

Dati e Algoritmi

O. Java

0.1 Iteratori

Un iteratore serve a navigare una lista

```
public interface Iterator<T> {
    T next() throws NoSuchElementException;
    boolean hasNext();
}
```

Se un ADT implementa questo interfaccia sappiamo che è possibile iterare sui suoi elementi.

```
public interface PositionalList<T> extends Container, Iterable<T>

Iterable<T> list = new PositionalList<T>();

for (T obj : list)
    System.out.println(obj);
```

Un iteratore lazy agisce direttamente sulla lista, uno di tipo snapshot agisce su una copia.
Spesso si implementa il lazy iterator e un metodo di clonazione.

1. Algoritmi

1.1 Problema computazionale

Def Un problema computazionale P è una relazione tra un insieme di istanze del problema (caso, situazioni) I e un insieme di soluzioni S . Si richiede che $\forall i \in I$ esistano ≥ 1 soluzioni $s \in S$

P. e quindi un sottinsieme di tutte le possibili combinazioni tra I e S , ovvero $I \times S$

$$P \subseteq I \times S \text{ t.c. } \forall i \in I \exists s \in S | (i, s) \in P$$

OSS

S non è necessariamente unico, un'istanza del problema può avere più soluzioni.

Esercizio 1

Somma di numeri interi

$$I = \mathbb{Z} \times \mathbb{Z}$$

$$S = \mathbb{Z}$$

$$P = \{(a, b, c) \mid a, b, c \in \mathbb{Z} \text{ e } a + b = c\} \subseteq I \times S$$

Esercizio 2

Dato un numero intero n , trovare due numeri interi la cui somma sia uguale a n

$$I = \mathbb{Z}$$

$$S = \mathbb{Z} \times \mathbb{Z}$$

$$P = \{(a, (b, c)) \mid a, b, c \in \mathbb{Z} \text{ e } a = b + c\} \subseteq I \times S$$

Questo è un esempio dove per la stessa istanza i esistono infinite soluzioni.

1.2 Algoritmo

Il fatto che sia possibile definire un problema come problema computazionale non significa che esso sia risolvibile al calcolatore.

In sostanza di un algoritmo il calcolatore non è in grado di fornire la soluzione dato un'istanza.

Tuttavia vale il contrario, se un problema è risolvibile al calcolatore allora esso ha un algoritmo e si può scrivere come problema computazionale.

Def Un algoritmo A che risolve un problema computazionale $P \subseteq I \times S$ è una procedura che dato $i \in I$ è in grado di individuare $s \in S$ tramite una sequenza finita e non ambigua di passi computazionali elementari avvenendo a $(i, s) \in P$

Def Un passo computazionale elementare è un'operazione che il calcolatore è in grado di svolgere.

Nel nostro caso il modello RAM (Random access machine) può svolgerne:

- operazioni aritmetiche e logiche elementari ($+, -, \cdot, /, \%, \dots$) (AND, NOT, ...)
- Trasferimento in lettura/scrittura da CPU a memoria
- Accesso ad un elemento di un array in tempo costante
- Esecuzione condizionale (if)
- Una funzione (necessaria solo per la ricorsione)

1.2.1 Algoritmi deterministici

Nel caso in cui un problema computazionale abbia più di una soluzione per la stessa istanza si definiscono due tipi di algoritmi.

- Deterministici: producono sempre la stessa soluzione data l'istanza.
- Non deterministici: ritornano una soluzione qualsiasi tra quelle possibili, spesso questo migliora le prestazioni.

1.3 Pseudocodice

Regole generali:

- Utilizzo di sole indentazioni
- Il tipo di una variabile solo se necessario
- uso new per creare un'istanza di un oggetto
- si può usare un for Each

Esempio

```
if T.isEmpty() return
q = new Queue()
q.enqueue(T.root())
while not q.isEmpty()
    v = q.dequeue()
    visit(v)
    for each c in T.children(v)
        q.enqueue(c)
```

1.4 Strutture dati

Una struttura dati è una collezione di oggetti condivisi di metodi per accedere e modificare la collezione.

Si distinguono due livelli di estrazione nella definizione di una struttura dati:

- Livello logico (interfaccia)
specifica l'organizzazione logica degli oggetti e i metodi di accesso/modifica
- Livello fisico (classe)
specifica il layout fisico degli oggetti e l'implementazione dei metodi
- Livello concreto (oggetto)

1.5 Correttezza di un algoritmo

Dimostrare la correttezza di un algoritmo risulta spesso essere un processo non banale.

- Può essere utile scrivere l'algoritmo in codice e poi scrivere un test che copre alcuni casi, ma questo non permette di dare una dimostrazione rigorosa se le istanze sono infinite
- Possiamo usare le dimostrazioni matematiche

1. Per esempio

2. Per controesempio

3. Per assurdo

è importante tenere presente le leggi di de Morgan: $!(A \text{ and } B) = !A \text{ or } !B$ e $!(A \text{ or } B) = !A \text{ and } !B$

4. Per induzione

Sono S_1, \dots, S_n una sequenza di enunciati, essi sono tutti veri se

- S_1 è vero

- $\forall i \in \mathbb{N} \setminus \{1\}$, preso come ipotesi S_i vero, si mostra che S_{i+1} è vero.

questo enunciato è equivalente ad un altro

- S_1 è vero, cosa base

- $\forall i \in \mathbb{N} \setminus \{1\}$, se S_1, \dots, S_i sono veri, si mostra che S_{i+1} è vero

Enunciato generale:

Si siano S_K, S_{K+1}, \dots enunciati, $K, L \in \mathbb{Z}^+$, $K \leq L$

- S_K, \dots, S_L sono veri, cosa base

- $\forall i \in \mathbb{Z}^*$ siano S_K, \dots, S_i veri allora si mostra S_{i+1} vero

5. per invarianti di ciclo

utile nel caso di cicli eseguiti K volte.

Definiamo un invarianto di ciclo, un enunciato S_i vero all'inizio del ciclo i -esimo

Per dimostrare la validità del ciclo è necessario mostrare che:

- S_i è vero prima del ciclo.

- Se S_i è vero all'inizio del ciclo i -esimo lo è anche all'inizio del successivo.

- S_i è vero al termine del ciclo.

Esempio 2

```
public static int findMax(int[] a) {  
    int max = a[0];  
    for (i = 1; /*inizio iterazione*/ i < a.length; i++)  
        if (a[i] > max)  
            max = a[i]; // nuovo candidato massimo  
    return max;  
}
```

S_i : All'inizio dell'iterazione i -esima max contiene il valore massimo dell'array $a[0 : i - 1]$

Dimostrazione

- S_i è vera all'inizio del primo ciclo perché max contiene $a[0]$ che è il massimo dell'array $a[0 : 0]$.
- Se S_i è vera all'inizio del ciclo i -esimo sarà vera anche all'inizio del ciclo successivo.
Utilizzando l'induzione matematica, prendo come ipotesi che che la variabile max contiene il valore massimo per l'array $a[0 : i - 1]$.
Durante l'esecuzione della $i + 1$ esecuzione verifico se $a[i]$ è maggiore del massimo.
Se questo è vero ho trovato un nuovo massimo per l'array $a[0 : i]$ che sposto in max .
Altrimenti max è già il massimo anche dell'array $a[0 : i]$.
- Similmente a quanto detto prima si dimostra anche che al termine nell'ultima iterazione max contiene l'elemento massimo di $a[0 : n]$

1.6 Prestazioni di un algoritmo

Si possono misurare le prestazioni di un algoritmo sotto molti punti di vista.

A noi interessano principalmente le prestazioni temporali e di occupazione di memoria

1.6.1 Dimensione di un'istanza

Def La dimensione o taglia del problema è una funzione che dato un esempio assegna un numero che fornisce una misura ragionevole in spazio e tempo all'istanza del problema.

Quindi istanze con dimensione uguale richiedono all'algoritmo tempo e spazio simili.

Queste misure ci permette inoltre di stimare le prestazioni su un'istanza mai vista.

OSS

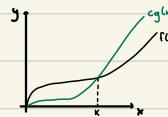
Determinare la funzione di dimensione è spesso complicato

- Solitamente la dimensione di un algoritmo che opera su una struttura lineare è uguale alla dimensione dell'array stesso
- Per strutture non lineari va valutato in base al problema

1.6.2 Notazione O-grande

$f(n) \in O(g(n))$ se $\exists k > 0, c > 0$ t.c. $f(n) \leq cg(n) \quad \forall n \geq k$

Ad un certo punto $f(n) \leq cg(n), c > 0$



OSS

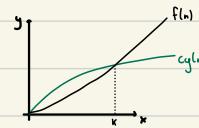
Se $f(n) \in O(n^2)$ Allora $f(n) \in O(n^3)$, $f(n) \in O(n^4)$

Sono proposizioni vere, ma sono poco utili.

1.6.3 Notazione Omega

Per "limitare" le funzioni dal basso introduciamo la notazione $\Omega(n)$

Si dice $f(n) \in \Omega(g(n))$ se $\exists k > 0, c > 0$ t.c. $f(n) \geq cg(n)$



quindi se $f(n) \in \Omega(n^2)$ allora $f(n) \in \Omega(n)$

1.6.4 Notazione Theta

Una funzione si dice $f(n) \in \Theta(g(n))$ se $f(n) \in O(g(n))$ e $f(n) \in \Omega(g(n))$

Questa notazione è spesso più utile e comoda, ma è difficile calcolarla

1.7 Analisi prestazionale

Come trovare O, Ω o Θ di un algoritmo? valutando le sue prestazioni.

Ese - ricerca sequenziale

n accessi se l'elemento non è presente

$\frac{n}{2}$ accessi in media se l'elemento è presente

Sic nel caso medio che in quello pessimo è $\Theta(n)$

Le prestazioni degli algoritmi dipendono anche dal tempo e lo spazio impiegati dal computer.

Nel nostro caso (modello RAM):

- operazioni logiche e aritmetiche in $\Theta(1)$
- ogni valore numerico occupa $\Theta(1)$
- Indirizzamento in array o in memoria in $\Theta(1)$

OSS

Gli algoritmi ricorsivi richiedono più spazio "implicito" per accomodare il call stack

Vi valutato in base alla massima profondità di ricorsione

1.7.1 Analisi Ammortizzata

Permette di trovare le prestazioni medie degli algoritmi su una grande quantità di operazioni.

ES - Inserimento in una lista:

Se la lista è piena l'operazione è $\Theta(n)$, altrimenti è $\Theta(1)$.

Se la lista viene ridimensionata secondo una costante moltiplicativa l'algoritmo è $\Theta(z)$.

Altrimenti se viene ridimensionata da una costante additiva è $\Theta(n)$.

1.7.2 Algoritmi ottimi

Per mostrare che un algoritmo è ottimo occorre trovare $h(n)$, il limite inferiore per il problema computazionale.

Poi, se l'algoritmo ha prestazioni $\Theta(h(n))$ si dice "ottimo".

1.8 Paradigma divide e conquista

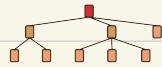
Utile per risolvere problemi complessi.

- Caso base: quando il problema è inferiore ad una soglia si risolve con una qualche strategia.
- Divisione: Si divide il problema in 2 o più partizioni.
- Ricorsione: Applica l'algoritmo alle suddivisioni.
- Conquista: Usa le soluzioni dei sottoproblemi per ottenere la soluzione del problema iniziale.

2. Alberi

Gli alberi sono un esempio di struttura non lineare, non hanno quindi un rapporto di precedente/Successivo

Oss: strutture lineari sono sottocasi di strutture non lineari.



Gli alberi si dicono strutture gerarchiche, quindi con un rapporto genitore - Figli

- Se l'albero non è vuoto uno dei suoi nodi non ha genitore (la radice)
- Ogni nodo diverso dalla radice ha un solo genitore e può avere figli
- L'insieme di nodi e rami deve essere连通 (connesso)

Def Un ramo (edge) di un albero T è la coppia (u, v) di nodi di T tale che u sia il genitore di v o viceversa

Def Nodi Figli dello stesso genitore si dicono fratelli

Def Un nodo avente almeno un figlio si dice interno

Def Un nodo senza figli si dice esterno o foglia

Def Un nodo s è un antenato del nodo n se e solo se $s \in h$ oppure s è un antenato del genitore di n

Def Un nodo d è un discendente del nodo n se e solo se n è un antenato di d

Def Sia un albero T e $v \in T$, il sottoalbero avente v come radice è l'albero definito da tutti e soli i discendenti di v

Def Un albero è ordinato se, per ogni nodo interno, esiste una relazione posizionale tra i suoi figli

Def Il grado di un nodo è il numero dei suoi figli

Def La dimensione di un albero è uguale al numero dei suoi nodi

Def Un percorso nell'albero T è una sequenza posizionale (lista) di nodi di T t.c. ogni coppia di nodi consecutivi della sequenza sia un ramo di T

Def La profondità di un nodo è il numero di antenati propri di v

Def L'altezza di un nodo è così definita:

```
function height(T, v)
    if(T.isExternal(v)) return 0;
    return 1 + max(cT.children(v).height(c));
```



Prop Le foglie hanno altezza 0

La radice ha profondità 0 e altezza massima. L'altezza della radice è anche l'altezza dell'albero

ES - foglie e figli

n_I nodi interni (ogni nodo interno ha almeno due figli)

$n_E \geq n_I + 1$ foglie

i) $h=0 \Rightarrow$ solo la radice $\Rightarrow n_I = 0$ e $n_E - 1 = n_I + 1 \quad \checkmark$

$h = h_I + 1 \Rightarrow$ per T di altezza h ho $n_{I,h}$ e $n_{E,h} \geq n_{I,h} + 1$.

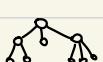


$$n_{I,h+1} = n_{I,h} + 1 \Rightarrow n_{I,h} = n_{I,h+1} - 1$$

$$\text{e } n_{E,h+1} \geq n_{E,h} + 2 \Rightarrow n_{E,h+1} \geq n_{I,h} + 3 \Rightarrow n_{E,h+1} \geq n_{I,h+1} + 2 \quad \checkmark$$

2) $n_{I,1} = 0 \Rightarrow$ solo la radice $\Rightarrow n_{E,1} = 1 \Rightarrow n_{E,1} \geq n_{I,1} + 1$

$n_I \rightarrow n_I + 1$ e quindi $n_E \rightarrow n_E - 1 + h$ ($h \geq 2$)



$$n_E \geq n_I + 1$$

$$n_E - 1 + h \geq n_I + 2 \Rightarrow n_E \geq n_I + 3 - h \quad (h \geq 2) \Rightarrow n_E \geq n_I + 1 \quad \checkmark$$

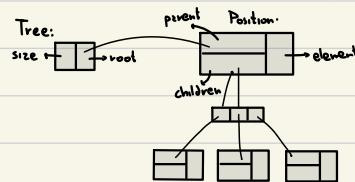
```
public interface Tree<E> extends Iterable<E> {
    private Position<E> root;
    private int numElements;

    //container
    public int size();
    public boolean isEmpty();

    //getters
    public Position<E> root();
    public Iterator<E> iterator();
    public Iterable<Position<E>> positions;

    //getters
    public boolean isInternal(Position<E> p) throws IllegalArgumentException;
    public boolean isExternal(Position<E> p) throws IllegalArgumentException;
    public boolean isRoot(Position<E> p) throws IllegalArgumentException;

    public Position<E> parent(Position<E> p) throws IllegalArgumentException;
    public Iterable<Position<E>> children(Position<E> p) throws IllegalArgumentException;
    public int numChildren(Position<E> p) throws IllegalArgumentException;
}
```



2.1 Algoritmi sugli alberi

2.1.1 profondità di un nodo

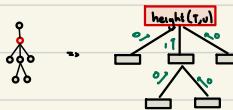
```
int depth(T, v)
    if (T.isRoot(v)) return 0
    return 1 + depth(T, T.parent(v))
```

Avvengono d iterazioni, dove d è la profondità di v (il risultato)

L'algoritmo è quindi $\Theta(d)$

2.1.2 Altezza di un nodo

```
int height(T, v)
    h = 0
    for each w of T.children(v)
        h = max{h, 1+height(T, w)}
    return h
```



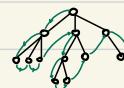
L'algoritmo viene chiamato su ogni nodo del sottoalbero con radice v , quindi l'algoritmo è $\Theta(n)$ con $n=|T_v|$

2.1.3 Attraversamenti di alberi

Si dice **attraversamento** di un albero una strategia che visita tutti i nodi una sola volta.

L'Attraversamento pre-ordine

1. Si visita la radice v
2. Per ogni figlio w di v si effettua in precedenza il sottoalbero sotteso come radice w



```
void preOrder(T, v)
```

```
// visita v
```

```
foreach w of T.children(v)
```

```
preOrder(T, w)
```

$\Theta(n+t)$ dove $n=|T_v|$ e t costo delle visite di v

L'Attraversamento post-ordine

Come il pre-ordine ma vengono prima visitati i figli di ogni nodo e poi le radici



```
void postOrder(T, v)
```

```
foreach w of T.children(v)
```

```
postOrder(T, w)
```

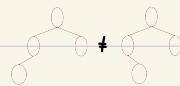
```
// visita v
```

Anche qui $\Theta(n+t)$

L'Attraversamento "per livelli," o "in ampiezz"'

2.2 Alberi binari

È un sottoalbero dell'albero "ordinato di grado 2"



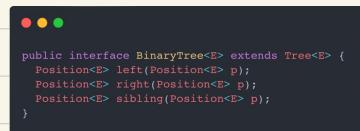
L possiamo chiamare i figli sinistro e destro (il sinistro precede il destro)

L se non ci sono nodi aventi un solo figlio l'albero binario si dice proprio o pieno, altrimenti è improprio

Def Un albero binario che abbia il massimo numero di nodi per ciascuno dei suoi livelli si dice triangolare
in questo caso: $n_f = 2^h$, $n_i = 2^h - 1$, $h = \log_2(n+1) - 1$

Def Un albero binario si dice completo se:

- Tutti i livelli j profondità minore di h hanno il massimo numero di nodi possibile
- Al livello $h-1$ tutti i nodi interni si trovano a sinistra di quelli esterni
- e solo il nodo più a destra può avere meno di due figli



2.2.1 Proprietà alberi binari

PROPRIETÀ Sia T un albero binario NON VUOTO

- $1 \leq n_f \leq 2^h$

DIM $n_f = 1 \rightarrow$ solo radice $\Rightarrow n_f = 1, 2^h = 1 \rightarrow 1 \leq n_f \leq 2^h \checkmark$

passo induzione forte, se $h > h$ vuole che $1 \leq n_f \leq 2^h$

considero S_L e S_R della radice, le loro altezze sono Lh e Rh e $Lh < h$, $Rh < h$

dato che $n_f = n_{f,S_L} + n_{f,S_R}$ $n_f \geq 1 + 0$ e $n_f \leq 2^{Lh} + 2^{Rh} \leq 2^{\max(Lh, Rh)} = 2^{\max(h, h)} = 2^h$

uno dei
due alberi può
essere vuoto

- $h \leq n_f \leq 2^h - 1$

DIM $h = 0 \rightarrow$ ho solo la radice, $n_f = 0$ e $0 \leq n_f \leq 0 \checkmark$

induzione \rightarrow Considero S_L e S_R con $Lh \leq LR$ (senza perdita di generalità $Rh \leq Lh$) dove $Lh \leq n_f \leq 2^{Lh} - 1$ e $Rh \leq n_f \leq 2^{Rh} - 1$

$$n_f = n_{f,L} + n_{f,R} + 1$$

$$n_f \leq Lh + 1 = h$$

$$n_f \leq (2^{Lh} - 1) + (2^{Rh} - 1) + 1 = 2^{Lh} + 2^{Rh} - 1 \leq 2^{\max(Lh, Rh)} - 1 = 2^h - 1 \quad \checkmark$$

$$h+1 \leq h \leq 2^{h+1}-1$$

DIM $n = n_2 + n_0 \Rightarrow h+1 \leq h \leq 2^h + 2^{h-1} - 1 = 2^{h+1} - 1 \quad \square$

$\log_2(n+1) - 1 \leq h \leq n-1$

DIM $h+1 \leq h \leq 2^{h+1}-1 \Rightarrow h \leq n \Rightarrow h \leq h-1$
 $\Rightarrow n \leq 2^{h-1} \Rightarrow n+1 \leq 2^{h+1} \Rightarrow h \geq \log_2(n+1) - 1 \quad \left\{ \Rightarrow \log_2(n+1) - 1 \leq h \leq n-1 \right\} \quad \square$

PROPRIETÀ Sia T un albero binario proprio e non vuoto.

$h+1 \leq h \leq 2^h$

$h \leq n_2 \leq 2^{h-1}$

$2h+1 \leq h \leq 2^h - 1$

$\log_2(n+1) - 1 \leq h \leq \lfloor \log_2(n) \rfloor$

$n_0 = h_2 + 1$

PROPRIETÀ Sia T un albero binario completo non vuoto

- h dimensione minima quando ha 1 nodo in h () $n_{min} = 1 + 2 + 4 + \dots + 2^{h-1} + 1 = 2^{h-1} + 1 = 2^h$

- h dimensione massima se è triangolare () $n_{max} = 1 + 2 + 4 + \dots + 2^{h-1} + 2^h = 2^{h+1} - 1$

quindi $2^h \leq h \leq 2^{h+1}-1 \Rightarrow \log_2(n+1) - 1 \leq h \leq \log_2(n)$

oss tra $\log_2(n+1) - 1$ e $\log_2(n)$ la distanza è di 1. Quando dato n si può trovare h

oss è conveniente implementare i binary complete tree con un array (complessità $\Theta(n)$ per add e remove)

2.2.2 Attraversamento simmetrico

Gli alberi binari ereditano gli stessi attraversamenti degli alberi, aggiungiamo inoltre la visita inorder

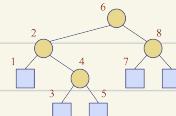
Si basa sull'idea di: prima il figlio sx, poi il genitore e infine il figlio dx

inorderTraversal(T, v)

```

sx = T.left(v)
dx = T.right(v)
if (sx) inorderTraversal(T, sx)
//visit v
if (dx) inorderTraversal(T, dx)

```



Viene eseguito in $\Theta(n)$.

2.2.3 Albero binario in un vettore

Per realizzare un albero binario si usa solitamente una struttura a nodi concatenati.

Ma anche una lista con indici, è sufficiente definire $p(v)$ che associa ogni nodo ad un indice.

int $p(T, v)$

```
if T.isRoot(v) return 1  
u = T.parent(v)  
if (T.right(u) == v) return 2*p(T, u)  
if (T.left(u) == v) return 2*p(T, u) + 1
```



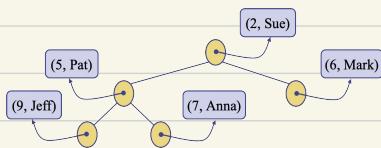
Facendo così si sprecano delle posizioni nell'array, in particolare si usa $O(2^h)$ (che nel caso peggiore è $O(2^n)$), al posto di $O(n)$.

Hc comunque delle applicazioni in quanto le prestazioni temporali uguali.

3. Heap

Un Heap è un albero binario completo T tale che

- memorizza in ciascun nodo $v \in T$ una coppia $(key, value)$, con key appartenente ad un insieme totalmente ordinato.
- $\forall v \in T, v \neq T.root() \Rightarrow key(T.parent(v)) \leq key(v)$



oss Esistono minHeap e maxHeap in base alla "direzione" della relazione.

oss Nella radice è presente la chiave minima (minHeap) o massima (maxHeap).

3.1 Heap Sort

Detto un insieme di n elementi su cui è definita una relazione d'ordine totale \leq

memorizzato su una struttura posizionata S , voglio ordinarlo su una struttura posizionata T

- Estraiamo gli elementi da S in ordine crescente e inseriamoli in un heap inizialmente vuoto

prestazioni $O(n \log n)$ $\left(O(\log_1 + \log_2 + \dots + \log(n)) \cdot \log(n) \right)$

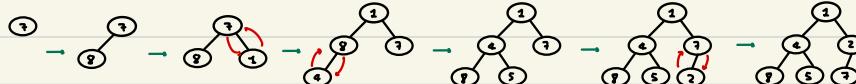
- Svuotiamo l'heap in T usando removeMin

prestazioni $O(n \log n)$

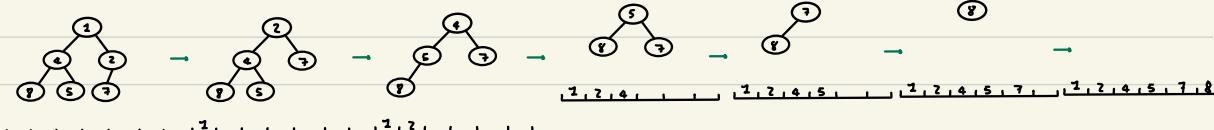
ES

7, 8, 1, 4, 5, 2,

Inserisco nell'Heap:



Rimuovo dall'Heap



È possibile sviluppare l'Heap sort "sul posto" senza ricorrere ad un heap esterno

ES

7, 8, 2, 4, 5, 2, 6, 3,

passo zero: trovo il minimo e lo porto in posizione 0

in nero: parte da non toccare

1, 8, 7, 4, 5, 2, 6, 3,

in verde: zona da leggere

1, 7, 8, 4, 5, 2, 6, 3,

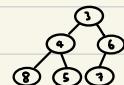
in rosso: heap

[...]

1, 2, 4, 3, 8, 5, 7, 6,

Rimozione dell'Heap e bubble down

1, 3, 4, 6, 8, 5, 7, 2,



2, 4, 5, 6, 8, 7, 3, 2,

[...]

1, 8, 7, 6, 5, 4, 3, 2,

Invertito la parte verde

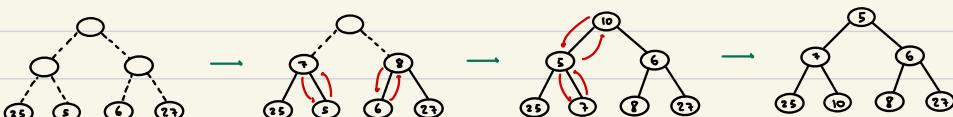
1, 2, 3, 4, 5, 6, 7, 8,

3.2 Bottom-Up Heap Construction

Dati n elementi vogliamo costruire in $\Theta(n)$ un heap che li contenga.

Esempio

Inseriamo $[23, 5, 6, 27, 7, 8, 10]$



4. Code Prioritarie

4.1 Strutture dati posizionali

Def Una relazione d'ordine definita in S è $f: S \times S \rightarrow \{\text{vero}, \text{falso}\}$ che abbia le proprietà:

$\forall a, b \in S, a \leq b \Rightarrow f(a, b) = \text{vero}$ e $a \geq b \Rightarrow f(a, b) = \text{falso}$

- riflessiva: $\forall a \in S, a \leq a$

- transitiva: $\forall a, b, c \in S, a \leq b, b \leq c \Rightarrow a \leq c$

- antisimmetrica: $\forall a, b \in S, a \leq b, b \leq a \Rightarrow a = b$

Inoltre la relazione d'ordine si dice totale se vale:

- totale: $\forall a, b \in S, a \leq b$ oppure $b \leq a$ (o entrambe) (qualsiasi elemento deve essere confrontabile)

Def Un insieme S in cui sia definita la relazione d'ordine totale f si dice totalmente ordinato secondo f

L'gli elementi di S si dicono confrontabili mediante f

L'h $X \subseteq S$ esiste minimo (m) e massimo (M)

dove m t.c. $\forall x \in X, m \leq x$ ($f(m, x) = \text{true}$)

dove M t.c. $\forall x \in X, x \leq M$ ($f(x, M) = \text{true}$)

Oss In generale nel corso assumiamo che i confronti siano $\Theta(1)$

4.2 Interfaccia code prioritarie

Def La code prioritaria è un contenitore che memorizza elementi ognuno dei quali è associato ad un livello di priorità

Si possono individuare subito due tipi di PQ, minPQ e maxPQ
L'estrazione prima la priorità massima
L'estrazione prima la priorità minima

oss I livelli di priorità devono appartenere ad un insieme totalmente ordinato

oss Possono esistere più elementi con stessa priorità, è importante gestire il caso in modo coerente

4.2.1 Implementazione

Ogni elemento ha:

- ↳ Un valore (value)
- ↳ Le priorità ad esso associate (chiave/key)

```
public interface Entry<K,V> {  
    public K getKey();  
    public V getValue();  
    public void setValue(V newVal);  
}
```

```
public interface PriorityQueue<K,V> extends Container {  
    void insert(K key, V value) throws InvalidKeyException;  
    Entry<K,V> getMin() throws EmptyPriorityQueueException;  
    Entry<K,V> removeMin() throws EmptyPriorityQueueException;  
}
```

Per implementare la relazione d'ordine totale tra le chiavi si può usare:

- java.lang.Comparable<T>

permette di usare .compareTo()

Ha poche classi litici perché non possiamo sovrscrivere l'ordinamento delle classi

es. String usa l'ordinamento lessicografico, non posso usare lo lunghezza

- java.util.Comparator<T>

ci permette di descrivere un comparatore custom

```
class ByLengthStringComp implements java.util.Comparator<String> {  
    public int compare(String s1, String s2) {  
        if (s1 == null || s2 == null) throw new ClassCastException();  
        if (s1.length() == s2.length()) return s1.compareTo(s2);  
        return s1.length() - s2.length();  
    }  
}
```

```
public class PQWithComparator<K,V> implements PriorityQueue<K,V> {  
    private Comparator<K> comp;  
    public PQWithComparator(Comparator<K> c) { comp = c; }  
}
```

Questo rende PQ modulare

4.3 Code prioritarie su liste

Si possono usare due strategie per implementare PQ su una lista

- Liste non ordinate

- insert in $\Theta(1)$

• min in $\Theta(n)$, bisogna confrontare tutta la lista

• removeMin in $\underline{\Theta(n)} + \underline{\Theta(1)}$
 _{min} _{rimozione}

- Liste ordinate

- insert in $\Theta(\log n)$ con una lista concatenata

• min in $\Theta(1)$

• removeMin in $\underline{\Theta(1)} + \underline{\Theta(1)}$
 _{min} _{rimozione}

Ora si può ottenere il massimo dei due paradigmi inserendo gli elementi ordinando e poi rimuovendo (caso non così reale)

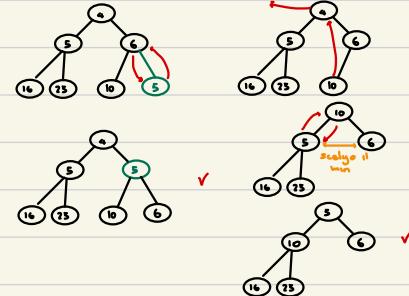
4.4 Code prioritarie su Heap

Risulta efficiente implementare una priority queue con un Heap implementato con un array

• size e isEmpty in $\Theta(1)$

• min in $\Theta(1)$

• insert e removeMin sono più complessi, $\Theta(\log n)$



5. Mappe

Un contenitore che contiene valori associati ad una chiave (unico)

Def Le chiavi devono essere uniche, ma non devono appartenere ad un insieme totalmente ordinato.

Devono essere confrontabili tra di loro (equals)

Le chiavi possono essere uguali ai valori o possono essere associate da un'applicazione (funzione)

Le mappe è in grado di ritornare il valore (unico) dato la chiave oppure "null" se non si trova la chiave

Di conseguenza i valori delle mappe NON possono essere "null" altrimenti non si può distinguere da un valore non trovato

Oss Si può ritornare Entry(key,value). Entry(key,null), null per ovviere.

La chiave può essere null, deve però essere unica.

```
● ● ●  
public interface Map<K,V> extends Container {  
    int size(); // numero di coppie  
    V put(K key, V val); // insert or replace  
    V get(K key);  
    V remove(K key);  
    Iterable<K> keys();  
    Iterable<V> values();  
}
```

Le mappe si può implementare con una lista

get, put, remove sono $O(n)$ (è causa della ricerca sequenziale)

5.1 Mappa ordinata su array

Se le chiavi sono ordinabili si può definire la SortedMap.

Possiamo ora usare il binary search sull'array ordinato

get ha prestazioni $O(\log n)$

put e remove sono $O(n)$ perché vengono spostati gli elementi dell'array

5.2 Mappa su binary search tree

Un Albero di ricerca binaria è un albero binario proprio, non vuoto.

Ogni nodo interno contiene un elemento appartenente ad un insieme totalmente ordinato

I nodi esterni contengono "null" (simplifica gli algoritmi)

Per ogni nodo interno 1. Gli elementi del sottoalbero sinistro sono $\leq x(v)$

2. Gli elementi del sottoalbero destro sono $\geq x(v)$

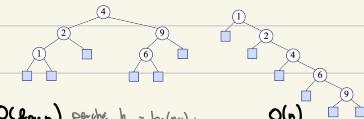


```
● ● ●  
function SearchInBSTsubTree(T, k, v)  
    if T.isExternal(v) return null // non trovato  
    else if k == key(v) return value(v) // trovato  
    else if k < key(v) return SearchInBSTsubTree(T, k, T.left(v))  
    else return SearchInBSTsubTree(T, k, T.right(v))
```

Memorizzare: dati in un albero di ricerca binaria ha le stesse prestazioni spazziali sintattiche delle liste $\Theta(n)$

Il metodo get si ottiene chiamando `SearchInBSTsubTree(T, key, T.root())` le prestazioni però sono $O(h)$, altezza dell'albero.
ed è possibile che $h = n$

È quindi importante "costruire" bene l'albero



$O(\log n)$ perché $h_m \approx \log(n+1)$

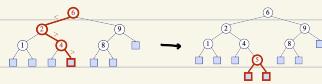
$O(n)$

Per ottenere l'albero binario nelle forme "migliori":

1. Inserisco le chiavi in un insieme ordinato A
2. Scelgo l'elemento centrale che sarà la radice
3. Itero con gli elementi a sinistra per costruire il semialbero sinistro
4. Itero con gli elementi a destra per costruire il semialbero destro

Questo processo ha prestazioni di $O(n)$ per aggiungere al BST già ordinato (avrebbe $O(n\log n)$ per riordinare una lista ordinata)

può avere prestazioni migliori:

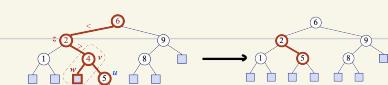


In modo simile possiamo ottimizzare remove

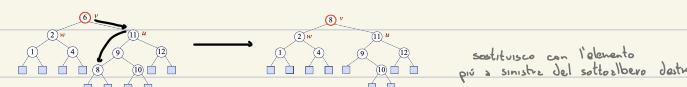
- Se il nodo non ha figli non "null"



- Se il nodo ha un solo figlio non "null"



- Se il nodo ha 2 figli non "null"



Quindi abbiamo ottenuto 3 algoritmi che sono $\Theta(\log n)$ nel caso medio, ma $O(n)$ nel caso pessimo

5.3 Mappa su Albero AVL

Un nodo interno v si dice bilanciato in altezza se le altezze dei suoi figli differiscono tra loro al massimo di un'unità

Un albero binario con tutti i nodi bilanciati si dice bilanciato o albero AVL

PROP Un albero è BST se e solo se tutti i suoi sottoalberi sono BST

PROP Un albero è bilanciato se e solo se tutti i suoi sottoalberi sono bilanciati

PROP Un albero è AVL se e solo se tutti i suoi sottoalberi sono AVL

PROP Un albero AVL di dimensione n ha $h = O(\log n)$

DIM In modo intuitivo: ad ogni "step" di altezza si tolgoano in media metà degli elementi

5.3.1 Algoritmi di inserimento e rimozione

- Il get ora è $O(\log n)$ sempre, non solo in casi specifici

- Il put esegue in $O(\log n)$ l'inserimento come sull'albero BST (insensio il nodo w)

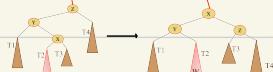
questo però in generale non ritorna un albero AVL

1. Verifico se l'albero AVL si è bilanciato, per fare ciò in $O(h)$ guardiamo agli antenati del nodo w appena inserito e verifichiamo in $O(1)$ che il nodo sia bilanciato
Ogni nodo dell'albero salvo la sua altezza, se la differenza tra l'altezza di due fratelli è > 1 allora il nodo è sbilanciato

2. evidenzio i nodi z, y, x (si dimostra che esistono almeno i nodi z e y non può essere $x = z$)

3. carico l'elemento intermedio tra x, y, z (può essere $x = y$ o $y = z$)

4. Se x è l'elemento intermedio



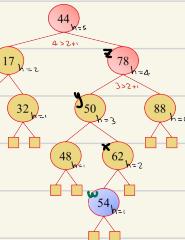
- Il remove si sviluppa in modo simile

1. rimuovo l'elemento e chiamo w il suo genitore

(se il genitore non esiste il nodo rimosso era la radice, in questo caso l'albero rimane sempre AVL)

2. eseguo le stesse operazioni su w per bilanciare l'albero

Quindi get, put e remove hanno prestazioni $O(\log n)$ ✓



6. Ordinamento

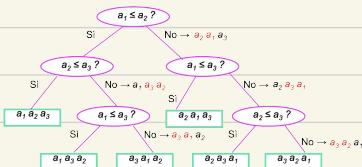
Si può dimostrare che qualunque algoritmo di ordinamento che traggia informazioni da confronti tra gli elementi da ordinare è $\Omega(n \log n)$ nel caso pessimo.

Dobbiamo quindi lavorare per migliorare le prestazioni nel caso migliore o medio.

DIM

Possiamo rappresentare un qualsiasi algoritmo di ordinamento posso usare un albero binario decisionale.

data la lista $S = (a_1, a_2, a_3)$ la lista ordinata si ottiene seguendo l'albero:



Il numero di confronti per arrivare alla soluzione peggiore è uguale all'altezza dell'albero

L'albero è un binario proprio: $h = \log_2 n! \Rightarrow h_{\min} = \lceil \log_2 n! \rceil$

Sappiamo che esistono almeno $n!$ permutazioni della lista iniziale.

Inoltre $\log_2(n!) \in \Theta(n \log n)$ quindi questo è il miglior algoritmo di ordinamento mediante confronti.

6.1 Bucket Sort

Un algoritmo di ordinamento senza confronti.

Opera su una lista S di n elementi, ognuno di tipo $[0, N-1]$ (n e N non hanno correlazioni)

- Crea un array auxiliaro di dimensione N in $\Theta(N)$

- Estraiamo gli elementi da S e facciamo + sulle celle dell'array creato corrispondente in $\Theta(n)$

- Ricostruiamo S ordinato da A in $\Theta(n+N)$

$$S = (5, 2, 5) \rightarrow S = (-, 2, 5) \rightarrow S = (-, -, 5) \rightarrow S = (-, -, -) \rightarrow S = (2, 5, 5)$$
$$A = (-, -, -, -, -) \quad A = (-, -, -, -, 1) \quad A = (-, 1, -, -, 1) \quad A = (-, 1, -, -, 2)$$

Quindi il tempo totale è $\Theta(n+N)$ e spazio aggiuntivo di $\Theta(N)$

Nel caso in cui $N = O(n)$ e quindi N aumenta all'aumentare di n , vero all'atto pratico, il bucket sort è $\Theta(n)$

6.2 Merge sort

Scritto da John Von Neumann, 1945

- Caso base: se $\dim S \leq 2 \Rightarrow S$ è ordinata
- Divisione: se $\dim S \geq 2 \Rightarrow$ divide S in 2 sequenze S_1 e S_2
- Ricorsione su S_1 e S_2
- Conquista: fonde S_1 e S_2 in una sequenza ordinata $\rightarrow S$ che era rimasta vuota.
richiede $\Theta(n)$
$$\begin{aligned} &\text{while } !S1.isEmpty() \& !S2.isEmpty() \\ &\quad \text{while } !S1.isEmpty() \\ &\quad \quad \text{while } !S2.isEmpty() \end{aligned}$$

Serve uno spazio aggiuntivo di $\Theta(n)$

Si $T(n)$ il tempo per ordinare n elementi. $T(n) = \begin{cases} b & \text{se } n=2 \\ 2T\left(\frac{n}{2}\right) + cn & \text{se } n \geq 2 \text{ (dove conquer)} \end{cases}$

$$T(n) = 2T\left(\frac{n}{2}\right) + cn = 2T\left(2T\left(\frac{n}{2}\right) + cn\right) + cn = 4T\left(\frac{n}{2}\right) + 2cn + cn = 2^k T\left(\frac{n}{2^k}\right) + cn$$

$$\text{quindi } T(n) = 2^k T\left(\frac{n}{2^k}\right) + cn$$

$$\text{se } k = \log_2 n \quad T(n) = n T\left(\frac{n}{n}\right) + cn = n T(1) + cn \in \Theta(n \log n)$$

6.2.1 Merge sort iterativo

- Ogni elemento è da considerarsi un array di lunghezza 1
- Ciascuna coppia di elementi consecutivi viene fusa in un array di lunghezza 2
- ⋮

6.3 Quick sort

- Caso base: se $\dim S \leq 2 \Rightarrow S$ è ordinata
- Divisione: scelto $x \in S$ (pivot), estratti tutti gli elementi di S inserendoli in 3 sequenze
 - L contiene gli elementi $< x$
 - E contiene gli elementi $= x$
 - G contiene gli elementi $> x$
- Ricorsione: ordina L e G (E è già ordinata)

- Conquista viene eseguita su S : prima L , poi E , infine G

Nel caso pessimo di scelta del pivot le prestazioni sono $\Theta(n^2)$

Usando invece l'elemento mediano come pivot si ottiene $\Theta(n \log n)$

Anche usando un pivot casuale si può mostrare che le prestazioni sono "molto probabilmente" $O(n \log n)$

6.3.1 Quick sort con divisione sul posto

È possibile operare sul posto, senza creare le sottosequenze L, E, G.

In questo modo lo spazio zygomatico è $\Theta(1)$ e non $\Theta(n)$.

- Scegliamo il pivot x e lo scambiamo con l'ultimo elemento
 - Usiamo due indici, left (può il primo elemento e aumenta) e right (può il penultimo elemento e diminuisce)

L' spostiamo left finché gli elementi esaminati sono < x

L' spostiamo night finché gli elementi sono >x

L'Quando ci fermiamo

se `left < right` → scrabbiamo gli elementi puntati

6.4 Quicksort vs Mergesort

- Mergesort \rightarrow $2n$ successi divisioni
 \downarrow
 $2n$ successi fusioni
 \downarrow
 $n - 2n$ successi per verificare l'ordine tra i due array ordinati (primo ciclo while)

quindi ci sono $5n$ excessi & $6n$ per livello $\Rightarrow 5n \log_2 h$ & excessi & $6n \log_2 h$

- Quicksort sul posto \rightarrow n confronti
 $\rightarrow O(n \log n)$ scambi

quindi $n \log_2 n \leq 2n \log_2 n$

7. Grafi

Un tipo di dato estratto in grado di rappresentare relazioni esistenti tra coppie di oggetti.

Un grafo $G = (V, E)$ è costituito da un insieme V di vertici o nodi

e di un insieme E di connessioni, tra coppie di vertici dette ram.

Due vertici si dicono adiacenti se esiste un ramo che li collega e sono gli estremi del ramo

ESEMPI

Rappresentare i legami di simpatia in un gruppo (facebook).

Rappresentare aeroporti e i voli che li collegano

Def - grafi orientati

Rappresentiamo i rami come $e = (u, v) \in E$ dove $u, v \in V$

Se la direzione del ramo conta il ramo si dice orientato altrimenti è non orientato

Se tutti i rami sono orientati il grafo si dice orientato

Se tutti i rami non sono orientati il grafo si dice non orientato

Altrettanto il grafo è misto

Def - rami incidenti

Un ramo si dice incidente con un vertice se il vertice è uno dei suoi estremi.

Il numero di rami incidenti con un determinato nodo v è detto il grado del nodo ($\deg(v)$)

Per i rami orientati vanno distinti rami incidenti uscenti ($\text{outdeg}(v)$) e rami incidenti entranti ($\text{indeg}(v)$)

Def - rami particolari

È possibile calcolare le stesse coppie di vertici con più rami. In questo caso si parla di rami multipli o paralleli

È anche possibile creare rami con un singolo vertice (auto-anelli)



Grafi senza rami multipli o auto-anelli si dice semplice

```
public interface Edge<T> extends Position<T> {}  
public interface Vertex<T> extends Position<T> {}  
public interface Graph<V,E>  
{  
    int numEdges();  
    int numVertices();  
  
    Iterable<Edge<E>> edges();  
    Iterable<Vertex<V>> vertices();  
  
    Iterable<Edge<E>> incidentEdges(Vertex<V> v);  
    Vertex<V> opposite(Vertex<V> v, Edge<E> e);  
    Vertex<V>[] endVertices(Edge<E> e)  
    boolean areAdjacent(Vertex<V> u, Vertex<V> v)  
  
    V replace(Vertex<V> v, V o);  
    E replace(Edge<E> e, E o);  
    Vertex<V> insertVertex(V o); // inserito isolato  
    V removeVertex(Vertex<V> v); // rimuove i rami incidenti (si può lanciare un'eccezione)  
    Edge<E> insertEdge(Vertex<V> u, Vertex<V> v, E o);  
    E removeEdge(Edge<E> e);  
}
```

Implementazione di un grafo

Def - percorso in un grafo

In un grafo un percorso è una sequenza di vertici e archi alternati:

- La sequenza inizia con un vertice
- Ciascun arco incidente al vertice che lo precede è quello che lo succede
- Nessun arco può comparire in due posizioni consecutive.

Prop Un percorso è semplice se tutti i suoi vertici sono distinti

Prop Un percorso è orientato se tutti i suoi archi sono orientati e vengono strutturati secondo la loro direzione.

Un ciclo è un percorso il cui vertice iniziale è uguale a quello finale

Def - sottografi

Dato un grafo $G = (V_G, E_G)$ il grafo $H = (V_H, E_H)$ è un sottografo di G se $V_H \subseteq V_G$, $E_H \subseteq E_G$

Tutti i vertici che sono estremi di un arco $\in E_H$ devono essere in V_H

Def Un grafo G è连通的 se esiste un percorso tra due qualsiasi vertici.

Oss Se un grafo è orientato è sufficiente un percorso, anche non orientato.

Def Un grafo G non è连通的, i suoi sottografi连通的 connnessi di dimensioni massime si dicono componenti connnessi di G (o isole)

Def - grafo ed albero

Un grafo privo di cicli è un albero

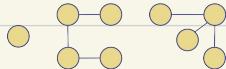
Diverso dagli alberi visti in precedenza, manca la definizione di una radice

Prop Un albero contiene il minimo numero di archi che rende connesso un grafo avente quell'insieme di vertici.

Def - foreste

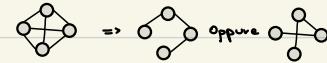
Un grafo privo di cicli è una foresta, i suoi componenti sono alberi

Prop Un albero è una foresta e una foresta connnessa è un albero



Def - spanning tree

Uno spanning tree di un grafo è un sottografo con gli stessi vertici e il numero minimo di archi che preservano la connessione del grafo.



7.1 Proprietà dei grafi

Sia $G = (V, E)$

1. Se $|E|=m$ allora $\sum_{v \in V} \deg(v) = 2m$ (ogni ramo ha 2 nodi)

2. Se G grafo orientato e $|E|=m$

$$\sum_{v \in V} \text{indeg}(v) = \sum_{v \in V} \text{outdeg}(v) = m$$

3. Se G grafo semplice e $|V|=n$, $|E|=m$

Se G non orientabile $m \leq \frac{n(n-1)}{2}$

Se G orientabile $m \leq n(n-1)$

4. Se G semplice non orientabile e $|V|=n$, $|E|=m$

Se G 连通的 $m \geq n-1$

Se G foresta $m \leq n-1$

Se G albero $m = n-1$

7.2 Implementazione di grafi

I grafi hanno molte realizzazioni, ma in pratica sono semplici variazioni sul tema di una delle tre categorie:

- Edge list
- Adjacency list
- Adjacency matrix

In tutti e tre i casi `numEdges`, `numVertices`, `opposite`, `endVertices`, `replace`, `insertEdge` e `removeEdge` sono $\Theta(1)$

Ci sono invece differenze sulle prestazioni di `incidentEdges`, `areAdjacent`, `insertVertex`, `removeVertex`

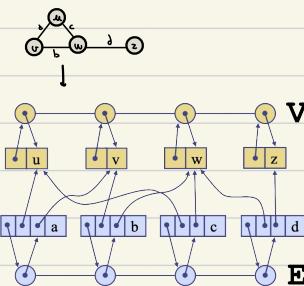
7.2.1 Edge list

Due liste, una per i rami E e una per i nodi V . (poniamo liste doppiamente concatenate)

Ogni elemento della lista punta a viene puntato dal nodo del grafo.

Inoltre i rami del grafo hanno due puntatori per indicare i suoi estremi.

È quindi facile andare dai rami ai nodi, è però difficile il contrario, dai nodi ai rami.

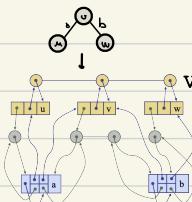


7.2.2 Adjacency list

Ogni vertice punta ad una lista che contiene riferimenti ai voci incidenti.

Ogni voci punta alle liste di incidenti dove è presente.

Questo risolve il problema di edge list nel passare dai nodi ai voci.

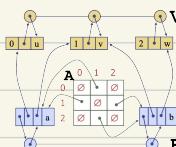


7.2.3 Adjacency list

Aggiunge una matrice A che rappresenta coppie di vertici adiacenti.

Venne aggiunto un indice i,j ogni vertice che viene usato come indice di riga e colonna della matrice.

Ogni cella di A contiene il riferimento al voci che collega i due vertici corrispondenti ai suoi indici.



7.2.4 Confronto

Sia un grafo con n voci e m voci.

	Edge List	Adjacent List	Adjacent Matrix
incidentEdges	$\Theta(m)$	$\Theta(\deg v)$	$\Theta(n)$
areAdjacent	$\Theta(m)$	$\Theta(\min\{\deg v, \deg w\})$	$\Theta(1)$
insertVertex	$\Theta(1)$	$\Theta(1)$	$\Theta(n^2)$
removeVertex	$\Theta(n)$	$\Theta(\deg v)$	$\Theta(n^2)$
Memory	$\Theta(nm)$	$\Theta(n+m)$	$\Theta(n^2)$

7.3 Attraversamento di Grafi

Un semplicissimo attraversamento si può fare attraversando prima i voci e poi i voci.

Questo attraversamento non gode però di particolari proprietà topologiche.

Esistono altri due fondamentali algoritmi di attraversamento.

- Depth-first search (DFS)

Parte da un voci, si allontana il più possibile e poi ritorna sui suoi passi.

- breath-first search (BFS)

Esplore i voci adiacenti e poi continua "a macchia d'olio".

7.4 Attraversamento depth-first (DFS)

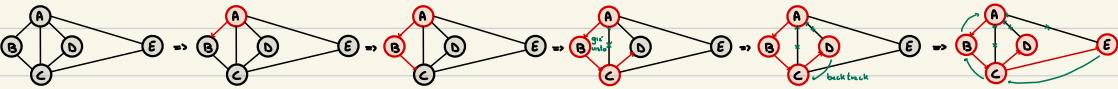
Svolte le posizioni già attraversate per poter tornare indietro da un vicolo cieco

Per fare ciò a servono etichette temporanee e vertici e rami, aggiungiamo alle loro interfacce setLabel e getLabel

All'inizio dell'attraversamento impostiamo tutti i rami e i vertici come "Unexplored" e scegliamo il vertice iniziale

```
DFS(G, i)
DFS(G, s):
    s.setLabel(VISITED)
    for each e of G.IncidentEdges(s)
        e.setLabel(VISITED) // visita il ramo
        w = G.opposite(s, e) // w è il vertice opposto al ramo
        if (w.getLabel() == UNEXPLORED) DFS(G, w)
```

Implementato l'algoritmo in modo ricorsivo ci risparmiamo l'implementazione del backtracking



Per visitare grafî non connessi

```
DFSearch(G)
for each v in G.vertices():
    if (v.getLabel() == UNEXPLORED) DFS(G, v)
```

Questo algoritmo può essere usato per individuare uno spanning tree

Quando troviamo dei rami che partono da nodi già visitati impostiamo la Label a BACK, questi sono i rami in eccesso

Allo stesso modo si trovano anche i possibili cicli del grafo

Un grafo è quindi una foresta se e solo se non ha rami di tipo BACK

7.4.1 Otttenere il percorso

L'algoritmo si può usare per trovare un percorso tra due nodi.

Semplicemente quando si arriva ad un nodo e gli si assegna l'etichetta VISITED si vede se si tratta del nodo finale, in quel caso è necessario terminare la ricorsione.

Per trovare il percorso possiamo usare uno stack per depurare il percorso da vicoli ciechi.

```
S = new Stack()
pathDFS(G, v, z, S)

pathDFS(G, v, z, S)
    v.setLabel(VISITED)
    S.push(v)
    if v == z return true
    for each e of G.incidentEdges(v)
        if e.getLabel() == UNEXPLORED
            w = G.opposite(v, e)
            if w.getLabel() == UNEXPLORED
                e.setLabel(DISCOVERY)
                S.push(e)
                if pathDFS(G, w, z, S) return true
                S.pop(e)
            else
                e.setLabel(BACK)
        S.pop(v)
    return false
```

7.4.2 Prestazioni DFS

La fase di inizializzazione a UNEXPLORED è $\Theta(nm)$

Per ogni iterazione chiamo

incidentEdges che è $\Theta(\deg(v)) \rightarrow \Theta(zm)$

opposite $\Theta(1) \rightarrow \Theta(m)$

quindi le prestazioni ottime sono $\Theta(nm)$

7.5 Attraversamento breadth-first BFS

Si allontana "a macchia d'olio" dal vertice iniziale.

Assegna ad ogni vertice un "livello" di distanza dal vertice iniziale.

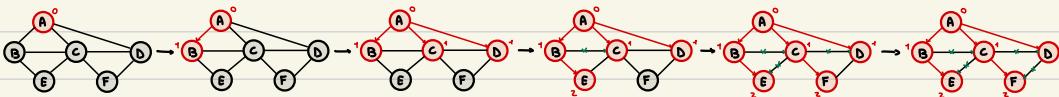
Algoritmo:

- Impostiamo tutti i nodi come UNEXPLORED
- Visitiamo il vertice iniziale e lo poniamo al livello 0
- Esaminiamo i vertici incidenti:
 - se porta ad un nodo nuovo \rightarrow DISCOVERY \rightarrow visitiamo il nodo e assegnamo livello +1 \rightarrow ricorsione
 - se porta ad un nodo già visto \rightarrow CROSS

```

BFS(G, s) // non ricorsivo
L0 = new empty sequence
L0.addLast(s)
s.setLabel(VISITED)
i = 0
while !Li.isEmpty()
    Li+1 = new empty sequence
    for each v of Li
        for each e of G.incidentEdges(v)
            if e.getLabel() == UNEXPLORED
                w = G.opposite(v,e)
                if w.getLabel() == UNEXPLORED
                    e.setLabel(DISCOVERY)
                    w.setLabel(VISITED)
                    Li+1.addLast(w)
                else
                    e.setLabel(CROSS)
    i = i + 1

```



BFS ha prestazioni ottime sull'adjacency list, $\Theta(n+m)$

I nodi DISCOVERY costituiscono una spanning forest di G

BFS trova sempre il percorso minimo di nodi DISCOVERY dal punto iniziale a quello finale.

In questo caso basta assegnare un parent ad ogni nodo. (per il nodo iniziale è NULL)
e poi seguire i parent del nodo a quello iniziale

7.6 Algoritmo di Dijkstra

Abbiamo già visto come usare un algoritmo BFS per trovare il numero minimo di nodi per passare da un nodo all'altro.

Nelle applicazioni reali spesso i nodi non sono tutti uguali: es. i navigatori

L'algoritmo di Dijkstra ci permette di trovare i percorsi minimi da una sorgente verso tutti i vertici.

Vogliamo fornire un grafo e ottenere l'albero dei percorsi minimi dal punto iniziale

Algoritmo:

- Creiamo e facciamo crescere una "nuvolà" attorno al vertice iniziale (i)

- La nuvolà si espanderà verso il vertice esterno con distanze minime da i

Quando un vertice entra nella nuvolà viene etichettato con la sua distanza da i

Dijkstra risponde alla domanda "come si calcola la distanza da i dei vertici esterni alle nuvole?"

L'algoritmo usa una serie di approssimazioni per eccesso.

Iniziando con $d(i) = 0$, $d(u) = +\infty$ Vuxi e l'invariante "la lunghezza del percorso minimo tra u e i è maggiore di $d(u)$ "

Si inizia con le nuvole vuote, e cui si inserisce i, unico elemento con $d(i) = 0$

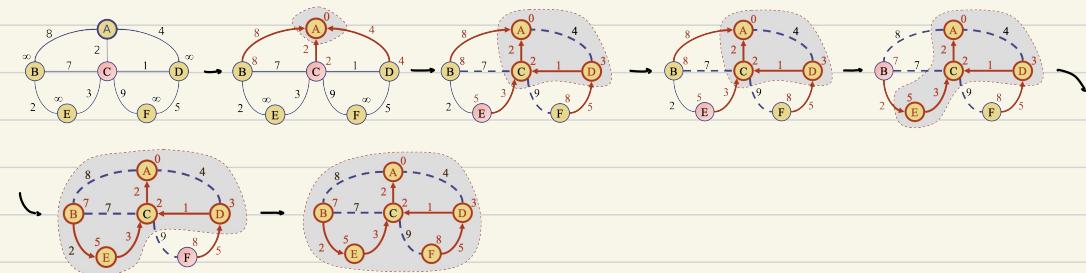
Poi si aggiornano i valori dei nodi adiacenti ad i in base ai pesi dei cammini che conducono ad essi e aggiungono il minore

In particolare $d(u) = d(i) + \text{"peso del cammino"}$

Finché la nuvola non copre tutto il grafo

Si può dimostrare che quando un nodo entra nella nuvola la sua etichetta di distanza da i non è solo la distanza da i, ma è proprio uguale

Durante l'esecuzione di Dijkstra si può assegnare un parent ad ogni nodo, il genitore di un nodo è il nodo che permette di raggiungere i più velocemente



7.6.1 Prestazioni dell'algoritmo di Dijkstra

- Inserisce tutti i nodi in un contenitore "esterno alle nuvole" in $\Theta(n)$

oss il contenitore "dentro la nuvola" non è necessario, non viene mai usato dall'algoritmo.

Come contenitore è comodo un Adaptable PQ

- Si effettuano n inserimenti in $O(n \log n)$ (oppure $\Theta(n)$ costruendo direttamente la struttura dati, tanto $d(u) = +\infty$ Vuxi)
- n estrazioni dell'elemento minimo $O(n \log n)$
- $O(m)$ aggiornamenti di etichetta (per rilassamento) $O(m \log n)$

In totale $O([n+m] \log n)$

```

Dijkstra(G, s)
    q = HeapAdaptableMinPQ( )
    s.setLocationAware(q.insert(0, s))
    s.setLabel(0)
    s.setParent(NULL) // per i percorsi, non "Dijkstra puro"
    for v of G.vertices()
        if v != s
            v.setLocationAware(q.insert(+∞, v))
            v.setLabel(+∞) // inutile inizializzare parent, verrà sovrascritto
    while !q.isEmpty()
        entry = q.removeMin()
        label = entry.getKey()
        vertex = entry.getValue()
        if label == +∞ break // G non è connesso, algoritmo terminato nel componente connesso di s
        for edge of G.incidentEdges(vertex)
            z = G.opposite(vertex, edge)
            newLabel = label + edge.getLabel()
            if newLabel < z.getLabel() // rilassamento
                z.setLabel(newLabel)
                z.setParent(edge) // per i percorsi, non "Dijkstra puro"
                q.replaceKey(z.getLocationAware(), newLabel)

```

8. Tabella

Una mappa con chiavi numeriche intere comprese tra $[0, N-1]$

Operazioni di get, put, remove sono $\Theta(1)$

I metodi keys, values sono $\Theta(N)$ al posto di $\Theta(n)$

0	valore 0
1	null
2	null
3	valore 1
4	valore 2
5	null
6	valore 3
7	null

Le tabelle non utilizzano la memoria in modo efficiente, occupa $\Theta(N)$, non dipende da n

Sì definisce il fattore di occupazione $\lambda = \frac{n}{N}$

Se le tabelle ha dimensione variabile l'operazione di inserimento può richiedere $\Theta(N)$ e può accadere ad ogni inserimento

8.1 Tabella hash

Se una mappa von chiavi numeri interi definiamo una funzione di hash

dominio: le chiavi

codominio: gli indici dei valori

In generale la funzione non è biunivoca, due indici possono avere lo stesso codice di hash queste sono dette collisioni

Le funzioni di hash che minimizzano le collisioni sono dette ottime

Ne sono esempio i cast (float \rightarrow int bit a bit)

Se devo comprimere 64 bit di informazione in 32 bit avrò delle collisioni, qui ci sono molti modi risolutivi.

La funzione di compressione ha il compito di ridurre il codice hash ad un insieme finito.

Si dice ottima una funzione che non avverte le collisioni.

Una buona soluzione è l'hash $h(k) \% N$ (migliore se N è un numero primo "distanza" da una potenza di 2)

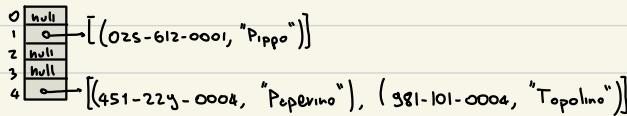
Sia h la funzione di hash e A l'array (inizializzato con $A[m] = \text{null}$ $\forall m$)

- In assenza di collisioni (k, v) si inserisce $A[h(k)] = v$
- Se ci sono possibili collisioni usiamo un array di valori

ES

Sono $(025-612-0001, \text{"Pippo")}, (451-224-0004, \text{"Paperino")}, (981-101-0004, \text{"Topolino")}$

e h t.c. $h(025-612-0001) = 1$ $h(451-224-0004) = 4$ $h(981-101-0004) = 4$



Quindi ora le operazioni diventano:

Calcolo di $h(\text{key}) \rightarrow$ accesso al bucket \rightarrow ricerca di key nel bucket
 $\Theta(1)$ $\Theta(1)$ $O(n)$ nel caso pessimo

Se la funzione di compressione è buona le prestazioni diventano $O(n/N) = O(\lambda)$

get put remove keys/values

Mappa in lista $O(n)$ $O(n)$ $O(n)$ $\Theta(n)$

Mappa in lista ordinata $O(\log n)$ $O(n)$ $O(n)$ $\Theta(n)$

Mappa in AVL $O(\log n)$ $O(\log n)$ $O(\log n)$ $\Theta(\log n)$

Mappa in una tabella hash $O(\lambda)$ $O(\lambda)$ $O(\lambda)$ $\Theta(n\lambda)$

8.2 Rehashing

Quando n supera certe dimensioni si può effettuare il rehashing.

- Si crea una tabella di dimensioni maggiori (e quindi si riduce λ)

• Si trasformano le coppie (key, value) calcolando la nuova $h(\text{key})$
L'operazione è $\Theta(N+n)$ ma vede l'analisi sommatozata