Basi di dati / Complementi di Basi di Dati

Organizzazione Fisica dei Dati (parte III)

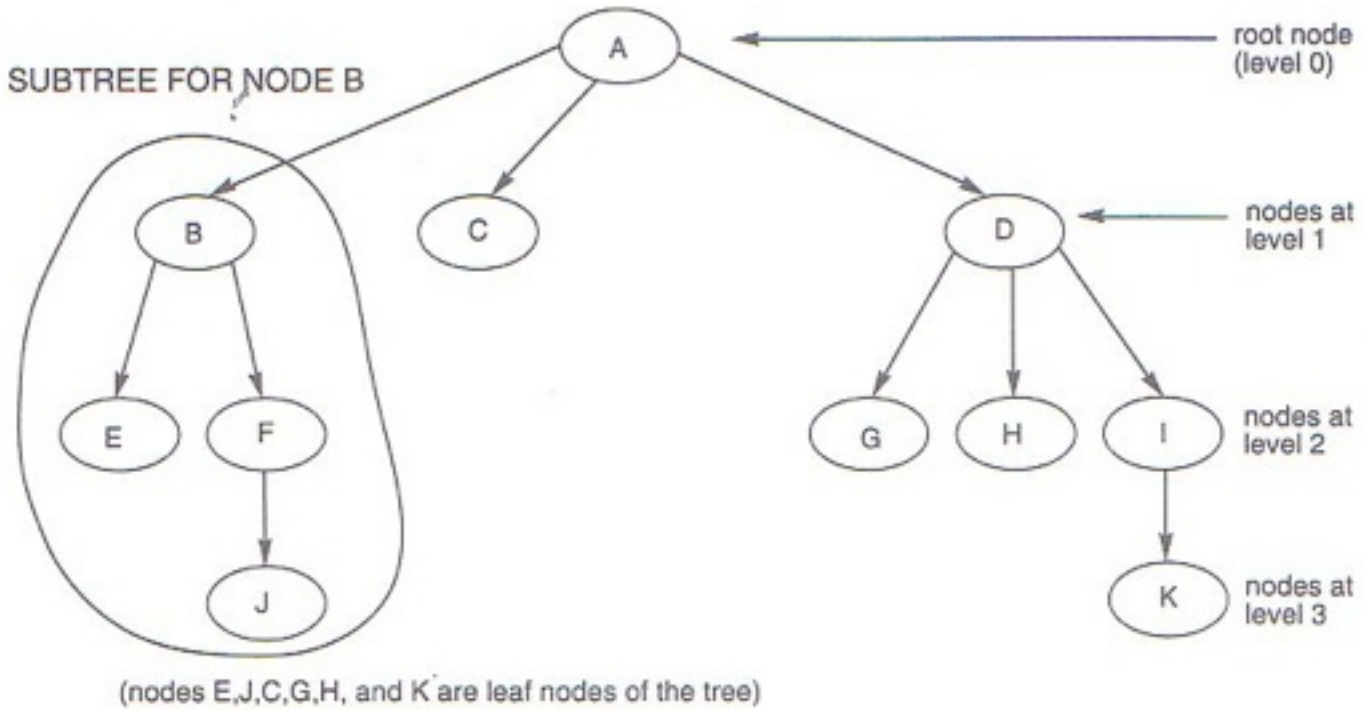
Indici Multilivello Dinamici

Angelo Montanari
Dipartimento di Matematica e Informatica
Università di Udine

Indici multilivello dinamici (B-alberi e B^+-alberi)

- Gli indici multilivello dinamici (B-alberi e B^+-alberi) sono casi speciali di strutture ad albero.
- Un albero è formato da nodi. Ogni nodo dell'albero, ad eccezione di un nodo speciale detto *radice*, ha un nodo padre e zero, uno o più nodi figli.
- Se un nodo non ha nessun nodo figlio è detto nodo foglia (*leaf node*). Ogni nodo che non sia un nodo foglia è detto nodo interno (*internal node*).
- Un sottoalbero di un nodo comprende quel nodo e tutti i suoi *discendenti* (nodi figli, figli dei nodi figli, etc.).

Un esempio


SUBTREE FOR NODE B
A
B
C
D
E
F
G
H
I
J
K
root node
(level 0)
nodes at
level 1
nodes at
level 2
nodes at
level 3
(nodes E,J,C,G,H, and K are leaf nodes of the tree)

Alberi di ricerca - 1

Un *albero di ricerca* è un tipo speciale di albero utilizzato per guidare la ricerca di record, dato il valore di uno dei loro campi.

- Un indice multilivello (statico) può essere visto come una variante degli alberi di ricerca.
- Ogni nodo dell'indice multilivello può avere *fo* puntatori e *fo* chiavi, con *fo* = *fan-out* dell'indice. In ogni nodo, il valore del campo indice consente di passare al nodo successivo finché non si raggiunge il blocco di dati che può contenere il record cercato. Ad ogni livello, si restringe la ricerca ad un sottoalbero radicato nel nodo corrente (ignorando tutti gli altri nodi dell'albero).

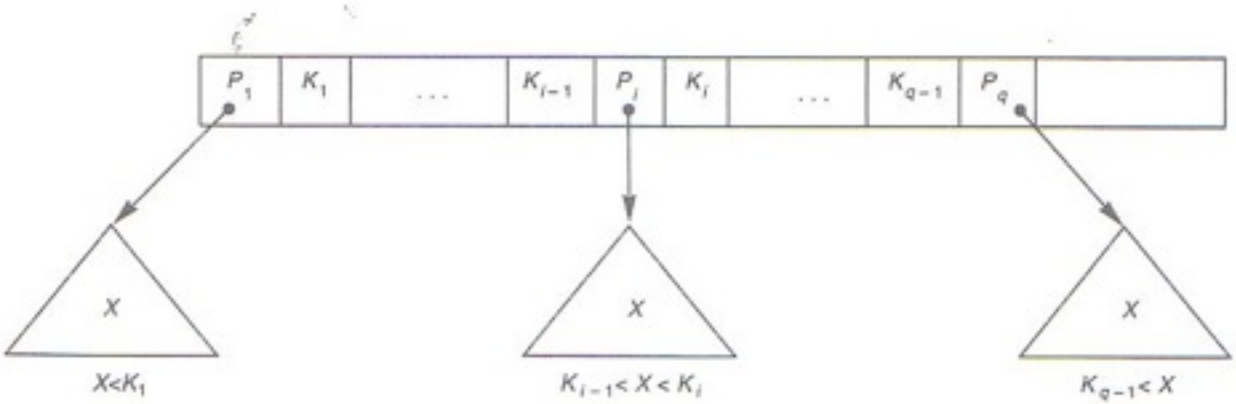
Alberi di ricerca - 2

- Un albero di ricerca di ***ordine p*** è un albero i cui nodi contengono al più *p-1 search value* e *p* puntatori nel seguente ordine

$$<P_1, K_1, P_2, K_2, .., P_{q-1}, K_{q-1}, P_q>$$

con $q \leq p$.

Ogni P_i è un puntatore ad un nodo figlio (o un puntatore nullo) e ogni K_i è un *search value* appartenente ad un insieme **totalmente ordinato** (assumiamo, per semplicità, che i *search value* K_i siano tutti distinti; non è difficile generalizzare la struttura proposta al caso in cui i *search value* non siano tutti distinti).



Alberi di ricerca - 3

- Ogni albero di ricerca deve soddisfare **due vincoli fondamentali**:

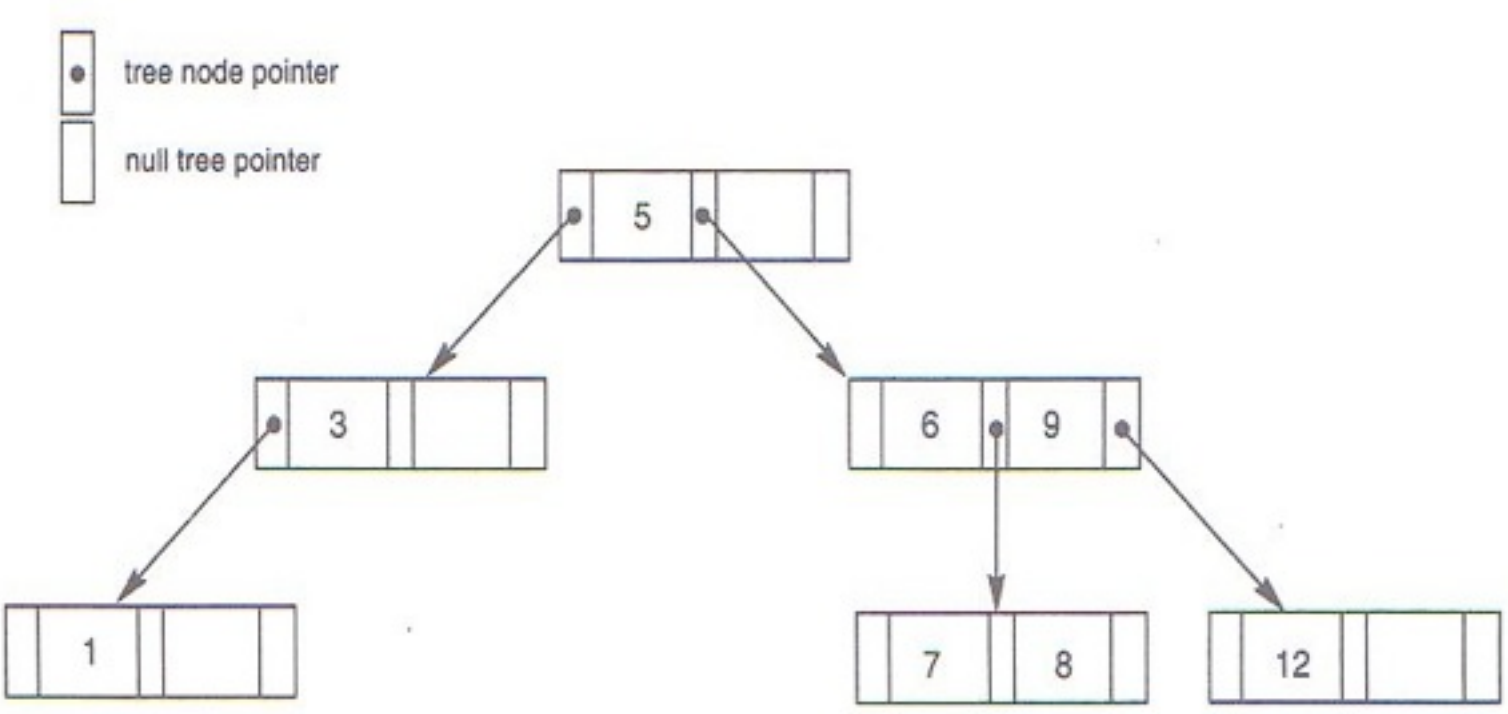
  (1) in ogni nodo, $K_1 < K_2 < ..... < K_{q-1}$;

  (2) per tutti i valori di X presenti nel sottoalbero puntato da P_i, vale la seguente relazione:

  - $K_{i-1} < X < K_i$ per $1 < i < q$;
  - $X < K_i$ per $i = 1$;
  - $K_{i-1} < X$ per $i = q$.

Alberi di ricerca - 4

Un esempio.



Alberi di ricerca - 5

- Un albero di ricerca può essere utilizzato per cercare record memorizzati su disco.
- I valori di ricerca (*search value*) possono essere i valori di uno dei campi del file (*search field*).
- Ad ogni valore di ricerca è associato un puntatore al record avente quel valore (oppure al blocco contenente quel record) nel file di dati.
- L'albero stesso può essere memorizzato su disco, assegnando **ad ogni nodo** dell'albero **un blocco**. Quando un nuovo record deve essere inserito, l'albero di ricerca deve essere aggiornato includendo il valore del campo di ricerca del nuovo record, col relativo puntatore al record (o al blocco che lo contiene), nell'albero di ricerca.

Alberi di ricerca - 6

- Per *inserire* (risp., *cancellare*) valori di ricerca nell'albero (risp., dall'albero) di ricerca sono necessari specifici algoritmi che garantiscano il rispetto dei due vincoli fondamentali.

  In generale, tali algoritmi non garantiscono che un albero di ricerca risulti sempre *bilanciato* (nodi foglia tutti allo stesso livello).

Perché vogliamo alberi di ricerca bilanciati?

*un **albero di ricerca bilanciato** evita che alcuni record richiedano molti più accessi a blocco di altri*

- In particolare, la *cancellazione* di record crea nell'albero nodi scarsamente popolati, aumentando la dimensione dello spazio inutilizzato e, conseguentemente, il numero di livelli.
- **Soluzione**: B-alberi e B^+-alberi.

B-alberi

- Un **B-albero** è un albero di ricerca con alcuni vincoli addizionali i quali garantiscono che l'albero sia bilanciato e lo spazio di memoria che risulta inutilizzato per effetto della cancellazione di record non diventi troppo grande.
- Per soddisfare tali vincoli addizionali, occorrono algoritmi di inserimento e cancellazione di record più complessi, soprattutto nei casi di:
  - inserimento di un record in un nodo pieno;
  - cancellazione di un record da un nodo che diventa pieno per meno del 50% della sua dimensione.

B-alberi: condizioni - 1

- Formalmente, un B-albero di *ordine p*, se usato come struttura di accesso su un campo chiave per la ricerca di record in un file di dati, deve soddisfare le seguenti condizioni:

(1) ogni nodo interno del B-albero ha la forma:

$$<P_1, <K_1, Pr_1>, P_2, <K_2, Pr_2>, ..., <K_{q-1}, Pr_{q-1}>, P_q>;$$

dove $q \leq p$.

P_i è un *tree pointer* (puntatore ad un altro nodo del B-albero), Pr_i è un *data pointer* (puntatore ad un record con valore del campo chiave di ricerca uguale a K_i o al blocco che contiene tale record);

(2) per ogni nodo, si ha che:

$$K_1 < K_2 < .... < K_{q-1};$$

(3) ogni nodo ha al più *p tree pointer*;

B-alberi: condizioni - 2

(4) per tutti i valori X della chiave di ricerca appartenenti al sottoalbero puntato da P_i, si ha che:

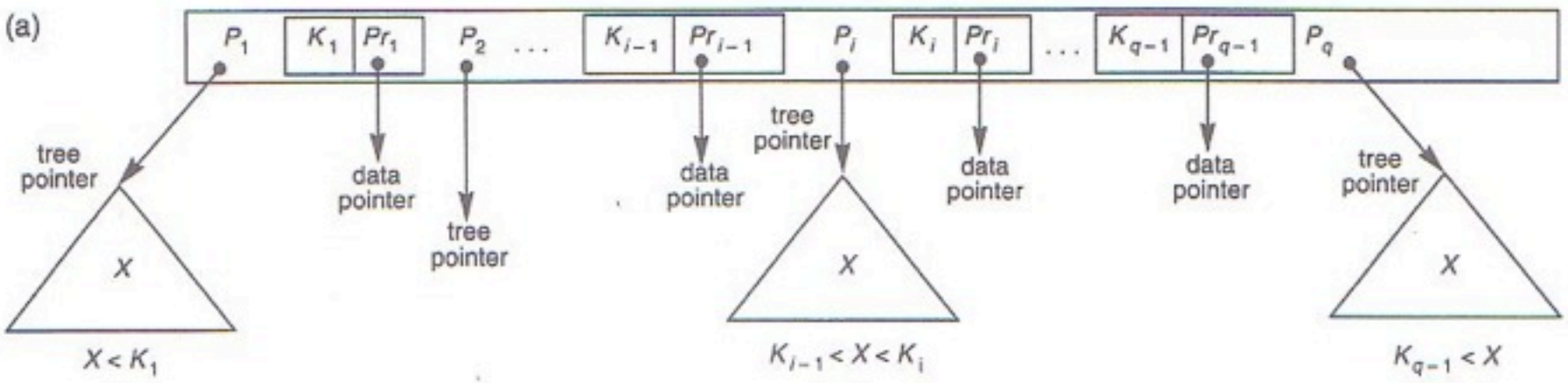
- $K_{i-1} < X < K_i$ per $1 < i < q$;
- $X < K_i$ per $i = 1$;
- $K_{i-1} < X$ per $i = q$;

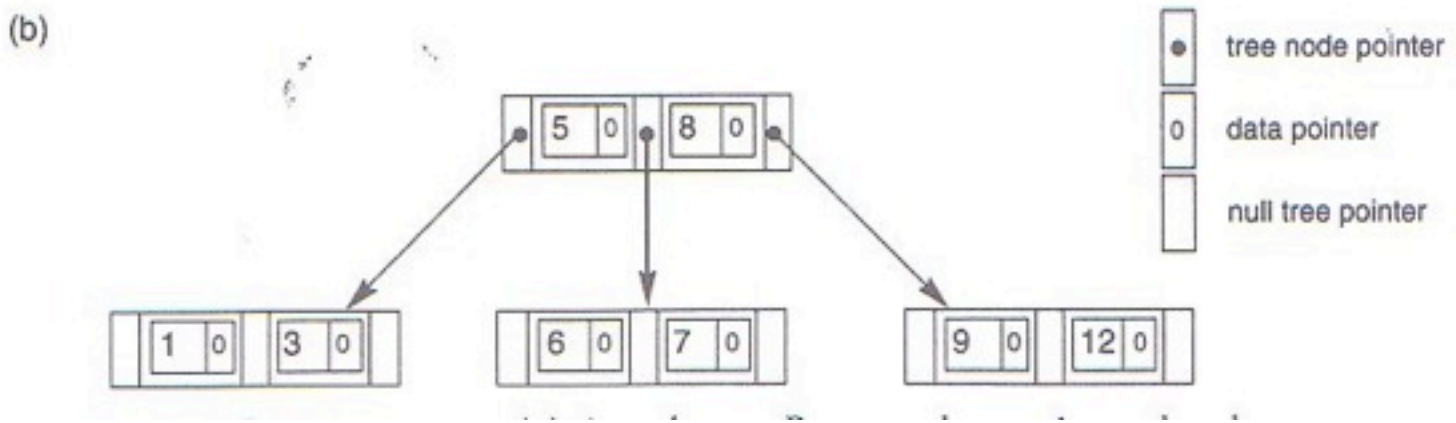
(5) ogni nodo, esclusa la radice, ha almeno $\lceil p/2 \rceil$ tree pointer (la radice ha almeno due *tree pointer*, a meno che non sia l'unico nodo dell'albero);

(6) un nodo con q *tree pointer*, con $q \leq p$, ha $q-1$ campi chiave di ricerca (e $q - 1$ *data pointer*);

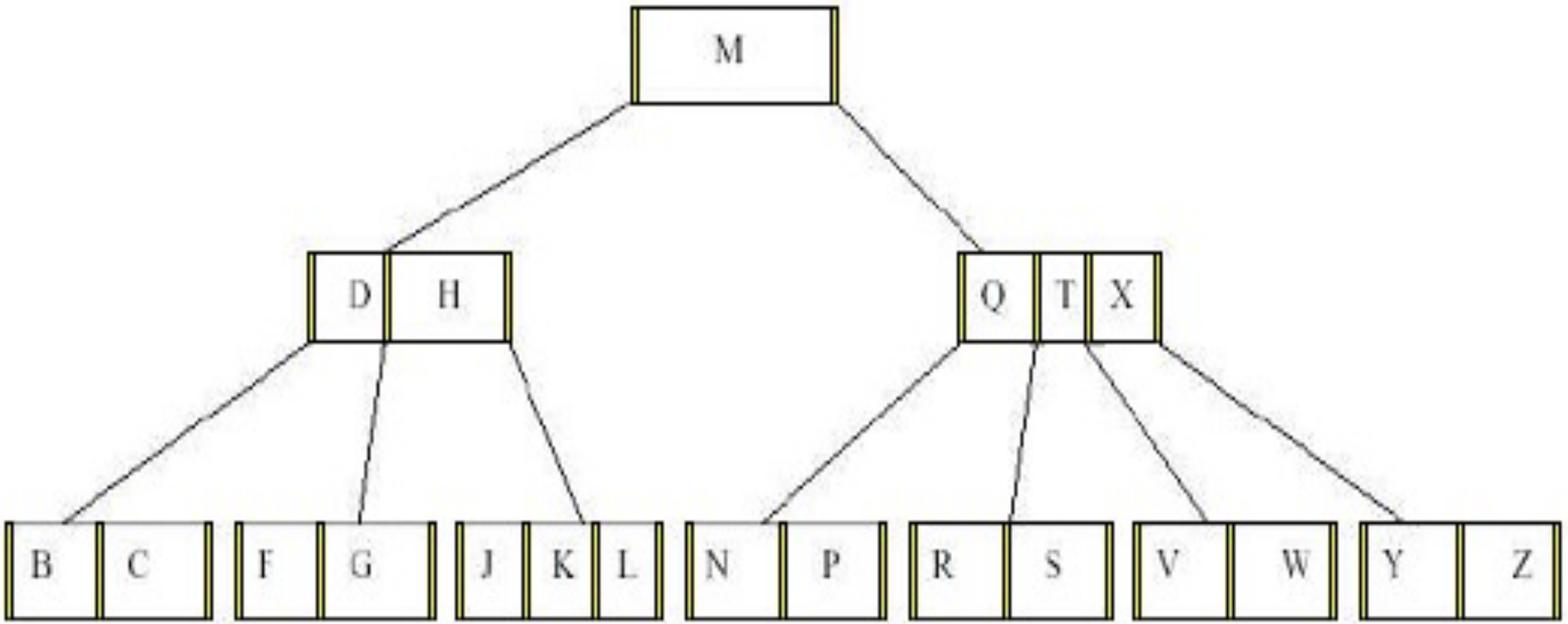
(7) tutti i nodi foglia sono posti allo stesso livello (i nodi foglia hanno la stessa struttura dei nodi interni, ad eccezione del fatto che tutti i loro *tree pointer* P_i sono nulli).

B-alberi: struttura





B-alberi: un esempio



B-albero di ordine p = 4.

Dell'altezza di un B-albero

Detta **h** l'altezza di un B-albero d'ordine **p = 2t,** con **n** valori di ricerca (chiavi) e grado minimo (numero di tree pointer in un nodo diverso dalla radice) **t ≥ 2**, si ha che:

$$\mathbf{h \leq \log_t ((n+1)/2)}$$

Prova. Se un B-albero ha altezza **h**, il numero dei suoi nodi è minimo se la radice contiene una sola chiave e tutti gli altri nodi contengono **t-1** chiavi. In tal caso, vi sono **2** nodi a profondità **1**, **2t** nodi a profondità **2**, $\mathbf{2t^2}$ nodi a profondità **3**, e così via, fino alla profondità **h** dove vi saranno $\mathbf{2t^{h-1}}$ nodi. Ne segue che il numero di chiavi **n** soddisfa la seguente disuguaglianza:

$$\mathbf{n \geq 1 + (t-1) \cdot \sum_{i=1,..,h} 2t^{i-1} = 1 + 2 \cdot (t-1) \cdot (t^h - 1)/(t - 1) = 2t^h - 1}$$

da cui la tesi.

Operazioni di base su un B-albero

Operazioni:

1. **ricerca** su un B-albero;
2. **creazione** di un B-albero;
3. **inserimento** in un B-albero;
4. **cancellazione** da un B-albero (cenni).

(see *Introduction to Algorithms*, Cormen et al., The MIT Press)

Assunzioni

- La radice di un B-albero sia sempre in memoria principale (non serve una DISKREAD; qualora la radice venga modificata occorre una DISKWRITE).

- Prima di passare un nodo come parametro attuale di un'operazione di ricerca, inserimento o cancellazione, viene eseguita una DISKREAD su di esso.

- Tutte le procedure sono di tipo *one-pass* (procedono dalla radice verso le foglie, senza necessità di back up).

Notazione

La radice di un B-albero **T** è **root[T]**.

L'ordine di **T** è **p = 2t**.

Per ogni nodo **x** di **T**:

- **n[x]** è il numero di chiavi contenute in **x**;
- $\mathbf{k_1[x]}$, $\mathbf{k_2[x]}$, .., $\mathbf{k_{n[x]}[x]}$ sono le chiavi contenute in **x** (memorizzate in ordine non decrescente);
- se **x** è un nodo interno, $\mathbf{c_1[x]}$, $\mathbf{c_2[x]}$, .., $\mathbf{c_{n[x]+1}[x]}$ sono gli **n[x] +1** puntatori contenuti in **x**;
- **leaf[x]** è un valore Booleano (VERO se **x** è una foglia, FALSO altrimenti).

Ricerca su un B-albero

BTREESEARCH(x,k)

i ← 1

while i ≤ n[x] and k > key_i[x]

do i ← i+1

if i ≤ n[x] and k = key_i[x]

then return (x,i)

if leaf[x] **then return** NIL

else DISKREAD(c_i[x])

return BTREESEARCH(c_i[x],k)

la procedura viene chiamata con la coppia di parametri attuali (root[T],K), dove K è la chiave oggetto della ricerca

Creazione di un B-albero

```
BTreeCreate(T)
    x ← AllocateNode()
    leaf[x] ← TRUE
    n[x] ← 0
    DiskWrite(x)
   root[T] ← x
```

la procedura BTreeCreate viene utilizzata per creare un nodo radice vuoto; le chiavi vengono aggiunte utilizzando la procedura BTreeIInsert

Inserimento in un B-albero

```
BTreeInsert(T,k)
    r ← root[T]
    if n[r] = 2t – 1 then
    s ← AllocateNode()
    root[T] ← s
    leaf[s] ← FALSE
    n[s] ← 0
    c1[s] ← r
    BTreeSplitChild(s,1,r)
    BTreeInsertNonfull(s,k)
    else BTreeInsertNonfull(r,k)
```

Splitting di un nodo

BTREESPLITCHILD(x,i,y)

z ← ALLOCATENODE()

leaf[z] ← leaf[y]; n[z] ← t - 1

for j ← 1 **to** t - 1 **do** $key_j[z] \leftarrow key_{j+t}[y]$

if not leaf[y] **then for** j ← 1 **to** t **do** $c_j[z] \leftarrow c_{j+t}[y]$

n[y] ← t - 1

for j ← n[x] + 1 **downto** i + 1 **do** $c_{j+1}[x] \leftarrow c_j[x]$

$c_{i+1}[x] \leftarrow z$

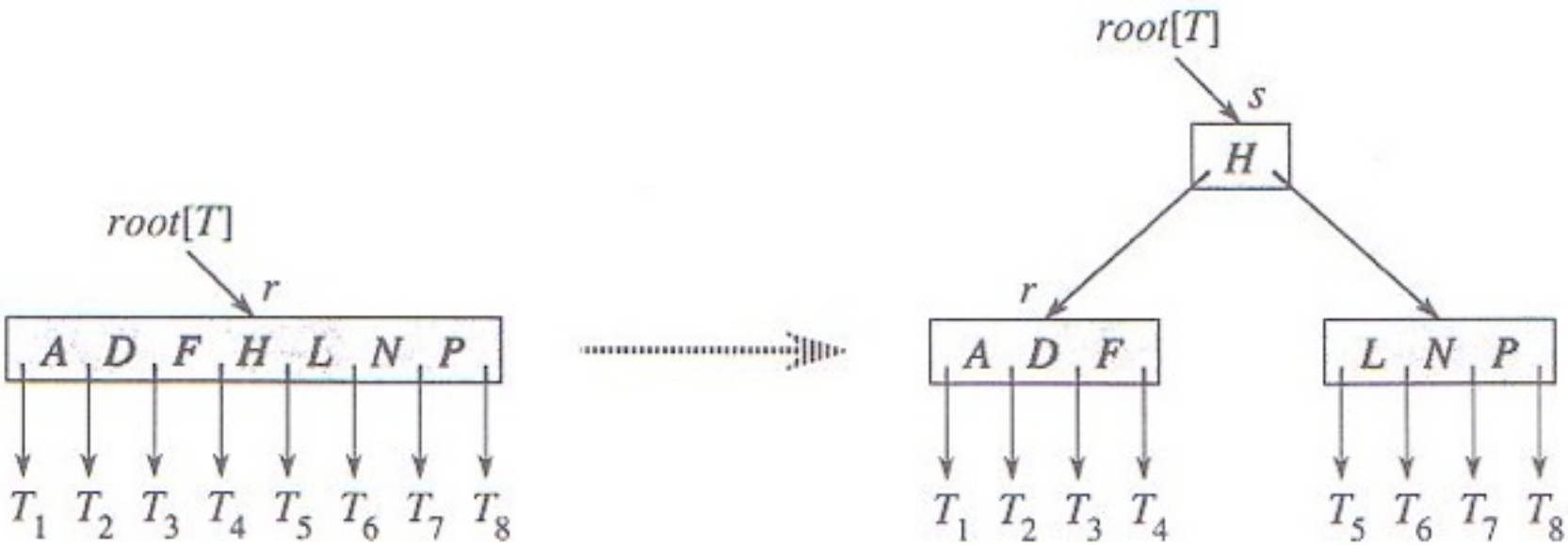
for j ← n[x] **downto** i **do** $key_{j+1}[x] \leftarrow key_j[x]$

$key_i[x] \leftarrow key_t[y]$; n[x] ← n[x] + 1

DISKWRITE(y); DISKWRITE(z); DISKWRITE(x)

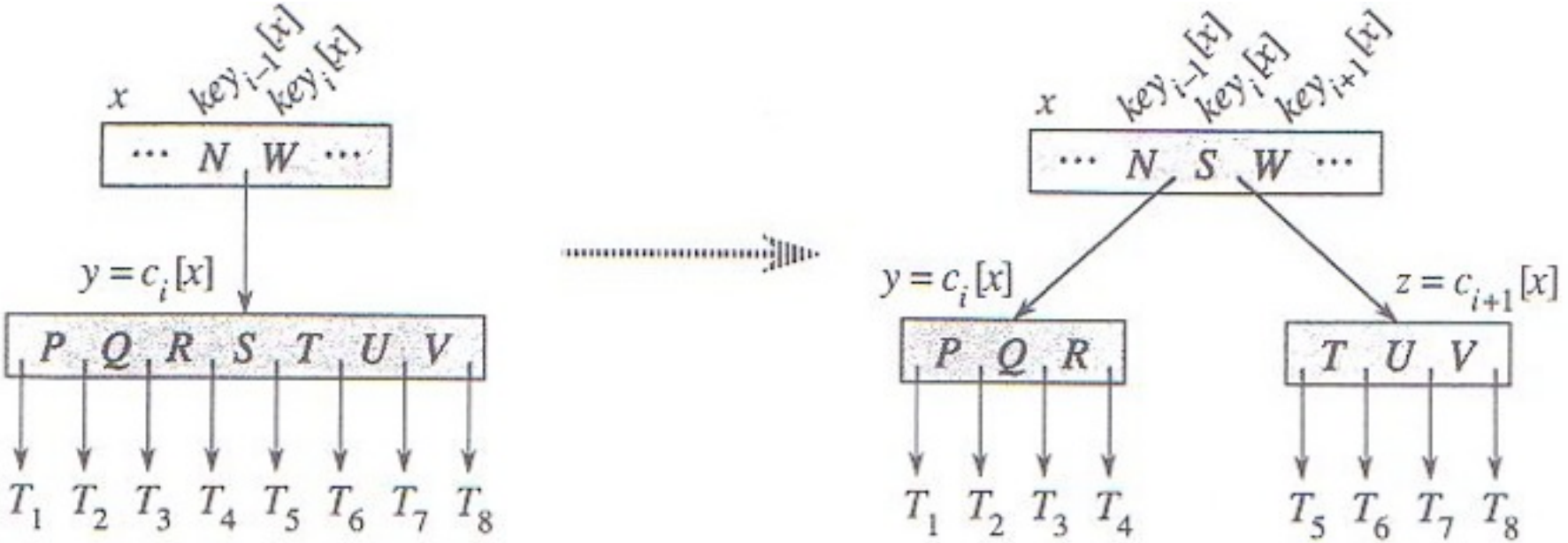
Splitting di un nodo: esempio 1

Esempio. Splitting della radice con t=4. Il nodo r è suddiviso in 2 nodi e viene creata una nuova radice s. Il nodo s contiene la chiave mediana H di r e ha le due parti di r come suoi figli. Tale splitting fa crescere di 1 l'altezza del B-albero.



Splitting di un nodo: esempio 2

Esempio. Splitting di un nodo con t=4. Il nodo y è suddiviso in 2 nodi, y e z, e la chiave mediana S di y è spostata nel nodo x genitore di y.



Inserimento in un nodo non pieno

BTREEINSERTNONFULL(x,k)

i ← n[x]

if leaf[x] **then while** i ≥ 1 and k < key_i[x]

do key_{i+1}[x] ← key_i[x]; i ← i - 1

key_{i+1}[x] ← k; n[x] ← n[x] + 1

DISKWRITE(x)

else while i ≥ 1 and k < key_i[x] **do** i ← i - 1

i ← i + 1; DISKREAD(c_i[x])

if n[c_i[x]] = 2t - 1

then BTREESPLITCHILD(x,i, c_i[x])

if k > key_i[x] **then** i ← i + 1

BTREEINSERTNONFULL(c_i[x],k)

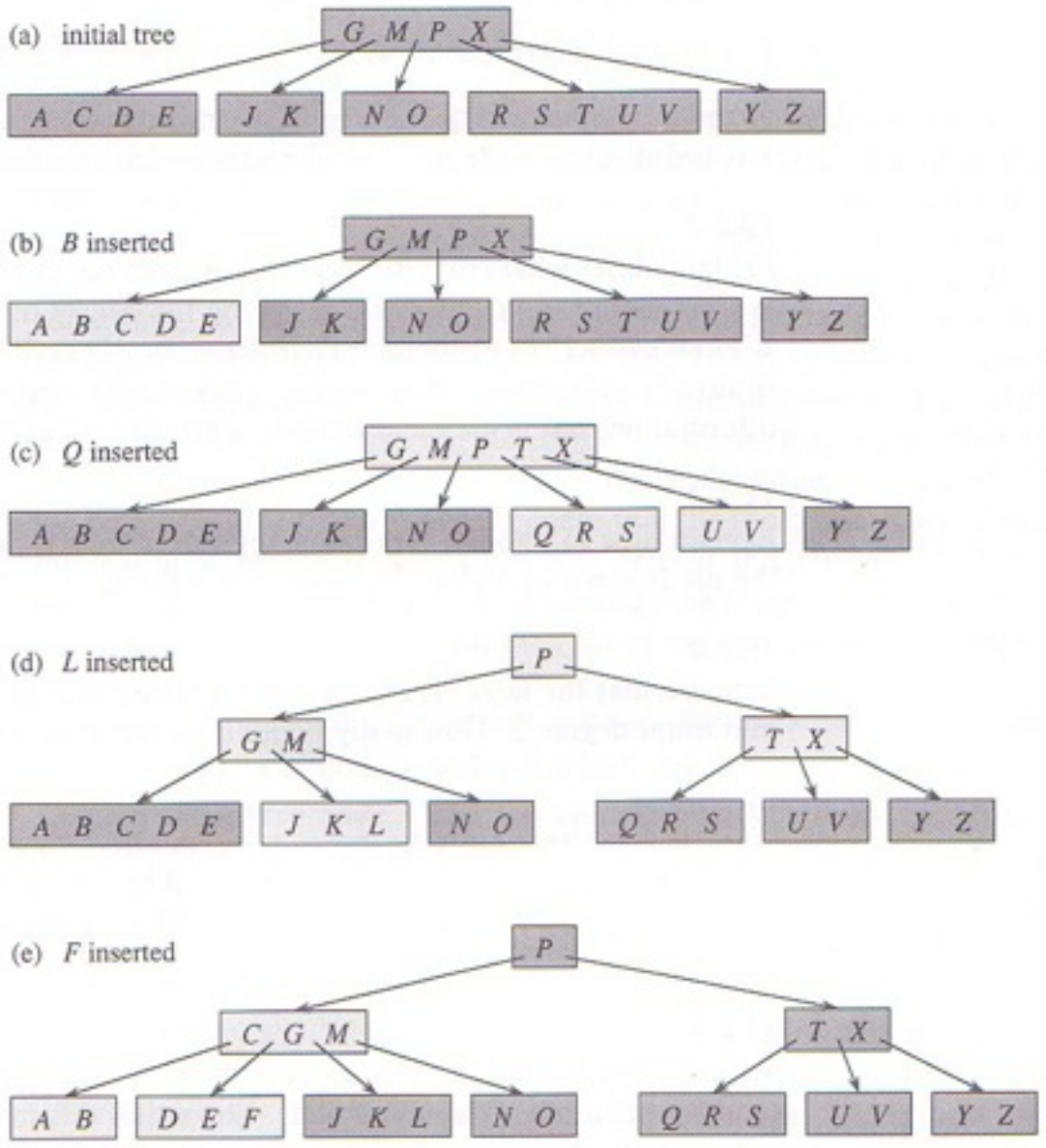
Osservazioni

La complessità sia dell'operazione di ricerca sia dell'operazione di inserimento è **logaritmica** (l'operazione di creazione di un B-albero vuoto ha costo costante)

Nel caso dell'operazione di inserimento, è facile vedere che in ogni istante è sufficiente mantenere in memoria un numero costante (e non logaritmico) di pagine

Esercizio. Costruire una procedura two-pass per l'operazione di inserimento di una chiave in un B-albero e confrontarla con la soluzione proposta.

Esempio di inserimento (t=3)



Cancellazione di una chiave da un B-albero - 1

Descrizione informale. Immaginiamo di dover eliminare la chiave **k** dal sottoalbero radicato in **x**. La struttura della procedura di cancellazione BTREEDELETE dovrà garantire che ogni qualvolta essa verrà chiamata ricorsivamente su un nodo **x**, il numero di chiavi in **x** risulti almeno pari a **t** (e non **t-1**).

In talune occasioni, occorrerà spostare preliminarmente una chiave da un nodo a un suo nodo figlio prima di poter eseguire una chiamata ricorsiva della procedura su tale figlio.

In tal modo anche le cancellazioni potranno essere fatte sostanzialmente con una **procedura one-pass** senza back up.

Cancellazione di una chiave da un B-albero - 2

Qualora il nodo radice **x** diventi un nodo interno privo di chiavi, esso verrà rimosso e il suo unico figlio $\mathbf{c_1[x]}$ diventerà la nuova radice dell'albero, che diminuirà la sua altezza di 1 e conserverà la proprietà di contenere almeno una chiave nel nodo radice (a meno che il B-albero non sia vuoto).

Si possono presentare le seguenti situazioni:

1. Se **k** appartiene a **x** e **x** è una foglia, si cancella **k** da **x**.
2. Se **k** appartiene a **x** e **x** è un nodo interno, vanno presi in considerazione i seguenti casi:

2a. Se il figlio **y** che precede **k** in **x** ha almeno **t** chiavi, si recupera il predecessore **k'** di **k** nel sottoalbero radicato in **y**; si cancella ricorsivamente **k'** (nel sottoalbero radicato in **y**) e si rimpiazza **k** con **k'** in **x**;

Cancellazione di una chiave da un B-albero - 3

2b. Simmetricamente, se il figlio **z** che segue **k** in **x** ha almeno **t** chiavi, si recupera il successore **k'** di **k** nel sottoalbero radicato in **z**; si cancella ricorsivamente **k'** (nel sottoalbero radicato in **z**) e si rimpiazza **k** con **k'** in **x**;

2c. Se sia **y** sia **z** hanno solo **t-1** chiavi, si spostano **k** e tutti i valori di **z** in **y**; in tal modo da **x** vengono rimossi sia **k** che il puntatore a **z** e **y** contiene **2t-1** chiavi. Il nodo **z** viene rilasciato e si cancella ricorsivamente **k** da **y**.

3. Se **k** non è presente in un nodo interno **x**, determinare la radice $\mathbf{c_i[x]}$ del sottoalbero che deve contenere **k**; se $\mathbf{c_i[x]}$ ha solo **t-1** chiavi, si esegue o il passo 3a o il passo 3b, in modo da garantire che la procedura venga richiamata su un nodo contenente almeno **t** chiavi. Eseguito il passo selezionato, si prosegue chiamando ricorsivamente la procedura sul figlio di **x** appropriato.

Cancellazione di una chiave da un B-albero - 4

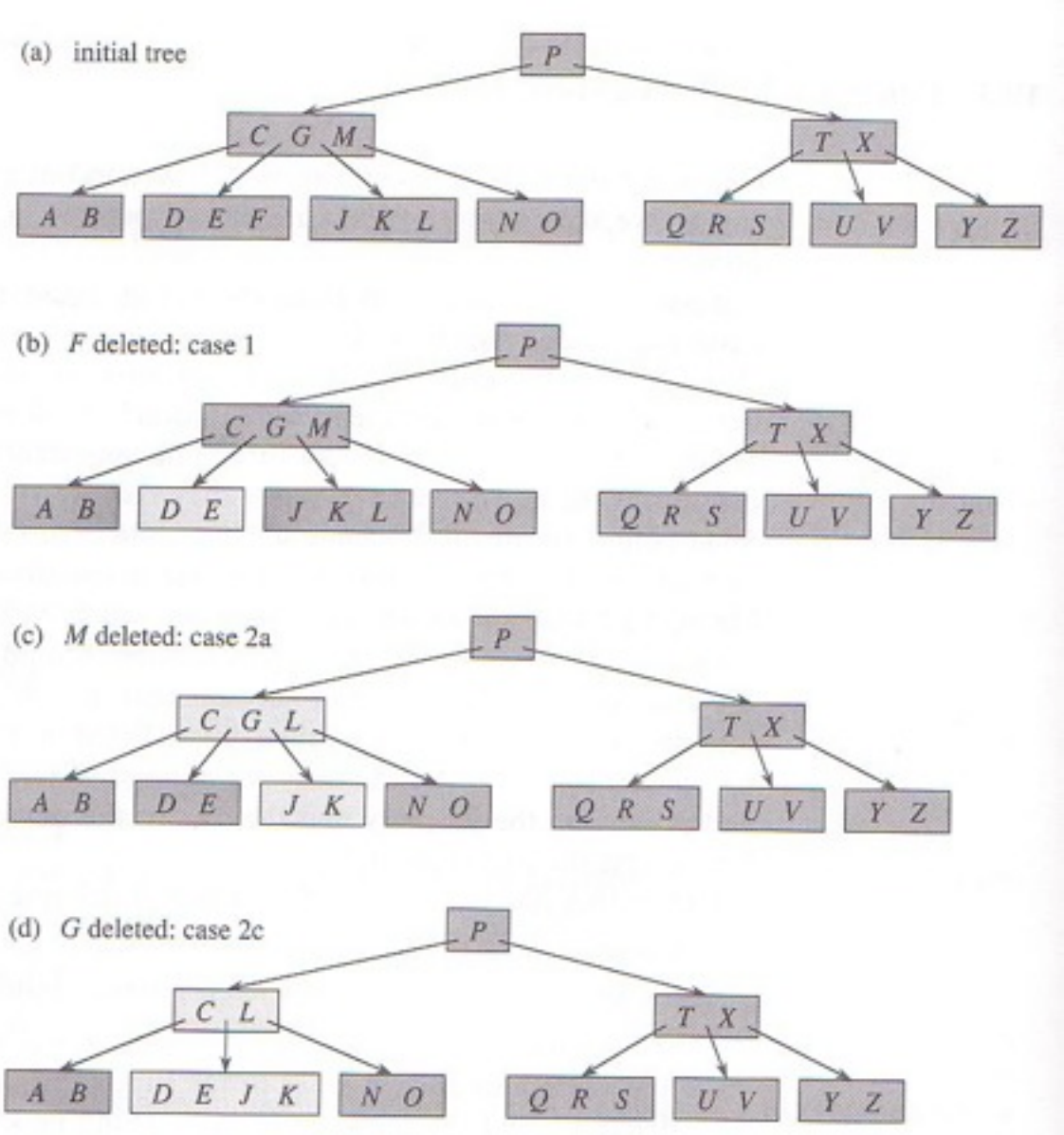
3a. Se $\mathbf{c_i[x]}$ ha solo **t-1** chiavi, ma possiede un vicino (sibling) immediato, destro o sinistro, **y** con almeno **t** chiavi, si sposta una chiave da **x** a $\mathbf{c_i[x]}$, si sposta una chiave da **y** in **x** e si sposta il figlio appropriato di **y** in $\mathbf{c_i[x]}$;

3b. Se $\mathbf{c_i[x]}$ e i suoi vicini hanno **t-1** chiavi, $\mathbf{c_i[x]}$ viene fuso con uno dei suoi vicini, operazione che impone anche di spostare una chiave da **x** nel nuovo nodo (chiave che diventa la mediana di tale nodo).

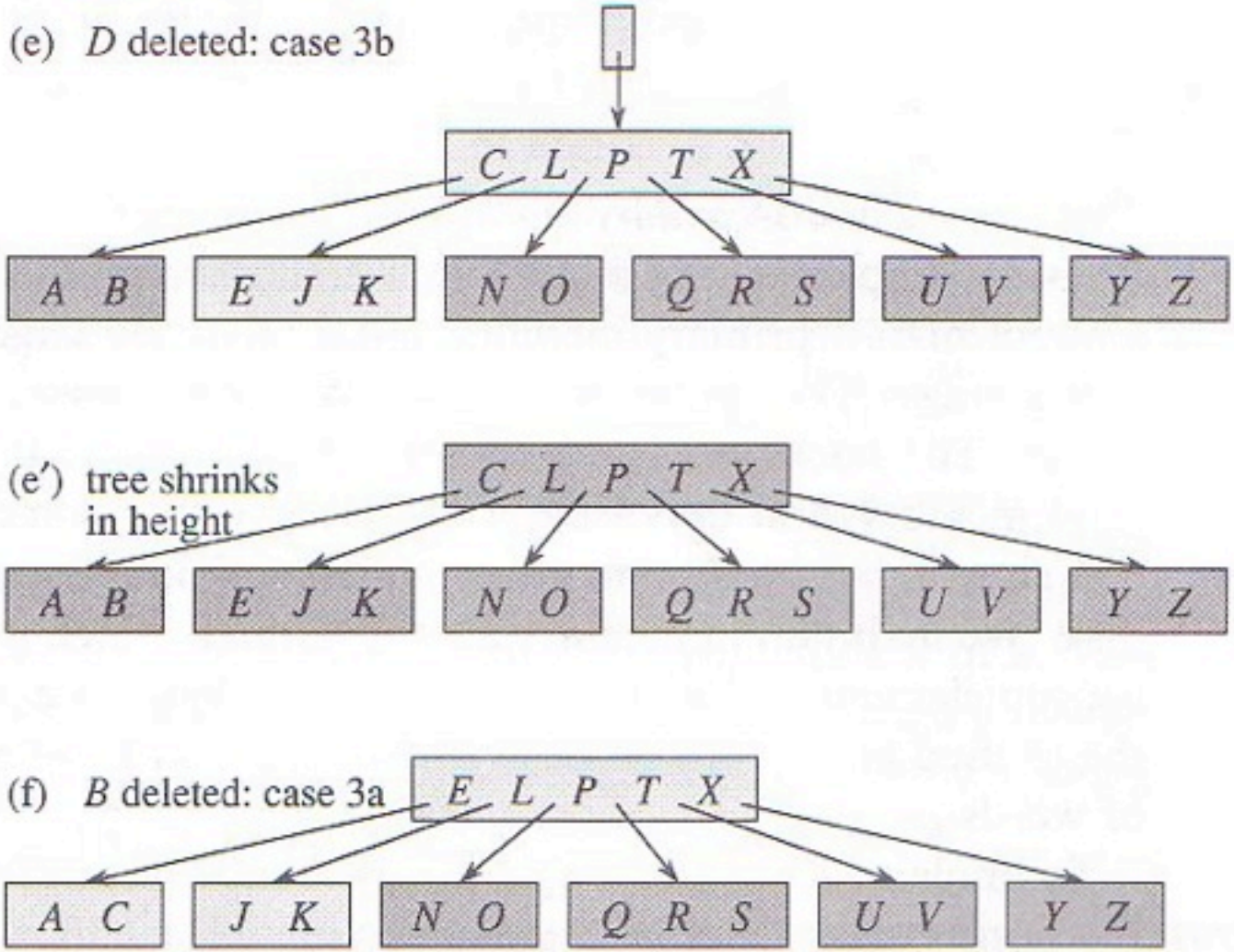
Osservazioni.

- Dato che la maggior parte delle chiavi si trova nelle foglie, la maggior parte delle cancellazioni coinvolgerà le foglie.
- Al pari delle operazioni di ricerca e inserimento, anche la complessità dell'operazione di cancellazione è **logaritmica**.

Esempio di cancellazione (t=3) - 1



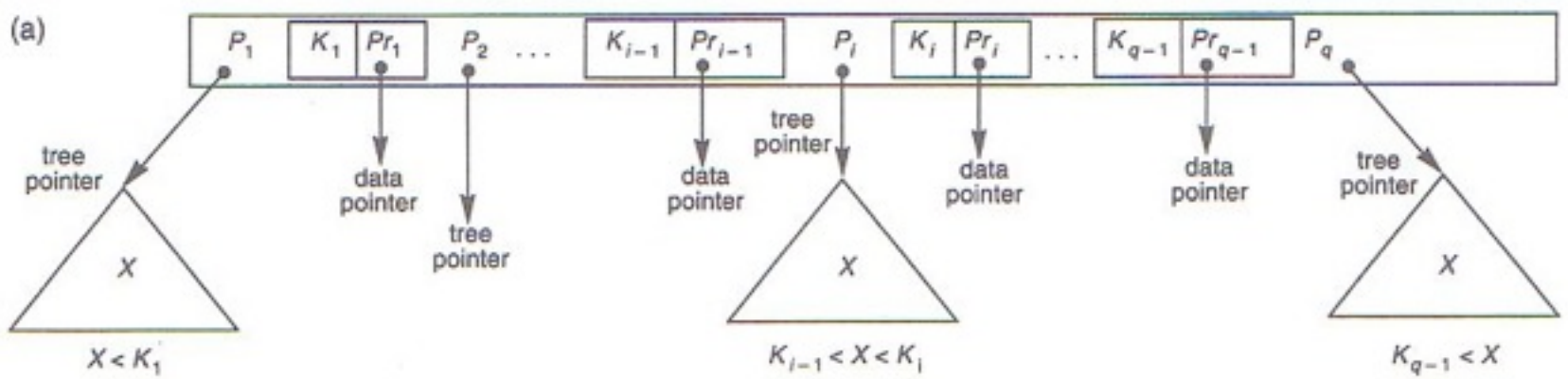
Esempio di cancellazione (t=3) - 2



B-alberi: come stabilire l'ordine? - 1

Esempio. Supponiamo che il campo di ricerca (chiave) sia composto da V di dimensione pari a 9 byte (abbreviato V=9 byte), un blocco da B=512 byte, un *record data pointer* da P_r=7 byte e un *block pointer* da P=6 byte.

Ogni nodo del B-albero può contenere al più *p tree pointer* e *p-1* campi chiave di ricerca.



B-alberi: come stabilire l'ordine? - 2

Tali puntatori possono essere tutti contenuti in un singolo blocco se ad ogni nodo del B-albero corrisponde un blocco:

$$p \cdot P + [(p-1) \cdot (P_r+V)] \leq B$$

$$p \cdot 6 + [(p-1) \cdot (7+9)] \leq 512$$

$$22 \cdot p \leq 528$$

E' ragionevole assegnare a **p** un valore elevato che soddisfi la disuguaglianza (ad esempio, **p=23**).

Il prossimo esempio suggerisce una modalità di calcolo del numero di blocchi e di livelli di un B-albero.

Numero livelli e numero blocchi - 1

Esempio. Supponiamo che il campo di ricerca dell'esempio precedente sia un *nonordering key field* e che si voglia costruire un B-albero su questo campo.

Assumiamo, inoltre, che ogni nodo del B-albero sia pieno al **69%**.

Ogni nodo conterrà (mediamente) $\mathbf{p \cdot 0.69 = 23 \cdot 0.69 = 16}$ puntatori (e **15** valori del campo chiave di ricerca).

Il *fan-out* medio sarà **fo=16**.

Numero livelli e numero blocchi - 2

Partendo dal nodo radice, vediamo quanti valori e puntatori esistono mediamente ad ogni livello:

- *radice: 1 nodo 15 entry 16 puntatori*
- *livello 1: 16 nodi 240 entry 256 puntatori*
- *livello 2: 256 nodi 3840 entry 4096 puntatori*
- *livello 3: 4096 nodi 61440 entry*

Ad ogni livello, è possibile determinare il numero di entry moltiplicando il numero totale di puntatori al livello precedente per 15 (il numero medio di entry in ogni nodo).

Date le dimensioni del blocco, la dimensione dei puntatori e la dimensione del campo chiave dell'esempio, un B-albero a due livelli avrà mediamente 3840 + 240 +15 entry, mentre un B-albero su tre livelli avrà mediamente 65535 entry.

Dai B-alberi ai B^+-alberi

- Un B^+-albero è un B-albero in cui i data pointer sono memorizzati solo nei nodi foglia dell'albero. La struttura dei nodi foglia differisce, quindi, da quella dei nodi interni.
- Se il campo di ricerca è un campo chiave, i nodi foglia hanno per ogni valore del campo di ricerca una entry ed un puntatore ad un record. Se un campo di ricerca non è un campo chiave, i puntatori indirizzano un blocco che contiene i puntatori ai record del file di dati, rendendo così necessario un passo aggiuntivo per l'accesso ai dati.
- I nodi foglia di un B^+-albero sono generalmente messi fra loro in relazione (ciò viene sfruttato nel caso di range query). Essi corrispondono al primo livello di un indice. I nodi interni corrispondono agli altri livelli di un indice.
- Alcuni valori del campo di ricerca presenti nei nodi foglia sono ripetuti nei nodi interni per guidare la ricerca.

Definizione dei B$^+$-alberi: condizioni - 1

- La struttura dei nodi interni (di ordine p) di un B$^+$-albero è la seguente:

(1) ogni nodo interno del B$^+$-albero ha la forma:

$$<P_1, K_1, P_2, K_2, ..., P_{q-1}, K_{q-1}, P_q >,$$

dove $q \leq p$ e ogni P_i è un *tree pointer* (puntatore ad un altro nodo del B$^+$-albero)

(2) per ogni nodo interno, si ha che $K_1 < K_2 < .... < K_{q-1}$

(3) ogni nodo interno ha al più *p tree pointer*

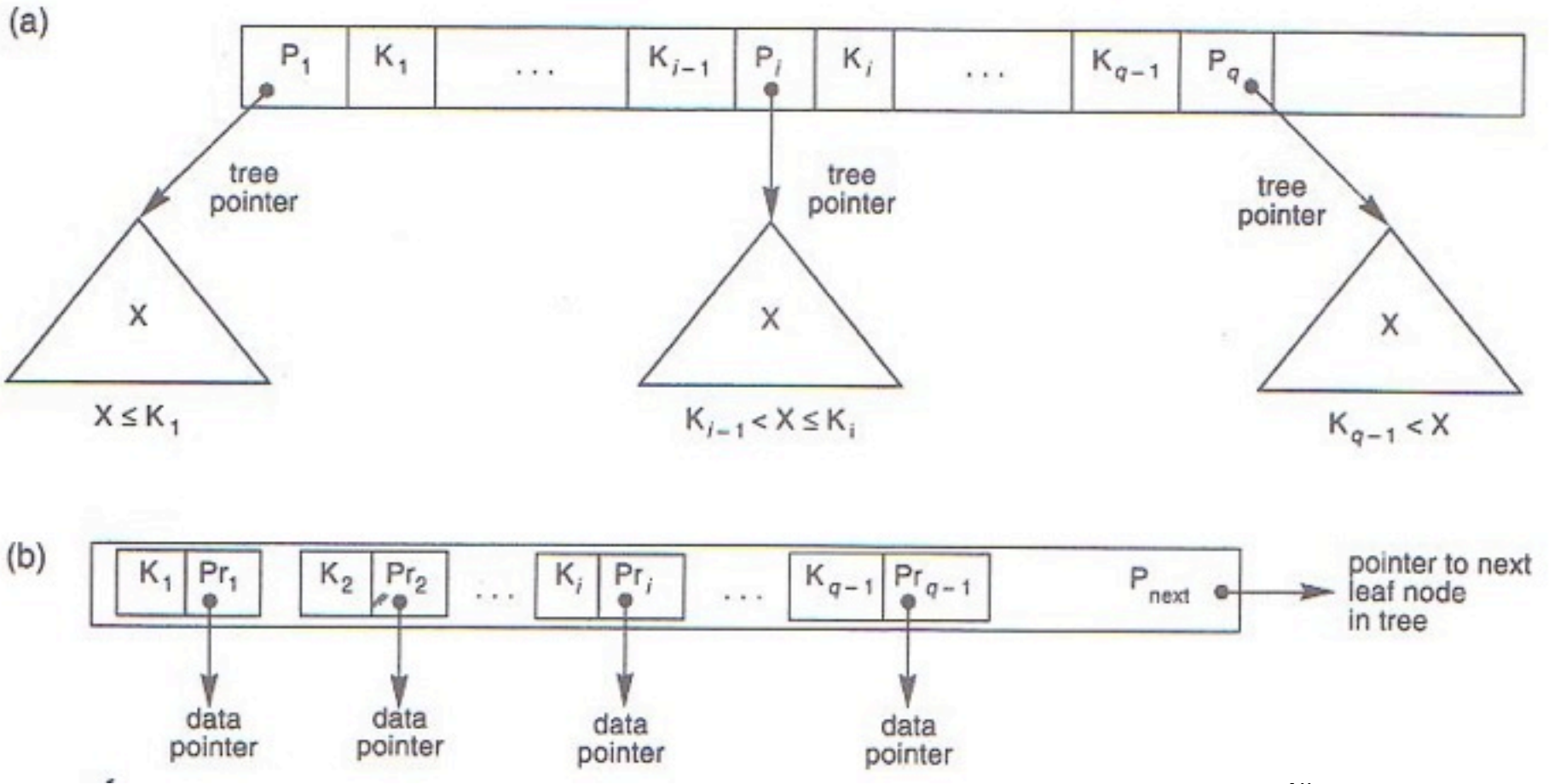
(4) per tutti i valori X della *search key* nel sottoalbero puntato da P_i, si ha che

- $K_{i-1} < X \leq K_i$ for $1 < i < q$,
- $X \leq K_i$ for $i = 1$,
- $K_{i-1} < X$ for $i = q$

B$^+$-alberi: condizioni - 2

(5) ogni nodo interno, esclusa la radice, ha almeno $\lceil p/2 \rceil$ *tree pointer*. La radice ha almeno 2 *tree pointer* se è un nodo interno.

(6) un nodo interno con q *tree pointer*, con $q \leq p$, ha $q-1$ campi di ricerca.



B^+-alberi: condizioni - 3

- La struttura dei nodi foglia (di ordine p_{leaf}) di un B^+-albero è la seguente:

(1) ogni nodo foglia è della forma:

$$< <K_1, Pr_1>, <K_2, Pr_2>, ..., <K_q, Pr_q> P_{next}>,$$

dove $q \leq p_{leaf}$, Pr_i è un *data pointer* e P_{next} è un *tree pointer* punta al successivo nodo foglia del B^+-albero

(2) per ogni nodo, si ha che $K_1 < K_2 < .... < K_q$

(3) ogni Pr_i è un *data pointer* che punta al record con valore del campo di ricerca uguale a K_i o ad un blocco contenente tale record (o ad un blocco di puntatori ai record con valore del campo di ricerca uguale a K_i, se il campo di ricerca non è una chiave)

(4) ogni nodo foglia ha almeno $\lceil p_{leaf}/2 \rceil$ valori

(5) tutti i nodi foglia sono dello stesso livello

Generazione di un B^+-alberi

- Per un B^+-albero costruito su un campo chiave,
  - i puntatori appartenenti ai nodi interni sono *tree pointer* a blocchi che sono nodi dell'albero
  - i puntatori appartenenti ai nodi foglia sono *data pointer* a record (o blocchi) del file dei dati, ad eccezione di P_{next} che è un *tree pointer* al successivo nodo foglia

B^+-alberi: come stabilire l'ordine? - 1

Esempio. Per calcolare l'ordine *p* (dei nodi interni) di un B^+-albero, supponiamo che il campo chiave di ricerca abbia dimensione *V=9* byte, il blocco *B=512* byte, il puntatore ad un record P_r*=7* byte e il puntatore ad un blocco *P=6* byte.

Un nodo interno di un B^+-albero può avere fino a *p tree pointer* e *p-1* valori del campo di ricerca (che devono essere contenuti in un singolo blocco):

$$\mathbf{p \cdot P + [(p-1) \cdot V] \leq B}$$

$$\mathbf{p \cdot 6 + [(p-1) \cdot 9] \leq 512}$$

$$\mathbf{15 \cdot p \leq 521}$$

B^+-alberi: come stabilire l'ordine? - 2

Si sceglie come valore di **p** il più alto valore che soddisfa la disuguaglianza, ossia **p=34**. Questo valore è più alto di quello del corrispondente B-albero (**p=23**): si ha un valore di *fan-out* più elevato e un numero maggiore di entry in ogni nodo interno rispetto al corrispondente B-albero.

I nodi foglia del B^+-albero hanno lo stesso numero di valori e di *data pointer*, più un *next pointer*.

L' ordine p_{leaf} dei nodi foglia è determinato nel modo seguente:

$$[p_{leaf} \cdot (P_r + V)] + P \leq B$$

$$[p_{leaf} \cdot (7 + 9)] + 6 \leq 512$$

$$16 \cdot p_{leaf} \leq 506$$

$$P_{leaf} = 31$$

Numero livelli e numero blocchi - 1

Esempio. Supponiamo di voler costruire un B^+-albero sul campo dell'esempio precedente. Assumiamo, inoltre, che ogni nodo del B^+-albero sia pieno al **69%.**

Ogni nodo interno (mediamente) avrà $\mathbf{p \cdot 0.69 = 34 \cdot 0.69 =}$ **23** puntatori (e **22** valori).

Ogni nodo foglia (mediamente) avrà $\mathbf{p_{leaf} \cdot 0.69 = 31 \cdot 0.69 =}$ **21** puntatori a record di dati.

Numero livelli e numero blocchi - 2

Un B^+-albero avrà il seguente numero medio di entry ad ogni livello:

- *radice: 1 nodo 22 entry 23 puntatori*
- *livello 1: 23 nodi 506 entry 529 puntatori*
- *livello 2: 529 nodi 11638 entry 12167 puntatori*
- *livello 3:12167 nodi 255507 entry*

Fissate le dimensioni del blocco, la dimensione del puntatore e la dimensione del campo chiave di ricerca, un B^+-albero di tre livelli avrà 255507 entry (contro le 65535 entry di un B-albero di tre livelli)

Operazioni di base su un B^+-albero

Operazioni:

1. **ricerca** su un B^+-albero;
2. **creazione** di un B^+-albero (standard);
3. **inserimento** in un B^+-albero;
4. **cancellazione** da un B^+-albero (esempio).

Assunzioni

- Per implementare gli algoritmi di ricerca, inserimento e cancellazione, è necessario associare dell'informazione aggiuntiva ad ogni nodo:

  - tipo di nodo (interno o foglia);

  - numero di entry correnti in ciascun nodo;

  - puntatore al nodo genitore e puntatori ai nodi vicini dello stesso livello

Algoritmo di ricerca

Ricerca di un record con chiave k:

x ← indirizzo blocco contenente il nodo radice

DISKREAD(x)

while (x non è un nodo foglia) **do**

if $k \leq x.K_1$ **then** $x \leftarrow x.P_1$

else if $k > x.K_{n[x]}$ **then** $x \leftarrow x.P_{n[x]+1}$

else cerca la entry i contenuta nel nodo x tale che $x.K_{i-1} < k \leq x.K_i$

$x \leftarrow x.P_i$

DISKREAD(x)

cerca la entry (k_i, Pr_i) con $k = k_i$ contenuta nel nodo x

if trovata **then** DISKREAD(Pr_i) **else** fallimento ricerca

Algoritmo di inserimento - 1

Inserimento di un record con chiave k (con S stack ausiliario inizialmente vuoto):

x ← indirizzo blocco contenente il nodo radice

DISKREAD(x)

while (x non è un nodo foglia) **do**

inserisci x in S # S contiene i nodi genitore necessari

in caso di split

if $k \leq x.K_1$ **then** $x \leftarrow x.P_1$ **else**

if $k > x.K_{n[x]}$ **then** $x \leftarrow x.P_{n[x]+1}$ **else**

cerca la entry i contenuta nel

nodo x tale che $x.K_{i-1} < k \leq x.K_i$

$x \leftarrow x.P_i$

DISKREAD(x)

Algoritmo di inserimento - 2

cerca la entry (k_i,Pr_i) con k = k_i contenuta nel nodo x

if trovata **then** il record è già presente (no inserimento)

else # inserisci una nuova entry per k

crea una entry (k,Pr) con Pr puntatore al nuovo record

if x non è pieno

then inserisci (k,Pr) in x nella posizione corretta

else # il nodo foglia x contiene p_{leaf} puntatori

copia x in un nodo temporaneo temp che può contenere un'entry in più dei nodi del B^+-albero

Algoritmo di inserimento - 3

inserisci l'entry (k,Pr) in temp nella posizione corretta

new ← un nuovo nodo foglia vuoto

$new.P_{next} \leftarrow x.P_{next}$

$j \leftarrow \lceil (p_{leaf} + 1)/2 \rceil$

x ← prime j entry in temp

$x.\ P_{next} \leftarrow new$

new ← le restanti entry in temp

$k \leftarrow k_j$

la coppia (K,new) va spostata nel nodo genitore;

se questo è pieno, va propagato lo split

Algoritmo di inserimento - 4

finito ← falso

repeat

if S è vuoto **then** #manca genitore

radice ← nuovo nodo interno vuoto

radice ← <x,k,new>

finito ← vero

else x ← pop(S)

if x non è pieno **then**

inserisci (k,new) in x nella posizione corretta

finito ← vero

Algoritmo di inserimento - 5

else # il nodo interno x è pieno (contiene p puntatori)

copia x in temp

inserisci (k,new) in temp nella posizione corretta

new ← un nuovo nodo interno vuoto

$j \leftarrow \lfloor (p + 1)/2 \rfloor$

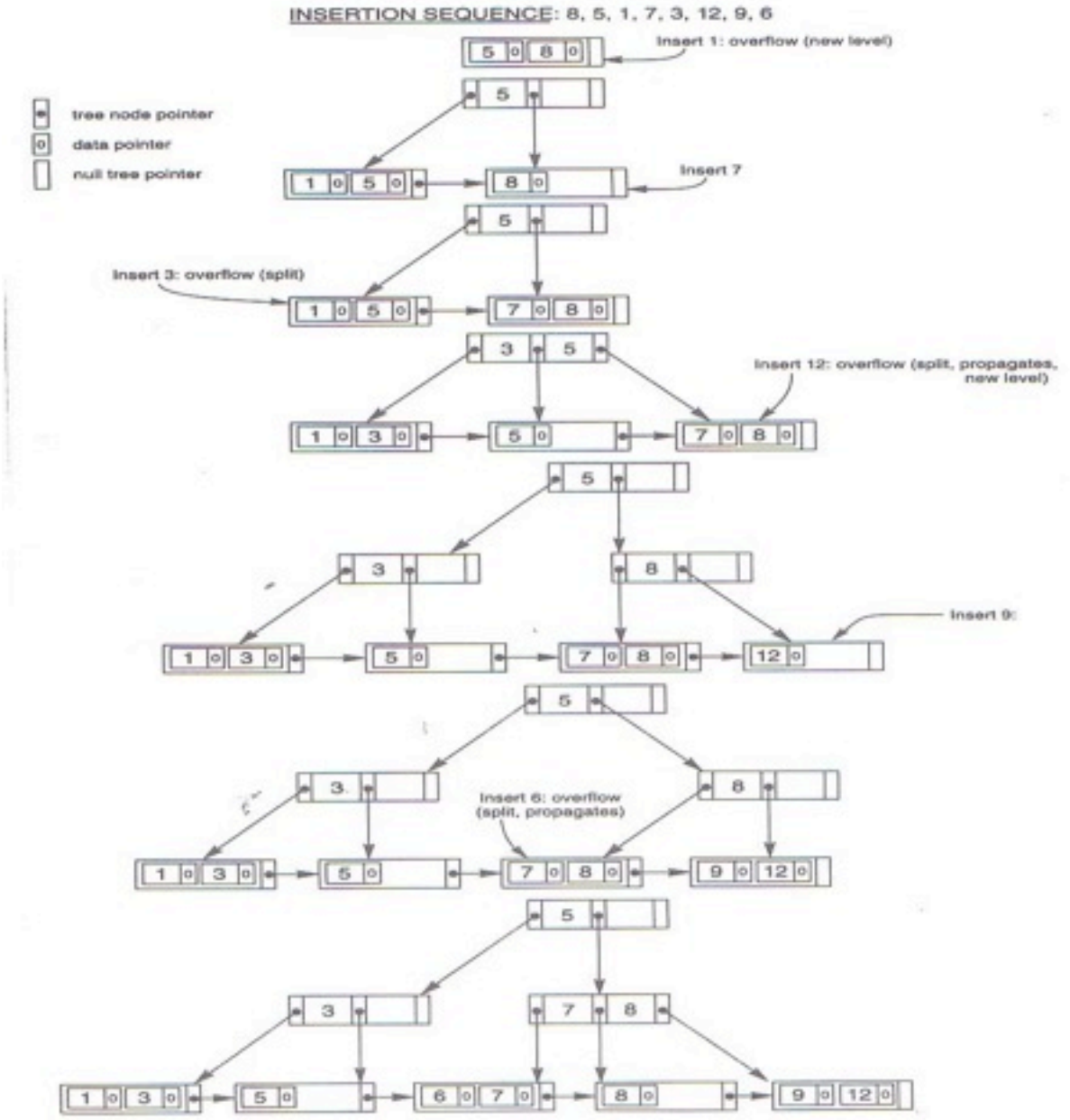
x ← le entry fino a P_j in temp

new ← le entry dopo P_{j+1} in temp

$k \leftarrow k_j$

until finito

Esempio inserimento (p=3 e $p_{leaf} = 2$)



Cancellazione in un B^+-albero - 1

- Caso 1. foglia ok e indice ok
- cancellare il record dalla foglia e riorganizzare la foglia
- se la chiave compare nell'indice, sostituirla con quella che la precede (nell'ipotesi che nella posizione i-esima di un nodo indice venga riportata la chiave più grande fra le chiavi appartenenti al sottoalbero raggiungibile attraverso il puntatore i-esimo)

Cancellazione in un B^+-albero - 2

- Caso 2. foglia troppo vuota e indice ok
- concatenare/bilanciare la foglia con un fratello
- sistemare il nodo genitore per riflettere la variazione a livello delle foglie (se il numero di foglie si riduce, vi sarà una chiave in meno nel genitore)

Cancellazione in un B^+-albero - 3

- Caso 3. foglia e indice troppo vuoti
- concatenare/bilanciare la foglia con un fratello (senza far scendere indice dal genitore)
- sistemare il nodo genitore indice per riflettere il cambiamento
- concatenare/bilanciare il nodo indice con un fratello
- continuare a concatenare/bilanciare se necessario

Esempio cancellazione (p=3 e $p_{leaf} = 2$)