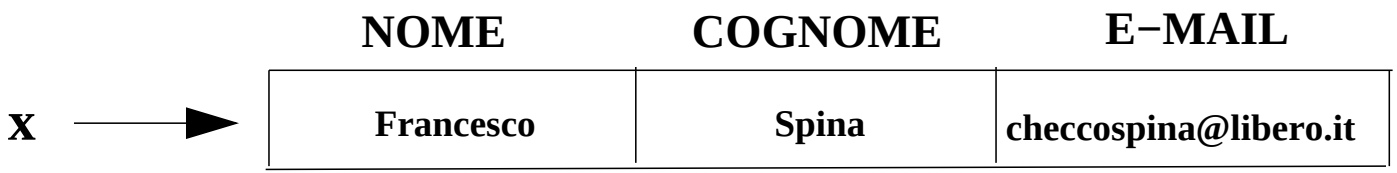
Algoritmi e Strutture di Dati I

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

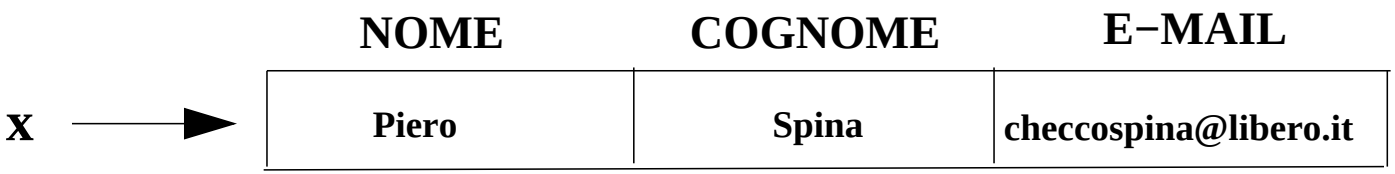
Oggetti e puntatori

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

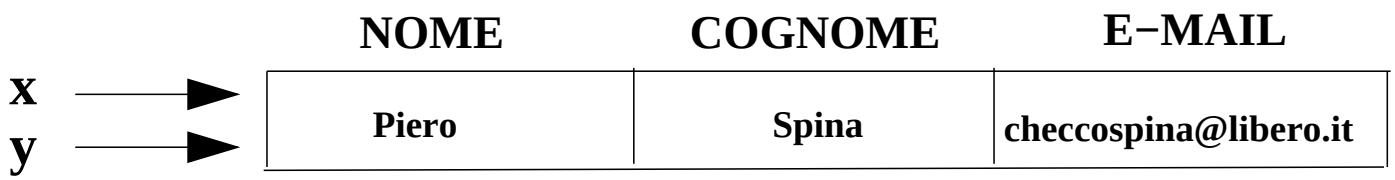


Per accedere ad un oggetto, devo conoscere il suo **indirizzo di memoria**. Un **puntatore** è una variabile che contiene un indirizzo di memoria.

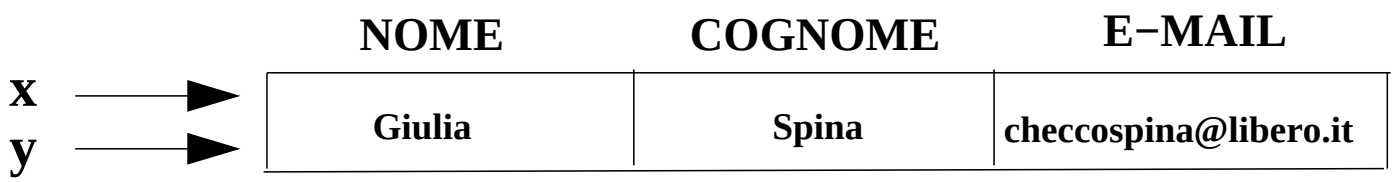
$$\text{NOME}[x] \leftarrow Piero$$



$$y \leftarrow x$$



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



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

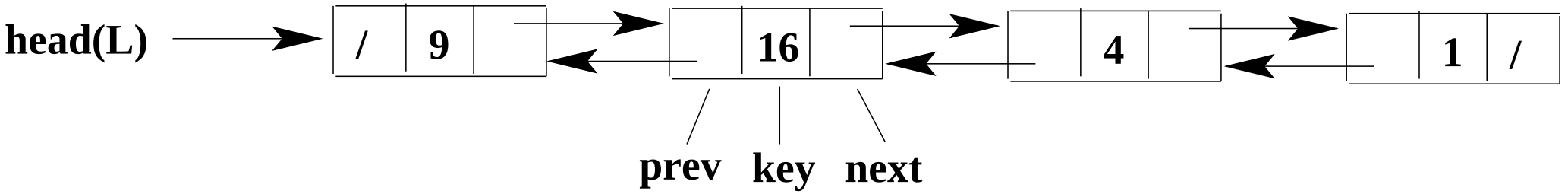
Useremo un modello di lista chiamato **lista doppia** o **lista bidirezionale**. Gli oggetti contenuti in una lista doppia hanno tre campi: *key*, *next* e *prev*. Sia x un oggetto della lista:

- **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)

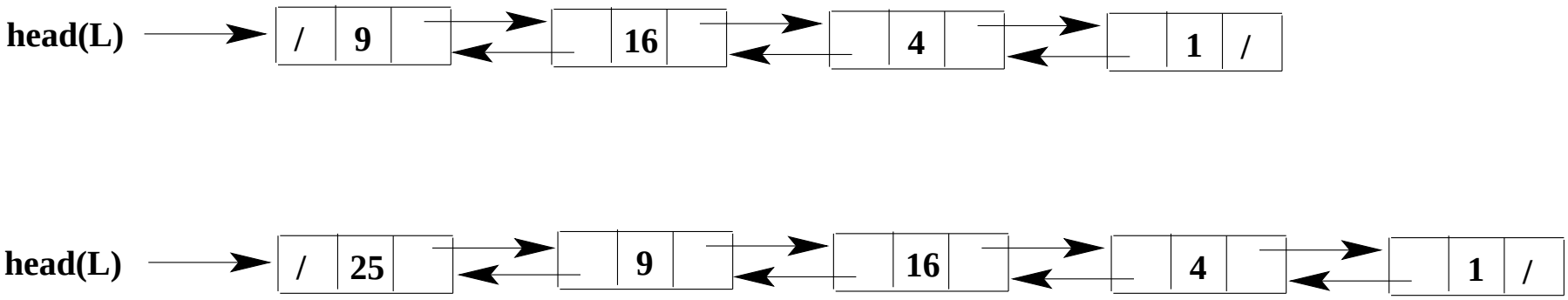
```
1: if head(L) = NIL then
2:     return TRUE
3: else
4:     return FALSE
5: end if
```

ListSearch(L,k)

```
1: x ← head(L)
2: while x ≠ NIL and key[x] ≠ k do
3:     x ← next[x]
4: end while
5: return x
```

Inserimento

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



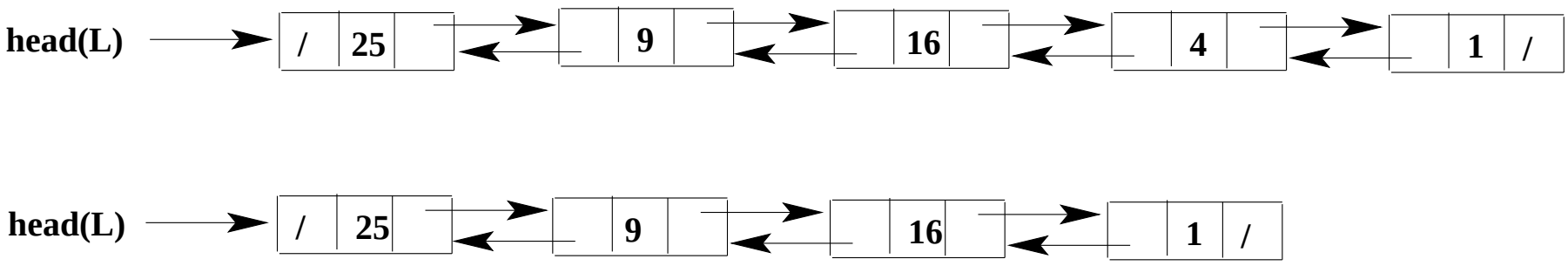
Inserimento

ListInsert(L,x)

1: $next[x] \leftarrow head(L)$
2: **if** $head(L) \neq$ NIL **then**
3: $\quad prev[head(L)] \leftarrow x$
4: **end if**
5: $head(L) \leftarrow x$
6: $prev[x] \leftarrow$ NIL

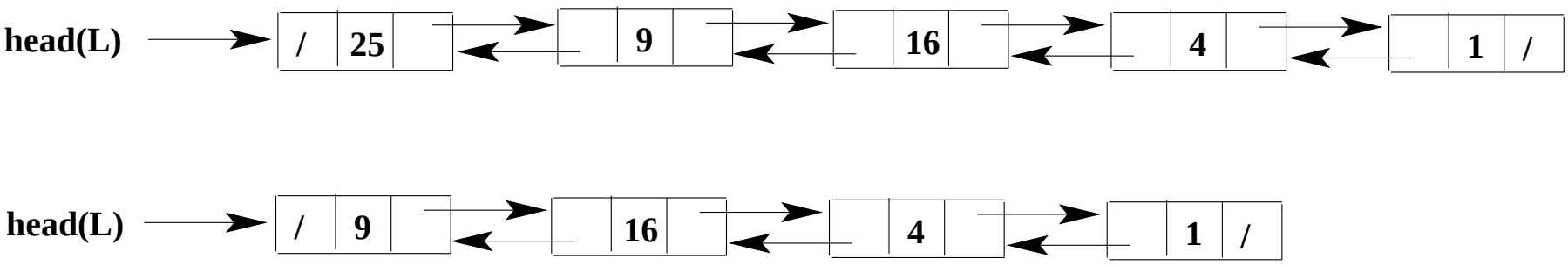
Cancellazione

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



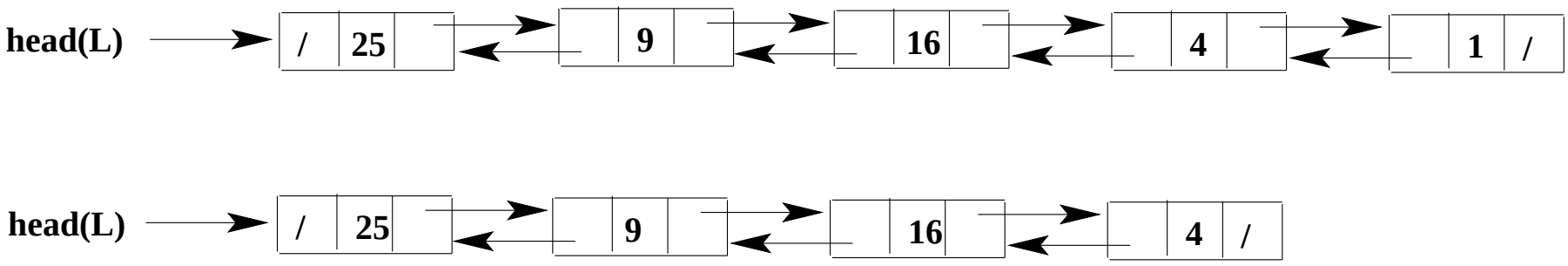
Cancellazione

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)

1: **if** $prev[x] \neq$ NIL **then**
2: $\quad next[prev[x]] \leftarrow next[x]$
3: **else**
4: $\quad head(L) \leftarrow next[x]$
5: **end if**
6: **if** $next[x] \neq$ NIL **then**
7: $\quad prev[next[x]] \leftarrow prev[x]$
8: **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.*

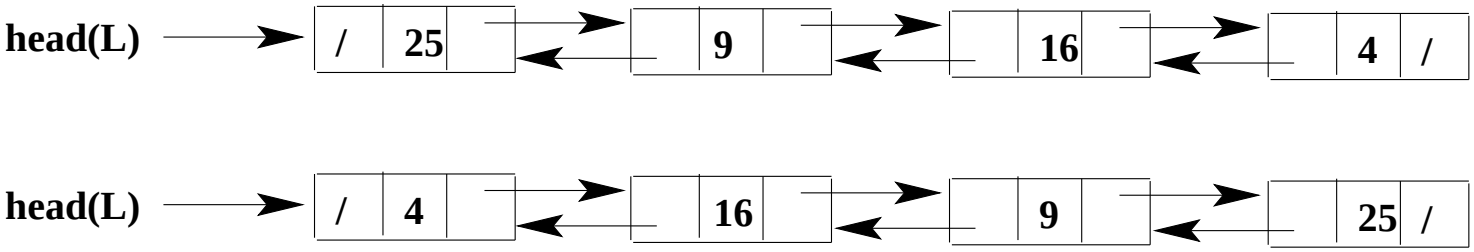
Esercizio *Sia L una lista. Si scriva una procedura $ListLength(L)$ che ritorna la lunghezza della lista L.*

ListLength(L)

1: $x \leftarrow head(L)$
2: $len \leftarrow 0$
3: **while** $x \neq$ NIL **do**
4: $\quad x \leftarrow next[x]$
5: $\quad len \leftarrow len + 1$
6: **end while**
7: **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)

```
1: y ← NIL
2: x ← head(L)
3: while x ≠ NIL do
4:     t ← next[x]
5:     next[x] ← prev[x]
6:     prev[x] ← t
7:     y ← x
8:     x ← t
9: end while
10: head(L) ← y
```

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

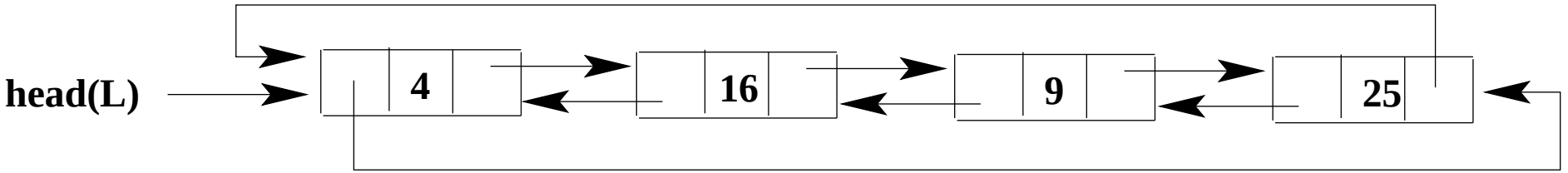
Esercizio *Siano L e M due liste. Si scriva una procedura $ListCompare(L, M)$ che confronta il contenuto di L e M.*

ListCompare(L,M)

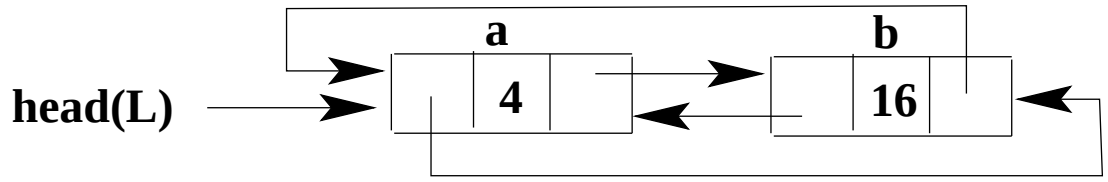
```
 1: x ← head(L)
 2: y ← head(M)
 3: while (x ≠ NIL) and (y ≠ NIL) do
 4:    if key[x] ≠ key[y] then
 5:       return FALSE
 6:    else
 7:       x ← next[x]
 8:       y ← next[y]
 9:    end if
10: end while
11: if (x = NIL) and (y = NIL) then
12:    return TRUE
13: else
14:    return FALSE
15: end if
```

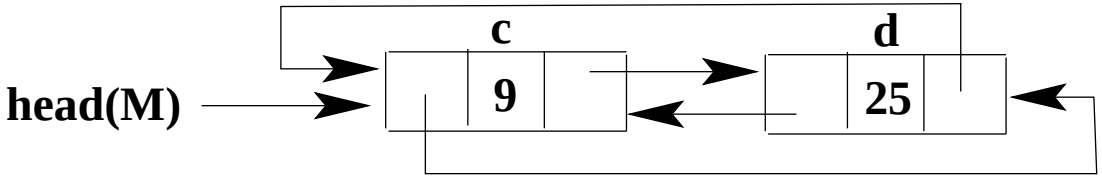
La complessità di $\Theta(min\{n, m\})$, con n è la lunghezza di L e m è la lunghezza di M.

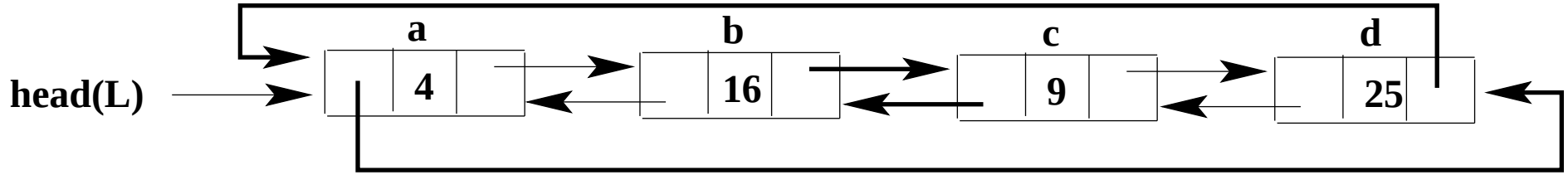
Lista circolare



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.*



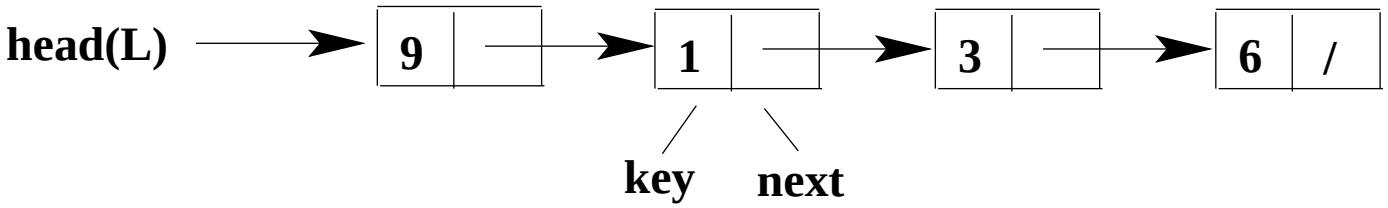




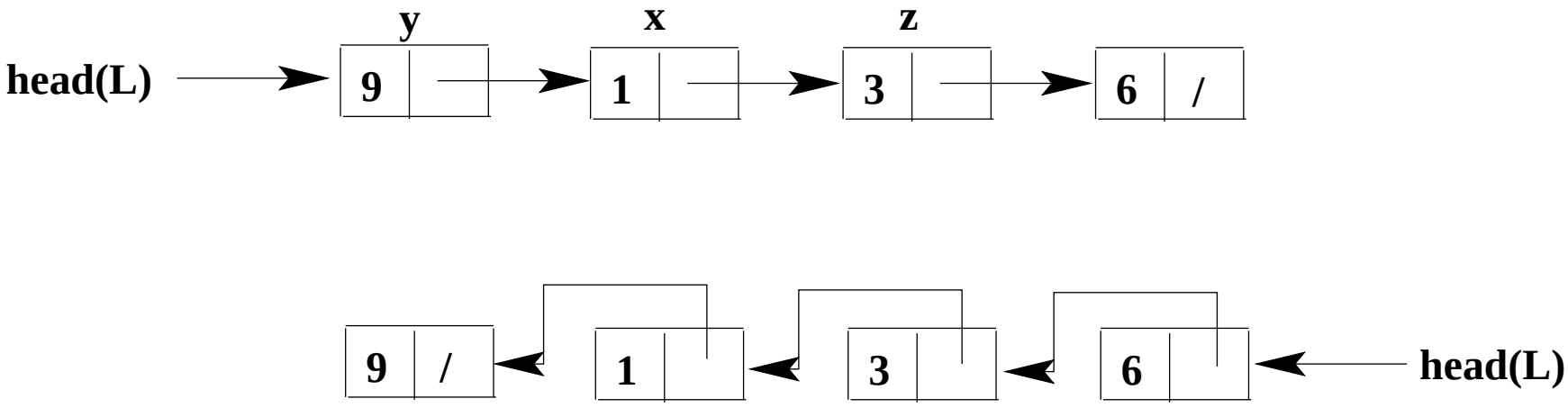
CircularListConcat(L,M)

```
1: if ListEmpty(L) then
2:     return head(M)
3: end if
4: if ListEmpty(M) then
5:     return head(L)
6: end if
7: a ← head(L)
8: b ← prev[head(L)]
9: c ← head(M)
10: d ← prev[head(M)]
11: next[b] ← c
12: prev[c] ← b
13: next[d] ← a
14: prev[a] ← d
15: 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)

1: $y \leftarrow$ NIL
2: $x \leftarrow head(L)$
3: **while** $x \neq$ NIL **do**
4: $\quad z \leftarrow next[x]$
5: $\quad next[x] \leftarrow y$
6: $\quad y \leftarrow x$
7: $\quad x \leftarrow z$
8: **end while**
9: $head(L) \leftarrow y$

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

StackEmpty(L)

1: **return** $ListEmpty(L)$

StackFull(L)

1: **return** FALSE

Push(L,x)

1: $ListInsert(L, x)$

Pop(L)

1: **if** $ListEmpty(L)$ **then**
2: **error** UNDERFLOW
3: **end if**
4: $h \leftarrow head(L)$
5: $ListDelete(L, h)$
6: **return** h

QueueEmpty(L)

1: **return** $CircularListEmpty(L)$

QueueFull(L)

1: **return** FALSE

Enqueue(L,x)

1: $M \leftarrow \emptyset$
2: $CircularListInsert(M, x)$
3: $CircularListConcat(L, M)$

Dequeue(L)

1: **if** $CircularListEmpty(L)$ **then**
2: **error** UNDERFLOW
3: **end if**
4: $h \leftarrow head(L)$
5: $CircularListDelete(L, h)$
6: **return** h