basta che ogni cliente specifichi come prima scelta nelle preferenze una cliente diversa dalla prima scelta degli altri: in effetti, il problema dei matrimoni stabili presenta innumerevoli varianti che sono state studiate per garantire ulteriori proprietà oltre a quella di avere un abbinamento perfetto.

ALVIE: problema dei matrimoni stabili

Osserva, sperimenta e verifica
`StableMarriage`

3.3 Liste randomizzate

Una configurazione sfavorevole dei dati o della sequenza di operazioni che agisce su di essi può rendere inefficiente diversi algoritmi se analizziamo la loro complessità nel caso pessimo. In generale, la strategia che consente di individuare le configurazioni che peggiorano le prestazioni di un algoritmo è chiamata *avversariale* in quanto suppone che un avversario malizioso generi tali configurazioni sfavorevoli in modo continuo. In tale contesto, la casualità riveste un ruolo rilevante per la sua caratteristica imprevedibilità: vogliamo sfruttare quest'ultima a nostro vantaggio, impedendo a un tale avversario di prevedere le configurazioni sfavorevoli (in senso algoritmico). Abbiamo già discusso nel Paragrafo 2.5.4 come la casualità possa essere impiegata in tal senso, applicandola all'algoritmo di ordinamento per distribuzione (*quicksort*) nella scelta del pivot. Ricordiamo che un algoritmo random o casuale (di cui l'algoritmo *quicksort* costituisce un esempio) fa uso di sequenze di scelte casuali.

Nel seguito descriviamo un algoritmo random per il problema dell'inserimento e della ricerca di una chiave k in una lista e dimostriamo che la strategia da esso adottata è vincente, sotto opportune condizioni. In particolare, usando una lista randomizzata di n elementi ordinati, i tempi *medi* o *attesi* delle operazioni di ricerca e inserimento sono ridotti a $O(\log n)$: anche se, al caso pessimo, tali operazioni possono richiedere tempo $O(n)$, è altamente improbabile che ciò accada.

Descriviamo una particolare realizzazione di liste randomizzate, chiamate **liste a salti** (*skip list*), la cui idea base (non random) può essere riassunta nel modo seguente (secondo quanto illustrato nella Figura 3.8). Partiamo da una lista *ordinata* di $n+2$ elementi, $L_0 = e_0, e_1, \ldots, e_{n+1}$, la quale costituisce il livello 0 della lista a salti: poniamo che il primo e l'ultimo elemento della lista siano i due valori speciali, $-\infty$ e $+\infty$, per cui vale sempre $-\infty < e_i < +\infty$, per $1 \leqslant i \leqslant n$. Per ogni elemento e_i della lista L_0

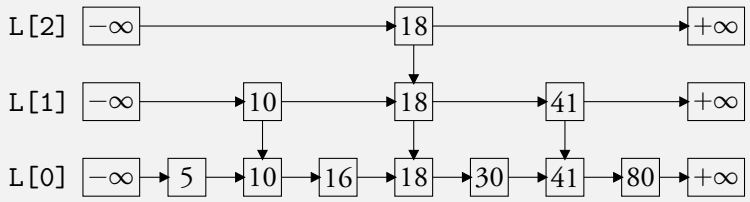


Figura 3.8 Un esempio di lista a salti.

($1 \leqslant i \leqslant n$) creiamo r_i copie di e_i se $r_i > 0$, dove 2^{r_i} è la massima potenza di 2 che divide i (nel nostro esempio, per $i = 1,2,3,4,5,6,7$, abbiamo $r_i = 0,1,0,2,0,1,0$). Ciascuna copia ha livello crescente $\ell = 1,2,\ldots,r_i$ e punta alla copia di livello inferiore $\ell - 1$: supponiamo inoltre che $-\infty$ e $+\infty$ abbiano sempre una copia per ogni livello creato. Chiaramente, il massimo livello o *altezza* h della lista a salti è dato dal massimo valore di r_i incrementato di 1 e, quindi, $h = O(\log n)$.

Passando a una visione orizzontale, tutte le copie dello stesso livello ℓ ($0 \leqslant \ell \leqslant h$) sono collegate e formano una sottolista L_ℓ, tale che $L_\ell \subseteq L_{\ell-1} \subseteq \cdots \subseteq L_0$:[1] come possiamo vedere nell'esempio mostrato nella Figura 3.8, le liste dei livelli superiori "saltellano" (*skip* in inglese) su quelle dei livelli inferiori. Osserviamo che, se la lista di partenza, L_0, contiene $n+2$ elementi ordinati, allora L_1 ne contiene al più $2 + n/2$, L_2 ne contiene al più $2 + n/4$ e, in generale, L_ℓ contiene al più $2 + n/2^\ell$ elementi ordinati. Pertanto, il numero totale di copie presenti nella lista a salti è limitato superiormente dal seguente valore:

$$\begin{aligned}(2+n) + (2+n/2) + \cdots + (2+n/2^h) &= 2(h+1) + \sum_{\ell=0}^{h} n/2^\ell \\ &= 2(h+1) + n\sum_{\ell=0}^{h} 1/2^\ell < 2(h+1) + 2n\end{aligned}$$

In altre parole, il numero totale di copie è $O(n)$.

Per descrivere le operazioni di ricerca e inserimento, necessitiamo della nozione di predecessore. Data una lista $L_\ell = e'_0, e'_1, \ldots, e'_{m-1}$ di elementi ordinati e un elemento x, diciamo che $e'_j \in L_\ell$ (con $0 \leqslant j < m-1$) è il **predecessore** di x (in L_ℓ) se e'_j è il massimo tra i minoranti di x, ovvero $e'_j \leqslant x < e'_{j+1}$: osserviamo che il precedessore è sempre ben definito perché il primo elemento di L_ℓ è $-\infty$ e l'ultimo elemento è $+\infty$.

[1] Con un piccolo abuso di notazione, scriviamo $L \subseteq L'$ se l'insieme degli elementi di L è un sottoinsieme di quello degli elementi di L'.

```
1  ScansioneSkipList( k ):      ⟨pre: gli elementi −∞ e +∞ fungono da sentinelle⟩
2    p = L[h];
3    WHILE (p != null) {
4      WHILE (p.succ.key <= k)
5        p = p.succ;
6      predecessore = p;
7      p = p.inf;
8    }
9    RETURN predecessore;
```

Codice 3.3 Scansione di una lista a salti per la ricerca di una chiave k.

La ricerca di una chiave k è concettualmente semplice. Ad esempio, supponiamo di voler cercare la chiave 80 nella lista mostrata nella Figura 3.8. Partendo da L_2, troviamo che il predecessore di 80 in L_2 è 18: a questo punto, passiamo alla copia di 18 nella lista L_1 e troviamo che il predecessore di 80 in quest'ultima lista è 41. Passando alla copia di 41 in L_0, troviamo il predecessore di 80 in questa lista, ovvero 80 stesso: pertanto, la chiave è stata trovata.

Tale modo di procedere è mostrato nel Codice 3.3, in cui supponiamo che gli elementi in ciascuna lista L_ℓ abbiamo un riferimento `inf` per raggiungere la corrispondente copia nella lista inferiore $L_{\ell-1}$. Partiamo dalla lista L_h (riga 2) e troviamo il predecessore p_h di k in tale lista (righe 4–6). Poiché $L_h \subseteq L_{h-1}$, possiamo raggiungere la copia di p_h in L_{h-1} (riga 7) e, a partire da questa posizione, scandire quest'ultima lista in avanti per trovare il predecessore p_{h-1} di k in L_{h-1}. Ripetiamo questo procedimento per tutti i livelli ℓ a decrescere: partendo dal predecessore p_ℓ di k in L_ℓ, raggiungiamo la sua copia in $L_{\ell-1}$, e percorriamo quest'ultima lista in avanti per trovare il predecessore $p_{\ell-1}$ di k in $L_{\ell-1}$ (righe 3–8). Quando raggiungiamo L_0 (ovvero, `p` è uguale a `null` nella riga 3), la variabile `predecessore` memorizza p_0, che è il predecessore che avremmo trovato se avessimo scandito L_0 dall'inizio di tale lista.

Il lettore attento avrà certamente notato che l'algoritmo di ricerca realizzato dal Codice 3.3 è molto simile alla ricerca binaria descritta nel caso degli array (Paragrafo 2.4.1): in effetti, ogni movimento seguendo il campo `succ` corrisponde a dimezzare la porzione di sequenza su cui proseguire la ricerca. Per questo motivo, è facile dimostrare che il costo della ricerca effettuata dal Codice 3.3 è $O(\log n)$ tempo, contrariamente al tempo $O(n)$ richiesto dalla scansione sequenziale di L_0.

Il problema sorge con l'operazione di inserimento, la cui realizzazione ricalca l'algoritmo di ricerca. Una volta trovata la posizione in cui inserire la nuova chiave, però, l'inserimento vero e proprio risulterebbe essere troppo costoso se volessimo continuare a mantenere le proprietà della lista a salti descritte in precedenza, in quanto questo po-

trebbe voler dire modificare le copie di tutti gli elementi che seguono la chiave appena inserita. Per far fronte a questo problema, usiamo la casualità: il risultato sarà un algoritmo random di inserimento nella lista a salti che non garantisce la struttura perfettamente bilanciata della lista stessa, ma che con alta probabilità continua a mantenere un'altezza media logaritmica e un tempo medio di esecuzione di una ricerca anch'esso logaritmico.

Notiamo che, senza perdita di generalità, la casualità può essere vista come l'esito di una sequenza di lanci di una moneta equiprobabile, dove ciascun lancio ha una possibilità su due che esca testa (codificata con `1`) e una possibilità su due che esca croce (codificata con `0`). Precisamente, diremo che la probabilità di ottenere `1` è $q = \frac{1}{2}$ e la probabilità di ottenere `0` è $1 - q = \frac{1}{2}$ (in generale, un truffaldino potrebbe darci una moneta per cui $q \neq \frac{1}{2}$).

Attraverso una sequenza di b lanci, possiamo ottenere una **sequenza random** di b bit casuali.[2] Ciascun lancio è nella pratica simulato mediante la chiamata a una primitiva `random()` per generare un valore reale r pseudocasuale appartenente all'intervallo $0 \leqslant r < 1$, in modo uniforme ed equiprobabile (tale generatore è disponibile in molte librerie per la programmazione e non è semplice ottenerne uno statisticamente significativo, in quanto il programma che lo genera è deterministico): il numero r generato pseudocasualmente fornisce quindi il bit `0` se $0 \leqslant r < \frac{1}{2}$ e il bit `1` se $\frac{1}{2} \leqslant r < 1$.

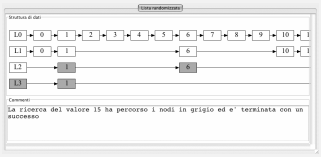
Osserviamo che i lanci di moneta sono eseguiti in modo indipendente, per cui otteniamo una delle quattro possibili sequenze di $b = 2$ bit (`00`, `01`, `10` oppure `11`) in modo casuale, con probabilità $\frac{1}{2} \times \frac{1}{2} = \frac{1}{4}$: in generale, le probabilità dei lanci si moltiplicano in quanto sono eventi indipendenti, ottenendo una sequenza di b bit casuali con probabilità $1/2^b$. Osserviamo inoltre che prima o poi dobbiamo incontrare un `1` nella sequenza se b è sufficientemente grande.

Utilizziamo tale concetto di casualità per inserire una chiave k in una lista a salti. Una volta identificati i suoi predecessori $p_0, p_1, \ldots, p_h$, in maniera analoga a quanto descritto per l'operazione di ricerca, eseguiamo una sequenza di lanci di moneta finché non otteniamo `1`. Sia $r \geqslant 1$ il numero di bit casuali (lanci) così generati per k, per cui i primi $r-1$ bit sono `0` e l'ultimo è `1`. Se $r \leqslant h+1$, creiamo r copie di k e le inseriamo nelle liste $L_0, L_1, L_2, \ldots, L_r$: ciascuna inserzione richiede tempo costante in quanto va creato un nodo immediatamente dopo ciascuno dei predecessori $p_0, p_1, \ldots, p_r$. Altrimenti, creiamo $h+1$ copie della chiave k, aggiorniamo tutte le liste già esistenti secondo quanto detto prima e creiamo una nuova lista L_{h+1} composta dalle chiavi $-\infty$, k e $+\infty$. Come vedremo, il costo dell'operazione è, in media, $O(\log n)$.

[2] La nozione di sequenza random R è stata formalizzata nella teoria di Kolmogorov in termini di incompressibilità, per cui qualunque programma che generi R non può richiedere significativamente meno bit per la sua descrizione di quanti ne contenga R. Per esempio, $R = \texttt{010101}\cdots\texttt{01}$ non è casuale in quanto un programma che scrive per $b/2$ volte `01` può generarla richiedendo solo $O(\log b)$ bit per la sua descrizione. Purtroppo è indecidibile stabilire se una sequenza è random anche se la stragrande maggioranza delle sequenze binarie lo sono.

ALVIE: liste a salti

Osserva, sperimenta e verifica
`SkipList`



Per stabilire la complessità media delle operazioni di ricerca e inserimento su una lista a salti (random), valutiamo un limite superiore per l'altezza media e per il numero medio di copie create con il procedimento appena descritto. La lista di livello più basso, L_0, contiene $n+2$ elementi ordinati. Per ciascun inserimento di una chiave k, indipendentemente dagli altri inserimenti abbiamo lanciato una moneta equiprobabile per decidere se L_1 debba contenere una copia di k (bit `0`) o meno (bit `1`): quindi L_1 contiene circa $n/2+2$ elementi (una frazione costante di n in generale), perché i lanci sono equiprobabili e all'incirca metà degli elementi in L_0 ha ottenuto `0`, creando una copia in L_1, e il resto ha ottenuto `1`. Ripetendo tale argomento ai livelli successivi, risulta che L_2 contiene circa $n/4+2$ elementi, L_3 ne contiene circa $n/8+2$ e così via: in generale, L_ℓ contiene circa $n/2^\ell+2$ elementi ordinati e, quando $\ell = h$, l'ultimo livello ne contiene un numero costante $c > 0$, ovvero $n/2^h + 2 = c$. Ne deriviamo che l'altezza h è in media $O(\log n)$ e, in modo analogo a quanto mostrato in precedenza, che il numero totale medio di copie è $O(n)$ (la dimostrazione formale di tali proprietà sull'altezza e sul numero di copie richiede in realtà strumenti più sofisticati di analisi probabilistica).

Mostriamo ora che la ricerca descritta nel Codice 3.3 richiede tempo medio $O(\log n)$. Per un generico livello ℓ nella lista a salti, indichiamo con $T(\ell)$ il numero medio di elementi esaminati dall'algoritmo di scansione, a partire dalla posizione corrente nella lista L_ℓ fino a giungere al predecessore p_0 di k nella lista L_0: il costo della ricerca è quindi proporzionale a $O(T(h))$.

Per valutare $T(h)$, osserviamo che il cammino di attraversamento della lista a salti segue un profilo a gradino, in cui i tratti orizzontali corrispondono a porzioni della stessa lista mentre quelli verticali al passaggio alla lista del livello inferiore. Percorriamo a ritroso tale cammino attraverso i predecessori $p_0, p_1, p_2, \ldots, p_h$, al fine di stabilire induttivamente i valori di $T(0), T(1), T(2), \ldots, T(h)$ (dove $T(0) = O(1)$, essendo già posizionati su p_0): notiamo che per $T(\ell)$ con $\ell \geqslant 1$, lungo il percorso (inverso) nel tratto interno a L_ℓ, abbiamo solo due possibilità rispetto all'elemento corrente $e \in L_\ell$.

1. Il percorso inverso proviene dalla copia di e nel livello inferiore (riga 7 del Codice 3.3), nella lista $L_{\ell-1}$, a cui siamo giunti con un costo medio pari a $T(\ell-1)$. Tale evento ha probabilità $\frac{1}{2}$ in quanto la copia è stata creata a seguito di un lancio della moneta che ha fornito `0`.

2. Il percorso inverso proviene dall'elemento $\hat{e}$ a destra di e in L_ℓ (riga 5 del Codice 3.3), a cui siamo giunti con un costo medio pari a $T(\ell)$: in tal caso, $\hat{e}$ non può avere una corrispettiva copia al livello superiore (in $L_{\ell+1}$). Tale evento ha probabilità $\frac{1}{2}$ a seguito di un lancio della moneta che ha fornito `1`.

Possiamo quindi esprimere il valore medio di $T(\ell)$ attraverso la media pesata (come per il *quicksort*) dei costi espressi nei casi 1 e 2, ottenendo la seguente equazione di ricorrenza per un'opportuna costante $c' > 0$:

$$T(\ell) \leqslant \frac{1}{2}T(\ell) + \frac{1}{2}T(\ell-1) + c' \tag{3.1}$$

Otteniamo una limitazione superiore per il termine $T(\ell)$ sostituendo la disuguaglianza con l'uguaglianza nell'equazione (3.1), analogamente a quanto discusso nel Paragrafo 2.5.4. Moltiplicando i termini dell'equazione così trasformata per 2 e risolvendo rispetto a $T(\ell)$ otteniamo

$$T(\ell) = T(\ell-1) + 2c' \tag{3.2}$$

Espandendo l'equazione (3.2), abbiamo $T(\ell) = T(\ell-1) + 2c' = T(\ell-2) + (2+2)c' = \cdots = T(0) + (2\ell)c' = O(\ell)$. Quindi $T(h) = O(h) = O(\log n)$ è il costo medio della ricerca.

In conclusione, le liste randomizzate sono un esempio concreto di come l'uso accorto della casualità possa portare ad algoritmi semplici che hanno in media (o con alta probabilità) ottime prestazioni in tempo e in spazio.

3.4 Opus libri: gestione di liste ammortizzate e di liste ad auto-organizzazione

L'efficacia dell'auto-organizzazione nella gestione delle liste, da sempre accertata a livello euristico e sperimentale, può essere mostrata in modo rigoroso facendo uso dell'analisi ammortizzata, permettendo di ottenere un'implementazione efficiente delle operazioni di ricerca, inserimento e cancellazione. Prima di descrivere e analizzare in dettaglio una tale organizzazione delle liste, discutiamo un semplice caso di liste ammortizzate.

3.4.1 Unione e appartenenza a liste disgiunte

Le liste possono essere impiegate per operazioni di tipo insiemistico: avendo già visto come inserire e cancellare un elemento, siamo interessati a gestire una sequenza arbitraria S di operazioni di unione e appartenenza su un insieme di liste contenenti un totale di m elementi. In ogni istante le liste sono *disgiunte*, ossia l'intersezione di due qualunque liste