

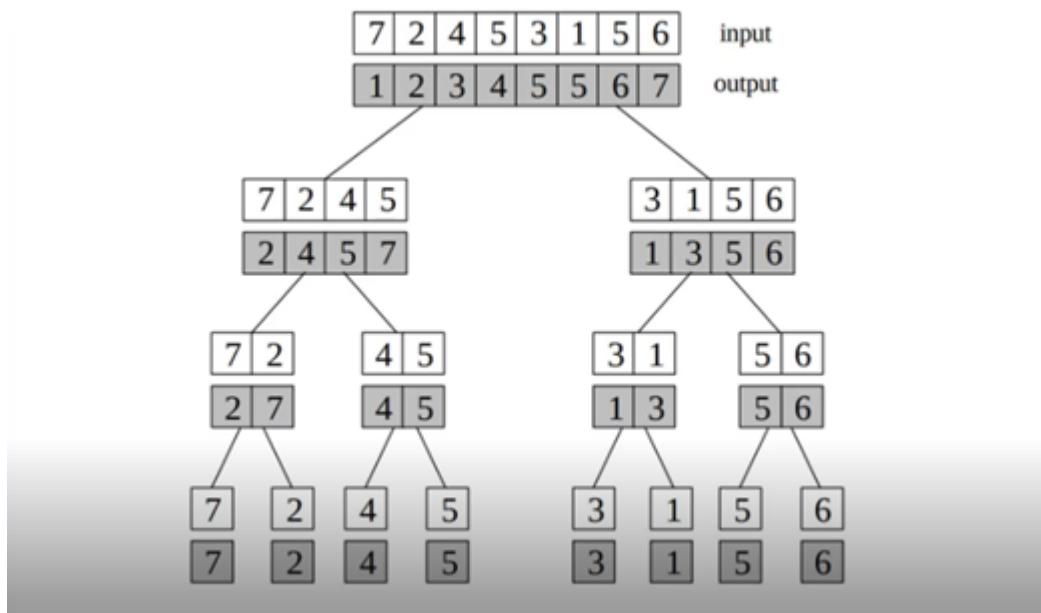
Algoritmi di Ordinamento

Mergesort

Il mergesort funziona in questo modo:

1. Riceve in input una sequenza
2. La divide a metà
3. Richiama se stessa sulle due metà
4. Fonde le due metà (*merge*) delle varie chiamate ricorsive in modo ordinato.

Ecco un disegno esplicativo del suo funzionamento:



Complessità:

Caso peggiore temporalmente $\Theta(n \log n)$

Caso migliore temporalmente $\Theta(n \log n)$

Pseudocode for Merge:

```
C = output [length = n]
A = 1st sorted array [n/2]
B = 2nd sorted array [n/2]
i = 1
j = 1
for k = 1 to n
    if A(i) < B(j)
        C(k) = A(i)
        i++
    else [B(j) < A(i)]
        C(k) = B(j)
        j++
end
(ignores end cases)
```

Il codice è quindi il seguente:

Funzione 3: Fonde le due metà.

// prende in input due sequenze o sotto sequenze, scorre le due sequenze, elemento per elemento, confrontando a mano a mano gli elementi tra di loro ,

l'elemento minore lo inserisce nella sequenza risultante, quello maggiore viene usato per il confronto successivo se una delle due sequenze termina gli elementi dell'altra sequenza(che sono già ordinati) vengono messi in coda theta(n) per ogni livello.

```
void fondi(vector<int>& v, unsigned int inizio, unsigned int centro, unsigned int fine)
```

```
{
```

```
    // dichiara e riempie i due vettori
```

```
    vector<int> vsinistra, vdestra;
```

```
    for (unsigned int i=inizio; i<=centro; ++i)
```

```
        vsinistra.push_back(v[i]);
```

```
for (unsigned int i=centro+1; i<=fine; ++i)  
    vdestra.push_back(v[i]);
```

// dichiara gli indici necessari

```
unsigned int indicesinistra = 0;  
unsigned int maxsin = vsinistra.size();  
unsigned int indicedestra = 0;  
unsigned int maxdes = vdestra.size();
```

// scorre i due vettori, elemento per elemento, confrontando a mano a mano gli elementi tra di loro

```
for (unsigned int i=inizio; i<=fine; ++i)
```

```
{
```

```
if (indicesinistra < maxsin && indicedestra < maxdes)
```

```
{
```

```
if (vsinistra[indicesinistra]<vdestra[indicedestra])
```

```
{
```

```
v[i] = vsinistra[indicesinistra];
```

```
indicesinistra++;
```

continue; // permette di evitare gli altri if e di passare direttamente alla prossima iterazione

```
}
```

```
else
```

```
{
```

```
v[i] = vdestra[indicedestra];  
indicedestra++; continue;  
}  
}
```

// se uno dei due vettori non confronta elementi, vuol dire che questi sono i più grandi e vengono aggiunti in coda

```
if (indicesinistra == maxsin && indicedestra < maxdes)  
{  
v[i] = vdestra[indicedestra];  
indicedestra++; continue;  
}
```

```
if (indicedestra == maxdes && indicesinistra < maxsin)
```

```
{  
v[i] = vsinistra[indicesinistra];  
indicesinistra++; continue;  
}
```

```
}
```

Funzione 2: Definisce le due metà (gli indici di inizio e di fine), le ordina chiamando fondi e grazie all'ultima chiamata di quest'ultima le mette assieme ordinandole.

```
void ms(vector<int>& v, unsigned int inizio, unsigned int fine)  
{
```

```

if (inizio < fine)

{
    unsigned int centro = (inizio+fine)/2;

    ms(v, inizio, centro);

    ms(v, centro+1, fine);

    fondi(v, inizio, centro, fine);
}

}

```

Funzione 1: Gestisce il caso di un vettore vuoto e chiama *ms* passando inizio e fine del vettore da ordinare.

```

void mergeSort(vector<int>& v)

{
    if (v.size() != 0)

        ms(v, 0, v.size()-1);

}

```

Costo della procedura merge:

Indichiamo con costo merge una funzione che va da N in R+ costomerge: $N \rightarrow R+$ t.c. costomerge(n) = il numero di operazioni necessarie a fondere due array, ciascuno lungo $n/2$, per ottenere un array lungo n .

$$\text{costomerge}(n) = \Theta(n)$$

Da dove deriva costomerge(n) = $\Theta(n)$?

Inizializzazione dei due cursori i e j (2 operazioni) per n volte ripeto

{ gestione della guardia del for (2 operazioni: una per l'inizializzazione o l'incremento di k, una per il confronto di k con n) confronto tra A(i) e B(j) (1 operazione) qualunque

sia l'esito del confronto, faccio altre 2 operazioni (una assegnazione di un valore a C(k) e un incremento di un cursore) }

costomerge(n) = $2 + 5n$ posso trascurare l'addendo 2 e la costante moltiplicativa 5, quindi ottengo un andamento lineare

Costo delle operazioni effettuate al livello j dell'albero della ricorsione

Al livello j dell'albero si affrontano 2^j sottoproblemi di tipo “merge”, ognuno di dimensione $n/2^j$ (“n fratto 2 alla j”) Il costo delle operazioni effettuate al livello j dell'albero è numero_problemi * costo_ciascun_problema = $2^j * \text{costomerge}(n/2^j) = \Theta(2^j * (n/2^j)) = \Theta(n)$ Ad ogni livello, faccio un numero di operazioni in $\Theta(n)$

Costo della procedura mergesort

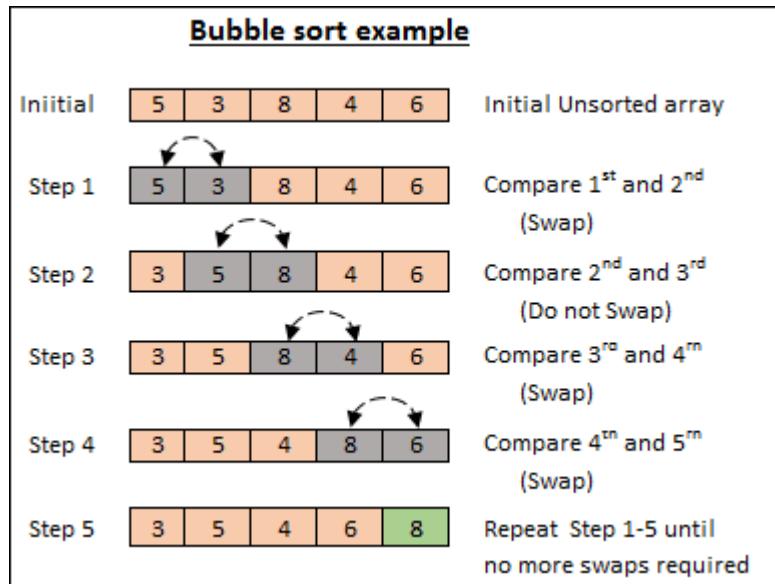
L'albero della ricorsione ha $\log_2 n + 1$ livelli Ad ogni livello dell'albero si eseguono $\Theta(n)$ operazioni. Trascuro il “+1” nel numero dei livelli: non ha impatto sull'analisi asintotica di complessità Quindi costo mergesort(n) = $\log_2 n * \Theta(n) = \Theta(n \log n)$

Bubble Sort

Il Bubble Sort funziona in questo modo:

1. Riceve in input una sequenza
2. Compara elementi consecutivi
3. Se non sono ordinati li scambia
4. Ripete finchè non avviene nessuno scambio

Ecco un disegno esplicativo del suo funzionamento:



Complessità:

Caso peggiore temporalmente $\Theta(n^2)$

Caso ottimo temporalmente $\Theta(n)$

Il codice è quindi il seguente:

```
void bubbleSort(vector<int>& v)
{
  unsigned int size = v.size();
  bool scambiati;
  for (unsigned int i=1; i<size; ++i)
  {
    scambiati = false;
    for (unsigned int j=0; j<size-i; ++j)
      if(v[j]>v[j+1])
      {
        scambia(v, j, j+1);
        scambiati = true;
      }
    if (!scambiati)
      break;
  }
}
```

```
scambiati = true;  
}  
  
if (!scambiati) return;  
  
}  
  
}
```

Insertion Sort

L'insertion sort funziona in questo modo:

1. Riceve in input una sequenza
2. Analizza elemento per elemento
3. Se è ordinato va avanti
4. Se non è ordinato lo confronta con quelli precedenti

Ecco un disegno esplicativo del suo funzionamento:

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

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

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

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

Complessità:

Caso peggiore temporalmente: $\Theta(n^2)$

Caso migliore temporalmente: $\Theta(n)$

Il codice è quindi il seguente:

```
void insertionSort(vector<int>& v)
{
    int current, prev;
    unsigned int size = v.size();
    for (unsigned int i=1; i<size; ++i)
    {
        current=i;
        prev=i-1;
        while(prev>=0 && v[current]<v[prev])
```

```
{  
    scambia(v, current, prev);  
    --current;  
    --prev;  
}  
}  
}
```

Selection Sort

Il selection sort funziona in questo modo:

1. Riceve in input una sequenza
2. Individua il minimo
3. Lo mette all'inizio
4. Passa al successivo restringendo il campo d'azione

Ecco un disegno esplicativo del suo funzionamento:

6	3	7	2	8	1*
1	3	7	2*	8	6
1	2	7	3*	8	6
1	2	3	7	8	6*
1	2	3	6	8	7*
1	2	3	6	7	8

Complessità:

Caso peggiore temporalmente: $\Theta(n^2)$

Caso ottimo temporalmente: $\Theta(n^2)$

Il codice è quindi il seguente:

```
void selectionSort(vector<int>& v)
{
    int current_min_index;
    unsigned int size = v.size();
    for (unsigned int i=0; i<size; ++i)
    {
        current_min_index = i;
        for (unsigned int j=i+1;j<size; ++j)
            if (v[current_min_index] > v[j])
                current_min_index = j;
        scambia(v, i, current_min_index);
    }
}
```

Quick Sort

Il quicksort è un algoritmo ricorsivo che fa due chiamate ricorsive e funziona nel seguente modo:

- Prende un elemento dell'array (il **pivot**)
- Partiziona gli elementi dell'array in modo tale che quelli a sinistra siano minori del pivot e quelli a destra siano maggiori o uguali al pivot
- Dopo queste operazioni il pivot è nella sua posizione finale
- Richiama quicksort sulle due parti dell'array ottenute

Ecco un disegno esplicativo del funzionamento:



Complessità:

L'efficienza del quicksort dipende da come scegliamo il pivot.

Caso peggiore temporalmente:

$\Theta(n^2)$ questo è il caso peggiore perchè come pivot viene scelto l'elemento **maggiore** o **minore** della sequenza e delle sottosequenze. Dato che ogni chiamata ha costo $\Theta(n)$ e che prendendo pivot maggiore o minore verrà ordinato un elemento per volta (n elementi) e la complessità sarà quindi data dal prodotto del costo di chiamata per n volte: $\Theta(n*n)=\Theta(n^2)$.

Caso migliore temporalmente: **$\Theta(n \log n)$** questo è il caso migliore perchè come pivot viene scelto l'elemento **mediano** della sequenza e delle sottosequenze.

Implementazione:

```
int partizionaInPlace(vector<int>& v, int inizio, int fine)

{
    int pivotIndex = inizio+rand()% (fine-inizio+1); // scelgo
    un indice a caso tra inizio e fine: sara' il mio pivot

    scambia(v, pivotIndex, inizio); // metto il pivot
    all'inizio della sequenza da riordinare

    int i = inizio+1;

    for (int j=inizio+1; j<=fine; ++j)

    {
        if (v[j] < v[inizio]) // confronto con il pivot che e'
        all'inizio

        {
            scambia(v, i, j);

            ++i;
        }
    }

    scambia(v, inizio, i-1);

    return i-1;
}

void qs(vector<int>& v, int inizio, int fine)

{
    if (inizio < fine)

    {
```

```

        int pivot_index=partizionaInPlace(v, inizio, fine);

        qs(v, inizio, pivot_index-1);

        qs(v, pivot_index+1, fine);

    }

}

void quickSortRandom(vector<int>& v)

{
    srand(time(NULL)); // srand ha un costo non trascurabile:
    poiché basta chiamarla una sola volta all'interno del
    programma per fissare il seme della

                // generazione pseudo-casuale, possiamo
    chiamarla in quickSortRandom prima di qs(v, 0, v.size()-1) e
    non chiamarla più!

    qs(v, 0, v.size()-1);

}

```

Strutture Dati

Cosa Sono:

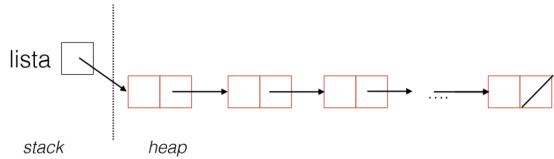
Le strutture dati sono una particolare organizzazione delle informazioni all'interno della memoria che semplificano la progettazione di algoritmi.

Una struttura dati efficiente rende altrettanto efficiente l'algoritmo che la manipola.

Elenco delle Strutture e Descrizione di Esse:

- Liste:

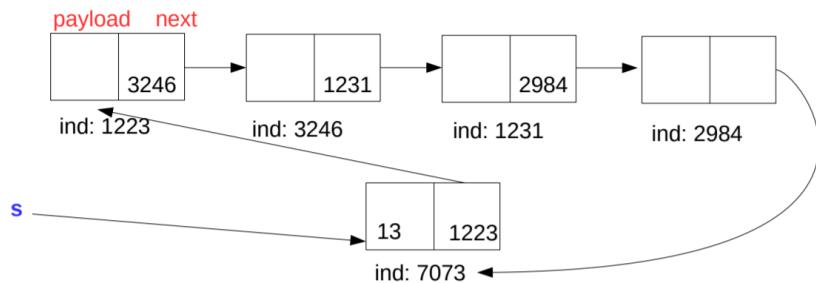
Semplici: sono struct collegate in sequenza da puntatori. Ogni struct contiene il dato



```
• struct cell {
    T info; //T tipo noto a priori
    cell *next;
};
```

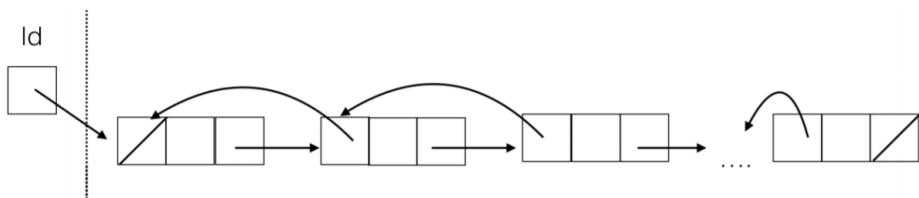
da memorizzare e un puntatore alla struct successiva.

Circolari: sono liste semplici con la differenza che il puntatore della struct che si



trova nell'ultima posizione della lista punta alla struct che si trova nella prima.

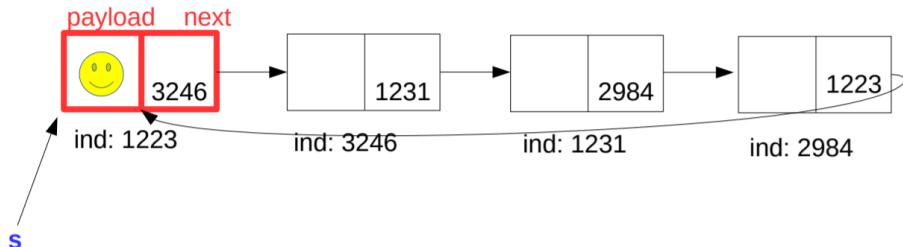
Doppiamente collegate: sono liste semplici nelle quali le struct oltre a contenere il puntatore alla struct successiva contengono il puntatore anche a quella precedente.



```
typedef int T;
struct cell {
    T head;
    cell *next;
    cell *prev;
};
typedef cell * lista_doppia;
lista_doppia ld;
```

Con sentinella: Aggiungendo una cella fittizia all'inizio della lista si può ovviare al problema di avere due casi distinti nel caso in cui la lista sia vuota o con degli elementi. Quindi la lista vuota sarà una lista con solo la sentinella e tutte le celle che vorremo aggiungere verranno inserite dopo la sentinella.

Avendo la sentinella come primo elemento non dovremo neanche preoccuparci di modificare il puntatore alla lista nel caso modificassimo l'ordine degli elementi o



eliminassimo il primo.

- Tabelle ad Accesso Diretto:

Sono dizionari basati sulla proprietà di accesso diretto alle celle di un array.

Sono costituite da un array in cui ogni cella contiene un dato (elemento) ed a ogni elemento è associata una chiave (di solito un numero intero che indica l'offset rispetto alla prima cella dell'array). Per accedere ad un determinato elemento è necessario sapere la chiave a cui è associato.

Lo svantaggio di questa struttura dati è che se volessimo adoperare solo le chiavi da 100 a 110 dovremmo ugualmente creare un array di size 110 e lasciare inutilizzate 100 celle sprecando molto spazio.

I fattore di carico (α) è il rapporto tra il numero degli elementi e il numero delle celle dell'array, più si avvicina a 1 più stiamo utilizzando bene lo spazio.

$$\alpha = \frac{\text{numero elementi}}{\text{numero celle array}}$$

- Tabelle di Hash:

Le Tabelle di Hash nascono per ovviare al problema dello spreco di spazio delle tabelle ad accesso diretto. Le chiavi possono non essere numeri e il loro ordine può anche essere casuale. Per accedere alla cella dell'array associata ad una chiave

esistono delle funzioni (dette di hash) che trasformano attraverso un algoritmo la chiave in un numero.

Ad esempio se la chiave è una stringa, una funzione di hash potrebbe essere la parte intera del rapporto tra la somma dei valori ascii delle lettere componenti la stringa e la size dell'array.

Una buona funzione di Hash dovrebbe distribuire gli elementi uniformemente all'interno dell'array. Ad esempio se hai 100 chiavi ed hai a disposizione 10 celle, una buona funzione di hash associa 10 chiavi ad ogni cella.

LA FUNZIONE DI HASH DEVE ESSERE CALCOLABILE IN TEMPO COSTANTE ALTRIMENTI SI PERDE IL VANTAGGIO DI USARE QUESTA STRUTTURA DATI.

Se volessimo aggiungere un elemento nella tabella in una posizione già occupata si crea una collisione. Per risolvere le collisioni si creano delle liste semplici (chiamate di collisione o bucket) in cui vengono memorizzati tutti gli elementi associati alla stessa cella dell'array e all'interno di quest'ultima si inserisce il puntatore alla lista.

Una funzione di Hash è **perfetta** quando non ci sono collisioni.

UNA FUNZIONE DI HASH PERFETTA DEVE ESSERE INIETTIVA E OVVIALEMENTE CALCOLABILE IN TEMPO COSTANTE.

- Alberi:

Gli alberi sono utili da utilizzare quando si hanno per le mani dati che presentano relazioni gerarchiche tra di loro. A differenza delle liste, ogni elemento può avere relazioni con più elementi.

Terminologia:

Un albero radicato è una coppia $T=(N, A)$ costituita da un insieme N di nodi (o vertici) e un insieme A (sottoinsieme di $N \times N$) di coppie di nodi detti archi.

Il fatto che un albero sia radicato significa che un vertice viene indicato come **radice** e differisce dagli altri perchè è **l'unico senza un padre**.

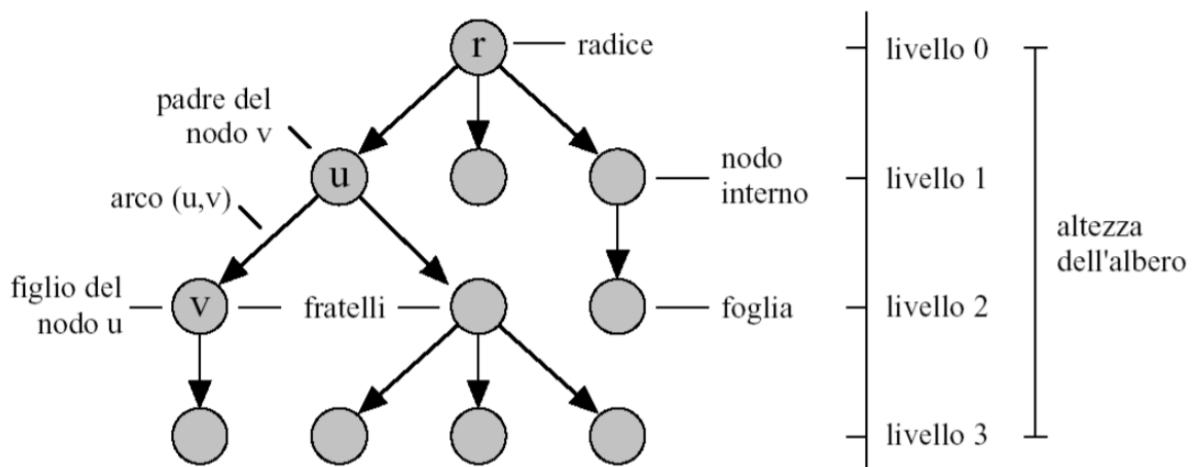
Il numero dei figli di un nodo si chiama **grado**; un nodo senza figli si chiama **foglia** e tutti gli altri si chiamano **nodi interni**; gli **antenati** sono tutti i nodi raggiungibili salendo di padre in padre e i **discendenti** sono i nodi raggiungibili scendendo di figlio in figlio.

Un albero è un particolare tipo di grafo: è un grafo connesso minimale (se si toglie un arco, non è più connesso) o, se vogliamo dare una caratterizzazione in termini della ciclicità, è un grafo aciclico massimale (se si aggiunge un arco, non è più aciclico).

La **profondità** di un nodo è il numero di archi che bisogna attraversare per raggiungerlo partendo dalla radice (che ha profondità 0).

I nodi con lo stesso genitore vengono detti **fratelli**; l'**altezza** di un albero è la massima profondità in cui si trova una foglia.

Un albero è un grafo connesso minimale.



Alberi speciali (d-ari):

Gli alberi speciali sono alberi che hanno un vincolo sul numero di figli di ogni nodo. Un albero d-ario è un albero i cui nodi hanno al più grado d. Un albero d-ario **completo** è un albero in cui ogni nodo ha esattamente grado d mentre un albero d-ario **quasi completo** ha ogni livello completo tranne eventualmente l'ultimo in cui tutti i nodi di grado d sono più a sinistra possibile.

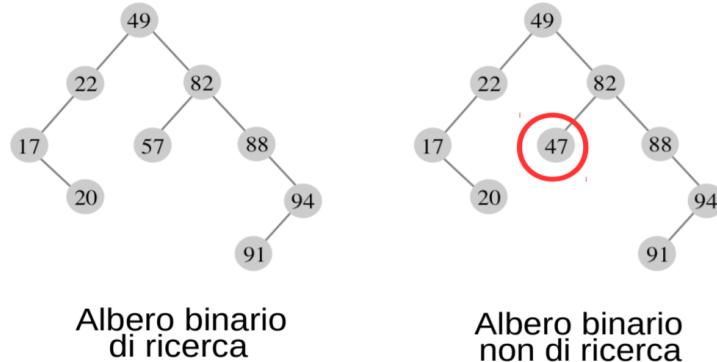
Un albero binario completo di altezza h ha $2^{h+1} - 1$ nodi.

Alberi binari di ricerca (BST):

Sono alberi binari e **non devono per forza essere completi o quasi completi**.

Ad ogni nodo è associato un elemento ed una chiave presa da un dominio totalmente ordinato, quindi ogni chiave deve essere confrontabile con le altre.

Le chiavi degli elementi nel sottoalbero sinistro di un vertice V sono \leq chiave(V) mentre gli elementi nel sottoalbero destro sono \geq chiave(V).



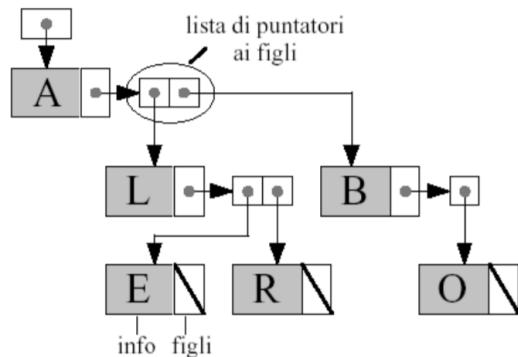
Alberi generici:

Gli alberi generici sono alberi che non hanno un vincolo sul numero di figli dei nodi.

Rappresentazioni di alberi:

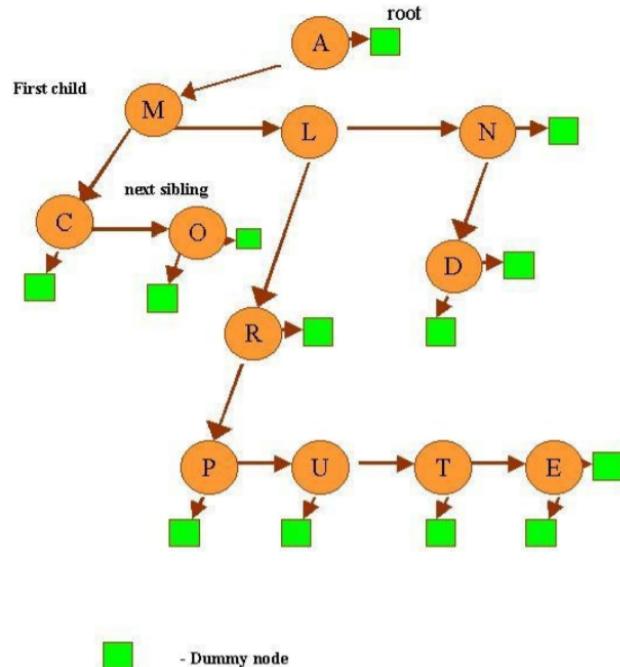
- Rappresentazioni collegate di alberi generici:

La struct dei nodi sarà composta dal campo info e da un puntatore a una lista che conterrà i puntatori a tutti i figli del nodo.



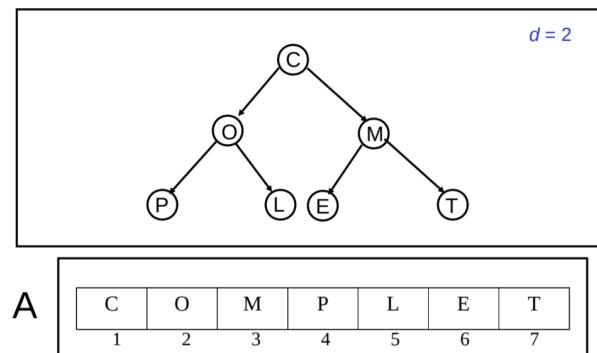
- Rappresentazioni collegate di alberi generici:

Nella rappresentazione primo figlio-fratello successivo di un albero con un numero arbitrario di figli la struct dei nodi avrà un puntatore ad un figlio ed un puntatore ad un suo fratello.



- Rappresentazione indicizzata di alberi d-ari quasi completi:

Inserisco i nodi dentro all'array nello stesso modo in cui li incontrerei facendo una visita in ampiezza.



In questo modo trovare i genitori è molto più facile:

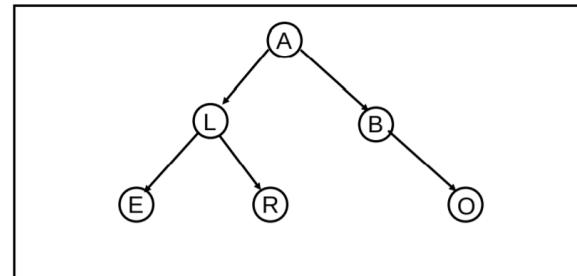
J-esimo figlio ($J \in \{1, \dots, d\}$) di i è in posizione: $d*(i-1)+J+1$.

- Rappresentazioni indicizzate di alberi generici:

Vettore dei padri

Per un albero con n nodi uso un array P con dimensione almeno n.

Ogni cella contiene il campo info e l'indice dell'array di suo padre.



Il padre di L sarà
nell'indice 3
dell'array

(L,3)	(B,3)	(A, -1)	(O,2)	(E,1)	(R,1)
1	2	3	4	5	6

DEFINIZIONI IMPORTANTI SUGLI ALBERI:

- Il J -esimo figlio ($J \in \{1, \dots, d\}$) di I è in posizione $d \times (I - 1) + J + 1$.
- Il padre di I (con $I \neq 1$ che è la radice) è in posizione $\text{floor}((I - 2)/d) + 1$.
- Un albero binario completo di altezza h ha: $2^{h+1} - 1$ nodi.
- Un albero binario quasi completo di altezza h ha: $2^h \leq n \leq 2^{h+1}$ nodi.
- L'altezza di un albero binario quasi completo con n nodi è: $\lfloor \log_2 n \rfloor$.

- Grafi:

Un grafo $G = (V, E)$ consiste in:

- un insieme V di vertici (o nodi) e un insieme E di coppie di vertici, detti archi o spigoli: ogni arco connette due vertici.

Con n si indica il numero dei vertici e con m il numero degli archi.

Dato un grafo $G = (V, E)$, se esiste un arco (x, y) in E , allora diciamo che:

- l'arco (x, y) è **incidente** sia in x che in y .
- x e y sono gli **estremi** dell'arco (x, y) .
- Se G è un grafo orientato, allora diciamo che l'arco (x, y) **esce** dal vertice x ed **entra** nel vertice y ; inoltre diciamo che y è adiacente a x , ma x non è adiacente a y .
- Se G è un grafo non orientato, allora diciamo che x ed y sono **adiacenti**.

Il **grado** di un vertice, indicato con d , è pari al numero di archi incidenti in esso.

In un grafo non orientato $G \sum_{v \in V} \delta(v) = 2m$

Sia G un grafo orientato: Il grado in ingresso di un vertice v è pari al numero di archi che entrano in v e si indica con δ in (v). Il grado in uscita di un vertice v è pari al numero di archi che escono da v e si indica con δ out (v).

$\delta(v) = \delta$ in (v) + δ out (v) è il grado del vertice v .

Un **cammino** in un grafo è una sequenza di vertici $[v_0, v_1, \dots, v_n]$ tali che ogni coppia di vertici consecutivi nella sequenza (v_i, v_{i+1}) è connessa da un arco. Un **cammino** è detto **semplice** se tutti i vertici $[v_0, v_1, \dots, v_n]$ ad eccezione al più del primo e dell'ultimo, sono distinti. Un cammino semplice in cui il primo e l'ultimo vertice sono coincidenti è detto **ciclo**.

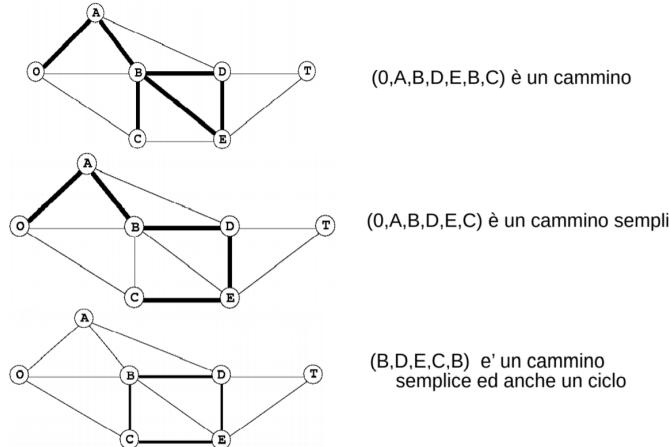
Grafi etichettati:

I grafi etichettati sono grafi in cui ai vertici sono associate **etichette** mentre agli archi possono essere associati dei valori numerici detti **pesi**.

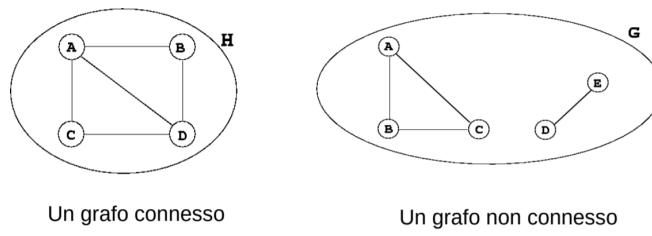
Un esempio di grafo etichettato può essere una mappa stradale in cui i vertici rappresentano i nomi delle città e il grado degli archi la distanza tra esse.

Nei grafi possono esistere vertici non connessi agli altri.

Cammini e cicli (grafo non orientato)

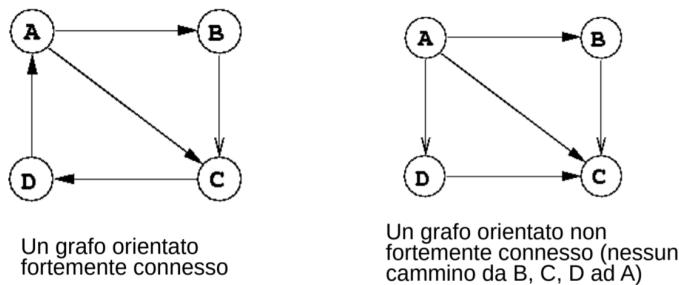


Un grafo non orientato $G=(V,E)$ è connesso se e solo se per ogni coppia (v,w) di



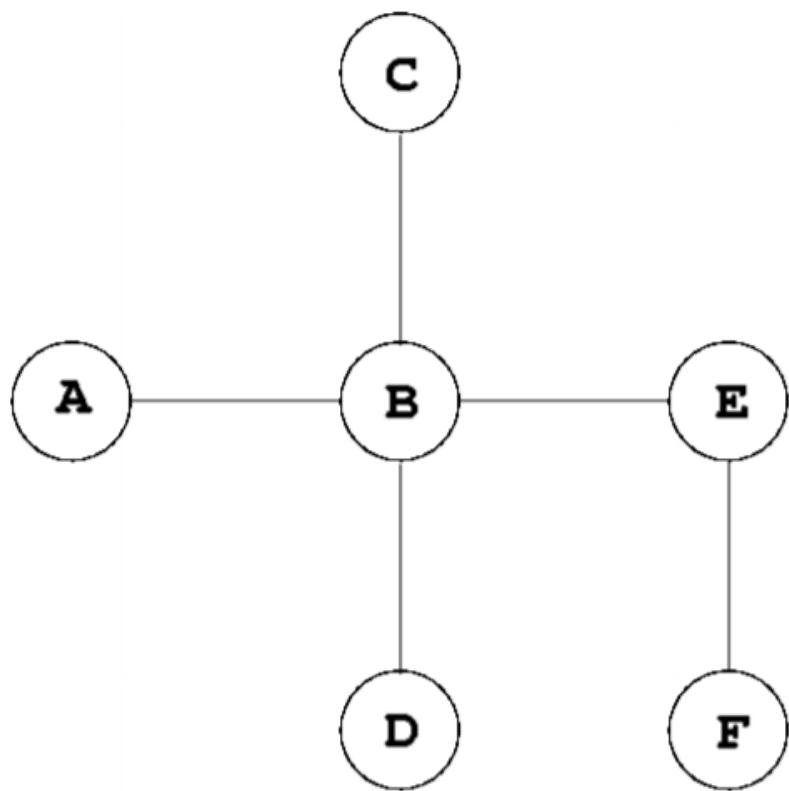
vertici di G esiste un cammino che li unisce.

Un grafo orientato G è fortemente connesso se e solo esiste un cammino da ogni vertice v di G ad ogni vertice v' di G con $v' \neq v$.



Alberi liberi:

Un albero libero è un grafo non orientato, connesso e senza cicli. A differenza di un albero radicato, un albero libero non ha radice.

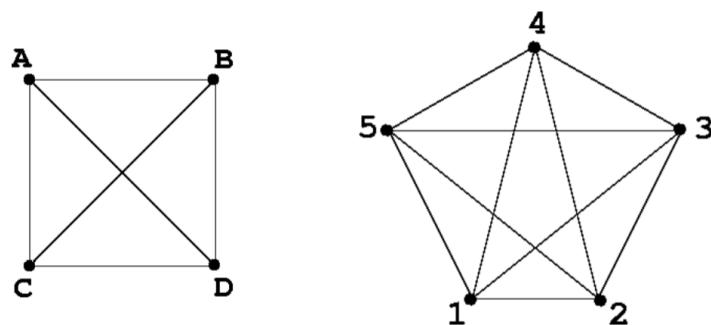


Proprietà:

- Se rimuoviamo un arco da un albero libero, il grafo risultante è non connesso.
- Sia n il numero di vertici di un albero libero H , allora H ha esattamente $n-1$ archi

Grafo completo:

Un grafo **non orientato** G , in cui ogni vertice è connesso da un arco ad ognuno degli altri $n-1$ vertici, è detto **grafo completo** e ha esattamente $n(n-1)/2$ archi (con $n =$



numero vertici)

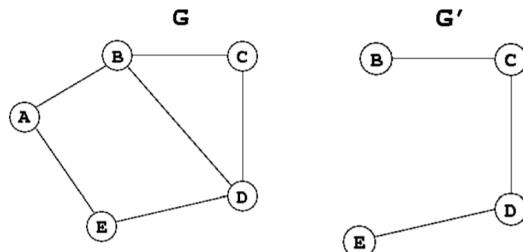
Un grafo orientato G in cui ogni vertice è connesso da un arco ad ognuno dei restanti $n-1$ vertici è detto completo e **ha esattamente $n(n-1)$ archi** (con n = numero vertici)

Sottografo:

Un sottografo di un grafo $G=(V,E)$ è un grafo $G'=(V',E')$ tale che $V' \subseteq V$ e $E' \subseteq E$.

Un sottografo G' è detto un **sottografo proprio** di G se G' non coincide con G .

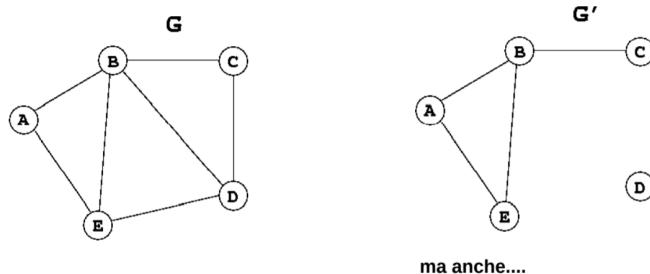
Esempio: un grafo G e un sottografo proprio G' di G



Sottografo di ricoprimento (spanning graph)

Un sottografo $G'=(V',E')$ di un grafo $G=(V,E)$ tale che $V'=V$ è detto sottografo di ricoprimento di G (grafo aventi gli stessi vertici)

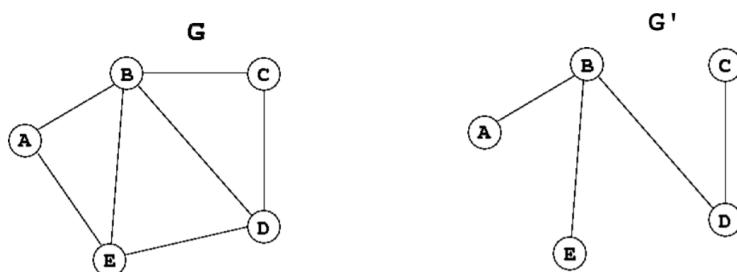
Esempio: G' un sottografo di ricoprimento del grafo G .



Albero di ricoprimento:

Un sottografo di ricoprimento $G'=(V,E')$ di un grafo $G=(V,E)$ tale che G' è un albero libero è detto albero di ricoprimento di G .

Esempio: G' un albero di ricoprimento del grafo G .



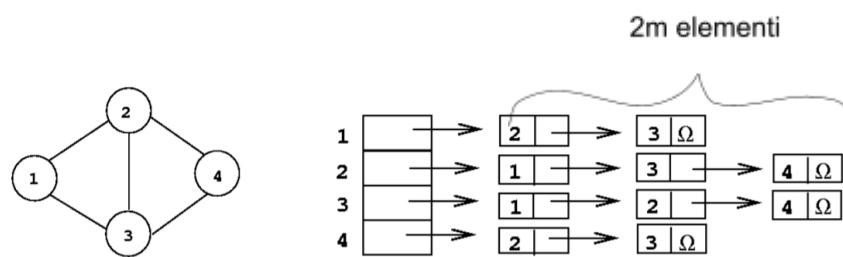
Strutture dati per grafi più diffuse:

- Liste di adiacenza:

Nelle liste di adiacenza ogni nodo è rappresentato da una cella di un array contenente un puntatore ad una lista che contiene tutti i nodi adiacenti ad esso.

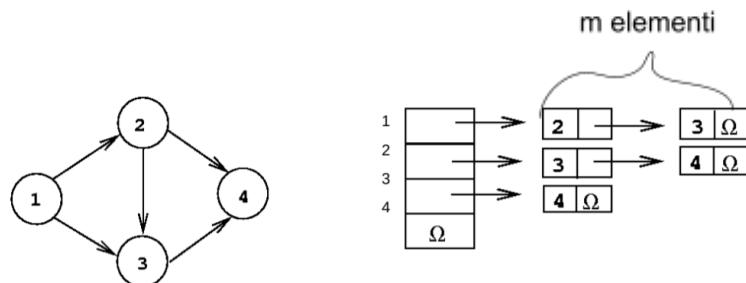
- In un grafo non orientato

In un grafo **non orientato** ogni arco è rappresentato **due volte**, una nella lista di adiacenza del nodo da cui esce e l'altra nella lista di adiacenza del nodo in cui entra.



Notare che in un grafo non orientato avente **m archi** ci saranno **2m elementi** nelle liste di adiacenza.

- In un grafo orientato:



In un grafo **orientato** ogni arco è rappresentato **solo una volta** perché rappresento solo gli archi uscenti dal nodo.

Notare invece che in un grafo orientato avente **m archi** ci saranno **m elementi** nelle liste di adiacenza

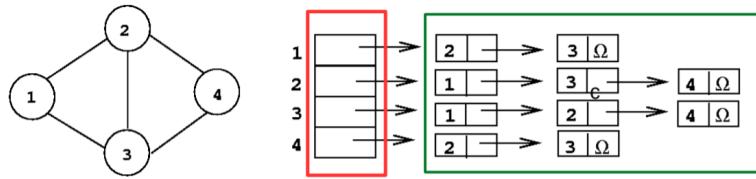
Liste di adiacenza occupazione spaziale

- Ogni arco del grafo G è descritto da

1. Un elemento nelle liste di adiacenza, se G è un grafo orientato.
 2. Due elementi (corrispondenti ai suoi due vertici estremi) se G è non orientato.
- Sia n il numero dei vertici ed m il numero di archi, **Le liste di adiacenza contengono un totale di:**
 - **m elementi se G è orientato.**
 - **$2m$ elementi se G non è orientato.**
- Se ogni lista di adiacenza è implementata come una lista singolarmente collegata, **la struttura dati richiede:**
 - **$n+m$ “blocchi” per un grafo orientato.**
 - **$n+2m$ “blocchi” per un grafo non orientato.**

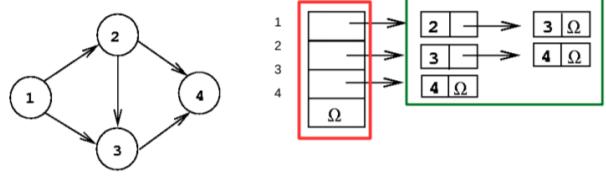
Grafi non orientati

Esempio:



Grafi orientati

Esempio:



- **n elementi per rappresentare i nodi**
- **$2m$ elementi in totale nelle liste di adiacenza**

- **n elementi per rappresentare i nodi**
- **m elementi nelle liste di adiacenza**

Spazio totale nell'ordine di $n + 2m$

Spazio totale nell'ordine di $n + m$

Algoritmo di Dijkstra (grafo):

E' un algoritmo molto complesso che serve a calcolare (in un grafo) il cammino con costo minimo da uno a tutti gli altri vertici.

Code con priorità

Sono code in cui gli elementi che entrano non escono come nelle code semplici (il primo che entra è anche il primo che esce) ma escono in base alla **priorità**, che è rappresentata dalle **chiavi** associate agli elementi.

Quindi per esempio se nella nostra coda abbiamo un elemento con chiave 3 ed inseriamo un elemento con chiave 4 essi si disporranno in modo tale da permettere all'elemento con chiave maggiore (ricordandosi che la chiave è anche la priorità) di uscire per primo.

Nel caso ci fossero nella stessa coda elementi con la stessa priorità, verranno disposti nell'ordine in cui sono entrati.

Code con priorità con liste semplici:

In una lista semplice possiamo fare in modo che gli elementi sono ordinati in modo decrescente di priorità, tale che il primo elemento della lista sia sempre quello con priorità massima.

Questa organizzazione semplifica molto le operazioni *deleteMax* e *findMax* che ritorneranno il primo elemento della lista.

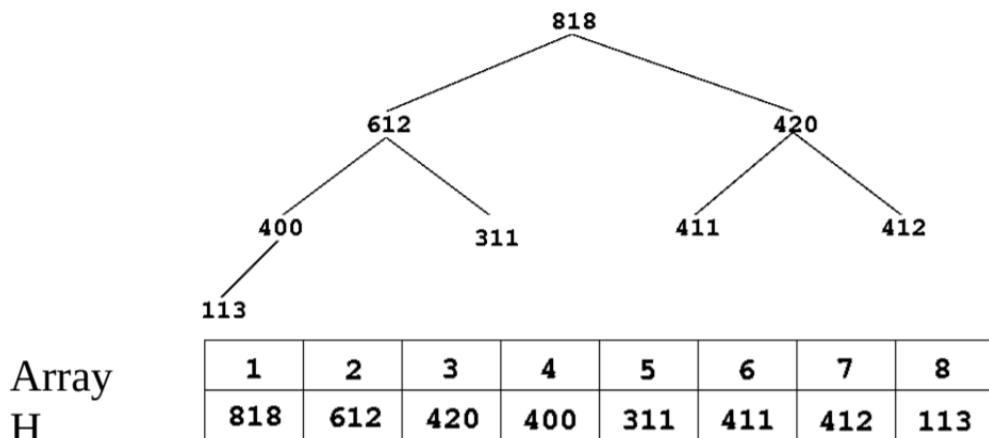
Heap Binario

L'Heap binario è un albero binario quasi completo in cui ogni nodo contiene un'etichetta e la chiave associata ad essa.

La particolarità di questo albero è che la chiave di un nodo è sempre maggiore di quella di entrambi i figli.

Uno Heap Binario può essere rappresentato come un array in cui i figli di un nodo alla posizione x sono memorizzati nelle posizioni $2x$ e $2x+1$.

I nodi si dispongono nell'array nello stesso ordine in cui verrebbero visitati da una visita in ampiezza. La cella 0 dell'array rimane vuota.



Come si può vedere nell'immagine qua sopra i figli del nodo con chiave 612 sono in posizione $H[2x2=4]$ e $H[2x2+1= 5]$.

Albero RossoNero

Sono un particolare tipo di Alberi Binari di Ricerca.

1. Ogni nodo ha colore rosso o nero.
2. La radice inizialmente è nera.
3. Ogni foglia è nera e contiene elemento null;
4. Entrambi i figli di ciascun nodo rosso sono neri;
5. Ogni cammino da un nodo a una foglia nel suo sottoalbero contiene lo stesso numero di nodi neri.
6. il cammino più lungo dal nodo root a una foglia è al massimo lungo il doppio del cammino più breve.
7. Sono quindi fortemente bilanciati.
8. Poiché le operazioni di ricerca di un valore, inserimento e cancellazione richiedono un tempo di esecuzione nel caso peggiore proporzionale all'altezza dell'albero, esse hanno complessità $\Theta(h) = \Theta(2\log_2(n + 1)) = \Theta(\log n)$

Quindi essi sono più complessi dei BST classici ma garantiscono una complessità delle operazioni di ricerca, inserimento e cancellazione che dipende dall'altezza dell'albero, che dipende a sua volta dal numero di nodi n secondo la formula:

$$2\log_2(n + 1).$$

Albero AVL

L'albero AVL è un albero binario di ricerca bilanciato.

Grazie a questa restrizione, l'altezza massima dell'albero è logaritmica nel numero dei nodi.

È per questo che questa struttura di dati permette di compiere l'inserimento, la ricerca e l'eliminazione di un elemento in $O(\log n)$. Inoltre se si considerano i costi ammortizzati di una serie di inserzioni, questi sono $O(1)$.

Questi vantaggi si pagano con un grande complessità.

Tipi di Dato (TDD)

Cosa Sono:

Un **tipo** è un insieme di valori. (es. bool {vero,falso})

`int`, `bool`, `char` [...] sono esempi di **tipo semplice** mentre un insieme composto da nome, indirizzo, codice fiscale [...] è un **tipo aggregato o composto**.

Un **dato** è un elemento il cui valore è preso da un tipo. (il dato “1” preso dal tipo “`int`”)

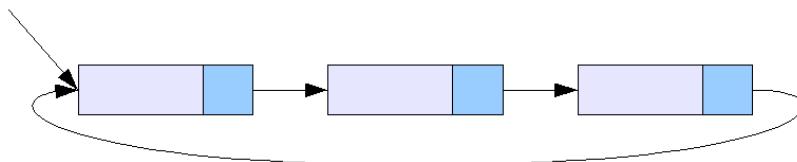
Un **tipo di dato** (concreto) **TDD**, è un tipo con una collezione di operazioni per manipolarne gli elementi (o membri). Più formalmente è un’algebra, solitamente eterogenea, su una segnatura. Una **segnatura** è una coppia **(S,O)**, con **S** insieme con elementi definiti sort, e **O** famiglia di operatori su queste sort. Es. `int`, `stack` e `push`, `pop` per uno stack di interi

Un **tipo di dato astratto** è una classe di algebre, solitamente eterogenee, sulla stessa segnatura.

I TDD specificano l’interfaccia (sintassi) e il significato che le operazioni nell’interfaccia devono assumere (semantica), consentendo di realizzare il principio dell’incapsulamento o **information hiding**.

Elenco dei TDD e descrizione di essi

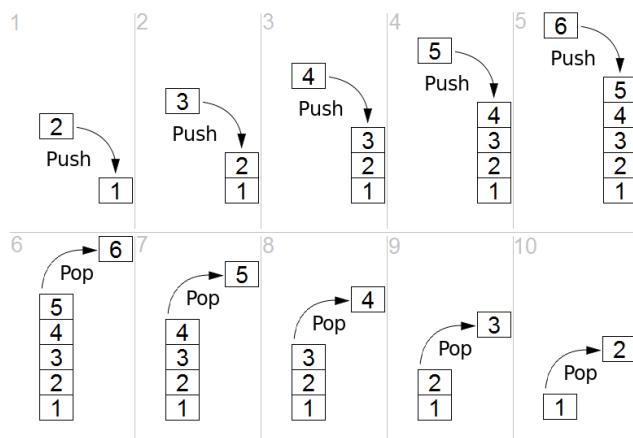
- **Liste**



Algebra eterogenea in cui le sort sono `List`, `N`, `Bool`, `Elem` (dove `Elem` è una sort non ulteriormente specificata) e le operazioni, con loro proprietà, sono:

- `emptyList` → `List`
- `set N x Elem x List` → `List`
- `add N x Elem x List` → `List`
- `addBack Elem x List` → `List`
- `addFront Elem x List` → `List`
- `removePos N x List` → `List`
- `get: N x List` → `Elem`
- `isEmpty List` → `Bool`
- `size List` → `N`

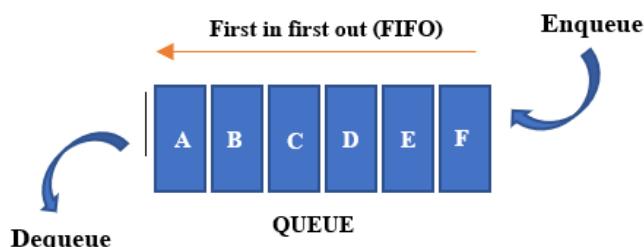
- **Pile (Stack)**



E' una sequenza di elementi di un certo tipo, in cui è possibile aggiungere o togliere elementi soltanto ad un estremo (la testa della sequenza). Una pila può essere quindi vista come un caso speciale di **lista** in cui l'ultimo elemento è anche il primo ad essere rimosso e non è possibile accedere ad alcun elemento che non sia quello in testa.

- `isEmpty() → result`
- `push(elem e)`
- `pop() → elem`
- `top() → elem`

- **Coda (Queue)**



E' una sequenza di elementi di un certo tipo, in cui è possibile aggiungere elementi ad un estremo e toglierne dall'estremo opposto. Una coda può essere quindi vista come una particolare lista in cui è possibile aggiungere elementi ad un estremo (il fondo) e togliere elementi dall'altro estremo (in testa).

- `isEmpty() → result restituisce true se S è vuota`
- `enqueue(elem e) aggiunge e come ultimo elemento di S`

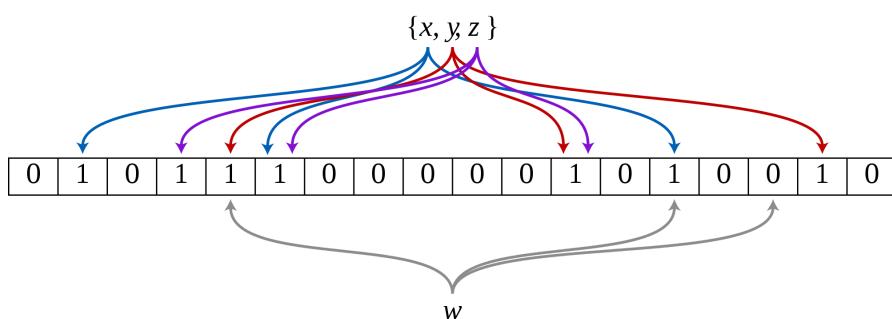
- `dequeue()` → elem toglie da S il primo elemento e lo restituisce
- `first()` → elem restituisce il primo elemento di S

- **Insiemi (Set)**

Sono un aggregato di elementi a valori omogenei (provenienti dallo stesso dominio) e distinti (**senza ripetizioni**).

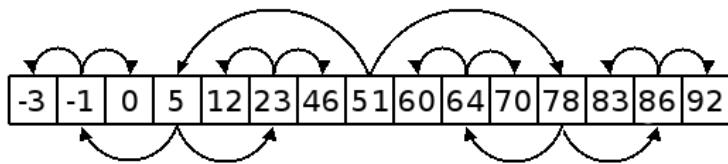
- `emptySet` → Set
- `insertElem` Elem x Set → Set
- `deleteElem` Elem x Set → Set
- `setUnion` Set x Set → Set
- `setIntersection` Set x Set → Set
- `setDifference` Set x Set → Set
- `isEmpty` Set → Bool
- `isSubset` Set x Set → Bool
- `size` Set → N
- `member` Elem xSet → Bool

- **BitVector**



Le operazioni degli operatori bit a bit servono a controllare, modificare o spostare i bit di un valore binario. Danno il risultato in base ai bit del/i valore/i.

- **Array Ordinato**



L'ordinamento rende meno efficienti alcune operazioni, ma più efficiente la ricerca, l'unione e l'intersezione.

- **Dizionario**



- insert (elem e, chiave k) aggiunge a S una nuova coppia (e,k)
- delete(chiave k) cancella da S la coppia con chiave k.
- search (chiave k) -> elem se la chiave k è presente in S restituisce l'elemento ad esso associato, altrimenti null.

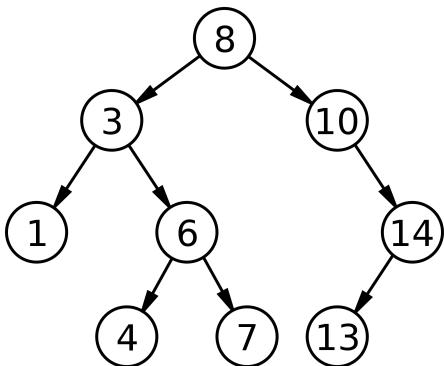
- **Tabelle Hash (strutture dati)**

Sono dizionari basati sulla proprietà di accesso diretto alle celle di un array

- Chiavi prese da un universo totalmente ordinato U (possono non essere numeri)
- Funzione hash: $h: U \rightarrow [0, m-1]$ (funzione che trasforma chiavi in indici)
- Elemento con chiave k in posizione $v[h(k)]$

ideali come supporto per TDD dizionario e TDD insieme.

- **Albero (Tree)**



Un albero radicato è una coppia $T = (N, A)$ costituita da un insieme N di nodi (o "vertici") e un insieme A (sottoinsieme di $N \times N$) di coppie di nodi dette archi.

//Approccio del libro di testo: definire le operazioni direttamente sull'insieme dei nodi:

- `numnodi()` → intero restituisce il numero di nodi presenti nell'albero
- `grado(nodo v)` → intero restituisce il numero di figli del nodo v
- `padre(nodo v)` → nodo restituisce il padre del nodo v nell'albero null se v è la radice
- `figli(nodo v)` → $[nodo, nodo, nodo\dots]$ restituisce uno dopo l'altro, i figli del nodo v
- `aggiungiNodo(nodo u, nodo v)` → nodo inserisce un nuovo nodo v come figlio di u nell'albero e lo restituisce se v è il primo nodo ad essere inserito nell'albero, diventa a radice
- `aggiungiSottoalbero(Albero a, nodo u)` inserisce nell'albero il sottoalbero a in modo che la radice di a diventi figlia di u
- `rimuoviSottoalbero(nodo v)` → Albero stacca e restituisce l'intero sottoalbero radicato in v . l'operazione cancella dall'albero il nodo v e tutti i suoi discendenti.

//versione "nostra", con etichette come argomenti:

- `emptyTree: -> Tree` /* `emptyTree` è la costante corrispondente all'albero vuoto */
- `emptyLabel: -> Label` /* `emptyLabel` è la costante corrispondente all'etichetta vuota */
- `isEmpty: Tree -> Bool` /* `isEmpty(t)` restituisce true se l'albero t è vuoto, false altrimenti */
- `addElem: Label x Label x Tree -> Tree` /* `addElem(labelOfNodeInTree, labelOfNodeToAdd, t)` aggiunge il nodo etichettato con

- labelOfNodeToAdd come figlio del nodo etichettato con labelOfNodeInTree. Caso particolare: aggiunta della radice, che si ottiene con labelOfNodeInTree uguale a emptyLabel (= nessun padre) e ha successo solo se l'albero t è vuoto. Se non esiste un nodo nell'albero etichettato con labelOfNodeInTree oppure se nell'albero esiste già un nodo etichettato con labelOfNodeToAdd non aggiunge nulla */
- deleteElem: Label x Tree -> Tree /* deleteElem(l, t) rimuove dall'albero il nodo etichettato con la Label l e collega al padre di tale nodo tutti i suoi figli. Fallisce se si tenta di cancellare la radice e questa ha dei figli (non si saprebbe a che padre attaccarli) oppure se non esiste un nodo nell'albero etichettato con l */
 - father: Label x Tree -> Label /* father(l, t) restituisce l'etichetta del padre del nodo etichettato l se il nodo esiste nell'albero (o sottoalbero) t e se il padre esiste. Restituisce la costante emptyLabel altrimenti */
 - children: Label x Tree -> List /* children(l, t) restituisce una lista di Label contenente le etichette di tutti i figli del nodo etichettato con l nell'albero t */
 - degree: Label x Tree -> Int /* degree(l, t) restituisce il numero dei figli del nodo etichettato con l se il nodo etichettato con l esiste; restituisce qualche valore concordato (ad es. -1) altrimenti */
 - ancestors: Label x Tree -> List /* ancestors(l, t) restituisce una lista di Label contenente le etichette di tutti gli antenati del nodo l ESCLUSA l'etichetta del nodo stesso */
 - leastCommonAncestor: Label x Label x Tree -> Label /* leastCommonAncestor(label1, label2, t) restituisce l'etichetta del minimo antenato comune ai nodi etichettati con label1 e label2 nell'albero t */
 - member: Label x Tree -> Bool /* member(l, t) restituisce true se il nodo etichettato con la label l appartiene all'albero t e false altrimenti */
 - numNodes: Tree -> Int /* numNodes(t) restituisce il numero di nodi in t */

Visita in ampiezza(BFS):

Pseudocodice:

visitaBFS(nodo r)

Coda C;

C.enqueue (r);

```

while (not c.isEmpty()) do

    u ← enqueue();

    if(u!=nullptr) then

        visita il nodo u;

        while(u ha ancora un figlio s)

            C.enqueue(s);

```

Visita in profondità(DFS) ricorsiva:

La visita opera in questo modo:

Parte dalla radice e procede visitando i nodi di figlio in figlio chiamando se stessa ricorsivamente. Una volta arrivata ad una foglia retrocede fino al primo antenato che ha ancora dei figli non visitati e ripete lo stesso procedimento fino ad arrivare di nuovo ad un'altra foglia.

```

visitaDFSric(nodo r)

if (r == nullptr) then return;

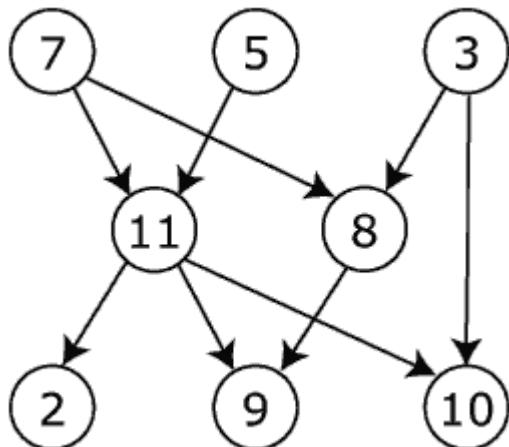
visito r;

while (r ha ancora un figlio s)

    visitaDFSric(s);

```

- **Grafo (Graph)**



Un grafo $G=(V,E)$ consiste in:

- un insieme V di vertici (o nodi)
- un insieme E di coppie di vertici, detti archi o spigoli: ogni arco connette due vertici.

// non orientato, con vertici etichettati in Label e archi etichettati in Weight

- emptyGraph: $\text{emptyGraph} \rightarrow \text{Graph}$ /* emptyGraph è la costante corrispondente al grafo vuoto */
- isEmpty: $\text{Graph} \rightarrow \text{Bool}$ /* isEmpty(g) restituisce true se g è il grafo vuoto e false altrimenti */
- addVertex: $\text{Label} \times \text{Graph} \rightarrow \text{Graph}$ /* addVertex(l, g) aggiunge un nuovo vertice etichettato l al grafo g e restituisce il grafo risultante da questo aggiornamento. Se l è già presente restituisce g senza modificarlo */
- addEdge: $\text{Label} \times \text{Label} \times \text{Weight} \times \text{Graph} \rightarrow \text{Graph}$ /* addEdge(source, dest, w, g) aggiunge nuovo arco tra i vertici etichettati con source e dest, rispettivamente, e assegna al nuovo arco peso w. Se non sono presenti tutti e due gli estremi del nuovo arco o se l'arco tra i due è già presente, addEdge restituisce g senza modificarlo */
- removeVertex: $\text{Label} \times \text{Graph} \rightarrow \text{Graph}$ /* removeVertex(l, g) cancella il vertice etichettato con l da g, se appartiene a g, e tutti gli archi incidenti in esso */
- removeEdge: $\text{Label} \times \text{Label} \times \text{Graph} \rightarrow \text{Graph}$ /* removeEdge(source, dest, g) cancella l'arco (source, dest) dal grafo g, se (source,dest) è un arco di g */
- degree: $\text{Label} \times \text{Graph} \rightarrow \text{Int}$ /* degree(l, g) calcola e restituisce il grado del vertice etichettato con l. Fallisce (restituendo ad esempio -1) se non esiste in g un vertice etichettato con l */

- `areAdjacent: Label x Label x Graph -> Bool` /* `areAdjacent(source, dest, g)` restituisce true se source e dest sono adiacenti (ovvero se esiste un arco (source, dest) che li collega), false altrimenti */
- `incidentEdges: Label x Graph -> List` /* `incidentEdges(l, g)` restituisce la lista di archi incidenti nel vertice etichettato con l; ogni arco è rappresentato dalla coppia di etichette degli estremi dell'arco stesso */
- `numVertices: Graph -> Int` /* `numVertices(g)` restituisce il numero di vertici del grafo */
- `numEdges: Graph -> Int` /* `numEdges(g)` restituisce il numero di archi del grafo */

Visita in ampiezza (BFS):

La BFS utilizza come strutture d'appoggio una coda e un albero di ricoprimento.

Pseudocodice:

visitaBFS(vertice s) → albero

```

rendi tutti i vertici non marcati;

T ← albero formato dal solo nodo s;

coda F; // creo una coda vuota

marca il vertice s;

F.enqueue(s);

while(not F.isEmpty()) do

    u ← F.dequeue();

    for each (arco(u, v) in G) do

        if (v non è ancora marcato) then

            F.enqueue(v);

            marca il vertice v;

            rendi u padre di v in T;

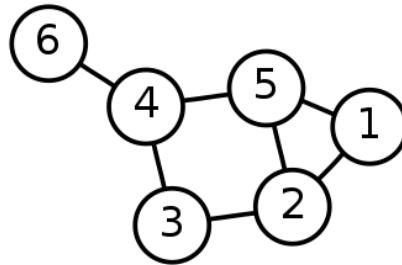
return T;

```

il procedimento è il seguente.

Si parte dal nodo su cui viene chiamata, si pone il booleano visitato = true, lo si inserisce nella coda e lo si pone come radice dell'albero di ricoprimento. Dopodichè si procede con la dequeue fino a quando la coda non è vuota e per ogni vertice che faccio uscire aggiungo i suoi vertici adiacenti, che hanno il booleano visitato = false, alla coda e li pongo come suoi figli nell'albero.

Ripeto il procedimento fino a quando tutti i vertici saranno stati visitati.



prendendo come esempio il grafo qua sopra:

Chiamo la visita sul vertice 6.

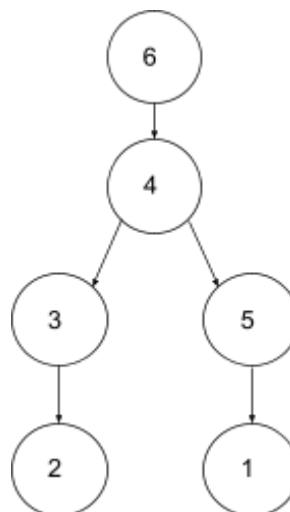
Inserisco il vertice 6 nella coda, pongo il suo booleano = true e lo metto come radice dell'albero. Procedo con la dequeue, faccio uscire il 6, inserisco nella coda il 4, lo pongo come visitato e lo metto come figlio di 6 nell'albero.

Faccio di nuovo la dequeue, faccio uscire il 4, inserisco nella coda il 3 e il 5, lo pongo come visitati e li metto come figli di 4 nell'albero. Continuo fino a quando non ho visitato tutti i vertici del grafo.

Ordine in cui entrano nella coda:

1	2	5	3	4	6
---	---	---	---	---	---

L'albero è il seguente:



Visita in Profondità (DFS):

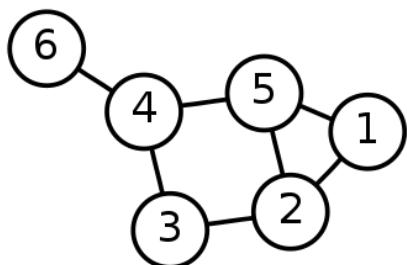
La DFS (depth-first search) è un algoritmo che consente di visitare un albero, o più in generale un grafo, in profondità.

La DFS svolta in modo ricorsivo non necessita di una coda come struttura d'appoggio ma richiede in ogni caso la costruzione di un albero di ricoprimento.

Infatti a grandi linee l'algoritmo procede in questo modo:

```
visitaDFSricorsiva(vertice v, albero T){  
    -marca il vertice v come visitato  
    -visita il vertice v  
    -per ogni arco w adiacente a v  
        --se w non è marcato  
            ---aggiunge l'arco (v, w) nell'albero T  
            ---richiama visitaDFSricorsiva su w  
}
```

Facciamo dunque un esempio del suo funzionamento; prendiamo come esempio questo grafo:



- Partiamo dal nodo 6; Esso viene marcato come visitato e viene stampato; vengono poi inseriti nell'albero (come figli di 6) i nodi adiacenti ad esso **non visitati**, quindi in questo caso solo il nodo 4, e viene richiamata la funzione su 4.
- Il nodo 4 viene marcato come visitato e viene stampato;

A questo punto bisogna vedere quale tra i nodi 5 e 3 viene per primo nella lista di adiacenza di 4, poniamo il caso che il primo sia 5.

- Il nodo 5 viene marcato come visitato e viene stampato;

A questo punto bisogna vedere quale tra i nodi 1 e 2 viene per primo nella lista di adiacenza di 5, poniamo il caso che il primo sia 2.

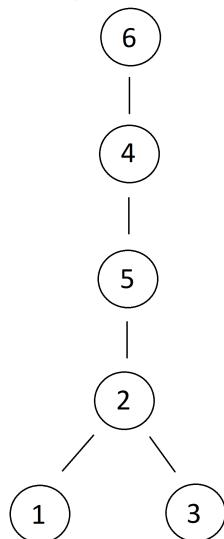
- d) Il nodo 2 viene marcato come visitato e viene stampato;

A questo punto bisogna vedere quale tra i nodi 1 e 3 viene per primo nella lista di adiacenza di 2, poniamo il caso che il primo sia 1.

-Il nodo 1 viene marcato come visitato e viene stampato e non ha nodi adiacenti **non visitati**.

-Il nodo 3 viene marcato come visitato e viene stampato e non ha nodi adiacenti **non visitati**.

L'albero di ricoprimento risultante è il seguente:



- Coda Prioritaria (Priority Queue)

insieme finito di coppie (chiave, valore) tale che chiave $\hat{}$ Key e valore $\hat{}$ Elem; sull'insieme Key è definita una relazione di ordine totale. Le operazioni del TDD PriorityQueue sono:

- emptyPriorityQueue: \rightarrow PriorityQueue /* emptyPriorityQueue è la coda di priorità vuota */
- findMax: PriorityQueue \rightarrow Elem (o in alternativa findMin) /* findMax(pq) = e se e è l'elemento in pq con la chiave massima (o minima per findMin) */

- insertElem: Elem x Key x PriorityQueue -> PriorityQueue /*
insertElem(e, k, pq) restituisce la coda di priorità pq' ottenuta inserendo
l'elemento e con chiave k in pq */
- deleteMax: PriorityQueue -> PriorityQueue (o in alternativa deleteMin)
/* deleteMax(pq) restituisce la coda di priorità pq' ottenuta cancellando
da pq l'elemento con chiave massima (minima) */

- Heap Binario (Binary Heap)

è un albero binario quasi completo tale che:

- ogni nodo v contiene un elemento elem(v) ed una chiave chiave(v)
presa da un dominio totalmente ordinato
- il valore della chiave in un nodo v è maggiore dei valori delle chiavi
associate ai suoi due figli

Una definizione analoga può essere data per uno heap in cui il valore della
chiave in un nodo v è minore dei valori delle chiavi associate ai suoi due figli

- findMax: PriorityQueue -> Elem (o in alternativa findMin) /* Restituisce
l'elemento nella radice dell'albero Nell'array si trova in posizione 1 –
Complessità temporale $\Theta(1)$ */
- insertElem: Elem x Key x PriorityQueue -> PriorityQueue Si inserisce
(k, e) come ultimo elemento dello heap (come ultimo elemento
dell'array H). La proprietà dell'ordinamento a heap viene ripristinata
spingendo il nodo v che contiene (k, e) verso l'alto tramite ripetuti
scambi di nodi
- muoviAlto (v,pq) while (v ≠ radice(T) and chiave(v) > chiave(padre(v))
do scambia v e padre(v) in T
- deleteMax: PriorityQueue -> PriorityQueue (o in alternativa deleteMin)
sostituiamo la radice dell'albero con l'ultimo elemento (l'elemento in
posizione n dell'array H che codifica lo heap – if v ha figli, then
muoviBasso(v, pq)
- muoviBasso(v, pq) while v ha almeno un figlio con chiave > chiave(v)
do sia u il figlio di v di chiave massima scambia v e u in T

Complessità

Big-O, Theta e Omega

Big-O

Siano f e g due funzioni dai numeri naturali ai numeri reali ≥ 0 ($f, g: \mathbb{N} \rightarrow \mathbb{R}_+$).

$f(n) = O(g(n))$ se due costanti $c > 0$ e $n_0 \geq 0$ tali che $f(n) \leq c g(n)$ per ogni $n \geq n_0$

(Big-O identifica una sovrastima)

Theta

Siano f e g due funzioni dai numeri naturali ai numeri reali ≥ 0 ($f, g: \mathbb{N} \rightarrow \mathbb{R}_+$).

$f(n) = \Theta(g(n))$ se tre costanti $c_1, c_2 > 0$ e $n_0 \geq 0$ tali che $c_1 g(n) \leq f(n) \leq c_2 g(n)$ per ogni $n \geq n_0$

(Theta identifica una stima esatta)

Omega

Siano f e g due funzioni dai numeri naturali ai numeri reali ≥ 0 ($f, g: \mathbb{N} \rightarrow \mathbb{R}_+$).

$f(n) = \Omega(g(n))$ se due costanti $c > 0$ e $n_0 \geq 0$ tali che $f(n) \geq c g(n)$ per ogni $n \geq n_0$.

(Omega identifica una sottostima)

Se una funzione è in Theta di un'altra funzione allora è anche in Omega e O.

Sia $f(n) = 3 \log n + 23 + 7n^3 + 12n \log_5 n + 6n$

- $f(n) = O(n^3)$? Sì
- $f(n) = \Omega(n^3)$? Sì
- $f(n) = \Theta(n^3)$? Sì

Grafico delle Complessità

Big-O Complexity Chart

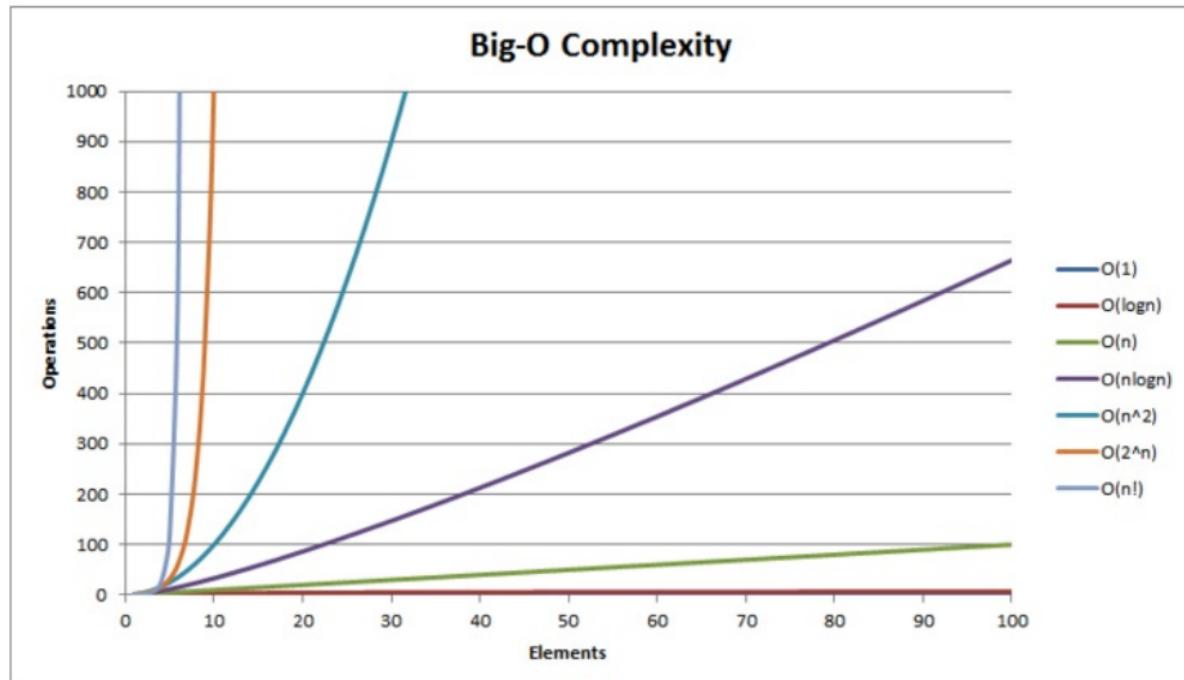


Tabelle delle Complessità

Legenda: le cose sottolineate sono quelle ipotizzate da noi

TDD List

Liste doppiamente collegate, circolari e con sentinella

Operazioni	Caso Migliore	Caso Peggio
<u>set</u> : $N \times \text{Elem} \times \text{List} \rightarrow \text{List}$	$\Theta(1)$	$\Theta(1)$
<u>add</u> : $N \times \text{Elem} \times \text{List} \rightarrow \text{List}$	$\Theta(1)$	$\Theta(n)$
<u>addBack</u> : $\text{Elem} \times \text{List} \rightarrow \text{List}$ Θ	$\Theta(1)$	$\Theta(1)$
<u>addFront</u> : $\text{Elem} \times \text{List} \rightarrow \text{List}$ $\Theta(1)$ $\Theta(1)$	$\Theta(1)$	$\Theta(1)$

$\text{removePos}: N \times \text{List} \rightarrow \text{List}$	$\Theta(1)$ (se $N==0$)	$\Theta(n)$ (se $N>=n$)
$\text{get}: N \times \text{List} \rightarrow \text{Elem}$	$\Theta(1)$ (se $N==0$)	$\Theta(n)$ (se $N>=n$)
$\text{isEmpty}: \text{List} \rightarrow \text{Bool}$	$\Theta(1)$	$\Theta(1)$
$\text{size}: \text{List} \rightarrow N$	$\Theta(n)$	$\Theta(n)$
$\text{emptyList}: \rightarrow \text{List}$	$\Theta(1)$	$\Theta(1)$

Liste Semplici

Operazioni	Caso Migliore	Caso Peggio
$\text{set}: N \times \text{Elem} \times \text{List} \rightarrow \text{List}$	$\Theta(1)$ [stesse considerazioni di prima, posso anche indicarlo con $\Theta(\min\{n, N\})$]	$\Theta(n)$ [stesse considerazioni di prima, posso anche indicarlo con $\Theta(\min\{n, N\})$]
$\text{add}: N \times \text{Elem} \times \text{List} \rightarrow \text{List}$	$\Theta(1)$	$\Theta(n)$
$\text{addBack}: \text{Elem} \times \text{List} \rightarrow \text{List}$ Θ	$\Theta(n)$	$\Theta(n)$
$\text{addFront}: \text{Elem} \times \text{List} \rightarrow \text{List}$ $\Theta(1)$ $\Theta(1)$	$\Theta(1)$	$\Theta(1)$
$\text{removePos}: N \times \text{List} \rightarrow \text{List}$	$\Theta(1)$ (se $N==0$)	$\Theta(n)$ (se $N>=n$)
$\text{get}: N \times \text{List} \rightarrow \text{Elem}$	$\Theta(1)$ (se $N==0$)	$\Theta(n)$ (se $N>=n$)
$\text{isEmpty}: \text{List} \rightarrow \text{Bool}$	$\Theta(1)$	$\Theta(1)$
$\text{size}: \text{List} \rightarrow N$	$\Theta(n)$	$\Theta(n)$
$\text{emptyList}: \rightarrow \text{List}$	$\Theta(1)$	$\Theta(1)$

Array Dinamico + size e maxsize (con ridimensionamento)

Operazioni	Caso Migliore	Caso Peggio
$\text{set}: N \times \text{Elem} \times \text{List} \rightarrow \text{List}$	$\Theta(1)$	$\Theta(1)$

$\text{add}: N \times \text{Elem} \times \text{List} \rightarrow \text{List}$	$\Theta(1)$ (aggiungo in fondo e non ridimensiono)	$\Theta(n)$
$\text{addBack}: \text{Elem} \times \text{List} \rightarrow \text{List}$	$\Theta(1)$	$\Theta(n)$
$\text{addFront}: \text{Elem} \times \text{List} \rightarrow \text{List}$	$\Theta(n)$	$\Theta(n)$
$\text{removePos}: N \times \text{List} \rightarrow \text{List}$	$\Theta(1)$	$\Theta(n)$
$\text{get}: N \times \text{List} \rightarrow \text{Elem}$	$\Theta(1)$	$\Theta(1)$
$\text{isEmpty}: \text{List} \rightarrow \text{Bool}$	$\Theta(1)$	$\Theta(1)$
$\text{size}: \text{List} \rightarrow N$	$\Theta(1)$	$\Theta(1)$
$\text{emptyList}: \rightarrow \text{List}$	$\Theta(1)$	$\Theta(1)$

TDD Set (Insiemi)

Liste (di ogni tipo)

Operazioni	Caso Migliore	Caso Peggio
emptySet	$\Theta(1)$	$\Theta(1)$
insertElem	$\Theta(1)$	$\Theta(n)$
deleteElem	$\Theta(1)$	$\Theta(n)$
setUnion	$\Theta(\max(n, m, n*m))$ non devono esserci doppiioni, se una delle due è vuota, l'unione è l'altra	$\Theta(\max(n, m, n*m))$
setIntersection	$\Theta(n*m)$	$\Theta(n*m)$
isEmpty	$\Theta(1)$	$\Theta(1)$
size	$\Theta(n)$	$\Theta(n)$
member	$\Theta(1)$	$\Theta(n)$

Bit Vector

Operazioni	Caso Migliore	Caso Peggio
<i>emptySet</i>	$\Theta(n)$	$\Theta(n)$
<i>insertElem</i>	$\Theta(1)$	$\Theta(1)$
<i>deleteElem</i>	$\Theta(1)$	$\Theta(1)$
<i>setUnion</i>	$\Theta(n)$	$\Theta(n)$
<i>setIntersection</i>	$\Theta(n)$	$\Theta(n)$
<i>isEmpty</i>	$\Theta(1)$ (<i>basta vedere se n[0] == 1</i>)	$\Theta(n)$
<i>size</i>	$\Theta(n)$ <i>devo contare gli 1</i>	$\Theta(n)$
<i>member</i>	$\Theta(1)$	$\Theta(1)$

Array Dinamici

Operazioni	Caso Migliore	Caso Peggio
<i>emptySet</i>	$\Theta(1)$	$\Theta(1)$
<i>insertElem</i>	$\Theta(1)$	$\Theta(n)$
<i>deleteElem</i>	$\Theta(1)$ (<i>non c'è shift</i>)	$\Theta(n)$
<i>setUnion</i>	$\Theta(\max(n, m, n*m))$	$\Theta(\max(n, m, n*m))$
<i>setIntersection</i>	$\Theta(n*m)$	$\Theta(n*m)$
<i>isEmpty</i>	$\Theta(1)$	$\Theta(1)$
<i>size</i>	$\Theta(1)$	$\Theta(1)$
<i>member</i>	$\Theta(1)$	$\Theta(n)$

Array Dinamici Ordinati

Operazioni	Caso Migliore	Caso Peggio
<i>emptySet</i>	$\Theta(1)$	$\Theta(1)$
<i>insertElem</i>	$\Theta(1)$ <i>se il primo elemento è già maggiore devo inserire in testa</i>	$\Theta(n)$ <i>ricerca binaria (logn) + shift (n)</i>
<i>deleteElem</i>	$\Theta(1)$ <i>se l'elemento da eliminare è minore del</i>	$\Theta(n)$

	<i>secondo devo eliminare dalla testa // stessa cosa se è maggiore del penultimo devo eliminare in coda</i>	
<i>setUnion</i>	$\Theta(n+m)$ tecnica del merge	$\Theta(n+m)$
<i>setIntersection</i>	$\Theta(1)$ quando l'intersezione è vuota	$\Theta(n+m)$
<i>isEmpty</i>	$\Theta(1)$	$\Theta(1)$
<i>size</i>	$\Theta(1)$	$\Theta(1)$
<i>member</i>	$\Theta(1)$	$\Theta(\log n)$ ricerca binaria

TDD Stack

Array Dinamici

Operazioni	Caso Migliore	Caso Peggio
<i>push:Elem x stack→stack</i>	$\Theta(1)$	$\Theta(n)$ (se devo ridimensionare)
<i>pop: Stack → Elem x Stack</i>	$\Theta(1)$	$\Theta(n)$ (se devo ridimensionare)
<i>top: Stack → Elem</i>	$\Theta(1)$	$\Theta(1)$
<i>emptyStack:→Stack</i>	$\Theta(1)$	$\Theta(1)$
<i>isEmpty: Stack → Bool</i>	$\Theta(1)$	$\Theta(1)$

Liste Doppiamente collegate, Circolari e con Sentinella

Operazioni	Caso Migliore	Caso Peggio
<i>push: Elem x Stack → Stack</i>	$\Theta(1)$	$\Theta(1)$
<i>pop: Stack → Stack x Elem</i>	$\Theta(1)$	$\Theta(1)$

<i>top: Stack → Elem</i>	$\Theta(1)$	$\Theta(1)$
<i>emptyStack: → Stack</i>	$\Theta(1)$	$\Theta(1)$
<i>isEmpty: Stack → Bool</i>	$\Theta(1)$	$\Theta(1)$

Liste Semplici

<i>Operazioni</i>	<i>Caso Migliore</i>	<i>Caso Peggio</i>
<i>push: Elem x Stack → Stack</i>	$\Theta(1)$	$\Theta(1)$
<i>pop: Stack → Stack x Elem</i>	$\Theta(1)$	$\Theta(1)$
<i>top: Stack → Elem</i>	$\Theta(1)$	$\Theta(1)$
<i>emptyStack: → Stack</i>	$\Theta(1)$	$\Theta(1)$
<i>isEmpty: Stack → Bool</i>	$\Theta(1)$	$\Theta(1)$

TDD Queue

Array Dinamici

<i>Operazioni</i>	<i>Caso Migliore</i>	<i>Caso Peggio</i>
<i>enqueue: Elem x Queue → Queue</i>	$\Theta(1)$	$\Theta(n)$ (se devo ridimensionare)
<i>dequeue: Queue → Elem x Queue</i>	$\Theta(n)$ (tolgo da pos==0 e devo shiftare a sinistra)	$\Theta(n)$
<i>first: Queue → Elem</i>	$\Theta(1)$	$\Theta(1)$
<i>emptyQueue: → Queue</i>	$\Theta(1)$	$\Theta(1)$
<i>isEmpty: Queue → Bool</i>	$\Theta(1)$	$\Theta(1)$

Liste Semplici

<i>Operazioni</i>	<i>Caso Migliore</i>	<i>Caso Peggio</i>

<i>enqueue: Elem x Queue → Queue</i>	$\Theta(n)$ (<i>metto in fondo</i>)	$\Theta(n)$
<i>dequeue: Queue → Elem x Queue</i>	$\Theta(1)$ (<i>prendo dalla testa</i>)	$\Theta(1)$
<i>first: Queue → Elem</i>	$\Theta(1)$	$\Theta(1)$
<i>emptyQueue: → Queue</i>	$\Theta(1)$	$\Theta(1)$
<i>isEmpty: Queue → Bool</i>	$\Theta(1)$	$\Theta(1)$

Liste Doppiamente collegate, Circolari e con Sentinella

<i>Operazioni</i>	<i>Caso Migliore</i>	<i>Caso Peggio</i>
<i>enqueue: Elem x Queue → Queue</i>	$\Theta(1)$	$\Theta(1)$
<i>dequeue: Queue → Elem x Queue</i>	$\Theta(1)$	$\Theta(1)$
<i>first: Queue → Elem</i>	$\Theta(1)$	$\Theta(1)$
<i>emptyQueue: → Queue</i>	$\Theta(1)$	$\Theta(1)$
<i>isEmpty: Queue → Bool</i>	$\Theta(1)$	$\Theta(1)$

Hash Table

Liste di Collisione

<i>Operazioni</i>	<i>Caso Migliore</i>	<i>Caso Peggio</i>
<i>insert(elem e, chiave k)</i>	$\Theta(1)$	$\Theta(1+n/m)$
<i>delete(chiave k)</i>	$\Theta(1)$	$\Theta(1+n/m)$
<i>search(chiave k) → elem</i>	$\Theta(1)$	$\Theta(1+n/m)$

TDD Tree

Alberi Binari con Liste Semplici

Operazioni	Caso Migliore	Caso Peggio
$\text{emptyTree} \rightarrow \text{Tree}$	$\Theta(1)$	$\Theta(1)$
$\text{emptyLabel} \rightarrow \text{Label}$	$\Theta(1)$	$\Theta(1)$
$\text{IsEmpty}:\text{Tree} \rightarrow \text{Bool}$	$\Theta(1)$	$\Theta(1)$
$\text{AddElem}:\text{Label} \times \text{Label} \times \text{Tree} \rightarrow \text{Tree}$	$\Theta(1)$	$\Theta(h)$
$\text{DeleteElem}:\text{Label} \times \text{Tree} \rightarrow \text{Tree}$	$\Theta(1)$	$\Theta(h)$
$\text{Father}:\text{Label} \times \text{Tree} \rightarrow \text{Label}$	<u>$\Theta(1)$</u>	<u>$\Theta(h)$</u>
$\text{Children}:\text{Label} \times \text{Tree} \rightarrow \text{List}<\text{Label}>$	<u>$\Theta(\text{grado}(v)+h)$</u>	<u>$\Theta(\text{grado}(v)+h)$</u>
$\text{Degree}:\text{Label} \times \text{Tree} \rightarrow \text{int}$	<u>$\Theta(\text{grado}(v)+h)$</u>	<u>$\Theta(\text{grado}(v)+h)$</u>
$\text{Ancestors}:\text{Label} \times \text{Tree} \rightarrow \text{List}<\text{Label}>$	<u>$\Theta(1)$ la radice non ha ancestors</u>	<u>$\Theta(h)$ cerco gli antenati dell'ultimo nodo</u>
$\text{LeastCommonAncestor}:\text{Label} \times \text{Label} \times \text{Tree} \rightarrow \text{Label}$	<u>$\Theta(1)$ l'ancestor in comune è il primo di entrambe le liste</u>	<u>$\Theta(\min(x,y))$ dove n ed m sono le dimensioni delle due liste di Ancestors</u>
$\text{Member}:\text{Label} \times \text{Tree} \rightarrow \text{Bool}$	$\Theta(1)$	$\Theta(h)$

Alberi Generici (Primo Figlio - Prossimo Fratello)

Operazioni	Caso Migliore	Caso Peggio
$\text{emptyTree} \rightarrow \text{Tree}$	$\Theta(1)$	$\Theta(1)$
$\text{emptyLabel} \rightarrow \text{Label}$	$\Theta(1)$	$\Theta(1)$
$\text{isEmpty}:\text{Tree} \rightarrow \text{Bool}$	$\Theta(1)$	$\Theta(1)$

<i>AddElem:Label x Label x Tree → Tree</i>	<u>$\Theta(1)$ nel caso in cui inserisco la radice</u>	<u>$\Theta(n)$</u>
<i>DeleteElem:Label x Tree → Tree</i>	<u>$\Theta(1)$ nel caso in cui provo a eliminare la radice</u>	<u>$\Theta(n)$</u>
<i>Father:Label x Tree → Label</i>	<u>$\Theta(1)$</u>	<u>$\Theta(n)$</u>
<i>Children:Label x Tree → List<Label></i>	<u>$\Theta(n)$</u>	<u>$\Theta(n)$</u>
<i>Degree:Label x Tree → int</i>	<u>$\Theta(n)$</u>	<u>$\Theta(n)$</u>
<i>Ancestors:Label x Tree → List<Label></i>	<u>$\Theta(1)$ la radice non ha antenati</u> <u>dell'ultimo nodo</u>	<u>$\Theta(n)$ cerco gli antenati dell'ultimo nodo</u>
<i>LeastCommonAncestor:Label x Label x Tree → Label</i>	<u>$\Theta(1)$ l'ancestor in comune è il primo di entrambe le liste</u>	<u>$\Theta(\min(x,y))$ dove x ed y sono le dimensioni delle due liste di Ancestors</u>
<i>Member:Label x Tree → Bool</i>	$\Theta(1)$	$\Theta(n)$

TDD Graph

Liste di Adiacenza con Vertici in un Array

Operazioni	Caso Migliore	Caso Peggio
<i>degree</i>	$\Theta(\delta(v))$	$\Theta(\delta(v))$
<i>incidentEdges</i>	$\Theta(\delta(v))$	$\Theta(\delta(v))$
<i>areAdjacent</i>	$\Theta(1)$	$\Theta(\min(\delta(x), \delta(y)))$
<i>addVertex</i>	$\Theta(1)$	$\Theta(n)$ se devo riallocare
<i>addEdge</i>	$\Theta(1)$	$\Theta(\min(\delta(x), \delta(y)))$ chiama <i>areAdjacent</i> per vedere se l'arco c'è già
<i>removeVertex</i>	$\Theta(\delta(v))$	$\Theta(m)$
<i>removeEdge</i>	$\Theta(\delta(x) + \delta(y))$	$\Theta(\delta(x) + \delta(y))$

Liste di Adiacenza con Vertici in una Lista

<i>Operazioni</i>	<i>Caso Migliore</i>	<i>Caso Peggio</i>
<i>degree</i>	$\Theta(n)$	$\Theta(n)$
<i>incidentEdges</i>	$\Theta(n)$	$\Theta(n)$
<i>areAdjacent</i>	$\Theta(n)$	$\Theta(n)$
<i>addVertex</i>	$\Theta(n)$	$\Theta(n)$
<i>addEdge</i>	$\Theta(n)$	$\Theta(n)$
<i>removeVertex</i>	$\Theta(m+n)$	$\Theta(m+n)$
<i>removeEdge</i>	$\Theta(n)$	$\Theta(n)$

Matrici di Adiacenza

<i>Operazioni</i>	<i>Caso Migliore</i>	<i>Caso Peggio</i>
<i>degree</i>	$\Theta(n)$	$\Theta(n)$
<i>incidentEdges</i>	$\Theta(n)$	$\Theta(n)$
<i>areAdjacent</i>	$\Theta(1)$	$\Theta(1)$
<i>addVertex</i>	$\Theta(n)$	$\Theta(n^2)$ richiede riallocazione
<i>addEdge</i>	$\Theta(1)$	$\Theta(1)$
<i>removeVertex</i>	$\Theta(n^2)$	$\Theta(n^2)$
<i>removeEdge</i>	$\Theta(1)$	$\Theta(1)$

TDD Priority Queue

Array

<i>Operazioni</i>	<i>Caso Migliore</i>	<i>Caso Peggio</i>
<i>emptyPriorityQueue: -> PriorityQueue</i>	$\Theta(1)$	$\Theta(1)$

<i>findMax: PriorityQueue -> Elem</i>	$\Theta(1)$	$\Theta(1)$
<i>insertElem: Elem x Key x PriorityQueue -> PriorityQueue</i>	$\Theta(n)$	$\Theta(2n) = \Theta(n)$ in caso di ridimensionante
<i>deleteMax: PriorityQueue -> PriorityQueue</i>	$\Theta(1)$	$\Theta(1)$ o $\Theta(n)$ in caso di ridimensionamento

Lista Semplice con Chiavi Ordinate in modo decrescente

<i>Operazioni</i>	<i>Caso Migliore</i>	<i>Caso Peggio</i>
<i>emptyPriorityQueue: -> PriorityQueue</i>	$\Theta(1)$	$\Theta(1)$
<i>findMax: PriorityQueue -> Elem</i>	$\Theta(1)$	$\Theta(1)$
<i>insertElem: Elem x Key x PriorityQueue -> PriorityQueue</i>	$\Theta(1)$	$\Theta(n)$
<i>deleteMax: PriorityQueue -> PriorityQueue</i>	$\Theta(1)$	$\Theta(1)$

TDD Heap Binario

Array

<i>Operazioni</i>	<i>Caso Migliore</i>	<i>Caso Peggio</i>
<i>emptyPriorityQueue: -> PriorityQueue</i>	$\Theta(1)$	$\Theta(1)$
<i>findMax: PriorityQueue -> Elem</i>	$\Theta(1)$ è in ordine quindi è il primo	
<i>insertElem: Elem x Key x PriorityQueue -> PriorityQueue</i>	$\Theta(1)$ senza ridimensionamento	$\Theta(log n)$ procedura muovi verso l'alto
<i>deleteMax: PriorityQueue -> PriorityQueue</i>	$\Theta(log n)$ senza ridimensionamento	$\Theta(log n)$ procedura muovi verso il basso

Domande Tipo d'Esame

Mergesort

Se all'esame si chiede lo pseudo-codice di merge...

Scegli un'alternativa:

- a. ...è fondamentale scrivere il "cuore" dell'algoritmo, in cui gli elementi delle sottosequenza vengono confrontati due a due ed il "vincitore" (il minore) viene copiato nella sequenza risultante. Si può omettere il codice relativo a cosa succede quando una delle due sottosequenze è terminata, ma bisogna almeno scrivere "si trascurano i casi estremi in cui una delle due sottosequenze è stata scandita completamente" ✓ Giusto
- b. ...è fondamentale scrivere anche lo pseudo-codice per i casi in cui la scansione di una delle due sotto-sequenze sia terminata, e gli elementi dell'altra vadano copiati nella sequenza risultato. La mancanza di questa parte dello pseudo-codice invalida la risposta.

Risposta Ideale su Mergesort

Esempio 1:

D4

Siccome è ricorsiva, per conoscere il costo di mergesort bisogna calcolare il costo su ogni livello ~~moltiplicato~~ moltiplicato per il numero dei livelli. (dove per livello si intende , il livello della ricorsione)

costo su ogni livello:

sappiamo che merge ha costo $\Theta(n)$ lineare

su ogni livello si hanno 2^j sottoproblemi di tipo merge, ognuno lungo $\frac{n}{2^j}$ e quindi risolvibile in $\Theta(\frac{n}{2^j})$

per sapere il costo su ogni livello bisogna moltiplicare il costo di merge per il numero di volte che viene chiamato:

$$2^j \times \Theta\left(\frac{n}{2^j}\right) = \Theta\left(2^j \times \frac{n}{2^j}\right) = \Theta(n)$$

ogni livello costa $\Theta(n)$

numero dei livelli:

sapendo che all'ultimo livello della ricorsione i sottoproblemi assumono dimensione 1, e su ogni livello i sottoproblemi sono di dimensione $\frac{n}{2^j}$, per quale j si ha che

$$\frac{n}{2^j} = 1 ? \quad n = 2^j \rightarrow \log_2 n = j$$

Quindi, n° di livelli = $\log_2 n + 1$ - perché si parte dal livello 0

In conclusione, il costo di mergesort è dato da:

$$\Theta(n) \times \log_2 n + \Theta(n) \rightarrow \Theta(n \log n)$$

e vale sia nel caso migliore sia nel caso peggiore

siccome è ricorsiva, per conoscere il costo di mergesort bisogna calcolare il costo di ogni livello moltiplicato per il numero dei livelli (dove per livello si intende il livello della ricorsione)

costo su ogni livello:

sappiamo che merge ha costo $\Theta(n)$ lineare su ogni livello si hanno 2^j sottoproblemi di tipo merge, ognuno lungo $n/2^j$ e quindi risolvibile in $\Theta(n/2^j)$

per sapere il costo di ogni livello bisogna moltiplicare il costo di merge per il numero di volte che viene chiamato $2^j \times \Theta(n/2^j) = \Theta(n)$. ogni Livello costa $\Theta(n)$.

numero dei livelli:

sapendo che all'ultimo livello della ricorsione i sottoproblemi assumono dimensione 1, e che su ogni livello i sottoproblemi sono di dimensione $n/2^j$ per quale j si ha che $n/2^j=1$?

$$n=2^j \rightarrow \log_2 n = j.$$

In conclusione il costo di merge sort è dato da $\Theta(n) * \log n = \Theta(n \log n)$

Esempio 2.1:

sage scutell

A) La complessità della funzione inconcerne deriva dal fatto dell'espressione
m° Ercelli x costo esercizio operazioni singole elette

A sua volta il costo delle operazioni deriva da

nº problemi \times costo - viscum problema

~~costo~~ marge

At ogni chilometro ricevete si ferme.

livello	N° problemi	Tipi problemi		1 2 3 4 5 6 7 8 9	
L0	1	m		3 2 5 1 6 9 4 7	
L1	2	m/2		1 2 3 4 5 3 2 5 1 6 9 4 7	L1
L2	4	m/4		2 3 4 5 3 2 5 1 6 9 4 7 3 2 5 1 6 9 4 7	L2
L3	8	m/8		3 2 5 1 6 9 4 7	L3
				m = 8 dimensione array intrevallo = 0..2020	

Indicazioni con le quali presso la considerazione, i valori nella tabella

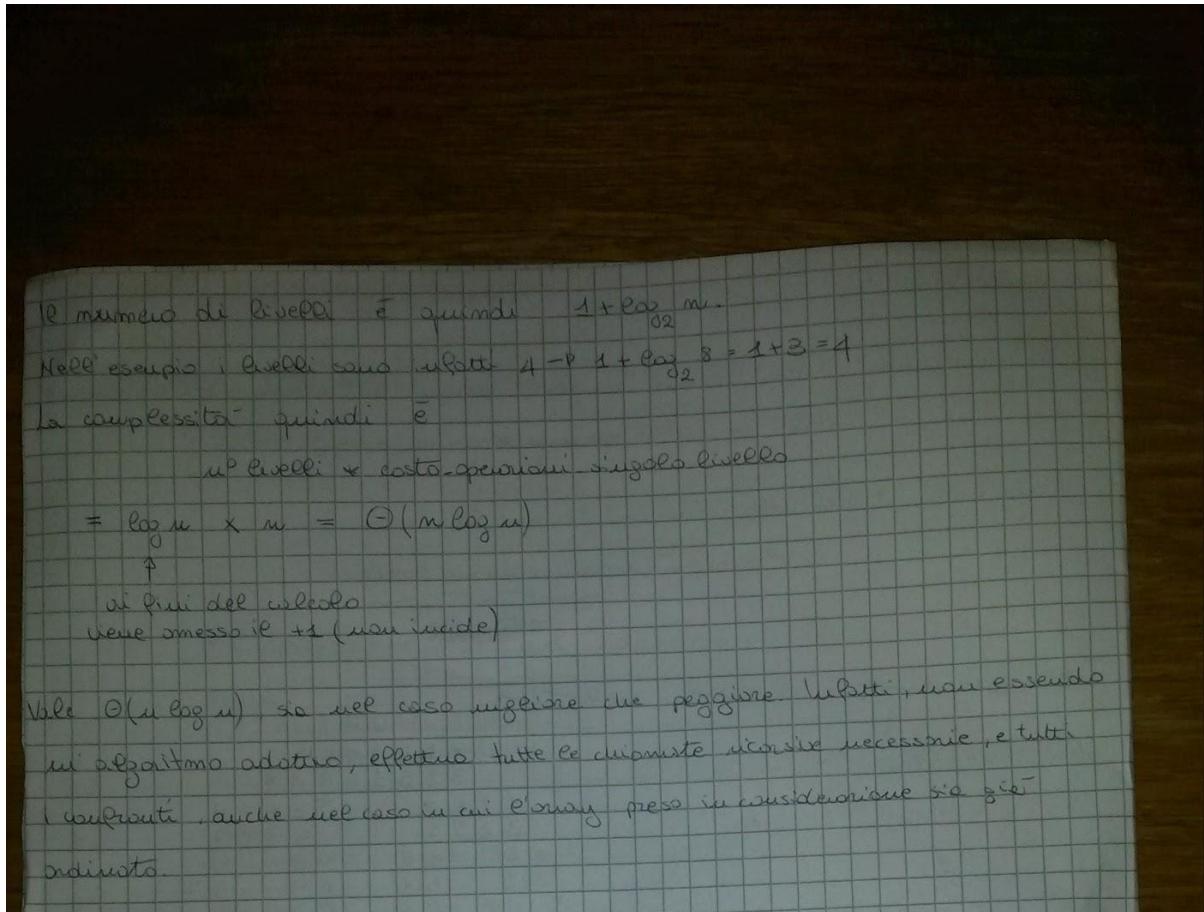
2^j per il numero di problemi e $u/2^j$ per la dimensione del singolo

Risolvendo la formula sopra, si ottiene il costo delle spese sui 1000000

$$2^j \times \underbrace{\Theta(u/2^j)}_{\text{costo merge}} = (2^j \times u/2^j) = m \rightarrow \Theta(u) \text{ le operazioni hanno complessità in } \Theta(u)$$

Per quanto riguarda il mo^do di livelli, consideriamo che se livelli più profondi dell'albero Δ la dimensione dei problemi è 1.

Esempio 2.2:



Come viene corretto mergesort:

Domanda 2, 2.25 punti											
Caratterizzazione (spiegare caso migliore e peggiore)		Complessità (dire che è Theta(n log n))		Operazioni per Livello (spiegare da dove viene Theta(n))			Merge (spiegare la funzione fondi)		Profondità (spiegare perché la profondità di MS è dell'ordine di log n)		
no	ok	no	ok	no	impr	ok	no	ok	no	impr	ok
-0,35		-0,35		-0,65	-0,3		-0,25		-0,65	-0,3	

Hashtable

Quali caratteristiche deve avere una buona tabella di Hash?

-La sua funzione deve essere calcolabile in tempo costante

-Essere uniformemente distribuita

NON DEVE ESSERE INIETTIVA A MENO CHE NON SI PARLI DI FUNZIONE PERFETTA

Matrice di Adiacenza

In un grafo privi di cappi (archi che tornano un vertice a se stesso) la diagonale cosa contiene?

SEMPRE 0 (non posso avere un arco che collega A con A perché sarebbe un cappio)

Se il grafo è non orientato, che proprietà ha la matrice di adiacenza?

LA MATRICE E' SIMMETRICA RISPETTO ALLA DIAGONALE

Altri Esempi di Domande

Scegli un'alternativa:

- a.
ad ASD abbiamo supportato encapsulation e information hiding mediante l'uso di namespace; questo è quanto di meglio la comunità informatica abbia inventato fino a questo momento e non esistono approcci migliori
- b.
ad ASD abbiamo supportato encapsulation e information hiding mediante l'uso di namespace; esistono approcci migliori e più eleganti, ad esempio la programmazione ad oggetti che studierete a "LPO" il prossimo anno
- c.
ad ASD non ci siamo mai posti il problema di cosa siano e di come supportare encapsulation e information hiding
- d.
i tipi di dato non hanno niente a che fare con encapsulation e information hiding
- e.
l'information hiding è una pratica obsoleta: le moderne tecniche di ingegneria del software prevedono di non nascondere più il funzionamento interno – deciso in fase di progetto – di una parte di un programma, ma di rendere qualunque dato, funzione e dettaglio implementativo accessibile a qualunque applicazione ed utente, per ragioni di sicurezza e modularità

Stato	Completato
Terminato	venerdì, 27 marzo 2020, 09:38
Tempo impiegato	48 secondi
Punteggio	1,00/1,00
Valutazione	10,00 su un massimo di 10,00 (100%)

Domanda **1**

Risposta corretta

Punteggio ottenuto 1,00 su 1,00

Contrassegna domanda

Formalmente, a cosa corrisponde l'interfaccia di un tipo di dato?

Scegli un'alternativa:

- a. a un insieme dotato di ordinamento
- b. a un elemento di un insieme
- c. a un'algebra su una segnatura eterogenea
- d. a una struttura dati
- e. a una segnatura many-sorted, ovvero eterogenea ✓

Risposta corretta.

La risposta corretta è: a una segnatura many-sorted, ovvero eterogenea

Alberi Rosso-Neri

Sono più complessi dei BST ma garantiscono una complessità delle operazioni di ricerca, inserimento e cancellazione che dipende dall'altezza dell'albero, che dipende a sua volta dal numero di nodi n secondo la formula: $2\log_2(n + 1)$

Algoritmo di Dijkstra

Cosa Serve? A calcolare (in un grafo) il cammino con costo minimo da un vertice ai tutti gli altri vertici.

Quick Sort

Come scegliere un buon pivot?

Idea scelta casuale(rand) del pivot:

Ad ogni chiamata ricorsiva,scegliamo come pivot uno dei numeri dell'array,a caso,con questo approccio alla scelta del pivot,abbiamo una versione di Quicksort

randomizzata , nel quale due esecuzioni diverse sullo stesso input possono svolgersi in modo diverso, il risultati delle due esecuzioni deve essere lo stesso.

Per ogni array di dimensione n, il tempo di esecuzione del quicksort randomizzato nel caso medio è **O(nlogn)**.

Descrivi complessità QuickSort caso migliore caratterizzando il caso migliore e spiegando come si calcola la complessità

Il caso migliore in assoluto è un caso ipotetico in cui scelgo sempre come pivot l'elemento mediano della porzione di array che sto considerando.

il caso medio il pivot è scelto in modo casuale in modo da dividere $\frac{1}{4}$ E $\frac{3}{4}$ di più grandi o viceversa.

Questo ci fa ricadere nel caso di complessità **O(nlogn)**.

Le operazioni per livello si calcolano come in merge sort ovvero ad ogni livello vengono fatte **O(n)** operazioni che coinvolgono sequenze lunghe $n/2^j$ e il numero di sequenze coinvolte sono 2^j .

La complessità quicksort nel caso peggiore è quando il pivot è l'elemento più grande o più piccolo , in quanto l'array viene partizionato in modo sbilanciato.

Come si caratterizza il caso migliore di quicksort?

Il caso migliore di quicksort si ha quando ad ogni chiamata di partition , il pivot è il mediano degli elementi compresi nella porzione dell'array A tra inizio e fine.
Ovviamente questo caso è teorico: non si può infatti calcolare l'elemento mediano senza riordinare prima gli elementi.

Nella complessità di quicksort nel caso migliore O(nlogn), da dove viene il fattore n?

Dal numero di operazioni svolte ad ogni livello dell'albero della ricorsione: al livello j si effettuano 2^j chiamate partition, ciascuna su una porzione di array lunga $(n/2^j)$.

Ciascuna di queste chiamate ha complessità lineare nella dimensione della porzione di array su cui viene chiamata, quindi **O(n/2^j)**.Ad ogni livello faccio $2^j * O(n/2^j)$ operazioni che determinano **O(n)**.

Nella complessità quicksort nel caso migliore ,O(nlogn) da dove viene il fattore logn?

Dal numero dei livelli dell'albero della ricorsione.

Quando si ricade nel caso peggiore di QuickSort?

Quando Partition chiamato su A tra inizio e fine seleziona sempre l'elemento minore o maggiore tra quelli compresi tra inizio e fine.

Come si calcola la complessità nel caso peggiore di quicksort?

Si sommano le operazioni fatte ad ogni livello dell'albero delle chiamate ricorsive.

tali operazioni sono dovute alla chiamata partition, che al livello 0 verrà chiamata su n elementi (quindi n operazioni), al livello 1 verrà chiamata su n-1 elementi,[...], al livello n verrà chiamata su un elemento e poi non ci saranno altre chiamate ricorsive.

questo calcolo si sviluppa nella sommatoria per i che va da 1 a n in **O(n^2)**.

Esempi di Domande da Quiz Passati

10/03

Supponendo che il file header includa la direttiva

```
const List EMPTY_SEQ = NULL;
```

dove List è implementata come descritto nelle slide, qual è il modo corretto di implementare la funzione

```
bool isEmpty(const List& l)
```

che restituisce true se la lista è vuota e false altrimenti?

La risposta corretta è:

```
bool isEmpty(const List& l)
{
    return (l == EMPTY_SEQ);
}
```

Quale delle seguenti affermazioni è vera, rispetto alla funzione size descritta nelle slide, nel codice e nei podcast?

Risposta corretta.

La risposta corretta è: La funzione size è implementata correttamente e il controllo isEmpty(l) evita che si acceda l->next, nel caso in cui l sia NULL

Una variabile di tipo puntatore....

La risposta corretta è:

...può essere passata per riferimento, se il suo valore viene modificato nel corpo della funzione e questa modifica deve essere visibile anche quando la funzione ha terminato la sua esecuzione

11/03

Si considerino le seguenti affermazioni: una sola è corretta. Indicare quale.

Ad ASD abbiamo supportato encapsulation e information hiding mediante l'uso di namespace; esistono approcci migliori e più eleganti, ad esempio la programmazione ad oggetti che studierete a "LPO" il prossimo anno.

13/03

Quando aggiungo in testa ad una lista semplice, l'indirizzo del primo elemento della lista.....

Risposta corretta.

La risposta corretta è: cambia: il nuovo indirizzo sarà l'indirizzo della cella creata all'interno della funzione "insert". E' per questo che dobbiamo passare il parametro formale corrispondente all'indirizzo di inizio lista, per riferimento

Quando scandisco una lista semplice "s", dal primo elemento all'ultimo, la condizione per verificare di avere raggiunto l'ultimo elemento della lista è.....

Risposta corretta.

La risposta corretta è: che il puntatore a next dell'elemento corrente sia NULL (o nullptr, o molto meglio emptyList, se ho definito una costante apposita)

Quando scandisco una lista circolare "s", dal primo elemento all'ultimo, la condizione per verificare di avere raggiunto l'ultimo elemento della lista è.....

Risposta corretta.

La risposta corretta è: che il puntatore a next dell'elemento corrente sia "s" (ovvero il puntatore all'inizio della lista)

Quando aggiungo un elemento in testa ad una lista circolare, l'indirizzo del primo elemento della lista.....

Risposta corretta.

La risposta corretta è: cambia: il nuovo indirizzo sarà l'indirizzo della cella creata all'interno della funzione "insert". E' per questo che dobbiamo passare il parametro formale corrispondente all'indirizzo di inizio lista, per riferimento. Inoltre, devo anche

preoccuparmi di aggiornare il "next" dell'ultimo elemento, che adesso punterà a quello appena inserito

16/03

La ricerca binaria ricorsiva presentata nelle slide di IP e la ricerca binaria ricorsiva presentata nelle slide di ASD....

...implementano lo stesso algoritmo di ricerca binaria, con un'unica differenza: nella versione di IP la funzione restituisce solo l'informazione che l'elemento cercato appartenga o meno alla struttura dati, nella versione di ASD viene restituita la posizione dell'elemento (se c'è).

17/03

ES4: Supponiamo che l'assegnazione “=” costi c_1 , il confronto “<” e “==” costino c_2 , l'incremento “++” costi c_3 , la stampa “cout<<” costi c_4 e la lettura da stdin “cin>>” costi c_5 .

```
int n;  
cin >> n;  
for (int i=0; i<n; ++i)  
    for (int j=0; j<n; ++j)  
        cout << "(" << i << "," << j << ")\n";
```

Risposta corretta.

La risposta corretta è: Sarà proporzionale a n^2 , visto che per ogni i (che va da 0 a n , quindi per circa n oppure $n+1$ volte) ripeto circa n operazioni (dovute al for annidato internamente, dove ho “per ogni j , con j che va da 0 a n ”). Quindi ho circa $n * n$ operazioni (quelle del ciclo esterno, per quelle del ciclo interno). Ma a calcolare il numero esatto di c_1, c_2, c_3, \dots , non ce la faccio proprio....

Perché nel podcast precedente si è tolta la base del logaritmo quando si è semplificato
 $7n \log_2 n + 7n \stackrel{?}{=} n \log n$

(ovvero, stiamo affermando che $7n \log_2 n + 7n$ appartiene a $\Theta(n \log n)$) ?

Risposta corretta.

La risposta corretta è: La risposta è stata data in un podcast del 9 marzo, nel ripasso dei logaritmi: si può sempre cambiare la base del logaritmo moltiplicando per una costante. Visto che quando calcoliamo la complessità degli algoritmi le costanti moltiplicative non ci interessano, anche la base del logaritmo diventa irrilevante e la possiamo trascurare

La complessità del frammento di codice mostrato nell'esercizio ES5, considerando i principi del calcolo di complessità (trascurare costanti moltiplicative, trascurare termini di ordine inferiore) è....

$\Theta(n^2)$

Qual è la complessità di `read_list` chiamata su uno stream di n elementi, dove l'inserimento dell'elemento nella lista avviene in testa?

Risposta corretta.

La risposta corretta è: $\Theta(n)$ sia nel caso migliore che nel caso peggiore

Qual è la complessità della funzione `print_list` chiamata su una lista `list` che contiene n elementi?

Risposta corretta.

La risposta corretta è: $\Theta(n)$ sia nel caso migliore che nel caso peggiore

Nell'esercizio 3, la docente ha insistito sul fatto che la complessità degli algoritmi va calcolata nel caso peggiore e nel caso migliore (a volte anche nel caso medio), perché spesso la complessità è diversa in questi due casi: perché non ha più specificato se stavamo ragionando nel caso migliore/peggiore negli altri esercizi?

Risposta corretta.

La risposta corretta è: Perché si è dimenticata di dirlo.... ma la ragione per cui si è dimenticata è che negli esercizi visti fino ad ora, a parte il #3, il caso migliore e peggiore coincidono o, detto in altro modo, non esistono un caso migliore ed uno peggiore: l'algoritmo ha la stessa complessità in ogni caso, ovviamente espressa in funzione di n

18/03

Qual è la complessità di `read_list` su uno stream di n elementi, supponendo che `last_insert` inserisca un elemento in coda alla lista `list` e che `list` sia una lista collegata semplicemente e con sentinella (il fatto che `list` sia circolare o no, non cambia la complessità: cambierebbe solo se fosse circolare e doppiamente collegata)

Theta (n^2) nel caso migliore e peggiore. Per la presenza della sommatoria; inoltre, non vale dire che se lo stream è vuoto, la complessità è costante. Questa informazione è già catturata dal fatto che se $n = 0$, la complessità è Theta (0^2) = Theta (0) che sta ad indicare, come Theta (1), una complessità costante. Non serve discriminare due casi migliore/peggiore, se sono già entrambi catturati da un'unica espressione.

19/03

Con Big-Oh(g (n)) si denota..

La risposta corretta è: La classe delle funzioni che "da un certo $n_0 \geq 0$ in poi" sono dominate da $c * g (n)$, con c costante moltiplicativa reale > 0 (questa c non è uguale per ogni funzione nella classe, ogni funzione potrà richiedere una costante diversa per essere dominata da $c * g(n)$ "da un certo punto in poi")

Con Omega(g (n)) si denota...

Risposta corretta.

La risposta corretta è: La classe delle funzioni che "da un certo $n_0 \geq 0$ in poi" dominano $c * g (n)$, con c costante moltiplicativa reale > 0 (questa c non è uguale per ogni funzione nella classe, ogni funzione potrà richiedere una costante diversa per dominare $c * g(n)$ "da un certo punto in poi")

Con Theta(g (n)) si denota...

Risposta corretta.

La risposta corretta è: La classe delle funzioni che appartengono all'intersezione tra Big-Oh(g (n)) e Omega(g (n))

La complessità di selection sort è...

Risposta corretta.

La risposta corretta è: Theta(n^2) nel caso migliore e peggiore

La complessità di insertion sort è...

Risposta corretta.

La risposta corretta è: Theta (n) nel caso migliore e Theta(n^2) nel caso peggiore

La complessità di bubble sort è...

Theta (n) nel caso migliore e Theta(n^2) nel caso peggiore

24/03

Qual è la complessità di mergesort?

Theta (n log n) nel caso migliore e peggiore. La complessità per ogni livello ricorsivo è: $\Theta(n)$. Il numero di livelli è $\log_2 n$

Nella complessità di mergesort, Theta (n log n), da dove viene il fattore "n" ?

Dal numero di operazioni svolte ad ogni livello dell'albero della ricorsione: al livello j si effettuano 2^j chiamate a merge, ciascuna su una porzione di array lunga $n/(2^j)$. Ciascuna di queste chiamate ha complessità lineare nella dimensione della porzione su

cui viene chiamata, quindi Theta ($n/(2^j)$): ad ogni livello faccio $2^j * \Theta(n/(2^j))$ operazioni, che determinano la complessità Theta(n).

Nella complessità di mergesort, Theta ($n \log n$), da dove viene il fattore "log n" ?

Dal numero dei livelli dell'albero della ricorsione. Le chiamate ricorsive totali su MergeSort

25/03

Sia A un array di lunghezza n; siano inizio e fine gli indici che delimitano la porzione di A (con inizio e fine compresi) nella quale viene chiamata partition. Qual è la complessità di partition nel caso migliore e peggiore?

Risposta corretta.

La risposta corretta è: Theta (fine-inizio) nel caso migliore e peggiore: la complessità dipende dalla dimensione della porzione di A su cui partition opera

Quale delle seguenti affermazioni su partition è corretta?

Risposta corretta.

La risposta corretta è: Partition può essere implementata "in place", senza strutture di appoggio: è una funzione molto efficiente per questa ragione

Qual è la complessità di quicksort?

Risposta corretta.

La risposta corretta è: Theta ($n \log n$) nel caso migliore e medio; Theta (n^2) nel caso peggiore

Come si caratterizza il caso migliore di quicksort?

Risposta corretta.

La risposta corretta è: Il caso migliore di quicksort si ha quando ad ogni chiamata di partition, il pivot è il mediano degli elementi degli elementi compresi nella porzione dell'array A tra inizio e fine. Ovviamente questo caso migliore è teorico: non si può infatti calcolare l'elemento mediano senza riordinare prima gli elem

Nella complessità di quicksort nel caso migliore, Theta ($n \log n$), da dove viene il fattore "n" ?

Risposta corretta.

La risposta corretta è: Dal numero di operazioni svolte ad ogni livello dell'albero della ricorsione: al livello j si effettuano 2^j chiamate a partition, ciascuna su una porzione di array lunga $n/(2^j)$. Ciascuna di queste chiamate ha complessità lineare nella dimensione della porzione di array su cui viene chiamata, quindi Theta ($n/(2^j)$): ad ogni livello faccio $2^j * \Theta(n/(2^j))$ operazioni, che determinano la complessità Theta(n)

Nella complessità di quicksort nel caso migliore, Theta ($n \log n$), da dove viene il fattore "log n" ?

La risposta corretta è: Dal numero dei livelli dell'albero della ricorsione

Quando si ricade nel caso peggiore di quicksort?

Risposta corretta.

La risposta corretta è: Quando partition chiamata su A tra "inizio" e "fine" seleziona sempre come pivot l'elemento maggiore o l'elemento minore tra quelli compresi tra "inizio" e "fine"

Come si calcola la complessità nel caso peggiore di quicksort?

Risposta corretta.

La risposta corretta è: Si sommano le operazioni fatte ad ogni livello dell'albero delle chiamate ricorsive. Tali operazioni sono dovute alla chiamata di partition, che al livello 0 verrà chiamata su n elementi (quindi n operazioni), al livello 1 verrà chiamata su $n-1$ elementi (quindi $n-1$ operazioni), al livello 2 verrà chiamata su $n-2$ elementi (quindi $n-2$ operazioni), ..., al livello n verrà chiamata su 1 elemento (una sola operazione, e poi non ci sono altre chiamate ricorsive). Questo calcolo si sviluppa nella sommatoria per i che va da 1 a n di i , in Theta (n^2)

Formalmente, a cosa corrisponde l'interfaccia di un tipo di dato?

La risposta corretta è: a una segnatura many-sorted, ovvero eterogenea

30/03

Riflessione #2 (associata a quiz)

Consideriamo addFront: Elem x List → List

La segnatura di questa funzione indica che l'inserimento **MODIFICA** la lista a cui si applica, oppure che **LA LASCIA INALTERATA** e restituisce una lista nuova, uguale a quella su cui abbiamo chiamato addFront nella quale però abbiamo aggiunto il nuovo elemento in testa?

Risposta parzialmente esatta.

Hai selezionato correttamente 1.

Chi ha votato per **MODIFICA** ha ragione, se volevamo implementare un'operazione "stateful" (ovvero, un'operazione che modifica gli argomenti a cui è applicata, quindi un'operazione con side effect)

Chi vota per **LASCIA INALTERATA LA LISTA ARGOMENTO E RESTITUISCE UNA LISTA NUOVA** ha ragione, se volevamo implementare una funzione che sia tale anche dal punto di vista matematico (un'operazione "stateless", senza side effect)

Come abbiamo constatato altre volte, spesso non esiste una soluzione giusta o sbagliata in assoluto.

Le risposte corrette sono: Modifica la lista passata come argomento, Lascia inalterata la lista passata come argomento e restituisce una lista nuova

[Domanda #1]

Sia $f(n) = 7n^3 + 12n \log_5 n + 3 \log n + 23$

Selezionare tutte le risposte corrette (possono essere più di una)

Le risposte corrette sono: $f(n) = \Omega(1)$, $f(n) = \Omega(n)$

Sia $f(n) = 7n^3 + 12n \log_5 n + 3 \log n + 23$

Risposta corretta.

Le risposte corrette sono: $f(n) = \Omega(\log n)$, $f(n) = \Omega(n \log n)$

Sia $f(n) = 7n^3 + 12n \log_5 n + 3 \log n + 23$

Le risposte corrette sono:

$f(n) = O(n^3)$,

$f(n) = \Omega(n^3)$,

$f(n) = \Theta(n^3)$,

$f(n) = O(n!)$

Analisi di complessità delle operazioni su liste, Struttura dati: liste semplici

Dimensione della lista: n

Nel contesto indicato sopra, qual è la complessità, nel caso migliore e peggiore, di
addFront: $\text{Elem} \times \text{List} \rightarrow \text{List}$

La risposta corretta è: $\Theta(1)$ nel caso migliore e peggiore

Analisi di complessità delle operazioni su liste, Struttura dati: liste semplici

Dimensione della lista: n

La posizione di Elem appartiene a \mathbb{N} , insieme dei numeri naturali

Nel contesto indicato sopra, qual è la complessità, nel caso migliore e peggiore, di
removePos: $\mathbb{N} \times \text{List} \rightarrow \text{List}$

Attenzione: \mathbb{N} non è la dimensione della lista ma è l'insieme dei numeri naturali, da cui si prende un valore che indica la posizione di Elem

La risposta corretta è: Theta (1) nel caso migliore, Theta (n) nel caso peggiore

Analisi di complessità delle operazioni su liste, Struttura dati: liste semplici

Dimensione della lista: n

La posizione di Elem appartiene a N, insieme dei numeri naturali

Nel contesto indicato sopra, qual è la complessità, nel caso migliore e peggiore, di set: N x Elem x List → List

Attenzione: N non è la dimensione della lista ma è l'insieme dei numeri naturali, da cui si prende un valore che indica la posizione di Elem

La risposta corretta è: Theta(1) nel caso migliore, Theta(n) nel caso peggiore

Analisi di complessità delle operazioni su liste, Struttura dati: liste semplici

Dimensione della lista: n

Nel contesto indicato sopra, qual è la complessità, nel caso migliore e peggiore, di addBack: Elem x List → List

La risposta corretta è: Theta (n) nel caso migliore e peggiore

Analisi di complessità delle operazioni su liste, Struttura dati: liste semplici

Dimensione della lista: n

N indica l'insieme dei numeri naturali ed è il codominio di size

Nel contesto indicato sopra, qual è la complessità, nel caso migliore e peggiore, di size: List → N

La risposta corretta è: Theta (n) nel caso migliore e peggiore

Analisi di complessità delle operazioni su liste, Struttura dati: liste semplici

Nel contesto indicato sopra, qual è la complessità, nel caso migliore e peggiore, di `emptyList:→List`

La risposta corretta è: Theta (1) nel caso migliore e peggiore

Analisi di complessità delle operazioni su liste, Struttura dati: liste semplici

Dimensione della lista: n

La posizione di `Elem` appartiene a `N`, insieme dei numeri naturali

Nel contesto indicato sopra, qual è la complessità, nel caso migliore e peggiore, di `add: N × Elem × List → List`

Attenzione: `N` non è la dimensione della lista ma è l'insieme dei numeri naturali, da cui si prende un valore che indica la posizione di `Elem`

La risposta corretta è: Theta (1) nel caso migliore; Theta (n) nel caso peggiore

Analisi di complessità delle operazioni su liste, Struttura dati: liste semplici

Dimensione della lista: n

Nel contesto indicato sopra, qual è la complessità, nel caso migliore e peggiore, di `isEmpty: List → Bool`

La risposta corretta è: Theta (1) nel caso migliore e peggiore

Analisi di complessità delle operazioni su liste, Struttura dati: liste semplici

Dimensione della lista: n

La posizione di Elem appartiene a N, insieme dei numeri naturali

Nel contesto indicato sopra, qual è la complessità, nel caso migliore e peggiore, di get: N x List → Elem

Attenzione: N non è la dimensione della lista ma è l'insieme dei numeri naturali, da cui si prende un valore che indica la posizione di Elem

La risposta corretta è:

Theta (1) (se la posizione di passata a get == 0) nel caso migliore;

Theta (n) (se la posizione passata a get >= n) nel caso peggiore

Analisi di complessità delle operazioni su liste, Struttura dati: liste doppiamente collegate, circolari e con sentinella. Dimensione della lista: n. La posizione di Elem appartiene a N, insieme dei numeri naturali. Nel contesto indicato sopra, qual è la complessità, nel caso migliore e peggiore, di get: N x List → Elem. Attenzione: N non è la dimensione della lista ma è l'insieme dei numeri naturali, da cui si prende un valore che indica la posizione di Elem

Theta (1) (se la posizione di passata a get == 0) nel caso migliore; Theta (n) (se la posizione passata a get >= n) nel caso peggiore.

Analisi di complessità delle operazioni su liste, Struttura dati: liste doppiamente collegate, circolari e con sentinella. Dimensione della lista: n. Nel contesto indicato sopra, qual è la complessità, nel caso migliore e peggiore, di isEmpty: List → Bool.

Theta (1) nel caso migliore e peggiore

Analisi di complessità delle operazioni su liste, Struttura dati: liste doppiamente collegate, circolari e con sentinella

Dimensione della lista: n

La posizione di Elem appartiene a N, insieme dei numeri naturali

Nel contesto indicato sopra, qual è la complessità, nel caso migliore e peggiore, di removePos: N x List → List

Attenzione: N non è la dimensione della lista ma è l'insieme dei numeri naturali, da cui si prende un valore che indica la posizione di Elem

Theta (1) nel caso migliore, Theta (n) nel caso peggiore.

Analisi di complessità delle operazioni su liste, Struttura dati: liste doppiamente collegate, circolari e con sentinella

Dimensione della lista: n

La posizione di Elem appartiene a N, insieme dei numeri naturali

Nel contesto indicato sopra, qual è la complessità, nel caso migliore e peggiore, di add: N x Elem x List → List

Attenzione: N non è la dimensione della lista ma è l'insieme dei numeri naturali, da cui si prende un valore che indica la posizione di Elem

Theta (1) nel caso migliore; Theta (n) nel caso peggiore

Analisi di complessità delle operazioni su liste, Struttura dati: liste doppiamente collegate, circolari e con sentinella

Dimensione della lista: n

Nel contesto indicato sopra, qual è la complessità, nel caso migliore e peggiore, di addFront: Elem x List → List

Theta (1) nel caso migliore e peggiore

Analisi di complessità delle operazioni su liste, Struttura dati: liste doppiamente collegate, circolari e con sentinella

Dimensione della lista: n

N indica l'insieme dei numeri naturali ed è il codominio di size

Nel contesto indicato sopra, qual è la complessità, nel caso migliore e peggiore, di size: List → N

Theta (n) nel caso migliore e peggiore

Analisi di complessità delle operazioni su liste, Struttura dati: liste doppiamente collegate, circolari e con sentinella

Dimensione della lista: n

La posizione di Elem appartiene a N, insieme dei numeri naturali

Nel contesto indicato sopra, qual è la complessità, nel caso migliore e peggiore, di set: N x Elem x List → List

Attenzione: N non è la dimensione della lista ma è l'insieme dei numeri naturali, da cui si prende un valore che indica la posizione di Elem

Theta(1) nel caso migliore, Theta(n) nel caso peggiore

Analisi di complessità delle operazioni su liste, Struttura dati: liste doppiamente collegate, circolari e con sentinella

Nel contesto indicato sopra, qual è la complessità, nel caso migliore e peggiore, di emptyList:→List

Theta (1) nel caso migliore e peggiore

Analisi di complessità delle operazioni su liste, Struttura dati: liste doppiamente collegate, circolari e con sentinella

Dimensione della lista: n

Nel contesto indicato sopra, qual è la complessità, nel caso migliore e peggiore, di addBack: Elem x List → List

Theta (1) nel caso migliore e peggiore

31/03

Come è organizzato molto spesso (ad esempio, nel codice relativo al TDD List) il main fornito dai docenti?

Con un certo numero di "registri" che possono contenere dati diversi del TDD implementato (ad esempio, una List in ogni registro), per permettere di condurre una sperimentazione accurata ed esaustiva senza dover rilanciare l'applicazione più volte, perdendo quindi i dati salvati tra un'esecuzione e l'altra e Offrendo all'utente un menu di opzioni corrispondenti alle operazioni implementate dal TDD

Per quale ragione i docenti non includono le funzioni di lettura (da file o standard input) e scrittura (su file o standard output) tra le funzioni che caratterizzano un namespace associato a un TDD?

Per ragioni logiche: le funzioni di lettura e scrittura non vengono facilmente descritte come operazioni in un'algebra eterogenea (che corrisponde al concetto di TDD), in quanto questo richiederebbe di specificare nell'algebra anche i tipi "file", "stream"; si

preferisce mantenere una corrispondenza 1-1 tra i prototipi definiti nel namespace e le operazioni che più caratterizzano il TDD

Quale delle seguenti affermazioni sulla struttura dati "array dinamico con size, maxsize e ridimensionamento" implementata dai docenti e resa disponibile durante il corso è corretta (possono esserci più risposte corrette)

I vector del C++ sono implementati in modo molto simile agli array dinamici con size, maxsize e ridimensionamento

Il ridimensionamento dell'array dinamico può avvenire quando si chiama la funzione add e quando si chiama la funzione removePos (in questo caso può avere un ridimensionamento per diminuire la dimensione dell'array, non per aumentarla)

maxsize indica la dimensione totale attualmente allocata per l'array, size indica il numero di elementi della lista (size può essere minore di maxsize, non è detto che tutto lo spazio disponibile sia usato)

Slide ASD Unite