Insiemi Dinamici

Insiemi che possono essere modificati

Elementi: oggetti =
eventualmente: chiavi + (eventuali) dati satellite;
l'insieme delle chiavi puo' essere totalmente ordinato

Operazioni tipiche (S:struttura, x: riferimento(puntatore) alla chiave, k: chiave)

Search(S,k): restituisce il puntatore x a un elemento in S tale che $key[x] = k$; `NIL` se S non contiene elementi con chiave k (interrogazione)

Insert(S,x): inserisce in S l'elemento puntato da x (modifica)

Delete(S,x): elimina x da S (modifica), x è un puntatore a un elemento

Minimum(S): restituisce l'elemento di S con chiave più piccola (interrogazione)

Maximum(S): restituisce l'elemento di S con chiave più grande (interrogazione)

Inoltre devono essere definiti

Null(S): predicato che riporta true se S è vuoto, false altrimenti

Empty(S): operazione che inizializza S alla rappresentazione dell'insieme vuoto (modifica) oppure

Empty(): funzione che riporta l'insieme vuoto (crea una struttura vuota)

PILE (STACK)

L'elemento rimosso dall'insieme con un'operazione DELETE *è l'elemento inserito più recentemente*

disciplina LIFO: Last-In, First-Out

Operazioni

EMPTY(S) crea la pila vuota

NULL(S) test pila-vuota

PUSH(S,X) = Insert

POP(S) = Delete: cancella da S l'elemento x in cima alla pila S e riporta x stesso

TOP(S) riporta l'elemento in cima alla pila S, senza modificare la pila

Implementazione mediante array

- Una pila di al più n elementi è rappresentata da un array $S[1..n]$.
- L'array ha un attributo $top[S]$: l'indice dell'elemento inserito in cima.
- Gli elementi della pila sono contenuti in $S[1..top[S]]$
  $S[top[S]]$: elemento in cima alla pila
  $S[1]$: elemento in fondo alla pila
- Pila vuota: $top[S] = 0$

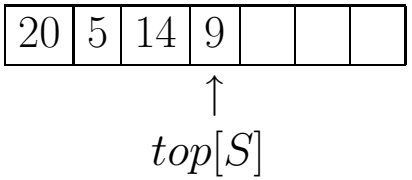
```
EMPTY(S)
1.  top[S] <- 0
```

```
NULL(S)
1. return top[S] = 0
```

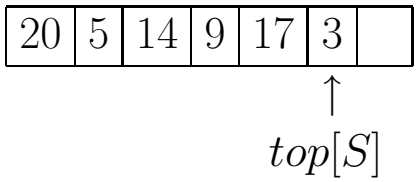
Implementazione mediante array

```
TOP(S)
1. if NULL(S)
2.    then error "underflow"
3.    else return S[top[S]]

PUSH(S,x)
1. if top[S] = length[S]
2.    then error "overflow"
3.    else top[S] <- top[S] + 1
4.         S[top[S]] <- x

POP(S)
1. if NULL(S)
2.    then error "underflow"
3.    else top[S] <- top[S] - 1
4.         return S[top[S] + 1]
```

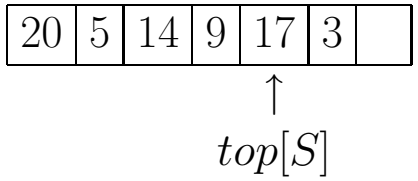
Esempio Implementazione



Dopo l'esecuzione di `PUSH(S,17)` e `PUSH(S,3)`:



Dopo l'esecuzione di `POP(S)`, che riporta 3:



Ora `TOP(S)` *– da non confondere con* $top[S]$ *– è uguale a* 17

CODE (QUEUE)

L'elemento rimosso dall'insieme con un'operazione DELETE *è l'elemento inserito da più tempo*

disciplina FIFO: first-in, first-out

Operazioni

EMPTY(S) crea la coda vuota

NULL(S) test coda-vuota

ENQUEUE(S,X) = Insert

DEQUEUE(S) = Delete: cancella da S l'elemento x in testa alla coda e riporta x stesso

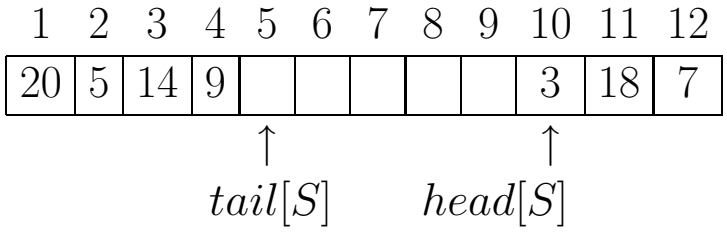
FRONT(S) riporta l'elemento in testa alla coda S, senza modificare la coda

Implementazione mediante array

- Una coda di al più $n-1$ elementi è rappresentata da un array $S[1..n]$.
- Attrubuti della coda: $head[S]$: indice della "testa" della coda: $S[head[S]]$ è l'elemento in testa
  $tail[S]$: indice della locazione in cui si inserirà il prossimo elemento
- Gli elementi della coda sono nelle posizioni con indici:

$$head[S], head[S]+1, ..., tail[S]-1$$

ma l'array è considerato circolare: dopo la locazione $length[S]$ viene la locazione 1.



- La coda è vuota quando $head[S] = tail[S]$
- inizialmente $head[S] = tail[S]$
- La coda è piena quando $head[S] = tail[S]+1$ oppure $tail[S] = length[S]$ e $head[S] = 1$ (una posizione resta sempre vuota)

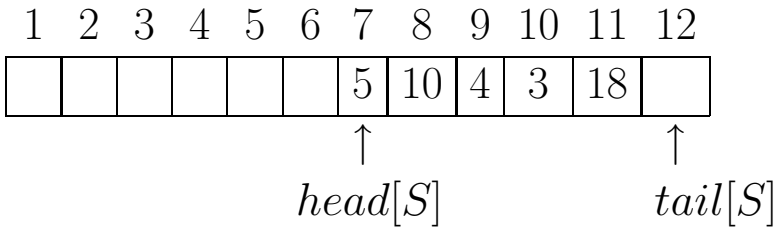
Implementazione mediante array (II)

```
EMPTY(S)                NULL(S)                          FRONT(S)
1.  head[S] <- 1        1. return head[S] = tail[S]      1. return S[head[S]]
2.  tail[S] <- 1
```

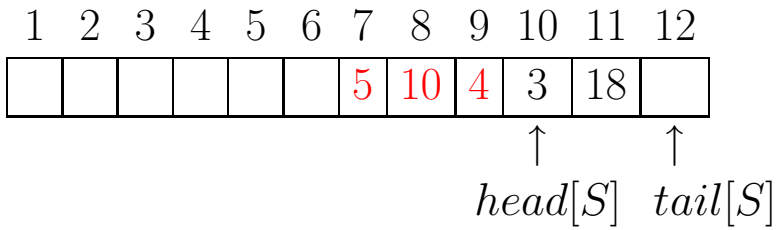
```
ENQUEUE(S,x)
1. if head[S] = (tail[S]+1) or (tail[S] = length[S] and head[S] = 1)
2.    then error "overflow"
3.    else S[tail[S]] <- x
           ;; tail[S] si sposta a destra
4.         if tail[S] = length[S]
5.            then tail[S] <- 1
6.            else tail[S] <- tail[S] + 1
```

```
DEQUEUE(S)
1. if NULL(S)
2.    then error "underflow"
3.    else x <- S[head[S]]
           ;; head[S] si sposta a destra
4.         if head[S] = length[S]
5.            then head[S] <- 1
6.            else head[S] <- head[S] + 1
7.         return x
```

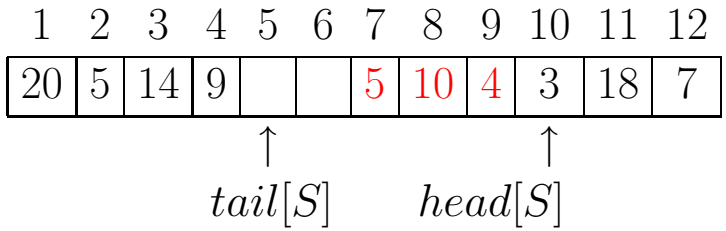
Esempio Implementazione



Dopo l'esecuzione di tre operazioni `DEQUEUE[S]`, che riportano, rispettivamente, 5, 10 e 4:



Dopo l'esecuzione delle operazioni `ENQUEUE(S,7 )`,
`ENQUEUE(S,20)`, `ENQUEUE(S,5)`, `ENQUEUE(S,14)` e `ENQUEUE(S,9)`:



N.B.: gli elementi in rosso non appartengono alla coda e possono essere sovrascritti

Liste concatenate

Insiemi di elementi di dimensione massima non prefissata

L'inserimento e la cancellazione avvengono sempre "in testa" alla lista

Operazioni (non "distruttive"):

Empty() riporta la lista vuota

Null(L) test lista-vuota

Insert(L,x) riporta la lista che si ottiene inserendo x in testa a L

Rest(L) riporta la lista che si ottiene cancellando da L l'elemento x in testa alla lista

First(L) riporta l'elemento in testa alla lista L, senza modificarla

Setlist(L,L') modifica la lista L ponendola uguale a L'.

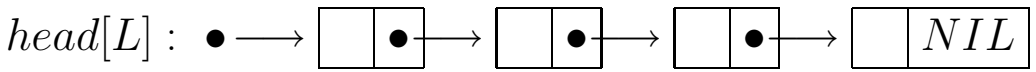
Le liste possono essere rappresentate mediante array come le pile

Un elemento x di una lista è un oggetto con almeno due campi
(ed eventualmente altri campi per dati satellite)

$key[x]$ campo chiave
$next[x]$ puntatore al successore dell'oggetto nella lista

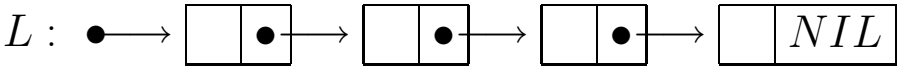
Una lista L è un oggetto con un campo:

$head[L]$ puntatore al primo elemento x della lista



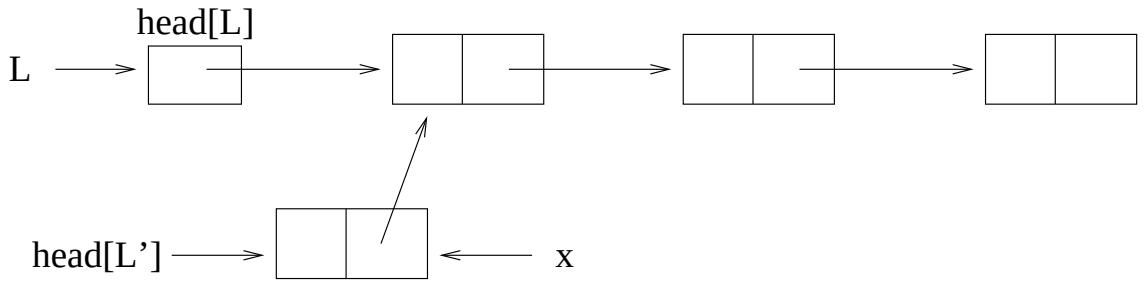
Lista vuota: $head[L] = NIL$

Se il linguaggio fornisce puntatori, la lista è rappresentata semplicemente da un puntatore a un elemento:



Implementazione delle operazioni di base

```
EMPTY()
1. ;; eventuale allocazione della memoria per L
2. head[L] <- NIL
3. return L

NULL(L)
1. return (head[L] = NIL)

INSERT(L,x)
1. ;; eventuale allocazione della memoria per L'
2. next[x] <- head[L]
3. head[L'] <- x
4. return L'
```



Implementazione delle operazioni di base (II)

```
REST(L)
1. if NULL(L)
2.    then error "empty list"
3.    else
4.         ;; eventuale allocazione della memoria
           ;; per L'
5.         head[L'] <- next[head[L]]
6.         return L'

FIRST(L)
1. if NULL(L)
2.    then error "empty list"
3.    else return head[L]
           ;; viene riportato un puntatore a un oggetto
```

Altre operazioni di base

Operazione di assegnazione per le liste

```
SETLIST(L,L')
1. head[L] <- head[L']
```

Occorre, eventualmente, definire anche un'operazione per liberare la memoria da un oggetto lista

Astrazione sul tipo di dati elemento:

```
KEY(x)
1. return key[x]

NEXT(x)
1. return next[x]

SET-KEY(x,k)
1. key[x] <- k

SET-NEXT(x,y)
1. next[x] <- y

MK-ELEM()
1. ;; allocazione della memoria per x
2. return x
```

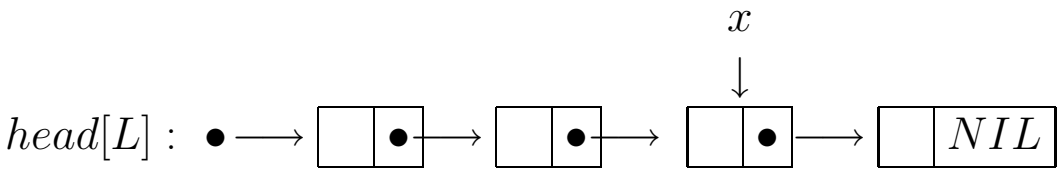
Occorre, eventualmente, definire anche un'operazione per liberare la memoria da un elemento

Ricerca di un elemento in una lista

```
SEARCH(L,k)
1. if NULL(L)
2.    then return NIL
3.    else x <- FIRST(L)
4.         while x != NIL and KEY(x) != k
5.            do x <- NEXT(x)
6.         return x
      ;; riporta un puntatore all'oggetto
      ;; NIL se la ricerca fallisce
```

Cancellazione di un elemento da una lista

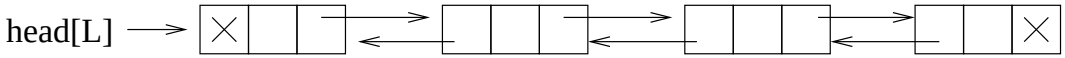
x è un puntatore a un elemento:




```
DELETE(L,x)
1. if not NULL(L) ;; se la lista e' vuota non
                  ;; c'e' niente da cancellare
2.    then if FIRST(L) = x     ;; x e' il primo elemento
3.            then SETLIST(L,REST(L))
4.         else y <- FIRST(L)
                ;; ricerca del predecessore di x nella lista
5.              while NEXT(y) != NULL and NEXT(y) != x
6.                 do y <- NEXT(y)
                ;; ora se NEXT(y) != NULL, y e'
                ;; il predecessore di x
7.              if NEXT(y) != NULL
8.                 then SET-NEXT (y,NEXT(x))
```

Liste doppie - Ricerca

Ogni elemento ha un puntatore al successore a uno al predecessore



Ogni elemento ha, oltre al campo *chiave* e al campo *next*, anche un campo *prev*:
$prev[x]$ punta al predecessore di x nella lista.

Il primo elemento x della lista ha $prev[x] = NIL$

Ricerca di un elemento

```
SEARCH(L,k)
1. x <- head[L]
2. while x != NIL e key[x] != k
3.    do x <- next[x]
4. return x
```

Liste doppie - Inserimento e Cancellazione

Inserimento in testa (operazione di modifica)

```
INSERT(L,x)
1. next[x] <- head[L]
2. if not NULL(head[L])
3.    then prev[head[L]] <- x
4. head[L] <- x
5. prev[x] <- NIL
```

Cancellazione

```
DELETE(L,x)
1. if prev[x] != NIL
2.    then next[prev[x]] <- next[x]
3.    else head[L] <- next[x]
4. if next[x] != NIL
5.    then prev[next[x]] <- prev[x]
```

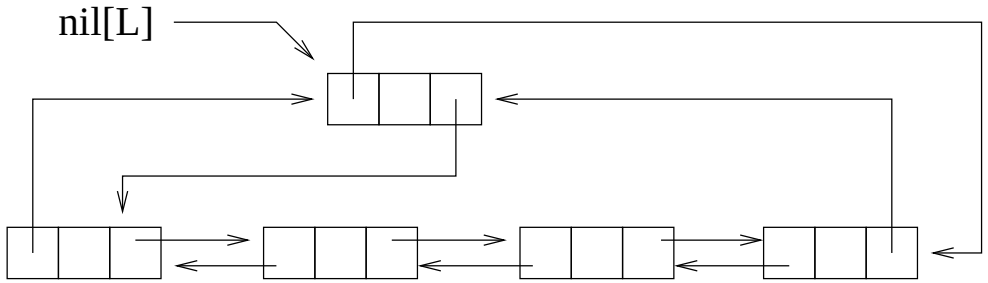
Sentinelle

Una "sentinella" è un oggetto fittizio che consente di semplificare i casi limite

Il codice di DELETE potrebbe essere più semplice se non dovessimo trattare a parte i casi in cui x è il primo o l'ultimo elemento della lista:

```
DELETE(L,x)
1. next[prev[x]] <- next[x]
2. prev[next[x]] <- prev[x]
```

- Si fornisce la lista L di un campo $nil[L]$ che serve a rappresentare NIL.
- $nil[L]$ è un oggetto dello stesso tipo degli elementi della lista.
- Ogni valore NIL nella rappresentazione standard ($prev[x]$ se x è il primo elemento e $next[x]$ se x è l'ultimo elemento) si sostituisce con un riferimento alla sentinella $nil[L]$.



- Il campo $next[nil[L]]$ punta al primo elemento della lista, $prev[nil[L]]$ punta all'ultimo elemento della lista
- Il primo elemento della lista (il vecchio $head[L]$) è dunque $next[nil[L]]$.
- Lista vuota: $next[nil[L]] = nil[L]$ e $prev[nil[L]] = nil[L]$.

Alcune operazioni di base

```
EMPTY(L)
1. next[nil[L]] <- nil[L]
2. prev[nil[L]] <- nil[L]
```

```
DELETE(L,x)
1. next[prev[x]] <- next[x]
2. prev[next[x]] <- prev[x]
```

```
SEARCH(L,k)
1. x <- next[nil[L]]
2. while x != nil[L] e key[x] != k
3.    do x <- next[x]
4. return x
```

```
INSERT(L,x)
1. next[x] <- next[nil[L]]
2. prev[next[nil[L]]] <- x
4. next[nil[L]] <- x
5. prev[x] <- nil[L]
```

Realizzazione di puntatori a oggetti: rappresentazione di liste semplici mediante array

Per rappresentare una lista utilizziamo due (o più) array

- *key* : contenente i valori delle chiavi
- *next* : contenente i riferimenti ai successori
- *head* :(intero) indice del primo elemento della lista
- Un riferimento è un indice dell'array

Esempio

rappresentazione della lista (4,5,1,21,45)

$head = 3$

key		21	4	45		5		1		
$next$		4	6	0		8		2		
	1	2	3	4	5	6	7	8	9	10

A è una struttura con tre componenti:
$head$ (un intero), key e $next$ (due array della stessa dimensione).

```
EMPTY(A)
1. head <- 0
```

```
NULL(A)
1. return (head = 0)
```

```
FIRST(A)
1. return key[head]
```

Per la realizzazione delle operazioni `INSERT` e `REST`, è necessario conservare memoria delle posizioni libere nell'array.

Per far ciò si utilizza una *lista libera*, all'interno dello stesso array, che collega tutte le posizioni libere:

$head = 3,\ free = 7$

key		21	4	45		5		1		
$next$	10	4	6	0	1	8	5	2	0	9
	1	2	3	4	5	6	7	8	9	10

La struttura che rappresenta una lista ha dunque un'ulteriore componente: $free$ (un intero), l'indice della prima posizione libera

Operazioni con le Liste Libere

Inizializzazione di una lista libera

```
EMPTY(A)
1. head <- 0
2. free <- 1
3. for i=1 to length[next]-1
4.      do next[i] <- i+1
5. next[length[next]] <- 0
```

Quando `free = 0`, l'array è pieno.

Allocazione di Oggetti

Una posizione viene prelevata dalla lista libera. La funzione riporta l'indice corrispondente

```
ALLOCATE (A)
1. if free=0
2.    then error "Lista piena"
3.    else x <- free ;; la prossima posizione libera
4.         free <- next[x]
                        ;; la lista libera cede la sua
                        ;; prima posizione
5.         return x
```

Deallocazione di Oggetti

Una posizione viene "liberata" e reinserita nella lista libera

```
FREE(A,x)
1. next[x] <- free
2. free <- x
```

Inserimento in testa con modifica della lista

Quando si inserisce un nuovo elemento, si preleva una componente dalla lista libera

```
INSERT(A,x)
1. i = ALLOCATE(A)
     ;; assumiamo che l'eventuale errore "Lista piena"
     ;; si propaghi automaticamente
2. key[i] <- x
3. next[i] <- head
4. head <- i
```

Cancellazione del primo elemento

Quando si cancella un elemento, si "libera" la sua posizione

```
DELETE-FIRST(A)
1. if NULL(A)
2.    then error "Lista vuota"
3.    else i <- head
4.         head <- next[i]
5.         FREE(i)
```

Esercizi

V-1 Scrivere codice per un'operazione che riporti l'ultimo elemento di una lista (non vuota). Considerare i diversi modi di rappresentare le liste (incluse liste doppie con sentinella).

V-2 Scrivere codice per un'operazione che, date due liste, riporti la loro concatenazione. Considerare i diversi modi di rappresentare le liste.

V-3 Cosa succede alla lista L se si elimina l'elemento x da REST(L)?

V-4 Scrivere il codice della procedura REST per le liste doppie e per la rappresentazione mediante array.