Strutture dati avanzate

Vedremo alcune strutture dati che permettono di eseguire in modo particolarmente efficiente un determinato insieme di operazioni.

Ognuna di tali strutture permetterà di eseguire in modo efficiente operazioni quali ***Insert***, ***Delete***, ***Search***, ***Minimum***, ***ExtractMin***, ***Union***, ecc.

B-alberi

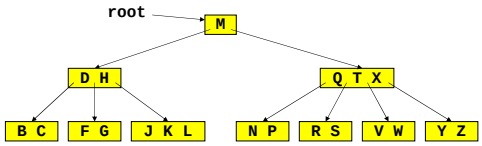
I ***B***-alberi sono alberi bilanciati particolarmente adatti per memorizzare grandi quantità di dati in memoria secondaria (su disco).

Sono simili agli alberi rosso-neri ma sono progettati in modo da minimizzare il numero di accessi a disco.

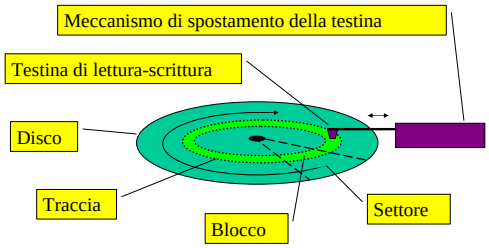
Le operazioni fondamentali sui ***B***-alberi sono ***Insert***, ***Delete*** e ***Search***.

A differenza dei nodi degli alberi rosso-neri che contengono una sola chiave ed hanno due figli, i nodi dei ***B***-alberi possono contenere un numero chiavi $n \geq 1$ ed $n+1$ figli.

Ecco un esempio di B-albero.



Il funzionamento di una unità disco magnetico può essere schematizzato nel seguente modo:



L'unità elementare di scrittura-lettura è il *blocco* (512 byte – 2 Kbyte).
Il S.O. gestisce una memoria virtuale organizzata in pagine di dimensione tra 2 e 10 kbyte.

L'accesso alle informazioni registrate su disco richiede alcune operazioni meccaniche:
- spostamento della testina (seek) e
- rotazione del disco (rotation latency).

Per questa ragione il *tempo di accesso*, il tempo richiesto per posizionare la testina di lettura scrittura e aspettare che il blocco/la pagina passi sotto la testina, può essere dell'ordine dei millisecondi.
Tipicamente 5-12 msec.

Il tempo di accesso alla memoria centrale, non richiedendo operazioni meccaniche ed è invece dell'ordine dei nanosecondi.
Tipicamente 10-100 nsec.

Spesso occorre più tempo per leggere i dati da disco in memoria centrale e per scrivere i risultati su disco di quanto serva per la loro elaborazione in memoria centrale.

Per questo, nel valutare la complessità delle operazioni, terremo separate le due componenti:

1. tempo di lettura-scrittura su disco (che assumiamo proporzionale al numero di pagine lette o scritte).
2. tempo di CPU (tempo di calcolo in memoria centrale).

7

L'operazione di lettura da disco è:

***DiskRead*(*x*)**

che, dato il riferimento ***x*** ad un oggetto, legge l'oggetto da disco in memoria centrale.

Assumiamo che sia possibile accedere al valore di un oggetto soltanto dopo che esso sia stato letto in memoria centrale.

Assumiamo inoltre che se l'oggetto da leggere si trova già in memoria centrale l'operazione ***DiskRead*(*x*)** non abbia alcun effetto.

8

L'operazione di scrittura su disco è :

***DiskWrite*(*x*)**

che, dato il riferimento ***x*** ad un oggetto presente in memoria centrale, scrive tale oggetto su disco.

Tipicamente l'elaborazione di un oggetto residente su disco segue lo schema:

***DiskRead*(*x*)**	
.....	**⊳ elaborazioni che accedono/modificano *x*.**
***DiskWrite*(*x*)**	**⊳ non necessaria se *x* resta invariato**
.....	**⊳ elaborazioni che non modificano *x*.**

9

Le operazioni di lettura/scrittura devono leggere e scrivere almeno una pagina e questo anche se l'oggetto ***x*** da leggere o scrivere è molto piccolo.

Siccome, con i ***B***-alberi, la maggior parte del tempo è utilizzata per le operazione di lettura e scrittura da disco è opportuno sfruttare al massimo ciascuna di tali operazioni.

10

Per questa ragione un nodo è generalmente grande quanto una pagina.

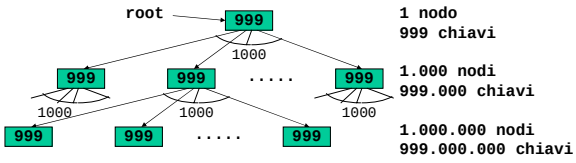
Il numero massimo di figli di un nodo è quindi limitato dalla dimensione di una pagina.

Un numero massimo di figli compreso tra 50 e 2000 è la norma.

11

Un elevato grado di diramazione riduce in modo drastico sia l'altezza dell'albero che il numero di letture da disco necessarie per cercare una chiave.

La figura seguente mostra che un ***B***-albero di altezza 2 con un grado di diramazione 1000 può contenere 10^9-1 chiavi (un miliardo).



12

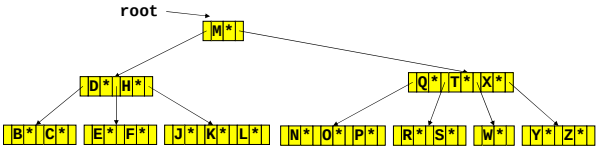
Poiché la radice può essere tenuta stabilmente in memoria, due accessi a disco sono sufficienti per cercare una qualsiasi chiave nella struttura.

Per semplicità di esposizione assumeremo che le informazioni "satellite" siano memorizzate assieme alle chiavi in tutti i nodi dell'albero.

Un'altra possibilità è mettere tali informazioni nelle foglie e mettere nei nodi interni soltanto chiavi e puntatori ai figli, massimizzando quindi il grado di diramazione dei nodi interni (***B*⁺*-alberi***)

13

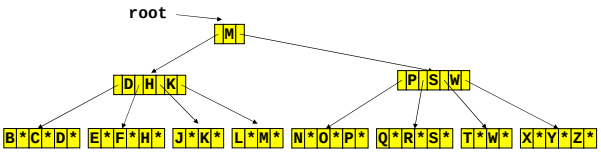
Esempio di ***B***-albero con grado massimo di diramazione 4 che usa la prima tecnica.



Un nodo deve poter contenere 3 chiavi, 4 puntatori ai figli e 3 puntatori alle informazioni associate: 10 campi in totale.

14

Esempio di ***B***-albero con grado massimo di diramazione 4 che usa la seconda tecnica.



Un nodo interno deve poter contenere 3 chiavi e 4 puntatori ai figli; una foglia deve poter contenere 3 chiavi e 3 puntatori alle informazioni associate: al massimo 7 campi in totale.

15

Definizione di *B*-albero

Un ***B***-albero ***T*** è un albero di radice ***root*[*T*]** tale che:

1. Ogni nodo ***x*** contiene i seguenti campi:
   a) ***n*[*x*]**: il numero di chiavi presenti nel nodo
   b) $key_1[x] \le key_2[x] \le .... \le key_{n[x]}[x]$: le ***n*[*x*]** chiavi in ordine non decrescente
   c) ***leaf* [*x*]**: valore booleano che è ***true*** se il nodo ***x*** è foglia, ***false*** altrimenti;

16

2. se il nodo non è una foglia contiene anche
   d) $c_1[x], c_2[x],...., c_{n[x]+1}[x]$: gli ***n*[*x*]+1** puntatori ai figli;
3. le chiavi $key_1[x], key_2[x],...., key_{n[x]}[x]$ di un nodo interno separano le chiavi dei sottoalberi. In altre parole, se k_i è una chiave qualsiasi del sottoalbero di radice $c_i[x]$ allora $k_1 \le key_1[x] \le k_2 \le ... \le k_{n[x]} \le key_{n[x]}[x] \le k_{n[x]+1}$.
4. Le foglie sono tutte alla stessa profondità ***h*** detta altezza dell'albero.

17

5. Vi sono limiti inferiori e superiori al numero di chiavi in un nodo e tali limiti dipendono da una costante ***t*** detta *grado minimo* del ***B***-albero.
   a) ogni nodo, eccetto la radice, ha almeno ***t* -1** chiavi e, se non è una foglia, almeno ***t*** figli.
   b) se l'albero non è vuoto, la radice contiene almeno una chiave e, se non è foglia, almeno due figli.
   c) ogni nodo ha al più **2*t* -1** chiavi e, se non è foglia, al più **2*t*** figli.

18

Ad ogni chiave sono generalmente associate delle informazioni ausiliarie. Assumeremo implicitamente che quando viene copiata una chiave vengano copiate anche tali informazioni.

I ***B***-alberi più semplici sono quelli con grado minimo ***t* = 2**. Ogni nodo interno ha **2**, **3** o **4** figli. In questo caso si dicono anche **2-3-4**-alberi.

19

Esercizio 20.
Perché non possiamo avere grado minimo ***t* = *1***?

Esercizio 21.
Calcolare il numero massimo di chiavi che può contenere un ***B***-albero in funzione del suo grado minimo ***t*** e della sua altezza ***h*.**

Esercizio 22.
Dire quale struttura dati si ottiene se in ogni nodo nero di un albero rosso-nero conglobiamo i suoi eventuali figli rossi.

20

Altezza di un *B*-albero

Proprietà.
Ogni ***B***-albero di grado minimo ***t*** contenente ***N*** chiavi ha altezza $h \le \log_t \frac{N+1}{2}$ (assumendo per convenzione che l'albero vuoto abbia altezza **-1**).

Dimostrazione.
Invece della diseguaglianza $h \le \log_t \frac{N+1}{2}$

dimostriamo quella equivalente $N \ge 2t^h - 1$.

21

Se l'albero è vuoto $h = -1$ ed $N = 0 \ge 2t^{-1} - 1$.

Supponiamo quindi che l'albero non sia vuoto e sia ***root*** la sua radice ed $h \ge 0$ la sua altezza.

Sia m_i il numero di nodi a livello ***i***.
Quindi $m_0 = 1$,
$m_1 = n[root] + 1 \ge 2$
$m_i \ge t\, m_{i-1} \ge t^{i-1} m_1$ per $i > 1$
ed $m_i \ge 2t^{i-1}$ per ogni $i \ge 1$.

22

Quindi
$$\begin{aligned} N &= \sum_x n[x] \\ &= n[root] + \sum_{i=1}^{h} \sum_{x \text{ di livello } i} n[x] \\ &\ge 1 + \sum_{i=1}^{h} 2t^{i-1}(t-1) \\ &= 1 + 2(t-1)\frac{t^h - 1}{t-1} \\ &\ge 1 + 2(t^h - 1) = 2t^h - 1 \end{aligned}$$

L'altezza di un ***B***-albero è $O(\log_t n)$ dello stesso ordine $O(\log_2 n)$ degli alberi rosso-neri, ma la costante nascosta nella ***O*** è inferiore di un fattore $\log_2 t$ che, per $50 \le t \le 2000$, è compreso tra ***5*** e ***11***.

23

Operazioni elementari

Adottiamo le seguenti convenzioni per le operazioni sui ***B***-alberi:

1. La radice del ***B***-albero è sempre in memoria.
2. I nodi passati come parametri alle procedure sono stati preventivamente caricati in memoria.

24

Defineremo le seguenti operazioni:
- ***BTree*** costruttore di un albero vuoto;
- ***Search*** che cerca una chiave nell'albero;
- ***Insert*** che aggiunge una chiave;
- ***Delete*** che toglie una chiave.

ci serviranno anche tre procedure ausiliarie:
- ***SearchSubtree***
- ***SplitChild***
- ***InsertNonfull***

25

Un ***B***-albero vuoto si costruisce con la procedura:

```
BTree(T)
  root[T] ← nil
```

la cui complessità è ***O(1).***

La procedura di ricerca in un ***B***-albero è:

```
Search(T, k)
  if root[T] = nil then return nil
  else return SearchSubtree(root[T], k)
```

che si limita a richiamare la procedura ausiliaria ***SearchSubtree***.

26

La procedura ausiliaria ***SearchSubtree*** è:

```
SearchSubtree(x, k)
  i ← Locate(x, k)   ▷ Ricerca binaria
                     ▷ key_{1..i-1}[x] < k ≤ key_{i..n[x]}[x]
  if i ≤ n[x] and k = key_i[x] return (x, i)
  if leaf[x] return nil
  DiskRead(c_i[x])
  return SearchSubtree(c_i[x], k)
```

27

Dove ***Locate*** esegue una ricerca binaria:

```
Locate(x, k)
  i ← 1, j ← n[x]+1
  ▷ INVARIANTE: key_{1..i-1}[x] < k ≤ key_{j..n[x]}[x]
  while i < j do   ▷ Ricerca binaria
    if k ≤ key_{⌊(i+j)/2⌋}[x] then j ← ⌊(i+j)/2⌋
    else i ← ⌊(i+j)/2⌋ +1
  ▷ key_{1..i-1}[x] < k ≤ key_{i..n[x]}[x]
  return i
```

28

Il numero di ***DiskRead*** è al più uguale all'altezza ***h*** dell'albero ed è quindi $O(\log_t N)$.

Il tempo di CPU di ***Search*** è:

$$\begin{aligned} T &\le (h+1)\log_2(2t-1) \\ &= O(\log_t N \log_2 t) \\ &= O(\log_2 N) \end{aligned}$$

29

Vediamo ora la ***Insert*** che aggiunge una chiave.

Non possiamo aggiungere la chiave ad un nodo interno perché dovremmo aggiungere anche un sottoalbero.

Quindi l'aggiunta di una chiave in un ***B***-albero può avvenire soltanto in una foglia e soltanto se la foglia non è piena (ossia non ha già il numero massimo $2t-1$ di chiavi).

30

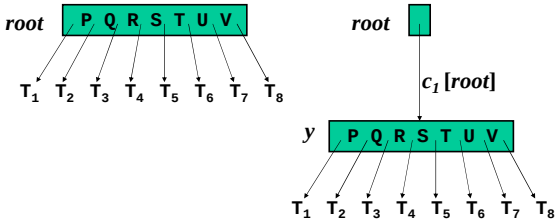
Possiamo garantirci che la foglia a cui arriveremo non sia piena se durante la discesa dalla radice a tale foglia ci assicuriamo ad ogni passo che il figlio su cui scendiamo non sia pieno.

Nel caso in cui tale figlio sia pieno chiamiamo prima una particolare funzione ***SplitChild*** che lo divide in due parti.

La stessa cosa dobbiamo fare all'inizio se la radice è piena.

31

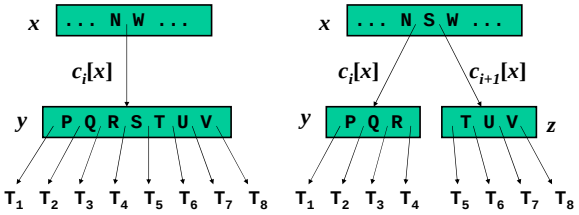
Ecco come funziona ***Insert*** se la radice è piena in un B-albero di grado minimo ***t = 4*** :



dopo di che richiama ***SplitChild***.

32

Ecco come funziona ***SplitChild*** su un figlio pieno in un ***B***-albero di grado minimo ***t = 4*** :



33

La funzione ausiliaria ***SplitChild*** è:

```
SplitChild(x, i, y)
  ▷ PRE: y è figlio i-esimo di x ed è pieno (e x non pieno)
  z ← AllocateNode()
  leaf[z] ← leaf[y]
  ▷ Sposta le ultime t -1 chiavi di y in z
  for j ← 1 to t -1 do
     key_j[z] ← key_{j+t}[y]
  ▷ Se y non è foglia sposta gli ultimi t puntatori di y in z
  if not leaf[y] then
    for j ← 1 to t do
       c_j[z] ← c_{j+t}[y]
  n[z] ← t -1
```

34

```
  ▷ Sposta la t-esima chiave di y in x
for j ← n[x] downto i do
   key_{j+1}[x] ← key_j[x]
for j ← n[x]+1 downto i+1 do
   c_{j+1}[x] ← c_j[x]
key_i[x] ← key_t[y]
c_{i+1}[x] ← z
n[x] ← n[x] +1
▷ Rimuove le ultime t chiavi da y
n[y] ← t -1
▷ Scrive su disco i nodi modificati
DiskWrite(x), DiskWrite(y), DiskWrite(z)
```

35

La procedura di inserzione in un ***B***-albero è:

```
Insert(T,k)
  if root[T] = nil then
     x ← AllocateNode()
     n[x] ← 1, key_1[x] ← k, leaf[x] ← true
     root[T] ← x
  else
     if n[root[T]] = 2t –1 then
        y ← root[T]
        x ← AllocateNode()
        n[x] ← 0, c_1[x] ← y, leaf[x] ← false
        root[T] ← x
        SplitChild(x,1,y)
     InsertNonfull(root[T],k)
```

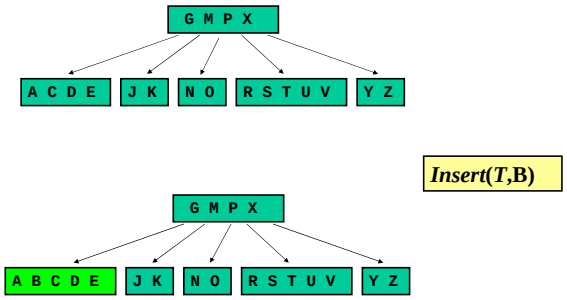
La funzione ausiliaria ***InsertNonfull*** è:

```
InsertNonfull(x, k)
  ▷ PRE: x non è pieno
  if leaf[x] then      ▷ Inserisce la chiave k nel nodo x
     i ← n[x]+1
     while i > 1 and key_{i-1}[x] > k do
        key_i[x] ← key_{i-1}[x], i ← i -1
     key_i[x] ← k
     n[x] ← n[x]+1
     DiskWrite(x)
```

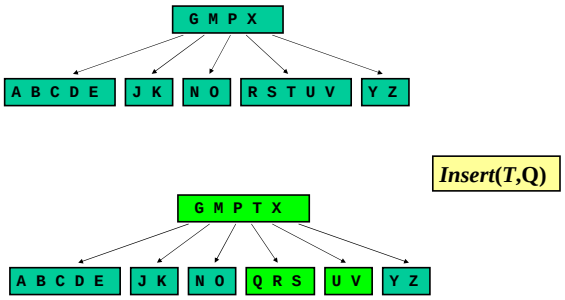
37

```
else                          ▷ Cerca il figlio in cui inserirla
   i ← Locate(x, k)           ▷ Ricerca binaria
   ▷ INVARIANTE: key_{1..i-1}[x] < k ≤ key_{i..n[x]}[x]
   DiskRead(c_i[x])
   if n[c_i[x]] = 2t -1 then
      SplitChild(x, i, c_i[x])
   if k > key_i[x] then i ← i +1
   InsertNonfull(c_i[x],k)
```

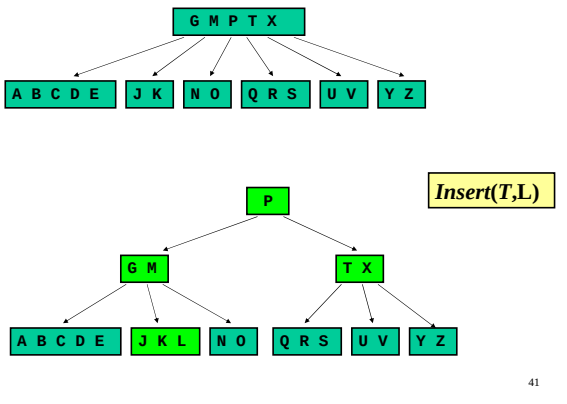
38



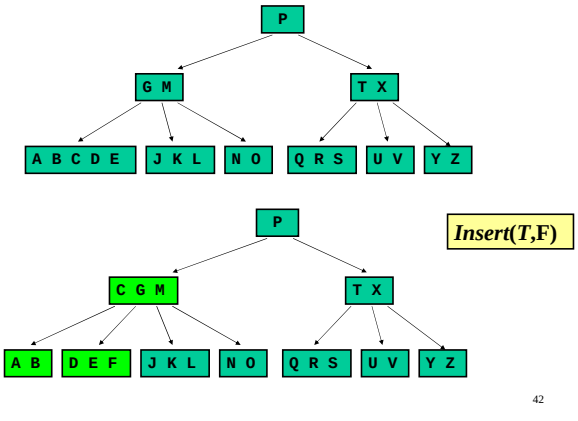
39



40



41



42

Esercizio 23.
Scrivere una funzione che cerca la chiave minima contenuta in un ***B***-albero ed una funzione che data un valore ***c*** cerca la minima chiave ***k*** > ***c*** presente nel ***B***-albero.

*Esercizio 24**.
In un ***B***-albero con grado minimo ***t = 2*** vengono inserite le chiavi ***1, 2, 3, …., n*** nell'ordine. Valutare il numero di nodi dell'albero in funzione di ***n***.

43

Delete: eliminare una chiave da un ***B***-albero.

Possiamo eliminare una chiave solo
1. da una foglia
2. solo se questa non ha il minimo numero di chiavi (***t*** **-1**) oppure è radice del ***B***-albero.

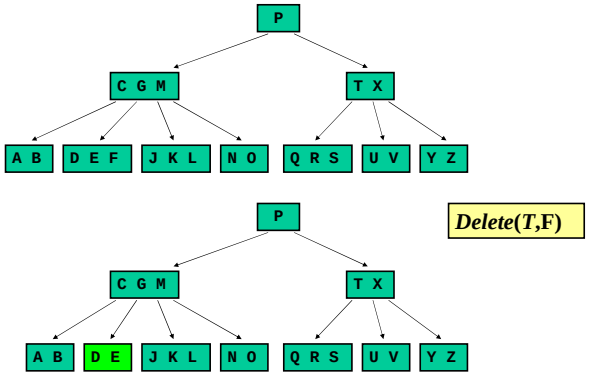
1. Se dobbiamo eliminare la chiave $k = key_i[x]$ da un nodo ***x*** non foglia, scambiamo prima la chiave ***k*** con la massima chiave ***k'*** del sottoalbero di radice $c_i[x]$.

44

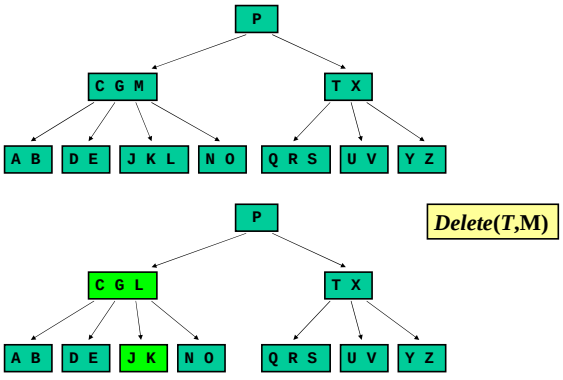
2. Scendendo dalla radice per cercare la chiave ***k*** da togliere ci dobbiamo anche assicurare che i nodi su cui ci spostiamo non abbiano mai il numero di chiavi minimo ***t*****-1**.

Per questo, se il nodo figlio su cui dobbiamo scendere ha solo ***t*****-1** chiavi prima di scendere prendiamo una chiave da uno dei suoi fratelli adiacenti o, se questo non è possibile, riuniamo tale figlio con uno dei fratelli.

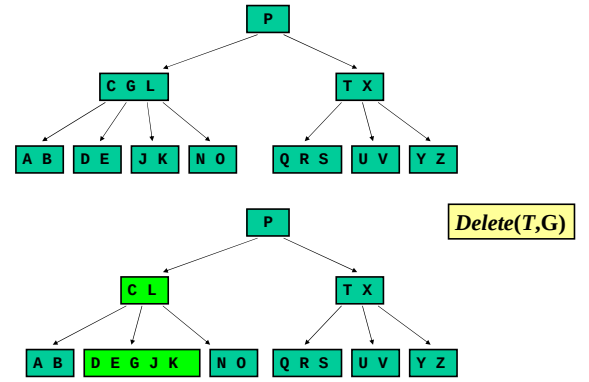
45



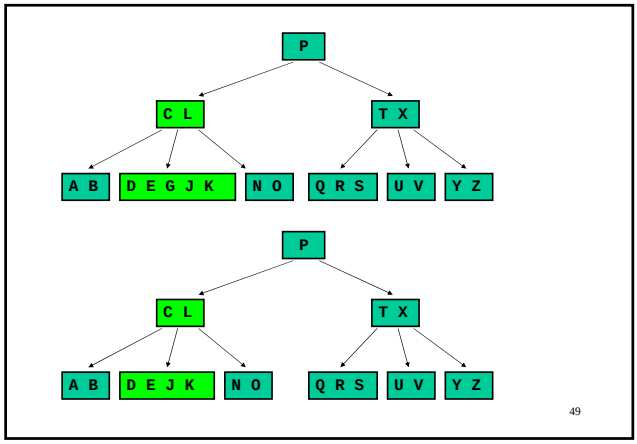
46

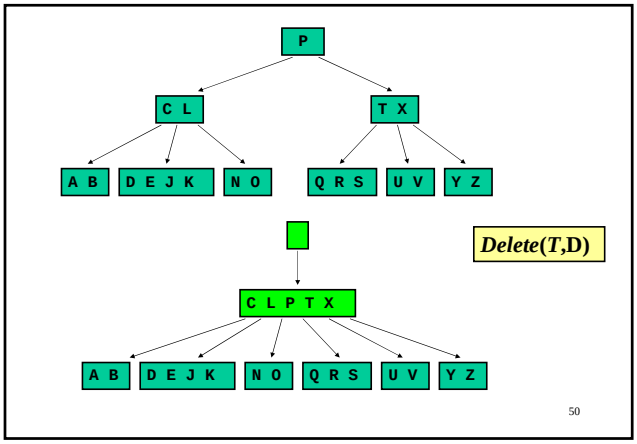


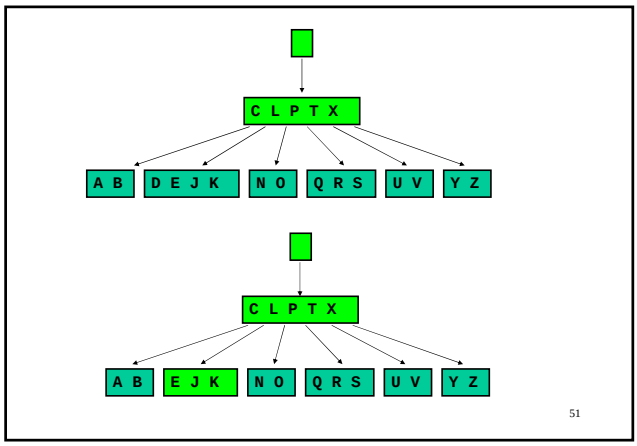
47

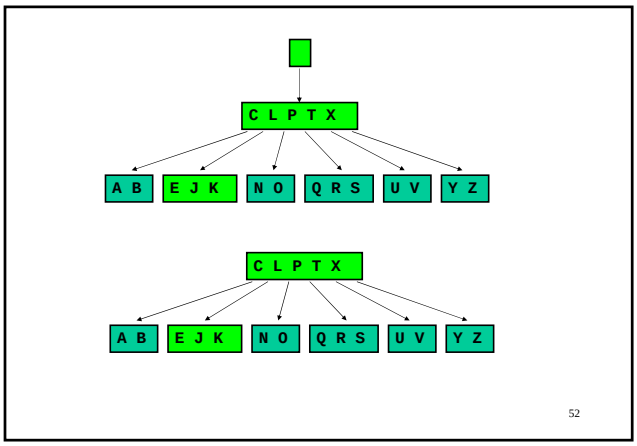


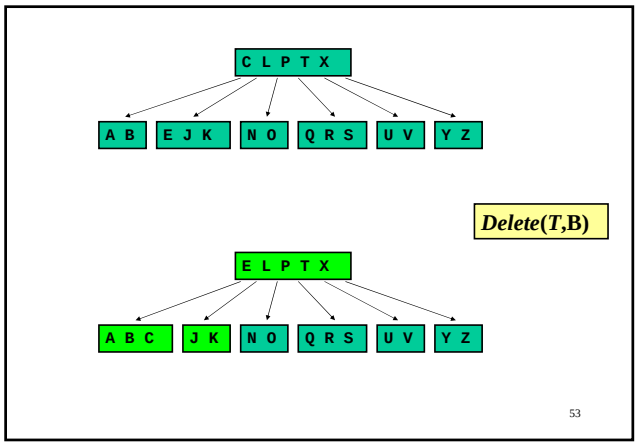
48

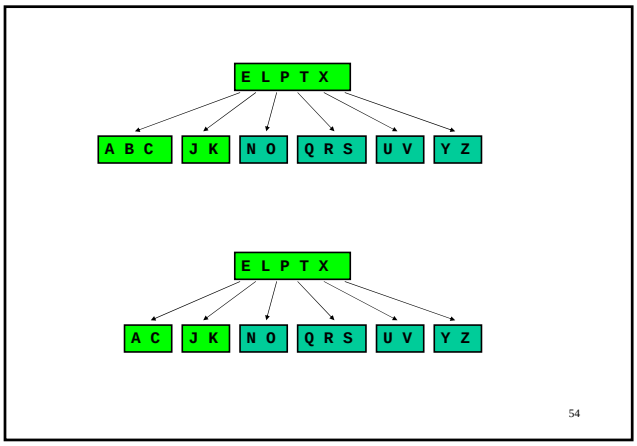

P
C L
T X
A B
D E G J K
N O
Q R S
U V
Y Z
P
C L
T X
A B
D E J K
N O
Q R S
U V
Y Z
49


P
C L
T X
A B
D E J K
N O
Q R S
U V
Y Z
Delete(T,D)
C L P T X
A B
D E J K
N O
Q R S
U V
Y Z
50


C L P T X
A B
D E J K
N O
Q R S
U V
Y Z
C L P T X
A B
E J K
N O
Q R S
U V
Y Z
51


C L P T X
A B
E J K
N O
Q R S
U V
Y Z
C L P T X
A B
E J K
N O
Q R S
U V
Y Z
52


C L P T X
A B
E J K
N O
Q R S
U V
Y Z
Delete(T,B)
E L P T X
A B C
J K
N O
Q R S
U V
Y Z
53


E L P T X
A B C
J K
N O
Q R S
U V
Y Z
E L P T X
A C
J K
N O
Q R S
U V
Y Z
54

La procedura di rimozione di una chiave da un ***B***-albero è:

```
Delete(T, k)
  if root[T] ≠ nil then
     DeleteNonmin(root[T], k)
     if  n[root[T]] = 0 then
        root[T] ← c1[root[T]]
```

essa si limita a richiamare la funzione ausiliaria ***DeleteNonmin*** sulla radice dell'albero dopo essersi assicurata che l'albero non sia vuoto.

Se al ritorno la radice non contiene alcuna chiave essa viene rimossa.

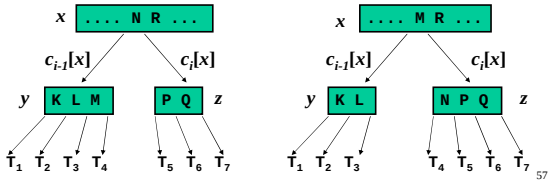
La procedura ***DeleteNonmin*** usa la procedura ***AugmentChild*** per assicurarsi di scendere sempre su di un nodo che non contiene il minimo numero di chiavi.

La procedura ***AugmentChild*** aumenta il numero di chiavi del figlio ***i***-esimo prendendo una chiave da uno dei fratelli adiacenti.

Se i fratelli adiacenti hanno tutti il minimo numero di chiavi allora riunisce il figlio ***i***-esimo con uno dei fratelli adiacenti.

La procedura ***AugmentChild*** è la seguente:

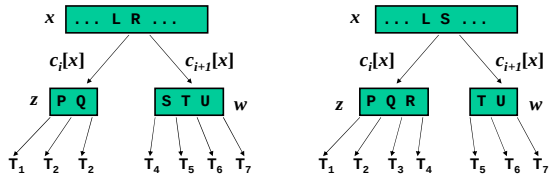
```
AugmentChild(x, i, z)
  ▷ PRE: z è il figlio i-esimo di x ed ha soltanto t -1 chiavi.
  if i > 1 then                          ▷ z ha un fratello alla sua sinistra
     y ← ci-1[x], DiskRead(y)
  if i > 1 and n[y] > t-1 then           ▷ si può prendere una chiave
                                         ▷ dal fratello sinistro
```




```
  for j ← n[z]+1 downto 2 do  ▷ Sposta avanti le chiavi in z
     keyj[z] ← keyj-1[z]
  key1[z] ← keyi-1[x]          ▷ Rotazione delle chiavi
  keyi-1[x] ← keyn[y][y]
  if not leaf[z] then          ▷ Sposta anche i puntatori in z
     for j ← n[z]+2 downto 2 do
        cj[z] ← cj-1[z]
     c1[z] ← cn[y]+1[y]
  n[z] ← n[z]+1, n[y] ← n[y]-1
  DiskWrite(x), DiskWrite(y), DiskWrite(z)
```

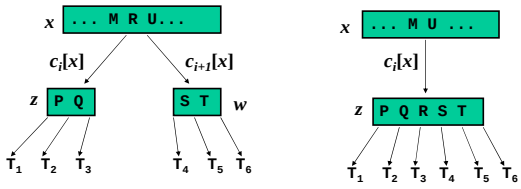


```
else                           ▷ i = 1 o il fratello sinistro y di z ha solo t -1 chiavi
  if i ≤ n[x] then             ▷ z ha un fratello alla sua destra
     w ← ci+1[x], DiskRead(w)
  if i ≤ n[x] and n[w] > t -1 then  ▷ si può prendere una
                                    ▷ chiave dal fratello destro
```




```
  keyn[z]+1[z] ← keyi[x]   ▷ Rotazione delle chiavi
  keyi[x] ← key1[w]
  for j ← 2 to n[w] do     ▷ Sposta indietro le chiavi in w
     keyj-1[w] ← keyj[w]
  if not leaf[w] then      ▷ Sposta anche i puntatori in w
     cn[z]+2[z] ← c1[w]
     for j ← 2 to n[w]+1 do
        cj-1[w] ← cj[w]
  n[z] ← n[z]+1, n[w] ← n[w]-1
  DiskWrite(x), DiskWrite(y), DiskWrite(z)
```

```
else   ▷ i = 1 oppure y ha solo t -1 chiavi e
       ▷ i = n[x]+1 oppure w ha solo t -1 chiavi
  if i ≤ n[x] then   ▷ z ha il fratello destro w con t -1 chiavi.
                     ▷ Possiamo riunire z, key_i[x] e w
```

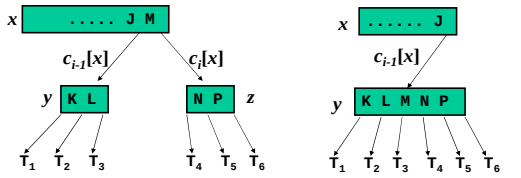


61

```
key_t[z] ← key_i[x]          ▷ Aggiunge key_i[x] a z
for j ← i +1 to n[x] do      ▷ Sposta indietro le chiavi in x
  key_{j-1}[x] ← key_j[x]
for j ← i +2 to n[x]+1 do    ▷ Sposta indietro i puntatori in x
  c_{j-1}[x] ← c_j[x]
n[x] ← n[x]-1
for j ← 1 to n[w] do         ▷ Sposta le chiavi di w in z
  key_{j+t}[z] ← key_j[w]
if not leaf[w] then          ▷ Sposta anche i puntatori di w in z
  for j ← 1 to n[w]+1 do
    c_{t+j}[z] ← c_j[w]
n[z] ← 2t -1
DiskWrite(x), DiskWrite(z)
```

62

```
else   ▷ i = n[x]+1 > 1 e quindi il fratello sinistro y ha
       ▷ t -1 chiavi. Possiamo riunire y, key_{i-1}[x] e z
```



63

```
key_t[y] ← key_{i-1}[x]      ▷ Aggiunge key_{i-1}[x] a y
n[x] ← n[x]-1
for j ← 1 to n[z] do         ▷ Sposta le chiavi di z in y
  key_{j+t}[y] ← key_j[z]
if not leaf[y] then          ▷ Sposta anche i puntatori di z in y
  for j ← 1 to n[z]+1 do
    c_{t+j}[y] ← c_j[z]
n[y] ← 2t -1
DiskWrite(x), DiskWrite(y)
```

64

La procedura ***DeleteNonmin*** è quindi:

```
DeleteNonmin(x, k)   ▷ PRE: x ha almeno t chiavi.
  i ← 1, j ← n[x]+1
  ▷ INVARIANTE: key_{1..i-1}[x] < k ≤ key_{j..n[x]}[x]
  while i < j do     ▷ Ricerca binaria
    if k ≤ key_{⌊(i+j)/2⌋}[x] then j ← ⌊(i+j)/2⌋
    else i ← ⌊(i+j)/2⌋ +1
  if i ≤ n[x] and k = key_i[x] then   ▷ Trovata k in x
    if leaf[x] then                   ▷ x è una foglia, posso togliere k
      for j ← i+1 to n[x] do          ▷ Sposta indietro le chiavi in x
        key_{j-1}[x] ← key_j[x]
      n[x] ← n[x]-1
      DiskWrite(x)
```

65

```
else       ▷ Trovata k in x ma x non è una foglia
  DiskRead(c_i[x])
  if n[c_i[x]] = t -1 then
    AugmentChild(x, i, c_i[x])
    if i = n[x]+2 then
      ▷ AugmentChild ha riunito c_{i-1}[x], key_{i-1}[x] e c_i[x] in c_{i-1}[x]
      i ← i - 1
  if key_i[x] ≠ k then
    ▷ AugmentChild ha spostato k nel figlio i-esimo c_i[x]
    DeleteNonmin(c_i[x],k)
  else     ▷ è rimasto k = key_i[x]
    ▷ Sposta la massima chiave del sottoalbero c_i[x] in key_i[x]
    DeleteMax(x,i,c_i[x])
```

66

```
else    ▷ k non è in x. Può stare nel sottoalbero i-esimo
   DiskRead(c_i[x])
   if n[c_i[x]] = t -1 then
      AugmentChild(x,i, c_i[x])
      if i = n[x]+2 then
         ▷ AugmentChild ha riunito c_{i-1}[x], key_{i-1}[x] e c_i[x] in c_{i-1}[x]
         i ← i - 1
   DeleteNonmin(c_i[x],k)
```


La procedura ***DeleteMax*** è:

```
DeleteMax(x, i, y)
   ▷ Sposta la massima chiave del sottoalbero di radice y in key_i[x]
   if leaf[y] then
      key_i[x] ← key_{n[y]}[y]
      n[y] ← n[y]-1
      DiskWrite(x), DiskWrite(y)
   else
      z ← c_{n[y]+1}[y]
      DiskRead(z)
      if n[z] = t -1 then
         AugmentChild(y, n[y]+1, z)
      DeleteMax(x, i, c_{n[y]+1}[y])
```


La procedura ***AugmentChild*** richiede al più ***5*** accessi a disco (due ***DiskRead*** e tre ***DiskWrite***) e tempo di CPU ***O(1)*** (considerando ***t*** costante).

Se ***y*** è radice di un sottoalbero di altezza ***h*** allora ***DeleteMax(x, i, y)*** nel caso pessimo effettua ***5h*** accessi a disco e richiede tempo di CPU ***O(h)***.

Se ***x*** è radice di un sottoalbero di altezza ***h*** allora ***DeleteNonmin(x, k)*** nel caso pessimo effettua ***6h*** accessi a disco e richiede tempo di CPU ***O(h)***.

Esercizio 26.

Mostrare come sia possibile aggiungere ad ogni nodo ***x*** di un ***B***-albero un campo ***height*** che contiene l'altezza del sottoalbero di radice ***x***.

Dire quali sono le modifiche da apportare a ***Insert*** e ***Delete***.

Assicurarsi che la complessità asintotica non aumenti.