

Alessandro Gerotto

Università degli studi di Udine

Docente *Carla Piazza*

Anno accademico 2022-23

Riassunto del corso di Algoritmi e Strutture dati

ALGORITMI DI ORDINAMENTO	7
Alcune precisazioni	8
Efficienza e analisi asintotica	8
Limite superiore	8
Limite inferiore	8
Limite stretto	8
Proprietá	9
Esempio	9
Costo e complessità	9
Stabilità	9
Note importanti	10
Insertion Sort	11
Costo e complessità	11
Correttezza	11
Selection sort	12
Merge Sort	13
La procedura Merge	13
Costo e complessità di merge sort	13
Esempio di esecuzione	14
Correttezza merge sort	14
Heap Sort	15
Complessità di heapSort	15
La procedura buildHeap	15
Complessità di buildheap	15
Correttezza di buildheap	16
Esempio	16
Quick Sort	17
Procedura partition	17
Complessità di quickSort	18
Correttezza di partition	18
Correttezza di quickSort	18
Esempio A=11, 15, 3, 5, 20, 1, 8, 25, 3, 10	19
Algoritmi di selezione	20
Il problema della selezione	20
SelectNaive	20
Complessità	20
Medians of median (Select)	21
Complessità di medians of median (Select)	21
Esempio di esecuzione del mom per il problema della selezione	21
Quick Sort ottimale	21
Ordinamento in tempo lineare 1: Counting Sort	22
Complessità	22
Esempio A={2, 5, 2, 7, 8, 8, 2, 3, 6}	22
Ordinamento in tempo lineare 2: Radix Sort	23
Esempio	23

Complessità	23
Ordinamento in tempo lineare 3: Bucket Sort	24
Complessità	24
STRUTTURE DATI	25
Alberi binari	26
Completezza	26
Bilanciamento	26
Altezza	26
Numero di nodi totali e foglie in un albero completo	26
Implementazione di code con priorità: max heap	27
Proprietà di una max heap	27
Implementazione di max heap	27
Operazioni sullo heap	27
Inserimenti	27
Cancellazione della radice (o chiave di priorità massima)	28
Incremento o decremento della priorità	28
La procedura Heapify	28
La correttezza di Heapify	29
Tabelle di Hash	30
Il problema	30
Esempio	30
Le soluzioni	30
Hash con Chaining	31
La funzione di hash	31
Funzioni di hash standard	31
Costi nei casi peggiori	32
Inserimento	32
Ricerca e cancellazione	32
Costi nei casi medi e ipotesi di hashing uniforme semplice	32
Esercizi sulle funzioni di hash	32
Hash con Open Addressing	34
Funzione di hash con sequenza di scansione	34
Funzioni di hash con sequenza di scansione standard	34
Inserimento, cancellazione e ricerca	35
Complessità ed Ipotesi di hashing uniforme	35
BST: Binary Search Tree	36
Procedure di visita	36
Visita pre-order 8-3-1-6-4-7-10-14-13	36
Visita in-order 1-3-4-6-7-8-10-13-14	36
Visita post-order 1-4-7-6-3-13-14-10-8	36
Complessità delle operazioni di visita	36
Esercizi sulle visite	37
Operazioni di ricerca	37
Ricerca del massimo	37
Ricerca di una chiave k	37
Ricerca del successore di una chiave k	38
Operazione di inserimento	38
Esempio di inserimento della chiave 5	38

Operazione di cancellazione	39
Esempio di cancellazione delle chiavi 7, 10, 3	39
Alberi rosso-neri	40
Definizione	40
Altezza ed altezza nera	40
Proprietà sull'altezza	40
Dimostrazione proprietà (3)	40
Dimostrazione che un RBT è bilanciato	40
Operazioni di rotazione (1)	41
Operazioni di inserimento O(log n)	41
Esempio	41
Operazioni di cancellazione O(log n)	41
Operazioni di ricerca O(log n)	43
Ricerca del massimo	43
Ricerca di una chiave k	43
Ricerca del successore/predecessore di una chiave k	43
Operazioni di fusione di due RBT O(log n)	43
Min heap vs RBT	44
B-Tree	45
Definizione dei BTree	45
Proprietà dei BTree	45
Altezza e altezza massima	45
Operazioni sui Btree	46
Operazioni di ricerca	46
Costi per la ricerca	46
Creazione albero vuoto	46
Operazioni di inserimento	47
Procedura di inserimento	47
Esempio di inserimento	48
Costo per l'inserimento	48
Operazioni di cancellazione	49
Gestione di insiemi disgiunti	50
1. MUF con liste concatenate	50
2. MUF con alberi	50
Union by rank	50
Path compression	51
Grafi non pesati	52
Memorizzazione di un grafo	52
Definizione cammino e distanza	52
Visite BFS - Ricerca in ampiezza	53
Complessità	54
Esempio	54
Proprietà	54
Esercizi	55
Visite DFS - Ricerca in profondità	55
Complessità	57
Esempio 1	57
Esempio 2	57

Proprietà	58
DAG: Grafi orientati aciclici	59
TP: Topological Sort	59
Algoritmo per la verifica della presenza di cicli	59
Algoritmo per trovare un TS	60
Cosa fare in presenza di cicli (nessun nodo senza archi uscenti)	61
Algoritmo per il calcolo delle scc (Kosaraju)	61
Grafi non orientati ed alberi	61
Grafi pesati	62
Albero minimo di copertura (MST)	62
Definizione di taglio	62
Definizione di leggero	62
Definizione di arco safe	62
Teorema	63
Domande interessanti	63
Algoritmo di Kruskal	64
Corollario	64
Algoritmo di Prim	65
Albero dei cammini minimi (SSSP e APSP).	66
Differenza tra MST e SSSP	66
Algoritmo di Dijkstra per il calcolo di SSSP	66
Algoritmo per il calcolo di APSP: floyd warshall	66
Esercizi primo parziale	67
[Merge] Esercizio 1.	67
[Merge] Esercizio 2.	67
[Merge] Esercizio 3.	68
[Select] Esercizio 4.	68
[Select] Esercizio 5.	69
[BST] Esercizio 6.	69
[BST] Esercizio 7.	69
[BST] Esercizio 8.	70
[BST] Esercizio 9.	70
[BST] Esercizio 10.	71
[BST] Esercizio 11.	71
[Tabelle di Hash] Esercizio 12.	71
[Tabelle di Hash] Esercizio 13.	71
[Tabelle di Hash] Esercizio 14.	72
[Tabelle di Hash] Esercizio 15.	72
[Heap] Esercizio 16.	73
[Ricerca binaria] Esercizio 17.	73
[Generico] Esercizio 17.	73

ALGORITMI DI ORDINAMENTO

Insertion Sort

Selection Sort

Merge Sort

Heap Sort

Quick Sort

Counting Sort

Radix Sort

Bucket Sort

Alcune precisazioni

Un algoritmo di ordinamento è un algoritmo che riceve in **input una sequenza di n interi** a_1, a_2, \dots, a_n e restituisce come **output una permutazione** degli n interi tale che siano ordinati dal più piccolo al più grande.

Un algoritmo di ordinamento può essere basato su:

- **scambi e confronti** come insertion sort, merge sort, heap sort e quick sort.
- **ipotesi sull'input** come counting sort, radix sort e bucket sort.

Efficienza e analisi asintotica

L'efficienza di un algoritmo viene stabilita in base ai suoi requisiti di **tempo**, ossia, si studia il suo comportamento asintotico nei casi migliore, medio e peggiore; L'efficienza viene valutata anche guardando i requisiti di **spazio**, che determinano se un algoritmo è **in-place** (la memoria allocata è **di dimensioni fisse**) o se è **non in-place** e quindi la memoria allocata **aumenta durante l'esecuzione**.

Il metodo utilizzato per studiare la complessità degli algoritmi è chiamato analisi asintotica. Tale metodo viene utilizzato per calcolare il **tasso di crescita** di una funzione basandosi sul confronto di altre funzioni di cui si conosce già la crescita. L'analisi della complessità di un algoritmo si basa su tre grandezze:

Limite superiore

$T(n) \in O(f(n))$: $\exists c, n_0 > 0$ t.c. $T(n) \leq c \cdot f(n) \forall n \geq n_0$

Studiando $T(n) = O(f(n))$ il tasso di crescita della complessità è **al massimo** $c \cdot f(n)$, ossia, nel calcolo di $O(f(n))$ si studia il comportamento dell'algoritmo nel caso peggiore. Se per esempio si assume che vale $f(n) \in O(g(n))$, stiamo dicendo che esiste un punto dopo il quale $f(n) \leq g(n)$ sempre.

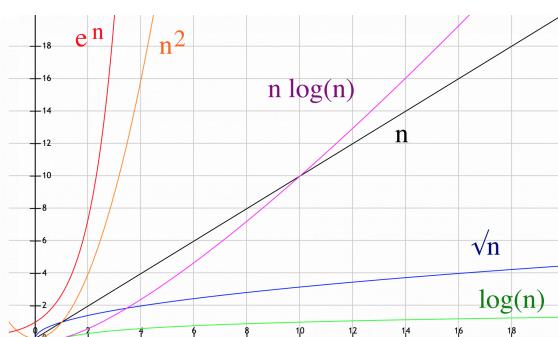
Limite inferiore

$T(n) \in \Omega(f(n))$: $\exists c, n_0 > 0$ t.c. $T(n) \geq c \cdot f(n) \forall n \geq n_0$

Studiando $T(n) = \Omega(f(n))$ il tasso di crescita della complessità è **al meno** $c \cdot f(n)$, ossia, nel calcolo di $\Omega(f(n))$ si studia il comportamento dell'algoritmo nel caso migliore. Se per esempio si assume che vale $g(n) \in \Omega(f(n))$, stiamo dicendo che esiste un punto dopo il quale $f(n) \geq g(n)$ sempre.

Limite stretto

$T(n) \in \Theta(f(n))$: $\exists c_1, c_2, n_0 > 0$ t.c. $c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n) \forall n \geq n_0$



Proprietà

La notazione asintotica gode di alcune **proprietà**:

1. $O(h(n) + g(n)) = O(h(n)) + O(g(n))$, $O(h(n) \cdot g(n)) = O(h(n)) \cdot O(g(n))$;
2. $f(n) \in O(g(n))$ sse $g(n) \in \Omega(f(n))$.
3. $f(n) \in O(g(n)) \wedge g(n) \in O(h(n)) \rightarrow f(n) \in O(h(n))$ (proprietà transitiva);
4. $f(n) \in O(f(n))$ (proprietà riflessiva);

Esempio

Se $f(n)$ è una funzione monotona non decrescente e $f(n) \in O(n^2)$, quali delle seguenti affermazioni sono vere?

- a. $f(n) \in O(n^2 \log n)$: vera: $f(n) \in O(n^2)$ e $n^2 \in O(n^2 \log n)$ per trans. $f(n) \in O(n^2 \log n)$.
- b. $f(n) \in O(n \log n)$: falsa: $f(n) \in O(n^2)$ ma $n^2 \notin O(n \log n)$.
- c. $f(n) \in O(\log n)$: falsa: $f(n) \in O(n^2)$ ma $n^2 \notin O(\log n)$.
- d. $3n^2 \in \Omega(f(n))$: vera: $f(n) \in O(n^2)$ e $n^2 \in O(3n^2)$ per transitività $f(n) \in O(3n^2)$.

Tieni presente che due polinomi dello stesso grado sono dello stesso ordine di grandezza, quindi Θ . Due polinomi di grado diverso quello più piccolo è O di quello più grande e viceversa quello più grande è Ω di quello più piccolo. Ad esempio:

$$\begin{array}{llll} n \in O(n^2) & 2n + 1 \in \Theta(n) & \log(n) \in O(\sqrt{n}) & 2^n \in O(3^n) \\ n^3 \in \Omega(n^2) & \log_2(n) \in \Theta(\log_3(n)) & \log(n!) \in \Theta(n \log(n)) & \end{array}$$

Costo e complessità

Per determinare la complessità di un algoritmo si associa ad ogni riga i del programma una **costante** c_i che indica il **costo** di esecuzione di **tal riga**. Il **costo totale** è quindi la somma delle costanti di riga per il numero di volte che tale riga viene eseguita. Nasce quindi un **problema** per stabilire il numero di esecuzioni delle righe che contengono cicli che non hanno il numero di iterazioni determinato a priori (A.e. in InsertionSort il primo for cicla per un numero di volte facilmente stabilibile, mentre il while cicla per un numero di volte non semplicemente determinabile). Per risolvere questo problema, si indica il numero di volte che tale riga viene eseguita con $t_j \in [0, j]$.

Stabilità

Un metodo di ordinamento si dice **stabile** se **preserva** l'ordine relativo dei dati con **chiavi uguali** all'interno del file da ordinare. Ad esempio se si ordina per anno di corso una lista di studenti già ordinata alfabeticamente un metodo stabile produce una lista in cui gli alunni dello stesso anno sono ancora in ordine alfabetico mentre un ordinamento instabile probabilmente produrrà una lista senza più alcuna traccia del precedente ordinamento.

Note importanti

$$\sum_{i=0}^n (x^a) = \frac{x^{n+1}-1}{x-1}$$

$$\sum_{i=0}^n (< 1) = cost$$

$$\sum_{i=0}^n (1) = n$$

$$\sum_{i=0}^n (c \cdot \log(i)) = n \log n$$

$T(n) = T(\alpha \cdot n) + T(\beta \cdot n) + \Theta(n) = \Theta(n)$ se $\alpha + \beta < 1$. (vedi medians of median)

$T(n) = T(\alpha \cdot n) + T(\beta \cdot n) + \Theta(n) = \Theta(n \log n)$ se $\alpha + \beta = 1$. (vedi mergesort)

$T(n) \leq \dots \Rightarrow$ il \leq si mette quando c'è la possibilità che non venga eseguita nessuna chiamata ricorsiva.

$\log_b a \geq 1$ se $b < a$

Algoritmi che si basano su scambi e confronti nel caso costano $\Omega(n \log n)$, ossia tutti almeno $n \log n$:

Un algoritmo ricorsivo è sempre non in place, ma se la ricorsione è eliminabile, l'algoritmo può essere in place

Insertion Sort

È un algoritmo di ordinamento **in-place** e **stabile**. Si assume che la sequenza da ordinare sia partizionata in una **sottosequenza già ordinata**, all'inizio composta da un solo elemento, e **una ancora da ordinare**. Alla k -esima iterazione, la sequenza già ordinata contiene k elementi. In ogni iterazione, viene **rimosso un elemento** dalla sottosequenza non ordinata e **inserito nella posizione corretta** della sottosequenza ordinata, estendendola così di un elemento. Per fare questo, un'implementazione tipica dell'algoritmo utilizza **due indici**. Il primo punta all'**elemento da ordinare** (inizialmente al secondo elemento); il secondo punta l'**elemento immediatamente precedente**. Se l'elemento puntato dal secondo indice è **maggior**e di quello a cui punta il primo indice, i due elementi vengono **scambiati di posto**; altrimenti il primo indice avanza. Il procedimento è ripetuto finché si trova nel punto in cui il valore del primo indice deve essere inserito.

```

InsertionSort(A)                                // Costi delle singole linee:
    for ( $j=2$  to  $A.\text{len}$ )                      //  $2cn$ 
         $k = A[j]$                                 //  $c(n-1)$ 
         $i = j-1$                                 //  $2c(n-1)$ 
        while ( $i \geq 0$  and  $a[i] > k$ )          //  $\sum_{j=2}^n t_j 2c$ 
             $a[i+1] = a[i]$                       //  $c\sum_{j=2}^n t_j (t_j - 1)$ 
             $i = i-1$                             //  $c\sum_{j=2}^n t_j (t_j - 1)$ 
         $A[i+1] = k$                             //  $c(n-1)$ 
    
```

Costo e complessità

$$T(n) = 2c \cdot n + 3c(n - 1) + 2c \cdot \sum_{j=2}^n t_j + 2c \cdot \sum_{j=2}^n (t_j - 1) = d \cdot n + e \cdot \sum_{j=1}^n t_j + f \text{ con } d, e, f \text{ costanti.}$$

In base a t_j l'algoritmo ha costo diverso e si divide in tre casi:

- **Migliore** ($t_j = 1$): non esegue mai il while $\rightarrow A$ ordinato: $T(n) = dn + en + f = \Theta(n)$;
- **Peggio** ($t_j = j$): esegue n volte for e while $\rightarrow A$ decrescente $T(n) = dn + e\sum_{j=2}^n (j) + f = \Theta(n^2)$;
- **Medio**: bisogna fare un ragionamento diverso: un vettore di lunghezza n ha $n!$ possibili disorientamenti. La probabilità di un generico disordinamento di essere l'input di Insertion Sort è $1/n$ (per distribuzione uniforme). Quindi il costo nel caso medio è $\Theta(n^2)$.

Correttezza

Bisogna dimostrare che **per ogni** vettore dato in input all'algoritmo, Insertion Sort termina con il vettore ordinato. Sfrutteremo l'induzione sul numero di elementi dell'array. Sia quindi n la cardinalità del vettore input:

$n = 1$ banalmente vero dato che un vettore con un elemento è ordinato per definizione;

$n > 1$ dopo le prime $n - 1$ iterazioni del ciclo for, $A[1, \dots, n - 1]$ è ordinato. Durante la n -esima iterazione il ciclo while termina solo quando la condizione non è più soddisfatta (quando il valore assegnato a key è inferiore al valore contenuto in $A[j]$ per $j > i + 1$). Quindi $A[n] = key$ viene inserito nella posizione corretta.

Selection sort

Il selectionSort è un algoritmo di ordinamento che opera **in place** ed in modo simile all'ordinamento per inserzione. Si parte dall'inizio e si **scorre tutto l'array** cercando il valore minimo. Successivamente si chiama la funzione di **swap** e si scambia di posizione il valore minimo trovato nell'iterazione precedente con quello in prima posizione. A questo punto si ripete questa operazione sul vettore senza considerare la prima posizione.

```
SelectionSort(A) {
    for (i = 1 to A.len()-1) {
        min = i
        for (j = i+1 to A.len()) {
            if (A[j] < A[min])
                min = j
        }
        swap(A, i, min)
    }
}
```

Merge Sort

Sfrutta la tecnica **divide-et-impera**: il problema viene diviso in sottoproblemi, che vengono risolti in maniera ricorsiva dividendosi ulteriormente. Questo procedimento viene ripetuto fino al raggiungimento del caso base, in cui i vari “pezzi” vengono riassemblati. **Non è in-place** perché utilizza un secondo vettore per ritornare l’output. È **stabile** solo se anche Merge lo è. Concettualmente, l’algoritmo funziona nel seguente modo:

1. Caso base: Se il vettore ha lunghezza 0 oppure 1, è **già ordinato**;
2. Il vettore viene **diviso in due** metà;
3. Ogni vettore viene **ordinato**, applicando **ricorsivamente** l’algoritmo (dividendo ulteriormente);
4. I due vettori ordinati vengono **fusi** con una procedura **merge** che confronta i primi elementi e inserisce il più grande in un **nuovo vettore** vuoto (che sarà l’output).

```
mergeSort(A, p, q)           // Θ(n log n)
    if (p < q)                // Θ(1)
        r = (p+q)/2            // Θ(1)
        mergeSort(A, p, r)      // T(n/2) → costo su metà elementi
        mergeSort(A, r+1, q)    // T(n/2) → costo su metà elementi
        merge(A, p, q, r)       // Θ(n)
```

La procedura Merge

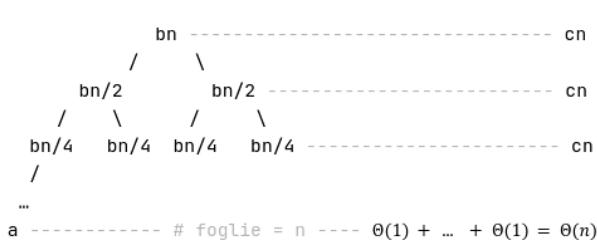
Ci sono **due vettori** A e B , di dimensione $q - p + 1 = n$ dove: A è composto da **due vettori ordinati** e giustapposti (precondizione) ($A[p..r]$ e $A[r+1..q]$) e B è inizialmente **vuoto**. Un ciclo **for** riempie B selezionando il valore **più piccolo** tra $A[i]$ e $A[j]$ (dove i inizialmente punta a $A[0]$ e j punta al primo valore della seconda porzione ordinata di A). Tale ciclo deve **verificare** ad ogni iterazione che gli indici siano **dentro** i rispettivi range ($i < r \wedge j < q$). Un **secondo for** si occupa poi di **copiare i valori** contenuti nel vettore B in A ;

Costo e complessità di merge sort

La complessità è definibile con la seguente **equazione ricorsiva di complessità**:

$$T(n) = \begin{cases} c & \text{se } n = 1 \\ 2T\left(\frac{n}{2}\right) + d \cdot n & \text{se } n > 1 \end{cases}$$

Ad ogni esecuzione di `mergeSort` si ha **due** chiamate ricorsive su **metà elementi**. Quindi il costo delle due chiamate ricorsive su se stesso, se sommato, sarà uguale a quello di `mergeSort`. Questo vale anche per le chiamate successive. Si crea quindi un **albero binario** di altezza $h = \log(n) + 1$ (il +1 è relativo ai **casi base**) dove ad ogni livello aumentano i nodi (ch. rec.) ma il **costo totale** rimane lo **stesso**:

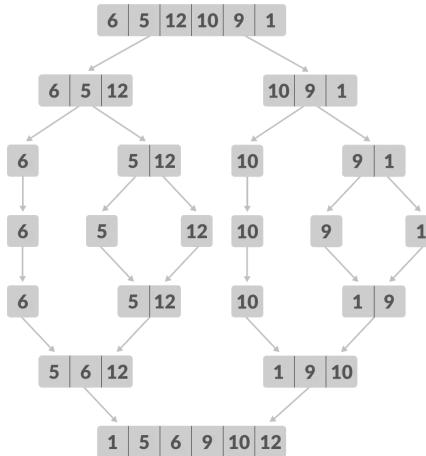


Avendo l’albero di altezza $\log(n)+1$, dove il costo per il singolo livello di chiamate ricorsive è bn , il suo costo sarà complessivo $bn \cdot \log(n)$ a cui si somma il costo del livello contenente i casi base (foglie). Il suo costo totale diventa:

$$bn \cdot \log(n) + an \rightarrow \Theta(n \log n).$$

Esempio di esecuzione

Passando come input alla procedura merge sort l'array $A = \{6, 5, 12, 10, 9, 1\}$ si ottiene il seguente grafico:



Correttezza merge sort

`mergeSort(A, p, q)` termina sempre e $A[p..q]$ al termine è ordinato.

Dimostrazione: per induzione sulla lunghezza della porzione di vettore da ordinare ($n = q - p + 1$).

Base: $n = 1$ (vettore con un solo elemento) quindi $p = q \rightarrow A[p..p]$ ordinato banalmente.

Passo: $n > 1$

ipotesi induttiva (so che funziona): `mergeSort(A, p, q)` termina con $A[p..q]$ ordinato se $q - p + 1 < n$.

tesi (da dimostrare): `mergeSort(A, p, q)` termina con $A[p..q]$ ordinato se $q - p + 1 = n$.

```

if (p < q)                                // il test ha successo
    r = (p+q)/2
    mergeSort(A, p, r)                      // per hp ind la chiamata termina con A[p..r] ordinato
    mergeSort(A, r+1, q)                    // per hp ind la chiamata termina con A[r+1..q] ordinato
    merge(A, p, q, r)                      // le due precondizioni per la correttezza di merge valgono
                                            // e dalla correttezza di merge ho che tale procedura termina
                                            // correttamente se valgono le precondizioni
  
```

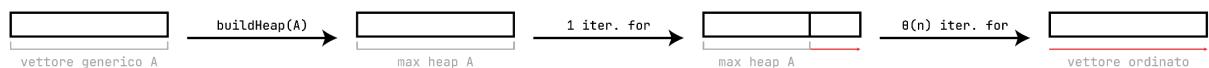
Heap Sort

È un algoritmo di ordinamento **in place** (heapify può essere riscritta in versione iterativa) e **non stabile**:

1. Costruisce un **array generico** contenente n elementi
2. Lo si rende una **max heap** con BuildHeap (in questo modo l'elemento max sta in prima posizione);
3. **Scambia** radice \leftrightarrow elemento in posizione i ;
4. **Decrementa** i ;
5. Se $i > 1$, si **ripete** il passo 2.

```
heapSort(A)                                //  $O(n \log n)$ 
    buildHeap(A)
    for(i = A.len downto 2) // eseguito  $\Theta(n)$  volte
        swap(A, 1, i)          //  $\Theta(1)$ 
        A.heapsize--
        heapify(A, 1)         //  $O(\log n) \rightarrow$  ripristina la heap
```

Ad ogni passo, il vettore è diviso in **due parti**: la prima è una **max heap**, la seconda è un **vettore ordinato**:



Complessità di heapSort

La procedura **heapify** costa $O(\log n)$ ed è in-place mentre, la procedura **buildHeap** costa $\Theta(n)$ ed è in-place (solo nella versione iterativa). Quindi **heapSort** ha costo $\Theta(n) + O(n \log n) = O(n \log n)$ che si divide in:

- **Caso migliore** (quando in A tutti gli elementi sono **uguali**) \rightarrow costo $\Theta(n)$.
- **Caso peggiore** \rightarrow costo $\Theta(n \log n)$;

La procedura buildHeap

La procedura **buildHeap** prende come input un vettore e ne **restituisce** la corrispettiva **max heap** (vettore che soddisfa la topologia delle max heap). L'idea per implementare l'algoritmo di buildheap è quella di chiamare **heapify** a partire dal primo nodo non foglia andando verso sinistra.

buildHeap(A) // $\Theta(n)$ ricorsivo, non in-place buildHeap(A[2]) buildHeap(A[3]) heapify(A, 1)	buildHeap(A) // $\Theta(n)$ iterativo, in-place A.heapsize = A.len for (i = floor (A.heapsize/2) downto 1) heapify(A, i)
--	--

Complessità di buildheap

La chiamata ad **heapify** ha costo $O(\log n)$ e viene eseguita circa $\Theta(n)$ volte. Quindi la complessità di buildheap è $O(n \log n)$. Tuttavia, si possono fare dei **conti più precisi** tenendo conto che 2^{h-j} nodi ossia, le foglie di un albero binario completo di altezza $h - j$, e su ognuno di questi nodi, il costo di **heapify** è $O(j) = c \cdot j$.

$$T(h) = \sum_{j=0}^h O(j) \cdot 2^{h-j} = \sum_{j=0}^h cj \cdot 2^{h-j} = c \cdot 2^h \sum_{j=0}^h \frac{j}{2^j} = O(2^h) \Rightarrow T(n) = O(n) = \Omega(n) \Rightarrow T(n) = \Theta(n).$$

Correttezza di buildheap

Lemma: `buildHeap(A)` termina con A max heap.

Invariante del ciclo for (cosa è vero all'inizio dell' i -esima iterazione del for): All'inizio dell' i -esima iterazione del for abbiamo $A[i + 1], A[i + 2], \dots, A[A.\text{len}]$ sono radici di max heap.

Dimostrazione per induzione su i

Base: $i = \text{floor}(A.\text{len}/2)$

$A[i + 1], A[i + 2], \dots, A[A.\text{len}]$ sono foglie, quindi radici di max heap.

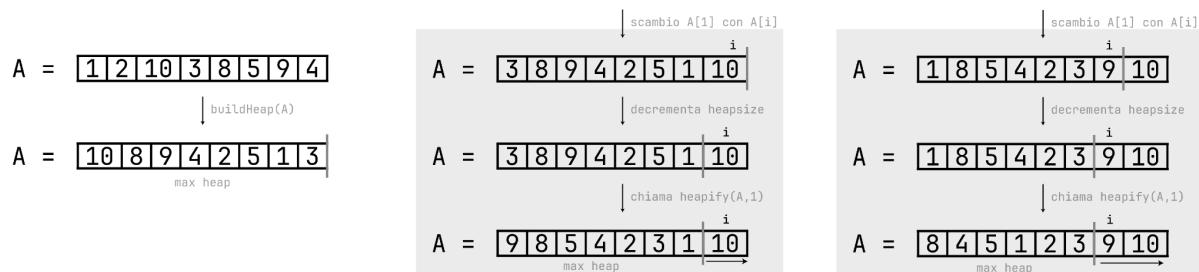
Passo:

ipotesi induttiva (so che funziona): Tutto funziona fino all'inizio dell' i -esima iterazione. All' i -esima iterazione $A[i + 1], A[i + 2], \dots, A[A.\text{len}]$ sono radici di max heap

tesi (da dimostrare): All'inizio dell'iterazione $i - 1$, $A[i], \dots, A[A.\text{len}]$ sono radici di max heap. Passando da un iterazione all'altra, si esegue `heapify(A, i)` e al termine A è una heap.

Per passare dall'ipotesi induttiva (corretta) alla tesi (da dimostrare) viene eseguita `heapify(A, i)` che è corretta.

Esempio

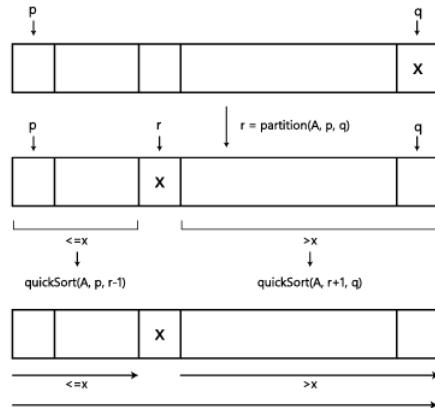


Quick Sort

`quickSort` è un algoritmo di ordinamento **non in place** e **non stabile** basato sul paradigma del *divide et impera*. Nella fase del *divide* si **sceglie** un elemento chiamato **pivot** (nota: la scelta influenza l'efficienza dell'algoritmo) che **suddivide** il vettore in due sottovettori:

- il sottovettore di **sinistra** contiene solo gli **elementi più piccoli** del pivot.
- il sottovettore di **destra** contiene solo gli **elementi più grandi** (o uguali) del pivot.

Durante la fase dell'*impera* si **ordina ricorsivamente** i due sottovettori, richiamando `quickSort` sui due vettori.



```

quickSort(A, p, q) // Θ(n)
  if(p < q)
    r = partition (A, p, q) // Θ(n)
    quickSort (A, p, r-1) // T(m)
    quickSort (A, r+1, q) // T(n-m-1)
  
```

```

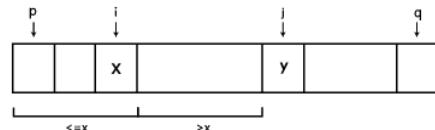
partition(A, p, q) // Θ(n)
  pivot = A[q]
  i = p - 1
  for (j = p to q)
    if(A[j] ≤ pivot)
      i++
      swap(A, i, j)
  return i
  
```

Procedura partition

In ogni istante, si usano **due indici** i e j . Nell'istante corrente, si assume per **vero** che i valori nell'intervallo:

- $[p .. i] \leq x$.
- $[i + 1 .. j - 1] > x$.

In questo modo basta scorrere la porzione $A[j..q]$, **scegliendo dove inserire** il valore in esame, chiamato y .



Ad esempio, all'istante corrente si sta analizzando il valore $A[j] = y$. Ci sono **due casi**:

- Se $y > x$: y si trova già nella posizione corretta e **basta incrementare** j ;
- Se $y \leq x$: **incremento** i e **scambio** $A[i]$ con $A[j]$.

Complessità di quickSort

Posto n come la lunghezza di A e m come la lunghezza del sottovettore sinistro, la complessità di quickSort è:

$$T(n) = \begin{cases} \theta(1) & \text{se } n \leq 1 \\ T(m) + T(n-m-1) + \theta(n) & \text{se } n > 1 \end{cases}$$

Di conseguenza, quickSort ha complessità diversa nei due casi:

- **Caso migliore e medio** (quando $m \approx n / ...$) \rightarrow costo $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$.
- **Caso peggiore** (quando $m = 0$ o m costante rispetto ad n) \rightarrow costo $\Theta(n^2)$;

Correttezza di partition

Lemma: $\text{partition}(A, p, q)$ termina restituendo r tale che $A[p .. r - 1] \leq A[r] < A[r + 1 .. q]$ (a sinistra di r ci siano solo valori minori o uguali ad $A[r]$ e a destra ci siano solo valori maggiori).

Invariante del ciclo for (cosa è vero all'inizio dell' j -esima iterazione del for): All'inizio dell' j -esima iterazione del for abbiamo $A[p .. i] \leq x < A[i + 1 .. j - 1]$.

Dimostrazione per induzione su j

Base: $j = p \rightarrow A[p .. p - 1] \leq x < A[p .. p - 1] \rightarrow$ le due porzioni sono vuote e quindi **banalmente vero**.

Passo:

ipotesi induttiva (so che funziona): Tutto funziona fino all'inizio dell' j -esima iterazione. All' j -esima iterazione $A[p .. i] \leq x < A[i + 1 .. j - 1]$

tesi (da dimostrare): All'inizio dell'iterazione $j + 1 \rightarrow A[p .. i] \leq x < A[i + 1 .. j + 1]$

Per passare dall'ipotesi induttiva (corretta) alla tesi (da dimostrare) viene eseguiti due casi:

1. **if**($A[j] \leq x$) \rightarrow incrementa i , fa uno swap e la tesi continua ad essere verificata.
2. **if**($A[j] > x$) \rightarrow non si fa nulla, e la tesi continua ad essere verificata

Correttezza di quickSort

Lemma: $\text{quickSort}(A, p, q)$ termina con $A[p .. q]$ ordinato.

Dimostrazione per induzione sul numero di chiamate ricorsive:

Base: $p \geq q$ (nessuna chiamata ricorsiva): $A[p .. q]$ contiene un solo elemento \rightarrow vettore **banalmente ordinato**.

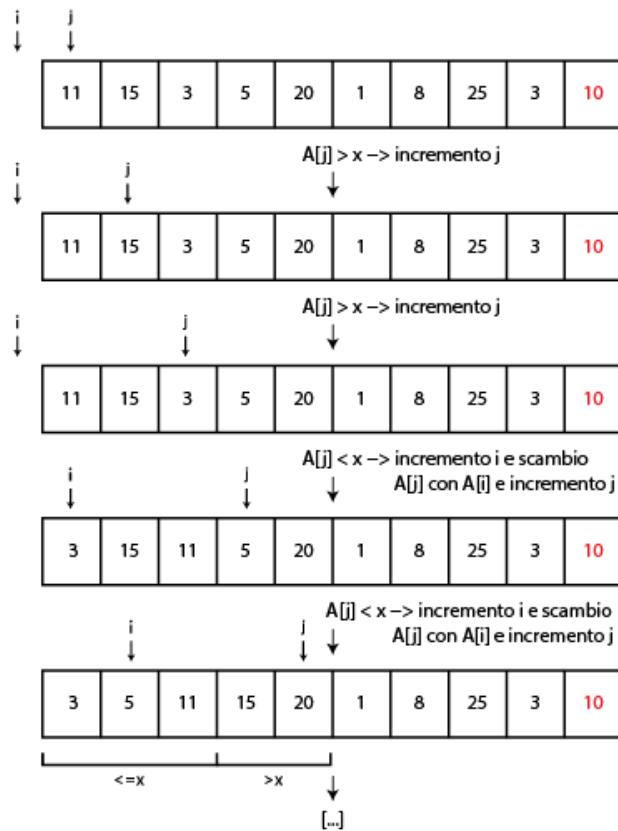
Passo:

ipotesi induttiva (so che funziona): il lemma vale fino al più k chiamate ricorsive.

tesi (da dimostrare): il lemma vale fino al più $k + 1$ chiamate ricorsive.

Facendo $k + 1$ Chiamate se ne sta facendo una in più rispetto a quelle dell'ipotesi induttiva, quindi sicuramente il test **if**($p < q$) è vero. Successivamente si esegue la chiamata a $\text{partition}(A, p, q)$ che termina sempre restituendo r tale che $A[p .. r - 1] \leq A[r] < A[r + 1 .. q]$ vista la correttezza dimostrata sopra. Facendo poi le due chiamate ricorsive partiranno al massimo $k - 1$ chiamate ricorsive, e per ipotesi induttiva la procedura è corretta.

Esempio $A = 11, 15, 3, 5, 20, 1, 8, 25, 3, 10$



Algoritmi di selezione

Il problema della selezione

Dato A vettore di lunghezza n e dato un indice $i \in [1..n]$ determinare l'elemento che finirebbe in posizione i se A fosse ordinato.

SelectNaive

Una prima soluzione per risolvere il problema sopra esposto sfrutta **partition** usando come perno r il valore $A[n]$, che in tempo **lineare** lo inserisce nella posizione corretta di A e fa in modo che tutti gli elementi alla sua sinistra siano minori o uguali a lui e quelli alla sua destra siano maggiori. Il problema si dirama in tre casi:

1. se $i < r$: richiama `selectNaive` su $A[p..r - 1]$.
2. se $i = r$: l'algoritmo termina restituendo $A[i]$.
3. se $i > r$: richiama `selectNaive` su $A[r + 1..q]$.

```
selectNaive(A, p, q, i)  // p ≤ i ≤ q
    if (p == q)
        return A[p]
    else
        r = partition(A, p, q)
        if (i == r)
            return A[r]
        if (i < r)
            return selectNaive(A, p, r-1, i)
        if (i > r)
            return selectNaive(A, r+1, q, i)
```

Complessità

Posto n come la lunghezza di A e m come la lunghezza del sottovettore sinistro, la complessità di `selectNaive` è:

$$T(n) = \begin{cases} \theta(1) & \text{se } n \leq 1 \\ T(m) + \theta(n) & \text{se } n > 1 \end{cases}$$

Di conseguenza, la complessità è variabile a seconda della dimensione della porzione sinistra m (e destra):

- **Caso migliore e medio** (quando $m = 0$ o quando $m \approx n / ...$) → costo $\Theta(n)$.
- **Caso peggiore** (quando m è del tipo $m = m - 1$) → costo $\Theta(n^2)$;

Come in [quickSort](#), per garantire che la complessità sia lineare bisogna prestare **attenzione alla scelta del perno**. Per avere sempre il miglior perno possibile, si sfrutta l'algoritmo medians of median.

Medians of median (Select)

É un algoritmo di selezione **non in place** e **non stabile**, utilizzato per fornire un **buon pivot** per un algoritmo di selezione esatta, più comunemente [quicksort](#) o [select](#), che seleziona l' i -esimo elemento più piccolo di un array inizialmente non ordinato. L'algoritmo Medians of median **trova una mediana approssimativa in tempo lineare**. Per scegliere il pivot che garantisce il costo migliore all'algoritmo di [quicksort](#) (e al problema della selezione) si sfrutta un algoritmo chiamato **median of medians** che:

- **Divide** il vettore in **sottovettori** di lunghezza 5 ($\Theta(1)$ per ogni blocchetto $\Rightarrow \Theta(n)$ avendo $n/5$ blocchi).
- **Ordina** i sottovettori, **estrae** ed **inserisce** il mediano di ognuno in B ($\Theta(n)$, dato che sono lunghi 5).
- **Cerca** in B il suo mediano, richiamando la procedura *mom ricorsivamente* ($T(n/5)$).

Complessità di medians of median (Select)

Posto n come la lunghezza di A e m come la lunghezza del sottovettore sinistro, il costo di `medianOfMedians` è:

$$T(n) = \begin{cases} \theta(1) & \text{se } n \leq 1 \\ T\left(\frac{n}{5}\right) + T(m) + \theta(n) & \text{se } n > 1 \end{cases}$$

Tenendo presente che:

$$T(n) = \begin{cases} \theta(1) & \text{se } n \leq 1 \\ T(\alpha \cdot n) + T(\beta \cdot n) + \theta(n) & \text{se } n > 1 \end{cases} = \begin{cases} \theta(1) & \text{se } n \leq 1 \\ \theta(n) & \text{se } n > 1 \end{cases} \quad \begin{array}{ll} \text{se } \alpha + \beta < 1 \rightarrow T(n) = \theta(n) \\ \text{se } \alpha + \beta = 1 \rightarrow T(n) = \theta(n \log n) \end{array}$$

Il costo di `medianOfMedians` diventa $\Theta(n)$:

$$T(n) = \begin{cases} \theta(1) & \text{se } n \leq 1 \\ T\left(\frac{n}{5}\right) + T\left(\frac{3n}{4}\right) + \theta(n) & \text{se } n > 1 \end{cases} \quad \text{dato che } \frac{n}{5} + \frac{3n}{4} < 1 \rightarrow T(n) = \theta(n)$$

Esempio di esecuzione del mom per il problema della selezione

Sia dato $A = 5, 6, 1, 2, 20, 15, 30, 48, 3, 4, 33, 12, 50$. Si vuole determinare l'elemento che finirebbe in posizione $i = 8$ se A è ordinato.

Dividiamo A in blocchi da 5 elementi: $A = 5, 6, 1, 2, 20 \ 15, 30, 48, 3, 4 \ 33, 12, 50$ e ordiniamo tali blocchi $A = 1, 2, 5, 6, 20 \ 3, 4, 15, 30, 48 \ 12, 33, 50$. Ora aggiungo i mediani di ogni blocco ad un nuovo vettore $B = 5, 15, 33$ e richiamo ricorsivamente medians of median su di esso, che viene diviso in blocchi da 5 (un unico blocco in questo caso) e ritorna il mediano, ossia 15, che sarà il **perno**.

Quick Sort ottimale

<pre>quickSortOttimale(A, p, q) // Θ(n log n) if(p < q) x = mediansOfMedian(A,p,q,(q-p+1)/2) // Θ(n) r = partition (A, p, q, x) // Θ(n) quickSortOttimale (A, p, r-1) // T(n/2) quickSortOttimale (A, r+1, q) // T(n/2)</pre>	<pre>partition(A, p, q, x) // Θ(n) i = p - 1 for (j = p to q) if(A[j] ≤ x) i++ swap(A, i, j) return i</pre>
--	---

Nota. ha la stessa equazione di complessità di [mergeSort](#) dove però α e β valgono esattamente $1/2$.

Ordinamento in tempo lineare 1: Counting Sort

É un algoritmo **stabile** e **non in place**. `contingSort` ha complessità $\Theta(n)$ sse vale la seguente **ipotesi sull'input**:

“il vettore A é un vettore di interi compresi tra 0 e k con $k \in O(n)$ ”

Questo viene dal fatto che $\exists n_0 \exists c > 0 \forall n \geq n_0 \quad k \leq cn$, quindi il numero piú grande che compare nel vettore è sicuramente minore o uguale ad cn e quindi certamente $cn \in O(n)$.

L'algoritmo **conta** il numero di **occorrenze** di ciascun valore presente nell'array da ordinare, memorizzando questa informazione in un array temporaneo di dimensione pari all'intervalle di valori. Il numero di ripetizioni dei valori inferiori indica la posizione del valore immediatamente successivo:

1. **Inizializza** il vettore delle occorrenze C di lunghezza $k + 1$ con tutti 0.
2. **Scandisce** A con un'indice i , per ogni valore scindito $A[i]$ faccio $C[A[i]] + 1$.
3. **Modifica** C in modo che in $C[j]$ sia scritto quanti elementi minori o uguali a j si trovano in A .
4. **Ottiene** B scandendo A da dx a sx (stabilitá) inserendo $A[i]$ in $C[A[i]]$ in B e decrementando $C[A[i]]$.

```
contingSort(A, B, k)           // Θ(n+k) → k ∈ O(n) per ipotesi sull'input → Θ(n)
    c = new array(k)             // Θ(k) ← C[0..k]
    for (j=0 to k)              // Θ(k)
        c[j] = 0
    for (j=1 to A.len())         // Θ(n)
        C[A[i]] = C[A[i]] + 1
    for (j=1 to k)              // Θ(k)
        C[j] = C[j] + C[j-1]
    for (i=A.len() down to 1)    // Θ(n) Andando da dx a sx, si garantisce la stabilitá
        B[C[A[i]]] = A[i]
        C[A[i]] = C[A[i]] - 1
```

Complessità

$T(n) = \Theta(n + k)$. Per ipotesi sull'input ($k \in O(n)$) → il costo è $\Theta(n)$. Se invece hp sull'input fosse stata:

- $k \leq n^2$ allora avrebbe avuto costo $\Theta(n + n^2) = \Theta(n^2)$.
- $k \leq n \log(\log n)$ allora avrebbe avuto costo $\Theta(n + n \log(\log n))$ (comunque migliore di merge sort).

Esempio $A = \{2, 5, 2, 7, 8, 8, 2, 3, 6\}$

Con una prima scansione di A individuo il valore minimo (2) e il valore massimo (8). Con la seconda scansione popolo l'array C , che tiene il conto delle frequenze di ciascun elemento nell'intervalle [2, 8] ottenendo il vettore $C = \{3, 1, 0, 1, 1, 1, 2\}$. Il significato dell'array C è il seguente: l'intero $0 + \min(A) = 2$ è presente 3 volte nell'array A , l'intero $1 + \min(A) = 3$ è presente 1 volta, [...]. Ora, inserisco in A tre volte l'intero 2, 1 volta l'intero 3, 0 volte l'intero 4, 1 volta l'intero 5 e così via. Dopo una terza scansione, l'array A é ordinato: $A = \{2, 2, 2, 3, 5, 6, 7, 8, 8\}$.

Ordinamento in tempo lineare 2: Radix Sort

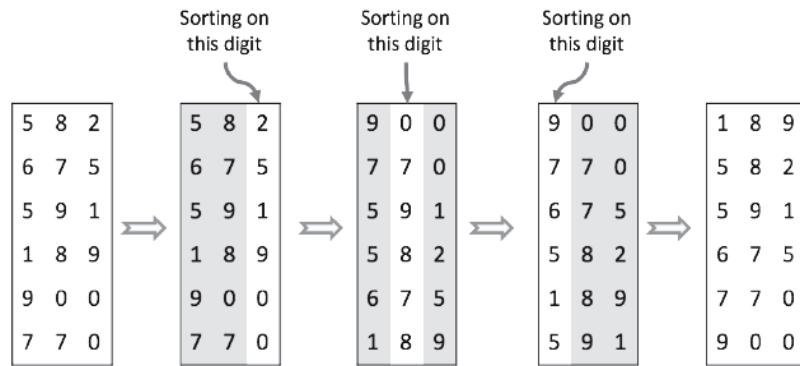
È un algoritmo **stabile** (se l'algoritmo per ordinare sulla singola cifra lo è) e **in place** (se l'algoritmo per ordinare sulla singola cifra lo è). radixSort ha complessità $\Theta(n)$ sse vale la seguente **ipotesi sull'input**:

"il vettore A è un vettore di interi di lunghezza n contenente interi aventi d cifre"

L'idea di radixsort è quella di ordinare gli elementi dell'array guardando le **cifre** dei singoli interi **da destra verso a sinistra**, partendo quindi dalle unità.

```
radixSort(A, d)
for (i=1 to d)
    // usa un ordinamento stabile per ordinare l'array A sulla cifra i
```

Esempio



Complessità

Radix sorta complessità $O(n \cdot d)$, dove d è la **media del numero di cifre** degli n elementi del vettore. La complessità è dunque **variabile** in base al valore d :

- Se d è costante $\rightarrow \Theta(n)$.
- Se $d \leq n \rightarrow \log(\log n) \rightarrow \Theta(n \cdot \log(\log n))$.
- Se $k > n$, radixsort peggiora rispetto ad algoritmi a tempo quasi lineare (a.e. quicksort/mergesort).

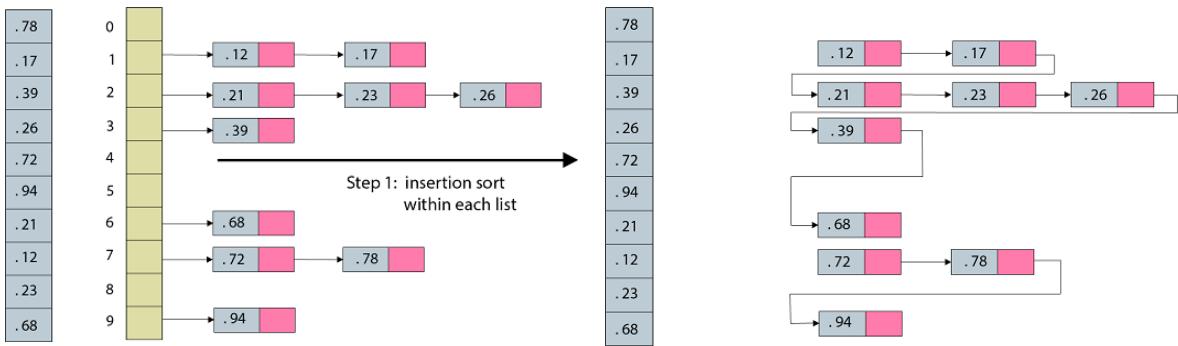
Ordinamento in tempo lineare 3: Bucket Sort

È un algoritmo **stabile** (se l'algoritmo usato al suo interno lo è) e **non in place**. `bucketSort` ha complessità $\Theta(n)$ se vale la seguente **ipotesi sull'input**:

“il vettore A è un vettore di interi di lunghezza n contenente valori numerici compresi nel range [0, 1) che si assume siano distribuiti uniformemente nell’intervallo [0, 1]”

Ciò significa che se n è il numero di elementi del vettore, l'intervallo $[0, 1)$ è diviso in n intervalli di uguale lunghezza, detti **bucket** (cesto). Ciascun valore dell'array è quindi **inserito nel bucket** a cui appartiene, i valori all'interno di ogni bucket **vengono ordinati** e l'algoritmo si conclude con la **concatenazione** dei valori contenuti nei bucket:

1. **Divide** $[0, 1)$ in n sottointervalli.
2. **Inserisce** ogni elemento di A nel corrispettivo sottointervalllo.
3. **Ordina** le liste concatenate (gli elementi di ogni sottointervalllo) in $\Theta(n)$ (ordino n elementi in $\Theta(1)$)
4. **Concatena** gli elementi dei sottointervalloli.



Complessità

Di conseguenza, la complessità di `bucketSort` su un vettore di lunghezza n è divisa nei casi:

- **Caso migliore e medio** (vale l'hp e si usa un algoritmo di ordinamento a costo $\Theta(n)$) \rightarrow costo $\Theta(n)$.
- **Caso peggiore** (vale l'hp e tutti gli elementi vengono inseriti in un'unica lista) \rightarrow costo $\Theta(n^2)$;

STRUTTURE DATI

Alberi binari

Code con priorità: max heap

Tabella di Hash

BST: Binary Search Tree

RBT: Red Black Tree

B-tree

Insiemi disgiunti

Grafi

MST: Minimum Spanning Tree

Alberi binari

Un **albero binario** è una struttura dati **astratta e dinamica** (è possibile allocare nuove celle di memoria in corso d'esecuzione) in cui gli elementi sono **oggetti** chiamati **nodi** e ognuno di essi contiene:

- Una **chiave key**: numero intero contenuto nel nodo;
- Un **puntatore al genitore** (null se non esiste);
- Un **puntatore al figlio sinistro** (null se non esiste);
- Un **puntatore al figlio destro** (null se non esiste);

Completezza

Un albero binario è definito **completo** quando tutti i livelli sono pieni. In altre parole, le foglie stanno tutte sullo stesso livello; Viene chiamato **quasi completo** se è un albero binario completo fino al penultimo livello, e l'ultimo livello è riempito da sx a dx.

Bilanciamento

Un albero è **bilanciato** se $h = \Theta(\log(n))$.

Altezza

L'altezza di un albero binario corrisponde al **cammino radice-foglia più lungo** (quanti archi/puntatori vengono attraversati). Ad esempio:

- Un albero con un solo nodo (root) ha altezza $h = 0$;
- Un albero con un unico figlio sinistro ha altezza $h = 1$;
- Un albero con un figlio sinistro e un figlio destro ha altezza $h = 1$;

Numero di nodi totali e foglie in un albero completo

- Il numero di **nodi** di un albero binario di altezza h è $2^{h+1} - 1$;

Il numero di foglie di un albero binario completo di altezza h è 2^h ($\approx 50\%$ di tutti i nodi).

Implementazione di code con priorità: max heap

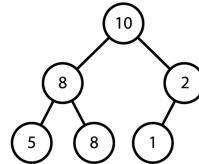
Una **max heap binaria** è un **albero binario quasi completo** (tutti i livelli dell'albero sono riempiti fino al penultimo livello che è riempito da sinistra verso destra) contenente n chiavi dove ogni nodo ha due figli che sono **minori o uguali** a lui. Formalmente si scrive come $\text{key}(x) \geq \text{key}(\text{left}(x)), \text{key}(\text{right}(x))$.

Proprietà di una max heap

1. L'elemento **massimo** si trova nella **radice** e la sua ricerca costa $\Theta(1)$.
2. L'elemento **minimo** si trova nelle **foglie**.
3. Gli elementi da $H.\text{heapsize}/2$ ad $H.\text{heapsize}$ sono **foglie**.
4. Una heap con n nodi ha **\Theta(\log n)**.

Implementazione di max heap

L'implementazione di max heap viene fatta usando **vettori sovrdimensionati**. La **radice** viene inserita nel **primo elemento** dell'array. Poi si inserisce ogni elemento del **secondo livello** della heap leggendo le key di tale livello **da sinistra verso destra**. Una volta terminate le chiavi del secondo, si passa al terzo livello e così via, fino al raggiungimento dell'ultima foglia. A.e. la heap qui sotto rappresenta l'array $H = \{10, 8, 2, 5, 8, 1\}$:



Operazioni sullo heap

Tenendo conto che tale array inizia all'indice 1, per accedere al nodo $H[i]$ della heap, si ha che:

<code>int parent(i){ return i/2 }</code>	<code>int left(i){ return 2i }</code>	<code>int right(i){ return 2i+1 }</code>
--	---------------------------------------	--

Inserimenti

1. **Inserisce** l'elemento p nella prima posizione disponibile rispettando la topologia degli alberi binari.
2. **Controlla** se $p \leq \text{parent}(p)$.
 - **Vero** → non serve fare nulla.
 - **Falso** → scambia p e $\text{parent}(p)$ e torna ricorsivamente al punto 2.

<pre>MaxHeapInsert (H, k) // O(log n) dove n è il numero di elementi nella max heap if (H.heapsize < H.length) // H.length è la lunghezza del vettore sovrdimensionato H.heapsize ++ H[H.heapsize] = k // H.heapsize è un numero compreso tra 0 e H.length e rappresenta l'effettiva porzione di H che viene usata i = H.heapsize while (i > 1 && H[i] > H[parent(i)]) // eseguito O(log n) volte swap (H, i, parent(i)) // Θ(1) i = parent(i) // Θ(1) else return "error"</pre>
--

Cancellazione della radice (o chiave di priorità massima)

1. **Scambia** chiave della **radice** ↔ chiave della **foglia più a destra**.
2. **Elimina** il vecchio valore max (ossia la foglia più a destra) **decrementando** heapsize.
3. **Controlla** se la nuova radice ha dei figli più grandi (richiama heapify):
 - **Vero** → scambia chiave della **nuova radice** ↔ **chiave maggiore** dei suoi **figli**.
ritorna al punto 3 ragionando sul sottoalbero radicato che aveva la chiave maggiore prima dello scambio.
 - **Falso** → non serve fare nulla.

```
MaxHeapRemoveMax(H)           // O(log n)
if (H.heapsize > 0)
    swap(H, 1, H.heapsize)   // scambia la key di root con quella della foglia più a dx.
    H.heapsize--              // elimina il vecchio max decrementando heapsize.
    heapify(H, 1)            // O(log n) esegue i test del punto 3
    return H[H.heapsize+1]    // ritorna il vecchio valore massimo
else
    return "error"
```

Incremento o decremento della priorità

Data la heap H , Se per esempio si vuole **incrementare** di 20 la priorità dell'elemento $H[5] = 8$, basta

- **sommare** a tale elemento 20: $H[5] = 28$
- **ordinare** il vettore in modo che soddisfi le proprietà della heap.

Analogamente si può fare nel caso di **decremento** della priorità.

La procedura Heapify

La procedura **heapify(A,i)** assume che il **figlio sinistro** $A[2i]$ e il **figlio destro** $A[2i + 1]$ del nodo i siano delle **radici di max heap** (precondizioni).

Se queste precondizioni sono soddisfatte heapify controlla se $A[i]$ è più piccolo di uno dei suoi figli. Se questo è il caso, la procedura fa **scambia** l'elemento $A[i]$ con il maggiore dei suoi figli e **richiama** se stessa ricorsivamente facendo scivolare quella che prima era la radice lungo un cammino dell'albero in modo da ristabilire la proprietà degli heap.

```
void heapify(H, i)      // O(log n)
l = left(i)
r = right(i)
if(l <= H.heapsize && H[l] > H[i])
    m = l
else
    m = i
if(r <= H.heapsize && A[r] > H[m])
    m = r;
if(m != i)
    swap(H, i, m);
    heapify(H, m);
```

La correttezza di Heapify

Lemma: `heapify(H, i)` termina con $H[i]$ radice di max heap se $H[2i]$ e $H[2i + 1]$ sono radici di max heap.

Dimostrazione: per induzione sul numero di chiamate ricorsive.

Base: $m = i$ (nessuna chiamata ricorsiva) Il test `if(m ≠ i)` fallisce \rightarrow (= falso del pt. 3).

Passo: $n > 1$

ipotesi induttiva (so che funziona): `heapify(H, i)` è corretta se vengono effettuate al più r chiamate ricorsive.

tesi (da dimostrare): `heapify(H, i)` è corretta se vengono effettuate al più $r + 1$ chiamate ricorsive.

Fatto lo scambio tra la vecchia radice (chiamiamola k) e il massimo dei suoi figli (chiamiamola a), il sottoalbero radicato nel figlio b rappresenta una max heap e per tanto non serve toccarlo. La chiamata ricorsiva fatta su k (dopo lo scambio, quindi k è figlio di a e fratello di b ora) rappresenta la $r + 1$ -esima chiamata ricorsiva e a sua volta attiva altre r chiamate ricorsive che per ipotesi induttiva sono corrette.

Tabelle di Hash

Il problema

Immaginiamo di avere un insieme **universo** U di cardinalità m in cui ogni elemento (che può contenere diversi campi) è identificabile tramite una **chiave univoca** $x.key$. In ogni istante è necessario gestire (mantenere in memoria e usare) un **sottoinsieme** K di cardinalità n che **varia dinamicamente** nel tempo. Che struttura dati conviene usare per implementare questa situazione, se si vuole che essa supporti **efficientemente** inserimenti, cancellazioni e ricerca sull'insieme K ?

Esempio

Immaginiamo di avere un insieme U rappresentante tutti gli studenti totali UniUd ed un insieme K degli studenti che frequentano il corso di ASD. K è molto più piccolo di U e varia dinamicamente nel tempo, perché ogni anno nuovi studenti si iscrivono al corso di ASD e altri studenti superano l'esame e quindi terminano il corso. Se si volesse mantenere in memoria solo l'insieme K , in cui gli studenti vengono identificati usando il loro relativo numero di matricola come chiave univoca, si può adottare diverse soluzioni:

Le soluzioni

1. Se si volesse usare un **vettore ad indirizzamento diretto** contenente tutti gli studenti dell'universo, contrassegnando con *nil* le posizioni occupate da studenti non iscritti ad ASD, si avrebbero:

- **Inserimenti** a costo $\Theta(1)$,
- **Cancellazioni** a costo $\Theta(1)$,
- **Ricerca** a costo $\Theta(1)$.

Tuttavia in termini di **spazio**, un tale vettore avrebbe un costo $\Theta(m)$ (ricordando che m è la cardinalità dell'universo). Questa rappresenta un'ottima soluzione in termini di tempo ma pessima in termini di spazio.

2. Se invece si volesse usare una **lista concatenata** in cui in ogni istante un elemento della lista è uno studente che ha un puntatore al successivo, si avrebbe una lista in cui ad ogni istante si memorizzano solo gli studenti frequentanti il corso di ASD. In questo modo l'occupazione in termini di **spazio** è esattamente dell'ordine della cardinalità di K , ossia un costo di $\Theta(n)$ ma si avranno le operazioni di:

- **Inserimenti** a costo $\Theta(1)$, *(si inserisce sempre in testa)*
- **Cancellazioni** a costo $O(n)$, *(nel caso peggiore si scorre tutta la lista per trovare un elemento)*
- **Ricerca** a costo $O(n)$. *(nel caso peggiore si scorre tutta la lista per trovare un elemento)*

Pertanto questa rappresenta una pessima soluzione in termini di tempo ma ottima in termini di spazio.

3. La struttura dati corretta da usare è una **tavella di hash** le cui possibili implementazioni sono due:

- a. Hash con Chaining.
- b. Hash con Open Addressing.

Hash con Chaining

Si usa un **vettore di liste concatenate** T (con cardinalità m) per memorizzare gli elementi del sottoinsieme K . Sorge però un dubbio: come **decidere dove** inserire un elemento in queste liste concatenate? Per determinare la posizione di inserimento di un nuovo elemento nella tabella di hash, data la chiave di tale elemento, si usa la formula $x.key \ mod \ |K|$.

La funzione di hash

La funzione di hash è una funzione $h: U \rightarrow \{0, \dots, |T| - 1\}$. Tale funzione, passandogli come input la chiave di un elemento dell'universo, restituisce un numero compreso tra 0 ed $|T|$, ossia $h(x.key) \in \{0, \dots, |T| - 1\}$.

- L'elemento x viene inserito nella lista $T[h(x.key)]$,
- Per cercare x scandisco la lista $T[h(x.key)]$,
- Per cancellare x lo cerco e lo eliminando $T[h(x.key)]$.

Sicuramente questa sorta di *mapping* dà luogo a delle **collisioni**, dato che la funzione di hash h **non** è una funzione **iniettiva** (dato che $|K| < |U|$). Il che vuol dire che *sicuramente* esistono due elementi differenti x e y appartenenti all'universo tali che $h(x.key) = h(y.key)$, ossia che tali elementi finiscono nella stessa posizione della tabella di hash. Nell'hash con chaining le liste concatenate **gestiscono** le collisioni inserendo elementi che finiscono nella stessa posizione di T uno di seguito all'altro nella stessa lista concatenata.

Una **buona** funzione di hash deve garantire:

1. **ogni** elemento dell'universo venga **mappato** in T e quindi che $h(x.key) \in \{0, \dots, |T| - 1\}$,
2. **equiprobabilità**. Circa $|U| / m$ elementi di U per ogni posizione di T ,
3. **minimo** numero di **collisioni** e quindi $\forall j \in \{0, \dots, |T|\} \exists x \text{ t.c. } h(x.key) = j$. In altri termini deve valere l'hp di hashing uniforme semplice e quindi h per essere una buona funzione di hash deve essere **suriettiva**.

Funzioni di hash standard

1. **Metodo della divisione**: le chiavi k intere degli elementi di U , vengono mappate tra 0 ed $m - 1$:

$$h(k) = k \cdot \text{mod. } m$$

È una buona funzione di hash perché garantisce la **suriettività** (essendo $|U| \gg m$), ma la distribuzione dipende da m : si cerca di avere m numero primo ed m lontano dalla potenza di 2.

2. **Metodo della moltiplicazione**: se le chiavi sono $0 \leq k < 1$, vengono mappate tra 0 ed $m - 1$:

$$h(k) = \text{floor}[km]$$

Mentre se invece le chiavi k sono dei valori interi nell'intervallo $0 \leq k < |U| - 1$, si sceglie un numero R compreso tra 0 e 1. Data una chiave k , la funzione di hash diventa:

$$h(k) = \text{floor}[(kR - \text{floor}[kR]) \cdot m]$$

Dove $kR - \text{floor}[kR]$ è sempre $\in [0, 1)$ e quindi ci si può ricondurre al metodo della moltiplicazione. È una buona funzione di hash in quanto garantisce l'**uniformità**. Altra caratteristica positiva è che m non influenza il risultato.

Nota: la prima parte di **bucket sort**, non è altro che la memorizzazione degli elementi del vettore in una tabella di hash usando come funzione il metodo della moltiplicazione (per questo bucket sort ha tempo lineare nel caso medio). Infatti dire che “il vettore è composto da elementi uniformemente distribuiti in $[0, 1)$ ” è come dire che soddisfa l'hp di hashing uniforme.

Costi nei casi peggiori

Inserimento

Nel caso peggiore, un inserimento di un elemento x in tabella di hash richiede di accedere alla sua chiave con $x.key$ e calcolarne la funzione di hash con $h(x.key)$. Questa operazione ha costo $\Theta(1)$. Fatto ciò, bisogna inserire x in testa alla rispettiva lista con anche qui costo $\Theta(1)$.

Ricerca e cancellazione

Nel caso peggiore, ossia quando tutti gli elementi inseriti in K sono finiti nella stessa posizione in T (e quindi nella stessa lista concatenata), per fare la ricerca o la cancellazione è necessario scorrere **tutta** la lista concatenata. Sapendo che n sono gli elementi in T in un certo istante, il costo di tali operazioni è $\Theta(n)$.

Costi nei casi medi e ipotesi di hashing uniforme semplice

Il costo nel caso medio può essere buono solo con l'ausilio di un'ipotesi, chiamata **ipotesi di hashing uniforme semplice**. Tale ipotesi afferma che:

“Estratto dall'universo un elemento casuale x senza guardarne la chiave, esso ha uguale probabilità ($1/n$) di finire in una qualsiasi posizione della tabella ($T[i]$)”

Indichiamo con $\alpha = |K|/|T|$ il **fattore di carico** che indica che in ogni lista della tabella contiene α elementi.

Teo: in una tabella di hash in cui le collisioni vengono gestite con chaining e vale l'ipotesi di hashing uniforme semplice, la **ricerca** di x , (sia con esito positivo che con esito negativo), costa in media $\Theta(1 + \alpha)$ dove il valore 1 rappresenta l'accesso al vettore (prima di accedere alla lista) e il calcolo $h(x.key)$. Nota:

- Se $|T| \geq |K|$ allora $\alpha \leq 1$ e il costo di **inserimenti, ricerca e cancellazione** vale $\Theta(1 + \alpha) = \Theta(1)$,
- Se $|T| < |K|$ allora $\alpha > 1$, e il costo di **inserimenti, ricerca e cancellazione** vale $\Theta(1 + \alpha) = \Theta(\alpha)$.

Dim: nel caso in cui x non viene trovato si calcola $h(x.key)$ con costo $\Theta(1)$ e si scandisce tutta la lista concatenata in posizione $T[h(x.key)]$ con costo $\Theta(1 + \alpha)$, in cui α rappresenta la lunghezza media della lista (dato che vale l'hyp di hashing uniforme). Nel caso in cui x viene invece trovato ed inserito in T come i -esimo elemento, dopo x sono stati inseriti in maniera uniforme $n - i$ elementi sulle n liste di cui $(n - i)/m$ elementi sono finiti nella stessa lista e davanti ad x . Dunque il costo della ricerca con successo di x è $\Theta(1) + (n - i)/m + 1$. La media (uniforme) su tutti gli elementi inseriti in tabella è

$$\sum_{i=1}^n \frac{1}{n} (\Theta(1) + \frac{n-i}{m} + 1) = [...] = \Theta(1 + \alpha).$$

Esercizi sulle funzioni di hash

Valutare se la data funzione di hash è una buona funzione o no. Per farlo, si procede ponendosi le domande:

1. c'è la possibilità di uscire dalla tabella?
2. per ogni possibile posizione della tabella, c'è almeno un elemento che finirà in quella posizione?

1. $h(k) = (k \cdot \text{mod. } m) + 3$.

Non è una buona funzione di hash perché potrei ottenere $m, m+1, m+2$ e quindi uscire dalla tabella.

2. $h(k) = (k \cdot \text{mod.}(m - 3)) + 3$.

$k \cdot \text{mod.}(m - 3)$ restituisce sempre un risultato compreso tra 0 e $m - 4$, quindi anche aggiungendogli 3 non si esce mai dalla tabella. Tuttavia $0 + 3 = 3$, ossia le posizioni $T[0]$, $T[1]$ e $T[2]$ della tabella resteranno sempre vuote. Quindi non è una buona funzione di hash.

3. $h(k) = 2k \cdot \text{mod.} m$.

Il $\text{mod.} m$ mi assicura che non si possa mai uscire dalla tabella. Tuttavia se $m = 2$, $T[1]$ non viene mai usata e se m è una potenza di 2 vengono usate solo posizioni pari. Quindi non è una buona funzione di hash.

4. $h(k) = (\max\{n - m, k\}) \cdot \text{mod.} m$. con n prefissato

Il $\text{mod.} m$ mi assicura che non si possa mai uscire dalla tabella. Tuttavia la posizione $n - m \cdot \text{mod.} m$ è molto più probabile rispetto alle altre. Quindi non è una buona funzione di hash.

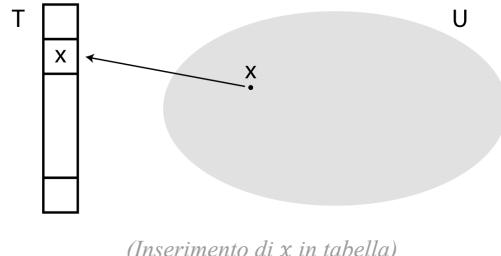
5. $h(k) = ((k \cdot \text{mod.} m) + 125) \cdot \text{mod.} m$.

Il $\text{mod.} m$ mi assicura che non si possa mai uscire dalla tabella. Tuttavia lo shift di 125 posizioni è inutile e quindi questa funzione è tanto buona quanto la più semplice $k \cdot \text{mod.} m$.

Hash con Open Addressing

Tutti gli elementi sono memorizzati nella tavola hash stessa, senza l'uso di strutture dati ausiliarie; ovvero ogni cella della tavola contiene un elemento dell'insieme dinamico o la costante *NIL*, che rappresenta che la cella non è mai stata usata. Ciò significa che dato x , si calcola la funzione di hash e tale valore viene memorizzato direttamente nella rispettiva cella di T .

Se tale posizione è già occupata però si verifica un problema (non potendo usare le liste concatenate). Per risolvere questo problema la soluzione è quella di aggiungere alla funzione di hash una valore numerico, chiamato **sequenza di scansione**, che rappresenta il numero di **tentativo** di inserimento della chiave in tabella.



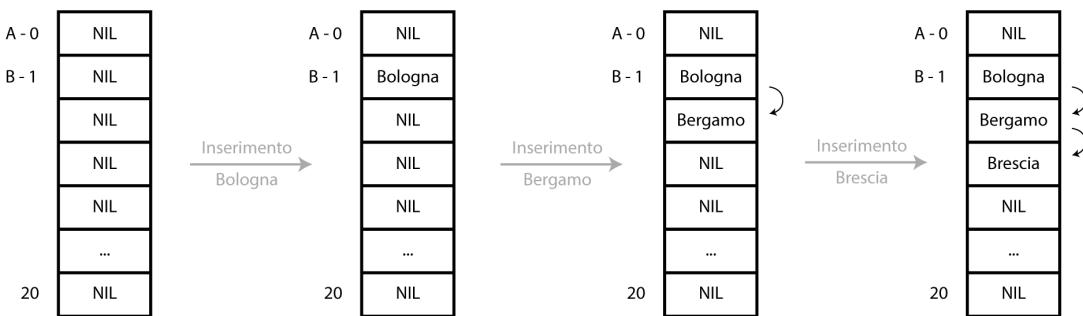
Funzione di hash con sequenza di scansione

È una funzione $h: U \times \{0, 1, \dots, m - 1\} \rightarrow \{0, 1, \dots, m - 1\}$ ricordando che m è la cardinalità della tabella. Una sequenza di scansione (a.e. $\{h(k, 0), h(k, 1), \dots, h(k, m - 1)\}$) non deve **mai contenere ripetizioni** e deve essere una **permutazione degli indici** $0, 1, \dots, m - 1$.

Funzioni di hash con sequenza di scansione standard

1. Scansione lineare: $h(k, i) = (h'(k) + i) \cdot \text{mod. } m$

Quando si incontra una collisione si usa l'**indice successivo** a quello che collide, fino a che non si trova una **casella libera**. Questo tipo di scansione causa *primary clustering*, ossia man mano che la tabella si riempie, si formano blocchi contigui di celle occupate. Non soddisfa l'ipotesi di hashing uniforme. Esempio:



2. Scansione lineare con passo: $h(k, i) = (h'(k) + p \cdot i) \cdot \text{mod. } m$

Quando si incontra una collisione si usa l'**indice dopo p indici** rispetto a quello che collide, sino a che non si trova una **casella libera**. Non soddisfa l'ipotesi di hashing uniforme.

3. Scansione quadratica: $h(k, i) = (h'(k) + c_1 \cdot i + c_2 \cdot i^2). \text{mod. } m$

Quando si incontra una collisione si usa l'**indice che collide elevato al quadrato** con normalizzazione rispetto alla grandezza della tabella dell'indice ottenuto, sino a che non si trovi una **casella libera**. Questo tipo di scansione non causa *primary clustering* ma causa *secondary clustering* (se $h(k_1, 0) = h(k_2, 0)$ allora k_1 e k_2 hanno la stessa sequenza di scansione). Non soddisfa l'ipotesi di hashing uniforme. Nota: c_1 e c_2 devono essere scelte in combinazione con m in modo da assicurare che ogni sequenza di scansione generi m posizioni distinte (per evitare che alcune celle non vengano mai scelte).

Inserimento, cancellazione e ricerca

Quando si vuole cancellare un elemento, lo si rimpiazza con una costante *DEL*. Così, quando si cerca un elemento bisogna **ispezionare sistematicamente** le celle finché non viene trovato l'elemento desiderato o finché non ci si accorge che l'elemento non si trova nella tavola (raggiungendo un *NIL* o avendo tentato senza successo la ricerca mediante tutte le funzioni di hash con sequenza di scansione $h(k, 0), h(k, 1), \dots, h(k, m - 1)$).

Inserimento $O(n) \rightarrow \Theta(n)$ nel caso peggiore	Cancellazione $O(n) \rightarrow \Theta(n)$ nel caso peggiore
<pre>hashInsertOpen(T, x, h) m = T.len() i = 0 if (i < m) j = h(x.key, i) while (T[j]≠nil && T[j]≠del && i<m) i = i + 1 j = h(x.key, i) if (i == m) { return "non c'è posto!" } else { T[j] = x return j } }</pre>	<pre>hashDeleteOpen(T, x, h) j = hashSearchOpen(T, x, h) if (j ≠ -1) T[j] = del</pre>
Ricerca $O(n) \rightarrow \Theta(n)$ nel caso peggiore	
<pre>hashSearchOpen(T, x, h) m = T.len() i = 0 if (i < m) j = h(x.key, i) while (T[j]≠nil && T[j]≠x && i<m) i = i + 1 j = h(x.key, i) if (i == m T[j] == nil) return "x non è in T" else return j }</pre>	

Complessità ed Ipotesi di hashing uniforme

L'ipotesi di hashing uniforme dichiara che tutte le permutazioni di $\{0, 1, \dots, m - 1\}$ ($n!$ permutazioni diverse) siano **equiprobabili** ($1/n!$). Nota, la scansione lineare non soddisfa l'ipotesi.

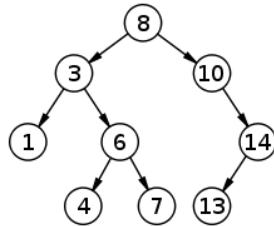
Teo. in una tabella di hash gestita con open addressing sotto ipotesi di hashing uniforme, il costo di inserimento, ricerca e cancellazione nel caso medio è $O(1/(1 - \alpha)) = \Theta(1)$ con $\alpha = n/m < 1$.

BST: Binary Search Tree

Un Albero di ricerca binaria (BST) è un albero binario radicato, i cui nodi memorizzano una chiave e ciascuno di essi ha due **sottoalberi distinti**. L'albero dovrebbe soddisfare la proprietà BST:

$$\forall x \in T \wedge \forall y \in T_{x.left} \Rightarrow y.key < x.key \text{ e } \forall x \in T \wedge \forall y \in T_{x.right} \Rightarrow y.key > x.key$$

Tale proprietà afferma che la **chiave di ogni nodo** deve essere **maggiori** di tutte le chiavi memorizzate nel **sottoalbero di sinistra** e **non maggiore** di tutte le chiavi nel **sottoalbero di destra**.



Procedure di visita

Percorrere tutto l'albero e analizzare il contenuto di ogni nodo 1 volta. Sono procedure **non in place**.

<code>preOrder(x) // O(h) → O(n)</code>	<code>inOrder(x) // O(h) → O(n)</code>	<code>postOrder(x) // O(h) → O(n)</code>
<pre> if (x ≠ nil) { print(x.key) preOrder(x.left()) preOrder(x.right()) </pre>	<pre> if (x ≠ nil) { inOrder(x.left()) print(x.key) inOrder(x.right()) </pre>	<pre> if (x ≠ nil) { postOrder(x.left()) postOrder(x.right()) print(x.key) </pre>

Visita pre-order 8-3-1-6-4-7-10-14-13

Si esamina il **contenuto** del nodo. Si procede analogamente sul **figlio sinistro** finché non si trova un nodo *nil*. Si procede analogamente sul **figlio destro** finché non si trova un nodo *nil*.

Visita in-order 1-3-4-6-7-8-10-13-14

Si procede sul **figlio sinistro** finché non si trova un nodo *nil*. Si esamina il **contenuto** del nodo. Si procede sul **figlio destro** finché non si trova un nodo *nil*. Nota: T è un *BST* \Leftrightarrow $inOrder(T.root)$ è un vettore ordinato.

Visita post-order 1-4-7-6-3-13-14-10-8

Si procede sul **figlio sinistro** finché non si trova un nodo *nil*. Si procede sul **figlio destro** finché non si trova un nodo *nil*. Si esamina il **contenuto** del nodo.

Complessità delle operazioni di visita

Poiché tutti i nodi vengono **visitati una sola volta**, la complessità algoritmica è certamente $\Omega(n)$. Inoltre è anche sicuramente $O(n)$ dato come si può dimostrare risolvendo $T(n) = T(m) + T(n - m - 1) + \Theta(1)$. Di conseguenza la complessità delle operazioni di visita è $\Theta(n)$.

Esercizi sulle visite

1. Dato il risultato di una visita (pre-order) è possibile ricostruire l'albero T ?

No, non è possibile perché non si conosce la posizione dei *nil*.

2. Dato il risultato di una pre-order e il risultato di una post-order, è possibile ricostruire l'albero T ?

No non è possibile. Un caso che lo dimostra è quello di una visita pre-order 1-2 e una visita post-order 2-1. Entrambe possono dare vita a due alberi differenti.

3. Dato il risultato di una visita pre-order e il risultato di una visita in-order, è possibile ricostruire l'albero T ?

Sì è possibile. Infatti, grazie alla pre-order, possiamo conoscere la radice dell'albero. Grazie alla in-order, possiamo sapere invece quali elementi staranno nel sottoalbero destro e nel sottoalbero sinistro.

Operazioni di ricerca

Ricerca del massimo

La chiave **massima** di un BST si trova, partendo dalla radice, in una **chiave che non ha figlio destro**.

```
BSTMax(x) // O(h) → O(n)
  if (x.right = nil)
    return x
  else
    return BSTMax(x.right)
```

```
BSTMax(x) // O(h) → O(n), in place
  x = T.root
  if (x = nil)
    return x
  while (x.right ≠ nil)
    x = x.right
  return x
```

Ricerca di una chiave k

Per cercare una chiave k in un BST si scandisce l'albero T partendo dalla radice $T.root$ e arrivato ad x si valuta

- se $x.key = k \vee x.key = nil$: ritorna x (che può essere un valore oppure *nil*, nel caso che $x \notin T$).
- se $x.key > k$: chiamata ricorsiva a **sinistra**.
- se $x.key < k$: chiamata ricorsiva a **destra**.

```
BSTSearch(x, k) // O(h) → O(n)
  if (x = nil || x.key = k)
    return x
  else
    if (x.key > k)
      return BSTSearch(x.left, k)
    else
      return BSTSearch(x.right, k)
```

Ricerca del successore di una chiave k

Dato un BST T e un nodo $x \in T$ si vuole trovare y che contiene la più piccola chiave più grande della di $x.key$. Ad esempio, nell'albero sopra, il successore del nodo 10 è 13 e il suo predecessore è 8.

- se x ha figlio destro: cerca il **valore minimo** nell'albero **radicato nel figlio destro** di x .
- se x ha figlio destro *nil*: **risale** finché non si trova un nodo al quale si è arrivati fa un figlio sinistro.

```
BSTSuccessor(x) // O(h) → O(n), in place
if (x.right ≠ nil)
    return BSTMin(x.right)
else
    y = x.parent
    while (y ≠ nil && x = y.right)
        x = y
        y = y.parent
    return y
```

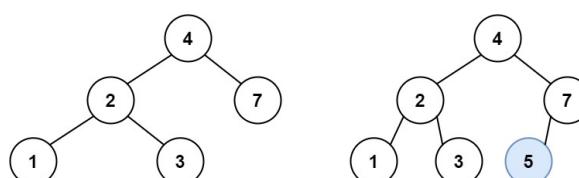
Operazione di inserimento

L'inserimento di un elemento in un albero binario di ricerca deve essere fatto in maniera che l'albero risultante dopo l'inserimento rispetti sempre le proprietà dei BST. L'algoritmo è simile a quello della ricerca: Si scende nell'albero come se si cercasse la chiave $z.key$ usando due puntatori x e y (all'inizio $x = T.root$ e $y = nil$). In ogni istante x è il genitore di y . Quando x arriva a *nil*, si aggancia z ad y .

```
BSTInsert(T, z) // O(h) → O(n)
y = nil
x = T.root
while (x ≠ nil)
    y = x
    if (x.key > z.key)
        x = x.left
    else
        x = x.right
    z.parent = y
    if (t = nil)
        T.root = z
    else
        if (y.key > z.key)
            y.left = z
        else
            y.right = z
```

Esempio di inserimento della chiave 5

Per farlo, si effettuano i controlli: $5 > 4 \rightarrow$ scendo a dx $\rightarrow 5 < 7 \rightarrow$ scendo a sx \rightarrow trovo nil \rightarrow inserisco.



Operazione di cancellazione

Per mantenere le proprietà dei BST anche dopo la cancellazione di un elemento z da un albero binario di ricerca T , bisogna distinguere 3 casi differenti:

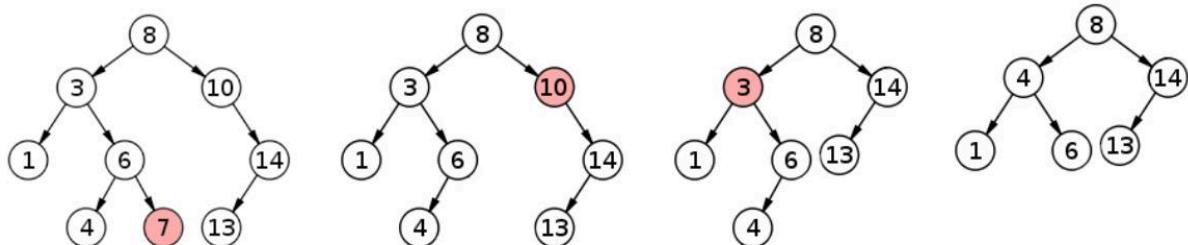
- Se z è una **foglia**: elimina z .
- Se z ha **un figlio**: elimina z ,
porta su l'albero radicato nell'unico figlio di z .
- Se z ha **due figli**: cerca il successore di z ,
scambia z con il suo successore,
elimina z .

Per farlo si opera nel seguente modo

1. Si individua il nodo x da eliminare (nel caso 1-2 $\Rightarrow x = z$, nel caso 3 $\Rightarrow x = BSTSuccessor(z)$).
2. Si elimina il nodo x sostituendolo con il suo figlio v (figlio eventualmente non *nil* di x).
3. Se $x \neq z$ si copia la chiave di x in z .

```
BSTDelete(T, z) //O(h) → O(n)
if (z.left == nil || z.right == nil)
    x = z
else
    x = BSTSuccessor(z)
y = parent(x)
if (x.left != nil)
    v = x.left
else
    v = x.right
if (y != nil)
    T.root = v
else
    if (x == y.left)
        y.left = v
    else
        y.right = v
if (v != nil)
    v.parent = y
if (z != x)
    z.key = x.key
```

Esempio di cancellazione delle chiavi 7, 10, 3



Alberi rosso-neri

Definizione

I RBT nascono come **soluzione** al problema nel caso peggiore dei BST in cui avendo un BST completamente **sbilanciato** da un lato, le operazioni su di esso risultano **poco efficienti**. Un albero rosso-nero è dunque un BST che, in aggiunta ai requisiti ordinari per un [albero binario di ricerca](#), soddisfa le seguenti proprietà:

1. ogni nodo ha un **colore** che può essere rosso o nero,
2. le **foglie** sono *nil* e di colore nero,
3. ogni **nodo rosso** ha sempre un **genitore nero** (oppure entrambi i figli di ciascun nodo rosso sono neri),
4. **ogni cammino** nodo-foglia contiene lo **stesso numero** di nodi **neri**.

Grazie a questi vincoli il **cammino radice-foglia più lungo** è al massimo lungo il doppio del cammino più breve. Ne risulta dunque un albero **fortemente bilanciato**.

Altezza ed altezza nera

L'**altezza** $h(T)$ è $\Theta(\log n_i)$ dove $n_i = 2^k - 1$ è il numero di **nodi interni** di un RBT di $h(T) = k$. Essa rappresenta la lunghezza del **cammino radice-foglia più lungo** (contando anche la radice).

L'**altezza nera** $bh(T)$ è il numero di **nodi neri** dal nodo x ad una foglia (x non si conta, la foglia sì).

Proprietà sull'altezza

1. $h(T) \geq bh(T)$: l'altezza è **sempre maggiore o uguale** dell'altezza nera.
2. $h(T) \leq 2bh(T)$: il **cammino radice-foglia più lungo** è al massimo il **doppio del cammino più breve**.
3. $n_i \geq 2^{bh(T)} - 1$: il **#nodi interni** (chiave) è \geq al **#nodi di un albero binario completo di altezza** $bh(T)$.

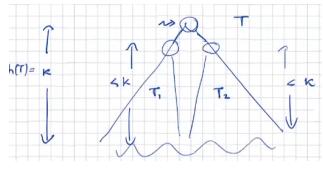
Dimostrazione proprietà (3)

Per induzione sull'altezza che $n_i \geq 2^{bh(T)} - 1$:

Base: $h(T) = 0$ (albero vuoto, composto da un'unica foglia) $\rightarrow h(T) = 0$, $bh(T) = 0$, $n_i = 0 \rightarrow 0 \geq 2^0 - 1$ ok.

Ipotesi induttiva (so che funziona): se $h(T) < k$ allora $n_i \geq 2^{bh(T)} - 1$.

Tesi (da dimostrare): se $h(T) = k$ allora $n_i \geq 2^{bh(T)} - 1$.



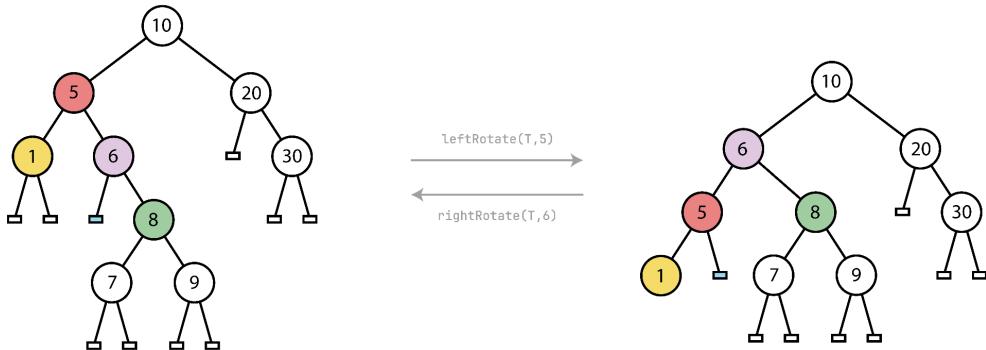
$$\begin{aligned}
 n_i(T) &= n_i(T_1) + n_i(T_2) + 1 && \text{che per ipotesi induttiva} \\
 &\geq 2^{bh(T_1)} - 1 + 2^{bh(T_2)} - 1 + 1 && \text{ed essendo le due altezze nere uguali} \\
 &\geq 2^{bh(T_1)} + 1 - 1 && \text{che dato che } bh(T) \leq bh(T_1) + 1 \\
 &\geq 2^{bh(T)} - 1.
 \end{aligned}$$

Dimostrazione che un RBT è bilanciato

Se T è un RBT, $n_i \geq 2^{bh(T)} - 1 \geq 2^{h/2} - 1$ e quindi $h \leq 2 \log(n_i + 1)$ e quindi $h = O(\log n)$ e dato per qualsiasi albero h è almeno $\log n$ (ossia $\Omega(\log n)$) allora l'altezza di un RBT è $\Theta(\log n)$ e i RBT sono bilanciati.

Operazioni di rotazione $\Theta(1)$

Esistono due tipi di rotazioni: `leftRotate(T, x)` e `rightRotate(T, x)` che funzionano come nell'esempio:

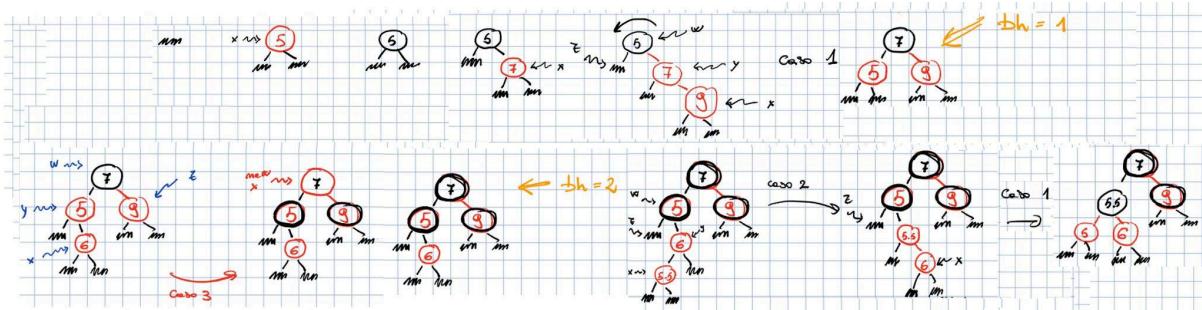


Operazioni di inserimento $O(\log n)$

1. Inserisci il nuovo nodo, sempre rosso, nella posizione nell'albero come se fosse un BST.
2. Se il nuovo nodo è **figlio di un nodo nero** la procedura **termina**.
3. Altrimenti si chiama: il nuovo nodo x , suo padre y , suo nonno w , suo zio z e si divide nei 3 casi:
 1. Se lo **zio di x è nero e opposto** ad x : rotazione **verso** lo zio con perno w e **ricolora** y e w .
 2. Se lo **zio di x è nero e stesso lato** di x : rot. **opposta** allo zio con perno y e **caso 1** con $x = y$.
 3. Se lo **zio di x è rosso**: ricoloro y , z e w e mi riconduco agli altri casi.

Se durante la procedura la **radice diventa rossa**, basata colorarla di nero e la procedura **termina**.

Esempio



Operazioni di cancellazione $O(\log n)$

La cancellazione avviene come se il RBT fosse un BST:

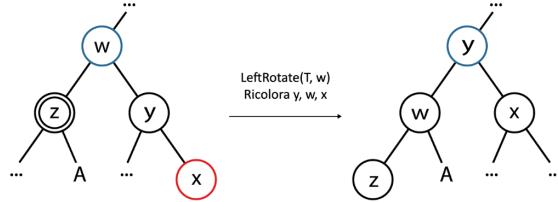
- Se z è una **foglia**: si elimina z .
- Se z ha un **figlio**: si elimina z sostituendolo con il sottoalbero radicato nel suo unico figlio.
- Se z ha **due figli**: si cerca il successore di z , si scambia z con il suo successore, si elimina z .

Se il nodo che viene eliminato **era nero**:

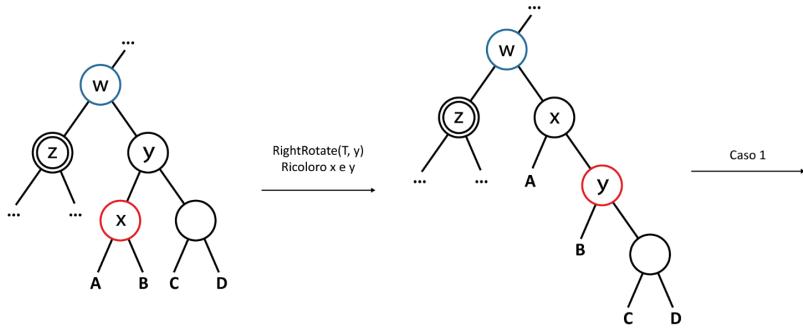
- Ricolora il nodo che sostituisce il nodo eliminato come **double black**.
- Se il double black prima era **red**, lo si **colora di nero** e la procedura **termina**,

- Altrimenti si divide nei 4 casi:

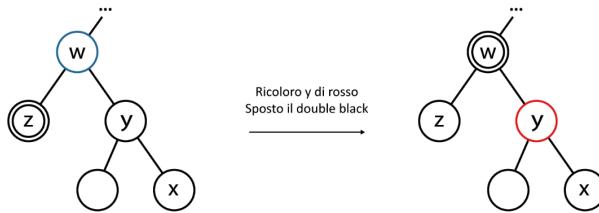
1. Db ha un **nipote rosso opposto** a lui: rotazione **verso** il db con perno w e **ricolora** y, w e x :



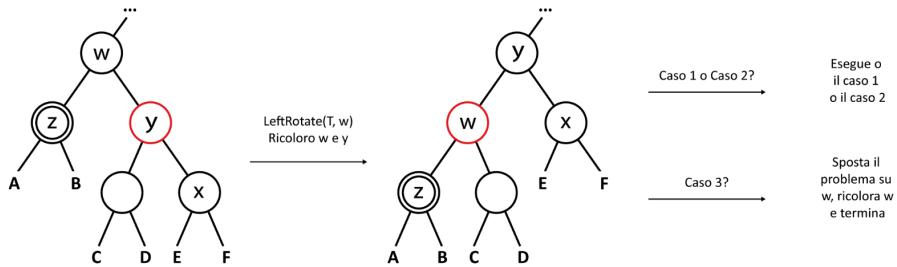
2. Db ha **nipote rosso dallo stesso lato**: rotazione **opposta** al db con perno y (fratello del db), **ricolora** y, w e x e caso 1:



3. Db ha **entrambi i nipoti neri** e il **fratello nero**: **ricoloro** y (fratello del db) di rosso, **sposto il db** al w . Se w era rosso, lo ricoloro di nero e la procedura termina, altrimenti ci si riconduce ad un altro caso.



4. Db ha **entrambi i nipoti neri** e il **fratello rosso**: rotazione **verso** il db con perno w , **ricolora** w e y . Si esegue poi la relativa (al caso in cui ci si trova) procedura.



Operazioni di ricerca $O(\log n)$

Ricerca del massimo

La chiave massima di un BST si trova, **partendo dalla radice**, in una chiave che **non ha figlio destro**.

```
RBTSearchMax(x)
if (x.right = nil)
    return x
else
    return BSTMax(x.right)
```

Ricerca di una chiave k

Per cercare una chiave k in un RBT si scandisce l'albero T partendo dalla radice x :

- se $x.key = k$ o $x.key = nil$ ritorna x .
- se $x.key > k$ chiamata ricorsiva a **sinistra**.
- se $x.key < k$ chiamata ricorsiva a **destra**.

```
RBTSearchKey(x, k)
if (x = nil || x.key = k)
    return x
else
    if (x.key > k)
        return BSTSearch(x.left, k)
    else
        return BSTSearch(x.right, k)
```

Ricerca del successore/predecessore di una chiave k

Dato un nodo $x \in T$ si vuole trovare y che contiene il **successore** di $x.key$:

- se x ha **figlio destro** si cerca il valore minimo nell'albero radicato nel figlio destro di x .
- se x ha **figlio destro nil** risale finché non si trova un nodo al quale si è arrivati da un figlio sx.

```
RBTSuccessor(x)
if (x.right ≠ nil)
    return BSTMin(x.right)
else
    y = x.parent
    while (y ≠ nil && x = y.right) {
```

```

x = y
y = y.parent
return y

```

Operazioni di fusione di due RBT $O(\log n)$

Dati T_1 e T_2 RBT ed una chiave k tale che il suo valore è compreso tra i valori contenuti in T_1 e T_2 , si vuole ottenere il RBT T dato dall'unione di k , T_1 e T_2 . Se $bh(T_1) = bh(T_2)$ si usa k come radice di T , che avrà come figlio sinistro T_1 e come figlio destro T_2 . Se $bh(T_1) > bh(T_2)$ si scorre i figli destri di T_1 fino a quando si trova un nodo nero x tale che $bh(x) = bh(T_2)$. A questo punto basta agganciare questi due nodi tramite il nodo di chiave k . Se il genitore di x è nero, si pone k rosso e la procedura termina, altrimenti si pone k rosso e si rende k figlio dell'attuale padre di x . Se il genitore di k è rosso, si richiama $RBTFixUp$ vista nelle [operazioni di inserimento](#). Il caso $bh(T_1) < bh(T_2)$ è duale.

Min heap vs RBT

	Min heap	RBT
Inserimenti	$O(\log n)$	$O(\log n)$
Cancellazione	$O(\log n)$	$O(\log n)$
Ricerca e cancellazione minimo	$O(\log n)$	$O(\log n)$

Le min heap richiedono di essere implementate con un vettore sovradimensionato mentre per i RBT **non serve** nessuna **stima sulla dimensione** a priori.

B-Tree

Un B-tree è una struttura dati che permette la **rapida localizzazione** dei file, riducendo il numero di volte che un utente necessita per accedere alla memoria (secondaria) in cui il dato è salvato. Essi **derivano dai BST**, in quanto ogni chiave appartenente al sottoalbero **sinistro** di un nodo è di **valore inferiore** rispetto a ogni chiave appartenente ai sottoalberi alla sua destra.

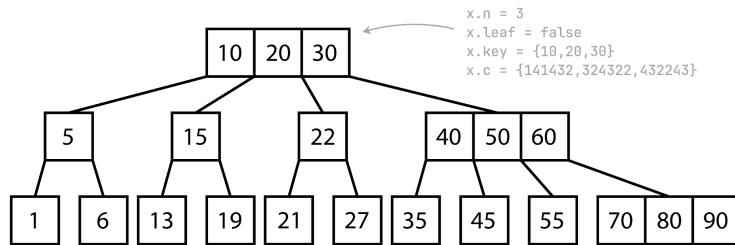
Inoltre, la loro struttura ne **garantisce il bilanciamento**: per ogni nodo, le altezze dei sottoalberi destro e sinistro differiscono al massimo di una unità.

Definizione dei BTrees

Ogni nodo x del B Tree contiene i seguenti campi:

1. $x.n \rightarrow$ intero che indica **quante** chiavi sono memorizzate nel nodo x ,
2. $x.leaf \rightarrow$ booleano che ha valore **vero** se x è una **foglia**,
3. $x.key \rightarrow$ vettore contenente i $2t - 1$ le **chiavi** (numeri interi) del nodo x ,
4. $x.c \rightarrow$ vettore dei **puntatori ai figli** contenente $2t$ valori.

Ad esempio di BTree di **grado** $t = 2$ (ogni nodo ha **almeno** $t - 1 = 1$ e al **massimo** $2t - 1 = 3$ chiavi):



Proprietà dei BTrees

Su ogni nodo valgono le seguenti **proprietà**:

1. Ogni nodo contiene **almeno** $t - 1$ (la **radice** ne contiene almeno 1) e al **massimo** $2t - 1$ chiavi,
2. Le chiavi in un nodo (vettore) sono in **ordine crescente** e non si ammettono ripetizioni,
3. I **figli** si trovano nelle prime $x.n + 1$ posizioni del vettore c ,
4. Tutte le **foglie** sono allo **stesso livello**.

Altezza e altezza massima

L'altezza aumenta solo quando viene fatto lo **split della radice**. Un BTree T di grado t con n chiavi ha **altezza**:

$$h \leq \log_t \frac{n+1}{2} \text{ (ossia } h \in O(\log_t \frac{n+1}{2})).$$

Dim: Per creare il BTree di **altezza massima**, ogni nodo deve contenere il **numero minimo** possibile di chiavi. Quindi la radice conterrà solo una chiave. Tale nodo necessita di almeno due figli che contengono $t - 1$ chiavi. A loro volta, questi due nodi, avranno al secondo livello t figli ciascuno. Al terzo livello ci sono quindi $2t$ nodi, ognuno dei quali contiene $t - 1$ chiavi. Al quarto livello, ci sono $2t^2$ nodi, ognuno dei quali ha $t - 1$ chiavi e così via. Si ottiene quindi:

$$\sum_{i=0}^{h-1} 2t^i (t - 1) + 1 = n \rightarrow 2(t - 1) \sum_{i=0}^{h-1} t^i + 1 = n \rightarrow 2(t^h - 1) + 1 = n \rightarrow h = \log_t \frac{n+1}{2}.$$

Operazioni sui Btree

Dato che la struttura dati risiede in memoria secondaria, per poter lavorare con i nodi bisogna **spostarli in memoria principale**. Una volta fatto, tali nodi devono essere **ricaricati** in memoria secondaria. Per questa ragione ci sono **due tipi di operazioni** (bisogna sempre tenere **separati** i due costi, anche quando sono uguali):

- **R/W** dalla memoria secondaria → costo $O(h) = (\log_t n)$.
- **CPU** (avvengono in memoria principale) → costo $O(t h) = (t \log_t n)$.

Obiettivo delle procedure è quello di cercare sempre di **limitare** il numero di operazioni R/W.

Operazioni di ricerca

Dato un BTree T di grado t e dato k , si vuole cercare k in T : partendo dalla radice:

1. Scandisce il vettore (nodo). Se trova:

- la chiave k : la procedura **termina** ritornando il nodo e la posizione dov'è contenuto k .
- l maggiore k : **scende nel figlio sinistro** di l e **ripete** il ragionamento dal punto 1.
- un valore *nil*: la procedura **termina** senza successo.

```
BTreeSearch(T, t, x, k) // x é in ram
  i = 1
  while (i <= x.n && x.key[i] < k)
    i++
  if (i <= x.n && x.key[i] == k)
    return (x, i)
  else
    if (x.leaf)
      return NIL
    else
      y = DiskRead(x.c[i])
      return BTreeSearch(T, t, y, k)
```

Costi per la ricerca

Il **numero di operazioni** di lettura/scrittura da disco dipende solo dall'**altezza** dell'albero. La ricerca della chiave minima e la ricerca della chiave massima hanno costi diversi in termini di operazioni di CPU e letture/scritture.

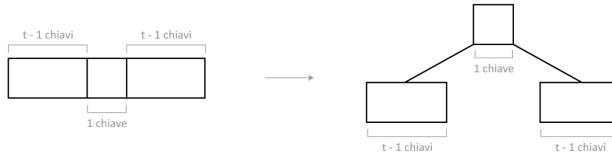
Nel **caso migliore** il costo è $O(1)$ per le operazioni di CPU e $O(1)$ per le operazioni di R/W.

Creazione albero vuoto

```
BTreeCreate(T, t)
  x = AllocateNewNode(t)
  x.leaf = true
  x.n = 0
  T.root = DiskWrite(x)
  return T
```

Operazioni di inserimento

Per inserire k in T BTTree di grado t , k viene inserito in una foglia che **esiste già** ma se tale foglia è **piena** (e quindi $x.n = 2t - 1$) allora si applica un'operazione di **splitting** che dato un nodo ne **genera due nuovi**:



Questa operazione viene fatta **ogni volta** che si incontra un nodo pieno **durante la discesa** (preliminare all'inserimento) dalla radice verso le foglie: Partendo dalla radice, si controlla se tale nodo è **pieno**, se lo è si **richiama la procedura di split**, se non lo è si **scende** al livello successivo e ci si **ripone la domanda**. Supponendo x ed y in ram, con y i-esimo figlio (pieno) e x padre di y (non pieno), si ha la procedura:

```
BTreeSplit(x, y, i, t)           // costi R/W: O(t), costi CPU: O(1)
z = AllocateNewNode(t)
z.leaf = y.leaf
z.n = t-1
for (j=1 to t-1)
    z.key[j] = y.key[t+j]
if (!y.leaf)
    for (j=1 to t)
        z.c[j] = y.c[t+j]
y.n = t-1
for (j=x.n down to i)
    c.key[j+1] = x.key[j]
x.key[i] = y.key[t]
for (j=x.n+1 down to i+1)
    x.c[j+1] = x.c[j]
x.c[i+1] = z
x.n = x.n + 1
DiskWrite(x, y, z)
```

Procedura di inserimento

Se la radice non è piena, richiama **BTreeInsertNotFull** sul nodo $T.root$. Altrimenti:

- **Crea** un nuovo nodo s con **0 chiavi e non foglia** e lo **genitore** della vecchia radice.
- **Richiama** **BTreeInsertNotFull** sul nodo s che sicuramente non è pieno.

```
BTreeInsert(T, t, k)           // costi R/W: O(logtn), costi CPU: O(t logtn), k non è in T
r = T.root
if (r.n = 2t-1) {             // se la radice è piena...
    s = AllocateNewNode(t)   // dato che la radice non ha un genitore dove inserire
    s.n = 0                  // crea un nuovo nodo s (radice) con 0 chiavi...
    s.leaf = false            // ...non foglia...
    s.c[1] = r               // ...con un unico figlio (la vecchia radice)...
    T.root = s               // ...e lo imposta come nuova radice di T
    BTreeSplit(s, r, 1, t)    // porta r (la vecchia radice) a dx di s (nuova radice)
    BTreeInsertNotFull(s, k, t) // sono certo che la radice non è piena → parto da s
else
    BTreeInsertNotFull(r, k, t) // sono certo che la radice non è piena → parto da r
```

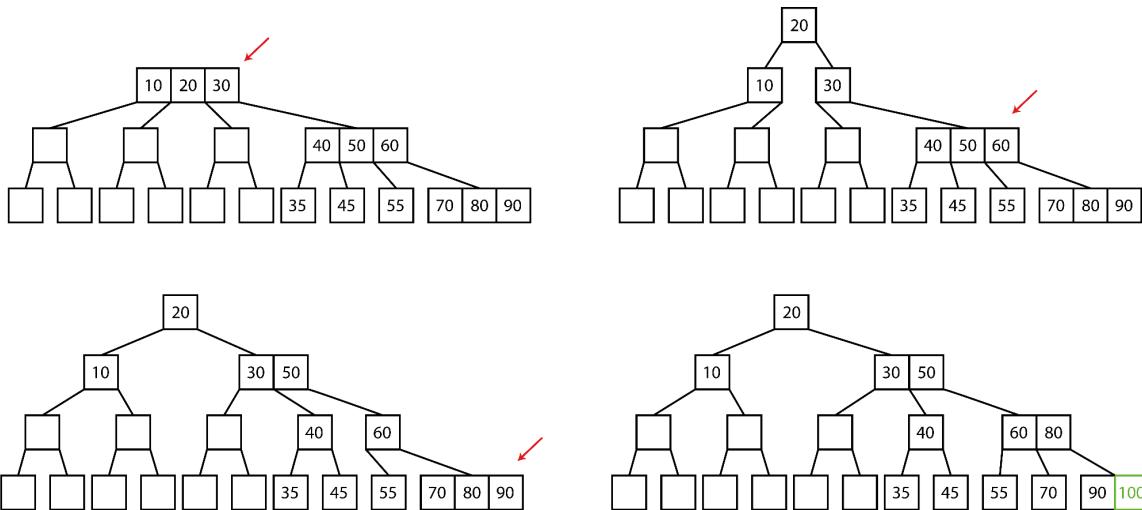
```

BTreeInsertNotFull(x, k, t) // PRE: sono certo che x non sia pieno e x in ram
    if (x.leaf) // x é una foglia: sposta verso dx le chiavi >k e inserisci k
        i = x.n
        while (i ≥ 1 && x.key[i]>k)
            x.key[i+1] = x.key[i]
            i = i - 1
        x.key[i+1] = k
        x.n = x.n + 1
        DiskWrite(x)
    else // x non é una foglia
        i = x.n
        while (i ≥ 0 && x.key[i] > k)
            i = i - 1
        y = DiskRead(x.c[i+1])
        if (y.n < 2t-1) // se y non é pieno:
            BTreeInsertNotFull(y, k, t) // richiamo la procedura su y
        else // se y è pieno:
            BTreeSplit(x, y, i+1, t) // splitta e determina da che parte scendere:
            if (k < x.key[i+1])
                y = DiskRead(x.c[i+1])
                BTreeInsertNotFull(y, k, t)
            else
                z = DiskRead(x.c[i+2])
                BTreeInsertNotFull(z, k, t)

```

Esempio di inserimento

Inserire la chiave 100 nel seguente Btree:



Costo per l'inserimento

L'inserimento di k in T ha un costo asintotico in termini di operazioni di CPU (ma non in termini di operazioni R/W) che può variare in base a k .

Operazioni di cancellazione

Se x si trova in una foglia...

1. con almeno t chiavi:
 - a. elimina x .
2. con esattamente $t-1$ chiavi e ha un fratello con almeno t chiavi:
 - a. porta su il valore più vino ad x in tale fratello,
 - b. abbassa un valore dal padre,
 - c. elimina x .

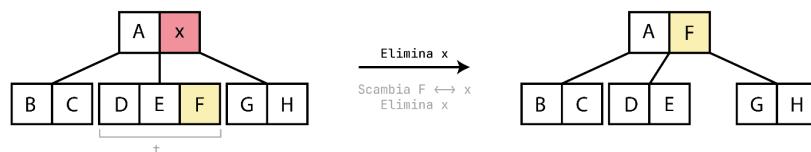


3. con esattamente $t-1$ chiavi e non ha fratelli con almeno t chiavi:
 - a. fonde x insieme a un dei suoi fratelli usando una chiave del padre come nesso,
 - b. elimina x .



Se x si trova in un nodo interno...

1. dove il figlio sinistro/destro di x ha almeno t chiavi:
 - a. scambia l'elemento maggiore/minore del figlio sinistro/destro di x al posto di x ,
 - b. elimina x .



2. dove nessun figlio di x ha almeno t chiavi:
 - a. elimina x ,
 - b. usa il minore (o il maggiore) tra le chiavi dei figli di x per chiudere il buco,
 - c. fa il merge tra i 2 figli portando giù quello che aveva sostituito x .



Il costo di queste operazioni è $O(t \log_t n)$ per operazioni di CPU e $O(\log_t n)$ per operazioni di R/W.

Gestione di insiemi disgiunti

Dato un insieme S di insiemi S_i tali che $S_i \cap S_j = \text{null}$ se $i \neq j$ e $S_i \neq \text{null}$ sono definite le operazioni **MUF**:

- **make(x)** → costruisce $\{x\}$
- **find(x)** → restituisce S_i tale che $x \in S_i$
- **union(x,y)** → dati $x \in S_i$ e $y \in S_j$ con $i \neq j$ ritorna $S_i \cup S_j$

Vengono implementati con **liste concatenate** (weighted union) o **alberi** (union by rank e path compression).

1. MUF con liste concatenate

Ognuno degli insiemi che si vuole gestire è memorizzato in una lista del tipo $S_i \rightarrow x_{i1} \rightarrow x_{i2} \rightarrow \dots \rightarrow x_{in} \rightarrow \text{null}$. Dove ogni insieme S_i è composto da degli oggetti x_{ij} che contengono campi:

- **x.key** → valore numerico.
- **x.next** → puntatore all'elemento successivo.
- **x.rap** → puntatore al primo elemento della lista concatenata .

Inoltre, solo il primo elemento della lista concatenata ha un campo aggiuntivo:

- **x.last** → puntatore all'ultimo elemento della lista concatenata.

% x è già del tipo corretto % x.key è già inizializzato	% Per unire si modifica y.rap di ogni % elemento di Sy impostandolo uguale a x.rap
make(x) <pre>x.rap = x x.next = nil x.last = x return x</pre> find(x) <pre>return x.rap</pre>	union(x, y) <pre>z = find(x) w = find(y) if (z ≠ w) link(z,w)</pre> link(z,w) % $z \neq w$ rap. <pre>z.last.next = w z.last = w.last while(w ≠ nil) w.rap = z w = w.next</pre>

- **make(x)**: $\Theta(1)$ e se ne fanno $n \rightarrow \Theta(n)$.
- **find(x)**: $\Theta(1)$ e se ne fanno al massimo $m \rightarrow \Theta(1) \cdot O(m) = O(m)$.
- **union(x,y)**: $\Theta(|S_y|)$ dove $|S_y|$ vale al massimo $n-1$, quindi $\Theta(|S_y|) = \Theta(n)$ di cui se ne fanno $n \rightarrow O(n^2)$.

E il costo complessivo per le operazioni MUF è $O(m+n^2)$

2. MUF con alberi

Ogni nodo contiene una chiave e un puntatore al genitore. Un nodo orfano punta a se stesso

Union by rank

Per ottimizzare il costo si può agganciare l'albero più basso a quello più alto (così l'altezza cresce solo quando si uniscono due alberi con altezza uguale). In ogni nodo si aggiunge un campo $x.rank$ che indica il rango (che nel caso non si usi la find con path compression è l'altezza dell'albero) dell'albero radicato in x .

```
make(x)
  x.parent = x
  x.rank = 0
  return x
```

```
unionByRank(x, y)
  z = find(x)
  w = find(y)
  if (z ≠ w)
    if (z.rank > w.rank)
      link(z, w)
    else
      link(w, z)
      if (w.rank = z.rank)
        w.rank = w.rank + 1
```

- **make**(x): $\Theta(1)$ e se ne fanno n
- **find**(x): $O(\log n)$ e se ne fanno al massimo m
- **union**(x,y): $O(\log n)$

E il costo complessivo per le operazioni MUF è $O(n) + O(m \log n)$.

Path compression

Quando si esegue una find tutti i nodi sul cammino x -radice vengono agganciati alla radice.

```
findPathCompression(x)
  if (x ≠ x.parent)
    x.parent = findPathCompression(x.parent)
  else return x
```

Grafi non pesati

Un **grafo** (orientato o non orientato) $G = (V, E)$ è composto da V insieme dei **nodi** ($V = \{1, \dots, n\}$) ed E insieme degli **archi**. Siano $u, v \in V$:

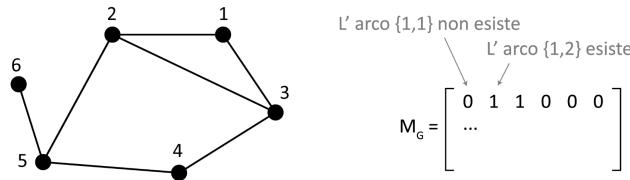
- nel caso di grafi **non orientati** gli archi vengono rappresentati con $\{u, v\}$. Tali archi non tengono conto dell'ordine ossia vale $\{u, v\} = \{v, u\}$.
- nel caso di grafi **orientati** gli archi vengono rappresentati con (u, v) e vale $(u, v) \neq (v, u)$. Pertanto tali archi hanno anche un verso.

Nota: un albero (senza numero massimo e senza un ordinamento per i figli) è un **grafo non orientato, aciclico** (non contiene cicli) e **connesso** ($\forall u, v \exists u - v$).

Memorizzazione di un grafo

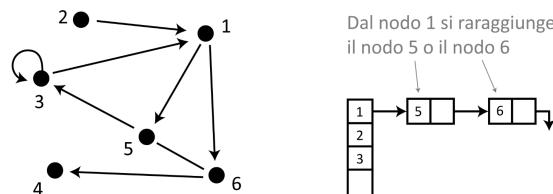
Un grafo può essere memorizzato in memoria in due modi, ovvero utilizzando:

1. una **matrice di adiacenza** M_G di dimensione $|V| \times |V|$ in cui $M_G[i, j] = 0$ se $\{u, v\} \notin E$ e $M_G[i, j] = 1$ se $\{u, v\} \in E$. Questa rappresentazione utilizza spazio $\Theta(|V|^2)$.



Osservazione: Se G è non orientato M_G è **simmetrica** con diagonale composta da soli zeri (dato che non ci sono archi che connettono nodi $\{u, u\}$, ossia lo stesso nodo). La matrice, nel caso di grafo orientato, non è necessariamente simmetrica. Di conseguenza una generica matrice booleana quadrata rappresenta un grafo orientato.

2. una **lista di adiacenza** L_G di dimensione $|V| = n$ in cui $L_G[i]$ lista di nodi raggiungibili con un arco dal nodo i . Tale rappresentazione utilizza spazio $\Theta(|V| + |E|)$ dove $0 \leq |E| \leq |V|^2$.



Definizione cammino e distanza

Un **cammino** da s a v è $(s, u)(u_1, u_2) \dots (u_{n-1}, v)$ di archi di G tali che $(s, u) \in E$, $(u_1, u_2) \in E$ e $(u_{n-1}, v) \in E$. La **lunghezza di un cammino** è il numero di archi nel cammino (nota: una sequenza vuota è un cammino di lunghezza 0 da s ad s). La **distanza** δ è una funzione a due parametri $\delta(s, v)$ che vale ∞ se non

esiste in G un cammino da s a v e vale l se esiste in G un cammino da s a v e non esiste un cammino di lunghezza $l - 1$.

Visite BFS - Ricerca in ampiezza

È un algoritmo di ricerca che partendo da un nodo sorgente permette di cercare il cammino fino ad un altro nodo scelto e connesso al nodo sorgente. Dato in input un grafo G ed un nodo s , $BFS(G, s)$ modifica tre vettori:

- $\text{color}[i]$ → contiene il **colore** del nodo i , che può essere: B , G , N .
- $d[i]$ → al termine contiene la **distanza** di s da i .
- $\pi[i]$ → al termine contiene il **nodo precedente** ad i , nel cammino minimo:

Grazie a π si può ricostruire a ritroso il **cammino minimo** tra due nodi. A.e. se si volesse sapere qual è il cammino minimo per giungere al nodo i da s si guarda ciò che è contenuto in $\pi[i]$, trovando il nodo j . Successivamente se $j \neq s$ si controlla $\pi[j]$ e così via fino a giungere a s . L'idea che sta dietro è la seguente:

1. **Inizializza** i tre vettori ed una lista vuota.
2. **Colora** s di grigio, imposta la distanza a 0 da se stesso e lo inserisce in coda.
3. La coda **presenta elementi**?

Sì:

- **estrae e salva** in una variabile u l'elemento in testa alla coda,
- il **colore** del primo, secondo, terzo ... nodo nella **lista di adiacenza** di u è **binaco**?"

Si: ($\text{adj}(u)$ ha ancora elementi non esplorati)

- lo **colora** di grigio
- **aumenta** la sua **distanza** da s
- **salva** in più l'**origine** da cui si è arrivati
- lo **inserisce in coda**

No: ($\text{adj}(u)$ è stata esplorata tutta)

- **colora** u di nero
- **rimuove** u dalla **coda**
- **torna** al punto 3.

No:

- BFS **termina**

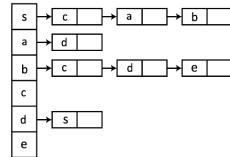
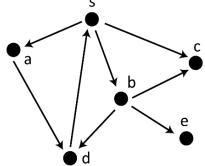
```

BFS(G, s)           // Θ(|V|^2) con matrici di Adj e O(|E|) con liste di Adj
    for (each v in V) // eseguito Θ(|V|) volte → inizializza i tre vettori
        color[v] = BIANCO // All'inizio i nodi sono tutti bianchi
        d[v] = +inf      // All'inizio i nodi sono tutti a distanza + infinito
        pi[v] = NIL       // All'inizio non si sa nulla dei predecessori
    Q = θ                // θ(1) → Q lista inizialmente vuota
    color[s] = GRIGIO   // θ(1) → s è il primo nodo ad essere visitato e diventa grigio
    d[s] = 0              // θ(1) → chiaramente s si trova a distanza zero da se stesso
    enqueue(Q, s)        // θ(1) → Inserisce s in testa alla lista Q
    while (Q ≠ θ)         // eseguito Θ(|V|) volte, il for interno Θ(|V|) → Θ(|V|^2)
        u = head(Q)       // Esamina il nodo in testa alla coda...
        for (each v in Adj(u)) { // ...scandendone la lista di Adj (eseguito Θ(|V|) volte)
            if(color[v] = BIANCO) { // se il nodo v trovato in lista di adj è bianco
                color[v] = GRIGIO // colora v di grigio perché viene visitato
                d[v] = d[u]+1     // v si trova a distanza nodo_precedente+1 da s
                pi[v] = u          // il nodo v è stato scoperto dal nodo u
                enqueue (Q, v)     // inserisce v in coda
            }
        }
        color[u] = NERO      // è stata esplorata tutta la lista di adj
        dequeue(Q)           // ...e lo tolgo dalla coda
    
```

Complessità

Il costo della visita BFS con **matrici di adiacenza** è $O(|V|) \cdot \Theta(|V|) = \Theta(|V|^2)$ perché il ciclo while viene eseguito $O(|V|)$ volte e dentro a tale ciclo c'è un ulteriore ciclo for che costa $\Theta(|V|)$. Il costo della visita BFS con **liste di adiacenza** è $O(|V| + |E|)$ dato che il ciclo while viene eseguito sempre $O(|V|)$ volte ma il ciclo for viene eseguito $\Theta(|L_G[u]|)$ volte e il suo costo è $\sum_{u \in V} \Theta(|L_G[u]|) = O(|E|)$.

Esempio



Ad esempio, nel grafo qui sopra, dopo aver scoperto s (che è l'unico a distanza 0 da se stesso) e colorato di grigio, si vorrà scoprire uno dei nodi a distanza 1 dal nodo s come ad esempio il nodo c , che una volta scoperto diventerà anch'esso grigio. Una volta scoperto c , non è possibile passare ad un nodo che si trova a distanza 2 da s , a meno che i nodi a distanza 1 da s non siano già stati visitati tutti. Di conseguenza, il terzo nodo che si scopre è il nodo a (sempre a distanza 1 da s), che verrà anch'esso colorato di grigio. A questo punto l'ultimo nodo da scoprire a distanza 1 da s è il nodo b , che viene colorato di grigio:

A questo punto, avendo finito di visitare la lista di adiacenza del nodo s , si può procedere a colorare s di nero e inserirlo nell'output della visita. La situazione ora è la seguente:

Ora si passa al nodo c e si ripete il ragionamento. Scandisco quindi la lista di adiacenza di c . Essendo tale lista vuota, si può passare alla colorazione di nero del nodo c :

A questo punto scandisco quindi la lista di adiacenza di a trovando a distanza minima solo il nodo d , che viene colorato di grigio. Anche con il nodo a la procedura è conclusa, quindi coloro di nero il nodo a :

Fatto ciò si scandisce la lista di adiacenza del nodo b , trovando c (già nero, quindi viene ignorato), d (già grigio, quindi viene ignorato anch'esso) ed il nodo e , che essendo ancora bianco viene colorato di grigio.

Avendo finito di visitare la lista di adiacenza di b , posso colorarlo di nero e in questo modo la procedura continua fino ad avere tutti i nodi neri.

Proprietà

Al termine di $BFS(G, s)$ si ha che $d[v] \geq \delta(s, v)$. questo perché $d[v]$ al termine contiene una distanza qualsiasi (anche la più lunga/corta) di s da v che deve essere maggiore o uguale a $\delta(s, v)$ che invece è la distanza minima tra i due nodi. Durante l'esecuzione di $BFS(G, s)$ se la coda $Q = [u_1 \dots u_k]$ si ha:

$$u_1 \dots u_k \text{ sono grigi} \quad \pi[u_i] = NIL \text{ se } u_i \neq s \quad d[u_1] \leq d[u_2] \leq \dots \leq d[u_k] \leq d[u_1] + 1$$

Esercizi

1. Dato $G = (V, E)$ e dati due nodi s, v di G restituire il cammino di lunghezza minima da s a v .

Per farlo, basta richiamare $BFS(G, s)$ e seguire il cammino minimo mostrato nel vettore $\pi[s]$ aggiungendo tali valori in una lista. Il costo è $\Theta(|V|^2)$ con matrici di adj, $O(|V| + |E|) + O(|V|)$ con lista di adj.

2. Dato $G = (V, E)$ orientato decidere se G è ciclico.

Per farlo, basta richiamare $BFS(G, s)$ e aggiungere una condizione che ritorna vero se si trova un arco da un nodo grigio ad uno non bianco (ignorando gli archi che mi riportano al predecessore $\pi[u]$).

3. Dato $G = (V, E)$ e dato un nodo s di G restituire il cammino di lunghezza massima da s .

Per farlo, basta richiamare $BFS(G, s)$ e scansiono d cercando il massimo in $O(|V| + |E|) + \Theta(|V|)$. Oppure, durante $BFS(G, s)$ memorizzo l'ultimo nodo che esce dalla coda in $O(|V| + |E|) \rightarrow \Omega(|V|)$.

Visite DFS - Ricerca in profondità

Nella visita BFS, in **presenza di cicli**, ad un certo punto si troverà un arco da grigio a nero e non si ha modo di sapere se tale arco chiude o meno un **ciclo** (occhio al caso di archi non orientati però). Per questo motivo si utilizza la visita DFS che è utile per stabilire se il grafo è **ciclico** o no. Anche in questo caso si utilizzano delle **strutture dati ausiliarie** per descrivere la visita:

- `color[i]` → contiene il **colore** del nodo i , che può essere: *B*, *G*, *N*.
- `π[i]` → al termine contiene il **nodo precedente** ad i , nel cammino minimo.
- `i[]` → vettore che annota il tempo in cui scopre un nodo (bianco → grigio).
- `f[]` → vettore che annota il tempo in cui il nodo è stato visitato (grigio → nero).

Partendo da un nodo, si percorre un qualsiasi cammino **fino** a quando si raggiunge un nodo che **non permette di trovare nodi bianchi** (mai visitati). Una volta giunti a tale nodo, si **torna** al nodo **precedente** e si **ripete** ricorsivamente il ragionamento. Una volta terminata la visita, essa riparte ricorsivamente sui restanti nodi bianchi “**staccati**” (se ci sono).

1. **Inizializza** i due vettori e una variabile time a 1.
2. Per ogni **nodo bianco** v del grafo si chiama `DFS_visit(G, v, time)`:

- **colora** v di grigio
- imposta il tempo di **inizio visita di** v a **time**
- **incrementa** time
- **per ogni nodo** u nella lista di adiacenza di v
 - se il colore è bianco (altrimenti non fa nulla)
 - salva** in pi l'**origine** da cui si è arrivati
 - richiama `DFS_visit(G, u, time)` e **salva** il valore di ritorno in **time**
 - **imposta il tempo di fine** visita di v a **time**
 - **incrementa** time
 - **colora** v di nero
 - **ritorna** time

```
DFS(G)                                // Θ(|V|^2) con matrici di Adj e Θ(|V|+|E|) con liste di Adj
  for (each v in V)                    // eseguito Θ(|V|) volte → inizializza i tre vettori
    color[v] = BIANCO                // All'inizio i nodi sono tutti bianchi
    π[v] = NIL                        // All'inizio non si sa nulla dei predecessori
  time = 0
  for (each v in V)
    if (color[v] == BIANCO)          // se ci sono nodi bianchi "staccati", la visita riparte
      DFS_visit(G, v, time)
```

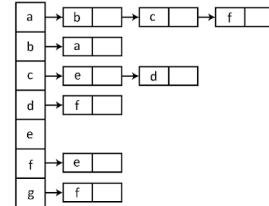
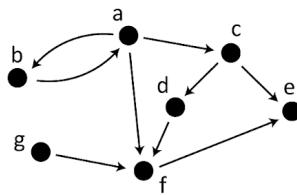
```
DFS_visit(G, v, time)
  color[v] = GRIGIO                  // arrivato sul nodo, lo colora di grigio
  i[v] = time                         // salvo il tempo di inizio visita
  time++
  for (each u in Adj[v])             // eseguito Θ(|V|) volte → scandisce adj di u
    if (color[u] == BIANCO)
      π[u] = v                      // annoto che u è stato scoperto da v
      time = DFS_visit(G, u, time)   // chiamata ric. dal nodo u (che diventa grigio, ...)
    f[v] = time                      // Θ(1) → salvo il tempo di fine visita
    time++                           // Θ(1) → incremento time
  color[v] = NERO                    // Θ(1) → colora di nero il nodo
  return time
```

Complessità

Il costo di della visita DFS è $\Theta(|V|) + \Theta(|V|) + \Theta(\sum_{v \in V} |Adj[v]|)$ che nel caso si usino **matrici di adiacenza** vale $\Theta(|V|) + \Theta(|V|) + \Theta(\sum_{v \in V} |[v]|) = \Theta(|V^2|)$, mentre nel caso di uso di **liste di adiacenza** il costo complessivo è $\Theta(|V|) + \Theta(|V|) + \Theta(\sum_{v \in V} |L_G[v]|) = \Theta(|V|) + \Theta(|V|) + \Theta(|E|) = \Theta(|V| + |E|)$.

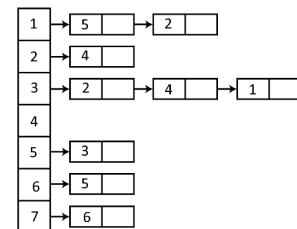
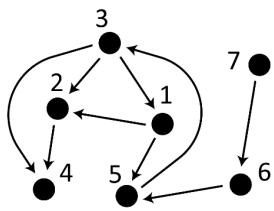
Esempio 1

Dato il grafo qui qui sotto, descrivere la sua visita DFS partendo dal nodo *a*:



- Tempo 1.** Partendo da *a*, si colora *a* di grigio.
- Tempo 2.** Da *a* → *b*, che diventa grigio.
- Tempo 3.** Da *b*, non ci sono archi che arrivano a nodi bianchi: si colora *b* di nero.
- Tempo 4.** Da *a* → *c*, che diventa grigio.
- Tempo 5.** Da *c* → *d*, che diventa grigio.
- Tempo 6.** Da *d* → *f*, che diventa grigio.
- Tempo 7.** Da *f* → *e*, che diventa grigio.
- Tempo 8.** Da *e*, non ci sono archi che arrivano a nodi bianchi: si colora *e* di nero.
- Tempo 9.** Da *f*, non ci sono archi che arrivano a nodi bianchi: si colora *f* di nero.
- Tempo 10.** Da *d*, non ci sono archi che arrivano a nodi bianchi: si colora *d* di nero.
- Tempo 11.** Da *c*, non ci sono archi che arrivano a nodi bianchi: si colora *c* di nero.
- Tempo 12.** Da *a*, non ci sono archi che arrivano a nodi bianchi: si colora *a* di nero.
- Tempo 13.** Partendo da *g*, si colora *g* di grigio.
- Tempo 14.** Da *g*, non ci sono archi che arrivano a nodi bianchi: si colora *g* di nero.

Esempio 2



$\pi = [N, 3, 5, 2, 1, N, N]$, $color = [N, N, N, N, N, N, N]$, $i = [1, 4, 3, 5, 2, 11, 13]$, $f = [10, 7, 8, 6, 9, 12, 14]$.

`DFS_visit(G, 1) → DFS_visit(G, 5) → DFS_visit(G, 3) → DFS_visit(G, 2) → DFS_visit(G, 4).`
`DFS_visit(G, 6). DFS_visit(G, 7).`

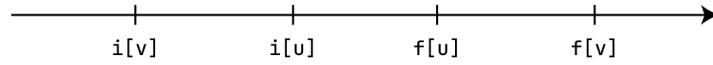
In π , c'è scritto che durante la visita si sono costruiti una foresta composta da 3 alberi (3 nodi con parent *nil*) che hanno come radice 1, 6 e 7. Scandendo π è possibile ricostruire gli alberi di cui uno è una sequenza e due sono dei nodi senza figli: 1-5-3-2-4, 6 e 7.

Proprietà

◆ Al termine di $DFS(G)$:

- i valori *nil* in π sono i nodi nei quali DFS_visit è stata chiamata da DFS .
- π contiene un **insieme di alberi** detto **foresta**.

Teorema: un nodo $u \in V$ è **descendente** di $v \in V$ nella foresta DFS sse $[i[u], f[u]] \subseteq [i[v], f[v]]$, ossia i tempi di inizio e fine visita del nodo u sono compresi nei tempi di inizio e fine visita di v :



◆ **Teorema delle parentesi:** Per ogni nodo $a, b \in V$ deve valere una delle seguenti:

$$[i[v], f[v]] \cap [i[u], f[u]] = \emptyset \quad [i[v], f[v]] \subseteq [i[u], f[u]] \quad [i[u], f[u]] \subseteq [i[v], f[v]]$$

◆ **Osservazione:** durante la visita, gli archi possono essere di tre tipi:

1. **grigio → bianco**: archi che sicuramente finisco nell'albero di visita.
2. **grigio → grigio (back edge)**: rappresentano un ciclo.
3. **grigio → nero**: possono essere:
 - *forward edge*, ossia archi da un nodo u ad un discendente di u nell'albero di visita.
 - *cross edge*, ossia archi da un nodo u ad un nodo v che non è discendente di u e quindi si trovano in due alberi diversi nella foresta DFS.

Teorema: Se durante la visita DFS si trova almeno un *back edge* allora in G c'è almeno un ciclo. Nota: il numero di cicli e *back edge* non coincidono: si hanno al massimo $\Theta(n^2)$ cicli mentre i back edge sono al più $O(n^2)$.

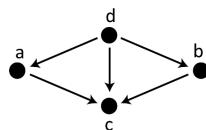
◆ **Teorema cammino bianco:** Dato un cammino composto da nodi tutti di colore bianco che ha origine nel nodo v e termina nel nodo a al tempo $i[v]$, a sarà certamente un discendente di v nella foresta DFS.

DAG: Grafi orientati aciclici

Sono la tipologia di grafi più **semplici** in assoluto in quanto assomigliano a degli alberi. Le **differenze** con questi ultimi sono il fatto che nei DAG un **nodo** può avere **più di un genitore** mentre negli alberi no. Dato che non sono presenti dei cicli (ogni grafo orientato e aciclico ha almeno un nodo senza archi uscenti e almeno un nodo senza archi entranti), è possibile stabilire un **ordinamento** dei nodi, detto **topological sort**.

TP: Topological Sort

Definizione: il topological sort (TS) è un ordinamento dei nodi di un DAG tale che per ogni arco $\{u, v\} \in E$ si ha $u \in V$ viene prima di $v \in V$ nel topological sort.



Ad esempio, nel DAG qui sopra:

- $a < b < c < d \rightarrow$ non è un TS in quanto $\{d, a\} \not\Rightarrow d < a$.
- $d < a < b < c \rightarrow$ è un TS in quanto per ogni arco $\{u, v\}$ si ha che u viene prima di v .
- $d < b < a < c \rightarrow$ è un TS in quanto per ogni arco $\{u, v\}$ si ha che u viene prima di v .

Algoritmo per la verifica della presenza di cicli

Dato un grafo orientato, stabilire se esso contiene cicli (stabilire se è o meno un DAG):

G è aciclico se e solo se in $DFS(G)$ non si trova nessun arco grigio \rightarrow grigio. Per finalizzare ciò basta aggiungere un test sul colore alla procedura di DFS. In questo modo si risolve l'esercizio in $\Theta(|V| + |E|)$ mediante l'uso di liste di adiacenza e in $\Theta(|V^2|)$ se si fa uso di matrici di adiacenza.

```

DFS(G) { ... } //  $\Theta(|V|^2)$  con matrici di Adj e  $\Theta(|V|+|E|)$  con liste di Adj

DFS_visit_ciclic_check(G, v, time) {

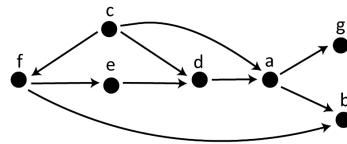
    color[v] = GRIGIO          //  $\Theta(1) \rightarrow$  arrivato sul nodo, la prima cosa da fare è colorarlo
    i[v] = time                //  $\Theta(1) \rightarrow$  salvo il tempo di inizio visita
    time++                      //  $\Theta(1) \rightarrow$  incremento time

    for (each u in Adj[v]) {    // eseguito  $\Theta(|V|)$  volte  $\rightarrow$  scandisce la lista di adj di u
        if (color[u] == BIANCO) {
            pi[u] = v           // annoto che u è stato scoperto da v
            time = DFS_visit_ciclic_check(G, u, time) // chiamata ric. dal nodo u
        }
        if (color[u] == GRIGIO) { // se trovo un arco grigio  $\rightarrow$  grigio ho un ciclo
            print("Il grafo è ciclico")
        }
    }

    f[v] = time                //  $\Theta(1) \rightarrow$  salvo il tempo di fine visita
    time++                      //  $\Theta(1) \rightarrow$  incremento time
    color[v] = NERO             //  $\Theta(1) \rightarrow$  colora di nero il nodo
    return time
}
  
```

Algoritmo per trovare un TS

Dato che G è un DAG (ad esempio quello rappresentato qui sotto), trovarne un ordinamento topologico:



Idea 1:

1. **Cerca** quali nodi possono finire in fondo al TS, ossia quelli che non hanno archi uscenti in $O(|V|)$.
2. **Elimina** uno (insieme ai suoi archi entranti) in $O(|V| + |E|)$.
3. **Aggiunge** tale nodo in fondo alla lista del TS ($\Theta(1)$).
4. **Itera** per $\Theta(|V|)$ volte.

Si ottiene così una procedura con costo $O(|V| \cdot (|V| + |E|))$.

Idea 2:

Si basa sull'idea che quando un nodo diventa nero, sicuramente tutti i nodi che raggiunge sono già neri:

1. **Inizializza** una lista vuota.
2. Usa la **visita DFS** per trovare i nodi senza archi uscenti (quelli che diventano neri).
3. Quando un nodo diventa nero, lo si **inserisce** in testa ad una lista.

Si ottiene così una procedura con costo $\Theta(|V| + |E|)$.

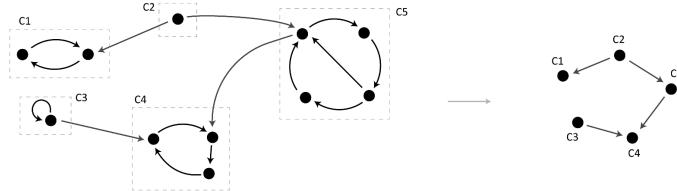
```

DFS_TS(G) {
    // Θ(|V|^2) con matrici di Adj e
    // Θ(|V|+|E|) con liste di Adj
    ...
    L = null
    ...
    DFS_visit_TS(G, u, time, L)
    ...
    return L
}

DFS_visit_TS(G, v, time, L) {
    ...
    Inserisci_in_testa(L, v)
    ...
    return L
}
  
```

Cosa fare in presenza di cicli (nessun nodo senza archi uscenti)

Nel caso in cui il grafo sia composto da cicli è possibile che non ci sia alcun nodo senza archi uscenti. Per superare il problema si **raggruppano** i nodi all'interno di tali cicli ottenendo dei *cluster* di nodi, detti **componenti fortemente connesse**, che se usati come nodi, formano un DAG. Ad esempio:



Definizione: Una componente fortemente connessa (scc) $C_i \subseteq V$, in un DAG G , è un insieme *massimale* (raggruppa più nodi possibili) di nodi mutamente raggiungibili. Le scc C_i rappresentano una *partizione* di V cioè:

- $C_1 \cup C_2 \cup \dots \cup C_m = V$. *(l'unione di tutte le scc forma l'insieme dei nodi del grafo)*
- $C_i \neq \emptyset$. *(ogni scc deve avere almeno un nodo)*
- $C_i \cap C_j = \emptyset$. *(due scc non condividono nessun nodo)*

Definizione: Il grafo delle scc è sempre un DAG (quindi aciclico) e viene identificato con $G_{scc} = (V_{scc}, E_{scc})$:

- V_{scc} è l'insieme delle componenti fortemente connesse C_i
- E_{scc} è l'insieme degli archi $(C_i, C_j) \in E_{scc}$ tali che $i \neq j$ e $\exists u \in C_i, v \in C_j$ tali che $(u, v) \in E$.

Algoritmo per il calcolo delle scc (Kosaraju)

Per determinare le componenti fortemente connesse si fa uso di $DFS(G)$, che calcola una foresta di visita DFS composta da diversi alberi in cui ognuno di essi è l'**unione di intere** scc del grafo.

1. Esegue $DFS(G)$ per ottenere il vettore dei tempi di fine visita f .
2. Calcola G^{-1} (grafo G con gli archi invertiti).
3. Esegue $DFS(G^{-1})$ e nel ciclo for di DFS procedo in ordine decrescente rispetto a f .

Gli alberi costruiti da questo algoritmo sono le componenti fortemente connesse di G . Il costo è $\Theta(|V| + |E|)$.

Grafi non orientati ed alberi

Un grafo non orientato è un **albero** se e solo se è **connesso** e **aciclico** ossia se ha una sola componente fortemente connessa. Le seguenti definizioni sono equivalenti:

1. G è un albero.
2. $\forall u, v \in V \ \exists! u \leftrightarrow v$.
3. G è connesso e se viene rimosso un arco si sconnette.
4. G è connesso e $|E| = |V| - 1$.
5. G è aciclico e $|E| = |V| - 1$.
6. G è aciclico e aggiungendo un arco G diventa ciclico.

Per decidere se un grafo è un albero bisogna decidere se il grafo è

- connesso → in π c'è un solo *nil*.
- aciclico → non ci sono archi grigio-grigio oltre a archi della forma $\{x, \pi[x]\}$.

Per farlo si usa DFS e il costo è $O(|V| + |E|)$.

Grafi pesati

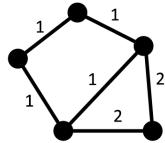
Un **grafo** non orientato, connesso e pesato $G = (V, E, W)$ è composto da V insieme dei **nodi**, E insieme degli **archi** e $W: E \rightarrow R^+$ che è una funzione che attribuisce ad ogni arco un **peso** positivo. Ci sono 2 problemi:

1. **albero minimo di copertura** (MST - con Kruskal e Prim).
2. **albero dei cammini minimi** (SSSP e APSP - con Dijkstra e Floyd Warshall).

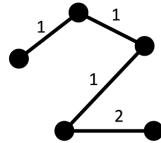
Albero minimo di copertura (MST)

Un **albero minimo di copertura** di un grafo G è un grafo G' che:

1. G' contiene tutti i nodi di G .
2. G' è connesso.
3. G' di peso (totale) minimo.



$$W(G) = 7.$$



$$W(G') = 5.$$

Definizione di taglio

Un taglio è una **partizione** dei nodi di un grafo in due blocchi e viene indicato con $(S, V \setminus S)$ dove $S \subseteq V$.

- Un arco $\{u, v\}$ **attraversa** il taglio $(S, V \setminus S)$ se $u \in S$ e $v \in V \setminus S$ oppure $u \in V \setminus S$ e $v \in S$.
- Un insieme di archi $A \subseteq E$ **rispetta** il taglio $(S, V \setminus S)$ se nessun arco di A attraversa il taglio.

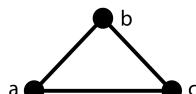
Definizione di leggero

Un arco $\{u, v\}$ è **leggero** per il taglio $(S, V \setminus S)$ se attraversa il taglio $(S, V \setminus S)$ ed è di **peso minimo** tra tutti gli archi che attraversano il taglio.

Definizione di arco safe

- dato $G = (V, E, W)$ non orientato, connesso e pesato,
- dato T mst di G ,
- dato $A \subseteq T$,
- dato $\{u, v\} \notin A$,

diciamo che $\{u, v\}$ è **safe** per se $A \cup \{\{u, v\}\} \subseteq T'$ con T' MST. Ad esempio nel grafo qui sotto $\{b, c\}$ è safe per A perché $\{\{a, b\}, \{b, c\}\}$ è un MST e anche $\{a, c\}$ è safe per A perché $\{\{a, b\}, \{a, c\}\}$ è un MST.



$$\begin{aligned} T &= \{\{a, b\}, \{b, c\}\} \text{ MST} \\ A &= \{\{a, b\}\} \end{aligned}$$

Teorema

- dato $G = (V, E, W)$ non orientato, connesso e pesato,
- dato un taglio $(S, V \setminus S)$,
- dato T mst di G ,
- dato $A \subseteq T$ e A rispetta il taglio $(S, V \setminus S)$,

se $\{u, v\}$ è leggero per il taglio, allora è **safe** per A .

MST(G)

```

A = vuoto           // conterrà il MST alla fine
while (|A| < |V| - 1)
    trova {u, v} safe per A // come si fa??? → kruskal o Prim
    A = A u {{u,v}}
return A

```

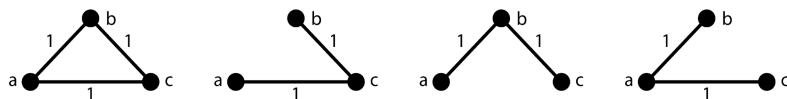
Domande interessanti

1. Che caratteristiche deve avere un grafo per avere due mst distinti?

1. deve avere archi con pesi ripetuti.
2. deve avere almeno un ciclo.
3. deve avere almeno un ciclo con 2 archi con lo stesso peso

2. T mst di G contiene tutti gli archi di peso minimo?

No, ad esempio in questo grafo (supponendo tutti gli archi di peso 1) ci sono 3 mst diversi:



3. T mst di G contiene un cammino di peso minimo per ogni coppia di nodi di G ?

No, ad esempio il grafo sulla destra è un mst di G ma non contiene il cammino minimo tra $\{b, c\}$:



4. T mst di G contiene almeno un cammino di peso minimo?

Sì, per la costruzione di un mst si prendono sempre gli archi di peso minimo scartando quelli di peso massimo.

Algoritmo di Kruskal

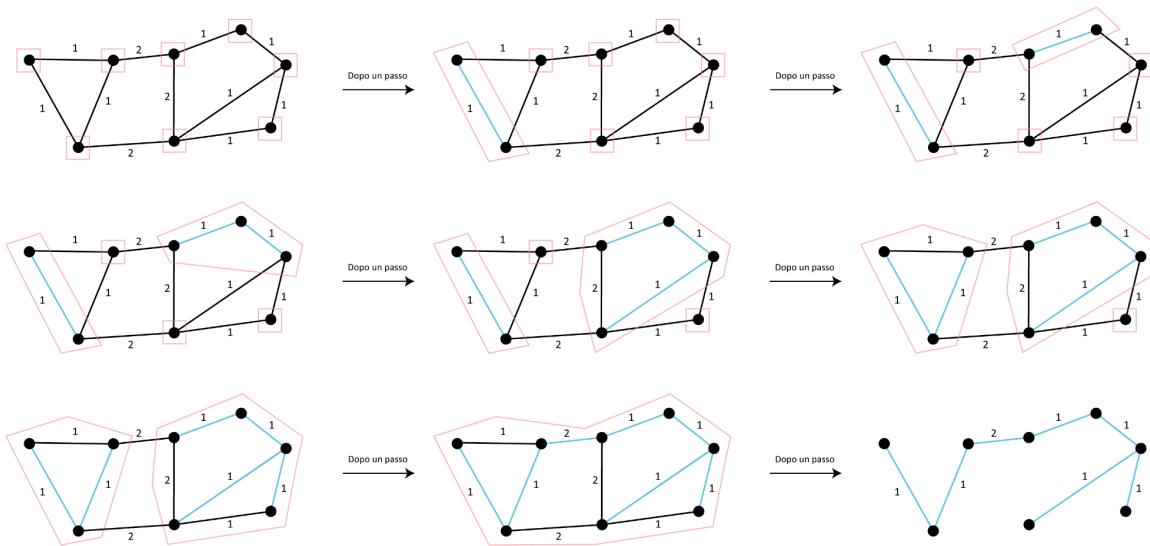
L'algoritmo di Kruskal è un algoritmo usato per **calcolare un MST** di un grafo non orientato, connesso e aciclico. Sfrutta gli **insiemi disgiunti** e si basa sul seguente **corollario**:

Corollario

Dato $G = (V, E, W)$ non orientato, connesso e pesato, dato $A \subseteq T$ con T mst, se A è formato dalle componenti fortemente connesse $C_1 \dots C_k$ e $\{u, v\}$ è di peso minimo tra archi che connettono due componenti distinte allora $\{u, v\}$ è safe per A .

L'**idea** dell'algoritmo di Kruskal è:

- **Ordina** gli archi in ordine crescente di costo.
- **Inizializza** una lista vuota $A \subseteq T$ dove creare il mst.
- **Crea** le scc con operazioni di **make(v)**.
- Il **corollario** dice “tra tutti gli archi rimasti sceglie uno di peso minimo”: se tale arco connette due scc distinte (**find(u) ≠ find(v)**):
 - si effettua una **unione** (**union(u, v)**),
 - altrimenti si **procede** con l'analisi degli archi,



```
Kruskal(G) // O(|E|log(|V|)), G=(V,E,W) non orientato, connesso e pesato
  sort(E, W) // O(|E| log(|E|)) → ordina gli archi
  A = ∅ // A dopo i passi A = {C1 - C2 - C3} e al termine A = {C1}
  for each (v in V) make(v) // eseguito θ(|V|) volte → Crea l'insieme singoletto (img 1)

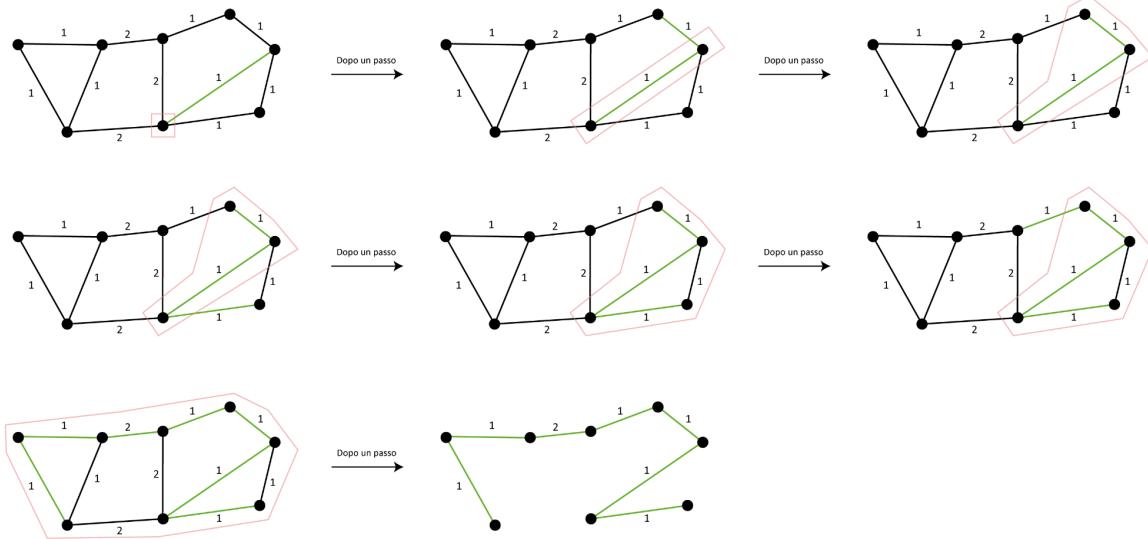
  for each ({u,v} in E) // eseguito θ(|E|) volte → Analizza archi in ordine crescente
    if (find(u)≠find(v)) // se i due rapp. sono diversi (i due insiemi sono disgiunti)
      A = A U {{u, v}} // aggiungo l'arco all'insieme A
      union(u, v) // unisco le due componenti
  return A
```

Usando **alberi con union by rank e pc** si ottiene il costo di $O(|E|\log(|E|) + O(|E|\alpha(|V|, |E|)))$ in cui dato che $\alpha(|V|, |E|)$ cresce lentissima il costo complessivo viene maggiorato a $O(|E|\log(|V|))$. Il costo **si abbassa** se si riesce ad **ordinare efficientemente** gli archi (a.e. gli archi hanno un **peso limitato** superiormente da una costante, in questo modo si può usare un algoritmo lineare per ordinare data l'hp sull'input).

Algoritmo di Prim

L'algoritmo di Prim è un algoritmo, alternativo a quello di Kruskal, usato per **calcolare un MST** di un grafo non orientato, connesso e aciclico. Per questo algoritmo si sfrutta una **min heap** per implementare una coda con priorità. Partendo da un nodo s , che si trova all'interno di un taglio:

1. **Sceglie** l'arco uscente dal taglio con peso minore.
2. **Si estende** il taglio anche a tale nuovo nodo.
3. **Si ripete** dal punto 1.



```

Prim(G,s)                                //  $O(|E|\log(|V|))$ , G=(V,E,W) non orientato, connesso e pesato

    for each (v in V)      // eseguito  $\Theta(|V|)$  volte
        pi[v] = null          // pi[v] contiene il nodo che connette v al MST (memorizza il MST)
        key[v] = inf            // key[v] contiene il peso dell'arco che connette v al MST

    key[s] = 0                  // s ha priorità 0 → più alta di tutti
    Q = V                      // inserisce in Q (coda con priorità key) i nodi
    BuildMinHeap(Q, key)      //  $\Theta(|V|)$ 

    while (Q ≠ ∅)           // eseguito  $\Theta(|V|)$  volte
        u = extractMin(Q)       // cerca min  $\Theta(1)+\text{heapify } \Theta(\log n) \rightarrow O(\log |V|)$ 
        for each (v in Adj[u])   // eseguito  $O(|adj[u]|)$  volte
            if (v in Q && key[v] > W({u,v}))
                key[v] = W({u, v})
                pi[v] = u
                decresekey(Q,v)    //  $O(\log(|V|))$  sistema nella min heap il nodo v scambiandolo
                                    // con il suo genitore fino alla sua posizione corretta

```

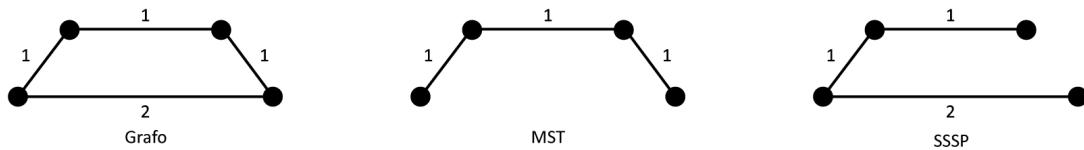
Albero dei cammini minimi (SSSP e APSP).

Il **peso** di un cammino è la **somma** dei singoli pesi degli archi che compongono un cammino. I cammini minimi possono essere da:

- **una sorgente s** (SSSP - Dijkstra): fissato s determinare $\forall v \in V$ un cammino di peso minimo da s a v .
- **ogni possibile sorgente** (APSP - Floyd Warshall): $\forall s \forall v$ determinare un cammino di peso minimo.

Differenza tra MST e SSSP

- **MST** → spendere il meno possibile per collegare tutti i nodi (ignora il numero di archi).
- **SSSP (o APSP)** → trovare la strada più *breve* per collegare due nodi (ignora i pesi).



Algoritmo di Dijkstra per il calcolo di SSSP

L'algoritmo di Dijkstra è un algoritmo usato per **cercare i cammini minimi** in un grafo con o senza ordinamento, ciclico e con pesi non negativi sugli archi.

```

Dijkstra(G, s)           // O(|E|log(|V|)), G=(V,E,W) non orientato, connesso e pesato
    for each (v in V)      // eseguito Θ(|V|) volte
        pi[v] = null       // pi[v] contiene il nome del nodo che connette v al SSSP
        d[v] = inf          // d[v] contiene il peso del cammino minimo che connette d a s
    d[s] = 0
    Q = V                  // Q coda con priorità d
    BuildMinHeap(Q, d)     // Θ(|V|)
    while (Q ≠ ∅)          // eseguito Θ(|E|) volte
        u = extractMin(Q)   // estrazione Θ(1)+heapify Θ(log n)→O(log|V|)
        for each (v in Adj[u])
            if (v in Q && d[v] > d[u] + W({u, v}))
                d[v] = d[u] + W({u, v})
                pi[v] = u
                decreaseKey(Q, v) // O(log(|V|))

```

Algoritmo per il calcolo di APSP: floyd warshall

$\forall s \forall v$ si determinare un cammino di peso minimo da s a v . Per farlo si crea una matrice e si richiama per ogni riga l'algoritmo di Dijkstra.

```

APSP(G=(V, E, W))      // O(|V| · |E|log(|V|))
    for each (v in V)
        Dijkstra(G, s)

```

Esercizi primo parziale

[Merge] Esercizio 1.

Dato A vettore di interi di lunghezza n e dato k , decidere se esistono $A[i]$ e $A[j]$ con $i \neq j$ tali che $A[i] + A[j] = k$.

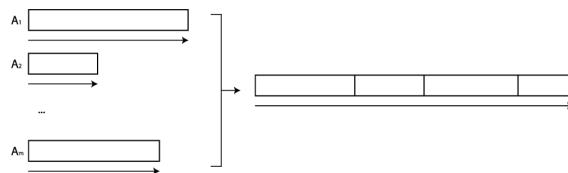
- Ordino A con mergeSort o heapSort in $O(n \log n)$.
- Chiamo $A[1] = x$ e $A[n] = y$ e controllo in $\Theta(n)$:
 - se $x + y = k \Rightarrow$ ritorno i e j ,
 - se $x + y < k \Rightarrow$ richiamo la procedura ricorsivamente su $i + 1$ e j ,
 - se $x + y > k \Rightarrow$ richiamo la procedura ricorsivamente su i e $j + 1$,
 - se $i = j \Rightarrow$ non ci sono due elementi che sommati danno k .

Il costo della procedura searchSum è $T(n) = T(n - 1) + \Theta(1) = O(n)$ se $n > 1$.

<pre>algoritmo(A, k) { mergeSort(A, 1, A.len) return searchSum(A, 1, A.len, k) }</pre>	<pre>searchSum(A, 1, A.len, k) { // O(n) if (p == q) // Θ(1) return "non trovati" else { if (A[p]+A[q] == k) return (p, q) else if (A[p]+A[q] < k) return searchSum(A, p+1, q, k) // T(n - 1) else return searchSum(A, p, q-1, k) // T(n - 1) } }</pre>
--	--

[Merge] Esercizio 2.

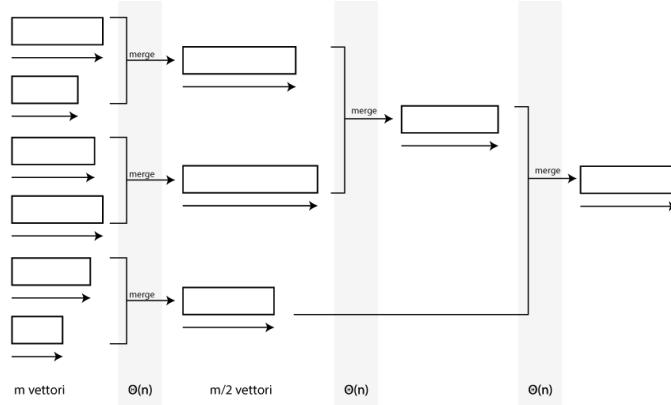
Dati m vettori $A_1 \dots A_m$ ordinati contenenti in tutto n elementi, fornire un algoritmo per ordinare gli n elementi.



[generalizzazione della procedura merge ad m vettori anziché 2] La soluzione è scorrere i vettori $A_i[j]$, cercando l'elemento minore tra tutti quelli in posizione j . Una volta trovato, lo si scrive nel vettore risultante e si incrementa l'indice j . In questo modo si faranno $\Theta(n)$ iterazioni (dato che alla fine A conterrà n elementi e per ogni iterazione se ne aggiunge 1) e, ad ogni iterazione, si trova il minimo tra $O(m)$ elementi in $O(m)$. In questo modo si ottiene una complessità totale di $O(n \cdot m)$ che nel caso peggiore diventa anche $\Theta(n \cdot m)$. Per ottimizzare questa soluzione si può:

- Costruire all'inizio una min heap H con tutti gli m minimo in $\Theta(m)$.
- Ad ogni iterazione, si estrae il minimo da H , lo si copia in A e si inserisce in H il successore del minimo estratto in tempo $O(\log m)$.

In questo modo si ha una soluzione in $O(n \log m)$. Un altro modo per ottimizzare questa soluzione è chiamare merge a due a due. In questo modo si ottiene un albero di chiamate a merge che costano $\Theta(n)$ alto $\Theta(\log m)$. Così si ottiene una procedura che ha costo $\Theta(n \log m)$.



[Merge] Esercizio 3.

Siano A e B vettori ordinati di lunghezza n . Trovare gli elementi comuni tra A e B .

1. Una soluzione che costa $\Theta(n + m)$ è quella di usare un merge modificato sui due vettori, che vada ad unirli (tenendo conto della provenienza delle chiavi) in un unico ordinato. Successivamente basta scorrere tale array, controllando le posizioni adiacenti a quella in esame e inserendo tale chiave in un nuovo vettore se nelle posizioni adiacenti compare una chiave con stesso valore ma provenienza diversa.
2. Un'altra soluzione in $\Theta(m \log n)$ è quella che usa la ricerca binaria di ogni elemento di B in A .

[Select] Esercizio 4.

Si ha un algoritmo $Select(A, n/2)$ che calcola la mediana in $O(n)$. Scrivere un algoritmo $O(n)$ per $Select(A, i)$.

```
selectNew(A, p, q, i) { // O(n)
    if (p == q)
        return A[i]
    else {
        j = select(A, p, q)           % j è la posizione attuale del mediano
        if ((p+q)/2 == i)
            return j
        else {
            partition(A, p, q, j)   % partition con perno j
            if (i < (p+q)/2)
                return selectNew(A, p, (p+q)/2-1, i)
            else
                return selectNew(A, (p+q)/2+1, q, i)
        }
    }
}
```

L'equazione ric. di complessità è $T(n) \leq T(n/2) + \Theta(n) \Rightarrow \sum_{i=0}^{h-1} b\left(\frac{n}{2^i}\right) + a = bn \cdot c + a \Rightarrow T(n) \leq O(n)$.

[Select] Esercizio 5.

Scrivere un algoritmo $O(n)$ che dato A di lunghezza n e un intero positivo $k \leq n$, trova i k numeri piú vicini alla mediana. A.e. $k = 3$ significa che bisogna trovare i 3 elementi piú vicini alla mediana se A fosse ordinato.



- Prima di tutto trovo la mediana M con l'algoritmo $select(A, 1, n, n/2)$.
- Costruisco un vettore B con le distanze $B[i] = M - A[i]$.
- Uso $select(B, 1, n, k)$ sul vettore delle distanze e salvo il valore che ritorna in una variabile n .
- Uso $partition$ su B con perno n .

[BST] Esercizio 6.

Dato un vettore di interi A ordinare A sfruttando un BST.

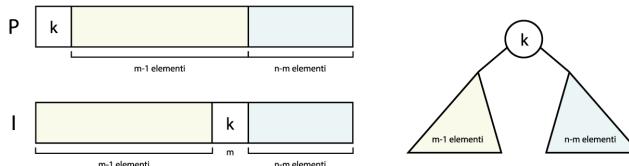
Usa la procedura di inserimento per creare un BST e poi richiamo la visita in-order. In questo modo si ottiene un vettore ordinato rappresentante A . Nel caso peggiore, (A ordinato) questa procedura costa $\Theta(n^2)$. Nel caso migliore (albero è bilanciato), ha costo $O(n \log n)$. Non é in place dato che si usa un albero e usa la in order.

```
Sort(A) {                                 $O(n^2)$ 
    T = createNewBST(...)                //  $\Theta(1)$ 
    for (i = 1 to A.len) {                  //  $n \cdot (\Theta(1) + O(n)) = O(n^2)$ 
        BSTInsert(T, createNewNode(A[i]))
    }
    inOrder(T, A)                         //  $\Theta(n)$ 
}
```

[BST] Esercizio 7.

P vettore contenente il risultato di una visita in pre order di un BST. Scrivi una procedura per ricostruire il BST.

Un albero binario non può essere ricostruito data una visita in preorder. Tuttavia un BST si. Data la visita in pre order, si sa che il primo elemento del vettore P rappresenta la radice. $P[0]$ è poi seguito da ciò che sarà il sottoalbero sinistro, e poi il sottoalbero destro.



- Copio P in un nuovo vettore I e lo ordino (I è il risultato della visita inorder sul BST).
- Inserisco in T il nodo $P[0] = k$.
- Cerco k in I e trovo che si trova in una certa posizione m .
- Si fa una chiamata ricorsiva sulla parte sinistra di lunghezza $m - 1$ per costruire il sottoalbero sinistro.
- Si fa una chiamata ricorsiva sulla parte destra di lunghezza $n - m$ per costruire il sottoalbero destro.

<pre> ReconstructBST(P) { I = newArray(P.len) // $\Theta(1)$ Copy(P, I) // $\Theta(n)$ Sort(I) // $\Theta(n \log n)$ T = createNewBST(...) // $\Theta(1)$ Reconstruct(P, I, T) // $\Theta(1)$ return T // $\Theta(1)$ } </pre>	<pre> Reconstruct(P, I, a, b, c, d, x) { x.key = P[a] if (a < b) { j = search(I, c, d, P[a]) // $\Theta(\log n)$ if (j > c) { y = createNewNode() y.parent = x x.left = y Reconstruct(P, I, a+1, a+j-c, j-1, y) } } } </pre>
---	---

[BST] Esercizio 8.

[Visita di Morris] Sia T un BST, è possibile ottenere una visita inorder in place e in tempo lineare?

Sì è possibile partendo dal minimo elemento presente nel BST, stampando la chiave e applicando iterativamente la procedura successore.

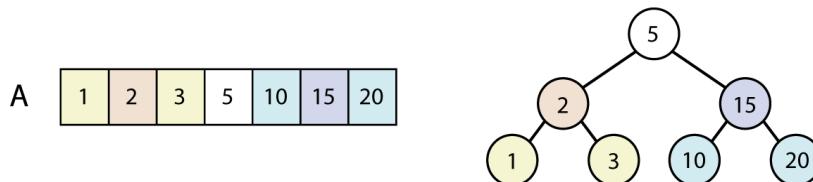
Nonostante la procedura possa sembrare quadratica, in realtà è lineare dato che la procedura successore non deve ogni volta partire dalla radice dell'albero e percorrere tutti gli n elementi del BST. Infatti, immaginando qualche esempio, si passa per uno stesso elemento dell'albero al massimo 3 volte (l'ultimo figlio destro del figlio sinistro quando cerca il suo successore, il se stesso e dopo aver cercato il suo successore).

<pre> inOrder(x) { // $\Theta(n)$ if (x ≠ nil) { inOrder(x.left()) print(x.key) inOrder(x.right()) } } </pre>	<pre> inOrderInPlace(T) { // $\Theta(n)$, in place x = BSTMin(T) while (x ≠ nil) { print(x.key) x = BSTSuccesor(x) } } </pre>
---	---

[BST] Esercizio 9.

Dato A vettore ordinato costruire un BST T contenente le chiavi di A di altezza $O(\log n)$ con $n = A.len$.

- Uso l'elemento che si trova in $\text{floor}((p + q)/2)$ come radice dell'albero.
- Richiamo la procedura sulla parte sinistra del vettore, cioè $A[p, \text{floor}((p + q)/2) - 1]$.
- Richiamo la procedura sulla parte destra del vettore, cioè $A[\text{floor}((p + q)/2) + 1, q]$.



[BST] Esercizio 10.

Dati T_1 e T_2 BST tali che l'altezza di T_1 è maggiore di quella di T_2 e tutte le chiavi di T_1 sono minori di quelle di T_2 . Si vuole fondere i due alberi in un unico.

- Si chiama inorder(T_1) per ottenere A_1 ordinato in $O(n)$.
- Si chiama inorder(T_2) per ottenere A_2 ordinato in $O(n)$.
- Si chiama merge(A_1, A_2) per ottenere A ordinato in $O(n)$.
- Trasformo A in un BST usando la procedura in $\Theta(n)$.

Nel caso non sia richiesto un albero bilanciato, basterebbe scendere in T_1 fino a raggiungere il nodo con la chiave massima ed agganciarci la radice di T_2 . Il costo in questo caso sarebbe $O(h)$ e $h = (\min(h_1, h_2))$.

[BST] Esercizio 11.

Esistono due alberi binari diversi tali che le visite in pre ordine e in post ordine producono lo stesso output? Esistono due BST diversi tali che le visite in pre ordine e in post ordine producono lo stesso output? Spiegare la risposta.

Si, due alberi binari diversi tali che le visite in pre ordine e in post ordine producono lo stesso output esistono (si prendano per esempio due alberi simmetrici: i risultati della visita in pre ordine e post ordine sono uguali). È invece impossibile trovare due BST diversi che diano lo stesso output con la visita in pre ordine e post ordine.

[Tabelle di Hash] Esercizio 12.

[Domanda 5 compitino 2019] Sia U un universo i cui elementi sono identificati da un chiave intera. Sia T una tabella di hash gestita tramite *open addressing* utilizzando $h(k, i) = (k + i) \bmod 1000$ come funzione di hash.

- Che dimensione può avere la tabella T , perchè?

La dimensione massima che la tabella può avere è $|T| = 1000$, in quanto tale funzione di hash restituisce numeri interi in un range $[0, 999]$.

- In quale intervallo varia l'indice i ? Perchè?

L'indice i rappresenta il tentativo di inserimento della chiave k nella tabella. Tale numero intero varia in un intervallo $[0, m - 1]$ dove m rappresenta la dimensione della tabella e quindi in questo caso tale indice varia nell'intervallo $[0, 999]$.

- Che dimensione può avere l'universo U ?

L'universo deve avere una cardinalità maggiore di 1000, altrimenti la tabella di hash non avrebbe senso e potenzialmente potrebbe essere molto più grande.

- La funzione rispetta l'ipotesi di hashing uniforme? Perchè?

L'ipotesi di hashing uniforme implica che tutte le permutazioni debbano essere equiprobabili. La funzione $h(k, i) = (k + i) \bmod 1000$ genera 1000 permutazioni diverse e quindi ci sono $1000! - 1000$ permutazioni che hanno probabilità 0 (a.e. la permutazione 0, 2, 1, 5, ... non viene mai generata per nessuna chiave dell'universo, mentre la permutazione 0, 1, 2, 3, ... ha probabilità maggiore di 0) e quindi non soddisfa l'ipotesi di hashing uniforme.

[Tabelle di Hash] Esercizio 13.

[Domanda 5 compitino 2018] Si devono memorizzare delle stringhe definite composte usando un alfabeto $\{a, c, g, t\}$, ognuna di lunghezza 10, in una hash table di lunghezza m , dove le collisioni sono gestite tramite

changing. Si propongono delle “buone” funzioni di hash nel caso in cui $m = 4^3 = 64$, $m = 2$ oppure $m = 2 \cdot (4^2) = 32$.

A vettore di lunghezza 10, in cui $A[i] \in \{a, c, g, t\}$. Come prima cosa si assegnano dei valori interi ad ogni carattere usando una funzione $f: \{a, b, c, d\} \rightarrow \{0, 1, 2, 3\}$: $f(a) = 0, f(b) = 1, f(c) = 2, f(d) = 3$.

- Nel caso di $m = 4^3 = 64$ abbiamo che $h(A) \in [0, \dots, 4^3 - 1]$. La funzione di hash più conveniente è

$$h(A) = (f(A[1]) \cdot 4^2 + f(A[2]) \cdot 4 + f(A[3]))$$

- Nel caso $m = 2$ tutti i vettori vanno nella posizione 0 o 1, tale funzione è $h(A) = f(A[1]) \text{ mod } 2$.

- Nel caso di $m = 2 \cdot (4^2) = 32$ abbiamo che $h(A) \in [0, \dots, 2^5 - 1]$. La funzione di hash più conveniente è

$$h(A) = (f(A[1]) + f(A[2]) \cdot 4 + (f(A[3]) \text{ mod } 2) \cdot 4^2)$$

[Tabelle di Hash] Esercizio 14.

[Domanda 6 compitino 2017] Bisogna memorizzare delle date nel formato $g/m/a$ nell’intervallo 01/01/1900 - 31/12/1999 (cioè $g \in [1, 31]$, $m \in [1, 12]$ e $a \in [1900, 1999]$) in una tabella di hash $H[0, \dots, l - 1]$ di dimensione l , nella quale le collisioni vengono gestite con chaining. Si assuma, per semplicità, che tutti i mesi abbiano 31 giorni. Quali delle tre funzioni di hash sono adatte allo scopo?

1. $h(g, m, a) = ((g - 1) + (m - 1) + (a - 1900)) \text{ mod } l$
2. $h(g, m, a) = (((g - 1) + (m - 1) + (a - 1900)) \cdot 100 \cdot 12 \cdot 31) \text{ mod } l$
3. $h(g, m, a) = ((g - 1) \cdot 100 \cdot 12 + (m - 1) \cdot 100 + (a - 1900)) \text{ mod } l$

La prima potrebbe andare bene, dato che $(g - 1) \in [0 \dots 30]$, $(m - 1) \in [0 \dots 11]$ e $(a - 1900) \in [0 \dots 99]$. Sommandoli ottengo numeri compresi tra $[0 \dots 140]$. Per poter essere una buona funzione di hash però, sicuramente $l \leq 140$.

La seconda funzione di hash non risulta una buona funzione dato che il calcolo $100 \cdot 12 \cdot 31$ verrà “inibito” dall’operazione di modulo. Tra l’altro, la funzione funziona bene solo se l non ha fattori in comune con $100 \cdot 12 \cdot 31$, dato che sennò alcune posizioni della tabella non verrebbero mai occupate.

Nella terza funzione, le cifre meno significative sono quelle dell’anno, che vengono inserite per prime dando vita alla stringa aa . Poi, moltiplicando per 100 il mese, otteniamo le due cifre relative al mese, arricchendo la stringa che diventerà $mmaa$. Moltiplicando ancora per 12 concateno le due cifre relative al giorno davanti, ottenendo così la data nel formato $ggmmaa$. Riassumendo, grazie a questa funzione si può ottenere un intero nella forma “ $ggmmaa$ ”, in cui le prime due cifre rappresentano il giorno, le due cifre seguenti il mese e le ultime due rappresentano l’anno. Quindi, facendo l’operazione di modulo, si ottiene il valore in cui la chiave $ggmmaa$ viene mappata.

[Tabelle di Hash] Esercizio 15.

[Domanda 6 compitino 2016] Bisogna memorizzare delle chiavi che sono multiple di 8 in una tabella di hash di dimensione m che gestisce le collisioni con chaining. Giudicare le seguenti funzioni:

- | | |
|---|--|
| 1. $h(k) = k \text{ mod } m$ | pessima, lascia tanti buchi vuoti nella tabella di hash. |
| 2. $h(k) = 8k \text{ mod } m$ | ancora peggio, salta di 64 celle. |
| 3. $h(k) = (k \text{ mod } 8) \text{ mod } m$ | male, mappa tutte le chiavi in 0. |

Una buona funzione di hash per questo problema è $h(k) = k/8 \text{ mod } m$, infatti, dividendo la chiave per 8, ci “sleghiamo” dal concetto di multiplo.

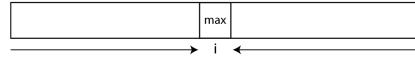
[Heap] Esercizio 16.

[Compitino 29-02-19] Data una min heap A di lunghezza n , è possibile ordinare A in ordine decrescente in tempo lineare?

No, non è possibile in quanto costruire una heap ha costo $\Theta(n)$ e ordinare un vettore, senza ipotesi aggiuntive sull'input, costa nel caso peggiore $\Omega(n \log n)$.

[Ricerca binaria] Esercizio 17.

È dato un vettore bitonico, ossia un vettore che fino ad un elemento in posizione i è crescente e dopo tale elemento diventa decrescente.



- Scrivere un vettore bitonico di lunghezza 4 contenente le chiavi 1, 2, 3, 4: $A = [1, 2, 4, 3]$.
- Quanti possibili vettori bitonici esistono con questi valori? $3 \cdot 2 = 6$ modi
- Scrivere una procedura ricorsiva efficiente per trovare il massimo di un vettore bitonico.

Si guarda l'elemento x in posizione centrale $(p + q)/2$ e lo si confronta con l'elemento y alla sua sinistra.

- se $y < x \Rightarrow$ scarta la porzione a sinistra di x con una chiamata ricorsiva `findMax(A, (p+q)/2, q)`.
- se $y > x \Rightarrow$ scarta la porzione a destra di x con una chiamata ricorsiva `findMax(A, p, (p+q)/2)`.

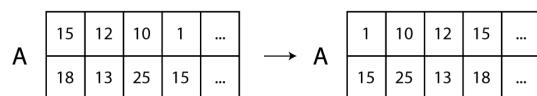
Tale procedura ha costo $O(\log n)$.

- Scrivere una procedura che renda il vettore bitonico un vettore ordinato.

Per farlo trovo il massimo in $O(\log n)$ e uso merge in $\Theta(n)$. In questo modo il costo complessivo sarà $\Theta(n)$.

[Generico] Esercizio 17.

Sia A un vettore di intervalli in cui ogni cella ha due campi $A[i].left$ e $A[i].right$ rappresentanti rispettivamente l'inizio e la fine dell'intervalle. Un valore k viene definito *coperto* se esiste almeno un intervallo nel vettore tale che $A[i].left \leq k \leq A[i].right$. Si vuole, dato un vettore di questo tipo, trovare un altro vettore di intervalli, che non contenga intervalli che si sovrappongono, senza perdere copertura.



- Ordino il vettore rispetto al campo *left* $\Rightarrow \text{mergeSort}(A, 1, A.\text{len}, \text{left})$.