

Appunti di Dati e algoritmi

Giacomo Simonetto

Primo semetre 2024-25

Sommario

Appunti del corso di Dati e algoritmi della facoltà di Ingegneria Informatica dell'Università di Padova.

Indice

| | | |
|-----------|--|-----------|
| 1 | Introduzione all'algoritmica | 4 |
| 1.1 | Progettazione di un algoritmo | 4 |
| 1.2 | Problema computazionale | 4 |
| 1.3 | Algoritmo | 4 |
| 1.4 | Complessità - efficienza di un algoritmo | 4 |
| 1.5 | Correttezza e invariante - efficacia di un algoritmo | 5 |
| 1.6 | Algoritmi ricorsivi | 6 |
| 2 | Alberi | 7 |
| 2.1 | Introduzione | 7 |
| 2.2 | Algoritmi base: profondità e altezza di un nodo | 8 |
| 2.3 | Visite: preorder e postorder | 9 |
| 3 | Alberi binari | 10 |
| 3.1 | Introduzione | 10 |
| 3.2 | Visite: inorder | 10 |
| 3.3 | Parser Tree | 11 |
| 3.4 | Implementazione in Java | 12 |
| 4 | Priority Queue | 13 |
| 4.1 | Introduzione | 13 |
| 4.2 | Implementazioni e relativa complessità | 13 |
| 5 | Heap | 14 |
| 5.1 | Introduzione | 14 |
| 5.2 | Algoritmi base: getmin, insert e remove | 15 |
| 5.3 | Costruzione di un heap inplace | 16 |
| 5.4 | HeapSort | 17 |
| 6 | Mappe | 18 |
| 6.1 | Introduzione | 18 |
| 6.2 | Implementazioni e relativa complessità | 18 |
| 7 | Hash Table | 19 |
| 7.1 | Introduzione | 19 |
| 7.2 | Algoritmi base: get, put, remove | 20 |
| 7.3 | Load Factor e complessità | 21 |
| 7.4 | Rehashing | 21 |
| 8 | Alberi binari di ricerca | 22 |
| 8.1 | Alberi binari di ricerca in generale | 22 |
| 8.2 | Multi-Way Search Tree - MWST | 24 |
| 8.3 | (2,4)-Tree | 25 |
| 8.4 | Red-Black Tree | 27 |
| 9 | Multimappe | 28 |
| 9.1 | Introduzione | 28 |
| 9.2 | Implementazione e relativa complessità | 28 |
| 10 | Grafi | 29 |
| 10.1 | Introduzione | 29 |
| 10.2 | Breadth First Search - BFS | 31 |
| 10.3 | Depth First Search - DFS | 34 |
| 10.4 | Confronto tra BFS e DFS | 35 |
| 10.5 | Cammini minimi su grafi pesati - Dijkstra | 36 |
| 10.6 | Grafi diretti | 38 |
| 10.7 | Grafi diretti senza cicli - DAG | 39 |

| | |
|---|-----------|
| 11 Ordinamenti | 40 |
| 11.1 Descrizione del problema | 40 |
| 11.2 MergeSort | 40 |
| 11.3 QuickSort | 41 |
| 11.4 Randomized QuickSort | 41 |
| 11.5 InsertionSort | 42 |
| 11.6 BucketSort | 42 |
| 11.7 RadixSort | 43 |

1 Introduzione all'algoritmica

1.1 Progettazione di un algoritmo

- specifica del problema computazionale
- progetto dell'algoritmo (pseudocodice)
- analisi dell'efficacia e efficienza dell'algoritmo (correttezza e complessità)
- codifica programma (linguaggio di programmazione)

1.2 Problema computazionale

Un problema computazionale è composto da tre componenti:

- I l'insieme delle istanze (input)
- S l'insieme delle soluzioni (output)
- $\pi \subseteq I \times S$ una relazione che associa ad ogni elemento di I un corrispettivo elemento di S .

Se esistono più soluzioni per una generica istanza, l'algoritmo ne calcola una arbitraria

1.3 Algoritmo

Un algoritmo è una procedura computazionale che trasforma un dato input in un output eseguendo una sequenza di passi elementari. Sfrutta il modello di calcolo *RAM* (Random Access Machine) con le seguenti caratteristiche:

- è simile al modello di Von Neumann
- in memoria sono contenuti i dati di input, output e dati intermedi
- la CPU utilizza operazioni elementari per risolvere il problema

Il problema che risolve l'algoritmo è un problema computazionale, ovvero trova le soluzioni abbinate ad una determinata istanza.

Pseudocodice

Per descrivere le azioni di un algoritmo si utilizza lo pseudocodice nella seguente struttura:

0: **Algoritmo:** MYALGORITHM(parametri)
 Input: descrizione dell'input
 Output: descrizione dell'output

1: descrizione dell'algoritmo in pseudocodice
2: ...
3: ...

1.4 Complessità - efficienza di un algoritmo

Taglia di un'istanza

La taglia di un'istanza è l'indice di misura della dimensione dei dati di input, scelto in base all'algoritmo.

Complessità temporale

La complessità temporale indica una stima asintotica del numero di operazioni elementari eseguite dall'algoritmo per una certa istanza. Suggerisce una stima del tempo di esecuzione di un algoritmo per una certa istanza ed è utilizzata per confrontare algoritmi diversi in base alla loro efficienza computazionale.

Caratteristiche

- la complessità si esprime in maniera generale in funzione della taglia dell'istanza
- è indipendente dall'implementazione e dal linguaggio usato
- non richiede di conoscere il tempo esatto di esecuzione, per cui non richiede l'implementazione dell'algoritmo in un certo linguaggio di programmazione, ma basta lo pseudocodice

Notazione

Per indicare una complessità asintotica, si utilizzano gli ordini di grandezza e le notazioni Ω , Θ , O .

- $f(n) \in O(g(n)) \Rightarrow \exists c > 0, n_0 \geq 1$ t.c. $f(n) \leq c g(n) \quad \forall n \geq n_0$
- $f(n) \in \Omega(g(n)) \Rightarrow \exists c > 0, n_0 \geq 1$ t.c. $f(n) \geq c g(n) \quad \forall n \geq n_0$
- $f(n) \in \Theta(g(n)) \Rightarrow f(n) \in O(g(n))$ e $f(n) \in \Omega(g(n))$
- $O(1) \leq O(\log n) \leq O(n) \leq O(n \log n) \leq O(n^2) \leq O(n^3) \leq O(2^n) \leq \dots$

Calcolo

Per trovare la complessità asintotica al caso pessimo si effettuano le seguenti analisi:

1. si cerca un upper-bound attraverso l'analisi del caso pessimo (in assoluto) e lo si esprime come $O(f(n))$
2. si cerca un lower-bound attraverso l'analisi di una istanza "cattiva" e lo si esprime come $\Omega(g(n))$
3. se $f(n) = g(n)$ e la complessità dell'algoritmo $\in O(f(n))$ e $\in \Omega(g(n))$, allora si conclude che l'algoritmo ha complessità al caso pessimo $\in \Theta(f(n))$

Osservazioni

- un algoritmo ha complessità ottima se è $O(\log n)$, buona se $O(n)$ o $O(n \log n)$, discreta se $O(n^2)$, pessima se $O(2^n)$ o $O(n^n)$
- alcune volte il valore n_0 è per istanze molto grandi e può capitare che un algoritmo di complessità $\Theta(n^3)$ sia più efficiente di uno di complessità $\Theta(\log n)$
- alcune volte, il caso pessimo si verifica per istanze patologiche che si verificano estremamente raramente, per cui si ricorre al calcolo della complessità al caso medio

1.5 Correttezza e invariante - efficacia di un algoritmo

Criteri di correttezza

Un algoritmo è corretto se:

- termina in un numero finito di passi
- restituisce una soluzione valida per il problema computazionale

Invariante

L'invariante è una proprietà costante espressa in funzione delle variabili dell'algoritmo e descrive lo stato ottenuto in ogni segmento dell'algoritmo stesso. Per avere senso, deve essere funzionale alla correttezza dell'algoritmo. L'invariante è spesso usato per verificare la correttezza di un ciclo o di una serie di chiamate ricorsive: lo si esprime in funzione delle variabili del ciclo o dei parametri della chiamata ricorsiva e si verifica che esso sia soddisfatto alla fine di ogni iterazione o chiamata ricorsiva.

Verifica della correttezza in un ciclo

Per verificare la correttezza di un ciclo si utilizza un processo induttivo:

1. si determina l'invariante per il ciclo
2. si dimostra che l'invariante vale per l'iterazione -1 (ovvero prima dell'inizio del ciclo)
3. si verifica che se vale per una certa iterazione i , allora vale per l'iterazione $i + 1$
4. si verifica che l'invariante alla fine del ciclo combacia con la condizione di correttezza della soluzione

1.6 Algoritmi ricorsivi

Definizione

Un algoritmo si dice ricorsivo se richiama se stesso su istanze di minore dimensione fino ad arrivare ad un caso base in cui la soluzione è immediata.

Esecuzione e stack

Ad ogni nuova chiamata ricorsiva, viene aggiunto un nuovo record di attivazione per tale chiamata nello stack. Nell'RDA sono contenute tutte le variabili e i riferimenti agli oggetti memorizzati nell'heap. Alla chiusura del metodo, si elimina l'RDA di tale metodo e si apre quello sottostante.

Esecuzione e recursion tree

All'esecuzione di un algoritmo ricorsivo si associa un albero della ricorsione con le seguenti caratteristiche:

- ogni nodo corrisponde ad una chiamata ricorsiva
- la radice corrisponde alla prima chiamata ricorsiva
- i figli di un nodo corrispondono alle chiamate ricorsive effettuate nell'esecuzione della chiamata associata al nodo padre
- le foglie corrispondono ai casi base

Differenza tra algoritmo iterativo e algoritmo ricorsivo

- un algoritmo iterativo, per definizione, deve contenere cicli al suo interno
- un algoritmo ricorsivo, per definizione, deve contenere almeno una chiamata a se stesso
- iterativo e ricorsivo sono mutualmente esclusivi: non esiste un algoritmo ricorsivo e iterativo
- un algoritmo ricorsivo può avere o non avere cicli, ma un algoritmo iterativo non ha chiamate ricorsive perché altrimenti sarebbe un algoritmo ricorsivo

Complessità di algoritmi ricorsivi

Per determinare la complessità di un algoritmo ricorsivo si può procedere in vari modi:

- contare il numero di operazioni per livello e moltiplicarlo per il numero di livelli dell'albero della ricorsione, come nel caso di mergesort
- contare tutte le operazioni eseguite ad ogni chiamata ricorsiva e sommarne i valori

Correttezza di algoritmi ricorsivi

Per determinare la correttezza di un algoritmo ricorsivo si procede con una dimostrazione per induzione:

1. si dimostra che l'algoritmo risolve correttamente i casi base
2. si verifica che se l'algoritmo è corretto per una determinata istanza, allora vale anche un'istanza di taglia superiore

2 Alberi

2.1 Introduzione

Definizione di albero

Un albero radicato è una collezione di nodi vuota o con le seguenti proprietà:

- $\exists r$ nodo detto radice
- se un generico nodo $n \neq r$, allora deve avere un unico padre
- da ogni nodo, risalendo di padre in padre si arriva ad r

Altre definizioni

- **antenato:**
un nodo x è antenato di y se $x = y$ o se x è antenato del padre di y
- **discendente:**
un nodo x è discendente di y se y è antenato di x
- **nodo interno:**
un nodo x è interno se ha dei figli
- **nodo esterno o foglia:**
un nodo x è esterno o una foglia se non ha figli
- **sottoalbero con radice v :**
indicato con T_v è un albero formato dai discendenti di v
- **albero ordinato:**
un albero si dice ordinato se è presente un ordinamento lineare tra i figli di un nodo
- **albero radicato:**
un albero si dice radicato se è composto da un nodo radice r che ha come figli, altri alberi radicati nei nodi figli di r
- **profondità di un nodo:**
la profondità di un nodo è la distanza tra il nodo e la radice
- **altezza di un nodo:**
l'altezza di un nodo è la distanza tra il nodo e la foglia più distante
- **livello:**
il livello i di un albero è l'insieme di nodi con profondità i
- **altezza di un albero:**
l'altezza di un albero è la massima profondità delle foglie

2.2 Algoritmi base: profondità e altezza di un nodo

Profondità di un nodo - ricorsiva

- L'algoritmo calcola la distanza tra il nodo v e la radice, risalendo di padre in padre
- La complessità dell'algoritmo è $\Theta(n)$, più precisamente $\Theta(d_v)$, con d_v profondità del nodo v .

```
0: Algoritmo: DEPTH_RIC( $v$ )  
   Input:       $v$  nodo dell'albero  $T$   
   Output:     profondità di  $v$   
  
1:   if (T.isRoot( $v$ )) then  
2:     return 0  
3:   else  
4:     return 1 + depth(T.parent( $v$ ))
```

Profondità di un nodo - iterativa

- L'algoritmo calcola la distanza tra il nodo v e la radice, risalendo di padre in padre
- La complessità dell'algoritmo è $\Theta(n)$, più precisamente $\Theta(d_v)$, con d_v profondità del nodo v .

```
0: Algoritmo: DEPTH_ITER( $v$ )  
   Input:       $v$  nodo dell'albero  $T$   
   Output:     profondità di  $v$   
  
1:    $d \leftarrow 0$   
2:    $u \leftarrow v$   
3:   while (!T.isRoot( $u$ )) do  
4:      $u \leftarrow T.parent(u)$   
5:      $d \leftarrow d + 1$   
6:   return  $d$ 
```

Altezza di un nodo

- L'algoritmo calcola prima l'altezza dei discendenti di v e prosegue verso l'alto fino a v
- La complessità dell'algoritmo è $\Theta(n)$, più precisamente $\Theta(n_v)$, con $n_v = \#$ nodi del sottoalbero T_v .

```
0: Algoritmo: HEIGHT( $v$ )  
   Input:       $v$  nodo dell'albero  $T$   
   Output:     altezza di  $v$   
  
1:    $h \leftarrow 0$   
2:    $w \leftarrow v$   
3:   for all  $w$  in T.children( $v$ ) do  
4:      $h \leftarrow \max\{h, 1 + \text{height}(w)\}$   
5:   return  $d$ 
```

2.3 Visite: preorder e postorder

Visita in preorder

- L'algoritmo prevede di visitare prima il nodo v e successivamente i figli
 - La complessità dell'algoritmo è $\Theta\left(n + \sum_{v \in T} t_v\right)$, con t_v costo della visita di un nodo
-

0: **Algoritmo:** PREORDER(v)

Input: v nodo dell'albero T

Output: /

```
1:   visita  $v$ 
2:   for all  $w$  in  $T.children(v)$  do
3:     preorder( $w$ )
```

Visita in postorder

- L'algoritmo prevede di visitare prima i figli e successivamente il nodo v
 - La complessità dell'algoritmo è $\Theta\left(n + \sum_{v \in T} t_v\right)$, con t_v costo della visita di un nodo
-

0: **Algoritmo:** POSTORDER(v)

Input: v nodo dell'albero T

Output: /

```
1:   for all  $w$  in  $T.children(v)$  do
2:     postorder( $w$ )
3:   visita  $v$ 
```

3 Alberi binari

3.1 Introduzione

Definizione di albero binario

Un albero binario è un albero ordinato in cui ogni nodo ha al più due figli. Ogni nodo è etichettato come figlio sinistro o figlio destro, in base alla posizione di ordinamento.

Altre definizioni

- **albero binario proprio:**
un albero binario si dice proprio se ogni nodo ha esattamente due figli
- **albero binario proprio estremo:**
un albero binario proprio si dice estremo se nessun figlio destro (o sinistro) ha discendenti, ovvero se è molto sbilanciato verso sinistra (o verso destra)
- **albero binario proprio perfettamente bilanciato:**
un albero binario proprio si dice perfettamente bilanciato se tutte le foglie sono allo stesso livello
- **albero binario completo:**
un albero binario con i livelli si dice completo se tutti i livelli da 0 a $i-1$ sono completi (contengono 2^i nodi) e l'ultimo livello i è popolato a partire da sinistra (approfondito nella sezione heap).

Proprietà di un albero binario proprio

Dati: n = numero di nodi in T h = altezza di T
 m = numero di foglie in T $n - m$ = numero di nodi interni in T

Un albero binario proprio ha le seguenti proprietà:

| binario generico | binario estremo | binario perf. bil. |
|---|---------------------|-------------------------|
| $m = n - m + 1$ | $m = n - m + 1$ | $m = n - m + 1$ |
| $h + 1 \leq m \leq 2^h$ | $h + 1 = m$ | $m = 2^h$ |
| $h \leq n - m \leq 2^h - 1$ | $h = n - m$ | $n - m = 2^h - 1$ |
| $2h + 1 \leq n \leq 2^{h+1} - 1$ | $2h + 1 = n$ | $n = 2^{h+1} - 1$ |
| $\log_2(n + 1) - 1 \leq h \leq \frac{n-1}{2}$ | $h = \frac{n-1}{2}$ | $h = \log_2(n + 1) - 1$ |

Dall'ultima proprietà si osserva che la miglior stima dell'altezza h di un generico albero binario è $O(n)$ e non è sempre detto che sia $O(\log n)$.

3.2 Visite: inorder

Visita inorder

- L'algoritmo prevede di visitare prima il figlio sinistro, poi il nodo stesso e infine il figlio destro.
- La complessità dell'algoritmo è $\Theta\left(n + \sum_{v \in T} t_v\right)$, con t_v costo della visita di un nodo

0: **Algoritmo:** INORDER(v)

Input: v nodo dell'albero T

Output: /

```
1:  if (T.left( $v$ )  $\neq$  null) then
2:    inorder(T.left( $v$ ))
3:  visita  $v$ 
4:  if (T.right( $v$ )  $\neq$  null) then
5:    inorder(T.right( $v$ ))
```

3.3 Parser Tree

Definizione

Il Parse Tree T associato ad una espressione aritmetica E (con operatori solo binari) è un albero binario proprio in cui i nodi foglia contengono le variabili/costanti di E e i nodi interni contengono gli operatori di E , in modo tale che:

- se $E = a$, con a costante/variabile, allora T è costituito da un'unica foglia contenente a
- se $E = E_1 op E_2$, la radice T contiene l'operatore op e ha come sottoalbero sinistro E_1 e come sottoalbero destro E_2

Notazione infissa

- L'algoritmo sfrutta la visita inorder degli alberi binari.
- La complessità dell'algoritmo è $\Theta(n)$, in quanto t_v costo della visita di un nodo è $\Theta(1)$.

0: **Algoritmo:** INFIX(T, v, L)
Input: Parse Tree $T, v \in T$, lista L
Output: E_v in notazione infissa nella lista L

```
1:  if (T.isExternal(v)) then
2:      L.addLast(v.getElement())
3:  else
4:      L.addLast("(")
5:      infix(T, T.left(v), L)
6:      L.addLast(v.getElement())
7:      infix(T, T.right(v), L)
8:      L.addLast(")")
```

Notazione postfissa

- L'algoritmo sfrutta la visita in postorder applicata agli alberi binari.
- La complessità dell'algoritmo è $\Theta(n)$, in quanto t_v costo della visita di un nodo è $\Theta(1)$.

0: **Algoritmo:** POSTFIX(T, v, L)
Input: Parse Tree $T, v \in T$, lista L
Output: E_v in notazione postfissa nella lista L

```
1:  if (T.isExternal(v)) then
2:      L.addLast(v.getElement())
3:  else
4:      postfix(T, T.left(v), L)
5:      postfix(T, T.right(v), L)
6:      L.addLast(v.getElement())
```

3.4 Implementazione in Java

Interfaccia Iterator

```
1 // cursore che permette di muoversi tra gli elementi di una collezione
2 public interface Iterator<E> {
3     /** Returns true if the scan of the collection is not over */
4     boolean hasNext();
5     /** Returns the next element in the collection */
6     E next();
7 }
```

Interfaccia Iterable

```
1 // struttura dati iterabile
2 public interface Iterable<E> {
3     /** Returns an iterator of the collection */
4     Iterator<E> iterator()
5 }
```

Interfaccia Tree

```
1 // Albero
2 public interface Tree<E> extends Iterable {
3     /** Returns the number of positions in the tree */
4     int size();
5     /** Returns true if the tree contains no positions */
6     boolean isEmpty();
7     /** Returns the Position of the root (or null if empty)*/
8     Position<E> root();
9     /** Returns the Position of p's parent (or null if p is the root) */
10    Position<E> parent(Position<E> p);
11    /** Returns an iterable containing p's children */
12    Iterable<Position<E>> children(Position<E> p);
13    /** Returns the number of children of p */
14    int numChildren(Position<E> p);
15    /** Returns true if p is internal */
16    boolean isInternal(Position<E> p);
17    /** Returns true if p is external */
18    boolean isExternal(Position<E> p);
19    /** Returns true if p is root */
20    boolean isRoot(Position<E> p);
21    /** Returns an iterator to all element in the tree */
22    Iterator<E> iterator();
23    /** Returns an iterable containing all positions in the tree */
24    Iterable<Position<E>> positions();
25 }
```

Interfaccia BinaryTree

```
1 // Albero binario
2 public interface BinaryTree<E> extends Tree<E> {
3     /** Returns the Position of p's left child(or null if empty)*/
4     public Position<E> left(Position<E> p);
5     /** Returns the Position of p's right child(or null if empty)*/
6     public Position<E> right(Position<E> p);
7     /** Returns the Position of p's sibling (or null p is an only child)*/
8     public Position<E> sibling(Position<E> p);
9 }
```

4 Priority Queue

4.1 Introduzione

Definizione

Una priority queue è una collezione di entry in cui le chiavi rappresentano una priorità e provengono da un insieme totalmente ordinato. Minore è il valore della chiave, maggiore è la priorità.

Metodi e interfaccia priority queue

```
1 // Priority queue
2 public interface PriorityQueue<k,v> {
3     int size();
4     boolean isEmpty();
5     /** Inserts and returns a new entry (key,value) */
6     Entry<K,V> insert(K key, V value);
7     /** Returns an entry with min key, without removing it */
8     Entry<K,V> min();
9     /** Returns and removes an entry with min key */
10    Entry<K,V> removeMin();
11 }
```

4.2 Implementazioni e relativa complessità

| metodo | unordered list | ordered list | heap |
|-----------|----------------|--------------|--------------|
| min | $O(n)$ | $O(1)$ | $O(1)$ |
| removeMin | $O(n)$ | $O(1)$ | $O(\log(n))$ |
| insert | $O(1)$ | $O(n)$ | $O(\log(n))$ |

Nell'implementazione con la **lista non ordinata**, l'inserimento impiega tempo costante, ma per l'accesso alla chiave minima e di conseguenza la relativa rimozione è necessario applicare un algoritmo di ricerca che impiega tempo lineare.

Nell'implementazione con la **lista ordinata**, l'accesso alla chiave minima e la relativa rimozione impiegano tempo costante, ma per l'inserimento è necessario applicare l'algoritmo di **insertionsort** che impiega tempo lineare.

Nell'implementazione con lo **heap**, l'inserimento e la rimozione impiegano un tempo logaritmico, per come definiti i metodi di inserimento e rimozione nell'heap (up heap bubbling e down heap bubbling), mentre l'accesso alla chiave minima impiega tempo costante. Quest'ultima implementazione offre una migliore efficienza computazionale e anche una migliore efficienza spaziale (se l'heap è implementato efficientemente con un array).

5 Heap

5.1 Introduzione

Definizione

Lo heap (mucchio in italiano) è un albero binario completo in cui i nodi contengono delle entry $\langle k, v \rangle$ e soddisfano una proprietà chiamata heap order property: siano $\langle k_2, v \rangle$ e $\langle k_3, v \rangle$ figli di $\langle k_1, v \rangle$, allora vale $k_1 \leq \min\{k_2, k_3\}$.

Proprietà

- un albero binario completo ha altezza $h = \lfloor \log n \rfloor$
- esiste ed è unico, l'albero binario completo di n entry
- le chiavi lungo un cammino dalla radice alle foglie formano una sequenza non decrescente
- per qualsiasi figlio u di v vale $v.getKey() \leq u.getKey()$
- la radice contiene la entry con chiave minima
- se le entry sono tutte distinte, la entry con chiave massima si trova in una foglia

Level numbering e rappresentazione tramite array

La rappresentazione Level numbering permette di assegnare un indice a ciascun nodo dello heap secondo le seguenti regole:

1. la radice ha indice 0 e si trova in $P[0]$
2. i figli di $P[i]$ hanno indici $2i + 1$ e $2i + 2$, si trovano in $P[2i + 1]$ e $P[2i + 2]$
3. il padre di $p[i]$ ha indice $\lfloor (i - 1)/2 \rfloor$ e si trova in posizione $P[\lfloor (i - 1)/2 \rfloor]$

Si osserva che i nodi di ciascun livello hanno indici crescenti da sinistra verso destra. In questo modo è possibile rappresentare uno heap su un array, ottenuto giustapponendo i nodi di ogni livello mantenendo le posizioni reciproche dei nodi da sinistra verso destra. Tale rappresentazione è space efficient perché c'è corrispondenza 1:1 tra un albero binario completo e un array.

Nodo Last

Per praticità, il nodo più a destra dell'ultimo livello è chiamato nodo Last e nella rappresentazione tramite array e level numbering si trova nell'ultima cella dell'array con indice $\lfloor n - 1 \rfloor$.

5.2 Algoritmi base: getmin, insert e remove

Accesso alla entry con chiave minima

- L'algoritmo restituisce la entry contenuta nella radice dell'heap
- La complessità dell'algoritmo è $O(1)$

0: **Algoritmo:** GETMIN()
 Input: riferimento implicito all'heap rappresentato tramite array
 Output: entry con chiave minima

1: return P[0]

Inserimento di una nuova entry

- L'algoritmo inserisce una nuova entry attraverso uno up-heap bubbling.
- La complessità dell'algoritmo è $\Theta(\log n)$, proporzionale all'altezza dell'heap.

0: **Algoritmo:** INSERT(k, v)
 Input: entry da inserire e riferimento implicito all'heap $P[]$
 Output: /

1: $e \leftarrow (k, v)$
2: $P[+ + last] \leftarrow e$
3: $i \leftarrow last$
4: **while** ($i > 0$ and $P[\lfloor (i-1)/2 \rfloor].getKey() \geq P[i].getKey()$) **do** ▷ up-heap bubbling
5: swap($P[i], P[\lfloor (i-1)/2 \rfloor]$)
6: $i \leftarrow \lfloor (i-1)/2 \rfloor$
7: return e

Rimozione della entry con chiave minima

- L'algoritmo rimuove la radice dell'heap attraverso un down-heap bubbling.
- La complessità dell'algoritmo è $\Theta(\log n)$, proporzionale all'altezza dell'heap.

0: **Algoritmo:** REMOVEMIN()
 Input: riferimento implicito all'heap $P[]$
 Output: entry con chiave minima rimossa

1: $minEntry \leftarrow P[0]$
2: $P[0] \leftarrow P[last - -]$
3: $i \leftarrow 0$
4: $j \leftarrow \text{indexMinChild}(P, i)$
5: **while** ($j \neq \text{null}$ and $P[i].getKey() > P[j].getKey()$) **do** ▷ down-heap bubbling
6: swap($P[i], P[j]$)
7: $i \leftarrow j$
8: $j \leftarrow \text{indexMinChild}(P, i)$
9: return $minEntry$

Rimozione di una generica entry

- L'algoritmo rimuove una generica entry dell'heap e ripristino la heap order property con un down-heap bubbling e un up-heap bubbling.
- La complessità dell'algoritmo è $\Theta(\log n)$, proporzionale all'altezza dell'heap.

```

0: Algoritmo: REMOVE( $l$ )
   Input:    indice della entry da rimuovere e riferimento implicito all'heap  $P[]$ 
   Output:  entry rimossa

1:   $retValue \leftarrow P[l]$ 
2:   $P[l] \leftarrow P[last - -]$ 
3:   $i \leftarrow l$ 
4:   $j \leftarrow \text{indexMinChild}(P, i)$ 
5:  while ( $j \neq \text{null}$  and  $P[i].\text{getKey}() > P[j].\text{getKey}()$ ) do            $\triangleright$  down-heap bubbling
6:       $\text{swap}(P[i], P[j])$ 
7:       $i \leftarrow j$ 
8:       $j \leftarrow \text{indexMinChild}(P, i)$ 
9:  while ( $i > 0$  and  $P[\lfloor (i-1)/2 \rfloor].\text{getKey}() \geq P[i].\text{getKey}()$ ) do    $\triangleright$  up-heap bubbling
10:      $\text{swap}(P[i], P[\lfloor (i-1)/2 \rfloor])$ 
11:      $i \leftarrow \lfloor (i-1)/2 \rfloor$ 
12:  return  $retValue$ 

```

5.3 Costruzione di un heap inplace

Costruzione di un heap da un array, con approccio Top-Down

- L'algoritmo inserisce le chiavi una ad una, incrementando j e costruisce l'heap con un up-heap bubbling per ogni chiave inserita
- La complessità dell'algoritmo è $\Theta(n \log n)$, ovvero la ripetizione di n volte dell'up-heap bubbling con complessità $\Theta(\log n)$

```

0: Algoritmo: MAKEHEAP_TOPDOWN()
   Input:    riferimento implicito all'heap  $P[]$ 
   Output:   entry con chiave minima rimossa

1:   for  $j \leftarrow 1$  to  $n - 1$  do
2:        $i \leftarrow j$ 
3:       // up-heap bubbling
4:       while ( $i > 0$  and  $P[\lfloor (i - 1)/2 \rfloor].\text{getKey}() \geq P[i].\text{getKey}()$ ) do
5:           swap( $P[i]$ ,  $P[\lfloor (i - 1)/2 \rfloor]$ )
6:            $i \leftarrow \lfloor (i - 1)/2 \rfloor$ 

```


Costruzione di un heap da un array, con approccio Bottom-Up

- L'algoritmo costruisce tanti sottoheap che si sviluppano dal livello i fino alla base. Ad ogni iterazione del for i sottoheap si uniscono a due a due con l'inserimento della entry dal livello $i - 1$ con un down-heap bubbling.
- La complessità dell'algoritmo è $\Theta(n)$, più vantaggioso rispetto al precedente approccio top-down.

0: **Algoritmo:** MAKEHEAP_BOTTOMUP()

Input: riferimento implicito all'heap $P[]$

Output: entry con chiave minima rimossa

```
1:   for  $j \leftarrow \lfloor (n-2)/2 \rfloor$  to 0 do           ▷  $P[\lfloor (n-2)/2 \rfloor]$  nodo più a dx nel penultimo livello
2:      $i \leftarrow j$ 
3:      $k \leftarrow \text{indexMinChild}(P, i)$ 
4:     // down-heap bubbling
5:     while ( $k \neq \text{null}$  and  $P[i].\text{getKey}() > P[k].\text{getKey}()$ ) do
6:       swap( $P[i], P[k]$ )
7:        $i \leftarrow k$ 
8:        $k \leftarrow \text{indexMinChild}(P, i)$ 
```

5.4 HeapSort

Ordinamento di un array attraverso una priority queue implementata come heap su array

- L'algoritmo si sviluppa in due fasi:
 - fase1: creazione di un max-heap inplace con approccio bottom-up, con complessità $\Theta(n)$
 - fase2: estrazione delle chiavi inplace con leggere modifiche, con complessità $\Theta(n \log n)$
- La complessità dell'algoritmo è $\Theta(n \log n)$,

0: **Algoritmo:** HEAPSORT(P)

Input: array P con chiavi da riordinare

Output: array P con chiavi ordinate

```
1:   for  $j \leftarrow \lfloor (n-2)/2 \rfloor$  to 0 do           ▷ 1. Creazione del max-heap con approccio bottom-up
2:      $i \leftarrow j$ 
3:      $k \leftarrow \text{indexMaxChild}(P, i)$            ▷ NB: max-heap: uso indexMaxChild()
4:     while ( $k \neq \text{null}$  AND  $P[i].\text{getKey}() < P[k].\text{getKey}()$ ) do       ▷ NB: max-heap:  $P[i] < P[k]$ 
5:       swap( $P[i], P[k]$ )
6:        $i \leftarrow k$ 
7:        $k \leftarrow \text{indexMaxChild}(P, i)$            ▷ NB: max-heap: uso indexMaxChild()
8:   last  $\leftarrow n - 1$ 
9:   for  $j \leftarrow 1$  to 0 do
10:    swap( $P[0], P[\text{last}]$ )           ▷ spostato chiave massima in fondo
11:    last  $\leftarrow n - j - 1$            ▷ escludo la chiave massima appena spostata
12:     $i \leftarrow 0$            ▷ down-heap bubbling di  $P[0]$ 
13:     $k \leftarrow \text{indexMaxChild}(P, i)$ 
14:    while ( $k \neq \text{null}$  AND  $P[i].\text{getKey}() < P[k].\text{getKey}()$ ) do
15:      swap( $P[i], P[k]$ )
16:       $i \leftarrow k$ 
17:       $k \leftarrow \text{indexMaxChild}(P, i)$            ▷ NB: max-heap: uso indexMaxChild()
```

6 Mappe

6.1 Introduzione

Definizione

Una mappa è una collezione di entry con chiavi distinte da un insieme U per cui è definito l'operatore $=$. La mappa implementa l'accesso, l'inserimento e la rimozione delle entry attraverso le chiavi.

Metodi e interfaccia in Java

```
1 public interface Map<K,V>{
2     int size();
3     boolean isEmpty();
4     /** Returns the value of the entry (k,v) with k = key if exists, otherwise null */
5     V get (K key);
6     /** If there is an entry (k,v) with k = key then update the value of that entry and
7     returns the old value, otherwise inserts a new entry (key, value) and returns null */
8     V put (K key, V value);
9     /** Removes the entry (k,v) with k = key and returns the value of the entry,
10     otherwise return null*/
11     V remove (K key);
12     Iterable<K> keySet();
13     Iterable<V> values();
14     Iterable<Entry<K,V>> entrySet();
15 }
```

6.2 Implementazioni e relativa complessità

| metodo | position-based list | array | tabella hash | MWST, (2,4)-Tree, RB-Tree |
|--------|---------------------|--------|------------------|---------------------------|
| get | $O(n)$ | $O(1)$ | $O(1 + \lambda)$ | $O(\log n)$ |
| put | $O(n)$ | $O(1)$ | $O(1 + \lambda)$ | $O(\log n)$ |
| remove | $O(n)$ | $O(1)$ | $O(1 + \lambda)$ | $O(\log n)$ |

Implementazione con position-based list

L'implementazione con una position-based list è efficiente dal punto di vista spaziale, in quanto lo spazio occupato in memoria è proporzionale al numero di entry, ma è molto svantaggiosa dal punto di vista computazionale in quanto i metodi utilizzano la ricerca in una lista che impiega complessità $O(n)$.

Implementazione con array

Si suppone di poter creare un array con gli indici $[0, |U| - 1]$ e di disporre di un mapping 1:1 tra l'insieme delle chiavi e l'indice dell'array. È possibile riservare una cella dell'array ad ogni possibile entry della mappa. In questo modo i vari metodi hanno complessità costante, ma nel caso in cui $|U| \gg \# \text{ entry}$, la struttura diventa molto svantaggiosa dal punto di vista della memoria occupata.

Implementazione con tabella hash

Simile all'implementazione precedente, ma utilizzando le hash table e le funzioni hash è possibile ridurre notevolmente la memoria occupata a discapito di una lievemente peggiore complessità computazionale. La complessità dipende dal load factor λ o indice di riempimento medio della tabella hash. Il load factor λ rappresenta un trade-off tra lo spazio occupato in memoria e la complessità computazionale, per λ grandi si avrà migliore complessità spaziale e peggiore complessità computazionale, viceversa per λ piccoli si avrà il contrario. In genere si sceglie $\lambda < 0.9$ e se $\lambda \in O(1)$, la complessità finale è $O(1)$.

Implementazione con alberi binari di ricerca

Supponendo che le chiavi provengano da un insieme totalmente ordinato, è possibile implementare la mappa attraverso particolari alberi binari di ricerca (2,4-Tree o Red-Black Tree) in cui la ricerca e i tre metodi hanno complessità $O(\log n)$.

7 Hash Table

7.1 Introduzione

Definizione

Una tabella hash è una struttura che possiede i seguenti elementi:

1. una funzione di hash $h : \{\text{chiavi}\} \rightarrow [0, N - 1]$
2. un bucket array A con capacità N , in cui ogni bucket $A[i]$ sono memorizzate tutte le entry $\langle k, v \rangle$ per cui $h(k) = i$, con $0 \leq i < N$
3. un metodo di risoluzione delle collisioni (chiavi distinte associate allo stesso bucket)

Funzione di hash

Una buona funzione di Hash deve essere il più possibile un processo random:

- $P[h(k) = i] = 1/N$, ovvero la distribuzione delle chiavi dei bucket deve tendere all'uniforme
- $P[h(k) = i \mid h(k') = j] = P[h(k) = i]$, ovvero gli assegnamenti devono essere indipendenti tra loro

Esempi di funzioni Hash per tipi built-in di Java:

- `byte, short, char, int`: $k \mapsto (\text{int})k$
- `float` (32 bit): $k \mapsto \text{Float.floatToIntBits}(k)$
- `long` (64 bit): $k \mapsto (\text{int})(k \gg 32 + (\text{int})k)$
- `double` (64 bits): $k \mapsto ((\text{int})(k \gg 32 + (\text{int})k)) \circ (\text{Double.doubleToIntBits}(k))$
- `string`: due possibilità:
 - Polynomial hash code: $h(S) = \sum_{i=0}^{k-1} s_i \cdot a^{k-i-1}$ con $a = 31, 33, 37, 39, 41$, Java usa $a = 31$
 - Cyclic Shift: somma carattere per carattere e dopo ogni somma si esegue una rotazione a sinistra di 5 posizioni (più facile implementazione rispetto al calcolo delle potenze di a)

Funzione di compressione

Per convertire un generico numero $\in \mathbb{N}$ generato dalla funzione di hash in un intero valido per accedere alla sequenza, si utilizzano due funzioni di compressione:

- **Division Method**: $i \mapsto i \bmod N$
è bene scegliere N primo e distante da potenze di 2: se $N = 2^p$ è come considerare solo i p bit meno significativi, se $N = 10^p$ è come considerare solo le p cifre meno significative in base 10.
- **Multiply-Add-Divide MAD**: $i \mapsto [(ai + b) \bmod p] \bmod N$
con p primo, $p > N$, $a, b \in [0, p - 1]$ scelti a caso, $a > 0$, più costoso ma con migliore distribuzione

Risoluzione delle collisioni

Per la risoluzione delle collisioni ricorre a due metodi:

- **Separate Chaining**: ogni bucket è visto come una *Map* più piccola implementata come lista
- **Open Addressing**: consiste nel salvare le entry direttamente nelle celle del bucket array, senza far ricorso a strutture aggiuntive, ma si complica la loro gestione; non affrontato a lezione

7.2 Algoritmi base: get, put, remove

Accesso ad un elemento - get

- L'algoritmo si basa sulla ricerca della chiave nei bucket con funzione di hash e sulla ricerca nella lista concatenata associata al bucket in questione.
- La complessità dell'algoritmo al caso medio è $\Theta(1 + \lambda)$.

0: **Algoritmo:** GET(k)
 Input: chiave k dell'elemento cercato e riferimento implicito al bucket array A
 Output: valore della entry con chiave k

1: **if** (\exists entry $(k, x) \in A[h(k)]$) **then**
2: return x
3: **else**
4: return null

Inserimento di un elemento - put

- L'algoritmo si basa sulla ricerca della chiave nei bucket con funzione di hash e sulla ricerca nella lista concatenata associata al bucket in questione.
- La complessità dell'algoritmo al caso medio è $\Theta(1 + \lambda)$.

0: **Algoritmo:** PUT(k, v)
 Input: entry (k, v) da inserire e riferimento implicito al bucket array A
 Output: vecchio valore della entry (k, v) se presente

1: **if** (\exists entry $(k, x) \in A[h(k)]$) **then**
2: sostituisci x con v
3: return x
4: **else**
5: inserisci (k, v) in coda al bucket $A[h(k)]$
6: incrementa di 1 la *size* della hashTable
7: return null

Rimozione di un elemento - remove

- L'algoritmo si basa sulla ricerca della chiave nei bucket con funzione di hash e sulla ricerca nella lista concatenata associata al bucket in questione.
- La complessità dell'algoritmo al caso medio è $\Theta(1 + \lambda)$.

0: **Algoritmo:** GET(k)
 Input: chiave k dell'elemento cercato e riferimento implicito al bucket array A
 Output: valore della entry con chiave k

1: **if** (\exists entry $(k, x) \in A[h(k)]$) **then**
2: rimuovi (k, x) da $A[h(k)]$
3: decrementa di 1 la *size* della hashTable
4: return x
5: **else**
6: return null

7.3 Load Factor e complessità

Load Factor

Per una tabella hash di capacità N , con n entry memorizzate, il load factor è $\lambda := \frac{n}{N}$. Rappresenta la lunghezza media delle liste nei bucket.

Complessità al caso pessimo

Si osserva che per una tabella hash di capacità N , con n entry memorizzate, la complessità dei tre metodi al caso pessimo è $\Theta(n)$. Se tutte le entry fossero contenute nello stesso bucket la ricerca (prevista da ogni metodo) di una chiave in una lista è $\Theta(n)$. Il risultato di n inserimenti nella tabella diventerà $\Theta(n^2)$.

Complessità al caso medio

Al caso medio, il costo di un inserimento si ottiene considerando il costo della ricerca della chiave k in nella lista i contenente n_i entry, ovvero $O(n_i + 1)$. Si ha, quindi:

$$O\left(\frac{1}{N} \sum_{i=0}^{N-1} (n_i + 1)\right) = O\left(\frac{1}{N} \sum_{i=0}^{N-1} n_i + \frac{1}{N} \sum_{i=0}^{N-1} 1\right) = O(\lambda + 1)$$

- se $\lambda \in O(1)$, allora la complessità finale sarà $\Theta(1)$
- in generale si mantiene $\lambda < 0.9$, in Java $\lambda \leq 0.75$
- il load factor λ rappresenta un trade-off tra lo spazio occupato in memoria e la complessità computazionale, per λ grandi si avrà migliore complessità spaziale e peggiore complessità computazionale, viceversa per λ piccoli si avrà migliore complessità computazionale, ma peggiore complessità spaziale.

7.4 Rehashing

Funzionamento

Con il popolamento della hashTable, il load factor cresce e si rischia di superare la soglia impostata. Per mantenere un load factor accettabile, è necessario eseguire un ridimensionamento del bucket array e di conseguenza è necessario eseguire un rehashing delle entry. Per ottenere un buon rehash è bene scegliere la nuova dimensione $N' > 2N$ con N' numero primo.

Complessità

Siccome so già che le chiavi in una hash table sono distinte, quando le inserisco nel nuovo bucket array ridimensionato non ho bisogno di eseguire la ricerca per verificare se esiste già una entry con la stessa chiave. Questo bypass mi permette di eseguire gli n inserimenti individualmente in tempo costante e complessivamente il rehash ha una complessità di $\Theta(n)$.

Il rehash viene eseguito solo dopo n inserimenti con load factor ottimale che possiedono un costo aggregato di $\Theta(n\lambda) = \Theta(n)$. Il costo del rehash ($\Theta(n)$) viene, ammortizzato da quello aggregato dei precedenti n inserimenti.

8 Alberi binari di ricerca

8.1 Alberi binari di ricerca in generale

Definizione

Un albero binario di ricerca è un albero binario in cui i nodi interni memorizzano entry con chiavi provenienti da un universo ordinato e per ogni nodo v con chiave k , le chiavi di $T.\text{left}(v)$ sono $< k$, mentre le chiavi di $T.\text{right}(v)$ sono $> k$. Le foglie sono sentinelle e non contengono entry.

Algoritmo di ricerca

- L'algoritmo si basa sull'ordinamento reciproco dei nodi all'interno dell'albero ed esegue una discesa dalla radice fino al nodo che contiene la entry cercata o alla foglia dove inserirla.
- La complessità dell'algoritmo è $\Theta(h)$, che può diventare $\Theta(n)$ se l'albero è molto sbilanciato.

0: **Algoritmo:** `TREESEARCH(k, v)`

Input: chiave k della entry cercata, nodo v da cui cercare e riferimento implicito all'ABR T

Output: nodo in T_v dove è presente la entry con chiave k o dove va inserita

```
1:  if ( $T.\text{isExternal}(v)$  OR  $v.\text{getElement}().\text{getKey}() = k$ ) then
2:      return  $v$ 
3:  if ( $k < v.\text{getElement}().\text{getKey}()$ ) then
4:      return TreeSearch( $k, T.\text{left}(v)$ )
5:  else
6:      return TreeSearch( $k, T.\text{right}(v)$ )
```

Limiti degli alberi binari di ricerca in generale

Il problema degli alberi binari di ricerca è che la complessità dei vari metodi varia tra $\Theta(\log n)$ se l'albero è perfettamente bilanciato e $\Theta(n)$ se l'albero è fortemente sbilanciato. In quest'ultimo caso l'ABR non migliora le prestazioni di una comune lista concatenata. Le possibili soluzioni sono:

- ribilanciare l'albero quando lo sbilanciamento supera una soglia prefissata (AVL o Red-Black Tree)
- rendere i nodi più capienti per assorbire meglio gli effetti di inserimento e rimozione (MW Search Tree o (2,4)-Tree)

Algoritmo di accesso - get

- L'algoritmo si basa sulla ricerca vista sopra.
- La complessità dell'algoritmo è $\Theta(h)$, che può diventare $\Theta(n)$ se l'albero è molto sbilanciato.

0: **Algoritmo:** `GET(k)`

Input: chiave k della entry cercata e riferimento implicito all'ABR T

Output: valore della entry con chiave k

```
1:   $w \leftarrow \text{TreeSearch}(k, T.\text{root}())$ 
2:  if ( $T.\text{isInternal}(w)$ ) then
3:      return  $w.\text{getElement}().\text{getValue}()$ 
4:  else
5:      return null
```

Algoritmo di inserimento - insert

- L'algoritmo si basa sulla ricerca vista sopra.
- La complessità dell'algoritmo è $\Theta(h)$, che può diventare $\Theta(n)$ se l'albero è molto sbilanciato.

0: **Algoritmo:** PUT(k, v)
 Input: entry (k, v) da inserire e riferimento implicito all'ABR T
 Output: vecchio valore della entry (k, v) se presente

1: $w \leftarrow \text{TreeSearch}(k, T.\text{root}())$
2: **if** ($T.\text{isInternal}(w)$) **then**
3: $x \leftarrow w.\text{getElement}().\text{getKey}()$
4: sostituisci v con x nella entry $w.\text{getElement}()$
5: return x
6: **else**
7: $\text{expandExternal}(w, (k, v))$
8: incrementa di 1 il numero di entry di T
9: return null

Algoritmo di rimozione - remove

- L'algoritmo si sviluppa in due casi:
 1. se il nodo da rimuovere ha una foglia come figlio, faccio salire l'altro figlio
 2. se il nodo da rimuovere non ha foglie come figli, sostituisco il nodo da rimuovere w con il precedente nella visita in order y (che avrà una foglia come figlio destro) e sposto il sottoalbero sinistro di y al posto di y
- La complessità dell'algoritmo è $\Theta(h)$, ottenuta dalla complessità aggregata di TreeSearch ($\Theta(h)$), del caso 1 ($\Theta(1)$) e del caso 2 ($\Theta(h)$), più altre operazioni ($\Theta(1)$).

0: **Algoritmo:** REMOVE(k)
 Input: chiave k della entry da rimuovere e riferimento implicito all'ABR T
 Output: valore della entry rimossa

1: $w \leftarrow \text{TreeSearch}(k, T.\text{root}())$
2: **if** ($T.\text{isInternal}(w)$) **then**
3: $avalue \leftarrow w.\text{getElement}().\text{getValue}()$
4: decrementa di 1 il numero di entry di T
5:

▷ Caso 1: w interno con almeno un figlio foglia

6: **if** ($T.\text{isExternal}(T.\text{left}(w))$ OR $T.\text{isExternal}(T.\text{right}(w))$) **then**
7: $u_L \leftarrow T.\text{left}(w)$ $u_R \leftarrow T.\text{right}(w)$
8: **if** $T.\text{isExternal}(u_L)$ **then** ▷ Caso 1.1: la foglia è il figlio sinistro
9: cancella w e u_L
10: fai salire u_R al posto di w
11: **else** ▷ Caso 1.2: la foglia è il figlio destro
12: cancella w e u_R
13: fai salire u_L al posto di w
14: **else** ▷ Caso 2: w interno con due figli
15: $y \leftarrow$ nodo con chiave massima del sottoalbero di sinistra
16: sposta il figlio sinistro di y al posto di y
17: sostituisci il nodo w da rimuovere con il nodo y
18: return $avalue$
19: **else**
20: return null

8.2 Multi-Way Search Tree - MWST

Definizione

Un Multi-Way Search Tree è un albero ordinato tale che ogni nodo interno ha $d \geq 2$ figli (v_1, v_2, \dots, v_d) detto d -node che soddisfa le seguenti caratteristiche:

- memorizza $d - 1$ entry $(k_1, x_1), (k_2, x_2), \dots, (k_{d-1}, x_{d-1})$ con $k_1 < k_2 < \dots < k_{d-1}$
- la chiave di ogni entry $e \in T_{v_i}$ soddisfa $k_{i-1} < e.getKey() < k_{i+1}$, supponendo $k_0 = 0, k_d = +\infty$

Proprietà

- per ogni nodo v con ad esempio 3 entry $(k_1, x_1), (k_2, x_2), (k_3, x_3)$ e 4 figli v_1, v_2, v_3, v_4 , le chiavi sono ordinate da sinistra verso destra nel seguente modo:

$$0 < \{\text{chiavi} \in T_{v_1}\} < k_1 < \{\text{chiavi} \in T_{v_2}\} < k_2 < \{\text{chiavi} \in T_{v_3}\} < k_3 < \{\text{chiavi} \in T_{v_4}\} < +\infty$$

- un MWST con n entry ha $n + 1$ foglie
- se tutti i nodi fossero dei d -node, allora l'albero viene detto d -ario, con:

$$\# \text{ nodi interni} = x, \quad \# \text{ entry} = (d - 1)x, \quad \# \text{ foglie} = (d - 1)x + 1$$

Algoritmo di ricerca

- L'algoritmo si basa sull'ordinamento reciproco dei nodi all'interno dell'albero ed esegue una discesa dalla radice fino al nodo che contiene la entry cercata o alla foglia dove inserirla.
- La complessità dell'algoritmo è $\Theta(h \cdot d_{max})$, con d_{max} = massimo valore di d in T_v

0: **Algoritmo:** MWTreeSearch(k, v)

Input: chiave k della entry cercata, nodo v da cui cercare e riferimento implicito al MWST T

Output: nodo in T_v dove è presente la entry con chiave k o dove va inserita

```
1:  if (T.isExternal(v)) then
2:      return v
3:  trova  $i$  tale che  $k_{i-1} < k \leq k_i$  con  $k_0 = 0, k_d = +\infty$ 
4:  if ( $k = k_i$ ) then
5:      return v
6:  else
7:      return MWTreeSearch( $k, v_i$ )
```

Algoritmo di accesso - get

- L'algoritmo si basa sulla ricerca vista sopra.
- La complessità dell'algoritmo è $\Theta(h \cdot d_{max})$.

0: **Algoritmo:** GET(k)

Input: chiave k della entry cercata e riferimento implicito al MWST T

Output: valore della entry con chiave k

```
1:   $w \leftarrow$  MWTreeSearch( $k, T.root()$ )
2:  if (T.isInternal(w)) then
3:      trova  $e \in w$  tale che  $e.getKey() = k$ 
4:      return  $e.getValue()$ 
5:  else
6:      return null
```

8.3 (2,4)-Tree

Definizione

Un (2,4)-Tree è un MWS-Tree tale che:

- ogni nodo interno è un d -node con $2 \leq d \leq 4$ con d figli e $d - 1$ entry
- tutte le foglie hanno la stessa profondità

Proprietà

- l'altezza di un (2,4)-Tree con $n > 0$ entry è $\Theta(\log n)$
- la complessità dei metodi di ricerca e di accesso è $\Theta(\log n)$

Algoritmo di inserimento - put

- L'algoritmo si basa sulla ricerca vista sopra e sul fatto che un nodo può contenere più entry
- La complessità dell'algoritmo è $\Theta(\log n)$ data dalla MWTreeSearch ($\Theta(\log n)$) e dallo split ($\Theta(\log n)$)

0: **Algoritmo:** PUT(k, x)

Input: entry (k, x) da inserire e riferimento implicito al (2,4)-Tree T

Output: valore della entry con chiave k

```
1:   $w \leftarrow \text{MWTreeSearch}(k, T.\text{root}())$ 
2:  if ( $T.\text{isInternal}(w)$ ) then                                ▷ Caso 1: se esiste già una entry con chiave  $k$ 
3:       $e \leftarrow \text{entry in } w \text{ con } e.\text{getKey}() = k$ 
4:       $y \leftarrow e.\text{getValue}()$ 
5:      sostituisci  $x$  con  $y$  in  $e.\text{value}()$ 
6:      return  $y$ 
7:  else                                                        ▷ Caso 2: bisogna inserire una nuova entry
8:       $e \leftarrow (k, x)$ 
9:      if ( $T.\text{isRoot}(w)$ ) then                                ▷ Caso 2.1: l'albero è vuoto  $\rightarrow$  creo radice e due figli
10:          $\text{expandExternal}(w, e)$ 
11:      else                                                    ▷ Caso 2.2: lo inserisco nel padre
12:          $u \leftarrow T.\text{parent}(w)$ 
13:         inserisci  $e$  in  $u$  aggiungendo una foglia  $w'$ 
14:         if  $u$  è un 5-node then                                ▷ se c'è overflow, invoco split
15:              $\text{split}(u)$ 
16:     incrementa di 1 il numero delle entry in  $T$ 
17:     return null
```

Algoritmo di split

- L'algoritmo propaga l'overflow verso l'alto e se serve aumenta l'altezza dell'albero:
 1. se ho la radice in overflow, spezzo la radice in due nodi: $(e_1, e_2, e_3, e_4) \rightarrow (e_1, e_2), (e_4)$ e creo una nuova radice con la entry "centrale" (e_3) e con i due nodi $(e_1, e_2), (e_4)$ come figli
 2. altrimenti spezzo il nodo in due $(e_1, e_2, e_3, e_4) \rightarrow (e_1, e_2), (e_4)$ e inserisco la entry "centrale" (e_3) nel padre; se il padre è in overflow, invoco lo split sul padre
- La complessità dell'algoritmo è $\Theta(\log n)$ in quanto ogni invocazione ha costo $O(1)$ e # incocazioni è proporzionale all'altezza dell'albero $\Theta(\log n)$

0: **Algoritmo:** SPLIT(u)

Input: nodo in overflow e riferimento implicito al (2,4)-Tree T

Output: albero senza nodi in overflow

```

1:  sia  $u = (e_1, e_2, e_3, e_4)$  con figli  $u_1, u_2, u_3, u_4, u_5$ 
2:  creo due nodi  $u' = (e_1, e_2)$  con figli  $u_1, u_2, u_3$  e  $u'' = (e_4)$  con figli  $u_4, u_5$ 
3:  if ( $T.isRoot(u)$ ) then                                ▷ se ho la radice in overflow
4:      crea una nuova radice contenente  $e_3$  e con figli  $u'$  e  $u''$  e cancella  $u$ 
5:  else                                                    ▷ se ho un generico nodo in overflow
6:       $v \leftarrow T.parent(u)$ 
7:      inserisci  $e_3$  in  $v$  con figli  $u'$  e  $u''$  e cancella  $u$ 
8:      if ( $v$  è un 5-node) then                                ▷ se il padre è in overflow
9:          split( $u$ )

```

Algoritmo di rimozione - remove

- L'algoritmo si basa sulla ricerca vista sopra e sul fatto che un nodo può contenere più entry
- La complessità dell'algoritmo è $\Theta(\log n)$ data dalla MWTreeSearch ($\Theta(\log n)$) e dal delete ($\Theta(\log n)$)

0: **Algoritmo:** REMOVE(k)

Input: chiave k della entry da rimuovere e riferimento implicito al (2,4)-Tree T

Output: valore della entry rimossa

```

1:   $w \leftarrow MWTreeSearch(k, T.root())$ 
2:  if ( $T.isInternal(w)$ ) then                                ▷ Caso 1: se esiste la entry con chiave  $k$ 
3:       $e \leftarrow$  entry in  $w$  con  $e.getKey() = k$ 
4:       $y \leftarrow e.getValue()$ 
5:      if ( $height(w) = 1$ ) then                                ▷ Caso 1.1: se la entry si trova alla base (non ha nodi figli)
6:          delete( $e, w$ )
7:      else                                                    ▷ Caso 1.2: se la entry ha nodi figli
8:           $v \leftarrow$  figlio di  $w$  a sx di  $e$ 
9:           $e' \leftarrow$  entry con chiave massima in  $T_v$ 
10:          $z \leftarrow$  nodo contenente  $e'$ 
11:         copia  $e'$  al posto di  $e$  in  $w$                                 ▷ sostituisco il nodo da eliminare
12:         delete( $e', z$ )                                ▷ elimino il sostituito dalla sua posizione originale
13:         decrementa di 1 il numero delle entry in  $T$ 
14:         return  $y$ 
15:  else                                                    ▷ Caso 2: se la entry non esiste
16:      return null

```

Algoritmo di delete

- L'algoritmo propaga l'underflow verso l'alto e se serve diminuisce l'altezza dell'albero:
 1. se ho la radice in underflow: imposto come nuova radice il nodo non foglia
 2. se il nodo in underflow ha un fratello con $d = 3, 4$: vado in prestito di una entry dal fratello ed eseguo una rotazione delle entry fratello \rightarrow padre \rightarrow nodo in underflow
 3. se il nodo in underflow ha un fratello con $d = 2$: vado in prestito dal padre e sposto la entry che lega i due fratelli dal padre al fratello, aggiungo il sottoalbero del nodo in underflow al fratello e richiamo ricorsivamente delete sul padre per la entry spostata
- La complessità dell'algoritmo è $\Theta(\log n)$ in quanto ogni invocazione ha costo $O(1)$ e # incocazioni è proporzionale all'altezza dell'albero $\Theta(\log n)$

0: **Algoritmo:** DELETE(u, e)

Input: nodo u , entry $e \in u$ da rimuovere e riferimento implicito al (2,4)-Tree T

Output: albero senza nodi in underflow

- 1: rimuove e da u e figlio vuoto o foglia discriminata da e
 - 2: applico i casi visti sopra
-

8.4 Red-Black Tree

Definizione

Un Red-Black Tree (RB-Tree) è un albero binario di ricerca i cui nodi hanno un colore rosso o nero e in cui valgono le seguenti proprietà:

- *Root Property*: la radice è nera
- *External Property*: le foglie sono nere
- *Red Property*: i figli di un nodo rosso sono necessario
- *Depth Property*: tutte le foglie hanno la stessa Black Depth

Correlazione tra (2,4)-Tree e Red-Black Tree

Si osserva che è possibile trasformare un (2,4)-Tree in un Red-Black Tree e viceversa:

- se ho un solo nodo nero e due foglie \rightarrow 2-nodo con una entry e due foglie
- se ho un nodo nero con un figlio rosso e tre foglie totali \rightarrow 3-nodo con due entry e 3 foglie
- se ho un nodo nero con due figli rossi e quattro foglie totali \rightarrow 4-nodo con tre entry e 4 foglie

9 Multimappe

9.1 Introduzione

Definizione

Una multimappa è una mappa che elimina il vincolo di unicità delle chiavi

Metodi e interfaccia in Java

```
1 public interface Map<K,V>{
2     int size();
3     boolean isEmpty();
4     /** Returns a collection of all the values of the entries (k,*) with k = key */
5     Iterable<V> get (K key);
6     /** Inserts a new entry (key, value) */
7     void put (K key, V value);
8     /** Removes the entry (k,v) with k = key and returns true if an entry is removed,
9     otherwise false */
10    boolean remove (K key, V value);
11    Iterable<K> keySet();
12    Iterable<V> values();
13    Iterable<Entry<K,V>> entrySet();
14 }
```

Indice primario e secondario

Nei database, l'accesso ai dati è reso efficiente attraverso l'accesso ad un indice primario realizzato tramite una mappa e in parallelo attraverso un indice secondario realizzato tramite una multimappa.

9.2 Implementazione e relativa complessità

Implementazione

È possibile implementare una multimappa tramite una mappa in cui le entry sono coppie (k, L_k) con k chiave e L_k una lista che memorizza i valori associati alla chiave. Nel caso in cui si vuole inserire più entry uguali (stessa chiave e valore), si può decidere se bloccare l'inserimento della seconda o memorizzarle entrambe. In quest'ultimo caso, al momento della rimozione si può decidere se rimuoverne solo una o tutte le entry uguali.

Complessità

Per migliore praticità si definisce $n = \#$ chiavi distinte e $s = \max_k |L_k|$ massima lunghezza delle liste associate alle chiavi. Di seguito i metodi e le relative complessità:

- $\text{get}(k)$ ha complessità $\Theta(\log n)$ (come quella della mappa, utilizzando la nuova definizione di n)
- $\text{put}(k,v)$ ha complessità $\Theta(\log n)$ (come quella della mappa, utilizzando la nuova definizione di n)
- $\text{remove}(k,v)$ ha complessità $\Theta(\log n + s)$

10 Grafi

10.1 Introduzione

Definizione

Un grafo G è definito come una coppia di insiemi (o collezioni se esistono elementi duplicati) (V, E) tali che V è l'insieme di vertici e E è una collezione di archi (o coppie di vertici).

Altre definizioni - concetti fondamentali

- **arco orientato**: un arco $(u, v) \in E$ si dice orientato se è stato definito un orientamento $(u \rightarrow v)$ tra i vertici su cui insiste
 - **arco multiplo**: un arco si dice multiplo se compare più volte nell'insieme E
 - **self loop**: un arco si dice self loop se insiste su un unico vertice, ad esempio $(u, u) \in E$
 - **vertici adiacenti**: due vertici u, v si dicono adiacenti se esiste un arco $(u, v) \in E$ che li collega
 - **vicini di un vertice**: i vicini di un vertice v sono tutti i vertici u tali che $(u, v) \in E$
 - **grado di un vertice**: il grado di un vertice $v \in V$, $\text{degree}(v)$, è il numero di archi incidenti su v
 - **grafo diretto**: un grafo si dice diretto se ogni arco in E è orientato
 - **grafo non diretto**: un grafo si dice non diretto se nessun arco in E è orientato
 - **grafo semplice**: un grafo si dice semplice se non ha archi multipli o self loop

 - **cammino**: il cammino è una sequenza ordinata di vertici adiacenti u_1, u_2, \dots, u_k con $(u_i, u_{i+1}) \in E$
 - **lunghezza di un cammino**: la lunghezza di un cammino è il numero di archi (u_i, u_{i+1}) o la somma dei pesi degli archi nel caso di archi pesati
 - **cammino semplice**: un cammino si dice semplice se non ha vertici ripetuti
 - **cammino minimo**: un cammino tra x e y si dice minimo se la sua lunghezza è la minima possibile
 - **distanza tra vertici**: dati due vertici x, y la loro distanza $\text{distanza}(x, y)$ è pari alla lunghezza del cammino minimo che li congiunge, se non esiste cammino minimo, $\text{distanza}(x, y) = +\infty$
 - **ciclo**: un ciclo è un cammino in cui $u_1 = u_k$, la lunghezza del ciclo è analoga a prima
 - **ciclo semplice**: un ciclo si dice semplice se non ha vertici ripetuti ad eccezione degli estremi

 - **sottografo**: un sottografo di G è definito come $G' = (V', E')$ tali che $V' \subseteq V$, $E' \subseteq E$
 - **sottografo di copertura**: un sottografo si dice di copertura (o spanning subgraph) se $V' = V$
 - **grafo connesso**: un grafo si dice connesso se $\forall u, v \in V \exists$ cammino da u a v
 - **grafo disconnesso**: un grafo si dice disconnesso se $\exists u, v \in V$ tali che $\text{distanza}(u, v) = +\infty$
 - **componenti connesse**: le componenti connesse di un grafo $G = (V, E)$ sono i sottografi connessi massimali di G , ovvero i sottografi $G_i = (V_i, E_i)$ con $1 \leq i \leq k$ tali che:
 - $G_i = (V_i, E_i)$ è sottografo connesso
 - $V = V_1 \cup V_2 \cup \dots \cup V_k$ con $V_i \cap V_j = \emptyset$
 - $E = E_1 \cup E_2 \cup \dots \cup E_k$ con $E_i \cap E_j = \emptyset$
 - $\forall i \neq j$ non esistono archi $\in E$ tra V_i e V_j
- Si osserva che se G è connesso, $k = 1$, inoltre la partizione in componenti connesse è univoca
- **albero (libero)**: un albero (libero) è un grafo $G = (V, E)$ connesso e senza cicli
 - **albero radicato**: un albero radicato è un grafo $G = (V, E)$ connesso e senza cicli tale che:
 - esiste un vertice $r \in V$ detto radice
 - per ogni $u \in V \setminus \{r\}$ esiste un unico padre $p(u) \in V$ e vale $E = \{(u, p(u)) : u \in V \setminus \{r\}\}$
 - per ogni $u \in V$, risalendo di padre in padre si arriva a r

Si osserva che un albero radicato è un albero libero e un albero libero si può trasformare in radicato scegliendo un opportuno vertice come radice

- **foresta**: una foresta è un grafo G non connesso, ovvero un insieme di alberi liberi disgiunti; si osserva che un albero è una foresta, ma una foresta non è un albero
- **spanning tree**: uno spanning tree di un grafo G è uno spanning subgraph connesso e senza cicli
- **spanning forest**: una spanning forest di un grafo G è uno spanning subgraph senza cicli

Primitive

- **traversal**: esplorazione del grafo (es. crawling)
- **connettività**: verifica se un grafo è connesso (es. reti wireless)
- **identificazione componenti connesse**: (es. reti wireless)
- **ricerca cammini minimi**: (es. navigatore)
- **ricerca minimum spanning tree**: (es. broadcast efficiente)
- **stima sulla distanza media/massima**: (es. social network)

Proprietà

Sia $G = (V, E)$ grafo semplice non diretto con $|V| = n$ e $|E| = m$, allora valgono le seguenti proprietà:

- $\sum_{v \in V} \text{degree}(v) = 2m$
- $m \leq \binom{n}{2}$, ovvero $m \in O(n^2)$
- se G è albero, allora $m = n - 1$
- se G è connesso, allora $m \geq n - 1$
- se G è senza cicli (foresta), allora $m \leq n - 1$

Implementazione / rappresentazioni

Un grafo può essere rappresentato/implementato in tre modi:

- **strutture base**:
utilizzo due liste L_V e L_E per rappresentare rispettivamente vertici e archi; questo metodo risulta scomodo negli algoritmi in cui è necessario esplorare i vertici adiacenti, per cui non viene molto utilizzata
- **liste di adiacenza**:
per ogni vertice $v \in V$ ho una lista $L(v)$ con i puntatori agli archi incidenti su v o ai vertici raggiungibili da v ; questa rappresentazione occupa spazio lineare nella taglia del grafo $\Theta(n + m)$ e consente l'accesso sequenziale ai vertici adiacenti in tempo lineare al grado di v
- **matrice di adiacenza**:
matrice $n \times n$ in cui le righe e le colonne sono in corrispondenza 1:1 con i vertici e ogni cella è popolata secondo $A[i_1, i_2] := \begin{cases} null & \text{se } (i_1, i_2) \notin E \\ \text{puntatore a } e = (i_1, i_2) \in E & \text{se } (i_1, i_2) \in E \end{cases}$. Si osserva che in un grafo non diretto la matrice è simmetrica. Inoltre permette l'accesso ad un arco in tempo costante, ma richiede che i vertici siano rappresentati come interi. Inoltre occupa uno spazio $\Theta(n^2)$, superlineare nella taglia del grafo che risulta vantaggiosa solo nel caso di grafi densi o per grafi con pochi vertici.
- **mappe di adiacenza**:
una via di mezzo tra le liste e la matrice di adiacenza, per ogni vertice gli archi incidenti su tale vertice sono memorizzati in una mappa (con tabella hash) per cui si ha un tempo di accesso medio lineare $O(1)$ e occupa uno spazio lineare nella taglia del grafo.

10.2 Breadth First Search - BFS

Algoritmo

L'algoritmo di Breath First Search o BFS prevede che:

- prima si visitano tutti i vertici adiacenti e infine si passa ai vicini dei vicini
- ogni vertice ha un campo *ID* che vale 1 se è stato visitato, 0 altrimenti
- ogni arco ha un campo *e.label* per memorizzare una opportuna etichetta o null altrimenti
- *incidentEdges(v)* restituisce un iteratore agli archi incidenti su *v*
- *opposite(v, e)*, con $e = (v, w)$ arco insistente su *v*, restituisce *w*

Durante l'esecuzione della BFS,

- vengono visitati tutti i vertici in C_s componente connessa che contiene *s*
- ogni arco in C_s viene etichettato come **DISCOVERY EDGE** o **CROSS EDGE**:
 - DISCOVERY EDGE** se l'arco è utilizzato per scoprire un nuovo vertice
 - CROSS EDGE** se l'arco porta ad un vertice già visitato
- partiziona C_s in livelli L_i in base alla distanza *i* dei vertici rispetto a *s*

0: **Algoritmo:** BFS(*G, s*)

Input: grafo $G = (V, E)$, $s \in C_s$, vertici con $v.ID = 0$, archi con $e.label = null$

Output: etichetta opportunamente ogni vertice e arco in C_s

```
1:  visita s e  $s.ID \leftarrow 1$ 
2:   $L_0 \leftarrow s$  (lista che contiene s)
3:   $i \leftarrow 0$ 
4:  while (!  $L_i.isEmpty()$ ) do
5:      crea una lista  $L_{i+1}$  vuota
6:      for all  $v$  in  $L_i$  do
7:          for all  $e$  in  $G.incidentEdges(v)$  do
8:              if ( $e.label = null$ ) then                                ▷ se trovo un arco non ancora visitato
9:                   $w \leftarrow G.opposite(v, e)$                         ▷ leggo il vertice opposto
10:                 if ( $w.ID = 0$ ) then                                    ▷ se vertice ancora da visitare
11:                      $e.label \leftarrow \text{DISCOVERY EDGE}$                 ▷  $e = (v, w)$  diventa DISCOVERY
12:                     visita vertice  $w$  e  $w.ID = 1$                     ▷ visito  $w$ 
13:                     inserisci  $w$  in  $L_{i+1}$                             ▷ inserisco  $w$  nella rispettiva lista
14:                 else                                                ▷ se vertice già visitato
15:                      $e.label \leftarrow \text{DISCOVERY EDGE}$ 
16:              $i \leftarrow i + 1$                                        ▷ passo al livello successivo
```

Correttezza e risultati

Alla fine dell'esecuzione della BFS su una componente connessa C_s si ottiene:

- quanto previsto di base dall'algoritmo (vedi sopra)
- i **DISCOVERY EDGE** formano uno spanning tree di C_s radicato in *s* chiamato BFS tree
- se un arco marcato **CROSS EDGE** separa vertici i cui indici di livello differiscono al più di 1
- se un vertice *v* appartiene alla lista L_i , allora $d(s, v) = i$

Complessità

La complessità di BFS è proporzionale al numero di archi della componente connessa C_s :

- se il grafo non è connesso, allora la complessità è $\Theta(m_s)$, con $m_s = \#$ archi di C_s
- se il grafo è connesso, allora la complessità è $\Theta(|E|)$

Generalizzazione su tutto il grafo

- L'algoritmo generalizza la BFS per visitare totalmente un grafo non connesso
- La complessità dell'algoritmo è $\Theta(|E| + |V|)$, supponendo di poter accedere agli elementi di V e E in tempo lineare.

0: **Algoritmo:** BFSCOMPLETE(G)
 Input: grafo $G = (V, E)$
 Output: etichetta opportunamente ogni vertice e arco in G

1: **for all** v **in** V **do**
2: $v.ID \leftarrow 0$
3: **for all** e **in** E **do**
4: $e.label \leftarrow null$
5: **for all** v **in** V **do**
6: **if** $v.ID = 0$ **then**
7: BFS(G, v)

Connettività

- L'algoritmo utilizza BFS modificato per contare le componenti connesse di un grafo G
- La complessità dell'algoritmo è $\Theta(|E| + |V|)$, come per BFSComplete.
- Si usa la BFS modificata, che accetta un terzo parametro ID utilizzato per marcare i vertici visitati

0: **Algoritmo:** CONNETTIVITÀ(G)
 Input: grafo $G = (V, E)$
 Output: restituisce # componenti connesse ed etichetta i vertici con ID distinti

1: **for all** v **in** V **do**
2: $v.ID \leftarrow 0$
3: **for all** e **in** E **do**
4: $e.label \leftarrow null$
5: $i \leftarrow 0$
6: **for all** v **in** V **do**
7: **if** $v.ID = 0$ **then**
8: $i \leftarrow i + 1$
9: BFS(G, v, i)
10: return i

Spanning Tree

- L'algoritmo utilizza la BFS per ottenere lo spanning tree di un grafo connesso
- La complessità dell'algoritmo è $\Theta(|E|)$, come per BFS normale.
- Si usa la BFS modificata, che accetta (e restituisce) una lista L dove salvare i DISCOVERY EDGES

0: **Algoritmo:** SPANNINGTREE(G, s)
 Input: grafo $G = (V, E)$ connesso, $s \in V$
 Output: lista L con gli archi marcati come DISCOVERY EDGES

1: $L \leftarrow$ lista vuota, $s \leftarrow$ vertice qualsiasi di V
2: return BFS(G, s, L)

Cammini minimi

- L'algoritmo utilizza la BFS per ottenere un cammino minimo tra due vertici
- La complessità dell'algoritmo è $\Theta(|E| + |V|)$, come per BFSComplete
- Si usa la BFS modificata che per ogni vertice usa $v.parent$ per il padre e $v.edge$ per l'arco al padre

0: **Algoritmo:** (G, s, t)
 Input: grafo $G = (V, E)$ connesso, $s, t \in V$
 Output: lista L con gli archi del cammino minimo da t a s

1: **for all** v **in** V **do** $v.ID \leftarrow 0, \quad v.parent \leftarrow \text{null}, \quad v.edge \leftarrow \text{null}$
2: **for all** e **in** E **do** $e.label \leftarrow \text{null}$
3: BFS(G, s)
4: **if** ($t.ID = 0$) **then** return null
5: **else**
6: $L \leftarrow$ lista vuota, e $w \leftarrow t$
7: **while** ($w \neq s$) **do**
8: aggiungi $w.edge$ a L
9: $w \leftarrow w.parent$
10: return L

Ciclicità

- L'algoritmo utilizza la BFS per ottenere un ciclo del grafo, se esiste
- La complessità dell'algoritmo è $\Theta(|E| + |V|)$, come per BFSComplete
- Si usa la BFS modificata che:
 - per ogni vertice usa $v.parent$ per il padre e $v.edge$ per l'arco al padre
 - negli ID di ciascun vertice salvo il livello/distanza dal vertice di partenza
 - restituisce un cross edge appena lo trova, altrimenti restituisce null

0: **Algoritmo:** (G)
 Input: grafo $G = (V, E)$ connesso
 Output: lista L con gli archi di un ciclo in G

1: **for all** v **in** V **do** $v.ID \leftarrow 0, \quad v.parent \leftarrow \text{null}, \quad v.edge \leftarrow \text{null}$
2: **for all** e **in** E **do** $e.label \leftarrow \text{null}$
3: $crossEdge \leftarrow$ BFS(G, s), con s vertice generico in V
4: **if** ($crossEdge = \text{null}$) **then** return null
5: **else**
6: $L \leftarrow$ lista con $crossEdge$
7: $u, v \leftarrow$ vertici di $crossEdge$
8: **if** ($u.ID > v.ID$) **then** ▷ riporto i vertici allo stesso livello
9: aggiungi $u.edge$ a $L, \quad u \leftarrow u.parent$
10: **else if** ($u.ID < v.ID$) **then**
11: aggiungi $v.edge$ a $L, \quad v \leftarrow v.parent$
12: **while** ($u \neq v$) **do** ▷ algoritmo modificato per LowestCommonAncestor
13: aggiungi $u.edge$ a $L, \quad u \leftarrow u.parent$
14: aggiungi $v.edge$ a $L, \quad v \leftarrow v.parent$
15: return L

10.3 Depth First Search - DFS

Algoritmo

L'algoritmo di Depth First Search o DFS prevede che:

- prima viene visitato un vertice e ricorsivamente i suoi discendenti e infine si passa ai vertici adiacenti
- ogni vertice ha un campo ID che vale 1 se È stato visitato, 0 altrimenti
- ogni arco ha un campo $e.label$ per memorizzare una opportuna etichetta o null altrimenti
- $incidentEdges(v)$ restituisce un iteratore agli archi incidenti su v
- $opposite(v, e)$, con $e = (v, w)$ arco insistente su v , restituisce w

Durante l'esecuzione della DFS,

- vengono visitati tutti i vertici in C_s componente connessa che contiene s
- ogni arco in C_s viene etichettato come **DISCOVERY EDGE** o **BACK EDGE**:
 - DISCOVERY EDGE** se l'arco è utilizzato per scoprire un nuovo vertice
 - BACK EDGE** se l'arco porta ad un vertice già visitato

0: **Algoritmo:** DFS(G, s)

Input: grafo $G = (V, E)$, $s \in C_s$, vertici con $v.ID = 0$, archi con $e.label = null$

Output: etichetta opportunamente ogni vertice e arco in C_s

```
1:  visita  $s$  e imposta  $s.ID = 1$ 
2:  for all  $e$  in  $G.incidentEdges(v)$  do
3:    if ( $e.label = null$ ) then
4:       $w \leftarrow G.opposite(v, e)$ 
5:      if ( $w.ID = 0$ ) then
6:         $e.label \leftarrow$  DISCOVERY EDGE
7:        DFS( $G, w$ )
8:      else
9:         $e.label \leftarrow$  BACK EDGE
```

Correttezza e risultati

Alla fine dell'esecuzione della DFS su una componente connessa C_s si ottiene:

- quanto previsto di base dall'algoritmo (vedi sopra)
- i **DISCOVERY EDGE** formano uno spanning tree di C_s radicato in s , chiamato DFS Tree, con uno sviluppo prevalentemente in altezza (e non in larghezza)
- se un arco è marcato **BACK EDGE**, allora collega il vertice ad un suo antenato (nel DFS tree)

Complessità

La complessità di DFS è proporzionale al numero di archi della componente connessa C_s :

- se il grafo non è connesso, allora la complessità è $\Theta(m_s)$, con $m_s = \#$ archi di C_s
- se il grafo è connesso, allora la complessità è $\Theta(|E|)$

Generalizzazione su tutto il grafo

- L'algoritmo generalizza la DFS per visitare totalmente un grafo non connesso
- La complessità dell'algoritmo è $\Theta(|E| + |V|)$, supponendo di poter accedere agli elementi di V e E in tempo lineare.

0: **Algoritmo:** DFSCOMPLETE(G)

Input: grafo $G = (V, E)$

Output: etichetta opportunamente ogni vertice e arco in G

```
1:   for all  $v$  in  $V$  do
2:        $v.ID \leftarrow 0$ 
3:   for all  $e$  in  $E$  do
4:        $e.label \leftarrow null$ 
5:   for all  $v$  in  $V$  do
6:       if  $v.ID = 0$  then
7:           DFS( $G, v$ )
```

10.4 Confronto tra BFS e DFS

- **in comune:**
 - su grafi non connessi, hanno la stessa complessità
 - possono essere usati indipendentemente per trovare connettività, spanning tree e cicli
- **vantaggi del BFS:**
 - permette di trovare i cammini minimi
- **vantaggi del DFS:**
 - è più space efficient in quanto richiede spazio proporzionale all'altezza dell'albero della ricor-sione (ogni istanza richiede spazio costante)
 - è più efficiente per trovare un cammino tra due vertici lontani tra loro
 - viene usato per trovare cicli nei grafi diretti

10.5 Cammini minimi su grafi pesati - Dijkstra

Definizione di grafi pesati

- Un grafo pesato è un normale grafo $G = (V, E, w)$ con l'aggiunta di una funzione $w : E \rightarrow \mathbb{R}_0^+$ tale che associa a ciascun arco di E un peso reale non negativo.
- La lunghezza di un cammino u_1, u_2, \dots, u_k è data dalla somma del peso degli archi $\sum_{i=1}^{k-1} w(u_i, u_{i+1})$

Algoritmo

L'algoritmo di Single-Source Shortest Path o (SSSP):

- determina le distanze tra s e gli altri vertici di V e i relativi cammini minimi
- suppone che non esistano cicli e in generale archi con peso negativo
- generalizza la BFS per grafi pesati
- ogni vertice ha un campo $v.D$ per la distanza da s e un campo $v.parent$ per il vertice padre
- utilizza una priority queue Q con $Q.decreaseKey(v.D, v)$ che aggiorna la chiave $v.D$

Durante l'esecuzione dell'algoritmo SSSP:

- vengono visitati tutti i vertici in C_s componente connessa che contiene s
- ad ogni vertice vengono impostati:
 - il campo $v.D$ con la distanza da s ($v.D = +\infty$ se non esiste cammino tra s e v)
 - il campo $v.parent$ con il riferimento al vertice padre nel "SSSP Tree"

0: **Algoritmo:** SHORTESTPATHS(G, s)

Input: grafo $G = (V, E)$, $s \in C_s$

Output: etichetta opportunamente ogni vertice in C_s

```
1:   $s.D \leftarrow 0$                                 ▷ inizializza vertice di partenza
2:   $s.parent \leftarrow \text{null}$ 
3:  for all  $v$  in  $V \setminus \{s\}$  do                ▷ inizializza altri vertici
4:     $v.D \leftarrow +\infty$ 
5:     $v.parent \leftarrow \text{null}$ 
6:   $Q \leftarrow$  priority queue con entry  $(v.D, v)$  per ogni  $v \in V$ 
7:  while ( $!Q.isEmpty()$ ) do                    ▷ finché ci sono nodi da visitare
8:     $(u.D, u) \leftarrow Q.removeMin()$ 
9:    for all  $(u, v)$  in  $G.incidentEdges(u)$  do    ▷ aggiorna la distanza dei vicini
10:     if  $(u.D + w(u, v) < v.D)$  then             ▷ edge relaxation
11:        $v.D \leftarrow u.D + w(u, v)$ 
12:        $v.parent \leftarrow u$ 
13:        $Q.decreaseKey(v.D, v)$ 
```

Correttezza e risultati

Alla fine dell'esecuzione dell'algoritmo di Dijkstra su una componente connessa C_s si ottiene:

- quanto previsto di base dall'algoritmo (vedi sopra)
- se $v.D = +\infty$, allora $v.parent = \text{null} \rightarrow$ se $v \notin C_s$ la sua distanza è $+\infty$ e non ha vertici "scopritori"
- se $v.D < +\infty$, il cammino ottenuto risalendo di padre in padre da v è il cammino minimo da s a v , con $d(s, v) = v.D$
- (nota invariante del for:) al momento dell'estrazione di una entry $(u.D, u)$, il campo $u.D = d(s, u)$

Complessità

La complessità dell'algoritmo di Dijkstra è dominata dai metodi della priority queue. La loro complessità dipende dall'implementazione della priority queue attraverso heap o linked-list:

| metodo | heap | linked-list |
|-------------------------|------------------|-------------|
| 1. costruzione iniziale | $\Theta(n)$ | $\Theta(n)$ |
| 2. removeMin | $\Theta(\log n)$ | $\Theta(n)$ |
| 3. decreaseKey | $\Theta(\log n)$ | $\Theta(1)$ |

Complessivamente si hanno n removeMin() e $\leq m$ edge relaxations, ovvero al più m decreaseKey():

| metodo | heap | linked-list |
|----------------------|--------------------|---------------|
| costruzione iniziale | $\Theta(n)$ | $\Theta(n)$ |
| n removeMin | $\Theta(n \log n)$ | $\Theta(n^2)$ |
| m decreaseKey | $\Theta(m \log n)$ | $\Theta(m)$ |

Per cui in totale si avrà:

- con l'heap la complessità è $\Theta(n + m) \log n$
- con la lista la complessità è $\Theta(n^2)$

In funzione del valore di m si potrà scegliere l'una o l'altra implementazione:

- con $m \leq n^2 / \log n \rightarrow$ heap
- con $m > n^2 / \log n \rightarrow$ linked list

Nota: usando un Fibonacci Heap è possibile implementare una priority queue e avere come complessità complessiva di ShortestPaths $\Theta(m + n \log n)$

10.6 Grafi diretti

Definizioni

Un grafo diretto è un grafo in cui ogni arco in E è costituito da una coppia orientata di vertici.

- **archi uscenti** da v sono gli archi della forma (v, u)
- **archi entranti** in v sono gli archi della forma (u, v)
- **grado uscente** di v $\text{outdegree}(v)$ è la somma degli archi uscenti da v
- **grado entrante** di v $\text{indegree}(v)$ è la somma degli archi entranti in v
- $\sum_{v \in V} \text{outdegree}(v) = \sum_{v \in V} \text{indegree}(v) = |E|$
- cammino, ciclo, distanza sono analoghi ai grafi non diretti
- **reachable**(v) è l'insieme di tutti i vertici raggiungibili da v con un cammino diretto
- **fortemente connesso** se per ogni coppia ordinata (u, v) di vertici, esiste un cammino da u a v
- **debolmente connesso** se esiste una coppia ordinata (u, v) di vertici, per cui non esiste un cammino da u a v

Rappresentazione e implementazioni

Per rappresentare gli archi uscenti ed entranti attraverso le liste di adiacenza, si può salvare nelle liste di adiacenza solo gli archi uscenti da ogni vertice. Inoltre $\text{incidentEdges}(v)$ restituisce una lista solo agli archi uscenti da v .

BFS su grafi diretti

Simile a quella per grafi non diretti, soltanto che gli archi vengono etichettati **DISCOVERY EDGE** o **ALTRO**. La complessità non viene alterata.

DFS su grafi diretti

Simile a quella per grafi non diretti, solo che un arco (u, v) non **DISCOVERY EDGE** può essere etichettato:

- **BACK EDGE** se v è antenato di u nel DFS Tree
- **ALTRO** negli altri casi (se v è discendente di u o se collega due rami dell'albero)

Per impostare correttamente un arco **BACK EDGE** o **ALTRO**, creo un nuovo tag per i vertici $v.ON$ che inizialmente è impostato per tutti a 0. All'inizio dell'esecuzione di $\text{DFS}(G, v)$, si setta $v.ON = 1$ e alla fine dell'esecuzione lo si reimposta $v.ON = 0$. In questo modo, ad un certo istante, tutti i vertici avranno $v.ON = 0$ tranne quelli per cui la chiamata ricorsiva è aperta e saranno tutti antenati del vertice visitato con l'ultima chiamata in esecuzione. Posso quindi inserire un controllo: se l'arco non **DISCOVERY** insiste su un vertice con $v.ON = 1$, allora è un **BACK EDGE**, altrimenti verrà marcato come **ALTRO**. Questa modifica non altera la complessità del DFS.

Confronto tra BFS e DFS su grafi diretti

Definiti $R_s = \text{reachable}(s)$ e $m_s = \sum_{v \in R_s} \text{outdegree}(v)$:

- **in comune:**
 - hanno la stessa complessità $\Theta(m_s)$
 - possono essere usati indipendentemente per determinare R_s in $\Theta(m_s)$
- **vantaggi del BFS:**
 - permette di trovare i cammini minimi (Dijkstra) in $\Theta(m_s)$
- **vantaggi del DFS:**
 - usato per determinare se G è fortemente connesso in $\Theta(|V| + |E|)$
 - usato per determinare cicli diretti in $\Theta(|V| + |E|)$

10.7 Grafi diretti senza cicli - DAG

Definizione

Un grafo diretto aciclico o (DAG) è un grafo diretto privo di cicli diretti (può avere cicli se gli archi fossero non orientati, l'importante è che non abbia cicli considerando l'orientamento degli archi).

Ordinamento topologico

Un ordinamento topologico per un DAG $G = (V, E)$ è un ordinamento dei vertici $v_1, v_2, \dots, v_n \in V$ tale che per ogni arco $(v_i, v_j) \in E$ si ha che $i < j$. Di seguito l'algoritmo:

- si utilizza una lista S con i vertici con indegree = 0; si estraggono i vertici da L , si inseriscono in una coda L e si riducono gli indegree dei vertici adiacenti. Quando un vertice ha indegree(v) = 0 viene inserito in S pronto per una nuova estrazione.
- La complessità dell'algoritmo è $\Theta(|E| + |V|)$: il popolamento iniziale di L è fatto in $\Theta(|E| + |V|)$ e il resto delle operazioni è proporzionale linearmente alla taglia del grafico.

0: **Algoritmo:** TOPOLOGICALSORT(G)

Input: DAG G

Output: coda L con vertici ordinati topologicamente

```
1:   $L \leftarrow$  coda vuota
2:   $S \leftarrow$  lista con vertici con  $v.indeg = 0$ 
3:  while ( $!S.isEmpty()$ ) do
4:     $v \leftarrow S.first().getElement()$  e  $S.remove(S.first())$ 
5:     $L.addLast(v)$ 
6:    for all  $e$  in  $G.incidentEdges(v)$  do
7:       $u \leftarrow G.opposite(v, e)$ 
8:       $u.indeg \leftarrow u.indeg - 1$ 
9:      if ( $u.indeg = 0$ ) then  $S.addLast(u)$ 
10: return  $L$ 
```

11 Ordinamenti

11.1 Descrizione del problema

Si vuole riordinare in ordine non decrescente una sequenza di chiavi provenienti da un universo totalmente ordinato, le specifiche generali sono le seguenti:

- **input:** sequenza S con n chiavi da ordinare
- **output:** sequenza S ordinata in senso non decrescente

Gli algoritmi si raggruppano in due categorie:

- **metodi basati sul confronto:**

I metodi studiati basati sul confronto sono MergeSort, HeapSort, QuickSort, RandomizedQuickSort e InsertionSort. Si può dimostrare che, per una istanza generica hanno complessità $\Omega(n \log n)$. Per questa proprietà, il MergeSort, l'HeapSort e il RandomizedQuickSort hanno complessità ottima.

- **metodi non basati sul confronto:**

I metodi studiati non basati sul confronto sono BucketSort e RadixSort. Questi hanno complessità computazionale migliore di quelli precedenti, ma a differenza dei primi, non possono essere implementati inplace e richiedono maggiore complessità spaziale.

11.2 MergeSort

Algoritmo

0: **Algoritmo:** MERGESORT(S)
 Input: come da specifica
 Output: come da specifica

1: **if** ($n \leq 1$) **then** return
2: **else**
3: S_1, S_2 sequenze vuote ▷ Divide
4: **for** $i \leftarrow 0$ **to** $\lceil n/2 \rceil - 1$ **do** $S_1[i] \leftarrow S[i]$
5: **for** $i \leftarrow \lceil n/2 \rceil$ **to** $n - 1$ **do** $S_2[i - \lceil n/2 \rceil] \leftarrow S[i]$
6: MergeSort(S_1) ▷ Conquer
7: MergeSort(S_2)
8: Merge(S_1, S_2, S) ▷ Combine

0: **Algoritmo:** MERGE(S_1, S_2, S)
 Input: sequenze ordinate S_1 e S_2 , sequenza vuota S
 Output: sequenza ordinata $S = S_1 \cup S_2$

1: $i \leftarrow 0, \quad j \leftarrow 0, \quad k \leftarrow 0$
2: **while** ($i < S_1.size()$ AND $j < S_2.size()$) **do**
3: **if** ($S_1[i] < S_2[j]$) **then** $S[k++] = S_1[i++]$
4: **else** $S[k++] = S_2[j++]$
5: **while** ($i < S_1.size()$) **do** $S[k++] = S_1[i++]$
6: **while** ($j < S_2.size()$) **do** $S[k++] = S_2[j++]$

Analisi

L'algoritmo si basa sul design pattern divide-et-impera. La complessità del metodo Merge è $\Theta(n)$. Analizzando l'albero della ricorsione si osserva che ad ogni livello la taglia aggregata di tutte le chiamate ricorsive è n , per cui ogni livello ha complessità aggregata $\Theta(n)$. L'albero della ricorsione è un albero binario completo con altezza $\Theta(\log n)$, per cui la complessità complessiva è $\Theta(n \log n)$.

11.3 QuickSort

Algoritmo

0: **Algoritmo:** QUICKSORT(S, a, b)
 Input: come da specifica, indici $0 \leq a, b \leq n - 1$
 Output: come da specifica

1: **if** ($a \geq b$) **then** return
2: **else**
3: $l \leftarrow \text{Partition}(S, a, b)$
4: QuickSort(S, a, l)
5: QuickSort($S, l + 1, b$)

0: **Algoritmo:** PARTITION(S, a, b)
 Input: sequenza S , indici $0 \leq a, b \leq n - 1$
 Output: indice l e sequenza S tale che $S[i] \leq S[l] \leq S[j]$ per $a \leq i < l < j \leq b$

1: $p \leftarrow S[b], \quad l \leftarrow a, \quad r \leftarrow b - 1$
2: **while** ($l \leq r$) **do**
3: **while** ($l \leq r$ AND $S[l] \leq p$) **do** $l \leftarrow l + 1$
4: **while** ($l \leq r$ AND $S[r] \geq p$) **do** $r \leftarrow r - 1$
5: **if** ($l \leq r$) **then** swap($S[l], S[r]$)
6: swap($S[l], S[b]$)
7: return l

Analisi

L'algoritmo si basa sul design patter divide-et-impera. La complessità del metodo Partition è $\Theta(b - a + 1)$. Analizzando l'albero della ricorsione si osserva che nel caso peggiore una chiave viene sempre sistemata, per cui la taglia aggregata diminuisce di 1 ad ogni livello. In totale ci saranno n livelli con taglia aggregata del livello i di $n - i$, per cui complessivamente si ha una complessità complessiva di $\Theta(n^2)$.

11.4 Randomized QuickSort

Il RandomizedQuickSort riprende quasi totalmente il QuickSort, solo che il valore sentinella p viene scelto in maniera casuale. In questo modo si ha maggiore probabilità di partizionare la sequenza a metà. Cambia solo il metodo Partition come segue:

1: $p \leftarrow S[b] \dots$ ▷ versione deterministica

1: $i \leftarrow \text{intero random in } [a, b]$ ▷ versione randomized
2: swap($S[i], S[b]$)
3: $p \leftarrow S[b] \dots$

In questo modo si ha, per un'istanza t_i , una complessità media $E[t_{i,RQS}] \in \Theta(n \log n)$ con probabilità $P(t_{i,RQS} \in \Theta(n \log n)) \geq 1 - 1/n$. Questo tipo di complessità viene definito complessità con alta probabilità.

11.5 InsertionSort

Algoritmo

0: **Algoritmo:** INSERTIONSORT(S)
 Input: come da specifica
 Output: come da specifica

1: **for** $i \leftarrow 1$ **to** $n - 1$ **do**
2: $curr \leftarrow S[i]$
3: $j \leftarrow i - 1$
4: **while** $(j \geq 0$ AND $S[j] \geq curr)$ **do**
5: $S[j + 1] \leftarrow S[j]$
6: $j \leftarrow j - 1$
7: $S[j + 1] \leftarrow curr$

Analisi

In prima approssimazione si nota che l'algoritmo ha complessità al caso pessimo $\Theta(n^2)$. Secondo un'analisi più attenta, dopo aver definito K il numero di inversioni, la complessità è $\Theta(n + K)$. In questo modo si osserva che se il numero delle inversioni è lineare nella taglia dei dati, allora l'algoritmo performa con complessità $\Theta(n)$. Questa situazione si verifica nel caso in cui la sequenza di input è quasi ordinata (es. inserimento di una chiave in una seq. ordinata).

Una inversione rispetto all'ordinamento crescente è una coppia di indici (i, j) con $0 \leq j < i \leq n - 1$ tale che $S[j] > S[i]$.

11.6 BucketSort

Algoritmo

0: **Algoritmo:** BUCKETSORT(S)
 Input: sequenza S con chiavi intere in $[0, N - 1]$
 Output: come da specifica

1: crea un array B di N code $B[0], B[1], \dots, B[N - 1]$
2: **for** $i \leftarrow 0$ **to** $n - 1$ **do**
3: $k \leftarrow S[i].getKey()$
4: $B[k].enqueue(S[i])$
5: $i \leftarrow 0$
6: **for** $k \leftarrow 0$ **to** $n - 1$ **do**
7: **while** $(!B[k].isEmpty())$ **do**
8: $S[i++] \leftarrow (B[k].dequeue())$
9: $k \leftarrow k + 1$

Analisi

La complessità di BucketSort è data dalla complessità dei due for: la prima scansione ha complessità $\Theta(n)$, la seconda $\Theta(N)$. La complessità complessiva è $\Theta(n + N)$. L'algoritmo è più vantaggioso dei precedenti algoritmi basati su confronti se $N \in O(n \log n)$. Il problema è che se N è molto grande, si potrebbe avere grande svantaggio sia dal punto di vista computazionale, che da quello spaziale.

11.7 RadixSort

Algoritmo

0: **Algoritmo:** RADIXSORT(S)

Input: sequenza S con chiavi $(c_{d-1}, c_{d-2}, \dots, c_1, c_0) \in [0, N^d - 1]$ con $c_i \in [0, N - 1]$

Output: come da specifica

1: **for** $i \leftarrow 0$ **to** $d - 1$ **do**

2: BucketSortMod(S, c_i) \triangleright significa ordina in modo stabile rispetto alla cifra c_i

Analisi

Ordinando in modo stabile, si avrà che alla fine di una generica iterazione i , le chiavi saranno ordinate secondo le i -esime cifre meno significative. Come algoritmo di ordinamento stabile si utilizza BucketSort modificato. La complessità risulterà, quindi, $\Theta(d(N + n))$, dove d è il numero delle cifre di una chiave, N è la base delle cifre delle chiavi e n è il numero di chiavi.

Se $d = 1$ si avrebbe la stessa complessità e i problemi del BucketSort, se invece si sfrutta la possibilità di ordinare per più cifre (es. ordino per 4 cifre alla volta), allora si avrebbe che la complessità diminuisce parecchio, specialmente per n molto grandi.

Ad esempio se io avessi un array con n chiavi con c cifre in $[0, n^c - 1]$, con BucketSort avrei complessità di $\Theta(n^d)$, svantaggiosa. Se scegliessi un certo N e d per cui $N^d - 1 \geq n^c - 1$, avrei complessità lineare. Per $d = c$, si avrebbe che $N = n$ e di conseguenza la complessità sarà $\Theta(cn)$, ovvero lineare.