Strutture dati
Array statici e dinamici

Algoritmi e strutture dati
Lezione 7, a.a. 2016-17

Ugo de'Liguoro, Andras Horvath

Sommario

- Obiettivi
  – Capire in che modo la scelta delle strutture dati per rappresentare *insiemi dinamici* influenzi il tempo di accesso ai dati
- Argomenti
  – Array parzialmente riempiti
  – Array dinamici
  – Tempo ammortizzato (aggregato)

Insiemi dinamici

Studiamo strutture per rappresentare **insiemi dinamici**:

- numero finito di elementi
- gli elementi possono cambiare
- il numero degli elementi può cambiare
- si assume che ogni elemento ha un attributo che serve da **chiave**
- le chiavi sono tutte diverse

Insiemi dinamici, operazioni

Esistono due tipi di operazioni:

- interrogazione (query)
- modifiche

Operazione tipiche:

- inserimento (insert)
- ricerca (search)
- cancellazione (delete)

Insiemi dinamici, operazioni

Operazione tipiche in caso di chiavi estratte da insiemi totalmente ordinati

- ricerca del minimo (minimum)
- ricerca del massimo (maximum)
- ricerca del prossimo elemento più grande (successor)
- ricerca del prossimo elemento più piccolo (predecessor)

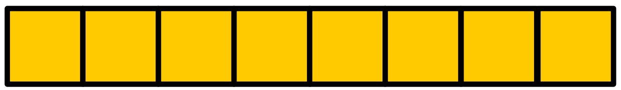
Complessità delle operazioni

- La **complessità**
  - è misurata in funzione della dimensione dell'insieme,
  - **dipende da che tipo di struttura dati si utilizza per rappresentare l'insieme dinamico**.
- Un'operazione molto costosa con una certa struttura dati può costare poco con un'altra.
- Quali operazioni sono necessarie dipende dall'applicazione.

Array

Un array è una sequenza di caselle:

- ogni casella può contenere un elemento dell'insieme
- le caselle hanno ognuna la stessa dimensione e sono allocate in memoria consecutivamente

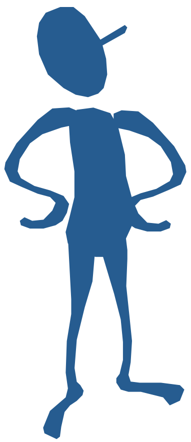


Array

base = indirizzo del primo elemento

$A[i]$ = valore all'indirizzo $base + i \cdot dim(valore)$

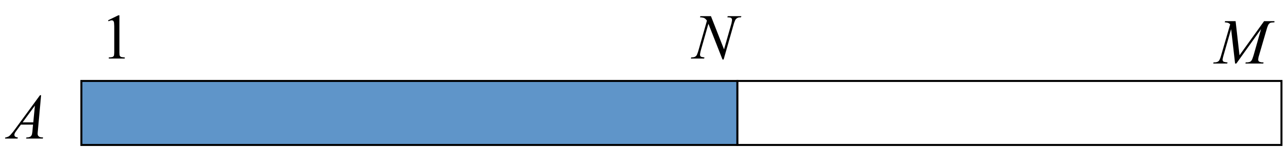
A	a	f	k	z	q	d	i	w
				i				



Con l'accesso diretto il tempo per leggere/scrivere in una cella è $O(1)$

Come rappresentare collezioni?

Una soluzione è l'array statico:



$A[i]$ elemento di posto i

$A.M$ dimensione dell'array

$A.N$ cardinalità della collezione

Invariante: $0 \leq N \leq M$

Array statico

Un array statico è un array in cui il **numero massimo di elementi è prefissato**:

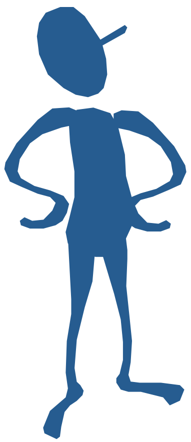
- *M* denota il numero massimo di elementi
- *N* denota il numero attuale di elementi
- gli *N* elementi occupano sempre le prime *N* celle del array

Ci interessa studiare

- quanto costano le varie operazioni
- quando conviene utilizzare questo tipo di array

Array statico: inserimento

L'inserimento ha costo $O(1)$



$\text{ARRAY-INSERT}(A, key)$

if $A.N < A.M$ **then**
    $A.N \leftarrow A.N + 1$
    $A[N] \leftarrow key$
    return $A.N$
else
    return nil
end if

Così si possono avere ripetizioni

Array statico: cancellazione

ARRAY-DELETE(A, key)

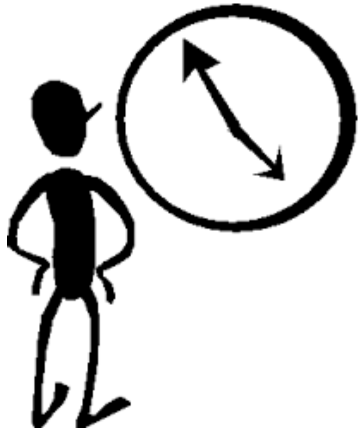
for $i \leftarrow 1$ to $A.N$ **do**
    if $A[i] = key$ **then**
        $A.N \leftarrow A.N - 1$
        for $j \leftarrow i$ to $A.N$ **do**
            $A[j] \leftarrow A[j+1]$
        end for
    end if
end for

Array statico: cancellazione

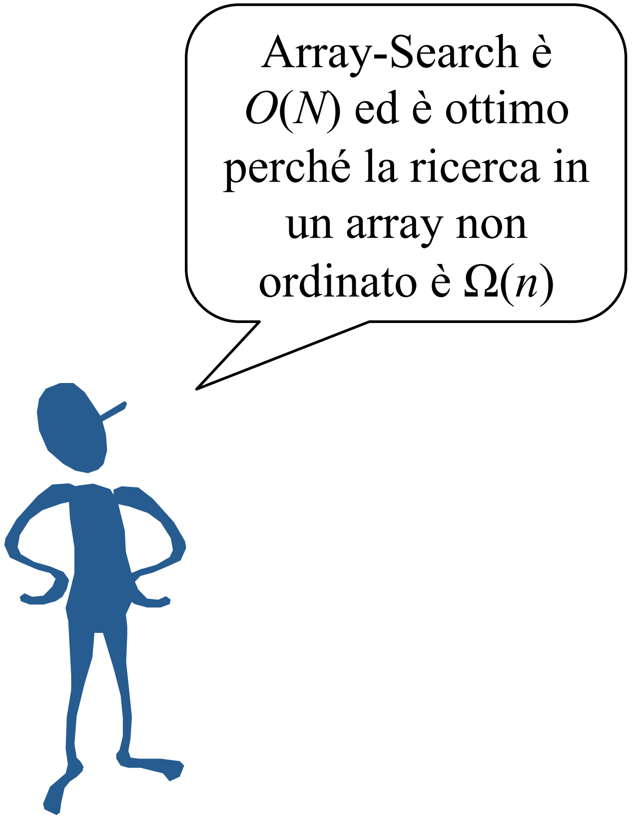
$\textsc{Array-Delete}(A, key)$

$deleted \leftarrow 0$
for $i \leftarrow 1$ to $A.N$ **do**
  if $A[i] = key$ **then**
    $deleted \leftarrow deleted + 1$
  else
    $A[i - deleted] \leftarrow A[i]$
  end if
end for
$A.N \leftarrow A.N - deleted$

Questa versione almeno ha tempo $O(N)$



Array statico: ricerca



$\textsc{Array-Search}(A, key)$

for $i \leftarrow 1$ to $A.N$ **do**

  if $A[i] = key$ **then**

    return i

  end if

end for

return nil

Array statico

Riassumendo:

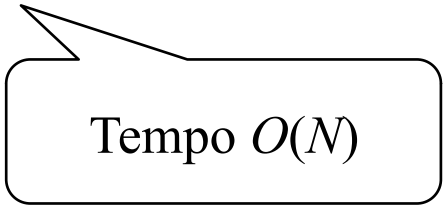
- Inserimento in tempo $O(1)$ (con ridondanza)
- Cancellazione in tempo $O(N)$
- Ricerca in tempo $O(N)$

E se l'array fosse ordinato?

Array statico

Possiamo avere:
- Ricerca in tempo $O(\log N)$ (usando Binary-Search)
- Cancellazione in tempo $O(N)$ (perché?)
- L'inserimento ?

Array statico: ins. ordinato



$\textsc{Array-InsertOrd}(A, key)$

if $A.N < A.M$ **then**
    $A.N \leftarrow A.N + 1$
    $A[N] \leftarrow key$
    $i \leftarrow N$
    while $i > 1 \ \wedge \ A[i-1] > A[i]$ **do**
            $\triangleright$ Inv. ?
        scambia $A[i-1]$ e $A[i]$
    end while
    return i
else
    return nil
end if

Array statico

- come si fa e quanto costa cercare il minimo e il massimo in un array ordinato?
- come si fa e quanto costa cercare il minimo e il massimo in un array non ordinato?
- come si realizzano e che complessità hanno le operazioni successor e predecessor in array ordinati e non ordinati?

Mantenendo l’array ordinato ci si guadagna in tutti i casi salvo in quello dell’inserimento

Array ridimensionabile

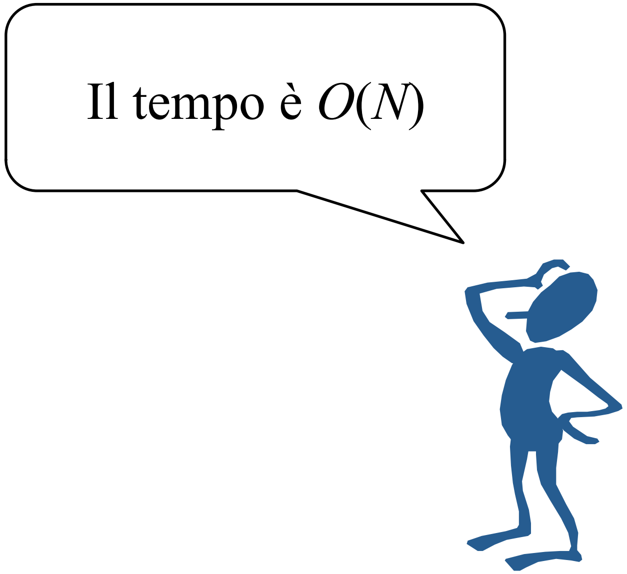
- cosa si può fare se non si conosce a priori il numero massimo di elementi (oppure se non si vuole sprecare spazio allocando molta più memoria del necessario)?
- si può "espandere" l'array quando risulti troppo piccolo …

Ma come si "espande" un array?

Array ridimensionabile

Array-Extend(A, n)

$B \leftarrow$ un array dimensione $A.M + n$
$B.M \leftarrow A.M + n$
$B.N \leftarrow A.N$
for $i \leftarrow 1$ **to** $A.N$ **do**
    $B[i] \leftarrow A[i]$
end for
return B



Array ridimensionabile

- prima idea:
  - allochiamo inizialmente spazio per *M* elementi (array di lunghezza *M*)
  - quando viene aggiunto un elemento, se l'array è pieno, espandiamo l'array di una cella
  - espandere costa tempo perché richiede di allocare memoria e copiare gli elementi dell'array

Array ridimensionabile

$\textsc{Dyn-Array-Insert-1}(A, key)$ $\quad\quad\triangleright$ Pre: $A.N \leq A.M$

if $A.N = A.M$ **then**
  $A \leftarrow \textsc{Array-Extend}(A, 1)$
end if
return $\textsc{Array-Insert}(A, key)$

Visto il costo di Array-Extend è chiaro che non conviene "espandere" l'array di 1 per ogni singolo inserimento

Abbiamo bisogno di un concetto nuovo di costo, che dipenda dalla "storia" degli inserimenti

Array ridimensionabile

- seconda idea:
  - problema della prima idea: se $N = M$ allora successivi inserimenti richiedono altrettante allocazioni
  - quando occorre, allocare più spazio di quanto strettamente necessario, in previsione di ulteriori inserimenti

Array ridimensionabile

- seconda idea in concreto:
  - inizialmente allochiamo un array di M elementi
  - quando l'array è pieno ne allochiamo uno nuovo di $2M$ elementi
  - quando il numero di elementi si riduce ad ¼ della dimensione, riallochiamo un array di dimensione ½ M

Array ridimensionabile

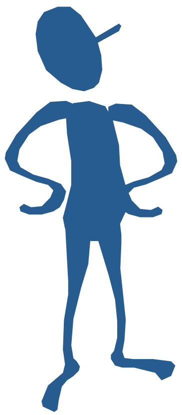
$\textsc{Dyn-Array-Insert-2}(A, key)$ $\qquad\qquad \triangleright$ Pre: $A.N \leq A.M$

if $A.N = A.M$ **then**

$\quad A \leftarrow \textsc{Array-Extend}(A, 2 \cdot A.M)$

end if

return $\textsc{Array-Insert}(A, key)$



Qui sembra esservi
un'investimento pagato in spazio
per un guadagno futuro in tempo

Array ridimensionabile

$\textsc{Dyn-Array-Delete}(A, key)$

$\textsc{Array-Delete}(A, key)$
if $A.N \leq 1/4 \cdot A.M$ **then**
  $B \leftarrow$ un array dimensione $A.M/2$
  $B.M \leftarrow A.M/2$
  $B.N \leftarrow A.N$
  for $i \leftarrow 1$ **to** $A.N$ **do**
    $B[i] \leftarrow A[i]$
  end for
  $A \leftarrow B$
end if

Tempo ammortizzato

Per confrontare diverse soluzioni nella realizzazione di ADT si valutano i tempi di una *sequenza* di operazioni, che determinano ciascuna la dimensione dell'ingresso della successiva

Tempo ammortizzato

Per confrontare diverse soluzioni nella realizzazione di ADT si valutano i tempi di una *sequenza* di operazioni, che determinano ciascuna la dimensione dell'ingresso della successiva

$$T_{amm}(m) = \frac{T_1(n_1) + \cdots + T_m(n_m)}{m}$$

Idea: attribuiamo a ciascuna operazione la media del costo totale: ***metodo dell'aggregazione***

Cormen, par. 17.1

Tempo ammortizzato

$T_i\,(n_i)$ tempo dell'i-esima operazione

$$T_{amm}(m) = \frac{T_1(n_1) + \cdots + T_m(n_m)}{m}$$

Num. delle operazioni nella seq.

Dimensione dei dati in ingresso dell'm-esima op.

Array ridim. Tempo ammortizzato

$$T^{(1)}_{amm}(m) = \frac{1}{m}\sum_{i=1}^{m} T_{D\text{-}Ins}(i) = \frac{1}{m}\sum_{i=1}^{m} O(i) = \frac{O(m^2)}{m} = O(m)$$

Con Dyn-Array-Insert-1 il tempo ammortizzato di ogni inserimento è lineare

Array ridim. Tempo ammortizzato

$$\begin{cases} T_{D\text{-}Ins}(2^i + 1) & = & 2^{i+1} \\ T_{D\text{-}Ins}(2^i + 2) & = & \cdots \quad = T_{D\text{-}Ins}(2^{i+1}) = O(1) \end{cases}$$

$$\begin{aligned} T^{(2)}_{amm}(m) &= \tfrac{1}{m} \textstyle\sum_{i=1}^{m} T_{D\text{-}Ins}(i) \\ &= \tfrac{1}{m} \textstyle\sum_{i=0}^{\log_2 m - 1} (2^i + O(1)) = \tfrac{1}{m} \left(\textstyle\sum_{i=0}^{\log_2 m - 1} 2^i + O(\log_2 m) \right) \\ &\leq \tfrac{1}{m} \cdot \tfrac{2^{\log_2 m} - 1}{2 - 1} + \tfrac{O(m)}{m} \\ &= \tfrac{m-1}{m} + O(1) \\ &= \tfrac{O(m)}{m} = O(1) \end{aligned}$$

Con Dyn-Array-Insert-2 il tempo
ammortizzato di ogni inserimento
è costante!

Liste

Algoritmi e strutture dati
Lezione 8, a.a. 2016-17

Ugo de'Liguoro, Andras Horvath

Sommario

- Obiettivi
  - Studiarele liste e gli algoritmi sulle liste dal punto di vista della loro complessità
- Argomenti
  - Liste semplici
  - Inserimento, ricerca e cancellazione
  - Copia ed inversione
  - Ordinamento

Che cosa è una lista

Una lista è una sequenza finita di valori:

$$L = [a_1, \dots , a_k]$$

dove $a_1, \dots , a_k \in A$, per qualche insieme A.

Se $k = 0$, allora $L = []$, o anche $L = nil$, è la lista vuota.

Le parti di una lista

Sia $k \neq 0$:

$$L = [\underbrace{a_1}, \underbrace{a_2, \ldots, a_k}]$$

$Head(L) = a_1$ $\qquad$ $Tail(L) = [a_2, \ldots, a_k]$

$$Cons(a, [a_1, \ldots, a_k]) = [a, a_1, \ldots, a_k]$$

$Cons(Head(L), Tail(L)) = L$ per ogni $L \neq nil$

Una definizione induttiva

Fissato un insieme di elementi A, l'insieme delle liste su A, List(A) o semplicemente List, è il più piccolo tale che:

- [] (ovvero nil) $\in$ List
- se a $\in$ A, L $\in$ List allora Cons(a, L) $\in$ List

Confrontiamo questa definizione con quella dei numeri naturali, Nat, che è il più piccolo insieme tale che:

- 0 $\in$ Nat
- se n $\in$ Nat allora Succ(n) $\in$ Nat (dove Succ(n) = n + 1)

Il tipo delle liste

In C++ definiremo un tipo List di valori di tipo T (negli esempi T = int); possiamo allora definire il tipo delle costanti e degli operatori sulle liste:

- nil : List (nil ha tipo List)
- Cons : $T \times List \rightarrow List$
- Head : $List \rightarrow T$ (Head(nil) = $\perp$)
- Tail : $List \rightarrow List$ (Tail(nil) = $\perp$)

$\perp$ = indefinito

Algebra delle liste

A questi operatori ne possiamo aggiungere altri definendoli ricorsivamente secondo lo schema

$$f(nil, x_1, \ldots , x_k) = g(x_1, \ldots , x_k)$$

$$f(Cons(z, L), x_1, \ldots , x_k) = h(f(L, x_1, \ldots , x_k), z, L, x_1, \ldots , x_k)$$

supposte note g ed h.

Ricorsione primitiva sulle liste

Algebra delle liste

Esempi:

IsEmpty : List → Bool (predicato "L è vuota")

IsEmpty (nil) = true
IsEmpty (Cons(z, L)) = false

Length : List → Int (funzione lunghezza)

Length (nil) = 0
Length (Cons(z, L)) = 1 + Length(L)

Algebra delle liste

Dati gli operatori Head e Tail ed il predicato IsEmpty, e supposto di avere un operatore a tre posti

$$\text{if-then-else} : \text{Bool} \times T \times T \rightarrow T \quad \text{per ogni } T$$

possiamo trasformare lo schema di ricorsione in una definizione più vicina al codice C++:

$f(L, x_1, \dots, x_k) =$

$\quad$ if IsEmpty (L) then $g(x_1, \dots, x_k)$

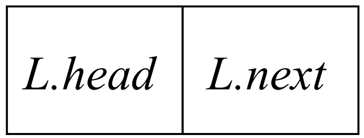
$\quad$ else $h(f(\text{Tail}(L), x_1, \dots, x_k), \text{Head}(L), \text{Tail}(L), x_1, \dots, x_k)$

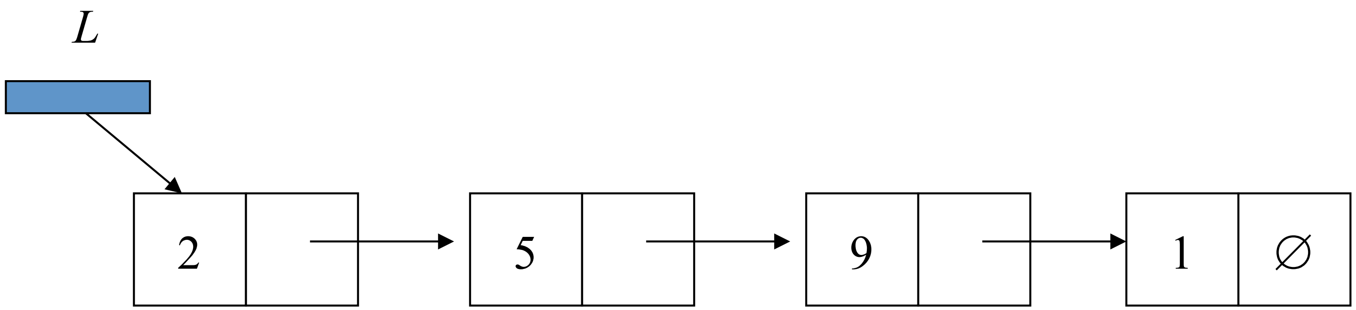
Esempio:

$\quad$ Length(L) = if IsEmpty (L) then 0 else 1 + Length(Tail(L))

Le liste semplici

Come struttura dati una lista è una sequenza di record, ciascuno dei quali contiene un campo che punta al successivo:



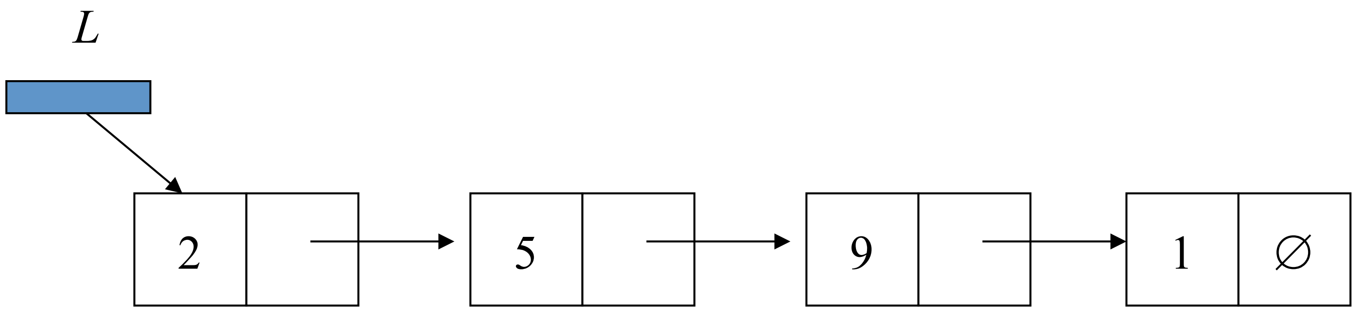


Le liste semplici

In C/C++:

```
typedef struct Nodo* List;
struct Nodo
{     T info;
      List next;
};
```

List è ricorsivo



Liste semplici

… o equivalentemente:

```
struct Nodo
{     T info;
      Nodo* next;
};
typedef Nodo* List;
```

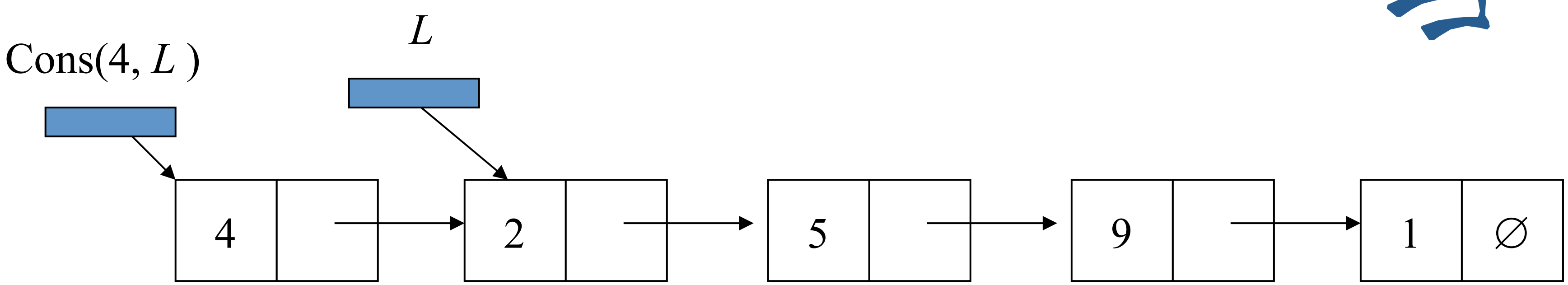
La funzione Cons

$\textsc{Cons}(x, L)$

$N.head \leftarrow x$

$N.next \leftarrow L$

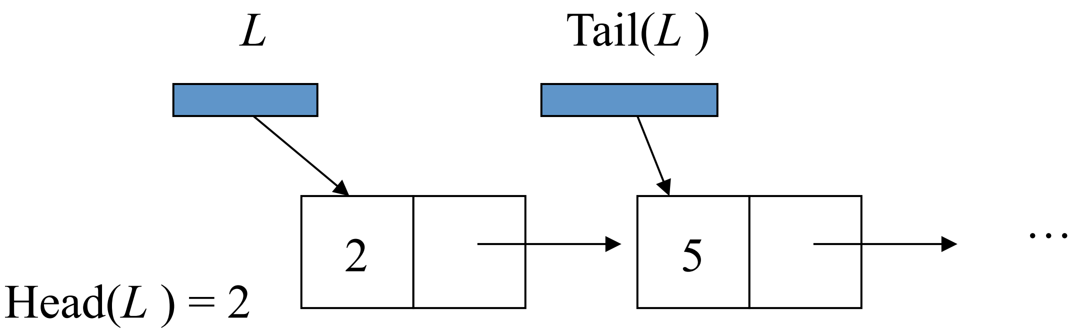
$\mathbf{return}\ N$



Le funzioni Head e Tail

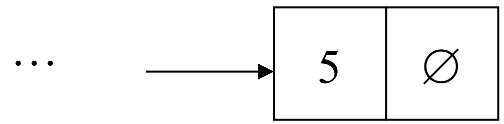
HEAD(L)
return $L.head$

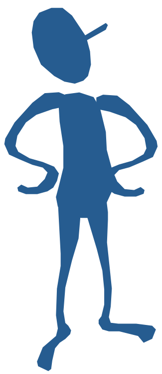
TAIL(L)
return $L.next$



La lista vuota

$\textsc{IsEmpty}(L)$
return $L = nil$





Al fondo di ogni lista c'è la lista vuota, ossia un puntatore a *nil*

$\textsc{TheEmptyList}()$
return nil

Lunghezza di una lista

Versione ricorsiva:

$\textsc{Length-Rec}(L)$

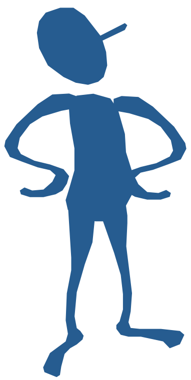
if $IsEmpty(L)$ **then**

  return 0

else

  return $1 + \textsc{Length-Rec}(Tail(L))$

end if



Lunghezza di una lista

LENGTH-TAIL-REC(L, n) $\triangleright$ post: ritorna $length(L) + n$

if ISEMPTY(L) **then**
    return n
else
    return LENGTH-TAIL-REC(TAIL$(L), n + 1)$
end if

LENGTH-ITER(L)

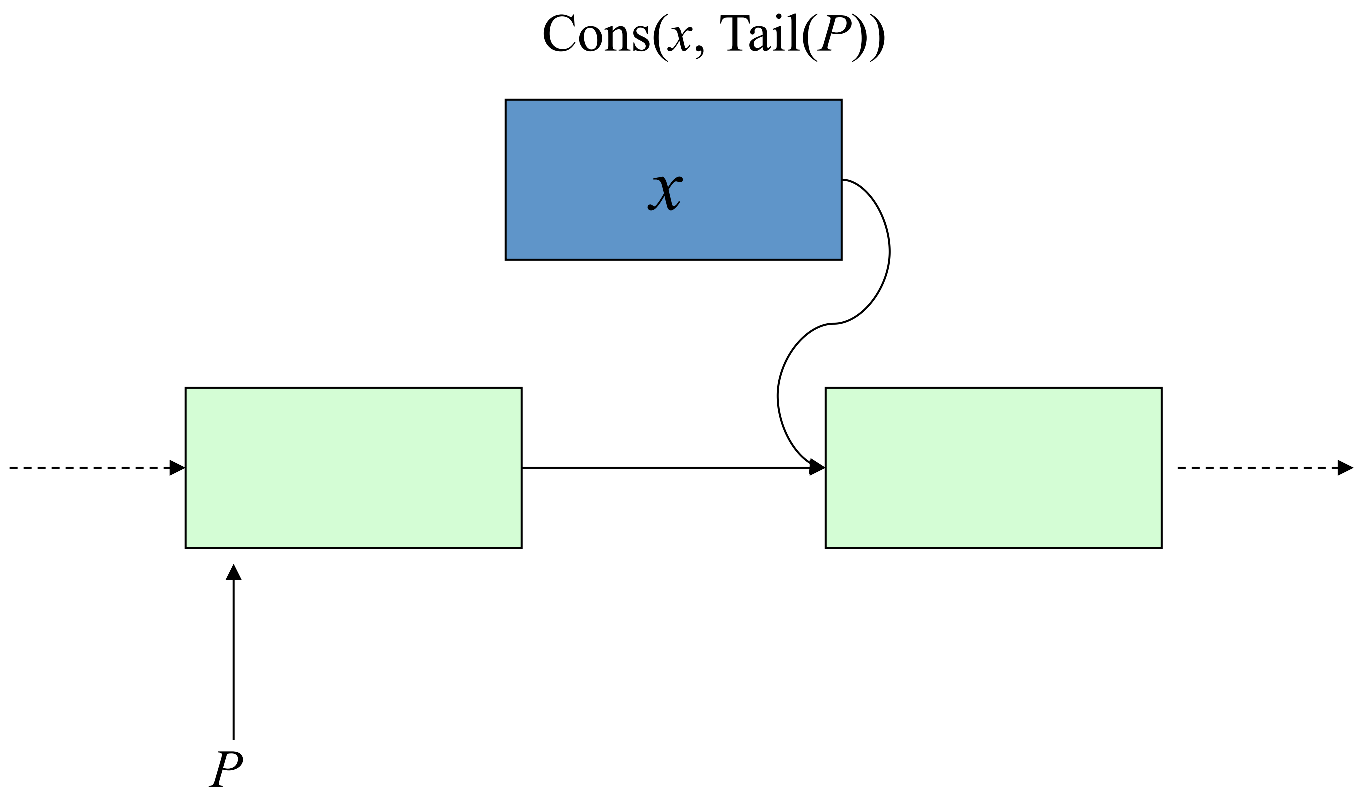
$n \leftarrow 0$
while not ISEMPTY(L) **do**
    $L \leftarrow$ TAIL(L)
    $n \leftarrow n + 1$
end while
return n

Ricerca in una lista

$\textsc{Search}(x, L)$
if $\textsc{IsEmpty}(L)$ **then**
    return false
else
    return $\textsc{Head}(L) = x$ **or** $\textsc{Search}(x, \textsc{Tail}(L))$
end if

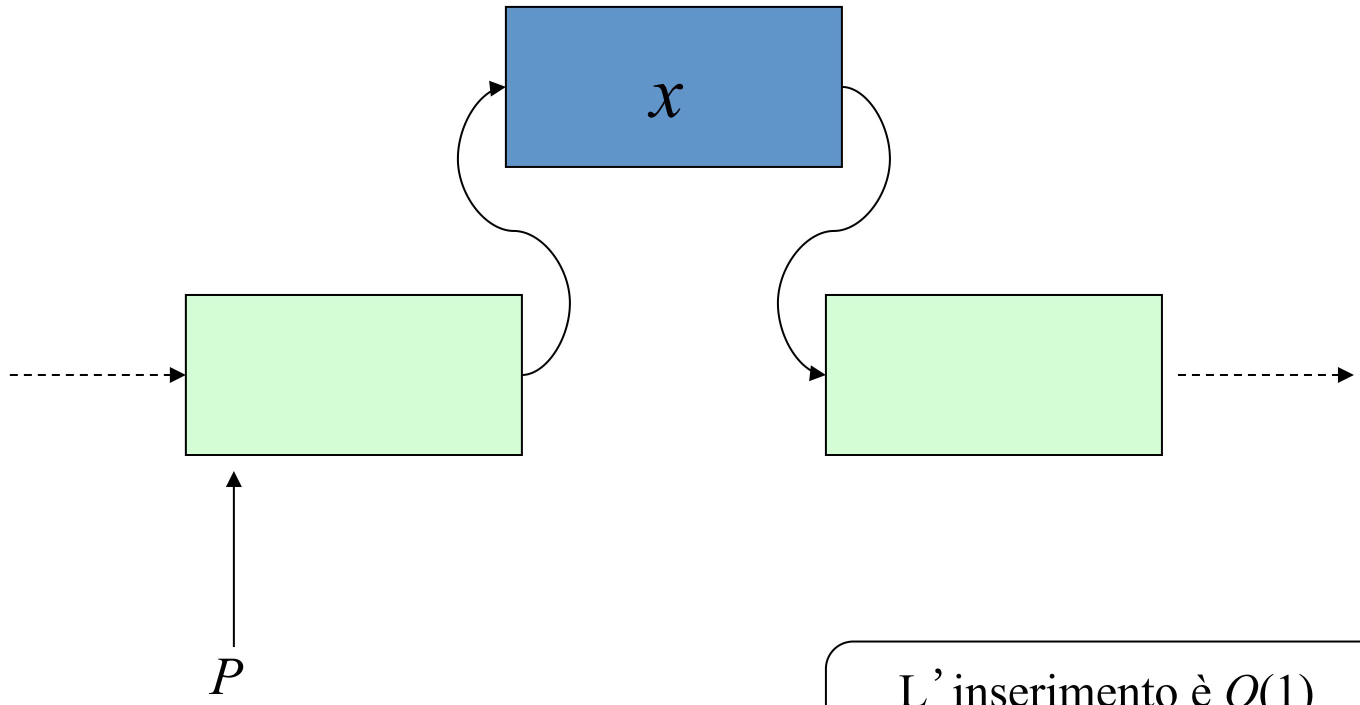
Se $L \neq \emptyset$ allora $x \in L$ solo se
$x = \text{Head}(L)$ oppure $x \in \text{Tail}(L)$

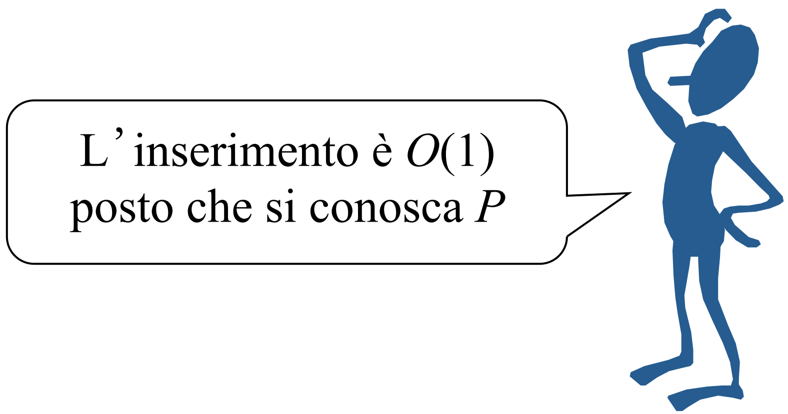
Inserimento in una lista



Inserimento in una lista

$$P.next = \mathrm{Cons}(x, \mathrm{Tail}(P))$$





Un metodo ricorsivo

$\textsc{Insert}(x, i, L)$

    ▷ post: x inserito davanti all'el. di posto i in L

if $i = 1$ **then**

    return $\textsc{Cons}(x, L)$

else

    $L.next \leftarrow \textsc{Insert}(x, i-1, \textsc{Tail}(L))$

    return L

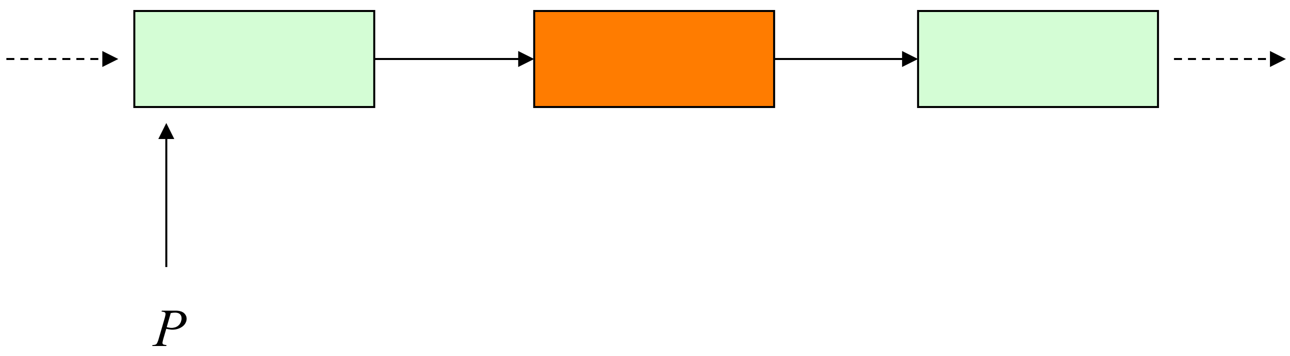
end if

Come sarà la versione iterativa?

Cancellazione da una lista

$temp \leftarrow \mathrm{Tail}(P)$

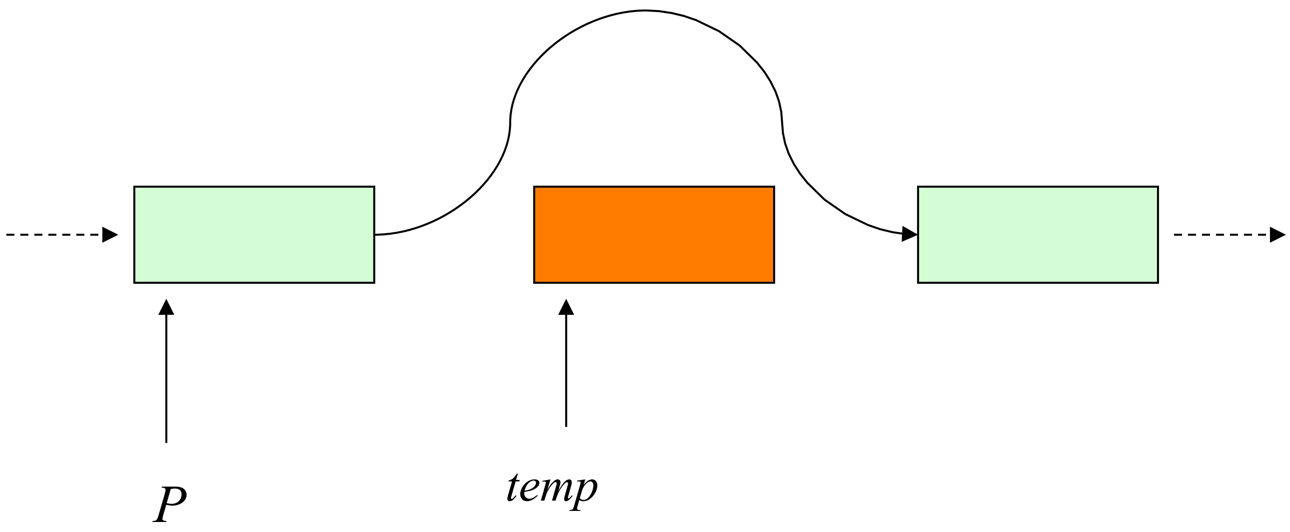
$P.next \leftarrow \mathrm{Tail}(\mathrm{Tail}(P))$



Cancellazione da una lista

$$temp \leftarrow \mathrm{Tail}(P)$$

$$P.next \leftarrow \mathrm{Tail}(\mathrm{Tail}(P))$$



Cancellazione da una lista

```
DeleteAll(x, L)
    ▷ post: ogni occ. di x è rimossa da L
if IsEmpty(L) then
    return nil
else
    if x = Head(L) then
        return DeleteAll(x, Tail(L))
    else
        L.next ← DeleteAll(x, Tail(L))
        return L
    end if
end if
```

Altri algoritmi sulle liste

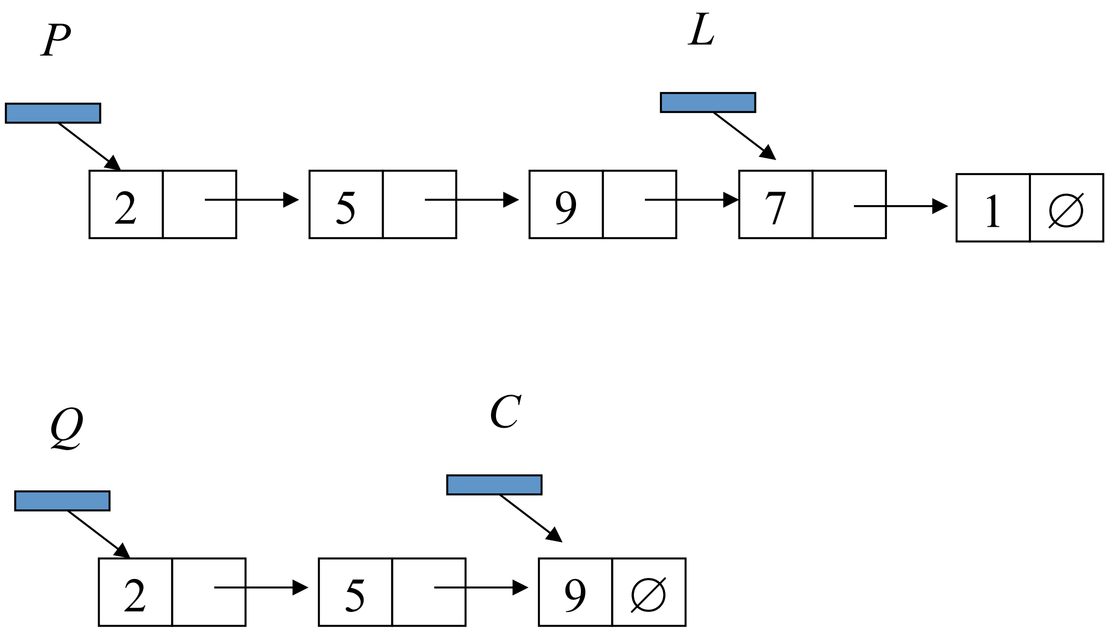
Per apprendere come manipolare (ricorsivamente) le liste consideriamo:

- Copia di una lista
- Inversione dell'ordine degli elementi in una lista
- Ricerca, inserimento e cancellazione in una lista ordinata
- Ordinamento di una lista per fusione (Merge-Sort), dopo averla divisa in due metà (circa)

Copia di una lista

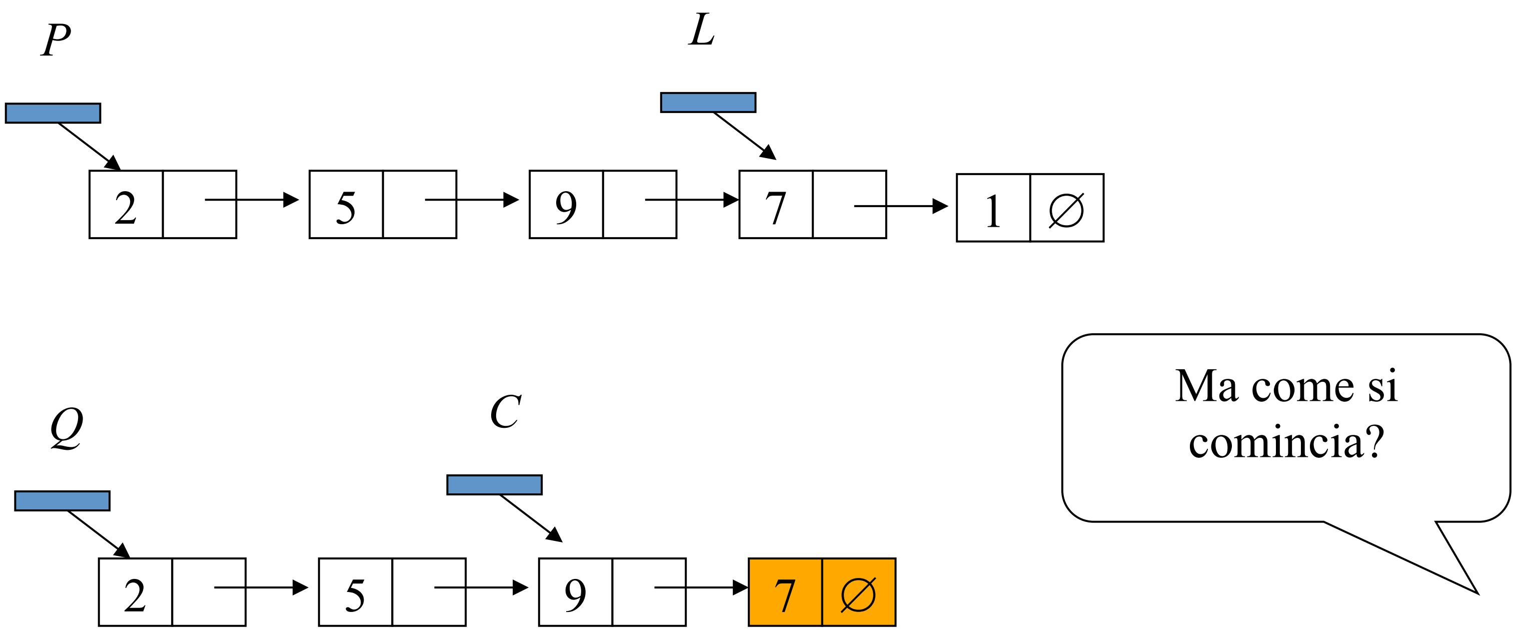
- Una lista è una struttura dati persistente, che può subire modifiche durante l'esecuzione di una procedura
- È allora utile poter duplicare una lista quando si vuole che l'effetto delle modifiche sia solo temporaneo

Copia di una lista



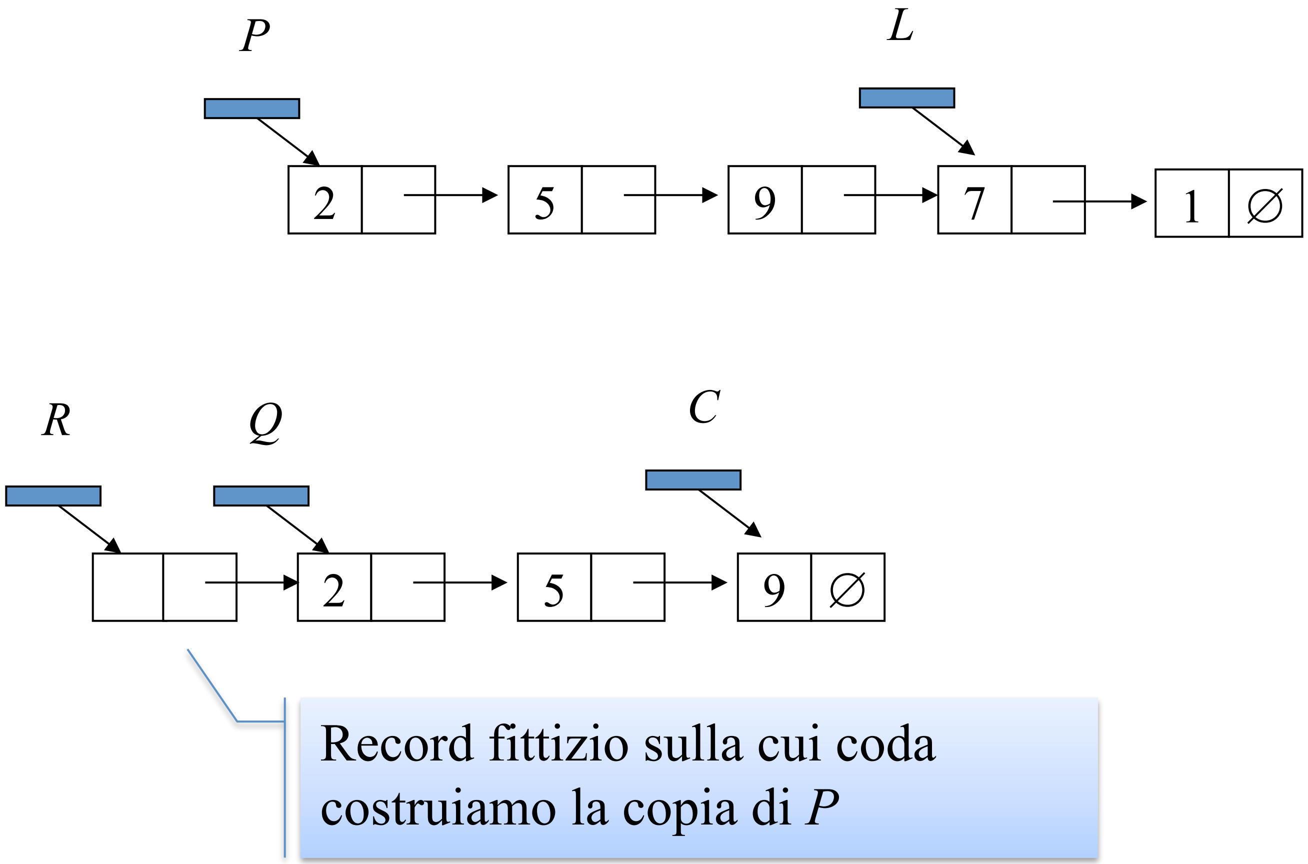
$$C.next = \text{Cons}(\text{Head}(L), nil)$$

Copia di una lista

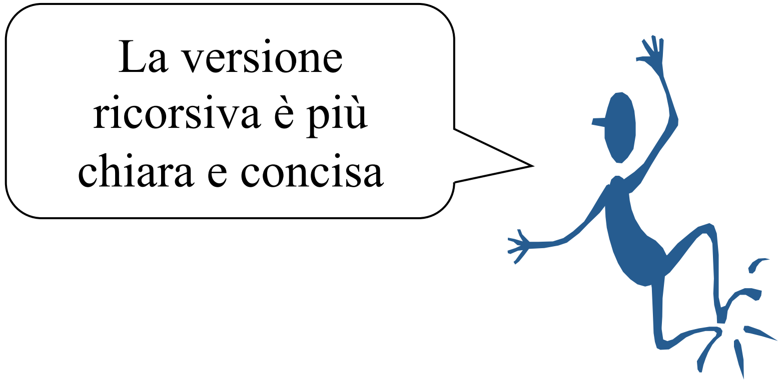


$$C.next = \text{Cons}(\text{Head}(L), nil)$$

Copia di una lista



Copia di una lista



$\textsc{Clone-Iter}(L)$

$R \leftarrow \textsc{Cons}(-\,, nil)$
$C \leftarrow R$
while $L \neq nil$ **do**
  $C.next \leftarrow \textsc{Cons}(\textsc{Head}(L), nil)$
  $C \leftarrow \textsc{Tail}(C)$
  $L \leftarrow \textsc{Tail}(L)$
end while
return $\textsc{Tail}(R)$

$\textsc{Clone-Rec}(L)$

if $\textsc{IsEmpty}(L)$ **then**
  return nil
else
  return $\textsc{Cons}(\textsc{Head}(L), \textsc{Clone-Rec}(\textsc{Tail}(L)))$
end if

Inversione di una lista

L'inversa della lista:

[1, 2, 3, 4, 5]

è la lista:

[5, 4, 3, 2, 1]

Esiste una soluzione iterativa?

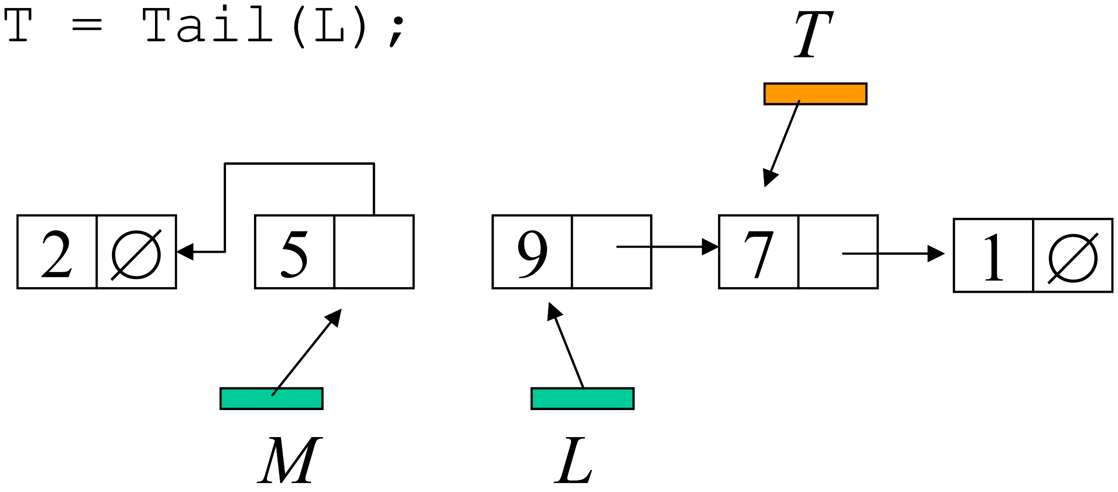
Inversione di una lista: iterazione

Problema: una lista semplice ha un verso solo (quindi l'algoritmo di inversione per un vettore non lo posso adattare)

Posso "girare" i puntatori mentre scandisco la lista con due puntatori

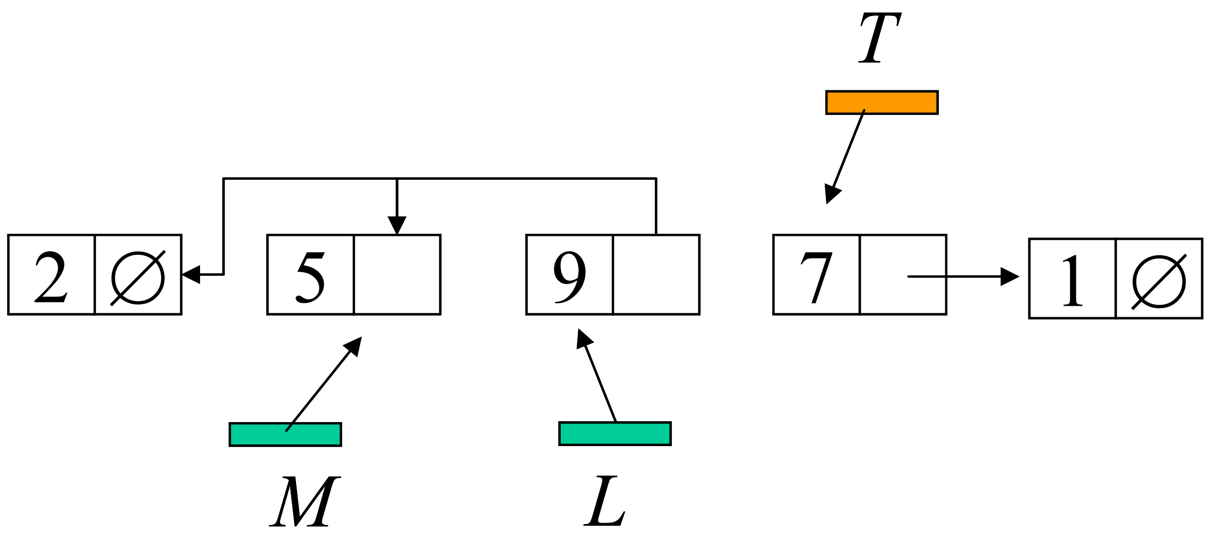
Inversione di una lista: iterazione

```
T = Tail(L);
```



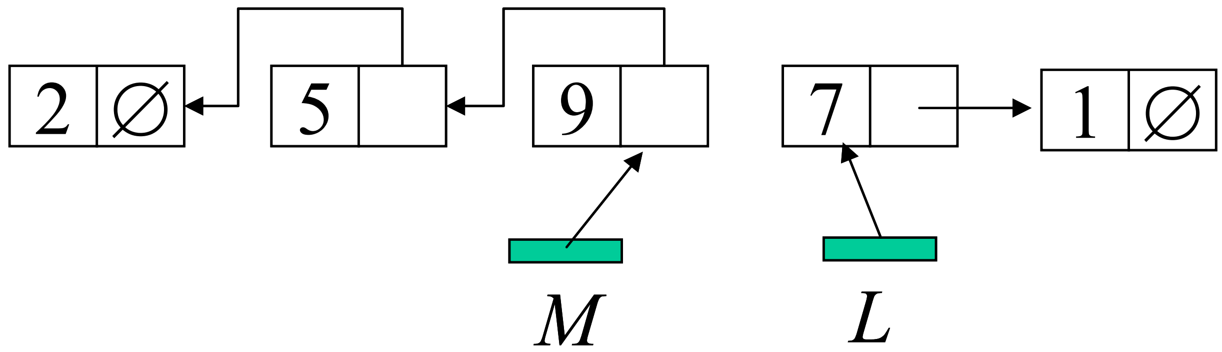
Inversione di una lista: iterazione

```
Tail(L) = M;
```



Inversione di una lista: iterazione

```
M = L; L = T;
```



Inversione di una lista: iterazione

$\textsc{Reverse-Iter}(L)$

$M \leftarrow nil$
while not $\textsc{IsEmpty}(L)$ **do**
  $T \leftarrow \textsc{Tail}(L)$
  $L.next \leftarrow M$
  $M \leftarrow L$
  $L \leftarrow T$
end while
return M

Inversione di una lista: ricorsione

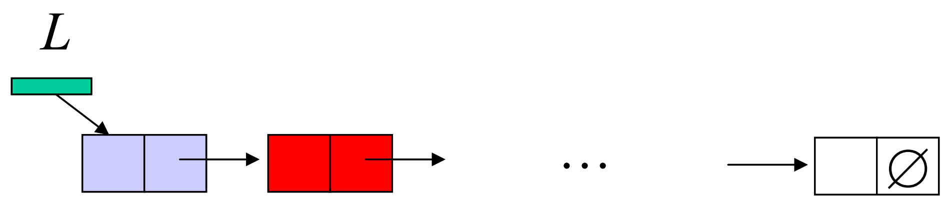
Come posso definire l'inversa induttivamente?

- L'inversa della lista vuota o con un solo elemento è la lista stessa
- L'inversa di una lista $[a|L]$ con almeno due elementi, è l'inversa di L con a aggiunto <u>in fondo</u>

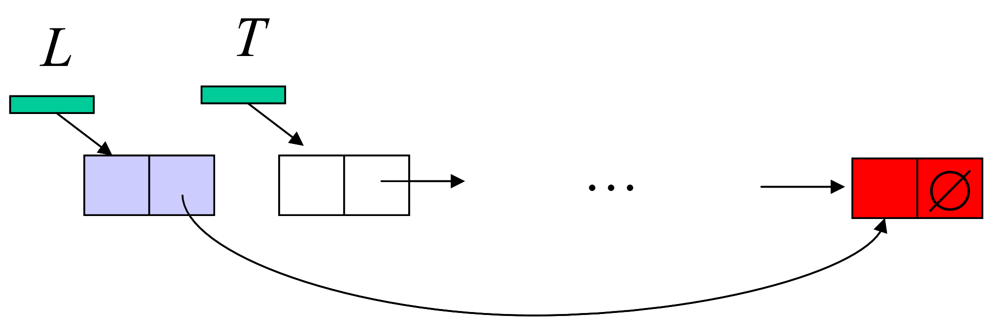
Per aggiungere in fondo devo scandire l'inversa di L?

Inversione di una lista: ricorsione

Se in una prima fase inverto ricorsivamente la coda di:

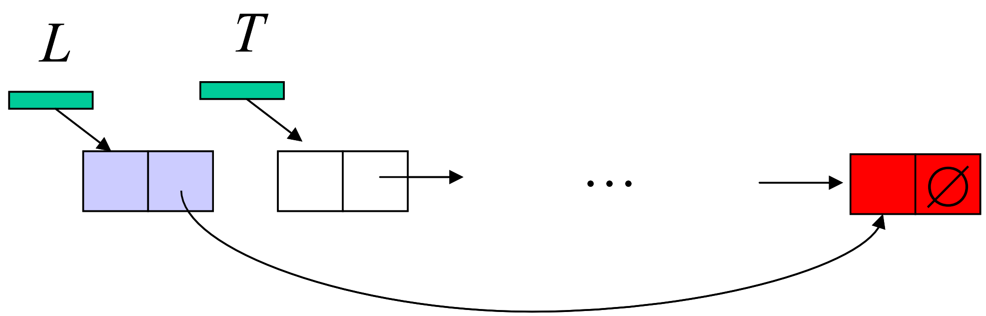


ottengo:

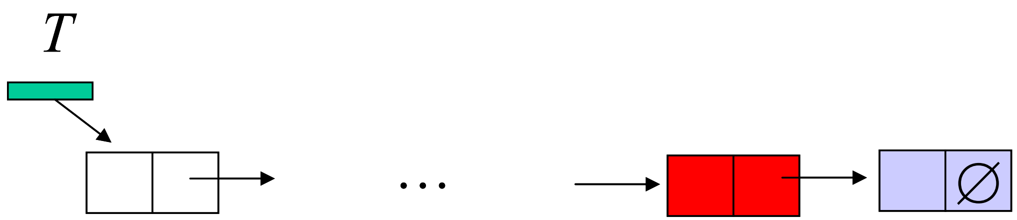


Inversione di una lista: ricorsione

Allora è facile passare da:

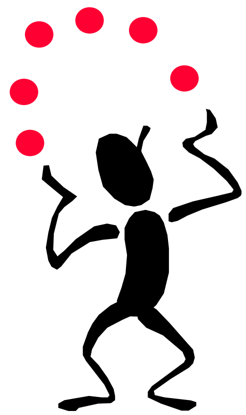


a:



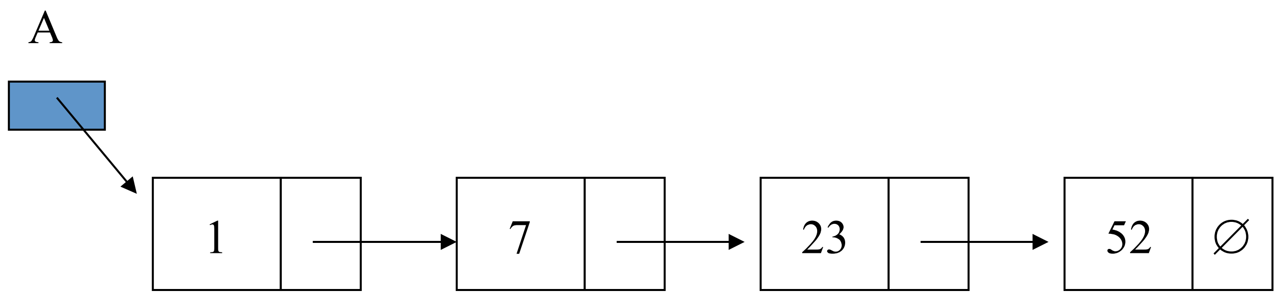
Inversione di una lista: ricorsione

Reverse-Rec(L)
if IsEmpty(L) **or** IsEmpty(Tail(L)) **then**
    return L
else
    $R \leftarrow$ Reverse-Rec(Tail(L))
    $L.next.next \leftarrow L$
    $L.next \leftarrow nil$
    return R
end if



Liste ordinate

$A = \{1, 7, 23, 52\}$



Possiamo rappresentare insiemi con liste ordinate

Ricerca ordinata

```
bool InOrd (int x, List L)
// pre:  L è ordinata
// post: true se x è in L
{   if (IsEmpty(L) || x < Head(L)) return false;
    else if (x == Head(L)) return true;
    else return InOrd (x, Tail(L));
}
```

Or valutato da sin. a des.

Se x < Head(L) allora $x \notin L$

… allora x > Head(L) quindi $x \in L$ sse $x \in Tail(L)$

Inserimento ordinato

```
List Insert (int x, List L)
// pre:  L è ordinata
// post: ritorna L ordinata e che contiene x
{   if (IsEmpty(L) || x < Head(L))
              return Cons(x, L)
    else if (x == Head(L)) return L;
    else
    {     Tail(L) = Insert(x, Tail(L));
          return L;
    }
}
```

Inserimento distruttivo nella coda

Cancellazione ordinata

```
List Delete (int x, List L)
// pre:  L è ordinata
// post: rimuove x da L (se presente)
{   if (IsEmpty(L) || x < Head(L)) return L;
    else if (x == Head(L))
    {     ListOfInt M = Tail(L); delete L;
          return M;
    }
    else
    {     Tail(L) = Delete(x, Tail(L));
          return L;
    }
}
```

Questo valore serve per "ricucire"la lista

Deallocazione necessaria del record puntato da L

In questo punto la lista viene ricucita

Unione

```
List Union (List L, List M)
// pre:  L ed M sono ordinate
// post: ritorna l'unione non distruttiva di L ed M
{  if (IsEmpty(L)) return Clone(M);
   else if (IsEmpty(M)) return Clone(L);
   else if (Head(L) == Head(M))
        return  Cons(Head(L), Union(Tail(L),Tail(M)));
   else if (Head(L) < Head(M))
        return Cons(Head(L), Union(Tail(L),M));
   else return Cons(Head(M), Union(L,Tail(M)));
}
```

Questo algoritmo è $O(n)$

Ordinamento per fusione

Come per i vettori; la fusione è più semplice, un po' più complicata la divisione:

Mergesort(L):
  dividi L in due parti uguali, L ed M
  L = Mergesort(L)
  M = Mergesort(M)
  ritorna la fusione ordinata di L ed M

Fondi (Merge)

```
List Merge (List L, List M)
// pre:  L ed M sono ordinate
// post: ritorna la fusione ordinata
//       distruttiva di L ed M
{  if (IsEmpty(L)) return M;
   else if (IsEmpty(M)) return L;
   else if (Head(L) <= Head(M))
        {  Tail(L) = Merge(Tail(L),M); return L; }
   else {  Tail(M) = Merge(L,Tail(M)); return M; }
}
```

Molto simile a Union: dove sono le differenze?

Dividi (Split): prima versione

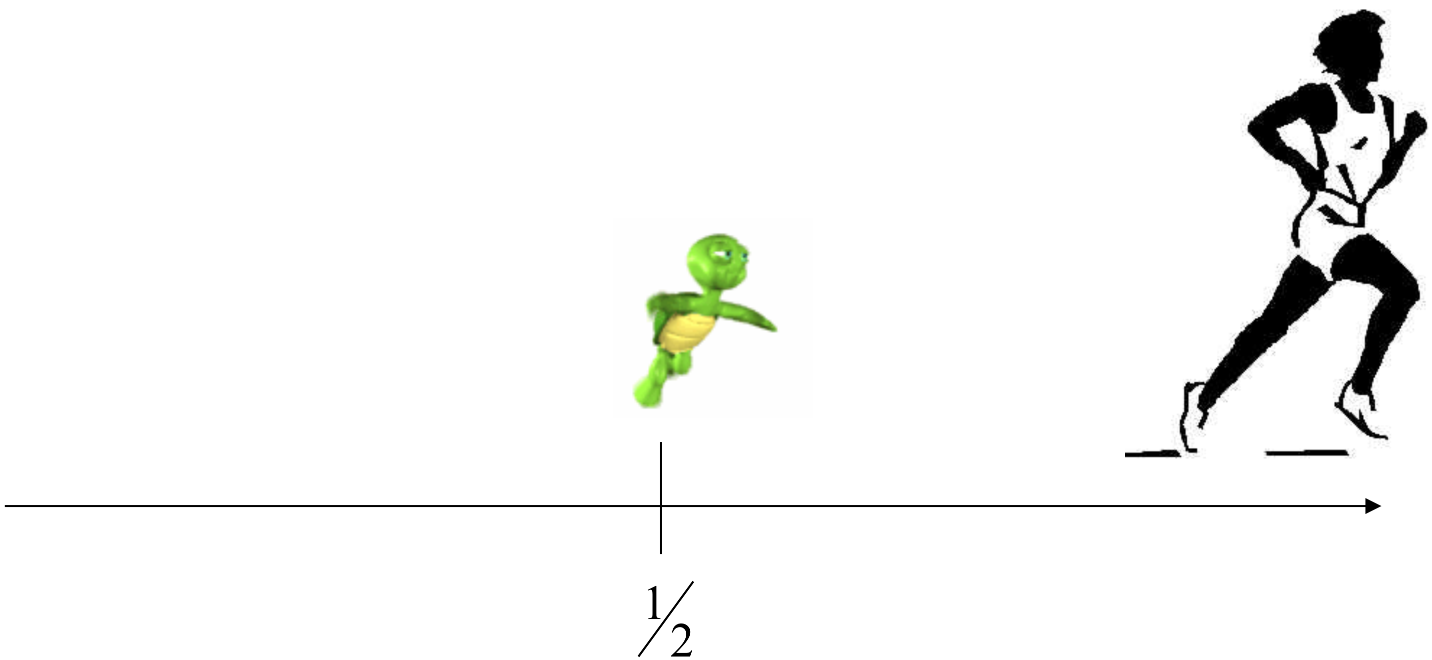
```
List SplitAfter (int n, List L)
// pre:  0 <= n <= lunghezza di L
// post: divide distr. L dopo n elementi ritornandone la
// seconda parte
{   if (n == 0) return L;
    else if (n == 1)
    {     List M = Tail(L); Tail(L) = TheEmptyList();
          return M;
    }
    else return SplitAfter(n-1, Tail(L));
}

List Split_1 (ListOfInt L)
{    int n = Length(L)/2;
     return SplitAfter(n, L);
}
```

Questo algoritmo scandisce una volta e mezzo la lista

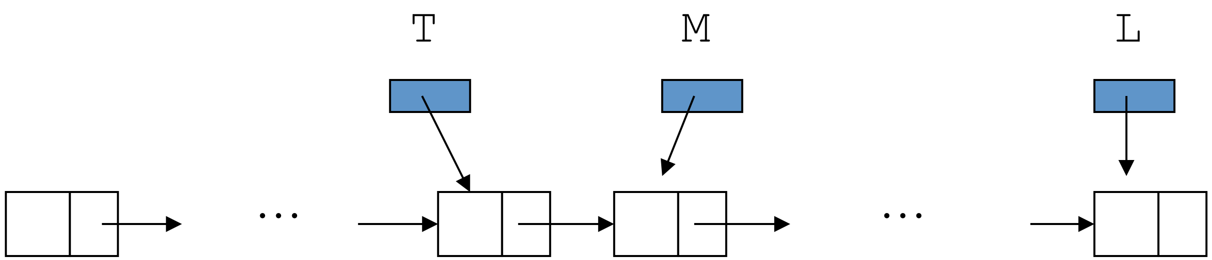
Dividi: seconda versione

Idea: percorriamo la lista con due puntatori, uno dei quali si muova a velocità doppia dell'altro.



Dividi: seconda versione

Idea: percorriamo la lista con due puntatori, uno dei quali si muova a velocità doppia dell'altro.



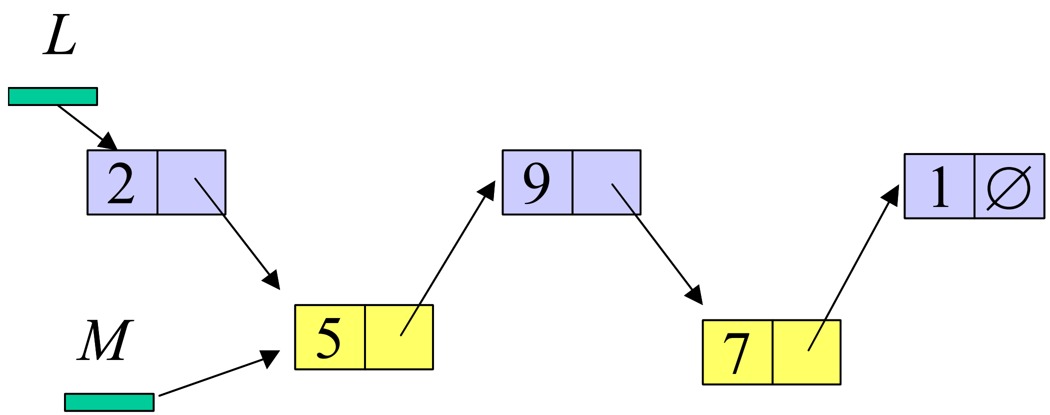
`M = Tail(T); T.next = TheEmptyList();`

Dividi: seconda versione

```
List Split_2 (List L)
{   if (IsEmpty(L) || IsEmpty(Tail(L)))
            return TheEmptyList();
    else
    {   List T, M = L;
        while (!IsEmpty(L) && !IsEmpty(Tail(L)))
        {     T = M; // T è il predecessore di M
              M = Tail(M);
              L = Tail(Tail(L));
        }
        M = Tail(T); T.next = TheEmptyList();
        return M;
    }
}
```

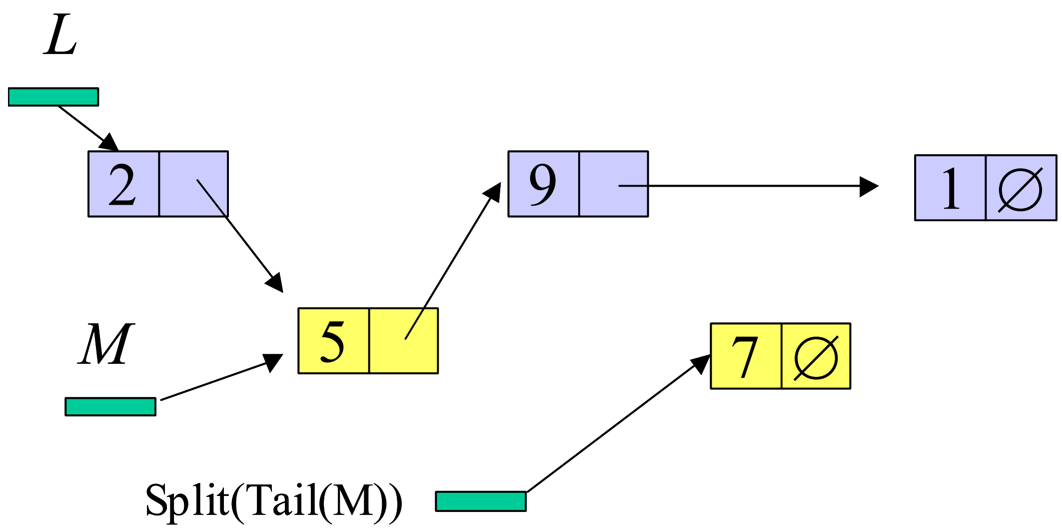
Dividi: terza versione

Idea: dividiamo la lista unendo gli elementi di posto pari in una nuova lista, e lasciando quelli di posto dispari.



Dividi: terza versione

Idea: dividiamo la lista unendo gli elementi di posto pari in una nuova lista, e lasciando quelli di posto dispari.



Dividi: terza versione

```
List Split_3 (List L)
// post: estrae da L gli el. di posto pari
//       e li ritorna in una lista senza allocazioni
{   if (IsEmpty(L) ||  IsEmpty(Tail(L)))
        return TheEmptyList();
    else  // L ha almeno due elementi
    {    List M = Tail(L);
         L.next = Tail(M);
         M.next = Split_3 (Tail(M));
         return M;
    }
}
```

Merge Sort

```
List MergeSort (List L)
// post: L è ordinata in senso crescente (distr.)
{     if (IsEmpty(L) || IsEmpty(Tail(L)))
          return L;
      else
      {   List M = Split(L);
          L = MergeSort(L);
          M = MergeSort(M);
          return Merge(L,M);
      }
}
```

Array, liste e tabelle hash

Corso di **Algoritmi e strutture dati**
Corso di Laurea in **Informatica**
Docenti: Ugo de'Liguoro, András Horváth

Indice

Sommario

- **obiettivo**: capire in che modo la scelta delle strutture dati per rappresentare **insiemi dinamici** influenzino il tempo di accesso ai dati

1. Insiemi dinamici

Studiamo strutture per rappresentare **insiemi dinamici**:

1. Insiemi dinamici

Studiamo strutture per rappresentare **insiemi dinamici**:

- numero finito di elementi

1. Insiemi dinamici

Studiamo strutture per rappresentare **insiemi dinamici**:

- numero finito di elementi
- gli elementi possono cambiare

1. Insiemi dinamici

Studiamo strutture per rappresentare **insiemi dinamici**:

- numero finito di elementi
- gli elementi possono cambiare
- il numero di elementi può cambiare

1. Insiemi dinamici

Studiamo strutture per rappresentare **insiemi dinamici**:

- numero finito di elementi
- gli elementi possono cambiare
- il numero di elementi può cambiare
- si assume che ogni elemento ha un attributo che serve da chiave

1. Insiemi dinamici

Studiamo strutture per rappresentare **insiemi dinamici**:

- numero finito di elementi
- gli elementi possono cambiare
- il numero di elementi può cambiare
- si assume che ogni elemento ha un attributo che serve da chiave
- le chiavi sono tutte diverse

1. Insiemi dinamici, operazioni

Esistono due **tipi di operazioni**:

- interrogazione (query)
- modifiche

1. Insiemi dinamici, operazioni

Esistono due **tipi di operazioni**:

- interrogazione (query)
- modifiche

Operazione tipiche:

- inserimento (insert)
- ricerca (search)
- cancellazione (delete)

1. Insiemi dinamici, operazioni

Operazione tipiche in caso di chiavi estratte da **insiemi totalmente ordinati:**

1. Insiemi dinamici, operazioni

Operazione tipiche in caso di chiavi estratte da **insiemi totalmente ordinati:**

- ricerca del minimo (minimum)

1. Insiemi dinamici, operazioni

Operazione tipiche in caso di chiavi estratte da **insiemi totalmente ordinati:**

- ricerca del minimo (minimum)
- ricerca del massimo (maximum)

1. Insiemi dinamici, operazioni

Operazione tipiche in caso di chiavi estratte da **insiemi totalmente ordinati:**

- ricerca del minimo (minimum)
- ricerca del massimo (maximum)
- ricerca del prossimo elemento più grande (successor)

1. Insiemi dinamici, operazioni

Operazione tipiche in caso di chiavi estratte da **insiemi totalmente ordinati:**

- ricerca del minimo (minimum)
- ricerca del massimo (maximum)
- ricerca del prossimo elemento più grande (successor)
- ricerca del prossimo elemento più piccolo (predecessor)

1. Complessità delle operazioni

- la **complessità**

1. Complessità delle operazioni

- la **complessità**
  - è misurata in funzione della dimensione dell’insieme,

1. Complessità delle operazioni

- la **complessità**
  - è misurata in funzione della dimensione dell'insieme,
  - **dipende da che tipo di struttura dati si utilizza per rappresentare l'insieme dinamico**

1. Complessità delle operazioni

- la **complessità**
  - è misurata in funzione della dimensione dell'insieme,
  - **dipende da che tipo di struttura dati si utilizza per rappresentare l'insieme dinamico**
- un operazione che è costosa con una certa struttura dati può costare poco con un'altra

1. Complessità delle operazioni

- la **complessità**
  - è misurata in funzione della dimensione dell'insieme,
  - **dipende da che tipo di struttura dati si utilizza per rappresentare l'insieme dinamico**
- un operazione che è costosa con una certa struttura dati può costare poco con un'altra
- quali operazioni sono necessarie dipende dall'applicazione

2. Tavole hash, introduzione

- con array e liste è facile implementare tanti tipi di operazioni
- ma con ognuna il costo di certi operazioni è $O(N)$
- le **tabelle hash** forniscono solo le operazioni di base (insert, search e delete) ma ognuna con tempo medio $O(1)$

2.1 Tavole a indirizzamento diretto

- un'idea preliminare a quella della tavole hash

2.1 Tavole a indirizzamento diretto

- un'idea preliminare a quella della tavole hash
- sia U l'universo delle chiavi: $U = \{0, 1, \ldots, m - 1\}$

2.1 Tavole a indirizzamento diretto

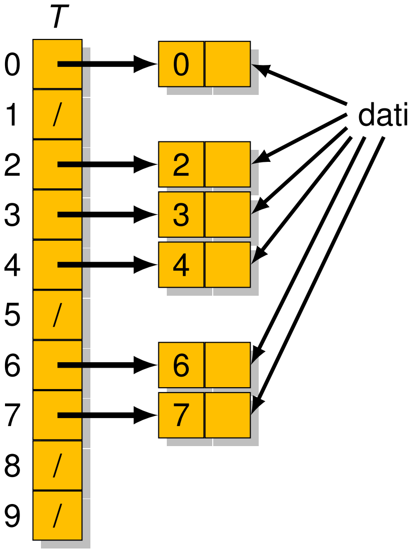
- un'idea preliminare a quella della tavole hash
- sia U l'universo delle chiavi: $U = \{0, 1, \ldots, m-1\}$
- l'insieme dinamico viene rappresentato con un array T di dimensione m in cui ogni posizione corrisponde ad una chiave

2.1 Tavole a indirizzamento diretto

- un'idea preliminare a quella della tavole hash
- sia U l'universo delle chiavi: $U = \{0, 1, \ldots, m-1\}$
- l'insieme dinamico viene rappresentato con un array T di dimensione m in cui ogni posizione corrisponde ad una chiave
- T è la **tavola a indirizzamento diretto** perché ogni sua cella corrisponde direttamente ad una chiave

2.1 Tavole a indirizzamento diretto

- universo delle chiavi: $U = \{0, 1, 2, ..., 9\}$
- insieme delle chiavi: $S = \{0, 2, 3, 4, 6, 7\}$



2.1 Tavole a indirizzamento diretto

- le operazioni sono semplicissime:

  TableInsert(T, x)
  $T[x.key] \leftarrow x$
  TableDelete(T, x)
  $T[x.key] \leftarrow nil$
  TableSearch(k)
  return $T[k]$

2.1 Tavole a indirizzamento diretto

- le operazioni sono semplicissime:

  TableInsert(T, x)
  $T[x.key] \leftarrow x$
  TableDelete(T, x)
  $T[x.key] \leftarrow nil$
  TableSearch(k)
  return $T[k]$
- operazioni in tempo $O(1)$

2.1 Tavole a indirizzamento diretto

- sembra una struttura molto efficiente

2.1 Tavole a indirizzamento diretto

- sembra una struttura molto efficiente
- da quale punto di viste non lo è?

2.1 Tavole a indirizzamento diretto

- sembra una struttura molto efficiente
- da quale punto di viste non lo è?
- quanto costa la struttura in termini di spazio?

2.1 Tavole a indirizzamento diretto

- sembra una struttura molto efficiente
- da quale punto di viste non lo è?
- quanto costa la struttura in termini di spazio?
- dipende dal contesto in cui viene utilizzata

2.1 Tavole a indirizzamento diretto

- considieriamo il seguente scenario:
  - studenti identificati con matricola composta da 6 cifre: abbiamo 10^6 possibili chiavi
  - T occupa $8 \cdot 10^6$ byte di memora (se un puntatore ne occupa 8)
  - di ogni studente memorizza 10^5 byte di dati (100kB)
  - ci sono 20000 studenti

2.1 Tavole a indirizzamento diretto

- consideriamo il seguente scenario:
  - studenti identificati con matricola composta da 6 cifre: abbiamo 10^6 possibili chiavi
  - T occupa $8 \cdot 10^6$ byte di memora (se un puntatore ne occupa 8)
  - di ogni studente memorizza 10^5 byte di dati (100kB)
  - ci sono 20000 studenti
- spazio occupato, ma non utilizzato:

$$8(10^6 - 20000) = 7840000\text{B} = 7.84\text{MB}$$

2.1 Tavole a indirizzamento diretto

- considieriamo il seguente scenario:
    - studenti identificati con matricola composta da 6 cifre: abbiamo 10^6 possibili chiavi
    - T occupa $8 \cdot 10^6$ byte di memora (se un puntatore ne occupa 8)
    - di ogni studente memorizza 10^5 byte di dati (100kB)
    - ci sono 20000 studenti
- spazio occupato, ma non utilizzato:

$$8(10^6 - 20000) = 7840000\text{B} = 7.84\text{MB}$$

- frazione di spazio occupato ma non utilizzato rispetto al totale:

$$\frac{7.84 \cdot 10^6}{8 \cdot 10^6 + 20000 \cdot 10^5} = 0.0039$$

  cioè circa 0.4%

2.1 Tavole a indirizzamento diretto

- consideriamo il seguente scenario:
  - studenti identificati con matricola composta da 6 cifre: abbiamo 10^6 possibili chiavi
  - T occupa $8 \cdot 10^6$ byte di memora (se un puntatore ne occupa 8)
  - di ogni studente memorizza 10^5 byte di dati (100kB)
  - ci sono 20000 studenti
- spazio occupato, ma non utilizzato:

$$8(10^6 - 20000) = 7840000\text{B} = 7.84\text{MB}$$

- frazione di spazio occupato ma non utilizzato rispetto al totale:
$$\frac{7.84 \cdot 10^6}{8 \cdot 10^6 + 20000 \cdot 10^5} = 0.0039$$
  cioè circa 0.4%
- quindi in questo contesto è ragionevole

2.1 Tavole a indirizzamento diretto

- se si memorizza solo 1kB di dati per studente:

$$\frac{7.84 \cdot 10^6}{8 \cdot 10^6 + 20000 \cdot 10^3} = 0.28$$

cioè circa 28% della memoria è occupata "inutilmente"

2.1 Tavole a indirizzamento diretto

- se si memorizza solo 1kB di dati per studente:

$$\frac{7.84 \cdot 10^6}{8 \cdot 10^6 + 20000 \cdot 10^3} = 0.28$$

  cioè circa 28% della memoria è occupata “inutilmente”

- se si memorizza solo 1kB di dati per studente e ci sono solo 200 studenti (quelli di un corso):

$$\frac{7.84 \cdot 10^6}{8 \cdot 10^6 + 200 \cdot 10^3} = 0.956$$

  cioè circa 95.6% della memoria è occupata “inutilmente”

2.2 Tavole hash

- l'indirizzamento diretto non è praticabile se l'universo delle chiavi è grande

2.2 Tavole hash

- l'indirizzamento diretto non è praticabile se l'universo delle chiavi è grande
- in ogni caso non è efficiente dal punto di vista della memoria utilizzata

2.2 Tavole hash

- l'indirizzamento diretto non è praticabile se l'universo delle chiavi è grande
- in ogni caso non è efficiente dal punto di vista della memoria utilizzata
- idea: utilizziamo una tabella T di dimensione m con m molto più piccolo di $|U|$

2.2 Tavole hash

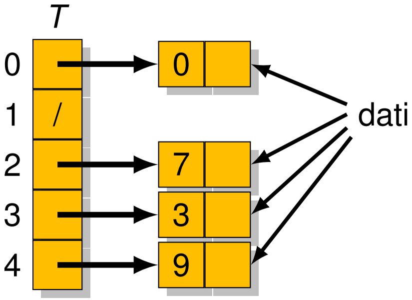
- l'indirizzamento diretto non è praticabile se l'universo delle chiavi è grande
- in ogni caso non è efficiente dal punto di vista della memoria utilizzata
- idea: utilizziamo una tabella T di dimensione m con m molto più piccolo di $|U|$
- la posizione della chiave k è determinata utilizzando una funzione

  $$h : U \to \{0, 1, \ldots, m - 1\}$$

  chiamata la funzione hash

2.2 Tavole hash

- universo delle chiavi: $U = \{0, 1, 2, ..., 9\}$
- insieme delle chiavi: $S = \{0, 3, 7, 9\}$
- funzione hash: $h(k) = k \bmod 5$
- $h(k)$ è il valore hash della chiave k



2.2 Tavole hash

- l'indirizzamento non è più diretto

2.2 Tavole hash

- l'indirizzamento non è più diretto
- l'elemento con chiave k si trova nella posizione $h(k)$

2.2 Tavole hash

- l'indirizzamento non è più diretto
- l'elemento con chiave k si trova nella posizione $h(k)$
- conseguenze:

2.2 Tavole hash

- l'indirizzamento non è più diretto
- l'elemento con chiave k si trova nella posizione $h(k)$
- conseguenze:
  - riduciamo lo spazio utilizzato

2.2 Tavole hash

- l'indirizzamento non è più diretto
- l'elemento con chiave k si trova nella posizione $h(k)$
- conseguenze:
  - riduciamo lo spazio utilizzato
  - perdiamo la diretta corrispondenza fra chiavi e posizioni

2.2 Tavole hash

- l'indirizzamento non è più diretto
- l'elemento con chiave k si trova nella posizione $h(k)$
- conseguenze:
  - riduciamo lo spazio utilizzato
  - perdiamo la diretta corrispondenza fra chiavi e posizioni
  - $m < |U|$ e quindi inevitabilmente **possono esserci delle collisioni**

2.2 Tavole hash

- nel caso dell'esempio precedente le coppie (0,5), (1,6), (2,7), (3,8) e (4,9) sono in collisione

2.2 Tavole hash

- nel caso dell'esempio precedente le coppie (0,5), (1,6), (2,7), (3,8) e (4,9) sono in collisione
- una buona funzione hash
  - posiziona le chiavi nelle posizioni $0, \ldots, m-1$ in modo apparentemente casuale e uniforme
  - quindi riduce al minimo il numero di collisioni

2.2 Tavole hash

- nel caso dell'esempio precedente le coppie (0,5), (1,6), (2,7), (3,8) e (4,9) sono in collisione
- una buona funzione hash
  - posiziona le chiavi nelle posizioni $0, \ldots, m-1$ in modo apparentemente casuale e uniforme
  - quindi riduce al minimo il numero di collisioni
- hash perfetto: una funzione che non crea mai collisione, cioè una funzione iniettiva:

$$k_1 \neq k_2 \implies h(k_1) \neq h(k_2)$$

- se $|U| > m$ allora, lo **hash perfetto è realizzabile solo se l'insieme rappresentato non è dinamico**

2.3 Tavole hash con concatenamento

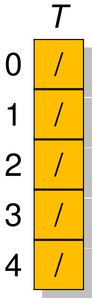
- come si fa a risolvere le collisioni che comunque possono verificarsi?

2.3 Tavole hash con concatenamento

- come si fa a risolvere le collisioni che comunque possono verificarsi?
- una possibile soluzione: **concatenando gli elementi in collisione in una lista**

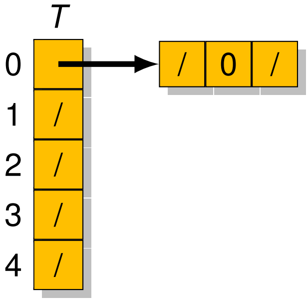
2.3 Tavole hash con concatenamento

- universo delle chiavi: $U = \{0, 1, 2, ..., 9\}$
- insieme delle chiavi: $S = \{0, 2, 3, 7, 9\}$
- funzione hash: $h(k) = k \bmod 5$



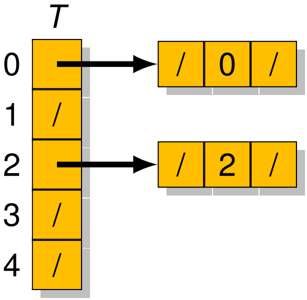
2.3 Tavole hash con concatenamento

- universo delle chiavi: $U = \{0, 1, 2, ..., 9\}$
- insieme delle chiavi: $S = \{0, 2, 3, 7, 9\}$
- funzione hash: $h(k) = k \bmod 5$



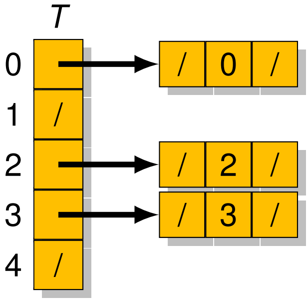
2.3 Tavole hash con concatenamento

- universo delle chiavi: $U = \{0, 1, 2, ..., 9\}$
- insieme delle chiavi: $S = \{0, 2, 3, 7, 9\}$
- funzione hash: $h(k) = k \bmod 5$



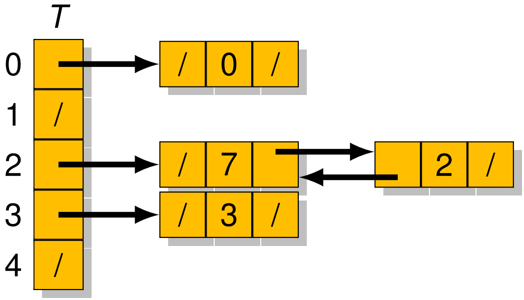
2.3 Tavole hash con concatenamento

- universo delle chiavi: $U = \{0, 1, 2, ..., 9\}$
- insieme delle chiavi: $S = \{0, 2, 3, 7, 9\}$
- funzione hash: $h(k) = k \bmod 5$



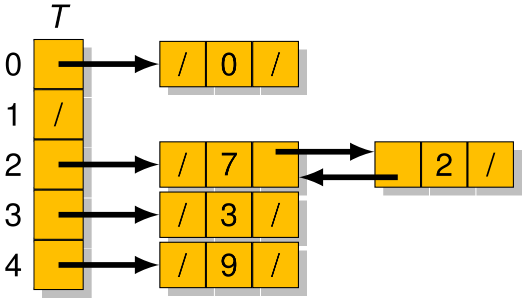
2.3 Tavole hash con concatenamento

- universo delle chiavi: $U = \{0, 1, 2, ..., 9\}$
- insieme delle chiavi: $S = \{0, 2, 3, 7, 9\}$
- funzione hash: $h(k) = k \bmod 5$



2.3 Tavole hash con concatenamento

- universo delle chiavi: $U = \{0, 1, 2, ..., 9\}$
- insieme delle chiavi: $S = \{0, 2, 3, 7, 9\}$
- funzione hash: $h(k) = k \bmod 5$



2.3 Tavole hash con concatenamento

- operazioni in caso di concatenamento:

$\textsc{HashInsert}(T, x)$
  $L \leftarrow T[h(x.key)]$
  $\textsc{ListInsert}(L, x)$

$\textsc{HashSearch}(T, k)$
  $L \leftarrow T[h(k)]$
  return $\textsc{ListSearch}(L, k)$

$\textsc{HashDelete}(T, x)$
  $L \leftarrow T[h(x.key)]$
  $\textsc{ListDelete}(L, x)$

2.3 Tavole hash con concatenamento

- operazioni in caso di concatenamento:

$\textsc{HashInsert}(T, x)$
  $L \leftarrow T[h(x.key)]$
  $\textsc{ListInsert}(L, x)$

$\textsc{HashSearch}(T, k)$
  $L \leftarrow T[h(k)]$
  return $\textsc{ListSearch}(L, k)$

$\textsc{HashDelete}(T, x)$
  $L \leftarrow T[h(x.key)]$
  $\textsc{ListDelete}(L, x)$

- come sono i tempi di esecuzione delle operazioni?

2.3 Tavole hash con concatenamento

- il valore hash di una chiave si calcola in tempo costante quindi l'inserimento si fa in tempo $O(1)$

2.3 Tavole hash con concatenamento

- il valore hash di una chiave si calcola in tempo costante quindi l'inserimento si fa in tempo $O(1)$
- la ricerca di un elemento con la chiave k richiede un tempo proporzionale alla lunghezza della lista $T[h(k)]$

2.3 Tavole hash con concatenamento

- il valore hash di una chiave si calcola in tempo costante quindi l'inserimento si fa in tempo $O(1)$
- la ricerca di un elemento con la chiave k richiede un tempo proporzionale alla lunghezza della lista $T[h(k)]$
- costo della ricerca dipende quindi dal numero di elementi e dalle caratteristiche della funzione hash

2.3 Tavole hash con concatenamento

- il valore hash di una chiave si calcola in tempo costante quindi l'inserimento si fa in tempo $O(1)$
- la ricerca di un elemento con la chiave k richiede un tempo proporzionale alla lunghezza della lista $T[h(k)]$
- costo della ricerca dipende quindi dal numero di elementi e dalle caratteristiche della funzione hash
- la cancellazione (di un elemento già individuato) richiede $O(1)$ perché la lista è doppiamente concatenata

2.3 Tavole hash con concatenamento

- analizziamo in dettaglio quanto costa una ricerca
- notazione:
  - m: numero di celle in T
  - n: numero di elementi memorizzati
  - $\alpha = n/m$: fattore di carico

2.3 Tavole hash con concatenamento

- qual è il **caso peggiore**?
- scenario:
  - l'universo delle chiavi: matricole con 6 cifre
  - $m = 200$
  - funzione hash: $h(k) = k \bmod 200$
- elenco di inserimento che rende pesante la ricerca:

2.3 Tavole hash con concatenamento

- qual è il **caso peggiore**?
- scenario:
  - l'universo delle chiavi: matricole con 6 cifre
  - $m = 200$
  - funzione hash: $h(k) = k \bmod 200$
- elenco di inserimento che rende pesante la ricerca:

$$000123, 100323, 123723, 343123, 333123, \ldots$$

2.3 Tavole hash con concatenamento

- qual è il **caso peggiore**?
- scenario:
  - l'universo delle chiavi: matricole con 6 cifre
  - $m = 200$
  - funzione hash: $h(k) = k \bmod 200$
- elenco di inserimento che rende pesante la ricerca:

$$000123, 100323, 123723, 343123, 333123, \ldots$$

- tutte le chiavi sono associate con la stessa cella di T!

2.3 Tavole hash con concatenamento

- qual è il **caso peggiore**?
- scenario:
  - l'universo delle chiavi: matricole con 6 cifre
  - $m = 200$
  - funzione hash: $h(k) = k \bmod 200$
- elenco di inserimento che rende pesante la ricerca:

$$000123, 100323, 123723, 343123, 333123, \ldots$$

- tutte le chiavi sono associate con la stessa cella di T!
- ricerca costa nel caso peggiore $\Theta(n)$

2.3 Tavole hash con concatenamento

- qual è il **caso peggiore**?
- scenario:
  - l'universo delle chiavi: matricole con 6 cifre
  - $m = 200$
  - funzione hash: $h(k) = k \mod 200$
- elenco di inserimento che rende pesante la ricerca:

$$000123, 100323, 123723, 343123, 333123, \ldots$$

- tutte le chiavi sono associate con la stessa cella di T!
- ricerca costa nel caso peggiore $\Theta(n)$
- qual è il **caso migliore**?

2.3 Tavole hash con concatenamento

- qual è il **caso peggiore**?
- scenario:
  - l'universo delle chiavi: matricole con 6 cifre
  - $m = 200$
  - funzione hash: $h(k) = k \bmod 200$
- elenco di inserimento che rende pesante la ricerca:

$$000123, 100323, 123723, 343123, 333123, \ldots$$

- tutte le chiavi sono associate con la stessa cella di T!
- ricerca costa nel caso peggiore $\Theta(n)$
- qual è il **caso migliore**?
- quando la lista $T[h(k)]$ è vuota oppure contiene solo un elemento

2.3 Tavole hash con concatenamento

- qual è il **caso peggiore**?
- scenario:
  - l'universo delle chiavi: matricole con 6 cifre
  - $m = 200$
  - funzione hash: $h(k) = k \bmod 200$
- elenco di inserimento che rende pesante la ricerca:

$$000123, 100323, 123723, 343123, 333123, \ldots$$

- tutte le chiavi sono associate con la stessa cella di T!
- ricerca costa nel caso peggiore $\Theta(n)$
- qual è il **caso migliore**?
- quando la lista $T[h(k)]$ è vuota oppure contiene solo un elemento
- ricerca costa nel caso migliore $O(1)$

2.3 Tavole hash, uniformità semplice

- qual è il costo nel **caso medio**?
- dipende dalla funzione hash
- assumiamo di avere una funzione che
  - sia facile da calcolare ($O(1)$)
  - goda della proprietà di uniformità semplice
- **uniformità semplice**: la funzione hash **distribuisce in modo uniforme le chiavi fra le celle** (ogni cella è destinazione dello stesso numero di chiavi)

2.3 Tavole hash, uniformità semplice

- la seguente funzione hash è uniforme semplice?

$$U = \{0, 1, 2, \ldots, 99\}, m = 10, h(k) = k \mod 10$$

2.3 Tavole hash, uniformità semplice

- la seguente funzione hash è uniforme semplice?

$$U = \{0, 1, 2, \ldots, 99\}, m = 10, h(k) = k \mod 10$$

- h restituisce l'ultima cifra della chiave
- l'ultima cifra è $c \in \{0, 1, 2, \ldots, 9\}$
- ognuno di questi numeri appare 10 volte come ultima cifra
- ogni cella è destinazione di 10 chiavi
- è uniforme semplice

2.3 Tavole hash, uniformità semplice

- la seguente funzione hash è uniforme semplice?

$$U = \{0, 1, 2, \ldots, 99\}, m = 19,$$

$$h(k) = \lfloor k/10 \rfloor + k \mod 10$$

2.3 Tavole hash, uniformità semplice

- la seguente funzione hash è uniforme semplice?

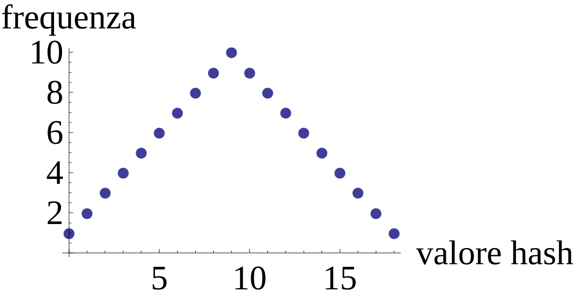
$$U = \{0, 1, 2, \ldots, 99\}, m = 19,$$

$$h(k) = \lfloor k/10 \rfloor + k \mod 10$$

- cioè h restituisce la somma delle cifre della chiave
- $h(k) = 0$ per $k = 0$
- $h(k) = 1$ per $k = 1$ e $k = 10$
- $h(k) = 2$ per $k = 2$ e $k = 11$ e $k = 20$
- ...

2.3 Tavole hash, uniformità semplice

- frequenza dei vari valori hash:



- non è uniforme semplice

2.3 Tavole hash con concatenamento

- **caso medio** con hashing uniforme semplice
- quanti elementi ci sono in una lista in media?
- sia n_i il numero di elementi nella lista $T[i]$ con $i = 0, 1, \ldots, m - 1$
- numero medio di elementi in una lista:

$$\bar{n} = \frac{n_0 + n_1 + ... + n_{m-1}}{m} = \frac{n}{m} = \alpha$$

2.3 Tavole hash con concatenamento

- tempo medio per **cercare un elemento che non c'è**:
  - tempo per individuare la lista è $\Theta(1)$
  - ogni lista ha la stessa probabilità di essere associata con la chiave (per l'uniformità semplice)
  - la lista ha in media α elementi e scandire la lista costa $\Theta(\alpha)$
- il tempo richiesto è $\Theta(1) + \Theta(\alpha) = \Theta(1 + \alpha)$
- attenzione: α non è costante!

2.3 Tavole hash con concatenamento

- tempo medio per **cercare un elemento che c'è**
- cerchiamo di capire quanto costa la ricerca di un elemento scelto a caso fra quelli presenti

2.3 Tavole hash con concatenamento

- tempo medio per **cercare un elemento che c'è**
- cerchiamo di capire quanto costa la ricerca di un elemento scelto a caso fra quelli presenti
- tempo per individuare la lista è sempre $\Theta(1)$

2.3 Tavole hash con concatenamento

- tempo medio per **cercare un elemento che c'è**
- cerchiamo di capire quanto costa la ricerca di un elemento scelto a caso fra quelli presenti
- tempo per individuare la lista è sempre $\Theta(1)$
- assumiamo che la ricerca riguardi l'i-esimo elemento inserito, denotato con x_i

2.3 Tavole hash con concatenamento

- tempo medio per **cercare un elemento che c'è**
- cerchiamo di capire quanto costa la ricerca di un elemento scelto a caso fra quelli presenti
- tempo per individuare la lista è sempre $\Theta(1)$
- assumiamo che la ricerca riguardi l'i-esimo elemento inserito, denotato con x_i
- per trovare x_i dobbiamo esaminare x_i stesso e tutti gli elementi che

2.3 Tavole hash con concatenamento

- tempo medio per **cercare un elemento che c'è**
- cerchiamo di capire quanto costa la ricerca di un elemento scelto a caso fra quelli presenti
- tempo per individuare la lista è sempre $\Theta(1)$
- assumiamo che la ricerca riguardi l'i-esimo elemento inserito, denotato con x_i
- per trovare x_i dobbiamo esaminare x_i stesso e tutti gli elementi che
  - hanno una chiave con lo stesso valore hash $h(x_i.key)$
  - sono stati inseriti dopo x_i (inserimento in testa)

2.3 Tavole hash con concatenamento

- quanti sono tali elementi?

2.3 Tavole hash con concatenamento

- quanti sono tali elementi?
- dopo x_i vengono inseriti $n - i$ elementi

2.3 Tavole hash con concatenamento

- quanti sono tali elementi?
- dopo x_i vengono inseriti $n - i$ elementi
- quanti di questi finiscono nella lista di x_i?

2.3 Tavole hash con concatenamento

- quanti sono tali elementi?
- dopo x_i vengono inseriti $n - i$ elementi
- quanti di questi finiscono nella lista di x_i?
- ogni elemento viene inserito nella lista di x_i con probabilità $\frac{1}{m}$ (uniformità semplice)

2.3 Tavole hash con concatenamento

- quanti sono tali elementi?
- dopo x_i vengono inseriti $n - i$ elementi
- quanti di questi finiscono nella lista di x_i?
- ogni elemento viene inserito nella lista di x_i con probabilità $\frac{1}{m}$ (uniformità semplice)
- quindi **in media** $\frac{n-i}{m}$ elementi precedono x_i nella lista di x_i

2.3 Tavole hash con concatenamento

- tempo per cercare x_i, calcolo del valore hash a parte, è proporzionale a
$$1 + \frac{n - i}{m}$$

2.3 Tavole hash con concatenamento

- tempo per cercare x_i, calcolo del valore hash a parte, è proporzionale a
$$1 + \frac{n - i}{m}$$
- tempo per cercare un elemento scelto a caso, calcolo del valore hash a parte, è proporzionale a
$$\frac{1}{n} \sum_{i=1}^{n} \left(1 + \frac{n - i}{m} \right)$$

2.3 Tavole hash con concatenamento

- elaboriamo la quantità precedente:

$$\frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{n-i}{m}\right) = \frac{1}{n}\sum_{i=1}^{n}1+\frac{1}{n}\sum_{i=1}^{n}\frac{n}{m}-\frac{1}{n}\sum_{i=1}^{n}\frac{i}{m} =$$

$$1+\frac{n}{m}-\frac{n(n+1)}{2nm} = 1+\frac{n-1}{2m} =$$

$$1+\frac{\alpha}{2}-\frac{\alpha}{2n}$$

- tempo richiesto in totale è

$$\Theta(1)+\Theta\left(1+\frac{\alpha}{2}-\frac{\alpha}{2n}\right) = \Theta(1+\alpha)$$

2.3 Tavole hash con concatenamento

- conclusione: in una tabella hash in cui le collisioni sono risolte mediante liste, nell'ipotesi di uniformità semplice, una ricerca richiede in media un tempo $\Theta(1 + \alpha)$
- cosa vuole dire in pratica $\Theta(1 + \alpha)$?

2.3 Tavole hash con concatenamento

- conclusione: in una tabella hash in cui le collisioni sono risolte mediante liste, nell'ipotesi di uniformità semplice, una ricerca richiede in media un tempo $\Theta(1 + \alpha)$
- cosa vuole dire in pratica $\Theta(1 + \alpha)$?
- se il numero di celle in T è proporzionale a n allora $n = O(m)$ e quindi $\alpha = O(1)$ e la ricerca richiede tempo $O(1)$

2.3 Tavole hash con concatenamento

- conclusione: in una tabella hash in cui le collisioni sono risolte mediante liste, nell'ipotesi di uniformità semplice, una ricerca richiede in media un tempo $\Theta(1 + \alpha)$
- cosa vuole dire in pratica $\Theta(1 + \alpha)$?
- se il numero di celle in T è proporzionale a n allora $n = O(m)$ e quindi $\alpha = O(1)$ e la ricerca richiede tempo $O(1)$
- quindi tutte le tre operazioni richiedono tempo $O(1)$ (se le liste sono doppiamente concatenate)

2.4 Funzioni hash

Significato della parola hash:

1. rifrittura, carne rifritta con cipolla, patate o altri vegetali
2. fiasco, pasticcio, guazzabuglio
3. (fig) rifrittume
4. (spec radio) segnali parassiti
5. nella locale slang «to settle sbs hash» mettere in riga qn, zittire o sottomettere qn, sistemare o mettere a posto qn una volta per tutte
6. anche hash sign (tipog) il simbolo tipografico

2.4 Funzioni hash

- una buona funzione hash è **uniforme semplice**

2.4 Funzioni hash

- una buona funzione hash è **uniforme semplice**
- ma questa è difficile da verificare perché di solito la distribuzione secondo la quale si estraggono le chiavi non è nota

2.4 Funzioni hash

- una buona funzione hash è **uniforme semplice**
- ma questa è difficile da verificare perché di solito la distribuzione secondo la quale si estraggono le chiavi non è nota
- le chiavi vengono interpretate come numeri naturali: ogni chiave è una sequenza di bit

2.4 Funzioni hash

- una buona funzione hash è **uniforme semplice**
- ma questa è difficile da verificare perché di solito la distribuzione secondo la quale si estraggono le chiavi non è nota
- le chiavi vengono interpretate come numeri naturali: ogni chiave è una sequenza di bit
- si cerca di utilizzare ogni bit della chiave

2.4 Funzioni hash

- una buona funzione hash è **uniforme semplice**
- ma questa è difficile da verificare perché di solito la distribuzione secondo la quale si estraggono le chiavi non è nota
- le chiavi vengono interpretate come numeri naturali: ogni chiave è una sequenza di bit
- si cerca di utilizzare ogni bit della chiave
- una buona funzione hash sceglie posizioni in modo tale da **eliminare eventuale regolarità** nei dati

2.4 Metodo della divisione

- il **metodo della divisione** assegna alla chiave k la posizione
$$h(k) = k \mod m$$

2.4 Metodo della divisione

- il **metodo della divisione** assegna alla chiave k la posizione
$$h(k) = k \mod m$$
- molto veloce

2.4 Metodo della divisione

- il **metodo della divisione** assegna alla chiave k la posizione
$$h(k) = k \mod m$$
- molto veloce
- bisogna scegliere bene m

2.4 Metodo della divisione

- stringhe come numeri naturali secondo il codice ASCII

$$\text{oca} \rightarrow 111 \cdot 128^2 + 99 \cdot 128^1 + 97 \cdot 128^0$$

2.4 Metodo della divisione

- stringhe come numeri naturali secondo il codice ASCII

$$\text{oca} \rightarrow 111 \cdot 128^2 + 99 \cdot 128^1 + 97 \cdot 128^0$$

- posizioni con diverse scelte di m

parola	$m = 2048$	$m = 1583$
le	1637	695
variabile	1637	1261
molle	1637	217
bolle	1637	680

2.4 Metodo della divisione

- stringhe come numeri naturali secondo il codice ASCII

$$\text{oca} \rightarrow 111 \cdot 128^2 + 99 \cdot 128^1 + 97 \cdot 128^0$$

- posizioni con diverse scelte di m

parola	$m = 2048$	$m = 1583$
le	1637	695
variabile	1637	1261
molle	1637	217
bolle	1637	680

- $m = 2^p$ è una buona scelta solo se si ha certezza che gli ultimi bit hanno distribuzione uniforme

2.4 Metodo della divisione

- stringhe come numeri naturali secondo il codice ASCII

$$\text{oca} \rightarrow 111 \cdot 128^2 + 99 \cdot 128^1 + 97 \cdot 128^0$$

- posizioni con diverse scelte di m

parola	$m = 2048$	$m = 1583$
le	1637	695
variabile	1637	1261
molle	1637	217
bolle	1637	680

- $m = 2^p$ è una buona scelta solo se si ha certezza che gli ultimi bit hanno distribuzione uniforme
- un numero primo non vicino a una potenza di 2 è spesso una buona scelta

2.4 Metodo della moltiplicazione

- **metodo della moltiplicazione**: con $0 < A < 1$

$$h(k) = \lfloor m(k\,A \mod 1) \rfloor$$

dove $x \mod 1$ è la parte frazionaria di x

2.4 Metodo della moltiplicazione

- **metodo della moltiplicazione**: con $0 < A < 1$

$$h(k) = \lfloor m(k\,A \mod 1) \rfloor$$

dove $x \mod 1$ è la parte frazionaria di x

- il valore di m non è critico, di solito si sceglie una potenza di 2

2.4 Metodo della moltiplicazione

- **metodo della moltiplicazione**: con $0 < A < 1$

  $$h(k) = \lfloor m(k\,A \mod 1) \rfloor$$

  dove $x \mod 1$ è la parte frazionaria di x
- il valore di m non è critico, di solito si sceglie una potenza di 2
- la scelta ottimale di A dipende dai dati ma $A = (\sqrt{5} - 1)/2$ è un valore ragionevole:

2.4 Metodo della moltiplicazione

- **metodo della moltiplicazione**: con $0 < A < 1$

$$h(k) = \lfloor m(k\,A \mod 1) \rfloor$$

  dove $x \mod 1$ è la parte frazionaria di x
- il valore di m non è critico, di solito si sceglie una potenza di 2
- la scelta ottimale di A dipende dai dati ma $A = (\sqrt{5} - 1)/2$ è un valore ragionevole:

parola	$m = 2048$
mille	1691
polli	678
molle	242
bolle	1508

2.5 Indirizzamento aperto

- con l'**indirizzamento aperto** tutti gli elementi sono memorizzati nella tavola T

2.5 Indirizzamento aperto

- con l'**indirizzamento aperto** tutti gli elementi sono memorizzati nella tavola T
- l'elemento con chiave k viene inserito nella posizione $h(k)$ se essa è libera

2.5 Indirizzamento aperto

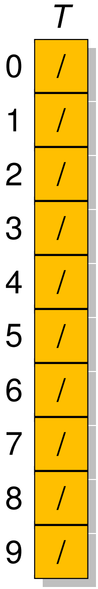
- con l'**indirizzamento aperto** tutti gli elementi sono memorizzati nella tavola T
- l'elemento con chiave k viene inserito nella posizione $h(k)$ se essa è libera
- se non è libera allora si cerca una posizione libera secondo un **schema di ispezione**

2.5 Indirizzamento aperto

- con l'**indirizzamento aperto** tutti gli elementi sono memorizzati nella tavola T
- l'elemento con chiave k viene inserito nella posizione $h(k)$ se essa è libera
- se non è libera allora si cerca una posizione libera secondo un **schema di ispezione**
- schema più semplice è l'**ispezione lineare**: a partire dalla posizione $h(k)$ l'elemento viene inserito nella prima cella libera

2.5 Indirizzamento aperto, ispezione lineare

- universo delle chiavi: $U = \{0, 1, 2, ..., 99\}$
- sequenza di inserimento: $88, 12, 2, 22, 33$
- funzione hash: $h(k) = k \bmod 10$



2.5 Indirizzamento aperto, ispezione lineare

- universo delle chiavi: $U = \{0, 1, 2, ..., 99\}$
- sequenza di inserimento: $88, 12, 2, 22, 33$
- funzione hash: $h(k) = k \bmod 10$

	T
0	/
1	/
2	/
3	/
4	/
5	/
6	/
7	/
8	88
9	/

2.5 Indirizzamento aperto, ispezione lineare

- universo delle chiavi: $U = \{0, 1, 2, ..., 99\}$
- sequenza di inserimento: $88, 12, 2, 22, 33$
- funzione hash: $h(k) = k \bmod 10$

	T
0	/
1	/
2	12
3	/
4	/
5	/
6	/
7	/
8	88
9	/

2.5 Indirizzamento aperto, ispezione lineare

- universo delle chiavi: $U = \{0, 1, 2, ..., 99\}$
- sequenza di inserimento: $88, 12, 2, 22, 33$
- funzione hash: $h(k) = k \bmod 10$

	T
0	/
1	/
2	12
3	2
4	/
5	/
6	/
7	/
8	88
9	/

2.5 Indirizzamento aperto, ispezione lineare

- universo delle chiavi: $U = \{0, 1, 2, ..., 99\}$
- sequenza di inserimento: $88, 12, 2, 22, 33$
- funzione hash: $h(k) = k \bmod 10$

	T
0	/
1	/
2	12
3	2
4	22
5	/
6	/
7	/
8	88
9	/

2.5 Indirizzamento aperto, ispezione lineare

- universo delle chiavi: $U = \{0, 1, 2, ..., 99\}$
- sequenza di inserimento: $88, 12, 2, 22, 33$
- funzione hash: $h(k) = k \bmod 10$

	T
0	/
1	/
2	12
3	2
4	22
5	33
6	/
7	/
8	88
9	/

2.5 Indirizzamento aperto

- in generale l'indirizzamento aperto può essere descritto con una funzione hash anche dell'ordine di ispezione:

$$h : U \times \{0, 1, 2, ..., m-1\} \rightarrow \{0, 1, 2, ..., m-1\}$$

2.5 Indirizzamento aperto

- in generale l'indirizzamento aperto può essere descritto con una funzione hash anche dell'ordine di ispezione:

$$h : U \times \{0, 1, 2, ..., m-1\} \to \{0, 1, 2, ..., m-1\}$$

- un elemento con la chiave k viene inserito
  - nella posizione $h(k, 0)$ se questa è libera

2.5 Indirizzamento aperto

- in generale l'indirizzamento aperto può essere descritto con una funzione hash anche dell'ordine di ispezione:

$$h : U \times \{0, 1, 2, ..., m-1\} \rightarrow \{0, 1, 2, ..., m-1\}$$

- un elemento con la chiave k viene inserito
  - nella posizione $h(k, 0)$ se questa è libera
  - altrimenti nella posizione $h(k, 1)$ se questa è libera

2.5 Indirizzamento aperto

- in generale l'indirizzamento aperto può essere descritto con una funzione hash anche dell'ordine di ispezione:

$$h : U \times \{0, 1, 2, ..., m-1\} \rightarrow \{0, 1, 2, ..., m-1\}$$

- un elemento con la chiave k viene inserito
  - nella posizione $h(k, 0)$ se questa è libera
  - altrimenti nella posizione $h(k, 1)$ se questa è libera
  - altrimenti nella posizione $h(k, 2)$ se questa è libera
  - ...

2.5 Indirizzamento aperto

- in generale l'indirizzamento aperto può essere descritto con una funzione hash anche dell'ordine di ispezione:

$$h : U \times \{0, 1, 2, ..., m-1\} \to \{0, 1, 2, ..., m-1\}$$

- un elemento con la chiave k viene inserito
  - nella posizione $h(k, 0)$ se questa è libera
  - altrimenti nella posizione $h(k, 1)$ se questa è libera
  - altrimenti nella posizione $h(k, 2)$ se questa è libera
  - ...
- l'ispezione è lineare se

$$h(k, i) = (h'(k) + i) \mod m$$

dove $h'(k)$ è la funzione hash “normale”

2.5 Indirizzamento aperto

- **inserimento in generale** con indirizzamento aperto

$\textsc{HashInsert}(T, x)$
  $i \leftarrow 0$
  while $i < m$ **do**
    $j \leftarrow h(x.key, i)$
    if $T[j] = nil$ **then**
      $T[j] \leftarrow x$
      return j
    $i \leftarrow i + 1$
  return nil

2.5 Indirizzamento aperto

- **ricerca in generale** con indirizzamento aperto

HashSearch(T, k)
  $i \leftarrow 0$
  while $i < m$ **do**
    $j \leftarrow h(k, i)$
    if $T[j] == nil$ **then**
      return nil
    if $T[j].key == k$ **then**
      return $T[j]$
    $i \leftarrow i + 1$
  return nil

2.5 Indirizzamento aperto

- quanto costa la **cancellazione in generale** con indirizzamento aperto?

2.5 Indirizzamento aperto

- quanto costa la **cancellazione in generale** con indirizzamento aperto?
- per cancellare un elemento, non possiamo semplicemente marcare la posizione in cui si trova con *nil*

2.5 Indirizzamento aperto

- quanto costa la **cancellazione in generale** con indirizzamento aperto?
- per cancellare un elemento, non possiamo semplicemente marcare la posizione in cui si trova con *nil*
- si può marcare gli elementi cancellati con *deleted*

2.5 Indirizzamento aperto

- quanto costa la **cancellazione in generale** con indirizzamento aperto?
- per cancellare un elemento, non possiamo semplicemente marcare la posizione in cui si trova con *nil*
- si può marcare gli elementi cancellati con *deleted*
- richiede modifiche alla procedura inserimento

2.5 Indirizzamento aperto

- quanto costa la **cancellazione in generale** con indirizzamento aperto?
- per cancellare un elemento, non possiamo semplicemente marcare la posizione in cui si trova con *nil*
- si può marcare gli elementi cancellati con *deleted*
- richiede modifiche alla procedura inserimento
- di solito l'indirizzamento aperto si usa quando non c'è necessità di cancellare

2.5 Indirizzamento aperto, schemi di ispezione

- l'ispezione lineare crea file di celle occupate, fenomeno chiamato **addensamento primario**

2.5 Indirizzamento aperto, schemi di ispezione

- l'ispezione lineare crea file di celle occupate, fenomeno chiamato **addensamento primario**
- **ispezione quadratica**:

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \mod m$$

2.5 Indirizzamento aperto, schemi di ispezione

- l'ispezione lineare crea file di celle occupate, fenomeno chiamato **addensamento primario**
- **ispezione quadratica**:

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \mod m$$

- con l'ispezione lineare e l'ispezione quadratica la sequenza dipende solo dal valore di hash, questo crea **addensamento secondario**: $h(k_1,0) = h(k_2,0)$ implica $h(k_1,i) = h(k_2,i)$ per ogni i

2.5 Indirizzamento aperto, schemi di ispezione

- l'ispezione lineare crea file di celle occupate, fenomeno chiamato **addensamento primario**
- **ispezione quadratica**:

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \mod m$$

- con l'ispezione lineare e l'ispezione quadratica la sequenza dipende solo dal valore di hash, questo crea **addensamento secondario**:
  $h(k_1, 0) = h(k_2, 0)$ implica $h(k_1, i) = h(k_2, i)$ per ogni i
- **doppio hashing**:

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \mod m$$

2.5 Indirizzamento aperto, schemi di ispezione

- l'ispezione lineare crea file di celle occupate, fenomeno chiamato **addensamento primario**
- **ispezione quadratica**:

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \mod m$$

- con l'ispezione lineare e l'ispezione quadratica la sequenza dipende solo dal valore di hash, questo crea **addensamento secondario**:
  $h(k_1,0) = h(k_2,0)$ implica $h(k_1,i) = h(k_2,i)$ per ogni i
- **doppio hashing**:

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \mod m$$

- con doppio hashing la sequenza dipende dalla chiave e non soltanto dal valore hash della chiave

2.5 Indirizzamento aperto, costo della ricerca

- consideriamo il caso ottimale dal punto di vista della funzione hash e lo schema di ispezione:

2.5 Indirizzamento aperto, costo della ricerca

- consideriamo il caso ottimale dal punto di vista della funzione hash e lo schema di ispezione:
  - la posizione di una chiave scelta a caso ha distribuzione uniforme
  - qualunque sequenza di ispezione ha la stessa probabilità

2.5 Indirizzamento aperto, costo della ricerca

- consideriamo il caso ottimale dal punto di vista della funzione hash e lo schema di ispezione:
  - la posizione di una chiave scelta a caso ha distribuzione uniforme
  - qualunque sequenza di ispezione ha la stessa probabilità
- consideriamo la ricerca di un elemento assente

2.5 Indirizzamento aperto, costo della ricerca

- denotiamo con X il numero di celle esaminate durante una ricerca senza successo

2.5 Indirizzamento aperto, costo della ricerca

- denotiamo con X il numero di celle esaminate durante una ricerca senza successo
- X è almeno 1: $P(X \geq 1) = 1$

2.5 Indirizzamento aperto, costo della ricerca

- denotiamo con X il numero di celle esaminate durante una ricerca senza successo
- X è almeno 1: $P(X \geq 1) = 1$
- bisogna esaminare almeno due celle se la prima è occupata:
$$P(X \geq 2) = \frac{n}{m}$$

2.5 Indirizzamento aperto, costo della ricerca

- denotiamo con X il numero di celle esaminate durante una ricerca senza successo
- X è almeno 1: $P(X \geq 1) = 1$
- bisogna esaminare almeno due celle se la prima è occupata:
$$P(X \geq 2) = \frac{n}{m}$$
- bisogna esaminare almeno tre celle con probabilità:
$$P(X \geq 3) = \frac{n}{m}\frac{n-1}{m-1}$$

2.5 Indirizzamento aperto, costo della ricerca

- denotiamo con X il numero di celle esaminate durante una ricerca senza successo
- X è almeno 1: $P(X \geq 1) = 1$
- bisogna esaminare almeno due celle se la prima è occupata:
$$P(X \geq 2) = \frac{n}{m}$$
- bisogna esaminare almeno tre celle con probabilità:
$$P(X \geq 3) = \frac{n}{m}\frac{n-1}{m-1}$$
- bisogna esaminare almeno i celle con probabilità:
$$P(X \geq i) = \frac{n}{m}\frac{n-1}{m-1}\cdots\frac{n-i+2}{m-i+2} \leq \alpha^{i-1}$$

2.5 Indirizzamento aperto, costo della ricerca

- numero medio di celle esaminate:

$$E[X] = \sum_{i=1}^{\infty} P(X \geq i) \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \frac{1}{1-\alpha}$$

2.5 Indirizzamento aperto, costo della ricerca

- numero medio di celle esaminate:

$$E[X] = \sum_{i=1}^{\infty} P(X \geq i) \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \frac{1}{1-\alpha}$$

- numero medio di ispezioni è minore di $1/(1-\alpha)$
- quanto vale $1/(1-\alpha)$ con certi valori di $\alpha < 1$?

2.5 Indirizzamento aperto, costo della ricerca

- numero medio di celle esaminate:

$$E[X] = \sum_{i=1}^{\infty} P(X \geq i) \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \frac{1}{1-\alpha}$$

- numero medio di ispezioni è minore di $1/(1-\alpha)$
- quanto vale $1/(1-\alpha)$ con certi valori di $\alpha < 1$?
- l'inserimento si analizza con lo stesso approccio

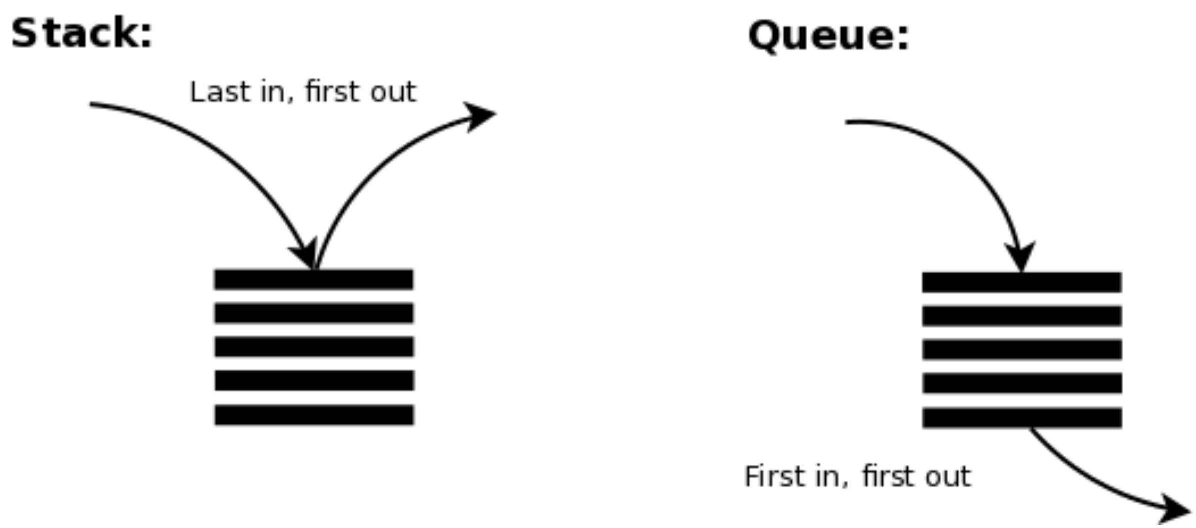
2.5 Indirizzamento aperto, costo della ricerca

- numero medio di celle esaminate:

$$E[X] = \sum_{i=1}^{\infty} P(X \geq i) \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \frac{1}{1-\alpha}$$

- numero medio di ispezioni è minore di $1/(1-\alpha)$
- quanto vale $1/(1-\alpha)$ con certi valori di $\alpha < 1$?
- l'inserimento si analizza con lo stesso approccio
- ricerca con successo richiede di esaminare meno celle

Pile, code

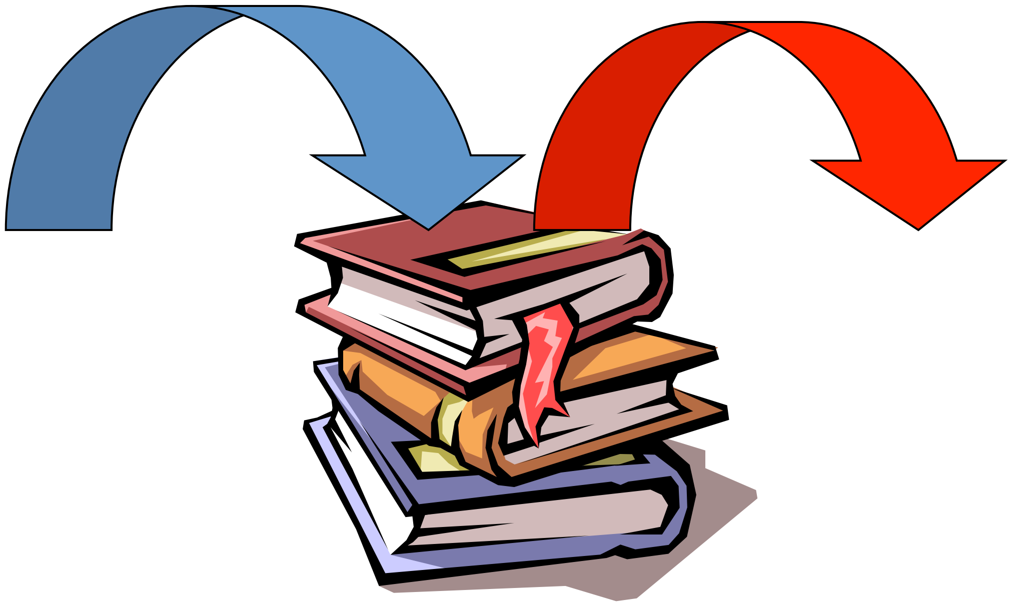


Algoritmi e strutture dati
Lezione 10, a.a. 2016-17

Ugo de'Liguoro, Andras Horvath

Pile: definizione informale

Una pila è una struttura dati lineare, i cui gli elementi possono essere aggiunti o sottratti da un solo estremo (Last In First Out - LIFO).



ADT delle pile

Una pila (*stack*) si definisce astrattamente come una struttura dati su cui siano definite alemeno quattro operazioni:

1. Push(e, S) : aggiunge e alla pila S
2. Pop(S) : elimina l'elemento emergente da S e lo ritorna
3. Empty(S): ritorna *true* se S non ha elementi.

Nota: se S è vuota, Pop(S) e Top(S) sono indefinite.

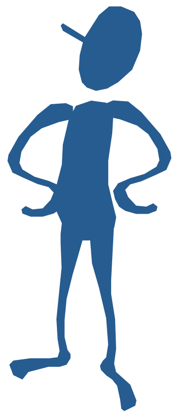
Pile realizzate con vettori

	0	1	2	3	4	5	6	7
$S =$	15	6	2	9				

$S.top = 3$

	0	1	2	3	4	5	6	7
Push(3, S) =	15	6	2	9	3			

$S.top = 4$



Pile realizzate con vettori

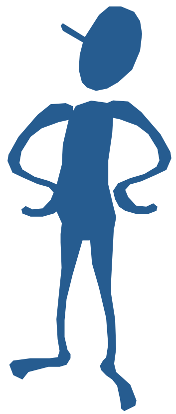
	0	1	2	3	4	5	6	7
$S =$	15	6	2	9	3			

$S.top = 4$

Pop(S) = 3

	0	1	2	3	4	5	6	7
$S' =$	15	6	2	9	3			

$S.top = 3$



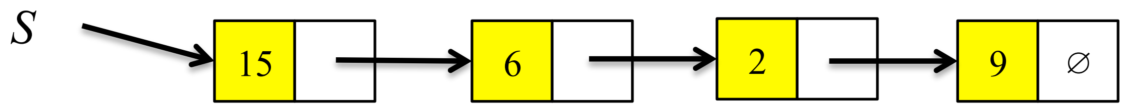
Pile realizzate con vettori

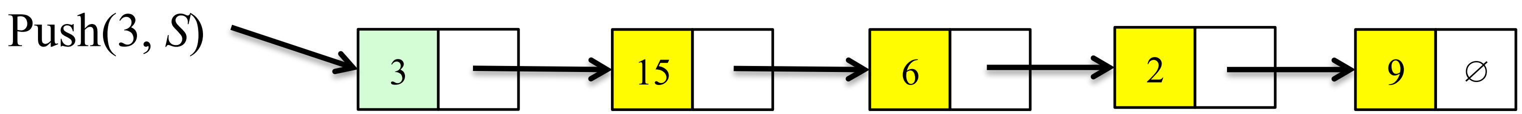
STACK-ARRAY-EMPTY(S)
return $S.top = -1$

STACK-ARRAY-PUSH(x, S)
if $S.top = dim(S)$ **then**
    error "overflow"
else
    $S.top \leftarrow S.top + 1$
    $S[S.top] \leftarrow x$
end if

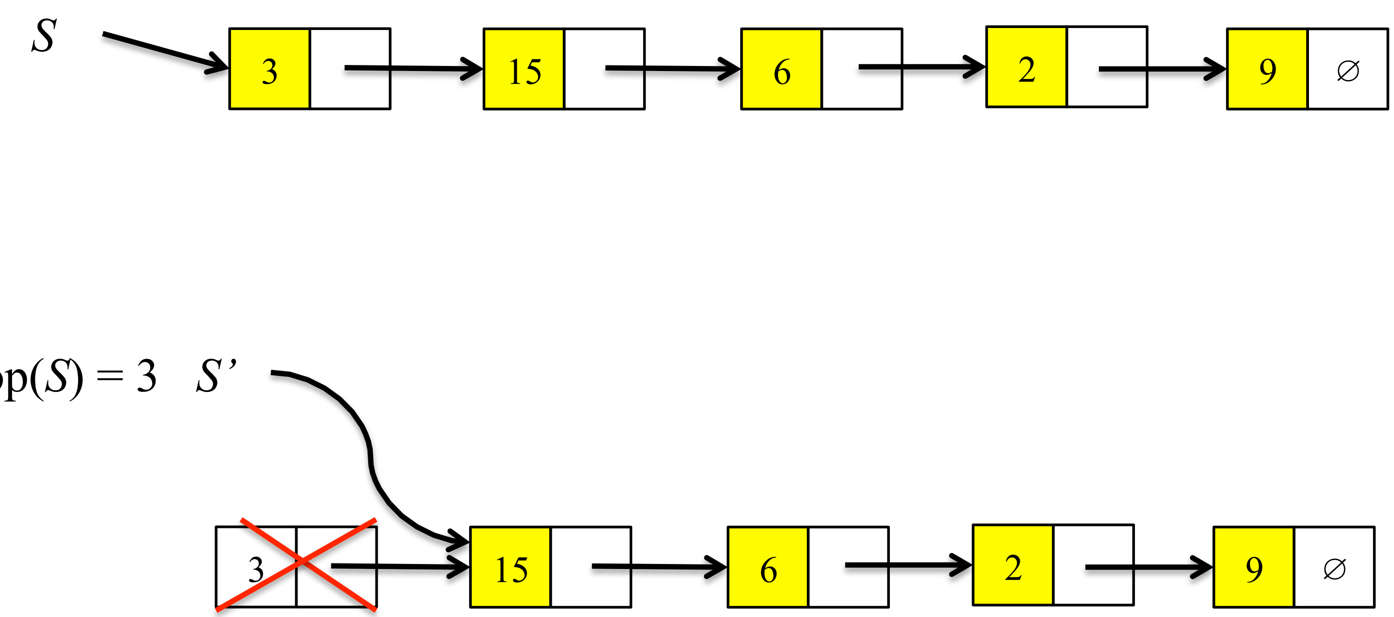
STACK-ARRAY-POP(S)
if STACK-ARRAY-EMPTY(S) **then**
    error "underflow"
else
    $S.top \leftarrow S.top - 1$
    return $S[S.top + 1]$
end if

Pile realizzate con liste





Pile realizzate con liste



Pile realizzate con liste

STACK-LIST-EMPTY(S)
return $S = nil$

STACK-LIST-PUSH(x, S)
$S \leftarrow$ CONS(x, S)

STACK-LIST-POP(S)
if STACK-LIST-EMPTY(S) **then**
    error "underflow"
else
    $x \leftarrow$ HEAD(S)
    $S \leftarrow$ TAIL(S)
    return x
end if

Code: definizione informale

Le code sono strutture lineari i cui elementi si inseriscono da un estremo e si estraggono dall'altro (First In First Out - FIFO)

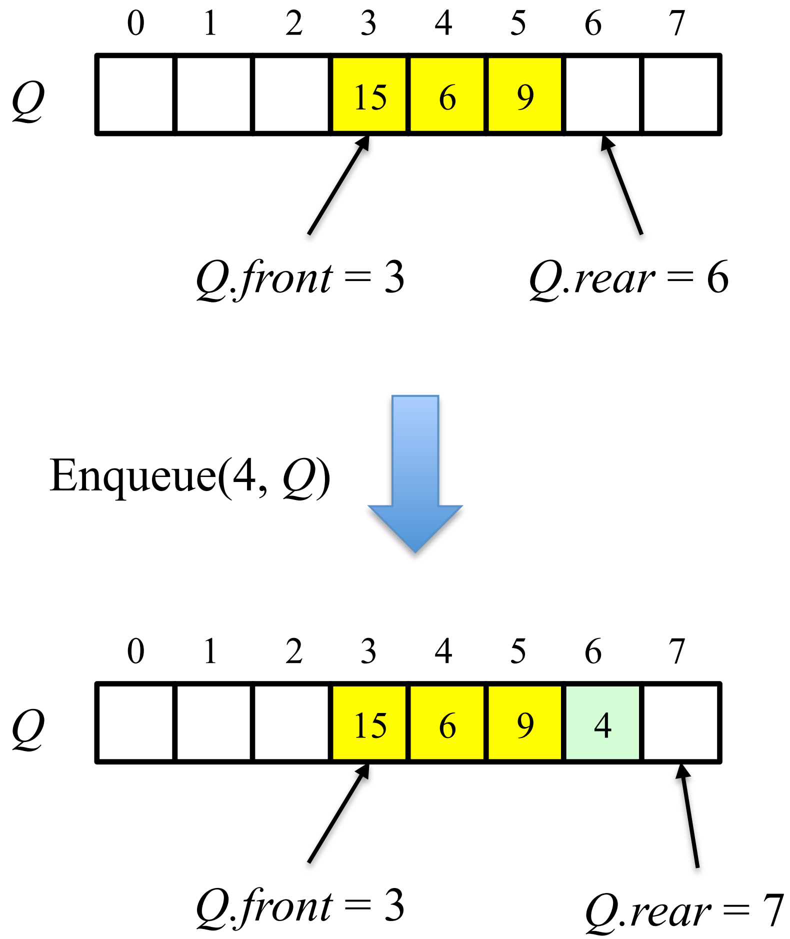
ADT delle code

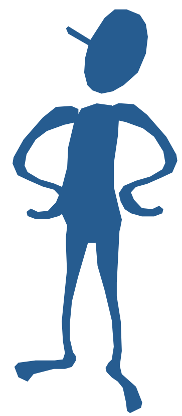
Una coda (*queue*) si definisce astrattamente come una struttura dati su cui siano definite alemeno le operazioni:

Enqueue(e, Q) :	aggiunge e come ultimo in Q
Dequeue(Q) :	elimina il primo da Q e lo ritorna
Empty(Q):	ritorna *true* se Q non ha elementi.

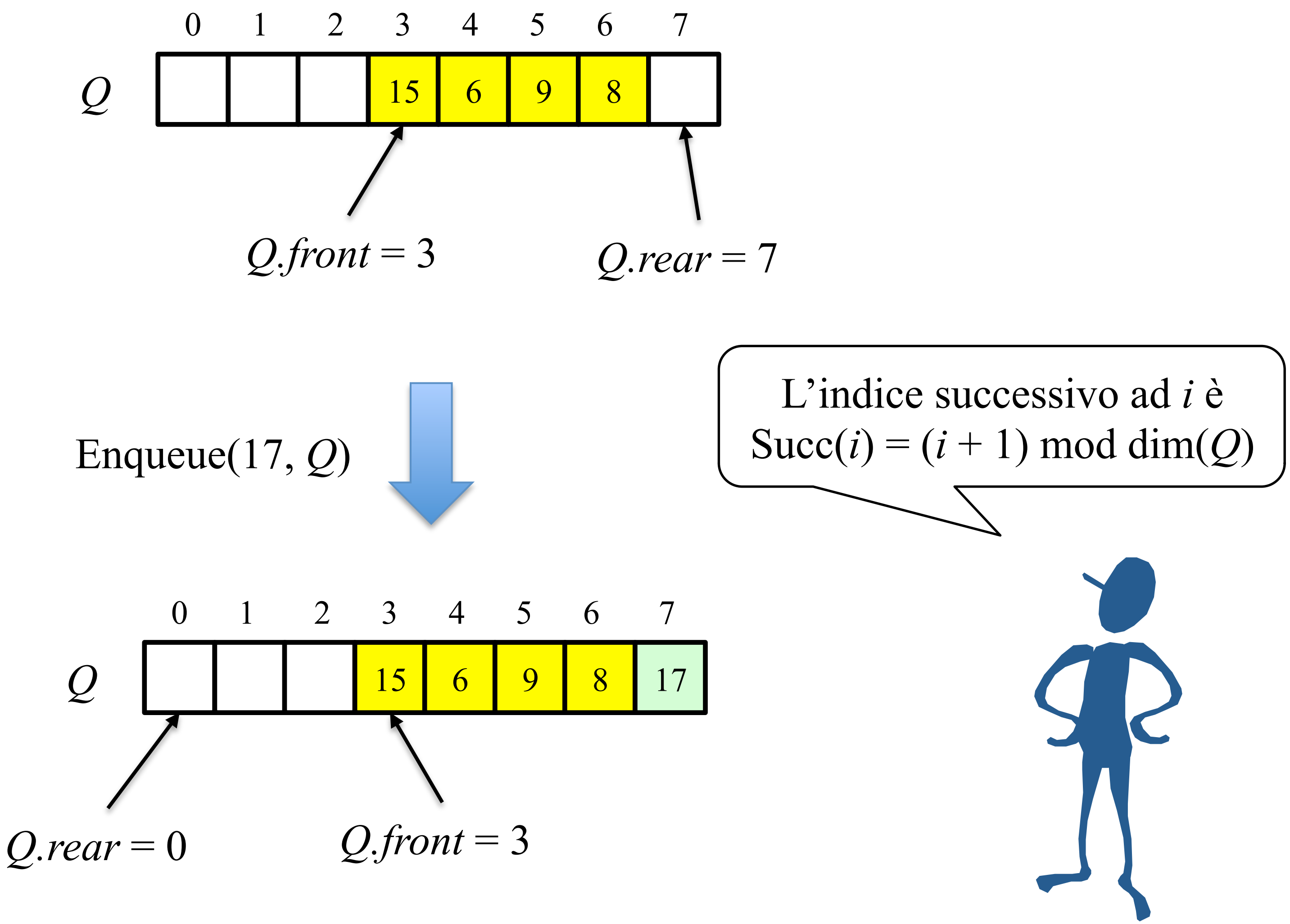
Nota: se Q è vuota Dequeue(Q) è indefinita.

Code realizzate con vettori

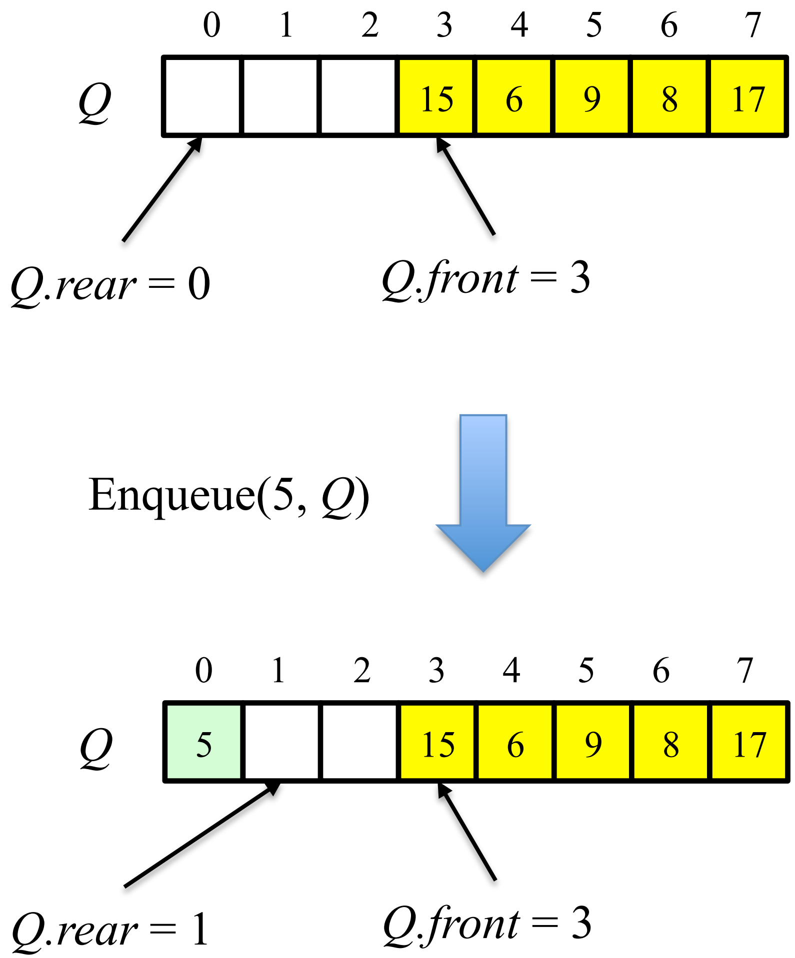


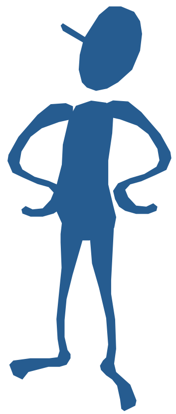


Code realizzate con vettori



Code realizzate con vettori



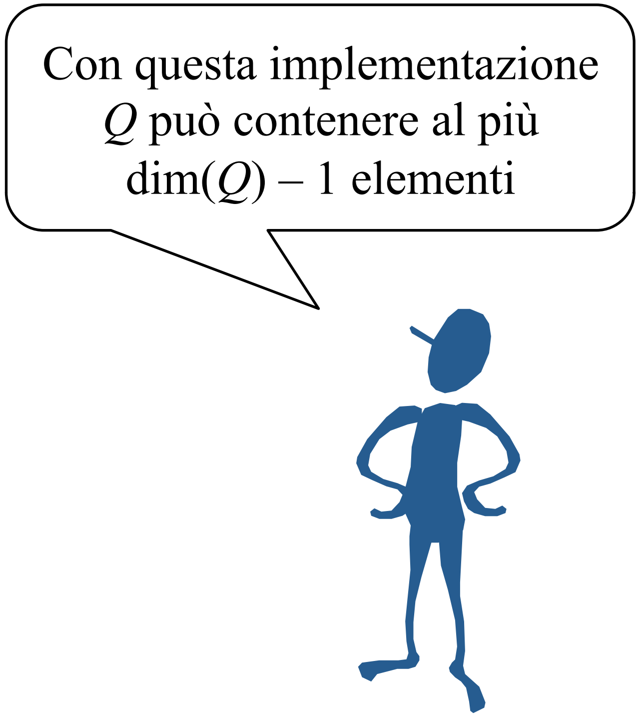


Code realizzate con vettori

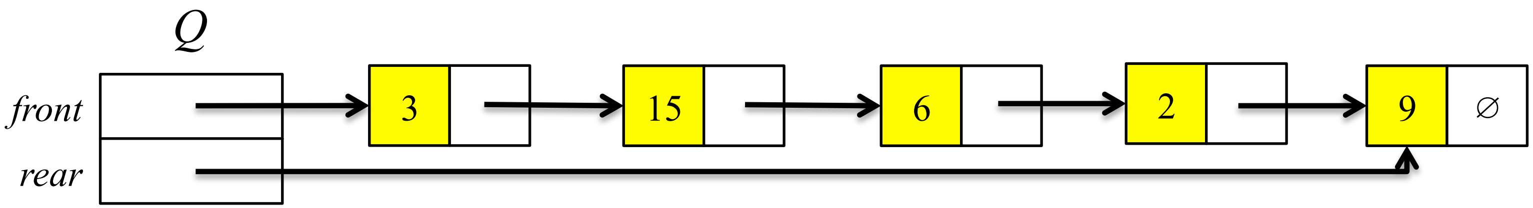
QUEUE-ARRAY-EMPTY(Q)
return $Q.front = Q.rear$

QUEUE-ARRAY-ENQUEUE(x, Q)
$sRear \leftarrow (Q.rear + 1) \bmod dim(Q)$
      $\triangleright$ $sRear$ = successore di $rear$ in Q
if $sRear = Q.front$ **then**
    error “overflow”
else
    $Q[Q.rear] \leftarrow x$
    $Q.rear \leftarrow sRear$
end if

QUEUE-ARRAY-DEQUEUE(Q)
if QUEUE-ARRAY-EMPTY(Q) **then**
    error “underflow”
else
    $x \leftarrow Q[Q.front]$
    $Q.front \leftarrow (Q.front + 1) \bmod dim(Q)$
    return x
end if



Code realizzate con le liste



Code realizzate con le liste

$\textsc{Queue-List-Empty}(Q)$
return $Q.front = Q.rear = nil$

$\textsc{Queue-List-Enqueue}(x, Q)$
if $\textsc{Queue-List-Enqueue}(Q)$ **then**
        $\triangleright$ x è il primo el. immesso in Q
    $Q.front \leftarrow Q.rear \leftarrow \textsc{Cons}(x, nil)$
else
    $Q.rear.next \leftarrow \textsc{Cons}(x, nil)$
    $Q.rear \leftarrow \textsc{Tail}(Q.rear)$
end if

$\textsc{Queue-List-Dequeue}(Q)$
if $\textsc{Queue-List-Empty}(Q)$ **then**
    error “underflow”
else
    $x \leftarrow \textsc{Head}(Q.front)$
    $Q.front \leftarrow \textsc{Tail}(Q.front)$
    if $Q.front = nil$ **then** $\triangleright$ x era il solo el. in Q
        $Q.rear = nil$
    end if
    return x
end if

Capitolo 6

Liste

6.1 Liste

Le liste sono rappresentate con puntatori a record di due campi *head* e *next*, contenenti l'informazione associata all'elemento di testa della lista ed il puntatore alla coda rispettivamente; la lista vuota è rappresentata dal puntatore *nil*. Negli svolgimenti si usino le funzioni $\textsc{Head}(L)$ e $\textsc{Tail}(L)$ che, se $L \neq nil$, ritornano $L.head$ e $L.next$ rispettivamente. La funzione $\textsc{Cons}(x, L)$ alloca un nuovo record N, assegna $N.head \leftarrow x$ e $N.next \leftarrow L$, quindi ritorna N.

Esercizio 1. Si dia lo pseudo-codice della funzione $\textsc{Multiples}(L)$ che data una lista semplice L di n elementi restituisca il numero degli elementi che hanno almeno una copia in L. Ad esempio: se $L = [3, 2, 3, 3, 5, 2]$ allora $\textsc{Multiples}(L) = 2$.
Suggerimento. Una semplice soluzione è $O(n^2)$, ma ne esiste una $O(n \log n)$.

Esercizio 2. Sia il *rango* di un elemento in una lista di interi la somma di quell'elemento con tutti quelli che lo seguono. Si scriva un algoritmo ottimo $\textsc{Rank}(L)$ che distruttivamente modifichi L in modo che gli elementi della lista risultante siano i ranghi degli elementi della lista data. Ad esempio: se $L = [3, 2, 5]$ allora diventa $L = [10, 7, 5]$.

Esercizio 3. Si dia lo pseudo-codice della funzione $\textsc{Reverse}(L)$ che data una lista semplice L di n elementi restituisca la lista inversa L^{-1}, costituita da tutti gli elementi di L in ordine inverso. Si richiede inoltre che $\textsc{Reverse}$ non sia distruttiva, ossia non alteri L, e che operi in tempo $O(n)$.

Esercizio 4. Una lista si dice *palindroma* se se le sue due metà, eliminando eventualmente l'elemento di posto medio se la lunghezza della lista è dispari, sono simmetriche ossia l'una l'inversa dell'altra.

Si dia lo pseudo-codice della funzione $\textsc{Palindrome}(L)$ che data una lista semplice L di n elementi decida se L è palindroma in tempo $O(n)$.

Esercizio 5. Si scriva un algoritmo $\textsc{OddEven}(L)$ ottimo che data una lista L di interi ne riordini distruttivamente gli elementi in modo che i dispari precedano i pari. L'algoritmo deve essere stabile, nel senso che l'ordine relativo tra i dispari e tra i pari deve essere preservato. Ad esempio: se $L = [3, 7, 8, 1, 4]$ allora si ottiene $L = [3, 7, 1, 8, 4]$.
Suggerimento. L'algoritmo $\textsc{OddEven}(L)$ restituisca una tripla $Odd, Last, Even$, dove Odd è la lista degli elementi dispari in L, $Even$ quella dei pari; essendo tuttavia L una lista semplice viene calcolato anche $Last$, che è l'ultimo elemento della lista dei dispari, il cui successivo deve essere $Even$. Si noti che, se in L non ci sono dispari $Odd = Last = nil$, cosa di cui si deve tener conto tanto quando si aggiunge un dispari in testa ad Odd, quanto quando si aggiunge un pari davanti ad $Even$. Una volta eseguito $\textsc{OddEven}(L)$ la lista richiesta sarà Odd se $Odd \neq nil$, $Even$ altrimenti.

Esercizio 6. Supponiamo di rappresentare insiemi finiti di interi con liste ordinate e senza ripetizioni. Si descrivano gli algoritmi $\textsc{Intersection}(A, B)$, $\textsc{Union}(A, B)$, $\textsc{Difference}(A, B)$ e $\textsc{SymDifference}(A, B)$

corrispondenti agli operatori insiemistici:

$$
\begin{array}{rcll}
A \cap B & = & \{x \mid x \in A \wedge x \in B\} & \text{intersezione} \\
A \cup B & = & \{x \mid x \in A \vee x \in B\} & \text{unione} \\
A \setminus B & = & \{x \mid x \in A \wedge x \notin B\} & \text{differenza} \\
A \Delta B & = & \{x \mid (x \in A \wedge x \notin B) \vee (x \in B \wedge x \notin A)\} & \text{differenza simmetrica}
\end{array}
$$

Tutti gli algoritmi devono essere ricorsivi e $O(n)$ dove n è la somma delle cardinalità, ossia delle lunghezze delle liste che rappresentano gli insiemi A, B. Inoltre, sebbene $A \Delta B = (A \cup B) \setminus (A \cap B)$ e la composizione di algoritmi lineari abbia costo lineare, si richiede di implementare $\textsc{SymDifference}(A, B)$ direttamente, ad imitazione degli algoritmi per gli altri operatori.