

Algoritmi e Strutture Dati

Strutture Dati Elementari

Insiemi

- Un *insieme* è una *collezione di oggetti* distinguibili chiamati *elementi* (o membri) dell'insieme.
- $a \in S$ significa che a è *un membro* de (o *appartiene* a) *l'insieme S*
- $b \notin S$ significa che b **NON** è un *membro* de (o *NON appartiene* a) *l'insieme S*
- **Esempi:**
 - \mathbb{I} denota l'insieme dei *numeri interi*
 - \mathbb{N} denota l'insieme dei *numeri naturali*
 - \mathbb{R} denota l'insieme dei *numeri reali*
 - \emptyset denota l'*insieme vuoto*

Insiemi Dinamici

- Gli *algoritmi* manipolano *collezioni di dati* come insiemi di elementi
- Gli insiemi rappresentati e manipolati da algoritmi in generale cambiano nel tempo:
 - *crescono in dimensione* (cioè nel numero di elementi che contengono)
 - *diminuiscono in dimensione*
 - *la collezione di elementi che contengono può mutare nel tempo*

Per questo vengono chiamati *Insiemi Dinamici*

Insiemi Dinamici

Spesso gli *elementi* di un insieme dinamico sono oggetti strutturati che contengono

- **una “*chiave*” identificativa *k* dell’elemento all’interno dell’insieme**
- **altri “*dati satellite*”, contenuti in opportuni campi di cui sono costituiti gli elementi dell’insieme**

I dati satellite non vengono in genere direttamente usati per implementare le operazioni sull’insieme.

Operazioni su Insiemi Dinamici

Esempi di operazioni su insiemi dinamici

➤ *Operazioni di Ricerca:*

- *Ricerca(S,k):*
- *Minimo(S):*
- *Massimo(S):*
- *Successore(S,x):*
- *Predecessore(S,x):*

Operazioni su Insiemi Dinamici

Esempi di operazioni su insiemi dinamici

➤ *Operazioni di Modifica:*

- *Inserimento(S,x):*
- *Cancellazione(S,x):*

Stack

Uno **Stack** è un insieme dinamico in cui l'elemento rimosso dall'operazione di **cancellazione** è predeterminato.

In uno **Stack** questo elemento è l'**ultimo elemento** inserito.

Uno **Stack** implementa una **lista** di tipo “**last in, first out**” (**LIFO**)

- **Nuovi elementi** vengono inseriti in **testa** e prelevati **dalla testa**

Operazioni su Stack

Due Operazioni di Modifica:

Inserimento: *Push*(*S*, *x*)

- aggiunge un elemento in cima allo Stack

Cancellazione: *Pop*(*S*)

- rimuove un elemento dalla cima dello Stack

Altre operazioni: *Stack-Vuoto*(*S*)

- verifica se lo Stack è vuoto (ritorna *True* o *False*)

Operazioni su Stack

Due Operazioni di Modifica:

Inserimento: $Push(S, x)$

- aggiunge un elemento in cima allo Stack

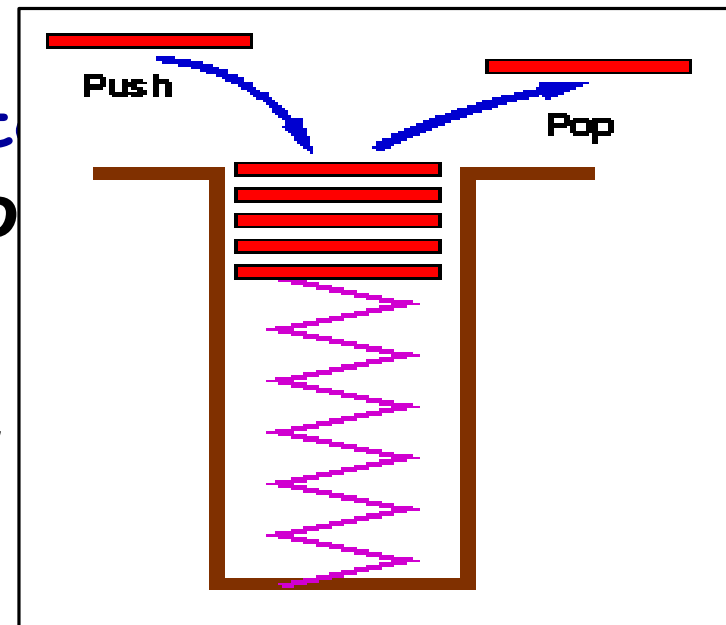
Cancellazione: $Pop(S)$

- rimuove un elemento dalla cima dello Stack

Altre operazioni: $Stack-Vuoto$

- verifica se lo Stack è vuoto
(*False*)

Uno **Stack** può essere immaginato come una **pila di piatti!**



Operazioni su Stack

```
Algoritmo Stack-Vuoto (S)
  IF top[S] = 0
    THEN return TRUE
    ELSE return FALSE
```

*top[S]: un intero che rappresenta, in ogni istante, il **numero di elementi presenti** nello Stack*

Operazioni su Stack

```
Algoritmo Stack-Vuoto (S)
  IF  $top[S] = 0$ 
    THEN return TRUE
    ELSE return FALSE
```

```
Algoritmo Push (S, x)
   $top[S] = top[S] + 1$ 
   $S[top[S]] = x$ 
```

Assumiamo qui che l'operazione di *aggiunta* di un elemento nello Stack S sia realizzata come l'aggiunta di un elemento ad un array

Operazioni su Stack

- **Problema:**
 - Che succede se eseguiamo un operazione di **pop** (estrazione) di un elemento **quando lo Stack è vuoto?**
 - Questo è chiamato **Stack Underflow**. É necessario implementare l'operazione di **pop** con un meccanismo per verificare se questo è il caso.

Operazioni su Stack

```
Algoritmo Stack-Vuoto (S)
```

```
  IF  $top[S] = 0$ 
```

```
    THEN return TRUE
```

```
    ELSE return FALSE
```

```
Algoritmo Push (S, x)
```

```
   $top[S] = top[S] + 1$ 
```

```
   $S[top[S]] = x$ 
```

```
Algoritmo Pop (S)
```

```
  IF Stack-Vuoto (S)
```

```
    THEN ERROR "underflow"
```

```
    ELSE  $top[S] = top[S] - 1$ 
```

```
      return  $S[top[S] + 1]$ 
```

Stack: implementazione

- **Problema:**
 - Che succede se eseguiamo un operazione di `push` (inserimento) di un elemento **quando lo Stack è pieno?**
 - Questo è chiamato **Stack Overflow**. É necessario implementare l'operazione di `push` con un meccanismo per verificare se questo è il caso. (**SEMPLICE ESERCIZIO**)

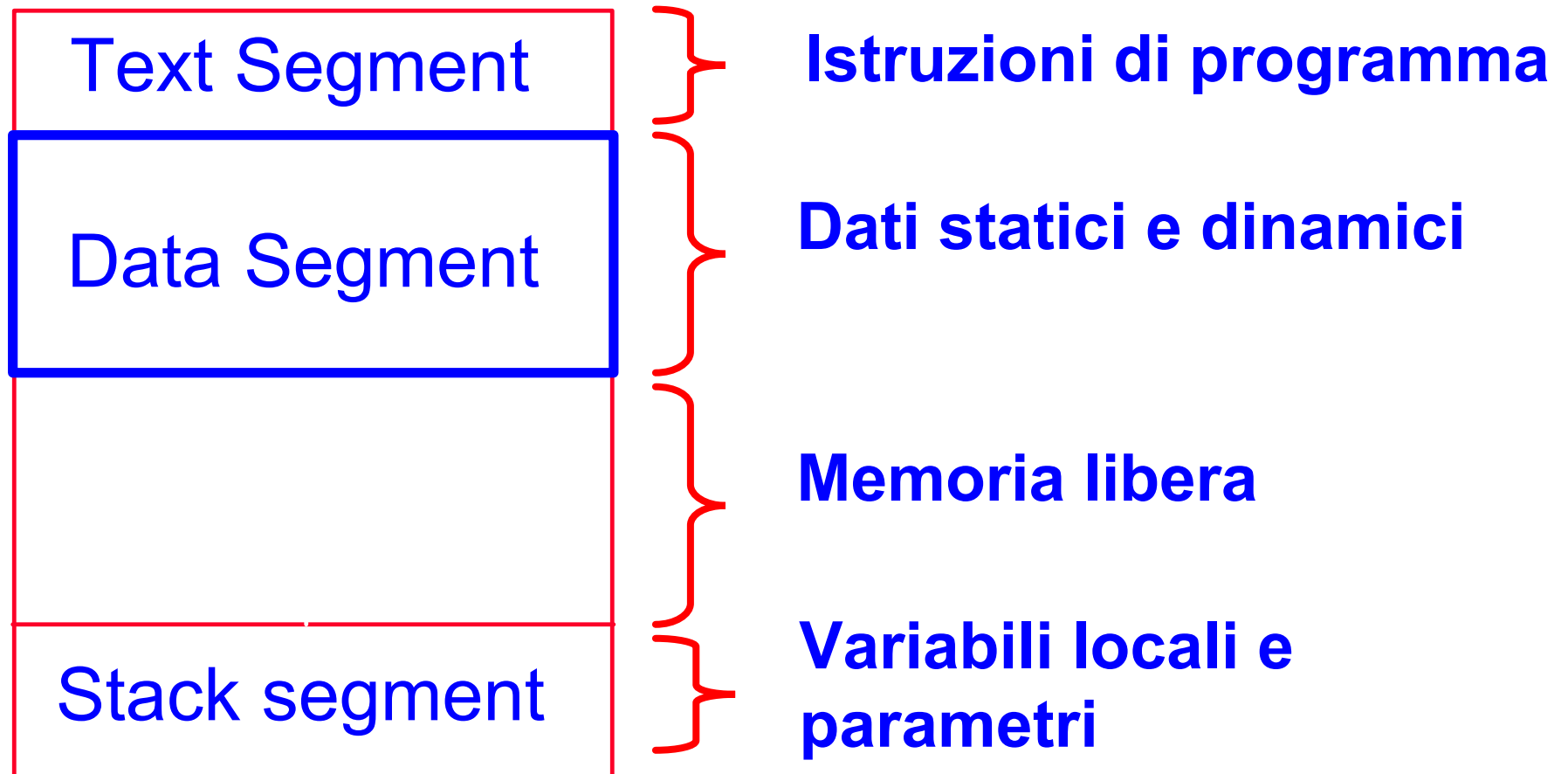
Stack: implementazione

- **Arrays**
 - **Permettono di implementare stack in modo semplice**
 - **Flessibilità limitata, *ma* incontra parecchi casi di utilizzo**
 - **La capacità dello Stack è limitata ad una quantità costante:**
 - **dalla memoria del computer**
 - **dalla dimensione della pila, etc**
- **Possibile implementarle con Liste Puntate.**

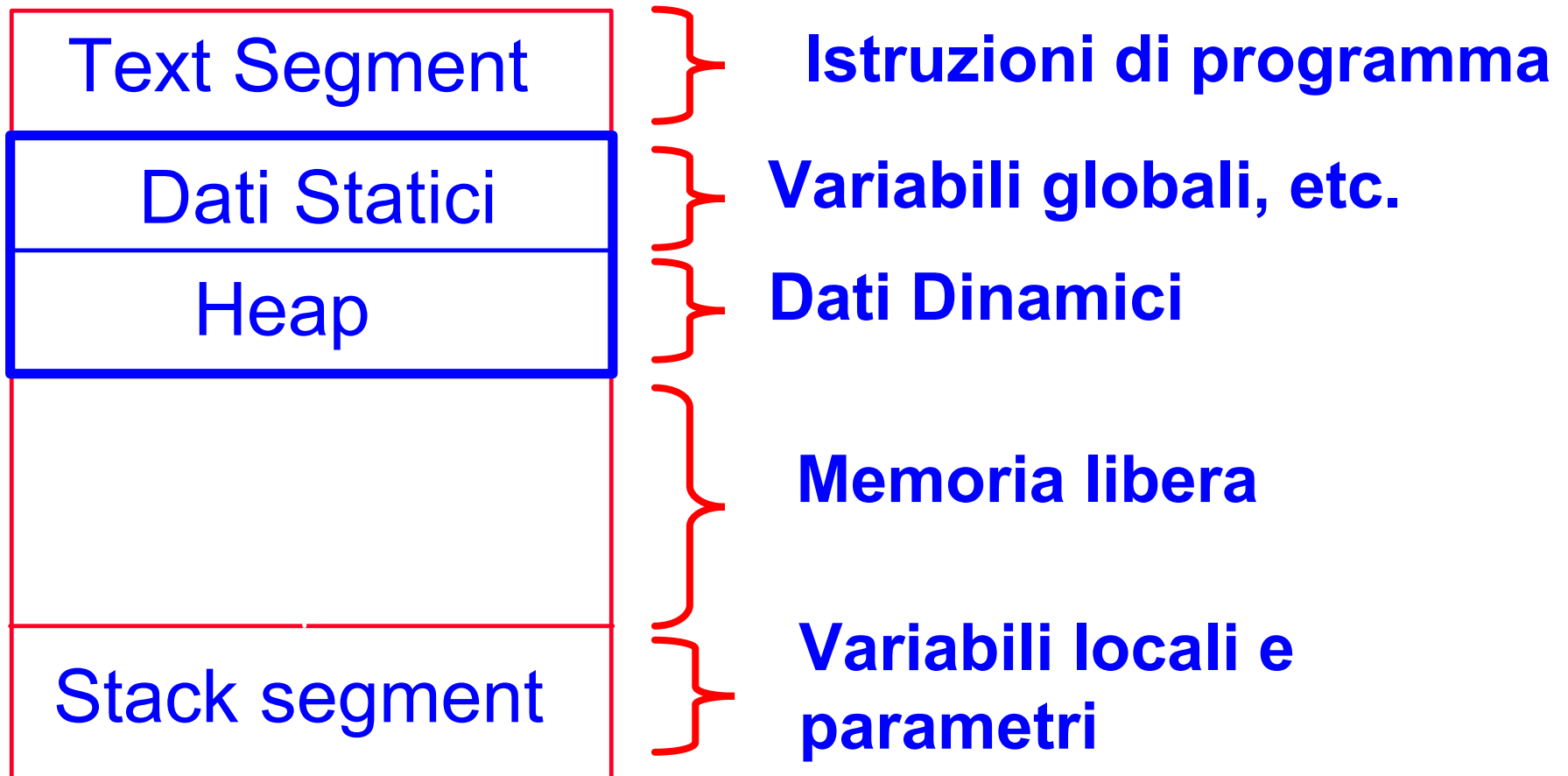
Stack: implementazione

- **Stacks sono molto frequenti in Informatica:**
 - Elemento chiave nel meccanismo che implementa la **chiamata/return** a funzioni/procedure
 - **Record di attivazione** permettono la ricorsione.
 - Chiamata: *push* di un record di attivazione
 - Return: *pop* di un record di attivazione

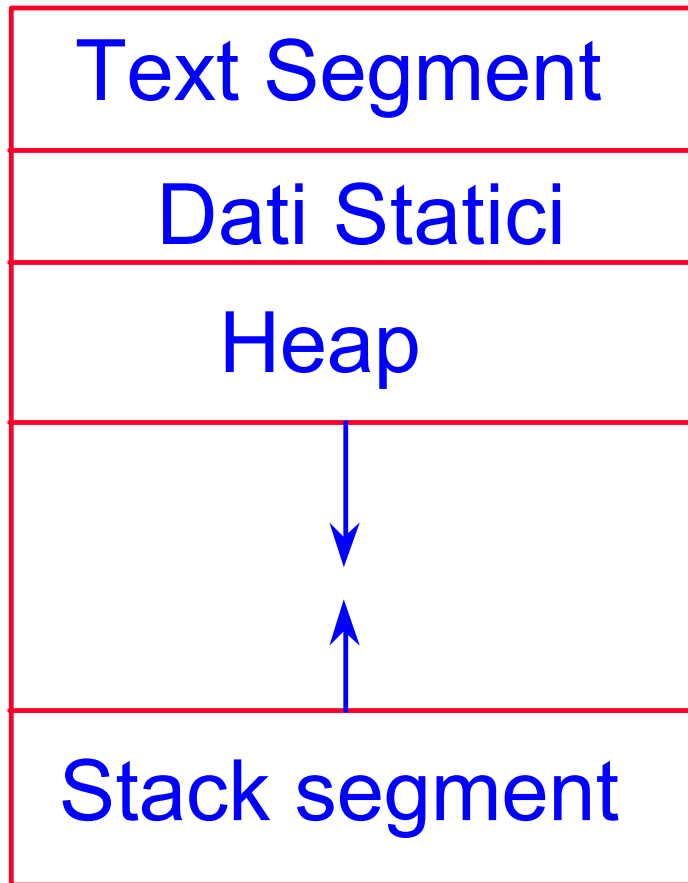
Gestione della memoria dei processi



Gestione della memoria dei processi



Gestione della memoria dei processi



**La memoria è allocata
e deallocata secondo
necessità**

Stack: applicazioni

- **Stacks sono molto frequenti:**
 - Elemento chiave nel meccanismo che implementa la **chiamata/return** a funzioni/procedure
 - **Record di attivazione** permettono la ricorsione.
 - Chiamata: *push* di un record di attivazione
 - Return: *pop* di un record di attivazione
- **Record di Attivazione contiene**
 - Argomenti (parametri) di funzioni
 - Indirizzo di ritorno
 - Valore di ritorno
 - Variabili locali della funzione

Stack di Record di Attivazione in LP

Programma

```
function f(int x,int y)
{
    int a;
    if ( term_cond )
        return ...;
    a = ...;
    return g( a );
}
```

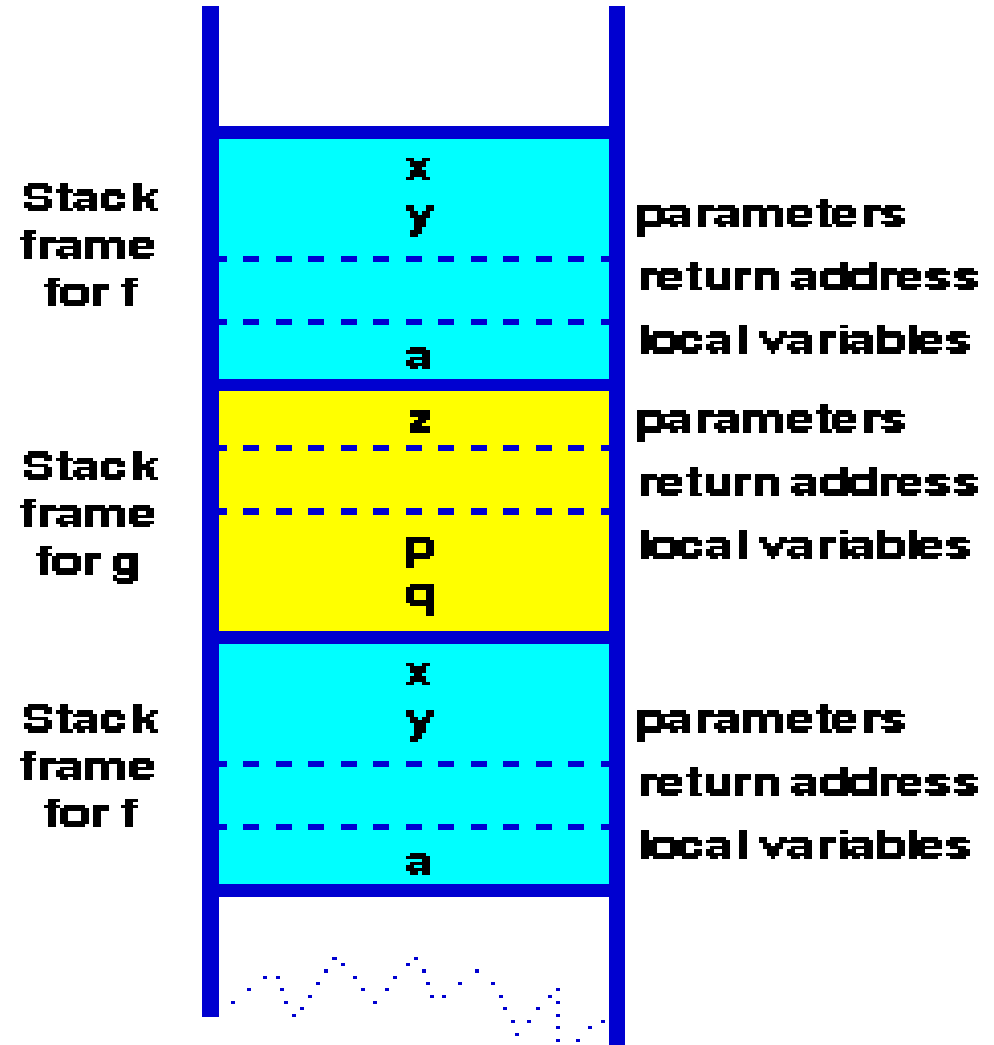
```
function g( int z )
{
    int p, q;
    p = ... ; q = ... ;
    return f(p,q);
}
```

Stack di Record di Attivazione in LP

Programma

```
function f(int x,int y)
{
  int a;
  if ( term_cond )
    return ...;
  a = ...;
  return g( a );
}
```

```
function g( int z )
{
  int p, q;
  p = ... ; q = ... ;
  return f(p,q);
}
```

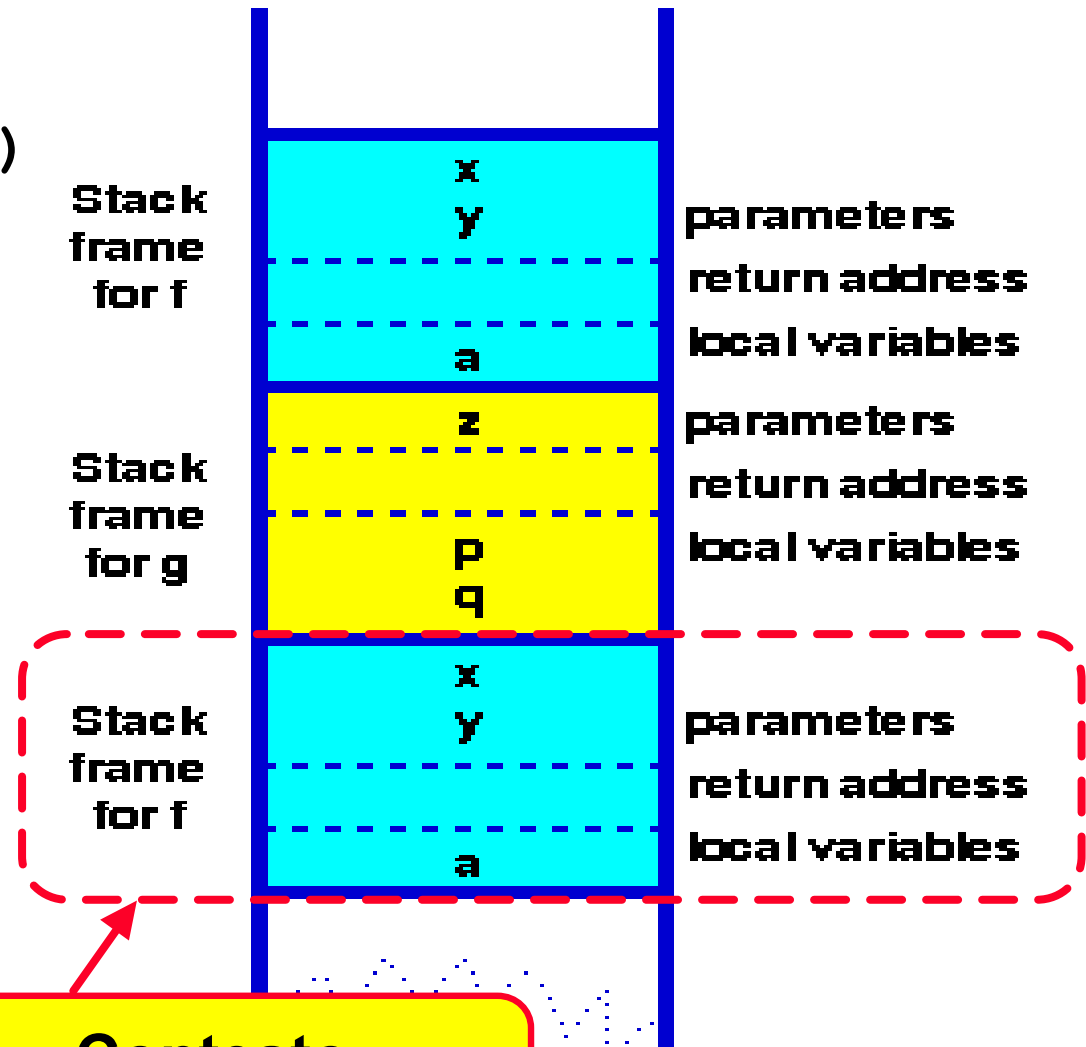


Stack di Record di Attivazione in LP

Programma

```
function f(int x,int y)
{
  int a;
  if ( term_cond )
    return ...;
  a = ...;
  return g( a );
}
```

```
function g( int z )
{
  int p, q;
  p = ... ; q = ... ;
  return f(p,q);
}
```



Contesto di esecuzione di f

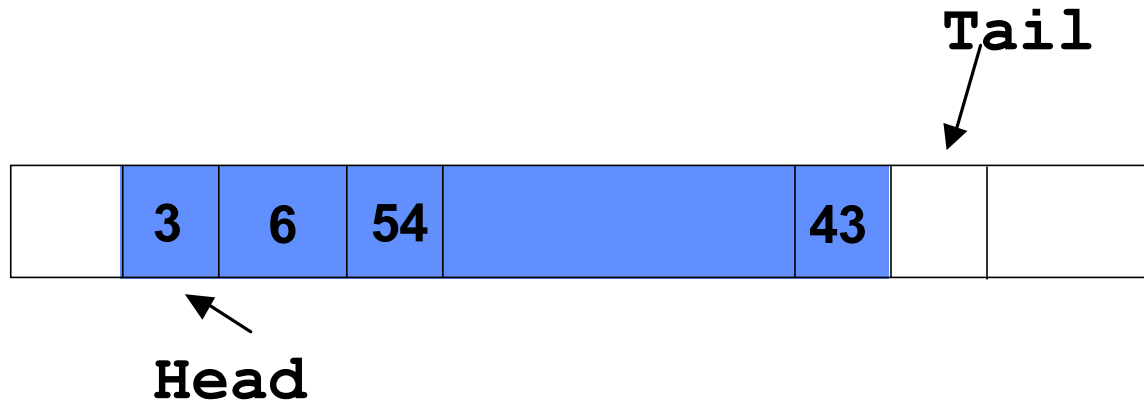
Code

Una Coda è un insieme dinamico in cui l'elemento rimosso dall'operazione di **cancellazione** è predeterminato.

In una **Coda** questo elemento è l'elemento che per più tempo è rimasto nell'insieme.

Una **Coda** implementa una lista di tipo “**first in, first out**” (**FIFO**)

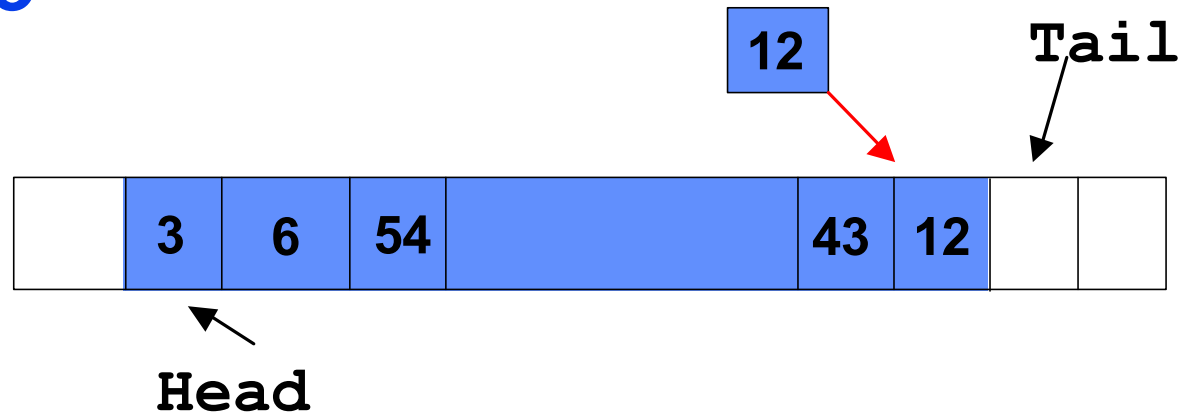
Code



Una **Coda** implementa una lista di tipo “**first in, first out**” (**FIFO**)

- Possiede una testa (**Head**) ed una coda (**Tail**)

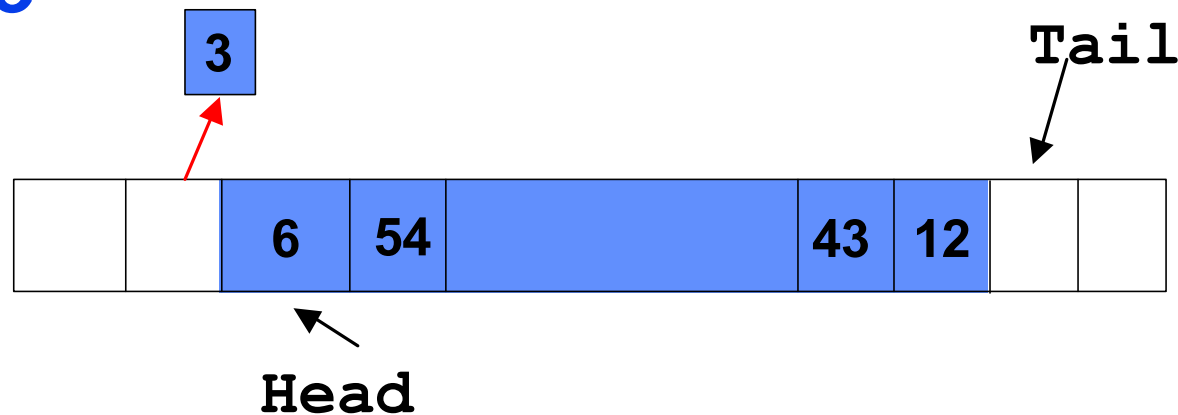
Code



Una **Coda** implementa una lista di tipo “**first in, first out**” (**FIFO**)

- Possiede una testa (**Head**) ed una coda (**Tail**)
- Quando si **aggiunge** un elemento, viene inserito **al posto della coda**

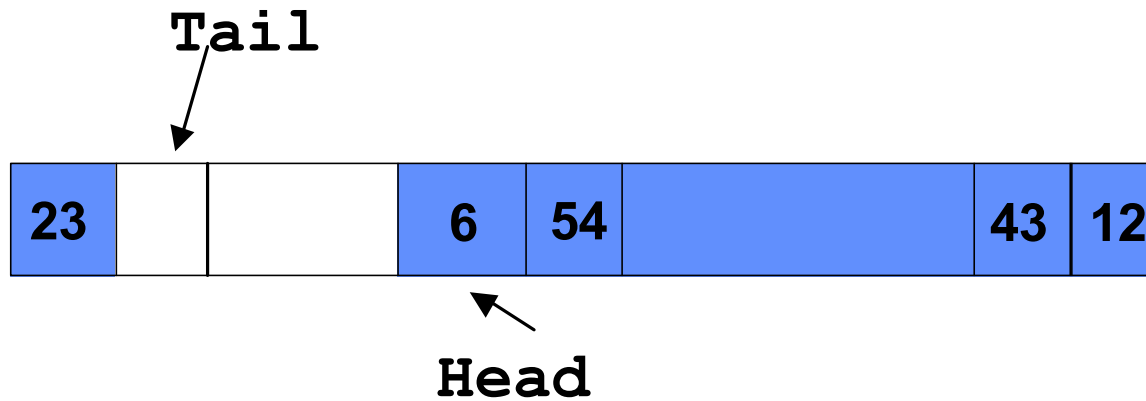
Code



Una **Coda** implementa una lista di tipo “**first in, first out**” (**FIFO**)

- Possiede una testa (**Head**) ed una coda (**Tail**)
- Quando si **aggiunge** un elemento, viene inserito **al posto della coda**
- Quando si **estrae** un elemento, viene estratto **dalla testa**

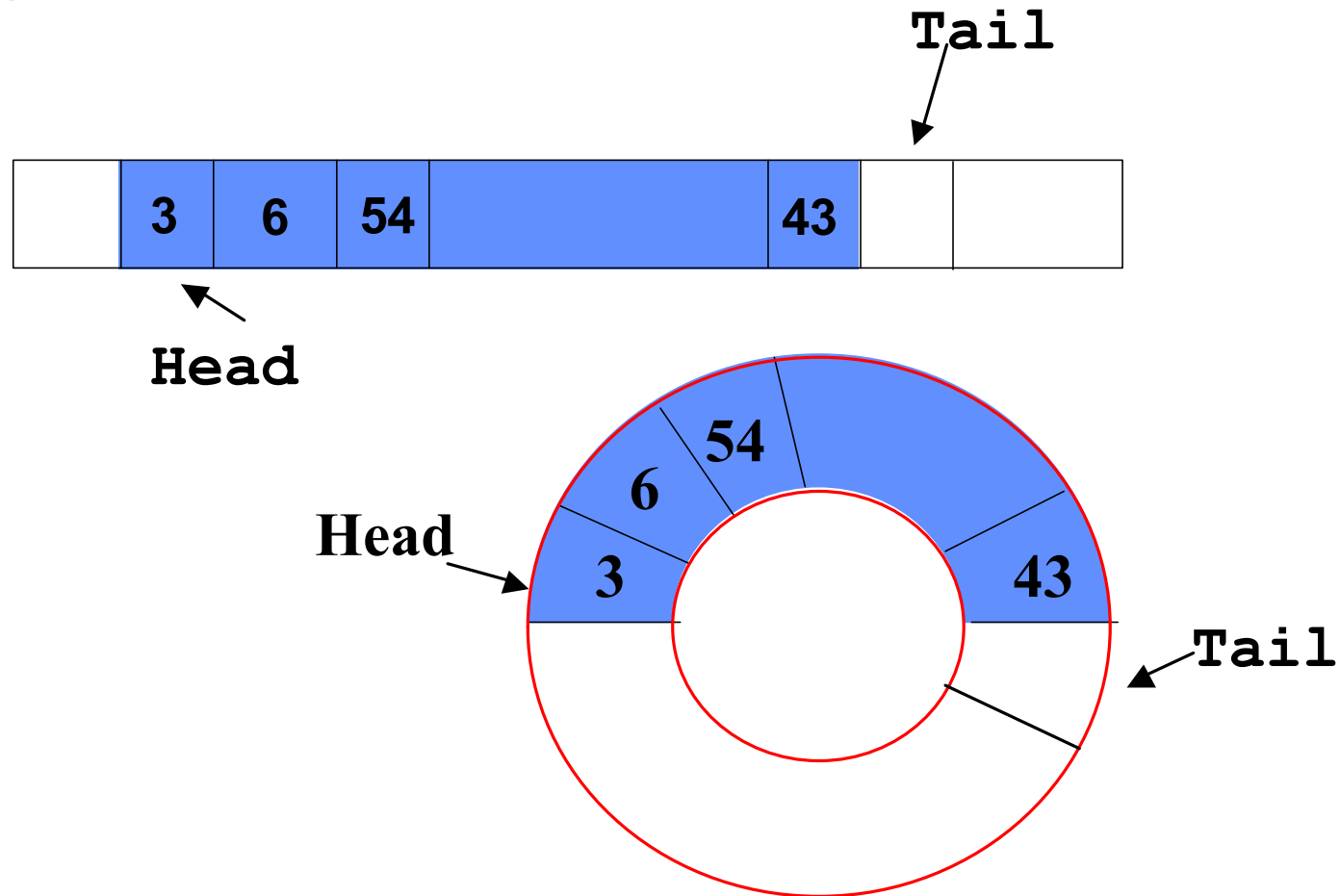
Code



Una **Coda** implementa una lista di tipo “**first in, first out**” (**FIFO**)

- La “**finestra**” dell’array occupata dalla **coda** si sposta lungo l’array!

Code



La “**finestra**” dell’array occupata dalla **coda** si sposta lungo l’array!

Array Circolare
implementato ad esempio
con una operazione di
modulo

Operazioni su Code

```
Algoritmo Accoda(Q, x)
  Q[Tail[Q]] = x
  IF Tail[Q] = Length[Q]
    THEN Tail[Q] = 1
    ELSE Tail[Q] = Tail[Q] + 1
```

Operazioni su Code

```
Algoritmo Accoda (Q, x)
  Q[Tail[Q]] = x
  IF Tail[Q] = Length[Q]
    THEN Tail[Q] = 1
    ELSE Tail[Q] = Tail[Q] + 1
```

```
Algoritmo Estrai-da-Coda (Q)
  x = Q[Head[Q]]
  IF Head[Q] = Length[Q]
    THEN Head[Q] = 1
    ELSE Head[Q] = Head[Q] + 1
  return x
```

Operazioni su Code: con modulo

```
Algoritmo Accoda (Q, x)
```

```
  Q[Tail[Q]] = x
```

```
  Tail[Q] = (Tail[Q] + 1) mod Length[Q]
```

```
Algoritmo Estrai-da-Coda (Q)
```

```
  x = Q[Head[Q]]
```

```
  Head[Q] = Head[Q] + 1 mod Length[Q]
```

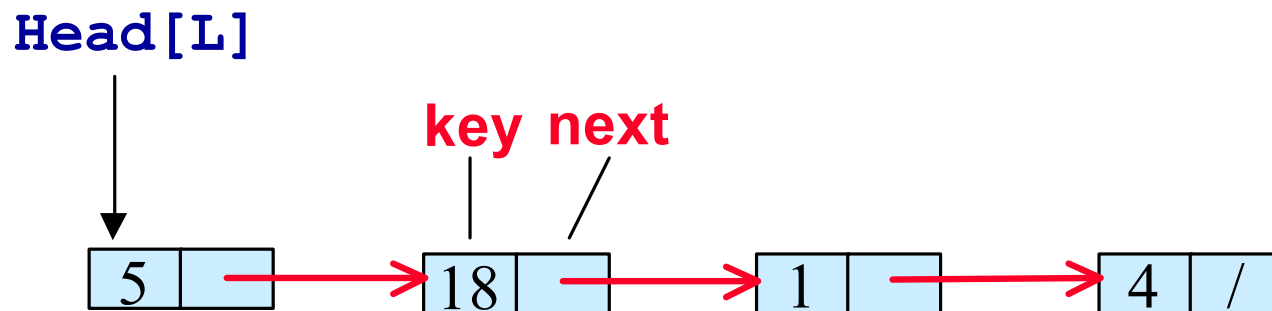
```
  return x
```

Mancano anche qui le verifiche del caso in cui la coda sia piena e/o vuota. (ESERCIZIO)

Liste Puntate

Una Liste Puntata è un insieme dinamico in cui ogni elemento ha una chiave (*key*) ed un riferimento all'elemento successivo (*next*) dell'insieme.

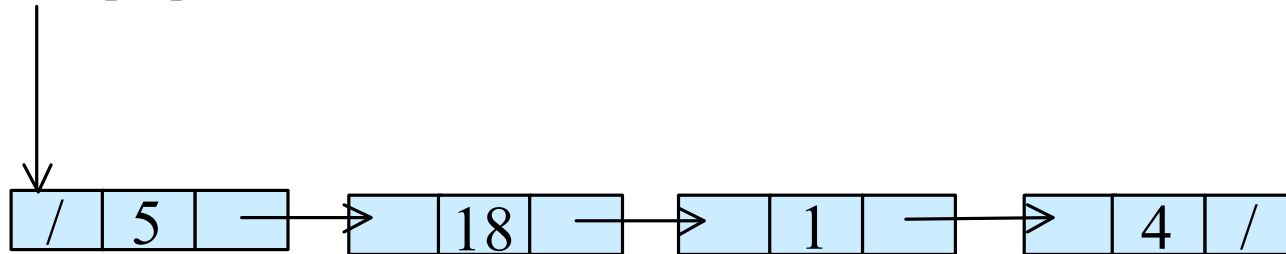
È una struttura dati ad accesso strettamente sequenziale!



Operazioni su Liste Puntate Doppie

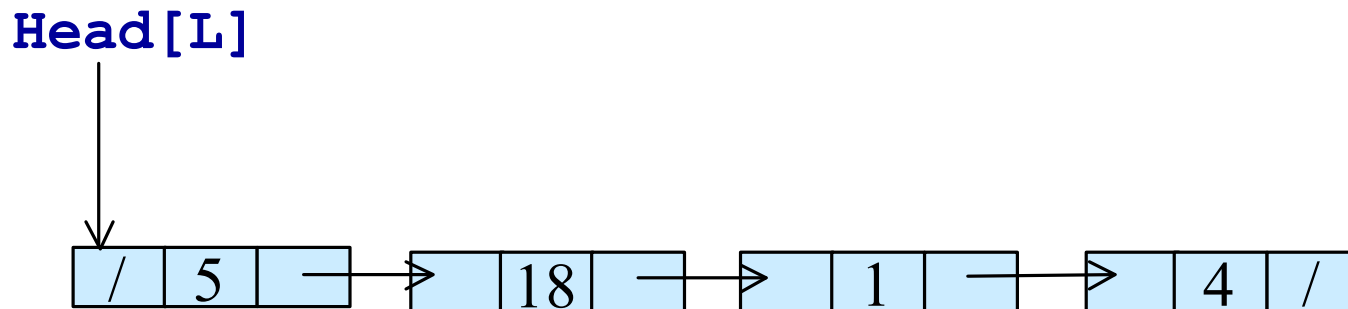
```
algoritmo Lista-Cerca-ric(L, k)
  IF L 1 NIL and key[L] 1 k THEN
    return Lista-Cerca-ric(next[L], k)
  return L
```

Head[L]



Operazioni su Liste Puntate Doppie

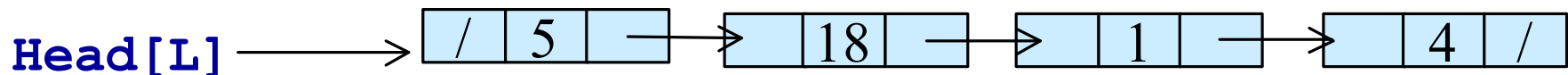
```
Algoritmo Lista-Inserisci(L, k)
  "alloca nodo new"
  key[new] = k
  next[new] = Head[L]
  Head[L] = new
```



Operazioni su Liste Puntate

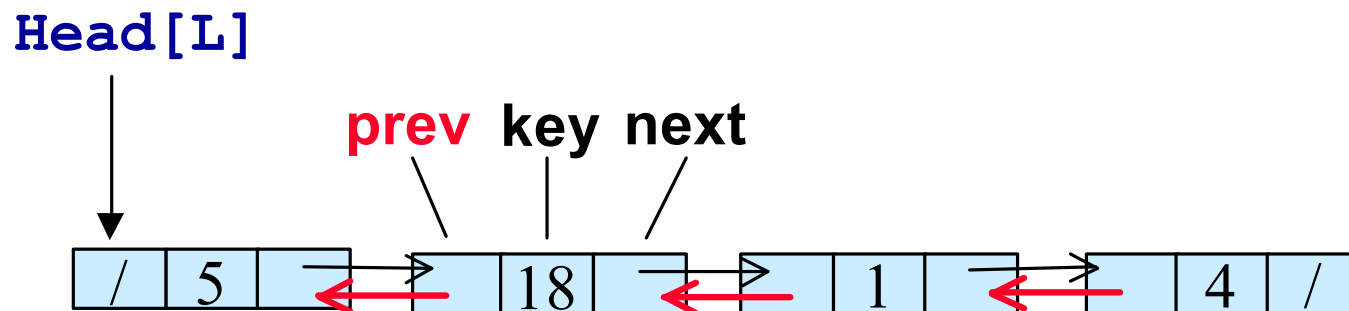
```
Algoritmo Lista-cancella-r(L, k)
  IF L 1 NIL THEN
    IF key[L] 1 k THEN
      next[L] = Lista-cancella-r(next[L], k)
    ELSE /* chiave k trovata in L */
      temp = L
      L = next[L]
      "dealloca temp"
  return L
```

```
Algoritmo Lista-cancella(L, k)
  Head[L] = Lista-cancella-r(Head[L], k)
```



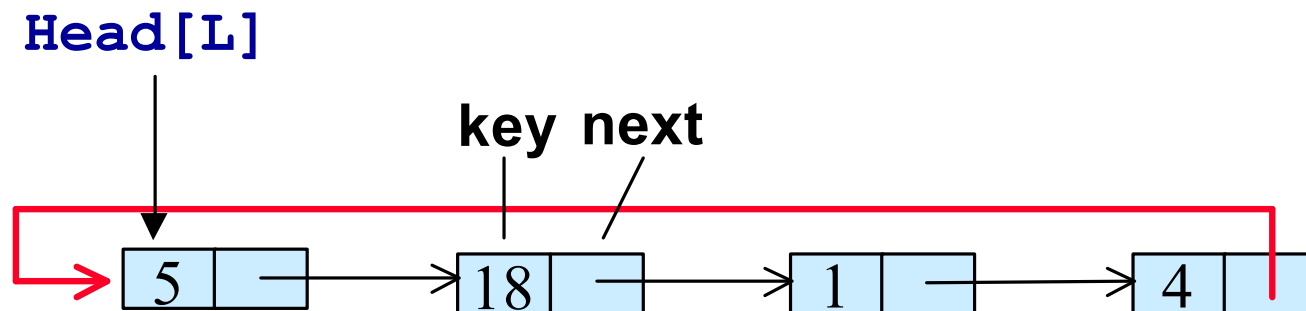
Liste Puntate Doppie

Una Lista Doppia Puntata è un insieme dinamico in cui in cui ogni elemento ha una chiave (*key*) e due riferimenti, uno all'elemento successivo (*next*) dell'insieme ed uno all'elemento precedente (*prev*) dell'insieme.



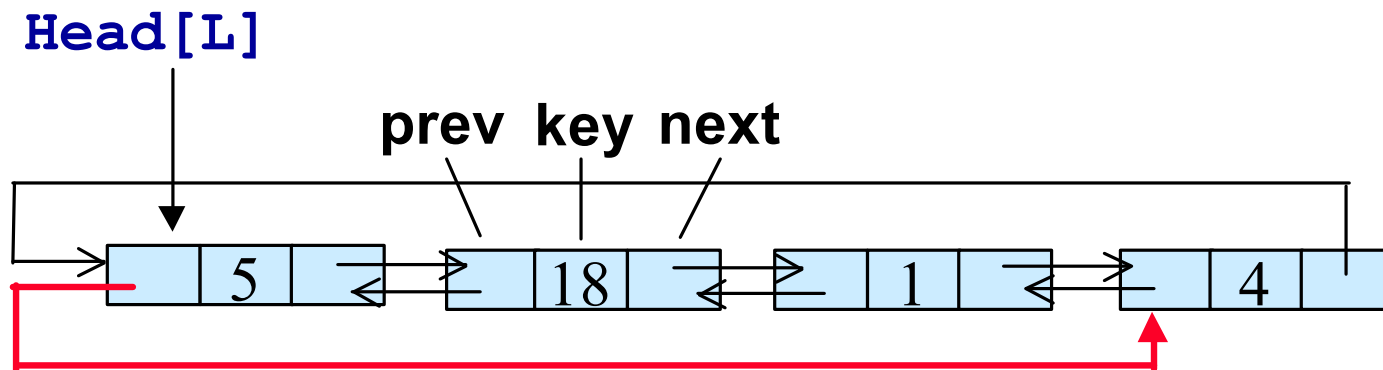
Liste Puntate Circolare

Una **Liste Circolare** puntata è un insieme dinamico in cui in cui ogni elemento ha una chiave (*key*) ed un riferimento all'elemento successivo (*next*) dell'insieme. **L'ultimo elemento ha un riferimento alla testa della lista**



Liste Puntate Circolare Doppia

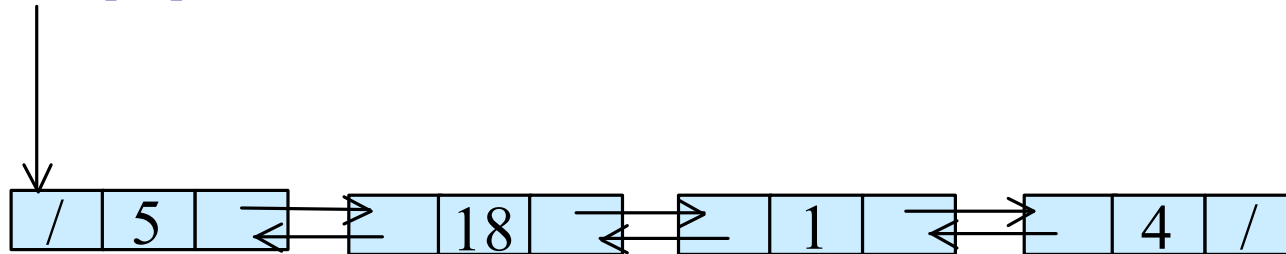
Una **Liste Circolare** puntata è un insieme dinamico in cui in cui ogni elemento ha una chiave (*key*) e due riferimenti, uno all'elemento successivo (*next*) dell'insieme ed uno all'elemento precedente (*prev*) dell'insieme. **L'ultimo elemento ha un riferimento (*prev*) alla testa della lista, il primo ha un riferimento (*next*) alla coda della lista**



Operazioni su Liste Puntate Doppie

```
Algoritmo Lista-cerca(L, k)
  x = Head[L]
  WHILE x ≠ NIL and key[x] ≠ k
    DO x = next[x]
  return x
```

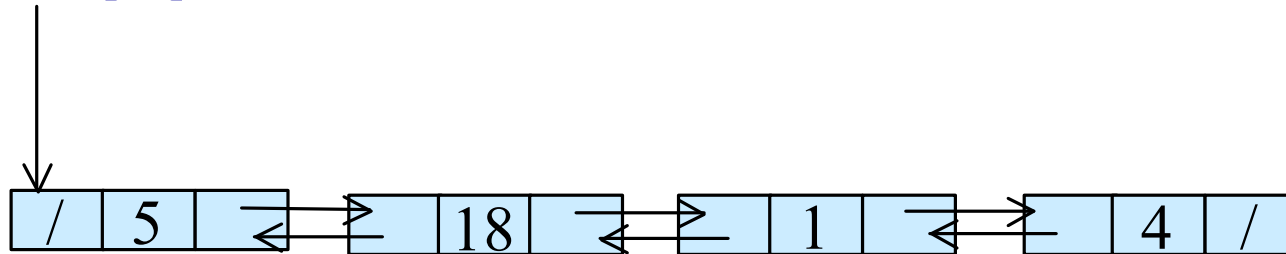
Head[L]



Operazioni su Liste Puntate

```
Algoritmo ListaD-Inserisci (L, k)
  "alloca nodo x"
  key[x]=k
  next[x]=Head[L]
  IF Head[L] ≠ NIL
    THEN prev[Head[L]]=x
  Head[L]=x
  prev[x]=NIL
```

Head[L]



Operazioni su Liste Puntate Doppie

```
Algoritmo ListaD-Cancella(L, k)
  x = Lista-Cerca(L, k)
  IF prev[x] ≠ NIL
    THEN next[prev[x]] = next[x]
    ELSE Head[L] = next[x]
  IF next[x] ≠ NIL
    THEN prev[next[x]] = prev[x]
```

Head[L]



Operazioni su Liste Puntate Doppie

```
Algoritmo ListaD-canc(L, k)
```

```
  x = Lista-Cerca-ric(L, k)
```

```
  IF x 1 NIL THEN
```

```
    IF next[x] 1 NIL THEN
```

```
      prev[next[x]] = prev[x]
```

```
      L = next[x]
```

```
    IF prev[x] 1 NIL THEN
```

```
      next[prev[x]] = next[x]
```

```
      L = prev[x]
```

```
    "dealloca x"
```

```
  return L
```

```
Algoritmo ListaD-cancella(L, k)
```

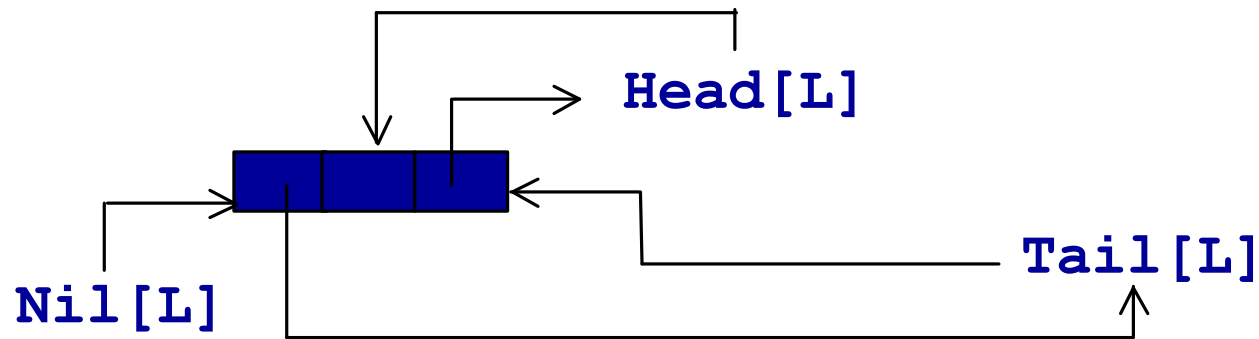
```
  Head[L] = ListaD-canc(Head[L], k)
```



Liste con Sentinella

La Sentinella è un elemento **fittizio** $Nil[L]$ che permette di realizzare le operazioni di modifica di una lista puntata in modo più semplice.

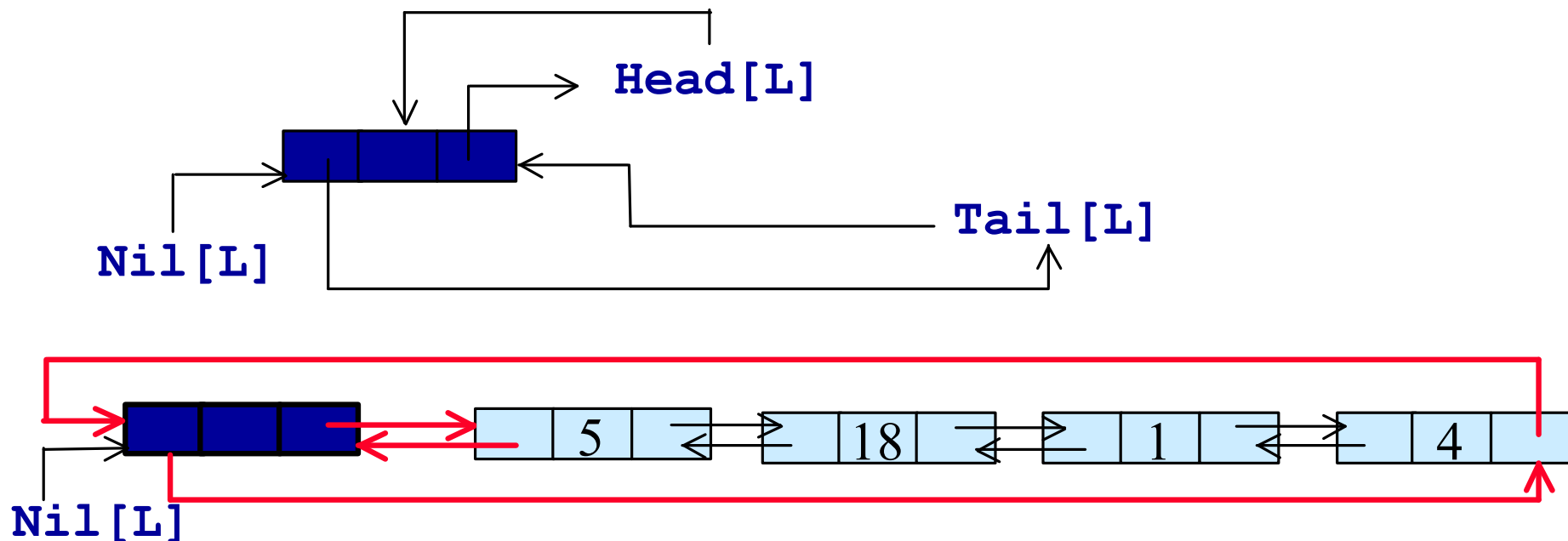
$Nil[L]$ viene inserito tra la testa e la coda della lista.



Liste con Sentinella

La **Sentinella** è un elemento **fittizio** $Nil[L]$ che permette di realizzare le operazioni di modifica di una lista puntata in modo più semplice.

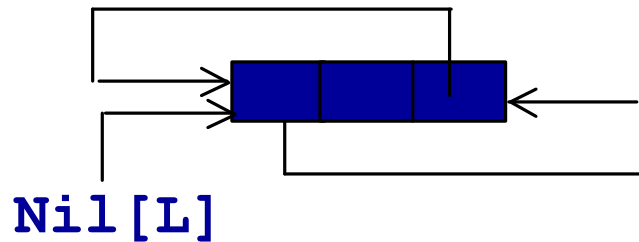
$Nil[L]$ viene inserito tra la testa e la coda della lista. ($Head[L]$ può essere eliminato)



Liste con Sentinella

Nil[L] viene inserito tra la testa e la coda della lista.

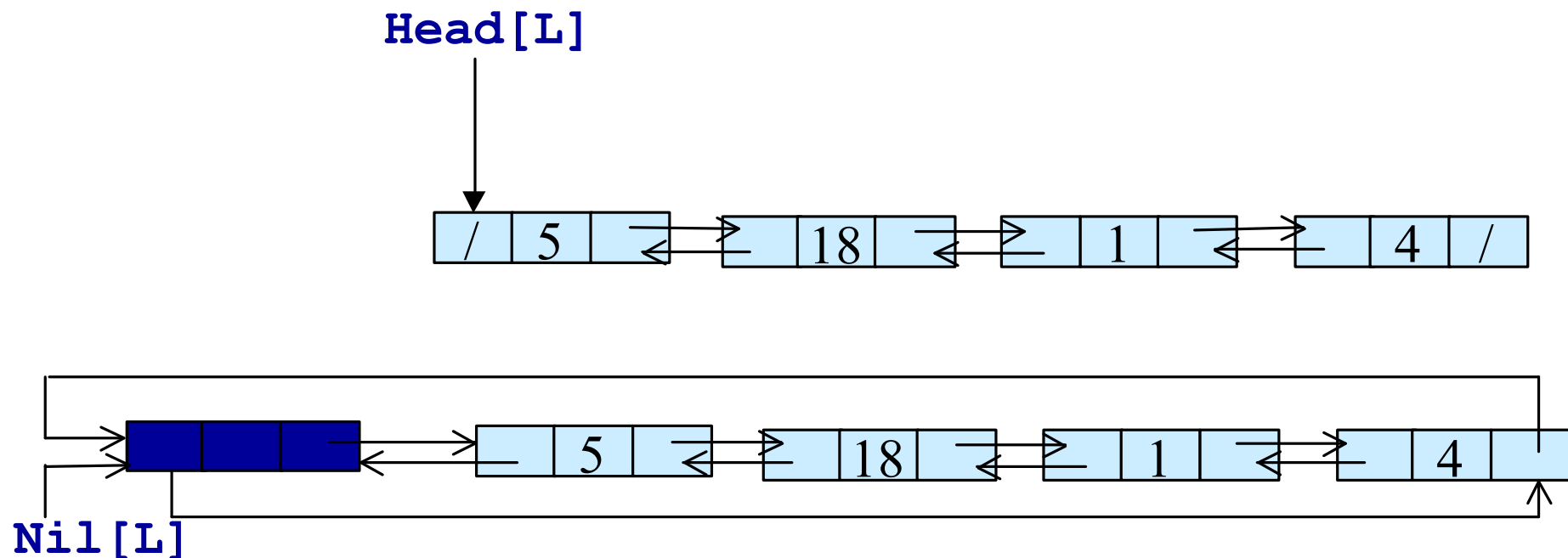
Nil[L] da sola rappresenta la lista vuota (viene sostituito ad ogni occorrenza di ***NIL***)



Liste con Sentinella

$Nil[L]$ viene inserito tra la testa e la coda della lista.

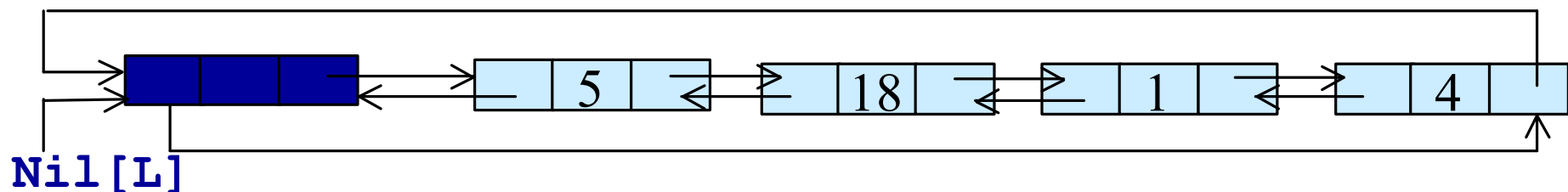
Questo trasforma una **lista (doppia)** in una **lista (doppia) circolare**



Liste con Sentinella

- La **Sentinella** è un elemento **fittizio** $Nil[L]$ che permette di realizzare le operazioni di modifica di una lista puntata in modo più semplice.

Perché non è più necessario preoccuparsi dei **casi limite** (ad esempio **cancellazione in testa/coda**)

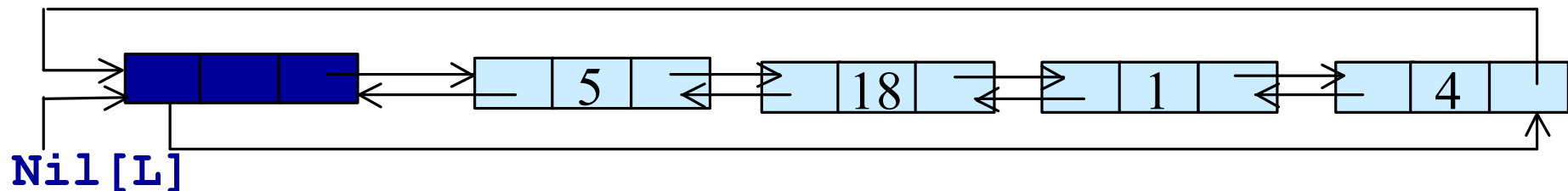


Operazioni su Liste con Sentinella

Algoritmo Lista-Cancella' (L, x)

$next[prev[x]] = next[x]$

$prev[next[x]] = prev[x]$



Operazioni su Liste con Sentinella

Algoritmo Lista-Cancella' (L, x)

$next[prev[x]] = next[x]$

$prev[next[x]] = prev[x]$

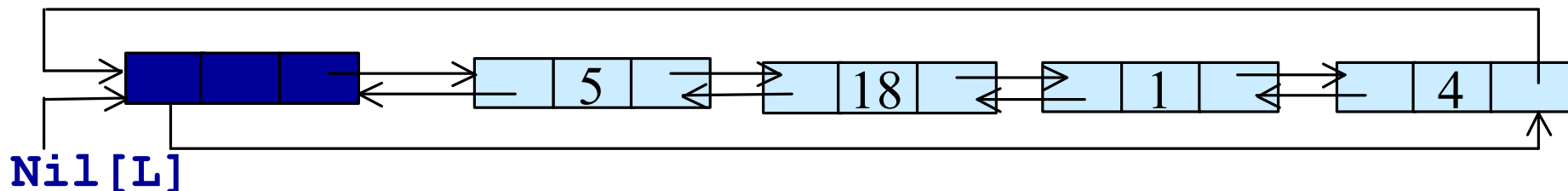
Algoritmo Lista-Inserisci' (L, x)

$next[x] = next[Nil[L]]$

$prev[next[Nil[L]]] = x$

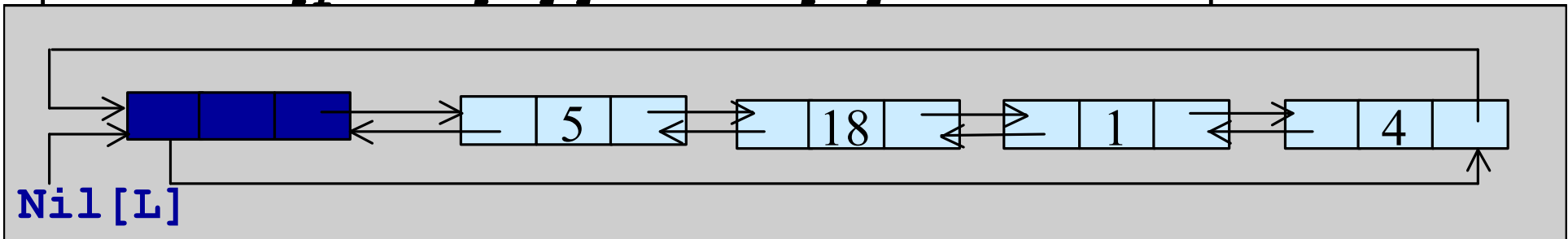
$next[Nil[L]] = x$

$prev[x] = Nil[L]$



Operazioni su Liste con Sentinella

```
Algoritmo Lista-Cancella' (L, x)  
  next [prev[x]] = next [x]
```



```
  prev [next [Nil[L]]] = x  
  next [Nil[L]] = x  
  prev[x] = Nil[L]
```

```
Algoritmo Lista-Cerca' (L, k)  
  x = next [Nil[L]]  
  WHILE x1 Nil[L] and key[x]1 k  
    DO x = next [x]  
  return x
```

Liste LIFO e FIFO

Tramite le liste puntate e loro varianti è possibile realizzare ad esempio implementazioni generali di:

- **Stack** come liste LIFO
- **Code** come liste FIFO (necessita in alcuni casi l'aggiunta di un **puntatore** alla **coda** della lista)

Esercizio: Pensare a quali tipi di lista sono adeguati per i due casi e riscrivere le operazioni corrispondenti

Implementazione di Puntatori

Come è possibile implemetare strutture dati puntate come le Liste o gli Alberi *senza utilizzare i puntatori?*

Alcuni linguaggi di programmazione *non ammettono puntatori* (ad esempio il *Fortran*)

É possibile utilizzare gli stessi algoritmi che abbiamo visto fin'ora in questi linguaggi di programmazione?

Implementazione di Puntatori

É necessario *simulare il meccanismo di gestione della memoria* utilizzando le strutture dati a disposizione.

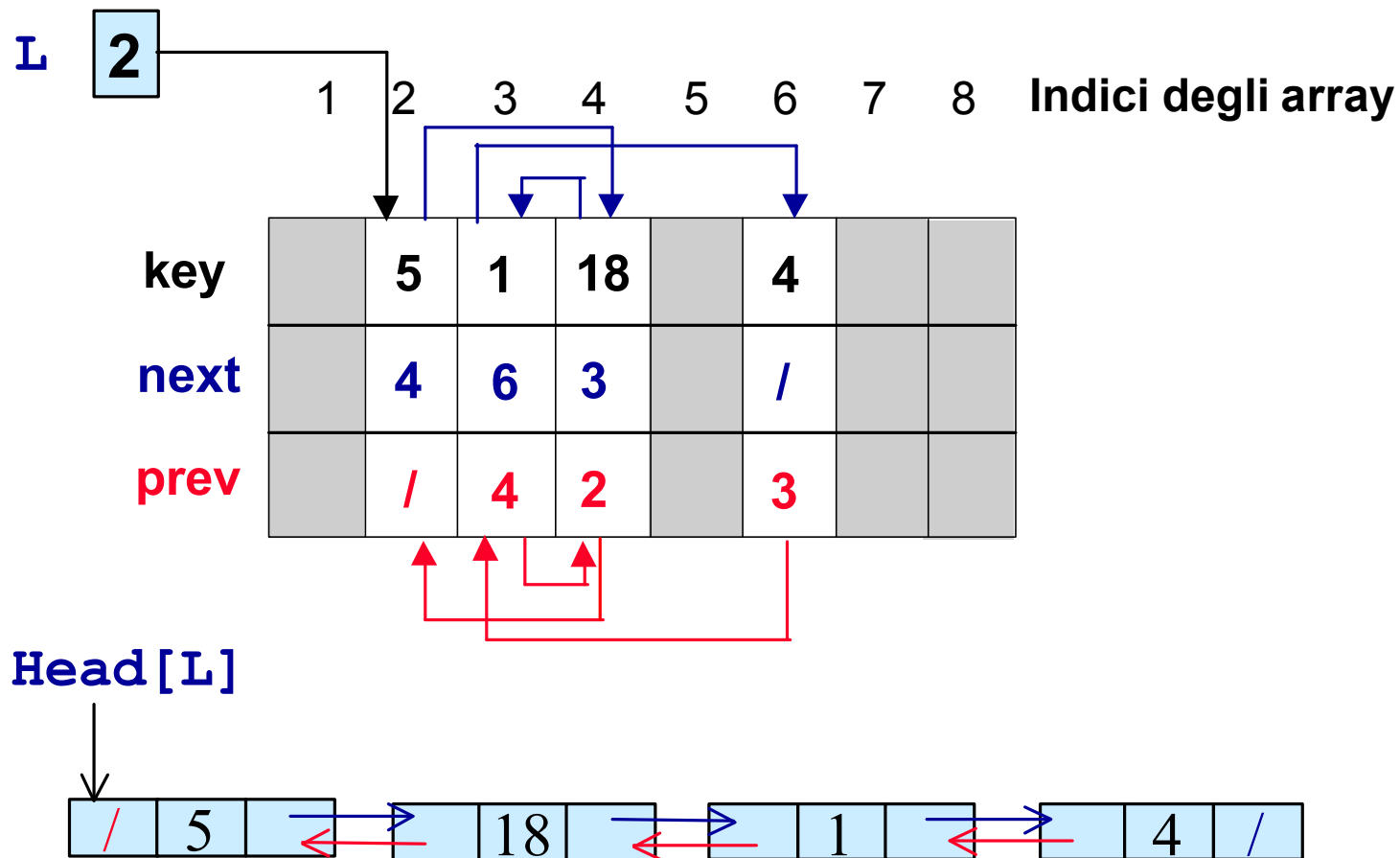
Ad esempio è possibile *utilizzare array* come contenitori di elementi di memoria.

Possiamo usare:

- *un array* key[] per contenere i valori delle chiavi della lista**
- *un array* next[] per contenere i puntatori (valori di indici) all'elemento successivo**
- *un array* prev[] per contenere i puntatori (valori di indici) all'elemento precedente**

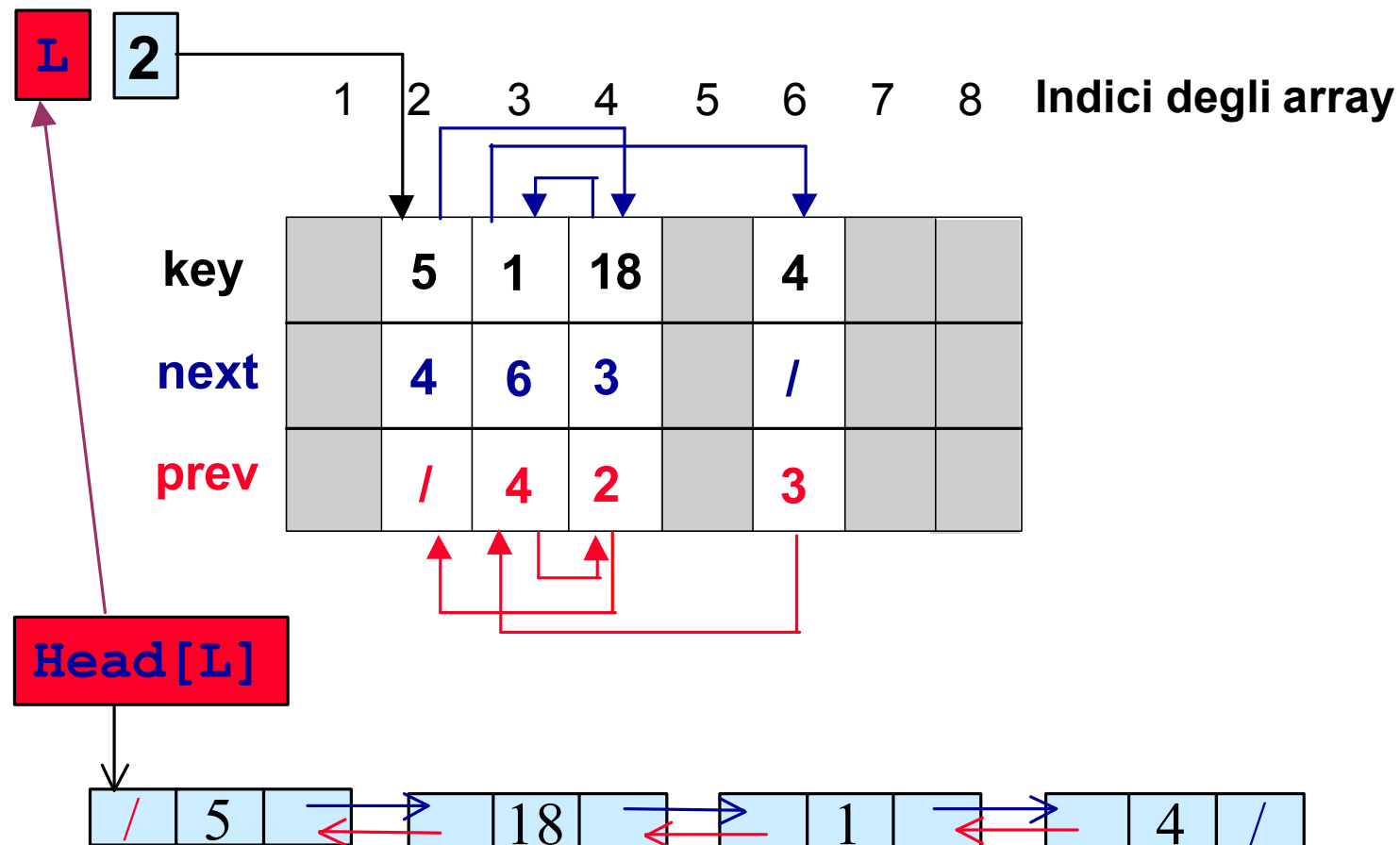
Implementazione di Puntatori

Implementazione di liste puntate doppie con tre array: `key[]`, `next[]` e `prev[]`



Implementazione di Puntatori

Implementazione di liste puntate doppie con tre array: `key[]`, `next[]` e `prev[]`



Implementazione di Puntatori

É necessario simulare il meccanismo di gestione della memoria utilizzando strutture dati a disposizione.

Ad esempio è possibile utilizzare array come contenitori di elementi di memoria.

Ma gli array hanno dimensione fissa e implementarvi strutture dinamiche può portare a sprechi di memoria

Possiamo allora sviluppare un vero e proprio meccanismo di allocazione e deallocazione degli elementi di memoria negli array.

Implementazione di Puntatori

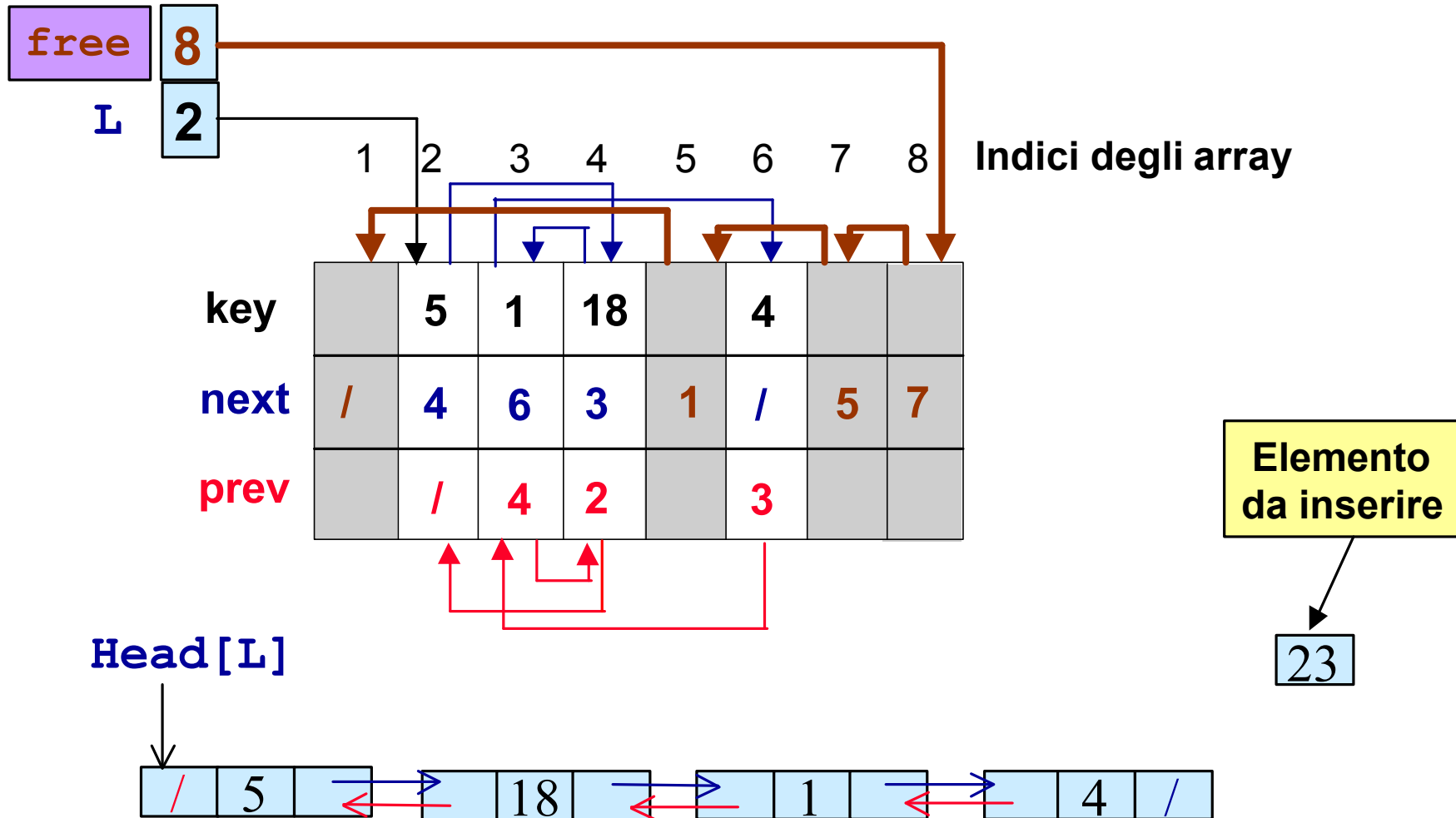
*Possiamo allora sviluppare un vero e proprio **mecanismo di allocazione e deallocazione** degli elementi di memoria negli array.*

Possiamo usare:

- un array **key[]** per contenere i **valori delle chiavi** della lista*
- un array **next[]** per contenere i **puntatori** (valori di indici) **all'elemento successivo***
- un array **prev[]** per contenere i **puntatori** (valori di indici) **all'elemento precedente***
- e una **variabile free** per indicare l'inizio di una **lista di elementi ancora liberi (free list)***

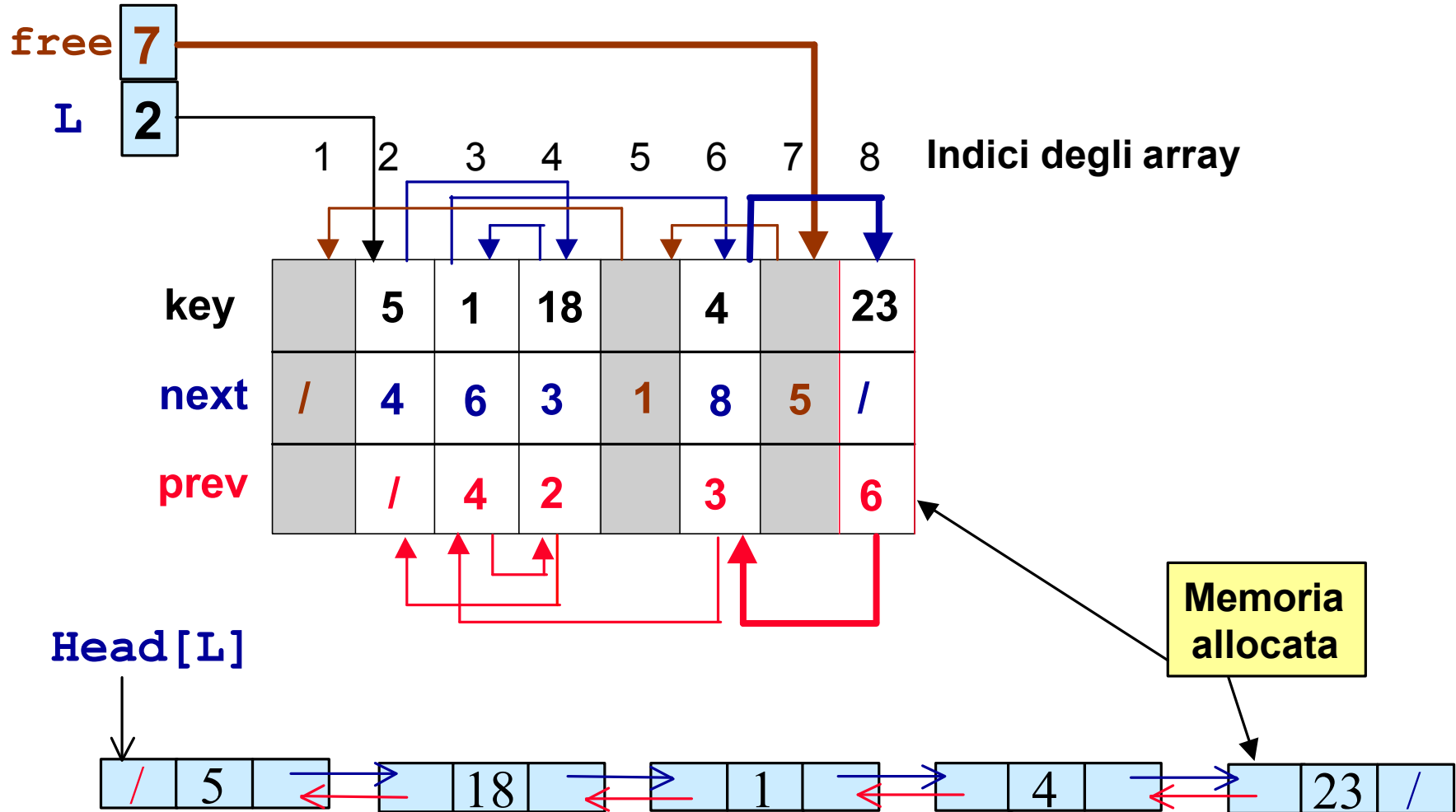
Allocazione memoria

Implementazione di liste puntate doppie con tre array: `key[]`, `next[]` e `prev[]`, `free` è la *free list*



Allocazione memoria

Implementazione di liste puntate doppie con tre array: `key[]`, `next[]` e `prev[]`, `free` è la *free list*

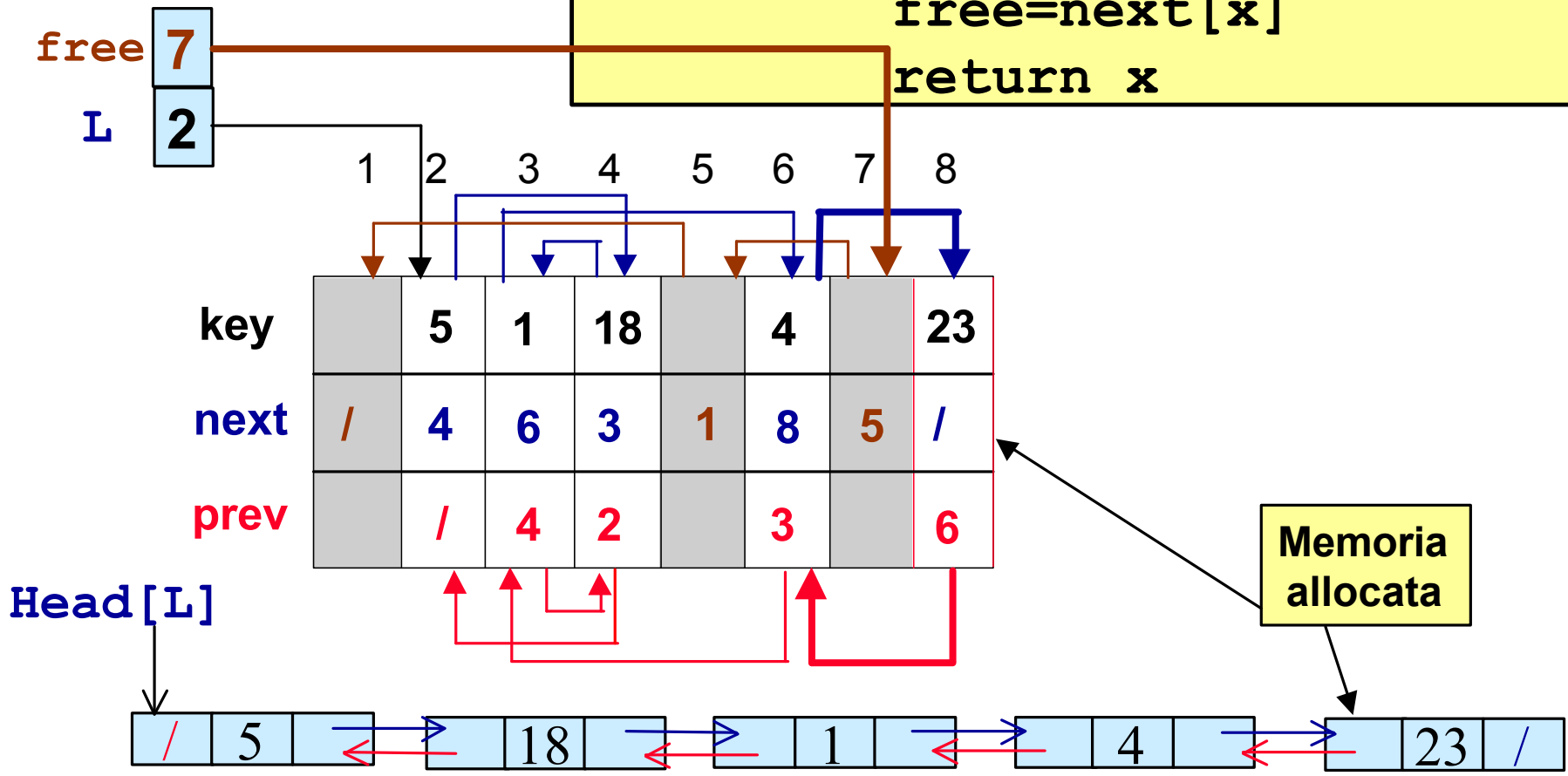


Allocazione memoria

Implementazione
array: key[], next

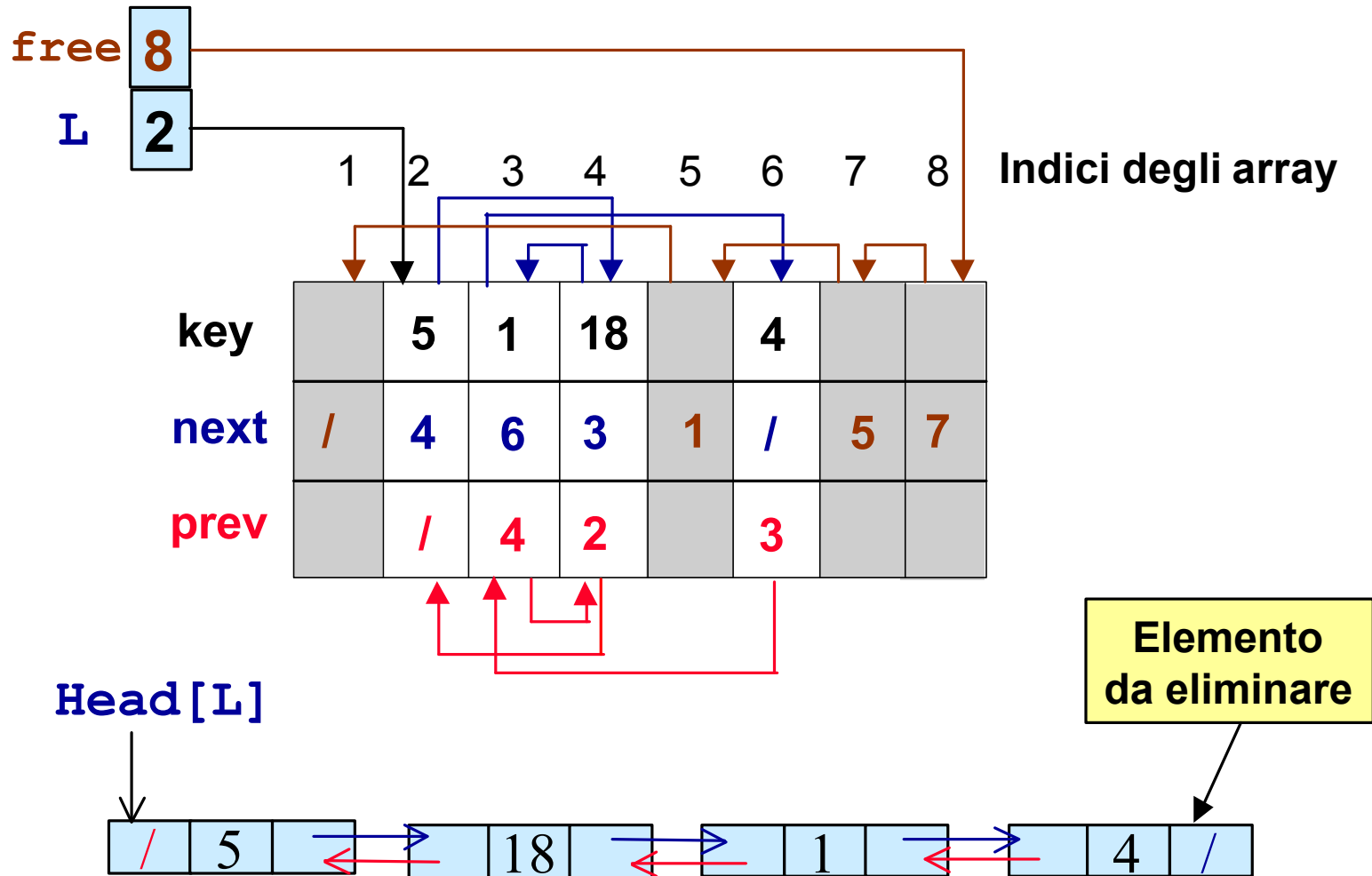
```

Alloca-elemento()
  IF free=NIL
    THEN ERROR "out of memory"
    ELSE x=free
        free=next[x]
        return x
    
```



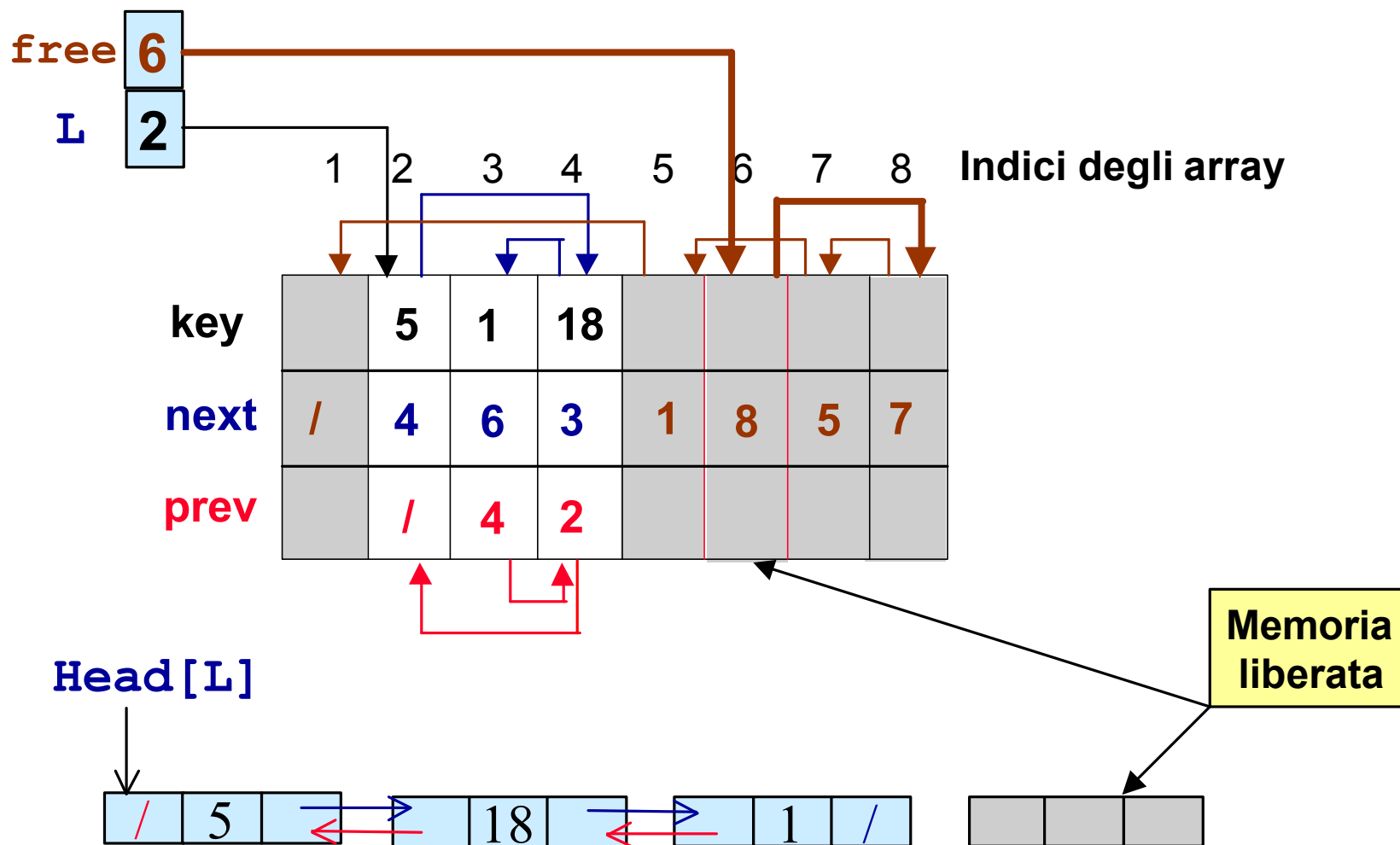
Deallocazione memoria

Implementazione di liste puntate doppie con tre array: `key[]`, `next[]` e `prev[]`, `free` è la *free list*



Deallocazione memoria

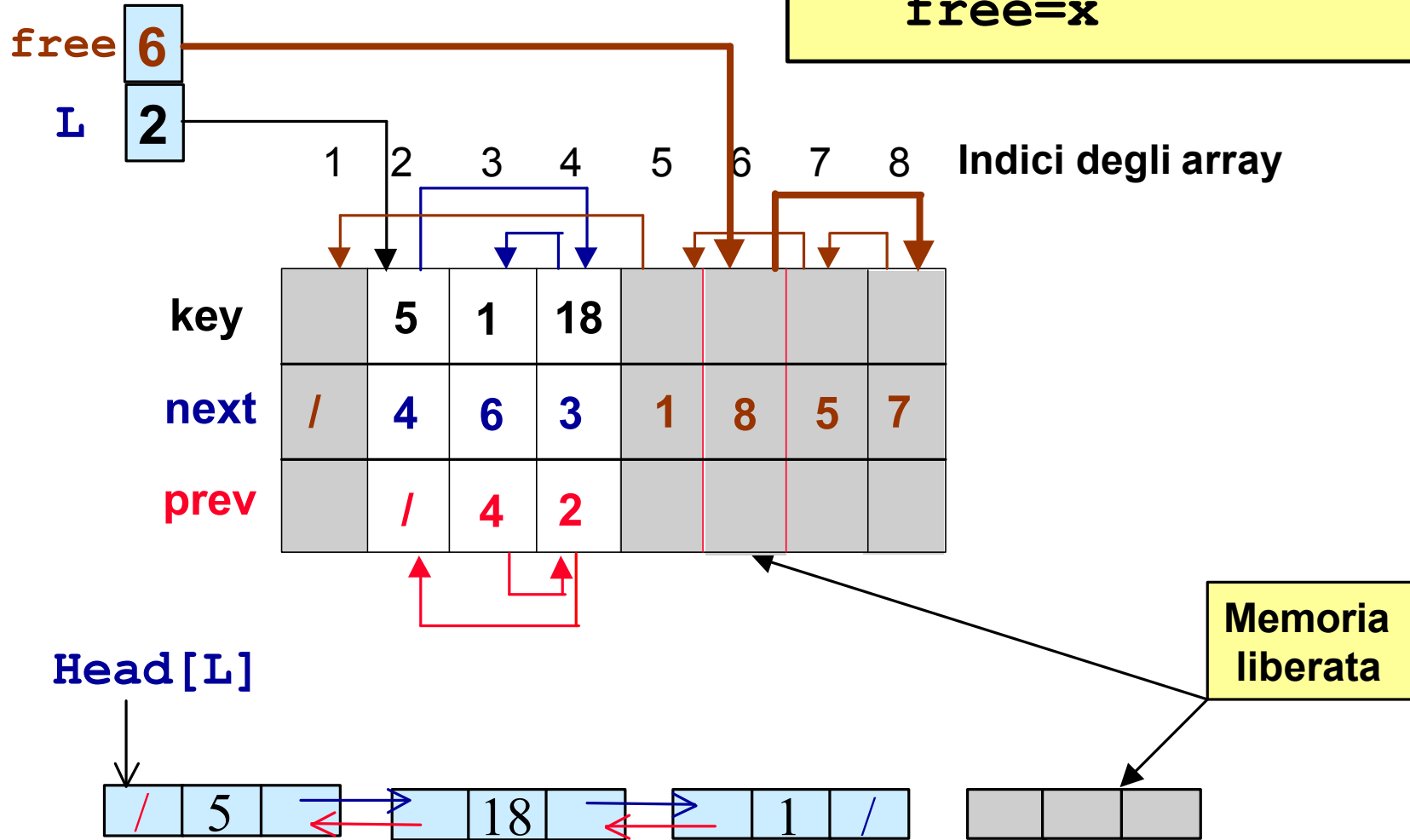
Implementazione di liste puntate doppie con tre array: `key[]`, `next[]` e `prev[]`, `free` è la *free list*



Deallocazione memoria

Implementazione di liste
array: `key[]`, `next[]` e `prev`

Dealloca-elemento (x)
`next[x]=free`
`free=x`



Alberi

Una Albero è un insieme dinamico che

- è *vuoto* oppure
- è composto da *k insiemi disgiunti* di nodi:
 - un *nodo radice*
 - *k* alberi ciascuno detto *sottoalbero i-esimo* (dove $1 \leq i \leq k$)
- Un tale albero si dice di *grado k*

Visita di Alberi

Gli alberi possono essere visitati (o attraversati) in diversi modi:

- ***Visita in Preordine: prima si visita il nodo e poi i suoi figli***
- ***Visita Inordine: prima si visita il figlio sinistro, poi il nodo e poi il figlio destro***
- ***Visita in Postordine : prima si visitano i figli, poi il nodo***

Visita di Alberi

Gli alberi possono essere visitati (o attraversati) in diversi modi:

***Visita in Profondità*: si visitano tutti i nodi lungo un percorso, poi quelli lungo un altro percorso, etc.**

***Visita in Ampiezza*: si visitano tutti i nodi a livello 0, poi quelli a livello 1, ..., poi quelli a livello h**

Visita di Alberi Binari: in profondità preordine

Visita-Preordine (T)

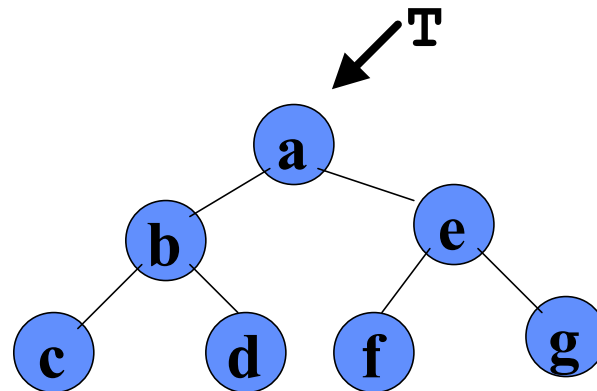
"vista T"

IF figlio-sx[T] != NIL

THEN Visita-Preordine(figlio-sx[T])

IF figlio-destro[T] != NIL

THEN Visita-Preordine(figlio-dx[T])



Sequenza: a b c d e f g

Visita di Alberi Binari: in profondità inordine

```
Visita-Inordine (T)
```

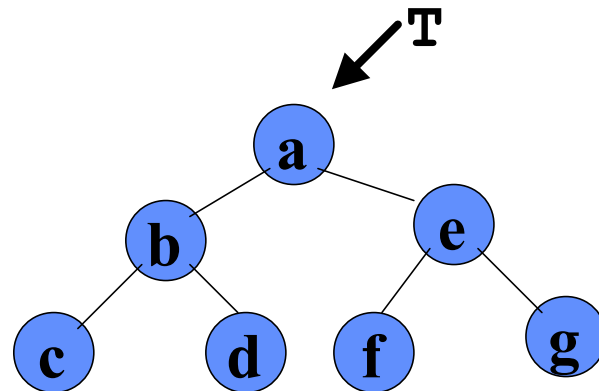
```
  IF figlio-sx[T] != NIL
```

```
    THEN Visita-Inordine (figlio-sx[T])
```

```
  "vista T"
```

```
  IF figlio-dx[T] != NIL
```

```
    THEN Visita-Inordine (figlio-dx[T])
```



Sequenza: c b d a f e g

Visita di Alberi Binari: in profondità postordine

```
Visita-Postordine (T)
```

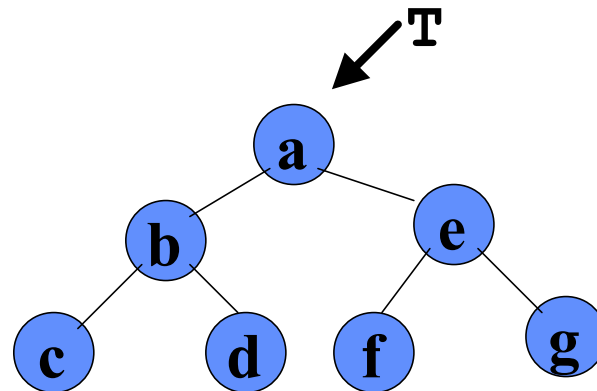
```
  IF figlio-sx[T] != NIL
```

```
    THEN Visita-Postordine (figlio-sx[T])
```

```
  IF figlio-dx[T] != NIL
```

```
    THEN Visita-Postordine (figlio-dx[T])
```

```
  "vista T"
```



Sequenza: **c d b f g e a**

Visita di Alberi k-ari: in ampiezza

Visita-Ampiezza (T)

“crea la coda vuota Q di **dimensione k**”

Accoda (Q, T)

REPEAT

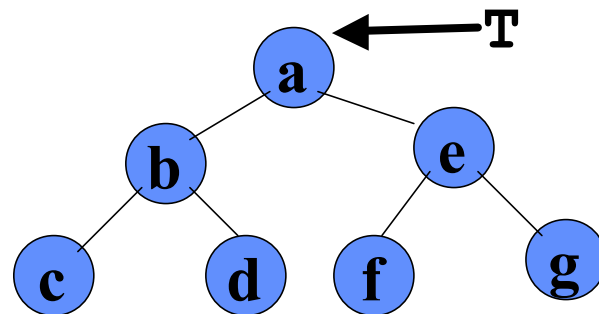
P = Estrai-da-Coda (Q)

“**visita P**”

FOR “ogni figlio F di P da sinistra”

DO Accoda (Q, F)

UNTIL Coda-Vuota (Q)



Sequenza: **a b e c d f g**

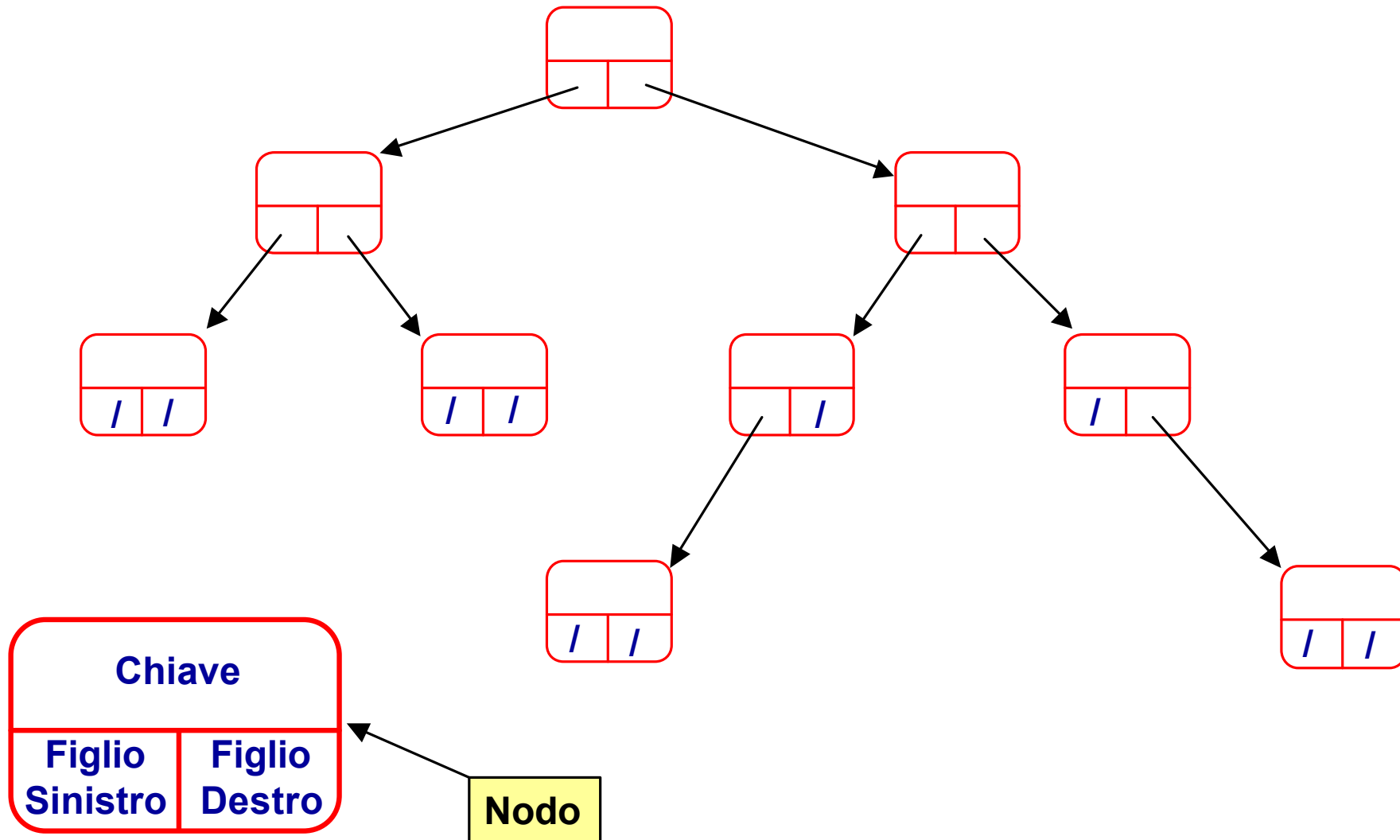
Implementazione di Alberi Binari

Come è possibile *implemetare strutture dati puntate di tipo *Albero?**

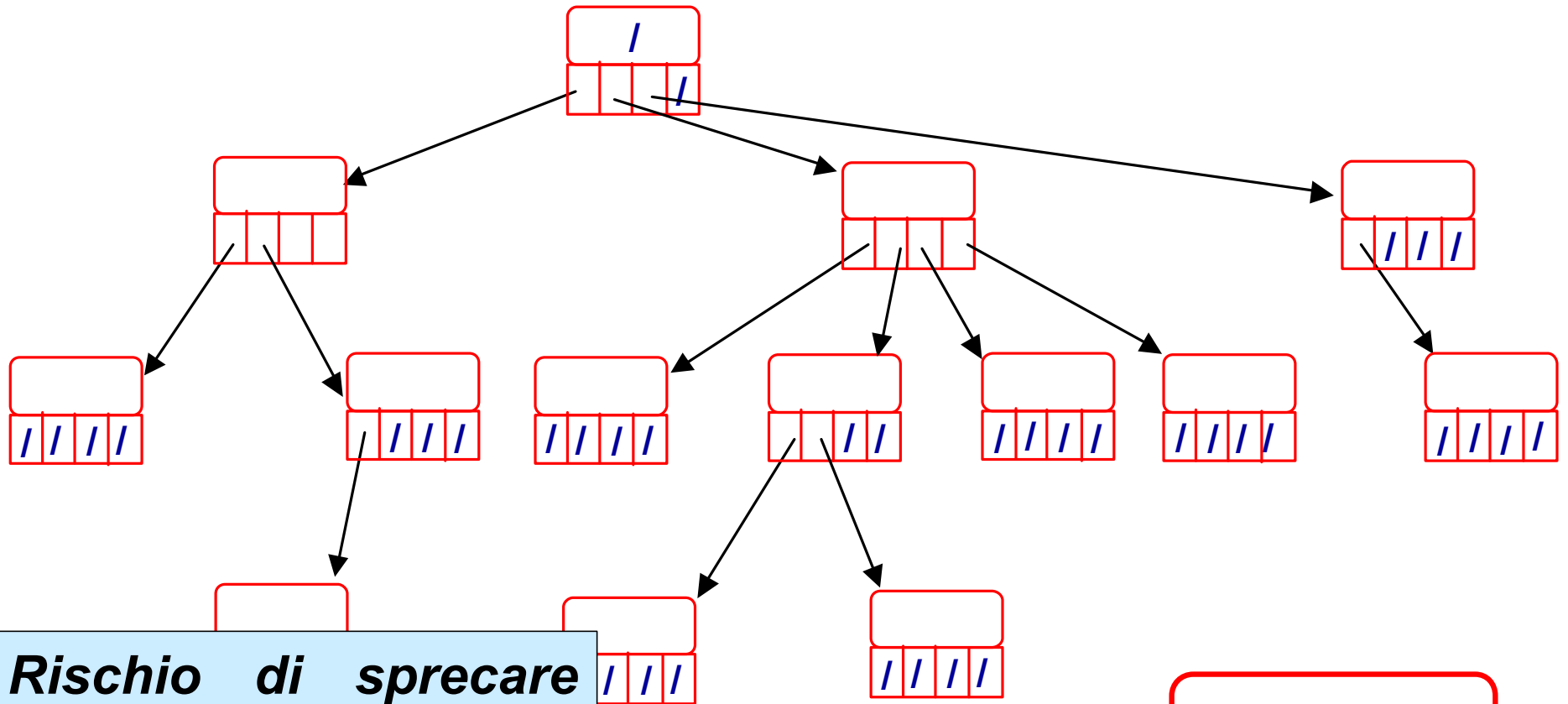
Gli alberi possono essere implementati facilmente utilizzando *tecniche simili* a quelle che impieghiamo *per implementare liste puntate*.

Se non abbiamo a disposizione puntatori, possiamo utilizzare ad esempio *opportuni array* simulando il meccanismo di gestione della memoria (*allocazione, deallocazione*)

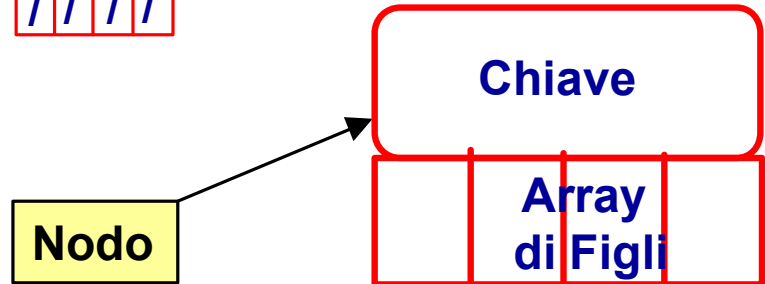
Implementazione di Alberi Binari



Implementazione di Alberi Arbitrari



Rischio di sprecare memoria se molti nodi hanno grado minore del grado massimo k .

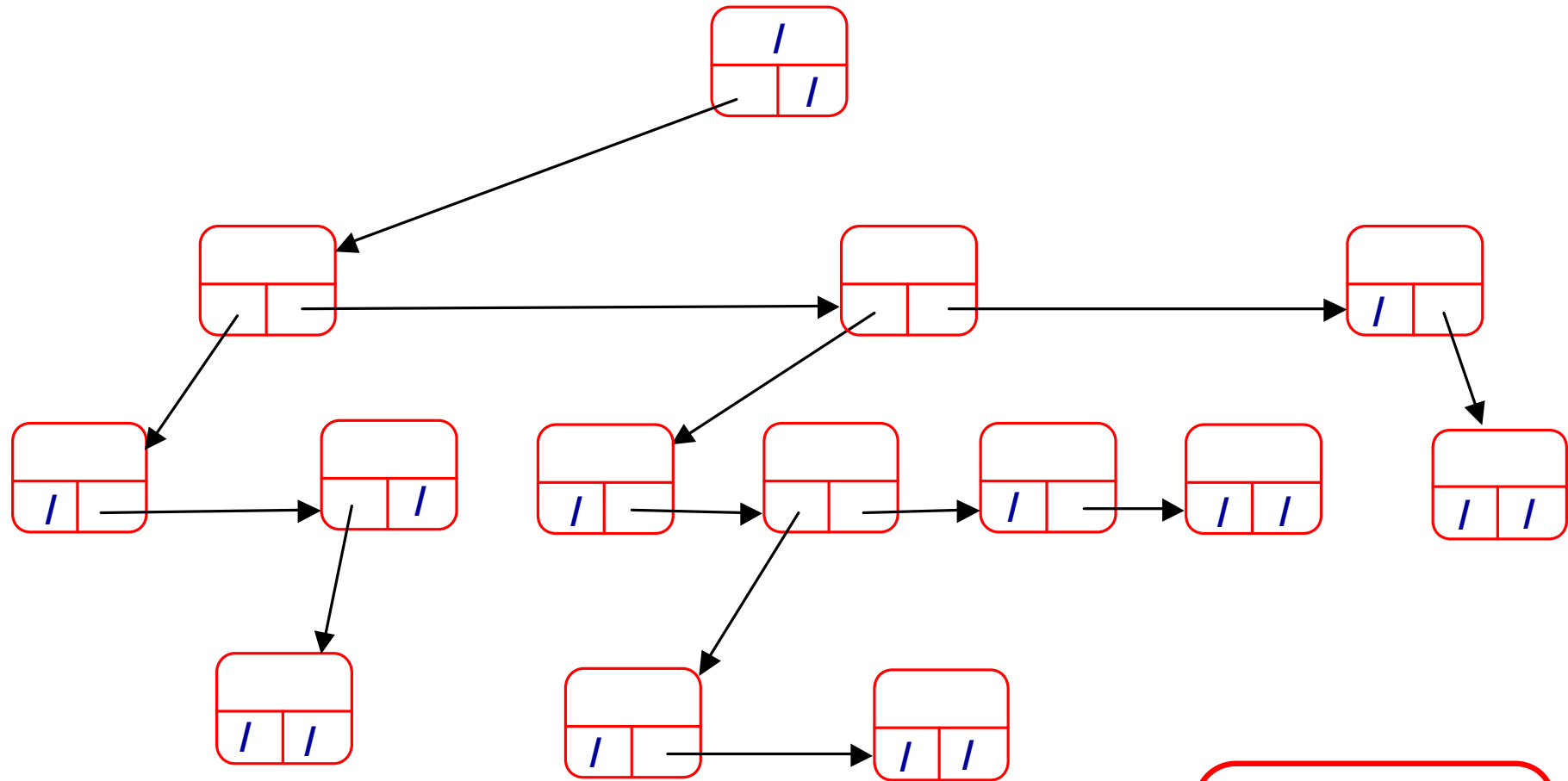


Ricerca in un albero generico

```
tree Search_Tree (T,k)
  if (T == NULL) then
    ris = NULL;
  else
    if (Key[T] == k) then
      ris = T;
    else
      i = Next_Son (T,0) ;
      while (i != 0) do
        ris = Search_Tree (son[T][i],k) ;
        if (ris == NULL) then
          i = Next_Son (T,i) ;
        else
          i = 0;
      return ris;
```

```
int Next_Son (T,idx)
  figlio = idx + 1;
  while (son[T][figlio] == NULL &&
        figlio <= MAX_SONS) do
    figlio = figlio + 1;
  if (figlio <= MAX_SONS)
    return figlio;
  else
    return 0;
```

Implementazione di Alberi Arbitrari



Soluzione: usare una lista di figli (*fratelli*).

Nodo



Ricerca in un albero generico

```
tree Search_Tree (T,k)
  if (T == NULL) then
    ris = NULL;
  else
    if (Key[T] == k) then
      ris = T;
    else
      figlio = First_Son(T);
      while (figlio != NULL) do
        ris = Search_Tree(figlio,k);
        if (ris == NULL) then
          figlio = Next_Son(T,son);
        else /* chiave gia` trovata */
          figlio = NULL;
      return ris;
```

```
tree First_Son(T)
  return (sx[T]);

tree Next_Son(T,son)
  return (dx[son]);
```