Capitolo 4

Strutture dati

> *Picking the wrong data structure for the job can be disastrous in terms of performance. Identifying the very best data structure is usually not as critical, because there can be several choices that perform similarly.*
>
> Steven S. Skiena, *The Algorithm Design Manual*

4.1 Strutture dati astratte

Alcune definizioni

Definizione 4.1.1 (Tipo di dato)**.** In un linguaggio di programmazione, un dato è un valore che una variabile può assumere.

Definizione 4.1.2 (Tipo di dato astratto)**.** Un modello matematico, dato da una collezione di valori e un insieme di operazioni ammesse su questi valori.

Definizione 4.1.3 (Tipi di dato primitivi)**.** Sono dei tipi di dati che vengono forniti direttamente dal linguaggio. Come ad esempio: int (`+,-,*,/, %`), boolean (`!, &&, ||`).

Ogni tipo di dato deve distinguere *specifica* ed *implementazione* di un tipo di dato astratto. La *specifica* è astratta, il "manuale d'uso" che nasconde i dettagli implementativi all'utente, mentre l'*implementazione* è la realizzazione vera e propria del tipo di dato.

Tabella 4.1: Differenza fra specifica ed implementazione

Specifica	Implementazione
Numeri reali	IEEE-754
Pile	Pile basate su vettori Pile basate su puntatori
Code	Code basate su vettori circolari Code basate su puntatori

Definizione 4.1.4 (Strutture di dati)**.** Le strutture di dati sono collezioni di dati, caratterizzate più dall'organizzazione della collezione piuttosto che dal tipo dei dati in esse contenute.

Le strutture dati sono un modo sistematico per organizzare i dati e su di esse sono definite un insieme di operatori che permettono di manipolare la struttura stessa. Le strutture dati possono essere caratterizzate in vari modi:

- *lineari*/*non lineari*: presentano (o meno) una sequenza al loro interno;
- *statiche*/*dinamiche*: possono variare (o meno) di dimensione o di contenuto;
- *omogenee*/*disomogenee*: si riferisce ai dati contenuti al loro interno.

Tabella 4.2: Implementazione delle strutture dati nei vari linguaggi.
Nota che Java distingue chiaramente la specifica dall'implementazione

Tipo	Java	C++	Python
Sequenze	`List`, `Queue`, `Deque`, `LinkedList`, `ArrayList`, `Stack`, `ArrayDeque`	`list`, `forward_list`, `vector`, `stack`, `queue`, `dequeue`	`list`, `tuple`
Insiemi	`Set`, `TreeSet`, `HashSet`, `LinkedHashSet`	`set`, `unordered_set`	`set`, `fronzenset`
Dizionari	`Map`, `HashTree`, `HashMap`, `LinkedHashMap`	`map`, `unordered_map`	`dict`
Alberi	-	-	-
Grafi	-	-	-

4.2 Sequenza

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

Le operazioni ammesse su una sequenza sono:

- L'aggiunta e la rimozione elementi, specificando la posizione (tipicamente un intero), l'elemento s_1 si trova in posizione pos_i ed esistono posizioni fittizie pos_0 e pos_{n+1};
- Accesso diretto alla testa e coda;
- Accesso sequenziale a tutti gli altri elementi.

Algoritmo 1: Specifica SEQUENCE

Una struttura dati *dinamica*, *lineare* che rappresenta una sequenza *ordinata* di valori, dove lo stesso valore può comparire più volte.

Sequence

```
// INTERPRETARE
boolean isEmpty      // true se la sequenza è vuota
boolean finished     // true se p è uguale a pos_0 o a
 pos_{n+1}

// LEGGERE
Pos head             // posizione del primo elemento
Pos tail             // posizione dell'ultimo elemento

// ITERARE
Pos next             // posizione dell'elem. che segue p
Pos prev             // posizione dell'elem. che precede p

// MODIFICA
// inserisce l'elemento di tipo Item nella
   posizione p,
// ritorna la nuova posizione,
// che diviene il predecessore di p
Pos insert(Pos p, Item v)
// rimuove l'elemento contenuto nella pos. p,
// ritorna il successore di p
Pos remove(Pos p)

// legge l'elemento di tipo Item
// contenuto nella posizione p
read(Pos p)

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

4.2.1 Implementazione delle sequenze

Di seguito vengono presentati alcuni esempi d'utilizzo dell'implementazione delle sequenze nei diversi linguaggi di programmazione utilizzati oggigiorno.

Codice 4.1: Implementazione delle liste in Java

```
List<String> lista = new LinkedList<String>();
lista.add("two");
lista.addFirst("one");
lista.addLast("three");

Result: [ "one", "two", "three" ]
```

Codice 4.2: Implementazione delle liste in C++

```
std::list<int> lista;
lista.push_front(2);
lista.push_front(1);
lista.push_back(3);

Result: [1,2,3]
```

Codice 4.3: Implementazione delle liste in Python

```
lista = ["one", "three"]
lista.insert(1, "two")

Result: [ 'one', 'two', 'three' ]
```

4.3 Insiemi

Un insieme è una struttura dati *dinamica*, *non lineare* che memorizza una *collezione non ordinata di elementi* senza valori ripetuti. L'ordinamento fra elementi è dato dall'eventuale relazione d'ordine definita sul tipo degli elementi stessi.

Le operazioni ammesse su un'insieme sono:

- operazioni di base: come inserimento, cancellazione e verifica di contenimento;
- operazione di ordinamento: massimo, minimo;
- operazioni insiemistiche: unione, intersezione, differenza;
- iteratori: effettuare operazione per ogni elemento contenuto nell'insieme.

Algoritmo 1: Struttura dati SET

Una struttura dati *dinamica*, *non lineare* che memorizza una *collezione non ordinata di elementi* senza valori ripetuti.

```
Set
// INTERPRETARE
int size                    // cardinalità dell'insieme
boolean contains            // true se x è contenuto
// OPERAZIONI DI BASE
// inserisce x nell'insieme, se assente
insert(ITEM k)
// rimuove x nell'insieme, se presente
remove(ITEM k)
// OPERAZIONI INSIEMISTICHE
static SET union(SET A, SET B)
static SET intersection(SET A, SET B)
static SET difference(SET A, SET B)
```

Codice 4.4: Implementazione degli insiemi in Java

```
List<String> lista = new LinkedList<String>();
Set<String> docenti = new TreeSet<>();
docenti.add("Alberto");
docenti.add("Cristian");
docenti.add("Alessio");

Result: { "Alberto", "Alessio", "Cristian" }
```

Codice 4.5: Implementazione degli insiemi in C++

```
std::set<std::string> frutta;
frutta.insert("mele");
frutta.insert("pere");
frutta.insert("banane");
frutta.insert("mele");
frutta.remove("mele")

Result: { "banane", "pere" }
```

Codice 4.6: Implementazione degli insiemi in Python

```
items = { "rock", "paper", "scissors", "rock" }
print(items)
print("Spock" in items)
print("lizard" not in items)

Result: { "rock", "paper", "scissors" }
False
True
```

4.4 Dizionari

Un dizionario è una struttura dati che rappresenta il concetto matematico di *relazione univoca* $R : D \to C$, o associazione chiave-valore, dove:

- l'insieme D è il dominio (gli elementi sono detti *chiavi*);
- l'insieme C è il codominio (gli elementi sono detti *valori*).

Le operazioni ammesse sui dizionari sono:

- ottenere il valore associato ad una particolare chiave (se presente) o **nil** se assente;
- inserire una nuova associazione chiave-valore, cancellando eventuali associazioni precedenti per la stessa chiave;
- rimuovere un'associazione chiave-valore esistente.

Algoritmo 1: Specifica dizionario

Un dizionario è una struttura dati che rappresenta il concetto matematico di *relazione univoca* o associazione chiave-valore.

DICTIONARY

ITEM lookup(ITEM k)	// restituisce il valore associato alla chiave k, **nil** altrimenti
ITEM insert(KEY k, ITEM v)	// associa il valore v alla chiave k
remove(KEY k)	// rimuove l'associazione della chiave k

Codice 4.7: Implementazione dei dizionari in Java

```
Map<String, String> capoluoghi = new HashMap<>();
capoluoghi.put("Toscana", "Firenze");
capoluoghi.put("Lombardia", "Milano");
capoluoghi.put("Sardegna", "Cagliari");
```

Codice 4.8: Implementazione dei dizionari in C++

```
std::map<std::string, int> wordcounts;
std::string s;

while (std::cin >> s && s != "end")
  ++wordcounts[s];
```

Codice 4.9: Implementazione dei dizionari in Python

```
v = {}
v[10] = 5
v["alberto"] = 42
v[10]+v["alberto"]

Result: 47
```

4.5 Alberi

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

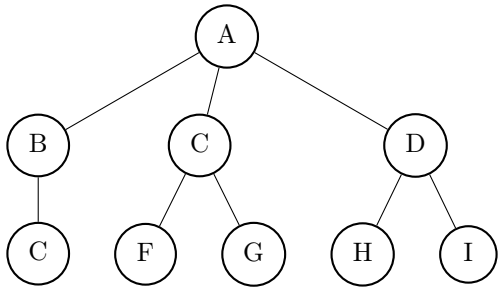

Figura 4.1: Un albero

Non vedremo implementazioni nei vari linguaggi in quanto non esiste una struttura dati definita riconosciuta universalmente.

4.6 Grafi

La struttura dati grafo è composta da:

- un insieme di elementi detti nodi o vertici;
- un insieme di coppie (ordinate oppure no) di nodi detti archi.

Tutte le operazioni su alberi e grafi ruotano attorno alla possibilità di effettuare visite su di essi, vedremo la specifica completa più avanti.

Nota. La scelta della struttura dati si riflette sull'efficienza e sulle operazioni ammesse.

4.7 Implementazione strutture dati elementari

4.7.1 Lista

Una lista è una sequenza di nodi, contenenti dati arbitrari e 1-2 puntatori all'elemento successivo e/o precedente.

La contiguità nella lista non implica che ci sia continuità nella memoria. Tutte le operazioni effettuate sulla lista hanno complessità $\mathcal{O}(1)$, ma per fare una ricerca dobbiamo spendere $\mathcal{O}(n)$.

Esistono diverse implmentazioni della lista, le quali possono essere:

- bidirezionale o monodirezionale;
- con sentinella o senza;
- circolare o non circolare.

Algoritmo 1: Struttura dati lista bidirezionale con sentinella in pseudocodice

```
LIST                          // bidirezionale con sentinella
LIST pred                     // predecessore
LIST succ                     // successore
LIST value                    // elemento

LIST List
    // la sentinella fa riferimento a sé stessa
    t.pred = t
    t.succ = t
    return t

POS head
    return succ

POS tail
    return pred

POS next
    return p.succ

POS prev
    return p.pred

boolean finished(POS p)
    return p = this

ITEM read(POS p)
    return p.value

write(POS p)
    return p.value

// posso fare queste operazioni essendo sicuro
// di avere sempre un predecessore
POS insert(POS p, ITEM v)
    LIST t = List t.value = v
    t.pred = p.pred
    p.pred.succ = t
    t.succ = p
    p.pred = t
    return p

POS remove(POS p)
    p.pred.succ = p.succ
    p.succ.pred = p.pred
    LIST t = p.succ
    delete p
    return t
```

Il costo delle operazioni di lettura, scrittura, inserimento e rimozione per questa struttura è $\mathcal{O}(1)$.

Codice 4.10: Lista bidirezionale *senza* sentinella in Java

```
class Pos {
    Pos succ;    /** Prossimo elemento della lista */
    Pos pred;    /** Precedente elemento della lista */
    Object v;    /** Valore */

    Pos(Object v) {
        succ = pred = null;
        this.v = v;
    }
}

public class List {
    private Pos head;        /** Primo elemento della lista */
    private Pos tail;        /** Ultimo elemento della lista */

    public List() {
        head = tail = null;
    }

    public Pos head()                  { return head; }
    public Pos tail()                  { return tail; }
    public boolean finished(Pos pos)   { return pos == null; }
    public boolean isEmpty()           { return head == null; }
    public Object read(Pos p)          { return p.v; }
    public void write(Pos p, Object v) { p.v = v; }

    public Pos next(Pos pos) {
        return (pos != null ? pos.succ : null);
    }

    public Pos prev(Pos pos) {
        return (pos != null ? pos.pred : null);
    }

    public void remove(Pos pos) {
        if (pos.pred == null) // sto inserendo in testa
            head = pos.succ;
        else
            pos.pred.succ = pos.succ;

        if (pos.succ == null) // sto inserendo in coda
            tail = pos.pred;
        else
            pos.succ.pred = pos.pred;
    }

    public Pos insert(Pos pos, Object v) {
        Pos t = new Pos(v);

        if (head == null) {
            head = tail = t; // Inserisci in una lista vuota
        } else if (pos == null) {
            t.pred = tail; // Inserisci alla fine
            tail.succ = t;
            tail = t;
        } else {
            t.pred = pos.pred; // Iserimento davanti ad una posizione esistente
            if (t.pred != null)
                t.pred.succ = t;
            else
                head = t;

            t.succ = pos;
            pos.pred = t;
```

Figura 4.2: `xkcd no. 379`

4.7.2 Pila

La pila è una struttura dati *dinamica*, *lineare* in cui l'elemento rimosso dall'operazione di cancellazione è predeterminato, ed è quello che "è rimasto per meno tempo nell'insieme" (con strategia LIFO, *Last-In-First-Out*).

Algoritmo 1: Specifica STACK

```
boolean isEmpty                          // restituisce vero se la pila è vuota
push(ITEM v)                             // inserisce v in cima alla pila
ITEM pop    // estrae l'elemento in cima alla pila e lo restituisce al chiamante
ITEM top                                 // legge l'elemento in cima alla pila
```

Ogni volta che viene effettuata una chiamata a funzione si usa implicitamente una pila, che memorizza tutti i record di attivazione delle chiamate effettuate. Sfrutteremo questo meccanismo implicito per visitare gli alberi, attraverso una visita in profondità.

Le pile possono essere implementate come:

- liste bidirezionali, dove il puntatore punta all'elemento top (non utilizzate);
- tramite vettore, dove la dimesione è limitata quindi si crea un *overhead* più basso.

Algoritmo 1: Struttura dati pila basata su vettore in pseudocodice

```
ITEM[] A                                 // elementi
int n                                    // cursore
int m                                    // dimesione massima

// crea una pila vuota
STACK Stack(int dim)
    STACK t = new STACK
    t.A = new int[0...dim-1]
    t.m = dim
    t.n = 0
    return t

// leggi l'elemento in cima alla pila
ITEM top
    precondition: n > 0
    return A[n]

// restituisce true se la pila è vuota
boolean isEmpty
    return n == 0

// estrae l'elemento in cima alla pila e lo
   restituisce al chiamante
ITEM pop
    precondition: n > 0
    ITEM t = A[n]
    n++
    return t

// inserisce v in cima alla pila
push(ITEM v)
    precondition: n < m
    n++
    A[n] = v
```

Codice 4.11: Pila basata su vettore circolare in Java

```java
public class VectorStack implements Stack {

    /** Vector containing the elements */
    private Object[] A;

    /** Number of elements in the stack */
    private int n;

    public VectorStack(int dim) {
        n = 0;
        A = new Object[dim];
    }

    public boolean isEmpty() {
        return n == 0;
    }

    public Object top() {
        if (n == 0)
            throw new IllegalStateException("Stack is empty");

        return A[n-1];
    }

    public Object pop() {
        if (n == 0)
            throw new IllegalStateException("Stack is empty");

        return A[--n];
    }

    public void push(Object o) {
        if (n == A.length)
            throw new IllegalStateException("Stack is full");

        A[n++] = o;
    }

}
```

4.7.3 Coda

La coda è una struttura dati *dinamica lineare* in cui l'elemento rimosso dall'operazione di cancellazione è predeterminato, ed è quello che "è rimasto per più tempo nell'insieme" (con strategia, FIFO, *First-In-First-Out*).

Algoritmo 1: Specifica QUEUE

```
boolean isEmpty                                  // restituisce vero se la coda è vuota
ITEM enqueue(ITEM v)                             // inserisce v in fondo alla coda
ITEM dequeue          // estrae l'elemento in cima alla coda e lo restituisce al chiamante
ITEM top                                         // legge l'elemento in testa alla coda
```

Nei sistemi operativi, i processi in attesa di utilizzare una risorsa vengono gestiti tramite una coda. La politica FIFO è onesta (*fair*) rispetto l'ordine in cui i processi sono stati inseriti.

Le code possono essere implementate come:

- liste monodirezionali, dove sono presenti due puntatori: uno alla testa (*head*) per l'estrazione, ed uno alla coda per l'inserimento;
- vettori circolari, il quale ha una dimensione limitata e crea un *overhead* più basso.

Algoritmo 1: Struttura dati coda basata su vettore circolare in pseudocodice

```
ITEM[] A                                         // elementi
int n                                            // dimensione attuale
int testa                                        // testa
int m                                            // dimesione massima

// crea una cosa vuota
QUEUE Queue(int dim)
    QUEUE t = new QUEUE
    t.A = new int[0...dim − 1]
    t.m = dim
    t.testa = 0
    t.n = 0
    return t

// legge l'elemento in testa alla coda
ITEM top
    precondition: n > 0
    return A[testa]

// restituisce true de la coda è vuota
ITEM isEmpty
    return n == 0

// estrae l'elemento in testa alla coda e lo
   restituisce al chiamante
ITEM dequeue
    precondition: n > 0
    ITEM t = A[testa]
    testa = (testa + 1) mod m
    n++
    return t

// inserisce v in fondo alla coda
ITEM enqueue
    precondition: n < m
    A[(testa + n) mod m] = v
    n++
```

Codice 4.12: Coda basata su vettore in Java

```
public class VectorQueue implements Queue {

    /** Element vector */
    private Object[] A;

    /** Current number of elements in the queue */
    private int n;

    /** Top element of the queue */
    private int head;

    public VectorQueue(int dim) {
        n = 0;
        head = 0;
        A = new Object[dim];
    }

    public boolean isEmpty() {
        return n == 0;
    }

    public Object top() {
        if (n == 0)
            throw new IllegalStateException("Queue is empty");

        return A[head];
    }

    public Object dequeue() {
        if (n == 0)
            throw new IllegalStateException("Queue is empty");

        Object t = A[head];
        head = (head+1) % A.length;
        n = n-1;
        return t;
    }

    public void enqueue(Object v) {
        if (n == A.length)
            throw new IllegalStateException("Queue is full");

        A[(head+n) % A.length] = v;
        n = n+1;
    }
}
```