

Algoritmi e Strutture Dati

Strutture dati speciali

Alberto Montresor

Università di Trento

2019/08/12

This work is licensed under a Creative Commons
Attribution-ShareAlike 4.0 International License.



Sommario

- 1 Introduzione
- 2 Code con priorità
 - Introduzione
 - Vettore heap
 - HeapSort
 - Implementazione code
- 3 Insiemi disgiunti
 - Introduzione
 - Realizzazione basata su liste
 - Realizzazione basata su alberi
 - Euristiche

Introduzione

Strutture dati viste finora

- Sequenze, insiemi e dizionari

Strutture “speciali”

- Se non tutte le operazioni sono necessarie, è possibile realizzare strutture dati più efficienti, “specializzate” per particolare compiti
- Le operazioni base (inserimento, cancellazione, lettura, etc.) possono non essere sufficienti: a volte servono operazioni speciali

Esempi

- Code con priorità
- Insiemi disgiunti
- Interval/segment tree
- K-D tree
- Trie
- Fenvick Tree
- Merkle tree
- Secondo Wikipedia, 338 pagine nella categoria strutture dati...

Code con priorità

Definizione (**Priority Queue**)

Una **coda con priorità** è una struttura dati astratta, simile ad una coda, in cui ogni elemento inserito possiede una sua "**priorità**"

- **Min-priority queue**: estrazione per valori crescenti di priorità
- **Max-priority queue**: estrazione per valori decrescenti di priorità

Operazioni permesse

- Inserimento in coda
- Estrazione dell'elemento con priorità di valore min/max
- Modifica priorità (decremento/incremento) di un elemento inserito

Specifica

MINPRIORITYQUEUE

% Crea una coda con priorità, vuota

MinPriorityQueue()

% Restituisce **true** se la coda con priorità è vuota

boolean isEmpty()

% Restituisce l'elemento minimo di una coda con priorità non vuota

ITEM min()

% Rimuove e restituisce il minimo da una coda con priorità non vuota

ITEM deleteMin()

% Inserisce l'elemento x con priorità p nella coda con priorità. Restituisce

% un oggetto **PRIORITYITEM** che identifica x all'interno della coda

PRIORITYITEM insert(ITEM x , **int** p)

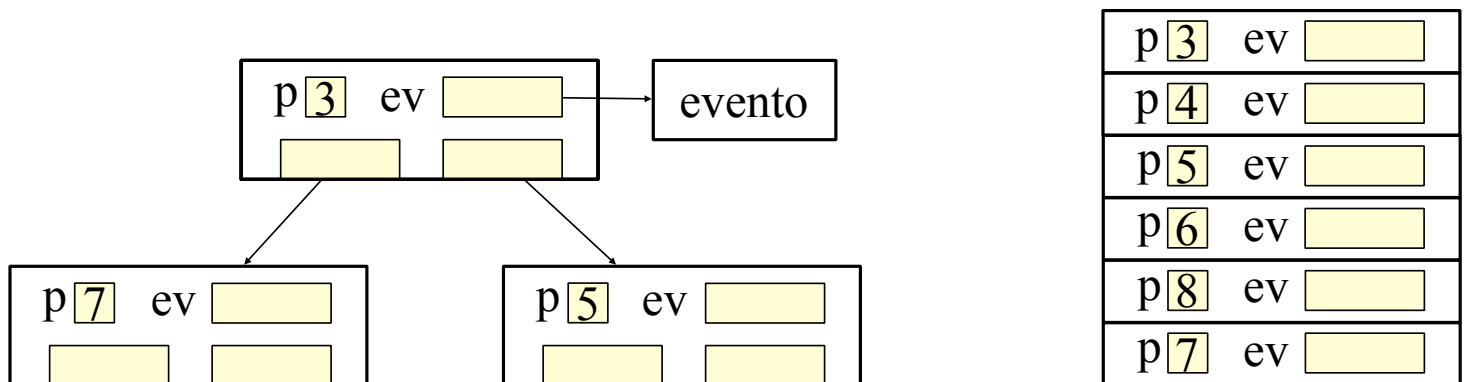
% Diminuisce la priorità dell'oggetto identificato da y portandola a p

decrease(**PRIORITYITEM** y , **int** p)

Applicazioni

Esempio di utilizzo: Simulatore event-driven

- Ad ogni evento è associato un timestamp di esecuzione
- Ogni evento può generare nuovi eventi, con timestamp arbitrari
- Una coda con min-priorità può essere utilizzata per eseguire gli eventi in ordine di timestamp



Applicazioni nelle prossime lezioni

- Algoritmo di Dijkstra
- Codifica di Huffman
- Algoritmo di Prim per gli alberi di copertura di peso minimo

Implementazioni

Metodo	Lista/vettore non ordinato	Lista Ordinata	Vettore Ordinato	Albero RB
min()	$O(n)$	$O(1)$	$O(1)$	$O(\log n)$
deleteMin()	$O(n)$	$O(1)$	$O(n)$	$O(\log n)$
insert()	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$
decrease()	$O(n)$	$O(n)$	$O(\log n)$	$O(\log n)$

Heap

Una struttura dati speciale che associa

- i vantaggi di un albero (esecuzione in tempo $O(\log n)$), e
- i vantaggi di un vettore (memorizzazione efficiente)

Heap

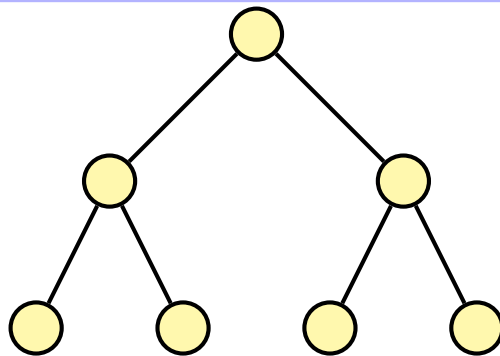
Storia

- Struttura dati inventata da J. Williams nel 1964
- Utilizzata per implementare un nuovo algoritmo di ordinamento: **HeapSort**
- Williams intuì subito che poteva essere usata per altri scopi
- Seguiamo l'approccio storico nel presentare gli heap
 - Prima HeapSort
 - Poi Code con priorità

Alberi binari

Albero binario perfetto

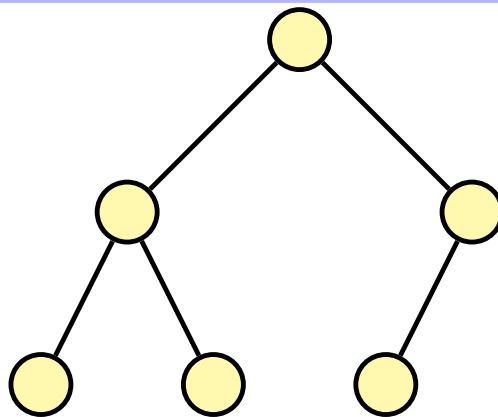
- Tutte le foglie hanno la stessa profondità h
- Nodi interni hanno tutti grado 2
- Dato il numero di nodi n , ha altezza $h = \lfloor \log n \rfloor$
- Dato l'altezza h , ha numeri di nodi $n = 2^{h+1} - 1$



Alberi binari

Albero binario completo

- Tutte le foglie hanno profondità h o $h - 1$
- Tutti i nodi a livello h sono “accatastati” a sinistra
- Tutti i nodi interni hanno grado 2, eccetto al più uno
- Dato il numero di nodi n , ha altezza $h = \lfloor \log n \rfloor$



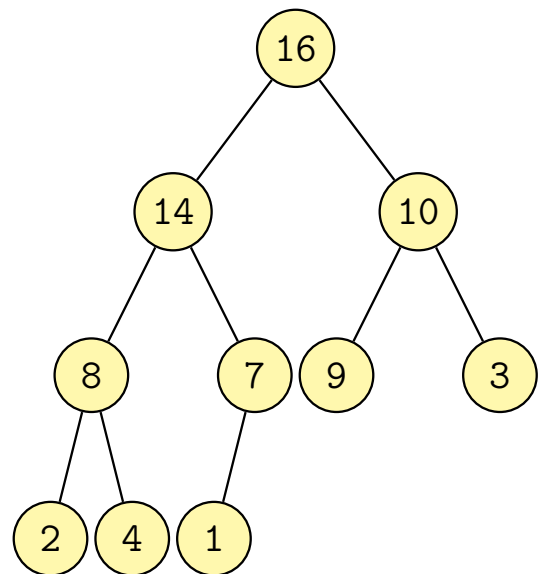
Alberi binari heap

Proprietà heap

Un **albero max-heap** (**min-heap**) è un albero binario completo tale che il valore memorizzato in ogni nodo è **maggiore** (**minore**) dei valori memorizzati nei suoi figli.

Note

Le definizioni e gli algoritmi per alberi max-heap sono simmetrici rispetto agli algoritmi per alberi min-heap



Alberi binari heap

- Un albero heap non impone una relazione di ordinamento totale fra i figli di un nodo
- Un albero heap è un **ordinamento parziale**
 - *Riflessivo*: Ogni nodo è \geq di se stesso
 - *Antisimmetrico*: se $n \geq m$ e $m \geq n$, allora $m = n$
 - *Transitivo*: se $n \geq m$ e $m \geq r$, allora $n \geq r$
- Ordinamenti parziali
 - Nozione più debole di un ordinamento totale...
 - ... ma più semplice da costruire

Alberi binari heap

Vettore heap

Un albero heap può essere rappresentato tramite un **vettore heap**

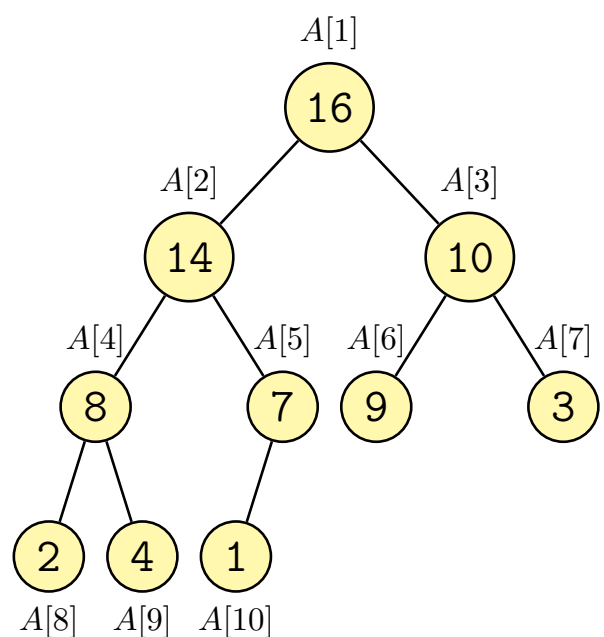
Memorizzazione ($A[1 \dots n]$)

Radice $root() = 1$

Padre nodo i $p(i) = \lfloor i/2 \rfloor$

Figlio sx nodo i $l(i) = 2i$

Figlio dx nodo i $r(i) = 2i + 1$



$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$	$A[10]$
16	14	10	8	7	9	3	2	4	1

Alberi binari heap

Vettore heap

Un albero heap può essere rappresentato tramite un **vettore heap**

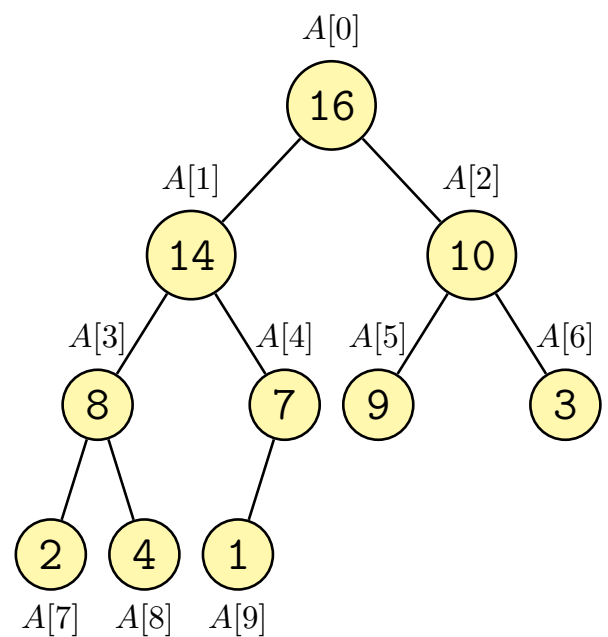
Memorizzazione ($A[0 \dots n - 1]$)

Radice $root() = 0$

Padre nodo i $p(i) = \lfloor (i - 1) / 2 \rfloor$

Figlio sx nodo i $l(i) = 2i + 1$

Figlio dx nodo i $r(i) = 2i + 2$



$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$
16	14	10	8	7	9	3	2	4	1

Alberi binari heap

Proprietà max-heap su vettore

$$A[i] \geq A[l(i)], A[i] \geq A[r(i)]$$

Proprietà min-heap su vettore

$$A[i] \leq A[l(i)], A[i] \leq A[r(i)]$$

HeapSort

Organizzazione heapsort()

Ordina un max-heap "in-place", prima costruendo un max-heap nel vettore e poi spostando l'elemento max in ultima posizione, ripristinando la proprietà max-heap

- **heapBuild()**

Costruisce un max-heap a partire da un vettore non ordinato

- **maxHeapRestore()**

Ripristina la proprietà max-heap

maxHeapRestore()

Input

Un vettore A e un indice i , tale per cui gli alberi binari con radici $l(i)$ e $r(i)$ sono max-heap

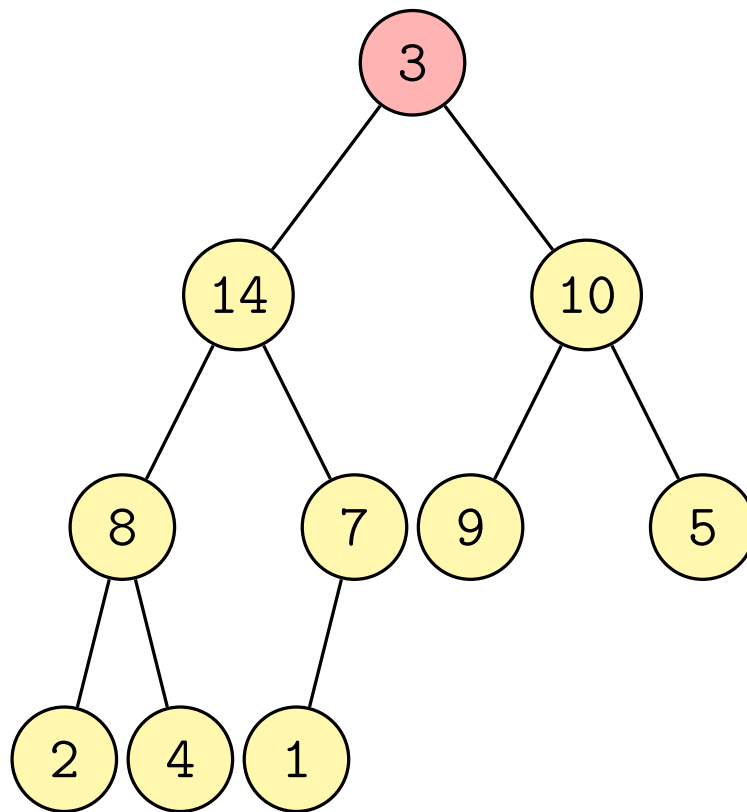
Osservazione

- E' possibile che $A[i]$ sia minore di $A[l(i)]$ o $A[r(i)]$
- In altre parole, non è detto che il sottoalbero con radice i sia un max-heap

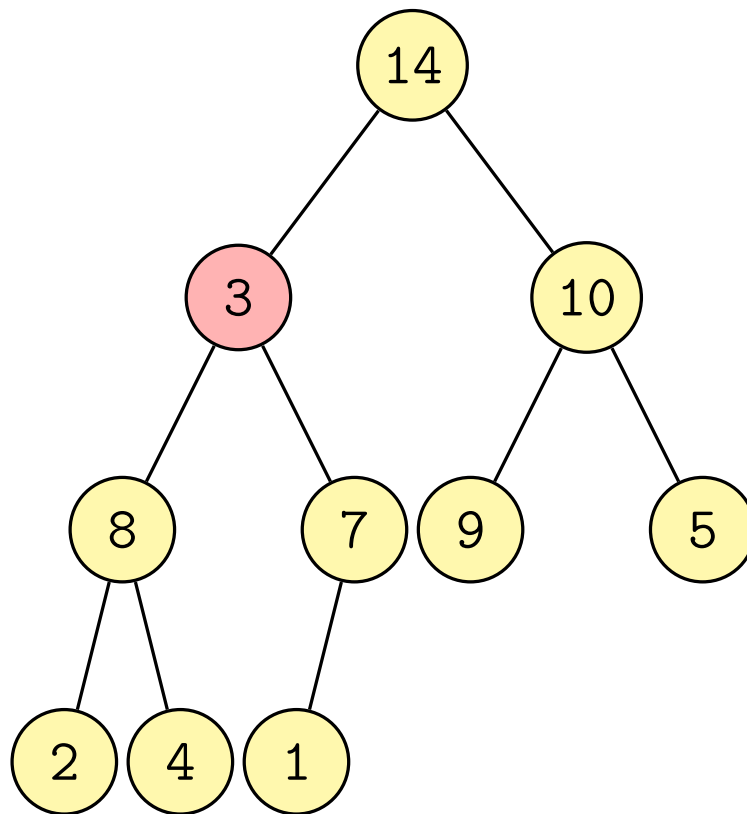
Goal

Modificare in-place il vettore A in modo tale che l'albero binario con radice i sia un max-heap

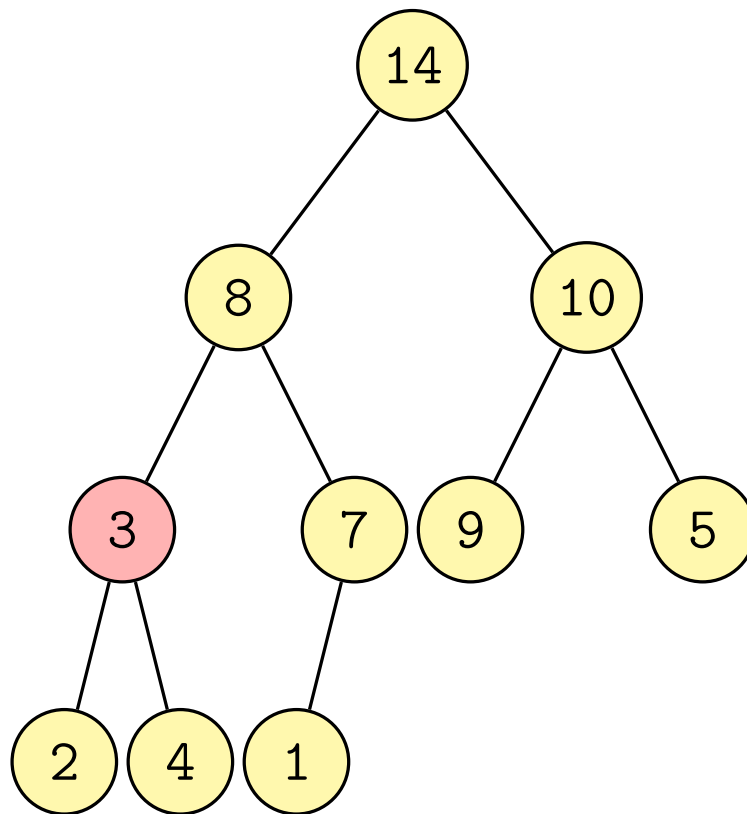
Esempio



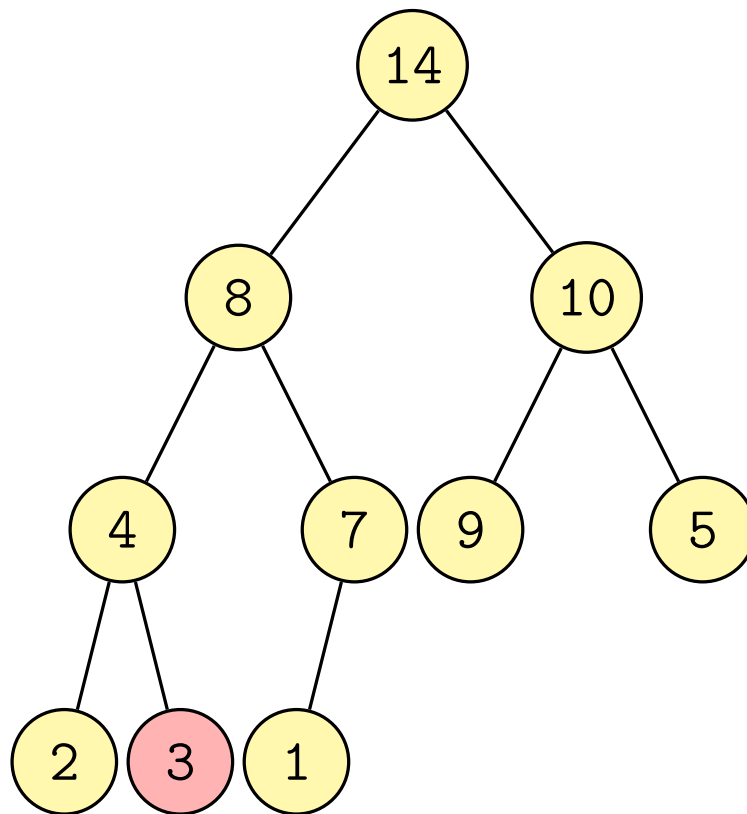
Esempio



Esempio



Esempio



Ripristinare la proprietà max-heap

```
maxHeapRestore(ITEM[] A, int i, int dim)
int max = i
if l(i) ≤ dim and A[l(i)] > A[max] then
    max = l(i)
if r(i) ≤ dim and A[r(i)] > A[max] then
    max = r(i)
if i ≠ max then
    A[i] ↔ A[max]
    maxHeapRestore(A, max, dim)
```

Complessità computazionale

Qual è la complessità computazionale di `maxHeapRestore()`?

- Ad ogni chiamata, vengono eseguiti $O(1)$ confronti
- Se il nodo i non è massimo, si richiama ricorsivamente `maxHeapRestore()` su uno dei figli
- L'esecuzione termina quando si raggiunge una foglia
- L'altezza dell'albero è pari a $\lfloor \log n \rfloor$

Complessità

$$T(n) = O(\log n)$$

heapBuild()

Principio di funzionamento

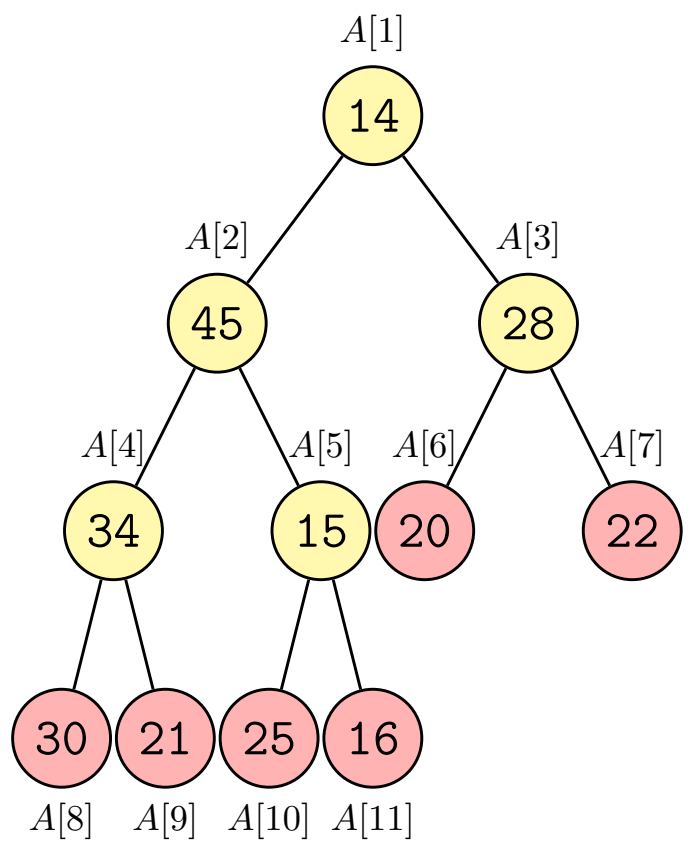
- Sia $A[1 \dots n]$ un vettore da ordinare
- Tutti i nodi $A[\lfloor n/2 \rfloor + 1 \dots n]$ sono foglie dell'albero e quindi heap contenenti *un* elemento
- La procedura `heapBuild()`
 - attraversa i restanti nodi dell'albero, a partire da $\lfloor n/2 \rfloor$ fino ad 1
 - esegue `maxHeapRestore()` su ognuno di essi

```
heapBuild(ITEM[] A, int n)
```

```
for  $i = \lfloor n/2 \rfloor$  downto 1 do  
    maxHeapRestore(A, i, n)
```

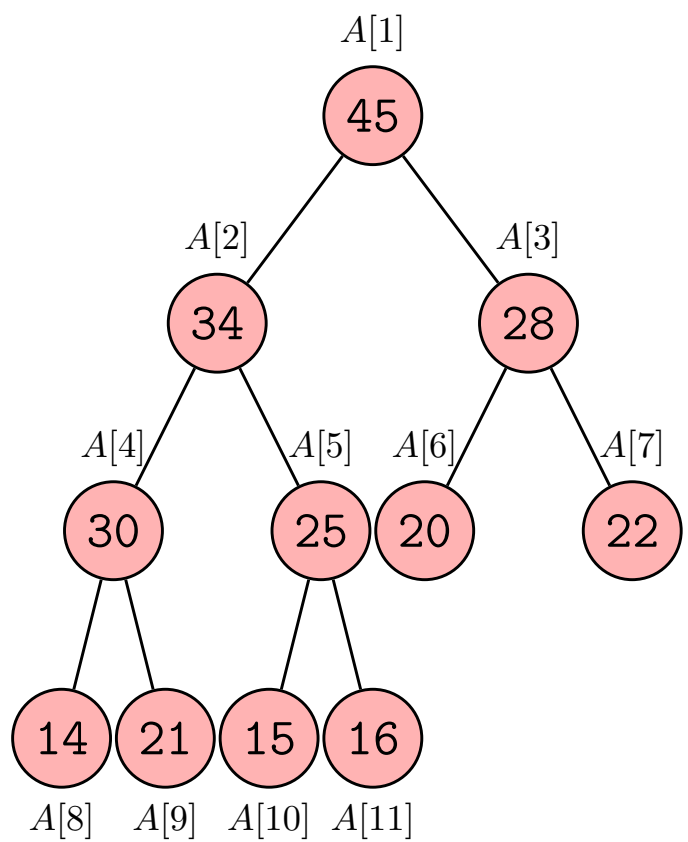
Esempio

- I nodi $A[\lfloor n/2 \rfloor + 1 \dots n]$ sono foglie dell'albero e quindi heap di un elemento
- Per ogni posizione da $\lfloor n/2 \rfloor$ fino ad 1, si esegue `maxHeapRestore()`



Esempio

- I nodi $A[\lfloor n/2 \rfloor + 1 \dots n]$ sono foglie dell'albero e quindi heap di un elemento
- Per ogni posizione da $\lfloor n/2 \rfloor$ fino ad 1, si esegue `maxHeapRestore()`



Complessità

```
heapBuild(ITEM[] A, int n)
```

```
for  $i = \lfloor n/2 \rfloor$  downto 1 do  
    maxHeapRestore(A, i, n)
```

Quant'è la complessità di HEAPBUILD()?

- Limite superiore: $T(n) = O(n \log n)$
- Limite inferiore: $T(n) = \Omega(n \log n)$?

Complessità

Le operazioni `maxHeapRestore()` vengono eseguite un numero decrescente di volte su heap di altezza crescente

Altezza	# Volte
0	$\lfloor n/2 \rfloor$
1	$\lfloor n/4 \rfloor$
2	$\lfloor n/8 \rfloor$
...	...
h	$\lfloor n/2^{h+1} \rfloor$

$$\begin{aligned}
 T(n) &\leq \sum_{h=1}^{\lfloor n \rfloor} \frac{n}{2^{h+1}} h \\
 &= n \sum_{h=1}^{\lfloor n \rfloor} \left(\frac{1}{2}\right)^{h+1} h \\
 &= n/2 \sum_{h=1}^{\lfloor n \rfloor} \left(\frac{1}{2}\right)^h h \\
 &\leq n/2 \sum_{h=1}^{+\infty} \left(\frac{1}{2}\right)^h h = n = O(n)
 \end{aligned}$$

Formula: $\sum_{h=1}^{+\infty} h x^h = \frac{x}{(1-x)^2}, \text{ per } x < 1$

heapSort()

Principio di funzionamento

- L'elemento in prima posizione contiene il massimo
- Viene collocato in fondo
- L'elemento in fondo viene spostato in testa
- Si chiama `maxHeapRestore()` per ripristinare la situazione
- La dimensione dello heap viene progressivamente ridotta (indice i)

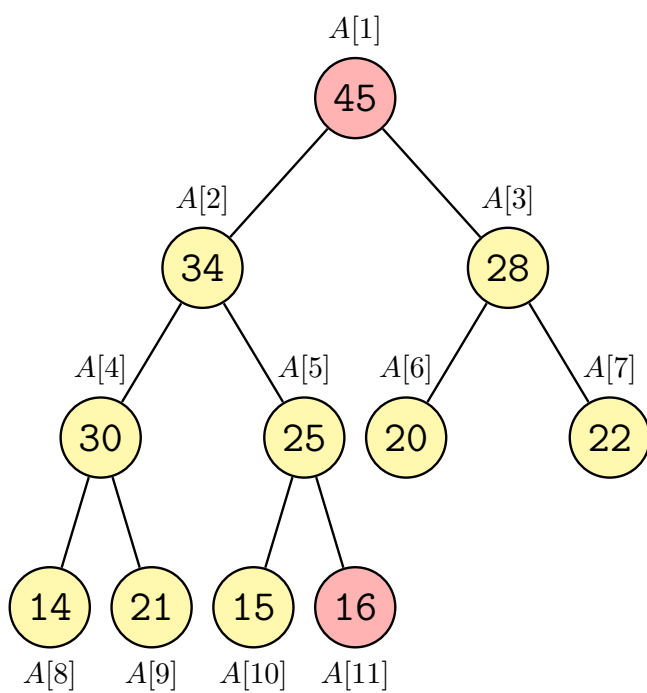
```
HEAPSORT(ITEM[] A, int n)
```

```
heapBuild(A, n)
```

```
for  $i = n$  downto 2 do
```

```
     $A[1] \leftrightarrow A[i]$   
    maxHeapRestore(A, 1,  $i - 1$ )
```

Esempio



A[1] A[2] A[3] A[4] A[5] A[6] A[7] A[8] A[9] A[10] A[11]

45	34	28	30	25	20	22	14	21	15	16
----	----	----	----	----	----	----	----	----	----	----

```
HEAPSORT(ITEM[] A, int n)
```

```
  heapBuild(A, n)
```

```
  for i = n downto 2 do
```

```
    | A[1] ↔ A[i]
```

```
    | maxHeapRestore(A, 1, i - 1)
```

Esempio

A[1]

14

HEAPSORT(ITEM[] A, int n)

heapBuild(A, n)

for $i = n$ downto 2 do

$A[1] \leftrightarrow A[i]$

maxHeapRestore(A, 1, $i - 1$)

A[1] A[2] A[3] A[4] A[5] A[6] A[7] A[8] A[9] A[10] A[11]

14	15	16	20	21	22	25	28	30	34	45
----	----	----	----	----	----	----	----	----	----	----

Complessità

Complessità

- La chiamata `heapBuild()` costa $\Theta(n)$
- La chiamata `maxHeapRestore()` costa $\Theta(\log i)$ in un heap con i elementi
- Viene eseguita con i che varia da 2 a n

$$T(n) = \sum_{i=2}^n \log i + \Theta(n) = \Theta(n \log n)$$

Implementazione code con priorità

Quale versione

Implementiamo una min-priority queue, in quanto negli esempi che vedremo in seguito daremo la precedenza a elementi con priorità minore

Dettagli implementativi

- Vedremo come strutturare un vettore che memorizza coppie $\langle \text{valore}, \text{priorità} \rangle$
- Vedremo come implementare `minHeapRestore()`
- Vedremo come implementare i singoli metodi

Memorizzazione

PRIORITYITEM

int *priority* % Priorità
ITEM *value* % Elemento
int *pos* % Posizione nel vettore heap

swap(PRIORITYITEM[] *H*, **int** *i*, **int** *j*)

$H[i] \leftrightarrow H[j]$

$H[i].pos = i$

$H[j].pos = j$

Inizializzazione

PRIORITYQUEUE

```

int capacity           % Numero massimo di elementi nella coda
int dim                % Numero attuale di elementi nella coda
PRIORITYITEM[] H        % Vettore heap

```

```

PRIORITYQUEUE PriorityQueue(int n)
┌   PRIORITYQUEUE t = new PRIORITYQUEUE
  t.capacity = n
  t.dim = 0
  t.H = new PRIORITYITEM[1...n]
└   return t

```

Inserimento

PRIORITYITEM insert(ITEM x , **int** p)

precondition: $dim < capacity$

$dim = dim + 1$

$H[dim] = \mathbf{new}$ PRIORITYITEM()

$H[dim].value = x$

$H[dim].priority = p$

$H[dim].pos = dim$

int $i = dim$

while $i > 1$ **and** $H[i].priority < H[p(i)].priority$ **do**

$\mathbf{swap}(H, i, p(i))$
 $i = p(i)$

return $H[i]$

minHeapRestore()

```
minHeapRestore(PRIORITYITEM[] A, int i, int dim)


---


int min = i
if  $l(i) \leq dim$  and  $A[l(i)].priority < A[min].priority$  then
    | min =  $l(i)$ 
if  $r(i) \leq dim$  and  $A[r(i)].priority < A[min].priority$  then
    | min =  $r(i)$ 
if  $i \neq min$  then
    | swap( $A, i, min$ )
    | minHeapRestore( $A, min, dim$ )
```

Cancellazione / lettura minimo

ITEM deleteMin()

precondition: $dim > 0$

swap($H, 1, dim$)

$dim = dim - 1$

minHeapRestore($H, 1, dim$)

return $H[dim + 1].value$

ITEM min()

precondition: $dim > 0$

return $H[1].value$

Decremento priorità

decrease(PRIORITYITEM x , **int** p)

precondition: $p < x.priority$

$x.priority = p$

int $i = x.pos$

while $i > 1$ **and** $H[i].priority < H[p(i)].priority$ **do**

$swap(H, i, p(i))$
 $i = p(i)$

Complessità

- Tutte le operazioni che modificano gli heap sistemano la proprietà heap
 - lungo un cammino radice-foglia (`deleteMin()`)
 - oppure lungo un cammino nodo-radice (`insert()`, `decrease()`)
- Poichè l'altezza è $\lfloor \log n \rfloor$, il costo di tali operazioni è $O(\log n)$

Operazione	Costo
<code>insert()</code>	$O(\log n)$
<code>deleteMin()</code>	$O(\log n)$
<code>min()</code>	$\Theta(1)$
<code>decrease()</code>	$O(\log n)$

Insiemi disgiunti – Merge-Find Set

Motivazioni

- In alcune applicazioni siamo interessati a gestire una collezione $S = \{S_1, S_2, \dots, S_k\}$ di **insiemi dinamici disgiunti**
 - $\forall i, j : i \neq j \Rightarrow S_i \cap S_j = \emptyset$
 - $\cup_{i=1}^k S_i = S$, dove $n = |S|$
- Esempio: componenti di un grafo

Operazioni fondamentali

- Creare n insiemi disgiunti, ognuno composto da un unico elemento
- `merge()`: Unire più insiemi
- `find()`: Identificare l'insieme a cui appartiene un elemento

Insiemi disgiunti

Rappresentante

- Ogni insieme è identificato da un **rappresentante** univoco
- Il rappresentante dell'insieme S_i è un qualunque membro di S_i
- Operazioni di ricerca del rappresentante su uno stesso insieme devono restituire sempre lo stesso oggetto
- Solo in caso di unione con altro insieme il rappresentante può cambiare

Memorizzazione

Invece di memorizzare oggetti, utilizziamo gli interi $1 \dots n$ e assumiamo che l'associazione intero-oggetto sia memorizzata esternamente

Specifica

MFSET

% Crea n componenti $\{1\}, \dots, \{n\}$

MFSET Mfset(**int** n)

% Restituisce il rappresentante della componente contenente x

int find(**int** x)

% Unisce le componenti che contengono x e y

merge(**int** x , **int** y)

Esempio

mfset(6)

1

2

3

4

5

6

merge(1,2)

1, 2

3

4

5

6

merge(3,4)

1, 2

3,4

5

6

merge(5,6)

1, 2

3,4

5,6

merge(1,3)

1, 2, 3, 4

5,6

merge(1,5)

1, 2, 3, 4, 5, 6

Applicazione: Componenti connesse dinamiche

Problema

Trovare le componenti connesse di un grafo non orientato **dinamico**

Algoritmo

- Si inizia con componenti connesse costituite da un unico vertice
- Per ogni $(u, v) \in E$, si esegue **merge**(u, v)
- Ogni insieme disgiunto rappresenta una componente connessa

MFSET cc(GRAPH G)

MFSET $M = \text{Mfset}(G.n)$

foreach $u \in G.V()$ **do**

foreach $v \in G.\text{adj}(u)$ **do**
 $M.\text{merge}(u, v)$

return M

Applicazione: Componenti connesse dinamiche

Complessità

$O(n) + m$ operazioni `merge()`

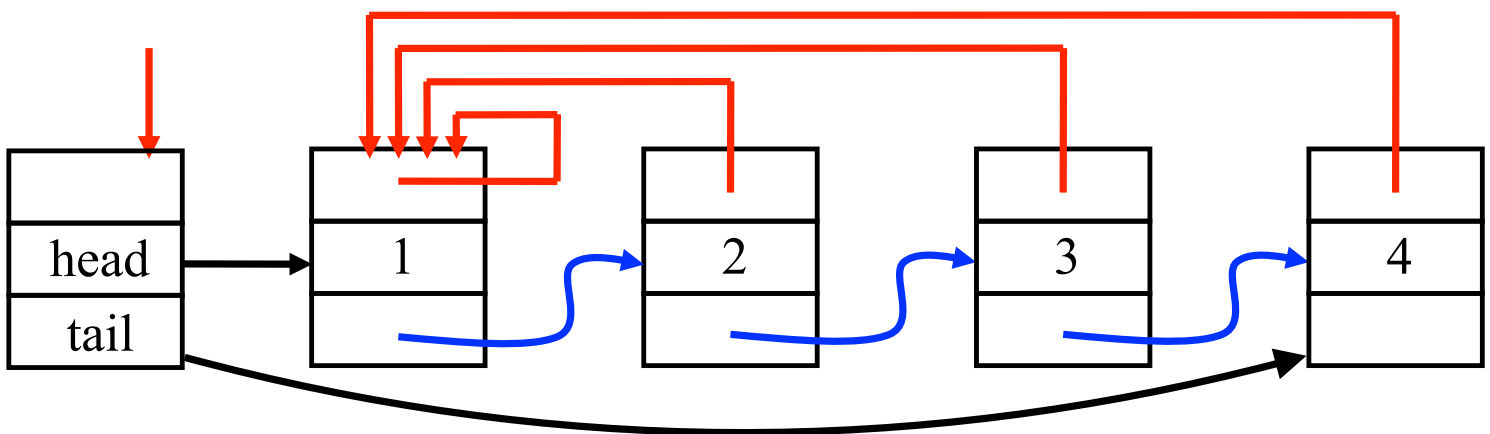
Motivazione

Questo algoritmo è interessante per la capacità di gestire grafi dinamici (in cui gli archi vengono aggiunti)

Realizzazione basata su insiemi di liste

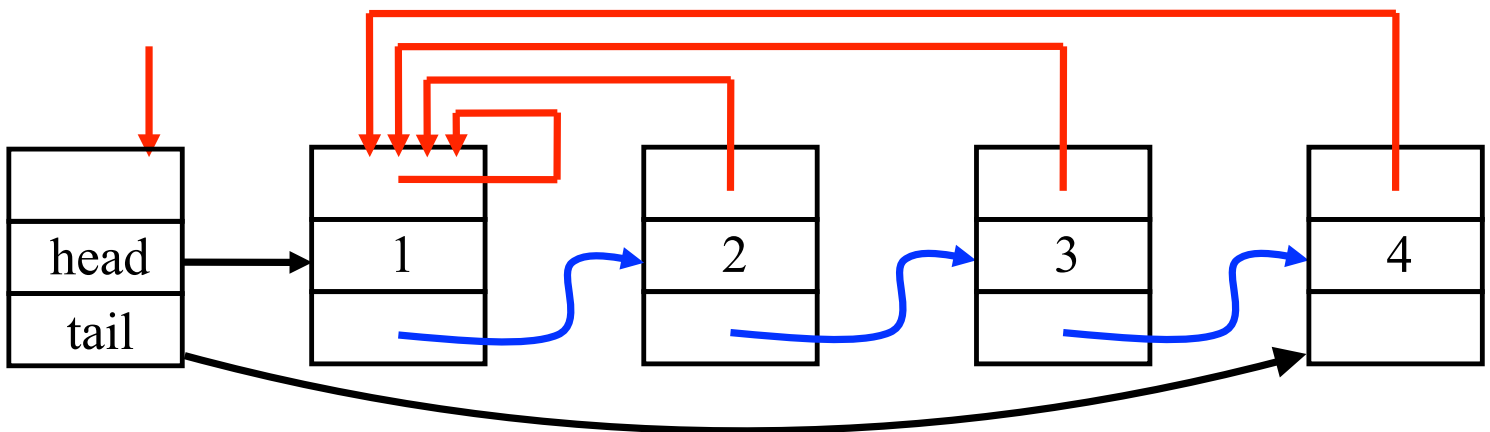
Ogni insieme viene rappresentato da una lista concatenata

- Il primo oggetto di una lista è il rappresentante dell'insieme
- Ogni elemento nella lista contiene:
 - un oggetto
 - un puntatore all'elemento successivo
 - un puntatore al rappresentante



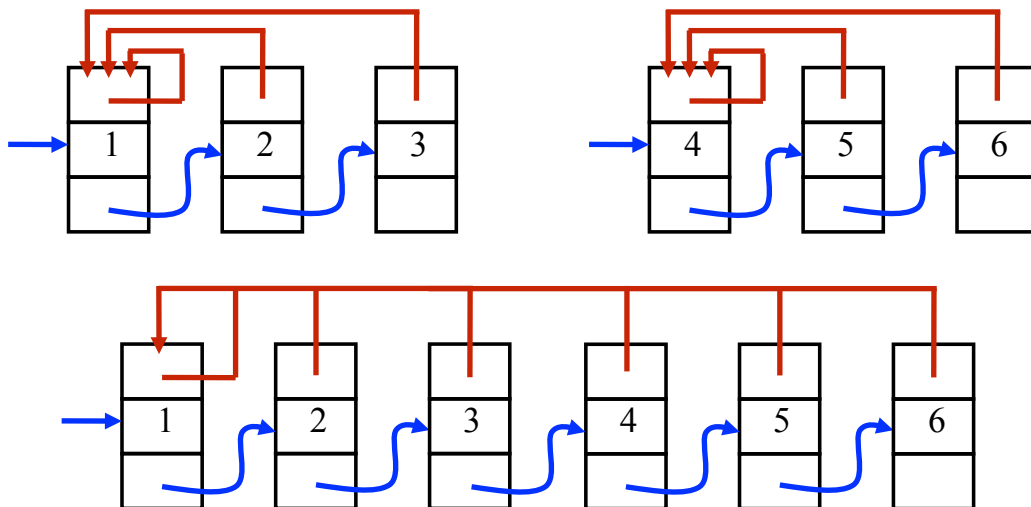
Operazione $\text{find}(x)$

- Si restituisce il rappresentante di x
- L'operazione $\text{find}(x)$ richiede tempo $O(1)$



Operazione $\text{merge}(x, y)$

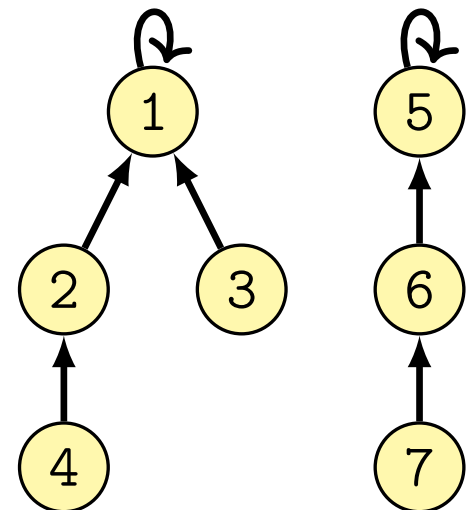
- Si "appende" la lista che contiene y alla lista che contiene x , modificando i puntatori ai rappresentanti nella lista "appesa"
- Costo nel caso pessimo per n operazioni: $O(n^2)$
- Costo ammortizzato: $O(n)$



Realizzazione basata su insieme di alberi (foresta)

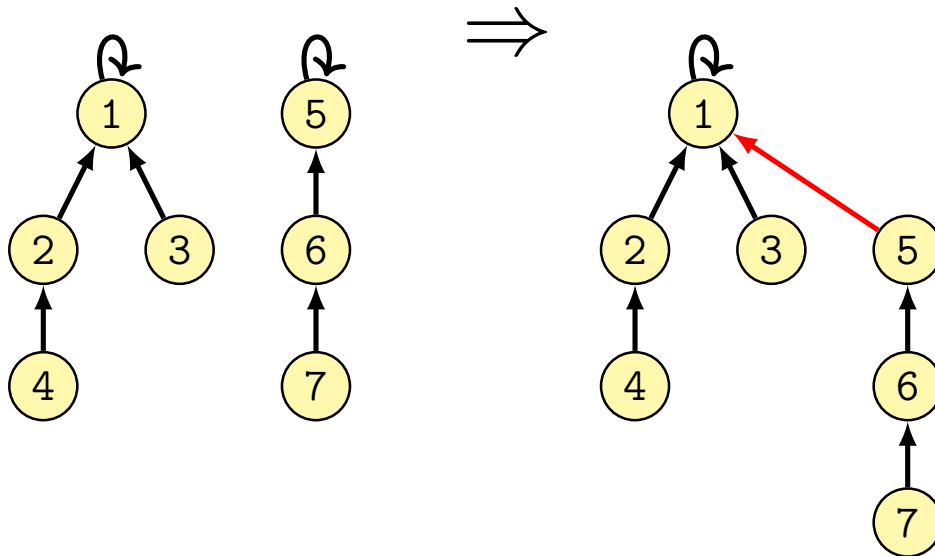
Ogni insieme viene rappresentato da un albero

- Ogni nodo dell'albero contiene:
 - un oggetto
 - un puntatore al padre
- La radice è il rappresentante dell'insieme
- La radice ha un puntatore a se stessa



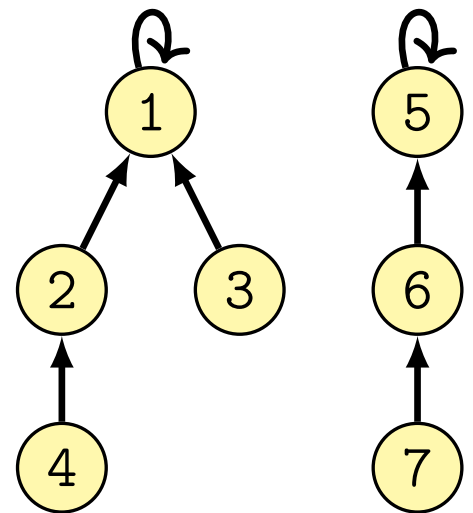
Operazione $\text{merge}(x, y)$

- Si aggancia l'albero radicato in y ad x
- Modificando il puntatore al padre di y
- Costo: $O(1)$



Operazione $\text{find}(x)$

- Risale la lista dei padri di x fino a trovare la radice e restituisce la radice come rappresentante
- Costo: $O(n)$ nel caso pessimo (perché?)



Tecniche euristiche

Algoritmo euristico

È un particolare tipo di algoritmo progettato per

- risolvere un problema più velocemente, qualora i metodi classici siano troppo lenti
- trovare una soluzione approssimata, qualora i metodi classici falliscano nel trovare una soluzione esatta

Euristiche applicate agli insiemi disgiunti

- Euristiche del **peso** (Liste)
- Euristiche del **rango** (Alberi)
- Euristiche della **compressione dei cammini** (Alberi)

Liste: Euristica sul peso

Strategia per diminuire il costo dell'operazione `merge()`

- Memorizzare nelle liste l'informazione sulla loro lunghezza
- Agganciare la lista più corta a quella più lunga
- La lunghezza della lista può essere mantenuta in tempo $O(1)$

Complessità

- Tramite analisi ammortizzata, è possibile vedere che il costo di $n - 1$ operazioni `merge` è $O(n \log n)$
- Quindi il costo ammortizzato delle singole operazioni è $O(\log n)$

Alberi: Euristiche sul rango

Strategia per diminuire il costo dell'operazione `find()`

- Ogni nodo mantiene informazioni sul proprio rango
- Il rango $rank[x]$ di un nodo x è il numero di archi del cammino più lungo fra x e una foglia sua discendente
- Rango \equiv altezza del sottoalbero associato al nodo
- Obiettivo: mantenere bassa l'altezza degli alberi

Alberi: Euristica sul rango

Alberi di rango uguale

- Si aggancia un albero alla radice dell'altro (indifferente)
- L'altezza cresce di 1



Alberi di rango diverso

- Si aggancia l'albero con rango più basso all'albero con rango più alto
- L'altezza resta inalterata



Complessità

Teorema

Un albero MFSET con radice r ottenuto tramite euristica sul rango ha almeno $2^{\text{rank}[r]}$ nodi.

Dimostrazione (informale)

- **Caso base:** All'inizio tutti gli alberi sono costituiti da un nodo singolo, con rank 0: quindi ogni albero ha almeno $2^0 = 1$ nodi, corretto.

Complessità

Teorema

Un albero MFSET con radice r ottenuto tramite euristica sul rango ha almeno $2^{\text{rank}[r]}$ nodi.

Dimostrazione (informale)

- **Caso 1:** Facendo merge di due alberi con radice x, y con rango $\text{rank}[x] > \text{rank}[y]$, il rango finale è pari a $\text{rank}[x]$, mentre il numero di nodi è sicuramente maggiore di $2^{\text{rank}[x]}$.

Complessità

Teorema

Un albero MFSET con radice r ottenuto tramite euristica sul rango ha almeno $2^{\text{rank}[r]}$ nodi.

Dimostrazione (informale)

- **Caso 2** :Facendo merge di due alberi con radice x, y con rango $\text{rank}[x] = \text{rank}[y]$, il rango finale è pari a $\text{rank}[x] + 1$ e ha almeno

$$2^{\text{rank}[x]} + 2^{\text{rank}[y]} = 2^{\text{rank}[x]} + 2^{\text{rank}[x]} = 2^{\text{rank}[x]+1}$$

come volevasi dimostrare

Complessità

Corollario

Un albero MFSET con radice r ed n nodi ha altezza inferiore a $\log n$.

Dimostrazione

$$n \geq 2^{\text{rank}[r]} \Leftrightarrow \text{rank}[r] \leq \log n$$

Complessità

L'operazione $\text{find}(x)$ ha costo $O(\log n)$

Algoritmo

MFSET

int[] *parent***int**[] *rank*MFSET Mfset(**int** *n*) MFSET *t* = **new** MFSET *t.parent* = **int**[1...*n*] *t.rank* = **int**[1...*n*] **for** *i* = 1 **to** *n* **do** | *t.parent*[*i*] = *i* | *t.rank*[*i*] = 0 **return** *t*

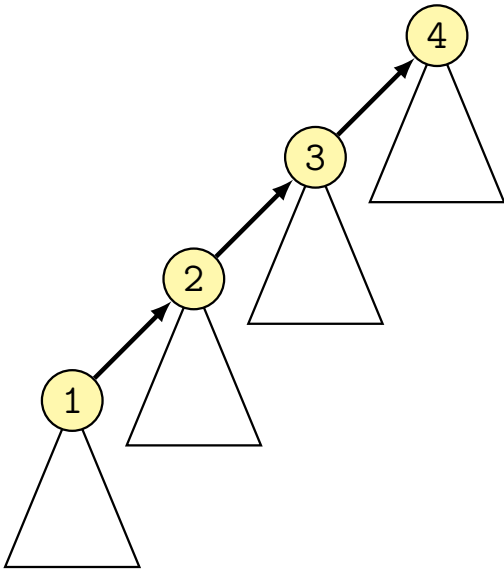
merge(**int** *x*, **int** *y*)

r_x = find(*x*)*r_y* = find(*y*)**if** *r_x* ≠ *r_y* **then** **if** *rank*[*r_x*] > *rank*[*r_y*] **then** | *parent*[*r_y*] = *r_x* **else if** *rank*[*r_y*] > *rank*[*r_x*] **then** | *parent*[*r_x*] = *r_y* **else** | *parent*[*r_x*] = *r_y* | *rank*[*r_y*] = *rank*[*r_y*] + 1

Alberi: Euristiche di compressione dei cammini

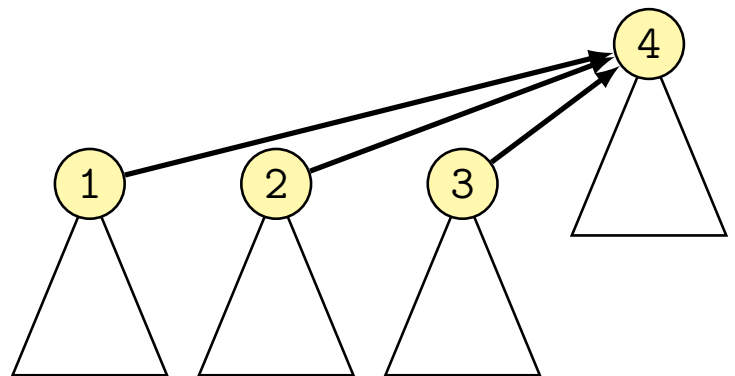
Operazione $\text{find}(x)$

L'albero viene "appiattito" in modo che ricerche successive di x siano svolte in $O(1)$



```
int find(int x)
```

```
if parent[x]  $\neq$  x then  
    parent[x] = find(parent[x])  
return parent[x]
```



Alberi: Euristiche sul rango + compressione cammini

Applicando entrambe le euristiche

- Il rango non è più l'altezza del nodo, ma il **limite superiore** all'altezza del nodo
- **Non** viene calcolato il rango corretto
 - Troppo difficile mantenere le informazioni di rango corretto
 - In ogni caso, non è necessario

Complessità

- Costo ammortizzato di m operazioni merge-find in un insieme di n elementi è $O(m \cdot \alpha(n))$
- La **funzione inversa di Ackermann** $\alpha(n)$ cresce lentamente
- Esempi: per $n \leq 2^{65536}$, $\alpha(n) \leq 5$

Complessità – Riassunto

Algoritmo	find()	merge()
Liste	$O(1)$	$O(n)$
Alberi	$O(n)$	$O(1)^+$
Liste + Euristiche sul peso	$O(1)$	$O(\log n)^*$
Alberi + Euristiche sul rango	$O(\log n)$	$O(1)^+$
Alberi + Euristiche sul rango + Compressione cammini	$O(1)^*$	$O(1)$

* Complessità ammortizzata

+ Si considera solo l'operazione di unione, non si considera l'identificazione dei rappresentanti tramite find()