

Algoritmi e Strutture Dati

Giacomo De Liberali

28 ottobre 2017

Indice

1 Fibonacci	4
1.1 Formula di Binet	4
1.2 Algoritmo ricorsivo	4
1.2.1 Albero di chiamata	5
1.2.2 Albero binario	6
1.3 Algoritmo iterativo 1	7
1.4 Algoritmo iterativo 2	7
2 Notazione asintotica	8
2.1 Definizione di O grande	8
2.2 Definizione di Omega grande	8
2.3 Definizione di Theta	8
2.3.1 Dimostrazione	9
2.3.2 Esempi	9
2.4 Proprietà delle classi	11
2.5 Definizione di omicron	11
2.6 Definizione di omega	11
2.7 Esempi	11
2.7.1 Esempio 1	11
2.7.2 Esempio 2	12
2.7.3 Esempio 3	12
2.7.4 Esempio 4	12
2.7.5 Esempio 5	12
2.7.6 Esempio 6	13
2.8 Dimostrazioni con i limiti	14
3 Complessità di un algoritmo	16
3.1 Metodo dell'iterazione	17
3.2 Metodo della sostituzione [p.70]	18
3.3 Teorema master - <i>metodo dell'esperto</i> [p.79]	18
3.3.1 Definizione	19
3.4 Quando non si può applicare	21
3.5 Dimostrazione	22
4 Tipo di dato	23
5 Struttura dati	23
5.1  Esempio	24
5.1.1 Realizzazione tramite array	24
5.1.2 Implementazione di <i>search</i>	24
5.1.3 Implementazione di <i>insert</i>	25
5.1.4 Implementazione di <i>delete</i>	25

5.1.5	Tecniche di miglioramento - Raddoppiamento e Dimezzamento	25
5.1.6	Realizzazione tramite strutture collegate	26
5.1.7	Implementazione di <i>insert</i>	26
5.1.8	Implementazione di <i>search</i>	26
5.1.9	Implementazione di <i>delete</i>	26
6	Vantaggi delle strutture indicizzate (array)	27
7	Vantaggi delle strutture collegate (liste)	27
8	Complessità di un algoritmo	28
8.1	✍ Esercizio	28
8.2	✍ Esercizio	28
8.3	✍ Esercizio	29
8.4	🏡 Esercizio per casa	30
8.5	✍ Esercizio	30
9	Pila	31
9.1	Implementazione tramite array	31
9.2	🏡 Esercizio per casa- Implementare uno <i>Stack</i> utilizzando due code	32
10	Coda	32
10.1	Implementazione tramite array circolari	33
10.2	✍ Esercizio	34
10.3	✍ Esercizio - Invertire una coda	34
11	Liste	35
11.1	Realizzazione con struttura collegata	35
11.1.1	Lista semplice	35
11.1.2	Lista doppiamente concatenata	35
11.1.3	Implementazione lista doppia	36
11.2	Invariante	36
11.3	Realizzazione lista doppia	37
11.3.1	Senza nodo sentinella	37
11.3.2	Con nodo sentinella	37
11.3.3	Realizzazione con array	38
12	Alberi	39
12.1	Alberi binari	39
12.2	Alberi k-ari	39
12.3	Completezza di un albero	40
12.3.1	✍ Esercizio	40
12.3.2	✍ Esercizio	41
12.4	Implementazione della struttura	42
12.4.1	Tramite array	42
12.4.2	Tramite strutture collegate	44
12.5	Rappresentazione binarizzata	45
12.6	Visite di alberi	46
12.6.1	Visita in profondità	46
12.6.2	Visita in ampiezza o a livelli	46
12.6.3	Tipi di visita	47
12.6.4	Teorema	48
12.7	Esercizi	49
12.7.1	✍ Esercizio - verificare completezza albero binario	49
12.7.2	✍ Esercizio - numero di nodi intermedi	50
12.8	Alberi bilanciati	51
12.8.1	✍ Esercizio : Costruire ricorsivamente un albero bilanciato	51

12.8.2	Esercizio : Stampare le chiavi di un livello k	52
12.8.3	Esercizio : Dimezzare le chiavi sui livelli pari di un albero	52
12.9	Alberi binari di ricerca	53
12.9.1	Operazioni	53
12.9.2	Esercizio : dimostrazione	56
12.9.3	Teorema	56
12.10	Alberi AVL	57
12.11B	Alberi	57
13	Algoritmi di Ordinamento	58
13.1	Algoritmi basati sul confronto	58
13.1.1	InsertionSort	58
13.1.2	MergeSort	59

1 Fibonacci

21 Settembre 2016

La funzione di *Fibonacci* è definita come

$$F_n = \begin{cases} 1 & \text{se } n = 1 \text{ o } n = 2 \\ F_{n-1} + F_{n-2} & \text{se } n \geq 3 \end{cases}$$

Ci sono diverse implementazioni dello stesso algoritmo, alcune più efficienti di altre. Ora vedremo tre differenti tipi di queste implementazioni.

1.1 Formula di Binet

La [formula di Binet](#) è collegata alla sezione aurea: il quadrato del numero è uguale al numero stesso addizionato a 1.

$$x^2 = x + 1 \rightarrow x^2 - x - 1 = 0$$

le soluzioni dell'equazione di secondo grado sono

$$\phi = \frac{1 + \sqrt{5}}{2}$$
$$\hat{\phi} = \frac{1 - \sqrt{5}}{2}$$

Provare per induzione che

$$F_n = \frac{1}{\sqrt{5}} (\phi^n - \hat{\phi}^n)$$

inizio con il caso base, $n = 1$. $Fib(1) = 1$, devo quindi provare che il risultato faccia 1:

$$\frac{1}{\sqrt{5}}(\phi - \hat{\phi}) = \frac{1}{5} \left(\frac{1 + \sqrt{5}}{2} - \frac{1 - \sqrt{5}}{2} \right) = \dots \text{ verificare che il risultato sia 1}$$

Una volta dimostrato il caso base, provo per $n \geq 3$. Sostituisco, raggruppo e ottengo:

$$F_n = F_{n-1} + F_{n-2} = \frac{1}{\sqrt{5}} \left[(\phi^{n-1} - \phi^{n-2}) + (\hat{\phi}^{n-1} - \hat{\phi}^{n-2}) \right] = \frac{1}{\sqrt{5}} \left[\underbrace{(\phi^{n-1} - \phi^{n-2})}_{\phi^n} - \overbrace{(\hat{\phi}^{n-1} + \hat{\phi}^{n-2})}^{\hat{\phi}^n} \right]$$

Per induzione (definizione ricorsiva, $\phi^{n-1} - \phi^{n-2} = \phi^n$ e $\hat{\phi}^{n-1} + \hat{\phi}^{n-2} = \hat{\phi}^n$), ho quindi dimostrato che $F_n = \frac{1}{\sqrt{5}} (\phi^n - \hat{\phi}^n)$. Matematicamente parlando l'algoritmo è corretto, ma nel caso pratico, a causa dell'arrotondamento dei floating point, non ha la garanzia teorica di correttezza (il risultato viene approssimato).

1.2 Algoritmo ricorsivo

22 Settembre 2016

La complessità è il numero di istruzioni eseguite dall'algoritmo una volta che viene lanciato.

```
Fibonacci(int n){  
    if(n<=2)  
        return 1;  
    else  
        return Fibonacci(n-1) + Fibonacci(n-2);  
}
```

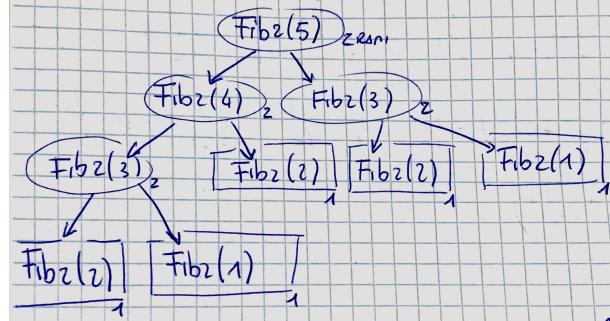
Determinare il numero di istruzioni eseguite dall'algoritmo La formula è

$$T_n = \begin{cases} 1 & \text{se } n = 1 \text{ o } n = 2 \\ 2 + T_{n-1} + T_{n-2} & \text{se } n \geq 3 \end{cases}$$

n	T_n
1	1
2	1
3	4
4	7

1.2.1 Albero di chiamata

Vogliamo ora rappresentare l'albero di chiamata dell'algoritmo ricorsivo $Fibonacci(5)$ con $n = 5$:



La complessità (*numero di istruzioni in questo caso*) di questo algoritmo con questi parametri è data dalla somma $T(5) = 13$

Ogni albero, come si vede dall'immagine, ha diversi nodi che si differenziano in

$$\begin{array}{ll} i(T_n) & \text{nodi interni} \\ f(T_n) & \text{nodi foglia} \end{array}$$

I *nodi interni* sono quelli non terminali (che hanno altri figli), mentre i *nodi foglia* sono quelli esterni (senza figli). La formula generale per determinare la complessità (numero di istruzioni in questo caso) è:

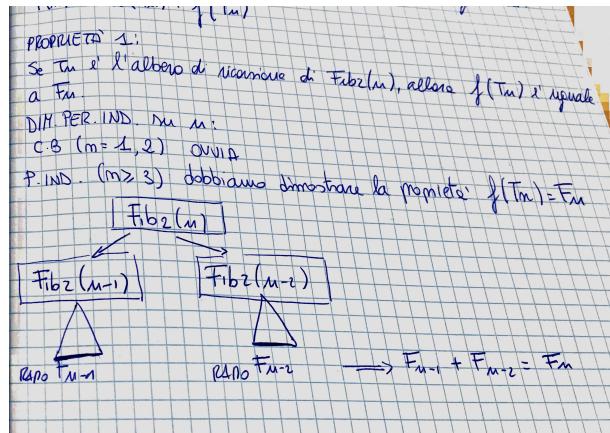
$$T_n = 2 \cdot i(T_n) + f(T_n)$$

Ora dobbiamo capire come determinare quanti sono i nodi foglia e quanti quelli interni, al fine di calcolare la complessità di questo algoritmo.

Se T_n è l'albero di ricorsione di $Fibonaccdi(2)$, allora

$$f(T_n) = F_n$$

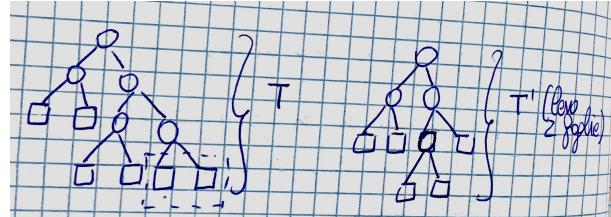
Dimostriamo ora per induzione. La base è $n = 1, 2$ la cui dimostrazione è ovvia. Procedo con il passo induttivo, per $n \geq 3$:



quindi $F_{n-1} + F_{n-2} = F_n$.

1.2.2 Albero binario

Se T è un albero binario (dove ogni nodo interno ha esattamente 2 figli), allora $i(T) = f(T) - 1$. Dimostrazione per induzione su $n =$ numero di vertici. Procedo con il test sul caso base con $n = 1$, la cui soluzione è ovvia. Applico il passo induttivo: suppongo di avere un albero con il numero di vertici $n \geq 2$ e ipotizzo che la regola precedentemente determinata sia vera fino a $n - 1$.



Tolgo due foglie dall'albero T , e costruisco l'albero T' :

$$\begin{cases} n(T') = n(T) - 2 & \text{dove } n(T) \text{ è il numero di vertici dell'albero } T \\ f(T') = f(T) - 1 & \text{rimuovo 2 foglie, ma il padre di queste due diventa anch'esso foglia} \\ i(T') = i(T) - 1 & \end{cases}$$

allora

$$\begin{aligned} i(T) &= i(T') + 1 \\ i(T') &= f(T') - 1 \\ i(T) &= f(T') \\ i(T) &= f(T) - 1 \end{aligned}$$

Riassumendo, la classe di questo algoritmo ricorsivo è

$$T(n) = 2 \cdot i(F_n) + f(T_n) = 2(F_n - 1) + F_n = 3F_n - 2 \approx F_n$$

che risulta equivalente ad una classe esponenziale.

Classi di complessità

Le classi di complessità di un algoritmo più comuni sono

$$\begin{aligned} T_n &= 2^n && \text{esponenziale} \\ T_n &= n^2 && \text{quadratico} \\ T_n &= \log_n && \text{logaritmico} \end{aligned}$$

🏡 Esercizio per casa

Dimostrare per induzione che

$$\forall n \geq 6, F_n \geq 2^{n/2}$$

Procedo con l'induzione su n , calcolando il caso base $n = 6$ e $n = 7$, in quanto *Fibonacci* ha due casi base.

$$F(6) = 8 \geq 2^{6/2} = 2^3 = 8 \checkmark$$

$$F(7) = 13 \geq 2^{7/2} \approx 11.3 \checkmark$$

a questo punto applico il passo induttivo per $n \geq 8$:

$$\begin{aligned} F_n &= F_{n-1} + F_{n-2} \\ &\geq 2^{\frac{n-1}{2}} + 2^{\frac{n-2}{2}} = 2^{\frac{n}{2}-\frac{1}{2}} + 2^{\frac{n}{2}-1} = 2^{\frac{n}{2}} \left(2^{-\frac{1}{2}} + 2^{-1} \right) = 2^{\frac{n}{2}} \underbrace{\left(\frac{1}{\sqrt{2}} + \frac{1}{2} \right)}_{>1} \geq 2^{\frac{n}{2}} \checkmark \end{aligned}$$

1.3 Algoritmo iterativo 1

Al posto di utilizzare un algoritmo ricorsivo, molto elegante ma poco efficiente, provo a riscrivere l'algoritmo di *Fibonacci* utilizzando un algoritmo iterativo:

```
Fibonacci(int n){
    int [n] F;
    F[1] = F[2] = 1;
    for(i=3;i<n;i++){
        F[i] = F[i-1] + F[i-2];
    }
    return F[n];
}
```

🏡 Esercizio per casa

Determinare la complessità della versione iterativa dell'algoritmo di *Fibonacci*.

Il numero di istruzioni che vengono eseguite in questa implementazione sono $3 + n^{\circ}$ *istruzioni ciclo for*.

Il numero di istruzioni eseguite in un ciclo *for*($i=k$ to k) sono:

$$\begin{aligned} n - k + 1 & \quad \text{le volte in cui il corpo del ciclo viene eseguito} \\ n - k + 2 & \quad \text{le volte in cui il test del ciclo viene eseguito} \end{aligned}$$

Quindi vengono eseguite $3 + (n - 2) + (n - 1) = 2n$ istruzioni. La classe di appartenenza è quindi *lineare*.

1.4 Algoritmo iterativo 2

Al posto di utilizzare l'algoritmo iterativo precedente, che utilizza un array, consideriamo un variante che utilizza 3 variabili in sostituzione dell'array:

```
Fibonacci(int n){
    int a=1,b=1,c;
    for(i=3;i<n;i++){
        c=a+b;
        a=b;
        b=c;
    }
    return c;
}
```

Riassumendo quindi i diversi algoritmi, possiamo definire la tabella seguente:

	T_n	Correttezza	Memoria
Fibonacci Binet	costante	NO	costante
Fibonacci ricorsivo	$\approx 2^n$	SI	lineare $\approx n$
Fibonacci iterativo 1	$\approx n$	SI	lineare n
Fibonacci iterativo 2	$\approx n$	SI	costante

2 Notazione asintotica

Ci sono diverse famiglie di algoritmi.

1. $O \rightarrow$ o grande
2. $\Omega \rightarrow$ omega grande
3. $\Theta \rightarrow$ theta
4. $o \rightarrow$ o piccola
5. $\omega \rightarrow$ omega piccola

2.1 Definizione di O grande

Una funzione $f(n)$ appartiene alla classe $O(g(n))$ quando

$$f(n) \in O(g(n)) \iff \exists c > 0 \exists n_0 \in \mathbb{N} \exists' \forall n \geq n_0 : f(n) \leq c \cdot g(n)$$

in pratica quando dopo un certo valore n_0 , $f(n)$ tenderà ad ∞ meno velocemente di $g(n)$.

2.2 Definizione di Omega grande

Una funzione $f(n)$ appartiene alla classe $\Omega(g(n))$ quando

$$f(n) \in \Omega(g(n)) \iff \exists c > 0 \exists n_0 \in \mathbb{N} \exists' \forall n \geq n_0 : c \cdot g(n) \leq f(n)$$

in pratica quando dopo un certo valore n_0 , $f(n)$ tenderà ad ∞ più velocemente di $g(n)$.

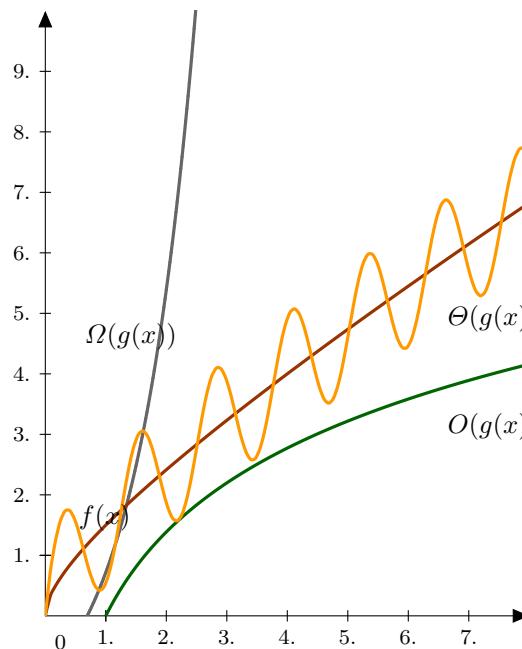
2.3 Definizione di Theta

Una funzione $f(n)$ appartiene a questa classe quando:

$$f(n) \in \Theta(g(n)) \iff \exists c_1 > 0 \exists c_2 > 0 \exists n_0 \in \mathbb{N} \exists' \forall n_0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

in pratica quando appartiene sia alla classe O che alla classe Ω , ovvero $f(n)$ e $g(n)$ tendono a ∞ allo stesso modo, senza che una delle due prevalga:

$$f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$$



2.3.1 Dimostrazione

La sintassi $f(n) = O(g(n))$ indica che $f(n)$ appartiene alla classe $O(g(n))$. Volendo rappresentare le classi come insiemi, possiamo definire:

$$\Theta(g(n)) = \Omega(g(n)) \cap O(g(n))$$

quindi esprimendolo in funzione di $f(n)$:

$$f(n) = \Theta(g(n)) \iff f(n) = \Omega(g(n)) \wedge O(g(n))$$

volendolo dimostrare, assumo sia vero che

$$f(n) = O(g(n))$$

$$f(n) = \Omega(g(n))$$

e procedo con la dimostrazione della tesi

$$f(n) = \Theta(g(n))$$

Ora esprimo in forma completa la mia ipotesi:

$$f(n) = O(g(n)) = \exists c' > 0 \exists n'_0 \in \mathbb{N} \ \exists' \ \forall n \geq n'_0 : f(n) \leq c' \cdot g(n)$$

$$f(n) = O(g(n)) = \exists c'' > 0 \exists n''_0 \in \mathbb{N} \ \exists' \ \forall n \geq n''_0 : c'' \cdot g(n) \leq f(n)$$

e anche la mia tesi

$$f(n) = \Theta(g(n)) = \exists c_1 > 0 \exists c_2 > 0 \exists n_0 \in \mathbb{N} \ \exists' \ \forall n : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

Sostituisco ora $c_2 = c'$ e $c_1 = c''$. Per far sì che la tesi sia valida devo prendere in considerazione un valore che soddisfi entrambe le equazioni dell'ipotesi, ovvero $n_0 = \max(n'_0, n''_0)$.

2.3.2 Esempi

Esempio 1 Verificare che

$$\sqrt{n+10} = \Theta(\sqrt{n})$$

Imposto la formula

$$\begin{aligned} \exists c_1 > 0 \exists c_2 > 0 \exists n_0 \in \mathbb{N} \ \exists' \ \forall n > n_0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \\ : c_1 \cdot \sqrt{n} \leq \sqrt{n+10} \leq c_2 \cdot \sqrt{n} \end{aligned}$$

e inizio a calcolare l'aprima parte $c_1 \cdot \sqrt{n} \leq \sqrt{n+10}$:

$$c_1 \cdot \sqrt{n} \leq \sqrt{n+10} \iff c_1^2 \cdot n \leq n+10 \iff (c_1^2 - 1)n \leq 10$$

da cui impongo $c_1^2 - 1 \leq 0 \iff c_1 \leq 1$. Scelgo arbitrariamente $c_1 = 1$ e ricavo $n_0 = 1$. Procedo ora a calcolare la seconda parte, $\sqrt{n+10} \leq c_2 \cdot \sqrt{n}$:

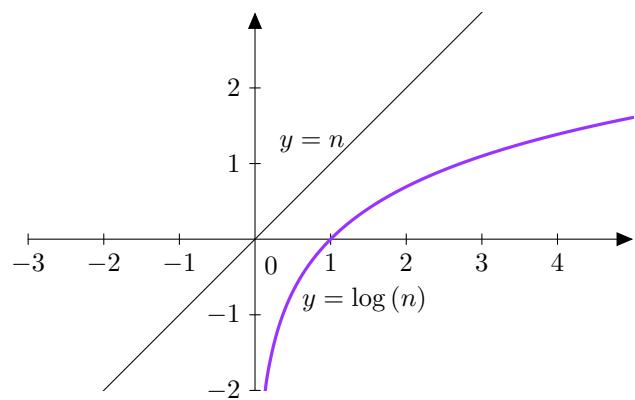
$$\sqrt{n+10} \leq c_2 \cdot \sqrt{n} \iff n+10 \leq c_2^2 \cdot n \iff 10 \leq n(c_2^2 - 1)$$

impongo $c_2^2 - 1 > 0$ da cui ricavo $c_2^2 > 1 \iff c_2 > 1$. Ora posso calolare $n_0 \geq \frac{10}{c_2^2 - 1}$. Arbitrariamente imposto $c_2 = \sqrt{2}$ (che è > 1), ricavando $n_0 = 10$.

Esempio 2 Verificare che

$$\log(n) = O(g(n))$$

Traccio un grafico delle due funzioni



Possiamo già notare dal grafico che sarà sempre vero.

2.4 Proprietà delle classi

	Riflessiva	Transitiva	Simmetrica	Antisimmetrica
$f(n) = O(g(n))$	✓	✓	✗	
$f(n) = \Omega(g(n))$	✓	✓	✗	
$f(n) = \Theta(g(n))$	✓	✓	✓	

2.5 Definizione di omicron

$$o(g(n)) = \{f(n) \mid \forall c > 0 \exists n_0 \in \mathbb{N} \exists' \forall n \geq n_0 : f(n) < c \cdot g(n)\}$$

Questa classe è una sotto classe di Omicron:

$$o(g(n)) \subseteq O(g(n))$$

2.6 Definizione di omega

$$\omega(g(n)) = \{f(n) \mid \forall c > 0 \exists n_0 \in \mathbb{N} \exists' \forall n \geq n_0 : c \cdot g(n) < f(n)\}$$

Questa classe è una sottoclasse di Omega:

$$\omega(g(n)) \subseteq \Omega(g(n))$$

2.7 Esempi

2.7.1 Esempio 1

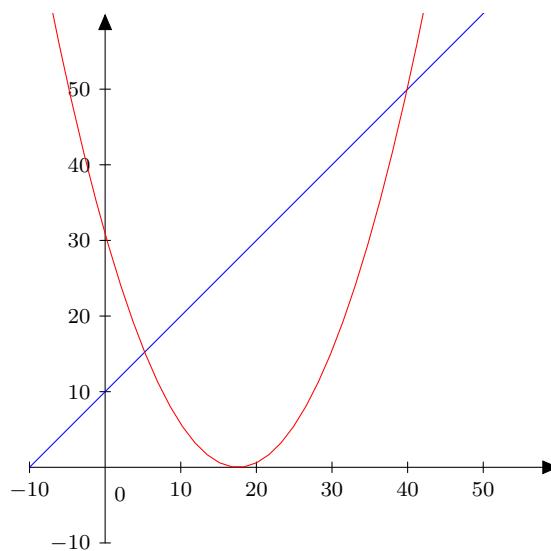
Determinare se la seguente uguaglianza è vera:

$$40n + 100 = O(n^2 + 10n + 300)$$

Noto intanto che il primo termine è lineare, mentre il secondo è quadratico.

$$n^2 - 30n + 200 = 0 \rightarrow n_{1,2} = 10, 20$$

da cui ricavo $n_0 = 20$ e arbitrariamente $c = 1$.



2.7.2 Esempio 2

Determinare se la seguente uguaglianza è vera:

$$\underbrace{\frac{1}{2}n^2 - 3n}_{f(n)} = \overbrace{n^2}^{g(n)}$$

Noto che entrambi i termini sono quadratici. Verifico che esista un c che soddisfi le condizioni della classe O :

$$\frac{1}{2}n^2 - 3n \leq c \cdot n^2 \iff \frac{1}{2}n - 3 \leq c \cdot n = \left(\frac{1}{2} - c\right)n \leq 3$$

da qui capisco che l'uguaglianza è vera $\forall c \geq \frac{1}{2}$. Determino $n_0 = 1$ in modo arbitrario, in quanto ininfluente data la precedente imposizione.

2.7.3 Esempio 3

Determinare se la seguente uguaglianza è vera:

$$\frac{1}{2}n^2 - 3n \geq c \cdot n^2$$

Noto che entrambi i termini sono quadratici. Verifico che esista un c che soddisfi le condizioni della classe Ω :

$$\frac{1}{2}n^2 - 3n \geq c \cdot n^2 = \left(\frac{1}{2} - c\right)n \geq 3 = n \geq \frac{3}{\frac{1}{2} - c} \iff \frac{1}{2} - c > 0 \rightarrow c < \frac{1}{2} \wedge c > 0$$

imposto $c = \frac{1}{14}$ (*un valore arbitrario dentro al suo range di esistenza*) e procedo al calcolo di n :

$$n \geq \frac{3}{\frac{1}{2} - \frac{1}{14}} = 7$$

2.7.4 Esempio 4

Determinare la classe di appartenenza di $f(n)$

$$\begin{aligned} n\sqrt{n} &= 9^{\log_3 n} \\ &= (3^2)^{\log_3 n} = 3^{2\log_3 n} = 3^{\log_3 n^2} = n^2 \end{aligned}$$

da cui ricaviamo che $f(n) = O(n^2)$

2.7.5 Esempio 5

Determinare se $f(n)$ appartiene o no alla classe

$$n! = O(n^n)$$

iniziamo a calcolare $n!$:

$$\begin{aligned} n! &= \underbrace{\frac{\leq n}{1} \cdot \frac{\leq n}{2} \cdot \frac{\leq n}{3} \cdot \dots \cdot \frac{\leq n}{n}}_{n \text{ volte}} \\ &\leq \underbrace{n \cdot n \cdot n \cdot \dots \cdot n}_{n \text{ volte}} \end{aligned}$$

da cui ricaviamo che $f(n) = O(n^n)$

2.7.6 Esempio 6

Determinare se $f(n)$ appartiene o no alla classe

$$\log(n!) = O(n \log n)$$

iniziamo a calcolare $\log(n!)$:

$$\begin{aligned}\log(n!) &= \log \prod_{i=1}^n i \\ &= \sum_{i=1}^n \log i \\ &\leq n \log(n)\end{aligned}$$

da cui ricaviamo che $f(n) = O(n \log n)$

2.8 Dimostrazioni con i limiti

- $f(n) = o(g(n))$. Se $f(n) = o(g(n))$ allora $f(n) = O(g(n))$, perché $o(g(n)) \subseteq O(g(n))$:

$$f(n) = o(g(n)) \iff \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

- $f(n) = \omega(g(n))$. Se $f(n) = \omega(g(n))$ allora $f(n) = \Omega(g(n))$, perché $\omega(g(n)) \subseteq \Omega(g(n))$:

$$f(n) = \omega(g(n)) \iff \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$$

- $f(n) = \Theta(g(n))$:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = L \implies f(n) = \Theta(g(n))$$

⌚ Esempio : verificare che $f(n)$ appartenga ad una classe:

$$n = O(n \log(\log n)) \rightarrow \lim_{n \rightarrow \infty} \frac{\cancel{n}}{\cancel{n} \log(\log n)} = 0 \quad \checkmark$$

Tuttavia non sempre è possibile utilizzare un limite per determinare l'appartenenza ad una classe. Può infatti accadere che non esista il limite di una funzione, per esempio in casi di funzione oscillante. Esempio:

$$(1 + \sin n)n = 2n$$

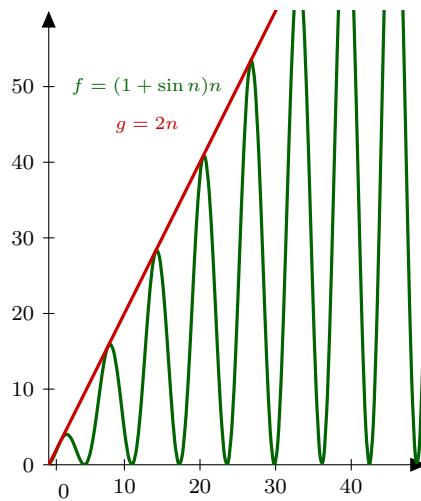
qui, non potendo applicare il limite in quanto insensitente, dobbiamo impostare:

$$\sin n \leq 1$$

$$1 + \sin n \leq 1 + 1 = 2$$

$$n(1 + \sin n) \leq 2n$$

Andiamo ora ad osservare il grafico:



Possiamo concludere affermando che $(1 + \sin n)n = O(2n)$.

Le diverse classi, semplicisticamente, possono essere viste come:

- $O \leq$
- $\Omega \geq$
- $\Theta =$
- $o <$
- $\omega >$

☞ Esercizio per casa: verificare che le seguenti funzioni appartengano alle rispettive classi:

a) $3n \log n = O(5n + \log n^3)$

b) $2^{n+n} = O(2^n)$

c) $2^n = \Omega(2^{n+k})$ con $k \in \mathbb{Z}^+$

Soluzione a):

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{3n \log n}{5n + \log n} \stackrel{l'Hôpital}{=} \lim_{n \rightarrow \infty} \frac{3(\log n + 1)}{\frac{3}{n} + 5} = \lim_{n \rightarrow \infty} \frac{n(\log n + 1)}{5} = +\infty$$

che significa $f(n) = \omega(g(n))$ e quindi $f(n) = \Omega(g(n))$.

Soluzione b):

$$\begin{aligned} 2^{n+n} &= \underset{\geq 2^n}{2^n} \cdot \underset{\geq 2^n}{2^n} \\ &> 2^n \end{aligned}$$

che significa $f(n) = \omega(g(n))$ e quindi $f(n) = \Omega(g(n))$.

☞ Esercizio : riordinare le seguenti funzione in ordine di crescita asintotica:

- | | |
|------------------------------|-------------------------------|
| a) n^2 | a) $\log(\log n)$ |
| b) $n \log n$ | b) $17 \log n$ |
| c) $n^3 + \log n$ | c) \sqrt{n} |
| d) \sqrt{n} | d) $n + 6 \log n$ |
| e) $n^2 + 2n \log n$ | e) $n \log n$ |
| f) $\log(\log n)$ | f) $10n^{\frac{3}{2}}$ |
| g) $17 \log n$ | g) $\{n^2, n^2 + 2n \log n\}$ |
| h) $10n^{\frac{3}{2}}$ | h) $5n^2 \cdot \log(\log n)$ |
| i) $n^5 - n^4 + 2n$ | i) $3n^2 + n^3 \log n$ |
| j) $5n^2 \cdot \log(\log n)$ | j) $n^3 + \log n$ |
| k) $3n^2 + n^3 \log n$ | k) $n^5 - n^4 + 2n$ |
| l) $n + 6 \log n$ | |

3 Complessità di un algoritmo

Determinare la complessità (numero di istruzione eseguite) dei seguenti codici. La complessità è espressa nei termini del caso peggiore, ovvero dobbiamo andare a determinare quante istruzioni eseguono queste porzioni di codice nel caso peggiore.

a) complessità = $O(T(\text{cond}) + \max\{T(\text{blocco1}), T(\text{blocco2})\})$

```
if(cond){
    // blocco 1
} else{
    // blocco 2
}
```

ovvero il numero di istruzioni richieste dalla condizione *cond* e il massimo di istruzioni richieste da uno dei due blocchi (teniamo in considerazione in più lungo).

b) complessità = $O(n \cdot T(\text{blocco}))$

```
for(i=1 to n){
    // blocco
}
```

ovvero il numero di cicli addizionato al numero di istruzioni eseguite in *blocco*.

c) complessità = $O(n \cdot T(\text{cond}) \cdot T(\text{blocco}))$

```
while(cond){
    // blocco
}
```

 Esercizio : analizzare il seguente algoritmo:

```
ricseq(L : lista, x : item) : boolean {
    foreach(y ∈ L){
        if(y = x)
            return true;
    }
    return false;
}
```

Calcoliamo i 3 differenti tempi di esecuzione:

$$T_{\text{best}}(n) = \Theta(1)$$

elemento nella 1° posizione della lista

$$T_{\text{avg}}(n) = \Theta\left(\frac{n}{2}\right)$$

elemento in media a metà della lista

$$T_{\text{wrost}}(n) = \Theta(n)$$

elemento in ultima posizione o non presente nella lista

3.1 Metodo dell'iterazione

Dato un algoritmo ricorsivo, dobbiamo cercare di ricondurci ad un algoritmo iterativo.

⇒ Esempio

$$T(n) = \begin{cases} c + T\left(\frac{n}{2}\right) & \text{con } n = 2^k \\ 1 & \end{cases}$$

Sviluppo l'algoritmo per cicli, fino a quando non arrivo a riconoscerne lo sviluppo:

$$\begin{aligned} T(n) &= c + T\left(\frac{n}{2}\right) && 1^{\circ} \text{ ciclo} \\ &= c + \left[T\left(\frac{n}{2^2}\right)\right] = 2c + T\left(\frac{n}{2^2}\right) && 2^{\circ} \text{ ciclo} \\ &= 2c + \left[c + T\left(\frac{n}{2^3}\right)\right] = 3c + T\left(\frac{n}{2^3}\right) && 3^{\circ} \text{ ciclo} \\ &\vdots \\ &= k \cdot c + T\left(\frac{n}{2^k}\right) && \frac{n}{2^k} = 1 \iff k = \log n \\ &= c \cdot \log n + 1 \end{aligned}$$

⇒ Esempio

$$T(n) = \begin{cases} 9T\left(\frac{n}{3}\right) + n & \text{se } n \geq 2 \\ 1 & \text{se } n = 1 \end{cases}$$

La serie geometrica $\sum_{i=0}^k q^i = \frac{q^{k+1} - 1}{q - 1}$. Se $q = 3$ allora $\sum_{i=0}^k 3^i = \frac{3^{k+1} - 1}{2}$.

Inizio a sviluppare per cicli:

$$\begin{aligned} T(n) &= 9T\left(\frac{n}{3}\right) + n && 1^{\circ} \text{ ciclo} \\ &= 9 \left[9T\left(\frac{n}{3^2}\right) + \frac{n}{3} \right] + n = 9^2 T\left(\frac{n}{3^2}\right) + 3n + n && 2^{\circ} \text{ ciclo} \\ &= 9 \left\{ 9 \left[9 \left(T\left(\frac{n}{3^3}\right) + \frac{n}{3^2} \right) \right] \right\} + n = 9^3 T\left(\frac{n}{3^3}\right) + 9n + 3n + n = 9^3 T\left(\frac{n}{3^3}\right) + n(3^2 + 3^1 + 3^0) && 3^{\circ} \text{ ciclo} \\ &\vdots \\ &= 9^k \cdot T\left(\frac{n}{3^k}\right) + n \cdot \sum_{i=0}^{k-1} 3^i && k^{\circ} \text{ ciclo} \\ &= 9^k \cdot T\left(\frac{n}{3^k}\right) + n \cdot \frac{3^k - 1}{2} && n = 3^k \iff k = \log_3 n \\ &= 9^{\log_3 n} + \frac{n \cdot 3^{\log_3 n} - 1}{2} \\ &= n^2 + \frac{n(n-1)}{2} \\ &= O(n^2) \end{aligned}$$

Sono quindi arrivato a dimostrare che questo algoritmo, riscritto in modalità iterativa, possiede complessità $O(n^2)$.

3.2 Metodo della sostituzione [p.70]

12 Ottobre 2016

➊ Esempio : sapendo che $\lfloor \frac{n}{2} \rfloor =$ il più grande intero inferiore a $\frac{n}{2}$

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \\ T\left(\lfloor \frac{n}{2} \rfloor\right) + n & \text{se } n \geq 2 \end{cases}$$

devo provare ad indovinare la classe della funzione. Ipotizzo che $T(n) = O(n)$. Dimostro per induzione la mia ipotesi:

$$\exists c > 0 \ \exists' \ T(n) \leq c \cdot n \quad \text{per } n \text{ sufficientemente grande}$$

impongo $n_0 = 1$ in modo arbitrario, e verifico la base dell'induzione:

$$T(1) \leq c \cdot 1 \implies 1 \leq c \implies c \geq 1$$

ora verifico che valga anche per $n \geq 2$:

$$T(n) = T\left(\lfloor \frac{n}{2} \rfloor\right) + n \leq c\lfloor \frac{n}{2} \rfloor + n \leq c\frac{n}{2} + n \leq c\frac{n}{2} + n = \left(\frac{c}{2} + 1\right)n \leq c \cdot n$$

se ora imposto $\frac{c}{2} + 1 \leq c$ e ottengo $c \geq 2$. A questo punto verifico che $c = 2$ soddisfi le condizioni iniziali.

3.3 Teorema master - *metodo dell'esperto* [p.79]

Questo metodo appartiene alla filosofia *Divide et imperat*. Si suddivide in 3 fasi:

- 1) Scomposizione del problema
- 2) Soluzione dei sottoproblemi
- 3) Ricomposizione delle soluzioni dei sottoproblemi

Di conseguenza

$$T(n) = T_S(n) + \sum_{i=1}^k T(n_i) + T_R(n)$$

ovvero la complessità dipende da il tempo di scomposizione, dal tempo di soluzione dei k sottoproblemi e infine da il tempo di ricomposizione. Noi indicheremo con $f(n)$ la somma tra tempo di scomposizione e ricomposizione. A questo punto

$$T(n) = f(n) + \sum_{i=1}^k T(n_i)$$

Il caso di studio che noi andremo a coprire riguarderà solamente le funzioni i cui sottoproblemi hanno medesima dimensione.

3.3.1 Definizione

Siano date le costanti

$$\begin{aligned} a &\geq 1 && \text{numero di sottoproblemi} \\ b &> 1 && \text{dimensione originale} \end{aligned}$$

e sia data $f(n)$. Sia $T(n)$ una funzione definita come

$$T(n) = \begin{cases} aT\left(\frac{n}{b}\right) + f(n) \\ 1 \end{cases}$$

dove $\frac{n}{b}$ rappresenta la dimensione dei sottoproblemi e $d = \log_b a$ è equivalente alla dimensione dell'albero di ricorsione di $T(n)$, allora $T(n)$ può essere asintoticamente limitata nei seguenti modi:

- 1) Se $f(n) = O(n^{d-\varepsilon})$, $\varepsilon > 0$ allora $T(n) = \Theta(n^d)$
- 2) Se $f(n) = \Theta(n^d)$ allora $T(n) = \Theta(n^d \cdot \log(n))$
- 3) Se $f(n) = \Omega(n^{d+\varepsilon})$, $\varepsilon > 0$ $\exists c < 1 \ \exists' n$ sufficientemente grande $aT\left(\frac{n}{b}\right) \leq c \cdot f(n)$, allora $T(n) = \Theta(f(n))$

Esempi

Esempio 1 Determinare la complessità di della funzione $T(n)$ definita come:

$$T(n) = c + T\left(\frac{n}{2}\right)$$

come prima cosa individuo le costanti:

$$\begin{aligned} a &= 1 \\ b &= 2 \end{aligned}$$

e la funzione $f(n) = c$. A questo punto determino la dimensione dell'albero di ricorsione:

$$d = \log_b a = \log_2 1 = 0$$

e trovo $g(n) = n^d = n^0 = 1$. Fatto ciò, confronto $f(n)$ e $g(n)$:

$$f(n) = \Theta(g(n))$$

siamo dunque nel 2° caso, dove $f(n) = \Theta(n^d)$, da cui ricavo:

$$T(n) = \Theta(n^d \cdot \log n) = \Theta(\log n)$$

Esempio 2 Determinare la complessità di della funzione $T(n)$ definita come:

$$T(n) = 3T\left(\frac{n}{2}\right) + n^2$$

come prima cosa individuo le costanti:

$$\begin{aligned} a &= 3 \\ b &= 2 \end{aligned}$$

e la funzione $f(n) = n^2$. A questo punto determino la dimensione dell'albero di ricorsione:

$$d = \log_b a = \log_2 3 \approx 1.6$$

e trovo $g(n) = n^d = n^{\log_2 3}$. Fatto ciò, confronto $f(n)$ e $g(n)$:

$$f(n) = \Omega(g(n))$$

dal momento che $n^{\log_2 3} < n^{2+\varepsilon}$ per $\varepsilon < 2 - \log_2 3$. Siamo dunque nel 3° caso. Verifico ora che $\exists c < 1 \ \exists' c'$ per n sufficientemente grande $af(\frac{n}{b}) \leq c \cdot f(n)$:

$$\begin{aligned} 3\left(\frac{n}{2}\right)^2 &\leq c \cdot n^2 \\ \frac{3}{4}n^2 &\leq c \cdot n^2 \iff \frac{3}{4} \leq c \leq 1 \end{aligned}$$

provata l'esistenza di c , posso concludere che $f(n) = \Omega(n^{d+\varepsilon})$, da cui ricavo (3° caso):

$$T(n) = \Theta(f(n)) = \Theta(n^2)$$

Esempio 3 Determinare la complessità di della funzione $T(n)$ definita come:

$$T(n) = 4T\left(\frac{n}{2}\right) + n^2$$

come prima cosa individuo le costanti:

$$a = 4$$

$$b = 2$$

e la funzione $f(n) = n^2$. A questo punto determino la dimensione dell'albero di ricorsione:

$$d = \log_b a = \log_2 4 = 2$$

e trovo $g(n) = n^d = n^2$. Fatto ciò, confronto $f(n)$ e $g(n)$:

$$f(n) = \Theta(g(n))$$

siamo dunque nel 2° caso, dove $f(n) = \Theta(n^d)$, da cui ricavo:

$$T(n) = \Theta(n^d \cdot \log n) = \Theta(n^2 \cdot \log n)$$

Esercizio per casa4 Determinare la complessità di della funzione $T(n)$ definita come:

$$T(n) = T\left(\frac{n}{2}\right) + 2^n$$

come prima cosa individuo le costanti:

$$a = 1$$

$$b = 2$$

e la funzione $f(n) = 2^n$. A questo punto determino la dimensione dell'albero di ricorsione:

$$d = \log_b a = \log_2 1 = 0$$

e trovo $g(n) = n^d = n^0 = 1$. Fatto ciò, confronto $f(n)$ e $g(n)$:

$$f(n) = \Omega(g(n))$$

dal momento che $2^{n+\varepsilon} \geq 1$ per $\varepsilon > 0$ e n sufficientemente grande. Siamo dunque nel 3° caso. Verifico ora che $\exists c < 1 \ \exists' c'$ per n sufficientemente grande $af(\frac{n}{b}) \leq c \cdot f(n)$:

$$\begin{aligned} 2^{\frac{n}{2}} &\leq c \cdot 2^n \\ 2^{\frac{n}{2}}/2^n &\leq c \\ \frac{2^{\frac{n}{2}} \cdot 2^{\frac{1}{2}}}{2^{\frac{n}{2}}} &\leq c \iff \sqrt{2} \leq c \implies c < 1 \end{aligned}$$

provata l'esistenza di c , posso concludere che $f(n) = \Omega(n^{d+\varepsilon})$, da cui ricavo (3° caso):

$$T(n) = \Theta(f(n)) = \Theta(2^n)$$

3.4 Quando non si può applicare

Il teorema master non si può applicare quando:

- 1) a non è una costante.

$$T(n) = \underbrace{2^n}_{\text{non costante}} T\left(\frac{n}{2}\right) + n^2$$

- 2) $f(n)$ è negativa.

$$T(n) = 2T\left(\frac{n}{2}\right) - \underbrace{n^2 \log n}_{\text{negativa}}$$

- 3) a non è ≥ 1 .

$$T(n) = \underbrace{\frac{1}{2}}_{\not\geq 1} T\left(\frac{n}{2}\right) + n^3$$

☞ Esempio Determinare la complessità del seguente codice:

```
myAlgorithm(int n) : int {
    int a, i, j;
    if (n > 1) {
        a=0;
        for(i=0 to n)
            for(j=1 to n)
                a = a + (i+1)*(j+4)           // ripetuto n^2 volte
            for(i=1 to 16)                  // per 16 volte richiama con
                a = a + myAlgorithm(n/4)     // sottoproblemi di
    } else {                                // dimensione n/4
        return n-1;
    }
}
```

Determiniamo $T(n)$:

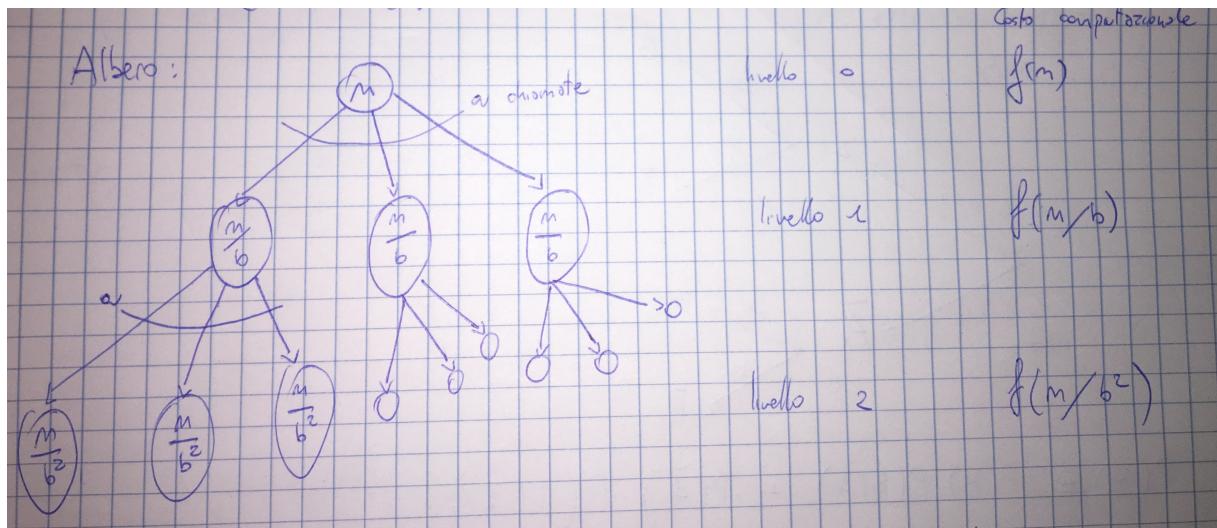
$$T(n) = \begin{cases} n^2 + 16T\left(\frac{n}{4}\right) \\ c \end{cases}$$

Trovo

$$\begin{aligned} a &= 16 \\ b &= 4 \\ f(n) &= n^2 \\ d &= \log_4 16 = 2 \\ g(n) &= n^2 \\ f(n) &= \Theta(g(n)) \implies 2^{\text{°}} \text{ caso} \\ T(n) &= \Theta(n^2 \cdot \log n) \end{aligned}$$

3.5 Dimostrazione

13 Ottobre 2016



- a) Al livello i la dimensione dei sottoproblemi è $\frac{n}{b^i}$
- b) Al livello i le singole chiamate hanno complessità $f\left(\frac{n}{b^i}\right)$
- c) Al livello i , vi saranno a^i vertici
- d) Il numero di livelli dell'albero è $i = \log_b n$
- e) Il numero di foglie è $a^{\log_b n} = n^d$

4 Tipo di dato

Un *tipo di dato* è un modello matematico che consiste in una collezione di valori sui quali sono ammesse certe operazioni. Un *tipo di dato* specifica cosa un'operazione deve fare, ma non come l'operazione può essere realizzata. E un *tipo di dato* non specifica nemmeno come gli oggetti della collezione possono essere organizzati in modo che le operazioni siano efficienti e la collezione stessa occupi poco spazio in memoria.

5 Struttura dati

Una *struttura dati* è una particolare organizzazione delle informazioni che permette di supportare in modo efficiente le operazioni di un tipo di dato. Una *struttura dati* specifica come organizzare i dati e come realizzare le operazioni.

Le strutture dati si diversificano in base a:

- disposizione degli elementi
- numero di elementi
- tipo degli elementi

La classe di una struttura dati può essere invece

- *lineare*: in cui gli elementi sono disposti in sequenza e in cui si distingue un ordine
- *non lineare*: in cui gli elementi non sono disposti in sequenza
- *statiche*: in cui il numero degli elementi rimane costante nel tempo
- *dinamiche*: in cui il numero degli elementi varia nel tempo
- *omogenee*: in cui gli elementi hanno tutti lo stesso tipo
- *non omogenee*: in cui i dati hanno tipo diverso

5.1 ➡ Esempio

Tipo di dato: Dizionario. Un dizionario S è formato da un insieme di copie chiave k e valore v . Le operazioni definite su questo tipo di dato sono:

```
search(dizionario S, chiave v) : valore
    // post: restituisce il valore della chiave k se presente, null altirimenti

insert(dizionario S, chiave k, valore v)
    // post: inserisce/aggiorna il valore della chiave k

delete(dizionario S, chiave k)
    // pre: la chiave k esiste nel dizionario
    // post: cancella da S la coppia (k,v)
```

5.1.1 Realizzazione tramite array

Andiamo ora a realizzare una struttura dati che permette la gestione di un dizionario tramite un array ordinato di elementi.

Dati: un array A di dimensione n contenente record con due campi ($key, info$) ordinati in base a key . $A.length$ contiene la dimensione dell'array.

In termini di spazio: $S(n) = \Theta(n)$.

5.1.2 Implementazione di *search*

```
search(dizionario S, chiave v)
    i = search_index(A,k,A.length);
    if(i == -1)
        return null;
    return A[i].info;

search_index(dizionario D, chiave k, int start, int end)
    if(start > end)
        return -1; // impera
    med = ⌊(start+end)/2⌋ // divide
    if(D[med].key == k)
        return med; // impera
    else if(D[med].key > k)
        return search_index(D,k,start,med-1); // impera
    else
        return search_index(D,k,med+1,end); // impera
```

Determino la complessità di *search_index*:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ T\left(\frac{n}{2}\right) + d & \text{se } n > 0 \end{cases}$$

risolvo con il *teorema master*:

$$\begin{aligned} d &= n^{\log_b a} = n^{\log_{\frac{1}{2}} 1} = n^0 = 1 \\ f(n) &= \Theta(1) \implies 2^{\circ} \text{ caso} \implies T(n) = \Theta(\log n) \end{aligned}$$

Il costo di *search* è il costo di *search_index* + il costo delle costanti, ed equivale a: $T(n) = \Theta(\log n)$.

5.1.3 Implementazione di *insert*

```
insert(dizionario S, chiave v, valore v)
    i=1;
    while(i ≤ A.length && A[i].key < k) // ripetuto i volte
        i++;
    if(i ≤ A.length && A[i].key == k) // costante
        A[i].info = v;
    else
        reallocate(A,A.length+1); // O(n)
        for(j=A.length downTo i+1) // (n-i+1) volte
            A[j] = A[j-1];
        A[i].key = k;
        A[i].info = v;
```

Determino la complessità di *insert*:

$$T(n) = c + O(n) + d_i + d(n - i) = c + O(n) = \Theta(n)$$

5.1.4 Implementazione di *delete*

```
delete(dizionario S, chiave v)
    i = search_index(A,k,i,A.length); // Θ(log n)
    for(j=i to A.length-1) // (n-1-i) volte = Θ(n)
        A[j] = A[j+1];
    reallocate(A,A.length-1); // O(n)
```

Determino la complessità di *delete*:

$$T(n) = \Theta(n) + O(n) = \Theta(n)$$

5.1.5 Tecniche di miglioramento - Raddoppiamento e Dimezzamento

Proviamo ad utilizzare una tecnica chiamata *raddoppiamento e dimezzamento*, che consiste nel fare il *reallocate* dell'array in un modo più smart.

Si mantiene un array di dimensione h dove:

$$\forall n > 0, n \leq h \leq 4n$$

Nelle prime n celle l'array contiene gli elementi della collezione, mentre il contenuto delle successive è indefinito (e quindi libero). Inizialmente, quando $n = 0$, poniamo $h = 1$. Ogni qual volta che $n > h$, l'array viene riallocato raddoppiandone la dimensione:

$$h = 2h$$

Ogni qual volta $n = \frac{h}{4}$, l'array viene riallocato dimezzandone la dimensione:

$$h = \frac{h}{2}$$

Lo spazio in memoria consumato da questo array è $h = \Theta(n)$. Con questa tecnica di analisi ammortizzata, il costo medio delle operazioni di inserimento/eliminazione rimane costante.

5.1.6 Realizzazione tramite strutture collegate

Andiamo a realizzare un astruttura data basata su una collezione L di n record collegati, con formato $(key, info, next, prev)$, dove $next$ e $prev$ sono puntatori al nodo successivo e precedente. Un attributo $L.head$ punta al primo elemento della lista. Se $L.head = null$ allora la lista è vuota. Lo spazio in memoria occupato da questa soluzione è $\Theta(n)$.

5.1.7 Implementazione di *insert*

Questo metodo prevede l'interimento in testa, in quanto risulta essere l'operazione più veloce ed efficiente con questo tipo di struttura dati.

```
insert(dizionario L, chiave k, valore v)
    p = nuovo record con chiave k e valore v
    p.next = L.head;
    if(L.head != null)
        L.head.prev = p;
    L.head = p;
    p.prev = null;
```

Determino la complessità di *insert*:

$$T(n) = \Theta(1)$$

che significa essere costante.

5.1.8 Implementazione di *search*

```
search(dizionario L, chiave k)
    x = L.head;
    while(x != null && x.key != k)
        x = x.next;
    if(x != null)
        return x.info;
    else
        return null;
```

Determino la complessità di *search*:

$$T(n) = O(n)$$

ovvero nel caso peggiore (elemento non presente o in ultima posizione) sono limitato superiormente dalla dimensione stessa della lista.

5.1.9 Implementazione di *delete*

```
delete(dizionario L, chiave k)
    x = L.head;
    while(x != null)
        if(x.key == k)
            if(x.next != null)
                x.next.prev = x.prev;
            if(x.prev != null)
                x.prev.next = x.next;
            else
                L.head = x.next;
            temp = x;
            x = x.next;
            remove(temp);
        else
            x = x.next;
```

Determino la complessità di *delete*:

$$T(n) = \Theta(n)$$

ovvero dipende linearmente dalla dimensione stessa della lista (devo sempre scorrere tutta la lista).

6 Vantaggi delle strutture indicizzate (array)

In un array di dimensione n , gli indici delle celle possono essere compresi nell'intervallo $[0, n - 1]$ in linguaggio C oppure tra $[1, n]$ in psudocodice. Ogni array ha le seguenti caratteristiche:

- è possibile accedere in lettura/scrittura agli elementi di una qualsiasi cella in un tempo costante
- (forte) gli indici sono consecutivi
- (debole) non è possibile aggiungere nuove celle ad un array \implies *riallocazione*

7 Vantaggi delle strutture collegate (liste)

Ogni lista ha le seguenti proprietà basilari:

- (forte) è sempre possibile aggiungere/rimuovere record
- (debole) gli indici dei record non sono necessariamente consecutivi

Sarà quindi preferibile utilizzare strutture collegate in presenza di una forte dinamicità, mentre sarà opportuno utilizzare strutture indicizzate quando, ad esempio, devo effettuare molte ricerche, in quanto ricerco in un array con tempo $\Theta(\log n)$.

8 Complessità di un algoritmo

8.1 Esercizio

Date le seguenti procedure A e B si determini la complessità asintotica della funzione $A(n)$ su input n .

```
A(int n)
    s=0;
    for(i=0 to n)
        s = s + B(i);
    return s;

B(int m)
    s=0;
    for(i=0 to m)
        s = s + 1;
    return s;
```

Per prima cosa, per calcolare la complessità di A , dobbiamo determinare quella di B .

$$T_B(m) = \Theta(m)$$

ora calcoliamo quella di A :

$$T_A(n) = \sum_{i=1}^n c \cdot i = c \cdot \sum_{i=1}^n i = c \cdot \frac{n(n+1)}{2} = \Theta(n^2)$$

Abbiamo notato che la funzione A somma i primi n numeri interi, ma con complessità n^2 risulta essere estremamente inefficiente. Si sarebbe potuto risolvere in tempo costante con l'operazione $\frac{n(n+1)}{2}$. La funzione B invece è la funzione identità, ovvero $B(m) = m$.

8.2 Esercizio

Si valuti l'ordine di grandezza della complessità nel caso peggiore della funzione

```
foo(n)
    if(n ≤ 2)
        return 1;
    else if(n > 321) // caso peggiore per n → ∞
        i = n/3;
        return (2 * foo(i) + (n*n*n) + 1);
    else
        return foo(n-1) + foo(n-2);
```

In quanto devo prendere in considerazione solo il caso peggiore (ovvero che $n \rightarrow \infty$), determino la complessità solo del ramo centrale del controllo *if*.

$$T(n) = 2T\left(\frac{n}{3}\right) + c$$

vi è un'unica chiamata ricorsiva, non 2

Applico il *teorema master* (vedi 3.3) e risolvo la ricorrenza:

$$\begin{aligned} a &= 1 \\ b &= 3 \\ f(n) &= c = \Theta(1) \\ d &= n^{\log_b a} = n^{\log_3 1} = n^0 = 1 = \Theta(1) \implies 2^{\text{°}} \text{ caso} \end{aligned}$$

da cui ricavo che $T(n) = \Theta(\log n)$.

8.3 Esercizio

Nell'ipotesi che $proc(m) = \Theta(\sqrt{m})$, si determini la complessità asintotica nel caso peggiore della funzione:

```
fun(int A[], int n)
    if(n<1)
        return 1;
    else
        t = fun(A,n/2);
        if(t > A[n])           // T(n/2)
            t = t + fun(A,n/2) // T(n/2)
        for(j=0 to n)          // n volte
            t = t + A[j] + proc(n) // complessità Θ(√n)
        return t;
```

calcolo la complessità iniziando ad aggiungere la complessità che trovo partendo dall'alto, ovvero:

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + \Theta(n\sqrt{n}) \\ &= 2T\left(\frac{n}{2}\right) + \Theta(n\sqrt{n}) \end{aligned}$$

Applico il *teorema master* (vedi 3.3) e risolvo la ricorrenza:

$$\begin{aligned} f(n) &= n\sqrt{n} = n^{\frac{3}{2}} \\ n^{\log_b a} &= n^{\log_2 2} = n \implies 3^{\circ} \text{ caso} \\ f(n) &= \Omega(n^{1+\varepsilon}) \\ \text{pongo } \varepsilon &= \frac{3}{2} - 1 = \frac{1}{2} \end{aligned}$$

arrivati a ciò verifico la condizione di regolarità (ovvero che la funzione sia decrescente):

$$\begin{aligned} af(n) &\leq cf(n) \quad \text{per } c < 1 \\ 2\frac{n}{2} \cdot \sqrt{\frac{n}{2}} &\leq c \cdot n\sqrt{n} \implies c \geq \frac{1}{\sqrt{2}} \end{aligned}$$

Dunque posso concludere per il 3 caso del teorema master che $T_{\text{fun}}(n) = \Theta(n \cdot \sqrt{n})$, ovvero che $f(n)$ predomina.

8.4 🏠 Esercizio per casa

Si valuti l'ordine di grandezza della complessità della funzione *foo* al crescere di n :

```
foo(int n)
    if(n ≤ 5)
        return 7;
    else
        if(n ≥ 75)      // caso peggiore
            for(i=1 to ⌊n/2⌋) // n/2 volte
                k = ⌊i/2⌋ // costante
            return (k * foo(k) + foo(n/2)); // c * T(n/2) + T(n/2)
        else
            return (foo(n+2) * foo(n-3) + n/2);
```

Determino:

$$T(n) = \frac{n}{2} + 2T\left(\frac{n}{2}\right)$$

e risolvo con il teorema master

$$a = 2$$

$$b = 2$$

$$f(n) = \frac{n}{2}$$

$$n^{\log_b a} = n^{\log_2 2} = n \implies 2^{\text{esponente}}$$

quindi la complessità è $T(n) = \Theta(n \log n)$.

8.5 ✎ Esercizio

Scrivere un algoritmo efficiente che dati in input due stringhe di lunghezza n , i cui caratteri appartengono all'insieme $\{a, b, \dots, x, y, z\}$, rappresentate mediante due array di caratteri $A[0 \dots n]$ e $B[0 \dots n]$, restituisca *true* se e solo se le due stringhe sono una l'anagramma dell'altra, e infine analizzarne al complessità.

Anagramma significa stesse lettere ma in ordine diverso. Il vincolo è di rimanere in tempo lineare.

```
int anagramma(char s1[], char s2[], int dim){
    int occ[26]; // vettore delle occorrenze
    int i;
    for(i=0; i<26; i++)
        occ[i] = 0; // svuoto il vettore delle occorrenze
    for(i=0; i<26; i++){
        // incremento l'indice della posizione di quel carattere di s1
        occ[s1[i] - 'a']++;
        // decremento l'indice della posizione di quel carattere di s2
        occ[s2[i] - 'a']--;
    }
    i=0;
    while(i<26 && occ[i] == 0) // controllo che il vettore
        i++; // delle occorrenze sia vuoto
    return (i == 26); // se sono arrivato alla fine, s1 e s2 sono anagrammi
}
```

Il tempo di questa implementazione è $T(n) = \Theta(n)$, quindi lineare.

9 Pila

2 Novembre 2016

Una *pila* è una sequenza di elementi di un certo tipo in cui è possibile aggiungere o togliere elementi soltanto ad un estremo (il *top* della sequenza). Il tipo di disciplina di accesso alla *pila* è *LIFO* - *Last In First Out*. Le operazioni definite sul tipo di dato *Stack* sono:

```
initStack() : Stack
// post: restituisce una pila vuota

stackEmpty(Stack S) : bool
// post: restituisce true se S è vuoto, false altrimenti

push(Stack S, Element e) : void
// post: aggiunge e come ultimo elemento di S

pop(Stack S) : Element
// pre: lo stack non è vuoto
// post: rimuove e restituisce l'ultimo elemento di S

top(Stack S) : Element
// pre: lo stack non è vuoto
// post: restituisce l'ultimo elemento di S senza rimuoverlo
```

9.1 Implementazione tramite array

Rappresentiamo uno *Stack* *S* con un array di dimensione *n*: *S*[1...*n*] e manteniamo un attributo *S.top* che è l'indice dell'ultimo elemento inserito.

```
initStack() : Stack
    s = allocate(n); // alloco con la tecnica del Raddoppiamento e Dimezzamento
    s.top = -1; // non vi sono elementi                               (anche con n=1)
    return S;

stackEmpty(Stack s)
    return s.top == -1;

push(Stack s, Element e)
    s.top++;
    s[S.top] = e;

pop(Stack s)
    s.top--;
    return s[s.top + 1];

top(Stack s) : Element
    return s[s.top];
```

Tutte queste implementazioni, supponendo che *allocate(n)* abbia tempo costante, sono anch'esse costanti, ovvero $\Theta(1)$.

9.2 🏠 Esercizio per casa- Implementare uno *Stack* utilizzando due code

L'idea è avere dentro lo stack due proprietà, *s.q1* e *s.q2* che rappresentano le due code. Tutti gli elementi vengono accodati in *q1*, e quando vado a prelevare sposto tutto da una pila all'altra per arrivare all'ultimo elemento. Una volta ottenuto l'ultimo elemento, inverto *q1* e *q2* per non dovr rispostare tutto indietro nuovamente.

```
push(Stack s, Element e)
    enqueue(s.q1, e)

pop(Stack s)
    e = dequeue(s.q1)
    while(!queueEmpty(s.q1))
        enqueue(e, s.q2)
        e = dequeue(s.q1)
    q = s.q1;
    s.q1 = s.q2;
    s.q2 = q;
    return e;
```

10 Coda

Una *coda* è una sequenza di elementi di un certo tipo in cui è possibile aggiungere elementi ad un estremo (*tail*) e rimuovere dall'estremo opposto (*head*). Il tipo di disciplina di accesso alla *coda* è *FIFO* - *First In First Out*. Le operazioni definite sul tipo di dato *Queue* sono:

```
initQueue() : Queue
// post: restituisce una coda vuota

queueEmpty(Queue q) : bool
// post: restituisce true se q è vuoto, false altrimenti

enqueue(Queue q, Element e) : void
// post: aggiunge e come ultimo elemento di q

dequeue(Queue q) : Element
// pre: la coda non è vuota
// post: rimuove e restituisce il primo elemento di q

first(Queue q) : Element
// pre: la coda non è vuota
// post: restituisce il primo elemento di q senza rimuoverlo
```

10.1 Implementazione tramite array circolari

Rappresentiamo una Coda Q con un array di dimensione n e manteniamo due attributi $Q.head$ che è l'indice della posizione da cui estrarre l'elemento, e $Q.tail$ che è l'indice della posizione in cui inserire un nuovo elemento. Con l'implementazione tramite array circolari, una volta raggiunta la fine del vettore, si ritorna alla prima posizione, in ordine circolare.

- Inizialmente $Q.head = Q.tail = 0$
- In ogni istante gli elementi della coda si trovano nel segmento $Q.head, Q.head + 1, \dots, Q.tail - 1$
- In ogni istante si garantisce che la coda abbia capacità massima di $n - 1$ elementi
- Una coda è vuota se $Q.head = Q.tail$
- Una coda è piena se $Q.head = Q.tail \bmod n$

```

initQueue()
    q = allocate(n);
    q.head = 0;
    q.tail = 0;
    return q;

queueEmpty(Queue q)
    return q.head == q.tail;

succ(int i)
    if(i == n)
        return i % n; // ==> return 0;
    else
        return i + 1;

// pre: la coda non è piena
enqueue(Queue q, Element e)
    q[q.tail] = e;
    q.tail = succ(q.tail);

// pre: la coda non è vuota
dequeue(Queue q)
    x = q[q.head];
    q.head = succ(q.head);
    return x;

// pre: la coda non è vuota
first(Queue q)
    return q[q.head];

```

Tutte queste implementazioni, supponendo che $allocate(n)$ abbia tempo costante, sono anch'esse costanti, ovvero $\Theta(1)$.

10.2 Esercizio

Implementare una coda utilizzando due pile ($q.p1$ e $q.p2$).

```
enqueue(Queue q, Element e)
    push(q.p1, x);

dequeue(Queue q)
    if(stackEmpty(q.p2))
        while(!stackEmpty(q.p1))
            x = pop(q.p1);
            push(q.p2, x)
    return pop(q.p2);
```

L'implementazione di `dequeue` è $O(n)$, ma in tempo ammortizzato è costante, infatti solo una piccola percentuale di perazioni richiederanno lo spostamento degli elementi da $p1$ a $p2$.

10.3 Esercizio - Invertire una coda

Invertire una coda in modo ricorsivo utilizzando l'interfaccia della coda e determinare la complessità dell'implementazione.

```
invert(Queue q)
    if(!queueEmpty(q))
        x = dequeue(q);
        inverti(q);
        enqueue(q, x);
```

La complessità è

$$T(n) = T(n - 1) + c = n \cdot c + d = \Theta(n)$$

11 Liste

9 Novembre 2016

Una lista è una sequenza di elementi di un certo tipo in cui è possibile aggiungere accedere e cancellare elementi in posizioni arbitrarie. Le operazioni permesse sul tipo *lista* sono:

```
creaLista() -> List
// post: restituisce una lista vuota

listEmpty(Lista l) -> boolean
// post: restituisce true se la lista è vuota, false altrimenti

search(List l, Item k) -> Node
// restituisce il primo elemento con valore k se esiste, null altrimenti

insert(List l, Node x)
// post: inserisce un elemento x nella lista

delete(List l, Node x)
// post: cancella l'elemento x dalla lista l
// pre: x appartiene alla lista
```

11.1 Realizzazione con struttura collegata

Una lista concatenata può essere *semplice* o *doppia*.

11.1.1 Lista semplice

Una lista ha un attributo *l.head* che è un puntatore al primo elemento nella lista. Una lista è vuota quando *L.head == null*. Un oggetto *x* che appartiene a questa lista, ha almeno le seguenti proprietà:

- *x.key*: il valore o chiave
- *x.next*: un puntatore all'elemento successivo nella lista

11.1.2 Lista doppiamente concatenata

Ha le stesse proprietà della lista semplice, ma un oggetto *x* che appartiene a questa lista, ha almeno le seguenti proprietà:

- *x.key*: il valore o chiave
- *x.next*: un puntatore all'elemento successivo nella lista
- *x.prev*: un puntatore all'elemento precedente nella lista

Una lista può inoltre:

- avere un puntatore all'ultimo elemento (*l.tail*)
- essere circolare: il puntatore *next* dell'ultimo elemento punta alla testa della lista
- avere un nodo sentinella: per gestire meglio i casi limite (l'informazione relativa la chiave è indefinita)
- avere i dati memorizzati in ordine oppure no
- ammettere duplicati oppure no

11.1.3 Implementazione lista doppia

metodo *search*:

```
search(List l, Item k)
    x = l.head;
    while(x != null && x.key != k)
        x = x.next;
    return x;
```

11.2 Invariante

Un invariante è un'asserzione vera prima, dopo e ad ogni iterazione del ciclo. Per dimostrare un invariante sono necessarie tre fasi:

1. inizializzazione (vera prima della prima iterazione del ciclo)
2. conservazione (se è vera prima di un iterazione, rimane vera prima della successiva iterazione).
 $INV \wedge Guardia \rightarrow INV$ (dopo l'esecuzione del corpo del ciclo)
3. conclusione: quando il ciclo termina l'invariante fornisce un utile proprietà che ci aiuta a dimostrare che l'algoritmo è corretto. $INV \wedge \neg Guardia \rightarrow (asserzione\ che\ rappresenta\ la\ post\ condizione)$

Per assicurarsi che il nostro ciclo termini dobbiamo definire una *funzione di terminazione*. Una funzione di terminazione è una funzione a valori naturali che decresce strettamente ad ogni iterazione del ciclo.

→ **Esempio** : per la funzione *search* vista sopra, la funzione di terminazione è numero degli elementi della lista non ancora vittitati, mentre l'invariante è "gli elementi da *l.head* a *x* escluso non contengono *k*" (altrimenti mi sarei fermato prima). Dimostriamo le 3 proprietà dell'invariante:

1. inizializzazione: all'inizio $x = l.head$, e quidni INV è vacuamente vero perché non ci sono elementi
2. conservazione: $INV \wedge (x \neq null \wedge x.key \neq k) \implies INV[x.next/x] \equiv$ gli elementi da *L.head* da *x.next* escluso hanno chiave diversa da *k*. È vera per ipotesi perché INV è vero \equiv gli elementi da *l.head* a *x* escluso hanno chiave diversa da *k* e $x.key \neq k$ per la guardia.
3. conclusione: il ciclo termina per 2 ragioni:
 - $x = null$: in questo caso l'invariante assicura che *k* non è presente in tutta la lista
 - $x \neq null \wedge x.key == k$: l'invariante assicura che *k* non è presente prima di *x*, dunque *x* è la prima occorrenza dell'elemento con chiave *k*

11.3 Realizzazione lista doppia

Andiamo ad implementare una lista doppiamente concatenata, prima senza l'ausilio del nodo snetinella e successivamente con l'aggiunta di quest'ultimo.

11.3.1 Senza nodo sentinella

```
insert(List l, Node x)
    x.next = l.head
    if(l.head != null)
        l.head.prev = x;
    l.head = x;
    x.prev = null;

delete(List l, Node x)
    if(x.prev != null)
        x.prev.next = x.next;
    else
        l.head = x.next;
    if(x.next != null
        x.next.prev = x.prev;
    rimuovi(x);
```

11.3.2 Con nodo sentinella

Una sentinella è un oggetto fittizio che consente di semplificare le condizioni di contorno. Supponiamo di fornire alla lista l un elemento che si chiama $l.NULL$ che rappresenta la costante *null* ma ha tutti i campi degli altri elementi della lista. La sentinella è posta fra la testa e la coda della lista. Nella lista viene quindi eliminato il puntatore $l.head$, che viene sostituito da $l.NULL.next$. Una lista vuota è formata solo dal nodo sentinella.

```
search(List l, Item k)
    x = l.NULL.next;
    while( x.key != k)
        l.NULL.key = k;
        x = x.next;
    return x;

insert(List l, Node x)
    x.next = l.NULL.next;
    l.NULL.next.prev = x;
    l.NULL.next = x;
    x.prev = l.NULL;

delete(List l, Node x)
    x.prev.next = x.next;
    x.next.prev = x.prev;
    rimuovi(x);
```

In questo modo nell'implementazione, le condizioni al contrario non necessitano di particolari attenzioni.

11.3.3 Realizzazione con array

10 Novembre 2016

Ora andremo a realizzare una lista doppia con un insieme di array: ogni proprietà di un nodo nella lista sarà memorizzato nella stessa posizione ma in array differenti (un array per ogni attributo dei nodi). In

next	-1	0			1
key	32	20			13
prev	1	4			-1
indice	0	1	2	3	4

questa rappresentazione una variabile contiene l'indice della testa della lista, mentre la costante NULL viene rappresentata da -1. In tabella (che rappresenta una lista quindi) vi sono 3 elementi: $l = 4$, ovvero l'indice del primo elemento della lista è contenuto nell'indice 4.

Inserimento Per allocare un nuovo elemento dobbiamo aver tenuto traccia delle posizioni libere. Queste posizioni libere saranno memorizzate in una lista semplice chiamata *freeList*.

```
allocateObject(){
    if(free == null)
        printf("errore\u00f9 spazio esaurito");
    else
        x = free;
        free = next[free];
        return x;
}
```

Cancellazione Per la cancellazione andremo ad aggiornare l'indice della prossima posizione libera.

```
freeObject(x){
    next[x] = free;
    free = x;
}
```

12 Alberi

Un albero (radicato) è una coppia $T = (N, A)$ dove N è un insieme finito di nodi fra cui si distingue un nodo r detto **radice**, e $A \subseteq N \times N$ è un insieme di coppie di nodi dette archi.

In un albero ogni nodo v eccetto la radice ha esattamente un genitore (o padre) tale che la coppia data da $(u, v) \in A$.

Un albero è un particolare tipo di grafo: è un grafo che risulta essere connesso, aciclico e non orientato.

Un nodo u può avere zero o più figli v tali che $(u, v) \in A$. Il numero di figli di un nodo è detto **grado del nodo**. Un nodo senza figli è detto **foglia** (o nodo esterno).

Un nodo non foglia è un **nodo interno**. Se due nodi hanno lo stesso padre sono fratelli. Camminando da un nodo u ad un nodo u' in T si forma una sequenza di nodi $\langle n_0, n_1, \dots, n_k \rangle$ tali che $U = n_0$, $U' = n_k$ e $\langle n_{i-1}, n_i \rangle \in A$ per $i = 1, \dots, k$. Questa sequenza viene detta cammino. La lunghezza del cammino è il numero di archi nel cammino o il numero di nodi che formano il cammino stesso - 1.

Sia x un nodo in un albero radicato T con radice r : allora un qualsiasi nodo y in un cammino che parte dalla radice r e arrivo fino al nodo x è detto **antenato** di x . Se y è antenato di x , allora x è discendente di y . Nota bene: ogni nodo è antenato e discendente di se stesso.

Se y è un antenato di x e $x \neq y$ allora y è un **antenato proprio** di x , e x è un **discendente proprio** di y . Il sottoalbero con radice di x è l'albero indotto dei discendenti di x con radice x .

La **profondità** di un nodo x è la lunghezza del cammino dalla radice ad x .

Un **livello** di un albero è costituito da tutti i nodi che stanno alla stessa profondità.

L'**altezza di un nodo** x è la lunghezza del più lungo cammino che scende da x ad una foglia.

L'**altezza di un albero** è l'altezza della sua radice. L'altezza è anche uguale alla profondità massima di un qualsiasi nodo dell'albero.

12.1 Alberi binari

Un albero binario è uno speciale albero nel quale ogni nodo ha al più due figli. Un albero binario è definito in modo ricorsivo:

- Un albero vuoto è un albero binario
 - Un albero formato da:
 - un nodo radice
 - un albero binario detto *sottoalbero sinistro* della radice
 - un albero binario detto *sottoalbero destro* della radice
- è un albero binario.

12.2 Alberi k-ari

Un albero k-ario è un albero in cui i figli di un nodo sono etichettati con interi positivi distinti e le etichette maggiori di k sono assenti.

Un albero binario infatti è un albero k-ario con $k = 2$.

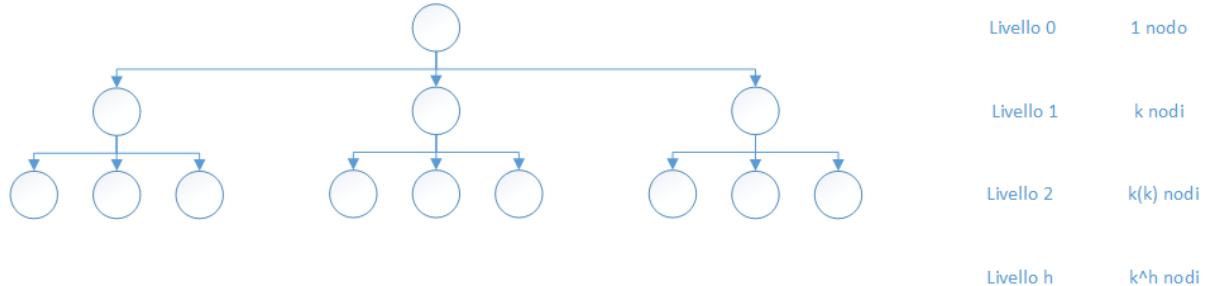
12.3 Completezza di un albero

Una albero k -ario completo è un albero k -ario in cui tutte le foglie hanno la stessa profondità e tutti i nodi interni hanno grado k .

12.3.1 Esercizio

Trovare il numero di foglie e il numero di nodi interni di un albero k -ario completo di altezza h .

Cominciamo con il disegnare l'albero, per ipotizzare il numero di foglie. Sapendo che è un albero completo, ogni nodo non foglia avrà esattamente k figli. Disegniamo con $k = 3$:



Abbiamo quindi ipotizzato che il numero di foglie sia

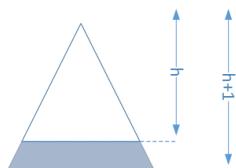
$$\#foglie(h) = k^h,$$

dobbiamo ora dimostrarlo per induzione. Il caso base risulta essere $h = 0$, quindi

$$\#foglie(h = 0) = k^h = k^0 = 1 \quad \checkmark$$

Il caso base risulta essere vero: l'albero è costituito solo dalla radice, che è l'unica foglia. Devo applicare l'ipotesi induttiva: assumo vero per h e dimostro per $h + 1$, quindi per alberi di altezza $h + 1$.

Un albero di altezza $h + 1$ è definito come



Il numero di nodi di profondità h sono k^h per ipotesi induttiva. Allora, perché l'albero è completo, ognuno di questi nodi ha esattamente k figli. Dunque

$$\#foglie(h + 1) = k^{h+1}$$

Il numero di nodi interni è:

$$\sum_{i=0}^{h-1} k^i = \frac{k^{h-1+1} - 1}{k - 1} = \frac{k^h - 1}{k - 1} \quad \text{con } k \neq 1$$

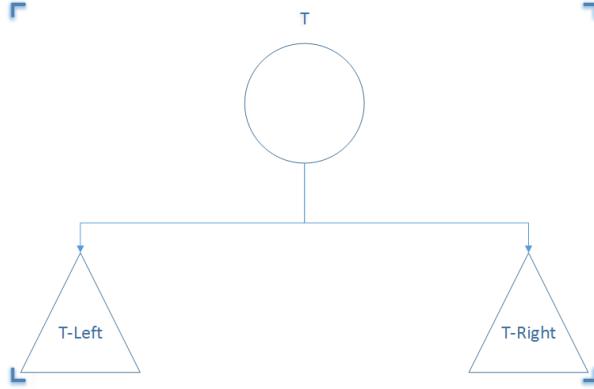
12.3.2 Esercizio

Dimostrare per induzione che il numero di foglie di un albero binario completo non vuoto con n nodi è $\frac{n+1}{2}$.

Inizio con la verifica del caso base, ovvero $n = 1$:

$$\#foglie(1) = \frac{1+1}{2} = 1 \quad \checkmark$$

Ora applico il passo induttivo: assumiamo che la proprietà sia vera per alberi con il numero di nodi minore di n , e lo dimostriamo per alberi con n nodi.



Sia T l'albero completo con n nodi radicato in r e con sottoalberi T_{left} e T_{right} . Poiché T è completo:

1. $\#nodi(T_{\text{left}}) = \#nodi(T_{\text{right}}) = \frac{n-1}{2}$
2. T_{left} e T_{right} sono completi

Per ipotesi induttiva allora:

$$\#foglie(T_{\text{left}}) = \frac{\frac{n-1}{2} + 1}{2} = \frac{n+1}{4}$$

e quindi

$$\#foglie(T) = \#foglie(T_{\text{left}}) + \#foglie(T_{\text{right}}) = 2 \cdot \frac{n+1}{4} = \frac{n+1}{2}$$

12.4 Implementazione della struttura

16 Novembre 2016

Realizzazione di un albero i cui dati sono un insieme di nodi (di tipo *Node*) e un insieme di archi. Le operazioni definite su questa struttura dati sono:

```
nextTree() : Tree
// post: restituisce un albero vuoto

treeEmpty(Tree t) : boolean
// post: restituisce true se l'albero t è vuoto

padre(Tree t, Node v) : Node
// pre: v appartiene all'albero
// post: restituisce il padre del nodo v in t, oppure null se v è la radice

figli(Tree t, Node v):List of Node
// pre: v appartiene all'albero
// post: restituisce i figli del nodo v
```

12.4.1 Tramite array

Vettore dei padri Sia $T = (N, A)$ un albero di n nodi numerati da 1 a n . Utilizziamo un vettore P di dimensione n le cui celle contengono coppie (info, parent).

- $\forall v \in [1, \dots, n]$, $P[v].info$ è il contenuto del nodo v
- $P[v].parent = u$ se vi è un arco $u, v \in A$
- se v è la radice $P[v].parent = 0$

Lo spazio occupato da questa implementazione per memorizzare un albero di n nodi è $\Theta(n)$.

```
// T(n) = Theta(1)
padre(Tree p, Node v)
    if(p[v].parent == 0)
        return null;
    else
        return p[v].parent;

// T(n) = Theta(n)
figli(Tree p, Node v)
    l = creaLista();
    for(i=1 to n)
        if(p[i].parent == v)
            inserisci(i, l);
    return l;
```

Vettore posizionale è utile per andare a memorizzare alberi k-ari completi con $k \geq 2$. Ogni nodo ha una posizione prestabilita nella struttura. Sia $T = (N, A)$ un albero k-ario completo con n nodi e sia P il vettore posizionale di dimensione n tale che $P[v]$ contiene l'informazione associata al nodo v .

- 0 è la posizione della radice
- l'i-esimo figlio di v è in posizione $(k \cdot v + 1 + i)$ per $i \in [0, k - 1]$
- il padre del nodo f (con $f \neq 0$) è in posizione

$$\begin{aligned} k \cdot v + 1 &\leq f \leq k \cdot v + 1 + k - 1 \\ k \cdot v &\leq f - 1 \leq k \cdot v + k - 1 \\ v &\leq \frac{f - 1}{k} \leq v + \frac{k - 1}{k} < v + 1 \\ v &\leq \frac{f - 1}{k} < v \\ v &= \left\lfloor \frac{f - 1}{k} \right\rfloor \quad \text{con } k \neq 0 \end{aligned}$$

```
// T(n) = Theta(1)
padre(Tree p, Node)
    if(v == 0)
        return null;
    else
        return Math.floor((v-1)/k) // parte intera inferiore

// T(n) = O(k)
figli(Tree p, Node v)
    l = creaLista();
    if(k*v+1 >= n) // un nodo foglia, in quanto
        return l;      // il suo primo figlio non è presente nell'array
    for(i=0 to k-1)
        inserisci(k*v+1+i, l);
    return l;
```

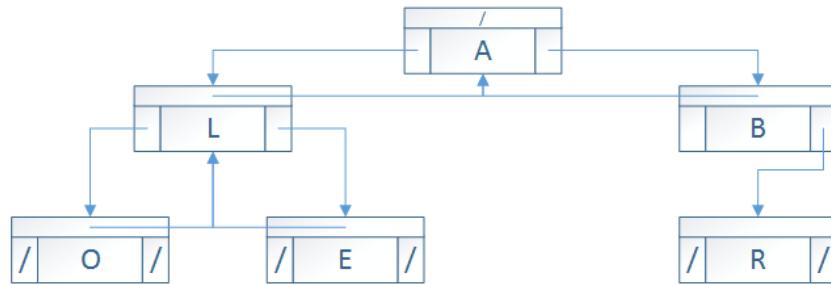
Lo svantaggio di questa rappresentazione sono le cancellazioni e gli inserimenti.

12.4.2 Tramite strutture collegate

In questa rappresentazione ogni nodo ha un puntatore al padre, uno al figlio sinistro e uno al figlio destro:

- $x.key$ = contenuto informativo
- $x.p$ = puntatore al padre
- $x.left$ = puntatore al figlio sinistro
- $x.right$ = puntatore al figlio destro

Inoltre l'albero possiede una proprietà $root$ che è un puntatore al primo nodo (radice) dell'albero. Lo spazio richiesto è $\Theta(n \cdot k)$. Se k è una costante, lo spazio diventa lineare ($\Theta(n)$). Con questa rappresentazione ogni nodo ha k figli (in quanto stiamo parlando di un albero completo), e se ogni nodo ha pochi figli con k grande diventa svantaggioso in termini di spazio.



```

// T(n) = Theta(1)
padre(Tree t, Nodo v)
    return v.p;

// T(n) = Theta(1)
figli(Tree t, Node v)
    l = creaLista();
    if(v.left != null)
        inserisci(v.left, l);
    if(v.right != null)
        inserisci(v.right, l);
    return l;
  
```

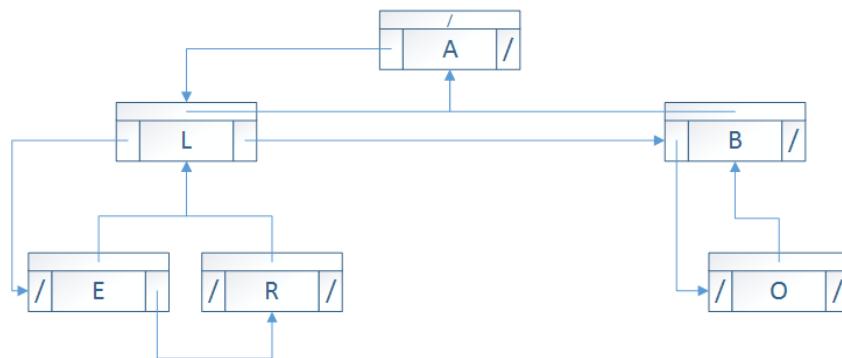
Se il numero di figli non è noto a priori, è possibile associare a ciascun nodo una lista di puntatori ai suoi figli, con un conseguente spreco di memoria. Per ovviare a questi problemi, la rappresentazione più adeguata per rappresentare un albero è la **rappresentazione binarizzata**.

12.5 Rappresentazione binarizzata

In questa rappresentazione ogni nodo ha un puntatore al padre, uno al figlio più a sinistra e uno al fratello a destra:

- $x.key$ = contenuto informativo
- $x.p$ = puntatore al padre
- $x.left-child$ = puntatore al figlio più a sinistra del nodo x
- $x.right-sibling$ = puntatore al fratello di x immediatamente a destra

Questa rappresentazione costa $\Theta(n)$ in memoria.



```

// T(n) = Theta(grado(v)) = Θ(n)
figli(Tree t, Node v)
    l = creaLista();
    iter = v.left-child;
    while(iter != null)
        inserisci(iter, l),
        iter = iter.right-sibling;
    return l;
  
```

12.6 Visite di alberi

Una visita generica di un albero si esegue nel seguente modo:

```
visitaGenerica(Node r)
    S = {r}
    while(S != ∅)
        estrai un nodo u da S
        visita(u);
        S = S ∪ {figli di u}
```

Teorema: l'algoritmo di visita, applicato alla radice di un albero con n nodi termina in $O(n)$ iterazioni. Lo spazio usato per memorizzare S è $O(n)$. La nostra ipotesi è che l'inserimento/cancellazione siano eseguiti in $\Theta(n)$.

Ogni nodo verrà inserito ed estratto da S una sola volta perché in un albero non si può tornare ad un nodo a partire dai suoi figli procedendo di figlio in figlio. Quindi le iterazioni del ciclo while saranno al più $O(n)$. Poiché ogni nodo compare al più una volta nella mia struttura S , lo spazio richiesto è $O(n)$.

12.6.1 Visita in profondità

Versione iterativa per alberi binari (nodi con proprietà *left* e *right*)

```
// T(n) = lineare = Theta(n)
// DepthFirstSearch
visitaDFS(Node r)
    Stack S
    Push(S,r);
    while(!stackEmpty(S))
        u = pop(S);
        if(u != null)
            visita(u);
            push(S,u.right);
            push(S,u.left);
```

Versione ricorsiva per alberi binari (nodi con proprietà *left* e *right*)

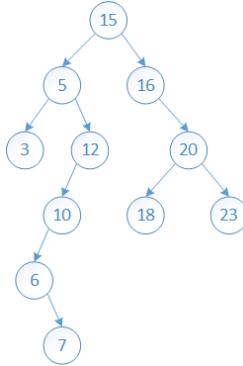
```
// T(n) = lineare = Theta(n)
visitaDFSRic(Node r)
    if(r != null) // albero non vuoto
        visita r;
        visitaDFSRic(r.left);
        visitaDFSRic(r.right);
```

12.6.2 Visita in ampiezza o a livelli

```
// BreadthFirstSearch (BFS). Visita a livelli
visitaBFS(Node r)
    Queue q;
    enqueue(q, r);
    while(!queueEmpty(q))
        u = dequeue(q);
        if(u != null)
            visit(u);
            enqueue(q,u.left);
            enqueue(q,u.right);
```

12.6.3 Tipi di visita

A seconda della posizione dell'operazione di visita del nodo abbiamo 3 tipi di visita diversi:



1. Visita in **preordine**: si visita prima la radice, poi si effettuano le chiamate ricorsive prima sul figlio sinistro e poi sul destro. Nel caso dell'albero in esempio il risultato sarebbe: $\langle 1, 2, 4, 5, 6, 3 \rangle$.

```
visita r;
visitaDFSric(r.left);
visitaDFSric(r.right);
```

2. Visita **simmetrica**: si visita il figlio sinistro, la radice e il figlio destro: $\langle 4, 2, 6, 5, 1, 3 \rangle$.

```
visitaDFSric(r.left);
visita r;
visitaDFSric(r.right);
```

3. Visita in **post-ordine**: si visita il figlio sinistro, il figlio destro e la radice: $\langle 4, 6, 5, 2, 3, 1 \rangle$.

```
visitaDFSric(r.left);
visitaDFSric(r.right);
visita r;
```

Tutte queste visite hanno complessità uguale a $\Theta(n)$.

12.6.4 Teorema

Se x è la radice di un sottoalbero di n nodi, la chiamata *visitaDFSric*, applicata al nodo x , richiede tempo $\Theta(n)$.

Dimostrazione Sia $T(n)$ il tempo richiesto dalla procedura quando è chiamata sulla radice di un sottoalbero di n nodi. Poiché *visitaDFSric* visita tutti gli n nodi del sottoalbero su cui è invocata, questa proprietà rappresenta un limite inferiore sotto al quale la complessità non può andare, quindi $T(n) = \Omega(n)$.

Ora devo trovare il limite superiore, che ipotizzo essere $O(n)$. Assumo che il sottoalbero sinistro abbia k nodi, con $k < n$, e il sottoalbero destro $n - k - 1$ nodi, quindi:

$$T(n) = \begin{cases} \text{costante } c & \text{se } n = 0 \\ \underbrace{T(k)}_{\substack{\text{sottoalbero destro} \\ \text{tutti i nodi}}} + \underbrace{T(n - k - 1)}_{\substack{\text{sottoalbero sinistro}}} + d & \text{se } n > 0 \end{cases}$$

Devo dimostrare che

$$T(n) = an + b$$

è una soluzione della ricorrenza $T(k) + T(n - k - 1) + d$. Per induzione, il caso base è $n = 0$:

$$T(0) = a \cdot 0 + b = b$$

suppongo

$$b = c$$

e formulo l'ipotesi induttiva: assumiamo vero che per tutti gli $m < n$, $T(m) = am + b$, e lo dimostro per n (con $n > 0$):

$$T(k) + T(n - k - 1) + d$$

applico l'ipotesi induttiva in quanto $k < n$ (non c'è la radice) e $n - k - 1 < n$

$$ak + b + a(n - k - 1) + b + d = a(k + n - k - 1) + 2b + d = an - a + 2b + d$$

che devo imporre uguale a $an + b$, quindi

$$an - a + 2b + d = an + b$$

da cui

$$a = b + d$$

ma avendo imposto $b = c$ all'inizio ottengo

$$a = c + d$$

Quindi

$$T(n) = an + b = \underbrace{(c + d)}_a \cdot n + \underbrace{b}_{b=c}$$

che è esattamente la funzione lineare della nostra occorrenza.

12.7 Esercizi

12.7.1 Esercizio - verificare completezza albero binario

Dato un albero, scrivere una funzione che ritorna *true* se l'albero è binario completo, *false* altrimenti e ritorni inoltre l'altezza dell'albero stesso. La firma della funzione è la seguente:

```
verificaCompleto(Node u) : <bool, int>
```

Un albero binario completo ha due proprietà:

1. ogni nodo interno ha n figli
2. tutte le foglie sono allo stesso livello

L'implementazione diventa quindi:

```
verificaCompleto(Node u)
    if(u == null)
        return <true, -1>; // -1 = albero vuoto, 0 = solo radice
    <completoSx, hSx> = verificaCompleto(u.left);
    <completoDx, hDx> = verificaCompleto(u.right);
    // verifico siano completi e abbiano stesso livello
    completo = completoSx && completoDx && (hSx == hDx);
    // possono avere altezze diverse a cui devo aggiungere la radice
    h = max(hSx, hDx) + 1;
    return <completo, h>;
```

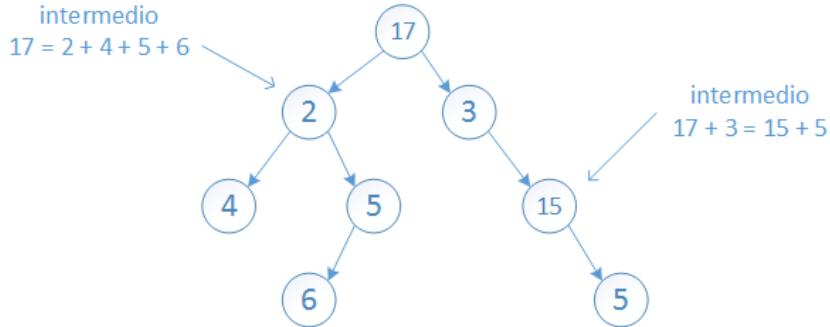
La complessità è

$$T(n) = \begin{cases} c & \text{se } n = 0 \\ T(k) + T(n - k - 1) + d & \text{se } n > 0 \end{cases}$$

che equivale ad $\Theta(n)$.

12.7.2 Esercizio - numero di nodi intermedi

Un nodo u di un albero binario si dice *intermedio* se la somma delle chiavi nei sottoalberi contenuti nei nodi del sottoalbero di cui u è radice, è uguale alla somma delle chiavi contenute nei nodi sul percorso che collega il nodo u alla radice, e da questo percorso u deve essere escluso.



Definito il tipo *Node* nel seguente modo:

```

typedef struct node {
    int key,
    struct node* left,
    struct node* right
} *Node;

```

si definisca la funzione che determini il numero di nodi intermedi di un albero:

```

int intermedi(Node u, int sum, int* sommaChiavi){
    int risSx, risDx, sommaChiaviSx, sommaChiaviDx;
    if(u == null){
        *sommaChiavi = 0;
        return 0;
    }
    risSx = intermedi(u.left, sum + u.key, &sommaChiaviSx);
    risDx = intermedi(u.right, sum + u.key, &sommaChiaviDx);
    *sommaChiavi = sommaChiaviSx + sommaChiaviDx + u.key;
    if(sommaChiavi == sum)
        return 1 + risSx + risDx;
    else
        return risSx + risDx;
}

```

La complessità risulta essere

$$T(n) = \begin{cases} c & \text{se } n = 0 \\ T(k) + T(n - k - 1) + d & \text{se } n > 0 \end{cases} = \Theta(n)$$

12.8 Alberi bilanciati

Un albero è bilanciato se

$$h = O(\log n)$$

Un albero binario completo è sempre bilanciato (in quanto completo). Viceversa un albero bilanciato non implica sia completo.

12.8.1 Esercizio : Costruire ricorsivamente un albero bilanciato

Dato un array v di n interi, scrivere una funzione efficiente che costruisca ricorsivamente un albero binario bilanciato tale che $v[i]$ sia l' $(i+1)$ -esimo campo $v.key$ in ordine di visita posticipata (*postordine*). La firma della funzione è la seguente:

```
Node costruisci(int v[], int dim);
```

Per scrivere questa funzione abbiamo bisogno di alcuni parametri che non abbiamo in questa firma. Andiamo quindi a definire una nuova funzione:

```
Node costruisciAux(int v[], Node padre, int inf, int sup){
    Node r;
    int med;
    if(inf > sup)
        return null;
    else{
        r = malloc(sizeof(struct node));
        r->key = v[sup];
        r->p = padre;
        med = (inf+sup)/2;
        r->left = costruisciAux(v,r,inf,med-1);
        r->right = costruisciAux(v,r,med,sup-1);
        return r;
    }
}
```

arrivati a questo punto richiamo la funzione iniziale:

```
Node costruisci(int v[], int dim){
    return costruisciAux(v,null,0,dim-1);
}
```

La complessità risulta essere

$$T(n) = \begin{cases} c & \text{se } n = 0 \\ 2T\left(\frac{n}{2}\right) + d & \text{se } n > 0 \end{cases}$$

risolvo con il teorema master:

$$\begin{aligned} n^{\log_a b} &= n^{\log_2 2} = n \\ f(n) &= d = \Theta(1) \Rightarrow 1^\circ \text{ caso del teorema} \\ f(n) &= O(n^{1-\varepsilon}) \text{ con } \varepsilon > 0 \end{aligned}$$

pongo $\varepsilon = 1$ da cui ricavo

$$T(n) = \Theta(n)$$

12.8.2 Esercizio : Stampare le chiavi di un livello k

Dato un albero binario T e un intero $k \geq 0$, stampare le chiavi contenute in T a livello k procedendo da sinistra a destra.

```
void stampaLivello(Node v, int k){
    if(u == null)
        return;
    else{
        if(k == 0)
            printf("%d", v.key);
        else{
            stampaLivello(v.left, k-1);
            stampaLivello(v.right, k-1);
        }
    }
}
```

La complessità è $O(n)$, dove n è il numero di nodi dell'albero, poiché il caso peggiore risulta essere $k = h(T)$. Se volessimo invece esprimere la complessità in funzione di k il caso peggiore risulta essere quando T è completo. In questo caso la complessità (sempre in funzione di k) risulta essere:

$$\sum_{i=0}^k 2^i = \frac{2^{k+1} - 1}{2 - 1} \approx 2^k$$

ovvero esponenziale.

12.8.3 Esercizio : Dimezzare le chiavi sui livelli pari di un albero

Sia T un albero generale i cui nodi hanno chiavi intere e proprietà key , $left-child$, $right-sib$. Scrivere una funzione ricorsiva che trasforma T dimezzando i valori di tutte le chiavi sui livelli pari dell'albero.

```
void trasforma(Node u){
    if(u != null) {
        u.key = u.key / 2;
        trasforma(u.right-sib);
        iter = u.left-child;
        while(iter){
            trasforma(iter.left-child);
            iter = iter-right-sib;
        }
    }
}
```

La complessità risulta essere $O(n)$.

12.9 Alberi binari di ricerca

Un albero binario di ricerca è un albero binario che soddisfa la seguenti proprietà:

- a) Sia x un nodo in un albero binario di ricerca. Se y è un nodo nel sottoalbero sinistro di x , allora $y.key \leq x.key$
- b) Se y è un nodo del sottoalbero destro di x , allora $y.key \geq x.key$

La proprietà di ricerca dell'albero binario di ricerca consente di elencare in ordine crescente le chiavi dell'albero stesso, visitandolo in ordine simmetrico.

Esercizio per casa Dimostrare per induzione sul numero di nodi dell'albero che le chiavi sono ordinate in modo crescente.

12.9.1 Operazioni

Le operazioni definite su un albero binario di ricerca sono:

- a) *treeSearch*
- b) *treeMinimum*
- c) *treeSuccessor*
- d) *treeInsert*
- e) *treeDelete*

a) *treeSearch* Ricerca un nodo con chiave k :

```
// post: restituisce un nodo con chiave k se esiste, null altrimenti
// complessità: O(T(h)), ovvero O(altezza albero)
Node treeSearch(Node x, Elemt k)
    if(x == null || x.key == k)
        return x;
    else
        if(x.key < k)
            return treeSearch(x.left, k);
        else
            return treeSearch(x.right, k);
```

I nodi incontrati durante la ricorsione formano un cammino verso il basso dalla radice dell'albero, quindi il tempo di esecuzione è $O(h(T))$. Se l'albero è bilanciato allora $T(n) = O(\log n)$. Se l'albero è fortemente sbilanciato allora $T(n) = O(n)$.

Esiste anche una versione iterativa del *treeSearch*:

```
// post: restituisce un nodo con chiave k se esiste, null altrimenti
Node treeSearch(Node x, Elemt k)
    while(x != null && x.key != k){
        if(x.key > k)
            x = x.left;
        else
            x = x.right;
    }
    return x;
```

b) ***treeMinimum*** Ricerca il nodo con chiave minore nel sottoalbero più in basso a sinistra.

```
// pre: x ∈ T
// post: restituisce il nodo con chiave minore nel sottoalbero radicato in x
// complessità: O(n)
Node treeMinimum(Node x){
    while(x != null)
        x = x.left;
    return x;
}
```

c) ***treeSuccessor*** Dato un nodo n in un albero binario di ricerca, il successore di x è il nodo che segue x nell'ordine stabilito da una visita simmetrica. Se tutte le chiavi sono distinte il successore di un nodo x è il nodo con la più piccola chiave maggiore di quella di x . Si distinguono due casi:

- 1) x ha un figlio destro: il successore di x è il minimo del sottoalbero destro di x
- 2) x non ha un figlio destro: il successore di x , se esiste, è l'antenato più prossimo di x il cui figlio sinistro è anche antenato di x . Per trovarlo si risale da x verso la radice, fino ad incontrare la "prima svolta a destra"

```
// complessità: O(h(T))
treeSuccessor(Node x){
    if(x.right != null)
        return treeMinimum(x.right); // O(h)
    else{
        y = x.p;
        while(y != null && x == y.right){ // O(h)
            x = y;
            y = y.p;
        }
        return y;
    }
}
```

d) ***treeInsert*** Tree è il tipo albero e ha un campo $root$ che contiene il nodo radice. Il nodo z da inserire è inizializzato con la chiave e $z.left = z.right = \text{null}$

```
// complessità: O(h(T))
treeInsert(Tree t, Node z){
    y = null;
    x = t.root;
    while(x != null){
        y = x;
        if(z.key < x.key)
            x = x.left;
        else
            x = x.right;
    }
    z.p = y;
    if(y == null)
        t.root = z;
    else
        if(z.key < y.key)
            y.left = z;
        else
            y.right = z;
}
```

e) *treeDelete* elimina il nodo z contenuto nell'albero t . Si distinguono 3 casi:

- 1) Se z non ha figli, modifichiamo il padre di z affinché punti anziche a z a *NULL*.
- 2) Se z ha un unico figlio, stacchiamo z creando un collegamento tra il padre e il figlio di z .
- 3) Se z ha due figli, troviamo il successore y di z , che si troverà nel sottoalbero destro di z , e facciamo in modo che y assuma la posizione di z nell'albero.

Per realizzare questa funzione utilizzeremo una procedura ausiliaria *trasplant* che, dato un albero t e due nodi u e v , sostituisce il sottoalbero con radice u con il sottoalbero di radice v .

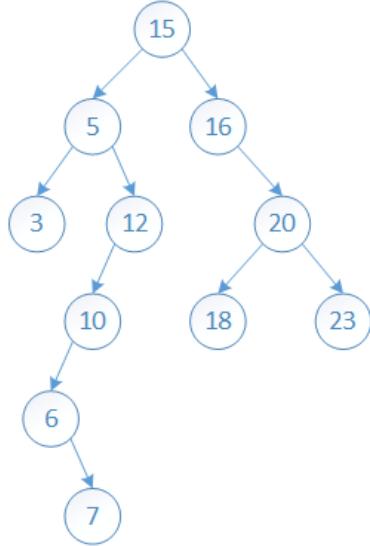
```
// complessità: Θ(1)
transplant(Tree t, Node u, Node v){
    if(u.p == NULL) // il vecchio nodo è radice
        t.root = v;
    else if(u == u.p.left) // il vecchio nodo è figlio sx
        u.p.left = v;
    else // il vecchio nodo è figlio destro
        u.p.right = v;
    if(v != NULL) // se è un nodo
        v.p = u.p;
}
```

Ora possiamo andare ed implementare *treeDelete*:

```
// complessità: O(h)
treeDelete(Tree t, Node z){
    if(z.left == NULL)
        transplant(t, z, z.right);
    else if(z.right == NULL)
        transplant(t, z, z.left);
    else {
        y = treeMinimum(z.right); // O(h)
        if(y.p != z) {
            transplant(t, y, r.right);
            y.right = z.right;
            z.right.p = y;
        }
        transplant(t, z, y);
        y.left = z.left;
        y.left.p = y;
    }
}
```

12.9.2 Esercizio : dimostrazione

Dimostrare che se un nodo in un albero binario di ricerca ha due figli, allora il suo successore non ha un figlio sinistro e il suo predecessore non ha un figlio destro.



Se prendiamo in considerazione il nodo 5, dobbiamo quindi dimostrare che il nodo 6 suo successore non ha un figlio sinistro e che il nodo 3 suo predecessore non ha un figlio destro.

Sia x un nodo con due figli. In una visita simmetrica i nodi del sottoalbero sinistro precedono x e quelli del sottoalbero destro lo seguono. Così il predecessore di x si troverà nel sottoalbero sinistro di x , mentre il successore di x , se esiste, si troverà nel sottoalbero destro di x stesso.

Sia s il successore di x . Assumiamo per assurdo (per arrivare ad una contraddizione) che s abbia un figlio sinistro y . Se così fosse y seguirebbe x nella visita perché si trova nel sottoalbero destro di x , ma precede s perché è nel sottoalbero sinistro di s . Dunque l'ordine della visita sarebbe:

$$x \rightarrow y \rightarrow s$$

Assurdo perché s non sarebbe il successore di x se avesse un figlio sinistro. In modo simmetrico si può dimostrare che il successore di x non ha un figlio destro.

12.9.3 Teorema

Le operazioni quali *search*, *treeMinimum*, *treeMaximum*, *successor*, *predecessor*, *insert* e *delete* sugli insiemi dinamici, possono essere svolte in tempo $O(h)$ usando un albero di ricerca binario di altezza h . L'unico nodo per mantenere queste operazioni efficienti è tenere l'albero bilanciato, in modo che esse siano svolte in tempo $O(\log n)$ dove n è il numero dei nodi (in un albero bilanciato infatti $h = \log n$).

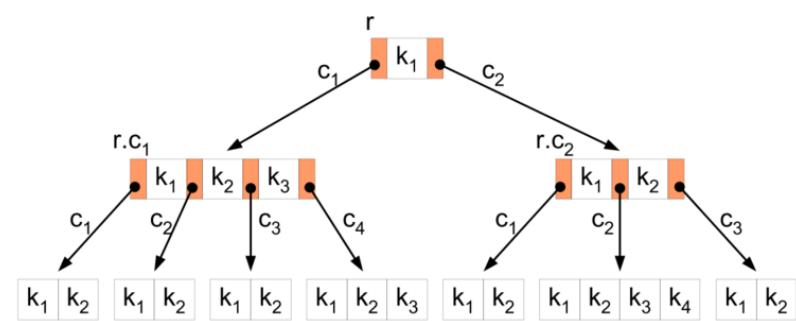
12.10 Alberi AVL

Gli alberi AVL (Adelson-Velskij e Landis) sono alberi binari di ricerca bilanciati. Oltre alla chiave mantengono un informazione sul bilanciamento. Il fattore di bilanciamento di un nodo è la differenza tra l'altezza del sottoalbero sinistro e quello destro. In un albero AVL il fattore di bilanciamento è sempre minore o uguale a 1 in ogni nodo.

12.11 B-Alberi

Sono degli alberi di ricerca bilanciati di grado minimo t , con $t \geq 2$, con le seguenti caratteristiche:

- 1) Tutte le foglie sono allo stesso livello, ovvero hanno stessa profondità.
- 2) Ogni nodo v diverso dalla radice mantiene un numero di chiavi $k(v)$ ordinate e $k_1(v) \leq k_2(v) \leq \dots \leq k(v)$ tale che $t - 1 \leq k(v) \leq 2t - 1$.
- 3) La radice mantiene almeno una chiave ed al più $2t - 1$ chiavi ordinate.
- 4) Ogni nodo interno v ha $k(v) + 1$ figli.
- 5) Le chiavi presenti nei figli $k_i(v)$ separano gli intervalli di chiavi memorizzate in ciascun sottoalbero: se c_1 è una qualunque chiave nel i-esimo sottoalbero di un nodo v allora $c_1 \leq k_1(v) \leq c_2 \leq k_2(v) \dots$



13 Algoritmi di Ordinamento

Il problema dell'ordinamento è che dato in input una sequenza di n numeri $\langle a_1, a_2, \dots, a_n \rangle$ si vuole produrre in output una permutazione della sequenza di input $\langle a_1, a_2, \dots, a_n \rangle$ tale che $\langle a_1 \leq a_2 \leq \dots \leq a_n \rangle$.

Per la soluzione di questo problema vi sono differenti approcci ed algoritmi.

13.1 Algoritmi basati sul confronto

13.1.1 InsertionSort

Algoritmo basato sulla tecnica incrementale: se ho k elementi già ordinati estendo la mia soluzione al $k+1$ -esimo elemento.

Questo algoritmo può essere paragonato all'ordinamento che eseguiamo quando riceviamo delle carte: la parte ordinata è la parte che teniamo in mano, l'altra sono le carte che dobbiamo ancora ricevere.

```
InsertionSort(Array a) {
    for(j=2 to a.length) {
        key = a[j]
        i = j-1;
        while(i > 0 AND key < a[i]) {
            a[i+1] = a[i];
            i = i - 1;
        }
        A[i+1] = key;
    }
}
```

Questo algoritmo presenta alcune peculiarità, ad esempio l'*ordinamento in loco*.

Ordinamento in loco : un algoritmo ordina sul posto se in ogni istante al più un numero costante di elementi dell'array di input sono memorizzati all'esterno dell'array.

Dimostrazione di correttezza Per dimostrare che questo algoritmo è corretto, dobbiamo trovare gli invarianti.

Invariante del ciclo esterno Il sotto-array $a[1, \dots, j - 1]$ è formato dagli elementi ordinati che originariamente erano nella posizione $a[1, \dots, j - 1]$.

❖ **Esercizio per casa** Dimostrare che questo sia un invariante.

Teorema l'algoritmo *insertionSort* ordina in loco n elementi eseguendo nel caso peggiore (input decrescente) $\Theta(n^2)$ confronti. Il ciclo esterno è eseguito $(n - 1)$ volte e il numero totale di confronti è:

$$\sum_{j=2}^n (j - 1) = \sum_{k=1}^{n-1} k = \frac{(n - 1 + 1)(n - 1)}{2} = \frac{n(n - 1)}{2} = \Theta(n^2)$$

Questo algoritmo è sensibile all'ordinamento degli elementi in input. Viene tuttavia utilizzato spesso in vettori di piccole dimensioni.

13.1.2 MergeSort

Algoritmo di ordinamento basato sulla tecnica del *divide te impera*. Si vuole ordinare un vettore $a[p, \dots, r]$ inizialmente $p = 1$ e $r = n = a.length$. I passi da svolgere sono:

- 1) *[divide]* si divide l'array in due sotto-array $a[p, \dots, q]$ e $a[q + 1, \dots, r]$ con $q = \lfloor \frac{p+r}{2} \rfloor$.
- 2) *[impera]* si ordinano i due sotto-array in modo ricorsivo utilizzando *mergeSort*. Se il problema è sufficientemente piccolo risolve direttamente.
- 3) *[ricomponi]* si fondono insieme i due sotto-array ordinati per generare un singolo vettore ordinato $a[p, \dots, r]$.

```
mergeSort(Array a, int p, int r){
    if(p < r){
        q = ⌊(p+r)/2⌋; // parte intera
        mergeSort(a,p,q);
        mergeSort(a,q+1,r);
        merge(a,p,q,r);
    } else{
        // p==r --> 1 elemento
        // p > r --> 0 elementi
        // caso base
        // non faccio nulla
    }
}
```

La procedura *merge*: $p \leq q < r$. Il sotto-array $a[p, \dots, q]$ è ordinato e anche il sotto-array $a[q + 1, \dots, r]$ è ordinato. Per le restrizioni che abbiamo sui parametri p , q ed r sappiamo che nessuno degli sotto-array è vuoto. L'output della funzione è la fusione dei due sotto-array in un unico array ordinato $a[p, \dots, r]$.

```
merge(Array a, int p, int q, int r){
    int n1 = q-p+1; // numero elementi a sinistra
    int n2 = r-q; // numero di elementi a destra
    crea array l di dimensione n1+1;
    crea array r di dimensione n2+1;
    for(i=0 to n1)
        l[i] = a[p+i-1];
    for(i=0 to n2)
        r[i] = a[q+i];
    l[n1+1] = ∞; // guardia per evitare controlli sui bounds
    r[n2+1] = ∞; // guardia per evitare controlli sui bounds
    i = 1; // indice per l
    j = 1; // indice per r
    for(k=p to r){
        if(l[i] <= r[j]) {
            a[k] = l[i];
            i++;
        }
        else {
            a[k] = r[j];
            j++;
        }
    }
}
```

Invariante: il sotto-array $a[p, \dots, k - 1]$ contiene ordinati gli $(k - p)$ elementi più piccoli presenti in $l[1, \dots, n1 + 1]$ e $r[1, \dots, n2 + 1]$. Inoltre $l[i]$ e $r[j]$ sono i più piccoli elementi dei loro array che non sono stati copiati in a .