Insiemi dinamici finiti

(ovvero il cui contenuto cambia nel tempo)

Il modo migliore di realizzare un insieme dinamico dipende dalle operazioni che devono essere fornite

Tipica realizzazione: ogni elemento è un oggetto di cui si conosce il puntatore x

Campi dell'oggetto x = chiave (identificatore univoco), rappresentata dall'attributo key[x] + dati satellite

Insiemi dinamici: operazioni di modifica

Operazioni	Vincoli e semantica
INSERT(S, x)	Si assume che tutti i campi dell'oggetto puntato da x siano già stati inizializzati
DELETE(S, x)	

Insiemi dinamici: operazioni di interrogazione

Operazioni	Vincoli e semantica
SEARCH(S, k)	Restituisce un puntatore x a un elemento di S tale che key[x] = k oppure NIL se un tale elemento non esiste in S
MINIMUM(S)	Assume che S sia totalmente ordinato; restituisce l'elemento con la chiave più piccola
MAXIMUM(S)	Assume che S sia totalmente ordinato; restituisce l'elemento con la chiave più grande
SUCCESSOR(S, x)	Assume che S sia totalmente ordinato; restituisce l'elemento successivo (dotato di chiave >) o NIL se x è l'elemento massimo
PREDECESSOR(S, x)	Assume che S sia totalmente ordinato; restituisce l'elemento precedente (dotato di chiave <) o NIL se x è l'elemento minimo

N.B. SUCCESSOR e PREDECESSOR sono spesso estese a insiemi con "chiavi" non univoche

Pila

Insieme con politica LIFO

INSERT diventa PUSH
DELETE diventa POP

Può essere realizzata mediante un array S[1 .. top[S]] dove l'attributo top[S] è l'indice dell'elemento inserito più di recente →

S[1] = elemento in fondo alla pila
S[top[S]] = elemento in cima alla pila

Se top[S] = 0, la pila è vuota

Pila: operazioni

```
STACK-EMPTY(S)
1 if top[S] = 0
2    then    return TRUE
3    else    return FALSE
```

O(1)

```
PUSH(S, x)
1 top[S] ← top[S] + 1
2  S[top[S]] ← x
```

O(1)

(manca gestione errori)

```
POP(S)
1 if STACK-EMPTY(S)
2    then    error "underflow"
3    else    top[S] ← top[S] – 1
4            return S[top[S] + 1]
```

O(1)

Coda

Insieme con politica FIFO

INSERT diventa ENQUEUE
DELETE diventa DEQUEUE

Una coda di al più n – 1 elementi può essere realizzata mediante un array Q[1 .. n] dove si procede in modo circolare, cioè la locazione 1 segue immediatamente la locazione n

Attributi della coda Q:

- head[Q] = indice della testa
- tail[Q] = indice della prossima locazione che sarà occupata →
  tail[Q] – 1 = indice dell'ultimo elemento in coda

Se head[Q] = tail[Q], la coda è vuota; inizialmente è head[Q] = tail[Q] = 1
Se head[Q] = tail[Q] + 1, la coda è piena

Coda: operazioni

```
ENQUEUE(Q, x)
1 Q[tail[Q]] ← x
2 if tail[Q] = length[Q]
3    then    tail[Q] ← 1
4    else    tail[Q] ← tail[Q] + 1
```

O(1)

(manca gestione errori)

```
DEQUEUE(Q)
1 x ← Q[head[Q]]
2 if head[Q] = length[Q]
3    then    head[Q] ← 1
4    else    head[Q] ← head[Q] + 1
5 return x
```

O(1)

(manca gestione errori)

Lista concatenata

Insieme dotato di ordine lineare degli elementi

Attributo della lista L:
head[L] = puntatore al primo elemento

Se head[L] = NIL, la lista è vuota

Lista concatenata semplice

Ogni oggetto contiene un campo puntatore (next) al prossimo elemento della lista

next[x] = NIL $\rightarrow$ x è l'ultimo elemento, o coda, della lista

Lista concatenata doppia (o lista bidirezionale)

Ogni oggetto, oltre a next, contiene il campo puntatore prev all'elemento precedente della lista

next[x] = NIL $\rightarrow$ x è l'ultimo elemento, o coda, della lista
prev[x] = NIL $\rightarrow$ x è il primo elemento, o testa, della lista

N.B. $\exists$ x $\mid$ next[x] = NIL $\vee$ prev[x] = NIL sse la lista non è circolare

Lista ordinata e non

È una classificazione ortogonale rispetto alla precedente (concatenata semplice o doppia)

L è ordinata se l'ordine lineare della lista corrisponde all'ordine lineare delle chiavi dei suoi elementi

Lista circolare

È una classificazione che riguarda solo le liste bidirezionali ed è ortogonale rispetto alla classificazione precedente (ordinata e non)

La lista bidirezionale L è circolare se prev della testa di L punta alla coda di L e next della coda di L punta alla testa

Lista bidirezionale non ordinata (non circolare): operazioni

```
LIST-SEARCH(L, k)
0 ▷ ricerca lineare
1 x ← head[L]
2 while x ≠ NIL e key[x] ≠ k
3    do  x ← next[x]
4 return x
5 ▷ restituisce il primo oggetto con
  chiave k, o NIL se nella lista non
  compare nessun oggetto con chiave k
```

$\Theta(n)$
nel caso pessimo

Lista bidirezionale non ordinata (non circolare): operazioni (cont.)

```
LIST-INSERT(L, x)
1 next[x] ← head[L]
2 if head[L] ≠ NIL
3    then    prev[head[L]] ← x
4 head[L] ← x
5 prev[x] ← NIL
```

O(1)

```
LIST-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]
```

O(1)
ma diventa $\Theta(n)$ (nel caso pessimo) se si cancella un elemento con una chiave data, per cui prima si invoca LIST-SEARCH

Sentinella di una lista bidirezionale non ordinata

Oggetto fittizio (rappresenta NIL) della lista L che semplifica la gestione dei casi limite relativi alla testa e alla coda di L

Attributo della lista L:
nil[L] = puntatore alla sentinella

Ogni riferimento a NIL deve essere sostituito con un riferimento alla sentinella
$\rightarrow$ L diventa circolare dove la sentinella è posta fra la testa e la coda

next[nil[L]] = puntatore alla testa di L ($\rightarrow$ head[L] non serve più)
prev[nil[L]] = puntatore alla coda di L

Se L è vuota, contiene solo la sentinella, dove
next[nil[L]] = nil[L]
prev[nil[L]] = nil[L]

Lista bidirezionale non ordinata con sentinella: operazioni

```
LIST-DELETE'(L, x)
1   next[prev[x]] ← next[x]
2   prev[next[x]] ← prev[x]
```

O(1)

```
LIST-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
```

Θ(n)
nel caso pessimo

```
LIST-INSERT'(L, x)
1 next[x] ← next[nil[L]]
2 prev[next[nil[L]]] ← x
3 next[nil[L]] ← x
4 prev[x] ← nil[L]
```

O(1)

Sentinelle: pro e contro

L’uso delle sentinelle all’interno di cicli riguarda la chiarezza del codice piuttosto che la velocità

Se vi sono molte piccole liste, la memoria extra usata per le sentinelle può costituire uno spreco da evitare

Realizzazione di strutture concatenate senza puntatori espliciti

Può avvenire

- con array multipli
- con un singolo array

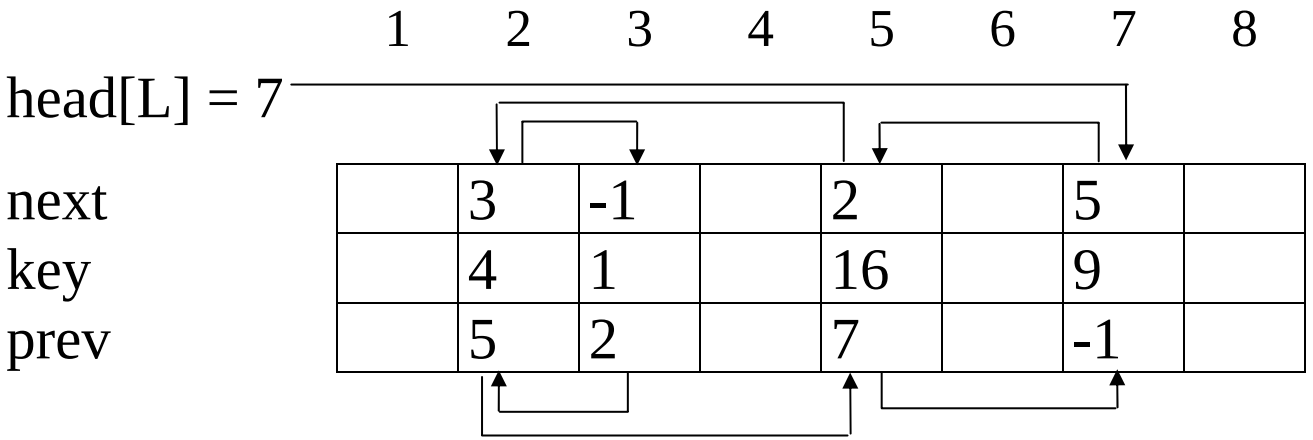
Strutture concatenate realizzate con array multipli

Si può usare un array per ogni campo degli elementi, cioè

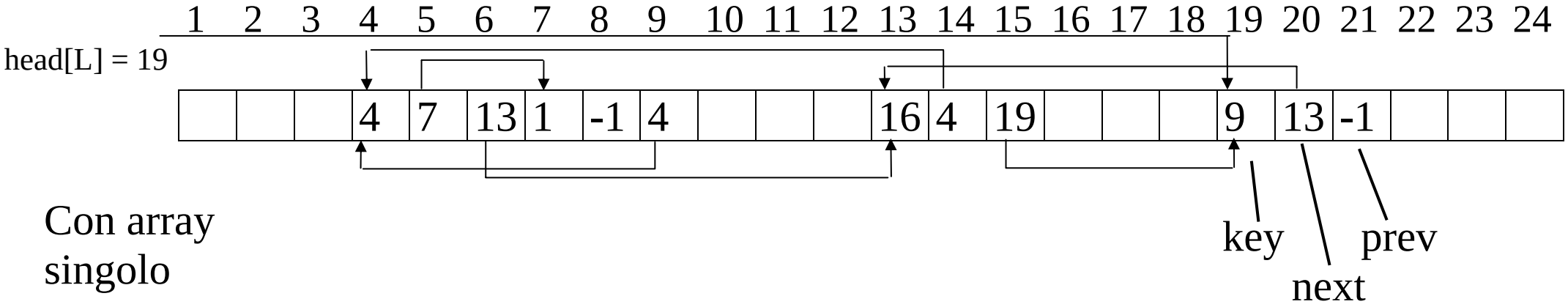
array next: contiene il campo next di tutti gli elementi
array prev: contiene il campo prev di tutti gli elementi
array key: contiene il campo chiave di tutti gli elementi
…

Accostando gli array come se fossero le righe di un'unica matrice, un oggetto è rappresentato da una colonna della matrice stessa
I puntatori si realizzano con gli indici (comuni) degli array (alla costante NIL si fa corrispondere un valore convenzionale che sicuramente non è un indice degli array, ad es. –1)

Realizzazione di strutture concatenate senza puntatori espliciti (cont.)



Con array multipli



Con array singolo

Strutture concatenate realizzate con un singolo array

Nella memoria del calcolatore un oggetto occupa un insieme contiguo di locazioni; un puntatore è l'indirizzo della prima di tali locazioni mentre le altre sono indirizzate aggiungendo un Δ (spostamento)

$\downarrow$

Analogia

Memoria $\rightarrow$ singolo array
Oggetto $\rightarrow$ sottoarray contiguo
Puntatore $\rightarrow$ indice del primo campo dell'oggetto
Puntatore + spostamento $\rightarrow$ indice di un campo dell'oggetto

Si tratta di una rappresentazione flessibile perché permette di memorizzare oggetti di lunghezza diversa (che però si usano raramente)

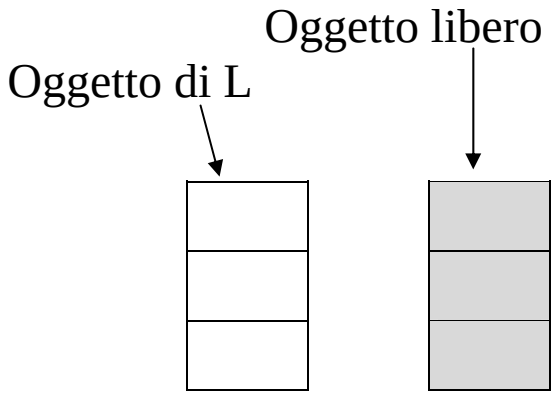
Strutture concatenate realizzate con array multipli

È conveniente gestire gli spazi dell'array non attualmente utilizzati per potere allocare nuovi oggetti (analogia con garbage collection)

Gli oggetti liberi vengono legati in una lista concatenata semplice, detta lista libera, che è gestita come una pila la cui testa è mantenuta nella variabile globale free → ogni oggetto dell'array è o nella lista L o nella lista libera (ma non in entrambe)

head[L] = 7
free = 4

	1	2	3	4	5	6	7	8
next	-1	3	-1	8	2	1	5	6
key		4	1		16		9	
prev		5	2		7		-1	



Si può usare una sola lista libera anche per servire più liste concatenate

Strutture concatenate realizzate con array multipli: operazioni

ALLOCATE-OBJECT()
0 ▷ variante di POP (sulla lista libera)
1 **if** free = NIL
2 **then** **error** “fine dello spazio libero”
3 **else** x ← free
4 free ← next[x]
5 **return** x

O(1)

FREE-OBJECT(x)
0 ▷ variante di PUSH (sulla lista libera)
1 next[x] ← free
2 free ← x

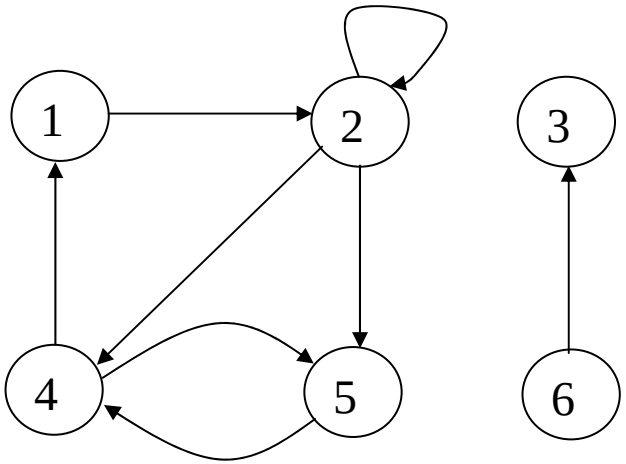
O(1)

Grafi

Grafo G = (V, E)

V = insieme finito dei vertici
E = insieme finito degli archi, dove ogni arco è una coppia di vertici

Grafo orientato
Ogni arco è una coppia ordinata e i due vertici appartenenti all'arco possono anche coincidere (l'arco da un vertice a se stesso si chiama cappio)

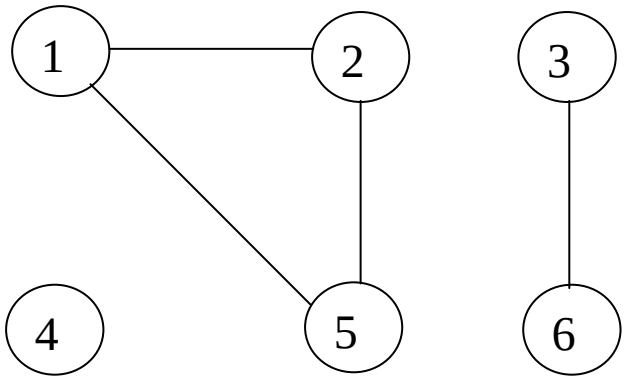


Esempio di grafo orientato, ciclico e non connesso
V = {1, 2, 3, 4, 5, 6}
E = {(1, 2), (2, 2), (2, 4), (2, 5), (4, 1), (4, 5), (5, 4), (6, 3)}

Grafi (cont.)

Grafo non orientato

Ogni arco è una coppia non ordinata e i due vertici appartenenti all'arco non possono coincidere



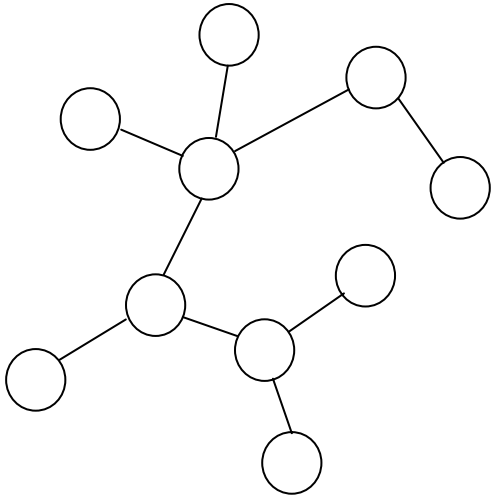
Esempio di grafo non orientato, ciclico e non connesso

V = {1, 2, 3, 4, 5, 6}

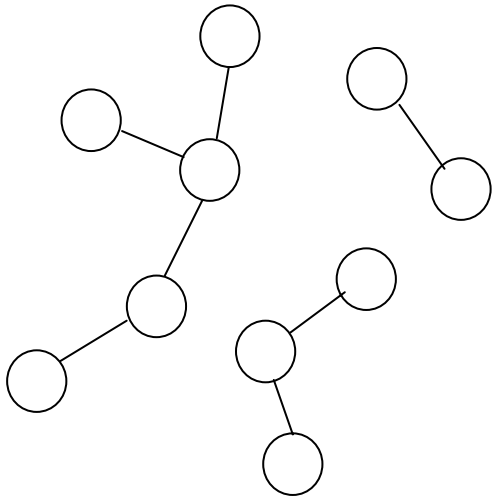
E = {(1, 2), (1, 5), (2, 5), (3, 6)}

Alberi

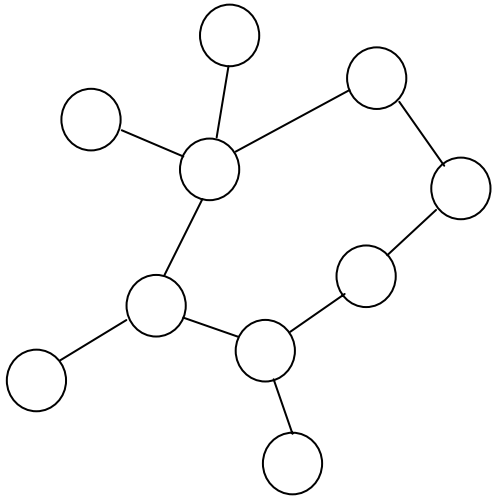
Albero libero = grafo non orientato, connesso e acicliclo
Foresta = grafo non orientato acicliclo che può essere sconnesso



Albero libero



Foresta



Né albero né foresta

Alberi (cont.)

Albero radicato = albero libero in cui un vertice, detto radice, si distingue dagli altri (i vertici degli alberi radicati vengono spesso chiamati nodi)

Albero binario = albero radicato meglio descritto in modo ricorsivo = struttura definita su un insieme finito di nodi che:

- non contiene alcun nodo (albero vuoto o nullo)
- è composta da tre insiemi disgiunti di nodi: la radice, un albero binario chiamato sottoalbero sinistro e un albero binario chiamato sottoalbero destro

Gli alberi radicati possono essere rappresentati usando puntatori espliciti oppure array

Alberi binari

Attributo dell'albero T:
root[T] = puntatore alla radice

root[T] = NIL $\leftrightarrow$ T è vuoto

Ogni oggetto nodo contiene tre campi puntatori:

p = puntatore al padre
left = puntatore al figlio sinistro
right = puntatore al figlio destro

p[x] = NIL $\leftrightarrow$ x è la radice
left[x] = NIL $\leftrightarrow$ x non ha il figlio sx
right[x] = NIL $\leftrightarrow$ x non ha il figlio dx

Alberi radicati con un numero limitato di figli

Se il numero max di figli di ogni nodo è k > 2 ($\rightarrow$ albero k-ario), con k noto a priori, si sostituiscono left e right con

$child_1$, $child_2$, …, $child_k$

dove a ogni campo child compete un array nel caso di rappresentazione (di un albero) con array multipli

Ciò però comporta uno spreco di memoria se k è grande e molti nodi hanno pochi figli

Alberi radicati con un numero arbitrario di figli

Ogni oggetto contiene tre campi puntatori:

p = puntatore al padre
left-child = puntatore al figlio sinistro
right-sibling = puntatore al fratello destro

L'albero ottenuto è quindi un albero binario che usa solo uno spazio O(n) per rappresentare un qualunque albero radicato con n nodi

Alberi radicati con un numero arbitrario di figli (cont.)