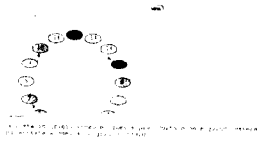
Da: Crescenzi, Gambosi, Grossi:
STRUTTURE DI DATI E ALGORITMI
PEARSON ED. 2006

lista contenente z è almeno 2^i. D'altra parte, al termine delle n operazioni di unione, la lunghezza di una qualunque lista è minore oppure uguale a $n + 1$: ne deriva che la lista contenente z ha lunghezza compresa tra 2^i e $n + 1$ (ovvero, $2^i \leqslant n + 1$) e che vale sempre $i = O(\log n)$. Quindi, ogni elemento z vede cambiare il riferimento `z.lista` al più $O(\log n)$ volte. Sommando tale quantità per gli $n + 1$ elementi coinvolti nelle n operazioni di unione, otteniamo un limite superiore di $O(n \log n)$ al numero di volte che la riga 11 viene globalmente eseguita: pertanto, la complessità in tempo delle n operazioni `Unisci` è $O(n \log n)$ e, quindi, il costo *ammortizzato* di tale operazione è $O(\log n)$. Al costo di queste operazioni, va aggiunto il costo $O(1)$ per ciascuna delle operazioni `Crea` e `Appartieni`.

Lo schema adottato per cambiare i riferimenti `z.lista` è piuttosto generale: ipotizzando di avere insiemi disgiunti i cui elementi hanno ciascuno un'etichetta (sia essa `z.lista` o qualunque altra informazione) e applicando la regola che, quando due insiemi vengono uniti, si cambiano solo le etichette agli elementi dell'insieme di cardinalità minore, siamo sicuri che un'etichetta non possa venire cambiata più di $O(\log n)$ volte. L'intuizione di cambiare le etichette agli elementi del più piccolo dei due insiemi da unire viene rigorosamente esplicitata dall'analisi ammortizzata: notiamo che, invece, cambiando le etichette agli elementi del più grande dei due insiemi da unire, un'etichetta potrebbe venire cambiata $\Omega(n)$ volte, invalidando l'argomentazione finora svolta.

ALVIE: **unione e appartenenza a liste disgiunte**

Osserva, sperimenta e verifica
`UnionFind`



3.4.2 Liste ad auto-organizzazione

L'auto-organizzazione delle liste è utile quando, per svariati motivi, la lista *non è necessariamente ordinata* in base alle chiavi di ricerca (contrariamente al caso delle liste randomizzate del Paragrafo 3.3). Per semplificare la discussione, consideriamo il solo caso della ricerca di una chiave k in una lista e adottiamo uno schema di scansione sequenziale, illustrato nel Codice 3.1: percorriamo la lista a partire dall'inizio verificando iterativamente se l'elemento attuale è uguale alla chiave cercata. Estendiamo tale schema per eseguire eventuali operazioni di auto-organizzazione al termine della scansione sequenziale (le operazioni di inserimento e cancellazione possono essere ottenute semplicemente, secondo quanto discusso nel Paragrafo 3.1).

```
MoveToFront( a, k ):
  p = a;
  IF (p == null || p.dato == k) RETURN p;
  WHILE ((p.succ != null) && (p.succ.dato != k))
    p = p.succ;
  IF (p.succ == null) RETURN null;
  tmp = a;
  a = p.succ;
  p.succ = p.succ.succ;
  a.succ = tmp;
  return a;
  RETURN
```

dice 3.5 Ricerca di una chiave k in una lista ad auto-organizzazione.

Tale organizzazione sequenziale può trarre beneficio dal **principio di località temrale**, per il quale, se accediamo a un elemento in un dato istante, è molto probabile accederemo a questo stesso elemento in istanti immediatamente (o quasi) successivi. guendo tale principio, sembra naturale che possiamo *riorganizzare* proficuamente gli menti della lista dopo aver eseguito la loro scansione. Per questo motivo, una lista gestita viene riferita come struttura di dati ad **auto-organizzazione** (*self-organizing elf-adjusting*). Tra le varie strategie di auto-organizzazione, la più diffusa ed efficaviene detta *move-to-front* (MTF), che consideriamo in questo paragrafo: essa consiste lo spostare l'elemento acceduto dalla sua posizione attuale alla cima della lista, senza biare l'ordine relativo dei rimanenti elementi, come mostrato nel Codice 3.5. Osserno che MTF effettua ogni ricerca senza conoscere le ricerche che dovrà effettuare in uito: un algoritmo operante in tali condizioni, che deve quindi servire un insieme di ieste man mano che esse pervengono, viene detto **in linea** (*online*).

Un esempio quotidiano di lista ad auto-organizzazione che utilizza la strategia MTF ostituito dall'elenco delle chiamate effettuate da un telefono cellulare: in effetti, è babile che un numero di telefono appena chiamato, venga usato nuovamente nel simo futuro. Un altro esempio, più informatico, è proprio dei sistemi operativi, dove rategia MTF viene comunemente denominata *least recently used* (LRU). In questo gli elementi della lista corrispondono alle pagine di memoria, di cui solo le prime r ono essere tenute in una memoria ad accesso veloce. Quando una pagina è richiesta, t'ultima viene aggiunta alle prime r, mentre quella a cui si è fatto accesso meno ntemente viene rimossa. Quest'operazione equivale a porre la nuova pagina in cima ista, per cui quella originariamente in posizione r (acceduta meno recentemente) va osizione successiva, r + 1, uscendo di fatto dall'insieme delle pagine mantenute nella oria veloce.

Per valutare le prestazioni della strategia MTF, il termine di paragone utilizzato sarà un algoritmo **fuori linea** (*offline*), denominato OPT, che ipotizziamo essere a conoscenza di *tutte* le richieste che perverranno. Le prestazioni dei due algoritmi verranno confrontate rispetto al loro costo, definito come la somma dei costi delle singole operazioni, in accordo a quanto discusso sopra: in particolare, contiamo il numero di elementi scanditi a partire dall'inizio della lista, per cui *accedere all'elemento in posizione* i *ha costo* i in quanto dobbiamo scandire gli i elementi che lo precedono. Lo spostamento in cima alla lista, operato da MTF, non viene conteggiato in quanto richiede un costo costante.

Tale paradigma è ben esemplificato dalla gestione delle chiamate in uscita di un telefono cellulare: l'ultimo numero chiamato è già disponibile in cima alla lista per la prossima chiamata e il costo indica il numero di *clic* sulla tastierina per accedere a ulteriori numeri chiamati precedentemente (occorrono un numero di clic pari a i per scandire gli elementi che precedono l'elemento in posizione i nell'ordine inverso di chiamata).

È di fondamentale importanza stabilire le regole di azione di OPT, perché questo può dare luogo a risultati completamente differenti. Nel nostro caso, OPT parte dalla stessa lista iniziale di MTF. Esaminate tutte le richieste in anticipo, OPT *permuta* gli elementi della lista solo una volta all'inizio, *prima* di servire le richieste. A questo punto, quando arriva una richiesta per l'elemento k in posizione i, restituisce l'elemento scandendo i primi i elementi della lista, senza però muovere k dalla sua posizione.

Notiamo che OPT permuta gli elementi in un ordine (per noi imprevedibile) che rende minimo il suo costo futuro. Inoltre, ai fini dell'analisi, presumiamo che le liste non cambino di lunghezza durante l'elaborazione delle richieste.

A titolo esemplificativo, è utile riportare i costi in termini concreti del numero di clic effettuati sui cellulari. Immaginiamo di essere in possesso, oltre al cellulare di marca MTF, di un futuristico cellulare OPT che conosce in anticipo le n chiamate che saranno effettuate nell'arco di un anno su di esso (l'organizzazione della lista delle chiamate in uscita è mediante le omonime politiche di gestione). Potendo usare entrambi i cellulari con gli stessi m numeri in essi memorizzati, effettuiamo alcune chiamate su tali numeri per un anno: quando effettuiamo una chiamata su di un cellulare, la ripetiamo anche sull'altro (essendo futuristico, OPT si aspetta già la chiamata che intendiamo effettuare). Per la chiamata j, dove $j = 0, 1, \ldots, n-1$, contiamo il numero di clic che siamo costretti a fare per accedere al numero di interesse in MTF e, analogamente, annotiamo il numero di clic per OPT (ricordiamo che MTF pone il numero chiamato in cima alla sua lista mentre OPT non cambia più l'ordine inizialmente adottato in base alle chiamate future). Allo scadere dell'anno, siamo interessati a stabilire il costo, ovvero il numero totale di clic effettuati su ciascuno dei due cellulari.

Mostriamo che, sotto opportune condizioni, il *costo di* MTF *non supera il doppio del costo di* OPT. In un certo senso, MTF offre una forma limitata di chiaroveggenza delle richieste rispetto a OPT, motivando il suo impiego in vari contesti con successo. In

Lista MTF	=	e_4	e_2	e_6	e_0	e_1	e_3	e_7	e_5
Lista PRM	=	e_0	e_1	e_2	e_3	e_4	e_5	e_6	e_7

Figura 3.9 Un'istantanea delle liste manipolate da MTF e PRM.

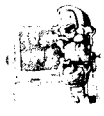
realtà quella adottata da OPT è una delle possibili permutazioni degli elementi della lista: mostriamo quindi una proprietà più generale. Presa una *qualunque* permutazione della lista iniziale, definiamo PRM come l'algoritmo che opera sulla lista permutata in analogia a quanto descritto per OPT (ovvero un elemento non viene cambiato di posizione dopo ogni accesso). I costi di PRM sono definiti analogamente a quelli di OPT per cui, quando la permutazione è quella fissata da OPT, i comportamenti di PRM e OPT coincidono. Mostrando in generale che il *costo di* MTF *non supera il doppio del costo di* PRM, otteniamo la dimostrazione anche per il caso specifico di OPT.

Formalmente, consideriamo una sequenza arbitraria di n operazioni di ricerca su una lista di m elementi, dove le operazioni sono enumerate da 0 a $n-1$ in base al loro ordine di esecuzione. Per $0 \leqslant j \leqslant n-1$, l'operazione j accede a un elemento k nelle lista come nel Codice 3.5: sia c_j la posizione di k nella lista di MTF e c'_j la posizione di k nella lista di PRM. Poiché vengono scanditi c_j elementi prima di k nella lista di MTF, e c'_j elementi prima di k nella lista di PRM, definiamo il costo delle n operazioni, rispettivamente,

$$\text{costo(MTF)} = \sum_{j=0}^{n-1} c_j \qquad e \qquad \text{costo(PRM)} = \sum_{j=0}^{n-1} c'_j \tag{3.3}$$

Vogliamo mostrare che $\text{costo(MTF)} \leqslant 2 \times \text{costo(PRM)} + O(m^2)$ per *ogni* permutazione iniziale della lista di m elementi, ovvero che

$$\sum_{j=0}^{n-1} c_j \leqslant 2 \sum_{j=0}^{n-1} c'_j + O(m^2) \tag{3.4}$$



Da tale diseguaglianza segue che MTF scandisce asintoticamente non più del doppio degli elementi scanditi da OPT quando $n \gg m^2$ (scegliendo nell'analisi la specifica permutazione, per noi imprevedibile, realizzata da OPT). Nel seguito proviamo una condizione più forte di quella espressa nella diseguaglianza (3.4) da cui possiamo facilmente derivare quest'ultima: a tal fine, introduciamo la nozione di **inversione**. Supponiamo di aver appena eseguito l'operazione j che accede all'elemento k, e consideriamo le risultanti liste di MTF e PRM: un esempio di configurazione delle due liste in un certo istante è quello riportato nella Figura 3.9.

Presi due elementi distinti x e y in una delle due liste, questi devono occorrere anche nell'altra: diciamo che l'insieme $\{x, y\}$ è un'inversione quando l'ordine relativo di occorrenza è diverso nelle due liste, ovvero quando x occorre prima di y (non necessariamente in posizioni adiacenti) in una lista mentre y occorre prima di x nell'altra lista. Nel nostro esempio, $\{e_0, e_2\}$ è un'inversione, mentre $\{e_1, e_7\}$ non lo è. Definiamo con Φ_j il numero di inversioni tra le due liste dopo che è stata eseguita l'operazione j: vale $0 \leqslant \Phi_j \leqslant \frac{m(m-1)}{2}$, per $0 \leqslant j \leqslant n-1$, in quanto $\Phi_j = 0$ se le due liste sono uguali mentre, se sono una in ordine inverso rispetto all'altra, ognuno degli $\binom{m}{2}$ insiemi di due elementi è un'inversione. Per dimostrare la (3.4), non possiamo utilizzare direttamente la proprietà che $c_j \leqslant 2c'_j + O(1)$, in quanto questa proprietà in generale non è vera. Invece, ammortizziamo il costo usando il numero di inversioni Φ_j, in modo da dimostrare la seguente relazione (introducendo un valore fittizio Φ_{-1} che specifichiamo in seguito):

$$c_j + \Phi_j - \Phi_{j-1} \leqslant 2c'_j \tag{3.5}$$

Possiamo derivare la (3.4) dalla (3.5) in quanto quest'ultima implica che

$$\sum_{j=0}^{n-1} (c_j + \Phi_j - \Phi_{j-1}) \leqslant 2 \sum_{j=0}^{n-1} c'_j$$

I termini Φ nella sommatoria alla sinistra della precedente diseguaglianza formano una cosiddetta **somma telescopica**, $(\Phi_0 - \Phi_{-1}) + (\Phi_1 - \Phi_0) + (\Phi_2 - \Phi_1) + \cdots + (\Phi_{n-1} - \Phi_{n-2}) = \Phi_{n-1} - \Phi_{-1}$, nella quale le coppie di termini di segno opposto si elidono algebricamente: da questa osservazione segue immediatamente che

$$\Phi_{n-1} - \Phi_{-1} + \sum_{j=0}^{n-1} c_j \leqslant 2 \sum_{j=0}^{n-1} c'_j \tag{3.6}$$

Ponendo $\Phi_{-1} = \frac{m(m+1)}{2}$, vale $\Phi_{-1} - \Phi_{n-1} = O(m^2)$: portando a destra del segno di diseguaglianza i primi due addendi nella (3.6), otteniamo così la disuguaglianza (3.4).

Possiamo quindi concentrarci sulla dimostrazione dell'equazione (3.5), dove il caso $j = 0$ vale per sostituzione del valore fissato per Φ_{-1}, in quanto $\Phi_0 \leqslant \frac{m(m-1)}{2}$ e $c_0 \leqslant m$. Ipotizziamo quindi che l'operazione $j > 0$ sia stata eseguita: a tale scopo, sia k l'elemento acceduto in seguito a tale operazione, e supponiamo che k occupi la posizione i nella lista di MTF (per cui $c_j = i$): notiamo che la (3.5) è banalmente soddisfatta quando $i = 0$ perché la lista di MTF non cambia e, quindi, $\Phi_j = \Phi_{j-1}$. Prendiamo l'elemento k' che appare in una generica posizione $i' < i$. Ci sono solo due possibilità se esaminiamo l'insieme $\{k', k\}$: è un'inversione oppure non lo è. Quando MTF pone k in cima alla lista, tale insieme diventa un'inversione se e solo se non lo era prima: nel nostro esempio se $k = e_3$ (per cui $i = 5$), possiamo riscontrare che, considerando gli elementi k' i

posizione da 0 a 4, due di essi, e_4 e e_6, formano (assieme a e_3) un'inversione mentre i rimanenti tre elementi non danno luogo a inversioni. Quando e_3 viene posto in cima alla lista di MTF, abbiamo che gli insiemi $\{e_3, e_4\}$ e $\{e_3, e_6\}$ non sono più inversioni, mentre lo diventano gli insiemi $\{e_2, e_3\}$, $\{e_0, e_3\}$ e $\{e_1, e_3\}$.

In generale, gli i elementi che precedono k nella lista di MTF sono composti da f elementi che (assieme a k) *danno* luogo a inversioni e da g elementi che *non danno* luogo a inversioni, dove $f + g = i$. Dopo che MTF pone k in cima alla sua lista, il numero di inversioni che cambiano sono esclusivamente quelle che coinvolgono k. In particolare, le f inversioni *non* sono più tali mentre appaiono g *nuove* inversioni, come illustrato nel nostro esempio. Di conseguenza la differenza nel numero di inversioni dopo l'operazione j è $\Phi_j - \Phi_{j-1} = -f + g$. Ne deriva che $c_j + \Phi_j - \Phi_{j-1} = i - f + g = (f + g) - f + g = 2g$.

Consideriamo ora la posizione c'_j dell'elemento k nella lista di PRM: sappiamo certamente che $c'_j \geq g$ perché ci sono almeno g elementi che precedono k, in quanto appaiono prima di k anche nella lista di MTF e non formano con k inversioni prima dell'operazione j. A questo punto, otteniamo l'equazione (3.5), in quanto $c_j + \Phi_j - \Phi_{j-1} = 2g \leqslant 2c'_j$, concludendo di fatto l'analisi ammortizzata.

ALVIE: liste ad auto-organizzazione

Osserva, sperimenta e verifica
`MoveToFront`

Osserviamo che tale analisi della strategia MTF sfrutta la condizione che l'algoritmo PRM non può manipolare la lista una volta che abbia iniziato a gestire le richieste. Purtroppo questa condizione è necessaria e la precedente analisi ammortizzata non è più valida se permettiamo anche all'algoritmo fuori linea di manipolare la sua lista: accedendo all'elemento in posizione i, l'algoritmo può ad esempio riorganizzare la lista in tempo $O(i)$ (pensiamo a un impiegato con la sua pila disordinata di pratiche: pescata la pratica in posizione i, può metterla in cima alla pila ribaltando l'ordine delle prime i nel contempo). In particolare, è possibile dimostrare che un algoritmo fuori linea che adotta tale strategia, denominato REV, ha un costo pari a $O(n \log n)$ mentre il costo di MTF risulta essere $\Theta(n^2)$, invalidando l'equazione (3.4) per n sufficientemente grande.

Tuttavia, MTF rimane una strategia vincente per organizzare le proprie informazioni. L'economista giapponese Noguchi Yukio ha scritto diversi libri di successo sull'organizzazione aziendale e, tra i metodi per l'archiviazione cartacea, ne suggerisce uno particolarmente efficace. Il metodo consiste nel mettere l'oggetto dell'archiviazione (un articolo, il passaporto, le schede telefoniche e così via) in una busta di carta etichettata. Le buste

sono mantenute in un ripiano lungo lo scaffale e le nuove buste vengono aggiunte in cima. Quando una busta viene presa in una qualche posizione del ripiano, identificata scandendolo dalla cima, viene successivamente riposta in cima dopo l'uso. Nel momento in cui il ripiano è pieno, un certo quantitivo di buste nel fondo viene trasferito in un'opportuna sede, per esempio una scatola di cartone etichettata in modo da identificarne il contenuto. Noguchi sostiene che è più facile ricordare l'ordine temporale dell'uso degli oggetti archiviati piuttosto che la loro classificazione in base al contenuto, per cui il metodo proposto permette di recuperare velocemente tali oggetti dallo scaffale. Possiamo facilmente riconoscere la strategia MTF nell'ordine ottenuto dal metodo di Noguchi, in base alla frequenza d'uso.

In conclusione, le liste ad auto-organizzazione presentano una serie di vantaggi, in quanto hanno buone prestazioni sotto certe condizioni, sono adattive rispetto alla distribuzione delle richieste, possiedono semplici algoritmi di manipolazione e, infine, non necessitano di informazioni ausiliarie per la gestione (a parte i puntatori di lista). Come ogni altra struttura, tuttavia, presentano anche alcuni svantaggi, poiché il costo della singola operazione al caso pessimo può essere lineare e, inoltre, ogni ricerca comporta comunque una ristrutturazione della lista.

3.4.3 Tecniche di analisi ammortizzata

Le operazioni di unione e appartenenza su liste disgiunte e quelle di ricerca in liste ad auto-organizzazione non sono i primi due esempi di algoritmi in cui abbiamo applicato l'analisi ammortizzata. Abbiamo già incontrato implicitamente un terzo esempio di tale analisi per valutare il costo delle operazioni di ridimensionamento di un array di lunghezza variabile (Paragrafo 2.1.3). Questi tre esempi illustrano tre diffuse modalità di analisi ammortizzata di cui diamo una descrizione utilizzando come motivo conduttore il problema dell'incremento di un contatore.

In tale problema, abbiamo un contatore binario di k cifre binarie, memorizzate in un array `contatore` di dimensione k i cui elementi valgono 0 oppure 1. In particolare, il valore del contatore è dato da $\sum_{i=0}^{k-1}$(`contatore`[i] $\times 2^i$) e supponiamo che esso contenga tutti 0 inizialmente.

Come mostrato nel Codice 3.6, l'operazione di incremento richiede un costo in tempo pari al *numero di elementi cambiati* in `contatore` (righe 4 e 7), e quindi O(k) tempo al caso pessimo: discutiamo tre modi di analisi per dimostrare che il costo ammortizzato di una sequenza di $n = 2^k$ incrementi è soltanto O(1) per incremento.

Il primo metodo è quello di **aggregazione**: conteggiamo il numero totale T(n) di passi elementari eseguiti e lo dividiamo per il numero n di operazioni effettuate. Nel nostro caso, conteggiamo il numero di elementi cambiati in `contatore` (righe 4 e 7), supponendo che quest'ultimo assuma il valore iniziale pari a zero. Effettuando n incrementi, osserviamo che l'elemento `contatore`[0] cambia (da 0 a 1 o viceversa) a ogni incre-