1. Strutture di dati elementari

Come discusso nel capitolo precedente, le sequenze sono uno dei tipi di dato astratto più importanti nella programmazione e nella progettazione di algoritmi. Molto spesso i dati di cui un programma ha bisogno vengono analizzati in sequenza, uno dopo l'altro.

Esempio (Il catalogo di Don Giovanni) Il famoso catalogo delle conquiste di Don Giovanni comprende italiane, tedesche, spagnole, francesi e turche. Tra queste ci sono contadine, cameriere, cittadine, duchesse, baronesse, marchesine e principesse. Il fedele servitore Leporello ha strutturato il catalogo come una sequenza ordinata per data di conquista; vorrebbe ora cancellare dal catalogo tutte le duchesse e copiare in un'altra sequenza L tutte le spagnole, mantenendone lo stesso ordine che avevano nel catalogo.

Nella procedura donGiovanni() sono utilizzate due posizioni p e q per scandire, rispettivamente, le sequenze L e *catalogo*. L è inizializzata alla sequenza vuota, q alla prima posizione del catalogo, e p alla prima posizione di L. Poiché all'inizio L è vuota, $p = pos_0 = pos_{|L|+1}$, la posizione che segue l'ultimo elemento. All'interno del ciclo **while**, la posizione p non viene mai modificata; viene quindi mantenuta sempre uguale a quella che segue l'ultimo elemento di L, in modo da inserire un nuovo elemento in fondo alla sequenza. In questo modo, gli elementi inseriti in L mantengono lo stesso ordine che avevano nel catalogo. Nel ciclo **while**, *prima* è verificata la nazionalità e *poi* è controllato il rango. Così, se una conquista è al tempo stesso sia duchessa che spagnola, viene prima inserita nella sequenza L e poi cancellata dal catalogo. L'operazione remove() restituisce la posizione del successore di q, come previsto dalla specifica; questa posizione viene assegnata a q, proseguendo la ricerca. ■

Per valutare la complessità della procedura donGiovanni(), occorre prima conoscere la complessità di ogni operazione impiegata. In altri termini, occorre conoscere la realizzazione della struttura astratta *sequenza*. Questo capitolo introduce le principali strutture di dati per la realizzazione di sequenze. Nella prima parte discuteremo di strutture basate su puntatori, che permettono di realizzare sequenze dinamiche. Nella seconda discuteremo di strutture basate su vettori, che se da un lato limitano il dinamismo, dall'altro si dimostrano estremamente semplici ed efficienti dal

```
donGiovanni(SEQUENCE catalogo)
    SEQUENCE L = Sequence()
    POS q = catalogo.head()
    while not catalogo.finished(q) do
        item v = catalogo.read(q)
        if v.nazione = spagnola then
            L.insert(L.next(L.tail()),v)
        if v.rango = duchessa then
            q = catalogo.remove(q)
        else
            q = catalogo.next(q)
```

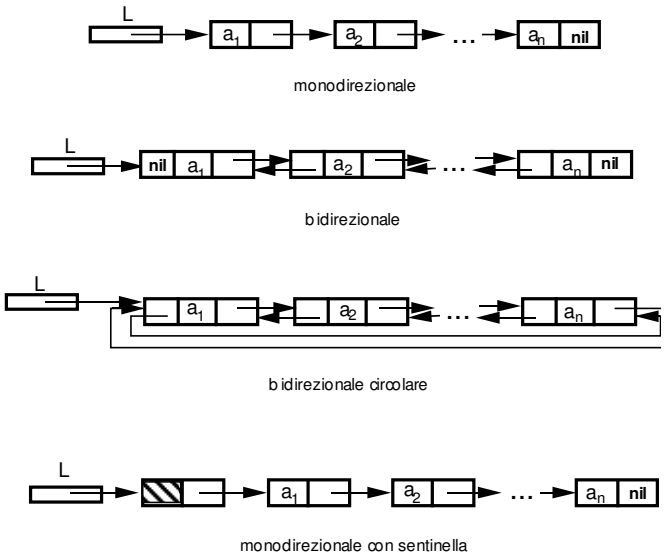

Figura 1.1: Alcune realizzazioni di una lista con puntatori. **TODO**: Problema figura tagliata a destra

punto di vista dell'occupazione di memoria.

1.1 Liste con puntatori

La prima realizzazione che analizziamo prende il nome di *lista* (*list*, in inglese) ed è basata sull'uso di puntatori, dove alla contiguità tra elementi della sequenza non corrisponde necessariamente una contiguità di memorizzazione. Questa realizzazione permette sia di ottenere complessità $O(1)$ per tutte le operazioni, sia di non porre limiti alla dimensione massima della sequenza.

Come mostrato nella Figura 1.1, ci sono vari modi possibili di realizzare le liste mediante puntatori. L'idea di base è di memorizzare una lista di n elementi in n record, usualmente detti celle, tali che l'i-esima cella contenga il valore dell'i-esimo elemento della lista e l'indirizzo (puntatore) della cella contenente l'elemento successivo (realizzazione monodirezionale) o gli indirizzi sia della cella successiva che precedente (realizzazione bidirezionale).

La prima cella è indirizzata da una variabile L di tipo puntatore, mentre l'ultima cella contiene il valore convenzionale **nil** come indirizzo della cella successiva (ovviamente, nel caso bidirezionale,

la prima cella contiene **nil** come indirizzo della cella precedente). Ricordiamo che gli indirizzi delle celle sono noti soltanto alla macchina, non al programmatore.

Entrambe le realizzazioni possono essere rese circolari, facendo sì che l'ultima cella contenga l'indirizzo della prima cella e, nel caso bidirezionale, che la prima cella contenga l'indirizzo dell'ultima. Per rendere più leggibile lo pseudocodice delle operazioni insert() e remove() quando operano sugli elementi estremi della lista, può essere conveniente introdurre una cella in più, detta *sentinella*, che è direttamente indirizzata da L, ma non contiene il valore di alcun elemento. Si hanno così otto realizzazioni possibili, a seconda che ci siano o no bidirezionalità, circolarità, e sentinella. La Figura 1.1 illustra quattro di queste otto possibilità. Tutte le otto realizzazioni richiedono $\Theta(n)$ spazio di memoria.

In questa sezione, descriveremo la realizzazione bidirezionale circolare con sentinella che permette di ottenere complessità $O(1)$ per ciascuna operazione semplificandone nel contempo lo pseudocodice, a scapito di un lieve spreco di memoria. Va però notato che è possibile ottenere complessità $O(1)$ per quasi tutte le operazioni (con l'esclusione dei soli tail() e prev() che hanno complessità $O(n)$) anche con una semplice realizzazione monodirezionale, senza circolarità né sentinelle, usando quindi minore quantità di memoria ma complicando lo pseudocodice delle operazioni.

Con la realizzazione bidirezionale circolare con sentinella, la posizione pos_i è uguale al puntatore della cella contenente a_i, per $1 \leq i \leq n$, mentre pos_0 e pos_{n+1} sono uguali all'indirizzo della sentinella (si veda la Figura 1.2). In questo caso, i tipi POS e LIST coincidono. Si noti che una lista di n elementi è formata da $n+1$ celle (che contengono gli n elementi più la sentinella) e che quindi una lista vuota contiene esattamente una cella (la sentinella) che ha il proprio indirizzo sia come indirizzo della cella precedente che della cella successiva.

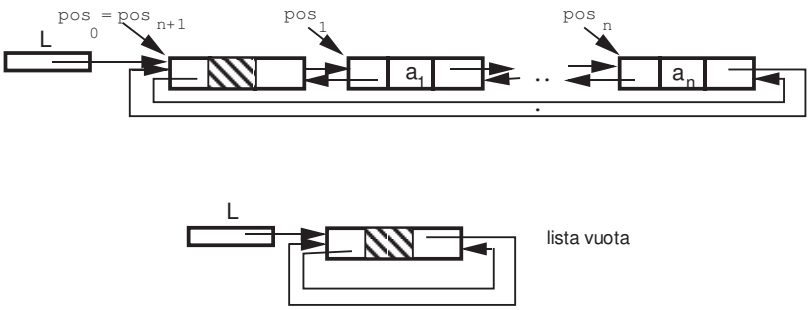

Figura 1.2: Realizzazione bidirezionale circolare con sentinella.

```
LIST
    LIST pred                          % Predecessore
    LIST succ                          % Successore
    item value                         % Elemento

    LIST List()
        LIST t = new LIST
        t.pred = t
        t.succ = t
        return t

    boolean isEmpty()
        return pred = succ = this

    POS head()
        return succ

    POS tail()
        return pred

    POS next(POS p)
        return p.succ

    POS prev(POS p)
        return p.pred

    boolean finished(POS p)
        return (p = this)

    item read(POS p)
        return p.value

    write(POS p, item v)
        p.value = v

    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 t;

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

Si noti che nelle procedure insert() e remove() la posizione $p = pos_i$ non viene modificata all'interno della procedura stessa, mentre viene restituita al chiamante la posizione corrispondente al nuovo elemento i-esimo. In accordo con la specifica, nel caso di insert() il nuovo elemento inserito diventa l'i-esimo, mentre quello che era il vecchio i-esimo diventa l'$(i+1)$-esimo, ecc. Analogamente, nel caso di remove(), il vecchio $(i+1)$-esimo elemento diventa l'i-esimo, ecc. (si veda la Figura 1.3).

Utilizzando i puntatori è molto gravoso, se non addirittura impossibile, verificare che gli input di ogni operazione siano corretti. Per esempio, la specifica di read(p) dovrebbe prevedere che p sia una posizione di un elemento della lista; essendo p realizzato con un puntatore, la verifica di questa precondizione richiederebbe la scansione della lista! Per ragioni di efficienza, la condizione non viene verificata, ed il corretto uso delle operazioni è quindi piena responsabilità del programmatore.

1.2 Realizzazione con vettori

Una realizzazione alternativa consiste nel memorizzare gli elementi della sequenza in un vettore. In tal caso, la posizione di un elemento è un indice del vettore e pos_i è proprio uguale ad i, per $1 \leq i \leq n$. In altri termini, alla contiguità tra elementi della sequenza corrisponde una contiguità di memorizzazione. Questa realizzazione permette agevolmente, cioè in tempo $O(1)$, di passare da un elemento al successivo o al precedente, di accorgersi se si tenta di superare un estremo della sequenza, di leggere o modificare il valore di un elemento, anche tramite accesso diretto (non sequenziale).

Lo svantaggio principale è che l'inserzione di un nuovo elemento o la cancellazione di uno vecchio richiedono un inaccettabile tempo $O(n)$, poiché è necessario scalare di una posizione a destra, in caso di inserzione, o a sinistra, in caso di cancellazione, tutti gli elementi che seguono quello inserito o cancellato.

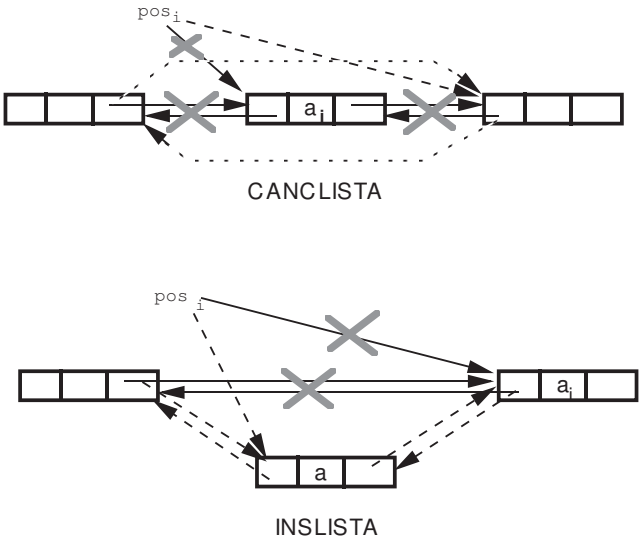

Figura 1.3: Aggiornamento dei puntatori nelle procedure remove() e insert(). **TODO**: “canclista” → “remove()”, "inslista” → “insert()”

Se gli inserimenti avvengono solo in fondo alla sequenza (con costo $O(1)$), questo approccio è ancora accettabile; tuttavia, si presenta un ulteriore problema: il vettore ha dimensione fissa, e il numero di elementi inseriti può crescere oltre la dimensione del vettore. Quando il vettore si esaurisce, una possibilità è quella di allocare un nuovo vettore di dimensione doppia, copiare tutti gli elementi dal vecchio al nuovo vettore, e infine liberare la memoria utilizzata dal vecchio vettore.

Chiaramente il costo di tale operazione è $O(n)$ nel caso pessimo, ma è possibile dimostrare che il costo “ammortizzato” degli inserimenti resta $O(1)$. Per semplificare i calcoli, assumiamo che il vettore abbia inizialmente dimensione 1 e consideriamo il costo di inserire n elementi. Sia c_i il costo dell'i-esimo inserimento:

$$c_i = \begin{cases} i & \text{se } \exists k \in \mathbb{Z} : i = 2^k + 1 \\ 1 & \text{altrimenti} \end{cases}$$

In altre parole, il costo è normalmente 1; ma un inserimento eseguito immediatemente dopo aver riempito completamente il vettore (ovvero quando il numero di elementi è pari ad una potenza di 2) costa i, perché dobbiamo copiare il vettore vecchio in quello nuovo. Il costo totale $C(n)$ per n inserimenti è quindi:

$$C(n) = \sum_{i=1}^{n} c_i \leq n + \sum_{k=0}^{\lfloor \log_2 n \rfloor} 2^k$$

La prima componente n è data dagli n inserimenti “puri”, mentre la seconda è data dal costo delle copiature. Semplificando l'espressione otteniamo:

$$C(n) \leq n + 2^{\lfloor \log_2 n \rfloor + 1} - 1 \leq n + 2n$$

Quindi il costo totale per n inserimenti è limitato superiormente da $3n$, che significa che il costo ammortizzato per ogni inserimento è $O(1)$.

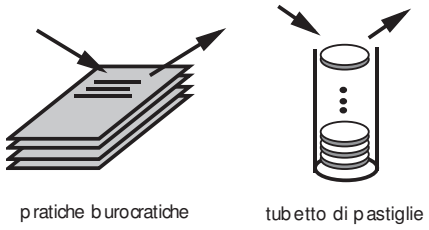

Figura 1.4: Meccanismo di accesso LIFO.

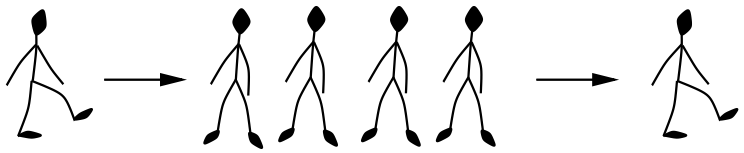

Figura 1.5: Meccanismo di accesso FIFO.

1.3 Pile e code

Se è noto a priori in che modo gli elementi di una sequenza possono essere inseriti o rimossi, è possibile pensare a realizzazioni efficienti basate su vettore.

Una *pila* (in inglese, *stack*) è una sequenza di elementi di un certo tipo in cui è possibile aggiungere o togliere elementi soltanto ad un estremo (la "testa") della sequenza. Può essere quindi vista come un caso speciale di sequenza in cui l'ultimo elemento inserito è anche il primo ad essere rimosso e non è possibile accedere ad alcun elemento che non sia quello in testa. Tale meccanismo di accesso è detto LIFO (dall'inglese "Last In First Out", ovvero "ultimo arrivato, primo servito").

Esempio (Pila) Una catasta di pratiche burocratiche che si accumulano sulla scrivania di un funzionario ministeriale è spesso (ingiustamente) trattata col meccanismo LIFO, in cui l'ultima pratica inserita in testa alla catasta sarà esaurita per prima. Un tubetto di pastiglie ha un meccanismo di accesso analogo, poiché l'ultima pastiglia inserita nel tubetto dal fabbricante è anche la prima ad essere estratta dall'acquirente (si veda la Figura 1.4). ■

Analogamente, una *coda* (in inglese, *queue*) è una sequenza di elementi di un certo tipo, in cui è possibile aggiungere elementi ad un estremo (il "fondo") e togliere elementi dall'altro estremo (la "testa"). Può essere considerata come una particolare sequenza in cui il primo elemento inserito è anche il primo ad essere rimosso e non è possibile accedere ad alcun elemento che non sia quello in testa o quello in fondo. Tale meccanismo di accesso è detto FIFO (dall'inglese "First In First Out", ovvero "primo arrivato, primo servito").

Esempio (Coda, o Fila d'attesa) In Inghilterra, una fila di persone ad uno sportello postale è gestita con la disciplina FIFO (si veda la Figura 1.5). ■

1.3.1 Specifica

Sebbene le operazioni delle pile e delle code possano essere simulate con le operazioni di una lista, è prassi comune utilizzare nomi leggermente diversi.

Le operazioni per il tipo di dato pila comprendono, oltre alla creazione di una pila vuota, l'interrogazione isEmpty(), per verificare che la pila sia vuota oppure no, l'operazione di lettura top(), per leggere il valore dell'elemento in testa alla pila, e quelle di modifica pop(), per eliminare l'elemento che si trova in testa alla pila, e push(), per aggiungere un elemento in testa alla pila. La specifica sintattica del tipo di dato è la seguente:

STACK

% Restituisce **true** se la pila è vuota
boolean isEmpty()

% Inserisce v in cima alla pila
push(**item** v)

% Estrae l'elemento in cima alla pila e lo restituisce al chiamante
item pop()

% Legge l'elemento in cima alla pila
item top()

Le operazioni per il tipo di dato coda comprendono isEmpty() e top() con la stessa specifica delle rispettive operazioni della pila, dequeue() che rimuove un elemento dalla testa della coda, e enqueue() che inserisce un elemento in fondo alla coda.

QUEUE

% Restituisce **true** se la coda è vuota
boolean isEmpty()

% Inserisce v in fondo alla coda
enqueue(**item** v)

% Estrae l'elemento in testa alla coda e lo restituisce al chiamante
item dequeue()

% Legge l'elemento in testa alla coda
item top()

Poiché pile e code sono casi particolari di sequenze, ogni realizzazione descritta per le liste funziona anche per le pile e per le code. Per esempio, le operazioni per le pile possono essere simulate con quelle delle liste nel modo seguente:

$$p.\mathsf{top}() = p.\mathsf{read}(p.\mathsf{head}())$$
$$p.\mathsf{pop}() = p.\mathsf{remove}(p.\mathsf{head}())$$
$$p.\mathsf{push}(v) = p.\mathsf{insert}(p.\mathsf{head}(), v).$$

Usando le realizzazioni delle liste con puntatori viste nel paragrafo precedente, la complessità di ciascuna operazione per le pile è $O(1)$.

È possibile realizzare in modo simile le operazioni della coda; ma è importante notare che la complessità di enqueue() è $O(1)$ se viene usata una realizzazione con liste bidirezionali, mentre diviene $\Theta(n)$ usando le altre realizzazioni.

Oltre alle realizzazioni per pile e code che si basano sulle liste, ci sono due realizzazioni specifiche, che utilizzano vettori, in cui tutte le operazioni delle pile e delle code continuano ad avere complessità $O(1)$. Tali realizzazioni sono utilizzabili quando si è in grado di limitare superiormente con un opportuno valore il numero massimo di elementi che possono essere presenti nella pila o nella coda.

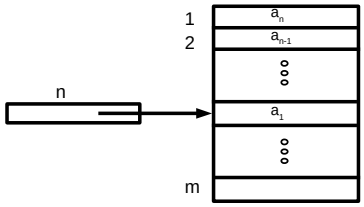

Figura 1.6: Realizzazione di una pila con un vettore.

1.3.2 Realizzazione di una pila con vettore

Come mostrato nella Figura 1.6, una realizzazione con vettore del tipo di dato pila consiste nel memorizzare gli n elementi della pila, in ordine inverso, nelle prime n posizioni di un vettore di dimensione *capacity*, con $capacity \geq n$, mantenendo una variabile *size* che identifica il numero di celle occupata. La pila vuota viene individuata dal valore 0 contenuto in *size*, mentre la pila satura, cioè che contiene il massimo numero consentito di elementi, è individuata dal valore $size = capacity$. pop() è realizzata decrementando *size*, mentre push() è realizzata inserendo il nuovo elemento nella posizione del vettore individuata da *size* e incrementando *size*. Eventuali tentativi di inserzione in una pila satura o di estrazione da una pila vuota possono sorgere in caso di errori di programmazione e non sono ammessi. Con questa realizzazione, ogni operazione della pila richiede tempo costante.

STACK

```
item[] A                          % Elementi
int size                          % Cursore
int capacity                      % Dim. massima

STACK Stack(int dim)
    STACK t = new STACK
    t.A = new int[0...dim-1]
    t.capacity = dim
    t.size = 0
    return t

item top()
    precondition: size > 0
    return A[size-1]

boolean isEmpty()
    return size = 0

item pop()
    precondition: size > 0
    size = size - 1
    return A[size]

push(item v)
    precondition: size < capacity
    A[size] = v
    size = size + 1
```

1.3.3 Realizzazione di una coda con vettore circolare

Si supponga di avere un vettore di m elementi, indicati da 0 a $m-1$, in cui l'elemento di indice 0 è considerato come successore di quello di indice $m-1$. La coda è contenuta in n posizioni consecutive del vettore, per esempio in verso orario, e i suoi elementi estremi sono individuati da due cursori. Per cancellare o per inserire un elemento, viene incrementato di uno (modulo m) il cursore di testa o di fondo, rispettivamente. In questo modo la coda "migra" lungo il vettore circolare e, ad ogni istante, si trova in posizioni consecutive del vettore secondo il verso orario.

Anziché utilizzare due cursori, per individuare la coda è sufficiente conoscere l'indice del primo elemento e la lunghezza della coda stessa, come mostrato nella Figura 1.7. Infatti, se i è la posizione dell'elemento in testa alla coda e ci sono *size* elementi in essa, allora quella dell'ultimo elemento della coda è $(i + size - 1)$ modulo m. La complessità di ogni operazione è $O(1)$.

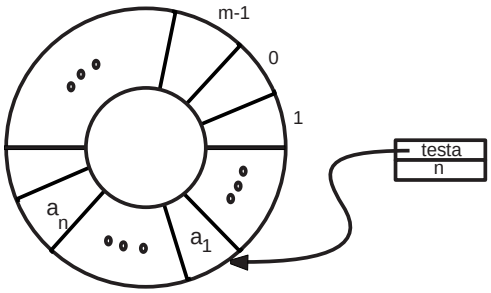

Figura 1.7: Realizzazione di una coda con un vettore circolare.

QUEUE

```
item[] A                                  % Elementi
int size                                  % Dim. attuale
int head                                  % Testa
int capacity                              % Dim. massima

QUEUE Queue(int dim)
    QUEUE t = new QUEUE
    t.A = new int[0...dim-1]
    t.capacity = dim
    t.head = 0
    t.size = 0
    return t

item top()
    precondition: size > 0
    return A[head]

boolean isEmpty()
    return size = 0

item dequeue()
    precondition: size > 0
    item t = A[head]
    head = (head+1) mod capacity
    size = size - 1
    return t

enqueue(item v)
    precondition: size < capacity
    A[(head+size) mod capacity] = v
    size = size + 1
```

1.4 Pile e procedure ricorsive

Una importante applicazione delle pile consiste nella gestione dell'esecuzione di procedure ricorsive. Infatti, l'esecuzione di una procedura ricorsiva prevede il "salvataggio" dei dati sui quali lavora la procedura al momento di una nuova chiamata ricorsiva interna. Tali dati sono successivamente "riesumati" quando la computazione interna termina e viene ripresa la computazione che era stata sospesa. Questo meccanismo è ovviamente di tipo LIFO, poiché la computazione "più interna" di una procedura ricorsiva, cioè quella iniziata per ultima, è sempre la prima a terminare. Per chiarire come procede il calcolo in una procedura ricorsiva, si consideri un problema "leggendario", divenuto ormai un classico della programmazione ricorsiva:

TORRI DI HANOI. *In un monastero tibetano ci sono n dischi d'oro, forati al centro, tutti di diametro diverso. I dischi sono infilati in un piolo verticale, accatastati per diametro decrescente a partire dal basso. I monaci devono spostare tutti i dischi dal piolo in cui si trovano in un altro piolo, formando una catasta identica a quella di partenza. È possibile usare un terzo piolo di appoggio per effettuare i trasferimenti. Si può sfilare da un piolo soltanto un disco per volta, e il disco sfilato deve essere subito infilato in un altro piolo evitando di sovrapporlo ad un disco di diametro più piccolo.* [1]

[1] Narra la leggenda che il mondo finirà prima che i monaci abbiano spostato tutti i dischi!

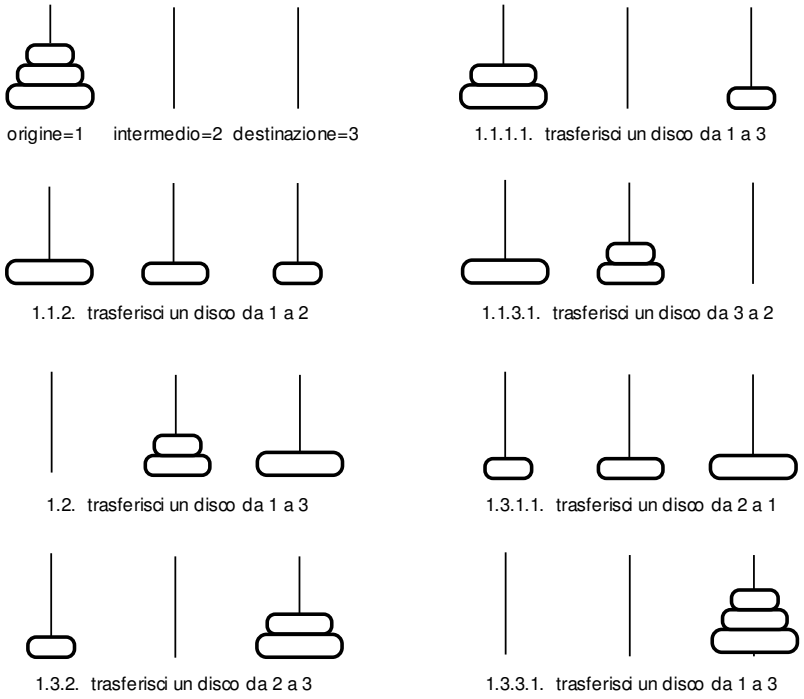

Figura 1.8: Sequenza di trasferimenti di dischi eseguiti dalla procedura ricorsiva hanoi-recursive() per $n = 3$.

Questo rompicapo si presta ad una semplice risoluzione ricorsiva. Indichiamo i tre pioli coi nomi *source*, *dest* e *middle*. Se c'è un solo disco, la soluzione è banale: si trasferisce direttamente il disco dal piolo *source* al piolo *dest*. Se ci sono n dischi, con $n > 1$, basta spostare i primi $n-1$ dischi della catasta dal piolo *source* al piolo *middle*, indi trasferire direttamente il disco rimasto dal piolo *source* al piolo *dest*, e infine spostare gli $n-1$ dischi dal piolo *middle* al piolo *dest*. Supponendo che *source*, *middle* e *dest* siano tre interi distinti (p.e. 1, 2 e 3) si ottiene la seguente procedura:

hanoi-recursive(**int** n, **int** *source*, **int** *dest*, **int** *middle*)

if $n = 1$ **then**
  % Trasferisci un disco da *source* a *dest*
else
(1) hanoi-recursive$(n-1, source, middle, dest)$
  % Trasferisci un disco da *source* a *dest*
(2) hanoi-recursive$(n-1, middle, dest, source)$

Benché sia facile dimostrare la correttezza di questa procedura usando il principio d'induzione, è molto difficile capire quale sia l'effettiva sequenza di trasferimenti eseguiti dalla procedura stessa.

Esempio (Torri di Hanoi) Si assuma $n = 3$, *source* $= 1$, *middle* $= 2$ e *dest* $= 3$. L'esecuzione della procedura hanoi-recursive() con questi dati provoca la seguente sequenza di chiamate ricorsive e di trasferimenti di dischi (si veda la Figura 1.8):

1. hanoi-recursive(3,1,3,2)
   1.1. hanoi-recursive(2,1,2,3)

1.1.1. hanoi-recursive(1,1,3,2)
1.1.1.1. trasferisci un disco da 1 a 3
1.1.2. trasferisci un disco da 1 a 2
1.1.3. hanoi-recursive(1,3,2,1)
1.1.3.1. trasferisci un disco da 3 a 2
1.2. trasferisci un disco da 1 a 3
1.3. hanoi-recursive(2,2,3,1)
1.3.1. hanoi-recursive(1,2,1,3)
1.3.1.1. trasferisci un disco da 2 a 1
1.3.2. trasferisci un disco da 2 a 3
1.3.3. hanoi-recursive(1,1,3,2)
1.3.3.1. trasferisci un disco da 1 a 3

La complessità della procedura hanoi-recursive() è esponenziale. Infatti, il numero $T(n)$ di trasferimenti di dischi è dato da:

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \\ 2T(n-1)+1 & \text{se } n \geq 2 \end{cases}$$

Sostituendo successivamente si ottiene:

$$\begin{aligned} T(n) &= 2T(n-1)+1 = \\ &= 2(2T(n-2)+1)+1 = 4T(n-2)+3 = \\ &\vdots \\ &= 2^{n-1}T(1)+2^{n-1}-1 = 2^n-1. \end{aligned}$$

Assumendo un trasferimento ogni microsecondo, per $n = 100$ occorrono ben 400.000 miliardi di secoli: nessun dubbio che il mondo finirà prima!

Vediamo come sia possibile eliminare manualmente la ricorsione dalla procedura hanoi-recursive(), trasformandola in una procedura iterativa equivalente chiamata hanoi-iterative(), utilizzando una pila. La procedura è organizzata attorno ad un ciclo **while** principale la cui esecuzione "simula" una singola chiamata di hanoi-recursive(). All'inizio di ogni esecuzione del ciclo principale, le variabili *n*, *source*, *dest* e *middle* rappresentano i parametri formali della chiamata simulata. Ogni chiamata ricorsiva "aperta" ma non ancora terminata viene simulata da una tupla contenente i parametri formali inserita nella pila. Per snellire il codice, la tupla è letta e scritta tramite parentesi angolari $\langle\rangle$.

L'invocazione ricorsiva (1) viene simulata dal ciclo **while** interno, che inserisce nella pila le successive chiamate per $n-1$, $n-2$, ..., 3, 2. In ognuna di queste chiamate, vengono scambiate le variabili *dest* e *middle*. Quando $n = 1$ (caso base), il disco nel piolo *source* viene spostato nel piolo *dest*. A questo punto, si estraggono dalla pila i parametri formali dell'ultima chiamata lasciata aperta, si sposta un disco da *source* a *dest*, e si simula l'invocazione ricorsiva (2) decrementando *n* di 1 e scambiando i pioli *source* e *middle* (i pioli *middle* e *dest* erano già stati scambiati) e ricominciando il ciclo, che continua fino a quando la pila non è vuota.

```
hanoi-iterative(int n, int source, int dest, int middle)
STACK S = Stack()
boolean finedelmondo = false
while not finedelmondo do
    while n ≠ 1 do
        S.push(⟨n, source, dest, middle⟩)
        n = n − 1
        dest ↔ middle
    % Trasferisci un disco da source a dest
    if S.isEmpty() then
        finedelmondo = true
    else
        ⟨n, source, dest, middle⟩ = S.pop()
        % Trasferisci un disco da source a dest
        n = n − 1
        source ↔ middle
```

Anche se questo è un caso semplice, con solo due chiamate, l'approccio seguito per la trasformazione non è intuitivo. Nei casi più complessi, seguire il filo logico della ricorsione è ancora più difficile. Esiste tuttavia un approccio sistematico per la trasformazione di *ogni* procedura ricorsiva in una iterativa equivalente, che consiste nel salvare, oltre ai parametri attuali delle chiamate (e le eventuali variabili locali), anche il "punto di ritorno", ovvero il punto del codice da cui riprendere l'esecuzione. Le regole da seguire sono le seguenti:

(1) Creare una pila vuota e iniziare un ciclo **while** principale con i valori dei parametri attuali ricevuti dalla chiamata.

(2) Creare una variabile *pos* che rappresenta il particolare spezzone di codice da eseguire nel ciclo **while**. Per questa variabile, utilizziamo un'etichetta START per indicare l'inizio di una chiamata; un'etichetta per indicare tutti gli spezzoni di codice che seguono il rientro da una chiamata (nel nostro caso, AFTER1 dopo la chiamata (1) e AFTER2 dopo la chiamata (2)); END per indicare il codice da eseguire al termine della chiamata (che qui coincide con AFTER2); e STOP per indicare che l'esecuzione è terminata.

(3) Sostituire ogni chiamata ricorsiva con un'istruzione che salva nella pila i valori dei parametri, delle variabili locali e l'etichetta da cui ripartire dopo il completamento della chiamata ricorsiva, seguita da istruzioni che modificano i parametri assegnando loro i valori che devono assumere nella chiamata ricorsiva.

(4) Scrivere il codice per END, che controlla se la pila è vuota e, in caso affermativo, termina il ciclo mettendo *pos* = STOP; altrimenti, estrae dalla pila i valori salvati dei parametri, delle eventuali variabili locali e della prossima etichetta da eseguire.

Tali regole sono valide se tutti i parametri sono passati per valore (si consideri per esercizio cosa accade quando ci sono parametri passati per reference). Applicando queste regole ad hanoi-recursive(), si ottiene la seconda versione di hanoi-iterative() presentata di seguito.

1.5 Reality Check

Nelle librerie Java, sono presenti due realizzazioni del tipo di dati astratto "sequenza" basate su vettori: `Vector` e `ArrayList`, entrambe nel package `java.util`. `Vector` è presente sin dalle prime versioni di Java (JDK 1.0), mentre `ArrayList` è stato aggiunto con la libreria Java Collections (JDK 1.2). `Vector` d'altronde è stato riscritto per realizzare l'interfaccia `List` delle

Collections, quindi `Vector` e `ArrayList` sembrano del tutto simili. Perché dunque ve ne sono due?

Leggendo fra le righe della documentazione di `ArrayList`, si scoprono alcuni dettagli interessanti: *The add operation runs in amortized constant time, that is, adding n elements requires $O(n)$ time. [...] The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost*. Questi commenti non sono presenti nella documentazione di `Vector`; invece, si fa specifico riferimento al fatto che `Vector` non cresce raddoppiando, ma allocando un nuovo vettore più lungo di un numero costante d di elementi.

Proviamo a calcolare il costo ammortizzato di n inserimenti quando l'incremento è costante e non si raddoppia. Si assuma che la dimensione iniziale del vettore sia d e denotiamo con c_i il costo dell'i-esimo inserimento:

$$c_i = \begin{cases} i & \text{se } i \bmod d = 1 \\ 1 & \text{altrimenti} \end{cases}$$

```
hanoi-iterative(int n, int source, int dest, int middle)
STACK S = Stack()
int pos = START
while pos ≠ STOP do
    if pos = START then
        if n = 1 then
            % Trasferisci un disco da source a dest
            pos = END
        else
            S.push(⟨n, source, dest, middle, AFTER1⟩)
            n = n − 1
            dest ↔ middle
    else if pos = AFTER1 then
        % Trasferisci un disco da source a dest
        S.push(⟨n, source, dest, middle, END⟩)
        n = n − 1
        source ↔ middle
        pos = START
    else if pos = END then
        if S.isEmpty() then
            pos = STOP
        else
            ⟨n, source, dest, middle, pos⟩ = S.pop()
```

Il costo totale $C(n)$ di n inserimenti è quindi $O(n^2)$, come illustrato qui sotto, e il costo

ammortizzato per singolo inserimento è $O(n)$:

$$\begin{aligned} C(n) &= \sum_{i=1}^{n} c_i \\ &\leq n + \sum_{j=1}^{\lfloor n/d \rfloor} d \cdot j \\ &= n + d\frac{(\lfloor n/d \rfloor + 1)\lfloor n/d \rfloor}{2} \\ &\leq n + \frac{(n/d+1)n}{2} = O(n^2) \end{aligned}$$

Che lezione trarre da questa lettura della documentazione? Una scelta a prima vista banale (meglio incrementare o raddoppiare?) si rivela di estrema importanza; e la scelta corretta potrebbe essere controintuitiva (meglio raddoppiare!). Chi ha scritto `Vector` non aveva seguito un buon corso di Algoritmi e Strutture Dati!

1.6 Esercizi

Esercizio 1.1 (I primi n interi)**.** Si scriva una procedura ricorsiva di complessità ottima che, dato un intero $n \geq 1$, costruisca due liste L ed M tali che $L = 1, 2, \ldots, n-1, n$ ed $M = n, n-1, \ldots, 2, 1$.

Esercizio 1.2 (Epurazione)**.** Si scriva una procedura che, data una lista L di interi, restituisca un'altra lista M che contenga solo gli elementi di L i cui valori non compaiono esattamente due volte (p.e., se $L = 5,7,3,2,2,1,2,3$, allora $M = 5,7,2,2,1,2$).

Esercizio 1.3. Si supponga di rappresentare un numero decimale d mediante una lista L tale che il valore dell'elemento i-esimo della lista sia uguale a quello dell'i-esima cifra del numero. Si scriva una funzione che effettua la somma di due numeri. (Esempio: se $d = 190$ e $d' = 17$ allora $L = 1,9,0$, $L' = 1,7$, e il risultato $d'' = d + d'$ è restituito nella lista $L'' = 2,0,7$).

Esercizio 1.4 (Tangentopoli)**.** Dopo essere stati inquisiti per tangentopoli, n politici hanno deciso di suicidarsi disponendosi in cerchio e uccidendo via via una persona ogni k (... ma l'ultimo rimasto si suiciderà poi per davvero?). Si scriva una procedura che, data la lista L degli aspiranti suicidi, e dato k, $1 \leq k \leq n$, stampi l'ordine delle vittime (p.e. se L = Caio, Sempronio, Tizio, Tullio e $k = 3$, allora l'ordine delle vittime è: Tizio, Sempronio, Tullio, Caio). Si assuma di poter cancellare gli elementi da L.

Esercizio 1.5. Si scriva una procedura ricorsiva che data una lista L di interi, la modifichi eliminando ogni elemento pari e replicando ogni elemento dispari tante volte quanti sono gli elementi pari che lo precedono (p.e., $L = 4,6,7,3,2,5$, allora si ha $L = 7,7,7,3,3,3,5,5,5,5$).

Esercizio 1.6. Si scrivano le procedure per le operazioni delle liste usando la realizzazione con puntatori monodirezionale non circolare e senza sentinella. Tutte le operazioni (tranne tail() e prev()) devono avere complessità $O(1)$. (Suggerimento: si definisca pos_i uguale all'indirizzo della cella contenente a_{i-1}, se $i > 0$, e $pos_i =$ **nil**, se $i = 1$).

Esercizio 1.7. È possibile memorizzare due pile in un unico vettore comune, qualora se ne faccia "crescere" una a partire dal primo elemento del vettore e l'altra a partire dall'ultimo. Si assuma di avere in memoria un solo vettore disponibile per memorizzare le pile. In tal caso, non potranno essere create più di due pile. Si riscrivano le procedure relative alle usuali operazioni sulle pile, includendo le opportune condizioni di errore qualora si cerchi di creare più di due pile.

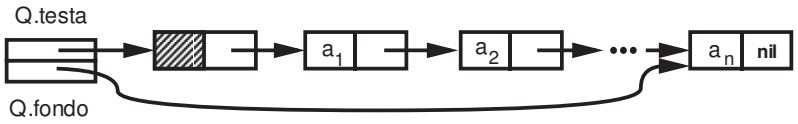

Figura 1.9: Realizzazione monodirezionale con sentinella di una coda. **TODO**: Rimuovere "Q."

Esercizio 1.8. Utilizzando le tecniche di trasformazione illustrate in questo capitolo, si fornisca una versione iterativa della procedura ricorsiva firstN(), che computa la lista dei primi n interi, mostrata nell'Esercizio 1.1

Esercizio 1.9. Il tipo di dato DEQUEUE (double ended queue) è una sequenza modificabile ad entrambi gli estremi, in cui è possibile inserire un elemento in testa o inserirlo in fondo, e cancellare un elemento dalla testa o cancellarlo dal fondo. Si fornisca una specifica ed una realizzazione mediante puntatori.

Esercizio 1.10. Si realizzino le operazioni della coda, modificando la realizzazione monodirezionale non circolare di una lista, ottenendo complessità $O(1)$ anche per enqueue(). Si usino n celle in modo che la prima cella sia indirizzata da un puntatore *testa* e l'ultima da un puntatore *fondo*, come mostrato nella Figura 1.9.

Esercizio 1.11 (Pigeonhole Sort)**.** Il *Pigeonhole Sort* ("ordinamento della piccionaia") è un'estensione del Counting Sort che permette di ordinare in tempo lineare coppie (chiave, valore) invece che singoli interi. Le chiavi devono essere valori interi compresi fra 1 e k. Il funzionamento è simile al Counting Sort; tuttavia, durante la scansione iniziale, se si incontra una coppia chiave-valore (i,v) si aggiunge v in fondo ad una lista $B[i]$ (invece di incrementare il contatore $B[i]$). Durante la scansione del vettore B, si trascrivono nuovamente tutte le coppie chiavi-valore, in ordine di chiave e mantenendo lo stesso ordinamento per i valori con la stessa chiave. Realizzate il Pigeonhole Sort nel vostro linguaggio preferito.

1.7 Soluzioni

Soluzione Esercizio 1.1

Una limitazione inferiore alla complessità del problema è $\Omega(n)$ perché un qualsiasi algoritmo deve almeno generare gli n elementi della lista L e della lista M. La procedura firstN() effettua un inserimento in testa alla lista L ed uno in fondo alla lista M ad ogni chiamata ricorsiva, in modo da memorizzare gli interi in senso crescente in L e decrescente in M.

```
firstN(int n, LIST L, LIST M)
if n ≥ 1 then
    L.insert(L.head(),n)
    M.insert(M.next(M.tail()),n)
    firstN(n−1,L,M)
```

Soluzione Esercizio 1.2

Si effettua una doppia scansione della lista L per selezionare un elemento da L, verificare che non compaia esattamente 2 volte, e inserirlo in M. La posizione p è quella dell'elemento di L da verificare, la posizione q è utilizzata per scandire L alla ricerca delle occorrenze dell'elemento di posizione p, mentre *count* serve a contare quante volte tale elemento si ripete. La posizione r è quella che segue l'ultimo elemento di M. La complessità è $O(n^2)$.

```
epurazione(LIST L, LIST M)
LIST M = List()
POS p = L.head()
while not L.finished(p) do
    count = 0
    POS q = L.head()
    while not L.finished(q) do
        if L.read(q) = L.read(p) then
            count = count + 1
        q = L.next(q)
    if count ≠ 2 then
        M.insert(M.next(M.tail()), L.read(p))
    p = L.next(p)
```

Soluzione Esercizio 1.4

Il problema può essere risolto semplicemente scorrendo la lista, ripartendo dalla testa della lista tutte le volte che si arriva in fondo.

```
tangentopoli(LIST L)
POS p = L.head()
while not L.isEmpty() do
    for i = 1 to k − 1 do
        p = L.next(p)
        if L.finished(p) then
            p = L.head()
    print L.read(p)
    p = L.remove(p)
```