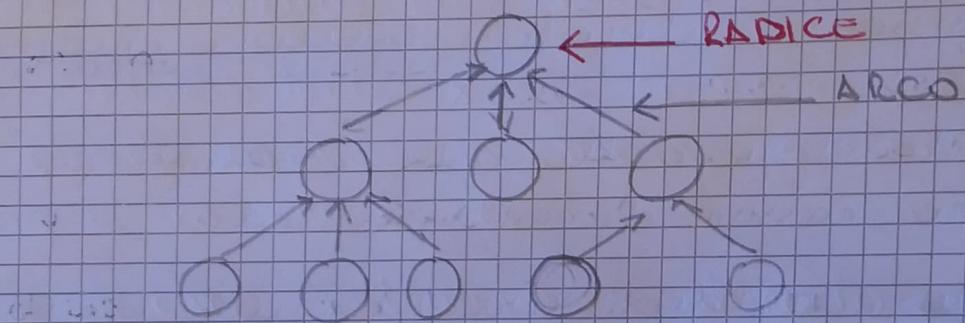


Alberi Radicati

Un albero radicato è un insieme di nodi su cui è definita una relazione binaria "x è figlio di y" o "y è genitore di x" tale che:

- 1) ogni nodo ha un solo genitore con l'eccezione della radice che non ha genitore
- 2) c'è un comune antenato di ogni nodo alla radice \Rightarrow l'albero è连通的 (connesso)



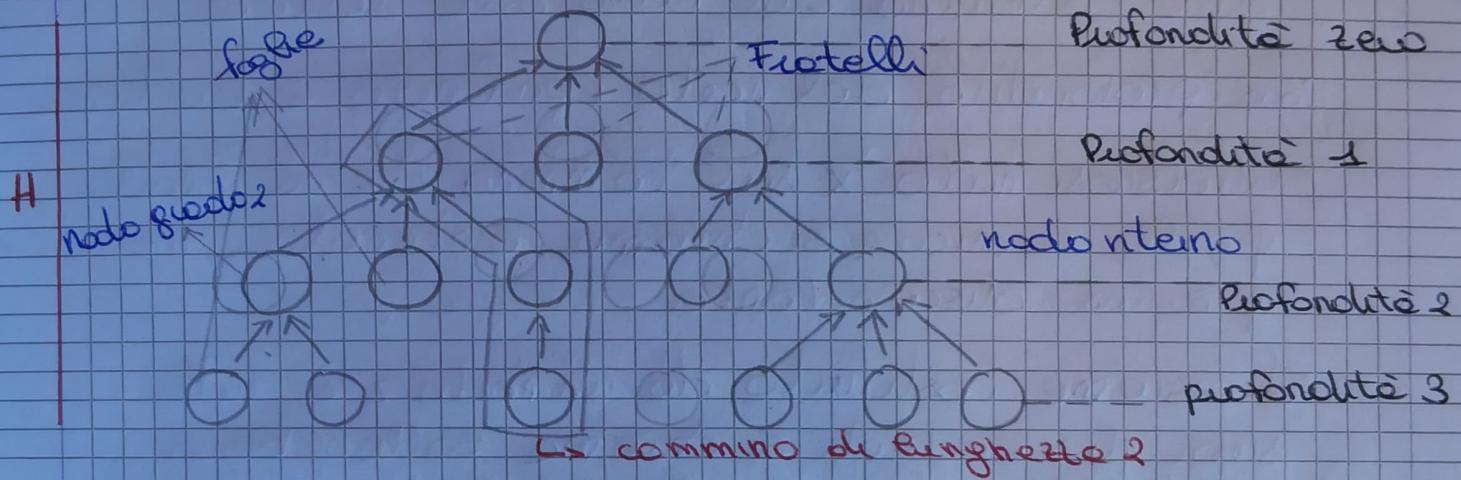
! In un albero non radicato la relazione che intercorre tra 2 nodi è bivoca e non sappiamo chi è la radice

Si può costruire un albero partendo dalla radice e aggiungendo di volta in volta un nodo y figlio di un nodo x.

\Rightarrow Se ho un albero con n nodi allora ho $n-1$ relazioni

Definizioni

- Due nodi che hanno lo stesso genitore si chiamano fratelli
- Il numero di figli di un nodo è il suo grado
- I nodi di grado zero sono foglie
- Un nodo non foglie è detto interno



- per essere foglie occorre non avere figli
- un albero con la sola radice, la stessa radice è una foglie

• Alberi binari

Può avere solo 2 figli: il destro e il sinistro

- l'ordine dei figli è significativo (ordinati)
- si considera come se sx precede dx

• Alberi di grado arbitrario

- non sappiamo il numero di figli di un nodo
→ non conta l'ordine

- Una sequenza di nodi tali che uno è il genitore del successivo si chiama **cammino**
- Il cammino percorre gli archi di connes.
- Il numero di archi del cammino è la sua lunghezza
- La **profondità** di un nodo è la lunghezza del cammino dal nodo allo radice
- La profondità del nodo + profondo è l'altezza
- Qualsiasi nodo x sul cammino che va oltre y allo radice si chiama **antenato**, mentre y è un **discendente**
- L'insieme costituito da un nodo z e tutti i suoi discendenti si chiama **sottoalbero** **radicato**
- Un albero **binario** è un albero ordinato in cui i nodi hanno al più ordine 2
- Un albero **binario** è completo se per ogni livello abbiamo tutti i nodi possibili
- Un albero **binario** completo di altezza H ha
 - 2^h foglie $h = \log_2(n. \text{ foglie})$
 - $2^h - 1$ nodi interni
 - $2^{h+1} - 1$ nodi

Rappresentazione di alberi binari

- Si rappresentano mediante oggetti e riferimenti come uno **albero**

Rappresentazione di alberi di grado arbitrario

- Per i figli uss i puntatore a liste
- Struttura figlio sinistro - fratello destro
La radice non ha sottoalberi dx
- Il puntatore * parent non punta al campo
puntatore del nodo precedente ma ad
un altro nodo (la radice)

Visite di alberi

Un albero può essere visitato
recursivamente con 2 opposte discipline:

- Visita preordine
 - Dopo aver visitato un nodo si visitano i suoi figli

Le operazioni vengono effettuate **TOP-DOWN**

Visita post ordine

- Dopo aver visitato i figli si visita il nodo
- Le operazioni vengono eseguite **BOTTOM-UP**

Visita simmetrica (albero binario)

- Si visita prima figlio sx, poi nodo infine figlio dx

HEAP E CODE DI PRIORITÀ

Una coda di priorità è una collezione di elementi:

- Ad ogni elemento si assegna 1 valore di priorità, che definiscono un ordinamento

Operazioni:

Si vogliono inserire efficientemente nuovi valori con priorità arbitraria

- Si vuole estrarre l'elemento con + alta probabilità

Applicazioni:

- Accesso ai processi (si assegna priorità)
- Simulazione sistemi ad eventi

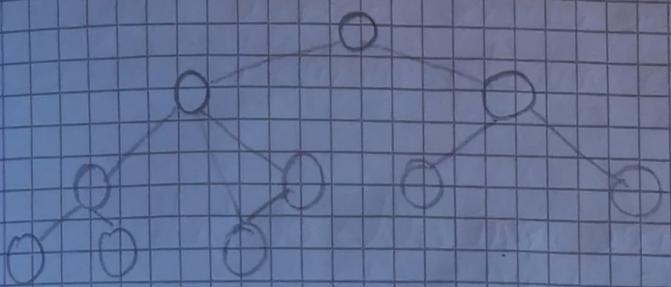
Si potrebbe realizzare tramite liste ordinate e non, tuttavia esse sono simmetriche in quanto il loro costo computazionale di un inserimento o di una ^{rimozione} ~~inserzione~~ rispettivamente ($\Theta(n)$) e ($\Theta(n)$) per l'ordinata e ($\Theta(1)$) e ($\Theta(n)$) per la non ordinata.

Nel cerchiamo una struttura con costo logaritmico ($\Theta(\log n)$) in rimozione e in aggiunta \Rightarrow Heap

→ Un heap è struttura dati che può essere usata per realizzare una coda di priorità.

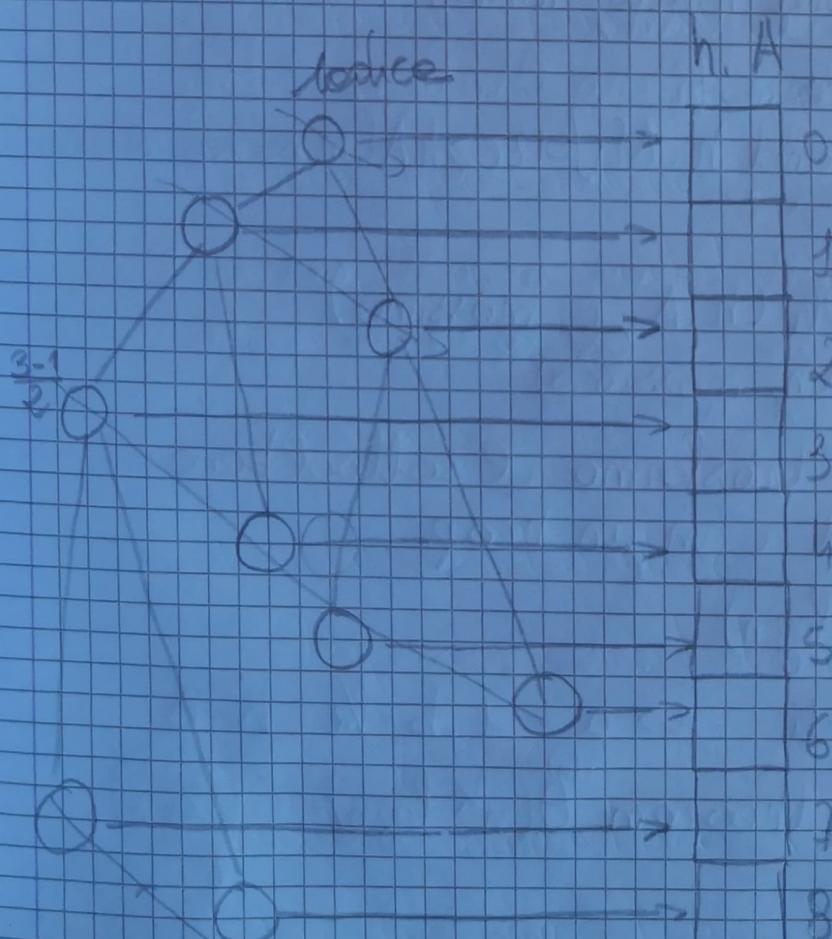
È un array i cui valori sono in relazione con le posizioni

Alberi binari quasi completi.

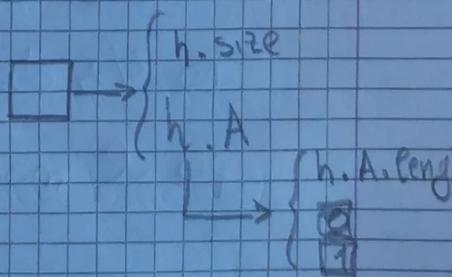


Gli heap sono rappresentati da alberi quasi completi, ossia alberi che hanno l'ultimo livello incompleto nella sua parte dx.

L'heap consiste di un array $h.A$ che codifica livello per livello, un albero binario quasi completo.



- dato il nodo
associato alla
posizione i :
- i nodi figli si trovano
in posizione $2i+1$ (sx)
e $2i+2$ (dx)
 - il nodo genitore
si trova in posizione
 $(i-1)/2$



In un max heap l'elemento memorizzato in posizione i deve avere valore maggiore o uguali a quello memorizzato nei figli

- La radice ha valore più alto di tutti.
per $j > 0$ $h.A[\text{parent}(j)] \geq h.A(j)$
- $(n/2) - 1$ nodi sono nodi interni (nei completi
ho sempre n nodi, perciò)

Max-Heapify (h, i)

L = left(i) /* indice figlio sx

R = Right(i) /* indice figlio dx

if ($l \leq h.\text{size} - 1$ and $h.A[l] > h.A[i]$) massimo = l

else massimo = i

if ($r \leq h.\text{size} - 1$ and $h.A[r] > h.A[\text{massimo}]$) massimo = r

if massimo $\neq i$

scambia coielli ($h.A, i, \text{massimo}$)

max-heapify ($h, \text{massimo}$)

funtione che trasforma il sottoalbero radicato ad i in un heap.

- costo $\Theta(\log n)$

- ho $2/3$ dei nodi $\Rightarrow a = 1, b = \frac{3}{2}, K = 0$

per il master theorem $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{K \log n}) = \Theta(\log n)$

Extract max

- estrae dall'heap il valore massimo (ossia la radice) lasciando il primo posto vuoto.
Il voto si imposta con un elemento dell'albero che anche dopo la modifica deve rimanere quasi completo, perciò l'unico valore che si può rimuovere è il nodo più a dx dell'ultimo nœud. \Rightarrow Questo potrebbe non essere il max quindi lancia max-heapify.

ha costo pari al costo di max-heapify $\Rightarrow \Theta(\log n)$

Inserimento

Incremento la dimensione di 1 e metto il nuovo elemento nella casella più a dx dell'ultimo nœud. Questo elemento potrebbe non essere in posizione corretta quindi confronto il suo valore col valore del suo genitore e sposto questi valori finché trovo genitore & vo

\rightarrow Il costo dell'algoritmo è dato dall'ottica dell'albero cioè $\Theta(\log n)$

Heap Sort

Assegno l'heap sull'array ($h.A=A$) e la lunghezza di heap è pari alla lunghezza di array.

Lancia build Max-heap (trasforma array in heap).

Poi partendo dall'ultima posizione (elim. + basso) fino all'alto (elim. ^{voltre} + alto) scambio celle avendo perciò il valore + alto è in fondo

One lenca max-heapify sub'array (esclusi
ultimo elemento) \Rightarrow che elimino.

Ritengo le procedure per tutti l'array così facendo ordino in ordine crescente l'array. (con $size = 2 \Rightarrow$ array ordinato)

Caso peggiore ha costo $\Theta(\log n)$

Levare in loco e non è stabile (scambi
anche volare uguali)

Quick Sort

• Ordinamento in fondo non stabile

• Tempo esecuzione

- Caso peggiore $\Theta(n^2)$
 - Caso migliore è medio $\Theta(n \log n) \rightarrow T(n) = \Theta(n^k \log n) = \Theta(n \log n)$

→ Divide et impera :

esistono i sottoarray rispetto ad un pivot
in modo tale che a sx lo abbiano solo numeri
minori e a dx maggiori

Alberi binari di ricerca (ABR)

Albero omogeneo: insieme di variabili omogenee accedute tramite chiavi

- la chiave è un intero, stringa o oggetto
- la chiave è unica <chiave - valore>

Lo strutturo come ABR (temp lineare)

<chiave - valore> \Rightarrow dizionario

- Definisco una relazione d'ordine fra le chiavi (\geq)

Sono realizzati con alberi radicati binari:

- hanno gli stessi valori "compi" degli alberi binari
- Sono strutturati in modo tale da avere a sx una chiave minore delle chiave x e a dx una chiave maggiore delle chiave x e deve valere per tutto il sottoalbero.
 \rightarrow parent ha valore intermedio tra sx e dx

\rightarrow Perchè devo verificare queste condizioni sia sulla radice sia sul nodo ottavo
a dx devo avere un valore maggiore del parent ma strettamente minore. Allo stesso

\rightarrow Il nodo più a sinistra termina con il valore minimo

Il nodo più a destra termina con il valore massimo

Per ricevere massimo/minimo usando while ($T \neq \text{NULL}$)
 $\rightarrow T \rightarrow dx \circ T \rightarrow sx$

Inserimento: cerco il nodo nell'albero fino a trovare il null e a posto di null ci metto il nodo

Conciliazione nodo

- Se ho uno figlio lo posso rimuovere
 - Se ha solo figlio dx/^{sx}, rimuovo il parent e lo sostituisco col figlio
- 1) • Se ha 2 figli, devo cercare il successore del nodo che devo rimuovere (cerco cioè a dx) e scelgo il più piccolo scendendolo a sx fino a NULL (non ha figlio ~~a dx~~ sx).
 - 2) Il successore ha al massimo grado 1 (può avere solo figlio dx) e lo rimuovo collegando il figlio al nodo precedente. Ora scombineremo il successore col nodo d'interesse

Complessità $\Theta(1)$ tree bypass

$\Theta(h)$ tree delete

$\Theta(h)$ tree minimum
↳ ottenere

Ricerca

```
if (x == NULL) or (K == x.Key) return x
if (K < x.Key) return ricerca ricorsiva (x.left, k)
else return ricerca ricorsiva (x.right, k)
```

Se visito simmetricamente produco valori in ordine crescente

Le visite costano sempre come il n. nodi presenti mai di meno. Se ho n oggetti, non posso avere $\Theta(\log n)$

Se $h = \log n$
Ogni step

TREE - TO - ARRAY

algoritmo che prende valori da un albero
e li copia in un array
costo $\Theta(n)$

TREE SORT $\Theta(n^2)$ $\xrightarrow{\text{peggiore}}$ $\xrightarrow{\text{migliore}}$ $\Theta(n \log n)$

Posso i dati da un array ad un obr questi
vengono ordinati e trasposti su ~~un~~ array
ordinati.

DOMANDE

In un obr bilanciato con n nodi quanto costa:

- Inserimento di un nodo
 - Cancellazione di un nodo
 - Ricerca di un nodo
- $\Theta(\log n)$ perché non visita
l'albero ma
descende a dx
o sx

In un obr sbilanciato: $\Theta(n)$

Alberi Rosso - Neri

Cercano di bilanciare l'albero (divide sempre
in 2 l'insieme di ricerca)

Si aggiunge all'obr dei valori non validi (null)
ossia nodi sentinella

Per tutti i nodi senza figli in realtà punto a
t. null

$$t \rightarrow \begin{cases} t.\text{Root} \\ t.\text{NULL} \end{cases}$$

\hookrightarrow node con campi non usati

Definizione

Un albero rosso-nero è un albero in cui:

- 1) Ogni nodo è rosso o nero
- 2) La radice t.root e le sentinelle sono nere
- 3) Se un nodo è rosso entrambi figli sono neri
- 4) tutti i cammini da t.root a t.null contengono lo stesso n. di nodi neri (t.null si prende sotto specie)

conseguenze - Altezza: cammino tra t.root et null

→ Tutti i cammini fino a T.null hanno K nodi neri

- Ogni cammino ha almeno $K-1$ archi
- Il cammino più lungo contiene nodi rossi e neri e ha $(2K-1)$ archi $\Rightarrow h$

• L'albero contiene un sottoalbero completo di profondità $h' = h/2 - 1$

• I nodi dell'albero sono almeno quelli dell'albero completo $2^{h'+1} - 1$

$$\hookrightarrow n \geq 2^{h'+1} - 1 \quad \text{con } h' = h/2 - 1$$

$$\Rightarrow n \geq 2^{(h/2-1)+1} - 1 = 2^{h/2} - 1$$

$$\Rightarrow n+1 \geq 2^{h/2} \quad \log_2(n+1) \geq \log_2(2^{h/2})$$

$$\log_2(n+1) \geq h/2 \Rightarrow \boxed{2 \log_2(n+1) \geq h}$$

max h

$O(\log n)$

$\Omega(\log n)$ per albero completo

$\Theta(\log n)$ per albero rosso nero

Poiché $\text{h} \in \Theta(\log n)$ allora tutte le funzioni impiegano $\Theta(\log n)$ per la consultazione.

Le procedure di inserimento e cancellazione non garantiscono che, una volta eseguite, l'albero sia ancora rosso-nero in quanto alcune proprietà potrebbero essere violate, perciò a seguito di tali procedure bisogna fare una funzione che ripristinino le caratteristiche, ma devono essere eseguite in $\Theta(\log n)$.

I nodi nuovi sono sempre rossi, poi sarà una funzione secondaria a ricolore i nodi.

Per ripristinare l'albero rosso-nero l'operazione di rotazione

- La rotazione cambia i colori in quanto cambia la posizione dei nodi intorno

- Il colore sempre un abu

- Esegue in $\Theta(1)$

- L'insieme dei sottoalberi è uguale

- I cammini sono uguali (verso f^{null})

Altro invece si allungano / si riducono per effetto delle rotazioni

Per ripulire l'albero rosso?

- Se inserisco un nodo rosso in un albero vuoto è sufficiente colorarlo di nero per mantenere valide la 2^ proprietà
- Altamente potrei violare la regola 3
Se ho inserito un nodo rosso ad un gen. rosso allora:

Vediamo il fratello del nodo genitore (lo zio, che esiste sempre):

Caso 1: nuovo nodo è figlio sinistro e lo zio è nero e figlio dx

dx (caso 1)

Caso 2: nuovo nodo è figlio dx e lo zio è nero e figlio dx

Caso 3: lo zio di nuovo è rosso

Caso 1:

• ncolo new.parent (nonno) se poi faccio una rotazione a dx {sopra si mantiene}

Caso 2:

• rotazione dx e muovo i nodi (unomino i nodi)

• ora riapplico il caso 1

Caso 3: (ogni)

• ncolo parente, zio è figlio nuovo → ncolo ph la rotazione solo
• potrebbe violare la 3. regola

↳ hmot