Corso di laurea in Matematica
Insegnamento di Informatica generale
Lezioni in modalità mista o a distanza

Strutture dati fondamentali: Insiemi dinamici ed operazioni su di essi con le strutture dati semplici

Giancarlo Bongiovanni
Ivano Salvo

SAPIENZA
UNIVERSITÀ DI ROMA

Queste dispense sono state realizzate sulla base delle slide preparate da T. Calamoneri e G. Bongiovanni per il corso di Informatica Generale tenuto a distanza nell'A.A. 2019/20

Strutture dati (1)

- In un linguaggio di programmazione, un dato è un valore che una variabile può assumere.
- Un **tipo di dato astratto** è un modello matematico, dato da una **collezione di valori** e un **insieme di operazioni** ammesse su questi valori.
- Le **strutture dati** sono particolari tipi di dato, caratterizzate dall'organizzazione dei dati, più che dal tipo di dato stesso.

Strutture dati (2)

- Una struttura dati è quindi composta da:
  - un modo sistematico di organizzare i dati
  - un insieme di operatori che permettono di manipolare la struttura
- Le strutture dati possono essere:
  - lineari o non lineari (a seconda che esista o no una sequenzializzazione)
  - statiche o dinamiche (a seconda che possano variare la dimensione nel tempo)
  - omogenee o disomogenee (rispetto ai dati contenuti).

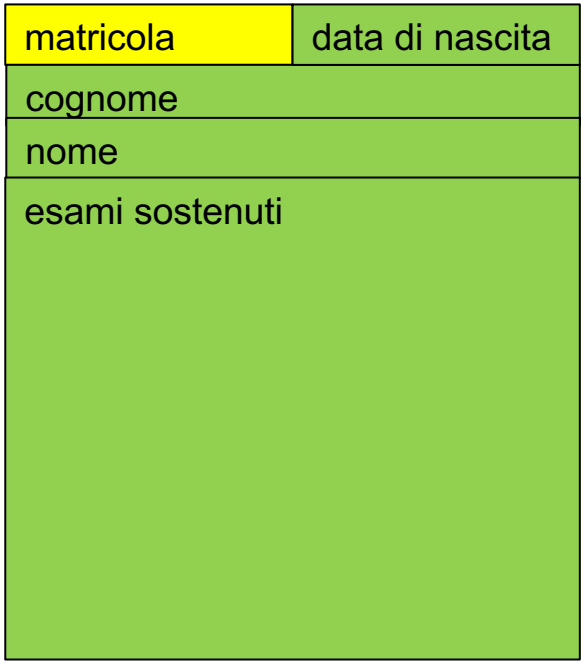
Insiemi dinamici (1)

Una struttura dati serve a memorizzare e manipolare ***insiemi dinamici***, cioè insiemi i cui elementi possono variare nel tempo in funzione delle operazioni compiute dall'algoritmo che li utilizza.

Insiemi dinamici (2)

Gli elementi dell'insieme dinamico (spesso chiamati anche ***record***) possono essere piuttosto complessi e contenere ciascuno più di un dato "elementare". In tal caso è abbastanza comune che essi contengano:

- una ***chiave***, utilizzata per distinguere un elemento da un altro nell'ambito delle operazioni di manipolazione dell'insieme dinamico; normalmente i valori delle chiavi fanno parte di un insieme totalmente ordinato (ad. es., sono numeri interi);
- ulteriori dati, detti ***dati satellite***, che sono relativi all'elemento stesso ma non sono direttamente utilizzati nelle operazioni di manipolazione dell'insieme dinamico.



Insiemi dinamici (3)

In altri casi, ogni elemento contiene solo la chiave e quindi coincide con essa.

Nel seguito ci riferiremo sempre a questa situazione semplificata, a meno che non venga esplicitamente specificato il contrario.

Questo perché la maggior parte degli algoritmi che vedremo dipendono essenzialmente solo dal valore delle chiavi.

Insiemi dinamici (4)

Le tipiche operazioni che si compiono su un insieme dinamico S (e quindi sulla struttura dati che ne permette la gestione), che si suppone totalmente ordinabile, si dividono in due categorie:

- *operazioni di interrogazione*
- *operazioni di modifica*

Insiemi dinamici (5)

Tipici esempi di operazioni di **interrogazione**, che non modificano la consistenza dell'insieme dinamico, sono:

- Search(S, k): recuperare l'elemento con chiave di valore *k,* se è presente in *S,* restituire un valore speciale nullo altrimenti;
- Min(S): recuperare il minimo valore presente in *S*;
- Max(S): recuperare il massimo valore presente in *S*;
- Predecessor(S, k): recuperare l'elemento presente in *S* che precederebbe quello di valore *k* (di cui supponiamo di conoscere la locazione) se *S* fosse ordinato;
- Successor(S, k): recuperare l'elemento presente in *S* che seguirebbe quello di valore *k* (di cui supponiamo di conoscere la locazione) se *S* fosse ordinato.

Insiemi dinamici (6)

Tipici esempi di operazioni di **manipolazione**, che invece modificano la consistenza dell'insieme dinamico, sono:

- Insert(S, k): inserire un elemento di valore *k* in *S*;
- Delete(S, k): eliminare da *S* l'elemento di valore *k* (di cui supponiamo di conoscere la locazione).

Insiemi dinamici (7)

Le differenti strutture dati che vedremo, se da un lato hanno in comune la capacità di memorizzare insiemi dinamici, dall'altro differiscono anche profondamente fra loro per le ***proprietà*** che le caratterizzano.

Sono proprio le proprietà della struttura dati ad essere l'elemento determinante nella scelta da effettuare quando si deve progettare un algoritmo per risolvere un problema.

Un esempio lo abbiamo già incontrato: la struttura dati Heap, senza la quale l'algoritmo Heapsort non potrebbe nemmeno essere pensato.

Vettori (1)

Prima di procedere allo studio di alcune interessanti strutture dati, vediamo il costo computazionale delle principali operazioni su insiemi dinamici quando questi vengano memorizzati su semplici vettori, disordinati o ordinati.

Osservazione

Il vettore è qui considerato come una **struttura statica**: con alcuni linguaggi di programmazione è possibile variarne dinamicamente la dimensione, ma ciò viene consentito solo "simulando" la struttura dati vettore con una struttura dati dinamica…

Vettori (2)

Search(S,k)

- vettore **disordinato**: bisogna scorrere il vettore: $O(n)$.
- vettore **ordinato**: ricerca binaria: $O(\log n)$

Min(S), Max(S)

- vettore **disordinato**: bisogna scorrere il vettore: $\Theta(n)$.
- vettore **ordinato**: primo o ultimo elemento: $\Theta(1)$

Predecessor(S,k), Successor(S,k)

- vettore **disordinato**: bisogna scorrere il vettore: $\Theta(n)$.
- vettore **ordinato**: elemento precedente o seguente: $\Theta(1)$

Vettori (3)

Insert(S,k)

- vettore **disordinato**: inserimento nella prima posizione libera: $\Theta(1)$.
- vettore **ordinato**: ricerca della posizione, scorrimento a destra degli elementi maggiori ed inserimento: $O(n)$

Delete(S,k)

- vettore **disordinato**: eliminazione e scambio con l'ultimo elemento: $\Theta(1)$.
- vettore **ordinato**: eliminazione e scorrimento a sinistra per coprire lo spazio lasciato: $O(n)$.

Vettori (4)

Riassumendo:

Struttura dati	Search(S,k)	Minimum(S) Maximum(S)	Predecessor(S,k) Successor(S,k)	Insert(S,k)	Delete(S,k)
Vettore qualunque	$O(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$O(1)$
Vettore ordinato	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$

Già da questo semplice confronto delle due strutture dati più semplici in assoluto, si può dedurre come non abbia senso dire che una struttura è migliore di un'altra: le **strutture dati possono essere più o meno adatte ad un certo algoritmo** e sta al buon progettista disegnare un algoritmo efficiente usando la struttura dati più adatta.

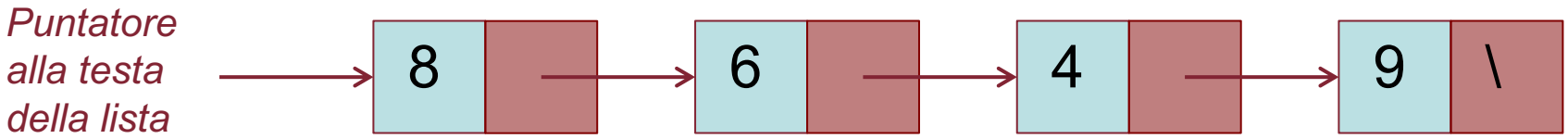
Pred e Succ sono $\theta(1)$ sui vettori ordinati se conosco l'indice. Altrimenti sono $\theta(\log n)$ perché prima devo trovare l'elemento con una ricerca.

Liste semplici (1)

La ***lista semplice*** è una struttura dati nella quale gli elementi sono organizzati in successione.

Proprietà specifiche delle liste:

- l'accesso avviene sempre ad una estremità della lista, per mezzo di un *puntatore* alla testa della lista;
- ogni elemento contiene un puntatore che consente l'accesso all'elemento successivo;
- è permesso solo un accesso sequenziale agli elementi.



Liste semplici (2)

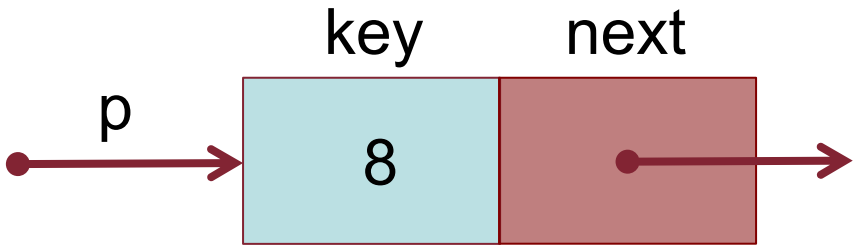
Differenze fra vettori e liste:

- *Vettore*: il numero degli elementi è prefissato e la successione degli elementi è realizzata mediante gli indici degli elementi. E' possibile l'accesso casuale.
- *Lista*: può crescere di dimensioni e la successione degli elementi viene implementata mediante un collegamento esplicito da un elemento ad un altro (di norma tramite un puntatore). E' possibile solo l'accesso sequenziale.

Liste semplici (3)

Ogni elemento di lista è un record a due campi:

- campo ***key***: contiene l'informazione vera e propria;
- campo ***next***: contiene il puntatore che consente l'accesso all'elemento successivo; nel caso dell'ultimo elemento della lista contiene un apposito valore *null* (visualizzato col simbolo \).



Nota su allocazione/deallocazione

I nodi delle liste vengono usualmente **creati in modo dinamico**, ogni qualvolta sia necessario avere memoria per un nuovo dato.

Tale memoria va **liberata** quando i dati non servono più.

Nello pseudocodice ignoreremo prevalentemente questo problema, e quando inseriremo un nuovo nodo in una lista sarà già pronto.

In linguaggio C ci sono primitive di "basso livello" (`malloc`, `calloc`, `free`) per fare queste operazioni (occorre passare la quantità di memoria desiderata).

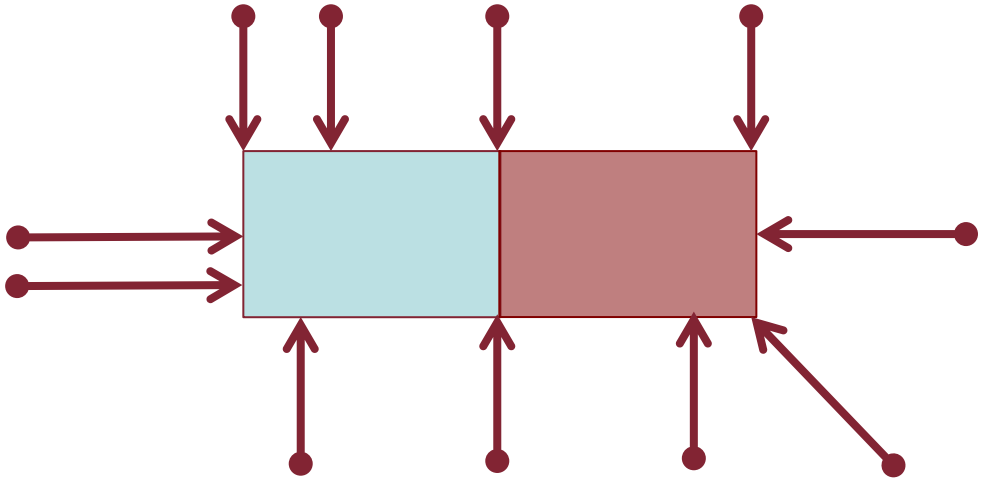
Linguaggi più moderni hanno primitive più astratte:

p = **new**(Lista) crea un nuovo nodo (che va riempito)

delete(p) elimina dalla memoria il nodo puntato da p.

Nota metodologica sui puntatori (1)

In tutta la trattazione delle strutture dinamiche, nel caso le immagini si riferiscano a record e puntatori si deve intendere che *il puntatore punta sempre all'intero record nel suo complesso*, indipendentemente dal punto specifico nel quale il puntatore tocca il record.



In altre parole, *tutti i puntatori sopra raffigurati puntano allo stesso record e nello stesso modo*.

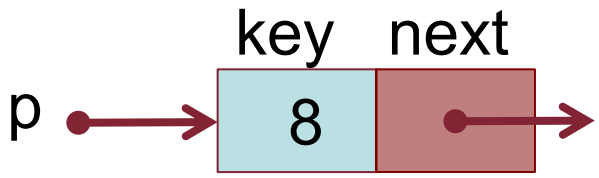
Nota metodologica sui puntatori (2)

Si deve ricordare sempre che il *valore di una variabile di tipo puntatore è un indirizzo di memoria*.

Il valore del puntatore è *l'indirizzo del primo byte* del record puntato, indipendentemente dalle dimensioni del record stesso.

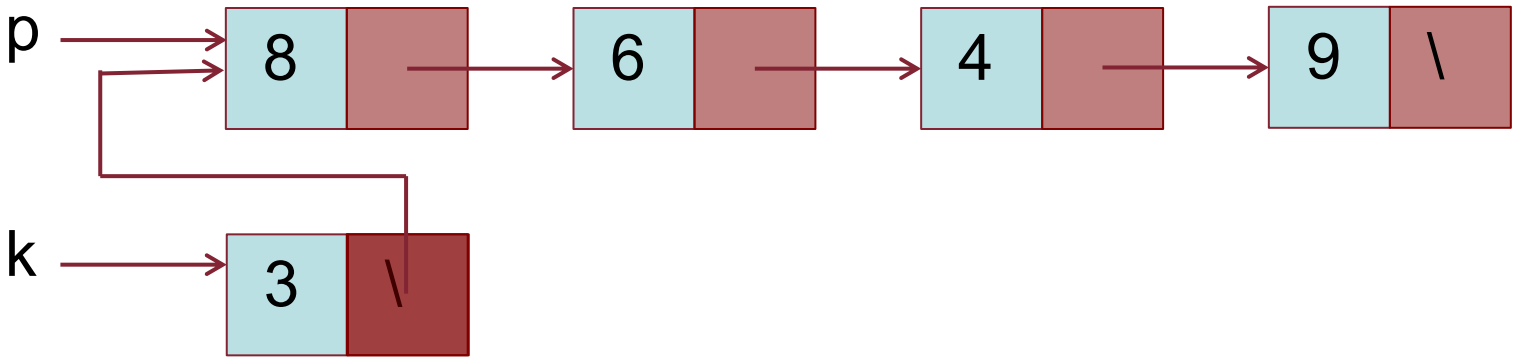
Nello pseudocodice viene utilizzata questa sintassi:

- *key[p]* indica il contenuto del campo key dell'elemento puntato da p. Corrisponde al `p->key` del C.
- *next[p]* indica il contenuto del campo next dell'elemento puntato da p, ossia l'indirizzo dell'elemento successivo (o null se esso è l'ultimo elemento); (In C: `p->next`).



Liste semplici (4) – Inserimento (in Testa)

```
Funzione Insert_in_testa (p: puntatore alla lista;
                          k: puntatore all'elemento da inserire)
   if (k ≠ NULL)
      next[k] ← p
   p ← k
   return p
```



Il costo computazionale dell'inserimento è $\Theta(1)$.

Inserimento di un valore

Se dovessi scrivere la stessa funzione, ma inserire un valore, piuttosto che un nodo già allocato in memoria, dovrei preoccuparmi di "creare" la memoria necessaria.

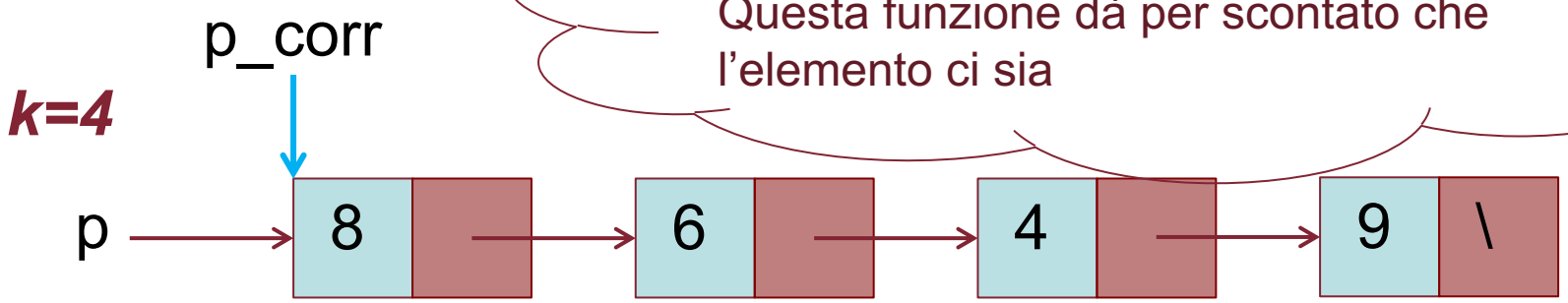
La funzione **new** ritorna un puntatore al nuovo record creato.

```
fun insert_in_Testa(lista L, valore v):
    K = new(Lista)
    key[K] = v
    next[K] = L
    return K
```

Liste semplici (5) - Ricerca

```
Funzione Search (p: puntatore alla lista; k: valore)
   p_corr = p
   while ((p_corr ≠ NULL) and (key[p_corr] ≠ k))
      p_corr ← next[p_corr]
   return p_corr
```

Nota Bene: usualmente si torna il puntatore del nodo contienente il valore Questa funzione dà per scontato che l'elemento ci sia



Il costo computazionale della ricerca è *O(n).*

Nota su ricorsione e liste

Le liste sono un tipico dato **induttivo**. Possiamo infatti definirle per induzione esattamente come i **naturali** vengono definiti induttivamente a partire da **0** e **successore**.

Dato un tipo A, le liste con informazioni di tipo A, **List(A)** sono:

- la lista vuota.
- oppure una lista che comincia con un'informazione a ∈ A seguita da una lista L ∈ **List(A)**.

Ne consegue che spesso i programmi **più naturali** sulle liste sono **ricorsivi**!

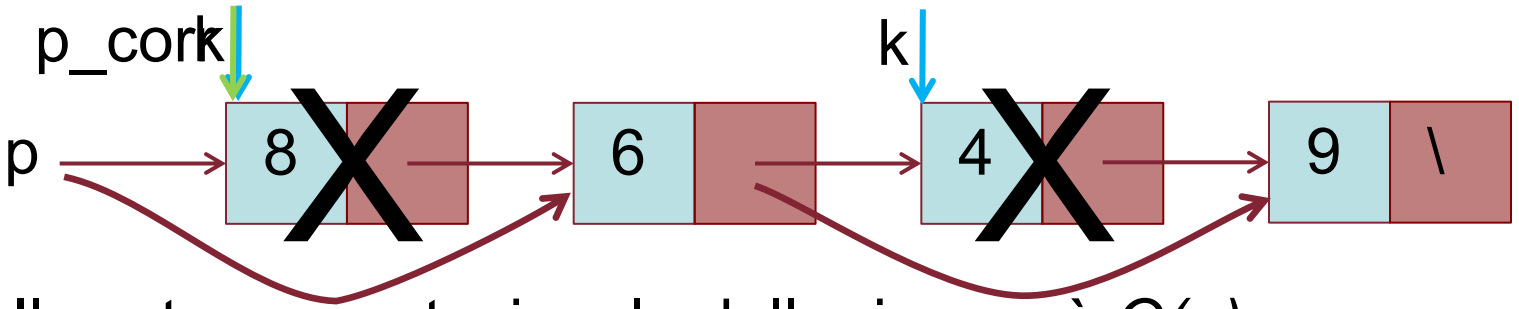
```
fun ricerca(lista L, valore k):
    if  L = NULL return NULL

    if k = key[L] return L

    return ricerca(next[L], k)
```

Liste semplici (6) - Eliminazione

```
Funzione Delete (p: punt. alla lista;
                 k: puntat. all'elem. da cancellare)
/* PREC: k appartiene a p */
   if (k ≠ NULL) // se k=NULL non c'è niente da cancellare
      if k = p                          // cancel. 1° elem
         p ← next[p]; free(k); return p
      p_corr ← p
      while (next[p_corr] ≠ k)
         p_corr ← next[p_corr] // qui, p_corr punta
                               //all'elem. che precede k
      next[p_corr] ← next[k]; free(k);
   return p
```



Il costo computazionale della ricerca è *O(n)*.

Eliminazione ricorsiva

L'usuale (come al solito non mancano eccezioni) **maggiore facilità dei programmi ricorsivi su liste** può essere spiegata con due argomenti diametralmente opposti:

- **semanticamente**, è naturale esprimere proprietà e computazioni su liste con equazioni ricorsive in accordo con la **natura induttiva** delle liste
- **operazionalmente**, una **scansione ricorsiva scorre una lista due volte**, una all'andata delle chiamate ricorsive e una al ritorno.

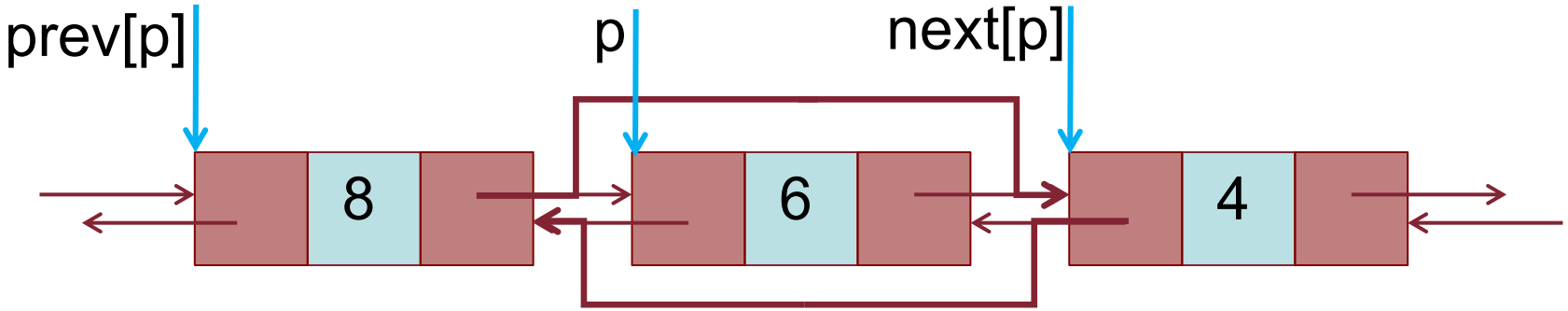
Nell'eliminazione, ad esempio, risistemo i pointer al ritorno, senza dovermi preoccupare di posizionarmi sul precedente!

```
fun elimina(lista L, lista K):
  /* PREC: K occorre in L */
  if  L = K return next[L]
  next[L] = elimina(next[L], K)
     /* next[L] cambia quando K è il prossimo elemento */
  return L
```

Liste doppiamente puntate

Alcuni problemi riscontrati nella lista semplice (ad esempio complessità lineare nella cancellazione) possono essere risolti organizzando la struttura dati in modo che da ogni suo elemento si possa accedere sia all'elemento che lo segue che a quello che lo precede nella lista, quando essi esistono. Tale struttura dati si chiama ***lista doppia*** o ***lista doppiamente concatenata*** o ***lista doppiamente puntata***:

```
…
next[prev[p]] ← next[p]
prev[next[p]] ← prev[p]
…
```



Liste

Riassumendo, per le liste semplici e doppie il costo computazionale delle varie operazioni è il seguente:

Struttura dati	Search(S,k)	Minimum(S) Maximum(S)	Predecessor(S,k)	Insert(S,k)	Delete(S,k)
Lista semplice	$O(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$O(n)$
Lista doppia	$O(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

OSS: L'operazione di delete è $\theta(1)$ nelle liste doppie a patto di avere il pointer dell'elemento da eliminare. Se devo cercare il valore da eliminare è $\theta(n)$ come per le liste semplici.

Corso di laurea in Matematica
Insegnamento di Informatica generale
Lezioni a distanza

Esercizi per casa

SAPIENZA
UNIVERSITÀ DI ROMA

Esercizi (1)

Progettare degli algoritmi che risolvano i seguenti problemi; calcolarne poi il costo computazionale:

1. data in input una lista tramite il puntatore al primo elemento, restituire il puntatore all'ultimo elemento;
2. data in input una lista tramite il puntatore al primo elemento, restituire il puntatore al penultimo elemento;
3. data in input una lista tramite il puntatore al primo elemento, restituire il puntatore alla stessa lista da cui sia stato eliminato l'ultimo elemento (***attenzione che la lista potrebbe avere un solo elemento***);
4. data in input una lista tramite il puntatore al primo elemento, restituire il puntatore di una lista che contenga gli stessi record della lista di partenza ma in ordine inverso (provare sia a creare una **nuova lista** (creando nuovi record) che a "smontare" e "rimontare" opportunamente i record iniziali). ***Verificare se sono più semplici le versioni ricorsive o quelle iterative***!

Esercizi (2)

Progettare degli algoritmi che risolvano i seguenti problemi; calcolarne poi il costo computazionale:

5. data in input una lista tramite il puntatore al primo elemento, restituire una coppia di liste, una con gli elementi di posto pari nella lista di partenza, ed una con gli elementi di posto dispari (**provare sia a creare nuovi record che solo spostare puntatori**);
6. data in input una lista di interi tramite il puntatore al primo elemento, stampare tutti i valori che compaiono almeno due volte nella lista;
7. data in input una lista ordinata di interi tramite il puntatore al primo elemento, ed un elemento da inserire, aggiungere tale elemento alla lista in modo da rispettare l'ordinamento;
8. ***Provare ad adattare gli algoritmi di ordinamento visti per i vettori. Discutere vantaggi e svantaggi. Come è preferibile fare uno scambio in una lista? E un inserimento?***

Soluz: inserimento in lista ordinata

Vediamo prima la soluzione **ricorsiva**. Lascio a voi riflettere su quale sia più semplice.

Notate la tecnica di "**riassegnare i puntatori**" al ritorno che permette di trattare in modo **uniforme ogni caso** (nodo da inserire in testa, in coda o nel mezzo della lista).

```
fun inserisciOrdRec(lista L, lista K):
  /* PREC: L ordinata */
  if  L = NULL return K
    /* sono arrivato in fondo, K evidentemente è maggiore */
  if key[K] <= key[L]
        next[K] ← L
          /* K ora è la testa della lista rimanente */
        return K
  next[L] ← inserisciOrdRec(next[L], K)
  return L;
```

inserim. ITERATIVO lista ordinata

Vediamo ora la soluzione **iterativa**. Studente avvisato…

```
fun inserisciOrdIte(lista L, lista K):
  /* PREC: L ordinata */
   if  L = NULL or key[K] <= key[L]
     /* se L è vuota, K è l'unico nodo del risultato */
      * oppure se K va inserito in testa */
      next[K] ← L
      return K
  Lcurr ← L /* mai perdere la testa… di L */
  while next[Lcurr] ≠ NULL do /* occorre fermarsi sull'ultimo */
      if  key[K] <= key[next[Lcurr]] /* occorre un pointer al prec.*/
                next[K] ← next[Lcurr] /* attenzione all'ordine */
                next[Lcurr] ← K
                return L /* la testa non cambia! */
   /* se finisco L senza inserire K, K è maggiore di tutti */
  next[Lcurr] ← K
  next[K] ← NULL
  return L;
```