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:***
  - $\mathbb{N}$ **denota l'insieme dei *numeri naturali***
  - $\mathbb{R}$ **denota l'insieme dei *numeri reali***
  - $\varnothing$ **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:***

- ***S = Inserimento(S,x):***
- ***S = 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 può essere visto come 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*

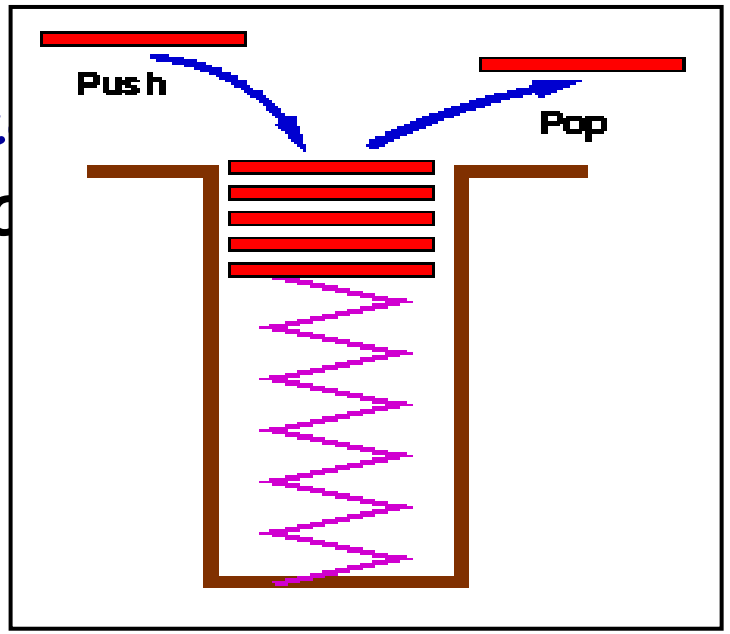
Cancellazione: `Pop(S)`

- *rimuove un elemento dalla cima dello Stack*

Altre operazioni: `Stack-Vuot`

- *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)**

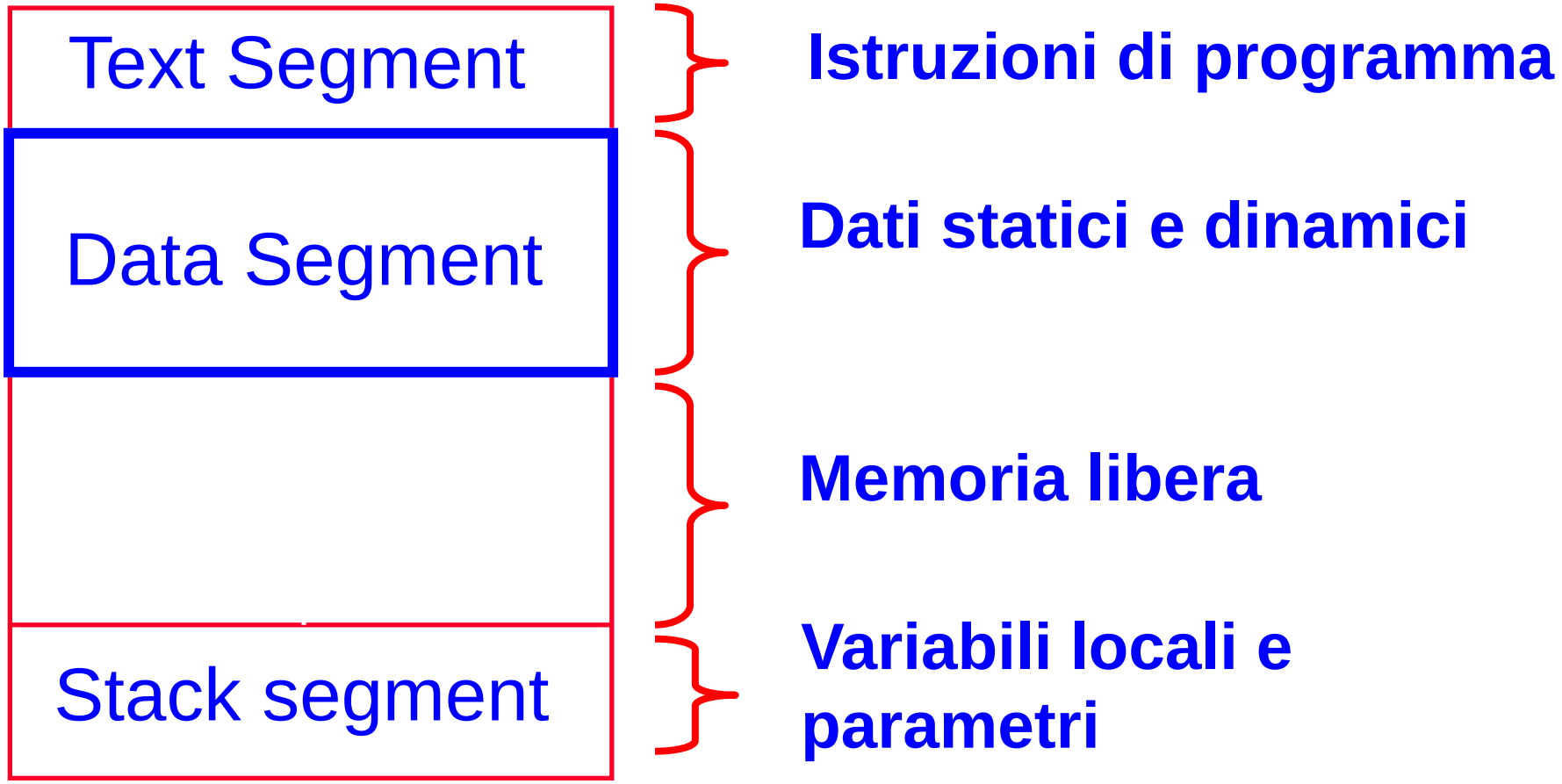
Stack: implementazione

- **Arrays**
  - **Permettono di implementare stack in modo semplice**
  - **Flessibilità limitata, *ma* incontra parecchi casi di utilizzo**
  - **La capacità dello Stack è tipicamente limitata ad una quantità costante:**
    - **dalla dimensione dell'array utilizzato**
    - **in generale dalla memoria disponibile del computer.**
- **Possibile implementarlo con Liste Puntate.**

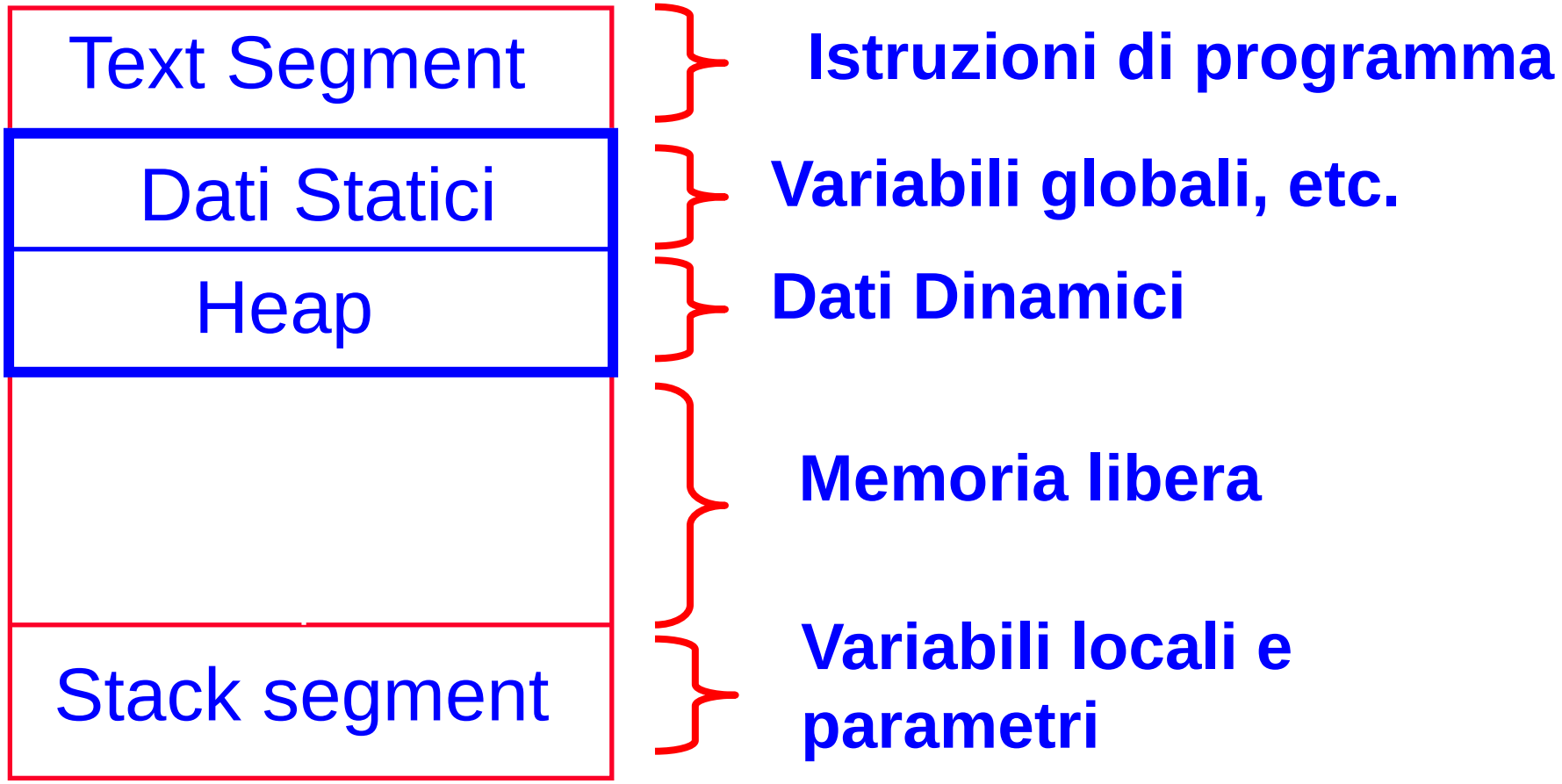
Stack: applicazione

- **Stack è molto frequente in Informatica:**
  - **Elemento chiave nel meccanismo che implementa la chiamata/ritorno di funzioni/procedure**
  - ***Record di attivazione* permettono la ricorsione.**
  - **Chiamata: *push* di un record di attivazione**
  - **Return: *pop* di un record di attivazione**

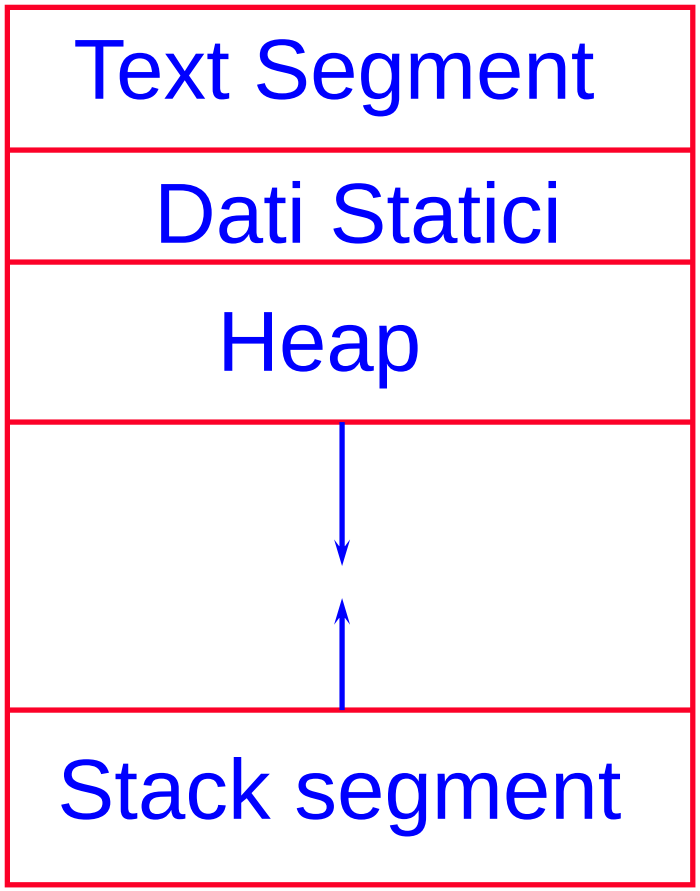
Gestione della memoria dei processi



Gestione della memoria dei processi



Gestione della memoria dei processi



La memoria è allocata e deallocata secondo necessità

Stack: applicazioni

- **Stacks è molto frequente:**
  - **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 formali) della funzione**
  - **Indirizzo di ritorno**
  - **Valore di ritorno della funzione**
  - **Variabili locali della funzione**

Stack di Record di Attivazione in LP

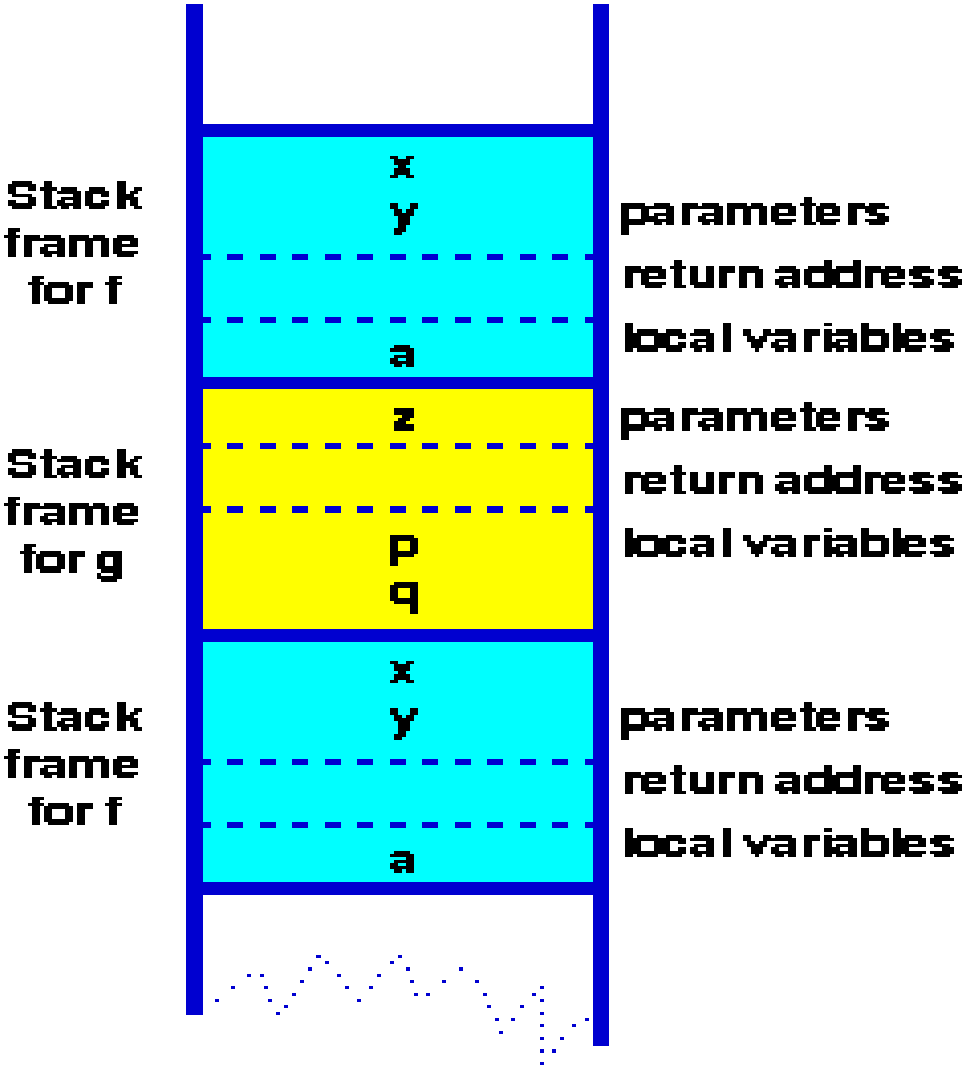
Programma

```
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

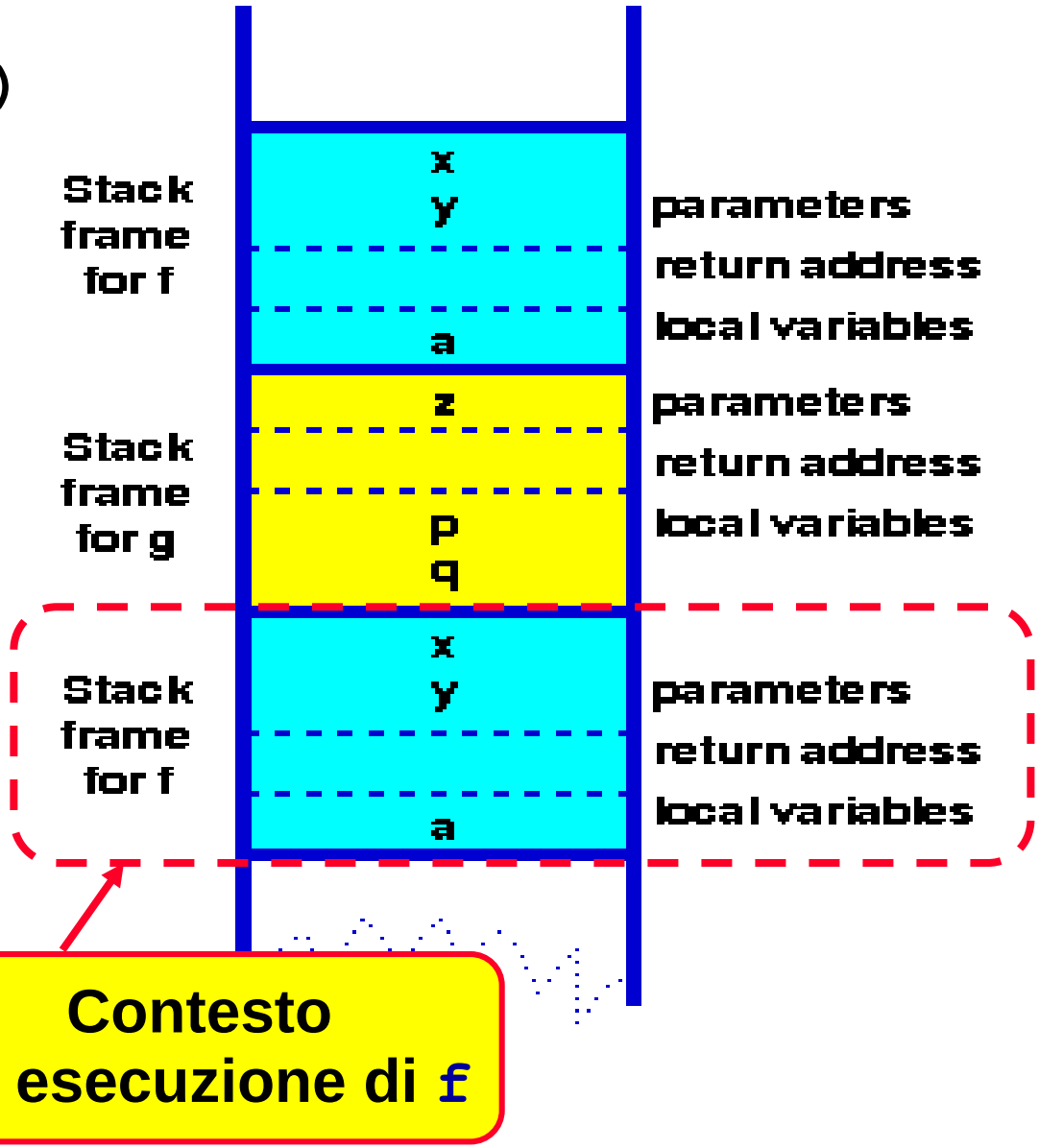
```
function f(int x,int y)
 {
   int a;
   if ( term_cond )
        return …;
   a = …;
   return g( a );
   }

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);
    }
```



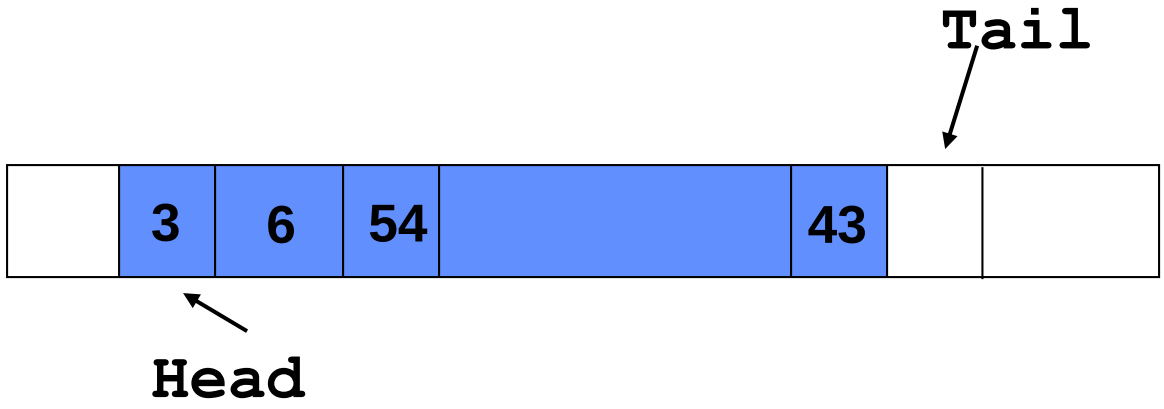
Code

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)

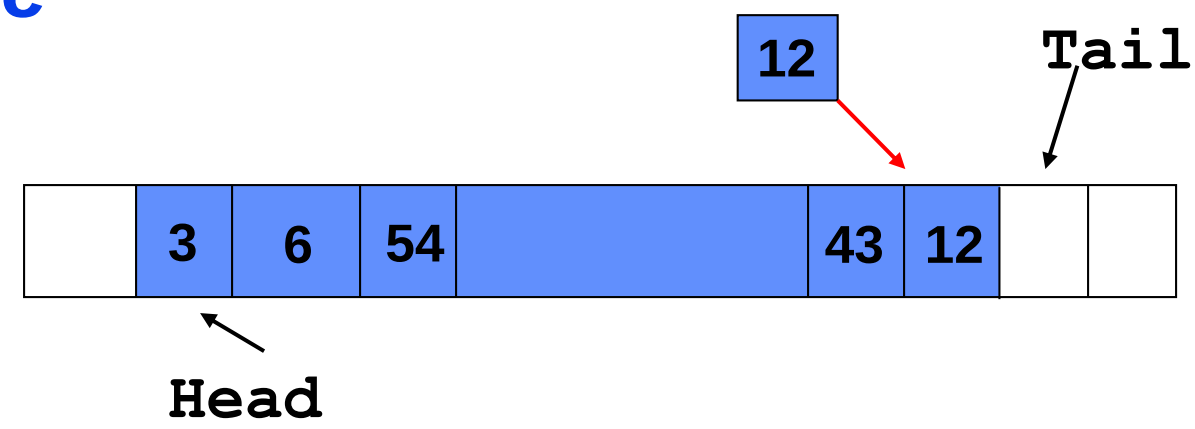
Code



Una Coda implementa una lista di tipo "first in, first out" (FIFO)

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

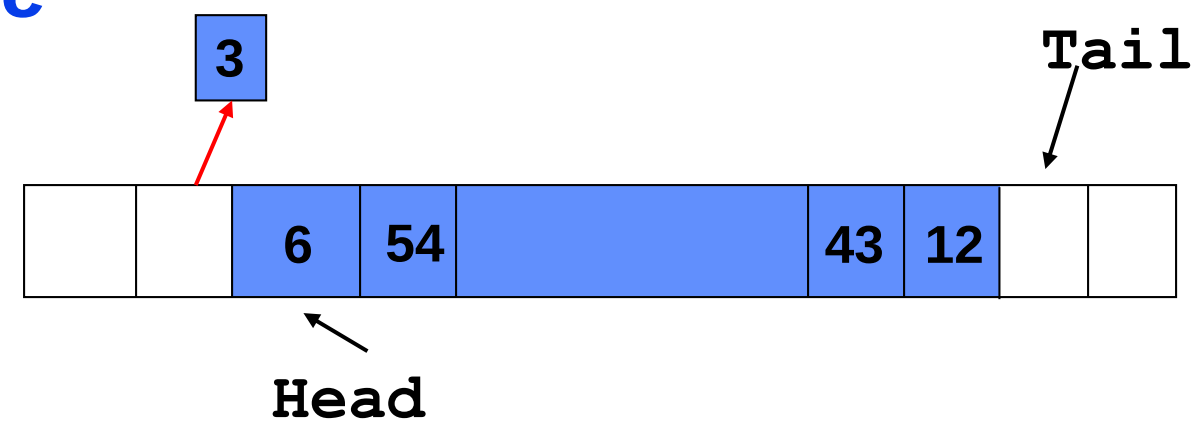
Code



*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 in coda **al posto referenziato da Tail***

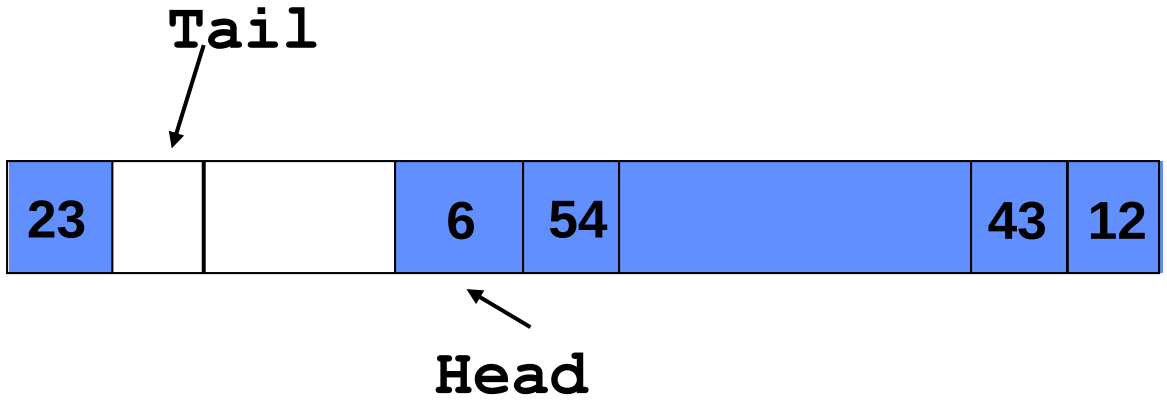
Code



*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 in coda **al posto referenziato da Tail***
- *Quando si **estrae** un elemento, viene estratto **dalla testa***

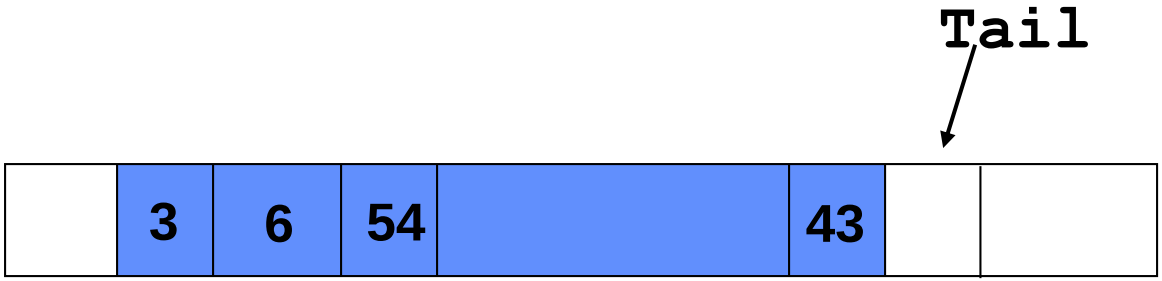
Code

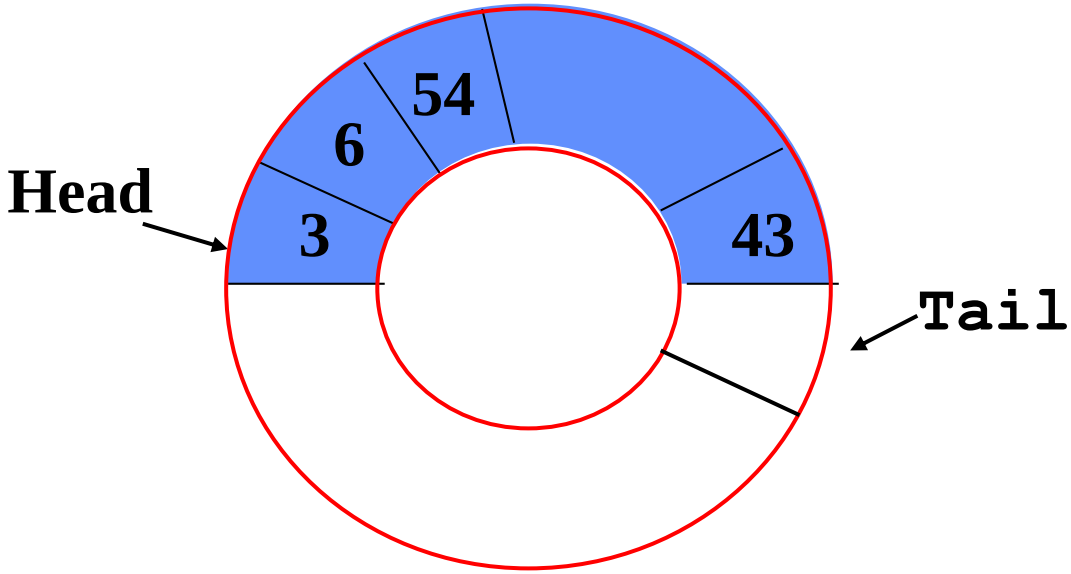


Una Coda implementa una lista di tipo "first in, first out" (FIFO)

- *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

Assumendo che l'array Q abbia indici 0…(n-1):

```
Algoritmo Accoda(Q,x)
   Q[Tail_Q] = x
   Tail_Q = (Tail_Q + 1) mod n
```

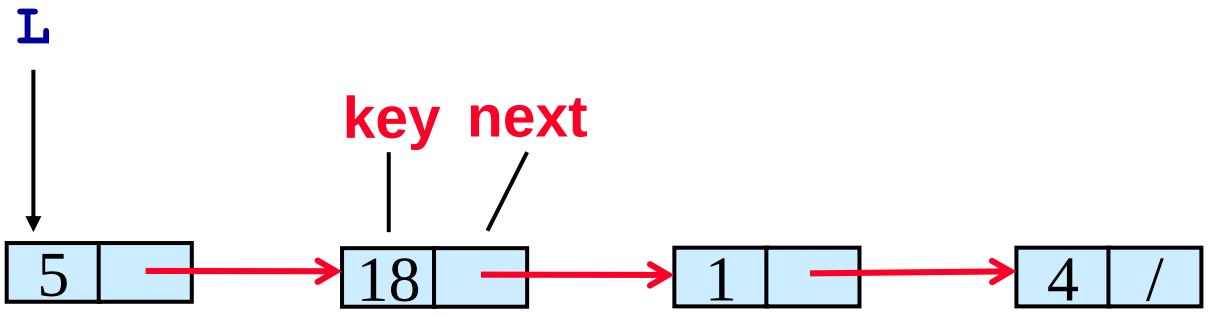
```
Algoritmo Estrai-da-Coda(Q)
    x = Q[Head_Q]
    Head_Q = (Head_Q + 1) mod n
    return x
```

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

Liste Puntate

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

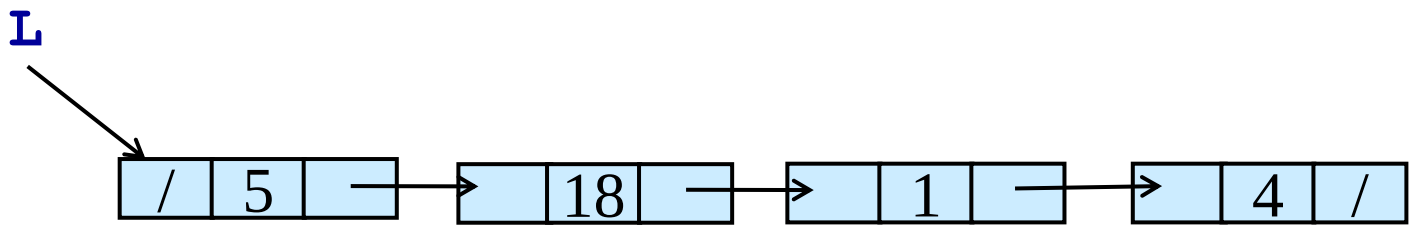
È una struttura dati ad accesso strettamente sequenziale!



Operazioni su Liste Puntate

```
algoritmo Lista-Cerca-ric(L,k)
   IF L ≠ NIL and L->key ≠ k THEN
         L = Lista-Cerca-ric(L->next,k)
   return L
```

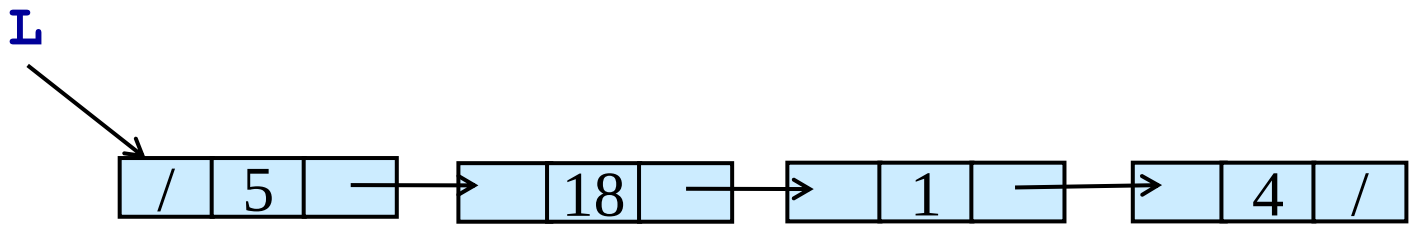
```
   …
   elem = Lista-Cerca-ric(L,k)
   …
```



Operazioni su Liste Puntate

```
Algoritmo Lista-Insert(L,k)
    "alloca nodo new"
    new->key = k
    new->next = L
    return new
```

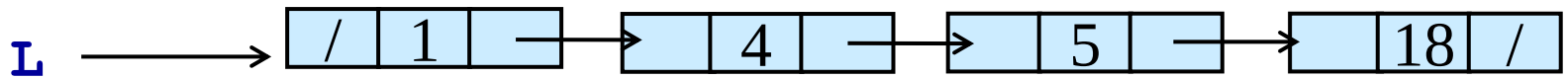
```
    …
    L = Lista-insert(L,k)
    …
```



Operazioni su Liste Puntate (ordinata)

```
Algoritmo Lista-Ord-insert(L,k)
   IF L ≠ NIL && L->key < k THEN
        L->next = Lista-Ord-insert(L->next,k)
   ELSE /* chiave k è la più piccola in L */
            "alloca nodo elem"
            elem->key = k
            elem->next = L
            L = elem
   return L
```

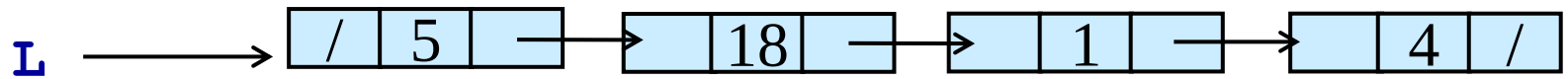
```
   …
   L = Lista-Ord-insert(L,k)
   …
```



Operazioni su Liste Puntate

```
Algoritmo Lista-cancella-r(L,k)
   IF L ≠ NIL THEN
      IF L->key = k THEN
         elem = L
         L = L->next
         "dealloca elem"
      ELSE /* k non trovata in L */
         L->next = Lista-cancella-r(L->next,k)
   return L
```

```
   …
   L = Lista-cancella-r(L,k)
   …
```



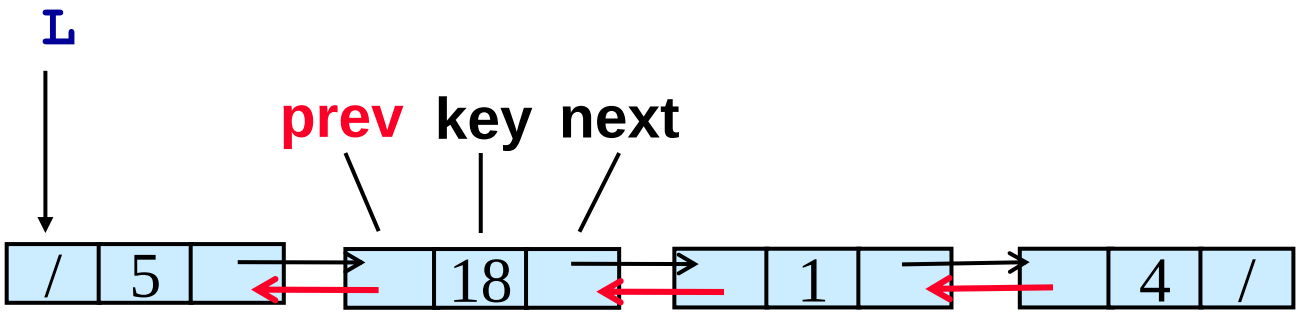
Esercizio

Scrivere un algoritmo ricorsivo che cancelli da una lista ordinata *L* tutti gli elementi con chiave compresa tra i valori k_1 e k_2 (con $k_1 \leq k_2$).

Liste Puntate Doppie

Una Lista Doppia Puntata è un insieme dinamico in cui in cui ogni elemento ha:

- ***una chiave (`key`)***
- ***un riferimento (`next`) all'elemento successivo dell'insieme***
- ***un riferimento (`prev`) all'elemento precedente dell'insieme.***

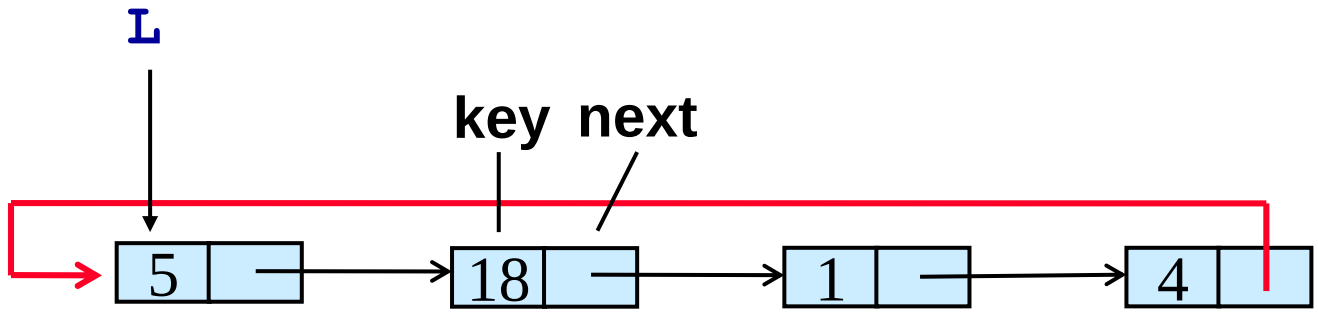


Liste Puntate Circolare

Una Lista Circolare puntata è un insieme dinamico in cui in cui ogni elemento ha:

- ***una chiave (`key`) e***
- ***un riferimento (`next`) all'elemento successivo dell'insieme.***

L'ultimo elemento ha un riferimento alla testa della lista

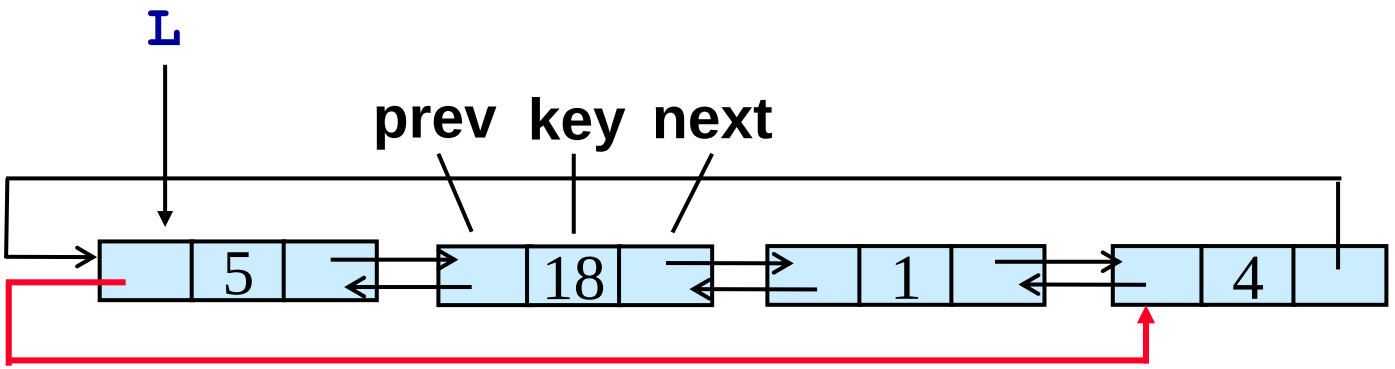


Liste Puntate Circolare Doppia

Una Lista Circolare puntata è un insieme dinamico in cui in cui ogni elemento ha:

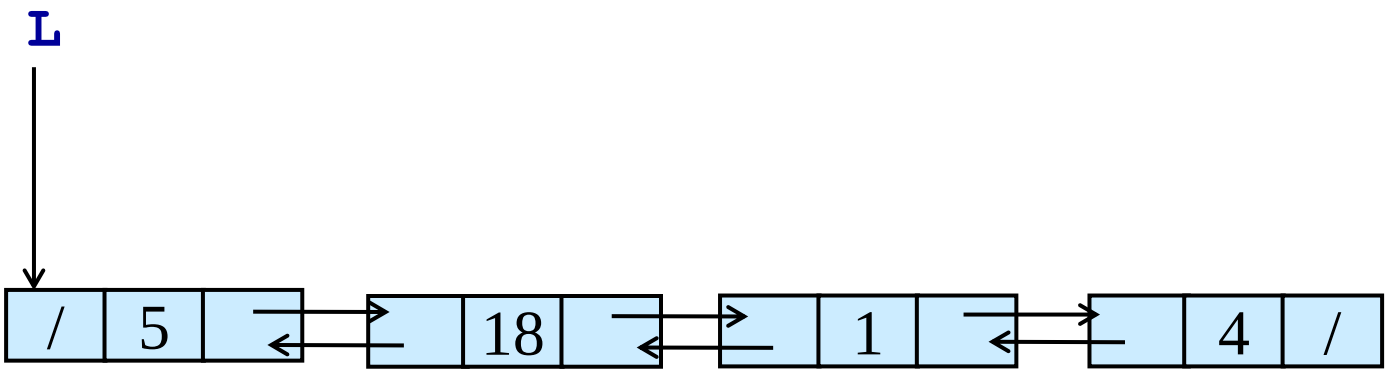
- ***una chiave (`key`)***
- ***un riferimento (`next`) all'elemento successivo dell'insieme***
- ***un riferimento (`prev`) all'elemento prec-dente dell'insieme.***

L'ultimo elemento ha un riferimento (`prev`) alla testa della lista, il primo ha un riferimento (`next`) alla coda della lista



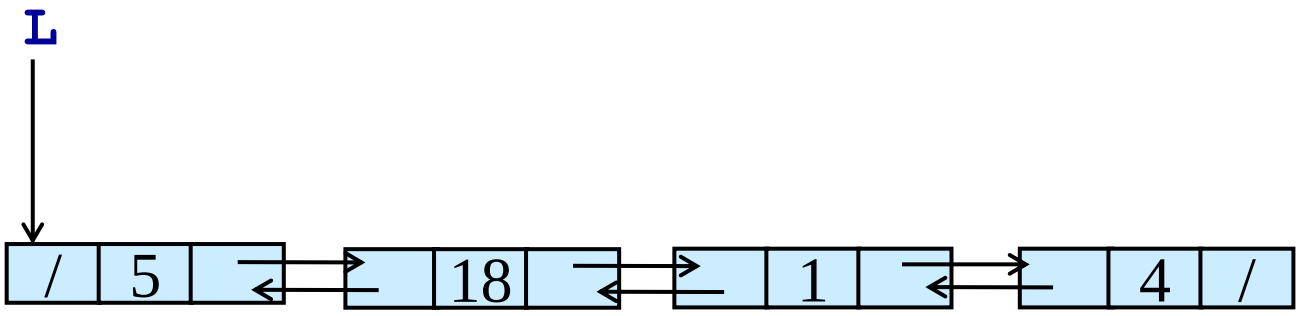
Operazioni su Liste Puntate Doppie

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



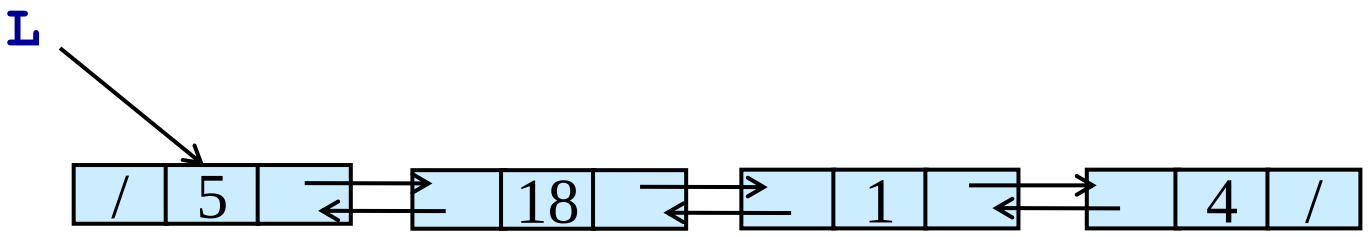
Operazioni su Liste Puntate

```
Algoritmo ListaD-Inserisci(L,k)
   "alloca nodo elem"
   elem->key = k
   elem->next = L
   IF L ≠ NIL
        THEN L->prev = elem
   L = elem
   L->prev = NIL
```



Operazioni su Liste Puntate Doppie

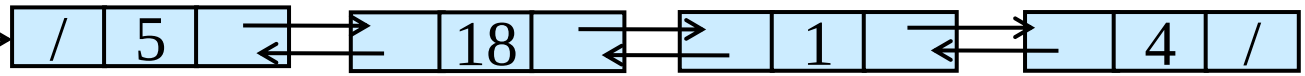
```
Algoritmo ListaD-Cancella(L,k)
   x = Lista-Cerca(L,k)
   IF x->prev ≠ NIL
        THEN x->prev->next = x->next
        ELSE L = x->next
   IF x->next ≠ NIL
        THEN x->next->prev = x->prev
   "dealloca x"
```



Operazioni su Liste Puntate Doppie

```
Algoritmo ListaD-canc(L,k)
 x = Lista-Cerca-ric(L,k)
 IF x ≠ NIL THEN
    IF x->next ≠ NIL
        THEN x->next->prev = x->prev
    IF x->prev ≠ NIL
        THEN x->prev->next = x->next
        ELSE L = x->prev
    "dealloca x"
 return L
```

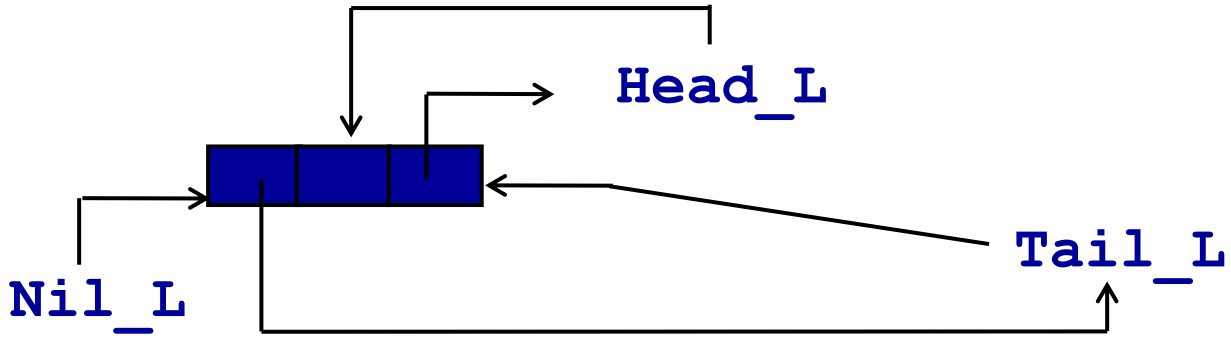
```
…
L = ListaD-canc(L,k)
…
```



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.

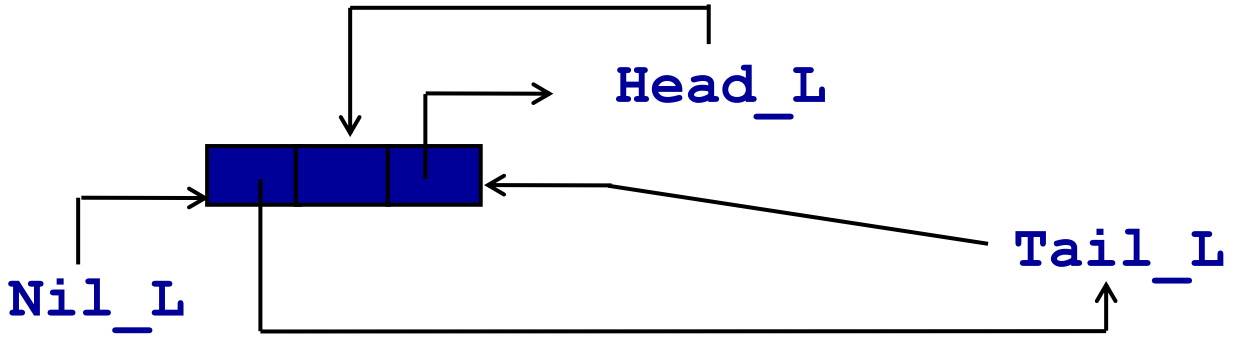
`Nil_L` viene inserito tra la testa e la coda della lista.

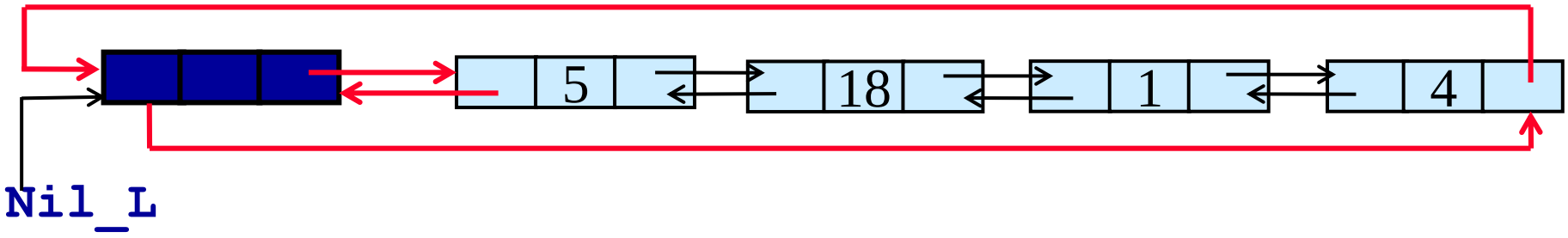


Liste con Sentinella

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 sostituito con un puntatore a `Nil_L`)

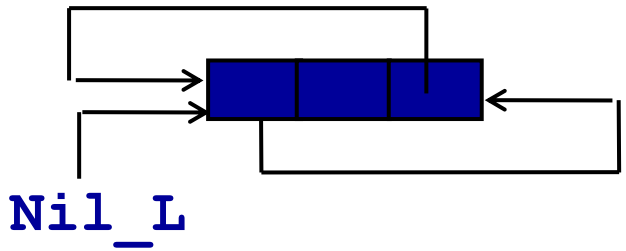




Liste con Sentinella

`Nil_L` ***viene inserito tra la testa e la coda della lista.***

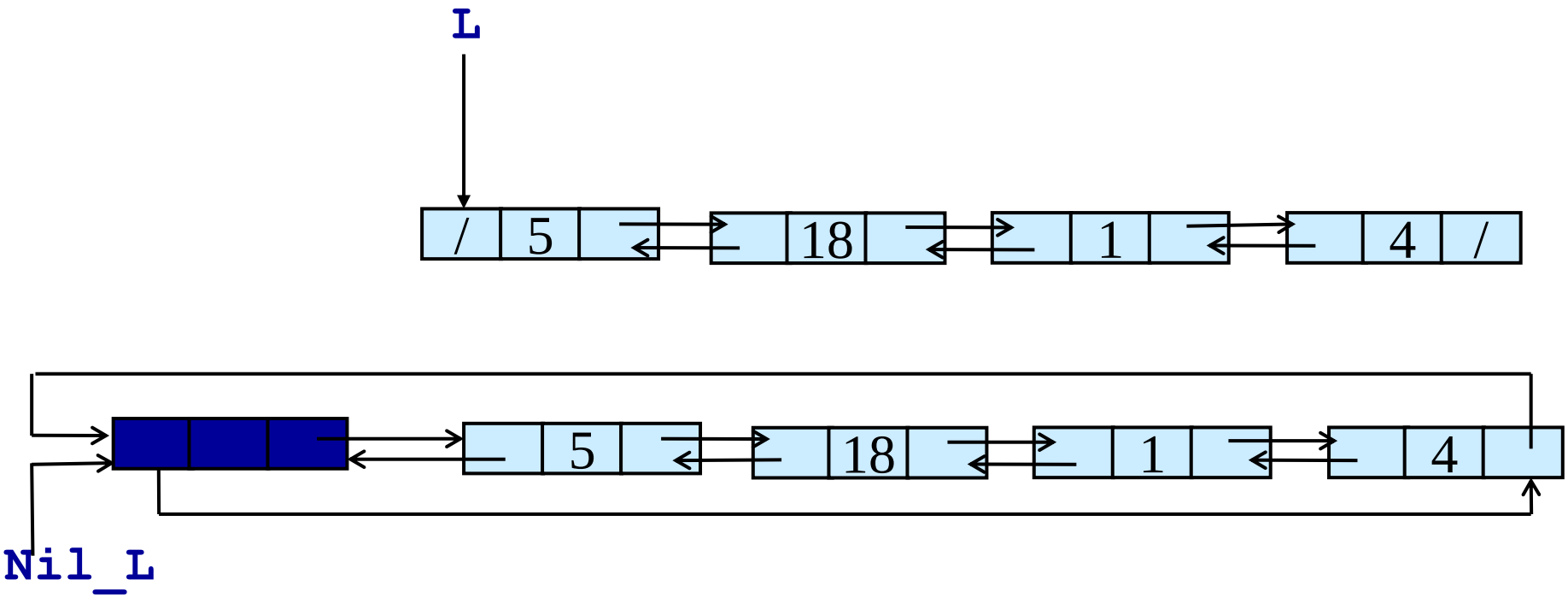
`Nil_L` ***da solo rappresenta la lista vuota (viene sostituito ad ogni occorrenza di*** **`NIL`***)*



Liste con Sentinella

`Nil_L` ***viene inserito tra la testa e la coda della lista.***

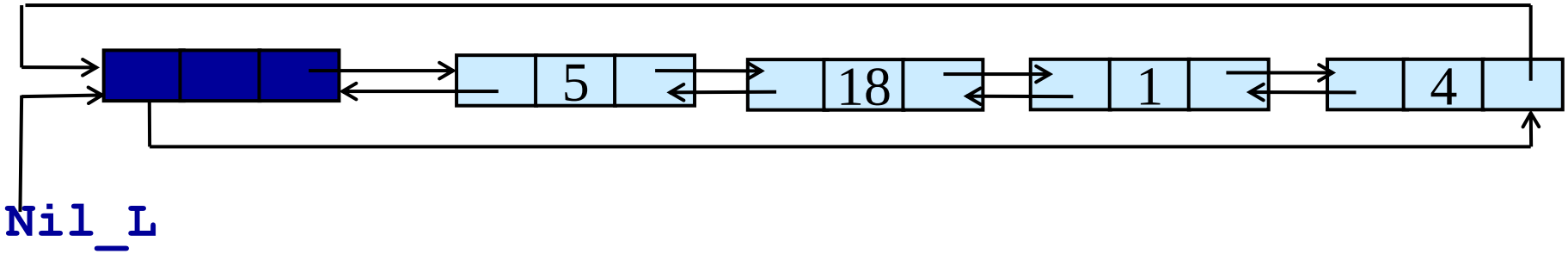
Questo trasforma una lista (doppia) in una lista (doppia) circolare



Liste con Sentinella

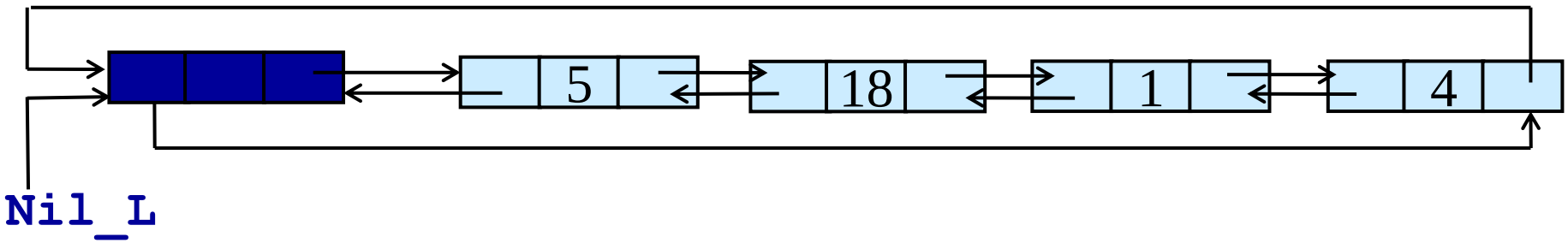
- ***La Sentinella è un elemento fittizio `Nil_L` che permette di realizzare le operazioni di modifica di una lista puntata in modo più semplice.***

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



Operazioni su Liste con Sentinella

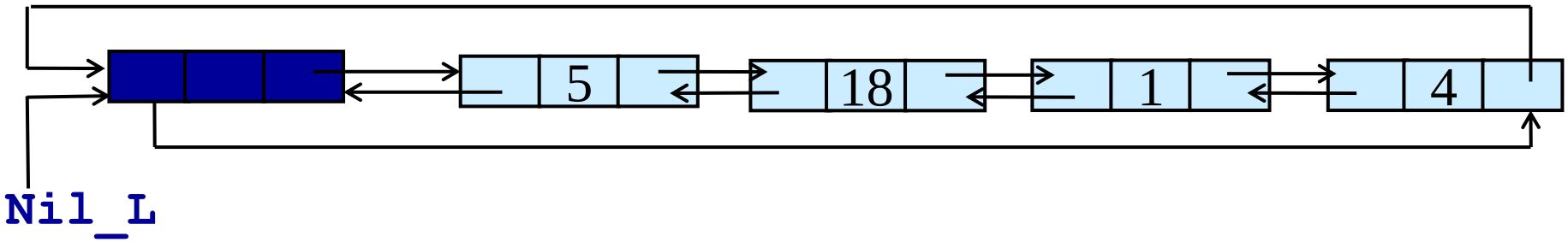
```
Algoritmo Lista-Cancella'(L,k)
   x = Lista-Cerca'(L,k)
   x->prev->next = x->next
   x->next->prev = x->prev
```



Operazioni su Liste con Sentinella

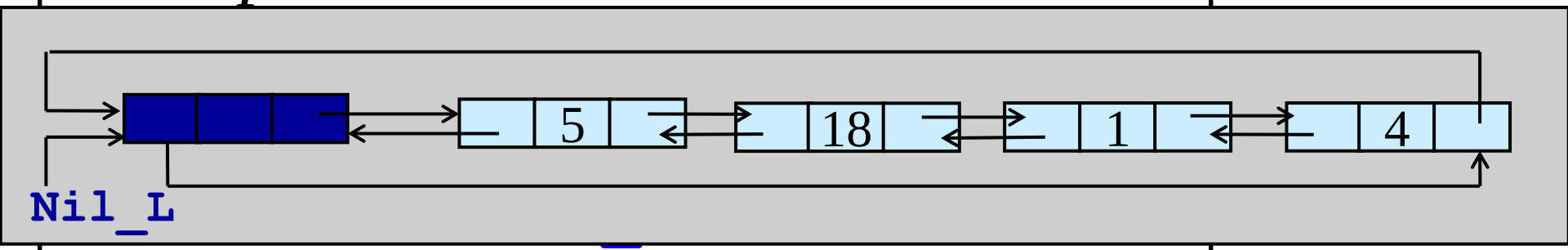
```
Algoritmo Lista-Cancella'(L,k)
   x = Lista-Cerca'(L,k)
   x->prev->next = x->next
   x->next->prev = x->prev
```

```
Algoritmo Lista-Inserisci'(L,x)
   x->next = Nil_L->next
   Nil_L->next->prev = x
   Nil_L->next = x
   x->prev = Nil_L
```



Operazioni su Liste con Sentinella

```
Algoritmo Lista-Cancella'(L,k)
   x = Lista-Cerca'(L,k)
   x->prev->next = x->next
```




```
   Nil_L->next->prev = x
   Nil_L->next = x
   x->prev = Nil_L
```

```
Algoritmo Lista-Cerca'(L,k)
   x = Nil_L->next
   WHILE x ≠ Nil_L and x->key ≠ k
      DO x = x->next
   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

Alberi

***Una Albero è un insieme dinamico* che**

- **è *vuoto* oppure**

- **è composto da *k+1 insiemi disgiunti* di nodi:**
  - **un insieme di cardinalità uno, detto *nodo radice***
  - ***k* alberi, ciascuno dei quali è detto *sottoalbero i-esimo* della radice (dove $1 \leq i \leq k$)**
- **Un tale albero si dice albero di *grado k***
- **Quando *k=2*, l'albero si dice binario**

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*

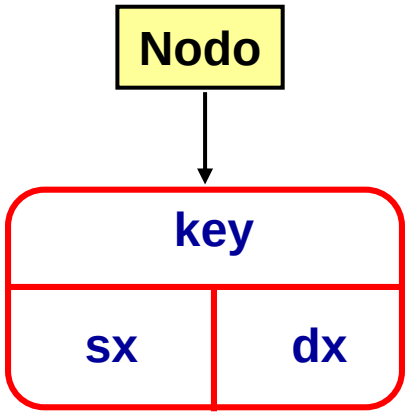
Visite in Profondità

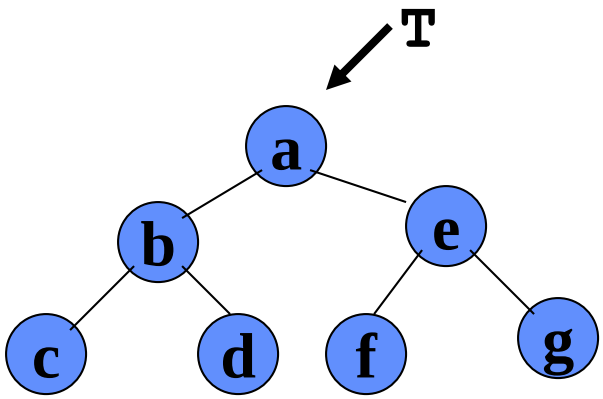
Gli alberi possono essere visitati (o attraversati) in profondità in diversi modi:

- ***Visita in Preordine: prima si visita il nodo e poi i suoi sottoalberi;***
- ***Visita Inordine (se binario): prima si visita il sottoalbero sinistro, poi il nodo e infine il sottoalbero destro;***
- ***Visita in Postordine : prima si visitano i sottoalberi, poi il nodo.***

Visita di Alberi Binari: in profondità preordine

```
Visita-Preordine(T)
    IF T ≠ NIL THEN
        Visita-Inordine(T->sx)
        "vista T"
        Visita-Inordine(T->dx)
```

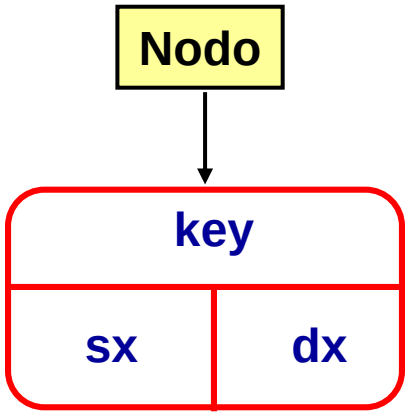


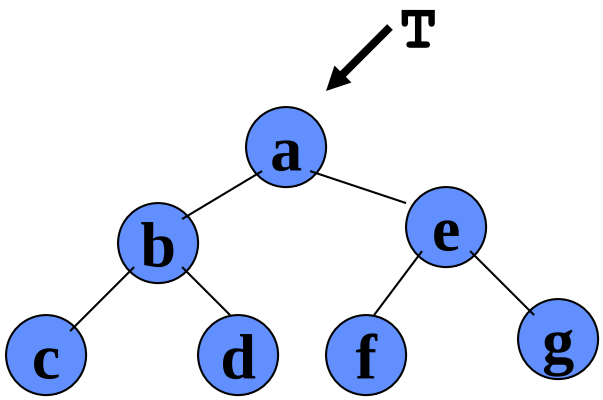


Sequenza: a b c d e f g

Visita di Alberi Binari: in profondità inordine

```
Visita-Inordine(T)
    IF T ≠ NIL THEN
        Visita-Inordine(T->sx)
        "vista T"
        Visita-Inordine(T->dx)
```

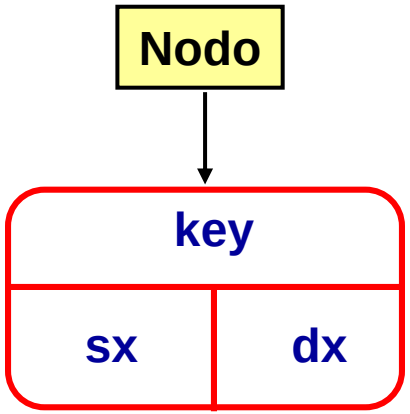


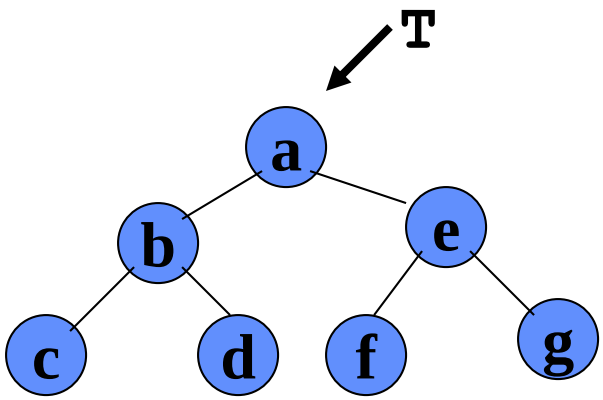


Sequenza: c b d a f e g

Visita di Alberi Binari: in profondità postordine

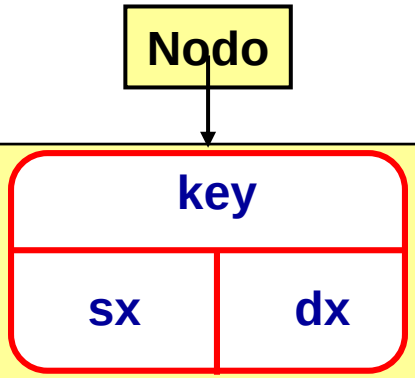
```
Visita-Postordine(T)
    IF T ≠ NIL THEN
        Visita-Postordine(T->sx)
        Visita-Postordine(T->dx)
        "vista T"
```





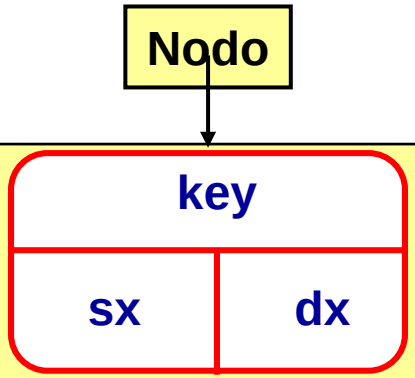
Sequenza: c d b f g e a

Visita Preorder Iterativa



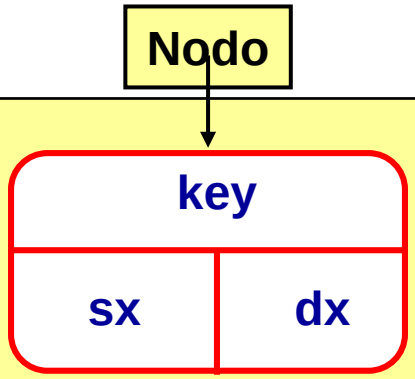

```
Visita-preorder-iter(T)
    stack = NIL
    curr = T
    WHILE (stack ≠ NIL OR curr ≠ NIL) DO
        IF (curr ≠ NULL) THEN /* Discesa a sx */
              "vista curr"
              push(stack,curr)
              curr = curr->sx
        ELSE    /* Risalita e discesa a dx */
             curr = top(stack)
             pop(stack)
             curr = curr->dx
```

Visita Inorder Iterativa



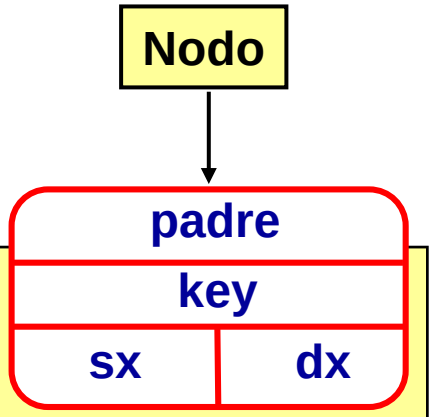

```
Visita-inorder-iter(T)
    stack = NIL
    curr = T
    WHILE (stack ≠ NIL OR curr ≠ NIL) DO
        IF (curr ≠ NULL) THEN /* Discesa a sx */
            push(stack,curr)
            curr = curr->sx
        ELSE /* Risalita e discesa a dx */
            curr = top(stack)
            pop(stack)
            "vista curr"
            curr = curr->dx
```

Visita Postorder Iterativa



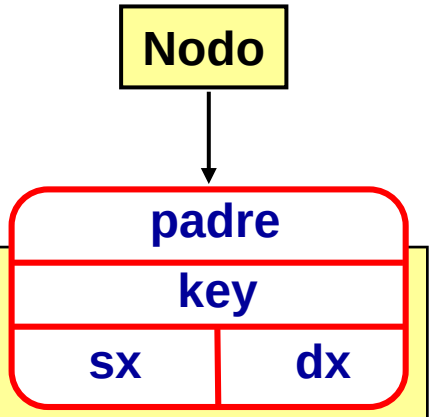

```
Visita-postorder-iter(T)
    stack = NIL, next = NIL
    curr = T, last = NIL
    WHILE (stack ≠ NIL OR curr ≠ NIL) DO
        IF (curr ≠ NIL) THEN /* Discesa a sx */
              stack = push(stack,curr)
              next = curr->sx
        ELSE
              curr = top(stack)  /* Risalita */
              IF (curr->dx ≠ NIL AND curr->dx ≠ last) THEN
                   next = curr->dx  /* Discesa a dx */
              ELSE       /* Risale da dx o dx vuoto */
                   "vista curr"
                   stack = pop(stack)
                   next = NIL
        last = curr  /* applica operazione di discesa */
        curr = next  /* o di risalita */
```

Visita Preorder Iterativa II



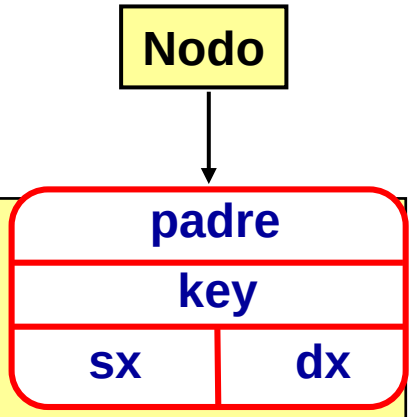

```
Visita-preorder-iter-2(T)
   curr = T
   last = NIL
   WHILE (curr ≠ NIL) DO
     IF (last = NIL) THEN
            "vista curr"
     IF (last = NIL AND curr->sx ≠ NIL) THEN
                curr = curr->sx /* Discesa a sx */
     ELSE IF (last ≠ curr->dx AND curr->dx ≠ NIL) THEN
            curr = curr->dx /* Discesa a dx */
            last = NIL
     ELSE    /* Risale da dx o dx vuoto */
              last = curr
              curr = curr->padre
```

Visita Preorder Iterativa II



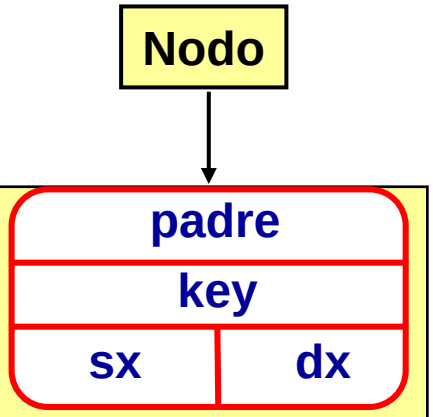

```
Visita-preorder-iter-2(T)
   curr = T
   last = NIL
   WHILE (curr ≠ NIL) DO
     IF (last = c->padre) THEN
           "vista curr"
     IF (last = c->padre AND curr->sx ≠ NIL) THEN
                  next = curr->sx /* Discesa a sx */
     ELSE  IF (last ≠ curr->dx AND curr->dx ≠ NIL) THEN
                  next = curr->dx /* Discesa a dx */
           ELSE   /* Risale da dx o dx vuoto */
                  next = curr->padre
     last = curr
     curr = next
```

Visita Inorder Iterativa II




```
Visita-inorder-iter-2(T)
    curr = T
    last = NIL
    WHILE (curr ≠ NIL) DO
      IF (last = NIL AND curr->sx ≠ NIL) THEN
            curr = curr->sx /* Discesa a sx */
      ELSE
          IF (last = curr->sx) THEN /* Risale da sx */
               "vista curr"
          IF(last = curr->sx AND curr->dx ≠ NIL)
               curr = curr->dx   /* Discesa a dx */
               last = NIL
          ELSE   /* Risale da dx */
               last = curr
               curr = curr->padre
```

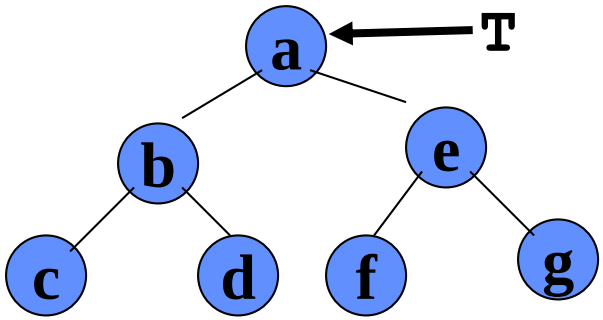
Visita Postorder Iterativa II




```
Visita-postorder-iter-2(T)
    curr = T
    last = NIL
    WHILE (curr ≠ NIL) DO
      IF (last = NIL AND curr->sx ≠ NIL) THEN
            curr = curr->sx    /* Discesa a sx */
      ELSE IF (last = curr->sx AND curr->dx ≠ NIL) THEN
            curr = curr->dx /* Discesa a dx */
            last = NIL
      ELSE    /* Risale da dx o dx vuoto */
              "vista curr"
              last = curr
              curr = curr->padre
```

Visita di Alberi k-ari: in ampiezza

```
Visita-Ampiezza(T)
   IF T ≠ NIL THEN
      "crea la coda vuota Q di dimensione ⌈((k−1)n)/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

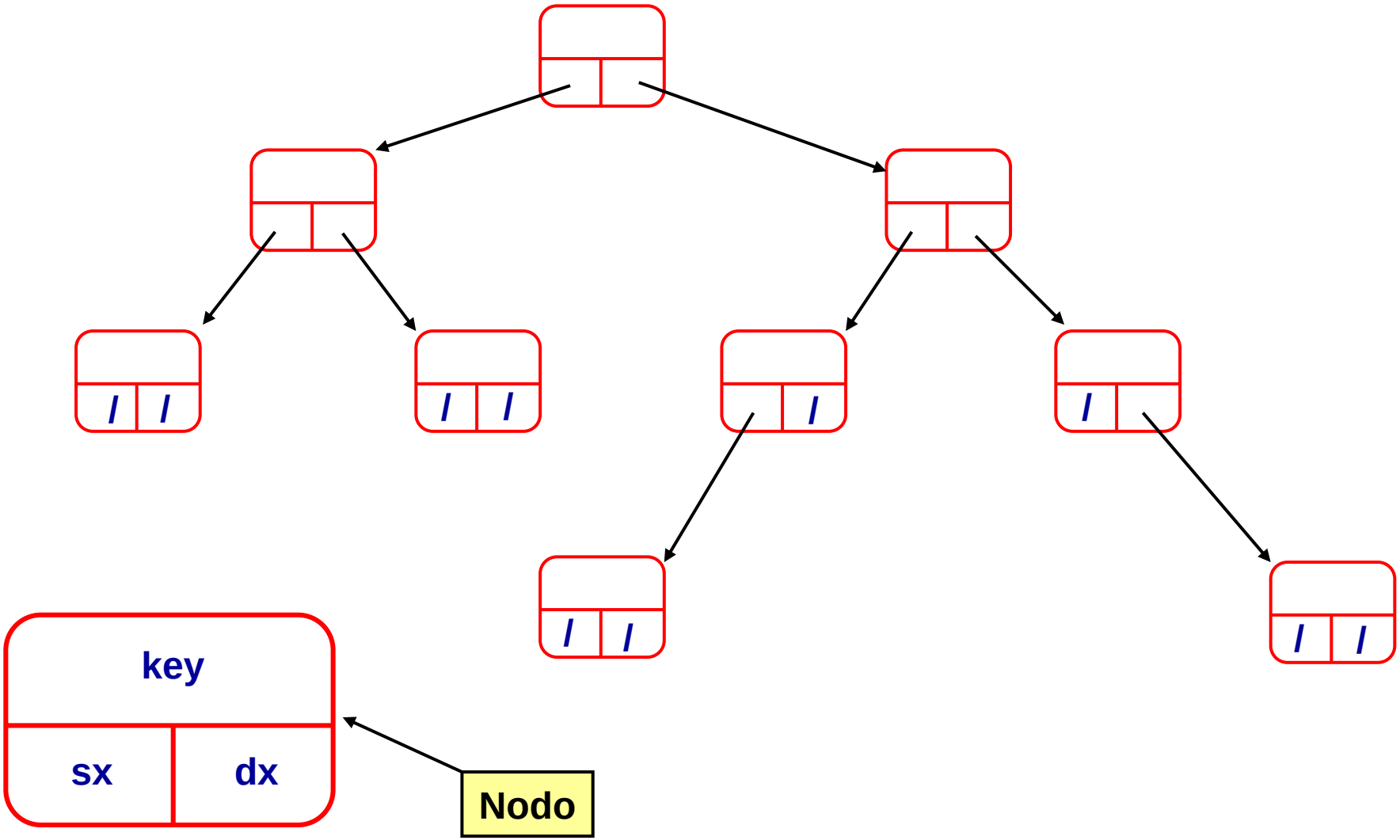
Implementazione di Alberi Binari

Come è possibile implementare 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, si possono utilizzare ad esempio opportuni array, simulando il meccanismo di gestione della memoria (allocazione, deallocazione)

Implementazione di Alberi Binari

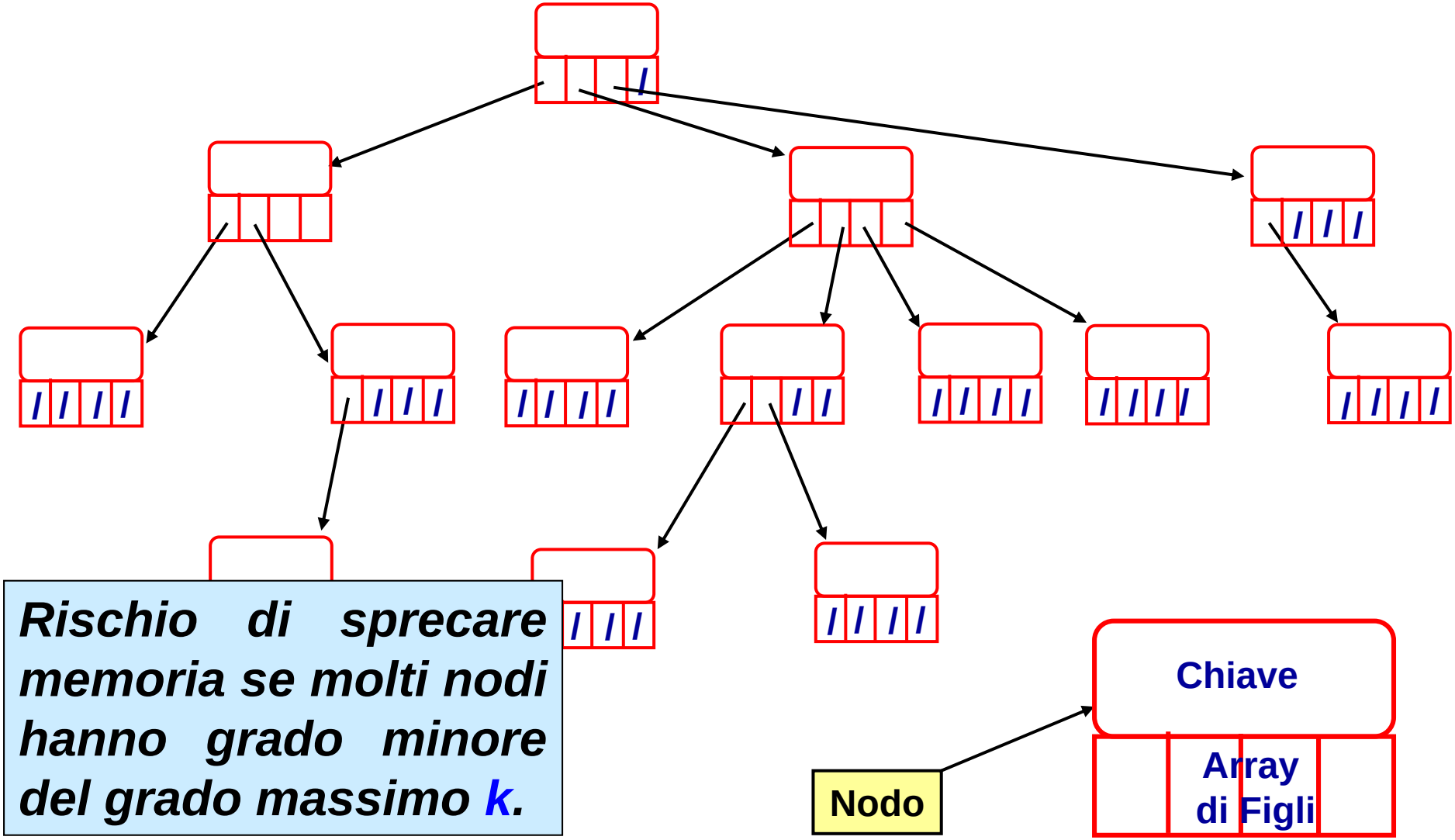


Ricerca in un albero binario

```
tree Search_Tree(T,k)
  if (T ≠ NIL)
     if (T->Key = k) then
         ris = T;
     else
         ris = Search_Tree(T->sx,k);
         if (ris = NIL) then
             ris = Search_Tree(T->dx,k);
return ris;
```

Indipendentemente dalla forma dell'albero, il tempo di ricerca è chiaramente lineare nel numero di nodi dell'albero nel caso peggiore (es. se la chiave non è presente).

Implementazione di Alberi Arbitrari



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

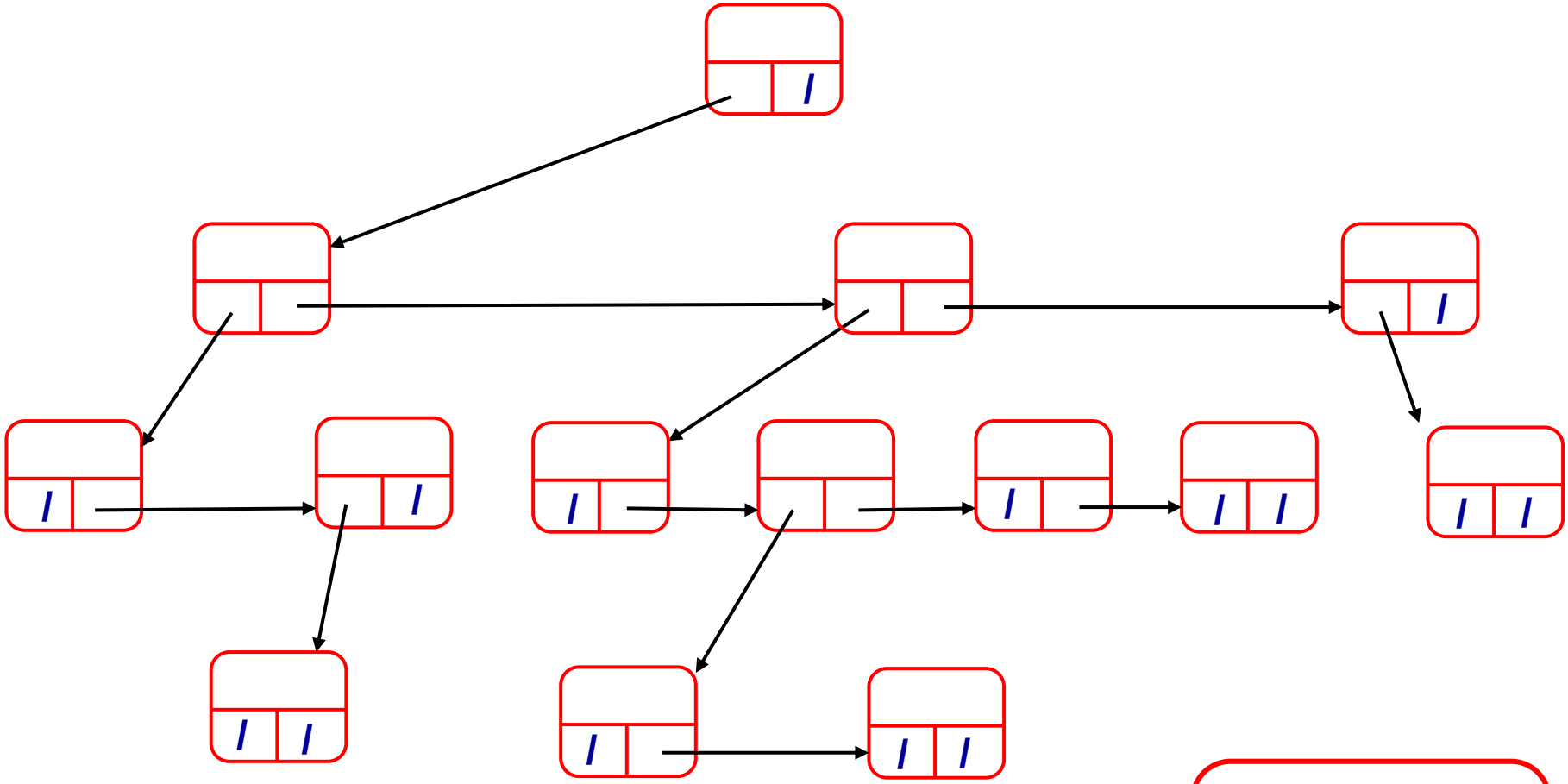
Ricerca in un albero generico

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

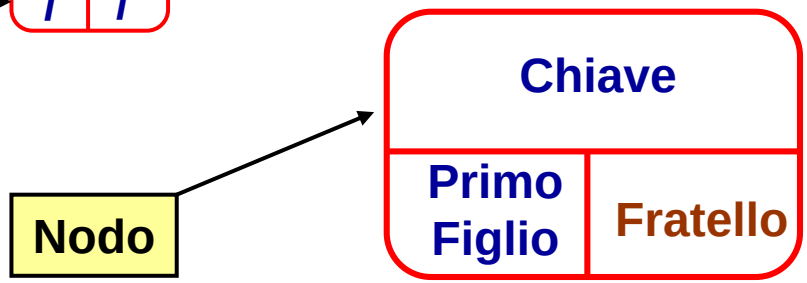
```
int Next_Son(T,id)
  s = id + 1;
  while (s ≤ MAX_SONS &&
           T->son[s] = NIL) do
       s = s + 1;
  if (s ≤ MAX_SONS)
      return s;
  else
      return 0;
```

Assumendo gli indici dell'array T->son da 1 a k

Implementazione di Alberi Arbitrari



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



Ricerca in un albero generico (basata su ricerca in alberi binari)

```
tree Search_Tree(T,k)
  if (T ≠ NIL)
     if (T->Key = k) then
         ris = T;
     else
         ris = Search_Tree(T->sx,k);
         if (ris = NIL) then
             ris = Search_Tree(T->dx,k);
return ris;
```

Implementazione di Puntatori

È possibile implementare 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.

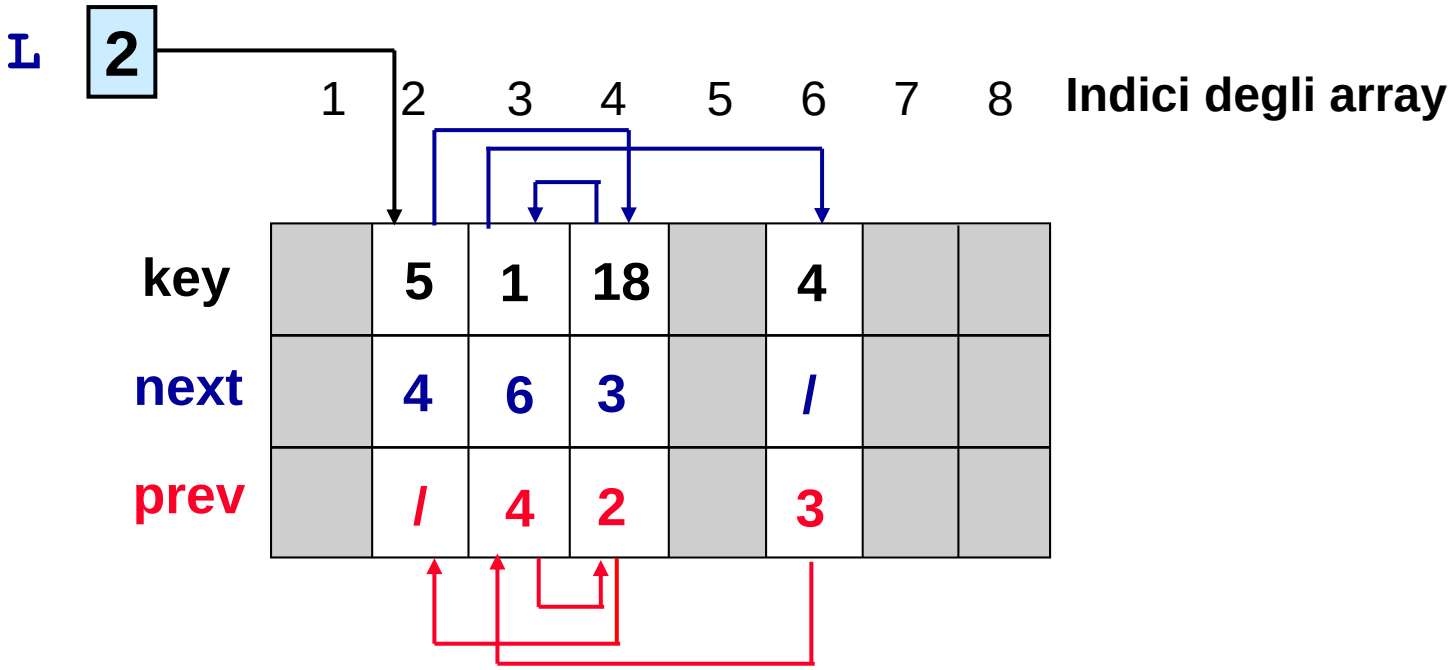
Ad esempio è possibile utilizzare array come contenitori di elementi di memoria.

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***

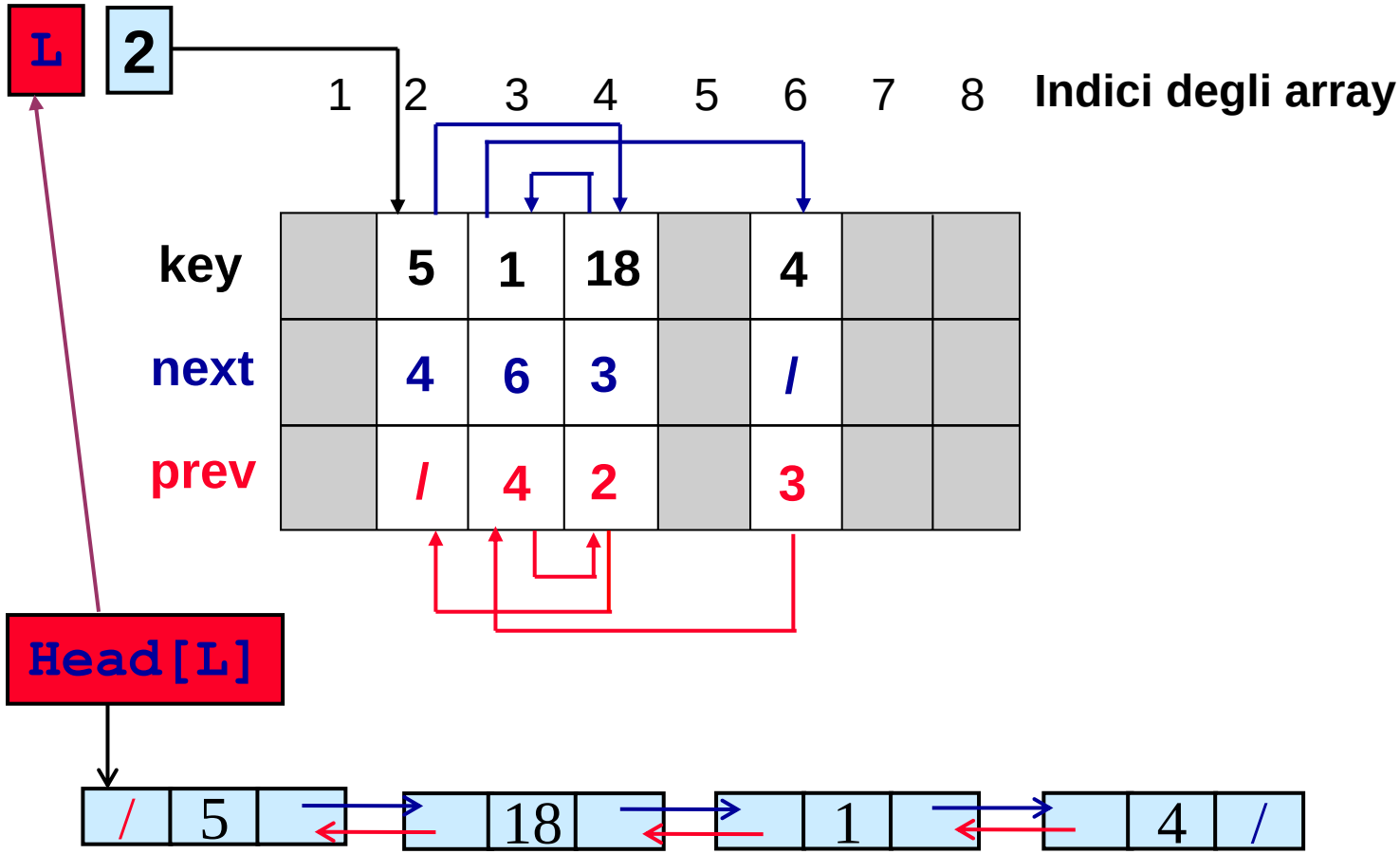
Implementazione di Puntatori

***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

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

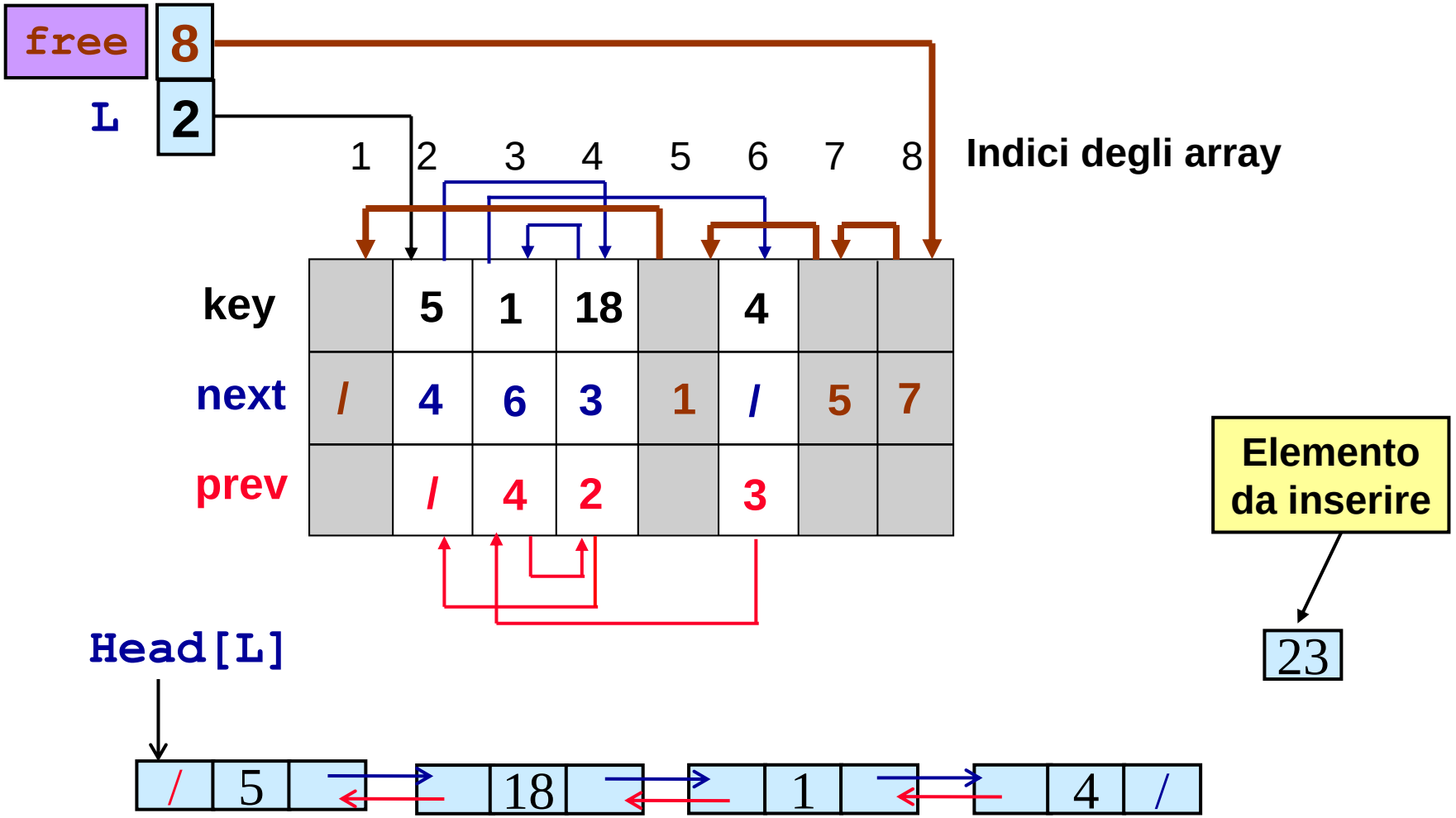
Possiamo usare:

- ***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)***

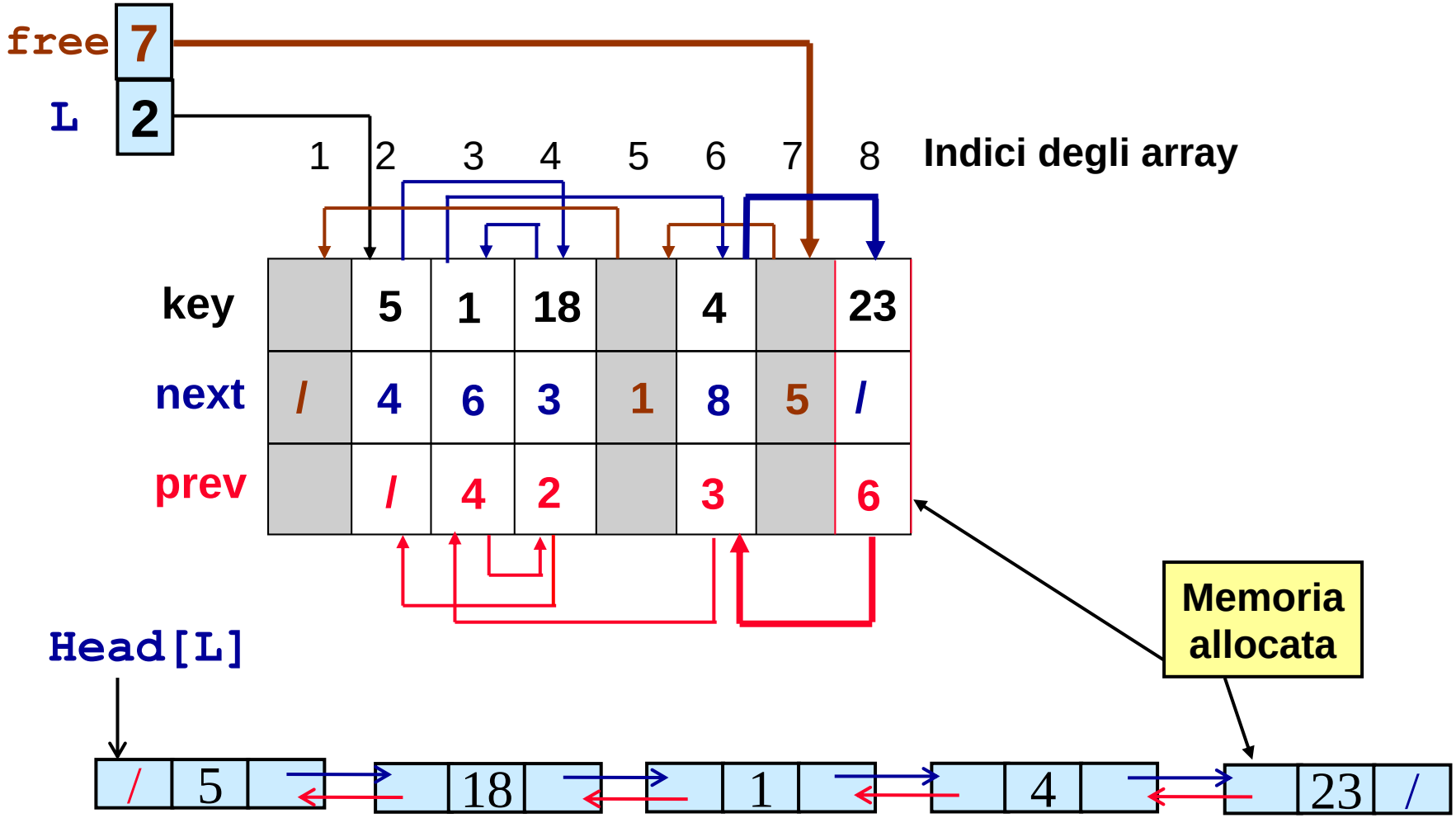
Allocazione memoria

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



Allocazione memoria

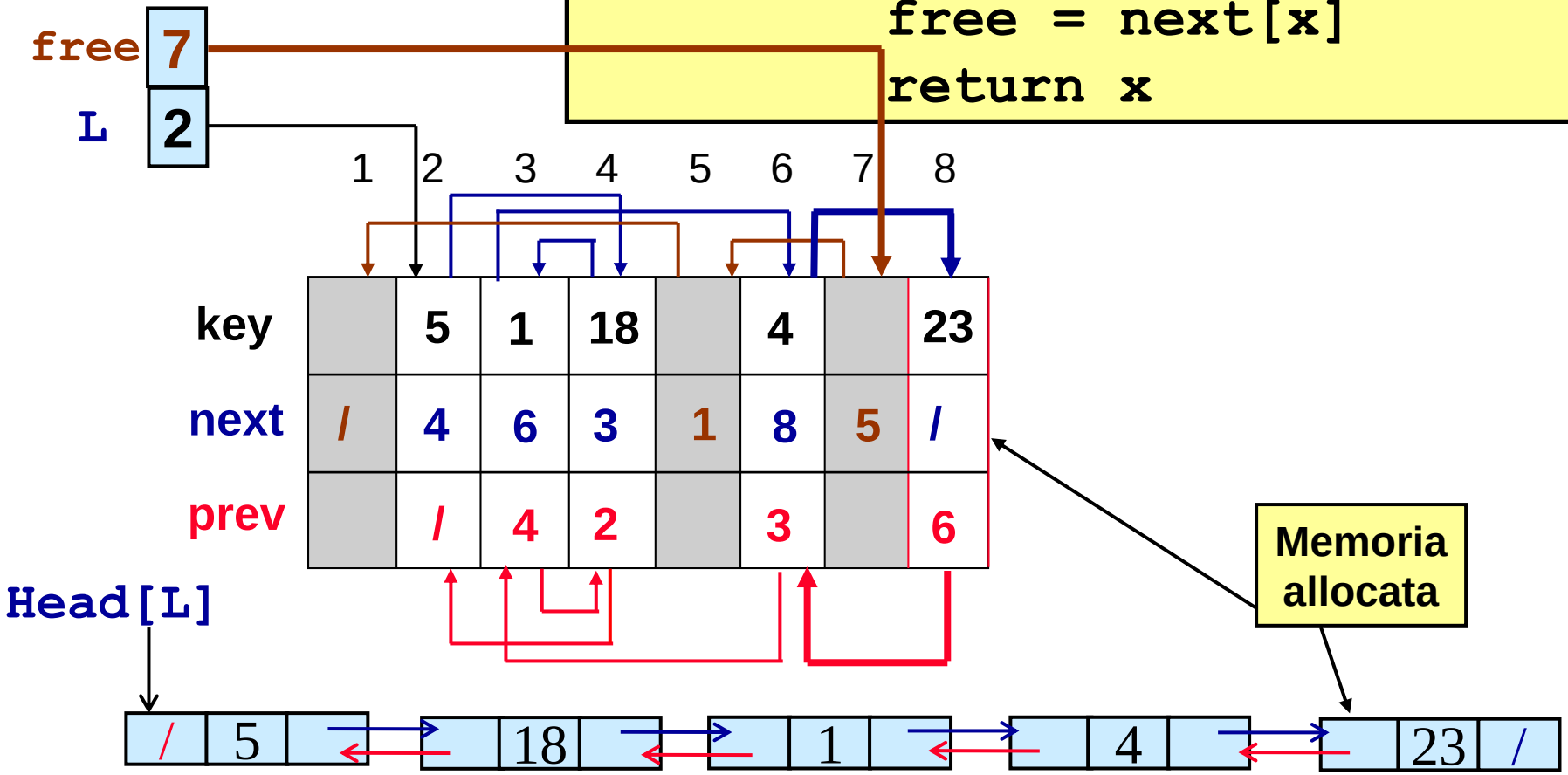
Implementazione di liste puntate doppie con tre array:* key[], next[] *e* prev[], free *è la free list



Allocazione memoria

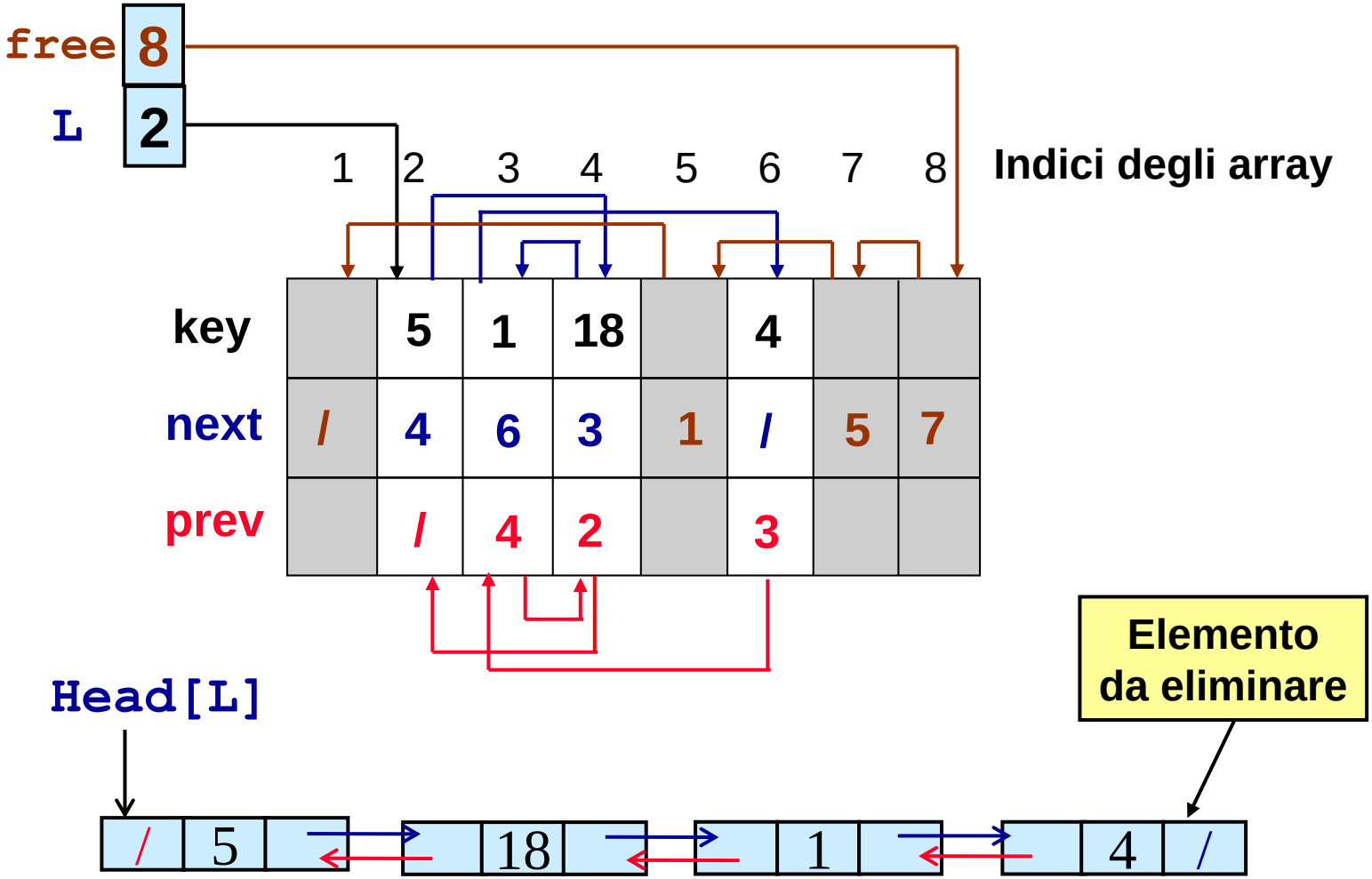
Implementazione array: key[], next

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



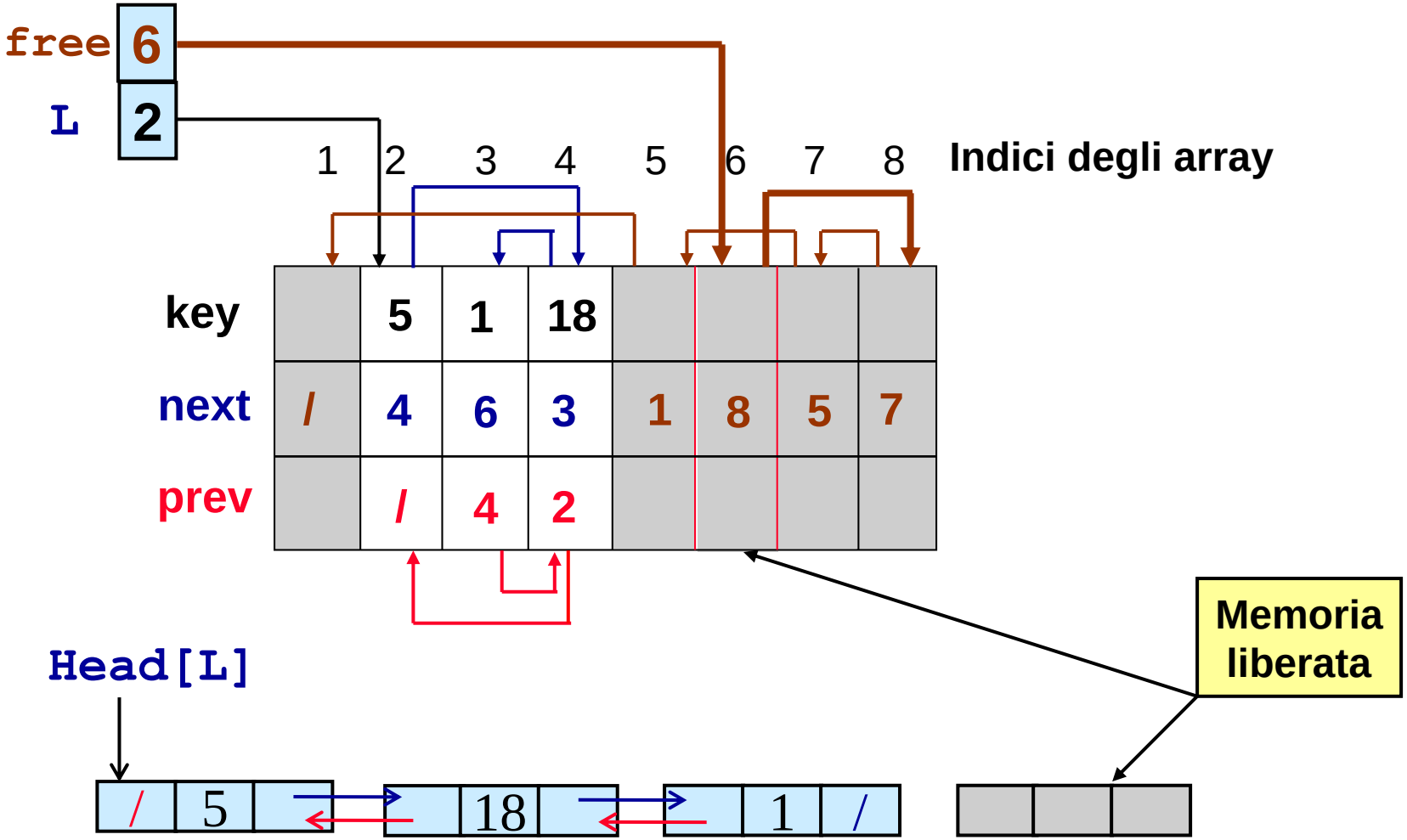
Deallocazione memoria

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
```