Capitolo 13

Liste, dizionari e percolazione

> Il 99 nero era stato un *nozoki* diretto al centro del triangolo dei bianchi; i bianchi, con il 100, si connettevano.
>
> Yasunari Kawabata, *Il Maestro di Go* (1942).

In questo capitolo discutiamo diversi tipi di strutture di dati e alcune applicazioni. Introduciamo un nuovo costrutto del linguaggio C, le `union`, e ci occupiamo delle *liste concatenate*: si tratta di strutture di dati elementari ma fondamentali in molti contesti. Discutiamo come costruire un semplice dizionario usando le liste, accennando al problema dell'analisi lessicale di un testo. Introduciamo quindi le liste concatenate in modo doppio, che si possono percorrere in modo semplice nei due versi.

Definiamo il concetto di *ricorsività*, con l'esempio semplice del calcolo del *fattoriale*, e di *albero* come approccio efficiente alla gestione dei dati. Un'applicazione utile in vari contesti scientifici e ingegneristici è la *ricostruzione delle componenti connesse di un cluster*: introduciamo un algoritmo elementare, poi usiamo le liste per migliorare le prestazioni del codice. Usando questi algoritmi possiamo studiare il problema della *percolazione*.

13.1 Le union

Una `union` è un tipo di variabile del linguaggio C che può contenere, in momenti diversi dell'esecuzione del codice, elementi di tipo diverso. Questa opportunità è sostanzialmente incompatibile con il paradigma che adottiamo di solito: una variabile è di un tipo dato, ben definito, e ogni ambiente sa quale tipo aspettarsi o restituire. Mantenere un controllo stretto sui tipi usati (anche in una semplice moltiplicazione) aiuta a fare meno errori di programmazione. Come discutiamo nel seguito ci sono però alcune situazioni in cui questa flessibilità è molto utile, e in questi casi il costrutto `union` è indispensabile.

La sintassi con cui si definisce una `union` è molto simile a quella usata per una `struct`, ma l'oggetto che si definisce è molto diverso. Definiamo una `union` di tipo `chameleon`

(un camaleonte, un animale cioè che cambia colore con facilità assecondando le esigenze del momento), e definiamo allo stesso tempo due istanze di questo tipo di `union`, che chiamiamo `theChameleonJohn` e `theChameleonMary`, riservando la memoria necessaria a conservarle:

```
union chameleon{
  double green;
  int red;
  char yellow;
} theChameleonJohn, theChameleonMary;
```

Sapendo che il tipo `double` richiede otto byte, il tipo `int` quattro byte e il tipo `char` un solo byte, ogni `union` di tipo `chameleon` riserva otto byte: una `union` riserva lo spazio necessario al piú grande dei suoi membri. Non riserviamo dunque, come nel caso di una `struct`, uno spazio di memoria per ognuna delle componenti, ma un solo spazio di memoria abbastanza grande da poter contenere uno qualsiasi dei membri della `union`. Le variabili `theChameleonJohn` e `theChameleonMary` possono contenere, in momenti diversi dell'esecuzione del codice, variabili di tipo `double`, `int` o `char`. Come per le `struct` i membri di una `union` si indicano con una notazione del tipo *nome_union.nome_elemento*, come in `theChameleonMary.red`.

Il tipo contenuto in un dato momento in una `union` è quello del dato che vi è stato messo per ultimo. Il programma non ha informazione su quale sia questo tipo, ed è compito del programmatore tenerne memoria (ovviamente un errore può essere catastrofico). Un modo consiste nel definire una variabile accessoria che sia aggiornata ogni volta che si aggiorni la `union`. Definiamo per esempio le costanti

```
#define GREEN 0
#define RED 1
#define YELLOW 2
```

e una nuova variabile per ognuna delle `union` che abbiamo definito

```
int colorOfTheChameleonJohn, colorOfTheChameleonMary;
```

Quando aggiorniamo la `union` ricordiamo sempre di aggiornare anche la variabile accessoria, come in

```
theChameleonJohn.green = 256.32;
colorOfTheChameleonJohn = GREEN;
```

oppure quando usiamo un intero,

```
theChameleonJohn.red = 25632;
colorOfTheChameleonJohn = RED;
```

Dal valore di `colorOfTheChameleonJohn` possiamo quindi sapere in ogni momento che tipo di valore abbiamo conservato nella `union`. Se dopo l'esecuzione dell'istruzione scritta sopra assumessimo che nella `union chameleon theChameleonJohn` c'è una variabile di

tipo `double` faremmo probabilmente un disastro. Le operazioni permesse sulle `union` e i meccanismi per accedere alle loro componenti sono gli stessi che per le `struct`. Inoltre si possono definire `struct` e array di `union`. Le `union` non sono usate frequentemente. Citiamo tre casi nei quali possono risultare utili.

1. In alcune situazioni è necessario allineare parole brevi (ad esempio variabili di tipo `char`) sul bordo di parole lunghe (per esempio di tipo `long int`): in questo caso un array di `union` può essere la soluzione piú efficiente.
2. Può essere utile inizializzare parti diverse di una parola lunga (per esempio fatta di otto byte) con variabili di dimensione piú piccola (per esempio fatte di un solo byte).
3. A volte può essere necessario trasmettere a un contesto diverso un oggetto il cui tipo può cambiare secondo le situazioni: una funzione può dover trasmettere una variabile di tipo `float` dopo aver ottenuto un certo risultato e una di tipo `int` per un risultato di tipo diverso.

Laboratorio 13.1 *Un fattoriale prudente e il suo logaritmo.*

Il risultato del calcolo del fattoriale $n!$ si può rappresentare in una variabile intera a 64 bit solo se $n \leqslant 20$. Scrivete una funzione che, se chiamata con un argomento $n \leqslant 20$, restituisca un valore di tipo `unsigned long long int`, che contiene il fattoriale calcolato esattamente, mentre per $n > 20$ restituisca un valore di tipo `double` che contiene il logaritmo del fattoriale ottenuto usando l'approssimazione di Stirling:

$$n! \simeq \sqrt{2\pi}\, n^{n+\frac{1}{2}}\, e^{-n}\ .$$

La funzione, che restituisce una `union`, dovrà essere chiamata tenendo conto di queste due possibilità.

13.1.1 Un esperimento virtuale

Descriviamo una tipica situazione in cui il costrutto `union` è particolarmente utile. Il contesto è quello del controllo di un esperimento, e dell'analisi *on-line* (cioè in diretta) dei suoi risultati. Sono frequenti le situazioni in cui un calcolatore è collegato, mediante un'adeguata interfaccia hardware, a un apparato sperimentale che raccoglie dati. Ovviamente qui dobbiamo limitarci a simulare l'esperimento, analizzando risultati generati in modo aleatorio dal nostro stesso codice. È importante però chiarire che la procedura che illustriamo è proprio la stessa che useremmo per controllare un vero esperimento.

Pensiamo, solo per fare un esempio, a un rivelatore che misura quel che succede a particelle che viaggiano in un mezzo fisico. Ipotizziamo che in ciascuna delle nostre misure possa realizzarsi una fra tre possibili situazioni. Nel primo caso la particella viaggia nel mezzo senza creare altre particelle, e la variabile interessante che l'esperimento misura e che ci interessa analizzare è la sua velocità. In questo caso l'apparecchio restituisce un

valore di tipo `double`. Il secondo caso è quello in cui la particella interagisce fortemente con il mezzo, e genera uno sciame di particelle; in questo caso l'apparato non è in grado di misurare le velocità delle particelle, ma può contarle, e restituire un valore di tipo `int`. Nel terzo caso nell'esperimento si verifica un errore: è un caso raro che a volte si realizza. Le ragioni per cui si verifica un errore possono essere diverse: le componenti elettroniche del rivelatore hanno reagito troppo lentamente, l'interazione fra le particelle è avvenuta troppo vicina al bordo del materiale (fuori, cioè, della "zona di confidenza"), c'è stato un errore nella memoria dell'apparato, eccetera. In questi casi vogliamo soltanto scrivere un messaggio d'errore su uno schermo, per avvisare il ricercatore che controlla l'esperimento, e l'esperimento produce direttamente questo messaggio d'errore, cioè una stringa di caratteri.

Siamo in una situazione in cui un'interfaccia hardware restituisce una variabile di tipo `double` o di tipo `int` o una serie di caratteri: una `union` è la struttura di dati ideale per gestire questi casi con la chiamata della stessa funzione.

Il codice veramente rilevante per questa operazione è molto semplice e compatto, mentre il codice complessivo che simula l'esperimento prendendo in modo aleatorio le decisioni sui risultati restituiti dall'interfaccia è un po' piú lungo ma comunque semplice: ne descriviamo le caratteristiche essenziali senza trattarlo nei dettagli.

Il tipo di `union` che serve è

```
union experimentData {
  double speed;
  int num;
  char *errorMessage;
};
```

La forma del codice che simula l'intero esperimento è del tipo di quello riportato nel Listato 13.1.

```
1 int main(void) {
2   int experimentOutput = 1;
3   int experimentNumber = 0;
4
5   srand(MY_SEED);
6   while (experimentOutput < 4) {
7     union experimentData scratchData;
8     experimentNumber++;
9     experimentOutput = setExperiment(experimentNumber);
10    scratchData = experimentBody(experimentOutput);
11    analyzeExperiment(experimentOutput, scratchData);
12  }
13  myEnd(experimentNumber);
14 }
```

Listato 13.1 La funzione `main` dell'esperimento virtuale.

Come dicevamo, la simulazione dell'esperimento è necessaria, ma non è il punto cruciale che vogliamo discutere: le parti rilevanti per il nostro ragionamento sono l'analisi dell'esperimento nella riga 11 e l'output finale dei risultati nella riga 13. Nel resto della funzione `main` si inizializza in primo luogo il generatore di numeri casuali (riga 5), e si itera l'esperimento su varie prove indipendenti. Il programma chiede l'input di un numero con la funzione `setExperiment` alla riga 9; il valore 4 provoca la fine dell'esperimento, mentre l'input del valore 3 lascia che la scelta del tipo di risultato sia fatta in modo aleatorio. La funzione `setExperiment`, che non riportiamo, decide il tipo di risultato della singola prova, e `experimentBody`, alla riga 10, sceglie in modo aleatorio i valori di output (la velocità della particella nel caso 0, il numero di particelle nel caso 1 e il tipo d'errore nel caso 2). L'output dell'esperimento è definito dalla `union experimentData scratchData` e dal valore di `experimentOutput`, che per i tre casi vale rispettivamente 0, 1 o 2.

La funzione `analyzeExperiment` del Listato 13.2 è molto semplice (proprio grazie all'uso di una `union`!) e riceve in input `scratchData` e `experimentOutput`.

```
1 void analyzeExperiment(unsigned long int experimentOutput,
2                         union experimentData newData) {
3   if (experimentOutput == EXP_SPEED) {
4     averageSpeed += newData.speed;
5   } else if (experimentOutput == EXP_JET) {
6     hystogramNumber[newData.num]++;
7   } else if (experimentOutput == EXP_ERROR) {
8     printf("ESPERIMENTO IN ERRORE: %s\n", newData.errorMessage);
9   }
10 }
```

Listato 13.2 La funzione `analyzeExperiment`.

La variabile `experimentOutput` ci dice di che tipo è stato il risultato dell'esperimento e quindi indica che tipo di valore è contenuto nella `union experimentData newData`. Nel caso di una velocità si tratta di un `double`, e nella riga 4 la nuova velocità è sommata ad `averageSpeed` per poi calcolarne la media. Nel caso di un gruppo di particelle si tratta di un `int`, e nella riga 6 aggiorniamo quindi un istogramma. Nel caso di un errore ci limitiamo a stampare il messaggio d'errore, nella riga 8.

Infine la funzione `myEnd` stampa la velocità media, l'istogramma della distribuzione del numero di particelle prodotte nelle interazioni, e il numero di errori fatti.

Laboratorio 13.2 *Un esperimento virtuale*

Descrivete un problema analogo a quello visto prima, che riguardi però il controllo della linea di produzione di un'industria. Scrivete un simulatore che tenga conto dei vari casi, e gestite con una `union` i vari casi permessi.

13.2 Liste concatenate

Il problema relativo alla gestione di piccole o grandi quantità di dati appare in un gran numero di contesti, e diventa spesso un collo di bottiglia, a volte drammaticamente stretto. I problemi interessanti possono essere molto diversi l'uno dall'altro: la lettura di un testo, la creazione di un dizionario, l'inserimento e la conservazione d'indirizzi, *database* di studenti i cui *record* si modificano nel tempo. Pensiamo anche ad applicazioni ingegneristiche come la raccolta di dati relativi a processi industriali, o al controllo e all'analisi di esperimenti in ogni ambito scientifico (segnali ottenuti nel trattamento di molecole di interesse biologico, informazioni su reazioni chimiche, controllo di esperimenti in cui particelle ad altissima energia collidono generando miriadi di nuove particelle).

13.2.1 Liste, stringhe e un dizionario

Discutiamo come sia possibile organizzare parole in una struttura che si possa consultare in modo efficiente. Usiamo inizialmente una struttura poco efficiente ma assolutamente elementare, che ci aiuta a comprendere il concetto di *lista concatenata*. Leggiamo quindi delle parole da un file di dati e le aggiungiamo progressivamente alla nostra lista. In un primo esempio creiamo una lista inserendovi tutte le parole a mano a mano che sono lette. In un secondo esempio eliminiamo le parole ripetute, creando un dizionario a partire da un testo. L'essenza di una struttura che fa riferimento a una struttura dello stesso tipo è nella definizione della `struct` di tipo `word` (parola) che segue,

```
struct word {
  char *pointerToString;
  struct word *pointerToNextWord;
};
```

La struttura è fatta di due puntatori. Il primo puntatore punta a un'area di memoria dove scriviamo la parola che abbiamo letto, sotto forma di stringa di caratteri (questo è utile perché la lunghezza della parola può essere variabile, e in questo modo la struttura `word` è indipendente da questi dettagli). Il secondo puntatore, che punta a una struttura *dello stesso tipo*, è quello cruciale: si tratta in questo caso della prossima parola della lista. Notate subito che non riserviamo nessuna memoria per i dati, ma soltanto la memoria necessaria ai due puntatori. La memoria per i dati va riservata separatamente. In un caso ancora piú semplice, in cui registriamo in ordine alcuni numeri telefonici, l'area di memoria necessaria a contenere il dato (il numero telefonico) avrebbe potuto essere riservata nella struttura stessa,

```
struct phoneNumber{
  unsigned long int thisNumber;
  struct phoneNumber *pointerToNextPhoneNumber;
};
```

In questo caso l'area di memoria necessaria per conservare il numero di telefono è riservata al momento della dichiarazione della struttura di tipo `phoneNumber` perché è sempre

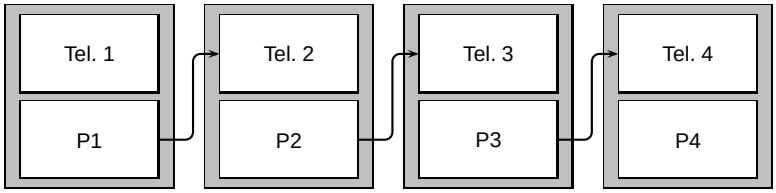

Figura 13.1 Una lista concatenata di numeri telefonici. I numeri sono conservati nei contenitori in alto, i contenitori in basso conservano un puntatore alla struttura successiva.

della stessa dimensione. È rimasta però invariata la caratteristica fondamentale di una lista concatenata, cioè il fatto che almeno un membro della struttura è un puntatore a una struttura dello stesso tipo (in questo caso il prossimo elemento della lista). La Figura 13.1 aiuta a chiarire l'organizzazione dei dati nel caso dei numeri di telefono, con i contenitori per i dati e i puntatori alla struttura successiva.

Una struttura di questo tipo lista è molto piú flessibile di quella di un array: in un array i dati sono incasellati in locazioni di memoria necessariamente consecutive, e non è possibile cambiare l'ordine di queste caselle, com'è invece necessario fare, per esempio, nel caso di strutture dinamiche, il cui ordine cambia durante l'esecuzione del codice. Nel caso di un array aggiungere locazioni di memoria (per esempio mediante una `realloc`) può, fra l'altro, diventare oneroso proprio a causa di questa necessaria contiguità dei dati. Può accadere che lo spazio totale richiesto dalla `realloc` sia disponibile soltanto in un'area diversa da quella impiegata; in questo caso per aumentare l'estensione del vettore è necessario copiarlo in una nuova parte della memoria, e ciò per un vettore di estensione elevata costa molto tempo di CPU.

Laboratorio 13.3 *Una lista concatenata*

Scrivete un programma che legga un testo (per esempio *L'infinito* di Giacomo Leopardi), riconosca le parole che lo compongono e le organizzi in una lista concatenata. L'organizzazione dettagliata della lista si può scegliere in vari modi, e può essere istruttivo provarne alcuni.

Vediamo quali sono i punti importanti del codice che analizza un testo. In primo luogo definiamo una struttura di tipo `word`:

```
struct word {
  char *pointerToString;
  struct word *pointerToNextWord;
} *wordList = NULL;
```

Non solo abbiamo definito un tipo di struttura, `word`, ma abbiamo anche dichiarato un puntatore a una struttura di questo tipo: `wordList`. Notate che non abbiamo riservato una struttura di tipo `word` ma soltanto un puntatore a una struttura inizializzato a `NULL` perché non avendo ancora letto alcuna parola la nostra lista è vuota, e perciò le associamo il valore `NULL`, che è definito dal compilatore. La parte essenziale del programma, tipicamente nella funzione `main`, ha la forma

```
myEnd = 0;
while (myEnd != 1) {
  myEnd = readWord();
  buildInverseList();
}
printInverseList();
```

La funzione `readWord` legge una parola, e la funzione `buildInverseList` la inserisce nella lista; si tratta di una lista inversa, perché i collegamenti che portano da un elemento all'altro partono dall'ultimo elemento inserito e risalgono fino al primo. Questo è il modo piú semplice per creare la lista. Notate che per semplicità non stiamo fornendo argomenti alle funzioni, e usiamo un certo numero di variabili globali. In generale è bene, in un compromesso fra semplicità di scrittura e robustezza del codice, cercare comunque di usare il minor numero possibile di variabili globali; ciò poiché ogni funzione del codice potrebbe modificarle, portando a possibili errori di programmazione.

La funzione `readWord` (Listato 13.3) legge il testo dalla variabile globale `fInput`, di tipo puntatore a `FILE` e ne individua le parole. La funzione che individua oggetti (in questo caso le parole) in un contesto è detta funzione di *analisi lessicale:* si tratta di una componente importante del processo di *parsing*, che è l'analisi della struttura grammaticale di un flusso di caratteri, e consiste proprio nella scomposizione del flusso in unità piú o meno complesse (gli elementi lessicali) e nella loro interpretazione alla luce delle regole della grammatica.

L'analisi lessicale è complessa perché una parola in un testo può essere delimitata in molti modi: nella maggior parte dei casi alla fine di una parola appaiono uno o piú spazi, ma può apparire un segno di punteggiatura (una virgola, un punto), o possono esserci parentesi o un semplice "fine riga". Realizzare un analizzatore lessicale completo è un compito difficile e non è il nostro scopo; per ora ci limitiamo a illustrare un analizzatore lessicale elementare, e lasciamo il suo miglioramento come esercizio. Nella funzione `readWord`, per semplificare la gestione della memoria, si assegna un numero massimo di caratteri per parola pari a `MY_MAX` caratteri: se per qualche motivo non è possibile rispettarlo, il programma si ferma in condizione d'errore. La variabile `myFlag` è azzerata quando la parola è completa e in questo caso la funzione rende il controllo alla funzione `main`, restituendo il valore `0` a meno che il file non sia finito, nel qual caso restituisce il valore `0`. Nel nostro analizzatore lessicale elementare una parola può finire soltanto con un solo spazio o con la fine del file: lasciamo come esercizio il compito di trattare in modo corretto il caso di piú spazi consecutivi. I caratteri di fine riga sono ignorati ma non sono considerati come segnali della fine di una parola; anche l'ultima parola di una riga deve

essere seguita da uno spazio. Quando la parola è completa aggiungiamo alla stringa il terminatore "\0". L'array di caratteri `myString` è stato dichiarato come variabile globale, ed è quindi visto da tutte le funzioni del programma. La variabile `myString` è usata per memorizzare temporaneamente le parole lette, finché non vengono inserite nella lista.

I caratteri che formano la parola sono letti a uno a uno con la funzione `fgetc`. La funzione `fgetc` (inclusa in `<stdio.h>`) ha come argomento il puntatore a un file (che deve essere stato aperto e deve essere accessibile) e restituisce il carattere letto dal file come un `char` convertito in un `int`, o, qualora sia stato letto il segnale di fine del file, restituisce `EOF`, che è definito negli header di sistema. Il suo prototipo è

```
int fgetc(FILE *stream);
```

La funzione riportata nel Listato 13.3 dovrebbe a questo punto risultare chiara.

```
1  int readWord(void) {
2    char myFlag = 1;
3    int j, myEnd = 0;
4    j = 0;
5    while ((j < MY_MAX) && (myFlag == 1)) {
6      myString[j] = fgetc(fInput);
7      if (myString[j] == ' ') {
8        myString[j] = '\0';
9        myFlag = 0;
10     } else if (myString[j] == '\n') {
11       j--; /*per ignorare il fine riga*/
12     } else if (myString[j] == EOF) {
13       myEnd = 1;
14       myString[j] = '\0';
15       myFlag = 0;
16     }
17     j++;
18   }
19   if (j >= MY_MAX-1) {
20     printf("Interruzione del programma: "
21            "la parola era troppo lunga.\n");
22     printf("Ricompilare con un valore di MY_MAX "
23            "piu grande di %d\n", MY_MAX);
24     exit(EXIT_FAILURE);
25   }
26   printf("Parola di lunghezza %d: %s\n",
27          strlen(myString), myString);
28   return myEnd;
29 }
```

Listato 13.3 La funzione `readWord` per l'analisi lessicale di un testo.

Notiamo che usare la funzione `fgetc` è un modo di procedere molto piú saggio che usare una `fscanf`, il cui uso è a volte complesso e può trarre in inganno, portando facilmente a errori di lettura.

La funzione `buildInverseList` costruisce la lista (Listato 13.4): si tratta di una funzione semplice e compatta che riserva, con una chiamata a `malloc`, la memoria necessaria a contenere la stringa appena letta, memorizzata nell'array di memoria globale `myString`. La funzione `strlen` (inclusa in `<string.h>`) ha come argomento il puntatore a una stringa e restituisce la lunghezza della stringa:

```
size_t strlen(const char *myString);
```

(discutiamo `size_t` nel Paragrafo 10.2.2).

Il puntatore `scratchPointer` indirizza la nuova area di memoria permanente (ossia riservata dal programma fino alla fine dell'esecuzione del codice) nella quale conserviamo la nuova parola. Per memorizzare in modo permanente (cioè fino alla fine dell'esecuzione del codice) le parole, copiamo il contenuto di `myString` in un'area della memoria appositamente riservata e puntata da `scratchPointer`. Se la richiesta di memoria dovesse fallire il puntatore `scratchPointer` verrebbe posto uguale a `NULL`, e in questa condizione il programma terminerebbe con un errore. Fatto questo, salviamo il puntatore all'ultima struttura `word` conservato in `wordScratchPointer` (saltare questo passo romperebbe la comunicazione fra i vari elementi della lista). Chiediamo poi nuova memoria per una nuova struttura di tipo `word` per contenere la nuova parola. Questo è il solo momento in cui riserviamo una nuova struttura `word` (prima avevamo riservato soltanto un puntatore a una struttura); anche qui verifichiamo che la `malloc` abbia avuto successo.

Resta poco da fare. Copiamo la parola appena letta dal buffer provvisorio `myString` nel buffer puntato da `scratchPointer`. Facciamo questo usando la funzione di sistema `strcpy` (inclusa in `<string.h>`) che copia una stringa (incluso il suo terminatore `\0`) da una locazione a un'altra. Il suo prototipo ha la forma

```
char *strcpy(char *destination, const char *source);
```

e il puntatore restituito è proprio il puntatore alla stringa di destinazione (nel nostro codice non è usato).

Non resta che riempire la struttura di tipo `word` che abbiamo appena creato copiando il puntatore alla nuova parola in `wordList->pointerToString` e copiando il puntatore alla penultima struttura (che avevamo saggiamente salvato alla riga 11) in `wordList -> pointerToNextWord;` in questo modo la nuova struttura che abbiamo aggiunto contiene un puntatore alla parola trovata, e un puntatore alla struttura creata precedentemente. La prima struttura creata punta al valore `NULL` che avevamo usato per inizializzare il primo puntatore dichiarato all'inizio del codice: un puntatore a `NULL` nella lista dei puntatori `wordList->pointerToNextWord` segnala che la lista è finita. A questo punto la funzione `buildInverseList` non dovrebbe avere piú segreti.

```
1 void buildInverseList(void) {
2   char *scratchPointer;
3   struct word *wordScratchPointer;
4
5   scratchPointer = (char *)malloc(strlen(myString));
6   if (scratchPointer == NULL) {
7     printf("Interruzione del programma: fallita "
8            "malloc numero 1\n");
9     exit(EXIT_FAILURE);
10  }
11  wordScratchPointer = wordList;
12  wordList = (struct word *)malloc(sizeof(struct word));
13  if (wordList == NULL) {
14    printf("Interruzione del programma: fallita "
15           "malloc numero 2\n");
16    exit(EXIT_FAILURE);
17  }
18  strcpy(scratchPointer, myString);
19  wordList->pointerToString = scratchPointer;
20  wordList->pointerToNextWord = wordScratchPointer;
21 }
```

Listato 13.4 La funzione `buildInverseList` che costruisce una lista concatenata.

Una comprensione piú chiara del funzionamento della concatenazione di questa lista si può ottenere guardando la funzione destinata a stampare la lista, che riportiamo nel Listato 13.5.

```
1 void printInverseList(void) {
2   struct word *wordScratchPointer = wordList;
3   while (wordScratchPointer != NULL) {
4     printf("%s\n", wordScratchPointer->pointerToString);
5     wordScratchPointer =
6       wordScratchPointer->pointerToNextWord;
7   }
8 }
```

Listato 13.5 La funzione `printInverseList` che stampa una lista concatenata.

Il meccanismo della funzione dovrebbe essere chiaro: il puntatore `wordList` (che qui è una variabile globale) punta alla struttura che contiene l'ultima parola letta. La si stampa e si passa alla parola precedente, mediante `wordScratchPointer->pointerToNextWord`, finché non si trova un puntatore a `NULL` che segnala che la lista di parole è finita, e la ricostruzione è stata completata.

Laboratorio 13.4 *Ancora liste e dizionari*

L'analizzatore lessicale che discutiamo è estremamente semplice. Miglioratelo tenendo conto della possibile presenza di punteggiatura, delle maiuscole e delle minuscole, del fatto che il carattere di fine riga termina una parola (a meno che un simbolo prescelto, per esempio un '`-`', non indichi che la parola è stata divisa in due parti). Considerate le lettere accentate, e il fatto che gli accenti a volte sono inseriti con un simbolo che segue la parola: piú in generale questo è un buon momento per riguardare tutta la struttura della tavola dei caratteri ASCII, che discutiamo nel Paragrafo 1.7.2 e nell'Appendice B, e costruire l'analizzatore lessicale tenendo conto del maggior numero possibile di casi interessanti.

Scrivete un codice che costruisce una lista diretta, cioè che comincia dalla prima parola letta e si svolge fino all'ultima. Il codice è molto simile a quello descritto per la lista inversa. Modificate infine il codice per eliminare le parole ripetute.

Per eliminare le parole ripetute basta aggiungere all'inizio di `buildInverseList` (Listato 13.4), quattro semplici righe di codice. Un puntatore locale percorre le parole trovate sinora e le confronta con l'ultima arrivata: se è stabilita una coincidenza la nuova parola non è registrata. Le due stringhe sono confrontate mediante la funzione di sistema `strcmp`, che è inclusa in `<string.h>` e ha come prototipo

```
int strcmp(const char *string1, const char *string2);
```

La funzione confronta le due stringhe cui puntano `string1` e `string2`; essa usa un meccanismo in cui a ogni stringa si fa corrispondere un valore numerico, in modo tale che a una stringa precedente in ordine alfabetico si assegna un valore numerico piú piccolo usando la codifica ASCII (Paragrafo 1.7.2 e Appendice B). La funzione restituisce quindi un intero minore di zero, uguale a zero o piú grande di zero se `string1` risulta, rispettivamente, piú piccola, uguale o piú grande di `string2`. Notate, nel codice che segue, che in questo caso la funzione è abbandonata con un `return` collocato all'interno di un blocco. Non tutti considerano questo costrutto come strutturato poiché abbiamo un blocco d'istruzioni in cui possiamo entrare da una parte sola, ma dal quale possiamo uscire in due modi: dalla fine o mediante il `return` Si tratta di una forma di cui è bene non abusare ma che talvolta è utile; è quella che si dice una *clausola di salvaguardia*. Ci riserviamo il diritto di uscire immediatamente dalla funzione se si verifica una certa condizione (nel caso in esame che la parola sia già contenuta nella lista).

```
struct word *wordScratchPointer, *pointerCheck;
for (pointerCheck = wordList; pointerCheck != NULL;) {
  if (strcmp(pointerCheck->pointerToString, myString) == 0) {
    return;
  }
  pointerCheck = pointerCheck->pointerToNextWord;
}
```

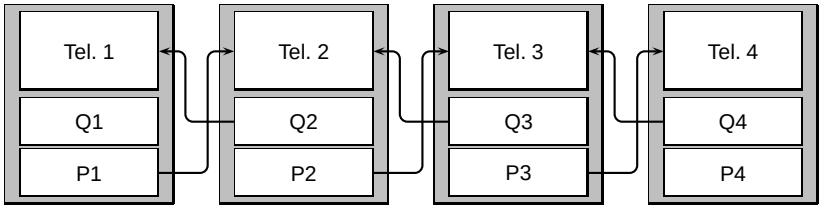

Figura 13.2 Una lista concatenata nelle due direzioni. La struttura è piú ricca che in una lista a collegamento unico ed è possibile percorrerla semplicemente nelle due direzioni, dalla testa alla coda e dalla coda alla testa. I puntatori P_i fanno riferimento alla prossima struttura, i puntatori Q_i alla struttura precedente.

Concludiamo questo paragrafo ricordando che quello delle liste con un solo collegamento è l'esempio piú semplice fra una gran quantità di casi possibili. Mostriamo nella Figura 13.2 un esempio di una lista concatenata nelle due direzioni. Grazie alla doppia struttura di puntatori si può percorrere questa lista nelle due direzioni, dalla testa alla coda o dalla coda alla testa. In questo caso ogni struttura conserva, oltre ai dati rilevanti, anche un puntatore alla struttura precedente e un puntatore alla struttura successiva.

13.2.2 Funzioni ricorsive: il calcolo del fattoriale

Per continuare la nostra analisi dei modi in cui si possono organizzare i dati, e di quelli con cui possiamo costruire e gestire le liste, è utile introdurre il concetto di *ricorsione*. Ci limitiamo a una breve introduzione; discutiamo l'argomento in modo molto piú dettagliato nel Paragrafo 15.3.1.

Si dice ricorsiva una funzione che è capace di chiamare se stessa. Non tutti i linguaggi di programmazione permettono di usare questo genere di funzioni: il C lo fa, e questa è una sua caratteristica molto interessante. Illustriamo il punto con il semplice esempio del calcolo del fattoriale, $n! \equiv 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$, definito per valori di n interi non negativi. Il Paragrafo 4.3.1 contiene un semplice codice non ricorsivo che calcola il fattoriale, mentre la funzione ricorsiva è illustrata nel Listato 13.6.

```
1 unsigned long long int factorial(unsigned int n) {
2   if (n == 0) return 1;
3   return (n * factorial(n - 1));
4 }
```

Listato 13.6 Calcolo del fattoriale con una funzione ricorsiva.

Il punto chiave è nella semplice identità matematica $n! = n \cdot (n-1)!$. Se è chiamata con un intero positivo, la funzione chiama se stessa finché non è chiamata con il valore zero, e in questo caso restituisce uno; a questo punto la funzione chiamata con argomento