Algoritmi e strutture dati

Tipi di dati

Tipo di una variabile

Attributo che specifica l'insieme di valori che la variabile può assumere e le relative operazioni.

- L'uso dei tipi varia tra i linguaggi di programmazione: in alcuni linguaggi, come il C, c'è la massima libertà sui tipi mentre in altri, come in Java e nei linguaggi ad oggetti, è molto più rigido.
- Se si prende una classe Java la documentazione fornisce: quali siano i dati che vengono rappresentati dagli oggetti e i metodi, ossia le operazioni, che si possono fare.

Si definisce solo il **cosa** si vuole rappresentare e il cosa bisogna essere in grado di fare, e non il come questi vengano implementati.

Tipo dizionario
Collezione di elementi ciascuno dei quali è caraterizzato da una chiave.

Le chiavi appartengono a un dominio *totalmente* ordinato, ossia è possibile fare delle operazioni di confronto:
$=, \neq, <, >, \leq, \geq$

Operazioni tipiche dei dizionari sono l'inserimento, la ricerca e la cancellazione

Strutture dati

Specifica organizzazione delle informazioni che permette di realizzare e implementare un determinato tipo di dati.

Uno stesso tipo è possibile rappresentarlo in modi diversi (si passa perciò dal cosa al **come** si vuole rappresentare un qualcosa) e si possono quindi avere varie strutture dati: $\text{stesso tipo} \neq \text{strutture dati}$

Dizionario
Il tipo dizionario può essere quindi implementato in maniere differenti:

1. Array ordinato in base alla chiave
   Ricerca (binaria): $\Theta(\log n)$; Inserimento: $\Theta(n)$
2. Array non ordinato
   Ricerca: $\Theta(n)$; Inserimento: $\Theta(1)$

Collezioni

Strutture indicizzate

Sono le strutture che di solito prendono il nome di array

- Sono allocate in una *porzione contigua* di memoria
- L'accesso si effettua mediante l'*indice* (ossia la posizione)
- Il tempo di accesso è *indipendente* dalla posizione del dato

Sono delle **strutture statiche**, e presentano perciò una limitazione: non è possibile aggiungere nuove posizioni.

Strutture collegate

Si basano su sistemi di collegamenti

- Non è necessario allocare l'intera struttura in una porzione contigua di memoria
- Gli elementi sono collegati tra loro
- Il passaggio da un elemento ad altri avviene tramite questi collegamenti
- Ci sono varie tipologie di collegamenti

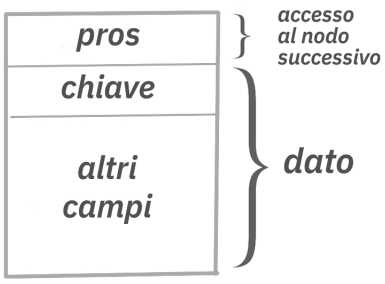
Sono delle **strutture dinamiche** perché evolvono dinamicamente man mano che si inseriscono e si cancellano elementi. Non è però possibile accedere direttamente agli elementi di queste strutture dati, ma bisogna ricorrere ad una certa strategia che dipende dal particolare tipo di struttura che si prende in considerazione.

Tra le strutture collegate troviamo le liste concatenate.

Liste concatenate: liste lineari

Insieme ordinato (inteso come uno in fila all'altro) di nodi collegati linearmente uno dopo l'altro. *Ad ogni nodo è possibile associare un unico successore*. Ogni nodo contiene:

- un dato della collezione (in genere formato da un certo insieme di campi, uno dei quali funge da campo chiave)
- l'informazione per accedere al nodo successivo



Si accede ai nodi tramite **riferimenti** (che vengono poi implementati mediante puntatori). Il riferimento è un'informazione che ci dice dove si trovi il nodo.

Notazione:

- `p` è il riferimento al nodo
- `p.chiave` è il valore della chiave
- `p.chiave <- p.chiave + 1` per aggiornare il valore della chiave
- `p.pros` è il riferimento al nodo sucessivo
- `null` è il riferimento nullo

Operazioni

Tutte le operazioni sottostanti richiedono un tempo di esecuzione nel caso peggiore pari a $\mathcal{O}(n)$, dove n è il numero di elementi presenti all'interno della lista lineare.

Ricerca elemento in base alla posizione

Si percorre la lista partendo dalla testa fino a raggiungere la posizione desiderata. Il costo è $\Theta(n)$.

```
FUNZIONE elemento(Lista L, intero i) -> Nodo
    p <- L
    WHILE p != null AND i > 0 DO
    |   p <- p.pros
    |   i <- i - 1
    RETURN p
```

Sia la `Lista` che il `Nodo` sono dei riferimenti, e quindi in realtà sono la stessa cosa. La differenza sta nel fatto che `Lista` sia un riferimento al primo nodo (testa), mentre `Nodo` è un riferimento a un nodo generico.

Ricerca elemento in base alla chiave

Si scorre la lista confrontando la chiave in ogni nodo fino a trovare quella cercata. Nel caso peggiore, se la chiave è presente nell'ultimo nodo o non è affatto presente, si eseguono $\Theta(n)$ passi.

```
FUNZIONE trova(Lista L, tipoChiave k) -> Nodo
    p <- L
    WHILE p != null AND p.chiave != k DO
    |   p <- p.pros
    RETURN p
```

Ricerca elemento in base alla chiave in una lista ordinata

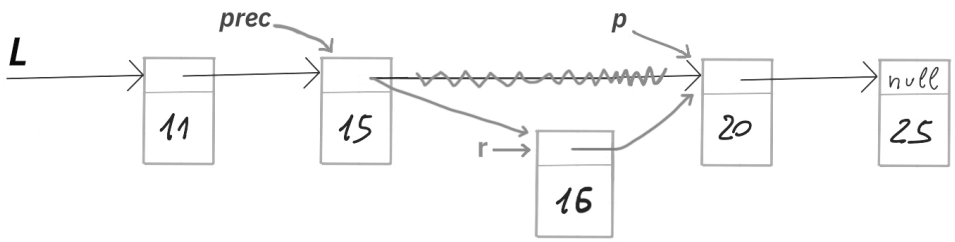
Anche qui si scorre la lista, ma si può interrompere il ciclo non appena si incontra un nodo con chiave maggiore di quella cercata. Il caso peggiore non è migliorato, rimane nell'ordine di $\Theta(n)$. Il caso medio corrisponde a circa $\frac{1}{2}n$.

```
FUNZIONE trova(ListaOrdinata L, tipoChiave K) -> Nodo
    p <- L
    WHILE p != null AND p.chiave < k DO
    |   p <- p.pros
    IF p = null OR p.chiave > k THEN
    |   RETURN null
    ELSE
    |   RETURN p
```

Inserimento in una lista ordinata

Si ricerca la posizione corretta proprio come nella ricerca per chiave in lista ordinata, che costa quindi $\Theta(n)$ nel caso peggiore. Una volta trovata la posizione, l'inserimento vero e proprio avviene in tempo $\mathcal{O}(1)$ in quanto si aggiornano solo dei puntatori.

```
FUNZIONE inserisci(ListaOrdinata L, elemento d) -> ListaOrdinata
    k <- d.chiave
    p <- L
    prec <- null
    WHILE p != null AND p.chiave < k DO
    |   prec <- p
    |   p <- p.pros
    |
    r <- riferimento a un nuovo nodo
    r.chiave <- k
    r.altri_campi <- d.altri_campi
    r.pros <- p
    |
    IF prec = null THEN     // inserimento in testa
    |   L <- r
    ELSE prec.pros <- r
    RETURN L
```

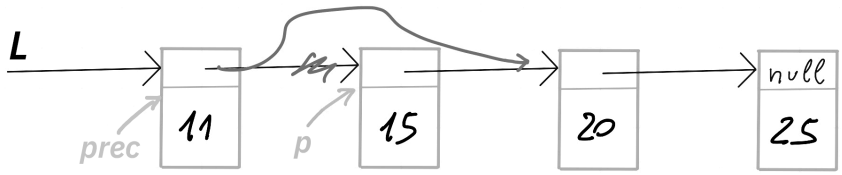


Inserimento in testa: è un caso limite che si verifica quando il nuovo nodo va inserito all'inizio della lista. Questo capita quando la lista è vuota o quando la chiave del nuovo nodo è più piccola di qualsiasi altro nodo già presente.
Inserimento in coda: si verifica quando il nuovo nodo va inserito alla fine della lista. Questo capita quando la lista è vuota (e quindi testa e coda coincidono) oppure quando la chiave del nuovo nodo è più grande di qualsiasi altro nodo già presente.

Cancellazione da lista ordinata

Si cerca il nodo da eliminare similmente alla ricerca per chiave, con un costo di $\Theta(n)$ nel peggiore dei casi. La rimozione vera e propria, che consiste nel modificare dei puntatori, è un'operazione a tempo costante.

```
FUNZIONE cancella(ListaOrdinata L, elemento d) -> ListaOrdinata
    k <- d.chiave
    p <- L
    prec <- null
    WHILE p != null AND p.chiave < k DO
    |   prec <- p
    |   p <- p.pros
    IF p = null OR p.chiave > k THEN
    |   RETURN L
    IF prec = null THEN     // cancellazione in testa (all'inizio)
    |   L <- p.pros
    ELSE                    // cancellazione interna
    |   prec.pros <- p.pros
    RETURN L
```



Cancellazione in testa: se il nodo da cancellare è proprio il primo della lista, non esiste un nodo precedente. In quel caso basta far puntare la testa `L` (il riferimento al primo nodo) al secondo nodo, cioè a `p.pros`. Così il vecchio primo nodo non è più raggiungibile (viene di fatto rimosso).
Cancellazione interna: se il nodo da eliminare è in mezzo (o alla fine) della lista, la testa rimane invariata. Bisogna però aggiornare il puntatore del **nodo precedente** `prec` in modo che salti il nodo `p` e punti direttamente al successivo di `p`, ossia `p.pros`.

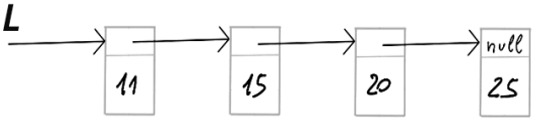
A seconda del linguaggio di programmazione utilizzato ci potrebbe essere bisogno di rilasciare la memoria se questa non è gestita da un garbage collector.

Implementazione

Tramite array

Si usa un array: ogni cella rappresenta un nodo, che contiene i suoi dati e l'indice (ossia l'indirizzo logico) del nodo successivo. Se l'indice è -1 significa che non ci sono nodi successivi (fine della lista). La lista è identificata dall'indice del primo nodo: per attraversarla si parte da quel valore e, leggendo di volta in volta l'indice memorizzato nel nodo, ci si sposta al successivo finché non si incontra -1.

Questa soluzione ha il vantaggio di concentrare tutti i dati in un'unica area di memoria. D'altro canto, la dimensione dell'array deve essere fissata o aggiornata dinamicamente.



Tramite puntatori

Nell'implementazione tramite puntatori, invece, ogni nodo è un'unità autonoma in memoria, dotata di un puntatore che rimanda fisicamente al nodo successivo; la lista è identificata dal puntatore che indica la testa (il primo nodo), e l'ultimo nodo ha `null` come successore. Questo approccio è più flessibile, ma in linguaggi come `C` serve gestire manualmente l'allocazione e deallocazione della memoria.

```c
// esempio in C
struct node {
    int chiave;
    // altri campi
    struct node *pros;
}

struct node *l;
```

Tipo Pila (o Stack)

Il tipo pila è una collezione di dati con organizzazione *Last-In-First-Out* (LIFO): l'ultimo elemento che viene inserito è il primo che viene rimosso. *È un tipo di dato, non una struttura.*

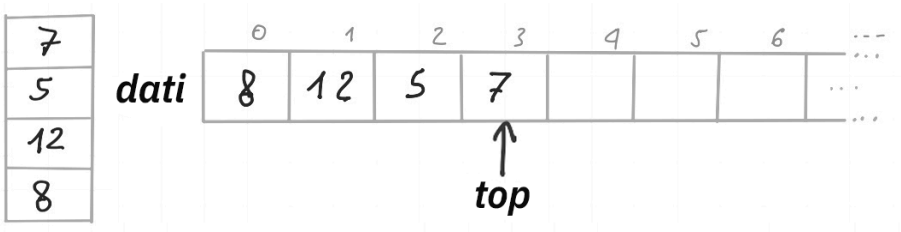
Lo stack della ricorsione non è altro che una pila di record di attivazione che segue il principio LIFO.

Operazioni:

- `isEmpty() -> boolean`
- `push(elemento)`
- `pop() -> elemento`
- `top() -> elemento`

Sono possibili due diverse implementazioni: mediante array e mediante liste lineari.

Implementazione mediante array



Si ha un array `dati` grande in cui si memorizzano gli elementi della pila e un indice che ci dice fin dove questo array è riempito: in sostanza indica la cima della pila.

Se gli indici dell'array vanno da 0 ad n, l'elemento di posto 0 rappresenta l'elemento più in basso della pila, e l'indice `top` indica l'ultima posizione occupata della pila, ed è quindi minore di n. Se la pila è vuota, questo indice `top` vale -1.

Operazioni

Tutte le operazioni sottostanti richiedono un tempo di esecuzione costante, ossia pari a $\mathcal{O}(1)$.

```
FUNZIONE isEmpty() -> boolean
    IF top = -1 THEN RETURN true
                ELSE RETURN false
```

```
PROCEDURA push(elemento x)
    top <- top + 1
    dati[top] <- x
```

```
FUNZIONE top() -> elemento
    RETURN dati[top]
```

```
FUNZIONE pop() -> elemento
    x <- dati[top]
    top <- top - 1
    RETURN x

FUNZIONE pop() -> elemento
    RETURN dati[top--] // incremento post-fisso
```

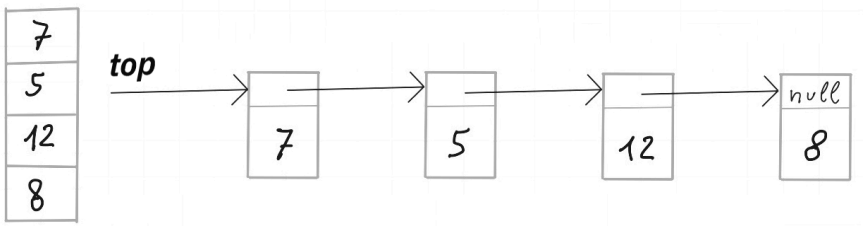
Nella `pop` l'elemento può anche essere lasciato lì, l'importante è rimuoverlo logicamente decrementando `top`

Problematiche

1. Se l'indice `top` contiene -1 le funzioni `top` e `pop` generano un errore in esecuzione. Non è però un errore rispetto al tipo pila in quanto queste operazioni sono definite su una pila non vuota, quindi è corretto che queste generino un errore se la pila è vuota *($\rightarrow$ nei linguaggi che le supportano si sollevano delle eccezioni)*. È un comportamento che fa parte del tipo pila.
2. Una `push` su un array pieno genererebbe un errore. Questo però non è un comportamento intrinseco del tipo pila, in quanto a livello teorico questo tipo di dato non presenta nessun tipo di limitazione per quanto riguarda la sua grandezza: ci si potrebbe procurare dell'altra memoria, continuando quindi a impilare gli elementi l'uno sopra l'altro. Questo **non** è un problema del tipo pila, ma è un limite di questa implementazione.

Inoltre il vantaggio degli array, ossia quello di accedere istantaneamente agli elementi, è utilizzato solamente per accedere all'elemento `top`: non è necessario accedere agli elementi sottostanti $\rightarrow$ l'array non è sfruttato a pieno.

Implementazione mediante liste lineari



Siccome nella pila ci interessa accedere sempre all'elemento che sta sopra, si rappresenta la pila mediante una lista il cui il primo elemento rappresenta proprio la cima, e andando avanti nella lista troviamo gli elementi sottostanti della pila. A noi non interessa accedere a questi altri elementi, e quindi questa risulta un'implementazione valida.

- Il riferimento iniziale della lista `top` è quindi proprio il primo elemento della pila

La pila vuota si rappresenta assegnando al riferimento `top` il valore `null`

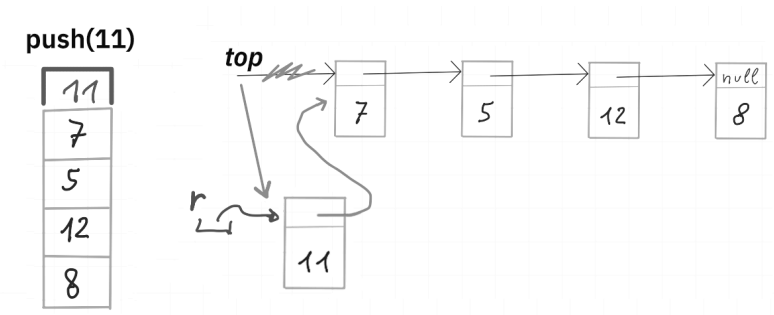
Operazioni

isEmpty

```
FUNZIONE isEmpty() -> boolean
    IF top = null THEN RETURN true
                  ELSE RETURN false
```

push

```
PROCEDURA push(elemento x)
    r <- riferimento a un nuovo nodo
    r.dato <- x
    r.pros <- top
    top <- r
```



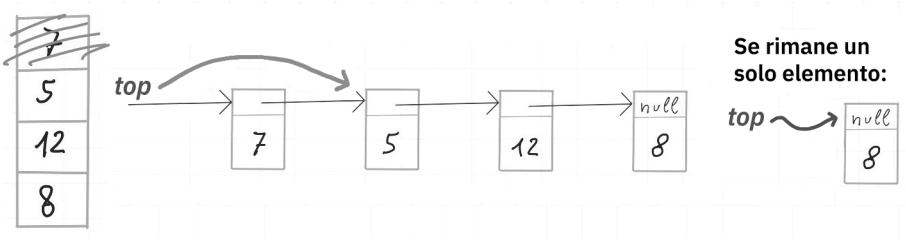
Con questa `push` la struttura evolve dinamicamente: si occupa lo spazio nel momento in cui si aggiunge un elemento. Perciò, a differenza dell'implementazione precedente, non c'è nessun limite al numero di elementi che si possono inserire nella pila.

top

```
FUNZIONE top() -> elemento
    RETURN top.dato
```

pop

```
FUNZIONE pop() -> elemento
    x <- top.dato
    top <- top.pros
    RETURN x
```



Se la `pop` viene implementata in linguaggi senza un garbage collector come il C, occore anche rilasciare la memoria.

Problematica

Se `top` punta a `null`, e di conseguenza la pila è vuota, sia le funzioni `top` che `pop` generano un errore. Sono però degli errori corretti che sono intrinsechi al tipo pila.

Può convenire utilizzare l'implementazione array in due casi: per risparmiare memoria in quanto non bisogna salvarsi oltre al dato anche un puntatore, e quando si conosce la grandezza massima della pila.

Tipo Coda

Il tipo coda è una collezione di dati con organizzazione *First-In-First-Out* (FIFO): il primo elemento che viene inserito è il primo che viene rimosso. *È un tipo di dato, non una struttura.*

Operazioni

- `isEmpty() -> boolean`
- `enqueue(elemento)` *mettere in coda*
- `dequeue() -> elemento` *prelevare*
- `first() -> elemento`

Sono possibili due diverse implementazioni: mediante array e mediante liste lineari.

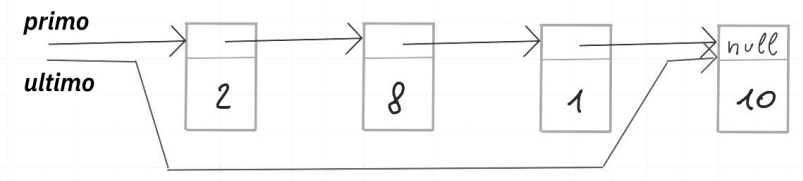
L'implementazione tramite array non è però molto valida in quanto, quando si preleva un elemento dalla coda, tutti gli elementi successivi andrebbero spostati: se si hanno n elementi in coda, prelevare un elemento costerebbe n.
Si potrebbe allora utilizzare un riferimento per il primo e per l'ultimo elemento della coda nell'array: quando si preleva si sposta il riferimento alla testa all'elemento successivo, e quando si aggiunge un elemento si sposta il riferimento alla coda sempre all'elemento successivo. Questo evita di spostare tutti gli elementi della coda. Quando il riferimento alla coda è immediatamente a destra rispetto a quello della testa, allora la coda è vuota.

Sorgono lo stesso due problemi:

1. siccome stiamo utilizzando un array, questo ha una capacità limitata $\rightarrow$ stesso problema riscontrato nelle pile
2. quando si fanno dei prelievi, la parte di array prima del riferimento alla testa non viene più utilizzata.

Per risolvere il secondo problema la coda viene implementata mediante un array circolare: quando il riferimento alla coda arriva all'ultima posizione dell'array, questo viene fatto ripartire dalla prima posizione.

Implementazione mediante liste lineari



Si utilizza una lista lineare che sfrutta un puntatore ausiliario: abbiamo il puntatore classico che contiene il riferimento del primo elemento della coda, mentre quello aggiuntivo punta all'ultimo elemento della coda. Quest'ultimo è necessario cosicché, quando si vuole aggiungere un elemento, si ha già pronto la posizione in cui andare ad "attaccare" il nuovo elemento.

- Senza questo puntatore bisognerebbe scorrere tutta la lista per trovarne la posizione dell'ultimo elemento. Questo approccio rende le operazioni di dequeue ed enqueue a tempo costante.
- *Il puntatore primo punta alla testa, mentre ultimo alla coda*

La coda vuota viene rappresentata con entrambi i puntatori a `null`.

Operazioni

Tutte le operazioni sottostanti richiedono un tempo di esecuzione costante, ossia pari a $\mathcal{O}(1)$.

isEmpty

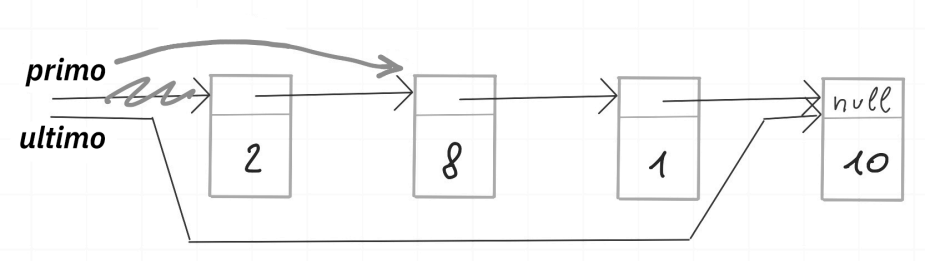
```
FUNZIONE isEmpty() -> boolean
    IF primo = null THEN RETURN true
                    ELSE RETURN false
```

first

```
FUNZIONE first() -> elemento
    RETURN primo.dato
```

dequeue

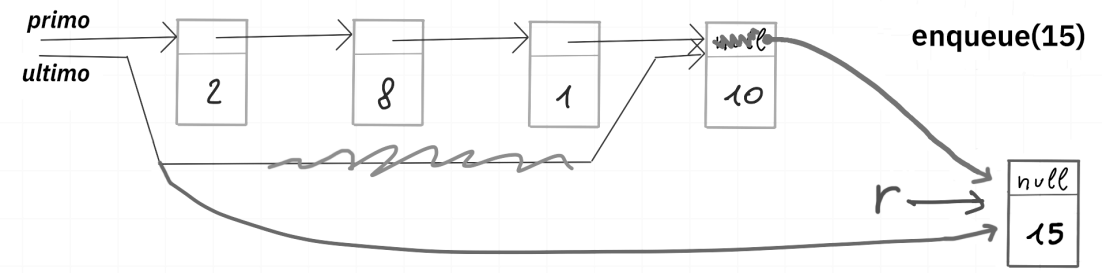
```
FUNZIONE dequeue() -> elemento
    x <- primo.dato
    primo <- primo.pros
    IF primo = null THEN
    |   ultimo <- null
    RETURN x
```



Facendo `dequeue()` potrebbe capitare che la coda si svuoti. Ciò avviene quando la coda contiene un elemento solo ed entrambi i puntatori contengono lo stesso riferimento allo stesso nodo. Con `dequeue()` entrambi questi puntatori conterranno poi `null`.

enqueue

```
PROCEDURA enqueue(elemento x)
    r <- riferimento a un nuovo nodo
    r.dato <- x
    r.pros <- null
    IF primo = null THEN    //caso di cosa vuota
    |   primo <- r
    |   ultimo <- r
    ELSE
    |   ulltimo.pros <- r
    |   ultimo <- r
```



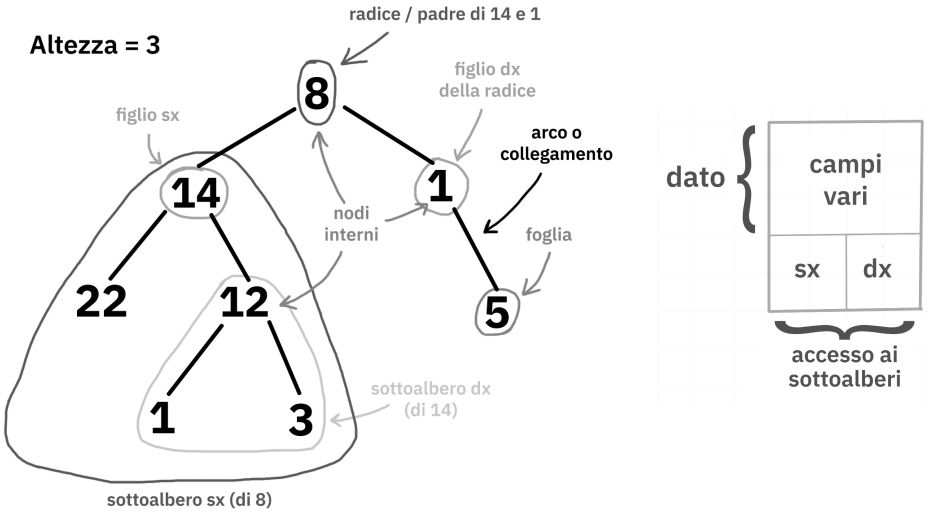
Se la coda è vuota, allora entrambi i puntatori contengono `null`. Facendo la `enqueue` di un elemento, a entrambi i puntatori verrà assegnato il riferimento del nuovo elemento.

Problematica

Se `primo` e `ultimo` puntano a `null`, e quindi la coda è vuota, le funzioni `dequeue` e `first` genereranno un errore. Questi sono comunque degli errori corretti che sono intrinsechi al tipo coda.

Alberi binari

Gli alberi binari sono delle strutture in cui ad ogni nodo possono essere associati due successori, detti figlio sinistro e destro.



Profondità:

- la radice ha profondità 0
- i figli di un nodo di profondità k hanno profondità $k+1$ *(definizione induttiva)*

Altezza dell'albero: è la massima profondità delle foglie

Ci sono varie rappresentazioni per i nodi degli alberi, una di queste che è molto frequente estende quella delle liste lineari: si ha una parte con tutti i dati e due parti con i puntatori ai figli, sinistro e destro. È una rappresentazione adatta per muoversi dalla radice alle foglie.

Attraversamento di alberi binari

La **strategia generica** per visitare tutti i nodi di un albero consiste nel visitare la radice, ed ogni volta che si visita un nodo si memorizza in un insieme S i suoi figli. Si preleva poi da S il nodo successivo da visitare.

```
ALGORITMO visitaGenerica(AlberoBinario r)
    S <- {r}
    WHILE s != Ø DO
    |   preleva un nodo v da S
    |   visita v
    |   S <- S u {figli di v}
```

A seconda di come questo insieme S viene gestito, si ottengono degli algoritmi diversi. Se lo si implementa come coda si ottiene la BFS, come pila invece si ha una DFS.

Visita in ampiezza (BFS)

Si utilizza come struttura per memorizzare i nodi una coda.

```
ALGORITMO visitaInAmpiezza(AlberoBinario r) // r è un puntatore alla radice
    c <- coda vuota
    c.enqueue(r)
    WHILE NOT(c.isEmpty()) DO
    |   n <- c.dequeue()
    |   IF n != null THEN
    |   |   visita nodo associato a n
    |   |   c.enqueue(n.sx)
    |   |   c.enqueue(n.dx)
```

Visita in profondità (DFS)

Si utilizza come struttura per memorizzare i nodi una pila. L'algoritmo iterativo è:

```
ALGORITMO visitaInProfondità(AlberoBinario r) // r è un puntatore alla radice
    p <- pila vuota
    p.push(r)
    WHILE NOT(p.isEmpty()) DO
    |   n <- p.pop()
    |   IF n != null THEN
    |   |   visita nodo associato a n
    |   |   p.push(n.dx)
    |   |   p.push(n.sx)
```

DFS ricorsiva

La DFS è basata su una pila, ed essendo la pila la struttura utilizzata per implementare la ricorsione è possibile fare una variante ricorsiva della DFS. L'albero binario può essere infatti visto ricorsivamente come:

- **caso base**: albero vuoto $\rightarrow$ non c'è nulla da visitare
- **passo induttivo**: un nodo (radice) con due alberi binari (sottoalbero sx e sottoalbero dx) $\rightarrow$ si visita la radice, si prosegue visitando ricorsivamente il sottoalbero sx e poi quello destro

Utilizzando questa scomposizione è possibile creare un algoritmo ricorsivo della DFS. Il caso base non è necessario implementarlo in quanto non è necessario svolgervi alcuna operazione.

Algoritmo in **ordine anticipato**: visita prima la radice, poi sottoalbero sinistro e infine quello destro.

```
ALGORITMO visitaInProfondità(AlberoBinario r)
    IF r != null THEN
    |   visita la radice
    |   visitaInProfondità(r.sx)
    |   visitaInProfondità(r.dx)
```

Algoritmo in **ordine simmetrico**: visita prima il sottoalbero sinistro, poi la radice e infine il sottoalbero destro.

```
ALGORITMO visitaInProfondità(AlberoBinario r)
    IF r != null THEN
    |   visitaInProfondità(r.sx)
    |   visita la radice
    |   visitaInProfondità(r.dx)
```

Algoritmo in **ordine posticipato**: visita prima il sottoalbero sinistro, poi quello destro e infine la radice.

```
ALGORITMO visitaInProfondità(AlberoBinario r)
    IF r != null THEN
    |   visitaInProfondità(r.sx)
    |   visitaInProfondità(r.dx)
    |   visita la radice
```

L'ordine posticipato può essere utilizzato per leggere delle espressioni negli alberi in notazione polacca inversa

Numero di nodi

Si supponga di voler calcolare il numero di nodi di un albero binario. Basandoci sulla precedente definizione ricorsiva:

$$\#\text{ nodi} = \begin{cases} \text{vuoto} & 0 \\ \text{un nodo con due sottoalberi} & 1 + \#\text{ nodi sottoalbero sx} + \#\text{ nodi sottoalbero dx} \end{cases}$$

```
FUNZIONE #nodi(AlberoBinario r) -> intero
    IF r = null THEN RETURN 0
    ELSE
    |   #sx <- #nodi(r.sx)
    |   #dx <- #nodi(r.dx)
    |   RETURN 1 + #sx + #dx
```

Relazione tra numero di nodi e altezza

Dato un albero binario di altezza h:

- il numero **minimo** di nodi si ottiene quando ogni nodo ha un solo figlio, ovvero l'albero è una lista. È uguale a $n = h + 1$
- il numero **massimo** di nodi si ottiene quando ogni nodo ha entrambi i figli, cioè quando l'albero è un albero binario completo. È pari a $n = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$

Dimostrazione numero massi di nodi: **Induzione su $\mathbf{h}$**

- **Base**: $h = 0$
  Si ha solo la radice $\rightarrow 2^{0+1} - 1 = 2 - 1 = 1$
- **Induzione**: $h - 1 \rightarrow h$ con $h > 0$
  Un albero T di altezza h è composto da una radice, da un sottoalbero sinistro T_s e da un sottoalbero destro T_d. Per massimizzare il numero di nodi bisogna supporre che T_s e T_d abbiano a loro volta il numero di nodi massimo. Se l'altezza di T è pari ad h, allora le altezze di T_s e T_d sono $h - 1$. Per ipotesi di induzione possiamo dire che:

$$\text{altezza}(T_s) = \text{altezza}(T_d) = 2^h - 1$$
$$\#\text{ nodi}(T) = 1 + \#\text{ nodi}(T_s) + \#\text{ nodi}(T_d) = \cancel{1} + 2^h - \cancel{1} + 2^h - 1 = 2^{h+1} - 1$$

Abbiamo quindi dimostrato che:

$$h + 1 \leq n \leq 2^{h+1} - 1 \quad \Rightarrow \quad \log_2(n+1) - 1 \leq h \leq n - 1$$
$$\uparrow \text{ albero completo} \qquad \uparrow \text{ albero degenerato in lista}$$

Albero binario quasi completo

Un albero binario è quasi completo quando è completo almeno fino al penultimo livello

Proprietà: un albero binario di altezza h è quasi completo se e solo se ogni nodo di profondità $< h - 1$ possiede entrambi i figli.

Dato un albero binario quasi completo di altezza h:

- il numero **minimo** di nodi si ottiene quando si ha un albero completo di altezza $h - 1$ e almeno un nodo di altezza h. È pari a $2^h - \cancel{1} + \cancel{1} = 2^h$
- il numero **massimo** di nodi si ottiene quando si ha un albero completo di altezza h, ed è uguale a $2^{h+1} - 1$. Da questo si constata che un albero completo sia anche un albero quasi completo.

Di conseguenza: $2^h \leq n \leq 2^{h+1} - 1 \quad \Rightarrow \quad h = \lfloor \log_2 n \rfloor$

Alberi generici

Gli alberi generici (o con radice) hanno un numero arbitrario di figli.

- Un nodo particolare è chiamato radice ed è scelto come punto di partenza. Ogni nodo ha un unico padre (eccetto la radice che ne è priva). I figli di uno stesso padre sono chiamati fratelli.

Definizione induttiva: un albero con radice è o la struttura vuota, oppure un nodo a cui sono associati $k \geq 0$ alberi.

Le definizioni di arco, foglie, nordi interni rimangono le stesse, così come quelle di profondità e altezza.

Grado di un nodo: è il numero di figli del nodo.
Grado dell'albero: il massimo grado dei nodi.

È possibile utilizzare le visite in ampiezza e in profondità definite negli alberi binari anche per gli alberi generici.

Rappresentazioni indicizzate

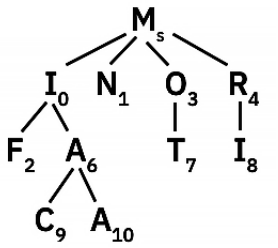
Vettore dei padri

Se si ha bisogno di una rappresentazione compatta che ci permetta di risalire ai padri partendo dai nodi figli si possono adottare due array paralleli:

- **array dei dati**: contiene le informazioni associate a ciascun nodo (i valori)
- **array dei padri**: contiene il riferimento del padre di ciascun nodo

In questa struttura il nodo radice, che non ha padre, viene gestito con -1. L'associazione tra gli elementi dei due array permette di ricostruire la gerarchia e navigare nell'albero.

	0	1	2	3	4	5	6	7	8	9	10
info	I	N	F	O	R	M	A	T	I	C	A
padre	5	5	0	5	5	-1	0	3	4	6	6



È una rappresentazione compatta e comoda se si ha bisogno di risalire dai figli ai padri

Vettore dei figli

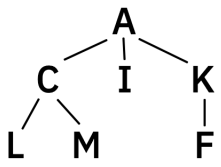
Se si ha bisogno di una rappresentazione che ci permetta di scendere ai figli a partire dal nodo padre

- **array dei dati**: contiene le informazioni associate a ciascun nodo (i valori)
- **array dei figli**: d array (dove d è il grado dell'albero) contenenti le posizioni dei figli di ogni nodo

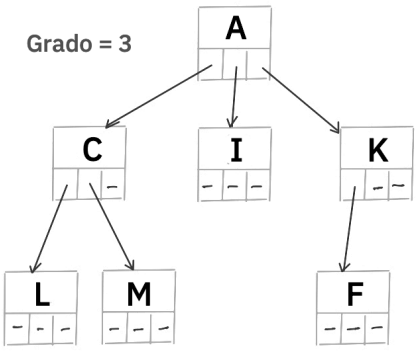
È una rappresentazione molto costosa: se c'è un nodo che ha grado molto alto mentre gli altri hanno grado molto basso, si dispone di tanti array solo per quel nodo. Inoltre se il grado dell'albero varia dinamicamente, e non è quindi fissato, bisogna aggiungere dinamicamente degli array.

Rappresentazioni collegate

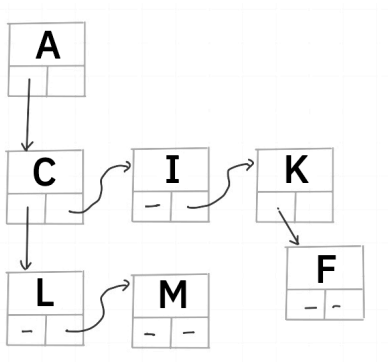
Si consideri il seguente albero generico. Sono possibili due diverse implementazioni:



Puntatori ai figli



Lista dei fratelli



Puntatori ai figli

Una rappresentazione con puntatori ai figli per un albero generico costituisce un'estensione diretta della struttura usata per gli alberi binari: invece di avere due soli puntatori (sinistro e destro), ogni nodo avrà un numero di puntatori pari al grado massimo dell'albero. In altre parole, se il nodo può avere fino a k figli, nel suo record saranno presenti:

- un campo per le informazioni
- un insieme di k puntatori, ciascuno riferito a un potenziale figlio

Questo meccanismo consente un rapido accesso a ciascun figlio, ma può comportare uno **spreco di spazio** se esiste almeno un nodo con grado molto elevato e la maggior parte degli altri nodi presenta un grado ridotto. In tal caso, bisogna comunque riservare in ogni record lo stesso numero di puntatori pari al grado massimo, anche se resteranno inutilizzati in molti nodi.

Lista dei fratelli

Una rappresentazione con lista dei fratelli per un albero generico è un'alternativa all'uso dei puntatori ai figli. Invece di memorizzare direttamente un puntatore per ciascun figlio, ogni nodo mantiene solo due puntatori:

- un puntatore al primo figlio (se presente)
- un puntatore al fratello successivo (se presente)

Questa rappresentazione riduce lo **spreco di spazio**, poiché ogni nodo ha sempre e solo due puntatori, indipendentemente dal numero di figli che può avere. Tuttavia, per accedere a un figlio specifico, potrebbe essere necessario percorrere la lista dei fratelli fino a trovare quello desiderato, rallentando l'accesso diretto.

Heap

Uno heap (o max-heap) è un albero binario quasi completo in cui la chiave contenuta in ciascun nodo è maggiore o uguale delle chiavi contenute nei figli. *Per comodità si considereranno heap in cui le foglie dell'ultimo livello si trovano il più a sinsitra possibile.*

La radice contiene di conseguenza l'elemento più grande. Essendo l'heap un albero binario quasi completo, la sua altezza è $h = \lfloor \log_2 n \rfloor$ dove n è il numero di nodi.

Sistemare uno heap con radice sbagliata

L'algoritmo "abbassa" la chiave iniziale della radice finché non trova la giusta collocazione. In pratica, salva la chiave della radice, confronta la radice con il figlio maggiore e, se il figlio è più grande, lo sposta verso l'alto e scende di un livello. Ripete finché non si arriva a una posizione in cui la chiave salvata è maggiore (o uguale) dei figli o si è in fondo all'albero; a quel punto "colloca" definitivamente la chiave salvata nel nodo corrente.

Procedura per sistemare uno heap con radice sbagliata (ossia non in ordine):

```
PROCEDURA risistema(Heap H)
    v <- H                  // v: nodo in esame
    x <- v.chiave           // x: chiave da sistemare
    y <- v.altri_campi      // y: campi associati a x
    da_collocare <- true
    DO
    |   IF v è una foglia THEN
    |   |   da_collocare <- false
    |   ELSE
    |   |   u <- figlio di v di valore max
    |   |   IF u.chiave > x THEN
    |   |   |   v.chiave <- u.chiave
    |   |   |   v.altri_campi <- u.altri_campi
    |   |   |   v <- u
    |   |   ELSE
    |   |   |   da_collocare <- false
    WHILE da_collocare
    v.chiave <- x
    v.altri_campi <- y
```

Sia h l'altezza dello heap, questo algoritmo svolge $\Theta(h)$ confronti ($\mathcal{O}(\log n)$ dove n è il numero di nodi dell'heap)

Costruzione di heap

Dato un albero binario quasi completo contentente gli elementi da ordinare, è possibile trasformarlo in uno heap in due modi:

1. **top-down**: si utilizza la tecnica divide-et-impera
2. **bottom-up**: si parte dai sottoalberi più piccoli fino a salire a quelli più grandi

Top-down (dividi-et-impera)

Nell'approccio top-down si procede ricorsivamente sui **sottoalberi**, trasformandoli in heap prima quello sinistro e poi quello destro. Alla fine, si "risistema" la **radice** (con la procedura risistema) per riportarla nella posizione corretta, nel caso non rispetti la proprietà di heap. Così, quando la ricorsione è conclusa, l'intero albero risulta uno heap.

```
PROCEDURA creaHeap(albero T)
    IF T != albero vuoto THEN
    |   creaHeap(T.sx)
    |   creaHeap(T.dx)
    |   risistema(T)
```

Questo approccio non viene utilizzato in quanto ricorsivo, e di conseguenza utilizza memoria aggiuntiva per lo stack

Bottom-up

Nella strategia bottom-up, si parte dalle **foglie** (che di per sé rispettano già la proprietà di heap) e si risale via via i livelli fino a raggiungere la radice (si sale quindi di profondità). A ogni livello, si scandiscono i nodi da **destra verso sinistra**. Per ciascuno di questi nodi, se la radice del sottoalbero non rispetta la proprietà di heap, si applica la procedura `risistema`, scambiandolo con il figlio appropriato (maggiore o minore a seconda che lo heap sia un max-heap o un min-heap). In questo modo, quando si arriva in cima, l'intero albero risulta correttamente trasformato in uno heap.

```
PROCEDURA creaHeap(Albero T)
    h <- altezza T
    FOR p <- h DOWN TO 0 DO
    |   FOR EACH nodo x di profondità p DO
    |   |   risistema(sottoalbero di radice x)
```

Questo algoritmo non utilizza memoria aggiuntiva.

Analisi confronti

Sia n il numero di nodi dello heap e sia $h = \lfloor \log_2 n \rfloor$ l'altezza.

Analisi immediata

- La procedura `creaHeao` effettua n chiamate di `risistema`
- Ogni chiamata a `risistema` è su un sottoalbero di altezza $h' \leq h$, e di conseguenza effettua al massimo $\mathcal{O}(h') = \mathcal{O}(h)$ confronti

$\#$ totale di confronti $= n\,\mathcal{O}(h) = \mathcal{O}(n \cdot h) = \mathcal{O}(n \log n)$

Con un'analisi più raffinata però si dimostra che il numero di confronti è $\Theta(n)$

Analisi migliorata

Sia p la profondità dell'albero presa in considerazione.

- $\#$ nodi alla profondità p: 2^p
- altezza di un sottoalbero con radice di profondità p: $h - p \quad \Rightarrow \quad \#$ confronti di risitema $= \Theta(h-p)$

$\#$ totale di confronti alla profodità $p = 2^p\,\Theta(h-p)$

Si calcoli ora il numero di confronti totale che viene eseguito a ciascuna profondità dell'albero:

$$\sum_{p=0}^{h} 2^p(h-p) = \sum_{p=0}^{h} h\,2^p - \sum_{p=0}^{h} p\,2^p = h\sum_{p=0}^{h} 2^p - \sum_{p=0}^{h} p\,2^p = h\,(2^{h+1}-1) - (h-1)\,2^{h+1} - 2 =$$

$$= \cancel{h\,2^{h+1}} - h - \cancel{h\,2^{h+1}} + 2^{h+1} - 2 = 2^{h+1} - h - 2 \approx 2n - \log_2 n = \Theta(n) \qquad \sum_{i=0}^{n} i\,2^i = (n-1)\,2^{n+1} + 2 \uparrow$$

Heap Sort

Prima implementazione

```
ALGORITMO heapSort(Array A) -> lista
    crea uno heap H a partire dall'array A
    |   - costruisci un albero binario quasi completo a partire da A
    |   - creaHeap(H)                                                    [1]

    X <- lista vuota
    WHILE H != Ø DO
    |   rimuovi da H il contenuto della radice e aggiungilo all'inizio di X
    |   rimuovi la foglia più a dx dell'ultimo livello e poni il suo contenuto nella radice
    |   risistema H                                                      [2]

    RETURN X
```

Numero di confronti

La procedura per creare uno heap `[1]` svolge $\Theta(n)$ confronti.
Il ciclo while viene eseguito una volta per ogni nodo, ossia per n iterazioni. All'interno del ciclo troviamo la procedura per risistemare lo heap `[2]` che svolge $\mathcal{O}(\log n)$ confronti. Di conseguenza all'interno del ciclo si svolgono in totale $\mathcal{O}(n \log n)$ confronti

$$\#\text{ totale di confronti} = \Theta(n) + \mathcal{O}(n \log n) = \mathcal{O}(n \log n)$$

Questo algoritmo però utilizza memoria aggiuntiva poichè bisogna creare una lista, e non è quindi in loco.

Implementazione in loco

Si ha un array di n elementi da ordinare. Si costruisce il relativo albero binario seguendo il procedimento della visita in ampiezza, ossia lo si riempe partendo dall'alto verso il basso e da sinistra verso destra: così facendo si ottiene un albero quasi completo che presenta all'ultimo livello tutti i figli spostati verso sinistra. Questo albero è quindi ideale per essere trasfortmato in uno heap.

Si effettua la seguente osservazione: considerando nell'albero l'elemento dell'array di indice i, i suoi due figli hanno rispettivamente nell'array indice $2i+1$ e $2i+2$.

- Così facendo, dall'array è possibile risalire direttamente ai figli di un certo elemento, e tramite la formula inversa è possibile risalire al padre di un certo elemento.

Sistemare uno heap con radice sbagliata tramite indici

Modifichiamo la procedura `risistema` utilizzando un approccio tramite gli indici. Sia r l'indice della radice ed l il primo indice dopo lo heap (la prima parte dell'array contiene lo heap, mentre partendo da l contiene la sequenza ordinata):

```
PROCEDURA risistema(Array A[0..n-1], intero r, intero l)
    v <- r          // v: posizione in esame
    x <- A[v]       // x: chiave da sistemare
    // per semplicità viene indicata solo la chiave
    da_collocare <- true
    DO
    |   IF 2*v + 1 >= l THEN    // v è l'indice di una foglia
    |   |   daCollocare <- false
    |   ELSE
    |   |   // u <- indice del figlio di v di valore max
    |   |   u <- 2*v + 1                        // primo figlio
    |   |   IF u+1 < l AND A[u+1] > A[u] THEN
    |   |   |   u <- u + 1                      // secondo figlio
    |   |   IF A[u] > x THEN
    |   |   |   A[v] <- A[u]
    |   |   |   v <- u
    |   |   ELSE
    |   |   |   daCollocare <- false
    |   WHILE daCollocare
```

Per come è implementato l'algoritmo i figli di un elemento in posizione i stanno alle posizioni $2i+1$ e $2i+2$. La procedura `risistema` prende il valore della radice e, se un figlio è più grande, lo scambia con quest'ultimo. Continua a "far scendere" il valore (spostandosi all'indice del figlio) finché non trova un posto in cui entrambi i figli siano più piccoli (o non esistano). In questo modo si ripristina la struttura di "max-heap" nella porzione dell'array che stiamo usando per l'ordinamento.

Costruzione di heap tramite indici

Utilizzando questa variante di `risistema` si reimplementa la procedura `creaHeap`

```
PROCEDURA creaHeap(Array A[0..n-1])
    FOR i <- n-1 DOWN TO 0 DO
        risistema(A, i, n)
```

- i è la radice del sottoalbero corrispondente all'elemento i dell'array
- dato che le foglie sono già degli heap di per sè, possiamo evitarle considerando $i \leftarrow \lfloor \frac{n}{2} \rfloor$

Partendo da $i = n-1$ e andando a ritroso verso $i = 0$, chiamiamo `risistema(A, i, n)`. Ogni chiamata "sistema" il sottoalbero con radice in i affinché rispetti la proprietà di max-heap. Dato che i nodi con indici più alti (in basso nell'albero) sono già stati trasformati in heap, l'elemento `A[i]` può scendere nel sottoalbero finché non trova la posizione corretta. Alla fine del ciclo, l'intero array (da 0 a $n-1$) risulta organizzato come un max-heap.

Heap Sort tramite indici

Con queste due nuove procedure è possibile creare un algoritmo per lo heap sort che funzioni in loco:

```
ALGORITMO heapSort(Array A[0..n-1])
    creaHeap(A)
    FOR l <- n-1 DOWN TO 1 DO
    |   scambia A[0] con A[l]
    |   risistema(A, 0, l)
```

La funzione `creaHeap(A)` rende l'intero array un max-heap, portando il valore massimo in `A[0]`. Poi nel ciclo for si scambia `A[0]` (massimo corrente) con `A[l]`, posizionandolo definitivamente in fondo all'array, e si richiama `risistema(A, 0, l)` per ripristinare la struttura di heap nelle sole posizioni da 0 a $l-1$. Ripetendo questi passi, alla fine tutti gli elementi risultano ordinati.

In conclusione:

- Heap sort è un algoritmo di ordinamento in loco, che utilizzando per `creaHeap` l'approccio bottom-up non richiede nessuna memoria aggiuntiva.
- Il numero di confronti è $\Theta(n \log n)$, e quindi il tempo è a sua volta $\Theta(n \log n)$ se i confronti costano $\mathcal{O}(1)$
- È un algoritmo di ordinamento **non stabile**

Operazioni su Heap

Trova elemento di chiave massima: $\mathcal{O}(1)$ passi
Per definizione nello heap la radice è l'elemento di chiave massima.

Cancella elemento di chiave massima: $\Theta(\log n)$ passi
Si toglie l'elemento di chiave massima che è la radice, si mette nella radice l'ultima foglia (ossia quella più a destra) ed infine si risistema lo heap con il relativo algoritmo `risistema`. Dipende quindi dalla profondità dell'heap.

Inserisci un nuovo elemento: $\Theta(\log n)$ passi
Essendo lo heap implementato tramite un vettore, si crea una nuova foglia e si aggiunge quindi l'elemento al vettore subito dopo l'indice l.
Si crea poi una nuova procedura `risistemaDalBasso`, che funziona in maniera opposta a `risistema`: si parte dalla foglia e si controlla che questa non sia maggiore del padre, altrimenti la si fa salire eventualmente fino alla radice con gli opportuni scambi. La complessità di questa funzione dipende anch'essa dalla profondità dell'heap.

Cancella un elemento di chiave x: $\Theta(\log n)$ passi
Supponendo di conoscere già la posizione dell'elemento da cancellare, si sostituisce all'elemento x l'ultima foglia (che viene eliminata). Sia f la chiave dell'ultima foglia (che ora si trova al posto di x). Si presentano due casi:

- se $f < x$ significa che rispetto al padre è in ordine, ma non è detto che sia sistemato rispetto ai figli. Di conseguenza si applica `risistema` a partire proprio da quel nodo.
- se $f > x$ significa che rispetto ai figli è in ordine, ma non è detto che sia sistemato rispetto al padre. Di conseguenza si applica `risistemaDalBasso` proprio a partire da quel nodo.

Modifica la chiave di un elemento: $\Theta(\log n)$ passi
Supponendo di conoscere già la posizione dell'elemento da modificare, si modifica la chiave con il nuovo valore: se la chiave è stata diminuita si applica `risistema`, se invece è stata aumentata si applica `risistemaDalBasso`.

La posizione dell'elemento può essere nota tramite l'utilizzo di strutture ausiliarie.

Code con priorità

La coda con priorità è una collezione di dati da cui gli elementi vengono prelevati secondo un criterio di priorità. Ogni elemento ha una chiave, dove una chiave più bassa indica una priorità maggiore.

Sono implementate mediante min-heap. Esistono implementazioni più efficienti basati su heap detti heap di Fibonacci.

Operazioni

`findMin()` $\mathcal{O}(1)$ passi
Restituisce l'elemento minimo della coda senza rimuoverlo

`deleteMin()` $\Theta(\log n)$
Restituisce l'elemento minimo della coda e lo rimuove

`insert(elemento e, chiave k)` $\Theta(\log n)$
Inserisce nella coda un elmento e con associata una chiave (priorità) k

`delete(elemento e)` $\Theta(\log n)$ *(se si conosce già la posizione dell'elemento)*
Cancella l'elemento e

`changeKey(elemento e, chiave d)` $\Theta(\log n)$ *(se si conosce già la posizione dell'elemento)*
Modifica la priorità dell'elemento e, assegnangoli come nuova chiave d

Ordinamenti basati su confronti

Problema dell'ordinamento:

- **Input**: n elementi $x_1, x_2, \cdots, x_n$ provenienti da un dominio $\mathcal{D}$ su cui è definita una relazione $\leq$ di ordine totale
- **Output**: sequenza $x_1, x_2, \cdots, x_n$ con $(j_1, j_2, \cdots, j_n)$ permutazione di $(1, 2, \cdots, n)$ t.c. $x_{j_1} \leq x_{j_2} \leq \cdots \leq x_{j_n}$

$\#$ permutazioni di n elementi $= n!$

Riassunto algoritmi di ordinamento

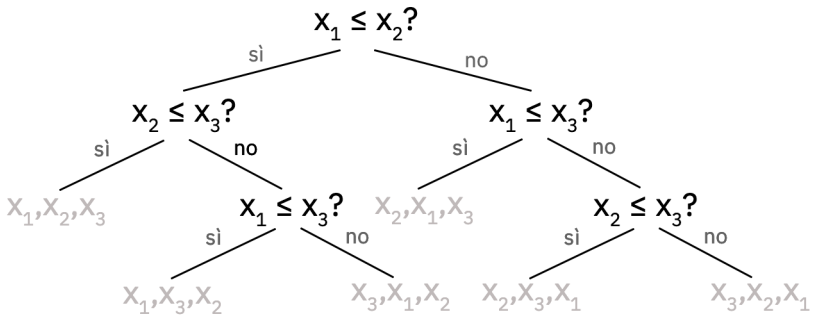
Algoritmo	Numero Confronti	Spazio	Note	Stabile
selectionSort	$\Theta(n^2)$ sempre	$\Theta(1)$	in loco	no
insertionSort	$\Theta(n^2)$ nel caso peggiore $n-1$ su array già ordinato	$\Theta(1)$	in loco	sì
bubbleSort	$\Theta(n^2)$ nel caso peggiore $n-1$ su array già ordinato	$\Theta(1)$	in loco	sì
mergeSort	$\Theta(n \log n)$	$\Theta(n)$	spazio $\Theta(n)$ per array ausiliario più $\Theta(\log n)$ per stack ricorsione	sì
quickSort	$\Theta(n^2)$ nel caso peggiore $\Theta(n \log n)$ nel caso migliore $\approx 1.39n \log_2 n$ in media	$\Theta(n)$ $\Theta(\log n)$	in loco spazio $\Theta(1)$ più stack ricorsione: - $\Theta(n)$ algoritmo base - $\Theta(\log n)$ algoritmo migliorate	no
heapSort	$\Theta(n \log n)$	$\Theta(1)$	in loco	no

Limite inferiore ordinamenti su confronti

Non è possibile ordinare array di n elementi utilizzando un numero di confronti che cresca meno di $n \log n$.

Questo può essere dimostrato tramite l'**albero di decisione**. Esso permette di rappresentare tutte le strade che la computazione di uno specifico algoritmo può intraprendere, sulla base dei possibili esiti dei test previsti dall'algoritmo stesso. Quindi in questo albero:

- i nodi interni sono delle domande
- le foglie sono i possibili risultati



Esempio

Questo rappresenta l'albero di decisione relativo ad un algoritmo di ordinamento che riceve in input tre elementi x_1, x_2, x_3. Il numero di foglie è pari a 3! = 6

Nel caso degli algoritmi di ordinamento basati su confronti, ogni test effettuato ha due soli possibili esiti. Quindi l'albero di decisione relativo a un qualunque ordinamento basato su confronti ha queste proprietà:

- è un **albero binario** che rappresenta tutti i possibili confronti che vengono effettuati dall'algoritmo
- ogni **nodo interno** rappresenta un singolo confronto tra chiavi, ed i due figli del nodo sono relativi ai due possibili esiti di tale confronto *(per esempio il primo elemento è maggiore / minore o uguale del secondo)*
- ogni **foglia** rappresenta una possibile soluzione del problema, la quale è una specifica permutazione della sequenza in ingresso (sono i possibili ordinamenti della sequenza, e quindi vi sono almeno $n!$ foglie)

Il numero di confronti necessari è dato dall'**altezza dell'albero**. Richiamando la relazione tra il numero di nodi di un albero binario e la sua altezza possiamo dire che: $\text{altezza albero binario} \geq \log_2(\#\text{nodi}) \geq \log_2(\#\text{foglie})$

Prendendo in considerazione l'albero di decisione degli ordinamenti su confronti: $\#\text{confronti} \geq \log_2(n!)$

Tramite l'approssimazione di Stirling possiamo dire che $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

Otteniamo quindi: $\log_2(n!) \approx \log_2\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right) = \log_2\sqrt{2\pi} + \frac{1}{2}\log_2 n + n\log_2 n - n\log_2 \mathrm{e} = \Theta(n \log n)$

Ogni algoritmo di ordinamento basato su confronti richiede perciò, nel caso peggiore, un numero di confronti dell'ordine di almeno $n \log n$.

Ordinamento senza confronti

Ogni algoritmo di ordinamento basato su confronti tra chiavi deve fare almeno $n \log n$ confronti: se ogni confronto impiega tempo costante, il tempo di esecuzione dell'algoritmo deve essere almeno $\Omega(n \log n)$. Esistono però algoritmi di ordinamento che sono più veloci per certi tipi di chiave perché non fanno confronti.

IntegerSort

Si supponga di dover ordinare n interi in un intervallo $[0..k-1]$. Si utilizza un array ausiliario Y di dimensione k per contare quante copie ci siano per ogni numero. Si osserva che `integerSort` non effettua alcun confronto fra elementi, e quindi per analizzarlo bisogna considerare il tempo di esecuzione nell'accezione più generale.

```
ALGORITMO integerSort(Array A[0..n-1], intero k)
    Sia Y[0..k-1] un array
    FOR i <- 0 TO k-1 DO
    |   Y[i] <- 0

    FOR i <- 0 TO n-1 DO
    |   x <- A[i]
    |   Y[x] <- Y[x] + 1

    j <- 0
    FOR i <- 0 TO k-1 DO
    |   WHILE Y[i] > 0 DO
    |   |   A[j] <- i
    |   |   j <- j + 1
    |   |   Y[i] <- Y[i] - 1
```

Così facendo si possono indicizzare k valori diversi, e il tempo di esecuzione risulta parti a $\mathcal{O}(n+k)$. Ogni elemento di Y deve infatti essere inizializzato e poi letto almeno una volta durante la ricostruzione dell'array di partenza.

Si osserva che il tempo $\mathcal{O}(n+k)$ è lineare nella dimensione dell'istanza se k è $\mathcal{O}(n)$.

Questo ordinamento tiene traccia solo dei numeri interi. Nel caso in cui questi numeri siano delle chiavi per dei record, questo algoritmo perde i dati associati.

BucketSort

Si supponga di dover ordinare n record con chiavi intere nell'intervallo $[0..k-1]$.

Si generalizza quindi il problema assumendo che gli elementi da ordinare non siano necessariamente numeri interi. Questo genera un problema in quando non è più possibile utilizzare dei contatori come nell'algoritmo di `integerSort` in quanto potrebbero esistere elementi diversi ma con la stessa chiave.

Si ovvia a questo problema utilizzando sempre un array ausiliario Y di dimensione k, ma i cui elementi sono delle liste anziché dei contatori. L'algoritmo, durante la lettura della sequenza, copia ogni record nella lista appropriata in base al valore della corrispondente chiave.
Alla fine è sufficiente concatenare tutte le liste ordinatamente, e quindi per valori di k crescenti.

```
ALGORITMO bucketSort(Array A[0..n-1], intero k)
    Sia Y[0..k-1] un array
    FOR i <- 0 TO k-1 DO        // [1] predisposizione bucket
    |   Y[i] <- coda vuota

    FOR i <- 0 TO n-1 DO        // [2] riempimento bucket
    |   x <- A[i].chiave
    |   Y[x].enqueue(A[i])

    j <- 0
    FOR i <- 0 TO k-1 DO        // [3] riempimento array A ordinato
    |   WHILE NOT Y[i].isEmpty() DO
    |   |   A[j] <- Y[i].dequeue()
    |   |   j <- j + 1
```

Si procede facendo l'analisi dell'algoritmo:

- `[1]` si svolgono k iterazioni di un'operazione a tempo costante $\rightarrow \Theta(k)$
- `[2]` si svolgono n iterazioni di due operazioni a tempo costante $\rightarrow \Theta(n)$
- `[3]` si svolgono k iterazioni, però l'operazione di `dequeue` sulla coda è eseguita n volte durante tutta l'esecuzione dell'algoritmo proprio perché la coda contiene esattamente n chiavi $\rightarrow \Theta(n+k)$

Il tempo di esecuzione totale di `bucketSort` è quindi anch'esso pari a $\mathcal{O}(n+k)$. Si nota che:

- se $k = \mathcal{O}(n) \rightarrow \text{tempo } \Theta(n)$
- se $k = \Theta(n^2) \rightarrow \text{tempo } \Theta(n^2)$. Conviene perciò usare un algoritmo come l'heapSort.

In conclusione, l'algoritmo di `bucketSort` non è in loco, è **stabile** in quanto utilizza le code ed è necessario conoscere il valore di k in anticipo.

L'algoritmo di bucket sort può essere utilizzato per ordinare le liste manipolando direttamente i puntatori: si colloca ciascuno nodo di A nella coda corrispondente alla chiave, e poi si concatenano le code in A una dopo l'altra. La complessità temporale dell'algoritmo rimane la stessa.

> Si supponga di ordinare un insieme di persone rispetto alla data del compleanno, si procedere come segue:
>
> 1. si ordina la lista tramite bucket sort rispetto al giorno
> 2. si ordina la lista tramite bucket sort rispetto al mese
>
> La stabilità di bucket sort ci garantisce che l'elenco finale sia ordinato correttamente, proprio perché viene mantenuto l'ordine relativo.
>
> Ordinare prima i giorni e poi i mesi ci permette di fare un unico bucket sort su tutti i dati rispetto al giorno (31 bucket) e poi di fare un altro bucket sort su tutti i dati rispetto al mese. Così facendo non c'è bisogno di utilizzare delle liste separate.
>
> Facendo invece il contrario, dopo aver ordinato rispetto al mese bisogna crearsi 12 elenchi separati in cui bisogna fare 12 bucket sort diversi.

RadixSort [not finished]

```
PROCEDURA bucketSort(Array A[0..n-1], intero b, intero t)
    Sia Y[0..b-1] un array
    FOR i <- 0 TO b-1 DO        // [1] predisposizione bucket
    |   Y[i] <- coda vuota
    
    FOR i <- 0 TO n-1 DO        // [2] riempimento bucket
    |   x <- t-esima cifra nella rappresentazione in base b di A[i].chiave
    |   Y[x].enqueue(A[i])
    
    j <- 0
    FOR i <- 0 TO b-1 DO        // [3] riempimento array A ordinato
    |   WHILE NOT Y[i].isEmpty() DO
    |   |   A[j] <- Y[i].dequeue()
    |   |   j <- j + 1
    
ALGORITMO radixSort(Array A[0..n-1])
    t <- 0
    Sia b la base da utilizzare
    WHILE (esiste chiave K in A tale che ⌊k/b^t⌋ != 0) DO   [a]
    |   bucketSort(A, b, t)
    |   t <- t+1
```

`[a]` Questa condizione controlla che A contenga una chiave composta da almeno $t+1$ cifre.

- in pratica ci si chiede se bisogna andare avanti ancora oppure no in base alla grandezza dei numeri

b è il numero di bucket, ma è anche la base che si prende in considerazione. t è la cifra rispetto alla quale si ordina, con $t=0$ la cifra meno significativa.

`[1]` Si predisponde l'array dei bucket

`[2]` Si mettono gli elementi nei bucket. Per piazzarli si estrae dalla chiave considerata la cifra di posto t in base b, e poi la si aggiunge alla coda corrispondente.

`[3]` Si travasa dai bucket nell'array o nella lista

La procedura `bucketSort` lavora in tempo $\mathcal{O}(n+b)$ dove però b è un numero fissato a priori.

L'algoritmo `radixSort` non fa altro che chiamare la procedura bucket sort. Il numero di chiamate che deve fare dipende dalla grandezza dei numeri: finché la condizione `[a]` è verificata bisogna applicare il bucket sort.

Si supponga di scegliere come base utilizzata $b=10$. Se bisogna ordinare n chiavi tra 0 e $10^{10}-1$, si fanno 9 iterazioni di bucket sort, con dei bucket molto piccoli. Scegliendo invece $b=10^3$ è possibile riordinare queste chiavi con solo 3 iterazioni di bucket sort soltano utilizzando un array di supporto di grandezza 1000. Inoltre il tempo di ...

Rappresentazione di partizioni

Definizione partizione

Dato un insieme A, definiamo partizione di A una famiglia si sottoinsiemi $a_1, a_2, \cdots, a_k \subseteq A$ tali che:

- $a_i \neq \varnothing$ per $i = 1, \cdots, k$
- $a_i \cap a_j = \varnothing$ per $i,\ j = 1, \cdots, k \quad i \neq j$
- $a_1 \cup a_2 \cup \cdots \cup a_k = A$

Ogni sottoinsieme $a_1, a_2, \cdots, a_k$ è detto parte di A *(e non partizione)*

Problema Union-Find

Il problema Union-Find consiste nel mantenere una collezione di insiemi disgiunti su cui effettuare le seguenti operazioni:

- `union(A, B)` unisce gli insiemi A e B in un unico insieme di nome A
- `find(x)` restituisce il nome dell'insieme che contiene l'elemento x
- `makeSet(x)` crea un nuovo insieme $\{x\}$ di nome x

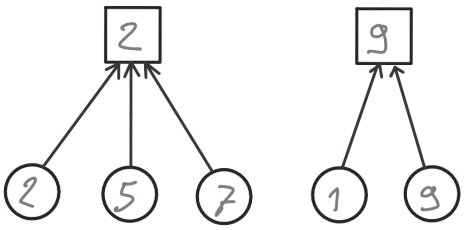
Si osserva che su n elementi, creati con n `makeset`, il numero di `union` che si possono chiamare è limitato superiormente da $n - 1$, dopodiché rimarrà un unico insieme contente tutti gli elementi.

Vi sono soluzioni elementari e soluzioni più evolute. In tutte le soluzioni presentate ogni insieme è rappresentato da un albero con radice dove: i nodi sono gli elementi dell'insieme, la radice è il nome dell'insieme, la foresta è l'insieme degli insiemi disgiunti, ognuno dei quali è un albero come qua descritto. A loro volta gli alberi sono rappresentati con puntatori verso l'alto (in queste strutture infatti torna utile utilizzare dei puntatori che dai figli risalgono al padre, ossia alla radice. Questo può essere implementato con il vettore dei padri).

quickFind

Negli algoritmi di tipo `quickFind` si adoperano alberi con radice di altezza 1 nei quali la radice rappresenta il nome dell'insieme e gli elementi dell'insieme sono le foglie. L'operazione che si svolge velocemente è appunto la ricerca.

Lo spazio utilizzato è $\mathcal{O}(n)$, con n il numero di elementi



Alberi che rappresentano gli insiemi 2 = {2,5,7} *e* 9 = {1,9}

Si analizzino ora le strategie e i costi per implementare le operazioni di questa rappresentazione, con n il numero di elementi dell'insieme:

- `makeSet(elemento e)` crea un albero dalla radice e e dal figlio e. $T(n) = \mathcal{O}(1)$
- `find(elemento e) -> nome` resituisce il nome dell'insieme che contiene e semplicemente risalendo tramite il puntatore al padre. $T(n) = \mathcal{O}(1)$
- `union(nome A, nome B)` unisce i due alberi sostituendo i puntatori delle foglie di B con puntatori verso la radice di A e cancella poi B. Nel caso peggiore B contiene $n - 1$ elementi, mentre A contiene un solo elemento. $T(n) = \mathcal{O}(n)$

quickFind con bilanciamento

È possibile migliorare gli algoritmi di tipo `quickFind` ottimizzando l'operazione di `union(A, B)`. Nella `quickFind` questa prevede di collegare tutti i nodi di B verso A, il che è sconveniente se la cardinalità di B è maggiore di quella di A. In tal caso è conveniente collegare i nodi di A verso B, tenendo conto che bisogna anche cambiare il nome della radice di B con quello della radice di A. Si osserva che vengono spostati al massimo $\frac{n}{2}$ elementi.

A primo impatto si direbbe che la complessità di questa operazione rimane $\mathcal{O}(n)$, ma tramite un'analisi di costo ammortizzato si osserva che se capitano operazioni di unione costose allora quelle dopo non lo saranno. Si può infatti dimostrare infatti che efettuando una sequenza di n `makeSet` e di $\mathcal{O}(n)$ `union` e `find` il tempo totale è $\mathcal{O}(n \log n)$. Ciò significa che in media queste `union` costano $\mathcal{O}(\log n)$.

Il **costo ammortizzato** dell'operazione di `union` è quindi $\mathcal{O}(\log n)$. *Il principio alla base del costo ammortizzato è che durante una sequenza di operazioni, in questo caso di union, ce n'è una che costa tanto, ma poi per tante altre volte costa poco. Si ha quindi un caso peggiore che però è "rarefatto" nel tempo.*

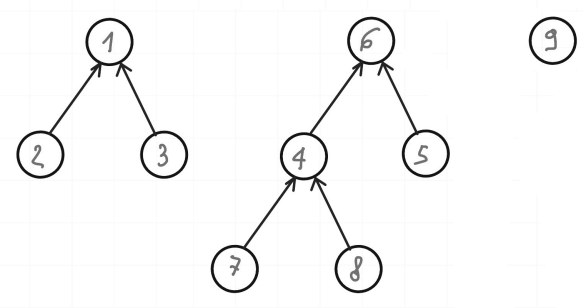
Nella radice si memorizza la cardinalità dell'insieme. Di conseguenza l'operazione `makeSet` memorizzerà 1 nella radice, mantenendo però una complessità costante. La `find` rimane invariata.

```
PROCEDURA Union(nome A, nome B)
    IF size(A) >= size(B) THEN
    |   sposta i puntatori dalle foglie di B ad A
    |   rimuovi la radice di B
    ELSE
    |   sposta i puntatori dalle foglie di A a B
    |   rinomina la radice di B come A
    |   rimuovi la vecchia radice di A
    size(A) <- size(A) + size(B)
```

quickUnion

Negli algoritmi di tipo `quickFind` si adoperano alberi con radice di varie altezze nei quali i nodi sono gli elementi dell'insieme e la radice è l'elemento che dà il nome all'insieme. L'operazione che si svolge velocemente è appunto l'unione.

Lo spazio utilizzato è $\mathcal{O}(n)$, con n il numero di elementi



Alberi che rappresentano gli insiemi
1 = {1,2,3} , 6 = {6,4,5,7,8} *e* 9 = {9}

Si analizzino ora le strategie e i costi per implementare le operazioni di quest a rappresentazione, con n il numero di elementi dell'insieme:

- `makeSet(elemento e)` crea un albero con un unico nodo (radice) che contiene l'elemento e. $T(n) = \mathcal{O}(1)$
- `find(elemento e) -> nome` accede al nodo e e segue i puntatori ai padri fino ad arrivare alla radice, che contiene il nome dell'insieme. Il tempo dipende quindi dalla distanza di e dalla radice; il caso peggiore si ottiene con un albero di grado 1 con tutti gli n elementi (ossia una lista). $T(n) = \mathcal{O}(n)$
- `union(nome A, nome B)` unisce i due alberi rendendo la radice di B figlia di A. Questo viene fatto collegando il puntatore della radice di B alla radice di A. $T(n) = \mathcal{O}(1)$

La `union` può far aumentare l'altezza degli alberi. Si supponga di fare la `makeSet` di n elementi e di unire tutti questi insiemi disgiunti consecutivamente: si viene a creare un albero di altezza $n - 1$, ossia una lista.

quickUnion con bilanciamento in altezza

È possibile migliorare gli algoritmi di tipo `quickUnion` cercando di evitare che l'albero abbia un'altezza eccessiva. Si indichi con `rank(x)` l'altezza di un albero `x`. Per evitare che l'altezza dell'albero cresca velocemente, nella `union(A, B)` non si attacca B sotto ad A ma si attacca l'albero di altezza minore sotto l'albero di altezza maggiore. Se `rank(B) > rank(A)` allora bisogna cambiare il nome della radice di B scambiandola con quella di A.

Nella radice si memorizza l'altezza dell'albero. Di conseguenza l'operazione `makeSet` deve memorizzare nella radice che il rank è 1. La `find` rimane invariata ed il suo tempo dipende ancora dall'altezza dell'albero.

```
PROCEDURA Union(nome A, nome B)
    IF rank(A) > rank(B) THEN
    |   rendi la radice di B figlia di quella di A
    ELSE IF rank(A) < rank(B) THEN
    |   rendi la radice di A figlia di quella di B
    |   scambia i contenuti dei nodi A e B
    ELSE
    |   rendi la radice di B figlia di quella di A
    |   rank(A) <- rank(A) + 1  // unico caso in cui aumenta l'altezza
```

Il tempo della `union` rimane costante, ma sarà più alta di quella di prima per via dei passi in più da svolgere.

Lemma: ogni albero "quickUnion" costruito effettuando operazioni di unione bilanciate contiene almeno $2^{\text{rank}(x)}$ nodi *(contiene quindi un numero di nodi esponenziale)*: $\#\text{nodi}(x) \geq 2^{\text{rank}(x)}$
Dimostrazione per induzione sul numero k di operazioni di unione con cui l'albero è stato costruito:

- **Base**: $k = 0$
  Non sono state effettuate unioni, di conseguenza l'albero ha solo la radice: $n = 1$ e $h = 0$, vale quindi $n \geq 2^0$
- **Induzione**: $k > 0$
  Si suppone che l'ipotesi induttiva valga per un numero minore di $k > 0$ unioni. Nell'ultima operazione `union(A, B)` gli insiemi A e B sono stati costruiti con meno di k union. Chiamando x l'albero finale che si viene a creare dopo l'unione, si dimostrano le tre casistiche:
  - `rank(A) > rank(B)` $\rightarrow \text{rank}(x) = \text{rank}(A)$

    $$\#\text{nodi}(x) = \#\text{nodi}(A) + \#\text{nodi}(B) \underset{\substack{\uparrow \\ \text{ipotesi induttiva}}}{\geq} 2^{\text{rank}(A)} + 2^{\text{rank}(B)} > 2^{\text{rank}(A)} = 2^{\text{rank}(x)}$$

    $$\Rightarrow \#\text{nodi}(x) > 2^{\text{rank}(x)}$$

  - `rank(A) < rank(B)` $\rightarrow \text{rank}(x) = \text{rank}(B)$
    dimostrazione analoga al caso sopra
  - `rank(A) = rank(B)` $\rightarrow \text{rank}(x) = \text{rank}(A) + 1$

    $$\#\text{nodi}(x) = \#\text{nodi}(A) + \#\text{nodi}(B) \underset{\substack{\uparrow \\ \text{ipotesi induttiva}}}{\geq} 2^{\text{rank}(A)} + 2^{\text{rank}(B)} \underset{\substack{\uparrow \\ \text{rank}(A) = \text{rank}(B)}}{=} 2 \cdot 2^{\text{rank}(A)} = 2^{\text{rank}(A)+1} = 2^{\text{rank}(x)}$$

    $$\Rightarrow \#\text{nodi}(x) \geq 2^{\text{rank}(x)}$$

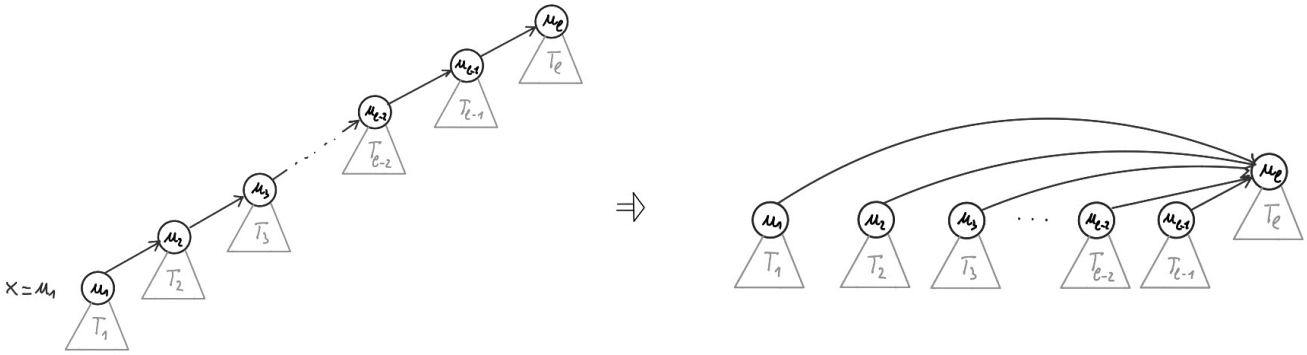
Corollario: se l'albero ha n nodi $\Rightarrow$ altezza $\leq \log_2 n$

Dal corollario consegue che la `find` utilizzi un tempo $\mathcal{O}(\log n)$

quickUnion bilanciata in altezza con compressione di cammino

Per migliorare i tempi di esecuzione bisogna diminuire l'altezza degli alberi. Si introduce quindi un'euristica che ha proprio lo scopo di comprimere il cammino visitato durante un'operazione `find`.

Per $l \geq 3$, siano $u_0, u_1, \cdots, u_{l-1}$ i nodi incontrati nel cammino esaminato da una `find(x)`, dove $u_0 = x$ ed u_{l-1} è la radice dell'albero contenente x. L'euristica di compressione del cammino rende tutti i nodi u_i per $0 \leq i \leq l-3$ figli della radice u_{l-1}.
Si nota che u_{l-2} è già figlio della radice u_{l-1}, motivo per il quale i casi con $l \leq 2$ non sono presi in considerazione



Le `find` successive quindi sono avvantaggiate da questa ristrutturazione che viene fatta. Più `find` vengono fatte e più gli alberi tengono ad appiattirsi.

Su può dimostrare che effettuando una sequenza di n `makeset` ed $\mathcal{O}(n)$ `union` e `find` il tempo totale è $\mathcal{O}(n \log^* n)$. Da questo si ricava che il costo ammortizzato della `find` sia $\mathcal{O}(\log^* n)$. Nella pratica questo costo può essere considerato costante per valori di n che possono essere rappresentati nei computer.

Riepilogo Union-Find

	makeSet	**union**	**find**
QuickFind	$O(1)$	$O(n)$	$O(1)$
QuickFind bilanciata	$O(1)$	$O(\log n)$ amm.	$O(1)$
QuickUnion	$O(1)$	$O(1)$	$O(n)$
QuickUnion bilanciata	$O(1)$	$O(1)$	$O(\log n)$
QuickUnion bilanciata con compressione cammino	$O(1)$	$O(1)$	$O(\log^* n)$ amm.