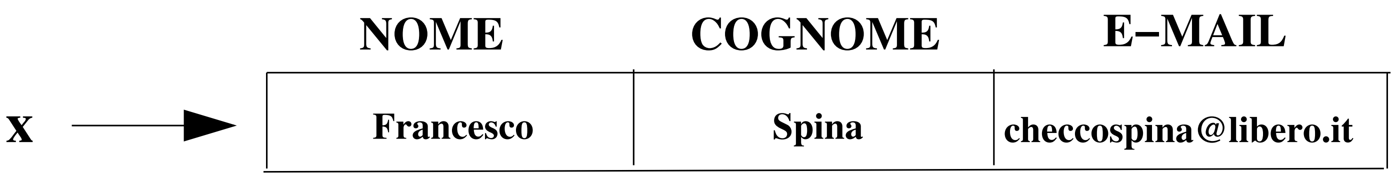
Algoritmi e Strutture di Dati I

Massimo Franceschet

http://www.sci.unich.it/~francesc

Oggetti e puntatori

Un **oggetto** è un'area di memoria suddivisa in sezioni chiamate **campi**.

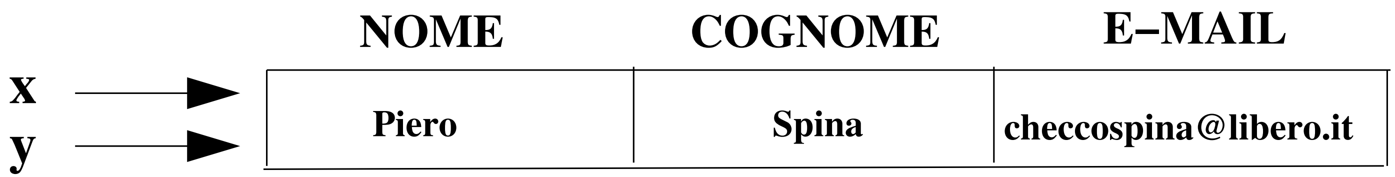


Per accedere ad un oggetto, devo conoscere il suo **indirizzo di memoria**. Un **puntatore** è una variabile che contiene un indirizzo di memoria.

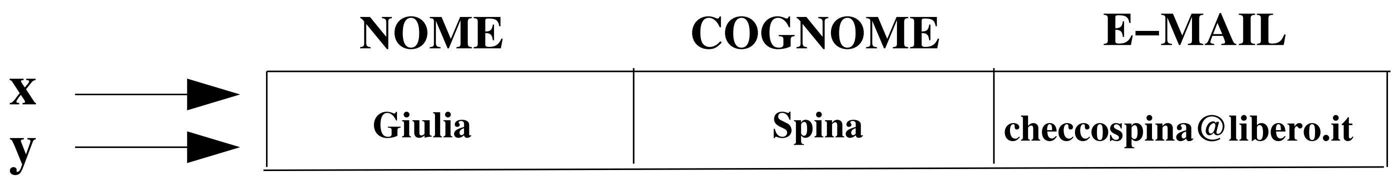
$$\mathrm{NOME}[x] \leftarrow \mathit{Piero}$$

	NOME	COGNOME	E–MAIL
x →	Piero	Spina	checcospina@libero.it

$$y \leftarrow x$$



$$\text{NOME}[y] \leftarrow Giulia$$



Liste (list)

La struttura di dati **lista** rappresenta un **insieme dinamico** nel seguente modo:

- non ci sono particolari politiche di **inserimento e cancellazione** degli elementi;
- non esiste una **capacità massima** della struttura;
- l'accesso agli elementi è **sequenziale**;
- la gestione della memoria è **dinamica**;
- gli elementi della lista sono **oggetti**.

Accesso: diretto vs. sequenziale

- **Accesso diretto**: posso accedere ad ogni elemento con una sola operazione senza dover passare per gli elementi che lo precedono. Il costo di accesso è **costante** indipendentemente da dove si trova l'elemento.
- **Accesso sequenziale**: posso accedere ad ogni elemento solo passando per tutti gli elementi che lo precedono nella struttura. Il costo di accesso **dipende dalla posizione** dell'elemento.

Gestione della memoria: dinamica vs. statica

- **Gestione statica**: la memoria viene allocata all'inizio del programma e viene deallocata alla fine del programma. Non abbiamo allocazioni o deallocazioni durante l'esecuzione del programma.
- **Gestione dinamica**: la memoria viene allocata durante l'esecuzione del programma quando mi serve e viene deallocata durante l'esecuzione del programma quando non ne ho più bisogno.

Una gestione dinamica della memoria è preferibile in quanto permette in ogni momento di usare esattamente la memoria che serve.

Lista doppia

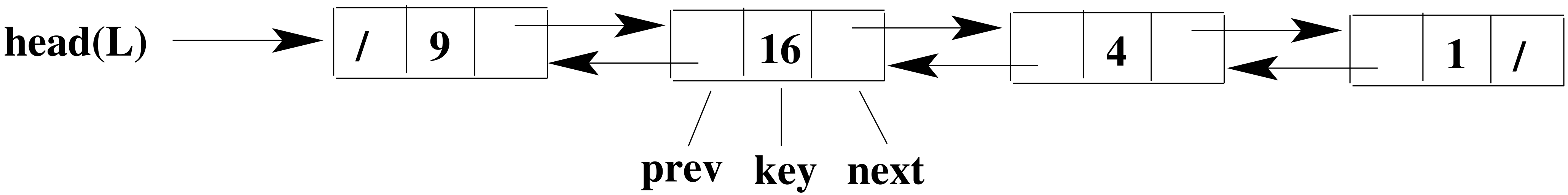
Useremo un modello di lista chiamato **lista doppia** o **lista bidirezionale**. Gli oggetti contenuti in una lista doppia hanno tre campi: *key*, *next* e *prev*. Sia x un oggetto della lista:

- **key**[**x**] contiene una **chiave** numerica all'oggetto x;
- **next**[**x**] è un **puntatore all'oggetto che segue** x, se tale oggetto esiste, oppure NIL se x è l'ultimo oggetto della lista.
- **prev**[**x**] è un **puntatore all'oggetto che precede** x, se tale oggetto esiste, oppure NIL se x è il primo oggetto della lista.

Possono inoltre esistere altri campi, chiamati **dati satellite**.

Una lista L possiede un attributo **head**(**L**) che contiene un puntatore al **primo oggetto** di L.

Lista doppia



Operazioni su liste

La struttura di dati lista possiede un attributo $head(L)$ e le seguenti quattro operazioni:

- $ListEmpty(L)$ che controlla se la lista L è vuota;
- $ListSearch(L, k)$ che ritorna un puntatore all'oggetto con chiave k, se esiste, e NIL altrimenti;
- $ListInsert(L, x)$ che inserisce l'oggetto puntato da x in testa alla lista;
- $ListDelete(L, x)$ che cancella l'oggetto puntato da x.

Ricerca

ListEmpty(L)

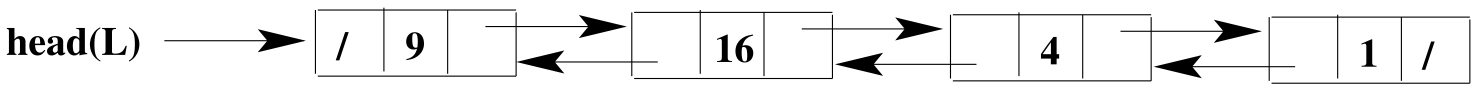
```
if head(L) = NIL then
    return TRUE
else
    return FALSE
end if
```

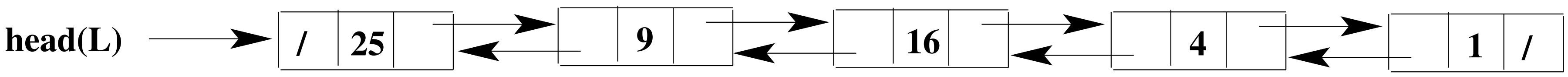
ListSearch(L,k)

```
x ← head(L)
while x ≠ NIL and key[x] ≠ k do
    x ← next[x]
end while
return x
```

Inserimento

Supponiamo di eseguire $Insert(L, x)$, ove x punta ad un oggetto con chiave 25.





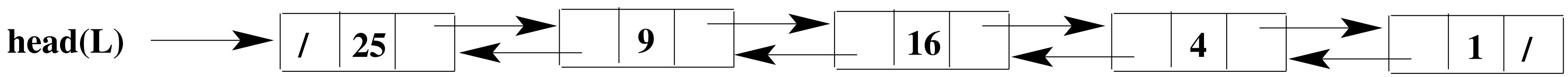
Inserimento

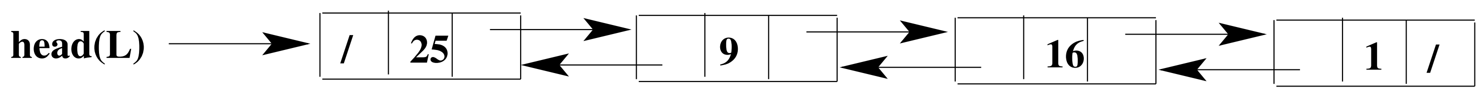
ListInsert(L,x)

$next[x] \leftarrow head(L)$
if $head(L) \neq$ NIL **then**
    $prev[head(L)] \leftarrow x$
end if
$head(L) \leftarrow x$
$prev[x] \leftarrow$ NIL

Cancellazione

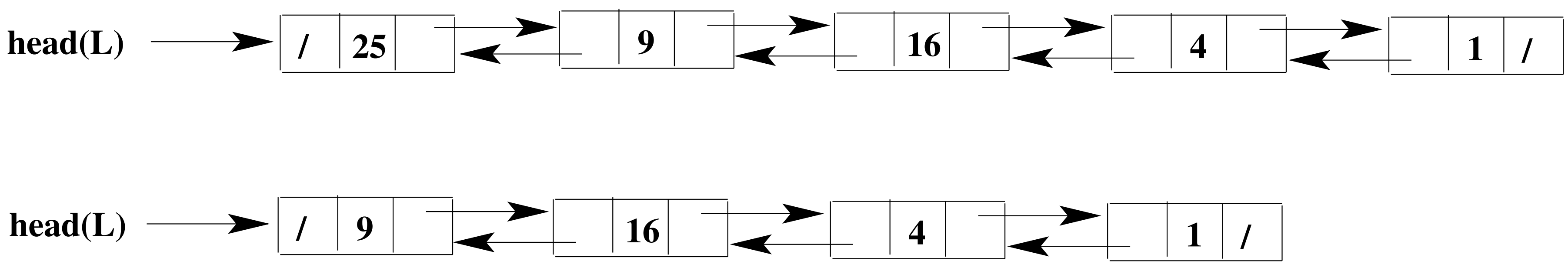
Supponiamo di eseguire $Delete(L, x)$, ove x punta all'oggetto con chiave 4.





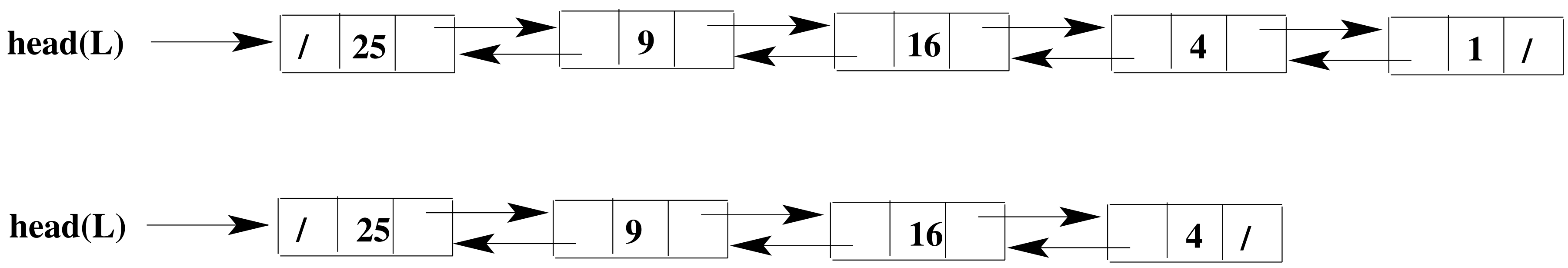
Cancellazione

Supponiamo di eseguire $Delete(L, x)$, ove x punta all'oggetto con chiave 25.



Cancellazione

Supponiamo di eseguire $Delete(L, x)$, ove x punta all'oggetto con chiave 1.



Cancellazione

ListDelete(L,x)

if $prev[x] \neq$ NIL **then**
    $next[prev[x]] \leftarrow next[x]$
else
    $head(L) \leftarrow next[x]$
end if
if $next[x] \neq$ NIL **then**
    $prev[next[x]] \leftarrow prev[x]$
end if

Complessità delle operazioni su liste

- La ricerca ha un **costo lineare** $\Theta(n)$ nella lunghezza n della lista. Il caso pessimo si ha quando non trovo l'elemento cercato;
- le operazioni di modifica (inserimento e cancellazione) hanno un **costo costante** $\Theta(1)$.

Esercizio *Sia L una lista. Si scriva una procedura $ListLength(L)$ che ritorna la lunghezza della lista L.*

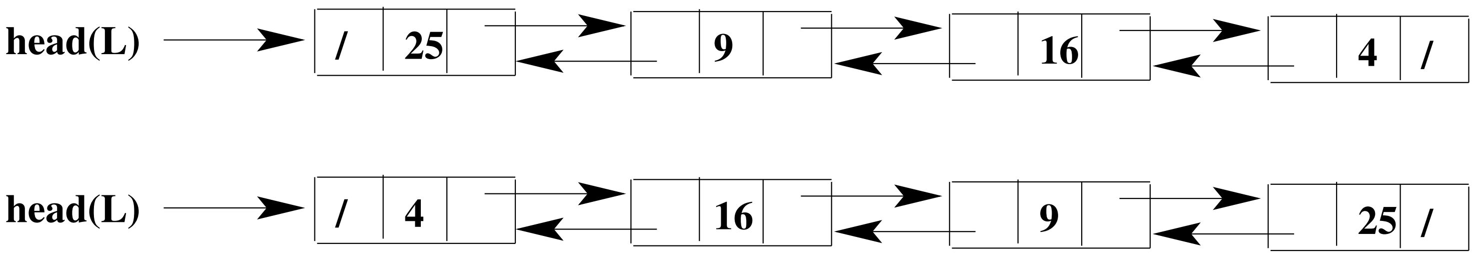
Esercizio *Sia* L *una lista. Si scriva una procedura* $ListLength(L)$ *che ritorna la lunghezza della lista* L.

ListLength(L)

$x \leftarrow head(L)$
$len \leftarrow 0$
while $x \neq$ NIL **do**
    $x \leftarrow next[x]$
    $len \leftarrow len + 1$
end while
return len

La complessità di $\Theta(n)$, con n è la lunghezza della lista L.

Esercizio *Sia L una lista. Si scriva una procedura $ListReverse(L)$ che inverte la sequenza contenuta in L.*



Esercizio *Sia L una lista. Si scriva una procedura $ListReverse(L)$ che inverte la sequenza contenuta in L.*

ListReverse(L)

$y \leftarrow$ NIL
$x \leftarrow head(L)$
while $x \neq$ NIL **do**
  $t \leftarrow next[x]$
  $next[x] \leftarrow prev[x]$
  $prev[x] \leftarrow t$
  $y \leftarrow x$
  $x \leftarrow t$
end while
$head(L) \leftarrow y$

La complessità di $\Theta(n)$, con n è la lunghezza della lista L.

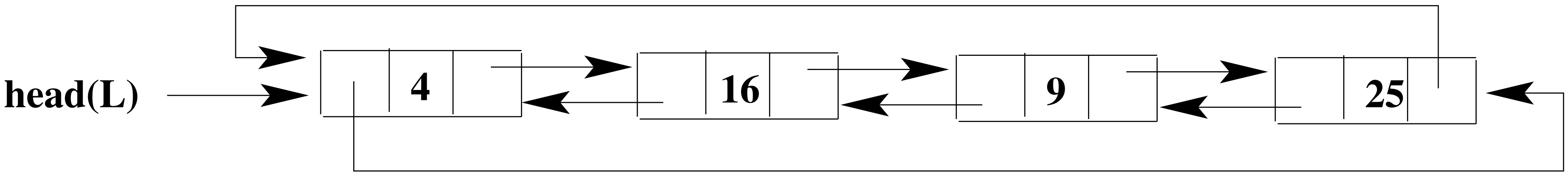
Esercizio *Siano* L *e* M *due liste. Si scriva una procedura* $ListCompare(L, M)$ *che confronta il contenuto di* L *e* M*.*

ListCompare(L,M)

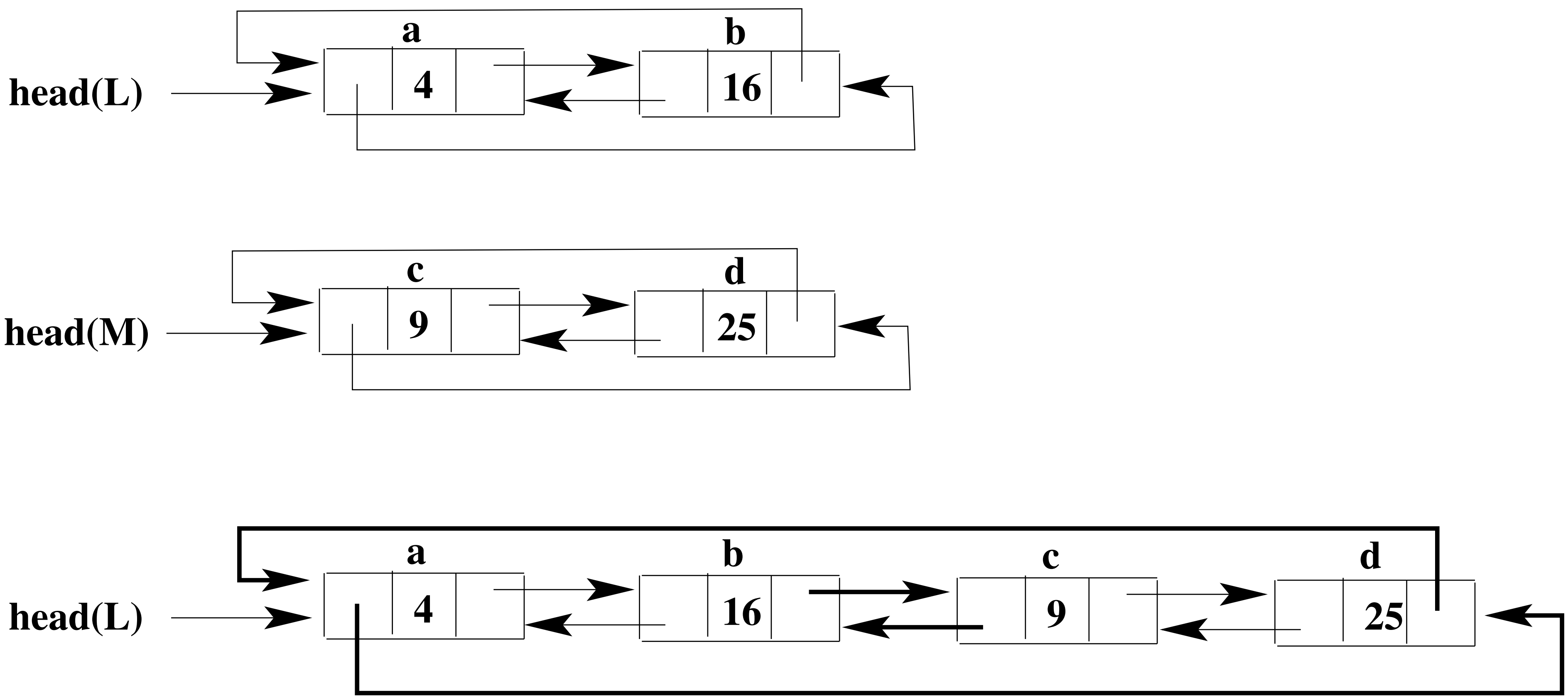
```
x ← head(L)
y ← head(M)
while (x ≠ NIL) and (y ≠ NIL) do
    if key[x] ≠ key[y] then
        return FALSE
    else
        x ← next[x]
        y ← next[y]
    end if
end while
if (x = NIL) and (y = NIL) then
    return TRUE
else
    return FALSE
end if
```

La complessità di $\Theta(min\{n, m\})$, con n è la lunghezza di L e m è la lunghezza di M.

Lista circolare



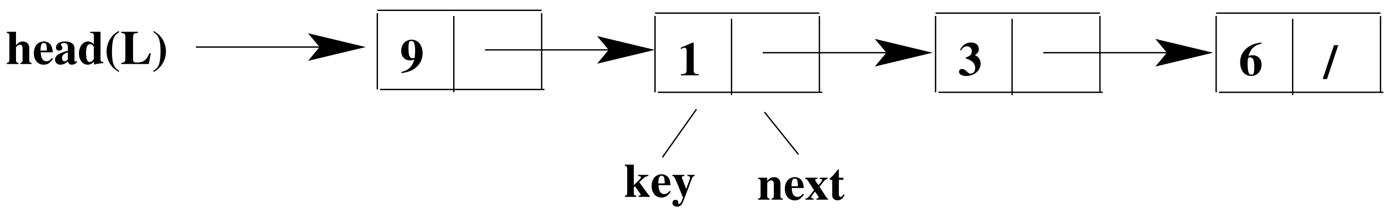
Esercizio *Siano L e M due liste circolari. Si scriva una procedura $CircularListConcat(L, M)$ che in tempo costante $\Theta(1)$ concatena le liste L e M e ritorna un puntatore alla lista concatenata.*



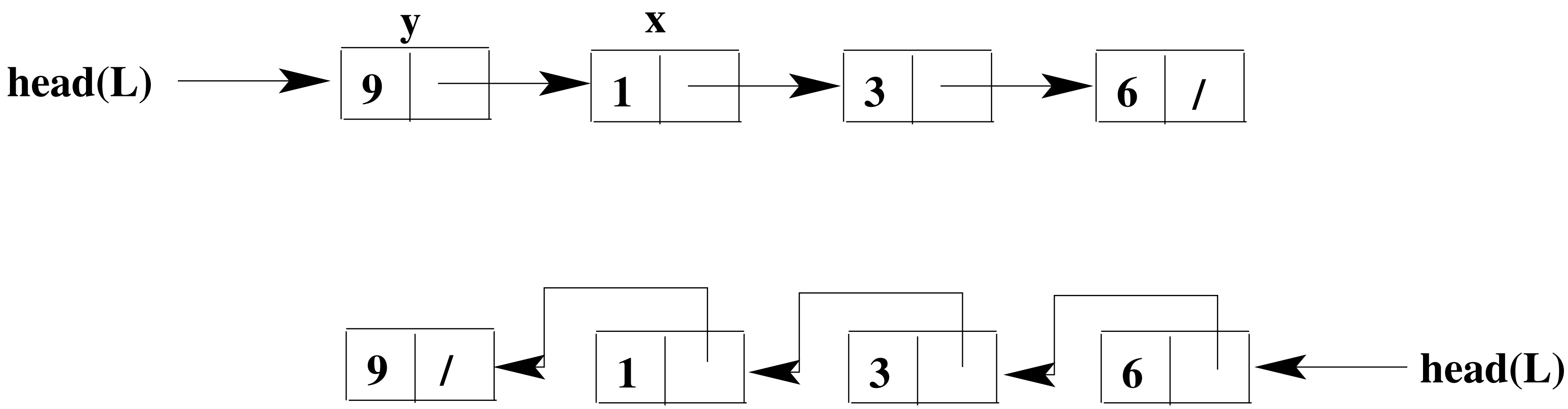
CircularListConcat(L,M)

if $ListEmpty(L)$ **then**
    return $head(M)$
end if
if $ListEmpty(M)$ **then**
    return $head(L)$
end if
$a \leftarrow head(L)$
$b \leftarrow prev[head(L)]$
$c \leftarrow head(M)$
$d \leftarrow prev[head(M)]$
$next[b] \leftarrow c$
$prev[c] \leftarrow b$
$next[d] \leftarrow a$
$prev[a] \leftarrow d$
return a

Lista singola



Esercizio *Sia L una lista singola di lunghezza n. Si scriva una procedura $SingleListReverse(L)$ che inverte in tempo $O(n)$ la sequenza contenuta in L.*



Esercizio *Sia L una lista singola di lunghezza n. Si scriva una procedura $SingleListReverse(L)$ che inverte in tempo $O(n)$ la sequenza contenuta in L.*

SingleListReverse(L)

$y \leftarrow \text{NIL}$
$x \leftarrow head(L)$
while $x \neq \text{NIL}$ **do**
    $z \leftarrow next[x]$
    $next[x] \leftarrow y$
    $y \leftarrow x$
    $x \leftarrow z$
end while
$head(L) \leftarrow y$

Esercizio *Implementare le strutture di dati pila e coda usando le liste (invece dei vettori).*

StackEmpty(L)

return $ListEmpty(L)$

StackFull(L)

return FALSE

Push(L,x)

$ListInsert(L, x)$

Pop(L)

if $ListEmpty(L)$ **then**
    error UNDERFLOW
end if
$h \leftarrow head(L)$
$ListDelete(L, h)$
return h

QueueEmpty(L)

return $CircularListEmpty(L)$

QueueFull(L)

return FALSE

Enqueue(L,x)

$M \leftarrow \emptyset$
$CircularListInsert(M, x)$
$CircularListConcat(L, M)$

Dequeue(L)

if $CircularListEmpty(L)$ **then**
  error UNDERFLOW
end if
$h \leftarrow head(L)$
$CircularListDelete(L, h)$
return h