

Appunti Algoritmi e Strutture Di Dati

by *Manai Monica, Perazzona Diego, Picciau Daniele, Fiori Andrea*

Indice

1 Introduzione	6
1.1 Problemi computazionali e algoritmi	6
1.1.1 Esempi di problemi computazionali e algoritmi	6
1.1.2 Come valutare l'algoritmo scelto	7
1.1.3 Complessità di un algoritmo	8
1.2 Strutture dati	8
1.2.1 Tipologie di strutture dati	9
1.2.2 Le sequenze	9
1.2.3 Gli Insiemi	10
1.2.4 I dizionari	11
1.3 I puntatori	12
1.3.1 Puntatori nulli	12
2 Iterazione, induzione e ricorsione	14
2.1 Iterazione	14
2.2 Induzione	14
2.2.1 Induzione completa	15
2.3 Ricorsione	16
2.4 Invarianti dei cicli	16
2.4.1 Invariante del ciclo <code>while</code> VS invariante del ciclo <code>for</code>	17
2.4.2 Invarianti dei cicli - Esempio pratico con il <code>selectionSort()</code>	19
3 Le liste	22
3.1 Liste concatenate	23
3.1.1 Inserimento in testa	23
3.1.2 Inserimento generico	24
3.1.3 Cancellazione elemento corrente	26
3.1.4 Creazione lista	28
3.1.5 Cancellazione intera lista	28
3.1.6 Attraversamento con funzione	29
3.1.7 Ricerca elemento con funzione	30

4 Alberi	31
4.1 Terminologia	32
4.2 Alberi binari	32
4.2.1 Creazione di un albero binario	33
4.2.2 Costruzione di un albero binario	33
4.3 Le visite	34
4.3.1 Visita in preorder (DFS)	35
4.3.2 Visita in postorder (DFS)	36
4.3.3 Visita inorder (DFS)	38
4.3.4 Attraversamento di un albero binario in C	39
4.3.5 Cancellazione di un albero	40
4.4 Alberi binari di ricerca (BST)	42
4.4.1 Implementazione della ricerca	43
4.4.2 Creazione di un nuovo nodo	45
4.4.3 Inserimento di un nodo	46
4.4.4 Ricerca del massimo e del minimo	48
4.4.5 Ricerca del successore-predecessore	51
4.4.6 Cancellazione di un nodo	54
4.4.7 Alberi BST: alcune osservazioni	57
4.4.8 Alberi BST: Alberi Adelson-Velsky and Landis (AVL)	58
5 Analisi della complessità degli algoritmi	61
5.1 Complessità e dimensione dell'input	61
5.2 Definizione di tempo e modello di calcolo	62
5.2.1 Random Access Machine (RAM)	62
5.3 Valutazione del caso pessimo, medio e ottimo	63
5.3.1 Tempo di calcolo della funzione <code>min()</code> (iterativa)	63
5.3.2 Tempo di calcolo della funzione <code>binarySearch()</code> (ricorsiva)	64
5.4 Ordini di complessità	65
5.4.1 Principali classi di efficienza asintotiche	66
5.4.2 Notazioni asintotiche	66
5.5 Le ricorrenze	67
5.5.1 Metodo di sostituzione (o per tentativi)	68
5.5.2 Metodo dell'esperto	68
5.5.3 Metodo dell'albero di ricorsione (analisi per livelli)	69
5.6 Ordinamento	70
5.6.1 Selection Sort	71
5.6.2 Insertion sort	72
5.6.3 Merge sort	74
6 Standard Template Library (STL)	78
6.1 Container	79
6.2 Sequence container (Contenitori Sequenziali)	79
6.2.1 Vector	79
6.2.2 List	82
6.2.3 Deque	83

6.2.4	Altri container	84
6.2.5	Associative container	84
6.3	Iteratori	85
6.4	Algoritmi della STL	90
6.4.1	Intervalli	91
6.4.2	Predicati	92
7	Divide-et-impera	94
7.1	Risoluzione problemi	94
7.1.1	Classificazione dei problemi	94
7.1.2	Definizione matematica del problema	94
7.1.3	Tecnica di soluzione dei problemi	94
7.2	Divide-et-impera	96
7.3	Minimo: Divide-et-impera	96
7.4	Esempio binary search	98
7.5	Quicksort (Hoare, 1961)	99
7.6	Pivot	100
7.7	Quicksort: procedura principale	101
7.7.1	Caso pessimo	101
7.7.2	caso ottimo	102
7.8	Moltiplicazione di matrici	102
7.8.1	Algoritmo di Strassen	103
7.9	Conclusioni	103
8	I Grafi	104
8.1	Introduzione	104
8.1.1	Definizioni utili	104
8.1.2	Esempi	105
8.1.3	Cammini e raggiungibilità	105
8.1.4	Connessione	106
8.1.5	Dimensioni del grafo	108
8.1.6	Grafi sparsi e densi	108
8.1.7	Isomorfismo	108
8.1.8	Relazioni tra alberi e grafi	110
8.2	Specifiche	111
8.2.1	Grafi dinamici	111
8.2.2	Specifiche ridotte - senza rimozioni	111
8.3	Memorizzazione	111
8.3.1	Matrici di adiacenza	112
8.3.2	Liste di adiacenza	114
8.3.3	Implementazione	116
8.3.4	Iterazione su nodi e archi	117
8.4	Visite	117
8.4.1	Visita: più complessa di quanto sembri	118
8.4.2	Algoritmo generico di attraversamento	120
8.5	BFS - Breadth-First Search	120

8.5.1	Algoritmo BFS	121
8.5.2	Numero di Erdős	121
8.5.3	Albero BFS	124
8.5.4	Complessità BFS	126
8.6	DFS - Depth-First Search	127
8.6.1	BFS vs DFS	127
8.6.2	Iterativa, stack esplicito, pre-order	127
8.6.3	DFS e componenti connesse	127
8.6.4	DFS e grafi non orientati aciclici	127
8.6.5	DFS e grafi orientati aciclici (DAG)	127
8.6.6	Schema	127
8.6.7	Classificazione degli archi	127
8.7	Ordinamento topologico	127
9	Analisi ammortizzata	128
9.1	Metodi per eseguire l'analisi	128
9.2	Esempio: Contatore binario	128
9.3	Metodo dell'aggregazione	129
9.4	Metodo degli accantonamenti	130
9.5	Metodo del potenziale	133
9.6	Analisi ammortizzata e strutture dati: Array VS Liste	134
9.6.1	Array dinamici	134
9.6.2	Array dinamici - Espansione	135
9.6.3	Analisi ammortizzata - Raddoppiamento del vettore	135
9.6.4	Analisi ammortizzata - Incremento del vettore	136
10	Programmazione Dinamica	138
10.1	Idea generale	138
10.1.1	Fasi della risoluzione del problema	138
10.2	Esempio - Coefficiente binomiale	138
10.3	Quando usare la programmazione dinamica	139
10.4	String Match Approssimato	140
10.4.1	Esempio	140
10.4.2	4 possibilità:	140
10.4.3	Esempio 1	142
10.4.4	Esempio 2	143
10.4.5	Esempio 3	144
10.4.6	Esempio 4	144
10.4.7	Algoritmo	145
10.4.8	Reality check	145
10.5	Insieme indipendente di intervalli pestai	145
10.5.1	Esempio: insieme indipendente di peso massimo	146
10.5.2	Pre-elaborazione	146
10.5.3	Individuazione sottostruttura ottima (forma ricorsiva)	148
10.5.4	Algoritmo iterativo	149
10.5.5	Costo computazionale	149

10.6 Zaino (Knapsack)	150
10.6.1 Definizione matematica della soluzione	150
10.6.2 Parte ricorsiva	150
10.6.3 Casi base	151
10.6.4 Zaino algoritmo	152
10.6.5 Esempio	153
10.6.6 Complessità computazionale	153
10.7 Memoization - Introduzione	153
10.7.1 Zaino con Memoization	153
10.7.2 Algoritmo zaino con Memoization	154
10.7.3 Complessità	154
10.7.4 Zaino Memoization: utilizzo tabella come cache	154
10.7.5 Dizionario vs Tabella	155
10.7.6 Approccio generale	156
10.7.7 Riassunto: programmazione dinamica/memoization	156
11 Hashing	157
11.1 Tabelle ad accesso diretto	158
11.1.1 Funzioni hash perfette	158
11.2 Minimizzare le collisioni	159
11.3 Come realizzare una funzione hash	160
11.3.1 Metodo dell'estrazione	161
11.3.2 Metodo dello XOR	161
11.3.3 Metodo della divisione	162
11.3.4 Metodo della moltiplicazione (Knuth)	164
11.4 Gestione delle collisioni	165
11.4.1 Liste di trabocco (concatenamento o chaining)	166
11.4.2 Liste di trabocco: analisi della complessità	167
11.4.3 Indirizzamento aperto (memorizzazione interna)	168
11.4.4 Indirizzamento aperto: analisi della complessità	169
11.4.5 Indirizzamento aperto: ispezione lineare	171
11.4.6 Indirizzamento aperto: ispezione quadratica	171
11.4.7 Indirizzamento aperto: doppio hashing	172
11.4.8 Indirizzamento aperto: cancellazione	172
11.5 Implementazione dell'hashing doppio	173
11.5.1 Definizione delle variabili	173
11.5.2 La funzione hash	174
11.5.3 La funzione scan	174
11.5.4 Le funzioni lookup e insert	175
11.5.5 La funzione remove	176
11.6 Complessità	176
11.6.1 Ristrutturazione	177

1 Introduzione

La parola **algoritmo**, un tempo utilizzata quasi esclusivamente da matematici e informatici, è oggi sempre più diffusa anche nel linguaggio comune.

Spesso viene impiegata in modo **vago o impreciso** per riferirsi a qualunque forma di automazione, decisione informatica o comportamento opaco dei **sistemi digitali**.

Oltre a definire genericamente sistemi informatici, capita spesso che il termine algoritmo venga **usato per qualunque descrizione** di un procedimento che **risolva un problema**. Dunque, diamo delle definizioni che colleghino i concetti di algoritmo e problema computazionale.

1.1 Problemi computazionali e algoritmi

Cos'è un problema computazionale?

Dati un dominio di input e un dominio di output, un **problema computazionale** è rappresentato dalla **relazione matematica** che associa ogni elemento del dominio in input ad uno o più elementi del dominio di output.

Esempio: Immaginiamo di dover seguire una ricetta. L'input è dato dagli ingredienti, l'output è dato dal piatto cucinato, mentre il sistema formale di calcolo è dato dal cuoco.

Cos'è un algoritmo?

Dato un problema computazionale, un algoritmo è un procedimento **effettivo** (di calcolo), espresso tramite un insieme di **passi elementari ben specificati** in un sistema formale di calcolo, che risolve il problema in un **tempo finito**.

L'espressione "**in un tempo finito**" implica che l'algoritmo deve terminare dopo un numero definito di passi, mentre "**passi elementari ben specificati**" si riferisce a operazioni descritte con precisione, realizzabili da un esecutore automatico.

1.1.1 Esempi di problemi computazionali e algoritmi

Per comprendere al meglio la differenza tra problema computazionale e algoritmo si considerino i seguenti problemi:

Problema del minimo

Il minimo di un insieme S è l'elemento di S che è minore o uguale ad ogni elemento di S .

$$\min(S) = a \Leftrightarrow \exists a \in S : \forall b \in S : a \leq b$$

Problema di ricerca

Sia $S = S_1, S_2, \dots, S_n$ una sequenza di dati ordinati e distinti, dove $S_1 \leq S_2 \leq \dots \leq S_n$. Eseguire una ricerca dalla posizione di un dato v in S consiste nel restituire l'indice corrispondente, se v è presente, oppure -1 se v non è presente.

$$lookup(S, v) = \begin{cases} i & \exists i \in 0, \dots, n - 1 : S_i = v \\ -1 & \text{altrimenti} \end{cases}$$

Dunque, esiste una distinzione ben precisa fra *il problema computazionale* che si vuole risolvere e gli *algoritmi* che lo risolvono.

Mentre un problema computazionale specifica quale relazione si desideri tra l'ingresso e l'uscita (cioè il risultato), l'algoritmo descrive la sequenza di azioni da eseguire per ottenere l'uscita desiderata a partire dall'ingresso.

In questo caso, dei possibili algoritmi che risolvono i problemi, sono i seguenti:

- **Problema del minimo:** Per trovare il minimo di un insieme, confronta ogni elemento con tutti gli altri; l'elemento che è minore di tutti è il minimo.
- **Problema di ricerca:** Per trovare un valore v nella sequenza S , confronta v con tutti gli elementi di S , in sequenza, e restituisci la posizione corrispondente; restituisci -1 se nessuno degli elementi corrisponde.

Gli algoritmi che vengono scelti per la risoluzione di un problema computazionale devono presentare **passi non ambigui** ed **eseguibili**, proprio per questo un generico algoritmo è caratterizzato dalle seguenti **proprietà**:

- **Atomicità:** l'algoritmo deve essere composto da passi elementari non scomponibili;
- **Non ambiguità:** l'algoritmo non deve generare errori;
- **Attualità:** l'algoritmo deve essere eseguibile;
- **Finitezza:** si intende che l'algoritmo deve terminare in un tempo finito;
- **Effettività:** l'algoritmo deve portare ad un risultato univoco.

1.1.2 Come valutare l'algoritmo scelto

Valutazione di un algoritmo

La valutazione di un algoritmo consiste nello stabilire se quest'ultimo risolve il problema in modo **efficiente** e **corretto**.

Per quanto riguarda il grado di **efficienza**:

- Alcuni problemi **non possono** essere risolti in modo **efficiente**, quindi non esistono algoritmi noti che risolvano il problema.
- Allo stesso modo, esistono soluzioni "ottime" che non possono essere portate ad un grado di efficienza maggiore proprio perché non è possibile essere più efficienti;
- Altri problemi presentano, banalmente, delle soluzioni che funzionano ma non sono particolarmente efficienti;

Invece, la **correttezza di un algoritmo** viene dimostrata tramite:

- Per quanto possibile, con una descrizione matematica;
- Oppure utilizzando una descrizione "informale".

N.B: alcuni problemi non possono essere risolti. Proprio perché alcuni problemi non possono essere risolti in maniera efficiente e corretta, vengono risolti in maniera approssimata.

1.1.3 Complessità di un algoritmo

La ricerca di un algoritmo efficiente e corretto, porta alla definizione di un ulteriore concetto, quello della "complessità di un algoritmo".

Complessità di un algoritmo

La **complessità di un algoritmo** è data dall'analisi delle risorse da esso impiegate per risolvere un problema, in funzione della dimensione e della tipologia dell'input.

Quando si parla di *risorse impiegate da un algoritmo*, ci si riferisce a:

- **Tempo**: ovvero il tempo impiegato per completare l'algoritmo.

Misurare il tempo di esecuzione di un algoritmo considerando il tempo di calcolo in senso assoluto (secondi, minuti, ecc.) rappresenta un **approccio errato** perché dipende da troppi parametri a seconda della piattaforma utilizzata per la misura:

- Frequenza del processore;
- Linguaggio di programmazione utilizzato;
- Ottimizzazione del compilatore utilizzato;
- Interazione tra memoria primaria (anche memorie cache) e secondaria;
- Velocità di trasmissione dati del bus;
- Processi attualmente in esecuzione;
- ecc ...

Proprio per questo motivo quando si parla di *tempo impiegato dall'algoritmo* ci si riferisce al **numero di operazioni rilevanti**, ovvero il numero di operazioni che caratterizzano lo scopo dell'algoritmo, come ad esempio, contare operazioni elementari come i confronti, che rappresentano una misura più stabile e indipendente dalla piattaforma (capitolo 5.2).

- **Spazio**: Quantità di memoria utilizzata;
- **Banda**: quantità di bit spediti.

1.2 Strutture dati

In un linguaggio di programmazione, un **dato** è un valore che una variabile può assumere.

Cos'è una struttura dati?

Una **struttura dati** è un modo per organizzare e memorizzare i dati sui supporti fisici.

Le strutture dati offrono il **vantaggio** di permettere una **buona organizzazione** dei dati permettendo così di **semplificare l'accesso** e la **modifica** degli stessi, influendo in modo diretto sull'efficienza dell'algoritmo.

Dunque, la caratteristica principale non è tanto il *tipo dei dati*, contenuti all'interno della struttura, ma il **modo in cui viene organizzata la collezione**.

Possiamo definire una struttura dati utilizzando **due elementi**:

- **Insieme di operatori**: ovvero l'insieme di operatori che permettono di manipolare la struttura (inserimenti, eliminazioni, ricerca, ordinamento, ecc...) ;
- **Modo sistematico di organizzare i dati**: indica il modo in cui i dati sono memorizzati e collegati tra loro (ad esempio, in modo sequenziale, a grafo o ad albero).

È importante notare che **non esiste una struttura dati adatta a qualsiasi compito**, quindi risulta utile considerare e conoscere i **vantaggi e svantaggi** di diverse strutture.

1.2.1 Tipologie di strutture dati

Le strutture dati possono essere classificate in base a diversi criteri:

- **Linearie e non linearie:** nelle strutture linearie gli elementi sono **disposti in sequenza** (ad esempio, array o liste), mentre nelle strutture non linearie ogni elemento **può collegarsi a più elementi** (ad esempio alberi o grafi);
- **Statiche e dinamiche:** nelle strutture statiche la **dimensione è fissa** (come ad esempio negli array), mentre nelle strutture dinamiche la dimensione **può variare nel tempo** (ad esempio liste collegate);
- **Omogenee e disomogenee:** le strutture omogenee presentano una collezione di **dati dello stesso tipo** (come ad esempio un array di interi), invece, le strutture disomogenee presentano collezioni di **dati di tipologie differenti**.

1.2.2 Le sequenze

Cos'è una sequenza?

Una **sequenza**, è una struttura dati **dinamica e lineare** che rappresenta una sequenza ordinata di valori, all'interno della quale uno stesso valore può **comparire anche più di una volta**.

L'**ordine** della collezione di dati all'interno della sequenza è **importante**, di tipo **posizionale**. Data una generica sequenza $S = S_1, S_2, \dots, S_n$ è possibile aggiungere o togliere elementi, mantenendo la struttura ordinata della sequenza: per indicare un generico elemento S_i della sequenza si utilizza il parametro pos_i , che rappresenta una **posizione logica** nella sequenza; dunque, è possibile accedere direttamente alla testa con pos_0 e alla coda con pos_n .

Alcuni esempi: di seguito verranno illustrate alcune operazioni effettuate sulle sequenze.

SEQUENCE

% Restituisce **true** se la sequenza è vuota
boolean isEmpty()

% Restituisce **true** se p è uguale a pos_0 oppure a pos_{n+1}
boolean finished(Pos p)

% Restituisce la posizione del primo elemento
Pos head()

% Restituisce la posizione dell'ultimo elemento
Pos tail()

% Restituisce la posizione dell'elemento che segue p
Pos next(Pos p)

% Restituisce la posizione dell'elemento che precede p
Pos prev(Pos p)

SEQUENCE (continua)

% Inserisce l'elemento v di tipo ITEM nella posizione p .
% Restituisce la posizione del nuovo elemento, che diviene il predecessore di
 p

Pos insert(Pos p , ITEM v)

% Rimuove l'elemento contenuto nella posizione p .
% Restituisce la posizione del successore di p ,
% che diviene successore del predecessore di p

Pos remove(Pos p)

% Legge l'elemento di tipo ITEM contenuto nella posizione p

ITEM read(Pos p)

% Scrive l'elemento v di tipo ITEM nella posizione p

write(Pos p , ITEM v)

Esempio di creazione di una sequenza il linguaggio C++

```
std::list<int> lista;  
lista.push_front(2);  
lista.push_front(1);  
lista.push_back(3);  
  
Result: [1, 2, 3]
```

1.2.3 Gli Insiemi

Cos'è un insieme?

Un **insieme** è una struttura dati **dinamica e non lineare** che memorizza una collezione **non ordinata** di elementi senza valori ripetuti.

A differenza delle sequenze, per ordinare un insieme, bisogna introdurre il concetto di **relazione d'ordine**. Infatti, nelle sequenze (come array o liste), l'ordine degli elementi non dipende **dal loro valore**, ma dalla **posizione che occupano**. Quindi non serve definire una "relazione d'ordine" tra gli elementi: la struttura stessa impone un ordine.

Gli insiemi, invece, sono per definizione una collezione di elementi unici ma senza un ordine posizionale. Quindi, in un **insieme**, l'ordinamento fra elementi è dato dall'**eventuale relazione d'ordine definita sul tipo degli elementi stessi**. Per questo motivo, se vogliamo ordinare gli elementi, è necessario specificare in che modo come avviene il confronto.

Esempio: nel caso degli **interi**, il confronto è immediato, poiché esiste un ordine naturale tra i valori numerici ($1 < 2 < 3 < \dots$). Per le **stringhe**, invece, l'ordinamento si basa sull'**ordine lessicografico**, ossia sul confronto dei caratteri secondo la sequenza alfabetica (ad esempio, "cane" precede "gatto" perché la lettera "c" viene prima della "g" nell'alfabeto).

Lavorando con gli insiemi, le operazioni ammesse sulle collezioni di dati sono le seguenti:

- **Operazioni base:** inserimento, cancellazione, verifica contenimento;
- **Operazioni di ordinamento:** massimo e minimo;
- **Operazioni di insiemistiche:** unione, intersezione, differenza;
- **Iteratori:** $\text{foreach } x \in S \text{ do}$

SET

% Restituisce la cardinalità dell'insieme
int = ize()

% Restituisce **true** se *x* è contenuto nell'insieme
boolean contains(ITEM *x*)

% Inserisce *x* nell'insieme, se non già presente
insert(ITEM *x*)

% Rimuove *x* dall'insieme, se presente
remove(ITEM *x*)

% Restituisce un nuovo insieme che è l'unione di *A* e *B*
SET union(SET *A*, SET *B*)

% Restituisce un nuovo insieme che è l'intersezione di *A* e *B*
SET intersection(SET *A*, SET *B*)

% Restituisce un nuovo insieme che è la differenza di *A* e *B*
SET difference(SET *A*, SET *B*)

Esempio di creazione di un insieme il linguaggio C++

```
std::set<std::string> frutta;
frutta.insert("mele");
frutta.insert("pere");
frutta.insert("banane");
frutta.insert("mele");
frutta.remove("mele");
Result: ["banane", "pere"]
```

1.2.4 I dizionari

Cos'è un dizionario?

Un **dizionario** è una struttura dati che rappresenta il concetto matematico di relazione univoca $R : D \rightarrow C$, o associazione chiave valore, dove:

- L'insieme *D* è il dominio (elementi detti chiavi);
- L'insieme *C* è il codominio (elementi detti valori)

Con questa tipologia di struttura dati, sono ammesse le seguenti operazioni:

- Ottenere il valore associato ad una particolare chiave (se presente), o **nil (null)** se assente;
- Inserire una nuova associazione chiave-valore, cancellando eventuali associazioni precedenti per la stessa chiave;
- Rimuovere unassociazione chiave-valore esistente;

DICTIONARY

% Restituisce il valore associato alla chiave *k* se presente, **nil**

altrimenti

ITEM lookup(ITEM *k*)

% Associa il valore *v* alla chiave *k*

insert(ITEM *k*, ITEM *v*)

% Rimuove l'associazione della chiave *k*

remove(ITEM *k*)

1.3 I puntatori

Cos'è un puntatore?

Un **puntatore** è una variabile che contiene un **indirizzo di memoria**. Spesso, l'indirizzo è la locazione di memoria di un'altra variabile.

N.B: un puntatore non può contenere un valore che non sia un indirizzo.

Gli usi tipici dei puntatori sono la creazione di strutture dati collegate come alberi e liste, la gestione di oggetti allocati dinamicamente e come parametri di funzioni.

Ogni puntatore ha un tipo associato. La differenza non sta nella rappresentazione dei puntatori, ma nel tipo dell'oggetto puntato. Le variabili dei puntatori devono essere dichiarate come tali:

```
int *p; //Puntatore a intero  
float *p; //Puntatore a float
```

Gli operatori speciali che vengono utilizzati con i puntatori sono * e &:

- & → è un operatore **unario** che restituisce l'**indirizzo di memoria** del suo operando;

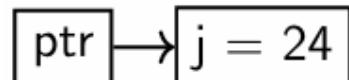
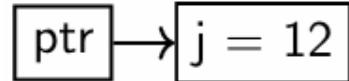
```
balptr = &balance //mette in balptr l'indirizzo di memoria di balance.
```

- * → è un operatore **unario** che restituisce il **valore** della variabile allocata all'indirizzo specificato dal suo operando.

```
value = *balptr //se balptr contiene l'indirizzo di balance, il valore della variabile balance viene messo in value.
```

Esempio: esercizio con puntatori

```
#include <iostream>  
int main() {  
    int j = 12;  
    int *ptr = &j;  
  
    cout << *ptr << endl;  
  
    j = 24;  
    cout << *ptr << endl;  
    cout << ptr << endl;  
    return 0;  
}
```



L'output del programma è il seguente:
12
24
0x7b03a928 (indirizzo di memoria)

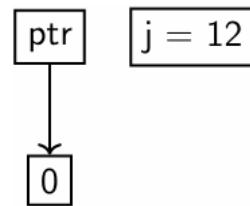
1.3.1 Puntatori nulli

Cos'è un puntatore?

Un puntatore può contenere il valore 0 a indicare che non punta ad alcun oggetto.
In questo caso viene detto **puntatore nullo**.

In questo caso, provando a stampare un puntatore nullo, si ottiene un errore.

```
#include <iostream>
int main() {
    int j = 12;
    int *ptr = 0;
    cout << *ptr << endl; // crash !
    return 0;
}
```



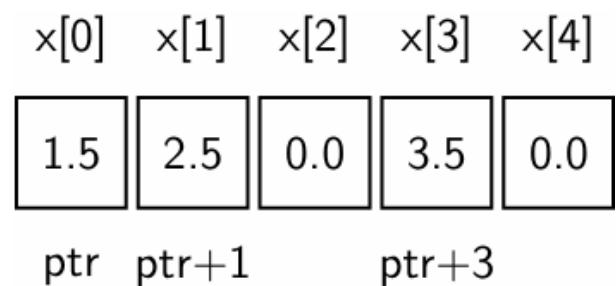
Segmentation violation (core dumped)!

Esempio: come i puntatori vengono utilizzati con gli array

```
//Alcune implementazioni di array in c++
int x[] = { 1, 2, 3, 4 };
char[] t = { 'C', 'i', 'a', 'o', '\0' };
char[] s = "Ciao";
int m[2][3] = { {11, 12, 13}, {21, 22, 23} };
```

```
int main() {
    float x[5];
    int j;

    for (j = 0; j < 5; j++) {
        x[j] = 0;
    }
    float *ptr = x;
    *ptr = 1.5; // x[0] = 1.5
    *(ptr+1) = 2.5; // x[1] = 2.5
    *(ptr+3) = 3.5; // x[3] = 3.5
}
```



Utilizzando gli array, non è necessario specificare che `ptr` punti ad un indirizzo, come fatto precedentemente (`*ptr = &j`), proprio perché `x` rappresenta di per sé l'indirizzo della prima posizione dell'array.

2 Iterazione, induzione e ricorsione

Iterazione, induzione e ricorsione sono concetti fondamentali che compaiono in varie forme nel modelli dei dati, nelle strutture dati e negli algoritmi.

2.1 Iterazione

Cosa si intende con iterazione?

Iterare significa eseguire in modo ripetitivo lo stesso compito, o versioni diverse dello stesso compito, fino al verificarsi di certe condizioni logiche.

Nell'informatica l'iterazione è un concetto che si trova in molte forme:

- **Nei modelli di dati:** concetti come le **liste** sono definiti in modo ripetitivo.
Esempio: Una lista è vuota, oppure è un elemento seguito da un altro, seguito da un altro ancora...;
- **I programmi e gli algoritmi:** utilizzano l'iterazione per eseguire compiti ripetitivi senza specificare uno per uno un gran numero di singoli passi;
- **I linguaggi di programmazione:** utilizzano, nella realizzazione di algoritmi iterativi, costrutti ciclici, come ad esempio i comandi `while` e `for` del `c++`;

2.2 Induzione

L'**induzione** è un concetto strettamente collegato alla ricorsione, ma appartiene più al **mondo matematico** che a quello della programmazione.

Nell'induzione, si dimostra una proposizione per il **caso base**, e poi si mostra che, se vale per un caso generico, allora vale anche per il successivo.

Cosa si intende con induzione?

La dimostrazione per induzione è una tecnica utile per dimostrare la verità di un **asserto**.

Cos'è un asserto?

Si definisce **asserto** (o proposizione) un'affermazione dotata di valore di verità (ossia può essere vera o falsa).

Esempio: in una dimostrazione induttiva si tenta di dimostrare che $S(n)$ vale per tutti gli interi di n non negativi o, più in generale, per tutti gli interi maggiori di un certo limite inferiore.

Dimostrare la verità di un asserto permette di esprimere le proprietà di un programma.

In particolare, la dimostrazione per induzione si basa sulla **definizione di una classe di oggetti (o fatti) strettamente correlati tra loro**.

Esempio: sia $S(n)$ un asserto arbitrario su un intero n . Nella sua forma più semplice (**induzione semplice**), una dimostrazione induttiva dell'asserto $S(n)$ prevede **due passi**:

- **Caso base:** si occupa di costruire uno o più oggetti semplici.
Nel caso dell'esempio, si dimostra che l'asserto $S(n)$ è vero per un valore particolare di n ;
- **Passo induttivo:** costruisce oggetti più grandi che dipendono da quelli appena precedenti.
Nel caso dell'esempio, si dimostra che per ogni $n \geq 0$, se $S(n)$ è vero, lo è anche $S(n + 1)$.



In una dimostrazione induttiva, **ogni istanza dell'asserto $S(n)$ dipende solo dall'asserto sul valore che precede n** . Se l'induzione parte da 0, per ogni intero n si deve dimostrare un asserto $S(n)$:

- La dimostrazione di $S(1)$ utilizza $S(0)$
- La dimostrazione di $S(2)$ utilizza $S(1)$
- e così via...

Ogni asserto dipende dal precedente in modo uniforme, e grazie alla dimostrazione del **passo induttivo** si garantisce la verità di tutti i passi successivi.

2.2.1 Induzione completa

Un'induzione nella quale si dimostra la verità di $S(n + 1)$ utilizzando come ipotesi induttiva soltanto $S(n)$ viene detta **induzione semplice**.

Cosa si intende con induzione completa?

Si parla di induzione completa (perfetta o forte) quando per dimostrare l'asserto S siamo autorizzati a utilizzare $S(i)$ per tutti i valori di i dalla base fino a n .

Quindi a differenza dell'induzione semplice, dove, per dimostrare $S(n + 1)$ si usa solo $S(n)$ (ovvero **solo ciò che si è ottenuto nel passo precedente**), nell'induzione completa per dimostrare che $S(n + 1)$ è vera, **si possono usare tutti i casi precedenti, non solo quello immediatamente precedente**.

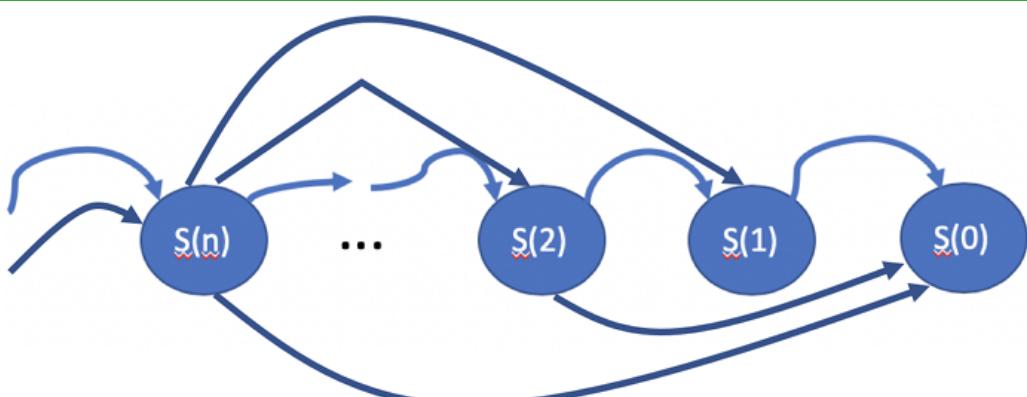
Anche in questo caso, per ottenere una dimostrazione induttiva dell'asserto $S(n)$ si utilizzano due passi:

- Si dimostra prima la base, $S(0)$;
- In seguito, si assumono le verità di $S(0), S(1), \dots, S(n)$ e a partire da questi si dimostra $S(n + 1)$.

Utilità dell'induzione completa

L'induzione completa, torna utile perché alcuni problemi non possono essere dimostrati basandosi solo sul passo precedente.

Ad esempio, se per calcolare $S(n)$ si ha bisogno sia di $S(n - 1)$ che di $S(n - 2)$ (come nella sequenza di fibonacci), allora serve induzione completa.



2.3 Ricorsione

Cosa si intende con ricorsione?

La **ricorsione** è una tecnica in cui un concetto viene definito, direttamente o indirettamente, in termini di se stesso.

È molto simile all'iterazione, ma invece di ripetere istruzioni tramite un ciclo (`for`, `while`), **si ripete richiamando la stessa funzione**. Anche se all'**apparenza** possono sembrare **più complessi** dei programmi iterativi, i programmi ricorsivi possono **risultare più semplici** da scrivere e analizzare.

Una funzione o procedura ricorsiva P può richiamare se stessa in due modi:

- **Direttamente**: quando dentro il corpo della funzione P c'è una chiamata a P;

```
void P() { P(); } // chiamata diretta
```

- **Indirettamente**: quando P chiama un'altra funzione Q, e poi Q chiama P, e così via ...

```
void P() { Q(); } // P chiama Q, che chiama P  
void Q() { P(); }
```

Induttività della ricorsione

Utilizzando la ricorsione si costruisce, **implicitamente**, una **definizione induttiva** di come funziona l'algoritmo e di **quanto tempo impiega** per completarsi.

Infatti quando un algoritmo impiega la ricorsione, si fa uso di una formula detta **equazione di ricorrenza**: il tempo che serve per risolvere un problema più grande dipende dal tempo che serve per risolvere i problemi più piccoli.

Esempio: si immagini di voler calcolare il fattoriale di 4

```
fatt(4)  
-> fatt(3)  
-> fatt(2)  
-> fatt(1)  
-> fatt(0)
```

Quindi il tempo totale per `fatt(4)` è la somma del tempo per `fatt(3)`, più un piccolo tempo extra per l'operazione `n * ...`

In generale, la ricorsione è **induttiva** perché per dimostrare una proprietà di una procedura ricorsiva, abbiamo bisogno di dimostrare un asserto sull'effetto della chiamata di questa procedura, costruendo un **caso base** e i casi successivi a partire da esso.

In queste dimostrazioni si procede spesso per **induzione sulla dimensione dell'argomento**, cioè sulla grandezza del parametro su cui agisce la ricorsione, che diminuisce a ogni chiamata fino al caso base (nel caso di $n!$, il parametro è n).

2.4 Invarianti dei cicli

Cos'è un invarianto?

Un **invariante** è una condizione sempre vera in un certo punto del programma.

Cos'è un invariante di ciclo (o asserzione induttiva)?

Invece, possiamo definire **invariante di ciclo** una condizione sempre vera all'inizio dell'iterazione di un ciclo. Gli invarianti di ciclo sono importanti per dimostrare la correttezza di **algoritmi iterativi**.

Più formalmente possiamo dire che: *Un invariante di ciclo è un asserto S che è vero ogni volta che ci si trova in un particolare punto del ciclo.*

L'asserto *S* viene poi dimostrato per induzione su un **parametro** che costituisce una **misura del numero di volte che il ciclo viene intrapreso**. Questo parametro può essere:

- Il numero di volte che abbiamo raggiunto la guardia di un ciclo;
- Oppure, il valore della variabile indice utilizzata dal ciclo stesso.

La dimostrazione segue quindi gli stessi passaggi di una dimostrazione induttiva:

- **Inizializzazione** (caso base): la condizione è vera alla prima iterazione di un ciclo;
- **Conservazione** (passo induttivo): se la condizione è vera prima di un'iterazione del ciclo, allora rimane vera al termine (quindi, prima della successiva iterazione);
- **Conclusione**: tutto ciò ha senso se, al termine, l'invariante rappresenta quello che voglio ottenere, quindi la "correttezza" dell'algoritmo.

2.4.1 Invariante del ciclo while VS invariante del ciclo for

In genere le dimostrazioni di correttezza per induzione usano il numero di iterazioni del ciclo per cui l'invariante vale. Quando la condizione diviene falsa, possiamo quindi utilizzare contemporaneamente l'invariante del ciclo e la falsità della condizione per concludere qualcosa di interessante su ciò che vale al termine del ciclo.

Utilizzare una tipologia di costrutto iterativo, rispetto ad un'altra può rendere una dimostrazione di correttezza di un ciclo più complessa. Ad esempio:

```
for (i = 0; i < n; i++) {  
    ...  
    ...  
}
```

Utilizzando un ciclo *for* il contatore è integrato nella struttura stessa del ciclo.

Il valore iniziale, la condizione di uscita e l'aggiornamento (*i++*) sono tutti dichiarati in modo esplicito e standard. Si sa con certezza

che il ciclo verrà eseguito un numero finito di volte (*n*), a meno che il corpo del ciclo non modifichi *i* in modo anomalo. Quindi, la terminazione è ovvia: quando *i* = *n*, il ciclo si ferma.

Invece, la struttura del *while* è più **generale e felssibile**.

Nel primo caso si usa un contatore *i* come nel *for*, quindi il comportamento è analogo, **ma il linguaggio non obbliga a farlo!**

Primo caso

```
i = 0;  
while (i < n) {  
    ...  
    i++;  
}
```

Secondo caso

```
x = 5;  
while (x != 0) {  
    x = f(x); // f potrebbe non  
              diminuire x!  
}
```

Nel secondo caso non c'è un contatore "ovvio" che cresce o diminuisce in modo regolare, e non

è certo che il ciclo finirà (magari $f(x)$ restituisce sempre un numero diverso da 0, o l'utente non scrive mai "exit").

Proprio per questo, **parte della dimostrazione di correttezza di un ciclo while consiste nel dimostrarne la terminazione**. Solitamente, la dimostrazione di terminazione viene fatta determinando **un'espressione E** , che coinvolge le variabili del programma stesso, tale che:

- Il valore di E **diminuisce (o aumenta)** di almeno un'unità a **ogni iterazione del ciclo**;
- Quando il ciclo si ferma (quindi la **condizione è falsa**), il valore di E è **pari ad una costante minima (o massima) prefissata**.

Dunque, basandosi sulle caratteristiche appena descritte, è possibile determinare l'espressione E , ad esempio, con la seguente dimostrazione.

Esempio: dimostrazione della terminazione di un ciclo while

```
integer Fattoriale(integer n)
1 integer i, fatt
2 i  $\leftarrow$  2
3 fatt  $\leftarrow$  1
4 while i  $\leq$  n do
5   fatt  $\leftarrow$  fatt * i
6   i  $\leftarrow$  i + 1
7 return fatt
```

La seguente funzione "Fattoriale" calcola $n!$ passato da parametro, assumendo che $n \geq 1$. Lo scopo è quello di dimostrare che il ciclo while (*righe 4-6*) deve terminare. Come detto precedentemente, per dimostrare la terminazione del ciclo bisogna determinare E . Dunque, si cerca una grandezza che:

- parta positiva;
- diminuisca ad ogni iterazione;
- diventi negativa quando il ciclo finisce.

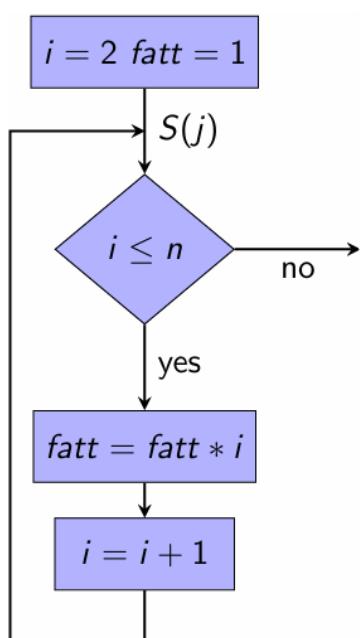
N.B.: Per dimostrare la terminazione **si preferisce trovare una funzione E che decresce verso un limite inferiore** (è più facile da gestire nei teoremi di terminazione).

Una scelta che risulta naturale è $E = n - 1$, poiché:

- i è piccolo all'inizio, e dato che deve essere $i \leq n \Rightarrow n - 1$ è positivo;
- Ad ogni iterazione i aumenta di 1 $\Rightarrow E$ diminuisce di 1;
- Quando $i = n + 1 \Rightarrow i > n \Rightarrow E = n - (n + 1) = -1$, cioè diventa negativo proprio quando il ciclo finisce.

A questo punto, dopo aver dimostrato la terminazione del ciclo while, è possibile dimostrare il funzionamento del codice tramite induzione.

Come definito in precedenza, l'invariante del ciclo è dato da un asserto, in questo caso $S(j)$.



Cosa afferma l'asserto?

Se si raggiunge il controllo del ciclo $i \leq n$ quando la variabile i ha un certo valore j , allora il valore della variabile fatt è $(j-1)!$!

In altre parole, prima di ogni iterazione, fatt contiene il fattoriale del numero precedente a i . Quindi:

- quando $i = 2$, fatt = $1!$;
- quando $i = 3$, fatt = $2!$;
- ecc...

Dimostriamo che l'asserto vale sia per il caso base che per il passo induttivo:

- Caso base:** nel caso base, prima del ciclo, $fatt = 1$ e i ha un valore $j = 2$. Dunque $fatt = 1 = (j - 1) = 2 - 1 = 1$ e la **base è dimostrata**.
- Passo induttivo:** nel passo induttivo, assumendo che $S(j)$ sia vera per un generico valore $j \leq n$, è stato dimostrato che la variabile $fatt$, prima della nuova iterazione, sia $fatt = (j - 1)!$; Se si vuole dimostrare che anche $S(j + 1)$, bisogna dimostrare che la variabile $fatt$, prima della nuova iterazione, sia $fatt = (j + 1 - 1)! = j!$; Si distinguono due casi:
 - Quando i ha valore $j > n$: il ciclo è già finito (la guardia non si attiva più), e dunque il valore di $fatt$, sarà il valore finale, ovvero $j!$;
 - Quando i ha valore $j \leq n$: Dopo aver verificato che per $S(j)$ vale $fatt(j - 1)!$ (con $i = j$) si esegue un'altra iterazione del ciclo.
 Dunque, in *riga 5* troviamo che $fatt = fatt \times i = (j - 1)! \times j$ che equivale alla definizione di $j!$, mentre in *riga 6* si ha $i = i + 1 = j + 1$. Ora al prossimo controllo della prossima iterazione si avrà $fatt = (j - 1)! \times j = j!$ e $i = j + 1$ che confermano l'asserzione $S(j + 1)$.

2.4.2 Invarianti dei cicli - Esempio pratico con il `selectionSort()`

Obiettivo del `selectionSort()`

Il `selectionSort()` prevede l'**ordinamento di un vettore** di n elementi i quali **vengono permutati** in modo che essi compaiano in un ordine non decrescente.

Possiamo descrivere il funzionamento del selection sort tramite il seguente pseudocodice:

`SelectionSort(ITEM[]A, integer n)`

```

1 integer  $i, j, p$ 
2 for  $i \leftarrow 1$  to  $n - 1$  do
3    $p \leftarrow i$ 
4   for  $j \leftarrow i + 1$  to  $n$  do
5     if  $A[j] < A[p]$  then
6        $p \leftarrow j$ 
7   % A questo punto,  $p$  è l'indice del primo elemento più piccolo in
      %  $A[i \dots n]$ 
8   % Scambiamo allora  $A[p]$  e  $A[i]$ 
9    $A[p] \leftrightarrow A[i]$ 

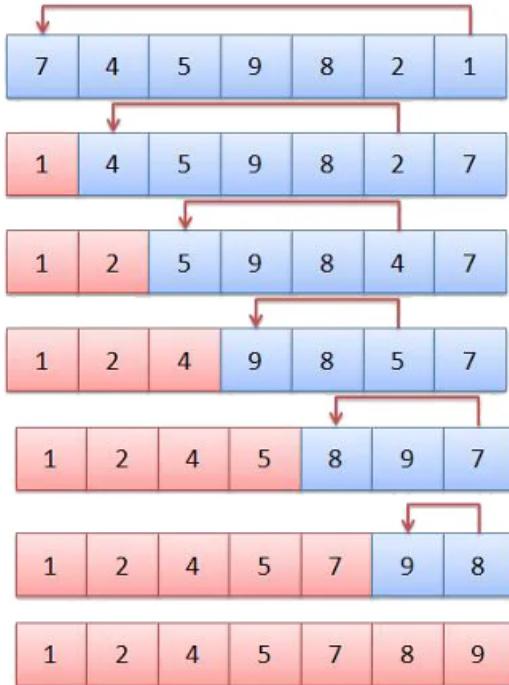
```

In particolare, come si può vedere in figura (a), l'ordinamento avviene nel seguente modo:

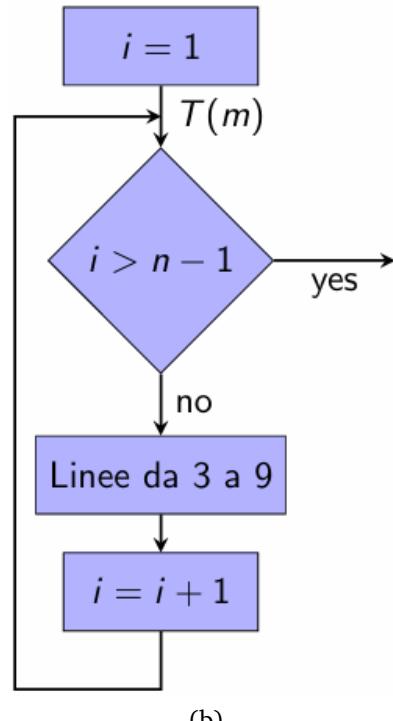
1. Nella prima iterazione troviamo (selezioniamo) uno degli elementi più piccoli tra tutti i valori in $A[1 \dots n]$ e lo scambiamo con $A[1]$;
2. Nella seconda iterazione troviamo uno degli elementi più piccoli tra tutti i valori in $A[2 \dots n]$ e lo scambiamo con $A[2]$;
3. Dopo l'iterazione i -esima, $A[1 \dots i]$ contiene gli i elementi più piccoli ordinati in modo non decrescente;

Come si può vedere dall'immagine (b), l'asserzione induttiva (o invariante di ciclo) è chiamata $T(m)$. Quest'ultima, come detto al capitolo 2.4, è una condizione che deve essere vera immediatamente prima del controllo di terminazione del ciclo, cioè prima di controllare $i > n - 1$.

Quando i ha un certo valore m , sono già stati selezionati gli $m - 1$ elementi più piccoli e ordinati nella posizione iniziale dell'array. In altre parole:



(a)



(b)

- La parte sinistra dell'array $A[1\dots(m-1)]$ è già ordinata e contiene gli elementi più piccoli;
- La parte destra $A[m\dots n]$ è ancora da ordinare.

Questa è la proprietà $T(m)$, che **rimane vera ad ogni iterazione del ciclo** ($i = m$), quindi è proprio l'**invariante di ciclo**.

Esempio: dimostrazione tramite induzione

Come detto al capitolo 2.4, per poter dimostrare un asserto tramite induzione, possiamo utilizzare valore della variabile indice del ciclo stesso, in questo caso m .

Dunque, si dice che: **si può dimostrare per induzione su m l'asserto $T(m)$.**

Cosa afferma l'asserto?

Se si raggiunge il controllo del ciclo $i > n - 1$ alla (linea 2) con $i = m$, allora:

- Gli elementi $A[1\dots(m-1)]$ sono ordinati in senso non decrescente:

$$A[1] \leq A[2] \leq \dots \leq A[m-1];$$
- Tutti gli elementi di $A[m\dots n]$ sono maggiori o uguali a ogni elemento di $A[1\dots(m-1)]$.

1. **Caso base:** quando $i = m = 1$ siamo all'inizio dell'algoritmo, e non abbiamo ancora ordinato nulla. Per vedere se $T(1)$ è vera si controlla che l'asserto sia corretto:
 - La parte (1) di $T(1)$ dice che $A[1\dots m-1] = A[1\dots 0]$ è ordinato, ma $A[1\dots 0]$ è vuoto, quindi è banalmente vero (una sequenza vuota è sempre ordinata).
 - La parte (2) di $T(2)$ dice che tutti gli elementi in $A[m\dots n] = A[1\dots n]$ sono maggiori o uguali a quelli in $A[1\dots m-1] = A[1\dots 0]$, ma anche in questo caso gli elementi in $A[1\dots 0]$ non esistono, quindi la condizione è automaticamente vera.

Quindi $T(1)$ è vera \Rightarrow **caso base dimostrato**.

2. **Passo induttivo:** in questo passaggio si assume che $T(m)$ sia vera per un generico $m \leq n-1$, che verifica le condizioni dell'asserto, quindi:
 - La prima parte dell'array $A[1\dots(m-1)]$ è già ordinata;
 - Tutti gli elementi compresi in $A[1\dots(m-1)]$ sono \leq di ogni elemento compreso in $A[m\dots n]$.

Ora si vuole dimostrare che anche $T(m + 1)$ sia vera. Durante l'iterazione con $i = m$:

- il ciclo interno (righe 3-9) **cerca il minimo** tra gli elementi non ordinati $A[m \dots n]$;
- poi **scambia** quel minimo con l'elemento $A[m]$ (quindi prima di tutti gli altri elementi non ordinati).

Anche dopo lo scambio rimane vero che:

- la porzione $A[1 \dots m]$ è ora ordinata (perché il nuovo elemento in $A[m]$ è il più piccolo tra quelli rimanenti);
- tutti gli elementi in $A[(m + 1) \dots n]$ rimangono \geq di quelli precedenti.

Quindi $T(m + 1)$ è vera perché subito dopo lo scambio il ciclo esterno incrementa la variabile i da m a $m + 1$: “*Dato che ora il valore di i è $m + 1$, l'asserzione $T(m + 1)$ risulta vera.*”

3. Conclusione:

nell'ultima iterazione, quando $i = m = n$

- i primi $n - 1$ elementi $A[1 \dots (n - 1)]$ sono già ordinati e al posto giusto;
- l'ultimo elemento $A[n]$ è automaticamente \geq di tutti gli altri.

Quando il programma termina, dunque, gli elementi di A sono in ordine non decrescente, cioè ordinati.

3 Le liste

Cos'è una lista (linked list)?

È una possibile implementazione per la struttura dati astratta, **sequenza**. È costituita da una **sequenza di nodi**, contenenti dati arbitrari e 1/2 puntatori all'elemento successivo e/o precedente.

Contiguità delle liste

È importante specificare che la contiguità degli elementi in una lista **non implica** una loro contiguità nella memoria (*contiguità lista \Rightarrow contiguità in memoria*).

Nelle liste, infatti, i nodi **non sono necessariamente salvati in celle di memoria adiacenti**, come accade invece negli array. Ogni nodo può trovarsi in una posizione qualunque della memoria, e il **collegamento logico** tra di essi è **garantito dai puntatori**, non dalla loro vicinanza fisica.

Esempio: possiamo immaginare che la CPU debba accedere a posizioni di memoria anche molto distanti tra loro per poter percorrere l'intera lista.

Nel contesto delle **linked list**, le operazioni di **inserimento o cancellazione**, nota la posizione, hanno **costo costante**, cioè $O(1)$.

Questo perché vengono utilizzati dei **passaggi fissi**, aggiornando un numero limitato di puntatori tra i nodi, indipendentemente dalla lunghezza della lista.

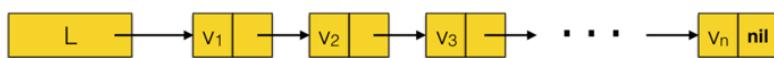
Esempio: immaginiamo la seguente lista $[a] \rightarrow [b] \rightarrow [c]$.

Se volessimo inserire un nuovo nodo x dopo a , sarebbe sufficiente:

1. creare il nodo x ;
2. impostare $x.next = b$;
3. impostare $a.next = x$.

Il numero di operazioni rimane invariato anche se la lista cresce, quindi il costo resta $O(1)$.

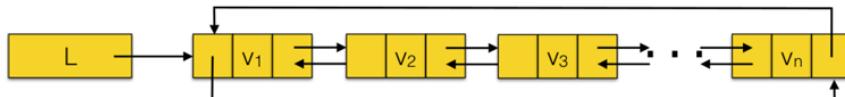
Al contrario, l'accesso a un elemento tramite il suo indice richiede di scorrere la lista dall'inizio fino al nodo desiderato, con un costo $O(n)$.



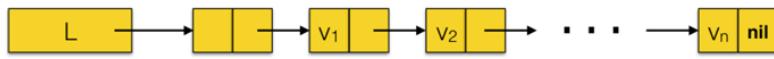
Monodirezionale



Bidirezionale



Bidirezionale circolare



Monodirezionale con sentinella

Le linked list presentano **dive-
rse possibili implemen-
tazioni**:

- Liste **monodirezionali** o **bidirezionali**;
- Liste con **sentinella** e **senza sentinella**;
- Liste **circolari** e **non circo-
lari**.

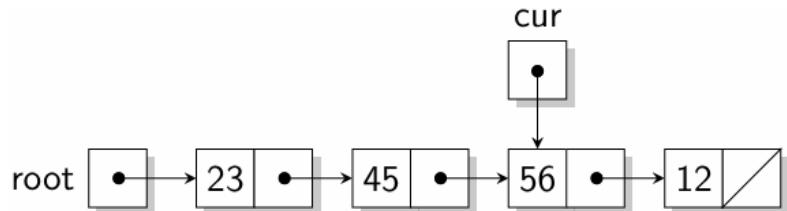
3.1 Liste concatenate

Come sono strutturate?

La **lista singolarmente concatenata** (monodirezionale) è la più semplice struttura dinamica basata su nodi. La lista è definita da un "root (o head) pointer" e sfrutta un **puntatore corrente** per la sua gestione.

Le liste concatenate vengono definite nel seguente modo:

```
struct slist {  
    int dato;  
    struct slist *next;  
};  
  
/* lista vuota */  
struct slist *root = NULL;
```



Come si può vedere, quando viene inizializzato il puntatore root (la testa della lista) a **null**, la lista è vuota.

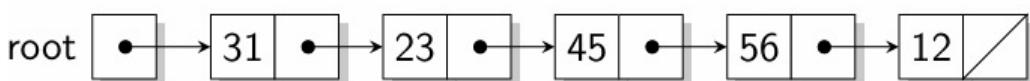
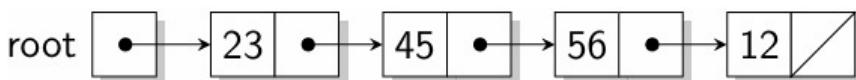
Per manipolare una lista semplicemente concatenata esistono **diverse funzioni**, tra cui:

- Inserimento di un nodo (prima del puntatore corrente);
- Cancellazione di un nodo (indicato dal puntatore corrente);
- Creazione nuova lista;
- Cancellazione lista;
- Attraversamento con esecuzione di funzione;
- Ricerca con esecuzione di funzione;
- Concatenazione;
- Conteggio;
- Inserimento ordinato.

3.1.1 Inserimento in testa

Tramite la seguente funzione è possibile inserire un nuovo elemento in testa alla lista.

```
1 slist *insert(slist *p, int elem) {  
2     slist *q = malloc(sizeof(slist));  
3  
4     if(!q) {  
5         fprintf(stderr, "Allocation error\n");  
6         exit(-1);  
7     }  
8     q->dato = elem;  
9     q->next = p;  
10    return q;  
11}
```



- **Tipo di ritorno:** La funzione restituisce un puntatore a `slist`. Il puntatore restituito sarà la nuova testa della lista;
- **Parametri:** i parametri definiti nella funzione sono (`slist *p, int elem`), rispettivamente il **puntatore corrente** alla testa della lista (che può essere `null` se la lista è vuota) e il **valore da inserire** nel nuovo nodo;
- **Funzionamento del codice:** per poter inserire un nuovo nodo in testa alla lista, supponiamo di possedere una struttura dati come quella definita al capitolo 3.1.
 - *riga 2:* creo un puntatore `*q` al nuovo nodo di grandezza `slist`;
 - *riga 4-7:* eseguo un controllo su `q` per assicurarmi che la `malloc` sia andata a buon fine;
 - *riga 8:* il campo `dato` del nuovo nodo prende il valore di `elem` passato tramite la funzione;
 - *riga 9:* Collega il nuovo nodo al resto della lista: il campo `next` del nuovo nodo punta all'elemento che era in testa (`p`).
 - **Se la lista è vuota** (`p == null`), la nuova lista contiene solo il nuovo nodo `q`, quindi:
`q -> next = null;`
 - **Se la lista non è vuota**, `q` è inserito prima della vecchia testa: `q -> old_head -> ...`
 - *riga 10:* restituisce il puntatore al nuovo nodo: ora `q` è la testa della lista aggiornata

Esempio di utilizzo

```

slist *head = NULL;          // lista vuota
head = insert(head, 10);    // ora head punta al nodo con dato 10
head = insert(head, 5);     // ora head punta al nodo con dato 5; next punta
                           al nodo 10

```

3.1.2 Inserimento generico

Questa funzione è un passo avanti rispetto alla `insert()`. `insert2()` permette un inserimento generico in qualunque posizione della lista (in testa, in mezzo o in coda).

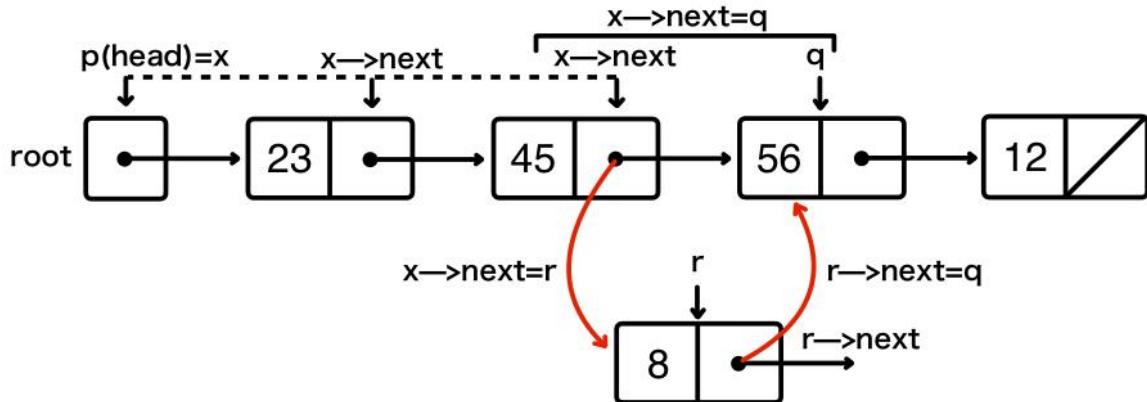
```

1 slist *insert2(slist *p, slist *q, int elem) {
2     slist *x;
3     slist *r = malloc(sizeof(slist));
4     if(!r) {
5         fprintf(stderr, "Allocation error\n");
6         exit(-1);
7     }
8     r->dato = elem;
9     if(p == q) { // in testa
10         r->next=p;
11         return r;
12     }
13     else{ // in mezzo o in coda (se q==NULL)
14         for(x= p; x && x->next != q; x = x->next);
15
16         // prima di collegare mi assicuro che q sia valido
17         if(x && x->next == q) {
18             r->next = q;
19             x->next = r;
20         }
21     }
22 }

```

```

21     return p;
22 }
23 }
```



- **Tipo di ritorno:** La funzione restituisce un puntatore a **slist**, ovvero il puntatore alla testa aggiornata della lista.
- **Parametri:** i parametri definiti nella funzione sono (**slist *p, slist *q, int elem**), rispettivamente:
 - il **puntatore corrente** alla testa della lista (che può essere **null** se la lista è vuota);
 - il **puntatore al nodo di riferimento** (davanti al quale inserire) oppure **null** se si vuole inserire in coda;
 - **l valore da inserire** nel nuovo nodo.
- **Funzionamento del codice:** per poter inserire un nuovo nodo in testa alla lista, supponiamo di possedere una struttura dati come quella definita al capitolo 3.1.
 - **riga 2:** **x** è un puntatore alla lista che serve come supporto per eseguire lo scorrimento della stessa;
 - **riga 3:** creo un puntatore ***r** al nuovo nodo di grandezza **slist**;
 - **riga 4-7:** eseguo un controllo su **r** per assicurarmi che la **malloc** sia andata a buon fine;
 - **riga 8:** inizializzo il campo **dato** del nuovo nodo con il valore del campo **elem** passato tra i parametri;
 - **riga 9-12:** viene **gestito l'inserimento in testa**. Infatti, se il nodo **q** coincide con **p**, mi basta semplicemente collegare il nuovo nodo alla vecchia testa (**r -> next = p**), e poi restituire il nuovo nodo (**return r**);
 - **riga 13-21:** viene **gestito l'inserimento in mezzo o in coda**.

Tramite il ciclo **for** si fa scorrere il puntatore di appoggio (**x**) partendo dalla testa (**p**) della lista. Lo scorrimento continua fintanto che è verificata la condizione **x->next != q**, e **presenta due possibilità di terminazione:**

1. La condizione diventa **x->next==q** poiché **q** è un nodo interno alla lista. In altre parole quando **x** precede il nodo **q** specificato nei parametri, si esce dal ciclo;
2. La condizione non si avvera perché il **q** passato nei parametri è **null**, e il ciclo termina le sue iterazioni.

Dato che lo scorrimento ha due possibili terminazioni, bisognerà effettuare un ulteriore controllo per determinare se si tratta di un inserimento **in mezzo o in coda**.

La condizione dell'**if** indica che **x != null** e che **x->next==q**, dunque:

1. Si ha un **inserimento in mezzo** quando si verifica il caso 1.

In questo caso (**r->next=q**) collega il nuovo nodo al nodo di riferimento, mentre

`(x->next=r)` collega il nodo precedente al nuovo nodo.

2. Si ha un **inserimento in coda** quando si verifica il caso 2. In questo caso `q==null`, per cui `r->next=null` diventando così l'ultimo nodo della lista, mentre il precedente ultimo nodo viene aggiornato con `x->next = r..`

Esempio di utilizzo

```
slist *head = NULL; // lista vuota

// Inserisco alcuni elementi in testa (uso insert2 con q == p)
head = insert2(head, head, 56); // lista: 56
head = insert2(head, head, 45); // lista: 45 -> 56
head = insert2(head, head, 23); // lista: 23 -> 45 -> 56
printList(head);

// Inserimento in mezzo (prima di 56)
slist *q = head->next->next; // q punta al nodo con dato 56
head = insert2(head, q, 8); // inserisco 8 prima di 56
printList(head); // lista: 23 -> 45 -> 8 -> 56

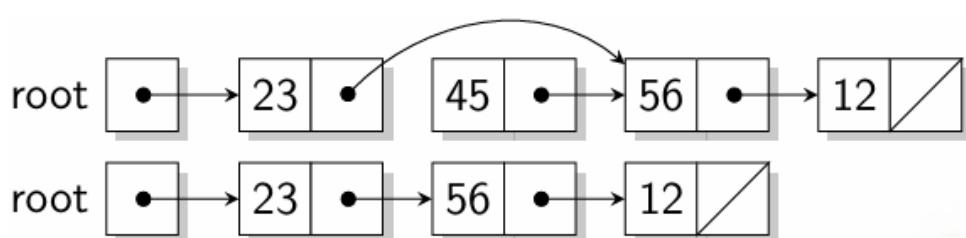
// Inserimento in coda (q == NULL)
head = insert2(head, NULL, 12); // inserisco 12 alla fine
printList(head); // lista: 23 -> 45 -> 8 -> 56 -> 12

return 0;
```

3.1.3 Cancellazione elemento corrente

La funzione `delete()` consente di eliminare un nodo specifico (`q`) da una lista semplicemente concatenata la cui testa è puntata da `p`, assumendo che `q != null`.

```
1 slist *delete(slist *p, slist *q) {
2     slist *r;
3     if(p == q) p = p->next;
4     else {
5         for(r = p; r && r->next != q; r = r->next);
6         if(r && r->next == q)
7             r->next = r->next->next;
8     }
9     free(q);
10    return p;
11 }
```



- **Tipo di ritorno:** la funzione restituisce un puntatore a `slist`, ovvero il puntatore alla testa aggiornata della lista dopo l'eliminazione;
- **Parametri:** i parametri definiti nella funzione sono (`slist *p, slist *q`), rispettivamente **il puntatore corrente** alla testa della lista e **il puntatore al nodo di riferimento** (davanti al quale rimuovere);
- **Funzionamento del codice:** per poter eliminare un nodo all'interno della lista, supponiamo di possedere una struttura dati come quella definita al capitolo 3.1.
 - *riga 2:* dichiarazione di un puntatore ausiliario `r`, che servirà per scorrere la lista.
 - *riga 3:* se `p == q`, significa che il nodo da eliminare è la testa della lista. In questo caso basta avanzare la testa al nodo successivo: `p = p->next;`
 - *riga 4-8:* se il nodo da eliminare non è quello in testa alla lista (`p!=q`), utilizzando il puntatore ausiliario, utilizzo un ciclo per scorrere tutti i nodi, **ricordando che il nodo q specificato nei parametri non è null**.

Proprio per questo motivo, a differenza di quanto detto nel capitolo 3.1.2, ho **un solo caso di terminazione**, e cioè quando `r->next=q`, ovvero il nodo precedente a quello da eliminare.

Trovato il predecessore di `q`, quest'ultimo si salta collegando `r->next` direttamente al nodo successivo a `q`, ovvero `r->next=r->next->next;`

 - *riga 9:* si libera la memoria allocata dal nodo eliminato `q`;
 - *riga 10:* restituzione del puntatore alla testa aggiornato.

Esempio di utilizzo

```

slist *head = NULL;

// Creo una lista: 23 -> 45 -> 56 -> 12
head = insert2(head, head, 56);
head = insert2(head, head, 45);
head = insert2(head, head, 23);
head = insert2(head, NULL, 12);

printList(head); // Output: 23 -> 45 -> 56 -> 12 -> NULL

// Elimino il nodo in testa
head = delete(head, head);
printList(head); // Output: 45 -> 56 -> 12 -> NULL

// Elimino un nodo in mezzo (quello con dato 56)
slist *q = head->next;
head = delete(head, q);
printList(head); // Output: 45 -> 12 -> NULL

// Elimino il nodo in coda
q = head->next;
head = delete(head, q);
printList(head); // Output: 45 -> NULL

return 0;

```

3.1.4 Creazione lista

La funzione `createlist()` crea una lista vuota e ne restituisce il puntatore radice.

```
1 slist *createlist(void) {
2     return NULL;
3 }
```

- **Tipo di ritorno:** la funzione restituisce un puntatore a `slist`, cioè al tipo di dato che rappresenta un nodo della lista;
- **Parametri:** il parametro è vuoto, perché non serve alcuna informazione esterna per creare una lista vuota;
- **Funzionamento del codice:** per poter creare una lista vuota, supponiamo di possedere una struttura dati come quella definita al capitolo 3.1. Il codice si occupa semplicemente di restituire `null`, che rappresenta una lista vuota (assenza di nodi).

Esempio di utilizzo

```
// Creazione di una lista vuota
slist *head = createlist();

// Inserimento di elementi
head = insert2(head, head, 10);
head = insert2(head, head, 20);
head = insert2(head, head, 30);

printList(head); // Output: 10 -> 20 -> 30 -> NULL

return 0;
```

3.1.5 Cancellazione intera lista

La funzione `destroylist()` distrugge una lista, assumendo che `p!=null`.

```
1 void destroylist(slist *p) {
2     while(p = delete(p,p)); // N.B: cancello in testa
3 }
```

- **Tipo di ritorno:** la funzione non restituisce alcun valore, poiché il suo scopo è esclusivamente quello di deallocare la memoria della lista passata come argomento;
- **Parametri:** Il parametro `p` rappresenta il puntatore alla testa della lista che si vuole distruggere;
- **Funzionamento del codice:** all'interno del ciclo `while`, viene eseguita ripetutamente l'istruzione `p = delete(p, p)`.

Ad ogni iterazione viene eliminato il primo nodo della lista, poiché la funzione `delete()` viene chiamata passando due volte lo stesso puntatore (`p, p`), cioè indicando che il nodo da cancellare è proprio la testa.

La funzione `delete()` restituisce il nuovo puntatore alla testa della lista (che è ora il nodo successivo): il ciclo `while` continua fino a quando `delete()` restituisce `null`, cioè finché non rimangono più nodi da cancellare.

Esempio di utilizzo

```
// Creazione di una lista
slist *head = createlist();

// Inserimento di elementi
head = insert2(head, head, 10);
head = insert2(head, head, 20);
head = insert2(head, head, 30);

printList(head); // Output: 10 -> 20 -> 30 -> NULL

destroylist(head); // Distruzione della lista

return 0;
```

3.1.6 Attraversamento con funzione

La funzione traverse() ha lo scopo di scorrere (visitare) tutti i nodi di una lista concatenata ed eseguire, su ciascuno di essi, un'operazione specificata dall'utente.

```
1 void traverse(slist *p, void (*op)(slist *)) {
2     slist *q;
3     for(q = p; q; q = q->next) (*op)(q);
4 }
```

- **Tipo di ritorno:** la funzione non restituisce alcun valore. Il suo scopo è **puramente esecutivo**, applicando una funzione a ogni elemento della lista;
- **Parametri:** i parametri definiti nella funzione sono (`slist *p, void (*op)(slist *)`), rispettivamente il **puntatore alla testa** della lista che si vuole attraversare e un **puntatore a funzione**, ovvero un parametro che rappresenta una funzione da richiamare per ogni nodo visitato. Tale funzione deve avere la stessa firma, cioè ricevere come parametro un puntatore a `slist` e non restituire nulla (tipo `void`);
- **Funzionamento del codice:** per poter scorrere una lista, supponiamo di possedere una struttura dati come quella definita al capitolo 3.1.
 - riga 2: viene dichiarato un puntatore `*q`, che servirà come puntatore di scorrimento.
 - riga 3: inizia il ciclo che scorre l'intera lista partendo dalla testa. Ad ogni iterazione viene chiamata la **funzione passata come parametro** (`op`), applicandola al nodo corrente: `(*op)(q)`. Il ciclo termina quando `q` diventa `null`, cioè quando è stato raggiunto l'ultimo nodo della lista.

```
/* Esempio: funzione op che stampa il contenuto del nodo */
void printelem(slist *q) {
    printf("\t-----\n\t| %5d |\n\t-----\n\t| %c | \n\t---%c---\n\t",
           q->dato,
           q->next ? '.' : 'X',
           q->next ? '|' : '-');
    if(q->next) printf(" | \n\t v\n");
}
```

Esempio di utilizzo

```
// Creazione della lista
slist *head = createlist();
head = insert2(head, head, 10);
head = insert2(head, NULL, 20);
head = insert2(head, NULL, 30);

// Attraversamento della lista e stampa di ciascun elemento
printf("Contenuto della lista:\n");
traverse(head, printelem);

return 0;
```

3.1.7 Ricerca elemento con funzione

4 Alberi

L'albero ordinato (spesso chiamato più semplicemente "albero") è una struttura informativa fondamentale, che si presta a rappresentare svariate situazioni, quali:

- Partizioni successive di un insieme in sottoinsiemi disgiunti;
- Organizzazioni gerarchiche di dati;
- Procedimenti enumerativi o decisionali.

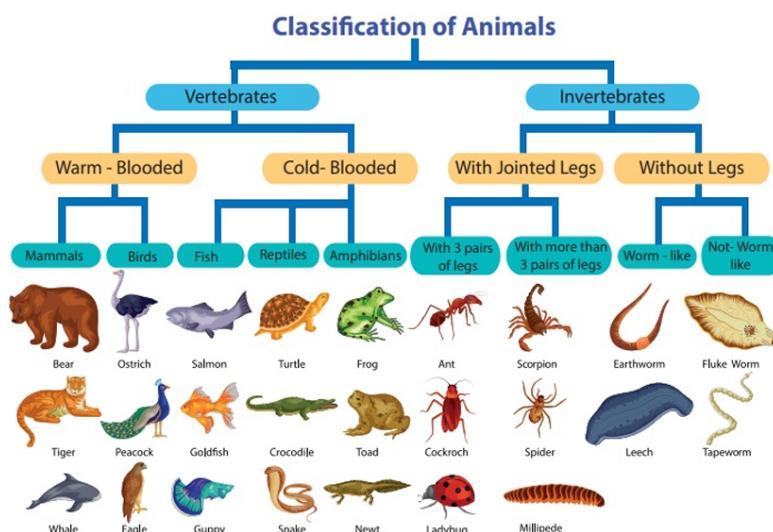
L'implementazione avviene, come per le liste, attraverso l'uso di **strutture autoreferenziali e puntatori**.

Struttura di un albero ordinato (o radicato)

Un **albero ordinato** è dato da un insieme finito di elementi (dunque una **struttura dati statica**) detti **nodi** e un insieme di **archi orientati** che connettono coppie di nodi.

Ogni albero presenta le seguenti proprietà:

- Un nodo dell'albero è designato come **nodo radice**;
- Ogni nodo n , a parte la radice, ha esattamente un **arco entrante**;
- Esiste un **cammino unico** dalla radice a ogni nodo;
- L'albero è **connesso**: non esistono elementi che non fanno parte dell'albero, ovvero che non sono connessi tramite il cammino di archi all'albero stesso.



un'organizzazione in cartelle, partendo dalla radice, per ogni elemento dell'albero, fino ad arrivare ai singoli file.

Dunque, l'albero rappresenta una struttura gerarchica nella quale a partire da un elemento molto grande (come in questo caso il "regno animale") si divide in sottoclassi, fino ad arrivare ai singoli elementi.

Un esempio che si avvicina maggiormente all'ambito dei sistemi operativi, come si può vedere in Figure 23, è un albero del file system (come quello di linux), dove si ha

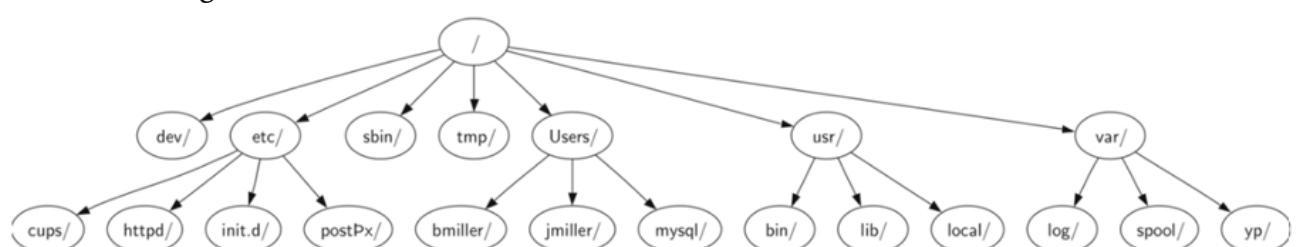


Figure 23: Albero del file system linux

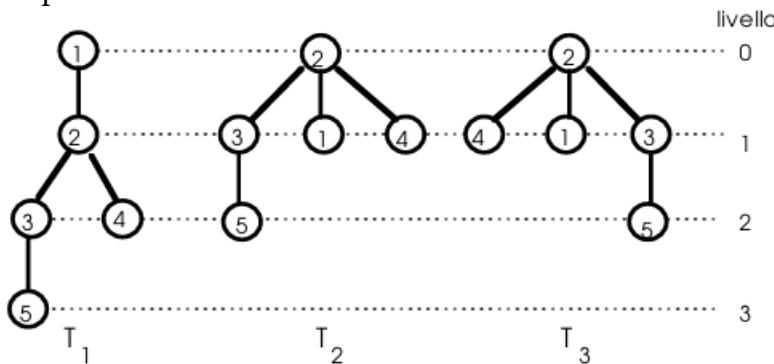
4.1 Terminologia

Sia T un albero ordinato di n nodi, con radice r .

Possiamo definire T_1, \dots, T_k gli insiemi disgiunti e non vuoti in cui sono partizionati tutti i nodi di T , ognuno dei quali avrà come radice un nodo r_1, \dots, r_k .

- Ciascun T_i è detto **sottoalbero** di T , mentre ciascun r_i è detto **figlio** di r ;
- I nodi r_1, \dots, r_k sono tra loro **fratelli**, ed r è il loro **padre**.
- Inoltre, un nodo senza figli è detto **foglia**, mentre la radice dell'albero (**root**) è l'unico **nodo senza padre**.

Oltre alla terminologia che assumono i nodi dell'albero in base alla loro posizione, l'albero in sé presenta alcune caratteristiche:



- **Profondità dei nodi (depth):** profondità del cammino semplice dalla radice al nodo (misurata in numero di archi percorsi). Ad esempio, la radice avrà profondità 0, i figli avranno profondità 1, i figli dei figli avranno profondità 2 e così via...
- **Altezza albero (height):** indica la profondità massima delle sue foglie. Ad esempio, T_1 ha altezza 3, mentre T_2 e T_3 altezza 2;

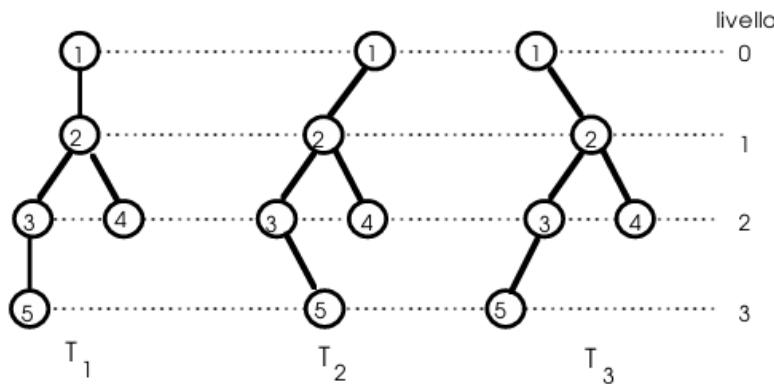
- **Livello (level):** si riferisce all'insieme di nodi alla stessa profondità;

4.2 Alberi binari

Cos'è un'albero binario?

Un albero binario è un particolare albero ordinato in cui ogni nodo ha **al più due figli**, e si fa distinzione tra figlio **sinistro** e figlio **destro**.

N.B: in alcune implementazioni può esserci un puntatore al padre.



La proprietà degli alberi binari è **molto delicata**: infatti, anche se due alberi T_1 e T_2 hanno gli stessi nodi e la stessa radice, essi risultano diversi se in T_1 un nodo u è figlio sinistro di un nodo v , mentre in T_2 lo stesso nodo u è figlio destro di v .

Nell'esempio mostrato, l'albero T_1 presenta al massimo due figli per

ogni nodo, ma la rappresentazione grafica non aiuta a comprendere se si tratta di un figlio destro o sinistro. Al contrario T_2 e T_3 sono alberi binari e distinti, poiché non hanno gli stessi figli destri e sinistri.

4.2.1 Creazione di un albero binario

Come per qualsiasi struttura dati (liste concatenate, doppiamente concatenate, ecc...) anche per poter utilizzare un albero è necessario definire una struttura e inizializzarla nel main:
Struttura dell'albero

```
struct inttree {  
    int data;  
    struct inttree *left, *right;  
};
```

Inizializzazione dell'albero

```
inttree *root = NULL;  
//Puntatore alla struttura
```

Per quanto riguarda la struttura dell'albero:

- **data**: contiene il valore del nodo;
- ***left**: è il puntatore al nodo sinistro;
- ***right**: è il puntatore al nodo destro.

Invece, l'inizializzazione dell'albero è effettuata tramite, **root** rappresenta la radice che all'inizio è impostata a **null** per indicare l'albero vuoto.

Alberi binari e definizione ricorsiva

Gli alberi binari ammettono una **definizione ricorsiva** come insieme finito di nodi, che può essere:

- Un **insieme vuoto**, cioè nessun nodo.
Rappresenta l'albero "vuoto" senza radice e senza figli, come quando viene inizializzato nel main e nessun è stato ancora allocato in memoria (`inttree *root = NULL;`)
- Oppure, un nodo radice con **due sottoalberi binari disgiunti**, uno sinistro e uno destro.

4.2.2 Costruzione di un albero binario

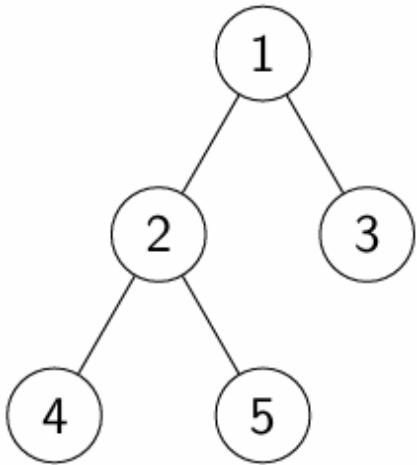
Dopo aver proceduto con la creazione della struttura e l'inizializzazione dell'albero, si procede con la sua creazione, allocando i vari nodi e definendo i puntatori per il figlio destro e sinistro.

```
1 int main(int argc, char *argv[]) {  
2     //Livello 0  
3     inttree *root = NULL;  
4     root = malloc(sizeof(inttree));  
5     root->data = 1;  
6  
7     //Livello 1  
8     root->left = malloc(sizeof(inttree));  
9     root->right = malloc(sizeof(inttree));  
10    root->left->data = 2;  
11    root->right->data = 3;  
12    root->right->left = NULL;  
13    root->right->right = NULL;  
14  
15    //Livello 2  
16    root->left->left = malloc(sizeof(inttree));  
17    root->left->right = malloc(sizeof(inttree));  
18    root->left->left->data = 4;  
19    root->left->right->data = 5;  
20    root->left->left->left = NULL;
```

```

21 root->left->left->right = NULL;
22 root->left->right->left = NULL;
23 root->left->right->right = NULL;
24 }

```



- **Blocco 1:** vengono allocati i nodi al livello 0 dell'albero.
 - riga 2: il puntatore `root` (`*root`), definito in precedenza, viene associato al nodo radice dell'albero, puntando a un'area di memoria allocata dinamicamente della dimensione della struttura `inttree`;
 - riga 3: inizializza il campo `data` del nodo radice a 1
- **Blocco 2:** vengono allocati i nodi al livello 1 dell'albero.
 - riga 6-7: il nodo radice presenta al suo interno due puntatori, `left` e `right`, i quali punteranno a due nodi `inttree` differenti, corrispondenti rispettivamente ai figli sinistro e destro;
 - riga 8-9: viene inizializzato il campo `data` dei figli della radice, assegnando 2 al figlio sinistro e 3 al figlio destro;
 - riga 10-11: come per il nodo radice, anche le strutture dei suoi nodi figli avranno dei puntatori `left` e `right`. In queste due righe i puntatori `left` e `right` del figlio destro vengono impostati a `NULL`, a indicare che esso non ha ulteriori discendenti.
- **Blocco 3:** vengono allocati i nodi al livello 2 dell'albero.
 - riga 13-14: i puntatori `left` e `right` del figlio sinistro della radice vengono associati a due nuovi nodi `inttree`;
 - riga 15-16: viene inizializzato il campo `data` di questi due nodi, assegnando 4 al figlio sinistro e 5 al figlio destro;
 - riga 17-20: i puntatori `left` e `right` di entrambi questi nodi vengono impostati a `NULL`, indicando che non hanno ulteriori figli.

4.3 Le visite

Cos'è una visita?

Una **visita** (o attraversamento) di un albero ordinato è una strategia che consiste nel seguire una "rotta" di viaggio che consente di **analizzare ogni nodo** dell'albero almeno una volta.

Una visita può essere eseguita in **due modi**:

- **Visita in profondità** (Deep-First Search - DFS): in questo caso si segue un percorso radice-foglia, ovvero per visitare un albero **si visita ricorsivamente** ognuno dei suoi **sottoalberi**, andando da sinistra verso destra. Questo tipo di visita presenta **tre varianti** (ordini):
 - Ordine anticipato (**preorder**);
 - Ordine posticipato (**postorder**);
 - Ordine simmetrico (**inorder**).

Un'analisi di questo tipo **richiede uno stack**: si visita l'albero utilizzando uno stack che però viene già reso disponibile tramite il **meccanismo di ricorsione**.

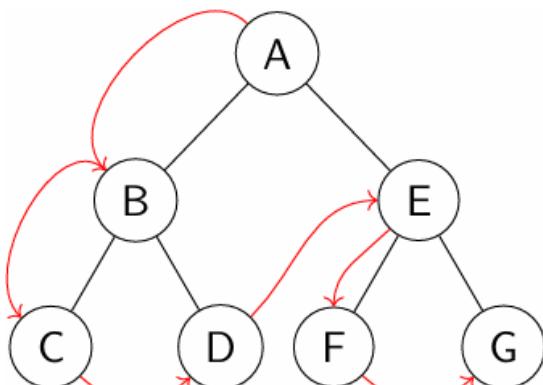
- **Visita in ampiezza** (Breadth First Search - BFS): partendo dalla radice, vengono visitati i nodi livello per livello, “orizzontalmente”, fino a raggiungere il livello massimo dell’albero. Proprio per questo, un analisi di questo tipo viene anche detta *ordine per livelli*.

Sia T un albero non vuoto di radice r . Se r non è una foglia ma ha $k > 0$ figli, indichiamo con T_1, \dots, T_k i k sottoalberi di T radicati nei k figli di r . Gli ordini di visita in profondità sono definiti ricorsivamente come segue:

4.3.1 Visita in preorder (DFS)

Funzionamento della visita in preorder

La **visita in preorder** di T consiste nell’esaminare r e poi nell’effettuare, nell’ordine, la previsita dei sottoalberi T_1, \dots, T_k .



(a) albero attraversato in preorder

dfs(Tree t)

```

1: if  $t \neq \text{nil}$  then
2:   % pre-order visit of  $t$ 
3:   print  $t$ 
4:
5:   dfs( $t.\text{left}()$ )
6:
7:   dfs( $t.\text{right}()$ )
8: end if

```

(b) Pseudocodice visita in preorder

Per capire il funzionamento della visita in preorder, dividiamo il procedimento in alcuni passi, vedendo come evolvono la sequenza di stampa (attraversamento dell’albero), e lo stack di chiamate (ricorsività della funzione `dfs()`):

1. In questo tipo di visita si parte sempre chiamando la funzione rispetto al nodo $A \rightarrow \text{dfs(Tree } A\text{)}$. Pertanto la sequenza di stampa inizia con A , e la chiamata viene memorizzata nello stack delle chiamate.
 - *Sequenza di stampa:* A
 - *Stack chiamate:* A
2. Dopo aver stampato il nodo A , la funzione richiama ricorsivamente `dfs` sul figlio sinistro del nodo specificato nei parametri della funzione `dfs(Tree A)`: succede quindi che la funzione viene rieseguita dall’inizio, usando come riferimento il nodo B , che verrà stampato.
 - *Sequenza di stampa:* $A - B$
 - *Stack chiamate:* $A - B$
3. Analogamente, dopo aver stampato il nodo B , la funzione richiama ricorsivamente la `dfs` sul figlio sinistro del nodo specificato nei parametri della funzione `dfs(Tree B)`, rieseguendo la funzione con C come riferimento.
 - *Sequenza di stampa:* $A - B - C$
 - *Stack chiamate:* $A - B - C$
4. A questo punto, viene chiamata `dfs(t.left())` sul nodo C , ma quest’ultimo è NULL poiché non presenta figli; la stessa cosa succede provando a chiamare `dfs(t.right())`.
 Quando entrambe le chiamate ai figli sinistro e destro terminano, significa che il nodo

in questione ha completato la propria porzione di codice ricorsivo e può andare avanti: andare avanti significa che la chiamata `dfs(t.left())` eseguita per `dfs(Tree B)` è stata completata.

- *Sequenza di stampa*: $A - B - C$
 - *Stack chiamate*: $A - B$

5. Il controllo torna al nodo B che, dopo aver completato la chiamata per il figlio sinistro, eseguirà la chiamata per il figlio destro (D) tramite `dfs(t.right())`.

 - *Sequenza di stampa*: $A - B - C$
 - *Stack chiamate*: $A - B - D$

6. Con la chiamata ricorsiva sul figlio destro di B , la funzione viene rieseguita con D come riferimento `dfs(Tree D)`, e come prima operazione D viene stampato.

 - *Sequenza di stampa*: $A - B - C - D$
 - *Stack chiamate*: $A - B - D$

7. Come nel caso del nodo C , anche D non presenta ulteriori figli, dunque le sue chiamate falliranno (NULL) e di conseguenza D completa la propria porzione di codice: il controllo torna nuovamente a B .

 - *Sequenza di stampa*: $A - B - C - D$
 - *Stack chiamate*: $A - B$

8. Nonostante il controllo sia tornato a B , esso ha completato le sue chiamate per i figli sinistro e destro, quindi ancora una volta il controllo passa al nodo superiore (in questo caso la radice r (A)).

 - *Sequenza di stampa*: $A - B - C - D$
 - *Stack chiamate*: A

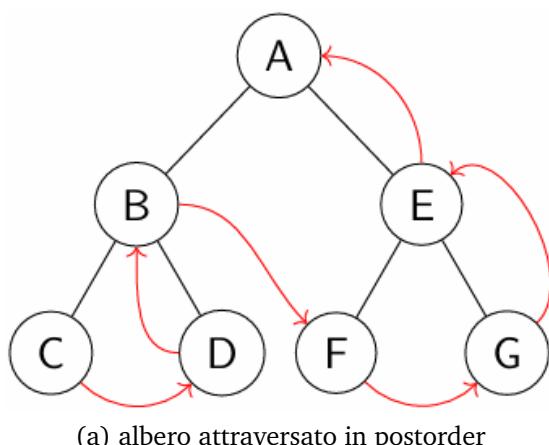
9. Ciò che succede è che in A la chiamata `dfs(t.left())` è stata completata (tutto il sottoalbero sinistro), quindi procedo con la chiama ricorsiva per il nodo destro della radice E : in questo modo **il procedimento si ripete in modo analogo** per tutto il sottoalbero destro.

 - *Sequenza di stampa*: $A - B - C - D - E - \dots$
 - *Stack chiamate*: $A - E - \dots$

4.3.2 Visita in postorder (DFS)

Funzionamento della visita in postorder

La **visita in postorder** di T consiste nell'effettuare, nell'ordine, la postvisita di T_1, \dots, T_k e poi nell'esaminare r .



```
dfs(Tree t)
1: if  $t \neq nil$  then
2:   dfs(t.left())
3:
4:   dfs(t.right())
5:
6:   % post-order visit of  $t$ 
7:   print  $t$ 
8: end if
```

(b) Pseudocode visita in postorder

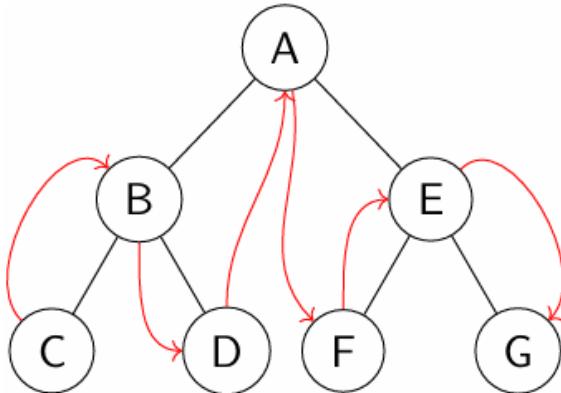
Per quanto riguarda la visita in postorder, anche in questo caso, dividiamo il procedimento in alcuni passi, monitorando l'evoluzione della sequenza di stampa e dello stack di chiamate:

1. Si parte come sempre dal nodo radice $A \rightarrow \text{dfs}(\text{Tree } A)$. Tuttavia, in postorder la **radice non viene stampata subito**: la funzione procede prima a richiamare $\text{dfs}(t.\text{left}())$, dunque $\text{dfs}()$ viene eseguita in modo ricorsivo rispetto al figlio sinistro (B).
 - **Sequenza di stampa:** –
 - **Stack chiamate:** A
2. Essendo nuovamente all'inizio della funzione, B non viene stampato, ma ancora una volta viene richiamata $\text{dfs}(t.\text{left}())$ in modo ricorsivo per il figlio sinistro di B (Ovvero C)
 - **Sequenza di stampa:** –
 - **Stack chiamate:** $A - B$
3. Il figlio sinistro di B è il nodo C , dunque verrà eseguita la funzione $\text{dfs}(\text{Tree } C)$. A questo punto la funzione riparte dall'inizio, ma C non ha ulteriori figli, quindi $\text{dfs}(t.\text{left}())$ e $\text{dfs}(t.\text{right}())$ restituiscono NULL e si passa alla stampa del nodo (C).
 - **Sequenza di stampa:** C
 - **Stack chiamate:** $A - B - C$
4. Il nodo C , dopo la stampa, ha completato la propria porzione di codice, dunque il controllo ritorna al nodo B .
 - **Sequenza di stampa:** C
 - **Stack chiamate:** $A - B$
5. Il nodo B , dopo aver completato la chiamata per il figlio sinistro, può procedere a richiamare la funzione ricorsiva per il figlio destro $\text{dfs}(t.\text{right}())$, ovvero il nodo D : ciò implica $\text{dfs}(\text{Tree } D)$.
 - **Sequenza di stampa:** C
 - **Stack chiamate:** $A - B - D$
6. Anche il nodo D non presenta figli, dunque $\text{dfs}(t.\text{left}())$ e $\text{dfs}(t.\text{right}())$ restituiscono NULL e si passa alla stampa del nodo stesso, con conseguente terminazione della sua porzione di codice. Infine, il controllo ritorna al nodo B .
 - **Sequenza di stampa:** $C - D$
 - **Stack chiamate:** $A - B$
7. A questo punto, il nodo B ha completato la visita dei propri figli sinistro (C) e destro (D). Di conseguenza si va oltre le chiamate $\text{dfs}(t.\text{left}())$ e $\text{dfs}(t.\text{right}())$ e si stampa B . Dopo la stampa di B , si ha la terminazione della sua porzione di codice ricorsivo, e il controllo ritorna al nodo A .
 - **Sequenza di stampa:** $C - D - B$
 - **Stack chiamate:** A
8. Terminata la visita del sottoalbero sinistro di A , la funzione procede ora con il figlio destro, eseguendo $\text{dfs}(t.\text{right}())$ e quindi in modo ricorsivo la funzione $\text{dfs}(\text{Tree } E)$.
 - **Sequenza di stampa:** $C - D - B$
 - **Stack chiamate:** $A - E$
9. Dunque, una volta che $\text{dfs}(\text{Tree } E)$ incomincia le sue chiamate ricorsive, viene **visitato tutto il sottoalbero destro**, con un **procedimento analogo** a quello utilizzato precedentemente per visitare il sottoalbero sinistro.
 - **Sequenza di stampa:** $C - D - B - \dots$
 - **Stack chiamate:** $A - E - \dots$

4.3.3 Visita inorder (DFS)

Funzionamento della visita inorder

Fissato $i \geq 1$, la **visita inorder** di T consiste nell'effettuare la invista di T_1, \dots, T_i , nell'esaminare r , e poi nell'effettuare la invista di $T_{(i+1)}, \dots, T_k$.



(a) albero attraversato inorder

dfs(Tree t)

```

1: if  $t \neq \text{nil}$  then
2:   dfs( $t.\text{left}()$ )
3:
4:   % in-order visit of  $t$ 
5:   print  $t$ 
6:
7:   dfs( $t.\text{right}()$ )
8: end if

```

(b) Pseudocodice visita inorder

Anche in questo caso analizziamo passo per passo il funzionamento della funzione `dfs()` e l'evoluzione della sequenza di stampa e dello stack di chiamate:

1. La visita incomincia chiamando `dfs(Tree A)` sul nodo radice (A). Come per la visita in postorder, la radice non viene subito stampata, ma si procede a richiamare ricorsivamente il suo figlio sinistro (B) tramite `dfs(t.left())`.
 - **Sequenza di stampa:** –
 - **Stack chiamate:** A
2. Quando la funzione `dfs(Tree B)` viene richiamata ricorsivamente sul nodo B , si riparte dall'inizio e nuovamente si chiama in modo ricorsivo il figlio sinistro di B (C).
 - **Sequenza di stampa:** –
 - **Stack chiamate:** $A - B$
3. Una volta che `dfs(Tree C)` è stata chiamata, la chiamata ricorsiva al figlio sinistro di C , all'interno della funzione, restituisce NULL proprio perché il nodo C non ha figli. Dunque si procede con l'operazione subito successiva, la stampa del nodo, per poi trovare un'altra chiamata ricorsiva al figlio destro di C che restituì anch'essa NULL per lo stesso motivo di prima.
 - **Sequenza di stampa:** C
 - **Stack chiamate:** $A - B - C$
4. Arrivati a questo punto la porzione di codice eseguita da C termina e il controllo torna al nodo B .
 - **Sequenza di stampa:** C
 - **Stack chiamate:** $A - B$
5. Quando il nodo B termina la chiamata al figlio sinistro (C), l'operazione successiva è la stampa del nodo stesso. Subito dopo la stampa viene eseguita una chiamata ricorsiva al figlio destro (nodo D).
 - **Sequenza di stampa:** $C - B$
 - **Stack chiamate:** $A - B$

6. Non appena `dfs(Tree D)` è in esecuzione vengono eseguite le istruzioni secondo l'ordine prestabilito: `dfs(t.left())` restituisce NULL (nessun figlio sinistro), *D* viene stampato e `dfs(t.right())`, non avendo figli, restituisce anch'esso NULL.
 - **Sequenza di stampa:** *C – B – D*
 - **Stack chiamate:** *A – B – D*
7. Dopo l'esecuzione di tutta la porzione di codice del nodo *D*, il controllo torna a *B*, ma anche quest'ultimo, avendo visitato figlio sinistro e destro, ha terminato le istruzioni nella sua porzione di codice. Quindi, in cascata, il controllo torna al nodo *A*, che, dopo aver visitato tutto il suo sottoalbero sinistro tramite la chiamata ricorsiva `dfs(t.left())` può essere stampato come prevede l'istruzione successiva della sua porzione di codice.
 - **Sequenza di stampa:** *C – B – D – A*
 - **Stack chiamate:** *A*
8. Infine, dopo la stampa del nodo *A*, viene effettuata una chiamata al sottoalbero destro tramite `dfs(t.right())`. Anche in questo caso, il **procedimento di visita è analogo** a quello utilizzato precedentemente per il **sottoalbero sinistro**.
 - **Sequenza di stampa:** *C – B – D – A – ...*
 - **Stack chiamate:** *A – ...*

4.3.4 Attraversamento di un albero binario in C

La funzione per l'attraversamento di un albero binario fa riferimento alla visita in preorder.

```

1 void preorder(inttree *p, void (*op)(inttree*)) {
2   if(p) {
3     (*op)(p);
4     preorder(p->left, op);
5     preorder(p->right, op);
6   }
7 }
```

- **Tipo di ritorno:** la funzione non restituisce alcun valore, infatti il suo scopo è unicamente quello di visitare i nodi dell'albero;
- **Parametri:** la funzione `preorder` accetta due parametri;
 - `inttree *p`: è un puntatore alla struttura del nodo di un albero, definita al capitolo 4.2.1. Solitamente viene passato il puntatore alla radice del nodo dell'albero, `*root`, definito nel main, per poter iniziare la visita di tutto l'albero. Successivamente, tramite le chiamate ricorsive, il puntatore non farà più riferimento alla radice ma ai vari nodi figli;
 - `void (*op)(inttree*)`: è un puntatore a funzione (`(*op)`), **cioè una funzione passata come argomento**, che accetta come parametro un puntatore alla struttura di un nodo dell'albero (`inttree*`) e non restituisce nulla (`void`).

In questo modo, passando come argomento a `preorder` il nome di una qualsiasi funzione che come argomento ha un puntatore alla struttura di un nodo dell'albero, è possibile eseguire un'operazione specifica all'interno della visita.

Ad esempio, può essere utilizzato per richiamare una funzione che si occupa della stampa dei nodi, così da **tenere traccia della sequenza di attraversamento**.

```

void stampaNodo(inttree *n) {
  printf("%d ", n->data);
```

```
}
```

- **Funzionamento del codice:** il corpo della funzione presenta un controllo condizionale e due chiamate ricorsive per l'esecuzione della visita dell'albero.
 - *riga 2:* viene effettuato un controllo di validità sul nodo che si sta visitando, assicurandosi che esista (cioè che non sia NULL). Questo controllo risulta utile nel momento in cui si raggiunge una foglia, i cui figli sinistro e destro non esistono, per fare in modo di cedere il controllo al nodo successivo;
 - *riga 3:* viene eseguita la funzione passata come parametro; (*op) rappresenta la funzione puntata da op, mentre p è l'argomento (il nodo attualmente visitato) che le viene passato;
 - *riga 4:* viene effettuata una chiamata ricorsiva a preorder, specificando come parametri il figlio sinistro (*p->left*) del nodo corrente e il puntatore alla funzione op.
In questo modo, la visita procede ricorsivamente su tutti i figli sinistri dell'albero;
 - *riga 5:* allo stesso modo, quando il figlio sinistro di un determinato nodo è stato visitato, con *p->right* si passa alla visita ricorsiva del figlio destro dello stesso nodo.

N.B: ovviamente per creare le altre tipologie di visite (postorder e inorder), è sufficiente invertire l'ordine delle istruzioni all'interno della condizione if secondo la metodologia prevista dal tipo di attraversamento.

Esempio di utilizzo

```
int main(int argc, char *argv[]) {
inttree *root = NULL;
root = malloc(sizeof(inttree));

// Allocazione di tutti i nodi e assegnazione del valore per ognuno
// come fatto al capitolo 4.2.2.
// ...
// ...

// Esecuzione della visita preorder
printf("Attraversamento in preorder: ");
preorder(root, stampaNodo);
printf("\n");

return 0;
}
```

4.3.5 Cancellazione di un albero

Per poter effettuare la cancellazione di un albero, la seguente implementazione fa leva sull'utilizzo di una struttura ricorsiva, in modo da **liberare correttamente** tutta la memoria occupata da un albero binario.

```
1 void destroytree (inttree *p) {
2     if (p->left){ // Se esiste un sottoalbero sinistro
3         destroytree (p->left); // Libera il sottoalbero
4     }
```

```

5
6 if (p->right) { // Se esiste un sottoalbero destro
7     destroytree (p->right); // Libera il sottoalbero
8 }
9
10 free (p); / Per finire libera la memoria della radice
11 }
```

- **Tipo di ritorno:** la funzione non restituisce alcun valore, infatti in suo scopo è unicamente quello di **rilasciare la memoria** precedentemente allocata per ciascun nodo dell'albero.
- **Parametri:** la funzione destroytree accetta un singolo parametro, ovvero `inttree *p`. Quest'ultimo è un puntatore alla struttura del nodo dell'albero, definita al capitolo 4.2.1. A partire dal nodo specificato (in genere la radice `root`), la funzione visita ricorsivamente tutti i sottoalberi (sinistro e destro), liberando la memoria associata a ciascun nodo.
- **Funzionamento del codice:** l'idea di base della funzione `destroy` è che "*non posso liberare un nodo finché non ho liberato i figli, perché se liberassi il nodo prima, perderei il puntatore ai figli e non potrei più raggiungerli per dealloca*rl*i*".

Dunque, la sequenza di cancellazione segue l'ordine di **visita postorder** discussa al capitolo 4.3.2, quindi, considerando lo stesso albero, $C - D - B - F - G - E - A$. In accordo con il funzionamento postorder, il codice presenta la **seguente logica**:

- *righe 2-4*: viene effettuato un controllo sul figlio sinistro del nodo passato come parametro. Se esiste, la funzione `destroytree` viene richiamata ricorsivamente fino a raggiungere una foglia (nodo privo di figli), che restituirà un controllo falso.
 - *riga 6-8*: quando il controllo su `p->left` risulta falso, si passa al controllo su `p->right`. Anche in questo caso viene richiamata la funzione `destroytree` in modo ricorsivo fino a che non si arriva ad una foglia, che per definizione non presenta figli.
 - *riga 10*: quando entrambi i controlli risultano falsi, significa che il nodo corrente non ha più figli, quindi può essere deallocato in sicurezza tramite `free(p)`.
- Al termine della liberazione, la funzione termina e il controllo ritorna al nodo superiore che era in attesa della conclusione della chiamata ricorsiva nello **stack delle chiamate**.

Esempio di utilizzo

```

int main(int argc, char *argv[]) {
inttree *root = NULL;
root = malloc(sizeof(inttree));

// Allocazione di tutti i nodi e assegnazione del valore per ognuno
// come fatto al capitolo 4.2.2.
// ...
// ...

// Ora, per liberare tutta la memoria occupata da questo albero, basta
// una sola chiamata
destroytree(root);

return 0;
}
```

4.4 Alberi binari di ricerca (BST)

Come detto precedentemente al capitolo 1.2.4 un **dizionario** in informatica è una **struttura dati astratta** che memorizza coppie (*chiave, valore*), quindi servono a memorizzare dati associativi come, ad esempio, una rubrica telefonica o una tabellla di simboli.

I dizionari consentono **tre operazioni fondamentali**:

- `lookup(Item k)` → cercare un elemento con chiave k ;
- `insert(Item k, Item v)` → inserire un elemento con chiave k e valore v ;
- `remove(Item k)` → rimuovere l'elemento con chiave k .

Un dizionario può essere implementato in vari modi:

Struttura dati	lookup	insert	remove
Vettore ordinato	$O(\log n)$	$O(n)$	$O(n)$
Vettore non ordinato	$O(n)$	$O(1)^*$	$O(1)^*$
Lista non ordinata	$O(n)$	$O(1)^*$	$O(1)^*$

*=Si assume che l'elemento sia già stato trovato, altrimenti $O(n)$

Queste tipologie di implementazioni presentano diversi **limiti di efficienza**. Ad esempio, i **vettori ordinati** permettono una ricerca veloce ma inserimenti e rimozioni lente,

poichè dovrei spostare tutti gli elementi ($O(n)$). Nei **vettori non ordinati**, invece, la ricerca richiede di scorrere tutti gli elementi ($O(n)$), mentre l'inserimento e la rimozione sono operazioni facili e veloci ($O(1)$) che prevedono un quantitativo di operazioni fisso, purché si conosca già la posizione dell'elemento su cui operare, come un'inserimento in testa e una rimozione dalla coda. Se invece fosse necessario inserire o rimuovere in una posizione specifica, il costo salirebbe nuovamente a $O(n)$.

Si conclude quindi che **utilizzare i vettori non è una buona idea**.

L'idea degli alberi binari di ricerca

Per risolvere il problema legato all'inefficienza dei vettori, vengono introdotti gli **alberi binari di ricerca** (BST - Binary Search Trees) proprio come una **struttura dati efficiente** per implementare i dizionari dinamici ordinati.

Cos'è un albero binario di ricerca?

Un **albero binario di ricerca** è un modo per realizzare una **struttura dati** che mantiene **i dati in ordine**, proprio come farebbe un vettore ordinato, grazie alla sua costruzione secondo **una logica dicotomica**. Inoltre, all'interno dell'albero **ogni nodo rappresenta una coppia** (*chiave, valore*).

La logica dicotomica implica che tra i nodi valgano le **seguenti proprietà**, che, per come sono definite, si applicano **in maniera ricorsiva**:

- Per ogni nodo u , tutte le chiavi contenute nel sottoalbero radicato nel figlio sinistro di u sono minori della chiave contenuta in u ;
- Per ogni nodo u , tutte le chiavi contenute nel sottoalbero radicato nel figlio destro di u sono maggiori della chiave contenuta in u .

"**Dicotomia**" significa *dividere in due*. In pratica, ogni volta che viene confrontata una chiave con quella del nodo corrente, si decide se "scendere a sinistra" o "scendere a destra", escludendo metà dell'albero.

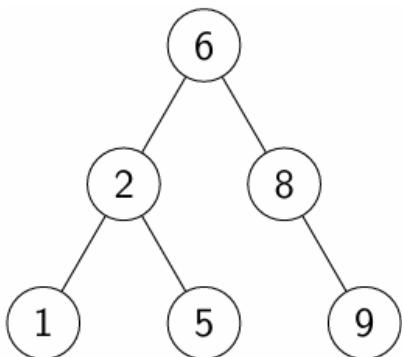
In questo modo tutte le operazioni di ricerca, inserimento e cancellazione sfruttano questa divisione **per ridurre progressivamente lo spazio di ricerca** (agevolare la ricerca).

Alcune precisazioni

Le chiavi dei nodi sono **maggiori o minori strette**, perché l'idea generale è che si utilizza una struttura dati per un dizionario, quindi **non è possibile che per una chiave** nella struttura **siano associati più valori**, ad esempio, non posso avere due nodi così strutturati:

- (10, Alberto)
- (10, Roberto)

Però, nulla vieta che ad una chiave sia associato un insieme (dinamico o non), ad esempio, (10, ["Alberto", "Roberto"])



Le operazioni previste dagli alberi binari di ricerca sono:

- **Ricerca** di un elemento nell'insieme;
- **Inserimento** di un elemento nell'insieme;
- Ricerca del **minimo** e del **massimo** dell'insieme;
- **Successore** e **predecessore** di un elemento nell'insieme;
- **Cancellazione** di un elemento dall'insieme.

4.4.1 Implementazione della ricerca

Negli alberi binari di ricerca, la ricerca di un nodo può essere effettuata in modalità **ricorsiva** oppure in modalità **iterativa**.

Ricerca ricorsiva

```

1 // ricerca RICORSIVA della chiave x (chiave == data)
2 bstree *search(bstree *p, int x) {
3 //se trovo x oppure ho finito la ricorsione allora ritorno p
4 if(p==NULL || p->data==x) return p;
5
6 // x > data...search the right subtree
7 else if(x > p->data) return search(p->right, x);
8
9 //x < data ... search the left subtree
10 else return search(p->left, x);
11 }
```

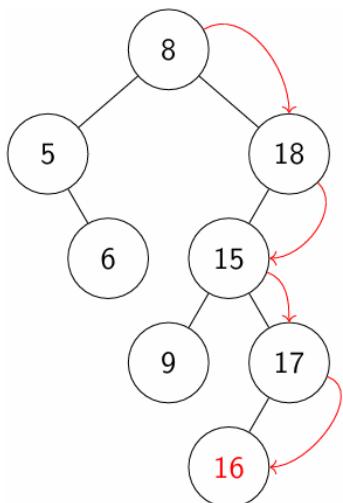
- **Tipo di ritorno:** la funzione restituisce un puntatore a un nodo della struttura bstree, la cui composizione è identica a quella di un albero binario vista al capitolo 4.2.1.

Il nodo restituito è:

- Il **nodo trovato** se la chiave x è presente nell'albero;
- Oppure NULL, se la chiave non esiste, cioè **se la ricerca termina su un puntatore nullo**. Dunque, il tipo di ritorno consente di sapere **dove si trova l'elemento**, oppure **che non esiste**.

- **Parametri:** la funzione bstree accetta due parametri;
 - bstree *p: è il puntatore al nodo corrente dell'albero binario di ricerca. In genere la prima chiamata della funzione riceve la radice dell'albero (root), ma nelle chiamate ricorsive il parametro p sarà aggiornato ai figli sinistro o destro.
 - int x: rappresenta la chiave da cercare (in questo caso un intero). Si confronta con il campo data contenuto nel nodo corrente.
- **Funzionamento del codice:** la funzione implementa una **ricerca dicotomica ricorsiva**. Nell'esempio riportato in Figure 35, viene effettuata la **ricerca del valore 16**.

Figure 35: Albero BST



- riga 4: la prima istruzione della funzione è un **controllo di terminazione** per fare in modo che, al ripetere della funzione ricorsiva, si capisca se la chiave del nodo è quella cercata o meno; Se p==NULL, significa che abbiamo raggiunto un ramo vuoto, la chiave non esiste e la funzione restituisce NULL, mentre se p->data==x abbiamo trovato il nodo desiderato e si ritorna il puntatore al nodo stesso.
- riga 7: questa seconda istruzione implementa la ricerca ricorsiva nel sotto albero destro. Se la chiave cercata x è **maggiori della chiave del nodo corrente** (p->data), allora, secondo la proprietà del BST (capitolo 4.4), il **valore** può trovarsi solo nel **sottoalbero destro**.

Dunque, la funzione richiama se stessa (in modo ricorsivo), passando come parametri il figlio destro del nodo sul quale si stava lavorando (p->right), e ancora una volta la chiave cercata (x). A questo punto la funzione ricomincia, e se il controllo di terminazione non si attiva, viene ricontrollato se x è maggiore o minore di p->data.

- riga 10: anche in questo caso viene implementata la ricerca ricorsiva ma nel sottoalbero sinistro. È **importante notare** che non viene specificato x<p->left proprio perché come descritto al capitolo capitolo 4.4 ("Alcune precisazioni"), **non possono essere presenti nodi con lo stesso valore di chiave**, dunque è scontato che sia l'unica altra opzione possibile.

Quindi, se la chiave cercata x è **minore della chiave del nodo corrente** (p->data), allora, secondo la proprietà del BST, il **valore** può trovarsi solo nel **sottoalbero sinistro**.

Anche in questo caso, viene richiamata ricorsivamente la funzione, con p->left come nuovo nodo di partenza, fino a che non si attiva il controllo di terminazione, o x è maggiore di p->data.

Ricerca iterativa

```

1 // ricerca ITERATIVA della chiave x (chiave == data)
2 bstree *itsearch(bstree *p, int x) {
3     bstree *q = p;
4     while (q && x != q->data) {
5         if (x < q->data) q = q->left;
6         else q = q->right;
7     }
8     return q;
9 }
```

Il **tipo di ritorno** e i **parametri**, sono uguali a quelli della ricerca in modalità ricorsiva, ciò che cambia è la **struttura interna della funzione**, che però produce lo stesso risultato.

- **Funzionamento del codice:** in questo caso si effettua una **ricerca dicotomica**, ovvero ad ogni passo si elimina metà dell'albero possibile, usando però una modalità iterativa.
 - *riga 3:* viene creato un puntatore locale (**q*) alla struttura bstree, il quale viene fatto puntare al parametro *p* che di solito rappresenta la radice (*root*). In questo modo partendo da *p*, **q* verrà usato per scorrere l'albero.
 - *riga 4:* il while continua finché il nodo esiste e la chiave non è stata trovata. L'**uscita dal while** rappresenta il **controllo di terminazione**: se *q* è NULL la chiave non esiste, altrimenti è stata trovata.
 - *riga 5-6:* viene definito il controllo condizionale per fare in modo di "scorrere" l'albero a destra se *x* > *q->data* o a sinistra se *x* < *q->data*.
 - *riga 8:* quando la condizione del while non viene più rispettata, la chiave del nodo viene restituita.

Esempio di utilizzo (valido per entrambi i tipi di ricerca)

Ipotizziamo di voler effettuare la ricerca del nodo 16, come illustrato in Figure 35.

```
int main(int argc, char *argv[]) {
bstree *root = NULL;
root = malloc(sizeof(bstree));

// Allocazione di tutti i nodi e assegnazione di chiave-valore per
// ognuno di essi, secondo la logica dicotomica (Figure 35).
// ...
// ...

bstree *res = search(root, 16); // Per la ricerca ricorsiva
//bstree *res = itsearch(root, 16); // Per la ricerca iterativa

if(res) printf("chiave trovata! valore = %d\n", res->data);
else printf("chiave NON presente nell'albero\n");

return 0;
}
```

4.4.2 Creazione di un nuovo nodo

```
1 // Funzione per creare un nodo
2 bstree *new_node(int x) {
3     bstree *p;
4     p = malloc(sizeof(bstree));
5     p->data = x;
6     p->left = NULL;
7     p->right = NULL;
8     return p;
9 }
```

- **Tipo di ritorno:** la funzione restituisce un puntatore alla struttura del nodo appena creato.
- **Parametri:** la funzione `new_node()` accetta come unico parametro un intero (`int x`), ovvero la chiave da salvare nel nodo.
- **Funzionamento del codice:** vengono definite le istruzioni per assegnare al nuovo nodo il valore della chiave e dei suoi figli sinistro e destro.
 - *riga 3-4:* viene dichiarato il puntatore `p` alla struttura `bstree`. Successivamente, tramite `malloc(sizeof(bstree))` viene allocata dinamicamente la memoria necessaria a contenere un nodo, e il puntatore `p` viene fatto puntare a questa nuova area di memoria.
 - *riga 5:* viene assegnato al campo `data` del nodo il valore `x`, fornito come parametro.
 - *riga 6-7:* i puntatori `left` e `right` vengono inizializzati a `NULL`. Ciò significa che, al momento della creazione, **il nodo non ha figli** ed è **una foglia**.
 - *riga 8:* la funzione restituisce il puntatore al nodo creato.

4.4.3 Inserimento di un nodo

Come per la ricerca dei nodi, anche l'**inserimento di un nodo** all'interno dell'albero può avvenire mediante due modalità: **ricorsiva** o **iterativa**.

Inserimento ricorsivo

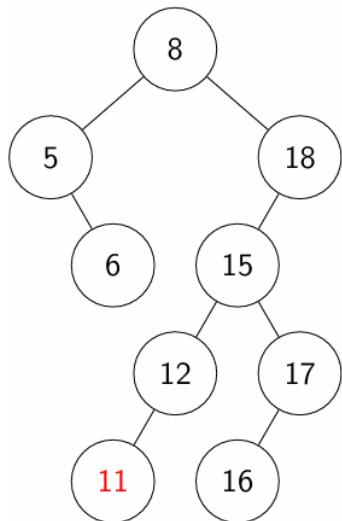
```

1 // Funzione per inserire un nodo
2 bstree *insert(bstree *p, int x) {
3 //void tree
4 if(p==NULL) return new_node(x);
5
6 //insert to right
7 else if(x > p->data) p->right = insert(p->right, x);
8
9 //insert to left
10 else p->left = insert(p->left, x);
11
12 return p;
13 }
```

- **Tipo di ritorno:** la funzione restituisce un puntatore a `bstree`.
Il valore restituito rappresenta la radice del sottoalbero risultante dopo l'inserimento;
- **Parametri:** la funzione accetta due parametri;
 - `bstree *p`: è il puntatore al nodo corrente dell'albero binario di ricerca.
In genere la prima chiamata della funzione riceve la radice dell'albero (`root`), ma nelle chiamate ricorsive il parametro `p` sarà aggiornato ai figli sinistro o destro.
 - `int x`: è la chiave da inserire nel campo `data` del nuovo nodo.
- **Funzionamento del codice:** la struttura del codice è simile a quella che viene utilizzata per effettuare la ricerca di un nodo (capitolo 4.4.1). Come avviene per la ricerca di un nodo, anche l'**inserimento sfrutta la logica dicotomica** utilizzata per la creazione dell'albero binario in modo tale che **in base al valore della chiave del nodo che si vuole inserire**, venga "esclusa" una metà dell'albero ad ogni ricorsione o iterazione (in questo caso ad ogni ricorsione).

Nell'esempio riportato in Figure 36 viene effettuato l'inserimento del valore 11.

Figure 36: Albero BST



- riga 3: se il primo controllo condizionale è verificato ($p=NULL$), significa che siamo arrivati alla posizione corretta dove deve essere inserito il nuovo nodo. Una volta trovata la posizione corretta, viene creato e restituito un nuovo nodo tramite la funzione `new_node(x)` vista al capitolo 4.4.2.
- riga 7: in questo secondo controllo condizionale viene controllato se la chiave x da inserire sia maggiore della chiave del nodo corrente ($x > p->data$). In caso affermativo viene richiamata la funzione in modo ricorsivo sul figlio destro del nodo corrente ($p->right$), come specificato dalle proprietà dei BST.
- riga 9: il terzo controllo condizionale viene eseguito nel caso in cui il nodo corrente **non sia nullo** ($p!=NULL$) e x **non sia maggiore** di $p->data$. Ciò vuol dire che $x < p->data$ e dunque, secondo le proprietà dei BST, la funzione viene richiamata in modo ricorsivo sul figlio sinistro del nodo corrente ($p->left$).
- riga 11: la funzione ritorna sempre p , cioè la radice del sottoalbero risultante, o per meglio dire, il padre del figlio appena creato.

Inserimento iterativo

```

1 bstree *itinsert(bstree *p, int x) {
2     bstree *y = NULL, *q = p;
3     while(q){ //scendo fino ad una foglia
4         y = q;
5         if (q->data > x) q = q->left;
6         else q = q->right;
7     }
8     if(y==NULL) return new_node(x); // se albero vuoto
9     else if(y->data > x) y->left=new_node(x); // inserisco sx
10    else y->right=new_node(x); // inserisco dx
11
12    return p;
13 }
```

Il **tipo di ritorno** e i **parametri**, sono uguali a quelli dell'inserimento in modalità ricorsiva, ciò che cambia è la **struttura interna della funzione**, che però produce lo stesso risultato.

- **Funzionamento del codice:** il procedimento sfrutta la stessa logica dicotomica vista nella versione ricorsiva, ma la navigazione avviene mediante un ciclo `while`. La visita scende nell'albero "scorrendo" ogni volta il puntatore verso il figlio sinistro o destro, finché non si arriva alla posizione corretta per inserire il nuovo nodo.
- riga 2: vengono dichiarati due puntatori alla struttura `bstree`.
 - `*q`: viene fatto puntare a `p`, che nella fase iniziale rappresenta il nodo radice (root);
 - `*y`: viene fatto puntare a `NULL`. Il motivo è che in seguito, `y` e `q` verranno fatti scorrere assieme, in questo modo `y` partendo "*in ritardo*" memorizzerà sempre il nodo precedente a `q`. Da questo punto di vista si può dire che `y` sia sempre il padre di `q`.
- riga 3: subito dopo viene utilizzato un ciclo `while` per controllare che il nodo corrente `q` non sia `NULL`. In questo modo si ha la sicurezza che il nodo corrente non sia una foglia ed

è possibile definire le successive istruzioni per poter scorrere l'albero in base alla chiave che si vuole ricercare.

- riga 4-6: alla prima iterazione, il nodo y che puntava a NULL prende il nodo q; In questo modo, in base al valore di x (controllo condizionale), q viene fatto scorrere verso il figlio destro o sinistro e y memorizzerà il nodo che è considerato il padre di questi ultimi.

Quando si esce dal ciclo while significa che q, andando a destra o a sinistra è arrivato ad un nodo NULL, quindi oltre una foglia. A questo punto viene eseguito uno dei controlli condizionali:

- riga 8: in questo controllo condizionale si gestisce il caso in cui y sia rimasto a NULL. Quando succede ciò, significa che anche q era a NULL e dunque l'albero era vuoto: viene quindi richiamata la funzione `new_node(x)` per creare ed inserire il primo nodo dell'albero.
- riga 9-10: in questi ultimi due controlli condizionali viene deciso se creare un figlio sinistro o un figlio destro. Si è detto che il puntatore y rappresenta il padre di q, in altre parole sarà l'ultimo nodo valido (foglia) alla terminazione del ciclo while. Dunque, per capire se il nuovo nodo debba essere un figlio destro si controlla che $x > y->data$, mentre per capire se debba essere un figlio sinistro si controlla che $x < y->data$.
- riga 12: viene restituita la radice originale dell'albero.

Esempio di utilizzo (valido per entrambi i tipi di inserimento)

Ipotizziamo di voler effettuare l'inserimento del nodo 11, come illustrato in Figure 36.

```
int main(int argc, char *argv[]) {
bstree *root = NULL;
/* N.B: In questo caso la malloc non serve perche' viene effettuata
dalla funzione di inserimento quando necessario tramite la funzione
"new_node()" (quindi solitamente al primo passaggio). */

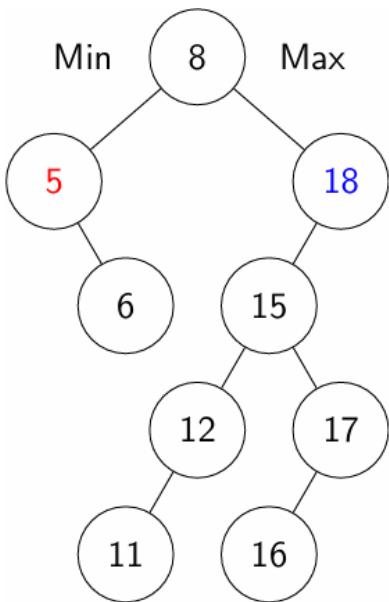
// Inserimento ricorsivo dei valori
root = insert(root, 8);
root = insert(root, 5);
// ...
// ...
root = insert(root, 11);

// Per l'inserimento iterativo basterebbe sostituire:
// root = itinsert(root, 11);
return 0;
}
```

4.4.4 Ricerca del massimo e del minimo

Ancora una volta la logica dicotomica con la quale vengono creati gli alberi binari torna utile per la ricerca del nodo dell'albero con valore di chiave massimo o minimo.

Partendo dalla radice, per trovare il **nodo minimo** va ispezionato sempre il **lato sinistro dell'albero binario**, mentre per trovare il **nodo massimo** va ispezionato **lato destro** dell'albero.



Esempio pratico

- **Nodo minimo:** Per trovare il **nodo con valore di chiave minimo** all'interno dell'albero binario, si effettua una ricerca sempre verso sinistra. Il nodo minimo è quello il cui figlio sinistro è NULL (nel caso dell'immagine il nodo 5);
- **Nodo massimo:** Per trovare il **nodo con valore di chiave massimo** all'interno dell'albero binario, si effettua una ricerca sempre verso destra. Il nodo minimo è quello il cui figlio destro è NULL (nel caso dell'immagine il nodo 18);

Lo stesso discorso vale per trovare il **massimo o il minimo di un sottoalbero** radicato a partire da un nodo specifico dell'albero: ad esempio, il massimo del sottoalbero radicato nel nodo con valore di chiave 17 è la radice stessa poiché il suo figlio destro è NULL.

Anche la ricerca del massimo e del minimo possono essere effettuate in modalità **ricorsiva** o in modalità **iterativa**. Il **costo** di entrambe le operazioni di ricerca è **proporzionale all'altezza dell'albero**.

Ricerca ricorsiva del massimo e del minimo

```

1 // Function to find the maximum value
2 bstree *find_maximum(bstree *p) {
3     if(p == NULL) return NULL;
4     else if(p->right != NULL) return find_maximum(p->right);
5     return p;
6 }
7
8 // Function to find the minimum value
9 bstree *find_minimum(bstree *p) {
10    if(p == NULL) return NULL;
11    else if(p->left != NULL) return find_minimum(p->left);
12    return p;
13 }
```

- **Tipo di ritorno:** la funzioni restituiscono un puntatore a bstree. Se l'albero non è vuoto, il valore restituito sarà il nodo che contiene la chiave massima o minima del BST;
- **Parametri:** entrambe le funzioni accettano un solo parametro, ovvero un puntatore alla struttura di un nodo bstree *p. In base alla logica della funzione, p è il puntatore al nodo dal quale si vuole iniziare la ricerca del massimo o del minimo.
Solitamente si utilizza root se si vuole esaminare l'intero albero, ma è possibile passare anche il puntatore alla radice di un sottoalbero.
- **Funzionamento del codice:** come detto precedentemente, la logica dietro la ricerca del valore massimo e del valore minimo è abbastanza semplice.
 - **riga 3/10:** se il puntatore passato come parametro è nullo (p==NULL), significa che l'albero è vuoto e non è possibile determinare alcun valore massimo o minimo;
 - **riga 4/11:** per implementare ciò che è stato detto precedentemente viene effettuato un controllo condizionale che differisce in base al tipo di funzione:

- nel caso della ricerca del massimo si verifica se il **figlio destro** del nodo corrente non è nullo;
- nel caso della ricerca del minimo si verifica se il **figlio sinistro** del nodo corrente non è nullo.

Nel caso in cui il **controllo condizionale sia vero**, la funzione viene richiamata in modo ricorsivo specificando il figlio verso il quale ci si vuole spostare (`p->right` per il massimo e `p->left` per il minimo), in modo tale che la funzione ricomincia con la posizione successiva da analizzare.

- *riga 5/12:* se invece i controlli condizionali risultassero falsi, significherebbe che il massimo/minimo è stato trovato perché il figlio destro/sinistro non esiste (`p->right==NULL` o `p->left==NULL`), uscendo così dalla ricorsione.

A questo punto viene restituito `p`, ovvero l'ultimo nodo valido visitato, il quale rappresenta il valore massimo/minimo dell'albero.

Ricerca iterativa del massimo e del minimo

```

1 bstree *itfind_minimum(bstree *p) {
2     if (p == NULL) return NULL;
3     while (p->left != NULL) p = p->left;
4     return p;
5 }
6
7 bstree *itfind_maximum(bstree *p) {
8     if (p == NULL) return NULL;
9     while (p->right != NULL) p = p->right;
10    return p;
11 }
```

Il **tipo di ritorno** e i **parametri**, sono uguali a quelli che vengono utilizzati con la modalità ricorsiva, ciò che cambia è la struttura interna della funzione che però produce lo stesso risultato.

Utilizzando un'altra modalità di esecuzione, il **funzionamento del codice** per forza di cose cambia, ma la funzione rimane comunque **molto semplice** e la maggior parte delle istruzioni rimangono invariate. I controlli condizionali che nella modalità ricorsiva vengono utilizzati per capire quando richiamare ricorsivamente la funzione, in questo caso vengono utilizzati come guardia nel ciclo `while` per fare in modo di "scorrere" l'albero a destra o a sinistra in base al valore che si vuole trovare (massimo o minimo).

Esempio di utilizzo (valido per entrambe le modalità)

```

1 int main(int argc, char *argv[]) {
2     bstree *root = NULL;
3     /* Creazione di un albero BST in logica dicotomica utilizzando la
4      funzione di inserimento, come fatto al capitolo (4.4.3) */
5     // ...
6     // ...
7
8     // Ricerca ricorsiva del massimo e del minimo
9     bstree *min_node = find_minimum(root);
10    bstree *max_node = find_maximum(root);
```

```

11
12 // Ricerca iterativa del massimo e del minimo
13 bstree *min_node = itfind_minimum(root);
14 bstree *max_node = itfind_maximum(root);
15
16 printf("Minimo trovato = %d\n", min_node->data);
17 printf("Massimo trovato = %d\n", max_node->data);
18
19 return 0;
20 }
```

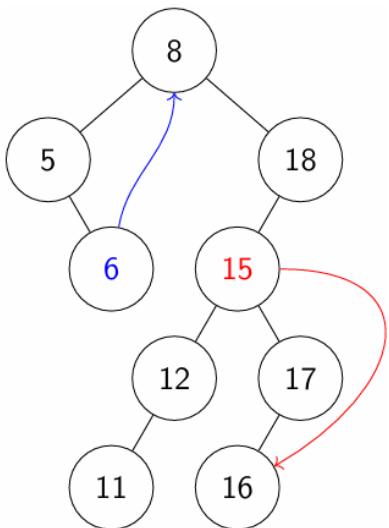
4.4.5 Ricerca del successore-predecessore

Definizione di successore

Il **successore** di un nodo u è il più piccolo nodo maggiore di u .

Allo stesso modo, possiamo dire che il **predecessore** di un nodo u è il più grande nodo minore di u .

Figure 38: Albero BST



Per trovare il successore di u , sono dati due casi:

- Caso 1:** Se il nodo u ha un figlio destro, il successore v è il minimo nodo del sottoalbero destro; Dunque in questo caso la situazione è molto semplice perché basta utilizzare la funzione vista al capitolo 4.4.4 per la ricerca del minimo. Trovando il nodo più piccolo, una **caratteristica determinante** di questo caso è che il successore non ha figlio sinistro.
- Caso 2:** Se il nodo u non ha un figlio destro, il successore è il **primo antenato** (v) di u , per cui u sta nel sottoalbero sinistro dell'antenato v .

Ad esempio, ipotizziamo di voler cercare il successore del nodo 6 come illustrato in Figure 38. Per prima cosa si controlla il sottoalbero radicato in 5, e ci si chiede "u (6) sta nel sottoalbero sinistro di v (5)?" In questo caso no, infatti 5 non è

il successore. Subito dopo, continuando a salire, si arriva al sottoalbero radicato in 8: "u (6) sta nel sottoalbero sinistro di v (8)?" In questo caso la risposta è sì, dunque 8 è il successore. Ciò è dato anche dal fatto che trovandosi nell'albero sinistro, il primo antenato v che incontro, **per le proprietà dei BST**, sarà sicuramente più grande del nodo u : in questo caso andrà effettuato **un controllo tra le chiavi dei nodi**.

Dunque, la struttura di un albero binario di ricerca consente di determinare il successore di un nodo **senza mai effettuare il confronto tra le chiavi** (quando il successore viene trovato con il **caso 1**) come invece avviene per la ricerca del massimo o del minimo.

Ricerca iterativa del successore

```

1 /* Definisco la funzione che ricerca il nodo antenato per gestire il
2 caso 2 */
3 bstree* ancestor(bstree *p, bstree *px) {
4     int x = px->data;
```

```

5 bstree *y = NULL; //l'antenato
6 bstree *q = p;
7 while (q && x != q->data) {
8     y = q;
9     if (x < q->data) q = q->left;
10    else q = q->right;
11 }
12 return y;
13 }
```

```

1 // Funzione principale per la ricerca del successore
2 bstree* successor(bstree *p, bstree *px){
3     if(px->right) return find_minimum(px->right); //Caso 1
4     bstree *y = ancestor(p, px); // Caso 2 (discesa)
5     while (px && y && px == y->right){ //Caso 2 (risalita)
6         px = y;
7         y = ancestor(p, px);
8     }
9     return y;
10 }
```

Per seguire un filo logico durante il discorso, analizziamo per prima la funzione `successor`.

- **Tipo di ritorno:** la funzione restituisce un puntatore (`y`) al nodo del successore cercato;
 - **Parametri:** la funzione accetta in ingresso due parametri;
 - `bstree *p`: un puntatore alla struttura di un nodo generico che viene utilizzato per passare il puntatore alla radice dell'intero albero;
 - `bstree *px`: anche `px` è un puntatore alla struttura di un nodo generico, ma viene utilizzato per passare il puntatore al nodo dell'albero del quale si vuole cercare il successore;
 - **Funzionamento del codice:** come detto precedentemente, quando ci si occupa della ricerca del successore, bisogna gestire due casi;
 - *riga 3*: in questa riga viene gestito il caso più semplice (caso 1). Viene effettuato un controllo condizionale sull'esistenza del figlio destro sul nodo del quale ci interessa trovare il successore (`px->right`). Se il controllo risulta vero, significa che il nodo `px` avrà un sottoalbero che, per le caratteristiche dei BST, conterrà solo nodi con un valore di chiave maggiore al suo.
- A questo punto, per trovare il successore di `px` basta utilizzare `find_minimum` o `itfind_minimum` sul figlio destro di `px` per trovare il nodo minimo dell'intero sottoalbero;
- *riga 4*: da questo punto in poi viene gestito il caso in cui il nodo di cui si vuole conoscere il successore **non abbia un figlio destro** (caso 2).

Funzionamento di `ancestor`: come detto precedentemente, bisogna risalire lungo l'albero, partendo dal nodo di cui vogliamo conoscere il successore. Poiché i nodi non hanno un puntatore al padre, per trovare il primo nodo "sopra" `px` dobbiamo necessariamente scendere partendo dalla radice. La funzione `ancestor` si occupa proprio di trovare il nodo che sta immediatamente sopra `px` lungo il cammino dalla radice. Il **tipo di ritorno e i parametri** sono quindi gli stessi della funzione `successor`.

- *riga 4*: il valore della chiave del nodo `px` viene salvato nella variabile `x`;
- *riga 5*: viene dichiarato un puntatore `y` e inizializzato a `NULL`: questo punterà di volta

- in volta al nodo più recente incontrato nel cammino;
- *riga 6*: viene creato un puntatore *q* e gli viene assegnata la radice *p*, per poter scorrere tutto l'albero partendo dall'alto;
 - *riga 7*: Per poter effettuare lo scorrimento si utilizza un ciclo *while* la cui condizione è quella di continuare la sua iterazione fino a che il nodo *q* è valido e fino a che le chiavi del nodo *px* e del nodo *q* non coincidono.
 - *riga 8*: ad ogni iterazione il nodo puntato da *q* viene salvato in *y*. In questo modo, **quando finalmente arriva a *px***, il nodo *y* sarà già impostato al nodo che lo precede lungo il cammino;
 - *riga 9-10*: infine, in queste righe vengono gestiti i controlli condizionali per quanto riguarda lo spostamento all'interno dell'albero e, come sempre fatto per la logica costruttiva degli alberi BST, ci si sposta sul figlio destro se la chiave cercata (*x*) del nodo *px* è più grande della chiave del nodo attuale *q*, altrimenti ci si sposta sul nodo sinistro.
 - *riga 12*: quando la condizione nel ciclo *while* non è più rispettata si restituisce *y*. Alla fine della funzione *ancestor()*, *y* **contiene il nodo padre (inteso come nodo immediatamente sopra) del nodo *px* lungo il cammino dalla radice**.
- *riga 5*: da qui in poi, nel codice del successore, viene eseguito un ulteriore *while* per verificare se il nodo antenato trovato con *ancestor* sia realmente il **successore** di *px*, oppure se sia necessario risalire ulteriormente.
Perché risalire? Perché se *px* è figlio destro di *y*, *y* non può essere il suo successore (per le proprietà dei BST).
 - *riga 6-7*: a questo punto del codice, una volta entrati nel *while* di risalita, abbiamo appurato che il nodo "appena sopra" il nodo *px* (ovvero (*y*)) non è il suo successore. Ciò vuol dire che non serve più memorizzare il nodo *px* originale, ma bisogna **risalire per trovare il primo nodo che è abbia un figlio sinistro**: il nodo trovato sarà di sicuro il successore del nodo *px* originale poiché avendo un figlio sinistro avrà un valore di chiave maggiore. Per fare ciò all'interno del *while*, *y* diventa il nuovo nodo originale "fittizio" e viene richiamata su di esso la funzione *ancestor()*;
 - *riga 9*: quando la guardia del *while* non è più rispettata siamo sicuri di aver trovato il successore del nodo originale restituendo *y* che contiene il risultato di *ancestors()*.

Esempio di utilizzo

Ipotizziamo di voler effettuare la ricerca del successore del nodo 17 come in Figure 38.

```
int main(int argc, char *argv[]) {
bstree *root = NULL;
/* Creazione dell'albero BST in logica dicotomica illustrato in
Figure 38 utilizzando la funzione di inserimento, come fatto al
capitolo (4.4.3) */
// ...
// ...

bstree *px = itsearch(root, 17); // Scelgo un nodo
bstree *succ = successor(root, px); // Individuo il suo successore
printf("Il successore del nodo %d e' %d\n", px->data, succ->data);
}
```

Passaggio da successore a predecessore

Il discorso intrapreso per quando riguarda la ricerca del successore ovviamente vale anche per la ricerca del **predecessore** di un determinato nod all'interno dell'albero.

- Il ragionamento con il *caso 1* e il *caso 2* vengono effettuati tenendo in considerazione **il figlio sinistro e non il figlio destro**;
- Nello specifico, per il *caso 1* al posto della funzione `find_minimum()` si utilizzerà `find_maximum()`.

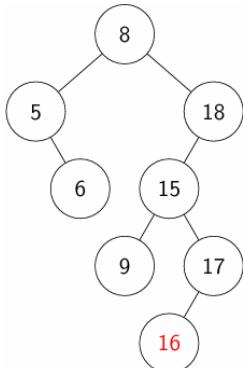
4.4.6 Cancellazione di un nodo

Tramite la cancellazione di un nodo viene **rimossa la chiave k** dall'albero T .

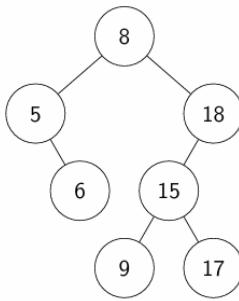
Durante la cancellazione di un determinato nodo possono sorgere **tre differenti casi**:

- *Caso 1 - Il nodo u da eliminare non ha figli*: in questo caso u è una foglia e viene semplicemente rimossa. **Eliminare le foglie non cambia l'ordine dei nodi rimanenti**;
- *Caso 2 - Il nodo u da eliminare ha un solo figlio f (destro o sinistro)*: in questo caso si elimina il nodo u rendendo f figlio del padre di u .

Come si può notare nell'esempio (b), nonostante il nodo con valore di chiave 18 sia la radice di un sottoalbero, quest'ultimo può essere collegato senza alcun tipo di problema al nodo con valore di chiave 8. Infatti, 18 era il figlio destro del nodo 8, dunque tutti i nodi sotto di esso saranno comunque maggiori di 8, **rispettando così le proprietà degli alberi BST**.



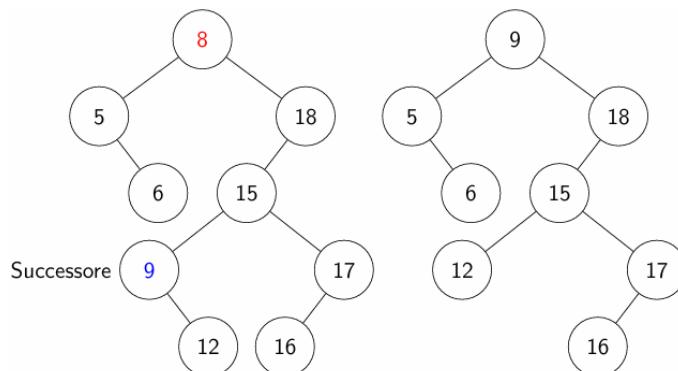
(a) Caso 1 - nessun figlio



(b) Caso 2 - un solo figlio

- *Caso 3 - Il nodo u da eliminare ha due figli*: in questo caso la logica dietro l'eliminazione del nodo diventa leggermente più complessa. Come illustrato in Figure 39, supponiamo di

Figure 39: Cancellazione di un nodo



A questo punto possiamo dire che il successore (v) **rispetti le seguenti proprietà**:

- È sicuramente **maggiore** dei nodi nel sottoalbero sinistro di u ;
- È sicuramente **minore** dei nodi nel sottoalbero destro di u .

voler eliminare il nodo u con valore di chiave 8. Per prima cosa bisogna individuare il successore di u , ovvero, come detto al capitolo 4.4.5, "*il più piccolo valore più grande di u* ": quando si va a rimuovere un nodo con due figli, il suo **successore** ha la caratteristica di essere il sostituto adatto a ricoprire quella posizione in modo tale che le proprietà degli alberi BST vengano rispettate.

Una volta che il successore è stato individuato, bisogna sovrascriverlo al nodo che si vuole cancellare, eliminando anch'esso dalla sua posizione attuale.

A questo punto il *caso 3* si può **ridurre ad uno dei due casi** visti in precedenza:

- **Caso 1: il successore è una foglia**, quindi può essere sovrascritto e rimosso senza ulteriori accorgimenti;
- **Caso 2: il successore ha solo un figlio destro.**

A differenza di un normale *caso 2*, quando si arriva ad esso passando dal *caso 3*, **non può mai capitare** che il **successore** abbia **solo figlio sinistro**, perché se avesse il figlio sinistro **non sarebbe il minimo** del sottoalbero destro (capitolo 4.4.4 - caso 1).

Cancellazione ricorsiva di un nodo

```

1 bstree *deleteNode(bstree *p, int x) {
2     if(p==NULL) return NULL;
3     if (x > p->data) p->right = deleteNode(p->right, x);
4     else if(x < p->data) p->left = deleteNode(p->left, x);
5     else{ //node found is p
6         if(p->left==NULL && p->right==NULL){ //Nessun figlio (caso 1)
7             free(p);
8             return NULL;
9         }
10        else if(p->left==NULL || p->right==NULL){ //Un figlio (caso 2)
11            bstree *temp;
12            if(p->left==NULL) temp = p->right;
13            else temp = p->left;
14            free(p);
15            return temp;
16        }
17        else{ //Due figli (caso 3)
18            bstree *temp = find_minimum(p->right);
19            p->data = temp->data;
20            p->right = deleteNode(p->right, temp->data);
21        }
22    }
23    return p;
24 }
```

- **Tipo di ritorno:** la procedura di cancellazione di un nodo restituisce un puntatore alla radice dell'albero, che **può cambiare** se il nodo cancellato è **proprio la radice**;
- **Parametri:** la funzione accetta in ingresso due parametri;
 - `bstree *p`: un puntatore alla struttura di un nodo generico che viene utilizzato per passare il puntatore alla radice dell'intero albero;
 - `int x`: un valore intero `x`, che rappresenta la chiave del nodo da cancellare.
- **Funzionamento del codice:** All'interno della funzione `deleteNode()` vengono gestiti tutti i casi di cancellazione del nodo.
 - *riga 2*: si gestisce il caso base il caso base: se il puntatore `p` è `NULL`, vuol dire che non esiste alcun albero o che non è stato trovato alcun nodo con valore `x`. In questo caso viene semplicemente restituito `NULL`;

- *riga 3-4*: in queste due righe vengono gestiti i controlli condizionali per la ricerca del nodo da rimuovere all'interno dell'albero. Se la chiave x è maggiore della chiave del nodo corrente ($x > p->data$), allora la ricerca avverrà richiamando ricorsivamente `deleteNode()` sulla radice del sottoalbero destro, altrimenti ($x < p->data$) la ricerca avviene, sempre in modo ricorsivo, ma sul sottoalbero sinistro;
- *riga 5*: se si entra in quest'ultimo caso significa che il nodo da eliminare è stato trovato ($x == p->data$). All'interno di esso vengono implementati i 3 casi differenti per gestire l'eliminazione del nodo in base alla sua posizione all'interno dell'albero;
- *riga 6-8*: il primo controllo condizionale gestisce il caso in cui il nodo da rimuovere sia una foglia (*caso 1*). Quest'ultimo è valido solo se entrambi i puntatori ai figli destro e sinistro sono `NULL`. Verificato ciò, si procede ad eliminare il nodo p tramite il comando `free()` e viene ritornato `NULL` come nuovo puntatore da assegnare al padre (o alla radice se p era la radice).;
- *riga 10-16*: il secondo controllo condizionale gestisce il caso in cui il nodo da rimuovere abbia un solo figlio (che sia destro o sinistro, ovvero *caso 2*). Quest'ultimo è valido solo se almeno uno dei due puntatori ai figli è nullo. Una volta all'interno del controllo condizionale è bene **prestare attenzione anche al meccanismo di ricorsione** che viene utilizzato per permettere di **salvare il figlio del nodo** che si vuole eliminare.

Facendo riferimento alla Figure 39, ipotizziamo di voler rimuovere il nodo con valore di chiave 9 ed essere posizionati in 15. La funzione inizia e ci si ferma nella *riga 4* (poiché $9 < 15$) dove avvengono due cose:

- Si scende a sinistra perché viene richiamata `deleteNode()` su $p->left$, ovvero 9;
- Allo stesso tempo, il puntatore al figlio sinistro (9) prenderà il risultato della chiamata appena avvenuta su `deleteNode()`.

Una volta su 9, si entra nell'ultimo *else* e, in seguito, nel controllo condizionale per i nodi con un solo figlio, poiché l'unico figlio di 9 è 12. Viene dichiarato un puntatore generico alla struttura di un nodo `temp`: quest'ultimo verrà utilizzato per salvare temporaneamente il figlio (12), così da poter fare la `free()` del nodo 9 senza perderlo.

A questo punto `temp` viene restituito a $p->left$ (il puntatore al figlio sinistro di 15) come risultato della funzione ricorsiva chiamata prima: in questo modo, 12 diventa figlio sinistro diretto di 15.

- *riga 18-21*: il terzo controllo condizionale gestisce il caso in cui il nodo da rimuovere abbia figlio destro e sinistro, ed ovviamente è valido se non si entra negli altri due casi prima. Precedentemente si è detto che, in teoria, per eliminare un nodo con due figli si dovrebbe trovare il suo successore (capitolo 4.4.5). Dal punto di vista implementativo, però, nel BST il successore coincide sempre con il nodo avente il valore minimo nel sottoalbero destro. Per questo motivo, nel codice non viene richiamata esplicitamente una funzione `successor()`, ma si utilizza direttamente la funzione `find_minimum(p->right)` (capitolo 4.4.4), che rappresenta esattamente tale concetto.

Trovato il suo successore, il valore della chiave di quest'ultimo viene semplicemente sovrascritto ($p->data = temp->data$) sul nodo da eliminare.

A questo punto il problema è che il successore **esiste ancora fisicamente al suo posto originario**. La sua rimozione è semplice: per definizione il successore di un nodo non ha un figlio sinistro, quindi si richiama ricorsivamente la funzione `deleteNode()` che ne gestirà l'eliminazione seguendo il *caso 1* se è una foglia o il *caso 2* se ha un figlio destro.

Esempio di utilizzo

Ipotizziamo di voler cancellare il nodo 8 come illustrato in Figure 39.

```
int main(int argc, char *argv[]) {  
bstree *root = NULL;  
/* Creazione dell'albero BST in logica dicotomica illustrato in  
Figure 39 utilizzando la funzione di inserimento, come fatto al  
capitolo (4.4.3) */  
// ...  
// ...  
  
root = deleteNode(root, 8); // Cancellazione del nodo 8  
printf("La nuova radice e' %d\n", root->data);  
}
```

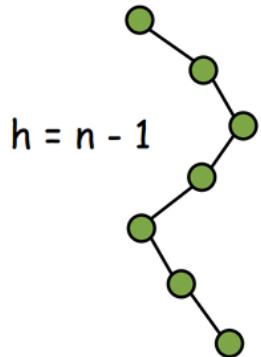
4.4.7 Alberi BST: alcune osservazioni

Tutte le operazioni effettuate fino ad ora (ricerca, inserimento, ecc...) sono confinate ai nodi posizionati lungo ad un cammino semplice dalla radice alla foglia.

Durante tutte queste operazioni un fattore impattante è il **tempo di ricerca**.

Tempo di ricerca

Il **tempo di ricerca** è limitato superiormente da h , dove h è l'altezza dell'albero.



Come si può vedere dall'immagine, avendo un BST di altezza h , il caso pessimo è $O(h)$ dove è previsto di arrivare nella **parte più profonda dell'albero**. Il problema sorge quando si hanno n nodi: il **caso pessimo** della struttura dell'albero è quello che viene generato quando gli **elementi vengono inseriti in ordine** perché l'albero può "allungarsi". Ad esempio, si pensi ad un albero dove vengono inseriti i seguenti nodi: 1->2->3->4->5-> e così via ..., ovviamente tutti nel ramo destro. Quindi un albero contenente n nodi, avrà altezza $h = O(n)$ e **tempo di ricerca** $O(h)$. In questo caso si parla di **albero non bilanciato**.

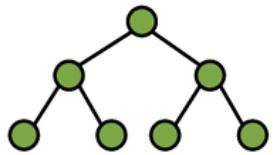
Gli alberi binari di ricerca, sono sì una buona idea per portare la ricerca binaria nel campo degli alberi, però se i dati sono inseriti in certi modi sbagliati (caso pessimo), si ottiene un'altezza dell'albero di $O(n)$ e quindi non si ha nessun vantaggio rispetto ad una banale **ricerca lineare** (liste).

Quando sono stati sviluppati gli alberi binari di ricerca, si è cercato di capire cosa succede in media, oltre che nel caso pessimo, e si è visto che facendo degli **inserimenti in ordine casuale**, l'altezza media dell'albero è $O(\log n)$. Nella realtà però, non ci si affida al caso ma si utilizzano delle **tecniche per mantenere l'albero bilanciato**, o per meglio dire, **ribilanciarlo**.

Cos'è un BST bilanciato?

Un BST "bilanciato" è un albero binario in cui l'altezza è proporzionale al logaritmo del numero di nodi ($h = \log_2(n + 1) - 1$).

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



Quindi, se l'albero è **ben bilanciato** l'altezza è limitata superiormente esattamente da $h = O(\log n)$. Quello che si vuole ottenere è un meccanismo per ribilanciare gli alberi non bilanciati e fare quindi in modo che la loro altezza sia più simile a $\log n$ piuttosto che n evitando casi particolari come il precedente. Un approccio possibile è quello degli **alberi AVL**.

4.4.8 Alberi BST: Alberi Adelson-Velsky and Landis (AVL)

Gli alberi AVL introducono alcune proprietà addizionali sugli alberi BST **per fare in modo che rimangano bilanciati**. Una di queste è il **fattore di bilanciamento**.

Fattore di bilanciamento

Il **fattore di bilanciamento** $\beta(v)$ di un nodo v è la differenza di altezza fra i sottoalberi destro e sinistro di v .

Cos'è un albero AVL?

Un albero binario di ricerca è un **albero AVL** se per ogni nodo v l'altezza del sottoalbero sinistro di v e quella del sottoalbero destro di v differiscono al massimo di 1 ($\beta(v) = h(left(v)) - h(right(v))$). In altre parole il **fattore di bilanciamento** deve essere un valore compreso tra $-1 \leq \beta(v) \leq 1$.

In un albero BST l'inserimento di una nuova foglia in un determinato punto dell'albero può causare uno **sbilanciamento**. Nello specifico il **nodo sbilanciato** è il primo antenato che, dopo l'inserimento del nuovo nodo, presenta un fattore di bilanciamento > 1 .

Per evitare gli sbilanciamenti viene utilizzato il concetto di **rotazione**.

Cos'è una rotazione?

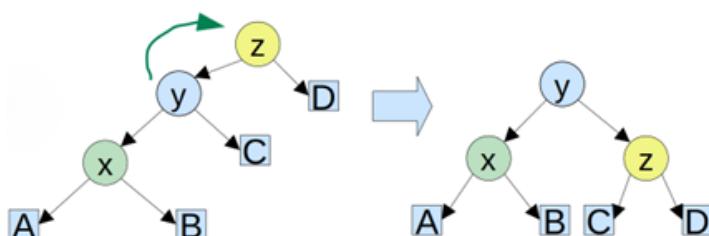
Una rotazione è un'operazione locale che viene eseguita sul **nodo sbilanciato** causando lo spostamento di tre nodi: il nodo sbilanciato stesso, suo figlio e il suo nipote.

Lo sbilanciamento viene riequilibrato **senza violare le proprietà strutturali** dei BST. Dunque, le **rotazioni** permettono di **abbassare** il fattore di bilanciamento.

Esistono solo due rotazioni elementari: **rotazione a sinistra** e **rotazione a destra**.

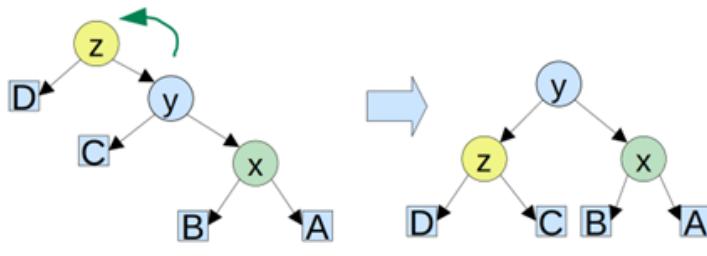
Invece, le **forme di sbilanciamento** di un albero BST sono quattro, e come detto prima, variano in base al punto di inserimento del nuovo nodo:

- Caso **left-left (LL)**: il caso **left-left** si verifica quando il nuovo nodo viene inserito nel sottoalbero sinistro del figlio sinistro del primo nodo, il quale diventa sbilanciato.



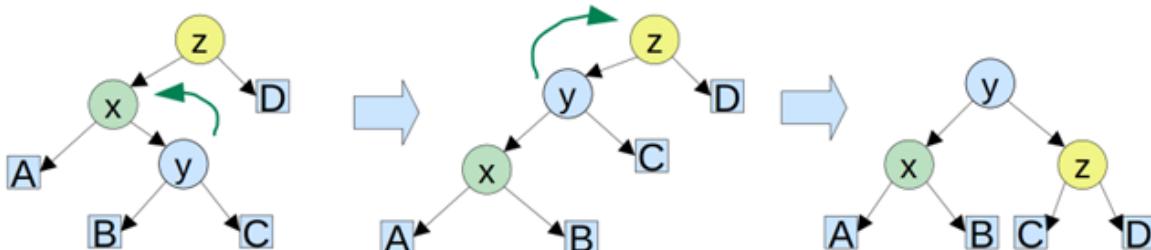
In questo caso il ramo sinistro "pesa" troppo e presenta un **fattore di bilanciamento** > 1 . Il caso **left-left** possiede una **forma lineare**, dunque, per ribilanciare l'albero basta effettuare una sola rotazione verso destra.

- Caso **right-right (RR)**: il caso **right-right** si verifica quando il nuovo nodo viene inserito nel sottoalbero destro del figlio destro del primo, il quale diventa sbilanciato



Anche in questo caso il ramo destro "pesa" troppo e presenta un **fattore di bilanciamento** < -1 . Il caso *right-right* possiede una **forma lineare**, e per ribilanciare l'albero, basta effettuare una sola rotazione verso sinistra.

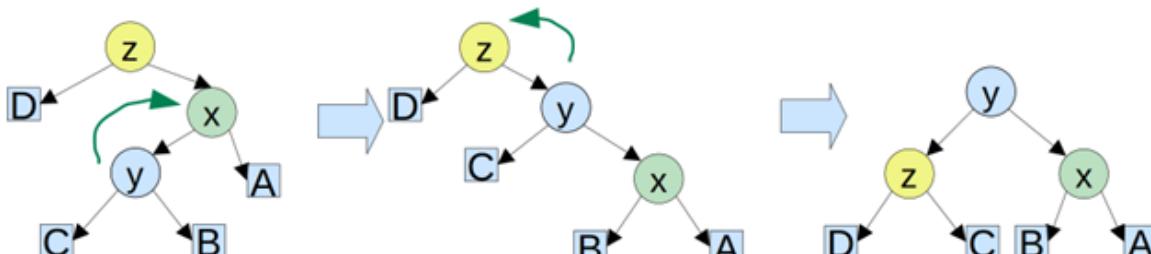
- Caso *left-right* (*LR*): il caso *left-right* si verifica quando il nuovo nodo viene inserito nel sottoalbero destro del figlio sinistro del primo nodo, il quale diventa sbilanciato.



Il fattore di bilanciamento in questo caso è > 1 . In questo caso il sottoalbero presenta una **forma a "zig-zag"** che, per essere bilanciata, necessita di una **doppia rotazione**:

1. **Rotazione a sinistra**: viene effettuata sul figlio destro del figlio sinistro del nodo sbilanciato, per fare in modo di riportarsi alla **forma left-left**;
2. **Rotazione a destra**: viene effettuata sul nodo sbilanciato per completare il bilanciamento.

- Caso *right-left* (*RL*): il caso *right-left* si verifica quando il nuovo nodo viene inserito nel sottoalbero sinistro del figlio destro del primo nodo, il quale diventa sbilanciato.



Presenta fattore di bilanciamento < -1 , e una forma a "zig-zag" che necessita di una doppia rotazione, in questo caso in senso inverso rispetto alla precedente:

1. **Rotazione a destra**: viene effettuata sul figlio sinistro del figlio destro del nodo sbilanciato, per fare in modo di riportarsi alla **forma right-right**;
2. **Rotazione a sinistra**: viene effettuata sul nodo sbilanciato per completare il bilanciamento.

Rotazione a sinistra/destra per il bilanciamento

```

1 bstree *rotateleft(bstree *x) {
2     bstree *y;
3     y = x->right;
4     x->right = y->left;
5     y->left = x;
6     return (y);
7 }
```

```

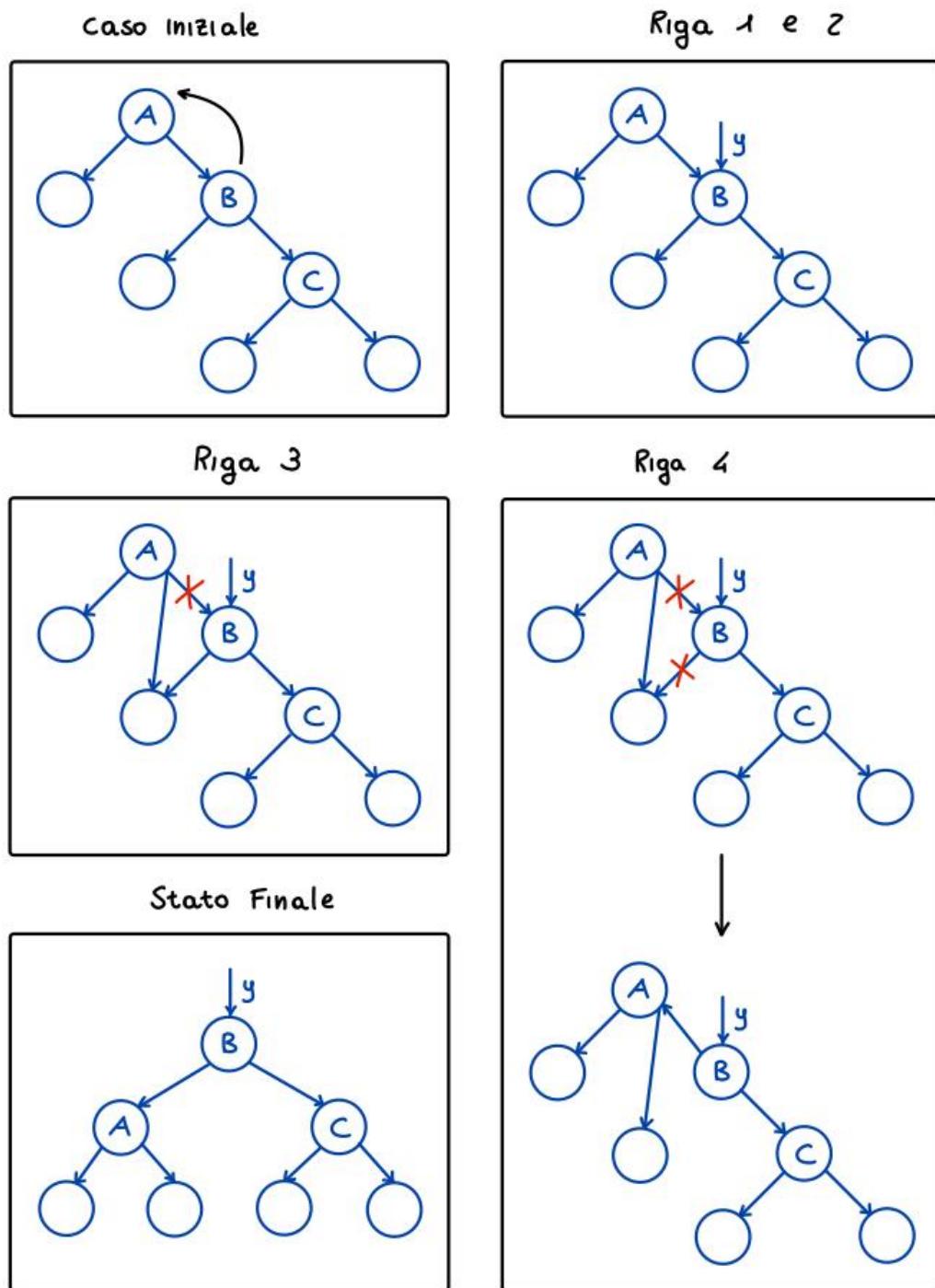
bstree * rotateright(bstree *x) {
    bstree *y;
    y = x->left;
    x->left = y->right;
    y->right = x;
    return (y);
}
```

- **Tipo di ritorno:** entrambe le funzioni ritornano un puntatore alla struttura di un nodo. Infatti, dopo la rotazione la radice dell'albero cambia: ciò che viene restituito è proprio la nuova radice dell'albero;
- **Parametri:** entrambe le funzioni accettano un unico parametro, ovvero il puntatore alla struttura del nodo che si vuole ruotare (`bstree *x`);
- **Funzionamento del codice:** la logica utilizzata per la creazione delle due funzioni di rotazione è la stessa, ciò che le differenzia è il verso della rotazione (sinistra o destra).

Funzionamento di `rotateleft` (e `rotateright`)

Per illustrare il funzionamento di `rotateleft` utilizziamo la seguente immagine.

Nel caso iniziale ciò che crea lo sbilanciamento è l'aggiunta del nodo *C*. Dunque deve essere effettuata una rotazione a sinistra specificando il puntatore al nodo *A* nei parametri della funzione. Ovviamente il funzionamento di `rotateright` è il medesimo, solo invertito.



5 Analisi della complessità degli algoritmi

Obiettivo

L'obiettivo dell'**analisi** degli algoritmi è quello di **stimare la loro complessità** in termini di **tempo di calcolo**.

Dunque, l'analisi della complessità degli algoritmi torna **utile** per **svariati motivi**:

- Stimare il tempo impiegato per un dato input;
- Stimare il più grande input gestibile in tempi ragionevoli;
- Confrontare l'efficienza di algoritmi diversi;
- Ottimizzare le parti più importanti.

5.1 Complessità e dimensione dell'input

Cos'è la complessità?

Possiamo definire la **complessità** come una funzione matematica che descrive l'andamento del tempo di calcolo in relazione alla **dimensione dell'input**.

$T : \text{"Dimensione dell'input"} \rightarrow \text{"Tempo di calcolo"}$

Dunque, maggiore è la dimensione dell'input, maggiore è il tempo di calcolo, con una **conseguente crescita delle risorse impiegate** dall'algoritmo per risolvere il problema.

Le due principali tipologie di risorse impiegate sono:

- **Risorse di complessità temporale**: cioè la quantità di tempo richiesta dall'algoritmo;
- **Risorse di complessità spaziale**: la quantità di memoria necessaria durante l'esecuzione.

In realtà, **complessità spaziale** diventa un problema secondario, che si andrà ad analizzare solo nel caso in cui, confrontando due algoritmi per determinarne il "*migliore*", abbiano la stessa complessità in termini di tempo.

Dimensione dell'input

Con dimensione dell'input intendiamo la sua **taglia**, e abbiamo due possibili casi:

- **Criterio di costo uniforme**: la taglia dell'input è il **numero di elementi di cui è costituito**. In altre parole, ogni elemento dell'input costa 1 unità, **indipendentemente** da quanti bit servono per rappresentarlo.

Esempio: per la ricerca del minimo in un vettore di n elementi, l'algoritmo che lo elabora avrà un costo **proporzionale a n** .

- **Criterio di costo logaritmico**: la taglia dell'input è il **numero di bit necessari per rappresentarlo**. In questo caso, il costo dell'elaborazione dipende direttamente dalla **lunghezza in bit** dei dati in ingresso, e non dal numero di elementi di cui è composto.
Esempio: avendo in ingresso un numero intero molto grande, si può considerare il numero di bit necessari per rappresentarli.

Nella pratica, se ogni elemento dell'input occupa un numero costante di bit, allora i due criteri (uniforme e logaritmico) forniscono risultati equivalenti, **a meno di una costante moltiplicativa** applicata alla dimensione dell'input. Ad esempio, un input costituito da n byte (criterio di costo uniforme) corrisponde a $8n$ bit (criterio di costo logaritmico).

5.2 Definizione di tempo e modello di calcolo

Tuttavia, come introdotto al capitolo 1.1.3, l'approccio più immediato per valutare il **tempo di esecuzione/calcolo** di un algoritmo, non è quello di misurare i secondi impiegati dal calcolatore poiché entrerebbero in gioco dei fattori esterni, non dipendenti dall'algoritmo stesso. Quindi misurando solo "quanti secondi impiega un'algoritmo", non si possono confrontare gli algoritmi in modo universale, ma solo "sul computer in quel determinato momento".

L'**analisi della complessità** vuole invece essere **indipendente dal calcolatore**, proprio per questo, un **approccio sicuramente migliore** è quello di considerare come "*tempo di calcolo*" il numero di istruzioni elementari eseguite.

Tempo \equiv numero istruzioni elementari

Un'**istruzione** viene considerata **elementare** se può essere eseguita in tempo "*costante*" dal processore (Il tempo di esecuzione **corrisponde** al numero di istruzioni elementari).

Cos'è il tempo di calcolo?

Poiché i problemi da risolvere hanno una dimensione che dipende dalla grandezza dei dati di ingresso, viene spontaneo esprimere il **tempo di calcolo** come: *il costo complessivo delle operazioni elementari in funzione della dimensione n dei dati in ingresso*.

Per poter capire quali istruzioni debbano essere considerate elementari, si utilizza il concetto di **modello di calcolo**, ovvero una rappresentazione semplificata ma rigorosa di un calcolatore, che definisce **quali operazioni sono ammesse e quanto costano**, tutto ciò in modo **indipendente dalle caratteristiche dell'hardware**.

Un buon modello di calcolo **soddisfa tre requisiti** fondamentali:

- **Astrazione**: deve permettere di **ignorare i dettagli irrilevanti** del calcolatore (ad esempio, non interessa conoscere la tipologia di processore e di quanta memoria disponga);
- **Realismo**: deve riflettere una situazione reale;
- **Potenza matematica**: deve consentire di trarre conclusioni matematiche (formali) sul costo computazionale.

5.2.1 Random Access Machine (RAM)

Il modello di calcolo che normalmente viene utilizzato è detto **Random Access Machine (RAM)** ed è caratterizzato da:

- **Memoria**: si assume che la memoria presenti una quantità infinita di celle di dimensione finita, poiché si vuole capire il funzionamento dell'algoritmo al crescere della dimensione dell'input, ciascuna delle quali è accessibile in tempo costante;
- **Processore (singolo)**: ha un processore singolo che esegue un insieme limitato di **istruzioni elementari** (come somma, sottrazione, moltiplicazione, operazioni logiche, salti condizionati, ecc...);
- **Costo delle istruzioni elementari**: ad ogni istruzione elementare viene assegnato, un **costo costante**, che rappresenta il tempo richiesto per eseguirla.

Lo scopo finale non è confrontare le prestazioni dei processori (compito dei benchmark), ma determinare se un algoritmo è più efficiente di un altro.

5.3 Valutazione del caso pessimo, medio e ottimo

Principalmente, il costo delle singole operazioni è valutato nel **caso pessimo**, ovvero sul dato di ingresso più sfavorevole tra tutti quelli di dimensione n .

In alternativa, si può considerare anche il **caso medio**, calcolando la media dei costi su tutti i possibili input di dimensione n , pesata in base alla probabilità con cui ciascun dato può verificarsi. Mentre, talvolta si introduce anche il **caso ottimo** che rappresenta il costo minimo dell'algoritmo su un input di dimensione n .

Il caso ottimo è **raramente utile**: infatti un buon comportamento nel caso ottimo **non garantisce prestazioni accettabili negli altri casi** e può dare un'**illusione di efficienza** non rappresentativa del comportamento complessivo dell'algoritmo.

Perché valutiamo il caso pessimo se può verificarsi molto raramente?

Il vantaggio del caso pessimo è dato dal fatto che questo tipo di valutazione non richiederà mai, per nessun dato di dimensione n , un tempo maggiore.

Invece, la valutazione nel caso medio sembra più realistica, ma è ignota la distribuzione di probabilità: spesso viene utilizzata una distribuzione uniforme che per molti problemi è irrealistica

5.3.1 Tempo di calcolo della funzione `min()` (iterativa)

In questo caso si vuole stimare il tempo di calcolo della funzione `min()` che si occupa di trovare l'elemento più piccolo all'interno di un vettore. Per prima cosa si controlla il numero delle operazioni elementari dalla quale è costituita la funzione in esame.

A questo punto possiamo:

- Indicare con C_h il costo richiesto per l'esecuzione dell'istruzione h -esima, perché non so esattamente quante operazioni macchina servano per eseguire l'istruzione (colonna "costo");
- Inoltre, effettuando una valutazione nel **caso pessimo**, per ogni h -esima istruzione si specifica il **massimo numero di volte** che questa viene eseguita (colonna "# Volte").

item `min(item[] A, int n)`

```

item min = A[0]
for i = 1 to n - 1 do
    if A[i] < min then
        min = A[i]
return min

```

	Costo	# Volte
item min = A[0]	c_1	1
for i = 1 to n - 1 do	c_2	n
if A[i] < min then	c_3	$n - 1$
min = A[i]	c_4	$n - 1$
return min	c_5	1

N.B: per quanto riguarda il ciclo for, è giusto scrivere che viene eseguito n volte, poiché anche la $n - 1$ esima volta la riga viene eseguita prima di capire che la condizione non è rispettata. Infatti come si può notare, le istruzioni al suo interno vengono eseguite $n - 1$ volte.

Dunque, il tempo di calcolo $T(n)$ di `min()` si ottiene sommando il prodotto del costo di ciascuna istruzione per il numero di volte che è stata eseguita:

$T(n) = c_1 + c_2(n) + c_3(n - 1) + c_4(n - 1) + c_5 = (c_2 + c_3 + c_4)n + (c_1 + c_5 - c_3 - c_4) = an + b$
 (Posso scrivere $an + b$ perché non si conosce il valore dei costi da $c_1 \dots c_5$).

Osservazioni sull'identificazione del caso

Per distinguere caso ottimo, medio e pessimo, occorre capire come varia il numero di operazioni dell'algoritmo al variare dell'input.

Nel caso della funzione `min()`, il numero di iterazioni del ciclo e il numero di confronti eseguiti sono **indipendenti dai valori dell'input**: il `for` viene sempre eseguito n volte e il confronto viene sempre effettuato $n - 1$ volte.

Di conseguenza, la funzione presenta un tempo di esecuzione $T(n)$ lineare sia nel caso pessimo sia nel caso medio.

5.3.2 Tempo di calcolo della funzione `binarySearch()` (ricorsiva)

In quest'altro caso si considera la funzione `binarySearch()` che si occupa di ricercare **la posizione** di un elemento all'interno di una sequenza ordinata, memorizzata in un vettore A . La logica che sta dietro la **ricerca binaria**, è la stessa che viene utilizzata per la ricerca di un nodo negli alberi binari: ogni volta che viene richiamata la funzione in modo ricorsivo, si elimina metà del vettore (**ricerca dicotomica** - capitolo 4.4).

int binarySearch(ITEM[] A, ITEM v, int i, int j)

	Costo	# ($i > j$)	# ($i \leq j$)
if $i > j$ then	c_1	1	1
return 0	c_2	1	0
else			
int $m \leftarrow \lfloor (i + j)/2 \rfloor$	c_3	0	1
if $A[m] \leftarrow v$ then	c_4	0	1
return m	c_5	0	0
else if $A[m] < v$ then	c_6	0	1
return <code>binarySearch(A, v, m + 1, j)</code>	$c_7 + T(\lfloor n/2 \rfloor)$	0	0/1
else			
return <code>binarySearch(A, v, i, m - 1)</code>	$c_7 + T(\lfloor (n - 1)/2 \rfloor)$	0	1/0

A differenza del caso precedente (capitolo 5.3.1), l'algoritmo esegue porzioni di codice differenti a seconda dei valori di input (i e j), che sono rispettivamente, l'**indice iniziale** del vettore e l'**indice finale** del vettore. Per avere una valutazione del tempo di calcolo si andrà a valutare il **caso pessimo** di **entrambe le porzioni** di codice, inserendo due colonne etichettate con "#", che indicano quante volte ciascuna riga viene eseguita :

- $i > j$: questa porzione di codice esegue direttamente la condizione di chiusura poiché se i è maggiore di j si sta indicando un insieme di elementi nullo in cui la posizione dell'elemento non può esistere.

In questo caso si può vedere come, nella colonna $\#(i > j)$, vengono eseguite solo le righe con costo c_1 e c_2 , poiché subito dopo la ricorsione termina, di conseguenza tutte le altre righe vengono eseguite 0 volte. Dunque il tempo di calcolo sarà dato da: $T(n) = c_1 + c_2 = c$, dove n è zero perché non esiste una grandezza (n) specifica per il vettore.

- $i < j$: in quest'altra porzione di codice il vettore viene suddiviso in due parti **sinistra** e **destra**. La parte sinistra ha grandezza $\lfloor (n - 1)/2 \rfloor$, mentre la parte destra $\lfloor n/2 \rfloor$.

Il caso pessimo prevede la ricerca di un elemento **maggiore del massimo contenuto** nel vettore, dunque v sarà sempre maggiore di $A[m]$ e l'algoritmo sceglierà sempre la parte di vettore più grande ($[n/2]$).

i=1	m=5	j=9						
5	14	35	38	60	83	94	143	180

i=6	m=7	j=9						
5	14	35	38	60	83	94	143	180
							i=8	j=9
5	14	35	38	60	83	94	143	180

m=8

(m), fino ad esaurire tutti gli elementi. Il costo della funzione è quindi: $T(n) = c_1 + c_2 + c_4 + c_6 + c_7 + T(\frac{n}{2}) = d + T(\frac{n}{2})$ dove d rappresenta la somma da $c_1 \dots c_7$.

Questo significa che l'equazione non è definita in modo semplice da un solo valore, ma da una **relazione di ricorrenza** (capitolo 5.5). Una tecnica che viene utilizzata per risolvere le

$$T(n) = \begin{cases} c & \text{se } n = 0 \\ T(\frac{n}{2}) + d & \text{se } n \geq 1 \end{cases}$$

relazioni di ricorrenza consiste nel **produrre una catena di uguaglianze** ottenute per sostituzioni successive.

Infatti possiamo scrivere che $T(n) = T(n/2) + d =$

$T(n/4) + 2d = \dots = T(n/2^k) + kd$, da cui possiamo ricavare che $\frac{n}{2^k} \Rightarrow k = \log n$.

Andando avanti con la ricorrenza si arriverà ad un punto in cui $T(n/2^k) + kd = T(1) + kd = [T(0) + d] + kd = T(0) + d(k + 1) = c + kd + d = d \log n + (c + d)$.

5.4 Ordini di complessità

Dopo aver analizzato gli algoritmi al capitolo 5.3 sono state ottenute due **funzioni di complessità**, con una **serie di parametri** che **non si è in grado di determinare**. Questa difficoltà viene aggirata utilizzando il concetto di **crescita asintotica**.

Concetto di crescita asintotica

Quando analizziamo un algoritmo, non ci interessa tanto il tempo preciso di esecuzione, ma **come cresce questo tempo** quando la dimensione dell'input n diventa molto grande.

- **Crescita lineare:** min: $an + b \Rightarrow O(n)$
- **Crescita logaritmica:** BinarySearch(): $d \log n + (c + d) \Rightarrow O(\log n)$

Questo concetto è utile perché permette di **confrontare algoritmi diversi** senza **preoccuparsi** delle costanti (che dipendono dall'hardware o dall'implementazione).

Permette di capire **quale algoritmo sarà più efficiente** su input grandi, anche se su input piccoli può sembrare più lento.

Esempio intuitivo: Supponiamo di avere due algoritmi.

- **Algoritmo A:** richiede $100n$ operazioni \rightarrow complessità $O(n)$;
- **Algoritmo B:** richiede n^2 operazioni \rightarrow complessità $O(n^2)$

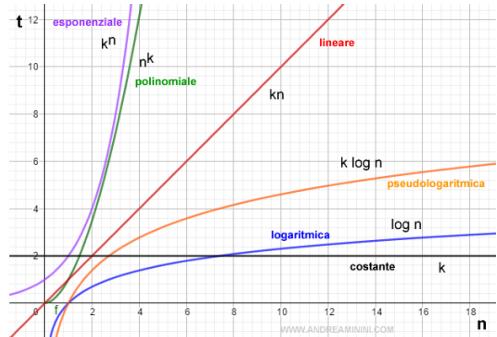
- Per $n = 10$, A fa 1000 operazioni, mentre B ne fa 100;
 - Per $n = 100$, A fa 100.000 operazioni, mentre B ne fa 1.000.000;
- Quindi asintoticamente A è il migliore, anche se su input piccolo B sembrava il migliore.

5.4.1 Principali classi di efficienza asintotiche

Nella seguente tabella vengono mostrati il numero di passi necessari per completare l'algoritmo in base alla complessità e alla dimensione dell'input.

$f(n)$	$n = 10^1$	$n = 10^2$	$n = 10^3$	$n = 10^4$	Tipo
$\log n$	3	6	9	13	logaritmico
n	10	100	1.000	10.000	lineare
$n \log n$	30	664	9.965	132.877	loglineare
n^2	10^2	10^4	10^6	10^8	quadratico
n^3	10^3	10^6	10^9	10^{12}	cubico
2^n	1.024	10^{30}	10^{300}	10^{3000}	esponenziale

(a)



(b)

- **Complessità logaritmica:** Tipicamente, il risultato di ridurre le dimensioni del problema di un fattore costante ad ogni iterazione. N.B: un algoritmo in questa classe di efficienza non può tenere conto di tutto il suo input, altrimenti avrebbe efficienza lineare;
- **Complessità lineare:** Algoritmi che effettuano un numero costante di iterazioni sull'input (ad esempio una ricerca sequenziale);
- **Complessità loglineare (o superlineare):** Algoritmi che necessitano di effettuare almeno una scansione completa dell'input, ma che riducono di un fattore costante le iterazioni intermedie (ad esempio gli algoritmi divide-et-impera);
- **Complessità quadratica:** Tipica degli algoritmi che sono basati su due iterazioni annidate. Ad esempio, alcuni algoritmi di ordinamento che effettuano operazioni su matrici $n \cdot n$;
- **Complessità cubica:** Tipica degli algoritmi che sono basati su tre iterazioni annidate, diversi algoritmi di algebra lineare ricadono in questa classe;
- **Complessità esponenziale:** Algoritmi che effettuano ricerca sui sottoinsiemi di un insieme di n elementi;
- **Complessità fattoriale:** Algoritmi che effettuano ricerca su permutazioni di un insieme di n elementi.

5.4.2 Notazioni asintotiche

Per **descrivere la crescita asintotica** di un algoritmo, quindi come cresce il tempo di esecuzione dell'algoritmo al crescere dell'input n , vengono utilizzate delle **notazioni**.

Notazione Big-O (O grande)

Describe il **limite superiore** della crescita di un algoritmo.

- Garantisce che il tempo di calcolo non esploderà oltre una certa funzione;
- Di conseguenza, viene utilizzata per **descrivere il caso peggiore**.

Esempio: se un algoritmo è $O(n)$, significa che al crescere dell'input non farà mai più di un numero di operazioni proporzionale a n .

Notazione Omega (Ω grande)

Describe il **limite inferiore** della crescita di un algoritmo.

- Indica il lavoro minimo che l'algoritmo deve svolgere, anche nel caso migliore;
- Serve a capire che **sotto una certa soglia di complessità non si può scendere**.

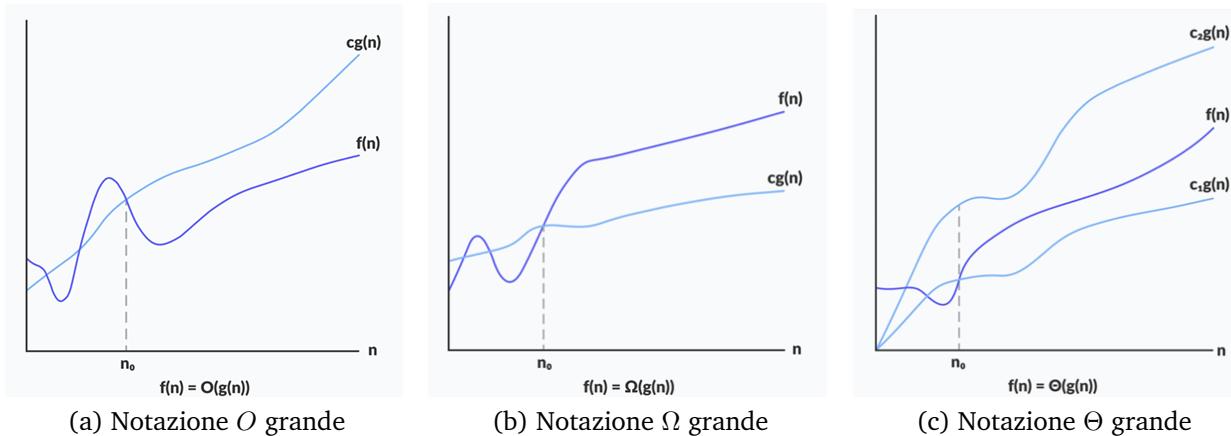
Esempio: se un algoritmo è $\Omega(n)$, significa che anche nel caso più favorevole deve comunque guardare almeno n elementi.

Notazione Theta (Θ grande)

Describe i due **limiti (inferiore e superiore)** della crescita di un algoritmo.

- Indica la **crescita reale** dell'algoritmo;
- Torna utile quando sappiamo che un algoritmo cresce esattamente come una certa funzione, permettendo di **classificarlo con precisione**.

Esempio: se un algoritmo è $\Theta(n \log n)$, significa che cresce proprio in quel modo: non più veloce, non più lento.



5.5 Le ricorrenze

Quando un algoritmo contiene una **chiamata ricorsiva a se stesso** il suo tempo di esecuzione spesso può essere descritto attraverso **una ricorrenza**.

Cos'è una ricorrenza?

Una **ricorrenza** è un'equazione che descrive una funzione (tipicamente il tempo di esecuzione $T(n)$) in termini del suo stesso valore calcolato su input più piccoli.

Le ricorrenze nascono quando un algoritmo ricorsivo **divide il problema in sottoproblemi più piccoli e richiama se stesso**.

Infatti, come è stato visto al capitolo 5.3.2, l'algoritmo `binarySearch()` dimezza il problema ad ogni passo tramite la seguente ricorrenza: $T(n) = T(\frac{n}{2}) + d$, che abbiamo visto avere complessità logaritmica $O(\log n)$.

Quindi, è possibile esprimere il tempo di esecuzione viene espresso come **la somma di**:

- Il **costo per dividere o combinare** il problema (nel caso precedente d);
- Il **costo per le chiamate ricorsive** su sottoproblemi più piccoli (nel caso precedente $T(\frac{n}{2})$).

Per risolvere le ricorrenze è possibile **utilizzare tre metodi** differenti: metodo di **sostituzione**, metodo dell'**esperto** o metodo dell'**albero di ricorsione**.

5.5.1 Metodo di sostituzione (o per tentativi)

Idea di base

Il metodo della sostituzione consiste nell'**ipotizzare la forma della soluzione** per poi utilizzare l'**induzione** matematica per **dimostrare che la soluzione ipotizzata funziona**.

Il metodo di sostituzione puo essere usato per determinare il limite inferiore o superiore di una ricorrenza, e **la sua applicazione ha senso** solo nei casi in cui è "facile" immaginare la **forma** della soluzione.

Ad esempio, una prima ipotesi può essere effettuata nei casi in cui l'algoritmo **rispecchia una delle descrizioni** di complessità (logaritmica, lineare, loglineare, ecc...) - Capitolo 5.4.1.

Una volta fatta l'ipotesi, si procede a produrre una **catena di ugianze** ottenute per **sostituzioni successive** (come già fatto al capitolo 5.3.2) cercando di arrivare all'ipotesi effettuata in precedenza.

Osservazioni

È importante notare che non esiste un metodo generale per formulare l'ipotesi della soluzione corretta di una ricorrenza, infatti se quest'ultima è simile ad una già vista, **ha senso provare una soluzione analoga**.

5.5.2 Metodo dell'esperto

Idea di base

Il metodo dell'esperto (Master Theorem) è una "**formula pronta**" per un'**intera famiglia di ricorrenze** tipiche, che dividono un problema di dimensione n in a sottoproblemi di dimensione n/b .

La seguente formula $T(n) = aT(n/b) + f(n)$ dove $a \geq 1$ e $b > 1$ fornisce subito il risultato:

- Ogni a -esimo sottoproblema viene risolto nel tempo $T(n/b)$;
 - Il costo per dividere il problema e combinare i risultati dei sottoproblemi è descritto da $f(n)$
- Dunque, questa metodologia di soluzione per le ricorrenze **risulta perfetta solo per algoritmi classici** come mergesort, binary search, quicksort, ecc...

Proprio perché viene utilizzata per una famiglia di algoritmi tipici di cui si conosce l'andamento reale del caso pessimo $\Theta(\dots)$, è possibile **individuare tre casi** in cui l'andamento dell'algoritmo può ricadere:

Date le costanti intere $a \geq 1$ e $b \geq 2$ e le costanti reali $c > 0$ e $\beta \geq 0$. Sia $T(n)$ data dalla relazione di ricorrenza:

$$T(n) = \begin{cases} c & \text{se } n \leq 1 \\ aT(n/b) + cn^\beta & \text{se } n > 1 \end{cases} \quad \text{con } \alpha = \frac{\log a}{\log b} = \log_b a : \quad T(n) = \begin{cases} \Theta(n^\alpha) & \alpha > \beta \\ \Theta(n^\alpha \log n) & \alpha = \beta \\ \Theta(n^\beta) & \alpha < \beta \end{cases}$$

Esempio del Caso 1

Immaginiamo di avere la ricorrenza $T(n) = 9T(n/3) + n$, si ha dunque:

- $a = 9, b = 3$
- $\alpha = \log_3 9 = 2$
- $\beta = 1$

Quindi $\alpha > \beta \Rightarrow T(n) = \Theta(n^\alpha) = \Theta(n^2)$

Esempio del Caso 2

Immaginiamo di avere la ricorrenza $T(n) = T(n/3) + 1$, si ha dunque:

- $a = 1, b = 3$
- $\alpha = \frac{\log 1}{\log 3} = 0$
- $\beta = 0$

Quindi $\alpha = \beta \Rightarrow T(n) = \Theta(n^\alpha \log n) = \Theta(\log n)$

Esempio del Caso 3

Immaginiamo di avere la ricorrenza $T(n) = 3T(n/4) + n$, si ha dunque:

- $a = 3, b = 4$
- $\alpha = \frac{\log 4}{\log 3} = 0.793$
- $\beta = 1$

Quindi $\alpha < \beta \Rightarrow T(n) = \Theta(n^\beta) = \Theta(n)$

5.5.3 Metodo dell'albero di ricorsione (analisi per livelli)

Idea di base

La **ricorsione** viene immaginata **come un albero**: ogni **nodo** è un **sottoproblema** con un determinato costo, e **ogni livello** dell'albero rappresenta un **passo** della ricorsione.

Quindi, si calcola il **costo di ogni livello** (somma dei costi dei singoli nodi per quel livello), e poi **si sommano tutti i livelli**. Possiamo immaginarlo come un processo di questo tipo:

- **Livello 0:** abbiamo l'espressione originale;
- **Livello 1:** espressione a cui è stata applicata un'espansione;
- **Livello 2:** vengono applicate due espansioni;
- ... si continua fino al caso base.

Livello	Espressione	Costo
0	n^2	$4^0 n^2$
1	$\frac{n^2}{2^2} \frac{n^2}{2^2} \frac{n^2}{2^2} \frac{n^2}{2^2}$	$4^1 \frac{n^2}{2^2}$
2	$\frac{n^2}{2^4} \frac{n^2}{2^4} \dots \frac{n^2}{2^4} \frac{n^2}{2^4}$	$4^2 \frac{n^2}{2^4}$
	...	
i	$\frac{n^2}{2^{2i}} \frac{n^2}{2^{2i}} \dots \frac{n^2}{2^{2i}} \frac{n^2}{2^{2i}}$	$4^i \frac{n^2}{2^{2i}}$
	...	
h	$T(1) T(1) \dots T(1) T(1)$	4^h

Per comprendere meglio il concetto si consideri la ricorrenza:

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \\ 4T(n/2) + n^2 & n = 2^h, h > 0 \end{cases}$$

Dunque, il **costo complessivo** di ciascun livello dell'albero (escluso l'ultimo che è il caso base) è sempre n^2 : i sottoproblemi sono più piccoli ma più numerosi. Invece, $4T(n/2)$ è il costo per la chiamata ricorsiva che genera 4 sottoproblemi di dimensione $n/2$.

- **Livello 0 (radice)**

- Si ha un solo problema di dimensione n .

- **Livello 1**

- Il problema si divide in 4 sottoproblemi (coefficiente 4);
- Il costo di ciascuno è $n^2/4$;
- Ognuno ha dimensione $n/2$;
- Costo totale del livello $4 \cdot (n^2/4) = n^2$

- **Livello 2**

- Ogni sottoproblema del livello 1 si divide di nuovo in 4 → in totale $4^2 = 16$ sottoproblemi;
- Il costo di ciascuno è $n^2/16$;
- Ognuno ha dimensione $n/4$;
- Costo totale del livello $16 \cdot (n^2/16) = n^2$

Quindi, seguendo il pattern, il numero di sottoproblemi cresce ad ogni livello di 4^i e la dimensione di ognuno diminuisce di $n/2^i$.

In questo caso specifico è facile capire la complessità della ricorrenza, poiché **il numero di sottoproblemi cresce esponenzialmente** (4^i), ma allo stesso momento la loro dimensione cala esponenzialmente ($n/2^i$) e questo permette al costo totale di ogni livello di rimanere costante (n^2), ma ripetuto per tutti i livelli. Dunque, la ricorrenza ha complessità $O(n^2 \log n)$.

Osservazioni

Grazie al fatto che la ricorsione viene suddivisa in livelli, questo approccio risulta molto utile per capire intuitivamente dove si concentra il costo (in alto, in basso o se è distribuito su tutti i livelli).

5.6 Ordinamento

Dato un generico problema, è possibile progettare un gran numero di **algoritmi differenti** per la sua risoluzione, ognuno caratterizzato dal proprio tempo di calcolo: alcuni molto lenti, altri molto veloci.

- Gli algoritmi con **complessità esponenziale** (tipo 2^n) **non sono accettabili** come soluzione nel momento in cui si lavora con **input grandi**, a meno che non sia possibile dimostrare che il problema posto sia inerentemente difficile;
- Invece, realizzando un'algoritmo di **complessità polinomiale** (tipo n^2 o $n \cdot \log n$), si ha già fatto un buon lavoro, ma si cerca comunque di "*abbassarne la complessità*".

Obbiettivo

In questi casi l'**obbiettivo principale** è quello di valutare gli algoritmi in base alla **tipologia dell'input**. Ovviamente gli algoritmi "migliori" sono quelli che garantiscono tempi di esecuzione accettabili anche su input molto grandi.

Infatti, in alcuni casi, gli **algoritmi si comportano diversamente** in base alle **caratteristiche dell'input** e conoscere in anticipo tali caratteristiche permette di scegliere il miglior algoritmo per quella determinata situazione.

Il **problema dell'ordinamento** è un buon esempio per mostrare questi concetti, proprio perché **comparando in tanti contesti** e avendo **soluzioni diverse** è perfetto per scegliere l'implementazione migliore in base all'input.

Problema dell'ordinamento (sorting)

Dato un vettore di n elementi, il problema dell'ordinamento prevede la **permutazione del vettore** per fare in modo che i suoi elementi compaiano in **ordine non decrescente**.

Se si volesse utilizzare un **approccio "demente"**, si potrebbero generare tutte le possibili permutazioni fino a che non se ne trova una già ordinata, in questo caso:

- Per verificare che un vettore A sia ordinato, basta un ciclo `for`, quindi richiede $O(n)$ tempo;
- Il numero di permutazioni possibili è $n!$.

Dunque, procedendo il questo modo si avrebbe una complessità $O(n \cdot n!)$, ovvero una **complessità superpolinomiale**. Per evitare ciò, l'approccio migliore è quello di utilizzare degli **algoritmi di ordinamento**, decisamente più adatti a questa tipologia di problema.

5.6.1 Selection Sort

Il selection sort è un semplice algoritmo polinomiale che è basato sulla proprietà che in una sequenza ordinata, il primo elemento ha valore minimo.

Algoritmo selection sort

In un **algoritmo selection sort** si cerca il minimo e si scambia tale elemento con quello nella prima posizione della **parte non ordinata del vettore**, riducendo il problema agli $n - 1$ restanti valori.

Dato un input del tipo $A = \{7, 4, 2, 1, 8, 3, 5\}$ con $n = 7$, l'algoritmo `selectionSort()` si comporta nel seguente modo:

Algorithm `selectionSort(Item[] A, int n)`

```

1: for  $i = 1$  to  $n - 1$  do
2:   int  $j = \min(A, i, n)$ 
3:    $A[i] \leftrightarrow A[j]$ 
4: end for
```

7	4	2	1	8	3	5
---	---	---	---	---	---	---

1	4	2	7	8	3	5
---	---	---	---	---	---	---

1	2	4	7	8	3	5
---	---	---	---	---	---	---

Algorithm `int min(Item[] A, int k, int n)`

```

1: int  $min = k$ 
2: for  $h = k + 1$  to  $n$  do
3:   if  $A[h] < A[min]$  then
4:      $min = h$ 
5:   end if
6: end for
7: return  $min$ 
```

1	2	3	7	8	4	5
---	---	---	---	---	---	---

1	2	3	4	8	7	5
---	---	---	---	---	---	---

1	2	3	4	5	7	8
---	---	---	---	---	---	---

1	2	3	4	5	7	8
---	---	---	---	---	---	---

(a)

(b)

Complessità del selectionSort()

In questo particolare algoritmo **non importa quale sia l'ordine iniziale dell'input** dato. Questo perché lo **spostamento di un valore** ha sempre **costo costante**.

L'algoritmo `selectionSort()` esegue un ciclo esterno che va da $i = 1$ a $n - 1$, e ad ogni iterazione, chiama la funzione `min()` su un sottoarray da i a n .

La funzione `min()` esegue un ciclo interno che fa $n - i$ confronti, quindi sempre meno confronti man mano che il vettore si riordina.

Sapendo che la somma dei primi k numeri naturali è $1 + 2 + \dots + k = \frac{k(k+1)}{2}$ possiamo dire che:

$$\sum_{k=1}^{n-1} k = \sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

Questo cresce come n^2 , quindi la complessità è quadratica $O(n^2)$, anche se l'array è già ordinato, perché deve **comunque cercare il minimo** nel sottoarray, anche se è già al posto giusto.

5.6.2 Insertion sort

L'algoritmo `insertionSort()` è un'algoritmo efficiente per ordinare piccoli insiemi di elementi. Si basa sul principio di ordinamento di una mano di carte da gioco.

Algoritmo insertion sort

Per seguire l'analogia, si considerino le carte una per volta, ad esempio, da sinistra verso destra: ogni volta che viene considerata una nuova carta, si inserisce nella posizione giusta rispetto alle altre carte già considerate e ordinate, traslando di una posizione verso destra tutte le carte maggiori.

Algorithm `insertionSort(Item[] A, int n)`

```
1: for  $i = 2$  to  $n$  do
2:   Item  $temp = A[i]$ 
3:   int  $j = i$ 
4:   while  $j > 1$  and  $A[j-1] > temp$  do
5:      $A[j] = A[j-1]$ 
6:      $j = j - 1$ 
7:   end while
8:    $A[j] = temp$ 
9: end for
```

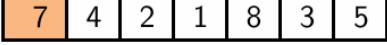
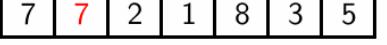
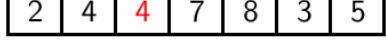
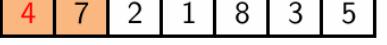
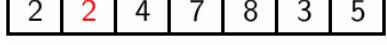
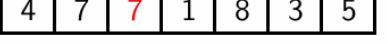
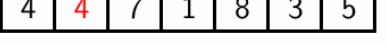
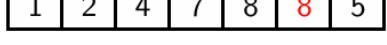
Immaginiamo di avere un input del tipo $A = \{2, 3, 1, 4, 7, 3\}$.

L'**idea di base** è la seguente: se si prende solo il primo elemento [2], in se per sé è già ordinato, potrebbe anche non essere la sua posizione giusta, ma preso come sottovettore è ordinato.

Vado poi a vedere il valore successivo [3] e lo confronto con il valore [2] (che è l'ultimo del vettore ordinato): [3] > [2] quindi la parte iniziale del vettore è ordinata $\{2, 3, \dots\}$.

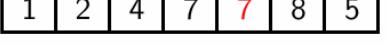
Andando avanti si troverà [1], più piccolo sia di [2] che di [3]. L'elemento minore viene messo in una variabile temporanea (`temp`), mentre gli elementi della parte già ordinata del vettore vengono traslati (partendo dal più grande) uno ad uno verso destra di una posizione, sovrascrivendo l'elemento alla propria destra. A questo punto all'inizio del vettore si è creato uno spazio vuoto [...] (dovuto alla traslazione della parte ordinata del vettore) su cui andrà inserita la variabile `temp` contenente l'elemento precedentemente sovrascritto, nonché il più piccolo elemento del vettore esplorato fino a quel momento.

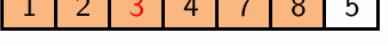
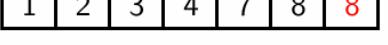
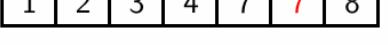
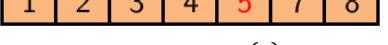
Ovviamente, nel caso in cui l'elemento da inserire vada posizionato nel mezzo del vettore ordinato, durante la traslazione dei singoli elementi ci si fermerebbe nel punto in cui `temp` sia maggiore dell'elemento `A[i]`. Ad esempio, si consideri un input del tipo $A = \{7, 4, 2, 1, 8, 3, 5\}$.

(a)

(b)



(c)

Per la logica che sta dietro al `selection sort()`, anche se l'input iniziale è già ordinato, l'algoritmo andrà a cercare comunque il valore minimo e lo sovrascriverà nella prima posizione della parte non ordinata dell'array, quindi, come detto al capitolo (capitolo 5.6.1), la complessità dell'algoritmo non dipende dall'input di ingresso.

Invece, a differenza del `selection sort()`, la **complessità** dell'`insertion sort()` dipende dalla **disposizione iniziale dei dati in ingresso** poiché, per come è strutturato l'algoritmo, man mano che si scorre l'array si andranno a posizionare i nuovi elementi nella posizione corretta rispetto a tutti gli altri.

Proprio per questa caratteristica, la complessità dell'`insertion sort()` cambia a seconda del caso di studio (capitolo 5.3):

- **Complessità nel caso pessimo:** il caso peggiore si ha quando l'input A è **ordinato alla rovescia**, ad esempio: $A = \{8, 7, 6, 5, 4, 3, 2, 1\}$.

In questo caso ad ogni iterazione del ciclo `for`, si entrerebbe anche nel ciclo `while` più interno poiché la variabile `temp` andrebbe spostata in prima posizione, e ciò richiederebbe un numero di spostamenti pari a

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

Complessità? Dunque la complessità sarebbe uguale a quella del `selection sort()` per la presenza dell'annidamento dei due cicli $\rightarrow O(n^2)$.

- **Complessità nel caso ottimo:** il caso migliore si ha quando l'input A è una sequenza già ordinata, quindi $A = 1, 2, 3, 4, 5, 6, 7, 8$

In questo caso, non avverà mai che $A[j-1] > A[j]$ sia maggiore di temp e così via..., non permettendo al ciclo while di verificarsi. Rimane solo il ciclo esterno, che esegue operazioni di costo costante.

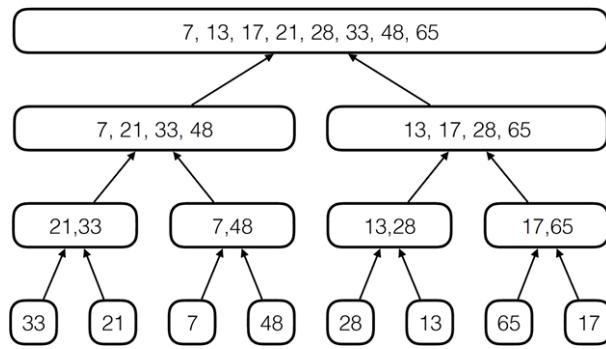
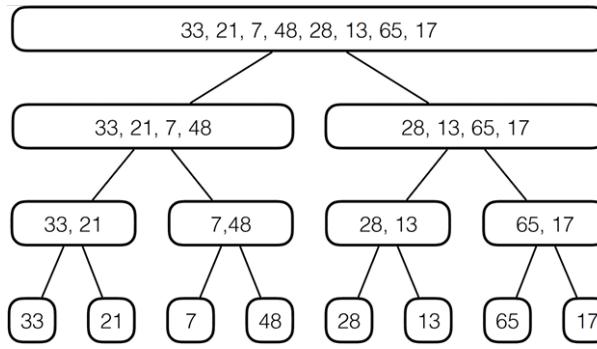
Complessità? In questo caso, la complessità per delle istruzioni costanti vale $O(n)$.

5.6.3 Merge sort

L'algoritmo MergeSort() è basato sulla tecnica **divide et impera**.

Algoritmo merge sort

- **Divide:** spezza il vettore di n elementi in due sottovettori di $n/2$ elementi;
- **Impera:** chiama MergeSort() ricorsivamente sui due sottovettori;
- **Combina:** una volta ottenuti singoli valori, questi vengono riuniti (merge) in modo ordinato, fino a riottenere i due sottovettori iniziali ordinati.



Algorithm mergeSort(Item[]A, int first, int last)

```

1: if first < last then
2:   int mid = ⌊(first + last)/2⌋
3:   mergeSort(A, first, mid)
4:   mergeSort(A, mid+1, last)
5:   merge(A, first, last, mid)
6: end if

```

La prima parte dell'algoritmo mergeSort() si occupa di effettuare le chiamate ricorsive per la suddivisione del vettore non ordinato fino ai singoli valori (come in figura d). Prendiamo in considerazione il vettore: $A = [4, 3, 2, 1]$.

Nella riga 2 viene calcolata la metà del vettore (se il vettore è dispari si andrà per eccesso).

In questo caso, il vettore viene diviso in parte sinistra [4, 3] e parte destra [2, 1].

Prima ricorsione (riga 2)

Una volta che il vettore è stato diviso in due parti, la prima istruzione ricorsiva (riga 3) richiama la funzione mergeSort(A, first, mid) per poter lavorare sulla parte sinistra, ossia [4, 3]. Dunque, la riga 2 viene rieseguita e il sottovettore viene ulteriormente suddiviso in:

- parte sinistra [4];
- parte destra [3].

Subito dopo, le istruzioni ricorsive vengono tentate nuovamente, ma:

- mergeSort(A, first, mid) non prosegue poiché il sottovettore ha un solo elemento [4];
- anche, mergeSort(A, mid+1, last) non prosegue perché anche [3] è un singolo elemento.

Raggiunto il caso base per entrambi i sottovettori, è possibile richiamare `merge()` per combinarli e ottenere il vettore ordinato [3, 4].

Seconda ricorsione (riga 3)

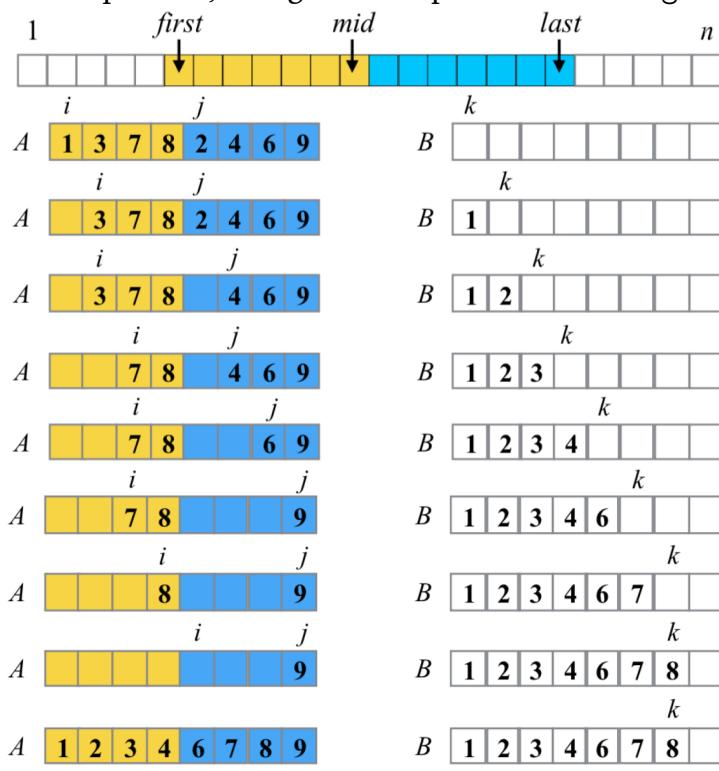
Terminata la ricorsione sulla parte sinistra del vettore iniziale, il controllo ritorna alla chiamata precedente, che ora può eseguire l'istruzione ricorsiva successiva (*riga 4*) sulla parte destra, utilizzando `mergeSort(A, mid+1, last)`.

Il procedimento è analogo al precedente: la *riga 2* suddivide il sottovettore destro in [2] e [1], mentre le istruzioni ricorsive (*righe 3 e 4*) non proseguono perché entrambi i vettori hanno un solo elemento. Infine viene richiamata `merge()` per ordinare i due elementi e ottenere [1, 2].

Merge finale (riga 4)

Terminate entrambe le ricorsioni, significa che i due sottovettori originari sono ora ordinati. L'ultima istruzione rimasta della chiamata principale è dunque la `merge()` finale, che combina i due sottovettori ordinati per ottenere un unico vettore ordinato (come illustrato in figura *e*).

Nello specifico, una generica operazione di `merge()` funziona in questo modo:



N.B: *first*, *last* e *mid* sono tali che $1 \leqslant \text{first} \leqslant \text{mid} \leqslant \text{last} \leqslant n$.

Bisogna specificare che il `merge()` agisce su dei sottovettori già ordinati: infatti il primo `merge()` viene effettuato sui singoli elementi, e da qui, ogni volta che si effettua un `merge()` si avranno sempre dei sottovettori ordinati, $A[\text{first}..\text{mid}]$ e $A[\text{id} + 1..\text{last}]$.

Per fondere le due metà ordinate, la procedura `merge()` si avvale di un vettore di appoggio B , utilizzato come parametro globale.

Vengono quindi utilizzati tre indici i , j e k per scandire, rispettivamente, $A[\text{first}..\text{mid}]$, $A[\text{mid} + 1..\text{last}]$ e $B[\text{start}..\text{end}]$. Ad ogni passo, sono confrontati gli elementi $A[i]$ e $A[j]$, il minore viene copiato in $B[k]$, e vengono incrementati di una posizione k e l'indice dell'elemento che risulta minore.

Il procedimento viene iterato fino a che una delle due metà è esaurita ($A[\text{first}..\text{mid}]$ oppure $A[\text{mid} + 1..\text{last}]$). Non appena una delle due metà si svuota per prima, è possibile proseguire l'ordinamento in modi differenti:

- Se la **prima metà è stata esaurita per prima** ($i > \text{mid}$) gli eventuali elementi $A[\text{j}..\text{end}]$ della metà non scandita si trovano già nella posizione corretta per l'ordinamento. Dunque, non è necessario spostare gli elementi non scanditi in A nel vettore di appoggio B , ma piuttosto, spostare tutti gli elementi già ordinati da B ad A ;
- Se invece **si svuota prima la seconda parte**, gli elementi non scanditi $A[\text{i}..\text{mid}]$ della prima metà vengono subito spostati nelle ultime posizioni $A[\text{k}..\text{last}]$ che competono loro nell'ordinamento finale. Infine, la posizione $B[\text{first}..\text{k} - 1]$ è ricopiata in $A[\text{first}..\text{k} - 1]$,

ottenendo così $A[first...last]$.

Il codice per il funzionamento del `merge()` è il seguente, illustrato in figura 67.

Figure 67: Pseudocodice della funzione `merge()`

Algorithm `merge(Item[]A, int first, int last, int mid)`

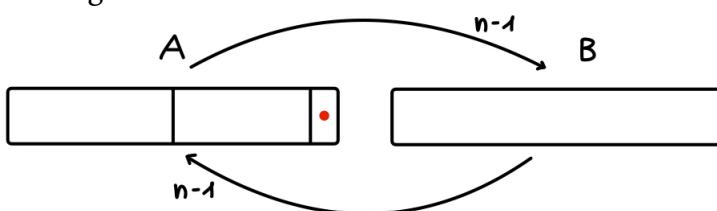
```

1: int  $i, j, k, h$ 
2:  $i = first; j = mid + 1; k = first$ 
3: while  $i \leq mid$  and  $j \leq last$  do
4:   if  $A[i] \leq A[j]$  then
5:      $B[k] = A[i]$ ; {N.B.:  $B$  è un vettore di appoggio usato come parametro globale}
6:      $i = i + 1$ 
7:   else
8:      $B[k] = A[j]$ ;
9:      $j = j + 1$ 
10:  end if
11:   $k = k + 1$ 
12: end while
13:  $j = last$ 
14: for  $h = mid$  downto  $i$  do
15:    $A[j] = A[h]$ 
16:    $j = j - 1$ 
17: end for
18: for  $j = first$  to  $k - 1$  do
19:    $A[j] = B[j]$ 
20: end for

```

Per capire quale sia la complessità di `mergeSort()` dobbiamo prima trovare separatamente la complessità di `merge()`.

Nel **caso pessimo** ogni valore in A deve essere trasferito in B , e questo accade quando rimangono tutti tranne un solo elemento della sottoparte destra di A . Quindi come illustrato



nell'immagine, in A è rimasto un singolo valore che è già nella sua posizione corretta, gli altri $n - 1$ valori sono stati trasferiti in B . A questo punto andranno tutti riportati in A , e anche in

questo caso l'operazione ha costo $n - 1$. Possiamo quindi dire che la complessità di `merge()` è $O(n)$ perché il **numero di operazioni cresce in modo lineare** rispetto al **numero totale di elementi** nei due sottovettori.

Osservazione sulla complessità di `merge()`

Il fatto che la complessità di `merge()` sia $O(n)$ è anche intuitivo dal momento che la funzione non contiene cicli annidati: il ciclo principale viene eseguito al massimo $n - 1$ volte e ogni iterazione ha costo costante. Ne deriva quindi un costo complessivo lineare.

A questo punto l'analisi della complessità si sposta sul capire quale sia la complessità dell'**intero algoritmo**. Quando analizziamo il `mergeSort()` vengono effettuate le seguenti operazioni:

1. Viene preso un array di dimensione n ;
2. Viene diviso in **due metà**: una di dimensione $n/2$ e l'altra di dimensione $n/2$;
3. Si applica ricorsivamente il `mergeSort()` su entrambe le metà.

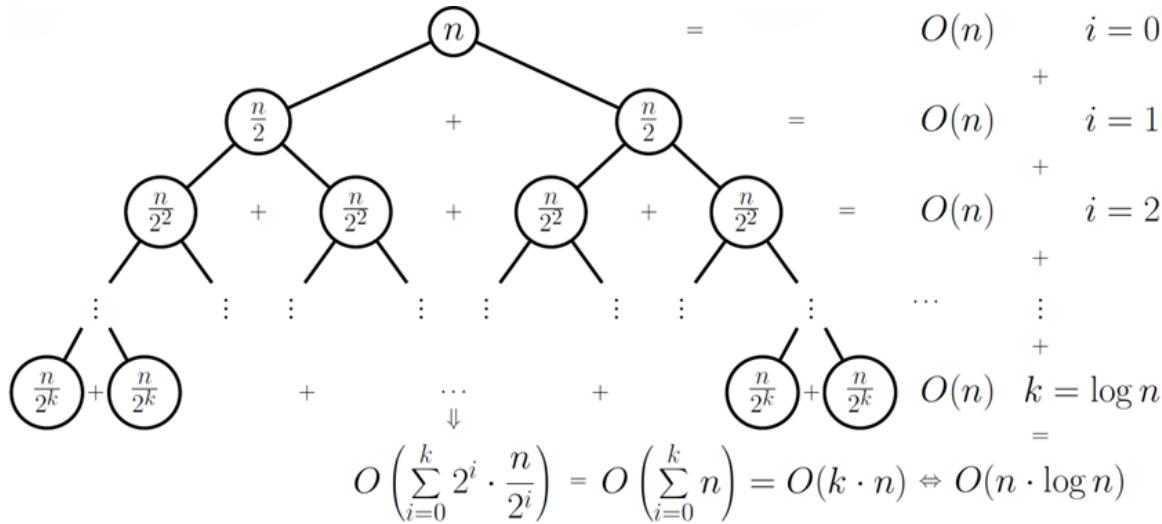
Quindi per ordinare l'array completo:

- Per ordinare la prima metà → costo $T(n/2)$;
- Per ordinare la seconda metà → costo $T(n/2)$.

Sommando i termini si ottiene: $T(n) = T(n/2) + T(n/2) + \text{costo del merge} = T(n) = 2T(n/2) + O(n)$, dove:

- $2T(n/2)$ è il tempo per ordinare le due metà;
- $O(n)$ è il tempo per fonderle con `merge()`.

Dopo aver ottenuto l'equazione del costo dell'algoritmo, per **trovare la complessità**, possiamo approfittare del fatto che `mergeSort()` sia un algoritmo ricorsivo e utilizzare il **metodo dell'albero di ricorsione** visto al capitolo 5.5.3.



Dunque, il **costo complessivo** di ciascun livello dell'albero è sempre n , perché il costo del `merge()` rimane costante per qualsiasi coppia di sottovettori.

Invece, $2T(n/2)$ è il costo per la chiamata ricorsiva che genera 2 sottovettori di dimensione $n/2$ ogni volta che viene richiamata fino ad ottenere i singoli valori.

Il **numero di sottovettori cresce esponenzialmente** (2^i), ma allo stesso momento la loro dimensione cala esponenzialmente ($n/2^i$) e il costo totale di ogni `merge()` è costante $O(n)$. Dunque, la ricorrenza ha complessità $O(n \log n)$.

Se invece si volesse utilizzare il **metodo dell'esperto** visto al capitolo 5.5.2, ricordando la formula generale $T(n) = aT(n/b) + cn^\beta$, applicata a $T(n) = 2T(n/2) + O(n)$, si avrà:

- $a = 2, b = 2$
- $\alpha = \log 2 / \log 2 = 1$
- $\beta = 1$

Quindi $\alpha = \beta$ e si ricade nel caso 2: $T(n) = \Theta(n^\alpha \cdot \log n) = \Theta(n \cdot \log n)$

6 Standard Template Library (STL)

La Standard Template Library (STL) è il cuore della libreria standard del C++, permette ai programmati l'utilizzo di algoritmi e strutture dati allo stato dell'arte, senza preoccuparsi della loro implementazione. Tutti i componenti della STL sono generici, possono quindi essere utilizzati come elementi di tipo arbitrario.

La STL porta alcuni vantaggi, tra cui:

- **Disponibilità di componenti generali:** Non offre soluzioni per un solo problema specifico, ma componenti generici che vanno bene per tutto.
- **Alto livello di astrazione:** Concentrarsi sul "cosa" vuoi fare, ignorando il "come" il computer lo gestisce.
- **Portabilità del codice:** Poiché la STL è uno standard internazionale del C++, il codice che scrivi usando queste librerie funzionerà ovunque ci sia un compilatore C++.
- **Non dover re-implementare ogni cosa from scratch:** Significa applicare il principio del riutilizzo del codice. Invece di scrivere manualmente le funzionalità di base, il programmatore utilizza i componenti già pronti della libreria.

Lo stato dell' arte

Con l'espressione "Stato dell'arte" si intende l'insieme degli algoritmi, delle tecnologie e delle metodologie che, ad oggi, offrono le prestazioni migliori in termini di efficienza, sicurezza e affidabilità.

Inoltre mette a disposizione:

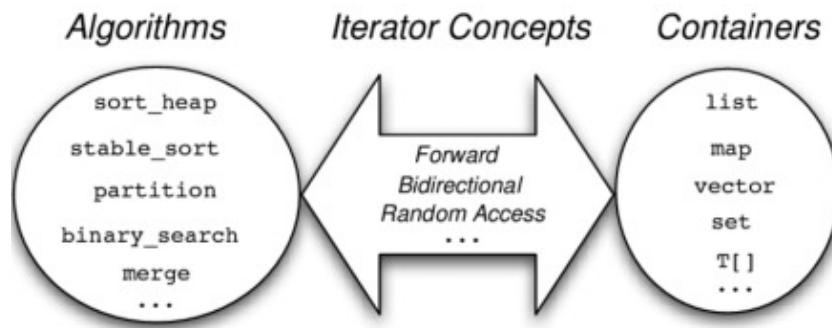
- **Varie strutture dati:** array dinamici, liste, alberi binari, ecc.
- **Vari algoritmi:** per ricerca, ordinamento, ecc.
- **Il tipo stringa.**
- **Classi per la gestione dell'I/O.**
- **Classi numeriche.**

La STL è basata sulla cooperazione di diversi componenti:

- **Contenitori (Container):** sono usati per la gestione di oggetti di un dato tipo. Possono essere implementati come array, liste, etc.
- **Iteratori (Iterator):** sono usati per visitare gli elementi di collezioni di oggetti (container o sottoinsiemi di essi).
- **Algoritmi (Algorithm):** sono usati per processare elementi di collezioni (ricerca, ordinamento, modifica, etc.).

6.1 Container

La progettazione della STL è basata sulla separazione tra dati e operazioni. I dati sono gestiti dalle **classi container**, mentre le operazioni sono definite da **algoritmi**. Gli iteratori sono il “collante” tra container e algoritmi.



STL fornisce diversi tipi di container per venire incontro a diverse necessità da parte del programmatore. Ognuno di essi ha vantaggi e svantaggi, la scelta deve essere effettuata a seconda delle operazioni che si vogliono effettuare. Esistono due tipi di container:

- **Sequenziali (Sequence container)**: vector, deque e list.
- **Associativi (Associative container)**: set, multiset, map e multimap.

6.2 Sequence container (Contenitori Sequenziali)

I Sequence container, contengono una collezione ordinata di elementi di un unico tipo. I principali sono:

- **vector**: archivia gli elementi in una disposizione lineare e consente l'accesso casuale a qualsiasi elemento.
- **list**: viene archiviata come elenco collegato bidirezionale di elementi in disposizione lineare. Consente inserimenti ed eliminazioni efficienti in qualsiasi posizione all'interno della sequenza.
- **deque**: si comporta come un vector ma è preferibile nelle implementazioni delle code.

6.2.1 Vector

Il vector è considerato il container di "default" del C++. Gli elementi al suo interno sono gestiti come in un **array dinamico**, ovvero un array la cui dimensione può cambiare durante l'esecuzione del programma. Le sue caratteristiche principali sono:

- + **Accesso casuale (Random Access)**: È possibile accedere direttamente a qualsiasi elemento tramite il suo indice, senza dover scorrere gli elementi precedenti.
- + **Inserimento/Rimozione in coda**: L'aggiunta o la rimozione di elementi alla fine del vettore è un'operazione molto veloce.
- + **Inserimento nel mezzo**: L'inserimento o la rimozione in posizioni diverse dalla coda è meno efficiente, in quanto richiede lo spostamento (shift) di tutti gli elementi successivi per fare spazio a quello nuovo o per riempire il vuoto.

- **Lentezza posizionale:** Potenzialmente lento se le operazioni di inserimento/cancellazione avvengono spesso in testa (inizio) o in posizioni casuali.
- **Costo di copia:** Se il tipo di dato contenuto è complesso, riordinare gli elementi (spostarli) può essere costoso.
- **Invalidazione di puntatori e iteratori:** Ogni operazione che modifica la capacità del vettore (es. un inserimento che fa scattare un ridimensionamento automatico) può far sì che il vettore venga spostato in un nuovo blocco di memoria. Di conseguenza, tutti i puntatori o iteratori che puntavano ai vecchi elementi diventano non validi (dangling pointers) e usarli causa errori nel programma.

Metodi principali:

Modificatori (Modifiers):

- `assign()`: Assegna nuovi valori al vettore, sostituendo completamente quelli vecchi.
- `push_back()`: Aggiunge un nuovo elemento alla fine del vettore.
- `pop_back()`: Rimuove l'ultimo elemento del vettore (riducendo la dimensione di 1).
- `insert()`: Inserisce nuovi elementi prima di una posizione specifica (indicata tramite un iteratore).
- `erase()`: Rimuove elementi da una specifica posizione o da un intervallo.
- `swap()`: Scambia il contenuto di due vettori dello stesso tipo. È un'operazione molto efficiente anche se le dimensioni sono diverse.
- `clear()`: Rimuove **tutti** gli elementi dal vettore.

Iteratori (Iterators):

- `begin()`: Restituisce un iteratore che punta al **primo** elemento del vettore.
- `end()`: Restituisce un iteratore che punta all'elemento teorico **successivo all'ultimo** (past-the-end). Non punta all'ultimo dato valido, ma segna la fine del container.

Gestione della Capacità (Capacity):

- `size()`: Restituisce il numero effettivo di elementi attualmente presenti nel vettore.
- `max_size()`: Restituisce il numero massimo teorico di elementi che il vettore può contenere.
- `capacity()`: Restituisce la dimensione dello spazio di memoria attualmente **allocato**. Indica il numero elementi il vettore può contenere prima di dover chiedere altra memoria al sistema.
- `resize(n)`: Ridimensiona il contenitore affinché contenga esattamente 'n' elementi. Se 'n' è minore dell'attuale size, gli elementi in eccesso vengono distrutti.
- `empty()`: Restituisce true se il container è vuoto (ovvero se la size è 0).
- `shrink_to_fit()`: Riduce la capacità (memoria allocata) per farla coincidere con la size (elementi effettivi), liberando la memoria inutilizzata.

Listing 1: Esempio di utilizzo dei metodi di Capacità

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
    // Dichiarazione di un vettore di interi vuoto.
    vector<int> g1;

    // Ciclo di riempimento: aggiunge gli interi da 1 a 5.
    for (int i = 1; i <= 5; i++)
        // push_back(): Metodo per l'inserimento efficiente in coda.
        g1.push_back(i);

    // Accesso casuale O(1): usa l'operatore [] per accedere al terzo
    // elemento (valore 3).
    cout << "\nReference operator [g] : g1[2] = " << g1[2];

    cout << "Output of begin and end: ";
    // Iterazione: usa gli iteratori begin() e end() per stampare il
    // contenuto (1 2 3 4 5).
    for (auto i = g1.begin(); i != g1.end(); ++i)
        cout << *i << " ";

    // size(): Numero attuale di elementi (5).
    cout << "\nSize : " << g1.size();

    // capacity(): Memoria allocata, sara >= size().
    cout << "\nCapacity : " << g1.capacity();

    // max_size(): Dimensione massima teorica.
    cout << "\nMax_Size : " << g1.max_size();

    // resize(4): Ridimensiona il vettore a 4 elementi. L'ultimo (5) viene
    // eliminato.
    g1.resize(4);

    cout << "\nSize : " << g1.size(); // Nuova dimensione (4)

    // empty(): Controllo se il vettore e' vuoto (restituisce false).
    if (g1.empty() == false)
        cout << "\nVector is not empty";
    else
        cout << "\nVector is empty";

    return 0;
}
```

6.2.2 List

Una lista è implementata come una lista di elementi doppiamente collegata, quindi ogni elemento ha il suo segmento di memoria e puntatori sia al predecessore che al successore. La lista non ha random access per cui per accedere ad un ipotetico decimo elemento, bisogna scorrere i primi nove elementi della catena.

Confronto con Vector

- + **Inserimento e cancellazione in $O(1)$:** Una volta trovata la posizione, l'operazione è rapidissima, in quanto si modificano solo i puntatori (link), senza spostare i dati.
- + **Ideale per oggetti pesanti:** Non essendoci costi di copia o spostamento degli elementi in memoria, è la scelta migliore per memorizzare oggetti di grandi dimensioni.
- **Accesso casuale in $O(N)$:** Per accedere all'elemento N , è necessario scorrere N nodi a partire dalla testa o dalla coda. Non supporta l'operatore `[]`.
- **Traversamento lento (Bad Memory Locality):** I nodi sono sparsi in memoria. La CPU non riesce a sfruttare la cache, rendendo lo scorrimento sequenziale meno efficiente rispetto al vector.
- **Maggiore consumo di memoria (Overhead):** Ogni nodo richiede memoria aggiuntiva per memorizzare i due puntatori (prev e next).

Metodi principali:

- `front()`: Ritorna il valore del primo elemento della lista.
- `back()`: Ritorna il valore dell' ultimo elemento della lista.
- `push_front()`: Aggiunge un nuovo elemento all' inizio della lista.
- `push_back()`: Aggiunge un nuovo elemento alla fine della lista.
- `pop_front()`: Rimuove il primo elemento della lista e riduce la grandezza di 1.
- `pop_back()`: Rimuove l'ultimo elemento della lista e riduce la grandezza di 1.
- `insert()`: Aggiunge un nuovo elemento prima di quello nella posizione specificata.
- `size()`: Ritorna il numero di elementi nella lista.
- `begin()`: Ritorna un iteratore che punta al primo elemento della lista.
- `end()`: Ritorna un iteratore che punta al teorico ultimo elemento della lista.

Listing 2: Esempio di utilizzo dei metodi delle list

```
#include <iostream>
#include <list>

using namespace std;

int main() {
    list<int> gqlist{12, 45, 8, 6};

    // push_front(33): Inserisce l'elemento 33 all'inizio della lista (testa)
    .
    gqlist.push_front(33);
```

```

// push_back(55) : Inserisce l'elemento 55 alla fine della lista (coda).
gqlist.push_back(55);

// Output finale: 33 12 45 8 6 55
for (auto i : gqlist) {
    cout << i << " ";
}
return 0;
}

```

6.2.3 Deque

Il termine deque è un'abbreviazione per “double-ended queue”. È un array dinamico implementato in maniera tale da rendere veloci inserimento e cancellazione sia in testa che in coda.

Confronto con Vector e List

- + **Accesso casuale in $O(1)$:** Accesso diretto ad ogni elemento tramite indice (come il vector), sebbene con un minimo overhead.
- + **Inserimento e cancellazione su entrambe le estremità ($O(1)$):** A differenza del vector, supporta `push_front()` e `pop_front()` in tempo costante.
- + **Non invalida puntatori in coda/testa:** Le operazioni di inserimento sui bordi non invalidano i riferimenti agli elementi esistenti.
- **Lentezza negli inserimenti/rimozioni centrali ($O(N)$):** Come il vector, le operazioni che richiedono lo spostamento dei dati sono costose.
- **Costo di copia:** Potenziale lentezza se il tipo di dato ha un elevato costo di copia durante il riordino degli elementi.
- **Non completamente cache-friendly:** Non essendo totalmente contiguo in memoria, è meno efficiente del vector nel sfruttare la cache della CPU.

Listing 3: Esempio di utilizzo dei metodi della deque

```

#include <deque>
#include <iostream>
using namespace std;

// Funzione helper per stampare il contenuto del deque.
void showdq(deque<int> g)
{
    deque<int>::iterator it;
    // Scorrimento sequenziale tramite iteratore begin() ed end().
    for (it = g.begin(); it != g.end(); ++it)
        cout << '\t' << *it;
    cout << '\n';
}

```

```

int main()
{
deque<int> gquiz; // Dichiarazione di un deque vuoto.

// Operazioni di Inserimento in testa e coda (O(1)).
gquiz.push_back(10); // Deque: {..., 10}
gquiz.push_front(20); // Deque: {20, 10, ...}
gquiz.push_back(30); // Deque: {20, 10, 30}
gquiz.push_front(15); // Deque: {15, 20, 10, 30}
// cout << "The deque gquiz is : ";
showdq(gquiz);

// size(): Stampa la dimensione attuale (4)
cout << "\ngquiz.size() : " << gquiz.size();
// max_size(): Stampa la dimensione massima teorica.
cout << "\ngquiz.max_size() : " << gquiz.max_size();

// at(2): Accesso casuale all'elemento in indice 2 (valore 10).
cout << "\ngquiz.at(2) : " << gquiz.at(2);
// front(): Elemento in testa (15).
cout << "\ngquiz.front() : " << gquiz.front();
// back(): Elemento in coda (30).
cout << "\ngquiz.back() : " << gquiz.back();

return 0;
}

```

6.2.4 Altri container

Dati i container fondamentali, STL offre anche supporto specifico per le seguenti strutture:
Metodi principali:

- Stack: container in cui gli elementi sono gestiti con politica LIFO.
- Queue: container in cui gli elementi sono gestiti con politica FIFO.
- Priority Queue: la coda con priorità.
- Associative container (Contenitori Associativi): supportano efficientemente interrogazioni circa la presenza e richiese di estrazione di un elemento. I principali sono: set, multiset, map e multimap.

6.2.5 Associative container

Sono collezioni raggruppate nelle quali la posizione corrente di un elemento dipende dal suo valore e da un dato criterio di ordinamento. L'ordine di inserimento non è tenuto in conto. Il criterio di ordinamento ha la forma di una funzione che compara un valore od una

determinata chiave. Sono tipicamente implementati come alberi binari.

Nella STL, sono predefiniti i seguenti associative container:

- Set: collezione nella quale gli elementi sono ordinati per valore. Ogni elemento può occorrere una sola volta, non sono ammessi duplicati.
- Multiset: come il Set, ma con la differenza che i duplicati sono ammessi.
- Map: contiene elementi che sono coppie chiave/valore. Ogni elemento ha quindi una chiave (sulla base della quale è effettuato l'ordinamento) ed un valore. Non sono ammessi duplicati nelle chiavi.
- Multimap: come le map, ma sono ammesse chiavi duplicate.

6.3 Iteratori

Un iteratore è un oggetto che può iterare tra gli elementi di un container (o parte di esso). Fornisce un metodo generale per accedere in successione a ciascun elemento di un container. Un iteratore rappresenta una certa posizione all'interno del container, ad esempio, se iter è un iteratore di un container, allora con `++iter` fa avanzare l'iteratore avanti in modo che punti al successivo elemento del container, mentre con `*iter` restituisce il valore dell'elemento puntato.

Operazioni Fondamentali:

- `*`: **Dereferenziazione**. Restituisce l'elemento (*il valore*) puntato dalla posizione corrente dell'iteratore.
- `->`: **Accesso a Membri**. Permette di accedere ai membri di una struttura o classe puntata dall'iteratore, senza dover dereferenziare esplicitamente.
- `++ / -`: **Avanzamento/Arretramento**. Muove l'iteratore all'elemento successivo (`++`) o precedente (`-`). (*Attenzione: non tutti gli iteratori supportano -*).
- `== / !=`: **Confronto**. Controlla se due iteratori puntano alla **stessa posizione** all'interno del container.
- `=`: **Assegnazione**. Assegna la posizione di un iteratore a un altro.

Ogni classe container nella Standard Template Library (STL) del C++ fornisce le stesse funzioni membro di base per l'utilizzo degli iteratori, definendo così un'interfaccia uniforme per l'accesso ai dati. Queste funzioni sono essenziali per permettere a tutti gli algoritmi STL di lavorare con qualsiasi tipo di container, indipendentemente dalla sua implementazione interna

cosa sono e funzioni membro di base?

Le funzioni membro di base sono i metodi standard che ogni classe container deve implementare per garantire la compatibilità con gli algoritmi della Standard Template Library (STL).

Queste funzioni definiscono l'interfaccia minima per l'utilizzo degli iteratori, permettendo così agli algoritmi di lavorare con qualsiasi container senza conoscerne i dettagli implementativi interni.

Le due funzioni membro fondamentali sono `begin()` ed `end()`, che definiscono il range di elementi su cui un algoritmo può operare:

- **begin()**: Restituisce un iteratore che punta all'inizio degli elementi validi del container,

ovvero alla posizione del primo elemento (se esiste).

- **end()**: Restituisce l'iteratore alla fine logica degli elementi, noto come past-the-end iterator. Questo iteratore punta alla posizione teorica immediatamente successiva all'ultimo elemento; non rappresenta un elemento valido e non deve mai essere dereferenziato.

Questo modello standardizzato offre due vantaggi cruciali per la programmazione generica in C++:

- **Semplice Criterio di Terminazione**: Il modello [begin(), end()] offre un criterio di terminazione universale e semplice per tutti i cicli di visita. Si può continuare a scorrere gli elementi finché l'iteratore corrente non raggiunge (non è uguale a) end().
- **Gestione dei Contenitori Vuoti**: I contenitori vuoti non richiedono trattamenti particolari o controlli esplicativi. Quando un container è vuoto, begin() è direttamente uguale a end(). Di conseguenza, il ciclo di visita non viene eseguito nemmeno una volta, garantendo che il codice non tenti mai di accedere a dati inesistenti.

Listing 4: Inserzione di elementi in un set e stampa usando gli iteratori

```
#include <iostream>
#include <set>

using namespace std;

int main() {
    // Definizione: crea un container 'set' che memorizza valori di tipo int.
    // Un set mantiene gli elementi unici e ordinati automaticamente.
    set<int> coll;

    // coll.insert(): Metodo usato per aggiungere nuovi elementi.
    coll.insert(3);
    coll.insert(1);
    coll.insert(5);
    coll.insert(4);

    // Tentativo di inserire il valore 1 una seconda volta.
    // In un set, l'inserimento non avviene perche' gli elementi devono
    // essere unici.
    coll.insert(1);
    coll.insert(6);
    coll.insert(2);

    // Stampa di tutti gli elementi usando gli iteratori.
    // L'iterazione avviene in ordine crescente (1, 2, 3, 4, 5, 6) grazie
    // alla natura del set.
    for (auto pos = coll.begin(); pos != coll.end(); ++pos) {
        cout << *pos << " ";
    }

    cout << endl;
    return 0;
}
```

Il set è composto da interi: gli elementi saranno in ordine ascendente (usa <). È possibile anche ordinarli in maniera diversa, ad esempio in ordine discendente, con una struttura del tipo: *set < int, greater < int >> coll*

NB: lo spazio è volutamente lasciato altrimenti » viene interpretato come operatore shift!

Il nuovo elemento è inserito col metodo insert automaticamente, push back o push front non sono ammessi.

Listing 5: Esempio di utilizzo di std::multimap con chiavi duplicate

```
#include <iostream>
#include <map> // Necessario per multimap
#include <string>
using namespace std;

// Alias per semplificare il tipo: multimap<chiave: int, valore: string>
typedef multimap<int, string> IntStringMap;

int main() {
    IntStringMap coll; // Dichiarazione del container multimap.

    // Inserimento di elementi: si usano coppie (chiave, valore).
    coll.insert(make_pair(5, "tagged"));
    coll.insert(make_pair(1, "this"));
    coll.insert(make_pair(4, "of"));
    coll.insert(make_pair(6, "strings"));

    // Inserimento con chiave duplicata (4): questo e' permesso in multimap.
    coll.insert(make_pair(4, "of"));

    coll.insert(make_pair(3, "multimap"));

    // Iterazione e stampa dei soli valori.
    // Il multimap garantisce l'ordinamento automatico in base alla chiave.
    for(IntStringMap::iterator pos = coll.begin(); pos != coll.end(); ++pos)
    {
        // pos->second: L'iteratore accede alla coppia; '.second' e' il
        // valore (stringa).
        // L'output finale riflettera' l'ordine delle chiavi: 1, 3, 4, 4, 5,
        // 6.
        cout << pos->second << " ";
    }

    cout << endl;
    return 0;
}
```

Listing 6: Map come array associativo

```
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main() {

    // Definisce un alias: chiave=stringa, valore=float.
    typedef map<string, float> StringFloatMap;

    StringFloatMap coll;

    // Inserimento tramite l'operatore di array associativo [].
    // Se la chiave non esiste, la crea; altrimenti, ne aggiorna il valore.
    coll["VAT"] = 0.15;
    coll["Pi"] = 3.1415;
    coll["an arbitrary number"] = 4983.223;
    coll["Null"] = 0;

    // Iterazione su tutti gli elementi (in ordine alfabetico di chiave).
    for (StringFloatMap::iterator pos = coll.begin(); pos != coll.end(); ++pos) {
        // pos->first e' la chiave (stringa); pos->second e' il valore (float).
        cout << "key: " << pos->first << " \\" "
            << "value: " << pos->second << endl;
    }
    return 0;
}
```

Gli iteratori possono avere altre funzionalità, oltre quelle di base. Le funzionalità addizionali dipendono dal tipo di container. Gli iteratori si dividono in due categorie:

- **Iteratori bidirezionali:** Possono agire in due direzioni, in avanti, con l'operatore di incremento o, indietro con l'operatore di decremento. In questa tipologia ricadono gli iteratori per le classi dei container list, set, multiset, map, multimap.
- **Iteratori ad accesso casuale:** Hanno gli operatori necessari a poter definire una “aritmetica degli iteratori” (analogia a quella dei puntatori). In questa tipologia ricadono gli iteratori per le classi dei container vector e deque.

NB: se si vuole scrivere codice generico (indipendente dal container) meglio usare solo gli operatori degli iteratori bidirezionali

6.4 Algoritmi della STL

La STL fornisce una serie di algoritmi per processare elementi in collezioni che offrono servizi di base quali ricerca, ordinamento, copia, ecc...

Gli algoritmi non sono funzioni membro delle classi container, ma funzioni globali che operano tramite iteratori. Il vantaggio è che sono implementati una volta per tutte indipendentemente dal container utilizzato.

Listing 7: Esempio di utilizzo degli algoritmi STL

```
#include <iostream>
#include <vector>
#include <algorithm> // Necessario per tutti gli algoritmi STL
using namespace std;

int main() {
    vector<int> coll;
    // Inserimento elementi in ordine arbitrario nel vettore.
    coll.push_back(2);
    coll.push_back(5);
    coll.push_back(1);
    coll.push_back(6);
    coll.push_back(4);
    coll.push_back(3);

    // 1. Ricerca del Minimo e Massimo:
    // min_element e max_element restituiscono un iteratore all'elemento trovato.
    auto pos_min = min_element(coll.begin(), coll.end());
    cout << "min: " << *pos_min << endl; // Stampa il valore minimo (1)

    auto pos_max = max_element(coll.begin(), coll.end());
    cout << "max: " << *pos_max << endl; // Stampa il valore massimo (6)

    // 2. Ordinamento (Sort):
}
```

```

// L'algoritmo sort riordina gli elementi nell'intervallo [begin(), end())
).
sort(coll.begin(), coll.end()); // Risultato in coll: 1 2 3 4 5 6

// 3. Ricerca specifica (Find):
// Trova la prima occorrenza del valore 3 e restituisce l'iteratore alla
// posizione.
auto pos = find(coll.begin(), coll.end(), 3);

// 4. Inversione (Reverse):
// Inverte l'ordine degli elementi dall'elemento trovato (pos) fino alla
// fine.
reverse(pos, coll.end()); // Risultato in coll: 1 2 3 6 5 4

// Stampa tutti gli elementi del vettore dopo le modifiche.
for (pos = coll.begin(); pos != coll.end(); ++pos)
    cout << *pos << " ";

cout << endl;
return 0;
}

```

6.4.1 Intervalli

Tutti gli algoritmi lavorano con uno o più intervalli (range) di elementi. Bisogna passare come argomento l'inizio e la fine di un intervallo, anziché l'intera collezione come un unico elemento. È compito dell'utente assicurarsi che l'intervallo sia valido. Ogni algoritmo processa intervalli chiusi a sinistra (come quelli usati dagli iteratori).

Listing 8: Algoritmo for_each()

```

#include <iostream>
#include <vector>
#include <algorithm> // Necessario per for_each
using namespace std;

// Funzione predicato: definita per stampare ogni elemento ricevuto in
// input.
void print (int elem) {
    cout << elem << ' ';
}

int main() {
    vector<int> coll;
    // Inserimento elementi da 1 a 9 nel vettore.
    for (int i=1; i<=9; ++i) {
        coll.push_back(i);
    }
}

```

```

// for_each: Applica la funzione 'print' a tutti gli elementi nel range [begin, end].
// range: coll.begin(), coll.end()
// operation: print
for_each (coll.begin(), coll.end(), print);

cout << endl;
return 0;
}

```

6.4.2 Predicati

Un predicato è un tipo speciale di funzione ausiliaria per gli algoritmi, restituiscono un valore booleano. Sono spesso usati come criteri di discriminazione nelle procedure di ordinamento e di ricerca e possono essere di due tipi: **unari** o **binari**. Inoltre devono restituire sempre lo stesso risultato dato lo stesso input.

- **Predicati unari:** Controllano specifiche proprietà di un singolo argomento. Esempio: funzione isPrime.
- **Predicati binari:** Usati per confrontare specifiche proprietà di due argomenti.

Necessario quando, ad esempio, voglio ordinare una collezione di elementi per cui non è definito l'operatore <.

Listing 9: Algoritmo find_if(): uso di un predicato per la ricerca

```

#include <iostream>
#include <list>
#include <algorithm> // Necessario per find_if
#include <cstdlib> // Necessario per la funzione abs()
using namespace std;

// PREDICATO: Funzione che definisce il criterio di ricerca.
// Deve restituire un valore booleano (true o false).
bool isPrime (int number) {
    int num = abs(number); // Considera solo il segno positivo
    if (num <= 1) return false; // 0 e 1 non sono primi

    // Controlla l'esistenza di un divisore > 1.
    for (int divisor = num/2; divisor > 1; --divisor) {
        if (num % divisor == 0) return false;
    }
    // Se il ciclo finisce senza trovare divisori, il numero e' primo.
    return true;
}

int main() {
    list<int> coll;
    // Inserisce elementi da 24 a 30.
}

```

```

for (int i=24; i<=30; ++i)
    coll.push_back(i);

// find_if: Cerca il primo elemento nel range [begin, end) per cui la
   funzione predicato (isPrime) restituisce true.
list<int>::iterator pos =
    find_if(coll.begin(), coll.end(), isPrime);

// Il valore trovato e' 29.
if (pos != coll.end()) {
    cout << *pos << " is first prime number found" << endl;
} else {
    cout << "no prime number found" << endl;
}
return 0;
}

```

7 Divide-et-impera

7.1 Risoluzione problemi

Dato un problema, non si hanno metodi generali per risolverlo in modo efficiente, tuttavia è possibile evidenziare 4 fasi:

1. Classificazione del problema;
2. Caratterizzazione della soluzione;
3. Tecnica di progetto;
4. Utilizzo di strutture dati.

Queste fasi non sono necessariamente sequenziali.

7.1.1 Classificazione dei problemi

Problemi decisionali Il dato in ingresso soddisfa una certa proprietà? **Soluzione:** risposta sì/no. Un problema di esempio può essere stabilire se un grafo è connesso.

Problemi di ricerca

Problemi in cui si ha:

- **Spazio di ricerca:** insieme di "soluzioni" possibili;
- **Soluzione ammissibile:** soluzione che rispetta certi vincoli;

Ad esempio la posizione di una sotto-stringa in una stringa.

Problemi di ottimizzazione

Problemi in cui ogni soluzione è associata ad una funzione di costo e si vuole la soluzione di costo minimo. ad esempio il cammino più breve tra due nodi (grafo pesati).

7.1.2 Definizione matematica del problema

Una cosa fondamentale da fare è definire bene il problema in modo formale. Spesso la formulazione è banale, ma può suggerire una prima idea di soluzione. Ad esempio, data una sequenza di n elementi, una permutazione ordinata è data dal minimo seguito da una permutazione ordinata degli $n - 1$ elementi (Selection Sort). La definizione matematica del problema può suggerire una possibile tecnica di soluzione.

Sottostruttura ottima → Programmazione dinamica;

Proprietà greedy → Tecnica greedy;

7.1.3 Tecnica di soluzione dei problemi

Divide-et-impera

Un problema viene suddiviso in sotto-problemi indipendenti, che vengono risolti ricorsivamente (top-down).

Ambito: problemi di decisione, ricerca.

Top-down

La strategia top-down è una strategia in cui un problema viene diviso in parti sempre più piccole fino a che non si ottengono componenti semplici da implementare.

Top-down è diverso da divide-et-impera, dato che quest'ultimo fa uso di top-down, che è una strategia di progettazione, mentre divide-et-impera è una tecnica algoritmica specifica con una struttura ben definita.

Programmazione dinamica

La soluzione viene costituita (bottom-up) a partire da un insieme di sotto-problemi potenzialmente ripetuti.

Ambito: problemi di ottimizzazione.

Bottom-up

Bottom-up, al contrario di top-down, parte da componenti più piccole e riutilizzabili e le combina per ottenere funzionalità più complesse fino ad ottenere un intero sistema.

Come funziona:

1. Si implementano piccole parti indipendenti;
2. Le si mettono insieme per formare parti più grandi;
3. Si continuano a comporre finché non si ottiene l'algoritmo/programma completo.

Memoization (o annotazione)

Versione top-down della programmazione dinamica.

Memoization (o annotazione)

Versione top-down della programmazione dinamica.

Tecnica greedy

Approccio "ingordo": si fa sempre la scelta localmente ottima.

Backtrack

Si procede per "tentativi" e si torna di tanto in tanto sui propri passi.

Ricerca locale

La soluzione ottima viene trovata "migliorando" via via soluzioni esistenti.

Algoritmi probabilistici

Meglio scegliere con giudizio (in maniera costosa) o scegliere a caso "gratuitamente".

7.2 Divide-et-impera

Questa tecnica, applicata alla risoluzione di un problema computazionale, consiste nel partizionare il problema in sotto-problemi più piccoli dello stesso tipo e indipendenti, risolverli ricorsivamente, e successivamente ricombinare, con poco sforzo, le soluzioni parziali per ottenere la soluzione del problema originale.

Lo "schema" di una procedura ricorsiva `divideEtImpera()` per risolvere un problema di P di dimensione n è il seguente, dove k è una costante intera prefissata:

```
divideEtImpera( $P$ , integer  $n$ )
  if  $n \leq k$  then
    risolvi  $P$  direttamente
  else
    dividi  $P$  in  $h$  sottoproblemi  $P_1, \dots, P_h$  di dimensione  $n_1, \dots, n_h$ 
    for  $i \leftarrow 1$  to  $h$  do divideEtImpera( $P_i, n_i$ )
    combina i risultati di  $P_1, \dots, P_h$  per ottenere quello di  $P$ 
  end if
```

La tecnica *divide-et-impera* permette di provare agevolmente la correttezza dell'algoritmo usando il principio di induzione e di impostarne facilmente le relazioni ricorrenti della funzione $T(n)$ di complessità:

$$T(n) = \begin{cases} c, & n \leq k \\ D(n) + C(n) + \sum_{i=1}^h T(n_i), & n > k \end{cases}$$

dove c è una costante, $D(n)$ è il numero di operazioni per dividere il problema e $C(n)$ è il numero di operazioni per combinare i risultati.

Nota

Se i dati sono partizionati in maniera bilanciata, cioè tutti gli n_i sono all'incirca uguali, allora l'algoritmo può risultare molto efficiente.

7.3 Minimo: Divide-et-impera

Si applichi ora il metodo divide-et-impera a un problema di ricerca del valore minimo in un array A :

```
int minrec(int[] A, int i, int j)
  if  $i == j$  then
    return A[i]
  else
    m =  $\lfloor (i + j)/2 \rfloor$ 
    return min(minrec(A, i, m), minrec(A, m + 1, j))
  end if
```

parte intera inferiore e parte intera superiore

$\lfloor x \rfloor$ rappresenta la parte intera inferiore, spesso chiamata "floor", ovvero se si ha un numero reale esso, se seguito da virgola, viene approssimato per difetto, ad esempio: $\lfloor 3,9 \rfloor = 3$.

Mentre, al contrario $\lceil x \rceil$ rappresenta la parte intera superiore, detta "ceiling", e si approssima per eccesso, ad esempio: $\lceil 3,1 \rceil = 4$.

Nell'algoritmo avviene che viene preso un valore medio della posizione dell'array A , con i che indica l'inizio di A e j che ne indica la fine, m è quindi l'indice che indica la metà dell'array. Ogni volta viene richiamata la funzione fino a quando non si arriva ad avere un singolo elemento nell'array, ovvero la condizione dell'"if".

L'array viene quindi diviso in tanti piccoli pezzi per trovare il minimo avendo una complessità:

$$T(n) = \begin{cases} 2T(n/2) + 1, & n > 1 \\ 1, & n = 1 \end{cases}$$

Analizzando la complessità dell'algoritmo, quindi, si ha un caso base, quando l'array contiene un solo elemento e quindi il costo risulta costante, perché basta restituire quell'elemento, ma nel caso ricorsivo con n elementi il costo è dato da $2T(n/2)$, ovvero due chiamate ricorsive, ciascuna su metà dell'input ($n/2$) e $+1$ che rappresenta il costo costante delle operazioni di divisione (ovvero il calcolo di m) e di combinazione (confronto finale tra i due minimi).

Risulta che l'algoritmo divide-et-impera non è conveniente.

7.4 Esempio binary search

```
int binarySearch(int[] S, int v, int i, int j)
  if i > j then
    return 0
  else
    m = ⌊(i + j)/2⌋
    if S[m] == v then
      return m
    else if S[m] < v then
      return binarySearch(S, v, m + 1, j)
    else
      return binarySearch(S, v, i, m - 1)
    end if
  end if
```

La procedura binary search è un particolare tipo di divide-et-impera. Il problema viene suddiviso in 2 sottoproblemi ($h=2$) approssimativamente uguali, mentre i costi $C(n)$ e $D(n)$ sono costanti. La particolarità sta nel fatto che la chiamata ricorsiva avviene in solo uno dei due sottoproblemi, invece che su entrambi.

Si può considerare un esempio "semplificato" di divide-et-impera, ma permette di enunciare una regola importante in questi casi: **se non tutti i sottoproblemi devono essere analizzati, è buona norma affrontare ricorsivamente i problemi più piccoli possibili.**

La complessità della ricerca può essere ulteriormente abbassata tenendo conto dei valori delle chiavi memorizzate nel vettore e della loro **distribuzione di probabilità**.

Esempio di ricerca: Dizionario

Se si vuole cercare la parola "casa" nel dizionario, non lo si apre a metà ma a circa $\frac{1}{4}$ del numero di pagine.

Si consideri una rappresentazione a vettore, con n chiavi numeriche e distribuite uniformemente nell'intervallo $[k_{min}, k_{max}]$. Dovendo cercare le chiavi k in $S[1\dots n]$, si tenta la ricerca in una posizione "ragionevolmente" vicina (non al centro), data da: $n * \frac{k - k_{min}}{k_{max} - k_{min}}$.

Il numeratore fornisce lo scarto tra i valori della chiave da cercare e quella più piccola, mentre il denominatore fornisce lo scarto tra i valori della chiave più grande e quella più piccola. Il loro rapporto indica quanto k si discosta dagli estremi.

Se $k = k_{min}$, allora il rapporto è 0 e conviene quindi cercare nella prima posizione del vettore, se $k = k_{max}$, il rapporto è 1 e allora conviene cercare nell'ultima posizione del vettore. La ricerca binaria "ignora" k , assume che il rapporto sia pari a $\frac{1}{2}$ e tenta in posizione centrale. Il metodo di ricerca risultante è detto di **interpolazione**.

La procedura di ricerca binaria `binarySearch()` può essere trasformata in una di interpolazione semplicemente sostituendo l'istruzione:

- $m \leftarrow \lfloor (i + j)/2 \rfloor$
con
- $m \leftarrow i + \lfloor (k - A[i]) * (j - i) / (A[j] - A[i]) \rfloor$

Infatti la ricerca viene effettuata in generale sulla porzione $A[i\dots j]$ del vettore, che contiene la più piccola chiave in $A[i]$ e la più grande in $A[j]$. La formula per m si ricava da quella ricerca

binaria, che era $i + \lfloor (j - i)/2 \rfloor$, sostituendo $(k - A[i]) * (j - i)/(A[j] - A[i])$ ad $\frac{1}{2}$.

Complessità

Con distribuzione uniforme delle chiavi, è possibile dimostrare che la complessità è $O(\log \log n)$. La funzione $\log \log n$ cresce molto lentamente, ma è paragonabile a $\log n$ per n piccolo. Pertanto, se ci sono poche chiavi, oppure se le chiavi non sono uniformemente distribuite, è più conveniente usare la ricerca binaria. Al contrario, conviene usare l'interpolazione se ci sono tante chiavi oppure se sono uniformemente distribuite.

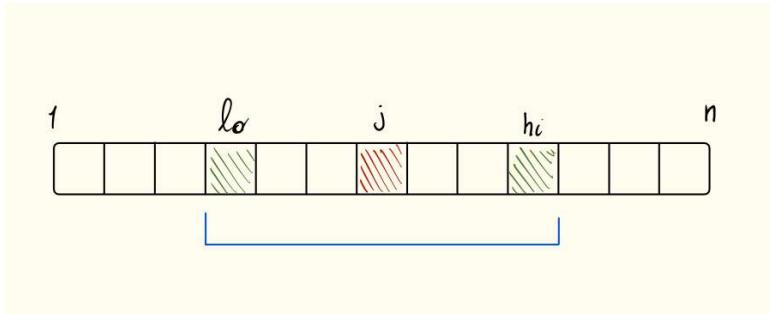
7.5 Quicksort (Hoare, 1961)

L'algoritmo di QuickSort è l'algoritmo praticamente più efficiente per ordinare gli elementi di un vettore.

Questo è basato sulla tecnica *divide-et-impera*, ma differisce dal "MergeSort" nel modo in cui divide il problema e combina i risultati.

Questo algoritmo ha un caso medio: $O(n \log n)$ e un caso pessimo: $O(n^2)$ che però viene evitato grazie a tecniche "euristiche" (ovvero, in generale, tecniche che non garantiscono di trovare la soluzione ottimale o perfetta, ma progettate per trovare una soluzione soddisfacente in un tempo ragionevole) e spesso è preferito ad altri algoritmi a causa di costanti moltiplicative più basse. Nella scelta, spesso si valuta anche la "stabilità" dell'ordinamento.

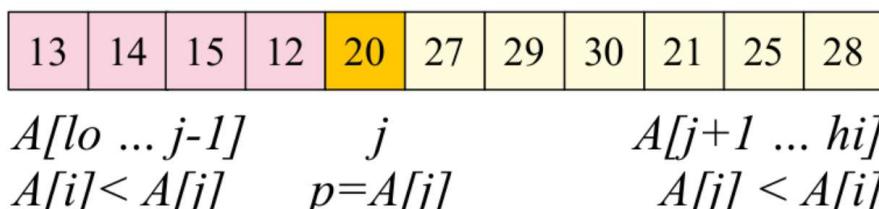
Si consideri un vettore $A[1, \dots, n]$ con indici lo, hi tali che $1 \leq lo \leq hi \leq n$:



La parte evidenziata in blu rappresenta la parte del vettore che si vuole ordinare.

Essendo un algoritmo di "divide-et-impera" si deve prima dividere, che è la parte più complessa, quindi si identifica un valore $p \in A[lo, \dots, hi]$ detto perno (*pivot*).

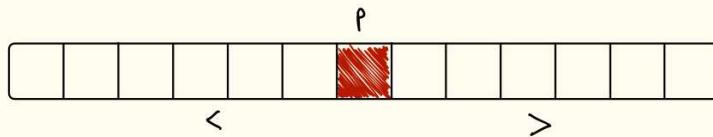
L'operazione di "divide" sposta tutti gli elementi del sottovettore a cui si fa riferimento in modo che il valore del perno sia poi posizionato in un certo punto del vettore e questo punto si indica con "j".



Tutti gli elementi più piccoli del valore del perno devono essere collocati prima del perno stesso e tutti gli elementi più grandi devono essere collocati dopo. I valori collocati prima e dopo non devono essere interamente ordinati perché si è ancora nella fase di "divide", ovvero

si porta il perno al centro in cui appunto la parte prima contiene solo valori minori di esso. A questo punto "impera" ordina i due sottovettori $A[lo, \dots, j - 1]$ e $A[j + 1, \dots, hi]$, richiamando ricorsivamente il *QuickSort*. Queste due parti potrebbero avere dimensione 0 nel caso " $j-1$ " sia più piccolo di " lo ".

La parte del "combina" non fa nulla, questo perché:



Tutti gli elementi più piccoli del perno stanno prima dello stesso, mentre tutti gli elementi più grandi stanno dopo ed entrambe le parti vengono riordinate tramite chiamate ricorsive sulle stesse parti. Alla fine tutti gli elementi sono ordinati nei sottovettori.

7.6 Pivot

int pivot(ITEM[] A, int lo, int hi)

```

ITEM pivot = A[lo] int j=lo
for i do=lo+1 to hi do
  if A[i] < pivot then j=j+1 swap(A, i, j)
  end if
end for
A[lo]=A[j]
A[j]=pivot
return j

```

swap(ITEM[] A, int i, int j)

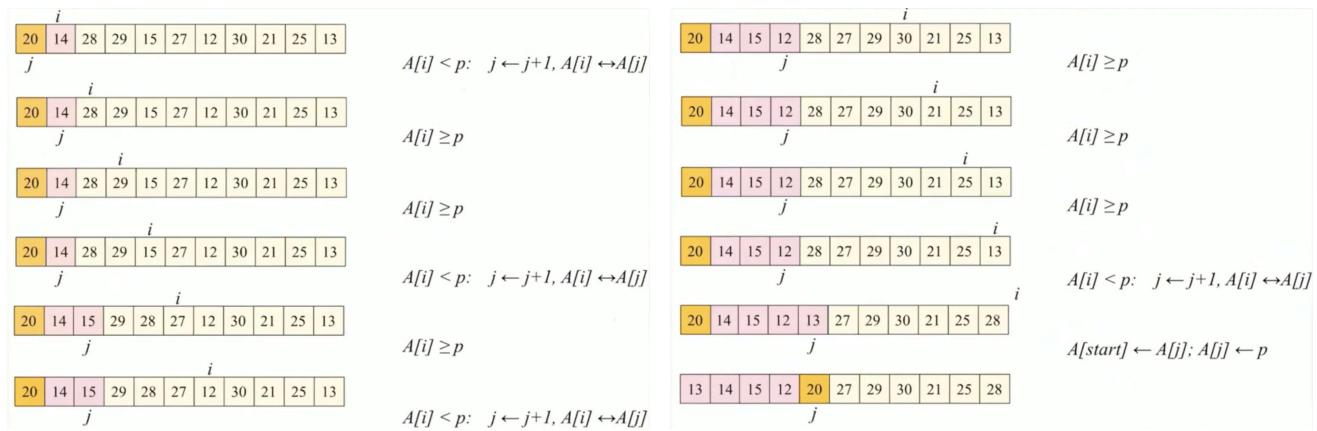
```

ITEM temp = A[i]
A[i] = A[j]
A[j] = temp

```

Questa versione del *pivot* sceglie come valore del pivot, appunto, quello che si trova in prima posizione. Il ciclo va dalla seconda casella fino ad arrivare all'ultima per poi uscire e cerca di capire cosa fare del valore in i . La parte del ciclo sposta i valori in modo che tutti quelli più grandi si trovano dopo e quelli piccoli prima, confrontando ogni valore $A[i]$ col pivot. Se trova un valore più piccolo del pivot esso va prima del perno.

La parte dopo il ciclo mette il perno al centro.



Il costo di $pivot()$ è: $\theta(n)$.

7.7 Quicksort: procedura principale

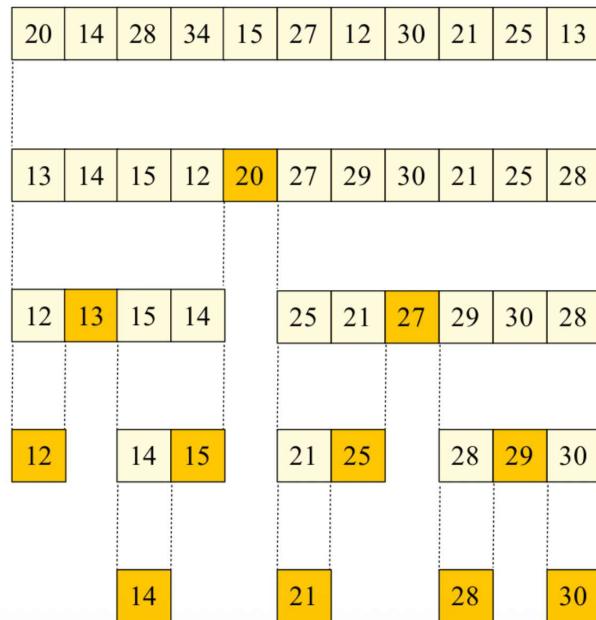
QuickSort(ITEM[] A, int lo, int hi)

```

if lo<hi then
    int j=pivot(A, lo, hi)
    QuickSort(A, lo, j-1)
    QuickSort(A, j+1, hi)
end if

```

Svolgimento ricorsione:



7.7.1 Caso pessimo

Nel caso pessimo il vettore è già ordinato, o in maniera crescente o in maniera decrescente, e si ottiene un costo pari a: $T(n) = T(n - 1) + T(0) + \theta(n) = T(n - 1) + \theta(n) = \theta(n^2)$

7.7.2 caso ottimo

Se il perno è sempre i valore mediano, ovvero quello che ha un numero di valori più piccoli e più grandi pari, allora la scelta del perno permette di dividere il vettore in due sottoparti più o meno uguali. In questo caso, il vettore con n elementi, viene sempre diviso in due sottoproblemi di dimensione $n/2 \rightarrow T(n) = 2T(n/2) + \theta(n) = \theta(n \log n)$.

Il problema è che il vettore non è sempre ben diviso in due parti. Il partizionamento nel caso medio di *QuickSort* è molto più vicino al caso ottimo che al caso peggiore (perché non ha senso ordinare un vettore già ordinato, quindi se si vuole effettuare un ordinamento ci saranno degli elementi sparsi nel vettore).

Nella realtà il caso medio dipende dall'ordine degli elementi e non dai loro valori, si devono considerare tutte le possibili permutazioni e può risultare difficile dal punto di vista analitico, ma si può arrivare a un'intuizione ovvero che alcuni partizionamenti saranno parzialmente bilanciati, mentre altri saranno pessimi e in media questi si alternano nella sequenza di partizionamento. I partizionamenti parzialmente bilanciati dominano quelli pessimi, ovvero dividono man mano il vettore in parti sempre più piccole.

Se si fa un'analisi probabilistica su tutte le possibili permutazioni si vede che, nel caso medio, il costo è $\theta(n \log n)$, quindi nel caso medio l'algoritmo ha lo stesso costo computazionale del *MergeSort*.

I fattori moltiplicativi, anche considerando le partizioni più sfavorevoli, sono comunque in favore del *QuickSort*.

7.8 Moltiplicazione di matrici

Siano A e B due matrici rispettivamente di dimensione $n_i * n_k$ e $n_k * n_j$, con n_k comune, e sia C il prodotto tra A e B , per calcolare $c_{i,j} = \sum_{k=1}^{n_k} a_{i,k} * b_{k,j}$: Per ogni $i \in [1, \dots n_i]$ e per ogni

```
matrixProduct(ITEM[][] A, B, C, int ni, nk, nj)
for i do=1 to ni do
    for j do=1 to nj do
        C[i,j]=0
        for k do=1 to nk do
            C[i,j] = C[i,j]+A[i,k]*B[k,j]
        end for
    end for
end for
```

$j \in [1, \dots n_j]$ si deve tradurre la sommatoria in codice e diventa un ulteriore ciclo che va da 1 a n_k in cui si va ad accumulare la sommatoria.

Avendo un ciclo su n_i , un ciclo su n_j e un ciclo su n_k , la complessità sarà: $T(n) = \theta(n_i * n_k * n_j)$ e se le matrici hanno tutte le stesse dimensioni viene fuori una complessità pari a $\theta(n^3)$.

7.8.1 Algoritmo di Strassen

Le matrici $n * n$ (quindi quadrate, ma si può fare anche con matrici rettangolari, aggiungendo degli zeri per renderle delle stesse dimensioni) vengono suddivise in quattro matrici $n/2 * n/2$.

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \quad B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}$$

Si può calcolare la matrice come:

$$C = \begin{bmatrix} A_{1,1}B_{1,1} + A_{1,2}B_{2,1} & A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ A_{2,1}B_{1,1} + A_{2,2}B_{2,1} & A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{bmatrix}$$

Ma ci sono 8 moltiplicazioni matriciali, avendo come equazione di ricorrenza: $8T(n/2) + n^2$ con $n > 1$ e si ha sempre una complessità pari a $\theta(n^3)$.

Strassen ha trovato un modo per ridurre la complessità dividendo in ulteriori 7 sottomatrici:

$$\begin{cases} X_1 = (A_{11} + A_{22}) * (B_{11} + B_{22}) \\ X_2 = (A_{21} + A_{22}) * B_{11} \\ X_3 = A_{11} * (B_{12} - B_{22}) \\ X_4 = A_{22} * (B_{21} - B_{11}) \\ X_5 = (A_{11} + A_{12}) * B_{22} \\ X_6 = (A_{21} - A_{11}) * (B_{11} + B_{12}) \\ X_7 = (A_{12} - A_{22}) * (B_{21} + B_{22}) \end{cases}$$

Si trova un'equazione di ricorrenza:

$$T(n) = \begin{cases} 7T(n/2) + n^2, & n > 1 \\ 1, & n = 1 \end{cases} \rightarrow T(n) = \theta(n^{\log_2 7}) \approx \theta(n^{2.81}) \quad (1)$$

riducendo la complessità e avendo come calcolo finale:

$$C = \begin{bmatrix} X_1 + X_4 - X_5 + X_7 & X_3 + X_5 \\ X_2 + X_4 & X_1 + X_3 - X_2 + X_6 \end{bmatrix}$$

Strassen è stato il primo a scoprire che era possibile moltiplicare in meno di n^3 moltiplicazioni scalari.

7.9 Conclusioni

Quando applicare *divide-et-impera*?

Quando i costi risultano essere migliori del corrispondente algoritmo iterativo.

Ulteriori vantaggi

È facile parallelizzare e c'è un utilizzo ottimale della cache.

8 I Grafi

8.1 Introduzione

Cosa sono i grafi?

Un grafo è una struttura che serve per rappresentare relazioni tra elementi.

I grafi sono considerati insiemi di entità, dette **nodi** o **vertici**, che sono collegate a coppie tra loro da delle relazioni. Queste relazioni vengono dette **lati** o **archi**, e a questi possono essere attribuiti dei **pesi**, dando origine a grafi detti **grafi pesati**.

Un **grafo orientato** G è dunque una coppia (V, E) dove V è un insieme finito ed E è una relazione binaria in V .

Gli elementi dell'insieme V sono detti **vertici**, l'insieme V è quindi chiamato **insieme dei vertici** di G . Allo stesso modo gli elementi dell'insieme E sono detti **archi** e l'insieme E è quindi chiamato **insieme degli archi** di G .

Grafi orientati e non orientati

In un **grafo orientato**, ogni arco ha una direzione: significa che il collegamento va da un vertice a un altro, in un verso preciso.

In un **grafo non orientato**, invece, gli archi non hanno direzione: il collegamento tra due vertici vale in entrambi i sensi.

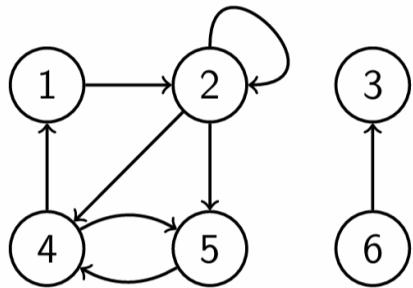
Quindi, mentre in un **grafo orientato** $G = (V, E)$, l'insieme degli archi E è composto da coppie ordinate di vertici, in un **grafo non orientato** $G = (V, E)$, l'insieme degli archi E è composto da coppie di vertici non ordinate.

8.1.1 Definizioni utili

- In un **grafo non orientato**, quando un arco (u, v) collega due vertici (in questo caso u e v), diciamo che l'arco è **incidente** nei vertici u e v , e che u e v sono **adiacenti** tra loro (cioè “collegati”). In questo caso, dato che il grafo non è orientato, l'adiacenza è bidirezionale, perchè anche l'arco è percorribile in entrambe le direzioni (da u a v e da v a u).
- In un **grafo orientato**, l'arco $[u, v]$ rappresenta la connessione dal vertice u al vertice v . In questo caso diciamo che l'arco è **incidente** nei vertici u e v ma con verso diverso, **uscente** da u ed **entrante** in v . In questo caso l'adiacenza è monodirezionale, segue il verso dell'arco, quindi diciamo che u è **adiacente** a v ma non è necessariamente vero il contrario (cioè a meno che non esista anche un arco $[v, u]$).
- Il **grado** di un vertice in un **grafo non orientato** coincide con il numero di archi incidenti in esso. Se un vertice ha grado 0, è **isolato**.
- In un **grafo orientato**, per un vertice, possono essere distinti tre tipi di grado. Il **grado uscente** di un vertice, è il numero di archi uscenti da esso; mentre il **grado entrante** è il numero di archi entranti nel vertice. Il **grado "totale"** del vertice è dato dalla somma di grado entrante e grado uscente, quindi dalla somma di tutti gli archi incidenti in esso.
- Un **sottografo** è un grafo formato da un sottoinsieme dei vertici e degli archi di un dato grafo, che mantiene la stessa struttura di connessioni.

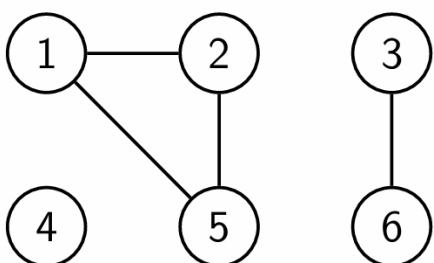
- Un arco che entra ed esce dallo stesso vertice è detto **cappio**.

8.1.2 Esempi



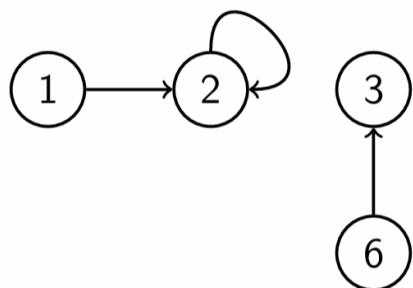
In questo caso abbiamo un grafo **orientato** con 6 nodi ($V = \{1, 2, 3, 4, 5, 6\}$) e 8 archi $E = \{[1, 2], [2, 2], [2, 4], [2, 5], [4, 1], [5, 4], [6, 3]\}$.

Dalle definizioni precedenti possiamo notare che l'arco $[2, 2]$ è un **cappio**.



In questo caso abbiamo un grafo **non orientato** con 6 nodi ($V = \{1, 2, 3, 4, 5, 6\}$) e 4 archi $E = \{(1, 2), (1, 5), (2, 5), (3, 6)\}$.

Dalle definizioni precedenti possiamo notare che il vertice 4 è **isolato**.



In questo caso abbiamo un **sottografo** del grafo orientato del primo esempio.

8.1.3 Cammini e raggiungibilità

Cosa si intende per *cammino*?

Un **cammino** è un modo per “muoversi” all’interno di un grafo passando da un vertice all’altro seguendo gli archi che li collegano.

Per essere più precisi, un cammino di lunghezza k che va da un vertice u a un vertice u' (in un grafo $G = (V, E)$), è una sequenza di vertici $\langle v_0, v_1, v_2, \dots, v_k \rangle$ che rispetta tre condizioni:

- Il primo vertice è il punto di partenza: $u = v_0$;
- L’ultimo vertice è il punto di arrivo: $u' = v_k$;
- Ogni coppia di vertici consecutivi nella sequenza è collegata da un arco del grafo, cioè: $(v_{i-1}, v_i) \in E$ per $i = 1, 2, \dots, k$

Conoscendo la definizione di **cammino**, possiamo dare la definizione di **lunghezza del cammino**. La lunghezza di un cammino è semplicemente il numero di archi che lo compongono. Quindi, se ho k archi, il cammino avrà lunghezza k .

Possiamo inoltre dire che un cammino è **semplice** se tutti i vertici che attraversa sono **distinti**, cioè non si passa mai due volte dallo stesso vertice.

Cosa si intende per *raggiungibilità*?

Diciamo che un vertice u' è **raggiungibile** da un altro vertice u attraverso un cammino p , se esiste un cammino p che parte dal vertice u e arriva al vertice u' .

Se il grafo è orientato il cammino p si indica $u \xrightarrow{p} u'$ (con la freccetta ondulata) e indica quindi un cammino che rispetti la direzione degli archi.

Cosa è un *sottocammino*?

Un **sottocammino** si un cammino $p = \langle v_0, v_1, v_2, \dots, v_k \rangle$, è una sottosequenza contigua dei suoi vertici. Questo vuol dire che non posso prendere ad esempio il primo e l'ultimo vertice saltando quelli in mezzo. Quindi ad esempio $\langle v_3, v_4, v_5 \rangle$ è un sottocammino di p , mentre $\langle v_3, v_6 \rangle$ non lo è.

Cosa è un *ciclo*?

In un grafo orientato $G = (V, E)$, un **ciclo** C di lunghezza $k > 2$ è una sequenza di nodi u_0, u_1, \dots, u_k tale che $(u_i, u_{i+1}) \in E$ per $0 \leq i \leq k - 1$ e $u_0 = u_k$. ($k > 2$ esclude cicli banali composti da coppie di archi (u, v) e (v, u) , che sono onnipresenti nei grafi non orientati)

Cicli: altre definizioni

In altre parole, in un **grafo orientato**, un cammino $\langle v_0, v_1, v_2, \dots, v_k \rangle$ forma un **ciclo** se $v_0 = v_k$ e il cammino **contiene almeno un arco**.

Quindi il **cappio** (o loop) che abbiamo visto prima, è un ciclo di lunghezza 1.

Inoltre se il cammino che forma il ciclo è **semplice**, anche il ciclo viene detto semplice.

Invece, in un **grafo non orientato**, un cammino $\langle v_0, v_1, v_2, \dots, v_k \rangle$ forma un **ciclo** (semplice) se: $v_0 = v_k$, $k \geq 3$, v_1, \dots, v_k sono distinti.

Si dice inoltre che un **grafo orientato senza cappi** è semplice mentre un grafo senza cicli è **aciclico**.

8.1.4 Connessione

La definizione di connessione è diversa per grafo orientato (per il quale bisogna fare una distinzione in più tra connessione **debole** e **forte**) e non orientato:

Quando un grafo non orientato si dice *connesso*?

Un grafo non orientato si dice **connesso** se ogni coppia di vertici è collegata tramite un cammino (eventualmente minimo). Ovvero se, per ogni coppia di vertici distinti $u, v \in V$ esiste almeno un cammino che collega u e v .

N.B. I vertici non devono essere connessi tutti direttamente (non devono necessariamente essere *adiacenti*), basta che ci sia un cammino che ti permetta di raggiungere un qualunque nodo (vertice) del grafo partendo da un qualunque altro nodo (vertice) dello stesso grafo.

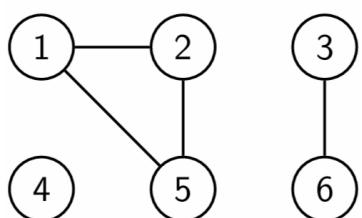
Quando un grafo orientato si dice *connesso*?

Un grafo orientato si dice **fortemente connesso** se due vertici qualsiasi sono raggiungibili l'uno dall'altro; quindi esiste un cammino orientato da u a v e viceversa.

È detto invece **debolmente connesso** se, ignorando la direzione degli archi (cioè trattandolo come se fosse non orientato), il grafo risultante è connesso.

Un'altra definizione importante è quella di **componente connessa**. Le **componenti connesse** di un grafo “sono le classi di equivalenza dei vertici secondo la relazione di (mutua) raggiungibilità”; ma vediamo cosa vuol dire per grafi orientati e non.

Quando un grafo non orientato **non è connesso**, può essere suddiviso in più parti isolate, dette **componenti connesse**. In pratica, all'interno di un grafo non connesso, possiamo individuare diversi sottografi massimali che, presi singolarmente, sono connessi. Diciamo sottografi **massimali** perché questo implica che l'aggiunta di un altro qualsiasi nodo del grafo non gli fa più rispettare la condizione di connessione. Facciamo un esempio:



In questo esempio i nodi $\langle 1, 2, 5 \rangle$ formano una componente connessa. Infatti se prendiamo il sottografo formato solo da questi 3 vertici, questo risulta connesso, e non possiamo aggiungere nessun altro vertice del grafo continuando ad avere una condizione di connessione.

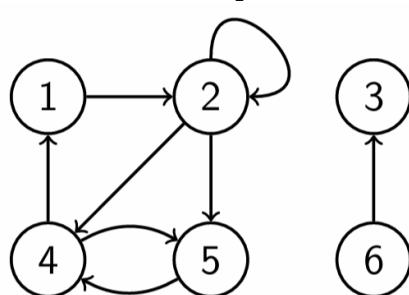
Se invece prendiamo come sottografo quello formato dai vertici $\langle 1, 2 \rangle$ e la connessione tra i due, questo non rappresenta una componente connessa in quanto se aggiungiamo il vertice 5 e i rispettivi archi di collegamento con i vertici 1 e 2, continua a mantenere la proprietà di connessione.

In questo esempio abbiamo infatti 3 componenti connesse:

- $\langle 1, 2, 5 \rangle$
- $\langle 3, 6 \rangle$
- $\langle 4 \rangle$

Grazie a questa definizione possiamo dire che un grafo non orientato è **connesso** se ha **una sola componente connessa**.

Dato che un grafo orientato può essere debolmente o fortemente connesso, dobbiamo dare la definizione di **componente fortemente connessa**. Funziona esattamente come nel caso precedente, solo che adesso gli archi sono direzionali, quindi il sottografo che prendiamo in esame dev'essere fortemente connesso. Anche in questo caso, un grafo si dice **fortemente connesso** se presenta **una sola componente fortemente connessa**. Anche in questo caso facciamo un esempio:



Anche in questo caso abbiamo 3 componenti fortemente connesse:

- $\langle 1, 2, 4, 5 \rangle$
- $\langle 3 \rangle$
- $\langle 6 \rangle$

3 e 6 non formano una componente fortemente connessa perché il nodo 3 è raggiungibile dal 6 ma non viceversa.

Invece $\langle 1, 2, 4, 5 \rangle$ formano una componente fortemente connessa perché ogni nodo è raggiungibile da un'altro dell'insieme. Ad esempio il nodo 1 è

collegato direttamente con il nodo 2 e, passando da questo nodo si possono raggiungere sia il nodo 4 che il nodo 5. Stesso ragionamento per gli altri nodi.

8.1.5 Dimensioni del grafo

Per descrivere un grafo, sono necessarie due quantità fondamentali: il numero di nodi e di archi nel grafo. Se si considerano $n = |V|$ come il numero totale di nodi nel grafo e $m = |E|$ come il numero totale di archi del grafo, è possibile definire alcune relazioni importanti tra queste due quantità, a seconda che il grafo sia orientato o meno:

- In un **grafo non orientato**, il numero di archi possibili è $m \leq \frac{n(n-1)}{2}$ perché ogni coppia di nodi può essere collegata al massimo da un arco. Quindi il numero di archi cresce, al massimo, proporzionalmente a n^2 ($O(n^2)$)
- In un **grafo orientato**, ogni coppia di nodi può avere due archi distinti (uno per ciascuna direzione), quindi il numero di archi è $m \leq n^2 - n$, che cresce anch'esso come n^2 ($O(n^2)$)

Quando si analizzano algoritmi che operano sui grafi, la loro complessità viene spesso espressa in funzione sia del numero di nodi n sia del numero di archi m . Ad esempio, un algoritmo può avere complessità $O(m + n)$, cioè proporzionale alla somma di nodi e archi del grafo, e questo può avvenire quando l'algoritmo deve "guardare" tutti i nodi e tutti gli archi almeno una volta.

8.1.6 Grafi sparsi e densi

Possiamo "catalogare" un grafo in base alle sue caratteristiche.

Ad esempio quando un grafo ha un arco tra tutte le coppie di nodi, (cioè quando ogni nodo è collegato "direttamente" a **tutti** gli altri nodi del grafo) è detto **completo**.

Un grafo inoltre può essere **sparsò** o **denso**:

- Un grafo è detto **sparsò** se ha pochi archi; grafi con ad esempio $m = O(n)$ o $m = O(n \log n)$ sono considerati sparsi.
- Un grafo è detto **denso** quando ha molti archi, come ad esempio $m = \Omega(n^2)$

Possiamo quindi dire che un grafo **completo** è anche **denso**, perché è quello che contiene il numero massimo di archi possibili rispetto ai nodi ed ha quindi la densità più alta che un grafo può avere. Ovviamente non è vero anche il contrario, un grafo denso non necessariamente è completo.

8.1.7 Isomorfismo

Introduciamo ora questa relazione tra 2 grafi:

Quando si dice che due garfi sono *isomorfi*?

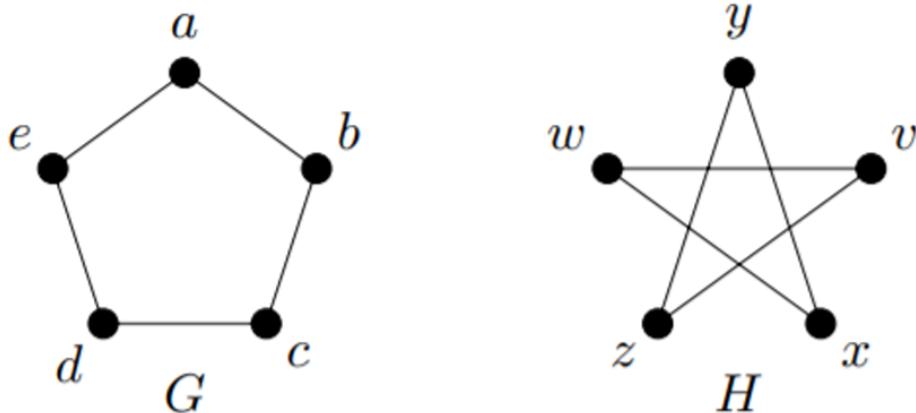
Due grafi $G = (V, E)$ e $G' = (V', E')$ sono **isomorfi** se esiste una funzione biiettiva (cioè una corrispondenza uno-a-uno) $f : V \rightarrow V'$ tale che $(u, v) \in E$ se e soltanto se $(f(u), f(v)) \in E'$.

Cioè se è possibile rietichettare i vertici di G come i vertici di G' mantenendo gli archi corrispondenti in G e G' .

In altre parole possiamo dire che due grafi sono isomorfi se:

- hanno lo stesso numero di vertici e di archi;
 - la connessione tra i vertici è la stessa, anche se i nomi o la disposizione dei vertici cambiano.
- Quindi se due vertici sono adiacenti in un grafo lo saranno anche nell'altro.

Per fissare meglio il concetto vediamo alcuni esempi:



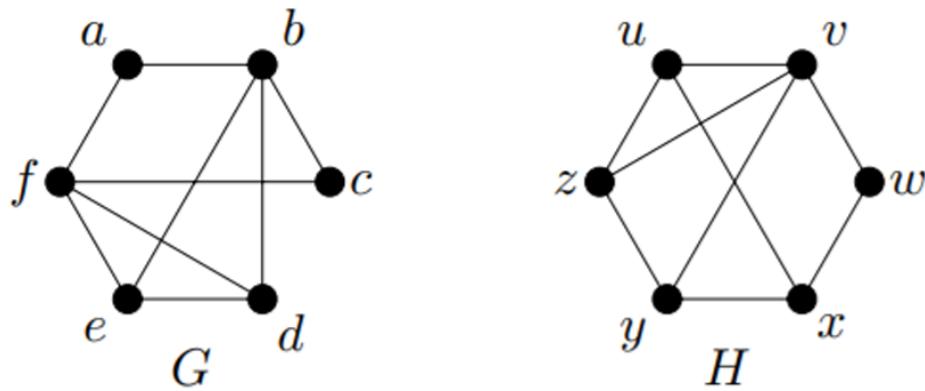
Questi due grafi sono **isomorfi**. Innanzitutto possiamo notare che hanno lo stesso numero di nodi e di archi, oltretutto tutti e 5 i nodi di entrambi i grafi hanno valenza (o grado) 2.

Nel grafo G abbiamo il nodo a adiacente ai nodi b ed e , il nodo c adiacente a b e d e quest'ultimo adiacente al nodo e ; nel grafo H, se partiamo dal nodo v , notiamo che questo è adiacente ai nodi w e z , il nodo x adiacente a w e y e quest'ultimo adiacente al nodo z .

Possiamo quindi vedere una corrispondenza del tipo:

- $a \iff v$ quindi definiamo $f(a) = v$
- $b \iff w$ quindi definiamo $f(b) = w$
- $c \iff x$ quindi definiamo $f(c) = x$
- $d \iff y$ quindi definiamo $f(d) = y$
- $e \iff z$ quindi definiamo $f(e) = z$

Vediamo un altro esempio:



In questo caso invece i due grafi **non sono isomorfi**. Infatti, nonostante abbiano entrambi 6 nodi e 9 archi, il grafo G ha 2 vertici di valenza 2 (a e c), 2 vertici di valenza 3 (d ed e) e 2 vertici di valenza 4 (b ed f), mentre il grafo H ha un vertice di valenza 2 (w), 4 vertici di valenza 4 (u, x, y e z) quindi non è possibile trovare un'equivalenza come invece è stato fatto nel caso precedente. (Ricordiamo che il grado di valenza dei vertici in un grafo non orientato è il numero di archi incidenti in esso)

8.1.8 Relazioni tra alberi e grafi

Esistono delle relazioni tra alberi e grafi, infatti, quando un grafo rispetta determinate caratteristiche, può essere considerato un albero.

Il primo tipo di albero che andiamo a considerare, quello con i requisiti meno stringenti, è il cosiddetto *albero libero*.

Quando un grafo è un albero libero?

Un grafo per poter essere considerato un **albero libero** deve rispettare due requisiti fondamentali:

- **deve essere connesso** → deve essere possibile raggiungere qualsiasi nodo partendo da un qualsiasi altro nodo, seguendo gli archi;
- **deve essere aciclico** → non devono esistere cicli, quindi percorsi chiusi che ti permettano, partendo da un nodo, di tornare allo stesso senza ripercorrere gli stessi archi.

Queste due proprietà possono essere riassunte dicendo che il numero di archi m deve essere esattamente uno in meno rispetto al numero dei nodi n : $m = n - 1$. Infatti, se ci fossero meno archi il grafo non sarebbe connesso, e se ce ne fossero di più il grafo conterebbe almeno un ciclo.

Una volta che si ha in mente cosa è un albero libero, possiamo capire cosa si intende per *albero radicato*.

Quando invece è un albero radicato?

Un **albero radicato** infatti è un albero libero nel quale è stato scelto un nodo come radice. A questo punto possiamo immaginare i nodi come disposti su più livelli, in base alla loro distanza dalla radice. Avremo quindi la radice su un livello, i suoi figli sul livello successivo, i figli dei suoi figli in quello dopo ancora, e così via. Viene stabilito quindi un ordinamento "verticale" tra i nodi.

A partire dagli alberi radicati possiamo definire un'ulteriore tipo di grafi che possono essere visti come alberi, ossia gli *alberi ordinati* (*ordered tree*)

Quando è un albero ordinato?

Un **albero ordinato** è un albero radicato nel quale viene deciso un ordine preciso tra i nodi sullo stesso livello, quindi tra i nodi figli di un certo nodo. In questo caso quindi il grafo dovrà rispettare tutte le caratteristiche elencate per i precedenti due, viene però aggiunto un ordinamento "orizzontale" tra i nodi.

Possiamo ora passare alla definizione di ciò che viene detta *foresta*.

Cosa è una foresta?

Una **foresta** è banalmente un insieme di alberi. È quindi un grafo non connesso, che presenta diverse componenti connesse, che rispettano le caratteristiche elencate sopra per poter essere considerati alberi.

8.2 Specifica

Prima di iniziare diamo una brevissima spiegazione di ciò che intendiamo con "specifica" in questo contesto. Una specifica è una descrizione formale delle operazioni che una struttura dati deve offrire, senza dire come è implementata.

In pratica daremo una descrizione astratta delle operazioni disponibili e dei loro effetti, senza entrare nel dettaglio della loro implementazione.

Altra piccola parentesi: poiché ogni grafo non orientato G può essere visto come un grafo orientato G' , ottenuto da G , sostituendo ogni arco $[u, v]$ con i due archi (u, v) e (v, u) , sarà fornita una specifica solo per grafi orientati, che saranno chiamati semplicemente grafi.

8.2.1 Grafi dinamici

Nella versione più generale, il grafo è una struttura dati dinamica che permette di aggiungere e rimuovere nodi e archi, per la quale non esiste uno standard universalmente adottato. Possiamo quindi vedere le principali operazioni che possono essere fatte sui grafi:

```
1 Graph()           //Crea un nuovo grafo
2 SET V()          //Restituisce l'insieme di tutti i nodi
3 int size()        //Restituisce il numero dei nodi
4 Set adj(NODE u)  //Restit. l'insieme dei nodi adiacenti a u
5 insertNode(NODE u) //Aggiunge il nodo u al grafo
6 insertEdge(NODE u, NODE v) //Aggiunge l'arco (u,v) al grafo
7 deleteNode(NODE u) //Rimuove il nodo u dal grafo
8 deleteEdge(NODE u, NODE v) //Rimuove l'arco (u,v) dal grafo
```

8.2.2 Specifica ridotta - senza rimozioni

In alcuni casi non è necessaria una struttura dinamica completa. Spesso, infatti, si utilizza un grafo che viene caricato inizialmente e che non viene più modificato, tranne eventualmente per l'aggiunta di nuovi nodi o archi. In questi casi utilizziamo grafi non dinamici, in cui sono permessi solo inserimenti ma non rimozioni.

```
1 Graph()           //Crea un nuovo grafo
2 SET V()          //Restituisce l'insieme di tutti i nodi
3 int size()        //Restituisce il numero dei nodi
4 Set adj(NODE u)  //Restit. l'insieme dei nodi adiacenti a u
5 insertNode(NODE u) //Aggiunge il nodo u al grafo
6 insertEdge(NODE u, NODE v) //Aggiunge l'arco (u,v) al grafo
```

Questo ha conseguenze anche sull'implementazione del grafo stesso.

8.3 Memorizzazione

La memorizzazione di un grafo può avvenire tramite uno dei due possibili approcci:

- tramite **matrici di adiacenza**
- tramite **liste di adiacenza**

8.3.1 Matrici di adiacenza

La memorizzazione di un grafo tramite matrici di adiacenza, è uno dei modi più semplici per rappresentare un grafo.

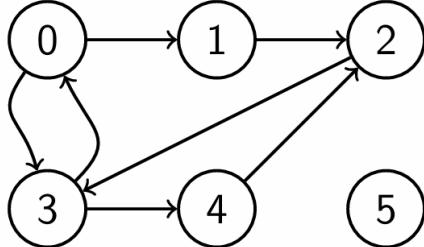
Matrice di adiacenza

Dato un grafo $G = (V, E)$, la **matrice di adiacenza** $M = [m_{uv}]$, è una matrice $n \times n$ tale che

$$m_{uv} = \begin{cases} 1 & \text{se } (u, v) \in E \\ 0 & \text{se } (u, v) \notin E \end{cases}$$

Grafi orientati

Quando il grafo è orientato lo spazio in memoria occupato dalla matrice è n^2 bit



	0	1	2	3	4	5
0	0	1	0	1	0	0
1	0	0	1	0	0	0
2	0	0	0	1	0	0
3	1	0	0	0	1	0
4	0	0	1	0	0	0
5	0	0	0	0	0	0

Come possiamo vedere dall'esempio, abbiamo un bit per ogni possibile arco, che è a 0 se l'arco tra quei nodi non esiste ed è a 1 quando l'arco esiste. In questo caso abbiamo i nodi di partenza degli archi scritti in colonna (che "numerano" quindi le righe) e i nodi di arrivo scritti in riga (che "numerano" quindi le colonne). Possiamo quindi vedere come esiste un arco da 0 a 1 in quanto il bit nella posizione $[0][1]$ (riga 0 e colonna 1) è a 1, ma non esiste il collegamento inverso, da 1 a 0, in quanto il bit $[1][0]$ (riga 1 e colonna 0) è a 0. Diventa quindi intuitivo capire che, se nel grafo non sono presenti *cappi* (quelli che si hanno quando un nodo punta a se stesso) la diagonale principale sarà composta da soli 0, proprio come nell'esempio.

Grafi non orientati

Quando il grafo non è orientato lo spazio in memoria occupato dalla matrice può essere di n^2 bit se vengono memorizzate tutte le celle della matrice come nel caso precedente, ma può essere inferiore se si nota che la matrice risultante sarebbe simmetrica rispetto alla diagonale principale, e quindi gli archi verrebbero memorizzati 2 volte: $m_{uv} = m_{vu}$.

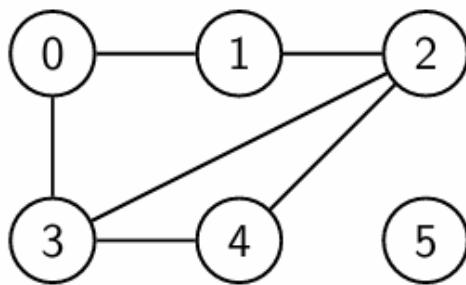
Per poter capire di quanti bit abbiamo bisogno, dobbiamo prima sapere se i *cappi* sono ammessi o meno, e quindi se dobbiamo o no memorizzare anche la diagonale.

Con $\frac{n^2}{2}$ possiamo memorizzare uno dei "triangoli" sopra o sotto la diagonale, più metà diagonale.

Se abbiamo quindi bisogno di memorizzare anche la diagonale abbiamo bisogno di altri $\frac{n}{2}$ bit e raggiungiamo un totale di $\frac{n(n+1)}{2}$ bit.

Se invece non abbiamo bisogno della diagonale, non abbiamo bisogno nemmeno di quei bit

che ne memorizzavano metà, e li dobbiamo quindi sottrarre da $\frac{n^2}{2}$, ottenendo un totale di $\frac{n(n-1)}{2}$ bit necessari.



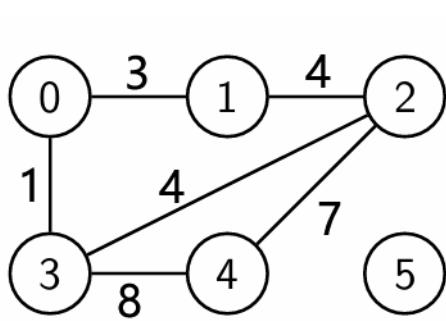
$$\begin{array}{ccccccc} & 0 & 1 & 2 & 3 & 4 & 5 \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \left(\begin{array}{cccccc} 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & & & & & 0 \end{array} \right) \end{array}$$

Grafi pesati

Quando abbiamo un grafo pesato andiamo a memorizzare nella matrice il peso p attribuito a quell'arco e non i precedenti 0/1 che indicavano solo l'esistenza o non esistenza dell'arco. La definizione della matrice di adiacenza diventa quindi:

$$m_{uv} = \begin{cases} p & \text{se } (u, v) \in E \\ 0 \text{ o } \pm \infty & \text{se } (u, v) \notin E \end{cases}$$

Il valore associato ad un arco non esistente cambia in base al problema, l'importante è che non sia un valore ammesso per i pesi degli archi esistenti.



$$\begin{array}{ccccccc} & 0 & 1 & 2 & 3 & 4 & 5 \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \left(\begin{array}{cccccc} 3 & 0 & 1 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 \\ 4 & 7 & 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 0 & 0 & 0 \\ 0 & & & & & 0 \end{array} \right) \end{array}$$

Lo spazio che occupano in memoria è quindi il numero di bit visti nei casi precedenti (che sarebbe il numero di celle che vado a memorizzare), moltiplicato per il numero di bit scelti per la memorizzazione dei pesi. Se quindi ho valori dei pesi ammessi solo da 0 a 15 userò 4 bit per ogni arco, e ogni cella della matrice varrà 4 bit, per questo moltiplico il numero delle celle per il numero dei bit. Prima invece ogni singola cella era da un solo bit, quindi non abbiamo moltiplicato per 1.

Complessità

Con l'implementazione tramite matrice verificare la presenza di un determinato arco richiede un tempo $O(1)$. Invece iterare su tutti gli archi richiede un tempo $O(n^2)$ sia per un grafo denso, che per uno sparso in quanto bisogna iterare in ogni caso su tutte le celle della matrice. Per questo l'utilizzo di un'implementazione tramite matrice è consigliata **solo in caso di grafi densi**. Inoltre un altro problema relativo a quest'implementazione è che, se si ha un numero elevato di nodi, n^2 può diventare enorme; si pensi che con soli 1000 nodi abbiamo 10^6 bit che sono circa 125KB e con 10.000 nodi arriviamo a 12,5MB.

8.3.2 Liste di adiacenza

Un altro modo per rappresentare un grafo, è tramite le **liste di adiacenza**.

L'idea alla base è quella di registrare solo i collegamenti che esistono davvero: per ogni nodo si mantiene una lista dei suoi vicini, invece di rappresentare tutte le possibili coppie di nodi come farebbe una matrice.

Lista di adiacenza

Una **lista di adiacenza** è un modo per rappresentare un grafo associando a ogni nodo l'elenco dei nodi a cui è collegato tramite un arco.

Si usa quindi un vettore con n celle, una per ciascun nodo, dove ogni cella conterrà il riferimento alla lista contenente tutti i nodi adiacenti al nodo in esame, quindi gli elementi $G.adj(u) = \{v \mid (u, v) \in E\}$.

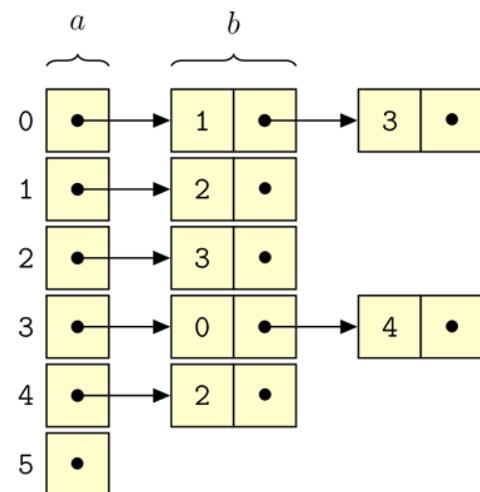
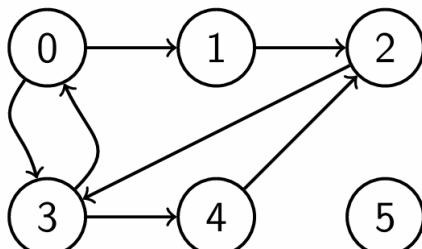
Solitamente vengono realizzate con delle **linked list** dove ogni elemento punta al successivo, anche se questa scelta non è obbligata, infatti qualunque struttura che permetta di memorizzare i vicini del nodo è adatta.

In questo modo la struttura è compatta ed efficiente, soprattutto quando il grafo è **sparso**, e permette di accedere facilmente ai nodi adiacenti e di iterare sugli archi del grafo.

Grafi orientati

Cosa rappresenta $G.adj(u) = \{v \mid (u, v) \in E\}$ in questo caso?

$G.adj(u)$ rappresenta l'insieme dei nodi del grafo G adiacenti al nodo u . Cioè l'insieme dei nodi che vogliamo memorizzare per ogni nodo del grafo. La formula ci dice che questi nodi rispettano la caratteristica $\{v \mid (u, v) \in E\}$, cioè si trovano nell'insieme formato da tutti i nodi v tali che l'arco **da u a v** esista nell'insieme degli archi di G .



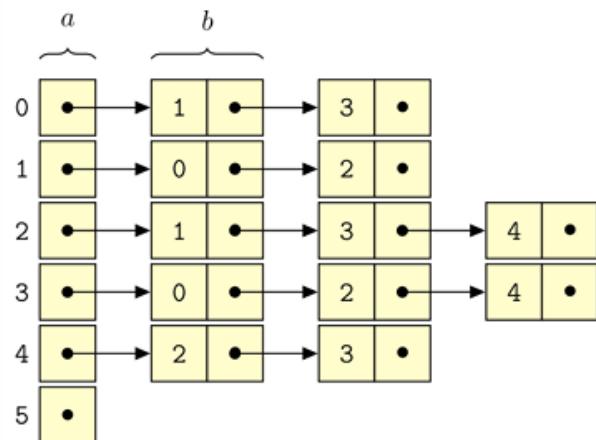
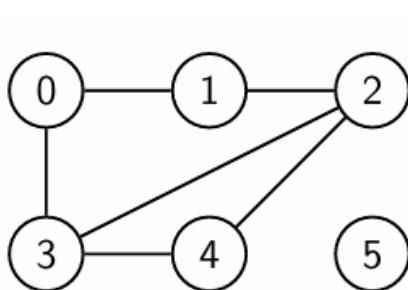
Lo spazio totale è dato da una parte fissa, un puntatore per ogni nodo ($a \times n$ bit, dove a è il numero di bit che occupa ogni cella, quindi ogni puntatore), e da una parte variabile, un elemento per ogni arco ($b \times m$ bit, dove b è il numero di bit che occupa ogni cella della lista, quindi bit del valore + bit del puntatore). Sommando otteniamo $an + bm$ bit occupati. Siccome a e b sono costanti, questo corrisponde a uno spazio $\Theta(n + m)$, cioè proporzionale al numero di nodi e di archi del grafo.

Grafi non orientati

Cosa rappresenta $G.\text{adj}(u) = \{v \mid (u, v) \in E\}$ in questo caso?

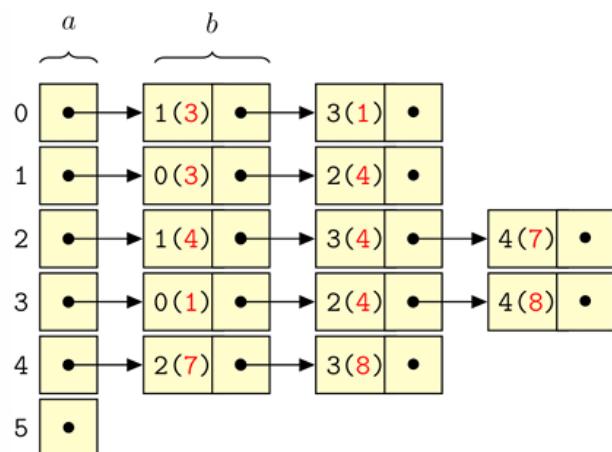
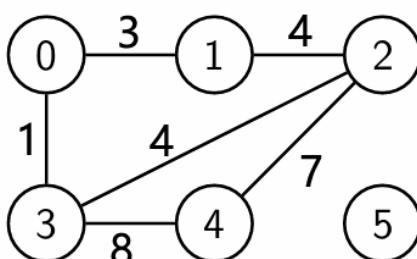
Vale lo stesso ragionamento visto precedentemente, però cambia leggermente l'interpretazione di $\{v \mid (u, v) \in E\}$, in questo caso si riferisce all'insieme formato da tutti i nodi v tali che l'arco (u, v) esista nell'insieme degli archi di G .

Inoltre il fatto che il grafo non è orientato implica la scrittura doppia di ogni arco, una per ogni nodo coinvolto.



In questo caso lo spazio totale è dato da $a n + 2 \cdot b m$ bit, il "2" è dato dal fatto che ogni arco viene memorizzato 2 volte. In ogni caso, siccome a e b sono costanti, questo corrisponde a uno spazio $\Theta(n + m)$, cioè proporzionale al numero di nodi e di archi del grafo, proprio come nel caso precedente

Grafi pesati



In questo caso il grafo non è orientato e quindi valgono le stesse cose dette precedentemente. L'unica differenza è che dobbiamo memorizzare anche il peso di ogni arco, quindi la costante b coltre a contenere i bit necessari per nodo + puntatore, ha anche i bit necessari per memorizzare il peso. Quindi sempre uno spazio $\Theta(n + m)$.

Complessità

Come visto prima lo spazio in memoria richiesto è $\Theta(n)$ per il vettore con i nodi e $\Theta(|G.adj(u)|)$ per ciascuna lista con un totale di $\Theta(n + \sum_{u \in V} |G.adj(u)|) = \Theta(n + m)$, che è ottimo.

La verifica della presenza di un arco richiede tempo $O(n)$, poiché è necessario scorrere la lista dei vicini di u . Invece, l'iterazione su tutti gli archi del grafo richiede tempo $O(n + m)$, dato che si attraversano tutte le liste di adiacenza.

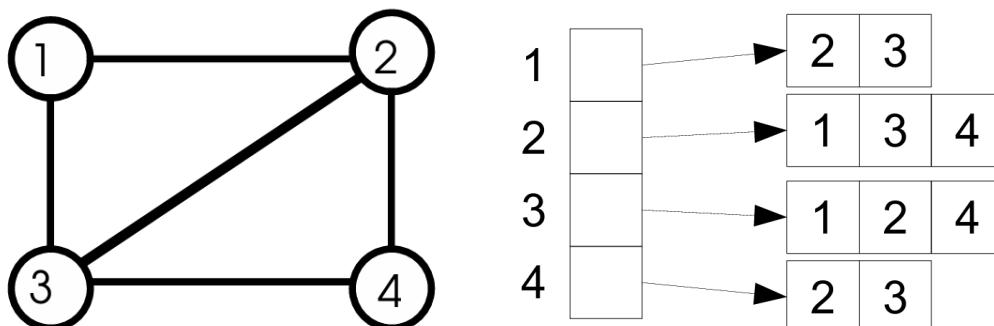
Queste complessità valgono allo stesso modo per grafi orientati, non orientati e pesati, poiché l'eventuale presenza di pesi non modifica la struttura delle liste ma solo il contenuto di ogni elemento. È ideale per i grafi **sparsi**.

8.3.3 Implementazione

Mentre l'implementazione di una matrice è fissa, esistono diversi modi per implementare una **lista di adiacenza**. La tabella seguente mostra le possibili strutture dati con cui è possibile implementarle:

Struttura	Java	C++	Python
Lista collegata	LinkedList	list	
Vettore statico*	[]	[]	[]
Vettore dinamico**	ArrayList	vector	list
Insieme	HashSet TreeSet	set	set
Dizionario	HashMap TreeMap	map	dict

In questo corso, se non diversamente specificato, si farà riferimento a un'implementazione basata su **vettori di adiacenza**.



Vettori di adiacenza

La rappresentazione tramite **vettori di adiacenza** utilizza un vettore, statico o dinamico, adj di dimensione n , indicizzato dai nodi del grafo (cioè ogni indice del vettore corrisponde al nodo con quell'identificatore).

Ogni elemento $adj[u]$ contiene un vettore di adiacenza (il riferimento al suo primo elemento), che memorizza tutti i nodi v adiacenti a u .

Inoltre assumeremo che:

- La classe `Node` sia uguale a `int` e che di conseguenza l'accesso alle informazioni abbia costo $O(1)$;
- Le operazioni per aggiungere nodi e archi abbiano costo $O(1)$ (per i vettori dinamici, anche se il costo dell'append non vale sempre $O(1)$, si fa riferimento all'analisi ammortizzata, dove il costo rimane $O(1)$);
- Dopo l'inizializzazione, il grafo sia **statico**, quindi una volta creato non cambia più (ad esempio viene mandato in input come file di testo).

Riassumendo

Per semplicità, assumiamo che il grafo sia rappresentato con liste di adiacenza basate su vettori, che i nodi siano semplici interi, che accedere e inserire elementi costi $O(1)$, e che il grafo non cambi mentre eseguiamo gli algoritmi.

8.3.4 Iterazione su nodi e archi

Precedentemente abbiamo discusso delle complessità per matrici e liste di adiacenza. Ora risulta quindi utile formalizzare lo schema di iterazione tipico utilizzato negli algoritmi sui grafi.

L'iterazione su tutti i nodi del grafo, rappresentata dallo pseudocodice seguente, ha costo $O(n)$, moltiplicato per il costo dell'operazione eseguita su ciascun nodo. Se l'operazione all'interno del ciclo ha costo c allora avremo un costo $O(n \cdot c)$; bisogna tenere a mente però che, se tale costo è costante per ogni nodo ($c = O(1)$), esso può essere trascurato e si ottiene nuovamente $O(n)$.

```
foreach  $u \in G.V()$  do
    /* Esegui operazioni sul nodo  $u$  */
end for
```

Se ho invece bisogno di iterare su tutti i nodi e tutti gli archi del grafo, operazione rappresentata dal seguente pseudocodice, avrà un costo computazionale $O(n + m)$ se il grafo è implementato tramite liste di adiacenza, e $O(n^2)$ se rappresentato tramite matrici.

```
foreach  $u \in G.V()$  do
    /* Esegui operazioni sul nodo  $u$  */
    foreach  $v \in G.adj(u)$  do
        /* Esegui operazioni sull'arco  $(u, v)$  */
    end for
end for
```

8.4 Visite

Visite: definizione generale

Una *visita* di un grafo è una procedura sistematica per esplorare un grafo, visitando almeno una volta ogni suo nodo ed arco.

Noi andremo però ad esaminare un problema più specico che presenta due famiglie principali di soluzione.

Caso di studio

Dato un grafo $G(V, E)$ e un vertice (radice o sorgente) $r \in V$, visitare una ed una sola volta tutti i nodi del grafo che possono essere raggiunti da r

Per "risolvere" il problema abbiamo due strade percorribili:

- **Visita in ampiezza - BFS** (Breadth-First Search);
 - **Visita in profondità - DFS** (Depth-first Search);
- che verranno aprofondite nel corso di questo capitolo.

Intanto rivediamo l'idea generale dietro queste famiglie di algoritmi che abbiamo già visto parlando degli alberi:

BFS

In questo genere di algoritmi la visita dei nodi avviene per livelli: prima si visita la radice, poi i nodi a distanza 1 dalla radice, poi quelli a distanza 2 ecc.

Una possibile applicazione è quella del calcolo dei cammini più brevi a partire da una singola sorgente.

DFS

In questo tipo di algoritmi la visita dei nodi avviene in profondità: a partire da un nodo, si visita uno dei suoi nodi adiacenti e si continua l'esplorazione lungo un cammino finché possibile, prima di tornare indietro.

È tipicamente implementata in modo ricorsivo ed è utilizzata, tra le altre cose, per l'ordinamento topologico e per il calcolo delle componenti connesse e fortemente connesse.

8.4.1 Visita: più complessa di quanto sembri

Per visitare un grafo si potrebbe pensare di ciclare per ogni nodo e per ogni arco, come nello pseudocodice seguente. Così facendo però non si tiene in considerazione la struttura del grafo e si itera su tutte le possibili coppie di nodi, non solo le coppie adiacenti. L'iterazione avviene quindi senza un criterio preciso.

Visit(Graph G)

```
foreach  $u \in G.V()$  do
    /* Visita il nodo  $u$  */
    foreach  $v \in G.V()$  do
        /* Visita l'arco  $(u, v)$  */
        // anche se, come abbiamo detto prima, la coppia  $(u, v)$  non corrisponde
        // necessariamente ad un arco; non è detto che  $u$  e  $v$  siano adiacenti
    end for
end for
```

Si potrebbe quindi pensare di visitare il grafo trattandolo come un albero, visitando quindi i nodi vicini al nodo in esame. Questo approccio è trattato dal codice seguente dove r è il nodo di partenza scelto come radice e Q è la coda in cui andremo a memorizzare i nodi vicini

BFSTraversal(Graph G , int r)

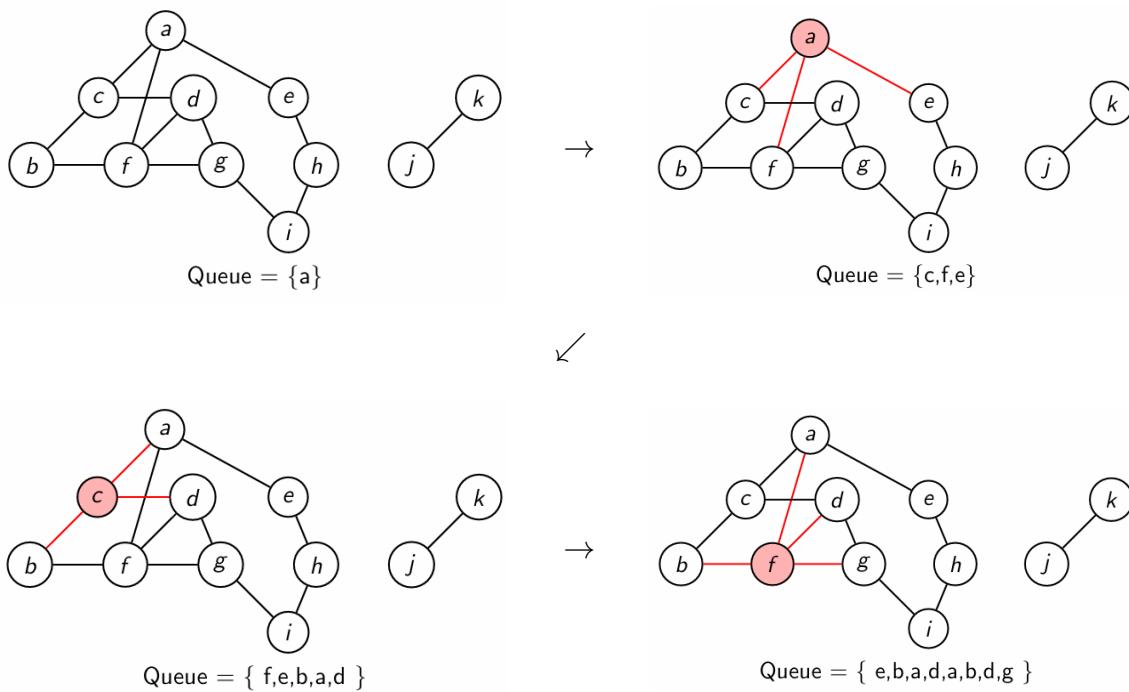
```

QUEUE  $Q$  = Queue()
 $Q$ .enqueue( $r$ )    // inserire  $r$  in fondo alla coda
while not  $Q$ .isEmpty() do
    Node  $u$  =  $Q$ .dequeue()    // prendere il primo elemento in testa alla coda
    /* visita il nodo  $u$  */
    foreach  $v \in G.\text{adj}(u)$  do
         $Q$ .enqueue( $v$ )    // inserire i vicini di  $u$  in fondo alla coda
    end for
end while

```

Questa soluzione potrebbe sembrare corretta a primo impatto ma non tiene conto dei **nodi già visitati**. Quindi, se ad esempio avessimo un grafo che contiene un ciclo, la coda crescerebbe in maniera sconsiderata e continueremmo a **visitare i nodi all'infinito**.

Vediamo un esempio passo passo:



Andando avanti il prossimo nodo esplorato sarebbe e , e si aggiungerebbero alla coda i nodi a e h . Già da qui notiamo che molti nodi si ripetono nella coda, e ogni volta che vengono rivisitati aggiungono i propri vicini alla coda, che riaggiungeranno a loro volta anche il nodo che li ha "chiamati", andando avanti in un loop infinito.

8.4.2 Algoritmo generico di attraversamento

Per risolvere i problemi di terminazione e ridondanza visti in precedenza, si introduce un algoritmo di attraversamento generico. La novità cruciale è l'uso di un meccanismo di marcatura (riga 3 e 10): un nodo viene aggiunto alla struttura di supporto solo se non è stato ancora visitato, garantendo che ogni vertice venga elaborato una sola volta.

visit(GRAPH G)

```
1: SET  $S = \text{Set}()$  // Insieme generico
2:  $S.\text{insert}(r)$  // Da specificare
3: /* marca il nodo  $r$  */
4: while  $S.\text{size}() > 0$  do
5:   NODE  $u = S.\text{remove}()$  // Da specificare
6:   /* visita il nodo  $u$  */
7:   foreach  $v \in G.\text{adj}(u)$  do
8:     /* visita l'arco  $(u, v)$  */
9:     if  $v$  non è ancora stato marcato then
10:      /* marca il nodo  $v$  */
11:       $S.\text{insert}(v)$  // Da specificare
12:    end if
13:  end for
14: end while
```

Questo algoritmo è un algoritmo generico in quanto abbiamo un insieme generico S per memorizzare i nodi, e non sono state specificate le modalità di inserimento e rimozione di un nodo dal suddetto insieme. Vediamo quindi quando questo si comporta come un algoritmo **BFS** e quando come uno **DFS**:

BFS vs DFS

- **BFS:** Quando l'insieme S è una **coda**, la logica di inserimento e rimozione è di tipo **FIFO** (*First In First Out*), la visita diventa una **visita in ampiezza**, perché andiamo a prendere il nodo che si trova nella coda da più tempo, e visitiamo quindi la coda per livelli.
- **DFS:** Quando l'insieme S è una **pila**, la logica di inserimento e rimozione è di tipo **LIFO** (*Last In First Out*), la visita diventa una **visita in profondità**, perché andiamo a prendere l'ultimo nodo inserito e ci addentriamo quindi il più possibile nel grafo prima di tornare indietro.

8.5 BFS - Breadth-First Search

Abbiamo detto quindi che gli algoritmi **BFS** effettuano una **visita in ampiezza** del grafo, per memorizzare i nodi utilizzano una **coda** e di conseguenza la logica di inserimento e rimozione è di tipo **FIFO**. Vediamo adesso quali sono gli obiettivi di questi algoritmi:

Obiettivi

- Visitare i nodi a distanze crescenti → visitare tutti i nodi a distanza k dalla sorgente prima di quelli a distanza $k + 1$

- Calcolare il cammino più breve → Le distanze sono misurate come il numero di archi attraversati
- Generare un **albero breadth-first** → Generare un albero contenente tutti i nodi raggiungibili da r , tale per cui il cammino dalla radice r al nodo u nell'albero corrisponda al cammino più breve da r a u nel grafo.

8.5.1 Algoritmo BFS

```
bfs(GRAPH G, NODE r)
1: QUEUE Q = Queue() // Coda perché è un BFS
2: Q.enqueue(r)
3: boolean[] visited = new boolean[G.size()] // Vettore per marcare i nodi
4: foreach u ∈ G.V() - {r} do
5:   visited[u] = false
6: end for
7: visited[r] = true
8: while not Q.isEmpty() do
9:   NODE u = Q.dequeue()
10:  /* visita il nodo u */
11:  foreach v ∈ G.adj(u) do
12:    /* visita l'arco (u, v) */
13:    if not visited[v] then
14:      visited[v] = true
15:      Q.enqueue(v)
16:    end if
17:  end for
18: end while
```

Come abbiamo già ripetuto varie volte, e come si può vedere dallo pseudocodice, questo algoritmo utilizza una **coda** (FIFO) come struttura dati di supporto e questo garantisce la visita a livelli (i nodi più vicini alla radice vengono aggiunti alla coda e visitati prima di quelli più lontani). Il processo si divide in tre parti principali:

- Si marca la sorgente come visitata e la si inserisce in coda;
- Si estrae un nodo dalla coda e si esplorano tutti i suoi vicini non ancora marcati;
- Ogni vicino scoperto viene marcato e aggiunto in fondo alla coda per essere processato successivamente.

Questo metodo garantisce che un nodo venga scoperto solo attraverso il cammino minimo possibile (in termini di numero di archi). Se esistesse un cammino più breve per raggiungere v , la BFS lo avrebbe già incontrato in un livello precedente.

8.5.2 Numero di Erdős

La storia della matematica è piena di personaggi eccentrici; Paul Erdős è uno di questi. Fu un matematico estremamente prolifico, che arrivò a pubblicare circa 1500 articoli, con più di 500 collaboratori diversi.

Come tributo scherzoso alla sua estrema prolificità, è nato il concetto di “numero di Erdős”, un coefficiente assegnato ad ogni autore scientifico per misurare la sua "distanza" da Erdős (https://en.wikipedia.org/wiki/Erdos_number).

Ragionamento dietro il numero di Erdős

- Erdős ha $distance = 0$ o $erdos = 0^*$
- I co-autori dei suoi articoli hanno $distance = 1$
- Chi è co-autore di un co-autore di Erdős e non è direttamente coautore di Erdős, avrà $distance = 2$
- Chi è co-autore di qualcuno con $distance = k$ e non è direttamente coautore di Erdős, avrà $distance = k + 1$
- Tutti gli altri avranno $distance = \infty$

*Il professore chiama questa variabile `erdos` mentre il libro la chiama `distance`

Possiamo quindi riscrivere l'algoritmo sfruttando queste definizioni e otteniamo :

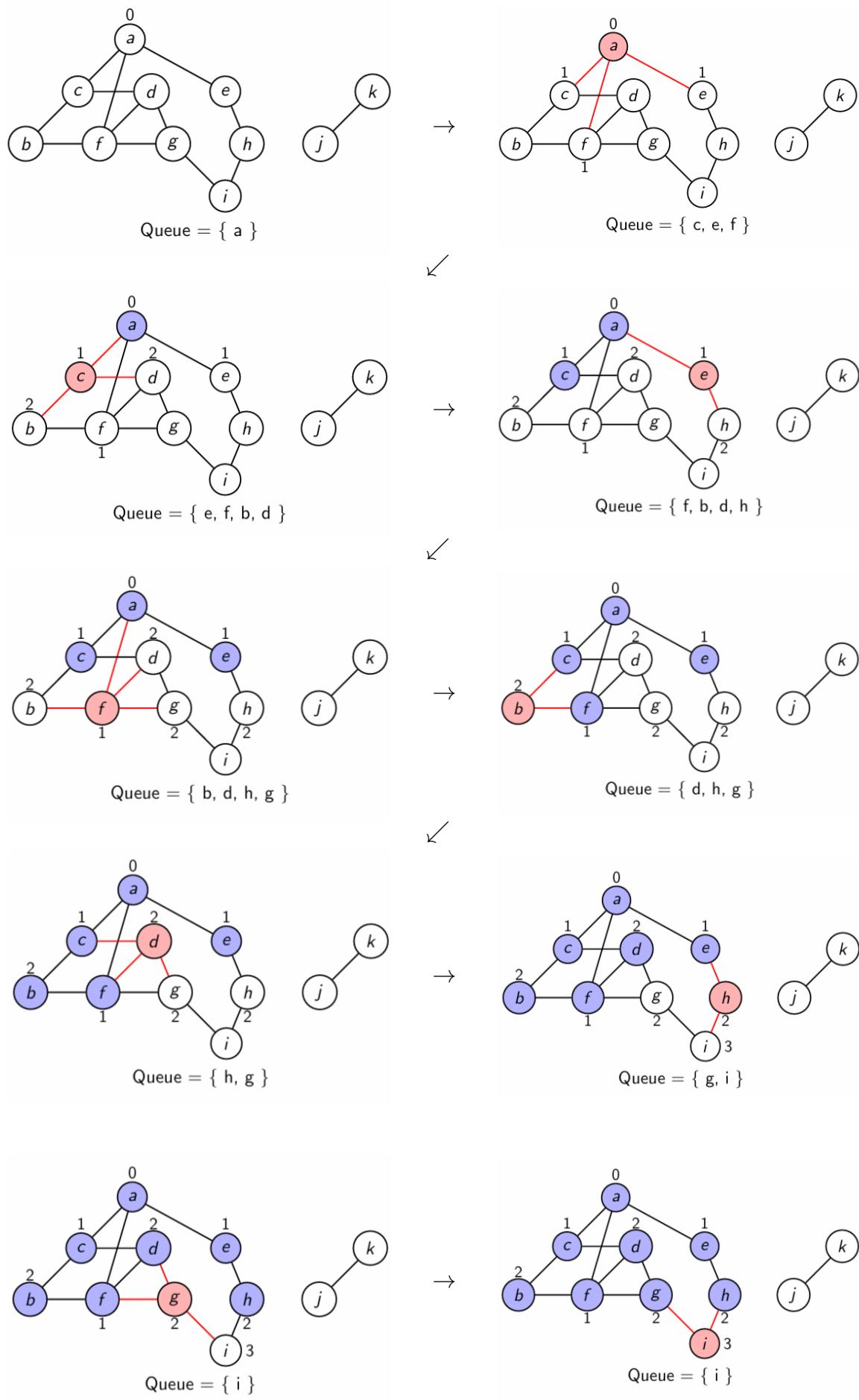
```
distance(GRAPH G, NODE r, int[] distance)
1: QUEUE Q = Queue()
2: Q.enqueue(r)
3: boolean[] visited = new boolean[G.size()] // Il prof la mette ma in questo caso è inutile
4: foreach u ∈ G.V() - {r} do
5:   distance[u] = ∞
6: end for
7: distance[r] = 0
8: while not Q.isEmpty() do
9:   NODE u = Q.dequeue()
10:  foreach v ∈ G.adj(u) do
11:    if distance[v] == ∞ then // Se il nodo v non è stato scoperto
12:      distance[v] = distance[u] + 1
13:      Q.enqueue(v)
14:    end if
15:  end for
16: end while
```

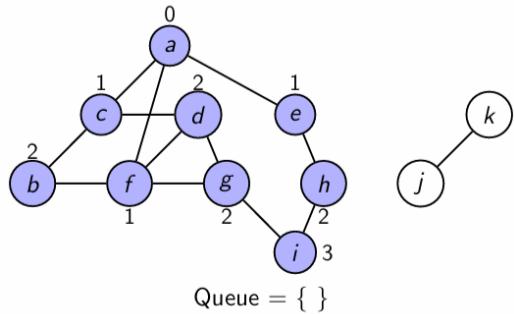
Come si può evincere dal commento, la riga 3, nonostante il professore la inserisca nello pseudocodice, è completamente inutile in questo contesto perché `visited` non viene mai usato in questo codice in quanto, per verificare se il nodo è già stato scoperto, si controlla la sua distanza.

Se un nodo ha distanza infinita, non è mai stato inserito in coda; non appena viene scoperto, riceve un valore numerico finito ($distance[v] = distance[u] + 1$) e non verrà mai più processato. Questo garantisce che ogni nodo venga aggiornato solo la prima volta che viene raggiunto, ovvero lungo il cammino minimo dalla sorgente.

Esempio

Vediamo ora un esempio visivo di come vengono calcolate le distanze con questo genere di algoritmi:





8.5.3 Albero BFS

Albero BFS

L'**Albero BFS** è un albero di copertura radicato in r che viene costruito dinamicamente durante la visita in ampiezza, utilizzando un vettore dei padri per tracciare il percorso più breve dalla sorgente a ogni nodo.

Infatti, oltre a visitare i nodi di un grafo, l'algoritmo BFS permette di definire una struttura ad albero, detta **Albero BFS**, che memorizza i cammini minimi (in termini di numero di archi) partendo da una radice r verso tutti i nodi raggiungibili.

Ricordiamo che, in un grafo non pesato, il **cammino minimo** coincide sempre con il cammino che attraversa il **minor numero di archi**.

Passo 1: Memorizzazione e Costruzione

```

distance(GRAPH G, NODE r, int[] distance, NODE[] parent )
1: QUEUE Q = Queue()
2: Q.enqueue(r)
3:
4: foreach u ∈ G.V() - {r} do
5:   distance[u] = ∞ // Si inizializzano tutte le distanze a ∞
6: end for
7:
8: distance[r] = 0 // Si mette la distanza della radice pari a 0
9: parent[r] = nil // Si mette il padre della radice come null
10:
11: while not Q.isEmpty() do
12:   NODE u = Q.dequeue()
13:   foreach v ∈ G.adj(u) do
14:     if distance[v] == ∞ then // Se il nodo v non è stato scoperto
15:       distance[v] = distance[u] + 1
16:       parent[v] = u
17:       Q.enqueue(v)
18:     end if
19:   end for
20: end while

```

Questo codice è come quello visto in precedenza, al quale però sono state aggiunte le parti scritte in rosso. Vediamo la logica che c'è dietro:

L'albero viene memorizzato in un vettore parent, dove ogni cella parent[v] contiene il nodo u che ha scoperto v . Durante l'esecuzione della BFS, quando esploriamo un nodo u e troviamo un vicino v non ancora visitato ($\text{distance}[v] = \infty$):

- Si aggiorna la distanza: $\text{distance}[v] = \text{distance}[u] + 1$;
- Si imposta il padre: $\text{parent}[v] = u$.

Passo 2: Ricostruzione del cammino

Per visualizzare il percorso minimo dalla radice r a un nodo destinazione s , si utilizza la procedura ricorsiva printPath. Questa risale il vettore dei padri fino alla radice e stampa i nodi durante il ritorno della ricorsione, garantendo l'ordine corretto.

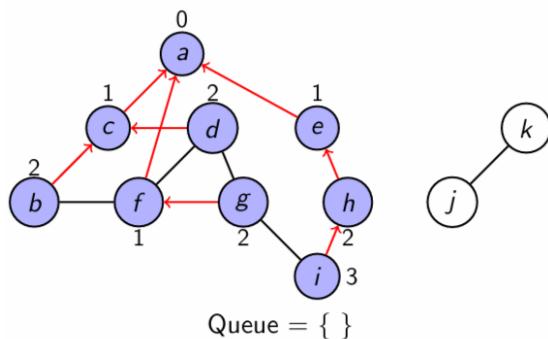
```
printPath(NODE r, NODE s, NODE[] parent)
1: if r == s then    // Caso base: siamo arrivati alla radice
2:   print s
3: else if parent[s] == nil then  // Il nodo s non è raggiungibile da r
4:   print "error"
5: else // Risalita ricorsiva verso la radice
6:   printPath(r, parent[s], parent)
7:   print s // Stampa il nodo durante il ritorno della ricorsione
8: end if
```

L'ordine delle istruzioni nel blocco else è fondamentale. Chiamando la funzione ricorsivamente su parent [s] prima di stampare s, ci assicuriamo che la stampa avvenga in ordine "top-down" (dalla radice alla destinazione), sfruttando lo stack delle chiamate della ricorsione per invertire l'ordine di risalita dei padri.

Appunto di Gemini sulla complessità spaziale

- Oltre alla memoria per il grafo, l'albero BFS richiede uno spazio aggiuntivo di $O(V)$ per memorizzare il vettore parent;
- La funzione printPath richiede uno spazio di stack ricorsivo proporzionale alla profondità dell'albero (nel caso peggiore $O(V)$).

Se ci riferiamo al grafo dell'esempio precedente, una volta terminata la visita avremo una figura come la seguente, nella quale gli archi rossi rappresentano il vettore dei padri:



8.5.4 Complessità BFS

(Ammetto di aver copiato e incollato da Gemini ciò che segue)

L'algoritmo BFS è estremamente efficiente, con una complessità temporale pari a $O(n + m)$. Questo risultato lo rende un algoritmo lineare rispetto alla dimensione della rappresentazione del grafo (nodi + archi).

Il motivo di questa efficienza risiede nella gestione oculata delle risorse durante la visita:

- **Sui nodi (n):** Grazie all'uso della coda e al controllo della distanza (o del colore), ogni nodo viene inserito e rimosso dalla coda una sola volta. Non ci sono quindi ricalcoli inutili sullo stesso vertice.
- **Sugli archi (m):** Ogni volta che un nodo viene estratto dalla coda, l'algoritmo esamina i suoi vicini. Questo significa che ogni arco viene analizzato una sola volta (nel caso di grafi orientati) o al massimo due (in quelli non orientati).

Tecnicamente, il numero di archi analizzati corrisponde alla somma dei gradi uscenti (d_{out}) di tutti i nodi.

$$m = \sum_{u \in V} d_{out}(u)$$

In sintesi, la BFS impiega un tempo proporzionale alla "grandezza" del grafo, rendendola una scelta ottimale per trovare cammini minimi in strutture dati di grandi dimensioni, a patto di utilizzare una lista di adiacenza per l'esplorazione dei vicini.

8.6 DFS - Depth-First Search

8.6.1 BFS vs DFS

8.6.2 Iterativa, stack esplicito, pre-order

8.6.3 DFS e componenti connesse

8.6.4 DFS e grafi non orientati aciclici

8.6.5 DFS e grafi orientati aciclici (DAG)

8.6.6 Schema

8.6.7 Classificazione degli archi

8.7 Ordinamento topologico

9 Analisi ammortizzata

Normalmente si vuole analizzare la complessità di strutture dati che evolvono nel tempo ed esse sono soggette a una serie di operazioni (operazioni dipendenti dalle strutture).

L'analisi ammortizzata è una tecnica di complessità che valuta il tempo per eseguire, **nel caso pessimo**, una sequenza di operazioni su una struttura dati.

In alcuni casi, le strutture dati possono avere operazioni più o meno costose, a seconda dello stato della struttura dati stessa e quindi alcune operazioni possono essere estremamente poco costose, mentre altre possono essere molto costose.

Importante differenza

- **Analisi caso medio:** Probabilistica, su singola operazione;
- **Analisi ammortizzata:** Deterministica, su operazioni multiple, caso pessimo.

Se calcolo la media, non la devo calcolare su una singola operazione presa in sé e per sé, ma su un insieme di operazioni che si considerano come una sequenza tipica di operazioni che vengono fatte su una struttura dati. A quel punto si esegue un'analisi di caso pessimo su operazioni multiple. Quest'analisi è spesso deterministica, quindi non probabilistica, perché si va a considerare la sequenza di operazioni che è la più costosa possibile tra tutte le sequenze di operazioni.

9.1 Metodi per eseguire l'analisi

Esistono 3 metodi per poter eseguire l'analisi ammortizzata:

- **Metodo dell'aggregazione:** Si calcola la complessità $T(n)$ per eseguire n operazioni in sequenza nel **caso pessimo**. Questa è una tecnica derivata dalla matematica;
- **Metodo degli accantonamenti:** Alle operazioni vengono assegnati dei **costi ammortizzati** (da cui deriva il termine "ammortizzata" per l'analisi) che possono essere maggiori/minori del loro costo effettivo. Questa è una tecnica che deriva dall'economia;
- **Metodo del potenziale:** Lo stato del sistema viene descritto come una **funzione di potenziale**. Questa tecnica deriva dalla fisica.

9.2 Esempio: Contatore binario

Si ha un contatore binario di k bit in un vettore A di booleani 0/1. Il bit meno significativo sta nella posizione $A[0]$ e il bit più significativo è alla posizione $A[k - 1]$.

Il valore del contatore è: $x = \sum_{i=0}^{k-1} A[i]2^i$. Si ha un'operazione detta *increment()* che incrementa il contatore di 1: L'operazione "increment" va a cercare il primo bit con valore 0, se non lo trova, ovvero se i bit hanno tutti valore 1, li trasforma in un valore 0 e passa al bit successivo, dal meno significativo verso il più significativo.

Quando arriva al primo bit vero, quindi quando la condizione del while non è rispettata, al bit i -esimo viene assegnato il valore 1.

```
increment(int[]A, int k)
```

```
1: int i=0
2: while i<k and A[i]==1 do
3:   A[i]=0
4:   i=i+1
5: end while
6: if i<k then
7:   A[i]=1
8: end if
```

x	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	1	1
4	0	0	0	0	0	1	0	0
5	0	0	0	0	0	1	0	1
6	0	0	0	0	0	1	1	0
7	0	0	0	0	0	1	1	1
8	0	0	0	0	1	0	0	0
9	0	0	0	0	1	0	0	1
10	0	0	0	0	1	0	1	0
11	0	0	0	0	1	0	1	1
12	0	0	0	0	1	1	0	0
13	0	0	0	0	1	1	0	1
14	0	0	0	0	1	1	1	0
15	0	0	0	0	1	1	1	1
16	0	0	0	1	0	0	0	0

9.3 Metodo dell'aggregazione

Si calcola la complessità $T(n)$ per eseguire n operazioni in sequenza nel caso pessimo. Poi si calcola il costo ammortizzato $T(n)/n$ come media su n operazioni.

Allora: come sequenza si devono considerare tutte le possibili sequenze di operazioni e nel caso pessimo si considera la peggiore in assoluto.

In questo caso (esempio del contatore), c'è un solo tipo di operazione, ovvero l'incremento. n operazioni sono quindi n operazioni di incremento e l'**aggregazione** è la sommatoria delle varie complessità individuali.

Nell'esempio il caso pessimo è $O(k)$ perché l'indice superiore è k , stabilito dal **while**.

Nel caso pessimo n operazioni costano: $k = \lceil \log(n + 1) \rceil$, cioè il numero di bit necessari per rappresentare n , una chiamata *increment()* costa $O(k)$, n operazioni costano $T(n) = O(nk)$ e il costo di un'operazione è $T(n)/n = O(k) = O(\log n)$.

Per fare n operazioni, il costo ammortizzato calcolato in questo modo delle varie operazioni è pari a $\log n$.

Si può osservare che il tempo necessario ad eseguire l'intera sequenza è proporzionale al numero di bit che vengono modificati.

x	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	#bit
0	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	2
3	0	0	0	0	0	0	1	1	1
4	0	0	0	0	0	1	0	0	3
5	0	0	0	0	0	1	0	1	1
6	0	0	0	0	0	1	1	0	2
7	0	0	0	0	0	1	1	1	1
8	0	0	0	0	1	0	0	0	4
9	0	0	0	0	1	0	0	1	1
10	0	0	0	0	1	0	1	0	2
11	0	0	0	0	1	0	1	1	1
12	0	0	0	0	1	1	0	0	3
13	0	0	0	0	1	1	0	1	1
14	0	0	0	0	1	1	1	0	2
15	0	0	0	0	1	1	1	1	1
16	0	0	0	1	0	0	0	0	5
	#bit	0	0	0	1	2	4	8	16

Quindi, quello che si va a fare è provare a calcolare il numero di bit che verranno cambiati, che è la dimensione del ciclo "**while**", si prova a sommarli tutti e si guarda cosa salta fuori utilizzando i metodi dell'analisi matematica.

Si può notare che:

- $A[0]$ viene modificato ad ogni incremento: $(\lfloor n/2^0 \rfloor)$
 - $A[1]$ viene modificato ogni 2 incrementi: $(\lfloor n/2^1 \rfloor)$
 - $A[2]$ viene modificato ogni 4 incrementi: $(\lfloor n/2^2 \rfloor)$
- Arrivando quindi a:
- $A[i]$ viene modificato ogni 4 incrementi: $(\lfloor n/2^i \rfloor)$

Analisi ammortizzata

- **Costo totale:** $T(n) = \sum_{i=0}^{k-1} \lfloor \frac{n}{2^i} \rfloor \leq n \sum_{i=0}^{k-1} \frac{1}{2^i} \leq n \sum_{i=0}^{+\infty} (\frac{1}{2})^i = 2n$. Il 2 rappresenta la sommatoria. Quindi "n" operazioni su un contatore binario costano $2 * n$
- **Costo ammortizzato:** Se divido $2n$ per n si ottiene il valore 2 come limite superiore e quindi: $T(n)/n \leq 2n/n = 2 = O(1)$

9.4 Metodo degli accantonamenti

Quello che si va a fare con questo metodo è assegnare un costo ammortizzato potenzialmente distinto ad ognuna delle operazioni possibili.

In questo caso (sempre il contatore binario) esiste un'unica operazione, quindi si assegna un costo all'unica operazione possibile.

Il costo ammortizzato può essere diverso dal costo effettivo, ovvero alcune operazione, quelle che coinvolgono tanti bit, sono più costose e quindi il costo ammortizzato di queste operazioni viene "pagato" dal **credito** che la struttura dati ha accumulato eseguendo operazioni costose. Le operazioni meno costose vengono caricate di un costo aggiuntivo detto **credito** e il costo ammortizzato per le operazioni meno costose è dato da: **costo ammortizzato = costo effettivo + credito prodotto**.

Il costo ammortizzato per le operazioni più costose è dato dal costo effettivo meno il credito consumato: **costo ammortizzato = costo effettivo – credito consumato**.

Nel determinare qual è il costo che si vuole dare alle operazioni, si ha un obiettivo, ovvero dimostrare che la somma dei costi ammortizzati " a_i " è un limite superiore alla somma dei costi

effettivi " c_i ":

$$\sum_{i=1}^n c_i \leq \sum_{i=1}^n a_i$$

Il segno " \leq " indica il limite superiore definito da a_i .

Si vuole dimostrare che questa relazione è sempre valida per come si sono creati i valori ammortizzati e anche che il valore ottenuto da " $\sum_{i=1}^n a_i$ " è poco costoso. Si deve anche assumere che esista sempre un credito dopo una serie di operazione, quindi l'operazione t -esima su n possibili operazioni. Il credito è espresso da questa formula:

$$\sum_{i=1}^t a_i - \sum_{i=1}^t c_i \geq 0$$

Ovvero la sommatoria dei costi ammortizzati meno la sommatoria dei costi reali.

Tale differenza deve sempre essere maggiore o uguale a zero.

Il costo effettivo dell'operazione di `increment()` è d , dove d è il numero di bit che cambiano valore, però, per ogni riga, d continua a variare.

L'idea è che $\sum_{i=1}^t c_i$ sarà pari a d per ognuna delle operazioni. Invece si deve trovare un modo più semplice per fare questa operazione, quindi ciò che si fa, è cercare di dire che questo costo $\sum_{i=1}^t c_i$ è minore o uguale di $\sum_{i=1}^t a_i$ e si decide che il costo ammortizzato è uguale a 2, quindi:

$$\sum_{i=1}^t c_i \leq \sum_{i=1}^t a_i = \sum_{i=1}^t 2 = 2t$$

Ma come si è stabilito?

Si sa che all'inizio si hanno tutti i bit a 0. Tutte le volte che si cambia un bit e lo si trasforma a 1 si ha un costo effettivo, ma, ad un certo punto nel futuro, quel bit a 1 tornerà a 0, quindi si paga subito quest'altro costo, pari a 1, per il futuro cambio del bit da 1 a 0.

Il costo ammortizzato dell'operazione `increment()` sarà quindi $2 = 1 + 1$:

- 1 per il cambio di un bit da 0 a 1 (**costo effettivo**)
- 1 per il futuro cambio dello stesso bit da 1 a 0.

Lo si paga subito, anche se questo cambio potrebbe non essere fatto in futuro Perché quel bit potrebbe rimanere a 1 sino al termine delle n operazioni, però questo è un limite superiore.

La somma di t operazioni sarà limitata superiormente da $2 * t$, quindi, su n operazioni, si avrà un costo, che è $O(n)$ (che sarebbe $2n$), e dividendo per n si ottiene $O(1)$.

Ricapitolando, il costo deriva dai cambi di bit, perché i bit che non vengono toccati, perché sono troppo avanti nella somma, non interessano. Quando si cambia un bit da 0 a 1, in futuro lo si potrebbe dover ri-cambiare a 0, quando arriverà il suo turno nei vari `increment` e lo si paga subito.

La somma dei costi ammortizzati è sempre più alta della somma dei costi effettivi e deve essere così secondo la relazione.

x	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	#bit	c_i	a_i	$\sum c_i$	$\sum a_i$
0	0	0	0	0	0	0	0	0	1	1	2	1	2
1	0	0	0	0	0	0	0	1	2	2	2	3	4
2	0	0	0	0	0	0	1	0	2	2	2	4	6
3	0	0	0	0	0	0	1	1	1	2	2	4	8
4	0	0	0	0	0	1	0	0	3	2	2	7	10
5	0	0	0	0	0	1	0	1	1	2	2	8	12
6	0	0	0	0	0	1	1	0	2	2	2	10	14
7	0	0	0	0	0	1	1	1	1	2	2	11	16
8	0	0	0	0	1	0	0	0	4	2	2	15	16
9	0	0	0	0	1	0	0	1	1	2	2	1	1
10	0	0	0	0	1	0	1	0	2	2	2	1	2
11	0	0	0	0	1	0	1	1	1	2	2	1	1
12	0	0	0	0	1	1	0	0	3	2	2	7	8
13	0	0	0	0	1	1	0	1	1	2	2	8	10
14	0	0	0	0	1	1	1	0	2	2	2	10	12
15	0	0	0	0	1	1	1	1	1	2	2	11	14
16	0	0	0	1	0	0	0	0	5	2	2	15	16
	#bit	0	0	0	1	2	4	8	16				

I costi ammortizzati per ogni riga sono uguali a 2 e la sommatoria dei costi ammortizzati $\sum a_i$ è maggiore della sommatoria dei costi effettivi $\sum c_i$.

Alla fine si otterrà:

x	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	c_i	a_i	$\sum c_i$	$\sum a_i$
1	0	0	0	0	0	0	0	1	1	2	1	2
2	0	0	0	0	0	0	1	0	2	2	2	3
3	0	0	0	0	0	0	1	1	1	2	2	4
4	0	0	0	0	0	1	0	0	3	2	2	7
5	0	0	0	0	0	1	0	1	1	2	2	8
6	0	0	0	0	0	1	1	0	2	2	2	10
7	0	0	0	0	0	1	1	1	1	2	2	11
8	0	0	0	0	1	0	0	0	4	2	2	15
9	0	0	0	0	1	0	0	1	1	2	2	16
10	0	0	0	0	1	0	1	0	2	2	2	18
11	0	0	0	0	1	0	1	1	1	2	2	19
12	0	0	0	0	1	1	0	0	3	2	2	22
13	0	0	0	0	1	1	0	1	1	2	2	23
14	0	0	0	0	1	1	1	0	2	2	2	25
15	0	0	0	0	1	1	1	1	1	2	2	26
16	0	0	0	1	0	0	0	0	5	2	2	31

9.5 Metodo del potenziale

Si ha una struttura dati con uno stato, una serie di bit a 0 e una serie di bit a 1. Si crea una **funzione di potenziale** ϕ che associa ad uno stato D della struttura dati la "quantità di lavoro" $\phi(D)$ che è stato contabilizzato nell'analisi ammortizzata, ma non ancora eseguito. Significa che, in altre parole, $\phi(D)$ rappresenta la quantità di energia potenziale immagazzinata in quello stato e il costo ammortizzato dipende dal costo effettivo a cui viene associata una variazione di potenziale, che può essere negativa, se sta consumando energia immagazzinata in quello stato, o positiva, se sta invece accumulando energia immagazzinata in quello stato. Quello che si va a fare è calcolare il costo ammortizzato per definizione come:

Costo ammortizzato

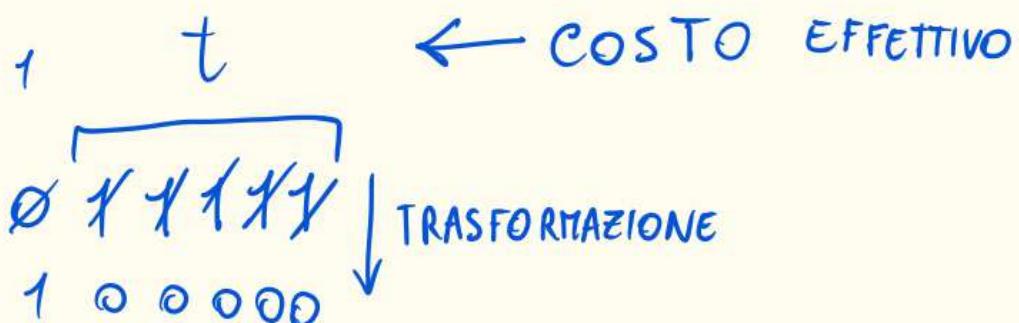
$$\text{Costo ammortizzato} = \text{Costo effettivo} + \text{Variazione di potenziale}$$

$$a_i = c_i + \phi(D_i) - \phi(D_{i-1})$$

Dove D_i è lo stato associato alla i-esima operazione.

In questo caso, utilizzando i concetti precedenti, si può scegliere come funzione di potenziale ϕ il numero di bit 1 presenti nel contatore, perché quelli sono i bit che in qualche modo si sono trasformati da 0 a 1 e che avranno un costo per essere ritrasformati al valore 0.

Tutte le volte che si chiama un increment, si indica con t il numero di bit a 1 incontrati a partire dal meno significativo, prima di incontrare uno 0, e si considera come costo effettivo $1 + t$:



La variazione di potenziale è $1 - t$ perché si sono tolti tutti i bit a 1 facenti parte di t e si è trasformato lo 0 a 1.

$$\begin{aligned}
A &= \sum_{i=1}^n a_i \\
&= \sum_{i=1}^n (c_i + \phi(D_i) - \phi(D_{i-1})) \\
&= \sum_{i=1}^n c_i + \sum_{i=1}^n (\phi(D_i) - \phi(D_{i-1})) \\
&= C + \phi(D_1) - \phi(D_0) + \phi(D_2) - \phi(D_1) + \cdots + \phi(D_n) - \phi(D_{n-1}) \\
&= C + \phi(D_n) - \phi(D_0)
\end{aligned}$$

Il costo ammortizzato si calcola come una sequenza di n operazione, quindi si vanno a sommare i costi ammortizzati, poi si sostituisce con la formula del costo ammortizzato, si separano le sommatorie del costo effettivo, che è quella che si conosce, è il costo effettivo delle operazioni, e la sommatoria di $(\phi(D_i) - \phi(D_{i-1}))$. Alla fine si ottiene $C + \phi(D_n) - \phi(D_0)$. Il costo è dato di n operazioni è dato dalla differenza di potenziale a partire dallo stato iniziale. Se la variazione di potenziale $\phi(D_n) - \phi(D_0)$ è non negativa, il costo ammortizzato A è un limite superiore al costo reale.

La formula del costo ammortizzato per il contatore binario nel metodo del potenziale è: $1 + t + 1 - t = 2$, dove $1 + t$ è il costo effettivo e $1 - t$ è la variazione del potenziale.

9.6 Analisi ammortizzata e strutture dati: Array VS Liste

Le sequenze lineari possono essere realizzate con:

- **Array:** Sequenze ad accesso diretto in cui, dato j , si accede direttamente all'elemento a_j . Il costo di ciascun accesso è costante ed è tutto memorizzato in locazioni di memoria consecutive.
- **Liste:** Sequenze ad accesso sequenziale in cui, dato j , si accede ad a_0, a_1, \dots, a_j attraversando la sequenza a partire dall'inizio (o dalla fine). Si ha un costo $O(p)$ per raggiungere a_{j+p} partendo a_j . Gli elementi della lista sono memorizzati in locazioni di memoria non necessariamente contigue.

Vantaggio delle liste

Nessun limite alla dimensione massima della sequenza

9.6.1 Array dinamici

Alcuni linguaggi di programmazione prevedono array che possono essere ridimensionati, detti dinamici.

Ci sono delle operazioni che sono necessarie per ridimensionare un array A di dimensione n :

- Creare un nuovo array B ;
- Copiare gli elementi di A in B ;
- Deallocare A dalla memoria;
- Ridenominare B come A .

Il costo del ridimensionamento è $O(n)$. Il ridimensionamento è necessario quando si vuole aggiungere l'elemento $n + 1 - \text{esimo}$ all'array di dimensione n e ogni aggiunta di un elemento $n + 1 - \text{esimo}$ ad una lista di dimensione n ha costo $O(1)$. In termini di memoria allocata si ha un vantaggio, ma in termini computazionali è un costo oneroso.

9.6.2 Array dinamici - Espansione

Il problema di un array dinamico è che non è noto a priori quanti elementi entreranno nella sequenza e quindi la capacità iniziale disponibile potrebbe non rivelarsi sufficiente. La soluzione è di allocare un vettore di capacità maggiore, si copia il contenuto del vecchio vettore nel nuovo e si rilascia il vecchio vettore. Il caso peggiore è quello in cui si devono fare continui inserimenti nel vettore.

L'espansione può avvenire in più modi, qui vengono considerati il **raddoppiamento** e l'**incremento costante**.

9.6.3 Analisi ammortizzata - Raddoppiamento del vettore

Il ridimensionamento comporta un costo elevato, perché implica la costruzione di un nuovo array con dimensione aumentata e la successiva copia degli elementi. Per evitare di eseguire questo procedimento ogni volta che si aggiunge un nuovo elemento, gli array dinamici si ridimensionano raddoppiando le dimensioni.

3

push(3)

3	5
---	---

push(5)

3	5	2	
---	---	---	--

push(2)

3	5	2	7
---	---	---	---

push(7)

3	5	2	7	9		
---	---	---	---	---	--	--

push(9)

Il costo effettivo di un'operazione di inserimento c_i è i se $i - 1$ è una potenza di 2 mentre è 1 in tutti gli altri casi.

$$c_i = \begin{cases} i & \exists k \in Z_0^+ : i = 2^k + 1 \\ 1 & \text{altrimenti} \end{cases}$$

<i>n</i>	costo
1	1
2	$1 + 2^0 = 2$
3	$1 + 2^1 = 3$
4	1
5	$1 + 2^2 = 5$
6	1
7	1
8	1
9	$1 + 2^3 = 9$
10	1
11	1
12	1
13	1
14	1
15	1
16	1
17	$1 + 2^4 = 17$

Assunzioni:

- Dimensione iniziale dell'array: 1
- Costo di scrittura di un elemento: 1

Il costo effettivo di n operazioni di inserimento in questo caso risulta essere:

$$\begin{aligned}
 T(n) &= \sum_{i=1}^n c_i \\
 &= n + \sum_{i=1}^{\lfloor \log n \rfloor} 2^j \\
 &= n + 2^{\lfloor \log n \rfloor + 1} - 1 \\
 &\leq n + 2^{\log n + 1} - 1 \\
 &= n + 2n - 1 = O(n)
 \end{aligned}$$

E si ottiene il costo ammortizzato di un'operazione di inserimento in questo modo:

$$T(n)/n = \frac{O(n)}{n} = O(1)$$

9.6.4 Analisi ammortizzata - Incremento del vettore

Costo effettivo di un'operazione di inserimento:

$$c_i = \begin{cases} i & (i \bmod d) = 1 \\ 1 & \text{altrimenti} \end{cases}$$

n	costo
1	1
2	1
3	1
4	1
5	$1 + d = 5$
6	1
7	1
8	1
9	$1 + 2d = 9$
10	1
11	1
12	1
13	$1 + 3d = 13$
14	1
15	1
16	1
17	$1 + 4d = 17$

Assunzioni:

- Dimensione iniziale dell'array: d
- Incremento: d
- Costo di scrittura di un elemento: 1

Nell'esempio $d=4$

Il costo effettivo di n operazioni di inserimento in questo caso risulta essere:

$$\begin{aligned}
 T(n) &= \sum_{i=1}^n c_i \\
 &= n + \sum_{j=1}^{\lfloor n/d \rfloor} d * j \\
 &= n + d \sum_{j=1}^{\lfloor n/d \rfloor} * j \\
 &= n + d * \frac{(\lfloor n/d \rfloor + 1)\lfloor n/d \rfloor}{2} \\
 &\leq n + \frac{(n/d + 1)n}{2} = O(n^2)
 \end{aligned}$$

E si ottiene il costo ammortizzato di un'operazione di inserimento in questo modo:

$$T(n)/n = \frac{O(n^2)}{n} = O(n)$$

10 Programmazione Dinamica

La programmazione dinamica è una tecnica che si ispira al *divide-et-impera* e infatti anche qui un problema più grande viene preso e diviso in più sottoproblemi, ma si aggiunge una caratteristica. Normalmente nel *divide-et-impera* i problemi sono ben separati, mentre nella programmazione dinamica i sottoproblemi si ripetono. Per questo motivo si utilizza una tabella delle soluzioni, cioè si memorizzano le soluzioni dei problemi che sono già stati risolti. Per fare una cosa del genere si deve capire come definire il sottoproblema e che informazioni memorizzare, dato che non si ha una memoria infinita e non si può memorizzare tutto, la memorizzazione deve essere efficiente. Nel caso in cui un sottoproblema debba nuovamente essere affrontato, si ottiene la sua soluzione dalla tabella, la quale è facilmente indirizzabile (il costo della procedura di lookup è $O(1)$).

10.1 Idea generale

Se non si sa con esattezza quale problema risolvere, si può provare a risolverli tutti e conservare i risultati ottenuti per poterli usare successivamente.

Rispetto al *divide-et-impera*, la programmazione dinamica presenta 3 differenze:

- È iterativa e non ricorsiva;
- Affronta i sottoproblemi dal "basso verso l'alto" (bottom-up) e non "dall'alto verso il basso" (top-down);
- Memorizza i risultati dei sottoproblemi in una tabella DP.

Procedendo dal basso verso l'alto, la programmazione dinamica risolve un sottoproblema comune una sola volta, mentre il *divide-et-impera* lo risolverebbe più volte.

10.1.1 Fasi della risoluzione del problema

Fasi principali

- Caratterizzare la struttura di una soluzione ottima;
- Definire iterativamente il valore di una soluzione ottima;
- Calcolare il **valore** di una soluzione ottima "bottom-up";
- Ricostruzione di una soluzione ottima.

10.2 Esempio - Coefficiente binomiale

$C(n, k) = \frac{n!}{k!(n-k)!}$ rappresenta il numero di modi di scegliere k oggetti da un insieme di n oggetti, con $0 \leq k \leq n$, ed è definibile ricorsivamente come segue:

$$C(n, k) = \begin{cases} 1 & \text{se } k = 0 \vee n = k \\ C(n-1, k-1) + C(n-1, k) & \text{altrimenti} \end{cases}$$

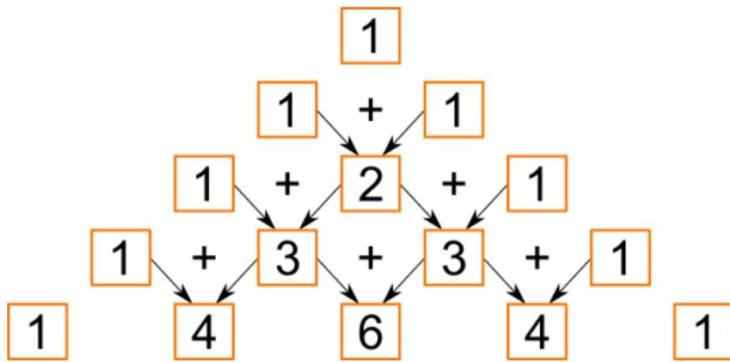
Purtroppo, la complessità dell'algoritmo *divide-et-impera* cresce come il numero di chiamate ricorsive, che è proprio uguale a $C(n, k)$. Ciò è dovuto all'elevato numero di sottoproblemi identici che vengono risolti più volte. Ad esempio $C(n-2, k-1)$ è richiamato due volte: per calcolare $C(n-1, k)$ e $C(n-1, k-1)$.

```

int C(int n, int k)
1: if n=k or k=0 then
2:   return 1
3: end if
4: return C(n-1,k-1)+C(n-1,k)

```

Un algoritmo di programmazione dinamica invece risolve i sottoproblemi per valori crescenti degli argomenti seguendo lo schema "**Triangolo di Tartaglia**", che memorizza i risultati una volta per tutte in una matrice DP di dimensione $n \times n$ in $\theta(n^2)$.



Tartaglia(int n, int[][] DP)

```

1: for i do=0 to n do
2:   DP[i, 0] = 1
3:   DP[i, i] = 1
4: end for
5: for i do=2 to n do
6:   for j do=1 to i-1 do
7:     DP[i, j]=DP[i-1, j-1]+DP[i-1, j]
8:   end for
9: end for

```

In questo, il valore di $DP(m, k)$, per ogni coppia di interi "m, k" tali che $0 \leq k \leq m \leq n$ può essere successivamente letto direttamente in tempo $O(1)$.

10.3 Quando usare la programmazione dinamica

Perché la programmazione dinamica sia applicabile, occorre che:

1. sia possibile combinare le soluzioni dei sottoproblemi per trovare la soluzione del problema più grande;
2. le decisioni prese per risolvere in modo ottimo un sottoproblema rimangano valide quando il problema diviene un pezzo di un problema;
3. una tecnica *divide-et-impera* sia inutilizzabile dal punto di vista computazionale.

Le prime due proprietà definiscono una **sottostruttura ottima**. Ma non sono sufficienti. Infatti, affinché un algoritmo basato sulla programmazione dinamica abbia complessità

polinomiale, occorre anche che:

4. ci sia un numero polinomiale di problemi da risolvere;
5. per evitare di risolvere più di una volta lo stesso problema, si utilizzi una tabella in cui si memorizzano le soluzioni di tutti i sottoproblemi, senza preoccuparsi se la soluzione di un particolare sottoproblema verrà poi utilizzata oppure no;
6. il tempo per combinare le soluzioni dei sottoproblemi e trovare la soluzione del problema più grande sia polinomiale.

10.4 String Match Approssimato

Date:

- una stringa $P = p_1 \dots p_m$, detta **pattern**,
- una stringa $T = t_1 \dots t_n$, detta **testo**, con $m \leq n$,

un'occorrenza $k - approssimata$ di P in T , con $0 \leq k \leq m$, è una copia della stringa di P in T in cui sono ammessi k "errori" tra caratteri di P e caratteri di T , del seguente tipo:

1. corrispondenti caratteri P, T sono diversi (**sostituzione**)
2. un carattere in P non è incluso in T (**inserimento**)
3. un carattere in T non è incluso in P (**cancellazione**)

Problema - Approximated string matching

trovare un'occorrenza $k - approssimata$ di P in T con k minimo ($0 \leq k \leq m$).

10.4.1 Esempio

$T = \text{questo}\text{\color{red}{e}}\text{unoscempio}$ (dove $n = 17$)

$P = \text{une}\text{\color{red}{s}}\text{empio}$ (dove $m = 9$).

Un'occorrenza $2 - approssimata$ di P parte dalla posizione 8 di T : $\text{questo}\text{\color{red}{e}}\text{unoscempio}$. Infatti, la prima e di P corrisponde ad una o di T (errore di tipo (1), **sostituzione**), mentre la c di T è un carattere non presente in P (errore di tipo (3), **cancellazione**).

Per applicare la programmazione dinamica, si deve trovare una formulazione ricorsiva che abbia la proprietà di sottostruttura ottima.

Definizione

Sia $DP[0 \dots m][0 \dots n]$ una tabella di programmazione dinamica tale che $DP[i][j]$ sia il minimo valore k per cui esiste un'occorrenza $k - approssimata$ di $P(i)$ in $T(j)$ che termina nella posizione j

10.4.2 4 possibilità:

- Se $P[i] = T[j]$, quel carattere è uguale, ci si sta chiedendo qual è la più corta occorrenza k -approssimata che termina in quel carattere. Ciò che si va a fare è dire: se il carattere è uguale non c'è errore, quindi non si somma un qualcosa qua e si va a vedere il carattere precedente. Il punto è che ci si sta limitando a considerare le sequenze che terminano in quella posizione. $DP[i - 1][j - 1] + 0$.

- Se $P[i] \neq T[j]$, allora si può fare una **sostituzione**, cioè si cambia un carattere nell'altro e quindi si somma 1: $DP[i - 1][j - 1] + 1$.

Queste due possibilità si escludono, perché non possono valere contemporaneamente, mentre le prossime due possono coesistere.

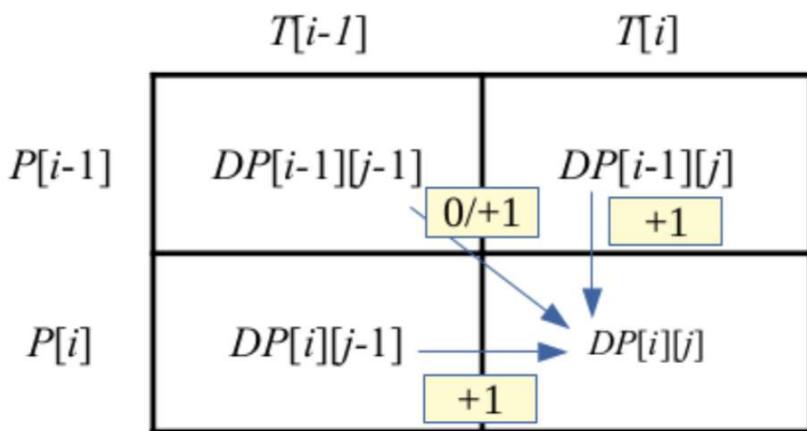
- $DP[i - 1][j] + 1$
- $DP[i][j - 1] + 1$

Si somma 1 perché si contano gli errori, non si sta massimizzando l'uguaglianza, ma si stanno minimizzando le differenze.

$$DP[i][j] = \begin{cases} 0 & i = 0 \\ i & j = 0 \\ \min\{DP[i - 1][j - 1] + \delta, DP[i - 1][j] + 1, DP[i][j - 1] + 1\} & \text{altrimenti} \end{cases}$$

$\delta = \text{iif}(P[i] = T[j], 0, 1)$

Quello che succede è che la formula avrà la forma evidenziata da $*$, cioè il minimo di $*_1$ dove δ è uguale a 0 o 1 a seconda che i caratteri siano uguali oppure diversi e poi $*_2$ e $*_3$, questo è il caso ricorsivo.



Nei primi due casi si nota la simmetria tra il testo e il pattern perché la i è per il pattern e la j per il testo. Se si cerca un pattern vuoto in un testo, qualunque sia la lunghezza del testo, il numero di cambiamenti che si devono fare è 0, perché in qualunque punto termina una stringa vuota di un testo. Invece se si cerca una parola in un testo vuoto, bisogna inserire tutti i caratteri di quella parola, quindi l'intera lunghezza del pattern, perciò si mette la lettera i .

T	A	B	A	B	A	
P	0	0	0	0	0	
B	1	1	0	1	0	1
A	2	1	1	0	1	0
B	3	2	1	1	0	1

Le frecce indicano come prendere il "punteggio" da aggiungere.

10.4.3 Esempio 1

T	(A)	B	A	B	A	
P	0	0	0	0	0	
B	1	1	0	1	0	1
A	2	1	1	0	1	0
B	3	2	1	1	0	1

SE GUARDO QUESTO NUMERO MI DICE C'È UN'OCCORRENZA 2-APPROXIMATA DI BAB IN A

T	A	B	A	B	A	
P	0	0	0	0	0	0
B	1	1	0	1	0	1
A	2	1	1	0	1	0
B	3	2	1	1	0	1

QUI MI DICE
CHE C'È
UN' OCCORRENZA
1-APPROSSIMATA
DI BAB IN
AB

10.4.4 Esempio 2

T	A	B	A	B	A	
P	•	0	0	0	0	0
B	1	•	1	0	1	0
A	2	1	1	0	1	0
B	3	2	1	1	0	1

- CASELLA DA CALCOLARE
 - CASELLA A SX
 - CASELLA SOPRA
 - CASELLA IN ALTO A SX
- TUTTE LE VOLTE CHE VADO A CALCOLARE UNA CASELLA, PRENDO IL MINIMO TRA ● + 1, ● + 1 E ● + 0 (PERCHÉ I CARATTERI SONO uguali), QUINDI PRENDO IL VALORE 0 DA ●.

T	A	B	A	B	A	
P	0	0	0	•	0	0
B	1	1	0	1	•	0
A	2	1	1	0	1	0
B	3	2	1	1	0	1

PRENDO IL MINIMO TRA ● + 1, ● + 1 E ● + 0 (PERCHÉ I CARATTERI SONO uguali), QUINDI PRENDO IL VALORE 0 DA ●.

Se si va a vedere la semantica, questo mi sta dicendo che esiste un'occorrenza 0 – approssimata di B in ABAB che termina nell'ultimo carattere. Se è 0 – approssimata significa che è perfetta, perché in effetti la B del pattern ci sta perfettamente.

10.4.5 Esempio 3

T	A	B	A	B	A	
P	0	0	0	0	0	0
B	1	1	0	1	0	1
A	2	1	1	0	1	0
B	3	2	1	1	0	1

Qui mi dice che esiste un'occorrenza 1-approssimata di B in ABA, che termina nell'ultimo carattere A (di T). C'è una B, ma non è nell'ultimo carattere, quindi posso prendere ● e sommargli 1

10.4.6 Esempio 4

T	A	B	A	B	A	
P	0	0	0	0	0	0
B	1	1	0	1	0	1
A	2	1	1	0	1	0
B	3	2	1	1	0	1

Queste sono sequenze k -approximate dell'intero pattern BAB in qualche parte del testo. Si deve restituire un'occorrenza della stringa BAB con k minore possibile, che qua si trova dove è lo 0.

Non è detto quindi che la soluzione finale si trovi nella casella in basso a destra. È possibile invece che la soluzione debba essere cercata (se esiste la mia occorrenza k -approximata e dov'è con k minore possibile).

Distanza di Levenshtein o distanza di editing

Date due stringhe, si vuole conoscere il numero minimo di operazioni necessario per trasformare una nell'altra (o viceversa). Es. la distanza di editing tra google e yahoo è 6 (cancellare gle e inserire ya, sostituire g con h). La relazione di ricorrenza è la stessa, ma $D[i, 0] = i$ e $D[0, j] = j$, perché $T(j)$ ha j caratteri in più di $P(0)$. Al termine, la distanza di editing sarà in $D[m, n]$.

10.4.7 Algoritmo

```
int stringMatching(ITEM[] P, ITEM[] T, int m, int n)
    int[][] DP=new int[0 ... m][0 ... n]
    for j do=0 to n do
        DP[0][j]=0
    end for
    for i do=0 to m do
        DP[i][0]=i
    end for
    for i do=1 to m do
        for j do=1 to n do
            DP[i][j]=min(DP[i-1][j-1]+iif(P[i]==T[j],0,1), DP[i-1][j]+1, DP[i][j-1]+1)
        end for
    end for
    int pos=0
    for j do=1 to n do
        if DP[m][j]<DP[m][pos] then
            pos=j
        end if
    end for
    return pos
```

Il primo `for` inizializza la prima riga con tutti gli zeri, il secondo `for` inizializza invece la prima colonna. Il resto è l'algoritmo che riempie la tabella e l'ultima parte serve a trovare il minimo.

10.4.8 Reality check

Approximate String Matching

Approximate String Matching è un esempio di string metric, si utilizza per misurare la distanza (come inverso della similarità) tra due stringhe. Può essere usato in:

- Fraud detection;
- Fingerprint analysis;
- Plagiarism detection;
- DNA-RNA analysis.

10.5 Insieme indipendente di intervalli pestai

Input

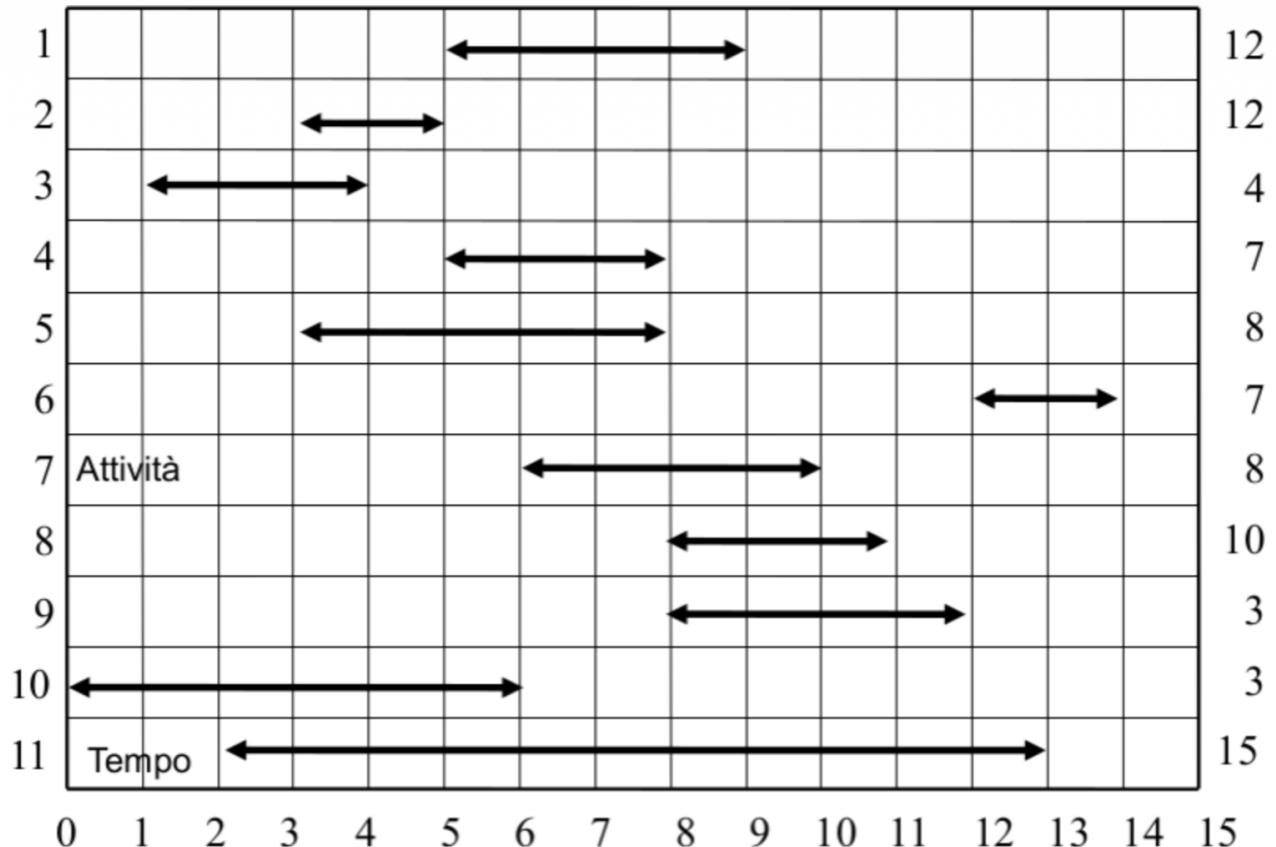
Siano dati n intervalli distinti $[a_1, b_1[, \dots, [a_n, b_n[$ della retta reale, aperti a destra, dove all'intervallo i è associato un profitto w_i , $1 \leq i \leq n$

Si hanno n intervalli della retta reale aperti a destra, in modo che più intervalli possano essere compatibili.

Due intervalli i e j si dicono disgiunti se: $b_j \leq a_i$ oppure $b_i \leq a_j$. Questa versione del problema associa ad ogni intervallo un peso. Il problema consiste nel trovare un sottoinsieme

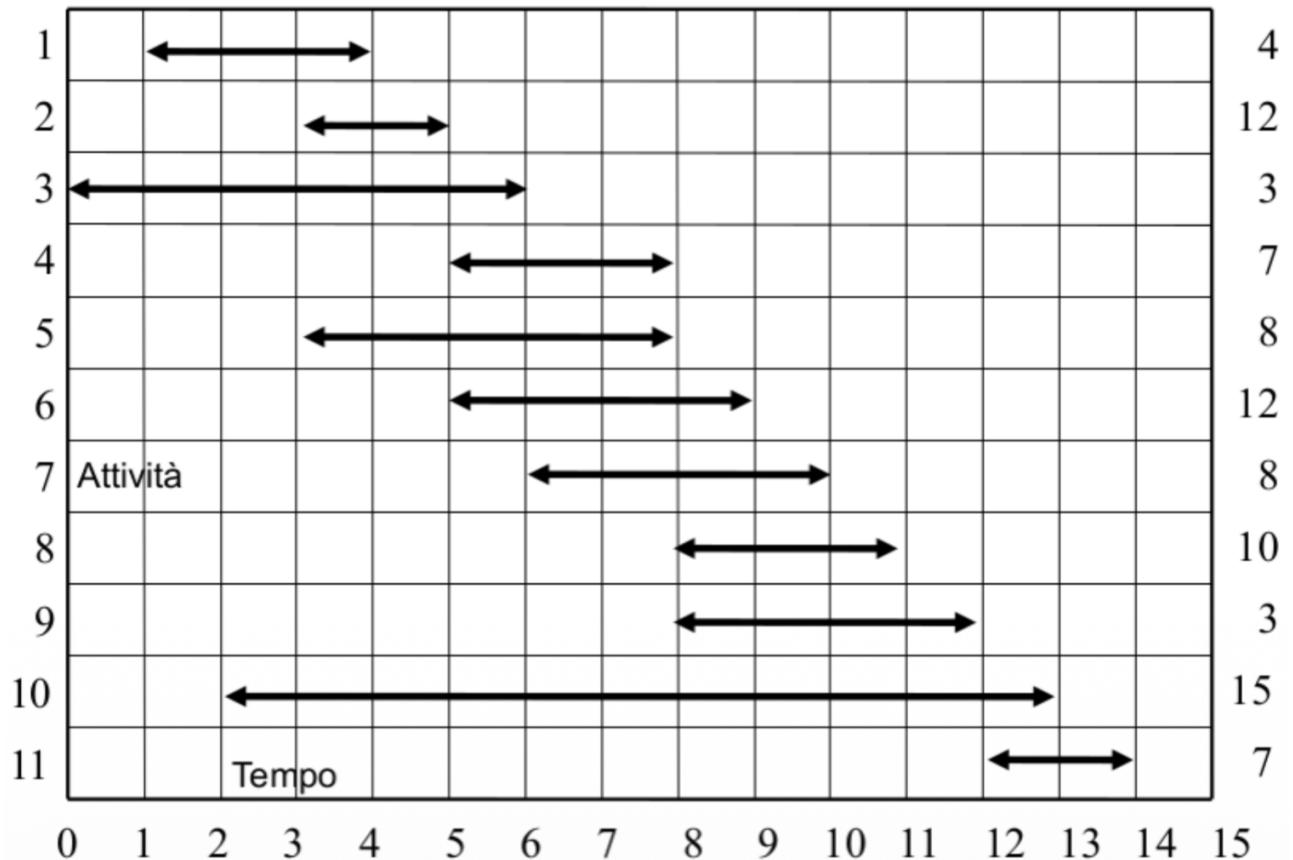
degli intervalli possibili, tale per cui essi siano disgiunti e la somma dei loro pesi sia il massimo possibile.

10.5.1 Esempio: insieme indipendente di peso massimo



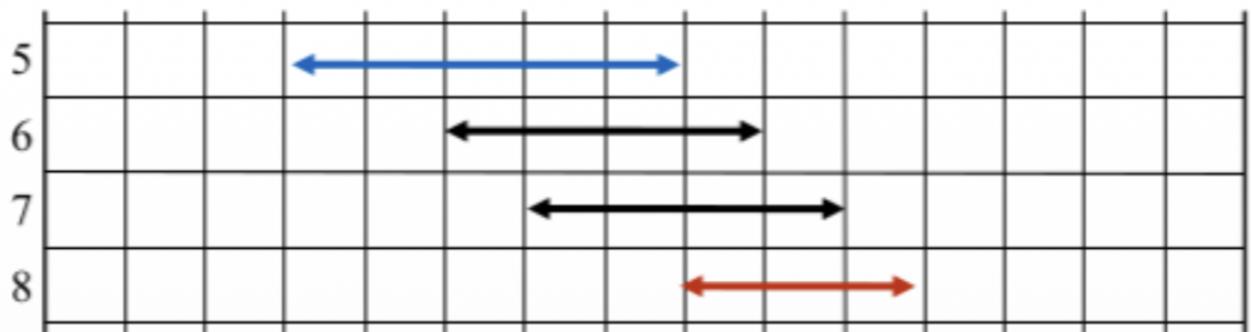
10.5.2 Pre-elaborazione

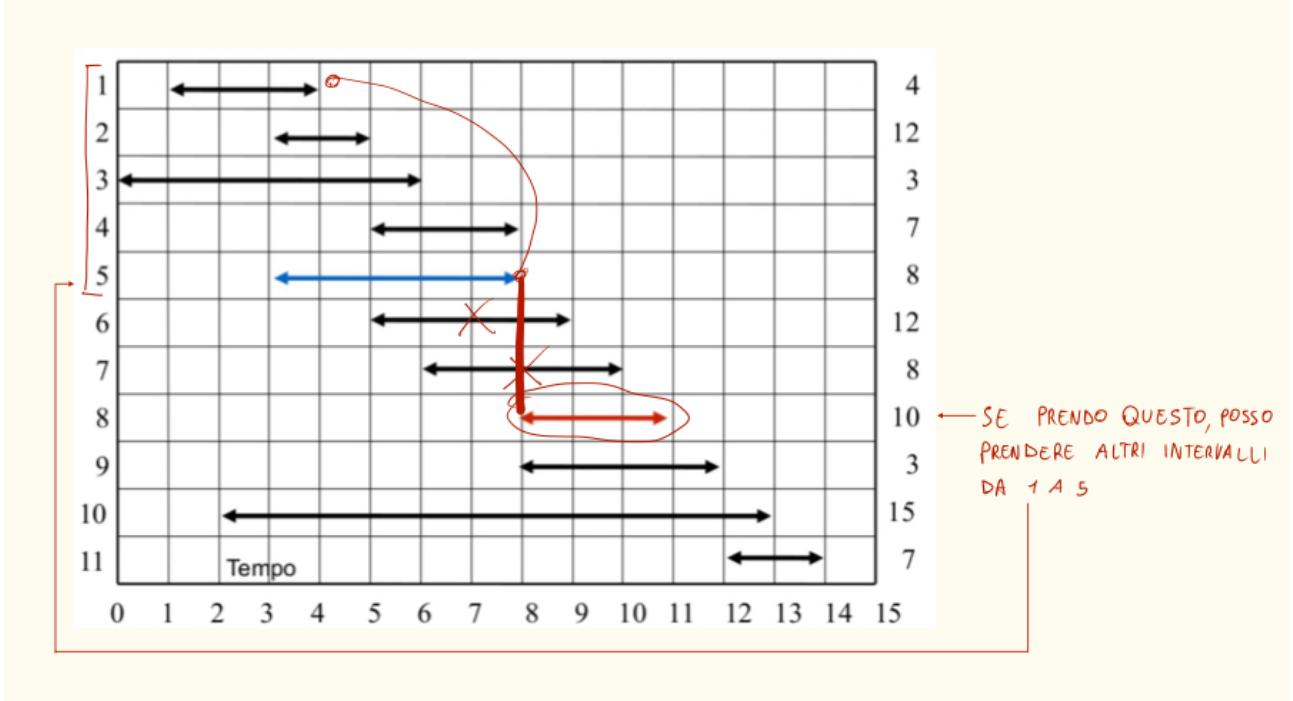
In alcuni problemi di programmazione dinamica, prima di operare sui dati, conviene ordinarli, ovvero fare una **pre-elaborazione**. In questo caso vengono ordinati gli intervalli per estremi finali non decrescenti e si ha un costo di ordinamento pari a $O(n \log n)$, perché si sta riordinando per tempo di fine.



Si fa anche una seconda pre-elaborazione che consiste nel definire il **predecessore** di i (quindi di ogni intervallo) come $p_i = j$, con $j < i$.

- j è il massimo indice tale che $[a_j, b_j]$ **non interseca** $[a_i, b_i]$;
- se non esiste $j \rightarrow p_i = 0$. Ovvero se non esiste l'indice j non c'è nessun predecessore.





10.5.3 Individuazione sottostruttura ottima (forma ricorsiva)

Siano $P[i]$ il sottoproblema dato dai primi i intervalli ed $S[i]$ una soluzione ottima di peso $D[i]$. Si hanno due opzioni:

- Se l'intervallo i -esimo non fa parte della soluzione ottima ($i \notin S[i]$) $\rightarrow D[i] = D[i - 1]$ (con $D[0] = 0$);
 - Se l'intervallo i -esimo fa parte della soluzione ottima ($i \in S[i]$) $\rightarrow D[i] = w_i + D[p_i]$
- N.B. Si dimostrano per assurdo.

Il problema ha sottostruttura ottima

La relazione di programmazione dinamica che si ottiene è: $D[i] = \max(D[i - 1], w_i + D[p_i])$

Una volta definita la sottostruttura ottima, con una relazione ricorsiva di programmazione dinamica, la tabella di programmazione dinamica viene riempita attraverso un algoritmo iterativo. Si noti che $D[n]$ è il costo (guadagno) della soluzione ottima del problema iniziale. A partire da essa, si calcolano a ritroso gli intervalli che hanno portato a tale soluzione.

10.5.4 Algoritmo iterativo

```
SET maxInterval(int[] a, int[] b, int[] w)
1: <ordinare gli intervalli per estremi finali crescenti>;<calcolare  $p_j$ >
2: int[] D=new int[0...n]; D[0]=0
3: for i=1 to n do
4:   D[i]=max(D[i-1], w[i]+D[p_i])
5: end for
6: i=n
7: SET S=Set()
8: while i>0 do
9:   if D[i-1]>w[i]+D[p_i] then
10:    i=i-1
11:   else
12:     S.insert(i); i=p_i
13:   end if
14: end while
15: return S
```

10.5.5 Costo computazionale

- Ordinamento intervalli: $O(n \log n)$;
- Calcolo predecessori: $O(n \log n)$;
- Riempimento tabella D: $O(n)$;
- Ricostruzione soluzione: $O(n)$;
- Costo totale: $O(n \log n)$.

Gli intervalli vanno ordinati per tempo non decrescente di fine ed eventuali intervalli con lo stesso tempo di fine possono essere ordinati in qualunque modo. Questo perché $D[i]$ rappresenta il massimo profitto ottenibile con i primi i intervalli. È quindi possibile escludere l'intervallo i – *esimo* se sceglierne uno precedente j (ma con lo stesso tempo di fine $b[i] = b[j]$ ha un valore $D[j]$ più alto).

10.6 Zaino (Knapsack)

Si ha un insieme di oggetti, caratterizzato da un peso e un profitto e si ha uno zaino con un limite di capacità. Si deve individuare un sottoinsieme di oggetti il cui peso totale sia inferiore alla capacità dello zaino e il valore totale degli oggetti sia massimale, ovvero il più alto o uguale al valore di qualunque altro sottoinsieme di oggetti.

Input

- Vettore w (weight), dove $w[i]$ è il peso dell'oggetto $i - esimo$;
- Vettore p (profit), dove $p[i]$ è il profitto dell'oggetto $i - esimo$;
- La capacità C dello zaino.

Si vuole un sottoinsieme $S \subseteq [1, \dots, n]$ tale per cui il volume totale espresso come sommatoria dei pesi degli oggetti deve essere minore o uguale alla capacità: $w(S) = \sum_{i \in S} w[i] \leq C$ e il profitto totale deve essere massimizzato: $\text{argmax}_{sp}(S) = \sum_{i \in S} p[i]$

Quale è il valore massimo per questo zaino, con $C = 12$?

Item id	1	2	3	4
Weight	12	4	6	2
Profit	26	8	12	4

Quale è il valore massimo per quest'altro zaino, con $C = 12$?

Item id	1	2	3	4
Weight	12	4	6	2
Profit	29	9	13	5

10.6.1 Definizione matematica della soluzione

Le tabelle di programmazione dinamica utilizzate fino ad ora erano tutte sotto forma di vettori. Siccome in questo problema si ha una realtà a 2 parametri (il numero di oggetti e la capacità), in questo particolare caso si deve usare una tabella di programmazione dinamica che è una matrice, un array bidimensionale.

Si definisce quindi un sottoproblema, che si va ad identificare con due indici " i " e " c ", $DP[i][c]$, come il massimo profitto che può essere ottenuto con i primi $i \leq n$ oggetti contenuti in uno zaino con capacità $c \leq C$.

Il problema originale è: si hanno " n " oggetti e una capacità " C ", $DP[n][C]$, ma si vuole sapere cosa succede se si ha una capacità leggermente più piccola, se si hanno meno oggetti, quindi si considera tutto lo spazio di tutti i possibili sottoproblemi presenti.

10.6.2 Parte ricorsiva

Quello che si va a fare è scegliere se prendere o non prendere un oggetto. Se un oggetto **non viene preso** è *come se non fosse mai esistito* e quindi: $D[i][c] = DP[i - 1][c]$, la capacità rimane

sempre la stessa e non c'è un'aggiunta del profitto.

Se invece l'oggetto viene preso: $DP[i][c] = DP[i - 1][c - w[i]] + p[i]$, si sottrae il peso alla capacità e si aggiunge il profitto relativo. L'indice "i" è sempre "i - 1" perché poi l'oggetto non si può più prendere, dato che ogni oggetto può essere preso solo una volta.

10.6.3 Casi base

Quali sono i casi base?

- Quando non si hanno più oggetti da scegliere, qual è il profitto massimo che si può ottenere? $\rightarrow 0$;
- Qual è il profitto massimo se non si ha più capacità? $\rightarrow 0$;
- Cosa succede se la capacità è negativa?

La capacità negativa può apparire dal fatto che " $c - w[i]$ ", se non si fa attenzione $w[i]$ potrebbe essere maggiore della capacità residua e quindi " $c - w[i]$ " risulterebbe minore di 0.

$$D[i, c] = \begin{cases} 0 & \text{se } i = 0 \vee c = 0 \\ -\infty & \text{se } c < 0 \\ \max(DP[i - 1][c - w[i]] + p[i], DP[i - 1][c]) & \text{altrimenti} \end{cases}$$

Se il " $c - w[i]$ " porta sullo spazio negativo allora si mette $-\infty$ e si farà un massimo tra $-\infty$ e un valore maggiore o uguale a 0, a questo punto vincerà il valore, altrimenti si possono fare i casi particolari:

$$\begin{cases} DP[i - 1][c] & \text{se } w[i] > c \\ \max(DP[i - 1][c - w[i]] + p[i], DP[i - 1][c]) & \text{se } w[i] \leq c \end{cases}$$

Una volta che si ha la formula la si traduce in codice (in maniera iterativa).

10.6.4 Zaino algoritmo

```

int knapsack(int[] w, int[] p, int n, int C)
1: DP = new int[0 … n][0 … C]
2: for i=0 to n do
3:   DP[i][0] = 0
4: end for
5: for c=0 to C do
6:   DP[0][c] = 0
7: end for
8: for i=1 to n do
9:   for c=1 to C do
10:    if w[i] ≤ c then
11:      DP[i][c] = max(DP[i-1][c-w[i]] + p[i], DP[i-1][c])
12:    else
13:      DP[i][c] = DP[i-1][c]
14:    end if
15:   end for
16: end for
17: return DP[n][C]

```

Nella riga 1, la tabella ha dimensione " $(n + 1) * (C + 1)$ " perché si ha bisogno di memorizzare anche i casi base. La complessità di questa operazione, ovvero della creazione della tabella, è: $\theta(1)$.

Nella righe 2-7, siccome il caso base è 0 se $i = 0$ o $c = 0$ si inizializza a 0. La complessità di questi cicli for è rispettivamente: $\theta(n)$ e $\theta(C)$.

Poi si hanno due cicli for per riempire il resto della tabella, nell'if, se $w[i] \leq c$ si fa il *max* fra i due casi, quindi quando si prende un oggetto ($DP[i - 1][c - w[i]] + p[i]$) e quando non lo si prende ($DP[i - 1][c]$), altrimenti si passa al caso in cui $w[i] > c$. La complessità dei due for è $\theta(n * C)$, mentre quella dell'if ...else è $\theta(1)$.

Alla fine si ritorna la tabella $DP[n][C]$.

10.6.5 Esempio

- $w = [4, 2, 3, 4]$
- $p = [10, 7, 8, 6]$
- $C = 9$

	c									
i	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10	10	10
2	0	0	7	7	10	10	17	17	17	17
3	0	0	7	8	10	15	17	18	18	25
4	0	0	7	8	10	15	17	18	18	25

10.6.6 Complessità computazionale

C può essere arbitrariamente grande, deve quindi essere un dato appropriato. C non è la dimensione del problema, è infatti parte dell'input.

La dimensione del problema è data da n pesi più n profitti più 1 capacità, " $n + n + 1$ ". La dimensione del problema risulta come " $2n + 1$ ", ma la C si usa nel calcolo della complessità: $T(n) = O(nC)$.

Questo è un algoritmo **pseudo-polinomiale**, perché sono necessari $k = \lceil \log C \rceil$ bit per rappresentare C e quindi la complessità risulta essere $T(n) = O(n2^k)$.

10.7 Memoization - Introduzione

La **Memoization** è una variante della programmazione dinamica, essa dice che "**Non è sempre necessario calcolare le soluzioni di tutti i sottoproblemi**".

10.7.1 Zaino con Memoization

Andando a riguardare la definizione ricorsiva del problema originale e il suo costo ($O(nC)$), non è detto che sia necessario risolvere tutti i sottoproblemi (dipende dai loro valori di volumi e capacità).

La **Memoization** permette un approccio alternativo: fonde l'approccio di memorizzazione della programmazione dinamica, ma è ricorsivo e risolve il problema "dall'alto verso il basso", top-down, si parte dal caso di interesse e su calcolano i sottoproblemi se non sono già stati risolti.

Gli approcci visti fin qui sono iterativi, dal basso verso l'alto e risolvono obbligatoriamente tutti i problemi.

Idea Base

È una procedura che utilizza la tabella come una cache per memorizzare le soluzioni dei problemi che viene inizializzata con un **valore speciale** che indica i sottoproblemi non ancora risolti (ad esempio un valore negativo).

Una procedura controlla se il sottoproblema è stato risolto:

- Se **si**: riutilizza il risultato;
- Se **no**: lo risolve (con chiamata ricorsiva sui sottoproblemi), memorizza il risultato in tabella e lo restituisce al chiamante.

10.7.2 Algoritmo zaino con Memoization

```
integer zaino(integer[] w, integer[] p, integer i, integer c, integer[][] DP)
1: if i==0 or c==0 then
2:   return 0
3: end if
4: if c<0 then
5:   return -∞
6: end if
7: if DP[i, c]=false then
8:   D[i,c] ← max(zaino(w, p, i – 1, c, DP), zaino(w, p, i – 1, c – w[i], DP)+p[i])
9: end if
10: return DP[i, c]
```

10.7.3 Complessità

Complessità della versione puramente ricorsiva dello Zaino Ricorsivo (senza controllo sulle operazioni già eseguite).

$$T(n) = \begin{cases} 1 & n \leq 1 \\ 2T(n-1) + 1 & n > 1 \end{cases}$$
$$T(n) = O(2^n)$$

10.7.4 Zaino Memoization: utilizzo tabella come cache

Non tutti gli elementi della matrice sono necessari alla risoluzione del problema.

- $w = [4, 2, 3, 4]$
- $p = [10, 7, 8, 6]$
- $C = 9$

	c									
i	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10	10	10
2	0	0	7	7	10	10	17	17	17	17
3	0	0	7	8	10	15	17	18	18	25
4	0	0	7	8	10	15	17	18	18	25

Come si ricostruisce la soluzione? Ossia, come si identificano gli oggetti che hanno portato al valore massimale $DP[n][C]$?

10.7.5 Dizionario vs Tabella

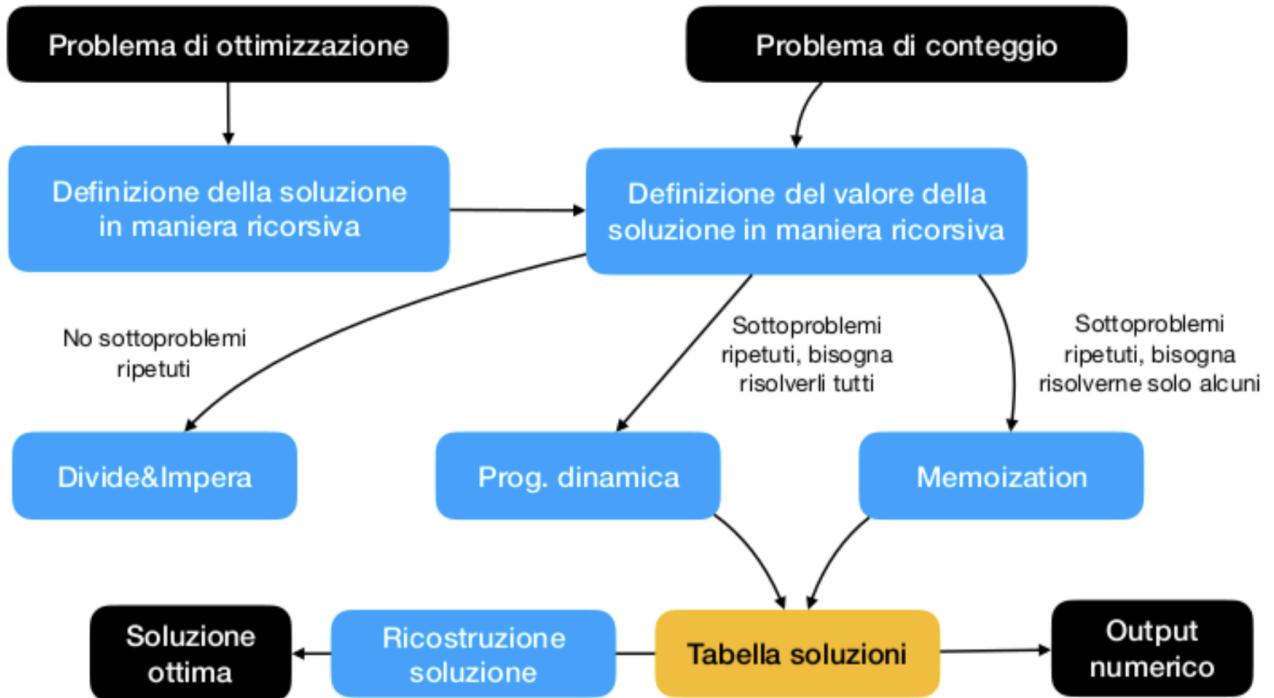
Inizializzazione tabella

- Il costo di inizializzazione è pari a $O(nC)$;
- Applicata in questo modo, non c'è alcun vantaggio nell'utilizzare la tecnica di memoization;
- Permette però di tradurre in fretta le espressioni ricorsive.

Utilizzo di un dizionario (hash table)

- Invece di utilizzare una tabella, si utilizza un dizionario;
- Non è necessario fare inizializzazione;
- Il costo di esecuzione è pari a $O(\min(2^n, nC))$

10.7.6 Approccio generale



10.7.7 Riassunto: programmazione dinamica/memoization

Fasi:

- Caratterizzare la **struttura** di una soluzione ottima;
- Dimostrare che la soluzione gode di una **sottostruttura ottima**;
- Definire ricorsivamente il **valore** di una soluzione ottima;
- Calcolare il **valore** di una soluzione "bottom-up" (prog. dinamica)/ "top-down" (memoization);
- **Ricostruzione** di una soluzione ottima.

11 Hashing

L'**hashing** è una **tecnica alternativa** agli alberi binari di ricerca (capitolo 4.4) o agli array associativi, utilizzata per realizzare i dizionari.

A differenza dell'implementazione tramite alberi, nella quale si riusciva a mantenere la stessa complessità nei casi di `insert()`, `lookup()` e `remove()`, nel caso ideale, la cosa migliore sarebbe quella di **mantenere una complessità costante** per tutti i tipi di operazioni, che sia inoltre **inferiore** a quella delle strutture ad albero.

Questa implementazione ideale prende il nome di **tabelle di hash**.

	Array non ordinato	Array ordinato	Lista	Alberi RB	Impl. ideale Hash table
<code>insert()</code>	$O(1), O(n)$	$O(n)$	$O(1), O(n)$	$O(\log n)$	$O(1)$
<code>lookup()</code>	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$	$O(1)$
<code>remove()</code>	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$
<code>foreach</code>	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$\Theta(n)O(m)$

Per comprendere il funzionamento delle tabelle hash dobbiamo prendere in considerazione il concetto di **insieme universo U** , ovvero un insieme di tutte le possibili chiavi, la cui grandezza dell'insieme varia arbitrariamente in base ai dati che si vogliono contenere.

Idea di base

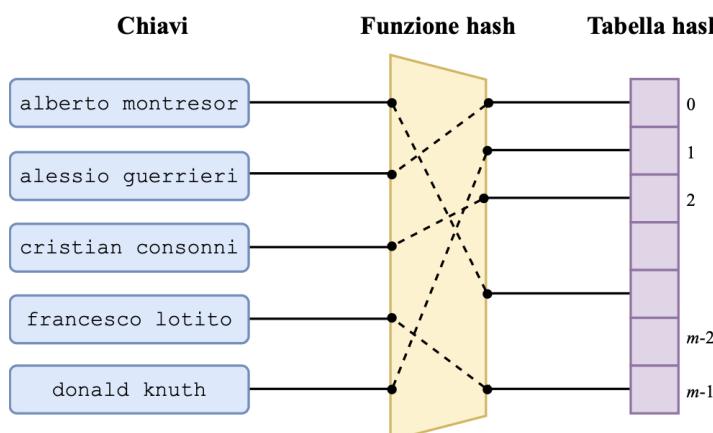
Quello che si vuole fare è memorizzare tutti i dati dell'insieme U in un vettore di dimensione (m) finita $T[0...m - 1]$, ed avere un meccanismo per cui, data una chiave, trovare rapidamente la posizione in cui è memorizzata.

Le chiavi possono essere delle stringhe, degli oggetti o dei numeri, e il compito delle tabelle hash è quello di trasformarle in un indice all'interno di esse. Per fare ciò viengono utilizzate le **funzioni hash**.

Cos'è una funzione hash?

Una **funzione hash** è una funzione H che mappa ciascuna chiave k appartenente all'insieme universo U ($k \in U$) nell'indice $H(k)$ di un vettore A , destinato a contenere la coppia (k, v) . Viene definita come $H : U \rightarrow \{0, 1, \dots, m - 1\}$.

Figure 78: Esempio di funzione hash



Come si può vedere dall'immagine, viene presa una chiave (in questo caso una stringa) che verrà poi trasformata in qualche modo, tramite la funzione hash (che può essere anche una black box), in un indice.

A questo punto sorge un **problema**: l'insieme delle chiavi è potenzialmente infinito, ma non si vuole che sia lo stesso anche per la tabella hash, dal momento in cui si potrebbe non disporre

dello spazio in memoria necessario per qualunque coppia chiave-valore. Dunque, quello che succede è che avvengono delle **collisioni**.

11.1 Tabelle ad accesso diretto

Prima di affrontare il problema delle collisioni, è utile analizzare un **caso particolare** in cui esse possono essere **completamente evitate**: le **tabelle ad accesso diretto**.

Questo approccio è applicabile solo quando l'insieme universo U delle chiavi è limitato e di dimensione ridotta, tale da poter essere rappresentato direttamente in memoria senza sprechi eccessivi. Utilizzando un approccio del genere è possibile utilizzare un vettore A della stessa dimensione dell'insieme U , quindi $m = |U|$, nel quale ogni chiave k che appartiene all'insieme U viene memorizzata direttamente nella posizione $A[k]$.

La funzione hash utilizzata è quindi la **funzione identità** $h(k) = k$, in questo caso una **funzione hash perfetta**, come quella illustrata in figura 78, che garantisce l'esecuzione delle operazioni di `insert()`, `lookup()` e `remove()` in tempo $O(1)$ nel caso peggiore.

In questo modo ogni chiave verrebbe memorizzata in una posizione distinta della tabella.

11.1.1 Funzioni hash perfette

Funzioni hash perfette

Una funzione hash h si dice **perfetta** se è **iniettiva**, ovvero se non da origine a collisioni:

$$\forall k_1, k_2 \in U : k_1 \neq k_2 \Rightarrow H(k_1) \neq H(k_2)$$

Tuttavia, questo metodo presenta un **limite fondamentale**:

oltre al fatto che se l'insieme universo U è molto grande, l'approccio non è praticabile, nel caso in cui il numero di chiavi memorizzate nel dizionario sia molto inferiore al massimo disponibile, quindi $m = |U|$, si incorrerebbe in uno **spreco di memoria** poiché molte posizioni del vettore verrebbero lasciate inutilizzate.

Per questo motivo, le **tabelle ad accesso diretto** sono utilizzabili solo in **contesti specifici** e rappresentano principalmente un modello teorico di riferimento per le tabelle hash.

Per evitare inutili sprechi di memoria, la dimensione m del vettore non deve essere determinata sulla base dell'intero universo U , ma piuttosto in funzione del **numero di chiavi attese**, ovvero la **quantità k di elementi** che si prevede saranno **effettivamente presenti nel dizionario** in un determinato momento.

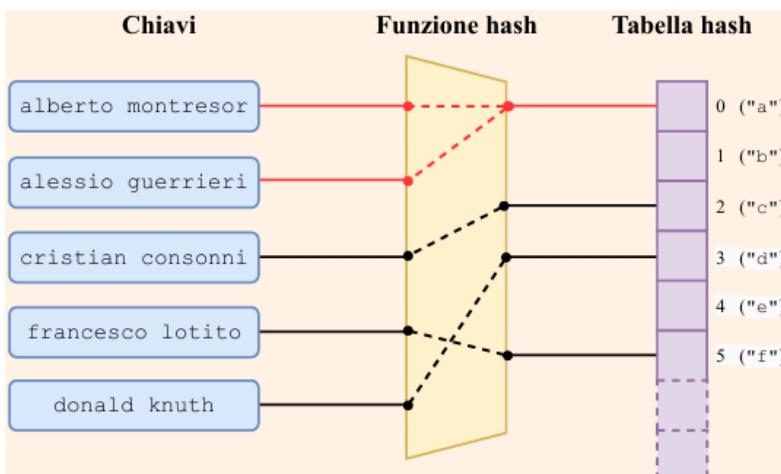
Quando si verifica una collisione?

Si verifica una **collisione** quando due chiavi distinte $k_1 \neq k_2$ vengono mappate dalla funzione hash sulla stessa posizione del vettore, ovvero quando $H(k_1) = H(k_2)$.

Le collisioni mettono quindi in luce due **aspetti fondamentali** nella progettazione di una tabella hash:

- La **scelta della funzione hash** è cruciale, infatti una funzione mal progettata può distribuire le chiavi in modo sbilanciato, causando una concentrazione eccessiva in alcune posizioni del vettore, lasciando quasi del tutto inutilizzate altre.

- È essenziale progettare dei **meccanismi** efficienti per la **gestione delle collisioni**, poiché



anche scegliendo una buona funzione di hash, quando il numero di chiavi possibili supera il numero di posizioni disponibili nella tabella, le collisioni sono inevitabili. Questi meccanismi permettono quindi di **garantire il corretto funzionamento** delle operazioni di inserimento, ricerca e cancellazione.

11.2 Minimizzare le collisioni

Se **non è possibile evitare** le collisioni, si cerca di **minimizzarne il numero**.

La **qualità di una funzione hash** dipende in modo cruciale dalla **sua capacità** di distribuire le chiavi dell'universo U negli indici $[0 \dots m - 1]$ della tabella hash **in modo uniforme**. Una distribuzione sbilanciata porta a un aumento del numero di collisioni in alcune celle, con conseguente peggioramento delle prestazioni del dizionario.

Uno dei criteri più diffusi per valutare la qualità di una funzione hash è l'**uniformità semplice**:

- Sia $P(k)$ la probabilità che una qualunque chiave k sia inserita nel dizionario;
- Sia $Q(i)$ la probabilità che una qualunque chiave finisca nella cella i -esima della tabella.

Uniformità semplice

Una funzione hash h gode di **uniformità semplice** se, per ogni posizione i della tabella, la probabilità che una chiave k venga mappata in i è uguale a $1/m$, quindi $Q(i) = 1/m$.

Quindi, l'evento che interessa è: "la chiave k_n scelta finisce nella cella i ", e ciò accade se la funzione hash, per qualche motivo, mappa una determinata chiave per quella cella: $h(k_1) = i$, oppure $h(k_2) = i$, e così via... definendo eventi tra loro mutualmente esclusivi.

$$Q(i) = \sum_{k \in U : h(k)=i} P(k)$$

Quando un evento può accadere tramite alternative mutualmente esclusive, la probabilità totale è la somma delle probabilità alternative.

Proprio per questo motivo ottengo $Q(i)$ sommando le probabilità che le chiavi k_n hanno di finire in i : se questa poi risulta uguale a $1/m$ la funzione hash gode di uniformità semplice.

Problema fondamentale

Per poter ottenere una **funzione hash con uniformità semplice**, la **distribuzione delle probabilità P** deve essere **nota**.

Ad esempio, si supponga di avere $m = 10$ celle nella tabella.

- 90% delle chiavi che iniziano con "A";

- 10% delle chiavi che iniziano con "Z".

Per costruire una funzione hash con uniformità semplice bisognerebbe fare in modo che le celle che iniziano con "A" occupino il 90% delle celle (0...8), mentre le chiavi che iniziano con "Z" occupino il 10% delle celle (cella 9) in modo che la probabilità che una chiave cada nelle celle 0...8 sia dello 0.9 e la probabilità che cada nella cella 9 sia dello 0.1.

Nella realtà però non sappiamo quali chiavi verranno inserite né con quale frequenza compariranno, quindi **non è possibile costruire una funzione hash perfettamente uniforme**. In sostanza la distribuzione esatta $P(k)$ non può essere nota e si va ad utilizzare tecniche **euristiche**.

Le tecniche euristiche sono delle funzioni hash che usano **formule matematiche** per distribuire bene le chiavi anche se non si conosce $P(k)$, sperando che nella pratica funzionino bene.

11.3 Come realizzare una funzione hash

Dal momento che per utilizzare le tecniche euristiche **si ha bisogno di numeri**, si presenta la necessità di **codificare le chiavi** in numeri naturali. Dunque, se la chiave non è un numero (ad esempio una stringa "ciao") bisogna prima convertirla.

Assunzione

Si assume che le chiavi possano essere tradotte in valori **numerici non negativi**, eventualmente interpretando la loro rappresentazione in memoria come un numero.

Esempio: trasformazione di stringhe

Consideriamo una chiave k costituita da una stringa di caratteri, la trasformazione in intero avviene seguendo dei passaggi logici:

1. $ord(c) \rightarrow$ **Codifica dei caratteri**: Si prende il valore ordinale binario del carattere c secondo una codifica standard (ad esempio ASCII);
2. $bin(k) \rightarrow$ **Rappresentazione binaria**: si concatenano i valori binari dei singoli caratteri che compongono la chiave k .
3. $int(b) \rightarrow$ **Interpretazione intera**: la sequenza di bit risultante viene interpretata come un unico grande numero intero.

Utilizzando una codifica ASCII a 8 bit, e concatenando i risultati di ogni lettera, si ottiene un numero a 24 bit

```
bin("DOG") = ord("D")  ord("O")  ord("G")
             = 01000100  01001111  01000111
int ("DOG") = 68 · 2562 + 79 · 256 + 71      = 4.476.743
```

Come detto alla fine del capitolo 11.2, le chiavi vanno ben distribuite proprio perché si ha il problema che il numero ottenuto è solitamente molto grande (nell'esempio precedente > 4 milioni), e la tabella hash non ha 4 milioni di righe, magari solo $m = 100$.

I metodi che seguiranno sono esattamente quelle tecniche euristiche che utilizzano formule matematiche progettate per **distribuire** (o "sparagliare") uniformemente questi grandi numeri all'interno delle poche celle disponibili.

11.3.1 Metodo dell'estrazione

Il metodo dell'estrazione è uno dei più semplici ed efficienti per definire una funzione hash su chiavi binarie.

Come funziona?

- Si assume che la **dimensione della tabella** sia una **potenza di due**, ovvero $m = 2^p$;
- Si seleziona un **blocco di p bit** dalla rappresentazione binaria della chiave;
- L'indice hash è ottenuto **convertendo questi bit in un numero intero**.

Esempi di utilizzo

Si sceglie una tabella con $m = 2^p = 2^{16} = 65536$ celle.

Per indirizzarle tutte, **necessiteranno** di $p = 16$ bit estratti dalla chiave, ad esempio:

- Si può scegliere il **blocco dei 16 meno significativi**

```
bin("Alberto") = 01000001 01101100 01100010 01100101 01110010 01110100  
                  01101111  
bin("Roberto") = 01010010 01101111 01100010 01100101 01110010 01110100  
                  01101111  
H("Alberto") = int(01110100 01101111) = 29.807  
H("Roberto") = int(01110100 01101111) = 29.807
```

- Oppure si prendono 16 bit partendo da una **posizione casuale interna**

```
bin("Alberto") = 01000001 01101100 01100010 01100101 01110010 01110100  
                  01101111  
bin("Alessio") = 01000001 01101100 01100010 01100101 01110011 01110011  
                  01101111  
H("Alberto") = int(00010110 11000110) = 5.830  
H("Alessio") = int(00010110 11000110) = 5.830
```

Come si può vedere dagli esempi, nonostante questa metodologia sia semplice ed efficiente, è **estremamente sensibile** alla scelta dei **bit selezionati per la chiave**, infatti, variazioni minime delle chiavi come anagrammi o suffissi simili produrranno **collisioni frequenti**.

Anche selezionare bit da altre parti della chiave può risultare inefficace, specialmente se le chiavi condividono prefissi o segmenti centrali comuni.

Proprio per questo motivo, **se non si conosce la distribuzione dei dati, questo approccio è sconsigliato**, in quanto scarta completamente l'informazione contenuta nei bit non selezionati.

11.3.2 Metodo dello XOR

Il metodo dello XOR nasce per risolvere il difetto del metodo dell'estrazione, nel quale, se prendi solo alcuni bit, **ignori completamente i primi**.

Quindi, se cambia un bit all'inizio della chiave, l'hash non cambia, mentre con lo XOR si vuole che **tutti i bit partecipino al calcolo dell'indice**.

Come funziona?

- Anche in questo caso si assume che la **dimensione della tabella sia una potenza di due**, ovvero $m = 2^p$;
- Si suddivide la rappresentazione binaria della chiave k in q sotto-blocchi, ciascuno di lunghezza pari a p bit (la dimensione dell'indirizzo della tabella);
- L'indice hash $h(k)$ è ottenuto effettuando lo XOR progressivo tra tutti i blocchi e **convertendo la somma finale in un numero intero**.

Esempio: si immagini che la chiave sia una striscia di carta lunga (tanti bit) e che la si voglia ridurre a un quadratino piccolo (l'indice p). Invece di tagliare un pezzo e buttare il resto (estrazione), si piega la striscia su se stessa sovrapponendo i bit.

Esempio di utilizzo

Utilizziamo sempre una tabella con $m = 2^p = 2^{16} = 65536$ celle, 16 bit di indirizzamento.

In questo caso, dovendo dividere la chiave in sotto-blocchi da 16 bit, risultano 5 gruppi, uno dei quali viene riempito con 8 zeri di "padding". Utilizzando il metodo XOR è come se facessimo tante addizioni in colonna per le quali non si considera il riporto.

```
bin("montresor") =
01101101 01101111 ⊕
01101110 01110100 ⊕
01110010 01100101 ⊕
01110011 01101111 ⊕
01110010 00000000

H("montresor") =
int(01110000 00010001) = 28.689
```

```
bin("sontremor") =
01110011 01101111 ⊕
01101110 01110100 ⊕
01110010 01100101 ⊕
01101101 01101111 ⊕
01110010 00000000

H("sontremor") =
int(01110000 00010001) = 28.689
```

Nonostante questo approccio permetta di utilizzare ogni singolo bit della chiave per influenzare il risultato finale, presenta alcune problematiche:

- **Vulnerabilità agli anagrammi:** due chiavi composte dagli stessi caratteri in ordine diverso (come nell'esempio sopra illustrato) possono generare lo stesso indice hash, e di conseguenza collisioni;
- L'**efficacia** dipende fortemente dalla suddivisione in blocchi e dall'eventuale padding applicato.

11.3.3 Metodo della divisione

I metodi basati sulla manipolazione dei bit visti ai capitoli 11.3.1 e 11.3.2 soffrono di problemi legati alla regolarità dei dati (ad esempio, suffissi identici) o alla commutatività (anagrammi). Questo perché, caratteri uguali in chiavi differenti, vengono rappresentati con la stessa codifica ASCII a 8 bit.

Il **metodo della divisione** cambia approccio, utilizzando una **logica aritmetica** al posto di una logica basata sui bit. Viene infatti utilizzato un sistema posizionale (come le unità, decine e centinaia), dove i caratteri più a sinistra valgono di più, ad esempio:

```
"AB"=ord("A")·2561 + ord("B")·2560
"BA"=ord("B")·2561 + ord("A")·2560
```

Come funziona?

Utilizzando la logica aritmetica si converte quindi la chiave in un **intero**, che grazie al sistema posizionale è **sempre diverso** (evitando così anagrammi), di cui si calcola il **resto della divisione per m** : il resto della divisione per m è **sempre un numero compreso tra 0 e $m - 1$** .

In questo modo è possibile indirizzare correttamente tutte le celle della tabella hash senza "**uscire dai bordi**".

La scelta del numero di celle (m)

La scelta di m determina la **qualità della distribuzione**: è necessario che sia **un numero dispari**, preferibilmente **un numero primo** non troppo vicino a una potenza di due.

Le chiavi reali spesso presentano dei pattern o regolarità (ad esempio, indirizzi di memoria che sono multipli di 4 o 8). Se m avesse un divisore in comune con il "passo" o il pattern delle chiavi, la funzione hash mapperebbe le chiavi solo su un sottoinsieme delle celle disponibili, lasciandone molte vuote e causando collisioni. Un numero primo non possiede divisor (a parte 1 e se stesso), minimizzando la probabilità di interazione con i pattern dei dati e garantendo una distribuzione più uniforme ("sparpagliamento") su tutta la tabella.

Esempio di utilizzo

Si sceglie un numero di celle secondo le specifiche descritte prima, ad esempio $m = 383$

$H(\text{"Alberto"}) = 18.415.043.350.787.183 \mod 383 = 221$
$H(\text{"Alessio"}) = 18.415.056.470.632.815 \mod 383 = 77$
$H(\text{"Cristian"}) = 4.860.062.892.481.405.294 \mod 383 = 130$

Dunque, il metodo della divisione è facile da implementare ed efficace **solo se m viene scelto con cura**, altrimenti possono verificarsi le seguenti problematiche:

- Se si sceglie un numero di celle per la tabella hash pari a $m = 2^p$, l'operazione modulo della chiave considera solo gli ultimi p bit per la creazione dell'indice.
In questo modo viene buttata via parte dell'informazione della chiave che poteva essere utilizzata per la creazione del suo indice all'interno della tabella hash, "regredendo" così al metodo dell'estrazione (capitolo 11.3.1);
- Se invece si sceglie un numero di celle pari a $m = 2^p - 1$, bisogna fare attenzione che la **base di rappresentazione dei dati** (es. 256) divisa per m **non dia resto 1**. Questo perché verrebbe a mancare il peso conferito a ciascun carattere tramite il sistema posizionale, facendo diventare la funzione hash una semplice somma, incapace di distinguere gli anagrammi.

Esempio pratico (in base 10)

Immaginamo che $m = 9$, con due numeri anagrammi: 12 e 21.

Calcoliamo l'indice della tabella hash (il resto della divisione per 9):

- $H(12)$: Normalmente è $1 \cdot 10 + 2$, ma col modulo 9, il 10 diventa un 1. Dunque: $1 \cdot 1 + 2 = 3$.
- $H(21)$: Normalmente è $2 \cdot 10 + 1$, ma col modulo 9, il 10 diventa un 1. Dunque: $2 \cdot 1 + 1 = 3$.

Come si può vedere, entrambe le chiavi vengono mappate con indice 3, creando collisione.

11.3.4 Metodo della moltiplicazione (Knuth)

A differenza del metodo della divisione (capitolo 11.3.3), nel quale ci si affidava alla scelta di un m primo per "rompere" i pattern delle chiavi, **il metodo della moltiplicazione** utilizza una costante C per mescolare i bit. Questo permette di ottenere indici ben distribuiti pur scegliendo un valore di m comodo per il calcolatore (ad esempio una potenza di 2).

Come funziona?

Il processo avviene in **due passaggi**:

- **Mescolamento:** si moltiplica il valore numerico associato alla chiave k per una costante C (spesso **irrazionale**) compresa tra 0 e 1, e si estrae la parte frazionaria (decimale) del risultato. Questa operazione "distrugge" i pattern della chiave originale;
- **Mappatura:** Si moltiplica questa parte frazionaria per m (il numero di celle) così da ottenere un indice valido tra 0 ed $m - 1$.

La formula che esprime questi passaggi è la seguente: $h(k) = \lfloor m(kC \bmod 1) \rfloor$

Perché conviene che C sia irrazionale?

Un numero razionale semplice (come ad esempio $0.5 = 1/2$) ha una rappresentazione binaria finita e regolare. Moltiplicare per esso equivale spesso a un semplice scorrimento di bit (shift), con il rischio di estrarre nuovamente pattern prevedibili.

Dunque è preferibile che C sia un **numero irrazionale**, cioè un numero non esprimibile tramite un rapporto di due numeri interi che ha una **rappresentazione decimale illimitata** (dopo la virgola le cifre continuano all'infinito) e **non periodica** (non formano una sequenza ripetitiva).

Nello specifico *Knuth* suggerì il valore $C = (\sqrt{5} - 1)/2$, poiché presenta delle caratteristiche matematiche interessanti che permettono una distribuzione più uniforme ("spagliata"), minimizzando le collisioni.

Esempio di utilizzo

Sia $m = 2^{16} = 65536$ celle e $C = (\sqrt{5} - 1)/2 \approx 0.6180339887$.

Se moltiplico $C \cdot k$, dove k è il valore numerico associato a ciascuna chiave dopo la codifica ASCII e la trasformazione in intero utilizzando il sistema posizionale, e estraggo la parte decimale, ottengo una parte decimale che risulterà distribuita in modo **pseudo-casuale**.

Dopodiché moltiplico la parte decimale per m ottenendo un indice valido tra 0 e $m - 1$.

$$H(\text{"Alberto"}) = 65.536 \cdot 0.78732161432 = 51.598$$

$$H(\text{"Alessio"}) = 65.536 \cdot 0.51516739168 = 33.762$$

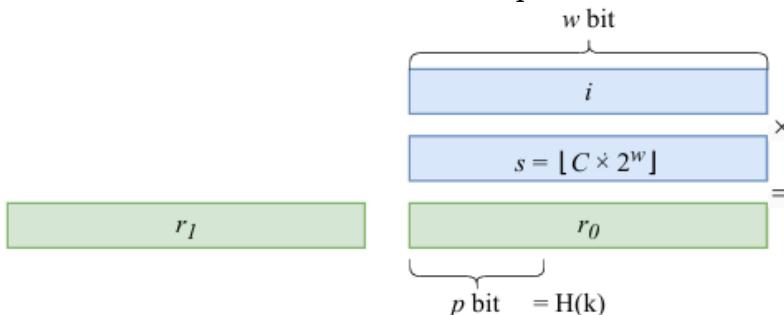
$$H(\text{"Cristian"}) = 65.536 \cdot 0.72143641000 = 47.280$$

A questo punto è possibile effettuare alcune osservazioni:

- il metodo della moltiplicazione funziona per qualsiasi valore di m , ma è **particolarmente efficiente** se $m = 2^p$;
- Il **risultato dipende fortemente dal valore di C** , infatti come detto prima valori irrazionali sono preferibili per garantire una buona distribuzione;
- È **indipendente da particolarità delle chiavi** (prefisi, suffissi).

A differenza di come è stato spiegato a livello teorico, a **livello pratico** per un computer fare calcoli con la virgola rappresenta un **problema** che causa **lentezza e minor precisione**. Dunque, per utilizzare questo metodo, un calcolatore ha la necessità di eseguire lo stesso calcolo utilizzando **solo aritmetica intera**, e per farlo in modo performante sfrutta la **dimensione delle parole del processore**.

Una CPU ha dei contenitori fissi (i registri) la cui dimensione può essere espressa con w , che solitamente nei calcolatori moderni equivale a 64 bit. Usando un w più piccolo (< 64) sprechere-



remmo spazio nei contenitori (bit inutilizzati), mentre utilizzando un w più grande (> 64) avremmo bisogno di due contenitori e di calcoli più complessi.

Lavorando in questo modo, l'idea è quella di sostituire la costante

reale C con una **costante intera** S , che contiene la moltiplicazione tra C (0.618..) e 2^{64} in modo tale da spostare la virgola di 64 posizioni verso destra, ma siccome il registro del computer ha solo 64 posti tutto quello che "avanza" ancora dopo la virgola cade nel vuoto e viene perso. In questo modo la parte decimale (dopo la virgola) scompare e all'interno di S rimane solo un numero intero gigantesco che "riempie" perfettamente la parola in memoria.

Come fatto prima a livello teorico, ora è necessario moltiplicare la costante S (intero) per k (anch'esso un intero rappresentativo del valore della chiave di grandezza $w = 64$ bit) in modo da ottenere il numero con la parte **decimale pseudo-casuale**.

Quando la CPU moltiplica due numeri interi a w bit viene prodotto un risultato **sempre grande il doppio** ($2w$ bit), e il processore salva il risultato in due registri separati automaticamente:

- Registro Alto (r_1): che contiene i primi w bit, che rappresentano la parte intera;
- Registro Basso (r_0): che contiene gli ultimi w bit, che rappresentano la parte frazionaria.

Di queste due parti ci interessa solo il registro basso (r_0), che rappresenta la **parte frazionaria pseudo-casuale**. Per ottenere l'indice finale tra 0 e $m - 1$ (assumendo $m = 2^p$), non serve eseguire un'altra moltiplicazione: è sufficiente estrarre i p bit più significativi di r_0 . Questo equivale matematicamente a moltiplicare per m e troncare, ma a livello hardware è un semplice spostamento di bit (shift), istantaneo per la CPU.

11.4 Gestione delle collisioni

Come abbiamo già detto al capitolo 11.1.1, quando due chiavi vengono mappate dalla funzione hash nella stessa posizione della tabella, si verifica una **collisione**.

Idea generale per la gestione delle collisioni

- **Inserimento**: se la posizione calcolata tramite la funzione hash risulta già occupata, è necessario trovare una **posizione alternativa** in cui memorizzare la chiave.
- **Ricerca**: se durante la ricerca di una chiave, essa non si trova nella posizione attesa, bisogna **cercarla nelle posizioni alternative**, secondo la **strategia adottata** in fase di inserimento.

Esempio con analogia

- **Inserimento:** ho il biglietto per il posto auto numero 5, ma arrivo ed è occupato. Seguendo la regola del parcheggio, provo il posto successivo finché non ne trovo uno libero, ad esempio, 6 occupato e 7 libero;
- **Ricerca:** quando torno a riprendere l'auto il biglietto dice ancora 5, ma a quel parcheggio non c'è la mia auto. Allora devo agire **analogamente** all'inserimento, quindi se so che il 5 era occupato controllo prima il 6 e poi il 7.

Dato che le collisioni sono inevitabili, soprattutto quando l'universo delle chiavi è molto più ampio della dimensione della tabella, è fondamentale disporre di un **meccanismo per gestirle in modo efficiente**.

Le posizioni alternative possono essere trovate all'**interno della tabella** o all'**esterno della tabella**, ciò evidenzia **due possibili approcci** per la gestione delle collisioni:

- Le posizioni alternative **esterne** alla tabella prendono il nome di **liste di trabocco**;
- Le posizioni alternative **interne** alla tabella prendono il nome di **indirizzamento aperto**;

A volte può succedere che quando si va a cercare una chiave la quale non si trova nel posto giusto, poi la si va a cercare in posti alternativi, e se non si trova, la si cerca ancora in altri posti alternativi finché:

- Si trova la chiave;
- Si trova una **cella vuota**. Se si trova un buco significa che la chiave non c'è (perché se ci fosse stata sarebbe dovuta essere lì o dopo).

In una tabella hash ben progettata, dove le chiavi sono ben "sparpagliate", si calcola l'hash, si va alla cella e si trova il dato. Ci si trova quindi nel caso medio, dove il costo è rappresentato da $O(1)$ e non importa se gli elementi sono 10 o 10 milioni, proprio perché l'accesso al dato è immediato.

Invece, in alcuni casi, ad esempio in presenza di molte collisioni o di una funzione hash mal progettata, nel caso medio il tempo può degradare fino a $O(n)$.

11.4.1 Liste di trabocco (concatenamento o chaining)

Le **liste di trabocco** rappresentano un approccio per la gestione delle collisioni che si basa sull'utilizzo di **liste monodirezionali**.

Idea generale

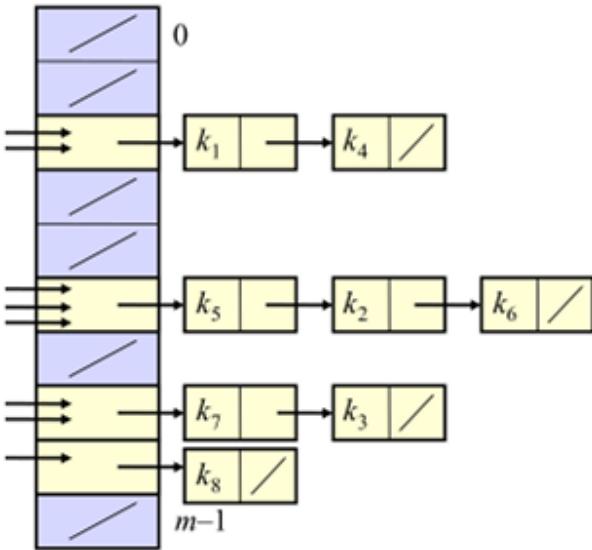
Ogni indice (slot) della tabella hash non contiene direttamente una chiave, ma un **puntatore alla testa di una struttura dinamica** (lista monodirezionale-vettore dinamico), in cui vengono memorizzate tutte le chiavi che **condividono lo stesso indice hash**.

Come struttura dinamica da utilizzare sono sufficienti le liste monodirezionali poiché o si trova o non si trova la chiave durante le operazioni di `insert()`, `lookup()`, `remove()`.

Facendo riferimento alla figura 81, nel momento in cui vengono inserite due chiavi nell'indice 2, tre nell'indice 5, e altre due nell'indice 7, esse sono aggiunte ad una lista (in corrispondenza dell'indice) che cresce in modo arbitrario.

Dunque, le operazioni effettuate sulle liste utilizzano la seguente logica:

Figure 81: liste di trabocco



- **insert() in coda:** viene utilizzato se devo inserire una chiave e verificare che questa non sia già stata inserita, devo scorrere tutta la lista. Se la chiave che sto inserendo è già presente mi fermo e sovrascrivo il valore, altrimenti arrivo in fondo, aggiungo l'elemento alla lista e lo memorizzo;
- **insert() in testa:** viene utilizzato quando la lista è vuota per inserire una chiave senza preoccuparsi che questa sia già presente;
- **lookup(), remove():** in entrambi i casi bisogna scorrere tutta la lista per cercare la chiave e restituire il valore corrispondente/rimuovere la coppia chiave-valore nella lista.

11.4.2 Liste di trabocco: analisi della complessità

A questo punto si vuole fare un'**analisi della complessità** di questo meccanismo in particolare, e per fare ciò bisogna definire alcuni parametri:

- $n \rightarrow$ numero di chiavi memorizzate in tabella hash;
- $m \rightarrow$ capacità della tabella hash;
- $\alpha = n/m \rightarrow$ si riferisce al **fattore di carico** che nel caso delle liste di trabocco può essere:
 - $\alpha = 0$ se la tabella è vuota;
 - $0 < \alpha < 1$ se la tabella non è completamente occupata;
 - $\alpha > 1$ se la tabella contiene tanti elementi, non esiste un limite superiore.

Partendo da questi parametri, si ottengono poi due informazioni importanti:

- $I(\alpha) \rightarrow$ numero medio di accessi alla tabella per la ricerca di una chiave non presente all'interno di essa (**ricerca con insuccesso**);
- $S(\alpha) \rightarrow$ numero medio di accessi alla tabella per la ricerca di una chiave presente all'interno di essa (**ricerca con successo**).

Analisi del caso pessimo

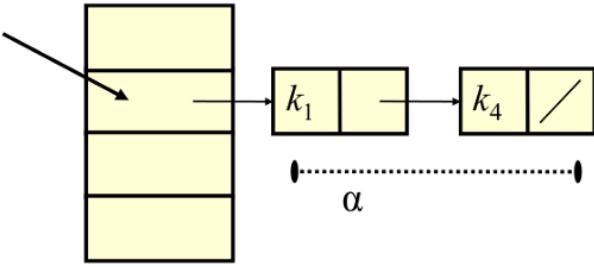
Il caso peggiore, si presenta nel momento in cui ci si ritrova ad avere dei **costi** che sono i **più elevati possibili**: questo avviene quando tutte le chiavi hanno lo **stesso hash**, ovvero tutte le chiavi vengono **collocate nella stessa lista**. Possiamo quindi dire che una struttura dati efficiente come le tabelle hash **si trasformano in una lista non ordinata**.

- $\text{insert}()$ ha costo $\theta(1)$ se non si controlla (inserimento in testa), mentre ha costo $\theta(n)$ nel caso in cui si debba controllare che la chiave sia già stata inserita.
- $\text{lookup}()$ e $\text{remove}()$ hanno entrambi costo $\theta(n)$ poiché in entrambi i casi bisognerà scorrere la lista.

Analisi del caso medio

Per quanto riguarda il caso medio, è necessario fare prima alcune assunzioni:

- Dipende da come le chiavi vengono distribuite;
- Si assume che la funzione hash implementata abbia un hashing uniforme semplice, e che sia possibile calcolarla in $\theta(1)$.



Dopodiché, per lo studio del caso medio, bisogna prendere in considerazione la lunghezza delle liste/vettori. L'idea generale è la seguente: Se ho $n = 1000$ chiavi e 100 celle nel vettore avrò un fattore carico di $\alpha = n/m = 1000/100 = 10$, ciò significa che il numero atteso di elementi in una lista (se come detto prima la distribuzione è uniforme) è 10. In sostanza, se il fattore di carico è tenuto abbastanza piccolo (2, 3 oppure 4), allora le **operazioni avranno un costo che deriva da α** , ad esempio:

- **Ricerca senza successo:** se sto effettuando un tipo di ricerca **senza successo** dovrò guardare tutta la lista di lunghezza α , quindi pago il costo della funzione hash $\theta(1)$ + il costo di analizzare in media l'intera lista di lunghezza α ;
- **Ricerca con successo:** se invece effettuo una ricerca con successo dove potrei trovare la chiave al primo elemento oppure all'ultimo, **in media la troverò a metà** e quindi il costo atteso risulta la somma tra il costo della funzione hash $\theta(1)$ + la metà della lunghezza della lista $\alpha/2$.

Dunque, mantenendo un valore di α abbastanza **piccolo** si garantisce il fatto che i **costi rimarranno costanti**, proprio perché il fattore di carico (α) **influenza il costo computazionale delle operazioni sulle tabelle hash**.

11.4.3 Indirizzamento aperto (memorizzazione interna)

Utilizzando un approccio a **indirizzamento aperto** viene evitato l'utilizzo strutture dati complesse per le quali si utilizzano liste e puntatori (capitolo 11.4.1), ma ci si limita a memorizzare tutte le chiavi direttamente nel vettore che costituisce la tabella hash.

Idea generale

In questo caso l'idea generale è molto semplice: ogni indice (slot) della tabella hash conterrà o una chiave, o un puntatore null.

- **Inserimento:** Se durante l'inserimento lo slot prescelto dalla funzione hash H è già occupato, cerco uno slot alternativo;
- **Ricerca:** dopo l'inserimento, vado a ricercare nello slot prescelto, ma, se anch'esso è occupato da una chiave che non è quella che sto cercando, cerco negli slot alternativi, uno dopo l'altro, fino a quando non trovo la chiave che sto cercando oppure trovo null a significare che ho trovato una casella veramente vuota e che quindi non posso più andare avanti.

Per verificare che all'interno di uno slot sia presente una chiave oppure null ci si basa sul concetto di **ispezione**, ovvero l'**esame di uno slot** durante la ricerca.

Per descrivere questa strategia, la funzione hash viene estesa aggiungendo un secondo parametro, quindi non prende più in ingresso solo l'insieme universo U delle chiavi, ma anche un indice (**indice di ispezione**) compreso fra 0 e $(m - 1)$.

$$H : \mathcal{U} \times \overbrace{[0 \dots m-1]}^{\text{Numero ispezione}} \rightarrow \overbrace{[0 \dots m-1]}^{\text{Indice vettore}}$$

La funzione $H(k, i)$ rappresenta la posizione esaminata quando si cerca nella tabella hash la chiave k (una di quelle dell'insieme U) alla i -esima ispezione, cioè dopo i tentativi falliti:

- Durante un inserimento, i conta quante volte si è trovata una cella occupata;
 - Durante una ricerca, quante volte si è trovata una cella occupata da una chiave diversa da k .
- Ad esempio, si prenda in considerazione la seguente **sequenza di ispezione** generata.

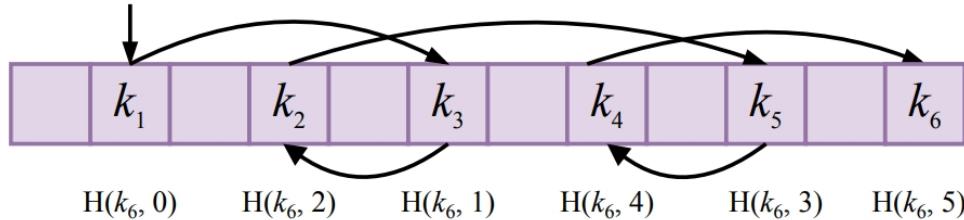


Figure 83: esempio di una sequenza di ispezione composta da 6 passi

Sequenza di ispezione

Una **sequenza di ispezione** $[H(k, 0), H(k, 1), \dots, H(k, m - 1)]$ è una permutazione degli indici $[0, \dots, m - 1]$, corrispondente all'ordine in cui vengono esaminati gli slot.

- Non si vuole esaminare lo stesso slot più di una volta;
- Potrebbe essere necessario esaminare tutti gli slot nella tabella.

In figura 83 si vede come, cercando k_6 , al crescere dell'indice di ispezione si viene portati in posizioni differenti della tabella hash, poiché ad un determinato indice corrisponde una chiave. All'**ultima operazione** della sequenza di ispezione, se la **cella è vuota**:

- **Inserimento**: la chiave viene inserita;
- **Ricerca**: posso dire che k_n cercato non c'è.

Se invece la **cella è piena**:

- **Inserimento**: se la chiave nella cella è uguale a quella da inserire, decido se aggiornarla o dare errore (duplicato). Se è diversa (considerando che stiamo parlando dell'ultima operazione della sequenza) la tabella è piena (Overflow/Errore).
- **Ricerca**: si deve controllare se la chiave in quella cella è quella che si cerca: se sì la chiave è stata trovata, altrimenti la chiave non esiste nella tabella.

11.4.4 Indirizzamento aperto: analisi della complessità

In un approccio ad indirizzamento aperto, il **fattore di carico** è un valore per tutta la tabella che varia da 0 a 1, il quale **indica quanto è saturo**, ovvero la percentuale di **riempimento globale** della tabella hash. A differenza di come è stato visto al capitolo 11.4.2 il significato di α cambia, perché cambia la capienza fisica delle celle:

- **Liste di trabocco**: ogni cella è un puntatore ad una lista che può allungarsi all'infinito, quindi se $m = 10$, $n = 20$, avrò $\alpha = 2$ a significare che **in media, ogni lista contiene 2 elementi**.
- **Indirizzamento aperto**: invece, in questo caso, ogni cella può contenere al massimo 1 elemento, quindi n (elementi) non potrà mai superare m celle, di conseguenza $0 \leq \alpha \leq 1$. Se $\alpha = 1$ la tabella è totalmente piena e si entra in **overflow**, facendo schizzare il costo delle operazioni a $O(n)$.

Esempio pratico: ipotizziamo dover trovare posto auto all'interno di un parcheggio.

- **Scenario A** ($\alpha = 0.1$): il parcheggio ha 100 posti e solo 10 macchine al suo interno. La probabilità che il posto assegnato con la funzione hash $h(k)$ sia occupato è bassissima (10%). Dunque, quasi sicuramente si troverà il posto auto al primo colpo, indicando ciò con $O(1)$.
- **Scenario B** ($\alpha = 0.9$): il parcheggio ha sempre 100 posti ma questa volta 90 macchine al suo interno. La funzione hash $h(k)$, molto probabilmente, calcolerà una posizione già occupata, per più volte, prima di trovare un posto auto libero all'interno del parcheggio. Ciò significa che bisogna girare tanto prima di trovare il 10% dei posti rimanenti.

In modo analogo all'esempio, il costo tende a $O(n)$ (cioè bisogna scorrere tutta la tabella) quando α si avvicina ad 1. Le collisioni diventano la norma, non l'eccezione, e le sequenze di ispezione si allungano a dismisura.

In un **contesto ideale**, per fare in modo che non si debba scorrere tutta la tabella e che quindi il costo non salga a $O(n)$, quello che si ricerca è una situazione di **hashing uniforme**.

Hashing uniforme ideale

Utilizzando un **hashing uniforme** ogni chiave (k_n) dell'insieme U ha la stessa probabilità di avere come sequenza di ispezione $H(k_n, 0), H(k_n, 1), \dots, H(k_n, m - 1)$ una qualsiasi delle $m!$ permutazioni di $[0, \dots, m-1]$.

Dunque la funzione hash non dà solo il punto di partenza, ma consegna l'intera lista delle celle da visitare, in ordine secondo una qualsiasi permutazione.

Esempio pratico

Prendiamo in considerazione due chiavi A e B , le quali per casualità, su una tabella da 5 posizioni partono dalla cella 2.

- Se si utilizzasse una regola fissa, ad esempio "*vai al successivo finché non trovi una cella vuota*", si creerebbe un "muro" (clustering) prima di poter inserire la chiave nell'indice:

- $A: 2 \rightarrow 3 \rightarrow 4 \rightarrow \dots$
- $B: 2 \rightarrow 3 \rightarrow 4 \rightarrow \dots$

Questo fino a che una delle due trova per prima la cella vuota e subito dopo la rimanente va +1 avanti per essere inserita (**primary clustering - capitolo 11.4.5**).

- Se usiamo l'**Hashing uniforme**, il sistema assegna due itinerari completi totalmente diversi, pescati a caso tra tutte le combinazioni possibili ($m!$), in modo tale da non creare dei "muri". In questo modo ogni chiave può essere inserita il prima possibile.
- $A: 2 \rightarrow 0 \rightarrow 4 \rightarrow 1 \rightarrow 3$
- $B: 2 \rightarrow 3 \rightarrow 1 \rightarrow 0 \rightarrow 4$

Nella pratica, implementare un vero hashing uniforme è difficile.

Non possiamo garantire che il percorso di ricerca sia perfettamente casuale per ogni chiave (troppo costoso per il processore), ma **bisogna obbligatoriamente garantire** che il percorso non entri in un "ciclo vizioso". La funzione di ispezione deve essere progettata matematicamente affinché, al variare di i da 0 a $m - 1$, vengano generati tutti gli indici della tabella esattamente una volta: dunque viene utilizzata **una singola permutazione**.

Per fare ciò vengono utilizzate tre tecniche principali:

- Ispezione lineare;
- Ispezione quadratica;
- Doppio hashing

11.4.5 Indirizzamento aperto: ispezione lineare

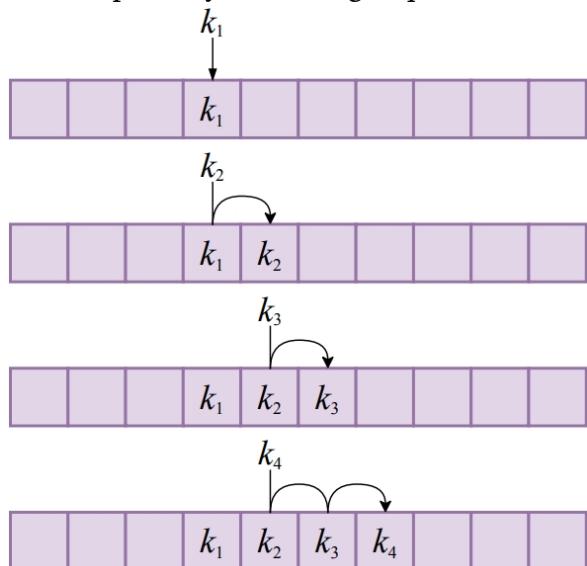
Metodologia utilizzata

In questo metodo, si utilizza un parametro δ che rappresenta la distanza tra due posizioni successive esaminate durante l'ispezione: $H(k, i) = (H_1(k) + \delta \cdot i) \bmod m$

Dunque, si prende la posizione di partenza e poi si va avanti per un numero fisso di caselle δ . Quindi, quando vado a fare l'ispezione "zeresima", con $i = 0$, vado a guardare nella funzione $H_1(k)$, con $i = 1$ vado a guardare nella funzione $H_1(k) + \delta$, con $i = 2$ nella funzione $H_1(k) + 2 \cdot \delta$, e così via.

Dividere per il modulo di m garantisce di ottenere sempre un indice interno alla tabella, in modo tale che quando si supera l'ultima posizione della tabella, l'ispezione riprenda dall'inizio, rendendo il processo circolare.

Questo metodo è semplice, ma presenta un limite importante, quello dell'**agglomerazione primaria** (primary clustering): quando si vuole inserire una chiave all'interno di un determinato



indice all'interno della tabella hash, essa collide con l'elemento già presente, e prosegue la sua ricerca fino a trovare la prima cella libera.

Dato che si sta utilizzando una metodologia a "**ispezione lineare**" (quindi si va avanti di una dimensione fissa), se la posizione iniziale della chiave cade all'interno di una sequenza di celle occupate, tutte queste verranno trovate già piene e la chiave finirà inevitabilmente per essere inserita immediatamente dopo la sequenza, allungando ulteriormente la quest'ultima.

Questo fenomeno porta alla formazione di sequenze sempre più lunghe di posizioni a distanza

δ l'una dall'altra, il che rallenta le operazioni e i **tempi medi** di inserimento e cancellazione crescono. Dunque, uno slot vuoto che si trova dopo i slot già occupati nella stessa sequenza viene riempito con probabilità $(i+1)/m$. L'ispezione lineare è quella "colpevole" dei muri, e il doppio hashing, come si vedrà al capitolo 11.4.7, è la "soluzione reale" al problema.

11.4.6 Indirizzamento aperto: ispezione quadratica

Metodologia utilizzata

L'unica differenza rispetto all'utilizzo di un ispezione lineare è che la **distanza** (offset) tra due posizioni successive nella sequenza di ispezione non è costante, ma **cresce quadraticamente** con i : $H(k, i) = (H_1(k) + \delta \cdot i^2) \bmod m$.

- con $\delta = 1$ e $i = 0$ si esamina prima la posizione $(H_1(k) + 0)$;
- con $\delta = 1$ e $i = 1$ si esamina la posizione $(H_1(k) + \delta \cdot 1)$;
- con $\delta = 1$ e $i = 2$ si esamina la posizione $(H_1(k) + \delta \cdot 4)$;
- con $\delta = 1$ e $i = 3$ si esamina la posizione $(H_1(k) + \delta \cdot 9)$, e così via;

Questa variazione di indice riduce l'effetto dell'**agglomerazione primaria**, tipico dell'ispezione lineare, poiché due chiavi con indirizzi iniziali differenti ($H_1(k) \neq H_1(k')$) tendono a generare sequenze di ispezione che divergono più rapidamente.

Tuttavia, se due chiavi condividono lo stesso indirizzo iniziale ($H_1(k) = H_1(k')$) le sequenze di ispezione coincidono completamente, e il problema si ripresenta in questo caso prendendo il nome di **agglomerazione secondaria**.

Un ulteriore limite, è che **la sequenza di ispezione generata non è una permutazione** dell'insieme $\{0, \dots, m - 1\}$: può quindi accadere che alcune posizioni libere non vengano mai raggiunte, impedendo l'inserimento di nuove chiavi anche se la tabella non è piena.

11.4.7 Indirizzamento aperto: doppio hashing

Il **doppio hashing** è il metodo che viene utilizzato maggiormente per risolvere il problema dell'agglomerazione.

Metodologia utilizzata

Vengono utilizzate **due funzioni hash**, la prima indica la posizione iniziale e la seconda indica l'offset, che quindi cambia per ogni chiave: $H(k, i) = (H_1(k) + i \cdot H_2(k)) \bmod m$.

In questo modo, anche se due chiavi k e k' hanno lo stesso indirizzo iniziale $H_1(k) = H_1(k')$, non è detto che abbiano lo stesso valore in H_2 (anzi, la probabilità è relativamente bassa), dunque tendono ad essere sparse in maniera più "*uniforme*" riducendo drasticamente la formazione di **agglomerati primari e secondari**.

Utilizzando questa metodologia si hanno al massimo m^2 sequenze di ispezione distinte:

- m scelte possibili per decidere da dove partire (cella $0, 1, \dots, m - 1$);
- m scelte possibili per decidere quanto lunghi fare i passi (offset di $1, 2, 3, \dots$)

Il numero totale di combinazioni uniche (coppie partenza-passo) è quindi $m \cdot m = m^2$.

Però non basta avere tanti percorsi diversi (m^2), bisogna essere sicuri che **ogni percorso sia valido**, ovvero che continuando a saltare, **si debbano toccare tutte le celle della tabella** prima di ritornare al **punto di partenza**. Per garantire ciò si usano principalmente due metodi:

1. *Metodo della potenza di 2*

- Si sceglie $m = 2^p$ (ad esempio, 16, 32, 64, ...), in modo tale che l'unico divisore di m sia 2;
- Ci si assicura che $H_2(k)$ dia sempre un **numero dispari**.

Funziona perché un numero dispari non è mai divisibile per 2, quindi sono per forza primi tra loro.

2. *Metodo del numero primo*:

- Si sceglie m come numero primo (11, 13, 17, ...);
- Ci si assicura che $H_2(k)$, dia un numero positivo minore di m ($0 < H_2 < m$);

Funziona perché un numero primo non divisibile per nulla tranne che per se stesso.

11.4.8 Indirizzamento aperto: cancellazione

Durante la gestione delle collisioni, utilizzando un approccio ad indirizzamento aperto, se si vuole cancellare un elemento dalla tabella hash **non è possibile** semplicemente sostituire la chiave che si vuole cancellare con un valore null.

Questo perché si potrebbe "*rompere*" **una sequenza di ispezione**: infatti, come si può vedere

in figura 85, durante la ricerca di una chiave k_n , il raggiungimento di una cella "null" non garantisce che k_n non sia presente altrove nella tabella hash, poiché potrebbe essere semplicemente in una posizione differente sempre all'interno della stessa. Contrassegnare la cella come "null" **arresterebbe l'algoritmo** di ricerca, **spezzando il collegamento verso le chiavi successive** nella sequenza di ispezione.

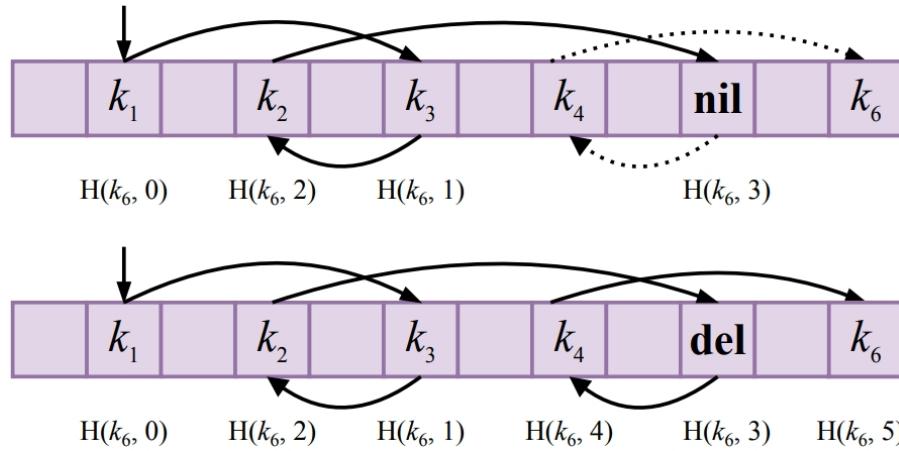


Figure 85: interruzione sequenza di ispezione

Proprio per questo motivo è necessario introdurre un **marcatore speciale** che indichi esplicitamente che la cella era occupata in passato ma è stata svuotata. L'elemento in questione è ***deleted* (del)** che permette di continuare l'ispezione fino a raggiungere la chiave ricercata (nel caso in cui sia effettivamente presente all'interno della tabella hash):

- Durante una **ricerca** il *deleted* viene trattato come uno slot pieno.
- Durante un **inserimento** il *deleted* viene trattato come uno slot vuoto, perché non ci interessa che sia stato cancellato in precedenza.

Tuttavia, aggiungere il marcatore *deleted* comporta lo svantaggio di aumentare i tempi di ricerca, che non dipendono più solamente dal fattore di carico α ($I(\alpha), S(\alpha)$ - capitolo 11.4.2), ma anche da quanti elementi sono stati cancellati.

11.5 Implementazione dell'hashing doppio

L'hashing doppio è indubbiamente la **tecnica migliore** da implementare **utilizzando** una gestione delle collisioni ad **indirizzamento aperto**, poiché permette di ridurre drasticamente clustering primari e secondari.

11.5.1 Definizione delle variabili

La tabella hash viene **memorizzata mediante due vettori paralleli**, a cui in seguito verrà assegnata la dimensione m (valore che detta la dimensione della tabella):

- *key* il quale memorizza le chiavi;
- *values* che memorizza i valori ad esse associati.

HASH

item [] key	% Vettore delle chiavi
item [] values	% Vettore dei valori
int m	% Dimensione della tabella

Ovviamente, come spiegato al capitolo 11.4.3 e 11.4.8, l'implementazione dell'indirizzamento aperto prevede che ogni cella di del vettore *key* contenga una chiave valida oppure uno dei due valori speciali *null* o *deleted*, che indicano rispettivamente una posizione mai utilizzata e una posizione occupata in passato ma che è stata cancellata.

11.5.2 La funzione hash

Questa funzione prende in ingresso un parametro "dim" ad indicare la dimensione della tabella hash. Inizialmente viene definita la dimensione della tabella, dando un valore alla variabile "*m*" tramite il parametro "dim" che la funzione prende in ingresso. Successivamente ci si occupa di inizializzare tutte le chiavi e i valori (in particolare solo le chiavi poiché i valori al momento non interessano) con il valore null.

HASH Hash(int dim)

```

HASH t = new HASH
t.m = dim
t.key = new item[0...dim - 1] = {nil}
t.values = new item[0...dim - 1] = {nil}
return t

```

11.5.3 La funzione scan

La funzione *scan* non fa parte dell'API della tabella hash, infatti non è un'operazione del dizionario come *lookup()*, *insert()* e *remove*, ma serve solo per essere richiamata all'interno di esse. La funzione *scan* prende in input la chiave *k* che si vuole ricercare all'interno della tabella hash, e un valore booleano *insert* che indica se si tratta di un'operazione di inserimento oppure no, mentre **ritorna un indice intero che indica la posizione della chiave**, e non il valore associato ad essa.

```

int scan(item k, boolean insert)
    int firstDeleted = -1                                % Prima posizione deleted
    int i = 0                                         % Numero di ispezione
    int j = H1(k)                                     % Posizione attuale
    while key[j] ≠ k and key[j] ≠ nil and i < m do
        if key[j] == deleted and firstDeleted == -1 then
            firstDeleted = j
            j = (j + H'(k)) mod m
            i = i + 1                                    % Prossima ispezione
        if insert and key[j] ≠ k and firstDeleted ≠ -1 then
            return firstDeleted                      % Riutilizza la prima cella cancellata trovata
        else
            return j                                  % Restituisci la cella trovata
    
```

- La variabile *firstDeleted* è un "trucco" molto intelligente per **ottimizzare le prestazioni** della tabella hash durante l'inserimento. Viene utilizzata per ricordare **la posizione della prima cella deleted**.

Durante una scansione per inserimento, dobbiamo obbligatoriamente arrivare fino in fondo (al *null*) per assicurarci che la chiave non esista già. Tuttavia, se lungo il percorso incon-

riamo delle celle *deleted*, la variabile *firstDeleted*, che inizialmente viene inizializzata a -1 (ad indicare "per ora non ho visto celle cancellate"), memorizza l'indice della prima di queste celle. Se alla fine della scansione la chiave non è stata trovata, l'algoritmo preferisce inserire il nuovo elemento nella posizione *firstDeleted* (riciclando lo spazio) piuttosto che in fondo alla sequenza (`null`). Questo mantiene la **tabella compatta e riduce i tempi di ricerca futuri**.

- La variabile *i* è l'indice del numero di ispezione (ispezione zero, ispezione uno, ispezione due, e così via...);
- La variabile *j* indica la posizione attualmente analizzata: viene inizializzata con il valore $H(k)$ dove H è la funzione hash di base, quella che ci dice il punto di partenza dove andare a cercare e *k* ovviamente è la chiave che stiamo cercando.

A questo punto inizia un ciclo che terminerà qualora:

- trovo la chiave che sto cercando;
- trovo una posizione `null` a significare che non è più possibile andare avanti, perché la chiave cercata non è stata trovata;
- Oppure, nessuna delle due condizioni precedenti si è verificata e il numero di ispezioni raggiunge *m* (*i* = *m*) a significare sostanzialmente che ho ispezionato tutta la tabella senza trovare quello che si stava cercando.

Invece, le due righe all'interno del ciclo while sono quelle che permettono di andare avanti:

- $i = i + 1$ somma +1 all'indice delle ispezioni;
- Per calcolare la prossima cella si utilizza una seconda funzione $H'(k)$ che è l'offset che si va a sommare alla posizione che si stava cercando, come prevede il doppio hashing (capitolo 11.4.7).

Terminato il ciclo while si presentano due possibilità:

- Se sono in **condizione di inserimento** e non ho trovato una chiave da sovrascrivere e *firstDeleted* non è più inizializzato con il valore iniziale (-1), a significare che ho riempito la prima cella *deleted*, allora restituisco proprio *firstDeleted*, poiché l'inserimento è avvenuto lì dentro;
- Altrimenti, sono in **condizione di ricerca** e restituisco *j*, che rappresenta l'ultimo indice guardato e se la chiave non è stata trovata, può essere un valore fittizio.

11.5.4 Le funzioni `lookup` e `insert`

Nel caso della funzione `lookup` viene richiamata la funzione `scan` passando il valore `false` ad indicare che non si tratta di un'operazione di inserimento ma solo di ricerca.

Nel caso della funzione `insert` viene richiamata la funzione `scan` passando il valore `true` ad indicare che si tratta di un'operazione di inserimento.

```
item lookup(item k)
  int i = scan(k, false)
  if key[i] == k then
    | return values[i]
  else
    | return nil
```

(a)

```
insert(item k, item v)
  int i = scan(k, true)
  if key[i] == nil or key[i] == deleted or key[i] == k then
    | key[i] = k
    | values[i] = v
  else
    | // Errore: tabella piena
```

(b)

In entrambi i casi ottengo un indice che viene restituito dalla funzione `scan`:

- Nel caso della `lookup` (a) si hanno due possibilità:
 - si ha trovato quello che si cercava e questo è rappresentato dal fatto che la chiave nell'indice i è uguale alla chiave k che si sta cercando, e allora viene restituito il valore associato a quella chiave;
 - Altrimenti ritorno `null` a significare che l'oggetto non è stato trovato.
- allo stesso modo, nel caso dell'`insert` (b), le possibilità sono:
 - se si ha trovato una **casella vuota** oppure una casella **deleted** oppure una casella che **già contiene la chiave k** , in tutti questi casi si va a sovrascrivere il valore k e il valore v sulle rispettive posizioni i -esime;
 - Altrimenti, la tabella hash è piena poiché non si sono trovate caselle a disposizione dove andare a scrivere il valore k e quindi bisognerà ritornare un qualche tipo di errore. Questo potrebbe essere anche verificato prima di fare una ricerca del genere se si mantenessero informazioni riguardanti il numero di chiavi effettivamente registrate nella tabella, allora a quel punto se il numero di chiavi ha raggiunto m si può subito ritornare che è impossibile effettuare un inserimento.

11.5.5 La funzione `remove`

Nuovamente la funzione `scan` viene passata con il valore `false`, poiché non si tratta di un'operazione di inserimento.

In questo caso la condizione è ancora più semplice:

- Trovo l'elemento e lo cancello, segnando la chiave come **deleted** (per non rompere la sequenza di ispezione) mentre il valore associato a quella determinata chiave; viene impostato a `null`;
- Non trovo l'elemento e non devo fare assolutamente nulla;

```
remove(item k)
  int i = scan(k, false)
  if key[i] == k then
    | key[i] = deleted
                                // Marca come cancellata
```

11.6 Complessità

ISPEZIONE	α	$I(\alpha)$	$S(\alpha)$
Lineare	$0 \leq \alpha < 1$	$\frac{(1 - \alpha)^2 + 1}{2(1 - \alpha)^2}$	$\frac{1 - \alpha/2}{1 - \alpha}$
Hashing doppio	$0 \leq \alpha < 1$	$\frac{1}{1 - \alpha}$	$-\frac{1}{\alpha} \ln(1 - \alpha)$
Liste di trabocco	$\alpha \geq 0$	$1 + \alpha$	$1 + \alpha/2$

Figure 92: Tabella riassuntiva complessità delle tecniche di gestione delle collisioni

La complessità viene misurata in base ad α (fattore di carico - capitolo 11.4.2).

Ricordiamo che per quanto riguarda le tecniche ad indirizzamento aperto il valore di α deve essere compreso tra 0 e 1, mentre per le liste di trabocco il valore di α deve essere maggiore o uguale a 0.

Le formule della complessità della ricerca con successo e della ricerca con insuccesso, per quanto riguarda le tecniche ad indirizzamento aperto, sono di difficile derivazione, mentre ricordiamo che il +1 nelle liste di trabocco indica l'accesso che devo fare alla memoria per iniziare a guardare la lista che stiamo scorrendo.

L'ispezione quadratica ha prestazioni molto vicine a quelle dell'Hashing Doppio (e quindi dell'hashing uniforme), poiché elimina il clustering primario. Tuttavia, a causa del clustering secondario, le sue prestazioni sono leggermente inferiori rispetto al doppio hashing puro, ma drasticamente superiori all'ispezione lineare.

Nella tabella in figura 92 viene omessa perché le sue formule di costo non sono semplici o perché viene considerata un'approssimazione del caso uniforme. Si preferisce mostrare gli estremi dell'indirizzamento aperto:

- **Il caso pessimo (per il clustering):** ispezione lineare;
- **Il caso ottimo (quasi casuale):** doppio hashing.

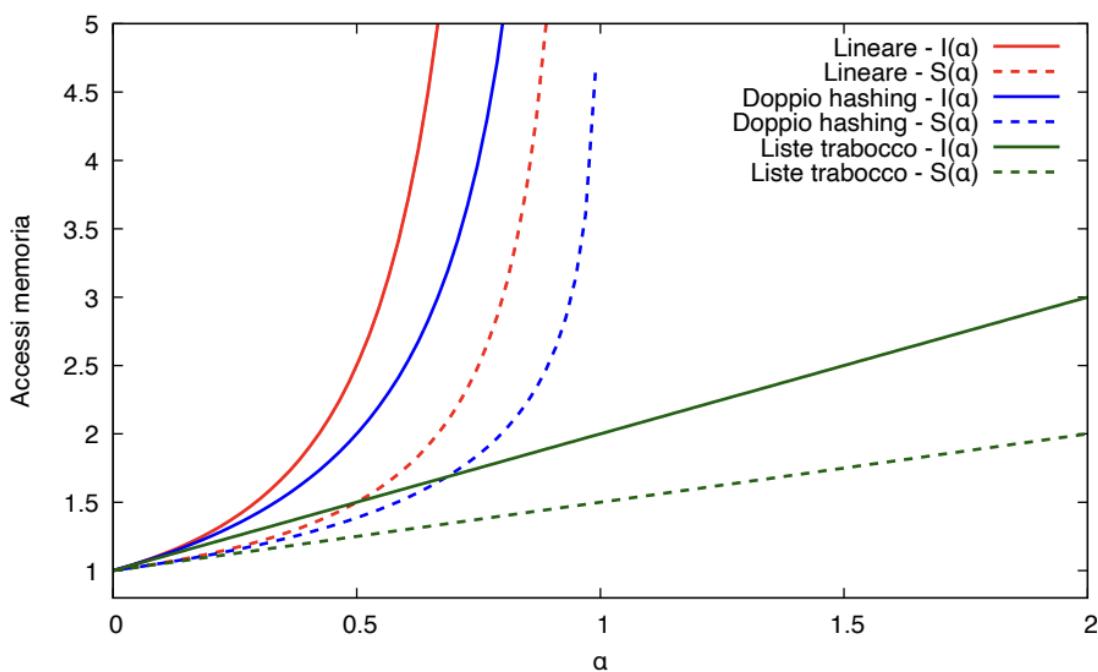


Figure 93: grafico dell'andamento delle complessità

Dal grafico si può notare come la regione che sta a sinistra dell'1 vale sia per le tecniche ad indirizzamento aperto che per le liste di trabocco, mentre la parte a destra dell'1 si riferisce esclusivamente alle liste di trabocco.

11.6.1 Ristrutturazione

Utilizzando dei metodi di ispezione (indirizzamento aperto), è opportuno evitare che la tabella raggiunga un fattore di carico troppo elevato, poiché il tempo medio di ricerca cresce rapidamente all'aumentare di α :

- Se α è basso (es. 0.2, tabella vuota all'80%), è velocissimo;

- Se α supera una certa soglia (es. 0.5 o 50%), le collisioni aumentano esponenzialmente e la tabella diventa lenta.

La soluzione non è aspettare che si blocchi, **ma intervenire prima**.

Dunque, quando il numero di elementi supera la soglia prefissata (ad esempio $\alpha > 0.5/0.75$), scatta l'**operazione di ristrutturazione**, che avviene in tre passaggi:

1. **Nuova Memoria**: si alloca una nuova tabella di dimensione doppia ($2m$) rispetto a quella vecchia;
2. **Re-inserimento (Rehashing)**: si prendono tutte le chiavi attive (non quelle DELETED) dalla vecchia tabella e si reinseriscono nella nuova utilizzando una nuova funzione hash (quindi rimappandone la posizione), poiché m è cambiato;
3. **Pulizia**: la vecchia tabella viene buttata via.

In questo modo il fattore di carico risulta al più dimezzato (non superiore a metà della soglia impostata precedentemente, ad esempio 0.25) e tutte le posizioni sono o libere o occupate: eventuali celle marcate come cancellate vengono eliminate.

Il costo di quest'operazione vale $O(m)$ nel caso pessimo, ma c'è da considerare che accade molto raramente. Quindi, se la maggior parte delle volte, come accade per le operazioni di `lookup`, `insert` e `remove` il costo è $O(1)$, allora è come se stessimo "spalmando" quel costo raro su tutti gli inserimenti veloci fatti prima.

In termini tecnici, si dice che il **costo ammortizzato** rimane costante $O(1)$.

Ovviamente il discorso vale anche al contrario. Se si cancellano tantissimi elementi e la tabella diventa vuota (ad esempio, $\alpha < 0.25$), si spreca memoria. In quel caso, si fa una ristrutturazione per dimezzare la tabella, mantenendo l'efficienza dello spazio.