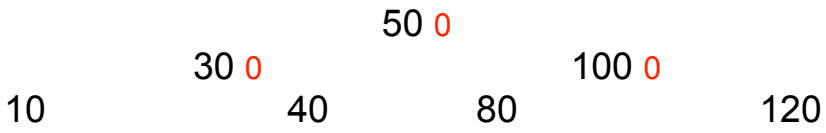
Introduzione agli algoritmi
Proff. E. Fachini – S. Caminiti
20 giugno 2020
Prova sulla seconda parte del programma

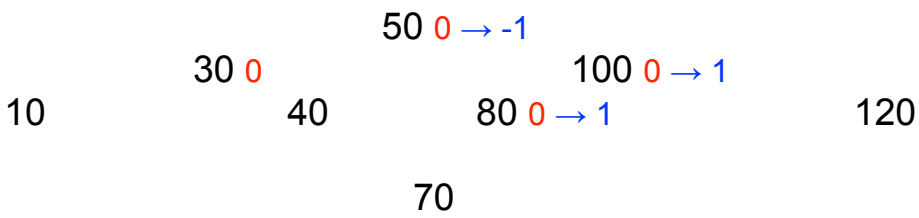
1. Si consideri l'operazione di inserimento in un AVL. Per che tipo di albero AVL l'inserimento di un nuovo nodo comporta solo l'aggiornamento del fattore di bilanciamento in tutti i nodi lungo il cammino dalla foglia alla radice, qualunque sia la sua altezza e qualunque sia il nodo nel quale si effettua l'inserimento? Si risponda motivando e servendosi di un esempio, nel quale esplicitare le variazioni del fattore di bilanciamento.

Sol. Ogni albero completo, cioè dove ogni nodo ha due o zero figli e le foglie sono tutte sullo stesso livello, è tale che un nuovo inserimento produce la necessità di aggiornare da 0 a 1 o a -1 tutti i fattori di bilanciamento lungo il cammino dal padre del nuovo nodo inserito fino alla radice. Questo perchè tutti i nodi hanno fattore di bilanciamento 0 ma la nuova foglia inserita aumenta di uno l'altezza di uno dei due sotto alberi vuoti del nodo foglia e questo aumento di altezza non viene mai compensato da un sotto albero di altezza minore incontrato salendo verso la radice visto che tutti i sotto alberi hanno la stessa altezza.

Esempio:



Inserendo 70 per esempio si ha:



2. Si imposti la relazione di ricorrenza che definisce il tempo di esecuzione della seguente funzione e la si risolva usando il metodo della sostituzione. Si commentino opportunamente i passaggi del calcolo, si disegni l'albero della ricorsione e come si giunge alla previsione sull'andamento del tempo di calcolo, si imposti l'induzione con chiarezza, sia nello scrivere quanto si vuole dimostrare sia nel formulare l'ipotesi induttiva.

```
Analizza(A,i.j)
input: A è un  array  di  interi  e i e j sono interi positivi
n = j-i+1
m = n/4
if n <1 then return 0
if n = 1 then return (A[1]);
x = Analizza(A,i,i+m) + Analizza(A,i+m+1,i+2m)
for h = i to j do
        x ← x + A[h];
end for
return (x);
```

Sol. La funzione è chiamata 2 volte su n/4 elementi e le istruzioni al di fuori delle chiamate sono eseguite in tempo lineare, quindi la relazione è:
$T(n) = 2T(n/4) + \Theta(n)$.
Si deve quindi risolvere la ricorrenza
$T(n) = 2T(n/4) + cn$ se $n > 1$
$T(n) = d$ se $n \leq 1$

L'albero della ricorsione per fare una previsione sull'andamento è qui abbozzato:

c n

c n/4 c n/4

$cn/4^2$ $c\ n/4^2$ $cn/4^2$ $c\ n/4^2$

Ponendo $n = 4^h$, l'albero ha altezza h, ogni nodo ha 2 figli e al livello i ogni nodo è etichettato con $c\ n/4^i$ Quindi, sommando su ogni livello si ha $2^i\ c\ n/4^i = c\ n/2^i$ da cui:
$T(n) = 2^h\ d + \sum_{i=0\ldots h-1} c\ n/2^i = 2^h\ d + c\ n \sum_{i=0\ldots h-1} 1/2^i \leq 2^h\ d + c\ n \sum_{i=0\ldots\infty} 1/2^i = O(n)$.
Infatti la serie di potenze converge a una costante e il termine $2^h\ d = n^{1/2}d$ è dominato dal termine lineare.

Ora verifichiamo che $T(n) = O(n)$ per induzione.
Faremo vedere che esistono due costanti positive k e n' tali che $T(n) \leq kn$, per ogni $n \geq n'$.
Sia vero per ipotesi induttiva per ogni $m < n$, consideriamo T(n):
$T(n) = 2T(n/4) + c\ n \leq$
$2k\ (n/4) + c\ n =$
$k/2\ (n) + c\ n =$
Vediamo se troviamo due valori per k e n' tali che

k/2 (n) + c n ≤ k n ⇔ 2c n ≤ k n ⇔ 2c ≤ k per ogni n ≥ 1, quindi la disuguaglianza è vera per k ≥ 2c e n ≥ 1.

Considerando il caso base abbiamo che T(1) = d e T(2) = 2d+c. Quindi T(2) ≤ 2k è vero prendendo k = 2(d+c) e per ogni n ≥ 1.
Poiché il termine additivo fornisce un limite inferiore lineare a T(n), possiamo concludere che T(n) = Θ(n).

3. Dato un array di interi che rappresenta la visita in preordine di un ABR, le cui chiavi sono tutte diverse, si definisca una algoritmo che dia in output l'ABR che l'ha determinata. Si descriva a parole l'idea algoritmica, si analizzi il tempo di esecuzione asintotico e si produca lo pseudocodice.

Sol. L'idea è la stessa utilizzata per costruire un albero binario qualunque da due sequenze, una rappresentante la visita inordine e una quella in preordine. In questo caso basta la visita in preordine perchè abbiamo a che fare con un ABR e quindi possiamo stabilire facilmente le chiavi che devono stare nel sotto albero sinistro della radice e quelle nel destro. Esaminando la preordine sappiamo che il primo elemento è la chiave della radice, poi tutte le chiavi più piccole di quella della radice devono andare nel sotto albero sinistro e le rimanenti nel destro. Ripetendo ricorsivamente il procedimento sulle chiavi del sotto albero sinistro e destro si costruisce tutto l'albero.
Il tempo di esecuzione dipende dalla forma dell'albero, infatti se ogni volta tutti i nodi andassero nel sotto albero sinistro, ogni volta la ricerca della prima chiave da inserire nel sottoalbero destro comporterebbe un numero di passi pari al numero delle chiavi e quindi la relazione di ricorrenza sarebbe:
$T(n) = T(0) + T(n-1) + \Theta(n)$, che ha soluzione in $\Theta(n^2)$ dando origine al caso peggiore. Nel caso migliore invece l'albero è degenere a sinistra, infatti in tal caso la ricerca delle chiavi si ferma dopo un unico confronto dando origine alla relazione di ricorrenza $T(n) = T(0) + T(n-1) + \Theta(1)$ che ha soluzione in $\Theta(n)$. Se le chiavi si distribuissero equamente tra il sotto albero sinistro e il destro e la relazione di ricorrenza sarebbe $T(n) = 2T(n/2) + \Theta(n)$, che ha soluzione in $\Theta(n \lg n)$.

```
costrABR(preordine, low, high)
{
 // Caso base
    if (low > high)  return NULL;
 // Il primo elemento in preordine è la chiave della radice
crea un nodo T, con T.key = preordine[low] e campi puntatori a NIL
    // se non ci sono altre chiavi viene dato in output il nodo T
    if (low == high)  return T;
    // Cerca il primo elemento maggiore della radice
     i = low;
    while i ≤ high and preordine[i] ≤ T.key do i++
  // Chiamiamo ricorsivamente la procedura sulle chiavi destinate al sottoalbero sinistro e destro
     T.left = costrABR( preordine,low+1, i-1);
```

```
    T.right = costrABR( preordine, i, high);
  return T;
}
```