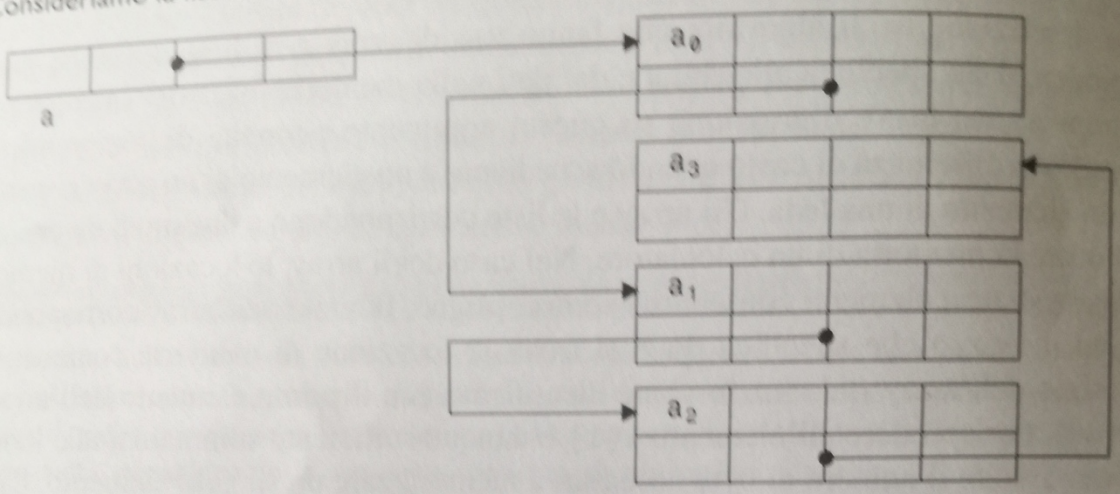
che precede... elemento di una lista ha un costo proporzionale ... (in generale, se partiamo da a_i, l'accesso ad a_{i+k} richiede $O(k)$ passi).

ESEMPIO 1.3

Consideriamo la lista a di 4 elementi di 4 byte ciascuno mostrata nella seguente figura.



Per accedere al terzo elemento, ovvero ad a_2, è necessario partire dall'inizio della lista, ovvero da a, per accedere ad a_0, poi ad a_1 e quindi ad a_2.

Le liste, d'altra parte, ben si prestano a implementare sequenze dinamiche: ciò a differenza degli array per i quali, come vedremo nel prossimo paragrafo, è necessario adottare particolari accorgimenti.

1.1.3 Array di dimensione variabile

Volendo utilizzare un array per realizzare una sequenza lineare **dinamica**, è necessario apportare diverse modifiche che consentano di effettuare il suo ridimensionamento: diversi linguaggi moderni, come C++, C# e JAVA usano tecniche simili per fornire array la cui dimensione può variare dinamicamente con il tempo. Prenderemo in considerazione l'inserimento e la cancellazione in fondo a un array a di n elementi per illustrarne la gestione del ridimensionamento. Allocare un nuovo array (più grande o più piccolo) per copiarvi gli elementi di a a ogni variazione della sua dimensione può richiedere $O(n)$ tempo per ciascuna operazione, risultando particolarmente oneroso in termini computazionali, sebbene sia ottimale in termini di memoria allocata. Con qualche piccolo accorgimento, tuttavia, possiamo fare meglio pagando tempo $O(n)$ cumulativamente per ciascun gruppo di $\Omega(n)$ operazioni consecutive, ovvero un costo medio costante per operazione. Inoltre possiamo garantire che il numero di celle allocate sia sempre proporzionale al numero degli elementi contenuti nell'array.

Sia d la taglia dell'array a ovvero il numero di elementi correntemente allocati in memoria e $n \leq d$ il numero di elementi effettivamente contenuti nella sequenza memorizzata in a. Ogni qual volta un'operazione di inserimento viene eseguita, se vi è spazio sufficiente ($n + 1 \leq d$), aumentiamo n di un'unità. Altrimenti, se $n = d$, allochiamo un array b di taglia 2d, raddoppiamo d, copiamo gli n elementi di a in b e poniamo a = b. Analogamente, ogni qualvolta un'operazione di cancellazione viene eseguita, diminuiamo n di un'unità. Quando $n = d/4$, dimezziamo l'array a: allochiamo un array b di taglia d/2, dimezziamo d, e copiamo gli n elementi di a in b (ponendo a = b). Queste operazioni di verifica ed eventuale ridimensionamento dell'array sono mostrate nel Codice 1.1.

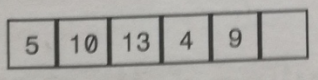
Codice 1.1 Operazioni di ridimensionamento di un array dinamico.

```
1  VerificaRaddoppio( ):        ⟨pre: a è un array di lunghezza d con n elementi⟩
2    IF (n == d) {
3      b = NuovoArray( 2 x d );
4      FOR (i = 0; i < n; i = i+1)
5        b[i] = a[i];
6      a = b;
7      d = 2 x d;
8  }
```

```
1  VerificaDimezzamento( ):     ⟨pre: a è un array di lunghezza d con n elementi⟩
2    IF ((d > 1) && (n == d/4)) {
3      b = NuovoArray( d/2 );
4      FOR (i = 0; i < n; i = i+1)
5        b[i] = a[i];
6      a = b;
7      d = d/2;
8  }
```

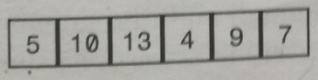
ESEMPIO 1.4

L'array a di taglia d = 6 contiene n = 5 elementi.



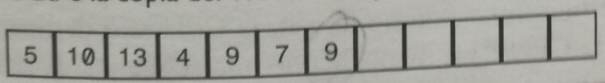
d = 6, n = 5

L'elemento 7 viene inserito in posizione d – 1 in tempo costante.



d = 6, n = 6

Poiché n = d, l'inserimento di un altro elemento richiede la creazione di un nuovo array di dimensione 2d e la copia del vecchio array nelle prime posizioni del nuovo.



d = 12, n = 7

Osserviamo che non è più possibile causare un ridimensionamento di a (raddoppio o dimezzamento) al costo di $O(n)$ tempo per ciascuna operazione: il prossimo teorema mostra che in effetti l'implementazione di tali operazioni mediante un array dinamico è molto efficiente. Ciò è dovuto al fatto che il costo di un ridimensionamento può essere concettualmente ripartito tra le operazioni che lo hanno preceduto, incrementando la loro complessità di un costo costante.

Teorema 1.1 *L'esecuzione di* n *operazioni di inserimento e cancellazione in un array dinamico richiede tempo* $O(n)$. *Inoltre* $d = O(n)$.

Dimostrazione Dopo un raddoppio, ci sono $m = d + 1$ elementi nel nuovo array di $2d$ posizioni. Occorrono almeno $m - 1$ richieste di inserimento per un nuovo raddoppio e almeno $m/2$ richieste di cancellazione per un dimezzamento. In modo simile, dopo un dimezzamento, ci sono $m = d/4$ elementi nel nuovo array di $d/2$ elementi, per cui occorrono almeno $m + 1$ richieste di inserimento per un raddoppio e almeno $m/2$ richieste di cancellazione per un nuovo dimezzamento. In tutti i casi, il costo di $O(m)$ tempo richiesto dal ridimensionamento può essere virtualmente distribuito tra le $\Omega(m)$ operazioni che lo hanno causato (a partire dal precedente ridimensionamento). Infine, supponendo che al momento della creazione dell'array si abbia $n = 1$ e $d = 1$, il numero di elementi nell'array è sempre almeno un quarto della sua taglia, pertanto $d = O(n)$. ❑

Esercizio svolto 1.1 Supponiamo che il dimezzamento di un array dinamico abbia luogo quando $n = d/2$, anziché quando $n = d/4$. Dimostrare che il Teorema 1.1 non è più vero.

Soluzione Per dimostrare che il teorema non è più verificato, definiamo una sequenza di $n = 2^k$ operazioni di inserimento e cancellazione la cui esecuzione richieda tempo $\Theta(n^2)$. Tale sequenza inizia con $n/2 = 2^{k-1}$ inserimenti: al termine della loro esecuzione l'array avrà dimensione $d = n/2$ e sarà, quindi, pieno. A questo punto la sequenza prosegue alternando un inserimento a una cancellazione: ciascuna di queste operazioni causa, rispettivamente, un raddoppio e un dimezzamento della dimensione dell'array e, quindi, richiede tempo $\Theta(n)$. In totale, quindi, l'esecuzione della seconda metà della sequenza ha un costo temporale pari a $\Theta(n^2)$.

1.2 Opus libri: scheduling della CPU

I moderni sistemi operativi sono ambienti di multi-programmazione, ovvero con-