Corso di laurea in Informatica
Introduzione agli Algoritmi
A.A. 2022/23

Strutture dati fondamentali: Insiemi dinamici ed operazioni su di essi

Tiziana Calamoneri

SAPIENZA
UNIVERSITÀ DI ROMA

Slides realizzate sulla base di quelle 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.

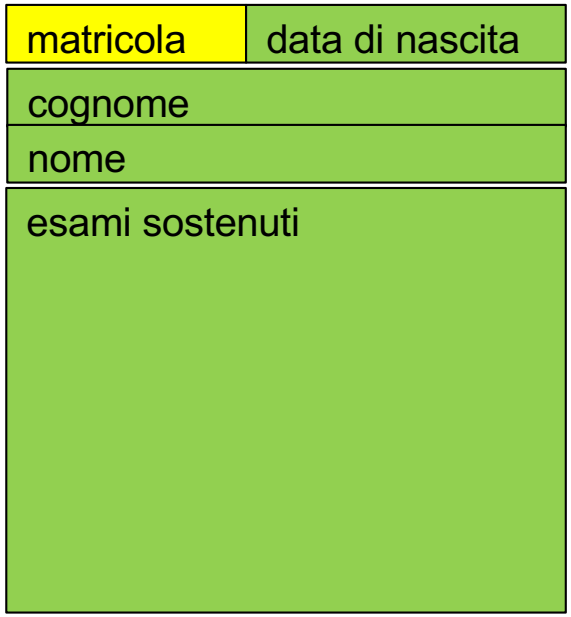
Strutture dati (2)

- 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.

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)/Max(S):

  recuperare il minimo/massimo valore presente in *S*;

- Predecessor(S, k)/Successor(S, k):

  recuperare l'elemento presente in *S* che precederebbe/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.

Arrays (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 array, disordinati o ordinati.

Osservazione

L'array è considerato come una **struttura statica**: con alcuni linguaggi di programmazione (ad esempio in Python) sembra possibile variarne dinamicamente la dimensione, ma ciò viene consentito solo "simulando" la struttura dati array con una struttura dati dinamica…

Arrays (2)

Search(S,k)

- array disordinato: bisogna scorrere l'array: $O(n)$.
- array ordinato: ricerca binaria: $O(\log n)$

Min(S), Max(S)

- array disordinato: bisogna scorrere l'array: $\Theta(n)$.
- array ordinato: primo o ultimo elemento: $\Theta(1)$

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

- array disordinato: bisogna scorrere l'array: $\Theta(n)$.
- array ordinato: elemento precedente o seguente: $\Theta(1)$

Arrays (3)

Insert(S,k)

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

Delete(S,k)

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

Arrays (4)

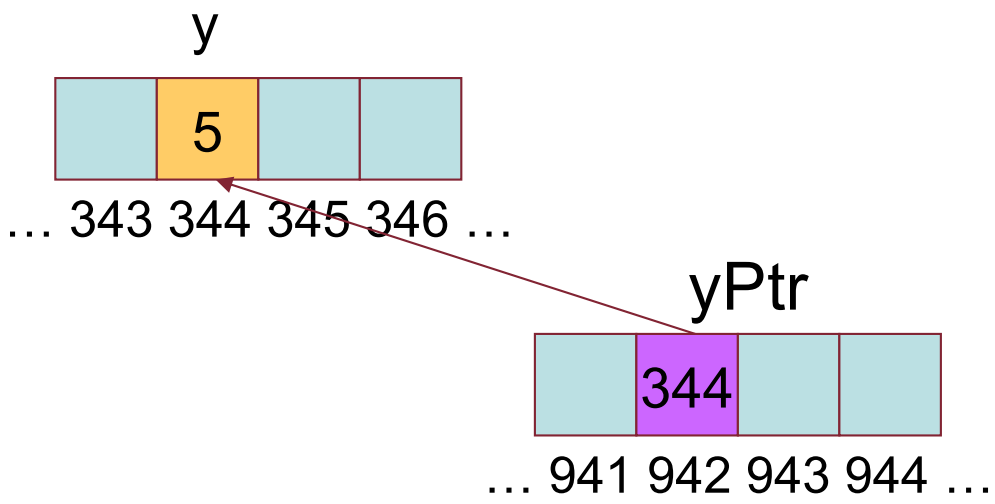
Riassumendo:

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.

Liste puntate semplici (1)

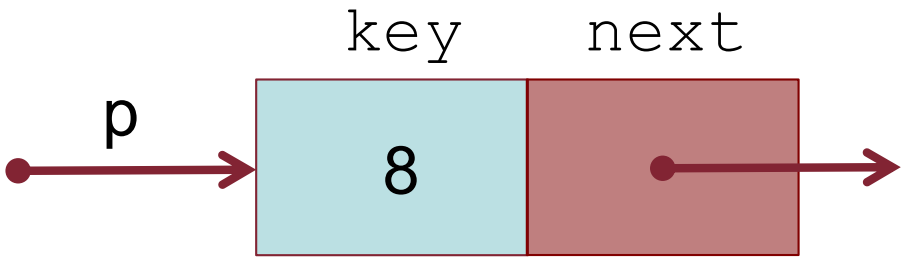
- Un puntatore è una variabile che assume come valore un indirizzo di memoria.
- Il nome di una variabile fa quindi riferimento ad un valore direttamente, mentre un puntatore lo fa indirettamente.
- Esempio:
  - `int y = 5;`
  - `int *yPtr;`
  - `yPtr = &y;`



Liste puntate semplici (2)

Ogni elemento di lista è un record a due campi:

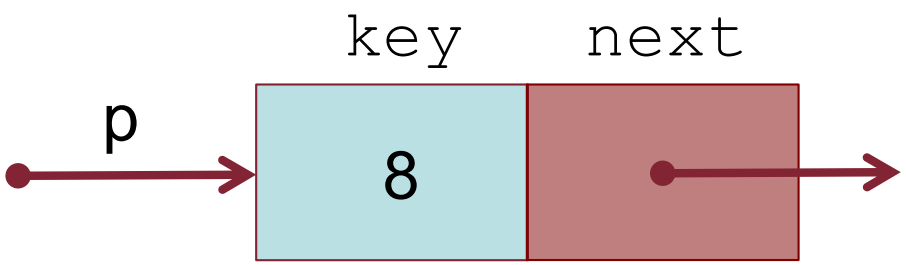
- campo ***key***: contiene l'informazione vera e propria;
- campo ***next***: contiene il puntatore che consente l'accesso all'elemento successivo; nel caso dell'ultimo elemento della lista contiene un apposito valore *null* (o *none*) (visualizzato col simbolo \).



Liste puntate semplici (3)

Nello pseudocodice viene utilizzata questa sintassi:

- `p->key` indica il contenuto del campo `key` dell'elemento puntato da p.
- `p->next` indica il contenuto del campo `next` dell'elemento puntato da p, ossia l'indirizzo dell'elemento successivo (o *None* se esso è l'ultimo elemento).

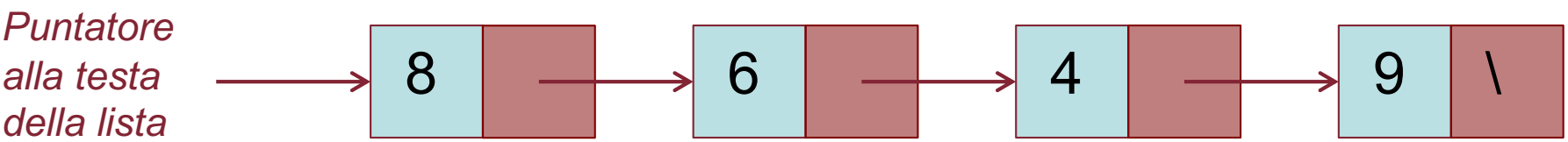


Liste puntate semplici (4)

La ***lista puntata 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.

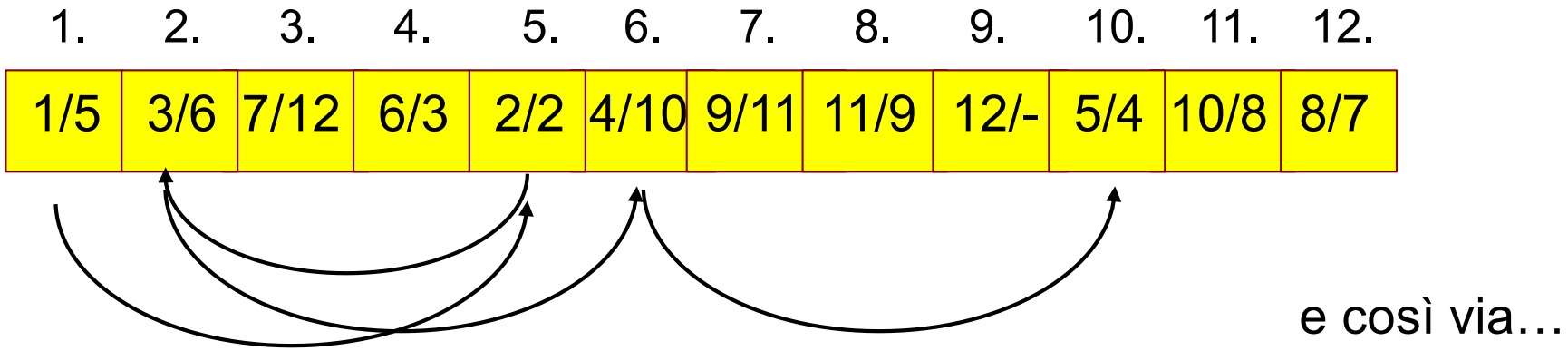


Una parentesi:
accesso diretto (casuale) ed accesso sequenziale (1)

Gli arrays, ordinati e non, godono di un **accesso diretto**: per accedere ad un dato basta conoscerne la posizione nell'array (cioè l'indice).

Segue che l'accesso a qualsiasi dato in un array ha un costo $\Theta(1)$.

Questo non è più vero nel solo specialissimo caso in cui ogni dato rappresenta l'elemento che lo segue nell'ordine:

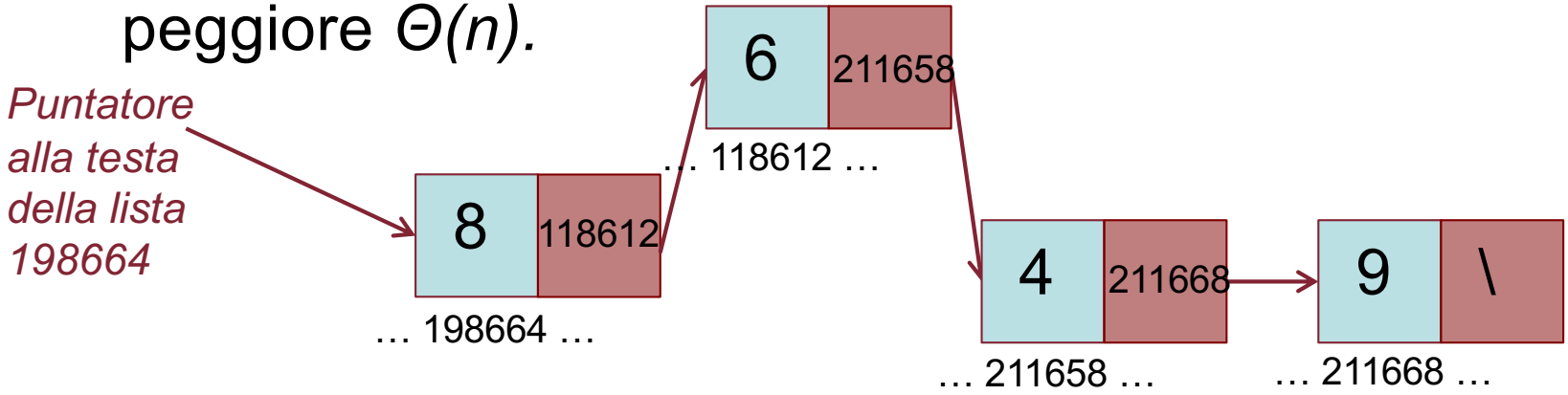


e così via…

Una parentesi:
accesso diretto (casuale) ed accesso sequenziale (2)

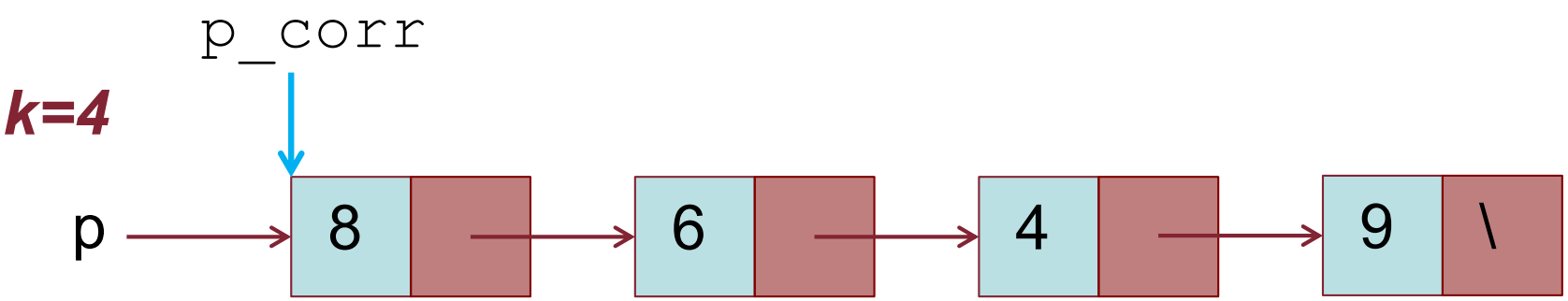
In una lista puntata, la successione degli elementi viene implementata mediante un collegamento esplicito da un elemento ad un altro (di norma tramite un puntatore). E' possibile quindi solo l'**accesso sequenziale**.

Segue che l'accesso a qualsiasi dato in una lista ha un costo proporzionale alla sua posizione, nel caso peggiore $\Theta(n)$.



Liste puntate semplici (5) - Ricerca

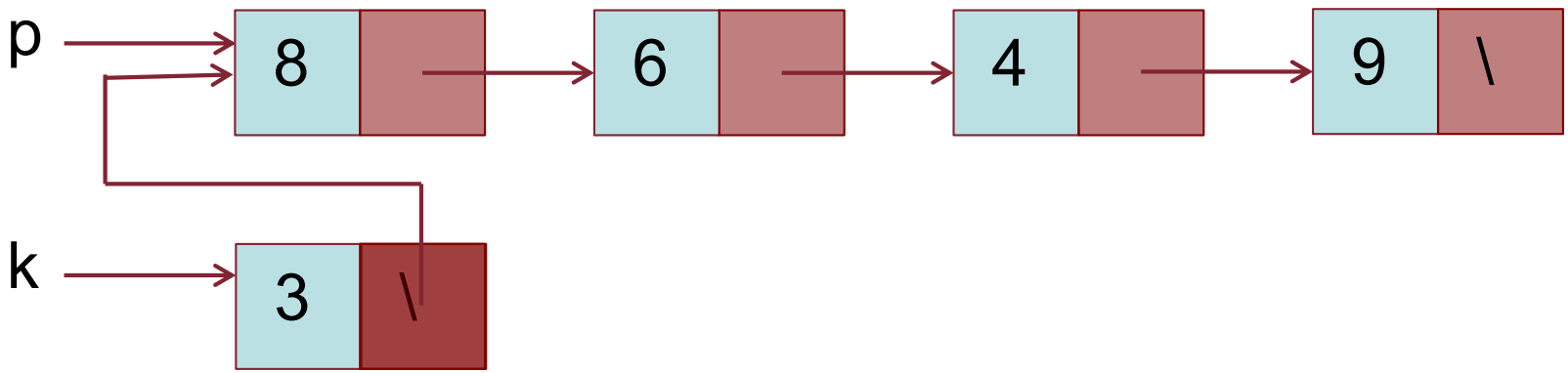
```
Funzione Search (p: puntatore alla lista; k: valore)
  p_corr = p
  while ((p_corr ≠ NULL) and (p_corr-> key ≠ k))
    p_corr = p_corr-> next
  return p_corr
```



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

Liste puntate semplici (6) - Inserimento

```
Funzione Insert_in_testa (p: puntatore alla lista;
                          k: punt. all'elem. da inserire)
   if (k ≠ None)
      k->next = p
   p = k
   return p
```



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

Liste puntate semplici (7) - Inserimento

Nella funzione `Insert_in_testa` abbiamo preso come parametro il puntatore `k` all'elemento da inserire.

In generale, questo va creato!!!

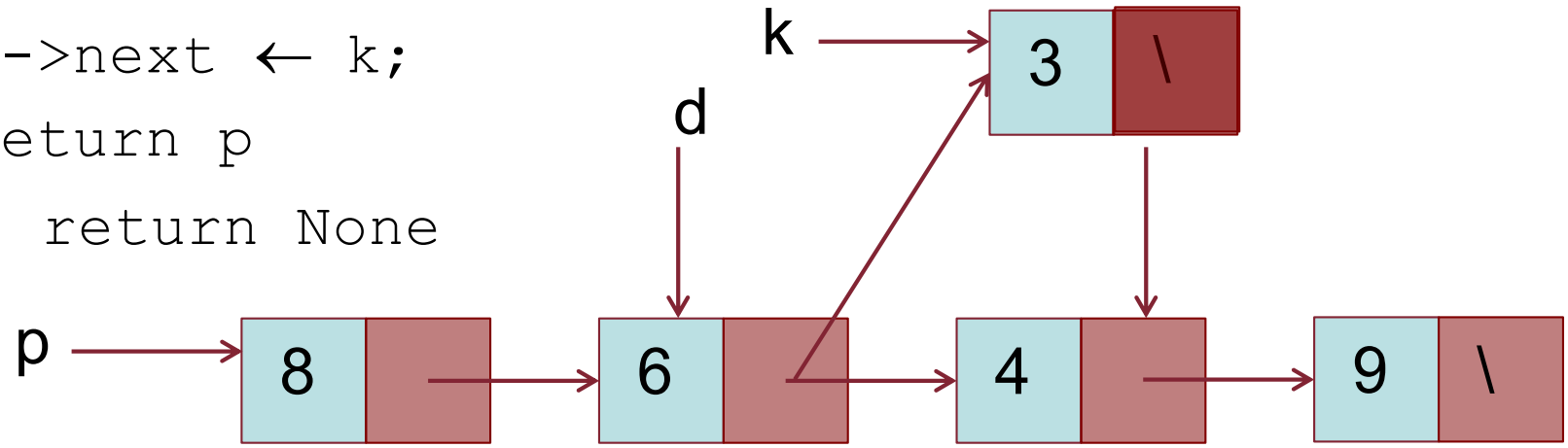
Questo è il trucco per far sì che la lista sia una struttura dati dinamica.

Tale creazione avviene tramite una **allocazione di memoria.**

Liste puntate semplici (8) - Inserimento

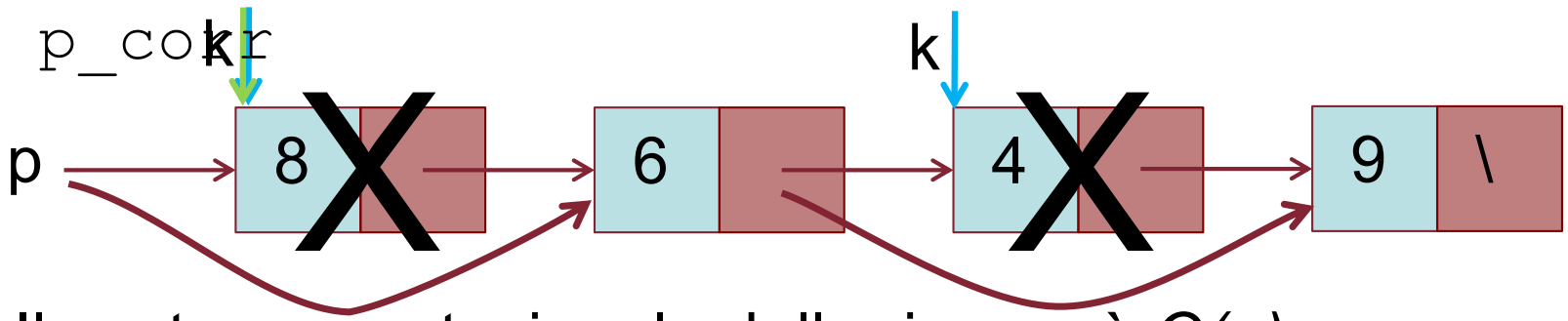
Oltre ad inserire in testa, possiamo pensare di inserire in una posizione qualsiasi, ad esempio dopo la posizione indicata da un puntatore:

```
Funzione Insert_Dopo_d(p: punt. testa;
    k: punt. a elem. da ins., d: punt. dopo cui ins.)
  if d ≠ None
     k->next ← d->next;
     d->next ← k;
     return p
  else return None
```



Liste puntate semplici (9) - Eliminazione

```
Funzione Delete (p: punt. alla lista;
                 k: puntat. all'elem. da cancellare)
   if (k ≠ None) // se k=NULL non c'è niente da cancellare
      if k = p                          // cancel. 1° elem
         p = p->next; return p
      p_corr = p
      while (p_corr-> next ≠ k)
         p_corr = p_corr-> next // qui, p_corr punta
                                //all'elem. che precede k
      p_corr-> next = k-> next
   return p
```



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

Liste puntate semplici (10) - Eliminazione

L’operazione opposta all’allocazione è quella che libera la memoria fisicamente, oltre che logicamente.

Nei moderni linguaggi di programmazione viene fatta automaticamente, ogni qual volta una zona di memoria non sia più referenziata da alcun puntatore.

Liste puntate semplici (11) - Eliminazione

Le liste sono strutture dati inerentemente ricorsive.

Pertanto, tutti gli algoritmi proposti possono essere implementati sia in versione iterative che ricorsiva.

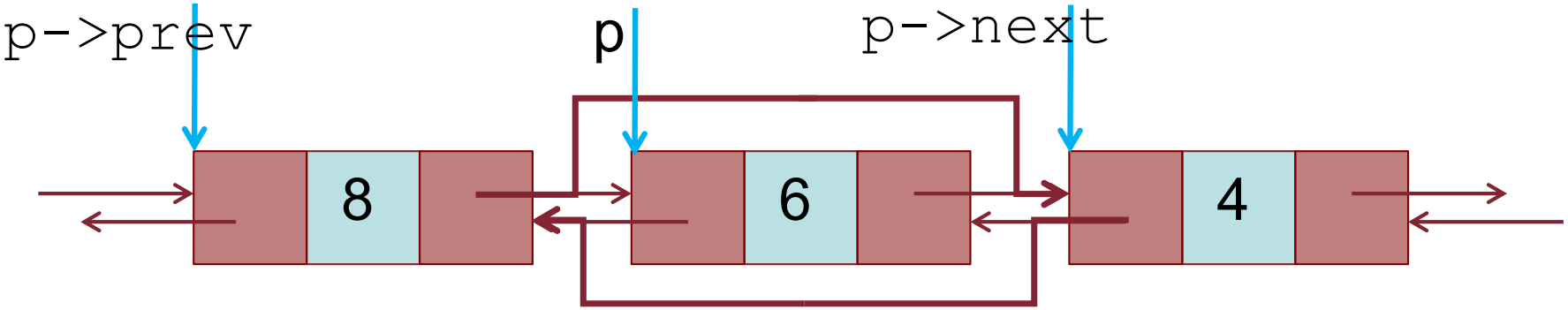
Vediamo la cancellazione:

```
Funzione Delete_Ric(p: punt. alla lista;
            k: puntat. all'elem. da canc.)
    if p==k
       p=p->next
    else
     p->next=Delete_Ric(p->next, k)
   return p
```

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***:

```
…(p-> prev) -> next = p-> next
  (p-> next)-> prev = p-> prev …
```



Liste puntate

Riassumendo, per le liste puntate 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	$O(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

Una digressione sull'oggetto *list* del Python (1)

L'oggetto ***list*** ha somiglianze e differenze sia con gli array che con le liste semplici. Infatti:

- come si fa negli array:
  - è possibile accedere ai suoi elementi tramite la loro posizione;
  - la sua lunghezza è nota tramite il comando *len*.
- Come si fa nelle liste semplici:
  - è possibile memorizzare dati non omogenei;
  - la lunghezza non è necessariamente fissa.

Una digressione sull'oggetto *list* del Python (2)

L'implementazione di questo oggetto è fatta collegando tramite puntatori elemento per elemento una lista semplice ad un array.

Costo computazionale delle operazioni semplici:

- *append* (inserimento in ultima posizione) si esegue in $\Theta(1)$;
- *insert* (inserimento in *i*-esima posizione) si esegue in $O(n)$ poiché bisogna far scorrere in avanti tutti gli elementi in posizione successiva alla *i*-esima;
- *pop* oppure *del* (elimina e restituisce oppure elimina l'elemento in *i*-esima posizione) si esegue in $O(n)$ poiché bisogna far scorrere indietro gli elementi in posizione successiva;
- *concatenate* (unifica due oggetti di tipo *list* in uno solo) costa $\Theta(k)$ dove k è la lunghezza del secondo oggetto.

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:

- data in input una lista puntata tramite il puntatore al primo elemento, restituire il puntatore all'ultimo elemento;
- data in input una lista puntata tramite il puntatore al primo elemento, restituire il puntatore al penultimo elemento;
- data in input una lista puntata tramite il puntatore al primo elemento, restituire il puntatore alla stessa lista da cui sia stato eliminato l'ultimo elemento;
- …

Esercizi (2)

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

- data in input una lista puntata 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 (N.B. non deve essere creato alcun record, ma bisogna "smontare" e "rimontare" opportunamente i record iniziali).
- data in input una lista puntata tramite il puntatore al primo elemento, restituire i puntatori a due liste, una con gli elementi di posto pari nella lista di partenza, ed una con gli elementi di posto dispari (anche qui, non bisogna creare nuovi record);
- …

Esercizi (3)

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

- data in input una lista puntata 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;
- data in input una lista puntata di interi tramite il puntatore al primo elemento, restituire la lista ordinata (senza creare nuovi record).