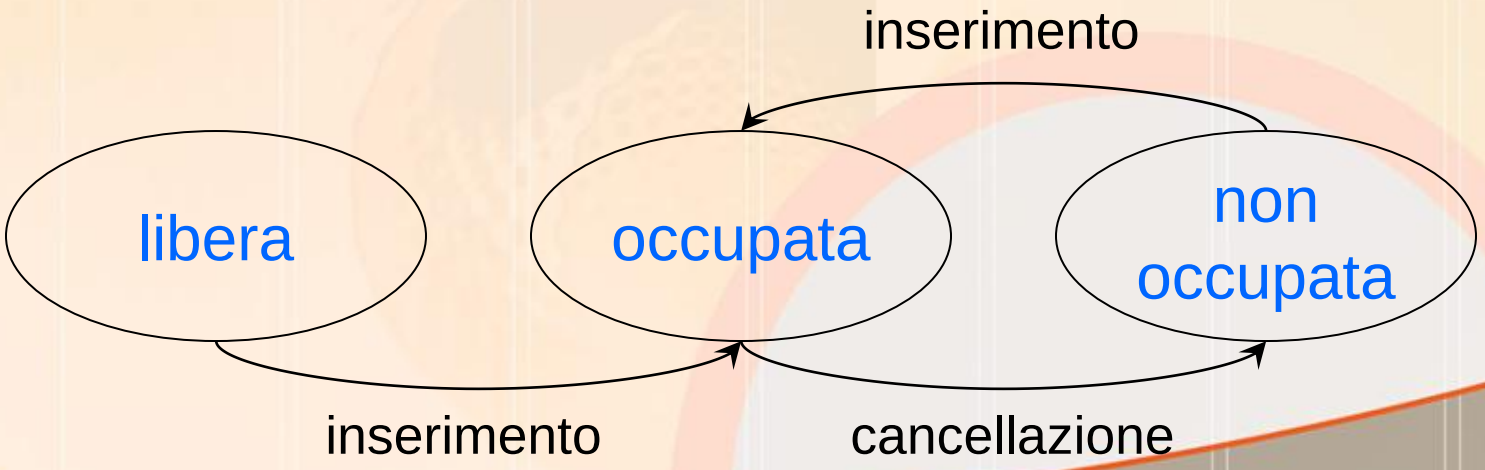
- *Nei metodi a indirizzamento aperto, a ogni valore di chiave* k_i *viene associata una sequenza di indirizzi*
  $H_0(k_i), H_1(k_i), \ldots, H_l(k_i)$ *con* $H_0(k_i) = H(k_i)$
- *Quando si inserisce* k_i*, vengono provati tutti gli indirizzi* $H_0(k_i), H_1(k_i), \ldots, H_l(k_i)$ *fino a quando non si trova l'indirizzo di un bucket non pieno*

Open addressing: ricerca

- *Per cercare k_i occorre cercare in tutti gli indirizzi $H_0(k_i)$, $H_1(k_i)$, …, $H_l(k_i)$ fino a quando*
  - *O non si trova k_i (ricerca con successo)*
  - *O si trova un bucket non pieno (ricerca con insuccesso)*
- *Nei metodi a indirizzamento aperto, occorre quindi prestare particolare attenzione a come si operano le cancellazioni*
  - *Se si cancella un record il corrispondente bucket diventa non pieno e quindi "interrompe" la ricerca*

Open addressing: cancellazione di record

- *La posizione di un record cancellato viene quindi marcata come "non occupata", e può diventare "occupata" (riutilizzata) in seguito a nuovi inserimenti*
- *Un bucket non è pieno se e solo se tutte le posizioni nel bucket sono "occupate"*



Scansione lineare (linear probing)

- *Ad ogni passo l'indirizzo è aumentato di un passo costante* s
  - $H_j(k_i) = (H_{j-1}(k_i) + s) \% P$
  - *Quindi:* $H_j(k_i) = (H(k_i) + s \times j) \% P$
- *Perché vengano generati tutti gli indirizzi occorre che* s *non abbia divisori comuni con* P
  - *Altrimenti sono generati solo* $P/MCD(P,s)$ *indirizzi*
- *Es.:* $P=10$, $s=4$, $MCD(10,4)=2$
  - *Gli indirizzi di* $k_i = 3$ *sono* $3, 7, 1, 5, 9, 3$

Clustering primario

- *Se un bucket j va in overflow,*
  è probabile che il cluster j+s vada in overflow,
  quindi che il cluster j+2s vada in overflow…
- *C'è un addensamento di record*
  in alcune pagine
  - *Es.: P=31, s=3, vengono generati gli indirizzi:*
    - *1234 → 25, 28, 0, 3, 6, 9, 12, …*
    - *245 → 28, 0, 3, 6, 9, 12, 15, …*
- *Il problema è dovuto alla linearità del passo di scansione s*

Scansione quadratica

- *Ad ogni passo l'indirizzo è aumentato di un passo lineare* $a + b(2j - 1)$
  - $H_j(k_i) = (H_{j-1}(k_i) + a + b(2j - 1)) \% P$
  - *Quindi:* $H_j(k_i) = (H(k_i) + a \times j + b \times j^2) \% P$
  - *Es.:* $P=31$, $a=3$, $b=5$, *vengono generati gli indirizzi:*
    - *1234* → *25, 2, 20, 17, 24, 6, …*
    - *245* → *28, 5, 23, 20, 27, 13, …*
  - *Quindi le liste non confluiscono (es., si veda 20)*
- *Rimane il problema del clustering secondario, dovuto a chiavi aventi lo stesso primo bucket*

Double hashing

- *Si cerca di eliminare i problemi dovuti al clustering secondario facendo uso di due funzioni hash, H' e H"*
- *Le sequenze di indirizzi sono date da:*
  - $H_0(k_i) = H'(k_i)$
  - $H_j(k_i) = (H_{j-1}(k_i) + H''(k_i)) \% P$ *(se* $j>0$*)*
- *Due chiavi generano ora la stessa sequenza di indirizzi se e solo se collidono sia con H' sia con H"*

Double hashing: considerazioni

- *La tecnica di double hashing ha, come effetto collaterale, quello di produrre una forte variabilità degli indirizzi generati, dipendentemente dal valore di* $H''(k_i)$
- *Considerando l'effettiva allocazione dei bucket in memoria secondaria ciò può appesantire notevolmente le operazioni di I/O*
  - *I bucket successivi sono "lontani" quindi aumenta la latenza*
- *Il double hashing approssima abbastanza bene il caso ideale di "hash uniforme" (random probing), in cui ogni indirizzo ha la stessa probabilità di essere generato al passo j-esimo*

Confronto tra i vari metodi

- *Per i metodi che non hanno l'area di overflow l'utilizzazione* u *è pari a* d
- *Con bucket di overflow si ottiene:*

$$u = \frac{N}{P \times C + P_{ov} \times C_{ov}}$$

Prestazioni in ricerca

- *Coalesced chaining*
  - *Ricerca con successo:* $E \approx 1 + \frac{1}{8d}\left(e^{2d} - 1 - 2d\right) + \frac{d}{4}$
  - *Ricerca con insuccesso:* $A \approx 1 + \frac{1}{4}\left(e^{2d} - 1 - 2d\right)$
- *Separate chaining con overflow area*
  - *Ricerca con successo:* $E \approx 1 + d/2$
  - *Ricerca con insuccesso:* $A \approx e\text{-}d + d$

Prestazioni: open addressing

- *Random probing: ricerca con insuccesso*
  - *Se abbiamo bucket di capacità unitaria, il costo di ricerca è pari al numero di bucket cui bisogna accedere per inserire un nuovo record*
  - *La probabilità di insuccesso ad ogni bucket è* d
  - *La probabilità di aver trovato* $r-1$ *bucket occupati* $\Pr\{costo=r\}$ *è quindi pari a* $(1-d)\,d^{r-1}$
  - *La distribuzione è geometrica con valor medio* $d/(1-d)$
  - *Il costo medio di insuccesso è quindi:* $A = d/(1-d)+1 = 1/(1-d)$

Prestazioni: open addressing

- *Random probing: ricerca con successo*
  - *Il numero medio di accessi è pari al costo medio per inserire tutti i record*

$$E \approx \frac{1}{N}\sum_{i=0}^{N-1}\frac{1}{1-i/P} = \frac{P}{N}\sum_{i=0}^{N-1}\frac{1}{P-i}$$

  - *Si può riscrivere la sommatoria come differenza di due somme armoniche*

$$E \approx \frac{P}{N}\left(\sum_{i=1}^{P}\frac{1}{i} - \sum_{i=1}^{P-N}\frac{1}{i}\right)$$

Prestazioni: open addressing

- *Random probing: ricerca con successo*
  - *Si ha che:* $\sum_{i=1}^{N} \frac{1}{i} = \log(n) + cost + O(n^{-1})$
  - *Quindi:*

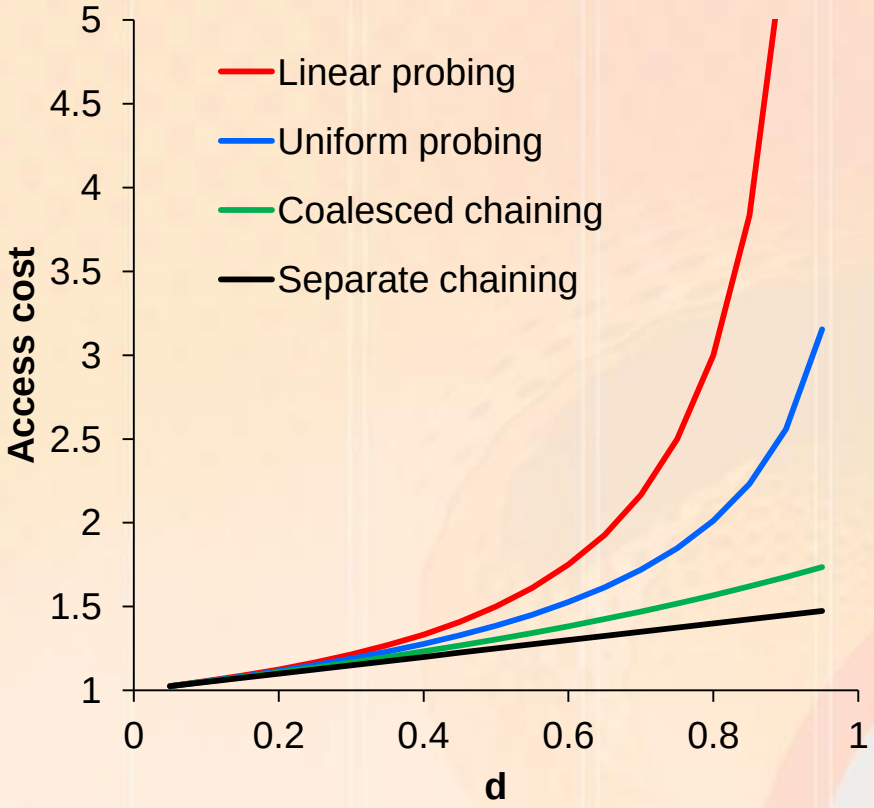
$$E \approx \frac{P}{N} \log\left(\frac{P}{P-N}\right) = \frac{P}{N} \log\left(\frac{1}{1-d}\right) = -\frac{P}{N} \log(1-d)$$

Prestazioni: open addressing

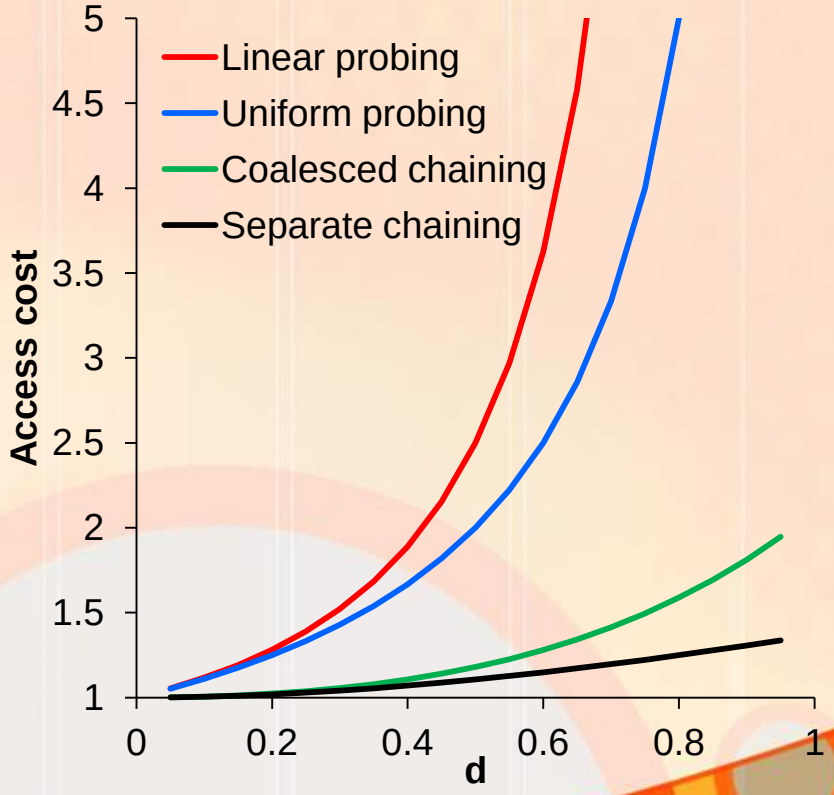
- *Linear probing*
  - *Ricerca con successo:* $E \approx \frac{1}{2}\left(1+\frac{1}{1-d}\right)$
  - *Ricerca con insuccesso:* $A \approx \frac{1}{2}\left(1+\frac{1}{(1-d)^2}\right)$
- *Se la capacità del bucket C aumenta, le prestazioni dei metodi ad indirizzamento aperto migliorano, in quanto:*
  - *A(C)=1+(A(C=1) –1)/C*
  - *E(C)=1+(E(C=1) –1)/C*

Confronto

- *Ricerca con successo*



- *Ricerca senza successo*



Problemi delle organizzazioni statiche

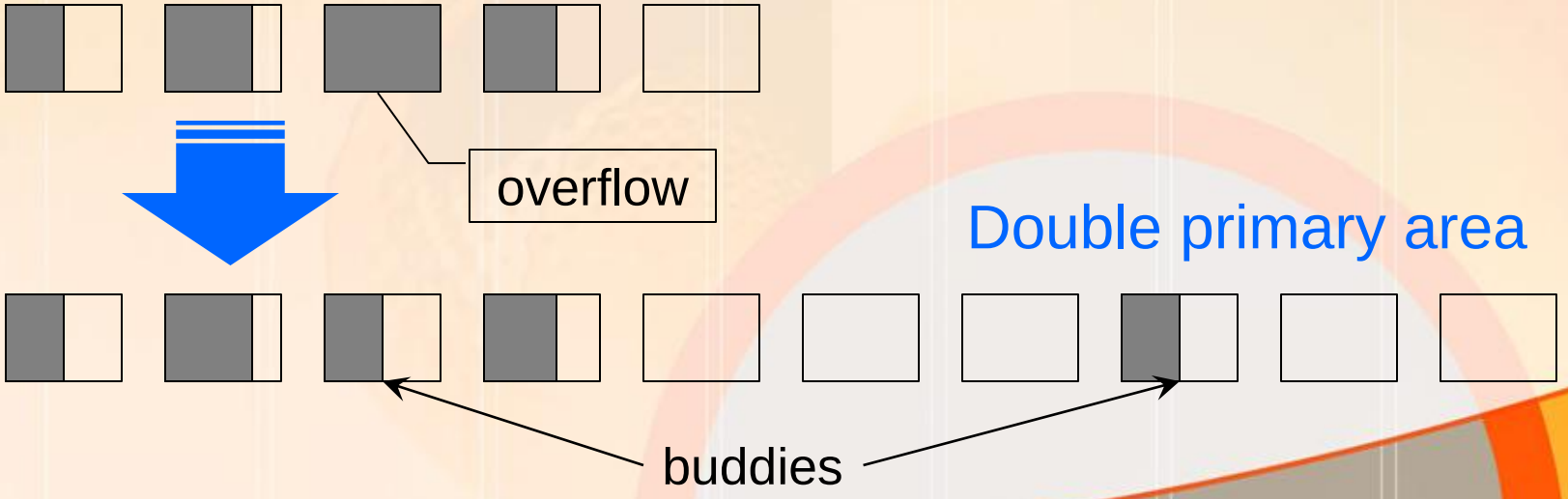
- *L'allocazione della memoria va effettuata al momento della progettazione iniziale*
  - *Se la stima è in eccesso, lo spazio è poco utilizzato*
  - *Se la stima è in difetto, si deve lavorare con un alto fattore di caricamento, quindi i costi aumentano*
- *Inoltre, se gli overflow sono in area primaria, c'è il vincolo* $d \leq 1$

Hashing dinamico

- *Queste tecniche adattano l'allocazione dell'area primaria al numero di tuple attuale*
- *Si dividono in*
  - *Con directory*
    - *Virtual hashing*
    - *Dynamic hashing*
    - *Extendible hashing*
  - *Senza directory*
    - *Linear hashing*
    - *Spiral hashing*

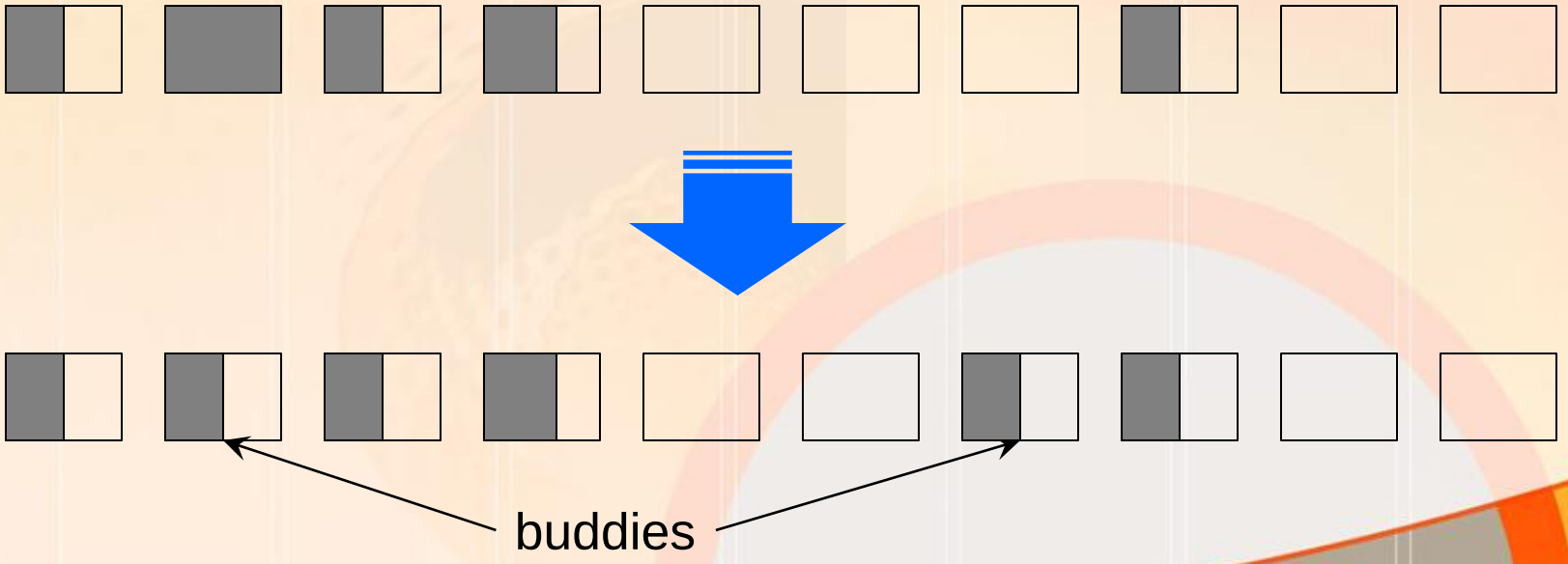
Virtual hashing

- *L'idea su cui si basa il Virtual hashing (Litwin '78) è raddoppiare l'area primaria quando si verifica un overflow in un bucket*
  - *Si ridistribuiscono i record tra il bucket saturo ed il suo "buddy", facendo uso di una nuova funzione hash*
  - *In pratica si esegue lo "split" del bucket saturo*



Virtual hashing: uso dei buddy

- *Se, successivamente, un altro bucket nell'area primaria originale va in overflow, e il suo buddy non è ancora in uso, si ridistribuiscono i suoi record tra il bucket e il suo buddy*



Virtual hashing: directory

- *Poiché, a un certo istante, solo alcuni buddy sono effettivamente in uso, è necessario fare uso di una struttura ausiliaria che permetta di determinare se occorre utilizzare la vecchia funzione hash o la nuova*
- *Sostanzialmente, si usa un vettore V di bit in cui V[i]=1 se e solo se il corrispondente bucket è in uso*

Virtual hashing: inizializzazione area primaria

- *Vengono allocati P_0 bucket di capacità C*
- *Si usa una funzione hash H_0 a valori in $[0, P_0 - 1]$*
- *Si pone $l=0$ (numero di raddoppi eseguiti)*
- *Si crea un vettore binario V, di dimensione pari a P_0, inizializzando tutti gli elementi al valore 1*
- *Dopo l raddoppi, l'area primaria contiene $P = 2^l P_0$ bucket, e il buddy del j-esimo bucket ($0 \leq j \leq 2^{l-1} P_0 - 1$) è il bucket di indirizzo $j + 2^{l-1} P_0$*

Virtual hashing: split di un bucket *j*

- Se $l=0$, oppure $l>0$ ma il buddy è già in uso o non esiste ($l>0$, ma $V[j+2^{l-1}P_0]=1$ o $j \geq 2^{l-1}P_0 - 1$)
  - l++, si raddoppia l'area primaria e il vettore V
  - I nuovi elementi di V valgono 0, eccetto $V[j+2^{l-1}P_0]$
  - Si usa la nuova funzione H_l a valori in $[0, 2^l P_0 - 1]$
  - Si ridistribuiscono le chiavi del bucket j usando la funzione H_l
- Se il buddy esiste e non è in uso ($V[j+2^{l-1}P_0]=0$)
  - $V[j+2^{l-1}P_0]=1$
  - Si ridistribuiscono le chiavi del bucket j usando la funzione H_l

Virtual hashing: esempio (i)

- P_0=7, C=3

112 1176	512 3270 841	723	6851	7830 1075 6647	2840 2665 2385	286

v
1
1
1
1
1
1
1

- *Si utilizza la famiglia di funzioni hash*
  $H_l(k) = k \% (2^l P_0)$
- *Supponiamo di inserire la chiave k=3820*
  $H_0(3820) = 5$
- *Il bucket 5 va in overflow*

Virtual hashing: esempio (ii)

- P_1=14

112 1176	512 3270 841	723	6851	7830 1075 6647	2665 2385	286
					3820 2840	

V
1
1
1
1
1
1
1
0
0
0
0
0
1
0

- *Si raddoppia l'area primaria e il vettore V e si ridistribuiscono le chiavi tra il bucket 5 e il suo buddy 12, facendo uso della funzione hash* $H_1(k) = k \% 14$

Virtual hashing: esempio (iii)

- $P_1=14$

112 1176	512 3270 841	723	6851	7830	2665 2385	286
				1075 6647 3343	3820 2840	

V
1
1
1
1
1
1
1
0
0
0
0
1
1
0

- *Se ora si deve inserire la chiave 3343 si ha $H_1(3343)=11$*
- *Poiché V[11]=0, si deve applicare $H_0(3343)=4$*
- *Il bucket 4 è saturo e quindi si splittare*
  - *in questo caso senza raddoppiare l'area primaria*
  - *V[11]=1 e si ridistribiscono le chiavi*
- *Se una nuova chiave vale 5485?*

Virtual hashing: ricerca (e inserimento)

- *Per cercare un valore di chiave è necessario sapere con quale funzione hash è stato allocato*
  - *Il vettore V è sufficiente allo scopo.*
- *Fornisce l'indirizzo del bucket in cui potrebbe trovarsi la chiave cercata (o che si vuole inserire)*

Address(k, l)

***Input**: Chiave k, livello l*

***Output**: Indirizzo del bucket a livello l in cui trovare k*

***if** (l<0) la chiave non esiste*

***else if** $V[H_l(k)] = 1$ **return** $H_l(k)$*

***else return** Address(k, l-1)*

Virtual hashing: funzioni hash

- *Il virtual hashing richiede una serie di funzioni hash* $H_0, H_1, \ldots, H_l, \ldots$ *che soddisfi le seguenti condizioni:*
  - ***Range condition**: La funzione* H_l *deve essere a valori in* $[0, 2^l P_0 - 1]$
  - ***Split condition**: Per ogni l>0, per ogni k, e per ogni valore di* $H_l(k)$*, deve valere la relazione:*

    $$H_l(k) = H_{l-1}(k) \quad \text{oppure} \quad H_l(k) = H_{l-1}(k) + 2^{l-1} P_0$$

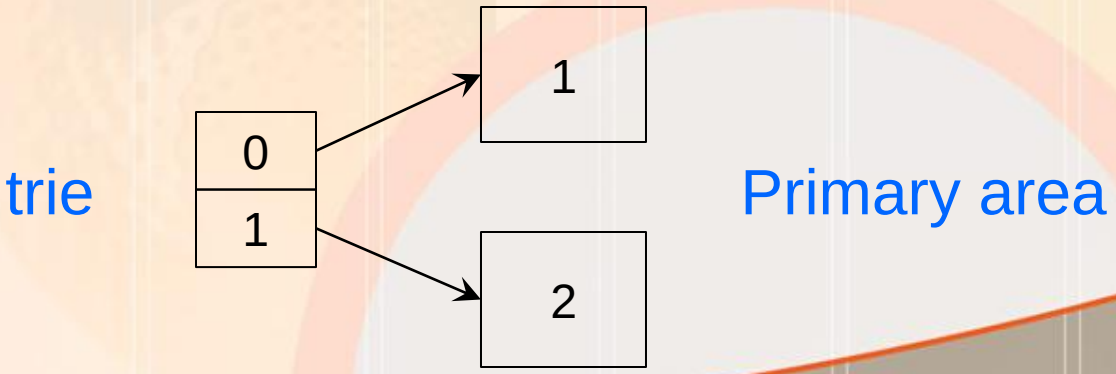
    ovvero lo split di un bucket deve lasciare una chiave nel bucket stesso, o allocarla nel buddy
- *La famiglia di funzioni* $H_l(k) = k \,\%\, (2^l P_0)$ *soddisfa entrambe le condizioni*

Dynamic hashing

- *Il dynamic hashing (Larson '78) evita di ricorrere a tecniche di raddoppio (che riducono l'utilizzazione e causano appesantimenti non trascurabili nel momento in cui l'area primaria viene raddoppiata), facendo uso di una struttura ausiliaria (directory) organizzata come un trie binario*
- *L'idea di base (comune all'extendible hashing) è quella di fare uso di una funzione hash che, dato il valore* k*, restituisce non un indirizzo, ma una pseudo-chiave binaria* $H(k) = b_0,b_1,b_2,\ldots$
- *La situazione ideale è quella in cui l'insieme di pseudo-chiavi da gestire è tale per cui* $\Pr\{b_i=1\}=1/2$*, ovvero vi è una ripartizione bilanciata per ogni posizione considerata*
- *Un semplice metodo per ottenere le pseudo-chiavi è utilizzare* k *come il seme di un generatore di numeri binari pseudo-casuali*

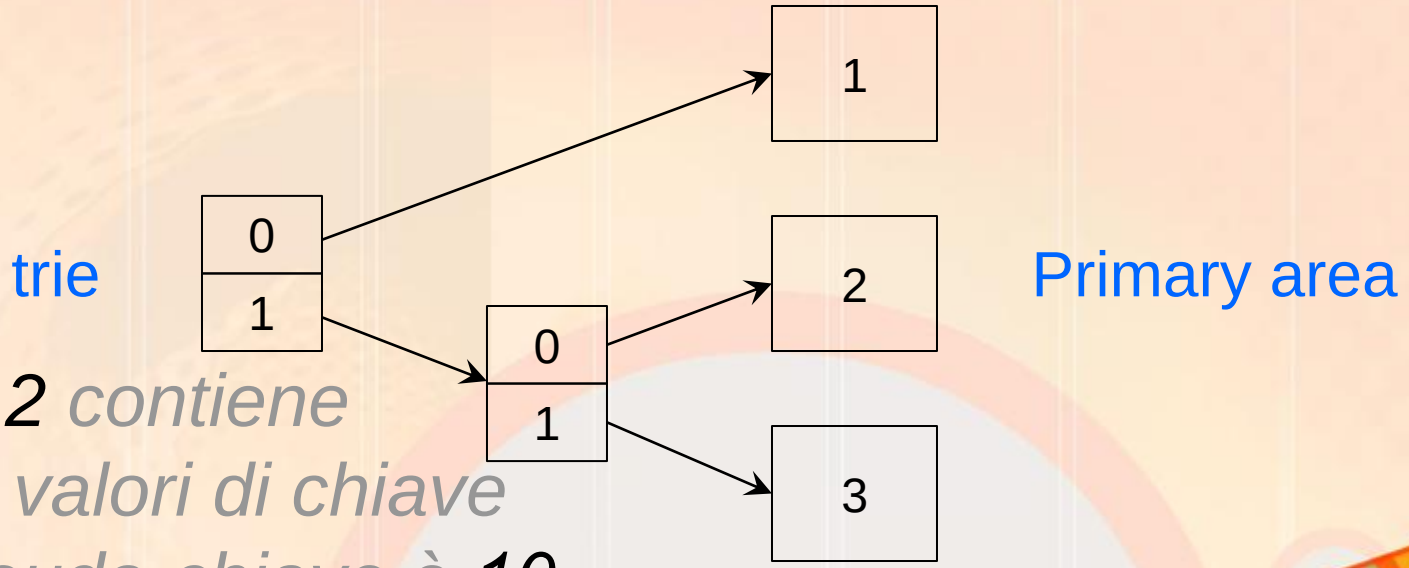
Dynamic hashing: uso del trie

- *Il trie serve a organizzare la ricerca, e i suoi nodi foglia indirizzano i bucket dell'area dati*
- *Per cercare (o inserire) un valore di chiave si segue, fino a una foglia, il cammino del trie corrispondente alla pseudo-chiave*
  - ***Esempio**: Il bucket 1 contiene tutti i valori di chiavi la cui pseudo-chiave è del tipo 0… e il bucket 2 quelli con pseudo-chiave 1…*



Dynamic hashing: overflow

- *L'espansione dell'area primaria avviene aggiungendo un bucket alla volta, ridistribuendo i record tra il bucket saturo e il suo buddy, e aggiungendo un nodo al trie*
  - ***Esempio**: se si deve eseguire lo split del bucket 2*



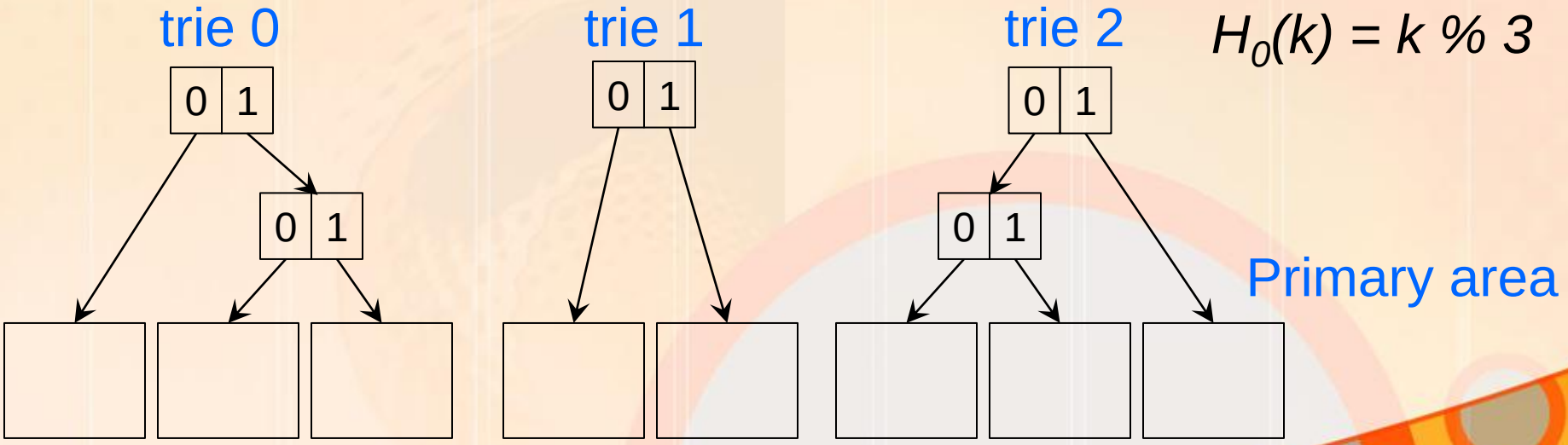
  - *Il bucket 2 contiene ora tutti i valori di chiave la cui pseudo-chiave è 10…, e il bucket 3 quelli con 11…*

Dynamic hashing: prestazioni

- *Se il trie è in memoria centrale basta un singolo accesso per recuperare un record*
- *Se il trie non è in memoria centrale, le prestazioni dipendono dal bilanciamento, in termini di numero di nodi indice da reperire*
- *Le prestazioni nel caso peggiore non sono buone*
  - *A seconda dell'insieme di pseudo-chiavi, l'inserimento di un nuovo record può comportare più di uno split.*
- *Se dopo una cancellazione in un bucket, il numero di record contenuti nel bucket e nel suo buddy diventa minore o uguale alla capacità C, i bucket vengono fusi, e si elimina una foglia dal trie*
- *L'utilizzazione media è di circa il 70%*

Dynamic hashing: variante

- *Si allocano inizialmente P bucket e si usa una qualsiasi funzione hash statica* H_0
- *A seguito di overflow, si genereranno P trie, le cui radici sono indirizzate da* H_0

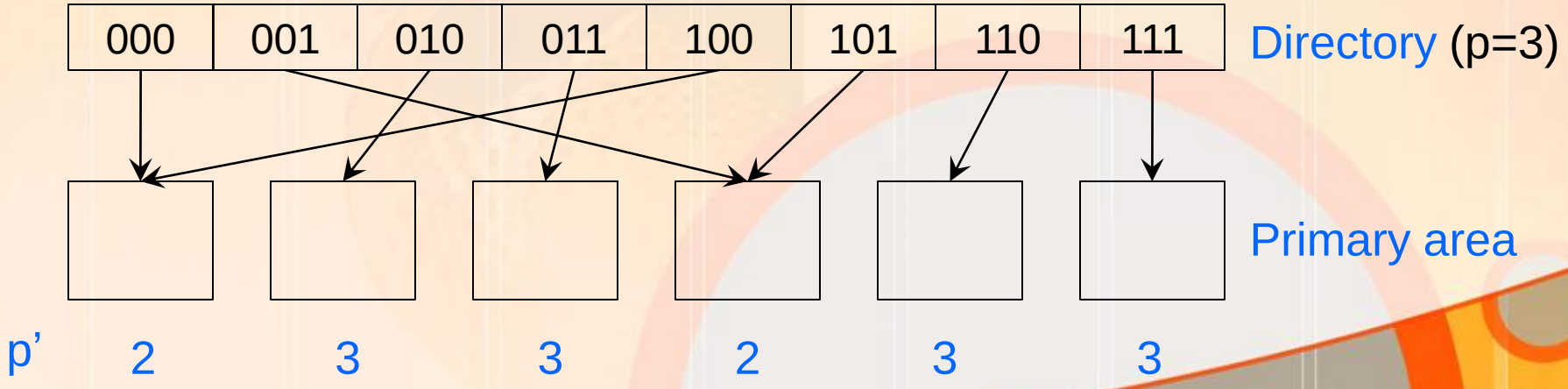


Extendible hashing

- *È molto simile al dynamic hashing, da cui si differenzia per la gestione della directory (Fagin, et al. ’79)*
- *Garantisce non più di due accessi I/O*
- *La directory consiste di 2^p celle con indirizzi $[0,2^p-1]$*
  - *$p \geq 0$ è detta profondità della directory*
- *Una funzione hash associa a ogni chiave una pseudo-chiave binaria, $H(k)=\ldots b_2,b_1,b_0$ di cui si considerano i p bit meno significativi per accedere direttamente a una delle 2^p celle della directory, ognuna contenente un puntatore a un bucket*

Extendible hashing: profondità del bucket

- *Ogni bucket ha una profondità locale p'≤p (valore mantenuto nel bucket), che indica il numero effettivo di bit usati per allocare le chiavi nel bucket stesso*
- ***Esempio**: Il bucket che contiene la chiave 258 (…001) ha p'=2, quindi contiene sia chiavi con pseudo-chiave …001 che …101*



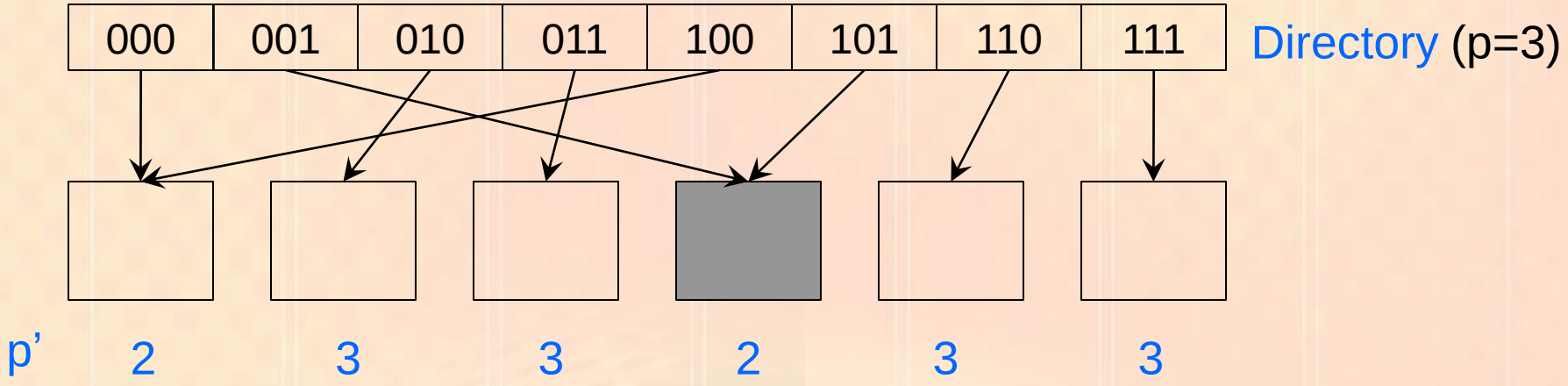
Extendible hashing: split di un bucket

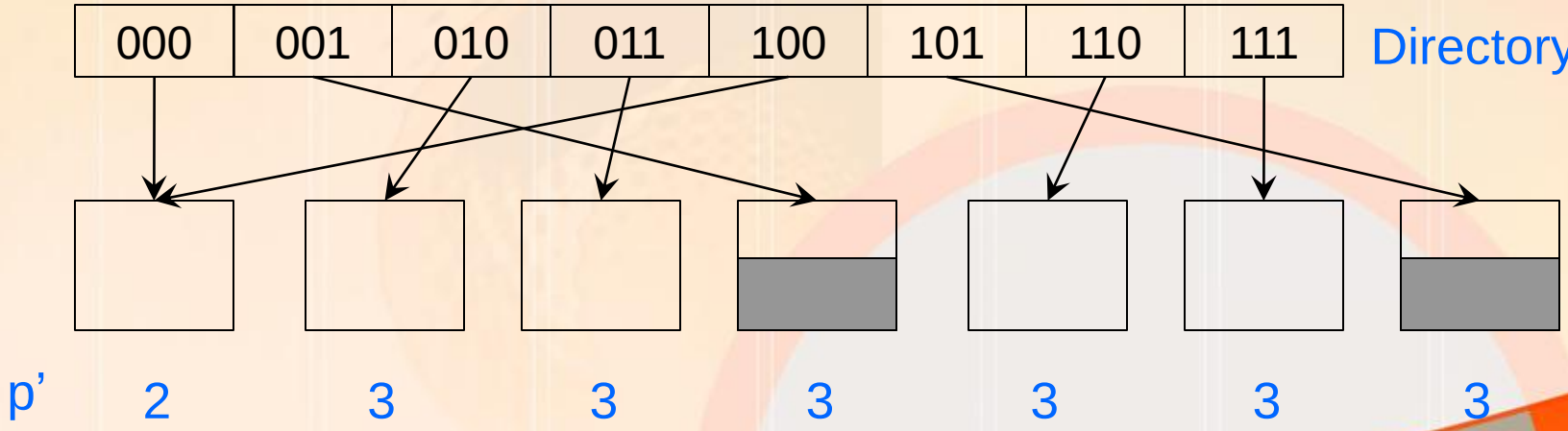
- *Inizialmente si ha un solo bucket con* $p' = 0$ *e* $p = 0$
- *Se si deve eseguire lo split di un bucket a profondità locale* p'*, si presentano 2 casi*
  - $p' < p$
  - $p' = p$

Extendible hashing: split di un bucket a p'<p

- *Si alloca un nuovo bucket e si distribuiscono le chiavi tra i due bucket facendo uso del (p'+1)-esimo bit delle pseudo-chiavi*
  - *Per i due bucket si pone a p'+1 il valore della profondità locale*
- *Poiché p' < p, esiste almeno una cella che può indirizzare il nuovo bucket*
  - *Si modifica pertanto il puntatore della/e cella/e*

Esempio di split di un bucket a p'<p

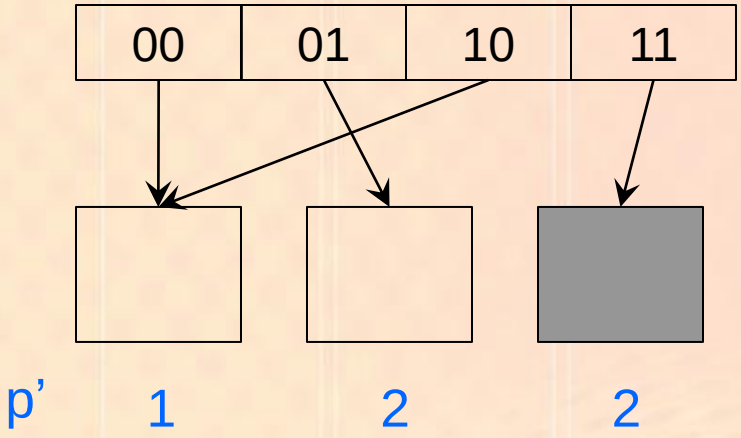


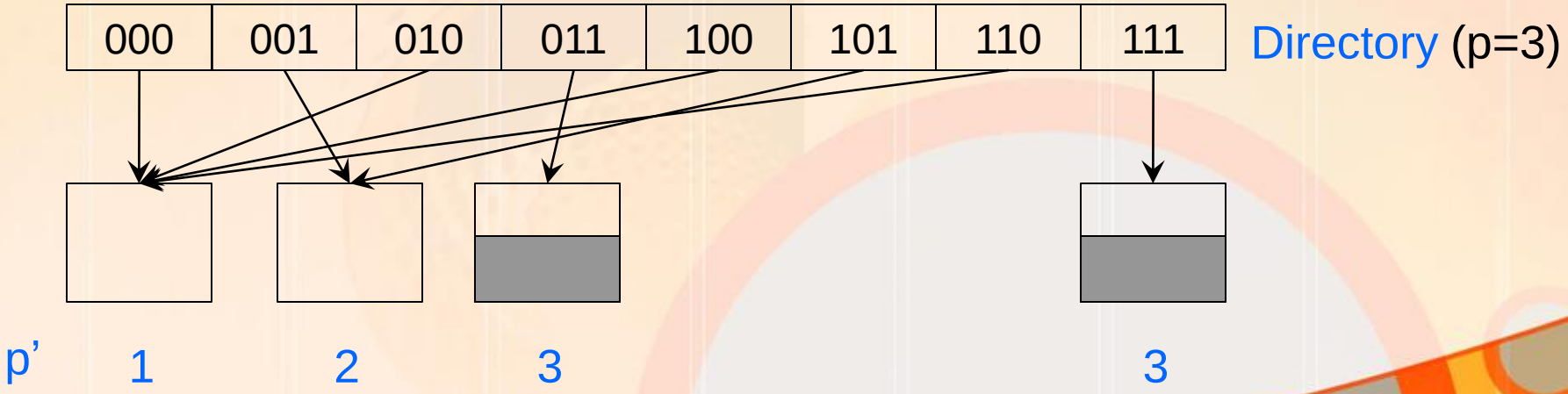


Extendible hashing: split di un bucket a p'=p

- *Poiché p' = p, non esiste alcuna cella che possa indirizzare il nuovo bucket eventualmente ottenuto dallo split*
- *Si raddoppia il direttorio, e si incrementa p di 1*
- *Si copiano i valori dei puntatori nelle nuove celle corrispondenti*
- *Si esegue lo split come nel caso p' < p*

Esempio di split di un bucket a p'<p

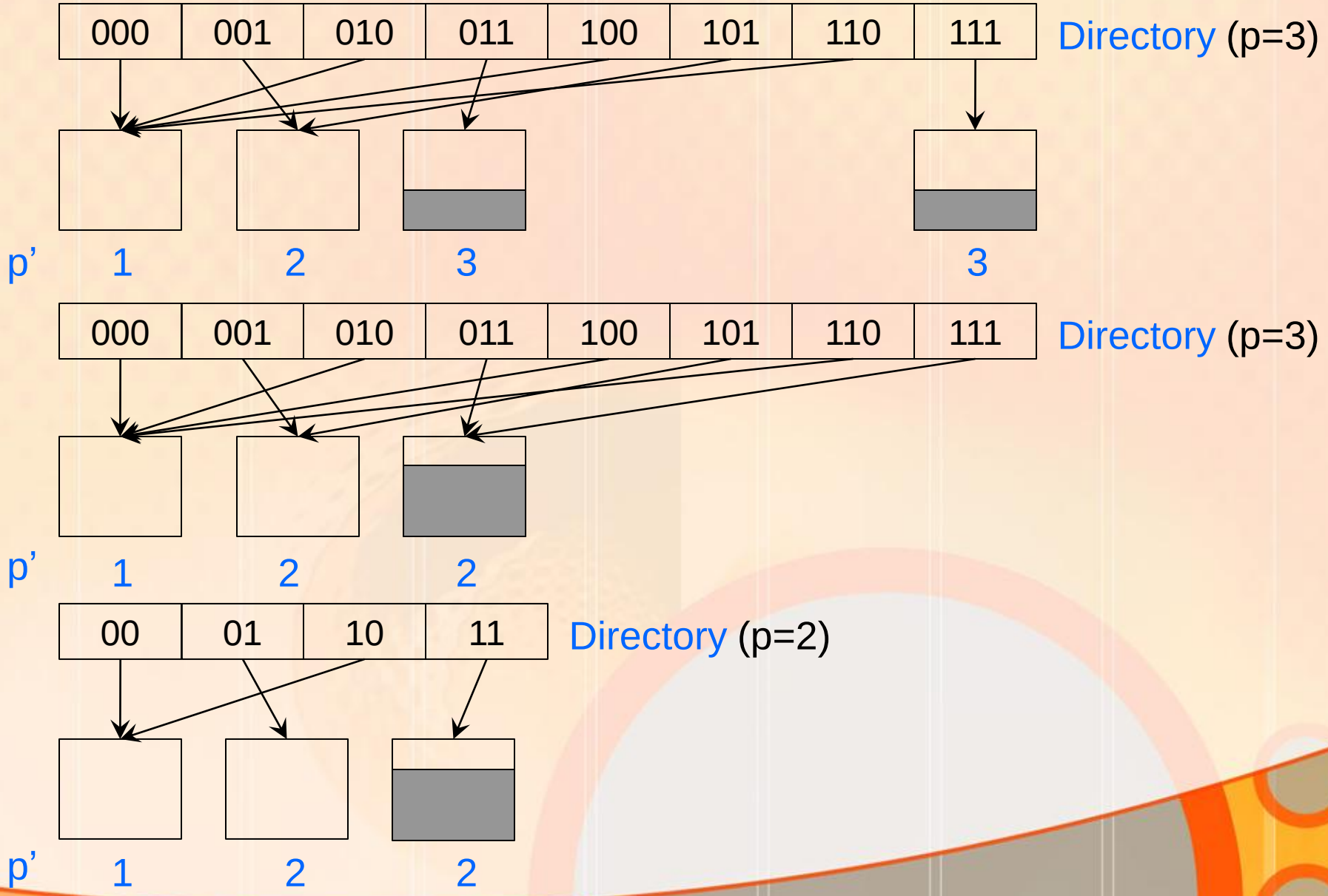




Extendible hashing: cancellazione

- *Se si cancella un record in un bucket a profondità p', e il numero di record contenuti nel bucket e nel suo buddy diventa minore o uguale alla capacità C, i bucket vengono fusi*
  - *Per il bucket risultante la profondità locale vale $p'-1$*
- *Se si fondono gli unici due bucket a profondità $p' = p$, è possibile contrarre la directory, dimezzandola*
  - *Poiché verificare che non esistono più bucket a profondità p richiede, nel caso peggiore, di esaminare metà dei bucket, è conveniente fare uso di una tabella delle profondità locali che, per ogni valore $p' \leq p$, mantiene il numero, $P(p')$, di bucket a profondità p'*

Esempio di contrazione della directory



Extendible hashing: considerazioni

- *Ogni raddoppio coinvolge tutta la directory*
  - *Problemi in caso di concorrenza*
- *Una soluzione prevede l'uso di una directory a più livelli*
  - *L'indice non è necessariamente binario*

Linear hashing

- *Tecnica ad espansione lineare:*

  "Non si esegue lo split del bucket in overflow, ma di un altro bucket, scelto secondo un criterio specifico"

  - *Non è necessaria una directory*
  - *Occorre gestire l'overflow (in aria primaria o separata)*
  - *L'area primaria cresce "linearmente"*
- *Nel caso del linear hashing, il bucket da dividere è quello successivo all'ultimo bucket diviso*

Linear hashing: gestione dell'area primaria

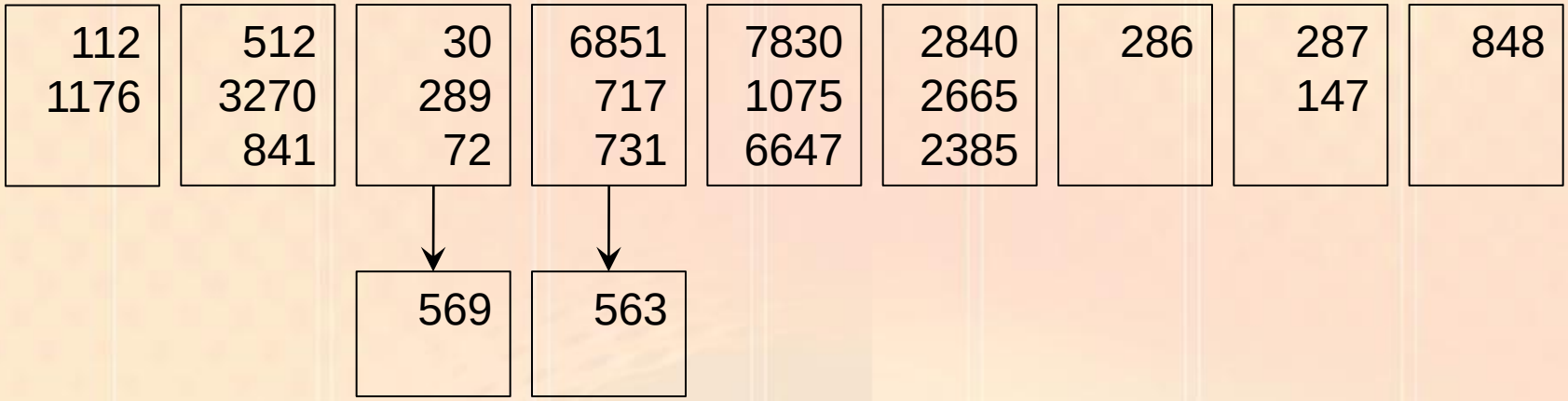
- *Inizialmente si allocano P_0 bucket e si usa la funzione hash $H_0(k) = k \% P_0$*
- *Si mantiene un puntatore (split pointer, SP) al prossimo bucket che deve essere suddiviso*
  - *Inizialmente $SP = 0$*
- *Se si verifica un overflow*
  - *Si aggiunge in coda un bucket di indirizzo $P_0 + SP$*
  - *Si riallocano i record del bucket SP (inclusi quelli eventualmente presenti in area di overflow) facendo uso della nuova funzione hash*
    $H_1(k) = k \% (2P_0)$
  - *si incrementa SP di 1*

Linear hashing: raddoppio dell'area primaria

- *Dopo P_0 overflow si è operata un'espansione completa dell'area primaria, in quanto il numero di bucket è ora pari a $2\ P_0$*
- *Ci si predispone per una nuova espansione ponendo $SP = 0$, $H_0(k) = H_1(k)$, e $H_1(k) = k \% (2^2\ P_0)$*
- *Durante la j-esima espansione si usano le funzioni hash $H_0(k) = k \%(2^{j-1}\ P0)$ e $H_1(k) = k \% (2^j\ P_0)$*
- *L'indirizzo dell'home bucket di una chiave è $H_0(k)$ se $H_0(k) \geq SP$, o $H_1(k)$ altrimenti*

Linear hashing: esempio (i)

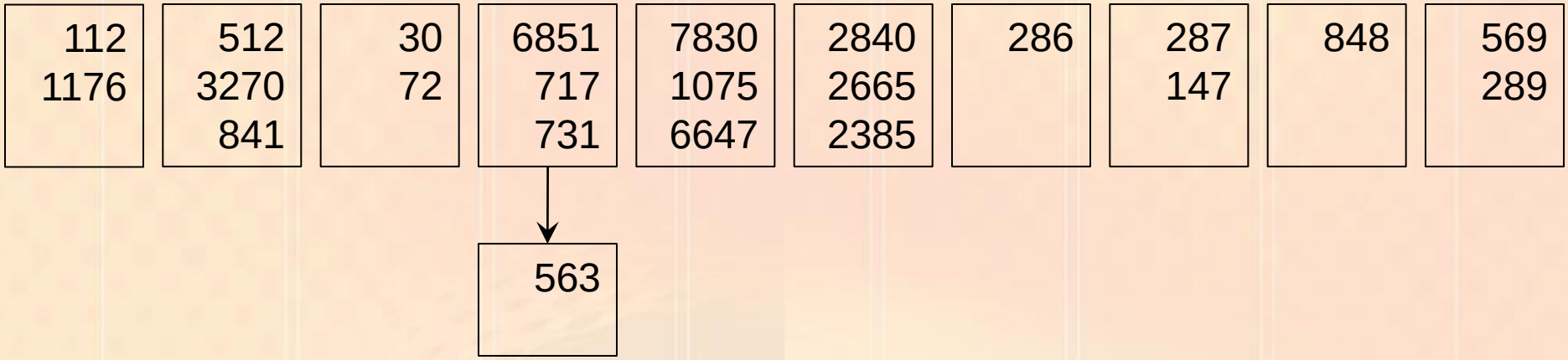
- P_0=7, C=3, C_{ov}=2, SP=2



- *L'overflow dei bucket 2 e 3 ha causato lo split dei bucket 0 e 1*
- *Inserendo la chiave 3820 ($H_0(3820) = 5$) si genera un overflow nel bucket 5*
  - *Si divide il bucket 2*

Linear hashing: esempio (ii)

- P_0=7, C=3, C_{ov}=2, SP=3



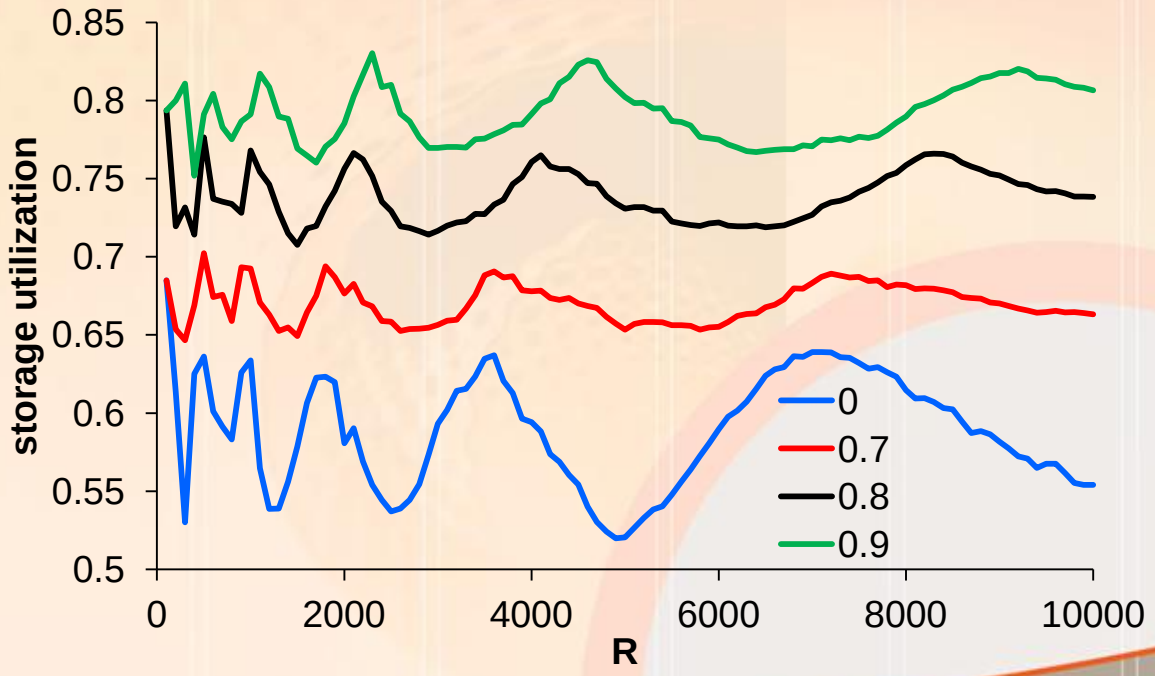
- *L'allocazione del bucket 9 ha causato l'eliminazione di un bucket di overflow*
  - *Il prossimo overflow causerà lo split del bucket 3*
- *Dopo 7 split l'area primaria è raddoppiata e si riparte da SP=0*

Linear hashing: pregi e difetti

+ *L'assenza di un direttorio e la politica di gestione degli split rendono semplice la realizzazione della struttura*
+ *La gestione dell'area primaria (espansione e contrazione) è immediata, in quanto i bucket vengono sempre aggiunti (e rimossi) in coda*
– *L'utilizzazione della memoria allocata è decisamente bassa (variabile tra 0.5 e 0.7)*
– *La gestione dell'area di overflow presenta problemi simili a quelli di un'area primaria statica*
– *Le catene di overflow relative ai bucket di indirizzo maggiore, non ancora suddivisi, possono diventare molto lunghe*

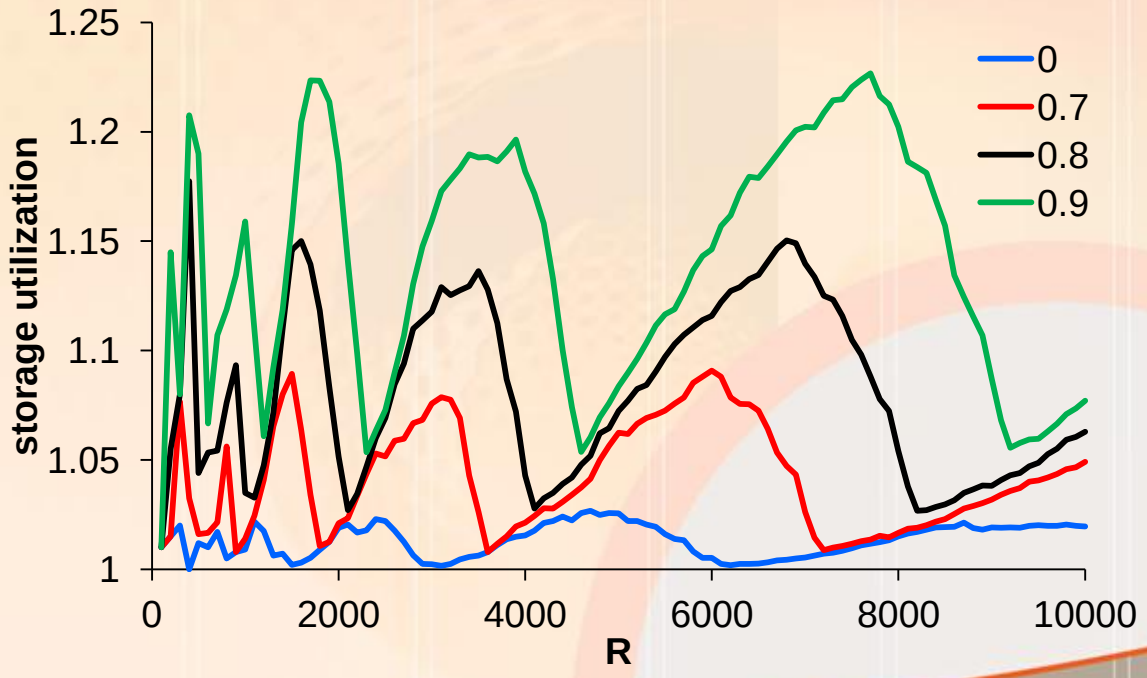
Linear hashing: utilizzo della memoria

- *Una variante, che migliora l'utilizzazione della memoria, prevede di non effettuare lo split di un bucket se questo non ha raggiunto un livello di utilizzazione minima u_{min}*



Linear hashing: prestazioni

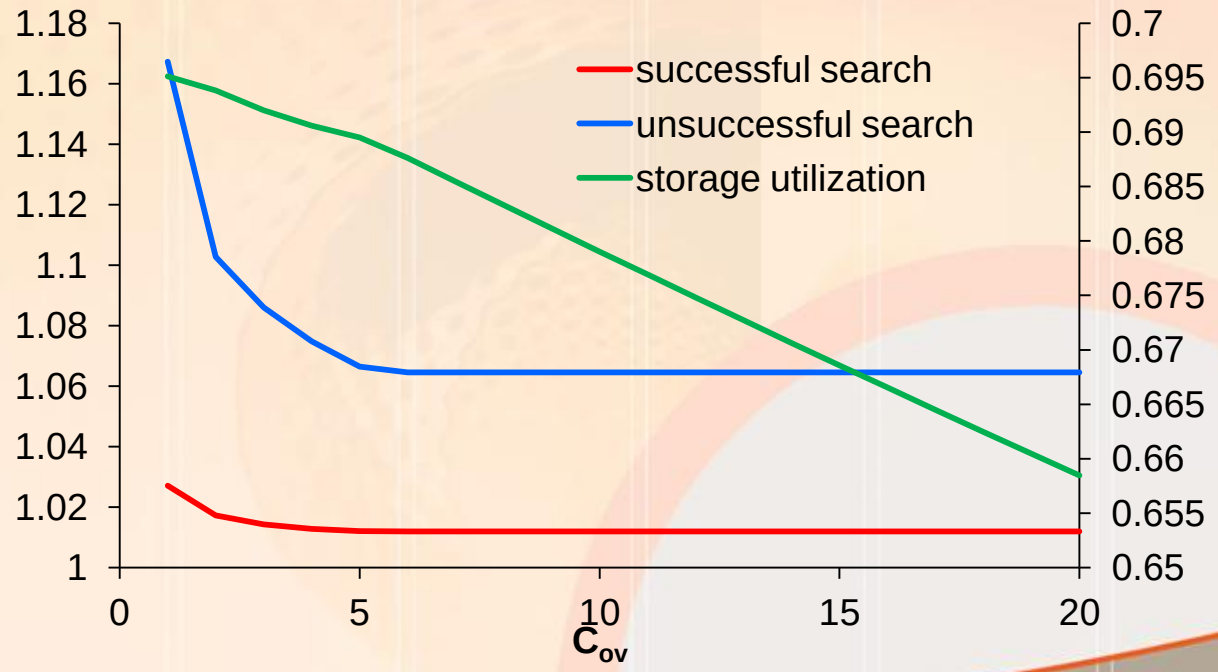
- *All'aumentare dell'utilizzazione di memoria aumentano anche i costi di ricerca, in quanto aumentano i record in overflow*
- *Costi di ricerca con successo*



I minimi dei costi di accesso corrispondono ai massimi dell'utilizzazione di memoria

Linear hashing: capacità di overflow

- *Aumentare C_{ov} riduce la lunghezza delle catene di overflow*
  - *Si riduce il costo della ricerca*
- *Oltre un certo limite si ha solo spreco di memoria*



Linear hashing ricorsivo

- *La caratteristica principale del Linear hashing ricorsivo (Ramamohanarao & Sacks-Davis '84) riguarda la gestione dell'area di overflow, che è organizzata dinamicamente facendo uso del Linear hashing stesso*
- *Si creano vari livelli di file hash dinamici (in media non più di 3), con il file al livello h (h=0 area primaria) che memorizza i suoi overflow nel file a livello h+1*
- *Al livello h si mantiene lo split pointer* SP_h

Linear hashing ricorsivo: operazioni

- *Inserimento:*
  - *Si calcola l'indirizzo $H^0(k)$*
  - *Se si verifica un overflow si passa al livello 1 con la funzione $H^1(k)$*
  - *Se va in overflow anche il bucket a livello L, si aggiunge un livello*
- *Split:*
  - *Quando il j-esimo bucket del livello h viene diviso, i record da ripartire sono quelli del bucket stesso e quelli in overflow, che sono in bucket dei livelli (h+1), …, L*

Linear hashing ricorsivo: ricerca

- *La ricerca di una chiave può richiedere un numero di accessi pari al numero di livelli*

```
h = 0;
while (h ≤ L)
    if (H^h_0(k) < SP_h) Address = H^h_1(k);
    else Address = H^h_0(k);
    if EXISTS(k, Address, h) return true;
    else  if FULL(Address, h) h++;
          else return false;
```

Linear hashing ricorsivo: indirizzamento

- *La memorizzazione degli overflow del livello h viene eseguita utilizzando come nuova "chiave" l'indirizzo del bucket del livello h stesso*
- *Il motivo per cui nei file di overflow l'allocazione non viene eseguita utilizzando il valore di chiave, k, dipende dal fatto che, in fase di split di un bucket, non si avrebbero informazioni su dove sono stati memorizzati i suoi overflow*

Linear hashing ricorsivo: esempio

- P_0=5, P_1=3, P_2=2, C=2

10 90	21 111	312 42	48 93	39 114	**25** **75**	46 56	127 247	*h=0* SP_0=3
20 76	101 77	32 **85**	103	29 34				*h=1* SP_1=2
54 100	97	**105**						*h=2* SP_2=1

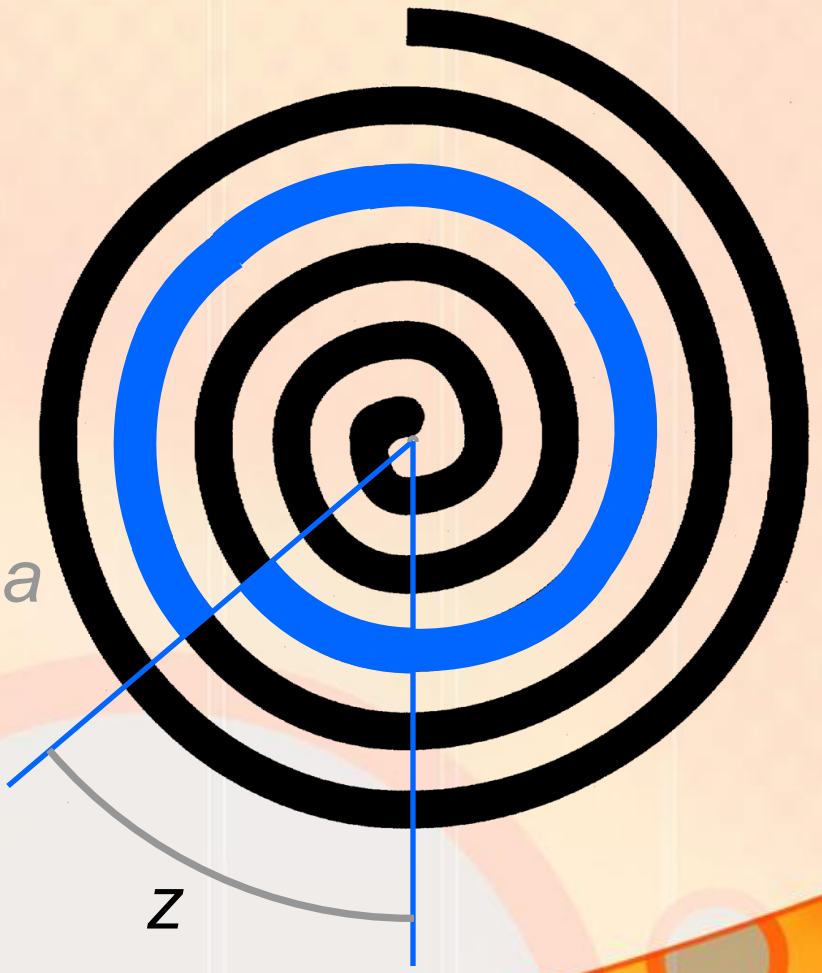
- *Gli overflow del bucket 5 del livello 0 sono nel bucket 5%3=2 del livello 1 (assieme a quelli del bucket 2)*
- *Gli overflow di tale bucket si trovano a livello 2 nel bucket 2%4=2*

Spiral hashing

- *Con il Linear hashing c'è una maggiore probabilità di avere overflow dai blocchi non ancora suddivisi durante l'espansione corrente*
  - *Infatti, l'uso di una funzione hash uniforme ha come conseguenza che ogni valore di $H_0(k)$ è ugualmente probabile, ma i bucket per cui si ha $H_0(k)<SP$ sono già stati suddivisi*
- *Lo spiral hashing (Martin '79) cerca di risolvere questo problema impiegando una funzione di tipo esponenziale, che consente di memorizzare i record più densamente nell'estremo iniziale dell'area primaria*
- *L'organizzazione deve il suo nome al fatto che lo spazio di memoria viene pensato come una spirale, invece che come una retta, e l'area primaria come una rivoluzione della spirale, univocamente definita da un angolo z*

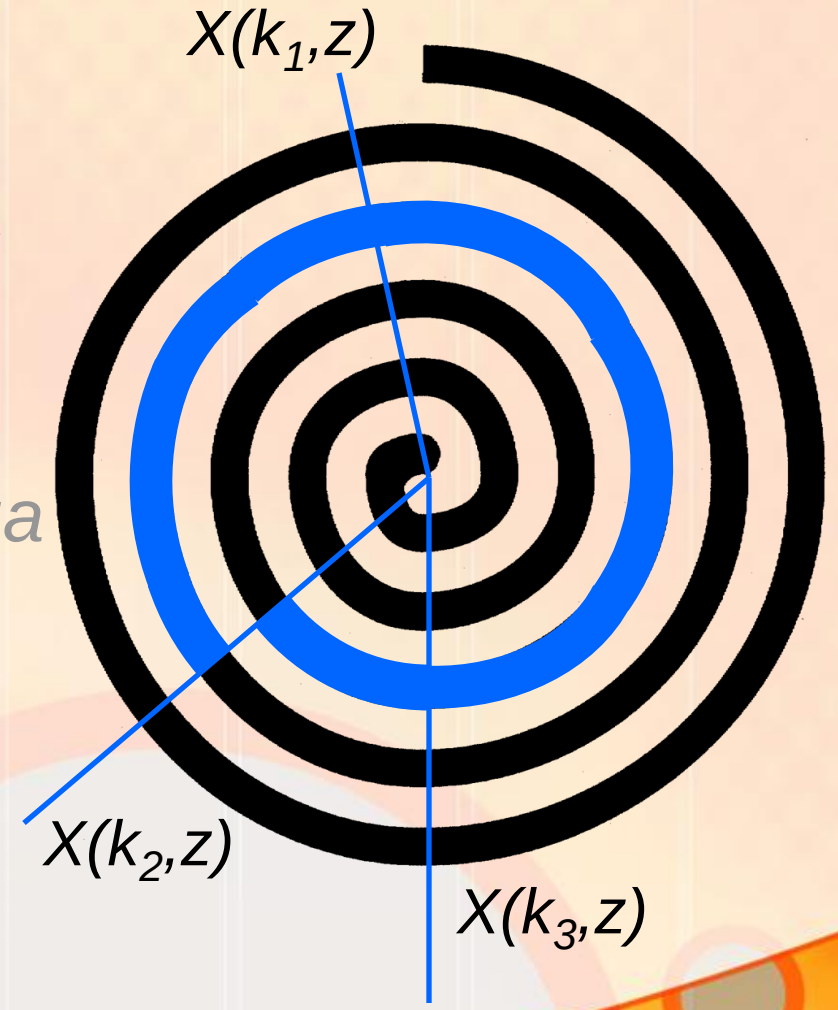
Spiral hashing: intuizione

- *L'area dati si espande con un fattore di crescita w*
  - *Maggiore è w, più rapidamente cresce l'area dati*
- *L'area dati giace lungo una rivoluzione della spirale*
- *Il numero delle pagine dell'area dati è* $P = \lfloor w^{z+1} \rfloor - \lfloor w^{z} \rfloor$
- *Il valore di z viene incrementato ad ogni espansione*
- *Inizialmente z=0 (P=1)*



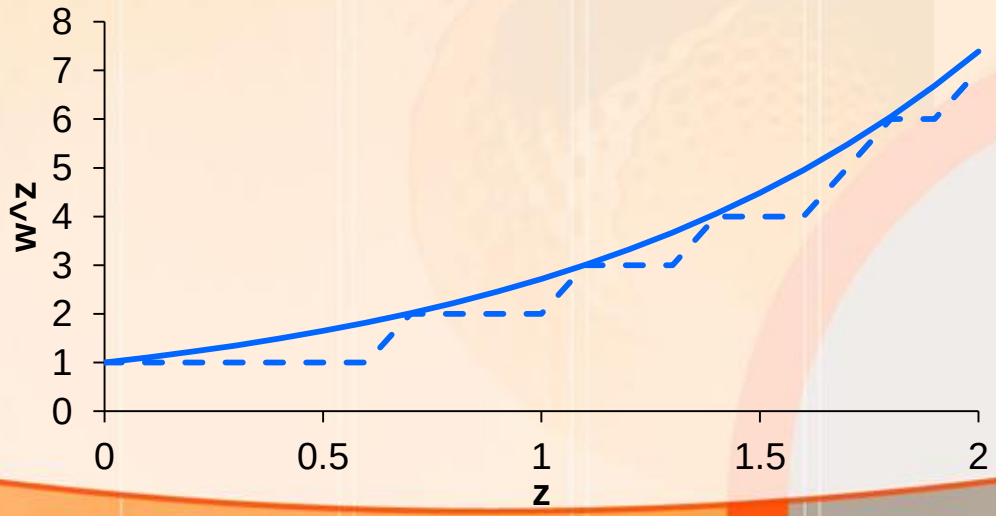
Spiral hashing: indirizzi logici (i)

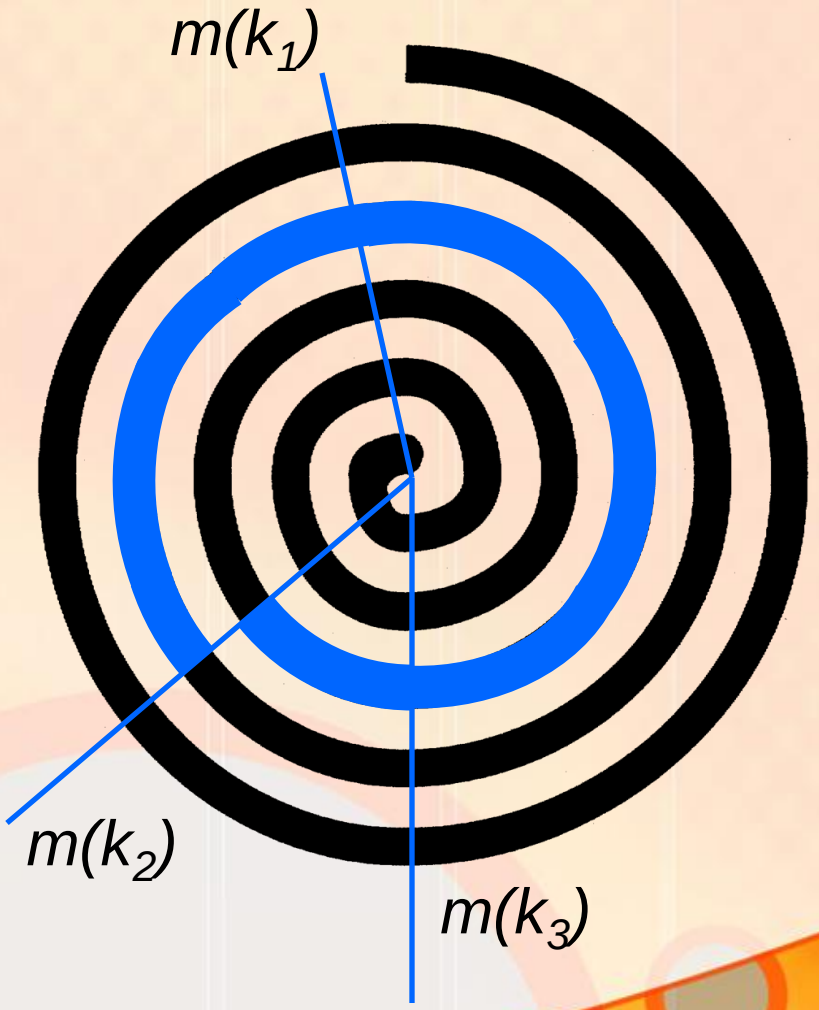
- *Sia* $H(k)$ *una funzione hash a valori in* $[0,1[$
- *La "direzione" di una chiave k è data dalla funzione*
  $X(k,z) = \lceil z - H(k) \rceil + H(k)$
  a valori in $[z,z+1[$ *e discontinua in* $z - \lfloor z \rfloor$



Spiral hashing: indirizzi logici (ii)

- *L'indirizzo logico di* k *è determinato dalla funzione esponenziale* $m=\lfloor w^{X(k,z)} \rfloor$
  - *Valore minimo* $\lfloor w^{z} \rfloor$
  - *Valore massimo* $\lfloor w^{z+1} \rfloor - 1$
- *L'indirizzo relativo è* $m_r=\lfloor w^{X(k,z)} \rfloor - \lfloor w^{z} \rfloor$



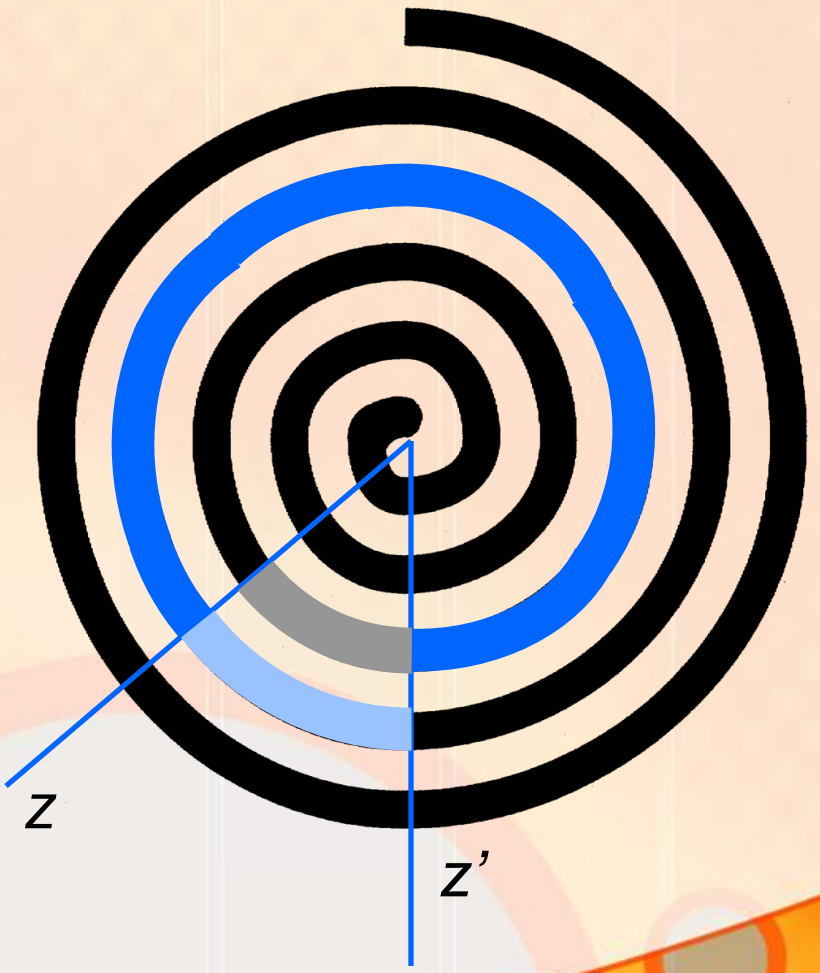


Spiral hashing: proprietà degli indirizzi

- *Se la funzione H(k) è uniforme, le chiavi vengono distribuite uniformemente in* [*z*,*z*+*1*[
- *L'andamento esponenziale fa sì che le pagine di indirizzo più basso ricevano più dati delle pagine di indirizzo più alto*
  - *Dipendenza inversa dalla pendenza della curva*
- *L'espansione dell'area dati avviene, come per il linear hashing, "rimuovendo" record dal bucket all'inizio del file ed allocandoli in un nuovo bucket creato alla fine del file*

Spiral hashing: split

- *Viene sempre espanso il bucket di indirizzo minore*
  - *In realtà si fa in modo che l'indirizzo del bucket in overflow non venga più generato*
- *I record del bucket* $\lfloor w^z \rfloor$ *vengono ridistribuiti nei nuovi bucket creati in fondo al file*
- *Il valore dell'angolo viene pertanto aumentato da* z *a* z'



Spiral hashing: nuovo valore di z

- z' *deve essere tale da garantire che* $\lfloor w^z \rfloor$ *non venga più generato:* $\lfloor w^{z'} \rfloor = \lfloor w^z \rfloor + 1$
- *Quindi:* $z' = \log_w(\lfloor w^z \rfloor + 1)$
- *Perciò il numero di bucket aggiunti vale:*

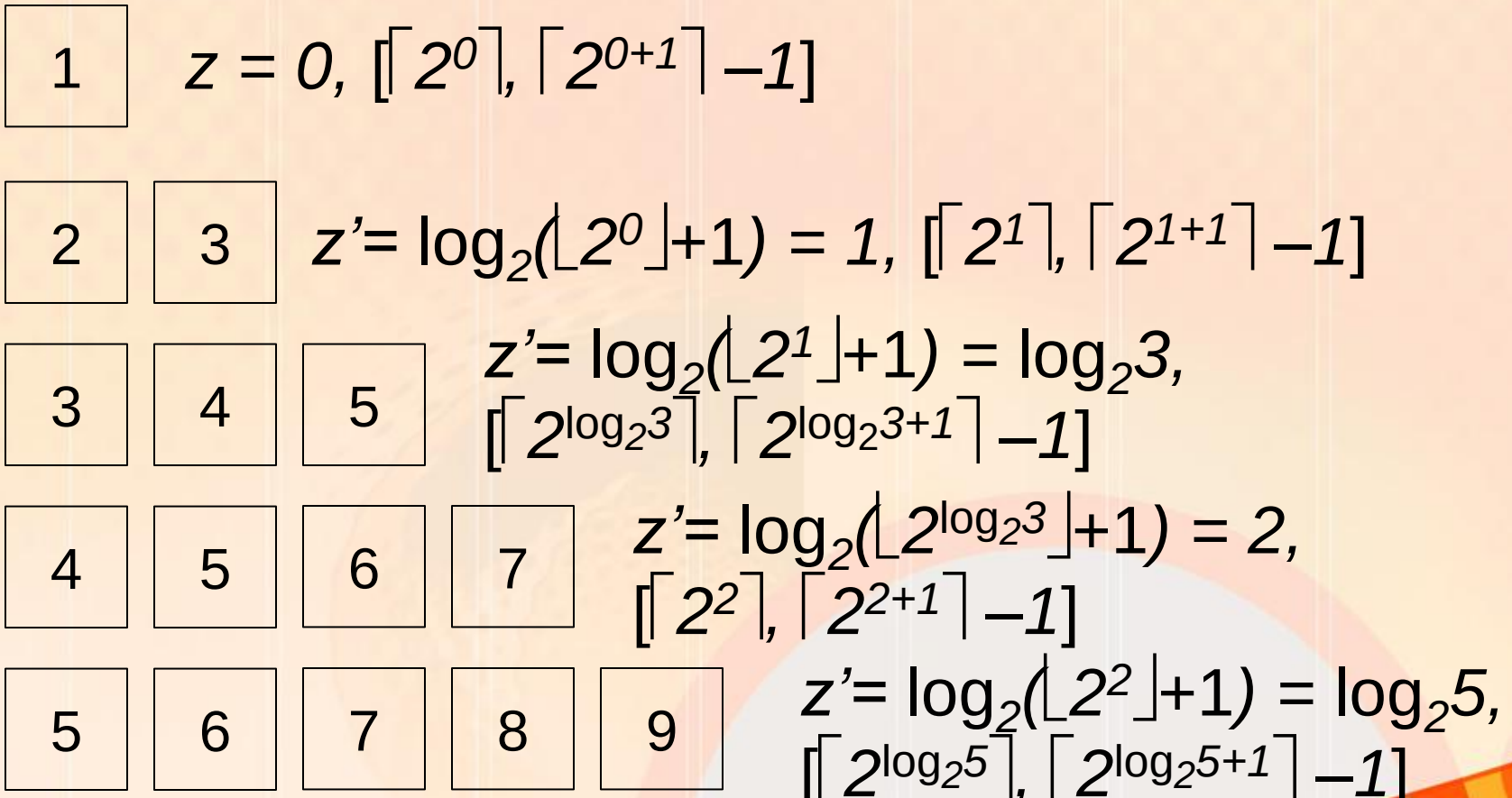
$$\lceil w^{z'+1} \rceil - \lceil w^{z'+1} \rceil = \lceil w(w^{z'}+1) \rceil - \lceil w \cdot w^z \rceil$$

cioè $\lceil w \rceil$ *o* $\lfloor w \rfloor$ *a seconda del valore di* z

- *Pertanto* $w - 1$ *determina la crescita dell'area primaria*

Spiral hashing: esempio

- *Con w=2 e z=0, otteniamo:*



Spiral hashing: espansione

- *La funzione* $X(k,z)$ *garantisce che gli indirizzi logici delle chiavi in bucket diversi dal primo non vengano modificate*
- $X(k,z)$ *e* $X(k,z')$ *infatti differiscono solo per quelle chiavi per cui* $H(k) \in [z - \lfloor z \rfloor, z' - \lfloor z' \rfloor[$, *ovvero tra i due punti di discontinuità*
- *Siccome* $z' = \log_w(\lfloor w^z \rfloor + 1)$, *tali chiavi sono solo quelle con indirizzo logico* $\lfloor w^z \rfloor$

Spiral hashing: indirizzi fisici

- *La cancellazione dei bucket in testa al file pone il problema del riutilizzo di tali pagine*
- *Perciò occorre "mappare" gli indirizzi logici m in indirizzi fisici ph(m)*
- *Lo schema di base è il seguente:*
  - *La numerazione parte da 0*
  - *Il primo bucket da aggiungere rimpiazza quello eliminato in testa*
  - *Gli altri bucket vengono aggiunti in coda*

Spiral hashing: esempio di allocazione

- *Con $w=3$ e $z=0$, otteniamo:*

1	2

$[\lceil 3^0 \rceil, \lceil 3^{0+1} \rceil - 1]$

3	2	4	5

$[\lceil 3^{\log_3 2} \rceil, \lceil 3^{\log_3 2+1} \rceil - 1]$

3	6	4	5	7	8

$[\lceil 3^{\log_3 3} \rceil, \lceil 3^{\log_3 3+1} \rceil - 1]$

9	6	4	5	7	8	10	11

$[\lceil 3^{\log_3 4} \rceil, \lceil 3^{\log_3 4+1} \rceil - 1]$

Spiral hashing: conversione indirizzi

- Siano $l = \lfloor (m-1)/w \rfloor$ e $h = \lfloor m/w \rfloor$, $ph(m)$ vale:
  - $ph(h)$ se $l<h$
  - $m - h - 1$, altrimenti
- Nell'esempio precedente per il bucket 6 si ha:
  - $l = \lfloor 5/3 \rfloor = 1$, $h = \lfloor 6/3 \rfloor = 2$
- Occorre quindi calcolare $ph(2)$ per cui:
  - $l = \lfloor 1/3 \rfloor = 0$, $h = \lfloor 2/3 \rfloor = 0$
- Quindi $ph(6)=ph(2)= 2 - 0 - 1 = 1$
- Per il bucket 7, invece: $l = 2$, $h = 2$, quindi $ph(7) = 7 - 2 - 1 = 4$

Spiral hashing: prestazioni (i)

- *Il vantaggio principale dello spiral hashing riguarda la netta riduzione dei fenomeni oscillatori presenti nel linear hashing*
- *w=2, C=10, C_{ov}=3*

Spiral hashing: prestazioni (ii)

- *Nel caso di split controllato, aumentare* ***w*** *provoca l'aumento dei costi di ricerca*
- *Con split non controllato questo non succede, ma si riduce l'utilizzo della memoria*
- $C=10$, $C_{ov}=3$, $R=15000$