

Algoritmi e strutture dati

Liste, pile e code



Fino a questo momento abbiamo avuto la possibilità di studiare algoritmi importanti senza avere bisogno di introdurre strutture dati concrete diverse dagli array, che conosceiamo già. Ma, in generale, cosa possiamo dire delle strutture dati? Quali caratteristiche ci interessano?

La **tassonomia** classica delle strutture dati prevede tre dimensioni ortogonali tra loro. Una struttura dati può essere **statica o dinamica**: intendiamo, con struttura dinamica, una struttura che è pensata per aggiungere e togliere elementi durante l'esecuzione di un algoritmo (esempio: l'array, come noi lo abbiamo usato, è una struttura statica); **compatta o sparsa**: tipicamente le strutture dinamiche sono sparse, cioè non possiamo fare ipotesi sulla posizione fisica degli elementi in memoria (esempio: gli array sono una struttura compatta, perchè indipendentemente da dove è memorizzato, possiamo assumere che sia fatto in maniera da avere n posizioni fisiche vicine tra loro); **basata o non basata sull'ordinamento** delle chiavi: se gli elementi sono disposti in maniera dipendente dal valore delle chiavi, allora sono basate sull'ordinamento, altrimenti no.

Un altro elemento fondamentale, dal punto di vista del trattamento teorico, è la differenza tra **chiave** e **dato satellite**. Nelle applicazioni che abbiamo visto, per esempio, ci siamo concentrati su ordinare chiavi, senza chiederci cosa è ad esse associato. Ciò che associamo ad una chiave si chiama appunto **dato satellite**, e normalmente questo è il contenuto informativo della chiave. Quando effettuiamo un movimento di chiave, implicitamente muoviamo anche i dati satellite, la cui dimensione è considerata costante.

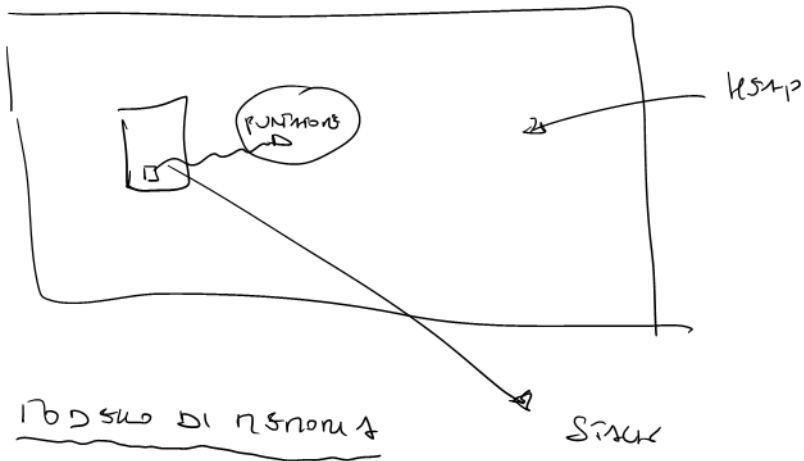
Array e liste, assieme ad altre che vedremo piú avanti, sono considerate strutture dati **concrete** (o **fondamentali**). Su di esse, a volte, è conveniente costruire strutture dati **astratte**, che nascondono l'implementazione soggiacente. Questo è ciò che viene fatto ad esempio nelle librerie offerte dai linguaggi di programmazione (si veda C++, per un esempio chiaro). La distinzione è a volte non completamente netta (come nel caso delle liste).

In tutto il resto di questo corso, tratteremo vari tipi di strutture dati, concrete ed astratte, ed algoritmi ad esse associati. Queste permettono di modellare un gran numero di problemi diversi, e gli algoritmi ad esse associati un igual numero di problemi tipici. In questo senso, possiamo pensare agli array come una struttura dati statica (quindi senza algoritmi di inserimento o cancellazione) associata al problema dell'ordinamento (che abbiamo risolto con diversi algoritmi). Quando studiamo strutture dati basate sull'ordinamento delle chiavi, potremmo, in linea teorica, associare anche ad esse il problema dell'ordinamento: normalmente questo non si fa, perchè si presta attenzione ad altri problemi (e perchè il problema dell'ordinamento lo abbiamo già risolto nel miglior modo possibile).

Puntatori e liste

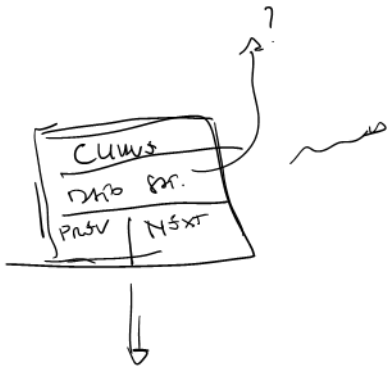
Una **lista concatenata** è una struttura dati dinamica, non basata sull'ordinamento, e sparsa. Ad essa, associamo le operazioni di inserimento, cancellazione, e ricerca. In certo modo, una lista è la versione dinamica di un array; ciò nonostante, l'operazione di ordinamento delle chiavi normalmente avviene attraverso la copiatura degli elementi su un oggetto di tipo array, e non direttamente. Le liste, in questa versione e le sue varianti, sono state introdotte da Newell, Shaw, e Simon, verso il 1955.





Un **puntatore** è un tipo di dato fondamentale e supportato da quasi tutti i linguaggi di programmazione. Quei linguaggi che non lo supportano offrono comunque le strutture dati principali in termini di oggetti e metodi. Nonostante ciò per un corretto uso di queste è necessario conoscere la loro struttura e funzionamento. Quindi immaginiamo che il nostro linguaggio offra un tipo puntatore e vediamo come funziona. Un puntatore è una **variable che contiene un indirizzo in memoria**. Viene associato al **tipo di dato puntato**, per cui un puntatore ad un intero è in generale diverso da un puntatore ad un tipo complesso, per esempio.

Nel caso di liste concatenate, immaginiamo un tipo di dato **elemento** (che contiene almeno la chiave ed un puntatore al contenuto) e due puntatori ad elemento, che chiamiamo **predecessore** e **successore**. Quindi stiamo dicendo che un elemento è di per sé un tipo di dato ricorsivo e come tale va dichiarato. Nella realtà dei linguaggi di programmazione i puntatori vanno creati, dando ordine al sistema operativo di allocare sufficiente spazio in memoria; nel nostro trattamento teorico non ci preoccupiamo di questi dettagli tecnici, che variano da linguaggio a linguaggio. Ai fini didattici, creiamo adesso una variabile L (un oggetto di tipo lista) come una struttura che contiene, almeno, un attributo $L.head$, di tipo puntatore ad elemento, che punta alla testa dell'oggetto. Si può pensare che contenga anche attributi tipo $L.numel$ che indica il numero di elementi presenti in ogni momento. All'inizio, $L.head = nil$ e $L.numel = 0$.



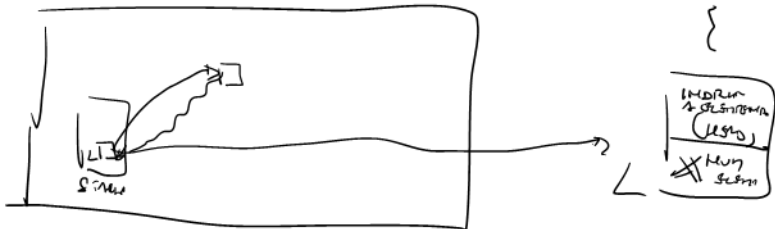
struct tlemb {
int char;
char tlemb *
next

struct Link {

struct tlemb * head

...

{

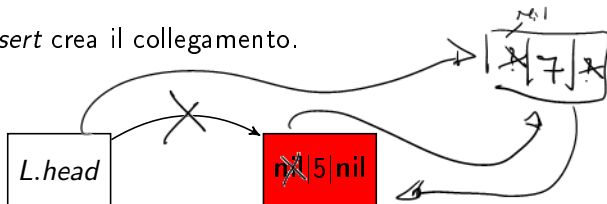


Puntatori e lista

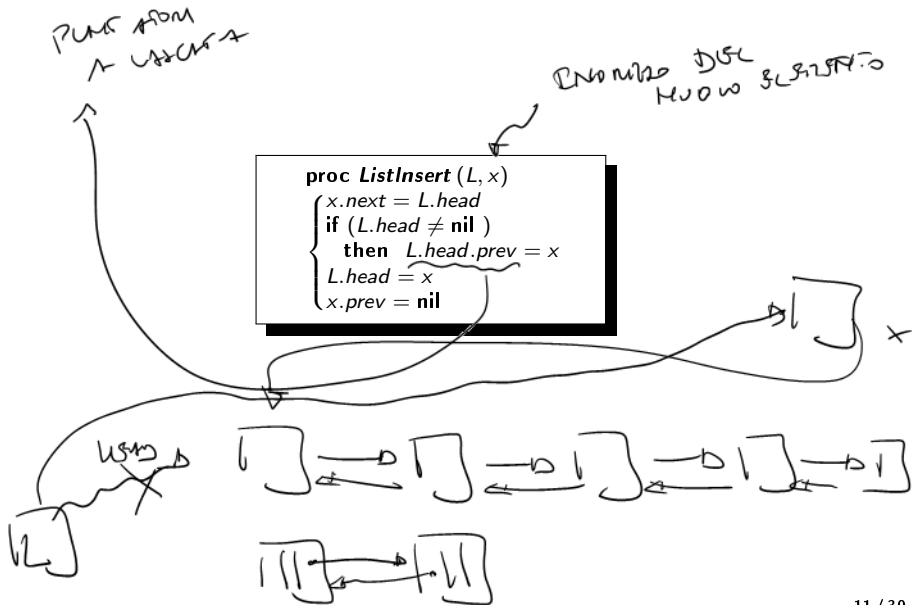
Concentriamoci nel caso di liste **non ordinate, doppiamente collegate**; quindi stabiliamo che un nodo x ha definiti la chiave ($x.key$), e i puntatori al prossimo ($x.next$) al precedente ($x.prev$) elemento. All'inizio, la lista è vuota. Immaginiamo che x sia un nuovo elemento, già allocato, per esempio con chiave 5.



L'operazione di *ListInsert* crea il collegamento.




Puntatori e lista



Il puntatore *L.head* punta sempre al primo elemento. Questo spiega la prima linea del codice. Se l'oggetto è vuoto, al primo inserimento *x* è sia il primo che l'ultimo elemento, quindi il suo prossimo (*next*) è vuoto. Se d'altra parte non è vuoto, allora l'elemento che segue *x* è quello che era il primo. Poiché si tratta di una lista doppiamente collegata (gli elementi puntano al loro predecessore), se non è vuoto (= se *L.head* non è **nil**) allora il predecessore di quello che prima era il primo elemento (*L.head.prev* - **attenzione**: i puntatori sono in cascata, e se un elemento puntato è non nullo, allora si può accedere a qualsiasi dei suoi puntatori) deve diventare *x*. Se invece era vuoto, questo puntatore non si modifica e rimane **nil**; questo spiega la seconda linea del codice. La terza linea è ovvia: posiziona il primo elemento (*x*). La quarta linea è altrettanto ovvia: non ci sono predecessori del primo elemento.

ultimo di corrente

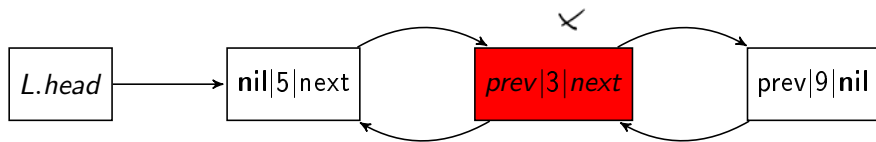


```
proc ListSearch (L, k)
{ x = L.head
  while (x ≠ nil) and (x.key ≠ k) x = x.next
  return x
```

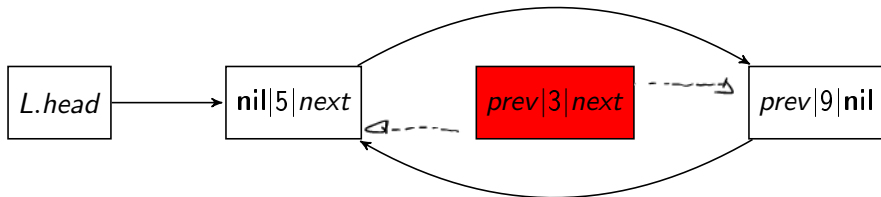
Il codice di *ListSearch* si spiega da sé.

Puntatori e lista

In quanto a *ListDelete*, abbiamo:



Eliminando *x*, dove *x* punta all'elemento che contiene la chiave 3, si ottiene:



Quindi il codice è:

```
proc ListDelete (L, x)  
  {  
    if (x.prev  $\neq$  nil )  
      then x.prev.next = x.next  
      else L.head = x.next  
    if (x.next  $\neq$  nil )  
      then x.next.prev = x.prev
```

Complessità delle operazioni su lista

La **correttezza** di queste operazioni è immediata da dimostrare, così come le loro **complessità**: l'inserimento prende $\Theta(1)$, in quanto inserisce sempre in testa all'oggetto, la cancellazione prende $\Theta(1)$ assumendo di conoscere il puntatore dell'elemento da cancellare, il quale si ottiene attraverso la ricerca che costa $\Theta(n)$ (nel caso peggiore). Anche se abbiamo detto che non ha molto senso parlare di inserimento e cancellazione in array (come struttura concreta), possiamo domandarci come, dal punto di vista della complessità, array e liste si possano confrontare.

Array vs liste: confronto

	array (c.peggiore e medio)	lista (c. peggiore e medio)
ricerca	$\Theta(n)$	$\Theta(n)$
minimo	$\Theta(n)$	$\Theta(n)$
massimo	$\Theta(n)$	$\Theta(n)$
successore	$\Theta(n)$	$\Theta(n)$
predecessore	$\Theta(n)$	$\Theta(n)$
inserimento	$\Theta(1)^*$	$\Theta(1)$
cancellazione**	$\Theta(1)$	$\Theta(1)$

*: assumendo di avere allocato abbastanza spazio, altrimenti costa $\Theta(n)$.

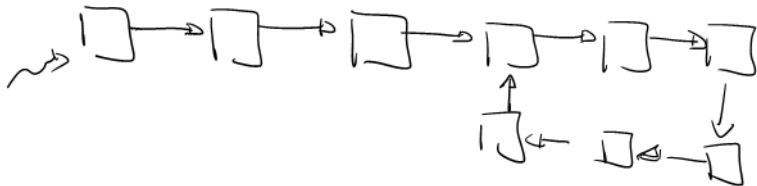
: per la cancellazione di una **chiave (senza conoscere l'indirizzo della sua posizione), va sommato il costo della ricerca.

Array e liste: confronto

Sebbene dal confronto le due soluzioni sembrano identiche, si osservi che gli array permettono l'accesso diretto, mentre le liste no. Questa differenza gioca un ruolo fondamentale quando le strutture dati di cui abbiamo bisogno sono indicizzabili, cioè quando la posizione di un elemento ed il suo contenuto sono confrontabili. Per esempio, un insieme di nomi è naturalmente memorizzato in un oggetto di tipo lista, ed un nome non è visto come un indice. Invece, un insieme (abbastanza piccolo) di numeri interi positivi può trovare posto in un array (come abbiamo fatto finora), e il contenuto di una posizione può essere anche visto come un indice.

esercizio 155:

lista
155



Soluzione 1. Una richiesta di conoscere il \star di partenza:

- Entriamo nel ciclo
- Sono tutte le hit finché non hit o la cella è vuota

$\mathcal{O}(n)$

Soluții [2]

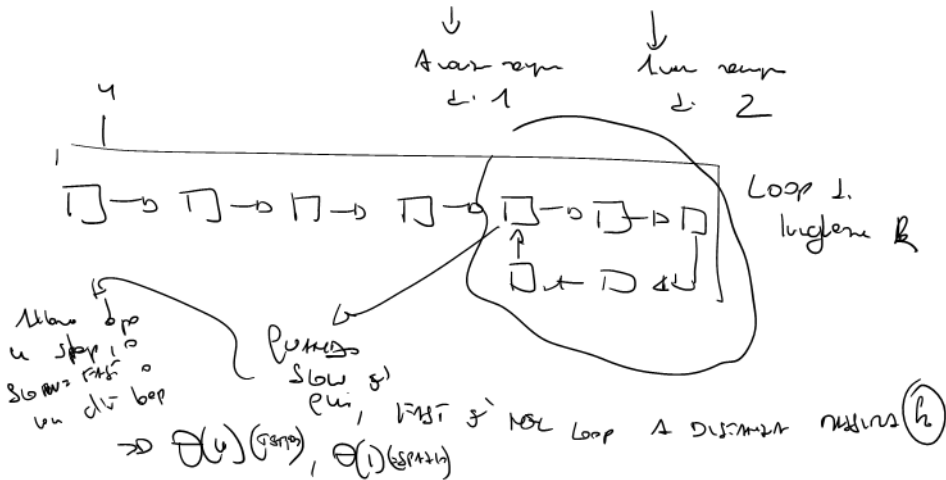
- Una metoda de aplicare
- Se cere la lista un anumit număr de indici de la început
- Se cere a MIL \Rightarrow NO LOOP
- " " a indici până la sfârșit \Rightarrow LOOP

Amplasarea de la început de la sfârșit se află în $\Theta(n)$

1. Metoda de aplicare în loc $(\text{spatiu } \Theta(n))$

Solution 13 :

- Utilize two pointers per given list
- Choose Slow & Fast



Le pile e le code sono entrambe due strutture dati astratte. Nella nostra tassonomia sono da ritenersi dinamiche e non basate sull'ordinamento. Possono essere entrambe implementate in maniera compatta (basate su array), o sparsa (basate su liste). La loro caratteristica distintiva è che l'accesso agli elementi non è libero, ma vincolato ad una **politica**. La ragione per cui può essere utile avere una politica di accesso è che questa può fare risparmiare dei dettagli implementativi, assicurando un certo ordine di inserimento ed estrazione.

Una **pila** (o **stack**) è una struttura dati astratta che implementa la politica **last in first out (LIFO)**. Una pila si utilizza in diversi contesti, tra cui valutazione di espressioni, processi di backtracking (per ricordarci le mosse che abbiamo fatto e l'ultimo punto di scelta), eliminazione della ricorsione, e molte altre. Il concetto stesso di pila è stato introdotto da Friedrich Bauer, attorno all'anno 1963, sebbene fosse già stato nominato da Turing molto prima.



Per implementare una pila ci possiamo basare su un array; avremo quindi una implementazione compatta. L'idea è quella di **mascherare** la struttura portante all'utente finale. Si può pensare come un **oggetto** con i suoi relativi **metodi**. Assumeremo quindi che S sia un array di interi, che interpretiamo come uno stack, e che viene dotato con i parametri (naturali) $S.top$ (inizializzato a 0) e $S.max$ (che indica la massima capacità di S). Le operazioni di inserimento ed eliminazione di un elemento nelle pile prendono il nome di *Push* e *Pop*, rispettivamente. La variabile S è un array dotato di struttura.

Pile su array

```
proc Empty (S)  
  { if (S.top = 0)  
    then return true  
    return false
```

```
proc Push (S, x)  
  { if (S.top = S.max)  
    then return "overflow"  
    S.top = S.top + 1  
    S[S.top] = x
```

```
proc Pop (S)  
  { if (Empty(S))  
    then return "underflow"  
    S.top = S.top - 1  
    return S[S.top + 1]
```

La **correttezza**, **complessità** e **terminazione** di queste operazioni sono assolutamente immediate.

Una **coda** (o **queue**) è una struttura dati elementare che implementa una politica **first in first out (FIFO)**. Permette di accedere ai dati attraverso inserimento ed eliminazione che prendono normalmente i nomi di *Enqueue* e *Dequeue*, rispettivamente. Anche una coda può avere diversi usi: in una **playlist**, ad esempio, le canzoni si trovano in una coda (circolare), in maniera che la canzone appena ascoltata si ripeterà **il più tardi possibile**; oppure nei processi di visita di strutture dati più complesse come i grafi, che vedremo. Non è noto un inventore per questa struttura dati; si tratta probabilmente di un concetto che è emerso in modo naturale.

Di nuovo, per implementare uno stack ci possiamo basare su un array. Assumeremo quindi che Q sia una coda (di nuovo, un array dotato di struttura), con parametri $Q.head$, $Q.tail$ (naturali), la cui inizializzazione è $Q.head = Q.tail = 1$. Per assicurare di essere capaci di distinguere perfettamente le due situazioni di pila vuota e pila piena, entrambe caratterizzate da $Q.head = Q.tail$, aggiungiamo una variabile, chiamata dim , inizialmente a 0 (perchè la coda è vuota).

```
proc Enqueue (Q, x)
  { if (Q.dim = Q.length)
    then return "overflow"
    Q[Q.tail] = x
    if (Q.tail = Q.length)
    then Q.tail = 1
    else Q.tail = Q.tail + 1
    Q.dim = Q.dim + 1
```

```
proc Dequeue (Q)
  { if (Q.dim = 0)
    then return "underflow"
    x = Q[Q.head]
    if (Q.head = Q.length)
    then Q.head = 1
    else Q.head = Q.head + 1
    Q.dim = Q.dim - 1
    return x
```

Di nuovo, **correttezza**, **complessità** e **terminazione** di queste operazioni sono triviali da dimostrare.

Pile e code possono anche essere implementate in maniera sparsa, utilizzando delle liste concatenate come supporto. Per quanto riguarda le pile, l'operazione di inserimento in testa ad una lista è proprio l'operazione di *Push*, e l'operazione di *Pop* è una semplificazione dell'operazione di delete. In questo caso, *S* è semplicemente una lista, senza campi aggiuntivi.

```
proc Empty (S)  
{  
  if (S.head = nil )  
    then return true  
  return false  
}
```

```
proc Push (S, x)  
{  
  Insert(S, x)  
}
```

```
proc Pop (S)  
{  
  if (Empty(S))  
    then return "underflow"  
  x = S.head  
  Delete(S, x)  
  return x.key  
}
```

Correttezza, complessità e terminazione di queste operazioni non presentano nessun problema.

Per implementare una coda attraverso una lista dobbiamo immaginare che la lista Q sia dotata del campo $Q.tail$ in aggiunta al campo $Q.head$. In questo modo possiamo implementare *Enqueue* semplicemente chiamando l'inserimento in una lista, e *Dequeue* semplicemente chiamando l'eliminazione di un elemento in una lista.

```
proc Empty (Q)  
{ if (Q.head = nil )  
  then return true  
  return false
```

```
proc Enqueue (Q, x)  
{ Insert(Q, x)
```

```
proc Dequeue (Q)  
{ if (Empty(Q))  
  then return "underflow"  
  x = Q.tail  
  Q.tail = x.prev  
  Delete(Q, x)  
  return x.key
```

Correttezza, complessità e terminazione di queste operazioni sono immediate.

La lista è la più semplice delle strutture dati dinamiche, e può essere implementata con collegamento semplice, doppio, oppure in maniere molto complesse che permettono di sfruttare delle euristiche di miglioramento delle complessità.