



NoiStudiamo

DISPENSA DI TEORIA



Programmazione II

Programmazione a oggetti in C++, template, complessità, ordinamento, liste, pile, code, alberi e grafi.

A CURA DI

Damiano La Pusata

Ottobre 2026

Indice dei capitoli

I nove capitoli della dispensa, dal passaggio dal C al C++ fino ai grafi.

1. Dal C al C++	3
2. Classi e oggetti	16
3. Ereditarietà e polimorfismo	26
4. Template e overloading degli operatori	35
5. La complessità computazionale	42
6. Ricorsione, ricerca e ordinamento	57
7. Liste, pile e code	67
8. Alberi e alberi binari di ricerca	80
9. Grafi	91

Dal C al C++

Il C++ nasce come estensione del C: tipi, operatori, strutture di controllo, array, puntatori e funzioni restano validi. Cambiano gli strumenti di contorno, più comodi e sicuri, e soprattutto il modo di pensare un programma, non più come sequenza di procedure ma come sistema di oggetti che collaborano.

Dalla programmazione procedurale agli oggetti

ASTRAZIONE

Il mondo è fatto di sistemi complessi in cui oggetti diversi interagiscono e cambiano comportamento in base a ciò che accade; costruire un software di grandi dimensioni è altrettanto complesso. Lo strumento per dominare la complessità è l'**astrazione**: un procedimento che **semplifica la realtà** concentrando l'attenzione solo sugli elementi importanti per lo scopo che ci interessa. Una piantina stradale è un'astrazione di una città: mostra strade e incroci e ignora il colore delle facciate. Per bere un caffè con la moka non serve conoscere la temperatura esatta dell'acqua o la finezza della macinatura: basta sapere che si riempie, si accende il fuoco e si aspetta. I dettagli ci sono, ma restano **nascosti** dietro un'interfaccia semplice.

PROCEDURALE CONTRO ORIENTATO AGLI OGGETTI

Nella **programmazione procedurale**, quella del linguaggio C, l'attenzione è rivolta alla **sequenza di operazioni**: un problema complesso si scompone in **procedure** (funzioni) che si eseguono una dopo l'altra fino alla soluzione. I dati e gli algoritmi restano **separati**: le `struct` da una parte, le funzioni che le manipolano dall'altra, spesso legate a variabili globali e header che bisogna portarsi dietro per riusare il codice. È difficile, così, modellare concetti di alto livello e costruire componenti davvero riutilizzabili.

Nella **programmazione orientata agli oggetti** (OOP, *Object Oriented Programming*) l'attenzione si sposta dalle procedure agli **oggetti**: entità che hanno un loro **stato** e sono capaci di eseguire certe **operazioni**. Un problema complesso diventa un **sistema** di oggetti che interagiscono. L'algoritmo non sparisce, ma perde il ruolo centrale a vantaggio del concetto di sistema. Si pensi all'interfaccia grafica di un PC: ci sono finestre (con dimensione e posizione) e bottoni (con colore e testo), e premendo un bottone si può aprire una finestra. Descrivere questo mondo come una lista di procedure sarebbe innaturale; descriverlo come oggetti che si scambiano richieste è immediato.

	PROGRAMMAZIONE PROCEDURALE	PROGRAMMAZIONE A OGGETTI
Punto di partenza	problema complesso	sistema complesso
Scomposizione	in procedure (funzioni)	in entità interagenti (oggetti)
Dati e algoritmi	separati (<code>struct</code> + funzioni)	uniti nella stessa classe
Domanda guida	«che cosa devo fare, in che ordine?»	«quali oggetti ci sono, cosa fanno?»

OGGETTI E CLASSI

Un **oggetto** si descrive elencando le sue **caratteristiche** (gli **attributi**, che ne definiscono lo **stato**) e i suoi **comportamenti** (i **metodi**, le funzionalità che mette a disposizione). Un'automobile ha come attributi velocità, colore, numero di porte, livello del carburante, marcia inserita; come metodi accelera, fermati, gira, cambia marcia, rifornisciti. Chi usa l'oggetto agisce **attivando i suoi comportamenti**, e questi si traducono in azioni o in un cambiamento di stato: se `auto_1` viaggia a 60 km/h, la chiamata `auto_1.fermati()` porta il suo attributo velocità a 0; la chiamata `auto_1.accelera(60)` lo riporta a 60.

DEFINIZIONE

Classe e oggetto

Una **classe** è il modello che descrive attributi (lo stato) e metodi (i comportamenti) comuni a un insieme di oggetti. Un **oggetto** è un'istanza della classe: ha la struttura definita dalla classe e un proprio stato. La relazione tra classe e oggetto è la stessa che lega un tipo a una variabile.

Una **classe** è il modello da cui si creano gli oggetti: incapsula gli attributi e i metodi comuni e fornisce un'**interfaccia pubblica** per interagire con essi. La relazione è la stessa che c'è tra **tipo e variabile**: `Automobile` è la classe, `auto_1` e `auto_2` sono due suoi oggetti (**istanze**), con la stessa struttura ma stati diversi (una rossa a 60 km/h, l'altra blu a 50). In altre parole, l'OOP combina strutture dati e algoritmi in entità software «impacchettate» dalla definizione di una classe. Un programma a oggetti è fatto di tanti oggetti che comunicano tramite **scambio di messaggi**: un oggetto, inviando un messaggio a un altro, gli chiede di eseguire un metodo.

I TRE PILASTRI DELLA OOP

Incapsulamento e occultamento. L'incapsulamento è la proprietà degli oggetti di **contenere insieme attributi e metodi**. L'occultamento (*information hiding*) consiste nel **nascondere all'esterno i dettagli implementativi**: dall'esterno si vede solo l'interfaccia (i metodi pubblici), non come sono realizzati né come sono memorizzati i dati. È il principio della moka: si usa senza smontarla. Il vantaggio pratico è enorme: si può cambiare l'implementazione interna senza toccare il codice che usa l'oggetto. Le classi permettono così di definire **tipi di dato astratti** (ADT).

DEFINIZIONE

Incapsulamento, ereditarietà, polimorfismo

L'**incapsulamento** raccoglie attributi e metodi nella stessa classe; l'**occultamento** nasconde all'esterno i dettagli implementativi. L'**ereditarietà** costruisce nuove classi estendendo quelle esistenti. Il **polimorfismo** permette a uno stesso metodo di assumere implementazioni diverse nelle sottoclassi, scelte durante l'esecuzione.

Ereditarietà. Spesso non serve partire da zero: se esiste già una classe simile a quella che vogliamo, la si **estende**. La classe generata (**sottoclasse** o classe derivata) riceve tutti gli attributi e i metodi della classe generatrice (**superclasse** o classe base) e si differenzia da essa **per estensione** (aggiunge attributi e metodi) o **per ridefinizione** (modifica i metodi ereditati). Le relazioni formano una **gerarchia**: Mezzi di trasporto → Mezzi a motore → Automobile, Moto, Autobus. Se una classe eredita da più classi base si parla di **ereditarietà multipla** (un Cavallo può essere sia Mezzo di trasporto sia Animale).

Polimorfismo. Le sottoclassi possono ridefinire un metodo ereditato mantenendone il nome: il metodo assume così **forme diverse** nella gerarchia. Tutti i veicoli a motore hanno il metodo `accelera`, ma l'automobile lo realizza con il pedale e la moto con la manopola. Durante l'esecuzione un oggetto di tipo «veicolo a motore» può essere in realtà un'automobile o una moto, e quando si chiama `accelera` deve partire la versione giusta. Lo strumento che lo garantisce è il **binding dinamico**: l'associazione tra la chiamata e il codice da eseguire avviene **a run time**, quando si conosce il tipo effettivo dell'oggetto.

VERIFICA

Il collegamento dinamico è lo strumento utilizzato per la realizzazione... a) della memoria dinamica; b) dell'ereditarietà; c) del polimorfismo; d) della composizione di classi.

Risposta **c)**. Il binding (collegamento) dinamico sceglie a tempo di esecuzione quale versione di un metodo eseguire in base al tipo effettivo dell'oggetto: è esattamente ciò che serve al polimorfismo. In C++ si ottiene dichiarando il metodo `virtual` (capitolo 3). L'ereditarietà è il prerequisito (senza gerarchia non ci sono versioni diverse tra cui scegliere), ma non è realizzata dal binding dinamico.

Quali sono i concetti fondamentali della OOP?

Astrazione (considerare solo gli elementi importanti per lo scopo), incapsulamento e occultamento (raccogliere attributi e metodi in una classe, esporre un'interfaccia e nascondere i dettagli), ereditarietà (costruire nuove classi estendendo quelle esistenti), polimorfismo (comportamenti diversi per lo stesso metodo nelle sottoclassi, realizzato con il binding dinamico).

DA RICORDARE

Le tre fasi dello sviluppo object oriented

Lo sviluppo object oriented si articola in tre fasi. Nell'**analisi** (OOA) si identificano i requisiti, i componenti del sistema e le loro relazioni logiche, senza preoccuparsi di come verranno implementati. Nel **design** (OOD) si specificano le gerarchie tra classi, le loro interfacce e i loro comportamenti, cercando moduli **altamente coesi** ma con **accoppiamento lasco** (*loose coupling*): un cambiamento in un componente non deve costringere a modificarne altri. Nella **programmazione** (OOP propriamente detta) si scrive il codice, lo si testa e lo si integra. Anche in un piccolo programma conviene partire chiedendosi «quali classi ci sono e chi deriva da chi», e solo dopo scrivere codice.

Il primo programma C++

DAL SORGENTE ALL'ESEGUIBILE

Il ciclo di vita di un programma è lo stesso del C. Si scrive il **codice sorgente** con un editor e lo si salva con estensione `.cpp`; il **compilatore** lo traduce in **codice oggetto** (`.o`); il **linker** collega il codice oggetto con quello delle librerie usate (I/O, funzioni matematiche...) e produce l'**eseguibile** (`.exe` su Windows, nessuna estensione su Linux). Se il compilatore segnala errori di sintassi si torna a correggere il sorgente; se il programma compila ma non fa ciò che ci si aspetta, l'errore è logico e si torna comunque al sorgente. In genere compilazione e collegamento si fanno con un solo comando.

```
$ g++ main.cpp -o main          # compila e collega: produce l'eseguibile "main"
$ ./main                        # lo esegue
$ g++ -Wall main.cpp -o main    # -Wall: mostra tutti gli avvisi (consigliato)
$ g++ main.cpp lista.cpp -o prog # piu' file sorgente in un solo eseguibile
```

DA RICORDARE

Compilare da riga di comando

Si può lavorare con un IDE (Visual Studio Code, Code::Blocks, Xcode...), ma è importante **saper compilare ed eseguire da riga di comando** con g++, che funziona allo stesso modo su qualunque macchina. Le tracce lo ripetono: «il codice deve essere fatto in modo tale da poter eseguire la compilazione su una macchina generica con il comando g++» e «non consegnare codice con errori di compilazione o runtime». L'opzione `-Wall` attiva tutti gli avvisi del compilatore: conviene usarla sempre e leggere gli avvisi, perché segnalano spesso errori logici reali.

STRUTTURA DEL PROGRAMMA

```
// ciao.cpp - il primo programma C++
#include <iostream>          // cin, cout, cerr (equivale a stdio.h del C)
using namespace std;        // evita di scrivere std::cout ogni volta

int main() {
    int anni;
    cout << "Quanti anni hai? ";          // << inserisce nel flusso di output
    cin >> anni;                          // >> estrae dal flusso di input
    cout << "Fra 10 anni ne avrai " << anni + 10 << endl; // endl = "\n" + svuota buffer
    return 0;
}
```

Rispetto al C cambiano tre cose. L'header è `<iostream>` (senza `.h`): contiene gli oggetti `cin`, `cout`, `cerr` per l'input/output. Le librerie del C restano disponibili con il prefisso `c` e senza `.h`: `<cmath>`, `<cstring>`, `<cstdlib>`, `<cctype>`. La seconda novità è il **namespace**. Tutti i nomi della libreria standard vivono nello spazio dei nomi `std`, quindi il loro nome completo è `std::cout`, `std::string`; la direttiva `using namespace std;` rende visibili quei nomi senza prefisso. Un namespace è un contenitore di nomi che evita conflitti: due librerie possono definire entrambe una funzione stampa senza scontrarsi, perché i nomi completi sono `libA::stampa` e `libB::stampa`. L'operatore `::` si chiama **operatore di risoluzione di visibilità** (*scope resolution*) e lo ritroveremo con le classi.

La terza novità è l'I/O a **flussi** (*stream*). Un flusso è un'astrazione che rappresenta dati che scorrono tra un produttore e un consumatore: `cout` è il flusso di uscita verso lo schermo, `cin` quello di ingresso dalla tastiera. L'operatore `<<` (inseritore) **inserisce** un valore nel flusso di uscita, l'operatore `>>` (estrattore) **estrae** un valore dal flusso d'ingresso e lo scrive nella variabile. Non servono specificatori di formato come `%d` o `%f`: il tipo della variabile è noto al compilatore, che sceglie da solo la conversione giusta. Gli operatori si possono concatenare: `cout << a << b << endl;`. Per includere un header scritto da noi si usano le virgolette: `#include "lista.h"` (le parentesi angolari sono riservate agli header di sistema).

IN C (PROGRAMMAZIONE I)	IN C++	NOTE
<code>#include <stdio.h></code>	<code>#include <iostream></code>	flussi <code>cin</code> , <code>cout</code> , <code>cerr</code>
<code>printf("%d\n", x);</code>	<code>cout << x << endl;</code>	nessun formato da indicare
<code>scanf("%d", &x);</code>	<code>cin >> x;</code>	niente <code>&</code>
<code>char s[50]; strcpy(s, "ciao");</code>	<code>string s = "ciao";</code>	classe <code>string</code>
<code>int* p = malloc(n * sizeof(int));</code>	<code>int* p = new int[n];</code>	<code>delete[] p</code> ; al posto di <code>free</code>
<code>void f(int* x) + f(&a)</code>	<code>void f(int& x) + f(a)</code>	passaggio per riferimento

IN C (PROGRAMMAZIONE I)	IN C++	NOTE
<code>#define N 10</code>	<code>const int N = 10;</code>	costante tipizzata
<code>NULL</code>	<code>nullptr</code>	puntatore nullo (C++11)
<code>FILE* f = fopen(...)</code>	<code>ifstream f(...);</code>	file come flussi

Input e output su console

COUT, CIN, CERR

I flussi standard sono tre: `cout` (standard output), `cin` (standard input) e `cerr` (standard error, per i messaggi d'errore; è lo `stderr` del C). Separare gli errori dall'output normale serve perché i due flussi si possono redirigere in modo diverso: se l'output va su file, gli errori continuano ad apparire a schermo. `endl` inserisce un a capo e **svuota il buffer** (*flush*), garantendo che il testo compaia subito; `"\n"` va a capo senza svuotare.

L'estrattore `>>` **salta gli spazi bianchi** iniziali (spazi, tabulazioni, a capo) e si ferma al primo spazio dopo il valore letto. Con `cin >> nome` e l'input «Mario Rossi» si legge solo «Mario»; il resto rimane nel buffer per la lettura successiva. Per leggere un'intera riga, spazi compresi, si usa `getline(cin, s)`, che legge fino all'a capo (escluso) e lo scarta. Attenzione a mescolare i due: dopo `cin >> n` nel buffer resta l'a capo, e una `getline` successiva leggerebbe una riga vuota; si risolve scartandolo con `cin.ignore()`. Esistono anche `cin.get(c)`, che legge un carattere alla volta compresi gli spazi, e `cin.getline(array, n)` per gli array di `char`.

CONTROLLO DEGLI ERRORI DI INTERPRETAZIONE

Se l'utente scrive «abc» dove ci si aspetta un intero, `cin >> n` **fallisce**: il flusso entra in uno stato di errore (*failbit*), `n` non riceve un valore valido e tutte le letture successive falliscono subito finché lo stato non viene ripristinato. L'espressione `cin >> n` vale `true` se la lettura è riuscita e `false` altrimenti, quindi si può usare direttamente come condizione. Per recuperare servono due passi: `cin.clear()` azzera lo stato d'errore e `cin.ignore(...)` scarta i caratteri sbagliati rimasti nel buffer. Le funzioni `cin.fail()`, `cin.eof()` e `cin.good()` permettono di interrogare lo stato del flusso.

```
// lettura di un intero con controllo dell'errore di interpretazione
#include <iostream>
#include <limits>           // numeric_limits
using namespace std;

int main() {
    int n;
    cout << "Inserisci un intero: ";
    while (!(cin >> n)) {           // false se l'input non e' un intero
        cerr << "Errore: non e' un numero intero, riprova: ";
        cin.clear();               // 1) azzera lo stato di errore (failbit)
        cin.ignore(numeric_limits<streamsize>::max(), '\n'); // 2) scarta la riga
    }
    cout << "Hai inserito " << n << endl;
    return 0;
}
```

ESERCIZIO

Esecuzione con input «abc» e poi «12»

La prima `cin >> n` trova «abc», non riesce a interpretarlo come intero e restituisce `false`: si entra nel `while`, si stampa l'errore su `cerr`, `clear()` ripristina il flusso e `ignore()` butta via «abc» fino all'a capo. Senza `ignore()` la lettura successiva ritroverebbe «abc» e il ciclo non finirebbe mai; senza `clear()` ogni lettura fallirebbe comunque. Al secondo giro si legge 12, la condizione è falsa e il programma stampa «Hai inserito 12».

Tipi, operatori e strutture di controllo

Tipi fondamentali, operatori e strutture di controllo sono quelli del C, con le stesse regole di precedenza, le stesse conversioni implicite (promozione verso il tipo «più preciso» nella graduatoria `long double`, `double`, `float`, `long`, `int`, `short`, `char`, `bool`) e lo stesso comportamento della divisione intera. Le differenze da ricordare sono poche. Il C++ ha il tipo `bool` nativo con le costanti `true` e `false` (convertibili in 1 e 0). Le **costanti** si dichiarano preferibilmente con `const`, che a differenza di `#define` ha un tipo controllato dal compilatore. Oltre al cast in stile C, `(int) x`, il C++ offre cast espliciti più leggibili: `static_cast<int>(x)` per le conversioni ordinarie, `const_cast` per togliere il `const`, `dynamic_cast` per le gerarchie di classi (capitolo 3) e `reinterpret_cast` per le conversioni a basso livello. La sintassi è lo stesso «stile funzione» dei template: il tipo di destinazione va tra parentesi angolari.

Le strutture di controllo (`if/else`, `switch`, `while`, `do/while`, `for`, `break`, `continue`) sono identiche; ricorda solo che l'`else` si lega sempre all'`if` più vicino e che in un `for` si può dichiarare la variabile di controllo nell'intestazione (`for (int i = 0; ...)`), con visibilità limitata al ciclo. Le regole di **visibilità** (*scope*) sono anch'esse quelle del C: una variabile locale è visibile dal punto in cui è definita fino alla fine del blocco `{ }` che la contiene; una variabile globale è visibile ovunque, tranne nei blocchi in cui una locale con lo stesso nome la **oscura**.

VERIFICA

Due domande «qual è l'output» sugli array

```
char str1[] = {'C','i','a','o'}; char str2[] = "Ciao"; cout << sizeof(str1) << " " << sizeof(str2);
```

Stampa **4 5**. `str1` è inizializzato carattere per carattere: quattro `char`, nessun terminatore. `str2` è inizializzato con una stringa letterale, che il compilatore completa con `'\0'`: cinque `char`. Solo `str2` è una stringa C valida; stampare `str1` con `cout` leggerebbe oltre la fine dell'array.

double mt[2][3]; quale espressione al posto di ???? stampa `mt[f][c]`? a) `mt(f+c)` b) `(mt+f)[c]` c) `mt[f] + c` d) `*(mt + f) + c`

Risposta **d**). Come in C, `mt + f` è il puntatore alla riga `f`, `/(mt + f)` è la riga stessa (cioè il puntatore al suo primo elemento), `/(mt + f) + c` punta all'elemento di colonna `c` e l'ultimo `//` ne prende il valore. La `c` è un indirizzo, non un valore; la `b`) applica `[c]` prima del `//` per precedenza, quindi equivale a `*((mt+f)[c])`, un'altra cosa.

Le stringhe

In C una stringa è un array di `char` terminato da `'\0'`, e ogni operazione passa per le funzioni di `<cstring>`: `strcpy` per assegnare, `strcmp` per confrontare, `strcat` per concatenare, `strlen` per la lunghezza. Le stringhe C esistono ancora in C++ (con gli stessi limiti: niente assegnamento con `=`, niente confronto con `==`, rischio di **buffer overflow** se si legge con `cin >>` più caratteri di quanti l'array ne contenga). Il C++ però offre la classe `string` (header `<string>`), un oggetto che gestisce da solo la memoria e cresce quando serve. È la scelta preferibile, salvo casi in cui siano richiesti esplicitamente gli array di `char`.

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string nome = "Mario";           // niente strcpy: si assegna con =
    string cognome("Rossi");
    string completo = nome + " " + cognome; // concatenazione con +
    cout << completo << " ha " << completo.length() << " caratteri" << endl;
    if (nome == "Mario")             // confronto con ==, <, > (niente strcmp)
        cout << "Ciao Mario" << endl;
    cout << completo[0] << endl;      // accesso al singolo carattere: M
    cout << completo.substr(6, 5) << endl; // da posizione 6, lunga 5: Rossi
    cout << completo.find("Ros") << endl; // posizione della prima occorrenza: 6
    string riga;
    getline(cin, riga);              // legge un'intera riga, spazi compresi
    cout << "[" << riga << "]" << endl;
    return 0;
}
```

OPERAZIONE	STRINGA C (CHAR[])	CLASSE STRING
assegnamento	strcpy(s, "ciao")	s = "ciao"
concatenazione	strcat(s, t)	s + t, s += t
confronto	strcmp(s, t) == 0	s == t, s < t
lunghezza	strlen(s)	s.length() o s.size()
carattere in posizione i	s[i]	s[i]
sottostringa / ricerca	strncpy, strstr	s.substr(pos, len), s.find(t)
lettura riga intera	cin.getline(s, n)	getline(cin, s)
conversione in numero	atoi(s), atof(s)	stoi(s), stof(s), stod(s)

Il metodo `find` restituisce la posizione della prima occorrenza, oppure la costante `string::npos` se non trova nulla. Insieme a `substr` e `stoi/stof` permette di spezzare una riga con separatori, come «1004,Mario,Rossi,25.5», una tecnica usata anche nella dispensa di laboratorio per leggere file di record. In alternativa si può usare `getline(flusso, campo, ' ', ')`, che legge fino al separatore indicato invece che fino all'a capo.

Le funzioni in C++

RIFERIMENTI E PASSAGGIO PER RIFERIMENTO

In C esiste solo il passaggio **per valore**: la funzione riceve una **copia** dell'argomento, e per modificare una variabile del chiamante bisogna passarne l'indirizzo e dereferenziarlo, come nel classico `swap` con i puntatori. Il C++ introduce il **riferimento** (*reference*): una variabile di tipo `int&` è **un altro nome per una variabile `int` già esistente**. Scrivendo `int& r = n;`, `r` e `n` indicano la stessa cella di memoria: modificare `r` modifica `n`, e `&r` coincide con `&n`. Un riferimento **deve essere inizializzato** quando lo si dichiara e **non può essere spostato** su un'altra variabile.

DEFINIZIONE

Riferimento

Un **riferimento** (`int& r = n;`) è un nome alternativo per una variabile già esistente. Deve essere inizializzato quando lo si dichiara e non può essere spostato su un'altra variabile. Un parametro di tipo riferimento permette alla funzione di lavorare direttamente sulla variabile del chiamante.

Il vantaggio arriva con i parametri. Se il parametro formale è un riferimento, `void scambiaR(int& a, int& b)`, la funzione lavora direttamente sulle variabili del chiamante: dentro si usano `a` e `b` come variabili normali, senza `*`, e la chiamata si scrive `scambiaR(x, y)`, senza `&`. È il **passaggio per riferimento**. Dietro le quinte un riferimento è un **puntatore costante** che il compilatore dereferenzia automaticamente: non aggiunge potenza rispetto ai puntatori, ma rende il codice più leggibile e meno soggetto a errori.

```
#include <iostream>
using namespace std;

void scambiaP(int* a, int* b) { // stile C: passo gli indirizzi
    int t = *a; *a = *b; *b = t; // e dereferenzio con *
}
void scambiaR(int& a, int& b) { // stile C++: a e b sono ALTRI NOMI di x e y
    int t = a; a = b; b = t; // niente *: si usano come variabili normali
}

int main() {
    int x = 1, y = 2;
    scambiaP(&x, &y); // passo gli indirizzi
    cout << x << " " << y << endl; // 2 1
    scambiaR(x, y); // passo le variabili: il riferimento e' implicito
    cout << x << " " << y << endl; // 1 2

    int n = 75;
    int& r = n; // r e' un alias di n (va inizializzato subito!)
    r = 80; // modifica n
    cout << n << " " << (&r == &n) << endl; // 80 1 (stesso indirizzo)
    return 0;
}
```

Il passaggio per riferimento si usa anche quando la funzione **non** deve modificare l'argomento ma l'argomento è **grande** (una string, un oggetto): copiarlo costerebbe tempo e memoria. In questo caso si passa un **riferimento costante**, `void stampa(const string& s)`: niente copia, e il `const` garantisce che la funzione non possa modificarlo. È la forma più comune per passare oggetti alle funzioni, per esempio una procedura di inserimento che riceve un riferimento costante a uno studente. Una funzione può anche **restituire** un riferimento, `int& max(int& a, int& b)`: in quel caso la chiamata è a sua volta un nome di variabile e si può usare a sinistra di un assegnamento, `max(x, y) = 0`; azzera la maggiore tra `x` e `y`. Non si deve mai restituire un riferimento (o un puntatore) a una variabile **locale**: al ritorno la variabile non esiste più.

VERIFICA

Qual è l'output? (passaggio per riferimento)

```
int boo(int &n) {
    int a = 0;
    for (; n > 0; n--)
        a += n;
    return a;
}
int main() { int n = 3; int m = boo(n); cout << n*m; }
```

Stampa **0**. Il ciclo somma $3 + 2 + 1$, quindi boo restituisce 6; ma n è passato **per riferimento**, quindi il `n--` del ciclo decrementa proprio la variabile n del main, che all'uscita vale 0. Il prodotto è $0 \cdot 6 = 0$. Se il parametro fosse `int n` (per valore) l'output sarebbe 18. Una sola & cambia tutto: conviene leggere sempre per prima cosa la firma della funzione.

PARAMETRI DI DEFAULT E OVERLOADING

Un parametro può avere un **valore di default**, scritto nell'intestazione: se nella chiamata manca il corrispondente argomento, si usa il default. I parametri con default devono essere **raggruppati a destra**, e nella chiamata si possono omettere solo gli ultimi: se si omette un argomento si devono omettere anche tutti quelli alla sua destra. Se una funzione ha un prototipo separato, il default va scritto **una sola volta**, di solito nel prototipo.

L'**overloading** (sovraccaricamento) delle funzioni permette di dare **lo stesso nome** a funzioni diverse, purché differiscano nel **numero** o nel **tipo** dei parametri. Il compilatore sceglie quale chiamare guardando gli argomenti: prima cerca una corrispondenza esatta, poi una che si ottiene con conversioni automatiche. Il solo tipo di ritorno **non basta** a distinguere due funzioni: `int f(int)` e `double f(int)` sono un errore.

```
#include <iostream>
#include <string>
using namespace std;

// parametri di default: vanno raggruppati a DESTRA
void saluta(string nome, string saluto = "Ciao", int volte = 1) {
    for (int i = 0; i < volte; i++)
        cout << saluto << " " << nome << endl;
}

// overloading: stesso nome, parametri diversi (numero o tipo)
int area(int lato) { return lato * lato; } // quadrato
int area(int base, int altezza) { return base * altezza; } // rettangolo
double area(double raggio) { return 3.14159 * raggio * raggio; } // cerchio

int main() {
    saluta("Anna"); // Ciao Anna
    saluta("Anna", "Buongiorno"); // Buongiorno Anna
    saluta("Anna", "Hey", 2); // Hey Anna (due volte)
    cout << area(3) << " " << area(3, 4) << " " << area(1.0) << endl; // 9 12 3.14159
    return 0;
}
```

FUNZIONI INLINE E VARIABILI STATIC

Una funzione dichiarata `inline` chiede al compilatore di **ricopiarne il corpo** in ogni punto in cui viene chiamata, invece di eseguire una vera chiamata (salto, passaggio dei parametri sullo stack, ritorno). Il programma diventa più veloce, ma più grande, perché il codice è duplicato: conviene per funzioni **brevi** chiamate molte volte. A differenza di una macro `#define`, una funzione `inline` controlla i tipi e valuta gli argomenti una sola volta. Per il compilatore moderno `inline` è un suggerimento, che può ignorare; i metodi definiti dentro una classe sono automaticamente `inline` (capitolo 2).

Una variabile locale dichiarata `static` è allocata nella zona dei dati statici, non sullo stack: viene inizializzata **una sola volta** e **conserva il valore tra una chiamata e l'altra** della funzione. Restano valide anche le altre classi di memorizzazione del C: `extern` dichiara una variabile globale definita in un altro file sorgente, `register` (ormai obsoleto) suggerisce di tenerla in un registro.

DA RICORDARE

Funzioni inline

Quale affermazione è vera per una funzione inline? a) viene eseguita più rapidamente poiché trattata come una macro internamente; b) viene eseguita più rapidamente perché le viene attribuita una priorità più alta; c) non viene eseguita più velocemente delle altre funzioni; d) nessuna delle precedenti.

La risposta è la **a)**: il compilatore sostituisce la chiamata con il corpo della funzione, «come una macro», ed evita il costo della chiamata. La **b)** è sicuramente falsa (non esiste una priorità di esecuzione). Per essere precisi, non è *letteralmente* una macro (i tipi sono controllati) e che `inline` è solo un suggerimento al compilatore.

Puntatori, riferimenti e const

I puntatori funzionano esattamente come in C: `&x` dà l'indirizzo di `x`, `//p` dà la variabile puntata, l'aritmetica dei puntatori avanza di multipli della dimensione del tipo, il nome di un array è un puntatore costante al suo primo elemento (`v[i]` equivale a `*(v + i)`), esistono i puntatori a puntatori (`int** pp`). Il puntatore nullo in C++ moderno si scrive `nullptr`; `NULL` e `0` funzionano ancora e si incontrano spesso nel codice esistente. Una funzione può anche **restituire** un puntatore, con lo stesso divieto visto per i riferimenti: mai l'indirizzo di una variabile locale.

Il qualificatore `const` applicato ai puntatori genera due casi da non confondere. Il trucco per leggerli è guardare **cosa c'è subito a sinistra dell'asterisco**, o leggere la dichiarazione da destra verso sinistra. In `const int* q` (si legge «`q` è un puntatore a un `int` costante») è costante il **valore puntato**: non si può scrivere `//q = 4`, ma si può spostare `q` su un'altra variabile. In `int// const p` («`p` è un puntatore costante a `int`») è costante il **puntatore**: `p` punta per sempre alla stessa variabile, ma attraverso di esso la si può modificare. Con `const int* const r` sono costanti entrambi.

DICHIARAZIONE	SI LEGGE	<code>*P = 4;</code>	<code>P = &Y;</code>
<code>int* p = &x;</code>	puntatore a <code>int</code>	sì	sì
<code>const int* p = &x;</code>	puntatore a <code>int</code> costante	no	sì
<code>int* const p = &x;</code>	puntatore costante a <code>int</code>	sì	no
<code>const int* const p = &x;</code>	puntatore costante a <code>int</code> costante	no	no

DA RICORDARE

Qual è l'istruzione errata?

```
int x, y;  
int *const p = &x;  
const int *q = &y;  
q = &x;  
*q = 4;
```

L'istruzione errata è `*q = 4;`. `q` è un puntatore a intero costante: può cambiare bersaglio (quindi `q = &x` è lecito, anche se `x` in sé non è `const`: si promette solo di non modificarla *attraverso* `q`), ma non può modificare il valore puntato. Il compilatore dà errore «assignment of read only location». Se invece ci fosse stato `p = &y`, l'errore sarebbe stato quello, perché `p` è un puntatore costante.

	PUNTATORE <code>INT* P</code>	RIFERIMENTO <code>INT& R</code>
Inizializzazione	facoltativa (può essere <code>nullptr</code>)	obbligatoria, su una variabile esistente
Cambiare bersaglio	sì (<code>p = &y</code>)	no, resta legato alla stessa variabile
Accesso al valore	<code>*p</code>	<code>r</code> (dereferenziazione implicita)
Valore nullo	possibile	non esiste un riferimento nullo
Uso tipico	strutture dinamiche (liste, alberi), array	parametri e valori di ritorno

Anche le funzioni stanno in memoria, e si può avere un **puntatore a funzione**: `int (*pf)(int);` dichiara un puntatore a una funzione che prende un `int` e restituisce un `int`, e `pf = f;` lo fa puntare alla funzione `f` (il nome di una funzione è il suo indirizzo). Si chiama poi con `pf(3)`.

La memoria dinamica: new e delete

Il sistema operativo assegna al programma tre zone di memoria: il **segmento del codice** (le istruzioni), il **segmento dei dati** (costanti e variabili globali e statiche) e lo **stack**, che cresce e decresce durante l'esecuzione con le variabili locali e i parametri di ogni chiamata. La memoria libera rimanente è lo **heap**, da cui si prende memoria **a richiesta** quando la quantità necessaria si conosce solo durante l'esecuzione. È la stessa mappa che in C si usa con `malloc`.

In C++ la memoria dinamica si gestisce con due **operatori**. `new tipo` alloca nello heap una variabile del tipo indicato e ne restituisce l'indirizzo; `new tipo[n]` alloca un array di `n` elementi, con `n` anche variabile. A differenza di `malloc` non serve calcolare i byte con `sizeof` né convertire il `void*` restituito, e con gli oggetti `new` chiama automaticamente il **costruttore** (capitolo 2). La memoria si libera con `delete p` per una singola variabile e con `delete[] p` per un array. `delete` rende riutilizzabile la memoria puntata ma non cancella il puntatore, che resta con il vecchio indirizzo (**puntatore pendente**, *dangling pointer*): conviene metterlo a `nullptr` se lo si riusa.

```

#include <iostream>
using namespace std;

int main() {
    int* p = new int(5);           // una variabile int nello heap, inizializzata a 5
    cout << *p << endl;
    delete p;                     // libera la singola variabile

    int n;
    cin >> n;
    int* v = new int[n];          // array dinamico: n deciso a run-time
    for (int i = 0; i < n; i++) v[i] = i * i;
    for (int i = 0; i < n; i++) cout << v[i] << " ";
    cout << endl;
    delete[] v;                   // array -> delete[] (con le parentesi!)

    // matrice dinamica righe x colonne: array di puntatori a righe
    int righe = 2, colonne = 3;
    int** m = new int*[righe];
    for (int i = 0; i < righe; i++)
        m[i] = new int[colonne];
    m[1][2] = 7;
    cout << m[1][2] << endl;
    for (int i = 0; i < righe; i++)
        delete[] m[i];            // prima le righe...
    delete[] m;                   // ...poi l'array di puntatori
    return 0;
}

```

DA RICORDARE

Errori classici con new e delete

Mescolare le forme: ciò che è allocato con `new[]` va liberato con `delete[]`, ciò che è allocato con `new` con `delete`. Mescolarli è un comportamento indefinito. **Memory leak:** perdere l'ultimo puntatore a un blocco senza averlo liberato (per esempio riassegnando `p = new int` due volte); la memoria resta occupata fino alla fine del programma. **Doppio delete:** liberare due volte lo stesso blocco fa quasi sempre crashare il programma. **Usare la memoria dopo il delete:** il puntatore è pendente e il contenuto non è più garantito. Nelle classi la memoria si libera nel **distruttore**.

Scegliere l'uso corretto di delete per l'espressione `ptr = new int[100]`: a) `delete ptr` b) `delete ptr[]` c) `delete[] ptr` d) `[] delete ptr`

Risposta **c)** `delete[] ptr`: le parentesi quadre vanno tra `delete` e il nome del puntatore.

I file con fstream

Per il C++ un file è semplicemente un **flusso esterno**: una sequenza di byte in memoria di massa, che diventa un flusso d'ingresso se lo si apre in lettura e d'uscita se lo si apre in scrittura. L'header `<fstream>` fornisce tre classi: `ifstream` per leggere, `ofstream` per scrivere, `fstream` per entrambe le cose. Si crea un oggetto passando il nome del file al costruttore (`ifstream in("input.txt");`) oppure lo si apre in un secondo momento con `in.open("input.txt")`. Una volta aperto, il file si usa **esattamente come** `cin` e `cout`: `in >> x` legge, `out << x` scrive, `getline(in, riga)` legge una riga. Alla fine si chiude con `close()`, che comunque viene chiamato in automatico quando l'oggetto esce di visibilità.

Un `ofstream` si apre per default in modalità `ios::out`, che **sovrascrive** il file; con `ios::app` (*append*) si scrive in coda al contenuto esistente: `ofstream out("log.txt", ios::app);`. Altri modi sono `ios::in` (lettura) e

`ios::binary` (file binari, letti e scritti con `read` e `write` invece che con `>>` e `<<`). Si può anche spostarsi in un punto del file con `seekg` e conoscere la posizione corrente con `tellg`, gli equivalenti di `fseek` e `ftell`.

```
// legge "matricola nome cognome media" da input.txt e scrive su output.txt
#include <iostream>
#include <fstream>          // ifstream (lettura), ofstream (scrittura)
#include <string>
using namespace std;

int main() {
    ifstream in("input.txt");          // apre il file in lettura
    if (!in) {                          // apertura fallita (file inesistente...)
        cerr << "Errore: impossibile aprire input.txt" << endl;
        return 1;
    }
    ofstream out("output.txt");         // apre (crea o sovrascrive) in scrittura

    int matricola;
    string nome, cognome;
    double media;
    // il ciclo continua finché la lettura dei 4 campi riesce
    while (in >> matricola >> nome >> cognome >> media) {
        out << matricola << ": " << nome << " " << cognome
            << " - media " << media << endl;    // stessa sintassi di cout
    }
    in.close();
    out.close();
    return 0;
}
```

Il modo robusto di leggere un file di record è mettere **la lettura stessa nella condizione del ciclo**: `while (in >> a >> b >> c)` continua finché tutti i campi vengono letti correttamente, e si ferma da solo alla fine del file o al primo dato malformato. Il controllo dell'apertura, `if (!in)`, è indispensabile: se il file non esiste il programma deve dirlo su `cerr` e terminare, invece di proseguire con dati inesistenti.

DA RICORDARE

Perché non usare `while (!in.eof())`

Molti programmi scrivono `while (!file.eof()) { file >> ...; }`. Il problema è che `eof()` diventa vero solo **dopo** che una lettura ha tentato di andare oltre la fine. Se il file termina con un a capo, l'ultima lettura «riuscita» lascia `eof()` falso, il ciclo fa un giro in più, la lettura fallisce e le variabili mantengono i valori del record precedente: l'ultimo studente viene inserito **due volte**. Con `while (in >> ...)` il problema non si pone.

Classi e oggetti

Le idee della programmazione a oggetti diventano codice: definizione di una classe, specificatori di accesso, metodi, costruttori e distruttore, che governano la nascita e la morte di ogni oggetto, fino al puntatore `this`, ai membri statici, alle funzioni friend e all'organizzazione del codice in più file.

Definire una classe

DALLA STRUCT ALLA CLASS

In C una `struct` raggruppa dati eterogenei, e le funzioni che li manipolano stanno fuori, ricevendo la struttura come parametro. Una **classe** mette insieme, nella stessa definizione, i **dati** (gli **attributi**, o **dati membro**) e le **funzioni** che operano su di essi (i **metodi**, o **funzioni membro**). La sintassi è quella di una `struct` con la parola chiave `class`, e come per una `struct` la definizione termina con un **punto e virgola** dopo la graffa chiusa: dimenticarlo è un errore di compilazione frequentissimo.

```
class NomeClasse {
public:           // membri accessibili da chiunque: l'interfaccia
    ...metodi...
protected:     // accessibili dalla classe e dalle classi derivate (capitolo 3)
    ...
private:        // accessibili solo dai metodi della classe stessa
    ...attributi...
};               // <- il punto e virgola e' obbligatorio
```

SPECIFICATORI DI ACCESSO

Gli **specificatori di accesso** realizzano l'occultamento dell'informazione. I membri `public` sono visibili da qualunque punto del programma e costituiscono l'**interfaccia** della classe. I membri `private` sono accessibili **solo dai metodi della classe stessa** (e dalle funzioni dichiarate `friend`, questo capitolo): un tentativo di accesso da fuori, come `a.num = 7` nel `main`, è un errore di compilazione. I membri `protected` si comportano come privati verso l'esterno, ma sono accessibili anche dalle **classi derivate**. Le sezioni possono comparire in qualunque ordine e anche più volte; la buona prassi è mettere prima la sezione pubblica (ciò che interessa a chi usa la classe), metodi pubblici e attributi privati.

In una `class` l'**accesso di default** è `private`: i membri scritti prima di qualunque specificatore sono privati. In una `struct` il **default** è `public`. È l'unica differenza tra le due parole chiave in C++: una `struct` può avere metodi, costruttori e tutto il resto. Per convenzione si usa `struct` per semplici aggregati di dati (come il nodo di una lista) e `class` per tutto il resto.

VERIFICA

Domande sui membri e sull'accesso

L'accesso di default a un membro di una classe è...

private. In `class abc { void f(); void g(); int n; };` tutti e tre i membri sono quindi privati.

L'operatore usato per accedere ai membri di un ADT usando un oggetto è: a) `.` b) `->` c) `*` d) nessuna delle precedenti

Risposta **a)**, il punto. La freccia `->` si usa quando si ha un **puntatore** all'oggetto: `p->stampa()` equivale a `(*p).stampa()`.

OGGETTI E CHIAMATA DEI METODI

Definita la classe, si creano gli **oggetti** (istanze) come si dichiarano le variabili: `Razionale r;`. Un oggetto sta alla sua classe come una variabile al suo tipo. Ai membri si accede con l'**operatore punto**: `r.stampa()` chiama il metodo `stampa` **su quell'oggetto**, che lavora sugli attributi di `r` e non su quelli di altri oggetti della stessa classe. Gli oggetti si possono anche creare dinamicamente con `new`, ottenendo un puntatore: `Razionale* p = new Razionale(1, 2);` e poi `p->stampa(); delete p;`. Si possono creare array di oggetti, `Razionale v[10];` (richiede un costruttore senza argomenti), o array di puntatori a oggetti, molto usati con l'ereditarietà.

METODI DEFINITI DENTRO E FUORI LA CLASSE

Un metodo si può **definire** direttamente nella classe, scrivendone il corpo, oppure solo **dichiarare** (con il prototipo) e definire altrove. Nella definizione esterna il nome va qualificato con la classe e l'operatore di risoluzione di visibilità: `void Razionale::inverti() { ... }`. Senza `Razionale::` il compilatore penserebbe a una normale funzione globale, che non può vedere `num` e `den`. I metodi definiti dentro la classe sono automaticamente **inline**, quindi conviene farlo solo per metodi brevi, come i `get` e i `set`; per quelli lunghi si scrive il prototipo nella classe e la definizione fuori.

```
#include <iostream>
using namespace std;

class Razionale {
public:
    // interfaccia: visibile dall'esterno
    Razionale(int n = 0, int d = 1) // costruttore con parametri di default
        : num(n), den(d) {}        // lista di inizializzazione dei membri
    // metodi get: leggono lo stato (const = non modificano l'oggetto)
    int getNum() const { return num; }
    int getDen() const { return den; }
    // metodo set: modifica lo stato controllando che il valore sia valido
    void setDen(int d) { if (d != 0) den = d; }
    // predicato: restituisce un bool su una proprietà dell'oggetto
    bool isIntero() const { return num % den == 0; }
    double converti() const { return (double)num / den; } // definito nella classe
    void inverti(); // solo dichiarato: definito fuori
    void stampa() const;
private:
    // dettagli nascosti
    int num, den;
};
// definizione "fuori linea": NomeClasse:: davanti al nome del metodo
void Razionale::inverti() {
    int t = num; num = den; den = t;
}
void Razionale::stampa() const {
    cout << num << "/" << den << endl;
}
```

```

int main() {
    Razionale a;           // 0/1 (tutti i default)
    Razionale b(5, 3);     // 5/3
    Razionale c(4);        // 4/1 (d prende il default)
    b.inverti();           // operatore . per accedere ai membri
    b.stampa();            // 3/5
    cout << c.isIntero() << " " << b.converti() << endl; // 1 0.6
    // a.num = 7;          // ERRORE: num e' private
    Razionale* p = new Razionale(1, 2); // oggetto dinamico
    p->stampa();            // con un puntatore si usa -> (equivale a (*p).stampa())
    delete p;
    a.stampa();
    return 0;
}

```

METODI GET, SET E PREDICATI

Poiché gli attributi sono privati, l'accesso dall'esterno passa per metodi dedicati: costruttore, metodi get e set, metodo di stampa. Un metodo **get** (*getter*, o di accesso) restituisce il valore di un attributo, per esempio `getNum()`. Un metodo **set** (*setter*, o modificatore) ne assegna uno nuovo, e ha il vantaggio di poter **controllare** che il valore sia valido prima di scriverlo: `setDen` rifiuta lo zero, cosa impossibile se l'attributo fosse pubblico. Un **predicato** è un metodo che restituisce un `bool` su una proprietà dell'oggetto, per esempio `isIntero()`, `isEmpty()`, `isScaduto()`.

I metodi che non modificano l'oggetto andrebbero dichiarati `const`, scrivendo `const` dopo la lista dei parametri: `int getNum() const`. Il compilatore impedisce allora al metodo di modificare gli attributi, e soprattutto permette di chiamarlo su oggetti costanti o ricevuti per riferimento costante. Se una funzione riceve `const Studente& s` e chiama `s.getMedia()`, il getter **deve** essere `const`, altrimenti la compilazione fallisce: è uno degli errori più comuni quando si usa il passaggio per riferimento costante richiesto dalle tracce.

Costruttori e distruttore

IL COSTRUTTORE

È bene che un oggetto nasca già in uno stato valido, senza dover chiamare un metodo di inizializzazione subito dopo averlo creato. Il **costruttore** è un metodo speciale eseguito **automaticamente alla creazione** di ogni oggetto. Ha lo **stesso nome della classe**, **non ha tipo di ritorno** (nemmeno `void`) e può avere un numero qualsiasi di parametri. Si può **sovraccaricare**: una classe può avere più costruttori con parametri diversi, e il compilatore sceglie quello che corrisponde agli argomenti forniti alla creazione.

DEFINIZIONE

Costruttore e distruttore

Il **costruttore** è il metodo con lo stesso nome della classe, senza tipo di ritorno, eseguito automaticamente alla creazione di ogni oggetto per portarlo in uno stato valido. Il **distruttore** (`~NomeClasse()`) è eseguito automaticamente quando l'oggetto viene distrutto e serve a liberare le risorse acquisite.

Il **costruttore di default** è quello chiamabile **senza argomenti**: serve per scrivere `Razionale a;` o per creare un array di oggetti. Se una classe **non dichiara nessun costruttore**, il compilatore ne genera uno di default implicito, che però non inizializza gli attributi di tipo base (restano con valori casuali). Se invece si dichiara **anche un solo costruttore con parametri**, quello di default implicito **non viene più generato**, e `Razionale a;` diventa un errore. Il modo compatto per avere sia il costruttore di default sia quelli con parametri è usare i **parametri di default**: `Razionale(int n = 0, int d = 1)` copre `Razionale a;`, `Razionale c(4);` e `Razionale b(5, 3);`. Se manca una combinazione, la creazione non compila: con i soli costruttori `Razionale()` e `Razionale(int, int)` la scrittura `Razionale fattore(4);` dà errore, perché non esiste un costruttore con un solo parametro.

LA LISTA DI INIZIALIZZAZIONE

Gli attributi si possono assegnare nel corpo del costruttore (`num = n; den = d;`), ma il modo preferito è la **lista di inizializzazione dei membri**: dopo la lista dei parametri si scrivono i **due punti** seguiti dagli inizializzatori separati da virgole, `: num(n), den(d)`. La differenza non è solo di stile. Nel corpo del costruttore gli attributi **esistono già** e ricevono un assegnamento; nella lista vengono **costruiti direttamente** con quel valore. Per questo la lista è **obbligatoria** in tre casi: per gli **attributi const** e per gli **attributi riferimento**, che non si possono assegnare dopo la creazione; per chiamare il **costruttore della classe base** in una classe derivata (capitolo 3); per gli attributi che sono oggetti di classi **senza costruttore di default**.

```
class C {
public:
    C(int param) : t(param), cInt(25), rInt(t) { } // unico modo per const e riferimenti
private:
    int t;
    const int cInt;
    int& rInt;
};
```

DA RICORDARE

Inizializzare un attributo dove lo si dichiara

In molti testi si legge che «non è permesso inizializzare un membro dato all'atto della sua definizione» (`class C { int t = 0; };`). Era vero fino al C++03; **dal C++11 è consentito** e il g++ attuale lo compila senza problemi: il valore indicato viene usato da ogni costruttore che non inizializzi diversamente quell'attributo. Nel C++ classico l'inizializzazione si fa nel costruttore, ed è la forma più portabile.

IL DISTRUTTORE

Il **distruttore** è il metodo speciale eseguito **automaticamente quando un oggetto viene distrutto**. Ha il nome della classe preceduto dalla **tilde**, `~Razionale()`, **non ha tipo di ritorno, non accetta parametri** e quindi **non si può sovraccaricare**: ce n'è al più uno. Se non lo si scrive, il compilatore ne genera uno vuoto. Serve soprattutto a **liberare le risorse acquisite dall'oggetto**, tipicamente la memoria allocata con `new` nel costruttore o nei metodi: in una lista concatenata, il distruttore scorre i nodi e li cancella uno per uno.

Quando muore un oggetto? Un oggetto **locale** viene distrutto quando esce di visibilità, cioè alla graffa che chiude il blocco in cui è stato dichiarato; più oggetti locali dello stesso blocco vengono distrutti **in ordine inverso** rispetto alla creazione (come una pila: l'ultimo creato è il primo distrutto). Un oggetto **globale** o `static` viene creato prima del `main` e distrutto dopo la sua fine. Un oggetto **dinamico** creato con `new` vive finché non si chiama `delete` sul suo puntatore: se non lo si fa mai, il distruttore non viene mai eseguito.

```

#include <iostream>
#include <string>
using namespace std;

class Traccia {
public:
    Traccia(string n) : nome(n) { cout << "costruisco " << nome << endl; }
    ~Traccia() { cout << "distruggo " << nome << endl; }
private:
    string nome;
};

Traccia g("globale"); // creato prima del main

int main() {
    cout << "inizio main" << endl;
    Traccia a("a");
    { // blocco interno
        Traccia b("b");
    } // b esce di visibilit : distrutto qui
    Traccia* p = new Traccia("dinamico");
    Traccia c("c");
    delete p; // un oggetto dinamico muore solo con delete
    cout << "fine main" << endl;
    return 0; // distrutti c, poi a (ordine inverso)
} // g viene distrutto dopo la fine del main

```

ESERCIZIO

Output di p2_ciclovita.cpp, riga per riga

«costruisco globale» compare **prima** di «inizio main». Poi si costruisce a, quindi b dentro il blocco interno: alla graffa chiusa del blocco b muore, e si legge «distruggo b». L'oggetto dinamico nasce con new, poi nasce c; il delete p distrugge subito l'oggetto dinamico, anche se   stato creato prima di c. Dopo «fine main», al return muoiono le variabili locali in ordine inverso, prima c e poi a; ultima, a programma ormai finito, muore la globale g. Le righe sono quindi: costruisco globale, inizio main, costruisco a, costruisco b, distruggo b, costruisco dinamico, costruisco c, distruggo dinamico, fine main, distruggo c, distruggo a, distruggo globale.

IL COSTRUTTORE DI COPIA

Il **costruttore di copia** crea un oggetto come **copia di un altro oggetto della stessa classe**. Ha la firma `NomeClasse(const NomeClasse& altro)`: il parametro   un **riferimento costante** (se fosse passato per valore, per copiarlo servirebbe... il costruttore di copia stesso, all'infinito). Viene chiamato automaticamente in tre situazioni: quando si **definisce un oggetto inizializzandolo con un altro** (`Razionale b = a;` oppure `Razionale b(a);`), quando si **passa un oggetto per valore** a una funzione (il parametro   una copia) e quando una funzione **restituisce un oggetto per valore**.

Se non lo si scrive, il compilatore genera un costruttore di copia che copia **attributo per attributo** (copia **superficiale**, *shallow copy*). Per attributi semplici va benissimo. Diventa un problema quando la classe ha un **puntatore a memoria dinamica**: la copia superficiale copia il puntatore, non l'area puntata, quindi i due oggetti finiscono per **condividere la stessa memoria**. Modificare uno modifica anche l'altro e, peggio, quando entrambi vengono distrutti il distruttore esegue due volte `delete[]` sullo stesso blocco. In questi casi bisogna scrivere un costruttore di copia che faccia una copia **profonda** (*deep copy*): allocare nuova memoria e copiarvi il contenuto.

```

#include <iostream>
#include <cstring>
using namespace std;
class MiaStringa {
public:
    MiaStringa(const char* s = "") {
        lung = strlen(s) + 1;
        testo = new char[lung];          // il costruttore alloca memoria...
        strcpy(testo, s);
    }
    MiaStringa(const MiaStringa& altra) { // costruttore di copia: copia PROFONDA
        lung = altra.lung;
        testo = new char[lung];          // nuova area di memoria
        strcpy(testo, altra.testo);      // con lo stesso contenuto
    }
    ~MiaStringa() { delete[] testo; }    // ...e il distruttore la libera
    void setCar(int i, char c) { testo[i] = c; }
    const char* get() const { return testo; }
private:
    char* testo;
    int lung;
};

int main() {
    MiaStringa a("ciao");
    MiaStringa b = a;                  // chiama il costruttore di copia
    b.setCar(0, 'm');
    cout << a.get() << " " << b.get() << endl; // ciao miao: sono indipendenti
    return 0;
}

```

DA RICORDARE

La «regola dei tre»

Se una classe ha bisogno di un **distruttore** scritto a mano (perché gestisce memoria dinamica), quasi certamente ha bisogno anche di un **costruttore di copia** e di un **operatore di assegnamento** `operator=` scritti a mano, per lo stesso motivo: le versioni generate dal compilatore copiano i puntatori invece dei dati. L'operatore di assegnamento si vede nella capitolo 4. Se non serve copiare oggetti, si può evitare il problema passando gli oggetti **per riferimento** e non copiandoli mai.

Il puntatore this

Quando si chiama `b.inverti()`, come fa il metodo a sapere che deve lavorare sugli attributi di `b`? Ogni metodo (non statico) riceve un parametro nascosto, il puntatore `this`, che contiene **l'indirizzo dell'oggetto su cui il metodo è stato chiamato**. Scrivere `num` dentro un metodo equivale a scrivere `this->num`. Di solito `this` resta implicito, ma serve esplicitamente in due casi. Il primo è distinguere un attributo da un parametro con lo stesso nome: in `Contatore(int valore) { this->valore = valore; }` a sinistra c'è l'attributo, a destra il parametro. Il secondo è **restituire l'oggetto stesso**: un metodo che termina con `return *this;` e ha tipo di ritorno riferimento alla classe permette di **concatenare le chiamate**. È la tecnica usata per gli inserimenti a catena, `lista.InsertHead(3).InsertHead(7);` e `bst.Insert(7).Insert(4).Insert(3);`.

```
#include <iostream>
using namespace std;
class Contatore {
public:
    Contatore(int valore = 0) { this->valore = valore; } // this-> distingue l'attributo
    Contatore& incrementa() { valore++; return *this; } // restituisce l'oggetto stesso
    int get() const { return valore; }
private:
    int valore;
};
int main() {
    Contatore c;
    c.incrementa().incrementa().incrementa(); // chiamate concatenate
    cout << c.get() << endl; // 3
}
```

Membri static, friend e sizeof

ATTRIBUTI E METODI STATIC

Normalmente ogni oggetto ha **la propria copia** di ogni attributo: due studenti hanno due matricole distinte. Un attributo dichiarato static invece è **uno solo, condiviso da tutti gli oggetti della classe**: appartiene alla classe, non al singolo oggetto, ed esiste anche se non è stato creato nessun oggetto. Va **dichiarato** dentro la classe e **definito** (con eventuale valore iniziale) **fuori**, una volta sola: `int Studente::contatore = 0;`. Se non gli si dà un valore esplicito viene inizializzato a 0. L'uso tipico è contare gli oggetti creati o generare codici univoci, per esempio i codici di una serie di prodotti.

Un **metodo** static appartiene anch'esso alla classe: si chiama con `NomeClasse::metodo()` (o anche tramite un oggetto) e, non essendo legato a nessun oggetto, **non ha il puntatore** `this`, quindi può usare solo gli attributi statici. Un attributo statico si può leggere sia come `Studente::contatore` sia tramite un oggetto qualsiasi, `s1.contatore`: il valore è lo stesso.

```
#include <iostream>
using namespace std;

class Studente {
public:
    Studente() { contatore++; matricola = contatore; } // ogni nuovo oggetto
    int getMatricola() const { return matricola; }
    static int getContatore() { return contatore; } // metodo statico
private:
    int matricola; // uno per ogni oggetto
    static int contatore; // UNO SOLO, condiviso da tutti gli oggetti
};
int Studente::contatore = 0; // definizione obbligatoria fuori dalla classe

int main() {
    Studente s1, s2, s3;
    cout << s2.getMatricola() << " " << Studente::getContatore() << endl; // 2 3
    cout << sizeof(Studente) << endl; // 4: i membri static non stanno nell'oggetto
    return 0;
}
```

VERIFICA

Qual è l'output? (attributo static)

```
class abc {
public:
    static int x;
    int i;
    abc() { i = ++x; }
};
int abc::x;
int main() { abc m, n, p; cout << n.x << " " << n.i << endl; }
```

Risposta **3 2**. `x` è statico e definito senza valore, quindi parte da 0. Ogni costruttore lo preincrementa e copia il nuovo valore nell'attributo `i` dell'oggetto in costruzione: `m.i = 1`, `n.i = 2`, `p.i = 3`, e alla fine `x = 3`. `n.x` è l'unico `x` condiviso (3), `n.i` è l'attributo di `n` (2). Le opzioni trappola sono «3 3» (chi pensa che anche `i` sia condiviso) e «1 3».

IL SIZEOF DI UNA CLASSE

Un oggetto occupa in memoria lo spazio dei suoi **attributi non statici** (più eventuale *padding* di allineamento, come per le `struct`). I **metodi non occupano spazio nell'oggetto**: il loro codice esiste una volta sola, nel segmento del codice, condiviso da tutte le istanze (lo collega all'oggetto giusto il puntatore `this`). Anche gli **attributi statici** non contano, perché stanno nella zona dei dati statici. Una classe senza attributi occupa comunque 1 byte, perché due oggetti distinti devono avere indirizzi distinti. Fa eccezione la capitolo 3: una classe con **metodi virtuali** contiene anche un puntatore nascosto alla tabella dei metodi virtuali (8 byte su un sistema a 64 bit).

VERIFICA

Qual è l'output? (sizeof)

```
class abc { void f(); void g(); int n; }; cout << sizeof(abc);
```

4: la dimensione dell'`int`; i metodi `f` e `g` non occupano spazio nell'istanza.

```
class abc { void f(); void g(); long double X; }; cout << sizeof(abc);
```

16: la dimensione del `long double` su g++ per architettura x86 a 64 bit (verificato). Non è un errore di compilazione: dichiarare metodi senza definirli è lecito finché non li si chiama.

FUNZIONI E CLASSI FRIEND

A volte una funzione esterna alla classe ha bisogno di accedere ai membri privati: il caso tipico è l'operatore `<<` per stampare un oggetto (capitolo 4). La classe può concedere questo privilegio dichiarandola `friend` al suo interno: `friend void foo(Classe& x);`. Una funzione amica **non è un membro** della classe (non si chiama con il punto, non ha `this`, non conta se la dichiarazione sta nella sezione pubblica o privata), ma **può accedere direttamente a tutti i membri**, anche privati e protetti. Si può dichiarare amica anche un'intera classe, `friend class Lista;`: tutti i metodi di `Lista` potranno accedere ai membri privati. È lo schema tipico delle strutture dati: il nodo tiene privati i suoi campi e dichiara amica la lista che lo gestisce. L'amicizia **non è simmetrica** (se `A` dichiara amica `B`, `B` non dichiara amica `A`) e **non si eredita**.

VERIFICA

class A { ... friend void foo(...); }; a) foo può accedere direttamente a tutti i membri di A pur non essendone membro; b) foo è un membro privato di A; c) foo può essere richiamata con la notazione A::foo; d) foo può accedere solo ai membri public di A.

Risposta **a)**. Le altre sono le classiche confusioni: una funzione friend non è un membro (quindi né b né c) e il suo privilegio è proprio l'accesso ai membri non pubblici (quindi non d).

Organizzare il codice in più file

Nei progetti reali il codice di una classe si divide in due file con lo stesso nome della classe: un **header** .h con la **definizione della classe** (attributi e prototipi dei metodi, cioè l'interfaccia) e un file .cpp con le **definizioni dei metodi** (l'implementazione). Il codice che usa la classe, detto **cliente**, include solo l'header. Così chi usa la classe vede l'interfaccia ma non i dettagli, e se si modifica l'implementazione basta ricompilare quel .cpp (**compilazione separata**). Ogni header va protetto da una **include guard** (#ifndef / #define / #endif) che impedisce di includerlo due volte nella stessa unità di compilazione, cosa che produrrebbe un errore di ridefinizione della classe.

VERIFICA

Un header definito dall'utente viene incluso usando la sintassi... a) #include "file.h" b) #include file.h c) #include <file> d) #include file.h

Risposta **a)**. Le virgolette dicono al preprocessore di cercare il file prima nella cartella del sorgente; le parentesi angolari sono per gli header di sistema (<iostream>, <string>).

```
// Studente.h - DICHIARAZIONE della classe (l'interfaccia)
#ifndef STUDENTE_H           // include guard: evita di includere il file due volte
#define STUDENTE_H
#include <string>
using namespace std;

class Studente {
public:
    Studente(int m, const string& n, double med);
    int getMatricola() const;
    void stampa() const;
private:
    int matricola;
    string nome;
    double media;
};
#endif
```

```
// Studente.cpp - DEFINIZIONE dei metodi (l'implementazione)
#include <iostream>
#include "Studente.h"          // virgolette: header scritto da noi

Studente::Studente(int m, const string& n, double med)
    : matricola(m), nome(n), media(med) {}
int Studente::getMatricola() const { return matricola; }
void Studente::stampa() const {
    cout << matricola << ": " << nome << " - media " << media << endl;
}

// compilazione
$ g++ Studente.cpp main.cpp -o prova      # si compilano tutti i .cpp, MAI i .h
```

Ereditarietà e polimorfismo

Ereditarietà e polimorfismo distinguono davvero la programmazione a oggetti: permettono di costruire nuove classi a partire da quelle esistenti e di trattare in modo uniforme oggetti di tipo diverso, scegliendo il comportamento corretto durante l'esecuzione.

Classi derivate

LA RELAZIONE «È UN»

Una **classe derivata** eredita attributi e metodi da una **classe base** già esistente e può **aggiungerne** di nuovi o **ridefinire** quelli ereditati. La relazione che l'ereditarietà esprime è «è un» (*is a*): uno studente borsista **è uno** studente, con in più una borsa di studio; un cerchio **è una** figura. Se la frase «X è un Y» suona strana, l'ereditarietà è probabilmente la scelta sbagliata. La sintassi indica dopo i due punti il **tipo di ereditarietà** e la classe base:

```
class ClasseDerivata : public ClasseBase {
    ...membri aggiunti o ridefiniti...
};
```

L'oggetto derivato **contiene** al suo interno un oggetto della classe base, più i nuovi attributi. Un `StudenteBorsista` ha quindi `matricola`, `nome` e `media` (ereditati) e `importo` (aggiunto); su di esso si possono chiamare sia i metodi della base (`getMatricola()`) sia quelli nuovi (`getImporto()`). Attenzione però all'accesso: i membri **privati** della base **esistono** nell'oggetto derivato ma **non sono accessibili** dai metodi della derivata, che deve passare per i metodi pubblici della base (i `get`). I membri **protetti** invece sono accessibili dalla derivata, ma restano nascosti all'esterno. È proprio per questo che esiste `protected`.

```
#include <iostream>
#include <string>
using namespace std;
class Studente {                                // classe BASE
public:
    Studente(int m, const string& n, double med)
        : nome(n), matricola(m), media(med) {} // stesso ordine della dichiarazione
    int getMatricola() const { return matricola; }
    double getMedia() const { return media; }
    void stampa() const {
        cout << matricola << ": " << nome << " - media " << media;
    }
protected:                                   // visibile anche nelle classi derivate
    string nome;
private:                                       // NON visibile nelle classi derivate
    int matricola;
    double media;
};
class StudenteBorsista : public Studente {    // classe DERIVATA
public:
    StudenteBorsista(int m, const string& n, double med, int imp)
        : Studente(m, n, med),               // 1) costruisce la parte "Studente"
```

```

        importo(imp) {} // 2) poi i nuovi attributi
int getImporto() const { return importo; } // metodo AGGIUNTO
void stampa() const { // metodo RIDEFINITO
    Studente::stampa(); // riusa la versione della base
    cout << " [borsa di " << importo << " euro]";
}
void saluta() const { cout << "Ciao " << nome << endl; } // nome e' protected: ok
// matricola e' private in Studente: qui si usa getMatricola()
private:
    int importo; // attributo AGGIUNTO
};
int main() {
    Studente s(1001, "Mario", 25.5);
    StudenteBorsista b(1002, "Anna", 28, 880);
    s.stampa(); cout << endl; // 1001: Mario - media 25.5
    b.stampa(); cout << endl; // 1002: Anna - media 28 [borsa di 880 euro]
    cout << b.getMatricola() << " " << b.getImporto() << endl; // ereditato + aggiunto
    b.saluta();
    return 0;
}

```

Il metodo stampa della derivata **ridefinisce** quello della base, cioè ne nasconde la versione ereditata: su un oggetto `StudenteBorsista` si esegue la nuova versione. Dentro di essa la versione della base resta raggiungibile qualificandola: `Studente::stampa()` stampa la parte comune e la derivata aggiunge solo ciò che manca, senza duplicare codice. Così la stampa di un borsista aggiunge alle informazioni comuni solo l'importo della borsa. Un dettaglio del costruttore: gli attributi vengono sempre inizializzati **nell'ordine in cui sono dichiarati nella classe**, non in quello della lista di inizializzazione; scrivere la lista nello stesso ordine evita sorprese (e l'avviso `-Wreorder di g++`).

I TIPI DI EREDITARIETÀ

Lo specificatore scritto prima della classe base (`public`, `protected` o `private`) stabilisce **come cambia la visibilità** dei membri ereditati. Le regole si ricordano con un'idea sola: la visibilità di un membro ereditato è **la più restrittiva** tra quella che aveva nella base e il tipo di ereditarietà. I membri privati della base non sono mai accessibili dalla derivata, qualunque sia il tipo.

MEMBRO NELLA BASE	EREDITARIETÀ PUBLIC	EREDITARIETÀ PROTECTED	EREDITARIETÀ PRIVATE
public	resta public	diventa protected	diventa private
protected	resta protected	resta protected	diventa private
private	inaccessibile	inaccessibile	inaccessibile

L'ereditarietà **pubblica** è di gran lunga la più usata, perché è l'unica che rispetta la relazione «è un»: l'interfaccia pubblica della base resta pubblica, e un oggetto derivato si può usare ovunque serva un oggetto base.

L'ereditarietà **privata** è quella **di default** per le `class` (per le `struct` il default è pubblico): se si dimentica lo specificatore, i membri pubblici della base diventano privati nella derivata e non sono più chiamabili dall'esterno. Serve a riusare l'implementazione di una classe nascondendola completamente all'utente. L'ereditarietà **protetta** mantiene accessibili i membri pubblici e protetti nelle classi derivate successive, nascondendoli all'esterno.

VERIFICA

```
class A {
    int a;
protected:
    char b;
public:
    float c;
};
class B : private A { ... };
```

Come eredita B i membri public e protected della classe A?

La classe B li eredita rendendoli entrambi privati: b e c sono accessibili dai metodi di B, ma non dall'esterno né da eventuali classi derivate da B. Il membro a (privato in A, perché in una class il default è privato) è ereditato ma inaccessibile. Le opzioni sbagliate tipiche dicono che B «eredita tutti i membri privati» o che li «rende public».

DA RICORDARE

Dimenticare public costa caro

Si consideri la dichiarazione `class D : public B1, B2 { ... };`. Lo specificatore vale **solo per la classe che segue**: B1 è ereditata pubblicamente, B2 **privatamente** (il default). Se si vuole l'ereditarietà pubblica da entrambe, bisogna scrivere public due volte: `class D : public B1, public B2`. Lo stesso errore, con una sola base, produce il messaggio «'Studente' is an inaccessible base of 'StudenteBorsista'» appena si prova a usare un puntatore alla base: se lo vedi, manca un public.

Costruttori, distruttori ed ereditarietà

ORDINE DI COSTRUZIONE E DISTRUZIONE

Un oggetto derivato contiene un oggetto base, e la parte base **deve esistere prima** che la si possa estendere. Per questo, quando si crea un oggetto derivato, si esegue **prima il costruttore della classe base**, poi quelli degli **attributi** che sono oggetti (nell'ordine di dichiarazione), infine il **corpo del costruttore della derivata**. La distruzione avviene **nell'ordine esattamente inverso**: prima il distruttore della derivata, poi quelli degli attributi, infine quello della base. È come costruire una casa: prima le fondamenta, poi gli impianti, poi l'arredamento; per demolirla si procede al contrario. I costruttori e il distruttore **non si ereditano**: ogni classe ha i propri, e ciascuno si occupa solo della propria parte.

```

#include <iostream>
using namespace std;

class Motore {
public:
    Motore() { cout << "  costruttore Motore" << endl; }
    ~Motore() { cout << "  distruttore Motore" << endl; }
};

class Veicolo {
public:
    Veicolo() { cout << "  costruttore Veicolo" << endl; }
    ~Veicolo() { cout << "  distruttore Veicolo" << endl; }
};

class Auto : public Veicolo {          // Auto E' UN Veicolo (ereditarieta')
public:
    Auto() { cout << "  costruttore Auto" << endl; }
    ~Auto() { cout << "  distruttore Auto" << endl; }
private:
    Motore m;                          // Auto HA UN Motore (composizione)
};

int main() {
    cout << "creazione:" << endl;
    Auto a;
    cout << "distruzione:" << endl;
    return 0;
}

```

```

creazione:
    costruttore Veicolo
    costruttore Motore
    costruttore Auto
distruzione:
    distruttore Auto
    distruttore Motore
    distruttore Veicolo

```

PASSARE I PARAMETRI AL COSTRUTTORE DELLA BASE

Se il costruttore della base ha parametri, la derivata deve chiamarlo **esplicitamente nella lista di inizializzazione**, con la sintassi `Derivata(parametri) : Base(argomenti), attributo(valore) { ... }`. È l'unico posto in cui lo si può fare: nel corpo del costruttore la parte base è già stata costruita. Se la derivata non lo chiama, il compilatore prova a usare il costruttore di default della base; se la base non ne ha uno, la compilazione fallisce. Nell'esempio, `StudenteBorsista(int m, const string& n, double med, int imp) : Studente(m, n, med), importo(imp) {}` passa matricola, nome e media al costruttore di `Studente` e inizializza da sé solo l'importo.

EREDITARIETÀ MULTIPLA E COMPOSIZIONE

Una classe può derivare da **più classi base**, elencandole separate da virgole, ciascuna con il proprio specificatore: `class Cavallo : public MezzoDiTrasporto, public Animale`. I costruttori delle basi vengono eseguiti **nell'ordine in cui le basi compaiono nella dichiarazione**, poi quello della derivata. Il rischio è l'**ambiguità**: se due basi hanno un membro con lo stesso nome, chiamarlo senza qualificarlo è un errore, e bisogna scrivere `MezzoDiTrasporto::stampa()` o `Animale::stampa()`. La parola chiave `virtual` davanti a una base (*ereditarietà virtuale*) serve a evitare che una base comune a due rami compaia due volte nell'oggetto finale.

```
#include <iostream>
using namespace std;
class B1 { public: B1() { cout << "oggetto della classe B1" << endl; } };
class B2 { public: B2() { cout << "oggetto della classe B2" << endl; } };
class D : public B1, public B2 {          // ereditarieta' multipla
public:
    D() { cout << "oggetto della classe D derivata da B1 e B2" << endl; }
};
int main() { D d1; return 0; }
```

La **composizione** è l'altro modo di riusare una classe: invece di derivarne una nuova, se ne mette **un oggetto come attributo**. Esprime la relazione «**ha un**» (*has a*): un'automobile **ha un** motore (non «è un» motore); un venditore **ha** un elenco di prodotti; uno **StudenteBorsista** **ha** una borsa di studio, che le tracce rappresentano con un attributo puntatore `BorsaDiStudio* borsa`. Nell'esempio di `p3_ordine.cpp`, `Auto` usa entrambe: deriva da `Veicolo` ed è composta da un `Motore`. La regola pratica è preferire la composizione quando la frase «X è un Y» non regge.

Polimorfismo e funzioni virtuali

PUNTATORI ALLA BASE E OGGETTI DERIVATI

Poiché un oggetto derivato è anche un oggetto base, un **puntatore (o un riferimento) alla classe base può puntare a un oggetto di una classe derivata**: `Studente* p = new StudenteBorsista(...)` è lecito. È questo che permette di mettere oggetti di tipo diverso nella **stessa struttura dati**: un array di `Figura*` che contiene cerchi e rettangoli, una pila di `Studente*` che contiene studenti normali e borsisti, una coda di `Item*` con libri e DVD. Il contrario non vale: un puntatore alla derivata non può puntare a un oggetto base, perché all'oggetto base mancano i membri aggiunti.

BINDING STATICO E DINAMICO

Il **binding** è il collegamento tra una chiamata di funzione e il codice che la implementa. Se una classe derivata ridefinisce un metodo e lo si chiama tramite un puntatore alla base, quale versione si esegue? Dipende da quando avviene il collegamento. Con il **binding statico**, deciso **in fase di compilazione**, il compilatore guarda il **tipo del puntatore** (`Base*`) e collega la versione della base, perché è l'unica informazione certa che ha. Con il **binding dinamico**, deciso **durante l'esecuzione**, si guarda il **tipo effettivo dell'oggetto puntato** e si esegue la versione della derivata. **In C++ il binding è statico per default**; per ottenere quello dinamico si dichiara il metodo `virtual` nella classe base.

DEFINIZIONE

Binding dinamico

Il **binding** è il collegamento tra una chiamata di funzione e il codice che la implementa. È **statico** se avviene in compilazione, in base al tipo del puntatore; è **dinamico** se avviene durante l'esecuzione, in base al tipo effettivo dell'oggetto. In C++ il binding dinamico si ottiene dichiarando il metodo `virtual`.

```

#include <iostream>
using namespace std;

class Base {
public:
    void nonVirtuale()      { cout << "Base::nonVirtuale" << endl; }
    virtual void virtuale() { cout << "Base::virtuale" << endl; }
    virtual ~Base() {}
};

class Derivata : public Base {
public:
    void nonVirtuale()      { cout << "Derivata::nonVirtuale" << endl; }
    void virtuale()         { cout << "Derivata::virtuale" << endl; } // override
};

int main() {
    Derivata d;
    Base* p = &d;           // puntatore alla BASE che punta a un oggetto DERIVATO
    p->nonVirtuale();        // binding STATICO: decide il tipo del puntatore -> Base
    p->virtuale();          // binding DINAMICO: decide il tipo dell'oggetto -> Derivata
    Base& r = d;             // lo stesso vale per i riferimenti
    r.virtuale();           // Derivata::virtuale
    Base b = d;             // COPIA solo la parte Base (slicing): niente polimorfismo
    b.virtuale();           // Base::virtuale
    return 0;
}

```

L'output è «Base::nonVirtuale», «Derivata::virtuale», «Derivata::virtuale», «Base::virtuale». Le prime due righe mostrano la differenza tra i due binding sullo stesso oggetto. La terza ricorda che il binding dinamico funziona anche con i **riferimenti**. L'ultima mostra il caso in cui **non** funziona: assegnando un oggetto derivato a un **oggetto** base (non a un puntatore o a un riferimento), si copia solo la parte base e il resto viene «tagliato» (*slicing*); b è a tutti gli effetti un oggetto Base. **Il polimorfismo passa sempre per puntatori o riferimenti.**

Il binding dinamico ha un piccolo costo. Ogni classe con metodi virtuali ha una **tabella dei metodi virtuali** (*vtable*) con gli indirizzi delle versioni corrette, e ogni oggetto contiene un puntatore nascosto alla tabella della propria classe; la chiamata virtuale passa per questa tabella invece di saltare direttamente al codice. Per questo è un po' meno efficiente del binding statico, e il C++, a differenza di Java, lascia al programmatore la scelta. Una volta dichiarato `virtual` nella base, un metodo resta virtuale in tutte le derivate, anche se lì non si ripete la parola chiave; dal C++11 si può aggiungere `override` dopo la firma nella derivata, così il compilatore segnala l'errore se la firma non corrisponde davvero a un metodo virtuale della base.

FUNZIONI VIRTUALI PURE E CLASSI ASTRATTE

A volte la classe base non ha un'implementazione sensata per un metodo: che area ha una «figura» generica? Si dichiara allora una **funzione virtuale pura**, scrivendo `= 0` al posto del corpo: `virtual double area() const = 0;`. Una classe che contiene **almeno una funzione virtuale pura** si dice **astratta**: **non si possono creare oggetti** di quella classe (`Figura f(0, 0);` non compila), ma si possono dichiarare puntatori e riferimenti, che è proprio ciò che serve al polimorfismo. Le classi derivate **devono** implementare tutte le funzioni virtuali pure ereditate; se ne lasciano anche una sola senza implementazione, sono astratte anch'esse. Una classe che implementa tutto si dice **concreta**. La classe astratta funziona da **contratto**: dichiara cosa ogni figura deve saper fare, e lascia a ciascuna sottoclasse il come.

DEFINIZIONE

Classe astratta

Una **funzione virtuale pura** si dichiara con `= 0` al posto del corpo. Una classe che contiene almeno una funzione virtuale pura è **astratta**: non si possono creare oggetti di quella classe, ma si possono usare puntatori e riferimenti, e le classi derivate devono implementare le funzioni virtuali pure.

```
#include <iostream>
using namespace std;

class Figura {                                // classe ASTRATTA
public:
    Figura(double x, double y) : x(x), y(y) {}
    virtual ~Figura() {}                     // distruttore virtuale (vedi oltre)
    virtual double area() const = 0;         // virtuale PURA: nessun corpo
    virtual void stampa() const {            // virtuale "normale": ha un comportamento base
        cout << "centro (" << x << ", " << y << ")";
    }
protected:
    double x, y;
};

class Cerchio : public Figura {
public:
    Cerchio(double x, double y, double r) : Figura(x, y), raggio(r) {}
    double area() const { return 3.14159 * raggio * raggio; } // obbligatoria
    void stampa() const {
        cout << "Cerchio, ";
        Figura::stampa();                    // riusa la versione della base
    }
private:
    double raggio;
};

class Rettangolo : public Figura {
public:
    Rettangolo(double x, double y, double b, double h) : Figura(x, y), base(b), alt(h) {}
    double area() const { return base * alt; }
    void stampa() const { cout << "Rettangolo, "; Figura::stampa(); }
private:
    double base, alt;
};

int main() {
    // Figura f(0, 0);                        // ERRORE: Figura e' astratta
    Figura* figure[3];                       // array di puntatori alla BASE
    figure[0] = new Cerchio(0, 0, 1);
    figure[1] = new Rettangolo(1, 1, 2, 3);
    figure[2] = new Cerchio(5, 5, 2);
    double totale = 0;
    for (int i = 0; i < 3; i++) {
        figure[i]->stampa();                 // ogni oggetto risponde a modo suo
        cout << " area " << figure[i]->area() << endl;
        totale += figure[i]->area();
    }
    cout << "Area totale: " << totale << endl;
    for (int i = 0; i < 3; i++)
        delete figure[i];                   // chiama il distruttore giusto grazie a virtual
    return 0;
}
```

```
}
```

Il ciclo nel main è l'essenza del polimorfismo: **lo stesso codice**, `figure[i]→area()`, esegue a ogni iterazione **una funzione diversa** a seconda dell'oggetto puntato. Se domani si aggiunge una classe `Triangolo`, il ciclo non va toccato. I vantaggi del polimorfismo sono proprio questi: una **stessa interfaccia** per lavorare con oggetti di classi diverse, un sistema più **flessibile** ed estendibile, la possibilità di costruire **strutture dati eterogenee** (array, liste, pile di oggetti di tipi diversi) e di gestire gerarchie di classi.

VERIFICA

Scegliere la corretta definizione di una funzione virtuale pura.

`virtual void f() = 0;`. Le varianti sbagliate tipiche sono `virtual void f() = 0 { }` (una virtuale pura non ha corpo nella dichiarazione), `void virtual f() = 0` e la mancanza di `virtual`.

Una classe astratta è una classe che: a) deve contenere tutte funzioni virtuali pure; b) deve contenere almeno una funzione virtuale pura; c) può non contenere funzioni virtuali pure; d) deve contenere funzioni virtuali pure definite fuori dalla classe.

Risposta **b)**. Basta **una** funzione virtuale pura per rendere astratta la classe; gli altri metodi possono essere normali o virtuali con implementazione, e la classe può avere attributi e costruttori (che le derivate chiamano nella lista di inizializzazione).

Che differenza c'è tra overloading e overriding?

L'**overloading** (sovraccaricamento) definisce **più funzioni con lo stesso nome e parametri diversi**, di solito nello stesso ambito; il compilatore sceglie quale chiamare in base agli argomenti, **in compilazione**. L'**overriding** (ridefinizione) avviene tra classe base e derivata: la derivata ridefinisce un metodo ereditato **con la stessa firma**; se il metodo è `virtual`, la versione da eseguire si sceglie **a run time** in base al tipo dell'oggetto (binding dinamico).

IL DISTRUTTORE VIRTUALE

Se si cancella un oggetto derivato **tramite un puntatore alla base**, `delete p`, quale distruttore viene chiamato? Il distruttore è un metodo come gli altri: se non è virtuale vale il binding statico, e si esegue **solo il distruttore della base**. La parte derivata non viene distrutta, e se aveva allocato memoria quella memoria si perde (*memory leak*). La regola è semplice: **se una classe ha metodi virtuali, ovvero è pensata per essere usata in modo polimorfico, il suo distruttore deve essere virtuale**. Con `virtual ~Figura() {}` il `delete figure[i]` chiama prima il distruttore di `Cerchio` o di `Rettangolo` e poi quello di `Figura`, nell'ordine corretto.

```
#include <iostream>
using namespace std;
class Base { public: ~Base() { cout << "~Base" << endl; } };           // NON virtuale
class Der : public Base {
public:
    Der() : dati(new int[1000]) {}
    ~Der() { delete[] dati; cout << "~Der" << endl; }
private:
    int* dati;
};
int main() { Base* p = new Der; delete p; }    // stampa solo ~Base: memory leak!
```

RICONOSCERE IL TIPO EFFETTIVO: DYNAMIC_CAST

Capita spesso di dover fare qualcosa **solo per alcuni oggetti** di una struttura eterogenea, per esempio calcolare il totale degli importi delle borse di studio dei soli studenti borsisti. Ci sono due soluzioni. La prima, più pulita, è **polimorfica**: si dichiara il metodo `virtual` già nella classe base, dove restituisce 0, e lo si ridefinisce nella derivata; poi si sommano i risultati di tutti gli oggetti senza chiedersi di che tipo siano. La seconda è il `dynamic_cast`: `dynamic_cast<StudenteBorsista*>(p)` restituisce il puntatore convertito se `p` punta davvero a uno `StudenteBorsista` (o a una sua derivata) e `nullptr` altrimenti. Funziona solo su classi **polimorfiche**, cioè con almeno un metodo virtuale (basta il distruttore).

```
#include <iostream>
using namespace std;
class Studente {
public:
    virtual ~Studente() {} // serve almeno un virtual per dynamic_cast
    virtual int getImportoBorsa() const { return 0; } // soluzione 1: virtual
};
class Borsista : public Studente {
public:
    Borsista(int i) : importo(i) {}
    int getImportoBorsa() const { return importo; }
private:
    int importo;
};
int main() {
    Studente* v[3] = { new Studente, new Borsista(880), new Borsista(600) };
    int tot1 = 0, tot2 = 0;
    for (int i = 0; i < 3; i++) {
        tot1 += v[i]->getImportoBorsa(); // soluzione 1: polimorfismo
        Borsista* b = dynamic_cast<Borsista*>(v[i]); // soluzione 2: dynamic_cast
        if (b != nullptr) // nullptr se NON e' un Borsista
            tot2 += b->getImportoBorsa();
    }
    cout << tot1 << " " << tot2 << endl; // 1480 1480
    for (int i = 0; i < 3; i++) delete v[i];
    return 0;
}
```

DA RICORDARE

Uno schema ricorrente

Molti programmi a oggetti seguono lo stesso disegno, che conviene avere già in mente. Una **classe base** con attributi privati o protetti, costruttore con lista di inizializzazione, metodi `get`, un metodo stampa **virtuale** e un **distruttore virtuale**. Una o più **classi derivate** che chiamano il costruttore della base, aggiungono i propri attributi e **ridefiniscono** stampa riusando `Base::stampa()`. Una **struttura dati di puntatori alla base** (array dinamico, lista, pila, coda o BST di Base*) che contiene oggetti dei vari tipi. Un `main` che crea gli oggetti con `new`, li inserisce nella struttura, la scorre chiamando i metodi virtuali, e alla fine libera tutto con `delete`. La dispensa di laboratorio costruisce questo scheletro per intero.

Template e overloading degli operatori

I template permettono di scrivere funzioni e classi una sola volta e di usarle con qualunque tipo; l'overloading degli operatori permette di usare con gli oggetti gli stessi operatori dei tipi predefiniti, a partire dalla stampa con `cout`.

La programmazione generica

Si osservino due funzioni di scambio, una per gli `int` e una per i `char`: il corpo è **identico**, cambia solo il tipo. Con l'overloading della capitolo 1 si dovrebbe scrivere una versione per ogni tipo, duplicando codice. La **programmazione generica** propone di scrivere **algoritmi astratti, indipendenti dal tipo dei dati**, e di lasciare che il tipo concreto venga deciso al momento dell'uso. In C++ lo strumento è il **template** (modello): una funzione o una classe in cui uno o più tipi sono **parametri**. Un template non è codice eseguibile: è uno **stampo** da cui il compilatore **genera automaticamente** le versioni concrete, una per ogni tipo con cui viene usato. Questa generazione si chiama **istanziamento** e avviene **a tempo di compilazione**, quindi non c'è alcuna perdita di prestazioni rispetto a scrivere le versioni a mano, e il controllo dei tipi resta completo. La libreria standard del C++ usa i template per offrire strutture contenitore già pronte (`vector`, `list`, `map`, ...); in questa dispensa però le strutture dati si implementano da sé, per capirne il funzionamento.

Template di funzioni

Un template di funzione si definisce premettendo la clausola `template <typename T>`, in cui `T` è un nome a scelta che fa da segnaposto per un tipo. Al posto di `typename` si può scrivere `class`: in questo contesto sono sinonimi (`class` è la forma storica, `typename` quella introdotta dallo standard ANSI/ISO). I parametri di tipo possono essere più d'uno: `template <typename T, typename U>`.

DEFINIZIONE

Template

Un **template** è una funzione o una classe in cui uno o più tipi sono parametri. Il compilatore genera automaticamente una versione concreta, detta **istanza**, per ogni tipo con cui il template viene usato.

```
#include <iostream>
#include <string>
using namespace std;
template <typename T>           // T e' un "segnaposto" per un tipo qualsiasi
T massimo(T a, T b) {
    return (a > b) ? a : b;      // richiede solo che T abbia l'operatore >
}
template <typename T>
void scambia(T& a, T& b) {
    T t = a;  a = b;  b = t;
}

template <typename T>
void stampaArray(const T v[], int n) {
    for (int i = 0; i < n; i++) cout << v[i] << " ";
    cout << endl;
}
```

```

int main() {
    cout << massimo(3, 7) << endl;           // istanza massimo<int>
    cout << massimo(3.5, 2.1) << endl;       // istanza massimo<double>
    cout << massimo(string("ape"), string("zebra")) << endl; // massimo<string>
    cout << massimo<double>(3, 7.5) << endl; // tipo esplicito: 3 diventa 3.0
    // cout << massimo(3, 7.5);             // ERRORE: T = int o T = double?
    int x = 1, y = 2;
    scambia(x, y);
    cout << x << " " << y << endl;         // 2 1
    char lettere[] = {'c', 'i', 'a', 'o'};
    stampaArray(lettere, 4);                // stampaArray<char>
    return 0;
}

```

Quando il compilatore incontra `massimo(3, 7)`, **deduce** `T = int` dagli argomenti e genera la funzione `int massimo(int a, int b)`; con `massimo(3.5, 2.1)` ne genera un'altra con `double`. Il template impone dei **requisiti impliciti** sul tipo: `massimo` usa `>`, quindi funziona con qualunque tipo che abbia l'operatore `>` (numeri, string, oppure classi nostre che lo definiscono, questo capitolo). Con una classe priva di `>` la compilazione fallisce nel punto dell'istanziamento. Il tipo si può anche indicare **esplicitamente** tra parentesi angolari: `massimo<double>(3, 7.5)` istanzia la versione `double` e converte 3 in 3.0.

DA RICORDARE

Quale chiamata genera un errore di compilazione?

```

template <typename T>
T max(T x, T y) { return (x > y) ? x : y; }

max(3, 7);    max(3.0, 7.0);    max(3, 7.0);    max<float>(3, 7.0);

```

La chiamata errata è `max(3, 7.0)`. Il compilatore deve dedurre un solo `T` dai due argomenti: dal primo ricava `int`, dal secondo `double`, e non sceglie da solo tra i due (per la deduzione dei template **non si applicano le conversioni implicite**). `max<float>(3, 7.0)` invece è corretta: il tipo è indicato esplicitamente e i due argomenti vengono convertiti in `float`. Provando davvero questo codice con `g++` e `using namespace std;`, si scopre che **tutte e quattro** le chiamate danno errore, per un motivo diverso: il nome `max` si scontra con `std::max` della libreria standard e il compilatore segnala un'ambiguità. Rinominando la funzione (per esempio `massimo`) resta solo l'errore di deduzione.

Un dettaglio importante: ogni istanza di un template è **una funzione diversa**. Se il template contiene una variabile locale `static`, ne esiste una copia **per ogni istanza**: `func<int>` e `func<double>` hanno ciascuna il proprio contatore.

```

#include <iostream>
using namespace std;
template <typename T>
void func(const T& x) {
    static int count = 0;           // UNA variabile static PER OGNI istanza del template
    cout << "x = " << x << " count = " << count << endl;
    ++count;
}
int main() {
    func<int>(1);
    func<int>(1);
    func<double>(1.1);              // altra funzione: count riparte da 0
    return 0;
}

```

VERIFICA

Qual è l'output? (template e static)

L'output è **x = 1 count = 0, x = 1 count = 1, x = 1.1 count = 0**. Le prime due chiamate usano la stessa istanza `func<int>`, quindi la variabile statica passa da 0 a 1. La terza usa `func<double>`, che è un'altra funzione con un altro `count`, inizializzato a 0. Per lo stesso motivo una funzione template che conta le proprie chiamate tiene un contatore separato per ogni tipo con cui viene usata.

Template di classi

DEFINIZIONE E ISTANZIAZIONE

Un **template di classe** è una classe in cui il tipo di alcuni attributi, parametri o valori di ritorno è un parametro. È lo strumento naturale per le **strutture dati contenitore**: una pila funziona allo stesso modo che contenga interi, stringhe o puntatori a studenti, e cambia solo il tipo degli elementi. A differenza dei template di funzione, per le classi il tipo va **sempre indicato esplicitamente** quando si crea un oggetto: `Pila<int> pi;`, `Pila<string> ps;`, `Pila<Studente*> studenti;`. `Pila` da solo non è un tipo, è uno stampo; `Pila<int>` è un tipo.

I metodi si possono definire dentro la classe, come sempre. Se li si definisce **fuori**, ogni definizione deve ripetere la clausola template e qualificare il nome con la classe **completa di parametro**: `template <typename T> void Pila<T>::push(const T& elem) { ... }`. Dimenticare `<T>` dopo il nome della classe è l'errore più comune.

```
#include <iostream>
#include <string>
using namespace std;
template <typename T>
class Pila {
public:
    Pila(int capacita = 4);
    ~Pila() { delete[] dati; }
    void push(const T& elem);
    T pop();
    bool isEmpty() const { return top == 0; } // definito nella classe: come sempre
private:
    T* dati;           // array dinamico di elementi di tipo T
    int dim;           // capacita' attuale
    int top;           // numero di elementi presenti
};

// fuori dalla classe: si ripete "template <typename T>" e si scrive Pila<T>::
template <typename T>
Pila<T>::Pila(int capacita) : dati(new T[capacita]), dim(capacita), top(0) {}

template <typename T>
void Pila<T>::push(const T& elem) {
    if (top == dim) { // pila piena: raddoppio lo spazio
        T* nuovo = new T[dim * 2];
        for (int i = 0; i < dim; i++) nuovo[i] = dati[i];
        delete[] dati;
        dati = nuovo;
        dim = dim * 2;
    }
    dati[top] = elem;
```

```

    top++;
}

template <typename T>
T Pila<T>::pop() {           // preconditione: pila non vuota
    top--;
    return dati[top];
}

int main() {
    Pila<int> pi;              // pila di interi
    for (int i = 1; i <= 6; i++) pi.push(i * 10);
    while (!pi.isEmpty()) cout << pi.pop() << " ";    // 60 50 40 30 20 10
    cout << endl;
    Pila<string> ps;           // stessa classe, elementi string
    ps.push("primo"); ps.push("secondo");
    cout << ps.pop() << endl;    // secondo
    return 0;
}

```

In questa pila l'array è allocato dinamicamente e, quando è pieno, push ne alloca uno grande il doppio, copia gli elementi e libera il vecchio. È l'**array di dimensione variabile**: la singola riallocazione costa $O(n)$, ma raddoppiando ogni volta il costo medio di un inserimento resta costante (capitolo 5).

VERIFICA

Data `template <class T> class abc { void f(); };`, qual è la definizione corretta di `f` fuori dalla classe? a) `template <class T> void abc<T>::f() {}` b) `template <class T> void abc::f() {}` c) `template <T> void abc<class T>::f() {}` d) `template <T> void abc<T>::f() {}`

Risposta **a)**: si ripete la clausola completa `template <class T>` e si qualifica il metodo con `abc<T>::`. Nella b) manca `<T>`, nella c) e nella d) la clausola `template` è scritta male.

PARAMETRI CHE NON SONO TIPI

Un template può avere anche parametri che sono **valori costanti**, non tipi, ed eventualmente un valore di default. Un esempio è una pila a dimensione fissa: `template <typename T, int n = 100> class Pila { T dati[n]; ... };`. Si istanzia con `Pila<int> pilaInt;` (100 elementi) oppure `Pila<double, 5> miniPila;` (5 elementi). Il valore deve essere noto **a tempo di compilazione**, perché diventa parte del tipo: `Pila<int, 5>` e `Pila<int, 10>` sono tipi diversi.

DOVE SI SCRIVE IL CODICE DI UN TEMPLATE

Il compilatore genera un'istanza solo quando vede l'uso del template con un tipo concreto, e per generarla deve **vedere il codice sorgente completo** del template, non solo i prototipi. Per questo non si possono distribuire template già compilati in una libreria binaria, e il codice di un template di classe **non si compila separatamente** come quello di una classe normale. La soluzione più comune è la **compilazione per inclusione**: l'header `.h` contiene la classe e, in fondo, un `#include` del file con le definizioni dei metodi (`#include "pila.cpp"`), così chiunque includa l'header riceve tutto il codice. In alternativa, e più semplicemente, si scrive tutto nell'header. Il file `.cpp` dei metodi **non va passato** a g++ sulla riga di comando: è già incluso. Lo standard prevedeva anche un modello di compilazione separata con la parola chiave `export`, che però non è mai stato supportato davvero ed è stato **rimosso in C++11**. Con un programma in un unico file il problema non si pone.

DA RICORDARE

Template e polimorfismo

Conviene confrontare i due modi di scrivere codice che funziona con tipi diversi. Una **funzione polimorfica** (capitolo 3) supporta tipi diversi perché riceve un puntatore o un riferimento a una classe base: la scelta del codice avviene **a run time**, funziona solo all'interno di una **gerarchia di classi** e con puntatori. Una **funzione template** è un **generatore automatico di funzioni sovraccaricate**: la scelta avviene **in compilazione**, funziona con qualunque tipo che rispetti i requisiti (anche `int` e `double`, che non appartengono a nessuna gerarchia), ma genera un eseguibile più grande perché il codice viene duplicato per ogni tipo. Scrivere un template significa ragionare in astratto, evitando qualunque dipendenza da tipi e costanti particolari.

Overloading degli operatori

L'IDEA

Per i tipi predefiniti scriviamo `a + b`, `a == b`, `cout << a` senza pensarci. Per una classe nostra queste espressioni non hanno significato finché non lo definiamo. L'**overloading degli operatori** (sovraccaricamento) permette di farlo: un operatore è semplicemente una **funzione con un nome speciale**, `operator` seguito dal simbolo (`operator+`, `operator==`, `operator<<`), e quando il compilatore incontra `a + b` con `a` di una nostra classe cerca una funzione `operator+` adatta. Il codice che usa la classe diventa leggibile come quello sui numeri: `ora1 + ora2` invece di `ora1.somma(ora2)`.

Ci sono dei limiti. Si possono ridefinire quasi tutti gli operatori, ma **non** `.` (accesso al membro), `::` (risoluzione di visibilità), `?:` (condizionale), `sizeof` e `.*`. **Non si possono inventare operatori nuovi**, né cambiare la **precedenza**, l'**associatività** o il **numero di operandi** di quelli esistenti: `*` resta binario e ha sempre precedenza su `+`. Almeno un operando deve essere di un tipo definito dall'utente: non si può ridefinire la somma tra due `int`. È buona norma, infine, dare all'operatore un significato **coerente** con quello abituale.

MEMBRO O FUNZIONE ESTERNA

Un operatore binario si può definire in due modi. Come **metodo della classe**, ha **un solo parametro**: l'operando sinistro è l'oggetto su cui è chiamato (`*this`), il parametro è l'operando destro. `a + b` diventa `a.operator+(b)`. Come **funzione esterna**, di solito dichiarata `friend` per accedere ai membri privati, ha **due parametri**, uno per operando: `a + b` diventa `operator+(a, b)`. Analogamente un operatore **unario** come metodo non ha parametri (`++a` diventa `a.operator++()`), come funzione esterna ne ha uno.

Quando si sceglie l'uno o l'altro? Gli operatori che **modificano l'oggetto** (`+=`, `-=`, `++`, `--`, `=`) sono di solito **membri** e restituiscono `*this` per riferimento. Gli operatori in cui l'**operando sinistro non è un oggetto della nostra classe** devono essere **funzioni esterne**: il caso decisivo è `operator<<`, dove l'operando sinistro è `cout`, un oggetto della classe `ostream` che non possiamo modificare. Gli operatori simmetrici come `+` e `==` si possono fare in entrambi i modi; nell'esempio `operator+` è un metodo, ma potrebbe essere anche una funzione `friend`.

```
#include <iostream>
using namespace std;

class Ora {
public:
    Ora(int o = 0, int m = 0, int s = 0) : ore(o), min(m), sec(s) { aggiusta(); }

    // operatori MEMBRI: l'operando sinistro e' l'oggetto stesso (*this)
    Ora& operator+=(int secondi) {                // a += 90
        sec += secondi;
        aggiusta();
        return *this;                            // permette a += 1 += ... e cout << (a += 1)
    }
};
```

```

}
Ora operator+(const Ora& altra) const { // a + b: nuovo oggetto, a e b invariati
    return Ora(ore + altra.ore, min + altra.min, sec + altra.sec);
}
Ora& operator++() { // ++a (prefisso): incrementa e restituisce a
    return *this += 1;
}
Ora operator++(int) { // a++ (postfisso): int fittizio
    Ora vecchia = *this; // copia del valore prima dell'incremento
    *this += 1;
    return vecchia;
}
bool operator<(const Ora& altra) const { // serve per ordinare o per un BST
    return totaleSecondi() < altra.totaleSecondi();
}
operator int() const { return totaleSecondi(); } // conversione Ora -> int
// operatori FRIEND: funzioni esterne con accesso ai membri privati
friend bool operator==(const Ora& a, const Ora& b);
friend ostream& operator<<(ostream& os, const Ora& x);
private:
    int totaleSecondi() const { return ore * 3600 + min * 60 + sec; }
    void aggiusta() { // riporta sec e min tra 0 e 59
        min += sec / 60; sec %= 60;
        ore += min / 60; min %= 60;
        ore %= 24;
    }
    int ore, min, sec;
};

bool operator==(const Ora& a, const Ora& b) {
    return a.ore == b.ore && a.min == b.min && a.sec == b.sec;
}
ostream& operator<<(ostream& os, const Ora& x) { // restituisce il flusso
    os << x.ore << ":" << x.min << ":" << x.sec;
    return os;
}

int main() {
    Ora a(10, 59, 30), b(1, 0, 45);
    cout << a << " + " << b << " = " << a + b << endl; // 10:59:30 + 1:0:45 = 12:0:15
    a += 40; cout << a << endl; // 11:0:10
    Ora c = a++; cout << c << " " << a << endl; // 11:0:10 11:0:11
    ++a; cout << a << endl; // 11:0:12
    cout << (a == c) << " " << (c < a) << endl; // 0 1
    int s = b; cout << s << endl; // 3645 (Ora -> int)
    Ora d = 5; cout << d << endl; // 5:0:0 (int -> Ora)
    return 0;
}

```

GLI OPERATORI DA CONOSCERE

`operator<<` **per la stampa**. È l'operatore più usato: permette di stampare un prodotto, uno studente o un'intera struttura dati con una sola istruzione. La firma è sempre la stessa: `friend ostream& operator<<(ostream& os, const Classe& x);`. Riceve il flusso per riferimento (i flussi non si copiano) e l'oggetto per riferimento costante, scrive sul **parametro** `os` (non su `cout`, così funziona anche con i file) e **restituisce** `os`: è questo che permette di concatenare `cout << a << b`, perché `cout << a` restituisce di nuovo `cout`. Simmetricamente, `friend istream& operator>>(istream& is, Classe& x)` legge un oggetto da un flusso (qui l'oggetto non è `const`, perché viene modificato).

Operatori di confronto. `operator==` e `operator<` restituiscono `bool` e si dichiarano `const`. Servono più di quanto sembri: un template come `massimo`, un algoritmo di ordinamento o un **BST di oggetti** (capitolo 8) confrontano gli elementi con `<` e `==`; definire questi operatori nella classe `Studiante` (per esempio confrontando le matricole) rende la classe utilizzabile con quelle strutture senza modificarle.

Incremento e decremento. Per distinguere il **prefisso** `++a` dal **postfisso** `a++`, il C++ usa un trucco: il postfisso si dichiara con un **parametro** `int fittizio`, mai usato. Il prefisso incrementa e restituisce l'oggetto stesso per riferimento; il postfisso deve restituire il **valore precedente**, quindi salva una copia, incrementa e restituisce la copia per valore. Nell'esempio, `Ora c = a++`; mette in `c` l'ora prima dell'incremento.

Assegnamento. `operator=` è generato automaticamente dal compilatore con una copia membro a membro, come il costruttore di copia, e come lui va riscritto quando la classe gestisce memoria dinamica (la «regola dei tre» della capitolo 2). Si definisce come membro, restituisce `*this` per riferimento (per permettere `a = b = c`) e deve gestire l'autoassegnamento `a = a`: prima di liberare la propria memoria controlla `if (this != &altro)`.

CONVERSIONI DI TIPO

Restano le **conversioni di dati e gli operatori di conversione di tipo**. Le conversioni tra una classe e un altro tipo si ottengono in due direzioni. **Da un altro tipo verso la classe:** un **costruttore con un solo argomento** (o con gli altri parametri di default) funge da conversione implicita. Poiché `Ora(int o = 0, int m = 0, int s = 0)` si può chiamare con un solo `int`, la scrittura `Ora d = 5`; costruisce l'ora 5:00:00. Se non si vuole questo comportamento si dichiara il costruttore `explicit`. **Dalla classe verso un altro tipo:** si definisce un **operatore di conversione**, un metodo senza tipo di ritorno dichiarato il cui nome è il tipo di destinazione, `operator int()` `const { ... }`. Con esso `int s = b`; converte l'ora nel numero totale di secondi. Le conversioni implicite vanno usate con parsimonia: rendono il codice compatto ma possono creare ambiguità quando il compilatore trova più strade possibili.

DA RICORDARE

Riepilogo delle firme

Metodi della classe `X`: `X& operator+=(const X& o)`, `X operator+(const X& o) const`, `bool operator==(const X& o) const`, `bool operator<(const X& o) const`, `X& operator++()` (prefisso), `X operator++(int)` (postfisso), `X& operator=(const X& o)`, `operator int() const` (conversione). Funzioni esterne amiche: `friend ostream& operator<<(ostream& os, const X& x)`, `friend istream& operator>>(istream& is, X& x)`, `friend X operator+(const X& a, const X& b)`. Nella maggior parte dei programmi basta `operator<<` per la stampa, e `operator<` o `operator==` se servono confronti tra oggetti.

La complessità computazionale

Come cresce il costo di un algoritmo quando cresce la dimensione dei dati? Il capitolo parte dalle idee di base, arriva alle notazioni asintotiche, costruisce una scala delle funzioni e un insieme di regole pratiche per leggere la complessità direttamente dal codice.

Che cosa misuriamo

IL TEMPO DIPENDE DALLA DIMENSIONE DELL'INPUT

Immagina di dover cercare un nome. Se l'elenco ha 10 nomi, qualunque metodo va bene. Se ne ha 10 milioni, il metodo fa la differenza tra un istante e un'ora. La **complessità computazionale** studia proprio questo: **come cresce il costo di un algoritmo quando cresce la dimensione dei dati** su cui lavora. La dimensione dell'input si indica con **n**: il numero di elementi dell'array da ordinare, di nodi della lista, di chiavi nell'albero. Il costo è una funzione di **n**, scritta $T(n)$.

Il costo si può misurare in **tempo** (quanto lavora l'algoritmo) o in **spazio** (quante celle di memoria usa, oltre a quelle che contengono l'input). In questo capitolo, salvo indicazione contraria, si parla di tempo. Ma il tempo in secondi dipende dal computer, dal compilatore, da cosa sta facendo il sistema operativo: due esecuzioni dello stesso programma non durano mai uguale. Per questo si fanno due **ipotesi semplificative**.

La **prima ipotesi** è contare le **operazioni elementari** invece dei secondi: un assegnamento, un confronto, un'operazione aritmetica, un accesso a una cella di un array, una lettura o scrittura costano tutte **1** (è il modello della macchina RAM). Il tempo diventa così una proprietà dell'algoritmo, non del computer. La **seconda ipotesi** è ancora più importante: trovare la funzione esatta del numero di operazioni è spesso difficilissimo, e comunque inutile. **Basta il comportamento asintotico**, cioè come si comporta la funzione **quando n diventa molto grande**. È lì che gli algoritmi si distinguono davvero.

CONTARE I PASSI: UN PRIMO ESEMPIO

Prendiamo un ciclo che somma gli elementi di un array di **n** interi: `for (int i = 0; i < n; i++) somma += v[i];`. L'inizializzazione `i = 0` si esegue una volta; il confronto `i < n` $n + 1$ volte (l'ultima è quella che fa uscire); il corpo e l'incremento n volte ciascuno. Il totale è qualcosa come $T(n) = 3n + 2$. Ora la domanda giusta non è «quanto vale esattamente», ma «come cresce»: se **n** raddoppia, $T(n)$ raddoppia (circa). Il 3 e il 2 dipendono da come si contano le operazioni e da quanto costa ciascuna sulla macchina vera, quindi non ci dicono nulla sull'algoritmo. Quello che conta è che $T(n)$ cresce **come n**: diremo che il ciclo è **lineare**, e scriveremo $T(n) = O(n)$.

DA RICORDARE

L'analogia del conto al ristorante

Quando si stima una spesa grande si trascurano le voci piccole. Se una cena costa $5n^2 + 2n + 10$ euro, con $n = 1000$ commensali fanno $5.000.000 + 2.000 + 10$ euro: il primo termine è il 99,96% del totale. Gli altri ci sono, ma **diventano irrilevanti al crescere di n**. E anche il 5 davanti a n^2 non cambia la «natura» del costo: che il prezzo a persona sia 5 o 7 euro, raddoppiando gli invitati il conto quadruplica comunque. La notazione asintotica formalizza esattamente queste due semplificazioni: **si tiene solo il termine che cresce più in fretta e si butta via la sua costante**.

CASO MIGLIORE, PEGGIORE E MEDIO

Lo stesso algoritmo, con input diversi della stessa dimensione, può fare un numero di passi diverso. Nella **ricerca sequenziale** di un valore in un array di n elementi, se il valore è nella prima cella basta 1 confronto; se è nell'ultima, o non c'è, ne servono n . Si distinguono quindi tre casi. Il **caso migliore** è l'input più favorevole (1 confronto). Il **caso peggiore** (o **pessimo**) è il costo **massimo** tra tutti gli input di dimensione n (n confronti). Il **caso medio** è il costo **mediato** su tutti gli input possibili: se il valore si trova con uguale probabilità $1/n$ in ciascuna posizione i , il numero medio di confronti è $(1 + 2 + \dots + n)/n = (n + 1)/2$, che cresce ancora come n .

Quando non è specificato, la complessità di un algoritmo si intende **nel caso peggiore**. È la garanzia più utile: qualunque sia l'input, l'algoritmo non farà peggio di così. Quando però il caso è indicato esplicitamente («nel caso ottimo», «nel caso peggiore»), va letto con attenzione: l'insertion sort è $O(n)$ nel caso migliore e $O(n^2)$ nel peggiore.

VERIFICA

Nel caso peggiore, qual è il numero di confronti necessari per la ricerca di un elemento all'interno di una lista concatenata semplice? a) $\log n$ b) $n/2$ c) $\log(n-1)$ d) n

Risposta **d) n** . In una lista si può solo scorrere un nodo dopo l'altro dalla testa; nel caso peggiore l'elemento è l'ultimo o non c'è, e si confrontano tutti gli n nodi. « $n/2$ » è il caso medio, arrotondato; i logaritmi sono il costo della ricerca binaria, impossibile su una lista perché non si può saltare al nodo centrale in tempo costante.

Le notazioni asintotiche

O GRANDE: IL LIMITE SUPERIORE

Ecco la definizione formale. Data una funzione $g(n)$, **$O(g(n))$** è l'insieme delle funzioni $f(n)$ per cui **esistono due costanti positive c e n_0 tali che $0 \leq f(n) \leq c \cdot g(n)$ per ogni $n > n_0$** . In parole: da un certo punto in poi ($n > n_0$), $f(n)$ sta **sotto** $g(n)$ moltiplicata per una costante opportuna. Si scrive $f(n) = O(g(n))$, che va letto « f appartiene a O di g » (il simbolo $=$ è un abuso di notazione per $\in$). La notazione O definisce quindi un **limite superiore** alla crescita di f : per n abbastanza grande, f cresce **al più** come g .

DEFINIZIONE

Notazione O grande

$f(n) = O(g(n))$ se esistono due costanti positive c e n_0 tali che $0 \leq f(n) \leq c \cdot g(n)$ per ogni $n > n_0$. La notazione O fornisce un **limite superiore** alla crescita di f : da un certo punto in poi f cresce al più come g .

Le due costanti servono a formalizzare le due semplificazioni dell'analogia del ristorante. La **c** assorbe le costanti moltiplicative (il 5 di $5n^2$); la **n_0** permette di ignorare i valori piccoli di n , dove i termini minori possono ancora pesare. Per mostrare che $8n + 2 = O(n)$ basta trovare una coppia che funzioni: con $c = 9$ serve $8n + 2 \leq 9n$, cioè $n \geq 2$, quindi $n_0 = 2$ va bene. Per un polinomio basta sommare i coefficienti: $5n^4 + 3n^3 + 2n^2 + 4n + 7 \leq (5 + 3 + 2 + 4 + 7) \cdot n^4 = 21n^4$ per ogni $n \geq 1$, perché ogni potenza di n è al più n^4 quando $n \geq 1$. Quindi è $O(n^4)$: **un polinomio è O della sua potenza massima**.

VERIFICA

Diciamo che $f(n) = O(g(n))$ se e solo se esistono $c, n_0 > 0$ tali che: a) $f(n) \geq c \cdot g(n)$ per ogni $n > n_0$; b) $c \cdot g(n) \leq f(n)$ per ogni $n > n_0$; c) $f(n) > c \cdot g(n)$ per ogni $n > n_0$; d) $0 \leq f(n) \leq c \cdot g(n)$ per ogni $n > n_0$.

La risposta è **$0 \leq f(n) \leq c \cdot g(n)$ per ogni $n > n_0$** : f sta sotto $c \cdot g$. Le varianti con $\geq$ descrivono invece la notazione Ω (limite inferiore). Trucco mnemonico: la O ha la forma di un «tetto» sopra la funzione.

OMEGA E THETA

Simmetricamente, $f(n) = \Omega(g(n))$ (omega grande) se esistono $c, n_0 > 0$ tali che $f(n) \geq c \cdot g(n)$ per ogni $n > n_0$: è un **limite inferiore**, f cresce **almeno** come g . Infine $f(n) = \Theta(g(n))$ (theta grande) se esistono tre costanti positive c_1, c_2 e n_0 tali che $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ per ogni $n > n_0$: f è stretta tra due multipli di g , quindi cresce **esattamente** come g . Equivalentemente, $f = \Theta(g)$ se e solo se $f = O(g)$ e $f = \Omega(g)$. Θ è una condizione più forte di O .

Un dettaglio che confonde: O grande dà un limite superiore, non il limite superiore. Se $T(n) = O(n^4)$, è vero anche che $T(n) = O(n^7)$ e $T(n) = O(n^4 \log n)$, perché sono funzioni ancora più grandi. Sono affermazioni vere ma poco informative; quando si conoscono più limiti superiori **si preferisce il più piccolo**. Quando si chiede «qual è la complessità» si intende sempre il limite più stretto, cioè in pratica il Θ .

NOTAZIONE	DEFINIZIONE (PER $n > n_0$)	SIGNIFICATO	ANALOGIA
$f = O(g)$	$0 \leq f(n) \leq c \cdot g(n)$	f cresce al più come g	$\leq$
$f = \Omega(g)$	$f(n) \geq c \cdot g(n)$	f cresce almeno come g	$\geq$
$f = \Theta(g)$	$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$	f cresce esattamente come g	$=$

COMPLESSITÀ DI UN ALGORITMO E DI UN PROBLEMA

Per un algoritmo A , dimostrare che il suo tempo è $O(t(n))$ stabilisce un **limite superiore** (A non fa peggio di così); dimostrare che è anche $\Omega(t(n))$ stabilisce un **limite inferiore**; se i due coincidono, la complessità di A è $\Theta(t(n))$. La notazione Ω si applica anche ai **problemi**: dire che un problema P ha complessità $\Omega(f(n))$ significa che **qualsiasi** algoritmo che lo risolve impiega almeno quel tempo. Un algoritmo A che risolve P si dice **ottimale** se P è $\Omega(f(n))$ e A è $O(f(n))$: non si può fare asintoticamente meglio. L'esempio classico è l'**ordinamento basato su confronti**, che ha complessità $\Omega(n \log n)$: ogni algoritmo che ordina confrontando elementi richiede almeno $n \log n$ passi nel caso peggiore, quindi merge sort, che è $O(n \log n)$, è ottimale.

VERIFICA

Due domande sulla definizione

Ogni algoritmo di ordinamento basato su confronti richiede almeno: a) $n \log n$ passi b) $n \cdot n$ passi c) $\log n$ passi d) nessuna delle precedenti

Risposta **a)**, nel caso peggiore: è il limite inferiore $\Omega(n \log n)$ del problema dell'ordinamento.

Cosa intendiamo quando diciamo che un algoritmo X è asintoticamente più efficiente di un altro algoritmo Y ?

Che **X è sempre migliore di Y per grandi input** (da un certo n_0 in poi). Per input piccoli può anche succedere il contrario: un algoritmo $O(n^2)$ con costanti piccole può battere uno $O(n \log n)$ con costanti grandi quando n vale 10. Le opzioni sbagliate parlano di «piccoli input» o invertono X e Y .

La scala delle funzioni

LE CLASSI DI COMPLESSITÀ

La notazione O divide le funzioni in **classi**: tutte quelle che crescono con lo stesso ordine di grandezza finiscono nella stessa classe, qualunque siano le costanti. Le classi che compaiono più spesso sono poche e si dispongono in una **scala**, dalla più lenta a crescere alla più veloce. **Imparare questa scala a memoria è metà del lavoro.**

CLASSE	NOME	TIPICO ESEMPIO
$O(1)$	costante	accesso a $v[i]$; inserimento in testa a una lista; push/pop di una pila
$O(\log n)$	logaritmica	ricerca binaria; ciclo che dimezza o raddoppia; ricerca in un BST bilanciato
$O(\sqrt{n}), O(n^c)$ con $c < 1$	sottolineare	ciclo con $i * i \leq n$
$O(n)$	lineare	scorrere un array o una lista; ricerca sequenziale; visita di un albero
$O(n \log n)$	quasi lineare	merge sort; quick sort nel caso medio
$O(n^2), O(n^3), O(n^c)$	polinomiale (quadratica, cubica...)	due o tre cicli annidati; selection, insertion, bubble sort
$O(2^n), O(c^n)$	esponenziale	provare tutti i sottoinsiemi di n elementi
$O(n!), O(n^n)$	oltre l'esponenziale	provare tutte le permutazioni

La scala da ricordare, con $c > 1$ costante, è quindi: $1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \dots < n^c < 2^n < 3^n < \dots < c^n < n! < n^n$. In forma compatta: $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^c) < O(c^n)$.

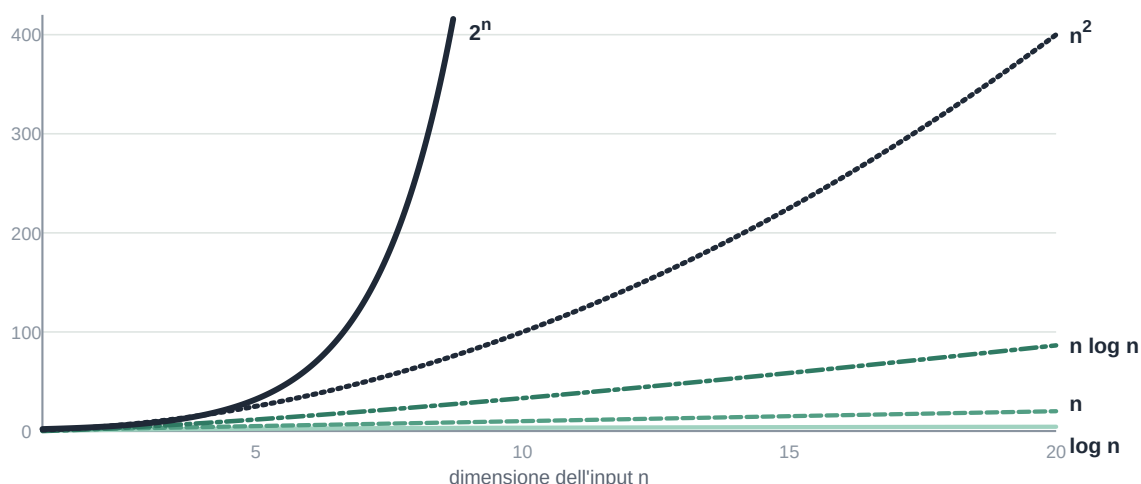


Figura 5.1. Crescita delle principali funzioni di complessità per n da 1 a 20

Il grafico mostra perché le differenze contano. Già con $n = 20$, 2^n supera il milione mentre n^2 vale 400; $\log n$ quasi non si stacca dallo zero. La tabella seguente rende l'idea per valori più grandi: un algoritmo esponenziale diventa inutilizzabile già con $n = 100$, mentre uno $O(n \log n)$ gestisce un milione di elementi con circa venti milioni di operazioni, poche decine di millisecondi su un computer moderno.

N	LOG N	N LOG N	N ²	2 ^N
10	3,3	33	100	1.024
100	6,6	664	10.000	1,3 · 10 ³⁰
1.000	10	9.966	1.000.000	1,1 · 10 ³⁰¹
1.000.000	20	≈ 2 · 10 ⁷	10 ¹²	incalcolabile

COME CONFRONTARE DUE FUNZIONI

Le domande «qual è la funzione asintoticamente più grande / più piccola» e «qual è la relazione tra T1 e T2» si risolvono con poche regole, da applicare nell'ordine. **Primo:** di ogni funzione si tiene solo il **termine dominante** (il più alto nella scala) e si eliminano le costanti moltiplicative. **Secondo:** si confrontano i termini dominanti con la scala. **Terzo,** se sono nella stessa famiglia, si usano queste regole di dettaglio.

Tra esponenziali vince la base maggiore: 5^n cresce più di 4^n , perché $5^n/4^n = (5/4)^n$ va all'infinito. Un fattore polinomiale non cambia il verdetto: $5^n \cdot n \log n$ resta più grande di $4^n \cdot n$. **Tra polinomi vince l'esponente maggiore:** $n^3 > n^{2,5} > n^2$. **Qualunque esponenziale (base > 1) batte qualunque polinomio,** e **qualunque polinomio n^ϵ ($\epsilon > 0$) batte qualunque potenza del logaritmo:** $(\log n)^{100}$ cresce meno di $\sqrt{n}$. **Tra prodotti con un fattore comune** si confronta il resto: $n \log^2 n$ contro $1,5 n \log n$ diventa $\log n$ contro $1,5$, e vince il primo.

Alcuni termini sembrano grandi e non lo sono. $1^n = 1$ è una costante. $1/2^n$ tende a zero: aggiungerlo a una funzione non cambia nulla. **La base del logaritmo non conta:** $\log_2 n$ e $\log_{10} n$ differiscono solo per una costante moltiplicativa ($\log_2 n = \log_{10} n / \log_{10} 2$), quindi sono la stessa classe $O(\log n)$. **Il logaritmo di una potenza è un logaritmo:** $\log(n^k) = k \log n = O(\log n)$.

DA RICORDARE

Domande «qual è la funzione più grande / più piccola»

Qual è la funzione asintoticamente più grande? a) $8n^3 + 1/2^n$ b) $(2 + n)^3 + \log n + \log^2 n$ c) 4^n d) n^3

Risposta **c) 4^n** . Le altre tre sono tutte $\Theta(n^3)$: in a) $1/2^n$ tende a zero, in b) $(2 + n)^3$ ha termine dominante n^3 . Un esponenziale batte qualsiasi polinomio. La a) è una trappola: il termine $1/2^n$ tende a zero e non la rende esponenziale.

Qual è la funzione asintoticamente più piccola? a) $3^n + 5n + \log^n n$ b) $(10 + n)^3$ c) 1^n d) $2n$

Risposta **c) 1^n** , che vale 1 per ogni n ed è quindi costante, $O(1)$. Vengono poi $2n$ (lineare), $(10 + n)^3$ (cubica) e infine la a) (esponenziale).

Qual è la relazione tra $T1 = O((5^n) n \log n)$ e $T2 = O((4^n) n)$?

$T1 > T2$: tra esponenziali vince la base maggiore, e i fattori polinomiali non contano.

Qual è la relazione tra $T1 = O(9 n \log^3 n)$ e $T2 = O(3^n \cdot n + 4^n \cdot n \log n)$?

$T1 < T2$: $T1$ è quasi lineare (polinomio per potenza di logaritmo), $T2$ è esponenziale (dominato da $4^n \cdot n \log n$).

ESERCIZIO

Quale relazione contiene almeno un errore? ($c > 1$)

a) $O(c) < O(\log n) < O(\log^c n)$: **corretta** (costante, poi logaritmo, poi potenza del logaritmo). b) $O(\log n) < O(n \log n) < O(\log^c n)$: **errata**, perché una potenza del logaritmo cresce meno di $n \log n$, quindi l'ultima disuguaglianza è al contrario. c) $O(n) < O(\log n) < O(n^c)$: **errata**, n cresce più di $\log n$. d) $O(\log n) < O(\log^c n) < O(n^2)$: **corretta**.

Regole di calcolo: somme e prodotti

Le domande del tipo « $T_1 = \dots$, $T_2 = \dots$, qual è l'ordine di grandezza di $G = T_1 + T_2$ (oppure $T_1 \cdot T_2$, oppure $\log(T_1/T_2)$)» si risolvono con quattro regole. Le prime tre si enunciano così: se $f(n) = O(g(n))$ allora $a \cdot f(n) = O(g(n))$ per ogni costante a (**le costanti moltiplicative spariscono**); se $f = O(g)$ e $g = O(h)$ allora $f = O(h)$ (**transitività**); se $f_1 = O(g_1)$, $f_2 = O(g_2)$ e $g_1 = O(g_2)$, allora $f_1 + f_2 = O(g_2)$ (**nella somma vince il termine più grande**). La quarta regola, per i **prodotti**, è che **gli ordini si moltiplicano**: $O(f) \cdot O(g) = O(f \cdot g)$.

REGOLA	IN FORMULA	ESEMPIO
costanti	$O(a \cdot f) = O(f)$	$5n^2 = O(n^2)$
somma	$O(f) + O(g) = O(\max(f, g))$	$n^2 + \sqrt{n} = O(n^2)$
prodotto	$O(f) \cdot O(g) = O(f \cdot g)$	$n^2 \cdot \sqrt{n} = O(n^{2.5})$
logaritmo di prodotto/quoziente	$\log(a \cdot b) = \log a + \log b$; $\log(a / b) = \log a - \log b$	$\log(n^c / n^k) = (c - k) \log n = O(\log n)$
potenze e radici	$\sqrt[n]{n} = n^{1/2}$; $n^a \cdot n^b = n^{a+b}$; $(n^a)^b = n^{a \cdot b}$	$n\sqrt{n} = n^{3/2}$; $\sqrt[n]{n^3} = n^{3/2}$

Il procedimento consigliato è sempre lo stesso: **prima semplificare ciascuna funzione** al suo termine dominante senza costanti, **poi** applicare l'operazione richiesta. Si sbaglia molto meno che a moltiplicare i polinomi per intero. Per i prodotti di potenze conviene **scrivere tutto come n elevato a un esponente frazionario** e sommare gli esponenti.

ESERCIZIO

Esercizi T1 e T2 svolti

$T_1 = 5n^2 + 2n - 10$, $T_2 = 5\sqrt{n} + 22$. $G = T_1 + T_2$?

Si semplifica: $T_1 = O(n^2)$, $T_2 = O(\sqrt{n}) = O(n^{1/2})$. Nella somma vince il più grande: **$G = O(n^2)$** . Le risposte da scartare sono $5(n^2 + \sqrt{n}) + 2n + 12$ (vera come uguaglianza, ma non è un ordine di grandezza semplificato) e $n^2\sqrt{n}$ (che sarebbe il prodotto).

Stesse T1 e T2. $G = T_1 \cdot T_2$?

Nel prodotto gli ordini si moltiplicano: $n^2 \cdot n^{1/2} = n^{2 + 1/2} = n^{5/2} = n^{2.5} = n^2\sqrt{n}$. Attenzione a non confonderla con $n^{3/2}$, che sarebbe $n\sqrt{n}$.

$T_1 = 5\sqrt{n^3}$, $T_2 = 5n\sqrt{n^5}$. $G = T_1 \cdot T_2$?

$T_1 = n^{3/2}$. $T_2 = n \cdot n^{5/2} = n^{1 + 5/2} = n^{7/2}$. Il prodotto è $n^{3/2 + 7/2} = n^{10/2} = n^5$. (Le opzioni trappola sono $n^{2.5}$ e $n^3\sqrt{n}$.)

$T_1 = n^c$, $T_2 = n^k$. $G = \log(T_1 / T_2)$?

$T_1 / T_2 = n^{c-k}$, quindi $G = \log(n^{c-k}) = (c-k) \cdot \log n$, e $(c-k)$ è una costante: **$G = O(\log n)$** . La trappola è n^{c-k} , che dimentica il logaritmo; un'altra è $\log(n^k) - \log(n^c)$, che ha i termini invertiti.

$T_1 = O(n \log n \log n)$, $T_2 = O(1.5 n \log n)$. Relazione?

$T_1 = n \log^2 n$. Il fattore comune $n \log n$ si semplifica: resta $\log n$ contro 1,5. Poiché $\log n$ cresce senza limite e 1,5 è costante, **$T_1 > T_2$** .

Leggere la complessità dal codice

Il problema più frequente presenta un frammento di codice e chiede «qual è la complessità di questa funzione?». Il metodo è contare **quante volte viene eseguita l'istruzione più interna**, perché è quella che domina il costo. Le regole di base sono queste. Un'istruzione semplice (assegnamento, confronto, aritmetica, I/O di un valore) costa **O(1)**. Il costo di una **sequenza** è la **somma** dei costi, quindi conta la più costosa. Il costo di un **if else** è il costo della condizione più il **massimo** dei due rami (nel caso peggiore si prende il ramo più caro). Il costo di un **ciclo** è la somma dei costi di tutte le sue iterazioni (condizione più corpo); se il corpo costa sempre uguale, è **numero di iterazioni × costo del corpo**. Il costo di una **funzione** è il costo del suo corpo (più il passaggio dei parametri); per le funzioni ricorsive serve una ricorrenza (questo capitolo).

RICETTA 1: PASSO COSTANTE → LINEARE

Se la variabile del ciclo cresce o decresce di una **quantità costante** ($i++$, $i += 2$, $i -= c$) e deve percorrere una distanza proporzionale a n , il ciclo fa circa n/c iterazioni: **O(n)**. La costante c , il punto di partenza e la presenza di $n/2$ non cambiano la classe: un ciclo `for (i = n/2; i <= n; i++)` fa $n/2 + 1$ iterazioni, ed è O(n).

```
for (int i = 0; i <= n; i += c) { /* O(1) */ } // O(n), con c costante positiva
```

RICETTA 2: PASSO MOLTIPLICATIVO → LOGARITMO

Se la variabile viene **moltiplicata o divisa** per una costante a ogni giro ($i = 2$, $i /= 2$, $i = 2i$, $i /= c$), il ciclo è **O(log n)**. Il motivo è semplice: partendo da 1 e raddoppiando si ottiene 1, 2, 4, 8, 16, ..., 2^k ; per arrivare a n servono k passi con $2^k = n$, cioè $k = \log_2 n$. È il numero di volte che si può dimezzare n prima di arrivare a 1. Con $n = 1024$ un ciclo che dimezza fa 11 iterazioni (1024, 512, ..., 2, 1), e con $n = 65.536$ ne fa 17: moltiplicare n per 64 ha aggiunto solo 6 iterazioni. Se la variabile cresce **elevandosi a potenza** ($i = i^2$, $i = \text{pow}(i, c)$) le iterazioni sono ancora meno: **O(log log n)**.

```
for (int i = 1; i <= n; i *= c) { /* O(1) */ } // O(log n), c costante > 1
for (int i = n; i > 0; i /= c) { /* O(1) */ } // O(log n)
for (int i = 2; i <= n; i = pow(i, c)) { ... } // O(log log n)
```

VERIFICA

```
for (int i = n; i > 0; i /= c) {
    cout << "Hello" << endl;
}
```

O(log n): i viene diviso per la costante c a ogni iterazione.

```
int a = 0, i = N;
while (i > 0) {
    a += i;
    i /= 2;
}
```

O(log N): stessa ricetta, scritta con un `while`. Il valore accumulato in a è irrilevante: conta solo come evolve la variabile che controlla il ciclo.

RICETTA 3: CICLI ANNIDATI INDIPENDENTI → PRODOTTO

Se i cicli annidati sono **indipendenti** (il numero di iterazioni del ciclo interno non dipende dalla variabile di quello esterno), il numero totale di esecuzioni del corpo è il **prodotto** delle iterazioni di ciascun ciclo. Due cicli lineari annidati danno $O(n \cdot n) = O(n^2)$; tre danno $O(n^3)$. Un ciclo lineare che ne contiene uno logaritmico dà $O(n \log n)$. È la regola più usata: si analizza ogni ciclo con le ricette 1 e 2, e si moltiplicano i risultati.

ESERCIZIO

Tre cicli annidati

```
void func(int n) {
    int count = 0;
    for (int i = n/2; i <= n; i++)           // O(n)      (ricetta 1)
        for (int j = 1; j <= n; j = 2*j)     // O(log n)  (ricetta 2)
            for (int k = 1; k <= n; k = k*2) // O(log n)  (ricetta 2)
                count++;
}
```

I tre cicli sono indipendenti, quindi si moltiplica: $n \cdot \log n \cdot \log n = O(n \log^2 n)$. Le opzioni trappola sono $O(n^2)$ (chi tratta tutti i cicli come lineari) e $O(n \log n)$ (chi vede un solo ciclo logaritmico). Verifica sperimentale: con $n = 1024$ il corpo viene eseguito $(513) \cdot 11 \cdot 11 = 62.073$ volte, con $n = 65.536$ viene eseguito 9.470.241 volte. Il rapporto (circa 153) è proprio quello di $n \log^2 n$: $(65.536 \cdot 16^2) / (1.024 \cdot 10^2) \approx 164$, a meno degli arrotondamenti.

```
int i, j, k = 0;
for (i = n/2; i <= n; i++)
    for (j = 2; j <= n; j = j * 2)
        k = k + n/2;
```

Ciclo esterno lineare, interno logaritmico: $O(n \log n)$. L'istruzione $k = k + n/2$ costa $O(1)$, anche se contiene n .

RICETTA 4: CICLI ANNIDATI DIPENDENTI → SOMMA

Se il ciclo interno **dipende** dalla variabile di quello esterno, il prodotto non è più corretto e bisogna **sommare** le iterazioni del ciclo interno per ogni valore dell'esterno. Il caso più comune è il **ciclo triangolare**: `for (i = 0; i < n; i++) for (j = 0; j < i; j++)`. Il corpo viene eseguito $0 + 1 + 2 + \dots + (n - 1)$ volte. Questa è la **serie aritmetica** di Gauss: $1 + 2 + \dots + n = n(n + 1)/2$, quindi il totale è $n(n - 1)/2 = n^2/2 - n/2 = O(n^2)$. Fare «circa la metà» delle iterazioni di due cicli pieni non cambia la classe: la costante $1/2$ sparisce. Con $n = 1024$ il conteggio misurato è 523.776, esattamente $1024 \cdot 1023 / 2$. È la complessità di selection sort e insertion sort.

Il caso opposto è istruttivo. Se il ciclo esterno **raddoppia** e l'interno arriva fino alla variabile esterna, `for (i = 1; i < n; i *= 2) for (j = 0; j < i; j++)`, sembrerebbe $O(n \log n)$ (esterno $\log n$, interno fino a n). Ma sommando: $1 + 2 + 4 + \dots$ fino a n è una **serie geometrica**, che vale meno di $2n$. Il totale è $O(n)$: con $n = 1024$ il conteggio misurato è 1023. Il prodotto delle stime dei singoli cicli dà sempre un limite superiore corretto, ma non sempre il più stretto.

SOMMA	VALORE	ORDINE
$1 + 2 + 3 + \dots + n$ (serie aritmetica)	$n(n + 1)/2$	$\Theta(n^2)$
$1^2 + 2^2 + \dots + n^2$	$n(n + 1)(2n + 1)/6$	$\Theta(n^3)$
$1 + 2 + 4 + \dots + 2^k$ (serie geometrica di ragione 2)	$2^{k+1} - 1$	$\Theta(2^k)$, cioè $\Theta(n)$ se $2^k = n$
$1 + x + x^2 + \dots + x^n$ (con $x \neq 1$)	$(x^{n+1} - 1)/(x - 1)$	$\Theta(x^n)$ se $x > 1$
$1 + 1/2 + 1/3 + \dots + 1/n$ (serie armonica)	$\approx \ln n$	$\Theta(\log n)$

RICETTA 5: ATTENZIONE A BREAK, RETURN E CONDIZIONI

Prima di moltiplicare, controlla che i cicli facciano davvero tutte le iterazioni. Un break o un return incondizionato nel corpo fa uscire subito dal ciclo: un ciclo interno che esegue una sola istruzione e poi break costa $O(1)$, non $O(n)$. Un if che salta il ciclo interno nella maggior parte delle iterazioni può abbassarne il costo. E un controllo iniziale come `if (n == 1) return;` non cambia nulla per n grande.

DA RICORDARE

Due cicli con il break

```
func(int n) {
    if (n == 1) return;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++) {
            cout << "???" ;
            break;           // il ciclo interno fa UNA sola iterazione
        }
}
```

$O(n)$. Il ciclo esterno fa n iterazioni, quello interno ne fa una sola per via del break. Misurato: con $n = 1024$ il corpo viene eseguito 1024 volte, non 1024^2 . La trappola è $O(n^2)$.

La stessa idea compare in una variante dove il break sta dentro un ciclo che stampa "###": la risposta è ancora **$O(n)$** . E se il break fosse dentro un if che non scatta mai, il ciclo tornerebbe a contare per intero: leggi la condizione.

IL METODO COMPLETO

Riassumendo, davanti a un frammento di codice si procede così. **Uno:** individua l'istruzione più interna (il corpo del ciclo più annidato). **Due:** per ogni ciclo, guarda **come cambia la variabile** di controllo: somma o sottrazione di una costante $\rightarrow$ n iterazioni; moltiplicazione o divisione per una costante $\rightarrow \log n$; elevamento a potenza $\rightarrow \log \log n$. **Tre:** controlla **break, return e condizioni** che accorciano i cicli. **Quattro:** se i cicli sono **indipendenti moltiplica**; se un limite dipende da un'altra variabile **somma** (serie aritmetica $\rightarrow n^2$, serie geometrica $\rightarrow n$). **Cinque:** tieni il termine dominante e scarta le costanti.

```
// conta quante volte viene eseguita l'istruzione piu' interna di alcuni cicli tipici
#include <iostream>
using namespace std;
long cicloSemplice(int n) {           // for i da 0 a n-1
    long c = 0;
    for (int i = 0; i < n; i++) c++;
    return c;
}
long cicloDimezza(int n) {           // i dimezzato a ogni giro
    long c = 0;
    for (int i = n; i > 0; i /= 2) c++;
    return c;
}
long dueCicliAnnidati(int n) {       // i e j indipendenti
    long c = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) c++;
    return c;
}
long triangolare(int n) {            // j dipende da i: j < i
    long c = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < i; j++) c++;
}
```

```

    return c;
}
long nLogQuadro(int n) {           // il quiz: i da n/2 a n, j e k raddoppiano
    long c = 0;
    for (int i = n / 2; i <= n; i++)
        for (int j = 1; j <= n; j = 2 * j)
            for (int k = 1; k <= n; k = k * 2) c++;
    return c;
}
long conBreak(int n) {           // il ciclo interno esce subito con break
    long c = 0;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++) { c++; break; }
    return c;
}
long geometrico(int n) {         // i raddoppia, j arriva fino a i
    long c = 0;
    for (int i = 1; i < n; i *= 2)
        for (int j = 0; j < i; j++) c++;
    return c;
}

int main() {
    int valori[] = {16, 1024, 65536};
    for (int n : valori)
        cout << "n=" << n << ": " << cicloSemplice(n) << " " << cicloDimezza(n) << " "
            << dueCicliAnnidati(n) << " " << triangolare(n) << " " << nLogQuadro(n)
            << " " << conBreak(n) << " " << geometrico(n) << endl;
    return 0;
}

```

CICLO	N = 16	N = 1.024	N = 65.536	CLASSE
cicloSemplice	16	1.024	65.536	$O(n)$
cicloDimezza	5	11	17	$O(\log n)$
dueCicliAnnidati	256	1.048.576	4.294.967.296	$O(n^2)$
triangolare	120	523.776	2.147.450.880	$O(n^2)$
nLogQuadro	225	62.073	9.470.241	$O(n \log^2 n)$
conBreak	16	1.024	65.536	$O(n)$
geometrico	15	1.023	65.535	$O(n)$

La tabella riporta i conteggi effettivi prodotti da p5_conta.cpp. Si leggono le classi direttamente dai numeri: quando n viene moltiplicato per 64 (da 1.024 a 65.536), un ciclo lineare fa 64 volte più iterazioni, uno logaritmico ne aggiunge appena 6, uno quadratico ne fa $64^2 = 4.096$ volte di più.

La complessità delle funzioni ricorsive

LE RELAZIONI DI RICORRENZA

Una funzione ricorsiva non ha cicli da contare: il lavoro è distribuito tra le chiamate. Il suo costo si esprime con una **relazione di ricorrenza**, un'equazione che definisce $T(n)$ in funzione del costo su input più piccoli. Si scrive guardando il codice: quante chiamate ricorsive fa la funzione, su quale dimensione, e quanto lavoro fa **oltre** alle chiamate. La funzione che calcola il fattoriale, `return n * fattoriale(n - 1);`, fa una chiamata su $n - 1$ più un lavoro costante: $T(n) = T(n - 1) + c$, con $T(0) = c$.

DEFINIZIONE

Divide et impera

Il paradigma **divide et impera** risolve un problema ricorsivamente in tre passi: **dividi** il problema in sottoproblemi più piccoli dello stesso tipo, **conquista** risolvendoli ricorsivamente, **combina** le loro soluzioni nella soluzione del problema originale.

Molti algoritmi ricorsivi seguono il paradigma **divide et impera** (*divide and conquer*), che ad ogni livello della ricorsione applica tre passi: **dividi** il problema in sottoproblemi più piccoli dello stesso tipo; **conquista**, cioè risolvi i sottoproblemi ricorsivamente (o direttamente, se sono abbastanza piccoli); **combina** le soluzioni dei sottoproblemi nella soluzione del problema originale. Merge sort e quick sort (capitolo 6) sono gli esempi classici.

LE CINQUE RICORRENZE FONDAMENTALI

In pratica basta riconoscere poche ricorrenze ricorrenti e conoscerne la soluzione. Il metodo per risolverle è lo **srotolamento**: si sostituisce la ricorrenza in se stessa finché non si vede lo schema. Per $T(n) = T(n - 1) + n$: $T(n) = T(n - 2) + (n - 1) + n = T(n - 3) + (n - 2) + (n - 1) + n = \dots = 1 + 2 + \dots + n = n(n + 1)/2$, cioè $\Theta(n^2)$. È la ricorrenza dei programmi che «eliminano un elemento alla volta» esaminando ogni volta tutti quelli rimasti.

RICORRENZA	SOLUZIONE	QUANDO SI PRESENTA	ESEMPIO
$T(n) = T(n - 1) + c$	$\Theta(n)$	una chiamata su $n - 1$, lavoro costante	fattoriale, potenza ingenua, stampa ricorsiva di una lista
$T(n) = T(n - 1) + n$	$\Theta(n^2)$	elimina un elemento ed esamina tutti gli altri	selection sort ricorsivo; quick sort nel caso peggiore
$T(n) = T(n/2) + c$	$\Theta(\log n)$	dimezza l'input, lavoro costante	ricerca binaria, potenza veloce
$T(n) = T(n/2) + n$	$\Theta(n)$ (circa $2n$)	dimezza ed esamina ogni elemento	$n + n/2 + n/4 + \dots < 2n$
$T(n) = 2T(n/2) + n$	$\Theta(n \log n)$	due metà più una scansione lineare	merge sort, quick sort nel caso migliore
$T(n) = 2T(n/2) + c$	$\Theta(n)$ (circa $2n$)	due metà, lavoro costante	contare i nodi di un albero bilanciato

La ricorrenza $T(n) = 2T(n/2) + n$, quella di merge sort, si capisce bene con l'**albero delle chiamate** della Figura 5.2. Al livello 0 c'è una chiamata su n elementi che fa n operazioni oltre alle chiamate ricorsive. Al livello 1 ci sono due chiamate su $n/2$ elementi, che insieme fanno $2 \cdot n/2 = n$ operazioni. Al livello 2 quattro chiamate su $n/4$: ancora n . Ogni livello costa n , e i livelli sono tanti quante le volte che si può dimezzare n prima di arrivare a 1, cioè $\log n$. Il totale è $n \cdot \log n$.

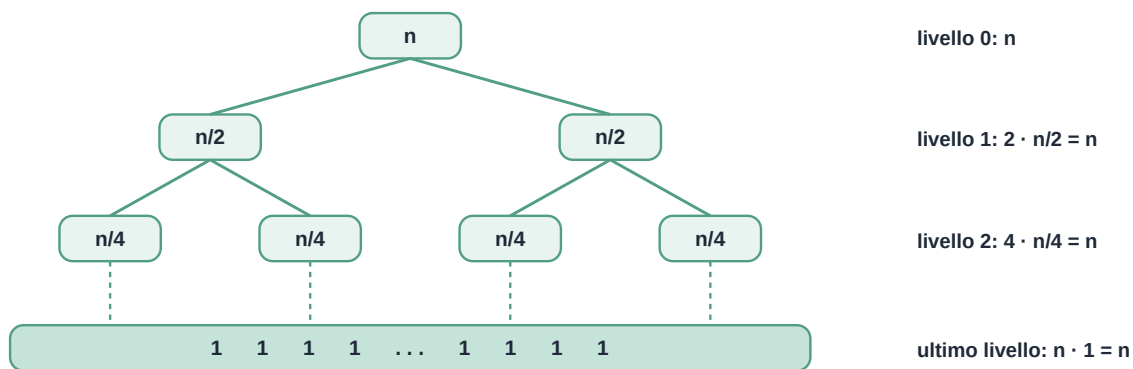


Figura 5.2. Albero delle chiamate di $T(n) = 2T(n/2) + n$: $\log n$ livelli, ciascuno di costo n

UN ESEMPIO: DUE MODI DI CALCOLARE UNA POTENZA

Il calcolo di x^n mostra quanto conti il modo di dividere il problema. La versione ingenua usa $x^n = x \cdot x^{n-1}$: una chiamata su $n-1$, quindi $T(n) = T(n-1) + c = \mathbf{O(n)}$, sia in tempo sia in spazio (ogni chiamata occupa un record di attivazione sullo stack). La versione veloce osserva che $x^n = (x^{n/2})^2$ se n è pari e $x \cdot (x^{(n-1)/2})^2$ se è dispari: **una sola** chiamata su $n/2$, quindi $T(n) = T(n/2) + c = \mathbf{O(\log n)}$. È fondamentale calcolare y una volta e poi fare $y * y$: scrivere $\text{potenza}(x, n/2) * \text{potenza}(x, n/2)$ farebbe due chiamate, $T(n) = 2T(n/2) + c$, e si tornerebbe a $O(n)$.

```
#include <iostream>
using namespace std;

long chiamate = 0;      // solo per contare le chiamate (variabile globale di prova)

// versione 1:  $x^n = x * x^{(n-1)}$        $T(n) = T(n-1) + c \rightarrow O(n)$ 
double potenzaLenta(double x, int n) {
    chiamate++;
    if (n == 0) return 1;           // caso base
    return x * potenzaLenta(x, n - 1); // una chiamata su n-1
}

// versione 2:  $x^n = (x^{(n/2)})^2$        $T(n) = T(n/2) + c \rightarrow O(\log n)$ 
double potenzaVeloce(double x, int n) {
    chiamate++;
    if (n == 0) return 1;
    double y = potenzaVeloce(x, n / 2); // UNA sola chiamata su n/2
    if (n % 2 == 0) return y * y;       // n pari:  $x^n = y * y$ 
    else return x * y * y;              // n dispari:  $x^n = x * y * y$ 
}

int main() {
    double r = potenzaLenta(2, 30);
    cout << r << " in " << chiamate << " chiamate" << endl; // 31 chiamate
    chiamate = 0;
    r = potenzaVeloce(2, 30);
    cout << r << " in " << chiamate << " chiamate" << endl; // 6 chiamate
    return 0;
}
```

Per x^{30} la versione lenta fa 31 chiamate (da $n = 30$ fino a $n = 0$), quella veloce 6 ($n = 30, 15, 7, 3, 1, 0$). La differenza diventa abissale per n grandi: con $n =$ un miliardo, 30 chiamate contro un miliardo. (La variabile globale `chiamate` serve solo a misurare; nei programmi veri le variabili globali vanno evitate.)

VERIFICA

Altre domande sulle funzioni ricorsive

Data la funzione ricorsiva che calcola la moltiplicazione $N \times M$ mediante somme successive, qual è la sua complessità?

La funzione calcola $N \cdot M = N + N \cdot (M - 1)$, cioè fa una chiamata su $M - 1$ e una somma: $T(M) = T(M - 1) + c$, quindi **$O(M)$** . Dipende dal parametro che viene decrementato, non dal prodotto.

Una funzione stampa ricorsivamente gli elementi di una lista concatenata: `boo(testa) { if (testa == nullptr) return; ... boo(testa->succ); }`. Complessità?

$O(n)$: una chiamata per nodo, $T(n) = T(n - 1) + c$. Anche lo spazio è $O(n)$, per i record di attivazione sullo stack.

Un caso di studio: il segmento di somma massima

Chiude il capitolo un problema che mostra come lo stesso compito possa avere algoritmi di complessità molto diverse. Dato un array S di n interi (con almeno un positivo), un **segmento** è una sequenza di elementi consecutivi $S[i..j]$; si vuole la **somma massima** di un segmento. Per $S = 2, -5, 3, 4, -1, 6, -8, 2$ la risposta è 12, data dal segmento 3, 4, -1, 6.

```
#include <iostream>
using namespace std;

// soluzione 1: tutti i segmenti, somma ricalcolata da capo          Theta(n^3)
int sommaMax1(int S[], int n) {
    int massimo = 0;
    for (int i = 0; i < n; i++)
        for (int j = i; j < n; j++) {
            int somma = 0;
            for (int k = i; k <= j; k++) somma += S[k];
            if (somma > massimo) massimo = somma;
        }
    return massimo;
}

// soluzione 2: somma(i..j) = somma(i..j-1) + S[j]                Theta(n^2)
int sommaMax2(int S[], int n) {
    int massimo = 0;
    for (int i = 0; i < n; i++) {
        int somma = 0;
        for (int j = i; j < n; j++) {
            somma += S[j];
            if (somma > massimo) massimo = somma;
        }
    }
    return massimo;
}

// soluzione 3: una sola passata (un prefisso negativo non conviene mai)  Theta(n)
int sommaMax3(int S[], int n) {
    int massimo = 0, somma = 0;
    for (int j = 0; j < n; j++) {
        if (somma > 0) somma += S[j];    // conviene estendere il segmento corrente
        else          somma = S[j];    // meglio ripartire da S[j]
        if (somma > massimo) massimo = somma;
    }
}
```

```

    return massimo;
}

int main() {
    int S[] = {2, -5, 3, 4, -1, 6, -8, 2};
    int n = sizeof(S) / sizeof(S[0]);
    cout << sommaMax1(S, n) << " " << sommaMax2(S, n) << " " << sommaMax3(S, n) << endl;
    return 0;    // 12 12 12: il segmento 3 4 -1 6
}

```

La **prima soluzione** prova tutte le coppie (i, j) e ricalcola ogni volta la somma da capo con un terzo ciclo: tre cicli annidati dipendenti. Il conteggio esatto è la somma, per ogni i , di $1 + 2 + \dots + (n - i)$, che si dimostra essere $\Theta(n^3)$. La **seconda** osserva che $\text{somma}(i..j) = \text{somma}(i..j - 1) + S[j]$: non serve ripartire da capo, e il terzo ciclo sparisce. Restano due cicli dipendenti, un ciclo triangolare: $\Theta(n^2)$. La **terza** sfrutta la struttura del problema: un segmento di somma massima non comincia mai con un prefisso di somma negativa (toglierlo aumenterebbe la somma). Quindi si scorre l'array una volta sola, accumulando; quando la somma accumulata diventa ≤ 0 conviene ricominciare dall'elemento corrente. Un solo ciclo: $\Theta(n)$, ed è **ottimale**, perché qualunque algoritmo deve almeno leggere tutti gli n elementi. Una **quarta** soluzione divide et impera (massimo tra la metà sinistra, la metà destra e il miglior segmento a cavallo del centro) ha ricorrenza $T(n) = 2T(n/2) + n$, cioè $\Theta(n \log n)$.

SOLUZIONE	IDEA	COMPLESSITÀ	N = 10.000 (CIRCA)
1	tutti i segmenti, somma ricalcolata	$\Theta(n^3)$	10^{12} passi: un quarto d'ora circa
2	somma aggiornata incrementalmente	$\Theta(n^2)$	10^8 passi: una frazione di secondo
4	divide et impera	$\Theta(n \log n)$	$1,3 \cdot 10^5$ passi
3	una passata, si scarta il prefisso negativo	$\Theta(n)$	10^4 passi

Riepilogo: le complessità da sapere

Questa tabella raccoglie le complessità da conoscere. Ognuna è spiegata nella parte indicata; qui servono per il ripasso finale.

DA RICORDARE

Il ragionamento conta più della memoria

Quasi tutte le righe della tabella si ricavano in pochi secondi ragionando sulla struttura: in un array si salta direttamente alla cella, in una lista no; con il puntatore alla coda si aggiunge in fondo senza scorrere, ma per togliere l'ultimo serve il penultimo, che in una lista semplice si raggiunge solo dalla testa; in un albero si scende di un livello alla volta, quindi il costo è l'altezza. Se un dato non torna in mente, conviene ricostruirlo così.

OPERAZIONE	COMPLESSITÀ	CAPITOLO
accesso a $v[i]$ in un array	$O(1)$	1
ricerca sequenziale (array o lista)	$O(n)$; caso medio $(n + 1)/2$ confronti	6
ricerca binaria (array ordinato)	$O(\log n)$	6
inserimento/rimozione in testa a una lista	$O(1)$	7
rimozione dell'ultimo elemento di una lista semplice (anche con puntatore alla coda)	$O(n)$: serve il penultimo	7
enqueue con puntatori a testa e coda	$O(1)$	7

OPERAZIONE	COMPLESSITÀ	CAPITOLO
enqueue con il solo puntatore alla testa	$O(n)$: si scorre fino all'ultimo	7
push e pop di una pila	$O(1)$	7
inserimento in array dinamico che raddoppia	$O(n)$ la singola riallocazione, $O(1)$ in media	4
selection sort	$O(n^2)$ in ogni caso	6
insertion sort	$O(n)$ migliore, $O(n^2)$ medio e peggiore	6
bubble sort con flag (sentinella)	$O(n)$ migliore, $O(n^2)$ medio e peggiore	6
merge sort	$O(n \log n)$ in ogni caso	6
quick sort	$O(n \log n)$ migliore e medio, $O(n^2)$ peggiore	6
ricerca, inserimento, cancellazione in un BST	$O(h)$: $O(\log n)$ se bilanciato, $O(n)$ se degenerare	8
visita pre/in/postorder di un albero	$\Theta(n)$	8
BFS e DFS con liste di adiacenza	$O(V + E)$, lineare in nodi + archi	9
algoritmi su grafi con matrice di adiacenza	$O(V^2)$	9

Ricorsione, ricerca e ordinamento

Ricerca e ordinamento sono il terreno classico su cui applicare l'analisi della complessità: dagli algoritmi elementari (insertion, selection e bubble sort) ai due algoritmi ricorsivi basati sul paradigma divide et impera, merge sort e quick sort.

La ricorsione, in breve

Una funzione è **ricorsiva** se richiama se stessa. Funziona se ogni chiamata lavora su un problema **più piccolo** dello stesso tipo e se esiste un **caso base** (il «tappo») risolto senza ricorsione: senza caso base, o se la dimensione non diminuisce, le chiamate non finiscono mai e lo stack si esaurisce (*stack overflow*). Ogni chiamata ha il proprio **record di attivazione** sullo stack, con i propri parametri e le proprie variabili locali: è per questo che la versione ricorsiva del fattoriale, `if (n == 0) return 1; return n * fattoriale(n - 1);`, funziona anche se tutte le chiamate usano un parametro di nome `n`.

Vale la pena ricordare che **per ogni funzione ricorsiva ne esiste una iterativa equivalente**, e che di solito la versione iterativa è più efficiente in tempo e in occupazione di stack (non paga il costo delle chiamate). La ricorsione vale la pena quando rende naturale un problema che ha già una struttura ricorsiva: le strutture dati ricorsive come le liste e soprattutto gli **alberi** (capitolo 8), e gli algoritmi **divide et impera** come merge sort e quick sort, che con la ricorsione si scrivono in poche righe e senza ricorsione diventerebbero complicati. Il costo di una funzione ricorsiva si calcola con una relazione di ricorrenza (capitolo 5).

DA RICORDARE

Prima stampa o prima chiamata?

Nelle funzioni ricorsive su una lista, la posizione della stampa rispetto alla chiamata decide l'ordine dell'output. Con `cout << testa->valore; boo(testa->succ);` si stampa il nodo **prima** di scendere: la lista esce **dalla testa alla coda**. Con `boo(testa->succ); cout << testa->valore;` la stampa avviene **al ritorno** dalle chiamate, quando la ricorsione risale: l'ultimo nodo viene stampato per primo e la lista esce **in ordine inverso**. È un caso frequente (capitolo 7) ed è lo stesso meccanismo delle visite preorder e postorder degli alberi.

La ricerca

La **ricerca sequenziale** scorre gli elementi uno dopo l'altro finché trova quello cercato o finisce la sequenza. È l'unica possibile quando gli elementi **non sono ordinati**, e anche l'unica possibile su una **lista concatenata**, dove si può solo passare da un nodo al successivo. Costa 1 confronto nel caso migliore, n nel peggiore e $(n + 1)/2$ in media (capitolo 5): $O(n)$. Con una piccola modifica (non fermarsi al primo e incrementare un contatore) conta quante volte un valore compare.

Se l'array è **ordinato** si può fare molto meglio con la **ricerca binaria** (o dicotomica). Si confronta il valore cercato con l'elemento **centrale**: se è uguale si è finito; se è minore, il valore può stare solo nella metà sinistra; se è maggiore, solo nella destra. A ogni passo **si scarta metà degli elementi rimasti**, quindi i confronti sono al più circa $\log_2 n$: per un milione di elementi bastano 20 confronti invece di un milione. È come cercare una parola nel dizionario aprendolo a metà invece che dalla prima pagina. La ricerca binaria richiede l'**accesso diretto** all'elemento centrale in tempo costante, quindi funziona sugli array ma **non sulle liste**.

```

#include <iostream>
using namespace std;

// ricerca sequenziale: restituisce l'indice di x, oppure -1          0(n)
int ricercaSequenziale(int v[], int n, int x) {
    for (int i = 0; i < n; i++)
        if (v[i] == x) return i;
    return -1;
}

// ricerca binaria (dicotomica) su array ORDINATO, iterativa      0(log n)
int ricercaBinaria(int v[], int n, int x) {
    int sx = 0, dx = n - 1;
    while (sx <= dx) {
        int m = (sx + dx) / 2;           // elemento centrale
        if (v[m] == x) return m;         // trovato
        if (x < v[m]) dx = m - 1;        // x puo' stare solo a sinistra
        else          sx = m + 1;        // x puo' stare solo a destra
    }
    return -1;                          // intervallo vuoto: non c'e'
}

// stessa ricerca in versione ricorsiva: T(n) = T(n/2) + c
int ricercaBinariaRic(int v[], int sx, int dx, int x) {
    if (sx > dx) return -1;              // caso base: non trovato
    int m = (sx + dx) / 2;
    if (v[m] == x) return m;             // caso base: trovato
    if (x < v[m]) return ricercaBinariaRic(v, sx, m - 1, x);
    return ricercaBinariaRic(v, m + 1, dx, x);
}

int main() {
    int v[] = {3, 8, 12, 15, 21, 30, 42, 57};
    cout << ricercaSequenziale(v, 8, 21) << " " << ricercaBinaria(v, 8, 21) << " "
         << ricercaBinariaRic(v, 0, 7, 21) << " " << ricercaBinaria(v, 8, 5) << endl;
    return 0;    // 4 4 4 -1
}

```

RICERCA	CASO MIGLIORE	CASO MEDIO	CASO PEGGIORE	REQUISITI
sequenziale	$O(1)$	$O(n)$	$O(n)$	nessuno
binaria	$O(1)$	$O(\log n)$	$O(\log n)$	array ordinato, accesso diretto

Gli ordinamenti elementari

Seguiamo i tre algoritmi elementari sullo stesso array, **5 2 4 6 1 3**, guardando come cambia dopo ogni passata. Le tracce sono state prodotte eseguendo il codice.

INSERTION SORT

L'**ordinamento per inserimento** funziona come quando si riordinano le carte da gioco in mano: si prende una carta alla volta e la si **inserisce al posto giusto** tra quelle già ordinate. All'inizio la parte ordinata è il solo primo elemento. A ogni passo i si prende $v[i]$, lo si salva in una variabile d'appoggio, si spostano di una posizione a destra tutti gli elementi della parte ordinata **maggiori** di lui, e lo si deposita nel buco rimasto.

PASSO	ELEMENTO INSERITO	ARRAY DOPO IL PASSO
inizio		5 2 4 6 1 3
$i = 1$	2	2 5 4 6 1 3
$i = 2$	4	2 4 5 6 1 3
$i = 3$	6	2 4 5 6 1 3 (6 è già al suo posto: un solo confronto)
$i = 4$	1	1 2 4 5 6 3 (1 scorre fino in testa)
$i = 5$	3	1 2 3 4 5 6

Il costo dipende dall'input. Nel **caso migliore**, array già ordinato, ogni elemento fa un solo confronto e non si sposta: $O(n)$. Nel **caso peggiore**, array ordinato al contrario, l'elemento di posizione i scorre fino in testa con i confronti e spostamenti: $1 + 2 + \dots + (n - 1)$, cioè $O(n^2)$. Anche nel caso medio è $O(n^2)$. Si può **migliorare** cercando la posizione d'inserimento con la **ricerca binaria** nella parte già ordinata: i confronti scendono a $O(n \log n)$ in totale, ma gli **spostamenti** restano $O(n^2)$ nel caso medio e peggiore, quindi la complessità complessiva non cambia. L'insertion sort è molto efficiente su array **quasi ordinati**.

SELECTION SORT

L'**ordinamento per selezione** cerca il **minimo** della parte non ancora ordinata e lo **scambia** con il primo elemento di quella parte. Dopo il passo i , le prime $i + 1$ posizioni contengono i più piccoli in ordine.

PASSO	MINIMO TROVATO	ARRAY DOPO LO SCAMBIO
$i = 0$	1 (in posizione 4)	1 2 4 6 5 3
$i = 1$	2 (già al suo posto)	1 2 4 6 5 3
$i = 2$	3 (in posizione 5)	1 2 3 6 5 4
$i = 3$	4 (in posizione 5)	1 2 3 4 5 6
$i = 4$	5 (già al suo posto)	1 2 3 4 5 6

Per trovare il minimo bisogna comunque guardare **tutti** gli elementi rimasti, anche se l'array è già ordinato: i confronti sono sempre $(n - 1) + (n - 2) + \dots + 1 = n(n - 1)/2$. Il selection sort costa quindi $O(n^2)$ in **qualsiasi caso**, migliore compreso. In compenso fa al più $n - 1$ scambi, il minimo possibile. **Non esiste un modo di migliorarlo** con gli strumenti visti: la ricerca binaria non aiuta (la parte da esaminare non è ordinata) e il pivot riguarda il quick sort.

BUBBLE SORT

Il **bubble sort** scorre l'array confrontando gli elementi **adiacenti** e li scambia se sono nell'ordine sbagliato. Alla fine della prima passata il **massimo** è «risalito» come una bolla fino all'ultima posizione; alla fine della seconda il secondo più grande è in penultima posizione, e così via. Dopo $n - 1$ passate l'array è ordinato, e a ogni passata si può evitare di confrontare gli elementi già sistemati in fondo.

PASSATA	ARRAY AL TERMINE DELLA PASSATA
1	2 4 5 1 3 6
2	2 4 1 3 5 6
3	2 1 3 4 5 6
4	1 2 3 4 5 6
5	nessuno scambio: con il flag l'algoritmo si ferma

La versione base ripete sempre $n - 1$ passate: **$O(n^2)$ in ogni caso**. La versione migliorata usa un **flag** (detto anche **sentinella**) che registra se nella passata è avvenuto almeno uno scambio: se una passata intera non scambia nulla l'array è già ordinato e ci si ferma. Nel caso migliore, array già ordinato, basta una passata: **$O(n)$** . Nel caso medio e peggiore resta $O(n^2)$.

```
#include <iostream>
using namespace std;

void scambia(int& a, int& b) { int t = a; a = b; b = t; }
// INSERTION SORT: inserisce v[i] al posto giusto nella parte già ordinata v[0..i-1]
void insertionSort(int v[], int n) {
    for (int i = 1; i < n; i++) {
        int appoggio = v[i];           // elemento da inserire
        int j = i - 1;
        while (j >= 0 && v[j] > appoggio) { // prima j >= 0, poi v[j]!
            v[j + 1] = v[j];           // sposta a destra i più grandi
            j--;
        }
        v[j + 1] = appoggio;           // inserisce nel buco lasciato libero
    }
}

// SELECTION SORT: cerca il minimo della parte non ordinata e lo mette in testa
void selectionSort(int v[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int posMin = i;
        for (int j = i + 1; j < n; j++)
            if (v[j] < v[posMin]) posMin = j;
        scambia(v[i], v[posMin]);
    }
}

// BUBBLE SORT con flag: si ferma quando una passata non fa scambi
void bubbleSort(int v[], int n) {
    bool scambiato = true;
    for (int passata = 0; passata < n - 1 && scambiato; passata++) {
        scambiato = false;
        for (int j = 0; j < n - 1 - passata; j++) // gli ultimi sono già a posto
            if (v[j] > v[j + 1]) {
                scambia(v[j], v[j + 1]);
            }
    }
}
```

```

        scambiato = true;
    }
}

void stampa(int v[], int n) {
    for (int i = 0; i < n; i++) cout << v[i] << " ";
    cout << endl;
}

int main() {
    int a[] = {5, 2, 4, 6, 1, 3}, b[] = {5, 2, 4, 6, 1, 3}, c[] = {5, 2, 4, 6, 1, 3};
    insertionSort(a, 6); selectionSort(b, 6); bubbleSort(c, 6);
    stampa(a, 6); stampa(b, 6); stampa(c, 6);
    return 0;
}

```

DA RICORDARE

Attenzione agli indici senza segno

Si immagini di scrivere questi algoritmi usando `size_t` per gli indici. `size_t` è **senza segno**: non può valere `-1`. Nell'insertion sort una condizione come `(array[j] > appoggio) && (j >= 0)` avrebbe due problemi. Il primo è che `j >= 0` è **sempre vero** per un tipo senza segno: quando `j` vale 0 e viene decrementato, invece di diventare `-1` diventa il numero enorme 18.446.744.073.709.551.615. Il secondo è l'ordine delle condizioni: `array[j]` viene letto **prima** di controllare `j`. Così, quando il nuovo elemento è il più piccolo visto finora, il programma legge fuori dall'array. Nella versione corretta `j` è un `int` e il controllo viene prima, `j >= 0 && v[j] > appoggio`: grazie al cortocircuito di `&&`, se `j` è `-1` la lettura non avviene. Nel quick sort lo stesso tipo fa sì che `i = p - 1` con `p = 0` e la chiamata su `q - 1` producano valori enormi invece di `-1`. Morale: **per gli indici che possono diventare negativi usa `int`**.

VERIFICA

Domande sugli ordinamenti elementari

Come è possibile migliorare l'efficienza dell'insertion sort? a) scegliendo opportunamente il pivot; b) sfruttando la ricerca binaria; c) mediante una selezione random dell'elemento iniziale; d) nessuna delle precedenti.

Risposta **b)**: la posizione d'inserimento si trova con la ricerca binaria nella parte già ordinata, riducendo i confronti. Le opzioni a) e c) riguardano il quick sort.

Come è possibile migliorare l'efficienza del selection sort? (stesse opzioni)

Risposta **d) nessuna delle precedenti**: il selection sort deve comunque esaminare tutti gli elementi rimasti per trovare il minimo, ed è $O(n^2)$ sempre.

Dopo i primi $n - 1$ confronti, cosa si può dire dell'array? (il minimo è al suo posto / il massimo è al suo posto / l'array è ordinato / nessuna)

Dipende dall'algoritmo, ed è questo il trucco. Nel **selection sort** i primi $n - 1$ confronti servono a trovare il minimo, che poi va in testa: **il minimo è al suo posto**. Nel **bubble sort** i primi $n - 1$ confronti sono la prima passata: **il massimo è al suo posto** (in fondo). Nell'**insertion sort** i primi $n - 1$ confronti non hanno un significato fisso (dipendono da quanto scorre ogni elemento): **nessuna delle precedenti**, salvo che l'array sia già ordinato.

[47, 3, 32, 21, 56, 92] dopo due confronti diventa [3, 21, 47, 32, 56, 92]. Quale algoritmo è stato usato? a) insertion sort b) selection sort c) quick sort d) nessuna

Risposta **d) nessuna**. Dopo due confronti l'insertion sort darebbe 3 47 32 21 ..., il bubble sort 3 32 47 21 ..., il selection sort non avrebbe ancora scambiato nulla (sta ancora cercando il minimo). Nemmeno dopo **due passate** il selection sort produce quella sequenza (darebbe 3 21 32 47 ...).

VERIFICA

Quale tra i seguenti algoritmi di ordinamento ha complessità asintotica lineare nel caso ottimo?

Il **bubble sort con sentinella** (flag). Ma è lineare nel caso ottimo anche l'**insertion sort**, con un array già ordinato: se tra le opzioni c'è anche lui, sono vere entrambe. Non lo sono il selection sort ($O(n^2)$ sempre), il merge sort ($O(n \log n)$ sempre) né il quick sort ($O(n \log n)$ nel caso migliore).

Merge sort

Il **merge sort** (ordinamento per fusione) applica il paradigma divide et impera. **Dividi**: l'array di n elementi viene diviso in due metà di $n/2$ elementi. **Conquista**: le due metà vengono ordinate ricorsivamente, con lo stesso algoritmo, finché si arriva a sottoarray di un solo elemento, che sono ordinati per definizione. **Combina**: la procedura **merge** fonde due sottoarray già ordinati in un unico array ordinato.

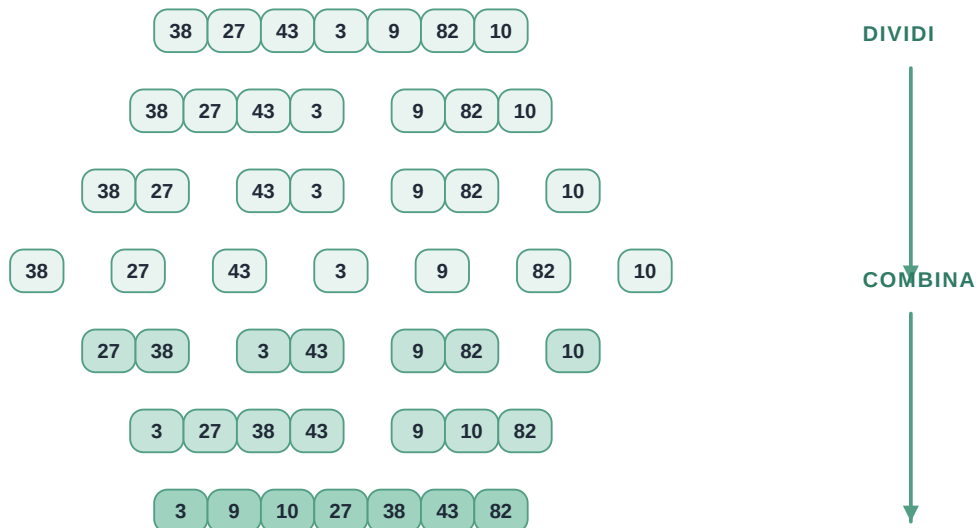


Figura 6.1. Merge sort su 38 27 43 3 9 82 10: in alto le divisioni, in basso le fusioni

Il cuore dell'algoritmo è la **fusione**, che si capisce pensando a due mazzi di carte già ordinati, a faccia in su: si confrontano le due carte in cima, si prende la più piccola e la si mette nel mazzo finale, e si ripete finché uno dei due mazzi finisce; poi si accodano le carte rimaste dell'altro. Ogni confronto sposta definitivamente un elemento, quindi fondere due sequenze di lunghezza totale n costa $O(n)$. Serve però un **array di appoggio** di n elementi: il merge sort non ordina «sul posto» e usa $O(n)$ di memoria aggiuntiva.

```
#include <iostream>
using namespace std;

// fonde le due meta' gia' ordinate v[p..q] e v[q+1..r]
void merge(int v[], int p, int q, int r) {
    int n = r - p + 1;
    int* tmp = new int[n]; // array di appoggio
    int i = p, j = q + 1, k = 0;
    while (i <= q && j <= r) { // finche' entrambe le meta' hanno elementi
        if (v[i] <= v[j]) tmp[k++] = v[i++]; // prende il piu' piccolo dei due
        else tmp[k++] = v[j++];
    }
    while (i <= q) tmp[k++] = v[i++]; // copia l'eventuale resto della sinistra
    while (j <= r) tmp[k++] = v[j++]; // ... o della destra
    for (k = 0; k < n; k++) v[p + k] = tmp[k]; // ricopia in v
    delete[] tmp;
}

void mergeSort(int v[], int p, int r) {
    if (p >= r) return; // 0 o 1 elementi: gia' ordinato
    int q = (p + r) / 2; // DIVIDI a meta'
    mergeSort(v, p, q); // CONQUISTA: ordina la sinistra
    mergeSort(v, q + 1, r); // ordina la destra
    merge(v, p, q, r); // COMBINA: fonde le due meta'
}

int main() {
    int v[] = {38, 27, 43, 3, 9, 82, 10};
    mergeSort(v, 0, 6);
    for (int i = 0; i < 7; i++) cout << v[i] << " ";
    cout << endl;
    return 0;
}
```

```
}
```

La ricorrenza è $T(n) = 2T(n/2) + n$: due chiamate su metà dell'input più la fusione lineare. Come visto nella Figura 5.2, la soluzione è $\Theta(n \log n)$: $\log n$ livelli di divisione, e a ogni livello le fusioni toccano complessivamente tutti gli n elementi. Poiché il merge sort divide **sempre a metà**, indipendentemente dai valori, la complessità è la **stessa in tutti e tre i casi**. Essendo $O(n \log n)$ e il problema dell'ordinamento $\Omega(n \log n)$, il merge sort è **ottimale**.

Quick sort

Il **quick sort** è anch'esso divide et impera, ma lavora al contrario del merge sort: il lavoro si fa **prima** delle chiamate ricorsive, nella divisione, e la combinazione non serve. Si sceglie un elemento detto **pivot** e si **partiziona** l'array in tre zone: a sinistra tutti gli elementi **minori o uguali** al pivot, al centro il pivot, a destra tutti quelli **maggiori**. A questo punto il pivot è nella sua **posizione definitiva**, e ogni elemento della zona sinistra è minore di ogni elemento della destra. Si ordinano ricorsivamente le due zone e si è finito: non c'è nulla da fondere.

La procedura **partition** qui usata (detta di Lomuto) prende come pivot l'**ultimo** elemento $v[r]$. Mantiene un indice i che delimita la zona dei «piccoli» ($v[p..i]$, inizialmente vuota) e scorre gli altri elementi con j : ogni volta che $v[j]$ è minore o uguale al pivot, allarga la zona dei piccoli e ci porta dentro $v[j]$ con uno scambio. Alla fine scambia il pivot con il primo elemento dopo la zona dei piccoli e ne restituisce la posizione.

J	$V[J] \leq 21?$	ARRAY DOPO IL PASSO (PIVOT 21)
0	10 sì: $i = 0$	10 32 11 34 15 7 21
1	32 no	10 32 11 34 15 7 21
2	11 sì: $i = 1$, scambio 32 e 11	10 11 32 34 15 7 21
3	34 no	10 11 32 34 15 7 21
4	15 sì: $i = 2$, scambio 32 e 15	10 11 15 34 32 7 21
5	7 sì: $i = 3$, scambio 34 e 7	10 11 15 7 32 34 21
fine	pivot in posizione $i + 1 = 4$	10 11 15 7 21 34 32

```
#include <iostream>
using namespace std;

void scambia(int& a, int& b) { int t = a; a = b; b = t; }
// partition: usa v[r] come pivot e restituisce la sua posizione finale
int partition(int v[], int p, int r) {
    int pivot = v[r];
    int i = p - 1; // v[p..i] contiene gli elementi <= pivot
    for (int j = p; j < r; j++)
        if (v[j] <= pivot) {
            i++;
            scambia(v[i], v[j]); // porta v[j] nella zona dei "piccoli"
        }
    scambia(v[i + 1], v[r]); // il pivot va subito dopo i piccoli
    return i + 1;
}

void quickSort(int v[], int p, int r) {
    if (p >= r) return; // 0 o 1 elementi
```

```

    int q = partition(v, p, r);          // ora v[q] e' al suo posto definitivo
    quickSort(v, p, q - 1);             // ordina gli elementi <= pivot
    quickSort(v, q + 1, r);             // ordina gli elementi > pivot
}
int main() {
    int v[] = {10, 32, 11, 34, 15, 7, 21};
    quickSort(v, 0, 6);
    for (int i = 0; i < 7; i++) cout << v[i] << " ";
    cout << endl;
    return 0;
}

```

La complessità dipende da **quanto è bilanciata la partizione**. Nel **caso migliore** il pivot divide l'array in due metà di circa $n/2$ elementi: la ricorrenza è $T(n) = 2T(n/2) + n$, come per il merge sort, e il costo è **$O(n \log n)$** . Nel **caso peggiore** il pivot è ogni volta il minimo o il massimo, e le due zone hanno 0 e $n - 1$ elementi: $T(n) = T(n - 1) + n$, cioè **$O(n^2)$** . Con il pivot preso in ultima posizione questo accade proprio quando l'array è **già ordinato** (o ordinato al contrario). Nel **caso medio** il quick sort è **$O(n \log n)$** : si dimostra che anche una partizione sbilanciata, purché in proporzione costante (per esempio 9 a 1), produce una ricorsione di profondità $O(\log n)$ con costo $O(n)$ per livello.

Per evitare il caso peggiore si **sceglie opportunamente il pivot**: a caso (*pivot random*), oppure come mediano tra il primo, il centrale e l'ultimo elemento. Nella pratica il quick sort è spesso il più veloce degli algoritmi di ordinamento, perché ordina **sul posto**, senza array di appoggio, e ha costanti molto piccole; il merge sort però garantisce $O(n \log n)$ anche nel caso peggiore.

VERIFICA

Quali di questi algoritmi di ordinamento sono basati sul paradigma divide et impera?

Merge sort e quick sort. Insertion, selection e bubble sort sono iterativi e non dividono il problema in sottoproblemi.

Che differenza c'è tra merge sort e quick sort?

Entrambi sono divide et impera, ma il merge sort divide in modo banale (a metà) e fa il lavoro nella **combinazione** (la fusione), mentre il quick sort fa il lavoro nella **divisione** (la partizione attorno al pivot) e non ha bisogno di combinare. Il merge sort è $O(n \log n)$ in tutti i casi ma usa un array di appoggio; il quick sort ordina sul posto, è $O(n \log n)$ nel caso medio ma $O(n^2)$ nel peggiore, quando il pivot è sempre il minimo o il massimo.

Riepilogo degli algoritmi

ALGORITMO	MIGLIORE	MEDIO	PEGGIORE	NOTE
ricerca sequenziale	$O(1)$	$O(n)$	$O(n)$	unica possibile su liste e dati non ordinati
ricerca binaria	$O(1)$	$O(\log n)$	$O(\log n)$	richiede array ordinato
insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	migliorabile nei confronti con la ricerca binaria
selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	al più $n - 1$ scambi; non migliorabile
bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	versione base
bubble sort con flag	$O(n)$	$O(n^2)$	$O(n^2)$	si ferma se una passata non scambia
merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	divide et impera; array di appoggio $O(n)$
quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	divide et impera; peggiore se il pivot è sempre min o max

DA RICORDARE

Ordinare oggetti invece di numeri

Capita spesso di dover ordinare **oggetti** (studenti per media, venditori per prezzo totale, «in ordine crescente o decrescente a scelta dell'utente»). Gli algoritmi restano identici: cambia solo il **confronto**. Se l'array contiene puntatori a oggetti si confrontano i campi tramite i `get`, per esempio `v[j] -> getMedia()` > `v[j+1] -> getMedia()`, e si scambiano i **puntatori**, non gli oggetti. Per l'ordine decrescente basta invertire il confronto. In alternativa si definisce `operator<` nella classe (capitolo 4) e si scrive un ordinamento template che funziona per qualunque tipo. Il bubble sort è il più semplice da scrivere senza errori: con n piccolo, la sua complessità quadratica non è un problema.

Liste, pile e code

Liste concatenate, pile e code diventano classi che incapsulano i propri nodi, offrono un'interfaccia di metodi e liberano la memoria nel distruttore; con i template possono contenere elementi di qualunque tipo.

Dagli array alle liste

Un insieme matematico è immutabile; gli insiemi manipolati dai programmi invece **crescono, si riducono e cambiano** durante l'esecuzione: si chiamano **insiemi dinamici**. Una **sequenza lineare** è un insieme finito di elementi disposti uno dopo l'altro, in cui ogni elemento ha una posizione. I due modi fondamentali di realizzarla corrispondono a due modi diversi di usare la memoria.

In un **array** gli elementi occupano celle **contigue**: noto l'indirizzo del primo, l'indirizzo dell'elemento di posizione i si calcola con una moltiplicazione, quindi l'**accesso è diretto** e costa $O(1)$, come aprire un libro direttamente a pagina 200. Il prezzo è la rigidità: la dimensione va decisa all'allocazione, e inserire o togliere un elemento in mezzo richiede di spostare tutti quelli che seguono, $O(n)$. In una **lista concatenata** gli elementi stanno in punti **qualsiasi** della memoria, e ciascuno memorizza, oltre al proprio valore, l'**indirizzo del successivo**. L'**accesso è sequenziale**: per raggiungere l'elemento di posizione i bisogna partire dalla testa e seguire i collegamenti, $O(i)$, come in una caccia al tesoro in cui ogni biglietto dice dove si trova il prossimo. In cambio la lista **cresce e si accorcia un nodo alla volta**, e inserire o togliere un elemento, una volta raggiunta la posizione, costa $O(1)$: basta cambiare due puntatori.

OPERAZIONE	ARRAY	LISTA CONCATENATA SEMPLICE
accesso all'elemento di posizione i	$O(1)$	$O(i)$, $O(n)$ nel caso peggiore
ricerca di un valore	$O(n)$; $O(\log n)$ se ordinato	$O(n)$ (niente ricerca binaria)
inserimento/rimozione in testa	$O(n)$ (si sposta tutto)	$O(1)$
inserimento/rimozione dopo un nodo dato	$O(n)$	$O(1)$
dimensione	fissata all'allocazione	cresce e cala a ogni operazione
memoria extra	nessuna	un puntatore per nodo

Esiste una via di mezzo: l'**array di dimensione variabile**, già visto con la pila template della capitolo 4. Quando è pieno si alloca un array nuovo e si copiano gli elementi, operazione che costa $O(n)$. Se però il nuovo array è **il doppio** del vecchio, le riallocazioni diventano sempre più rare, e su una sequenza di n inserimenti il costo totale resta $O(n)$: il costo **distribuito** (o ammortizzato) di ogni inserimento è $O(1)$.

La lista concatenata semplice

NODI E LISTA

Ogni elemento della lista è un **nodo**, con due campi: l'informazione (il **valore**, o **chiave**) e il **puntatore al nodo successivo**. L'ultimo nodo ha come successivo il puntatore nullo, che segna la fine. La lista è identificata dal **puntatore alla testa**, l'unico punto d'accesso: perso quello, si perde tutta la lista. Il nodo si può definire come struct o come class; nel secondo caso i campi sono privati e la classe Lista è dichiarata friend del nodo, così solo lei può manipolarli.

DEFINIZIONE

Lista concatenata

Una **lista concatenata** è una sequenza di nodi, ciascuno dei quali contiene un valore e il puntatore al nodo successivo. La lista è identificata dal puntatore alla testa; l'ultimo nodo ha come successivo il puntatore nullo.



Figura 7.1. Una lista concatenata semplice con i valori 3, 5, 7; la barra indica il puntatore nullo

```
#include <iostream>
using namespace std;

class Lista; // dichiarazione anticipata
class Nodo {
public:
    Nodo(int v, Nodo* s = nullptr) : valore(v), succ(s) {}
private:
    int valore;
    Nodo* succ; // puntatore al nodo successivo
    friend class Lista; // la lista puo' usare valore e succ
    friend ostream& operator<<(ostream& os, const Lista& l);
};

class Lista {
public:
    Lista() : testa(nullptr) {}
    ~Lista();
    void inserisciTesta(int v);
    void inserisciCoda(int v);
    void inserisciOrdinato(int v);
    Nodo* cerca(int v) const;
    bool rimuovi(int v);
    void inverti();
    friend ostream& operator<<(ostream& os, const Lista& l);
private:
    Nodo* testa; // unico punto d'accesso alla lista
};

void Lista::inserisciTesta(int v) { // 0(1)
    testa = new Nodo(v, testa); // il nuovo nodo punta alla vecchia testa
}

void Lista::inserisciCoda(int v) { // 0(n): bisogna arrivare all'ultimo
    Nodo* nuovo = new Nodo(v);
```

```

    if (testa == nullptr) { testa = nuovo; return; } // lista vuota
    Nodo* p = testa;
    while (p->succ != nullptr) p = p->succ; // si ferma sull'ultimo nodo
    p->succ = nuovo;
}

void Lista::inserisciOrdinato(int v) { // mantiene la lista crescente
    if (testa == nullptr || v < testa->valore) { inserisciTesta(v); return; }
    Nodo* p = testa;
    while (p->succ != nullptr && p->succ->valore < v) // si ferma sul predecessore
        p = p->succ;
    p->succ = new Nodo(v, p->succ); // aggancia tra p e il suo successivo
}

Nodo* Lista::cerca(int v) const { // O(n): ricerca sequenziale
    Nodo* p = testa;
    while (p != nullptr && p->valore != v)
        p = p->succ;
    return p; // nullptr se non trovato
}

bool Lista::rimuovi(int v) { // rimuove la prima occorrenza di v
    Nodo* ant = nullptr; // nodo precedente
    Nodo* pos = testa; // nodo corrente
    while (pos != nullptr && pos->valore != v) {
        ant = pos;
        pos = pos->succ;
    }
    if (pos == nullptr) return false; // non trovato
    if (ant == nullptr) testa = pos->succ; // era il primo: si sposta la testa
    else ant->succ = pos->succ; // altrimenti si "scavalca" pos
    delete pos;
    return true;
}

void Lista::inverti() { // gira i puntatori uno alla volta
    Nodo* prec = nullptr;
    Nodo* corr = testa;
    while (corr != nullptr) {
        Nodo* succ = corr->succ; // 1) salva il resto della lista
        corr->succ = prec; // 2) gira il puntatore
        prec = corr; // 3) avanza di un nodo
        corr = succ;
    }
    testa = prec; // l'ultimo nodo e' la nuova testa
}

Lista::~Lista() { // libera tutti i nodi
    while (testa != nullptr) {
        Nodo* da_eliminare = testa;
        testa = testa->succ;
        delete da_eliminare;
    }
}

ostream& operator<<(ostream& os, const Lista& l) {
    for (Nodo* p = l.testa; p != nullptr; p = p->succ)
        os << p->valore << " ";
    return os;
}

```

```

int main() {
    Lista l;
    l.inserisciTesta(5); l.inserisciTesta(3); l.inserisciCoda(7);
    cout << l << endl;           // 3 5 7
    l.inverti();                 cout << l << endl;   // 7 5 3
    l.rimuovi(5);                 cout << l << endl;   // 7 3
    cout << (l.cerca(3) != nullptr) << (l.cerca(9) != nullptr) << endl; // 10
    Lista o;
    int dati[] = {8, 2, 5, 9, 1};
    for (int x : dati) o.inserisciOrdinato(x);
    cout << o << endl;           // 1 2 5 8 9
    return 0;
}

```

INSERIMENTO

L'**inserimento in testa** è l'operazione più semplice e costa $O(1)$. Si crea il nuovo nodo facendolo puntare **alla vecchia testa** (passo 1 della Figura 7.2), poi si sposta la testa sul nuovo nodo (passo 2). L'ordine conta: se si spostasse prima la testa, il riferimento al resto della lista andrebbe perso. Con un costruttore che riceve valore e successivo, i due passi stanno in una riga: `testa = new Nodo(v, testa);`. Funziona anche con la lista vuota, perché il nuovo nodo riceve come successivo `nullptr`.



Figura 7.2. Inserimento in testa del valore 9: prima si collega il nuovo nodo (1), poi si sposta la testa (2)

Per **inserire dopo un nodo p** già noto si fa la stessa cosa in mezzo alla lista: il nuovo nodo punta al successore di p, poi p punta al nuovo nodo (`p->succ = new Nodo(v, p->succ);`). L'**inserimento in coda** invece richiede di **scorrere tutta la lista** per trovare l'ultimo nodo, quindi costa $O(n)$, a meno di mantenere anche un puntatore all'ultimo nodo (questo capitolo). L'**inserimento ordinato**, che mantiene la lista crescente, scorre fino al nodo **dopo il quale** va messo il nuovo valore e lo aggancia lì; il caso del nuovo minimo va trattato a parte, perché cambia la testa.

RICERCA E RIMOZIONE

La **ricerca** è sequenziale: si parte dalla testa e si avanza finché si trova il valore o si arriva a `nullptr`. Per la **rimozione** di un nodo non basta trovarlo: bisogna modificare il **nodo precedente**, che è l'unico a puntarlo. Per questo si scorre la lista con **due puntatori**, pos sul nodo corrente e ant sul precedente (Figura 7.3). Trovato il nodo, ci sono due casi. Se ant è nullo il nodo da togliere è il **primo**, e si sposta la testa; altrimenti il precedente «scavalca» il nodo: `ant->succ = pos->succ`. Infine si libera la memoria con `delete pos`. Dimenticare il `delete` non dà errori, ma lascia un nodo irraggiungibile in memoria (*memory leak*).

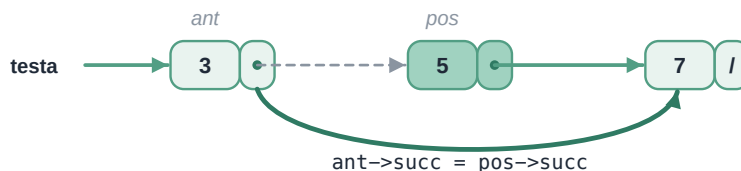


Figura 7.3. Rimozione del nodo con valore 5: il precedente viene collegato al successivo

INVERSIONE E DISTRUZIONE

Per **invertire** una lista sul posto si scorre con tre puntatori, *prec*, *corr* e *succ*. A ogni passo si salva il resto della lista in *succ*, si gira il puntatore del nodo corrente verso il precedente, e si avanzano *prec* e *corr* di un nodo. Alla fine *corr* è nullo e *prec* punta all'ultimo nodo originale, che diventa la nuova testa. Il **distruttore** deve liberare tutti i nodi: scorre la lista salvando ogni volta il successivo prima di cancellare il nodo corrente, perché dopo il delete il campo *succ* non è più leggibile.

ESERCIZIO

Completare la funzione `inverti()`

```
void inverti(Nodo* testa) {
    Nodo* prev = nullptr;
    Nodo* current = testa;
    Nodo* next;
    while (current != nullptr) {
        next = current->succ;
        current->succ = prev;
        prev = current;
        current = next;
    }
    // AGGIUNGERE RIGA QUI
}
```

La riga mancante è `testa = prev;`: all'uscita dal ciclo *current* e *next* sono nulli, mentre *prev* punta al vecchio ultimo nodo, che è la nuova testa. Una precisazione importante: così com'è scritta, la funzione riceve *testa* **per valore**, quindi l'assegnamento cambia solo la copia locale e il chiamante continua a puntare al vecchio primo nodo, diventato l'ultimo. Perché funzioni davvero il parametro va passato per riferimento, `Nodo*& testa`, oppure la funzione deve restituire *prev*. Nel metodo `inverti()` di `p7_lista.cpp` il problema non c'è, perché *testa* è un attributo della classe.

LE DOMANDE «COSA FA LA FUNZIONE `BOO()`»

Capita spesso di dover capire cosa fa una funzione che scorre una lista. Il metodo è sempre lo stesso: individua **cosa viene restituito** (spesso c'è una variabile calcolata ma mai restituita, messa apposta per confondere), **quando si avanza** e **cosa si stampa o si conta**, e prova a eseguire la funzione a mano su una lista di due o tre nodi.

```

#include <iostream>
using namespace std;
struct Nodo { int valore; Nodo* succ; };

void stampaAvanti(Nodo* p) {          // stampa, POI chiama: dalla testa alla coda
    if (p == nullptr) return;
    cout << p->valore << " ";
    stampaAvanti(p->succ);
}
void stampaIndietro(Nodo* p) {        // chiama, POI stampa: dalla coda alla testa
    if (p == nullptr) return;
    stampaIndietro(p->succ);
    cout << p->valore << " ";
}
int ordinata(Nodo* p) {               // 1 se la lista e' in ordine non decrescente
    return p == nullptr || p->succ == nullptr ||
        (p->valore <= p->succ->valore && ordinata(p->succ));
}
int main() {
    Nodo c = {7, nullptr}, b = {5, &c}, a = {3, &b};    // lista 3 -> 5 -> 7
    stampaAvanti(&a);   cout << endl;    // 3 5 7
    stampaIndietro(&a); cout << endl;    // 7 5 3
    cout << ordinata(&a) << endl;        // 1
    return 0;
}

```

VERIFICA

Domande sulle funzioni di lista

void boo(Nodo* testa) { if (testa == nullptr) return; cout << testa->valore << endl; boo(testa->succ); }

Stampa tutto il contenuto della lista **dalla testa alla coda**: si stampa il nodo, poi si passa al resto (come stampaAvanti).

void boo(Nodo* testa) { if (testa == nullptr) return; boo(testa->succ); cout << testa->valore << endl; }

Stampa la lista **in ordine inverso**: la chiamata ricorsiva arriva fino in fondo e le stampe avvengono al ritorno, dall'ultimo nodo al primo (come stampaIndietro).

int Lista::boo() { Nodo* aus = testa; int c = 0, i = 0; while (aus) { c++; if (aus->valore == 10) i++; aus = aus->succ; } return c; }

Restituisce il **numero di elementi della lista**. La variabile i conta le occorrenze di 10, ma non viene restituita: è il distrattore. Analogamente, una funzione boo(Lista<T>& testa, T& x, int i = 20) che azzerava i e lo incrementa quando $x == q \rightarrow \text{getValore}()$ conta **gli elementi uguali a x**: il default 20 viene sovrascritto subito.

int func(Nodo* p) { return p == nullptr || p->succ == nullptr || (p->valore <= p->succ->valore && func(p->succ)); } restituisce 1 se e solo se...

Gli elementi della lista sono ordinati in modo non decrescente. Una lista vuota o di un solo elemento è ordinata; altrimenti il primo deve essere $\leq$ del secondo e il resto deve essere ordinato. La funzione ordinata di p7_ricorsive.cpp è questa, ed è verificata.

UN PROBLEMA CLASSICO: INDIVIDUARE E RIMUOVERE UN CICLO

Un problema classico chiede una funzione `detectAndRemoveLoop()` che verifica se una lista contiene un **ciclo** (l'ultimo nodo, invece di puntare a `nullptr`, punta a un nodo precedente) e in tal caso lo rimuove. Scorrere la lista finché si arriva in fondo non funziona: se c'è un ciclo, il fondo non arriva mai. La soluzione classica è l'algoritmo di **Floyd**, detto della **lepre e della tartaruga**. Due puntatori partono dalla testa: `slow` avanza di un nodo alla volta, `fast` di due. Se la lista non ha cicli, `fast` arriva alla fine. Se c'è un ciclo, entrambi ci entrano e, come due corridori su una pista circolare a velocità diverse, il più veloce prima o poi raggiunge il più lento: quando `slow == fast` il ciclo è dimostrato.

Per rimuoverlo bisogna trovare l'**ultimo nodo** del ciclo. Una proprietà del procedimento (dimostrabile con un po' di aritmetica sulle distanze) dice che, se si riporta `slow` in testa e si fanno avanzare entrambi di **un** nodo alla volta, si incontrano esattamente nel **nodo in cui inizia il ciclo**. Quindi basta avanzare finché `slow->next` e `fast->next` coincidono: a quel punto `fast` è l'ultimo nodo, e `fast->next = nullptr` spezza il ciclo. Il caso in cui il ciclo parte dalla testa va trattato a parte.

```
// Individuare e rimuovere un ciclo in una lista
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* next;
    Node(int val) : data(val), next(nullptr) {}
};

void printList(Node* head) {
    for (Node* p = head; p != nullptr; p = p->next) cout << p->data << " ";
    cout << endl;
}

bool detectAndRemoveLoop(Node* head) {
    if (head == nullptr || head->next == nullptr) return false;
    // FASE 1 (Floyd, "lepre e tartaruga"): slow avanza di 1, fast di 2
    Node* slow = head;
    Node* fast = head;
    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast) break;           // si sono incontrati: c'e' un ciclo
    }
    if (slow != fast) return false;       // fast ha raggiunto la fine: niente ciclo
    // FASE 2: slow riparte dalla testa, entrambi avanzano di 1;
    // si incontrano all'INIZIO del ciclo, quindi fast->next e' l'ultimo nodo
    slow = head;
    if (slow == fast) {                    // il ciclo parte proprio dalla testa
        while (fast->next != slow) fast = fast->next;
    } else {
        while (slow->next != fast->next) {
            slow = slow->next;
            fast = fast->next;
        }
    }
    fast->next = nullptr;                  // FASE 3: si spezza il ciclo
    return true;
}

int main() {
    Node* head = new Node(50);
    head->next = new Node(20);
```

```

head->next->next = new Node(15);
head->next->next->next = new Node(4);
head->next->next->next->next = new Node(10);
head->next->next->next->next->next = head->next->next;    // 10 -> 15: il ciclo
if (detectAndRemoveLoop(head))
    cout << "Lista linkata dopo la rimozione del ciclo" << endl;
printList(head);                                     // 50 20 15 4 10
while (head) { Node* t = head->next; delete head; head = t; }
return 0;
}

```

Liste doppie, circolari e con sentinella

In una **lista doppiamente concatenata** ogni nodo ha **due puntatori**, prec al precedente e succ al successivo, oltre al valore: il **numero minimo di campi** di un nodo è quindi **tre**. Il vantaggio è che la lista si può percorrere nei due sensi e che, dato un nodo, si raggiunge il precedente in $O(1)$: per rimuovere un nodo non serve più scorrere con un secondo puntatore. Lo svantaggio è che ogni inserimento e rimozione deve aggiornare **il doppio dei puntatori**, e dimenticarne uno corrompe la lista.

Per **rimuovere il nodo X** il precedente deve saltare X in avanti e il successivo deve saltarlo all'indietro:
 $X \rightarrow \text{prec} \rightarrow \text{succ} = X \rightarrow \text{succ}$; e $X \rightarrow \text{succ} \rightarrow \text{prec} = X \rightarrow \text{prec}$; (Figura 7.4). Se X può essere il primo o l'ultimo nodo servono i controlli sui puntatori nulli, come in `p7_doppia.cpp`. Per l'**inserimento in testa** il nuovo nodo punta in avanti alla vecchia testa, la vecchia testa (se esiste) punta indietro al nuovo nodo, e la testa si sposta sul nuovo nodo.

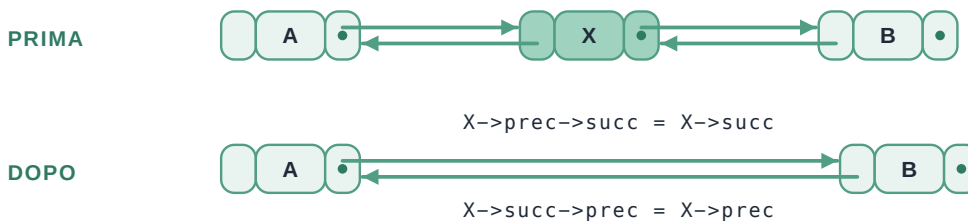


Figura 7.4. Rimozione del nodo X da una lista doppia

```

#include <iostream>
using namespace std;
struct Nodo {
    int valore;
    Nodo* prec;           // nodo precedente
    Nodo* succ;           // nodo successivo
};
// inserimento in testa in una lista doppia
void inserisciTesta(Nodo*& testa, int v) {          // testa passata per riferimento!
    Nodo* nuovo = new Nodo{v, nullptr, testa};
    if (testa != nullptr) testa->prec = nuovo;      // il vecchio primo punta al nuovo
    testa = nuovo;
}
// rimozione del nodo x (x qualsiasi, anche primo o ultimo)
void rimuovi(Nodo*& testa, Nodo* x) {
    if (x->prec != nullptr) x->prec->succ = x->succ; // il precedente scavalca x
    else testa = x->succ;                          // x era il primo
    if (x->succ != nullptr) x->succ->prec = x->prec; // il successivo scavalca x
    delete x;
}
int main() {
    Nodo* testa = nullptr;
    for (int i = 4; i >= 1; i--) inserisciTesta(testa, i); // 1 2 3 4
    rimuovi(testa, testa->succ->succ);                     // toglie 3
    rimuovi(testa, testa);                                 // toglie 1
    for (Nodo* p = testa; p; p = p->succ) cout << p->valore << " "; // 2 4
    cout << endl;
    while (testa) rimuovi(testa, testa);
    return 0;
}

```

Nella funzione `inserisciTesta` il parametro è `Nodo*& testa`, un **referimento a un puntatore**: la funzione deve poter cambiare il puntatore alla testa del chiamante, e passandolo per valore cambierebbe solo una copia. È lo stesso problema della funzione `inverti()` vista sopra: una funzione `InserTestaLista(Nodo* head, int data)` che assegnasse `head = nuovo_ptr` cambierebbe solo la propria copia, e il chiamante non se ne accorgerebbe. Con le classi il problema sparisce, perché la testa è un attributo.

In una **lista circolare** l'ultimo nodo punta al **primo** invece che a `nullptr` (nella versione doppia, anche il primo punta all'ultimo con `prec`). Non c'è un «fondo»: da qualunque nodo si può percorrere tutta la lista, e per fermarsi si controlla di essere tornati al punto di partenza. Una lista **con sentinella** contiene un **nodo speciale**, la **sentinella**, che non contiene informazione utile e serve solo a marcare l'inizio e la fine. In una lista doppia circolare con sentinella, il primo elemento è il successore della sentinella e l'ultimo è il suo predecessore, entrambi raggiungibili in $O(1)$. Soprattutto, la lista **non è mai davvero vuota** (c'è sempre la sentinella) e ogni nodo vero ha sempre un precedente e un successivo: spariscono tutti i casi particolari (lista vuota, primo nodo, ultimo nodo) e il codice di inserimento e rimozione diventa più semplice e leggibile.

VERIFICA

Domande su liste doppie e sentinella

Qual è il numero minimo di campi per ciascun nodo di una lista doppiamente linkata? a) 1 b) 2 c) 3 d) 4

Risposta **c) 3**: il valore e i due puntatori.

Quale segmento di codice elimina correttamente il nodo puntato da X (né primo né ultimo)?

$X \rightarrow \text{prec} \rightarrow \text{succ} = X \rightarrow \text{succ}$; $X \rightarrow \text{succ} \rightarrow \text{prec} = X \rightarrow \text{prec}$; Le opzioni sbagliate usano il punto al posto della freccia su un puntatore ($X.\text{prec}$, non compila) oppure collegano il nodo a se stesso ($X \rightarrow \text{prec} \rightarrow \text{succ} = X \rightarrow \text{prec}$).

Qual è una delle conseguenze dell'aggiunta di un nodo sentinella in una lista doppiamente linkata? a) l'inserimento diventa $O(\log n)$; b) il codice è più leggibile; c) la lista diventa circolare; d) la lista diventa una coda.

Risposta **b)**. La sentinella elimina i casi particolari, e il codice si semplifica. La circolarità è una scelta di implementazione che si accompagna spesso alla sentinella, non una conseguenza obbligata della sua aggiunta. Le altre due sono false.

Le pile

Una **pila** (*stack*) è una sequenza in cui si inserisce e si estrae **sempre dalla stessa estremità**, la **cima**. È la politica **LIFO** (*Last In, First Out*): l'ultimo elemento entrato è il primo a uscire, come in una pila di piatti, dove si appoggia e si prende sempre quello in cima. Le operazioni sono **push** (inserisce in cima), **pop** (estrae l'elemento in cima e lo restituisce), **top** o **peek** (legge la cima senza toglierla) e **isEmpty** (dice se è vuota). Lo stack delle chiamate di funzione è una pila: l'ultima funzione chiamata è la prima a terminare.

DEFINIZIONE

Pila e coda

Una **pila** (*stack*) segue la politica **LIFO**: si inserisce e si estrae dalla stessa estremità, la cima. Una **coda** (*queue*) segue la politica **FIFO**: si inserisce dal fondo e si estrae dalla testa.

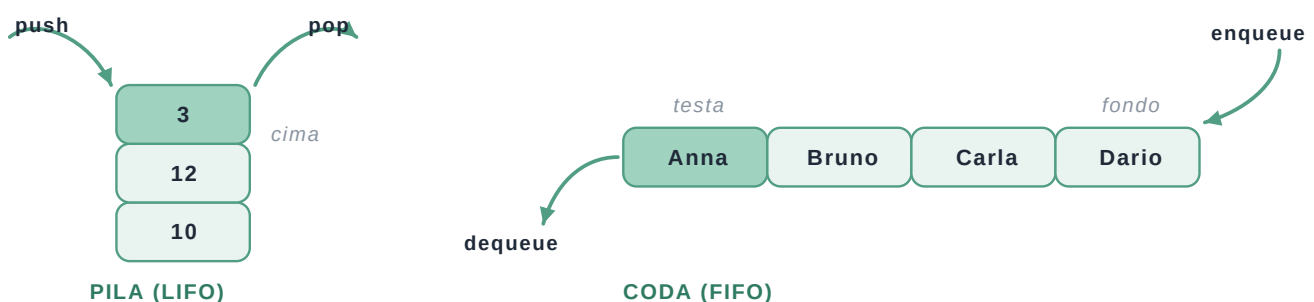


Figura 7.5. A sinistra una pila (si entra e si esce dalla cima), a destra una coda (si esce dalla testa, si entra dal fondo)

Una pila si realizza in due modi. Con un **array** e un indice **top**: **push** scrive in $v[\text{top}]$ e incrementa, **pop** decrementa e legge, tutto $O(1)$. Ma se l'array è **statico** la **dimensione massima della pila è limitata** e bisogna gestire la pila piena; con un array dinamico che raddoppia (capitolo 4) il limite sparisce. Con una **lista concatenata**, **push** è un inserimento in testa e **pop** una rimozione dalla testa: entrambi $O(1)$, e la pila cresce senza limiti. È la realizzazione più comune di una pila di oggetti.

```
#include <iostream>
#include <string>
using namespace std;
```

```

template <typename T>
class Pila {                                // pila (LIFO) realizzata con una lista
public:
    Pila() : cima(nullptr) {}
    ~Pila() { while (!isEmpty()) pop(); }
    void push(const T& x) {                  // O(1): inserimento in testa
        cima = new Elem{x, cima};
    }
    T pop() {                               // O(1): rimozione dalla testa
        Elem* e = cima;                    // preconditione: pila non vuota
        T x = e->dato;
        cima = e->sotto;
        delete e;
        return x;
    }
    const T& top() const { return cima->dato; } // legge senza togliere
    bool isEmpty() const { return cima == nullptr; }
private:
    struct Elem { T dato; Elem* sotto; };    // nodo interno alla classe
    Elem* cima;
};

int main() {
    Pila<int> p;
    p.push(10); p.push(12); p.push(8); p.push(4);
    p.pop(); p.pop();                        // tolgono 4 e 8
    p.push(3);
    cout << p.pop() << " " << p.pop() << endl; // 3 12: servono 2 pop per estrarre 12
    Pila<string> s;
    s.push("a"); s.push("b");
    cout << s.top() << endl;                 // b
    return 0;
}

```

VERIFICA

Domande sulle pile

Si eseguono push(10); push(12); push(8); push(4); pop(); pop(); push(3);. Quanti pop sono necessari per estrarre il numero 12?

Dopo le quattro push la pila contiene, dal fondo alla cima, 10 12 8 4. Le due pop tolgono 4 e 8; la push aggiunge 3: la pila è 10 12 3. Per estrarre il 12 bisogna prima togliere il 3: servono **2 pop**. Il main di p7_pila.cpp esegue esattamente questa sequenza.

Quale di queste strutture dati implementa una politica LIFO? a) BST b) coda c) stack d) lista

Risposta **c) stack**, cioè la pila. La coda è FIFO.

L'implementazione di una pila con un array statico: a) trasforma la pila in una coda; b) limita la dimensione massima della pila; c) consente di effettuare tutte le operazioni in tempo costante; d) non è possibile.

La risposta è **b)**. A rigore anche la c) è vera (con un indice di cima, push e pop sono $O(1)$): ma è vera per qualunque buona implementazione, mentre il limite di dimensione è la **conseguenza specifica** dell'array statico.

Le code

Una **coda** (*queue*) è una sequenza in cui si inserisce da un'estremità, il **fondo**, e si estrae dall'altra, la **testa**. È la politica **FIFO** (*First In, First Out*): il primo entrato è il primo a uscire, come la fila alla cassa o in un ambulatorio. Le operazioni sono **enqueue** (inserisce in fondo), **dequeue** (estrae dalla testa) e **isEmpty**.

Con una **lista concatenata** la scelta naturale è estrarre dalla testa ($O(1)$) e inserire in fondo. Se la classe mantiene **solo il puntatore alla testa**, ogni enqueue deve scorrere tutta la lista per trovare l'ultimo nodo: **$O(n)$** . Se mantiene **anche il puntatore all'ultimo nodo** (il fondo), enqueue aggancia il nuovo nodo dopo l'ultimo e aggiorna il fondo: **$O(1)$** . Attenzione al caso della coda che si svuota: quando dequeue toglie l'ultimo elemento rimasto, anche il puntatore al fondo va rimesso a `nullptr`. Con un **array** la coda si realizza con due indici, inizio e fine; per non sprecare le celle liberate in testa si usa un **array circolare**, in cui gli indici, arrivati in fondo, ripartono da 0 ($\text{fine} = (\text{fine} + 1) \% \text{MAX}$). È la scelta naturale quando la coda ha una dimensione massima fissata, per esempio dieci elementi.

```
#include <iostream>
#include <string>
using namespace std;
template <typename T>
class Coda {                                // coda (FIFO) con puntatori a testa e fondo
public:
    Coda() : testa(nullptr), fondo(nullptr), n(0) {}
    ~Coda() { while (!isEmpty()) dequeue(); }
    void enqueue(const T& x) {                // O(1): si aggiunge dopo l'ultimo
        Elem* e = new Elem{x, nullptr};
        if (fondo == nullptr) testa = e;     // coda vuota: e' anche il primo
        else fondo->succ = e;
        fondo = e;
        n++;
    }
    T dequeue() {                             // O(1): si toglie il primo
        Elem* e = testa;                     // preconditione: coda non vuota
        T x = e->dato;
        testa = e->succ;
        if (testa == nullptr) fondo = nullptr; // la coda e' rimasta vuota
        delete e;
        n--;
        return x;
    }
    bool isEmpty() const { return testa == nullptr; }
    int size() const { return n; }
    // x guadagna al piu' 3 posizioni, usando SOLO enqueue e dequeue
    void enqueuePriority(const T& x) {
        int vecchi = n;                       // elementi presenti prima di x
        int davanti = vecchi > 3 ? vecchi - 3 : 0; // quanti devono restare davanti a x
        for (int i = 0; i < davanti; i++)      // 1) i primi "davanti" vanno in fondo
            enqueue(dequeue());
        enqueue(x);                           // 2) x si accoda dopo di loro
        for (int i = 0; i < vecchi - davanti; i++) // 3) gli scavalcati tornano dopo x
            enqueue(dequeue());
    }
private:
    struct Elem { T dato; Elem* succ; };
    Elem* testa;                             // da qui si estrae
    Elem* fondo;                             // qui si inserisce
    int n;
};
```

```
int main() {
    Coda<string> c;
    c.enqueue("Anna"); c.enqueue("Bruno"); c.enqueue("Carla");
    c.enqueue("Dario"); c.enqueue("Elena");
    c.enqueuePriority("Franco"); // raccomandato: supera Elena, Dario, Carla
    while (!c.isEmpty()) cout << c.dequeue() << " ";
    cout << endl; // Anna Bruno Franco Carla Dario Elena
    return 0;
}
```

Il metodo `enqueuePriority` risolve un problema tipico: inserire un paziente «raccomandato» che **guadagna al più tre posizioni**, usando **solo `enqueue` e `dequeue`** e mantenendo l'ordine degli altri. L'idea è la **rotazione**: estrarre un elemento dalla testa e reinserirlo in fondo non cambia l'ordine ciclico della coda. Se ci sono vecchi elementi, quelli che devono restare davanti al raccomandato sono vecchi $- 3$ (o nessuno, se sono meno di tre). Si spostano in fondo, si accoda il raccomandato, poi si spostano in fondo anche i tre scavalcati: dopo un giro completo l'ordine è quello voluto. Con Anna, Bruno, Carla, Dario, Elena in coda, Franco raccomandato finisce dopo Bruno.

VERIFICA

Domande sulle code

L'operazione di `enqueue` nell'implementazione con una lista linkata con puntatori alla testa e alla coda avviene in tempo...

Costante, $O(1)$: si aggancia il nuovo nodo dopo quello puntato dalla coda.

L'operazione di `enqueue` nell'implementazione con una lista linkata con il solo puntatore alla testa avviene in tempo...

Lineare, $O(n)$: per inserire in fondo bisogna raggiungere l'ultimo nodo scorrendo dalla testa.

Avendo i puntatori al primo e all'ultimo elemento di una lista concatenata semplice, quale operazione ha un costo che dipende dalla lunghezza della lista? a) eliminare il primo; b) inserire in testa; c) eliminare l'ultimo; d) aggiungere alla fine.

Risposta **c) eliminare l'ultimo elemento**. Per toglierlo bisogna aggiornare il **penultimo** (il suo `succ` diventa `nullptr`, e diventa il nuovo ultimo), e in una lista semplice il penultimo si raggiunge solo scorrendo dalla testa: $O(n)$. Le altre tre operazioni sono $O(1)$. È il motivo per cui in una coda su lista si **estrae dalla testa** e si **inserisce in fondo**, e non il contrario.

STRUTTURA	POLITICA	INSERIMENTO	ESTRAZIONE	REALIZZAZIONE TIPICA
pila	LIFO	push in cima, $O(1)$	pop dalla cima, $O(1)$	lista (testa = cima) o array con indice top
coda	FIFO	enqueue in fondo, $O(1)$	dequeue dalla testa, $O(1)$	lista con testa e fondo, o array circolare

Alberi e alberi binari di ricerca

Gli alberi binari di ricerca organizzano i dati in modo che ricerca, inserimento e cancellazione costino quanto l'altezza dell'albero. Il capitolo insegna a costruirli a mano, a eseguire le tre visite con un metodo grafico, a cancellare un nodo nei tre casi possibili.

Gli alberi

DALLE LISTE AGLI ALBERI

In una lista ogni elemento ha **un solo successore**. Un **albero** generalizza la lista: ogni elemento può avere **più successori**. È la struttura naturale per rappresentare **gerarchie** (l'organigramma di un'azienda, le cartelle di un disco, i capitoli e paragrafi di un libro) e **partizioni ricorsive** di un insieme. Formalmente, un **grafo diretto** è un insieme di nodi collegati da archi orientati; un **albero** è un grafo diretto in cui **ogni nodo ha al più un arco entrante** (un solo «genitore») e un numero qualsiasi di archi uscenti, e in cui esiste **una sola radice**, cioè un solo nodo senza archi entranti. Ne segue che **da ogni nodo alla radice c'è un unico cammino**. Se un nodo avesse due archi entranti, la struttura non sarebbe più un albero.

IL VOCABOLARIO

Il vocabolario degli alberi viene dagli alberi genealogici, ma si disegnano **capovolti**, con la radice in alto. La **radice** è il nodo senza genitore, in cima. Una **foglia** è un nodo senza figli. Se c'è un arco da A a B, A è il **padre** (o genitore) di B e B è un **figlio** di A; due nodi con lo stesso padre sono **fratelli**. I nodi nel cammino dalla radice a un nodo sono i suoi **ascendenti** (o antenati); i nodi che stanno «sotto» di lui sono i suoi **discendenti**. Ogni nodo che non è una foglia è la radice di un **sottoalbero**, formato da lui e da tutti i suoi discendenti: da qui la natura ricorsiva degli alberi.

Il **livello** (o profondità) di un nodo è la sua distanza dalla radice: la radice ha livello 0, i suoi figli livello 1, i nipoti livello 2. Si può calcolare risalendo verso la radice e contando i passi. I fratelli hanno lo stesso livello, ma non tutti i nodi dello stesso livello sono fratelli (5 e 20 sì, 4 e 15 no). La **profondità** (o **altezza**) dell'albero è la lunghezza del cammino più lungo dalla radice a una foglia, cioè il livello massimo: l'albero della Figura 8.1 ha altezza 3.

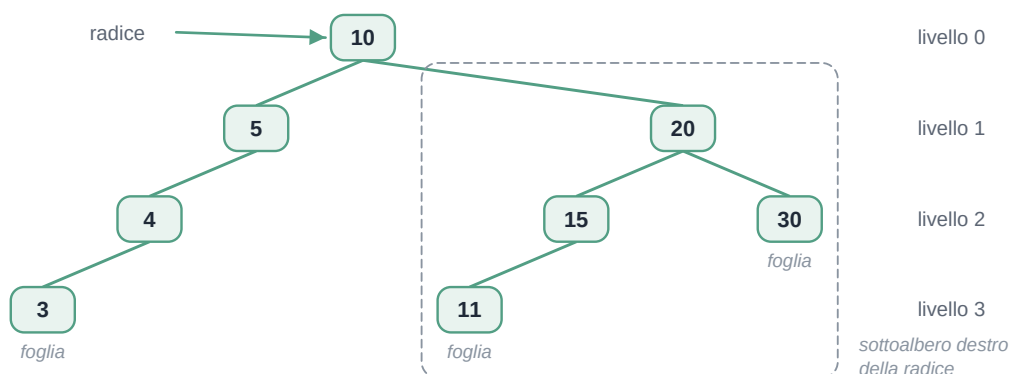


Figura 8.1. L'albero usato in tutto il capitolo, ottenuto inserendo 10 5 20 4 3 15 30 11

LA DEFINIZIONE RICORSIVA

Il modo più utile di pensare un albero è **ricorsivo**: un albero è un insieme di nodi che **o è vuoto, oppure** ha un nodo detto **radice** da cui discendono zero o più **sottoalberi, che sono a loro volta alberi**. Questa definizione è la chiave di tutto il codice sugli alberi. Ogni funzione ricorsiva su un albero ha la stessa forma: **caso base**, l'albero vuoto (puntatore nullo), per cui la risposta è immediata; **passo ricorsivo**, si risolve il problema sui sottoalberi e si combinano i risultati con l'informazione della radice. Per esempio, il numero di nodi di un albero è 0 se è vuoto, altrimenti 1 (la radice) più i nodi del sottoalbero sinistro più quelli del destro.

ALBERI BINARI

Un albero è **binario** se ogni nodo ha **al più due figli**, distinti in **figlio sinistro** e **figlio destro** (la distinzione conta: un nodo con il solo figlio sinistro è diverso da uno con il solo figlio destro). In C++ un nodo di albero binario è come un nodo di lista con **due puntatori** invece di uno, uno per ciascun figlio, più eventualmente un puntatore al **padre**.

Tre proprietà quantitative tornano nelle domande e nell'analisi della complessità. **Al livello k** un albero binario contiene **al più 2^k nodi** (1 radice, 2 figli, 4 nipoti, ...). Un albero binario di **altezza h** contiene quindi **al più $2^{h+1} - 1$ nodi** in tutto ($1 + 2 + 4 + \dots + 2^h$). Viceversa, un albero binario con n nodi ha altezza **almeno $\log_2 n$** circa: quando tutti i livelli sono pieni l'albero si dice **bilanciato** (tutte le foglie allo stesso livello e ogni nodo interno con esattamente due figli) e la sua **altezza è $O(\log n)$** . All'estremo opposto c'è l'albero **degenere**, in cui ogni nodo interno ha un solo figlio: è di fatto una lista, e la sua altezza è $n - 1$.

VERIFICA

Quanti nodi può contenere al massimo un albero binario di profondità n?

$2^{n+1} - 1$: 2^k nodi al livello k, sommati per k da 0 a n (serie geometrica). Un albero di profondità 3 ne contiene al più 15. Attenzione a non confondere con 2^n (i nodi del solo ultimo livello).

Qual è la differenza tra un albero bilanciato e un albero degenere?

In un albero bilanciato i livelli sono (quasi) pieni e l'altezza è $O(\log n)$; in un albero degenere ogni nodo non foglia ha un solo figlio, l'albero è una catena e l'altezza è $n - 1$. Poiché le operazioni su un BST costano $O(\text{altezza})$, la differenza è quella tra $O(\log n)$ e $O(n)$.

L'albero binario di ricerca

LA PROPRIETÀ FONDAMENTALE

Un **albero binario di ricerca** (BST, *Binary Search Tree*) è un albero binario in cui ogni nodo contiene una **chiave** e vale questa **proprietà**: per ogni nodo x, **tutte le chiavi del sottoalbero sinistro di x sono minori della chiave di x**, e **tutte quelle del sottoalbero destro sono maggiori o uguali**. In formula: se y è nel sottoalbero sinistro di x allora $y.\text{valore} < x.\text{valore}$, altrimenti $y.\text{valore} \geq x.\text{valore}$. Le chiavi uguali vanno quindi a destra (in alcuni testi si ammette $\leq$ a sinistra e $\geq$ a destra; qui un valore uguale finisce sempre a destra).

DEFINIZIONE

Albero binario di ricerca

Un **albero binario di ricerca** è un albero binario in cui, per ogni nodo x, tutte le chiavi del sottoalbero sinistro sono minori della chiave di x e tutte quelle del sottoalbero destro sono maggiori o uguali.

Attenzione: la proprietà riguarda **tutto il sottoalbero**, non solo i figli diretti. Nell'albero della Figura 8.1, 11 sta nel sottoalbero destro di 10 (perché $11 \geq 10$) e nel sottoalbero sinistro di 20 e di 15 (perché $11 < 20$ e $11 < 15$): è in regola con **tutti** i suoi antenati. Il BST è una ricerca binaria «messa in forma di struttura dati»: a ogni nodo il confronto con la chiave dice da che parte proseguire, e l'altra metà viene scartata. A differenza dell'array ordinato, però, il BST è **dinamico**: inserire e cancellare non richiede di spostare gli altri elementi.

COME SI INSERISCE UNA CHIAVE

I nuovi elementi vengono **sempre inseriti come nuove foglie**. Per inserire la chiave k si parte dalla radice e si scende: se k è **minore** della chiave del nodo corrente si va a **sinistra**, altrimenti a **destra**. Si continua finché si trova un posto vuoto (un puntatore nullo): lì si attacca il nuovo nodo. La procedura iterativa usa due puntatori, x che scende e y che lo insegue e alla fine è il **padre** del nuovo nodo; se y è nullo l'albero era vuoto e il nuovo nodo diventa la radice. Il costo è proporzionale al numero di livelli attraversati: $O(h)$, dove h è l'altezza.

Costruire un BST a mano

IL PROCEDIMENTO

Molte domande partono da una **sequenza di chiavi inserite in un BST inizialmente vuoto**. Il procedimento è meccanico, e conviene seguirlo sempre per intero, disegnando l'albero a ogni passo. **La prima chiave diventa la radice**, e non si sposta più. Per ogni chiave successiva si parte dalla radice e ci si chiede «è minore?»: se sì si va a sinistra, se no a destra; si ripete sul nodo raggiunto finché si trova un posto libero. Un buon controllo finale: leggendo i nodi da sinistra a destra (come se li si proiettasse su una riga), devono risultare **in ordine crescente**.

1) inserisco 10



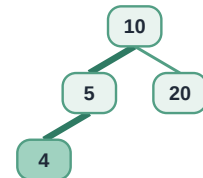
2) inserisco 5



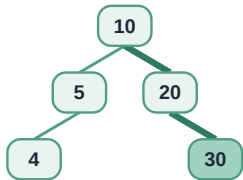
3) inserisco 20



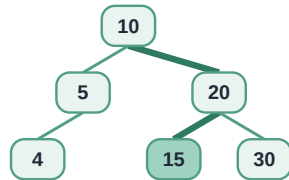
4) inserisco 4



5) inserisco 30



6) inserisco 15



7) inserisco 11

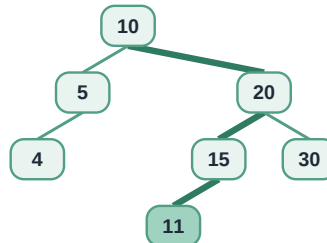


Figura 8.2. Costruzione del BST con la sequenza 10 5 20 4 30 15 11: in grigio il nodo inserito, in grassetto il percorso seguito

ESERCIZIO

Costruzione passo per passo: 10 5 20 4 30 15 11

10 è la radice. **5** < 10: figlio sinistro di 10. **20** ≥ 10: figlio destro di 10. **4** < 10 (sinistra, su 5), 4 < 5: figlio sinistro di 5. **30** ≥ 10 (destra, su 20), 30 ≥ 20: figlio destro di 20. **15** ≥ 10 (destra, su 20), 15 < 20: figlio sinistro di 20. **11** ≥ 10 (destra), 11 < 20 (sinistra, su 15), 11 < 15: figlio sinistro di 15. Il risultato è l'albero della Figura 8.2, passo 7: è lo stesso della Figura 8.1 senza il 3.

LE DOMANDE TIPICHE

ESERCIZIO

Figli di un nodo

Sequenza inserita in un BST vuoto: 8 5 7 2 3 6 14 16. Quali sono i due figli del nodo con chiave 5?

Costruiamo. 8 radice. $5 < 8$: sinistro di 8. $7: < 8, \geq 5$: **destro di 5**. $2: < 8, < 5$: **sinistro di 5**. $3: < 8, < 5, \geq 2$: destro di 2. $6: < 8, \geq 5, < 7$: sinistro di 7. $14 \geq 8$: destro di 8. $16: \geq 8, \geq 14$: destro di 14. I figli di 5 sono **2 e 7** (risposta c). Le opzioni trappola, «3 e 6», nascono dal mettere ogni nuova chiave sotto l'ultimo nodo inserito invece di ripartire dalla radice.

ESERCIZIO

Padre di un nuovo nodo

Nel BST della Figura 8.2 (passo 7) si inserisce un nuovo nodo con chiave 13. Che chiave ha il padre?

Si scende dalla radice: $13 \geq 10 \rightarrow$ destra (20); $13 < 20 \rightarrow$ sinistra (15); $13 < 15 \rightarrow$ sinistra (11); $13 \geq 11 \rightarrow$ destra, dove il posto è libero. Il nuovo nodo diventa figlio destro di **11**, che è quindi il padre. Il trucco è non fermarsi al primo nodo «vicino» come valore (15 o 10): bisogna scendere fino a un puntatore nullo.

QUALE SEQUENZA GENERA QUESTO ALBERO?

La domanda inversa mostra un BST e chiede quale tra alcune sequenze lo produce. La regola che la risolve è una sola: **ogni nodo deve comparire nella sequenza dopo tutti i suoi antenati**. Un nodo infatti può essere inserito nella sua posizione solo se il percorso che lo porta lì esiste già; se un antenato arrivasse dopo, il nodo sarebbe finito più in alto. Quindi: la **prima** chiave della sequenza deve essere la **radice**; e per ogni nodo si controlla che padre, nonno e così via lo precedano. L'ordine tra nodi di rami diversi invece è libero: lo stesso albero si può ottenere da molte sequenze diverse.

ESERCIZIO

Quale sequenza genera il BST 10 (5 (4), 20 (15 (11), 30))?

a) **10 20 5 11 15 4 30**: 11 arriva **prima** di 15, che è suo padre. Inserito prima, 11 finirebbe come figlio sinistro di 20, e 15 come figlio destro di 11. Sbagliata. b) **10 5 20 30 11 15 4**: stesso problema, 11 prima di 15. Sbagliata. c) **10 5 20 4 30 15 11**: la radice è prima; 5 e 20 dopo 10; 4 dopo 5; 30 e 15 dopo 20; 11 dopo 15. Ogni nodo segue i suoi antenati: **corretta** (è la sequenza della Figura 8.2).

Una scorciatoia utile: la **visita preorder di un BST è sempre una sequenza che lo genera** (la radice viene prima di tutto il suo sottoalbero, ricorsivamente), e lo stesso vale per la visita per livelli. Se tra le opzioni c'è la preorder dell'albero, è quella giusta.

Le tre visite

LE DEFINIZIONI

Visitare un albero significa esaminare tutti i suoi nodi, ciascuno esattamente una volta (per stamparli, contarli, sommarli...). In una lista l'ordine naturale è uno solo, dalla testa alla coda; in un albero binario ce ne sono tre notevoli, che differiscono solo per il **momento in cui si visita la radice** rispetto ai due sottoalberi. Il prefisso del nome dice proprio questo: *pre* = prima, *in* = in mezzo, *post* = dopo.

DEFINIZIONE

Visite di un albero binario

La visita **preorder** elabora la radice, poi il sottoalbero sinistro, poi il destro. La visita **inorder** elabora il sottoalbero sinistro, la radice, il destro. La visita **postorder** elabora il sottoalbero sinistro, il destro, infine la radice.

VISITA	ORDINE	MNEMONICO
preorder (anticipata)	RADICE, sottoalbero sinistro, sottoalbero destro	R S D
inorder (simmetrica)	sottoalbero sinistro, RADICE, sottoalbero destro	S R D
postorder (posticipata)	sottoalbero sinistro, sottoalbero destro, RADICE	S D R

Il sinistro viene **sempre prima del destro**; cambia solo dove si inserisce la R. E le definizioni sono **ricorsive**: «visita il sottoalbero sinistro» significa applicare **la stessa visita** a quel sottoalbero, che a sua volta ha una radice e due sottoalberi. Il codice è la traduzione letterale delle definizioni: tre funzioni identiche in cui cambia solo la posizione della cout.

```
void preOrder(Nodo* p) {           // R S D
    if (p == nullptr) return;
    cout << p->valore << " ";      // prima la radice
    preOrder(p->sx);
    preOrder(p->dx);
}
void inOrder(Nodo* p) {           // S R D
    if (p == nullptr) return;
    inOrder(p->sx);
    cout << p->valore << " ";      // radice in mezzo
    inOrder(p->dx);
}
void postOrder(Nodo* p) {         // S D R
    if (p == nullptr) return;
    postOrder(p->sx);
    postOrder(p->dx);
    cout << p->valore << " ";      // radice per ultima
}
```

SVOLGERE UNA VISITA A MANO

Applicando la definizione in modo ricorsivo all'albero della Figura 8.1 si ottiene la visita, ma è facile perdersi nei livelli. Per la **preorder**: si scrive la radice, 10; poi si visita in preorder il sottoalbero sinistro, che ha radice 5: si scrive 5, poi il suo sinistro (radice 4: si scrive 4, poi il suo sinistro, 3, che è una foglia), poi il destro di 5, che è vuoto. Tornati a 10, si visita il destro: 20, poi il suo sinistro (15, poi 11), poi il suo destro (30). Risultato: **10 5 4 3 20 15 11 30**.

Per la **inorder** si scende a sinistra il più possibile prima di scrivere qualcosa: il primo nodo scritto è il più a sinistra, 3; poi si risale scrivendo 4 e 5 (che non hanno figli destri), poi la radice 10, poi si passa al sottoalbero destro, dove di nuovo si scende a sinistra fino a 11, poi 15, 20, 30. Risultato: **3 4 5 10 11 15 20 30**. Per la **postorder** un nodo si scrive solo quando entrambi i suoi sottoalberi sono finiti: prima 3, poi 4 e 5; poi nel ramo destro 11, 15, 30, poi 20; la radice per ultima. Risultato: **3 4 5 11 15 30 20 10**.

IL METODO DEL GIRO INTORNO ALL'ALBERO

C'è un metodo grafico che dà tutte e tre le visite senza ragionare sulla ricorsione, ed è il più sicuro quando si lavora a mano. Si immagina di **percorrere il contorno dell'albero** partendo da sinistra della radice e girando in senso antiorario, restando sempre attaccati ai nodi e agli archi, come se si passasse un dito intorno alla sagoma dell'albero. Durante il giro si passa **tre volte vicino a ogni nodo**: una volta alla sua **sinistra** (scendendo), una volta **sotto** (passando dal sottoalbero sinistro al destro), una volta alla sua **destra** (risalendo). Allora: la **preorder** elenca i nodi nell'ordine in cui si passa alla loro **sinistra**; la **inorder** nell'ordine in cui si passa **sotto**; la **postorder** nell'ordine in cui si passa alla loro **destra**.

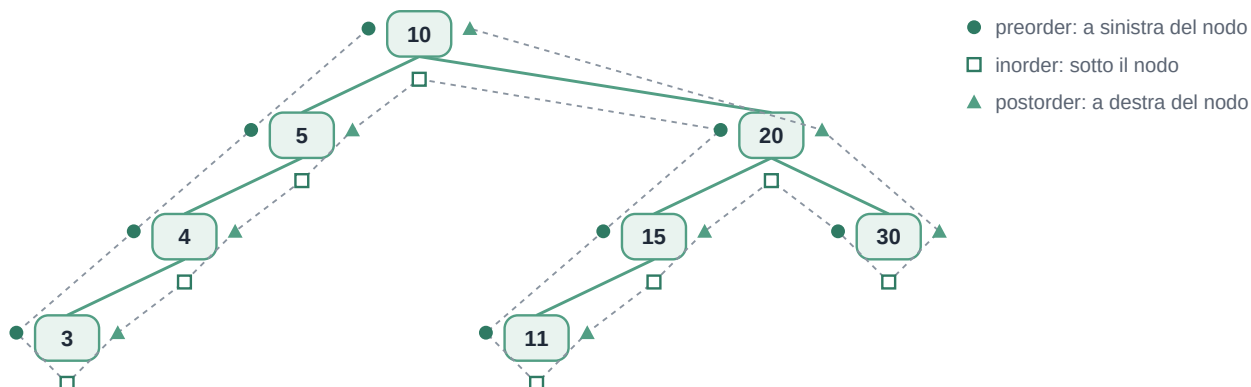


Figura 8.3. Il giro intorno all'albero: i pallini danno la preorder, i quadrati la inorder, i triangoli la postorder

Seguendo la linea tratteggiata della Figura 8.3 dalla sinistra della radice si incontrano, nell'ordine: i pallini di 10, 5, 4, 3 (preorder fin qui: 10 5 4 3); il quadrato e il triangolo di 3; il quadrato e il triangolo di 4, poi di 5; il quadrato di 10; il pallino di 20, di 15, di 11; e così via fino al triangolo di 10, che è l'ultimo punto del giro. Leggendo solo un tipo di simbolo alla volta si ottengono le tre sequenze. Per disegnarlo su carta basta fare, per ogni nodo, un puntino a sinistra, uno sotto e uno a destra, e numerare i puntini nell'ordine in cui il dito li tocca.

DA RICORDARE

Quattro proprietà che fanno risparmiare tempo

- 1. La inorder di un BST è la sequenza delle chiavi in ordine crescente.** Per la proprietà del BST, tutto il sottoalbero sinistro (chiavi minori) viene scritto prima della radice e tutto il destro (chiavi maggiori) dopo, a ogni livello. È il modo standard per **stampare un BST in ordine** e un controllo immediato: se la inorder non è crescente, l'albero è sbagliato. Si possono anche **ordinare n valori** inserendoli in un BST e facendone la inorder.
- 2. Nella preorder la radice è il primo elemento; nella postorder è l'ultimo.** Anche ricorsivamente: in una preorder, subito dopo la radice compare la radice del sottoalbero sinistro.
- 3. Le foglie compaiono nello stesso ordine relativo (da sinistra a destra) in tutte e tre le visite.** Nell'esempio le foglie sono 3, 11, 30 in tutte e tre: utile per scartare opzioni sbagliate.
- 4. La preorder di un BST, reinserita in un BST vuoto, ricostruisce lo stesso albero** (questo capitolo).

VERIFICA

Visita dopo una cancellazione

Nel BST 10 (5 (4 (3)), 20 (15 (11), 30)) si elimina il nodo con chiave 20. Qual è la sequenza dei nodi visitati da una visita postorder? a) 3 4 5 10 11 15 30 b) 10 5 4 3 30 15 11 c) 3 4 5 11 15 30 10 d) 3 11 4 15 5 30 10

20 ha due figli: si sostituisce con il suo **successore**, il minimo del sottoalbero destro, che è 30 (questo capitolo). L'albero diventa 10 (5 (4 (3)), 30 (15 (11))). Postorder: sottoalbero sinistro 3 4 5, sottoalbero destro 11 15 30, radice 10. Risposta **c) 3 4 5 11 15 30 10**. Per scartare in fretta le altre: la a) è crescente, quindi è una inorder; la b) comincia con la radice, quindi è una preorder; la d) ha le foglie in ordine sbagliato.

Lo stesso schema vale con le altre visite e altri nodi. Il procedimento è sempre: **prima** ridisegnare l'albero dopo l'operazione, **poi** fare la visita con il metodo del giro.

IL COSTO DI UNA VISITA

Una visita esamina ogni nodo esattamente una volta e per ciascuno fa un lavoro costante (la stampa, più due chiamate), quindi il suo costo è $\Theta(n)$: almeno n passi servono perché ogni nodo va stampato ($\Omega(n)$), e non ne servono di più ($O(n)$). Formalmente, se il sottoalbero sinistro ha k nodi e il destro $n - k - 1$, vale $T(n) = T(k) + T(n - k - 1) + d$, con $T(0) = c$ costante (il test del puntatore nullo), che si dimostra essere lineare qualunque sia la forma dell'albero. Lo spazio usato dallo stack delle chiamate è invece proporzionale all'**altezza**.

Esiste anche una quarta visita, **per livelli** (o in ampiezza): si scrivono i nodi livello per livello, da sinistra a destra (10 5 20 4 15 30 3 11). Non si fa con la ricorsione ma con una **coda**: si mette in coda la radice, poi finché la coda non è vuota si estrae un nodo, lo si stampa e si accodano i suoi figli. È la stessa idea della visita in ampiezza dei grafi (capitolo 9).

```

#include <iostream>
using namespace std;
struct Nodo { int valore; Nodo* sx; Nodo* dx; };
Nodo* inserisci(Nodo* r, int v) {
    if (!r) return new Nodo{v, nullptr, nullptr};
    if (v < r->valore) r->sx = inserisci(r->sx, v); else r->dx = inserisci(r->dx, v);
    return r;
}
// visita per livelli (in ampiezza): usa una coda di puntatori ai nodi
void perLivelli(Nodo* radice, int n) {      // n = numero di nodi dell'albero
    if (radice == nullptr) return;
    Nodo** coda = new Nodo*[n];           // coda realizzata con un array
    int testa = 0, fondo = 0;
    coda[fondo++] = radice;                // enqueue della radice
    while (testa < fondo) {                // finche' la coda non e' vuota
        Nodo* p = coda[testa++];          // dequeue
        cout << p->valore << " ";
        if (p->sx) coda[fondo++] = p->sx;  // enqueue dei figli, prima il sinistro
        if (p->dx) coda[fondo++] = p->dx;
    }
    cout << endl;
    delete[] coda;
}
int main() {
    Nodo* r = nullptr;
    int chiavi[] = {10, 5, 20, 4, 3, 15, 30, 11};
    for (int k : chiavi) r = inserisci(r, k);
    perLivelli(r, 8);                      // 10 5 20 4 15 30 3 11
    return 0;
}

```

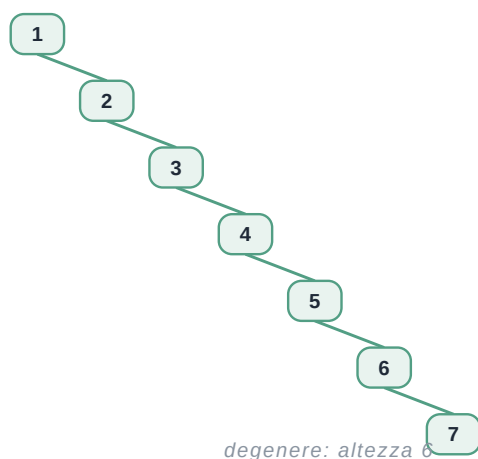
Ricerca, minimo, massimo e successore

La **ricerca** di una chiave k segue lo stesso percorso dell'inserimento: si parte dalla radice; se il nodo è nullo k non c'è, se la chiave del nodo è k l'abbiamo trovata, altrimenti si scende a sinistra se k è minore e a destra se è maggiore. Si può scrivere in forma **ricorsiva**, che ricalca la definizione, o **iterativa**, con un ciclo `while` e un puntatore che scende: le due versioni sono equivalenti, la seconda non usa lo stack. Il **minimo** si trova scendendo **sempre a sinistra** finché c'è un figlio sinistro; il **massimo** scendendo **sempre a destra**. Nella Figura 8.1 il minimo è 3, il massimo 30.

Il **successore** di un nodo x è il nodo con la chiave immediatamente successiva, cioè quello che segue x nella inorder. Ci sono due casi. Se x ha un **sottoalbero destro**, il successore è il **minimo di quel sottoalbero** (un passo a destra, poi tutto a sinistra): il successore di 10 è 11. Se x **non ha sottoalbero destro**, si **risale** verso la radice finché si arriva da un figlio sinistro: il successore è il primo antenato di cui x sta nel sottoalbero sinistro. Il successore di 5 è 10 (5 non ha figli destri; risalendo, 5 è figlio sinistro di 10). Il successore di 30 non esiste (30 è il massimo). Il successore serve alla cancellazione.

Tutte queste operazioni scendono (o risalgono) lungo un solo cammino, quindi costano **$O(h)$** , dove h è l'altezza dell'albero. Il punto cruciale è che **h dipende dall'ordine di inserimento**. Inserendo le chiavi in ordine casuale l'albero tende a essere bilanciato e h è $O(\log n)$: con un milione di chiavi, una ventina di livelli. Inserendole **già ordinate** (1, 2, 3, ...) ogni nuova chiave finisce a destra dell'ultima e l'albero **degenera in una lista**: $h = n - 1$, e ogni operazione costa $O(n)$. Esistono BST che si ribilanciano da soli (AVL, rosso neri), ma non sono trattati in questa dispensa.

INSERIMENTO 1 2 3 4 5 6 7



INSERIMENTO 4 2 6 1 3 5 7

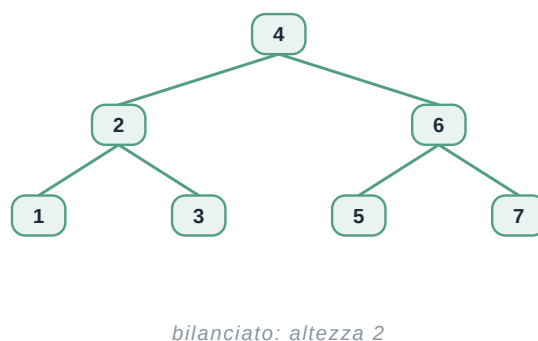


Figura 8.4. Le stesse sette chiavi inserite in ordine diverso: un albero degenere e uno bilanciato

OPERAZIONE	CASO GENERALE	ALBERO BILANCIATO (CASO OTTIMO)	ALBERO DEGENERE (CASO PEGGIORE)
ricerca	$O(h)$	$O(\log n)$	$O(n)$
inserimento	$O(h)$	$O(\log n)$	$O(n)$
minimo, massimo, successore	$O(h)$	$O(\log n)$	$O(n)$
cancellazione	$O(h)$	$O(\log n)$	$O(n)$
visita pre/in/postorder	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

VERIFICA

Domande sulla complessità e sul codice

Qual è la complessità asintotica nel caso ottimo della procedura di inserimento in un albero binario di ricerca contenente n chiavi?

Logaritmica in n : è il caso dell'albero bilanciato, in cui il percorso dalla radice a una foglia è lungo $O(\log n)$. (In senso stretto il caso più favorevole sarebbe una radice con un posto libero subito sotto, ma la domanda si riferisce al caso dell'albero bilanciato.)

```
NodeBST<T>* ptr = root; while (ptr->getParent()) { cout << ptr->getKey() << endl; } while (ptr) { cout << ptr->getKey() << endl; ptr = ptr->getRight(); }
```

Il primo ciclo non viene mai eseguito, perché la radice non ha padre: `getParent()` restituisce nullo. Il secondo stampa la radice e poi scende sempre a destra: **stampa le chiavi della radice e di tutti i figli destri a partire dalla radice**, cioè il percorso verso il massimo.

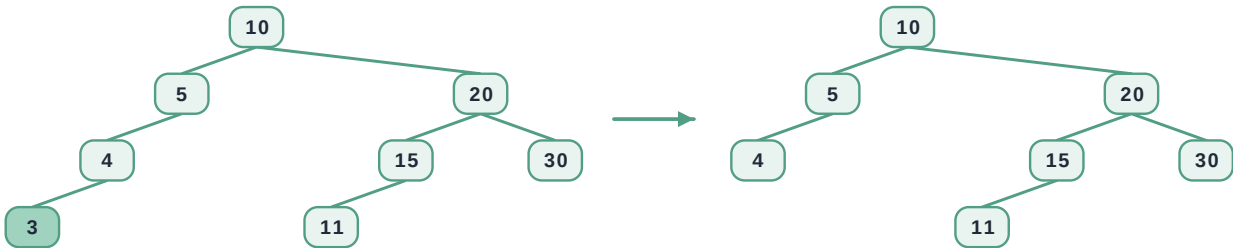
La cancellazione

Cancellare un nodo z è l'operazione più delicata, perché dopo la rimozione l'albero deve restare un BST. Ci sono **tre casi**, illustrati nella Figura 8.5 sullo stesso albero di partenza.

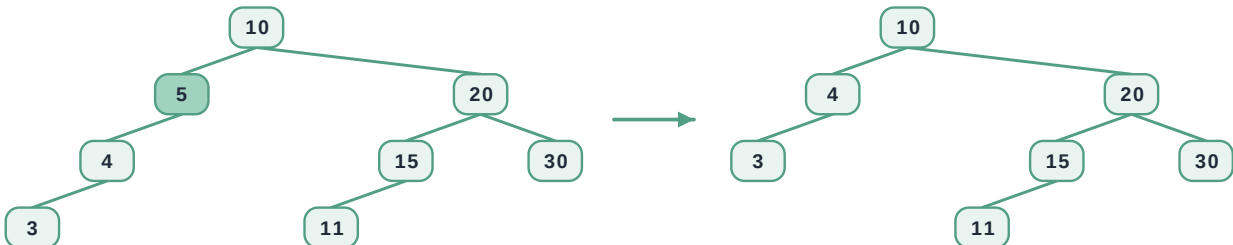
Caso 1: z è una foglia. Si toglie e basta: il puntatore che il padre aveva verso z diventa nullo. **Caso 2: z ha un solo figlio.** Il figlio (con tutto il suo sottoalbero) **sale al posto di z** : il padre di z adotta il figlio di z . La proprietà del BST resta vera, perché quel sottoalbero stava già dalla parte giusta rispetto a tutti gli antenati di z . **Caso 3: z ha due figli.** Non si può far salire uno dei figli, perché l'altro resterebbe senza posto. Si cerca allora il **successore y** di z , che sta certamente nel sottoalbero destro di z (è il suo minimo) e **non ha figlio sinistro** (altrimenti non sarebbe il minimo). y prende **il posto di z** : è la scelta giusta perché y è maggiore di tutto il sottoalbero sinistro di z e minore o uguale a tutto il resto del sottoalbero destro.

Il caso 3 ha due sottocasi. **3a:** y è proprio il **figlio destro** di z . Allora y sale al posto di z portandosi dietro il suo sottoalbero destro, e adotta il sottoalbero sinistro di z . Nella Figura 8.5, cancellando 20 sale 30, che adotta 15. **3b:** y sta **più in basso** nel sottoalbero destro. Allora prima si stacca y dalla sua posizione, facendo salire al suo posto il suo figlio destro (che può anche essere nullo); poi y adotta il sottoalbero destro di z ; infine sale al posto di z e adotta il sottoalbero sinistro. Nella Figura 8.5, cancellando la radice 10 il successore è 11: 11 viene staccato da 15, sale in radice e adotta 5 a sinistra e 20 a destra.

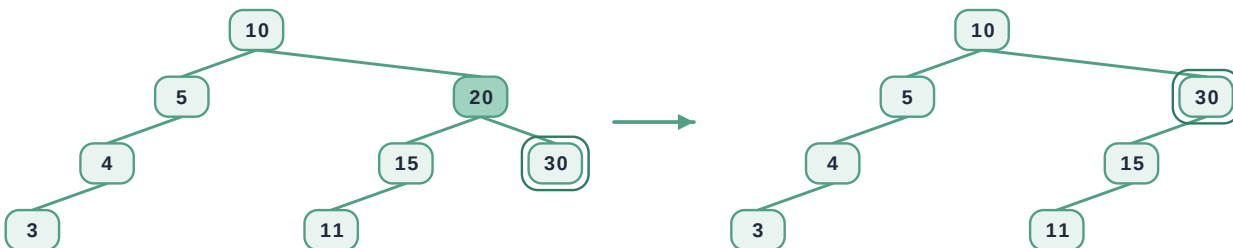
CASO 1: RIMUOVO 3, UNA FOGLIA



CASO 2: RIMUOVO 5, UN SOLO FIGLIO



CASO 3A: RIMUOVO 20, IL SUCCESSORE 30 È FIGLIO DIRETTO



CASO 3B: RIMUOVO 10, IL SUCCESSORE 11 STA PIÙ IN BASSO

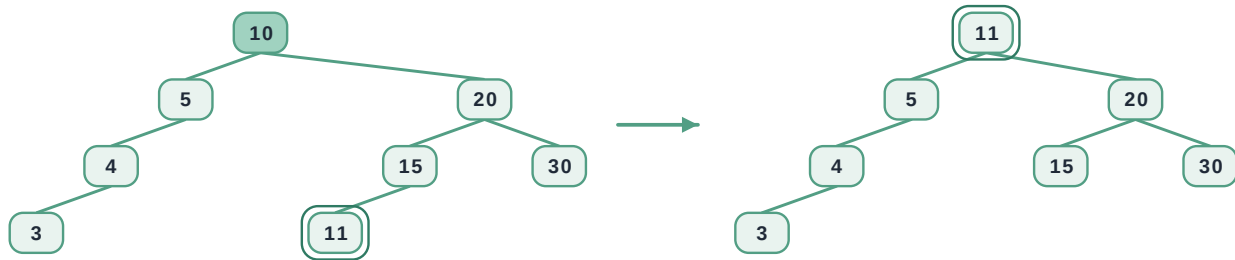


Figura 8.5. I casi della cancellazione; in grigio il nodo da eliminare, cerchiato due volte il successore

La cancellazione si implementa con una procedura ausiliaria, **trapianta** (*transplant*), che **sostituisce il sottoalbero di radice u con il sottoalbero di radice v**: fa puntare a v il padre di u (o la radice, se u era la radice) e aggiorna il padre di v. Con trapianta i casi 1 e 2 si unificano (una foglia è un nodo con il figlio destro nullo, che viene «trapiantato» al suo posto), e il caso 3 si scrive in poche righe. La cancellazione costa **$O(h)$** : la ricerca di z e del suo successore scendono lungo un cammino, il resto sono aggiornamenti di puntatori in tempo costante.

VERIFICA

Domande sulla cancellazione

Quali saranno i figli del nodo con chiave 10 dopo aver eliminato il nodo con chiave 5? (albero della Figura 8.1)

5 ha un solo figlio, 4 (caso 2): 4 sale al posto di 5 portandosi dietro 3. I figli di 10 diventano **4 e 20**.

Si può usare il predecessore al posto del successore?

Sì: anche il **predecessore** (il massimo del sottoalbero sinistro) va bene, per simmetria. La convenzione più diffusa, adottata anche qui, usa però il successore.

Grafi

Il grafo è la struttura dati più generale: liste e alberi ne sono casi particolari. Si studiano le definizioni di base, le due rappresentazioni in memoria e le due visite fondamentali, in ampiezza e in profondità.

Definizioni

Un **grafo** $G = (V, E)$ è formato da un insieme **V di nodi** (o vertici) e da un insieme **E di archi**, ciascuno dei quali collega due nodi. In un **grafo orientato** (o direzionato) ogni arco è una **coppia ordinata** (u, v) : l'arco è **uscente** da u ed **entrante** in v , e si dice che v è **adiacente** a u (non necessariamente il viceversa). In un **grafo non orientato** gli archi sono **coppie non ordinate**: l'arco $\{u, v\}$ collega i due nodi in entrambi i sensi, la relazione di adiacenza è simmetrica e non sono ammessi archi da un nodo a se stesso (*self loop*), che invece sono permessi nei grafi orientati. Un albero è un grafo particolare, connesso e senza cicli, con una radice.

DEFINIZIONE

Grafo

Un **grafo** $G = (V, E)$ è formato da un insieme V di nodi e da un insieme E di archi. In un grafo **orientato** ogni arco è una coppia ordinata (u, v) ; in un grafo **non orientato** è una coppia non ordinata.

Il **grado** di un nodo, in un grafo non orientato, è il numero di archi incidenti su di esso; in un grafo orientato è la somma del **grado entrante** e del **grado uscente**. Un **cammino** di lunghezza k da u a v è una sequenza di nodi $v_0, v_1, \dots, v_k$ con $v_0 = u, v_k = v$ e ogni coppia consecutiva collegata da un arco; v è **raggiungibile** da u se esiste un cammino da u a v . Il cammino è **semplice** se non ripete nodi. Un **ciclo** è un cammino che torna al nodo di partenza ($v_0 = v_k$); un grafo senza cicli si dice **aciclico**.

Un grafo non orientato è **connesso** se ogni coppia di nodi è unita da un cammino. Le **componenti connesse** sono i «pezzi» del grafo: le classi di equivalenza della relazione «è raggiungibile da»; un grafo connesso ha una sola componente. Per i grafi orientati si parla di grafo **fortemente connesso** se per ogni coppia (u, v) esiste un cammino da u a v e uno da v a u ; le **componenti fortemente connesse** sono le classi dei nodi mutuamente raggiungibili. Un **sottografo** $G' = (V', E')$ di G ha $V' \subseteq V$ ed $E' \subseteq E$. Un grafo non orientato è **completo** se ogni coppia di nodi è adiacente.

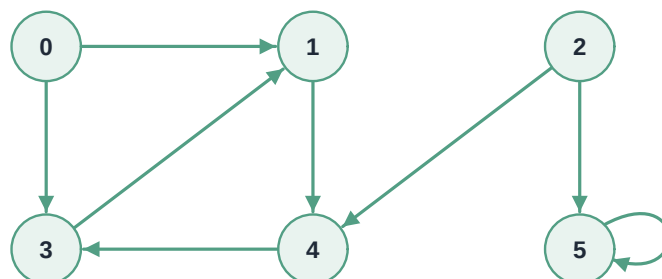


Figura 9.1. Il grafo orientato d'esempio, con i nodi numerati da 0

Nel grafo della Figura 9.1, il nodo 0 ha grado uscente 2 e grado entrante 0; $1 \rightarrow 4 \rightarrow 3 \rightarrow 1$ è un **ciclo**, quindi il grafo non è aciclico; il nodo 5 ha un self loop. Le componenti fortemente connesse sono $\{1, 3, 4\}$ (ciascuno raggiunge gli altri attraverso il ciclo), $\{0\}$, $\{2\}$ e $\{5\}$: da 0 si raggiunge 1 ma da 1 non si torna a 0.

Rappresentare un grafo

Ci sono due modi fondamentali di memorizzare un grafo, con vantaggi opposti. Si indica con $|V|$ il numero di nodi e con $|E|$ il numero di archi (si usano anche n e m , oppure N e M).

Con le **liste di adiacenza** si usa un **array di $|V|$ liste**, una per nodo: la lista $Adj[u]$ contiene tutti i nodi v per cui esiste l'arco (u, v) . È la stessa idea della lista concatenata della capitolo 7, ripetuta per ogni nodo. In un grafo orientato la somma delle lunghezze di tutte le liste è $|E|$ (ogni arco compare una volta); in un grafo non orientato è $2|E|$ (l'arco $\{u, v\}$ compare sia nella lista di u sia in quella di v). Lo spazio occupato è quindi $O(|V| + |E|)$, ed è la rappresentazione adatta ai grafi **sparsi**, con pochi archi rispetto al massimo possibile. Per sapere se esiste l'arco (u, v) però bisogna scorrere la lista di u .

Con la **matrice di adiacenza** si usa una matrice A di $|V| \times |V|$ elementi, con $a_{ij} = 1$ se (i, j) è un arco e 0 altrimenti. In un grafo non orientato la matrice è simmetrica. Lo spazio è $O(|V|^2)$ qualunque sia il numero di archi, ma verificare se un arco esiste costa $O(1)$. È adatta ai grafi **densi** e ai grafi piccoli. Il rovescio è che elencare i vicini di un nodo costa sempre $O(|V|)$, perché bisogna leggere un'intera riga: per questo un algoritmo che esamina tutti i vicini di tutti i nodi su una matrice costa $O(|V|^2)$.

NODO	LISTA DI ADIACENZA	RIGA DELLA MATRICE (COLONNE 0...5)
0	1 → 3	0 1 0 1 0 0
1	4	0 0 0 0 1 0
2	4 → 5	0 0 0 0 1 1
3	1	0 1 0 0 0 0
4	3	0 0 0 1 0 0
5	5	0 0 0 0 0 1

	LISTE DI ADIACENZA	MATRICE DI ADIACENZA
spazio	$O(V + E)$	$O(V ^2)$
l'arco (u, v) esiste?	$O(\text{grado di } u)$	$O(1)$
elencare i vicini di u	$O(\text{grado di } u)$	$O(V)$
visita BFS o DFS	$O(V + E)$	$O(V ^2)$
adatta a	grafi sparsi	grafi densi o piccoli

Una realizzazione a oggetti delle liste di adiacenza usa tre classi: Edge (un elemento della lista, con il nodo di arrivo e il puntatore al successivo), Node (la testa di una lista, con un metodo AddEdge che inserisce in testa) e Graph (un array dinamico di Node, uno per nodo). Il programma seguente usa la stessa struttura in forma compatta, con l'inserimento **in coda** alla lista, così che i vicini vengano esaminati in ordine crescente come nelle tabelle.

La visita in ampiezza (BFS)

La **visita in ampiezza** (*Breadth First Search*) parte da un nodo sorgente s ed esplora il grafo a **cerchi concentrici**: prima tutti i nodi a distanza 1 da s , poi tutti quelli a distanza 2, e così via, come le onde prodotte da un sasso in uno stagno. Scopre così ogni nodo raggiungibile da s e ne calcola la **distanza** $d[v]$, cioè il numero minimo di archi da attraversare per arrivarci. Lo strumento che garantisce l'ordine «per distanza» è una **coda** (FIFO): i nodi scoperti per primi sono i primi a essere esaminati.

Per non visitare due volte lo stesso nodo (in un grafo, a differenza di un albero, a un nodo si può arrivare da più strade, e i cicli porterebbero a girare per sempre), ogni nodo ha un **colore**. **Bianco**: non ancora scoperto. **Grigio**: scoperto e messo in coda, ma i suoi vicini non sono ancora stati esaminati; i nodi grigi sono la **frontiera** tra ciò che è stato scoperto e ciò che non lo è. **Nero**: scoperto e con tutti i vicini esaminati. All'inizio tutti i nodi sono bianchi, tranne s che è grigio, ha $d[s] = 0$ ed è in coda. Poi, finché la coda non è vuota, si estrae un nodo u e si scorre la sua lista di adiacenza: ogni vicino v **bianco** diventa grigio, riceve $d[v] = d[u] + 1$ e **pred[v] = u** (il nodo da cui è stato scoperto), e viene accodato. Finita la lista, u diventa nero.

```
BFS(G, s)
  for each u in V - {s}: color[u] = WHITE; d[u] = ∞; pred[u] = NULL
  color[s] = GRAY; d[s] = 0; pred[s] = NULL; Q.Enqueue(s)
  while Q non vuota:
    u = Q.Dequeue()
    for each v in Adj[u]:
      if color[v] == WHITE:
        color[v] = GRAY; d[v] = d[u] + 1; pred[v] = u; Q.Enqueue(v)
    color[u] = BLACK
```

ESERCIZIO

BFS da 0 sul grafo della Figura 9.1

Coda iniziale: [0], con $d[0] = 0$. Si estrae **0**: i vicini 1 e 3 sono bianchi, diventano grigi con $d = 1$ e $\text{pred} = 0$; coda [1, 3]. Si estrae **1**: il vicino 4 è bianco, $d[4] = 2$, $\text{pred}[4] = 1$; coda [3, 4]. Si estrae **3**: il vicino 1 è già grigio o nero, non succede nulla; coda [4]. Si estrae **4**: il vicino 3 è già nero; la coda è vuota e la visita finisce. Ordine di visita: **0, 1, 3, 4**, con distanze 0, 1, 1, 2. I nodi **2 e 5 restano bianchi**: non sono raggiungibili da 0 (gli archi tra loro e il resto escono da 2, non vi entrano).

I puntatori pred formano un **albero della visita in ampiezza** (*breadth first tree*) con radice s , che contiene tutti i nodi raggiungibili: il cammino da s a v in questo albero è un **cammino minimo** da s a v . Per stamparlo si risale da v seguendo pred fino a s , con una procedura ricorsiva che stampa prima il cammino fino al predecessore e poi v : costa $O(|V|)$, perché ogni chiamata riguarda un cammino con un nodo in meno. Nell'esempio il cammino minimo da 0 a 4 è $0 \rightarrow 1 \rightarrow 4$.

La **complessità** della BFS con liste di adiacenza è $O(|V| + |E|)$, cioè **lineare nella dimensione del grafo**. Ogni nodo entra in coda al più una volta (quando passa da bianco a grigio), quindi le operazioni sulla coda costano $O(|V|)$ in tutto; la lista di adiacenza di ogni nodo viene scorsa una sola volta, quando il nodo esce dalla coda, quindi in totale si esaminano $O(|E|)$ archi. Con la matrice di adiacenza, per trovare i vicini di ogni nodo bisogna leggerne un'intera riga, e la complessità sale a $O(|V|^2)$.

La visita in profondità (DFS)

La **visita in profondità** (*Depth First Search*) esplora il grafo **andando il più avanti possibile** lungo un cammino prima di tornare indietro, come chi esplora un labirinto seguendo sempre il primo corridoio che trova e, arrivato a un vicolo cieco, torna all'ultimo bivio per provare il corridoio successivo (*backtracking*). Gli archi vengono esplorati a partire dal nodo **scoperto più di recente** che abbia ancora archi uscenti inesplorati. Al posto della coda della BFS si usa (implicitamente, tramite la ricorsione) una **pila**: lo stack delle chiamate.

La DFS usa gli stessi tre colori e registra per ogni nodo due **marche temporali**, con un contatore *time* che avanza a ogni evento: **d[u]**, il tempo di **scoperta** (quando *u* passa da bianco a grigio), e **f[u]**, il tempo di **fine** (quando tutta la lista di adiacenza di *u* è stata esplorata e *u* diventa nero). Per ogni nodo vale $d[u] < f[u]$. A differenza della BFS, la DFS qui presentata non parte da un solo nodo: la procedura principale scorre **tutti** i nodi e lancia una nuova visita da ciascuno che sia ancora bianco. Il grafo dei predecessori non è quindi un albero ma una **foresta** (*depth first forest*), con un albero per ogni nuova partenza.

```
DFS(G)
  for each u in V:
    color[u] = WHITE; pred[u] = NULL
  time = 1
  for each u in V:
    if color[u] == WHITE:
      DFS-Visit(u)

DFS-Visit(u)
  color[u] = GRAY
  d[u] = time++
  for each v in Adj[u]:
    if color[v] == WHITE:
      pred[v] = u
      DFS-Visit(v)
  color[u] = BLACK
  f[u] = time++
```

ESERCIZIO

DFS sul grafo della Figura 9.1

Si parte da **0** ($d = 1$). Il primo vicino bianco è 1: si scende in **1** ($d = 2$), poi nel suo vicino **4** ($d = 3$), poi nel vicino di 4, **3** ($d = 4$). L'unico vicino di 3 è 1, già grigio: 3 è finito ($f = 5$). Si torna a 4, che non ha altri vicini ($f = 6$), poi a 1 ($f = 7$), poi a 0, il cui secondo vicino 3 è ormai nero ($f = 8$). Il primo albero della foresta è finito. La procedura principale trova bianco il nodo **2** ($d = 9$): il vicino 4 è nero, il vicino **5** è bianco ($d = 10$); il self loop di 5 punta a un nodo grigio, quindi 5 finisce ($f = 11$) e poi 2 ($f = 12$). Il risultato coincide con quello di `p9_grafo.cpp`: d/f valgono 1/8, 2/7, 9/12, 4/5, 3/6, 10/11 per i nodi da 0 a 5.

Anche la DFS costa $\Theta(|V| + |E|)$ con liste di adiacenza: la procedura principale esamina ogni nodo una volta, `DFS-Visit` viene chiamata una sola volta per nodo (solo sui bianchi, che diventano subito grigi) e scorre una volta la sua lista. La DFS è la base di molti altri algoritmi, per esempio quello che calcola le **componenti fortemente connesse** di un grafo orientato con due visite in profondità: con le liste di adiacenza è lineare, con la matrice di adiacenza diventa **quadratico in $|V|$** .

```
// Grafo orientato con liste di adiacenza: visite in ampiezza (BFS) e profondita' (DFS)
#include <iostream>
using namespace std;
struct Arco {
    int dest;
    Arco* succ;
};
enum Colore { BIANCO, GRIGIO, NERO };

class Grafo {
public:
    Grafo(int nodi) : n(nodi) {
        adj = new Arco*[n];
        for (int i = 0; i < n; i++) adj[i] = nullptr;
    }
    // elemento di una lista di adiacenza
    // nodo di arrivo
    // un puntatore di testa per nodo
```

```

        colore = new Colore[n];
        d = new int[n];
        f = new int[n];
        pred = new int[n];
    }
    ~Grafo() {
        for (int i = 0; i < n; i++) // libera ogni lista
            while (adj[i]) { Arco* a = adj[i]; adj[i] = a->succ; delete a; }
        delete[] adj; delete[] colore; delete[] d; delete[] f; delete[] pred;
    }
    void aggiungiArco(int u, int v) { // arco u -> v, in coda alla lista di u
        Arco* nuovo = new Arco{v, nullptr};
        if (adj[u] == nullptr) { adj[u] = nuovo; return; }
        Arco* p = adj[u];
        while (p->succ) p = p->succ;
        p->succ = nuovo;
    }
    void stampa() const {
        for (int u = 0; u < n; u++) {
            cout << u << ":";
            for (Arco* a = adj[u]; a; a = a->succ) cout << " " << a->dest;
            cout << endl;
        }
    }
    void bfs(int s);
    void dfs();
    void cammino(int s, int v) const; // da usare dopo bfs(s)
private:
    void dfsVisita(int u);
    int n; // numero di nodi
    Arco** adj; // array di liste di adiacenza
    Colore* colore; // stato di ogni nodo durante la visita
    int* d; // BFS: distanza da s; DFS: tempo di scoperta
    int* f; // DFS: tempo di fine
    int* pred; // predecessore nell'albero di visita (-1 = nessuno)
    int tempo;
};

void Grafo::bfs(int s) {
    for (int u = 0; u < n; u++) { colore[u] = BIANCO; d[u] = -1; pred[u] = -1; }
    int* coda = new int[n]; // coda FIFO: ogni nodo entra al piu' una volta
    int testa = 0, fondo = 0;
    colore[s] = GRIGIO; d[s] = 0;
    coda[fondo++] = s; // enqueue(s)
    cout << "BFS da " << s << ":";
    while (testa < fondo) { // finche' la coda non e' vuota
        int u = coda[testa++]; // dequeue
        cout << " " << u << "(d=" << d[u] << ")";
        for (Arco* a = adj[u]; a; a = a->succ) { // tutti i vicini di u
            int v = a->dest;
            if (colore[v] == BIANCO) { // v scoperto ora
                colore[v] = GRIGIO;
                d[v] = d[u] + 1;
                pred[v] = u;
                coda[fondo++] = v; // enqueue(v)
            }
        }
        colore[u] = NERO; // lista di u esaminata tutta
    }
    cout << endl;
}

```

```

delete[] coda;
}

void Grafo::cammino(int s, int v) const {           // stampa il cammino minimo s -> v
    if (v == s) cout << s;
    else if (pred[v] == -1) cout << "nessun cammino da " << s << " a " << v;
    else { cammino(s, pred[v]); cout << " -> " << v; }
}

void Grafo::dfs() {
    for (int u = 0; u < n; u++) { colore[u] = BIANCO; pred[u] = -1; }
    tempo = 1;
    for (int u = 0; u < n; u++)                    // riparte da ogni nodo non ancora visitato
        if (colore[u] == BIANCO) dfsVisita(u);
    for (int u = 0; u < n; u++)
        cout << "nodo " << u << ": d=" << d[u] << " f=" << f[u]
            << " pred=" << pred[u] << endl;
}

void Grafo::dfsVisita(int u) {
    colore[u] = GRIGIO;
    d[u] = tempo++;                                // scoperta
    for (Arco* a = adj[u]; a; a = a->succ) {
        int v = a->dest;
        if (colore[v] == BIANCO) {
            pred[v] = u;
            dfsVisita(v);                          // si scende subito in profondita'
        }
    }
    colore[u] = NERO;
    f[u] = tempo++;                                // fine: tutti i vicini esplorati
}

int main() {
    Grafo g(6);
    int archi[8][2] = {{0,1}, {0,3}, {1,4}, {2,4}, {2,5}, {3,1}, {4,3}, {5,5}};
    for (int i = 0; i < 8; i++) g.aggiungiArco(archi[i][0], archi[i][1]);
    g.stampa();
    g.bfs(0);
    g.cammino(0, 4); cout << endl;
    g.cammino(0, 5); cout << endl;
    g.dfs();
    return 0;
}

```

VERIFICA

Domande sui grafi

Qual è la complessità asintotica della visita in profondità in un grafo con N nodi e M archi implementato con liste di adiacenza?

Lineare in $M + N$, cioè $O(N + M)$. Lo stesso vale per la visita in ampiezza.

Qual è la complessità della procedura per il calcolo delle componenti fortemente connesse di un grafo orientato con V nodi ed E archi, rappresentato con matrice di adiacenza?

Quadratica in V , $O(V^2)$: l'algoritmo si basa su visite in profondità, e con la matrice ogni visita deve leggere tutte le V righe di V elementi, indipendentemente dal numero di archi.

Qual è il numero minimo di archi da attraversare per effettuare una visita in ampiezza in un grafo con N nodi e M archi?

La risposta è **M** : per essere sicura di scoprire tutti i nodi raggiungibili, la BFS deve esaminare tutte le liste di adiacenza, cioè tutti gli archi (in un grafo non orientato ogni arco compare in due liste ed è esaminato due volte). Se invece si contano solo gli archi che portano a un nodo nuovo, quelli dell'albero della visita, sono al più $N - 1$.

Che differenza c'è tra BFS e DFS?

La BFS usa una **coda** ed esplora per livelli di distanza dalla sorgente: calcola i **cammini minimi** (in numero di archi) e produce un albero. La DFS usa la **ricorsione** (cioè una pila) e scende il più possibile prima di tornare indietro: registra i tempi di scoperta e di fine e produce una foresta. Entrambe costano $O(|V| + |E|)$ con liste di adiacenza. Negli alberi, la visita per livelli è una BFS e le visite pre/in/postorder sono DFS.