

Appunti di Algoritmi e Strutture Dati

Luca Facchini

Matricola: 245965

Corso tenuto dal prof. Montresor Alberto

Università degli Studi di Trento

A.A. 2024/2025

Sommario

Appunti del corso di Algoritmi e Strutture Dati tenuto dal prof. Montresor Alberto presso l'Università degli Studi di Trento nell'anno accademico 2024/2025.

1

¹Le immagini e gli algoritmi (identificati da Algorithm ##) presenti in questo documento sono stati presi dai materiali forniti dal professor Montresor Alberto (alberto.montresor@unitn.it) durante il corso, sono condivisi sotto licenza Creative Commons Attribution-ShareAlike 4.0 International License (CC BY-SA 4.0). Come conseguenza le sole immagini sono soggette a questa licenza, il contenuto testuale in quanto appunti personali tratti da lezioni ed altro è soggetto alla licenza "genitore" del quale questo documento fa parte

Indice

1	Analisi di Algoritmi	5
1.1	Modelli di calcolo	5
1.1.1	Definizioni	5
1.1.2	Esempi di Analisi	6
1.1.3	Ordini di Complessità	7
1.2	Notazione asintotica	7
1.2.1	Notazioni O , Ω , Θ	7
1.2.2	Esempi e Esercizi	8
1.3	Complessità problemi v/s algoritmi	9
1.3.1	Moltiplicazione numeri complessi	9
1.3.2	Sommare numeri binari	9
1.3.3	Moltiplicare numeri binari	9
1.4	Algoritmi di Ordinamento	10
1.4.1	Selection Sort	11
1.4.2	Insertion Sort	11
1.4.3	Merge Sort	12
2	Analisi di funzioni	14
2.1	Notazione asintotica	14
2.1.1	Definizioni	14
2.2	Proprietà della notazione asintotica	14
2.2.1	Regola Generale	14
2.2.2	Proprietà delle notazioni	15
2.2.3	Altre funzioni di costo	17
2.2.4	Giocando con le espressioni	17
2.2.5	Classificazione delle funzioni	17
2.3	Ricorrenze	17
2.3.1	Introduzione	17
2.3.2	Metodo dell'albero di ricorsione, o per livelli	18
2.3.3	Metodo di sostituzione	19
2.3.4	Metodo dell'esperto, o delle ricorrenze comuni	22
3	Alberi	24
3.1	Introduzione	24
3.2	Alberi Binari	24
3.2.1	Implementazione	25
3.2.2	Visite	26
3.3	Alberi Generici	27
3.3.1	Visite	27

4	Alberi Binari di Ricerca	29
4.1	Alberi Binari di Ricerca	29
4.1.1	Ricerca - <code>lookupNode()</code>	30
4.1.2	Minimo & Massimo	31
4.1.3	Successore e Predecessore	31
4.1.4	Inserimento - <code>insertNode()</code>	32
4.1.5	Cancellazione - <code>remove()</code>	33
4.2	Alberi Binari di Ricerca Bilanciati	34
4.2.1	Definizione	34
4.2.2	Alberi <i>Red-Black</i>	34
4.2.3	Inserimento	35
4.2.4	Teoremi su un albero <i>Red-Black</i>	39
4.2.5	Cancellazione	40
5	Grafi	42
5.1	Introduzione	42
5.1.1	Definizioni	42
5.1.2	Specifica	43
5.1.3	Memorizzazione	43
5.2	Visite dei grafi	45
5.2.1	Visita in ampiezza BFS	46
5.2.2	Visita in profondità - DFS	48
6	Hashing	54
6.1	Introduzione	54
6.1.1	Preambolo	54
6.1.2	Tabelle Hash	54
6.2	Funzioni Hash	55
6.2.1	Introduzione	55
6.2.2	Funzioni Hash Semplici	55
6.3	Gestione delle collisioni	56
6.3.1	Liste/Vettori di trabocco	56
6.3.2	Indirizzamento aperto	57
6.3.3	Analisi complessità	59
7	<i>Divide-et-impera</i>	60
7.1	Introduzione	60
7.2	Torre di Hanoi	61
7.3	QuickSort	62
7.4	Algoritmo di Strassen	62
8	Analisi ammortizzata	64
8.1	Contatore Binario	64
8.1.1	Metodo dell'aggregazione	64
8.1.2	Metodo degli accantonamenti	65
8.1.3	Metodo del potenziale	65
8.2	Vettori Dinamici	66
8.2.1	Inserimento	66
8.2.2	Cancellazione	67

9	Strutture dati speciali	69
9.1	Code con priorità	69
9.1.1	Introduzione	69
9.1.2	Vettore <i>Heap</i>	70
9.1.3	<i>HeapSort</i>	71
9.1.4	Implementazione code con priorità	72

Capitolo 1

Analisi di Algoritmi

1.1 Modelli di calcolo

1.1.1 Definizioni

Complessità

Definizione 1.1. La **complessità** di un algoritmo è definita come la quantità di **tempo** necessaria per eseguirlo in funzione della **dimensione dell'input**.

Le domande spontanee che ci si pone sono dunque:

- Come definire la dimensione dell'input?
- Come misurare il tempo?

Dimensione dell'input

Definizione 1.2. Per definire la **dimensione dell'input** abbiamo due criteri:

Costo Logaritmico Il costo logaritmico è definito come il numero di bit necessari per rappresentare l'input.

Costo Uniforme La taglia dell'input è definita come il numero di elementi da cui è composto.

Ma in molti casi possiamo assumere che tutti gli elementi siano rappresentati dallo stesso numero di bit costante e che coincidono a meno di costante moltiplicativa

Tempo

Definizione 1.3. Un'istruzione si considera elementare se può essere eseguita in tempo "costante" dal processore, dunque un esempio ne è la moltiplicazione o una funzione matematica ad esempio $\cos(d)$, ma una istruzione come il massimo tra due numeri non è elementare.

Modello di calcolo

Definizione 1.4. Un modello di calcolo è definito come la rappresentazione astratta di un calcolatore che rispetta i seguenti criteri:

Astrazione Il modello deve permettere di nascondere i dettagli.

Realismo Il modello deve riflettere una situazione reale.

Potenza Matematica Il modello deve permettere di dimostrare "formalmente" la complessità di un algoritmo.

Esempio di modello di calcolo è la **Macchina di Turing**.

1.1.2 Esempi di Analisi

Tempo di calcolo $\min()$

Sappiamo che ogni istruzione richiede un tempo costante per essere eseguita e che ogni operazione potenzialmente ha una costante diversa dalle altre e che ogni istruzione viene eseguita un numero di volte diversa dalle altre.

ITEM $\min(\text{ITEM}[]\ A, \text{int } n)$		
	Costo	# Volte
ITEM $\min = A[1]$	c_1	1
for $i = 2$ to n do	c_2	n
if $A[i] < \min$ then	c_3	$n - 1$
$\min = A[i]$	c_4	$n - 1$
return \min	c_5	1

Otteniamo quindi che il tempo di calcolo è:

$$\begin{aligned} T(n) &= c_1 + c_2n + c_3(n - 1) + c_4(n - 1) + c_5 \\ &= (c_2 + c_3 + c_4)n + (c_1 + c_5 - c_3 - c_4) = an + b \end{aligned}$$

Tempo di calcolo di $\text{binarySearch}()$

In questo algoritmo il vettore viene suddiviso in due parti: Parte SX: $\lfloor (n - 1)/2 \rfloor$ e Parte DX: $\lfloor n/2 \rfloor$.

int $\text{binarySearch}(\text{ITEM}[]\ A, \text{ITEM } v, \text{int } i, \text{int } j)$			
	Costo	# ($i > j$)	# ($i \leq j$)
if $i > j$ then	c_1	1	1
return 0	c_2	1	0
else			
int $m = \lfloor (i + j)/2 \rfloor$	c_3	0	1
if $A[m] = v$ then	c_4	0	1
return m	c_5	0	0
else if $A[m] < v$ then	c_6	0	1
return $\text{binarySearch}(A, v, m + 1, j)$	$c_7 + T(\lfloor n/2 \rfloor)$	0	0/1
else			
return $\text{binarySearch}(A, v, i, m - 1)$	$c_7 + T(\lfloor (n - 1)/2 \rfloor)$	0	1/0

A questo punto dobbiamo fare delle assunzioni:

- Assumiamo che la n potenza di 2 sia: $n = 2^k$.
- L'elemento cercato non è presente.
- Ad ogni passo andiamo sempre a destra in quanto il numero di elementi da valutare è maggiore: $n/2$.

possiamo ora suddividere il problema in due casistiche:

$$\begin{aligned} i > j \quad (n = 0) \quad T(n) &= c_1 + c_2 \\ i \leq j \quad (n > 0) \quad T(n) &= T(n/2) + c_1 + c_2 + c_3 + c_4 + c_6 + c_7 \\ &= T(n/2) + d \end{aligned}$$

unendo i due casi otteniamo la **Relazione di ricorrenza**:

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

ottenuta la relazione generalmente per un numero n di elementi otteniamo che il tempo è dato da:

$$\begin{aligned} T(n) &= T(n/2) + d \\ &= T(n/4) + 2d \\ &\dots \\ &= T(1) + kd \\ &= T(0) + (k+1)d \\ &= kd + (c+d) = d \log n + e \end{aligned}$$

ottenendo quindi che il tempo di calcolo è $O(\log n)$ di natura logaritmica.

1.1.3 Ordini di Complessità

$f(n)$	$n = 10^1$	$n = 10^2$	$n = 10^3$	$n = 10^4$	Tipo
$\log n$	3	6	9	13	Logaritmica
\sqrt{n}	3	10	31	100	sub-lineare
n	10	100	1000	10000	Lineare
$n \log n$	30	664	9965	132877	log-lineare
n^2	10^2	10^4	10^6	10^8	Quadratica
n^3	10^3	10^6	10^9	10^{12}	Cubica
2^n	1024	10^{30}	10^{301}	10^{3010}	Esponenziale

1.2 Notazione asintotica

1.2.1 Notazioni O , Ω , Θ

Notazione O

Definizione 1.5. Sia $g(n)$ una funzione di costo; indichiamo con $O(g(n))$ l'insieme delle funzioni $f(n)$ tali per cui:

$$\exists c > 0, \exists m \geq 0 : f(n) \leq cg(n), \forall n \geq m$$

La seguente notazione si legge: $f(n)$ è "O grande" (big O) di $g(n)$, con un abuso di notazione si scrive $f(n) = O(g(n))$ ¹. Inoltre per la precedente definizione possiamo dire che $g(n)$ è un **limite asintotico superiore** per $f(n)$, in quanto dopo qualche valore m la funzione $g(n)$ è sempre maggiore di $f(n)$. Inoltre per questo motivo sappiamo che $f(n)$ cresce al più come $g(n)$.

Notazione Ω

Definizione 1.6. Sia $g(n)$ una funzione di costo; indichiamo con $\Omega(g(n))$ l'insieme delle funzioni $f(n)$ tali per cui:

$$\exists c > 0, \exists m \geq 0 : f(n) \geq cg(n), \forall n \geq m$$

La seguente notazione si legge: $f(n)$ è "Omega" di $g(n)$, con un abuso di notazione si scrive $f(n) = \Omega(g(n))$ ¹. Inoltre per la precedente definizione possiamo dire che $g(n)$ è un **limite asintotico inferiore** per $f(n)$, in quanto dopo qualche valore m la funzione $g(n)$ è sempre minore di $f(n)$. Inoltre per questo motivo sappiamo che $f(n)$ cresce almeno come $g(n)$.

¹ Questo è un abuso di notazione in quanto $O(g(n))$ è una classe di funzioni e non può essere eguagliata una singola funzione, il simbolo più appropriato sarebbe $f(n) \in O(g(n))$

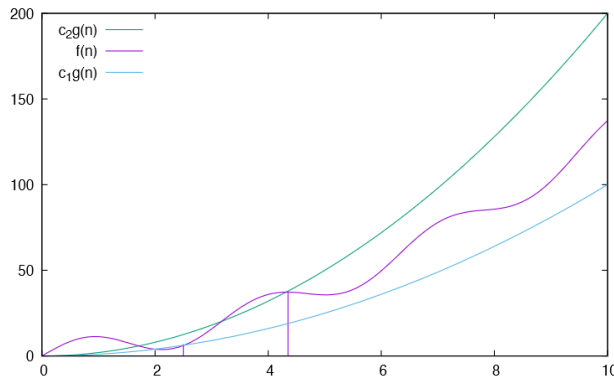
Notazione Θ

Definizione 1.7. Sia $g(n)$ una funzione di costo; indichiamo con $\Theta(g(n))$ l'insieme delle funzioni $f(n)$ tali per cui:

$$\exists c_1 > 0, \exists c_2 > 0, \exists m \geq 0 : c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq m$$

La seguente notazione si legge: $f(n)$ è "Theta" di $g(n)$, con un abuso di notazione si scrive $f(n) = \Theta(g(n))$ ¹. Inoltre per la precedente definizione possiamo dire che $f(n)$ cresce esattamente come $g(n)$, detto ciò $f(n) = \Theta(g(n))$ se e solo se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$.

Esempio grafico



1.2.2 Esempi e Esercizi

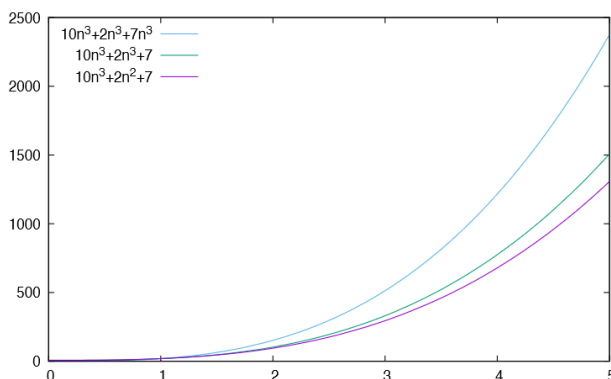
Esempio 1

$$f(n) = 10n^3 + 2n^2 + 7 \stackrel{?}{=} O(n^3)$$

Dobbiamo provare che $\exists c > 0, \exists m \geq 0 : f(n) \leq cn^3, \forall n \geq m$.

$$\begin{aligned} f(n) &= 10n^3 + 2n^2 + 7 \\ &\leq 10n^3 + 2n^3 + 7 && \forall n \geq 1 \\ &\leq 10n^3 + 2n^3 + n^3 && \forall n \geq \sqrt[3]{7} \\ &= 13n^3 \stackrel{?}{\leq} cn^3 \end{aligned}$$

Che è verificata per qualsiasi $c \geq 13$ e $m \geq \sqrt[3]{7}$, arrotondiamo m ad un intero superiore ottenendo $m = 2$.



1.3 Complessità problemi v/s algoritmi

1.3.1 Moltiplicazione numeri complessi

Per moltiplicare numeri complessi bisogna svolgere la seguente operazione:

$$(a + bi)(c + di) = [ac - bd] + [ad + bc]i$$

dunque dai parametri a, b, c, d otteniamo che il risultato da restituire: $ac - bd$ e $ad + bc$.

Domande Considerando che addizioni e sottrazioni costino $c_1 = 0.01$ e moltiplicazione costi $c_2 = 1$ possiamo chiederci:

- Quanto costa l'algoritmo?
- Si può fare meglio?
- Qual'è il ruolo del modello di calcolo?

Dato che si devono fare almeno 4 moltiplicazioni e 2 somme otteniamo che il costo totale è $4c_2 + 2c_1 = 4.02$.

Ora si può fare meglio? La risposta è no in quanto se si potesse fare meglio si potrebbe fare meglio allora bisognerebbe trovare un algoritmo che esegua meno di 4 moltiplicazioni e 2 somme, ma per fare ciò bisognerebbe cambiare il modello di calcolo.

1.3.2 Sommare numeri binari

Algoritmo elementare della somma - sum()

Ipotizziamo che l'operazione da fare sia la somma di due bit singoli e generare il riporto, assegniamo a questa operazione costo c . Dopo ciò indichiamo con n il numero massimo di bit tra i due numeri da sommare, otteniamo dunque che:

- Richiede di esaminare tutti gli n bit
- Costo totale $cn = O(n)$

Esiste allora un algoritmo più efficiente?

La risposta è no in quanto se esistesse un algoritmo di tale genere allora potremmo cambiare un solo bit di uno dei due numeri e ottenere un risultato diverso senza che l'algoritmo lo vada ad analizzare, il che è impossibile.

1.3.3 Moltiplicare numeri binari

Algoritmo elementare del prodotto - prod()

L'operazione elementare per moltiplicare due numeri binari è la seguente:

$$\begin{array}{cccccccccccccccc}
 & & & & & & & & 1 & 0 & 1 & 1 & 1 & 0 & 1 & * \\
 & & & & & & & & 1 & 1 & 0 & 1 & 1 & 1 & 0 & \\
 \hline
 & & & & & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
 & & & & & & & 1 & 0 & 1 & 1 & 1 & 0 & 1 & & \\
 & & & & & & 1 & 0 & 1 & 1 & 1 & 0 & 1 & & & \\
 & & & & 1 & 0 & 1 & 1 & 1 & 0 & 1 & & & & & \\
 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & \\
 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & & & & & & & & \\
 \hline
 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 &
 \end{array}$$

deduciamo che:

- Dobbiamo accedere a tutti i bit
- Il costo totale è $cn^2 = O(n^2)$ perché dobbiamo fare n somme di n bit.

Soluzione Divide-et-impera

La base del principio **Divide-et-impera** è la seguente:

Divide Dividere il problema in sotto-problemi più piccoli.

Impera risolvi i sotto-problemi in modo ricorsivo.

Combina combina le soluzioni dei sotto-problemi per ottenere la soluzione del problema originale.

La soluzione divide-et-impera per il problema della moltiplicazione binaria è la seguente:

$$\begin{aligned} X &= a \cdot 2^{n/2} + b & X &= \overset{a}{\text{Parte SX}} \quad \overset{b}{\text{Parte DX}} \\ Y &= c \cdot 2^{n/2} + d & Y &= \overset{c}{\text{Parte SX}} \quad \overset{d}{\text{Parte DX}} \\ XY &= ac \cdot 2^n + (ad + bc) \cdot 2^{n/2} + bd \end{aligned}$$

Ora possiamo scrivere l'algoritmo: La funzione di costo associata all'algoritmo è:

Algorithm 1 boolean[] pdi(boolean[] X, boolean[] Y, int n)

```

1: if n = 1 then
2:   return X[1] · Y[1]
3: else
4:   spezza X in a e b e Y in c e d
5:   return pdi(a, c, n/2) · 2^n + (pdi(a, d, n/2) + pdi(b, c, n/2)) · 2^{n/2} + pdi(b, d, n/2)

```

$$T(n) = \begin{cases} c_1 & n = 1 \\ 4T(n/2) + c_2 \cdot n & n > 1 \end{cases}$$

Nota: moltiplicare per 2^n corrisponde a uno shift a sinistra di n posizioni, svolta in tempo lineare.

Ora in quanto abbiamo 4 chiamate ricorsive, assumendo che c_1 sia il tempo per moltiplicare due bit e che c_2 sia il tempo per sommare due numeri binari otteniamo che il tempo di calcolo è $c_2 \cdot 4^i \cdot \frac{n}{2^i} = T(1) \cdot 4^{\log_2 n} = c_1 \cdot n^{\log_2 4} = c_1 \cdot n^2$.

Ma allora è possibile fare meglio? Long story short: si in quanto è stato provato nel 2021 l'esistenza di un algoritmo di complessità $O(n \log n)$.

1.4 Algoritmi di Ordinamento

Introduzione L'obiettivo di questa sezione è valutare la complessità degli algoritmi in base all'input, in alcuni casi gli algoritmi si comportano diversamente in base all'input se siamo a conoscenza dell'input questa ci consente di scegliere un algoritmo più adeguato alla nostra soluzione.

Come analizziamo gli algoritmi Possiamo analizzare l'efficienza degli algoritmi in base a diversi casi:

Caso Pessimo Questa analisi è la più importante in quanto sappiamo che questa restituisce il limite superiore al tempo di esecuzione qualsiasi sia l'input.

Caso Medio Questa analisi è la più complessa in quanto bisogna definire il "caso medio" e cosa si intende per "medio", ma è utile con una distribuzione uniforme degli input.

Caso Ottimo Utile solo se conosciamo qualcosa sull'input, altrimenti non risulta utile se abbiamo un input arbitrario.

1.4.1 Selection Sort

Algoritmo Selection Sort:

Algorithm 2 selectionSort(Item[] A, int n)

```

1: for  $i = 1$  to  $n - 1$  do
2:   int  $min \leftarrow \min(A, i, n)$ 
3:    $A[i] \leftrightarrow \min(A, i, n)$ 

```

Algoritmo di supporto min: Avendo analizzato il seguente algoritmo notiamo come in ogni caso, ottimo,

Algorithm 3 int min(Item[] A, int i, int n)

```

1: int  $min \leftarrow i$ 
2: for  $j = i + 1$  to  $n$  do
3:   if  $A[j] < A[min]$  then
4:      $min \leftarrow j$ 
5: return min

```

medio e pessimo, il "ciclo" esterno della funzione selectionSort() viene eseguito $n - 1$ volte, mentre il ciclo interno della funzione min() viene eseguito $n - i$ dove i è il valore dell'iterazione del ciclo esterno, quindi $n - 1 + n - 2 + n - 3 + \dots + 1 = \frac{n(n-1)}{2}$ volte, otteniamo quindi che il tempo di calcolo è $O(n^2)$ in quanto questo si può approssimare a $\frac{n^2}{2}$.

$$\sum_{i=1}^{n-1} n - i = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

1.4.2 Insertion Sort

L'algoritmo di insertion sort è efficiente per ordinare piccoli insiemi, il concetto dietro a questo si basa sull'inserimento dell'elemento preso in analisi al posto giusto.

Algoritmo Insertion Sort:

Algorithm 4 insertionSort(Item[] A, int n)

```

1: for  $i = 2$  to  $n$  do
2:   Item  $temp \leftarrow A[i]$ 
3:   int  $j \leftarrow i$ 
4:   while  $j > 1$  and  $A[j - 1] > temp$  do
5:      $A[j] \leftarrow A[j - 1]$ 
6:      $j \leftarrow j - 1$ 
7:    $A[j] \leftarrow temp$ 

```

Il costo di esecuzione non dipende esclusivamente dalla dimensione ma anche dall'ordine degli elementi in ingresso.

Caso Pessimo Il costo dunque nel **caso pessimo** è $O(n^2)$ in quanto vengono eseguiti $n - 1$ cicli esterni e $n - 1$ cicli interni, ottenendo dunque $(n - 1) + (n - 2) + \dots + 1 = \frac{n(n-1)}{2} = O(n^2)$.

Caso Medio Nel **caso medio** il costo rimane $O(n^2)$ in quanto il ciclo interni viene eseguito $n/2$ volte, ottenendo dunque $\frac{n(n-1)}{4} = O(n^2)$.

Caso Ottimo Nel **caso ottimo** il costo è $O(n)$ in quanto il ciclo interno non viene mai eseguito.

Questo ci porta a dire che il `insertionSort()` è un algoritmo di ordinamento utile nei casi in cui l'input è già ordinato o quasi ordinato, nei casi nei quali non conosciamo la natura dell'input è meglio utilizzare un algoritmo di ordinamento differente.

1.4.3 Merge Sort

L'algoritmo di `mergeSort()` è un algoritmo di ordinamento basato sul principio **divide-et-impera**.

Divide: Spezza virtualmente il vettore di n elementi in sotto-vettori di $n/2$ elementi.

Impera: Chiama `mergeSort()` ricorsivamente sui due sotto-vettori.

Combina: Unisci (**merge**) i due sotto-vettori ordinati in un unico vettore ordinato.

In input si ha:

- A : vettore di n elementi.
- $start, end, mid$ sono tali che $1 \leq start < mid < end \leq n$.
- I sotto-vettori $A[start, \dots, mid]$ e $A[mid + 1, \dots, end]$ sono ordinati.

In output si hanno i due sotto-vettori fusi in un unico sotto-vettore ordinato, tramite un vettore di appoggio B .

Funzione di appoggio Merge:

Algorithm 5 Merge(Item[] A , **int** $start$, **int** end , **int** mid)

```

1: int  $i, j, k, h$ 
2: int  $i \leftarrow start$ 
3: int  $j \leftarrow mid + 1$ 
4: int  $k \leftarrow start$ 
5: while  $i \leq mid$  and  $j \leq end$  do
6:   if  $A[i] \leq A[j]$  then
7:      $B[k] \leftarrow A[i]$ 
8:      $i \leftarrow i + 1$ 
9:   else
10:     $B[k] \leftarrow A[j]$ 
11:     $j \leftarrow j + 1$ 
12:    $k \leftarrow k + 1$ 
13:  $j \leftarrow end$ 
14: for  $h = mid$  downto  $i$  do
15:    $A[j] \leftarrow A[h]$ 
16:    $j \leftarrow j - 1$ 
17: for  $j = start$  to  $k - 1$  do
18:    $A[j] \leftarrow B[j]$ 

```

Il costo computazionale di Merge() è $O(n)$, questa è la base del costo computazionale di `mergeSort()`.

Funzione completa mergeSort():

Algorithm 6 mergeSort(Item[] A, **int** start, **int** end)

```
1: if start < end then  
2:   int mid ← (start + end)/2  
3:   mergeSort(A, start, mid)  
4:   mergeSort(A, mid + 1, end)  
5:   merge(A, start, end, mid)
```

Assumendo per semplificare che $n = 2^k$ dove k è un intero allora l'altezza dell'albero è esattamente $k = \log n$, in questo modo tutti i sotto-vettori hanno dimensione che è potenza di 2. Così facendo il costo computazionale di mergeSort() è:

$$T(n) = \begin{cases} c & n = 1 \\ 2T(n/2) + dn & n > 1 \end{cases}$$

dove c è il costo di un'operazione elementare, d è il costo di Merge() e n è il costo di copiare i valori da B a A .

Capitolo 2

Analisi di funzioni

2.1 Notazione asintotica

2.1.1 Definizioni

Si rimanda al sezione 1.2 per le definizioni di O , Ω e Θ .

2.2 Proprietà della notazione asintotica

2.2.1 Regola Generale

Da qui si prende in considerazione la seguente espressione polinomiale:

$$f(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0, \quad a_k > 0 \Rightarrow f(n) = \Theta(n^k)$$

Limite Superiore $\exists c > 0, \exists m \geq 0 : f(n) \leq cn^k, \forall n \geq m$

$$\begin{aligned} f(n) &= a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0 \\ &\leq a_k n^k + |a_{k-1}| n^{k-1} + \dots + |a_1| n + |a_0| \\ &\leq a_k n^k + |a_{k-1}| n^k + \dots + |a_1| n^k + |a_0| n^k \\ &= (a_k + |a_{k-1}| + \dots + |a_1| + |a_0|) n^k \quad \forall n \geq 1 \\ &\stackrel{?}{\leq} cn^k \end{aligned}$$

questa è vera per $c \geq (a_k + |a_{k-1}| + \dots + |a_1| + |a_0|) > 0$ e per $m = 1$

Limite Inferiore $\exists d > 0, \exists m \geq 0 : f(n) \geq dn^k, \forall n \geq m$

$$\begin{aligned} f(n) &= a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0 \\ &\geq a_k n^k - |a_{k-1}| n^{k-1} - \dots - |a_1| n - |a_0| \\ &\geq a_k n^k - |a_{k-1}| n^k - \dots - |a_1| n^k - |a_0| n^k \\ &= (a_k - |a_{k-1}| - \dots - |a_1| - |a_0|) n^k \quad \forall n \geq 1 \\ &\stackrel{?}{\geq} dn^k \end{aligned}$$

questa è vera se: $d \leq a_k - \frac{|a_{k-1}|}{n} - \dots - \frac{|a_1|}{n} - \frac{|a_0|}{n} > 0 \Leftrightarrow n > \frac{|a_{k-1}| + \dots + |a_1| + |a_0|}{a_k}$

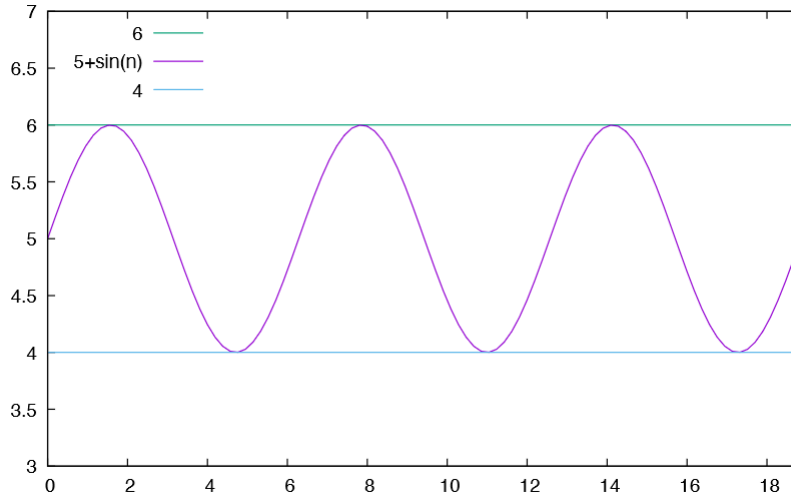
Casi Particolari**Complessità di $f(n) = 5$**

$$f(n) = 5 \geq c_1 n^0 \Rightarrow c_1 \leq 5$$

$$f(n) = 5 \leq c_2 n^0 \Rightarrow c_2 \geq 5$$

$$\Rightarrow f(n) = \Theta(n^0) = \Theta(1)$$

Complessità di $f(n) = 5 + \sin(n)$ La complessità di calcolo di $f(n)$ è $\Theta(1)$, in quanto $\sin(n)$ è una funzione oscillante tra -1 e 1 , quindi $5 + \sin(n)$ oscilla tra 4 e 6 .

**2.2.2 Proprietà delle notazioni****Dualità**

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

Dimostrazione.

$$f(n) = O(g(n)) \Leftrightarrow f(n) \leq cg(n), \forall n \geq m$$

$$\Leftrightarrow g(n) \geq \frac{1}{c} f(n), \forall n \geq m$$

$$\Leftrightarrow g(n) \geq c' f(n), \forall n \geq m, c' = \frac{1}{c}$$

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

□

Eliminazione di costanti

$$f(n) = O(g(n)) \Leftrightarrow af(n) = O(g(n)), \forall a > 0$$

$$f(n) = \Omega(g(n)) \Leftrightarrow af(n) = \Omega(g(n)), \forall a > 0$$

Dimostrazione.

$$f(n) = O(g(n)) \Leftrightarrow f(n) \leq cg(n), \forall n \geq m$$

$$\Leftrightarrow af(n) \leq acg(n), \forall n \geq m, \forall a > 0$$

$$\Leftrightarrow af(n) \leq c' g(n), \forall n \geq m, c' = ac > 0$$

$$\Leftrightarrow af(n) = O(g(n)), \forall a > 0$$

□

Sommatoria (sequenza di algoritmi)

$$f_1(n) = O(g_1(n)), f_2(n) = O(g_2(n)) \Rightarrow f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$$

$$f_1(n) = \Omega(g_1(n)), f_2(n) = \Omega(g_2(n)) \Rightarrow f_1(n) + f_2(n) = \Omega(\max(g_1(n), g_2(n)))$$

Dimostrazione (Lato O).

$$f_1(n) = O(g_1(n)) \wedge f_2(n) = O(g_2(n)) \Rightarrow$$

$$f_1(n) \leq c_1 g_1(n) \wedge f_2(n) \leq c_2 g_2(n) \Rightarrow$$

$$f_1(n) + f_2(n) \leq c_1 g_1(n) + c_2 g_2(n) \Rightarrow$$

$$f_1(n) + f_2(n) \leq \max\{c_1, c_2\}(2 \cdot \max(g_1(n), g_2(n))) \Rightarrow$$

$$f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$$

□

Prodotto (cicli annidati)

$$f_1(n) = O(g_1(n)), f_2(n) = O(g_2(n)) \Rightarrow f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$$

$$f_1(n) = \Omega(g_1(n)), f_2(n) = \Omega(g_2(n)) \Rightarrow f_1(n) \cdot f_2(n) = \Omega(g_1(n) \cdot g_2(n))$$

Dimostrazione.

$$f_1(n) = O(g_1(n)) \wedge f_2(n) = O(g_2(n)) \Rightarrow$$

$$f_1(n) \leq c_1 g_1(n) \wedge f_2(n) \leq c_2 g_2(n) \Rightarrow$$

$$f_1(n) \cdot f_2(n) \leq c_1 c_2 g_1(n) g_2(n)$$

□

Simmetria

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

Dimostrazione. Grazie alla proprietà della dualità, si ha che:

$$f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n)) \Rightarrow g(n) = \Omega(f(n))$$

$$f(n) = \Theta(g(n)) \Rightarrow f(n) = \Omega(g(n)) \Rightarrow g(n) = O(f(n))$$

□

Transitività

$$f(n) = O(g(n)), g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$$

Dimostrazione.

$$f(n) = O(g(n)) \wedge g(n) = O(h(n)) \Rightarrow$$

$$f(n) \leq c_1 g(n) \wedge g(n) \leq c_2 h(n) \Rightarrow$$

$$f(n) \leq c_1 c_2 h(n) \Rightarrow$$

$$f(n) = O(h(n))$$

□

2.2.3 Altre funzioni di costo

Logaritmi v/s funzioni lineari

Proprietà dei Logaritmi Vogliamo provare che $\log(n) = O(n)$. Dimostriamo per induzione che

$$\exists c > 0, \exists m \geq 0 : \log n \leq cn, \forall n \geq m \quad \text{definizione di } O$$

Dimostrazione.

Caso Base. ($n = 1$): $\log 1 = 0 \leq 0 \cdot 1 = 0$

Ipotesi induttiva: sia $\log k \leq ck, \forall k \leq n$.

Passo Induttivo. Dimostriamo la proprietà per $n + 1$:

$$\begin{aligned} \log(n+1) &\leq \log(n+n) = \log 2n && \forall n \geq 1 \\ &= \log 2 + \log n && \log ab = \log a + \log b \\ &= 1 + \log n && \log 2 = 1 \\ &\leq 1 + cn && \text{per induzione} \\ &\stackrel{?}{\leq} c(n+1) && \text{Obbiettivo} \\ 1 + cn &\leq c(n+1) \Leftrightarrow c \geq 1 \end{aligned}$$

□

2.2.4 Giocando con le espressioni

Es 1 È vero che $\log_a n = \Theta(\log n)$?

Si: $\log_a n = (\log_a 2) \cdot (\log_2 n) = \Theta(\log n)$

Es 2 È vero che $\log n^a = \Theta(\log n)$, per $a > 0$?

Si: $\log n^a = a \log n = \Theta(\log n)$

Es 3 È vero che $2^{n+1} = \Theta(2^n)$?

Si: $2^{n+1} = 2 \cdot 2^n = \Theta(2^n)$

Es 4 È vero che $2^n = \Theta(3^n)$?

Ovviamente $2^n = O(3^n)$

Ma: $3^n = \left(\frac{3}{2} \cdot 2\right)^n = \left(\frac{3}{2}\right)^n \cdot 2^n$: Quindi non esiste $c > 0$ tale per cui $\left(\frac{3}{2}\right)^n \cdot 2^n \leq c2^n$, quindi $2^n \neq O(3^n)$

2.2.5 Classificazione delle funzioni

È possibile definire un ordinamento delle principali classi estendendo le relazioni che abbiamo dimostrato:

Per ogni $r < s, h < k, a < b$:

$$o(1) \subset O(\log^r n) \subset O(\log^s n) \subset O(n^h) \subset O(n^h \log^r n) \subset O(n^h \log^s n) \subset O(n^k) \subset O(a^n) \subset O(b^n)$$

2.3 Ricorrenze

2.3.1 Introduzione

Equazioni di ricorrenza Quando si calcola la complessità di un algoritmo ricorsivo, questa viene espressa tramite un'equazione di ricorrenza, ovvero una formula definita in maniera ricorsiva.

MergeSort

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

Forma Chiusa L'obiettivo è quello di ottenere, quando possibile, una **formula chiusa** che rappresenti la classe di complessità della funzione.

MergeSort

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n) & n > 1 \\ \Theta(1) & n \leq 1 \end{cases} \implies T(n) = \Theta(n \log n)$$

2.3.2 Metodo dell'albero di ricorsione, o per livelli

Introduzione "Srotoliamo" la ricorrenza in un albero i costi ai vari livelli della ricorsione.

Esempio 1

$$T(n) = \begin{cases} T(n/2) + b & n > 1 \\ c & n \leq 1 \end{cases}$$

È possibile risolvere questa ricorrenza nel modo seguente:

$$\begin{aligned} T(n) &= b + T(n/2) \\ &= b + b + T(n/4) \\ &= b + b + b + T(n/8) \\ &= \dots \\ &= \underbrace{b + b + \dots + b}_{\log n} + T(1) \end{aligned}$$

Assumiamo per semplicità che $n = 2^k$. $T(n) = b \log n + c = \Theta(\log n)$

Esempio 2

$$T(n) = \begin{cases} 4T(n/2) + n & n > 1 \\ 1 & n \leq 1 \end{cases}$$

È possibile risolvere questa ricorrenza nel modo seguente:

$$\begin{aligned} T(n) &= n \sum_{j=0}^{\log(n)-1} 2^j && \underbrace{+ 4^{\log n}}_{\text{somma degli } T(1)} \\ &\quad \text{\small } n \text{ per prop. log.} && \\ &\Rightarrow n \cdot \frac{\overbrace{2^{\log n}} - 1}{2 - 1} && + 4^{\log n} \\ &\quad \text{\small usando: } \forall x \neq 1: \sum_{j=0}^k x^j = \frac{x^{k+1} - 1}{x - 1} && \\ &= n(n - 1) && + 4^{\log n} \\ &\quad \text{\small cambiamento di base} && \\ &= n^2 - n && \underbrace{+ n^2} \\ &= 2n^2 - n = \Theta(n^2) \end{aligned}$$

Esempio 3

$$T(n) = \begin{cases} 4T(n/2) + n^3 & n > 1 \\ 1 & n \leq 1 \end{cases}$$

Da questa equazione notiamo che il primo livello ha costo n^3 , il secondo $4\left(\frac{n}{2}\right)^3$ il terzo $4^2\left(\frac{n}{2^2}\right)^3$ e così via. Quindi possiamo scrivere la sommatoria:

$$\begin{aligned}
T(n) &= n^3 + 4\left(\frac{n}{2}\right)^3 + \dots + 4^{\log n - 1} \left(\frac{n}{2^{\log n - 1}}\right)^3 && + 4^{\log n} \\
&= \sum_{i=0}^{\log n - 1} 4^i \left(\frac{n}{2^i}\right)^3 && + 4^{\log n} \\
&= n^3 \sum_{i=0}^{\log n - 1} \left(\frac{2^{2i}}{2^{3i}}\right) && + 4^{\log n} \\
&= n^3 \sum_{i=0}^{\log n - 1} (2^{2i-3i}) && + 4^{\log n} \\
&= n^3 \sum_{i=0}^{\log n - 1} (2^{-1 \cdot i}) && + 4^{\log n} \\
&= n^3 \sum_{i=0}^{\log n - 1} \left(\frac{1}{2}\right)^i && \text{cambiamento di base} \\
&\leq n^3 \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i && + n^2 \\
&\quad \text{usando: } \forall x, |x| \leq 1: \sum_{i=0}^{\infty} x^i = \frac{1}{1-x} \\
&= n^3 \cdot \frac{1}{1 - \frac{1}{2}} && + n^2 \\
&= 2n^3 + n^2
\end{aligned}$$

1

Abbiamo dunque dimostrato che $T(n) \leq 2n^3 + n^2 = O(n^3)$ però non possiamo, tramite la dimostrazione precedente, dire che $T(n) = \Theta(n^3)$ in quanto siamo passati ad una disequazione. In questo particolare caso d'altra parte possiamo notare che $T(n) \geq n^3$ il che ci porta ad affermare che $T(n) = \Omega(n^3)$ e quindi $T(n) = \Theta(n^3)$.

2.3.3 Metodo di sostituzione

Introduzione È il metodo in cui si cerca di **indovinare (guess)** la soluzione e si prova a dimostrarla per **induzione**.

Primo esempio

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + n & n > 1 \\ 1 & n \leq 1 \end{cases}$$

¹In quanto la sommatoria tende a crescere allora abbiamo potuto sostituire $\log n$ con ∞ e modificare il segno di uguaglianza con quello di \leq in quanto la sommatoria verso l'infinito converge è più grande di quella finita

Notiamo come il costo dei vari livelli sia $n + n/2 + n/4 + \dots$. Dunque possiamo ipotizzare di poter scrivere:

$$\begin{aligned}
 T(n) &= n \cdot \sum_{i=0}^{\log n} \left(\frac{1}{2}\right)^i \\
 &\leq n \cdot \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i \\
 &= n \cdot \underbrace{\frac{1}{1 - \frac{1}{2}}}_{\text{usando } \forall x, |x| < 1: \sum_{i=0}^{\infty} x^i = \frac{1}{1-x}} \\
 &= 2n
 \end{aligned}$$

2

Limite superiore Tentiamo quindi ora di dire che $T(n) = O(n)$:

Caso Base $T(1) = 1 \stackrel{?}{\leq} 1 \cdot c \Leftrightarrow \forall c \geq 1$

Passo Induttivo Dimostriamo la disequazione per $T(n)$

$$\begin{aligned}
 T(n) &= T(\lfloor n/2 \rfloor) + n \\
 &\stackrel{\text{ipotizzato } O(n)}{\leq} \underbrace{c \lfloor n/2 \rfloor}_{\text{intero inferiore}} + n \\
 &\leq c \cdot \underbrace{n/2}_{\text{intero inferiore}} + n \\
 &= n(c/2 + 1) \\
 &\stackrel{?}{\leq} cn \\
 &\Leftrightarrow c/2 + 1 \leq c \Leftrightarrow c \geq 2
 \end{aligned}$$

Dunque abbiamo provato che: $T(n) \leq cn$, nel caso base $c \geq 1$ e nel passo induttivo $c \geq 2$. In quanto deve valere per entrambi i casi allora il primo valore utile di c è 2, avendo provato che per $n = 1$ la disequazione vale e che per tutti i successivi valori di n la disequazione vale allora possiamo dire che $T(n) = O(n)$.

Limite Inferiore Tentiamo di dimostrare che $T(n) = \Omega(n)$:

Caso Base Dimostriamo che $T(1) = 1 \stackrel{?}{\geq} 1 \cdot d \Leftrightarrow \forall d \leq 1$

²Abbiamo potuto usare il \leq in quanto la sommatoria in questione all'infinito è sempre maggiore di quella finita e ci stiamo calcolando il costo massimo

Passo Induttivo Dimostriamo la disequazione per $T(n)$:

$$\begin{aligned}
 T(n) &= T(\lfloor n/2 \rfloor) + n \\
 &\stackrel{\text{per ipo. induttiva sostituzione}}{\geq} \overbrace{d \lfloor n/2 \rfloor}^{\text{intero inferiore}} + n \\
 &\geq d \cdot \left(\frac{n}{2} - 1 \right) + n \\
 &= \left(\frac{d}{2} - \frac{1}{n} + 1 \right) n \stackrel{?}{\geq} dn \\
 &\Leftrightarrow \frac{d}{2} - \frac{1}{n} + 1 \geq d \\
 &\Leftrightarrow d \leq 2 - \frac{2}{n}
 \end{aligned}$$

Abbiamo quindi dimostrato che $T(n) \geq dn$, nel caso base $d \leq 1$ e nel passo induttivo $d \leq 2 - \frac{2}{n}$. In quanto deve valere per entrambi i casi allora il primo valore utile di d è 1, avendo provato che per $n = 1$ la disequazione vale e che per tutti i successivi valori di n la disequazione vale allora possiamo dire che $T(n) = \Omega(n)$.

Conclusione Avendo provato che $T(n) = O(n)$ e $T(n) = \Omega(n)$ e ricordando che se $T(n) = O(n) \wedge T(n) = \Omega(n) \Leftrightarrow T(n) = \Theta(n)$ concludendo che la funzione di costo di $T(n)$ cresce in maniera lineare.

Terzo esempio - Difficoltà matematiche

$$T(n) = \begin{cases} T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + 1 & n > 1 \\ 1 & n \leq 1 \end{cases}$$

Limite Superiore Dalla seguente si può notare come il costo di ogni livello sia 1 e che il numero di livelli sia $\log n$. Inoltre la ricorsione viene eseguita su due rami, quindi possiamo scrivere la seguente sommatoria:

$$\begin{aligned}
 T(n) &= \sum_{i=0}^{\log n} 2^i \\
 &= 1 + 2 + 4 + \dots + \frac{n}{4} + \frac{n}{2} + n \\
 &= O(n)
 \end{aligned}$$

Proviamo ora a dimostrare che $T(n) = O(n)$:

Passo Induttivo Ipotizzando che $\forall k < n : T(k) \leq ck$, dimostriamo che $T(n) \leq cn$:

$$\begin{aligned}
 T(n) &= T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1 \\
 &\leq c \left\lfloor \frac{n}{2} \right\rfloor + c \left\lceil \frac{n}{2} \right\rceil + 1 \\
 &\leq cn + 1 \\
 &\stackrel{?}{\geq} cn \Rightarrow 1 \leq 0 \quad \text{impossibile}
 \end{aligned}$$

Sebbene la dimostrazione sia fallita ma l'intuizione ci dice che $T(n) = O(n)$

Proviamo dunque a dimostrarlo per un **ordine inferiore**: $cn + 1 \leq cn$

Passo Induttivo Ipotizzando che $\exists b > 0, \forall k < n : T(k) \leq ck - b$ allora dimostriamo la disequazione per $T(n)$:

$$\begin{aligned} T(n) &= T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1 \\ &\leq c\left\lfloor \frac{n}{2} \right\rfloor - b + c\left\lceil \frac{n}{2} \right\rceil - b + 1 \\ &= cn - 2b + 1 \\ &\stackrel{?}{\leq} cn - b \\ &\Rightarrow cn - 2b + 1 \leq cn - b \Rightarrow b \geq 1 \end{aligned}$$

Dimostriamo il passo base per $b = 1$: $T(1) = 1 \stackrel{?}{\leq} 1 \cdot c - b \Leftrightarrow \forall c \geq b + 1$

Limite Inferiore Proviamo ora a dimostrare che $T(n) = \Omega(n)$:

Passo Induttivo Ipotizzando che $\forall k < n : T(k) \geq dk$, dimostriamo che $T(n) \geq dn$:

$$\begin{aligned} T(n) &= T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1 \\ &\geq d\left\lfloor \frac{n}{2} \right\rfloor + d\left\lceil \frac{n}{2} \right\rceil + 1 \\ &= dn + 1 \\ &\stackrel{?}{\geq} dn \Rightarrow 1 \geq 0 \end{aligned}$$

Il che è vero $\forall d$ in quanto d è positivo per ipotesi.

Caso base Dimostriamo la disequazione per $T(1)$:

$$T(1) = 1 \geq 1 \cdot d \Leftrightarrow d \leq 1$$

Dunque abbiamo provato che $T(n) = \Omega(n)$

Avendo dimostrato in precedenza che $T(n) = O(n)$ e $T(n) = \Omega(n)$ possiamo concludere che $T(n) = \Theta(n)$ e quindi la funzione di costo cresce linearmente.

2.3.4 Metodo dell'esperto, o delle ricorrenze comuni

Ricorrenze comuni Esiste un'ampia classe di ricorrenze che si risolvono facilmente facendo ricorso a qualche teorema, ogni teorema è applicabile ad una particolare classe di ricorrenze.

Ricorrenze lineari con partizione bilanciata

Teorema 2.1. Siano a, b costanti intere tali che $a \geq 1$ e $b \geq 2$, e esistano c, β costanti reali tali che $c > 0$ e $\beta \geq 0$. Sia $T(n)$ data dalla seguente relazione di ricorrenza:

$$T(n) = \begin{cases} aT(n/b) + cn^\beta & n > 1 \\ 1 & n \leq 1 \end{cases} \quad (2.1)$$

Posto: $\alpha = \frac{\log a}{\log b} = \log_b a$ allora:

$$T(n) = \begin{cases} \Theta(n^\alpha) & \alpha > \beta \\ \Theta(n^\alpha \log n) & \alpha = \beta \\ \Theta(n^\beta) & \alpha < \beta \end{cases} \quad (2.2)$$

Assunzioni Assumiamo che n sia una potenza intera di b , ovvero $n = b^k, k = \log_b n$.

Influisce sul risultato?

- Supponendo che l'input abbia dimensione $b^k + 1$
- Estendiamo l'input fino alla dimensione b^{k+1} (**padding**)
- L'input è stato esteso di un fattore b
- Il che non cambia la complessità computazionale

Dimostrazione caso 1**Dimostrazione caso 2** $\alpha = \beta$

Dimostrazione. Ne segue che: $q = b^{\alpha-\beta} = 1$ e dunque la funzione $T(n)$:

$$T(n) = dn^\alpha + cb^{k\beta} \sum_{i=0}^{k-1} q^i \quad (2.3)$$

$$= n^\alpha d + cn^\beta k \quad q^i = 1^i = 1 \quad (2.4)$$

$$= n^\alpha d + cn^\alpha k \quad \alpha = \beta \quad (2.5)$$

$$= n^\alpha (d + ck) \quad (2.6)$$

$$= n^\alpha \left[d + c \frac{\log n}{\log b} \right] \quad k = \log_b n \quad (2.7)$$

□

e come conseguenza $T(n) = \Theta(n^\alpha \log n)$

Ricorrenze lineari con partizione bilanciata (Estesa)

Teorema 2.2. Sia $a \geq 1, b > 1, f(n)$ asintoticamente positiva e sia:

$$T(n) = \begin{cases} aT(n/b) + f(n) & n > 1 \\ \Theta(1) & n \leq 1 \end{cases}$$

Sia $\alpha = \log_b a$ si distinguono i seguenti casi:

(1)	$\exists \epsilon > 0 : f(n) = O(n^{\alpha-\epsilon})$	\Rightarrow	$T(n) = \Theta(n^\alpha)$
(2)	$f(n) = \Theta(n^\alpha)$	\Rightarrow	$T(n) = \Theta(f(n) \log n)$
(3)	$\exists \epsilon > 0 : f(n) = \Omega(n^{\alpha+\epsilon}) \wedge$ $\exists c : 0 < c < 1,$ $\exists m \geq 0 : af(n/b) \leq cf(n),$ $\forall n \geq m$	\Rightarrow	$T(n) = \Theta(f(n))$

Capitolo 3

Alberi

3.1 Introduzione

Albero Radicato (*Rooted Tree*)

Definizione 3.1. Un albero intero consiste in un insieme di nodi orientati che connettono coppie di nodi, con le seguenti proprietà:

1. Un nodo dell'albero è designato come nodo **radice**
2. Ogni nodo n , a parte la radice, ha esattamente un arco entrante
3. Esiste un cammino unico dalla radice ad ogni nodo
4. L'albero è connesso

Albero Radicato Ricorsivo

Definizione 3.2. Un albero è dato da:

1. Un insieme vuoto, oppure
2. Un nodo **radice** e zero o più **sotto-alberi**, ognuno dei quali è un albero.

La radice è connessa ai sotto-alberi tramite archi orientati.

Profondità nodi (*Depth*)

Definizione 3.3. Si definisce **profondità** di un nodo n la lunghezza del cammino semplice dalla radice al nodo n (misurato in numero di archi).

Livello (*Level*)

Definizione 3.4. Si definisce come **livello** di un albero l'insieme di tutti i nodi alla stessa profondità.

Altezza dell'albero (*Height*)

Definizione 3.5. Si definisce come **altezza** di un albero la profondità massima dei nodi dell'albero.

3.2 Alberi Binari

Definizione 3.6. Un **albero binario** è un albero radicato nel quale ogni nodo ha al massimo due figli, identificati come figlio **sinistro** e figlio **destro**.

Specifica delle API

Algorithm 7 Tree

```
% Costituisce un nuovo nodo, contenente v, senza figli o genitori.
Tree(Item v)
% Legge il valore memorizzato nel nodo
Item read()
% Modifica il valore memorizzato nel nodo
write(Item v)
% Restituisce il padre, oppure nil se questo nodo è radice
Tree parent()
% Restituisce il figlio sinistro (destro) di questo nodo; restituisce nil se assente
Tree left()
Tree right()
% Inserisce il sotto-albero radicato in t come figlio sinistro (destro) di questo nodo
insertLeft(Tree t)
insertRight(Tree t)
% Distrugge (ricorsivamente) il figlio sinistro (destro) di questo nodo
deleteLeft()
deleteRight()
```

3.2.1 Implementazione

Campi memorizzati nei nodi:

parent Puntatore al nodo padre

left Puntatore al figlio sinistro

right Puntatore al figlio destro

Operazioni di base: Implementazione operazioni API:

Algorithm 8 Tree

```
function TREE(ITEM v)
  TREE t ← new TREE
  t.value ← v
  t.left ← t.right ←
  t.parent ← nil
  return t

function INSERTLEFT(TREE t)
  if left == nil then
    left ← t
    t.parent ← this

function INSERTRIGHT(TREE t)
  if right == nil then
    right ← t
    t.parent ← this

function DELETELEFT
  if left ≠ nil then
    left.deleteLeft()
    left.deleteRight()
    left ← nil
```

```

function DELETERIGHT
  if right  $\neq$  nil then
    right.deleteLeft()
    right.deleteRight()
    right  $\leftarrow$  nil

```

3.2.2 Visite

Visita di un albero / ricerca Una **visita** è una strategia per analizzare (visitare) tutti i nodi di un albero. Le visite possono essere:

Visita in profondità (*Depth-First search* DFS) Usata per visitare un albero, si visita ricorsivamente ognuno dei suoi **sotto-alberi**.

Tre varianti: pre/in/post visita

Richiede uno **stack**

Visita in ampiezza (*Breadth-First search* BFS) Usata per visitare ogni **livello** dell'albero, partendo dalla radice.

Richiede una **queue**

Algorithm 9 dfs(TREE *t_i*)

```

if t  $\neq$  nil then
  % pre-order visit of t
  print t
  dfs(t.left)
  % in-order visit of t
  print t
  dfs(t.right)
  % post-order visit of t
  print t

```

Esemmpi di applicazione

Contare i nodi in post-visita:

Algorithm 10 countNodes(TREE *t*)

```

if t  $\neq$  nil then
  c  $\leftarrow$  1
  c  $\leftarrow$  c + countNodes(t.left)
  c  $\leftarrow$  c + countNodes(t.right)
  return c
else
  return 0

```

Stampare espressioni con In-visita:

Algorithm 11 `int printExpression(TREE t)`

```

if  $t.\text{left}() == \text{nil}$  and  $t.\text{right}() == \text{nil}$  then
    print  $t.\text{read}()$ 
else
    print "("
    print  $\text{printExpression}(t.\text{left}())$ 
    print  $t.\text{read}()$ 
    print  $\text{printExpression}(t.\text{right}())$ 
    print ")"

```

3.3 Alberi Generici

Definizione 3.7. Un **albero generico** è un albero radicato nel quale ogni nodo può avere un numero arbitrario di figli.

Definizione delle API

Algorithm 12 `Tree`

```

% Costituisce un nuovo nodo, contenente  $v$ , senza figli o genitori.
Tree(Item  $v$ )
% Legge il valore memorizzato nel nodo
Item read()
% Modifica il valore memorizzato nel nodo
write(Item  $v$ )
% Restituisce il padre, oppure nil se questo nodo è radice
Tree parent()
% Restituisce il primo figlio, oppure nil se è una foglia
Tree leftmostChild()
% Restituisce il prossimo fratello, oppure nil se è l'ultimo figlio
Tree rightSibling()
% Inserisce il sotto-albero radicato in  $t$  come primo figlio di questo nodo
insertChild(Tree  $t$ )
% Inserisce il sotto-albero radicato in  $t$  come prossimo fratello di questo nodo
insertSibling(Tree  $t$ )
% Distrugge (ricorsivamente) l'albero radicato identificato dal primo figlio di questo nodo
deleteChild()
% Distrugge (ricorsivamente) l'albero radicato identificato dal prossimo fratello di questo nodo
deleteSibling()

```

3.3.1 Visite

Breadth-First Search

Algorithm 13 `bfs(TREE t)`

```

QUEUE  $Q = \text{Queue}()$ 
 $Q.\text{enqueue}(t)$ 
while not  $Q.\text{isEmpty}()$  do
    TREE  $u \leftarrow Q.\text{dequeue}()$ 
    print  $u$ 
     $u \leftarrow u.\text{leftmostChild}()$ 

```

```
while  $u \neq \text{nil}$  do  
     $Q.\text{enqueue}(u)$   
     $u \leftarrow u.\text{rightSibling}()$ 
```

Memorizzazione

Esistono diversi modi per memorizzare un albero, più o meno indicati a seconda del numero massimo e medi dei figli presenti:

1. Realizzazione con vettore di figli
2. Realizzazione con primo figlio e prossimo fratello
3. Realizzazione con vettore dei padri

Capitolo 4

Alberi Binari di Ricerca

4.1 Alberi Binari di Ricerca

Dizionario

Definizione 4.1. Un **dizionario** è una struttura dati che implementa le seguenti funzionalità:

- `Item lookup(Item k)`: restituisce l'elemento con chiave k se presente nel dizionario.
- `insert(Item k, Item v)`: inserisce l'elemento i con chiave k e valore v nel dizionario.
- `remove(Item k)`: elimina l'elemento con chiave k dal dizionario.

Possibili Implementazioni di seguito sono riportate le possibili implementazioni di un dizionario:

Struttura	lookup	insert	remove
Vettore Ordinato	$O(\log n)$	$O(n)$	$O(n)$
Vettore non Ordinato	$O(n)$	$O(1)^*$	$O(1)^*$
Lista non Ordinata	$O(n)$	$O(1)$	$O(1)^*$

* Assumendo che l'elemento sia già stato trovato, altrimenti $O(n)$.

Idea ispiratrice Portare l'idea di ricerca binaria negli alberi.

Memorizzazione

- Le **associazioni chiave-valore** vengono memorizzate in un albero binario
- Ogni nodo u contiene una coppia: $(u.key, u.value)$
- Le chiavi devono appartenere ad un insieme **totalmente ordinato**

Proprietà

1. Le chiavi contenute nei nodi del sotto-albero sinistro di un nodo u sono minori di $u.key$
2. Le chiavi contenute nei nodi del sotto-albero destro di un nodo u sono maggiori di $u.key$

Specifica

Getters

- **Item** `key()`: restituisce la chiave dell'elemento memorizzato nel nodo
- **Item** `value()`: restituisce il valore dell'elemento memorizzato nel nodo
- **Node** `left()`: restituisce il figlio sinistro del nodo
- **Node** `right()`: restituisce il figlio destro del nodo
- **Node** `parent()`: restituisce il genitore del nodo

Dizionario

- **Item** `lookup(Item k)`: restituisce l'elemento con chiave k se presente nel dizionario
- `insert(Item k, Item v)`: inserisce l'elemento i con chiave k e valore v nel dizionario
- `remove(Item k)`: elimina l'elemento con chiave k dal dizionario

Ordinamento

- **Tree** `successorNode(Node u)`: restituisce il nodo con chiave successiva a $u.key$
- **Tree** `predecessorNode(Node u)`: restituisce il nodo con chiave precedente a $u.key$
- **Tree** `min()`: restituisce il nodo con chiave minima
- **Tree** `max()`: restituisce il nodo con chiave massima

Funzioni interne

- **Node** `lookupNode(Tree T, Item k)`: restituisce il nodo con chiave k se presente nell'albero T
- **Node** `insertNode(Tree T, Item k, Item v)`: inserisce l'elemento i con chiave k e valore v nell'albero T
- **Node** `removeNode(Tree T, Item k)`: elimina l'elemento con chiave k dall'albero T

4.1.1 Ricerca - lookupNode()

La funzione `Item lookup(Tree T, Item k)` restituisce il presente nell'albero T con chiave k se presente, altrimenti restituisce `nil`. Implementazione con dizionario:

Algorithm 14 `lookupNode(ITEM k)`

```

TREE  $t \leftarrow \text{lookupNode}(tree, k)$ 
if  $t \neq \text{nil}$  then
    return  $t.value()$ 
else
    return nil

```

Versione Iterativa:

Algorithm 15 `lookupNode(ITEM k)`

```

TREE  $t \leftarrow \text{root}()$ 
while  $t \neq \text{nil}$  and  $u.key \neq k$  do
    if  $k < t.key()$  then
         $t \leftarrow t.left()$ 
    else
         $t \leftarrow t.right()$ 
return  $t$ 

```

Versione Ricorsiva:

Algorithm 16 lookupNode(ITEM k)

```

function LOOKUPNODE(TREE  $t$ , ITEM  $k$ )
  if  $t = \text{nil}$  or  $t.\text{key}() = k$  then
    return  $t$ 
  if  $k < t.\text{key}()$  then
    return LOOKUPNODE( $t.\text{left}()$ ,  $k$ )
  else
    return LOOKUPNODE( $t.\text{right}()$ ,  $k$ )
  
```

4.1.2 Minimo & Massimo

Algorithm 17 TREE min(TREE t)

```

TREE  $u \leftarrow t$ 
while  $u.\text{left}() \neq \text{nil}$  do
   $u \leftarrow u.\text{left}()$ 
return  $u$ 
  
```

Algorithm 18 TREE max(TREE t)

```

TREE  $u \leftarrow t$ 
while  $u.\text{right}() \neq \text{nil}$  do
   $u \leftarrow u.\text{right}()$ 
return  $u$ 
  
```

Queste due funzioni sono implementabili in nel modo mostrato solo in quanto assumiamo che l'albero sia un albero binario di ricerca ben formato, se ciò non fosse vero, sarebbe necessario scorrere l'intero albero. (Non in questo capitolo)

4.1.3 Successore e Predecessore

Successore

Definizione 4.2. Il **successore** di un nodo u è il più piccolo nodo maggiore di u .

Per rispondere a questo problema, possiamo distinguere diversi casi:

1. Se u ha un figlio destro allora il successore sarà il minimo del sotto-albero destro
2. Se u non ha un figlio destro, allora bisognerà risalire l'albero fino a trovare il nodo radice di un sotto-albero che contiene u a sinistra

Algorithm 19 TREE successorNode(TREE u)

```

if  $u = \text{nil}$  then
  return  $t$ 
if  $u.\text{right}() \neq \text{nil}$  then
  return MIN( $u.\text{right}()$ )
  
```

▷ Se $u = \text{nil}$, non ha successore
 ▷ Caso 1 - Se u ha un figlio destro

```

else
    TREE  $p \leftarrow u.parent()$ 
    while  $p \neq \mathbf{nil}$  and  $u == p.right()$  do
         $u \leftarrow p$ 
         $p \leftarrow p.parent()$ 
    return  $p$ 

```

▷ Caso 2 - Se u non ha un figlio destro

Predecessore

Definizione 4.3. Il **predecessore** di un nodo u è il più grande nodo minore di u .

Per rispondere a questo problema, possiamo distinguere diversi casi:

1. Se u ha un figlio sinistro allora il predecessore sarà il massimo del sotto-albero sinistro
2. Se u non ha un figlio sinistro, allora bisognerà risalire l'albero fino a trovare il nodo radice di un sotto-albero che contiene u a destra

Algorithm 20 TREE predecessorNode(TREE u)

```

if  $u = \mathbf{nil}$  then
    return  $t$ 
if  $u.left() \neq \mathbf{nil}$  then
    return MAX( $u.left()$ )
else
    TREE  $p \leftarrow u.parent()$ 
    while  $p \neq \mathbf{nil}$  and  $u == p.left()$  do
         $u \leftarrow p$ 
         $p \leftarrow p.parent()$ 
    return  $p$ 

```

▷ Se $u = \mathbf{nil}$, non ha predecessore
 ▷ Caso 1 - Se u ha un figlio sinistro
 ▷ Caso 2 - Se u non ha un figlio sinistro

4.1.4 Inserimento - insertNode()

La funzione `insertNode(TREE t , ITEM k , ITEM v)` inserisce un'associazione chiave-valore (k, v) nell'albero t , se la chiave k è già presente, il valore viene aggiornato, se $t = \mathbf{nil}$, viene restituito un nuovo nodo con chiave k e valore v , altrimenti si restituisce l'albero t inalterato.

Implementazione dizionario Questa è l'implementazione del dizionario con la funzione `insertNode()`:

Algorithm 21 insertNode(ITEM k , ITEM v)

```

tree  $\leftarrow$  insertNode(tree,  $k, v$ )

```

Implementazione

Algorithm 22 TREE insertNode(TREE T , ITEM k , ITEM v)

```

TREE  $p \leftarrow \mathbf{nil}$ 
TREE  $u \leftarrow T$ 
while  $u \neq \mathbf{nil}$  and  $u.key() \neq k$  do
     $p \leftarrow u$ 
     $u \leftarrow \text{iff}(k < u.key(), u.left(), u.right())$ 
if  $u \neq \mathbf{nil}$  and  $u.key() == k$  then
     $u.value \leftarrow v$ 

```

```

else
    TREE new ← new ITEM (k, v) LINK(p, new, k)
    if p == nil then
        T ← new
    return T

```

Definizione della funzione `link()`:

Algorithm 23 `link(TREE p, TREE u, ITEM k)`

```

if u ≠ nil then
    u.parent ← p
if p ≠ nil then
    if k < p.key() then
        p.left ← u
    else
        p.right ← u

```

4.1.5 Cancellazione - `remove()`

La funzione `tree = removeNode(TREE t, ITEM k)` elimina l'elemento con chiave k dall'albero t , se la chiave k non è presente, l'albero t viene restituito inalterato.

In ogni caso bisogna prima cercare il nodo all'interno dell'albero poi il procedimento da eseguire dipende dai figli del nodo da eliminare. Assumiamo che il nodo da eliminare sia u e il suo genitore sia p .

Caso 1 - Nessun figlio

Se il nodo da eliminare non ha figli, allora basta eliminare il nodo e aggiornare il genitore del nodo da eliminare.

Caso 2 - Un figlio

Se il nodo da eliminare ha un solo figlio, allora dato che l'albero è un albero binario di ricerca, il figlio del nodo da eliminare può essere spostato al posto del nodo da eliminare, in quanto il figlio è maggiore o minore del genitore del nodo da eliminare.

Caso 3 - Due figli

Se il nodo da eliminare ha due figli allora le cose si complicano.

1. Identificare il nodo successore s del nodo da eliminare, per definizione il successore non ha figlio sinistro.
2. Si prende l'albero che ha come radice il nodo s e si "stacca" dal resto dell'albero
3. L'eventuale sotto-albero destro del nodo da eliminare viene attaccato al nodo padre di s
4. Ora il nodo s non ha figlio destro può essere spostato al posto del nodo da eliminare

Implementazione

Algorithm 24 `TREE removeNode(TREE T, ITEM k)`

```

TREE t
TREE u ← lookupNode(T, k)
if u ≠ nil then

```

▷ Cerco il nodo da eliminare

```

if  $u.\text{left}() == \text{nil}$  and  $u.\text{right}() == \text{nil}$  then                                ▷ Caso 1 - Nessun figlio
    LINK( $u.\text{parent}()$ , nil,  $k$ )
    delete  $u$ 
else if  $u.\text{left}() \neq \text{nil}$  and  $u.\text{right}() \neq \text{nil}$  then                    ▷ Caso 3 - Due figli
    TREE  $s \leftarrow u.\text{successorNode}()$ 
    LINK( $u.\text{parent}()$ ,  $s.\text{right}()$ ,  $s.\text{key}()$ )
     $u.\text{key} \leftarrow s.\text{key}()$ 
     $u.\text{value} \leftarrow s.\text{value}()$ 
    delete  $s$ 
else if  $u.\text{left}() == \text{nil}$  and  $u.\text{right}() \neq \text{nil}$  then                ▷ Caso 2 - Un figlio destro
    LINK( $u.\text{parent}()$ ,  $u.\text{right}()$ ,  $k$ )
    if  $u.\text{parent}() == \text{nil}$  then
         $T \leftarrow u.\text{right}()$ 
    delete  $u$ 
else                                                                    ▷ Caso 2 - Un figlio sinistro
    LINK( $u.\text{parent}()$ ,  $u.\text{left}()$ ,  $k$ )
    if  $u.\text{parent}() == \text{nil}$  then
         $T \leftarrow u.\text{left}()$ 
    delete  $u$ 
return  $T$ 

```

Dimostrazione. **Caso 1: Nessun figlio** Eliminare foglie non cambia l'ordinamento dell'albero.

Caso 2: Un figlio (destro o sinistro) Se u è il figlio destro (o sinistro) di p , allora tutti i valori nel sotto-albero di f sono maggiori (o minori) di p . Quindi f può essere spostato come figlio destro (o sinistro) di p al posto di u .

Caso 3: Due figli Il successore s è sicuramente \geq dei nodi nel sotto-albero sinistro di u lo stesso successore è \leq dei nodi nel sotto-albero destro di u . Quindi s può essere spostato al posto di u . E ora si ha da gestire il sotto-albero destro di s gestibile con il caso 2. \square

4.2 Alberi Binari di Ricerca Bilanciati

4.2.1 Definizione

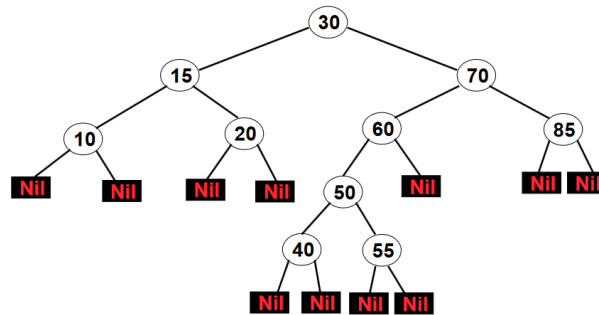
Definizione 4.4. L'altezza di ABR nel caso pessimo è $O(n)$, dove n è il numero di nodi.

Definizione 4.5. L'altezza di un ABR nel caso medio dipende dall'ordine di inserimento delle chiavi ed è $O(\log n)$. Nel caso generale di inserimenti e cancellazioni casuali non è presente una garanzia sull'altezza.

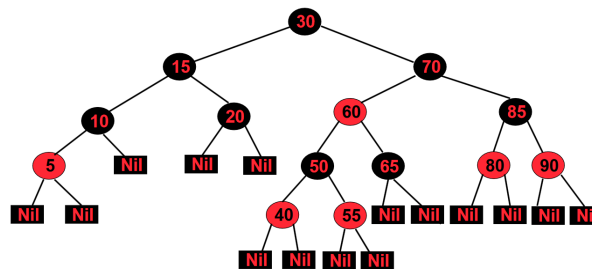
4.2.2 Alberi *Red-Black*

Gli alberi *Red-Black* sono alberi binari di ricerca in cui:

- Ogni nodo è colorato di **rosso** o **nero**
- Le **chiavi** vengono mantenute **solo nei nodi interni** dell'albero
- Le foglie sono costituite da nodi **fittizi** colorati di nero (Nil)
- Vengono rispettati i seguenti vincoli:
 1. La radice è nera
 2. Tutte le foglie sono nere
 3. Entrambi i figli di un nodo rosso sono neri

Figura 4.2: Esempio di albero non *Red-Black*

4. Ogni cammino semplice da un nodo u ad una delle foglie contenute nel suo sotto-albero ha lo stesso numero di nodi neri

Figura 4.1: Esempio di albero *Red-Black*

Se un albero generico è "troppo" sbilanciato, allora potrebbe non rispettare le proprietà di un albero *Red-Black*.

4.2.3 Inserimento

Quando si va a modificare la struttura dell'albero allora è possibile che determinate condizioni vengano violate, per questo motivo è necessario introdurre delle operazioni di **ri-bilanciamento** come la **rotazione** e il **ri-coloramento**. Le rotazioni a loro volta possono essere di due tipi: **sinistra** e **destra**.

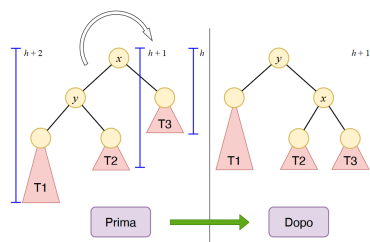
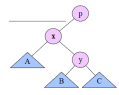


Figura 4.3: Esempio di rotazione a destra

Rotazione a destra

Rotazione a sinistra Assumiamo che la situazione all'interno del nostro albero di ricerca sia la seguente:

Per una rotazione a sinistra si seguono questi passaggi:



1. far diventare B figlio destro di x
2. far diventare x figlio sinistro di y
3. far diventare y figlio di p dove p è il genitore vecchio di y

Implementazione di tale rotazione:

Algorithm 25 rotateLeft(TREE x)

```

TREE  $y \leftarrow x.$ right()
TREE  $p \leftarrow x.$ parent()
 $x.$ right  $\leftarrow y.$ left()
if  $y.$ left()  $\neq$  nil then
     $y.$ left.parent  $\leftarrow x$ 
 $y.$ left  $\leftarrow x$ 
 $x.$ parent  $\leftarrow y$ 
 $y.$ parent  $\leftarrow p$ 
if  $p \neq$  nil then
    if  $p.$ left() ==  $x$  then
         $p.$ left  $\leftarrow y$ 
return  $y$ 

```

Inserimento in alberi *Red-Black* Per alberi del genere *Red-Black* l'inserimento di un nodo può violare le proprietà dell'albero, di base inseriamo il nodo come un nodo rosso e eseguiamo il normale inserimento per un albero binario di ricerca. Dopo l'inserimento è necessario verificare se le proprietà n. 3 e n. 4 sono state violate, in tal caso è necessario eseguire delle operazioni di **ri-bilanciamento**. La funzione

Algorithm 26 insertNode(TREE T , ITEM k , ITEM v)

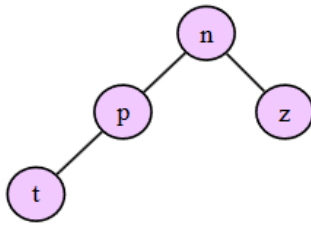
```

TREE  $p \leftarrow$  nil                                ▷ Genitore del nodo da inserire
TREE  $u \leftarrow T$ 
while  $u \neq$  nil and  $u.$ key()  $\neq k$  do             ▷ Cerco la posizione del nodo da inserire
     $p \leftarrow u$ 
     $u \leftarrow$  iff( $k < u.$ key(),  $u.$ left(),  $u.$ right())
if  $u \neq$  nil and  $u.$ key() ==  $k$  then
     $u.$ value  $\leftarrow v$                                 ▷ Aggiorno il valore in quanto la chiave è già presente
else
    TREE  $new \leftarrow$  new ITEM ( $k, v$ )
    LINK( $p, new, k$ )
    BALANCEINSERT( $new$ )
    if  $p ==$  nil then
         $T \leftarrow new$                                 ▷ Se il genitore è nullo, allora il nuovo nodo è la radice
return  $T$                                              ▷ Restituisco l'albero

```

balanceInsert() è una funzione che si occupa di ri-bilanciare l'albero dopo l'inserimento di un nodo, questa funzione è necessaria in quanto l'inserimento di un nodo potrebbe violare le proprietà dell'albero *Red-Black*.

Il funzionamento di linea generale prevede: lo spostamento verso l'alto lungo il percorso di inserimento, ripristinare il vincolo dei figli di un nodo rosso che devono essere neri, spostare le violazioni verso l'alto (rotazione) rispettando il vincolo dei nodi neri (ri-coloramento). Terminiamo la funzione con il ri-coloramento della radice se necessario.



I nodi coinvolti ricorsivamente sono i seguenti:

- Il nodo inserito t
- Suo padre p
- Suo nonno n
- Suo zio z

La seguente testata di funzione è comune a tutte le funzioni di ri-bilanciamento:

Algorithm 27 `balanceInsert(TREE t)`

 TREE $p \leftarrow t.\text{parent}()$

 TREE $n \leftarrow \text{iff}(p \neq \text{nil}, p.\text{parent}(), \text{nil})$

 TREE $z \leftarrow \text{iff}(n = \text{nil}, \text{nil}, \text{iff}(n.\text{left}() == p, n.\text{right}(), n.\text{left}()))$

È possibile distinguere l'inserimento in 7 casi differenti quali:

Caso 1 Nuovo nodo t non ha padre, dunque è la radice dell'albero. In tal caso, lo coloriamo di nero.

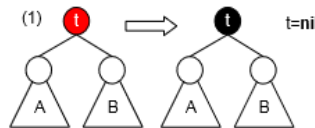


Figura 4.4: Nuovo nodo t come radice.

Caso 2 Il padre p di t è nero; in tal caso, non c'è nessuna violazione delle proprietà dell'albero *Red-Black* e inseriamo il nodo t come nodo rosso.

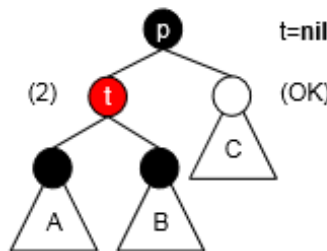
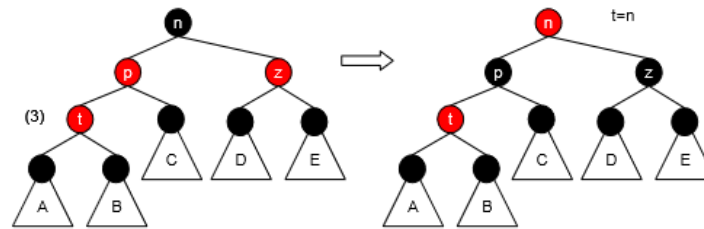


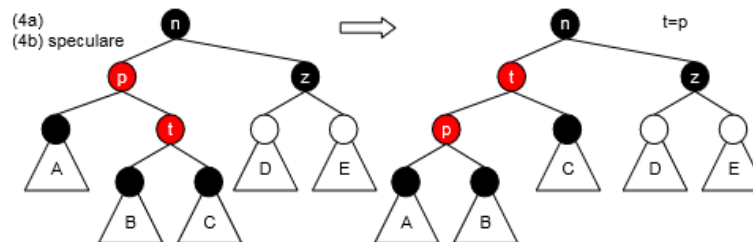
Figura 4.5: Il padre p di t è nero.

Caso 3 Caso in cui l'elemento da inserire sia rosso e il padre sia rosso e lo zio sia rosso. Coloriamo dunque p e z di nero e n di rosso (l'altezza nera rimane invariata). La problematica ora sorge sul nonno n che potrebbe violare le proprietà 1 e/o 3 dell'albero *Red-Black*, in tal caso dobbiamo eseguire una ricorsione su n ponendo $t = n$ e ripetendo il processo.



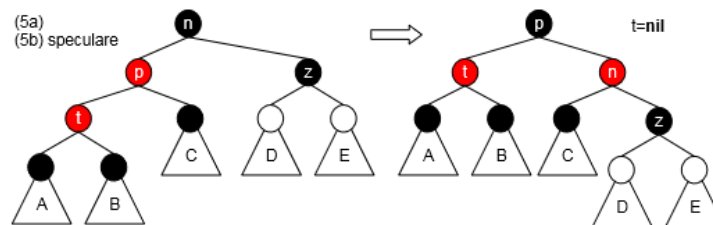
Caso 4 (a,b) In questo caso t è rosso, p è rosso e z è nero. Questo caso si divide in due sotto-casi ma il procedimento è speculare per entrambi (sostituendo *left* con *right* e viceversa).

Assumendo che t sia figlio destro di p e che p sia figlio sinistro di n , allora eseguiamo una rotazione a sinistra su p in modo da rendere t figlio sinistro di n con p come figlio sinistro di t . Ora abbiamo una violazione delle proprietà 3 e 4. Ricadiamo però nel caso 5(a) in quanto p è figlio sinistro di n e t è figlio sinistro di p .



Caso 5 (a,b) In questo caso t è rosso, p è rosso e z è nero. Questo caso si divide in due sotto-casi, come il precedente, ma il procedimento è speculare per entrambi (sostituendo *left* con *right* e viceversa).

Assumendo che t sia figlio sinistro di p e che p sia figlio sinistro di n , allora eseguiamo una rotazione a destra su n in modo da rendere p nodo radice, t figlio sinistro di p e n figlio destro di p . Ora coloriamo p di nero e n di rosso. In quanto la posizione relativa di z rispetto a n non cambia allora i vincoli per l'albero *Red-Black* sono rispettati.



Conclusioni I casi 1 e 2 non richiedono ulteriori operazioni, i casi 3, 4 e 5 richiedono una rotazione e un ri-coloramento (con eventuali ricorsioni). In generale, l'inserimento di un nodo in un albero *Red-Black* richiede $O(\log n)$ operazioni. È implementabile con il seguente pseudo-codice:

Algorithm 28 balanceInsert(TREE t)

```

 $t$ .color  $\leftarrow$  RED
while  $t \neq \mathbf{nil}$  do
    TREE  $p \leftarrow t$ .parent()
    TREE  $n \leftarrow \text{iff}(p \neq \mathbf{nil}, p.\text{parent}(), \mathbf{nil})$ 
    TREE  $z \leftarrow \text{iff}(n = \mathbf{nil}, \mathbf{nil}, \text{iff}(n.\text{left}() == p, n.\text{right}(), n.\text{left}()))$ 
    if  $p = \mathbf{nil}$  then                                     ▷ Caso 1
         $t$ .color  $\leftarrow$  BLACK
         $t \leftarrow \mathbf{nil}$ 
    else if  $p$ .color == BLACK then                       ▷ Caso 2
         $t \leftarrow \mathbf{nil}$ 
    else if  $z$ .color == RED then                         ▷ Caso 3
         $p$ .color  $\leftarrow$  BLACK
         $z$ .color  $\leftarrow$  BLACK
         $n$ .color  $\leftarrow$  RED
         $t \leftarrow n$ 
    else
        if  $t == p.\text{right}()$  and  $p == n.\text{left}()$  then   ▷ Caso 4.a
            ROTATELEFT( $p$ )
             $t \leftarrow p$ 
        else if  $t == p.\text{left}()$  and  $p == n.\text{right}()$  then ▷ Caso 4.b
            ROTATERIGHT( $p$ )
             $t \leftarrow p$ 
        else
            if  $t == p.\text{left}()$  and  $p == n.\text{left}()$  then   ▷ Caso 5.a
                ROTATERIGHT( $n$ )
            else if  $t == p.\text{right}()$  and  $p == n.\text{right}()$  then ▷ Caso 5.b
                ROTATELEFT( $n$ )
             $p$ .color  $\leftarrow$  BLACK
             $n$ .color  $\leftarrow$  RED

```

4.2.4 Teoremi su un albero *Red-Black*

Teorema 4.1. In un albero RB, un sotto-albero di radice u contiene $n \geq 2^{\text{bh}(u)} - 1$ nodi interni (nodi senza foglie fittizie).

Dimostrazione. Si procede per ricorsione sull'altezza dell'albero:

Base: se $h = 0$ allora u è una foglia **Nil** e il sotto-albero con radice u contiene: $n \geq 2^{\text{bh}(u)} - 1 = 2^0 - 1 = 0$ nodi interni. ✓

Passo: supponendo che $h > 1$ e che la tesi sia vera per alberi di altezza $< h$, Allora u è un nodo interno con due figli non fittizi. Inoltre ogni figlio v di u ha un'altezza nera $\text{bh}(v)$ pari a $\text{bh}(u)$ se v è rosso o $\text{bh}(u) - 1$ se v è nero. Quindi il sotto-albero con radice v contiene almeno $2^{\text{bh}(v)} - 1$ nodi interni, per ipotesi induttiva. In quanto considerando anche il nodo u , allora il numero di nodi interni nel sotto-albero con radice u è almeno $n \geq 2 \cdot (2^{\text{bh}(v)} - 1) + 1 = 2^{\text{bh}(u)} - 2 + 1 = 2^{\text{bh}(u)} - 1$. ✓ □

Teorema 4.2. In un albero RB, almeno la metà dei nodi dalla radice ad una foglia sono neri.

Dimostrazione. Per il vincolo (2) di un albero RB, se un nodo è rosso allora i suoi figli devono essere neri. Quindi la situazione nella quale si ottengono più nodi rossi possibili è nel caso questi siano con colori alterni. Quindi almeno la metà dei nodi dalla radice ad una foglia sono neri. □

Teorema 4.3. In un albero RB dati due cammini dalla radice a due foglie non è possibile che uno sia più lungo del doppio dell'altro.

Dimostrazione. Per il vincolo (4) di un albero RB, ogni cammino dalla radice ad una foglia deve avere lo stesso numero di nodi neri. Per il teorema precedente almeno la metà dei nodi in ognuno di questi cammini sono neri.

Quindi al limite uno dei due cammini è costituito solo da nodi neri e l'altro è costituito da nodi alternati neri e rossi, rendendo la lunghezza del cammino con nodi alternati esattamente il doppio del cammino con nodi neri per (4). \square

Teorema 4.4. L'altezza massima di un albero RB con n nodi è al più $2 \log(n + 1)$.

Dimostrazione.

$$\begin{aligned} n \geq 2^{\text{bh}(r)} - 1 &\Leftrightarrow n \geq \overbrace{2^{\frac{h}{2}}}^{\text{bh}(r) \leq \frac{h}{2}} - 1 \\ &\Leftrightarrow n + 1 \geq 2^{\frac{h}{2}} \\ &\Leftrightarrow \log(n + 1) \geq \frac{h}{2} \\ &\Leftrightarrow 2 \log(n + 1) \geq h \end{aligned}$$

\square

Dunque conseguenza di questo teorema è che la complessità totale di un albero RB è $O(\log n)$, in quanto:

1. $O(\log n)$ per scendere fino al punto di inserimento del nodo
2. $O(1)$ per inserire il nodo
3. $O(\log n)$ per risalire e ri-bilanciare l'albero (caso peggiore caso: 3)

4.2.5 Cancellazione

La cancellazione di un nodo in un albero *Red-Black* è più complessa rispetto all'inserimento, in quanto la cancellazione di un nodo potrebbe violare le proprietà dell'albero. La procedura di cancellazione è simile a quella di un albero binario di ricerca, ma con delle operazioni di ri-bilanciamento. La procedura di cancellazione è composta da 8 casi differenti, con 4 casi principali e 4 casi simmetrici.

In generale:

- Se il nodo da eliminare è rosso allora:

Altezza nera invariata

Non sono stati creati nodi rossi consecutivi

La radice resta nera

- Se il nodo da eliminare è nero allora:

Potrebbe essere violato il vincolo 1 in quanto la radice potrebbe essere diventata rossa

Potrebbe essere violato il vincolo 3 in quanto potrebbero esserci nodi rossi consecutivi se padre e figlio sono rossi

Potrebbe essere violato il vincolo 4 in quanto l'altezza nera è diminuita

Algorithm 29 balanceDelete(TREE T , TREE t)

```

while  $t \neq \text{null}$  and  $t.\text{color}() = \text{black}$  do
    TREE  $p \leftarrow t.\text{parent}()$                                 ▷ Ottengo il genitore del nodo da eliminare
    if  $t = p.\text{left}()$  then                                    ▷ Sottocasi con  $t$  figlio sinistro
        TREE  $f \leftarrow p.\text{right}()$                             ▷ Ottengo il fratello del nodo da eliminare
        TREE  $ns \leftarrow f.\text{left}()$                             ▷ Ottengo il nipote sinistro del fratello
        TREE  $nd \leftarrow f.\text{right}()$                            ▷ Ottengo il nipote destro del fratello
        if  $f.\text{color}() = \text{red}$  then                                ▷ Caso 1
             $p.\text{color} \leftarrow \text{red}$ 
             $f.\text{color} \leftarrow \text{black}$ 
            ROTATELEFT( $p$ )
        else
            if  $ns.\text{color}() = \text{black}$  and  $nd.\text{color}() = \text{black}$  then    ▷ Caso 2
                 $f.\text{color} \leftarrow \text{red}$ 
                 $t \leftarrow p$ 
            else if  $ns.\text{color}() = \text{red}$  and  $nd.\text{color}() = \text{black}$  then    ▷ Caso 3
                 $ns.\text{color} \leftarrow \text{black}$ 
                 $f.\text{color} \leftarrow \text{red}$ 
                ROTATERIGHT( $f$ )
            else if  $nd.\text{color}() = \text{red}$  then                                ▷ Caso 4
                 $f.\text{color} \leftarrow p.\text{color}()$ 
                 $p.\text{color} \leftarrow \text{black}$ 
                 $nd.\text{color} \leftarrow \text{black}$ 
                ROTATELEFT( $p$ )
                 $t \leftarrow T$ 
    else
        ▷ Sottocasi con  $t$  figlio destro, tralasciati in quanto simmetrici

```

Dunque seppure complicata la cancellazione risulta efficiente in quanto:

- Dal caso (1) si passa ad uno dei casi (2,3,4)
- Dal caso (2) si risale ad uno degli altri casi, ma si risale di un livello
- Dal caso (3) si passa al caso (4)
- Nel caso (4) si termina

La complessità totale di una cancellazione in un albero *Red-Black* è $O(\log n)$.

Capitolo 5

Grafi

5.1 Introduzione

Problemi relativi ai grafi Per lo scopo del corso i nostri obiettivi riguardanti i grafi li possiamo dividere in due categorie:

Problemi in grafi non pesati Studieremo problemi riguardanti grafi non pesati, ovvero grafi in cui gli archi non hanno un peso associato. In particolare, ci occuperemo di:

- Ricerca del cammino più breve tra due nodi.
- Componenti (fortemente) connesse, verifica ciclicità, ordinamento topologico.

Problemi in grafi pesati Studieremo problemi riguardanti grafi pesati, ovvero grafi in cui gli archi hanno un peso associato. In particolare, ci occuperemo di:

- Cammini di peso minimo.
- Alberi di copertura di peso minimo.
- Flusso massimo.

5.1.1 Definizioni

Grafo Orientato (*directed*)

Definizione 5.1. Un grafo orientato è una coppia $G = (V, E)$ dove V è un insieme finito di nodi (*node*) o vertici (*vertex*) ed E è un insieme finito di coppie di nodi (u, v) detti anche archi (*edge*) o lati (*link*) orientati.

Grafo Non Orientato (*undirected*)

Definizione 5.2. Un grafo non orientato è una coppia $G = (V, E)$ dove V è un insieme finito di nodi (*node*) o vertici (*vertex*) ed E è un insieme finito di coppie di nodi non orientati (u, v) detti anche archi (*edge*) o lati (*link*) non orientati.

Vertici

Definizione 5.3 (Adiacenza). Un vertice v è detto **adiacente** ad un vertice u se esiste un arco (u, v) .

Definizione 5.4 (Incidenza). Un arco (u, v) è detto **incidente** al vertice u e al vertice v .

In un grafo indiretto la relazione di adiacenza è simmetrica, ovvero se v è adiacente ad u allora u è adiacente a v .

Dimensioni del grafo

Numero di nodi: $|V| = n$

Numero di archi: $|E| = m$

Teorema 5.1 (Relazioni tra n e m). In un grafo non orientato con n nodi e m archi vale che $m \leq \frac{n(n-1)}{2} = O(n^2)$.

In un grafo orientato con n nodi e m archi vale che $m \leq n(n-1) = O(n^2)$.

La complessità è espressa in termini di n e m es. $O(n+m)$.

Casi Speciali

Definizione 5.5 (Grafo Completo). Un grafo con un arco fra tutte le coppie di nodi è detto **grafo completo**.

Definizione 5.6 (Grafo sparso/denso (informale)). Si dice che un grafo è **sparso** se ha "pochi archi", ovvero grafi con $m = O(n)$, $O(n \log n)$, e **denso** se ha "molti archi", ovvero grafi con $m = \Omega(n^2)$.

Definizione 5.7 (Albero libero). Un **albero libero** (*free tree*) è un grafo connesso con $m = n - 1$.

Definizione 5.8 (Albero radicato). Un **albero radicato** (*rooted tree*) è un albero libero in cui uno dei nodi è designato come radice.

Proprietà

Definizione 5.9 (Grado). Nei grafi non orientati il **grado** (*degree*) di un nodo è il numero di archi incidenti su di esso.

Nei grafi orientati si distinguono il **grado entrante** e il **grado uscente** di un nodo, rispettivamente il numero di archi entranti e uscenti da esso.

Cammino

Definizione 5.10 (Cammino). In un grafo $G = (V, E)$ orientato o meno, un **cammino** C di lunghezza k tra i nodi u_0, u_1, \dots, u_k è una sequenza di nodi tale che $(u_i, u_{i+1}) \in E$ per $i = 0, \dots, k-1$.

5.1.2 Specifica

5.1.3 Memorizzazione

Matrice di aderenza - Grafi orientati

Se si sceglie di memorizzare un grafo tramite una matrice di aderenza allora si avrà una matrice A di dimensione $n \times n$ dove n è il numero di nodi del grafo. La cella A_{ij} sarà pari a 1 se esiste un arco tra il nodo i e il nodo j , 0 altrimenti.

Esempio Assumendo che il grafo orientato

$$G = (\{0, 1, 2, 3, 4, 5\}, \{(0, 1), (1, 2), (0, 3), (3, 0), (2, 3), (3, 4), (4, 2)\})$$

sia memorizzato tramite una matrice di aderenza si avrà la seguente matrice:

$$\begin{matrix} & 0 & 1 & 2 & 3 & 4 & 5 \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{matrix}$$

Lista di adiacenza - Grafi orientati

Se si sceglie di memorizzare un grafo tramite una lista di adiacenza allora si avrà una lista di n elementi, uno per ogni nodo del grafo. Ogni elemento della lista sarà a sua volta una lista contenente i nodi adiacenti al nodo corrispondente.

Esempio Assumendo che il grafo orientato

$$G = (\{0, 1, 2, 3, 4, 5\}, \{(0, 1), (1, 2), (0, 3), (3, 0), (2, 3), (3, 4), (4, 2)\})$$

sia memorizzato tramite una lista di adiacenza si avrà la seguente lista:

```

0 -> 1 -> 3
1 -> 2
2 -> 3
3 -> 0 -> 4
4 -> 2
5
```

Matrice di aderenza - Grafi non orientati

Se si sceglie di memorizzare un grafo tramite una matrice di aderenza allora si avrà una matrice A di dimensione $n \times n$ dove n è il numero di nodi del grafo. La cella A_{ij} sarà pari a 1 se esiste un arco tra il nodo i e il nodo j , 0 altrimenti. In un grafo non orientato la matrice sarà simmetrica rispetto alla diagonale principale.

Esempio Assumendo che il grafo non orientato

$$G = (\{0, 1, 2, 3, 4, 5\}, \{(0, 1), (1, 2), (0, 3), (2, 3), (3, 4), (4, 2)\})$$

sia memorizzato tramite una matrice di aderenza si avrà la seguente matrice:

$$\begin{array}{c}
\begin{array}{cccccc}
& 0 & 1 & 2 & 3 & 4 & 5 \\
0 & \left(\begin{array}{cccccc}
& 1 & 0 & 1 & 0 & 0 \\
& & 1 & 0 & 0 & 0 \\
& & & 1 & 1 & 0 \\
& & & & 1 & 0 \\
& & & & & 0 \\
& & & & & & 0
\end{array} \right)
\end{array}
\end{array}$$

Lista di adiacenza - Grafi non orientati

Se si sceglie di memorizzare un grafo tramite una lista di adiacenza allora si avrà una lista di n elementi, uno per ogni nodo del grafo. Ogni elemento della lista sarà a sua volta una lista contenente i nodi adiacenti al nodo corrispondente. In un grafo non orientato la lista di adiacenza non conterrà duplicati.

Esempio Assumendo che il grafo non orientato

$$G = (\{0, 1, 2, 3, 4, 5\}, \{(0, 1), (1, 2), (0, 3), (2, 3), (3, 4), (4, 2)\})$$

sia memorizzato tramite una lista di adiacenza si avrà la seguente lista:

```

0 -> 1 -> 3
1 -> 0 -> 2
2 -> 1 -> 3 -> 4
3 -> 0 -> 2 -> 4
4 -> 3 -> 2
5
```

Matrice di adiacenza - Grafici non orientati pesati

Se si sceglie di memorizzare un grafo tramite una matrice di adiacenza allora si avrà una matrice A di dimensione $n \times n$ dove n è il numero di nodi del grafo. La cella A_{ij} sarà pari al peso dell'arco tra il nodo i e il nodo j , 0 altrimenti. In un grafo non orientato la matrice sarà simmetrica rispetto alla diagonale principale.

Esempio Assumendo che il grafo non orientato pesato

$$G = (\{0, 1, 2, 3, 4, 5\}, \{(0, 1, 3), (1, 2, 4), (0, 3, 1), (2, 3, 4), (3, 4, 8), (4, 2, 7)\})$$

sia memorizzato tramite una matrice di adiacenza si avrà la seguente matrice:

$$\begin{matrix} & 0 & 1 & 2 & 3 & 4 & 5 \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \begin{pmatrix} & 3 & 0 & 1 & 0 & 0 \\ & & 4 & 0 & 0 & 0 \\ & & & 4 & 7 & 0 \\ & & & & 8 & 0 \\ & & & & & 0 \end{pmatrix} \end{matrix}$$

Lista di adiacenza - Grafici non orientati pesati

Se si sceglie di memorizzare un grafo tramite una lista di adiacenza allora si avrà una lista di n elementi, uno per ogni nodo del grafo. Ogni elemento della lista sarà a sua volta una lista contenente i nodi adiacenti al nodo corrispondente e il peso dell'arco. In un grafo non orientato la lista di adiacenza non conterrà duplicati.

Esempio Assumendo che il grafo non orientato pesato

$$G = (\{0, 1, 2, 3, 4, 5\}, \{(0, 1, 3), (1, 2, 4), (0, 3, 1), (2, 3, 4), (3, 4, 8), (4, 2, 7)\})$$

sia memorizzato tramite una lista di adiacenza si avrà la seguente lista:

```
0 -> 1(3) -> 3(1)
1 -> 0(3) -> 2(4)
2 -> 1(4) -> 3(4) -> 4(7)
3 -> 0(1) -> 2(4) -> 4(8)
4 -> 3(8) -> 2(7)
5
```

Liste di adiacenza - variazioni sul tema

Sia il grafo orientato che il grafo non orientato possono essere memorizzati tramite liste di adiacenza in diversi modi che variano a seconda del linguaggio di programmazione, di seguito alcuni esempi:

Struttura	Java	Python	C++
Lista collegata	LinkedList	list	
Vettore statico	<code>[]</code>	list	<code>[]</code>
Vettore dinamico	ArrayList	list	vector
Insieme	HashSetTreeSet	set	set
Dizionario	HashMapTreeMap	dict	map

5.2 Visite dei grafi

Il problema Dato un grafo $G = (V, E)$ e un vertice $r \in V$ (**radice, sorgente**) visitare una volta e una sola volta tutti i nodi connessi a r .

Visita in ampiezza *Breath First Search* (BFS) Questo genere di visita dei nodi viene eseguita "per livelli" e si visita prima la radice e poi i nodi a distanza 1 dalla radice, poi i nodi a distanza 2 e così via, viene usata calcolare cammini più brevi da una singola sorgente.

Visita in profondità *Depth First Search* (DFS) Questo genere di visita dei nodi viene eseguita "in profondità" e si visita la radice e poi si scende il più possibile in profondità prima di risalire, viene usata per ordinamento topologico, analisi di componenti connesse e componenti fortemente connesse.

Problemi In entrambi i casi si deve tener conto del fatto che un grafo può essere ciclico e quindi si deve evitare di visitare più volte lo stesso nodo e di entrare in loop infiniti.

Algoritmo generico di attraversamento

Algorithm 30 graphTrasversal(GRAPH G , NODE r)

```

 $S \leftarrow \text{Set}()$ 
 $S.\text{insert}(r)$ 
{ marca il nodo  $r$  }
while  $S.\text{size}() > 0$  do
    NODE  $u \leftarrow S.\text{remove}()$  { visita il nodo  $u$  }
    for NODE  $v \in G.\text{adj}(u)$  do
        if  $v \notin S$  then { marca il nodo  $v$  }
             $S.\text{insert}(v)$ 

```

5.2.1 Visita in ampiezza BFS

Obbiettivi BFS Gli obbiettivi per la ricerca BFS sono: Visitare prima tutti i nodi a distanza k poi $k + 1$ e così via, calcolare il cammino più breve da r a tutti gli altri nodi misurando la lunghezza degli archi attraversati, generare un albero **breadth-first** con radice r , quindi contenere tutti i nodi raggiungibili da r tale per cui un cammino dalla radice r al nodo u al con il cammino più breve.

Algoritmo generico di visita in ampiezza

Algorithm 31 BFS(GRAPH G , NODE r)

```

Queue  $Q \leftarrow \text{Queue}()$ 
 $Q.\text{enqueue}(r)$ 
boolean []  $visited \leftarrow \text{new boolean } [G.\text{size}()]$ 
for  $u \in G.V() - \{r\}$  do
     $visited[u] \leftarrow \text{false}$ 
 $visited[r] \leftarrow \text{true}$ 
while not  $Q.\text{isEmpty}()$  do
    NODE  $u \leftarrow Q.\text{dequeue}()$ 
    { visita il nodo  $u$  }
    for  $v \in G.\text{adj}(u)$  do
        { visita l'arco  $(u, v)$  }
        if not then  $visited[v]$ 
             $visited[v] \leftarrow \text{true}$ 
             $Q.\text{enqueue}(v)$ 

```

Calcolo minima distanza

Il principale problema risolto è quello del calcolo della minima distanza tra due nodi di un grafo, per farlo sfruttiamo la BFS e una coda. Il risultato di ciò è il seguente algoritmo:

Algorithm 32 distance(GRAPH G , NODE r , int $[]$ $distance$)

```

Queue  $Q \leftarrow \text{Queue}()$ 
 $Q.\text{enqueue}(r)$ 
foreach  $u \in G.V() - \{r\}$  do
     $distance[u] \leftarrow \infty$ 
 $distance[r] \leftarrow 0$ 
while not  $Q.\text{isEmpty}()$  do
    NODE  $u \leftarrow Q.\text{dequeue}()$ 
    for  $v \in G.\text{adj}(u)$  do
        if  $distance[v] = \infty$  then ▷ Se  $v$  non è stato scoperto ancora
             $distance[v] \leftarrow distance[u] + 1$ 
             $Q.\text{enqueue}(v)$ 

```

Albero BFS

L'albero BFS è un albero radicato con radice r utile per ottenere il cammino più breve tra r e tutti gli altri nodi del grafo. Questo genere di albero è solitamente memorizzato in un vettore di padri detto *parent*. Come

Algorithm 33 distance(..., int $[]$ $parent$)

```

[...]
 $parent[r] \leftarrow \text{nil}$ 
while not  $Q.\text{isEmpty}()$  do
    NODE  $u \leftarrow Q.\text{dequeue}()$ 
    for  $v \in G.\text{adj}(u)$  do
        if  $distance[v] = \infty$  then
             $distance[v] \leftarrow distance[u] + 1$ 
             $parent[v] \leftarrow u$ 
             $Q.\text{enqueue}(v)$ 

```

algoritmo ausiliario per la "stampa" di questo albero si può usare il seguente:

Algorithm 34 printPath(NODE r , NODE s , NODE $[]$ $parent$)

```

if  $s == r$  then
    print  $s$ 
else if  $parent[s] == \text{nil}$  then
    print "Error"
else
    PRINTPATH( $r, parent[s], parent$ )
    print  $s$ 

```

Complessità BFS

La complessità dell'algoritmo di visita in ampiezza è $O(n + m)$, dove n è il numero di nodi inserito nella coda, e ciò avviene una sola volta, inoltre in quanto un nodo viene estratto tutti i suoi archi vengono visitati una sola volta. Dunque il numero di archi analizzati è:

$$m = \sum_{u \in V} d_{out}(u)$$

dove $d_{out}(u)$ è il grado uscente del nodo u . (In un grafo non orientato coincide con il grado del nodo).

5.2.2 Visita in profondità - DFS

La visita in profondità viene usata per la risoluzione di diversi problemi, a differenza della **BFS** la **DFS** considera tutti i nodi del grafo anche quelli non connessi ad un singolo nodo. Come risultato abbiamo non un albero ma una foresta *depth-first* $G_f = (V, E_f)$ formata da un insieme di alberi *depth-first*. Solitamente per memorizzare questo viene usato o uno *Stack* implicito (attraverso la ricorsione) o uno *Stack* esplicito.

Algoritmo stack implicito Usando uno stack implicito otteniamo il seguente algoritmo: Anche in questo

Algorithm 35 dfs(GRAPH G , NODE u , boolean \square *visited*)

```

visited[ $r$ ]  $\leftarrow$  true
{ visita il nodo  $r$  (pre-order) }
foreach  $v \in G.\text{adj}(u)$  do
  if not visited[ $v$ ]
    { visita l'arco  $(u, v)$  }
    DFS( $G, v, \text{visited}$ )
{ visita il nodo  $r$  (post-order) }
```

caso la complessità per la visita di tutti i nodi è $O(n + m)$.

Algoritmo stack esplicito Usando uno stack esplicito otteniamo il seguente algoritmo:

Algorithm 36 dfs(GRAPH G , NODE r)

```

Stack  $S \leftarrow$  Stack()
 $S.\text{push}(r)$ 
boolean  $\square$  visited  $\leftarrow$  new boolean [ $G.\text{size}()$ ]
for  $u \in G.V() - \{r\}$  do
  visited[ $u$ ]  $\leftarrow$  false
while not  $S.\text{isEmpty}()$  do
  NODE  $u \leftarrow S.\text{pop}()$ 
  if not visited[ $u$ ]
    { visita il nodo  $u$  (pre-order) }
    visited[ $u$ ]  $\leftarrow$  true
    for  $v \in G.\text{adj}(u)$  do { visita l'arco  $(u, v)$  }
       $S.\text{push}(v)$ 
```

In questo algoritmo il nodo può essere inserito nella pila più volte, il controllo viene fatto all'estrazione e non all'inserimento, la complessità anche in questo caso è $O(n + m)$ in quanto somma di $O(m)$ visite agli archi, $O(m)$ inserimenti e estrazioni e $O(n)$ visite ai nodi.

Tuttavia la visita in *post-order* è più complessa da implementare con uno stack esplicito, in quanto bisogna introdurre un "tag" per identificare se il nodo è in stato "discovery" ovvero se è stato scoperto ma non visitato o se è in stato "finish" ovvero quando è stato estratto e poi re-inserito, solo quando ha questo stato allora i suoi vicini vengono inseriti nello stack. Quando viene infine estratto un nodo con il tag "finish" allora si può procedere con la visita del nodo.

Analisi di componenti (fortemente) connesse

La visita in profondità è utile per l'analisi di componenti connesse e fortemente connesse di un grafo. Prima di affrontare l'argomento è necessario introdurre diverse definizioni:

Definizione 5.11 (Componente connessa). Una componente connessa in un grafo non orientato è un sottoinsieme di nodi C tale che per ogni coppia di nodi $u, v \in C$ esiste un cammino da u a v . Quindi un grafo G' sotto-grafo di G deve essere un sotto-grafo connesso e massimale di G .

Definizione 5.12 (Componente fortemente connessa). Una componente connessa in un grafo orientato è un sottoinsieme di nodi C tale che per ogni coppia di nodi $u, v \in C$ esiste un cammino da u a v e da v a u .

Definizione 5.13 (Raggiungibilità). In un grafo non orientato si dice che un nodo v è **raggiungibile** da un nodo u se esiste un cammino da u a v , ne segue che la relazione di raggiungibilità è simmetrica e dunque se v è raggiungibile da u allora u è raggiungibile da v . In un grafo orientato si dice che un nodo v è **raggiungibile** da un nodo u se esiste un cammino da u a v e non necessariamente da v a u .

Definizione 5.14 (sotto-grafo). Un grafo $G' = (V', E')$ è un sotto-grafo di un grafo $G = (V, E)$ se $V' \subseteq V$ e $E' \subseteq E$.

Definizione 5.15 (Massimalità di un sotto-grafo). Un sotto-grafo $G' = (V', E')$ di un grafo $G = (V, E)$ è detto massimale $\Leftrightarrow \nexists$ un sotto-grafo $G'' = (V'', E'')$ di G tale che G'' sia connesso ed "più grande di G' " (ovvero $G' \subseteq G'' \subseteq G$).

Problema L'obiettivo è quello di verificare se un grafo è connesso o meno, e se non lo è trovare le componenti connesse o fortemente connesse.

Soluzione Usando la DFS analizziamo se tutti i nodi sono marcati quando "non possiamo andare più avanti" e se non lo sono allora siamo in presenza di un grafo sconnesso e quelli marcati fino ad ora costituiscono una componente connessa. Usiamo per il problema un vettore id contenete l'identificativo delle componenti con $id[u]$ uguale all'identificativo della componente connessa a cui appartiene il nodo u .

Algorithm 37 `int [] cc(GRAPH G)`

```

int []  $id \leftarrow$  new int [ $G.size()$ ]
foreach  $u \in G.V()$  do
     $id[u] \leftarrow -1$ 
int  $count \leftarrow 0$ 
foreach  $u \in G.V()$  do
    if  $id[u] == 0$  then
         $count \leftarrow count + 1$ 
         $CCDFS(G, count, u, id)$ 
return  $id$ 

```

Come funzione ricorsiva di supporto si ha:

Algorithm 38 `ccdfs(GRAPH G, int count, NODE u, int [] id)`

```

 $id[u] \leftarrow count$ 
foreach  $v \in G.adj(u)$  do
    if  $id[v] == 0$  then
         $CCDFS(G, count, v, id)$ 

```

Analisi di presenza o meno di cicli

Definiamo in primo luogo un ciclo:

Definizione 5.16 (Ciclo per un grafo non orientato). In un grafo non orientato $G = (V, E)$, un ciclo C di lunghezza $k > 2$ è una sequenza di nodi u_0, u_1, \dots, u_k tale che $(u_i, u_{i+1}) \in E$ per $i = 0, \dots, k-1$ e inoltre $u_k = u_0$.

Definiamo quindi un grafo aciclico un grafo che non contiene cicli.

Problema L'obiettivo è quello di verificare se un grafo è aciclico (**true**) o meno (**false**).

Algorithm 39 `boolean hasCycleRec(GRAPH G , NODE u , NODE p , boolean \square $visited$)`

```

 $visited[u] \leftarrow \text{true}$ 
foreach  $v \in G.\text{adj}(u) - \{p\}$  do
    if  $visited[v]$  then
        return true
    else if HASCYCLEREC( $G, v, u, visited$ ) then
        return true
return false

```

Questa è la funzione ricorsiva di supporto che tiene in considerazione eventuali componenti sconnesse, per la funzione principale si ha:

Algorithm 40 `boolean hasCycle(GRAPH G)`

```

boolean  $\square$   $visited \leftarrow \text{new } \text{boolean} [G.\text{size}()]$ 
foreach  $u \in G.V()$  do
     $visited[u] \leftarrow \text{false}$ 
foreach  $u \in G.V()$  do
    if not  $visited[u]$  then
        if HASCYCLEREC( $G, u, \text{nil}, visited$ ) then
            return true
return false

```

Grafi orientati La definizione di ciclo per un grafo orientato è leggermente diversa:

Definizione 5.17 (Ciclo per un grafo orientato). In un grafo orientato $G = (V, E)$, un ciclo C di lunghezza $k \geq 2$ è una sequenza di nodi u_0, u_1, \dots, u_k tale che $(u_i, u_{i+1}) \in E$ per $i = 0, \dots, k-1$ e inoltre $u_k = u_0$.

Inoltre come per i grafi non orientati un grafo aciclico è un grafo che non contiene cicli. Definiamo anche un **DAG** un grafo orientato aciclico (*Directed Acyclic Graph*).

Problema L'obiettivo è quello di verificare se un grafo è aciclico (**true**) o meno (**false**).

Il suddetto problema non può essere risolto con l'algoritmo precedente, questo in quanto non basta rimuovere la condizione sul nodo di provenienza p , ma bisogna considerare solo la direzione degli archi, si veda la sotto-sotto-sezione "Analisi presenza o meno di cicli orientati" presente dopo la successiva sotto-sotto-sezione

Classificazione degli archi

Prima di poter parlare di Classificazione degli archi è necessario introdurre la definizione di albero di copertura DFS:

Definizione 5.18 (Albero di copertura DFS). Dato un grafo $G = (V, E)$ allora esiste un albero di copertura DFS $G_f = (V_f, E_f)$ tale che $V_f = V$ e E_f è un sottoinsieme di E tale che per ogni nodo $u \in V$ esiste un cammino da r a u in G_f .

L'algoritmo di visita in profondità permette di classificare gli archi (u, v) non presi in considerazione dall'albero di copertura DFS in tre categorie:

- Se (u, v) è un arco tale che u è im "antenato" di v in G_f allora (u, v) è un **arco in avanti**
- Se (u, v) è un arco tale che u è un "discendente" di v in G_f allora (u, v) è un **arco all'indietro**
- In ogni altro caso (u, v) è un **arco di attraversamento**

Algorithm 41 dfs-schema($\text{GRAPH } G, \text{NODE } u, \text{int } \&time, \text{int } [] dt, \text{int } [] ft$)

```

{Visita il nodo  $u$  (pre-order)}
 $time \leftarrow time + 1$ 
foreach  $do v \in G.\text{adj}(u)$ 
    { visita l'arco  $(u, v)$  (qualsiasi) }
    if  $dt[v] == 0$  then ▷ Il nodo non è stato ancora visitato
        {Visita l'arco  $(u, v)$  albero }
        DFS-SCHEMA( $G, v, time, dt, ft$ )
    else if  $dt[u] > dt[v] \ \& \ ft[v] == 0$  then ▷ Nodo adiacente già visitato in precedenza e non ancora finito
        {Visita l'arco  $(u, v)$  all'indietro }
    else if  $dt[u] < dt[v] \ \& \ ft[v] \neq 0$  then ▷ Nodo adiacente già visitato e finito
        {Visita l'arco  $(u, v)$  in avanti }
    else ▷ Tutti gli altri casi
        {Visita l'arco  $(u, v)$  di attraversamento }
{Visito il nodo  $u$  (post-order)}
 $time \leftarrow time + 1$ 
 $ft[u] = time$ 

```

Si può ora definire l'algoritmo di classificazione degli archi. Usiamo all'interno dell'algoritmo le seguenti variabili $time$ è il contatore del tempo passato, dt è il vettore contenente i tempi di *discovery time* per ogni nodo del grafo e ft è il vettore contenente i tempi di *finish* di ogni vettore. Si noti come alla fine dell'algoritmo otteniamo due vettori dt e ft popolati, su questi "tempi" di *discovery* può essere formulato un teorema:

Teorema 5.2. Data una visita DFS di un grafo $G = (V, E)$, per ogni coppia di nodi $u, v \in V$, solo una delle seguenti condizioni è vera:

- Gli intervalli $[dt[u], ft[u]]$ e $[dt[v], ft[v]]$ sono non-sovrapposti, allora u, v **non sono discendenti l'uno dell'altro** nella **foresta DF**
- L'intervallo $[dt[u], ft[u]]$ è contenuto nell'intervallo $[dt[v], ft[v]]$, allora u è **un discendente di** v in un albero DF
- L'intervallo $[dt[v], ft[v]]$ è contenuto nell'intervallo $[dt[u], ft[u]]$, allora u è **un antenato di** v in un albero DF

Analisi presenza o meno di cicli orientati

Grazie alle conoscenze apprese dalla sotto-sotto-sezione precedente possiamo ora formulare un teorema riguardante la presenza o meno di cicli orientati:

Teorema 5.3 (Graf orientati aciclici). Un grafo orientato è aciclico se e solo se non esistono archi all'indietro nel grafo.

Dimostrazione. Procediamo per entrambe le direzioni:

- **se:** Esiste un ciclo, allora sia u il primo nodo di questo e sia (v, u) un arco del ciclo. Allora il cammino che connette u ad v sarà visitato, ed eletto come cammino dell'albero di copertura, prima o poi. Quando si raggiungerà il nodo v si "scopre" l'esistenza di (v, u) ovvero un arco all'indietro.
- **solo se:** Se esiste un arco all'indietro (u, v) , dove v è un antenato di u , allora esiste un cammino da v a u e un arco da u a v ovvero un ciclo.

□

Applichiamo dunque il teorema appena dimostrato tramite il seguente algoritmo che sfrutta gli algoritmi "hasCycleRec" e "dfs-schema" precedentemente definiti.¹

Algorithm 42 `boolean hasCycleRec(GRAPH G , NODE u , int &time, int [] dt, int ft)`

```

time ← time + 1
dt[u] ← time
foreach  $v \in G.\text{adj}(u)$  do
  if dt[v] == 0 then
    if HASCYCLEREC( $G, v, \text{time}, dt, ft$ ) then
      return true
  else if ft[v] == 0 then
    return true
time ← time + 1
ft[u] ← time
return false

```

Ordinamento topologico

La DFS può essere usata anche per stabilire un **ordinamento topologico** di un grafo DAG. Un ordinamento topologico è definito come segue:

Definizione 5.19 (Ordinamento topologico). Dato un DAG G un **ordinamento topologico** di G è un ordinamento lineare dei suoi nodi tale che se $u, v \in E$ allora u precede v nell'ordinamento.

Da questa definizione possiamo dedurre che per un DAG possano esistere più ordinamenti, inoltre se consideriamo un grafo qualunque nel quale sono presenti cicli non può esistere un ordinamento.

Problema L'obiettivo è quello prendere in *input* un DAG e restituire un ordinamento topologico.

Soluzione "Naive" Si prende un nodo senza archi entranti (che esiste sempre) si aggiunge questo nodo all'ordinamento e si "elimina" il nodo dal grafo, si ripete il procedimento fino a quando non si è eliminato tutti i nodi.

Algoritmo Eseguiamo una DFS con l'operazione di visita, in *post-order*, che consiste nell'aggiungere il nodo in testa ad una lista, si restituisce la lista ottenuta (ribaltata). Questo funziona in quanto quando un nodo è "finito" allora tutti i suoi discendenti sono stati scoperti e aggiunti.

Algorithm 43 `STACK topSort(GRAPH g)`

```

STACK  $S \leftarrow \text{Stack}()$ 
boolean []visited ← boolean ( $G.\text{size}()$ , false )
foreach  $u \in G.V()$  do
  if not visited[u] then
    TS-DFS( $G, u, \text{visited}, S$ )
return  $S$ 

```

¹Va modificato l'algoritmo "hasCycle" inizializzando dt e ft a vettori di 0 e $time$ a 0. Per il resto l'algoritmo è identico, omissis quindi brevità.

Algorithm 44 ts-dfs(GRAPH G , NODE u , **boolean** $[]$ $visited$, STACK S)

```
 $visited[u] \leftarrow \mathbf{true}$   
foreach  $v \in G.\text{adj}(u)$  do  
    if not  $visited[v]$  then  
        TS-DFS( $G, v, visited, S$ )  
 $S.\text{push}(u)$ 
```

Grazie al teorema precedentemente dimostrato possiamo affermare che l'algoritmo funziona in quanto se un nodo è "finito" allora tutti i suoi discendenti sono stati scoperti e aggiunti.

Applicazioni in the real world L'ordinamento topologico è utilizzato in diversi campi, tra cui: L'ordine di valutazione delle celle in uno *spreadsheet*, l'ordine di compilazione di un **Makefile**, la risoluzione di dipendenze in un **package manager** e la risoluzione di dipendenze in un **task scheduler**.

Componenti fortemente connesse

Capitolo 6

Hashing

6.1 Introduzione

6.1.1 Preambolo

Prima di iniziare a parlare di hashing, è necessario definire cos'è un dizionario

Definizione 6.1 (Dizionario). Un dizionario è una struttura dati utilizzata per memorizzare un insieme dinamico di elementi, costituiti da **coppie** $\langle k, v \rangle$, dove k è la chiave e v è il valore associato alla chiave. Le coppie sono uniche rispetto alla chiave, il valore è un **"dato satellite"** associato alla chiave.

Le operazioni di un dizionario sono:

- $\text{lookup}(key) \rightarrow value$: inserisce la coppia $\langle k, v \rangle$ nel dizionario
- $\text{insert}(key, value)$: cerca la chiave k nel dizionario e restituisce il valore associato
- $\text{delete}(key)$: elimina la coppia $\langle k, v \rangle$ dal dizionario

Le strutture dati che implementano un dizionario sono ad esempio: Le tabelle di simboli, i dizionari di Python, le mappe di Java, gli oggetti di JavaScript, ecc. . .

Implementazione ideale L'implementazione ideale di un dizionario, ovvero quella che garantisce le complessità di $O(1)$ per $\text{insert}, \text{lookup}, \text{remove}$ è la tabella hash.

6.1.2 Tabelle Hash

Definizioni

Una **funzione hash** è definita come: $h : \mathcal{U} \rightarrow \{0, 1, \dots, m - 1\}$, dove \mathcal{U} è l'universo delle chiavi e m è la dimensione della tabella hash.

La coppia $\langle k, v \rangle$ è memorizzata nella cella $h(k)$ della tabella hash.

	Array non ordinato	Array ordinato	Lista	Albero RB	Impl. Ideale Hash
$\text{insert}()$	$O(1), O(n)$	$O(n)$	$O(1), O(n)$	$O(\log n)$	$O(1)$
$\text{lookup}()$	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$	$O(1)$
$\text{remove}()$	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$
foreach	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$ $O(m)$

Collisioni

Quando due chiavi k_1 e k_2 hanno lo stesso valore di hash, ovvero $h(k_1) = h(k_2)$, si verifica una collisione. Idealmente si vorrebbe che la funzione hash sia senza collisioni.

6.2 Funzioni Hash

6.2.1 Introduzione

Definizione 6.2 (Funzione Hash Perfetta). Una funzione hash h si dice **perfetta** se è **iniettiva**, ovvero se non ci sono collisioni.

$$\forall k_1, k_2 \in \mathcal{U} \quad k_1 \neq k_2 \Rightarrow h(k_1) \neq h(k_2)$$

Ottenere una funzione hash perfetta è pressoché impossibile, in quanto spesso la chiave ha una dimensione maggiore della tabella hash e/o un dominio molto grande.

Per ovviare a questo cerchiamo di minimizzare il numero di collisioni, questo cercando di distribuire **uniformemente** le chiavi negli indici $[0, m - 1]$ della tabella hash.

Definizione 6.3 (Uniformità Semplice). Sia $P(k)$ la probabilità che una chiave k venga inserita in tabella, Sia $Q(i)$ la probabilità che una chiave venga inserita nella cella i dove:

$$Q(i) = \sum_{k \in \mathcal{U}: h(k)=i} P(k)$$

Allora una funzione hash h gode di **uniformità semplice** se:

$$\forall i \in [0, \dots, m - 1] \quad Q(i) = \frac{1}{m}$$

6.2.2 Funzioni Hash Semplici

Prima di definire delle funzioni di hash, dobbiamo prima assumere che le chiavi possano essere tradotte in valori numerici, anche interpretando la loro rappresentazione in memoria come un numero.

Esempio 6.1 (Trasformazione di stringhe). Di seguito sono riportate alcune funzioni di trasformazione di stringhe in numeri:

- $\text{ord}(c)$: valore ordinale binario del carattere c in qualche codifica
- $\text{bin}(k)$: rappresentazione binaria della chiave k concatenando i valori ordinari dei caratteri della chiave
- $\text{int}(b)$: valore numerico associato al numero binario b
- $\text{int}(k) = \text{int}(\text{bin}(k))$

Funzioni hash semplici Alcuni esempi di funzioni hash semplici sono:

- Estrazione $\Rightarrow m = 2^p$: $h(k) = \text{int}(b)$ dove b è un sottoinsieme di p bit di $\text{bin}(k)$
Molto veloce, ma con molte collisioni
- XOR $\Rightarrow m = 2^p$: $h(k) = \text{int}(b)$ dove b è dato dalla somma modulo 2, effettuata bit a bit, di sottoinsiemi di p bit di $\text{bin}(k)$
Permutazioni della stessa chiave danno lo stesso risultato
- Metodo della Divisione $\Rightarrow m$ dispari meglio se primo, $h(k) = \text{int}(k) \bmod m$.
Non va bene $m = 2^p$: solo i p bit meno significativi vengono considerati
Non va bene $m = 2^p - 1$: permutazione di stringhe con set di caratteri uguali hanno lo stesso valore hash
Vanno bene: Numeri primi, distanti da potenze di 2 (e di 10)

- Metodo della Moltiplicazione $\Rightarrow m$ qualsiasi meglio se potenza di 2, C costante $0 < C < 1$. Sia $i = \text{int}(k)$ e $H(k) = \lfloor m \cdot (C \cdot i - \lfloor C \cdot i \rfloor) \rfloor$

C è una costante che si sceglie in base a m

C è un numero irrazionale, quindi non si ha un pattern di collisioni

Implementazione della moltiplicazione Si sceglie un valore $m = 2^p$, sia w la dimensione in bit della parola di memoria: $i, m \leq 2^w$. Sia $s = \lfloor C \cdot 2^w \rfloor$, allora: $i \cdot s$ può essere scritto come $r_1 \cdot 2^w + r_0$ dove r_1 contiene la parte intera di $i \cdot C$ e r_0 la parte frazionaria. Si restituiscono i p -bit più significativi di r_0 come valore hash.

6.3 Gestione delle collisioni

6.3.1 Liste/Vettori di trabocco

Idea Generale

L'idea di usare una lista di trabocco è quella di memorizzare in una cella della tabella hash non solo il valore corrispondente alla chiave, ma anche una lista di coppie $\langle k, v \rangle$ le quali chiavi hanno lo stesso valore di hash. Quando si vuole aggiungere una coppia $\langle k, v \rangle$ e k ha lo stesso valore di hash di una chiave già presente, si "fa puntare" la testa della lista di trabocco alla nuova coppia appena inserita e la nuova coppia "punta" alla vecchia testa.

Operazioni

- **insert**: si inserisce la coppia $\langle k, v \rangle$ in testa alla lista di trabocco
- **lookup**: si scorre la lista di trabocco fino a trovare la chiave k e si restituisce il valore associato
- **delete**: si scorre la lista di trabocco fino a trovare la chiave k e si elimina la coppia $\langle k, v \rangle$

Analisi complessità

In quanto si hanno n chiavi memorizzate nella tabella hash e m è la capacità della tabella, si ha che il fattore di carico $\alpha = \frac{n}{m}$. Inoltre definendo $I(\alpha)$ come il numero medio di accessi alla tabella per la ricerca di una chiave con insuccesso e $S(\alpha)$ come il numero medio di accessi alla tabella per la ricerca di una chiave con successo, si ha che:

Caso peggiore:

- $\text{insert}() \rightarrow \Theta(1)$
- $\text{lookup}(), \text{delete}() \rightarrow \Theta(n)$

Caso medio: Dipende da come le chiavi sono distribuite nella tabella hash, assumendo un hash uniforme semplice e costo di hashing $O(1)$: Allora il valore atteso della lunghezza di una lista pari a $\alpha = \frac{n}{m}$. Per la ricerca distinguiamo due casi:

- **Successo**, costo atteso: $\Theta(1) + \alpha$
- **Insuccesso**, costo atteso: $\Theta(1) + \frac{\alpha}{2}$

Nel caso medio si può osservare come il costo computazionale sia influenzato dal fattore di carico α , nel caso ottimale se fossimo in una situazione nella quale $n = O(m)$ e $\alpha = O(1)$, allora per ogni operazione il costo sarebbe $O(1)$.

6.3.2 Indirizzamento aperto

Idea

La tecnica dell'Indirizzamento aperto punta a risolvere la complessità, a livello strutturale, delle liste di trabocco. L'idea è quella di memorizzare le chiavi nella tabella hash stessa, ed ogni slot può contenere o una chiave o **nil** (indicante che la cella è vuota).

Inserimento Quando si vuole inserire una chiave k e la cella $h(k)$ è occupata, si cerca la prima cella vuota a partire da $h(k)$.

Operazioni

Ricerca Quando si vuole cercare una chiave k si parte da $h(k)$ e si scorre la tabella fino a trovare la chiave k o una cella vuota.

Funzione Hash La funzione hash deve essere estesa, infatti questa in caso di Indirizzamento aperto assumerà la forma:

$$H : \mathcal{U} \times \overbrace{\{0, 1, \dots, m-1\}}^{\text{Numero ispezione}} \rightarrow \overbrace{\{0, 1, \dots, m-1\}}^{\text{Indice vettore}}$$

Ispezione

Definiamo il termine **ispezione**

Definizione 6.4 (Ispezione). L'ispezione è l'esame di uno slot durante la ricerca.

Definiamo inoltre il termine **sequenza di ispezione**

Definizione 6.5 (Sequenza di Ispezione). Una sequenza di ispezione $[H(k, 0), H(k, 1), \dots, H(k, m-1)]$ è una sequenza di **permutazione** degli indici $[0, 1, \dots, m-1]$ corrispondente all'ordine in cui vengono esaminati gli slot.

Questa rispetta le seguenti regole: Non ci sono ripetizioni di uno slot già visitato e, nel caso la tabella sia piena, potrebbe essere necessario visitare tutti gli slot.

Tecniche di ispezione La situazione ideale prende il nome di *hashing uniforme* ovvero ogni chiave ha la stessa probabilità di avere come sequenza di ispezione una qualsiasi delle $m!$ permutazioni di $[0, 1, \dots, m-1]$. In quanto ciò è impossibile, si ricorre a tecniche di ispezione:

Ispezione Lineare La sequenza di ispezione è data da $H(k, i) = (H_1(k) + h \cdot i) \bmod m$ la quale genera le varie sequenze: $[H_1(k), H_1(k) + h, H_1(k) + 2h, \dots]$ che determina il primo elemento. Al massimo sono presenti m sequenze di ispezione. Incontriamo ora il problema del *clustering*, ovvero la formazione di gruppi di chiavi che si trovano vicine tra loro.

Definizione 6.6 (Agglomerazione Primaria (*primary clustering*)). L'agglomerazione primaria è la formazione di lunghe sotto-sequenze occupate che tendono a crescere in quanto uno slot vuoto preceduto da uno slot occupato ha probabilità $\frac{(i+1)}{m}$ di essere occupato.

Conseguenza diretta del clustering è l'incremento dei tempi medi di inserimento e cancellazione.

Ispezione Quadratica La sequenza di ispezione è data da $H(k, i) = (H_1(k) + h \cdot i^2) \bmod m$ con questa tecnica dopo il primo elemento ($H_1(k, 0)$) le ispezioni successive saranno distanti tra loro di $1, 4, 9, \dots$ slot. Come risultato non si ha però una **permutazione** in quanto alcuni slot potrebbero non essere mai visitati, anche in questo caso abbiamo al massimo m sequenze di ispezione distinte. Elimiamo di fatto l'agglomerazione primaria, ma introduciamo l'**agglomerazione secondaria**.

Definizione 6.7 (Agglomerazione Secondaria (*secondary clustering*)). Se due chiavi hanno la stessa ispezione iniziale allora le loro sequenze sono identiche.

Doppio Hashing La sequenza di ispezione è data da $H(k, i) = (H_1(k) + i \cdot H_2(k)) \bmod m$ con $H_2(k)$ che fornisce l'*offset* delle successive ispezioni. In questo caso si hanno m^2 sequenze di ispezione distinte, inoltre se si vuole garantire una permutazione completa allora $H_2(k)$ deve essere co-primo con m .

Cancellazione

Come probabilmente si è già intuito non si possono sostituire le chiavi con **nil**, in quanto si potrebbe interrompere la sequenza di ispezione. Per ovviare a questo problema si introduce il concetto di **tombstone**, ovvero un valore speciale, **deleted**, usato al posto di **nil** dopo la cancellazione di una chiave. Questo permette di mantenere la sequenza di ispezione intatta.

Durante la ricerca se ci si imbatte in un *tombstone* si continua la ricerca, in quanto la chiave potrebbe essere presente in una cella successiva, mentre in caso di inserimento si sovrascrive il *tombstone* con la nuova chiave. Il tempo di ricerca ora non dipende più da α , inoltre se si rende possibile la cancellazione il concatenamento diventa più comune.

Implementazione - Hashing doppio

Di seguito una possibile implementazione dell'Indirizzamento aperto con doppio hashing:

Algorithm 45 HASH

ITEM \square K	▷ Tabella delle chiavi
ITEM \square V	▷ Tabella dei valori
int m	▷ Dimensione della tabella
HASH	
function HASH(int dim)	
HASH $t \leftarrow$ new HASH	
$t.m \leftarrow dim$	
$t.K \leftarrow$ new ITEM [$dim - 1$]	
$t.V \leftarrow$ new ITEM [$dim - 1$]	
for $i \leftarrow 0$ to $dim - 1$ do	
$t.K[i] \leftarrow$ nil	

Di seguito l'algoritmo di ricerca, senza che venga considerata la cancellazione:

Algorithm 46 SEARCH

```

int
function SCAN(ITEM  $k$ )
    int  $i \leftarrow 0$ 
    int  $j \leftarrow H_1(k)$ 
    while  $keys[j] \neq k$  &  $keys[j] \neq$  nil &  $i < m$  do
         $j \leftarrow (j + H_2(k)) \bmod m$ 
         $i \leftarrow i + 1$ 
    return  $j$ 

```

Se si vuole implementare la cancellazione, si deve aggiungere un controllo per i *tombstone*, ovvero se si incontra un *tombstone* si continua la ricerca tranne nel caso in cui si stia inserendo una nuova chiave.

Algorithm 47 SEARCH

```

int
function SCAN(ITEM  $k$ , boolean  $insert$ )
    int  $delpos = m$ 
    int  $i \leftarrow 0$ 
    int  $j \leftarrow H_1(k)$ 
    while  $keys[j] \neq k$  &  $keys[j] \neq \text{nil}$  &  $i < m$  do
        if  $keys[j] = \text{DELETED}$  &  $delpos = m$  then
             $delpos \leftarrow j$ 
         $j \leftarrow (j + H_2(k)) \bmod m$ 
         $i \leftarrow i + 1$ 
    if  $insert$  &  $keys[j] \neq k$  &  $delpos < m$  then
        return  $delpos$ 
    else if
        then return  $j$ 

```

Si riporta ora il codice delle API per il *lookup*, l'*insert* e il *remove*:

Algorithm 48 API

```

ITEM
function LOOKUP(ITEM  $k$ )
     $j \leftarrow \text{SCAN}(k, \text{false})$ 
    if  $keys[j] == k$  then
        return  $values[j]$ 
    else
        return nil

function INSERT(ITEM  $k$ , ITEM  $v$ )
     $j \leftarrow \text{SCAN}(k, \text{true})$ 
    if  $keys[j] == \text{nil}$  or  $keys[j] == \text{DELETED}$  or  $keys[j] == k$  then
         $keys[j] \leftarrow k$ 
         $values[j] \leftarrow v$ 
    else if
        then
            ▷ Errore, tabella hash piena

function REMOVE(ITEM  $k$ )
     $j \leftarrow \text{SCAN}(k, \text{false})$ 
    if  $keys[j] == k$  then
         $keys[j] \leftarrow \text{DELETED}$ 
         $values[j] \leftarrow \text{nil}$ 

```

Fattore di carico Il fattore di carico α è sempre compreso tra 0 e 1, ma la tabella può andare in overflow, ovvero avere più chiavi di slot disponibili.

6.3.3 Analisi complessità

La complessità delle operazioni coi vari metodi di ispezione è la seguente:

Metodo	α	$I(\alpha)$	$S(\alpha)$
Lineare	$0 \leq \alpha < 1$	$\frac{1-\alpha^2+1}{2(1-\alpha)^2}$	$\frac{1-\frac{\alpha}{2}}{1-\alpha}$
Hashing Doppio	$0 \leq \alpha < 1$	$\frac{1}{1-\alpha}$	$-\frac{1}{\alpha} \ln(1-\alpha)$
Liste di Trabocco	$\alpha \geq 0$	$1 + \alpha$	$1 + \frac{\alpha}{2}$

Capitolo 7

Divide-et-impera

7.1 Introduzione

Classificazione dei problemi Prima di iniziare bisogna saper classificare i problemi, infatti non tutti i problemi possono essere risolti con la tecnica *divide-et-impera*. Di seguito vengono elencati i vari tipi di problemi:

Problemi decisionali Sono problemi per i quali la risposta è un semplice "sì" o "no". Ad esempio: "Esiste un cammino di lunghezza k tra due nodi u e v ?".

Problemi di ricerca Sono problemi per i quali si cerca una soluzione all'interno di uno spazio di ricerca, in questo genere di problemi le soluzioni ammissibili sono quelle che rispettano certi vincoli e si cerca la migliore tra queste. Ad esempio: "qual'è la posizione di una data sotto-stringa all'interno di una stringa?".

Problemi di ottimizzazione Sono problemi per i quali si cerca la soluzione migliore tra tutte le soluzioni ammissibili. Ad esempio: "qual'è il cammino più breve tra due nodi u e v ?".

Definizione matematica Spesso è fondamentale definire bene il problema in modo formale, questo in quanto una definizione formale permette di capire meglio il problema e di trovare una soluzione più facilmente (in alcuni casi),

Tecniche di soluzione problemi

Divide-et-impera Un problema viene suddiviso in sotto-problemi più piccoli, indipendenti risolti ricorsivamente (*top-down*). La soluzione del problema originale viene costruita a partire dalle soluzioni dei sotto-problemi. Lavora in modo efficiente nei problemi di decisione e ricerca.

Programmazione dinamica La soluzione viene costruita a partire dalle soluzioni dei sotto-problemi, ma i sotto-problemi vengono risolti una sola volta e la soluzione viene memorizzata in una tabella. Tecnica spesso usata per i problemi di ottimizzazione.

Memorization (o annotazione) Semplicemente la versione *top-down* della programmazione dinamica, in cui si memorizzano i risultati dei sotto-problemi per evitare di ricalcolarli.

Tecnica Greedy Si costruisce la soluzione in modo incrementale, scegliendo ogni volta la soluzione migliore locale. Questa tecnica non garantisce la soluzione ottima.

Backtrack Si costruisce procedendo per "tentativi" e si torna indietro quando se non si trova una soluzione. Questa tecnica è spesso usata per i problemi di decisione.

Ricerca locale Si parte da una soluzione iniziale e si cerca di migliorarla iterativamente. Questa tecnica è spesso usata per i problemi di ottimizzazione.

Algoritmi probabilistici Si basano su tecniche di campionamento casuale. Questi algoritmi non garantiscono la soluzione ottima, ma spesso sono più efficienti.

Le tre fasi di *divide-et-impera* La tecnica *divide-et-impera* si divide in tre fasi: *Divide* ovvero la divisione del problema in sotto-problemi più piccoli, *Impera* ovvero la risoluzione dei sotto-problemi in modo ricorsivo e *Combina* ovvero la combinazione delle soluzioni dei sotto-problemi per ottenere la soluzione del problema originale.

7.2 Torre di Hanoi

La torre di Hanoi è un gioco che consiste in tre pioli e n dischi di diametro diverso, i dischi sono impilati in ordine decrescente di diametro su uno dei pioli. Lo scopo del gioco è spostare tutti i dischi da un piolo di partenza ad un piolo di arrivo, rispettando le seguenti regole:

- Si può spostare un solo disco alla volta.
- Un disco può essere spostato solo se è il disco più in alto su un piolo.
- Un disco può essere spostato solo su un piolo vuoto o su un disco più grande.
- Si può usare il piolo di mezzo come piolo di appoggio.

Questo problema può essere risolto in modo ricorsivo, dividendo il problema in sotto-problemi più piccoli. La soluzione è la seguente:

Algorithm 49 hanoi(int n ,int src ,int $dest$, int $middle$)

```

if  $n = 1$  then
    print  $src \rightarrow dest$ 
else
    HANOI( $n - 1$ ,  $src$ ,  $middle$ ,  $dest$ )
    print  $src \rightarrow dest$ 
    HANOI( $n - 1$ ,  $middle$ ,  $dest$ ,  $src$ )

```

Concettualmente il presente algoritmo funziona in questo modo: sposto $n - 1$ dischi dal piolo di partenza al piolo di mezzo, sposto il disco rimanente dal piolo di partenza al piolo di arrivo e infine sposto i $n - 1$ dischi dal piolo di mezzo al piolo di arrivo.

Costo computazionale Il costo del problema è esprimibile con la seguente equazione di ricorrenza:

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(n-1) + 1 & n > 1 \end{cases}$$

dunque il costo del problema è $O(2^n)$, questo in quanto vengono effettuate due chiamate ricorsive per ogni chiamata ricorsiva, inoltre tutte le volte che eseguo una chiamata ricorsiva ho costo 1.

7.3 QuickSort

Il **QuickSort** è un algoritmo di ordinamento basato sulla tecnica *divide-et-impera*. Dove il caso medio ha costo $O(n \log n)$, il caso peggiore ha costo $O(n^2)$. Viene usato il **QuickSort** rispetto al **MergeSort** per la sua minore complessità costante ed il suo minor uso di memoria, ciò in quanto il **QuickSort** non necessita di un array ausiliario.

Input L'input dell'algoritmo è un array A di n elementi, inoltre come parametri si passano l'indice di inizio e l'indice di fine dell'array.

Divide Viene scelto un valore $p \in A[start \dots end]$ detto perno (*pivot*), vengono spostati gli elementi minori di p a sinistra e quelli maggiori a destra.

Impera Viene chiamato ricorsivamente l'algoritmo su due sotto-array: $A[start \dots p-1]$ e $A[p+1 \dots end]$, in quanto avendo spostato a destra i maggiori e a sinistra i minori, il perno è già nella posizione corretta.

Combina Non è necessario, in quanto l'array è già ordinato.

Scelta del perno Di seguito viene riportato l'algoritmo per la scelta del perno e per la divisione dell'array:

Algorithm 50 `int pivot(ITEM A[], int start, int end)`

```

ITEM pivot ← A[start]
int i ← start
for int j ← start + 1 to end do
    if A[j] < pivot then
        i ← i + 1
        SWAP(A[i], A[j])
SWAP(A[i], A[start])
return i

```

Al'interno del presente algoritmo viene usata la seguente funzione **swap**:

Algorithm 51 `swap(ITEM a, ITEM b)`

```

ITEM temp ← a
a ← b
b ← temp

```

Algoritmo principale Dopo aver definito la funzione **pivot** possiamo definire l'algoritmo principale:

Algorithm 52 `quickSort(ITEM A[], int start, int end)`

```

if start < end then
    int j ← PIVOT(A, start, end)
    QUICKSORT(A, start, j - 1)
    QUICKSORT(A, j + 1, end)

```

7.4 Algoritmo di Strassen

L'algoritmo di Strassen è un algoritmo di moltiplicazione di matrici basato sulla tecnica *divide-et-impera*. Normalmente per il calcolo di una matrice $C = A \cdot B$ si ha un costo di $O(n^3)$, dato dal seguente algoritmo:

Algorithm 53 matrixMultiplication(float $\square\square$ A, B, C , int n_i, n_j, n_k)

```

for int  $i \leftarrow 1$  to  $n_i$  do

    for int  $j \leftarrow 1$  to  $n_j$  do
         $C[i][j] \leftarrow 0$ 
        for int  $k \leftarrow 1$  to  $n_k$  do
             $C[i][j] \leftarrow C[i][j] + A[i][k] \cdot B[k][j]$ 
  
```

Approccio *divide-et-impera* Il principio di *divide-et-impera* si basa sulla divisione di una matrice $n \times n$ in quattro sotto-matrici $n/2 \times n/2$, in modo tale da ridurre il costo della moltiplicazione.

Equazione di ricorrenza L'equazione di ricorrenza dell'algoritmo di Strassen è la seguente:

$$T(n) = \begin{cases} 1 & n = 1 \\ 8T(n/2) + n^2 & n > 1 \end{cases}$$

Il che per il teorema del master ci dà un costo di $O(n^{\log_2 8}) = O(n^3)$, dunque l'algoritmo di Strassen non è più efficiente dell'algoritmo classico.

Algoritmo di Strassen L'algoritmo di Strassen prevede il calcolo di sette matrici $n/2 \times n/2$, e la ricorrenza su queste.¹ Il che porta alla seguente equazione di ricorrenza:

$$T(n) = \begin{cases} 1 & n = 1 \\ 7T(n/2) + O(n^2) & n > 1 \end{cases}$$

Portando il costo dell'algoritmo a $O(n^{\log_2 7}) \approx O(n^{2.81})$.

¹Il funzionamento dell'algoritmo di Strassen non è di grande importanza per il corso, basti sapere che questo è migliore rispetto all'approccio *divide-et-impera*

Capitolo 8

Analisi ammortizzata

La **analisi ammortizzata** è una tecnica di analisi degli algoritmi che permette di valutare il tempo richiesto per eseguire, nel caso pessimo, una sequenza di operazioni su una struttura dati. Questa tecnica tiene in considerazione operazioni più o meno costose e se le operazioni costose sono poco frequenti, il costo totale delle operazioni può essere ammortizzato su tutte le operazioni.

Metodi Gli algoritmi di analisi ammortizzata si basano su tre metodi:

- **Metodo dell'aggregazione** Si calcola la complessità $T(n)$ per eseguire n operazioni in sequenza nel caso pessimo
- **Metodo degli accantonamenti**: Alle operazioni vengono assegnati **costi ammortizzati** che possono essere maggiori o minori del costo effettivo dell'operazione.
- **Metodo del potenziale**: Lo stato del sistema viene rappresentato da una **funzione di potenziale**

8.1 Contatore Binario

Sia un contatore binario di k bit con un vettore A di booleani 0/1 con il bit meno significativo in $A[0]$ e il bit più significativo in $A[k-1]$. Il valore del contatore è dato da: $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$. L'operazione `increment()` che incrementa il contatore di 1 è la seguente:

Algorithm 54 `increment(int []A, int k)`

```
int i ← 0
while i < k & A[i] = 1 do
    A[i] ← 0
    i ← i + 1
if i < k then
    A[i] ← 1
```

8.1.1 Metodo dell'aggregazione

Con il metodo dell'aggregazione si calcola la complessità per eseguire n operazioni in sequenza nel caso pessimo (in questo caso avendo una sola operazione non distinguiamo tra caso pessimo e caso medio).

Si calcola il costo ammortizzato $\frac{T(n)}{n}$ come **media** su n operazioni.

Analisi grossolana Per n operazioni sono necessari $k = \lceil \log(n+1) \rceil$ bit. Il costo di `increment()` è $O(k)$, quindi il costo di n operazioni è $T(n) = O(nk)$ e il costo di una singola operazione è $\frac{T(n)}{n} = O(k) = O(\log n)$.

Analisi più stretta Con la analisi precedente non si tiene conto del fatto che la maggior parte delle operazioni `increment()` sono a costo $O(1)$ in quanto non si deve propagare il riporto, inoltre il tempo di esecuzione di una singola operazione è proporzionale al numero di bit che vengono cambiati, ma vengono modificati pochi bit per ogni operazione in quanto non sempre si ha un riporto e non sempre la propagazione del riporto avviene per tutti i bit. Infatti su 8 bit ad esempio il bit in $A[0]$ viene cambiato ogni operazione, il bit in $A[1]$ viene cambiato ogni 2 operazioni, il bit in $A[2]$ viene cambiato ogni 4 operazioni e così via fino al bit in $A[i]$ che viene cambiato ogni 2^i operazioni.

Il costo totale di n operazioni è quindi:

$$T(n) = \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor \leq n \sum_{i=0}^{k-1} \frac{1}{2^i} \leq n \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i = 2n$$

Quindi il costo ammortizzato di una singola operazione è $\frac{T(n)}{n} \leq \frac{2n}{n} = 2 = O(1)$.

8.1.2 Metodo degli accantonamenti

Si assegna un costo ammortizzato *potenzialmente* distinto ad ogni operazione possibile.

Il costo ammortizzato può essere diverso dal costo effettivo, le operazioni meno costose vengono caricate di un costo aggiuntivo detto "**credito**", i crediti accumulati sono usati per pagare le operazioni più costose.

Obbiettivi L'obiettivo è dimostrare che la somma dei costi ammortizzati a_i è un limite superiore ai costi effettivi c_i e che il valore così ottenuto è "poco" costoso. Dunque:

$$\sum_{i=1}^n c_i \leq \sum_{i=1}^n a_i$$

Da notare che la dimostrazione deve valere per tutte le sequenze (caso pessimo) e il credito dopo l'operazione t -esima è espresso dalla seguente formula:

$$\sum_{i=1}^t a_i - \sum_{i=1}^t c_i \geq 0$$

Denotando il costo effettivo dell'operazione `increment` : d dove d è il numero di bit che vengono cambiati, il costo ammortizzato è 2 in quanto 1 per il cambio di un bit da 0 a 1 e 1 per il cambio di un bit da 1 a 0. Allora ne consegue che:

- In ogni istante, il credito è pari al numero di bit a 1.
- Il costo totale è un $O(n)$.
- Il costo ammortizzato di una singola operazione è $O(1)$.

8.1.3 Metodo del potenziale

Come anticipato il metodo del potenziale prevede l'utilizzo di una funzione di potenziale Φ che associa ad uno stato S della struttura data la "quantità di lavoro" $\Phi(S)$ che è stato contabilizzato nell'analisi ammortizzata, ma non ancora eseguito. Con un parallelismo tratto dalla "fisica" $\Phi(S)$ rappresenta la quantità di energia potenziale che è stata accumulata nello stato S e che può essere liberata in futuro.

Costo Ammortizzato Il costo ammortizzato di un'operazione a_i è definito come:

$$a_i = c_i + \Phi(S_i) - \Phi(S_{i-1})$$

dove c_i è il costo reale dell'operazione a_i e S_i è lo stato associato alla i -esima operazione. Il costo ammortizzato di una sequenza di operazioni è definito come:

$$\begin{aligned}
 A &= \sum_{i=1}^n a_i \\
 &= \sum_{i=1}^n (c_i + \Phi(S_i) - \Phi(S_{i-1})) \\
 &= \underbrace{\sum_{i=1}^n c_i}_C + \sum_{i=1}^n (\Phi(S_i) - \Phi(S_{i-1})) \\
 &= C + \Phi(S_1) - \Phi(S_0) + \Phi(S_2) - \Phi(S_1) + \dots + \Phi(S_n) - \Phi(S_{n-1}) \\
 &= C + \Phi(S_n) - \Phi(S_0)
 \end{aligned}$$

Se la variazione di potenziale è non negativa, allora il costo ammortizzato A è un limite superiore al costo reale C .

Analisi del Contatore Binario Scegliamo come funzione potenziale Φ il numero di bit a 1 nel contatore binario. Allora nell'operazione `increment()` definiamo t come il numero di bit a 1 incontrati a partire da $A[0]$. Il costo effettivo dell'operazione è $1 + t$, la variazione di potenziale è $1 - t$ e il costo ammortizzato è $1 + t + 1 - t = 2$. In quanto di "media" il costo di un'operazione è 2 e il costo totale di n operazioni è $2n$ il costo ammortizzato di una singola operazione è $O(1)$.

8.2 Vettori Dinamici

Il problema della gestione di un vettore dinamico è uno dei problemi più comuni in informatica.

8.2.1 Inserimento

La dimensione di un vettore è fissa e se si vuole aggiungere un elemento in coda al vettore, se il vettore è pieno, bisogna allocare dello spazio aggiuntivo. In un vettore inoltre il costo di un inserimento nella posizione n è $O(n)$ in quanto bisogna spostare tutti gli elementi di una posizione, mentre il costo di un inserimento in coda è $O(1)$ e quindi prediligiamo l'inserimento in coda.

Soluzione Si può risolvere il problema allocando alla necessità un vettore di dimensione maggiore (di quanto maggiore?) e copiando gli elementi del vettore originale nel nuovo vettore. E.G. `java.util.Vector`, `java.util.ArrayList`.

Anche se possano sembrare simili uno di questi due metodi è più efficiente dell'altro:

- `java.util.Vector` usa un incremento fisso della dimensione del vettore, ad esempio di 10 elementi.
- `java.util.ArrayList` raddoppia la dimensione del vettore.

Ora è vero che il costo di raddoppio è maggiore rispetto all'incremento fisso, ma in quanto a lungo andare il costo del raddoppio è ammortizzato su $n/2$ operazioni, il costo ammortizzato di un inserimento è $O(1)$. Questo è dimostrabile e esplicabile dalla seguente funzione:

$$c_i = \begin{cases} i & \exists k \in \mathbb{Z}_0^+ : i = 2^k + 1 \\ 1 & \text{altrimenti} \end{cases}$$

Il costo totale dell'inserimento è quindi $T(n) = \sum_{i=1}^n c_i = O(n)$ e il costo ammortizzato di un inserimento è $\frac{T(n)}{n} = \frac{O(n)}{n} = O(1)$.

8.2.2 Cancellazione - e qui le cose si complicano

Prima di analizzare il costo ammortizzato della cancellazione bisogna chiedersi:

- Quanto costa togliere un elemento da un vettore?
- Quanto costa togliere un elemento da un vettore **non ordinato**?

Il costo di una cancellazione in un vettore ordinato è $O(n)$ in quanto bisogna spostare tutti gli elementi di una posizione, mentre il costo di una cancellazione in un vettore non ordinato è $O(1)$ in quanto si può spostare l'ultimo elemento al posto dell'elemento da cancellare.

Detto ciò necessitiamo di ridurre la dimensione del vettore per evitare sprechi di memoria, ma quale è il **fattore di carico** $\alpha = \frac{\text{dim}}{\text{capacità}}$ che dobbiamo considerare? (Considerando che quando contraiamo il vettore dobbiamo allocare un nuovo vettore, copiare gli elementi e de-allocare il vecchio vettore).

Strategia Banale Una strategia banale è quella di ridurre la dimensione del vettore quando $\alpha \leq \frac{1}{2}$ il che sembrerebbe una strategia ragionevole, ma in realtà non lo è. Infatti quando consideriamo la cancellazione per la definizione di analisi ammortizzata dobbiamo considerare il caso pessimo nel quale le istruzioni vengono eseguite, finché esiste una sola istruzione che può essere eseguita allora eseguire n operazione ha solo un caso che quindi coincide con il caso pessimo. Se invece consideriamo due istruzioni (inserimento e cancellazione) allora con un fattore di carico $\alpha = \frac{1}{2}$ abbiamo un caso pessimo che si presenta come segue:

Ops:	I	I	I	I	I	R	I	R	I	R	I	R	I
Dim:	1	2	3	4	5	4	5	4	5	4	5	4	5
Cap:	8	8	8	8	8	4	8	4	8	4	8	4	8

Sostanzialmente in questo caso pessimo ad ogni operazione di cancellazione si ha un costo $O(k)$ in quanto bisogna copiare tutti gli elementi nel nuovo vettore.

Strategia Ottimale La strategia ottimale è quella di ridurre la dimensione del vettore quando $\alpha \leq \frac{1}{4}$, in quanto in questo modo il costo ammortizzato di una cancellazione è $O(1)$. Infatti se consideriamo il caso pessimo in cui ad ogni cancellazione si riduce la dimensione del vettore, il costo totale di n operazioni è $O(n)$ e il costo ammortizzato di una singola operazione è $O(1)$.

Funzione di potenziale

Possiamo definire una funzione di potenziale Φ come segue:

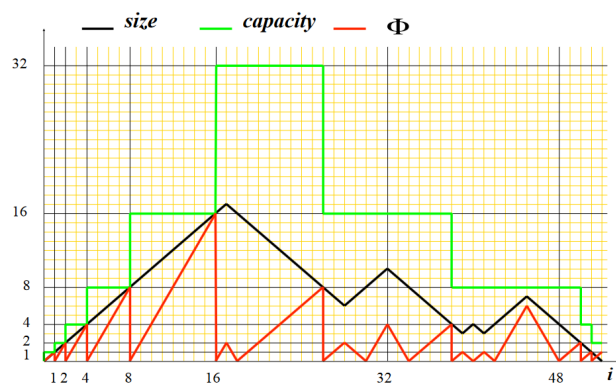
$$\Phi = \begin{cases} 2 \cdot \text{dim} - \text{cap} & \alpha \geq \frac{1}{2} \\ \text{cap}/2 - \text{dim} & \alpha \leq \frac{1}{2} \end{cases}$$

Quindi il carico dopo una espansione/contrazione sarà $\alpha = \frac{1}{2}$ con $\Phi = 0$

Prima di una espansione $\alpha = 1$ con $\Phi = \text{dim}$

Prima di una contrazione $\alpha = \frac{1}{4}$ dunque capacità = $4 \cdot \text{dim}$ con $\Phi = \text{dim}$.

Di seguito un grafico della funzione di potenziale Φ insieme al variare della dimensione *size* e alla capacità *capacity*.

Figura 8.1: Grafico della funzione di potenziale Φ

Capitolo 9

Strutture dati speciali

9.1 Code con priorità

9.1.1 Introduzione

Una coda con priorità è una struttura dati astratta, simile ad una coda, in cui ogni elemento ha un valore di priorità associato. Le code con priorità possono essere divise in *Min-Priority Queue* e *Max-Priority Queue*, a seconda che l'elemento con la priorità più bassa o più alta sia quello che viene estratto per primo.

Operazioni Permesse

Sono permesse le operazioni di inserimento in coda, estrazione dell'elemento con la priorità più alta e la modifica della priorità di un elemento.

Specifica

Algorithm 55 MINPRIORITYQUEUE

	▷ Crea una coda con capacità massima n , vuota
PRIORITYQUEUEPriorityQueue(int n)	
	▷ Restituisce true se la coda è vuota, false altrimenti
boolean ISEMPTY	
	▷ Inserisce l'elemento ITEM x con priorità int p minima
ITEM MIN	
	▷ Restituisce l'elemento con priorità minima
ITEM DELETEMIN	
	▷ Inserisce l'elemento ITEM x con priorità int p
PRIORITYITEM INSERT(ITEM x , int p)	
	▷ Modifica la priorità dell'elemento PRIORITYITEM x a int p
DECREASEKEY(PRIORITYITEM x , int p)	

Reality Check - simulatore *event-driven*

Un esempio di utilizzo delle code con priorità è la simulazione di un sistema *event-driven*, in cui gli eventi con un timestamp associato, sono inseriti in coda con priorità in base al tempo di esecuzione. Ogni evento può generarne altri con timestamp arbitrati. Una coda *min-priority* è perfetta per questo tipo di simulazione in quanto possiamo estrarre l'evento con timestamp minore.

Altre applicazioni

Altre applicazioni includono:

- Algoritmo di Dijkstra
- Codifica di Huffman
- Algoritmo di Prim per gli alberi di copertura di peso minimo

Implementazioni

Le code con priorità possono essere implementate con diverse strutture dati, di seguito una tabella riassuntiva con le relative complessità:

Metodo	Lista/Vettore non ordinato	Vettore ordinato	Lista ordinata	Albero RB
min()	$O(n)$	$O(1)$	$O(1)$	$O(\log(n))$
deleteMin()	$O(n)$	$O(n)$	$O(1)$	$O(\log(n))$
insert()	$O(n)$	$O(n)$	$O(n)$	$O(\log(n))$
decreaseKey()	$O(n)$	$O(\log(n))$	$O(n)$	$O(\log(n))$

Spesso per la memorizzazione viene usata la memoria *heap* che è una struttura dati speciale che associa i vantaggi di un albero per l'esecuzione ($O(\log(n))$) ed inoltre prende i vantaggi di un vettore.

9.1.2 Vettore *Heap*

Alberi binari

Definizione 9.1 (Albero binario perfetto). Si definisce **albero binario perfetto** un albero binario che rispetti tutte le seguenti condizioni:

- Tutte le foglie hanno la stessa profondità h
- Tutti i nodi interni hanno esattamente grado 2
- Dato il numero di nodi n , ha altezza $h = \lfloor \log_2(n) \rfloor$
- Data l'altezza h , ha $n = 2^{h+1} - 1$ nodi

Definizione 9.2 (Albero binario completo). Si definisce **albero binario completo** un albero binario che rispetti tutte le seguenti condizioni:

- Tutte le foglie hanno profondità h o $h - 1$
- Tutti i nodi al livello h sono "accatastati a sinistra"
- Tutti i nodi interni hanno grado 2 eccetto al più un solo nodo con grado 1
- Dato il numero di nodi n , ha altezza $h = \lfloor \log_2(n) \rfloor$

Alberi binari *heap*

Definizione 9.3 (Albero $\{max-min\}$ -*heap*). Un **albero *max-heap*** o **albero *min-heap*** è un albero binario completo tale che il vettore memorizzato in ogni nodo è maggiore (minore) dei valori memorizzati nei suoi figli.

È utile notare come un albero *max-heap* non impone una relazione di **ordinamento totale** fra i figli di un nodo questo però non esclude che non sia un **ordinamento parziale**. Questo è verificabile in quanto si applica: la proprietà di **riflessività** (Ogni nodo è \geq se stesso), la proprietà di **anti-simmetria** ($n \geq m \wedge m \geq n \Rightarrow n = m$) e la proprietà **transitiva** ($n \geq m \wedge m \geq p \Rightarrow n \geq p$). Un ordinamento parziale è più debole ma più semplice da costruire.

Vettore *heap* Un albero *heap* può essere rappresentato tramite un **vettore *heap***. In particolare rispettiamo le seguenti regole per la memorizzazione ($A[1 \dots n]$):

Radice $\text{root}() = 1$

Padre nodo i $p(i) = \lfloor \frac{i}{2} \rfloor$

Figlio sinistro nodo i $l(i) = 2i$

Figlio destro nodo i $r(i) = 2i + 1$

vengono rispettate le seguenti proprietà: su un albero *max-heap* $A[i] \geq A[l(i)]$ e $A[i] \geq A[r(i)]$. Su un albero *min-heap* $A[i] \leq A[l(i)]$ e $A[i] \leq A[r(i)]$.

9.1.3 *HeapSort*

L'algoritmo *HeapSort* è un algoritmo di ordinamento *in-place* che sfrutta un *max-heap* nel vettore e sposta l'elemento più grande in fondo al vettore. L'algoritmo è composto da due fasi: $\text{heapBuild}()$ e $\text{maxHeapRestore}()$.

Funzione $\text{maxHeapRestore}()$

L'algoritmo $\text{maxHeapRestore}()$ prende in input un vettore A ed un indice i tale per cui gli alberi binari con radici $l(i)$ e $r(i)$ siano *max-heap*. Notare come ciò non esclude il fatto che $A[i]$ sia minore di $A[l(i)]$ o $A[r(i)]$. L'algoritmo $\text{maxHeapRestore}()$ ha come obiettivo quello di modificare *in-place* il vettore A in modo che l'albero con radice i sia un *max-heap*.

Algorithm 56 $\text{MAXHEAPRESTORE}(\text{ITEM } A, \text{int } i, \text{int } \text{dim})$

```

int  $\text{max} \leftarrow i$ 
if  $l(i) \leq \text{dim} \ \& \ A[l(i)] > A[\text{max}]$  then
    int  $\text{max} \leftarrow l(i)$ 
if  $r(i) \leq \text{dim} \ \& \ A[r(i)] > A[\text{max}]$  then
    int  $\text{max} \leftarrow r(i)$ 
if  $\text{max} \neq i$  then
     $\text{SWAP}(A, i, \text{max})$ 
     $\text{MAXHEAPRESTORE}(A, \text{max}, \text{dim})$ 

```

Complessità La complessità dell'algoritmo $\text{maxHeapRestore}()$ dove, ad ogni chiamata vengono eseguiti $O(1)$ confronti, se il nodo i non è il massimo si chiama ricorsivamente $\text{maxHeapRestore}()$ su uno dei due figli, l'esecuzione termina quando si raggiunge una foglia ed l'altezza dell'albero è $\lfloor \log(n) \rfloor$. Allora la complessità è $O(\log(n))$ in quanto il numero massimo di chiamate ricorsive è $T(n) = O(\log(n))$.

Dimostrazione della correttezza dell'algoritmo

Teorema 9.1. Al termine dell'esecuzione, l'albero radicato in $A[i]$ rispetta la proprietà di *max-heap*.

Dimostrazione. La dimostrazione è per induzione sull'altezza h dell'albero:

Base ($h = 0$): l'albero è una foglia e rispetta la proprietà di *max-heap*.

Induzione ($\forall i \leq h \rightarrow A[i]$ albero radicato rispetta la proprietà di *max-heap*): Si distinguono 2 casi:

- $A[i] \geq A[l(i)]$ e $A[i] \geq A[r(i)]$: l'albero radicato in $A[i]$ rispetta la proprietà di *max-heap*
- $A[i] < A[l(i)]$ e $A[i] > A[r(i)]$: si scambia $A[i]$ con $A[l(i)]$ dopo lo scambio siamo sicuri che $A[i] \geq A[l(i)]$, $A[i] \geq A[r(i)]$ allora il sotto-albero $A[r(i)]$ è inalterato e rispetta la proprietà *heap*, il sotto-albero $A[l(i)]$ potrebbe non rispettare la proprietà *heap* e quindi si chiama ricorsivamente $\text{maxHeapRestore}()$ su $A[l(i)]$, questo ha altezza minore di h e quindi dopo la chiamata rispetta la proprietà di *max-heap*.
- Il caso simmetrico si dimostra in modo analogo.

Per il principio di induzione, il corretto funzionamento dell'algoritmo è dimostrato. □

Funzione heapBuild()

Il principio di funzionamento dell'algoritmo heapBuild() è così descrivibile: a partire da un vettore $A[1 \dots n]$ da ordinare, dove tutti i nodi $A[\lfloor \frac{n}{2} \rfloor + 1 \dots n]$ sono foglie dell'albero e quindi *heap* contenenti 1 elemento. La procedura heapBuild() attraversa i nodi restanti da $\lfloor \frac{n}{2} \rfloor$ a 1 ed esegue maxHeapRestore() su ogni nodo.

Algorithm 57 HEAPBUILD(ITEM \square A , int n)

```

for  $i \leftarrow \lfloor \frac{n}{2} \rfloor$  to 1 do
    MAXHEAPRESTORE( $A, i, n$ )

```

Dimostrazione. Quando viene eseguito l'algoritmo heapBuild() si ha che all'inizio di ogni interazione del ciclo **for** i nodi $[i + 1 \dots n]$ sono radice di un *heap*.

All'inizio del ciclo "prendiamo" $i = \lfloor \frac{n}{2} \rfloor$, supponendo che $\lfloor \frac{n}{2} \rfloor + 1$ non sia una foglia, allora ha almeno un figlio sinistro che sarà allocato in posizione $2\lfloor \frac{n}{2} \rfloor + 2$ ovvero $n + 1$ o $n + 2$ il che è un assurdo in quanto n è la dimensione massima del nostro vettore. Allora $\lfloor \frac{n}{2} \rfloor + 1$ è una foglia e quindi un *heap* di un solo elemento. Possiamo applicare l'algoritmo maxHeapRestore() sul nodo i in quanto $2i < 2i + 1 \leq n$ e questi sono entrambi radici di un *heap*. Quando l'algoritmo termina tutti i nodi $[i \dots n]$ sono radici di un *heap* rendendo il nodo 1 la radice di un *heap* e quindi A un *heap* completo. \square

Complessità La complessità dell'algoritmo heapBuild(), data dal numero di chiamate a maxHeapRestore() è $O(n \log n)$ in quanto il numero di chiamate a maxHeapRestore() va da $\lfloor \frac{n}{2} \rfloor$ a 1 il che anche se minore di n è comunque $O(n)$. Inoltre la complessità di maxHeapRestore() è $O(\log n)$. Il che porta ad una complessità totale di $O(n \log n)$.

Funzione heapSort()

Una volta costruito l'*heap* grazie alle funzioni heapBuild() e maxHeapRestore() possiamo assemblare l'algoritmo di ordinamento *HeapSort*.

Principio di funzionamento *HeapSort* si basa sul fatto che il primo elemento contenga il massimo, viene quindi scambiato con l'ultimo elemento, viene quindi eseguito maxHeapRestore() sui nodi $[1 \dots i - 1]$, dove i è un contatore da n a 2, si ripete il processo fino a che non si arriva a 2.

Algorithm 58 HEAPSORT(ITEM \square A , int n)

```

    HEAPBUILD( $A, n$ )
    for  $i \leftarrow n$  downto 2 do
        SWAP( $A, 1, i$ )
        MAXHEAPRESTORE( $A, 1, i - 1$ )

```

Complessità In quanto l'algoritmo heapBuild() ha complessità $\Theta(n)$, l'algoritmo heapMaxRestore() ha complessità $\Theta(\log i)$, allora possiamo esprimere la complessità dell'algoritmo *HeapSort* come:

$$T(n) = \sum_{i=2}^n \log i + \Theta(n) = \Theta(n \log n)$$

Dunque la complessità dell'algoritmo *HeapSort* è $\Theta(n \log n)$.

9.1.4 Implementazione code con priorità

Di seguito viene riportata una possibile implementazione di una coda con priorità utilizzando un *min-heap*, e dunque minHeapRestore().

Memorizzazione**Algorithm 59** PRIORITYITEM

int <i>priority</i>	▷ Priorità
ITEM <i>item</i>	▷ Elemento
int <i>pos</i>	▷ Posizione nel vettore <i>heap</i>

Algorithm 60 swap(PRIORITYITEM[] *H*, **int** *i*, **int** *j*)

```

PRIORITYITEM temp ← H[i]
H[i] ← H[j]
H[j] ← temp
H[i].pos ← i
H[j].pos ← j

```

Inizializzazione**Algorithm 61** PRIORITYQUEUE

int <i>capacity</i>	▷ Numero massimo di elementi nella coda
int <i>dim</i>	▷ Numero di elementi attualmente nella coda
PRIORITYITEM[] <i>H</i>	▷ Vettore heap
PRIORITYQUEUE	

```

function PRIORITYQUEUE(int n)
  PRIORITYQUEUE t = new PRIORITYQUEUE
  t.capacity ← n
  t.dim ← 0
  t.H ← new PRIORITYITEM[1...n]
  return t

```

Inserimento**Algorithm 62** PRIORITYITEM insert(ITEM *x*, **int** *p*)

```

precondition dim < capacity
dim ← dim + 1
H[dim] = new PRIORITYITEM
H[dim].value ← x
H[dim].priority ← p
H[dim].pos ← dim
int i ← dim
while i > 1 & H[i].priority < H[p(i)].priority do
  SWAP(H, i, p(i))
  i ← p(i)
return H[i]

```

minHeapRestore()

Algorithm 63 minHeapRestore(PRIORITYITEM[] *A*, int *i*, int *dim*)

```

int min ← i
if l(i) ≤ dim & A[l(i)].priority < A[min].priority then
    int min ← l(i)
if r(i) ≤ dim & A[r(i)].priority < A[min].priority then
    int min ← r(i)
if min ≠ i then
    SWAP(A, i, min)
    MINHEAPRESTORE(A, min, dim)

```

Cancellazione e/o Lettura minimo

Algorithm 64 ITEM deleteMin()

```

precondition dim > 0
SWAP(H, 1, dim)
dim ← dim − 1
MINHEAPRESTORE(H, 1, dim)
return H[dim + 1].value

```

Algorithm 65 ITEM min()

```

precondition dim > 0
return H[1].value

```

Decremento della priorità

Algorithm 66 decrease(PRIORITYITEM *x*, int *p*)

```

precondition p < x.priority
x.priority ← p
int i ← x.pos
while i > 1 & H[i].priority < H[p(i)].priority do
    SWAP(H, i, p(i))
    i ← p(i)

```

Complessità

Da notare come tutte le operazioni che modificano gli *heap* lasciano inalterata la correttezza della struttura dati, questo o lungo un cammino radice-foglia nel caso della funzione `deleteMin()` o lungo un cammino foglia-radice nel caso delle funzioni `insert()` e `decrease()`. In quanto l'altezza è $\lfloor \log n \rfloor$ la complessità di "sistemare" l'*heap* è $O(\log n)$. Da questo possiamo dire che le operazioni sul *heap* seguono la seguente complessità:

Operazione	Complessità
<code>insert()</code>	$O(\log n)$
<code>deleteMin()</code>	$O(\log n)$
<code>min()</code>	$\Omega(1)$
<code>decrease()</code>	$O(\log n)$