

Dati e algoritmi

0. Indice

- [0. Indice](#)
- [1. Nozioni](#)
 - [1.1. Problema computazionale](#)
 - [1.1.1. Esempi](#)
 - [1.1.2. Esercizi](#)
 - [1.2. Algoritmo e modello di calcolo](#)
 - [1.2.1. Algoritmo](#)
 - [1.2.2. Modello di calcolo RAM \(Random Access Machine\)](#)
 - [1.3. Pseudocodice](#)
 - [1.3.1. Esempio - Trasposta di una matrice \$n \times n\$ di interi](#)
 - [1.4. Taglia di un'istanza](#)
 - [1.4.1. Esempi](#)
 - [1.5. Struttura dati](#)
 - [1.5.1. Struttura dati - definizione](#)
 - [1.5.2. Esercizio](#)
 - [1.5.3. Esercizio](#)
- [2. Analisi degli algoritmi](#)
 - [2.1. Complessità in tempo](#)
 - [2.1.1. Requisiti per la complessità in tempo](#)
 - [2.1.2. Complessità \(in tempo\) al caso pessimo - definizione](#)
 - [2.1.3. Difficoltà nel calcolo di \$t_A\(n\)\$](#)
 - [2.1.4. Esempio \(arrayMax\)](#)
 - [2.2. Analisi asintotica](#)
 - [2.2.1. Esempio \(arrayMax\)](#)
 - [2.3. Ordini di grandezza](#)
 - [2.3.1. O-grande](#)
 - [2.3.2. Omega-grande](#)
 - [2.3.3. Theta](#)
 - [2.3.4. o-piccolo](#)
 - [2.3.5. Proprietà degli ordini di grandezza](#)
 - [2.3.6. Strumenti matematici](#)
 - [2.4. Analisi di complessità in pratica](#)

- 2.4.1. Limite superiore (upper bound) - definizione
- 2.4.2. Limite inferiore (lower bound) - definizione
- 2.4.3. Limiti superiori e inferiori
- 2.4.4. Terminologia per complessità
- 2.4.5. Esempio (prefix averages).
- 2.4.6. Esercizi
- 2.4.7. Regola di buon senso
- 2.4.8. Efficienza asintotica degli algoritmi
- 2.5. Analisi di correttezza
 - 2.5.1. Induzione
 - 2.5.2. Correttezza
 - 2.5.3. Invariante
 - 2.5.4. Correttezza di un ciclo tramite invariante
 - 2.5.6. Ricorsione
 - 2.5.7. Complessità di algoritmi ricorsivi
 - 2.5.8. Correttezza di algoritmi ricorsivi
- 3. Ripasso di Java
 - 3.1. Caratteristiche di Java e dell'approccio Object-Oriented
 - 3.1.1. Modularità
 - 3.1.2. Astrazione e encapsulamento (information hiding)
 - 3.1.3. Ereditarietà (inheritance)
 - 3.2. Programmazione generica (generics)
 - 3.3. ADT elementari
 - 3.3.1. Lista (list)
 - 3.3.2. Pila (stack) e coda (queue)
 - 3.4. Collection framework di Java
- 4. Alberi
 - 4.1. Alberi generali
 - 4.1.1. Definizioni e proposizioni
 - 4.1.2. Interfacce
 - 4.1.3. Calcolo della profondità di un nodo (algoritmo ricorsivo)
 - 4.1.4. Calcolo dell'altezza di un nodo
 - 4.1.5. Visite di alberi
 - 4.2. Alberi binari
 - 4.2.1. Interfaccia BinaryTree
 - 4.2.2. Proprietà
 - 4.2.3. Alberi binari propri estremi
 - 4.2.4. Visite di alberi binari

- 5. Priority queue
 - 5.1. Entry
 - 5.1.1. Esempio
 - 5.2. Priority Queue - definizione
 - 5.2.1. Esempio
 - 5.3. Implementazione di Priority Queue tramite liste
 - 5.3.1. Lista non ordinata
 - 5.3.2. Lista ordinata
 - 5.4. Implementazione di Priority Queue tramite Heap
 - 5.4.1. Proprietà
 - 5.4.2. Implementazione di alberi binari su array
 - 5.4.3. Implementazione di alberi binari completi su array
 - 5.4.4. Implementazione di heap su array
 - 5.4.5. Implementazione dei metodi della Priority Queue su heap
 - 5.5. Costruzione di uno heap a partire da n entry date
 - 5.5.1. Soluzioni banali
 - 5.5.2. Approccio top-down
 - 5.5.3. Approccio bottom-up
 - 5.6. Sorting tramite Priority Queue
 - 5.6.1. Complessità di psSort(S)
 - 5.6.2. HeapSort in-place
 - 5.7. Implementazione di Priority Queue in Java
 - 5.7.1. Esempio
- 6. Mappe
 - 6.1. Definizione, interfaccia, applicazioni
 - 6.2. Implementazioni semplici, inefficienti
 - 6.2.1. Lista non ordinata
 - 6.2.2. Array di taglia -U-
 - 6.3. Mappe in Java
 - 6.4. Implementazione Mappa tramite Tabella Hash
 - 6.4.1. Tabelle Hash
 - 6.4.2. Generazione hash code
 - 6.4.3. Compression function
 - 6.4.4. Risoluzione delle collisioni
 - 6.4.5. Implementazione dei metodi della Mappa
 - 6.4.6. Load factor
 - 6.4.7. Complessità di get, put e remove
 - 6.4.8. Rehashing

- [6.5. Alberi binari di ricerca](#)
 - [6.5.1. Esempio \(solo chiavi\)](#)
 - [6.5.2. TreeSearch per un albero binario di ricerca \\$T\\$](#)
 - [6.5.3. Osservazioni sugli Alberi Binari di Ricerca](#)
 - [6.6. Multi-Way Search Tree](#)
 - [6.6.1. Definizione, osservazioni, proposizione](#)
 - [6.6.2. Ricerca in un MWS-Tree](#)
 - [6.6.3. Implementazione dei metodi della Mappa: get](#)
 - [6.6.4. \(2,4\)-Tree](#)
 - [6.6.5. Implementazione dei metodi della Mappa: put e remove](#)
 - [6.7. Red-Black Tree](#)
 - [6.7.1. \(2,4\)-Tree e Red-Black Tree](#)
 - [6.7.2. Metodi della Mappa su Red-Black Tree](#)
 - [6.8. Multimappa](#)
 - [6.8.1. Metodi caratterizzanti](#)
 - [6.8.2. Esempio](#)
 - [6.8.2. Complessità dei metodi](#)
 - [6.9. Esercizi](#)
 - [6.9.1. Mappe pt 2](#)
 - [6.9.2. Mappe pt 2](#)
 - [6.9.3. File Esercizi Mappe](#)
-

Lezione 01

1. Nozioni

1.1. Problema computazionale

Un problema computazionale è costituito da

- un insieme I di *istanze* (i *possibili input*)
- un insieme S di *soluzioni* (i *possibili output*)
- una relazione Π che a ogni istanza $i \in I$ associa *una o più soluzioni* $s \in S$

(!) Osservazione

Π è un sottoinsieme del prodotto cartesiano $I \times S$

1.1.1. Esempi

Somma di Interi (\mathbb{Z})

- $\mathcal{I} = \{(x, y) : x, y \in \mathbb{Z}\};$
- $\mathcal{S} = \mathbb{Z};$
- $\Pi = \{((x, y), s) : (x, y) \in \mathcal{I}, s \in \mathcal{S}, s = x + y\}.$
Ad es: $((1, 9), 10) \in \Pi; ((23, 6), 29) \in \Pi; ((13, 45), 31) \notin \Pi$

Ordinamento di array di interi

- $\mathcal{I} = \{A : A = \text{array di interi}\};$
- $\mathcal{S} = \{B : B = \text{array ordinati di interi}\};$
- $\Pi = \{(A, B) : A \in \mathcal{I}, B \in \mathcal{S}, B \text{ contiene gli stessi interi di } A\}.$
Ad es. $(< 43, 16, 75, 2 >, < 2, 16, 43, 75 >) \in \Pi$
 $(< 7, 1, 7, 3, 3, 5 >, < 1, 3, 3, 5, 7, 7 >)$
 $(< 13, 4, 25, 17 >, < 11, 27, 33, 68 >)$

Ordinamento di array di interi (ver.2)

- $\mathcal{I} = \{A : A = \text{array di interi}\};$
- $\mathcal{S} = \{P : P = \text{permutazioni}\};$
- $\Pi = \{(A, P) : A \in \mathcal{I}, P \in \mathcal{S}, P \text{ ordina gli interi di } A\}.$
Ad es. $(< 43, 16, 75, 2 >, < 4, 2, 1, 3 >) \in \Pi$
 $(< 7, 1, 7, 3, 3, 5 >, < 2, 4, 5, 6, 1, 3 >) \in \Pi$
 $(< 7, 1, 7, 3, 3, 5 >, < 2, 5, 4, 6, 1, 3 >) \in \Pi$
 $(< 13, 4, 25, 17 >, < 1, 2, 4, 3 >)$

! Osservazione

- Istanze diverse possono avere la stessa soluzione (come la somma)
- Un'istanza può avere diverse soluzioni (come l'ordinamento ver. 2)

1.1.2. Esercizi

Esercizio

Specificare come problema computazionale Π la verifica se due insiemi finiti di oggetti da un universo U sono disgiunti oppure no.

$$I \equiv \{(A, B) : A, B \subseteq U, A, B \text{ finiti}\}$$

$$S \equiv \{\text{true}, \text{false}\}$$

$$\Pi \equiv \{((A, B), s) \text{ se } A \cap B = \emptyset \text{ allora } s = \text{true}, \text{ se } A \cap B \neq \emptyset \text{ allora } s = \text{false} \mid s \in S, (A, B) \in I\}$$

Esercizio

Specificare come problema computazionale Π la ricerca dell'inizio e della lunghezza del più lungo segmento di 1 consecutivi in una stringa binaria.

$$I \equiv \{A : A \text{ è una stringa binaria}\}$$

$$S \equiv \{(i, l) : i, l \in \mathbb{N}_0\}$$

$$\Pi \equiv \{(A, (i, l)) : i \text{ la casella di inizio del segmento di 1 consecutivi più numeroso, } l \text{ la lunghezza del segmento di 1 consecutivi più lungo}\}$$

1.2. Algoritmo e modello di calcolo

1.2.1. Algoritmo

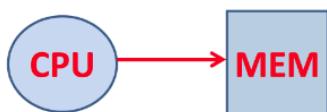
Un *algoritmo* procedura computazionale ben definita che trasforma un dato *input* in un *output* eseguendo una sequenza finita di *operazioni elementari*.

□

L'algoritmo fa riferimento a un *modello di calcolo*, ovvero un'astrazione di computer che definisce l'insieme di operazioni elementari.

Le operazioni elementari sono: assegnamento, operazioni logiche, operazioni aritmetiche, indicizzazione di array, return di un valore da parte di un metodo, ecc.

1.2.2. Modello di calcolo RAM (Random Access Machine)



In questo modello input, output, dati intermedi (e il programma) si trovano in memoria.

Un algoritmo A risolve un *problema computazionale* $\Pi \subseteq I \times S$ se:

1. A calcola una funzione da I a S e quindi,
 - riceve come input istanze $i \in I$
 - produce come output soluzioni $s \in S$
2. Dato $i \in I$, A produce in output $s \in S$ tale che $(i, s) \in \Pi$.
Se Π associa più soluzioni a una istanza i , per tale istanza A ne calcola una (quale dipende da come è stato progettato).

1.3. Pseudocodice

Per semplicità e facilità di analisi, descriviamo un algoritmo utilizzando uno pseudocodice strutturato come segue:

Algoritmo nome (parametri)

Input: breve descrizione dell'istanza di input

Output: breve descrizione della soluzione restituita in output

Descrizione chiara dell'algoritmo tramite costrutti di linguaggi di programmazione e, se utile ai fini della chiarezza, anche tramite linguaggio naturale, dalla quale sia facilmente determinabile la sequenza di operazioni elementari eseguita per ogni dato input.

"Algoritmo nome (parametri)" è la firma (signature) dell'algoritmo; i parametri sono l'input.

Si può usare il linguaggio naturale per cose semplici, che sarebbero lunghe da scrivere con i costrutti del linguaggio di programmazione.

Usare sempre questa struttura per lo pseudocodice!

Per maggiori dettagli fare riferimento alla dispensa sulla [Specifica di Algoritmi in Pseudocodice](#), disponibile sul Moodle del corso.

1.3.1. Esempio – Trasposta di una matrice $n \times n$ di interi

Problema computazionale: $\Pi \subseteq I \times S$

$I \equiv \{A : \text{matrice } n \times n \text{ di interi}\}$

$S \equiv \{B : \text{matrice } n \times n \text{ di interi}\}$

$\Pi \equiv \{(A, B) : A \in I, B \in S, B = A^t\}$

Algoritmo: idea per $n = 4$

a_{00}	a_{01}	a_{02}	a_{03}
a_{10}	a_{11}	a_{12}	a_{13}
a_{20}	a_{21}	a_{22}	a_{23}
a_{30}	a_{31}	a_{32}	a_{33}

Scambio ciascuna entry a_{ij} della matrice triangolare superiore ($a_{ij} : i = 0, \dots, n-2, j > i$) con a_{ji} (l'omologa del triangolo inferiore)

Algoritmo transpose(A)

Input: matrice A $n \times n$ di interi a_{ij} , $0 \leq i, j \leq n$.

Output: matrice A^t .

```
for i <- 0 to n-2 do{
    for j <- i+1 to n-1 do{
        scambia a_ij con a_ji
    }
}
return A
```

1.4. Taglia di un'istanza

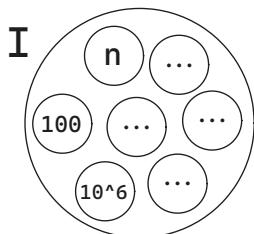
L'analisi di un algoritmo (ad esempio la determinazione della sua complessità) viene solitamente fatta *partizionando le istanze in gruppi in base alla loro taglia (size)*, in modo che le istanze di un gruppo siano tra loro *confrontabili*.

La *taglia di un'istanza* è espressa da uno o più valori che ne caratterizzano la grandezza.

1.4.1. Esempi

- Se ho n elementi da ordinare con un MergeSort, la taglia è n . In questo caso fa riferimento alla dimensione fisica dell'istanza.

La taglia serve per dividere in gruppi omogenei da analizzare separatamente.



- Ordinamento di un array:
Taglia n = numero di elementi dell'array
- Trasposta di una matrice:
Prendendo una generica matrice $n \times m$ ho più modi di definire la taglia, in base al tipo di analisi che si vuole condurre:
 - Taglia $x = n * m$
 - Taglia (n, m) , con n e m rispettivamente il numero di righe e numero di colonne
 - (Se $n = m$) Taglia n = numero di righe/colonne

1.5. Struttura dati

Le strutture dati sono usate dagli algoritmi per organizzare e accedere in modo sistematico ai dati di input e ai dati generati durante l'esecuzione.

1.5.1. Struttura dati - definizione

Una *struttura dati* è una collezione di oggetti corredata di *metodi* di accesso e/o modifica.

Vi sono due *livelli di astrazione*:

1. *Livello logico*: specifica l'organizzazione logica degli oggetti della collezione, e la relazione input-output di ciascun metodo (a questo livello si parla di *Abstract Data Type* o ADT)
2. *Livello fisico*: specifica il layout fisico dei dati e la realizzazione dei metodi tramite algoritmi.

1.5.1.1. Esempio

In Java a livello logico ho (gerarchia di) *interfacce*, mentre a livello fisico ho (gerarchia di) *classi*.

□

1.5.2. Esercizio

Siano A e B due array di n e m interi, rispettivamente. Scrivere un algoritmo in pseudocodice per calcolare il seguente valore:

$$v = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} (A[i] \cdot B[j])$$

1.5.3. Esercizio

Sia A un array di n interi distinti, ordinato in senso crescente, e x un valore intero. Scrivere due algoritmi in pseudocodice che implementano la ricerca binaria per trovare x in A . Entrambi gli algoritmi restituiscono l'indice i tale che $A[i] = x$, se x appare in A , e -1 altrimenti. Il primo algoritmo deve essere iterativo e usare un solo ciclo, e il secondo algoritmo deve essere ricorsivo.

2. Analisi degli algoritmi

L'analisi di un algoritmo A mira a studiarne l'*efficienza* e l'*efficacia*. In particolare, essa può valutare la *complessità* di

tempo e spazio, e la **correttezza** di *terminazione* (se termina o rimane in un loop) e della *soluzione del problema computazionale*.

Noi ci concentreremo sulla complessità di tempo e sulla correttezza della soluzione del problema computazionale.

2.1. Complessità in tempo

L'*obiettivo* è *stimare il tempo di esecuzione ("running time") di un algoritmo* al fine di valutarne l'efficienza e poterlo confrontare con altri algoritmi per lo stesso problema.

2.1.1. Requisiti per la complessità in tempo

La complessità in tempo deve:

1. Riguardare *tutti gli input*;
2. Permettere di *confrontare algoritmi* (senza necessariamente determinare il tempo di esecuzione esatto);
3. Essere *derivabile dallo pseudocodice*.

2.1.1.1. Esempio

Stimare sperimentalmente il tempo di esecuzione di un algoritmo (ad esempio in Java con `System.currentTimeMillis()`) è utile, ma non soddisfa i requisiti. **Perché?**

- Non si possono considerare tutti gli input (se infiniti);
- Richiede di implementare l'algoritmo con un programma e l'impatto dell'implementazione può influenzare il confronto tra algoritmi;
- La stima dipende dall'ambiente hardware/software.

2.1.1.2. Approccio che soddisfa i requisiti

- Analisi al *caso pessimo (worst-case)* in funzione della taglia dell'istanza (requisiti 1 e 2)
- Conteggio delle operazioni elementari nel modello RAM (requisiti 2 e 3)
- Analisi asintotica (per semplificare il conteggio)

💡 Osservazione

Esiste anche l'analisi al caso medio (*average case*) e l'analisi probabilistica.

2.1.2. Complessità (in tempo) al caso pessimo - definizione

Sia A un algoritmo che risolve $\Pi \subseteq I \times S$.

La complessità (in tempo) al caso pessimo di A è una funzione $t_A(n)$ definita come *il massimo numero di operazioni elementari che A esegue per risolvere un'istanza di taglia n* .

In altre parole, se chiamiamo $t_{A,i}$ il numero di operazioni eseguite da A per l'istanza i , abbiamo che

$$t_A(n) = \max\{t_{A,i} : \text{istanza } i \in I \text{ di taglia } n\}$$

La definizione rispetta i tre requisiti definiti sopra.

2.1.3. Difficoltà nel calcolo di $t_A(n)$

Determinare $t_A(n)$ per ogni n è arduo, se non impossibile, perché è difficile identificare l'istanza peggiore di taglia n e perché è difficile contare il massimo numero di operazioni richieste per risolvere tale istanza peggiore (il conteggio richiederebbe anche una specifica dettagliata del set di operazioni elementari del modello RAM).

Ma non è necessario determinare esattamente $t_A(n)$, infatti il tempo di esecuzione, che la complessità vuole stimare, dipende da tanti fattori che è impossibile quantificare in modo preciso, in più le diverse operazioni elementari del modello RAM possono avere impatto diverso sui tempi di esecuzione a seconda delle architetture.

Ci accontentiamo dei *limiti superiori* e i *limiti inferiori* a $t_A(n)$.

2.1.4. Esempio (arrayMax)

Algoritmo arrayMax(A)

Input: array $A[0 \div n - 1]$ di $n \geq 1$ interi.

Output: max intero in A .

```
currMax<-A[0]
for i<-1 to n-1 do{
    if(currMax < A[i]) then currMax<-A[i];
}
return currMax
```

Taglia dell'istanza: n (ragionevole)

2.1.4.1. Stima di $t_{\text{arrayMax}}(n)$

È facile vedere che per una qualsiasi istanza di taglia n :

- al di fuori del ciclo for `arrayMax` esegue un numero costante (rispetto a n) di operazioni;
- in ciascuna delle $n-1$ iterazioni del ciclo for si esegue un numero costante di operazioni.

Esistono allora quattro costanti $c_1, c_2, c_3, c_4 > 0$ tali che per ogni n valga $t_{\text{arrayMax}}(n) \leq c_1 n + c_2$ (cioè il *limite superiore*) e $t_{\text{arrayMax}}(n) \geq c_3 n + c_4$ (cioè il *limite inferiore*). Non è necessario stimare le quattro costanti per le stesse ragioni per cui non è necessario stimare esattamente la complessità.

2.2. Analisi asintotica

Si ricorre quindi all'analisi asintotica, ignorando i *fattori moltiplicativi costanti* (rispetto alla taglia dell'istanza) e i *termini additivi non dominanti*.

2.2.1. Esempio (`arrayMax`)

Riprendendo l'esempio, nel caso di `arrayMax` possiamo affermare che $t_{\text{arrayMax}}(n)$ è *al più* proporzionale a n (limite superiore) ed è anche *almeno* proporzionale a n (limite inferiore), ne consegue quindi che $t_{\text{arrayMax}}(n)$ è *proporzionale* a n (limite stretto).

□

Per esprimere affermazione come quelle fatte sopra si usano gli **ordini di grandezza**: $O(\cdot)$, $\Omega(\cdot)$, $\Theta(\cdot)$, $o(\cdot)$.

Lezione 03

2.3. Ordini di grandezza

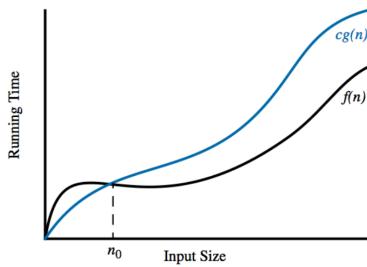
Siano $f(n)$, $g(n)$ funzioni da \mathbb{N} a $\mathbb{R}^+ \cup \{0\}$.

2.3.1. O-grande

$f(n) \in O(g(n))$ se $\exists c > 0$ e $\exists n_0 \geq 1$, costanti rispetto a n , tali che

$$f(n) \leq c \cdot g(n) \quad \forall n \geq n_0$$

Cioè se $f(n)$ è al più proporzionale a $g(n)$, ovvero se $f(n)$ non cresce asintoticamente più di $c \cdot g(n)$.



2.3.1.1. Esempi

$f(n)$	$O(\cdot)$	c	n_0
$3n + 4$ per $n \geq 1$	$O(n)$	4	4
$n + 2n^2$ per $n \geq 1$	$O(n^2)$	3	1
2^{100} per $n \geq 1$	$O(1)$	2^{100}	1
$c_1n + c_2$ per $n \geq 1, c_1, c_2 > 0$	$O(n)$	$c_1 + c_2$	1

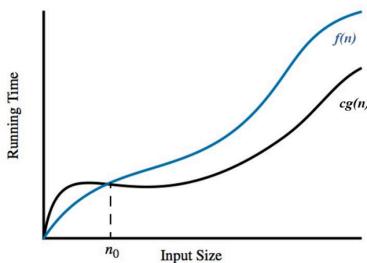
Si può dire che $3n + 4 \in O(n^5)$, $c = 7$, $n_0 = 1$, ma non sarebbe molto utile.

2.3.2. Omega-grande

$f(n) \in \Omega(g(n))$ se $\exists c > 0$ e $\exists n_0 \geq 1$, costanti rispetto a n , tali che

$$f(n) \geq c \cdot g(n) \quad \forall n \geq n_0$$

Cioè $f(n)$ è almeno proporzionale a $g(n)$.



2.3.2.1. Esempi

$f(n)$	$\Omega(\cdot)$	c	n_0
$3n + 4$ per $n \geq 1$	$\Omega(n)$	1	4
$n + 2n^2$ per $n \geq 1$	$\Omega(n^2)$	1	1
2^{100} per $n \geq 1$	$\Omega(1)$	2^{100}	1
$c_1n + c_2$ per $n \geq 1, c_1, c_2 > 0$	$\Omega(n)$	1	1

2.3.3. Theta

$f(n) \in \Theta(g(n))$ se

$$f(n) \in O(g(n)) \text{ e } f(n) \in \Omega(g(n))$$

Cioè $f(n)$ è (*esattamente*) proporzionale a $g(n)$.

2.3.3.1. Esempi

- $f(n) = 3n + 4 \in \Theta(n)$;
- $f(n) = n + 2n^2 \in \Theta(n^2)$;
- $f(n) = 2^{100} \in \Theta(1)$;
- $t_{\text{arrayMax}}(n) \in \Theta(n)$.

2.3.4. o-piccolo

$f(n) \in o(g(n))$ se

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

Cioè $f(n)$ è *asintoticamente* più piccola (cresce meno) di $g(n)$.

2.3.4.1. Esempi

- $f(n) = 100n$ per $n \geq 1 \implies f(n) \in o(n^2)$;
- $f(n) = \frac{3n}{\log_2 n}$ per $n \geq 1 \implies f(n) \in o(n)$.

2.3.5. Proprietà degli ordini di grandezza

1. $\max\{f(n), g(n)\} \in \Theta(f(n) + g(n))$ per ogni $f(n), g(n) : \mathbb{N} \rightarrow \mathbb{R}^+ \cup \{0\}$;

2. $\sum_{i=0}^k a_i n^i \in \Theta(n^k)$, se $a_k > 0$, k, a_i costanti, e $k \geq 0$.
Ad esempio: $(n+1)^5 \in \Theta(n^5)$.

Cioè tengo il termine con l'esponente maggiore;

3. $\log_b n \in \Theta(\log_a n)$, se $a, b > 1$ sono costanti;

! Osservazione

La proprietà deriva dalla relazione $\log_b n = (\log_a n)(\log_b a)$ e, grazie a essa, *la base dei logaritmi*, se costante, si omette *negli ordini di grandezza*, a meno che il logaritmo non sia all'esponente.

4. $n^k \in o(a^n)$, se $k > 0$, $a > 1$ sono costanti;

5. $(\log_b n)^k \in o(n^h)$ se b, k, h sono costanti, con $b > 1$ e $h, k > 0$.

2.3.6. Strumenti matematici

2.3.6.1. Parte bassa

$\forall x \in \mathbb{R}$, si definisce $\lfloor x \rfloor$ come il più grande intero tale che sia $\leq x$.

2.3.6.1.1. Esempi

- $\lfloor \frac{3}{2} \rfloor = 1$;
- $\lfloor 3 \rfloor = 3$.

2.3.6.2. Parte alta

$\forall x \in \mathbb{R}$, si definisce $\lceil x \rceil$ come il più piccolo intero tale che sia $\geq x$.

2.3.6.2.1. Esempi

- $\lceil \frac{3}{2} \rceil = 2$;
- $\lceil 3 \rceil = 3$.

2.3.6.3. Modulo

$\forall x, y \in \mathbb{Z}$, con $y \neq 0$, si definisce $x \bmod y$ come il resto della divisione intera x/y (l'operatore "%" in Java).

2.3.6.3.1. Esempi

- $29 \bmod 7 = 4$;
- $80 \bmod 4 = 0$.

2.3.6.4. Sommatorie notevoli

$\forall n \in \mathbb{Z}$ e $a \in \mathbb{R}$, con $n \geq 0$ e $a > 0$ vale:

- $\sum_{i=0}^n i = \frac{n(n+1)}{2} \in \Theta(n^2)$;
- $\sum_{i=0}^n a^i = \frac{a^{n+1}-1}{a-1} \in \Theta(a^n)$ per $a > 1$;
- $\sum_{i=1}^n a^i = \frac{a^{n+1}-1}{a-1} - 1 \in \Theta(a^n)$ per $a > 1$.

2.4. Analisi di complessità in pratica

Dato un algoritmo A e detta $t_A(n)$ la sua complessità al caso pessimo, si cercano limiti asintotici superiori e/o inferiori a $t_A(n)$.

2.4.1. Limite superiore (upper bound) – definizione

$$t_A(n) \in O(f(n))$$

Si prova argomentando che per ogni n "abbastanza grande" e per ciascuna istanza di taglia n l'algoritmo esegue $\leq c \cdot f(n)$ operazioni, con c costante (e che non serve determinare).

2.4.2. Limite inferiore (lower bound) – definizione

$$t_A(n) \in \Omega(f(n))$$

Si prova argomentando che per ogni n "abbastanza grande", esiste un'istanza di taglia n per la quale l'algoritmo esegue $\geq c \cdot f(n)$ operazioni, con c costante (e che non serve determinare).

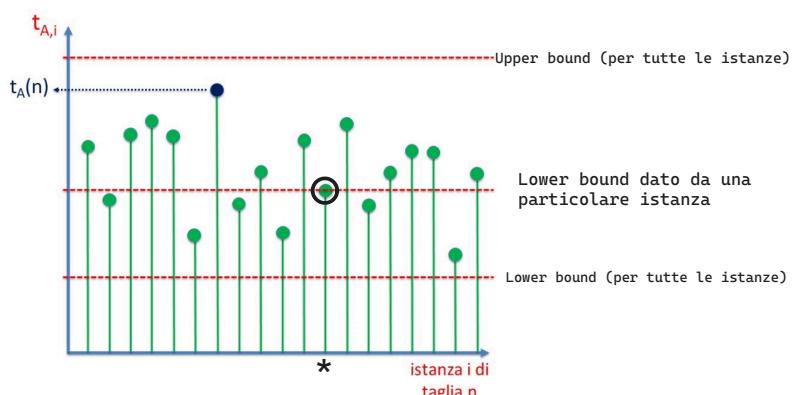
In alcuni casi è comodo argomentare che per ciascuna istanza di taglia n l'algoritmo esegue $\geq c \cdot f(n)$ operazioni.

⚠ Attenzione

Sia che si provi $t_A(n) \in O(f(n))$ o che si provi $t_A(n) \in \Omega(f(n))$

- $f(n)$ deve essere più vicino possibile alla complessità vera (*tight bound*)
- $f(n)$ deve essere più semplice possibile, quindi senza costanti e termini additivi di ordine inferiore, solo con i *termini essenziali*!

2.4.3. Limiti superiori e inferiori



2.4.4. Terminologia per complessità

- logaritmica: $\Theta(\log n)$, base 2 o costante > 1
- lineare: $\Theta(n)$
- quadratica: $\Theta(n^2)$
- cubica: $\Theta(n^3)$
- polinomiale $\Theta(n^c)$, $c > 0$ costante

- esponenziale: $\Omega(a^n)$, $a > 1$ costante
- polilogaritmica: $\Theta((\log n)^c)$, $c > 0$ costante

2.4.5. Esempio (prefix averages)

Si consideri il seguente problema computazionale.

Dato un array di n interi $X[0 \dots n-1]$ calcolare un array $A[0 \dots n-1]$ dove $A[i] = \left(\sum_{j=0}^i X[j]\right) \frac{1}{i+1}$, per $0 \leq i < n$.

Vedremo adesso due algoritmi di cui uno banale e inefficiente, poi uno più furbo ed efficiente. Per entrambi gli algoritmi vale la seguente specifica di input-output:

Input: $X[0 \dots n-1]$ array di n interi.

Output: $A[0 \dots n-1] : A[i] = \left(\sum_{j=0}^i X[j]\right) \frac{1}{i+1}$ per $0 \leq i < n$.

2.4.5.1. Algoritmo inefficiente

Algoritmo prefixAverages1

```

for i <- 0 to n-1 do{
    a <- 0;
    for j <- 0 to i do {
        a <- a+X[j];
    }
    A[i] <- a/(i+1);
}
return A

```

- Fuori dal for esterno: $\Theta(1)$ operazioni
- Per ciascuna iterazione i del for esterno ($i = 0, \dots, n-1$) :
 - $\Theta(i+1)$ operazioni nel for interno
 - $\Theta(1)$ altre operazioni

La complessità è quindi:

$$t_{pA1}(n) \in \Theta\left(1 + \sum_{i=0}^{n-1} i + 1\right) = \Theta\left(\sum_{i=0}^{n-1} i\right) = \Theta\left(\frac{(n-1)n}{2}\right) = \Theta(n^2).$$

2.4.5.2. Algoritmo efficiente

Algoritmo prefixAverages2

```

s <- 0;
for i <- 0 to n-1 do{
    s <- s + X[i];
    A[i] <- s/(i+1);
}

```

```

    }
    return A

```

- Fuori dal for: $\Theta(1)$ operazioni
- Iterazioni i del for ($i = 0, \dots, n - 1$): $\Theta(1)$ operazioni

La complessità è quindi: $t_{pA2}(n) \in \Theta\left(1 + \sum_{i=0}^{n-1} 1\right) = \Theta(n)$

Possiamo affermare che $t_{pA2}(n) \in o(t_{pA1}(n))$, cioè è migliore.

Lezione 04

2.4.6. Esercizi

2.4.6.1. Analisi complessità in tempo

Sia A un algoritmo che ricevuto in input un array X di $n \geq 1$ interi esegue

- $c_1 n$ operazioni per ogni intero pari in X
- $c_2 \lceil \log_2 n \rceil$ operazioni per ogni intero dispari in X
dove c_1 e c_2 sono costanti positive.

Analizzare la complessità in tempo dell'algoritmo esprimendola con $\Theta(\cdot)$.

Per un'istanza particolare (es: array di tutti interi pari) ho $\Theta(n^2)$, ho quindi ricavato un lower bound.

- Taglia dell'istanza: n
- Complessità: $t_A(n)$
- Upper bound: \forall istanza di taglia n , A esegue
 $\leq n \max\{c_1 m, c_2 \lceil \log_2 n \rceil\} \leq n \max\{c_1, c_2\} \max\{n, \log_2 n\} = n \max\{c_1, c_2\} n$ operazioni
 $\implies t_A(n) \in O(n^2)$ (1)
- Lower bound basato su una particolare istanza:
Sia X un array di interi pari \implies per tale X , A esegue
 $n \cdot c_1 \cdot n = c_1 n^2$ operazioni $\implies t_A(n) \in \Omega(n^2)$ (2)
- Lower bound basato su tutte le istanze:
 \forall istanza di taglia n , A esegue
 $\geq n \min\{c_1 n, c_2 \lceil \log_2 n \rceil\} \geq n \min\{c_1, c_2\} \min\{n, \log_2 n\} = n \min\{c_1, c_2\} \lceil \log_2 n \rceil$ operazioni $\implies t_A(n) \in \Omega(n \log n)$
Questo lower bound è peggiore rispetto a quello trovato prima.

2.4.6.1. Descrizione insertion sort

Il seguente pseudocodice descrive l'algoritmo di ordinamento chiamato InsertionSort (Si assume che la sequenza S da ordinare sia una variabile globale che dopo la fine dell'algoritmo è accessibile e ordinata).

Algoritmo InsertionSort(S)

Input: Sequenza $S[0 \dots n - 1]$ di n chiavi.

Output: Sequenza S ordinata in senso crescente.

```
for i<-1 to n-1 do{
    curr<-S[i];
    j<-i-1;
    while((j>=0) AND (S[j] > curr)) do{
        S[j+1]<-S[j];
        j<-j-1;
    }
    S[j+1]<-curr;
}
```

Notazione

Nel caso degli algoritmi di ordinamento, in analogia al libro di testo, usiamo il termine *sequenza* per denotare la collezione di elementi da ordinare che assumiamo rappresentata come array.

1. Trovare opportune funzioni $f_1(n)$, $f_2(n)$, $f_3(n)$ tali che le seguenti affermazioni siano vere, per una qualche costante $c > 0$ e per n abbastanza grande.
 - Per ciascuna istanza di taglia n l'algoritmo esegue $\leq cf_1(n)$ operazioni.
 - Per ciascuna istanza di taglia n l'algoritmo esegue $cf_2(n)$ operazioni.
 - Esiste un'istanza di tagli a n per la quale l'algoritmo esegue $\geq cf_3(n)$ operazioniLa funzione $f_1(n)$ deve essere la più piccola possibile, mentre le funzioni $f_2(n)$ e $f_3(n)$ devono essere le più grandi possibili.
2. Sia $t_{IS}(n)$ la complessità al caso pessimo dell'algoritmo. Sfruttando le affermazioni del punto precedente trovare un upper bound $O(\cdot)$ e un lower bound $\Omega(\cdot)$ per $t_{IS}(n)$.

2.4.7. Regola di buon senso

Una complessità polinomiale (o migliore) implica un algoritmo efficiente, mentre una complessità esponenziale implica un algoritmo inefficiente.

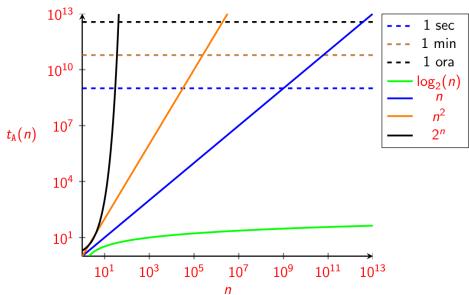
Supponiamo che la complessità $t_A(n)$ sia espressa in nanosecondi e definiamo

$$n_\tau \equiv \max \text{ taglia di un'istanza risolvibile in tempo } \tau$$

Per ottenere n_τ , risolviamo $t_A(n_\tau) = \tau$ rispetto a n_τ .

2.4.7.1. Esempi

$t_A(n)$	n_τ per $\tau = 10^9$ (1 sec)	n_τ per $\tau = 60 * 10^9$ (1 min)	n_τ per $\tau = 3600 * 10^9$ (1 ora)
$\log_2 n$	$2^{10^9} = \infty$	∞	∞
n	10^9	$6 * 10^{10}$	$3.6 * 10^{12}$
n^2	$10^{4.5}$	$\approx 8 * 10^{4.5}$	$\approx 1.8 * 10^6$
2^n	≈ 30	≈ 36	≈ 42



2.4.7.2. Esempio

La crittografia a chiave pubblica è molto usata dai protocolli di sicurezza.

L'invio di un messaggio cifrato da Alice a Bob funziona (in maniera approssimata) in questo modo:

- Bob possiede una chiave privata k_1 e una chiave pubblica k_2 ;
- Alice invia a Bob un messaggio m cifrato con k_2 ;
- Bob decifra il messaggio ricevuto da Alice con k_1 ;

L'*Algoritmo RSA* sviluppato nel 1977 da Rivest, Shamir, Adleman (*Turing Award 2002*) è il più famoso algoritmo di crittografia a chiave pubblica. La sua "sicurezza" si basa sulla difficoltà di risolvere il seguente problema.

2.4.7.2.1. Integer factorization (per interi prodotti di 2 primi)

Dato un intero N prodotto di due primi p, q , determinare p e q .

Che taglia dell'istanza scelgo?

Posso scrivere, ad esempio, questo algoritmo banale per risolvere il problema:

```
for p<-2 to floor{sqrt{N}} do{
    if(N mod p = 0) then return {p, N/p};
}
```

Esprimiamo la complessità in funzione del numero n bit che servono per rappresentare N . Se $p, q \in \Theta(\sqrt{N})$ la complessità è $\Theta(\sqrt{N}) = \Theta(2^{n/2})$

2.4.7.2.1.1. Esempio numerico

Prendendo $n = 1024$ ottengo $2^{n/2} = 2^{512} \geq 10^{154}$. Sul computer più potente del mondo ($< 10^{19}$ flop/sec) l'algoritmo banale richiederebbe circa $\frac{10^{154}}{10^{19}} = 10^{135}$ secondi. Considerando che un anno ha meno di 10^8 secondi, sarebbero più di 10^{127} anni di calcolo.

💡 Nota bene

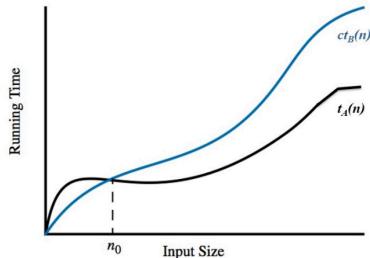
Esistono algoritmi più efficienti, ma sempre con complessità esponenziale.

↳ Dalla motivazione del A.M. Turning Award 2002 assegnato a Rivest, Shamir e Adleman

RSA is used in almost all internet-based commercial transactions. Without it, commercial online activities would not be as widespread as they are today. It allows users to communicate sensitive information like credit card numbers over an unsecure internet without having to agree on a shared secret key ahead of time. Most people ordering items over the internet don't know that the system is in use unless they notice the small padlock symbol in the corner of the screen. RSA is a prime example of an abstract elegant theory that has had great practical application.

2.4.8. Efficienza asintotica degli algoritmi

Dati A , B due algoritmi che risolvono il problema computazionale Π , $t_A(n)$ e $t_B(n)$ sono le complessità rispettivamente di A e B al caso pessimo. Se $t_A(n) \in o(t_B(n))$, allora A è "asintoticamente più efficiente" di B .



Nota bene: n_0 potrebbe essere *molto* grande.

⚠️ Caveat sull'analisi asintotica al caso pessimo

1. *Le costanti trascurate potrebbero essere elevate e, in pratica, potrebbero avere un impatto elevato sulle prestazioni.*
2. *Il caso pessimo potrebbe essere costituito solo da istanze patologiche* mentre per tutte le istanze di interesse la complessità potrebbe essere asintoticamente migliore.
Cosa fare in questo caso?
 - Restringere il dominio delle istanze, mantenendo quelle di interesse ed escludendo quelle patologiche.
 - Fare un'analisi al caso medio.

2.5. Analisi di correttezza

2.5.1. Induzione

Per provare che una proprietà $Q(n)$ è vera $\forall n \geq n_0$ si procede così:

- Si sceglie un intero $k \geq 0$;
- **Base:** si dimostra $Q(n_0)$, $Q(n_0 + 1)$, ..., $Q(n_0 + k)$;
- **Passo induttivo:** si fissa un valore $n \geq n_0 + k$ arbitrario e si dimostra che $Q(m)$ vera $\forall m : n_0 \leq m \leq n \implies Q(n+1)$ vera.

➊ Osservazioni

- " $Q(m)$ vera $\forall m : n_0 \leq m \leq n$ " è chiamata *ipotesi induttiva*.

- La dimostrazione deve valere *per ogni* $n \geq n_0 + k$
- Di solito, ma non sempre, $k = 0$ (il libro di testo descrive l'induzione con $n_0 = 1$) .

2.5.1.1. Esempio

$$Q(n) : \sum_{i=0}^n i = \frac{n(n+1)}{2} \quad \forall n \geq 0$$

Nota bene: Implica che $Q(n) \in \Theta(n^2)$.

- $n_0 = 0, k = 0$
- Base: $Q(0) : \sum_{i=0}^0 i = 0 = \frac{0 \cdot 1}{2} \quad \checkmark$
- Passo induttivo: Fisso $n \geq n_0 + k = 0$ arbitrario.
Ipotesi induttiva: $\sum_{i=0}^m i = \frac{m(m+1)}{2} \quad \forall 0 \leq m \leq n$
Dimostriamo che $Q(n+1)$ è vera:

$$\begin{aligned} \sum_{i=0}^{n+1} i &= n+1 + \sum_{i=0}^n i = n+1 \frac{n(n+1)}{2} = \text{per ipotesi induttiva} \\ &= \frac{(n+1)(n+2)}{2} \implies Q(n+1) \text{ è vera} \end{aligned}$$

2.5.1.2. Esercizio

Dimostrare per induzione la seguente proprietà:

$$Q(n) : \sum_{i=0}^n i^2 = \frac{n(n+1)(2n+1)}{6} \quad \forall n \geq 0$$

Nota bene: Implica che $\sum_{i=0}^n i^2 \in \Theta(n^3)$

① Osservazione

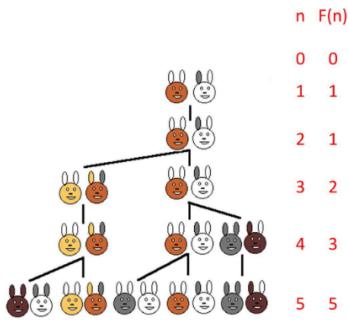
In generale $\sum_{i=0}^n i^k \in \Theta(n^{k+1})$, con k costante.

Lezione 05

2.5.1.3. Successione di Fibonacci - esempio

La *successione di Fibonacci* è definita così:
$$\begin{cases} F(0) = 0 \\ F(1) = 1 \\ F(n+1) = F(n) + F(n-1) \end{cases} \quad \forall n \geq 1.$$

Si può pensare a $F(n)$ come il numero di coppie di conigli all'inizio del mese n se una coppia genera un'altra coppia ogni mese, a partire dal terzo mese, i conigli non muoiono, all'inizio del mese 1 c'è una coppia neonata.



Dimostrare che

$$F(n) = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n) \quad (\implies F(n) \in \Theta(N\phi^n))$$

$\forall n \geq 0$, dove $\phi = \frac{1+\sqrt{5}}{2}$ (*golden ratio*) e $\hat{\phi} = \frac{1-\sqrt{5}}{2}$.

2.5.1.3.1. Induzione fallace

Dimostriamo $F(n) = 0$, $\forall n \geq 0$ ($n_0 = 0$):

- $k = 0$
- Base: $F(0) = 0 \implies \checkmark$
- Passo induttivo: fissato $n \geq 0$ e assumendo $F(m) = 0$ per ogni $0 \leq m \leq n$ (ipotesi induttiva), si ha $F(n+1) = F(n) + F(n-1) = 0 + 0$

Il passo induttivo non è corretto quando $n = 0$, perché $F(n+1) = F(n) + F(n-1)$ vale solo per $n \geq 1$. Allora in questo caso devo usare una base più ampia: $n_0 = 0$, $k = 1$.

2.5.2. Correttezza

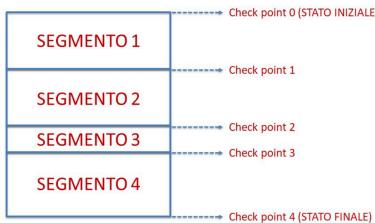
Per provare la correttezza di terminazione, cioè appunto la terminazione, è necessario assicurarsi che i cicli (inclusi i GOTO) e l'eventuale ricorsione abbiano termine.

Gli step dell'approccio generale alla soluzione del problema computazionale sono:

- *Definire lo stato iniziale* dell'algoritmo e quello *finale* che esso *dove raggiungere*;
- *Decomporre l'algoritmo in segmenti* e definire per ogni segmento lo stato in cui l'algoritmo si deve trovare del termine del segmento (*checkpoint*);
- *Dimostrare* che a partire dallo stato iniziale *si raggiungono* in successione *gli stati specificati* per la fine di ogni segmento. In particolare, lo stato che deve valere alla fine dell'ultimo

segmento deve coincidere (o implicare) lo stato finale desiderato.

I *segmenti notevoli* sono cicli (for, while, repeat-until).



⚠️ Osservazione

I cicli sono i segmenti più difficili da analizzare (perché definiscono *implicitamente* molte operazioni) ma, insieme alla ricorsione, costituiscono gli strumenti essenziali per la scrittura di algoritmi o programmi interessanti.

2.5.3. Invariante

Per provare la correttezza di un ciclo (for, while, repeat-until, ...) bisogna dimostrare che al termine della sua esecuzione vale una certa *proprietà \mathcal{L}* che rappresenta uno stato ed è *funzionale alla correttezza dell'algoritmo*. A tal fine si fa uso di un *invariante*.

Un *invariante* per un ciclo è una *proprietà* espressa in funzione delle variabili usate nel ciclo, che descrive lo *stato* in cui si trova l'esecuzione alla fine di una generica iterazione del ciclo.

💡 Nota bene

L'invariante deve essere scelto finalizzandolo alla correttezza e alla prova della proprietà \mathcal{L} .

2.5.4. Correttezza di un ciclo tramite invariante

Per dimostrare che alla fine del ciclo vale una certa proprietà \mathcal{L} , si individua un opportuno invariante e si dimostra che:

1. Esso *vale all'inizio del ciclo* (subito prima che il ciclo inizi);
2. Esso *vale alla fine di ciascuna iterazione*, che si dimostra induttivamente provando che se vale alla fine di una generica iterazione, vale alla fine della successiva;

- Alla fine dell'ultima iterazione, l'invariante implica la proprietà \mathcal{L} (in alcuni casi coincide con essa).

∅ Terminologia

L'esecuzione di un *ciclo* consiste di più di zero iterazioni delle istruzioni in esso contenute (*corpo del ciclo*).

Ad esempio `for i<-1 to n do {corpo del ciclo}` è un ciclo che esegue sempre n iterazioni.

2.5.4.1. Esempio (arrayMax)

Algoritmo arrayMax(A)

Input: Array $A[0 \dots n-1]$ di $n \geq 1$ interi.

Output: massimo intero di A .

```
currMax<-A[0];
for i<-1 to n-1 do{
    currMax<-max{currMax, A[i]};
}
return currMax;
```

Proprietà \mathcal{L} : alla fine del ciclo, $currMax$ è il massimo intero in A (quindi il valore restituito è corretto).

Invariante: Alla fine della iterazione $i \geq 0$,

$currMax = \max\{A[0], A[1], \dots, A[i]\}$.

! Osservazione

La fine dell'iterazione $i=0$ corrisponde all'inizio del ciclo.

- All'inizio del ciclo ($i=0$) l'invariante è reso vero dall'assegnamento $currMax <- A[0]$;
- Disponiamo l'invariante uguale a vero a fine iterazione $i < n-1 \implies currMax = \max\{A[0], \dots, A[i]\}$ e dimostriamo che vale anche alla fine della successiva iterazione.
Nella iterazione $i+1$ l'istruzione $currMax <- \max\{currMax, A[i+1]\}$ assicura che alla fine di tale iterazione
 $currMax <- \max\{A[0], \dots, A[i+1]\}$;
- Alla fine dell'ultima iterazione ($i = n-1$):
Se vale l'invariante, cioè se $currMax = \max\{A[0], \dots, A[n-1]\}$, allora $currMax$ è effettivamente il massimo intero, che è la proprietà \mathcal{L} .

2.5.4.2. Esempio (`arrayFind`)

Algoritmo `arrayFind(A)`

Input: Elemento x , array $A[0 \div n - 1]$ di n elementi.

Output: indice $i \in [0, n)$ tale che $A[i] = x$, se esiste, altrimenti -1 .

```
i <- 0;
while i < n do{
    if(x = A[i]) then return i;
    else i <- i+1;
}
return -1;
```

Proprietà \mathcal{L} : Il valore trovato è corretto.

Invariante: Alla fine di una generica iterazione $x \neq A[j] \quad \forall 0 \leq j < i$ con i valore della variabile omonima alla fine dell'iterazione corrente.

1. All'inizio del ciclo ($i=0$) l'invariante vale per **vacuità** dato che il range $0 \div i-1$ è vuoto;
2. Supponiamo l'invariante vero alla fine di una generica iterazione $\implies x \neq A[j] \quad \forall 0 \leq j < i < n$.

Alla fine della successiva iterazione:

- Se $x = A[i] \implies i$ non cambia e si esce dal ciclo;
- Se $x \neq A[i] \implies i$ diventa $i+1$.

In entrambi i casi l'invariante continua a valere;

3. Fine ultima iterazione, ci sono due possibili uscite:

- $u_1 : x = A[i]$: In questo caso si restituisce i e chiaramente \mathcal{L} vale;
- $u_2 : i = n$ in questo caso si restituisce -1 .
Dato che l'invariante mi dice che $x \neq A[j] \quad \forall j : 0 \leq j < n = i$ so che x non è in A e quindi restituire -1 è corretto $\implies \mathcal{L}$ vale.

2.5.6. Ricorsione

Un **algoritmo ricorsivo** è un algoritmo che invoca sé stesso (su istanze sempre più piccole) sfruttando la nozione di induzione.

La soluzione di un'istanza di taglia n è ottenuta **direttamente** se $n = n_0, n_0 + 1, \dots, n_0 + k$ (caso base), altrimenti **riducendosi** alla soluzione di $r \geq 1$ istanze di taglia minore di n : se $n > n_0 + k$. Se $r = 1$ si parla di **linear recursion**.

2.5.6.1. Esempi

Algoritmo linearSum(A, n)

Input: array A , intero $n \geq 1$.

Output: $\sum_{i=0}^{n-1} A[i]$.

```
if (n = 1) then{
    return A[0];
}
else{
    return linearSum(A, n-1) + A[n-1];
}
```

Taglia dell'istanza: $n; n_0 = 1$ e $k = 0$.

Se voglio trovare la somma di tutti gli elementi di un array A , la prima invocazione sarà `linearSum(A, |A|)`.

Algoritmo reverseArray(A, i, j)

Input: array A , indici $i, j \geq 0$.

Output: array A con gli elementi in $A[i:j]$ ribaltati.

```
if(i < j) then{
    swap(A[i], A[j]);
    reverseArray(A, i+1, j-1);
}
return
```

Taglia dell'istanza: $n = j - i + 1$.

Caso base: $n \leq 1$.

La prima invocazione per ribaltare tutto A è `reverseArray(A, 0, |A|-1)`.

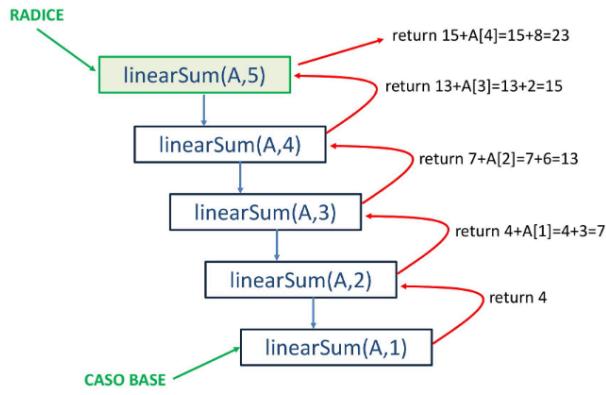
Lezione 06

2.5.6.2. Esecuzione di un algoritmo ricorsivo

All'esecuzione di un algoritmo ricorsivo su una data istanza è associato un *albero della ricorsione* (o *recursion trace*) tale che:

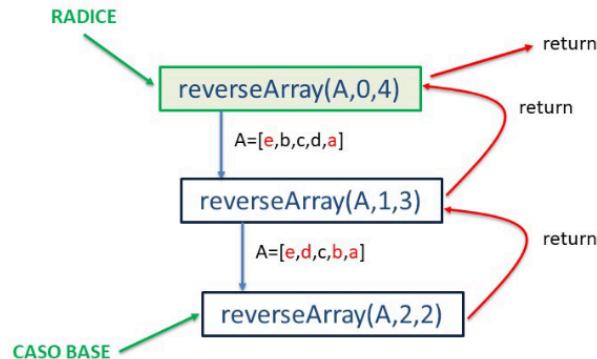
- Ogni nodo corrispondente a un'invocazione ricorsiva distinta fatta durante l'esecuzione dell'algoritmo;
- La *radice* dell'albero corrisponde alla prima invocazione, i *figli* di un nodo x sono associati alle invocazioni ricorsive fatte direttamente dall'invocazione corrispondente a x ;
- Le *foglie* dell'albero rappresentano i *casi base*.

L'albero della ricorsione per `linearSum(A, 5)`, con $A = [4, 3, 6, 2, 8]$ risulta:



Avvengono un numero costante di operazioni per chiamata, quindi il numero totale sarà proporzionale al numero di chiamate. La complessità è allora $\Theta(n)$ (giustificazione a seguire).

L'albero della ricorsione per `reverseArray(A, 0, 4)`, con $A = [a, b, c, d, e]$ sarà invece:

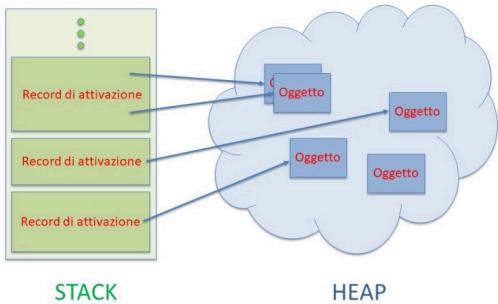


Alla fine dell'ultima invocazione si ha $A = [e, d, c, b, a]$

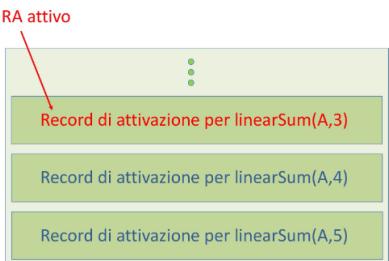
Nell'esecuzione di un programma (in Java) entrano di solito in gioco die spazi di memoria:

- **Stack**: spazio destinato a memorizzare variabili locali ai metodi e riferimenti a oggetti.
 - Per ogni invocazione di un metodo viene inserito un *record di attivazione* (abbreviato in RA; in inglese si usa il termine *activation record* o *activation frame*) contenente variabili e riferimenti a oggetti relativi a quell'invocazione.
 - Un RA viene eliminato quando l'invocazione del metodo corrispondente finisce l'esecuzione.
 - I RA sono inseriti/eliminati con politica LIFO (Last In First Out); in ogni istante possono essere acceduti i dati relativi all'ultimo RA inserito, ma non quelli di altri RA.

- *Heap*: spazio destinato a memorizzare gli oggetti.



Supponiamo di eseguire `linearSum(A, 5)` e consideriamo l'invocazione ricorsiva `linearSum(A, 3)`. Quando viene creato il RA per tali invocazione ricorsiva, lo stato della Stack è il seguente:



Ricorda

Un algoritmo ricorsivo è un algoritmo che invoca sé stesso su istanze più piccole.

Un *algoritmo iterativo* è un algoritmo non-ricorsivo.

Osservazione

Il termine *iterativo* è usato per evidenziare che (tranne per casi banali) *un algoritmo non-ricorsivo fa uso di cicli per poter elaborare istanze di qualsiasi taglia*.

Mentre i cicli sono essenziali in un algoritmo iterativo, un algoritmo ricorsivo può non avere cicli (come `linearSum` e `reverseArray`), ma può anche averli (come `MergeSort`).

2.5.7. Complessità di algoritmi ricorsivi

La *complessità* di un algoritmo ricorsivo A può essere *stimata tramite l'Albero della Ricorsione*.

Consideriamo l'albero associato all'esecuzione di A su un'istanza i di taglia n :

- A ogni nodo è attribuito un *costo* pari al numero di operazioni eseguite dall'invocazione corrispondente a quel nodo, *escluse quelle fatte dalle invocazioni ricorsive al suo interno*;
- Il numero totale di operazioni eseguite da A per risolvere i si ottiene sommando i costi associati a tutti i nodi.
Per ottenere un *upper bound a $t_A(n)$* si ricava una stima per eccesso del numero totale di operazioni che valga per tutte le istanze i di taglia n , mentre per ottenere un *lower bound a $t_A(n)$* si trova un'istanza particolare o si fa una stima inferiore che vada bene per tutte le istanze.

2.5.7.1. Esempi

2.5.7.1.1. Complessità di `linearSum(A, n)`

Algoritmo `linearSum(A, n)`

Input: array A , intero $n \geq 1$.

Output: $\sum_{i=0}^{n-1} A[i]$.

```
if (n = 1) then{
    return A[0];
}
else{
    return linearSum(A, n-1) + A[n-1];
}
```

L'albero della ricorsione associato a una generica istanza di taglia n :

- n nodi associati a `linearSum(A, j)`, $j = n - 1, \dots, 1$;
- Costo associato a ciascun nodo: $\Theta(1)$ operazioni (esclude quello delle chiamate ricorsive)-
 - ⇒ Ogni istanza di taglia n richiede $\Theta(n)$ operazioni.
 - ⇒ $t_{\text{linearSum}}(n) \in \Theta(n)$ è la complessità al caso pessimo.

2.5.7.1.2. Complessità di `reverseArray(A, i, j)`

Algoritmo `reverseArray(A, i, j)`

Input: array A , indici $i, j \geq 0$.

Output: array A con gli elementi in $A[i:j]$ ribaltati.

```
if(i < j) then{
    swap(A[i], A[j]);
    reverseArray(A, i+1, j-1);
}
```

```

}

return

```

L'albero della ricorsione associato a una generica istanza di taglia n ($n = j - i + 1$) :

- $\lfloor \frac{n}{2} \rfloor + 1$ nodi associati;
- Costo associato a ciascun nodo: $\Theta(1)$ operazioni (esclude quello delle chiamate ricorsive).
 - ⇒ Ogni istanza di taglia n richiede $\Theta(n)$ operazioni.
 - ⇒ $t_{\text{reverseArray}}(n) \in \Theta(n)$ è la complessità al caso pessimo.

2.5.7.2. Calcolo efficiente di potenze

Dato $x \in \mathbb{R}$ e $n \geq 0$ intero, calcolare $p(x, n) = x^n$.

È fondamentale osservare che

$$p(x, n) = \begin{cases} 1 & n = 0 \\ x \cdot p(x, \frac{n-1}{2})^2 & n > 0 \text{ dispari} \\ p(x, \frac{n}{2})^2 & n > 0 \text{ pari} \end{cases}$$

Algoritmo power(x, n)

Input: $x \in \mathbb{R}$ e $n \geq 0$.

Output: $p(x, n)$.

```

if (n == 0) then{
    return 1;
}
if (n è dispari) then{
    y <- power(x, (n-1)/2);
    return x*y*y;
}
else{
    y <- power(x, n/2);
    return y*y;
}

```

2.5.7.2.1. Complessità di power(x, n)

Supponiamo $n \geq 1$ (consideriamo n come taglia dell'istanza) :

- Alla i -esima chiamata ricorsiva l'algoritmo viene invocato per un esponente $n_i \leq \frac{n}{2^i}$. Di conseguenza l'albero della ricorsione avrà $O(\log n)$ nodi;

- Ogni invocazione di `power` esegue $\Theta(1)$ operazioni, esclusa l'eventuale chiamata ricorsiva. Quindi il costo associato a ciascun nodo è $\Theta(1)$.
 $\Rightarrow t_{\text{power}}(n) \in O(\log n)$.

2.5.7.2.2. Applicazione di `power` al calcolo di $F(n)$

Il seguente algoritmo efficiente sfrutta la formula $F(n)\frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n)$ e l'algoritmo `power` visto in precedenza:

Algoritmo `powerFib(n)`

Input: intero $n \geq 0$.

Output: $F(n)$.

```
ψ <- ((1 + sqrt{5}) / 2);
ψ^ <- ((1 - sqrt{5}) / 2);
return (power(ψ, n) - power(ψ^, n)) / sqrt{5};
```

Da quanto provato per `power`, otteniamo che la complessità di `powerFib` è $O(\log n)$.

2.5.8. Correttezza di algoritmi ricorsivi

Per provare la correttezza (o una qualsiasi proprietà) di un algoritmo ricorsivo A si ricorre all'induzione.

Sia n la taglia dell'istanza:

- Si dimostra la correttezza per i casi vase $n \in [n_0, n_0 + k]$;
- Supponendo che A risolva correttamente tutte le istanze di taglia $m \in [n_0, n]$, per un qualche $n \geq n_0 + k$, si dimostra che esso risolve correttamente tutte le istanze di taglia $n + 1$.

2.5.8.1. Correttezza di `linearSum(A, n)` per $n \geq 1$

Caso base ($n = 1$): correttezza banale.

Passo induttivo: Fisso $n \geq 1$ arbitrario.

Ipotesi induttiva: `linearSum(A, j)` sia corretto $\forall A, \forall 1 \leq j \leq n$.

Considero `linearSum(A, n+1)` con A array di $\geq n + 1$ elementi:

`linearSum(A, n+1)` restituisce $\text{linearSum}(A, n) + A[n] = \sum_{i=0}^{n-1} A[i] + A[n] = \sum_{i=0}^n A[i]$ per ipotesi induttiva.

\Rightarrow Il valore restituito è corretto.

 **Riepilogo sulle nozioni di base**

- Nozioni di: problema computazionale, algoritmo (che risolve un problema computazionale), taglia dell'istanza, struttura dati;
- Specifica di un algoritmo tramite pseudocodice;
- Complessità al caso pessimo di un algoritmo:
 - Definizione;
 - Analisi asintotica espressa tramite ordini di grandezza ($O(\cdot)$, $\Omega(\cdot)$, $\Theta(\cdot)$);
- Tecniche di dimostrazione: esempio, controesempio, per assurdo, induzione;
- Invarianti e loro uso per provare la correttezza di cicli;
- Algoritmi ricorsivi e loro analisi:
 - Complessità: tramite l'albero della ricorsione o tramite guess e induzione;
 - Correttezza: tramite induzione.

[lezione 07 & 08] – svolgimento esercizi

[Lezione 09](#)

3. Ripasso di Java

Per scrivere un programma stand-alone, chiamato `MyProgram.java` è necessario che la `public class` abbia lo stesso nome del file e che abbia il metodo `main`, dal quale inizia l'esecuzione.

```
import ... ; // import a package or a class
public class MyProgram {
    public static void main(String[] args) {
        /*...*/ // corpo del metodo main
    }
    /* eventuali altri metodi */
}
```

Per compilare il programma si usa il comando `javac MyProgram.java` da terminale, che crea un `bytecode` `MyProgram.class` eseguibile sulla `Java Virtual Machine` (JVM). Per eseguirlo si usa il comando `java`

`MyProgram`, che richiede che la directory corrente sia nel `CLASSPATH`, altrimenti si usa `java -cp - MyProgram`.

In alternativa, si può usare in *Integrated Development Environment* (IDE), come IntelliJ IDEA, Eclipse, Jbuilder.

3.1. Caratteristiche di Java e dell'approccio Object-Oriented

Le caratteristiche principali sono la modularità, l'astrazione e encapsulamento (information hiding) e l'ereditarietà (Inheritance).

3.1.1. Modularità

Le classi vengono viste come un insieme di oggetti che interagiscono insieme; Main è la classe principale, mentre le altre rappresentano tipi di oggetti. Un'applicazione è un insieme di *oggetti interagenti*.

Un *oggetto* è un'*istanza di una classe* che definisce variabili di istanza (in inglese "instance variables" o "fields") e metodi ("methods").

Per definire una variabile `i` di tipo `Integer`, alla quale assegno un oggetto di tipo `Integer`, dopo averlo creato, scrivo: `Integer i = new Integer(5);`.

3.1.2. Astrazione e encapsulamento (information hiding)

Con l'*information hiding* si astrae la *specificità* delle funzionalità, separandola dalla loro *implementazione*, quindi si possono usare le funzionalità solamente in base alla specifica.

Visto che l'implementazione delle funzionalità è nascosta, gli errori sono più confinati (si ha *robustezza*) e c'è la possibilità di cambiare l'implementazione senza cambiare la specifica (si ha *adattabilità*).

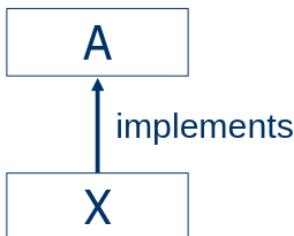
Nel caso delle strutture dati, questo approccio dà origine alla nozione di *Abstract Data Type* (ADT), che in Java si realizza tramite l'utilizzo di interfacce, classi e classi astratte.

Un'*interfaccia* è un insieme di dichiarazioni di metodi senza corpo e di costanti.

Una *classe* è la definizione di costanti, variabili e metodi con corpo.

Una *classe astratta* è una via di mezzo tra un'interfaccia e una classe (ci sono alcuni metodi senza corpo).

```
public interface A {  
    public int a1();  
    public boolean a2();  
}  
  
public class X implements A {  
    public int a1() { /*...*/ }  
    public boolean a2() { /*...*/ }  
    public void b() { /*...*/ }  
}  
  
A var1 = new A(); //NO!  
A var1 = new X(); //OK!
```



Ad esempio: `java.lang.String implements java.lang.Comparable`.

⚠ Attenzione

Non si può creare un oggetto di un'interfaccia, sono legati solo alle classi.

3.1.3. Ereditarietà (inheritance)

Con l'ereditarietà si *importano* in una classe le caratteristiche di

un'altra classe, aggiungendone di nuove e/o specializzandone alcune.

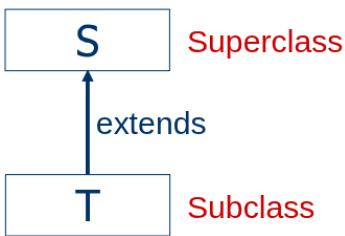
In Java una *sottoclasse estende una superclasse*

```
public class S {  
    public void a() { /*...*/ }  
    public void b() { /*...*/ }  
    public int c() { /*...*/ }  
}  
  
public class T extends S {  
    /* inherits b() and c() */  
    float y; // added
```

```

public void a() { /*...*/ } // specialized
public boolean d() { /*...*/ } // added
}

```



① Osservazione

Ogni classe estende implicitamente `java.lang.Object`.

3.1.3.1. Ereditarietà multipla per interfacce

Un'Interfaccia (*non* una classe) può estender più interfacce.

```

public interface A { /*...*/ }
public interface B { /*...*/ }
public interface C extends A, B { /*...*/ }

```

① Osservazione

`A` e `B` non possono contenere metodi con la stessa firma.

```

public class D implements A, B {
    // deve implementare tutti i metodi di A e B
}

public class D implements C {
    // deve implementare tutti i metodi di A e B
    // e quelli (eventuali) aggiuntivi di C
}

```

3.2. Programmazione generica (generics)

La programmazione generica è un tipo di polimorfismo, introdotto da Java SE5. Permette l'uso di *variabili di tipo* nella definizione di classi, interfacce e metodi. Si parla in questo caso di *classi/interfacce generiche* e *metodi generici*.

Nel creare un'istanza di una classe generica si specifica il tipo (*actual type*) da sostituire con la variabile di tipo. L'*actual type* non può essere un tipo primitivo (come per esempio un intero), ma deve essere una classe che estende `Object`.

Nell'invocazione di un metodo generico, la sostituzione della variabile di tipo con un *actual type* è fatta automaticamente in base ai parametri passati.

L'uso delle classi/interfacce generiche evita il ricorso a parametri di tipo `Object` e l'uso frequente di cast.

```
public interface MyInterface<E> {
    public int size();
    public boolean method1 (E var1);
    public E method2 ();
}

public class MyClass<E> implements MyInterface<E>{
    E var;
    public int size() { /*...*/ };
    public boolean method1 (E var1) { /*...*/ };
    public E method2 () { /*...*/ };
    public E method3 (float var2) { /*...*/ };
}

MyInterface<Integer> x = new MyClass<Integer>();

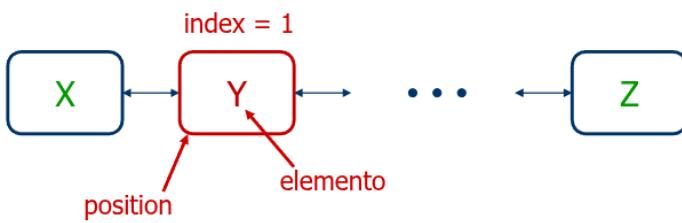
public class MyClass1<E extends S> { /*...*/ } // E può essere sostituito
solo da oggetti di tipo che estende/implementa la classe/interfaccia S
```

3.3. ADT elementari

3.3.1. Lista (list)

Una *lista* (*list*) è una collezione di elementi organizzati secondo un ordine lineare tale per cui è identificabile il primo elemento, il secondo e così via fino all'ultimo.

- In una lista *index-based* un elemento può essere acceduto tramite un *indice intero* che indica il numero di elementi che lo precedono;
- In una lista *position-based* ogni elemento è contenuto in un contenitore detto *nodo* (*position*).



3.3.1.1. Lista index-based

```

public interface List<E> {

    /** Returns the number of elements in this list. */
    public int size();

    /** Returns whether the list is empty. */
    public boolean isEmpty();

    /** Inserts an element e to be at index i, shifting all elements
     * after this. */
    public void add(int i, E e);

    /** Returns the element at index i, without removing it. */
    public E get(int i);

    /** Replaces the element at index i with e, returning the previous
     * element at i. */
    public E set(int i, E e);

    /** Removes and returns the element at index i shifting left
     * subsequent elements. */
    public E remove(int i)
}

```

! Osservazione

I metodi `public int size()` e `public boolean isEmpty()` si troveranno in ogni ADT che vedremo.

Si implementa questo tipo di lista *tramite array*. I metodi possono essere implementati con complessità $O(1)$, tranne `add` e `remove`, che richiedono complessità $O(n)$ (dove n è il numero di elementi nella lista).

! Osservazione

Nel caso in cui l'array sia pieno, l'inserimento di un elemento x è implementato creando un nuovo array di capacità maggiore (solitamente doppia), trasferendo tutte le entry dal vecchio al nuovo array, e inserendo l'elemento x nel nuovo array.

3.3.1.2. Lista position-based

```
public interface Position<E> {  
    /** Return the element stored at this position */  
    public E getElement();  
}  
  
public interface PositionalList<E> {  
    /** Returns the number of elements in this list. */  
    public int size();  
    /** Returns whether the list is empty. */  
    public boolean isEmpty();  
    /** Returns the first node in the list. */  
    public Position<E> first();  
    /** Returns the last node in the list. */  
    public Position<E> last();  
    /** Returns the node after a given node in the list. */  
    public Position<E> after(Position<E> p);  
    /** Returns the node before a given node in the list. */  
    public Position<E> before(Position<E> p);  
    /** Inserts an element at the front of the list. */  
    public void addFirst(E e);  
    /** Inserts an element at the back of the list. */  
    public void addLast(E e);  
    /** Inserts an element after the given node in the list. */  
    public void addAfter(Position<E> p, E e);  
    /** Inserts an element before the given node in the list. */  
    public void addBefore(Position<E> p, E e);  
    /** Removes a node from the list, returning the element stored there. */  
    public E remove(Position<E> p);  
    /** Replaces the element stored at the given node, returning old  
     * element. */  
    public E set(Position<E> p, E e);  
}
```

Si implementa questo tipo di lista tramite *doubly-linked* list. Ogni nodo diventa un oggetto a sé, con variabili di istanza che "puntano" al predecessore e al successore. L'inizio e la fine della lista sono delimitati da dei *nodi sentinella*. I metodi possono essere implementati con complessità $O(1)$, ma la lista può essere scandita solo sequenzialmente,

! Osservazione

È possibile implementare una lista index-based tramite soubled-linked list, o una lista position-based tramite array, ma con scarsi vantaggi.

3.3.2. Pila (stack) e coda (queue)

Una *pila (stack)* è una collezione di elementi inseriti e rimossi in base al principio *Last-In First-Out* (LIFO).

```
public interface Stack<E> {  
    /** Returns the number of elements in this list. */  
    public int size();  
    /** Returns whether the list is empty. */  
    public boolean isEmpty();  
    /** Inserts an element e at the top of the stack. */  
    public void push(E e);  
    /** Returns the element at the top of the stack without removing it.  
     */  
    public E top();  
    /** Removes and returns the element at the top of the stack. */  
    public E pop();  
}
```

Una *coda (queue)* è una collezione di elementi inseriti e rimossi in base al principio *First-In First-Out* (FIFO).

```
public interface Queue<E> {  
    /** Returns the number of elements in this list. */  
    public int size();  
    /** Returns whether the list is empty. */  
    public boolean isEmpty();  
    /** Inserts an element e at the rear of the queue. */  
    public void enqueue(E e);  
    /** Returns but does not remove the first element of the queue (null  
     * if empty). */  
    public E first();  
    /** Removes and returns the first element of the queue (null if  
     * empty). */  
    public E dequeue();  
}
```

Entrambe le strutture dati possono essere implementate tramite *doubly-linked list*. La pila effettua inserimento (*push*) e rimozione (*pop*) in testa alla lista, mentre la coda effettua l'inserimento (*enqueue*) in coda e la rimozione (*dequeue*) in testa alla lista. Tutti i metodi possono essere implementati con complessità $O(1)$ ma, a differenza della lista, non si ha accesso a element intermedi a meno di no rimuovere quelli che li precedono nell'ordine di accesso.

⚠️ Osservazione

È possibile usare anche una *singly-linked list*.

3.4. Collection framework di Java

"*Collection*" è un termine generico per indicare una struttura dati.

Il *collection framework di Java* è un'architettura unificata di strutture dati e algoritmi implementato nel package `java.util`.

Esso contiene:

- *interfacce* di varie collection;
- *classi* che implementano le interfacce;
- *algoritmi polimorfi* per operare su collection (*metodi statici della classe Collections*), come ad esempio il sorting;
- ampio uso della *programmazione generica*.

Il package `java.util` contiene diverse implementazioni di Liste, Pile e Code. Tra esse si segnalano le seguenti:

- Lista
 - Interfaccia: `List`;
 - Classe `ArrayList`: implementazione index-based;
 - Classe `LinkedList`: implementazione sia index-based che position-based. Tuttavia l'implementazione position-based non utilizza esplicitamente il concetto di position ma si basa sull'uso di iteratori.
- Pila e Coda
 - Interfaccia unificata: `Deque` (Double-ended queue);
 - Classe `LinkedList`: implementazione unificata di Pila e Coda (e Lista);

- Classe Stack: implementazione della Pila basata su Vector. Si consiglia tuttavia l'uso di classi, come `LinkedList`, che implementano l'interfaccia `Deque` che è più completa.

ⓘ Info

I nomi di alcuni metodi differiscono da quelli riportati precedentemente che seguono il libro di testo.

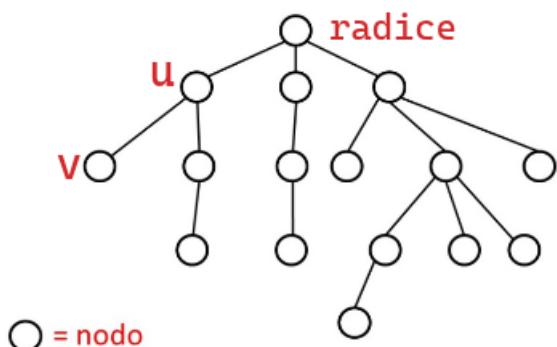
ⓘ Riepilogo sul ripasso di Java

- Programma stand-alone;
- Caratteristiche di Java e dell'approccio Object-Oriented;
- ADT elementari:
 - Lista (index-based e position-based);
 - Pila
 - Coda
- Collection framework di Java, con particolare attenzione alle interfacce e classi che implementano Lista, Pila e Coda.

4. Alberi

4.1. Alberi generali

Un *albero* è una collezione di *nodi* caratterizzata da una *struttura gerarchica* che si dipana da un nodo *radice* tramite relazioni di tipo padre-figlio.



Nel disegno *u* è padre di *v*, *v* è figlio di *u*.

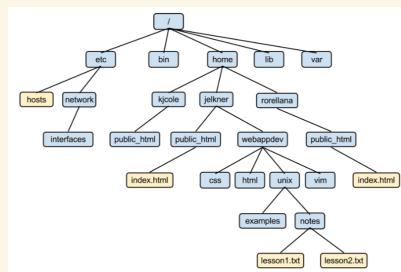
ⓘ Osservazioni

Le relazioni padre-figlio costituiscono un insieme di collegamenti minimi che introducono un legame (connessione) tra tutti i nodi.

Una lista è un caso estremo di albero con una struttura gerarchica lineare.

💡 Campi applicativi

1. Strutture dati: mappe, priority queue;
2. Esplorazione risorse: filesystem, siti di e-commerce;



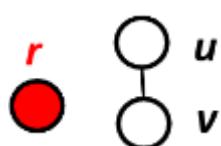
3. Sistemi distribuiti e reti di comunicazione:
sincronizzazione, broadcast, gathering;
4. Analisi di algoritmi: albero della ricorsione;
5. Classificazione: alberi di decisione;
6. Compressione di dati (codici di Huffman);
7. Biologia computazionale: alberi filogenetici.

4.1.1. Definizioni e proposizioni

Un *albero radicato* (*rooted tree*) T è una collezione di nodi che, se non è vuota, soddisfa le seguenti proprietà:

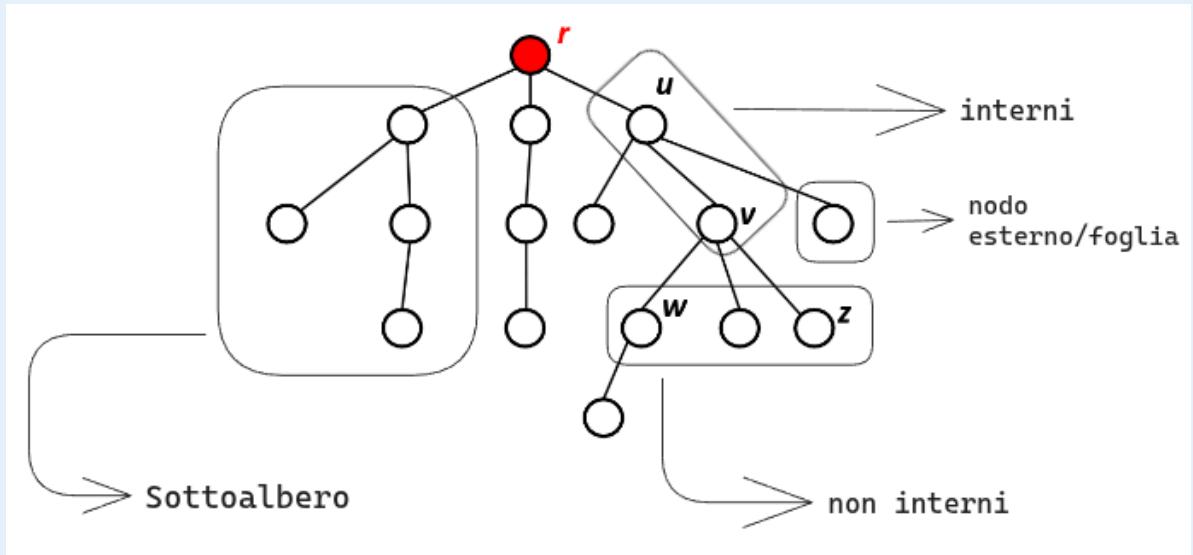
- \exists un nodo speciale $r \in T$ (r è chiamato *radice*);
- $\forall v \in T, v \neq r: \exists! u \in T: u$ è padre di v (v è figlio di u);
- $\forall v \in T, v \neq r: \text{risalendo di padre in padre si arriva a } r$ (ovvero ogni nodo è discendente dalla radice).

Nel libro di testo la terza condizione manca. Senza questa, la seguente collezione con u padre di v e v padre di u sarebbe un albero, che ha poco senso. Questa collezione piuttosto è una foresta di alberi.



Terminologia

- x è *antenato* di y se $x = y$ oppure x è antenato del padre di y ;
- x è *descendente* di y se y è antenato di x ;
- I *nodi interni* sono quei nodi con almeno 1 figlio;
- I *nodi esterni* (o *foglie*) sono i nodi senza figli;
- T_v è un albero formato da tutti i discendenti di v (quindi include v);
- T è un *albero ordinato* se per ogni nodo interno $v \in T$ è definito un ordinamento lineare tra i figli u_1, u_2, \dots, u_k di v .



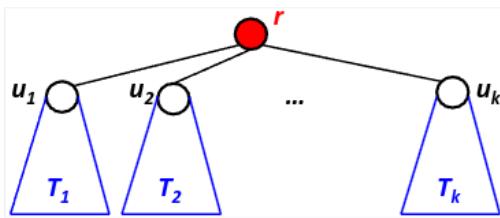
Si può definire ricorsivamente cos'è un albero radicato.

Un *albero radicato* T è una collezione di nodi che, se non è vuota, risulta partizionata in questo modo:

$$T = \{r\} \cup T_1 \cup T_2 \cup \dots \cup T_k$$

per un qualche $k \geq 0$, dove:

- r è radice con figli u_1, u_2, \dots, u_k ;
- $\forall i, 1 \leq i \leq k: T_i$ è un albero non vuoto con radice u_i ($\implies T_i \equiv T_{u_i}$).



Chiaramente, le due definizioni di albero radicato (ricorsiva e non) sono equivalenti.

Si definisce la *profondità* di un nodo v in un albero T in due modi alternativi:

1. $\text{depth}_T(v) = |\text{antenati}(v)| - 1;$
2. • Se $v = r$ radice $\Rightarrow \text{depth}_T(v) = 0;$
• Altrimenti $\text{depth}_T(v) = 1 + \text{depth}_T(\text{padre}(v)).$

Il *livello* è l'insieme dei nodi a profondità i (*froalli* ≥ 0).

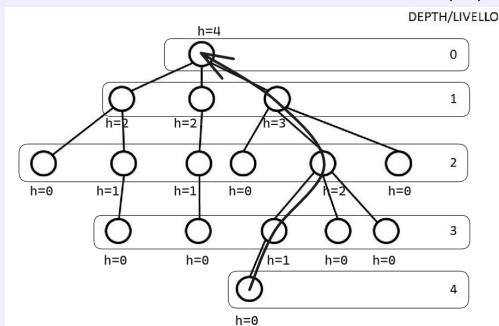
L'*altezza* di un nodo v in un albero T ($\text{height}_T(v)$) si definisce come:

- Se v è foglia $\Rightarrow \text{height}_T(v) = 0;$
- Altrimenti $\text{height}_T(v) = 1 + \max_{w:w \text{ figlio di } v} (\text{height}_T(w)).$

L'*altezza di un albero* T si definisce $\text{height}(T) = \text{height}_T(r)$, con r radice di T .

Proposizione

Dato un albero T , $\text{height}(T) = \max_{v \in T: v \text{ foglia}} (\text{depth}_T(v)).$

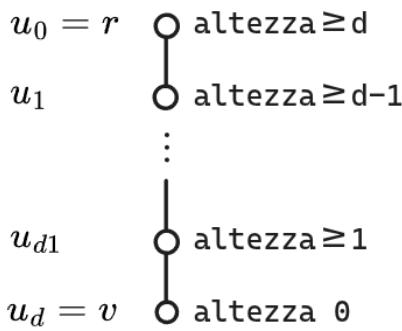


4.1.1.1. Dimostrazione

Sia h l'altezza dell'albero e $d = \max_{v \in T: v \text{ foglia}} (\text{depth}_T(v))$. Proviamo $h \geq d$ e $h \leq d$.

4.1.1.1.1. $h \geq d$

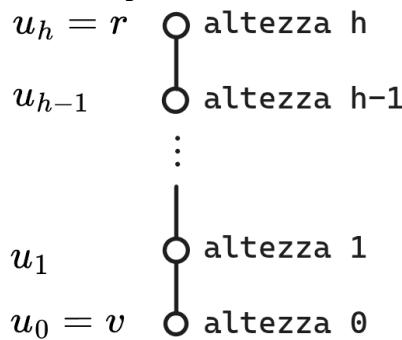
Sia v una foglia a profondità d e sia r la radice di T . allora esiste un percorso da v a r .



È immediato vedere che
 $\text{height}_T(u_i) \geq d - i \forall d \geq i \geq 0 \implies \text{height}_T(r) \geq d \implies h \geq d.$

4.1.1.1.2. $h \leq d$

Per assurdo, se $h \geq d$ deve esistere un percorso in T dalla radice r a una foglia v .



In questo caso, la foglia v ha h antenati diversi da v , che sono u_1, u_2, \dots, u_h , e quindi $\text{depth}_T(v) = h > d$, che contraddice il fatto che d è la massima profondità di una foglia.

□

4.1.2. Interfacce

4.1.2.1. Iterator e Iterable

Prima di definire l'interfaccia `Tree` definiamo due interfacce.

`Iterator` è un "cursore" che permette di enumerare (scan) gli elementi di una collezione.

```

public interface Iterator<E> {
    /** Returns true if the scan of the collection is not over */
    boolean hasNext();
    /** Returns the next element in the collection */
    E next();
}
    
```

`Iterable` è una collezione che rende disponibile un iteratore ai suoi elementi.

```
public interface Iterable<E> {  
    /** Returns an iterator of the collection */  
    Iterator<E> iterator()  
}
```

4.1.2.1. Tree

L'interfaccia `Tree` rappresenta l'implementazione tipica di un albero.

Notare che il puntatore alla radice è l'unico punto di accesso e che ogni nodo è un oggetto a sé stante (ad esempio di una classe che implementa `Position`) e offre metodi per accedere al padre e ai figli.

```
public interface Tree<E> extends Iterable<E> {  
    /** Returns the number of positions in the tree */  
    int size();  
    /** Returns true if the tree contains no positions */  
    boolean isEmpty();  
    /** Returns the Position of the root (or null if empty) */  
    Position<E> root();  
    /** Returns the Position of p's parent (or null if p is the root) */  
    Position<E> parent(Position<E> p);  
    /** Returns an iterable containing p's children */  
    Iterable<Position<E>> children(Position<E> p);  
    /** Returns the number of children of p */  
    int numChildren(Position<E> p);  
    /** Returns true if p is internal */  
    boolean isInternal(Position<E> p);  
    /** Returns true if p is external */  
    boolean isExternal(Position<E> p);  
    /** Returns true if p is root */  
    boolean isRoot(Position<E> p);  
    /** Returns an iterator to all element in the tree */  
    Iterator<E> iterator();  
    /** Returns an iterable containing all positions in the tree */  
    Iterable<Position<E>> positions();  
}
```

! Osservazioni

`iterator()` deriva dal fatto che `Tree<E>` estende `Iterable<E>` (dove `E` rappresenta il tipo dei dati contenuti nei nodi). Assumiamo complessità $\Theta(1)$ per tutti i metodi, tranne `children`, `iterator` e `positions`, e che sia possibile enumerare i figli di un nodo (tramite `children`) in tempo proporzionale al loro numero (quindi ogni figlio è enumerato in tempo costante).

4.1.3. Calcolo della profondità di un nodo (algoritmo ricorsivo)

Algoritmo `depth(v)`

Input: $v \in T$.

Output: profondità di v in T .

```
if(T.isRoot(v)) then{
    return 0; //Caso base
}
else{
    return 1 + depth(T.parent(v));
}
```

! Osservazione

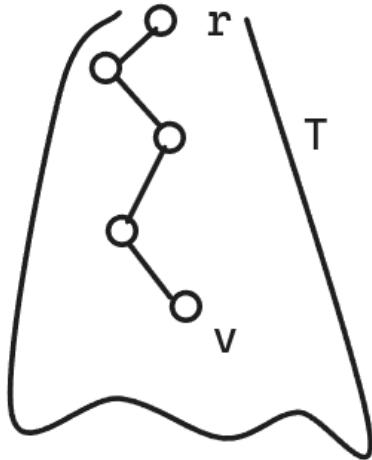
Come nel libro di testo, definiamo gli algoritmi di base per gli alberi come metodi di una classe astratta che implementa l'interfaccia `Tree`, non specificando l'albero T come parametro in quanto esso è associato implicitamente all'istanza (`this`) da cui si invoca il metodo.

4.1.3.1. Complessità di `depth`

Studiamo l'albero della ricorsione associato a `depth(v)`, $v \in T$. Se d_v è la profondità di v :

- Vi sono $d_v + 1$ invocazioni ricorsive di `depth`, una per ogni antenato di v ;
- Avvengono $\Theta(1)$ operazioni in ciascuna invocazione ricorsiva.
 \Rightarrow Vi sono $d_v + 1$ nodi dell'albero della ricorsione di costo $\Theta(1)$ ciascuno.

\implies Complessità di $\text{depth}(v) \in \Theta(d_v + 1)$.



① Osservazione

Il "+1" è giustificato dal fatto che se $d_v = 0$ (ovvero v è la radice), comunque $\text{depth}(v)$ richiede $\Theta(1)$ operazioni; tuttavia per semplificare la notazione scriveremo $\Theta(d_v)$ intendendo $\Theta(d_v + 1)$.

4.1.3.2. Diversa taglia dell'istanza

Se volessi esprimere la complessità in funzione di $n = |T|$?

Usando la profondità d come taglia dell'istanza, le possibili istanze di taglia sono tutti i nodi di tutti i possibili alberi T (di qualsiasi cardinalità $|T|$) \implies La complessità al caso pessimo è $\Theta(d)$.

Usando il numero di nodi n come taglia dell'istanza, le possibili istanze di taglia n sono tutti i nodi appartenenti ad alberi T , con $|T| = n \implies$ La complessità al caso pessimo è $\Theta(n)$.

4.1.3.2.1. Dimostrazione di complessità

L'upper bound è $O(n)$ perché in un albero con n nodi ogni nodo v ha profondità $\text{len} \implies \text{depth}(v)$ richiede $O(n)$ operazioni.

Il lower bound è $\Omega(n)$ perché in un albero che è una catena di n nodi, l'ultimo nodo della catena è una foglia v di profondità $n - 1$, e quindi $\text{depth}(v)$ richiede $\Omega(n)$ operazioni.

4.1.3.3. Versione iterativa

Algoritmo $\text{depthITER}(v)$

Input: $v \in T$.

Output: profondità di v in T .

```

d <- 0;
u <- v;
while(!T.isRoot(u)) do{
    u <- parent(u);
    d <- d+1;
}
return d;

```

La sua complessità è $\Theta(d_v)$.

4.1.4. Calcolo dell'altezza di un nodo

Algoritmo height(v)

Input: $v \in T$.

Output: altezza di v in T .

```

h <- 0;
foreach w ∈ T.children(v) do{
    h <- max{h, 1 + height(w)};
}
return h;

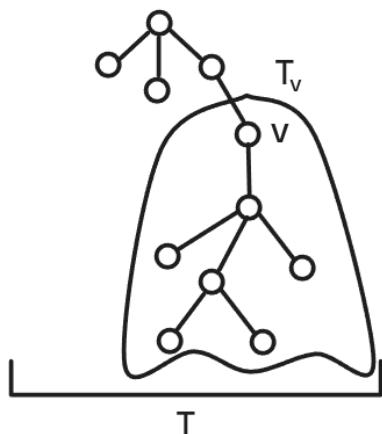
```

Tecnicamente, `T.children(v)` è una collezione "Iterable" che contiene i figli di v . Con "`foreach w ∈ T.children(v)`" si enumerano i figli di v , ciascuno dei quali viene generato in tempo $\Theta(1)$.

4.1.4.1. Complessità di height

Studiamo l'albero della ricorsione associato a `height(v)` con $v \in T$:

- Ha un nodo (cioè un'invocazione ricorsiva) per ogni $u \in T_v$;
 - Il costo associato al nodo $u \in T_v$ è $\Theta(c_u + 1)$, dove c_u è il numero di figli di u
- ⇒ La complessità di `height(v)` $\in \Theta(\sum_{u \in T_v} (c_u + 1))$



Sia n_v il numero di nodi in T_v (quindi il numero di discendenti di v). Riscriviamo $\sum_{u \in T_v} (c_u + 1)$ in funzione di n_v :

$$\sum_{u \in T_v} (c_u + 1) = (\sum_{u \in T_v} c_u) + \sum_{u \in T_v} 1 = (\sum_{u \in T_v} c_u) + n_v.$$

Proposizione (8.4 del libro di testo)

$$\sum_{u \in T_v} c_u = n_v - 1$$

$$\implies \sum_{u \in T_v} (c_u + 1) = 2n_v - 1$$

\implies La complessità di `height(v)` è $\Theta(n_v)$.

4.1.4.1.1. Dimostrazione della proposizione

In generale la relazione è vera perché ogni nodo di T_v , tranne v , è calcolato *esattamente* una volta in $\sum_{u \in T_v} c_u$ come figlio di suo padre.

□

Dall'analisi fatta discende che la complessità per calcolare l'altezza di tutto l'albero T (invocando `height(r)` con r radice di T) è $\Theta(n)$, con $n = |T|$, quindi lineare nel numero di nodi dell'albero.

Lezione 11

4.1.5. Visite di alberi

La *visita di un albero* T è la scansione sistematica di tutti i nodi di T che permette di eseguire una qualche operazione (*visita*) ad ogni nodo.

Rappresentano un *design pattern algoritmico* che può essere istanziato per il calcolo di valori e/o per impostazione di opportune variabili associate ai nodi.

Per gli alberi generali studieremo la visita in *preorder* e in *postorder*.

4.1.5.1. Preorder

Una visita in *preorder* visita *prima il padre e poi (ricorsivamente) i sottoalberi radicati nei figli*.

Con questa modalità, le operazioni svolte per un nodo possono dipendere da quelle svolte per i suoi antenati.

Algoritmo `preorder(v)`

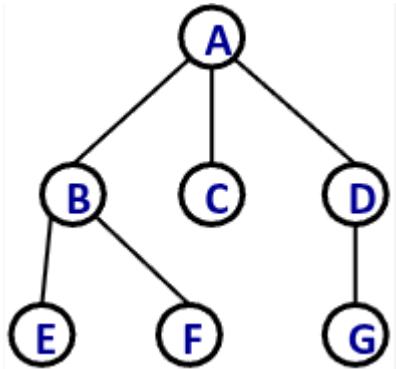
Input: nodo $v \in T$.

Output: risultante dalla visita di T_v .

```
visita v;
foreach w ∈ T.children(v) do{
    preorder(w);
}
```

La *chiamata iniziale* da effettuare per visitare tutto T è `preorder(T.root())`.

Nel *caso base* v è una foglia (il ciclo `foreach` non esegue istruzioni, infatti v non ha figli).



Assumendo che `children()` esamini i figli da sinistra a destra, l'ordine della visita in preorder per l'albero sovrastante è A, B, E, F, C, D, G .

4.1.5.2. Postorder

Una visita in *postorder* visita prima (ricorsivamente) i sottoalberi radicati nei figli, poi il padre.

Con questa modalità, le operazioni svolte per un nodo possono dipendere da quelle svolte per i suoi discendenti.

Algoritmo postorder(v)

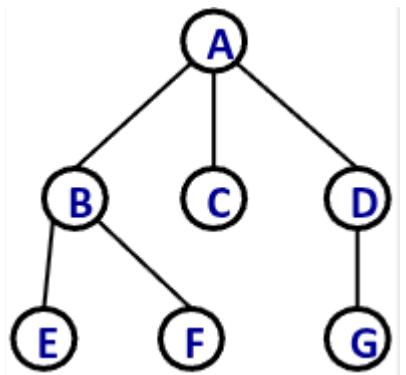
Input: nodo $v \in T$.

Output: risultante dalla visita di T_v .

```
foreach w ∈ T.children(v) do{
    postorder(w);
}
visita v;
```

La *chiamata iniziale* da effettuare per visitare tutto T è `postorder(T.root())`.

Nel *caso base* v è una foglia (il ciclo `foreach` non esegue istruzioni, infatti v non ha figli).



Assumendo che `children()` esamini i figli da sinistra a destra, l'ordine della visita in preorder per l'albero sovrastante è E, F, B, C, G, D, A .

Sia T un albero ordinato e siano $u, v \in T$ due nodi allo stesso livello.

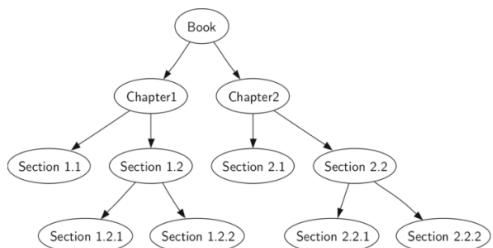
Diciamo che u è *a sinistra* di v (e quindi v è a destra di u) se u viene prima di v nella visita in *preorder*.

! Osservazione

La definizione è coerente con il modo di disegnare gli alberi.

4.1.5.3. Esempi

Quale visita è opportuno utilizzare per stampare l'indice di un libro, rappresentato tramite il seguente albero?



Quella corretta è la visita in preorder, infatti la sequenza di visita è "Book: Chapter1, Section 1.1., Section 1.2, Section 1.2.1, Section 1.2.2, Chapter 2, Section 2.1, Section 2.2, Section 2.2.1, Section 2.2.2".

Un esempio analogo è rappresentato dalla struttura di un file system come sequenza di cartelle e file.

4.1.5.3.1. Esempio di algoritmo basato sulla visita in preorder (allDepths)

Vogliamo progettare un algoritmo `allDepths` che, dato un albero T , calcoli la profondità di ogni nodo $v \in T$ e la memorizzi in un campo `v.depth`.

Proviamo ad adattare la visita in preorder, definendo la visita di un nodo v in questo modo: se v è radice si imposta la profondità a zero, altrimenti si imposta la profondità a $1 +$ la profondità del padre, che è già impostata, dato che il padre è già stato visitato.

Algoritmo `allDepths(T, v)`

Input: $v \in T$, e `u.depth` impostato correttamente per u padre di v .

Output: `z.depth` impostato correttamente $\forall z \in T_v$.

```
//visita
if(T.isRoot(v)) then{
    v.depth <- 0;
}
else{
    v.depth(v) <- 1 + T.parent(v).depth;
}

//visita ricorsiva dei sottoalberi radicati nei figli
forall w ∈ T.children(v) do{
    allDepths(w);
}
```

(!) Osservazioni

Per impostare il campo `depth` per tutti i nodi di T invoco `allDepths(T, T.root())`.

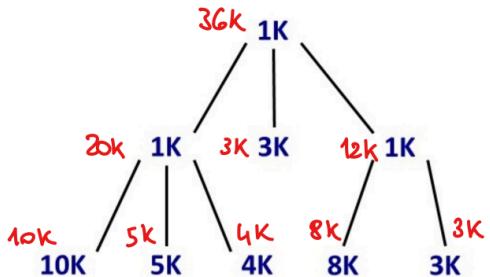
`allDepths` non restituisce alcun valore (non c'è un `return`), ma modifica i modi dell'albero che si considerano come oggetti globali che sopravvivono all'esecuzione dell'algoritmo.

4.1.5.3.2. Esempi di algoritmi basati sulla visita in postoder

L'algoritmo `height` è un esempio di visita in postorder dato che l'altezza di un nodo viene calcolata solo dopo aver calcolato quelle dei figli.

Si consideri un file system gerarchico in cui la struttura è rappresentata da un albero T dove i nodi interni corrispondono alle cartelle e i nodi foglia ai file. Ogni nodo v ha un campo `v.loc-size` che memorizza lo spazio occupato dal nodo, escludendo quello dei discendenti.

Progettare un algoritmo `diskSpace` che dato un tale T calcoli, per ogni nodo $v \in T$, lo spazio aggregato occupato dai suoi discendenti e lo memorizzi in un campo `v.aggr-size`.



Nell'esempio sovrastante sono rappresentati in blu i valori dei campi `loc-size`, mentre in rosso quelli dei campi `aggr-size` alla fine dell'algoritmo.

Algoritmo `diskSpace(T, v)`

Input: $v \in T$ e `u.loc-size` impostato $\forall u \in T$.

Output: `v.aggr-size`, `z.aggr-size` impostato correttamente $\forall z \in T_v$.

```

v.aggr-size <- v.loc-size;
for each w ∈ T.children(v) do{
    v.aggr-size <= v.aggr-size + diskSpace(T, w);
}
return v.aggr-size;
  
```

Alternativamente si può scegliere di non far restituire `v.aggr-size` (con input e output analoghi):

```

v.aggr-size <- v.loc-size;
for each w ∈ T.children(v) do{
    diskSize(T, w);
    v.aggr-size <= v.aggr-size + w.aggr-size;
}
  
```

In questa versione la chiamata ricorsiva è isolata, infatti non restituisce nessun valore.

⌚ Osservazioni per la scrittura di algoritmi ricorsivi

- Un algoritmo ricorsivo può utilizzare sia *variabili globali* che sopravvivono a tutta l'esecuzione dell'algoritmo, sia *variabili locali alle singole invocazioni* che rimangono in vita solo durante l'esecuzione dell'invocazione corrispondente.
- Gli eventuali valori restituiti dalle invocazioni ricorsive (se ce ne sono) devono essere "utilizzati" in qualche modo, altrimenti vanno perduti.

4.1.5.5. Complessità

4.1.5.5.1. Complessità di `preorder(T.root())`

Sia n il numero di nodi di T . Consideriamo l'albero della ricorsione associato all'esecuzione di `preorder(T.root())` :

- Ha n nodi associati alle invocazioni ricorsive che sono *esattamente* una per ogni nodo di T ;
- Il costo del nodo associato all'invocazione di `preorder(v)`, con $v \in T$ generico, è $\Theta(t_v + c_v + 1)$, dove t_v è il costo di "visita v " e c_v è il numero di figli di v .
 \Rightarrow La complessità di `preorder(T.root())` è
 $\Theta\left(\sum_{v \in T} (t_v + c_v + 1)\right) = \Theta\left((\sum_{v \in T} t_v) + (\sum_{v \in T} (c_v + 1))\right) = \Theta\left(n + \sum_{v \in T} t_v\right).$

! Osservazione

Si ha la stessa complessità per la visita in `postorder` e `inorder` (per gli alberi binari).

Dall'analisi discende che le visite consentono di enumerare tutti i nodi di T in tempo lineare in $|T|$ e, se $t_v \in \Theta(1)$, la complessità totale sarebbe $\Theta(n = |T|)$.

4.1.5.5.2. Complessità di `allDepths(T, T.root())`

`allDepths(T, T.root())` ha lo stesso schema della visita in `preorder` e $t_v \in \Theta(1) \forall v \in T$, quindi la complessità totale è $\Theta(n)$, dove $n = |T|$.

4.1.5.5.3. Complessità di `diskSum(T, T.root())`

`diskSum(T, T.root())` ha lo stesso schema della visita in `postorder` e $t_v \in \Theta(c_v + 1)$, con c_v il numero di figli di v . Allora la complessità è $T\left(n + \sum_{v \in T} t_v\right) = \Theta\left(n + \sum_{v \in T} (c_v + 1)\right) = \Theta(n)$.

Riepilogo sugli alberi generali

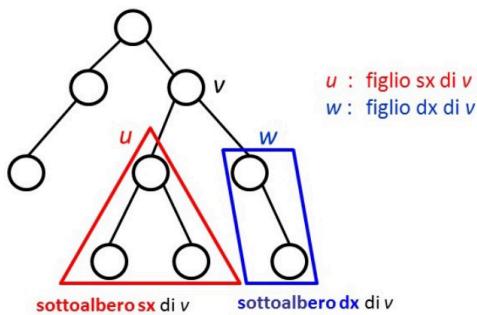
- Definizione: albero, antenati, discendenti, sottoalbero, profondità di un nodo, altezza di un nodo, profondità di un nodo e di un albero;
- Relazione tra altezza di un albero e profondità delle foglie;
- Algoritmi per il calcolo della profondità e altezza di un nodo e dell'altezza di un albero;
- Visite: preorder, postorder e le loro applicazioni;
- Algoritmi di visita come template generali.

4.2. Alberi binari

Per *arietà* di un albero si intende il *massimo numero di figli di un nodo interno*.

Un *albero binario* T è un albero ordinato in cui:

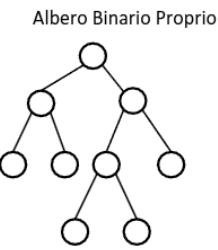
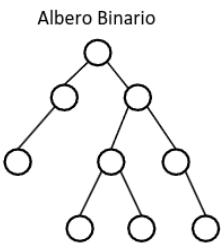
- Ogni *nodo interno* ha ≤ 2 *figli*;
- Ogni nodo non radice è etichettato come *figlio sinistro* (sx) o *destro* (dx) di suo padre;
- Se ci sono entrambi i figli, il figlio sinistro viene prima del figlio destro nell'ordinamento dei figli di un nodo.



Un *albero binario proprio* T è un albero binario tale che *ogni nodo interno ha esattamente 2 figli*.

Terminologia

In letteratura gli alberi *propri* ("proper" in inglese) sono anche chiamati *pieni* ("full" in inglese).



4.2.1. Interfaccia BinaryTree

```

public interface BinaryTree<E> extends Tree<E> {
    /**
     * Returns the Position of p's left child (or null if it doesn't
     * exists) */
    Position<E> left(Position<E> p);
    /**
     * Returns the Position of p's right child (or null if it doesn't
     * exists) */
    Position<E> right(Position<E> p);
    /**
     * Returns the Position of p's sibling (or null if no sibling
     * exists) */
    Position<E> sibling(Position<E> p);
}
  
```

4.2.2. Proprietà

Sia T un albero binario proprio non vuoto, dove $n = |T|$ è il numero di nodi in T , m ($\circ n_E$) è il numero di foglie in T , $n - m$ ($\circ n_I$) è il numero di nodi interni in T e h è l'altezza di T .

1. $m = n - m + 1$, ovvero le foglie sono uguali al numero di nodi interni più uno; ci permette di stabilire quanto spazio - al massimo - verrà occupato;
2. $h + 1 \leq m \leq 2^h$;
3. $h \leq n - m \leq 2^h - 1$;
4. $2h + 1 \leq n \leq 2^{h+1} - 1$;
5. $\log_2(n + 1) - 1 \leq h \leq \frac{n-1}{2}$.

[Lezione 12](#)

4.2.2.1. Dimostrazioni

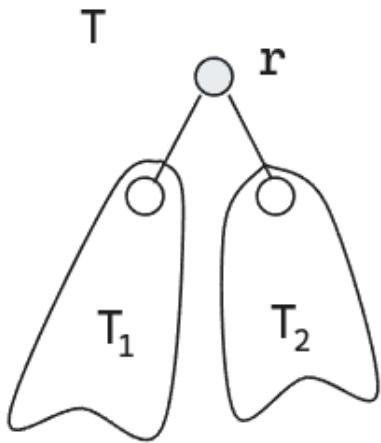
4.2.2.1.1. (1)

Proviamo la prima proprietà per induzione sull'altezza $h \geq 0$ di T . Caso base: $h = 0 \implies T$ equivale a un singolo nodo $\implies n = 1$ e $m = 1 \implies \checkmark$.

Passo induttivo: Fisso $h \geq 0$ arbitrario.

Ipotesi induttiva: la proprietà (1) vale per tutti gli alberi binari propri di altezza $\leq h$.

Sia T un albero binario proprio di altezza $h+1 \geq 1$, sia T_1 di altezza h_1 il sottoalbero con radice il figlio sinistro della radice di T , mentre T_2 di altezza h_2 quello del figlio destro.



Allora $h+1 = 1 + \max\{h_1, h_2\} \implies h_1, h_2 \leq h \implies$ Per T_1 e T_2 vale l'ipotesi induttiva.

Sia m il numero di foglie di T , n il numero di nodi di T , m_i il numero di foglie di T_i , n_i il numero di nodi di T_i , $i = \{1, 2\}$.

$$m = m_1 + m_2, \quad n = n_1 + n_2 + 1.$$

$$\begin{aligned} m &= m_1 + m_2 = (n_1 - m_1 + 1) + (n_2 + m_2 + 1) = \text{per ipotesi induttiva:} \\ &= (n_1 + n_2 + 1) - (m_1 + m_2) + 1 = n - m + 1. \end{aligned}$$

□

! Osservazione

In un albero binario proprio T , dato che il numero di foglie è uguale al numero di nodi interni aumentato di uno, il numero totale di nodi è dispari.

4.2.2.1.2. (2)

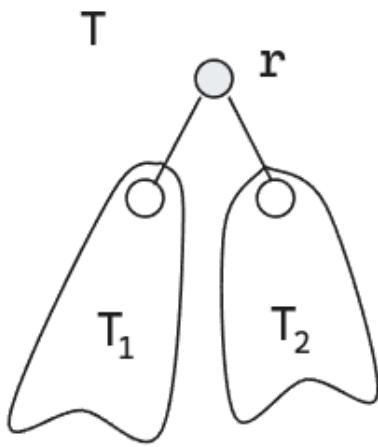
Dimostriamo che $m \leq 2^h$ per induzione su $h \geq 0$.

Caso base: $h = 0 \implies T$ equivale a un singolo nodo $\implies m = 1 \implies \checkmark$.

Passo induttivo: Fisso $h \geq 0$ arbitrario.

Ipotesi induttiva: la proprietà vale per tutti gli alberi binari propri di altezza $\leq h$.

Sia T un albero binario proprio di altezza $h+1 \geq 1$, sia T_1 di altezza h_1 il sottoalbero con radice il figlio sinistro della radice di T , mentre T_2 di altezza h_2 quello del figlio destro.



Allora $h+1 = 1 + \max\{h_1, h_2\} \implies h_1, h_2 \leq h \implies$ Per T_1 e T_2 vale l'ipotesi induttiva.

Sia m il numero di foglie di T , m_i il numero di foglie di T_i , $i = \{1, 2\}$.

$$m = m_1 + m_2 \text{ e devo dimostrare che } m \leq 2^{h+1}.$$

$$m = m_1 + m_2 \leq 2^{h_1} + 2^{h_2} \leq \text{per ipotesi induttiva:}$$

$$\leq 2^h + 2^h = 2^{h+1}.$$

□

Dimostro che $m \geq h+1$ per induzione su $h \geq 0$.

Caso base: $h = 0 \implies T$ equivale a un singolo nodo $\implies m = 1 \implies \checkmark$.

Passo induttivo: Fisso $h \geq 0$ arbitrario.

Ipotesi induttiva: la proprietà vale per tutti gli alberi binari propri di altezza $\leq h$.

In maniera analoga a prima, la proprietà vale per T_1 e T_2 .

$$m = m_1 + m_2 \geq (h_1 + 1) + (h_2 + 1) \geq \text{per ipotesi induttiva:}$$

$$\geq \max\{h_1, h_2\} + 2 = (h + 1) + 1.$$

□

4.2.2.1.3. (3), (4), (5)

Per provare (3): $h \leq n - m \leq 2^h - 1$, sostituisco m in (2) con $n - m + 1$ (da (1)) e ottengo $h + 1 \leq n - m + 1 \leq 2^h \implies h \leq n - m \leq 2^h - 1$.

□

Per provare (4): $2h + 1 \leq n \leq 2^{h+1} - 1$ sommo (2) e (3).

□

(5) deriva da (4), risolvendo rispetto a h :

$$2h + 1 \leq n \implies h \leq \frac{n-1}{2}$$

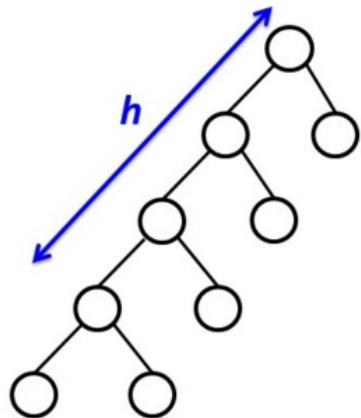
$$2^{h+1} - 1 \geq n \implies 2^{h+1} \geq n + 1 \implies h + 1 \geq \log_2(n + 1) \implies h \geq \log_2(n + 1) - 1.$$

□

Osservazione

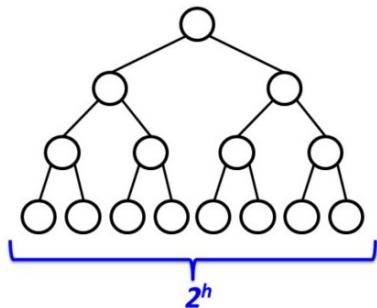
La quinta proprietà, (5), implica che in un albero binario proprio con n nodi, l'altezza è compresa tra $\Omega(\log n)$ e $O(n)$.

4.2.3. Alberi binari propri estremi



In questo esempio si ha $m = h + 1$ e $n = 2h + 1$. È importante notare come le parti sinistre di (2), (3) e (4) e la parte destra di (5) continuano a valere.

È un albero *molto sbilanciato (skewed)*.



In questo esempio ci sono 2^i nodi al livello i , $0 \leq i \leq h$, quindi ci saranno $m = 2^h$ (equivalente al numero di nodi al livello h), quindi $n = \sum_{i=0}^h 2^i = \frac{2^{h+1}-1}{2-1} = 2^{h+1} - 1$. È importante notare come le parti destre di (2), (3) e (4) e la parte sinistra di (5) continuano a valere.

È un albero *perfettamente bilanciato*.

⚠ Attenzione

In assenza di altre ipotesi, il migliore upper bound all'altezza di un albero binario con n nodi è $O(n)$, non $O(\log n)$.

4.2.4. Visite di alberi binari

Oltre alle visite in preorder e postorder, per gli alberi binari si definisce anche la *visita inorder*, che visita *prima*

(ricorsivamente) il sottoalbero sinistro, poi il padre, poi (ricorsivamente) il sottoalbero destro.

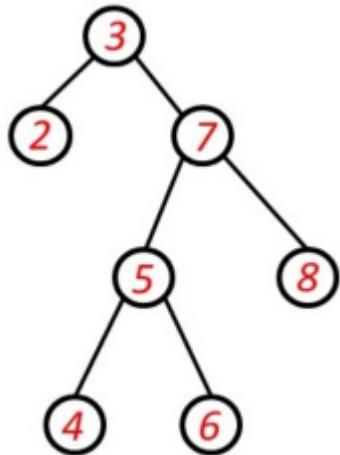
Algoritmo `inorder(v)`

Input: $v \in T$.

Output: visita inorder di T_v

```
if(T.left(v) != null) then{
    inorder(T.left(v));
}
visita v;
if(T.right(v) != null) then{
    inorder(T.right(v));
}
```

La chiamata iniziale per visitare tutto T è `inorder(T.root())`.



! Osservazione

È un esempio di albero binario di ricerca, che studieremo più avanti, dove la visita `inorder` tocca gli elementi presente in ordine crescente di valore.

4.2.4.1. Complessità di `inorder(T.root())`

La complessità è $\Theta(n + \sum_{v \in T} t_v)$, dove $n = |T|$ e t_v è il costo della visita di v .

Si fa la stessa analisi di `preorder` per dimostrare la complessità.

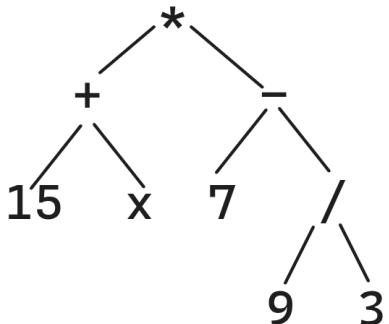
4.2.4.2. Applicazioni: espressioni aritmetiche

Il *Parse Tree* T associato a un'espressione aritmetica E (con operatori solo binari) è un albero binario proprio i cui nodi

foglia contengono le costanti/variabili di E e i nodi interni contengono gli operatori di E , in modo tale che:

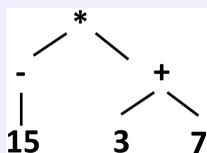
- Se $E = a$, con a una costante/variabile, allora T è costituito da un'unica foglia contenente a ;
- Se $E = (E_1 \text{ Op } E_2)$, la radice di T contiene Op e ha come sottoalbero sinistro il Parse Tree associato a E_1 , mentre come sottoalbero destro il Parse Tree associato a E_2 .

Ad esempio, posso scrivere un'espressione con la *notazione infissa*: $E = ((15 + x) * (7 - (9 \div 3)))$; altrimenti posso usare la *notazione postfissa* o *polacca inversa*: $E = 15x + 793 \div -*$. Il Parse Tree per quest'espressione E è



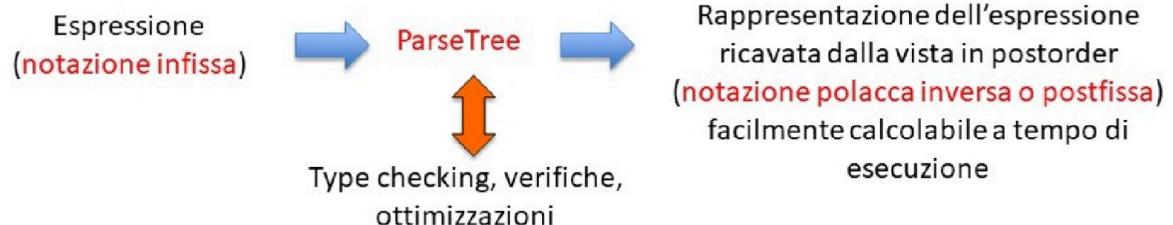
① Osservazioni

- Le parentesi sono implicite nella struttura dell'albero;
- Se si ammettono operatori unari, l'albero non è più proprio,



ad esempio: $-15 * (3 + 7)$

- Nei compilatori avviene il seguente processo:



Nelle sezioni successive vedremo come a partire dal Parse Tree T di un'espressione E si possano generare le rappresentazioni di E in notazione postfissa e infissa, usando le visite rispettivamente in postorder e inorder.

Sia E_v l'espressione associata al sottoalbero T_v , con v nodo di T .

4.2.4.2.1. Generazione dell'espressione in notazione infissa

Si utilizza lo schema della visita inorder.

Algoritmo infix(T, v, L)

Input: Parse Tree T for E , $v \in T$, Lista L .

Output: Aggiunta a L di E_v in notazione infissa.

```
if ( $T$ .isExternal( $v$ )) then{
     $L$ .addLast( $v$ .getElement());
}
else{
     $L$ .addLast('(');
    infix( $T$ ,  $T$ .left( $v$ ),  $L$ );
     $L$ .addLast( $v$ .getElement());
    infix( $T$ ,  $T$ .right( $v$ ),  $L$ );
     $L$ .addLast(')');
}
```

La *chiamata iniziale* per esprimere in notazione infissa tutta l'espressione E è `infix(T , T .root(), L)`, con $L = \emptyset$.

① Osservazione

T e L sono variabili globali.

4.2.4.2.1.1. Complessità di `infix(T , T .root(), L)`

`infix(T , T .root(), L)` ha la stessa struttura della visita inorder, quindi la complessità è $\Theta(n + \sum_{v \in T} t_v)$, dove $n = |T|$ e t_v è il costo della visita di v , che in questo caso è $\Theta(1)$, quindi la complessità è $\Theta(n)$.

4.2.4.2.2. Generazione dell'espressione in notazione postfissa

Si utilizza lo schema della visita in postorder.

Algoritmo ipostix(T, v, L)

Input: Parse Tree T for E , $v \in T$, Lista L .

Output: Aggiunta a L di E_v in notazione postfissa.

```
if ( $T$ .isExternal( $v$ )) then{
     $L$ .addLast( $v$ .getElement()); //visita di  $v \in \Theta(1)$ 
}
else{
}
```

```

postfix(T, T.left(v), L);
postfix(T, T.right(v), L);
L.addLast(v.getElement()); //visita di v ∈ Θ(1)
}

```

La *chiamata iniziale* per esprimere in notazione postfissa tutta l'espressione E è `postfix(T, T.root(), L)`, con $L = \emptyset$.

4.2.4.2.2.1. Complessità di `postfix(T, T.root(), L)`

L'analisi è analoga a `infix`, quindi la complessità è $\Theta(n)$, con $n = |T|$

.

Riepilogo alberi binari

- Definizioni: albero binario proprio;
- Visita inorder e le loro applicazioni;
- Algoritmi di visita come template generali.

[lezione 13 & 14] - Risoluzione di esercizi

5. Priority queue

5.1. Entry

Una *Entry* è una coppia `(chiave, valore)`, dove la chiave proviene da un dominio K e il valore da un dominio V .

```

public interface Entry<K,V> {
    /** Returns the key of the entry */
    K getKey();
    /** Returns the value of the entry */
    V getValue();
}

```

5.1.1. Esempio

$\text{Entry} \equiv \text{STUDENTE} \begin{cases} \text{key} = \text{numero matricola} \\ \text{value} = \text{info come anagrafica, esami , ecc.} \end{cases}$

5.2. Priority Queue – definizione

Una *Priority Queue* è una collezione di entry le cui chiavi rappresentano *priorità* e provengono da un universo totalmente ordinato K .

Come tipo di dato astratto, la Priority Queue deve permettere di *trovare/rimuovere la entry di massima priorità* e *inserire una nuova entry*.

Le chiavi delle entry *non* sono necessariamente distinte.

Conventionalmente si assume che più piccolo è il valore della chiave e più alta è la priorità, ma vedremo anche utilizzi in cui vale l'opposto.

```
public interface PriorityQueue<K, V> {  
    int size();  
    boolean isEmpty();  
    /** Inserts and returns a new entry (key, value) */  
    Entry<K, V> insert(K key, V value);  
    /** Returns an entry with min key, without removing it */  
    Entry<K, V> min();  
    /** Returns and removes an entry with min key */  
    Entry<K, V> removeMin();  
}
```

⚠️ Osservazione

Se esistono più entry con chiave minima, `min` e `removeMin` ne restituiscono (e `removeMin` ne rimuove) una arbitraria.

5.2.1. Esempio

Le seguenti sono una sequenza di operazioni a partire da una Priority Queue vuota.

Operazione	Output	Priority Queue risultante
insert(5, A)	(5, A)	(5, A)
insert(9, C)	(9, C)	(5, A)(9, C)
insert(3, B)	(3, B)	(5, A)(9, C)(3, B)
insert(7, D)	(7, D)	(5, A)(9, C)(3, B)(7, D)
min()	(3, B)	(5, A)(9, C)(3, B)(7, D)

Operazione	Output	Priority Queue risultante
removeMin()	(3, B)	(5, A)(9, C)(7, D)
size()	3	(5, A)(9, C)(7, D)
isEmpty()	FALSE	(5, A)(9, C)(7, D)

💡 Applicazioni delle Priority Queue

- Algoritmo di Dijkstra per trovare i cammini minimi su un grafo;
- Pattern discovery;
- Scheduling di processi in sistema operativo o di richieste di banda nelle reti in base a priorità per garantire QoS;
- Simulazione discreta a eventi.

Lezione 15

5.3. Implementazione di Priority Queue tramite liste

5.3.1. Lista non ordinata

Sia P una lista *non ordinata* di n entry (`PositionalList<Entry<K, V>>`) . I metodi della Priority Queue vengono implementati nelle seguenti maniere:

Metodo `min()`

```
v <- P.first();
e <- v.getElement();
while (P.after(v) != null) do {
    v <- P.after(v);
    if (v.getElement().getKey() < e.getKey()) then {
        e <- v.getElement();
    }
}
return e;
```

Complessità: $\Theta(n)$, infatti devo vedere tutte le entry per trovare quella con chiave minima.

Metodo `insert(k, x)`

```

e <- (k,x);
P.addLast(e);
return e;

```

Complessità: $\Theta(1)$, perché posso inserire l'entry dove voglio.

Metodo removeMin()

```

v <- P.first();
e <- v.getElement();
while (P.after(v) != null) do {
    v <- P.after(v);
    if (v.getElement().getKey() < e.getKey()) then {
        e <- v.getElement();
    }
}
P.remove(v);
return e;

```

Complessità: $\Theta(n)$ come il caso di min.

5.3.2. Lista non ordinata

Sia P una lista *ordinata* (in senso crescente) di n entry (`PositionalList<Entry<K,V>>`) . I metodi della Priority Queue vengono implementati nelle seguenti maniere:

Metodo min()

```

return P.first().getElement();

```

Complessità: $\Theta(1)$, infatti la entry con chiave minima è in testa.

Metodo removeMin()

```

v <- P.first();
P.remove(P.first());
return v;

```

Complessità: $\Theta(1)$, come il caso di min.

Metodo insert(k,x)

```

e <- (k,x);
v <- P.first();
while (v != null) do{
    if (v.getElement().getKey() >= k) then {
        P.addBefore(v,e);
        return e;
    }
    else{
        v <- P.after(v);
    }
}
P.insertLast(e);
return e;

```

Complessità: $\Theta(n)$, perché devo trovare la posizione in cui inserire la entry.

📋 Riepilogo implementazione Priority Queue tramite Liste

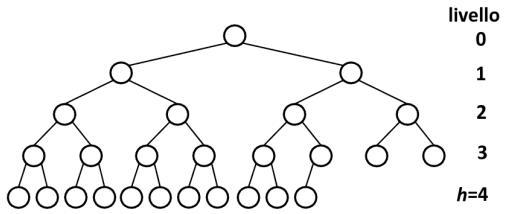
Lista	insert	min	removeMin
Non ordinata	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Ordinata	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

Per le strutture dati non discutiamo l'implementazione dei metodi size e isEmpty in quanto sono banali.

5.4. Implementazione di Priority Queue tramite Heap

Un *albero binario completo* T è un albero binario di altezza $h \geq 0$ tale che:

- $\forall i, 0 \leq i \leq h - 1$, il livello i ha 2^i nodi (ovvero il massimo numero);
- Al livello $h - 1$ tutti i nodi interni sono alla sinistra delle eventuali foglie e hanno tutti due figli tranne, eventualmente, quello più a destra che, se ha un solo figlio, ha il figlio sinistro.



☰ Proposizione

Un albero binario completo con n nodi ha altezza $h = \lfloor \log_2 n \rfloor$.

Osservo che al livello h (l'ultimo livello dell'albero) il numero di nodi è compreso tra 1 e 2^h .

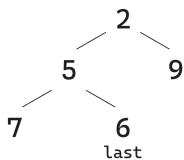
$$\begin{aligned} 1 + \sum_{i=0}^{h-1} (2^i) &\leq n \leq 2^h + \sum_{i=0}^{h-1} (2^i) = \sum_{i=0}^h (2^i) \\ \iff 1 + 2^h - 1 &= 2^h \leq n \leq 2^{h+1} - 1 \\ \iff h &\leq \log_2(n) < h + 1 \\ \implies \lfloor \log_2(n) \rfloor &= h \end{aligned}$$

□

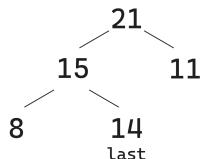
➊ Osservazione

$\forall n \geq 1$ esiste un unico albero binario completo con n nodi.

Un *min-heap* (o semplicemente *heap*) è un albero binario completo in cui ogni nodo v memorizza una entry e soddisfa la *heap-order property*: la chiave in v è *minore o uguale* della chiave in ciascun figlio di v .



Un *max-heap* è uno heap in cui la definizione della heap-order property cambia: la chiave in v è *maggior o uguale* della chiave in ciascun figlio di v .



Il *nodo last* di uno heap di altezza h è il nodo più a destra del livello h .

! Osservazione

Uno heap è caratterizzato da due proprietà: è un *albero binario completo* e segue la *heap-order property* (per ogni nodo).

5.4.1. Proprietà

Sia P uno heap con n entry. Dalla definizione si ricavano facilmente le seguenti proprietà.

1. Le chiavi incontrate lungo un cammino dalla radice verso le foglie formano una sequenza non decrescente;
2. Per qualsiasi discendente u di un nodo $v \in P$ si ha che $e_u.getKey() \geq e_v.getKey();$
3. La radice contiene una entry con chiave minima;
4. Se le chiavi sono tutte distinte, la entry con chiave massima (e_{max}) si trova in una foglia di P .

La prima e seconda proprietà sono conseguenze immediate della heap-order property, mentre la terza è un corollario della seconda in quanto ogni nodo è discendente della radice.

Si prova la quarta proprietà per assurdo, infatti se e_{max} avesse un discendente e , si avrebbe che la chiave di e dovrebbe essere maggiore della chiave di e_{max} , che contraddice l'ipotesi.

□

! Osservazione

Da queste proprietà si evince che uno heap non assicura necessariamente un ordinamento totale tra le entry, ma assicura l'ordinamento solo lungo percorsi radice-foglia.

5.4.2. Implementazione di alberi binari su array

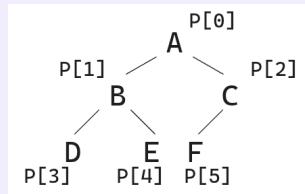
Il seguente schema (*level numbering*) consente di mappare un albero binario su un array $P = P[0], P[1], \dots$:

- Radice $\rightarrow P[0];$
- Figli di $P[i]$ $\rightarrow P[2i+1], P[2i+2];$
- Padre di $P[i]$ $\rightarrow P[\lfloor \frac{i-1}{2} \rfloor].$

! Osservazione

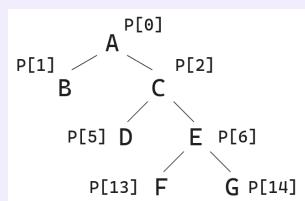
La rappresentazione è *space-efficient* per alberi molto bilanciati (ad esempio per alberi binari completi), ma non lo è affatto per alberi sbilanciati, come si vede nei seguenti esempi.

5.4.2.1. Albero molto bilanciato



$P \equiv A \ B \ C \ D \ E \ F$, il numero di nodi dell'albero è 6, come $|P| = 6$, quindi c'è la massima space-efficiency.

5.4.2.2. Albero molto sbilanciato



$P \equiv A \ B \ C \ [] \ [] \ D \ E \ [] \ [] \ [] \ [] \ F \ G$, il numero di nodi dell'albero è 7, mentre $|P| = 15$; ho una space-efficiency minima.

5.4.3. Implementazione di alberi binari completi su array

È facile dimostrare che usando il level numbering per mappare un albero binario completo con $n \geq 1$ nodi e altezza h su un array P , si ha che:

- $\forall i, 0 \leq i < h$, i 2^i nodi del livello i presi da sinistra a destra, sono mappati in $P[2^i - 1], P[2^i], \dots, P[2^{i+1} - 2]$;
- I nodi del livello h , presi da sinistra a destra, sono mappati in $P[2^h], \dots, P[n - 1]$. Il nodo last è quindi mappato in $P[n - 1]$.

Livello	Indici
0	0
1	$1 \div 2$
2	$3 \div 6$

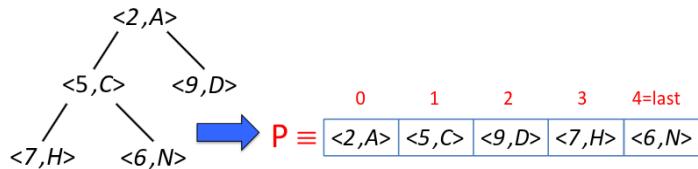
Livello	Indici
:	:
j	$2^j - 1 \div 2^{j+1} - 2$
:	:
$h - 1$	$2^{h-1} - 1 \div 2^h - 2$
h	$2^h - 1 \div n - 1$

! Osservazione

Il level numbering definisce quindi una corrispondenza 1:1 (biunivoca) tra array e alberi binari completi.

5.4.4. Implementazione di heap su array

Se non diversamente specificato, assumeremo sempre che uno heap sia realizzato tramite array.



5.4.4.1. Esercizio

Sia P un array di n entry $P[0], P[1], \dots, P[n - 1]$ le cui chiavi sono in ordine non decrescente. P rappresenta uno heap? Motivare la risposta.

In generale è vero dato che se vedo l'albero binario completo che corrisponde a P , le entry nel nodo associato a $P[i]$ avrà chiave minore o uguale delle entry nei nodi associati ai figli di $P[i]$ ($P[2i + 1], P[2i + 2]$) grazie all'ordinamento di P .

! Osservazione

Non è vero il viceversa, ovvero che uno heap sia sempre associato a un array ordinato.

5.4.5. Implementazione dei metodi della Priority Queue su heap

💡 Notazione

- $P[i] \equiv \text{entry}$, la cui chiave si ottiene con `P[i].getKey()` e il valore `P[i].getValue()`;
- $P[\text{last}]$ è la entry più a destra al livello h .

Sia P uno heap con n entry implementato tramite array.

5.4.5.1. min

Metodo `min()`

```
return P[0];
```

Complessità: $\Theta(1)$;

5.4.5.2. insert

Per realizzare il metodo `insert` inseriamo la nuova entry come successore (nel level numbering) del nodo `last`, poi ricostruiamo la heap-order property dello heap lungo il cammino dal nodo `last` alla radice.

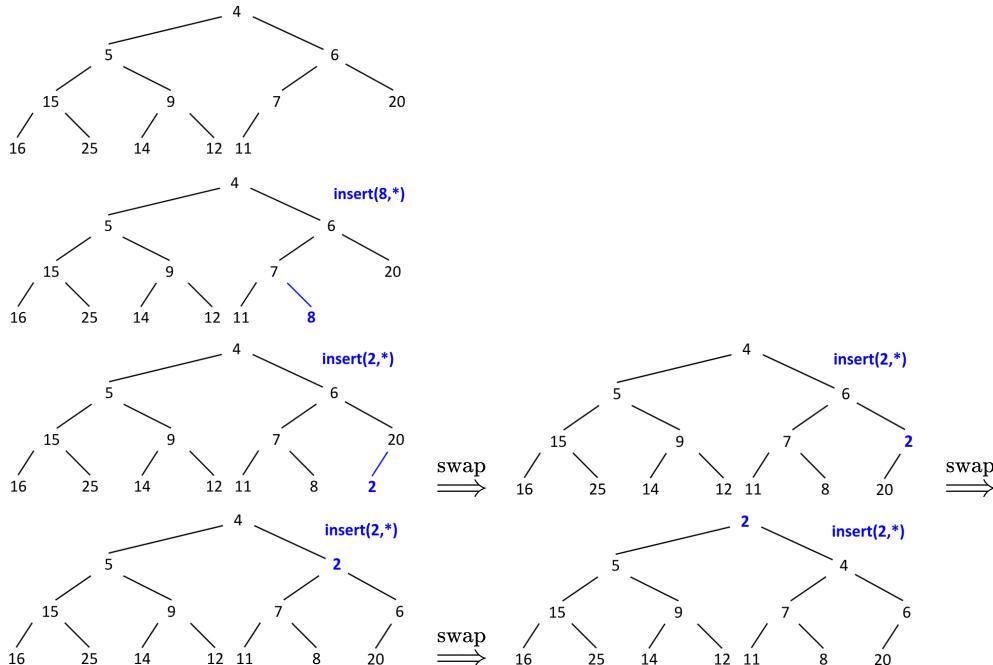
Metodo `insert(k, x)`

```
e <- (k, x);
P[++last] <- e;
i <- last;
//Up-heap bubbling
while ((i > 0) AND (P[ \lfloor (i-1)/2 \rfloor ].getKey() > P[i].getKey())) do {
    swap(P[i], P[ \lfloor (i-1)/2 \rfloor ]);
    i <- \lfloor (i-1)/2 \rfloor;
}
return e;
```

⚠️ Osservazione

In caso di overflow, si devono prima trasferire le entry in un array più capiente (di solito di taglia doppia di quello corrente) .

5.4.5.2.1. Esempio (solo chiavi)



5.4.5.2.3. Complessità

Consideriamo l'esecuzione di `insert` su uno heap P con n entry, e sia $h = \lfloor \log_2(n+1) \rfloor$ l'altezza risultante dopo l'inserimento.

- La complessità è proporzionale al numero di iterazioni del while, dato che ogni iterazione richiede $\Theta(1)$ operazioni;
- Il numero di iterazioni del while è $\leq h$;
- Esiste un'istanza che richiede esattamente h iterazioni (quella in cui si inserisce una entry con chiave minore di tutte quelle presenti).
 \Rightarrow Complessità $\in \Theta(\log n)$.

! Osservazione

La complessità non tiene conto del costo del trasferimento delle entry in un array più grande nel caso in cui l'array si riempia (*overflow*). Tuttavia, è facile vedere che raddoppiando la taglia dell'array ogni volta che si presenta un overflow, il costo di ciascun overflow non supera asintoticamente il costo aggregato delle precedenti invocazioni di `insert`, quindi può essere nascosto (*ammortizzato*) da quest'ultimo.

5.4.5.3. removeMin

Per realizzare `removeMin` rimuoviamo la entry presente nella radice dello heap, mettiamo la entry $P[\text{last}]$ nella radice, poi ricostruiamo la heap-order property dello heap a partire dalla radice verso le foglie.

Sia `indexMinChild(P, i)` un metodo che restituisce l'indice del figlio di $P[i]$ con chiave minima ($2i+1$ o $2i+2$), se $P[i]$ è un nodo interno (cioè $2i+1 \leq \text{last}$), e restituisce null se $P[i]$ è foglia.

Metodo `removeMin()`

```
minentry <- P[0];
P[0] <- P[last--];
i <- 0;
j <- indexMinChild(P, i);
//Down-heap bubbling
while ((j != null) AND (P[i].getKey() > P[j].getKey())) do{
    swap(P[i], P[j]);
    i <- j;
```

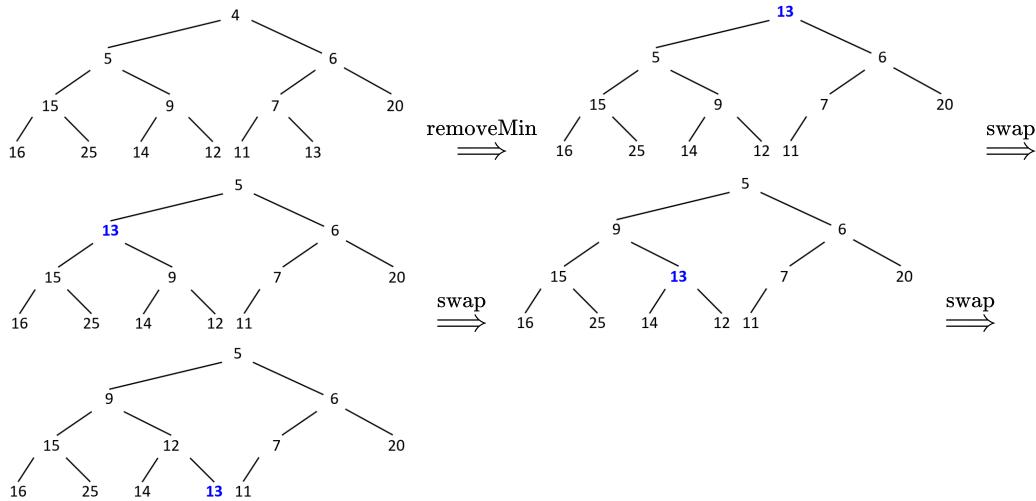
```

    j <- indexMinChild(P, i);
}

return minentry;

```

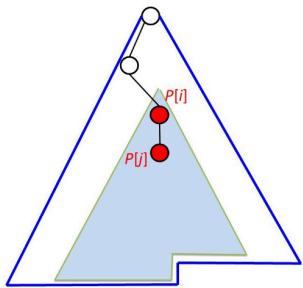
5.4.5.3.1. Esempio (solo chiavi)



5.4.5.3.2. Correttezza

La correttezza discende dal seguente invarianto per il while:

- Sia $P[j]$ figlio di $P[i]$ con chiave minima (se $P[i]$ è interno);
- Le uniche coppie antenato-discendente che possono violare la heap-order property estesa sono coppie in cui l'antenato è $P[i]$.



5.4.5.3.3. Complessità

Consideriamo l'esecuzione di `removeMin` su uno heap P con n entry; sia $h = \lfloor \log_2(n - 1) \rfloor$ l'altezza risultante dopo la rimozione.

- Complessità proporzionale al numero di iterazioni del while, dato che ogni iterazione richiede $\Theta(1)$;
 - Numero di iterazioni del while $\leq h$;
 - Esiste un'istanza che richiede esattamente h iterazioni (quella in cui la entry in $P[\text{last}]$) ha chiave maggiore di tutte quelle presenti).
- ⇒ Complessità $\in \Theta(\log n)$.

Riepilogo sulle implementazioni della Priority Queue

Struttura	min	insert	removeMin
Lista non ordinata	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
Lista ordinata	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
Heap (implementazione efficiente in spazio su array)	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$

5.5. Costruzione di uno heap a partire da n entry date

La specifica input-output per il problema è la seguente:

Input: Array P con n entry $P[0 \div n - 1]$.

Output: Array P riorganizzato per rappresentare uno heap.

Un algoritmo si dice *in-place* se usa $O(1)$ memoria aggiuntiva oltre a quella necessaria per l'input.

Per esempio: mergeSort non è un algoritmo *in-place*, mentre insertionSort lo è.

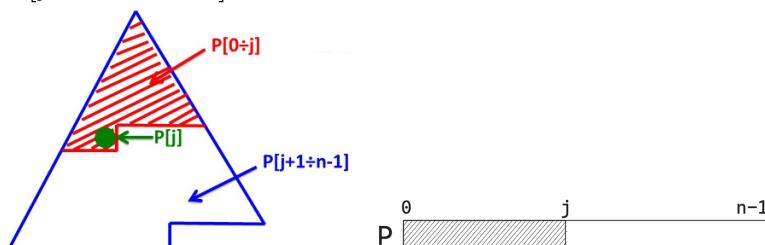
5.5.1. Soluzioni banali

1. Ordinare P in senso non decrescente usando insertionSort (tempo $\Theta(n^2)$, *in-place*) o mergeSort (tempo $\Theta(n \log n)$, non *in-place*);
2. Trasferisco le entry da P a un array di appoggio Q e lo rimetto in P invocando n volte insert (tempo $\Theta(n \log n)$, lo vedremo, non è *in-place*).

5.5.2. Approccio top-down

Con questo metodo si eseguono $n - 1$ iterazioni successive, mantenendo il seguente invarianto alla fine di ciascuna iterazione j , con $1 \leq j \leq n - 1$:

- $P[0 \div j]$ contiene le stesse entry iniziali, riordinate in modo da formare uno heap;
- $P[j + 1 \div n - 1]$ rimane immutato.



Per l'implementazione si effettua un up-heap bubbling da $P[j]$ in ciascuna iterazione j .

```

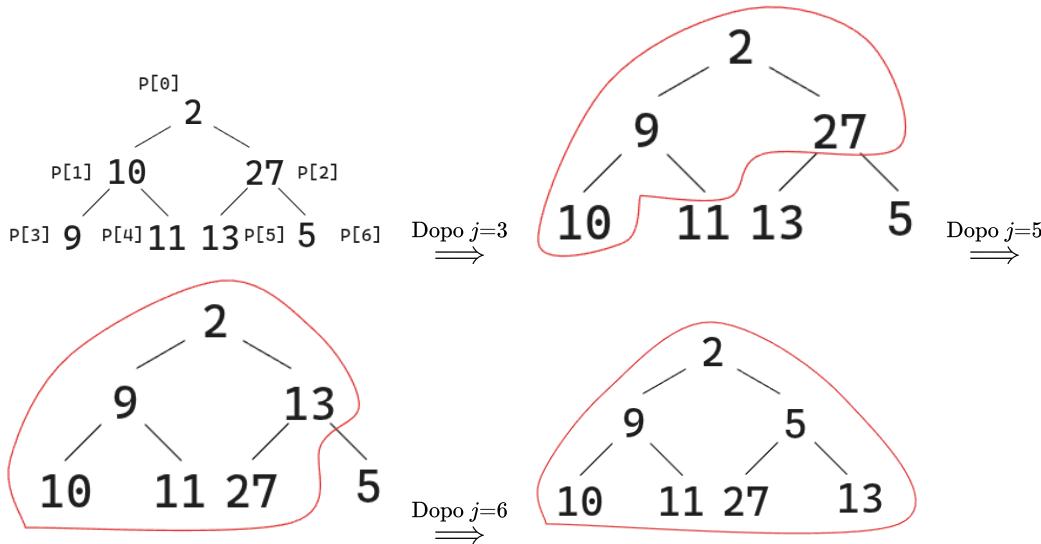
for j <- 1 to n-1 do{
    //Up-heap bubbling a partire da P[j]
    i <- j;
    while ((i > 0) AND (P[\lfloor(i-1)/2\rfloor].getKey() > P[i].getKey())) do{
        swap(P[i], P[\lfloor(i-1)/2\rfloor]);
        i <- \lfloor(i-1)/2\rfloor;
    }
}
last <- n-1;

```

La *correttezza* discende immediatamente dall'invariante.

5.5.2.1. Esempio

$$P \equiv [2 \ 10 \ 27 \ 9 \ 11 \ 13 \ 5]$$



5.5.2.2. Complessità

Dimostriamo che la complessità è $\Theta\left(\sum_{j=0}^{n-1} \log j\right)$:

- Vale che $O\left(\sum_{j=1}^{n-1} \log j\right)$: la dimostrazione è banale, dato che l'iterazione j equivale a inserire $P[j]$ in $P[0 \div j - 1]$;
- Vale che $\Omega\left(\sum_{j=1}^{n-1} \log j\right)$: considerando come istanza "cattiva" quella in cui P è inizialmente ordinato in senso decrescente.

Dimostriamo ora che $\sum_{j=1}^{n-1} \log j \in \Theta(n \log n)$

! Osservazione

La base (non indicata) dei logaritmi è 2, ma comunque la prova vale per qualsiasi base costante.

5.5.2.2.1. Dimostrazione

Dimostriamo prima che $\sum_{j=1}^{n-1} \log j \in O(n \log n)$:

$$\sum_{j=1}^{n-1} \log j \leq \sum_{j=1}^{n-1} \log n = (n-1) \log n < n \log n \implies \sum_{j=1}^{n-1} \log j \in O(n \log n).$$

Ora dimostriamo che $\sum_{j=1}^{n-1} \log j \in \Omega(n \log n)$:

$$\sum_{j=1}^{n-1} \log j \geq \sum_{j=\lfloor \frac{n}{2} \rfloor}^{n-1} \log j \geq \sum_{j=\lfloor \frac{n}{2} \rfloor}^{n-1} \log \lfloor \frac{n}{2} \rfloor \geq \lfloor \frac{n}{2} \rfloor \cdot \log \lfloor \frac{n}{2} \rfloor \implies \sum_{j=1}^{n-1} \log j \in \Omega(n \log n).$$

□

L'analisi mostra che la *complessità* della costruzione top-down di un heap a partire da un array di n entry è $\Theta(n \log n)$.

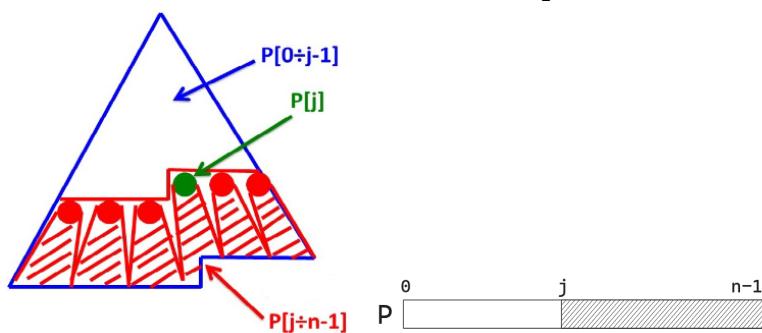
Inoltre dimostra che la complessità richiesta dall'invocazione di n insert in un heap inizialmente vuoto è $\Theta(n \log n)$ (ovvero la seconda soluzione banale).

Lezione 17

5.5.3. Approccio bottom-up

Con questo metodo si eseguono $\lfloor \frac{n}{2} \rfloor$ iterazioni successive, mantenendo il seguente invarianto alla fine di ciascuna iterazione j , con $\lfloor \frac{n-2}{2} \rfloor \geq j \geq 0$:

- $P[0 \div j-1]$ rimane immutato;
- $P[j \div n-1]$ contiene le stesse entry iniziali, riordinate in modo da essere una foresta di heap.



Per l'implementazione si effettua un down-heap bubbling da $P[j]$ in ciascuna iterazione j .

Ricorda

$P[\lfloor \frac{n-2}{2} \rfloor]$ è il nodo interno più a destra del penultimo livello.

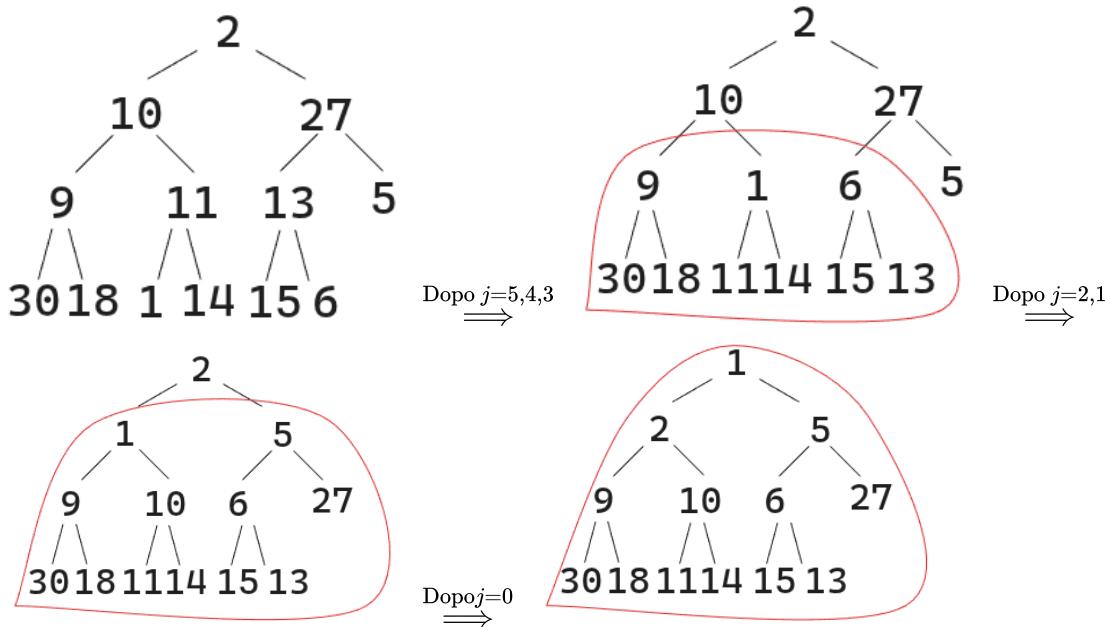
```

last <- n-1;
for j <- \floor{(n-2)/2} downto 0 do{
    //Down-heap bubbling a partire da P[j]
    i <- j;
    k <- indexMinChild(P,i);
    while ((k != null) AND (P[i].getKey() > P[k].getKey())) do{
        swap(P[i], P[k]);
        i <- k;
        k <- indexMinChild(P,i);
    }
}

```

La *correttezza* discende immediatamente dall'invariante.

5.5.3.1. Esempio



5.5.3.2. Complessità

Sia $t_{P[j]}$ il costo del down-heap bubbling a partire da $P[j]$.

- Per ogni nodo al livello i , $0 \leq i \leq h - 1$ ($h = \lfloor \log_2 n \rfloor$) il down-heap bubbling costa $O(h - i)$;
- Ci sono 2^i nodi al livello i .

La complessità è quindi $O\left(\sum_{j=0}^{\lfloor(n-2)/2\rfloor} t_{P[j]}\right) = O\left(\sum_{i=0}^{h-1} 2^i(h-i)\right)$.

Dimostriamo ora che $\sum_{i=0}^{h-1} 2^i(h-i) \in O(n)$, che implica che la complessità totale della costruzione bottom-up dello heap è $O(n)$ ($\implies \Theta(n)$).

5.5.3.2.1. Dimostrazione

Dimostriamo prima che $\sum_{l=1}^h l\left(\frac{1}{2}\right)^l < 3$:

Osserviamo che $\forall l \geq 1$ vale $l\left(\frac{1}{2}\right)^l \leq \left(\frac{3}{4}\right)^l$, dimostrabile per induzione.

$$\sum_{l=1}^j l\left(\frac{1}{2}\right)^l \leq \sum_{l=1}^h \left(\frac{3}{4}\right)^l = \frac{\left(\frac{3}{4}\right)^l - \frac{3}{4}}{\frac{3}{4} - 1} = \frac{\left(\frac{3}{4}\right)^l - \left(\frac{3}{4}\right)^{h+1}}{1 - \frac{3}{4}} < \frac{\frac{3}{4}}{\frac{1}{4}} = 3.$$

□

Dimostriamo quindi che $\sum_{i=0}^{h-1} 2^i(h-i) \in O(n)$:

$$\sum_{i=0}^{h-1} 2^i(h-i) = 2^h \sum_{i=0}^{h-1} \frac{2^i}{2^h}(h-i) = 2^h \cdot \sum_{i=0}^{h-1} \frac{h-i}{2^{h-i}}$$

Cambio di variabile $l = h-i$:

$$2^h \cdot \sum_{i=0}^{h-1} \frac{h-i}{2^{h-i}} = 2^h \sum_{l=1}^h l\left(\frac{1}{2}\right)^l < 2^h \cdot 3$$

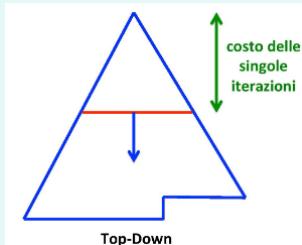
$$h = \lfloor \log_2 n \rfloor \implies 2^h \cdot 3 \in O(n)$$

$$\implies \sum_{i=0}^{h-1} 2^i(h-i) \in O(n).$$

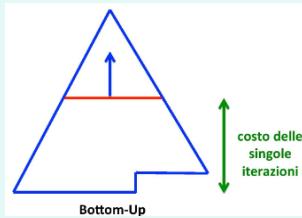
□

⌚ Confronto approcci top-down e bottom-up

Nell'approccio *top-down* vi sono 2^i operazioni di costo $\Theta(i)$, quindi *più aumenta la taglia del livello, più aumenta il costo dell'up-heap bubbling*. In tutto sarà $\Theta(n \log n)$.



Nell'approccio *bottom-up* vi sono 2^i operazioni di costo $\Theta((\log n) - i)$, quindi *più aumenta la taglia del livello, più aumenta il costo del down-heap bubbling*. In tutto sarà $\Theta(n)$.



Entrambe le soluzioni possono essere eseguite direttamente su array senza utilizzare spazio aggiuntivo, quindi sono *in-place*.

5.6. Sorting tramite Priority Queue

Sia $S = S[0]S[1]\dots S[n-1]$ una sequenza di n chiavi da ordinare.

Algoritmo pqSort(S)



Si divide l'algoritmo in due fasi. Nella *fase A* si inseriscono le n chiavi in P una alla volta, invocando il metodo `insert` (considerando le chiavi come entry); nella *fase B* si rimuovono le n chiavi da P una alla volta, invocando il metodo `removeMin`.

5.6.1. Complessità di psSort(S)

- Sia P una *lista non ordinata*: La fase A ha complessità $\Theta(n)$, la fase B $\Theta(\sum_{i=1}^n i) \in \Theta(n^2)$ (viene effettuata con un `SelectionSort`);
- Sia P una *lista ordinata*: La fase A ha complessità $\Theta(\sum_{i=1}^n i) \in \Theta(n^2)$ (viene effettuata con un `InsertionSort`), la fase B $\Theta(n)$;
- Sia P uno heap (su array): Sia la fase A che la B hanno complessità $\Theta(\sum_{i=1}^n \log i) \in \Theta(n \log n)$ (vengono effettuate con un `HeapSort`). Con una costruzione bottom-up, la fase A scende a $\Theta(n)$, mentre la B rimane a $\Theta(n \log n)$.

⚠️ Osservazione

`InsertionSort` e `SelectionSort` possono essere implementati in-place in modo semplice. Ora vediamo come implementare `HeapSort`.

5.6.2. HeapSort in-place

Per realizzare `HeapSort` in-place, implementiamo una variante di `HeapSort` in-place usando la stessa sequenza S come sequenza di input, priority queue e sequenza di output. La variazione rispetto a quella presentata prima consiste nell'usare un max-heap invece che uno heap standard.

Nella fase A riorganizziamo $S[0 : n - 1]$ in modo che le chiavi rappresentino un max-heap (la chiave in un nodo interno è maggiore o uguale delle chiavi nei figli).

Nella fase B si riorganizza $S[0 : n - 1]$ in modo che le chiavi risultino ordinate.

5.6.2.1. Fase A: $S \rightarrow$ max-heap

La trasformazione di S in un max-heap è implementata come segue.

Sia `indexMaxChild(S, i)` un metodo che restituisce l'indice del figlio di $S[i]$ con chiave massima ($2i + 1$ o $2i + 2$) se $S[i]$ è un nodo interno

(cioè $2i + 1 \leq \text{last}$), se invece $S[i]$ è foglia restituisce `null`.

```

last  <- n-1;
for j <- \floor{(n-2)/2} downto 0 do{
    //Down-heap bubbling a partire da S[j]
    i <- j;
    k <- indexMaxChild(S,i);
    while ((j != null) AND (S[i] < S[k])) do {
        swap(S[i], S[k]);
        i <- k;
        k <- indexMaxChild(S,i);
    }
}
}

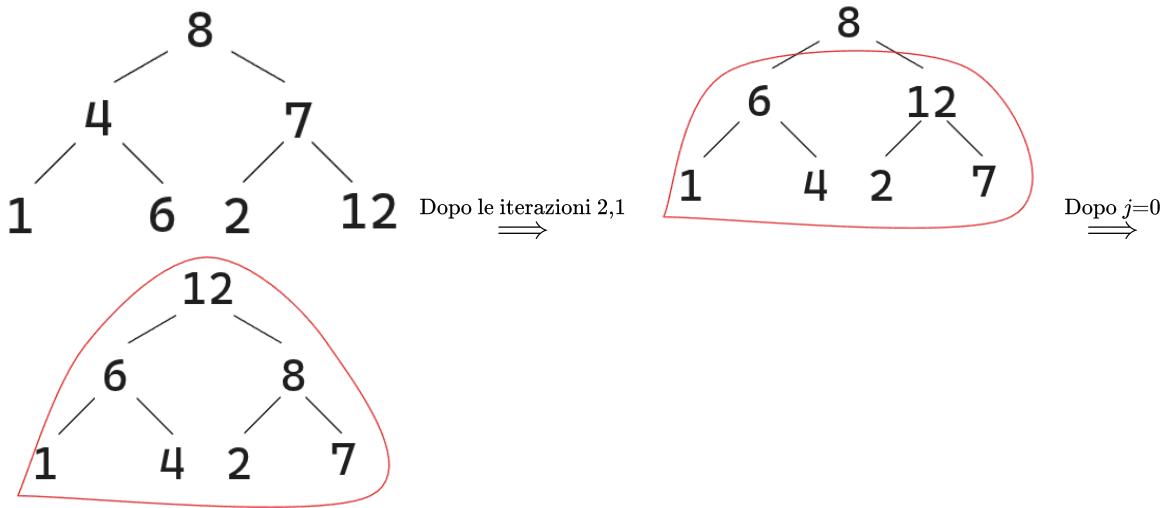
```

① Osservazione

È la costruzione bottom-up di un max-heap (considerando solamente le chiavi).

5.6.2.1.1. Esempio

$$S = [8 \ 4 \ 7 \ 1 \ 6 \ 2 \ 12]$$

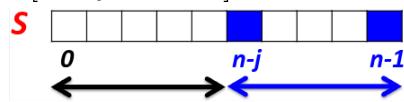


5.6.2.2. Fase B max-heap $\rightarrow S$ ordinata

La fase B è basata sul un ciclo `for` di $n - 1$ iterazioni, che mantiene il seguente invarianto alla j -esima iterazione ($j = 0, \dots, n - 1$, dove $j = 0$ è l'inizio del ciclo).

- $S[0 : n - j - 1]$ contiene le $n - j$ chiavi più piccole, organizzate come max-heap;

- $S[n - j \div n - 1]$ contiene le j chiavi più grandi in ordine crescente.



L'implementazione è la seguente.

```

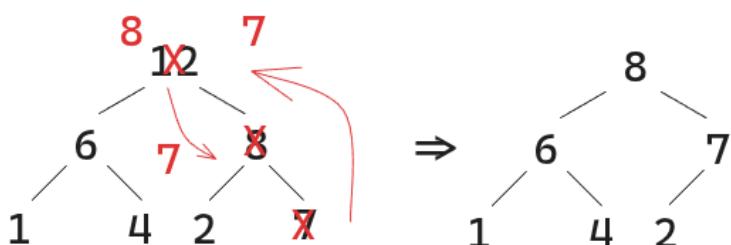
last <- n-1; //segnalà l'ultima cella di S che fa parte del max-heap
for j <- 1 to n-1 do{
    //Down-heap bubbling a partire da S[0]
    swap(S[last], S[0]);
    last <- n-j-1;
    i <- 0;
    k <- indexMaxChild(S,i); //deve tenere conto che il confine del max-
    heap segnalato da last arretra in ciascuna iterazione
    while((k != null) AND (S[i] < S[k])) do{
        swap(S[i], S[k]) do {
            swap(S[i], S[k]);
            i <- k;
            k <- indexMaxChild(S,i);
        }
    }
}
}

```

5.6.2.2.1. Esempio

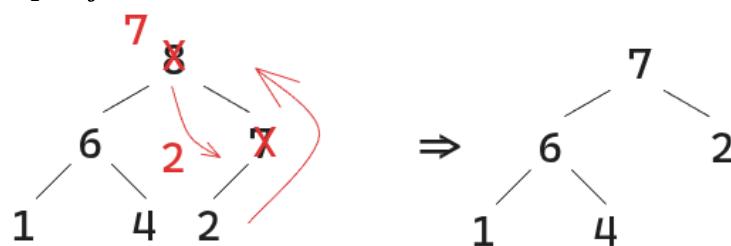
$S \equiv [12\ 6\ 8\ 1\ 4\ 27]$ ottenuto dopo la fase A.

Dopo l'iterazione $j = 1$ diventa :



$S \equiv [8\ 6\ 7\ 1\ 4\ | 12]$

Dopo $j = 2$ diventa :



$S \equiv [7\ 6\ 2\ 1\ 4\ | 8\ 12]$

5.6.2.3. Complessità

- La fase A equivale alla costruzione bottom-up $\Rightarrow \Theta(n)$;
- La fase B equivale all'esecuzione di $n - 1$ `removeMax` da heap progressivamente più piccoli $\Rightarrow \Theta\left(\sum_{i=1}^{n-1} \log i\right) \in \Theta(n \log n)$.
 \Rightarrow La complessità di `HeapSort` in place è $\Theta(n \log n)$.

[lezione 18] - Risoluzione di esercizi

Lezione 19

5.7. Implementazione di Priority Queue in Java

Il package `java.util` contiene la classe `PriorityQueue<E>`, i cui oggetti sono Priority Queue Q implementate tramite heap.

Le sue caratteristiche sono:

- Q non contiene coppie `(chiave, valore)`, ma oggetti di tipo E ;
- La priorità è definita utilizzando tutto l'oggetto come chiave, come se gli elementi di Q fossero le chiavi di entry vuote;
- Di default, gli elementi di Q sono confrontati in base all'ordine naturale associato al tipo E ;

In pratica, se vogliamo usare la classe per realizzare una Priority Queue le cui entry sono coppie `(chiave, valore)` dobbiamo rappresentare le entry tramite una classe che estende l'interfaccia `Comparable`, definire il metodo `compareTo` imponendo un *ordinamento naturale delle entry* basato sulle chiavi.

5.7.1. Esempio

```
class MyEntry implements Comparable<MyEntry>{  
    /* Variabili e metodi che definiscono la entry */  
    @Override  
    public int compareTo(MyEntry x) {  
  
        /* Confronto tra chiamante e x basato sulla chiave */  
  
    }  
}
```

E creiamo la Priority Queue come segue: `PriorityQueue<MyEntry> Q = new PriorityQueue<MyEntry>()` .

A questo punto, il metodo `Q.poll()` restituirà la entry con chiave minima.

Riepilogo generale su Priority Queue

- Priority Queue: definizione e implementazione tramite liste;
- Albero binario completo: definizione e altezza;
- Heap: definizione, mapping efficiente su array tramite level numbering;
- Implementazione dei metodi della Priority Queue tramite heap (su array);
- Costruzione (top-down e bottom-up) di uno heap partendo da un array di n entry;
- `pqSort`
 - Implementazione con lista non ordinata (`SelectionSort`);
 - Implementazione con lista ordinata (`InsertionSort`);
 - Implementazione con heap su array (`HeapSort`).

6. Mappe

6.1. Definizione, interfaccia, applicazioni

Una Mappa è una collezione di entry che permette di ricercare inserire e rimuovere entry in base alle loro chiavi (come un *indice*).

Varie applicazioni sono:

- Database (Ed: studenti su Uniweb, con chiave il numero di matricola e come valore tutte le informazioni dello studente);
- Compilatori (Es: per il type checking di variabili, che hanno come chiave il nome della variabile, mentre come valore il tipo);
- Motori di ricerca (Es: le liste invertite, che hanno come chiave una parola, mentre come valore una lista di documenti, come le pagine web);
- Data analysis (Es: conteggio di frequenze di oggetti, che hanno come chiave un oggetto e come valore il numero di occorrenze).

Una *Mappa* è una collezione di entry con *chiavi distinte* provenienti da un universo U su cui è definito l'operatore " $=$ ", che supporta i metodi `get`, `put` e `remove`.

```

public interface Map<K, V> {
    int size();
    boolean isEmpty();
    V get (K key);
    V put (K key, V value);
    V remove (K key);
    Iterable<K> keySet();
    Iterable<V> values();
    Iterable<Entry<K,V>> entrySet();
}

```

- `get(K key)` : se esiste `(key, x)` restituisce `x`, altrimenti restituisce `null`;
- `put(K key, V value)` : se esiste `(key, x)` mette `value` al posto di `x` restituisce `x`, altrimenti inserisce l'entry `(key, value)` e restituisce `null`;
- `remove(K key)` : se esiste `(key, x)` rimuove la entry e restituisce `x`, altrimenti restituisce `null`;
- `keySet()`, `values()`, `entrySet()` : restituiscono strutture (`Iterable`) contenti, rispettivamente, le chiavi, i valori e le entry della mappa che possono essere enumerate da iteratori (`iterator`).

La *mappa* è vista come *associative array*, nel senso che la chiave della entry è usata come un "indice" di accesso alla mappa.

6.2. Implementazioni semplici, inefficienti

6.2.1. Lista non ordinata

Sia n il numero di entry nella Mappa.

Allora la complessità di `get`, `put` e `remove` è $\Theta(n)$, infatti tutti e tre i metodi richiedono la ricerca di una entry con la chiave data, se una tale entry non c'è devono guardarla tutte. Sono quindi *inefficienti* nel *tempo*.

6.2.2. Array di taglia $|U|$

Si assume che U sia finito e che sia disponibile un mapping 1:1 tra U e gli indici in $[0, |U|-1]$. Tale mapping potrebbe non essere banale da trovare.

La complessità di `get`, `put` e `remove` p $\Theta(1)$.

Risulta *inefficiente* nello *spazio* se $|U|$ è molto più grande del numero di entry della mappa.

6.2.2.1. Esempio

Una Mappa per tutti i cittadini italiani ha $n \approx 6 \cdot 10^7$, la chiave può essere il codice fiscale (ovvero una stringa di 16 caratteri alfanumerici provenienti da un alfabeto di 36 caratteri). Allora ci sarebbe $|U| = 36^{16} \approx 8 \cdot 10^{24}$.

Il mapping tra U e gli indici in $[0, |U| - 1]$ si può ottenere venendo un codice fiscale come un intero rappresentato in base 36, usando un semplice mapping tra lettere/cifre e interi in $[0, 35]$.

0	1	2	3	4	5	6	7	8	9	A	B	C	D	...	Z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	...	25

! Osservazione

L'universo delle chiavi *non è necessariamente ordinato*. In caso lo sia, si parlerebbe di *Mappa ordinata* ("Sorted Map" in inglese) e, in questo caso, sarebbero possibili implementazioni più efficienti al caso pessimo.

! Osservazione

La Mappa assume che le chiavi siano tutte distinte. Una variante della Mappa, chiamata *Multimap*, permette di avere più entry con la stessa chiave.

Noi studieremo implementazioni basate sulle tabelle hash per la Mappa e sugli alberi di ricerca (binari e non) per la Mappa ordinata; discuteremo anche come implementare una Multimap.

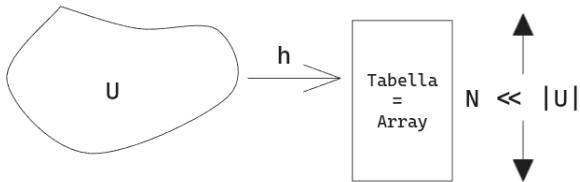
6.3. Mappe in Java

Nel package `java.util` troviamo:

- Interfaccia `Map<K,V>`, la quale contiene i metodi visti prima. Al suo interno è definita un'interfaccia `statica Map.Entry<k,v>`. `Map.Entry` è statica nel senso che è associata alla classe `Map` e non ai singoli oggetti della classe (in realtà tutte le interfacce nested sono static per default). `Map.Entry` può essere usata ovunque l'interfaccia `Map` sia visibile.
- Classe `HashMap<K,V>`, che è l'implementazione di una Mappa tramite tabella hash con separate chaining.
- Classe `TreeMap<K,V>`, che è l'implementazione di una Mappa tramite Red-Black Tree.

6.4. Implementazione Mappa tramite Tabella Hash

6.4.1. Tabelle Hash



L'obiettivo è ottenere prestazioni in tempo simili a quelle della soluzione tramite array di taglia $|U|$, ma garantendo efficienza in spazio.

In particolare, vogliamo un mapping (funzione h) da U a un array di taglia $N \ll |U|$ tale che un *qualsiasi* insieme di n entry, ovvero n chiavi di U , siano distribuite bene da h tra gli indici $[0, N - 1]$, possibilmente usando un valore N non troppo più grande di n . In questo modo si punta ad avere accesso a una *qualsiasi* delle n entry in tempo "quasi" costante, usando uno spazio non troppo maggiore di n .

Una Tabella Hash è definita dai seguenti tre ingredienti principali:

- *Funzione hash* $h : U = \{\text{chiavi}\} \rightarrow [0, N - 1]$.
La convenzione di Java stabilisce:
$$h : k \xrightarrow{\text{hash code}} \mathbb{Z} \xrightarrow{\text{compression function}} [0, N - 1]$$
- *Bucket Array* A di capacità N .
 $A[i]$ rappresenta l' i -esimo bucket al quale vengono associate tutte le entry $\langle k, v \rangle$ tali che $h(k) = i$ ($0 \leq i < N$).
- *Metodo di risoluzione delle collisioni* che sono costituite da chiavi distinte associate allo stesso bucket dalla funzione hash.

L'idea è di usare il bucket $A[i]$ per memorizzare tutte le entry $\langle k, v \rangle$ con $h(k) = i$ gestendo le collisioni con una struttura ausiliaria.

La scelta della funzione hash (nelle due componenti) diventa cruciale. In particolare: h deve essere veloce da calcolare e deve "assomigliare" il più possibile a un processo random (*uniform hashing*) che associa a ogni chiave in U un intero su $[0, N - 1]$ tale che $\forall k \neq k' \in U$ e $\forall i, j \in [0, N - 1]$ valga:

1. $Pr[h(k) = i] = \frac{1}{N}$: La probabilità che k sia assegnata al bucket $A[i]$ è la stessa per ogni $i \Rightarrow$ non ci sono bucket "privilegiati" da h ;

2. $Pr[h(k) = i | h(k') = j] = Pr[h(k) = i] = \frac{1}{N}$: Sapere che k' è mappata sul bucket $A[j]$ non cambia la probabilità che k sia mappata sul bucket $A[i]$ \Rightarrow *h offusca possibili correlazioni tra chiavi.*

! Osservazione

Le prestazioni di una Tabella Hash sono tanto migliori quanto più la funzione h assomiglia a una funzione random che soddisfa le due proprietà elencate.

Ω Metodo `HashCode` in Java

Il metodo `hashCode()` della classe `Object` restituisce un `int` che dipende dall'indirizzo in memoria dell'oggetto. Non assicura quindi che l'`hashCode` di un oggetto rimanga inalterato in diverse esecuzioni di un programma o in diverse implementazioni di Java. Questo non è un mapping "puro" da U a `int`. Il metodo `hashCode` può essere riscritto in vari modi a seconda del tipo di chiavi, restituendo sempre un `int`.

6.4.2. Generazione hash code

6.4.2.1. Hashcode per chiavi numeriche

Vediamo come trasformare in Java tipi numerici in `int` ($\mathbb{Z} \equiv \text{int}$):

- `byte`, `short`, `char`, `int` $k \rightarrow (\text{int})\ k;$
- `float` (32 bit) $k \rightarrow \text{Float.toFloatBits}(k);$
- `long` (64 bit) $k \rightarrow (\text{int}) ((k \gg 32) + (\text{int}) k)$, dove " $k \gg 32$ " rappresenta i 32 bit *più* significativi, mentre "`(int) k`" rappresenta i 32 bit meno significativi.

! Osservazione

Solo "`(int) k`" perde l'informazione di metà dei bit

- `double` (64 bit) $k \rightarrow \text{Double.doubleToLongBits}(k) \rightarrow \text{int}$, dove il metodo indicato restituisce `long`, poi si effettua la trasformazione come per `long`.

6.4.2.2. Hashcode per una stringa di `char`

Sia $S \equiv s_0s_1\dots s_{k-1}$ una stringa di `char`.

- *Hash code banale*: $h(S) = \sum_{i=0}^{k-1} s_i$.
Non è un buon hashcode, infatti $h(\text{stop}) = h(\text{spot}) = h(\text{tops})$.
- *Polynomial hash code*: $h(S) = \sum_{i=0}^{k-1} s_i \cdot a^{k-1-i}$.
Per le parole inglesi vanno bene valori $a = 31, 33, 37, 39, 41$. La classe `String` in Java implementa `hashCode` in questo modo con $a = 31$.

6.4.2.2.1. Hashcode basato su cyclic shift

Nell'hashcode basato su cyclic shift si sommano i singoli caratteri applicando dopo ogni addizione un cyclic shift alla somma parziale. In pratica, uno shift di cinque posizioni va bene.



Calcolo dell'hash code:

```

h <- s_0 //h a 32 bit
for i <- 1 to k-1 do{
    h <- (h << 5) ! (h >> 27); //cyclic shift di 5 posizioni a sinistra
    h <- h + s_i;
}
return h;

```

6.4.3. Compression function

6.4.3.1. Division Method

Dato i l'intero prodotto dall'hashcode: $i \rightarrow i \bmod N$, dove N è la capacità del Bucket Array.

! Osservazione

Per una migliore distribuzione degli hash code tra gli indici del Bucket Array, conviene scegliere N *primo e distante da una potenza di due*.

Delle scelte di N poco adeguate sarebbero:

- $N = 2^p \implies i \bmod N$ equivale a prendere i p bit meno significativi di i , mentre è meglio che $i \bmod N$ dipenda da tutti i bit di i .
- $N = 10^p \implies i \bmod N$ equivale a prendere le p cifre meno significative di i in base 10. Come prima no dipende da tutta la rappresentazione di i in base 10.

6.4.3.2. Multiply-Add-Divide (MAD) Method

In questo caso si ha $i \rightarrow [(ai + b) \bmod p] \bmod N$, dove $p > N$ con p primo, $a, b \in [0, p - 1]$ scelti a caso, $a > 0$.

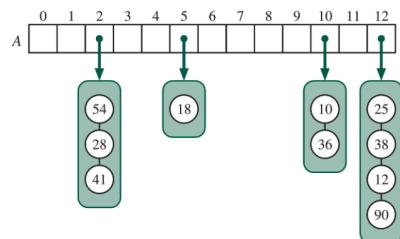
① Osservazione

Il metodo MAD, leggermente più costoso dal punto di vista computazionale, assicura una migliore distribuzione degli hash code tra gli indici del Bucket Array.

6.4.4. Risoluzione delle collisioni

Una *collisione* avviene quando si hanno $\langle k_1, v_1 \rangle, \langle k_2, v_2 \rangle$ con $k_1 \neq k_2$, ma $h(k_1) = h(k_2)$.

Nel *separate chaining* ogni bucket è visto come una Map più piccola, implementata tramite lista.



Nell'*open addressing* non si fa ricorso a strutture ausiliarie ma vengono memorizzate le entry direttamente nelle celle del Bucket Array. In questo modo si risparmia spazio, ma si complica la gestione delle collisioni (in caso di collisione, si usa una legge per cercare un'altra cella, occupando la prima cella libera). Non studieremo questo approccio.

Lezione 20

6.4.5. Implementazione dei metodi della Mappa

Si consideri l'implementazione di una Mappa tramite una tabella hash (A, h) con separate chaining.

Metodo get(k)

```
if (exists una entry (k,x) in A[h(k)]) then {
    return x;
}
else{
```

```

    return null;
}

```

Metodo put (k, v)

```

if ( $\exists$  una entry (k,x) in A[h(k)]) then {
    sostituisce x con v;
    return x;
}

else{
    inserisci (k,v) in cosa al bucket A[h(k)];
    incrementa di 1 la size della tabella;
    return null;
}

```

Metodo remove (k)

```

if ( $\exists$  una entry (k,x) in A[h(k)]) then {
    rimuovi (k,x) da A[h(k)];
    decerementa di 1 la size della tabella;
    return x;
}

else{
    return null;
}

```

6.4.5.1. Esercizio

Creare una tabella hash di capacità $N = 11$ con le $n = 11$ chiavi: 12, 44, 13, 88, 23, 94, 11, 39, 20, 16, 5. La funzione hash è: $h(k) = (3k + 5) \text{ mod } 11$.

Indice i	Bucket $A[i]$
0	13
1	94 39
2	
3	
4	
5	44 88 11
6	

Indice i	Bucket $A[i]$
7	
8	12 23
9	16 5
10	20

6.4.6. Load factor

Per una tabella hash di capacità N che memorizza n entry, il *load factor* λ è definito come

$$\lambda = \frac{n}{N}$$

In altre parole, λ rappresenta la lunghezza media di un bucket,

! **Osservazione**

Il load factor ha un impatti sgnificativo sulla efficienza computazionale di una tabella hash.

6.4.7. Complessità di `get`, `put` e `remove`

Studieremo la complessità al caso pessimo, in funzione del numero di entry presenti nella tabella hash, e la complessità al caso medio, in funzione del load factor λ della tabella hash.

⚡ **Nota bene**

Assumeremo sempre che il calcolo della funzione hash per un dato valore k della chiave richiede $O(1)$ operazioni.

6.4.7.1. Complessità al caso pessimo

Si consideri una tabella hash contenente n entry, si assuma che per una data chiave k il valore $h(k)$ sia calcolabile in tempo costante.

La complessità al caso pessimo di `get`, `put` e `remove` è $\Theta(n)$ ed è *dominata dalla ricerca* della entry con chiave k , necessaria per tutti e tre i metodi.

Che sia $O(n)$ è banale, mentre che sia $\Omega(n)$ motivata dalla seguente istanza:

- Tutte le entry sono nello stesso bucket in cui viene mappata la chiave k ;
- k non è presente nella tabella e quindi si devono guardare tutte le n entry nel bucket $A[h(k)]$.

6.4.7.2. Complessità al caso medio

☰ Teorema

Sotto l'ipotesi di uniform hashing, in una tabella hash con separate chaining e load factor λ , la complessità al caso medio di `get`, `put` e `remove` è

$$O(1 + \lambda)$$

Tale complessità vale per:

1. Qualsiasi chiave k non presente: la media è fatta su tutti i possibili valori di $h(k)$, che sono equiprobabili sotto l'ipotesi di uniform hashing;
2. Una chiave k presente: la media è fatta assumendo k scelta a caso, con probabilità uniforme tra le chiavi presenti nella tabella.

☰ Corollario

Quando $\lambda \in O(1)$, i tre metodi hanno complessità $O(1)$ al caso medio.

6.4.7.2.1. Dimostrazione

Dimostriamo il punto (1), saltiamo la dimostrazione di (2).

Assumiamo che k non sia presente.

Sia n_i il numero di entry presenti nel bucket di indice i , $0 \leq i < N$, allora $\sum_{i=0}^{N-1} n_i = n$.

La complessità dei tre metodi è dominata da quella richiesta per la ricerca di k , che è $O\left(\frac{1}{N} \sum_{i=0}^{N-1} (n_i + 1)\right)$.

- $\frac{1}{N}$ serve per fare la media su tutti i possibili bucket in cui può essere mappata k ;
- $\Theta(n_i + 1)$ è il numero di operazioni richieste per cercare k nel bucket i .

Si ha che:

$$\frac{1}{N} \cdot \sum_{i=0}^{N-1} (n_i + 1) \cdot \frac{1}{N} \cdot \sum_{i=0}^{N-1} n_i + \frac{1}{N} \sum_{i=0}^{N-1} 1 = \frac{n}{n} + 1 = \lambda + 1$$

□

6.4.8. Rehashing

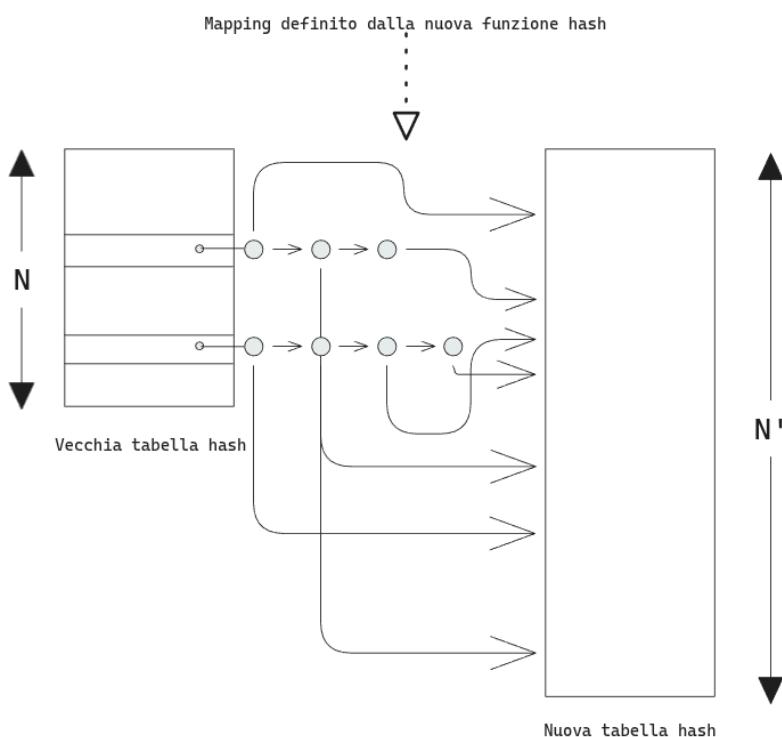
! Osservazione

In pratica si suggerisce di mantenere $\lambda < 0.9$.

In Java, la classe `HashMap` del package `java.util`, che implementa la tabella hash con separate chaining, usa $\lambda \leq 0.75$ come default.

Quando λ supera la soglia prefissata si esegue un *rehash*:

1. Creazione di un nuovo bucket array di capacità $N' \geq 2N$;
2. Scelta di una nuova funzione hash;
3. Trasferimento delle entry dalla vecchia alla nuova tabella hash.



! Osservazioni

- In un rehash può essere necessario cercare un numero primo $\geq 2N$, che potrebbe essere costoso (ma noi ignoriamo tale costo).
- (1) Il trasferimento di n entry da una tabella all'altra può essere implementato in tempo $\Theta(n)$ al caso pessimo.

- (2) Dato che un rehash si esegue quando λ supera una data costante, e la capacità del bucket array almeno raddoppia, il rehash di n entry è preceduto da $\Omega(n)$ put senza rehash. Quindi il costo del rehash viene *ammortizzato* (ovvero nascosto) da quello aggregato dei put.

6.4.8.1. Dimostrazione

Giustifichiamo le osservazioni (1) e (2).

Per giustificare (1) diciamo che per spostare le entry da una tabella all'altra, posso eseguire n inserimenti nelle nuove tabelle *disabilitando* il controllo se ogni nuova chiave inserita è già presente, perché so già che le n entry da trasferire hanno tutte chiavi distinte. Allora il trasferimento delle n entry costa $\Theta(n)$ al caso pessimo.

Per giustificare (2) analizziamo un generico rehash: $N \rightarrow N_{new} \geq 2N$. Sia $n = \lambda N$ il numero di entry da trasferire. Consideriamo due casi:

- Caso 1: quello considerato è il primo rehash eseguito, allora le n entry sono state tutte inserite in precedenza, senza rehash e prima del rehash considerato;
- Caso 2: quello considerato non è il primo rehash nella vita della mappa. Consideriamo allora gli ultimi due rehash eseguiti: $N_{old} \rightarrow N \geq 2N_{old} \rightarrow N_{new} \geq 2N$. Il primo dei due rehash ($N_{old} \rightarrow N$) ha trasferito $n_{old} = \lambda N_{old}$ entry; il secondo dei due rehash ($N \rightarrow N_{new}$) ha trasferito $n = \lambda N$ entry.

Allora tra i due rehash devono esserci stati almeno $n - n_{old}$ inserimenti e vale $n - n_{old} = \lambda N - \lambda N_{old} \geq \lambda N - \lambda \frac{N}{2} = \lambda \frac{N}{2} = \frac{n}{2}$

⌚ Pro e contro delle Hash Table

Pro:

- Facile implementazione
- Buone prestazioni (al caso medio)
- Non richiede che le chiavi vengano da un universo ordinato

Contro:

- Complessità elevata al caso pessimo
- Alea dovuta alla bontà della funzione hash

- Spreco di spazio (per mantenere un basso load factor)

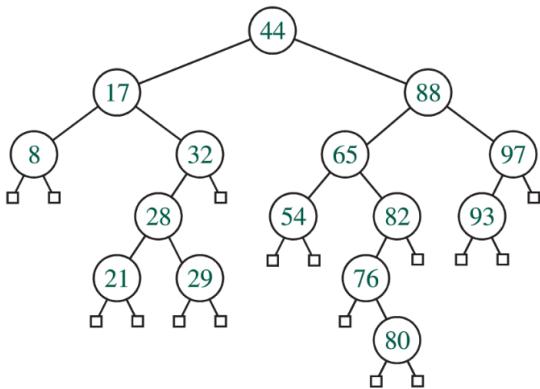
6.5. Alberi binari di ricerca

Un *albero binario di ricerca* è un albero binario proprio i cui *nodi interni* memorizzano entry con chiavi provenienti da un universo ordinato, tale che, per ogni nodo interno v la cui entry ha chiave k , le *chiavi nel sottoalbero sinistro di v* sono **minori di k** , mentre le *chiavi nel sottoalbero destro di v* sono **maggiori di k** .

⌚ Osservazione

I nodi foglia sono "sentinelle" che delimitano i confini dell'albero e possono essere implementati con puntatori `null` nei loro padri.

6.5.1. Esempio (solo chiavi)



⚡ Nota bene

La visita *inorder* tocca le entry in ordine non decrescente.

6.5.2. TreeSearch per un albero binario di ricerca T

Algoritmo TreeSearch(k, v)

Input: chiave k , nodo $v \in T$.

Output: nodo di T_v con chiave k (se esiste) o foglia in posizione "giusta" per k .

```

if (T.isExternal(v) OR (v.getElement().getKey() = k)) then{
    return v; //Caso base
}
    
```

```

}
if(k < v.getElement().getKey()) then{
    return TreeSearch(k, T.left(v));
}
else{
    return TreeSearch(k, T.right(v));
}

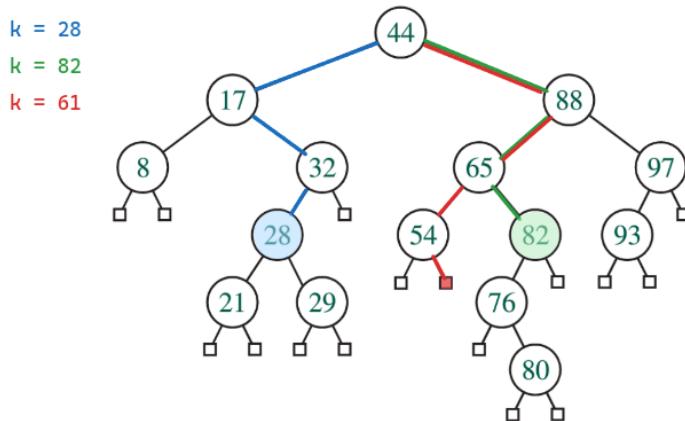
```

! Osservazione

Una foglia "giusta" per k è una foglia che, se trasformata in un nodo interno, può contenere una entry con chiave k senza violare la proprietà di ABR.

La *chiamata iniziale* per ricercare in tutto l'albero ha `v = T.root()`.

6.5.2.1. Esempi



6.5.2.2. Complessità

Sia h l'altezza di T . Analizziamo `TreeSearch(k, T.root())` usando l'albero della ricorsione:

- L'albero della ricorsione ha $\leq h+1$ nodi, dato che ciascuna invocazione ricorsiva di `TreeSearch` scende di un livello in T ;
 - Ogni invocazione esegue $\Theta(1)$ operazioni, oltre a un'eventuale chiamata ricorsiva;
 - Esistono istanze per cui `TreeSearch` scende effettivamente lungo un cammino di lunghezza $\Theta(h)$ (quando restituisce la foglia più profonda).
- ⇒ Complessità $\Theta(h)$

⚡ Nota bene

Con $\Theta(h)$ intendiamo $\Theta(h + 1)$, che copre il caso $h = 0$.

⚠ Attenzione

Senza ipotesi sul grado di bilanciamento di T , h può essere $\Theta(n)$

.

6.5.2.3. Implementazione dei metodi della Mappa

6.5.2.3.1. get

Metodo `get(k)`

```
w <- TreeSearch(k, T.root());
if (T.isExternal(w)) then{
    return null;
}
else{
    return w.getElement().getValue();
}
```

La *complessità* è $\Theta(h)$ perché è dominata dalla `TreeSearch`.

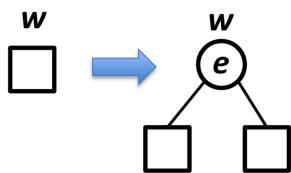
[Lezione 21](#)

6.5.2.3.2. put

Metodo `put(k, x)`

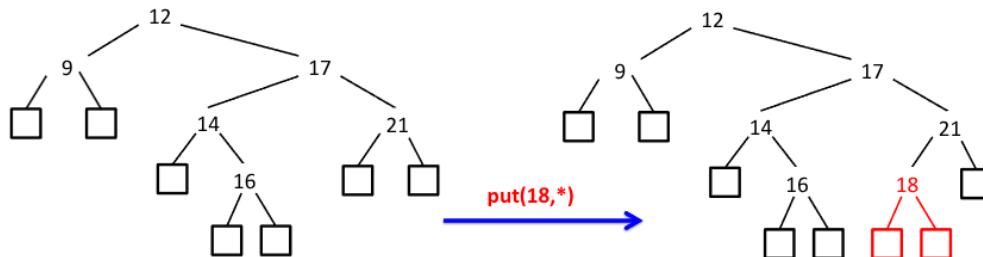
```
w <- TreeSearch(k, T.root());
if (T.isInternal(w)) then {
    y <- w.getElement().getValue();
    sostituisce x a y nella entry w.getElement();
    return y;
}
expandExternal(w, e = (k, x));
incrementa numero di entry di T di 1;
return null;
```

Il metodo `expandExternal(w, e)` trasforma w in nodo interno contenente la entry e :



La *complessità* è $\Theta(h)$ perché è dominata dalla TreeSearch.

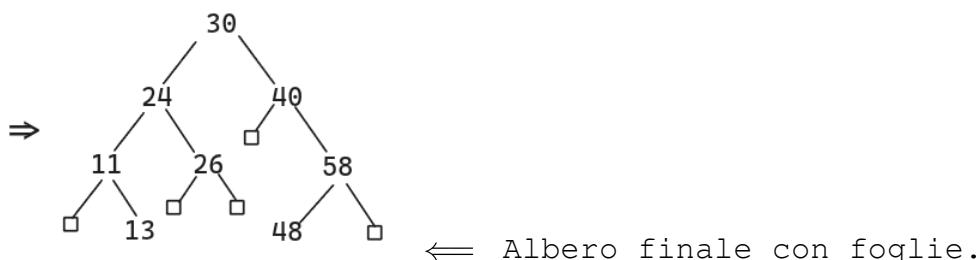
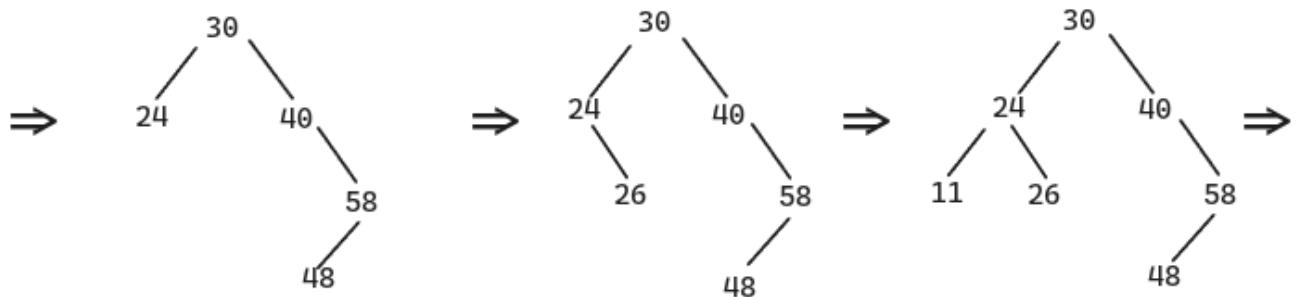
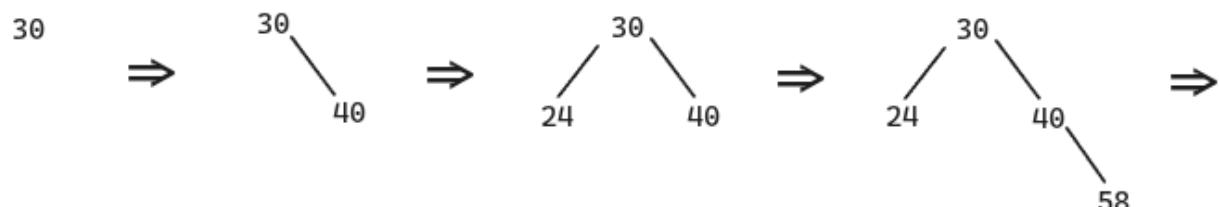
6.5.2.3.2.1. Esempio (solo chiavi)



6.5.2.3.2.2. Esercizio

Inserire un albero di ricerca vuoto otto entry con chiavi 30, 40, 24, 58, 48, 26, 11, 13 e far vedere l'albero risultante dopo ciascun inserimento.

(senza foglie)



6.5.2.3.3. remove

Metodo remove(k)

```
w <- TreeSearch(k, T.root());
if (T.isExternal(w)) then {
    return null;
}
else{
    value <- w.getElement().getValue();
    decrementa il numero di entry in T di 1;
    if((T.isExternal(T.left(w)) OR (T.isExternal(T.right(w)))) then {
        //Caso 1: w interno con almeno 1 figlio foglia
        //funziona correttamente anche se entrambi i figli di w sono
        entrambi foglie
        u_L <- T.left(w);
        u_R <- T.right(w);
        if (T.isExternal(u_L)) then {
            cancella w e u_L;
            fai salire u_R al posto di w;
        }
        else{
            cancella w e u_R;
            fai salire u_L al posto di w;
        }
    }
    else{
        //Caso 2: w interno con due figli interni
        y <- nodo con chiave max nel sottoalbero sinistro di w;
        sposta la entry in y nel nodo w;
        cancella y e il suo figlio destro;
        fai salire il figlio sinistro di y al posto di y;
    }
}
return value
```

6.5.2.3.3.1. Caso 1

In questo caso w ha almeno un figlio foglia.

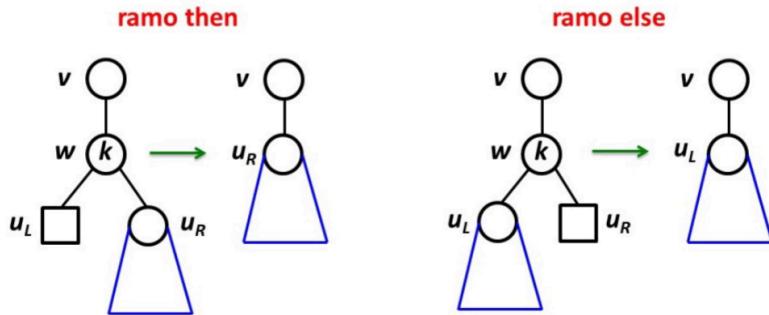
```
u_L <- T.left(w);
u_R <- T.right(w);
if (T.isExternal(u_L)) then {
```

```

cancella w e u_L;
fai salire u_R al posto di w;
}

else{
    cancella w e u_R;
    fai salire u_L al posto di w;
}

```



⚠️ Osservazione

Se w era radice, allora la nuova radice è il figlio messo al suo posto.

💡 Nota bene

Il caso 1 funziona correttamente anche se entrambi i figli di w , u_L e u_R , sono foglie.

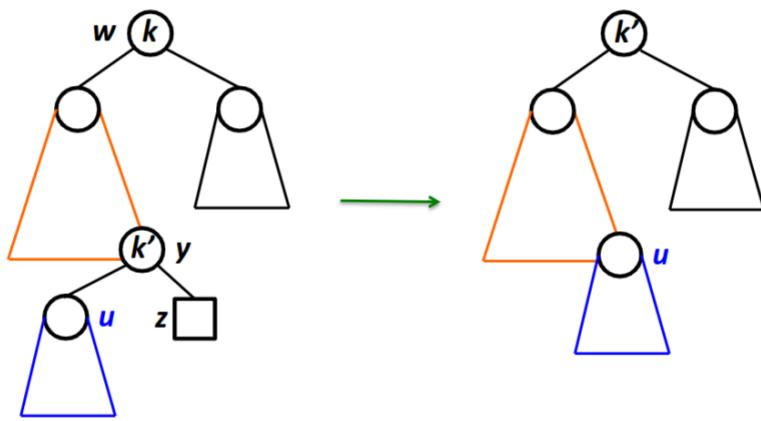
6.5.2.3.3.2. Caso 2

In questo caso w ha due figli interni.

```

y ← nodo con chiave max nel sottoalbero sinistro di w;
sposta la entry in y nel nodo w;
cancella y e il suo figlio destro;
fai salire il figlio sinistro di y al posto di y;

```



La prima istruzione ha $\Theta(h)$ operazioni, mentre le altre tre $\Theta(1)$ operazioni.

! Osservazione

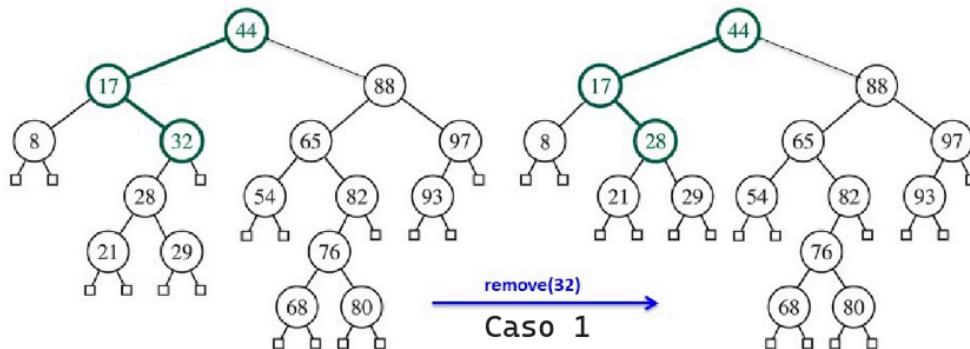
y è il predecessore "interno" di w nella visita inorder e contiene la chiave massima (k') tra quelle nel sottoalbero sinistro di w , cioè quella che precede k nell'ordine crescente delle chiavi.

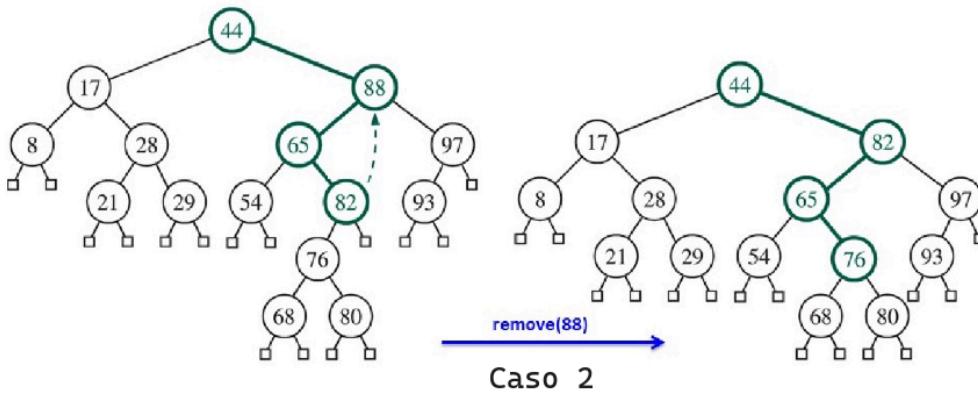
6.5.2.3.3.3. Complessità

Sia h l'altezza di T . La complessità di `remove(k)` è ottenuta sommando i seguenti contributi:

- `TreeSearch` : $\Theta(h)$;
- Nel caso 2, la ricerca di y : $\Theta(h)$;
- Altre operazioni: $\Theta(1)$;
 \Rightarrow Complessità $\in \Theta(h)$.

6.5.2.3.3.4. Esempio





6.5.2.3. TreeSearch iterativo

Algoritmo TreeSearch(T, k)

Input: ABR T e una chiave k ;

Output: nodo $w \in T$ contenente una entry con chiave k , se esiste, o una foglia "giusta" per k .

```
w <- T.root();
while (T.isInternal(w)) do{
    x <- w.getElement().getKey();
    if(x = k) then{
        return w;
    }
    if(k < x) then{
        w <- T.left(w);
    }
    else{
        w <- T.right(w);
    }
}
return w;
```

La complessità è $\Theta(h)$, infatti:

- vengono eseguite $\Theta(1)$ operazioni fuori dal while;
- vengono eseguite $\Theta(h)$ operazioni del while al caso pessimo;
- vengono eseguite $\Theta(1)$ operazione in ciascuna iterazione del while.

6.5.3. Osservazioni sugli Alberi Binari di Ricerca

Una debolezza degli Alberi Binari di Ricerca si vede quando sono molto sbilanciati (ovvero quando $h \in \Theta(n)$ con n il numero di entry): la complessità dei tre metodi `get`, `put` e `remove` degradano sino a diventare quelle ottenibili con una semplice lista non ordinata.

Allora un ABR non ha nessun valore aggiunto, al caso pessimo, rispetto alle liste o alla Tabella Hash.

Alcune direzioni possibili per migliorare gli ABR sono:

1. Ribilanciare T ogni volta che lo sbilanciamento supera una soglia prefissata (es: AVL, Red-Black Tree);
2. Rendere i nodi più capienti in modo da *assorbire* meglio gli effetti di inserimenti o rimozioni che potrebbero portare a uno sbilanciamento dell'albero (es: $(2, 4)$ -Tree).

In seguito vedremo implementazioni efficienti della Mappa Ordinata basate su varianti degli Alberi Binari di Ricerca e discuteremo come generalizzare la Mappa per permettere chiavi duplicate (Multimappa). In particolare tratteremo di Multi-Way Search Tree (MWS-Tree), $(2, 4)$ -Tree, faremo cenni sui Red-Black Tree e parleremo di Multimappe.

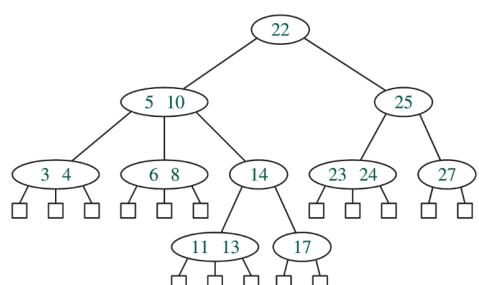
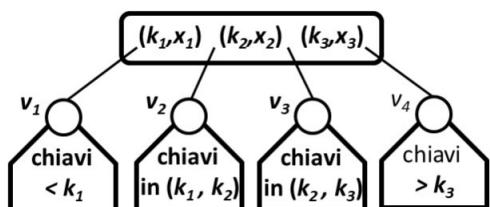
6.6. Multi-Way Search Tree

6.6.1. Definizione, osservazioni, proposizione

Un *Multi-Way Search Tree* (MWS-Tree) T è un albero *ordinato* tale che ogni nodo interno ha *almeno due figli*, ogni nodo interno con $d \geq 2$ figli v_1, v_2, \dots, v_d (*d-node*) soddisfa le seguenti proprietà:

- *Memorizza $d - 1$ entry* $(k_1, x_1), (k_2, x_2), \dots, (k_{d-1}, x_{d-1})$ dove $k_1 < k_2 < \dots < k_{d-1}$;
- Per $1 \leq i \leq d$ vale che la chiave di ogni entry è memorizzata in un nodo T_{v_i} soddisfa la relazione $k_{i-1} < e.getKey() < k_i$ (assumendo $k_0 = -\infty$ e $k_d = +\infty$);

Per convenzione, le foglie sono delle sentinelle e non memorizzano entry (come negli ABR).



! Osservazione

I Multi-Way Search Tree generalizzano gli Alberi Binari di Ricerca permettendo ai nodi di contenere più entry, cosa che rende più agevole il bilanciamento (nella variante $(2, 4)$ -Tree che vedremo in seguito).

! Osservazione

Un Albero Binario di Ricerca è un MWS-Tree in cui ogni nodo è un 2-node.

≡ Proposizione

Un MWS-Tree che memorizza n entry ha $n+1$ foglie.

6.6.1.1. Dimostrazione

Sia T un MWS-Tree che memorizza n entry.

Definiamo:

- $A = \{\text{nodi interni di } T\}$;
- $B = \{\text{foglie di } T\}$
- $\forall v \in T$ sia d_v il numero di figli di v :
 - se v è foglia, allora $d_v = 0$ e v non contiene entry;
 - se v è interno, allora $d_v > 0$ e v contiene $d_v - 1$ entry;

Ricordiamo la proprietà per cui $\sum_{v \in T} d_v = |A| + |B| - 1$ e inoltre vale che $\sum_{v \in T} d_v = \sum_{v \in A} d_v$.

$$\begin{aligned} n &= \sum_{v \in A} (d_v - 1) = (\sum_{v \in A} d_v) - |A| = |A| + |B| - 1 - |A| = |B| - 1 \\ \implies |B| &= n + 1 \end{aligned}$$

□

! Osservazione

Se tutti i nodi interni fossero dei d-node l'MWS-Tree T sarebbe un albero d-ario.

Detto x il numero dei nodi interni e y il numero di foglie T sappiamo che $y = (d - 1)x + 1$, che è coerente con la proposizione

precedente dato che l'albero in questo caso memorizzerebbe $n = (d - 1)x$ entry.

6.6.2. Ricerca in un MWS-Tree

Algoritmo MWTreeSearch(k, v)

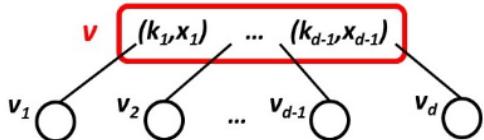
Input: chiave k , nodo $v \in T$.

Output: nodo di T_v contenente una entry con chiave k , se esiste, o foglia in posizione "giusta" per k .

```

if(T.isExternal(v)) then{
    return v;
}
//Siano (k1, x1), ..., (k{d-1}, x{d-1}) le entry in v, con k1 < ... < k{d-1}, e siano v1, v2, ..., vd i figli di v (Vedi immagine sottostante)
Trova i tale che k{i-1} < k <= ki; //si assuma k0 = -∞, kd = +∞
if (k = ki) then{
    return v;
}
else{
    return MWTreeSearch(k, vi);
}

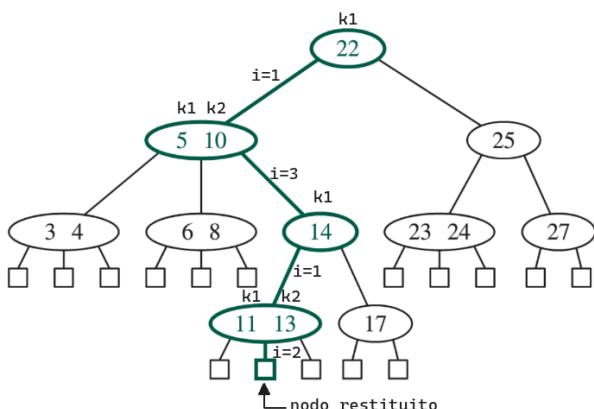
```



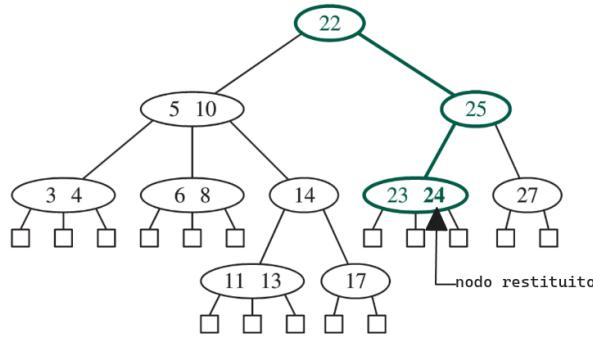
La *chiamata iniziale* per cercare in tutto l'albero corrisponde a $v = T.root()$.

6.6.2.1. Esempi

MWTreeSearch(12, T.root())



```
MWTreeSearch(24, T.root())
```



6.6.2.2. Complessità

Ipotesi: le entry in un nodo interno sono memorizzate in una lista ordinata.

L'albero della ricorsione per `MWTreeSearch(k, T.root())` :

- Ha $\leq h+1$ chiamate ricorsive, dove h è l'altezza di T ;
- Il costo associato a ciascuna chiamata ricorsiva è $\Theta(d_{max})$ al caso pessimo, dove d_{max} è il massimo numero di figli di un nodo
 \implies Complessità $\in \Theta(hd_{max})$.

La complessità non migliora in generale quella della TreeSearch per gli ABR in quanto per i MWS-Tree non abbiamo limiti superiori su altezza e massimo numero di figli, se non (per entrambi) il limite banale di n , dove n è il numero di entry in T .

Lezione 22

6.6.3. Implementazione dei metodi della Mappa: get

Metodo `get(k)`

```
w <- MWTreeSearch(k, T.root());
if(T.isExternal(w)) then{
    return null;
}
else{
    trova e∈w tale che e.getKey() = k;
    return e.getValue();
}
```

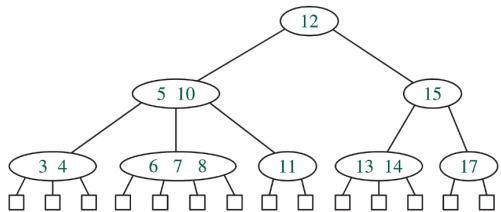
La **complessità** è dominata da quella di `MWTreeSearch`, ed è quindi $O(d_{max}h)$, dove d_{max} è il massimo numero di figli di un nodo e h è l'altezza dell'albero.

6.6.4. (2, 4)-Tree

Un $(2, 4)$ -Tree è un MWS-Tree tale che ogni nodo interno è un d -node con $2 \leq d \leq 4$ (quindi ha d figli e $d - 1$ entry) e tale che tutte le foglie hanno la stessa profondità.

! Osservazione

Si può definire un $(2, 3)$ -Tree con $2 \leq d \leq 3$ per il quale si applicano gli stessi algoritmi e le stesse complessità. Il vantaggio del $(2, 4)$ -Tree è che si generalizza al B-Tree, che è una struttura dati molti usata per la realizzazione di indici in memoria secondaria (ad esempio nelle basi di dati).



6.6.4.1. Altezza

☰ Proposizione

Un $(2, 4)$ -Tree con $n > 0$ entry ha altezza $\Theta(\log n)$.

Il seguente corollario è una conseguenza immediata della proposizione e di quanto visto prima.

☰ Corollario

In un $(2, 4)$ -Tree con n entry, la complessità di `MWTreeSearch` e del metodo `get` della mappa sono $\Theta(\log n)$

6.6.4.1.1. Dimostrazione

Sia T un $(2, 4)$ -Tree con n entry e altezza h , sia m_i il numero di nodi al livello i , per $0 \leq i \leq h$. Si ha che: $m_0 = 1$, $2 \leq m_1 \leq 4$, $2^2 \leq m_2 \leq 4^2$, ..., $2^i \leq m_i \leq 4^i$, ..., $2^h \leq m_h \leq 4^h$.

- So che, per definizione, le foglie sono tutti i nodi del livello h ;

- Poiché il $(2, 4)$ -Tree è un MWS-Tree, so che il numero di foglie è $n + 1$.

$$\begin{aligned} \implies 2^h \leq m_h = n + 1 \leq 4^h &\implies h \leq \log_2(n + 1) \leq \log_2(4^h) = 2h \implies h \leq \log_2(n + 1) \\ \text{e } h \geq \frac{\log_2(n+1)}{2} & \\ \implies h \in \Theta(\log n) & \end{aligned}$$

Questo conclude la prova della Proposizione.

Il corollario segue immediatamente dall'analisi fatta per il MWS-Tree e dal fatto che per il $(2, 4)$ -Tree vale che $h \in \Theta(\log n)$ e $d_{max} \leq 4$.

6.6.5. Implementazione dei metodi della Mappa: put e remove

6.6.5.1. put

L'idea per realizzare il metodo `put(k, x)` consiste nei seguenti punti:

- Se la chiave non è presente, inserisci la nuova entry $e = (k, x)$ in un nodo giusto per k ad altezza 1.
- Se il nodo in cui è stata inserita e va in *overflow* (ovvero ha quattro entry e cinque figli), invoca il metodo *split*, che *ripristina le proprietà* del $(2, 4)$ -Tree sfruttando la flessibilità sul numero di entry ammissibili in un nodo e *propagando*, se necessario, *l'overflow verso l'alto* che, se arriva alla radice, fa crescere di uno l'altezza dell'albero.

Metodo `put(k, x)`

```
w <- MWTreeSearch(k, T.root());
if (T.isInternal(w)) then {
    e <- entry in w con chiave k;
    y <- e.getValue();
    sostituisci x a y in e;
    return y;
}
//Caso in cui w è foglia
e <- (k, x);
if (T.isRoot(w)) then{
    expandExternal(w, e);
}
else{
    u <- T.parent(w);
    inserisci e in u aggiungendo una foglia w';
```

```

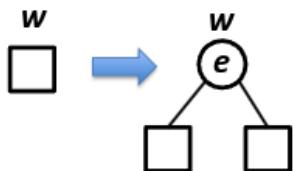
if (u è un 5-node) then { //overflow
    Split(u);
}
incrementa il numero di entry in T di 1;
return null;

```

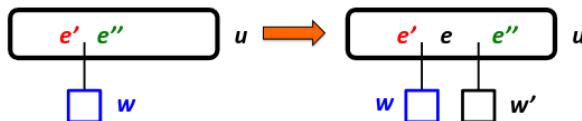
! Osservazione

Se w è interno, vengono effettuate $\Theta(1)$ operazioni.

Nel caso in cui w è sia foglia che radice viene chiamato il metodo `expandExternal(w,e)`:

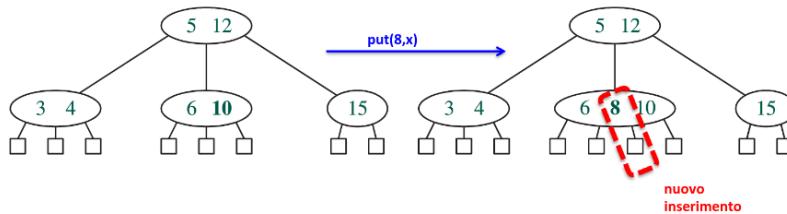


Nel caso in cui w è foglia ma non radice si inserisce e in u , aggiungendo una foglia w' :



⚡ Nota bene

e' oppure e'' potrebbe non esistere.



6.6.5.1.1. Metodo Split (ricorsivo)

Metodo `Split(u)`

Input: 5-node $u \in T$, unica violazione delle proprietà di $(2, 4)$ -Tree.

Output: Ripristino delle proprietà di $(2, 4)$ -Tree.

Sia $u = (e_1, e_2, e_3, e_4)$ con figli u_1, u_2, u_3, u_4 .

Crea due nuovi nodi $u' = (e1, e2)$ con figli u_1, u_2, u_3 e $u'' = (e4)$ con figli u_4, u_5 ;

```

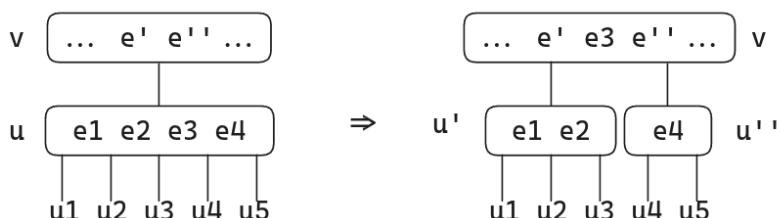
if (!T.isRoot(u)) then{
    v <- T.parent(u);
    inserisci e3 in v con figlio sinistro u' e figlio destro u'';
    cancella u;
    if(v è un 5-node) then {
        Split(v);
    }
}
else{
    crea una nuova radice contenente e3 e con due figli u', u';
    cancella u;
}

```

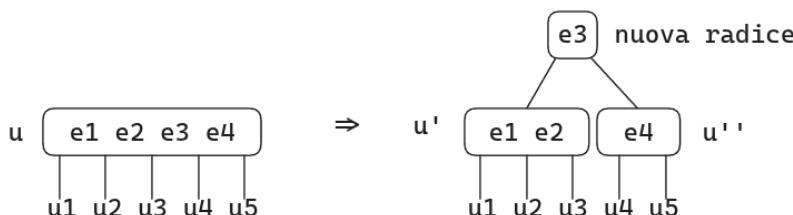
① Osservazione

Il primo ramo `if` del metodo ha $\Theta(1)$ operazioni, esclusa l'eventuale chiamata ricorsiva, mentre il ramo `else` fa aumentare l'altezza di T di uno.

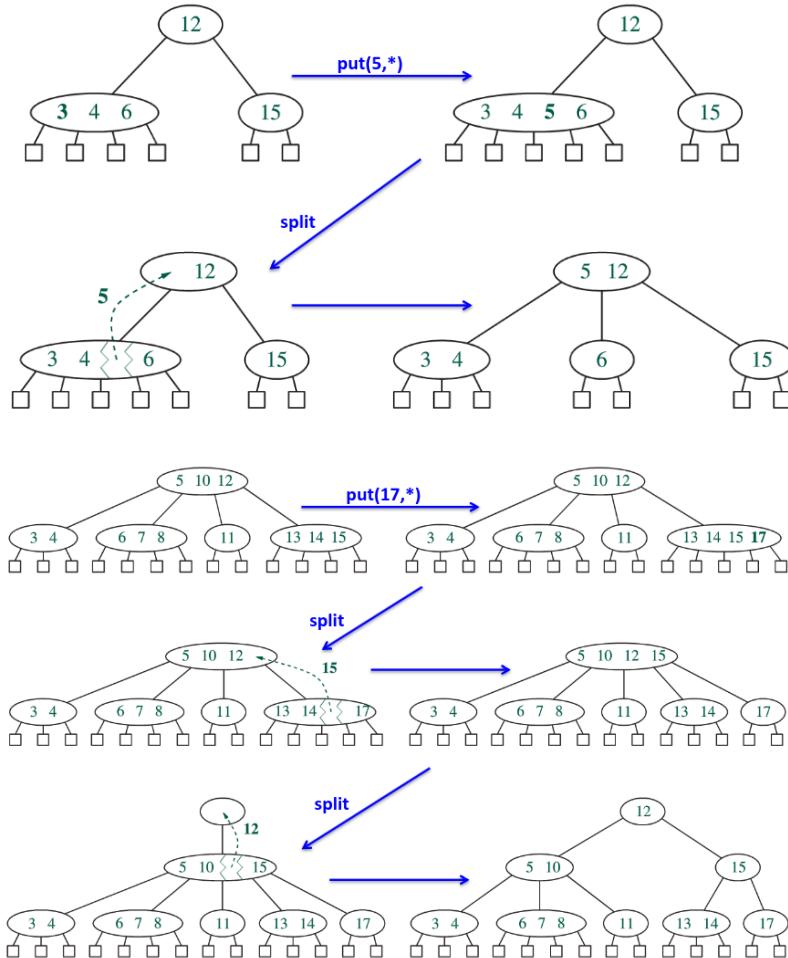
Graficamente, il caso in cui u non è radice si rappresenta così:



Mentre il caso in cui u è radice si rappresenta così:



6.6.5.1.2. Esempi con overflow



6.6.5.1.3. Complessità

Proposizione

Per una mappa con n entry implementata tramite un $(2,4)$ -Tree, la complessità di `put` è $\Theta(\log n)$.

La complessità è dominata da `MWTreeSearch` e da `Split`.

- `MWTreeSearch` effettua $\Theta(\log n)$ operazioni;
- `Split` è un algoritmo ricorsivo.
 - La prima invocazione avviene su un nodo ad altezza 1;
 - Le successive invocazioni, una per livello sino ad arrivare eventualmente alla radice, eseguono $\Theta(1)$ operazioni per invocazione
 \implies in totale per lo `Split` vengono effettuate $\Theta(\log n)$ operazioni.
 - \implies La complessità finale è $\Theta(\log n)$.

6.6.5.2. `remove`

L'idea per realizzare il metodo `remove(k)` consiste nei seguenti punti:

- Si rimuovono sono entry in nodi ad altezza 1;
- Se la entry e da rimuovere è in un nodo ad altezza maggiore di 1, si sostituisce con una entry e' ad altezza 1 e si rimuove quella;
- Se il nodo ad altezza 1 da cui è stata rimossa una entry va in *underflow* (ovvero contiene zero entry), *ripristina le proprietà* del (2,4)-Tree sfruttando la flessibilità sul numero di entry per nodo *propagando*, se necessario, *l'underflow verso l'alto* che, se arriva alla radice, fa diminuire di uno l'altezza dell'albero.

⚡ Nota bene

La rimozione di una entry da un nodo e la gestione dell'eventuale underflow sarà effettuata tramite il metodo `Delete`.

Lezione 23

Metodo `remove(k)`

```
w <- MWTreeSearch(k, T.root());
if (T.isExternal(w)) then {
    return null;
}
else{
    trova e∈w tale che e.getKey() = k;
    y <- e.getValue();
    if(altezza(w) = 1) then{ //se e solo se ha foglie
        Delete(e,w);
    }
    else{
        v <- figlio di w a sinistra di e;
        e` <- entry con chiave max in T_v; //O(log n) operazioni
        z <- nodo contenente e`; //z è ad altezza 1
        metti una copia di e` al posto di e in w;
        Delete(e`,z);
    }
}
```

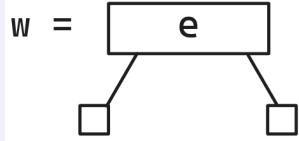
```

decrementa di 1 il numero di entry in T;
return y;

```

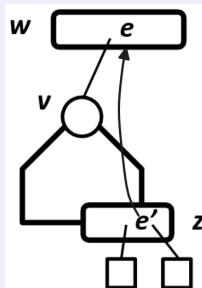
! Osservazione

Nel caso in cui l'altezza di w sia uno, si rimuove e e la foglia alla sua destra. Se w va in underflow, cioè se e era l'unica entry in w , la Delete ripristina le proprietà del $(2, 4)$ -Tree.



! Osservazione

Nel caso in cui l'altezza di w sia maggiore di uno, si sostituisce e' al posto di e in w e si rimuove e' da z insieme al suo figlio destro. Se dopo questa rimozione z va in underflow, la Delete ripristina le proprietà del $(2, 4)$ -Tree.



6.6.5.2.1. Metodo Delete (ricorsivo)

Metodo Delete(e, u)

Input: $u \in T$ con entry e , con un figlio foglia o vuoto a sinistra o destra di e .

Output: rimozione di e da T ripristinando le proprietà di $(2, 4)$ -Tree.

Siano A, X i figli di u discriminati da e dove X è foglia o vuoto.



```

rimuovi e, X;
if (u non ha più entry) then{
    Caso 1: u radice;
}

```

Casi 2 e 3: u con un fratello (sinistro o destro) d-node, d = 3, 4

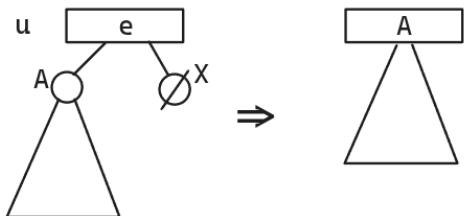
Casi 4 e 5: u con solo fratelli 2-node;

//Le operazioni da eseguire sono elencate in seguito

}

6.6.5.2.1.1. Caso 1: u radice

Operazione: imposta A come nuova radice del (2, 4)-Tree;

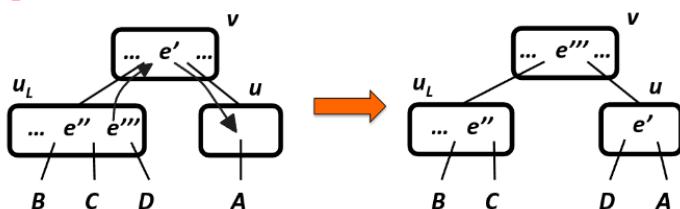


! Osservazione

Questo è il caso in cui l'altezza dell'albero diminuisce di uno.

6.6.5.2.1.2. Caso 2: u ha un fratello u_L a sinistra che è un d-node, con $d \geq 3$

Operazione:

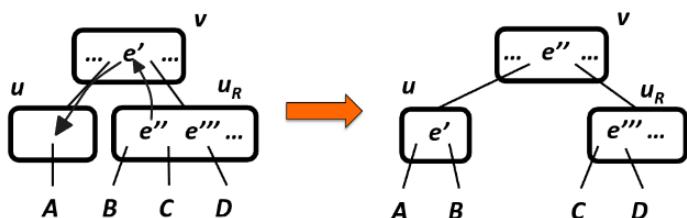


! Osservazione

Con questa rotazione la Delete termina la sua esecuzione.

6.6.5.2.1.3. Caso 3: u ha un fratello u_R a destra che è un d-node, con $d \geq 3$

Operazione:

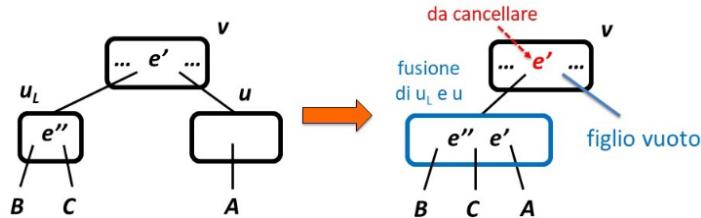


! Osservazione

Con questa rotazione la `Delete` termina la sua esecuzione.

6.6.5.2.1.4. Caso 4: u ha un fratello u_L a sinistra che è un 2-node

Operazione:



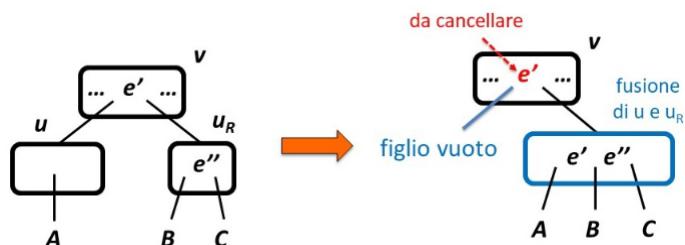
`Delete(e', v);`

! Osservazione

Viene propagato l'underflow verso l'alto se la rimozione di e' da v genera underflow, cioè se e' era l'unica entry in v .

6.6.5.2.1.5. Caso 5: u ha un fratello u_R a destra che è un 2-node

Operazioni:



`Delete(e', v);`

! Osservazione

Viene propagato l'underflow verso l'alto se la rimozione di e' da v genera underflow, cioè se e' era l'unica entry in v .

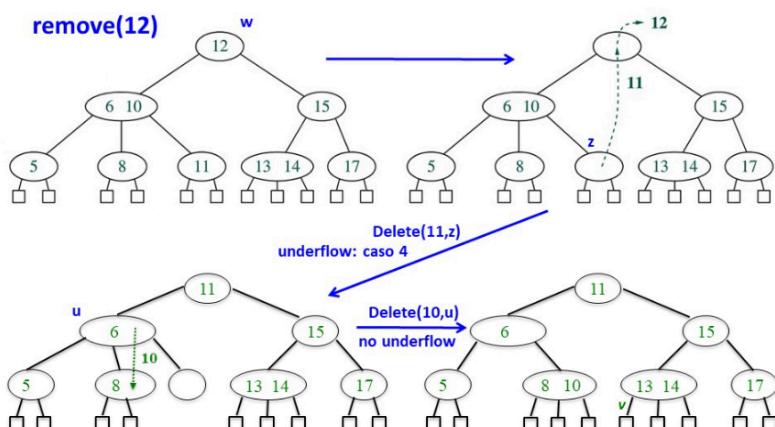
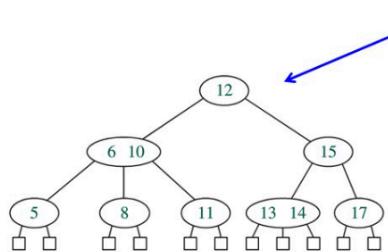
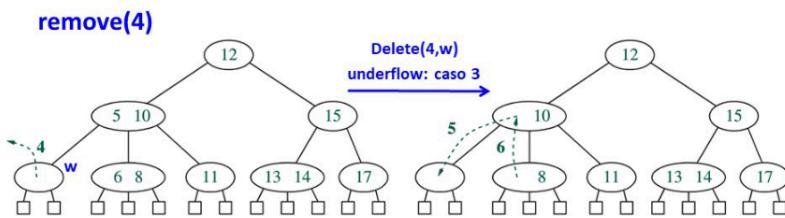
! Osservazione

I cinque casi possono essere esaminati nell'ordine 1-2-3-4-5 sino a trovare il primo che può essere applicato (uno sicuramente sarà trovato perché i cinque casi coprono tutti i possibili scenari).

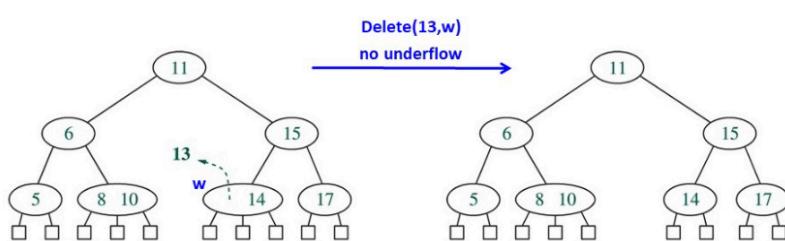
! Osservazione

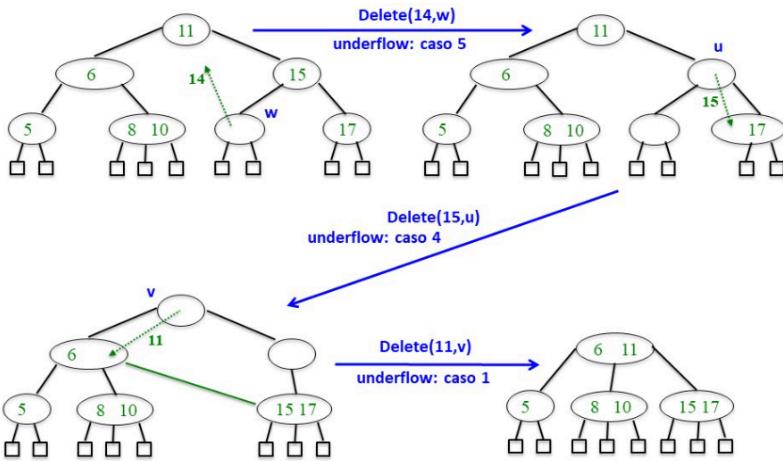
È facile vedere che in tutti e cinque i casi, le operazioni fatte mantengono valide le proprietà del $(2, 4)$ -Tree.

6.6.5.2.2. Esempi



remove(13)





6.6.5.2.3. Complessità

Proposizione

Per una mappa con n entry implementata tramite un $(2,4)$ -Tree, la complessità di `remove` è $\Theta(\log n)$.

La complessità è dominata da:

- `MWTreeSearch` : $\Theta(\log n)$ operazioni;
 - Eventuale ricerca della entry e' con altezza 1 da sostituire al posto di e , se e è ad altezza maggiore di uno: $\Theta(\log n)$ operazioni;
 - `Delete` : algoritmo ricorsivo:
 - $\leq h$ invocazioni ricorsive perché parte da un nodo ad altezza uno e verrà invocata lungo il percorso da questo nodo alla radice, al più una volta per livello;
 - Ogni invocazione ricorsiva richiede $\Theta(1)$ operazioni;
 \Rightarrow Complessità di `Delete` è proporzionale all'altezza h del $(2,4)$ -Tree che abbiamo dimostrato essere $\Theta(\log n)$;
 \Rightarrow La complessità di `remove(k)` è $\Theta(\log n)$.
-

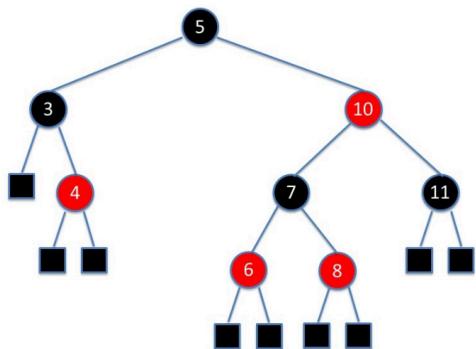
6.7. Red-Black Tree

I Red-Black Tree sono Alberi Binari di Ricerca che, a differenza del caso generale, hanno altezza sempre logaritmica nel numero di nodi.

Un *Red-Black Tree* (RB-Tree) T è un ABR i cui nodi hanno un colore rosso o nero e in cui valgono le seguenti proprietà:

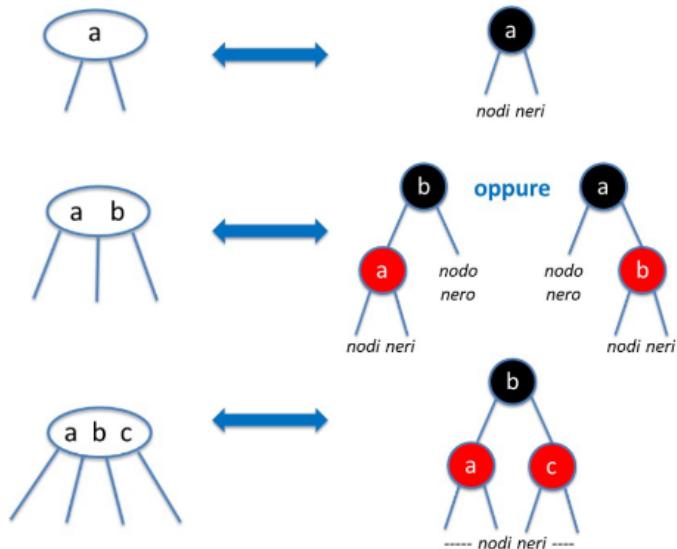
- La radice è nera (*Root Property*);
- Le foglie sono nere (*External Property*);
- I figli di un nodo rosso sono neri (*Red Property*);

- Tutte le foglie hanno la stessa *black depth*, ovvero lo stesso numero di antenati propri neri (*Depth Property*).

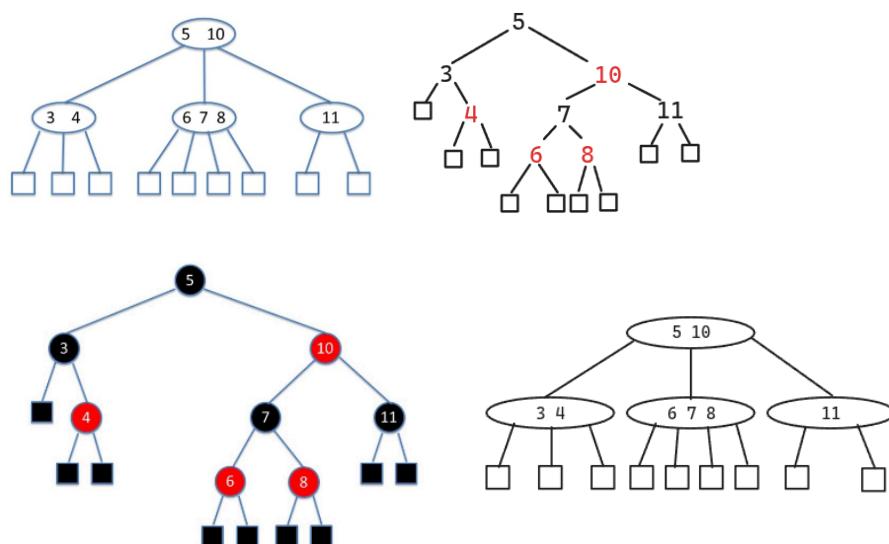


6.7.1. $(2, 4)$ -Tree e Red-Black Tree

È possibile trasformare un $(2, 4)$ -Tree in un Red-Black Tree e viceversa, applicando le seguenti trasformazioni dalla radice verso le foglie:



6.7.1.1. Esempi



6.7.2. Metodi della Mappa su Red-Black Tree

Valgono le seguenti proprietà:

- Un Red-Black Tree contenente n entry ha altezza $\Theta(\log n)$. È una diretta conseguenza delle trasformazioni: passando da un $(2, 4)$ -Tree a un Red-Black Tree l'altezza al più raddoppia, mentre nel caso opposto al più si dimezza.
- I metodi `get`, `put` e `remove` della Mappa possono essere implementati in un Red-Black Tree con complessità $\Theta(\log n)$ al caso pessimo.

! Osservazione

L'implementazione dei metodi della mappa risulta efficiente in pratica a parte la `TreeSearch`, per tutti e tre i metodi le altre operazioni sono in numero costante.

In Java la classe `TreeMap<k, V>` implementa una Mappa tramite Red-Black Tree.

6.8. Multimappa

Una *Multimappa* è una Mappa che ammette la presenza di più entry con la stessa chiave. Questa differenza richiede una modifica della specifica dei metodi caratterizzanti.

6.8.1. Metodi caratterizzanti

- `get(k)` : restituisce una collezione (eventualmente vuota) con tutti i valori associati alle entry con chiave k ;
- `put(k, v)` : inserisce *sempre* una nuova entry (k, v) senza intaccare le altre entry con chiave k già presenti. Non restituisce alcun output;
- `remove(k, v)` : rimuove una entry con chiave k e valore v , se tale entry esiste. Restituisce un boolean: `true` se si è rimossa la entry, `false` altrimenti.

! Osservazione

Se esistono più entry (k, v) si può scegliere se rimuoverne una arbitraria o tutte.

Una Multimappa può essere implementata tramite una Mappa in cui le entry sono costituite da coppie (k, L_k) , dove k è una chiave e L_k una lista, non vuota, di valori associati alla chiave.

Se $L_k = \{v_1, v_2, \dots, v_l\}$, allora (k, L_k) rappresenta, in modo compatto, le l entry $(k, v_1), (k, v_2), \dots, (k, v_l)$.

6.8.1.1. Implementazione

- `get(k)` : se esiste una entry (k, L_k) restituisce L_k , altrimenti restituisce una collezione vuota;
- `put(k, v)` : se esiste una entry (k, L_k) si aggiunge semplicemente il valore v a L_k , altrimenti si aggiunge la entry $(k, L_k = \{v\})$ alla struttura;
- `remove(k, v)` :
 - Se esiste una entry (k, L_k) e $L_k = \{v\}$, si rimuove la entry (k, L_k) e si restituisce `true`;
 - Se esiste una entry (k, L_k) e L_k contiene v e altri valori, si rimuove v da L_k e si restituisce `true`;
 - In tutti gli altri casi si restituisce `false`, senza modificare la Multimappa.

⚠️ Osservazione

Nel caso esistano più coppie della entry (k, v) da rimuovere, si può decidere di rimuovere una sola o tutte.

Lezione 24

6.8.2. Esempio

Consideriamo una graduatoria come esempio di Multimappa.

Una entry è strutturata così: `(cognome, punteggio)`.

Si ha una collezione delle seguenti entry: `(Bianchi, 75)`, `(Bianchi, 73)`, `(Bianchi, 60)`, `(Rossi, 15)`, `(Rossi, 96)`, `(Verdi, 59)`.

La Multimappa risultante è: `(Bianchi, 75-73-60)`, `(Rossi, 15-96)`, `(Verdi, 59)`.

Dall'esecuzione di `get(Bianchi)`, `put(Rossi, 30)` e `remove(Bianchi, 75)` risulta:

Operazione	Output
<code>get(Bianchi)</code>	75-73-60

Operazione	Output
put(Rossi, 30)	Non restituisce output. (Rossi, 15-96) diventa (Rossi, 15-96-30).
remove(Bianchi, 75)	TRUE. La entry (Bianchi, 75-73-60) diventa (Bianchi, 73-60).

💡 Indici primari e secondari

Nelle Basi di Dati, l'accesso ai dati è reso efficiente dall'utilizzo di *indici primari* e *secondari*.

Un *Indice primario* è (una struttura di accesso a) una collezione di entry basata su una chiave che *non ammette duplicati*. Viene *realizzato tramite una Mappa*.

Un esempio è il numero di matricola degli studenti.

Un *indice secondario* è (una struttura di accesso a) una collezione di entry basata su una chiave che *ammette duplicati*. Viene *realizzato tramite Multimappa*.

Ad esempio il cognome degli studenti.

6.8.2. Complessità dei metodi

Ipotesi: L_k implementata tramite lista.

Definiamo:

- n come numero di *chiavi distinte* presenti nella Multimappa (quindi le entry potrebbero essere molte di più);
- $s = \max_k |L_k|$, il massimo è su tutte le chiavi k presenti nella Multimappa;
- `get(k)` : Stessa complessità della Mappa, usando però la nuova definizione di n ;
- `put(k, v)` : Stessa complessità della Mappa, usando però la nuova definizione di n e assumendo che v sia inserito in testa o in coda a L_k ;
- `remove(k, v)` : alla complessità di `remove(k)` della Mappa (con la nuova definizione di n) si aggiunge un termine additivo $\Theta(s)$ per la ricerca di v in L_k .

💡 Riepilogo complessità per la Mappa

Considerando una Mappa con n entry avrò:

Metodo	Tabella Hash	ABR	(2, 4)-Tree e RB-Tree
get(k)	$\Theta(1 + \lambda)$	$\Theta(h)$	$\Theta(\log n)$
put(k, v)	$\Theta(1 + \lambda)$	$\Theta(h)$	$\Theta(\log n)$
remove(k)	$\Theta(1 + \lambda)$	$\Theta(h)$	$\Theta(\log n)$

💡 Nota bene

- Per l'ABR h è l'altezza dell'albero, che può assumere valori compresi tra $\Theta(\log n)$ e $\Theta(n)$;
- Le complessità per la Tabella Hash sono al caso medio e λ è il load factor.

📋 Riepilogo complessità per la Multimappa

Considerando una Multimappa con n chiavi distinte avrò:

Metodo	Tabella Hash	ABR	(2, 4)-Tree e RB-Tree
get(k)	$\Theta(1 + \lambda)$	$\Theta(h)$	$\Theta(\log n)$
put(k, v)	$\Theta(1 + \lambda)$	$\Theta(h)$	$\Theta(\log n)$
remove(k)	$\Theta(s + 1 + \lambda)$	$\Theta(s + h)$	$\Theta(s + \log n)$

💡 Nota bene

- Per l'ABR h è l'altezza dell'albero, che può assumere valori compresi tra $\Theta(\log n)$ e $\Theta(n)$.
- Il termine s nelle complessità di remove indica il massimo numero di entry con la stessa chiave.
- Nella complessità per la Tabella Hash, il termine λ (dove λ è il load factor) si riferisce al tempo medio di ricerca della chiave, nel caso di remove, il termine s per la ricerca della entry è worst-case.

📋 Riepilogo Mappe

- Mappa: definizione come ADT;

- Tabelle Hash:
 - Ingredienti principali;
 - Funzione hash: hashCode e compression function;
 - Risoluzione delle collisioni tramite chaining;
 - Implementazione dei metodi della Mappa e loro complessità al caso medio sotto l'ipotesi di uniform hashing;
 - Load factor e rehashing;
- Alberi Binari di Ricerca:
 - Definizione;
 - Metodo TreeSearch;
 - Implementazione dei metodi della Mappa;
- (2, 4)-Tree:
 - Definizione di Multi-Way Search (MWS) Tree;
 - Relazione tra numero di entry e foglie in un MWS-Tree;
 - Metodo MWTreeSearch;
 - Definizione di un (2, 4)-Tree;
 - Altezza di un (2, 4)-Tree;
 - Implementazione dei metodi della Mappa;
- Cenni sui Red-Black Tree;
- Implementazione di una Multimappa.

6.9. Esercizi

6.9.1. Mappe pt 2

6.9.1.1. Slide 83

Progettare e realizzare un algoritmo iterativo efficiente che determini l'altezza di un (2, 4)-Tree.

Algoritmo 24Height(T)

Input: (2, 4)-Tree T.

Output: Altezza di T.

Idea: Sfruttare il fatto che tutte le foglie sono alla stessa profondità, scendendo lungo un qualsiasi cammino radice-foglia tenendo traccia del numero di nodi incontrati.

```
h <- 0;
v <- T.root();
```

```

while(T.isInternal(v)) do {
    v <- figlio più a sx di v;
    h <- h + 1;
}
return h

```

Correttezza: Banale.

Complessità:

- h_T iterazioni del while ($h_T \equiv$ altezza di T)
- $\Theta(1)$ operazioni per iterazione
 \implies Complessità $\in \Theta(h_T) = \Theta(\log n)$, con n il numero di entry in T

6.9.2. Mappe pt 2

6.9.2.1. Slide 73

Sia T un albero binario di ricerca dove ogni nodo $v \in T$ memorizza una variabile `v.size` il numero di entry in T_v (inclusa quella in v). Progettare un algoritmo ricorsivo che conti quante entry in T hanno chiave $\leq k$ e analizzarne la complessità.

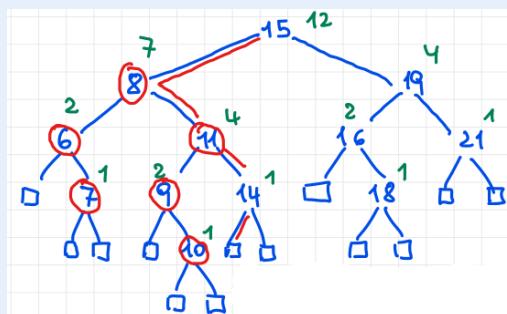
Algoritmo: CountLE(k, v)

Input: chiave k , nodo $v \in T$

Output: numero di entry in T_v con chiave $\leq k$

Prima invocazione: `v = T.root()`

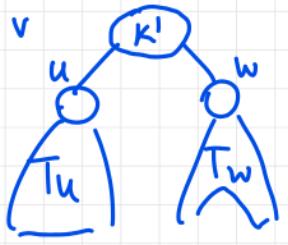
Esempio



(I verdi le variabili size)

Per $k = 13$ l'output è: 6

Idea:



- Se $k' > k \Rightarrow$ restituisco $\text{CountLE}(k, u)$ dato che né k' né le chiavi in T_w possono contribuire all'output
- Se $k' \leq k$ restituisco $1 + u.size + \text{CountLE}(k, w)$ dato che certamente sia k' che tutte le chiavi in T_u contribuiscono all'output

```
if(T.isExternal(v)) then {
    return 0; //CASO BASE
}
if(v.getElement().getKey() > k) then {
    return CountLE(k, T.left(v));
}
else {
    return 1 + T.left(v).size() + CountLE(k, T.right(v));
}
```

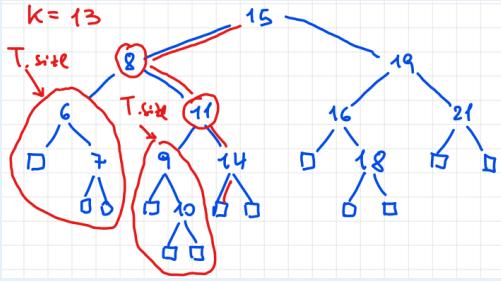
Complessità: Stessa struttura di TreeSearch

- $\leq h$ invocazioni ricorsive
- $\Theta(1)$ operazioni in ciascuna invocazione ricorsiva (tranne in invocazioni ricorsive al suo interno)
 \Rightarrow Complessità $\in \Theta(h)$, con h l'altezza di T

6.9.2.2. Slide 81

Analizzare la complessità dell'algoritmo appena sviluppato, assumendo che al posto della variabile `v.size` si abbia a disposizione un metodo `T.size(v)` che restituisce il numero di entry in T_v , con complessità lineare nel valore restituito.
(*Suggerimento:* esprimere la complessità come somma di due termini, uno dei quali è il costo aggregato di tutte le invocazioni del metodo `size`).

Esempio



```

if (T.isExternal(v)) then {
    return 0; //CASO BASE
}
if (v.getElement().getKey() > k) then {
    return CountLE(k, T.left(v));
}
else {
    return 1 + T.size(T.left(v)) + CountLE(k, T.right(v));
}

```

Complessità di `CountLE(k, T.root())`

Chiamo X il numero di operazioni eseguite, escludendo quelle relative a `T.size`, e chiamo Y il numero aggregato di operazioni di tutte le chiamate a `T.size`.

$$\Rightarrow \text{Complessità} \in \Theta(X + Y)$$

So già che $X \in \Theta(h)$ dall'analisi dell'algoritmo che usa le variabili `v.size`.

Sia m il valore finale restituito da `CountLE(k, T.root())`; sia m_u il valore restituito da `T.size(u)`; sia $U = \{u : \text{l'algoritmo invoca } T.size(u)\}$, nell'esempio $U = \{6, 9\}$.

Si ha che $m \geq \sum_{u \in U} m_u$ e al caso pessimo $\sum_{u \in U} m_u \in \Theta(m)$.

Ed è facile vedere che $Y \in \Theta(\sum_{u \in U} m_u) \Rightarrow Y \in O(m)$.

$$\Rightarrow \text{Complessità di } \text{CountLE}(k, \text{T.root}) \text{ è } \Theta(\underbrace{h}_X + \underbrace{m}_Y).$$

💡 Suggerimento

Sostituire `T.size(T.left(v))` con `CountLE(k, T.left(v))`

Il caso aggregato di tutte le invocazioni di `CountLE` fatta in sostituzione di `T.size` sarà $\Theta(m)$ (con m valore finale restituito)

$$\Rightarrow \text{Complessità} \in \Theta(h + m)$$

6.9.2.3. Slide 80, secondo esercizio

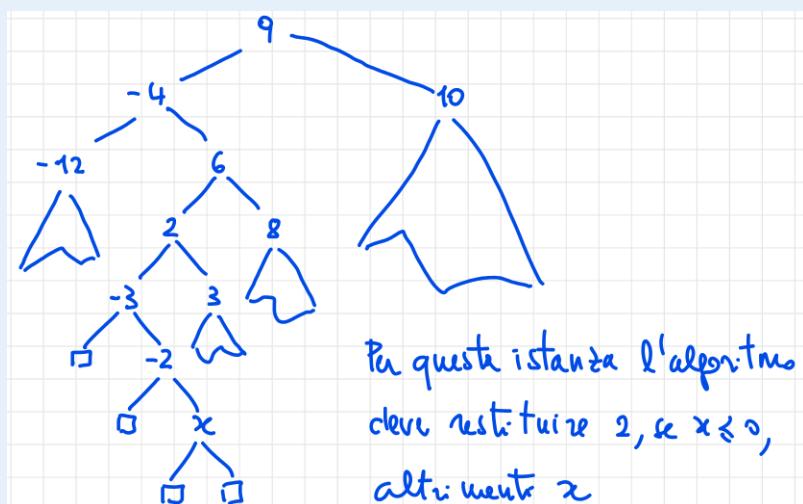
Progettare un algoritmo iterativo che, dato un albero binario di ricerca T contenente entry con chiavi reali distinte, determina la più piccola chiave positiva presente in T , e analizzarne la complessità. Se in T non esistono chiavi positive, l'algoritmo deve restituire 'no positive key'.

Algoritmo: MinPositive(T)

Input: ABR T con chiavi reali distinte.

Output: Minima chiave positiva (se esiste) o "no positive key" altrimenti.

Esempio



Idea:

- Si parte dalla radice;
- Per ogni nodo esaminato v con chiave k
 - Se $k > 0$: salva k come candidato e vai sul figlio sinistro;
 - Se $k \leq 0$: vai sul figlio destro;

```

v <- T.root();
minPk <- +∞;
while (T.isInternal(v)) do{
    k <- v.getElement().getKey();
    if (k > 0) then {
        minPk <- k;
        v <- T.left(v);
    }
    else{

```

```

        v <- T.right(v);
    }
}

if(minPk < +∞) then{
    return minPk;
}
else {
    return "no positive key";
}

```

Complessità: $\Theta(h)$, dove h è l'altezza di T .

È dominata dal while:

- $\leq h$ iterazioni;
- $\Theta(1)$ operazioni per iterazione.

[[Maps2526-p1.pdf#page=82 | Lezione 25]

6.9.3. File Esercizi Mappe

6.9.3.1. Slide 18

Sia T un albero binario di ricerca le cui entry rappresentano studenti di un'università. Ogni studente è associato a una entry (k, x) , dove k è la matricola, e x indica se lo studente è straniero ($x = 1$) o italiano ($x = 0$). Per ogni nodo $v \in T$ esiste un intero $v.\text{numStr}$ che riporta il numero di studenti stranieri in T_v (sottoalbero con radice v). Si vuole progettare un algoritmo ricorsivo `MinMatStraniero` per determinare la più piccola matricola di uno studente straniero in T . Se non ci sono studenti stranieri in T l'algoritmo restituisce `null`.

1. Descrivere tramite pseudocodice la generica invocazione di `MinMatStraniero`, specificandone con attenzione l'input e l'output;
2. Analizzare la complessità di `MinMatStraniero`

Algoritmo `MinMatStraniero(T, v)`

Input: ABR T e $v \in T$.

Output: minima matricola di uno straniero in T_v , se ne esiste almeno uno, o `null` altrimenti.

Prima invocazione: $v = T.\text{root}()$

Idea:

- se v è foglia o $v.\text{numStr} = 0 \implies \text{null}$;
- se v è interno:
 - Se u (figlio sinistro di v) ha $u.\text{numStr} > 0 \implies \text{return MinMatStraniero}(T, u)$;
 - Se $u.\text{numStr} = 0$ e la entry in v ha flag 1 $\implies \text{return chiave di } v$;
 - Altrimenti $\text{return MinMatStraniero}(T, w)$ (w figlio destro di v).

```

if(T.isExternal(v) OR v.numStr = 0) then {
    return null;
}
m <- T.left(v).numStr;
x <- v.getElement().getValue()
if(m > 0) then return MinMatStraniero(T, T.left(v))
else{
    if(x = 1) then return v.getElement().getKey()
    else return MinMatStraniero(T, T.right(v))
}

```

Complessità: $\Theta(h)$ con h altezza di T . Stessa analisi di TreeSearch (viene fatta al massimo una chiamata ricorsiva per nodo, quindi al massimo proporzionale all'altezza).

Fare una versione iterativa dell'algoritmo per esercizio.

6.9.3.2. Slide 23

Sia D un documento di N parole, rappresentato da un array $D[0], D[1], \dots, D[N - 1]$.

1. Progettare un algoritmo che, usando una Mappa d'appoggio, restituisca la parola con il massimo numero di occorrenze in D . Se ne esistono più di una, l'algoritmo ne restituisce una arbitraria tra esse.
2. Analizzare la complessità dell'algoritmo scegliendo una opportuna implementazione per la Mappa.

Algoritmo MostFrequentWord(D)

Input: Documento $D = D[0], \dots, D[N - 1]$ di N parole.

Output: Una parola w con il massimo numero di occorrenze.

Soluzioni banali:

- Usare due cicli for, contando per ogni parola le sue occorrenze
 $\Rightarrow \Theta(n^2)$;
- Ordinare in maniera lessicografica il documento ($\Theta(n \log n)$) e contare il numero di parole in tempo lineare $\Rightarrow \Theta(n \log n)$.

Idea:

- Scansione lineare di D ;
- Mantenere in una mappa di appoggio M coppie (parola, numero occorrenze), tenendo anche traccia della parola più frequente.

```

M <- mappa vuota;
maxCount <- 0; //si può inizializzare maxCount, ma non serve, visto che prende il valore della parola con più occorrenze
for i <- 0 to N-1 do{
    count <- M.get(D[i]);
    if(count = null) then count <- 0;
    M.put(D[i], ++count);
    if(count > maxCount) then{
        maxCount <- count;
        maxWord <- D[i];
    }
}
return maxWord

```

Complessità:

- $\Theta(N)$ operazioni (escluse le get e le put sulla mappa);
- N get e N put sulla Mappa, la cui complessità dipende dalla implementazione scelta per la Mappa;

HashTable: N get e N put richiedono $\Theta(N(1 + \lambda))$ operazioni al caso medio, dove λ = load factor della tabella.

(2, 4)-Tree/Red Black Tree: N get e N put richiedono $\Theta(N \log m)$ operazioni al caso pessimo, con m il numero di parole distinte in D .

6.9.3.3. Slide 30

Siano T e U due $(2, 4)$ -Tree di altezza h e tali che la massima chiave in T è minore della minima chiave in U .

1. Progettare un algoritmo di complessità $O(h)$ per fondere T e U in un unico $(2, 4)$ -Tree TU ;

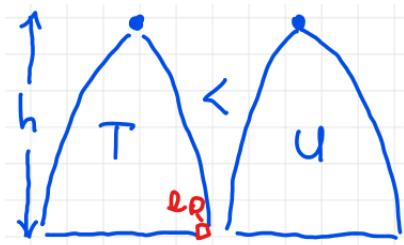
2. Dire quali valori può assumere l'altezza di TU e se la radice contiene una o più entry.

Algoritmo 24TreeMerge(T, U)

Input: $(2, 4)$ -Tree T e U di altezza h ciascuno e la massima chiave in T è più piccola della minima chiave in U .

Output: $(2, 4)$ -Tree TU fusione di T e U .

Idea:



- $e = \text{entry con chiave massima in } T;$
- metto e in un nuovo nodo r che diventa la radice del nuovo $(2-4)$ -Tree TU e la rimuovo da T invocando `Delete`.

```

v <- T.root();
z <- figlio più a destra di v;
while(T.isInternal(z)) do {
    v <- z;
    z <- figlio più a destra di v
}
e <- entry in v con chiave massima
Crea un nuovo nodo radice r contenente e, con figlio sinistro T e figlio
destro U
TU <- (2, 4)-Tree con radice r
TU.Delete(e, v)
return TU
    
```

Complessità:

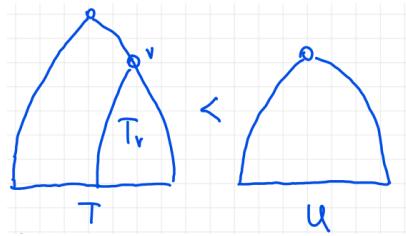
- ricerca di e : $\Theta(h)$
- creazione di TU e r : $\Theta(1)$
- `Delete` : $\Theta(h)$
 $\implies \Theta(h)$

Se `Delete` propagasse l'underflow alla radice, allora l'altezza di TU rimarrebbe h , altrimenti l'altezza di TU sarebbe $h+1$ e la radice conterrebbe una sola entry.

6.9.3.4. Slide 37

Siano T e U due $(2, 4)$ -Tree contenenti rispettivamente n ed m entry e tali che la massima chiave in T è minore della minima chiave in U . Progettare un algoritmo di complessità $O(\log n + \log m)$ per fondere T e U in un unico $(2, 4)$ -Tree TU .

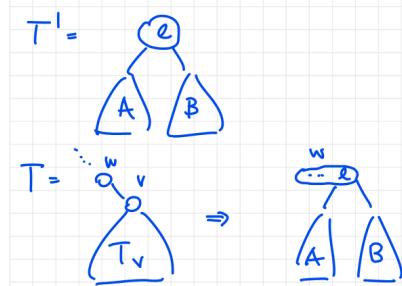
Suggerimento: Se i due alberi hanno altezze diverse, fondere l'albero di altezza minore con un opportuno sottoalbero dell'altro di uguale altezza (utilizzando l'algoritmo sviluppato per l'esercizio precedente).



Idea:

- Identifica il nodo v alla destra di T con la stessa altezza di U ;
- Fondi T_v e U in un unico $(2, 4)$ -Tree T' con il metodo `(2, 4)-TreeMerge`;
- Se T' ha la stessa altezza di T_v , allora sostituisco T_v con T' ;
- Altrimenti T' avrà l'altezza di T_v+1 e dall'esercizio precedente so che in questo caso la sua radice ha una entry. Allora metto T' al posto di T_v fondendo la sua radice con il padre di v e invocando `Split` se il padre di v va in overflow.

In questo secondo caso ecco cosa accade:



Se w va in overflow con e , si invoca `Split`.

Aggiungere i dettagli come esercizio.