

4 Strutture dati

4.1 Strutture dati astratte

“ 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*

Alcune definizioni

Definizione 4.1 (Tipo di dato). In un linguaggio di programmazione, un dato è un valore che una variabile può assumere.

Definizione 4.2 (Tipo di dato astratto). Un modello matematico, dato da una collezione di valori e un insieme di operazioni ammesse su questi valori.

Definizione 4.3 (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 1: Differenza fra specifica ed implementazione

Specifiche	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 4.4 (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 2: Implementazione delle strutture dati nei vari linguaggi.
Nota che Java distingue chiaramente la specifica dall'implementazione

Tipo	Java	C++	Python
Sequenze	<code>List</code> , <code>Queue</code> , <code>Deque</code> , <code>LinkedList</code> , <code>ArrayList</code> , <code>Stack</code> , <code>ArrayDeque</code>	<code>list</code> , <code>forward_list</code> , <code>vector</code> , <code>stack</code> , <code>queue</code> , <code>dequeue</code>	<code>list</code> , <code>tuple</code>
Insiemi	<code>Set</code> , <code>TreeSet</code> , <code>HashSet</code> , <code>LinkedHashSet</code>	<code>set</code> , <code>unordered_set</code>	<code>set</code> , <code>frozensest</code>
Dizionari	<code>Map</code> , <code>HashTree</code> , <code>HashMap</code> , <code>LinkedHashMap</code>	<code>map</code> , <code>unordered_map</code>	<code>dict</code>
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 fintizie pos_0 e pos_{n+1} ;
- Accesso diretto alla testa e coda;
- Accesso sequenziale a tutti gli altri elementi.

Specifico SEQUENCE	
Una struttura dati <i>dinamica, lineare</i> che rappresenta una sequenza <i>ordinata</i> di valori, dove lo stesso valore può comparire più volte.	
Sequence	
// INTERPRETARE	
bool isEmpty	// vero se la sequenza è vuota
bool finished	// vero 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 nei diversi linguaggi

Codice 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 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 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.

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  
bool contains     // vero 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: 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 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 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
```

4.3.1 Dizionari

Un dizionario è una struttura dati che rappresenta il concetto matematico di *relazione univoca* $R : D \rightarrow C$, 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.

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 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 8: Implementazione dei dizionari in C++

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

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

Codice 9: Implementazione dei dizionari in Python

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

Result: 47
```

4.3.2 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.

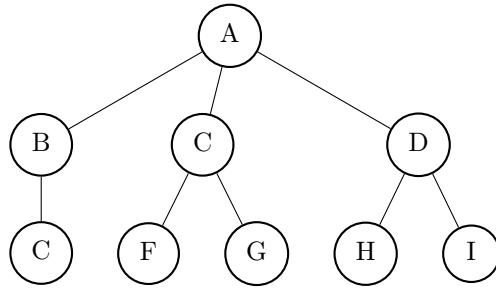


Figura 1: Un albero

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

4.3.3 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 ha riflessi sull'efficienza e sulle operazioni ammesse.

4.4 Implementazione strutture dati elementari

4.4.1 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.

Struttura dati lista bidirezionale con sentinella in pseudocodice

LIST	<i>// bidirezionale con sentinella</i>	ITEM read(Pos p)
LIST pred	<i>// predecessore</i>	└ return p.value
LIST succ	<i>// successore</i>	write(Pos p)
LIST value	<i>// elemento</i>	└ return p.value
LIST List		<i>// posso fare queste operazioni essendo sicuro</i>
	<i>// la sentinella fa riferimento a sé stessa</i>	<i>// di avere sempre un predecessore</i>
	t.pred = t	POS insert(Pos p, ITEM v)
	t.succ = t	LIST t = List t.value = v
	return t	t.pred = p.pred
POS head		p.pred.succ = t
	└ return succ	t.succ = p
POS tail		p.pred = t
	└ return pred	return p
POS next		POS remove(Pos p)
	└ return p.succ	p.pred.succ = p.succ
POS prev		p.succ.pred = p.pred
	└ return p.pred	LIST t = p.succ
bool finished(Pos p)		delete p
	└ return p = this	return t

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

4.4.2 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*).

Codice 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;
        }
    }

    return t;
}
```



Figura 2: xkcd no. 379

Specifica STACK

```

bool isEmpty                                // restituisce vero se la pila è vuota
push(ITEM v)                                  // inserisce v in cima alla pila
ITEM pop                                     // estraе 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隐式 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 dimensione è limitata quindi si crea un *overhead* più basso.

```

Struttura dati pila basata su vettore in pseudocodice

ITEM[] A           // elementi
int n             // cursore
int m             // dimensione 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 vero se la pila è vuota
bool isEmpty
    return n == 0

// estraie 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 11: Pila basata su vettore circolare in 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;  
    }  
}
```

4.4.3 Coda

La coda è una struttura dai *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*).

Specifiche QUEUE	
bool isEmpty	// restituisce vero se la coda è vuota
ITEM enqueue(ITEM v)	// inserisce v in fondo alla coda
ITEM dequeue	// estraie 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.

Struttura dati coda basata su vettore circolare in pseudocodice		
ITEM[] A	// elementi	// restituisce vero se la coda è vuota
int n	// dimensione attuale	ITEM isEmpty
int testa	// testa	return n==0
int m	// dimensione massima	
// crea una cosa vuota		// estraie l'elemento in testa alla coda e lo restituisce al chiamante
QUEUE Queue(int dim)		ITEM dequeue
QUEUE t = new QUEUE		precondition: n > 0
t.A = new int[0...dim - 1]		ITEM t = A[testa]
t.m = dim		testa = (testa + 1) mod m
t.testa = 0		n++
t.n = 0		
return t		return t
// legge l'elemento in testa alla coda		
ITEM top		// inserisce v in fondo alla coda
precondition: n > 0		ITEM enqueue
return A[testa]		precondition: n < m
		A[(testa + n) mod m] = v
		n++

Codice 12: Pila 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;
}

```
