Corso di laurea in Informatica
Introduzione agli Algoritmi

Strutture dati fondamentali: Insiemi statici ed insiemi dinamici. operazioni su di essi con le strutture dati semplici (array e liste concatenate)

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

Strutture dati

- Una struttura dati è composta da:
  - un modo sistematico di organizzare i dati
  - un insieme di operatori che permettono di manipolare la struttura

- Le strutture dati possono essere:
  - omogenee o disomogenee (rispetto ai dati contenuti).
  - statiche o dinamiche (a seconda che possano o meno variare la dimensione nel tempo)
  -

Insiemi dinamici

Un insieme dinamico è una struttura dati in cui la dimensione può cambiare dinamicamente durante l'esecuzione del programma.
Possono crescere o ridursi automaticamente per ospitare un numero variabile di elementi senza dover specificare una dimensione massima in anticipo.
Sono utili quando la dimensione dell'insieme è incerta o può cambiare nel tempo.

Esempi di insiemi dinamici includono **le liste a puntatori** in cui gli elementi possono essere aggiunti o rimossi senza restrizioni sulla dimensione.

Insiemi statici

Un insieme statico è una struttura dati omogenei in cui la dimensione è fissa e deve essere specificata al momento della creazione.
Non possono cambiare di dimensione durante l'esecuzione del programma e sono generalmente più efficienti in termini di spazio e accesso.
Sono utili quando si conosce a priori la dimensione massima dell'insieme e si vuole evitare l'overhead di gestire una dimensione variabile.
Esempi di insiemi statici includono **array** in cui la dimensione è fissata durante la dichiarazione e non può cambiare senza creare una nuova struttura dati.

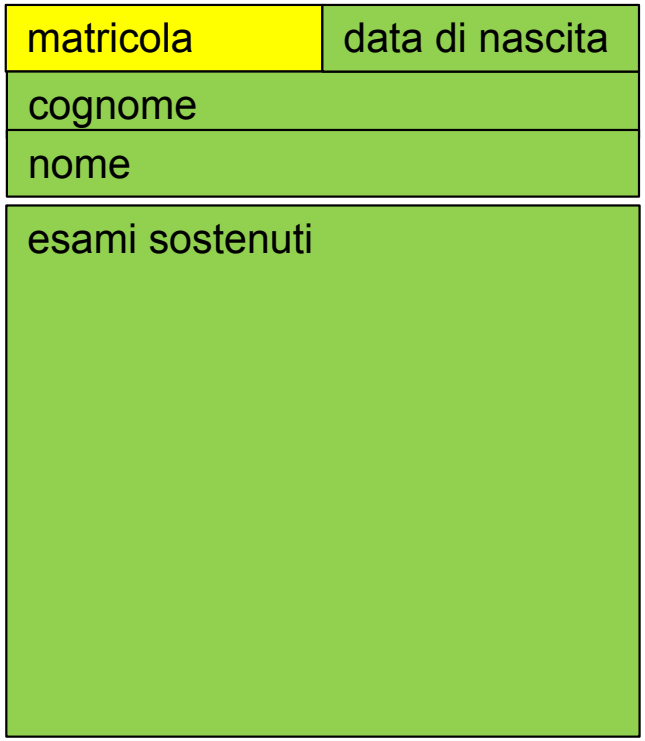
Ad esempio in C con $int\ A[5]$ viene creato un array A di 5 interi (le 5 locazioni contengono all'inizio valori casuali e per avere valori definiti bisogna assegnarglieli.

Insiemi statici o insiemi dinamici?

In generale, la scelta tra un insieme dinamico e uno statico dipende dalle esigenze specifiche del problema e dalle operazioni previste sull'insieme.

Gli insiemi dinamici offrono maggiore flessibilità ma possono richiedere più memoria e avere prestazioni leggermente inferiori rispetto agli insiemi statici, che sono più efficienti ma meno flessibili nella dimensione.

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; 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.

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.

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

- $Search(S, x)$: recuperare l'elemento con chiave di valore x, se è presente in S, restituire un valore speciale altrimenti;
- $Min(S)$: recuperare il minimo valore presente in S;
- $Max(S)$: recuperare il massimo valore presente in S;

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

- $Insert(S, x)$: inserire un elemento di valore x in S;
- $Delete(S, x)$: eliminare da S l'elemento a valore x se presente

Array (o vettore)

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

Osservazione

L'array 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 array con una struttura dati dinamica.

In sintesi, gli array possono essere considerati strutture dati statiche o dinamiche in base al modo in cui vengono implementati in un dato linguaggio di programmazione.

Posso ad esempio parlare della struttura dati lista di python come ad un array dinamico.

Array

Search(S, x)
- 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: $\Theta(1)$

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

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

Vettori

Riassumendo:

struttura dati	$search(S, x)$	$\min(S),\ \max(S)$	$insert(S, x)$	$delete(S, x)$
vettore qualunque	$O(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
vettore ordinato	$O(\log n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$

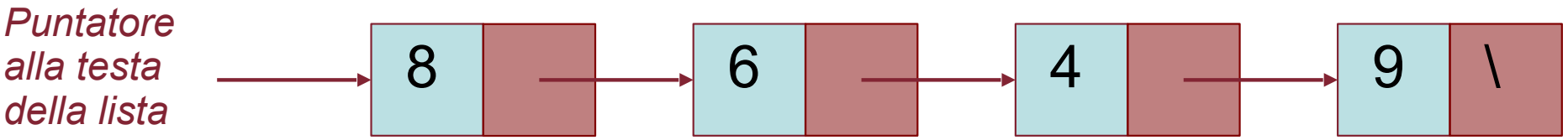
Già da questo semplice confronto delle due strutture dati più semplici in assoluto, si può dedurre come non abbia senso dire che una struttura è migliore di un'altra: le strutture dati possono essere più o meno adatte ad un certo algoritmo e sta al buon progettista disegnare un algoritmo efficiente usando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

Liste semplici

La ***lista semplice (o lista concatenata)*** 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.



NOTA. Si deve ricordare sempre che il *valore di una variabile di tipo puntatore è un indirizzo di memoria*. Un puntatore è una variabile di tipo particolare che contiene l'indirizzo in memoria di un'altra variabile

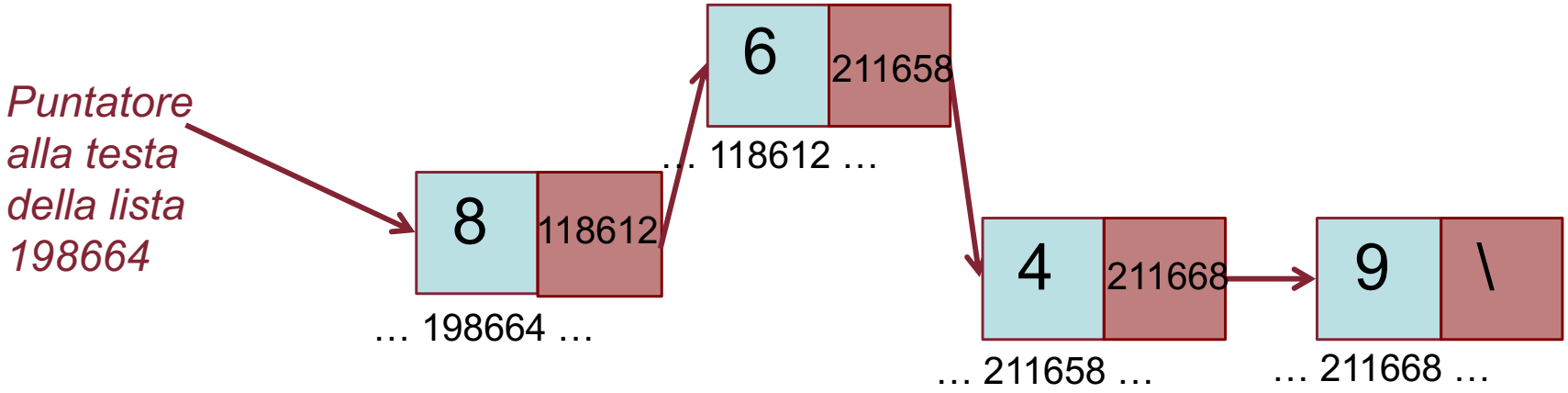
Il valore del puntatore è *l'indirizzo del primo byte* del record puntato, indipendentemente dalle dimensioni del record stesso.

accesso diretto

I vettori, ordinati e non, godono di un **accesso diretto**: per accedere ad un dato basta conoscerne la posizione nel vettore (cioè l'indice).
Segue che l'accesso a qualsiasi dato in un array ha un costo *Θ(1).*

accesso sequenziale

In una lista, 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 $\Theta(n)$.



Riassumendo:

Differenze fra vettori e liste:

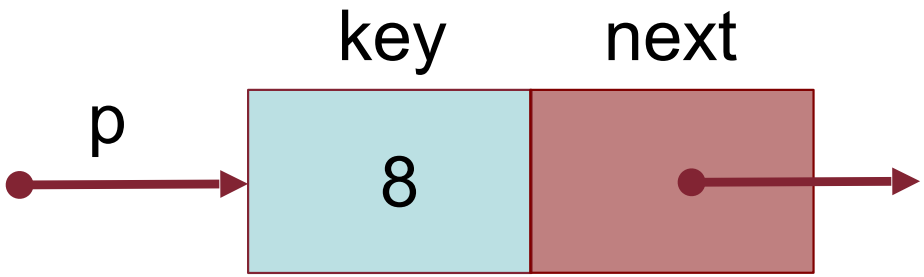
- Vettore: il numero degli elementi è prefissato e la successione degli elementi è realizzata mediante gli indici degli elementi. E' possibile l'accesso diretto.

- Lista puntata: 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 puntate semplici

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 *None* (visualizzato col simbolo \).



:

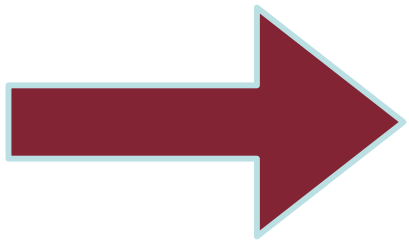
- *p.key* indica il contenuto del campo key dell'elemento puntato da p.
- *p.next* indica il contenuto del campo next dell'elemento puntato da p, ossia l'indirizzo dell'elemento successivo (o None se esso è l'ultimo elemento).

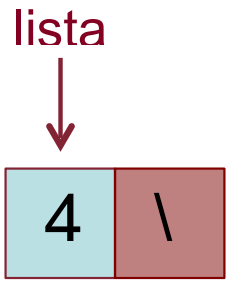
Ad esempio in python possiamo definire la classe nodo come segue:

```
class Nodo:
    def __init__(self, key = None, next = None):
        self.key  = key
        self.next = next
```

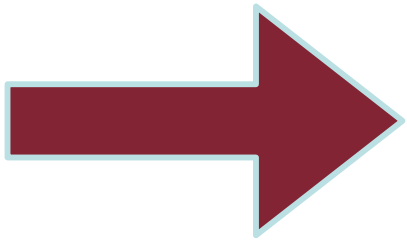
A questo punto possiamo generare nodi e concatenarli ad ottenere liste:

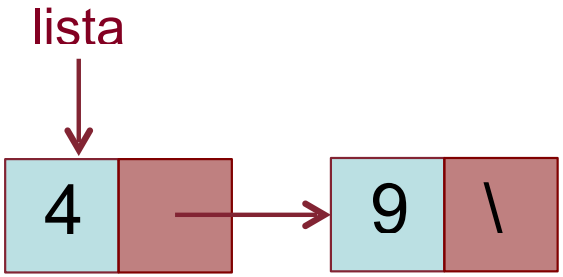
```
>>> lista=Nodo(4)
```



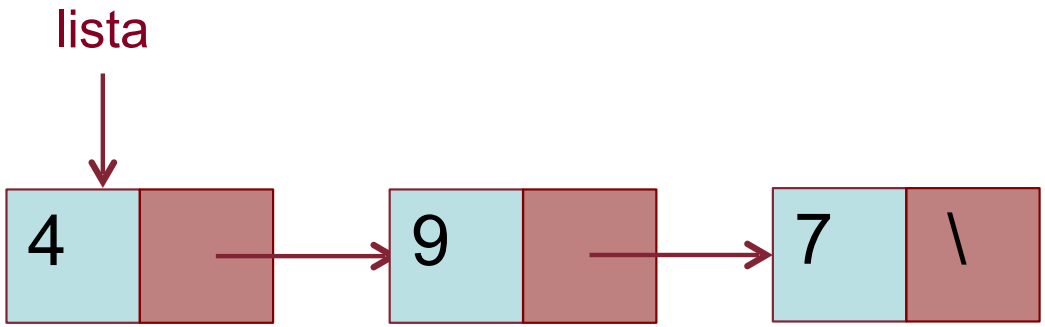



```
>>> lista.next=Nodo(9)
```

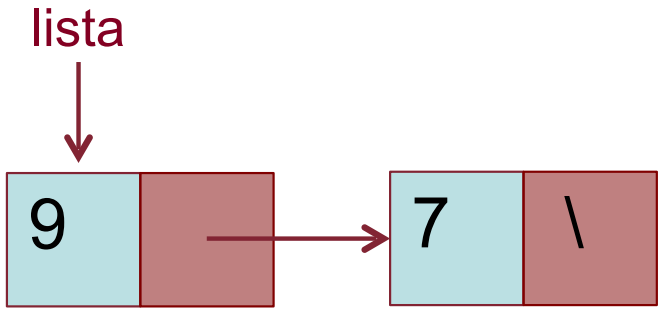




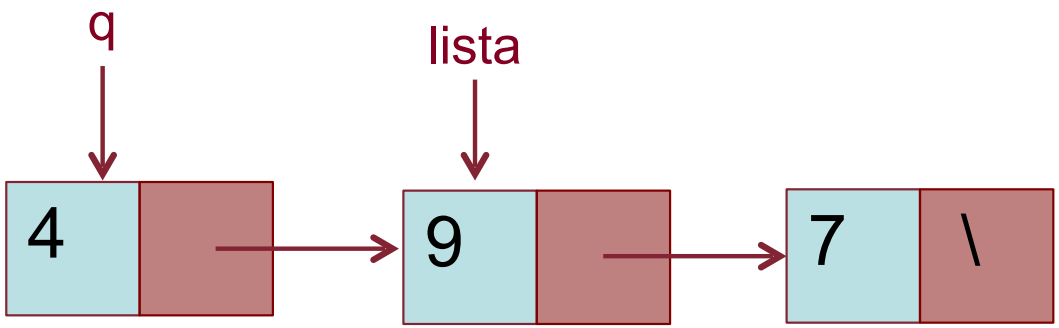
```
>>> lista.next.next = Nodo(7)
```




```
>>> lista = lista.next
```



Nota che se *lista* era l'unico puntatore a referenziare il il nodo a valore 4 allora, come conseguenza dell'ultima istruzione il nodo a valore 4 non essendo più referenziato non solo non compare più tra i nodi della lista ma lo spazio da lui occupato viene restituito al sistema per essere poi riutilizzato. Nel caso in cui invece il nodo era referenziato anche da un altro puntatore ad esempio *q*, il risultato dell'istruzione sarebbe stato il seguente:

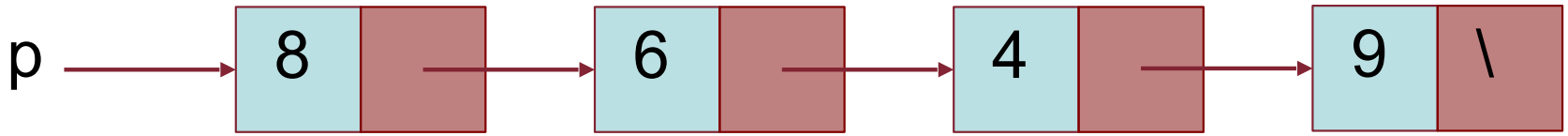


Liste puntate - Creazione

La funzione *Crea* prende in input un array A, genera una lista semplice contenente un nodo per ogni elemento dell'array e termina restituendo la testa alla lista creata.

```
def Crea(A):
    if A==[]: return None
    p = Nodo(A[0])
    q = p
    for i  in range(1,len(A)):
        q.next = Nodo(A[i])
        q = q.next
    return p
```

Il risultato di *Crea*($[8, 6, 4, 9]$) è la creazione della seguente lista:



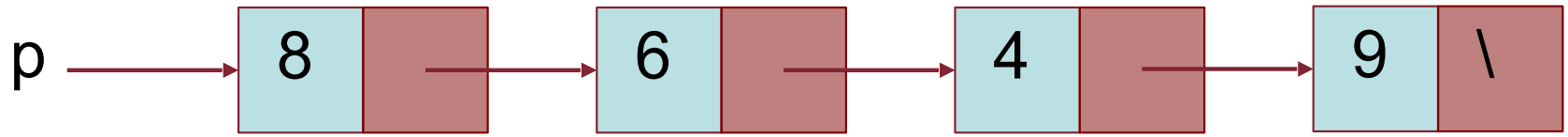
Il costo computazionale della creazione è $\Theta(n)$.

Liste puntate - Stampa

La funzione *Stampa* prende in input il puntatore p alla testa di una lista semplice e stampa i contenuti di ciascun nodo della lista

```
def Stampa(p):
    while p:
        print(p.key)
        p = p.next
```

Considera la seguente lista:



Il risultato di *Stampa*(p) è la seguente stampa:

8
6
4
9

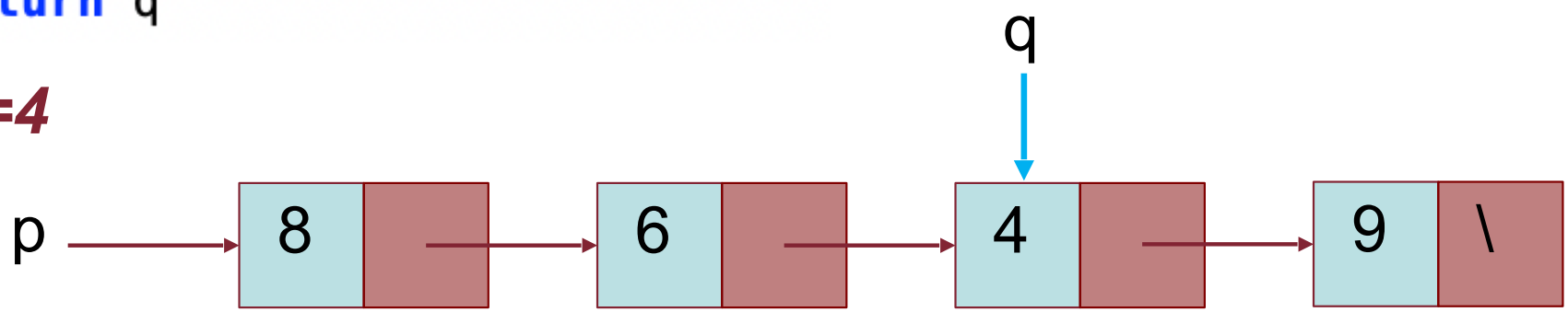
Il costo computazionale della stampa è $\Theta(n)$.

Liste puntate - Ricerca

La funzione *Ricerca* dato il puntatore p alla testa di una lista semplice e un valore x restituisce il puntatore al primo nodo della lista contenente il valore x , il puntatore restituito sarà None se nella lista un tale nodo non è presente.

```
def Ricerca(p, x):
    q = p
    while q != None and q.key != x:
        q = q.next
    return q
```

x=4



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

I puntatori vengono passati per valore quindi il risultato sarebbe stato lo stesso con il seguente programma:

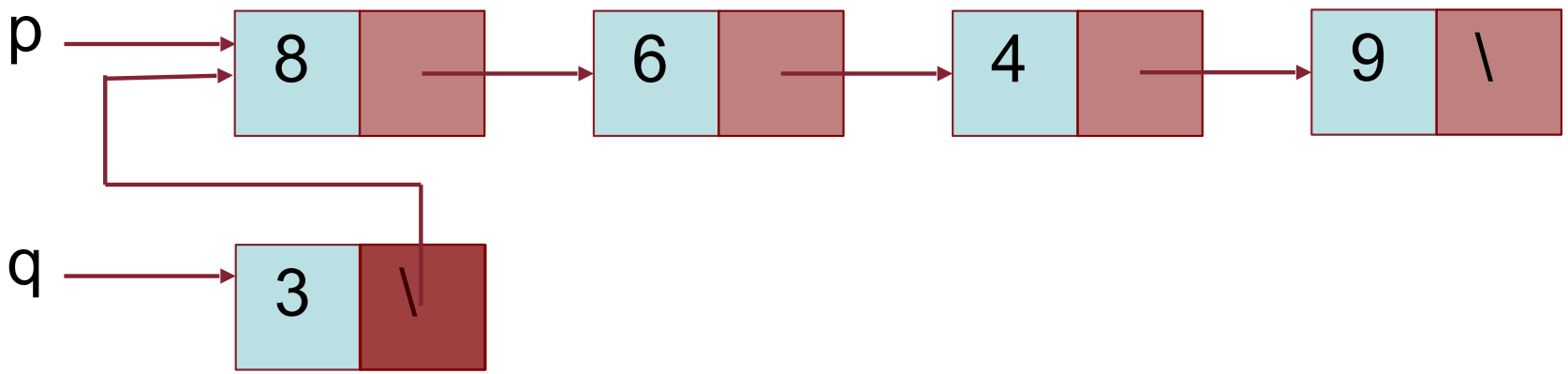
```
def Ricerca(p, x):
    while p != None and p.key != x:
        p = p.next
    return p
```

Liste semplici - Inserimento

La funzione *Inserisci* dato il puntatore *p* alla testa di una lista semplice ed un intero *x* ad un nodo inserisce un nodo con quel valore in testa alla lista e restituisce la nuova testa della lista.

```
def Inserisci(p, x):
    q = Nodo(x, p)
    return q
```

Il risultato di *Inserisci*(*p*, 3) è la seguente lista con testa *q*



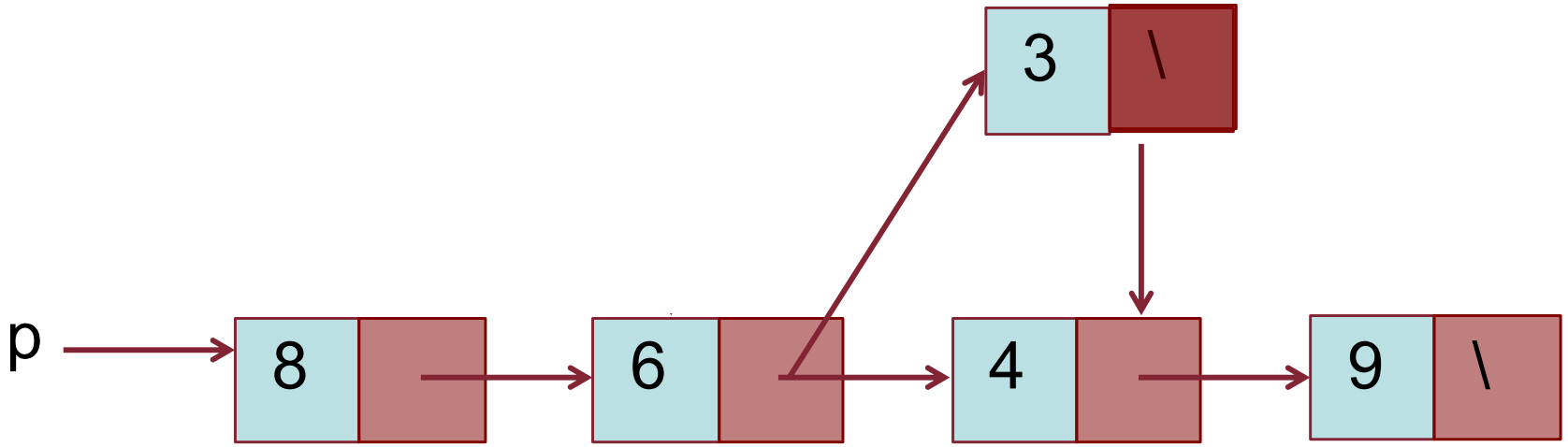
Il costo computazionale dell'inserimento in testa è $\Theta(1)$.

Liste semplici - Inserimento(2)

Oltre ad inserire in testa, possiamo pensare di inserire il valore x in una posizione qualsiasi, ad esempio DOPO l'eventuale prima occorrenza di un nodo con valore y:

```
def Inserisci1(p, x, y):
    while p != None and p.key != y:
        p = p.next
    if p!= None:
        p.next = Nodo(x, p.next)
```

Il risultato di *Inserisci*1(p, 3, 6) è la seguente lista

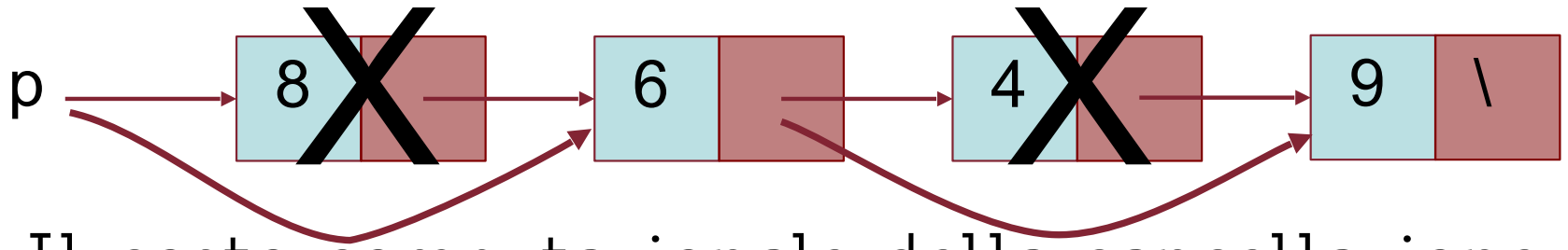


Il costo computazionale di questo inserimento è $\Theta(1)$.

Liste semplici - Eliminazione

La funzione *cancella*, dato il puntatore *p* alla testa di una lista ed il valore x cancella l'eventuale prima occorrenza di x dalla lista e restituisce la testa della lista (eventualmente modificata).

```
def Cancella(p, x):
    if p!= None:
        if p.key == x:
            p  = p.next
        else:
            q = p
            while q.next!= None and   q.next.key != x:
                q = q.next
            if q.next != None:
                q.next = q.next.next
    return p
```



Il costo computazionale della cancellazione è $O(n)$.

Liste semplici - Eliminazione di nodi

Ricorda che:
Con *q = Nodo()* viene creato un nodo puntato da *q*. Per poterlo eventualmente cancellare e liberare così la memoria occupata basta rendere il nodo inaccessibile ad esempio con l'istruzione *q = None*.
Un nodo che non è referenziato da alcun puntatore è di fatto inaccessibile al programma e viene quindi automaticamente restituito dal sistema in modo che lo spazio da lui occupato possa essere riutilizzato.
Nota che se ci sono altri puntatori che referenziano il nodo puntato da *q* lo spazio di memoria non viene liberato in quanto il nodo risulta ancora referenziato anche dopo l'istruzione *q = None*.

Liste puntate semplici - Eliminazione(2)

Le liste sono strutture dati inerentemente ricorsive.
Pertanto, tutti gli algoritmi proposti possono essere implementati sia in versione iterativa che ricorsiva.

Vediamo la cancellazione ricorsiva dell'eventuale prima occorrenza dell'intero x dalla lista :

```
def CancellaR(p, x):
    if p == None : return p
    if p.key == x: return p.next
    p.next = CancellaR(p.next, x)
    return p
```

L'oggetto *list* di python ha somiglianze e differenze sia con gli array che con le liste a puntatori:

Come per gli array:

- è possibile accedere ai suoi elementi tramite la loro posizione.
- la sua lunghezza è nota in tempo costante tramite il comando *len*.

Come per le liste a puntatori:

- è possibile memorizzare dati non omogenei.
- la lunghezza non è necessariamente fissa.

In C, l'accesso agli elementi di un array tramite l'indice è un'operazione a tempo costante $\Theta(1)$. Ciò è vero perché gli array sono strutture di dati contigue in memoria e contenenti elementi omogenei quindi calcolare la posizione di un elemento in un array a partire dalla posizione del primo elemento dell'array è molto efficiente.

In python l'accesso ad un elemento di una lista è generalmente considerato un'operazione a tempo costante ma ci sono alcune considerazioni da fare. Python implementa le liste come array di puntatori a oggetti (in una lista gli oggetti non sono necessariamente omogenei) quindi l'accesso agli elementi di una lista coinvolge due elementi di indirezione: prima l'accesso all'elemento dell'array che contiene il puntatore all'oggetto e poi l'accesso all'oggetto effettivo. Tuttavia poiché il puntatore all'oggetto è di dimensione fissa, l'accesso è comunque O(1) in termini di complessità temporale.

Costo computazionale delle operazioni python sulla lista A:

- $A.append(x)$ (inserimento di x nell'ultima posizione di A): $\Theta(1)$

- $A.insert(i,x)$ (inserimento di x nell' i-esima posizione di A): $O(n)$ poiché bisogna far scorrere di una posizione verso destra tutti gli elementi in posizione successiva alla i-esima;

- $pop()$ estrazione dell'ultimo elemento della lista: $\Theta(1)$

- $pop(i)$ estrazione da A dell'elemento in i-esima posizione: $O(n)$ poiché bisogna far scorrere di una posizione verso sinistra tutti gli elementi dalla posizione i+1 in poi

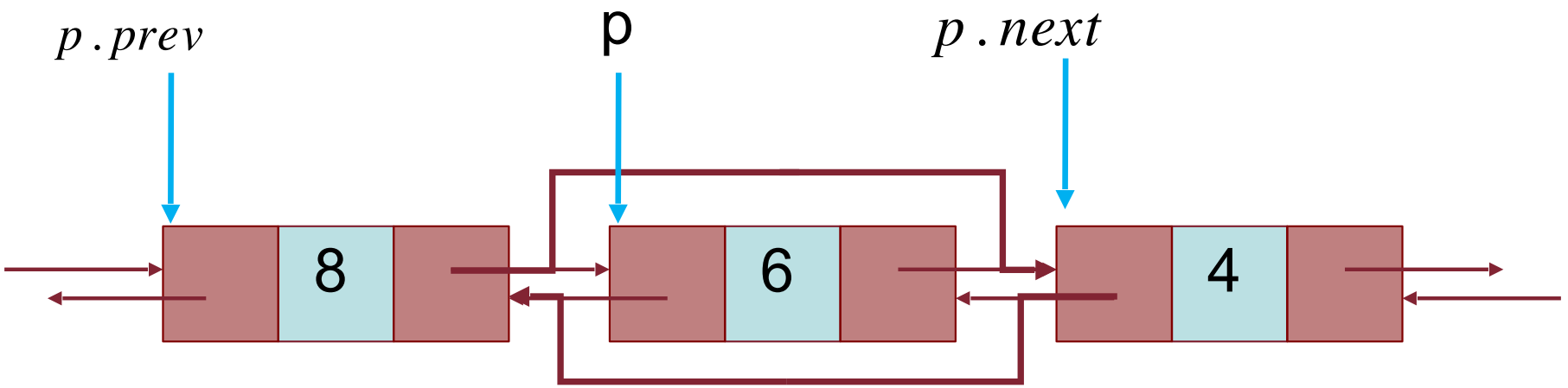
- $A.extend(B)$ (accoda ad A gli elementi della lista B): $\Theta(n_1)$ dove n_1 è la lunghezza di B.

Liste doppiamente puntate

E' possibile riorganizzare la struttura della lista in modo che da ogni suo elemento si possa accedere sia all'elemento che lo segue che a quello che lo precede, quando essi esistono. Tale struttura dati si chiama ***lista doppia*** o ***lista doppiamente concatenata*** o ***lista doppiamente puntata.***

Ad esempio in python possiamo definire la classe dei nodi di queste liste come:

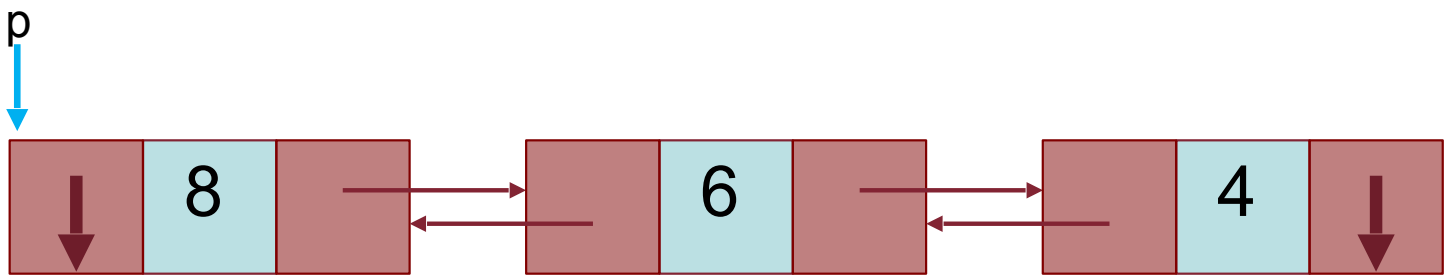
```
class NodoD:
    def __init__(self, key = None, prev = None, next = None):
        self.key  = key
        self.next = next
        self.prev = prev
```



Liste doppiamente puntate - Creazione

```
def creaDoppia(A):
    if A==[]: return None
    p=NodoD(A[0])
    q=p
    for i in range(1, len(A)):
        q.next = NodoD(A[i], q)
        q = q.next
    return p
```

```
>>> p=creaDoppia([8,6,4])
>>> stampa(p)
8
6
4
```

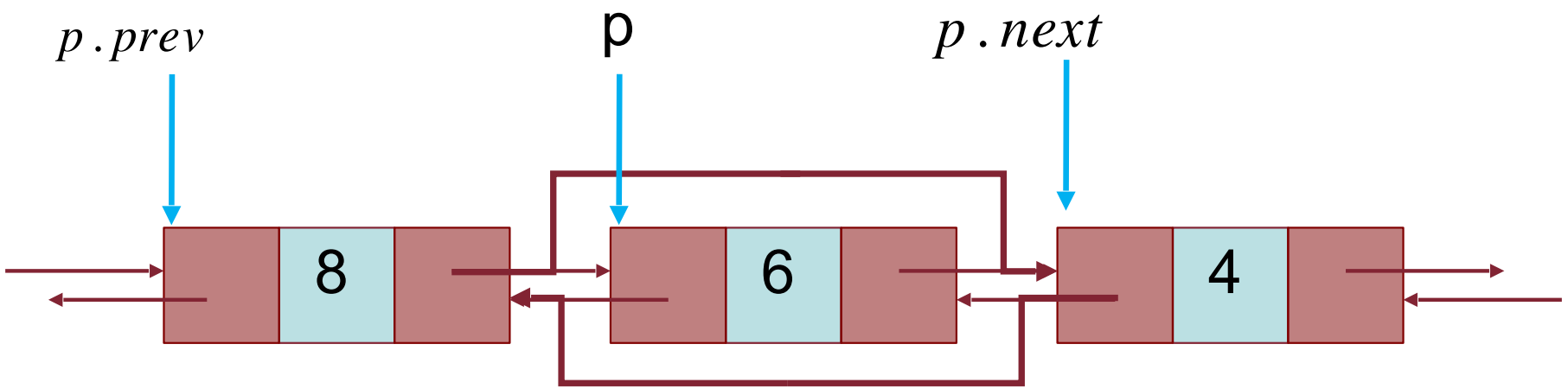


Il costo computazionale della creazione è $O(n)$.

Liste doppiamente puntate

Se voglio "estromettere" dalla lista l'elemento puntato da p:

```
…
p.prev.next = p.next
p.next.prev = p.prev
…
```



Corso di laurea in Informatica

Introduzione agli Algoritmi

Esercizi per casa

SAPIENZA
UNIVERSITÀ DI ROMA

Esercizi (1)

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

- data una lista tramite il puntatore al suo primo elemento, restituire il puntatore all'ultimo elemento se la lista ha almeno un elemento, *None* altrimenti;
- data una lista tramite il puntatore al suo primo elemento, restituire il puntatore al penultimo elemento se la lista ha almeno due elementi, *None* altrimenti;
- data una lista tramite il puntatore al suo primo elemento, restituire il puntatore alla stessa lista da cui sia stato eliminato l'ultimo elemento;
- data una lista tramite il puntatore al suo 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).

Esercizi (2)

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

- data una lista tramite il puntatore al suo 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 nodi);
- data una lista di interi tramite il puntatore al suo primo elemento, stampare tutti i valori che compaiono almeno due volte nella lista;
- data una lista ordinata di interi tramite il puntatore al suo primo elemento ed un intero x, aggiungere x alla lista in modo da rispettare l'ordinamento;
- data in input una lista di interi tramite il puntatore al primo elemento, restituire la lista ordinata (senza creare nuovi nodi).

Esercizi (3)

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

- Dato il puntatore ad una lista doppiamente puntata restituire la lunghezza della lista;
- Dato il puntatore ad una lista doppiamente puntata stampare gli elementi della lista in ordine inverso (i.e. dall'ultimo al primo);
- Dato il puntatore ad una lista di interi doppiamente puntata ed un intero x cancellare tutte le occorrenze di x dalla lista;
- Dato un intero x ed il puntatore al primo elemento di una lista di interi ordinata e doppiamente puntata inserire l'intero x nella lista in modo da mantenere l'ordine;
- Dato il puntatore ad una lista di interi doppiamente puntata ed ordinata cancellare dalla lista tutti i duplicati.