Università degli Studi di Napoli
Federico II

A.A 2023-2024

# Algoritmi e Strutture Dati

# Indice

# Lista degli Algoritmi

# Introduzione

## Estratto

Il seguente documento rappresenta una raccolta di appunti, riorganizzata e migliorata nelle vesti grafiche, del corso di Algoritmi e Strutture Dati del professor M. Benerecetti, per il corso di laurea in Informatica A.A 2023-2024, della Federico II di Napoli.

All'interno del capitolo 9 sono presenti gli esercizi proposti dall'allora dottorando Francesco Altiero durante il tutorato di Algoritmi e Strutture Dati svoltosi nell'A.A 2022-2023.

Si tiene a specificare che questo documento non ha lo scopo di **sostituire libri di testo**, note del professore o lezioni frontali, nonostante siano tutte fonti utilizzate per la stesura, ma quello di integrare al materiale più complesso una rilettura degli argomenti da un punto di vista meno tecnico e quindi più lento nel soffermarsi nei passaggi critici.

Si ringraziano **Valentino Bocchetti** e **Pasquale Miranda** ideatori e gestori del progetto *Unina Docs*,

Si ringraziano inoltre tutti i revisori che hanno contribuito al documento con consigli e proposte di modifica, invitando a dare feedback in caso di errori.

Si ringraziano in fine i lettori nella speranza che questo lavoro sia stato utile.

*Redazione a cura di* **Francesco Donnarumma**, **Giuseppe Di Martino**, **Giorgio Di Fusco**

*Revisione testo a cura di* Riccardo Elena, Valentino Bocchetti

*Si ringraziano* Giuseppe Di Martino, Luigi Dota, Gianluca Fiorentino, Salvatore Carleo per il materiale fornito

# Primo sguardo all'analisi algoritmica

## 1.1 Definizione e proprietà degli algoritmi

Prima d'iniziare ad analizzare un algoritmo dobbiamo prima capire cos'è un algoritmo.

**Algoritmo**

Un **algoritmo** è una procedura ben definita che prende un certo valore, o un insieme di valori, come **input** e genera un valore, o un insieme di valori, come **output**.

Quindi un algoritmo lo possiamo interpretare come *una sequenza finita di passi* che, se eseguiti da un **esecutore**, portano alla soluzione di un **problema computazionale** ben definito.

In generale un algoritmo gode delle seguenti proprietà:

- **Non ambiguità:** tutti i passi che definiscono l'algoritmo devono essere non ambigui e chiaramente comprensibili dall'esecutore;
- **Generalità:** la sequenza di passi da eseguire dipende esclusivamente dal problema generale da risolvere, non dai dati che ne definiscono un'istanza specifica;
- **Correttezza:** un algoritmo è **corretto** se produce il risultato corretto a fronte di qualsiasi istanza del problema ricevuta in ingresso. Può essere stabilita, ad esempio, tramite:
  - dimostrazione formale (matematica);
  - ispezione informale;
- **Efficienza:** misura delle risorse computazionali che esso impiega per risolvere un problema. Alcuni esempi sono:
  - tempo di esecuzione;
  - memoria impiegata;
  - altre risorse: banda di comunicazione.

### 1.1.1 Il tempo di esecuzione di un algoritmo

Spesso, quando si analizzano le proprietà di un algoritmo si parla di **complessità computazionale**. Questa può essere definita come segue:

**Complessità computazionale**

La **complessità computazionale** è il costo, in termini di tempo e memoria, necessari per eseguire un algoritmo, quindi per risolvere un problema.

Per determinare la complessità computazionale di un algoritmo bisognerà quindi definire al meglio il concetto di *tempo di esecuzione*.

**Tempo di esecuzione**

Il **tempo di esecuzione** di un programma per un particolare input è il numero di operazioni primitive che vengono eseguite o "passi".

Il tempo di esecuzione può dipendere da vari fattori:

- Hardware su cui viene eseguito;
- Compilatore/Interprete utilizzato;
- Tipo e dimensione dell'input;
- Altri fattori: casualità

Una misura del costo computazionale soddisfacente deve:

1. Basarsi su un **modello computazionale** in modo tale da poter definire il concetto di **passo algoritmico** nel modo più indipendente possibile dal tipo di esecutore;
2. Svincolarsi dalla configurazione dei dati in ingresso, ad esempio basandosi sulle configurazioni più sfavorevoli (caso peggiore), così da garantire che le prestazioni nei casi reali saranno al più costose quanto il caso analizzato;
3. Essere una funzione della dimensione dell'input;
4. Essere asintotica, cioè fornire un'idea dell'andamento del costo all'aumentare della dimensione dell'input[1].

Una misura del costo computazione di un algoritmo deve quindi fornire una descrizione del costo asintotico nel caso peggiore, ovvero l'insieme degli input per i quali l'algoritmo richiede il massimo tempo di esecuzione.

### La macchina di Turing

Alla base di un modello computazionale sta l'idea che ogni istruzione atomica abbia un **costo unitario**. Un noto modello computazionale è il modello della **macchina di Turing (MdT)**. Si tratta di una macchina astratta che manipola i dati contenuti su un nastro di lunghezza potenzialmente infinita, secondo un insieme prefissato di regole ben definite. Questo modello è largamente utilizzato nella teoria della calcolabilità e nello studio della complessità degli algoritmi, in quanto è di notevole aiuto agli studiosi nel comprendere i limiti del calcolo meccanico; la sua importanza è tale che oggi, per definire in modo formalmente preciso la nozione di algoritmo, si tende a ricondurlo alle elaborazioni effettuabili con macchine di Turing.

Una MdT, come mostrato in Figura 1.1, è composta da:

- Un **nastro di lunghezza infinita** costituito da celle le quali possono contenere una quantità d'informazione finita.
- Una **testina**, un **processore**, un **programma**

Le operazioni che può compiere in una **singola unità di tempo** sono:

- Leggere o scrivere nella cella puntata attualmente dalla testina;
- Muoversi di una cella a destra, a sinistra oppure restare ferma.

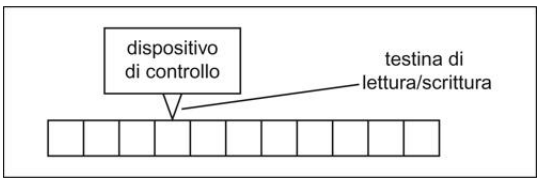



Figura 1.1: Macchina di Turing

## 1.2 ALGORITMO CONTA COPPIE

Proviamo a studiare un algoritmo che fa quanto segue:

- **Input:** $n \in \mathbb{N}^*$, dove per $\mathbb{N}^*$ si intende $\mathbb{N} \setminus \{0\}$.
- **Output:** numero di coppie $(i, j)$ con $1 \leq i \leq j \leq n$

**Esempio:** $n = 4 \rightarrow 10$ casi: $(1,1), (1,2), (1,3), (1,4), (2,2), (2,3), (2,4), (3,3), (4,4)$

[1] Senza essere troppo precisi nell'analisi della grandezza di tale input. Nel nostro caso di studio infatti non stiamo considerando la macchina effettiva che esegue il calcolo.

### 1.2.1 Un metodo di risoluzione naïf

Il primo step per risolvere un problema difficile è quello di **decomporlo in problemi più semplici** seguendo l'approccio *divide et impera*. L'idea quindi sarebbe quella di generare tutte le coppie e successivamente contare quelle che soddisfano la condizione posta. Si ottiene così:

```
1 RIS = 0
2 for i = 1 to n do
3     for j = 1 to n do
4         if i ⩽ j then
5             RIS = RIS + 1
6 return RIS
```

Algoritmo 1.1: Conta1(n)

Bisogna adesso porsi il problema di come associare un tempo di esecuzione all'Algoritmo 1.1. Nel farlo assumiamo che il valore dell'input sia **misura della complessità dell'algoritmo**[2]. Analizziamo la complessità di *ogni linea* contandone le operazioni elementari e valutando il numero di volte in cui queste vengono eseguite:

1. L'istruzione contiene una singola operazione elementare che viene eseguita a tempo costante.
2. Il ciclo `for` viene eseguito $n+1$ volte. Infatti, oltre alle $n$ iterazioni date dall'estremo superiore, viene effettuata un'iterata in più per il controllo che permette all'esecutore di uscire dal ciclo una volta che la condizione risulta falsa. Il contributo della riga sarà quindi:
$$2 \cdot \sum_{i=1}^{n+1} = 2 \cdot (n+1)$$
3. Questo ciclo si trova annidato al primo ciclo `for`. Quindi ciascuna delle sue $n+1$ iterazioni sarà ripetuta $n$ volte. In totale:
$$2 \cdot \sum_{i=1}^{n} (\sum_{j=1}^{n+1} 1) = 2 \cdot \sum_{i=1}^{n} (n+1) = 2 \cdot n \cdot (n+1)$$
4. L'istruzione `if` si trova all'interno dei due cicli for, sarà quindi ripetuta:
$$3 \cdot \sum_{i=1}^{n} (\sum_{j=1}^{n} 1) = 3 \cdot n^2$$
volte.
5. È impossibile determinare a priori il numero di volte in cui l'istruzione all'interno dell'istruzione `if`, essendo questo dipendente dal soddisfacimento o meno della condizione imposta alla coppia $(i, j)$. Nel caso in questione possiamo osservare che, se $i = 1$, il corpo dell'`if` viene eseguito $n$ volte; con $i = 2$ viene eseguito $n-1$ volte, ... con $i = k$ viene eseguito $n - (k+1)$, dove $k$ è il valore corrente di $i$. In generale:
$$\sum_{i=1}^{n} (n - i + 1)$$

Per calcolare l'ultima sommatoria basta spezzarla in tre sommatorie:

$$\sum_{i=1}^{n} (n - i + 1) = \sum_{i=1}^{n} n - \sum_{i=1}^{n} i + \sum_{i=1}^{n} 1 = n^2 - \frac{n(n+1)}{2} + n = n(n+1) - \frac{n(n+1)}{2} = \frac{n(n+1)}{2} \tag{1.1}$$

Ora per calcolare il tempo di esecuzione, basta sommare:

$$\begin{aligned} T_1(n) &= 1 + 2 \cdot (n+1) + 2 \cdot (n^2 + n) + 3n^2 + \frac{n(n+1)}{2} \\ &= 1 + 2n + 2 + 2n^2 + 2n + 3n^2 + \frac{n^2}{2} + \frac{n}{2} \\ &= \frac{11}{2} n^2 + \frac{9}{2} n + 4 \end{aligned}$$

Il nostro algoritmo ha tempo **quadratico**. È facile convincersi del fatto che l'algoritmo1.1 è molto laborioso in quanto deve generare ben $n^2$ coppie quando potrebbe sicuramente generarne di meno.

[2] In generale non sarà così ma assumiamo che nel nostro modello computazionale i dati vengano rappresentati attraverso un sistema unario.

### 1.2.2 Migliorare l'algoritmo tramite l'analisi

L'analisi del tempo di esecuzione dell'Algoritmo 1.1 ci ha permesso di prevedere all'$i$-esima iterazione del `for` a linea 2 quante volte veniva eseguita il corpo dell'`if`. Proviamo a usare questa conoscenza a nostro vantaggio per migliorare l'algoritmo.

Ciò che si era notato era che, fissato $i$, non c'è bisogno di effettuare alcun controllo in quanto il numero di coppie $(i, j)$ che soddisfano alla nostra condizione è pari a $\sum_{i=1}^{n}(n - i + 1)$, ovvero:

$$\frac{n(n+1)}{2} \tag{1.2}$$

A questo punto proviamo a scrivere il nuovo codice utilizzando questo concetto.

```
1  RIS = 0
2  for i = 1 to n do
3      RIS = RIS + (n-i+1)
4  return RIS
```

Algoritmo 1.2: Conta2(n)

Analizzando nuovamente, istruzione per istruzione, il costo computazionale dell'algoritmo si evince che $T_2(n) = 9n + 4$. Infatti:

1. Tempo costante;
2. $2\sum_{i=1}^{n+1} 1 = 2n + 2$;
3. $(3 + 3 + 1)n = 7n$;
4. Tempo costante.

Il nostro algoritmo è migliorato in quanto adesso il tempo di esecuzione è lineare. Da quadratico a lineare è un grande miglioramento, ma possiamo fare di meglio. Infatti, adesso l'algoritmo non fa nient'altro che calcolare $\sum_{i=1}^{n} i$ di cui, come abbiamo già osservato, esiste una diretta correlazione con la somma dei primi $n$ numeri. I due problemi sono infatti isomorfi. Dunque abbiamo:

```
1  RIS = (n(n+1))/2
2  return RIS
```

Algoritmo 1.3: Conta3(n)

**Tempo di esecuzione per ogni istruzione:**

1. $2 + 3 = 5$
2. 1

Dall'analisi si evince che $T_3(n) = costante$.

## 1.3 Somma della massima sottosequenza

Proviamo a studiare un algoritmo che fa quanto segue:

- **Input:** $n \in \mathbb{N}^*, A$. Dova A è un array di dimensione N
- **Output:** La somma della massima sotto-sequenza di A

Esempio

Sia $n = 7$, e consideriamo l'array: {1, -3, 8, 2, -10, 1, 7}. In questo caso, la massima sottosequenza ha somma 10, infatti: $MaxSum = 8 + 2$.

### 1.3.1 Introduzione e prima risoluzione del problema

Una prima idea, potrebbe essere quella di applicare un algoritmo ***brute force***. Quindi dobbiamo applicare i seguenti passi:

1. Generare tutte le possibili sotto-sequenze.
2. Calcolarne la somma.
3. Aggiornare il max.

Per generare tutte le possibili sotto-sequenze possiamo scegliere di rappresentarla con una coppia $(i, j)$, dove $i$ è l'indice del primo elemento, e $j$ quello dell'ultimo, tale che $i \leq j$.

```
1  MAX = 0
2      for i=1 to n do
3          for j=i to n do
4              SUM = 0
5              for k=i to j do
6                  SUM = SUM + A[k]
7              if SUM > MAX then
8                  MAX = SUM
9  return MAX
```

Algoritmo 1.4: MaxSum1(A,n)

Effettuando un'analisi non approfondita ci rendiamo conto dai tre cicli `for` innestati che l'algoritmo in questione ha un costo cubico ($T_1 = O(n^3)$) rispetto all'input. Notare che nell'algoritmo 1.4 vengono effettuate diverse volte le stesse operazioni, questo perché andiamo a calcolare ogni volta da capo la somma, infatti:

$$\sum_{k=i}^{j+1} A[k] = \sum_{k=i}^{j} A[k] + A[k+1] \tag{1.3}$$

Dove $\sum_{k=i}^{j} A[k]$ è l'operazione ripetuta più volte. Per migliorare questo aspetto, possiamo conservare la somma e aggiungere solo il successivo per ogni sequenza. Vediamo come evolve il codice dopo questa modifica.

```
1  MAX = 0
2  for i=1 to n do
3      SUM = 0
4      for j=i to n do
5          SUM = SUM + A[j]
6          if SUM > MAX then
7              MAX = SUM
8  return MAX
```

Algoritmo 1.5: MaxSum2(A,n)

Questa modifica ci consente di ridurre il costo del nostro algoritmo da cubico a quadratico. $T_2 = (O(n^2))$.

### 1.3.2 Algoritmo MaxSum lineare

Prima di procedere a scrivere l'algoritmo, fermiamoci a ragionare sul come possiamo "scartare" parti di array, in modo da non doverlo controllare tutto per cercare la massima sotto-sequenza. Per semplicità consideriamo di avere solo tre valori: $a$, $b$, $c$. Di questi, ipotizziamo siano vere:

1. $a < b$;
2. $b < c$.

Quindi per transitività sappiamo che $a < c$. Potrà sembrare una banalità, ma così facendo risparmiamo un confronto, ed è proprio la chiave per ottimizzare l'algoritmo. Ipotizziamo sia vero quanto segue:

$$\sum_{z=i}^{j} A[z] \geq 0 \text{ , con } i \leq j \leq r-1 \tag{1.4}$$

Si hanno allora due possibili casi:

1. $\sum_{z=i}^{r} A[z] \geq 0$, allora:

$$\sum_{z=i}^{k} A[z] = \sum_{z=i}^{j} A[z] + \sum_{z=j+1}^{k} A[z] \geq \sum_{z=j+1}^{k} A[z]$$

Se $a$ un numero sommo una quantità positiva, ottengo un numero più grande, banalmente. Di conseguenza è inutile calcolare le sottosequenze interne ad $i$ ed $r$.

2. $\sum_{z=i}^{r} A[z] < 0$, allora:

$$\sum_{z=j}^{k} A[z] = \sum_{z=i}^{r} A[z] + \sum_{z=r+1}^{k} A[z] < \sum_{z=r+1}^{k} A[z]$$

Sappiamo che la prima sommatoria è negativa semplicemente perché per ipotesi $\sum_{z=i}^{j-1} A[z] + \sum_{z=j}^{r} A[z] < 0$.

Ricapitolando, $\sum_{z=j}^{k} A[z] \leq \sum_{z=r+1}^{k} A[z]$, cioè è inutile calcolare ogni sottosequenza che inizia tra $i$ ed $r$ e termina oltre $r$.

Facendo questa analisi siamo riusciti a eliminare un gran numero di calcoli e confronti superflui. Andiamo a vedere come evolve l'algoritmo.

```
1  MAX = 0
2  SUM = 0
3  for i=1 to n do
4      SUM = SUM + A[i]
5      if SUM < 0 then
6          SUM = 0
7      else
8          MAX = MAX(SUM, MAX)
9  return MAX
```

Algoritmo 1.6: MaxSum3(A, n)

Andando a fare un'analisi del codice vediamo che abbiamo un solo `for` che va da 1 a $n$, ovvero, andiamo a eseguire tante istruzioni quanto $n$.

In conclusione, andando a eliminare confronti e calcoli superflui, siamo riusciti nell'intento di rendere il nostro algoritmo lineare. $T_3 = O(n)$.

# STRUTTURE DATI ELEMENTARI

## 2.1 CHE COS'È UNA STRUTTURA DATI

Gli insiemi rappresentano un concetto fondamentale per l'informatica e la matematica. Mentre gli insiemi matematici sono immutabili, gli insiemi manipolati dagli algoritmi possono crescere, ridursi o cambiare nel tempo. Per questo motivo questi insiemi sono detti ***dinamici***.

**Insieme Dinamico**

Sia n ≥ 0 e sia S = $\{e_1, e_2, ..., e_n\}$ un ***insieme***, cioè una collezione di oggetti distinguibili, che si denotano come elementi (o attributi). Un insieme è detto ***dinamico*** se e soltanto se la sua cardinalità può variare nel tempo, cioè può variare il numero dei suoi elementi.

Gli algoritmi per la risoluzione di un problema possono richiedere vari tipi di operazioni da svolgere sugli insiemi.

**Struttura dati astratta**

Si dice **struttura dati astratta** un insieme dinamico definita da un punto di vista logico, descrivendo cioè soltanto le associazioni logiche tra i dati e le operazioni mediante le quali utilizzare la struttura.

Le **caratteristiche** principali che differenziano le strutture dati astratte sono:

- La possibilità di cambiare o meno dimensione durante l'esecuzione (**dinamica** o **statica**);
- Il fatto che i dati siano tutti dello stesso tipo oppure no (**omogenea** o **eterogenea**);
- Il fatto che sia possibile accedere direttamente a un elemento, o che sia invece necessario scorrere tutti gli elementi precedenti (**accesso diretto** o **sequenziale**);

**Struttura dati concreta**

Una struttura dati **concreta** è la rappresentazione nella memoria del computer di una struttura dati astratta.

Esempio

Un esempio di struttura dati astratta potrebbe essere una sequenza di numeri, una sua possibile struttura concreta può essere un vettore.

La caratteristica principale che differenzia le strutture di dati concrete è il fatto che i dati siano memorizzati in memoria in locazioni di memoria contigue oppure no. Le strutture dati vengono memorizzate in memoria ed esistono soltanto all'interno del programma che le utilizza; quando il programma termina, i dati inseriti nella struttura non sono più utilizzabili; non è possibile conservare dati in queste strutture, né usarle per comunicare dati tra un programma e un altro.

### 2.1.1 Gli elementi di un insieme dinamico

In una tipica implementazione di un insieme dinamico, ogni elemento è rappresentato da un oggetto i cui attributi possono essere esaminati e manipolati se c'è un puntatore all'oggetto. Per alcuni tipi di insiemi dinamici si suppone inoltre che uno degli attributi dell'oggetto sia una **chiave** di identificazione. Se le chiavi sono tutte diverse, possiamo pensare all'insieme dinamico come a un insieme di valori chiave. L'oggetto può contenere **dati satelliti**, che vengono trasportati in altri attributi dell'oggetto oppure includere ulteriori attributi che possono essere manipolati dalle operazioni svolte sull'insieme; questi attributi possono contenere dati o puntatori ad altri oggetti dell'insieme.

### 2.1.2 Operazioni sulle strutture dati

Le operazioni su un insieme dinamico possono essere raggruppate in due categorie: le **interrogazioni** che restituiscono semplicemente informazioni sull'insieme; le **operazioni di modifica** che cambiano l'insieme.

Consideriamo un insieme con una relazione d'ordine $\leq$ e assumiamo che l'elemento NIL non sia mai presente all'insieme $(S, \leq)$. Per tale insieme possiamo definire le seguenti operazioni:

- RICERCA(S,k): restituisce un elemento di S oppure NIL se $k \notin S$.
- INSERIMENTO(S,k): restituisce un nuovo insieme $S' = S \cup \{k\}$.
- CANCELLAZIONE(S,k): restituisce un nuovo insieme $S' = S \setminus \{k\}$.

Gli algoritmi per le operazioni sugli insiemi sfruttano le caratteristiche della rappresentazione dell'insieme e questo significa che le operazioni di modifica (inserimento e cancellazione) **dovranno mantenere intatte quelle caratteristiche** (ad esempio se devo aggiungere un elemento in una sequenza ordinata, devo aggiungerlo nella giusta posizione in modo da lasciare ordinata la sequenza). Chiaramente la ricerca è un operazione che non modifica la struttura dati e quindi preserva naturalmente le proprietà della struttura, mentre per le altre due, più vincoli avremo e più complesso sarà definire delle operazioni.

L'operazione di ricerca non si limita solo alla ricerca dell'elemento $k$ nell'insieme $S$, posso infatti ampliare tale operazione con altre operazioni di ricerca:

- SUCCESSORE(S,k): restituisce l'elemento con la più piccola chiave $a > k$.
- PREDECESSORE(S,k): restituisce l'elemento con la più grande chiave $a < k$.
- MINIMO(S,k): restituisce un puntatore all'elemento di S con la chiave più piccola.
- MASSIMO(S,k): restituisce un puntatore all'elemento di S con la chiave più grande.

## 2.2 ARRAY

L'**array** è un potente strumento ampiamente usato in programmazione. Gli array servono ad immagazzinare ed organizzare i dati nella memoria di un calcolatore: la loro potenza deriva soprattutto dal fatto che forniscono un modo molto semplice ed efficace di eseguire e fare riferimento a computazioni su collezioni di dati che condividono attributi comuni.

**Array**

Un ***array*** è un insieme ***statico***, in cui la *dimensione* dello stesso è dunque *prefissata*.

È possibile vedere gli array come un'applicazione fatta in questo modo:

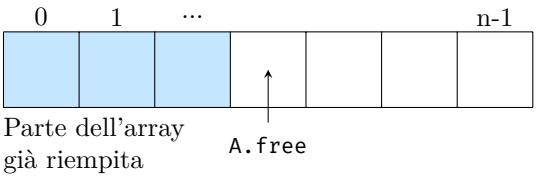
$$\forall n \in \mathbb{N}, I_n = \{0, 1, 2, ..., n-1\} \mapsto d \in A \tag{2.1}$$

La quale ha come dominio l'insieme dei primi $n$ numeri e come codominio gli elementi contenuti nell'array. Per denotare quindi l'$i$-esimo elemento di $A$ si è soliti utilizzare la seguente notazione: $A[i] = d$. Un modo alternativo per vedere l'array è considerarlo come l'unione di ciascuno dei suoi singoli elementi, ovvero:

$$A = \bigcup_{i=0}^{n-1} \{A[i]\} \tag{2.2}$$

### 2.2.1 Il problema dell'inserimento

Affrontiamo ora il problema dell'inserimento in un array. Consideriamo quindi la nostra struttura dati fatta in questo modo:




In questa struttura abbiamo un puntatore chiamato `A.free` che mira alla prima cella vuota dell'array. Se volessimo inserire un nuovo elemento, ammesso che l'array non sia pieno, si tratterebbe semplicemente di eseguire una operazione a *tempo costante*. Questo perché l'array è una struttura che gode di **accesso diretto** alla memoria, grazie al fatto che le celle dell'array sono *contigue*. Per i vari casi di studio che seguiranno, si faranno le seguenti assunzioni:

- S è un insieme di dati dinamico contenuto in un intervallo contiguo di A (la cui lunghezza è predeterminata);
- S non ammette duplicati;

Per inserire un nuovo elemento all'interno del vettore, la prima operazione da effettuare è quella di assicurarsi che l'elemento non sia già presente nella struttura dati. Essendo l'array non ordinato, il problema della ricerca si riduce nell'esecuzione lineare di una lettura ed un confronto, a partire dall'inizio o dalla fine della struttura, fino all'esaurimento dell'array. Tale operazione ci tornerà parecchio utile sia per l'inserimento che per la cancellazione di un elemento in un vettore.

```
1  pos = A.free - 1
2  while A[pos] ≠ k && pos ⩾ 0 do
3    pos = pos - 1
4  return pos
```

Algoritmo 2.1: Search(A,k)

L'algoritmo 2.1 restituisce la posizione dell'elemento nell'array se l'ha trovato, altrimenti restituisce -1. Andando a fare un'analisi approssimativa del codice è facilmente osservabile che l'algoritmo ha un *tempo di esecuzione lineare* dato dal controllo sequenziale di ogni singola cella del vettore. Di conseguenza il caso peggiore, nel caso della ricerca di un elemento in un vettore non ordinato, è ottenuto nel caso in cui l'array sia pieno, ovvero quando `A.free = n-1`.

Grazie all'algoritmo di ricerca è possibile implementare gli algoritmi di inserimento e rimozione di un elemento. Infatti, prima di eseguire tali operazioni bisogna innanzitutto verificare la presenza dell'elemento nell'insieme. Nel caso dell'inserimento di un nuovo elemento, qualora il vettore dovesse essere già pieno, è possibile adoperare una funzione usiliaria `Resize` che effettui il ridimensionamento del vettore[1]. Nel caso della cancellazione di un elemento, invece, non è necessario effettuare alcun ridimensionamento ma semplicemente sostituire all'interno della cella in cui si trova l'elemento da cancellare con il valore dell'ultimo elemento dell'array ed infine decrementare `A.free` di 1.

```
1  pos = Search(A, k)
2  if pos = -1 then
3    if A.free = length(A) then
4      A = Resize(A)
5    A[A.free] = k
6    A.free = A.free + 1
```

Algoritmo 2.2: Insert(A,k)

```
1  pos = Search(A, k)
2  if pos ⩾ 0 then
3      A[pos] = A[A.free-1]
4      A.free = A.free - 1
```

Algoritmo 2.3: Delete(A,k)

Consideriamo il caso in cui l'insieme dinamico memorizzato nell'array sia un insieme all'interno del quale è stata definita una relazione d'ordine ≤. In questa situazione ogni elemento dell'array deve soddisfare alla seguente condizione:

$$\forall i\big(\ 1 \le i \le n \Rightarrow S[i] \le S[i+1]\big)$$

Grazie a questo vincolo aggiuntivo possiamo osservare che è possibile migliorare notevolmente le operazioni di ricerca. Infatti, nel caso dei vettori non ordinati, la ricerca dell'elemento massimo o minimo richiede sempre tempo lineare dovendo scorrere l'array nella sua interezza mentre, in un vettore ordinato, l'operazione richiede una singola operazione di lettura del primo elemento (nel caso della ricerca del minimo) o dell'ultimo elemento del vettore (nel caso della ricerca del massimo).

Nonostante i vantaggi ottenuti, nel caso della ricerca di un elemento, notiamo un peggioramento dal punto di vista computazionale nel caso delle operazioni di inserimento e cancellazione. Infatti, se volessimo inserire un nuovo elemento nell'array si può constatare facilmente che questo genere di operazione sia più complicata nel caso degli array ordinati in quanto prima di inserire un qualsiasi elemento sarà necessario cercare il suo precedente, far slittare gli elementi successivi e infine mettere l'elemento nella casella appena liberata. Per un array non ordinato, al contrario, l'operazione si esegue facilmente a tempo costante in quanto basterebbe inserire il nuovo elemento alla fine del vettore. Scegliere i **vincoli della struttura dati** ha un impatto rilevante sulla complessità delle operazioni. Il punto di equilibrio dipenderà da fattori quali la dinamicità dell'insieme sul quale si sta operando, il numero di operazioni che si vogliono eseguire e la frequenza delle stesse. Da questo breve esempio si deduce che non esiste una struttura dati "perfetta": tutto dipende dalle operazioni che si vogliono eseguire su di esse.

[1] Allocando una porzione di memoria maggiore ed eseguendo la copia degli elementi dell'array originale nel nuovo vettore così ottenuto.

### 2.2.2 La ricerca binaria

Come già detto in precedenza, mentre negli array non ordinati il problema della ricerca richiede sempre un tempo lineare, negli array ordinati il problema può essere risolto in *tempo logaritmico*. Sfruttando infatti la proprietà dell'ordinamento è possibile definire un algoritmo ricorsivo chiamato RICERCA-BINARIA oppure RICERCA-DICOTOMICA il quale confronta la chiave *key* ricercata con quella dell'elemento centrale dell'array: se questa è uguale, la ricerca termina con successo, se è maggiore, la ricerca procede richiamando lo stesso metodo sulla prima metà dell'array mentre, se è minore, nella seconda metà dell'array. In sintesi RICERCA-BINARIA opera nel seguente modo:

1. Divide la sequenza che prende in ingresso determinando il valore $q = \lfloor n/2 \rfloor$;
2. Effettua il controllo $S[q] = key$, dove $key$ è il valore da ricercare; se la condizione è falsa allora:
   (a) Se $S[q] < key$ allora richiama se stesso nella sottosequenza di destra;
   (b) Se $S[q] > key$ allora richiama se stesso nella sottosequenza di sinistra.

```
1  if i ⩽ j then
2      q = ⌊(i+j)/2⌋
3      if A[q] < k then
4          ret = BinSearchRec(A, k, q+1, j)
5      else if A[q] > k then
6          ret = BinSearchRec(A, k, i, q-1)
7      else
8          ret = q
9  else
10     ret = -1
11 return ret
```

Algoritmo 2.4: BinSearchRec(A,k,i,j)

Ora andiamo a fare un'analisi di questo algoritmo per vedere se abbiamo effettivamente migliorato il tempo di esecuzione. Localmente ogni chiamata della funzione ha tempo costante $c$, quindi per calcolare il tempo di esecuzione dell'algoritmo basta fare

$$T_{Bs}(n) = \sum_{l=0}^{h+1} c$$

dove $l$ rappresenta il livello della chiamata che stiamo eseguendo ed $h+1$ è il livello che corrisponde o al caso $A[q] = k$ oppure al caso $i < j$ dove l'elemento non è presente. A questo punto bisogna capire chi è $h$. Per fare ciò ci basta osservare che ad ogni chiamata ricorsiva andiamo sempre a considerare la metà dell'array mentre nell'ultima chiamata si considera una sola cella. Quindi:

$$\frac{n}{2^h} = 1 \iff n = 2^h \iff \log_2(n) = \log_2(2^h) \iff \log_2(n) = n \tag{2.3}$$

Sostituendo nella sommatoria, otteniamo:

$$\sum_{l=0}^{h+1} c = \sum_{l=0}^{\log_2(n)+1} c = c \sum_{l=0}^{\log_2(n)+1} 1 = c\log_2(n) + c \tag{2.4}$$

Dall'equazione 2.4 osserviamo che il tempo di esecuzione della ricerca binaria $T_{Bs}(n)$ è logaritmica rispetto alla dimensione dell'array il che rappresenta un notevole miglioramento rispetto all'algoritmo SEARCH(A,K) il quale abbiamo visto avere un costo computazionale lineare rispetto alla grandezza del vettore. Tuttavia, nonostante il risparmio di tempo dato dalla ricerca binaria, non si può dire altrettanto per gli altri algoritmi di inserimento e cancellazione (INSERT e DELETE) i quali richiedono un tempo lineare rispetto alla dimensione del vettore. Infatti, inserire un singolo elemento all'interno di un vettore non ordinato prevede innanzitutto la ricerca della posizione corretta dove inserire l'elemento che ha un costo logaritmico, lo slittamento di una posizione degli elementi del vettore il quale può avere alla peggio un costo lineare se l'elemento viene inserito nella prima cella più il costo dato dall'assegnazione che viene eseguito a tempo costante:

$$T_{insert} = \log_2(n) + n + C$$

Invece, nel caso di un inserimento che necessita anche il ridimensionamento del vettore mediante la funzione RESIZE si ha un overhead lineare aggiuntivo dato dalla copia di tutti gli elementi. Al fine di ridurre il tempo di esecuzione dato dall'operazione di ridimensionamento, si può pensare di allocare nuova memoria per una grandezza doppia rispetto alla lunghezza del vettore originale, anziché limitarsi ad aggiungere una sola cella. In questo modo infatti si riduce in maniera logaritmica la necessità di dover ridimensionare il vettore a fronte di un nuovo inserimento.

**Lista concatenata**

Una ***lista concatenata*** è un insieme dinamico in cui ogni elemento ha una chiave (*key*) ed un riferimento all'elemento successivo (*next*) dell'insieme. È una struttura ad accesso ***sequenziale***. Il singolo elemento di una lista è chiamato ***nodo***.

Per capire meglio come poter definire una lista, proviamo a considerare la sommatoria dei primi $n$ numeri:

$$\sum_{i=0}^{n} i = \begin{cases} 0 & \text{se } n = 0 \\ n + \sum_{i=0}^{n-1} & \text{se } n \geq 1 \end{cases}$$

Con questa definizione possiamo calcolare la sommatoria dei primi $n$ numeri, la quale è pari a 0 se $n = 0$ ed è pari a $n$ più la sommatoria dei primi $n - 1$ numeri quando $n \geq 1$. Allo stesso modo, possiamo definire una lista $L$ nel seguente modo:

$$L = \begin{cases} \varnothing & \text{se vuota} \\ \text{Un nodo con un puntatore ad una lista L'} & \text{altrimenti} \end{cases} \tag{2.5}$$

Una lista può avere varie forme: può essere singolarmente o doppiamente concatenata, ordinata oppure no, circolare oppure no. In una lista **semplicemente concatenata** è presente, oltre all'attributo chiave, un secondo attributo chiamato ***next*** contenente l'indirizzo di memoria del nodo successivo. Nelle **liste doppiamente concatenate** è presente anche l'attributo ***prev*** contenente un puntatore al nodo precedente. Dato un nodo $x$, se $x.prev = NIL$, l'elemento $x$ non ha un predecessore e quindi è il primo elemento della lista, chiamato **testa** o **head** della lista. Se $x.next = NIL$, l'elemento $x$ non ha successore e quindi è l'ultimo elemento della lista, che è detto anche **coda** o **tail**. Un attributo $L.head$ punta al primo elemento della lista. Se $L.head = NIL$, la lista è vuota.

Se una lista è **ordinata** l'ordine lineare della lista corrisponde all'ordine lineare delle chiavi memorizzate negli elementi della lista; l'elemento minimo è la testa della lista e l'elemento massimo è la coda della lista. Una lista può essere **non ordinata**; gli elementi di questa lista possono presentarsi in qualsiasi ordine. In una **lista circolare**, il puntatore *prev* della testa della lista punta alla coda e il puntatore *next* della coda della lista punta alla testa.

Una struttura del genere ha i suoi vantaggi dal punto di vista computazionale, come l'inserimento e la cancellazione (se sono sull'elemento da cancellare) a tempo costante, ma anche i suoi svantaggi, ovvero proprio il fatto che si ha un accesso *sequenziale* alla memoria. Ciò significa che, per raggiungere un elemento della lista, bisognerà necessariamente scorrere tutta la struttura fino al nodo richiesto.

### 2.3.1 Ricerca in una lista concatenata

Data la struttura particolare delle liste concatenate non è possibile accedere direttamente ad un qualsiasi elemento come accade negli array bensì è necessario scorrere ogni nodo a partire dalla testa fino a quando non si trova l'elemento desiderato. Questo significa che per accedere ad un nodo di una lista $L$ è necessario effettuare una semplice ricerca lineare che restituisce un puntatore al nodo ricercato. Anche supponendo di avere una lista ordinata doppiamente concatenata e voler implementare la ricerca binaria, per raggiungere l'elemento della sottosequenza di grandezza $n/2^i$ bisogna partire dall'indice mediano della sequenza precedente; quindi il costo complessivo sarà:

$$\sum_{i=1}^{\log n} \left(\frac{n}{2^i}\right) = n \sum_{i=1}^{\log n} \left(\frac{1}{2}\right)^i = n - 1$$

```
1  ris = NULL
2  if L ≠ NULL then
3      if L→key = k then
4          ris = L
5      else
6          ris = Search(L→next, k)
7  return ris
```

Algoritmo 2.5: SearchRec(L, k)

```
1  ris = NULL
2  tmp = L
3  while tmp ≠ NULL && ris = NULL then
4      if L→key = k then
5          ris = tmp
6      else
7          tmp = tmp→next
8  return ris
```

Algoritmo 2.6: SearchIter(L, k)

### 2.3.2 Inserimento in una lista non ordinata

Dato un nodo $x$, l'algoritmo List-Insert(Algoritmo 2.8) inserisce un nodo con chiave `x` nella testa della lista concatenata sfruttando il fatto che ogni elemento della lista è indipendente dagli altri e non sono disposti in maniera contigua. Questa operazione prende il nome di **inserimento in testa**. Prima di eseguire l'inserimento bisogna invocare una funzione NewNode per la creazione di un nuovo nodo.

```
1  tmp = AllocaNodo()
2  tmp→key = k
3  return tmp
```

Algoritmo 2.7: NewNode(k)

```
1  ret = Search(L, k)
2  if ret = NULL then
3    tmp = NewNode(k)
4    tmp→next = L
5    L = tmp
6  return L
```

Algoritmo 2.8: Insert(L, k)

### 2.3.3 Inserimento in una lista ordinata

Inserire un nodo all'interno di una lista ordinata non ha lo stesso costo dell'inserimento in testa. Infatti prima di poter inserire il nodo bisogna garantire il vincolo dell'ordinamento andando a fare una ricerca della posizione esatta in cui andare ad inserire il nuovo nodo, più precisamente andando a cercare all'interno della lista il nodo **predecessore**. L'algoritmo che si ottiene avrà quindi un costo lineare dato dall'operazione di ricerca. Si ha così l'algoritmo InsertInOrderedList(L,k).

```
1  tmp = L
2  p = NULL
3  while tmp ≠ NIL && tmp→key < k do // Ricerca del predecessore
4      p = tmp
5      tmp = tmp→next
6  if tmp = NIL || tmp→key > key then // Inserimento di un nuovo nodo
7      new = NewNode(tmp, k)
8      if p ≠ NIL then  // Se esiste il predecessore si aggiorna il successivo
9          p→next = new
10     else  // Altrimenti si inserisce in testa
11         L = new
12 return L
```

Algoritmo 2.9: InsertInOrderedList(L, k)

```
1  if L = NIL || L→key > k then
2      L = NewNode(L, k)
3  else if L→key < k then
4      L→next = InsertInOrderRec(L→next, k)
5  return L
```

Algoritmo 2.10: InsertInOrder-Rec(L,k)

### 2.3.4 Cancellazione di un nodo in una lista

La cancellazione di un nodo da una lista non può essere fatto con un costo minore del lineare, che sia ordinata o meno la lista. Infatti la cancellazione comprende due passi: la ricerca del nodo con la chiave $k$ e la cancellazione del nodo e l'aggiornamento dei puntatori del nodo precedente e di quello successivo. Si ottiene così l'algoritmo Cancella$(L, k)$. Si supponga che la lista non contenga duplicati.

```
1  tmp = L
2  p = NULL
3  while tmp ≠ NULL && tmp→key ≠ k do
4      p = tmp
5      tmp = tmp→next
6  if tmp ≠ NULL then
7      if p ≠ NULL then
8          p→next = tmp→next
9      else
10         L = L→next
11 deallocate(tmp)
12 return L
```

Algoritmo 2.11: DeleteIter(L,k)

```
1  if L ≠ NULL then
2      if L→key = k then
3          tmp = L
4          L = L→next
5          deallocate(tmp)
6      else
7          L→next = DeleteRec(L→next,
               k)
8  return L
```

Algoritmo 2.12: DeleteRec(L, k)

Gli stack e le code sono un primo esempio di struttura dati astratta che possono essere implementate sia con un array che con una lista concatenata.

### 2.4.1 Stack

**Stack**

Uno ***stack*** è struttura dati avente politica di inserimento e cancellazione di tipo **LIFO**: Last In, First Out.

In una struttura di tipo stack gli inserimenti e le cancellazioni avvengono sempre e solo in testa alla lista ed è per questo motivo che il l'ultimo elemento ad essere stato inserito sarà sempre il primo ad essere estratto. Alcune operazioni in uno stack S sono:

- PUSH(S, k), aggiunge l'elemento $k$ **in cima** alla lista;
- POP(S), rimuove il primo elemento **dalla cima** della lista.

Sono possibili altre funzioni, come VISUALIZZAINTESTA, STACKVUOTO o STACKPIENO, che rispettivamente mostrano qual è l'elemento in testa, controlla se lo stack è vuoto oppure se è pieno.

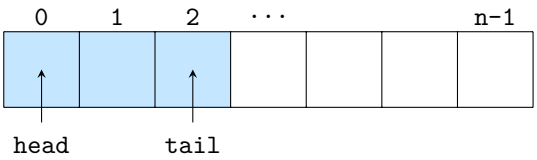
### 2.4.2 Code

**Coda**

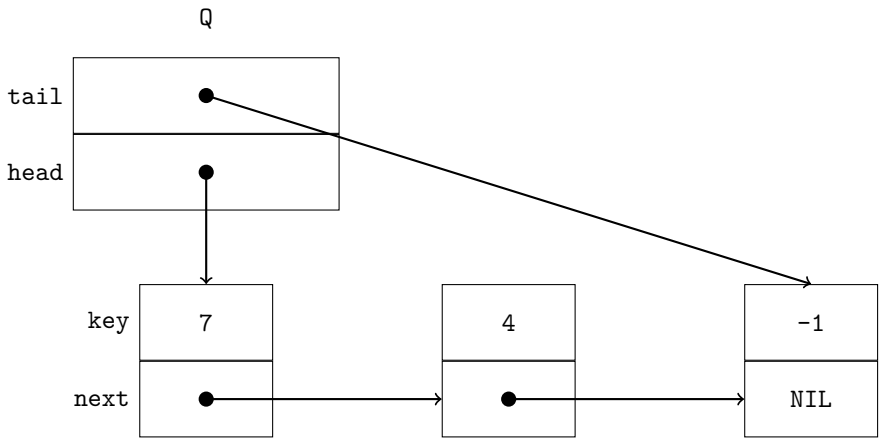
Una ***coda*** è una struttura dati avente politica di accesso e rimozionedi tipo **LIFO**: First In, First Out.

Al contrario di uno stack, quindi, in una coda gli inserimenti avvengono in coda alla lista, mentre le cancellazioni avvengono in testa alla lista. Alcune operazioni possibili[2] in una coda Q sono:

- QUEUE(S, k), aggiunge l'elemento $k$ in coda;
- DEQUEUE(S), estrae l'elemento in testa.




(a) Implementazione di una coda mediante array




(b) Implementazione di una coda mediante liste concatenate

[2] Anche per le code è possibile definire altre funzioni come per gli stack.

# STRUTTURE DATI RAMIFICATE: GLI ALBERI

## 3.1 GLI ALBERI

**Albero radicato**

Un albero è un insieme di elementi chiamati **nodi**, sui quali vengono definite relazioni di discendenza. Tra questi si distinguono gli **alberi radicati** nei quali uno dei vertici si distingue da tutti gli altri; questo vertice si chiama **radice** dell'albero.

Consideriamo un **nodo** $x$ in un albero radicato $T$ con radice $r$. Un nodo qualsiasi $y$ in un cammino semplice unico da $r$ a $x$ è detto **antenato** di $x$. Se $y$ è un antenato di $x$, allora $x$ è **discendente** di $y$. Se $y$ è un antenato di $x$ e $x \neq y$, allora $y$ è un **antenato proprio** di $x$ e $x$ è un **discendente proprio** di $y$. Il **sottoalbero con radice in** $x$ è l'albero indotto dai discendenti di $x$, con radice in $x$.

Se l'ultimo arco nel cammino semplice dalla radice $r$ di un albero $T$ a un nodo $x$ è $(y, x)$, allora $y$ è il **padre** di $x$ e $x$ è un **figlio** di $y$. La radice è l'unico nodo in $T$ che non ha un padre. Se due nodi hanno lo stesso padre, sono **fratelli**. Un nodo senza figli è un **nodo esterno** o **foglia**. Un nodo non foglia è un **nodo interno**.

Il *numero massimo di figli* che può avere un nodo $x$ in un albero radicato T è detto **grado** di $x$. La *lunghezza* del cammino semplice dalla radice $r$ a un nodo $x$ è la **profondità** di $x$ in T. Un **livello** in un albero è costituito da tutti i nodi che stanno alla stessa profondità. Un livello si dice **saturo** se ogni suo nodo ha il massimo numero possibile di nodi.

L'**altezza** di un nodo in un albero è il *numero di archi nel più lungo cammino semplice che scende dal nodo a una foglia*; l'altezza di un albero è uguale alla profondità massima di un nodo qualsiasi dell'albero.

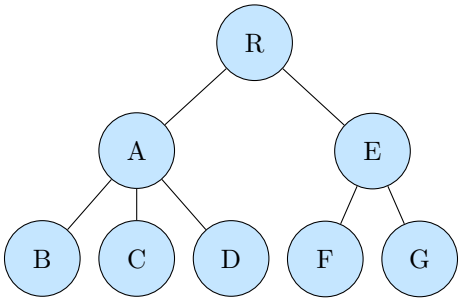



Figura 3.1: Un albero radicato con radice R e altezza 2

**Albero ordinato**

Un **albero ordinato** è un albero radicato in cui i figli di ogni noto sono ordinati. Ovvero, se un nodo ha $k$ figli, allora c'è un primo figlio, un secondo figlio,..., e un $k$-esimo figlio.

### 3.1.1 Gli alberi binari

**Albero binario**

Un **albero binario** T è particolare tipo di albero in cui ogni nodo ha grado al più due. Ricorsivamente, un albero binario può essere costruito come segue:

1. un nodo **radice**;
2. un albero binario detto **sottoalbero sinistro** della radice;
3. un albero binario detto **sottoalbero destro** della radice.

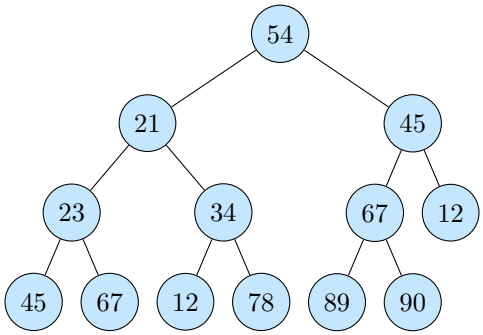



Figura 3.2: Un albero binario di altezza 3

#### Tipologie di alberi binari

Tra le varie tipologie di alberi binari troviamo:

1. Alberi binari pieni
2. Alberi binari completi

**Albero binario pieno**

Un albero binario si dice **pieno** se tutte le foglie sono sullo stesso livello e ogni nodo interno ha due figli. Per $n$ nodi un albero binario pieno ha l'altezza minima possibile.

**Albero binario completo**

Un albero binario si dice **completo** se tutti i livelli, tranne al più l'ultimo, sono saturi e le foglie dell'ultimo livello sono tutte a sinistra.

**Osservazione**

Un albero binario pieno è un albero completo ma non viceversa. L'albero mostrato in Figura 3.2 è completo ma non pieno.

#### Proprietà degli alberi binari

Gli alberi binari sono particolarmente interessanti perché godono delle seguenti proprietà che andremo a dimostrare:

**Proposizione**

Un albero binario pieno di altezza $h$ ha $2^{h+1} - 1$ nodi.

**Dimostrazione** La prova è per induzione sull'altezza $h$.

1. **Caso base** ($h = 0$): un albero binario di altezza $h = 0$ ha un solo nodo, la radice, e vale quindi:

$$2^{0+1} - 1 = 1$$

Quindi il caso base è dimostrato.

2. **Passo induttivo** ($h > 1$): assumiamo per ipotesi induttiva che un albero binario completo di altezza $h$ abbia $2^{h+1} - 1$ nodi, e dimostriamo che un albero binario $T$ completo di altezza $h+1$ ha $2^{(h+1)+1} - 1 = 2^{h+2} - 1$ nodi.

   Come è fatto $T$? Ha la radice, più un sottoalbero sinistro $T_s$ e un sottoalbero destro $T_d$ entrambi completi e di altezza $h$. Per ipotesi induttiva quindi entrambi gli alberi $T_s$ e $T_d$ hanno rispettivamente $2^{h+1} - 1$ nodi. Quindi il numero di nodi di $T$ sarà dato dalla somma dei nodi dei sottoalberi sinistro e destro e della sua radice. Detto $m$ il numero di nodi di $T$ abbiamo quindi:

$$m = 1 + (2^{h+1} - 1) + (2^{h+1} - 1) = (2 \cdot 2^{h+1}) - 1 = 2^{h+2} - 1$$

■

**Corollario**

Un albero binario pieno di altezza $h$ ha esattamente $2^h$ foglie.

**Proposizione**

Sia $T$ un albero binario completo di altezza $h$. Allora il numero di nodi di $T$ è compreso tra $2^h$ e $2^{h+1} - 1$.

**Dimostrazione** Per dimostrare questo risultato abbiamo bisogno di determinare il numero massimo e minimo di nodi di un albero binario quasi completo di altezza $h$. Il numero massimo di nodi che un albero binario quasi completo di altezza $h$ può avere è pari al numero di nodi di albero binario completo di altezza $h$, ossia $max = 2^{h+1} - 1$. Ora osserviamo che l' albero binario quasi completo di altezza $h$ con numero minimo di nodi è illustrato in Figura 3.3 (dove $T_s$ e $T_d$ sono degli alberi binari completi di altezza $h-2$):

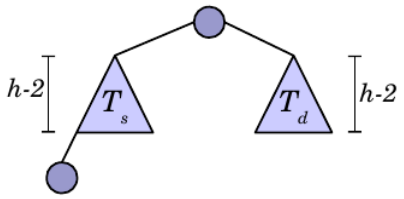



Figura 3.3

Allora: $\#_{nodi}(T) = 1 + 1 + \#_{nodi}(T_s) + \#_{nodi}(T_d)$ dove $\#_{nodi}(T_s) = \#_{nodi}(T_d) = 2^{h-1} - 1$. Quindi $\#_{nodi}(T) = 2 + (2^{h-1} - 1) + (2^{h-1} - 1) = 2 \cdot 2^{h-1} = 2^h$.

■

**Proposizione**

L'altezza di un albero binario completo $T$ con $n$ nodi è $h = \lfloor \log n \rfloor$.

**Dimostrazione** Per la proposizione precedente abbiamo che $2^h \leq n \leq 2^{h+1} - 1 < 2^{h+1}$, cioè, appplicando i logaritmi, $h \leq \log n < h + 1$ e quindi, applicando la funzione pavimento, $h \leq \lfloor \log n \rfloor < h + 1$.Possiamo concludere che $h = \lfloor \log n \rfloor$.

■

## 3.2 ALBERI BINARI DI RICERCA

Finora abbiamo esplorato le proprietà algebriche delle strutture dati, ma ora ci concentreremo su un tipo particolare di struttura: gli Alberi Binari di Ricerca (ABR). Prima di addentrarci in questo argomento, vale la pena fare un confronto con le strutture dati lineari che abbiamo precedentemente studiato.

Le strutture dati lineari, come le liste, gli array e le code, organizzano i dati in una sequenza unidimensionale, simile a una fila di oggetti allineati in una vetrina. Queste strutture sono efficaci per molte applicazioni, ma possono avere limitazioni quando si tratta di ricerche efficienti o di mantenere un ordine specifico dei dati.

Gli ABR, al contrario, adottano una struttura ad albero, in cui ogni nodo ha due sottoalberi (sinistro e destro), e i dati sono organizzati in modo gerarchico. Questa struttura a due dimensioni offre vantaggi distinti, in particolare nella ricerca rapida e nell'ordinamento automatico dei dati come si vedrà nei paragrafi successivi.

Un **albero binario di ricerca (ABR)** è una struttura dati organizzata in un albero binario, che può essere rappresentato da una struttura dati concatenata in cui ogni nodo è un oggetto. Oltre a una chiave *key* e ai dati satelliti, ogni nodo dell'albero contiene gli attributi $left$ e $right$ che puntano ai nodi che corrispondono al figlio sinistro e al figlio destro.

**Proprietà di ordinamento degli alberi binari di ricerca**

Sia $x$ un nodo in un albero ABR. Se $y$ è un nodo nel sottoalbero sinistro di $x$, allora $y.key \leq x.key$. Se $y$ è un nodo nel sottoalbero destro di $x$, allora $y.key \geq x.key$. In altre parole, i valori più piccoli sono sempre a sinistra e i valori più grandi sono sempre a destra.

Si osservi l'albero binario di ricerca in Figura 3.4 per un esempio di albero binario di ricerca. In questo albero, per ogni nodo $x$, le chiavi di tutti i nodi nel sottoalbero sinistro di $x$ sono minori o uguali a $x.key$, e le chiavi di tutti i nodi nel sottoalbero destro di $x$ sono maggiori o uguali a $x.key$.

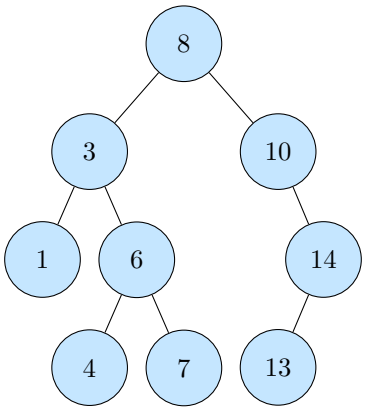



Figura 3.4: Esempio di albero binario di ricerca

Dalla proprietà degli alberi ABR si può dedurre che per ogni albero di ricerca T, sia il sottoalbero sinistro che quello destro sono alberi binari di ricerca. È possibile quindi definire induttivamente un albero binario di ricerca:

$$T = \begin{cases} \varnothing & \text{se vuoto} \\ \begin{cases} \text{un sottoalbero ABR sinistro } T_1 \wedge \ \forall y \in T_1 \quad y.key \leq root.key \\ \text{un sottoalbero ABR destro } T_2 \wedge \ \forall y \in T_2 \quad y.key \geq root.key \end{cases} & \text{altrimenti} \end{cases}$$

### 3.2.1 Operazioni di visita in un albero binario

**Visitare** un albero significa esaminare sequenzialmente tutti i suoi nodi. Le tipologie di visita si suddividono in due categorie: le **visite in profondità** e le **visite in ampiezza**.

#### Visite in ampiezza

Esistono tre tipi principali di visite in ampiezza:

1. Nella **visita in ordine** si visita il sottoalbero sinistro, quindi si esamina la radice e infine si visita il sottoalbero destro;
2. Nella **visita in pre-ordine** si visita prima la radice e quindi si visitano il sottoalbero sinistro e quello destro;
3. Nella **visita in post-ordine** si visita prima il sottoalbero sinistro, poi quello destro ed infine il nodo in radice.

```
1 if T ≠ nil then
2 Visita(T)
3 Visita-PreOrder(T→left)
4 Visita-PreOrder(T→right)
```

Algoritmo 3.1: Visita-PreOrder(T)

```
1 if T ≠ nil then
2 Visita-PostOrder(T→left)
3 Visita-PostOrder(T→right)
4 Visita(T)
```

Algoritmo 3.2: Visita-PostOrder(T)

```
1 if T ≠ nil then
2 Visita-InOrder(T→left)
3 Visita(T)
4 Visita-InOrder(T→right)
```

Algoritmo 3.3: Visita-InOrder(T)

#### Visita in profondità

Vediamo ora un altro tipo di visita chiamata **visita in ampiezza** (o **per livelli**). L'idea della visita in ampiezza è quella di visitare dapprima la radice dell'albero, poi i figli della radice, poi i figli dei figli della radice e così via fino alle foglie. In questo modo i nodi al livello $i$ saranno visitati solo dopo che tutti i nodi del livello $i-1$ sono stati visitati. La difficoltà che si incontra in questo tipo di visita è dato dal fatto che non esiste alcun collegamento tra i nodi di uno stesso livello. Per questo motivo sarà necessario avvalersi di una *struttura dati ausiliaria* dove inserire le informazioni sull'ordine dei nodi da visitare. La struttura dati più adatta a questo scopo è la **coda**. Infatti, ogni volta che si visita un nodo, i suoi figli vengono inseriti in coda. In questo

modo, quando si visita un nodo, si è sicuri che tutti i nodi di livello inferiore sono già stati visitati. L'algoritmo 3.4 mostra come implementare la visita in ampiezza.

Esempio

Si consideri l'albero binario mostrato in Figura 3.5. L'algoritmo di visita in ampiezza inizia visitando il nodo in radice che viene inserito in coda:

| 1 |
|---|

La lettura avviene rimuovendo dalla coda l'elemento in testa, quindi il nodo in radice viene rimosso dalla coda e i suoi figli vengono inseriti in coda:

| 3 | 2 |
|---|---|

Successivamente, il nodo 2 viene rimosso e si inseriscono in coda i suoi nodi figli:

| 5 | 4 | 3 |
|---|---|---|

Una volta aver estratto il nodo 3 ed aver inserito i nodi figli, bisogna solo estrarre gli elementi in coda:

| 7 | 6 | 5 | 4 |
|---|---|---|---|

In questo modo la lettura dell'albero in ampiezza produce il seguente output: `1 2 3 4 5 6 7`

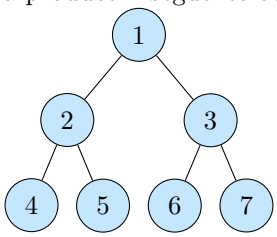



Figura 3.5

```
1  Q = {T}
2  while Q ≠ {} do
3    x = Head(Q)
4    Visita(x)
5    Q = Enqueue(Q, x→ left)
6    Q = Enqueue(Q, x→ right)
7    Q = Dequeue(Q)
```

Algoritmo 3.4: BFS(T)

### 3.2.2 Operazioni di ricerca e modifica in un albero binario di ricerca

**Ricerca di un elemento**

Supponiamo di avere un albero binario di ricerca $T$ e si voglia implementare un algoritmo SEARCH che verifichi se un elemento $k$ appartiene all'albero. Sfruttando la proprietà dell'albero ABR basterà effettuare un controllo tra $k$ e la chiave memorizzata nel nodo in radice.

- Se $x.key = k$ abbiamo trovato l'elemento;
- Se $k < x.key$ allora bisogna cercare $k$ nel sottoalbero sinistro;
- Se $k > x.key$ allora bisogna cercare $k$ nel sottoalbero destro.

Si ottiene quindi l'algoritmo 3.5 molto simile alla ricerca binaria all'interno di un array ordinato. La differenza è che, mentre la ricerca binaria divide l'array in due parti uguali, la ricerca in un albero binario di ricerca divide l'albero in due sottoalberi che possono avere dimensioni molto diverse. Il numero massimo di confronti nel caso peggiore sarà infatti pari alla lunghezza del percorso più lungo, quindi il tempo di esecuzione sarà pari all'altezza dell'albero.

```
1  if T ≠ NIL then
2    if T→key < k then
3      return Search(T→right, k)
4    else if T→key > k then
5      return Search(T→left, k)
6    else
7      return T
```

Algoritmo 3.5: Search(T,k)

### Inserimento e ricerca del minimo e del massimo

Supponiamo di voler inserire un elemento $x$ in un albero $T$. L'operazione di inserimento consiste nell'ottenere un nuovo albero, detto $T'$, posto in questo modo:

$$T' = T \cup \{x\}$$

Supposto che non debbano esserci chiavi duplicate, esistono due casi possibili:

1. **L'albero iniziale è vuoto**, quindi basta effettuare un inserimento in testa
2. **L'albero iniziale non è vuoto**, quindi $x$ deve essere inserito nel sottoalbero vuoto corretto. Infatti, se $x \notin T$, allora inserirlo in $T$ significa cercare un sottoalbero vuoto (o alla peggio una foglia) che soddisfa la proprietà dell'ordinamento.

```
1  if T = NIL then
2    x = Alloca()
3    x→key = k
4    x→left = x→right = NIL
5    T = x
6  else if T→key < x→key then
7    Insert(T→right, x)
8  else
9    Insert(T→left, x)
```

Algoritmo 3.6: Insert(T,x)

La ricerca del minimo e del massimo sono operazioni banali in un albero binario di ricerca grazie al vincolo di ordinamento globale che questi offrono. Infatti il minimo e il massimo si troveranno sempre nel nodo più a sinistra e nel nodo più a destra, rispettivamente. Ne consegue che il costo della ricerca sarà lineare sull'altezza dell'albero dato che sarà necessario seguire un percorso dalla radice fino a una foglia. Più precisamente, il minimo si troverà sempre lungo il ramo più a sinistra dell'albero e sarà caratterizzato dal fatto di non avere un figlio sinistro. Dualmente, il massimo sarà il primo nodo del percorso estremo destro che non ha un figlio destro.

```
1  ret = T
2  if T ≠ NIL then
3    x = Search-Min(T→left)
4    if x ≠ NIL then
5      ret = x
6  return ret
```

Algoritmo 3.7: Search-Min(T)

```
1  ret = T
2  if T ≠ NIL then
3    x = Search-Max(T→right)
4    if x ≠ NIL then
5      ret = x
6  return ret
```

Algoritmo 3.8: Search-Max(T)

### Ricerca del successore e del predecessore

Dato un nodo in un albero binario di ricerca, a volte è importante trovare il suo successore nell'ordine stabilito da un attraversamento simmetrico. Se tutte le chiavi sono distinte, il successore di un nodo $x$ è il nodo con la chiave più piccola che è maggiore di $x.key$. La struttura di un albero binario di ricerca consente di determinare il successore di un nodo senza mai dover confrontare le chiavi. L'algoritmo SEARCH-SUCC$(T, k)$ verifica se l'albero $T$ è vuoto, altrimenti si aprono tre strade:

- $x.key < k$: la radice contiene un valore minore della chiave, quindi se il successore esiste sarà a destra del nodo $x$, ovvero proprio il risultato che darà la chiamata SEARCH-SUCC$(T \to right, k)$;
- $x.key > k$: il successore sarà il risultato della chiamata SEARCH-SUCC$(T \to left, k)$, se $T \to left = \varnothing$ allora il successore sarà proprio il nodo $x$ (a destra avrò solo valori maggiori di $x$ quindi già so che il miglior candidato è $x$ stesso).
- $x.key = k$: è evidente che collassa al caso $x.key < k$, poiché se il successore esiste sarà nel sottoalbero destro.

```
1  if T ≠ NIL then
2    if T→key ≤ k then
3      return Search-Succ(T→right, k)
4    else
5      x = Search-Succ(T→left, k)
6      if x = NIL then
7        return T
8      else
9        return x
```

Algoritmo 3.9: Search-Succ(T,k)

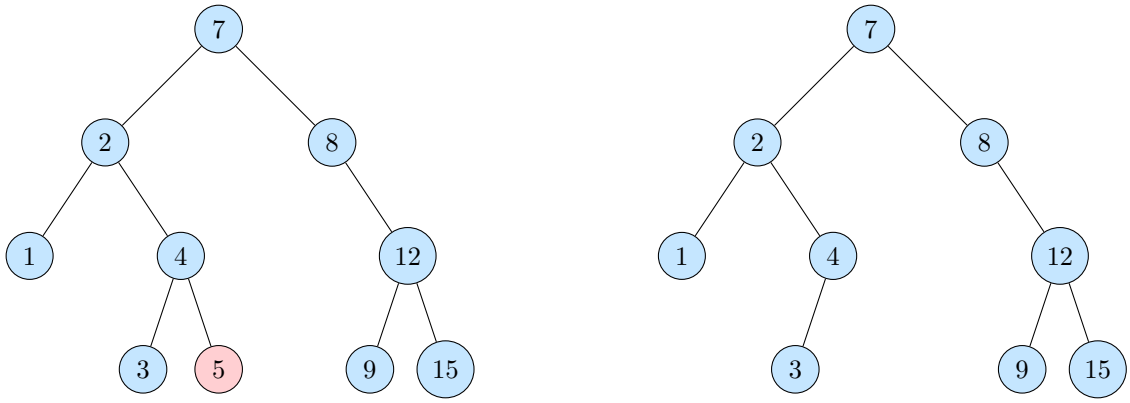
Per la versione iterativa il ragionamento è diverso. Se il nodo $k$ è presente nell'albero $T$ allora il successore sarà semplicemente il minimo del sottoalbero destro. Se invece $k$ non è presente nell'albero, allora il successore sarà il primo nodo che si incontra salendo verso la radice che ha un figlio sinistro. Se non esiste tale nodo, allora il successore non esiste.

```
node = T
succ = NIL
while nome ≠ NIL || node→key ≠ k do
    if node→key ≤ k then
      node = node→right
    else
      succ = node
      node = node→left
if node ≠ NIL || node→ right = NIL then
  succ = NIL
else
  succ = Search-Min(node→right)
return succ
```

Algoritmo 3.10: Search-Succ-Iter(T,k)

### Cancellazione di un nodo

Consideriamo l'albero binario di ricerca mostrato in Figura 3.6a. Supponiamo di voler cancellare il nodo con chiave 5. In questo caso il nodo da eliminare non ha figli e possiamo cancellarlo semplicemente impostando il puntatore `right` del padre a NIL. Questo è il caso più semplice di cancellazione.

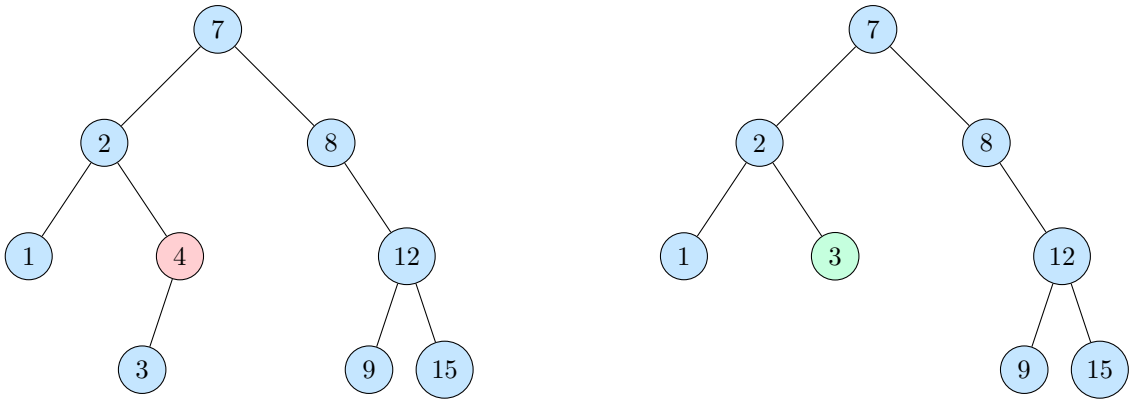



(a) Cancellazione in un albero binario di ricerca: caso 1

(b) Albero risultante dalla cancellazione

Figura 3.6

Consideriamo ora il caso in cui il nodo da eliminare abbia un solo figlio come nel caso del nodo con chiave 4 mostrato in Figura 3.7a. In questa situazione si procede eliminando il nodo e collegando il padre del nodo da eliminare con il figlio del nodo da eliminare. Nel caso in cui il nodo da eliminare sia la radice, allora il figlio del nodo da eliminare diventa la nuova radice dell'albero. (Figura 3.7b)



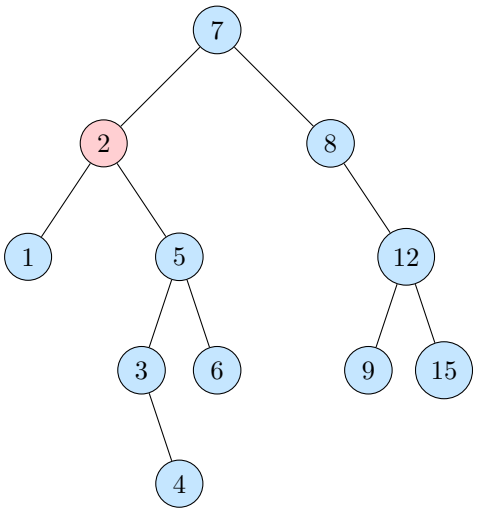

(a) Cancellazione in un albero binario di ricerca: caso 2

(b) Albero risultante dalla cancellazione

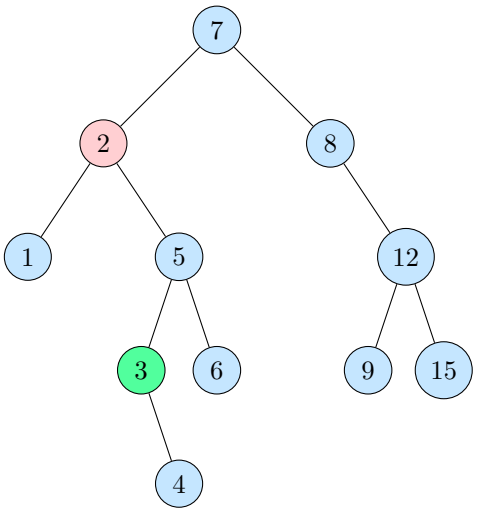
Figura 3.7

Il caso più complesso è quello in cui il nodo da eliminare ha due figli. Per esempio, consideriamo la cancellazione del nodo con chiave 2 all'interno del albero binario di ricerca mostrato in Figura 3.8a. In questo caso è necessario trovare il **successore del nodo da eliminare**, ovvero il nodo con la chiave più piccola nel sottoalbero destro del nodo da eliminare che nel nostro caso è rappresentato dal nodo di chiave 3 (Figura 3.8b).

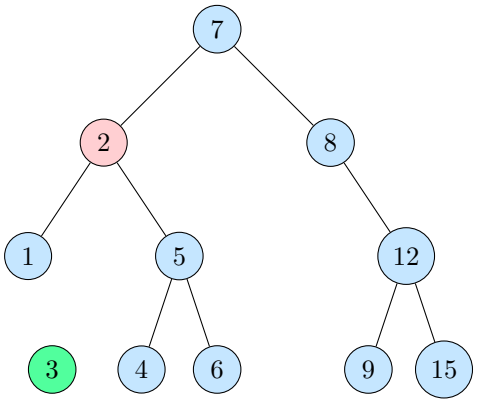
Il successore è garantito essere privo di figlio sinistro[1], quindi può essere eliminato usando uno dei due casi precedenti. La ricerca del successore si ricondurrà quindi alla ricerca del nodo minimo all'interno del sottoalbero destro del nodo da eliminare. Una volta trovato questo minimo sarà sufficiente **staccarlo** dalla sua posizione attuale e sostituirlo al nodo da eliminare. Gli eventuali figli destri del successore verranno poi attaccati al padre. (Figura 3.8d)



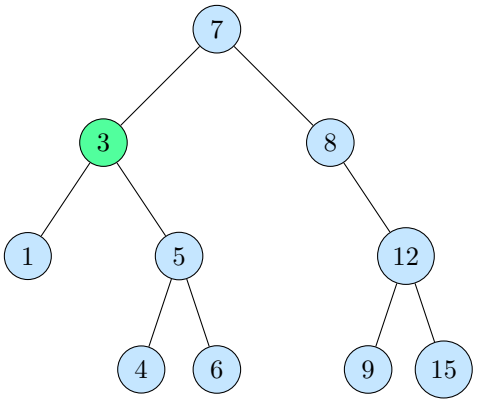

(a) Cancellazione in un albero binario di ricerca: caso 3




(b) Individuazione del successore del nodo da eliminare




(c) Staccaggio del successore




(d) Albero risultante dalla cancellazione

Figura 3.8

Per questo motivo sarà necessario definire tre algoritmi: CANCELLA$(T, k)$, CANCELLA-RADICE$(T)$ e STACCA-MINIMO$(T)$. L'algoritmo CANCELLA$(T, k)$ è l'algoritmo principale che si occupa di cancellare un nodo $k$ dall'albero $T$. Sulla base dei confronti si determina il percorso da seguire fino a quando non si arriva al nodo da eliminare, sarà poi compito dell'algoritmo DELETE-ROOT$(T)$ di discriminare i casi appena descritti.

Nel caso in cui $T$ abbia un solo figlio basterà aggiornare $T$ con l'indirizzo del figlio e deallocare la vecchia radice. Qualora invece il nodo in radice avesse due figli sarà l'algoritmo STACCA-MINIMO$(T, P)$ ad occuparsi di staccare il minimo del sottoalbero destro del nodo da eliminare. Il parametro $P$ serve per ricordarsi il padre di $T$ durante la discesa. STACCA-MINIMO non fa altro che restituire la chiave del nodo minimo presente nel sottoalbero sinistro in modo tale da eseguire una sovrascrittura delle chiavi. Dopo di che aggiorna i puntatori del nodo padre in modo tale che questo venga allacciato ai nodi restanti del sottoalbero destro.

[1] Se così non fosse allora esisterebbe un nodo con chiave minore

```
if T ≠ NIL then
  if T→key < k then
    //Cancellazione nel sottoalbero sx
    T→left = Delete(T→left, k)
  else if T→key > k then
    //Cancellazione nel sottoalbero dx
    T→right = Delete(T→right, k)
  else
    //Cancellazione in radice
    T = Delete-Root(T)
return T
```

Algoritmo 3.11: Cancella(T,k)

```
if T ≠NIL then
  // Caso 1: T ha due figli
  if T→ left  ≠ NIL && T→ right  ≠ NIL then
    tmp = Stacca-Minimo(T,T→ right)
    T → key = tmp
  else
  // Caso 2: T ha un solo figlio
    tmp = T
    if T→ right ≠ NIL then
      T = T →  right
    else
      T = T→ left
    Dealloca(tmp)
return T
```

Algoritmo 3.12: DELETE-ROOT(T)

```
if T≠NIL then
  //Caso base: T ha un figlio sinistro
  if T→left ≠ NIL then
    return Stacca-Minimo(T→left, T)
  else
  tmp = T→ key
  // Aggiorno i puntatori del padre
  // Caso 1: T era figlio sinistro del padre
    if P →left = T then
      P→left = T→right
  // Caso 2: T era figlio destro del padre
    else
      P→right = T→right
 return tmp
```

Algoritmo 3.13: STACCA-MINIMO(T,P)

# 3.3 ALBERI AVL

## 3.3.1 Alberi bilanciati

Gli alberi bilanciati di ricerca sorgono come una soluzione ai problemi di inefficienza che possono emergere negli alberi binari di ricerca. Infatti, la creazione di un albero binario di ricerca non bilanciato, in cui un sottoalbero ha molte più profondità rispetto all'altro, può portare a un'operazione di ricerca con una complessità temporale peggiorativa, che può diventare lineare rispetto al numero di elementi nell'albero. Questo è chiaramente inaccettabile in applicazioni in cui è necessaria una risposta rapida.

Gli **alberi bilanciati di ricerca** affrontano questo problema stabilendo rigorosi vincoli sulla loro struttura capaci di garantire un buon bilanciamento tra le altezze dei sottoalberi sinistro e destro di ciascun nodo. Questo bilanciamento delle altezze è ottenuto mediante regole di inserimento e rimozione particolari, e il risultato è un albero in cui le operazioni di ricerca, inserimento e cancellazione hanno una complessità temporale tipicamente logaritmica, garantendo prestazioni prevedibili e ottimali in qualsiasi scenario. Nelle prossime sezioni esploreremo le specifiche regole e le varianti di alberi bilanciati di ricerca, come gli alberi AVL e gli alberi rosso-neri per comprendere appieno come questi vincoli migliorino notevolmente l'efficienza delle operazioni sugli alberi binari di ricerca.

**Alberi bilanciati di ricerca**

Una classe di alberi binari di ricerca si dice **bilanciata** se

$$h(T) = \Theta(\log_2 n) \qquad \text{dove } n = |T| \text{ è la cardinalità di T}$$

**Osservazione**

Gli alberi binari completi appartengono alla classe degli alberi bilanciati.

Un'altra classe di alberi binari che soddisfa alle nostre esigenze è quella degli **alberi perfettamente bilanciati**.

**Alberi perfettamente bilanciati**

Un albero binario di ricerca T si dice **perfettamente bilanciato** (**ABP**) se:

$$\forall x \in T \qquad ||x \to left| - |x \to right|| \leq 1 \tag{3.1}$$

ovvero la differenza, per ogni nodo di $T$, in valore assoluto tra la cardinalità del sottoalbero sinistro e quello destro è al più uno.

Esempio

L'albero binario di ricerca in Figura 3.9 è perfettamente bilanciato, infatti per ogni nodo la differenza tra il numero di nodi nel sottoalbero sinistro e destro è al più uno. Preso ad esempio il nodo in radice, la differenza tra i suoi due sottoalberi è:

$$||root \to left| - |root \to right|| = |2 - 3| = | - 1| = 1 \leq 1$$

Gli alberi perfettamente bilanciati riescono a garantire un tempo di esecuzione logaritmico per le operazioni di ricerca, inserimento e cancellazione. Tuttavia, questa classe di alberi è molto restrittiva, poiché richiede numerosi controlli a fronte di ogni inserimento o cancellazione. Per questo motivo, gli alberi perfettamente bilanciati non sono molto utilizzati in pratica.

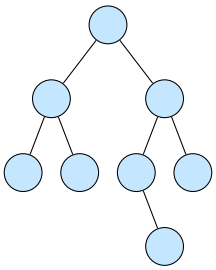


Figura 3.9: Esempio di albero perfettamente bilanciato

### 3.3.2 Gli alberi AVL

La classe di alberi **AVL (Adelson-Velskii e Landis)** rappresenta una classe di alberi bilanciati con buone prestazioni e gestione relativamente semplice. È praticamente una versione più permissiva di APB.

**Alberi AVL**

Un albero $T$ è AVL se e solo se:

$$\forall x \in T \qquad |h(x \to left) - h(x \to right)| \leq 1 \tag{3.2}$$

ovvero se, per ogni nodo dell'albero, la differenza in modulo dell'altezza del sottoalbero sinistro e il sottoalbero destro è al più uno.

Analogamente a quanto visto per le strutture dati precedenti, è possibile definire un albero AVL in modo ricorsivo: un albero binario di ricerca $T$ appartiene alla classe AVL se e soltanto se la differenza tra l'altezza del suo sottoalbero destro e il suo sottoalbero sinistro è minore o uguale di uno e i due sottoalberi appartengono alla classe AVL.

La variabilità negli alberi AVL è abbastanza ampia, infatti con lo stesso numero di nodi è possibile generare molti alberi AVL. Ma la cosa interessante è che **ogni albero perfettamente bilanciato è avl** poiché è di fatto un albero completo, il quale rispetta la proprietà di AVL per definizione.

Nel caso degli AVL però non possiamo immediatamente definire una relazione tra l'altezza e il numero dei nodi (con $n$ nodi possiamo infatti generare diversi AVL con altezza differente). Per raggiungere il nostro obiettivo dobbiamo restringere la classe AVL in una particolare sottoclasse così da far valere la proprietà per tutti gli AVL.

Generalmente, una proprietà valida per una sottoclasse non si estende alla superclasse, ma noi andremo a prendere degli alberi AVL con altezza peggiore possibile ed è intuitivo pensare che se per gli alberi peggiori vale $O(\log n)$ allora anche per tutti gli altri la relazione sarà verificata.

### 3.3.3 Alberi avl minimi

Gli alberi AVL minimi sono una sottoclasse degli alberi AVL che ci permette di definire una relazione tra l'altezza e il numero dei nodi. Questa classe di alberi è definita come segue:

**Alberi AVL minimi**

Fissato $h$, l'**albero avl minimo** di altezza $h$ è l'albero AVL col *minor numero di nodi possibile* sotto il quale l'albero perde la proprietà di essere AVL.

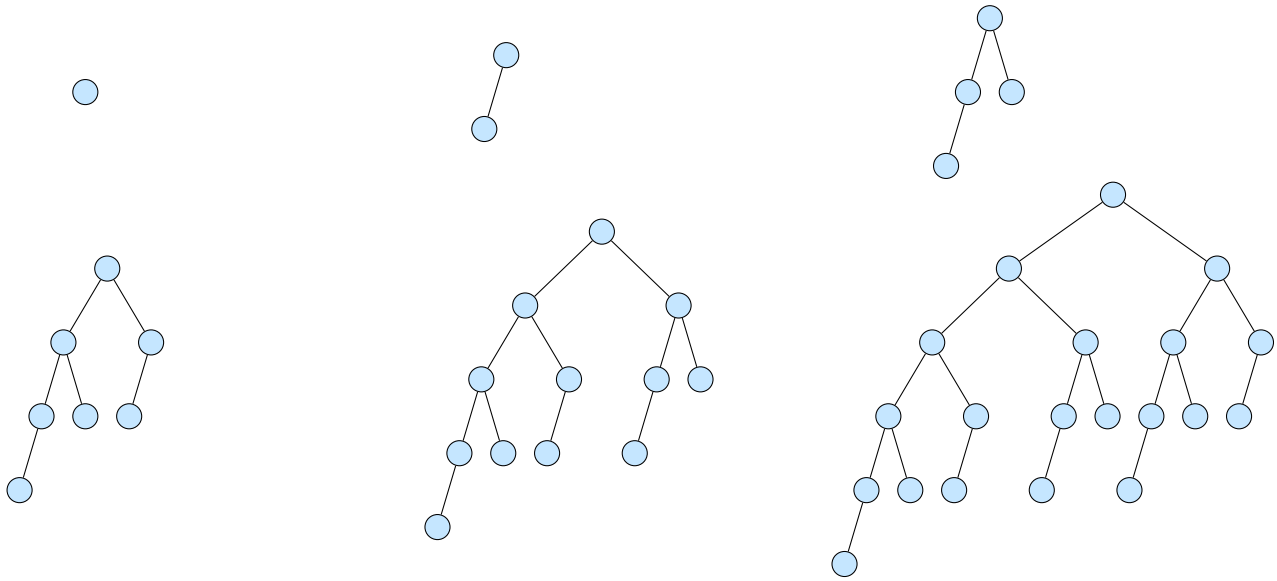


Figura 3.10: Alberi AVL minimi di altezza 0, 1, 2, 3, 4 e 5

Osservando la Figura 3.10 notiamo una certa regolarità: infatti un albero AVL minimo di altezza $h$ sarà sempre composto da due alberi AVL minimi di altezza $h-1$ e $h-2$. Vale infatti la seguente proposizione.

**Proposizione**

Sia T un albero AVL minimo di altezza $h$, vale quindi la seguente proprietà:

$$N(h) = \begin{cases} h & \text{se } h = 0, 1 \\ 1 + N(h-1) + N(h-2) & \text{se } h > 1 \end{cases} \tag{3.3}$$

dove $N(h)$ rappresenta il numero di nodi di $T$.

**Dimostrazione** Procediamo per induzione. Per $h = 0$ e $h = 1$ è immediato vedere che $N(0) = 0$ ed $N(1) = 1$, in quanto l'albero vuoto e l'albero con la sola radice sono entrambi alberi AVL.

Per $h > 1$, indichiamo con $T_h$ l'AVL minimo di altezza $h$. Per definizione, $T_h$ avrà almeno un sottoalbero AVL di altezza $h-1$, sia questo il sottoalbero $T_{sx}$. Per assurdo, assumiamo che $T_{sx}$ non sia minimo. Questo vorrà dire che esisterà un albero AVL, detto $T'_{sx}$ di altezza $h-1$ che conterrà il minor numero di nodi per tale altezza. Se consideriamo quindi un albero chiamato $T'_h$ costituito dalla radice $r$ e con uno dei sottoalberi proprio l'albero $T'_{sx}$, questo conterrà ovviamente meno nodi dell'albero $T_h$.

Questo, però, contraddirebbe l'ipotesi iniziale secondo la quale $T_h$ fosse l'AVL minimo di altezza $h$. $T_h$ sarà quindi necessariamente costituito da una radice $r$ con sottoalberi gli alberi AVL con numero minimo di nodi $T_{h-1}$ e $T_{h-2}$. Da ciò segue l'equazione 3.3.

■

**Osservazione**

Un albero AVL minimo di altezza $h$ con numero di nodi pari ad $n$ è l'albero AVL di **altezza massima** tra tutti gli alberi AVL con $n$ nodi. Ciò nonostante questi riescono a mantenere un rapporto logaritmico tra l'altezza e il numero di nodi come si vedrà più avanti.

**Proposizione**

Se $T$ è un albero binario di ricerca appartenente alla classe AVL con $n$ nodi e $T'$ è un albero binario di ricerca AVL minimo con $n$ nodi allora vale:

$$h(T') \geq h(T)$$

A parità di nodi, quindi, gli AVL minimi sono quelli che hanno l'altezza peggiore.

**Dimostrazione** Sia $T'$ un AVL minimo di altezza $h$ con n nodi. Ciò che vogliamo dimostrare è che, preso un secondo albero AVL $T$ con lo stesso numero di nodi di $T'$, si abbia:

$$h \leq h'$$

Per assurdo, supponiamo che $h > h'$. Se esistesse un albero AVL T con altezza maggiore di $T'$, si potrebbe ottenere un albero AVL minimo $T''$ di altezza $h(T'') = h'$ e con meno nodi di $T'$ contraddicendo così la definizione di AVL minimo di $T'$.

Sappiamo, per ipotesi, che l'albero $T$ è più alto di $T'$. È possibile quindi effettuare un taglio da $T$ dei nodi che si trovano ad altezza maggiore di $h'$ ottenendo così un albero $T''$ di altezza $h'$ e con meno nodi di $T'$. Bisogna dimostrare che $T''$ è un albero AVL. Per farlo, è sufficiente dimostrare che $T''$ è un albero binario di ricerca e che la differenza tra l'altezza del sottoalbero sinistro e destro di ogni nodo è al più 1.

Supponendo per assurdo che $T''$ non sia AVL allora violerebbe la condizione di essere AVL: per ogni nodo, cioè, la differenza dei due sottoalberi è al più uno. Supponendo che esista un nodo $x$ che viola la proprietà AVL allora i suoi sottoalberi hanno delle altezze maggiori di 1 la cui differenza è maggiore di 1. Se esistesse questo nodo, però, una struttura del genere sarebbe esistita anche nell'albero $T$ in quanto appartenenti all'insieme dei nodi non tagliati. Quindi o $T$ non era un albero AVL o $T''$ è sicuramente un albero AVL. Quindi $T''$ è un albero AVL con la stessa altezza di $T'$ ma con meno nodi. Quindi si ottiene così la contraddizione ricercata.

■

### 3.3.4 Relazione tra altezza e numero di nodi

Sia $n$ il **numero minimo di nodi** di un albero AVL di altezza $h$. È possibile definire una funzione $N(h)$ che restituisce il numero di nodi per un albero AVL di altezza $h$.

| **h** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **N(h)** | 1 | 2 | 4 | 7 | 12 | 20 | 33 | 54 | 88 | 143 | 232 |

Tabella 3.1: Relazione tra altezza e numero di nodi in un albero AVL minimo

Per qualsiasi albero AVL minimo si ha

$$N(h) = \begin{cases} 1 & \text{se } h = 0 \\ 2 & \text{se } h = 1 \\ 1 + N(h-1) + N(h-2) & \text{se } h \geq 2 \end{cases}$$

che è molto simile alla funzione di Fibonacci:

$$Fib(x) = \begin{cases} 0 & \text{se } x = 0 \\ 1 & \text{se } x = 1 \\ Fib(x-1) + Fib(x-2) & \text{se } x \geq 2 \end{cases}$$

| **h** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Fib(h)** | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 |

Tabella 3.2: Funzione di Fibonacci

**Proposizione**

Si ha:

$$N(h) = Fib(h+3) - 1 \tag{3.4}$$

**Dimostrazione** Si dimostra per induzione:

- **Caso base:** Si ha $N(0) = Fib(3) - 1 = 1$ e $N(1) = Fib(4) - 1 = 2$

- **Ipotesi induttiva:** Supponiamo la relazione vera per $h-1$ e $h-2$, ovvero le relazioni:

$$N(h-1) = Fib(h-1+3) = Fib(h+2)$$
$$N(h-2) = Fib(h-2+3) = Fib(h+1)$$

Allora:

$$\begin{aligned} N(h) &= 1 + N(h-1) + N(h-2) \\ &= 1 + Fib(h+2) - 1 + Fib(h+1) - 1 \\ &= Fib(h+2) + Fib(h+1) - 1 \\ &= Fib(h+3) - 1 \end{aligned}$$

■

La forma chiusa del numero di Fibonacci è:

$$Fib(k) = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^h - \left(\frac{1-\sqrt{5}}{2}\right)^h\right]$$

che, per n sufficientemente grande, può essere approssimato a:

$$Fib(h) \cong \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^h$$

Quindi:

$$\sqrt{5}Fib(h) = \left(\frac{1+\sqrt{5}}{2}\right)^h \Longrightarrow h = \log_{\frac{1+\sqrt{5}}{2}}\left(\sqrt{5}Fib(h)\right)$$

Dunque, applicando l'equazione 3.4:

$$N(h) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+3} - 1$$

Di conseguenza l'altezza $h$ sarà logaritmica rispetto al numero di nodi. Infatti, indicato con $n$ il numero di nodi di un albero AVL minimo di altezza $h$ si ottiene:

$$\begin{aligned} n &= \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+3} - 1 \\ \sqrt{5}(n+1) &= \left(\frac{1+\sqrt{5}}{2}\right)^{h+3} \\ \Rightarrow h+3 &= \log_{\frac{1+\sqrt{5}}{2}}(\sqrt{5}(n+1)) \\ \Rightarrow h &= \log_{\frac{1+\sqrt{5}}{2}}(\sqrt{5}(n+1)) - 3 \\ &= \frac{\log_2(\sqrt{5}(n+1))}{\log_2\left(\frac{1+\sqrt{5}}{2}\right)} - 3 \\ &= \Theta(\log n) \end{aligned}$$

### 3.3.5 Implementazione di un albero AVL

Poiché la definizione di albero AVL è legata all'altezza, bisogna trovare un modo per reperire tale informazione per ciascun nodo dell'albero. Per calcolare l'altezza si potrebbe applicare un algoritmo ricorsivo. Infatti, un albero binario o risulta vuoto e la sua altezza sarà $-1$ oppure è composto da una radice dalla quale si radicano eventualmente due alberi binari:

$$h = 1 + max(T \rightarrow left,\ T \rightarrow right) \tag{3.5}$$

Si ottiene in questo modo l'algoritmo 3.14 per il calcolo dell'altezza di un albero binario.

```
1  if T = NIL then
2    return -1
3  else
4    return 1 + max(Altezza(T→left), Altezza(T→right))
```

Algoritmo 3.14: Altezza(T)

L'algoritmo 3.14 risulta troppo pesante da utilizzare ogni qualvolta si voglia effettuare un controllo del corretto bilanciamento in un albero AVL in quanto questo risulta lineare sul numero dei nodi dell'albero del quale calcola l'altezza. Per questo motivo può risultare più pratico inserire un nuovo campo, denominato $h$, all'interno di ciascun nodo, in modo tale da ridurre il tempo di "calcolo" dell'altezza. Si ottiene quindi un nodo come mostrato in Figura 3.11. Il calcolo dell'altezza diventa quindi un semplice accesso in lettura del campo $h$ del nodo. Si ottiene quindi una versione migliore dell'algoritmo ALTEZZA:

```
if T = NIL then
  return -1
else
  return T→h
```

Algoritmo 3.15: Altezza(T)

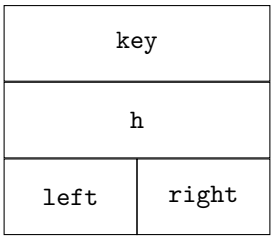



Figura 3.11: Nodo di un albero AVL

### 3.3.6 Inserimento in un albero AVL

Condideriamo un generico albero AVL $T$, che sappiamo essere anche binario di ricerca oltre che completo. Applicare l'algoritmo 3.6 per effettuare l'**inserimento di un nodo** all'interno di $T$ non è sufficiente di per sè per costruire un albero binario di ricerca che soddisfi la proprietà 3.2. Sarà necessario quindi modificare l'algoritmo introducendo un'algoritmo che effettui un controllo sul bilanciamento e, nel caso si verifichi uno sbilanciamento dell'albero, lo rimetta "a posto".

```
if T ≠ NIL then
  if T→ key < k then
    T→ right = Insert-AVL(T→ right, k)
    T = Bilancia-Destra(T)
  else if T→ key > k then
    T→ left = Insert-AVL(T→ left, k)
    T = Bilancia-Sinistra(T)
  else
    T = Alloca-Nodo-AVL()
    T→ key = k
    T→ left = T→ right = NIL
    T→ h = 0
return T
```

Algoritmo 3.16: Insert-AVL(T,k)

Per non aggravare il costo computazionale dell'algoritmo INSERT (Algoritmo 3.16) ci aspettiamo che le operazioni all'interno degli algoritmi di bilanciamento abbiano un costo costante $\Theta(1)$. Lo sbilanciamento in un albero AVL può essere risolto con una **sequenza di operazioni di bilanciamento**, anche dette **rotazioni**, che staranno alla base degli algoritmi BILANCIA-SINISTRA (Algoritmo 3.20) e BILANCIA-DESTRA.

### 3.3.7 Le rotazioni

Le rotazioni sono delle operazioni che permettono di ripristinare le proprietà di un albero bilanciato a fronte di operazioni che ne modificano la struttura. Queste sono di due tipi: **sinistra** e **destra**[2]. Oltre alle rotazioni singole a sinistra e a destra, è possibile effettuare anche delle **rotazioni doppie** come la rotazione doppia a sinistra, ovvero una rotazione a sinistra seguita da una rotazione a destra[3].

#### La rotazione sinistra

Consideriamo l'albero mostrato in Figura 3.12. Ciascun sottoalbero che si dirama dai nodi $x$ e $y$ hanno altezza $h$ in modo tale che il vertice $x$ non vede alcuno sbilanciamento, vale infatti:

$$||x \to left| - |x \to right|| = |(h+1) - h| = 1 \leq 1$$

Effettuando un inserimento di un nodo nel sottoalbero $A$ si ottiene una violazione in radice della proprietà 3.2 (Figura 3.13). Infatti, aggiungendo un nodo in $A$, si modifica la sua altezza che, da $h$, diventa $h+1$. Questo inserimento fa scendere l'albero radicato nel nodo $y$ ad una altezza $h+2$. Si ha così la violazione:

$$||x \to left| - |x \to right|| = |(h+2) - h| = 2 \not\leq 1$$

[2] Per destra e sinistra si intende il sottoalbero che scatena lo sbilanciamento in un nodo e non il senso della rotazione.
[3] La rotazione doppia a destra è duale

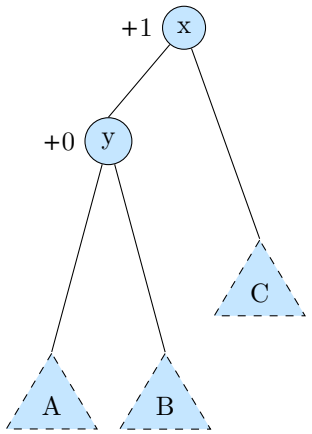



Figura 3.12: Albero AVL bilanciato: accanto a ciascun vertice è presente la differenza, in valore assoluto, tra le altezze dei sottoalberi.

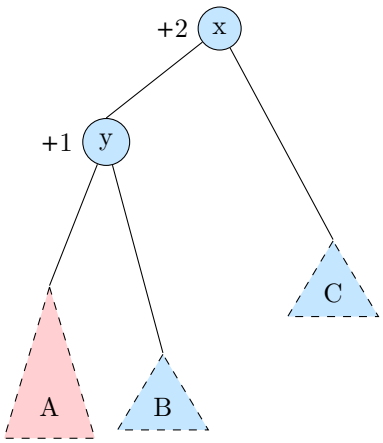



Figura 3.13: Albero AVL sbilanciato: il nodo $x$ vede aumentata la sua altezza

Come è possibile sistemare la violazione in questa situazione? Essendo stato il sottoalbero sinistro $A$ ad aver generato la violazione nel nodo $x$ si adotterà un'operazione di **rotazione singola sinistra** che avrà l'effetto di ruotare il nodo $x$ con il suo nodo figlio sinistro, ovvero il nodo $y$. L'operazione è molto semplice e consta dei seguenti passaggi:

1. Il nodo $y$, ovvero il figlio sinistro del nodo in cui si riscontra la violazione della proprietà 3.2, **sale in radice**;
2. Il nodo $x$ diventa **figlio destro** del nodo $y$;
3. Nel momento in cui $y$ ha come figlio destro il nodo $x$ e come figlio sinistro il vecchio albero $A$, si sposta l'albero $B$ come **figlio sinistro** del nodo $x$. È corretto fare questo spostamento in quanto tutti i nodi di $B$ continuano a stare a destra del nodo $y$ e a sinistra del nodo $x$.

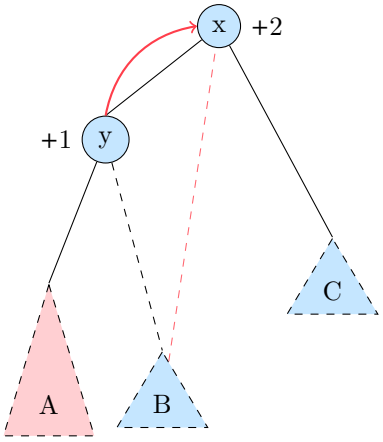



Figura 3.14: Rotazione singola sinistra

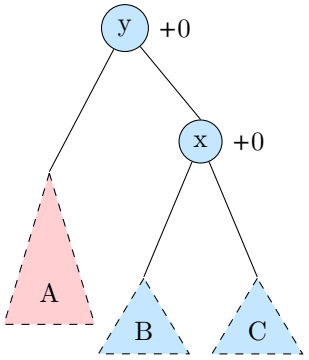



Figura 3.15: Albero risultante dopo la rotazione singola sinistra

In questo modo l'albero $A$ è salito di un livello mentre l'albero $C$ è sceso di un livello. Si osserva inoltre che, a fronte di un inserimento (che determina una rotazione), **l'operazione di ribilanciamento ripristina l'altezza dell'albero**. Infatti prima e dopo la rotazione si avrà:

$$T \to h = h + 2$$

Si ottiene così l'albero in Figura 3.15 che è ancora un albero AVL. Infatti:

$$||x \to left| - |x \to right|| = |(h+1) - (h+1)| = 0 \leq 1$$

L'algoritmo ROTAZIONE-SX(Algoritmo 3.17) prende in ingresso l'indirizzo della testa dell'albero e deve lavorare su due nodi: la radice che viene data in ingresso e la radice del sottoalbero sinistro che genera lo sbilanciamento. È facile vedere che l'algoritmo 3.17 è eseguibile a tempo costante.

```
1  newT = T→left
2  T→left = newT→right
3  newT→right = T
4  T→h = max(Altezza(T→left), Altezza(T→right))+1
5  newT→h = max(Altezza(newT→left), Altezza(newT→right))+1
6  return newT
```

Algoritmo 3.17: Rotazione-Sx(T)

L'operazione di rotazione singola sinistra non fa altro che sollevare di un livello il sottoalbero $A$ e far scendere $C$. Il sottoalbero $B$ resta allo stesso livello dopo l'operazione di rotazione. Quindi se l'inserimento venisse fatto in $B$ determinando un'incremento

dell'altezza di $B$ (sbilanciando di conseguenza il nodo $x$), l'operazione di rotazione singola sinistra lascerebbe immutato lo sbilanciamento.

Per questo motivo sarà necessario utilizzare una doppia operazione di bilanciamento che consiste di una prima rotazione a destra per portarci nel caso di uno sbilanciamento a sinistra e successivamente ripristinare l'albero con una rotazione a sinistra.

### La rotazione destra

La rotazione destra è duale alla rotazione sinistra. Si consideri l'albero AVL mostrato in Figura 3.16. In questo caso l'inserimento di un nodo nel sottoalbero $B$ del figlio destro della radice genera una violazione della proprietà 3.2 come si vede in Figura 3.17. Infatti:

$$||x \to left| - |x \to right|| = |(h) - (h+2)| = 2 \nleq 1$$

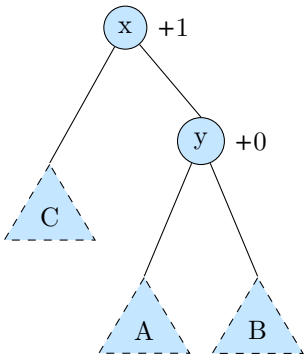



Figura 3.16: Albero AVL ben bilanciato

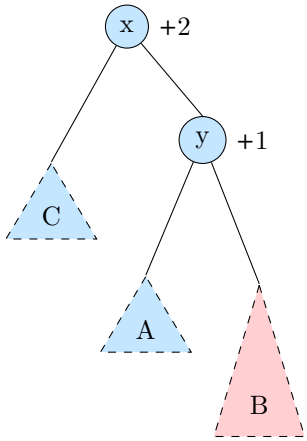



Figura 3.17: Albero AVL sbilanciato

In questo caso si applica una rotazione singola destra che ha l'effetto di far scendere il sottoalbero $C$ e far salire il sottoalbero $B$. L'altezza del sottoalbero $A$ rimane invariata come mostrato in Figura3.18.

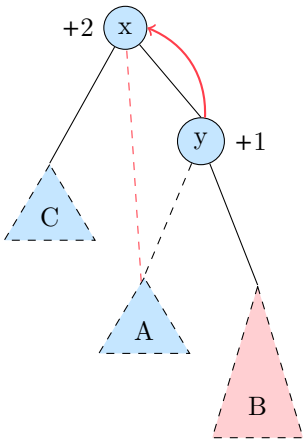



Figura 3.18: Rotazione singola destra

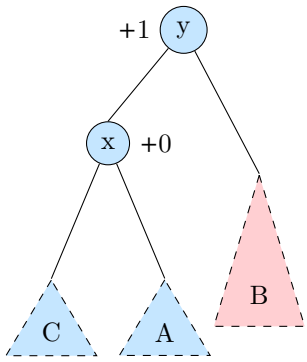



Figura 3.19: Albero risultante dopo la rotazione singola destra

L'algoritmo ROTAZIONE-DX (Algoritmo 3.18) è duale all'algoritmo ROTAZIONE-SX (Algoritmo 3.17).

```
1  newT = T→right
2  T→right = newT→left
3  newT→left = T
4  T→h = max(Altezza(T→left), Altezza(T→right))+1
5  newT→h = max(Altezza(newT→left), Altezza(newT→right))+1
6  return newT
```

Algoritmo 3.18: Rotazione-Dx(T)

### La doppia rotazione sinistra

Si consideri l'albero $T$ come mostrato in Figura 3.20. I sottoalberi $A$ e $D$ hanno altezza $h-2$ mentre i sottoalberi $B$ e $C$ hanno altezza $h-3$. L'albero radicato in $x$ risulta così ben bilanciato, infatti vale:

$$||x \to left| - |x \to right|| = |(h+1) - (h)| = +1 \leq 1$$

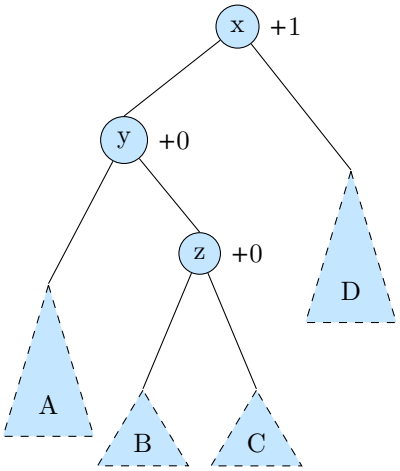



Figura 3.20: Albero AVL ben bilanciato

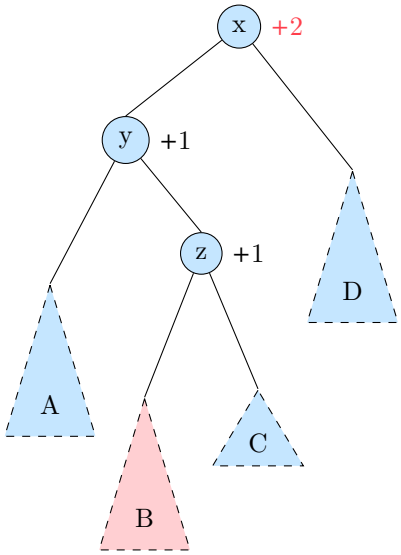



Figura 3.21: Inserimento nel sottoalbero $B$

Supponiamo di inserire un nodo nel sottoalbero $B$ radicato in $z$. Ovviamente, a seguito dell'inserimento l'altezza di tale sottoalbero risulterà incrementata causando uno sbilanciamento sul nodo $x$ dato dal suo sottoalbero sinistro come mostrato in Figura 3.21:

$$||x \to left| - |x \to right|| = |(h) - (h-2)| = +2 \not\leq 1$$

L'idea alla base della **doppia rotazione** è quella di *spostare il peso aggiuntivo* che si trova nel sottoalbero destro del nodo $y$ verso il suo sottoalbero sinistro in modo tale da ricondurci al caso di uno sbilanciamento a sinistra, risolvibile con una rotazione sinistra semplice. È necessario quindi effettuare una prima rotazione semplice destra che sposti tale peso verso il sottoalbero sinistro (vedi Figura 3.22) e successivamente una rotazione semplice sinistra che risolva il problema (Figura 3.24).

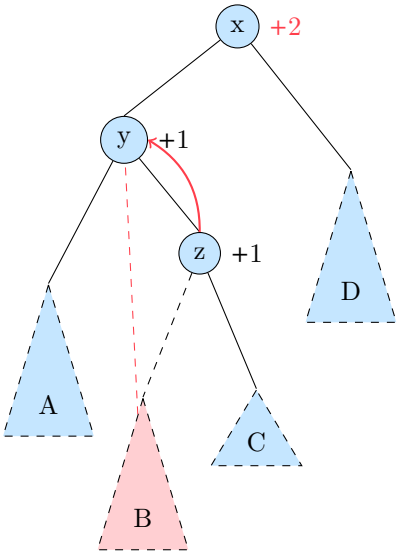



Figura 3.22: Prima rotazione semplice destra

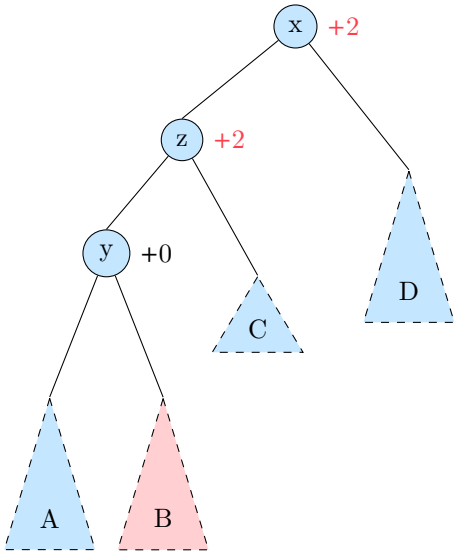



Figura 3.23: Situazione a seguito della rotazione singola destra

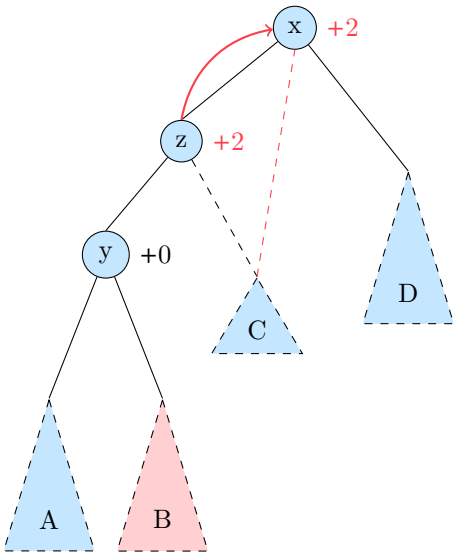



Figura 3.24: Rotazione singola sinistra

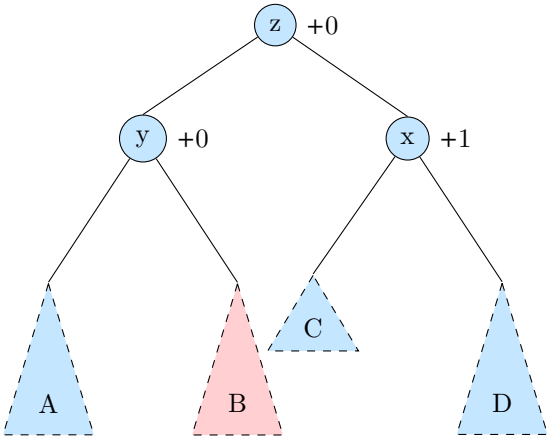



Figura 3.25: Albero risultante

Si ottiene così l'albero in Figura 3.25 che è ancora un albero AVL. Infatti:

$$||x \to left| - |x \to right|| = |(h-1) - (h-1)| = 1 \leq 1$$

```
1  T→left = Rotazione-Destra(T→left)
2  T = Rotazione-Sinistra(T)
3  return T
```

Algoritmo 3.19: ROTAZIONE-DOPPIA-SX

Date queste operazioni di rotazione possiamo scrivere dunque l'algoritmo BILANCIA-SX($T$) (Algoritmo 3.20). L'algoritmo BILANCIA-DX sarà duale.

```
1  if T≠ NIL then
2    if Altezza(T → left) - Altezza(T→ right) > 1 then
3      Sx = T→ left
4      if Altezza(Sx → left) ⩾ Altezza(Sx → right) then
5        T = Rotazione-Sinistra(T)
6      else
7        T = Rotazione-Doppia-Sinistra(T)
8    else
9      T→ h = max(Altezza(T→ left), Altezza(T→ right))+1
10 return T
```

Algoritmo 3.20: Bilancia-Sx(T)

**Osservazione**

A fronte di un inserimento in un albero AVL possiamo osservare che **possono avvenire al più due rotazioni**. Infatti, se un inserimento causa uno sbilanciamento, questo può essere risolto con una rotazione semplice o con una rotazione doppia. In entrambi i casi, l'altezza dell'albero viene incrementata di al più un livello. Se un inserimento non causa uno sbilanciamento, ciò significa che l'altezza dell'albero non è stata incrementata e non sarà necessaria alcuna rotazione.

### 3.3.8 Cancellazione in un albero AVL

Come per l'inserimento di un nodo all'interno di un albero AVL, anche l'operazione di cancellazione richiede un'operazione di *ribilanciamento* dell'albero in quanto, facendo diminuire l'altezza di un sottoalbero, non è garantito che sul padre, la differenza in altezza con il sottoalbero fratello rispetti la proprietà 3.2.

Analogamente a quanto visto nel caso della cancellazione in un albero binario di ricerca, anche per gli alberi AVL si discriminano tre casi: cancellazione di un nodo foglia, cancellazione di un nodo con un solo figlio e cancellazione di un nodo con due figli. In questo caso, però, non è detto che una singola operazione di ribilanciamento sia sufficiente a riportare l'albero in uno stato di bilanciamento. Infatti, la cancellazione di un nodo può causare uno sbilanciamento in un nodo diverso da quello cancellato. Per questo motivo, l'operazione di cancellazione prevede un'operazione di ribilanciamento dell'albero che potrebbe essere eseguita in modo ricorsivo fino a che non si raggiunge la radice dell'albero. Dovremo quindi effettuare delle modifiche degli algoritmi visti per la cancellazione in un albero binario di ricerca.

```
1  if T ≠ NIL then
2    if T → key > k then
3      T → left = Delete(T → left, k)
4      T = Bilancia-Destra(T)
5    else if T → key < k then
6      T → right = Delete(T → right, k)
7      T = Bilancia-Sx(T)
8    else
9      T = Delete-Root(T)
10 return T
```

Algoritmo 3.21: DELETE(T,k)

```
1  if T ≠ NIL then
2    tmp = T
3    if T→ left = NIL then
4      T = T→ right
5    else if T→ right = NIL then
6      T = T→ left
7    else
8      tmp = Stacca-Min(T→ right)
9      T→ key = tmp → key
10     T = Bilancia-Sx(T)
11   free(tmp)
12 return T
```

Algoritmo 3.22: DELETE-ROOT(T)

```
1  if T≠ NIL && P≠ NIL then
2    if T → left ≠ NIL then
3      ret = Stacca-MinAVL(T → left, T)
4      newT = Bilancia-Dx(T)
5    else
6      ret = T
7      newT = T → right
8  if T = P → left then
9    P → left = newT
10 else
11   P → right = newT
12 return ret
```

Algoritmo 3.23: Stacca-MinAVL(T,P)

Osserviamo un particolare caso di cancellazione: la cancellazione della foglia di minor altezza. Consideriamo l'albero AVL di altezza cinque mostrato in Figura 3.26. Supponiamo di voler cancellare la foglia più a destra ad altezza 2. La cancellazione di tale nodo causa uno sbilanciamento che si propaga fino alla radice. Si dimostra per induzione, che in un albero AVL, la foglia con altezza minore si trova sempre a metà altezza dell'albero. Ne segue che il numero di rotazioni necessarie per ri-bilanciare l'albero è lineare sull'altezza dell'albero, ovvero logaritmico sul numero dei nodi.

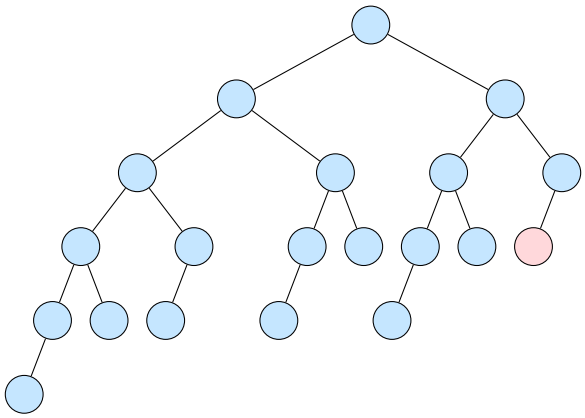


Figura 3.26: Cancellazione di un nodo foglia in un albero AVL di altezza cinque

## 3.4 ALBERI ROSSI E NERI

Nelle sezioni precedenti si è visto che un albero binario di ricerca di altezza $h$ può implementare qualsiasi operazione sugli insiemi dinamici in un tempo lineare sull'altezza dell'albero. Queste operazioni sono, quindi, veloci se l'altezza dell'albero resta contenuta. A fronte di una sequenza di operazioni di inserimento e cancellazione, l'altezza dell'albero può variare notevolmente inficiando le prestazioni delle operazioni stesse. Per questo motivo, si è cercato di definire una struttura dati che garantisca un'altezza contenuta dell'albero binario di ricerca. Così come gli alberi AVL, anche gli alberi rossi e neri rappresentano uno dei tanti modi[4] in cui gli alberi binari di ricerca vengono "bilanciati" per garantire che le operazioni elementari sugli insiemi dinamici richiedano un tempo logaritmico sul numero dei nodi.

[4] Splay Trees, B Alberi, ecc.

### 3.4.1 Proprietà degli alberi rosso-neri

Un **albero rosso-nero** è un albero binario di ricerca con un campo aggiuntivo di memoria per ogni nodo: il **colore** del nodo, che può essere RED o BLACK. Assegnando dei vincoli al modo in cui i nodi possono essere colorati, si garantisce che l'altezza dell'albero sia logaritmica sul numero dei nodi.

Ciascun nodo dell'albero contiene i campi: *color, key, left, right, h* (Figura 3.27). Il colore di ciascun nodo può essere espresso mediante una funzione di tipo booleana che ad ogni nodo assegna un bit a 0 o 1 a seconda che il colore sia rosso o nero. In questo modo, il colore di un nodo può essere rappresentato con un singolo bit di memoria a differenza del campo altezza che è rappresentato da un valore intero.

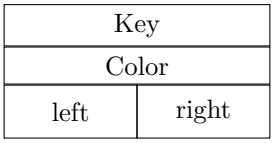



Figura 3.27: Implementazione di un nodo di un albero R&B

**Alberi R-B**

Un albero rosso e nero è un albero binario di ricerca che rispetta i seguenti vincoli:

1. Ogni nodo è colorato di rosso o di nero;
2. Ogni foglia è colorata di nero e non contiene dati. Queste foglie sono dette **foglie esterne** o **NIL**;
3. La radice è nera;
4. Ogni nodo rosso ha solo figli neri;
5. Per ogni nodo $x$, ogni percorso da $x$ a un nodo NIL contiene lo stesso numero di nodi neri. Questa quantità è detta **altezza nera** di $x$ e viene indicata con $bh(x)$.

La Figura 3.28 mostra un esempio di albero rosso-nero. In questo caso, il numero di nodi neri lungo ogni percorso da un nodo a una foglia esterna è sempre 3. È facile osservare che tutti gli alberi AVL possono essere alberi rossi e neri, secondo un'opportuna colorazione, ma non tutti gli alberi rossi e neri possono essere degli alberi AVL. Si consideri l'albero mostrato in Figura 3.29. Questo albero è un albero rosso nero in quanto rispetta ogni proprietà, ma non è un albero AVL in quanto la differenza di altezza tra il sottoalbero sinistro e quello destro del nodo $x$ è maggiore di 1. In Figura 3.31 è mostrata la gerarchia degli alberi binari di ricerca perfettamente bilanciati, AVL e rossi e neri.

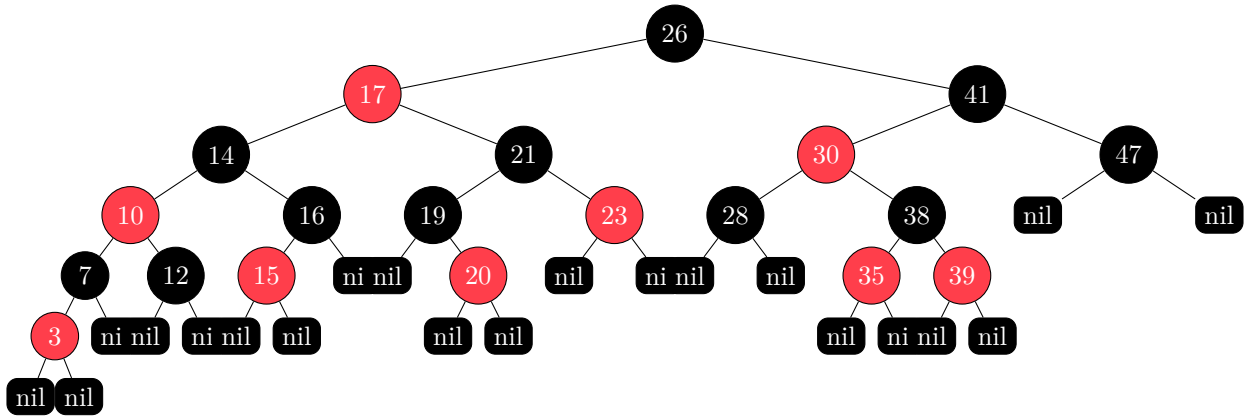



Figura 3.28: Albero rosso e nero

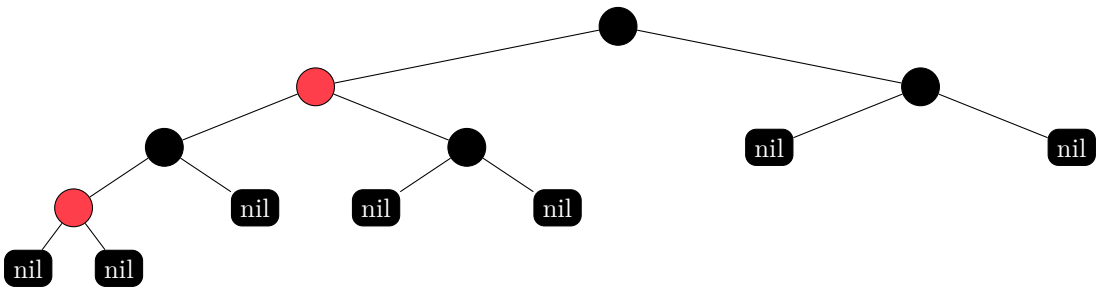
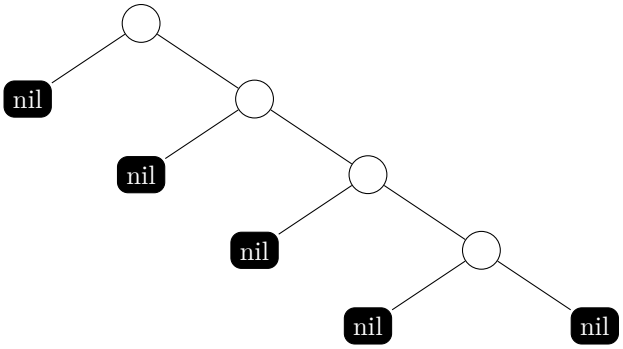



Figura 3.29: Albero rosso e nero che non è un albero AVL

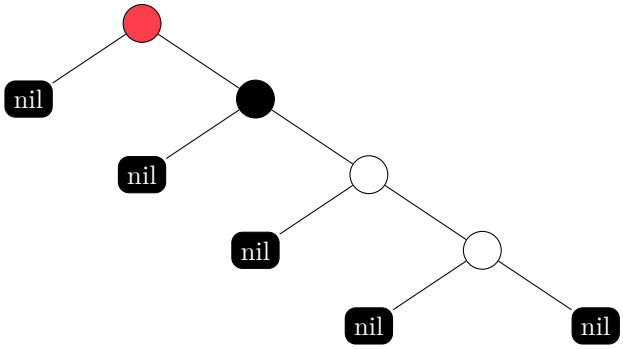
**Osservazione**

Come si evince dal diagramma mostrato in Figura 3.31 un albero pieno può essere colorabile come un albero rosso e nero. Un modo per colorarlo, infatti, è fare tutti i suoi nodi di nero. Questo perché gli alberi pieni hanno tutti i livelli saturi e di conseguenza tutti i percorsi da qualche nodo a una foglia avranno la stessa lunghezza. Un altro modo per colorarlo è alternando un livello rosso con un livello nero. Questo perché, essendo l'albero pieno, tutti i nodi di un livello hanno lo stesso numero di figli e, quindi, tutti i percorsi da qualche nodo a una foglia avranno la stessa lunghezza.

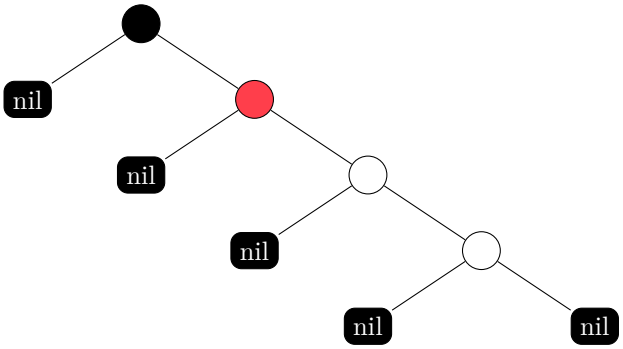
Non tutti gli alberi, però, sono colorabili. Sia ad esempio $T$ un albero binario come mostrato in Figura 3.30a. Da come si vede da questo esempio, nonostante i vincoli definiti per gli alberi rosso neri siano meno restrittivi di quanto visto per gli alberi AVL e APB, non tutti gli alberi binari di ricerca possono essere colorati come alberi rossi e neri. In particolare, un altro esempio di albero non colorabile è dato dalla classe degli alberi degeneri.



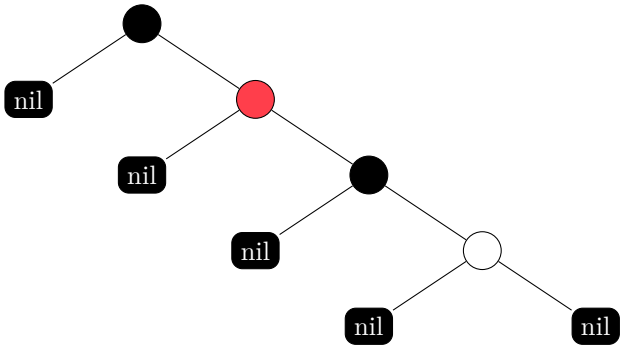

(a)




(b) Il colore della radice è ininfluente. Se è rossa, allora per il vincolo 3 il figlio dovrà essere nero. Così facendo notiamo però che il vincolo 4 non è rispettato. Il percorso che dalla radice arriva alla foglia ha un solo nodo nero, mentre il percorso che arriva alla foglia successiva ne ha due.




(c) Proviamo quindi a colorare la radice di nero. Il colore del figlio destro a questo punto non potrà essere che rosso se non si vuole squilibrare l'albero vioando il vincolo 5.




(d) Poiché si è colorato il secondo nodo di rosso, per il vincolo 4 il suo figlio dovrà essere nero. A questo punto però il vincolo 5 non è rispettato. Il percorso che dalla radice arriva alla foglia ha un solo nodo nero, mentre il percorso che arriva alla foglia successiva ne ha due.

Figura 3.30

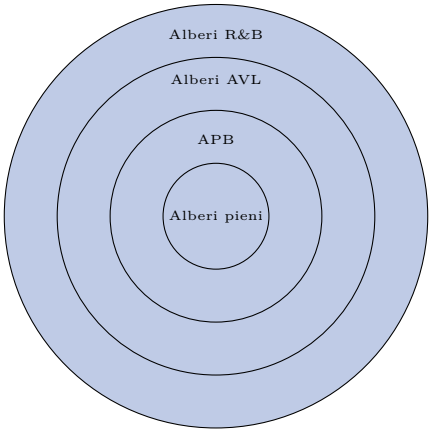



Figura 3.31: Relazione tra alberi perfettamente bilanciati, AVL e R&B

### 3.4.2 Altezza di un albero rosso-nero

I vincoli imposti sui nodi di un albero rosso-nero garantiscono che l'altezza dell'albero sia logaritmica sul numero dei nodi. In particolare, vincolare ogni nodo rosso ad avere solo figli neri e fissare l'altezza nera di ogni nodo a $bh(x)$, garantisce il contenimento dell'altezza dell'albero.

**Lemma**

Il numero di nodi interni di un sottoalbero radicato in $x$ è maggiore o uguale di $2^{bh(x)} - 1$.

**Dimostrazione** Si dimostra per induzione sull'altezza $h(x)$ dell'albero radicato in $x$.

**Caso base:** $h(x) = 0$. In questo caso, $x$ è una foglia esterna e, per definizione, non ha figli. Il numero di nodi interni è quindi 0 e $2^{bh(x)} - 1 = 2^0 - 1 = 0$.

**Passo induttivo:** $h(x) > 0$. In questo caso, $x$ è un nodo interno e ha due figli: $s$ e $d$. Qual è la loro altezza nera? Distinguiamo due casi:

- $s$ è rosso. Allora $bh(s) = bh(x) \geq bh(x) - 1$;
- $s$ è nero. Allora $bh(s) = bh(x) - 1 \geq bh(x) - 1$.

In maniera analoga abbiamo che $bh(d) \geq bh(x) - 1$. Il numero dei nodi interni radicato in $x$ sarà pari a:

$$1 + \#_{int}(s) + \#_{int}(d)$$

Poiché $h(s), h(d) < h(x)$ possiamo applicare l'ipotesi induttiva e ottenere:

$$\#_{int}(s) \geq 2^{bh(s)} - 1 \geq 2^{bh(x)-1} - 1$$

e

$$\#_{int}(d) \geq 2^{bh(d)} - 1 \geq 2^{bh(x)-1} - 1$$

Allora:

$$\begin{aligned} \#_{int}(x) &= 1 + \#_{int}(s) + \#_{int}(d) \\ &\geq 1 + 2 \cdot \left(2^{bh(x)-1} - 1\right) \\ &= 2^{bh(x)} - 1 \end{aligned}$$

■

**Teorema**

L'altezza massima di un albero rosso-nero con $n$ nodi è al più $2\log(n+1)$.

**Dimostrazione** Sia $h$ l'altezza dell'albero. Per il vincolo 5, in ogni percorso da un nodo ad una foglia almeno la metà dei nodi sono neri. Allora, l'altezza nera dell'albero dovrà essere almeno $h/2$. Per il Lemma precedente, il numero di nodi interni dell'albero è almeno $2^{bh(T)} - 1$. Allora:

$$\begin{aligned} n &\geq 2^{bh(T)} - 1 && \text{(per il Lemma precedente)} \\ &\geq 2^{h/2} - 1 && \text{(per il vincolo 5)} \\ n + 1 &\geq 2^{h/2} && \text{(spostando l'1 al primo membro)} \\ \log(n+1) &\geq \frac{h}{2} && \text{(applicando il logaritmo)} \\ 2\log(n+1) &\geq h && \text{(moltiplicando per 2)} \end{aligned}$$

In questo modo abbiamo trovato un limite superiore all'altezza dell'albero che dimostra il buon bilanciamento dell'albero.

■

**Corollario**

In un albero rosso nero le operazioni di ricerca, inserimento e cancellazione hanno un costo logaritmico sul numero dei nodi.

### 3.4.3 Inserimento in un albero rosso-nero

L'inserimento in un albero rosso-nero è simile all'inserimento in un albero binario di ricerca. Esattamente come per gli alberi binari di ricerca, l'algoritmo di inserimento di un nodo $z$ in un albero rosso-nero cerca un cammino discendente dalla radice dell'albero fino al nodo $y$ che ne diventerà padre.

Una volta identificato il nodo $y$, il nodo $x$ viene inserito come figlio sinistro di $y$ se $z \rightarrow key \leq y \rightarrow key$, destro altrimenti. Tuttavia, nel caso di alberi rosso-neri bisognerà risolvere il problema di mantenere i vincoli imposti sui nodi dell'albero. In particolare, l'inserimento di un nodo potrebbe violare il vincolo 5, che impone che tutti i cammini da un qualsiasi nodo alle foglie sue discendenti abbiano lo stesso numero di nodi neri. Per questo motivo, l'inserimento di un nodo implicherebbe una sua colorazione di rosso. Tuttavia, questa colorazione potrebbe violare il vincolo 4, che impone che ogni nodo rosso abbia solo figli neri.

In generale, l'inserimento di un nodo potrebbe richiedere una serie di rotazioni e ricolorazioni dei nodi dell'albero. L'idea alla base per il ripristino della proprietà red-black, se necessario, al ritorno dalle chiamate ricorsive presenti nell'Algoritmo 3.24 è la seguente: spostare le violazioni verso l'alto rispettando sempre il vincolo 5. Se la violazione arriva alla radice, la coloriamo di nero. Così facendo le operazioni di ripristino saranno necessarie solo quando due nodi consecutivi risultano rossi. In Figura 3.32 è mostrato un esempio di inserimento di un nodo in un albero rosso-nero.

```
1  if not IS-NIL(T) then
2    if T→ key < k then
3      T→ right = Insert-RB(T→ right,k)
4      T = Bilancia-Destra-RB(T)
5    else if T→ key > k then
6      T→ left = Insert-RB(T→ left,k)
7      T = Bilancia-Sinistra-RB(T)
8  else
9    T = NewNodeRB(k)
10   T→ color = RED
11 return T
```

Algoritmo 3.24: Insert-RB(T,k)

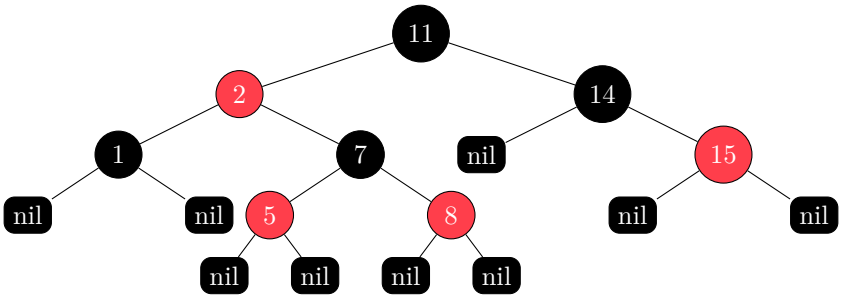



Figura 3.32: Albero dove inserire il nodo $z$

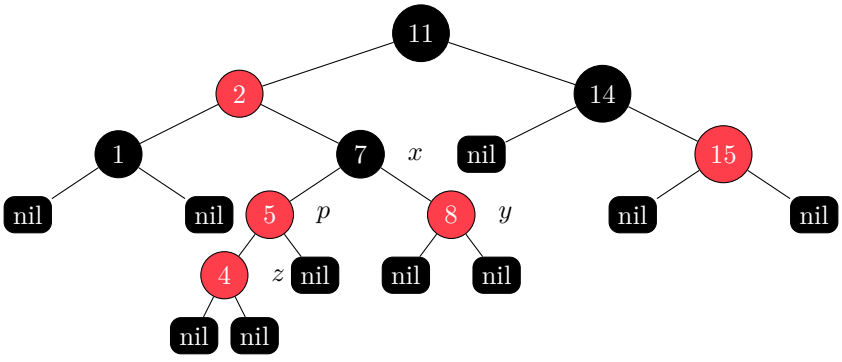



Figura 3.33: L'inserimento del nodo $z$ causa una violazione del vincolo 4. Per risolvere il problema, il nodo $y$ viene ricolorato di nero e il suo genitore $x$ di rosso.

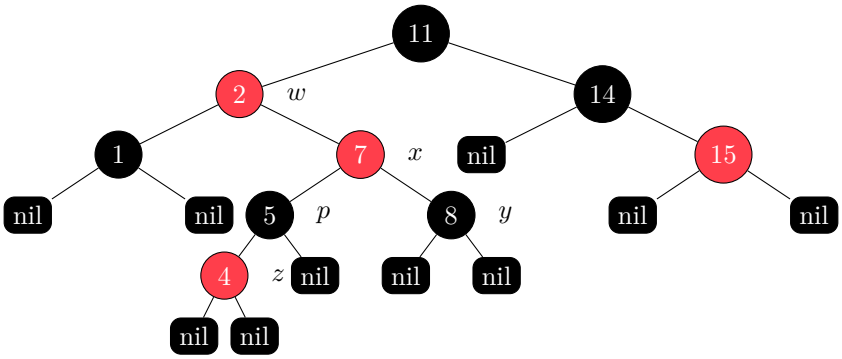



Figura 3.34: In questo caso si procede con la ricolorazione del nodo $p$ e $y$ da rossi a neri mentre il nodo $x$ diventa nero per non alterare il numero di nodi neri. A questo punto però si viola il vincolo 4 sul nodo $w$. Per risolvere lo sbilanciamento sarà quindi necessario cambiare il colore di $x$ e della radice violando così il vincolo 2 per evitare di violare il vincolo 5. Sarà quindi necessario procedere con una rotazione a destra per ripristinare il vincolo 2.

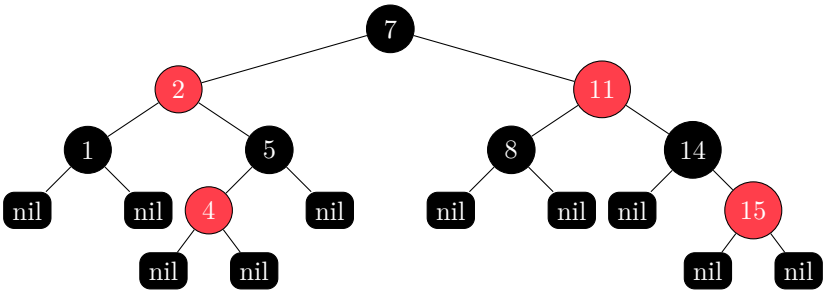



Figura 3.35: Albero finale

### Il bilanciamento del sottoalbero sinistro

Come detto in precedenza, le operazioni di ripristino sono necessarie solo quando due nodi consecutivi sono rossi. Tra l'altro, se la radice dell'albero è sempre nera, non si presenterà mai la necessità di ribilanciare in un albero (o sottoalbero) di altezza minore di 3 in quanto non si possono verificare violazioni. A fronte di queste osservazioni possiamo distinguere tre casi possibili che possono scatenare una violazione del vincolo 4 a fronte dell'inserimento di un nuovo nodo $z$:

1. Lo zio $y$ di $z$ è rosso (Figura 3.36a);
2. Lo zio $y$ di $z$ è nero e $z$ è il figlio destro del figlio sinistro del padre $p$ (Figura 3.36b);
3. Lo zio $y$ di $z$ è nero e $z$ è il figlio sinistro del figlio sinistro del padre $p$ (Figura 3.36c).

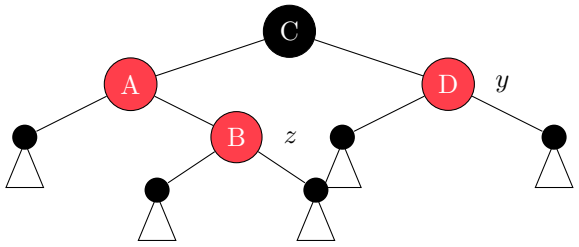
L'Algoritmo Bilancia-Sinistra-RB (Algoritmo 3.25) sulla base di questi tre casi, provvede a ripristinare la proprietà red-black dell'albero. Con al massimo due rotazioni si riesce a risolvere il problema di bilanciamento. Questo però non deve stupire in quanto gli alberi rosso-neri sono più tolleranti degli alberi AVL e quindi è normale che ci siano meno rotazioni. Il vantaggio computazionale degli alberi rosso-neri rispetto agli alberi AVL sarà ben visibile nella fase di cancellazione. L'Algoritmo Tipo-Violazione-Sinistra (Algoritmo 3.26) calcola il tipo di violazione che si è verificata.

```
1  if ha_figlio(T→left)
2    v = Tipo-Violazione_Sinistra(T→left,T→right)
3    case v of
4      1: T = Caso1(T)
5      2: T = Caso2(T)
6      3: T = Caso3(T)
7  return T
```

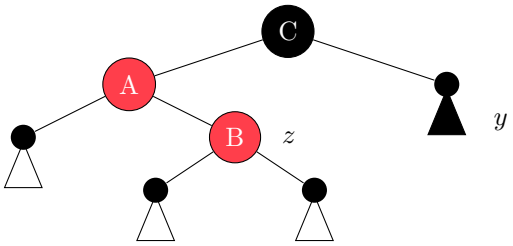
Algoritmo 3.25: Bilancia-Sinistra-RB(T)

```
1  v = 0
2  if S→color = RED && D→ color = RED then
3    if S→left→color = RED || S→right→color=R then
4      v = 1
5  else
6    if S→color = RED && S→right→color = RED then
7      v = 2
8    else
9    if S→left→color = RED then
10     v = 3
11 return v
```

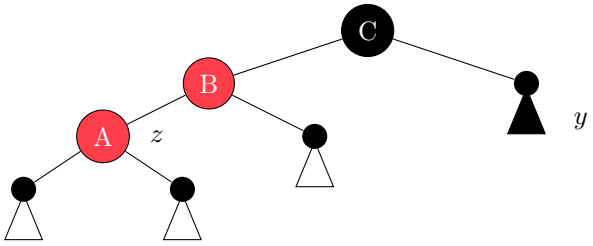
Algoritmo 3.26: Tipo-Violazione-Sinistra(S,D)




(a) Caso 1




(b) Caso 2




(c) Caso 3

Figura 3.36

```
1  T→right→color = BLACK
2  T→left→color = BLACK
3  T→color = RED
4  return T
```

Algoritmo 3.27: Caso1(T)

```
1  T→left = Rotazione-Sx(T→left)
2  T = Caso3(T)
3  return T
```

Algoritmo 3.28: Caso2(T)

```
1  T = Rotazione-Sx(T)
2  T→color = BLACK
3  T→right→color = RED
4  return T
```

Algoritmo 3.29: Caso3(T)

### 3.4.4 La cancellazione in un albero rosso-nero

Analogamente ad altre operazioni elementari su un albero rosso-nero di $n$ nodi, la cancellazione di un nodo richiede un tempo logaritmico sul numero di nodi. A differenza però degli altri algoritmi di cancellazione, la cancellazione in un albero rosso-nero è più complessa in quanto richiede maggiori controlli per mantenere i vincoli imposti.

**Osservazione**

Le operazioni di ripristino del bilanciamento sono necessarie solo quando il nodo cancellato è nero. Infatti, nel caso della cancellazione, non si può decidere a priori il colore del nodo da staccare. Qualora si dovesse eliminare un nodo nero siamo certi di aver modificato l'altezza nera dell'albero violando così il vincolo 5. Nel caso della cancellazione di un nodo rosso invece, si potrebbe al massimo aver avvicinato due nodi neri lungo un percorso ma la cosa non intacca alcun vincolo. Ripristinare questo vincolo però non è così complicato. L'idea alla base è quella di ignorare di aver violato il vincolo globale effettuando una **propagazione del colore nero** sul nodo che lo andrà a sostituire.

Se il nodo figlio era rosso allora è possibile colorarlo di nero, ma se questo fosse nero allora verrà colorato di un nuovo colore chiamato **doppio nero** (dal contributo che offre nel calcolo dell'altezza nera). In questo modo, il vincolo 5 verrà ripristinato. Tuttavia, questa introduzione di un terzo colore vìola il vincolo 1 che stabiliva che ogni nodo possa essere colorato o di rosso o di nero. Il problema per il ribilanciamento diventa quindi quello di eliminare i nodi doppio nero. L'algoritmo di cancellazione adotta una strategia simile a quella vista per la cancellazione negli alberi AVL.

```
if !nil(T) then
  if T→ key < k then
    T→ right = Delete-RB(T→ right,k)
    T = Bilancia-Canc-Destra-RB(T)
  else if T→ key > k then
    T→ left = Delete-RB(T→ left,k)
    T = Bilancia-Canc-Sinistra-RB(T)
  else
    T = Delete-Root-RB(T)
return T
```

Algoritmo 3.30: Delete-RB(T,k)

```
if T→ color= RED then
T→ color = BLACK
else
T→ color = DOUBLE_BLACK
```

Algoritmo 3.31: Propagate-Black(T)

```
if !nil(T) then
  tmp = T
  if nil(T→ left) then
    T = T → right
    if tmp → color = BLACK then
      Propagate-Black(T)
  else if nil(T→ right) then
    T = T→ left
    if tmp → color = BLACK then
      Propagate-Black(T)
  else
    tmp = Stacca-Min-RB(T→ right,T)
    T = Bilancia-Canc-Destra-RB(T)
  free(tmp)
return T
```

Algoritmo 3.32: Delete-Root-RB(T)

L'algoritmo Stacca-Min-RB è lo stesso visto per gli alberi AVL:

```
ret = nil
if !nil(P) && !nil(T) then
  if !nil(T→ left) then
    ret = Stacca-Min-RB(T→ left,T)
    newT = Bilancia-Canc-Sinistra-RB(T)
  else
    ret = T
    newT = T→ right
    if T → color = BLACK then
      Propagate-Black(newT)
    if P → left = T then
      P → left = newT
    else
      P → right = newT
return ret
```

Algoritmo 3.33: Stacca-Min-RB(T)

### Il bilanciamento del sottoalbero sinistro

Anche in questo caso, il bilanciamento del sottoalbero sinistro[5] è simile a quello visto per gli alberi AVL. In particolare, l'algoritmo Bilancia-Canc-Sinistra-RB (Codice 3.34) provvede a ripristinare la proprietà red-black dell'albero eseguendo rotazioni e cambiamenti di colore. I vincoli che si possono violare a seguito della cancellazione di un nodo sono i seguenti:

- **Violazione del vincolo 3:** la radice può essere un nodo rosso;
- **Violazione del vincolo 4:** se il padre e uno dei figli del nodo cancellato erano rossi;
- **Violazione del vincolo 5:** altezza nera cambiata.

I casi che possono scatenare una violazione sono invece 4:

1. Il colore del fratello è rosso (Figura 3.37a);
2. I nipoti sono neri (Figura 3.38a);
3. Il nipote sinistro è rosso e il nipote destro è nero (Figura 3.39a);
4. Il nipote destro è rosso (Figura 3.40a).

```
if !nil(T→ right) then
  v = Violazione_Sx(T→ left, T→right)
  case v of:
    1: T = Caso1(T)
       T = Bilancia-Canc-Sinistra(T→left)
    2: T = Caso2(T)
    3: T = Caso3(T)
    4: T = Caso4(T)
return T
```

Algoritmo 3.34: Bilancia-Canc-Sinistra-RB(T)

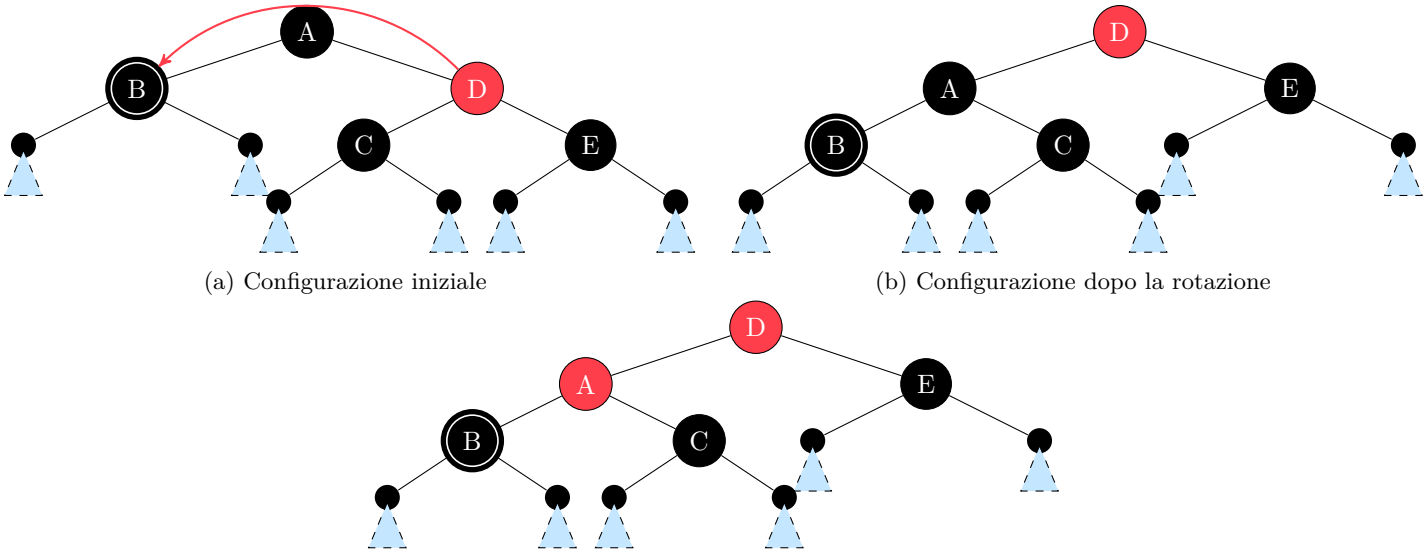
[5] Nel caso di sbilanciamento a destra si procede in modo simmetrico.

```
1  v = 0
2  if X→ color = DOUBLE-BLACK then
3  if W→ color = RED then
4  v = 1
5  else if W→ right → color = BLACK && W → left → color = BLACK
6  v = 2
7  else if W→ left → color = RED && W → right → color = BLACK
8  v = 3
9  else
10 v = 4
11 return v
```

Algoritmo 3.35: VIOLAZIONE_SX(X,W)

Per comprendere il caso che effettua lo sbilanciamento si effettua un controllo sugli alberi del sottoalbero destro del padre del nodo doppio nero. Si ottiene così l'Algoritmo VIOLAZIONE-SX (Codice 3.35). Analizziamo ora la gestione dei vari casi.

**Caso 1:** il fratello è rosso. In questo caso, il fratello può essere colorato di nero e il padre può essere colorato di rosso. Successivamente si applica una rotazione destra sul fratello. A questo punto di determina una violazione sul nodo $E$ in quanto vede diminuita la sua altezza nera. Per risolvere la violazione del sottoalbero destro si ricolora la radice $D$ di nero ritrovandoci così nella situazione di una violazione del vincolo 5 nel sottoalbero sinistro. Quest'ultima violazione si risolve facilmente colorando il nodo $A$ di rosso. A questo punto ci si ritrova con un sottoalbero sbilanciato a sinistra di $A$ che è risolvibile riconducendosi al caso 2, 3 o 4.



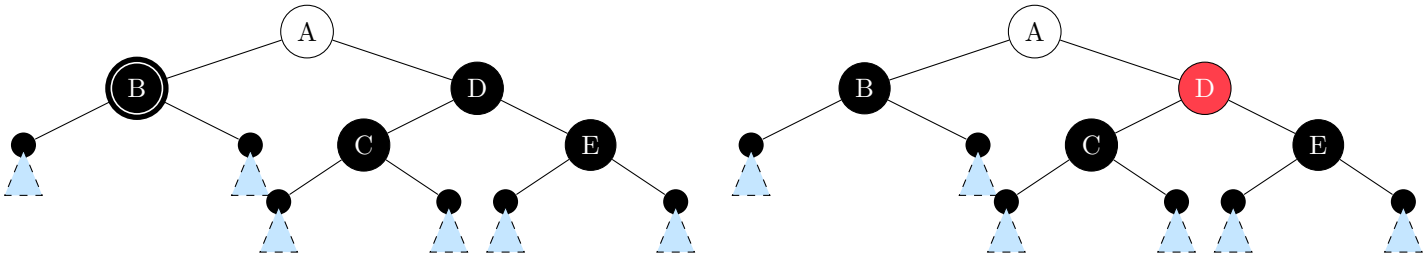

(a) Configurazione iniziale

(b) Configurazione dopo la rotazione

(c) Configurazione dopo la ricolorazione

Figura 3.37: Caso 1

**Caso 2:** il fratello è nero e i nipoti sono neri. In questo caso, il fratello può essere colorato di rosso e il doppio nero può essere propagato al padre. Ciò significa che il nodo $A$ diventerà doppio nero se prima già era nero, semplicemente nero altrimenti (vedi Figura 3.38b). Questo caso è il più semplice in quanto non richiede alcuna rotazione. Nel caso in cui il nodo $A$ sia diventato doppio nero sarà necessario effettuare un nuovo ribilanciamento. L'Algoritmo CASO2 (Codice 3.37) si occupa di gestire questa situazione.



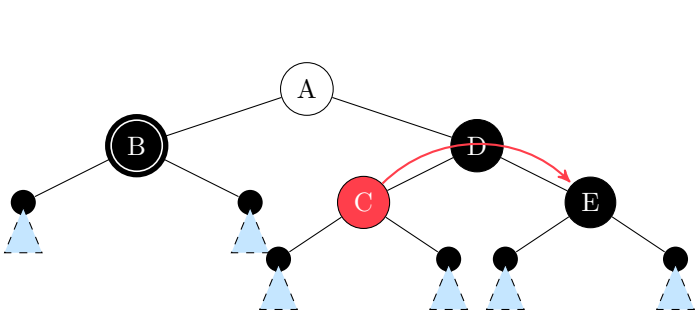

(a) Configurazione iniziale

(b) Ribilanciamento caso 2

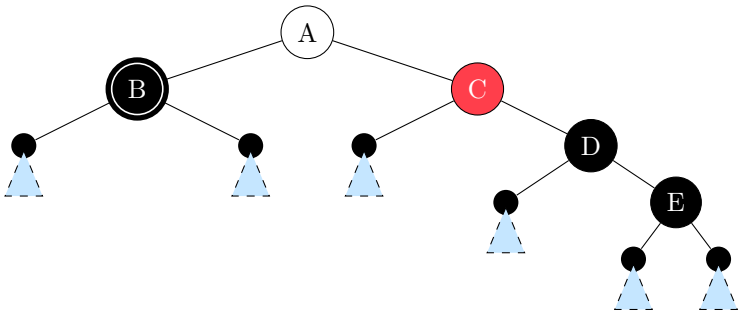
Figura 3.38: Caso 2

**Caso 3:** il fratello è nero, il nipote sinistro è rosso e il nipote destro è nero. In questo caso, ruotiamo il fratello con il suo figlio sinistro, cambiamo il colore del padre e quello del figlio destro. L'Algoritmo CASO3 (Codice 3.38) si occupa di gestire questa

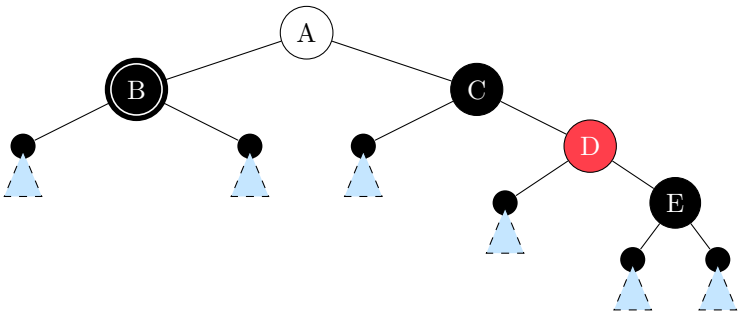
situazione. Alla fine di questa procedura il sottoalbero sinistro del nodo $C$ introduce una violazione di tipo 5 che si risolve ricolorando i nodi. Alla fine della procedura ci si riconduce al caso 4.




(a) Configurazione iniziale e senso della rotazione



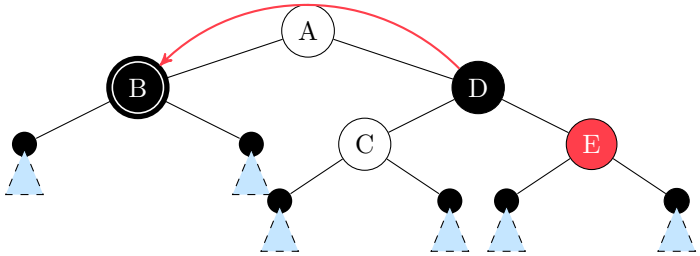

(b) Configurazione dopo la prima rotazione



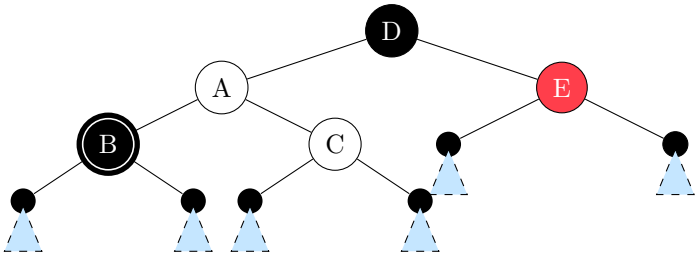

(c) Configurazione finale dopo la ricolorazione

Figura 3.39: Caso 3

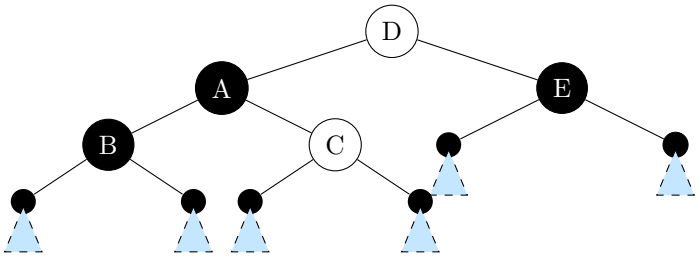
**Caso 4:** il fratello è nero e il nipote destro è rosso. In questo caso eseguiamo una rotazione del padre del nodo doppio nero con il fratello. Dopo la rotazione il nodo $B$ continua a pesare come un doppio nero, di conseguenza procediamo cambiando i colori opportunamente ed eliminando il nero in più sul nodo. L'Algoritmo Caso4 (Codice 3.39) si occupa di gestire questa situazione.




(a) Configurazione iniziale




(b) Configurazione dopo la rotazione




(c) Configurazione dopo la ricolorazione

Figura 3.40: Caso 4

```
1  T = Rotazione-Destra(T)
2  T→ color = BLACK
3  T→ right → color = RED
4  return T
```

Algoritmo 3.36: Caso1(T)

```
1  T rightarrow right → color = RED
2  T → left → color = BLACK
3  Propagate-Black(T)
4  return T
```

Algoritmo 3.37: Caso2(T)

```
T rightarrow right =  Rotazione-Sinistra(T
    → right)
T rightarrow right → color = BLACK
T → right → right → color = RED
T = Caso4(T)
return T
```

Algoritmo 3.38: Caso3(T)

```
T = Rotazione-Sinistra(T)
T → right → color = T → color
T → color = T → left → color
T → left → color = BLACK
T → left → left → color = BLACK
return T
```

Algoritmo 3.39: Caso4(T)

# Ricorsione ed iterazione

## 4.1 La ricorsione

In informatica il concetto di ricorsione è estremamente importante in quanto permette di ottenere spesso una descrizione chiara e concisa degli algoritmi. Diciamo che un algoritmo è *ricorsivo* se figurano in esso delle procedure ricorsive, ovvero procedure che invocano, direttamente o indirettamente, sé stesse. Come abbiamo visto fino a questo momento, questa tipologia di algoritmi costituiscono il metodo più naturale di risoluzione di un problema in quanto sono facili da comprendere e analizzare. Infatti:

- La correttezza di un programma ricorsivo si dimostra facilmente utilizzando il *principio di induzione*;
- Il calcolo della complessità temporale di un algoritmo ricorsivo si riduce alla soluzione delle equazioni di ricorrenza (come si vedrà nel Capitolo 6).

Affinché un algoritmo ricorsivo non continui a girare all'infinito deve avere le seguenti proprietà:

1. Devono esistere dei criteri, detti **criteri di base** (o anche *casi base*), per i quali la procedura non richiami sé stessa;
2. Ogni volta che la procedura chiama sé stessa, essa deve essere più vicina ai criteri di base.

Una procedura ricorsiva dotata di queste due proprietà si dice **ben definita**.

Il prodotto dei numeri interi positivi da 1 a $n$ compresi prende il nome di "fattoriale di $n$" e si indica con il simbolo $n!$:

$$n! = 1 \cdot 2 \cdot 3 \cdot \cdots (n-2) \cdot (n-1) \cdot n \tag{4.1}$$

Per definizione si pone $0! = 1$ quindi è possibile definire una funzione ben definita per tutti i numeri naturali. Si dimostra per induzione che:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n-1)! & \text{se } n > 0 \end{cases} \tag{4.2}$$

Questa definizione di $n!$ è ricorsiva, infatti quando si usa $(n-1)!$ si sta eseguendo una "chiamata" alla funzione stessa. Questa definizione è ben definita in quanto il valore $n!$ è esplicitamente assegnato quando $n = 0$ ed inoltre il valore $n!$ è definito in funzione di un valore più piccolo di $n$, più vicino quindi al valore di base 0. Per calcolare, ad esempio, 4! saranno necessari nove passaggi. Infatti:

1. $4! = 4 \cdot 3!$
2. $3! = 3 \cdot 2!$
3. $2! = 2 \cdot 1!$
4. $1! = 1 \cdot 0!$
5. $0! = 1$
6. $1! = 1 \cdot 1 = 1$
7. $2! = 2 \cdot 1 = 2$
8. $3! = 3 \cdot 2 = 6$
9. $4! = 4 \cdot 6 = 24$

Vediamo ora una procedura che calcola il fattoriale:

```
1 if N = 0 then
2   fact = 1
3 else
4   x = factorial_rec(n-1)
5   fact = N * x
6 return fact;
```

Algoritmo 4.1: FACTORIAL_REC(N)

Non tutti i linguaggi di programmazione, però, supportano la ricorsione[1]. La ricorsione, infatti, non è un meccanismo proprio del calcolatore il quale, come si è detto quando si è parlato di macchine astratte (vedi Sezione 1.1.1), ha un insieme limitato di operazioni ed opera mediante assegnazioni e salti[2].

I linguaggi che permettono la ricorsione utilizzano in fase di esecuzione uno **stack** per tenere traccia della sequenza di chiamata delle varie procedure ricorsive. In testa a tale stack è sempre presente il *record di attivazione* contenente il **contesto** della procedura correntemente attiva. Tale contesto comprende:

- le variabili locali alla procedura;
- i parametri ad essa passati;
- l'indirizzo di ritorno, ovvero il contenuto del *program counter* nel momento in cui la procedura è stata invocata.

Quando una procedura è invocata viene posto sulla pila un nuovo record di attivazione. Ciò indipendentemente dal fatto che siano già presenti nella pila altri record di attivazione relativi alla stessa procedura. Per questo motivo la ricorsione è molto dispendiosa in termini di memoria. Più in avanti vedremo alcuni esempi di algoritmi ricorsivi che non richiedono l'utilizzo di uno stack.

## 4.2 LA TRADUZIONE DA RICORSIVO AD ITERATIVO

### 4.2.1 La memoria di lavoro

Come abbiamo visto, la ricorsione è un meccanismo che non è proprio del calcolatore. Per questo motivo, quando si scrive un programma ricorsivo, il compilatore deve tradurlo in un programma iterativo. Alla base di questa traduzione c'è il concetto di **memoria di lavoro** che è una memoria ausiliaria che viene utilizzata per salvare i dati necessari al calcolo di una procedura ricorsiva. In particolare, la memoria di lavoro è costituita da uno **stack** che viene utilizzato per salvare i **parametri formali** delle varie chiamate ricorsive. Ogni volta che è necessario eseguire una sottochiamata ricorsiva ci si avvale della memoria di lavoro per salvare i parametri formali della procedura corrente per poterli poi recuperare al ritorno dalla chiamata ricorsiva. In questo modo, la memoria di lavoro permette di simulare la ricorsione mediante iterazione.

### 4.2.2 Uno schema generale per la traduzione

#### Il ciclo while

La traduzione di un algoritmo ricorsivo in un algoritmo iterativo richiede un approccio basato su un grande ciclo `while` il quale gestisce al suo interno le due fasi principali di una chiamata ricorsiva: l'**inizializzazione** di una nuova chiamata ricorsiva e il **ripristino** dei dati precedentemente utilizzati al *ritorno* di una queste ultime. Per capire quale fase bisogna gestire ci avvaliamo di una variabile booleana `start`[3] che assume valore `true` quando siamo nella fase di *inizializzazione* di una nuova chiamata ricorsiva e `false` quando siamo nella fase di *ripresa* di una chiamata sospesa.

Oltre alla condizione sulla variabile `start`, è necessario controllare anche lo **stato dello stack**. Infatti, se lo stack è vuoto significa che *non ci sono chiamate ricorsive sospese* e quindi non è necessario effettuare alcuna ripresa e si può procedere fino alla fine dell'algoritmo. In caso contrario sarà necessario *ripristinare il contesto della chiamata ricorsiva sospesa* per eseguire le varie sottochiamate e proseguire con l'esecuzione dell'algoritmo.

#### Il punto di ritorno: la variabile last

Al ritorno da una chiamata ricorsiva bisogna cercare di capire il punto dal quale riprendere l'esecuzione della chiamata precedentemente sospesa. Per fare ciò è necessario individuare il cosiddetto **punto di ritorno**, ovvero l'istruzione che segue immediatamente la chiamata ricorsiva appena terminata. In generale, un modo per discriminare il punto di ritorno a fronte di più chiamate ricorsive è quello di utilizzare una variabile `last` la quale, *ogni volta che si termina una chiamata ricorsiva*, viene aggiornata con il valore del parametro attuale che sarà sicuramente diverso dai corrispettivi delle altre sottochiamate ricorsive (in caso contrario, infatti, si avrebbero chiamate ridondanti e si finirebbe in un loop).

[1] Pascal e Cobol ad esempio non la permettono.
[2] Infatti in un calcolatore non è neanche definito il concetto di ciclo ed iterazione.
[3] Negli esempi presenti nel capitolo capiterà di sostituire tale variabile con una vera e propria condizione sui parametri attuali.

È possibile quindi delineare uno schema generale per la traduzione di un algoritmo ricorsivo in uno puramente iterativo come mostrato in Figura 4.1.

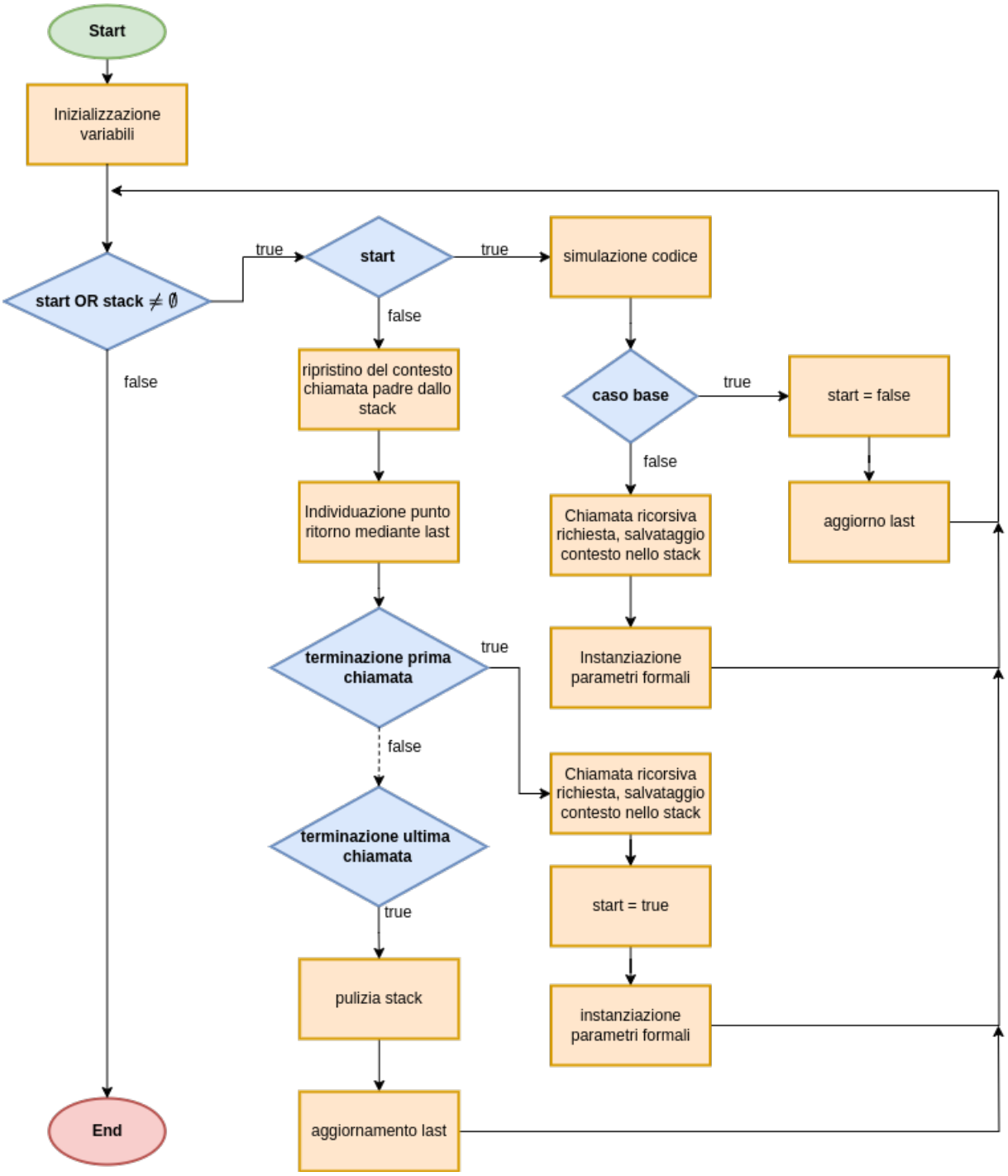



Figura 4.1: Schema algoritmo iterativo che simula algoritmo ricorsivo

### 4.2.3 La funzione fattoriale

Osservando l'algoritmo 4.1 notiamo che si effettuano nuove chiamate ricorsive ogni qualvolta la variabile `N` è maggiore o uguale a zero. Per questo motivo possiamo pensare di porre come condizione di loop nel ciclo `while` il controllo `cn ≥ n`.

Per poter ripristinare il contesto della chiamata padre, è necessario salvare in memoria i parametri formali della procedura ricorsiva. Nel caso della funzione fattoriale, per simulare l'inizio di una chiamata ricorsiva, è necessario verificare se il parametro `N` è uguale a `0` e, in caso affermativo, assegnare il valore `1` alla variabile `ret`.

Nel caso in cui `N` sia diverso da `0` è necessario eseguire una nuova chiamata ricorsiva. Per permetterlo è necessario innanzitutto salvare nello stack il valore corrente di `N` e assegnare a `cn` (*current n*) il valore `N-1` (**instanziazione parametri formali**). Sarà quindi necessario un unico stack per salvare il valore di `N`. Non è necessario salvare il valore di `fact` in quanto questo viene calcolato al ritorno dalla chiamata ricorsiva.

Chiaramente, una volta arrivati al caso base il valore `cn` sarà uguale a `-1` ma lo stack non sarà vuoto e sarà necessario effettuare nuove iterazioni per svuotarlo per ottenere il valore finale. Si ha così un primo esempio di traduzione di un algoritmo ricorsivo

in un algoritmo iterativo:

```
1  cn = n
2  stack = NIL
3  while (cn ⩾ 0 || stack ≠ NIL) do
4    if cn ⩾ 0 then
5      // caso base
6      if cn = 0 then
7        r = 1
8        ret = r
9        cn = -1
10     // avvio sottochiamata
11     else
12       // salvataggio contesto
13       stack = push(stack, cn)
14       // aggiorno il parametro per la sottochiamata
15       cn = cn - 1
16   // ritorno da sottochiamata
17   else
18     // ripristino contesto
19     cn = top(stack)
20     x = ret
21     r = cn * x
22     stack = pop(stack)
23     ret = r
24     cn = -1
25 return ret
```

Algoritmo 4.2: Traduzione iterativa della funzione fattoriale

### 4.2.4 L'esempio del MergeSort

Consideriamo l'algoritmo di ordinamento `Merge Sort`. Questo è un algoritmo di ordinamento ricorsivo basato sul concetto del "divide et impera". La sua idea principale consiste nel dividere un array non ordinato in due metà, ordinare ciascuna metà separatamente, e infine combinare le due metà ordinate in un unico array ordinato. L'algoritmo opera ricorsivamente fino a quando l'array non può essere più diviso, e poi procede a combinare le parti ordinate.

```
1  if p<r then
2    q = (p+r)/2
3    MergeSort(A,p,q)
4    MergeSort(A,q+1,r)
5    Merge(A,p,q,r)
```

Algoritmo 4.3: MergeSort(A,p,r)

Notiamo fin da subito che l'algoritmo `MergeSort` richiama se stesso due volte. Quando un compilatore traduce un programma scritto in un linguaggio di alto livello in un linguaggio di basso livello, come il linguaggio macchina, non può utilizzare la ricorsione in quanto non esiste in esso. Il compilatore deve quindi tradurre la ricorsione in iterazione.

Nell'algoritmo 4.3 compaiono quattro variabili: `p, r, q, A`. Durante la fase di traduzione in iterazione, notiamo però che solo le variabili `p`, `q` ed `r` vengono modificate. La variabile `A` rappresenta un puntatore alla prima cella dell'array da ordinare e che quindi non cambia mai mentre la variabile `q` viene calcolata in ogni sottochiamata dati i valori correnti di `p` ed `r` (salvati nelle variabili `cp` e `cr`), per questo motivo non sarebbe necessario memorizzare la variabile `q` ma ciò nonostante utilizzeremo tre stack, uno per ogni variabile `p, q, r`.

Per discriminare il punto di ritorno, avendo a che fare con due chiamate ricorsive, utilizzeremo la variabile `last` che verrà aggiornata ogni volta che si termina una chiamata ricorsiva. Infatti, ciascuna sottochiamata assume parametri formali diversi e ogni volta che termina deve comunicare al padre il valore del suo terzo parametro per poter proseguire con l'esecuzione. Per questo motivo, la variabile `last` conterrà il valore della variabile `cr`. Al ritorno da ciascuna sottochiamata, basterà quindi controllare se il valore di `last` è uguale al valore di `cr` per capire se si è terminata la prima o la seconda sottochiamata. Se, al termine di una iterazione, `last` è uguale a `q` allora sicuramente si sta tornando dalla terminazione della prima chiamata ricorsiva. Se invece `last` è uguale a `cr` allora non sarà necessario effettuare nuove chiamate ricorsive e si può procedere fino alla fine dell'algoritmo.

```
1  cp = p
2  cr = r
3  stackR = NIL
4  stackP = NIL
5  stackQ = NIL
6  start = true
7  while (start || stackR ≠ NIL) do
```

```
  if start then
    if cp < cr then
      // simulazione codice fino alla prima chiamata ricorsiva
      q = (cp+cr)/2
      // avvio prima sottochiamata, salvataggio contesto
      stackR = push(stackR, cr)
      stackP = push(stackP, cp)
      stackQ = push(stackQ, q)
      // aggiornamento parametri formali
      cr = q
    // simulazione terminazione, la chiamata padre potrebbe essere stata sospesa
    else
      last = cr
      start = false
  else
    // ripristino contesto
    cr = top(stackR)
    cp = top(stackP)
    q = top(stackQ)
    // individuazione punto di ritorno
    if last ≠ cr then
      cp = q + 1
      start = true
    else
    // fine della seconda sottochiamata, terminazione chiamata corrente e pulizia record
      Merge(A,cp,q,cr)
      stackR = pop(stackR)
      stackP = pop(stackP)
      stackQ = pop(stackQ)
      last = cr
```

Algoritmo 4.4: MergeSort_Iter(A,p,r)

### 4.2.5 La funzione di Fibonacci

Sia $Fib(n)$ la funzione che restituisce l'$n$-esimo numero della successione di Fibonacci. Tale valore viene calcolato come segue:

$$Fib(n) = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ Fib(n-1) + Fib(n-1) & \text{se } n > 1 \end{cases} \tag{4.3}$$

Possiamo, quindi, definire un algoritmo ricorsivo che segua in maniera diretta l'equazione 4.3.

```
if n ≤ 1 then
  r = n
else
  x = Fib(n-1)
  r = x+ Fib(n-2)
return r
```

Algoritmo 4.5: Fib(n)

Per eseguire la traduzione dell'algoritmo 4.5 in uno iterativo, è necessario utilizzare uno stack per salvare i valori precedentemente calcolati e che verranno utilizzati per calcolare il valore finale. L'algoritmo richiama se stesso due volte, per questo motivo utilizzeremo la variabile `last` per discriminare il punto di ritorno, se `last` è diverso da `cn - 2` allora si sta tornando dalla prima chiamata ricorsiva, altrimenti si sta tornando dalla seconda chiamata ricorsiva.

Si ha quindi:

```
cn = n
stack = NIL
stack_X = NIL
start = true
while (start || stack ≠ NIL) do
  if start then
    // Controllo del caso base
    if cn ⩽ 1 then
      r = cn
      ret = r
      last = cn
```

```
12        start = false
13      else
14        // Salvataggio del contesto
15        stack = push(stack, cn)
16        cn = cn - 1
17    else
18      // Ripristino del contesto
19      cn = top(stack)
20      // Individuazione del punto di ritorno
21      if last ≠ cn - 2 then
22        // Operazioni da eseguire al termine della prima sottochiamata
23        x = ret
24        stack_X = push(stack_X,x)
25        cn = cn - 2
26        start = true
27      else
28        // Operazioni da eseguire al termine della seconda sottochiamata
29        x = top(stack_X)
30        r = x + ret
31        // Ripulisco lo stack
32        stack = pop(stack)
33        // Aggiornamento last
34        ret = r
35        last = cn
36  return r
```

Algoritmo 4.6: FIB_ITER(n)

### 4.2.6 Un algoritmo generico

Consideriamo l'algoritmo 4.7:

```
1   ret = 0
2   z = 0
3   if (p ⩽ r) then
4     q = (p+r)/2
5     if (k = A[q]) then
6       z = A[q]
7     ret = z + Algo(A,q+1,r,k)
8     if (ret > 0) then
9       ret = ret + Algo(A,p,q-1,k)
10  return ret
```

Algoritmo 4.7: ALGO(A,p,r,k)

Per tradurre l'algoritmo 4.7 in uno iterativo è bene osservare che qualsiasi variabile che viene scritta prima e letta dopo una chiamata ricorsiva deve essere salvata all'interno di uno stack apposito. Per questo motivo si useranno tre stack per salvare i valori di `p`, `q` e `ret`. Inoltre, è necessario utilizzare la variabile `last` per discriminare il punto di ritorno. Infatti, l'algoritmo richiama se stesso due volte, per questo motivo è necessario utilizzare la variabile `last` per capire se si sta tornando dalla prima o dalla seconda chiamata ricorsiva. In questo caso, il valore che viene sempre modificato ad ogni chiamata ricorsiva e che contraddistingue ciascuna chiamata è `p`. Per questo motivo, la variabile `last` conterrà il valore di `cp`. Si ha quindi:

```
1   start = true
2   cp = p
3   cr = r
4   stackP = stackQ = stackRet = NIL
5   while (start || stackP ≠ NIL) do
6     if (start) then
7       // Istruzioni fino alla prima chiamata ricorsiva
8       ret = 0
9       z = 0
10      if (cp ⩽ cr) then
11        q = (cp+cr)/2
12        if (k = A[k]) then
13          z = A[k]
14        // Salvataggio del contesto
15        stackP = push(stackP,cp)
16        stackQ = push(stackQ,q)
17        cp = q+1
18      else
19        // Simulazione terminazione chiamata
```

```
20        result = ret
21        last = cp
22        start = false
23    else
24      // Ripristino contesto
25      cp = top(stackP)
26      q = top(stackQ)
27      // Individuazione punto di ritorno
28      if (cp ≠ last) then
29        // Impostazione del valore della variabile z
30        if(k=A[q]) then
31          z =A[q]
32        else
33          z = 0
34        // Prima istruzione successiva al ritorno dalla prima chiamata
35        ret = z + result
36        if(ret>0) then
37          // Inizializzazione seconda chiamata, salvataggio del contesto
38          stackRet = push(stackRet, ret)
39          cr = q-1
40          start = true
41        else
42          // Terminazione chiamata
43          result = ret
44          last = cp
45          // Pulizia stack
46          stackP=pop(stackP)
47          stackQ=pop(stackQ)
48      else
49        // Ritorno dalla seconda chiamata
50        ret = top(stackRet)
51        ret = ret + result
52        // Terminazione chiamata
53        result = ret
54        last = cp
55        // Pulizia stack
56        stackP = pop(stackP)
57        stackQ = pop(stackQ)
58        stackRet = pop(stackRet)
59  return ret
```

Algoritmo 4.8: ALGO_ITER(A,p,r,k)

## 4.3 GLI ALGORITMI RICORSIVI NEGLI ALBERI BINARI

Quando abbiamo visto lo schema generale per la traduzione di un algoritmo ricorsivo si è visto come discriminare tra le due fasi di un algoritmo ricorsivo. Ricapitolando:

- L'inizializzazione di nuova chiamata ricorsiva avviene quando la condizione `start` risulta vera;
- La ripresa del controllo dopo la terminazione di una sottochiamata avviene invece quando `start = false` e quando ci sono ancora elementi all'interno dei vari stack ( `stack` $\neq \emptyset$).

Tra le varie assunzioni fatte, inoltre, si è adoperata una variabile `last` aggiornata al valore del parametro attuale modificato da restituire alla chiamata padre per distinguere il punto di ritorno. In questa sezione osserveremo però che *questo meccanismo non funziona nel caso degli algoritmi che operano sugli alberi binari.*

### 4.3.1 L'algoritmo Search

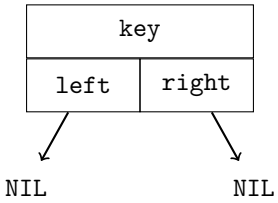
Consideriamo un algoritmo di ricerca in un albero binario generico.

```
1  if T≠ NIL then
2    if T→ key ≠ k then
3      ret = Search(T→ left,k)
4      if ret ≠ NIL then
5        ret = Search(T → right,k)
6  return ret
```

Algoritmo 4.9: SEARCH(T,k)

In questo caso, l'unico parametro che viene modificato nel corso delle varie sottochiamate ricorsive è rappresentato dalla testa del sottoalbero sul quale viene eseguita la ricerca. Non ha senso inizializzare uno stack per la variabile `ret` in quanto questa viene sempre calcolata ad ogni sottochiamata ed il vecchio valore non viene più utilizzato.

Come già anticipato, non è possibile utilizzare la strategia adottata finora per l'individuazione del punto di ritorno. Infatti, quando viene effettuata una chiamata su un nodo foglia si avrà che sia `last = ct = T → left` che `last = ct = T → right` sono uguali a `NIL`, quindi al ritorno dalle sottochiamate ricorsive sarebbe impossibile discriminare quale delle due chiamate si è appena conclusa.




Per risolvere tale problema, osserviamo che la prima sottochiamata viene effettuata sempre sul sottoalbero sinistro quindi, se ad un certo punto dell'iterazione si vede che `last` è uguale a `NIL` vorrà dire che la chiamata appena terminata è quella di sinistra. Si ha quindi:

```
1  ct = T
2  start = true
3  stackT = NIL
4  while (start || stackT ≠ NIL) do
5    ret = ct
6    if (start) then
7      if (ct → key ≠ k) then
8        // Salvataggio contesto
9        stackT = push(stackT, ct)
10       ct = ct → left
11     else
12       result = ret
13       start = false
14       last = ct
15     else
16       // Ripristino contesto
17       ct = top(stackT)
18       // Individuazione punto di ritorno
19       if (last = ct → left && ct → right ≠ NIL) then
20         ret = result
21         if (ret = NIL) then
22           ct = ct → right
23           start = true
24         else
25           stackT = pop(stackT)
26           result = ret
27           last = ct
28       else
29         stackT = pop(stackT)
30         result = ret
31         last = ct
32 return ret
```

Algoritmo 4.10: SEARCH_ITER(T,k)

### 4.3.2 L'algoritmo PrintTree

Consideriamo adesso l'algoritmo che si occupa di stampare in post-ordine le chiavi dei nodi presenti in un albero binario:

```
1  if T≠ NIL then
2    PrintTree(T→ left)
3    PrintTree(T→ right)
4    print(T→ key)
```

Algoritmo 4.11: PRINTTREE(T)

Dal momento che le varie sottochiamate vengono applicate su sottoalberi diversi sarà necessario salvare in uno stack il riferimento al sottoalbero corrente nel momento in cui si effettua una chiamata ricorsiva. Si ha quindi:

```
1  ct = T
```

```
start = true
stackT = NIL
while (start || stackT ≠ NIL) do
  if (start) then
    if (ct ≠ NIL) then
      // Salvataggio contesto
      stackT = push(stackT, ct)
      ct = ct → left
    else
      last = ct
      start = false
  else
    ct = top(stackT)
    // Individuazione punto di ritorno
    if (last = ct → left && ct → right ≠ NIL) then
      ct = ct → right
      start = true
    // Fine della seconda sottochiamata
    else
      print(ct → key)
      stackT = pop(stackT)
      last = ct
```

Algoritmo 4.12: PrintTree_Iter(T)

### 4.3.3 L'algoritmo Height

L'algoritmo Height calcola l'altezza di un albero binario di ricerca. Nella sua versione ricorsiva l'algoritmo si richiama in post-ordine fino al raggiungimento delle foglie:

```
h = -1
if T ≠ NIL then
  sx = Height(T→ left)
  dx = Height(T→right)
  h = 1 + max (sx,dx)
return h
```

Algoritmo 4.13: Height(T)

La traduzione dell'algoritmo 4.13 in uno iterativo richiede l'utilizzo di uno stack per salvare i riferimenti ai sottoalberi correnti, ai valori di `sx` e `dx`. Non sarà necessario uno stack per il valore $h$ in quanto questo viene sempre calcolato ad ogni sottochiamata e non viene mai utilizzato il valore precedente. Per discriminare il punto di ritorno si può utilizzare la variabile `last` che conterrà il valore del sottoalbero corrente. Si ha quindi:

```
ct = T
h = -1
start = true
stackT = stackSx = stackDx = NIL
while (start || stackT ≠ NIL) do
  if (start) then
    if (ct ≠ NIL) then
      stackT = push(stackT, ct)
      ct = ct → left
    else
      h = -1
      last = ct
      start = false
  else
    ct = top(stackT)
    if (last = ct → left && ct → right ≠ NIL) then
      // Salvataggio contesto
      stackSx = push(stackSx, sx)
      stackDx = push(stackDx, dx)
      ct = ct → right
      start = true
    else
      if (ct ≠ NIL) then
        sx = top(stackSx)
        dx = top(stackDx)
        h = 1 + max(sx,dx)
      else
```

```
28          h = -1
29          stackT = pop(stackT)
30          stackSx = pop(stackSx)
31          stackDx = pop(stackDx)
32          last = ct
33  return h
```

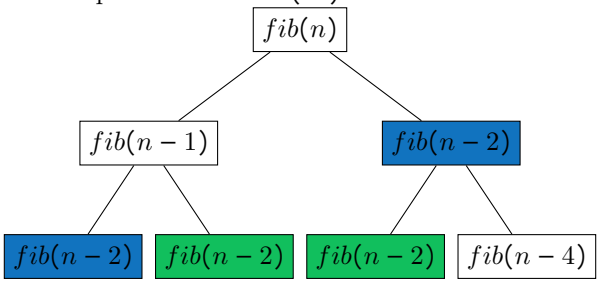
Algoritmo 4.14: HEIGHT_ITER(T)

## 4.4 LA RICORSIONE IN CODA

Come detto all'inizio del capitolo la ricorsione può richiedere una grande allocazione di memoria dato dallo stack dei record di attivazione per ognuna delle chiamate effettuate col rischio di incorrere in overflow di memoria.

Esempio

Si può dimostrare, ad esempio, che nell'esecuzione dell'Algoritmo FIB($n$) (Algoritmo 4.5) le chiamate FIB(0) e FIB(1) sono calcolati FIB($n+1$) volte. Nella computazione di FIB(29) sono chiamati 832040 = FIB(30) volte.




Per evitare tale dispendio di memoria è possibile pensare di costruire una ricorsione definita in modo tale che nel caso ricorsivo l'**ultima operazione** da eseguire è la chiamata ricorsiva. Infatti, se l'ultima istruzione effettuata nella funzione chiamante risulta la chiamata ricorsiva, allora non è necessario specificare alcun indirizzo di ritorno nella funzione chiamante ed è possibile sostituire tale valore con l'indirizzo di ritorno della funzione principale da cui la ricorsiva veniva chiamata.

Consideriamo ad esempio l'algoritmo di ricerca sugli alberi binari di ricerca:

```
1  ret = T
2  if T ≠ NIL then
3    if (T→ key < k) then
4      ret = SearchABR(T → right, k)
5    else if (T→ key > k) then
6      ret = SearchABR(T → left, k)
7  return ret
```

Algoritmo 4.15: SEARCHABR(T,k)

In questa situazione si osserva che non è assolutamente necessario salvare il contesto degli antenati (le chiamate padri) in quanto il calcolo della variabile `ret` avviene in modo indipendente dai suoi precedenti. Non è pertanto necessaria alcuna memoria aggiuntiva in quanto la ricerca di un nodo si riduce, nel caso degli alberi binari di ricerca, alla discesa lungo un percorso ben definito.

```
1  ret = T
2  while (ret ≠ NIL && ret → key ≠ k) do
3    if (ret → key < k) then
4      ret = ret → right
5    else
6      ret = ret → left
7  return ret
```

Algoritmo 4.16: SEARCHABR_ITER(T,k)

La categoria di algoritmi per i quali non serve uno stack per il salvataggio del contesto adoperano una tipologia di ricorsione che prende il nome di **ricorsione in coda**. Sintatticamente, la ricorsione in coda corrisponde ad un algoritmo che effettua una sola chiamata ricorsiva senza dover effettuare infine una risalita[4] lungo lo stack di attivazione per il recupero del contesto precedente. In modo del tutto analogo è possibile riscrivere l'algoritmo ricorsivo 4.1 per il calcolo del fattoriale $n!$ in modo tale che effettui una ricorsione in coda:

[4] Per convenzione gli stack evolvono verso il basso.

```
t = 1
while (n>0) do
  t = t*n
  n = n-1
return t
```

Algoritmo 4.17: FATTORIALE_ITER(N)

Come si può osservare, non è necessario alcuno stack per il salvataggio del contesto delle chiamate interrotte in quanto il calcolo del risultato altro non fa che utilizzare le variabili condivise nel corso dell'iterazione: `t` per il calcolo del risultato e il parametro attuale `n` che viene decrementato di iterazione in iterazione.

# LA CRESCITA DELLE FUNZIONI

Il tasso di crescita del tempo di esecuzione di un algoritmo fornisce una semplice caratterizzazione dell'efficienza dell'algoritmo; inoltre, ci consente di confrontare le prestazioni relative di algoritmi alternativi. Quanto operiamo con dimensioni dell'input abbastanza grandi da rendere rilevante soltanto il tasso di crescita del tempo di esecuzione, stiamo studiando l'efficienza **asintotica** degli algoritmi. In altre parole, ci interessa sapere come aumenta il tempo di esecuzione di un algoritmo al crescere della dimensione dell'input *al limite*, quando la dimensione dell'input cresce senza limitazioni.

## 5.1 NOTAZIONE ASINTOTICA

Le notazioni che usiamo per descrivere il tempo di esecuzione asintotico di un algoritmo sono definite in termini di funzioni il cui dominio è l'insieme dei numeri naturali $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$.

### 5.1.1 Notazione Theta grande

**Notazione Theta grande**

Per una data funzione $g(n)$, indichiamo con $\Theta(g(n))$ l'insieme delle funzioni:

$$\Theta(g(n)) = \{f(n) \mid \text{esistono delle costanti positive } c_1, c_2 \text{ e } n_0 \text{ tali che } 0 \le c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n) \ \ \forall\ n \ge n_0\} \quad (5.1)$$

Una funzione $f(n)$ appartiene all'insieme $\Theta(g(n))$ se esistono delle costanti positive $c_1$ e $c_2$ tali che possa essere "racchiusa" fra $c_1 \cdot g(n)$ e $c_2 \cdot g(n)$, per valori sufficientemente grandi di $n$. Poiché $\Theta(g(n))$ è un insieme, dovremmo scrivere "$f(n) \in \Theta(g(n))$" per indicare che la funzione $f(n)$ è un elemento di $\Theta(g(n))$. Invece, con un abuso di notazione, di solito si trova scritto "$f(n) = \Theta(g(n))$" per esprimere lo stesso concetto. Un altro modo per definire che $f(n)$ e $g(n)$ sono **asintoticamente equivalenti** è il seguente:

$$f(n) = \Theta(g(n)) \iff \lim_{n \to +\infty} \frac{f(n)}{g(n)} = k > 0 \quad (5.2)$$

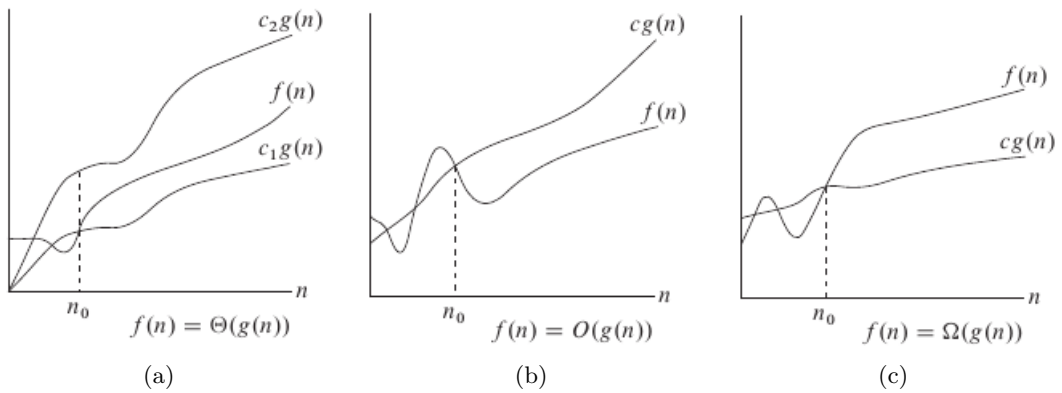



Figura 5.1: Esempi grafici delle notazioni $\Theta$, O grande e $\Omega$

La Figura 5.1a presenta un grafico intuitivo delle funzioni $f(n)$ e $g(n)$, dove $f(n) = \Theta(g(n))$. Per tutti i valori di $n$ a destra di $n_0$, il valore di $f(n)$ coincide o sta sopra $c_1 g(n)$ o sta sotto $c_2 g(n)$. In altre parole, per ogni $n \geq n_0$, la funzione $f(n)$ è uguale a $g(n)$ a meno di un fattore costante. Si dice che $g(n)$ è un **limite asintoticamente stretto** per $f(n)$.

Esempio

Dimostriamo che $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. Per farlo dobbiamo determinare le costanti positive $c_1$, $c_2$ e $n_0$ in modo che:

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

per qualsiasi $n \geq n_0$. Dividendo per $n^2$, si ha:

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

La disuguaglianza destra può essere resa valida per qualsiasi valore di $n \geq 1$ scegliendo una costante $c_2 \geq \frac{1}{2}$. Analogamente, la disuguaglianza sinistra può essere resa valida per qualsiasi valore di $n \geq 7$ scegliendo una costante $c_1 \leq \frac{1}{14}$. Quindi scegliendo $c_1 = \frac{1}{14}$, $c_2 = \frac{1}{2}$ e $n_0 = 7$, possiamo verificare che $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.

### 5.1.2 Notazione O grande

La notazione $\Theta$ limita asintoticamente una funzione da sopra e da sotto. Quando invece abbiamo soltanto un **limite asintotico superiore**, utilizziamo la notazione O grande.

**Notazione O grande**

Per una data funzione $g(n)$, denotiamo con $O(g(n))$ l'insieme delle funzioni:

$$O(g(n)) = \{f(n) \mid \exists\, c, n_0 \in \mathbb{R}^+ \text{ tali che } 0 \leq f(n) \leq cg(n) \;\; \forall\, n \geq n_0\} \tag{5.3}$$

La notazione O si usa per assegnare un limite superiore a una funzione, a meno di un fattore costante. Un altro modo per definire che $f(n)$ *non cresce più velocemente* di $g(n)$ è il seguente:

$$f(n) = O(g(n)) \iff \lim_{n \to +\infty} \frac{f(n)}{g(n)} = 0 \tag{5.4}$$

$f(n)$ è quindi un infinito superiore rispetto a $g(n)$.

La figura 5.1b illustra il concetto intuitivo che sta dietro questa notazione. Per qualsiasi valore n a destra di $n_0$, il valore della funzione $f(n)$ coincide o sta sotto $cg(n)$.

Notiamo che $f(n) = \Theta(g(n))$ implica $f(n) = O(g(n))$, in quanto la notazione $\Theta$ è una notazione più forte della notazione O. Secondo la notazione degli insiemi possiamo scrivere $\Theta(g(n)) \subseteq O(g(n))$.

### 5.1.3 Notazione Omega grande

Così come la notazione O grande fornisce un limite asintotico *superiore* a una funzione, la notazione $\Omega$ fornisce un **limite asintotico inferiore**.

**Notazione Omega grande**

Per una data funzione $g(n)$, denotiamo con $\Omega(g(n))$ l'insieme delle funzioni

$$\Omega(g(n)) = \{f(n) \;:\; \exists c, n_0 \in \mathbb{R}^+ \text{ tali che } 0 \leq cg(n) \leq f(n) \;\; \forall\, n \geq n_0\} \tag{5.5}$$

Un altro modo per dire che $f(n)$ *non cresce più lentamente* di $g(n)$ è attraverso il seguente limite che esprime il fatto che $g(n)$ è un infinito superiore rispetto a $f(n)$:

$$f(n) = \Omega(g(n)) \iff \lim_{n \to +\infty} \frac{f(n)}{g(n)} = +\infty \tag{5.6}$$

Il concetto intuitivo che sta dietro la notazione $\Omega$ è illustrato nella figura 5.1c. Per tutti i valori di n a destra di $n_0$, il valore di $f(n)$ coincide o sta sopra $cg(n)$.

Molte delle proprietà delle relazioni fra numeri reali si applicano anche ai confronti asintotici. Supponiamo che $f(n)$ e $g(n)$ siano asintoticamente positive.

### 5.2.1 Proprietà transitiva

**Teorema**

Valgono le seguenti proprietà:

$$f(n) = \Theta(g(n)) \land g(n) = \Theta(h(n)) \implies f(n) = \Theta(h(n)) \tag{5.7}$$
$$f(n) = O(g(n)) \land g(n) = O(h(n)) \implies f(n) = O(h(n)) \tag{5.8}$$
$$f(n) = \Omega(g(n)) \land g(n) = \Omega(h(n)) \implies f(n) = \Omega(h(n)) \tag{5.9}$$

Dimostriamo per comodità la (2.5), le altre proprietà si dimostrano in modo analogo. Per definizione di O grande:

$$\exists n_0 > 0 \quad \exists c_1 > 0 \quad \forall n \geq n_0 \quad f(n) \leq c_1 g(n)$$
$$\exists n_1 > 0 \quad \exists c_2 > 0 \quad \forall n \geq n_0 \quad g(n) \leq c_2 h(n)$$

Assumendo queste due formule vere fissiamo $n_2 = \max\{n_0, n_1\}$ quindi

$$\exists n_2 > 0 \quad \exists c_1, c_2 > 0 \quad \forall n \geq n_2 \quad f(n) \leq cg(n) \quad \land \quad g(n) \leq c_2 h(n)$$

Maggiorando:

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

Ponendo $\overline{c} = c_1 c_2$ si ottiene

$$\exists n_2 > 0 \quad \exists \overline{c} > 0 \quad \forall n \geq n_2 \quad f(n) \leq \overline{c} h(n)$$

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

■

### 5.2.2 Proprietà riflessiva

**Teorema**

Valgono le seguenti proprietà:

$$f(n) = \Theta(f(n)) \tag{5.10}$$
$$f(n) = O(f(n)) \tag{5.11}$$
$$f(n) = \Omega(f(n)) \tag{5.12}$$

### 5.2.3 Proprietà simmetrica

**Teorema**

Vale la seguente proprietà

$$f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n)) \tag{5.13}$$

Dalla definizione di notazione $\Theta$:

$$\exists n_0 > 0 \quad \exists c_1, c_2 > 0 \quad c_1 g(n) \leq f(n) \leq c_2 g(n) \tag{5.14}$$

possiamo estrarre delle costanti utili per dimostrare il teorema, infatti:

$$c_1 g(n) \leq f(n) \leftrightarrow f(n) \leq \frac{1}{c_1} f(n)$$
$$f(n) \leq c_2 g(n) \leftrightarrow \frac{1}{c_2} \leq g(n)$$

(5.15)

quindi ponendo

$$c_3 = \frac{1}{c_1}$$
$$c_4 = \frac{1}{c_2}$$

possiamo scrivere

$$\exists n_0 > 0 \quad \exists c_3, c_4 > 0 \quad c_4 f(n) \leq g(n) \leq c_3 f(n) \tag{5.16}$$

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

■

### 5.2.4 Simmetria trasposta

**Teorema**

Vale la seguente proprietà

$$f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n)) \tag{5.17}$$

## 5.3 RICERCA DELLE COSTANTI MEDIANTE METODO GRAFICO

Attraverso la definizione mediante limite delle notazioni O grande, $\Theta$ e $\Omega$ è possibile ottenere un metodo per la ricerca delle costanti $c_1$, $c_2$ ed $n_0$ senza dover applicare necessariamente la definizione analitica. Infatti, posto

$$h(n) = \frac{f(n)}{g(n)}$$

è possibile studiare il grafico della sua funzione per ottenere le costanti ricercate. Osserviamo infatti che, se $f(n) = \Theta(g(n))$ allora vale

$$\exists n_0 \quad \exists c_1, c_2 > 0 \quad \forall n > n_0 \quad c_1 g(n) < f(n) < c_2 g(n)$$

dividendo per $g(n)$ si ottiene:

$$\exists n_0 \quad \exists c_1, c_2 > 0 \quad \forall n > n_0 \quad c_1 < \frac{f(n)}{g(n)} < c_2$$

e sostituendo il rapporto $\frac{f(n)}{g(n)} = h(n)$ :

$$\exists n_0 \quad \exists c_1, c_2 > 0 \quad \forall n > n_0 \quad c_1 < h(n) < c_2$$

Il metodo grafico consiste nella ricerca delle due costanti $c_1$ e $c_2$ che delimitano l'andamento asintotico della funzione $h(n)$. Supposto quindi che esista e sia limitato il limite

$$\lim_{n \to +\infty} h(n) = l > 0$$

si ottiene che la retta $y = l$ è un asintoto orizzontale per la funzione $h(n)$.

Per tale asintoto possono presentarsi tre situazioni:

1. $h(n)$ **tende a** $y = l$ **dall'alto.** Scelto un $n_0$ in cui la funzione $h(n)$ è *decrescente in un intervallo illimitato* si prendono i valori

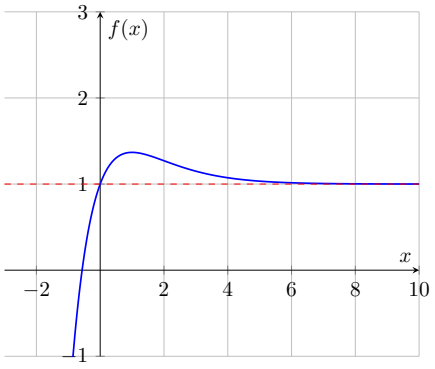
   $$\begin{cases} y = h(n_0) \\ l \end{cases}$$

   che individuano le due costanti che fanno valere la disuguaglianza

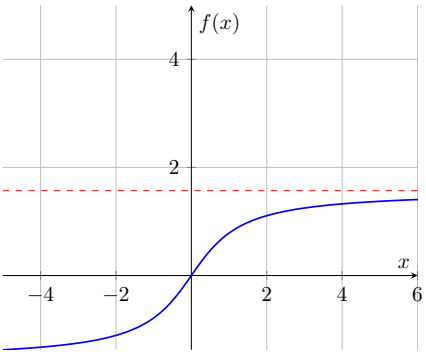
   $$c_1 \leq h(n) \leq c_2$$

2. $h(n)$ **tende a** $y = l$ **dal basso.** Si procede in maniera analoga al caso precedente, però oltre allo studio della monotonia mediante la derivata prima sarà necessario studiare anche il **segno della funzione** dato che siamo alla ricerca di *costanti positive*. All'interno della semiretta a destra dove la funzione assume segno positivo si può scegliere a piacere un punto $x_t$ e valutare la funzione in quel punto: $y_1 = h(x_t)$. Quindi l'insieme delle costanti $(y_1, l)$ rappresenta una coppia di costanti che garantiscono la proprietà da dimostrare.

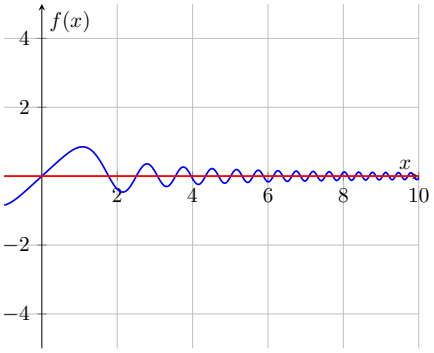
3. $h(n)$ **tende a** $y = l$ **in modo oscillante.** In questo caso, poco frequente, $n_0$ si deve trovare nell'intervallo in cui l'oscillazione si smorza. All'interno di questo intervallo andiamo a selezionare i valori di minimo e massimo locale.




(a) Caso 1: $h(n)$ tende al limite dall'alto




(b) Caso 2: $h(n)$ tende al limite dal basso




(c) Caso 3: $h(n)$ tende al limite in modo oscillante

Figura 5.2

Esempio

Siano

$$f(n) = n$$
$$g(n) = 4n - 10$$

Dimostriamo attraverso il metodo grafico che vale

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

Si calcola innanzitutto il limite

$$\lim_{n\to\infty} \frac{4n-10}{n} = 4 \tag{5.18}$$

Esiste quindi un asintoto orizzontale di equazione $y = 4$. Siamo interessati alla ricerca delle due costanti che verificano la condizione:

$$\exists n_0 > 0 \quad \exists c_1, c_2 > 0 \quad c_1 n \leq 4n - 10 \leq c_2 n \quad \forall n > n_0 \tag{5.19}$$

Studiamo quindi la monotonia della funzione $h(n) = \frac{4n-10}{n}$:

$$h'(n) = \frac{10}{n^2} > 0 \quad \forall n > 0$$

La funzione quindi è strettamente crescente in tutto l'intervallo $(0, +\infty)$ Per cercare due costanti positive è necessario che la funzione $h(n)$ sia positiva

$$h(n) = \frac{4n-10}{n} > 0 \quad \iff \quad n > \frac{5}{2}$$

Prendendo a piacere un $n_0$ all'interno dell'intervallo $(\frac{5}{2}, +\infty)$ si troveranno quindi due costanti $c_1$ e $c_2$ che verificano la disuguaglianza 5.19. Infatti, preso $n_0 = 3$ si valuta la funzione $h(n)$ in quel punto ottenendo

$$h(3) = \frac{4 \cdot 3 - 10}{3} = \frac{2}{3}$$

mentre la costante $c_2$ era stata data dal limite 5.18.

# Il metodo divide et impera: gli alberi di ricorsione

## 6.1 Le ricorrenze

Nel Capitolo 4 si è definito il concetto di ricorsione e di metodo divide et impera. Nel metodo **divide et impera** un problema viene risolto in modo **ricorsivo**, applicando tre passi ad ogni livello di ricorsione.

- **Divide:** questo passo divide il problema in un certo numero di sottoproblemi che sono istanze più piccole dello *stesso problema*.
- **Impera:** i sottoproblemi vengono risolti in modo *ricorsivo*. Quando si è arrivati ad una dimensione sufficientemente piccola delle istanze, queste vengono risolte direttamente.
- **Combina:** le soluzioni dei sottoproblemi vengono combinate per generare la soluzione del problema generale.

Quando i sottoproblemi sono abbastanza grandi da essere risolti ricorsivamente si ha il cosiddetto **caso ricorsivo**. Una volta che i sottoproblemi diventano sufficientemente piccoli da non richiedere ricorsione si dice che la ricorsione "ha toccato il fondo" e che si è raggiunto il **caso base**. Da notare come tutto questo procedimento abbia come base teorica il **principio di induzione**.

**Ricorrenza**

Una **ricorrenza** è un'equazione, detta anche **equazione di ricorrenza**, che descrive una funzione in termini del suo valore di input con input più piccoli. Una ricorrenza per il tempo di esecuzione di un algoritmo divide et impera si basa sui tre passi del paradigma di base.

Esempio

Supposto $T(n)$ sia il tempo di esecuzione di un problema di dimensione $n$. Se la dimensione del problema è sufficientemente piccola, per esempio $n \leq c$ per qualche costante $c$, la soluzione diretta richiede un tempo costante, che si indica con $\Theta(1)$. Supposto che la suddivisione del problema generi $a$ sottoproblemi e che la dimensione di ciascun sottoproblema sia $1/b$ volte la dimensione del problema originale allora servirà tempo $T(n/b)$ per risolvere un sottoproblema di dimensione $n/b$ e quindi, per risolverne $a$, servirà un tempo $a \cdot T(n/b)$. Se impieghiamo un tempo $D(n)$ per dividere il problema in sottoproblemi e un tempo $C(n)$ per combinare le soluzioni dei sottoproblemi nella soluzione del problema originale, si ottiene la ricorrenza:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{negli altri casi} \end{cases}$$

I sottoproblemi non devono necessariamente essere espressi mediante una frazione costante della dimensione del problema iniziale. Ad esempio, una versione ricorsiva della ricerca lineare all'interno di un array non creerebbe un solo sottoproblema in quanto dovrebbe richiamare sé stessa, fino a quando l'elemento da ricercare non è stato trovato, su un array contenente un elemento in meno rispetto alla chiamata padre. La ricorrenza che si ottiene sarà allora:

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

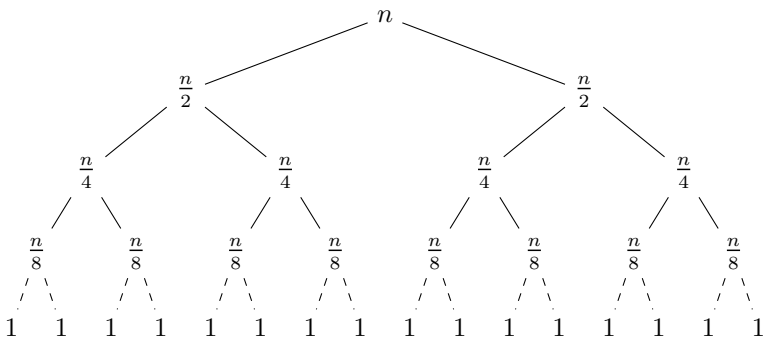
In un **albero di ricorsione** ogni nodo rappresenta il costo di un singolo sottoproblema da qualche parte nell'insieme delle chiamate ricorsive di funzione. Sommando i costi all'interno di ogni livello dell'albero è possibile determinare il costo totale di un algoritmo ricorsivo. Questa analisi prende il nome di **metodo dell'albero di ricorsione**.

Esempio

Si consideri l'equazione:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ 2T(n/2) + \Theta(n) & \text{se } n > 1 \end{cases} \tag{6.1}$$

Data l'equazione 6.1 è possibile costruire l'albero di ricorsione come mostrato di seguito. Ogni nodo interno rappresenta l'input di una chiamata ricorsiva.




Poiché le dimensioni dei sottoproblemi si dimezzano ogni volta che si scende di un livello, alla fine si dovrà raggiungere una condizione al contorno rappresentata dalle foglie dell'ultimo livello. A quale distanza dalla radice si trovano le foglie? La dimensione del sottoproblema per un nodo alla profondità $i$ è $n/2^i$. Quindi, la dimensione del sottoproblema diventa 1 quando

$$\frac{n}{2^i} = 1$$

ovvero quando

$$n = 2^i$$

cioè

$$i = \log_2 n$$

Dunque l'albero ha $\log_2 n + 1$ livelli. Possiamo determinare inoltre il costo a ogni livello dell'albero. Ogni livello ha due volte i nodi del livello precedente; quindi il numero di nodi alla profondità $i$ è $2^i$.

Poiché le dimensioni dei sottoproblemi diminuiscono di un fattore 2 ogni volta che si scende di un livello rispetto alla radice, ogni nodo alla profondità $i$ (per $i = 0, 1, 2, ..., \log_2 n - 1$) ha un costo di $(n/2^i)$. Moltiplicando il costo di ciascun nodo per il numero di nodi su ogni livello si ottiene che il costo, livello per livello, di tutti i nodi interni è esattamente $\Theta(n)$. L'ultimo livello, all'altezza $\log_2 n$, ha $2^{\log_2 n} = n^{\log_2 2} = n$ nodi, ciascuno con un costo $T(1)$, per un costo totale pari a $nT(1)$ che è $\Theta(1)$, in quanto supponiamo che $T(1)$ sia una costante.

A questo punto sommiamo i costi di tutti i livelli per determinare il costo dell'albero intero:

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_2 n} \Theta(n) + \Theta(1) \\ &= \log_2 n \cdot \Theta(n) + \Theta(1) \\ &= \Theta(n \log_2 n) \end{aligned}$$

Esempio

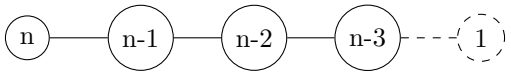
Si consideri la funzione fattoriale $n!$. Il *fattoriale* di un numero naturale $n$ può essere definito sia mediante un approccio iterativo che in maniera ricorsiva. Si dimostra infatti grazie al principio di induzione che:

$$n! = \begin{cases} 1 & \text{se } n = 1 \\ n(n-1)! & \text{se } n > 1 \end{cases}$$

Il tempo di esecuzione per il calcolo del fattoriale può essere espresso tramite la seguente equazione di ricorrenza:

$$T_F(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ T_F(n-1) + \Theta(1) & \text{se } n > 1 \end{cases} \tag{6.2}$$

che può essere rappresentato mediante un albero di ricorsione degenere:




Per ogni nodo conosciamo il *contributo locale* dato da $\Theta(1)$ mentre si vede facilmente che l'altezza totale dell'albero è esattamente $n$. Il costo sarà quindi:

$$\begin{aligned} T_F(n) &= \sum_{i=0}^{n-1} \Theta(1) + \Theta(1) \\ &= (n-1) \cdot \Theta(1) + \Theta(1) \\ &= n \cdot \Theta(1) \\ &= \Theta(n) \end{aligned}$$

### 6.2.1 Forma generale delle equazioni di ricorrenza

**Forma generale delle equazioni di ricorrenza**

La **forma generale delle equazioni di ricorrenza con un solo parametro** è del tipo:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq k \\ \sum_{i=1}^{Z(n)} T(f_i(n)) + g(n) & \text{se } n > k \end{cases} \tag{6.3}$$

Dove:

- $Z(n)$ è il **numero di chiamate ricorsive** esprimibile come funzione di $n$;
- Le funzioni $f_i(n)$ rappresentano le dimensioni dell'input dei sottoproblemi;
- Deve essere $f_i(n) < n$ per garantire la **convergenza** del metodo induttivo;
- La funzione $g(n)$ rappresenta il **contributo locale**;

**Osservazione**

Al variare della funzione $Z(n)$ varierà la forma dell'albero di ricorrenza. Infatti, sia $T(n) = \sqrt{n} \cdot T(\sqrt{n})$. Questa equazione genererà un albero dove ogni livello avrà $\sqrt{n}$ volte la dimensione dell'input: al primo livello saranno quindi $\sqrt{n}$ nodi mentre al secondo $\sqrt{n} \cdot (\sqrt{\sqrt{n}})$ nodi e così via generando quello che si dice un **albero variabile**.

Esempio

Sia

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

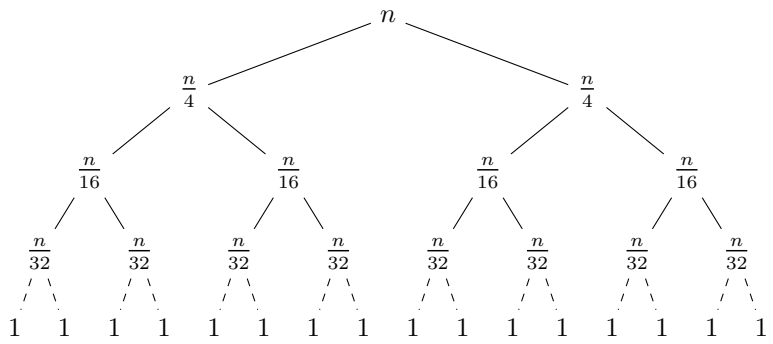
Volendo esprimere l'equazione della ricorrenza come l'abbiamo vista nella forma generale sarà:

$$\sum_{i=1}^{2} T(n/4) + n^2$$

Dove $z(n) = 2$, $g(n) = n^2$, $f_i(n) = n/4$. Il coefficiente 2 ci dice che l'albero generato sarà *binario*.

I passi per risolvere questa equazione saranno quindi i seguenti:

1. **Disegno dell'albero binario:**




2. **Estrapolazione della relazione tra il livello e il numero dei nodi.** Se:

$$f_i = f_j \quad \forall i, j \ (1 \leq i, j \leq z(n))$$

allora tutti i nodi prendono lo stesso input. In questo caso: $f_i = n/4 \quad \forall \ i(1 \leq i \leq 2)$ e ad ogni livello si ha che il **termine generale** è $\frac{n}{4^i}$

3. **Associazione del costo di ogni livello:** ciascun nodo ha un costo quadratico rispetto all'input che ricevono, questo ci è dato dall'informazione sul contributo locale presente nell'equazione di ricorrenza. Oltre a questo, ogni livello ha $2^i$ nodi essendo un albero binario. Questo significa che la somma dei nodi su ciascun livello sarà dato da:

$$2^i\Big(\frac{n}{4^i}\Big)^2$$

che può essere riscritto come:

$$\begin{aligned} 2^i\Big(\frac{n}{4^i}\Big)^2 &= n^2 \cdot \frac{2^i}{4^{2i}} \\ &= n^2 \cdot 2^{i-4i} \\ &= n^2 \cdot 2^{-3i} \\ &= \frac{n^2}{2^{3i}} \end{aligned}$$

4. **Calcolo dell'altezza dell'albero cercando il livello delle foglie.** Per ottenere delle foglie bisogna raggiungere il caso base. Quindi:

$$\frac{n}{4^i} = 1 \Leftrightarrow n = 4^i \Leftrightarrow i = \log_4 n \Leftrightarrow i = \frac{\log_2 n}{2}$$

5. **Calcolo i contributi delle foglie:** nel caso di **alberi pieni** le foglie si trovano tutte sullo *stesso livello*[1]. Nel caso di questa equazione ci troviamo davanti ad un albero pieno in quanto la funzione $f_i$ è unica e il suo termine decresce in maniera quadratica.

Se l'altezza è $h = \frac{\log_2 n}{2}$ il numero delle foglie sarà $2^h$, ovvero:

$$\begin{aligned} 2^h &= 2^{\frac{\log_2 n}{2}} \\ &= 2^{\frac{1}{2} \cdot \log_2 n)} \\ &= (2^{\log_2 n})^{\frac{1}{2}} \\ &= \sqrt{n} \end{aligned}$$

6. **Calcolo del costo totale:** Si somma il costo delle foglie che sarà $1 \cdot \sqrt{n} = \sqrt{n}$ per il costo dei nodi interni.

$$\sqrt{n} + \sum_{i=0}^{h-1}(\frac{n^2}{8^i}) = \sqrt{n} + \sum_{i=0}^{\frac{\log_2 n}{2}-1}(\frac{n^2}{8^i})$$

Portando fuori dalla sommatoria il termine $n^2$ si ottiene:

$$\sqrt{n} + n^2 \cdot \sum_{i=0}^{\frac{\log_2 n}{2}-1}(\frac{1}{8^i})$$

[1] In generale quando le funzioni $f_i$ sono diverse non ci troveremo davanti ad alberi pieni e sarà impattante nel calcolo del costo generale.

Ottenendo così una *serie geometrica*. Notiamo inoltre che la *ragione* della serie geometrica è minore di quindi, questo garantisce che la serie:

$$\sum_{i=0}^{\infty}(\frac{1}{8})^i$$

converge e in particolare la sua somma vale:

$$\frac{1}{1-\frac{1}{8}} = \frac{8}{7}$$

Quindi possiamo dire in generale che, data una serie geometrica convergente vale:

$$\underbrace{\sum_{i=0}^{0} x^i}_{\Theta(1)} \leq \sum_{i=0}^{z} x^i \leq \underbrace{\sum_{i=0}^{\infty} x^i}_{\Theta(1)} = \frac{1}{1-x}$$

Quindi:

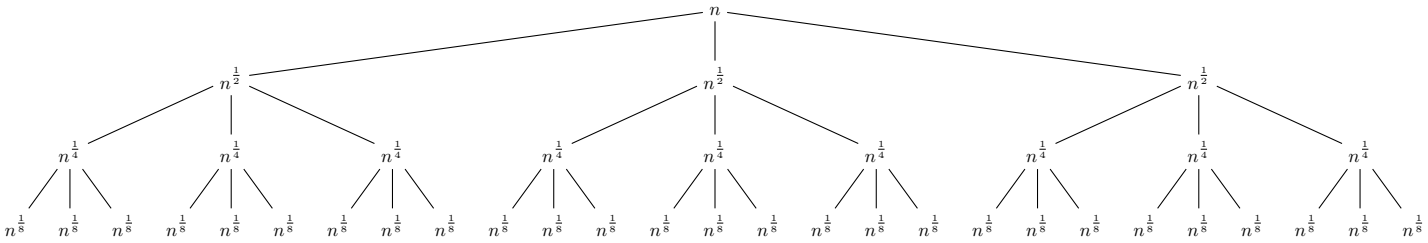
$$\sqrt{n} + n^2 \cdot \underbrace{\sum_{i=0}^{\frac{\log_2 n}{2}-1} (\frac{1}{8^i})}_{\Theta(1)} = \Theta(\sqrt{n} + n^2) = \Theta(n^2)$$

**Esempio**

Sia

$$T(n) = \begin{cases} 1 & \text{se } n \leq 2 \\ 3 \cdot T(\sqrt{n}) + 1 & \text{se } n > 2 \end{cases}$$

Nella nostra analisi la funzione $f_i(n)$ mette in relazione la dimensione dell'input tra nodo padre e nodo figlio. Quindi in particolare indica la *velocità della decrescita*. Più veloce sarà più basso sarà l'albero di ricorrenza. Essendo unica la funzione è garantito il fatto che abbiamo a che fare con un albero pieno dove ogni nodo genera tre nodi figli.



Il termine generale dei nodi per ciascun livello sarà dato da:

$$n^{\frac{1}{2^i}}$$

Dove ciascun nodo contribuisce in modo costante. Il costo di ciascun livello sarà quindi:

$$3^i$$

Per trovare l'altezza si pone

$$n^{\frac{1}{2^i}} = 2$$

Applicando il logaritmo si ottiene:

$$\begin{aligned} \log_2 n^{\frac{1}{2^i}} &= \frac{1}{2^i} \cdot \log_2 n \\ &= 1 \\ &\iff \log_2 n = 2^i \\ &\iff \log_2(\log_2 n) = i \end{aligned}$$

Le foglie saranno quindi:

$$3^{\log_2(\log_2 n)}$$

Quindi il costo dell'equazione di ricorrenza sarà:

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_2(\log_2 n))-1} 3^i + 3^{\log_2(\log_2 n)} \\ &= (\log_2 n)^{\log_2 3} + \sum_{i=0}^{\log_2(\log_2 n))-1} 3^i \\ &= (\log_2 n)^{\log_2 3} + \frac{3^{\log_2(\log_2 n)} - 1}{2} \\ &= (\log_2 n)^{\log_2 3} + \frac{(\log_2 n)^{\log_2 3} - 1}{2} \\ &= \frac{2(\log_2 n)^{\log_2 3} + (\log_2 n)^{\log_2 3} - 1}{2} \\ &= \frac{3(\log_2 n)^{\log_2 3} - 1}{2} \\ &= \Theta((\log_2 n)^{\log_2 3}) \end{aligned}$$

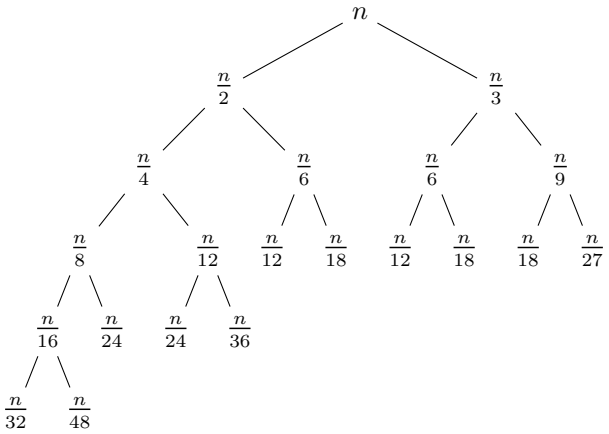
Consideriamo l'equazione

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

In questo esempio abbiamo due funzioni:

$$f_1(n) = \frac{n}{2} \qquad f_2(n) = \frac{n}{3}$$

Scriviamo l'albero di ricorrenza:




A differenza di un albero pieno, in questo caso esistono rami più veloci e rami più lenti. Questo perché le espressioni che corrispondono alle dimensioni dell'input sono diverse. L'albero che ne esce non sarà quindi pieno: ci saranno rami che arriveranno più lentamente alle foglie mentre altri avranno una profondità molto piccola.

In casi del genere, associato il contributo di ogni nodo e capito il termine generale per ogni livello, si calcola il costo dei due alberi pieni relativi all'altezza del ramo più veloce e del ramo più lento. Il costo dell'albero sarà compreso tra questi due limiti, uno *inferiore* e uno *superiore*. Si procede quindi per **approssimazione**.

Per il primo albero si usa la lunghezza del ramo più veloce, ovvero i termini che decrescono seguendo la successione:

$$\frac{n}{3}, \frac{n}{9}, \frac{n}{27}, \dots$$

È chiaro che il termine generale, livello dopo livello, è:

$$\frac{n}{3^i}$$

Ciò significa che l'altezza dell'albero sarà:

$$\frac{n}{3^i} = 1 \implies n = 3^i \iff \log_3 n = i$$

Per il secondo albero useremo la lunghezza del ramo più lento, ovvero i termini che decrescono seguendo la successione: $\frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \dots$. Il termine generale sarà:

$$\frac{n}{2^i}$$

e l'altezza dell'albero sarà:

$$\frac{n}{2^i} = 1 \implies i = \log_2 n$$

Per ogni livello il contributo dato dalla somma dei nodi è:

- **Livello 1:** $\frac{n}{2} + \frac{n}{3} = \frac{5n}{6}$
- **Livello 2:** $\frac{n}{4} + \frac{n}{6} + \frac{n}{6} + \frac{n}{9} = \frac{25}{36}n = (\frac{5}{6})^2 n$
- **Livello 3:** $\frac{n}{8} + \frac{n}{12} + \frac{n}{12} + \frac{n}{8} + \frac{n}{12} + \frac{n}{18} + \frac{n}{18} + \frac{n}{27} = \frac{125}{216}n = (\frac{5}{6})^3 n$

Si ottiene quindi che il termine generale, livello dopo livello sarà dato da:

$$(\frac{5}{6})^i n$$

A questo punto si possono calcolare le due sommatorie:

$$T'(n) = \sum_{i=0}^{\log_3 n} \left(\frac{5}{6}\right)^i n$$

e

$$T''(n) = \sum_{i=0}^{\log_2 n} \left(\frac{5}{6}\right)^i n$$

Sarà quindi:

$$\begin{aligned} T'(n) &= \sum_{i=0}^{\log_3 n} \left(\frac{5}{6}\right)^i n \\ &= n \cdot \sum_{i=0}^{\log_3 n} \left(\frac{5}{6}\right)^i \\ &\leq n \cdot \frac{1}{1 - \frac{5}{6}} \\ &= 6n \implies \Theta(n) \end{aligned}$$

e analogamente:

$$\begin{aligned} T''(n) &= \sum_{i=0}^{\log_2 n} \left(\frac{5}{6}\right)^i n \\ &\leq 6n \implies \Theta(n) \end{aligned}$$

Quindi, essendo:

$$T'(n) \leq T(n) \leq T''(n)$$

si ha:

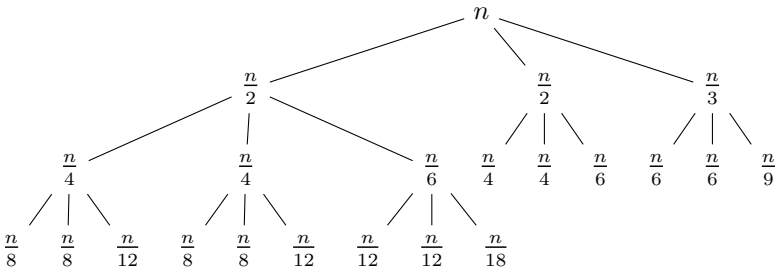
$$T(n) = \Theta(n)$$

Esempio

Si consideri la ricorrenza:

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

In questo caso si ottiene un albero di ricorrenza di questo tipo:




Si ripete lo stesso ragionamento fatto nell'esempio precedente, in questo caso però cambia il contributo generale dei livelli, è infatti:

- **Livello 1:** N
- **Livello 2:** $\frac{n}{2} + \frac{n}{2} + \frac{n}{3} = \frac{4}{3}n$
- **Livello 3:** $\frac{n}{4} + \frac{n}{4} + \frac{n}{6} + ... + \frac{n}{9} = (\frac{4}{3})^2 n$

Si dimostra per induzione che il termine generale è:

$$\left(\frac{4}{3}\right)^i n$$

Le due equazioni dei limiti inferiori e superiori diventano quindi:

$$T'(n) = \sum_{i=0}^{\log_3 n} \left(\frac{4}{3}\right)^i n$$

e

$$T''(n) = \sum_{i=0}^{\log_2 n} \left(\frac{4}{3}\right)^i n$$

dove però la ragione è maggiore di 1 e quindi non maggiorabile tramite la serie infinita. Occorre quindi applicare la formula chiusa:

$$\begin{aligned} T'(n) &= \sum_{i=0}^{\log_3 n} \left(\frac{4}{3}\right)^i n = \\ &= n \cdot \frac{(4/3)^{\log_3 n+1} - 1}{1/3} = \\ &= 3n\Big(4/3 \cdot (4/3)^{\log_3 n} - 1\Big) = \\ &= 4n \cdot (4/3)^{\log_3 n} - 3n = \\ &= 4n \cdot n^{\log_3 4/3} - 3n = \\ &= n^{\log_3 4/3+1} - 3n \end{aligned}$$

e analogamente

$$\begin{aligned} T''(n) &= \sum_{i=0}^{\log_2 n} \left(\frac{4}{3}\right)^i n = \\ &= n \cdot \frac{(4/3)^{\log_2 n+1} - 1}{1/3} = \\ &= \cdots = \\ &= n^{\log_2 4/3+1} - 3n \end{aligned}$$

Ottenendo così due funzioni distinte, essendo la prima un polinomio di grado $1 + \log_3 4/3$ e la seconda un polinomio di grado $1 + \log_2 4/3$. Scriveremo quindi:

$$\Omega(1 + \log_3 4/3) \leq T(n) \leq O(1 + \log_2 4/3)$$

e non varrà il $\Theta$.

# Strutture dati ramificate: i grafi

## 7.1 Introduzione

Nel Capitolo 4 abbiamo visto come le strutture dati ramificate siano utili per rappresentare dati che hanno una struttura gerarchica. In questo capitolo vedremo come le strutture dati ramificate possono essere utilizzate per rappresentare dati che hanno una struttura non gerarchica, come ad esempio i grafi.

I grafi sono strutture dati che permettono di rappresentare relazioni tra entità. Per esempio, un grafo può essere utilizzato per rappresentare una rete di computer, dove i computer sono le entità e le connessioni tra i computer sono le relazioni. Un altro esempio è quello di un grafo che rappresenta una mappa stradale, dove i nodi sono le città e gli archi sono le strade che collegano le città. Per questo motivo, i grafi sono estremamente utili per risolvere problemi di ottimizzazione e di ricerca.

## 7.2 Definizioni

**Grafo**

Un **grafo** $G$ è una coppia ordinata $(V, E)$ dove $V$ è un insieme finito di elementi detti **nodi** e $E$ è un insieme di coppie di elementi di $V$ detti **archi**.

Come anticipato, i grafi sono estremamente utili per la rappresentazione di relazioni binarie all'interno di un insieme di elementi. L'insieme degli archi, $E$, può essere infatti visto come una relazione binaria su $V$: se $(u, v) \in E$, allora $u$ e $v$ sono collegati da un arco:

$$E = \{(u, v) \in V \times V : u \text{ è collegato a } v\} \subseteq V \times V \tag{7.1}$$

In particolare, se $E$ rappresenta una relazione binaria simmetrica[1], allora il grafo è detto **non orientato**. Altrimenti, se $E$ rappresenta una relazione binaria asimmetrica, allora il grafo è detto **orientato** e gli archi verranno rappresentati con delle frecce.

Esempio

Il grafo $G = (V, E)$ in Figura 7.1a è un grafo non orientato, dove

$$V = \{A, B, C, D, E, F\}$$

ed

$$E = \{(A, D), (A, E), (B, E), (B, F), (C, E), (C, F), (D, E), (E, F)\}$$

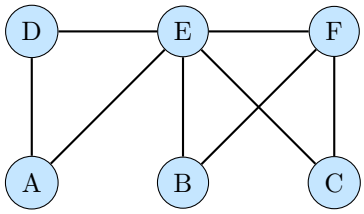
Il grafo $G_2 = (V, E)$ in Figura 7.1b è un grafo orientato, dove

$$V = \{A, B, C, D, E, F\}$$

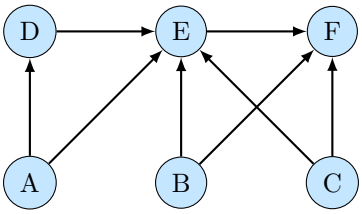
e

$$E = \{(A, D), (A, E), (B, E), (B, F), (C, E), (C, F), (D, E), (E, F)\}$$

[1] Una relazione binaria $\rho$ in un insieme $S$ si dice simmetrica se e solo se, per ogni $x, y \in S$ si ha $x \rho y \iff y \rho x$, ovvero $(x, y) \in G \iff (y, x) \in G$, dove $G$ è il grafico della relazione $\rho$.




(a) Grafo non orientato




(b) Grafo orientato

Figura 7.1

**Osservazione**

Sia $G = (V, E)$ un grafo, chiaramente il numero massimo di archi possibili sarà:

$$|E| \leq |V| \times |V| = |V|^2$$

Inoltre, se il grafo è non orientato, allora

$$|E| \leq \frac{|V| \times (|V| - 1)}{2} = \binom{|V|}{2}$$

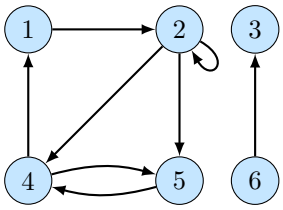
Infatti, se il grafo è non orientato, allora:

$$(u, v) \in E \iff (v, u) \in E$$

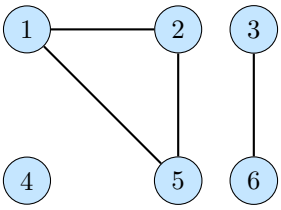
Molte definizioni per i grafi orientati e non orientati sono simili, sebbene alcuni termini abbiano significati leggermente diversi nei due contesti.

**Archi entranti, uscenti ed incidenti**

Se $(u, v)$ è un arco in un grafo orientato $G = (V, E)$, diciamo che $(u, v)$ **esce** dal vertice $u$ ed **entra** nel vertice $v$. Se $(u, v)$ è un arco in un grafo non orientato, diciamo che $(u, v)$ è **incidente** nei vertici $u$ e $v$.




(a) Grafo orientato




(b) Grafo non orientato

Figura 7.2

Esempio

In Figura 7.2a il vertice 2 ha due archi entranti e tre archi uscenti. In particolare, un arco che esce ed entra nello stesso vertice viene chiamato **cappio**. In Figura 7.2b il vertice 2 ha due archi incidenti: $(1, 2)$ e $(2, 5)$. In un grafo non orientato non è possibile avere cappi.

**Relazione di adiacenza**

Se $(u, v)$ è un arco di un grafo $G = (V, E)$, diciamo che il vertice $v$ è **adiacente** al vertice $u$.

Se il grafo non è orientato, la relazione di adiacenza è simmetrica. Se il grafo è orientato, la relazione di adiacenza non è necessariamente simmetrica.

Esempio

In Figura 7.2a e 7.2b il vertice 2 è adiacente al vertice 1, perché l'arco $(1, 2)$ appartiene ad entrambi i grafi. Il vertice 1 non è adiacente al vertice 2 nel grafo mostrato nella Figura 7.2a perché l'arco $(2, 1)$ non appartiene al grafo.

**Grado di un vertice**

Il **grado** di un vertice in un grafo non orientato è il numero di archi che incidono nel vertice. Un vertice il cui grado è 0 si dice **isolato**. In un grafo orientato, il **grado uscente** di un vertice è il numero di archi che escono dal vertice; il **grado entrante** di un vertice è il numero di archi che entrano nel vertice. Il **grado** di un vertice in un grafo orientato è la somma del suo grado entrante e del suo grado uscente.

Esempio

Il vertice 2 nella Figura 7.2b ha grado 2 mentre il vertice 4 è isolato. Nel grafo in Figura 7.2a il vertice 2 ha grado entrante 2, grado uscente 3 e grado 5.

### 7.2.1 Percorsi in un grafo

**Percorso in un grafo**

Un **percorso** di **lunghezza** $k$ da un vertice $u$ ad un vertice $u'$ in un grafo $G = (V, E)$ è una sequenza di vertici $\langle v_0, v_1, \ldots, v_k \rangle$ tali che

$$u = v_0$$
$$u' = v_k$$

e

$$\forall i = 1, 2, \ldots, k \qquad (v_{i-1}, v_i) \in E$$

La **lunghezza** del percorso, o **cammino**, è il numero di archi presenti nel cammino. Il cammino **contiene** i vertici $v_0, v_1, \ldots, v_k$ e gli archi $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ (c'è sempre un cammino di lunghezza 0 da $u$ ad $u$). Se c'è un cammino $p$ da $u$ ad $u'$, diciamo che $u'$ è **raggiungibile** da $u$ attraverso $p$ e si denota con il simbolo $u' \rightsquigarrow u$. Un cammino si dice **semplice** se tutti i vertici nel cammino sono distinti.

Esempio

Nella Figura 7.2a, il cammino $\langle 1, 2, 5, 4 \rangle$ è un cammino semplice di lunghezza 3. Il cammino $\langle 2, 5, 4, 5 \rangle$ invece non è semplice.

**Sottopercorso in un grafo**

Un **sottopercorso** di un percorso $p = \langle v_0, v_1, \ldots, v_k \rangle$ è una sottosequenza contigua dei suoi vertici. Ovvero, per ogni $0 \leq i \leq j \leq k$, la sottosequenza dei vertici $\langle v_i, v_{i+1}, \ldots, v_k \rangle$ è un sottopercorso di $p$.

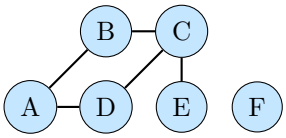
Osservazione

Se la lunghezza di un percorso semplice è limitata dal numero di vertici presenti in un grafo, ovvero $|V|$, il numero di percorsi semplici in un grafo sarà anch'esso limitato in quanto queste saranno tutte e sole le sequenze di lunghezza minore od uguale della cardinalità di $V$.

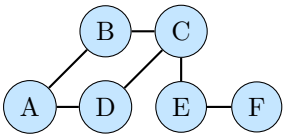
**Ciclo**

In un grafo orientato un cammino $\langle v_0, v_1, \ldots, v_k \rangle$ forma un **ciclo** se $v_0 = v_k$ e il cammino contiene almeno un arco. Il ciclo è **semplice** se $v_1, ..., v_k$ sono anche distinti. Un cappio è un ciclo di lunghezza 1. Un grafo orientato senza cappi si dice **semplice**. In un grafo non orientato un cammino $\langle v_0, v_1, \ldots, v_k \rangle$ forma un **ciclo (semplice)** se $k \geq 3$, $v_0 = v_k$ e $v_1, \ldots, v_k$ sono distinti. Un grafo senza cicli è detto **aciclico**.

Alcuni tipi di grafi hanno dei nomi speciali. Un **grafo completo** è un grafo non orientato in cui ogni coppia di vertici è adiacente. Un grafo aciclico e non orientato è una **foresta**; un grafo connesso, aciclico e non orientato è detto **albero libero**.



(a) Grafo non connesso



(b) Grafo connesso

Figura 7.3

**Osservazione**

Gli alberi binari studiati nel capitolo 3 non sono altro che particolari tipi di grafi connessi, aciclici e non orientati in cui uno dei vertici si distingue da tutti gli altri, la radice, ed ogni vertice ha un grado entrante pari ad uno ed un grado uscente pari al massimo di due.

## 7.3 Rappresentazione dei grafi

Ci sono due metodi standard per la rappresentazione di un grafo $G = (V, E)$:

1. come una collezione di **liste di adiacenza**;
2. come una **matrice di adiacenza**.

Entrambi i metodi possono essere applicati sia ai grafi orientati che a quelli non orientati. Di solito, si preferisce la rappresentazione con liste di adiacenza, perché permette di rappresentare in modo compatto i grafi sparsi e richiede una minore occupazione di memoria al variare dei vertici del grafo.[2]

### 7.3.1 Le liste di adiacenza

La **rappresentazione con liste di adiacenza** di un grafo $G = (V, E)$ consiste in un array $Adj$ di $|V|$ liste, una per ogni vertice in $V$. Per ogni $u \in V$, la lista di adiacenza $Adj[u]$ contiene tutti i vertici $v$ tali che esista un arco $(u, v) \in E$. Ovvero $Adj[u]$ include tutti i vertici **adiacenti** ad $u$ in $G$.

Le liste di adiacenza (vedi Figura 7.4b) possono essere facilmente adattate per rappresentare i **grafi pesati**, cioè i grafi per i quali ogni arco ha un **peso** associato. In questi casi, il peso $w(u, v)$ dell'arco $(u, v) \in E$ viene memorizzato semplicemente assieme al vertice $v$ nella lista di adiacenza di $u$.

### 7.3.2 Le matrici di adiacenza

Uno svantaggio potenziale della rappresentazione con liste di adiacenza è che non c'è modo più veloce per determinare se un particolare arco $(u, v)$ è presente nel grafo invece che scorrere la lista $Adj[u]$. Per porre un rimedio a questo svantaggio, si può rappresentare il grafo con una **matrice di adiacenza**, al costo di usare una maggiore quantità di memoria.

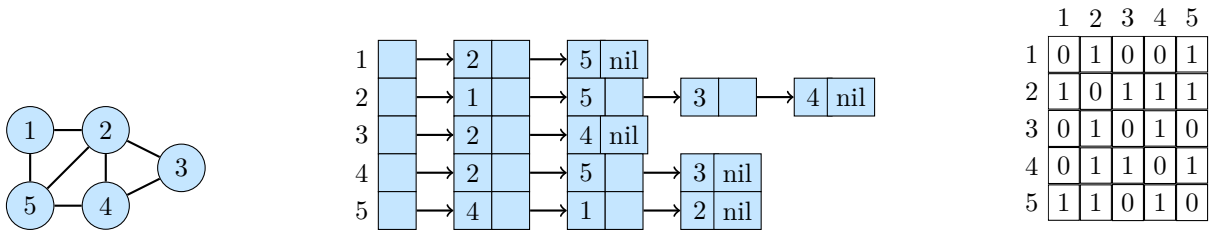
Per la **rappresentazione con matrice di adiacenza** (vedi Figura 7.4c) di un grafo $G = (V, E)$ si suppone che i vertici siano numerati $1, 2, \ldots, |V|$ in modo arbitrario. La rappresentazione con matrice di adiacenza di un grafo $G$ consiste in una matrice $A = (a_{ij})$ di dimensioni $|V| \times |V|$ che descrive la **funzione caratteristica** in $E$:

$$a_{ij} = \begin{cases} 1 & \text{se } (i, j) \in E \\ 0 & \text{altrimenti} \end{cases} \tag{7.2}$$

## 7.4 Visita in ampiezza

La **visita in ampiezza** (Breadth-First-Search) è uno dei più semplici algoritmi di ricerca nei grafi e sta alla base di molti algoritmi che operano con i grafi. Dato un grafo $G = (V, E)$ e un vertice distinto $s$, detto **sorgente**, la visita in ampiezza ispeziona sistematicamente gli archi di $G$ per "scoprire" tutti i vertici raggiungibili da $s$ individuando la distanza da $s$ ad ognuno dei vertici raggiungibili. La visita in ampiezza è chiamata così infatti perché espande la frontiera fra i vertici scoperti e quelli da scoprire in maniera uniforme lungo l'ampiezza della frontiera. Questo significa che l'algoritmo scopre tutti i vertici che si trovano ad una distanza $k$ da $s$, prima di scoprire i vertici a distanza $k + 1$.

[2] Una matrice richiederebbe un costo spaziale quadratico sul numero di vertici ($|V|^2$) mentre la rappresentazione attraverso liste di adiacenza solo un costo lineare ($|V|$).




(a) (b) (c)

Figura 7.4: Due rappresentazioni di un grafo non orientato. **(7.4a)** Un grafo non orientato $G$ con cinque vertici e sette archi. **(7.4b)** Una rappresentazione con liste di adiacenza di $G$. **(7.4c)** Una rappresentazione con matrice di adiacenza di $G$.

Per tenere traccia del lavoro svolto ed evitare di raggiungere nodi già visitati in precedenza, la visita in ampiezza colora i vertici di bianco, di grigio o di nero:

- **Bianco:** il nodo non è stato ancora visitato;
- **Grigio:** il nodo è stato visitato ma potrebbe avere di vicini non visitati;
- **Nero:** il nodo è stato visitato ed anche tutti i suoi vicini.

Inizialmente tutti i vertici sono bianchi; successivamente possono diventare grigi ed infine neri. Per questo motivo possiamo implementare l'algoritmo Init(G) (Algoritmo 7.1) che si occupa di inizializzare un grafo colorando tutti i suoi vertici di bianco.

```
1  for each v in V do
2    Color[v] = White
```

Algoritmo 7.1: Init(G)

Un vertice viene **scoperto** quando viene incontrato per la prima volta durante la visita; in quel momento cessa di essere un vertice bianco. Se $(u, v) \in E$ e il vertice $u$ è nero, allora il vertice $v$ è grigio oppure nero: ovvero tutti i vertici adiacenti ai vertici neri sono sempre scoperti. I vertici grigi possono avere qualche vertice bianco adiacente per questo motivo essi rappresentano la frontiera fra i vertici scoperti e quelli da scoprire.

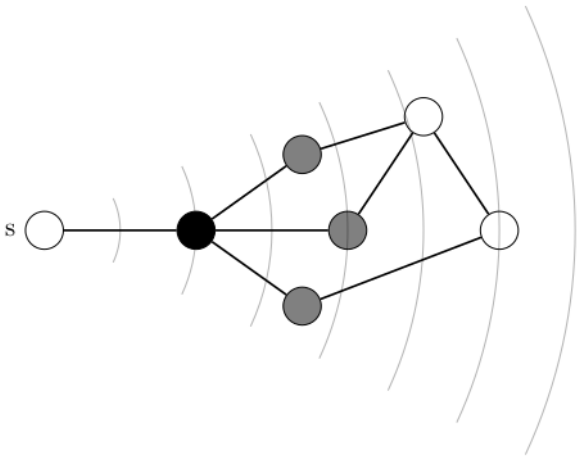



Figura 7.5: L'algoritmo di visita in ampiezza estende di volta in volta la frontiera dei vertici da visitare

La vista in ampiezza costruisce una coda $Q$ che, inizialmente contiene soltanto il vertice sorgente $s$. A questo punto si accodano tutti i vertici presenti colorati di bianco scoperti esplorando la lista di adiacenza e li si colora di grigio. Una volta aver eseguito questa operazione si estrae il nodo in testa alla coda, lo si colora di nero e si aggiungono in coda i vertici bianchi adiacenti scoperti esplorando la lista di adiacenza. L'algoritmo viene eseguito fino a quando la coda non risulta vuota.

```
1  Init(G)
2  Q={s}
3  Color[s] = Gray
4  while Q ≠ {} do
5    x = Head(Q)
6    foreach v ∈ Adj(x) do
7      if Color[v] = b then
8        Q = Enqueue(Q,v)
9        Color[v] = Gray
10   Q = Dequeue(Q)
11   Color[x] = Black
```

Algoritmo 7.2: BFS(G,s)

### 7.4.1 Una versione migliorata dell'algoritmo BFS

L'Algoritmo 7.2 visita i vertici raggiungibili di un grafo al crescere della loro distanza dal vertice sorgente $s$. Ciò significa che ciascun vertice viene raggiunto tramite il cammino più breve possibile da $s$. Infatti, per ogni vertice $v$ raggiungibile da $s$, il cammino semplice che va da $s$ a $v$ corrisponde ad un cammino minimo da $s$ a $v$, cioè un cammino che contiene il minor numero di archi possibile.

**Distanza di un vertice**

La **distanza** di un vertice $v$ dal vertice sorgente $s$ è il numero di archi nel cammino minimo da $s$ a $v$, e si indica con $d(s, v)$.

Osserviamo che l'algoritmo BFS determina una *relazione gerarchica* fra i vertici di un grafo. Infatti, la visita in ampiezza costruisce un albero, detto **albero breath first (BF)**, che inizialmente contiene soltanto la sua radice, ovvero il vertice sorgente $s$. Quando un vertice bianco $v$ viene scoperto durante l'ispezione della lista di adiacenza di un vertice $u$ già scoperto, il vertice $v$ e l'arco $(u, v)$ vengono aggiunti all'albero. Il vertice $u$ rappresenta così il **predecessore** o **padre** di $v$ nell'albero BF e dato che un vertice verrà scoperto sempre e solo una volta, ogni vertice avrà sempre e solo un padre nell'albero BF. È così univocamente determinata la funzione

$$p : V \to \mathbb{N} \cup \{nil\}$$

che associa ad ogni nodo l'indice del vertice padre. A partire da questa osservazione possiamo definire un raffinamento dell'Algoritmo 7.2 che costruisce l'albero BF a partire dai percorsi minimi da $s$ a tutti i vertici raggiungibili.

L'Algoritmo 7.3 calcola quindi la distanza di ogni vertice raggiungibile da $s$ come il numero di archi nel cammino minimo da $s$ al vertice. Questa quantità è memorizzata nell'array $d$. Chiaramente, la distanza da $s$ a $v$ è data dalla distanza da $s$ al suo predecessore, più 1. Useremo il valore $\infty$ per indicare che un vertice non è raggiungibile da $s$. Oltre al vettore $d$ che memorizza le distanze, l'algoritmo utilizza un vettore $p$ che memorizza i predecessori di ogni vertice nell'albero BF. Il predecessore di un vertice $v$ è il vertice $u$ che ha scoperto $v$ per la prima volta. Il predecessore di $s$ è indicato con il valore $nil$.

```
1  for each x in V do
2    Color[x] = white
3    d[x]=∞
4    p[x] = nil
5  Q={s}
6  Color[s] = Gray
7  d[s] = 0
8  p[s] = nil
9  while Q ≠ {} do
10   x = Head(Q)
11   foreach v ∈ Adj(x) do
12     if Color[v] = b then
13       Q = Enqueue(Q,v)
14       Color[v] = Gray
15       d[v] = d[x] + 1
16       p[v] = x
17   Q = Dequeue(Q)
18   Color[x] = Black
```

Algoritmo 7.3: BFS(G,s)

### 7.4.2 Correttezza dell'algoritmo BFS

Prima di dimostrare le varie proprietà della visita in ampiezza, analizziamo il suo tempo di esecuzione con un grafo di input $G = (V, E)$.

Dopo l'inizializzazione, nessun vertice sarà più colorato di bianco, quindi il test nella riga 12 garantisce che ciascun vertice venga accodato al più una volta, e di conseguenza, venga eliminato dalla coda al più una volta. Le operazioni di inserimento e cancellazione dalla coda richiedono un tempo costante, quindi il tempo totale dedicato alle operazioni con la coda risulta lineare sulla dimensione dei vertici del grafo $G$.

Poiché la lista di adiacenza di ciascun vertice viene ispezionata soltanto quando il vertice viene rimosso dalla coda, ogni lista di adiacenza viene ispezionata al più una volta. Poiché l'algoritmo ispeziona le liste di adiacenza di ciascun vertice, la cui somma totale [3]delle lunghezze è pari ad $|E|$, allora il tempo di esecuzione totale di BFS risulta lineare nella dimensione della rappresentazione con liste di adiacenza del grafo $G$: $E + V$.

[3]Di tutte le liste.

**Teorema**

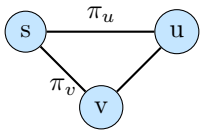
Dato un arbitrario grafo $G$ ed un vertice $s \in V$, al termine dell'esecuzione dell'Algoritmo BFS devono valere:

1. $\forall v \in V$, $v$ sarà visitato se e soltato se $v$ è *raggiungibile* da $s$;
2. $\forall v \in V$ vale: $d[v] = \delta(v, s)$;
3. $\forall v \in V \setminus \{s\}$, se $v$ è raggiungibile allora è ottenibile un percorso minimo da $s$ a $v$ concatenando il percorso minimo da $s$ a $p[v]$.

Il teorema appena enunciato garantisce la correttezza dell'algoritmo BFS. Al contrario degli algoritmi finora visti, dove si è utilizzato il principio di induzione per dimostrare la correttezza, la dimostrazione della correttezza dell'algoritmo BFS non è così immediata in quanto l'algoritmo è stato definito in maniera iterativa. Per dimostrare quindi tale correttezza bisogna studiare il comportamento del programma durante l'esecuzione e dimostrare che le tre proprietà enunciate dal teorema sono sempre valide. Questa tipologia di analisi prende il nome di **analisi a invariante**.

**Proprietà della funzione $\delta$ (Disuguaglianza triangolare)**

Per ogni coppia $(u, v)$ di vertici di un grafo $G$, se $(u, v) \in E$ allora $\delta(s, v) \leq \delta(s, u) + 1$ per ogni vertice $s$.




**Dimostrazione** Supponiamo che la proprietà sia falsa. Allora esiste una coppia $(u, v)$ di vertici di un grafo $G$ tale che $(u, v) \in E$ e $\delta(s, v) > \delta(s, u) + 1$ per ogni vertice $s$.

Sia $\pi_u$ un cammino minimo da $s$ a $u$ e sia $\pi_v$ un cammino minimo da $s$ a $v$ e consideriamo il percorso $\pi_u \cdot v$. Questo percorso è un cammino da $s$ a $v$ e la sua lunghezza è $\delta(s, u) + 1$. Poiché $\delta(s, v) > \delta(s, u) + 1$, allora $\pi_u \cdot v$ è un cammino da $s$ a $v$ più corto di $\pi_v$, il che è assurdo perché $\pi_v$ è un cammino minimo da $s$ a $v$. ■

**Proprietà A**

Durante l'esecuzione di BFS$(G, s)$, per ogni vertice $v \in V$ si ha:

$$d[v] \geq \delta(s, v)$$

**Dimostrazione** Per dimostrare l'invarianza della proprietà A si procede per induzione sul numero di accodamenti in $Q$. Questo perché le modifiche ai valori di $d[v]$ avvengono soltanto in corrispondenza degli accodamenti dei vertici in $Q$. Tutto ciò che non è un accodamento, infatti, non modifica i valori di $d[v]$. Indichiamo tale numero con $k$:

1. **Caso base:** Sia $k = 0$. All'inizio dell'esecuzione di BFS$(G, s)$, $Q$ contiene soltanto il vertice $s$. Poiché $d[s] = 0$ e $\delta(s, s) = 0$, la proprietà A è verificata.

2. **Caso induttivo:** Sia $k > 1$ e assumiamo che la proprietà sia verificata per $k - 1$ accodamenti. Tra il $k - 1$-esimo ed il $k$-esimo accodamento non avviene alcuna modifica ai valori di $d$. Sia $x$ l'ultimo vertice accodato e supponiamo di eseguire l'accodamento di un vertice $v$. Essendo $x$ già presente nella coda la sua distanza è stata già calcolata e vale, per ipotesi induttiva:

   $$d[x] \geq \delta(s, x)$$

   Maggiorando entrambi i membri della disuguaglianza triangolare con $+1$ si ottiene:

   $$d[x] + 1 \geq \delta(s, x) + 1$$

   Dovendo accodare $v$ si presuppone che esista un arco che vada da $x$ a $v$, ovvero $(x, v) \in E$. Per la proprietà della funzione $\delta$ si ha allora:

   $$\delta(s, x) + 1 \geq \delta(s, v)$$

   Per transitività si ottiene:

   $$d[x] + 1 \geq \delta(s, v)$$

   Per la definizione di $d[v]$ si ha quindi l'enunciato:

   $$d[v] \geq \delta(s, v)$$

■

Una seconda proprietà invariante che andiamo a dimostrare descrive la relazione fra le stime delle distanze dei vertici accodati.

**Proprietà B**

Durante l'esecuzione di BFS$(G, s)$, se $Q = \langle v_1, v_2, \ldots, v_k \rangle$ allora valgono le seguenti proprietà:
1. $d[v_k] \leq d[v_1] + 1$;
2. $d[v_i] \leq d[v_{i+1}]$ per $i = 1, 2, \ldots, k-1$.

**Dimostrazione** Come già osservato in precedenza, le stime delle distanze vengono modificate soltanto in corrispondenza delle operazioni eseguite sulla coda $Q$, per questo motivo bisogna tenere in considerazione tutte le operazioni di accodamento e decodamento. Sia $k$ il numero di tali operazioni e procediamo per induzione su $k$:

- **Caso base:** Sia $k = 0$. All'inizio dell'esecuzione di BFS$(G, s)$, $Q$ contiene soltanto il vertice $s$. Poiché $d[s] = 0$ le due proprietà sono banalmente verificate.
- **Caso induttivo:** Sia $k > 1$, bisogna dimostrare che la proprietà sia valida dopo l'inserimento e la rimozione di un vertice nella coda. Se il vertice $v_1$ viene eliminato dalla coda, il vertice $v_2$ diventa la nuova testa (se la coda si svuota, allora ci troviamo nuovamente nel caso base). Per l'ipotesi induttiva si ha:

  $$d[v_1] \leq d[v_2]$$

  ma allora si ha:

  $$d[v_k] \leq d[v_1] + 1 \leq d[v_2] + 1$$

  e le restanti disuguaglianze restano inalterate. L'inserimento di un nuovo vertice richiede invece un esame più attento del codice. Quando inseriamo nella coda un vertice $v$, questo diventa l'ultimo elemento della coda, ovvero $v_{k+1}$ e si pone:

  $$d[v_{k+1}] = d[x] + 1$$

  dove $x$ rappresenta la testa della coda prima dell'inserimento di $v$, il che dimostra la prima proprietà. Per la seconda proprietà bisogna dimostrare che:

  $$d[v_k] \leq d[v_{k+1}]$$

  Chiaramente, prima dell'accodamento del vertice $v$ valeva la seguente relazione per vertice $v_k$:

  $$d[v_k] \leq d[v_1] + 1 = d[v_{k+1}]$$

  che è la tesi della seconda proprietà.

■

**Proprietà C**

È possibile partizionare l'insieme dei vertici $V$ in varie partizioni $V_0, V_1, \ldots, V_k$ tali che:
- $V_0 = \{s\}$;
- $V_1$ contiene i vertici adiacenti di $s$ meno $s$ stesso;
- $V_2$ contiene i vertici adiacenti di $V_{i-1}$ meno i vertici già presenti in $V_0, V_1$.
- $V_i$ contiene i vertici adiacenti di $V_{i-1}$ meno i vertici già presenti in $V_0, V_1, \ldots, V_{i-1}$.
- $V_\infty$ contiene i vertici non raggiungibili da $s$.

Si osserva che un vertice non raggiungibile $v$ non può mai essere accodato. Infatti, se per assurdo si suppone che $v$ sia il primo vertice non raggiungibile ad essere accodato, allora deve esistere un vertice $x \in Q$ per cui $(x, v) \in E$. Essendo $x$ un vertice raggiungibile allora esiste sicuramente un percorso dalla sorgente al vertice $v$ che passa per $x$. Ma allora $v$ è raggiungibile, il che è assurdo. Quindi, se $v$ non è raggiungibile, allora $v$ non può essere accodato. Grazie a questa osservazione è possibile dimostrare la correttezza dell'algoritmo BFS semplicemente dimostrando il seguente lemma.

**Lemma**

Per ogni $i \in \mathbb{N}$, per ogni vertice $v \in V_i$ vale che esiste un unico istante durante l'esecuzione dell'algoritmo BFS tale per cui:
1. $v$ viene colorato di grigio e aggiunto alla coda $Q$;
2. $d[v] = i$;
3. il predecessore di $v$ è impostato ad un vertice $u \in V_{i-1}$ e $(p[v], v) \in E$ con $v \neq s$.

**Dimostrazione** Si dimostra per induzione su $i$:

1. **Caso base:** $i = 0$ e $v = s$. All'inizio dell'esecuzione di BFS$(G, s)$, $Q$ contiene soltanto il vertice $s$ e le tre proprietà sono banalmente vere.
2. **Caso induttivo:** sia $i > 0$ e si supponga il lemma vero per $V_j$ con $j < i$. Se $v \in V_i$ arbitrario allora esiste per forza un momento in cui questo verrà colorato di grigio ed accodato. Ciò significa che esiste un percorso da $s$ a $v$ che passa per un

vertice $x$ che è stato accodato al più $i-1$-esimo accodamento. Quindi, $x \in V_{i-1}$. Per ipotesi induttiva allora $d[x] = i-1$ e $p[x] \in V_{i-2}$. Poiché $x$ è stato accodato al più $i-1$-esimo accodamento, allora $v$ è stato accodato al $i$-esimo accodamento. Quindi, $d[v] = d[x] + 1 = (i-1) + 1 = i$ e $p[v] = x$.

■

## 7.5 VISITA IN PROFONDITÀ

A differenza della visita in ampiezza che si propone di visitare un grafo per distanze crescenti, la visita in profondità (DEPTH-FIRST-SEARCH) si propone di visitare un grafo esplorando tutti i nodi visitabili lungo i vari percorsi che si diramano a partire da un nodo sorgente $s$.

La visita in profondità è stata già descritta in relazione alle strutture dati ramificate come gli alberi dove, a differenza dei grafi, è implicita nella struttura il concetto di *figlio sinistro* e *figlio destro*. In un grafo, invece, non esiste alcuna relazione gerarchica tra i vari nodi e sarà quindi necessario adottare una strategia diversa da quella utilizzata per la visita in ampiezza che faceva uso di una coda per memorizzare i nodi da visitare.

Nella visita in profondità, ci avvaleremo di quattro array ausiliari per memorizzare lo stato dei nodi del grafo:

1. `Color[]` che memorizza lo stato dei nodi codificati per colore: `White` se il nodo non è stato ancora visitato, `Gray` se il nodo è stato visitato ma non tutti i suoi vicini sono stati visitati, `Black` se il nodo è stato visitato e tutti i suoi vicini sono stati visitati;
2. `d[]` che memorizza il tempo di scoperta di un nodo;
3. `f[]` che memorizza il tempo di completamento di un nodo;
4. `p[]` che memorizza il predecessore di un nodo.

Per memorizzare i vari istanti di tempo assumiamo di dichiarare una variabile globale `time` che verrà incrementata ad ogni chiamata ricorsiva della funzione DFS-VISIT. Si hanno quindi i seguenti algoritmi: `Init(G)`, `DFS(G)` e `DFS-Visit(G,u)`. L'algoritmo `Init(G)` inizializza gli array ausiliari `pred[]` e `color[]`, mentre l'algoritmo `DFS(G)` esegue la visita in profondità del grafo $G$ richiamando la funzione `DFS-Visit(G,u)` per ogni nodo $u \in V$ che non è ancora stato visitato.

```
1  for each v ∈ V do
2    Color[v] = White
3    Pred[v] = nil
4  time = 1
```

Algoritmo 7.4: Init(G)

```
1  Init(G)
2  for each v ∈ V do
3    if Color[v] = White then
4      DFS-Visit(G,v)
```

Algoritmo 7.5: DFS(G)

```
1  Color[s] = Gray
2  time = time+1
3  d[s] = time
4  for each v ∈ Adj(s) do
5    if Color[v] = White then
6      Pred[v] = s
7      DFS-Visit(G,v)
8  time = time+1
9  f[s] = time
10 Color[s] = Black
```

Algoritmo 7.6: DFS-Visit(G,s)

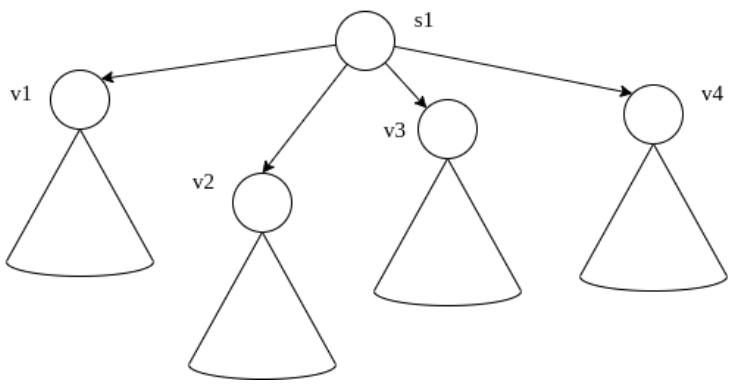



Figura 7.6: Foresta generata dalla visita in profondità di un grafo.

Come nella visita in ampiezza, quando un vertice $v$ viene scoperto durante un'ispezione della lista di adiacenza di un vertice $s$ già scoperto, la visita in profondità registra questo evento assegnando $pred[v] = s$. Diversamente dalla visita in ampiezza però, dove l'ispezione delle liste di adiacenza produceva un albero dei predecessori (l'albero BF), la visita in profondità genera una foresta di alberi, detta **foresta depth first (DF)** (vedi Figura 7.6), che è composta da vari **alberi depth first (DF)** in quanto la visita può essere ripetuta da più sorgenti. L'albero DF di un vertice $s$ è composto da tutti i vertici raggiungibili da $s$ e da tutti gli archi $(s, v)$ tali che $v$ è il primo vertice scoperto durante la visita di $s$.

Volendo fare una analisi del tempo di esecuzione dell'algoritmo DFSsi osserva che ciascuna chiamata ricorsiva della funzione DFS-Visit richiede un tempo costante per inizializzare il colore del vertice $u$ e un tempo lineare sulla dimensione della lista di adiacenza di $u$ per ispezionarla completamente. Poiché la somma delle lunghezze di tutte le liste di adiacenza è pari a $|E|$, il tempo totale richiesto per ispezionare tutte le liste di adiacenza è lineare su tale grandezza. Inoltre, poiché ogni vertice viene visitato una sola volta, si avrà che il numero totale di chiamate sia lineare sul numero dei vertici del grafo. Quindi, il tempo totale richiesto per eseguire la visita in profondità di un grafo $G = (V, E)$ è pari a $|V| + |E|$.

**Osservazione**

La visita in profondità non garantisce la scoperta dei percorsi minimi da $s$ a tutti i vertici raggiungibili. Infatti, il percorso seguito dall'algoritmo dipende dalla scelta dell'ordine in cui vengono esplorati i vertici nelle liste di adiacenza. Al contrario, la visita in ampiezza garantisce la scoperta dei percorsi minimi da $s$ a tutti i vertici raggiungibili in quanto questa procede in maniera sistematica seguendo distanze progressive.

### 7.5.1 Proprietà della visita in profondità

Come già detto in precedenza, la visita in profondità genera un **sottografo dei predecessori** $G'$ che prende il nome di **foresta depth first**. Poniamo $G'$ come segue:

$$
\begin{aligned}
G' &= (V', E') \\
V' &= V \\
E' &= \{(p[v], v) : v \in V \setminus \{s\} \wedge p[v] \neq nil\}
\end{aligned}
$$

Come nella visita in ampiezza, i vertici vengono colorati durante la visita in profondità per indicare il loro stato. Inizialmente tutti i vertici sono bianchi. Un vertice diventa grigio quando viene **scoperto** durante la visita; diventa nero quando viene **completato**, ovvero quando la sua lista di adiacenza è stata completamente ispezionata. Questa tecnica garantisce che ogni vertice vada a finire in un solo albero DF, in modo che questi alberi siano disgiunti.

Va notato inoltre che la struttura ad albero di un albero DF è determinata dalla sequenza di chiamate ricorsive di DFS-Visit. Infatti, ogni volta che viene scoperto un vertice $v$ durante la visita di un vertice $u$, il vertice $v$ diventa un figlio di $u$ nell'albero DF. Inoltre, il vertice $u$ diventa il predecessore di $v$ nell'albero DF. Per questo motivo, la struttura della foresta DF rispecchia esattamente la struttura delle chiamate ricorsive di DFS-Visit.

Un'altra importante proprietà della visita in profondità è che i tempi di scoperta e completamento hanno una **struttura di parentesi**. Se rappresentiamo la scoperta del vertice $u$ con una parentesi aperta "$(u$" e il suo completamento con una parentesi chiusa "$u)$", allora la storia delle scoperte e dei completamenti produce un'espressione ben formata, nel senso che le parentesi sono opportunamente annidate.

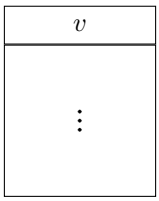
**Teorema della struttura a parentesi**

Dato $G = (V, E)$, al termine di DFS$(G)$, per ogni coppia di vertici $u$ e $v$ vale una delle seguenti condizioni:

1. $d[v] < d[u] < f[u] < f[v]$;
2. $d[u] < d[v] < f[v] < f[u]$;
3. $d[v] < f[v] < d[u] < f[u]$;
4. $d[u] < f[u] < d[v] < f[v]$;

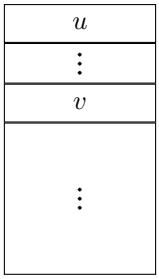
**Dimostrazione** Dimostriamo il teorema mostrando che la condizione $d[v] < d[u] < f[v] < f[u]$ non può mai verificarsi. Ragioniamo quindi per assurdo ed ipotizziamo che tale condizione possa verificarsi. Per questo motivo esisterà un momento nell'esecuzione dell'algoritmo in cui verrà assegnato il tempo $d[v]$.

Consideriamo quindi uno stack sul quale vengono inseriti i vari record di attivazione per ciascuna chiamata di DFS-Visit. Quando viene eseguita la chiamata DFS-Visit$(G, v)$, ovvero nell'istante $d[v]$ il record di attivazione per la chiamata viene inserito in cima alla pila:




Istante $d[v]$

Più avanti con le chiamate si arriva all'istante $d[u]$ e il record di attivazione della chiamata DFS-Visit$(G, u)$ viene inserito in cima alla pila:




Istante $d[u]$

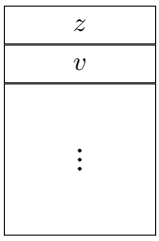
A questo punto si dovrebbe assegnare l'algoritmo di fine visita del vertice $v$ e per farlo si dovrebbe avere il record della chiamata DFS-Visit$(G, v)$ in cima allo stack ma al suo posto c'è il record del vertice $u$. Per terminare $v$ quindi si potrebbe pensare o di aggiungere un nuovo record per DFS-Visit$(G, v)$ in cima allo stack oppure di rimuovere il record del vertice $u$. Entrambe le soluzioni, però, non sono ammesse in quanto, come sappiamo, ogni vertice entra nello stack una ed una sola volta ed inoltre, per rimuovere il record del vertice $u$ andrebbe prima assegnato il suo tempo di completamento (ci troviamo ancora in un istante compreso tra $d[u]$ ed $f[u]$). Per questo motivo questa sequenza è impossibile da ottenere. ■

**Corollario (Annidamento degli intervalli dei discendenti)**

Per ogni vertice $u, v \in V$, il vertice $u$ è discendente di $v$ nella foresta DF se e soltanto se:

$$d[v] < d[u] < f[u] < f[v] \tag{7.3}$$

**Dimostrazione** Siano $v$ e $z$ due vertici di $V$ si supponga di avere la seguente condizione sullo stack:




Ciò significa che la chiamata a DFS-Visit$(G, v)$ ha generato la chiamata sul vertice $z$. Allora esisterà sicuramente un arco $(v, z) \in E$ che corrisponderà ad un arco nel sottografo dei predecessori. Allora sicuramente $p[z] = v$. Dato lo stack di attivazione possiamo mappare quindi ogni sequenza di vertici con una sequenza nella foresta depth first. In questo modo si dimostra che la condizione è sufficiente.

Sia ora $u$ discendente di $v$ nella foresta depth first e ragioniamo per induzione sulla lunghezza del percorso $\pi$ da $v$ ad $u$:

- **Caso base:** sia $|\pi| = 1$ allora esiste un solo arco tra $v$ ed $u$ e $p[u] = v$. Se $p[u] = v$ allora sicuramente si avrà la sequenza $d[v] < d[u] < f[u] < f[v]$ per il teorema della struttura parentesi.
- **Caso induttivo:** sia $|\pi| > 1$ e sia l'implicazione valida per $|\pi| - 1$. Possiamo decomporre $\pi$ in due parti:

  $$\pi = \underbrace{\langle v \dots z \rangle}_{\pi'} + \langle z, u \rangle$$

  Chiaramente $\pi'$ ha lunghezza $|\pi| - 1$, quindi per ipotesi induttiva vale $d[v] < d[z] < f[z] < f[v]$. Resta da capire dove si collocano gli istanti $d[u]$ e $f[u]$ all'interno di tale sequenza. Osserviamo però che l'arco $\langle z, u \rangle$ ha dimensione uno quindi $p[u] = z$ e vale:

  $$d[v] < d[z] < d[u] < f[u] < f[z] < f[v]$$

  per il teorema della chiusura parentesi.

■

**Teorema del percorso bianco**

Dato un grafo $G = (V, E)$ ed eseguita una visita in profondità su $G$ allora, per ogni vertice $u, v \in V$ con $v \neq u$ vale che $u$ è discendente di $v$ nella foresta depth first se e soltanto se all'istante $d[v]$ esiste un percorso in $G$ fatto solo di vertici bianchi da $v$ ad $u$.

**Dimostrazione** ( $\Longrightarrow$ :) partiamo dall'ipotesi che il vertice $u$ diventi discendente di $v$ all'interno della foresta depth first. Bisogna dimostrare allora che esiste almeno un percorso bianco[4] da $v$ ad $u$. Chiaramente, il percorso da $v$ ad $u$ nella foresta (che coincide con un percorso nel grafo) all'istante $d[v]$ è tutto bianco. Consideriamo infatti un generico vertice $z$ all'interno del percorso da $v$ ad $u$. Se $z$ è discendente di $v$ allora per il corollario 7.3 vale:

$$d[v] < d[z] < f[z] < f[v]$$

e ciò significa che all'istante iniziale $d[v]$ tale vertice sarà per forza bianco non essendo stato ancora scoperto.

( $\Longleftarrow$ :) Esista un percorso bianco da $v$ a $u$ nel grafo $G$ e dimostriamo che ogni vertice in $\pi$ diventa discendente di $v$. Per assurdo, esista almeno un vertice che non sia discendente di $v$. Sia $t$ il primo vertice non discendente di $v$ incontrato lungo il percorso da $v$ ad $u$ nella foresta depth first. Se $t$ è il primo vertice non raggiungibile chiaramente il suo predecessore lo sarà, sia esso il vertice $z$. Quindi vale sicuramente:

$$d[v] < d[z] < f[z] < f[v]$$

Poiché $z$ è il predecessore di $t$ esiste un arco $(z, t)$ nella FDF. Per il teorema della struttura parentesi però, $t$ non può essere scoperto prima di $d[v]$, non può essere scoperto dopo la fine di $f[z]$ e quindi deve per forza essere scoperto dopo l'inizio di $d[v]$ il che implica che $t$ è discendente di $v$ nella foresta depth first, il che è assurdo. ■

**Osservazione**

Anche se esistesse un arco da $z$ a $t$ non si può dire che $d[z] < d[t]$ in quanto potrebbe esistere un arco da $v$ a $t$ che viene esplorato prima.

### 7.5.2 Caratterizzazione degli archi

Possiamo definire quattro tipi di archi in base alla foresta depth first prodotta da una visita in profondità del grafo $G$:

1. **Archi d'albero:** sono gli archi nella foresta depth first. L'arco $(u, v)$ è un arco d'albero se $v$ viene scoperto la prima volta durante l'esplorazione di $(u, v)$.
2. **Archi all'indietro:** sono quegli archi $(u, v)$ che collegano un vertice $u$ a un antenato $v$ in un albero DF. I cappi, che possono presentarsi nei grafi orientati, sono considerati archi all'indietro.
3. **Archi in avanti:** sono gli archi $(u, v)$ (diversi dagli archi d'albero) che collegano un vertice $u$ ad un discendente $v$ in un albero DF.
4. **Archi di attraversamento:** tutti gli altri archi. Possono connettere i vertici nello stesso albero DF, purché un vertice non sia un antenato dall'altro, oppure possono connettere i vertici di alberi DF differenti.

Ad esempio, si consideri il grafo mostrato in Figura 7.7a e sia 1 il primo vertice dell'insieme $V$, allora la visita in profondità genera il sottografo dei predecessori mostrato in Figura 7.7b dove:

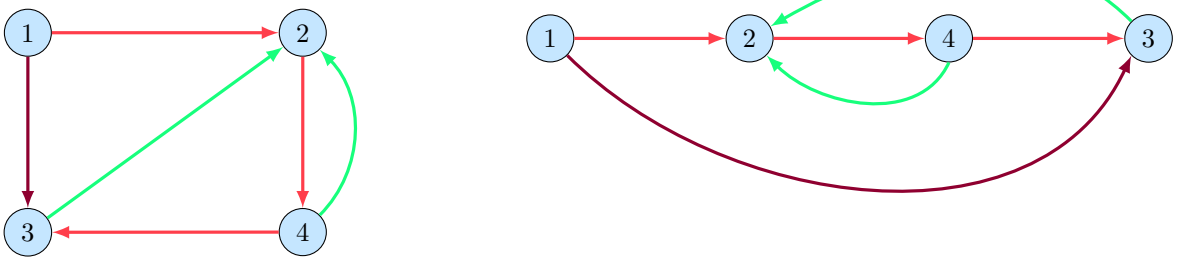
- gli archi rossi sono gli archi d'albero;
- gli archi verde chiaro sono gli archi all'indietro;
- gli archi porpora sono gli archi in avanti
- gli archi verde scuro sono gli archi di attraversamento.

Si consideri il grafo mostrato in Figura 7.8a. Supposto 2 il primo vertice del grafo, eseguendo la BFS si ottiene l'albero DF mostrato in Figura 7.8b.

L'algoritmo DFS può essere utilizzato per classificare gli archi di un grafo $G$. In particolare, si usa il colore del vertice che si raggiunge durante la visita dell'arco $(u, v)$ per effettuare la discriminazione tra le varie tipologie:

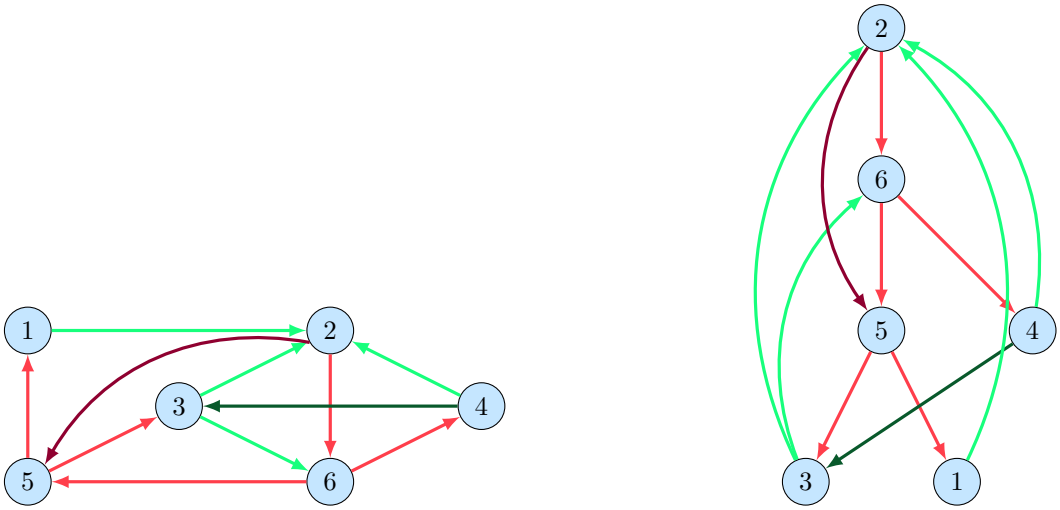
- Se $v$ è bianco allora l'arco è un **arco d'albero**;
- Se $v$ è grigio allora l'arco è un **arco di ritorno**;

[4] Non è detto che questo sia unico infatti.




(a) (b)

Figura 7.7




(a) (b)

Figura 7.8

- Se $v$ è nero allora l'arco è un **arco in avanti** o un **arco di attraversamento**:
  - se inoltre $d[u] < d[v]$ allora è un **arco in avanti**;
  - se inoltre $d[v] < d[u]$ allora è un **arco di attraversamento**.

È possibile quindi modificare l'algoritmo DFS-Visit$(G, s)$ per visualizzare la tipologia di tutti gli archi ispezionati:

```
1  Color[s] = Gray
2  time = time+1
3  d[s] = time
4  for each v ∈ Adj(s) do
5    if Color[v] = White then
6      print("(s,v) e' d'albero")
7      Pred[v] = s
8      DFS-Visit(G,v)
9    else
10     if Color[v] = Gray then
11       print("(s,v) e' all'indietro")
12     else if(d[s]<d[v]) then
13       print("(s,v) e' in avanti")
14     else
15       print("(s,v) e' d'attraversamento")
16 time = time+1
17 f[s] = time
18 Color[s] = Black
```

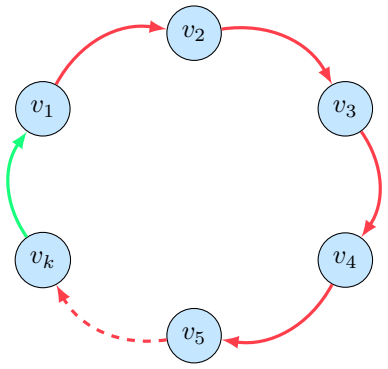
Algoritmo 7.7: Print-DFS-Visit(G,s)

### 7.5.3 Verifica dei grafi aciclici

Grazie alla caratterizzazione degli archi appena visti possiamo osservare che l'algoritmo DFS è in grado di verificare se un grafo è **ciclico** o meno. Se si osservano attentamente i grafi mostrati in Figura 7.7a e 7.8a notiamo infatti la presenza di vari cicli che

vengono descritti dagli archi etichettati di verde chiaro, ovvero gli archi all'indietro.

Assumiamo che $G = (V, E)$ sia un grafo orientato in cui ci sia un ciclo semplice $\langle v_1, v_2, \ldots, v_k \rangle$ e sia $v_1$ il primo vertice scoperto dalla BFS. Nell'istante $d[v_1]$ gli altri vertici sono bianchi, quindi, per il teorema del percorso bianco esiste un percorso bianco da $v_1$ a $v_2, v_2, \ldots, v_i$ per $i = \{2, 3, \ldots, k\}$. Se esiste tale percorso bianco allora i vertici $v_2, v_3, \ldots, v_i$ sono discendenti di $v_1$ e gli archi di tale percorso sono archi dell'albero. Una volta arrivati al vertice $v_k$ l'unico arco ancora da percorrere è quello che lo conduce all'antenato $v_1$ e tale arco risulta quindi un arco all'indietro.




È possibile quindi definire un algoritmo il quale, dato un grafo orientato, restituisca `true` se aciclico e `false` altrimenti.

```
1  Init(G)
2  for each v ∈ V do
3    if Color[v] = White then
4      ret = Aciclico-Visit(G,v)
5      if ret = false then
6        return false
7  return true
```

Algoritmo 7.8: ACICLICO(G)

```
1  Color[s] = Gray
2  for each v ∈ Adj(s) do
3    if Color[v] = White then
4      ret = Aciclico-Visit(G,v)
5      if ret = false then
6        return false
7    else
8      if Color[v] = Gray then
9        return false
10 return true
```

Algoritmo 7.9: ACICLICO-VISIT(G,s)

### 7.6.1 Relazione d'ordine nei grafi

**Ordinamento topologico**

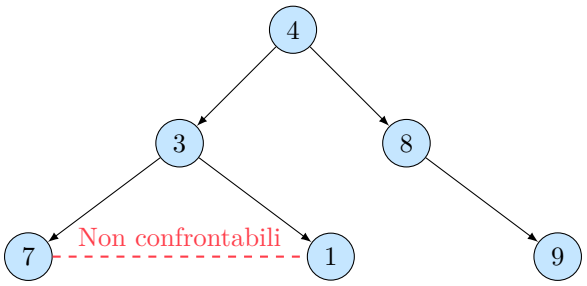
Sia $G = (V, E)$ un grafo orientato aciclico, si definisce **ordinamento topologico** di $G$ una permutazione $\pi$ di $V$ tale che:

$$\forall (v, u) \in E, \quad v \text{ precede } u \text{ in } \pi \tag{7.4}$$

Se due vertici $v$ e $u$ sono presenti in un ordinamento topologico allora si diranno *confrontabili* e lo si indica con $v < u$.

Esempio

Si consideri il seguente grafo:



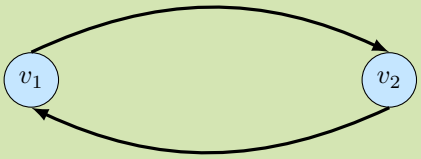

È possibile ricavare le seguenti relazioni:

$$\begin{array}{ll} 4 < 3 & 4 < 8 \\ 4 < 7 & 4 < 1 \\ 4 < 9 & 3 < 7 \\ 3 < 1 & 8 < 9 \end{array}$$

Notare il fatto che non è possibile determinare alcuna relazione tra i vertici 7 ed 1 oppure 3 ed 8 in quanto non esiste alcun arco in $G$ che li collega. In generale, infatti, la struttura del grafo induce una relazione d'ordine *parziale*.

**Osservazione**

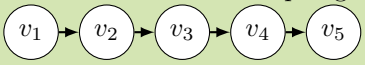
Se nel grafo sono presenti cicli non è possibile determinare alcun ordinamento topologico. Infatti, preso ad esempio il seguente grafo:




non è possibile determinare quale nodo preceda l'altro all'interno di un possibile ordinamento. Lo stesso ragionamento vale anche nel caso in cui si considerano i grafi non orientati dove non è definito il concetto di precedenza. Per questo motivo, la nozione di ordinamento topologico si applica solo ai grafi orientati aciclici.

**Osservazione**

Dato un grafo orientato $G = (V, E)$ è possibile ricavare più di un ordinamento topologico. Nel caso in cui l'insieme $E$ sia vuoto (grafo senza archi) allora il numero di permutazioni possibili corrisponde a $|V|!$. Nel caso in cui il grafo si riduca ad una sequenza di vertici è possibile avere un unico ordinamento topologico. Si consideri ad esempio il seguente grafo:




Tale grafo ammette un unico ordinamento topologico dato dalla permutazione $\pi = \langle v_1, v_2, v_3, v_4, v_5 \rangle$. Tale ordinamento non è solo l'unico ma anche totale.

### 7.6.2 Proprietà dei grafi aciclici

**Proposizione**

Se $G = (V, E)$ è un grafo aciclico e $G'$ è un suo sottografo allora $G'$ è aciclico.

**Dimostrazione** Per ipotesi si ha $G' = (V', E')$, con:

$$V' \subseteq V$$
$$E' \subseteq E \cap (V' \times V')$$

Supponiamo per assurdo che $G'$ sia un grafo ciclico e consideriamo un percorso ciclico di $G'$, sia esso $\pi$:

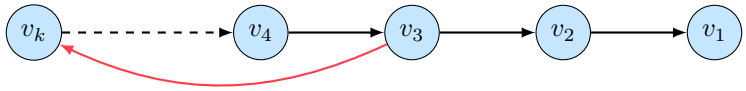
$$\pi = \langle v_1, v_2, \cdots, v_k, v_1 \rangle$$

Chiaramente ciascun arco nel percorso $\pi$ è un arco presente in $E'$. Poiché $E'$ è un sottoinsieme di $E$, quindi ogni arco di $\pi$ appartiene ad $E$ e $\pi$ risulta quindi un percorso in $G$ che di fatto risulta ciclico, contro le nostre ipotesi. ■

**Proposizione**

Se $G$ è un grafo aciclico allora esiste almeno un vertice $v \in V$ tale che $v$ ha grado entrante pari a 0.

**Dimostrazione** Ragionando per contrapposizione supponiamo che per ogni vertice di $V$ ci sia almeno un arco entrante. Senza ledere di generalità consideriamo un vertice di $V$, tale $v_1$. Per l'ipotesi appena posta esiste almeno un arco che entra in $v_1$ ed esiste quindi un vertice $v_2 \neq v_1$ dal quale si diparte tale arco. Ragionando allo stesso modo possiamo considerare un arco che parta da un vertice $v_3$ e arrivi in $v_2$ e così via fino ad arrivare all'ultimo vertice $v_n \in V$. Poiché il grado entrante di $v_n$ è maggiore di 1 ciò significa che esiste almeno un arco che parte da uno dei vertici $\{v_1, \ldots, v_{n-1}\}$ (non potrebbe essere altrimenti) e arriva in $v_n$. Ciò, però, dimostra l'esistenza di un percorso ciclico nel grafo $G$, il quale per ipotesi era stato definito aciclico. Mostrato l'assurdo si dimostra l'enunciato.




■

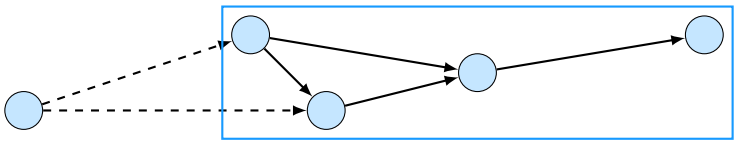
**Proposizione**

Dato un grafo aciclico $G = (V, E)$, se $G$ è aciclico allora esiste almeno un ordinamento topologico di $G$.

**Dimostrazione** Dato che $G$ è aciclico esiste almeno un vertice $v_1 \in V$ con grado entrante pari a zero. Chiaramente un vertice del genere deve essere posto all'inizio di qualsiasi ordinamento topologico $\pi$ in quanto non esiste alcun vertice che lo preceda. Escluso tale vertice possiamo quindi considerare il sottografo:

$$G_1 = (V \setminus \{v_1\}, E \setminus \{(v_1, u) \in E \mid u \in V\})$$

ottenuto rimuovendo $v_1$ da $V$ e ciascun arco che collegava $v_1$.



$G_1$

Chiaramente, essendo $G_1$ un sottografo di un grafo aciclico, anche $G_1$ risulta aciclico e quindi esisterà un vertice $v_2$ con grado entrante pari a zero. Tale vertice può essere quindi inserito a destra del vertice $v_1$ all'interno dell'ordinamento topologico $\pi$. È possibile iterare questo procedimento fino a quando non si raggiunge il sottografo vuoto, ottenendo così l'ordinamento topologico:

$$\pi = (v_1, v_2, \ldots, v_k)$$

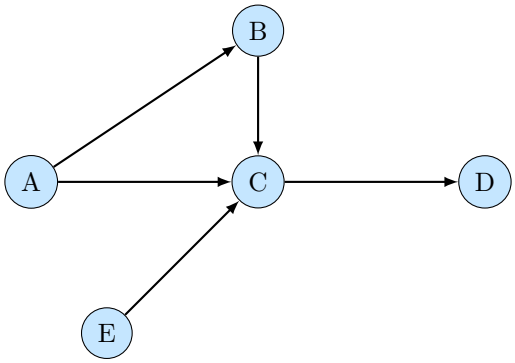
■

**Osservazione**

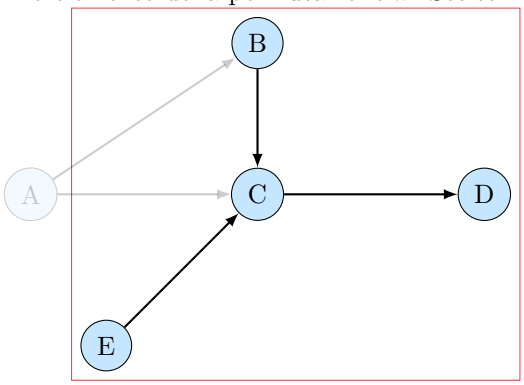
Chiaramente, l'ordinamento topologico così ottenuto dipende dalla scelta del vertice $v_i$ con grado entrante pari a zero il quale può essere più di uno. All'interno di uno stesso grafo, infatti, l'ordine dei vertici con grado entrante pari a zero all'interno di una permutazione è indifferente e possono quindi essere messi uno dopo o prima dell'altro.

Esempio

Consideriamo il seguente grafo:



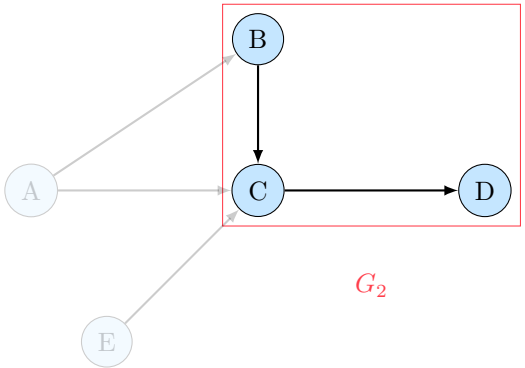

In questo caso esistono due vertici con grafo entrante pari a zero: il vertice A ed il vertice E. Possiamo quindi scegliere uno qualsiasi tra i due come primo elemento della permutazione $\pi$. Scelto il vertice A si ottiene:




$\pi = (A)$

$G_1$

Il sottografo $G_1$ così ottenuto ha ancora due vertici con grado entrante pari a zero: B ed E. Scelto quindi il vertice E si ottiene il nuovo sottografo $G_2$:




$\pi = (A, E)$

$G_2$

Il grafo $G_2$, così come il resto dei suoi sottografi ottenuti iterando il procedimento, hanno un unico vertice con grado entrante pari a zero, è quindi univoca la sequenza dei vertici estratti. Si ottiene così l'ordinamento topologico:

$$\pi = (A, E, B, C, D)$$

## 7.6.3 Algoritmo del grado entrante

Per trasformare la procedura appena studiata in un algoritmo che determini un ordinamento topologico bisogna capire come calcolare il *grado entrante* di ogni vertice per poter selezionare i vertici con grado entrante nullo. Le liste di adiacenza non sono adatte a fornire tale informazione in quanto esprimono (mediante la loro dimensione) soltanto il grado uscente di ogni vertice.

Chiaramente, il grado entrante di un vertice può essere ottenuto dalla dimensione della sua **lista di incidenza**, ovvero una

lista che descrive tutti i vertici dai quali si dipartono degli archi orientati verso il vertice selezionato[5]. La generazione di tali liste, però, risulta essere un'operazione costosa in quanto richiede di scorrere l'intero insieme degli archi. Per questo motivo, si preferisce utilizzare un algoritmo che calcoli il grado entrante di ogni vertice in modo più efficiente. L'Algoritmo 7.11 imposta innanzitutto il grado entrante di ogni vertice a 0 e poi, per ogni vertice $v \in V$, incrementa il grado entrante di tutti i vertici adiacenti ad $v$. Chiaramente, l'algoritmo risulta essere lineare sulla dimensione del grafo.

Calcolato il grado entrante di ciascun vertice è possibile implementare l'algoritmo 7.10 che, partendo dai vertici con grado entrante pari a zero, li inserisce in una coda $Q$ e li estrae uno alla volta. Per ogni vertice estratto $v$ si itera sui suoi adiacenti $u$ e si decrementa il loro grado entrante. Se il grado entrante di $u$ diventa pari a zero allora viene inserito nella coda $Q$. L'algoritmo termina quando la coda $Q$ risulta vuota.

```
1  Q = NIL
2  // Funzione che associa ad ogni vertice il proprio grado entrante
3  Ge = GradoEntrante(G)
4  for each v ∈ V do
5    // Accodo i vertici con grado zero
6    if Ge[v]=0 then
7      Q = Enqueue(Q,v)
8  while (Q ≠ NIL) do
9    v = Head(Q)
10   print(v)
11   // Aggiorno il grado entrante degli adiacenti
12   for each u ∈ Adj[v] do
13     Ge[u] = Ge[u]-1
14     if Ge[u]=0 then
15       Q = Enqueue(Q,u)
16 Q= Dequeue(Q)
```

Algoritmo 7.10: OrdinamentoTopologico(G)

```
1  for each v ∈ V do
2    Ge[v] =0
3  for each v ∈ V do
4    for each u ∈ Adj[v] do
5      Ge[u]=Ge[u]+1
```

Algoritmo 7.11: GradoEntrante(G)

Chiaramente, l'algoritmo 7.10 risulta lineare sulla dimensione del grafo. Infatti:

$$\begin{aligned} T_{OrdinamentoTopologico} &= T_{GradoEntrante} + |V| + (|V| + |E|) \\ &= (|V| + |E|) + |V| + (|V| + |E|) \\ &= 3 \cdot |V| + 2 \cdot |E| \\ &\simeq |V| + |E| \end{aligned}$$

### 7.6.4 Calcolo dell'ordinamento topologico mediante DFS

È possibile calcolare l'ordinamento topologico di un grafo mediante l'algoritmo DFS in modo molto semplice. Infatti, l'ordinamento topologico di un grafo $G$ è semplicemente l'ordinamento inverso dei tempi di fine visita dei vertici di $G$. Infatti, se $G$ è aciclico allora non esistono archi all'indietro e quindi non esistono archi che collegano vertici con tempi di fine visita diversi. Inoltre, se $G$ è aciclico allora esiste almeno un vertice con grado entrante pari a zero e quindi tale vertice sarà il primo ad essere visitato dall'algoritmo DFS. Il vertice successivo da visitare sarà il vertice adiacente al primo e così via. Sfruttando questa osservazione è possibile inserire i vertici in uno stack man mano che questi vengono visitati. Una volta terminata la visita, basterà estrarre i vertici dallo stack per ottenere l'ordinamento topologico. L'algoritmo 7.12 mostra come implementare tale procedura.

Il tempo di esecuzione di tale algoritmo è pari a quello dell'algoritmo DFS. Infatti, l'unico costo aggiuntivo è quello di inserire i vertici nello stack, operazione che richiede tempo costante. L'ordinamento topologico viene quindi calcolato in tempo lineare sulla dimensione del grafo.

```
1  tempo = 0
2  Init(G)
3  S = NIL
4  for each v ∈ V do
5    if Color[v] = White then
6      S = OrdinamentoTopologico-DFS-Visit(G,v,S)
7  return S
```

Algoritmo 7.12: OrdinamentoTopologico-DFS(G)

[5] Qualora il grafo fosse stato implementato mediante matrici di adiacenza sarebbe stato sufficiente calcolare la sua trasposta

```
Color[v] = Gray
time = time+1
d[v] = time
for each u ∈ Adj[v] do
  if Color[u] = White then
    S = OrdinamentoTopologico-DFS-Visit(G,u,S)
time = time+1
f[v] = time
Color[v] = Black
S = Push(S,v)
return S
```

Algoritmo 7.13: OrdinamentoTopologico-DFS-Visit(G,v,S)

## 7.7 Calcolo delle componenti fortemente connesse

Il concetto di **componente fortemente connessa** è strettamente legato al concetto di grafo fortemente connesso.

**Raggiungibilità e connessione**

Un grafo non orientato è **connesso** se ogni coppia di vertici è collegata attraverso un cammino. Le **componenti connesse** di un grafo è un insieme di coppie di vertici che soddisfano la condizione di raggiungibilità. Un grafo orientato è **fortemente connesso** se due vertici qualsiasi sono raggiungibili l'uno con l'altro. Se $G$ è un grafo orientato non fortemente connesso, ed il grafo non orientato sottostante è connesso, allora si dice che $G$ è **debolmente connesso**.

Esempio

Ad esempio, il grafo non orientato $G = (V, E)$ mostrato in Figura 7.9 è un grafo connesso, infatti ogni vertice è raggiungibile da qualsiasi altro vertice. Al contrario, il grafo non orientato mostrato in Figura 7.10 non è un grafo connesso in quanto non tutti i vertici sono reciprocamente raggiungibili, in questo caso il grafo è detto **sconnesso**.

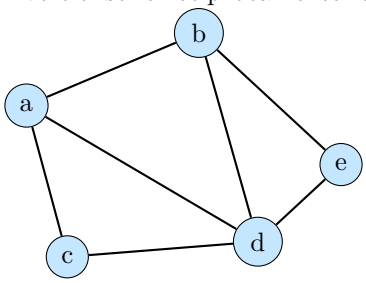



Figura 7.9

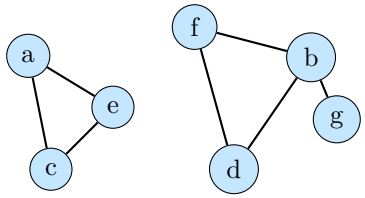



Figura 7.10

**Componente fortemente connessa**

Una **componente fortemente connessa** di un grafo orientato $G = (V, E)$ è un insieme massimale di vertici $C \subseteq V$ tale che per ogni coppia di vertici $u, v \in C$ si ha $u \rightsquigarrow v$ e anche $v \rightsquigarrow u$; ovvero i vertici $u$ e $v$ sono raggiungibili l'uno dall'altro.

**Osservazione**

Se in un grafo orientato esistono vertici mutualmente raggiungibili vuol dire che il grafo è ciclico.

Esempio

Si consideri il grafo $G$ in Figura 7.11. Tale grafo contiene quattro componenti fortemente connesse: $C_1, C_2, C_3$ e $C_4$.

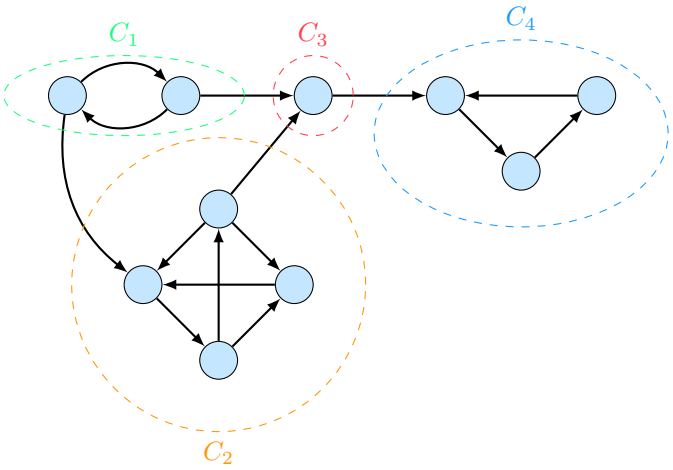



Figura 7.11: Un grafo orientato $G$

La relazione di reciproca raggiungibilità $RR$ in un grafo $G = (V, E)$ è una relazione di equivalenza. Infatti, essa è:

- **riflessiva**: ogni vertice è raggiungibile da se stesso;
- **simmetrica**: se $u$ è raggiungibile da $v$ allora $v$ è raggiungibile da $u$;
- **transitiva**: se $u$ è raggiungibile da $v$ e $v$ è raggiungibile da $w$ allora $u$ è raggiungibile da $w$.

Per questo motivo è possibile definire le **classi di equivalenza** indotte dalla relazione di mutua raggiungibilità. Ogni classe di equivalenza è una componente fortemente connessa. Infatti, se due vertici sono mutualmente raggiungibili allora questi apparterranno alla stessa componente fortemente connessa.

**Grafo delle componenti**

L'insieme quoziente della relazione $RR$, ovvero l'insieme di tutti gli insiemi di vertici mutualmente raggiungibili, è detto **insieme delle componenti fortemente connesse** di $G$ ed è rappresentato mediante il **grafo delle componenti** $G_{CFC}$ il quale è equivalente a $G$:

$$G_{CFC} = (V_{CFC}, E_{CFC}) \tag{7.5}$$

dove $V_{CFC}$ contiene un nodo per ogni componente fortemente connessa di $G$ e

$$E_{CFC} = \{(u, v) \mid u, v \in V_{CFC}, \exists \text{ un arco in } E \text{ da un vertice } u \text{ o un vertice } v\}$$

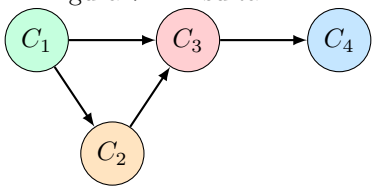
**Proprietà**

Sia $G = (V, E)$ un grafo, allora $u \in V$ è raggiungibile dal vertice $v \in V$ se e soltanto se esiste un percorso da un qualsiasi vertice della componente fortemente connessa di $u$ ad un qualsiasi vertice della componente fortemente connessa di $v$.

Data questa proprietà possiamo osservare che il **problema della raggiungibilità** tra due vertici può essere studiato decomponendolo mediante l'analisi del grafo delle componenti. Bisogna trovare quindi il modo di calcolare le componenti fortemente connesse presenti in un grafo.

Esempio

Il grafo delle componenti del grafo mostrato in Figura 7.11 risulta:




il che rappresenta una forma molto più compatta, che preserva ancora le informazioni relative alla raggiungibilità dei vertici, del grafo $G$.

**Attenzione**

Il grafo delle componenti è **sempre aciclico**. Infatti, se esistesse un ciclo tra due componenti fortemente connesse queste apparterrebbero alla stessa classe di equivalenza del grafo connesso.

**Osservazione**

Nei grafi orientati, la relazione di equivalenza di raggiungibilità e reciproca raggiungibilità non si equivalgono. Si osservino infatti i seguenti grafi, nella loro versione orientata e non:

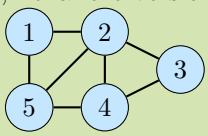



Figura 7.12: Grafo non orientato $A$

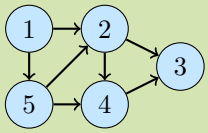



Figura 7.13: Grafo orientato $B$

In un grafo orientato ogni coppia di vertici collegati da un arco è mutualmente raggiungibile non essendo determinato alcuna orientazione e si ha $R = RR$. Al contrario, se si osserva il grafo $B$ si può osservare che i soli ed unici vertici mutualmente raggiungibili sono i singoli vertici. Cioè:

$$R = \{(1,2),(1,5),(5,2),(2,4),(2,3),(4,3),(1,1),(2,2),(3,3),(4,4),(5,5),(5,3),(1,4),(1,3)\}$$
$$RR = \{(1,1),(2,2),(3,3),(4,4),(5,5)\}$$

e vale $RR \subseteq R$. Nei grafi non orientati è sufficiente quindi verificare la raggiungibilità tra due vertici per sapere se appartengono alla stessa componente fortemente connessa.

Poiché la relazione di reciproca raggiungibilità determina una partizione nell'insieme dei vertici $V$ di un grafo $G$, è possibile quindi considerare per ogni vertice $v \in V$ il sottografo $CFC(v)$ (che corrisponde ad una classe di equivalenza), ovvero la componente fortemente connessa che contiene $v$ tra i suoi vertici (un vertice del grafo delle componenti). Chiaramente vale $CFC(v) = (V_v, E_v)$ dove:

$$V_v = \{u \in V \mid u \text{ è reciprocamente raggiungibile con } v\}$$

ed:

$$E_v = E \cap (V_u \times V_u)$$

Per costruire il grafo delle componenti quindi è sufficiente trovare l'insieme $V_v$ (il calcolo dell'insieme $E_v$ è banale una volta determinato $V_v$) per ogni vertice del grafo $G$. Ogni componente fortemente connessa rappresenta infatti un **sottografo indotto** dalla scelta dei suoi vertici.

### 7.7.1 Calcolo delle componenti connesse in un grafo non orientato

Dato un grafo non orientato $G$, è possibile pensare di applicare una visita in profondità (Algoritmo 7.5) per risolvere il problema della raggiungibilità (e per l'osservazione precedente anche il problema della mutua raggiungibilità). Tipicamente, infatti, la DFS costruisce su $G$ una foresta, chiamata Foresta Depth First (FDF) o anche sottografo dei predecessori, costituita da alberi DF (Depth First) . Chiaramente, tutti i vertici che appartengono ad uno stesso albero DF formano una componente connessa del grafo $G$. Possiamo quindi pensare di mappare ciascun vertice con il rispettivo albero DF di appartenenza (numerati da 1 a $k$), ottenendo così la corrispondenza che associa a ciascun vertice il "numero" dell'albero DF di appartenenza:

$$CC : V \longrightarrow \mathbb{N}$$

e ricavare in questo modo, per ogni $v \in V$ l'insieme $V_v$ ricercato.

$$V_v = \{u \in V \mid CC[u] = CC[v]\}$$

Sarà quindi sufficiente modificare l'algoritmo DFS-Visit in modo tale che, per ogni vertice scoperto lungo l'esplorazione del grafo, gli si associ il numero dell'albero DF che si sta attualmente costruendo.

Notiamo che la complessità temporale del meccanismo appena descritto sarà dato dal tempo della DFS che è lineare sul grafo $G$ più il tempo necessario a costruire l'insieme $V_v$ ovvero $k \cdot |V|$ al quale si aggiunge un costo $k \cdot |E|$ per la costruzione dei sottografi:

$$T_{DFS_{CC}} = |V| + |E| + \big((k \cdot |V|) + (k \cdot |E|)\big)$$

dove $k$ rappresenta il numero totale di componenti connesse ottenute e vale:

$$1 \leq k \leq |V|$$

### 7.7.2 Proprietà delle componenti fortemente connesse

**Teorema 1**

Se due vertici compaiono nella stessa componente fortemente connessa, allora nessun percorso tra i due vertici esce da quella componente.

**Dimostrazione** Siano $u$ e $v$ due vertici appartenenti alla stessa componente fortemente connessa. Allora esistono percorsi sia da $v$ a $u$ che da $u$ a $v$. Sia $w$ un vertice lungo qualche percorso $u \rightsquigarrow w \rightsquigarrow v$. Poiché c'è un percorso $v \rightsquigarrow u$, $u$ è raggiungibile da $w$ tramite $w \rightsquigarrow v \rightsquigarrow u$. Quindi $w$ e $u$ sono nella stessa componente fortemente connessa. Dall'arbitrarietà di $w$ segue l'enunciato. ■

**Teorema 2**

In ogni visita DFS, tutti i vertici nella stessa componente fortemente connessa compaiono nello stesso albero depth first.

**Dimostrazione** Sia $r$ il primo vertice di una componente fortemente connessa, visitato da DFS (Algoritmo 7.5). Poiché esso è il primo vertice ad essere scoperto allora, per il Teorema del percorso bianco (vedi 7.5.1), tutti gli altri vertici nella componente fortemente connessa devono essere ancora bianchi.

Chiaramente, per definizione di componente fortemente connessa, esiste un percorso da $r$ a tutti gli altri vertici della componente fortemente connessa. Infatti questi percorsi non escono mai dalla componente fortemente connessa di $r$ per il teorema precedente e i vertici di tutti i percorsi nella componente fortemente connessa sono bianchi. Quindi, per il teorema del percorso bianco, ogni vertice nella componente fortemente connessa sarà un discendente di $r$ nell'albero depth first. ■

### 7.7.3 Calcolo delle componenti fortemente connesse in un grafo orientato

Grazie alle proprietà della visita in profondità, il calcolo delle componenti connesse risulta immediato nei grafi non orientati in quanto è stato sufficiente associare ad ogni vertice il numero dell'albero DF di appartenenza ed in base a tale numero costruire i vari sottografi $CFC(v)$.

Stessa cosa non si può dire per i grafi orientati in quanto la visita in profondità da sola non è in grado di fornire le informazioni necessarie per calcolare le componenti fortemente connesse. Infatti, il sottografo dei predecessori descrive soltanto quali vertici possono essere raggiunti dal vertice che li scopre ma non il viceversa.

Dal Teorema 2, però, osserviamo che una componente connessa non può essere divisa in due alberi DF. Quindi tutti e soli i vertici appartenenti alla componente fortemente connessa della radice si troveranno sempre e solo in unico albero depth first. Chiaramente, in un singolo albero depth first è possibile avere più componenti fortemente connesse. Per identificare una componente fortemente connessa all'interno di un albero depth first è necessario sapere quali vertici dell'albero possono raggiungere la radice $r$ dell'albero DF $T_r$ che li ha scoperti:

$$CFC(r) = \{v \in T_r \mid v \rightsquigarrow r\}$$

Una volta identificata la componente fortemente connessa della radice basterà separare i vertici così ottenuti dal resto dell'albero $T_r$ e considerare il sottoalbero $T_{r'}$ che si ottiene rimuovendo i vertici della componente fortemente connessa e ripetere il ragionamento con la nuova radice così ottenuta.

Esempio

Si consideri il seguente grafo orientato mostrato in Figura 7.14. Eseguendo una visita in profondità si ottengono gli alberi DF mostrati in Figura 7.15 e 7.16.

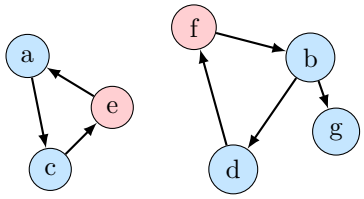



Figura 7.14

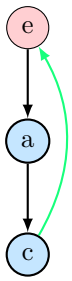



Figura 7.15

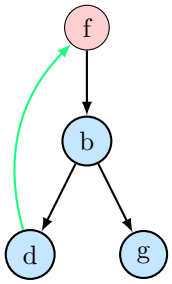



Figura 7.16

Nel primo albero si osserva che tutti i vertici possono raggiungere la radice $e$ e vale quindi $CFC(e) = \{a, e, c\}$. Nel secondo albero, invece, solo i vertici $b$ e $d$ sono in grado di raggiungere la radice $f$ e vale quindi $CFC(f) = \{b, d, f\}$ mentre il vertice $g$ rappresenta la seconda CFC presente nell'albero, $CFC(g) = \{g\}$.

Per calcolare quali vertici sono in grado di raggiungere la radice $r$ di un albero DF in maniera efficiente è possibile utilizzare il concetto di **grafo trasposto**.

**Grafo trasposto**

Si consideri un grafo orientato $G = (V, E)$, si definisce **grafo trasposto** il grafo il grafo $G^T = (V, E^T)$ ottenuto invertendo l'orientamento degli archi del grafo $G = (V, E)$.

$$E^T = \{(u, v) \ : \ (v, u) \in E\}$$

Esempio

Considerato il grafo $G$ visto nella Figura 7.14, il grafo trasposto $G^T$ risulta:

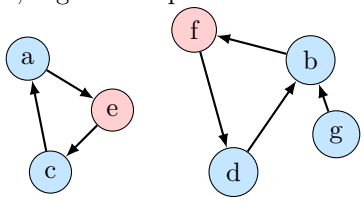



Figura 7.17

**Teorema 3**

Se in un grafo orientato $G$ esiste un cammino da $u$ a $v$, allora in $G^T$ esiste un cammino da $v$ a $u$, e viceversa.

**Corollario**

Se $G$ è un grafo orientato e $G^T$ è il grafo trasposto di $G$, allora per ogni vertice $v$ di $G$ vale:

$$CFC(v) = CFC^T(v)$$

Dato il risultato descritto dal Teorema 3 si potrebbe pensare di applicare una DFS nel grafo $G^T$ a partire dalle radici $r_i$ degli alberi depth first ottenuti dalla DFS sul grafo $G$ in modo tale da trovare i vertici che raggiungono tali vertici $r_i$. Non è detto però a priori che i vertici così raggiunti appartengano alla stessa componente fortemente connessa del vertice $r_i$.

Infatti è possibile che la visita DFS nel grafo trasposto a partire da un vertice $r_i$ possa saltare da un albero depth first[6] all'altro. Si consideri ad esempio il grafo mostrato in Figura 7.18. Applicando una visita in profondità a partire dal vertice $a$ si ottengono due alberi depth first come mostrato in Figura 7.19:

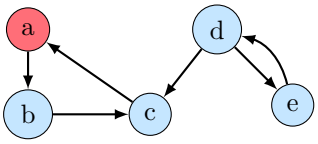



Figura 7.18

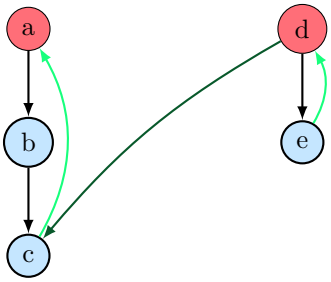



Figura 7.19

Calcolando il grafo trasposto e applicando la DFS sul vertice $a$ si ottiene un solo albero depth first che contiene due componenti fortemente connesse (vedi Figura 7.21).

[6] Della prima visita DFS nel grafo $G$

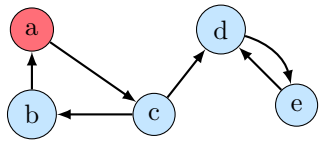



Figura 7.20: Grafo trasposto

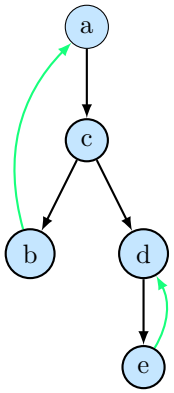



Figura 7.21

La sola visita del grafo trasposto non risolve da solo il problema della mutua raggiungibilità per colpa dell'arco di attraversamento $\langle d, c\rangle$. Infatti, collegando due alberi DF diversi si stanno connettendo vertici di componenti fortemente connesse diverse[7]. Bisogna trovare quindi un modo per inibire tali archi, fare in modo cioè che, quando la DFS (applicata sul grafo trasposto) li trova, non li segua[8].

Osserviamo innanzitutto che un arco di attraversamento, se esiste, segue sempre un verso opposto rispetto ai tempi di scoperta dei vari alberi depth first. Ovvero, dati due alberi depth first con radici in $r_j$ e $r_i$ con $j < i$, un eventuale arco di attraversamento andrà sempre all'indietro come mostrato in figura:

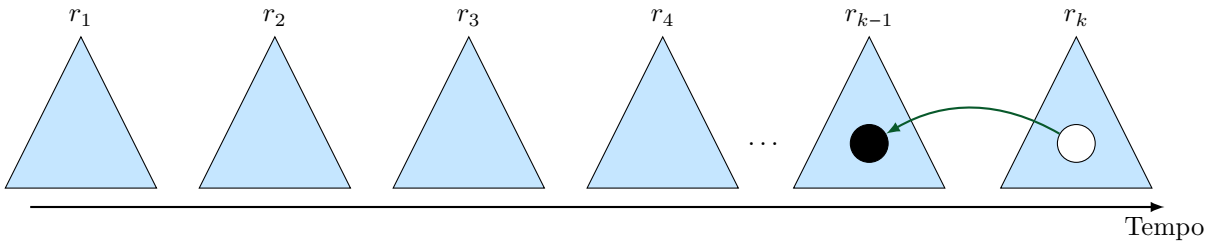



Figura 7.22: Foresta depth first generata da una visita in profondità su un grafo

Ovviamente, se nel grafo $G$ gli archi di attraversamento collegano alberi nuovi ad alberi vecchi, eseguendo una visita in profondità sul grafo trasposto si avrà un'inversione della direzione di tali archi (che andranno quindi da alberi depth first vecchi ad alberi depth first nuovi).

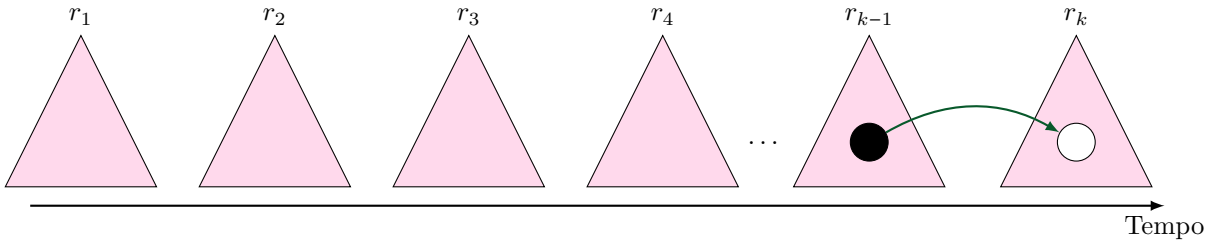



Figura 7.23: Direzione degli archi di attraversamento nel grafo trasposto

Al termine dell'esplorazione di tutti i vertici bianchi raggiungibili da una sorgente $s$, l'Algoritmo DFS_Visit$(G, s)$ (Algoritmo 7.6) lo colora di nero. Questo significa quindi che un arco di attraversamento come mostrato nella Figura 7.22 va sempre da vertici colorati di bianco a vertici colorati di nero[9]. Chiaramente, nel caso di una DFS nel grafo trasposto l'esatto contrario: un arco di attraversamento collega sempre un vertice nero ad un vertice bianco.

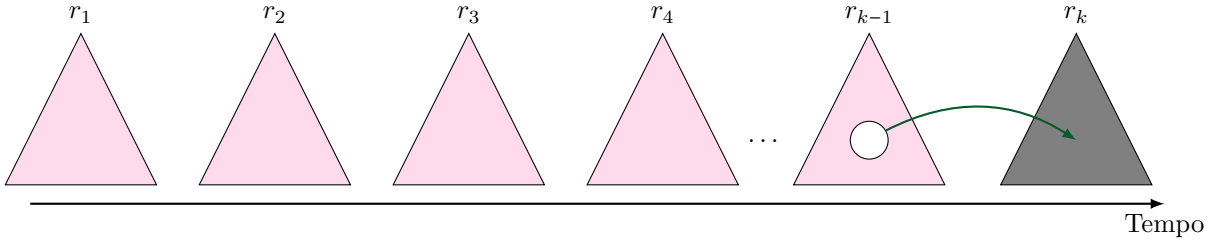



Figura 7.24

[7] Il che è totalmente possibile ma non aiuta nel nostro tentativo di limitare la ricerca dei vertici mutualmente raggiungibili ai soli vertici di un singolo albero depth first.

[8] Non è pensabile eliminarli in quanto questo tipo di operazione altererebbe il grafo. Si ricordi, però, che nella BFS, i percorsi seguiti percorrono soltanto vertici bianchi e grigi. Basterà quindi assicurarsi che tali archi puntino a vertici neri.

[9] Questo in quanto gli alberi $T_{r_k}$ vengono costruiti sempre e solo dopo la terminazione della costruzione dell'albero $T_{r_{k-1}}$.

Prendiamo quindi l'albero $T_{r_k}$ e consideriamo di eseguire una DFS applicata sul grafo trasposto. Chiaramente, essendo tale albero l'ultimo costruito nella prima DFS, non possono esistere altri alberi in cui un eventuale arco di attraversamento possa puntare andando in avanti. Per questo motivo l'applicazione dell'Algoritmo DFS_Visit($G^T, r_k$) può solo raggiungere vertici dell'albero $T_{r_k}$. Terminata tale procedura tutti i vertici (per semplicità) di $T_{r_k}$ saranno colorati di nero. A questo punto DFS_Visit($G^T, r_{k-1}$) vedrà inibiti gli archi di attraversamento che puntano a $T_{r_k}$.

Essendo l'albero $T_{r_k}$ l'ultimo albero della foresta depth first ottenuta nella prima DFS si avrà che il vertice $r_k$ sarà il vertice che avrà il massimo tempo di fine visita, sarà ovvero l'ultimo vertice ad essere eliminato dallo stack. Per inibire, quindi, gli archi di attraversamento sarà sufficiente eseguire l'algoritmo DFS_Visit selezionando i vertici in ordine inverso rispetto ai tempi di fine visita ottenuti nella prima DFS.

Possiamo dunque pensare di modificare l'Algoritmo DFS in modo tale da salvare in uno stack i vertici in ordine di tempo di fine visita. In questo modo, una volta terminata la prima DFS, sarà sufficiente eseguire una seconda DFS sul grafo trasposto in ordine inverso rispetto ai tempi di fine visita ottenuti nella prima DFS. Ogni albero depth first ottenuto in questa seconda visita corrisponderà ad una componente fortemente connessa del grafo $G$.

```
1  S = ∅
2  Init(G)
3  for each v ∈ V do
4    if Color[v] = White then
5      S = DFS_Visit(G,v,S)
6  return S
```

Algoritmo 7.14: DFS(G)

```
1  Color[v] = Gray
2  for each u ∈ Adj[v] do
3    if Color[u] = White then
4      S = DFS_Visit(G,u,S)
5  S = Push(S,v)
6  Color[v] = Black
7  return S
```

Algoritmo 7.15: DFS_Visit(G,v,S)

```
1  Init(G^T)
2  while (S ≠ ∅)
3    v = Top(S)
4    if Color[v] = White then
5      DFS_Visit2(G^T,v)
6    S = Pop(S)
```

Algoritmo 7.16: DFS2($G^T$,S)

```
1  Color[v] = Gray
2  for each u ∈ Adj[v] do
3    if Color[u] = White then
4      Pred[u] = v
5      DFS_Visit2(G^T,u)
```

Algoritmo 7.17: DFS_Visit2($G^T$,v)

```
1  S = DFS(G)
2  G^T = GrafoTrasposto(G)
3  DFS2(G^T ,S)
```

Algoritmo 7.18: CFC(G)

Il calcolo del grafo trasposto si può ottenere calcolando la matrice trasposta della matrice di adiacenza oppure, nel caso delle liste di adiacenza costruendo delle nuove liste a partire da quelle del grafo $G$:

```
1  V^T = ∅
2  for each v ∈ V
3    V^T = V^T ∪ {v}
4  for each v ∈ V do
5    for each u ∈ Adj[v] do
6      Adj^T[u] = InserisciInTesta(Adj^T[u],v)
7  return (V^T, Adj^T)
```

Algoritmo 7.19: GrafoTrasposto(G)

È immediato osservare che l'algoritmo CFC($G$) ha una complessità lineare sulla dimensione del grafo $G$.

## 7.8 Cammini minimi nei grafi

Lo studente Ciro Esposito deve andare a sostenere lo scritto di Algoritmi e Strutture Dati che si terrà in un'aula della sede di Ingegneria a Piazzale Tecchio ma come al solito la Circumvesuviana lo ha fatto fare tardi. Non appena arrivato alla stazione centrale di Napoli si pone il problema di trovare la strada più corta possibile dalla stazione a Fuorigrotta e così apre Google Maps per vedere i vari percorsi disponibili e cercare quello più breve.

Come mostrato in Figura 7.25 l'applicazione associa a ciascun percorso un tempo di percorrenza ottenuto sommando i tempi di percorrenza necessari ad attraversare i vari punti lungo il percorso. Riuscirà il nostro studente ad arrivare in tempo?

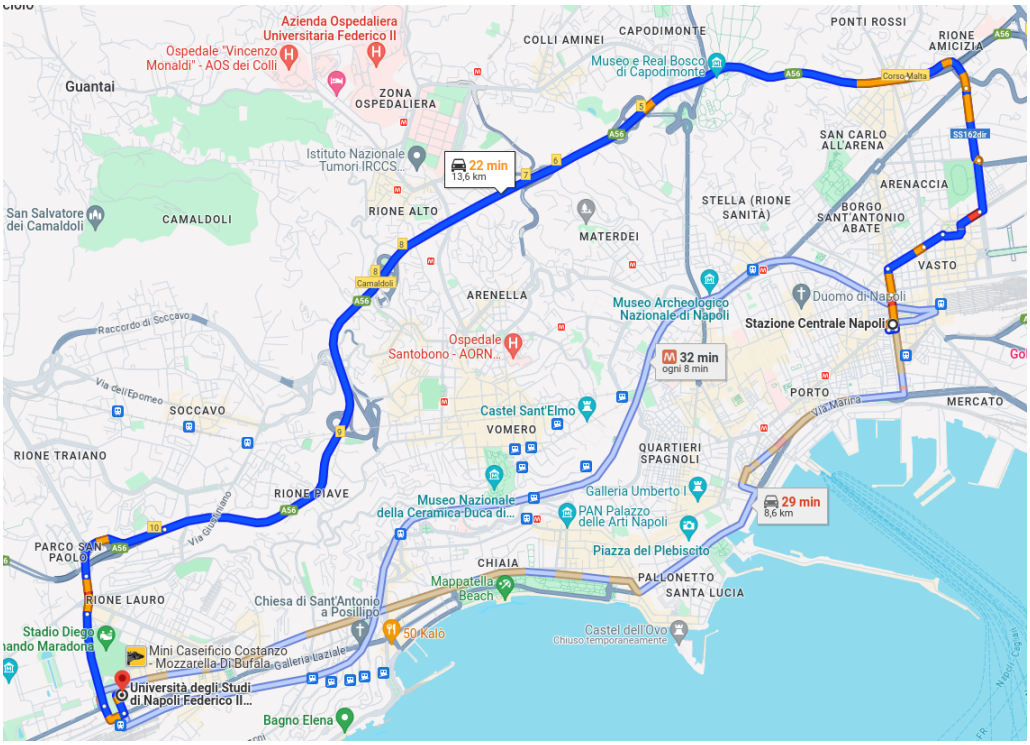



Figura 7.25: I possibili percorsi per raggiungere la sede di Piazzale Tecchio a partire dalla stazione Centrale di Napoli

### 7.8.1 I grafi pesati

Per risolvere questa tipologia di problema che attanaglia migliaia di studenti ritardatari si fa uso del concetto di **grafo pesato**.

**Grafo pesato**

Un **grafo pesato** è una grafo orientato $G = (V, E, w)$ in cui a ciascun arco viene associato un **peso** di valore reale espresso dalla funzione:

$$w : E \longrightarrow \mathbb{R} \tag{7.6}$$

Dato un percorso $\pi = \langle v_0, v_1, \ldots, v_k \rangle$ è possibile considerare il **peso** $w(\pi)$ di tale percorso ottenuto sommando i pesi degli archi che compongono $\pi$:

$$w(\pi) = \sum_{i=1}^{k} w(v_{i-1}, v_i) \tag{7.7}$$

La ricerca del percorso minimo equivale quindi alla ricerca del percorso che minimizzi la quantità $w(\pi)$.

**Cammino minimo**

Il **peso di un cammino minimo** $\delta(u, v)$ da $u$ a $v$ è definito come:

$$\delta(u, v) = \begin{cases} min\{w(\pi) : u \overset{\pi}{\rightsquigarrow} v\} & \text{Se esiste un cammino da } u \text{ a } v \\ \infty & \text{Altrimenti} \end{cases} \tag{7.8}$$

Un **cammino minimo** dal vertice $u$ al vertice $v$ è definito come un cammino qualsiasi $\pi$ con peso $w(\pi) = \delta(u, v)$.

In questa sezione ci occuperemo dell'analisi del **problema dei cammini minimi da sorgente unica**: dato un grafo $G = (V, E, w)$, vogliamo trovare un cammino minimo che va da un dato vertice **sorgente** $s \in V$ a ciascun vertice $v \in V$.

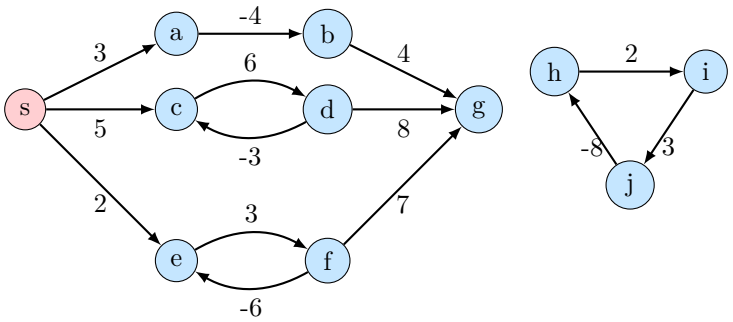
Poiché la funzione $w$ è una funzione a valori reali, in alcuni casi del problema dei cammini minimi da sorgente unica possono presentarsi degli archi in cui i pesi sono negativi. Se il grafo $G = (V, E)$ non contiene cicli di peso negativo che sono raggiungibili dalla sorgente $s$, allora per ogni $v \in V$, il peso del cammino minimo $\delta(s, v)$ resta ben definito, anche se ha un valore negativo. Tuttavia, se esiste un ciclo di peso negativo che è raggiungibile da $s$, i pesi dei cammini minimi non sono ben definiti. Nessun cammino da $s$ ad un vertice del ciclo può essere un cammino minimo in quanto è sempre possibile trovare un cammino di peso minore che segue il cammino "minimo" proposto e poi attraversare il ciclo di peso negativo.

**Osservazione**

Se il grafo ha pesi negativi e ci sono cicli (rispetto ad una sorgente) il percorso minimo **non esiste**.

Esempio

Consideriamo il seguente grafo pesato:




Se volessimo calcolare un percorso minimo dalla sorgente $s$ al vertice $f$ notiamo l'esistenza di un ciclo di peso negativo. Infatti, calcolando il peso dei vari percorsi possibili si osserva che non esiste un possibile percorso minimo:

$$w(\langle s, e, f\rangle) = 2 + 3 = 5$$
$$w(\langle s, e, f, e, f\rangle) = 5 + (-6) + 3 = 2$$
$$w(\langle s, e, f, e, f, e, f\rangle) = 2 + (-6) + 3 = -1$$

Alcuni algoritmi per cammini minimi, come l'Algoritmo di Dijkstra, suppongono che tutti i pesi degli archi del grafo di input siano non negativi. Altri, come l'Algoritmo di Bellman-Ford, accettano archi di peso negativo nel grafo di input a patto di non avere cicli di peso negativo raggiungibili dalla sorgente.

Esempio

Come abbiamo appena visto, un cammino minimo non può avere un ciclo di peso negativo. Non può avere nemmeno un ciclo di peso positivo, perché eliminando il ciclo dal cammino si ottiene un cammino con la stessa sorgente, la stessa direzione e un peso più piccolo.

Se $\pi = \langle v_0, v_1, v_2, \ldots, v_k\rangle$ è un cammino e $\xi = \langle v_i, v_{i+1}, \ldots, v_j\rangle$ è un ciclo di peso positivo in questo cammino (cosicché $v_i = v_j$ e $w(\xi) > 0$), allora il cammino $\pi' = \langle v_0, v_1, \ldots, v_i, v_{j+1}, v_{j+2}, \ldots, v_k\rangle$ ha peso $w(\pi') = w(\pi) - w(c) < w(\pi)$, e quindi $\pi$ non può essere un cammino minimo da $v_0$ a $v_k$.

### 7.8.2 Il rilassamento

Gli algoritmi descritti in questa sezione usano la tecnica del **rilassamento** (vedi Figura 7.26) Per ogni vertice $v \in V$ conserviamo in un array:

$$d : V \longrightarrow \mathbb{R} \tag{7.9}$$

le stime dei vari percorsi minimi a partire da un vertice sorgente $s$ : $\delta(s, v)$. L'attributo $d[v]$ è detto **stima del cammino minimo**. Il processo di rilassamento di un arco $(u, v)$ consiste nel verificare se, passando per un vertice $u$, è possibile migliorare il cammino minimo per arrivare a $v$ precedentemente trovato e, in tal caso, aggiornare i valori $d[v]$ e $pred[v]$. Può essere visto come un rilassamento del vincolo:

$$d[v] \leq d[u] + w(u, v) \tag{7.10}$$

che, per disuguaglianza triangolare deve essere soddisfatto se:

$$d[u] = \delta(s, u)$$
$$d[v] = \delta(s, v)$$

Ovvero, se $d[u] \leq d[u] + w(u, v)$ non occorre esercitare alcuna "pressione" affinché il vincolo 7.10 sia rispettato. L'effetto di un passo di rilassamento può ridurre il valore della stima del cammino minimo $d[v]$ e aggiornare il campo $pred[v]$ del predecessore di $v$. L'esecuzione del passo di rilassamento viene eseguito a tempo costante.

```
1  if d[v] > d[u]+w(u,v)
2    d[v] = d[u] +w(u,v)
3    pred[v] = u
```

Algoritmo 7.20: RELAX(u,v,w)

### 7.8.3 Proprietà dei cammini minimi e del rilassamento

**Lemma 1**

Sia $G = (V, E, w)$ un grafo pesato e $\pi = \langle v_1, v_2, v_3, \ldots, v_{k-1}, v_k \rangle$ un percorso minimo da $v_1$ a $v_k$. Per ogni coppia $(i, j)$ tale che $1 \leq i < j \leq k$ allora il sottopercorso di $\pi$:

$$\pi_{i,j} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$$

è un percorso minimo da $v_i$ a $v_j$, $\pi_{i,j}$ prende il nome di **grafo infisso** del percorso $\pi$.

**Dimostrazione** Se scomponiamo il cammino $\pi$ in:

$$v_0 \overset{\pi_{0i}}{\rightsquigarrow} v_i \overset{\pi_{ij}}{\rightsquigarrow} v_j \overset{\pi_{jk}}{\rightsquigarrow} v_k$$

allora abbiamo:

$$\begin{aligned} w(\pi) &= w(\pi_{0i}) + w(\pi_{ij}) + w(\pi_{jk}) = \\ &= \sum_{z=1}^{i-1} w(v_z, v_{z+1}) + \sum_{z=i}^{j-1} w(v_z, v_{z+1}) + \sum_{z=j}^{k-1} w(v_z, v_{z+1}) \end{aligned}$$

Supponiamo adesso che ci sia un percorso $\pi'_{ij}$ da $v_i$ a $v_j$ con peso $w(\pi'_{ij}) < w(\pi_{ij})$. Allora il cammino $\pi'$:

$$v_0 \overset{\pi_{0i}}{\rightsquigarrow} v_i \overset{\pi'_{ij}}{\rightsquigarrow} v_j \overset{\pi_{jk}}{\rightsquigarrow} v_k$$

è un cammino da $v_0$ a $v_k$ il cui peso:

$$w(\pi') = w(\pi_{0i}) + \underbrace{w(\pi'_{ij})}_{< w(\pi_{ij})} + w(\pi_{jk}) < w(\pi_{0i}) + w(\pi_{ij}) + w(\pi_{jk}) = w(\pi)$$

è minore di $w(\pi)$ che contraddice l'ipotesi secondo la quale $\pi$ sia un cammino minimo da $v_0$ a $v_k$. ■

**Corollario 1**

Dato $G = (V, E, w)$ e $\pi = \langle v_1, v_2, \ldots, v_{k-1}, v_k \rangle$ un percorso minimo da $v_1$ a $v_k$ allora:

$$\delta(v_1, v_k) = \delta(v_1, v_{k-1}) + w(v_{k-1}, v_k) \tag{7.11}$$

**Dimostrazione** Possiamo pensare di scomporre $\pi$ come segue:

$$\pi = v_1 \overset{\pi_{1,k-1}}{\rightsquigarrow} v_{k-1} \rightsquigarrow v_k$$

e per il Lemma 1 il percorso $\pi_{1,k-1}$ risulta minimo. Quindi:

$$w(\pi_{1,k-1}) = \delta(v_1, v_{k-1})$$

da cui segue l'asserto. ■

**Lemma 2 (Disuguaglianza triangolare)**

Dato $G = (V, E, w)$ e $s \in V$, per ogni arco $(u, v) \in E$ vale:

$$\delta(s, v) \leq \delta(s, u) + w(u, v) \tag{7.12}$$

**Lemma 3**

Dato $G = (V, E, w)$ e $(u, v) \in E$, immediatamente dopo l'esecuzione dell'Algoritmo RELAX$(u, v, w)$ varrà che:

$$d[v] \leq d[u] + w(u, v)$$

**Dimostrazione** Immediata per come è scritto l'algoritmo. ■


Figura 7.26: Rilassamento dell'arco $(u, v)$ con peso $w(u, v) = 2$. La stima del cammino minimo è illustrata all'interno del vertice. **(7.26a)** Poiché $v[d] > d[v] + w(u, v)$, il valore $d[v]$ diminuisce.**(7.26b)** Qui, $d[v] \leq d[u] + w(u, v)$ prima del rilassamento quindi RELAX$(u, v, w)$ non modifica la stima.

**Lemma 4**

Dato $G = (V, E, w)$ e posto[a]:

$$\forall v \in V \setminus \{s\}(d[v] = \infty)$$
$$d[s] = 0$$

lungo una qualsiasi sequenza di operazioni di rilassamento vale sempre:

$$\forall v \in V(d[v] \geq \delta(s, v))$$

[a]Nei grafi non pesati l'algoritmo INIT inizializzava le distanze a $\infty$ poiché non si sapeva nulla a priori sulla raggiungibilità dei vertici e $d[s] = 0$. Anche nei grafi pesati è corretto inizializzare $d[s] = 0$ in quanto ci sarebbe altrimenti un ciclo negativo che impedisce l'esistenza dei percorsi minimi.

**Dimostrazione** Si dimostra per induzione sul numero $i$ delle operazioni di rilassamento.

- **Caso base:** Sia $i = 0$. Non è stata eseguita alcuna operazione di rilassamento quindi la proprietà risulta vera per come sono stati inizializzati i vertici.
- **Passo induttivo:** Sia $i > 0$, ciò significa che è stata eseguita almeno una operazione di rilassamento. Supponiamo vero l'asserto prima della $i$-esima operazione di rilassamento. Si consideri l'arco $(x, y) \in E$ ed eseguiamo l'operazione RELAX$(x, y, w)$ che andrà a modificare la stima di $y$. Sicuramente vale:

  $$\forall v \in V \setminus \{y\}\big(d[v] \geq \delta(s, v)\big)$$

  Consideriamo quindi i due casi:

  1. **Caso 1:** se prima dell'operazione di relax vale $d[y] \leq d[x] + w(x, y)$ allora $d[y] \geq \delta(s, y)$
  2. **Caso 2:** se prima dell'operazione di relax vale $d[y] > d[x] + w(x, y)$ allora si pone $d[y] = d[x] + w(x, y)$. Prima di tale operazione ovviamente valeva $d[x] \geq \delta(s, x)$ per ipotesi induttiva. Per il Lemma 2 si ha $\delta(s, y) \leq \delta(s, x) + w(x, y)$. Sommando entrambi i membri per la stessa quantità si ha:

     $$d[x] + w(x, y) \geq \delta(s, x) + w(x, y) \geq \delta(s, y)$$

     Quindi:

     $$d[y] = d[x] + w(x, y) \geq \delta(s, y)$$

■

**Corollario 2**

Se non c'è un cammino da $s$ a $v$ allora si ha sempre $d[v] = \delta(s, v) = \infty$.

**Dimostrazione** Ovvia dalla precedente. Se un vertice $v$ non è raggiungibile da $s$ allora si avrà in ogni momento $d[v] \geq \delta(s, v)$ e non può essere meno di $\infty$. Per questo motivo l'algoritmo di rilassamento è corretto sui vertici non raggiungibili. ■

**Lemma 5**

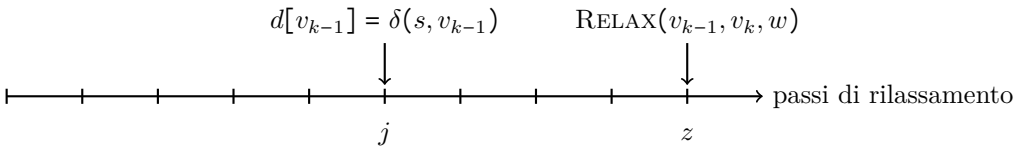
Dato $G = (V, E, w)$ e $s \in V$ e sia $\pi = \langle v_1, v_2, \ldots, v_{k-1}, v_k \rangle$ un percorso minimo da $v_1 = s$ a $v_k$.Inizializzando $d[v_i] = \infty$ per ogni $1 < i \leq k$ e $d[s] = 0$ e presa una sequenza arbitraria di rilassamenti che contiene RELAX$(v_{k-1}, v_k, w)$ (l'ultimo arco del percorso $\pi$), se prima di tale rilassamento valeva:

$$d[v_{k-1}] = \delta(s, v_{k-1})$$

allora dopo tale passo varrà:

$$d[v_k] = \delta(s, v_k)$$

**Dimostrazione** Consideriamo una retta orientata sulla quale vengono disposti i passi di rilassamento applicati sul percorso $\pi$:




Vorremmo assicurarsi che anche prima della $z$-esima operazione valga ancora $d[v_{k-1}] = \delta(s, v_{k-1})$. Chiaramente questo è assicurato dal modo in cui abbiamo implementato l'algoritmo di rilassamento. Infatti, per il Lemma 4 tale quantità può solo aumentare ma se diventa più grande il passo di rilassamento non lo modificherà. Si hanno quindi i seguenti casi:

1. **Caso 1:** si ha
$$d[v_k] \leq d[v_{k-1}] + w(v_{k-1}, v_k) = \delta(s, v_{k-1}) + w(v_{k-1}, v_k)$$
Per il Corollario 1:
$$\begin{aligned} \delta(s, v_k) &= \delta(s, v_{k-1}) + w(v_{k-1}, v_k) \\ &= \delta(s, v_k) && \text{Per il Lemma 4} \end{aligned}$$

2. **Caso 2:** Se $d[v_k] > d[v_{k-1}] + w(v_{k-1}, v_k)$ allora:
$$\begin{aligned} d[v_k] &= d[v_{k-1}] + w(v_{k-1}, v_k) \\ &= \delta(s, v_{k-1}) + w(v_{k-1}, v_k) && \text{Per ipotesi} \\ &= \delta(s, v_k) && \text{Per Corollario 1} \end{aligned}$$

■

### 7.8.4 L'algoritmo di Bellman-Ford

L'algoritmo di Bellman-Ford (Algoritmo 7.22) risolve il problema dei cammini minimi da sorgente unica nel caso generale in cui i pesi degli archi possono essere negativi. Dato un grafo orientato $G = (V, E, w)$ con sorgente $s \in V$, l'algoritmo di Bellman-Ford restituisce un valore booleano che indica se esiste oppure no un ciclo di peso negativo che è raggiungibile dalla sorgente. Se un tale ciclo esiste, l'algoritmo indica che il problema non ha soluzione. Se un tale ciclo non esiste, l'algoritmo fornisce i cammini minimi e i loro pesi.

L'Algoritmo usa il rilassamento, riducendo progressivamente il valore stimato $d[v]$ per il peso di un cammino minimo dalla sorgente $s$ a ciascun vertice $v \in V$, fino a raggiungere il peso effettivo $\delta(s, v)$ di un cammino minimo.

```
1  for each v ∈ V do
2    d[v] = ∞
3    d[s] = 0
```

Algoritmo 7.21: INIT(G,s)

```
1  Init(G,s)
2  for i = 1 to |V|−1 do
3    for each (u,v) ∈ E do
4      Relax(u,v,w)
5  for each (u,v) ∈ E do
6    if d[v] > d[u] + w(u,v) then
7      return false
8  return true
```

Algoritmo 7.22: BELLMAN-FORD(G,w,s)

L'algoritmo di Bellman-Ford viene eseguito nel tempo $O(VE)$ perché l'inizializzazione richiede un tempo $\Theta(V)$, ciascuno dei $|V| - 1$ passaggi sugli arghi nelle righe 2-4 richiede un tempo $\Theta(E)$ e il ciclo `for` nelle righe 5-7 richiede un tempo $O(E)$.

#### Correttezza di Bellman-Ford

**Lemma 6**

Sia $G = (V, E, w)$ un grafo orientato pesato con sorgente $s$. Supponiamo che $G$ non contenga cicli di peso negativo che sono raggiungibili da $s$. Allora, dopo $|V| - 1$ iterazioni del ciclo `for` delle righe 2-4 di BELLMAN-FORD, si ha $v[d] = \delta(s, v)$ per tutti i vertici che sono raggiungibili da $s$.

**Dimostrazione** Sia $v$ un qualsiasi vertice raggiungibile da $s$ e sia $\pi = \langle v_0, v_1, v_2, v_3, \ldots, v_{k-1}, v_k \rangle$, dove $v_0 = s$ e $v_k = v$, un percorso minimo da $s$ a $v$. Tale cammino essendo minimo e aciclico avrà al massimo $|V| - 1$ archi, quindi $k \leq |V| - 1$. Ciascuna delle $|V| - 1$ iterazioni del ciclo `for` delle righe 2-4 rilassa tutti gli $|E|$ archi. Fra gli archi rilassati nella $i$-esima iterazione, per $i = 1, 2, \ldots, k$, c'è l'arco $(v_{i-1}, v_i)$. Per il Lemma 5 si avrà quindi $d[v] = d[v_k] = \delta(s, v_k) = \delta(s, v)$. ■

**Corollario 3**

Sia $G = (V, E, w)$ un grafo orientato pesato con sorgente $s$. Per ogni vertice $v \in V$, esiste un cammino da $s$ a $v$ se e soltato se l'algoritmo BELLMAN-FORD termina con $d[v] < \infty$ quando viene eseguito sul grafo $G$.

**Teorema (Correttezza di Bellman-Ford)**

Supponiamo di eseguire l'algoritmo di BELLMAN-FORD su un grafo orientato pesato con sorgente $s$. Se $G$ non contiene cicli di peso negativo che sono raggiungibili da $s$, allora l'algoritmo restituisce TRUE, si ha $v[d] = \delta(s, v)$ per tutti i vertici $v \in V$. Se $G$ contiene un ciclo di peso negativo che è raggiungibile da $s$ allora l'algoritmo restituisce FALSE.

**Dimostrazione** Supponiamo che il grafo $G$ non abbia cicli di peso negativo raggiungibili dalla sorgente $s$. Per ogni vertice $v$ raggiungibile da $s$ il Lemma 6 assicura che, al termine dell'algoritmo, vale $d[v] = \delta(s, v)$. Se $v$ non è raggiungibile da $s$ allora l'asserzione è vera per il Corollario 2.

Al termine dell'algoritmo, per tutti gli archi $(u, v) \in E$ si ha:

$$\begin{aligned} v[d] &= \delta(s, v) \\ &= \delta(s, u) + w(u, v) && \text{(Per la disuguaglianza triangolare)} \\ &= d[u] + w(u, v) \end{aligned}$$

Quindi nessuno dei controlli fa sì che l'algoritmo di BELLMAN-FORD restituisca FALSE, quindi l'algoritmo restituisce TRUE.

Supponiamo che il grafo $G$ contenga un ciclo di peso negativo che è raggiungibile dalla sorgente $s$, indichiamo questo ciclo con $\xi = \langle v_0, v_1, v_2, \ldots, v_k \rangle$, dove $v_0 = v_k$. Allora:

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 \tag{7.13}$$

Supponiamo per assurdo che l'algoritmo di BELLMAN-FORD restituisca TRUE. Quindi. $d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)$ per ogni $i = 1, 2, \ldots, k$. Sommando le disuguaglianze lungo il ciclo $\xi$ si ottiene:

$$\begin{aligned} \sum_{i=1}^{k} d[v_i] &\leq \sum_{i=1}^{k} (d[v_{i-1}] + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i) \end{aligned}$$

Poiché $v_0 = v_k$, ogni vertice di $\xi$ appare una sola volta in ciascuna delle sommatorie $\sum_{i=1}^{k} d[v_i]$ e $\sum_{i=1}^{k} d[v_{i-1}]$, quindi:

$$\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}]$$

Inoltre, per il Corollario 3, $d[v_i]$ è finito per $i = 1, 2, \ldots, k$; pertanto:

$$0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

che contraddice la disuguaglianza 7.13. Quindi l'algoritmo di BELLMAN-FORD restituisce TRUE se il grafo $G$ non contiene cicli di peso negativo che sono raggiungibili dalla sorgente, altrimenti restituisce FALSE. ■

### 7.8.5 L'algoritmo di Dijkstra

L'algoritmo di Dijkstra risolve il problema dei cammini minimi da sorgente unica in un grafo orientato pesato $G = (V, E, w)$ nel caso in cui tutti i pesi degli archi non siano negativi ($\forall (u, v) \in E w(u, v) \geq 0$ ).

L'algoritmo si basa sull'idea di partizionare l'insieme $V$ in due insiemi, $S$ e $Q$, che variano nel tempo. L'insieme $Q$, che inizialmente equivale a $V$, conserva i vertici non ancora elaborati mentre l'insieme $S$, inizialmente vuoto, conserva man mano i vertici i cui pesi finali dei cammini minimi dalla sorgente $s$ sono stati già determinati. L'algoritmo seleziona ripetutamente il vertice $u \in Q$ con la stima minima del cammino minimo, aggiunge $u \in S$ e rilassa tutti gli archi che escono da $u$.

```
Init(G,s)
S = ∅
Q = V
while Q ≠ ∅
  u = Extract_Min(Q)
  S = S ∪ {u}
  Q = Q \ {u}
  for each v ∈ Adj[u] do
    Relax(u,v,w)
```

Algoritmo 7.23: DIJKSTRA(G,w,s)

Il tempo di esecuzione dell'algoritmo di Dijkstra (Algoritmo 7.23) dipende dall'implementazione dell'insieme $Q$. Infatti, l'operazione di estrazione del minimo non è banale. Infatti, se $Q$ fosse implementato mediante una coda od un array allora potrebbe costare, nel caso peggiore, un tempo lineare sull'insieme dei vertici. In tal caso si avrebbe:

$$T_{Dijkstra} = |V| \cdot (T_{\text{Extract\_Min}}) + |E| \cdot T_{Relax} = |V|^2 + |E| = O(|V|^2)$$

Nella sezione 8.5 del capitolo sugli algoritmi di ordinamento (Capitolo 8) si introdurrà il concetto di **heap binario** che si rivelerà uno strumento importante per l'implementazione delle cosiddette **code di priorità**. Tali strutture dati sono particolarmente interessanti in quanto l'elemento con priorità più alta si trova in testa alla coda mentre quello con priorità più bassa si troverà, appunto, in coda. Volendo implementare l'insieme $Q$ come una coda di priorità è possibile trovare il minimo in un tempo pari ad $O(n \cdot \log n)$ riducendo di conseguenza il tempo dell'Algoritmo di Dijkstra.

### Correttezza dell'algoritmo di Dijkstra

**Teorema**

L'algoritmo di Dijkstra, eseguito su un grafo $G = (V, E, w)$ termina con $d[u] = \delta(s, u)$ per tutti i vertici $u \in V$. Ovvero, ogni volta che un vertice viene messo nell'insieme $S$ allora questo avrà una stima corretta.

**Dimostrazione** Inizialmente, $S = \varnothing$, quindi la proprietà risulta vera. Vogliamo dimostrare che in ogni iterazione vale:

$$d[u] = \delta(s, u)$$

per il vertice aggiunto all'insieme $S$. Supponiamo per assurdo che $u$ sia il primo vertice per il quale $d[u] \neq \delta(s, u)$ quando esso viene aggiunto all'insieme $S$. Chiaramente $u \neq s$ dato che $s$ è il primo vertice ad essere aggiunto all'insieme $S$ e $d[s] = \delta(s, s) = 0$ in quell'instante. Poiché $u \neq s$ è anche vero che $S \neq \varnothing$ appena prima che $u$ venga aggiunto ad $S$.

Deve esistere quindi qualche cammino da $s$ a $u$, perché altrimenti $d[u] = \delta(s, u) = \infty$ per il Corollario 2 e questo violerebbe la nostra ipotesi che $d[u] \neq \delta(s, u)$. Poiché esiste almeno un cammino, allora deve esistere un cammino minimo $p$ da $s$ a $u$. Prima di aggiungere $u$ all'insieme $S$, il cammino $p$ collega un vertice in $S$, cioè $s$, ad un vertice in $Q$, cioè $u$. Consideriamo il primo vertice $y$ lungo $p$ tale che $y \in Q$ e sia $x \in S$ il predecessore di $y$. Allora, come illustra la Figura 7.27, il cammino $p$ può essere scomposto in $s \overset{p_1}{\rightsquigarrow} x \overset{p_{p2}}{\rightsquigarrow} u$.

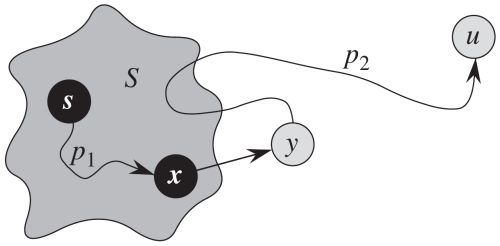



Figura 7.27

Asseriamo che $d[y] = \delta(s, y)$ quando $u$ viene aggiunto a $S$. Per dimostrare ciò, osserviamo che $x \in S$. Allora, poiché $u$ è stato scelto come il primo vertice per il quale $d[u] \neq \delta(s, u)$ quando esso viene aggiunto a $S$, era vero che $d[x] = \delta(s, x)$ quando $x$ è stato aggiunto a $S$. L'arco $(x, y)$ è stato rilassato in quell'istante, quindi l'asserzione è dimostrata per il Lemma 5.

Adesso possiamo ottenere una contraddizione per dimostrare che $d[u] = \delta(s, u)$. Poiché $y$ precede $u$ in un cammino minimo da $s$ a $u$ e tutti i pesi degli archi non sono negativi, si ha $\delta(s, y) \leq \delta(s, u)$ e quindi:

$$\begin{aligned} d[y] &= \delta(s, y) \\ &= \delta(s, u) \\ &= d[u] \end{aligned}$$

Tuttavia, poiché entrambi i vertici $u$ e $y$ si trovavano in $Q$ quando $u$ venne scelto, si ha $d[u] \leq d[y]$. Quindi le due disuguaglianze sono delle uguaglianze:

$$d[y] = \delta(s, y) = \delta(s, u) = d[u]$$

Di conseguenze $d[u] = \delta(s, u)$, che contraddice la nostra scelta di $u$. Concludiamo quindi che $d[u] = \delta(s, u)$ quando $u$ viene aggiunto a $S$ e che questa uguaglianza permane in tutti gli istanti successivi. ■

# Algoritmi di ordinamento

## 8.1 Il problema dell'ordinamento

**Ordinamento**

Sia $A$ una sequenza di n numeri $\langle a_1, a_2, ..., a_n \rangle$ con $a_i \in \mathbb{Z} \quad \forall\ 1 \leq i \leq n$. **Ordinare** $A$ significa trovare una sequenza $A'$ chiamata **permutazione ordinata** di $A$, ovvero un riarrangiamento $\langle a'_1, a'_2, ..., a'_n \rangle$ tale che

$$a'_1 \leq a'_2 \leq \cdots \leq a'_n \tag{8.1}$$

Per ordinare una sequenza si può pensare di generare tutte le permutazioni possibili a partire da tale sequenza e verificare la proprietà di ordinamento 8.1. Da questa idea possiamo dire con certezza che almeno un algoritmo che risolva il problema dell'ordinamento esista dato che, essendo l'ordinamento totale esisterà sicuramente una soluzione. Ovviamente questo approccio è tanto banale quanto complesso. Algoritmi di questo tipo vengono chiamati **algoritmi di forza bruta (brute force)**.

Nel caso del problema della massima sottosequenza contigua contigue visto in 1.3, l'algoritmo di forza bruta aveva un costo quadratico. In questo caso invece, la differenza è abissale. Data una sequenza di $n$ elementi il numero di permutazioni sarà $n!$, controllare la condizione 8.1 implica scorrere ogni elemento di ciascuna permutazione raggiungendo così il costo di $\Omega(n \cdot n!)$ che è molto ma molto grande rispetto ad $n^2$.

Nel corso di questo capitolo si vedrà che esistono algoritmi più "sensati" per la ricerca di una sequenza ordinata. Il problema da porsi sarà quello della loro *costruzione*. Tutti gli algoritmi che si vedranno in questo capitolo avranno un tempo limitato superiormente da $n^2$, quindi sarà: $T(n) = O(n^2)$. L'idea alla base sarà quella di ordinare mediante l'utilizzo del *confronto*, volta per volta, di coppie di elementi. Chiaramente, nel migliore dei casi il costo sarà lineare (meno di così non sarà possibile scendere). Questa analisi ci permette quindi di limitare l'intervallo all'interno del quale selezionare algoritmi che siano "sensati".

## 8.2 Insertion Sort

L'algoritmo **Insertion sort** è un buon metodo per ordinare un piccolo numero di elementi. La sua idea basilare è molto banale ed è simile al modo in cui molte persone ordinano le carte: preso un mazzo di carte, si inseriscono le carte nella loro posizione corretta una dopo l'altra dopo averle confrontate con quelle precedentemente ordinate. In ogni momento quindi, l'algoritmo vede la sequenza divisa in due parti: una ordinata e una disordinata. Consideriamo quindi una sequenza $A$, l'algoritmo InsertionSort (Algoritmo 8.1) ordina i numeri di input **in place**, nel senso che i numeri sono risistemati all'interno dell'array $A$ senza l'utilizzo di alcuna struttura ausiliaria. Quando la procedura è completata, l'array di input $A$ contiene la sequenza di output ordinata.

```
1  for j=2 to n do
2    elem = A[j]
3    // Inserisce A[j] nella sequenza ordinata A[1, ... ,j-1]
4    i = j-1
5    while i ≥ 1 && elem < A[i] do
6      A[i+1] = A[i]
7      i = i-1
8    A[i+1] = elem
```

Algoritmo 8.1: InsertionSort(A,n)

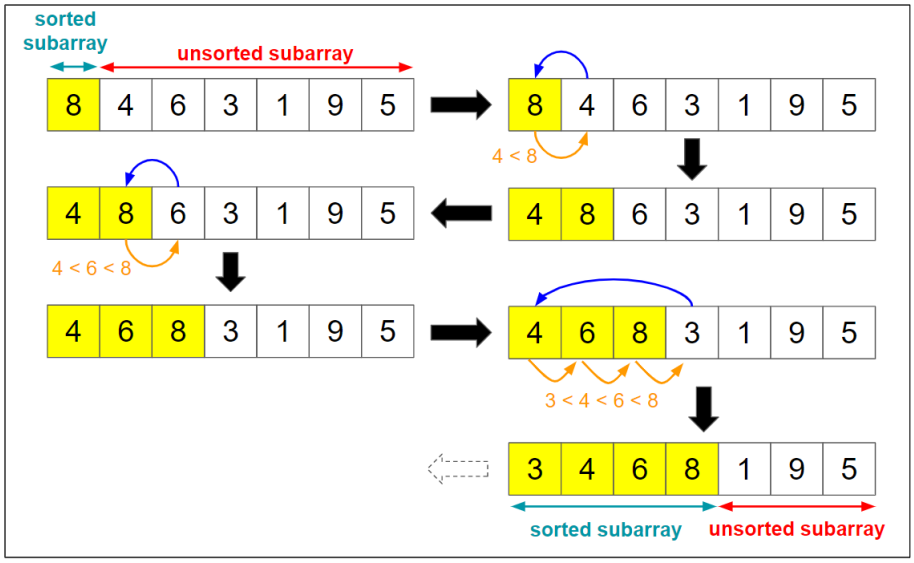



Figura 8.1: Esempio di applicazione di INSERTION SORT

### 8.2.1 Invarianti di ciclo e correttezza di InsertionSort

Il tempo richiesto dalla procedura INSERTION-SORT dipende dall'input: occorre più tempo per ordinare un migliaio di numeri che tre numeri. Inoltre, INSERTION-SORT può richiedere quantità di tempo differenti per ordinare due sequenze di input della stessa dimensione a seconda della *qualità* della sequenza. Infatti non si può dire a priori il numero di esecuzioni del ciclo `while` al rigo 5 dell'Algoritmo 8.1.

In casi come questi si andranno a fare due analisi distinte e separate, una prima per quelle classi di istanze "buone" che offriranno tempi di esecuzioni brevi e una seconda analisi per quelle istanze dette anche "peggiori". Bisogna capire però come *formalizzare la dipendenza dalla bontà dell'input del tempo*. Per rispondere a questa domanda si introducono una serie di parametri, detti $t_j$, che indicano il numero di iterazioni della testa del ciclo `while`, *fissato* l'indice $j$.

Poiché abbiamo che $j$ varia da 2 a $n$ avremo rispettivamente: $t_2, ..., t_n$. Complessivamente la testa del `while` sarà eseguita quindi $\sum_{j=2}^{n} t_j$ volte. Una volta introdotti questi parametri possiamo dire quindi che le istruzioni 6 e 7 saranno eseguite rispettivamente $\sum_{j=2}^{n}(t_j - 1)$ volte. Si ha così un **prospetto parametrico** del costo istruzione per istruzione.

| Istruzione | # Operazioni | Esecuzioni |
|---|---|---|
| 1 | 2 | $n$ |
| 2 | 2 | $n-1$ |
| 3 | 2 | $n-1$ |
| 5 | 4 | $\sum_{j=2}^{n} t_j$ |
| 6 | 3 | $\sum_{j=2}^{n} t_{j-1}$ |
| 7 | 1 | $\sum_{j=2}^{n} t_{j-1}$ |
| 8 | 2 | $n-1$ |

Il tempo di esecuzione dell'algoritmo è la somma dei tempi di esecuzione per ogni istruzione eseguita.

Per calcolare $T(n)$ sommiamo i prodotti delle colonne **Numero operazioni** e **Numero di volte**:

$$\begin{aligned} T(n) &= 2n + 2(n-1) + 2(n-1) + 4\sum_{j=2}^{n} t_j + 3\sum_{j=2}^{n}(t_j - 1) + \sum_{j=2}^{n}(t_j - 1) \\ &= 9n - 7 + 4\sum_{j=2}^{n} t_j + 4\sum_{j=2}^{n}(t_j - 1) \end{aligned} \tag{8.2}$$

L'equazione 8.2 non è una espressione precisa poiché dipendente dai parametri $t_j$ però a partire da questa formula possiamo iniziare a fare le due analisi distinte, una per il **caso peggiore** e una per il **caso migliore**.

#### Caso peggiore

Nel caso peggiore, ovvero quello di una sequenza ordinata in senso inverso, nella testa del `while` la condizione $elem < A[i]$ è sempre vera, quindi la condizione farà sempre entrare nel `while` e il ciclo dipenderà unicamente dall'indice $i \geq 1$. L'algoritmo dovrà confrontare ogni elemento $A[j]$ con ogni elemento dell'intera sottosequenza ordinata $A[1..j-1]$ e quindi sarà $t_j = j$ per $j = 2, 3, ..., n$.

Sarà quindi:

$$\begin{aligned} T(n) &= 9n - 7 + 4\sum_{j=2}^{n} j + 4\sum_{j=2}^{n}(j-1) \\ &= 9n - 7 + 4 \cdot \left(\frac{n(n+1)}{2} - 1\right) + 4 \cdot \frac{n(n-1)}{2} \\ &= 9n - 7 + 4 \cdot ((n^2 + n) - 2) + 4 \cdot \frac{n(n-1)}{2} \\ &= 9n - 7 + 4n^2 + 4n - 8 + 2n^2 - 2n \\ &= 6n^2 + 11n - 15 \end{aligned} \tag{8.3}$$

Possiamo concludere quindi che, nel caso peggiore $T(n) = O(n^2)$.

### Caso migliore

In Insertion-Sort il caso migliore si verifica se l'array è già ordinato. Per ogni $j = 2, 3, ..., n$, troviamo che $A[i] < elem$ nella testa del ciclo `while`, quando $i$ ha il suo valore iniziale $j - 1$. Quindi $t_j = 1$ per $j = 2, ..., n$ e il tempo di esecuzione nel caso migliore è:

$$\begin{aligned} T(n) &= 9n - 7 + 4\sum_{j=2}^{n} t_j + 4\sum_{j=2}^{n}(t_j - 1) \\ &= 9n - 7 + 4\sum_{j=2}^{n} 1 + 4\sum_{j=2}^{n}(0) \\ &= 9n - 7 + n - 1 \\ &= 10n - 8 \end{aligned} \tag{8.4}$$

Nel caso migliore quindi $T(n) = O(n)$.

### Caso medio

L'equazione 8.2 ci aveva dato una formula generale, dipendente dai parametri $tj$, del costo di Insertion-Sort. Ci si può chiedere però se, fissato l'indice $j$, possa esistere un parametro $t_{medio}$. Nulla vieta di calcolare $t_{medio}$ come la media aritmetica dei vari parametri, assumendo una equiprobabilità delle tipologie di input:

$$t_{medio} = \frac{\sum_{k=1}^{j} k}{j} = \frac{\frac{j(j+1)}{2}}{j} = \left\lfloor \frac{j+1}{2} \right\rfloor$$

E così la 8.2 diventa:

$$\begin{aligned} T_{medio}(n) &= 9n - 7 + 4\sum_{j=2}^{n} \frac{j+1}{2} + 4\sum_{j=2}^{n}\left(\frac{j+1}{2} - 1\right) \\ &= 9n - 7 + 2\sum_{j=2}^{n}(j+1) + 2\sum_{j=2}^{n}(j-1) \\ &= 9n - 7 + 2\sum_{j=2}^{n}(j+1) + 2\sum_{j=1}^{n-1} j \\ &= 9n - 7 + 2\sum_{j=2}^{n}(j+1) + \frac{n(n-1)}{2} \end{aligned} \tag{8.5}$$

Possiamo dire quindi che anche nel caso medio Insertion-Sort ha un comportamento quadratico: $T_{medio}(n) = O(n^2)$.

## 8.3 Merge Sort

Per Insertion Sort abbiamo usato un approccio **incrementale**: dopo avere ordinato il sottoarray $A[1..j-1]$, abbiamo inserito un singolo elemento $A[j]$ nella posizione appropriata, ottenendo il sottoarray ordinato $A[1, ..., j]$. L'algoritmo Merge-Sort è un tipo di algoritmo come quelli visti nel Capitolo 6, ovvero un algoritmo ricorsivo. La ricorsione, in questo caso, ridurrà di molto il tempo di esecuzione rispetto ad Insertion-Sort nel caso peggiore.

L'algoritmo MergeSort è conforme al paradigma divide et impera. Esso opera nel modo seguente:

- **Divide:** divide la sequenza degli $n$ elementi da ordinare in due sottosequenze di $n/2$ elementi ciascuna.
- **Impera:** ordina le due sottosequenze in modo ricorsivo utilizzando l'algoritmo merge sort.

- **Combina:** fonde le due sottosequenze ordinate per generare la sequenza ordinata.

La ricorsione tocca il fondo quando la sequenza da ordinare ha lunghezza 1, in quel caso non c'è nulla da fare, in quanto ogni sequenza di lunghezza 1 è già ordinata.

```
1  // Verifica dell'intervallo
2  if p<r then
3    q = ⌊ p+r / 2 ⌋
4    MergeSort(A,p,q)
5    MergeSort(A,q+1,r)
6    Merge(A,p,q,r)
```

Algoritmo 8.2: MergeSort(A,p,r)

L'operazione chiave dell'algoritmo Merge Sort è la *fusione* di due sottosequenze ordinate nel passo "combina". Per effettuare la fusione, utilizziamo una procedura ausiliaria chiamata Merge$(A, p, q, r)$, dove $A$ è un array e $p$, $q$, $r$ sono indici dell'array tali che $p \leq q \leq r$. La procedura assume che i sottoarray $A[p..q]$ e $A[q+1...r]$ siano ordinati; li **fonde** per formare un unico array ordinato che sostituisce il sottoarray corrente $A[p..r]$.

### 8.3.1 Correttezza dell'algoritmo

Poiché $p$ ed $r$ indicano l'indice iniziale e finale della sequenza $A$ possiamo dire che $A$ ha $r-p+1$ elementi. Affinché Merge-Sort funzioni bisogna garantire la convergenza del passo "divide", ovvero:

$$r-p+1 > \underbrace{q-p+1}_{1^a\,sottosequenza} \quad \wedge \quad \underbrace{r-p+1}_{2^a\,sottosequenza} \quad > \quad \underbrace{r-q}_{r-(q+1)+1}$$

dove

$$q = \lfloor \frac{p+r}{2} \rfloor$$

e sappiamo che

$$\lfloor \frac{p+r}{2} \rfloor \leq \frac{p+r}{2}$$

ma allora risulta:

$$r-p+1 > \frac{p+r}{2} - p + 1 \implies 2r > p+r \implies r > p$$

ed essendo la nostra ipotesi proprio $p < r$ abbiamo dimostrato la prima implicazione. Analogamente si procede per la seconda:

$$r-p+1 > r - \frac{p+r}{2} \implies -p > -\left(\frac{p+r}{2}+1\right) \implies 2p < p+r+2 \implies p < r+2$$

Resta solo da dimostrare che, per qualsiasi input, l'algoritmo faccia un numero finito di chiamate ricorsive e che quindi, prima o poi, raggiungerà il caso base. Quando r è molto vicino a p, ovvero quando $r < p+1$ è evidente che sarà $q = \lfloor (p+r)/2 \rfloor = p$ e quindi entrambe le chiamate ricorsive non verranno effettuate poiché la condizione del blocco condizionale if sarà alla prima chiamata $p < p$ ed alla seconda $p+1 < r$ (entrambe ovviamente false).

### 8.3.2 L'algoritmo Merge

```
1  i=p
2  k=p
3  j=q+1
4  while i ⩽ q && j ⩽ r do
5    if A[i] ⩽ A[j] then
6      B[k] = A[i]
7      i=i+1
8    else
9      B[k] = A[j]
10     j=j+1
11   k =k+1
12 if i < q then
13   j=i
14 while k ⩽ r do
15   B[k] = A[j]
16   j = j+1
17   k = k+1
```

Algoritmo 8.3: Merge(A,p,q,r)

Il costo di esecuzione di Merge è ovviamente lineare poiché si andranno a fare almeno $n$ scritture in memoria:

$$T_{Merge} = \Theta(n)$$

### 8.3.3 Analisi del costo di Merge Sort

Per calcolare il tempo di esecuzione dell'algortimo MERGE SORT (Algoritmo 8.2) si definisce una funzione di ricorrenza del tipo:

$$T_{MergeSort}(N) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ T_{Merge}(N) + \Theta(1) + 2T_{MergeSort}(N/2) & \text{se } n > 1 \end{cases}$$

che, sommando $T_{Merge}(N) + \Theta(1)$, diventa:

$$T_{MergeSort}(N) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ 2T_{MergeSort}(N/2) + \Theta(N) & \text{se } n > 1 \end{cases} \tag{8.6}$$

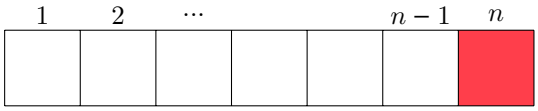
Notiamo che l'equazione 8.6 è uguale all'equazione 6.1 vista nel Capitolo 6. Possiamo concludere quindi che il costo di esecuzione di MERGE SORT è:

$$T_{MergeSort} = \Theta(n \log_2 N)$$

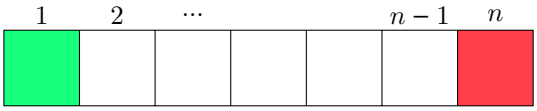
## 8.4 SELECTION SORT

L'idea che sta alla base dell'algoritmo SELECTION SORT può essere visto come il duale di quella dell'algoritmo INSERTION SORT. Infatti, selezionata una posizione della sequenza si trova successivamente l'elemento da metterci dentro.

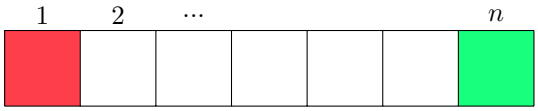
Come si vede in Figura 8.2, se iniziamo dall'ultima posizione, ovvero $N$, allora si dovrà trovare il massimo elemento della sequenza evidenziata in azzurro e poi scambiarlo con quello in posizione $N$ (se avessimo preso l'elemento in prima posizione allora si sarebbe dovuto cercare l'elemento più piccolo). Il procedimento si ripete riducendo di volta in volta la sequenza da ordinare che avrà dimensione $N$ poi $N-1$ fino a quando, arrivati alla posizione 2, non si sarà ottenuta la sequenza ordinata (infatti alla fine dell'ultima iterazione il minimo già si troverà nella posizione 1).



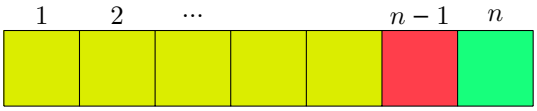

(a) All'inizio l'array non è ordinato, selezioniamo quindi l'ultima cella dove dovrebbe trovarsi il massimo.




(b) Chiamo l'algoritmo FINDMAX sulla sequenza $1, n-1$: se il massimo si trova nella cella 0, evidenziata in verde, eseguo lo swap con la cella rossa.




(c) A questo punto il massimo si trova in posizione corretta.




(d) Riduciamo la sottosequenza e ripetiamo il ragionamento. La parte evidenziata in verde risulta già ordinata mentre resta da ordinare la parte evidenziata in giallo.

Figura 8.2: Esempio di applicazione dell'Algoritmo 8.4

```
1 for i =n to 2 do
2   j = FindMax(A,i)
3   Swap(A,i,j)
```

Algoritmo 8.4: SELECTIONSORT(A,n)

```
1 max = i
2 for i = 2 to n
3 if A[max] < A[i] then
4   max = i
5 return max
```

Algoritmo 8.5: FINDMAX(A,n)

### 8.4.1 Analisi del costo di SelectionSort

Poiché sappiamo che la scelta del massimo non può essere fatta con un algoritmo meno che lineare (poiché bisogna confrontare comunque ogni elemento della successione) sembrerebbe che questa sia una soluzione ottimale. Questo algoritmo però è un esempio di come usare usare soluzioni ottime per dei sottoproblemi non rende l'algoritmo ottimale. Ma ciò non significa che l'idea alla base sia pessima; infatti nella sezione 8.5 vedremo come abbassare il tempo di esecuzione ad un $\Theta(n \log_2 n)$.

Si ha quindi:

$$\begin{aligned} T_{SS}(N) &= \Theta(N) + \sum_{i=2}^{N} T_{FindMax}(i) + \Theta(1) \\ &= \Theta(N) + \Theta\Big(\sum_{i=2}^{N} i\Big) \\ &= \Theta(N) + \Theta\Big(\Big(\sum_{i=1}^{N} i\Big) - 1\Big) \\ &= \Theta(N) + \Theta\Big(\frac{n(n+1)}{2} - 1\Big) \\ &= \Theta(n^2) \end{aligned} \tag{8.7}$$

## 8.5 HEAP SORT

L'algoritmo SELECTIONSORT (Algoritmo 8.4) si è dimostrato essere il peggior algoritmo visto fino a questo momento. Il suo problema non è legato all'idea che sta alla sua base quando piuttosto alla sua *implementazione*. Il punto è che l'algoritmo FINDMAX (Algoritmo 8.5) **non conserva informazioni sull'ordinamento parziale**. L'algoritmo HEAPSORT cerca di risolvere questa problematica introducendo una struttura dati detta **heap** (mucchio), per gestire la conservazione di queste informazioni e *ridurre il numero di confronti per la ricerca del massimo.*

### 8.5.1 Gli alberi heap

Dalle proprietà degli alberi binari pieni[1] si ha che:

$$N_{\text{nodi interni}} = \sum_{i=0}^{h-1} 2^i = 2^h - 1 \tag{8.8}$$

$$h = \lfloor \log_2 n \rfloor \tag{8.9}$$

$$N_{\text{nodi}} = 2^{h+1} - 1 \tag{8.10}$$

da cui si evince che non tutti gli insiemi possono essere rappresentati da un albero binario pieno (ad esempio un insieme con 5 elementi).

Per garantire che insiemi di ogni cardinalità possano essere rappresentati da un albero heap si usano gli **alberi binari completi**, ovvero un albero sul quale si impone un rilassamento dei vincoli degli alberi binari pieni. In particolare:

- Tutte le foglie sono a livello $h$ oppure $h-1$;
- Tutti i nodi interni hanno grado 2 tranne al più un nodo.

**Albero heap**

Un **albero heap** è un albero binario di altezza $h$ tale che, per ogni nodo $i$:
1. tutte le foglie hanno profondità $h$ o $h-1$;
2. tutti i nodi interni hanno grado 2, eccetto al più uno;
3. entrambi i nodi $j$ e $k$ figli di $i$ sono *non maggiori* di $i$.

**Osservazione**

Le prime due condizioni definiscono la forma dell'albero, in particolare un albero completo mentre la terza condizione definisce l'**architettura** dell'albero. In definitiva, un albero heap è un albero binario completo tale che per ogni nodo entrambi i figli non sono maggiori del padre (proprietà di ordinamento parziale). Nell'albero in figura 8.3 quindi, con l'arco che preserva le informazioni relative all'ordinamento:$x \geq y$, $x \geq z$ $y \geq k$, $x \geq k$ (per transitività).

[1] Vedi 3.1.1

Esempio

Sono esempi di alberi heap i seguenti alberi:

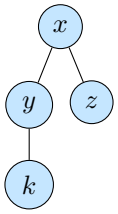



Figura 8.3: Esempio di albero heap

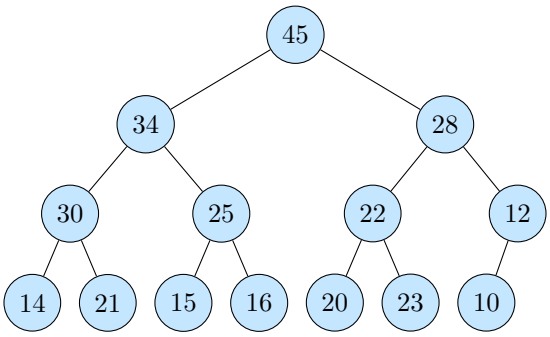



Figura 8.4

### 8.5.2 Il problema della rappresentabilità

Dato un qualsiasi $n \in \mathbb{N}$, siamo sicuri che tutti gli insiemi possono essere rappresentati da un albero binario completo? In generale infatti possiamo saltare da alberi pieni contenenti $2^{h+1} - 1$ nodi ad alberi con $2^{(h+1)+1} - 1 = 2^{h+2} - 1$ nodi. Non esiste però nessun albero pieno con un numero di nodi compreso tra $2^{h+1} - 1$ e $2^{h+2} - 1$. Se si riuscisse a dimostrare che si possono costruire alberi binari completi a partire da $2^{h+1} - 1$ nodi allora si potrebbe garantire la proprietà ricercata.

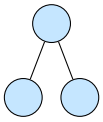
Sia $h = 1$ Allora è possibile considerare due alberi pieni contenenti:

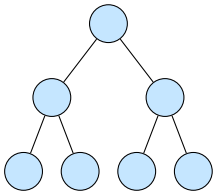
$$2^{h+1} - 1 = 2^{1+1} - 1 = 2^2 - 1 = 4 - 1 = 3$$

e

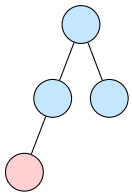
$$2^{h+2} - 1 = 2^{1+2} - 1 = 2^3 - 1 = 8 - 1 = 7$$

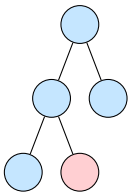
nodi ciascuno come mostrato in figura:

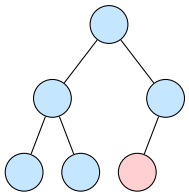




Ad esempio, aggiungendo un nodo all'albero pieno con tre nodi si ottiene un albero completo avente 4 nodi compreso tra i due alberi pieni.







Procedendo induttivamente con questa estensione possiamo dire quindi che il problema della cardinalità è stato risolto: dall'arbitrarietà di $h$ possiamo costruire alberi completi per ogni numero di nodi $x$ tali che:

$$2^{h+1} - 1 \leq x \leq 2^{h+2} - 1$$

Il rilassamento che abbiamo fatto garantisce la proprietà 8.9? Bisogna dimostrare cioè che anche per un albero completo vale la proprietà di essere *il più corto con quel numero di nodi* e che quindi è possibile ancora scrivere l'altezza in corrispondenza del numero di nodi. Chiaramente, negli alberi completi non vale più la proprietà 8.10 poiché abbiamo dimostrato che un albero completo **può avere un numero di nodi qualsiasi**. Ma è vero anche che $2^h \leq n \leq 2^{h+1}$ e quindi $n$ o è una potenza di due o si trova tra due potenze di due.

Dunque, sapendo che $\forall n \in \mathbb{N}, \exists h \geq 0$

$$2^h - 1 \leq n \leq 2^{h+1} - 1 \Rightarrow 2^{h+1} - 1 \leq n \leq 2^{h+2} - 1$$

applicando il logaritmo a tutti i membri:

$$\log(2^{h+1} - 1) \leq \log n \leq \log(2^{h+1} - 1) \Rightarrow \lfloor \log(2^{h+1} - 1) \rfloor \leq \lfloor \log n \rfloor \leq \lfloor \log(2^{h+2} - 1) \rfloor$$

Quindi $h \leq \lfloor \log n \rfloor \leq h+1$, ed essendo $\lfloor \log n \rfloor$ un intero si ha $\lfloor \log n \rfloor = h$. Dunque, anche gli alberi completi ottimizzano l'altezza per il numero di nodi.

### 8.5.3 Implementazione degli alberi heap

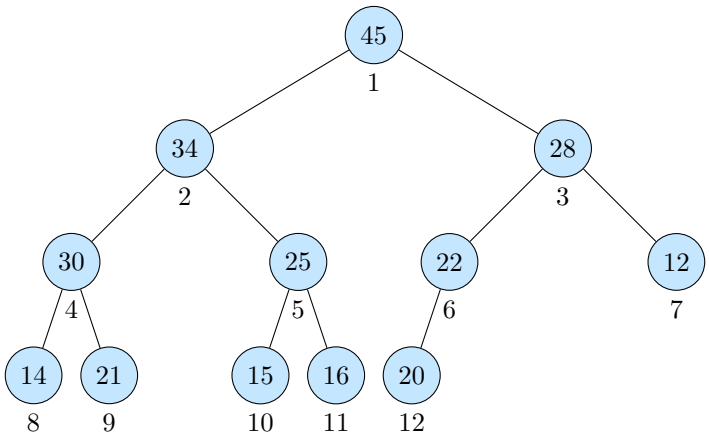
L'albero heap è una **struttura dati astratta**, bisogna quindi porsi il problema di come implementarla attraverso una **struttura dati concreta**. In particolare, uno heap può essere implementato:

- come un albero a puntatori;
- come un array.

#### Da albero ad array

Come osservato in precedenza, **qualsiasi sequenza numerica può essere rappresentata in un albero completo** in un modo del tutto naturale ponendo come unica condizione il fatto che i nodi dell'albero siano **contigui**.

Si consideri ad esempio l'albero heap mostrato in figura 8.4. È possibile rappresentare tale albero mediante un array eseguendo una scansione livello per livello:




Quindi gli indici dell'array andranno da sinistra verso destra livello per livello:

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 45 | 34 | 28 | 30 | 25 | 22 | 12 | 14 | 21 | 15 | 16 | 20 |

Dunque **un heap può essere implementato come un array** $A$ in cui:

- la radice dello heap sta nella posizione $A[0]$ dell'array.
- Se il nodo $i$ dello Heap sta nella posizione $i$ dell'array (cioè $A[i]$) allora:
  - il figlio sinistro di $i$ sta nella posizione $2i$;
  - il figlio destro di $i$ sta nella posizione $2i + 1$;

Dunque **un array sarà un albero heap** se soddisfa le condizioni:

$$A[i] \geq A[2i] \tag{8.11}$$
$$A[i] \geq A[2i + 1] \tag{8.12}$$

Da questa corrispondenza si ottiene anche che le posizioni nell'array che contengono le foglie sono quelle per cui $2i > n$, da cui $i > \frac{n}{2}$, ovvero nella **seconda metà di destra dell'array**. Più precisamente i nodi interni andranno da $1 \leq i \leq \lfloor n/2 \rfloor$, mentre le foglie da $\lfloor n/2 \rfloor \leq i \leq n$.

Un array $A$ che rappresenta un albero heap è caratterizzato dall'attributo `heapsize` il quale denota il numero di nodi dell'albero heap correntemente memorizzati nel vettore. Chiaramente si ha sempre:

$$0 \leq \texttt{heapsize} \leq \texttt{A.lenght}$$

Anche se ci possono essere dei numeri memorizzati in tutto l'array $A$, soltanto i numeri $A[1, \ldots, heapsize]$ sono elementi validi dell'heap. La radice dell'albero viene sempre memorizzata in $A[1]$. Se $i$ è l'indice di un nodo, gli indici del padre PARENT($i$), del figlio sinistro LEFT($i$) e del figlio destro RIGHT($i$) possono essere calcolati mediante le seguenti procedure:

```
1 return ⌊n/2⌋
```

Algoritmo 8.6: PARENT(i)

```
1 return 2i
```

Algoritmo 8.7: LEFT(i)

```
1 return 2i + 1
```

Algoritmo 8.8: RIGHT(i)

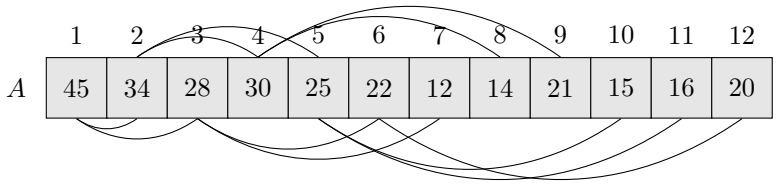



Figura 8.5

### 8.5.4 L'algoritmo Heapify: conservare la proprietà dell'heap

Come osservato fino a questo punto, un vettore rappresentante un albero heap gode di un ordinamento parziale dato dalle relazioni 8.11 e 8.12. Basta osservare l'array $A$ mostrato in Figura 8.5 per convincersi del fatto che tale vettore non sia totalmente ordinato.

Volendo ripetere il ragionamento fatto per l'algoritmo SELECTIONSORT (Algoritmo 8.4) si potrebbe pensare di utilizzare un algoritmo FINDMAX che, dato un array $A$ e un indice $i$, restituisca l'indice $j$ del massimo elemento dell'array $A[i, \ldots, n]$. Tale algoritmo però non è sufficiente per garantire che l'array $A$ sia di nuovo un heap. Infatti, se si considera l'array $A$ mostrato in Figura 8.5 e si applica l'algoritmo FINDMAX$(A, 1)$ si ottiene che il massimo elemento è 45 e che si trova nella posizione 1. Se si scambia l'elemento 45 con l'elemento 20 si ottiene l'array $A'$ mostrato in Figura 8.6 che non è un heap.

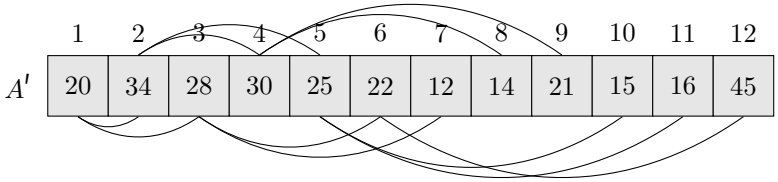



Figura 8.6

Per questo motivo, per mantenere la proprietà di heap è necessario usare l'algoritmo HEAPIFY (Algoritmo 8.9) che, dato un array $A$ e un indice $i$, **ripristina la proprietà di heap** nell'albero binario con radice in $i$ assumendo che i sottoalberi con radici in LEFT($i$) e RIGHT($i$) siano heap e che l'unico elemento che non rispetta la proprietà di heap sia $A[i]$. L'algoritmo HEAPIFY non fa altro che far "scendere" l'elemento $A[i]$ nell'albero fino a che l'albero con radice in $i$ non sia un heap.

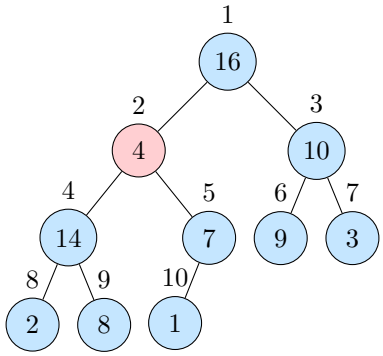
**Esempio**

Supponiamo di avere il vettore: $A = [16, 4, 10, 14, 7, 9, 3, 2, 8, 1]$. La Figura 8.7 mostra l'azione dell'algoritmo HEAPIFY.

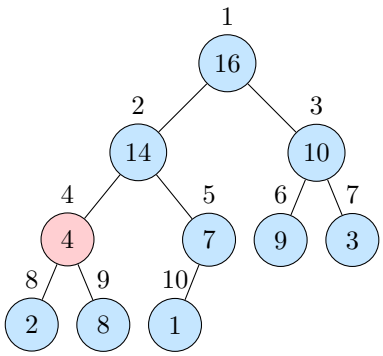
Ad ogni passo, viene determinato il più grande tra $A[i]$, $A[2i]$ e $A[2i + 1]$; il suo indice viene memorizzato nella variabile `max`. Se $A[i]$ è più grande, allora il sottoalbero con radice nel nodo $i$ è un heap e la procedura termina. Altrimenti, uno dei due figli ha l'elemento più grande e $A[i]$ viene scambiato con $A[max]$; in questo modo, il nodo $i$ e i suoi figli soddisfano la proprietà di heap. Il nodo con indice $max$, però, adesso ha il valore originale $A[i]$ e, quindi, il sottoalbero con radice in $max$ potrebbe violare la proprietà di heap. Di conseguenza, deve essere richiamata ricorsivamente la procedura HEAPIFY$(A, max)$ per quel sottoalbero.

```
1  Sx = Left(i)
2  Dx = Right(i)
3  if (Sx ⩽ heapsize && A[i]<A[Sx]) then
4  max = Sx
5  else
6  max = i
7  if (Dx ⩽ heapsize && A[max] < A[Dx]) then
8  max = Dx
9  if (max ≠ i) then
10 Swap(A,i,max)
11 Heapify(A,max)
```

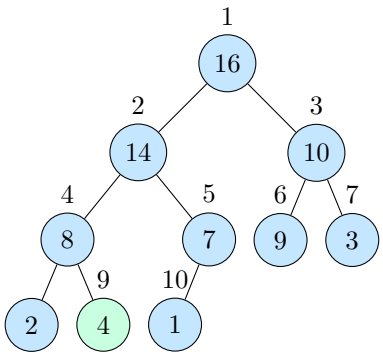
Algoritmo 8.9: HEAPIFY(A,i)




(a) La configurazione iniziale, con $A[2]$ nel nodo $i = 2$ che viola le proprietà di heap.




(b) Le proprietà di heap vengono ripristinate nel nodo 2 scambiando $A[2]$ con $A[4]$; ma questo distrugge la proprietà di heap nel nodo 4, sarà necessario richiamare l'algoritmo Heapify$(A, 4)$.




(c) La chiamata ricorsiva Heapify$(A, 4)$ scambia $A[4]$ con $A[9]$, il nodo 4 è sistemato e la chiamata Heapify$(A, 9)$ non apporta modifiche in quanto le foglie rispettano sempre la proprietà heap.

Figura 8.7

Il tempo di esecuzione di Heapify in un sottoalbero di dimensione $n$ con radice in un nodo $i$ è pari al tempo $\Theta(1)$ per sistemare le relazioni fra gli elementi, più il tempo per eseguire Heapify in un sottoalbero con radice in uno dei figli del nodo $i$. Dunque si può affermare che Heapify ha un tempo di esecuzione pari a:

$$T_{Heapify}(n) = O(\log n) \tag{8.13}$$

L'algoritmo 8.9 non funziona per sequenze che non siano heap. Infatti, se così fosse si potrebbe usare Heapify per cercare il massimo di una sequenza in un tempo minore di $\Theta(n)$.

### 8.5.5 Trasformare un array in un heap: l'algoritmo Costruisci-Heap

È possibile utilizzare l'Algoritmo Heapify dal basso verso l'alto per convertire un array $A[1, \ldots, n]$ in un heap:

- Ovviamente una foglia rispetta sempre le proprietà degli heap, quindi gli ultimi $\lceil \frac{n}{2} \rceil$ elementi dell'array sono già degli heap. Prendendo i padri delle foglie si può applicare Heapify poiché tutti i loro sottoalberi rispettano la proprietà necessaria.
- È sufficiente inserire nello heap solo i primi $\lfloor \frac{n}{2} \rfloor$ elementi, utilizzando Heapify per ripristinare la terza condizione degli heap sul sottoalbero del nuovo elemento.

Si ha quindi l'Algoritmo Costruisci-Heap (Algoritmo 8.10).

```
1 heapsize = n
2 for(i= ⌊n/2⌋ down to 1 ) do
3   Heapify(A,i)
```

Algoritmo 8.10: Costruisci-Heap(A,n)

**Esempio**

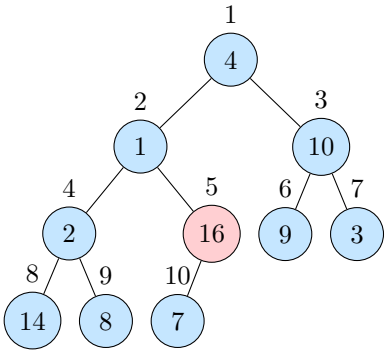
Supponiamo di voler costruire un albero heap a partire dall'array $A = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]$. La Figura 8.8 mostra il funzionamento dell'Algoritmo 8.10.

Ogni chiamata di Costruisci-Heap costa un tempo $O(\log n)$ e ci sono $O(n)$ di queste chiamate. Quindi, il tempo di esecuzione sarà $O(n \log n)$. Questo limite superiore, sebbene corretto, non è asintoticamente stretto.

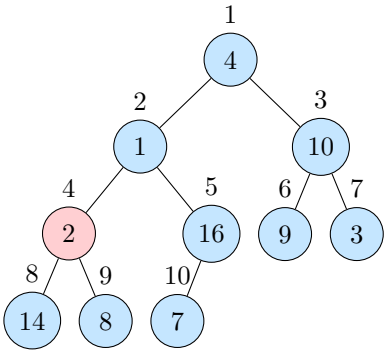
Possiamo ottenere un limite più stretto osservando che il tempo per eseguire Heapify (Algoritmo 8.9) in un nodo varia con l'altezza del nodo nell'albero, e le altezze della maggior parte dei nodi sono piccole. L'analisi più rigorosa si basa sulla proprietà che un heap di $n$ elementi ha un'altezza $\lfloor n/2 \rfloor$ e, per ogni $h$, al massimo $\lceil n/2^{h+1} \rceil$ nodi di altezza $h$.

Il tempo richiesto dalla procedura Heapify quando viene chiamata per un nodo di altezza $h$ è $O(h)$, quindi possiamo dire che il costo totale di Costruisci-Heap è limitato superiormente da:

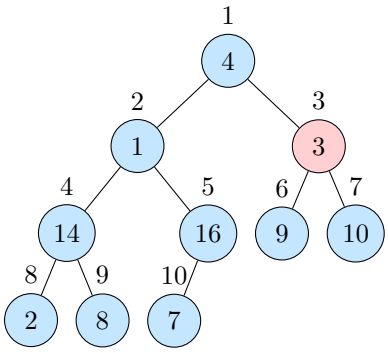
$$\sum_{h=0}^{\lfloor \log n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) = O\Big(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\Big)$$



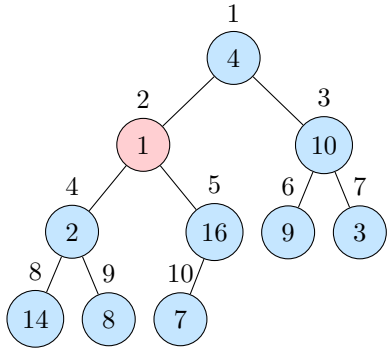

(a) Albero binario rappresentato dall'array $A$, l'indice $i$ punta a $\lfloor 10/2 \rfloor = 5$ prima della chiamata a HEAPIFY$(A, 5)$.



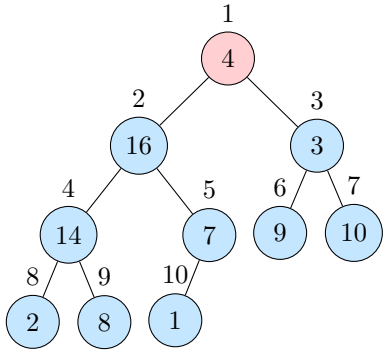

(b) Struttura dati risultante dopo HEAPIFY$(A, 5)$. L'indice $i$ per l'iterazione successiva fa riferimento al nodo 4.



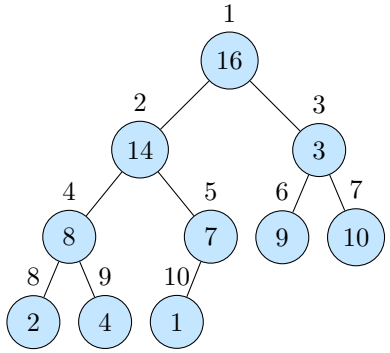

(c)




(d)




(e)




(f) Heap finale alla fine del ciclo for.

Figura 8.8

**Formula**

Derivando entrambi i lati della serie geometrica infinita e moltiplicando per $x$, si ha:

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} \tag{8.14}$$

per $|x| < 1$.

L'ultima sommatoria può essere calcolata ponendo $x = 1/2$ nella formula 8.14:

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

Quindi, il tempo di esecuzione di COSTRUISCI-HEAP può essere limitato così:

$$O\Big(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\Big) = O\Big(n \sum_{h=0}^{\infty} \frac{h}{2^h}\Big) = O(n) \tag{8.15}$$

Dunque possiamo costruire un heap da un array non ordinato in tempo lineare.

### 8.5.6 L'algoritmo HeapSort

Come anticipato, l'algoritmo HeapSort è una variazione di SelectionSort in cui la ricerca dell'elemento massimo è facilitata dal mantenimento della sequenza in uno heap. L'algoritmo inizia utilizzando Costruisci-Heap per costruire un heap nell'array di input $A[1..n]$, dove $n$ è la lunghezza di $A$.

Poiché l'elemento più grande dell'array è memorizzato nella radice $A[1]$, esso può essere inserito nella sua posizione finale corretta scambiandolo con $A[n]$. Se adesso "togliamo" il nodo $n$ dall'heap (diminuendo la variabile *heapsize*), notiamo che i figli della radice restano degli alberi heap, ma la nuova radice potrebbe violare la proprietà dell'heap. Per ripristinare questa proprietà, tuttavia, basta una chiamata di Heapify$(A, 1)$ (Algoritmo 8.9), che lascia un heap in $A[1, .., n-1]$. L'algoritmo HeapSort poi ripete questo processo per il sottoalbero heap di dimensione $n-1$ e così via fino ad un heap di dimensione due.

```
1 Costruisci-Heap(A)
2 for (i = n down to 2) do
3   Swap(A,1,i)
4   heapsize = heapsize -1
5   Heapify(A,1)
```

Algoritmo 8.11: HeapSort(A)

La procedura HeapSort (Algoritmo 8.11) impiega un tempo $O(n \log n)$, in quanto la chiamata di Costruisci-Heap impiega $O(n)$ e ciascuna delle $n-1$ chiamate di Heapify impiega un tempo $O(n \log n)$.

## 8.6 Quick Sort

L'AlgoritmoQuickSort, come MergeSort, è basato sul paradigma divide et impera. Questi sono i tre passi del processo divide et impera per ordinare un generico sottoarray $A[p..r]$.

1. **Divide:** partizionare l'array $A[p..r]$ in due sottoarray $A[p..q-1]$ e $A[q+1..r]$ (eventualmente vuoti) tali che ogni elemento di $A[p..q-1]$ sia minore o uguale ad $A[q]$ che, a sua volta, è minore o uguale ad ogni elemento di $A[q+1..r]$. Calcolare l'indice $q$ come parte di questa procedura di partizionamento.
2. **Impera:** ordinare i due sottoarray $A[p..q-1]$ e $A[q+1..r]$ chiamando ricorsivamente QuickSort.
3. **Combina:** poiché i sottoarray sono già ordinati, non occorre alcun lavoro per combinarli: l'intero array $A[p..r]$ è già ordinato.

```
1 if p<r
2   q=Partiziona(A,p,r)
3   QuickSort(A,p,q)
4   QuickSort(A,q+1,r)
```

Algoritmo 8.12: QuickSort$(A, p, r)$

Per ordinare un intero array $A$ di lunghezza $n$, la chiamata iniziale è QuickSort$(A, 1, n)$.

**Osservazione**

L'algoritmo MergeSort decompone l'array in maniera aritmetica dividendo a metà iterativamente l'array in sottoarray. Così facendo la decomposizione in sottoprobremi risulta più semplice a discapito della fase fusione che è più complessa. L'algoritmo QuickSort decompone in modo "intelligente" (pagando di più in termini computazionali) in modo tale da semplificare la fusione dei sottoarray così ottenuti.

L'algoritmo QuickSort (Algoritmo 8.12) è corretto se valgono le seguenti proprietà:

1. La partizione $A[p..q]$ è strettamente minore della sequenza $A[p..r]$.
2. La partizione $A[q+1..r]$ è strettamente minore della sequenza $A[p..r]$.
3. Al termine di Partiziona$(A, p, r)$ deve valere:

$$\forall\ i \in [p, q]\Big(\forall j \in [q+1, r](A[i] \leq A[j])\Big)$$

che esprime il fatto che ogni elemento di una sottopartizione sinistra deve essere sempre minore od uguale ad un elemento di una sottopartizione di destra.

Le proprietà 1 e 2 possono essere sintetizzate nella proprietà:

$$p \leq q < r \tag{8.16}$$

Se questa proprietà non fosse soddisfatta l'algoritmo non funzionerebbe. Infatti è importante escludere i valori $q = p-1$ e $q = r$ per i seguenti motivi: se $q = p-1$ allora QuickSort$(A, p, p-1)$ starebbe passando una istanza vuota, quindi non farebbe nulla. Il problema nasce con la sequenza a destra che andrà da $p$ ad $r$, andando conseguentemente in loop. In modo analogo si ha per $q = r$.

### 8.6.1 Partizionare l'array: l'algoritmo Partiziona

L'elemento chiave dell'algoritmo 8.12 è la procedura Partiziona che riarrangia il sottoarray $A[p..r]$ sul posto. I passi fondamentali nell'algoritmo di partizionamento (Algoritmo 8.13) sono:

1. Estendere le due partizioni verso l'interno finché non si incontra una coppia non correttamente disposta;
2. Finché le due partizioni non si sono incrociate ($i \geq j$):
   (a) Scambiare le coppie non correttamente disposte;
   (b) Estendere ancora le due partizioni verso l'interno finché non si trova un'altra coppia non correttamente disposta.

Quando viene chiamato l'algoritmo siamo certi che valga la condizione $p < r$. L'algoritmo Partiziona seleziona un valore $x \in A$, detto **pivot** e inserisce nella partizione sinistra gli elementi $a_i \leq x$ e nella partizione a destra gli elementi $a_i \geq x$ (non possiamo mettere il minore stretto perché l'algoritmo deve funzionare per tutte le istanze). Nel nostro algoritmo **il pivot è preso nel primo elemento della sottosequenza**[2].

| Partizione di tutti gli elementi minori o uguali al pivot | Partizione di tutti gli elementi maggiori o uguali al pivot |
|---|---|

Per scorrere la sequenza si fa uso di due indici:

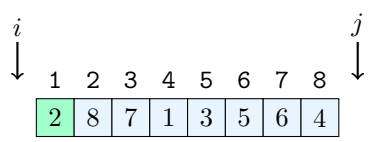
1. Un indice $i$ che partirà da sinistra per essere incrementato fino a quando non trova un valore $y \geq x$
2. Un indice $j$ che partendo da destra viene decrementato fino a quando non trova un valore $z \leq x$

Una volta fermati gli indici si effettua uno scambio e si ripete il processo fino a quando gli indici non si incrociano: vale cioè la condizione $i \geq j$.

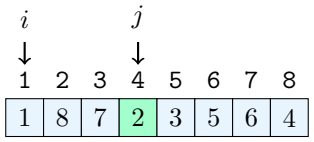
```
1  x=A[p]
2  j=r+1
3  i=p-1
4  repeat
5  repeat
6  j=j-1
7  until (A[j] ≤ x)
8
9  repeat
10 i=i+1
11 until (A[i]≥ x)
12
13 if (i<j)
14 Swap(A,i,j)
15 until (i  ≥ j)
16 return j
```

Algoritmo 8.13: Partiziona(A,p,r)

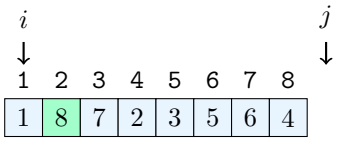
[2] Possiamo prendere qualsiasi elemento dell'array in realtà come pivot, a patto che venga messo in prima posizione successivamente.



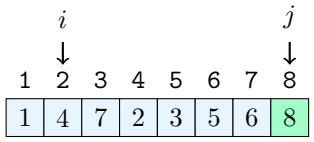

(a) La procedura di ordinamento inizia con l'invocazione di QUICKSORT$(A, 1, 8)$ il quale chiama PARTIZIONA$(A, 1, 8)$. Il pivot (evidenziato in verde) viene selezionato nella posizione $A[1]$ mentre gli indici $i, j$ assumono i valori $i = 0$ e $j = 9$



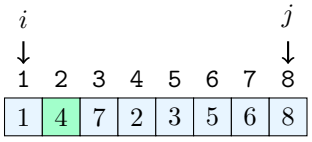

(b) Alla prima iterazione del `repeat-until` le variabili $i$ e $j$ assumono i valori $i = 1$ e $j = 4$ e vengono scambiati gli elementi $A[1] = 2$ e $A[4] = 1$



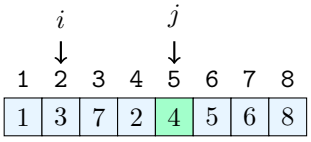

(c) Nella ripetizione successiva gli indici assumono valori $i = 2$ e $j = 1$ e la PARTIZIONA restituisce $q = 1$. A questo punto viene invocato QUICKSORT$(A, 1, 1)$ che termina immediatamente e QUICKSORT$(A, 2, 8)$ il quale invoca PARTIZIONA$(A, 2, 8)$. Il pivot è $A[2] = 8$, gli indici $i, j$ inizialmente valgono $i = 1$ e $j = 9$



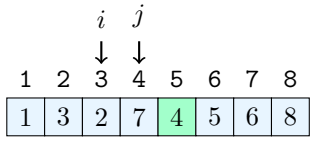

(d) Dopo la prima ripetizione gli indici assumono i valori $i = 2$ e $j = 8$ e si scambiano i valori $A[2] = 8$ e $A[8] = 4$



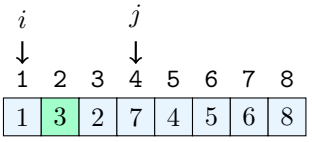

(e) Nella ripetizione successiva gli indici assumono i valori $i = 8$ e $j = 7$ e la procedura restituisce il valore $j = 7$. Ottenuto $q = 7$ si effettuano le chiamate ricorsive QUICKSORT$(A, 2, 7)$ e QUICKSORT$(A, 8, 8)$ la quale termina immediatamente. QUICKSORT$(A, 2, 7)$ attiva PARTIZIONA$(A, 2, 7)$. Il pivot è $A[2] = 4$, gli indici iniziali sono $i = 1, j = 8$.



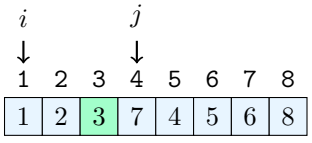

(f) Alla prima ripetizione del `repeat-until` gli indici assumono i valori $i = 2$ e $j = 5$ e si effettua lo swap tra $A[2] = 4$ e $A[5] = 3$.



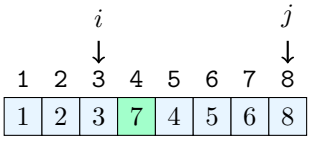

(g) Nella ripetizione successiva gli indici assumono i valori $i = 3$ e $j = 4$ e si effettua lo swap delle celle $A[3] = 7$ e $A[4] = 2$.



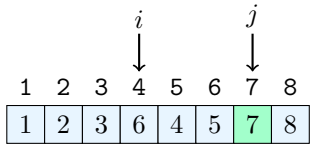

(h) Essendo $i < j$ si effettua un'ulteriore ripetizione del blocco e gli indici assumono i valori $i = 4, j = 3$. La procedura termina restituendo quindi $j = 3$. Adesso $q = 3$ e tocca invocare le procedure QUICKSORT$(A, 2, 3)$ e QUICKSORT$(A, 4, 7)$. QUICKSORT$(A, 2, 3)$ attiva PARTIZIONA$(A, 2, 3)$: il pivot è $A[2] = 3$ e gli indici iniziali assumono i valori $i = 1$ e $j = 4$.



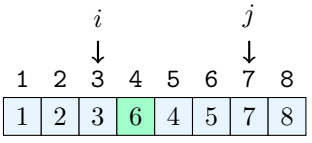

(i) Alla prima ripetizione gli indici $i, j$ assumono i valori $i = 2, j = 3$ e si esegue lo swap delle celle $A[2] = 3$ e $A[3] = 2$.



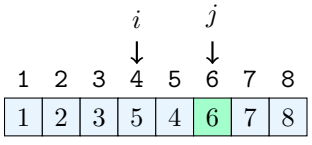

(j) Alla ripetizione successiva $i = 3$ e $j = 2$ e la procedura termina restituendo $j = 2$. A questo punto si continua con la procedura QUICKSORT$(A, 4, 7)$ che invoca PARTIZIONA$(A, 4, 7)$ il quale imposta il pivot nella cella $A[4] = 7$ e gli indici $i, j$ assumono i valori $i = 3$ e $j = 8$.



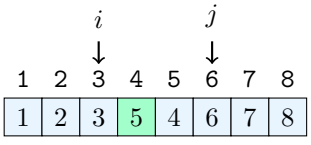

(k) Alla prima ripetizione gli indici assumono i valori $i = 4, j = 7$ e si effettua lo swap delle celle $A[4] = 7$ e $A[7] = 6$.



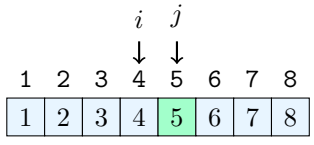

(l) Nella ripetizione successiva si raggiunge la condizione di uscita con $i = 7$ e $j = 6$ e la procedura restituisce $j = 6$. A questo punto si chiama QUICKSORT$(A, 4, 6)$ il quale a sua volta invoca PARTIZIONA$(A, 4, 6)$. Il pivot selezionato è $A[4] = 6$, gli indici inizialmente assumono i valori $i = 3$ e $j = 7$



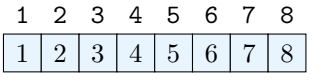

(m) Alla prima ripetizione gli indici assumono i valori $i = 4$ e $j = 6$ e si swappano le celle $A[4] = 6$ e $A[6] = 5$




(n) Alla seconda ripetizione si raggiunge la condizione di uscita e si restituisce $j = 5$. A questo punto QUICKSORT$(A, 4, 5)$ invoca PARTIZIONA$(A, 4, 5)$, il pivot diventa $A[4] = 5$ mentre gli indici $i, j$ assumono i valori $i = 3$ e $j = 6$




(o) A questo punto, durante la prima ripetizione di PARTIZIONA$(A, 4, 5)$ gli indici assumono i valori $i = 4$ e $j = 5$ e si esegue lo scambio di $A[4] = 5$ e $A[5] = 4$




(p) Infine nella ripetizione successiva si raggiunge la condizione di uscita e l'algoritmo termina ottenendo così l'array ordinato.

Figura 8.9

**Correttezza dell'algoritmo Partiziona**

Al ritorno di PARTIZIONA (Algoritmo 8.13) la variabile restituita $j$ deve soddisfare la proprietà 8.16 per garantire la correttezza dell'algoritmo, ovvero:

$$p \leq j < r$$

Per verificare ciò basta dimostrare che siano impossibili i seguenti casi:

$$j < p \tag{8.17}$$
$$j \geq r \tag{8.18}$$

Per fare ciò basta verificare che non si verificano i casi:

- $j = p - 1$, essendo $i = p - 1$, è l'unico $j < p$ che potrebbe verificarsi vista la scrittura del nostro algoritmo;
- $j = r$, anche se $j = r + 1$ l'algoritmo 8.13 fa almeno un decremento sulla variabile $j$.

**Prima dimostrazione** Partendo dal secondo punto, dimostriamo che $j$ non può mai assumere il valore $r$. Per restituire $j = r$ significa che gli indici $i$ e $j$ si sono incrociati su $r$.

Notiamo il fatto che $j$ può fermarsi sul valore $r$ solo nella prima iterazione del `repeat` esterno. Quindi bisogna dimostrare che l'indice $i$ non arrivi mai ad $r$ essendo questo l'unico modo che ha per uscire dal ciclo con $j = r$.

Alla prima iterazione del `repeat-until` esterno l'indice $j$ viene decrementato ad $r$ ed essendo falsa la condizione $A[r] \leq A[p] = x$ l'algoritmo passa alla linea 8 effettuando un incremento dell'indice $i$ portandolo a $i = p$.

A questo punto si esce dal secondo blocco `repeat-until` e l'esecuzione passa alla linea 11 che esegue lo scambio tra $A[p]$ e $A[r]$. Dato che $i = p$ e $j = r$ con $p < r$ verrà eseguita una seconda iterazione del `repeat-until` esterno che decrementerà nuovamente l'indice $j$.

■

**Seconda dimostrazione** Per dimostrare che $j \neq p - 1$ basta garantire che il primo `repeat-until` interno non si ripeta all'infinito.

Poiché non possiamo garantire di essere alla prima iterazione non è detto che $x$ sia ancora nella prima posizione della sequenza poiché potrebbero esserci stati degli scambi, ma se questi sono avvenuti significa certamente che in $A[p]$ c'è un valore minore od uguale a $x$ e ciò garantisce che $j$ si fermerà sicuramente in $p$ (questo è un ragionamento molto astratto, infatti se ci sono stati degli scambi allora $i > p$ e quindi $j$ si fermerà sicuramente prima di arrivare a $p$ essendo la condizione di uscita).

Ora poiché la variabile $i$ può essere solo incrementata è evidente che non potrà essere minore di $p$ ma allora ciò significa che se $j$ arriva a $p$ in quella stessa iterazione la condizione $i \geq j$ sarà verificata e l'algoritmo terminerà con un valore $j \geq p$.

■

**Ordinamento**

L'algoritmo PARTIZIONA garantisce che, alla sua terminazione, siano verificate le seguenti proprietà:

$$\forall\ z : p \leq z \leq j \qquad A[z] \leq x \tag{8.19}$$
$$\forall\ t : j + 1 \leq t \leq r \qquad A[t] \geq x \tag{8.20}$$

Queste ultime due possono essere unite in:

$$\forall\ z,t : p \leq z \leq j < t \leq r \qquad A[z] \leq x \leq A[t] \tag{8.21}$$

che rappresenta la proprietà da garantire:

$$\forall\ i : p \leq i \leq q\ \wedge\ \forall\ j :\ q + 1 \leq j \leq r \qquad A[i] \leq A[j] \tag{8.22}$$

### 8.6.2 Prestazioni di QuickSort

Le tecniche viste fino a questo momento per studiare il costo computazionale degli algoritmi cercano di calcolare il costo di ogni linea e moltiplicarlo per il numero di esecuzioni. Nel caso del QUICKSORT il costo dell'algoritmo dipende dalla bontà dell'input. Facendo però delle analisi a priori è possibile trovare un limite asintotico superiore per PARTIZIONA. Infatti, sapendo che $p < r$ si ha che la somma dei `repeat-until` interni sarà $n + 1$ o $n + 2$ (poiché l'algoritmo praticamente termina o con $i = j$ se l'input è dispari oppure con $i = j + 1$ se l'input è pari) mentre il corpo dell'if al massimo viene eseguito per $n/2$ volte e dunque:

$$T_{Partiziona} = \Theta(n) + O(n) = \Theta(n) \tag{8.23}$$

Scrivendo l'equazione di ricorrenza di QUICKSORT si ha:

$$T_{QuickSort}(n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ T_{QuickSort}(q) + T_{QuickSort}(n - q) + T_{Partiziona}(n) & \text{se } n > 1 \end{cases} \tag{8.24}$$

**Caso migliore e caso peggiore per QuickSort**

A causa dell'algoritmo PARTIZIONA non è possibile sapere a priori come viene diviso l'input tra le due chiamate poiché dipende da come viene scelto il pivot e dall'istanza di input. Ma posso fare i seguenti ragionamenti:

- Se tutti gli elementi sono uguali PARTIZIONA divide la sequenza in due parti quasi uguali; In questo caso $q = \lfloor n/2 \rfloor$ e si ha $T_{QuickSort}(n) = \Theta(n \log n)$ essendo

$$T_{QuickSort}(n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ T_{QuickSort}(\frac{n}{2}) + T_{QuickSort}(\frac{n}{2}) + \Theta(n) & \text{se } n > 1 \end{cases} \tag{8.25}$$

  come visto per MERGESORT in 8.6.

- Se invece si ha in input una sequenza già ordinata, questa viene suddivisa in una partizione da un solo elemento (quella di sinistra) e l'altra con gli elementi restanti. In questo caso si avrà:

$$T_{QuickSort}(n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ T_{QuickSort}(1) + T_{QuickSort}(n-1) + \Theta(n) & \text{se } n > 1 \end{cases} \tag{8.26}$$

  e quindi $T_{QuickSort}(n) = \Theta(n^2)$ come visto nell'esempio del calcolo del fattoriale in 6.2.

Da questi brevi calcoli si ottengono funzioni che non coincidono asintoticamente. C'è da chiedersi se queste due tipologie di istanze rappresentano il caso migliore e il caso peggiore per questo algoritmo. Si osserva che qualsiasi istanziazione dell'equazione di ricorrenza 8.24 genererà sempre un albero binario completo diverso che avrà sempre un numero fissato di foglie, ovvero tante quanti sono gli elementi da ordinare. Cioè:

$$\begin{aligned} \#_{foglie} &= n \\ \#_{nodi} &= 2n - 1 \end{aligned}$$

Fino a questo momento abbiamo ottenuto la funzione che risolve la funzione di ricorrenza lavorando sull'albero di ricorsione che ne deriva: sommando, livello per livello, i contributi si otteneva il costo totale. Dato che PARTIZIONA divide la sequenza in due parti di grandezza $q$ ed $n - q$ si avrà che ogni livello avrà sempre un costo lineare e quindi ci aspetteremo di ottenere costi più alti al crescere dell'altezza degli alberi, al variare dell'albero di ricorrenza.

Consideriamo il caso dell'albero generato dall'equazione di ricorrenza 8.26 mostrato in figura 8.10.

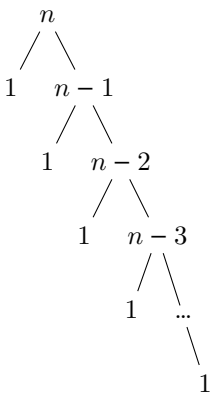



Figura 8.10: Albero di ricorrenza dell'equazione 8.26

Per questo tipo di albero il livello 0 e il livello 1 hanno lo stesso contributo, però dal livello 2 in poi il contributo di ogni livello è pari a:

$$\forall\ l \geq 1 \qquad Costo_l = n - l + 1$$

Di conseguenza il tempo di esecuzione sarà:

$$T(n) = n + \sum_{l=1}^{n} (n - l + 1) = \Theta(n^2) \tag{8.27}$$

Nel caso dell'equazione 8.25 l'albero di ricorrenza è fatto come mostrato in figura 8.11. E il costo sarà dato dall'equazione:

$$T(n) = \sum_{l=0}^{h} n = n \sum_{l=0}^{h} 1 = n \log_2 n \tag{8.28}$$

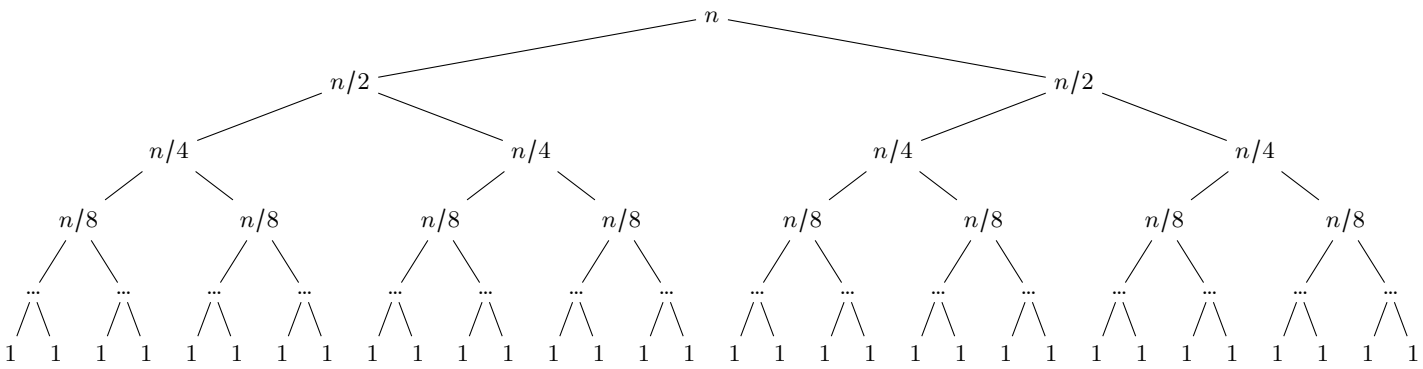



Figura 8.11: Albero di ricorrenza dell'equazione 8.25

Si dimostra così che queste due tipologie di alberi rappresentano, nel primo caso, il caso peggiore e il caso migliore per l'algoritmo QuickSort. Essendo le equazioni non coincidenti asintoticamente resta da studiare il caso medio.

### Partizionamento bilanciato

Il bilanciamento del partizionamento influisce sulla ricorrenza che descrive il tempo di esecuzione.

Supponiamo, per esempio, che l'algoritmo di partizionamento produca sempre una ripartizione proporzionale 3 a 1, che a prima vista potrebbe sembrare molto sbilanciata. In questo caso si ottiene la ricorrenza:

$$T(n) \leq T(3n/4) + T(n/4) + cn \tag{8.29}$$

sul tempo di esecuzione di QuickSort, dove abbiamo esplicitamente incluso la costante $c$ nascosta nel termine $\Theta(n)$. La Figura 8.12 illustra l'albero di ricorsione per questa ricorrenza. Notiamo che ogni livello dell'albero ha un costo $cn$, finché non viene raggiunta una condizione al contorno alla profondità $\log_4 n = \Theta(\lg n)$, dopo la quale i livelli hanno al massimo un costo $cn$.

La ricorsione termina alla profondità $\log_{4/3} n = \Theta(\lg n)$. Il costo totale di QuickSort è dunque $O(n \log n)$. Pertanto, con una ripartizione proporzionale 3 a 1 a ogni livello di ricorsione, che intuitivamente sembra molto vicina al caso peggiore, QuickSort viene eseguito nel tempo $O(n \log n)$ - asintoticamente uguale a quello che si ha nel caso migliore.

In effetti anche una ripartizione 99 a 1 determina un tempo di esecuzione pari a $O(n \log n)$. La ragione è che qualsiasi ripartizione con *proporzionalità costante* produce un albero di ricorsione di profondità $\Theta(\log n)$, dove il costo in ogni livello è lineare.

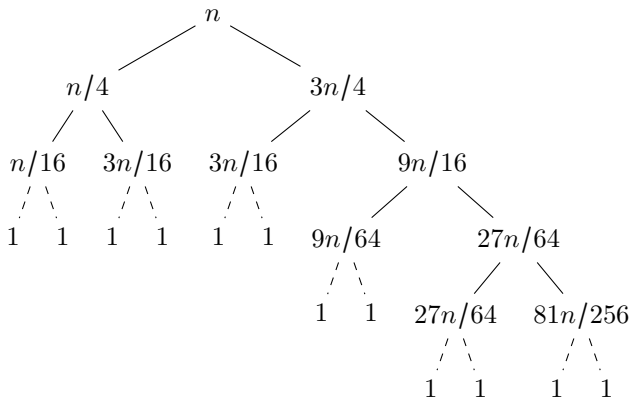



Figura 8.12: Un'albero di ricorsione per QuickSort quando Partiziona genera sempre una ripartizione 3 a 1, determinando un tempo di esecuzione pari a $O(n \log n)$. I nodi mostrano le dimensioni dei sottoproblemi, con i costi per livello a destra. Questi costi includono la costante $c$ implicita nel termine $\Theta(n)$.

### Analisi del caso medio per QuickSort

Per trovare una funzione di tempo medio bisogna applicare un'analisi simile a quella vista per il caso medio di InsertionSort(Algoritmo 8.1). Il nostro obiettivo sarà quello di definire un'equazione di ricorrenza di tempo medio basato su determinate funzioni di tempo medio sui sottoproblemi. Infatti, essendo l'approccio ricorsivo un modo di decomporre il problema, se vogliamo decomporre il tempo medio in istanze di sottoproblemi si deve capire prima quella che deve essere la computazione media locale.

Poiché Partiziona determina un valore $q$ per ogni nodo locale dell'albero di ricorsione, fissato un valore $q$ si determinano ben due sottoproblemi. L'idea sarà quella di generare due sottoproblemi di dimensione $q$ ed $n - q$, fissato un valore $q$. Senza ledere di generalità ragioneremo su delle sequenze che non contengono duplicati[3]. In questo caso, le dimensioni delle partizioni sono univocamente determinate dal **rango** del pivot.

[3] Maggiori saranno i duplicati uguali al pivot, maggiori saranno le probabilità di avere partizionamenti bilanciati a metà.

**Rango**

Data una sequenza e un pivot $x$, il **rango** del pivot è il numero di elementi minori o uguali ad $x$ nella sequenza. Di conseguenza, data una sequenza di $n$ valori si ha:

$$1 \leq r(x) \leq n$$

Esempio

Sia $A = [1, 3, 7, 9, 12, 15]$ con $x = 1$. Allora il rango di $x$ sarà: $r_A(x) = 1$

Esiste una corrispondenza tra il rango di x e il valore di q: infatti la scelta del rango determina le dimensioni delle partizioni. Infatti, come si vede nella Tabella 8.1, per $r_A(x) \geq 2$ si ha che $q = r_A(x) - 1$.

| $r_A(x)$ | q |
|---|---|
| 1 | 1 |
| 2 | 1 |
| 3 | 2 |
| 4 | 3 |
| ⋮ | ⋮ |
| $n-1$ | $n-2$ |
| $n$ | $n$ |

Tabella 8.1: Corrispondenza tra il rango del pivot e il valore di $q$

Di fatto scegliere un pivot significa trovare un rango che determina il valore di q. Quindi è possibile pensare di dividere le varie istanze di sequenze di n elementi in tante classi di equivalenza, dove ciascuna classe corrisponde all'insieme delle istanze con un determinato rango.

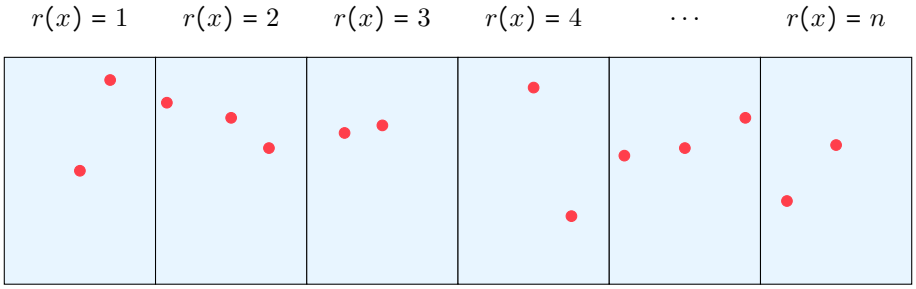



Figura 8.13: Suddivisioni delle istanze (punti rossi) in classi di equivalenze. Ogni istanza con lo stesso rango appartiene alla stessa classe di equivalenza.

Fatta questa suddivisione delle istanze posso esprimere il tempo medio come segue:

$$T_m(n) = \frac{1}{n}[\sum_{r=1}^{n} T_M^r(n)] \tag{8.30}$$

dove $T_M^r(n)$, fissato un rango $r_A(x)$, esprime il tempo medio su sequenze di dimensione n.

Ovvero:

$$T_M^r(n) = T_M(q_r) + T_M(n - q_r) + \Theta(n) \tag{8.31}$$

Sostituendo nell'equazione 8.30 si ottiene:

$$T_M(n) = \frac{\sum_{r=1}^{n}(T_M(q_r) + T_M(n - q_r) + \Theta(n))}{n} \tag{8.32}$$

È possibile semplificare l'equazione 8.32 togliendo la dipendenza dal rango:

$$\begin{aligned} T_M(n) &= \frac{1}{n}\Big(T_M(1) + T_M(n-1) + \Theta(n) + \sum_{r=2}^{n}\big(T_M(q_r) + T_M(n-q_r) + \Theta(n)\big)\Big) \\ &= \frac{1}{n}\Big(\Theta(1) + O(n^2) + \Theta(n) + \sum_{q=1}^{n-1}\big(T_M(q) + T_M(n-q) + \Theta(n)\big)\Big) \\ &= \frac{1}{n}\Big(O(n^2) + \sum_{q=1}^{n-1}\Theta(n) + 2\cdot\sum_{q=1}^{n-1}T_M(q)\Big) \\ &= \frac{1}{n}\Big(O(n^2) + \Theta(n^2) + 2\cdot\sum_{q=1}^{n-1}T_M(q)\Big) \\ &= \frac{1}{n}\Big(\Theta(n^2) + 2\cdot\sum_{q=1}^{n-1}T_M(q)\Big) \\ &= \frac{\Theta(n^2)}{n} + \frac{2}{n}\sum_{q=1}^{n-1}T_M(q) \\ &= \Theta(n) + \frac{2}{n}\sum_{q=1}^{n-1}T_M(q) \end{aligned} \tag{8.33}$$

Resta da trovare una soluzione per l'equazione $T_M(q)$. Per farlo useremo il **metodo di sostituzione**. L'ipotesi alla base del metodo di sostituzione è la seguente:

$$\exists\, c\Big(\forall n \geq 2(T_M(n) \leq cn\log_2 n)\Big) \tag{8.34}$$

**Dimostrazione:** Si dimostra per induzione:

- **Caso base:** Sia $n = 2$. $T_M(2)$ deve essere minore o uguale di una qualche costante c moltiplicato per $\log_2 2$, ovvero:

  $$T_M(2) \leq c\cdot 2\log_2 2 = 2c$$

  ma

  $$T_M(2) = \Theta(2) + \frac{2}{2}\sum_{q=1}^{1}T_M(q) = \Theta(2) + \Theta(1)$$

  queste due $\Theta$ sono costanti che discendono da due operazioni diverse dell'algoritmo (una è la costante di PARTIZIONA mentre la seconda è la costante data dal confronto per vedere se siamo in un caso base o meno). Possiamo quindi rinominarle in $k_1$ e $k_2$:

  $$T_M(2) = k_1 + k_2$$

  bisogna dimostrare quindi che è vera la proprietà:

  $$k_1 + k_2 = 2c$$

  Presa

  $$c > \frac{k_1 + k_2}{2} \tag{8.35}$$

  il caso base è soddisfatto.

- **Passo induttivo ($n \geq 2$):** Sia vera la tesi per qualsiasi numero minore di n. Si ha

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

  Essendo $1 \leq q \leq n-1$ è sicuramente $q < n$ e quindi possiamo maggiorare usando il passo induttivo:

  $$T_M(q) \leq cq\log_2 q$$

  Si ha così:

  $$\begin{aligned} T_M(n) &= \Theta(n) + \frac{2}{n}\sum_{q=1}^{n-1}T_M(q) \\ &\leq \Theta(n) + \frac{2}{n}\sum_{q=1}^{n-1}(cq\log_2 q) \end{aligned} \tag{8.36}$$

Usando la proprietà 8.38 si ha poi:

$$\begin{aligned}\Theta(n) + \frac{2}{n}\sum_{q=1}^{n-1}(cq\log_2 q) &\leq \Theta(n) + \frac{2c}{n}\left(\frac{n^2\log n}{2} - \frac{n^2}{8}\right)\\ &= \Theta(n) + cn\log n - \frac{cn}{4}\\ &\leq cn\log n \end{aligned} \tag{8.37}$$

■

**Proposizione**

Vale la seguente proprietà:

$$\sum_{q=1}^{n-1} q\log q \leq \frac{n^2\log nn}{2} - \frac{n^2}{8} \tag{8.38}$$

**Dimostrazione** Si ha:

$$\begin{aligned}\forall\ i \leq q \leq n-1 \qquad q < n &\Rightarrow \log_2 q \leq \log_2 n\\ &\Rightarrow q\log_2 q \leq q\log_2 n\\ &\Rightarrow \sum_{q=1}^{n-1} q\log_2 q \leq \sum_{q=1}^{n-1} q\log_2 n =\\ &= \log_2 n\sum_{q=1}^{n-1} q\\ &= \log_2 n\left(\frac{n(n-1)}{2}\right)\\ &= \log_2 n\left(\frac{n^2-2}{2}\right.\\ &= \frac{n^2\log_2 n}{2} - \frac{n\log_2 n}{2}\end{aligned}$$

■

Si dimostra in questo modo che l'equazione di ricorrenza 8.30 ha una soluzione del tipo polilogaritmica:

$$T_{QS_m} = \Theta(n^2\log_2 n) \tag{8.39}$$

## 8.7 Analisi degli algoritmi di ordinamento

Si dimostra che il **problema generale dell'ordinamento** non può essere risolto in tempo minore di $\Omega(n\log n)$ se risolto mediante algoritmi **basati sui confronti**.

Ogni algoritmo di ordinamento per confronti si può vedere come un processo decisionale per individuare una *permutazione ordinata* di una sequenza. Ovvero, dati due elementi $a_i$ e $a_j$, eseguiamo uno dei test $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i > a_j$ oppure $a_i > a_j$ per determinare il loro ordine relativo. In questa sezione supponiamo, senza perdere di generalità, che tutti gli elementi di input siano distinti. Fatta questa ipotesi, confronti della forma $a_i = a_j$ sono inutili, quindi possiamo supporre che non saranno fatti confronti di questo tipo.

### 8.7.1 Alberi di decisione

Gli ordinamenti per confronti possono essere visti astrattamente in termini di **alberi di decisione**. Un albero di decisione è un albero binario che rappresenta i confronti fra elementi che vengono effettuati da un particolare algoritmo di ordinamento che opera su input di una data dimensione. Il controllo, lo spostamento dei dati e tutti gli altri aspetti dell'algoritmo vengono ignorati. La figura 8.14 illustra un esempio di albero di decisione che corrisponde all'algoritmo Insertion-Sort che opera su una sequenza di input di tre elementi.

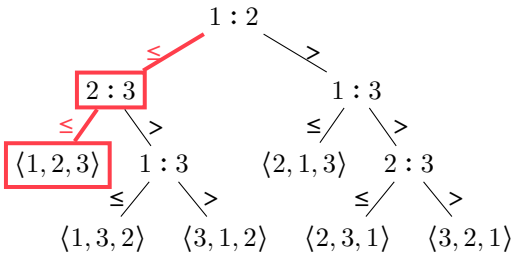



Figura 8.14: Albero di decisione di un algoritmo che opera su tre elementi.

In un albero di decisione, ogni nodo interno è annotato con $i : j$ per qualche $i$ e $j$ nell'intervallo $1 \leq i, j \leq n$, dove $n$ è il numero di elementi nella sequenza di input. Ogni foglia è annotata con una permutazione $\langle \pi(1), \pi(2), ..., \pi(n) \rangle$.

L'esecuzione dell'algoritmo di ordinamento corrisponde a tracciare un cammino semplice dalla radice dell'albero di decisione fino a una foglia. Ogni nodo interno rappresenta un confronto $a_i \leq a_j$. Il sottoalbero sinistro detta i successivi confronti per $a_i \leq a_j$; il sottoalbero destro detta i successivi confronti per $a_i \geq a_j$.

Quando raggiunge una foglia, l'algoritmo ha stabilito l'ordinamento

$$a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$$

Poiché qualsiasi algoritmo di ordinamento corretto deve essere in grado di produrre ogni permutazione del suo input, una *condizione necessaria* affinché un ordinamento per confronti sia corretto è che ciascuna delle $n!$ permutazioni di $n$ elementi appaia come una delle foglie dell'albero di decisione e che ciascuna di queste foglie sia raggiungibile dalla radice attraverso un percorso che corrisponde a una effettiva esecuzione dell'ordinamento per confronti (queste foglie saranno chiamate "raggiungibili").

La **lunghezza del cammino semplice più lungo** dalla radice di un albero di decisione a una delle due foglie raggiungibili rappresenta il **numero di confronti** che svolge il corrispondente algoritmo di ordinamento nel caso peggiore. Di conseguenza il numero di confronti nel caso peggiore per un dato algoritmo di ordinamento basato sui confronti è *uguale all'altezza del suo albero di decisione.*

**Teorema**

Sia $T$ un albero di decisione che ordina $n$ elementi distinti. $T$ ha un altezza almeno pari a $\Theta(n \log_2 n)$.

**Dimostrazione** Da quanto detto in precedenza T ha $n!$ foglie, se ne avesse di più sarebbe ridondante, se ne avesse di meno non ha ancora fatto tutti i confronti e quindi sarebbe incompleto. Essendo gli esiti dei confronti solo di due tipi (minore o maggiore) si ha che $T$ è un albero binario (il quale avrà al massimo $2^h$ foglie). Da queste informazione possiamo dire dunque che :

$$n! \leq 2^h$$

Per calcolare il tempo medio di un algoritmo basterà quindi dividere la **lunghezza del percorso esterno** (LPE), ovvero la somma delle lunghezze dei cammini dalla radice alle foglie, per il numero medio di confronti:

$$T_M(n) = \frac{LPE}{n!}$$

Dato un albero saprò calcolare il percorso medio, ma in generale devo trovare un albero che minimizza l'espressione precedente. Poiché non è possibile minimizzare il numero $n$ si può pensare di minimizzare la lunghezza del percorso esterno. Gli alberi completi (vedi 3.1.1) rispondono a questa necessità. Infatti questi alberi, fissato un numero di nodi, avranno la minor altezza possibile, minimizzando così la quantità $LPE$.

In un albero completo valgono le seguenti proprietà:

$$N_h + N_{h-1} = n! \tag{8.40}$$

$$N_h + 2N_{h-1} = 2^h \tag{8.41}$$

dove $N_h$ è il numero di foglie alla profondità $h$ e $N_{h-1}$ è il numero di foglie alla profondità $h$.

Sottraendo la prima equazione alla seconda si ottiene:

$$N_{h-1} = n! - 2^h \tag{8.42}$$

Abbiamo quindi:

$$\begin{aligned} LPE &= h \cdot N_h + (h-1)N_{h-1} \\ &= h \cdot N_h + h \cdot N_{h-1} - N_{h-1} \\ &= h \cdot (N_h + N_{h-1}) - N_{h+1} \\ &= h \cdot n! - N_{h-1} \\ &= h \cdot n! - 2^h + n! \end{aligned} \tag{8.43}$$

Il numero medio di confronti sarà allora:

$$\begin{aligned} T_M(n) &= \frac{LPE}{n!} \\ &= \frac{h \cdot n! - 2^h + n!}{n!} \\ &= h - \frac{2^h}{n!} + 1 \end{aligned}$$

Sapendo che $h = \log n!$ si ha:

$$T_M(n) = \log n! - \frac{2^{\log n!}}{n!} + 1 = \log n! \tag{8.44}$$

Ma abbiamo già dimostrato che $\log n! = \Theta(n \log n)$ e quindi possiamo concludere dicendo che il tempo di esecuzione di un algoritmo di ordinamento per una sequenza arbitraria di input $n$ non può essere, nel caso medio e nel caso peggiore, migliore di $\Theta(n \log n)$.

■

| **Algoritmo** | **Tempo di esecuzione nel caso peggiore** | **Tempo di esecuzione nel caso medio** |
|---|---|---|
| **Insertion Sort** | $\Theta(n^2)$ | $\Theta(n^2)$ |
| **Mergesort** | $\Theta(n \log n)$ | $\Theta(n \log n)$ |
| **Heapsort** | $O(n \log n)$ | - |
| **Quicksort** | $\Theta(n^2)$ | $\Theta(n \log n)$ |

Tabella 8.2: Confronto dei tempi di esecuzione degli algoritmi di ordinamento

# ESERCIZI ED APPROFONDIMENTI

## 9.1 RICORSIONE E NOTAZIONE ASINTOTICA

### 9.1.1 Alberi di ricorsione

Esercizio

Si risolva la seguente ricorrenza, calcolandone l'**andamento asintotico**:

$$T(n) = \begin{cases} 1 & \text{se } n \leq 2 \\ \sqrt[3]{n^2} \cdot T(\sqrt[3]{n^2}) + n & \text{altrimenti} \end{cases}$$

**Svolgimento** Riprendendo la formula generale delle equazioni di ricorrenza (Equazione 6.3) notiamo che la ricorrenza proposta segue una forma analoga, e potrà essere descritta di conseguenza attraverso quello che in generale viene chiamato **albero variabile** in quanto il numero dei sottoproblemi è espresso in funzione dell'input di ciascun sottoproblema:

- Il numero di chiamate ricorsive
- Una funzione della dimensione dell'input dei sottoproblemi
- Il contributo locale

$$T(n) = \sqrt[3]{n^2} \cdot T(\sqrt[3]{n^2}) + n$$

Sfruttando le proprietà delle potenze, l'equazione della ricorrenza può essere riscritta nel modo seguente:

$$T(n) = n^{\frac{2}{3}} \cdot T(n^{\frac{1}{3}}) + n$$

Come già anticipato, il numero di chiamate ricorsive è espresso mediante una funzione nella dimensione dell'input e non un valore costante. Nella descrizione dell'albero di ricorrenza dovremo quindi tenerne conto per il calcolo del contributo di ciascun livello come mostrato in tabella 9.1.

L'albero di ricorsione ha in radice un solo nodo, ovvero la prima chiamata all'Algoritmo ricorsivo che riceve in ingresso un input di dimensione $n$. Tale procedura ha un contributo locale lineare sulla dimensione dell'input che rende facile il calcolo del costo del livello. Tale procedura eseguirà $n^{\frac{2}{3}}$ chiamate che ricevono in ingresso un input pari a $n^{\frac{1}{3}}$. Il costo del livello sarà dato quindi dalla somma dei contributi locali di ciascun sottoproblema, ovvero:

$$\underbrace{n^{\frac{1}{3}} + \ldots n^{\frac{1}{3}}}_{n^{\frac{2}{3}} \text{ volte}} = n^{\frac{1}{3}} \cdot n^{\frac{2}{3}} = n^{\frac{1+2}{3}} = n^{\frac{3}{3}} = n^1 = n$$

Un singolo sottoproblema del secondo livello chiamerà a sua volta $(n^{\frac{1}{3}})^{\frac{2}{3}}$ sottoproblemi. Per calcolare il numero di sottoproblemi generato dai sottoproblemi del livello precedente basterà quindi moltiplicare il numero dei sottoproblemi generati nel livello precedente per il numero di sottoproblemi generato da ciascun padre, ovvero:

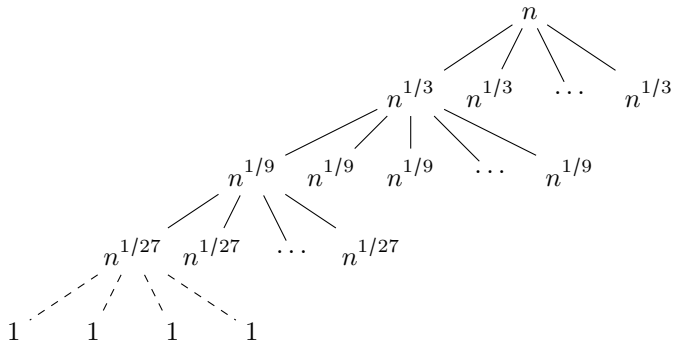
$$n^{\frac{2}{3}} \cdot (n^{\frac{1}{3}})^{\frac{2}{3}} = n^{\frac{2}{3}} \cdot n^{\frac{2}{9}} = n^{\frac{6+2}{9}} = n^{\frac{8}{9}}$$

il contributo del livello sarà quindi dato nuovamente dal prodotto tra il contributo locale, lineare sulla dimensione dei sottoproblemi, e il numero di sottoproblemi del livello. La dimensione di ciascun sottoproblema è pari a $(n^{\frac{1}{3}})^{\frac{1}{3}} = n^{\frac{1}{9}}$. Si ha allora:

$$n^{\frac{8}{9}} \cdot n^{\frac{1}{9}} = n$$

| Livello | Input | Contributo | Rami | Costo livello |
|---|---|---|---|---|
| 0 | $n$ | $n$ | 1 | $n$ |
| 1 | $n^{\frac{1}{3}}$ | $n^{\frac{1}{3}}$ | $n^{\frac{2}{3}}$ | $n$ |
| 2 | $n^{\frac{1}{9}}$ | $n^{\frac{1}{9}}$ | $n^{\frac{8}{9}}$ | $n$ |
| 3 | $n^{\frac{1}{27}}$ | $n^{\frac{1}{27}}$ | $n^{\frac{26}{27}}$ | $n$ |
| $i$ | $n^{\frac{1}{3^i}}$ | $\vdots$ | $\vdots$ | $n$ |
| **Costo totale** | | | | $\sum_{i=0}^{h} Costo_{Livello_i}$ |

Tabella 9.1




Osservando la dimensione dell'input al crescere dei livelli è possibile notare che questa segue una successione esprimibile come:

$$s_n(h) = n^{\frac{1}{3^h}}$$

Grazie a tale successione è possibile determinare l'altezza dell'albero. Infatti, sapendo che il caso base è raggiunto quando la dimensione dell'input è minore o uguale a 2, risolvendo la disequazione ottenuta è possibile ricavare l'altezza necessaria per il calcolo della stima asintotica dell'equazione di ricorrenza proposta. Si ha quindi:

$$\begin{aligned}
& n^{\frac{1}{3^h}} \leq 2 \\
& \iff \log_2(n^{\frac{1}{3^h}}) \leq \log_2 2 \\
& \iff \frac{1}{3^h} \log_2 n \leq 1 \\
& \iff \frac{3^h}{\log_2 n} \leq 1 \\
& \iff 3^h \leq \log_2 n \\
& \iff h \leq \log_3(\log_2 n)
\end{aligned}$$

Il contributo totale dato dall'albero di ricorrenza è dato dalla somma dei contributi dei singoli livelli per il numero totale di livelli (l'altezza dell'albero):

$$\begin{aligned}
T(n) &= \sum_{i=0}^{h} n \\
&= n \sum_{i=0}^{h} 1 \\
&= n(h+1) \\
&= n\big(\log_3 \log_2 n + 1\big) \\
&= n(\log_3 \log_2 n) + n \\
&= \Theta(n \log_3 \log_2 n)
\end{aligned}$$

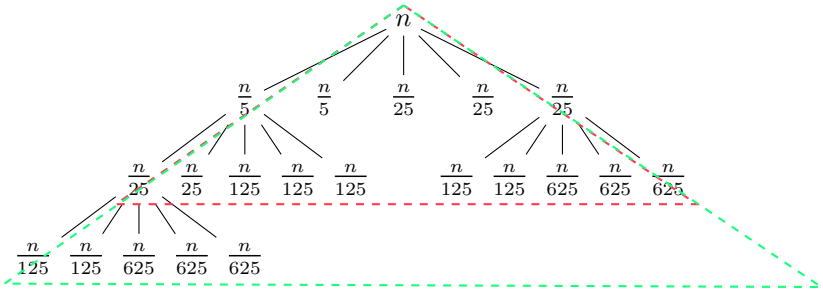
■

Esercizio

Si risolva la seguente ricorrenza, calcolandone l'**andamento asintotico**:

$$T(n) = \begin{cases} 1 & \text{se } n \leq 1 \\ 2 \cdot T(\frac{n}{5}) + 3 \cdot T(\frac{n}{25}) + 4 & \text{altrimenti} \end{cases}$$

**Svolgimento** Iniziamo costruendo l'albero di ricorsione. Ciascuna chiamata ricorsiva genera cinque sottochiamate ricorsive: due sottochiamate con input ridotto di un quinto dell'input della chiamata padre e tre sottochiamate in cui l'input viene ridotto di un venticinquesimo.




Osservando l'albero appena costruito notiamo subito l'esistenza di due percorsi che hanno velocità diverse: i sottoproblemi che vedono ridurre il proprio input di un venticinquesimo finiranno molto prima rispetto a quelli dove l'input si riduce, chiamata dopo chiamata, di solo un quinto. Per ricavare una stima asintotica dell'equazione di ricorrenza sarà necessario quindi studiare le stime dei due alberi di ricorsione $T_1(n) \leq T(n) \leq T_2(n)$ evidenziati rispettivamente in verde e rosso come mostrato in figura.

| Livello | $\text{Input}_1$ | $\text{Input}_2$ | Contributo locale | Rami | Costo livello |
|---|---|---|---|---|---|
| 0 | $n$ | $n$ | 4 | 1 | 4 |
| 1 | $\frac{n}{5}$ | $\frac{n}{25}$ | 4 | 5 | 20 |
| 2 | $\frac{n}{25}$ | $\frac{n}{625}$ | 4 | 25 | 100 |
| 3 | $\frac{n}{125}$ | $\frac{n}{15625}$ | 4 | 125 | 500 |
| $i$ | $\frac{n}{5^i}$ | $\frac{n}{5^{2i}}$ | 4 | $5^i$ | $5^i \cdot 4$ |

Tabella 9.2

Dalla tabella 9.2 si osserva che l'input dei sottoproblemi appartenenti al sottoalbero indotto $T_1$ segue un pattern definito dalla successione:

$$\frac{n}{5^h}$$

dove $h$ rappresenta l'altezza di $T_1$. Per determinare tale altezza bisogna determinare il livello in cui si raggiungono le foglie, ovvero il livello in cui, stando alla definizione dell'equazione di ricorrenza, vale:

$$\frac{n}{5^h} \leq 1$$

Risolvendo per $h$ si ottiene:

$$\frac{n}{5^h} \leq 1 \iff n \leq 5^i \implies h = \log_5 n$$

Il costo totale dell'albero $T_1$ si otterrà allora svolgendo la somma dei contributi livello per livello:

$$\begin{aligned} T_1(n) &= \sum_{i=0}^{h} 4 \cdot 5^i = 4 \cdot \sum_{i=1}^{h} 5^i = 4 \cdot \frac{i - 5^{h+1}}{1 - 5} \\ &= \cancel{4}\frac{1 - 5^{h+1}}{\cancel{-4}} = -\left(1 - 5^{\log_5 n + 1}\right) \\ &= -\left(1 - 5 \cdot 5^{\log_5 n}\right) = -\left(1 - 5 \cdot n\right) \\ &= 5n - 1 = \Theta(n) \end{aligned}$$

Svolgendo calcoli analoghi per il secondo albero si ottiene che l'altezza dell'albero $T_2$ vale $h = \frac{1}{2}\log_5 n$. Quindi:

$$T_2(n) = \sum_{i=0}^{h} 4 \cdot 5^i = 5\sqrt{n} - 1 = \Theta(\sqrt{n})$$

Dato che le stime asintotiche dei due alberi non sono confrontabili possiamo concludere affermando di aver trovato un limite asintotico inferiore ed un limite asintotico superiore, ovvero: $\Omega(n) \leq T(n) \leq O(\sqrt{n})$. ■

Esercizio

Si risolva la seguente ricorrenza, calcolandone l'**andamento asintotico**:

$$T(n) = \begin{cases} 1 & \text{se } n \leq 4 \\ 8 \cdot T(\frac{n}{4}) + \sqrt{n} & \text{altrimenti} \end{cases}$$

Esercizio

Si risolva la seguente ricorrenza, calcolandone l'**andamento asintotico**:

$$T(n) = \begin{cases} 1 & \text{se } n \leq 27 \\ 3n^2 \cdot T(\sqrt[3]{n}) + 2n^3 & \text{altrimenti} \end{cases}$$

Esercizio

Si risolva la seguente ricorrenza, calcolandone l'**andamento asintotico**:

$$T(n) = \begin{cases} 1 & \text{se } n \leq 2 \\ 2 \cdot T(\sqrt[4]{n}) + \log(2n) & \text{altrimenti} \end{cases}$$

### 9.1.2 Notazione asintotica

Esercizio

Si dimostri, **esplicitando il procedimento seguito nella sua interezza**, la verità o la falsità della seguente affermazione:

$$\log_2(n^{2n}) + n - \log_2 n = \Theta(\log_2 n^n)$$

In caso affermativo, trovare le costanti che assicurano la validità della relazione.

**Svolgimento** Per dimostrare la relazione possiamo avvalerci dell'equivalenza descritta dall'Equazione 5.2. Posto $f(n) = \log_2(n^{2n}) + n - \log_2 n$ e $g(n) = \log_2 n^n$ si ha:

$$f(n) = \Theta(g(n)) \iff \lim_{n\to+\infty} \frac{f(n)}{g(n)} = l \in \mathbb{R}^+$$

Per dimostrare la veridicità della relazione è sufficiente quindi svolgere il limite di tale rapporto:

$$\begin{aligned}
\lim_{n\to+\infty} \frac{\log_2(n^{2n}) + n - \log_2 n}{\log_2 n^n} &= \lim_{n\to+\infty} \frac{2n \cdot \log_2(n) + n - \log_2 n}{n \cdot \log_2 n} && \text{Per le proprietà dei logaritmi} \\
&= \lim_{n\to+\infty} \frac{n \cdot \left(2 \cdot \log_2 n + 1 - \frac{\log_2 n}{n}\right)}{n \cdot \log_2 n} && \text{Mettendo } n \text{ in evidenza} \\
&= \lim_{n\to+\infty} \frac{\cancel{n} \cdot \left(2 \cdot \log_2 n + 1 - \frac{\log_2 n}{n}\right)}{\cancel{n} \cdot \log_2 n} && \text{Semplificando} \\
&= \lim_{n\to+\infty} \frac{2 \cdot \log_2 n + 1 - \frac{\log_2 n}{n}}{\log_2 n} \\
&= \lim_{n\to+\infty} \frac{\log_2 n\left(2 + \frac{1}{\log_2 n} - \frac{1}{n}\right)}{\log_2 n} && \text{Mettiamo in evidenza } \log_2 n \\
&= \lim_{n\to+\infty} \frac{\cancel{\log_2 n}\left(2 + \frac{1}{\log_2 n} - \frac{1}{n}\right)}{\cancel{\log_2 n}} && \text{Semplificando} \\
&= \lim_{n\to+\infty} 2 + \frac{1}{\log_2 n} - \frac{1}{n} = 2 > 0
\end{aligned}$$

Verificata la relazione, restano da trovare le costanti $n_0$, $c_1$ e $c_2$ che assicurano la relazione:

$$\exists n_0\Big(\exists c_1, c_2 > 0\big(\forall n > n_0(c_1 < \frac{f(n)}{g(n)} < c_2)\big)\Big)$$

Riscriviamo la disequazione sostituendo gli effettivi valori per $f(n)$ e $g(n)$:

$$c_1 \leq \frac{\log_2(n^{2n}) + n - \log_2 n}{\log_2 n^n} \leq c_2$$

che può essere riscritta come segue:

$$c_1 \cdot \log_2 n^n \leq \log_2(n^{2n}) + n - \log_2 n \leq c_2 \cdot \log_2 n^n$$

La ricerca dell'esistenza delle costanti $n_0$, $c_1$ e $c_2$ equivale alla risoluzione delle due disequazioni nelle incognite $c_1$ e $c_2$:

$$\begin{cases} c_1 \cdot \log_2 n^n \leq \log_2(n^{2n}) + n - \log_2 n \\ \log_2(n^{2n}) + n - \log_2 n \leq c_2 \cdot \log_2 n^n \end{cases}$$

L'obiettivo deve essere quello di ottenere una relazione indipendente dal valore di $n$.

**Suggerimento**

All'interno di queste tipologie di dimostrazioni viene in aiuto il concetto di **maggiorazione** e **minorazione** delle funzioni che fanno largo utilizzo della proprietà transitiva. Supponiamo di voler dimostrare che $f(x) < A$, **maggiorare** una funzione $f(x)$ consiste nel trovare una funzione $g(x)$ possibilmente più semplice di $f$, tale che $|f(x)| \leq g(x)$ e tale che valga pure $g(x) < A$. Stando a quanto detto varrà la seguente catena di disuguaglianze:

$$f(x) \leq g(x) < A \implies f(x) < A$$

che dimostra la relazione cercata. Dualmente per la minorazione di funzione.

**!** Se vogliamo maggiorare una frazione, dobbiamo maggiorare il numeratore e minorare il denominatore.

Iniziamo studiando la prima delle due disequazioni. Si osserva che nel membro a destra esiste un unico termine sottrattivo. Osservando i grafici mostrati in Figura 9.1 osserviamo che:

$$\forall n \geq 0 (\log n < n)$$

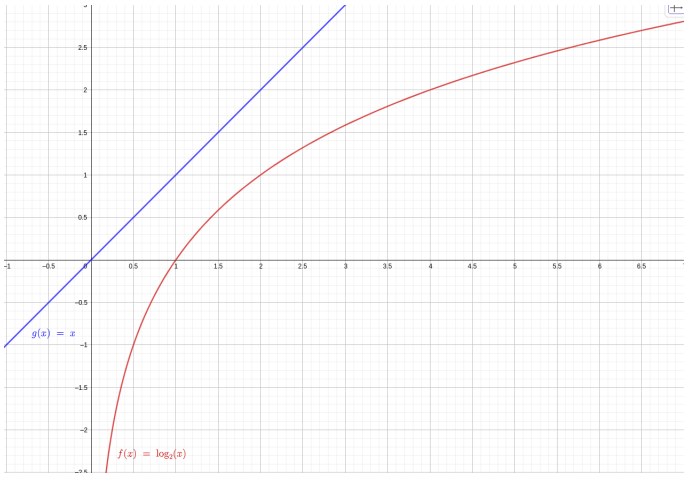



Figura 9.1

Pertanto è possibile pensare di minorare la relazione sostituendo $n$ al termine sottrattivo presente ottenendo così una funzione più piccola[1]:

$$c_1 n \log_2 n \leq 2n \log_2 + n - n = 2n \log_2 < 2n \log_2 n + n - \log_2 n \qquad \text{Minorando}$$

$$c_1 \leq \frac{2n \log_2 n}{n \log_2 n}$$

$$c_1 \leq \frac{2\cancel{n \log_2 n}}{\cancel{n \log_2 n}}$$

$$c_1 \leq 2$$

E otteniamo quindi $c_1 = 2$. Nel svolgere la seconda disequazione possiamo sfruttare il fatto che, per ogni $n \geq 4$ vale: $n + \log_2 n \leq n \log_2 n$. Allora è possibile eseguire la seguente maggiorazione:

$$2n \log_2 n + n - \log_2 n < 2n \log_2 n + n + \log_2 n \leq 2n \log_2 n + n \log_2 n = 3n \log_2 n \leq c_2 n \log_2 n$$

da cui si ottiene $c_2 \geq 3$ e quindi $c_3 = 3$.

[1] Sottraendo un qualcosa di più grande si ottiene l'effetto desiderato.

Esercizio

Si dimostri, **esplicitando il procedimento seguito nella sua interezza**, la verità o la falsità della seguente affermazione: se $f(n) = \Theta(\sqrt{g(n)})$ e $g(n) = \Theta((k(n))^4)$ allora:

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

In caso affermativo, trovare le costanti che assicurano la validità della relazione.