Algoritmi e strutture dati

Giacomo Fantoni
Telegram: @GiacomoFantoni
Github: https://github.com/giacThePhantom/AlgoritmiStruttureDati

19 febbraio 2020

Indice

Capitolo 1

Introduzione

Problema computazionale

Dati un dominio di input e uno di output, un problema computazionale è rappresentato dalla funzione matematica che associa un elemento del dominio di output ad ogni elemento del dominio di input.

Algoritmo

Dato un problema computazionale, un algoritmo è un procedimento effettivo espresso tramite una funzione di passi elementari ben specificati in un sistema formale di calcolo che risolve il problema in tempo finito.

1.1 Descrizione di un algoritmo

Per descrivere un algoritmo si rende necessario utilizzare un linguaggio formale ben definito detto pseudo-codice, indipendente dall'implementazione effettiva ma con dettaglio sufficiente a descrivere i passaggi necessari alla descrizione dell'algoritmo.

1.1.1 Pseudo-codice

- $a = b$.
- $a \leftrightarrow b \equiv tmp = a; a = b; b = tmp$.
- $T[]A = \textbf{new}\ T[1 \ldots n]$.
- $T[][]A = \textbf{new}\ T[1 \ldots n][1 \ldots m]$.
- Tipi in grassetto.
- **and**, **or**, **not**.

- $==, \neq, \geq, \leq$.
- $+, -, \cdot, /, \lfloor x \rfloor, \lceil x \rceil, \log, x^2, \cdots$.
- $iif(condizione, v_1, v_2)$.
- **if** *condizione* **then** *istruzione*.
- **if** *condizione* **then** $istruzione_1$ **else then** $istruzione_2$.
- **while** *condizione* **do** *istruzione*.
- **foreach** $elemento \in insieme$ **do** *istruzione*.
- **return**
- % commento.

1.2 Valutazione degli algoritmi

1.2.1 Efficienza

Si definisce complessità di un algoritmo l'analisi delle risorse necessarie per la sua risoluzione, in funzione di tipologia e dimensione di input. Le risorse si distinguono in tempo, memoria e banda (per gli algoritmi distribuiti).

Tempo

Il numero di secondi necessari alla risoluzione dell'algoritmo dipende da troppi fattori, si utilizzano pertanto tecniche di analisi che prendono in considerazione il numero di operazioni rilevanti, quelle che caratterizzano lo scopo dell'algoritmo.

1.2.2 Correttezza

Per valutare la correttezza di algoritmi si devono considerare le invarianti:

- Invariante: una condizione che deve rimanere vera sempre in un certo punto del programma.
- Invariante di ciclo: una condizione che deve rimanere vera all'inizio dell'iterazione di un ciclo.
- Invariante di classe: una condizione sempre vera al termine dell'esecuzione di un metodo di una classe.

Invariante di ciclo e algoritmi iterativi

L'invariante di ciclo permette di dimostrare la correttezza degli algoritmi iterativi attraverso il principio di induzione:

- Inizializzazione (caso base): l'invariante è vera prima della prima iterazione.
- Conservazione (passo induttivo): se la condizione è vera prima di un'interazione allora rimane vera al suo termine.
- Conclusione: quando il ciclo termina l'invariante deve rappresentare la correttezza dell'algoritmo.

Capitolo 2

Analisi di algoritmi

Per definire la complessità di un algoritmo occorre definire una funzione che ha come dominio la dimensione dell'input e come insieme immagine il tempo.

Dimensione dell'input

La dimensione dell'input può essere definita secondo due criteri:

- Criterio di costo logaritmico: la taglia dell'input è il numero di bit necessari a rappresentarlo.
- Criterio di costo uniforme: la taglia dell'input è il numero di elementi di cui è costituito.

In molti casi si può assumere che gli elementi siano costituiti da un numero costante di bit, e in tal caso le due misure coincidono a meno di una costante moltiplicativa.

Definizione di tempo

Si definisce il tempo come il numero di istruzioni elementari necessarie al completamento dell'algoritmo. Si definisce elementare una funzione che può essere svolta in tempo costante dal processore.

2.1 Modelli di calcolo

Un modello di calcolo è una rappresentazione astratta del calcolatore. L'astrazione permette di nascondere dei dettagli, il suo realismo permette di riflettere con un certo grado di precisione una situazione reale e la potenza matematica di trarre conclusioni formali sul costo.

2.1.1 Macchina di Turing

Una macchina di Turing è una macchina ideale che manipola i dati contenuti su un nastro di lunghezza infinita secondo un insieme prefissato di regole. Ad ogni passo la macchina di Turing:

- Legge il simbolo sotto la testina.
- Modifica il proprio stato interno.
- Scrive un nuovo simbolo nella cella.
- Muove la testina a destra o a sinistra.

Questo modello è fondamentale per lo studio della calcolabilità ma è troppo dettagliato per l'analisi.

2.1.2 Random Access Machine (RAM)

Memoria

La memoria è costituita da un numero infinito di celle di dimensione finita a cui si accede, indipendentemente dalla posizione, in tempo costante.

Processore (singolo)

Un insieme di istruzioni simili a quelle reali: algebriche, logiche e di salto.

Costo delle istruzioni elementari

Uniforme e ininfluente ai fini dell'analisi.

2.2 Funzioni di costo, analisi asintotica

Per studiare la complessità di un algoritmo si analizza il suo comportamento asintotico, ovvero quando la dimensione del suo input tende a infinito.

2.2.1 Notazione O

Sia $g(n)$ una funzione di costo, si indica con $O(g(n))$ l'insieme delle funzioni $f(n)$ tali per cui:

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

$g(n)$ si dice limite asintotico superiore di $f(n)$, ovvero $f(n)$ cresce al più come $g(n)$.

2.2.2 Notazione Ω

Sia $g(n)$ una funzione di costo, si indica con $\Omega(g(n))$ l'insieme delle funzioni $f(n)$ tali per cui:

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

$g(n)$ si dice limite asintotico inferiore di $f(n)$, ovvero $f(n)$ cresce almeno quanto $g(n)$.

2.2.3 Notazione Θ

Sia $g(n)$ una funzione di costo, si indica con $\Theta(g(n))$ l'insieme delle funzioni $f(n)$ tali per cui:

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

$f(n)$ cresce esattamente come $g(n)$ e $f(n) = \Theta(g(n))$ se e solo se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$.

2.2.4 Proprietà della notazione asintotica

Espressioni polinomiali

Enunciato

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

Limite superiore

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

Dimostrazione

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

Vera per $c \geq (a_k + |a_{k-1}| + \cdots + |a_1| + |a_0|)$ e per $m = 1$.

Limite inferiore

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

Dimostrazione

$$\begin{aligned} f(n) &= a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 \\ &\geq a_k n^k - |a_{k-1}| n^{k-1} - \cdots - |a_1| n - |a_0| \\ &\geq a_k n^k - |a_{k-1}| n^{k-1} - \cdots - |a_1| n^{k-1} - |a_0| n^{k-1} \qquad \forall n \geq 1 \\ &= (a_k + |a_{k-1}| + \cdots + |a_1| + |a_0|) n^k \\ &\overset{?}{\geq} d n^k \end{aligned}$$

Vera per $d \leq a_k - \frac{|a_{k-1}|}{n} - \frac{|a_{k-2}|}{n} - \cdots - \frac{|a_1|}{n} - \frac{|a_0|}{n} > 0 \Leftrightarrow n > \frac{|a_{k-1}+\cdots+|a_0|}{a_k}$.

Dualità

Enunciato

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

Dimostrazione

$$\begin{aligned} f(n) = O(g(n)) &\Leftrightarrow f(n) \leq c g(n), \forall n \geq m \\ &\Leftrightarrow g(n) \geq \frac{1}{c} f(n), \forall n \geq m \\ &\Leftrightarrow g(n) \geq c' = f(n), \forall n \geq m, c' = \frac{1}{c} \\ &\Leftrightarrow g(n) = \Omega(f(n)) \end{aligned}$$

Eliminazione delle costanti

Enunciato

$$f(n) = O(g(n)) \Leftrightarrow af(n) = O(g(n)), \forall a > 0 \tag{2.1}$$

$$f(n) = \Omega(g(n)) \Leftrightarrow af(n) = \Omega(g(n)), \forall a > 0 \tag{2.2}$$

Dimostrazione

$$\begin{aligned} f(n) = O(g(n)) &\Leftrightarrow f(n) \leq c g(n), \forall n \geq m \\ &\Leftrightarrow af(n) \leq acg(n), \forall n \geq m, \forall a \geq 0 \\ &\Leftrightarrow af(n) \leq c' g(n), \forall n \geq m, c' = ac > 0 \\ &\Leftrightarrow af(n) = O(g(n)) \end{aligned}$$

La dimostrazione è analoga per il limite inferiore.

Sommatoria

Enunciato

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

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

Dimostrazione

$$\begin{aligned} f_1(n) = O(g_1(n)) \wedge f_2(n) = O(g_2(n)) &\Rightarrow \\ f_1(n) \leq c_1 g_1(n) \wedge f_2(n) \leq c_2 g_2(n) &\Rightarrow \\ f_1(n) + f_2(n) \leq c_1 g_1(n) + c_2 g_2(n) &\Rightarrow \\ f_1(n) + f_2(n) \leq \max(c_1, c_2)(2\max(g_1(n), g_2(n))) &\Rightarrow \\ f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)) & \end{aligned}$$

La dimostrazione è analoga per il limite inferiore.

Prodotto

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

Dimostrazione

$$\begin{aligned} f_1(n) = O(g_1(n)) \wedge f_2(n) = O(g_2(n)) &\Rightarrow \\ f_1(n) \leq c_1 g_1(n) \wedge f_2(n) \leq c_2 g_2(n) &\Rightarrow \\ f_1(n) \cdot f_2(n) \leq c_1 c_2 g_1(n) g_2(n) &\Rightarrow \\ f_1(n) f_2(n) = O(g_1(n) g_2(n)) & \end{aligned}$$

Simmetria

Enunciato

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

Dimostrazione Grazie alla proprietà di dualità:

$$\begin{aligned} f(n) = \Theta(g(n)) \Rightarrow \quad & f(n) = O(g(n)) \Rightarrow \quad & g(n) = \Omega(f(n)) \\ f(n) = \Theta(g(n)) \Rightarrow \quad & f(n) = \Omega(g(n)) \Rightarrow \quad & g(n) = O(f(n)) \end{aligned}$$

Transitività

Enunciato

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

Dimostrazione

$$\begin{aligned} f(n) = O(g(n)) \wedge g(n) = O(h(n)) &\Rightarrow \\ f(n) \leq c_1 g(n) \wedge g(n) \leq c_2 h(n) &\Rightarrow \\ f(n) \leq c_1 c_2 h(n) &\Rightarrow \\ f(n) = O(h(n)) & \end{aligned}$$

Logaritmi

Enunciato Si vuole provare che $\log n = O(n)$. Si dimostri pertanto per induzione che:

$$\exists c > 0, \exists m \geq 0 : \log n \leq cn, \forall n \geq m$$

Dimostrazione

- Caso base, $n = 1$: $\log 1 = 0 \leq cn = c\dot{1} \Leftrightarrow c \geq 0$.
- Ipotesi induttiva: sia $\log k \leq ck, \forall k \leq n$.
- Passo induttivo: si dimostri la proprietà per $n + 1$.

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

$1 + cn \leq c(n+1) \Leftrightarrow c \geq 1$

2.2.5 Notazioni o, ω

Notazione o

Sia $g(n)$ una funzione di costo, si indica con $o(g(n))$ l'insieme delle funzioni $f(n)$ tali per cui:

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

Notazione ω

Sia $g(n)$ una funzione di costo, si indica con $\omega(g(n))$ l'insieme delle funzioni $f(n)$ tali per cui:

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

Significato

Utilizzando il concetto di limite si noti come:

- $\lim_{x\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) = o(g(n))$.
- $\lim_{x\to\infty} \frac{f(n)}{g(n)} \neq 0 \Rightarrow f(n) = \Theta(g(n))$.
- $\lim_{x\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) = \omega(g(n))$.
- $f(n) = o(g(n)) \Rightarrow f(n) = O(g(n))$.
- $f(n) = \omega(g(n)) \Rightarrow f(n) = \Omega(g(n))$.

2.2.6 Classificazione delle funzioni

Espandendo le relazioni dimostrate è possibile ottenere un ordinamento delle principali espressioni. Si consideri 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

Definizioni

- Equazione di ricorrenza: calcolare la complessità di un algoritmo ricorsivo richiede le creazione di un'equazione di ricorrenza, una formula matematica definita in maniera ricorsiva.
- Forma chiusa: l'obiettivo è partire dall'equazione di ricorrenza e trasformarla in una forma chiusa in modo da comprendere la classe di complessità dell'algoritmo.

2.3.1 Analisi per livelli

Questo metodo di risoluzione delle equazioni ricorsive detto anche metodo dell'albero di ricorsione consiste nell'espandere la ricorrenza in un albero i cui nodi rappresentano i costi ai vari livelli della ricorsione considerando poi il livello più basso in cui tutte le chiamate sono state ricondotte al caso base. Si otterrà pertanto una sommatoria da cui si potrà derivare la forma chiusa e la classe di complessità.

2.3.2 Metodo della sostituzione

Questo metodo di risoluzione delle equazioni ricorsive, detto anche metodo per tentativi, consiste nel cercare di indovinare una soluzione in base alla propria esperienza e di tentare di dimostrare tale soluzione per induzione.

2.3.3 Metodo delle ricorrenze comuni

Questo metodo di risoluzione delle equazioni ricorsive, detto anche metodo dell'esperto, mette a disposizione dei teoremi che permettono di risolvere facilmente ampie classi di equazioni di ricorrenza.

Ricorrenze lineari con partizione bilanciata

Teorema Siano a e b costanti intere tali che $a \geq 1$ e $b \geq 2$ e c e β costanti reali tali che $c > 0$ e $\beta \geq 0$. Sia $T(n)$ l'equazione di ricorrenza nella forma:

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

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

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

Estensione delle ricorrenze lineari con partizione bilanciata

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

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

Sono dati tre casi.

- $\exists \varepsilon > 0 : f(n) = O(n^{\log_b a - \varepsilon}) \Rightarrow T(n) = \Theta(n^{\log_b a})$.
- $f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(f(n) \log n)$.
- $\exists \varepsilon > 0 : f(n) = \Omega(n^{\log_b a + \varepsilon}) \wedge \exists c : 0 < c < 1, \exists m > 0 : af(\frac{n}{b}) \leq cf(n), \forall n \geq m \Rightarrow T(n) = \Theta(f(n))$.

Ricorrenze lineari di ordine costante

Siano $a_1, \ldots, a_n$ costanti intere non negative, con h costante positiva, $c > 0$ e $\beta \geq 0$ costanti reali e $T(n)$ una relazione di ricorrenza nella forma:

$$T(n) = \begin{cases} \sum\limits_{1 \leq i \leq h} a_i T(n-i) + cn^\beta & n > m \\ \Theta(1) & n \leq m \leq h \end{cases}$$

Posto $a = \sum\limits_{1 \leq i \leq h} a_i$ allora:

- $T(n) = \Theta(n^{\beta+1})$ se $a = 1$.
- $T(n) = \Theta(a^n n^\beta)$ se $a \geq 2$.

2.4 Relazione tra complessità di un problema e di un algoritmo

Un problema ha complessità $O(f(n))$ se esiste un algoritmo che lo risolve con complessità $O(f(n))$. Un problema ha complessità $\Omega(f(n))$ se tutti gli algoritmi che lo risolvono hanno complessità $\Omega(f(n))$.

2.4.1 Complessità in tempo di un algoritmo

La quantità di tempo richiesta per input di dimensione n:

- $O(f(n))$: per tutti gli input l'algoritmo costa al più $f(n)$.
- $\Omega(f(n))$: per tutti gli input l'algoritmo costa almeno $f(n)$.
- $\Theta(f(n))$: per tutti gli input l'algoritmo richiede $f(n)$.

2.4.2 Complessità in tempo di un problema computazionale

La quantità di tempo richiesta per input di dimensione n:

- $O(f(n))$: complessità del miglior algoritmo che risolve il problema.
- $\Omega(f(n))$: dimostrare che nessun algoritmo può risolvere il problema in un tempo inferiore a $\Omega(f(n))$.
- $\Theta(f(n))$: algoritmo ottimo.

2.5 Valutare algoritmi in base alla tipologia di input

In alcuni casi gli algoritmi si comportano in maniera diversa in base a caratteristiche dell'input. Conoscerle in anticipo permette di scegliere il miglior algoritmo per la situazione.

2.5.1 Tipologie di analisi

- Analisi del caso pessimo: il tempo di esecuzione è il limite superiore al tempo di esecuzione per un qualisasi input.
- Analisi del caso medio: molto difficile in quanto si deve trovare una distribuzione uniforme degli input.
- Analisi del caso ottimo: ha senso se si conoscono caratteristiche dell'input.

2.5.2 Algoritmi di ordinamento

Il problema di ordinamento è rappresentato da una sequenza $A = a_1, \ldots, a_n$ di valori in input e dà in output una sequenza $B = b_1, \ldots, b_n$ permutazione di B tale per cui $\forall 0 < i < n-1, b_i \leq b_{i+1}$.

Selection sort

Cerco il minimo e lo metto nella posizione corretta, riducendo il problema ai restanti $n-1$ valori.

```
: selectionSort(item [] A, int n)
    for i = 1 to n-1 do
        int min = min(A, i, n)
        A[i] ↔ A[min]
```

```
: min(item [] A, int i, int n)
    %Posizione del minimo parziale
    int min = i
    for j = i + 1 to n do
        if A[j] ≤ A[min] then
            %Nuovo minimo parziale
            min = j
        return min
```

Complessità

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

InsertionSort

Algoritmo efficiente per ordinare piccoli insiemi. In cui si prende l'i-esimo elemento e lo si mette nella posizione corretta rispetto agli elementi precedenti, proseguendo fino alla fine.

```
: insertionSort(item [] A, int n)
    for i = 2 to n do
        item temp = A[i]
        int j = i
        while j > 1 and A[j-1] > temp do
            A[j] = A[j - 1]
            j -= 1
        A[j] = temp
```

Correttezza e complessità DA COMPLETARE

MergeSort

Il mergeSort si basa sulla tecnica di divide-et-impera in quanto divide il vettore di n elementi in due sottovettori di $\frac{n}{2}$ elementi, chiama il mergeSort ricorsiva-

mente su quei due elementi e unisce (merge) le due sequenze ordinate. Si sfrutta il fatto che i due sottovettori sono già ordinati per ordinare più velocemente.

```
: merge(item [] A, int first, int last, int mid)
int i, j, k , h
i = first
j = mid + 1
k = first
while i ≤ mid and j ≤ last do
    if A[i] ≤ A[j] then
        B[k] = A[i]
        i += 1
    else
        B[k] = A[j]
        j += 1
    k += 1
j = last
for h = mid down to i do
    A[j] = A[h]
    j -= 1
for j = first to k-1 do
    A[j] = B[j]
```

merge()

Costo computazionale $O(n)$

mergeSort()

```
: mergeSort(item [] A, int first, int last)
if first < last then
    int mid = ⌊(first + last)/2⌋
    mergeSort(A, first, mid)
    mergeSort(A, mid + 1, last)
    merge(A, first, last, mid)
```

Costo computazionale Si assuma per semplificare che $n = 2^k$ in modo che $k = \log n$ e tutti i sottovettori hanno dimensioni di potenze esatte di due.

Si ottiene così l'equazione di ricorrenza:

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

Si noti come ad ogni chiamata ricorsiva si svolga un'operazione di merge di costo $O(1)$ e k vari tra 0 e $\log n$. Si ottiene pertanto $O(\sum_{i=0}^{k} s^i \frac{n}{2^i}) = O(\sum_{i=0}^{k} n) = O(kn) \Leftrightarrow O(n \log n)$.

2.6 Analisi ammortizzata

Si intende per analisi ammortizzata una tecnica di analisi di complessità che valuta il tempo per eseguire nel caso pessimo una sequenza di operazioni su una struttura dati. Esistono operazioni più o meno costose e se le operazioni costose sono meno frequenti allora il loro costo può essere ammortizzato da quelle meno costose. A differenza dell'analisi del caso medio è deterministica su operazioni multiple e verifica il caso pessimo.

2.6.1 Metodi per l'analisi ammortizzata

Metodo dell'aggregazione

In questo metodo si calcola la complessità $T(n)$ per eseguire n operazioni in sequenza nel caso pessimo. Grazie alla sequenza si considera l'evoluzione della struttura dati data una sequenza di operazioni, considerando la sequenza pessima e sommando insieme tutte le complessità individuali. Successivamente questo $T(n)$ viene diviso per il numero di operazioni in modo da verificare la complessità ammortizzata di un'operazione nella sequenza.

Metodo degli accantonamenti

Alle operazioni vengono assegnati costi ammortizzati che possono essere minori o maggiori del loro costo effettivo. Questa differenza è dovuta al fatto che le operazioni meno costose vengono caricate di un costo aggiuntivo detto credito: $costo\ ammortizzato = costo\ effettivo + credito\ prodotto$. Questo credito accumulato viene speso dalle operazioni più costose: $costo\ ammortizzato = costo\ effettivo - credito\ consumato$. Si deve dimostrare che la somma dei costi ammortizzato a_i è un limite superiore alla somma dei costi effettivi: $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} a_i$ e che il valore così ottenuto è "poco costoso". Considerando il caso pessimo, la dimostrazione deve valere per tutte le sequenze e il credito dopo la t-esima operazione deve essere sempre positivo: $\sum_{i=1}^{t} a_i \leq \sum_{i=1}^{t} c_i \geq 0$.

Metodo del potenziale

Lo stato del sistema viene descritto attraverso una funzione di potenziale $\Phi(D)$. Le operazioni meno costose devono incrementare $\Phi(D)$ e quelle più costose decrementarlo. Il costo ammortizzato è pari al costo effettivo sommato alla differenza di potenziale: $a_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$. Per una sequenza di operazioni di lunghezza n, se $\Phi(D_n) - \Phi(D_0) \geq 0$ il costo ammortizzato A è un limite superiore al costo reale.

Capitolo 3

Strutture dati

Si intende per dato in un linguaggio di programmazione un valore che una variabile può assumere. Un tipo di dato astratto è una collezione di valori e un insieme di operazioni ammesse su quei valori. Si dicono primitivi i tipi di dati forniti direttamente dal linguaggio di programmazione. La definizione di un tipo di dato astratto si divide nella specifica, una descrizione di alto livello di come può essere utilizzata e nell'implementazione, la realizzazione vera e propria di basso livello. Le strutture dati sono collezioni di dati, caratterizzate dall'organizzazione della collezione stessa più che dal tipo di dati contenuti. Le strutture dati sono caratterizzate da un insieme di operatori che permettono di manipolarne la struttura e un modo sistematico per organizzare l'insieme di dati. Le strutture dati si dividono in:

- Lineari o non lineari rispetto alla presenza o meno di una sequenza.
- Statiche o dinamiche se la dimensione della struttura può variare o meno.
- Omogenee o disomogenee se possono contenere uno o diversi tipi di dati.

3.1 Sequenza

Una sequenza è una struttura dati dinamica, lineare che rappresenta una sequenza ordinata di valori, dove un valore può comparire più di una volta. È importante l'ordine all'interno della sequenza.

Operazioni ammesse

- Data una posizione è possibile aggiungere o eliminare un elemento in quella posizione. Si considerano anche per comodità le posizioni pos_0 e pos_{n+1}.
- È possibile accedere direttamente alla testa e alla coda della sequenza.
- È possibile accedere sequenzialmente a tutti gli altri elementi.

Specifica

: Sequence

%Restituisce True se la sequenza è vuota
boolean `isEmpty()`

%Restituisce True se p è uguale a pos_0 o a pos_{n+1}
boolean `finished`(Pos p)

%Restituisce la posizione del primo elemento
Pos `head()`

%Restituisce la posizione dell'ultimo elemento
Pos `tail()`

%Restituisce la posizione dell'elemento che segue p
Pos `next`(Pos p)

%Restituisce la posizione dell'elemento che precede$p$
Pos `prev`(Pos p)

%Inserisce l'elemento v di tipo Item nella posizione p
%Restituisce la posizione del nuovo elemento che diventa predecessore di p
Pos `insert`(Pos p, Item v)

%Rimuove l'elemento alla posizione p
%Restituisce la posizione del successore di p
%Che diventa il successore del predecessore di p
Pos `remove`(Pos p)

%Legge l'elemento di tipo Item contenuto nella posizione p
Item `read`(Pos p)

%Scrive l'elemento v di tipo Item nella posizione p
`write`(Pos p, Item v)

3.2 Insiemi

Per insieme si intende una struttura dati dinamica, non lineare che memorizza una collezione non ordinata di elementi senza valori ripetuti. L'ordinamento tra i valori sarà dato dall'eventuale relazione d'ordine definita sul tipo di elementi stessi.

Operazioni ammesse

- Operazioni base:
  - Inserimento.
  - Cancellazione.
  - Verifica contenimento.
- Operazioni di ordinamento:
  - Massimo.
  - Minimo.
- Operazioni insiemistiche:
  - Unione.
  - Intersezione.
  - Differenza.
- Iteratori.
  - **foreach** $x \in S$ **do**.

Specifica

: Set
%Restituisce la cardinalità dell'insieme
int size()
%Restituisce True se x è contenuto nell'insieme
boolean contains(Item x)
%Inserisce x nell'insieme se non è già presente
insert(Item x)
%Rimuove x dall'insieme se è presente
remove(Item x)
%Restituisce un nuovo insieme che è l'unione di A e B
Set union(Set A, Set B)
%Restituisce un nuovo insieme che è l'intersezione di A e B
Set intersection(Set A, Set B)
%Restituisce un nuovo insieme che è la differenza tra A e B
Set Difference(Set A, Set B)

3.3 Dizionari

Un dizionario è una struttura dati che rappresenta il concetto matematico di relazione univoca $R : D \to C$ o associazione chiave-valore. L'insieme D è il dominio ed è costituito dalle chiavi, l'insieme C è il codominio ed è costituito dai valori.

Operazioni ammesse

- Ottenere il valore associato ad una particolare chiave se presente altrimenti **nil**.
- Inserire una nuova associazione chiave-valore cancellando eventualmente associazioni precedenti per la stessa chiave.
- Rimuovere un'associazione chiave-valore esistente.

Specifica

: Dictionary

```
%Restituisce il valore associato alla chiave k se presente, nil
  altrimenti
Item lookup(Item k)
%Associa il valore v alla chiave k
insert(Item k, Item v)
%Rimuove l'associazione della chiave k
remove(Item k)
```

3.4 Alberi e grafi

3.4.1 Alberi ordinati

Un albero ordinato è dato da un insieme finito di elementi detti nodi, uno dei quali è designato come radice e i rimanenti, se esistono sono partizionati in insiemi ordinati e disgiunti, anch'essi alberi ordinati.

3.4.2 Grafi

La struttura di un grafo è composta da un insieme di elementi detti nodi o vertici e un insieme di coppie (ordinate o no) di nodi detti archi.

3.4.3 Operazioni

Tutte le operazioni su grafi e alberi ruotano intorno alla possibilità di effettuare visite su di essi.

3.5 Lista

Una lista o linked list è una struttura dati contenente una sequenza di nodi contenente dati arbitrari, $1-2$ puntatori all'elemento successivo e/o a quello precedente. La contiguità nella lista è diversa dalla contiguità in memoria e tutte le operazioni su una lista hanno costo $O(1)$.

Possibili implementazioni

Bidirezionale o monodirezionale: in base al numero di puntatori e se puntano a quello successivo e/o a quello precedente. Con sentinella o senza sentinella: in base alla presenza di una sentinella, una struttura che punta sempre al primo elemento della lista. Circolare o non circolare: si dice circolare se il puntatore dell'ultimo elemento punta al primo invece di essere **nil**.

Lista bidirezionale con sentinella

: List

```
List pred %Predecessore
List succ %Successore
Item value %Valore
List List()
    List t=new List
    t.pred = t
    t.succ = t
    return t
boolean isEmpty()
    return pred = succ = this
Pos head()
    return succ
Pos tail()
    return pred
Pos next(Pos p)
    return p.succ
Pos prev(Pos p)
    return p.pred
boolean finished(Pos p)
    return (p = this)
Item read(Pos p)
    return p.value
Item write(Pos p, Item v)
    p.value = v
Pos insert(Pos p, Item v)
    List t = List()
    t.value = v
    t.pred = p.pred
    p.pred.succ = t
    t.succ = p
    p.pred = t
    return t
Pos remove(Pos p)
    p.pred.succ = p.succ
    p.succ.pred = p.pred
    List t = p.succ
    delete p
    return t
```

3.6 Pila

Una pila o stack è una struttura dati lineare e dinamica in cui l'elemento rimosso dall'operazione di cancellazione è determinato, ovvero quello che per meno tempo è rimasto nell'insieme (LIFO: last-in, first-out). Possibili implementazioni sono una lista bidirezionale con un puntatore all'elemento top o tramite vettore, con dimensione limitata ma overhead più basso.

Specifica

: Stack

```
%Restituisce True se la pila è vuota
boolean isEmpty()
%Inserisce v in cima alla pila
push(Item v)
%Estrae l'elemento in cima alla pila e lo restituisce
Item pop()
%Legge l'elemento in cima alla pila
Item top()
```

Lista basata su vettore

: Stack

```
Item [] A %Elementi
int n %Cursore
int m %Dimensione massima
Stack Stack(int dim)
    Stack t = new Stack
    t.A = new int [1...dim]
    t.m = dim
    t.n = 0
    return t
Item top()
    precondition: n > 0
    return A[n]
boolean isEmpty()
    return n=0
Item pop()
    precondition: n > 0
    Item t = A[n]
    n = n - 1
    return t
push(Item v)
    precondition: n < m
    n = n + 1
    A[n] = v
```

3.7 Coda

Una coda è una struttura dati lineare e dinamica in cui l'elemento rimosso dall'operazione di cancellazione è determinato, ovvero quello che per più tempo

è rimasto nell'insieme (FIFO: first-in, first-out, questo tipo di politica si dice fair). Si può implementare attraverso liste monodirezionali con il puntatore head per l'estrazione e il puntatore tail per l'inserimento o tramite array circolari, con dimensione limitata ma overhead più basso. Per i secondi la circolarità è ottenuta attraverso l'operazione modulo e si deve prestare attenzione ai problemi di overflow.

Specifica

: Queue

```
%Restituisce True se la coda è vuota
boolean isEmpty()
%Inserisce v in fondo alla coda
enqueue(Item v)
%Estrae l'elemento in testa alla coda e lo restituisce
Item dequeue()
%Legge l'elemento in testa alla coda
Item top()
```

Coda basata su vettore circolare

: Queue

```
Item [] A %Elementi
int n %Dimensione attuale
int testa %Testa
int m %Dimensione massima
Queue Queue(int dim)
    Queue t = new Queue
    t.A = new int [1 ... dim - 1]
    t.m = dim
    t.testa = 0
    t.n = 0
    return t
Item top()
    precondition: n > 0
    return A[testa]
boolean isEmpty()
    return n=0
Item dequeue()
    precondition: n > 0
    Item t = A[testa]
    testa = (testa + 1) mod m
    n = n - 1
    return t
enqueue(Item v)
    precondition: n < m
    A[(testa + n) mod m] = v
    n = n + 1
```

Capitolo 4

Alberi

4.1 Alberi radicati

Un albero radicato può essere definito in due modi.

4.1.1 Definizioni

Definizione chiusa

Un albero radicato consiste in una serie di nodi e un insieme di archi orientati che connettono coppie di nodi con le seguenti proprietà:

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

Definizione ricorsiva

Un albero radicato è dato da:

- Un insieme vuoto.
- Un nodo radice e zero o più sottoalberi ognuno dei quali è un albero, la radice è connessa alla radice di ogni sottoalbero con un arco orientato.

4.1.2 Terminologia

- Il nodo senza archi entranti è detto radice (root).
- Per generare i sottoalberi (subtrees) da un albero si elimina la radice e tutti i suoi archi uscenti.

- Nodi con lo stesso genitore sono detti fratelli (siblings).
- Considerando un arco, il nodo da cui parte è detto genitore (parent) del nodo in cui arriva, detto figlio (child).
- I nodi senza archi uscenti sono detti foglie (leaves).
- I nodi nè foglie nè radice sono detti interni (internal nodes).
- La lunghezza del cammino semplice dalla radice ad un nodo, misurato nel numero di archi è detto profondità (depth) del nodo.
- I nodi alla stessa profondità formano un insieme chiamato livello (level).
- La profondità massima dell'albero si dice altezza (height).

4.2 Visite di alberi

La visita di un albero o ricerca è una strategia per visitare tutti i nodi di un albero. Il costo computazionale di una visita su un albero su n nodi è $\Theta(n)$ in quanto ogni nodo viene visitato un'unica volta.

4.2.1 Visita in profondità

La visita in profondità o depth-first-search (dfs) si compie visitando ricorsivamente tutti i sottoalberi dell'albero, richiede uno stack ed esiste in tre varianti: pre, in e post order.

4.2.2 Visita in ampiezza

La visita in ampiezza o breadth-fisrt-search (bfs) visita completamente ogni livello prima di passare al successivo partendo dalla radice. Richiede una queue.

4.3 Albero binario

Un albero binario è un albero radicato in cui ogni nodo ha al massimo due figli, indicati con figlio destro e sinistro. Due alberi binari differiscono anche se uno stesso nodo è designato come figlio sinistro invece che destro o viceversa.

4.3.1 Specifica

: Tree

```
%Costruisce 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 o nil se è il nodo radice
Tree Parent()
%Restituisce il figlio sinistro (destro) di questo nodo o nil se assente
Tree left()
Tree right()
%Inserisce il sottoalbero 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()
```

4.3.2 Memorizzazione e implementazione

Ogni nodo deve memorizzare oltre al proprio valore la reference al nodo padre (parent), la reference ai figli sinistro (left) e destro (right).

Implementazione

: Binary tree

```
Tree(Item v)
    Tree t = new Tree
    t.parent = nil
    t.left = t.right = nil
    t.value = v
    return t
insertLeft(Tree T)
    if left == nil then
        T.parent = this
        left = T
insertRight(Tree T)
    if right == nil then
        T.parent = this
        right = T
deleteLeft()
    if left ≠ nil then
        left.deleteLeft()
        left.deleteRight()
        left = nil
deleteRight()
    if right ≠ nil then
        right.deleteLeft()
        right.deleteRight()
        right = nil
```

4.3.3 Visita in profondità

```
: dfs(Tree t)
  if t ≠ nil then
      %visita in pre-ordine
      print t
      dfs(t.left)
      %visita in in-ordine
      print t
      dfs(t.right)
      %visita in post-ordine
      print t
```

4.4 Alberi generici

4.4.1 Specifica

```
: Tree
  %Costruisce 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 o nil se è il nodo radice
  Tree Parent()
  %Restituisce il primo figlio o nil se è un nodo foglia
  Tree leftmostchild()
  %Restituisce il prossimo fratello o nil se assente
  Tree rightSibling()
  %Inserisce il sottoalbero t come primo figlio di questo nodo
  insertChild(Tree t)
  %Inserisce il sottoalbero t come prossimo fratello di questo nodo
  insertSibling(Tree t)
  %Distrugge l'albero radicato nel primo figlio
  deleteChild()
  %Distrugge l'albero radicato nel prossimo fratello
  deleteSibling()
```

Depth-first search

```
: dfs(Tree t)
if t ≠ nil then
    %visita in pre-ordine
    print t
    Tree u = t.leftMostChild()
    while u ≠ nil do
        dfs(u)
        u = u.rightSibling()
    %visita in post-ordine
    print t
```

Breadth-first search

```
: bfs (Tree t)
Queue Q = Queue()
Q.enqueue(t)
while not Q.isEmpty() do
    Tree u = Q.dequeue()
    %Visita per livelli nodo u
    print u
    u = u.leftMostChild()
    while u ≠ nil do
        Q.enqueue(u)
        u = u.rightSibling()
```

4.4.2 Memorizzazione

Esistono vari modi per salvare un albero, scelti in base al numero massimo e medio dei figli presenti.

Vettore dei figli

Nel nodo viene memorizzato il genitore e un vettore contenente i figli, che a seconda del loro numero può portare a uno spreco dello spazio.

Primo figlio, prossimo fratello

```
: Tree

Tree parent %Reference al padre
Tree child %Reference al primo figlio
Tree sibling %Reference al prossimo fratello
Item value %Valore memorizzato nel nodo
Tree(Item v)
    Tree t = new Tree
    t.value = v
    t.parent = t.child = t.sibling = nil
    return t
insertChild(Tree t)
    t.parent = self
    t.sibling = child
    child = t
insertSibling(Tree t)
    t.parent = parent
    t.sibling = sibling
    sibling = t
deleteChild()
    Tree newChild = child.rightSibling()
    delete(child)
    child = newChild
deleteSibling()
    Tree newBrother = sibling.rightSibling()
    delete(sibling)
    sibling = newBrother
delete(Tree t)
    Tree u = t.leftMostChild()
    while u ≠ nil do
        Tree next = u.rightSibling()
        delete(u)
        u = next
```

Vettore dei padri

L'albero è rappresentato da un vettore i cui elementi contengono il valore associato al nodo e l'indice della posizione del padre nel vettore.

Capitolo 5

Alberi di ricerca

Si vuole portare la ricerca binaria negli alberi in modo da implementare un dizionario.

- Le associazioni chiave-valore vengono memorizzate in un albero binario.
- Ogni nodo u contiene la coppia $u.key$ e $u.value$.
- Le chiavi devono appartenere ad un insieme totalmente ordinato.

Le chiavi contenute nel sottoalbero sinistro di u sono minori di $u.key$ e quelle contenute nel sottoalbero destro maggiori. In questo modo è possibile realizzare un algoritmo di ricerca dicotomica.

5.1 Specifica

: Search Binary Tree	
Tree parent	`remove`(Item k)
Tree left	%Ordinamento
Tree right	Tree `successorNode`(Tree t)
Item value	Tree `predecessorNode`(Tree t)
Item key	Tree `min`()
%Getters	Tree `max`()
Item `key`()	%Funzioni interne
Item `value`()	**private**: Tree `lookupNode`(Tree t, Item k)
Tree `parent`()	**private**: Tree `insertNode`(Tree t, Item k, Item v)
Tree `right`()	**private**: Tree `removeNode`(Tree t, Item k)
Tree `left`()	
%Dizionario	
Item `lookup`(Item k)	
`insert`(Item k, Item v)	

: Dictionary
Tree tree `Dictionary()`   tree = **nil**

5.1.1 Ricerca

La funzione *lookUpNode(Tree t, Item k)* restituisce il nodo dell'albero t che contiene la chiave k o **nil** se non presente.

Implementazione nel dizionario

: Item `lookup`(Item k)

Tree t = `lookupNode`(*tree, k*)
if $t \neq$ **nil then**
  return t.`value`
else
  return nil

Implementazione

Iterativa

: Tree `lookupNode` (Tree t, Item k)

Tree u = t
while $u \neq$ **nil and** u.`key` $\neq k$
do
  if $k < u$.`key` **then**
  u = u.`left`
  else
  u = u.`right`
return u

Ricorsiva

: Tree `lookupNode` (Tree t, Item k)

if $t ==$ **nil or** t.`key` $== k$
then
  return t
else if $k < t$.`key` **then**
  return
  `lookupNode`(t.`left`, k)
else
  return
  `lookupNode`(t.`right`, k)

5.1.2 Minimo e massimo

Implementazione

```
: Tree min (Tree t)
Tree u = t
while u.left() ≠ nil do
    u = u.left
return u
```

```
: Tree max (Tree t)
Tree u = t
while u.right ≠ nil do
    u = u.right
return u
```

5.1.3 Successore e predecessore

Si definisce successore di un nodo u il più piccolo nodo maggiore di u.

Implementazione

```
: successorNode(Tree t)
if t = nil then
    return t
if t.left ≠ nil then
    return max(t.left)
else
    Tree p = t.parent()
    while p ≠ nil and t = p.left do
        t = p
        p = p.parent
    return p
```

```
: predecessorNode(Tree t)
if t = nil then
    return t
if t.right() ≠ nil then
    return min(t.right)
else
    Tree p = t.Parent
    while p ≠ nil and t = p.right do
        t = p
        p = p.Parent
    return p
```

5.1.4 Inserimento

L'operazione *insertNode(Tree t, Item k, Item v)* inserisce un'associazione chiave-valore *(k, v)* nell'albero t. Se la chiave è già presente sostiutuisce il valore associato, altrimenti viene inserita una nuova associazione. Se $T =$ ***nil*** restituisce il primo nodo dell'albero, altrimenti restituisce t inalterato.

Implementazione dizionario

```
: insert(Item k, Item v)
tree = insertNode(tree, k, v)
```

Implementazione

```
: Tree insertNode(Tree t, Item k, Item v)
Tree p = nil
Tree u = t
while u ≠ nil and u.key ≠ k do
    p = u
    if k < u.key then
        u = u.left
    else
        u = u.right
if u ≠ nil and u.key == k then
    u.value = v
else
    Tree new = Tree(k, v)
    link(p, new, k)
    if p = nil then
        t = new
    return t
```

```
: 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
```

5.1.5 Cancellazione

L'operazione *Tree removeNode(Tree t, Item k)* rimuove il nodo contenente la chiave k dall'albero t e restituisce la radice dell'albero possibilmente cambiata.

Implementazione dizionario

```
: remove(Item k)
tree = removeNode(tree, k)
```

Implementazione

```
: Tree removeNode(Tree T, Item k)
Tree t
Tree u = lookupNode(T, k)
if u ≠ nil then
    if u.left == nil and u.right == nil then
        link(u.parent, nil, k)
        delete u
    else if u.left ≠ nil and u.right ≠ nil then
        Tree s = successorNode()
        link(s.parent, s.right, s.key)
        u.key() = s.key()
        u.value() = s.value()
        delete s
    else if u.left ≠ nil and u.right == nil then
        link(u.parent, s.left, k)
        if u.parent = nil then
            T = u.left
    else
        link(u.parent, s.right, k)
        if u.parent = nil then
            T = u.right
return T
```

Dimostrazione correttezza

- Caso 1: Eliminare foglie non cambia l'ordine dei nodi rimanenti.
- Caso 2: Se u è il figlio destro (sinistro) di p, tutti i valori nel sottoalbero di f sono maggiori (minori) di p, pertanto f può essere attaccato come figlio destro (sinistro) di p al posto di u.
- Caso 3: il successore s è sicuramente $\geq$ dei nodi del sottoalbero sinistro di u e sicuramente $\leq$ dei nodi del sottoalbero destro di u, pertanto può essere sostituito a u e a quel punto si ricade nel caso 2.

5.2 Costo computazionale

Tutte le operazioni sono confinate ai nodi posizionati lungo un cammino semplice dalla radice ad una foglia, pertanto detta h altezza di un albero, avranno complessità $O(h)$. Il caso pessimo si ha nel caso in cui l'altezza h sia uguale a n ($O(n)$), il caso ottimo quando $h = \log n$ ($O(\log n)$).

5.3 Alberi di ricerca bilanciati

Per mantenere un grado di complessità il più vicino possibile a $O(\log n)$ si utilizzano tecniche di bilanciamento per tenere sotto controllo l'altezza di un albero. Si introduce un fattore di bilanciamento $\beta(v)$, ovvero la massima differenza di altezza fra i sottoalberi di v.

- Alberi AVL: $\beta(v) \leq 1$ per ogni nodo v, bilanciamento ottenuto tramite rotazioni.
- B-Alberi: $\beta(v) = 0$ per ogni nodo v, specializzati per strutture in memoria secondaria.
- Alberi 2-3: $\beta(v) = 0$ per ogni nodo v, bilanciamento ottenuto tramite merge/split, grado variabile.

Rotazioni

Rotazione sinistra Si prende un nodo di un albero, si fa diventare suo figlio destro il figlio sinistro del figlio destro, successivamente si pone il nodo iniziale come figlio sinistro del vecchio nodo destro e il vecchio genitore del nodo iniziale diventa il genitore del vecchio nodo destro.

Rotazione destra Si prende un nodo di un albero, si fa diventare suo figlio sinistro il figlio destro del suo figlio sinistro, successivamente si pone il nodo iniziale come figlio destro del vecchio nodo sinistro e il vecchio genitore del nodo iniziale diventa il genitore del vecchio nodo sinistro.

```
: Tree rotateLeft(Tree x)
    Tree y ← x.right
    Tree p ← x.parent
    x.right ← y.left
    if y.left ≠ nil then
        y.left.parent ← x
    y.left ← x
    x.parent ← y
    y.parent ← p
    if p ≠ nil then
        if p.left == x then
            p.left ← y
        else
            p.right ← y
    return y
```

```
: Tree rotateRight(Tree x)
    Tree y ← x.left
    Tree p ← x.parent
    x.right ← y.right
    if y.right ≠ nil then
        y.right.parent ← x
    y.right ← x
    x.parent ← y
    y.parent ← p
    if p ≠ nil then
        if p.left == x then
            p.left ← y
        else
            p.right ← y
    return y
```

5.3.1 Alberi red-black

Un albero red-black è un albero binario di ricerca in cui ogni nodo è colorato di rosso o nero, le chiavi vengono salvate solo nei nodi interni all'albero e le foglie sono costituiti da nodi speciali **Nil**. Un albero red-black è costruito in modo che rispetti questi vincoli:

- La radice è nera.
- Tutte le foglie sono nere.
- Entrambi i figli di un nodo rosso sono neri.
- Tutti i cammini semplici da un nodo u a una delle foglie contenute nel sottoalbero radicato in u hanno lo stesso numero di nodi neri.

Memorizzazione

```
: Tree
Tree parent;
Tree left;
Tree right;
int color;
Item key;
Item value;
```

I nodi **Nil** sono nodi sentinella il cui scopo è evitare di trattare diversamente i puntatori ai nodi dai puntatori **nil**. Al posto di un puntatore **nil** si utilizza un puntatore ad un nodo speciale **Nil**. Ne esiste solo uno per risparmiare memoria e un nodo con figli **Nil** corrisponde ad una foglia nell'albero binario di ricerca.

Altezza nera

L'altezza nera $b(v)$ di un nodo v è il numero di nodi neri lungo ogni percorso da v escluso ad ogni foglia inclusa del sottoalbero. L'altezza nera di un albero red-black è pari all'altezza nera della sua radice. Quando esistono più colorazioni che rispettano i limiti possono esistere diverse altezze nere per lo stesso albero.

Inserimento

Quando si vuole inserire un nodo in un albero red-black si ricerca la posizione usando la stessa procedura per gli alberi di ricerca e si colora il nuovo nodo di rosso. Si possono pertanto violare dei vincoli in questo modo e si rende necessario aggiungere nella funzione *insertNode(Tree T, Item k, Item v)* un processo di bilanciamento (eseguito successivamente alla chiamata della funzione *link*). Il principio generale per il bilanciamento consiste nel spostarsi verso l'alto lungo il percorso di inserimento, ripristinare il vincolo dei figli neri di un nodo rosso spostando le violazioni verso l'alto mantenendo l'altezza nera dell'albero e colorando la radice di nero alla fine. Queste operazioni sono necessarie unicamente quando due nodi consecutivi sono rossi.

balanceInsert(Tree t) I nodi coinvolti sono il nodo inserito t, suo padre p, suo nonno n e suo zio z. E si possono verificare sette casi diversi in cui avviene una violazione.

- Caso 1: il nuovo nodo t non ha padre: è il primo nodo ad essere inserito o si è risaliti fino alla radice, si colora t di nero.
- Caso 2: il padre p di t è nero: non si viola nessun vincolo.
- Caso 3: t rosso, p rosso, z rosso: se z è rosso si possono colorare di nero p, z e di rosso n. Poichè tutti i cammini che passano per z e p passano per n l'altezza nera non è cambiata. Il problema ora potrebbe sussistere sul nonno, si pone pertanto $t = n$ e il ciclo continua.
- Caso 4a (4b): t rosso, p rosso, z nero: si assuma che t sia figlio destro (sinistro) di p e p figlio sinistro (destro) di n. Una rotazione a sinistra (destra) a partire dal nodo p scambia i ruoli di t e p ottenendo il caso 5a (5b) essendo entrambi i nodi coinvolti nel cambiamento rossi, l'altezza nera non cambia.
- Caso 5a (5b): t rosso, p rosso, z nero: si assuma che t sia figlio sinistro (destro) di p e p figlio sinistro (destro) di n. Una rotazione a destra su n porta ad una situazione in cui t e n sono figli di p. Colorando n di rosso e p di nero ci si ritrova in una situazione in cui tutti i vincoli sono rispettati.

```
: balancedInsert(Tree t)
    t.color() ← RED
    while t ≠ nil do
        Tree p ← t.parent %padre
        Tree n %nonno
        Tree z %zio
        if p ≠ nil then
            n ← p.parent
        else
            n ← nil
        if n == nil then
            z ← nil
        else
            if n.left == p then
                z ← p.right
            else
                z ← p.left
        if p == nil then
            %Caso 1
            t.color ← BLACK
            t ← nil
        else if p.color == BLACK then
            %Caso 2
            t ← nil
        else if z.color == RED then
            %Caso 3
            p.color ← z.color ← BLACK
            n.color ← RED
            t ← n
        else
            if t == p.right and p == n.left then
                %Caso 4a
                rotateLeft(p)
                t ← p
            else if t == p.left and p == n.right then
                %Caso 4b
                rotateRight(p)
                t ← p
            else
                if t == p.left and p == n.left then
                    %Caso 5a
                    rotateRight(n)
                else if t == p.right and p == n.right then
                    %Caso 5b
                    rotateLeft(n)
                p.color ← BLACK
                n.color ← RED
                t ← nil
```

Complessità La complessità totale per un inserimento è $O(\log n)$, con tre passaggi: $O(\log n)$ per scendere fino al punto di inserimento, $O(1)$ per effettuare l'inserimento e $O(\log n)$ per risalire e sistemare le violazioni. È possibile implementare un inserimento top-down che aggiusta l'albero mano a mano che scende fino al punto di inserimento.

Altezza albero red-black

Teorema In un albero red-black un sottoalbero di radice u contiene almeno $n \geq 2^{bh(u)} - 1$ nodi interni.

Dimostrazione Si dimostra per induzione sull'altezza (non sull'altezza nera).

- Caso base $h = 0$: u è una foglia **nil** e il sottoalbero con radice in u contiene $n \geq 2^{bh(u)} - 1 = 2^0 - 1 = 0$ nodi interni.
- Passo induttivo $h > 1$: allora u è un nodo interno con due figli tali che ogni figlio v ha un'altezza nera $bh(v)$ pari a $bh(u)$ se rosso o a $bh(u) - 1$ se nero. Per ipotesi induttiva ogni figlio ha almeno $2^{bh(u)} - 1$ nodi interni. Pertanto il sottoalbero con radice in u ha almeno $n \geq 2^{bh(u)-1} - 1 + 2^{bh(u)-1} - 1 + 1 = 2^{bh(u)-1} - 1$ nodi.

Teorema In un albero red-black almeno la metà dei nodi dalla radice ad una foglia deve essere nera.

Dimostrazione Per il secondo vincolo se un nodo è rosso, entrambi i suoi figli devono essere neri, pertanto la situazione in cui sono presenti il maggior numero di nodi rossi è il caso in cui rossi e neri sono alternati, dimostrando il teorema.

Teorema In un albero red-black nessun percorso da un nodo v ad una foglia è lungo più del doppio del percorso da v ad un'altra foglia.

Dimostrazione Per definizione ogni percorso da un nodo ad una qualsiasi foglia contiene lo stesso numero di nodi neri. Dal lemma precedente almeno la metà di questi nodi sono neri, pertanto al limite uno dei due percorsi è costituito da soli nodi neri mentre l'altro è costituito da nodi neri e rossi alternati.

Teorema L'altezza massima di un albero red-black contenente n nodi interni è al più $2 \log(n + 1)$.

Dimostrazione

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

Cancellazione

L'algoritmo di cancellazione per gli alberi red-black è costruito su quello di cancellazione per gli alberi generici. Dopo la cancellazione è necessario decidere se ribalanciare o meno. Le operazioni di ripristino sono necessarie solo dopo la cancellazione di un nodo nero in quando se il nodo cancellato è rosso l'altezza nera rimane invariata, non sono creati nodi rossi consecutivi e la radice resta nera. Se il nodo cancellato è nero si possono rompere i vincoli uno, tre e quattro. L'algoritmo seguente ripristina i vincoli con rotazioni e cambiamenti di colore. Ci sono quattro casi possibili con i corrispettivi simmetrici.

Complessità La cancellazione è concettualmente complicata ma efficiente in quanto dal caso 1 si passa al 2, 3 o 4, dal caso 2 si passa agli altri risalendo l'albero, dal caso 3 si passa al caso 4 e al caso 4 termina. È possibile visitare un massimo di $O(\log n)$ di casi, ognuno dei quali risolti in $O(1)$.

```
: balancedDelete(Tree T, Tree t)
while T ≠ t and t.color == BLACK do
    Tree p = t.parent %Padre
    if t == p.left then
        Tree f = p.right %Fratello
        Tree ns = f.left %Nipote sinistro
        Tree nd = f.right %Nipote destro
        if f.color == RED then
            %Caso 1a
            p.color = RED
            f.color = BLACK
            rotateLeft(p)
            %t viene lasciato inalterato,
              pertanto si ricade nei casi 2,
              3 o 4
        else
            if ns.color == nd.color ==
              BLACK then
                %Caso 2a
                f.color = RED
                t = p
            else if ns.color == RED and
              nd.color == BLACK then
                %Caso 3a
                ns.color = BLACK
                f.color = RED
                rotateRight(f)
                %t viene lasciato
                  inalterato, pertanto si
                  ricade nel caso 4
            else if nd.color == RED
              then
                %Caso 4a
                f.color = p.color()
                p.color = BLACK
                nd.color = BLACK
                rotateLeft(p)
                t = T
    else
        Tree f = p.left %Fratello
        Tree ns = f.left %Nipote sinistro
        Tree nd = f.right %Nipote destro
        if f.color == RED then
            %Caso 1b
            p.color = RED
            f.color = BLACK
            rotateRight(p)
            %t viene lasciato inalterato,
              pertanto si ricade nei casi 2,
              3 o 4
        else
            if ns.color == nd.color ==
              BLACK then
                %Caso 2b
                f.color = RED
                t = p
            else if nd.color == RED and
              ns.color == BLACK then
                %Caso 3b
                nd.color = BLACK
                f.color = RED
                rotateLeft(f)
                %t viene lasciato
                  inalterato, pertanto si
                  ricade nel caso 4
            else if ns.color == RED
              then
                %Caso 4a
                f.color = p.color()
                p.color = BLACK
                ns.color = BLACK
                rotateRight(p)
                t = T
```

Capitolo 6

Hashing

Le tabelle hash sono l'implementazione ideale per dizionari, insiemi dinamici di coppie chiave-valore indicizzati sulla chiave. Si sceglie una funzione hash H che mappa ogni chiave $k \in \mathcal{U}$ in un intero $H(k)$. La coppia chiave-valore viene memorizzata in un vettore nella posizione $H(k)$ che viene detto tabella hash.

6.0.1 Definizioni

- L'insieme delle possibili chiavi è rapprentato dall'insieme universo $\mathcal{U}$ di dimensione u.
- Il vettore $T[0 \dots m-1]$ ha dimensione m.
- Una funzione hash è definita: $H : \mathcal{U} \to \{0, \dots, m-1\}$.
- Quando due o più chiavi nel dizionario hanno lo stesso valore di hash avviene una collisione, idealmente non dovrebbero avvenire.

Tabelle ad accesso diretto

Si utilizzano le tabelle ad accesso diretto nel caso particolare in cui $\mathcal{U} \subset \mathbb{Z}^+$. Si utilizza come funzione hash l'identità $H(k) = k$ e $m = |\mathcal{U}|$. Presenta dei problemi quando u è molto grande e uno spreco di memoria quando u non è grande ma il numero di chiavi effettivamente registrate in memoria è molto minore di m.

Funzioni hash perfette

Una funzione hash si dice perfetta se è iniettiva, ovvero $\forall k_1, k_2 \in \mathcal{U}, k_1 \neq k_2 \Rightarrow H(k_1) \neq H(k_2)$. Ci sono dei problemi in quanto lo spazio delle chiavi è spesso grande, sparso e non conosciuto ed è pertanto spesso impraticabile ottenere una funzione di hash perfetta.

6.1 Funzioni hash

Se non è possibile eliminare le collisioni si cerca per lo meno di minimizzare il loro numero, cercando funzioni che distribuiscano le chiavi uniformemente negli indici $[0, \ldots, m-1]$ della tabella hash.

Uniformità semplice

Sia $P(k)$ la probabilità che una chiave sia inserita nella tabella. Sia $Q(i)$ la probabilità che una chiave finisca nella cella i: $Q(i) = \sum_{k \in \mathcal{U}: h(k)=i} P(k)$. Una funzione hash gode dell'uniformità semplice se: $\forall i \in [0, \ldots, m-1] : Q(i) = \frac{1}{m}$.

6.1.1 Come realizzare una funzione hash

Per realizzare una funzione hash con uniformità semplice è necessario che la distribuzione P sia nota, cosa non completamente possibile nella realtà. Si utilizzano pertanto tecniche euristiche. Pertanto si assuma che ogni chiave può essere tradotta in numeri interi non negativi anche interpretando la loro rappresentazione in memoria come un numero. Si intenda pertanto con *bin(k)* la rappresentazione binaria della chiave.

Estrazione

Si consideri $m = 2^p$ e $H(k) = int(b)$, dove b è un sottoinsieme di p bit presi da *bin(k)*. Presenta dei problemi in quanto selezionare bit presi dal suffisso della chiave può generare collisioni con alta probabilità (o anche in generale).

XOR

Si consideri $m = 2^p$ e $H(k) = int(b)$ dove b è dato dalla somma modulo 2 effettuata bit a bit di sottoinsiemi di p bit di *bin(k)*. Presenta dei problemi in quanto le permutazioni possono generare lo stesso valore di hash.

Metodo della divizione

Si consideri m numero dispari, meglio se primo. $H(k) = int(k) \mod m$. Non vanno bene $m = 2^p$ in quanto considera unicamente i p bit meno significativi e $m = 2^p - 1$ in quanto permutazioni con set di elementi con dimensione 2^p hanno lo stesso valore di hash. Si devono pertanto scegliere numeri primi distanti da potenza di 2 e 10.

Metodo della moltiplicazione

Si consideri un m qualsiasi, meglio se potenza di 2, $C \in]0; 1[$ una costante reale e $i = int(k)$. $H(k) = \lfloor m(C \cdot i - \lfloor C \cdot i \rfloor) \rfloor$.

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

Reality check

Il metodo della moltiplicazione non fornisce hashing uniforme. Esistono test per valutare la bontà di una funzione hash: Avalanche effect che se si cambia un bit nella chiave, deve cambiare almeno la metà dei bit del valore hash e test statistici con il chi-quadro. Si devono implementare inoltre funzioni hash crittografiche.

6.2 Le collisioni

Per gestire le collisioni si rende necessario trovare posizioni alternative per le chiavi e se la chiave non si trova nella posizione attesa la si deve cercare nelle posizioni alternative. Questa ricerca deve essere $O(1)$ nel caso medio e $O(n)$ nel caso pessimo. Presentano problemi in quanto sono strutture dati complesse vista la presenza di liste e puntatori.

6.2.1 Liste o vettori di trabocco (Concatenamento o chaining)

Le chiavi con lo stesso valore di hash vengono memorizzate in una lista monodirezionale o vettore dinamico. Si memorizza un puntatore alla testa della lista nello slot $H(k)$-esimo della tabella hash. Le operazioni sono di inserimento in testa *insert*, di ricerca *lookup* o rimozione *remove* che implicano una scansione della tabella per cercare la chiave.

Analisi complessità

Si considerino i seguenti valori:

n	Numero di chiavi memorizzate nella tabella hash
m	Capacità della tabella hash
$\alpha = \frac{n}{m}$	fattore di carico
$I(\alpha)$	Ricerca con insuccesso o numero medio di accessi alla tabella per una chiave non presente
$S(\alpha)$	Ricerca con successo o numero medio di accessi alla tabella per una chiave presente

Caso pessimo Tutte le chiavi sono collocate in un'unica lista: *insert* $\Theta(1)$, *lookup(), remove()* $\Theta(n)$.

Analisi del caso medio Il caso medio dipende dalla distribuzione delle chiavi, si assuma pertanto hashing uniforme semplice e costo di calcolo della funzione di hashing $\Theta(1)$. Il valore atteso della lunghezza della lista è di $\alpha = \frac{n}{m}$. Una ricerca senza successo visita tutte le chiavi nella lista corrispondente, pertanto ha costo atteso $\Theta(1) + \alpha$, mentre una ricerca con successo visita in media la metà delle chiavi nella lista corrispondente, pertanto $\Theta(1) + \frac{\alpha}{2}$. Il fattore di carico influenza il costo computazionale delle operazioni sulla tabella hash e se $n = O(m)$, $\alpha = O(1)$, pertanto tutte le operazioni sono $O(1)$.

6.2.2 Indirizzamento aperto

Nell'indirizzamento aperto tutte le chiavi vengono memorizzate nella tabella stessa e ogni slot contiene una chiave oppure un valore **nil**. Nell'inserimento se uno slot è utilizzato se ne cerca uno alternativo. Nella ricerca si cerca nello slot prescelto e poi negli slot alternativi fino a che si trova la chiave oppure un nodo **nil**.

Definizioni

- Un'ispezione è l'esame di uno slot durante la ricerca.
- La funzione di hash viene estesa: $H : \mathcal{U} \times [0, \ldots, m-1] \to [0, \ldots, m-1]$, dove il secondo insieme del dominio rappresenta il numero di ispezione metre il codominio l'indice del vettore.
- Sequenza di ispezione: una sequenza di ispezione $[H(k,0), H(k,1), \ldots, H(k,m-1)]$ è una permutazione degli indici corrisondente all'ordine in cui vengono visitati gli slot. Non si vogliono visitare gli slot più di una volta e potrebbe rendersi necessario visitare tutti gli slot della tabella.

Fattore di carico Il fattore di carico è compreso tra 0 e 1 e la tabella può andare in overflow.

Tecniche di ispezione

Hashing uniforme La situazione ideale prende il nome di hashing uniforme, in cui ogni chiave ha la stessa probabilità di avere come sequenza di ispezione una delle qualisasi $m!$ permutazioni degli indici.

Ispezione lineare La funzione di $H(k,i) = (H_1(k) + h \cdot i) \mod n$. La sequenza di ispezione è determinata dal primo elemento. Al massimo m sequenze di ispezione diverse sono possibili.

Agglomerazione primaria Causa lunghe sotto-sequenze occupate che tendono a diventare più lunghe: uno slot vuoto preceduto da i slot pieni viene riempito con probabilità $\frac{i+1}{m}$. I tempi medi di inserimento e cancellazione crescono.

Ispezione quadratica La funzione di $H(k,i) = (H_1(k)+h\cdot i^2) \mod n$. Dopo il primo elemento $H_1(k,0)$, le ispezioni successive hanno un offset che dipende da una funzione quadratica nel numero di ispezione i. La sequenza risultante non è una permutazione, al massimo m sequenze di ispezioni distinte sono possibili.

Agglomerazione secondaria Se due chiavi hanno la stessa ispezione iniziale le loro sequenze sono identiche.

Doppio hashing Funzione: $H(k,i) = (H_1(k) + iH_2(k)) \mod n$. Esistono pertanto due funzioni ausiliare, H_1 fornisce la prima ispezione, H_2 fornisce l'offset rispetto alla prima ispezione. Sono possibili al massimo m^2 sequenze di ispezione. Per garantire una permutazione completa $H_2(k)$ e m devono essere primi tra loro. Pertanto se si sceglie $m = 2^p$ $H_2(k)$ deve restituire numeri dispari. Se si sceglie m primo, $H_2(k)$ deve restituire numeri minori di m.

Cancellazione

Non si possono sostituire le chiavi da eliminare con **nil** in quanto potrebbe introdurre errori nella ricerca che terminerebbe troppo presto. Si utilizza pertanto un valore speciale **deleted** che vengono trattati come slot vuoti dal'inserimento e come slot pieni dalla ricerca. Il tempo di ricerca non dipende più da α ma il concatenamento diventa più comune.

Implementazione dell'hashing doppio

```
: Hash
    Item [] K %Tabella delle chiavi
    Item [] V %Tabella dei valori
    int m %Dimensione della tabella
    Hash Hash(int dim)
        Hash t = new Hash
        t.m = dim
        t.K = new Item [0, ..., dim − 1] t.V = new Item [0, ..., dim − 1]
         for i = 0 to dim-1 do
            t.K[i] = nil
        return t
    int scan(Item k, boolean insert)
        int c = m
        int i = 0
        int j = H(k)
        while K[j] ≠ k and K[j] ≠ nil and i < m do
            if K[j] == deleted and c == m then
                c = j
            j = (j + H′(k)) mod n
            i += 1
        if insert and K[j] ≠ k and c < m then
            j = c
        return j
    Item lookup(Item k)
        int i = scan(k, false)
        if K[i] == k then
            return V[i]
        else
            return nil
    insert(Item k, Item v)
        int i = scan(k, true)
        if K[i] == nil or K[i] == deleted or K[i] == k then
            K[i] = k
            V[i] = v
        else
            %Errore, tabella hash piena
    remove(Item k)
        int i = scan(k, false)
        if k[i] = k then
            K[i] = deleted
```

6.3 Complessità

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	$\alpha \geq 0$	$1+\alpha$	$1+\frac{\alpha}{2}$

Si noti come la complessità aumenti con l'aumentare di α. Pertanto sopra una soglia prefissata t_α, solitamente tra 0.5 e 0.75 si alloca una nuova dimensione $2m$ e si reinseriscono tutte le chiavi presenti nella tabella. Si dimezza pertanto il fattore di carico e si eliminano tutti gli elementi **deleted**. Nel caso pessimo c'è un costo di $O(m)$ per la ristrutturazione nel caso pessimo con costo ammortizzato costante.

Capitolo 7

Insiemi e dizionari

7.1 Insiemi

7.1.1 Insiemi realizzati con vettori booleani

Implementazione

: Set (vettore booleano)

```
boolean [] V
int size
int dim
Set Set(int m)
    Set t = new Set
    t.size = 0
    t.dim = =m
    t.V = [false ] * m
    return t
boolean contains(int x)
    if 1 ≤ x ≤ dim then
        return V[x]
    else
        return false
int size()
    return size
insert(int x)
    if 1 ≤ x ≤ dim then
        if not V[x] then
            size += 1
            V[x] = true
remove(int x)
    if 1 ≤ x ≤ dim then
        if V[x] then
            size == 1
            V[x] = false
```

```
Set union(Set A, Set B)
    Set C= Set(max(A.dim, B.dim))
    for i = 1 to A.dim do
        if A.contains(i) then
            C.insert(i)
    for i = 1 to B.dim do
        if B.contains(i) then
            C.insert(i)
    return C
Set intersection(Set A, Set B)
    Set C= Set(min(A.dim, B.dim))
    for i = 1 to min(A.dim, B.dim)
     do
        if A.contains(i) and
         B.contains(i) then
            C.insert(i)
    return C
Set difference(Set A, Set B)
    Set C= Set(A.dim)
    for i = 1 to A.dim do
        if A.contains(i) and not
         B.contains(i) then
            C.insert(i)
    return C
```

Caratteristiche

Gli insiemi possono essere memorizzati come un vettore di m elementi, se si vogliono salvare gli interi da 1 a m. Questo tipo di implementazione è molto semplice ed è efficiente verificare se un elemento appartiene ad un insieme. Come svantaggi presenta lo spreco di memoria, in quanto la memoria occupata è sempre $O(m)$ indipendentemente dagli elementi salvati. E alcune operazioni sono inefficienti in $O(m)$.

7.1.2 Insiemi realizzati con liste

Liste non ordinate

Operazioni di ricerca, inserimento e cancellazione $O(n)$, operazioni di inserimento assumendo assenza $O(1)$. Operazioni di unione intersezione, differenza $O(nm)$.

```
: Set difference(Set A, Set B)
Set C = Set()
foreach s ∈ A do
    if not B.contains(s) then
        C.insert(s)
    return C
```

Liste ordinate

Ricerca $O(n)$ per liste e $O(\log n)$ per i vettori, inserimento e cancellazione $O(n)$, unione, intersezione e differenza $O(n)$.

```
: List intersection(List A, List B)
List C = List()
Pos p = A.head()
Pos q = B.head()
while not A.finished(p) and not B.finished(q) do
    if A.read(p) = B.read(q) then
        C.insert(C.tail(), A.read(p))
        p = A.next(p)
        q = B.next(q)
    else if A.read(p) ¡ B.read(q) then
        p = A.next(p)
    else
        q = B.next(q)
return C
```

7.1.3 Strutture dati complesse

Alberi bilanciati

Si ottengono insiemi con ordinamento, con ricerca, inserimento, cancellazione $O(\log n)$, iterazione $O(n)$.

Tabelle hash

Si ottengono insiemi senza ordinamento, con ricerca, inserimento, cancellazione $O(1)$, iterazione $O(m)$.

7.2 Bloom filters

I bloom filters sono una via di mezzo tra gli insiemi realizzati con i vettori booleani e le tabelle hash, sono una struttura dati dinamica con bassa occupazione di memoria che non offre la possibilità di cancellazioni, nessuna memorizzazione e dà risposte probabilistiche.

7.2.1 Specifica

L'operazione di *insert(x)* inserisce l'elemento x nel bloom filter. ***booleans*** *contains(x)* se restituisce **false** l'elemento non sicuramente è presente, ma è possibile che ci siano dei falsi positivi. Si deve fare un trade-off tra occupazione di memoria e probabilità di un falso positivo. Indicata con ε la probabilità di un falso positivo, i bloom filters richiedono $1.44 \log_2(\frac{1}{\varepsilon})$ bit per elemento inserito.

7.2.2 Applicazioni

In chrome vengono utilizzati per indicare i siti con possibile malware, o in generale vengono utilizzati quando una verifica locale permette di evitare operazioni di I/O più costose. I falsi positivi inoltre possono essere utilizzati per mascherare un messaggio e garantire così un certo livello di privacy.

7.2.3 Implementazione

Vengono implementati attraverso un vettore booleando A di m bit inizializzato a **false** e k funzioni di hash $H_1, \ldots, H_k : U \to [0, \ldots, m-1]$.

```
: insert(x)
  for i = 1 to k do
      A[H_i(x)] = true
```

```
: boolean contains(x)
  for i = 1 to k do
      if A[H_i(x)] == false
       then
        return
      false
  return true
```

7.2.4 Caratterizzazione matematica

Dati n oggetti, m bit e k funzioni hash, la probabilità di un falso positivo è:

$$\varepsilon = (1 - e^{-k\frac{n}{m}})^k$$

Dati n oggetti e m bit il valore ottimale per k è:

$$k = \frac{m}{n} \ln 2$$

Dati n oggetti e una probabilità di falsi positivi ε il numero di bit m richiesti è pari a:

$$m = \frac{n \ln \varepsilon}{(\ln 2)^2}$$

Capitolo 8

Grafi

8.0.1 Definizioni

Grafo orientato Si dice grafo orientato la coppia $G = (V, E)$, dove V è l'insieme di nodi o vertici ed E è l'insieme di coppie ordinate (u, v) di nodi dette anche archi.

Grafo non orientato Si dice grafo non orientato la coppia $G = (V, E)$, dove V è l'insieme di nodi o vertici ed E è l'insieme di coppie non ordinate (u, v) di nodi dette anche archi.

- Un vertice v è detto adiacente a u se esiste un arco (v, u).
- Un arco (v, u) è detto incidente a v e u.
- In un grafo non orientato la relazione di adiacenza è simmetrica.
- $n = |V|$ numero di nodi.
- $m = |E|$ numero di archi.
- In un grafo non orientato $m \leq \frac{n(n-1)}{2} = O(n^2)$.
- In un grafo orientato $m \leq n^2 - n = O(n^2)$.
- La complessità di algoritmi sui grafi deve essere espressa sia un termini di n che di m.
- Un grafo con un arco tra tutte le coppie di nodi è detto completo.
- Un grafo si dice sparso se ha "pochi archi", ovvero con $m = O(n)$ o $m = O(n \log n)$.
- Un grafo si dice dento se ha "molti archi", ovvero con $m = \Omega(n^2)$.
- Un albero libero è un grafo connesso con $m = n - 1$.

- Un albero radicato è un albero libero sul quale è stata scelta una radice.
- Un insieme di alberi è un grafo detto foresta.
- Il grado di un nodo è il numero di archi incidenti su di esso.
- Nei grafi orientati si divide il grado in grado entrante per gli archi che incidono su di esso e in grado uscente per gli archi che incidono da esso.
- Si dice cammino C di lunghezza k in un grafo $G = (V, E)$ una sequenza di nodi $u_0, \ldots, u_k$ tali che $(u_i, u_{i+1}) \in E$ per $0 \leq i \leq k-1$.
- Si dice semplice un cammino i cui nodi sono tutti distinti.
- Si dice peso di un arco un valore associato all'arco dato da una funzione di peso $w : V \times V \to \mathbb{R}$.
- Un nodo v si dice raggiungibile da un nodo u se esiste almeno un cammino da u a v.
- Un grafo non orientato $G = (V, E)$ è connesso se e solo se ogni suo nodo è raggiungibile da ogni altro nodo.
- G' è detto sottografo di G $G' \subseteq G$ se e solo se $V' \subseteq V$ e $E' \subseteq E$.
- G' è sottografo massimale di G se e solo se non esiste un altro sottografo G'' tale che G'' è connesso e più grande di G'.
- Un grafo $G' = (V', E')$ è una componente connessa di G se e solo se è un sottografo connesso e massimale di G.
- In un grafo non orientato $G = (V, E)$ un ciclo C di lunghezza $k > 2$ è una sequenza di nodi $u_0, \ldots, u_k$ tale che $(u_i, i_{i+1}) \in E$ per ogni $0 \leq i \leq k-1$ e $u_0 = u_k$. Un ciclo è detto semplice se tutti i suoi nodi sono distinti.
- Un grafo è detto aciclico se non contiene cicli. Se orientato è detto DAG (directed acyclic graph).
- Dato un DAG, un suo ordinamento topologico è un ordinamento lineare dei suoi nodi tale che se $(u, v) \in E$, allora u appare prima di v nell'odinamento.
- Un grafo orientato $G = (V, E)$ si dice fortemente connesso se e solo se ogni suo nodo è raggiungibile da ogni altro suo nodo.
- Un grafo $G' = (V', E')$ è una componente fortemente connessa di G se e solo se è un sottografo connesso e massimale di G.

8.0.2 Specifica

In alcuni casi il grafo è dinamico ma sono possibili solo inserimenti e non è necessaria la parte di eliminazione (grafo caricato all'inizio).

: Graph
`Graph()` %Crea un nuovo grafo
Set `V()` %Restituisce l'insieme di tutti i nodi
int `size()` %Restituisce il numero di nodi
Set `adj(Node` u`)` %Restituisce i nodi adiacenti al nodo u
`insertNode(Node` u`)` %Inserisce il nodo u nel grafo
`insertEdge(Node` u`, Node` v`)` %Inserisce l'arco tra u e v nel grafo
`deleteNode(Node` u`)` %Elimina il nodo u dal grafo
`deleteEdge(Node` u`, Node` v`)` %Elimina l'arco tra i nodi u e v dal grafo

8.1 Memorizzare grafi

8.1.1 Matrice di adiacenza

Le matrici di adiacenza sono ideali per i grafi densi. Un grafo orientato viene salvato in una matrice di dimensione $n \times n$ bit tale che:

$$m_{uv} = \begin{cases} 1 & (u,v) \in E \\ 0 & (u,v) \notin E \end{cases}$$

Per il grafo non orientato basta salvare la matrice sopra la diagonale principale esclusa in modo da occupare solamente $\frac{n(n-1)}{2}$ bit.

Grafi pesati Per i grafi pesati la matrice di booleani diventa una matrice di reali tale che:

$$m_{uv} = \begin{cases} peso & (u,v) \in E \\ +\infty & (u,v) \notin E \end{cases}$$

Analisi complessità

- Spazio richiesto $O(n^2)$.
- Verificare se u è adiacente a v richiede $O(1)$.
- Iterare su tutti gli archi richiede $O(n^2)$

8.1.2 Liste di adiacenza

Le liste di adiacenza sono ideali per i grafi sparsi. Viene creato un vettore di dimensione n tale che in posizione u si trovi un insieme tale che contenga gli adiacenti. Occupa $an + bm$ bit. Dove a è la dimensione del puntatore al nodo adiacente e b è la dimensione dell'adiacente. Per il grafo non orientato viene salvato anche l'arco inverso e occupa $an + 2bm$.

Grafi pesati Gli elementi della lista puntata sono coppie nodo di arrivo-peso.

Analisi complessità

- Spazio richiesto $O(n + m)$.
- Verificare se u è adiacente a v richiede $O(n)$.
- Iterare su tutti gli archi richiede $O(n + m)$

8.1.3 Iterazioni su nodi e archi

: Iterazione su tutti i nodi del grafo

```
foreach u ∈ G.V() do
    Esegui operazioni sul nodo
     u
```

: Iterazione su tutti i nodi e archi del grafo

```
foreach u ∈ G.V() do
    Esegui operazioni sul nodo
     u;
    foreach v ∈ G.adj(u) do
        Esegui operazioni
         sull'arco (u, v)
```

8.2 Visite dei grafi

Dato un grafo $G = (V, E)$ e $r \in V$ radice o sorgente si intende per visita visitare una e una sola volta tutti i nodi che possono essere raggiunti da r.

: `graphTraversal(Graph` G, `Node` r`)`

```
Set S = Set() %Insieme generico
S.insert(r) %Da specificare
{Marca il nodo r}
while S.size() > 0 do
    Node u = S.remove() %Da specificare
    {Visita il nodo u}
    foreach v ∈ G.adj(u) do
        {Visita l'arco (u,v)}
        if v non è stato marcato then
            Marca il nodo v
            S.insert(v) %Da specificare
```

8.2.1 Visita in ampiezza o breadth-first search

In questo tipo di visita si visitano i nodi per livelli crescenti a partire dalla sorgente. Gli obiettivi di questa visita sono di visitare i nodi a distanza k prima dei nodi a distanza $k+1$, calcolare il cammino più breve da r a tutti gli altri nodi (distanze misurate come numero di archi attraversati). Generare un albero breadth-first contenente tutti i nodi raggiungibili da r tale per cui il cammino dalla radice r al nodo u nell'albero corrisponde al cammino più breve nel grafo.

```
: bfs(Graph G, Node r)
Queue Q = Queue()
Q.enqueue(r)
boolean [] visited = new boolean [G.size()]
foreach u ∈ G.V() - {r} do
    visited[u] = false
visited[r] = true
while not Q.empty() do
    Node u = Q.dequeue()
    {Visita il nodo u}
    foreach v ∈ G.adj(u) do
        {Visita l'arco (u,v)}
        if not visited[v] then
            visited[v] = true
            Q.enqueue(v)
```

Calcolo dei cammini più brevi o del numero di Erdös

L'algoritmo di erdos oltre a determinare la lunghezza del cammino più breve genera l'arco di copertura bfs che contiene tali cammini.

```
: erdos(Graph G, Node r, int [] erdos, Node [] parents)
Queue Q = Queue()
Q.enqueue(r)
foreach u ∈ G.V() - {r} do
    erdos[u] = ∞
erdos[r] = 0
parents[r] = nil
while not Q.empty() do
    Node u = Q.dequeue()
    {Visita il nodo u}
    foreach v ∈ G.adj(u) do
        if erdos[v] = ∞ then
            erdos[v] = erdos[u] + 1
            parents[v] = u
            Q.enqueue(v)
```

```
: printPath(Node r, Node s, Node [] parents)
if r = s then
    print s
else if parents[s] == nil then
    print "error"
else
    printPath(r, parents[s], parents)
    print s
```

Complessità

La complessità di una bfs è $O(m + n)$ in quanto ognuno degli n nodi viene salvato nella coda al più una volta. Ogni volta che un nodo viene estratto tutti i suoi archi vengono analizzati una sola volta. Il numero di archi analizzati è pertanto $m = \sum_{u \in V} out_d(u)$, dove $out_d(u)$ è l'out-degree del nodo u.

8.2.2 Visita in profondità o depth-first search

La visita in profondità è una visita ricorsiva: per ogni nodo adiacente si visita ricorsivamente tale nodo, visitando ricorsivamente i suoi adiacenti e così via. Questo tipo di ricerca è spesso una subroutine in altri problemi, visita tutto il grafo, nono solo i nodi raggiungibili dalla radice e genera pertanto una foresta formata da una collezione di alberi depth-first. Si realizza attraverso uno stack che viene reso implicito attraverso la ricorsione.

Implementazione ricorsiva

```
: dfs(Graph G, Node u, boolean [] visited)
visited[u] = true
{visita il nodo u in pre-order}
foreach v ∈ G.adj(u) do
    if not visited[v] then
        {Visita l'arco (u,v)}
        dfs(G, v, visited)
{visita il nodo u in post-order}
```

Implementazione iterativa

Una dfs ricorsiva in casi di grafi molto profondi può superare la dimensione dello stack del linguaggio, si rende pertanto esplicito lo stack utilizzando una versione iterativa. Un nodo può essere inserito nella pila più volte in quanto il controllo viene fatto all'estrazione. Ha complessità $O(m+n)$ con $O(m)$ visite degli archi, $O(m)$ estrazioni e inserimenti e $O(n)$ visite dei nodi. Per la visita in post order quando un nodo viene scoperto viene inserito nello stack con il tag *discovery*, quando viene estratto un nodo con il tag *discovery* viene re-inserito con il tag *finish* e vengono inseriti tutti i suoi vicini. Quando viene estratto un nodo con il tag *finish* viene fatta la post-visita.

```
: dfs(Graph G, Node r)
Stack S = Stack()
S.push(r)
boolean [] visited = new boolean [G.size()]
foreach u ∈ G.V() do
    visited[u] = false
while not S.empty() do
    Node u = S.pop()
    if not visited[u] then
        {Visita il nodo u in pre-order}
        visited[u] = true
        foreach v ∈ G.adj(u) do
            {Visita l'arco (u,v)}
            S.push(v)
```

Componenti connesse

Ci si pone il problema di verificare se il grafo è connesso e se non lo è di identificare le sue componenti connesse. Un grafo risulta connesso se al termine della dfs tutti i suoi nodi sono marcati, altrimenti la visita deve ricominciare da capo da un nodo non marcato identificando una nuova componente del grafo. L'appartenenza ad una componente connessa viene identificata attraverso un vettore *id[]* tale per cui *id[u]* è l'identificatore della componente connessa di appartenenza.

```
: int [] cc(Graph G)
int [] id = new int [G.size()]
foreach u ∈ G.V() do
    id[u] = 0
int counter = 0
foreach u ∈ G.V() do
    if id[u] == 0 then
        counter += 1
        ccdfs(G, counter, u,
          id)
return id
```

```
: int [] ccdfs(Graph G, int
counter, Node u, int [] id)
id[u] = counter
foreach v ∈ G.adj(u) do
    if id[v] == 0 then
        ccdfs(G, counter, v,
          id)
```

Grafo non orientato aciclico

```
: boolean [] hasCycleRec(Graph G, Node u, Node p, boolean []
visited)
visited[u] = true
foreach v ∈ G.adj(u) - {p} do
    if visited[u] then
        return true
    else if hasCycleRec(G, v, u, visited) then
        return true
return false
```

```
: boolean [] hasCycle(Graph G)
boolean [] visited = new boolean [G.size()]
foreach u ∈ G.V() do
    visited[u] = false
foreach u ∈ G.V() do
    if not visited[u] then
        if hasCycleRec(G, u, nil, visited) then
            return true
return false
```

Grafo orientato aciclico

Classificazione degli archi Ogni qual volta si esamina un arco da un nodo marcato ad un nodo non marcato tale arco diventa un arco dell'albero di copertura dfs T. Gli archi non inclusi nell'albero possono essere divisi in tre categorie:

- Se u è un antenato di v in T (u, v) è detto arco in avanti.
- Se u è un discendente di v in T (u, v) è detto arco all'indietro.
- Se i primi due non sussistono viene detto arco di attraversamento.

Indicato con time un contatore, il vettore *dt* il discovery time, o tempo di scoperta di un nodo e *ft* il finish time o tempo di fine di un nodo.

```
: dfs-schema(Graph G, Node u, int & time, int [] dt, int [] ft)
{Visita il nodo u in pre-order}
time += 1
dt[u] = time
foreach v ∈ G.adj(u) do
    {Visita l'arco (u, v) qualsiasi}
    if dt[v] = 0 then
        {Visita l'arco (u, v) albero}
        dfs-schema(G, v, time, dt, ft)
    else if dt[u] > dt[v] and ft[v] = 0 then
        Visita l'arco (u, v) indietro
    else if dt[u] < dt[v] and ft[v] ≠ 0 then
        Visita l'arco (u, v) avanti
    else
        Visita l'arco (u, v) attraversamento
{Visita il nodo u in post-order}
time +=1
ft[u] = time
```

Classificare gli archi permette di dimostrare proprietà rispetto a questa classificazione e pertanto costruire degli algoritmi migliori. In particolare data una visita dfs di un grafo G 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 e v non sono discendenti l'uno dell'altro nella foresta DF.
- L'intervallo $[dt[u], ft[u]]$ è contenuto in $[dt[v], ft[v]]$, allora u è un discendente di v nella foresta DF.
- L'intervallo $[dt[v], ft[v]]$ è contenuto in $[dt[u], ft[u]]$, allora v è un discendente di u nella foresta DF.

Teorema un grafo è aciclico se e solo se non esistono archi all'indietro nel grafo.

Dimostrazione Se esiste un ciclo, sia u il primo nodo del ciclo che viene visitato e $(v, u$ un arco del ciclo. Il cammino che connette v a u verrà prima o poi visitato e da v verrà scoperto l'arco all'indietro (v, u). 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.

```
: hasCycle(Graph G, Node u, int & time, int [] dt, int [] ft)
time += 1
dt[u] = time
foreach v ∈ G.adj(u) do
    if dt[v] == 0 then
        if hasCycle(G, v, time, dt, ft) then
            return true
    else if dt[u] > dt[v] and ft[v] == 0 then
        return true
time +=1
ft[u] = time
return false
```

Ordinamento topologico

L'algoritmo per un ordinamento topologico consiste in una dfs in cui l'operazione di visita consiste nell'aggiungere il nodo in testa ad una lista a tempo di fine (ovvero post-ordine). L'output sarà pertanto una sequenza di nodi ordinata per tempo decrescente di fine. In questo modo quando un nodo è finito tutti i suoi discendenti sono stati aggiunti alla lista e aggiungendolo in testa è nell'ordine corretto.

```
: Stack topSort(Graph G)
Stack S = Stack()
boolean [] visited = boolean [G.size()]
foreach u ∈ G.V() do
    visited[u] = false
foreach u ∈ G.V() do
    if not visited[u] then
        ts-dfs(G, visited, S)
return S
```

```
: Stack ts-dfs(Graph G, Node u, boolean [] visited, Stack S)
visited[u] = true
foreach v ∈ G.adj(u) do
    if not visited[v] then
        ts-dfs(G, v, visited, S)
S.push(u)
```

Algoritmo di Kosaraju

Utilizzato per trovare le componenti fortemente connesse di un grafo orientato. Per farlo effettua una visita dfs al grafo, calcola il grafo trasposto G_t ed esegue una visita dfs sul grafo G_t utilizzando *cc* esaminando i nodi in ordine inverso di tempo di fine della prima visita. Le componenti connesse e i relativi alberi DF rappresentano le componenti fortemente connesse di G. (Sostituito dall'algoritmo di Tarjan che richiede una sola visita). Applicando l'algoritmo topologico si

```
: int [] scc(Graph G)
Stack S = topSort(G) %First visit
Graph G^T = transpose(G) %Graph transposal
return ccG^T, S %Second visit
```

è sicuri che se un arco (u, v) non appartiene ad un ciclo, u viene listato prima di v nella sequenza ordinata e gli archi di un ciclo vengono listati in un qualche ordine che è ininfluente. Si utilizza *topSort()* per ottenere i nodi in ordine decrescente di tempo di fine. Essendo che ogni passo di questo algoritmo richiede $O(n + m)$, la complessità totale è $O(m + n)$.

Calcolo del grafo trasposto Dato un grafo orientato $G = (V, E)$ il corrispettivo grafo trasposto $G_t = (V, E_t)$ è un grafo con gli stessi nodi ma con gli archi orientati nel senso opposto: $E_t = \{(u, v)|(v, u) \in E\}$. L'algoritmo per il suo calcolo ha costo $O(m + n)$.

```
: int [] transpose(Graph G)
Graph G^T = Graph(G)
foreach u ∈ G.V() do
    G^T.insertNode(u)
foreach u ∈ G.V() do
    foreach v ∈ G.adj(u) do
        G^T.insertEdge(v, u)
return G^T
```

Calcolo delle componenti connesse Invece di esaminare i nodi in ordine arbitrario questa versione li esamina nell'ordine LIFO memorizzato nello stack.

```
: int [] cc(Graph G, Stack S)
int [] id = new int [G.size()]
foreach u ∈ G.V() do
    id[u] = 0
int counter = 0
while not S.empty() do
    u = S.pop()
    if id[u] == 0 then
        counter += 1
        ccdfs(G, counter, u,
          id)
return id
```

```
: ccdfs(Graph G, int counter,
Node u, int [] id)
id[u] = counter
foreach v ∈ G.adj(u) do
    if id[u] == 0 then
        ccdfs(G, counter, v,
          id)
```

Dimostrazione di correttezza Si costruisca il grafo delle componenti $C(G) = (V_C, E_C)$, dove $V_C = \{C_1, \ldots, C_k\}$, dove C_i è la i-esima SCC di G e $E_C = \{(C_i, C_j) | \exists (u_i, u_j) \in E : u_i \in C_i \wedge u)j \in C_j\}$. Il grafo delle componenti è aciclico e inoltre $C(G^T) = [C(G)]^T$. I discovery e finish time del grafo delle componenti corrispondono ai corrispettivi del primo nodo visitato in C: $dt(C) = \min\{dt(u) | u \in C\}$ e $ft(C) = \max\{ft(u) | u \in C\}$.

Teorema Siano C e C' due distinte SCCs nel grafo orientato $G = (V, E)$. Se c'è un arco $(C, C') \in E_C$, allora $ft(C) > ft(C')$.

Corollario Siano C_u e C_v due SCCs distinte nel grafo orientato $G = (V, E)$, se cè un arco $(u, v) \in E_T$ tale che $u \in C_u$ e $v \in C_v$, allora $ft(C_u) <$

$ft(C_v)$.

$$\begin{aligned}(u,v) \in E_T \Rightarrow \\ (v,u) \in E \Rightarrow \\ (C_v, C_u) \in E_C \Rightarrow \\ ft(C_v) > ft(C_u) \Rightarrow \\ ft(C_u) < ft(C_v)\end{aligned}$$

Conclusione Se le componenti C_u e C_v sono connesse da un arco $(u,v) \in E_T$ allora dal corollario $ft(C_u) < ft(C_v)$ e dall'algoritmo la visita di C_v inizierà prima della visita di C_u. Non esistono cammini tra C_u e C_v in E_T altrimenti il grafo sarebbe ciclico, pertando dall'algoritmo la visita di C_v non raggiungerà mai C_u, pertanto *cc()* assegnerà correttamente gli identificatori delle componenti ai nodi.

Capitolo 9

Strutture dati speciali

9.1 Code con priorità

Una coda con priorità è una struttura dati astratta simile ad una coda in cui ogni elemnto inserito possiede una priorità. Si dividono in min-priority queue in cui l'estrazione avviene per valore crescente di priorità e max-priority queue, in cui l'estrazione avviene per valore decrescente di priorità. Le operazioni permesse sono di inserimento in coda, estrazione dell'elemento con priorità di valore min/max e di modifica di priorità di un elemento inserito.

9.1.1 Specifica

: MinPriorityQueue
%Crea una coda con priorità vuota `MinPriorityQueue()` %Restituisce **true** se la coda con priorità è vuota **boolean** `isEmpty()` %Restituisce l'elemento minimo di una coda con priorità non vuota Item `min()` %Rimuove e restituisce l'elemento minimo di una coda con priorità non vuota Item `DeleteMin()` %Inerisce l'elemento x con priorità p all'interno della coda con priorità, restituendo un oggetto PriorityItem che identifica x all'interno della coda PriorityItem `insert`(Item *x,* **int** *p*) %Diminuisce la priorità dell'oggetto identificato da y portandola a p `decrease`(PriorityItem *x,* **int** *p*)

9.1.2 Implementazioni

Metodo	Lista o vettore non ordinato	Lista ordinata	Vettore ordinato	Albero RB
min()	$O(n)$	$O(1)$	$O(1)$	$O(\log n)$
deleteMin()	$O(n)$	$O(1)$	$O(n)$	$O(\log n)$
insert()	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$
decrease()	$O(n)$	$O(n)$	$O(\log n)$	$O(\log n)$

L'implementazione più comune è attraverso gli heap, strutture speciali che associano i vantaggi di un albero (esecuzioni in tempo $O(\log n)$) e vantaggi di un vettore (memorizzazione efficiente).

9.1.3 Heap

Definizioni di alberi

Albero binario perfetto In un albero binario perfetto tutte le foglie hanno la stessa altezza h, tutti i nodi interni hanno grado 2 e dato n il numero di nodi ha altezza $h = \lfloor \log n \rfloor$. Data l'altezza h ha numero di nodi $n = 2^{h+1} - 1$.

Albero binario completo In un albero binario completo tutte le foglie hanno altezza h o $h - 1$, tutti i nodi a livello h sono accatastati a sinistra e tutti i nodi hanno grado 2 tranne al più uno. Dato il numero di nodi n ha altezza $h = \lfloor \log n \rfloor$.

Alberi binari heap

Un albero binario max-heap (min-heap) è un albero completo tale che il valore memorizzato nel nodo è maggiore (minore) dei valori memorizzati nei suoi figli. Un albero heap non impone una relazione di ordinamento totale tra i figli di un nodo, pertanto è un ordinamento parziale.

Memorizzazione

Gli alberi binari heap sono memorizzati tramite un vettore heap $A = [0, \ldots, n-1]$ tale per cui:

- La radice si trova all'indice $i = 0$, ovvero *root()=0*.
- Il padre del nodo salvato all'indice i si trova in *p(i)* $= \lfloor \frac{(i-1)}{2} \rfloor$.
- Il figlio sinistro del nodo salvato all'indice i si trova in *l(i)* $= 2i + 1$.
- Il figlio destro del nodo salvato all'indice i si trova in *r(i)* $= 2i + 2$.

Proprietà max-heap sul vettore: $A[i] \geq A[l(i)], [i] \geq A[r(i)]$. Proprietà min-heap sul vettore: $A[i] \leq A[l(i)], [i] \leq A[r(i)]$.

heapsort()

L'*heapsort()* ordina un max-heap "in-place" prima costruendo un max-heap nel vettore e poi spostando l'elemento max in ultima posizione, ripristinando la proprietà max-heap. Utilizza pertanto le funzioni *heapBuild()* che costruisce un max-heap a partire dal vettore non ordinato e *maxHeapRestore()* che ripristina le proprietà max-heap.

maxHeapRestore() Riceve come input un vettore A e un indice i, tale per cui gli alberi binari con radici $l(i)$ e $r(i)$ sono max-heap e il suo obiettivo è modificare "in-place" il vettore A in modo tale che l'albero binario con radice i sia un max-heap. Essendo che ad ogni chiamata vengono eseguiti $O(1)$ confronti, e se il nodo i non è massimo si chiama ricorsivamente sui figli e che l'esecuzione termina quando si raggiunge una foglia e l'altezza dell'albero è pari a $\lfloor \log n \rfloor$ $T(n) = O(\log n)$.

```
: maxHeapRestore(Item [] A, int i, int i)
int max = i
if l(i) ≤ dim and A[l(i)] > A[max] then
    max = l(i)
if r(i) ≤ dim and A[r(i)] > A[max] then
    max = r(i)
if i ≠ max then
    A[i] ↔ A[max] maxHeapRestore(A, max, dim)
```

Dimostrazione della correttezza per induzione sull'altezza Al termine dell'esecuzione l'albero radicato in $A[i]$ rispetta la proprietà max-heap.

- Caso base $h = 0$: se l'altezza è zero esiste un unico nodo che rispetta la proprietà.
- Ipotesi induttiva: l'algoritmo funziona correttamente su tutti gli alberi di altezza inferiore ad h.
- Passo induttivo:
  - Caso 1: $A[i] \geq A[l(i)], A[i] \geq A[r(i)]$, allora l'albero radicato in $A[i]$ rispetta la proprietà di max-heap e l'esecuzione termina.
  - Caso 2: $A[l(i)] > A[i], A[r(i)] > A[i]$, viene fatto uno scambio $A[i] \leftrightarrow A[l(i)]$, dopo lo scambio $A[i] \geq A[l(i)], A[i] \geq A[r(i)]$, il sottoalbero radicato in $r(i)$ rimasto inalterato, mentre il sottoalbero radicato in $l(i)$ può aver perso la proprietà, pertanto si svolge la chiamata ricorsiva su di esso che ha altezza minore di h, pertanto è corretto.
  - Caso 3: simmetrico rispetto al caso 2.

heapBuild() Sia $A[1,\dots,n]$ un vettore da ordinare, tutti i nodi $A[\lfloor \frac{n}{2} \rfloor + 1,\dots,n]$ sono foglie dell'albero e pertanto heap contenenti un elemento. La procedura *heapBuild()* attraversa i restanti nodi dell'albero a partire da $\lfloor \frac{n}{2} \rfloor$ fino ad 1 ed esegue *maxHeapRestore()* su ognuno di essi. L'algoritmo ha complessità $\Theta(\log n)$. Aggiungere dimostrazione, non la capisco.

```
: heapBuild(Item [] A, int n)
for i = ⌊n/2⌋ down to 1 do
    maxHeapRestore(A, i, n)
```

Correttezza Si dimostra attraverso l'invariante di ciclo: all'inizio di ogni iterazione del ciclo **for** i nodi $[i+1,\dots,n]$ sono radice di uno heap.

- Inizializzazione: all'inizio $i = \lfloor \frac{n}{2} \rfloor$. Si supponga che $\lfloor \frac{n}{2} \rfloor + 1$ non sia una foglia, pertanto esiste almeno il figlio sinistro $2\lfloor \frac{n}{2} \rfloor + 2$, con indice superiore ad n, pertanto assurdo. La dimostrazione vale per tutti gli indici successivi.
- Conservazione: È possibile applicare *maxHeapRestore()* al nodo i in quanto $2i \leq 2i+1 \leq n$ sono entrambi radici di heap. Al termine dell'iterazione tutti i nodi $[1,\dots,n]$ sono radici di heap.
- Conclusione: al termine $i = 0$, pertanto il nodo 1 è radice di uno heap.

Implementazione *heapsort* Il primo elemento contiene il massimo, viene collocato in fondo, l'elemento in fondo viene collocato in testa, si chiama *maxHeapRestore()* per ripristinare la situazione e la dimensione dello heap viene progressivamente ridotta. L'algoritmo ha complessità $\Theta(n \log n)$ in quanto *heapBuild()* ha costo $\Theta(n)$ e la chiamata di *maxHeapRestore()* costa $\Theta(\log i)$ in un heap con i elementi. Pertanto $T(n) = \sum_{i=2}^{n} \log i + \Theta(n) = \Theta(n \log n)$.

```
: heapsort(Item [] A, int n)
heapBuild(A, n)
for i = n down to 2 do
    A[1] ↔ A[i]
    maxHeapRestore(A, 1, i-1)
```

Correttezza Si dimostra la correttezza per invariante di ciclo. Al passo i il sottovettore $A[i+1,\dots,n]$ è ordinato, $A[1,\dots,i] \leq A[i+1,\dots,n]$ e $A[1]$ è la radice di un vettore heap di dimensione i. COMPLETARE DIMOSTRAZIONE

9.1.4 Implementazione di code con priorità

Si vedrà l'implementazione di una min-priority queue, visto che la max-priority è speculare.

Memorizzazione

```
: PriorityItem
int priority
Item value
int pos;
```

```
: swap(PriorityItem [] H, int i, int j)
H[i] ↔ H[j]
H[i].pos = i
H[j].pos = j
```

Inizializzazione

```
: PriorityQueue
int capacity
int dim
PriorityItem [] H
PriorityQueuePriorityQueue(int n)
    PriorityQueue t = new PriorityQueue
    t.capacity = n
    t.dim = 0
    t.H = new PriorityItem [1, . . . , n]
    return t
```

Inserimento

: PriorityItem `insert(`Item x, **int** p`)`

precondition: dim < capacity
dim += 1
H[dim] = **new** `PriorityItem()`
H[dim].value = x
H[dim].priority = p
H[dim].pos = dim
int i = dim
while $i > 1$ **and** *H[i].priority* < *H[*`p(`i`)`*].priority* **do**
    `swap(`H, i, `p(`i`))`
    i = `p(`i`)`
return H[i]

minHeapRestore()

: `minHeapRestore(`PriorityItem *[]* A, **int** i, **int** dim`)`

int min = i
if `l(`i`)` ≤ dim **and** *A[*`l(`i`)`*].priority* < *A[min].priority* **then**
    min = `l(`i`)`
if `r(`i`)` ≤ dim **and** *A[*`r(`i`)`*].priority* < *A[min].priority* **then**
    min = `r(`i`)`
if $i \neq min$ **then**
    `swap(`A, i, min`)`
    `minHeapRestore(`A, min, dim`)`

Cancellazione e lettura minimo

: Item `DeleteMin()`

precondition: dim > 0
`swap(`H, *1*, dim`)`
dim -= 1
`minHeapRestore(`H, *1*, dim`)`
return H[dim+1].value

: Item `min()`

precondition: dim > 0
return H[1].value

Decremento priorità

```
: decrease(PriorityItem x, int p)
precondition: p < x.priority
x.priority = p
int i = x.pos
while i > 1 and H[i].priority < H[p(i)].priority do
    swap(H, i, p(i))
    i = p(i)
```

Complessità

Tutte le operazioni che modificano gli heap devono sistemare la proprietà heap lungo un cammino radice-foglia (*deleteMin()*) oppure lungo un cammino nodo-radice (*insert()*, *decrease()*). Essendo l'altezza $\lfloor \log n \rfloor$ il costo di tali operazioni è $O(\log n)$.

Operazione	Costo
insert()	$O(\log n)$
deleteMin()	$O(\log n)$
min()	$\Theta(1)$
decrase()	$O(\log n)$

9.2 Insiemi disgiunti - Merge-find set

In alcune applicazioni si è interessati a gestire una collezione $S = \{S_1, \ldots, S_k\}$ di insiemi dinamici disgiunti tali che:

- $\forall i, j : i \neq j \Rightarrow S_i \cap S_j = \emptyset$.
- $\bigcup_{i=1}^{k} S_i = S$.

Un esempio di questi insiemi sono le componenti di un grafo. Le operazioni fondamentali sono:

- Creare n insiemi disgiunti, ognuno composto da un unico elemento.
- *merge()*: unire più insiemi.
- *find()*: identificare l'insieme a cui appartiene un elemento.

Rappresentante

Ogni insieme è identificato da un rappresentante univoco, un qualunque membro dell'insieme. Le operazioni di ricerca del rappresentante su uno stesso insieme devono restituire sempre lo stesso oggetto. Il rappresentante può cambiare solo in caso di unione.

9.2.1 Specifica

```
: Mfset
%Crea n componenti {1},...,{n}
Mfset Mfset(int n)
%Restituisce il rappresentante della componente x
int find(int x)
%Unisce le componenti che contengono x e y
merge(int x, int y)
```

9.2.2 Componenti connesse dinamiche

Per trovare le componenti connesse di un grafo non orientato dinamico si inizia con componenti connesse costituite da un unico vertice, per ogni $(u, v) \in E$ si esegue *merge(u, v)* e alla fine ogni insieme disgiunto rappresenta una componente connessa.

```
: Mfset cc(Graph G)
Mfset M = Mfset(G.n)
foreach u ∈ G.V() do
    foreach v ∈ G.adj(u) do
        M.merge(u, v)
return M
```

Questo algoritmo ha complessità $O(n) + m$ operazioni *merge()*, interessante per la capacità di gestire grafi dinamici.

9.2.3 Implementazione con insieme di liste

Ogni insieme viene rappresentato da una lista concatenata: il primo oggetto di una lista è il rappresentante dell'insieme e ogni elemento della lista contiene un oggetto, un puntatore all'elemento successivo e un puntatore al rappresentante.

Operazione *find(x)*

Si restituisce il rappresentante di x, l'operazione *find(x)* richiede tempo $O(1)$.

Operazione *merge(x, y)*

Si appende la lista che contiene y alla lista che contiene x modificando i puntatori ai rappresentanti nella lista appesa. Nel caso pessimo per n operazioni il costo è $O(n^2)$, il costo ammortizzato è $O(n)$.

9.2.4 Implementazione con insieme di alberi

Ogni insieme viene rappresentato da un albero, ogni nodo dell'albero contiene un oggetto e un puntatore al padre, la radice è il rappresentante dell'insieme e ha un puntatore a sè stessa.

Operazione *find(x)*

Risale la lista dei padri di x fino a trovare la radice e restituisce la radice come rappresentante. Ha costo $O(n)$ nel caso pessimo

Operazione *merge(x, y)*

Si aggancia l'albero radicato in y a y modificando il puntatore al padre di y, con costo $O(1)$.

9.2.5 Tecniche euristiche

Si dice algoritmo euristico un particolare tipo di algoritmo progettato per risolvere un problema più velocemente se i metodi classici siano troppo lenti, trovare una soluzione approssimata qualora i metodi classici falliscano nel trovare una soluzione esatta.

Liste: euristica sul peso

Si utilizza per diminuire il costo dell'operazione *merge()* si memorizza nella lista l'informazione sulla loro lunghezza e si aggancia la lista più corta a quella più lunga. La lunghezza della lista può essere mantenuta in tempo $O(1)$. Tramite analisi ammortizzata si nota come il costo di $n-1$ operazioni merge è $O(n \log n)$, pertanto il costo ammortizzato delle singole operazioni è $O(\log n)$.

Alberi: euristica sul rango

Si utilizza per diminuire il costo dell'operazione *find(x)*: ogni nodo mantiene informazioni sul proprio rango, *rank[x]* di un nodo x è il numero di archi del cammino più lungo fra x e una foglia sia discendente, ovvero l'altezza del sottoalbero associato al nodo. L'obiettivo è mantenere bassa l'altezza degli alberi. Un albero *Mfset* con radice r ottenuto tramite euristica sul rango ha almeno $2^{ranl[r]}$ nodi. Inoltre lo stesso albero con radice r e n nodi ha altezza inferiore a $\log n$.

```
: Mfset
    int [] parent
    int [] rank[]
    Mfset Mfset(int n)
        Mfset t = new Mfset()
        t.parent = int [1...n]
        t.rank = int [1...n]
        for int i = 1 to n do
            t.parent [i] = i
            t.rank [i] = 0
        return t
```

```
: merge(int x, int y)
    Item r_x = find(x)
    Item r_y = find(y)
    if r_x ≠ r_y then
        if rank[r_x] > rank[r_y] then
            parent[r_y] = r_x
        if rank[r_y] > rank[r_x] then
            parent[r_x] = r_y
        else
            parent[r_x] = r_y
            rank[r_y] += 1
```

Alberi: euristica di compressione dei cammini

Nell'operazione *find(x)* l'albero viene appiattito in modo che ricerche successive di x siano svolte in $O(1)$.

```
: int find(int x)
  if parent[x] ≠ x then
      parent[x] =
        find(parent[x])
  return parent[x]
```

Alberi: euristica sul rango e compressione dei cammini

Applicando entrambe le euristiche il rango non è più l'altezza del nodo ma il limite superiore alla sua altezza. Non viene calcolato il rango corretto. Il costo ammortizzato di m operazioni merge-find in un insieme di n elementi è $O(m\alpha(n))$, dove $\alpha(n)$ o funzione inversa di Ackermann cresce lentamente.

Capitolo 10

Scelta

TO DO

Capitolo 11

Risoluzione problemi

Dato un problema è possibile evidenziare quattro fasi, non necessariamente sequenziali in modo trovare una soluzione efficiente:

- Classificazione del problema.
- Caratterizzazione della soluzione.
- Tecnica di progetto.
- Utilizzo di strutture dati.

11.1 Classificazione dei problemi

Problemi decisionali

Determinare se un dato di ingresso soddisfa una certa proprietà.

Problemi di ricerca

È presente uno spazio di ricerca, un insieme di soluzioni possibili, una soluzione ammissibile, che rispetta certi vincoli.

Problemi di ottimizzazione

Ogni soluzione viene associata ad una funzione di costo, si vuole trovare la soluzione di costo minimo.

Problemi di approssimazione

Questo tipo di problemi nasce quando trovare la soluzione ottima è computazionalmente impossibile e pertanto ci si accontenta di una soluzione approssimata con costo basso, ma l'ottimalità è sconosciuta.

11.2 Definizione matematica del problema

È fondamentale definire il problema dal punto id vista matematico, spesso la formulazione è banale ma può suggerire una prima soluzione.

11.3 Tecniche di soluzione problemi

Divide-et-impera

Un problema viene suddiviso in sotto-problemi indipendenti che vengono risolti ricorsivamente (top-down) nell'ambito di problemi di decisione e ricerca.

Programmazione dinamica

La soluzione viene costruita bottom-up a partire da un insieme di sotto-problemi potenzialmente ripetuti nell'ambito dei problemi di ottimizzazione.

Memoization

Una versione top-down della programmazione dinamica.

Tecnica greedy

Approccio ingordo: si fa sempre la scelta localmente ottima.

Backtrack

Si procede per tentativi tornando quando necessario all'indietro.

Ricerca locale

La soluzione ottima viene trovata migliorando via via soluzioni esistenti.

Algoritmi probabilistici

Algoritmi in cui certe scelte avvengono in maniera casuale.

Capitolo 12

Divide-et-impera

In questo metodo di risoluzione problemi ci sono tre fasi:

- Divide: si divide il problema in sotto-problemi più piccoli e indipendenti.
- Impera: si risolvono i sotto-problemi ricorsivamente.
- Combina: si uniscono le soluzioni dei sottoproblemi.

12.1 Le torri di Hanoi

Le torri di Hanoi è un gioco matematico costituito da tre pioli con n dischi di dimensioni diverse. Inizialmente i dischi sono impilati in ordine decrescente nel piolo di sinistra. Lo scopo del gioco è impilare in ordine decrescente i dischi sul piolo di destra; senza mai impilare un disco più grande su uno più piccolo, muovendo al massimo un disco alla volta e utilizzando il piolo centrale come appoggio.

```
: hanoi(int n, int src, int middle)
if n = 1 then
    print src → dest
else
    hanoi(n -1, src, middle,
      dest)
    print src → dest
    hanoi(n -1, middle, dest,
      src)
```

Questo algoritmo trova la soluzione ottima, la ricorrenza è

$$T(n) = 2T(n-1) + 1$$

con costo computazionale $O(2^n)$.

12.2 Quicksort

Quicksort è un algoritmo di ordinamento basato su divide-et-impera, nel caso medio è $O(n \log n)$, nel caso pessimo $O(n^2)$. Il fattore costante è migliore di quello di Mergesort, non utilizza memoria addizionale e presenta tecniche euristiche per evitare il caso pessimo.

12.2.1 Caratterizzazione

Input

Un vettore $A[1 \dots n]$, indici *start*, *end* tali che $1 \leq start \leq end \leq n$.

Divide

Si sceglie un valore $p \in A[start \dots end]$ detto perno o pivot. Si spostano gli elementi del vettore $A[start \dots end]$ in modo tale che:

- $\forall i \in [start \dots j-1] : A[i] \leq p$.
- $\forall i \in [j+1 \dots end] : A[i] \geq p$.

L'indice j viene calcolato in modo tale da rispettare tale condizione, il perno viene messo in posizione $A[j]$.

Impera

Ordina i due sottovettori $A[start \dots j-1]$ e $A[j+1 \dots end]$ richiamando ricorsivamente Quicksort.

Combina

Non fa nulla: il primo vettore, $A[j]$ e il secondo vettore formano già un vettore ordinato.

```
: int pivot(Item [] A, int start, int end)
    Item p = A[start]
    int j = start  for int i = start +1 to end do
    |  if A[i] < p then
    |  |  j += 1
    |  |  A[i] ↔ A[j]
    A[start] = A[j]
    A[j] = p
    return j
```

```
: 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)
```

Capitolo 13

Programmazione dinamica

Si dice programmazione dinamica un metodo per dividere un problema ricorsivamente in sottoproblemi in modo che ogni sottoproblema venga rivolto una volta sola e la sua soluzione venga memorizzata in una tabella. Se un sottoproblema deve essere risolto nuovamente si ottiene la sua soluzione dalla tabella (lookup in $O(1)$).

13.1 Approccio generale

Se si parte da un problema di ottimizzazione si definisce la soluzione in maniera ricorsiva, successivamente si definisce il valore della soluzione in maniera ricorsiva (primo passo per problemi di conteggio). Se da qui non sono presenti sottoproblemi ripetuti si utilizza divide et impera; se sono presenti sottoproblemi ripetuti e bisogna risolverli tutti si utilizza la programmazione dinamica, se non devono essere risolti tutti si usa memoization. Dalla tabella delle soluzioni si ottiene l'output numerico o la ricostruzione della soluzione che permette di arrivare alla soluzione ottima.

13.1.1 Evitare di risolvere i problemi più di una volta

Quando si risolve un problema si memorizza il risultato ottenuto in una tabella DP, che può essere un vettore, una matrice o un dizionario. La tabella deve contenere un elemento per ogni sottoproblema che deve essere risolto.

- Casi base: si memorizzano i casi base direttamente nella tabella.
- Iterazione bottom-up: si parte dai sottoproblemi che possono essere risolti direttamente a partire dai casi base, si sale verso problemi sempre più grandi fino a raggiungere il problema originale.

13.1.2 Ricostruire la soluzione

Per ricostruire la soluzione si parte dalla definizione ricorsiva del problema, la dimensione di n indica l'indice sulla tabella che si deve controllare e si definisce un controllo su quale delle chiamate ricorsive viene realmente effettuata.

13.2 Memoization

Questa tecnica fonde l'approccio di memorizzazione della programmazione dinamica con quello top-down di divide-et-impera: si crea una tabella DP inizializzata con un valore speciale ad indicare che un certo sottoproblema non è ancora stato risolto. Ogni volta che si deve risolvere un sottoproblema si controlla nella tabella se è già stato risolto precedentemente:

- già risolto: si utilizza il risultato della tabella.
- non risolto: si calcola il risultato e lo si memorizza

In questo modo ogni sottoproblema viene calcolato una sola volta e memorizzato come nella versione bottom-up. Questa tecnica presenta vantaggi quando invece di utilizzare una tabella si utilizza un dizionario in quanto non è più necessario fare inizializzazione.

Implementazione 13.1: Memoization automatica in Python

```
1  from functools import wraps
2
3  def memo(func):
4      cache = {}
5      @wraps(func)
6      def wrap(*args):
7          if args not in cache:
8              cache[args] = func(*args)
9          return cache[args]
10     return wrap
```