Università Degli Studi Di Catania

Corso Di Laurea in Informatica

Riassunti di Programmazione II

Autore
Salvo Polizzi

Docente
Prof. Marco Moltisanti

Indice

Lista dei Codici

Parte I

Cenni di Complessità e Tecniche Ricorsive

Capitolo 1

Ripasso sugli Array

In questo primo capitolo faremo un breve ripasso sugli **array** e la loro *manipolazione*.

1.1 Definizione e inizializzazione di un array

Molto banalmente, l'**array** rappresenta una **sequenza di dati** che si trovano in **locazioni di memoria consecutive**, e che hanno una **dimensione fissa**. Tali caratteristiche che definiscono un array, ci permettono di avere il vantaggio di poter *indicizzare* le celle di memoria di un array. Ora però dobbiamo capire come può essere inizializzato un array.

1.1.1 Tipi di inizializzazione

Essenzialmente possiamo **inizializzare un array** in 2 modi:

- **Staticamente**: in tal caso utilizzeremo una zona di memoria detta **pila** o **stack**
- **Dinamicamente**: in tal caso utilizzeremo una memoria detta **heap**, e che ha delle caratteristiche diverse rispetto allo stack.

Inizializzazione statica

Vediamo degli esempi di inizializzazione statica e poi li commentiamo:

```
0 #include<iostream>
1
2 int main() {
3     int array1[100];
4     int array2[]={11,12,13,14};
5 }
```

Codice 1.1: Allocazione Statica di array

Diciamo come premessa che gli array di questo programma sono stati **inizializzati staticamente**. Come abbiamo già detto, gli array hanno una dimensione fissa. Di conseguenza, la dimensione può essere una e una sola. In riga 4, ad esempio, abbiamo inizializzato l'array di interi, ovvero nel quale ogni cella di memoria occupa 4 byte, con dimensione 100. Quindi, se utilizzassimo la funzione *sizeof*, che ci restituisce la dimensione in byte del dato passato, avremmo che *array1* ha dimensione 400. In tal caso però abbiamo semplicemente dichiarato l'array, ma non lo abbiamo inizializzato. Nel secondo caso, abbiamo omesso la dimensione, ma attraverso la **lista di inizializzazione** abbiamo inizializzato le prime 4 celle dell'array. Ma allora ci possiamo chiedere: *possiamo non specificare la dimensione*? La risposta è dipende. Nel caso di array2 in riga 5, il compilatore riesce a dedurre la dimensione facendo una semplice divisione tra la dimensione totale dell'array e la dimensione del tipo utilizzato, un'operazione del tipo:

```
0 int n_array2= sizeof(array2)/sizeof(int); //ricavo la dimensione
```

Allocazione dinamica

Abbiamo visto nella sezione precedente, che l'allocazione nello stack, molto semplicemente, avviene in maniera quasi **automatica**. Le uniche cose che dobbiamo specificare sono la dimensione, e, l'eventuale inizializzazione. L'unica problematica che possiamo riscontrare nell'allocazione nello *stack* è che è una risorsa con **dimensioni limitate**, che potrebbe portare a uno *stack overflow*. Per questo ricorriamo all'**allocazione dinamica**:

```
0 #include<iostream>
1
2 int main() {
3
4 int* array3 = new int [100];
5
6 }
```

Codice 1.2: Allocazione dinamica di array

Nel caso di allocazione dinamica, per prima cosa notiamo che dobbiamo **allocare** o **deallocare manualmente** i dati. La parola chiave utilizzata per allocare una certa porzione di memoria, nel linguaggio C++, è la **new**. Nel caso dell'array inizializzato in riga 5, abbiamo dichiarato la *variabile puntatore*, che è necessaria nell'allocazione dinamica, poichè stiamo riservando un certo spazio, il quale indirizzo della prima cella è specificato nella variabile stessa. In tal caso, però, *sizeof* ci restituirà una dimensione che dipende esclusivamente dall'architettura del computer utilizzato, che rappresenta lo **spazio di indirizzamento** del nostro processore. Abbiamo però detto anche che dovremo **deallocare** la nostra zona di memoria allocata in precedenza. La parola chiave da utilizzare in questo caso, per il linguaggio C++, è la **delete**. Supponiamo quindi di inizializzare l'*array3* visto in precedenza con un ciclo for, e poi lo deallochiamo:

```
0
1     for(int i=0; i<100; i++){
2         array3[i] = i;
3         std::cout << array1[i] << std::endl;
4     }
5
6     //delete array3; //perdo il riferimento al primo elemento->
      memory leak
7     delete [] array3; //deallochiamo l'intero array
```

Codice 1.3: Deallocazione di array

La *delete* essenzialmente ha lo scopo di indicare al sistema operativo che quella **zona di memoria** è **riutilizzabile** ma che **non è affidabile**. Come possiamo notare dalle righe 6-7, però, dobbiamo fare molta attenzione a come deallochiamo un array. Nel caso di riga 6, stiamo semplicemente *perdendo il riferimento alla prima cella dell'array*, ma non stiamo deallocando l'intero blocco di memoria contenente l'array. In riga 7, utilizzando le parentesi quadre, deallochiamo correttamente l'intero array. Vediamo adesso di applicare delle tecniche di manipolazione di un array con un esercizio.

1.2 Esercizio sulla manipolazione di un array

Vediamo il testo di questo esercizio:

Esercizio 1. Scrivere un programma che sfrutta un metodo 'leggiElementi()' per inizializzare un array di double di dimensione non nota, chiedendo all'utente di inserire un valore per volta fino a quando non viene inserito il valore zero. Successivamente, stampare gli elementi mediante un altro metodo 'stampaElementi()'

Cerchiamo di sviluppare passo per passo il nostro esercizio:

- Partiamo dalla premessa che la **limitazione** principale dell'**array** è il fatto che devo avere una dimensione fissa. Di conseguenza dovrò prevedere una dimensione massima definendo una costante che ci indichi appunto tale dimensione massima:

```
0     #include<iostream>
1
2     #define MAX_N 100 //costante che indica la dimensione
      massima
3
4
```

- Passiamo alla funzione **leggiElementi**(). Ciò che dobbiamo fare adesso è definire la **firma della funzione**, e definire il **corpo della funzione**. Per quanto riguarda la firma; il tipo di ritorno sarà *void*, poichè dobbiamo semplicemente leggere gli elementi, e i parametri quindi che dobbiamo prevedere

sono un'array di double, e una variabile che faccia da contatore di posizioni dell'array valide. Tale variabile, in particolare, la **passeremo per riferimento**, in modo tale che una volta che incrementeremo tale contatore, staremo effettivamente incrementando la variabile e non una sua copia. Nel corpo della funzione invece chiederemo con lo standard input *cin* di immettere degli elementi, e, quando verrà digitato 0, questo verrà escluso dall'array finale:

```
0  void leggiElementi(double v[], int num) {
1
2
3    num = 0; //inizializzo a 0 num
4    double temp; // variabile dove metteremo il valore dato dall
       'utente
5
6    for(int i=0; i < MAX_N; i++) {
7      cout << "Inserire un elemento. Inserire 0 per terminare l'
       inserimento: ";
8      cin >> temp;
9      if(temp == 0) //se temp    0, usciamo dal ciclo
10       break;
11     else {// se temp    diverso da 0, aggiungiamo il valore
       all'array
12       v[i] = temp;
13       num++;
14     }
15   }
16 }
17
```

- Possiamo adesso occuparci adesso del metodo **StampaElementi**(). Per quanto riguarda la firma, anche in questo caso il tipo di ritorno sarà void, e come parametri passeremo l'array e la variabile *contatore*, che **passeremo per valore** poichè non ci serve modificarla, ma ci serve solo per scorrere l'array e stampare gli elementi. Il corpo della funzione, molto semplicemente, scorre l'array con un ciclo for stampando i suoi elementi:

```
0  void stampaElementi(double v[], int num) {
1    for(int i =0; i < num; i++) {
2      cout << "Elemento " << i <<"-esimo = " << v[i] << endl;
3    }
4    cout << endl;
5  }
6
```

- Nel **main** ci occupiamo di dichiarare l'array e la variabile contatore, e richiamiamo la funzione passando come parametri l'array e la variabile dichiarate:

```
0  int main() {
1
2    double array[MAX_N]; //array con dimensione fissa MAX_N
```

```
3     int num;
4
5     leggiElementi(array, num);
6     stampaElementi(array, num);
7
8     cout << array[67] << endl;
9  }
10
```

Capitolo 2

Ricorsione

In questo breve capitolo affronteremo la **ricorsione** e cercheremo di capire come funziona e i suoi vantaggi e svantaggi

2.1 Torri di Hanoi

Facciamo un gioco. Supponiamo di avere tre *paletti* all'interno del quale vi sono dei dischi disposti in ordine decrescente, che chiamiamo in ordine S,A,D, come possiamo vedere in figura:

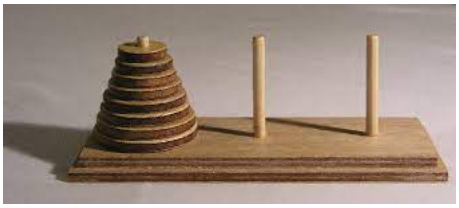


Figure 2.1: Torri di Hanoi

S rappresenterà la *sorgente*, ovvero il paletto dove ci saranno inizialmente i dischi; A è il paletto *ausiliario* e D sarà il nostro paletto di *destinazione*. Lo scopo del gioco è quello di spostare i dischi da S a D, in modo tale che si trovino in **ordine decrescente**, ovvero non ci può mai essere un disco più grande sopra e uno più piccolo sotto, e inoltre possiamo spostare anche i dischi in gruppo.

2.1.1 Soluzione al gioco

Per chi non è riuscito a risolvere il gioco o per gli impazienti vediamo la seguente immagine, cercando di spiegare come è stato risolto il gioco:

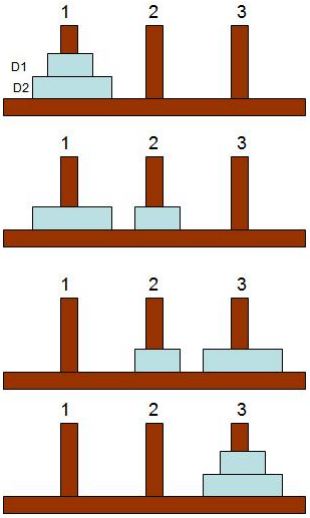


Figure 2.2: Soluzione del gioco

Nel caso della figura si utilizzano solo 2 dischi, ma questo processo può essere ripetuto con un qualsiasi numero di dischi. Quello che abbiamo fatto quindi è:

- Prendere il disco più piccolo da S, e inserirlo in A;
- Abbiamo preso il disco più grande e lo abbiamo inserito in D
- Infine il disco più piccolo che era in A, lo abbiamo impilato in D

Ciò che abbiamo fatto essenzialmente è **utilizzare la stessa strategia** di spostare $n-1$ dischi da un paletto all'altro, fino a che li abbiamo spostati tutti. Ma allora perchè abbiamo fatto questo gioco? Cosa c'entra la **ricorsione**? Il fatto di utilizzare la stessa strategia equivale a dire che abbiamo utilizzato un **algoritmo ricorsivo**, ovvero una **funzione che richiama se stessa**, basandoci su un **caso base** (1 disco da spostare in un paletto) e un **passo induttivo** ($n-1$ dischi da spostare).

2.2 Induzione e Ricorsione

Dopo esserci divertiti moltissimo con il gioco delle torri di Hanoi (LOL), cerchiamo di capire, come introdotto nella sezione precedente, perchè vi è una correlazione stretta fra **principio di induzione** e **ricorsione**. Facciamo un esempio con la funzione **successore di un numero**. Tale funzione è un *induzione*, infatti presenta, preso un $n \in \mathbb{N}$:

- Un **caso base** per il quale se $n = 0$, allora, se chiamiamo la funzione successore come $s(n)$, allora $s(n) = 1$
- Un **passo induttivo** per il quale se $n > 0$, allora $s(n) = 1 + s(n-1)$

Prestiamo attenzione a quanto scritto. Possiamo notare che tale funzione successore non è nient'altro che una **funzione ricorsiva**. Infatti l'unica informazione che abbiamo ce la da il caso base, ovvero che $s(0) = 1$. Nel passo induttivo costruiamo la funzione dal caso base, facendo in modo che la funzione **richiami se stessa**, fino ad arrivare al caso base. Facciamo appunto un esempio prendendo $n = 3$, e troviamo $s(3)$ applicando la funzione successore appena descritta:

$$\begin{aligned} s(3) &= \mathbf{1} + \mathbf{s(2)} \\ 1 + \mathbf{s(2)} &= 1 + \mathbf{1} + \mathbf{s(1)} \\ 1 + 1 + \mathbf{s(1)} &= 1 + 1 + \mathbf{1} + \mathbf{s(0)} \\ 1 + 1 + 1 + \mathbf{s(0)} &= 1 + 1 + 1 + \mathbf{1} = 4 \end{aligned} \tag{2.1}$$

Quindi abbiamo ricavato il successore di 3 in maniera induttiva attraverso una funzione ricorsiva che richiama se stessa. Una volta arrivati al caso base troviamo il risultato.

2.2.1 Esempio della funzione somma

Se ancora abbiamo qualche dubbio, facciamo un secondo esempio, in modo tale da capire bene come funzioni la ricorsione. Supponiamo di dover costruire *ricorsivamente* la **funzione somma** $somma(a, b)$ dati $a, b \in \mathbb{N}$:

- **Caso base**: il caso base in questo caso è che se $b = 0$, allora essendo *elemento neutro*, avremo che:
$$somma(a, 0) = a \tag{2.2}$$
- **Passo induttivo**: visto che vogliamo costruire questa funzione ricorsivamente, dobbiamo fare in modo che si richiami la funzione stessa. Per definizione la somma tra a e b è la somma di a con 1 per b volte. Di conseguenza, se vorremmo richiamare la funzione stessa, dovremo fare in modo di sommare 1 alla funzione che avrà $b - 1$ come secondo parametro, dunque:
$$b > 0 \quad 1 + somma(a, b - 1) \tag{2.3}$$

2.3 Implementazione della ricorsione

Veniamo adesso alla parte che ci interessa di più, ovvero implementare codice. Implementiamo, quindi, una *funzione ricorsiva* in C++. Prendiamo come funzione da implementare il **fattoriale** di un $n \in \mathbb{N}$. Avremo quindi che l'induzione sarà:

- **Caso Base**: per $n = 0$, il fattoriale di n sarà:
$$n! = 1 \tag{2.4}$$
- **Passo Induttivo**: per $n > 0$, il fattoriale di n sarà:
$$n! = n \cdot fatt(n - 1) \tag{2.5}$$

Il codice di questa funzione, molto semplicemente, sarà il seguente:

```
0 int fattoriale(int n) {
1     if(n==0) // CASO BASE
2         return 1;
3     return n*fattoriale(n-1); //PASSO INDUTTIVO
4 }
```

Come possiamo vedere, nella pratica, le funzioni ricorsive sono *molto semplici* da implementare e *molto leggibili*, ma hanno anche dei difetti che vedremo più nel dettaglio nella sezione successiva. Quindi, analizzando il codice, possiamo dire che:

- Per il *caso base*, abbiamo che se $n = 0$, allora torneremo 1;
- Altrimenti se $n > 0$, allora torneremo il prodotto di n per il fattoriale di $n-1$, ovvero richiamiamo la funzione stessa (**ricorsione**).

2.4 Ricorsione e Iterazione

In questo capitolo, fino ad adesso, abbiamo spiegato cosa significa funzione *ricorsiva*, come costruirla e come implementarla. Adesso cerchiamo di capire la differenza principale tra **funzioni ricorsive** e **funzioni iterative** sia in termini di *leggibilità del codice*, sia in termini di *velocità di esecuzione*.

2.4.1 Sequenza di Fibonacci

Prima di spiegare le differenze tra ricorsione e iterazione, cerchiamo di implementare la **sequenza di fibonacci**. Si tratta appunto di una sequenza in cui *ogni numero è la somma dei due numeri precedenti*. Per fare l'esempio la sequenza è del tipo:

$$fib = \{1, 1, 2, 3, 5, 8, 13, 21, ...\} \tag{2.6}$$

Implementazione Ricorsiva

Vediamo di implementare, prima, la sequenza in maniera ricorsiva. Per induzione, avremo:

- **Caso Base**: in questo caso abbiamo 2 casi base. Infatti sia per $n = 0$, che per $n = 1$, avremo che:
$$fib(n) = 1 \tag{2.7}$$
- **Passo Induttivo**: anche nel passo induttivo, poichè devo fare la somma tra 2 numeri, dovrò *richiamare la funzione 2 volte* e tale tipo di ricorsione è detta **ricorsione doppia**. In tal caso avremo che per $n \geq 2$:
$$fib(n) = fib(n-1) + fib(n-2) \tag{2.8}$$

Anche in questo caso l'implementazione è molto semplice:

```
0 int fibonacci(int n) {
1     if(n==0 || n==1) //CASO BASE
2         return 1;
3     return fibonacci(n-1)+fibonacci(n-2); //PASSO INDUTTIVO
4 }
```

Codice 2.1: Implementazione di funzione ricorsiva

Come spiegato in precedenza, quindi, avendo 2 casi base, utilizzeremo l'*or*, in modo tale che se $n = 0$ o $n = 1$, torneremo 1; altrimenti torneremo, tramite ricorsione doppia, $fib(n-1) + fib(n-2)$

Implementazione Iterativa

Adesso, cerchiamo di implementare sempre la **sequenza di Fibonacci**, in maniera **iterativa**. Prima di implementare la sequenza, però, dobbiamo sapere che la **ricorsione è sempre eliminabile**. Nella sequenza di Fibonacci, e anche negli esempi visti prima, avevamo inoltre una **ricorsione in coda**, ovvero la ricorsione è l'ultima cosa che facciamo. In questo caso particolare, possiamo sempre *sostituire la ricorsione con un iterazione*. Cerchiamo di capire in semplici passi come costruiamo iterativamente la sequenza:

- Per quanto riguarda il **caso base**, è analogo a quello precedente.
- Poichè abbiamo detto che dobbiamo fare un iterazione, utilizzeremo innanzitutto un **ciclo**. Noi utilizzeremo un ciclo *for*. Questo sommerà, come abbiamo detto in precedenza, a partire dalle sequenze $fib(2)$. Infatti i casi 0 e 1 li vediamo già con il caso base.
- All'interno del ciclo ci serviranno 2 variabili che ci tengano i valori da sommare, che possiamo chiamare sum_1 e sum_2, che contengono rispettivamente la somma degli $n-1$ e la somma degli $n-2$. Quindi, nell'inizializzazione delle 2 variabili avremo che $sum_1 = 1$ e $sum_2 = 1$, poichè infatti stiamo considerando l'iterazione a partire da $n = 2$ e, quindi, abbiamo che $sum(n-1) = 1$ e $sum(n-2) = 1$, ovvero i primi 2 numeri della sequenza, che corrispondono a 0 e 1, nella sequenza sono entrambi 1 (vedi equazione 2.6).
- Adesso ci chiediamo: come facciamo iterativamente a conservarci le 2 somme, e in particolare $sum(n-1)$, e poi sommarle? Analizziamo prima il problema:
  - Per $n = 2$, abbiamo che $sum(n-2) = 1$ e $sum(n-1) = 1$
  - Per $n = 3$, abbiamo che $sum(n-2) = 1$ e $sum(n-1) = 2$
  - Per $n = 4$, abbiamo che $sum(n-2) = 2$ e $sum(n-1) = 3$
  - ....

  Ciò che possiamo notare è che il $sum(n-2)$ del numero n è lo stesso del $sum(n-1)$ del numero $n-1$. Ci serve quindi una *variabile aggiuntiva* che mi contenga il $sum(n-2)$ in modo tale che posso **scambiare** il valore precedente di $sum(n-1)$ del numero $n-1$ con quello di $sum(n-2)$ del numero n.

- Alla fine, quindi, avremo che il valore di $fib(n)$ sarà dunque nella variabile sum_1, che è dunque quella che ritorneremo.

Vediamo quindi l'implementazione in modo tale da capire quanto scritto:

```
int fibonacci_iterativa(int n) {
    if(n == 0 || n == 1)// CASO BASE
        return 1;
    int sum_1=1, sum_2=1, x; //sum_1=sum(n-1) e sum_2=sum(n-2), x
    variabile per swap
    for(int i=2; i<=n; i++) {
        x = sum_2;
        sum_2 = sum_1;
        sum_1 = x+sum_2;
    }
    return sum_1;
}
```

2.4.2 Conclusioni su Ricorsione e Iterazione

Adesso la fatidica domanda è: *perchè abbiamo una versione* **iterativa** *e* **ricorsiva** *delle nostre funzioni?* Come la maggiorparte di noi potrebbe pensare, vedendo anche quante righe di codice abbiamo scritto, potremmo dire che sicuramente le *funzioni ricorsive* sono quelle più leggibili, facili da scrivere e veloci in termini di tempo di esecuzione; infatti ad esempio per scrivere la sequenza di Fibonacci in maniera iterativa abbiamo dovuto ragionare molto di più su come implementarla. In realtà è vero che le funzioni ricorsive sono leggibili e facili da scrivere, ma aggiungono un **sovraccarico imponente**, tanto che se noi provassimo a eseguire la funzione di fibonacci ricorsivamente con $n = 20000$, avremmo un tempo di esecuzione lunghissimo (se avete corraggio provateci). Questo perchè la ricorsione comporta diverse chiamate alla stessa funzione che sono molto dispendiose sia dal punto di vista del carico di lavoro del processore sia quello della memoria. E allora la domanda a seguire potrebbe essere: *perchè allora utilizziamo la ricorsione se ha questi svantaggi*? Se in realtà sappiamo che il numero di elementi che dobbiamo maneggiare è un numero molto limitato o per determinate *strutture dati*, la ricorsione può essere uno strumento molto utile ed efficace da utilizzare.

Capitolo 3

Complessità

In questo capitolo andremo a dare dei cenni sul concetto di **complessità**, analizzando i vari casi e alcune notazioni che riguardano il **comportamento asintotico**. Non ci dilungheremo per molto su tale argomento, poichè è abbastanza complesso e ne vedrete una narrazione può approfondita al corso di algoritmi.

3.1 Costo di un Algoritmo

Quando parliamo di *complessità*, ciò di cui parliamo riguarda appunto il **costo di un algoritmo**, ovvero il *prezzo* che devo pagare per ottenere l'output dall'algoritmo stesso. In tal caso il **costo dipende dall'input**. In particolare, se consideriamo un codice del tipo:

```
0 if(x == 0){
1     ...
2 }
```

Nel caso di tale codice, possiamo considerare 2 costi:

- Il **costo associato al controllo**, che viene chiamato **guardia**. Nel caso del codice 3.1, il costo associato al controllo riguarda la riga 0.
- Il **costo associato al corpo**. Ovviamente tale costo si riferisce al corpo della funzione stessa. Nel caso del codice 3.1, riguarda ciò che viene scritto all'interno delle graffe.

Quello che potremo sicuramente dire è che solitamente il costo del corpo è sempre quello più preponderante. Ma, quindi, cos'è il **costo computazionale** o **complessità**? Lo possiamo rappresentare in tal modo:

$$\sum_{i=0}^{n} c_i \tag{3.1}$$

ovvero la **sommatoria dei costi delle istruzioni**. Una considerazione molto importante da fare a riguardo è la seguente: anche se tendenzialmente si pensa il contrario, il *numero di istruzioni* è trascurabile rispetto a *quante volte le devo eseguire*. Se immaginiamo infatti di dover eseguire 1000 istruzioni una volta, oppure 100000 volte 2 istruzioni, capiamo che il secondo caso è sicuramente quello più preponderante dal punto di vista del costo computazionale. Quindi la cosa importante da comprendere è che il costo computazionale dipende in gran parte dalla dimensione N dell'input.

3.1.1 Caso Migliore, Peggiore e Medio

Supponiamo adesso di avere i seguenti array da **ordinare in maniera crescente**:

- Il primo array è formato da:

$$Arr_1 = \{0, 1, 2, 3, 4, 5, 6\} \tag{3.2}$$

- Il secondo array è formato da:

$$Arr_2 = \{6, 5, 4, 3, 2, 1, 0\} \tag{3.3}$$

Quello che possiamo notare subito è che il tempo di esecuzione per ordinare il primo array è sicuramente minore rispetto al secondo. In questo caso diremo che:

- Il primo array rappresenta il **caso migliore**, ovvero quello che viene eseguito **più velocemente**, infatti vediamo che l'array è già ordinato in maniera crescente
- Il secondo array rappresenta il **caso peggiore**, ovvero quello che viene eseguito **più lentamente**, infatti in tal caso dovremmo riordinare l'intero array.

Ciò che possiamo inoltre dire è che tra il *caso migliore* e il *caso peggiore*, vi sono una serie di **casi medi**, che non rappresentano gli estremi. E inoltre il caso **dipende dall'input e dalla natura del problema**: se infatti nell'esempio di prima avessimo dovuto ordinare in maniera decrescente l'array, sicuramente il secondo sarebbe stato il caso migliore e il primo il caso peggiore. L'esistenza dei vari casi inoltre vale per *qualsiasi algoritmo*. Da ciò possiamo fare le nostre deduzioni sulla **complessità temporale** di un algoritmo.

3.1.2 Complessità Temporale

Quella che abbiamo analizzato nell'esempio della sezione precedente, possiamo chiamarla anche **complessità temporale**, che è la **complessità riguardante il tempo di esecuzione di un algoritmo**. Sulla complessità temporale possiamo fare 2 assunzioni:

- **Assunzione 1**: La Complessità Temporale:

  - E' **proporzionale** al **numero di istruzioni** eseguite, intese complessivamente (ovvero per l'intero programma)
  - E'**indipendente dall'hardware considerato**

  Quindi, quando parliamo di complessità temporale, consideriamo quanto ci mette il codice a essere eseguito *dato un input*, indipendentemente dall'hardware. Ciò ci permette di fare la nostra seconda assunzione.

- **Assunzione 2**: La complessità temporale:
  - **Dipende** dalla **dimensione dell'input**
  - Valutiamo il suo **comportamento asintotico**

  Ciò significa che vogliamo valutare il comportamento di un algoritmo per **dimensioni che tendono ad infinito**, infatti per N piccoli le valutazioni potrebbero non valere o essere insignificanti.

Consideriamo tale codice:

```
0 for(int i=0; i<N; i++){
1     swap(i,i+1);
2     swap(i,i+2);
3     v[i]=-4;
4 }
```

In base a quanto detto ci interessa molto sapere il costo dell'istruzione in riga 3, ovvero la valutazione dipenderà dal fatto che stiamo usando l'array e dalla *dimensione N* di tale array. Non valutiamo dunque l'algoritmo in sè, ma valutiamo le **dimensioni delle strutture dati** che utilizziamo.

3.2 Notazione Asintotica

Abbiamo detto che per valutare la complessità di un algoritmo ci interessa sapere il suo **comportamento asintotico**, in modo tale da stimare quanto aumenta il tempo al crescere dell'input. In particolare esistono 3 tipi principale di notazione asintotica:

- Notazione O
- Notazione Ω
- Notazione Θ

Andiamo a capire, quindi, cosa effettivamente sono, e perchè ci servono per descrivere il comportamento asintotico di un programma.

3.2.1 Notazione O

La **notazione** O definisce il **limite asintotico superiore** e si applica a delle funzioni f con un limite asintotico superiore costituito dalla funzione $g(n)$. Tale notazione è descritta nel seguente modo:

$$O(g(n)) = \{f(n) : \exists c, n_0 | \ 0 \leq f(n) \leq c \cdot g(n) \ \forall n \geq n_0\} \tag{3.4}$$

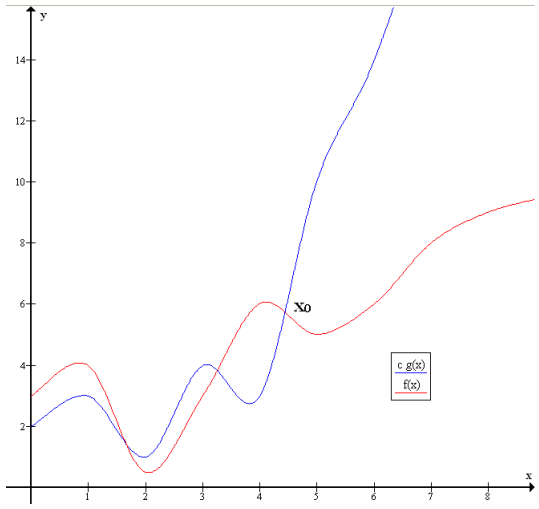


Figure 3.1: Notazione O

Quindi, per cercare di spiegare in modo semplice tale notazione, potremmo dire che $O(g(n))$ è l'insieme di funzioni $f(n)$ tali che esistano le *costanti* c, n_0, tale che $f(n)$ sia sempre maggiore o uguale a 0 e minore o uguale $cg(n)$ per ogni $n \geq n_0$. L'importante è quindi che esista un valore di c che rende vera l'affermazione. Inoltre visto che abbiamo stabilito che $O(g(n))$ è l'insieme delle funzioni $f(n)$ posso stabilire una **relazione** tra essi scrivendo:

$$f(n) = O(g(n)) \tag{3.5}$$

dove il simbolo '$=$' non significa '*uguale*', ma significa '**è**': di conseguenza leggeremo '$f(n)$ *è* $O(g(n))$'. Quando valuteremo i nostri algoritmi tendenzialmente valuteremo la *notazione* O dando quindi un *limite superiore* al *caso peggiore, medio e migliore*. Quindi se per esempio abbiamo un ciclo del tipo:

```
0 while(i<n){
1     if(i%2 == 0)
2         i++;
3     i++;
4 }
```

In tal caso il numero massimo di passi che faremo sarà al più n (limite superiore), quindi la complessità per tale ciclo sarà dell'ordine di $O(n)$. Inoltre possiamo dire che la notazione $O()$ suddivide in *classi di equivalenza* gli algoritmi, ponendo nella stessa classe gli algoritmi con lo stesso ordine di grandezza. Per cui avremo algoritmi di complessità:

- **costante**: $1, ...$
- **lineare**: n
- **logaritmica**: $logn$
- **polinomiale**: $n \log n, n^2, n^3 ...$
- **esponenziale**
- ecc.

In definitiva possiamo dire che se un algoritmo A prende al massimo tempo $t(n)$, allora $O(t(n))$ è un limite superiore. Possiamo vedere tale immagine come un spiegazione di tale concetto:

log n	⇒	complessità logaritmica	O(log n)
c1*n+c2	⇒	complessità lineare	O(n)
n^2+n	⇒	complessità quadratica	O(n^2)
n^k +c3*n	⇒	complessità polinomiale	O(n^k)
2^n+n	⇒	complessità esponenziale	O(k^n)

Figure 3.2: Classi di Equivalenza nella notazione O()

3.2.2 Notazione Ω

La **notazione** Ω definisce il **limite asintotico inferiore** ed è descritta nel seguente modo:

$$\Omega(g(n)) = \{f(n) : \exists c, n_0 | \ 0 \leq c \cdot g(n) \leq f(n) \ \forall n \geq n_0\} \tag{3.6}$$

Inoltre stabiliamo la **relazione** tra $f(n)$ e $\Omega(g(n))$ come facevamo per la notazione O:

$$f(n) = \Omega(g(n)) \tag{3.7}$$

dove il simbolo '$=$' significa '**è**'. In generale possiamo dire che dato un algoritmo A che prende tempo di almeno $t(n)$ per qualsiasi soluzione dell'algoritmo stesso, allora $\Omega(t(n))$ è un limite asintotico inferiore

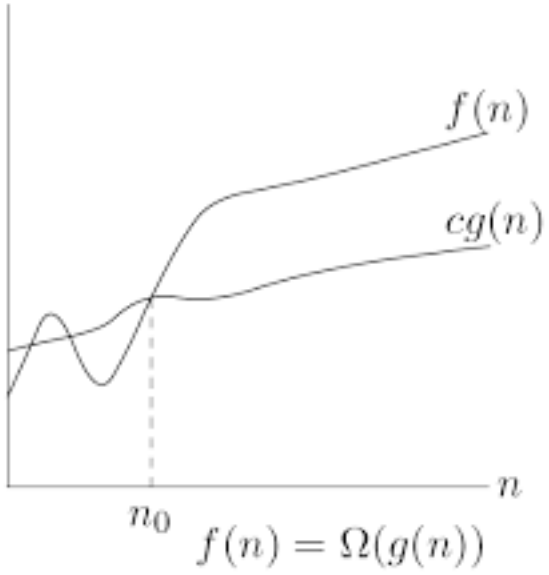


Figure 3.3: Notazione Ω

3.2.3 Notazione Θ

La notazione Θ definisce il **limite asintotico stretto** ed è descritta in tal modo:

$$\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0 | \ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\} \tag{3.8}$$

In questo caso avremo che le $f(n)$, a partire da n_0, sarà sempre compresa tra $c_1 g(n)$ e $c_2 g(n)$. Si definisce quindi sia il limite inferiore che quello superiore. In tal caso, poichè limite asintotico superiore e inferiore essenzialmente "coincidono", possiamo dire che se esiste un problema con complessità $\Omega(t(n))$ che viene risolto con un algoritmo di complessità $O(t(n))$, allora tale caso è ottimale e il problema ha complessità $\Theta(t(n))$

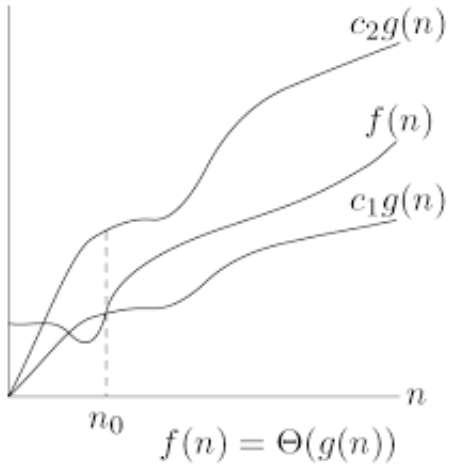


Figure 3.4: Notazione Θ

In questo caso possiamo dire che:

$$f(n) = \Theta(g(n)) \iff f(n) = \Omega(g(n)) \; e \; f(n) = O(g(n)) \tag{3.9}$$

3.2.4 Funzioni notevoli

Vediamo adesso una serie di **funzioni notevoli**, ovvero funzioni che sono quelle più tipiche che possiamo incontrare nell'analisi della complessità di un algoritmo:

- **Funzione costante**: è la funzione del tipo $f(n) = c$, dove c è appunto la costante, ed è rappresentata graficamente nel seguente modo:

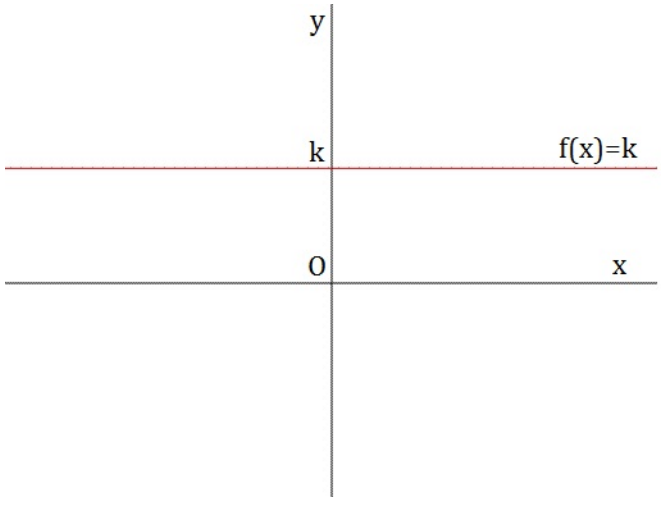


Figure 3.5: Funzione Costante

- **Funzione logaritmica**: è la funzione del tipo $f(n) = \log_b n$, con $b > 1$, ed è rappresentata nel seguente modo:

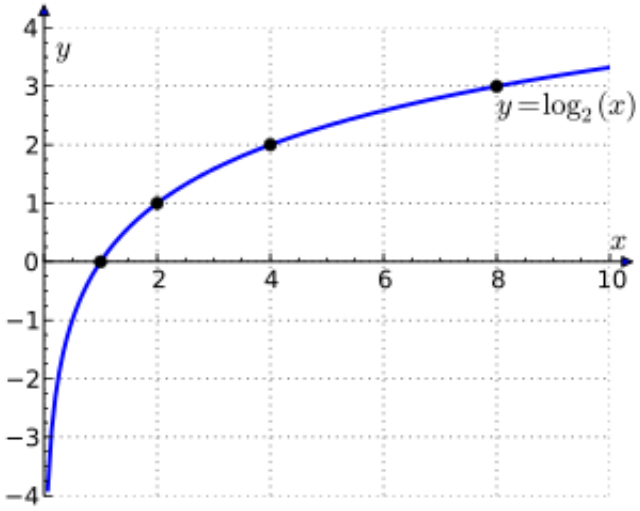


Figure 3.6: Funzione Logaritmica

- **Funzione Lineare**: è la funzione del tipo $f(n) = a \cdot n$, ed è rappresentata nel seguente modo:

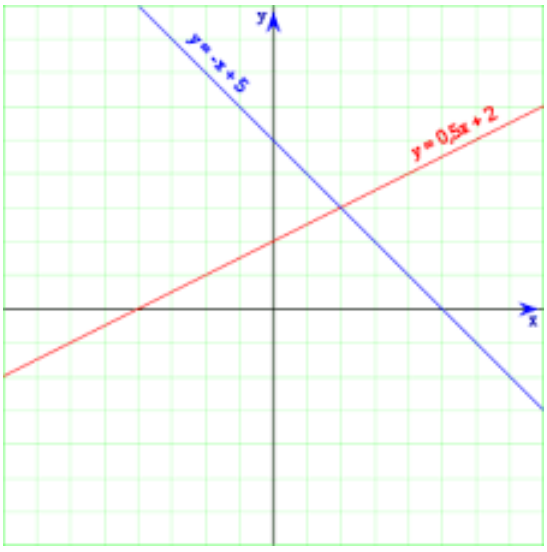


Figure 3.7: Funzione Lineare

- **Funzione Polinomiale**: è la funzione del tipo $f(n) = n^a$, con $a > 1$, ed è rappresentata nel seguente modo:

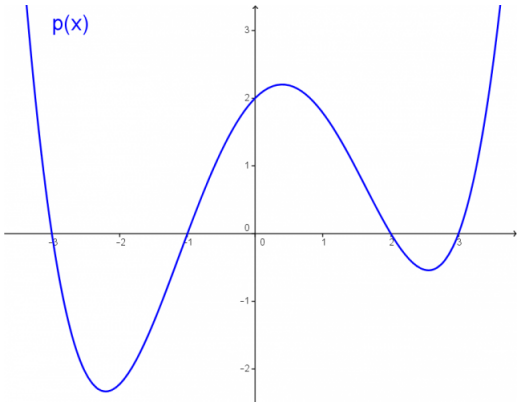


Figure 3.8: Funzione Polinomiale

- **Funzione Esponenziale**: è la funzione del tipo $f(n) = a^n$, con $a > 1$, ed è rappresentata nel seguente modo:

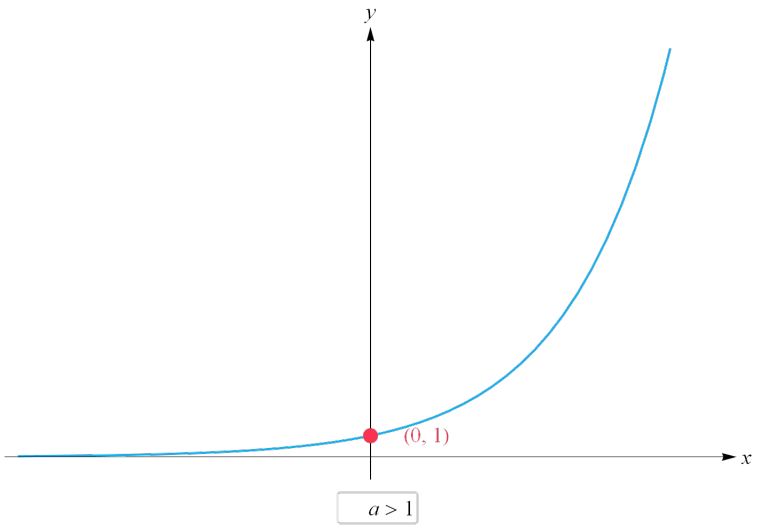


Figure 3.9: Funzione Esponenziale

3.3 Analisi di algoritmi

Vediamo adesso di applicare le notazioni asintotiche viste nel calcolo della complessità di un algoritmo semplice, cercando di tralasciare le varie dimostrazioni

riguardanti il calcolo della complessità a un corso di algoritmi. In generale possiamo dire che:

- $O(1)$ è la complessità per funzioni o blocchi di istruzioni che non contengono cicli, ricorsione o chiamate ad altre funzioni.
- $O(n)$ è la complessità di un ciclo quando le sue variabili sono aumentate di una quantità costante ($i+=c$).
- $O(n^c)$ è la complessità per cicli annidati in base al numero di volte vengono eseguite le istruzioni
- $O(\log n)$ è la complessità di un ciclo le cui variabili sono incrementate o decrementate moltiplicando o dividendo per una costante ($i*=c$)

Vediamo, ad esempio, il seguente programma:

```
0 for(int i=1; i<=n; i*=c){
1     for(int j=n/2; i<=n; i++){
2         for(int z=1; z<=n; z=2*j){
3             O(1)
4         }
5     }
6 }
```

Codice 3.1: Costo associato a un algoritmo

Analizziamo, quindi la complessità di questo piccolo programma con 3 for annidati:

- Iniziamo calcolando il costo associato al controllo in riga 0: in questo caso la domanda da fare è quanti passi dovro fare in questo ciclo? Poichè la variabile i viene incrementata ad ogni passata di $i \cdot c$ avremo $\log n$ passi (vedremo dopo perchè), allora avremo che il suo *comportamento asintotico* e dunque la sua *complessità computazionale* sarà di $\mathbf{O}(\log \mathbf{n})$
- Nel ciclo for in riga 1, a differenza di come abbiamo visto per la riga 0, abbiamo $\frac{n}{2}$ passi. Visto che però, a livello asintotico, non cambia la complessità tra $\frac{n}{2}$ e n poichè tende ad infinito, allora la complessità computazionale di tale ciclo sarà di $\mathbf{O}(\mathbf{n})$
- Nel ciclo for in riga 2, abbiamo n passi. In tal caso vediamo che la variabile z viene incrementata ad ogni passata di $2 \cdot j$. Cerchiamo di capire quanti passi farà tale ciclo: Supponiamo che $64 < n < 128$. Se immaginassimo la linea dei numeri naturali, avremmo che i passi sarebbero i seguenti: $z = 1 \rightarrow z = 2 \rightarrow z = 4 \rightarrow z = 8 \rightarrow z = 16 \rightarrow z = 32 \rightarrow z = 64$, quindi se contiamo sono 6 passi. Tali passi, in realtà, non sono altro che il risultato di $\log_2 64 = 6$. Ciò che quindi possiamo dire è che quando la variabile ha un incremento del tipo $i \cdot c$, dove c è una costante, la sua complessità sarà di $\mathbf{O}(\log \mathbf{n})$.
- Per calcolare la complessità complessiva del programma, che abbiamo supposto che abbia all'interno del ciclo più interno una complessità $O(1)$, ovvero costante, dovremo moltiplicare le complessità dei vari cicli, ovvero $n \cdot \log n \cdot \log n \rightarrow \mathbf{O}(\mathbf{n} \log \mathbf{n^2})$

3.3.1 Esempio

Un esercizio molto banale è per esempio quello di trovare la complessità di una funzione $U(n) = \gamma(n) + S(n)$, nel quale:

- $\Gamma(n) = n^5 + n^4 + n$: in questo caso la complessità asintotica è di $O(n^5)$
- $S(n) = \log(n) + \frac{n}{2} + \frac{n^2}{4}$: in questo caso la complessità è dell'ordine di $O(n^2)$
- In conclusione calcolando il totale avremo che la complessità finale sarà di circa $U(n) = O(n^5) + O(n^2) = O(n^5)$

3.4 Ricorrenze

Abbiamo parlato nel capitolo precedente della **ricorsione**, e abbiamo notato che poteva avere qualche *contro*, soprattutto per tutte le chiamate che devono essere fatte e quindi anche dal punto di vista dell'ottimizzazione. In realtà gli algoritmi ricorsivi possono avere una grande potenza, se applicate secondo il paradigma **divide et impera**. Vedremo anche nelle successive sezioni come si calcola la *complessità di una funzione ricorsiva* attraverso le **relazioni di ricorrenza**.

3.4.1 Divide et Impera

Andiamo, quindi, a descrivere il paradigma **divide et impera**, che si sviluppa nei seguenti 3 passaggi:

1. **Divide**: dividere il problema in **sottoproblemi**, ovvero istanze più piccole del problema stesso, riducendo quindi anche l'input.
2. **Impera**: risolvere i sottoproblemi *ricorsivamente* fino ad arrivare alla risoluzione del problema minimo, ovvero il **caso base**
3. **Combina**: combinare le soluzioni in modo tale da ricostruire la soluzione al problema originale

Seguendo tale paradigma, vedremo quanto possono essere potenti degli algoritmi ricorsivi.

3.4.2 Sommatorie

Abbiamo visto fino ad adesso la complessità di funzioni che utilizzavano al loro interno delle iterazioni, ovvero dei cicli. Vediamo adesso come calcolare la complessità di una ricorsione utilizzando una o più relazioni di ricorrenza. Prima, però, cerchiamo di ripassare il concetto di **sommatoria**. Se infatti all'interno delle iterazioni ripetiamo lo stesso corpo n volte, dovremo sommare i tempi di esecuzione, o il costo computazionale di ogni ciclo, e, per questo, ci vengono in aiuto le sommatorie. La sommatoria si rappresenta come:

$$\sum_{i=1}^{n} i \tag{3.10}$$

ovvero una somma di una variabile per n volte. La sommatoria, inoltre, gode della **proprietà di linearità** per cui:

$$\sum_{k=1}^{n}(c \cdot a_k + b_k) = \sum_{k=1}^{n} c \cdot a_k + \sum_{k=1}^{n} b_k = c \cdot \sum_{k=1}^{n} a_k + \sum_{k=1}^{n} b_k \tag{3.11}$$

cioè comunque prese una delle 3 uguaglianze otterremo lo stesso risultato. Inoltre la peculiarità di tale proprietà è che si applica anche a sommatorie con termini asintotici:

$$\sum_{k=1}^{n} \Theta(a_k) = \Theta(\sum_{k=1}^{n} a_k) \tag{3.12}$$

Serie

Una volta capito il concetto di *sommatoria*, possiamo introdurre il concetto di **serie**, che non sono altro che *particolari sommatorie*. Esse sono essenzialmente di 3 tipi:

- **Serie Aritmetiche**: La sommatoria $\sum_{k=1}^{n} k = 1 + 2 + 3 + .. + n$ è una *serie aritmetica*. Si dimostra per induzione che la serie aritmetica è uguale a:

  $$\sum_{k=1}^{n} k = \frac{1}{2} n(n+1) \tag{3.13}$$

  Possiamo dire quindi che $\sum_{k=1}^{n} k^2, \sum_{k=1}^{n} k^3$... sono anch'esse serie aritmetiche. La serie aritmetica $\frac{1}{2}n(n+1)$, inoltre, corrisponde ad una complessità dell'ordine di $\Theta(n^2)$. Vi sono però anche le serie corrispondenti a ordini di grandezza di $\Theta(n^3)$ o $\Theta(n^4)$.

- **Serie Geometriche**: La sommatoria $\sum_{k=0}^{n} x^k = 1 + x^2 + x^3 + .. + x^n$ è una *serie geometrica*. Si dimostra per induzione che la serie geometrica è uguale a:

  $$\sum_{k=1}^{n} x^k = \frac{x^{n+1} - 1}{x - 1} \tag{3.14}$$

- **Serie Armoniche**: La sommatoria $\sum_{k=0}^{n} \frac{1}{k} = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + .. + \frac{1}{n}$ è una *serie armonica*. Si dimostra per induzione che la serie armonica è uguale a:

  $$\sum_{k=0}^{n} \frac{1}{k} = \ln n + O(1) \tag{3.15}$$

  dove $O(1)$ rappresenta un tempo costante.

Per calcolare le serie si utilizza l'induzione. Dimostriamo per esempio che $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$:

- **Caso Base** ($k = 1$):

  $$\frac{1(1+1)}{2} = \frac{2}{2} = 1 \tag{3.16}$$

  quindi per il caso base è verificata la formula

- **Passo Induttivo**: Supponiamo che per $k = n$ valga la formula, ovvero è risolta. Dimostriamo che la soluzione per $n + 1$ assume la stessa forma dell'originale:

$$\begin{aligned}\sum_{k=1}^{n+1} k &= \sum_{k=1}^{n} k + (n+1) = \\ &= \frac{n(n+1)}{2} + n + 1 = \\ &= (n+1)(\frac{1}{2}n + 1) = \\ &= \mathbf{\frac{1}{2}(n+1)(n+2)}\end{aligned} \tag{3.17}$$

  Quindi abbiamo usato l'ipotesi induttiva per cui la formula $\frac{n(n+1)}{2}$ valeva per n, e la abbiamo utilizzata per dimostrare che vale per $n+1$. Infatti possiamo vedere che la soluzione per $n+1$ assume la stessa forma dell'originale, nel quale $n + 1$ rappresenta la n dell'originale, e $n + 2$ rappresenta $n + 1$ dell'originale

3.4.3 Ricorrenze Fondamentali

Vediamo adesso una serie di **equazioni di ricorrenza**, utilizzate per calcolare il costo computazionale di funzioni ricorsive:

- $C_N = C_{N-1} + N$, per $N \geq 2, C_1 = 1$: Questa formula viene utilizzata per programmi ricorsivi che ciclano sull'input *eliminando* un elemento per volta. In questo caso C_N, che rappresenta il costo totale, è circa $\frac{N^2}{2}$. Si dimostra per induzione che C_N è la *serie aritmetica* $\sum_{k=1}^{N} k$.
- $C_N = C_{\frac{N}{2}} + 1$, per $N \geq 2, C_1 = 1$. Questa formula viene utilizzata per programmi ricorsivi che dimezza l'input ad ogni passo. In questo caso C_N è circa $\ln(N)$
- $C_N = C_{\frac{N}{2}} + N$, per $N \geq 2, C_1 = 0$. Questa formula viene utilizzata per programmi ricorsivi che dimezzano l'input ed esaminano, eventualmente, ogni elemento ad ogni passo. In questo caso C_N è circa $2N$.
- $C_N = 2C_{\frac{N}{2}} + N$, per $N \geq 2, C_1 = 0$. Questa formula si usa tipicamente per quei programmi dove suddividiamo l'input a metà e ne analizziamo entrambe le metà. In questo caso C_N è circa $N \log(N)$.
- $C_N = 2C_{\frac{N}{2}} + 1$, per $N \geq 2, C_1 = 1$. Questa formula si usa tipicamente per quei programmi che dimezzano l'input ed eseguono una quantità di lavoro costante $(+1)$ per entrambe le metà. In questo caso C_N è circa $2N$.

Capitolo 4

Algoritmi di Ricerca e Ordinamento

Introduciamo in tale capitolo una classe fondamentale di algoritmi, ovvero quelli che riguardano **ricerca** e **ordinamento**. In particolare vedremo un tipo di ricerca binaria, e, algoritmi di ordinamento di un'array.

4.1 Ricerca

Per iniziare a parlare di algoritmi di ricerca vi consiglio di prendere un mazzo di carte di qualsiasi tipo:

- Una volta preso il mazzo scegliete 10 carte di un seme e ordinatele. Il nostro obiettivo è ricercare 3 carte in modo **lineare**, ovvero si confronta ogni carta con la carta che vogliamo cercare.
- Contate quanti confronti fate per trovare ogni carta, partendo ovviamente dall'inizio. Alla fine fate una media di quanti confronti si fanno.
- Potrete notare infine che il numero di confronti è di media circa 5, ovvero $\frac{n}{2}$

Tale algoritmo che stiamo utilizzando si chiama **ricerca lineare**, ed ha una complessità di circa $O(n)$, ricordiamo infatti che a livello asintotico non vi è differenza fra $\frac{n}{2}$ ed n e inoltre possiamo dire, più in generale che al più vanno fatti n confronti in tale tipo di ricerca. Tradotto in codice, si tratta di una funzione del genere:

```
0 bool linearSearch(int array[], int n, int key) {
1     for(int i=0; i<n; i++) {
2         if(array[i]==key) {
3             return true; //Ritorno true se trovo l'elemento che
      stavo cercando
4         }
5     }
6     return false; //Ritorno false se non ho trovato l'elemento che
      stavo cercando
```

```
}
```

Codice 4.1: Ricerca Lineare

4.1.1 Ricerca Dicotomica o Binaria

Per ricercare un elemento all'interno dell'array, esiste un tipo di ricerca più *efficiente* rispetto alla ricerca lineare che abbiamo visto, che è la **ricerca dicotomica**. Quello che potreste fare questa volta è:

- Prendere le solite 10 carte di prima e riordinatele
- Una volta che sapete la carta da ricercare, sapendo che l'array di carte è ordinato, potete fare la seguente ricerca:
  - Suddividete il mazzo a metà e vedete se la carta ricercata è maggiore o uguale o minore della carta presente nel mezzo
  - Ripetete queste suddivisioni fino a che vi ritroverete con 2 carte per il quale sceglierete quella richiesta
- Nel frattempo che suddividete i vari sottoarray annotate il numero di confronti che fate.

Se tutto è andato bene dovreste aver fatto circa $\log_2 n = \log_2 10$ confronti. Come possiamo evidentemente notare il numero di confronti in questo caso è sicuramente minore, avendo una complessità di $O(\log n)$. Questo rende molto efficiente l'algoritmo di ricerca binaria. Analizziamo adesso l'implementazione in codice:

```
//ricerca dicotomica(binary search): dividiamo a meta l'array e
    cerchiamo un elemento
//supponiamo che l'array sia gia ordinato
bool binarySearch(int array[], int n, int key) {
    bool found = false;

    int start = 0;
    int end = n;

    //Utilizziamo un ciclo while perche non sappiamo il numero di
    iterazioni che dovremmo fare
    while(!found && (start != end)) {

        int midpoint=(start+end)/2;

        if (array[midpoint] == key)
            found = true;
        else if (key < array[midpoint]) {
            // considerare la meta inferiore
            // aggiornare end e midpoint
            end = midpoint;
            // midpoint = (end-start)/2;
        }
        else{
            // considerare la meta superiore
```

```
23              // aggiornare start e midpoint
24              start = midpoint + 1;
25              // midpoint = (end - start)/2;
26          }
27          // verificare ad ogni iterazione se key e >= dell'elemento
        in posizione n/2
28      }
29
30      return found;
31  }
```

Codice 4.2: Ricerca Dicotomica

Vediamo di comprendere come si sviluppa questo algoritmo:

- Inizializziamo innanzitutto la variabile booleana found a false, che verrà aggiornata una volta trovata la *key*, ovvero l'elemento richiesto. Inoltre inizializziamo gli indici di *start* ed *end* rispettivamente a 0 e n. Tali variabili individuano l'inizio e la fine dell'array che viene ogni volta suddiviso
- Utilizziamo un ciclo *while*, non sapendo il numero di iterazioni, e come condizioni imponiamo che fino a quando found è false e start è diverso da end, possiamo fare la passata del ciclo. Una volta entrati nel ciclo inizializziamo il midpoint, ovvero l'elemento della metà come media del nuovo eventuale start, se consideriamo la metà superiore, ovvero di destra, e del nuovo eventuale end, se consideriamo la metà inferiore.
- Se l'elemento presente in midpoint coincide con l'elemento ricercato aggiorniamo found a true. Altrimenti, se il nostro elemento ricercato è **minore** di quello di presente in midpoint, **aggiorniamo il nuovo indice di end** del nuovo array, poichè infatti se cosideriamo la parte di sinistra allora end sarà quello che nell'array precedente era il midpoint. Al contrario, se vediamo che l'elemento ricercato è **maggiore o uguale** all'elemento presente nel midpoint, in questo caso **sarà lo start a dover essere aggiornato** in quanto ci stiamo spostando nella metà a destra.

4.2 Algoritmi Iterativi di Ordinamento

Abbiamo visto degli algoritmi di *ricerca*. Abbiamo supposto, però, in entrambi gli algoritmi, che l'array che utilizzavamo per ricercare un elemento fosse già ordinato. Molto spesso, però, ci troveremo array casuali e di conseguenza disordinati, e che quindi sono da *ordinare*. Per questo esistono appositi **algoritmi di ordinamento**, più o meno efficienti, che hanno come scopo quello di **ordinare un array**. In questa sezione vediamo l'esempio di alcuni algoritmi di ordinamento iterativi e, nella prossima sezione, alcuni algoritmi di ordinamento ricorsivi.

4.2.1 Ordinamento per Scambio

L'algoritmo di ordinamento più semplice da analizzare e implementare è l'**ordinamento per scambio**. Prendete quindi il vostro mazzo di carte e mescolatelo:

- Una volta mescolato il mazzo, disponete le carte sul tavolo con il loro ordine.
- Fissate la prima carta e scambiate la carta quando ne trovate una minore
- Ripetete questa iterazione fino alla fine del vostro array di carte o fino a quando non è ordinato.

Quindi, molto semplicemente, questo algoritmo si basa nell'avere alla i-esima iterazione, i elementi ordinati. Questo avviene *scambiando la carta fissata* ogni volta che ne troviamo una minore. L'implementazione sarà quindi:

```
0  //Funzione swap per lo scambio dei valori
1  void swap(int& a, int& b) {
2      int temp = a;
3      a=b;
4      b=temp;
5  }
6
7  void ordinamentoScambio(int array[], int n) {
8      for(int i=0; i<n; i++) {
9          for(int j=i+1; j<n; j++) {
10             if(array[j]<array[i]) //Confrontiamo l'elemento j-esimo
       con l'elemento i-esimo fissato
11                 swap(array[i], array[j]); //Scambio
12         }
13     }
14 }
```

Codice 4.3: Ordinamento per Scambio

Come possiamo notare per questo algoritmo, e come noteremo per gli altri algoritmi di ordinamento, la complessità di tale algoritmo è di $O(n^2)$, ovvero una **complessità quadratica**.

4.2.2 Selection Sort

Il primo vero algoritmo che introduciamo, sicuramente più efficiente di quello per scambio, è il **selection sort**:

- Prendete come al solito le 10 carte e mescolatele
- Una volta mescolate, disponetele sul tavolo con il loro ordine.
- Fissate la prima carta, scorrete tutto l'array dalla seconda carta in poi e aggiornate l'**indice** dell'array ogni volta che trovate una carta minore.
- Alla fine *selezionate* la carta con indice dell'array che ha il valore minore e scambiatela. Ripetete queste operazioni per l'intero array di carte.

Come possiamo quindi notare, in questo caso non facciamo gli scambi ogni volta che troviamo una carta minore, ma fissiamo una carta, scorriamo tutto l'array dalla i+1-esima carta fino alla fine e **aggiorniamo l'indice** della carta ogni volta che ne troviamo una minore. Questo ragionamento si implementa col seguente codice:

```
0  //Funzione swap per lo scambio dei valori
1  void swap(int& a, int& b) {
2      int temp = a;
3      a=b;
4      b=temp;
5  }
6
7  void selectionSort(int array[], int n) {
8      int indexMin; //Variabile che contiene l'indice minore
9
10     for(int i=0; i<n; i++) {
11         indexMin = i;
12         for(int j=i+1; j<n; j++) {
13             if(array[j]<array[indexMin])
14                 indexMin = j; // indice dell'elemento piu piccolo
       dell'array non ordinato
15         }
16     swap(array[i], array[indexMin]); //Dopo aver aggiornato
       indexMin faccio lo scambio
17
18     }
19 }
```

Codice 4.4: Selection Sort

Analizzando il codice, vediamo che:

- La variabile che conterrà l'indice dell'array che ha il valore minore è indexMin
- Abbiamo 2 cicli: quello più esterno che inizializza IndexMin con i, e dopo aver fatto i vari aggiornamenti di IndexMin fa lo scambio tra il valore contenuto in posizione i e quello contenuto in indexMin.
- Il ciclo interno, che rappresenta il fulcro del selection sort, aggiorna indexMin con l'indice j-esimo, ogni volta che trova in posizione j un valore minore rispetto a quello di indexMin.

Come già accennato, notiamo che la complessità è dell'ordine di $O(n^2)$, ma in questo caso abbiamo meno chiamate della funzione swap, e quindi un vantaggio in termini di efficienza.

4.2.3 Insertion Sort

Vediamo adesso un algoritmo di ordinamento che è un pò più complesso dei precedenti, ovvero l'**insertion sort**. L'idea di base, riprendendo sempre il nostro mazzo di carte è la seguente:

- Mescolate le vostre carte e prendetele in mano
- Ordinate l'array fissando l'ultima carta e *inserendola* al posto giusto facendo scorrere le altre verso destre, un pò come quando giocate a poker.
- Ripetete questa azione fino a quando l'array non è ordinato

Abbiamo ad ogni iterazione quindi i posizioni ordinate e $n - i$ posizioni non ordinate. Il fulcro di questo algoritmo è quindi di prendere la $i + 1 - esima$ carta e attraverso una ricerca lineare o dicotomica, inserirla nella posizione corretta scorrendo le carte rimanenti verso destra. Vediamo quindi un'implementazione:

```
0  void insertionSort(int array[], int n) {
1      int temp,j;
2
3      for(int i=1; i<n; i++) {
4          temp = array[i];
5          for(j=i; j>0; j--) { //Controllo il sotto-array a sinistra
6              if(temp < array[j-1]) //Scorriamo tutte le carte
       maggiori della i-esima verso destra
7                  array[j] = array[j-1];
8              else
9                  break;
10         }
11         array[j]=temp;
12     }
13 }
```

Analizziamo tale implementazione: come possiamo vedere ci serviamo di 2 cicli for. Quello più esterno fissa l'elemento $i - esimo$ dell'array, e il ciclo for più interno che confronta l'$i - esimo$ elemento con il sottoarray di sinistra (infatti l'indice di partenza è 1). Ci serviamo di una variabile *temp*, che conterrà l'elemento i-esimo, e definiamo l'indice j fuori dal ciclo interno in quanto ci serve per assegnare correttamente l'elemento i-esimo. Il fulcro di questo algoritmo sta proprio nel ciclo interno, che confronta l'i-esimo elemento con gli elementi alla sua sinistra fino a quando questo è minore di questi ultimi, e nel frattempo con l'istruzione $array[j] = array[j-1]$ è come se facessimo scorrere verso destra la carta $j-1$ in quanto essa è maggiore dell'elemento i-esimo. Se avviene la condizione per il quale l'elemento i-esimo è maggiore del precedente, allora rompiamo il ciclo (*break*) poichè abbiamo trovato la posizione j-esima esatta nel quale inserire la carta. In questo caso, come avevamo visto in precedenza, abbiamo una *complessità quadratica* di $O(n^2)$

4.3 Algoritmi Ricorsivi di Ordinamento

Abbiamo descritto fino ad ora *algoritmi di ordinamento iterativi*. In realtà vi sono degli algoritmi di ordinamento ancora più efficienti, anche dal punto di vista della complessità, che però si basano sulla **ricorsione**. Infatti, come già detto nella sezione 3.4.1, se tali algoritmi ricorsivi vengono creati secondo la logica **divide et impera** sono molto potenti. Analizziamo il problema dell'*ordinamento* sotto questo aspetto quindi, ovvero cercando di seguire i 3 passi del paradigma divide et impera:

1. **Suddividere il problema in sottoproblemi**: nel caso dell'ordinamento si traduce nel frazionare l'array in parti sempre più piccole fino ad arrivare a sottoproblemi che si riconducono al caso base, ovvero un elemento.

2. **Risolvere i sottoproblemi**: poichè siamo arrivati al caso base, ovvero 1 elemento, nel caso dell'ordinamento non si deve fare nulla
3. **Combinare le soluzioni**: per la maggiorparte degli algoritmi di ordinamento ricorsivi, questo passo rappresenta la **logica** su cui si basa l'**algortimo**. Vedremo ad esempio il *mergesort*, che ha una procedura apposita per combinare le soluzioni

4.3.1 Merge Sort

Analizziamo quindi come nostro primo algoritmo di ordinamento ricorsivo il **merge sort**, che è uno di quegli algoritmi che basa la sua logica principalmente nel passo di *combinazione*. In questo caso la procedura di combinazione è realizzata tramite una **funzione merge**.

Procedura Merge

Addentriamoci quindi nel passo di combinazione, analizzando la procedura *merge*. Essenzialmente la funzione prende un array di dimensione n e lo suddivide in **2 sottoarray**, che sono già ordinati, e li **combina** nell'array più grande in modo tale che mantengano l'ordine esatto. Vediamo quindi inizialmente uno *pseudocodice* che ci aiuti a capire bene come si applichi tale procedura:

Algorithm 1 Algoritmo di Combinazione Merge

$Merge(A, p, q, r)$ ▷ Passiamo l'array A, l'indice di inizio p, l'indice di fine r e l'indice q tale che $p \leq q \leq r$
$n_1 = q - p + 1$ ▷ Dimensione sottoarray L di sinistra
$n_2 = r - q$ ▷ Dimensione sottoarray R di destra
$L \leftarrow A[p \rightarrow q]$ ▷ Creo l'array L
$R \leftarrow A[q \rightarrow r]$ ▷ Creo l'array R
$L[n_1 + 1] \leftarrow \infty$
$R[n_2 + 1] \leftarrow \infty$
while ($k \leq r$, $i \leq n_1$, $j \leq n_2$) **do** ▷ Combinazione degli elementi dei sottoarray nell'array principale A
    if ($L[i] \leq R[j]$) **then**
        $A[k] = L[i]$
        i++
    else
        $A[k] = R[j]$
        j++
    end if
    k++

Essenzialmente quindi quello che si fa nella procedura è:

- Prendere come parametri l'array principale A, il suo indice di inizio p, il suo indice di fine r, e il suo indice $p \leq q \leq r$.

- Creare 2 sottoarray, che sono già stati ordinati, uno che individua la parte sinistra di A, ovvero la parte che va dall'indice p a q, che viene chiamato L, e l'altro che individua la parte destra di A, ovvero che va dall'indice q a r, che viene chiamato R.

- Si individua la fine dell'array L ed R, inserendo $L[n_1 + 1] = \infty$, dove n_1 è l'indice di fine di L, e $R[n_2 + 1] = \infty$, dove n_2 è l'indice di fine di R.

- Infine si scorrono tutti e 3 gli array. Se l'elemento $i - esimo$ di L è minore o uguale dell'elemento $j - esimo$ di R, allora per l'ordinamento verrà inserito in posizione $k - esima$ di A e viene incrementata l'indice i dell'array L e l'indice k dell'array A, si ripete questo confronto fino a combinare tutti gli elementi dei sottoarray nell'array principale in maniera che siano ordinati.

Facciamo un *esempio* supponendo di avere il seguente array A di 4 elementi, che è una parte dell'array principale, supponendo di essere arrivati al passo di combinazione e quindi avendo gli elementi ordinati a 2 a 2:

...	4	6	3	5	...

La suddivisione nei 2 sottoarray sarà la seguente:

L

4	6

R

3	5

Dunque il ciclo della procedura merge per combinare gli elementi del sottoarray in maniera ordinata nell'array A si sviluppa in questo modo:

- Gli indici di inizio sono i seguenti $i = 0, j = 0, k = 0$.

- Alla **prima iterazione** si confronta $L[i] \rightarrow L[0]$ con $R[j] \rightarrow R[0]$ e, poichè abbiamo che $3 < 4$, quindi $R[j] < L[i]$, ad $A[k] \rightarrow A[0]$ viene assegnato il valore 3, e viene incrementata la $j \rightarrow j = 1$. L'array A sarà alla prima iterazione:

...	3	...	...	...	...

- Alla **seconda iterazione** si ripete la procedura precedente, ovvero si confronta $L[i] \rightarrow L[0]$ con $R[j] \rightarrow R[1]$ e, poichè abbiamo che $4 < 5$, quindi $L[i] < R[j]$, ad $A[k] \rightarrow A[1]$ viene assegnato il valore 4, e viene incrementata la $i \rightarrow i = 1$. L'array A sarà alla seconda iterazione:

...	3	4	...	...	...

- Alla **terza iterazione** si ripete il confronto confrontando $L[1]$ con $R[1]$, e poichè $5 < 6$, e quindi $R[1] < L[1]$, ad $A[k]$ viene assegnato il valore 5 e viene incremetata $j \rightarrow j = 2 = n_2 + 1 \rightarrow \infty$.L'array A sarà alla terza iterazione:

...	3	4	5	...	...

- Alla **quarta iterazione** si ripete il confronto confrontando $L[1]$ con $R[2]$, e poichè $6 < \infty$, e quindi $L[1] < R[1]$, ad $A[k]$ viene assegnato il valore 6 e viene incremetata $i \rightarrow i = 2 = n_1 + 1 \rightarrow \infty$.L'array A sarà alla quarta iterazione:

...	3	4	5	6	...

Quindi alla quarta iterazione l'array è ordinato

Analizzando l'esempio fatto possiamo dire che tale procedura, ovvero la procedura merge, **impiega tempo** n e, ciò inoltre vale sia nel caso peggiore che in quello migliore. In questo caso, quindi, definiamo la complessità in termini di:

$$\Theta(n) \tag{4.1}$$

infatti definiamo sia il limite superiore (caso peggiore: n passi), sia quello inferiore (caso migliore: n passi).

Procedura Ricorsiva

Veniamo adesso alla parte che mi permetta di effettuare l'ordinamento ricorsivamente. Vediamo, come prima, inizialmente lo pseudocodice che riesca a farci comprendere bene come funziona l'algoritmo:

Algorithm 2 Procedura Ricorsiva Merge Sort

$MergeSort(A, p, r)$ ▷ Passiamo ricorsivamente come parametri l'array, l'indice di inizio p e l'indice di fine r
if $(p < r)$ **then**
  $q = (p + r)/2$ ▷ Calcoliamo l'indice q per suddividere l'array principale in 2 sottoarray
  $MergeSort(A, p, q)$ ▷ Applichiamo la procedura merge sort fino a quando i sottoarray diventano di dimensione 1
  $MergeSort(A, q + 1, r)$
  $Merge(A, p, q, r)$
end if=0

Quindi il fulcro di tale algoritmo si sviluppa nel richiamare ricorsivamente la procedura merge sort fino a quando non arriviamo al caso base, quindi a 1 elemento. Una volta terminate tutte le chiamate ricorsive si procede con la combinazione, e quindi con il merge. Possiamo dire che, nonostante l'algoritmo di merge

sort sia molto efficiente dal punto di vista della complessità temporale, che è circa di $\theta(nlogn)$, dato che suddividiamo l'array in 2 e analizziamo entrambe le metà (sezione 3.4.3), come abbiamo già detto, in realtà è impattante dal punto di vista della **complessità spaziale**, infatti occupiamo molta memoria proprio perchè abbiamo bisogno degli array ausiliari di appoggio.

Implementazione in C++

Una volta spiegato teoricamente come funzione il Merge Sort, vediamo come implementarlo in codice:

```
0  //Procedura Merge
1  void merge(int A[], int p, int q, int r) {
2      int n1 = q-p+1;
3      int n2 = r-q;
4
5      int* L = new int[n1+1];
6      int* R = new int[n2+1];
7
8
9
10     for(int i=0; i<n1; i++) {
11         L[i] = A[p+i];
12     }
13
14     for(int j=0; j<n2; j++) {
15         R[j] = A[q+1+j]; //Partiamo con gli indici da 0, quindi
       dobbiamo sommare 1 a q
16     }
17
18     int i=0, j=0, k=p; //Riscrivo gli indici per lo scope limitato
       al ciclo for
19
20     //Elementi sentinella
21     L[n1] = INT_MAX;
22     R[n2] = INT_MAX;
23
24
25     for(k=p; k<=r; k++) {
26         if(L[i]<R[j]){
27             A[k]=L[i];
28             i++;
29         }else {
30             A[k]=R[j];
31             j++;
32         }
33     }
34
35
36     delete [] L;
37     delete [] R;
38 }
39
40
41 //Procedura Ricorsiva
42 void mergeSort(int A[], int p, int r) {
```

```
43      if(p<r) {
44          int q = (p+r)/2; //Divisione tra interi implementata come
        il floor -> In alternativa si pu  usare, includendo cmath, la
        funzione floor()
45          mergeSort(A,p,q);
46          mergeSort(A,q+1,r);
47          merge(A,p,q,r);
48      }
49  }
```

Codice 4.5: MergeSort

Come possiamo notare la procedura ricorsiva *mergesort* è identica a come l'avevamo teorizzata nello pseudocodice. Soffermandoci, invece, sulla procedura *merge*, è importante soffermarsi sulla questione degli **indici**. Infatti, per far si che l'array R sia correttamente assegnato, dobbiamo fare molta attenzione agli indici. In particolare se nello pseudocodice avevamo scritto $R \leftarrow A[q \rightarrow r]$, in realtà dobbiamo far partire l'array da $q+1$, altrimenti avremmo che l'elemento in posizione q verrebbe contato 2 volte. Agli *elementi sentinella* viene assegnato il valore INT_MAX in modo tale da riconoscere che siamo arrivati alla fine. Per il resto il codice si presenta in maniera quasi identica a come lo avevamo descritto in maniera teorica.

4.3.2 Quick Sort

Un altro algoritmo di ordinamento ricorsivo ancora più efficiente del merge sort, è il **quick sort**. Infatti alla base del quick sort non vi sono array di appoggio ma gli **elementi** vengono **ordinati sul posto**.

Procedura Partition

Se, quindi, nel merge sort il passo chiave era il passo di combinazione *merge*, in questo caso il passo chiave è la **suddivisione dell'array**. La procedura di suddivisione viene chiamata **partition** e si sviluppa nel seguente modo:

- Si prende un *elemento a caso*, tendenzialmente l'ultimo, a prescindere dal suo valore effettivo. Tale elemento viene chiamato **pivot**.
- Individuati gli *indici di inizio e fine* dell'array, che chiameremo rispettivamente p ed r, si inizializzano gli indici i, j tali che $i = p - 1$ e $j = p$
- Si fa **scorrere j** da p a r. Se l'elemento $A[j] \leq$ pivot allora faccio **avanzare i** e **scambio** $A[i]$ con $A[j]$.
- Una volta che abbiamo scorso tutto l'array, **scambio il pivot** con $A[i+1]$ e **restituisco** $i+1$, ovvero la posizione finale del pivot.

Quindi l'obbiettivo di tale procedura è quello di suddividere l'array in **partizioni** in modo tale che abbiamo la parte sinistra con gli elementi minori del pivot, e la parte destra con quelli maggiori. Infatti lo scambio possiamo notare solo nel caso in cui l'elemento $A[j] \leq$ pivot: questo porta facendo avanzare i e facendo lo scambio a creare le partizioni. Restituendo la posizione finale del pivot, riusciamo con la procedura ricorsiva a partizionare l'intero array in maniera che sia ordinato.

Procedura Ricorsiva

Vediamo adesso come viene strutturata la **procedura ricorsiva** nel quick sort. Poichè si basa sulla *partition*, quindi le chiamate si baseranno sulla posizione finale del pivot. Vediamo quindi un esempio in pseudocodice:

Algorithm 3 Procedura Ricorsiva QuickSort

$QUICKSORT(A, p, r)$
if $p < r$ **then**
  $q = PARTITION(A, p, r)$
  $QUICKSORT(A, p, q-1)$
  $QUICKSORT(A, q+1, r)$

Quindi in tal caso, scegliendo come pivot un elemento casuale dell'array, dopo aver richiamato la procedura partition, il valore dell'indice di tale pivot viene assegnato a q e si procede con le chiamate ricorsive a partire da tale indice.

Considerazioni sull'efficienza dell'algoritmo Abbiamo compreso che in tale algoritmo il *passo chiave* si basa su come viene suddiviso l'array secondo la procedura partition. Nel caso in cui l'elemento del pivot che viene scelto coincida con il minimo o il massimo allora ci troveremmo nel **caso peggiore**, con una complessità dell'ordine di $O(n^2)$, poichè infatti ci ritroveremmo una partizione di lunghezza $n-1$ e una di lunghezza 0 e, di conseguenza dovremo partizionare l'array molte volte. Quindi, possiamo capire da tali considerazioni fatte, che quello che conta nel quick sort è il **caso medio**: infatti su array molto grandi è molto più probabile prendere un elemento che non sia nè il minimo nè il massimo che viceversa. In tal caso l'algoritmo è molto efficiente, sia perchè dal punto di vista della *complessità spaziale* non occupa molto visto che non utilizza array di appoggio, come nel mergesort, sia dal punto di vista della complessità temporale, che, nel caso medio, è dell'ordine di:

$$O(n \log n) \tag{4.2}$$

Implementazione in C++

Seguendo pedissequamente quanto detto nella teoria, implementiamo il quick sort in C++:

```
0
1  //Funzione swap per lo scambio di elementi
2  void swap(int& a, int& b) {
3      int temp = a;
4      a=b;
5      b=temp;
6  }
7
8  //Procedura Partition
9  int partition(int array[], int p, int r) {
10     int i = p-1;
11     int j = p;
```

```
12
13      int& pivot = array[r];
14      while(j<r) {
15          if(array[j] <= pivot) {
16              i++;
17              swap(array[i],array[j]);
18          }
19          j++;
20      }
21      swap(pivot, array[i+1]);
22      return i+1;
23  }
24
25  //Procedura QuickSort
26  void quickSort(int array[], int p, int r) {
27      if(p<r) {
28          int q = partition(array,p,r);
29          quickSort(array,p,q-1);
30          quickSort(array,q+1,r);
31      }
32  }
```

Codice 4.6: QuickSort

Parte II

Programmazione a Oggetti

Capitolo 5

Programmazione orientata agli Oggetti

Affrontiamo in questo capitolo, molto brevemente, un argomento fondamentale per noi programmatori, che è la **programmazione orientata a oggetti**. Ad oggi, infatti, si tratta del *paradigma di programmazione* più utilizzato nel mondo del lavoro. Vedremo quindi i concetti di base che riguardano la programmazione a oggetti e le peculiarità della programmazione a oggetti.

5.1 Principi di Base

Nella *programmazione a oggetti* abbiamo 4 **principi di base**:

1. **Astrazione**: consiste nel *generalizzare i concetti da implementare* in modo tale da avere **interfacce** comuni a tutti agli oggetti di quel tipo, in modo tale che tali interfacce contengano le funzionalità essenziali. La capacità di astrarre, comunque, è fondamentale per sistemi complessi, nel quale possiamo avere centinaia di classi.

2. **Incapsulamento** (*information hiding*): è una proprietà degli oggetti per il quale esistono dei **modificatori di accesso** con il quale si può decidere quali informazioni rendere **pubbliche** (*public*) o **private** (*private*). L'incapsulamento delle informazioni, dunque, è alla base della *progettazione corretta delle classi*.

3. **Ereditarietà**: consiste in un **ulteriore livello di astrazione**, in modo tale da avere un livello più alto rispetto a un interfaccia comune (*classe base*), grazie al quale tutte le proprietà della classe base vengano estese alle *classi derivate*, e inoltre, in quest'ultime, abbiamo la possibilità di aggiungere nuove specifiche.

4. **Polimorfismo**: consiste nell'avere funzioni che hanno la stessa *firma* (o *signature*) ma **assumono diversi comportamenti** a seconda del chiamante.

5.2 Classi

Nella programmazione a oggetti è fondamentale comprendere il concetto di **classe**. Essenzialmente una classe è una *collezione di dati e funzioni* nel quale:

- I dati, che vengono chiamati **attributi** della classe, sono tutte le **variabili** che definiamo all'interno della nostra classe
- Le funzioni, che vengono chiamati **metodi** della classe, sono essenzialmente quelle che ci permettono di modellare il comportamento degli oggetti istanza della classe.

L'insieme di attributi e metodi della classe vengono inoltre chiamati **membri della classe**.

5.2.1 Classe vs Struct

Nei linguaggi C/C++ ci si può confondere il concetto di **classe** con il concetto di **struct**. Vediamo quindi le differenze:

- **Struct**: è la parola chiave che identifica una collezione di dati e oggetti, come per la classe. A differenza di quest'ultima, però, **tutti i dati e le funzioni sono public**, di conseguenza non è pensata per la OOP, infatti:
  - Non permette astrazione
  - Non permette incapsulamento
  - Il modificatore di accesso di default è public
- **Classe**: come la struct si tratta di una collezione di dati e oggetti che però fornisce la **possibilità di modificare i livelli di accesso ai membri della classe**. Inoltre il modificatore di accesso di default è private.

In conclusione si può dire che per la programmazione a oggetti la classe è lo strumento più indicato. Anche perchè la struct è un costrutto ereditato dal linguaggio C, che inizialmente non era pensato per la programmazione orientata agli oggetti.

5.2.2 Classe vs Oggetto

Un'altra distinzione fondamentale da comprendere è quella tra **classe** e **oggetto**:

- L'**oggetto** è un'**istanza della classe** che viene allocata in memoria
- La **classe** è la **definizione di un prototipo**

Quindi il passaggio da classe a oggetto avviene quando tramite un'istanza della classe viene allocata una parte di memoria per quel determinato oggetto.

5.2.3 Costruttore

Abbiamo detto che il passaggio da classe a oggetto avviene tramite l'*istanziazione*. Il **metodo responsabile dell'istanziazione di un oggetto** nella OOP viene chiamato **metodo costruttore**, ed ha le seguenti caratteristiche:

- Ha lo *stesso nome della classe*;
- Non ha *nessun tipo di ritorno*
- E' responsabile dell'*inizializzazione dei valori degli attributi*

La classe, quindi, ha sempre bisogno di un costruttore per poter essere istanziata. Inoltre esistono diversi tipi di costruttore:

- **Costruttore di Default**: è un costruttore senza parametri. Decidiamo noi, quindi, i valori da assegnare agli attributi.
- **Costruttore completo**: definiamo tutte le variabili tramite i parametri che decidiamo di passare.
- **Costruttore di Copia**: copia bit a bit lo stato, ovvero tutte le variabili inizializzate, del costruttore

Vediamo quindi l'esempio di una classe *persona*, con i diversi tipi di costruttori:

```
0  class Persona {
1      //Definiamo gli attributi della classe con modificatore private
2      private:
3          string nome, cognome, email;
4          int eta = 0;
5      public:
6          Persona() {} //Costruttore di Default: nessun parametro
7
8          Persona(string n, string c, string em, int e): nome(n),
       cognome(c), email(em), eta(e) {} // Costruttore "completo" con
       inizializzazione diretta degli attributi
9
10         Persona(Persona& p1) { //Costruttore di Copia
11             nome = p1.nome;
12             cognome = p1.cognome;
13             email = p1.email;
14             eta = p1.eta;
15         }
16 };
```

Codice 5.1: Tipi Di Costruttori

Vediamo di analizzare questa classe:

- Il **costruttore di default** in questo caso inizializzerà gli attributi nome, cognome ed email con la stringa vuota, mentre età con 0. Infatti la stringa, poichè anch'essa è una classe, non ha bisogno di essere inizializzata con la stringa vuota per il costruttore di default poichè chi ha definito la classe *string* ha definito come costruttore di default la stringa vuota.

- Il **costruttore completo** invece prende tutti i parametri che servono per l'inizializzazione delle variabili, e le inizializziamo mediante **lista di inizializzazione** o *inizializzazione diretta*, che ha la sintassi che possiamo vedere in riga 8, e ha appunto la semantica di assegnare i valori nelle parentesi all'attributo in questione. Viene anche detta inizializzazione diretta poichè in questo caso non entriamo nel corpo del costruttore, ma una volta che viene istanziato l'oggetto, vengono *contestualmente* inizializzate le variabili.

- Per quanto riguarda il **costruttore di copia**, esso prende come parametro il riferimento costante a un oggetto dello stesso tipo, e inizializza le variabili copiando bit a bit il valore degli attributi. Esiste, di default, inoltre, un costruttore di copia che effettua una cosiddetta *shallow copy*, ovvero vengono copiati tutte le variabili che hanno tipo primitivo nella classe, ma *i puntatori non vengono copiati*.

5.2.4 Invocazione dei Metodi

Per quanto riguarda i metodi della classe, abbiamo 2 modi di definire tali metodi:

- Si dice che il metodo è definito **inline** se la sua definizione è all'interno della classe. In questo caso quando istanzieremo l'oggetto, verrà allocata anche la parte di memoria per il metodo, ovvero l'allocazione in memoria avviene a **tempo di compilazione**. Quando verrà invocato, quindi, il metodo sarà già presente in memoria.

- Se il metodo non è invece definito all'interno ma all'**esterno** della classe, dichiarando il suo prototipo all'interno della classe stessa, allora la sua chiamata e la sua conseguente allocazione in memoria avverranno a **tempo di esecuzione**.

Esempio Prendiamo nuovamente la nostra classe *persona* e definiamo un metodo **print** prima inline e poi all'esterno:

```
0 class Persona() {
1
2     //............
3
4     void Print() {
5         std::cout << "Mi chiamo " << nome << " " << cognome << ",
      ho " << eta << " anni e la mia email    " << email << std::endl
      ;
6     }
7 }
```

In questo caso il metodo è definito inline, ovvero all'interno della classe. Vediamo, quindi come viene definito all'esterno:

```
0 class Persona() {
1
```

```
    //.............

    void Print();
};

void Persona::print() {
    cout << "Mi chiamo " << nome << " " << cognome << ", ho " <<
    eta << " anni e la mia email    " << email << endl;
}
```

In questo caso utilizziamo l'*operatore di scope* "::" che indica che stiamo definendo un metodo della classe Persona.

5.2.5 Distruttore

Nell'esempio visto in precedenza nella classe *persona* non abbiamo visto il caso in cui tra gli attributi ci fosse un'array. In quel caso avremmo dovuto allocarlo dinamicamente in tutti i costruttori. Abbiamo quindi visto come avviene l'allocazione in memoria di attributi e metodi, ma come avviene la deallocazione? Esiste il **metodo distruttore** che ha questo scopo, ovvero quello di deallocare tutti gli attributi. Per gli attributi definiti *staticamente*, come quelli della classe *persona* esiste un **distruttore di default** che dealloca automaticamente la memoria. Se invece abbiamo un array allocato dinamicamente, dobbiamo definire il metodo distruttore altrimenti quella parte di memoria allocata non viene deallocata, andando incontro a un *memory leak*, ovvero si ha una parte di memoria occupata e inutilizzabile.

Esempio Riprendiamo sempre la nostra classe *persona* e definiamo un nuovo attributo, stavolta un array che chiamiamo temperatura, che supponiamo presenti la temperatura misurata in 100 giorni. Per questo array, inoltre dovremo definire un distruttore. Vediamo quindi l'implementazione della nuova classe Persona:

```
class Persona {
    string nome, cognome, email;
    int eta=0;
    int* temperatura; //Nuovo array temperature

    public:
        //Ridefiniamo i costruttori per l'allocazione dell'array
    temperatura
        Persona() {
            temperatura = new int[100];
        }

        Persona(string name, string surname, string em, int e):
    nome(name), cognome(surname), email(em), eta(e) {
            temperatura = new int[100];
        }

        Persona(Persona& p1) {
            nome = p1.nome;
            cognome = p1.cognome;
```

```
18              email = p1.email;
19              eta = p1.eta;
20              temperatura = new int[100];
21          }
22
23
24          //.........
25
26
27          //Metodo distruttore: deallochiamo solo l'array allocato
        dinamicamente.
28          ~Persona() {
29              delete [] temperatura;
30          }
31  };
```

La *sintassi* del distruttore in C++ è quindi quella in riga 28, ovvero si mette la *tilde* " " seguita dal nome della classe e nessun parametro.

5.3 Static, Const e Friend

Iniziamo a vedere l'utilizzo di alcune importanti *keyword* nella OOP, iniziando da **static** e **const**.

5.3.1 Variabili static

Quando definiamo una funzione e inizializziamo una nuova variabile al suo interno, il suo *scope*, ovvero la sua visibilità e ciclo di vita è limitato al corpo della funzione stessa. Ad esempio:

```
0  void fun() {
1      int a = 10; //scope di a limitato alla funzione stessa
2  }
3
4
5  int main() {
6      fun();
7      //a=20; -> Non e possibile utilizzare a al di fuori di fun
8  }
```

Notiamo che, se decommentassimo la riga 7, avremmo un *errore di compilazione*, infatti la visibilità di a è limitata a *fun*, e non può essere utilizzata al dì fuori. Analizziamo, invece, il seguente codice:

```
0  void fun() {
1      static int a = 10;
2  }
```

In questo caso abbiamo inizializzato a **staticamente**, il ché significa che a viene inizializzata all'interno di fun ma non viene distrutta al dì fuori della funzione. Ovviamente, ciò non implica il fatto che a diventi una *variabile globale*, ovvero non posso utilizzarla al di fuori della funzione, ma implica il fatto che l'inizializzazione viene fatta solo la prima volta; dopo di chè la variabile permane in memoria e in tutte le chiamate successive della funzione sarà aggiornata, ovvero negli utilizzi successivi della funzione utilizzerò il valore aggiornato di a. E' per questo che le variabili static sono molto spesso utilizzate come *variabile contatore*:

```
void fun_static() {
    static int a=0;
    cout << "static access to a=" << a << endl;
    a++;
}

int main() {
    fun();
    for(int i=0; i<5; i++)
        fun_static();
    //cout << a << endl -> Variabile static non pu  essere
    utilizzata al di fuori della funzione
}
```

Codice 5.2: Variabili Static

Ciò che il terminale ci dirà eseguendo tale codice è:

```
static access to a=0
static access to a=1
static access to a=2
static access to a=3
static access to a=4
```

Poichè, dunque, abbiamo richiamato la funzione 5 volte, e, ad ogni chiamata ad a veniva sommato 1, si aggiorna ad ogni chiamata il valore di a.

Static nella OOP

Lo scopo che assumono **le variabili static nella OOP** si basa essenzialmente sulla loro caratteristica principale di permanere in memoria. Analizziamo, per esempio, il seguente codice:

```
class A {
    int a = 0;
    static int b;

    public:
        A() {
            b++;
        }

        void print() {
            cout << "a = " << a <<", b = " << b  << endl;
```

```
11          }
12 };
13
14 //Variabile static inizializzata al di fuori della classe in C++
15 int A::b=0;
```

Codice 5.3: Static nella OOP

Prendiamo come esempio le variabili private della classe A:

- Per quanto riguarda la variabile a, quando vado ad allocare della memoria per la classe ci sarà una parte di questa memoria riservata per a, e tale variabile verrà **creata a ogni istanza della classe**, per questo viene detta **variabile di istanza**.
- Per quanto riguarda la variabile b, la sua creazione avviene solo una volta, tanto che il linguaggio C++ non permette la sua inizializzazione all'interno della classe. Poichè, quindi, tale variabile viene creata una sola volta, *il suo valore sarà uguale per ogni istanza della classe*, e per questo viene chiamata **variabile di classe**.

Di conseguenza se avessimo il seguente *main*, considerando sempre la classe A del codice 5.3, avremmo che la variabile $b = 3$:

```
0 int main() {
1     //Chiamo 3 volte il costruttore, di conseguenza b=3
2     A a1;
3     A a2;
4     A a3;
5 }
```

Se invece considerassimo la chiamate una alla volta, come ad esempio:

```
0 int main() {
1     A a1;
2     a1.print(); //a=0, b=1
3
4     A a2;
5     a2.print();//a=0, b=2
6
7     A a3;
8     a3.print();//a=0, b=3
9 }
```

5.3.2 Const

La keyword **const** ha proprio lo scopo di indicare quando una variabile è costante, ovvero il suo valore è immutabile. Essa si può riferire a:

- Valori Costanti

- Puntatori Costanti a Valori
- Puntatori a Valori Costanti
- metodi e funzioni

Un esempio di utilizzo possiamo vederlo col seguente codice:

```
const int fun(const int a) { //Non posso modificare il valore della
     variabile che ritorno all'esterno
    return a*2;
}


int main() {
    const int a = 100;
    //a=1000; -> NO! Il valore di a    immutabile


    int b = 10;
    const int *c = &b; // puntatore a valore costante

    b=20;

    cout << b << " " << *c << endl;

    int* const x = &b; //puntatore costante a valore -> Si DEVE
    inizializzare in fase di dichiarazione
    //x=&b; ERRORE!
    *x=100;
    cout << "x= " << *x << endl;


    const int* const y = &b; //puntatore costante a valore costante


    const int z = fun(10);
    //z = 4;
}
```

Inoltre, all'interno di una classe, la keyword const viene molto spesso utilizzata nei metodi, soprattutto i *getter*, per impedire la modifica dei parametri. La sintassi è come quella che possiamo vedere nel metodo getNome della seguente classe:

```
class Persona {
    string nome;
    string cognome;

    public:
        Persona(string n, string c): nome(n), cognome(c) {}

        string getNome() const{ //Il valore di nome all'interno
    della classe    immodificabile
            return this->nome;
        }
```

```
10 };
```

5.3.3 Friend

La keyword **friend** ci permette di *fornire ad una funzione o a una classe esterne* **accesso ai membri privati e protetti della classe in cui appare**. Un'esempio può essere quello dell'overloading di un operatore di stampa di una classe:

```
0  class A {
1  int attribute=0;
2
3  public:
4      A();
5
6      friend ostream& operator<<(ostream& os, const A& a) {
7          os << "Sono l'attributo di A: " << a.attribute << endl;
8          return os;
9      }
10 }
```

Come possiamo vedere, grazie alla *friendship* accediamo all'attributo privato della classe A. Oltre che per i metodi, la keyword *friend* viene utilizzata anche per le classi. In particolare esistono 2 tipi di *friendship* per le classi:

- **simple friendship**: dichiariamo friend una classe già dichiarata
- **elaborate friendship**: dichiariamo friend una classe non ancora dichiarata. In particolare diciamo al compilatore che se dichiareremo una classe che ha quel nome indicato, sarà friend della nostra classe.

5.4 Ereditarietà, Polimorfismo e Template

Come probabilmente già sapete, l'**ereditarietà** è quel meccanismo della OOP che permette di ereditare delle caratteristiche da una *classe madre* nelle sue derivate, aggiungendo in queste ultime magari metodi o attributi utili alla determinata classe oppure fare *overiding* di alcuni metodi, ovvero modificare il corpo di alcuni metodi nelle classi derivate. Quando si deriva una classe, inoltre, si specifica un **modificatore di accesso**, che vengono usati in maniera differente rispetto a quanto si fa all'interno delle classi e stanno ad indicare *come modifico l'accesso a metodi o attributi da una classe madre A a una sua derivata B*. Vediamo attraverso questa tabella di capire quindi come viene modificato l'accesso agli attributi

Modificatore di Accesso	Membri di A	Membri di B
public A	private in A protected in A public in A	Non esiste protected in B public in B
protected A	private in A protected in A public in A	Non esiste protected in B protected in B
private A	private in A protected in A public in A	Non esiste private in B private in B

Table 5.1: Modificatori di Accesso per l'ereditarietà

Ereditarietà Multipla

Il nostro amatissimo linguaggio C++ ci permette, inoltre, di avere un **ereditarietà multipla**, grazie al quale posso derivare una classe da più classi. In tal caso vengono sequenzialmente chiamati i costruttori delle classi da cui derivo. Ad esempio:

```
0  class A{
1      public:
2          A(){cout << "constructor A" << endl;}
3  };
4
5  class B{
6      public:
7          B(){cout << "constructor B" << endl;}
8  };
9
10 class C: public A, public B{
11     public:
12         C(){cout << "constructor C" << endl;}
13 };
14
15
16 int main() {
17     C c;
18 }
```

Codice 5.4: Ereditarietà Multipla

```
0  constructor A
1  constructor B
2  constructor C
```

Ovviamente ciò avviene anche nell'ereditarietà semplice, ma, in tal caso, verranno chiamati sequenzialmente il costruttore della classe madre, e poi quello della classe figlia.

5.4.1 Binding e Polimorfismo

La caratteristica fondamentale che ci permette di avere l'ereditarietà, è quella del **polimorfismo**, che, da vocabolario è

l'assumere forme, aspetti, modi di essere diversi secondo le varie circostanze

Nella OOP il polimorfismo è quella caratteristica grazie al quale oggetti di classe diverse rispondono in maniera diversa a uno stesso metodo. In tal senso diventa fondamentale il concetto di **binding**: quando *compiliamo* il codice, tutte le istruzioni vengono tradotte in codice macchina, e vengono allocate in un certo indirizzo di memoria. Le funzioni, inoltre, quando vengono chiamate vengono allocate nello stack in un *record di attivazione*. Ecco che entra in gioco il binding: ovvero

*l'***associazione** *della* **chiamata della funzione** *e il suo* **codice**

In questo caso parleremo di:

- **Binding Statico** se tale associazione avviene a *tempo di compilazione*
- **Binding Dinamico** se tale associazione avviene a *tempo di esecuzione*

Su tale concetto si basa il polimorfismo. Vediamo quindi del codice:

```
0  class A {
1
2      public:
3          A(){cout << "costruttore di A" << endl;}
4          virtual void stampa() {cout << "sono la classe A" << endl;}
       //tipo definito a runtime
5          void foo() {cout << "sono foo di A" << endl;}//tipo
       definito a compiletime
6
7  };
8
9
10 class B: public A{
11     public:
12         B(){cout << "costruttore di B" << endl;}
13         void stampa() {cout << "sono la classe B" << endl;}//tipo
       definito a runtime
14         void foo() {cout << "sono foo di B" << endl;}//tipo
       definito a compiletime
15
16 };
17
18
19 int main() {
20     A a;
21     B b;
22
23     cout << "chiamata ai metodi delle istanze di A e B" << endl;
24
25     a.stampa(); //sono la classe A
26     b.stampa();// sono la classe B
27
```

```
28
29      a.foo();// sono foo di A
30      b.foo();// sono foo di B
31
32      cout << "creo dei puntatori di a e b ed assegno a ciascuno l'
        indirizzo di A e B" << endl;
33
34
35      A* pa = &a;
36      A* pb = &b;
37
38      //binding dinamico
39      pa->stampa();//sono la classe A
40      pb->stampa();//sono la classe B
41
42      //binding statico
43      pa->foo();//sono la classe A
44      pb->foo();//sono la classe A
45  }
```

Analizziamo bene questo codice: concentriamoci soprattutto sui metodi *stampa()* e *foo()*. Come possiamo notare, nella classe B abbiamo fatto un **overriding** di tali metodi, ovvero abbiamo ridefinito il corpo dei metodi. La differenza sta nel fatto che abbiamo dichiarato **virtual** il metodo *stampa()* nella classe madre, mentre il metodo *foo()* no. Per quanto riguarda la chiamata dei metodi per le variabili di istanza a, b, ovviamente il *binding* avviene a tempo di compilazione, ovvero staticamente, infatti quando vengono istanziate, sappiamo già a quale istanze della classe appartengono. Il discorso cambia quando invece parliamo di **puntatori**, o equivalentemente **reference**. Infatti in tal caso entra in gioco il polimorfismo. Per quanto riguarda il metodo *foo()*, il binding, proprio per l'assenza della keyword *virtual*, avviene sempre staticamente per cui poichè pa, pb sono *puntatori ad* A, verrà chiamato sia per pa che per pb il metodo *foo()* di A. Invece per il metodo *stampa()*, grazie all'uso della keyword **virtual**, vi è un binding dinamico, per cui a tempo di esecuzione avviene l'associazione tra la chiamata della funzione e il suo codice. Ciò permette, quando chiamiamo $pb->stampa()$, che venga chiamato il metodo *stampa()* di B.

Pure Virtual Un'altra considerazione da fare è quella sulla classi **pure virtual**, ovvero quelle classi che non possono essere istanziate. La sintassi è del tipo:

```
0  class A{
1      public:
2      A();
3      virtual void stampa() = 0; //Metodo pure virtual
4  }
```

Quando una classe presenta un metodo pure virtual siamo obbligati a definirlo in tutte le classi derivate. Tale caratteristica può essere utile quando devo creare una *collezione di oggetti* per il quale dichiaro il caso base.

5.4.2 Template

Uno strumento importantissimo per la definizione di funzioni e metodi sono i **template**, grazie al quale *riusciamo a definire* **tipi generici**. La sintassi è la seguente:

```
0 template<typename T>
```

T in questo caso rappresenta un tipo generico. Possiamo utilizzare tale sintassi sia per funzioni che per le classi. Una volta, che istanziamo una classe template, inoltre, dovremo specificare il tipo che stiamo andando ad utilizzare. Se per esempio avessimo una classe A del tipo:

```
0 template<typename T>
1 class A {};
```

Nel main verrà istanziata nel seguente modo:

```
0 int main() {
1     A<int> primo;
2 }
```

Parte III

Strutture Dati

Capitolo 6

Liste

L'ultima parte del corso è incentrata sulle **strutture dati**, che

sono entità astratte con comportamenti predefiniti per **organizzare dati**

La struttura dati che sicuramente conoscerete molto bene è l'array, che è una struttura dati che ci permette di organizzare dati dello stesso tipo in locazioni di memoria consecutive. Riassumendo le loro caratteristiche possiamo evidenziare dei vantaggi e degli svantaggi per gli array:

- *vantaggi*: Possiamo accedere ad ogni elemento *a tempo costante* ($O(1)$). Infatti l'array ci permette di accedere alla loro posizione in memoria tramite l'indicizzazione.
- *svantaggi*: lo svantaggio principale è che gli array hanno dimensione massima prefissata, per cui se ad esempio volessimo avere un'organizzazione di dati tale per cui ci serve aggiornare il numero di elementi da contenere nella struttura dati, l'array non sarebbe la scelta migliore.

6.1 Introduzione alle Liste

Una volta fatta una piccola introduzione sulle strutture dati in generale, iniziamo a parlare di **liste**. Se volessimo dare una definizione molto generale, possiamo dire che

la lista è una **sequenza di elementi** *i quali sono* **collegati l'uno all'altro**

Inoltre, a differenza dell'array, la lista *non viene preallocata* ma viene **popolata mentre inseriamo gli elementi**, ovvero ha una dimensione non prefissata.

6.2 Liste Linkate Semplici

Partendo dal più semplice tipo di lista, ovvero le **liste linkate semplici**, analizziamo come è fatta. Innanzitutto dobbiamo dire che la lista ha come elementi costitutivi

dei **nodi** che contengono:

- Un **valore**
- Un **puntatore**, o *link*, al **nodo successivo**, che è chiamato anche *successivo*.

Riformulando meglio la definizione generale per le liste linkate semplici, possiamo quindi dire che

la lista linkata semplice è **una sequenza di nodi** *in cui ogni nodo ha un valore e un* **collegamento** *(puntatore) all'elemento successivo*

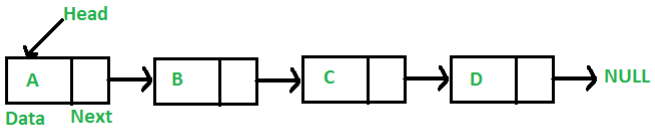


Figure 6.1: Esempio di Lista Linkata Semplice

La domanda che ci si pone, quindi, è: *come faccio ad accedere alla lista*? Utilizziamo una **head**, ovvero un puntatore al primo elemento della lista. Per *scorrere la lista*, quindi, partirò dalla head e seguirò il collegamento fino a quando non arrivo all'elemento che ci interessa. Quando termina la lista, come ultimo collegamento avremo un **NIL**, ovvero un *nullptr*, che indica il termine della lista. Vediamo quindi una prima implementazione:

```
0
1  //Classe Nodo
2  template<typename T>
3  class Node {
4      T val;
5      Node<T>* next;
6
7      template<typename U> //Nome del typename diverso poich  T
       gi   utilizzato per node
8      friend class List; //Posso dichiarare friend la classe List
       senza averla dichiarata -> elaborate friendship
9
10     public:
11         Node(T val): val(val) {
12             next = nullptr;
13         }
14
15         Node<T>* getNext() const {return this->next;}
16
17         //Overload operatore di redirezione dell'output
18         //Per accedere ai campi privati di node utilizzo il friend
19         friend ostream& operator<<(ostream& out, const Node<T> &
       node) {
20             out << "node val=" << node.val<< " - next=" << node.
       next;
```

```
21              return out;
22          }
23  };
```

Codice 6.1: Nodo della lista linkata semplice

Per quanto riguarda la classe **nodo** abbiamo:

- **attributi**: una variabile *val* che indica il valore contenuto nel nodo, e una variabile *next*, che indica il puntatore al successivo.
- **metodi**:
  - *costruttore*: imposta il valore del nodo, e il successivo a nullptr.
  - *getNext()*: ritorna il puntatore al successivo
  - *overloading dell'operatore di redirezione dell'output*: stampa per ogni il nodo il suo valore e il suo puntatore al successivo.

Vediamo quindi la classe lista:

```
0  //Classe Lista
1  template<typename T>
2  class List {
3      Node<T>* head;
4
5      public:
6      List(): head(nullptr) {} //HEAD va inizializzata nullptr
7  };
```

Codice 6.2: Lista Linkata Semplice

Essenzialmente quindi, poichè non abbiamo ancora visto le operazioni fattibili con le liste, definiamo semplicemente la *head*, ovvero il puntatore al primo elemento della lista, che va ovviamente inizializzato a *nullptr*, poichè l'inizializzazione della lista è una *lista vuota*.

6.2.1 Operazioni sulle Liste Linkate Semplici

Abbiamo visto un'introduzione alle liste, ma adesso vogliamo capire quali **operazioni** possiamo fare su tali liste. Le operazioni fattibili sono le seguenti:

- **inserimento**
- **accesso**
- **ricerca**
- **cancellazione**
- **copia**
- **controllo lista vuota**

Non entreremo nel dettaglio di tutte le operazioni, ma affronteremo sicuramente le più importanti come inserimento, cancellazione e ricerca, vedendone anche una implementazione.

Controllo Lista Vuota

Una delle operazioni fondamentali è quella del **controllo della lista vuota**, che è anche un operazione che serve a poter fare inserimento o cancellazione e consiste, quindi, semplicemente controllare se la head è *nullptr*. L'esempio può essere un metodo *isEmpty()* della classe *List* come il seguente:

```
0 //Controllo Lista Vuota
1     bool isEmpty() {
2         return head == nullptr; //se head    nullptr ritorna true
3     }
```

In questo caso quindi è un metodo abbastanza semplice che non ha bisogno di essere commentato.

Inserimento in Testa

L'operazione di **inserimento in testa** ha lo scopo di aggiungere un nuovo nodo alla testa, aggiornando di conseguenza la *head*. I passaggi da fare sono i seguenti:

1. *Controllare se la lista è* **vuota** tramite il metodo *isEmpty()*. Se lo è, in tal caso bisogna aggiornare il valore di *head* con il nuovo nodo creato.
2. Se la lista *non è vuota*, allora bisogna:
   - **creare** il nuovo **nodo**
   - **impostare il successivo** (*puntatore*) del nuovo nodo assegnandogli la testa della vecchia lista, ovvero il vecchio valore di *head*.
   - **aggiornare il valore di head**

Ricapitolando ciò che abbiamo scritto a parole in codice, possiamo creare un metodo *insertHead()* con la seguente implementazione:

```
0 void insertHead(T val) {
1 
2         //Inserimento in Testa se la lista    vuota
3         if(this->isEmpty()) {
4             this->insert(val);
5             return;
6         }
7 
8         //Inserimento in Testa se la lista non    vuota
9         Node<T>* temp = new Node<T>(val);
10        temp->next = head;
11        this->head = temp;
12    }
```

Codice 6.3: Inserimento in Testa

Quindi, tradotto in codice, per quanto riguarda l'inserimento in testa in una lista non vuota, la strategia che adottiamo è quella di creare un nodo *temp*, al quale assegniamo il valore passato come parametro. Poi impostiamo il successivo di *temp* con la *head* della vecchia lista e aggiorniamo *head*.

Inserimento in Coda

Nell'operazione di **inserimento in coda**, il nostro obiettivo è quello di creare un nuovo nodo e *piazzarlo alla fine della lista*. I passaggi che ci occorrono sono dunque i seguenti:

1. Se la **lista è vuota**, allora **inserisco il nuovo nodo in testa**
2. Se la **lista non è vuota**, allora bisogna:
   - **scorrere la lista** fino **all'ultimo elemento**
   - **inserire l'ultimo elemento**, ovvero:
     - *creare* un *nuovo nodo f* ar puntare il vecchio next al nuovo nodo

Tradotto in codice, vediamo l'implementazione dell'inserimento in coda:

```
0  void insertTail(T val) {
1          if(this->isEmpty()) {
2              this->insertHead(val);
3              return;
4          }
5
6          Node<T>* ptr = head;
7          while(ptr->getNext() != nullptr) { //Devo controllare se il
       successivo     null
8              ptr = ptr->getNext();
9          }
10
11         Node<T>* temp = new Node<T>(val);
12         ptr->next=temp;
13     }
```

Possiamo fare delle ulteriori considerazioni su tale codice:

- Notiamo innanzitutto che per scorrere la lista utilizziamo un *ciclo while*, infatti è più indicato dato che non sappiamo quante iterazioni dovremo fare, a differenza dell'array. La condizione che imponiamo è quella di avanzare finchè il successivo del nodo *ptr* è diverso da *nullptr*, infatti, come abbiamo già detto, il successivo dell'ultimo elemento è **NIL**, ovvero *nullptr*. Di conseguenza usciremo dal ciclo quando ptr conterrà l'ultimo elemento.
- Una volta trovato l'ultimo elemento, semplicemente creiamo un nuovo nodo *temp*, al quale assegnamo il valore dato in input, e aggiorniamo il successivo di ptr con il nuovo nodo, cosicchè quello che nella vecchia lista era l'ultimo nodo, adesso diventa il penultimo.

Inserimento Ordinato

Nel caso dell'**inserimento ordinato**, dobbiamo effettuare le seguenti operazioni:

- Dobbiamo *scorrere la lista* e *trovare la corretta posizione* nel quale inserire il nuovo nodo.
- Dobbiamo *"staccare" i collegamenti*, ovvero cambiare il successivo sia del nodo precedente che del nuovo nodo affinchè il successivo del nodo precedente sia il nuovo nodo, e il successivo del nuovo nodo sia il successivo del nodo precedente nella vecchia lista.

Tradotto in codice, il metodo che si implementa è il seguente:

```
0  void insertInOrder(T val) {
1          if(this->isEmpty()) {
2              this->insertHead(val);
3              return;
4          }
5          if(head->val >= val) { //Controllare se il valore da
       inserire sia minore uguale della testa
6              this->insertHead(val);
7              return;
8          }
9
10         Node<T>* ptr = head;
11         while(ptr->getNext() && (val >= ptr->val)) { //Controlliamo
        se ptr e' diverso da nullptr e se il valore da inserire e'
       maggiore del nodo
12             if(val <= ptr->next->val)
13                 break;
14             ptr = ptr->getNext();
15         }
16
17         if(!(ptr->next)) { //controllo se il successivo di ptr
       nullptr
18             this->insertTail(val);
19             return;
20         }
21
22         Node<T>* toInsert = new Node<T>(val); //creo il nuovo nodo
23         toInsert->next = ptr->next;
24         ptr->next = toInsert;
25     }
```

In ordine, i passi che eseguiamo sono i seguenti:

- Controlliamo se la lista è vuota, e, in tal caso dovremo inserire per forza in testa.
- Inoltre facciamo un altro controllo per vedere se il valore da inserire è minore o uguale a quello della testa, e, anche in questo caso, inseriremo in testa.
- Si crea quindi un nuovo nodo ptr, a cui assegniamo inizialmente la *head*, che ci servirà per trovare la corretta posizione in cui dovrà essere inserito. Infatti dobbiamo fare in modo che quando usciamo dal ciclo il *successivo di ptr* (next) sia il nuovo nodo. Infatti la condizione per cui funzioni tale metodo

è che il valore del successivo di ptr sia maggiore o uguale del valore passato come parametro e che il valore di ptr sia minore o uguale al valore passato come parametro.

- Una volta usciti dal ciclo while, avremo la corretta posizione di *ptr*, per cui se il successivo è nullptr, allora siamo nella coda, e , di conseguenza, possiamo inserire in coda. Altrimenti:
  - *creiamo il nuovo nodo*
  - *assegniamo al successivo del nuovo nodo il successivo di ptr*, ovvero il vecchio successivo di ptr nella vecchia lista.
  - *Aggiorniamo il successivo di ptr*, assegnandogli il nuovo nodo.

Cancellazione

Per quanto riguarda l'operazione di **cancellazione**, il nostro obiettivo è, banalmente, quello di eliminare uno specifico nodo. Inoltre, come per l'inserimento, possiamo:

- *cancellare la testa*
- *cancellare la coda*
- *cancellare uno specifico modo*

Per quanto riguarda la **cancellazione in testa** dobbiamo:

- Controllare se la lista vuota
- Se la lista è non vuota, spostare la testa al successivo, e cancellare l'elemento in testa della vecchia lista

Se traduciamo questi passaggi in codice otteniamo il seguente metodo:

```
0 void deleteHead() {
1         if(this->isEmpty()) {
2             cout << "Empty list!" << endl;
3             return;
4         }
5         Node<T>* temp = head;
6         head = head->next;
7 
8         delete temp;
9     }
```

Essenzialmente, quindi, ci prendiamo un riferimento alla testa con *temp*, così da eliminare la vecchia testa e aggiornare head.

Cancellazione in Coda

Nella **cancellazione in coda** utilizziamo tale strategia: creiamo un *puntatore al nodo corrente* e uno al *nodo precedente* cosicchè una volta arrivati al termine della lista possiamo imporre a null il successivo del nodo precedente, ed eliminare il nodo corrente corrispondente alla coda:

```
0  void deleteTail() {
1          if(this->isEmpty()) {
2              cout << "Empty list!" << endl;
3              return;
4          }
5
6
7          Node<T>* cur = head;//puntatore al nodo corrente
8          Node<T>* prev = nullptr; //puntatore al nodo precedente
9          while(cur->next) { //Controlliamo se il puntatore al
       successivo sia diverso da nullptr
10             prev = cur;
11             cur = cur->next;
12         }
13
14         prev->next = nullptr;
15         delete cur;
16     }
```

Complessità per le operazioni su una lista linkata La complessità delle operazioni di inserimento su una lista linkata sono le seguenti:

- **inserimento in testa**: $O(1)$, infatti abbiamo già un puntatore alla testa
- **inserimento ordinato**: $O(n)$, infatti faremo al più n passi , dove n è la dimensione della lista, per trovare l'elemento esatto nel quale inserire
- **inserimento in coda**: $\Theta(n)$, infatti faremo esattamente n passi.

Per quanto riguarda le operazioni di cancellamento la situazione è speculare, infatti:

- **cancellazione della testa**: $O(1)$
- **cancellazione della coda**: $\Theta(n)$
- **cancellazione di un nodo**: $O(n)$

6.3 Liste Doppiamente Linkate

Fino ad ora abbiamo trattato di *liste linkate semplici*, ovvero delle liste che avevano dei collegamenti semplicemente con il nodo successivo. Aggiungiamo adesso una nuova *feature* introducendo le **liste doppiamente linkate**. Dando una definizione generale possiamo dire che le liste doppiamente linkate sono

liste che hanno collegamenti sia al nodo successivo che al nodo precedente

Per cui avremo che il **nodo** di una lista doppiamente linkata avrà:

- un **puntatore** *all'elemento* **successivo**
- un **puntatore** *all'elemento* **precedente**

Inoltre la **lista** conterrà:

- un **puntatore alla testa**
- un **puntatore alla coda**

Infatti voglio poter *scorrere*, in una lista doppiamente linkata, *sia dalla destra che dalla sinistra*.

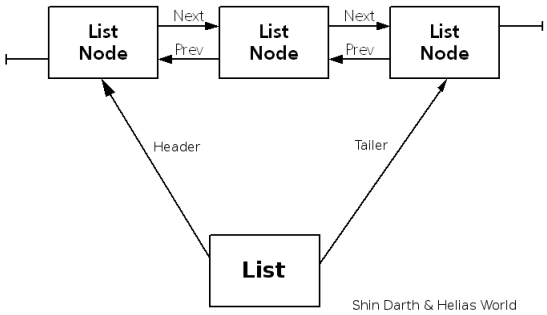


Figure 6.2: Lista Doppiamente Linkata

6.3.1 Nodo della lista doppiamente Linkata

Abbiamo già detto che il **nodo** di una lista doppiamente linkata deve avere le seguenti caratteristiche:

- Avere un **puntatore al nodo successivo**
- Avere un **puntatore al nodo precedente**
- Avere un **valore**

Inizialmente, quindi, il precedente e il successivo sono inizializzati a nullptr. Vedremo che poi nella gestione dei nodi nella lista dovremo fare molta attenzione a come gestiamo tali collegamenti.

```
0 template <typename T>
1 class DLNode {
2   T val;
3   DLNode<T> *next;
4   DLNode<T> *prev;
5
```

```
6     template <typename U>
7     friend class DLList;
8
9     public:
10
11    DLNode(T val) : val(val), next(nullptr), prev(nullptr) {}
12
13    DLNode<T>* getNext() const { return this->next; }
14    DLNode<T>* getPrev() const { return this->prev; }
15
16  };
```

Codice 6.4: Nodo della lista doppiamente Linkata

In questo caso abbiamo creato anche dei metodi *getNext()* e *getPrev()* che ritornano, rispettivamente, il successivo e il precedente. Inoltre abbiamo impostato come friend la classe DLList, che deve essere ancora inizializzata.

6.3.2 Implementazione della classe DLList

In questo caso gli attributi della lista doppiamente linkata sono 2, ovvero il **puntatore alla testa** e il **puntatore alla coda**. Poichè inoltre la lista doppiamente linkata ha per ogni nodo un collegamento al nodo precedente e uno al successivo, in questo caso abbiamo 2 NIL, infatti il precedente della testa è nullptr, e lo stesso vale per il successivo della coda.

```
0  template<typename T>
1  class DLList {
2    DLNode<T> *head;
3    DLNode<T> *tail;
4
5    public:
6
7    DLList() {
8      head = nullptr;
9      tail = nullptr;
10   }
11 };
```

Notiamo che, come per le liste linkate semplici, inizialmente head e tail vanno inizializzate a nullptr, questo infatti permette di evitare dei problemi sull'allocazione in memoria dei 2 attributi.

6.3.3 Operazioni sulle liste doppiamente linkate

Le operazioni che possiamo effettuare sulle liste doppiamente linkate sono le stesse delle liste linkate semplici. In questo caso, però, dobbiamo porre attenzione al fatto che i nodi abbiano 2 collegamenti (puntatori), e, inoltre, abbiamo anche un puntatore alla coda che ci aiuta in operazioni come l'inserimento in coda, ecc.

Inserimento

Per quanto riguarda l'inserimento, che sia in testa o in coda, dobbiamo prestare attenzione ai seguenti fattori:

- Controllare se la lista è vuota. In tal caso inizializzare la testa con il valore passato come parametro, e uguagliare la coda alla testa.
- Se la lista è non vuota, in tal caso il nuovo nodo da inserire avrà come successivo la testa della vecchia lista. Se l'inserimento avviene in coda, in tal caso il nuovo nodo avrà come precedente la coda della vecchia lista.
- Infine bisogna aggiornare la testa o la coda.

Vediamo quindi l'implementazione:

```
0  void insertHead(T val) {
1      if(this->isEmpty()) {
2        head = new DLNode<T>(val);
3        tail = head;
4        return;
5      }
6
7      DLNode<T> *toInsert = new DLNode<T>(val);
8      toInsert->next = head;
9      head->prev = toInsert;
10     head = toInsert;
11   }
12
13   void insertTail(T val) {
14     if(this->isEmpty()) {
15       this->insertHead(val);
16       return;
17     }
18
19     DLNode<T> *toInsert = new DLNode<T>(val);
20     toInsert->prev = tail;
21     tail->next = toInsert;
22     tail = toInsert;
23   }
```

Cancellazione

Per quanto riguarda la **cancellazione**, in tal caso è importante fare molta attenzione ai collegamenti da cambiare o imporre a null. Vediamo quindi un implementazione e spieghiamo poi il funzionamento:

```
0  void removeHead() {
1      if(this->isEmpty()) {
2        std::cout << "Empty list" << std::endl;
3        return;
4      }
5
```

```
6       if(head == tail) {
7         DLNode<T> *ptr = head;
8         head = nullptr;
9         tail = nullptr;
10        delete ptr;
11      }
12
13      DLNode<T> *ptr = head;
14      ptr->next->prev = nullptr;
15      head = ptr->next; //head->next
16
17      delete ptr;
18    }
19
20    void removeTail() {
21      if(this->isEmpty()) {
22        std::cout << "Empty list" << std::endl;
23        return;
24      }
25
26      if(head == tail) {
27        DLNode<T> *ptr = head;
28        head = nullptr;
29        tail = nullptr;
30        delete ptr;
31      }
32
33      DLNode<T> *ptr = tail;
34      ptr->prev->next = nullptr;
35      tail = ptr->prev; //tail->prev
36
37      delete ptr;
38    }
```

In questo caso:

- Si controlla se la lista è vuota. Se non è vuota si controlla se contiene un solo nodo (ovvero la testa coincide con la coda), e, in tal caso, si impongono a nullptr sia testa che coda e si elimina il ptr, con cui abbiamo preso il riferimento della testa.
- Nel caso di **cancellazione in testa**, prendiamo un riferimento alla testa. Bisogna poi imporre il precedente del successivo a null , poichè la testa deve essere eliminata (ptr-¿next-¿prev = nullptr), e spostare la vecchia testa al successivo (head = ptr-¿next).
- Nel caso di **cancellazione in coda**, prendiamo un riferimento alla coda. In tal caso, quello che dobbiamo imporre a null è il successivo del precedente (ptr-¿prev-¿next = nullptr;), poichè dobbiamo eliminare la coda, e spostare la coda al precedente (tail = ptr-¿prev;)

Capitolo 7

Pile e Code

Abbiamo analizzato fino ad ora, come strutture dati, solo le *liste*, escludendo gli array. La lista è una struttura dati che ha delle *operazioni tipiche*, e che si differenzia dall'array solo per non avere una dimensione fissata a priori, ovvero è una struttura dati dinamica. Parlando invece di **pile** e **code**, dobbiamo cambiare il nostro punto di vista. Infatti le operazioni di tale strutture dati, e anche di quelle che vedremo in seguito, sono *atipiche* e *astratte*, ovvero a prescindere da come le implementiamo, non cambia il risultato delle operazioni che facciamo in tali strutture dati.

7.1 Pile

La **pila**, o *stack*, è una struttura dati di tipo **LIFO**, ovvero *last-in, first-out*. Si tratta quindi di una struttura dati astratta nel quale le operazioni di inserimento e rimozione avvengono sempre dalla *cima della pila*. Possiamo immaginare una pila di piatti, per il quale se vogliamo aggiungere o prendere un piatto possiamo o togliere o mettere piatti solo in cima, altrimenti cadrebbe tutta la pila. Riassumendo quanto detto, possiamo dire che la pila ha i seguenti elementi costitutivi:

- **Attributi** della Pila:
  - Dimensione
  - Accedere alla cima della pila
- **Metodi** della Pila:
  - *push()*, ovvero aggiungere un elemento alla cima della pila
  - *pop()*, ovvero rimuovere un elemento dalla cima
  - *isEmpty()*: controllo della pila vuota.

7.1.1 Implementazione della Pila

Proprio per la sua caratteristica di struttura dati astratta, possiamo implementare la pila in diversi modi. Due modi possono essere:

- Implementazione tramite **array**
- Implementazione tramite **lista**

In tal caso, l'implementazione dipende dal problema che ci viene posto. Infatti, nella pila implementata tramite array dobbiamo considerare una dimensione massima, mentre con l'implementazione tramite lista, non abbiamo una dimensione prefissata.

Implementazione tramite Lista

Per l'implementazione tramite lista potremmo pensare di creare una nuova classe *stack*, che sia una specializzazione della classe *List* vista nel capitolo precedente (sezione 6.2). L'unico attributo che ci servirà sarà una variabile privata *size*, che ci darà la dimensione della pila. I metodi che andremo a implementare saranno quindi:

- *top()*: ovvero un metodo che ci restituisce la testa della lista, ovvero la cima della pila
- *push()*: ovvero il metodo con il quale inseriremo in testa alla lista un elemento
- *pop()*: ovvero il metodo che rimuove la testa e ci restituisce il nodo in cima, ovvero in testa.

Quindi, l'implementazione sarà come segue:

```
0  template<typename T>
1  class Stack : protected List<T> {
2
3    int size = 0;
4
5    public:
6
7    Stack() : List<T>() {}
8
9    bool isEmpty() {
10     return size == 0;
11   }
12
13   Node<T> * top() {
14     if(isEmpty())
15       return nullptr;
16
17     return List<T>::getHead();
18   }
19
20   void push(T val) {
21     List<T>::insertHead(val);
22     size++;
23   }
24
25   Node<T>* pop() {
26     if(isEmpty())
27       return nullptr;
28
```

```
29      Node<T>* ptr = top();
30      List<T>::removeHead();
31      size--;
32      return ptr;
33    }
34  };
```

Delle considerazioni importanti da fare riguardano la derivazione **protected** della classe *List*. Infatti non vogliamo che l'utente usi qualche metodo della classe Lista che non è consentito usare al di fuori di essa. Quindi per ovviare a tale problema, facciamo una derivazione protected. Inoltre molta attenzione va prestata quando utilizziamo metodi di classe $List < T >$, in quanto è necessario utilizzare l'operatore di risoluzione di scope per le classi template.

Implementazione tramite Array

Nell'**implementazione tramite array**, ci costruiamo una nuova classe che come struttura dati per contenere la pila utilizza l'array appunto, e utilizziamo l'attributo *top*, che ci restituisce la cima, *size*, che indica la dimensione attuale della pila e *maxSize*, che indica la dimensione massima della pila. Per quanto riguarda i metodi sono esattamente gli stessi di quelli visti nell'implementazione dinamica tramite lista:

```
0  template<typename T>
1  class StaticStack {
2      T* array;
3      int top = -1;
4      int size = 0;
5      int maxSize = -1;
6
7      public:
8          StaticStack(int _maxsize): maxSize(_maxsize) {
9              array = new T[maxSize];
10         }
11
12         T getTop() {
13             if(isEmpty())
14                 return -1;
15             return array[top];
16         }
17
18         void push(T val) {
19             if(top == maxSize-1)
20                 return;
21
22             array[++top] = val;
23         }
24
25         T pop() {
26             if(isEmpty())
27                 return -1;
28
```

```
29              return array[top--];
30          }
31
32          bool isEmpty() {
33              return top == -1;
34          }
```

Ovviamente, il metodo *isEmpty*, utilizzato in entrambe le implementazioni ci serve per sapere se la struttura è vuota. Inoltre quando dovremo fare un *push*, ovvero aggiungere un nuovo elemento alla pila, prima dovremo incrementare (partendo da -1) e poi aggiungere il nuovo elemento, mentre nel metodo*pop* prima restituiremo l'elemento in cima e poi decrementeremo la cima.

7.1.2 Complessità delle operazioni della Pila

Poichè abbiamo sempre un riferimento alla cima (*top*), la complessità di tutte le operazioni sarà a **tempo costante**, infatti:

- *push*: $O(1)$
- *pop*: $O(1)$
- *top*: $O(1)$

7.2 Code

La **coda**, o *queue*, è una struttura dati di tipo **FIFO**, ovvero *first-in, first-out*. Ovvero, a differenza della pila, in questo caso il primo ad entrare nella coda è il primo che viene servito: ciò significa che gli elementi che si aggiungono si *accodano*, mentre la coda viene servita a partire dalla *testa*. Di conseguenza possiamo dire che le caratteristiche della coda, per quanto riguarda attributi e metodi, sono i seguenti:

- **Attributi**:
  - *Head* e *Tail*, ovvero la testa e la coda della struttura dati
  - *dimensione*
- **Operazioni**:
  - *enqueue*, ovvero l'inserimento in coda, che corrisponde al metodo *insertTail()* della lista linkata
  - *dequeue*, ovvero l'estrazione dalla coda, che corrisponde al metodo *removeHead* della lista linkata.

Come per la pila, anche per la coda, essendo una struttura dati astratta, esistono diversi tipi di implementazione. Anche in questo caso, infatti, vedremo un implementazione tramite lista e un implementazione tramite array.

7.2.1 Implementazione tramite Lista

Anche in questo caso nell'implementazione tramite lista creiamo una nuova classe *queue* come specializzazione della classe lista al quale aggiungiamo:

- Un attributo *size* che conterrà la dimensione della struttura dati
- Un metodo *enqueue* che corrisponderà a un *insertTail* della lista, e incrementiamo la dimensione
- Un metodo *dequeue* che estrae il valore dalla testa e la rimuove tramite *removeHead* della lista. In questo caso viene decrementata la dimensione.

```
0  class queue: protected DLList<T> {
1      protected:
2          int size = 0;
3      public:
4          queue(): DLList<T>() {}
5
6          bool isEmpty() {return size == 0;}
7
8          void enqueue(T val) {
9              DLList<T>::insertTail(val);
10             size++;
11             return;
12         }
13
14         DLnode<T> dequeue() {
15             if(this->isEmpty()) {
16                 return ;
17             }
18
19             DLnode<T> ptr = *(DLList<T>::head);  // dereferenzio la
       testa cosicche quando verr  eliminata non perder  il valore
20             DLList<T>::deleteHead();
21             size--;
22
23             return ptr;
24         }
```

Come possiamo vedere anche in questo caso ereditiamo **protected** per evitare di utilizzare metodi della classe *DLList* al di fuori della coda.

7.2.2 Implementazione tramite Array

Per l'implementazione tramite array della coda si pone un problema. Infatti può capitare l'eventualità per cui *head* e *tail* si trovino nella stessa posizione o che la testa sia in posizioni più avanzate rispetto alla coda e ciò sarebbe un controsenso. Per questo bisogna cercare un alternativa a un *array lineare*. La soluzione migliore è quella di implementare un **array circolare** attraverso l'utilizzo dell'operatore di *modulo*. In questo caso quindi creiamo una nuova classe *staticQueue* che ha le seguenti caratteristiche:

- **Attributi**:
  - L'array
  - *size* e *maxSize*, che ci indicheranno rispettivamente la dimensione attuale e la dimensione massima della coda.
  - *head* e *tail*, che inizialmente vengono inizializzati a -1.
- **Metodi**:
  - *enqueue*: in questo caso incrementiamo la coda utilizzando l'operatore di modulo con maxSize, e assegniamo all'indice tail dell'array il nuovo valore. Incrementiamo inoltre la dimensione
  - *dequeue*: in questo caso incrementiamo la testa utilizzando l'operatore di modulo con maxSize, e restituiamo il valore della testa. Decrementiamo inoltre la dimensione.

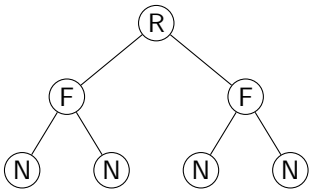
```
0  template<typename T>
1  class staticQueue {
2      T* arr;
3
4      int size = 0;
5      int maxSize = MAX_SIZE;
6
7      int head = -1;
8      int tail = -1;
9
10     public:
11         staticQueue(int maxSize = MAX_SIZE) :  maxSize(maxSize) {
12             this->array = new T[maxSize];
13         }
14
15
16         void enqueue(T val) {
17             if(size == maxSize) {
18                 cout << "Queue is full" << endl;
19                 return;
20             }
21             if(head == -1)
22                 head = 0;
23
24             tail = (++tail % maxSize);
25
26             array[tail] = val;
27             size++;
28         }
29
30         T dequeue() {
31             if(size == 0) {
32                 cout <<"Queue is empty" << endl;
33                 return -1;
34             }
35
36             size--;
```

```
37              T val = array[head];
38              head = (++head % maxSize);
39              return val;
40          }
```

Capitolo 8

Alberi Binari Di Ricerca

In natura, un *albero* si costituisce di una radice e all'estremità da dei rami che hanno delle foglie. In informatica un albero ha la struttura opposta, ovvero è una struttura che parte da una radice, che si sviluppa nei suoi rami e infine con le foglie. Un esempio può essere il seguente albero:



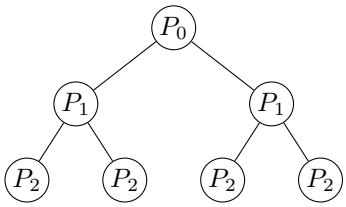
In tal caso abbiamo che per ogni nodo, ovvero ogni "cerchio" dell'albero, si può stabilire una gerarchia *genitore-figlio*, quasi come una parentela. Infatti, come possiamo notare dall'albero visto in precedenza, il nodo **R** rappresenta la radice, quelli che sono denominati con **F**, sono i *figli* del nodo radice, quelli denominati con **N**, i nipoti eccetera. Per gli alberi, un'altra caratteristica importante è l'**arietà**, ovvero il *numero di figli possibile*, che ci indica il **grado dell'albero**. Per esempio, possiamo avere un albero binario, ovvero un albero in cui ogni nodo può avere al più 2 figli; un albero ternario, ovvero un albero in cui ogni nodo può avere al più 3 figli ecc. In questo corso ci occuperemo solo di **alberi binari**.

8.1 Caratteristiche e Proprietà degli Alberi Binari

Prima di vedere la parte che piace di più a noi informatici, ovvero l'implementazione in codice della struttura dati, dobbiamo conoscerne delle **caratteristiche e proprietà di base**

8.1.1 Altezza e Livello dell'Albero

Analizziamo il seguente albero binario, infatti ogni nodo ha 2 figli:



Per capire il concetto di *profondità* abbiamo dato dei nomi significativi ai nodi. Infatti notiamo che abbiamo nominato la radice con P_0, il ché significa che ha profondità 0. I figli sono denominati con P_1, ovvero hanno profondità 1, eccetera. Tale profondità di ogni nodo viene chiamata **livello dell'albero**, per cui la radice si trova a livello 0, i suoi figli a livello 1, eccetera. Il *numero di livelli* totale di un albero viene chiamata **altezza dell'albero**.

8.1.2 Numero Massimo di Nodi di un Albero Binario

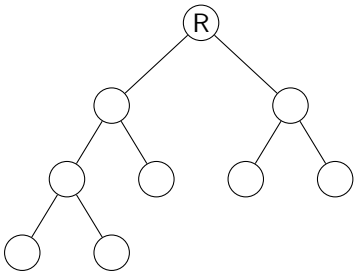
Trattandosi di alberi binari, il conto per trovare il numero massimo di nodi, conoscendo l'altezza dell'albero è molto semplice. Infatti, tenendo conto del fatto che la radice possiede solo un nodo, possiamo dire che detto n come *numero di nodi*, allora

$$n \leq 2^h - 1 \tag{8.1}$$

dove h rappresenta l'altezza dell'albero. Possiamo inoltre affermare che si dice **albero binario completo** un albero che ha esattamente $2^h - 1$ nodi, dove h è l'altezza dell'albero. L'albero visto nella precedente sezione, per esempio, è completo.

8.1.3 Albero Bilanciato

Per introdurre la definizione di **albero bilanciato**, bisogna comprendere bene cosa si intenda per **sottoalbero**, che, fissato un nodo, è *l'albero che ha come radice il figlio del nodo*. Inoltre possiamo dire che tale definizione può essere applicata ricorsivamente per ogni nodo dell'albero. Prendiamo in considerazione tale albero:



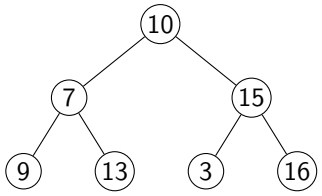
Possiamo notare ad esempio che il nodo *radice*, ha un sottoalbero destro e un sottoalbero sinistro. Una volta compreso il concetto di sottoalbero, possiamo dire che quando la differenza tra l'altezza tra i 2 sottoalberi della radice è al più 1, allora l'albero di dice bilanciato.

8.2 Alberi Binari di Ricerca

Un particolare tipo di alberi binari, sono gli **alberi binari di ricerca**, detti anche **BST** (*binary search trees*), che hanno la seguente costruzione:

- Per *ogni nodo dell'albero*, i valori del suo **sottoalbero sinistro** sono **minori** del valore stesso
- Per Per *ogni nodo dell'albero*, i valori del suo **sottoalbero destro** sono **maggiori** del valore stesso

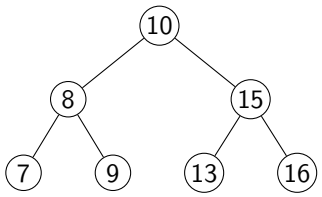
Analizziamo il seguente albero:



Possiamo notare che il seguente albero binario *non è un BST*. Infatti abbiamo i seguenti errori:

- $9 > 7$: infatti abbiamo detto che i valori presenti nel sottoalbero sinistro devono essere minori del nodo di riferimento
- $13 > 10$: infatti se possiamo notare che vale la proprietà per cui $13 > 7$, poichè la definizione è ricorsiva, dobbiamo avere la situazione per cui anche il sottoalbero sinistro della radice deve avere tutti i nodi minori del valore della radice stessa, e, se $13 > 10$ tale proprietà non viene mantenuta
- $3 < 10$: nel sottoalbero destro della radice vi devono essere valori maggiori della radice, quindi tale nodo non rispetta le proprietà dell'albero binario.

Analizziamo invece quest'altro albero:



Possiamo ben notare che tale albero **è un BST**, infatti rispecchia tutte le condizioni per poterlo essere.

8.2.1 Implementazione del BST

Una volta compreso come teoricamente viene costruito un BST, andiamo a implementarlo. Suddividiamo l'implementazione del nodo e dell'albero in 2 classi diverse.

Implementazione del Nodo

Molto semplicemente, il nodo abbiamo visto che come attributi presenta una *chiave*, ovvero il valore associato al nodo, un *figlio destro* e un *figlio sinistro*. Per tale classe, oltre al costruttore che impone figlio sinistro e destro a nullptr e la chiave con il valore passato come parametro, implementeremo quindi anche dei metodi setter per assegnare un valore ai figli sinistri e destri e al valore, e allo stesso modo dei metodi getter per ottenere i valori del nodo:

```
0  template<typename T>
1  class BSTNode{
2
3      protected:
4      T key;
5      BSTNode<T>* left; //FIGLIO SINISTRO
6      BSTNode<T>* right; //FIGLIO DESTRO
7
8      template<typename U>
9      friend class BST;
10
11     public:
12
13     BSTNode(T key): key(key) {
14         left = nullptr;
15         right = nullptr;
16         parent = nullptr;
17     }
18
19     void setLeft(BSTNode<T>* left) {
20         this->left = left;
21     }
22
23
24     void setRight(BSTNode<T>* right) {
25         this->right = right;
26     }
27
28     void setKey(T key) {this->key = key;}
29
30     BSTNode<T>* getLeft() {return this->left;}
31
32     BSTNode<T>* getRight() {return this->right;}
33 };
```

Codice 8.1: Classe BSTNode (nodo di un BST)

Implementazione della classe BST

Per quanto riguarda la classe BST, l'attributo che abbiamo è un nodo *root*, rappresentante la radice dell'albero, e dei metodi *insert()*, per inserire un nodo nell'albero rispettando le proprietà del BST. Tale procedura sarà, proprio per le caratteristiche dell'albero, una **procedura ricorsiva** che controlla, dato un nodo e una chiave, che il nodo sia una foglia, ovvero che non abbia o figlio sinistro o destro, e se la chiave è minore o uguale del nodo attuale inseriamo un nuovo figlio sinistro, altrimenti

un nuovo figlio destro. Se il nodo non è una foglia, si fa un nuovo controllo sulla chiave per vedere se è minore uguale della chiave del nodo o maggiore in maniera ricorsiva richiamando la procedura insert sul rispettivo figlio destro o sinistro. Per comprendere meglio tale procedura vediamo il codice:

```
template<typename T>
class BST {
    BSTNode<T>* root; //radice


    public:
    BST() {root = nullptr;}

    bool isEmpty() {
        return root == nullptr;
    }


    BSTNode<T>* getRoot() {return root;}
    void insert(BSTNode<T>* ptr, T key) {
        if(!(ptr->left) && key <= ptr->key) {
            ptr->left = new BSTNode<T>(key);
            return;
        } else if(!(ptr->right) && key > ptr->key) {
            ptr->right = new BSTNode<T> (key);
            return;
        }

        if(key <= ptr->key)
            insert(ptr->left, key);
        else
            insert(ptr->right, key);
    }
};
```

Un overloading di tale metodo si può fare quando vogliamo inserire un nodo a partire dalla radice. In tal caso il metodo sarà il seguente:

```
void insert(T key) {
        if(this->isEmpty()) {
            root = new BSTNode<T> (key); // inserisco la radice se
    l'albero    vuoto
            return;
        }

        insert(root, key);
    }
```

8.2.2 Visita dell'Albero

Per quanto riguarda gli alberi non esiste una *stampa*, ma si può **visitare l'albero** in diversi modi. Per visita di un albero si intende il modo e la sequenza in cui vogliamo

vedere i nodi dell'albero. Se poi ci interessa anche stamparli possiamo durante la visita fare una stampa della chiave del nodo. Bisogna però specificare che si tratta di 2 cose diverse. Esistono diversi modi di visitare l'albero:

- **Visita Preorder**
- **Visita Postorder**
- **Visita Inorder**

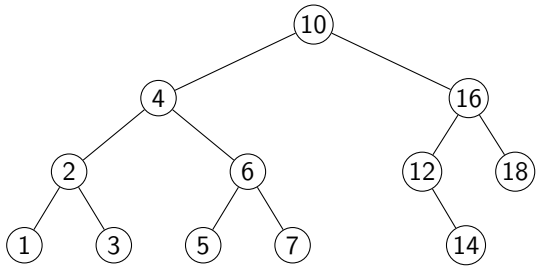
Tutti i modi di visitare l'albero hanno in comune il fatto di essere procedure ricorsive, che si applicano ad ogni nodo.

Visita Preorder

La **visita preorder** prevede i seguenti passi:

- Visitare la *radice*
- Visitare il *sottoalbero sinistro*
- Visitare il *sottoalbero destro*

Consideriamo quindi il seguente BST:



Se visitassimo tale BST in maniera preorder stampando le chiavi associate ai nodi che visitiamo, la sequenza di chiavi stampate sarebbe la seguente:

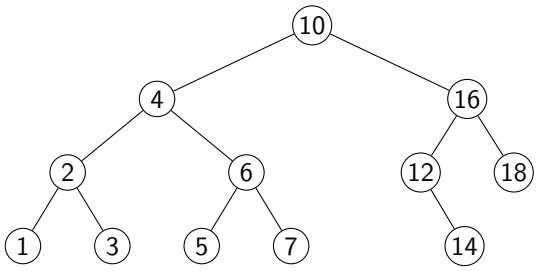
$$10 - 4 - 2 - 1 - 3 - 6 - 5 - 7 - 16 - 12 - 14 - 18 \tag{8.2}$$

Visita Postorder

La **visita postorder** prevede i seguenti passi:

- Visito il *sottoalbero sinistro*
- Visito il *sottoalbero destro*
- Visito la *radice*

Considerando quindi il precedente BST, che è il seguente:



In tal caso se immaginassimo di stampare la sequenza di nodi che andiamo a visitare, dobbiamo considerare il fatto che la visita del sottoalbero sinistro e destro si applica ad ogni nodo, e quindi la sequenza di stampa sarebbe la seguente:

$$1 - 3 - 2 - 5 - 8 - 6 - 4 - 14 - 12 - 18 - 16 - 10 \tag{8.3}$$

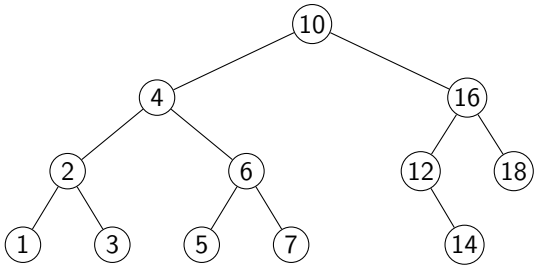
Se ci facciamo caso, inoltre, possiamo notare che la visita postorder dell'albero coincide a visitare la pila che contiene le chiavi dell'albero: infatti la radice sta in cima e al di sotto i suoi elementi.

Visita Inorder

La **visita inorder** di un albero prevede i seguenti passi:

- Visito il *sottoalbero sinistro*
- Visito la *radice*
- Visito il *sottoalbero destro*

Riprendiamo quindi il nostro solito BST:



In tal caso la sequenza di chiavi associate alla visita che facciamo ai nodi che stamperemmo sarebbe la seguente:

$$1 - 2 - 3 - 4 - 5 - 6 - 7 - 10 - 12 - 14 - 16 - 18 \tag{8.4}$$

Notiamo, proprio per la proprietà dell'albero di avere in ogni sottoalbero sinistro di un nodo valori minori del nodo stesso e, in ogni sottoalbero destro valori maggiori del nodo stesso ricorsivamente per ogni nodo, fa si che la visita inorder sia una visita dell'albero in maniera *ordinata*, ovvero la sequenza di chiavi associate ai nodi che visitiamo si presenta secondo un ordinamento crescente.

8.2.3 Ricerca

La **ricerca** di un nodo in un BST è una delle proprietà caratterizzanti della struttura dati. Infatti, proprio per come è strutturato l'albero, possiamo ricercare un nodo, data una chiave di partenza, con la *ricerca dicotomica*. Infatti sappiamo che il sottoalbero destro di ogni nodo, contiene i nodi che hanno chiavi con valore maggiore rispetto al nodo, mentre i nodi del sottoalbero sinistro hanno chiavi con valore minore rispetto al nodo. Per cui, possiamo ricorsivamente attuare una procedura per cui se la chiave da ricercare è minore della chiave del nodo attuale, la ricerchiamo nel suo sottoalbero sinistro, altrimenti nel suo sottoalbero destro. L'implementazione dunque, sarà la seguente:

```
0  BSTNode<T>* search(BSTNode<T>* ptr, T key) {
1      if(ptr == nullptr)
2        return nullptr;
3      if(ptr->key == key)
4        return ptr;
5
6      if(key <= ptr->key)
7        return search(ptr->left, key);
8      else
9        return search(ptr->right, key);
10
11     return nullptr;
12   }
```

Complessità Asintotica di visita e ricerca Per la visita dell'albero abbiamo che la complessità è dell'ordine

$$\Theta(h) \tag{8.5}$$

dove h rappresenta l'altezza dell'albero. Infatti dobbiamo fare sempre h iterazioni per visitare l'intero albero. Per la ricerca di una chiave invece la complessità è dell'ordine di

$$O(h) \tag{8.6}$$

infatti in tal caso dobbiamo fare al più h passi, infatti non sempre la chiave si trova in una foglia dell'albero.

8.2.4 Minimo e Massimo

Vista la suddivisione dell'albero, trovare **minimo** e **massimo** è molto semplice, infatti per la proprietà di maggioranza di ogni sottoalbero destro, e di minoranza del sottoalbero sinistro, per trovare il minimo *si scorre l'albero a partire della radice sempre a sinistra*, ovvero scorrendo ogni figlio sinistro, fino a trovare la foglia che rappresenta il minimo. Specularmente, per trovare il massimo, basta *scorrere l'albero a partire della radice e andando a destra*, fino a trovare la foglia che rappresenta il massimo.

L'implementazione del minimo quindi è la seguente:

```
0  BSTNode<T>* min(BSTNode<T>* from) {
1      if(isEmpty()) {
2        return NULL;
3      }
4
5      BSTNode<T>* ptr = from;
6      while(ptr->left) {
7        ptr = ptr->left;
8      }
9
10     return ptr;
11   }
```

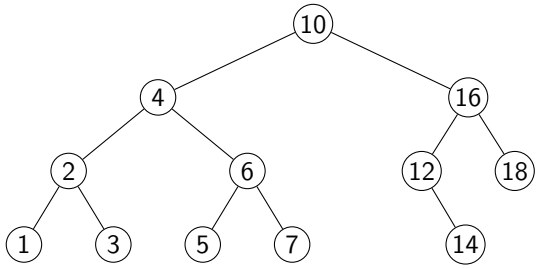
Codice 8.2: Minimo dell'albero

L'implementazione del massimo è la seguente:

```
0  BSTNode<T>* max(BSTNode<T>* from) throw() {
1      if(isEmpty()) {
2        throw "Empty BST!";
3      }
4
5      BSTNode<T>* ptr = from;
6      while(ptr->right) {
7        ptr = ptr->right;
8      }
9
10     return ptr;
11   }
```

8.2.5 Successore e Predecessore

Un altra importante caratteristica dei BST è che per i nodi è possibile determinare un **successore** e un **predecessore** in base alla chiave che contengono. Analizziamo, per esempio, il seguente albero:



Possiamo affermare, ad esempio, che il successore di 10 è 12, oppure che il successore di 7 è 10. Allo stesso modo possiamo dire che il predecessore di 5 è 4, oppure che il predecessore di 4 e 3. Che algoritmo possiamo utilizzare per trovare successore e predecessore? Prima di vedere come procedere per trovare successore e predecessore, poichè ci servirà risalire l'albero, inseriamo nell'implementazione del nodo, un

puntatore al genitore del nodo, *parent*, in modo tale da poter effettivamente risalire l'albero. Vediamo, quindi, per punti come trovare successore predecessore:

- **successore**: distinguiamo 2 casi:
  - Se il nodo considerato ha un *sottoalbero destro*, allora il successore sarà il **minimo del sottoalbero destro**
  - Se il nodo considerato non ha un sottoalbero destro, allora bisogna risalire l'albero, fino a che trovo l'antenato più prossimo al nodo considerato, risalendo l'albero tramite i *parent*, ovvero i nodi genitori, il cui figlio sinistro è anch'esso antenato di x.

  Per capire meglio vediamo l'implementazione:

```
0   BSTNode<T>* successor(BSTNode<T>* x) {
1   if(this->isEmpty()) {
2     return nullptr;
3   }
4
5   // 1. x ha un sottoalbero destro
6
7   if(x->right)
8     return this->min(x->right); // minimo del sottoalbero
    destro
9
10  // 2. x non ha un sottoalbero destro
11  // il successore di x    l'antenato pi  prossimo di x
12  // il cui figlio sinistro    anche un antenato di x
13
14  BSTNode<T>* y = x->parent;
15
16  //usciremo dal ciclo quando l'antenato y ha come figlio
    sinistro l'antenato x
17  while(x != nullptr && x == y->right) {
18    x = y;
19    y = y->parent;
20  }
21
22  return y;
23
24 }
25
```

Codice 8.3: Successore

- **predecessore**: specularmente, per trovare il predecessore abbiamo 2 casi:
  - Se il nodo considerato ha un *sottoalbero sinistro*, allora il predecessore sarà **il massimo del sottoalbero sinistro del nodo considerato**.
  - Se il nodo considerato non ha un sottoalbero sinistro, allora dobbiamo risalire l'albero fino a trovare l'antenato più prossimo al nodo il cui figlio destro è anch'esso antenato di x.

  L'implementazione sarà dunque la seguente:

```
0     BSTNode<T>* successor(BSTNode<T>* x) {
1     if(this->isEmpty()) {
2       return nullptr;
3     }
4
5     // 1. x ha un sottoalbero sinistro
6
7     if(x->left)
8       return this->max(x->left);
9
10    // 2. x non ha un sottoalbero sinistro
11    // il predecessore di x    l'antenato pi  prossimo di x
12    // il cui figlio destro    anche un antenato di x
13
14    BSTNode<T>* y = x->parent;
15
16    while(x != nullptr && x == y->left) {
17      x = y;
18      y = y->parent;
19    }
20
21    return y;
22
23  }
24
```

8.2.6 Cancellazione di un Nodo

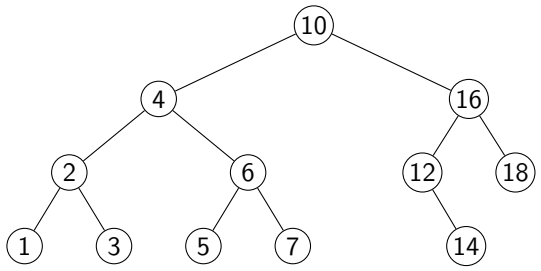
Per quanto riguarda la **cancellazione di un nodo di un BST**, dobbiamo considerare 3 casistiche:

1. Il nodo da eliminare è una **foglia**
2. Il nodo da eliminare ha **un solo figlio**
3. Il nodo da eliminare ha **2 figli**.

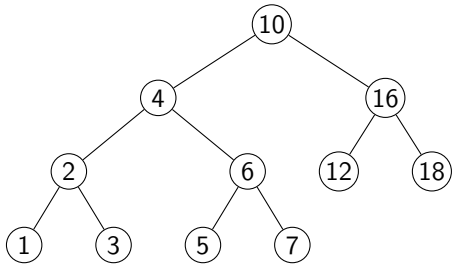
Analizziamo le 3 casistiche, cercando di capire come agire:

1. Nel caso 1 in cui il nodo è una foglia dobbiamo:
   - *rimuovere il collegamento tra il genitore e la foglia*
   - *eliminare la foglia*

   Se avessimo dunque il seguente albero:



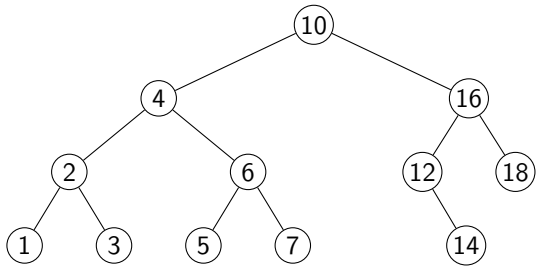
Se eliminassimo il nodo con chiave 14 avremmo il seguente nuovo BST:



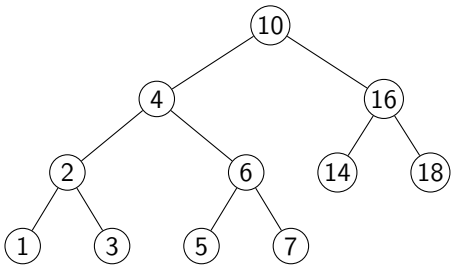
2. Nel caso 2, ovvero quello in cui il nodo ha un figlio, dobbiamo:

   - *collegare l'unico figlio al genitore*
   - *rimuovere i collegamenti dal nodo e per il nodo*
   - *eliminare il nodo*

   Riprendendo l'esempio del precedente BST, ovvero:

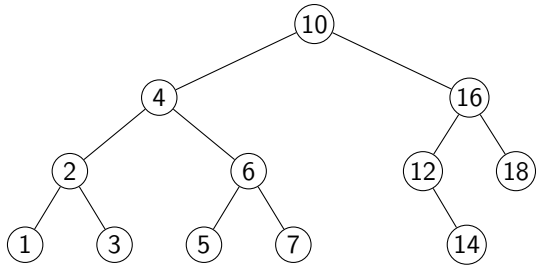


   Se eliminassimo il nodo con chiave 12, che ha un figlio, avremmo il seguente nuovo albero:



3. Nel caso 3, ovvero il caso in cui il nodo ha 2 figli, non vogliamo rimuovere il nodo, poichè altrimenti non esisterebbe più il BST, ma vogliamo *rimuovere la chiave associata al nodo*. La procedura che adottiamo è la seguente: *sostituiamo* il valore da eliminare e *riorganizziamo l'albero* in modo tale da mantenerne le proprietà. In particolare sostituiamo la chiava del nodo con il suo *successore* in modo tale che la proprietà del BST rimanga invariata. Inoltre, poichè il successore si trova sempre nel sottoalbero destro, visto che il nodo ha 2 figli, avrà al più un figlio destro. A questo punto potrò inserire nel nodo che conteneva il successore del nodo che volevamo eliminare, la chiave da eliminare, ed eliminiamo tale nodo, che rientrerà sicuramente nei 2 casi precedenti.

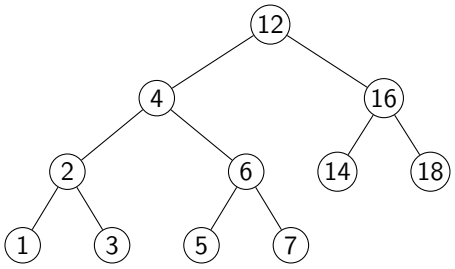
   Quindi, prendendo in considerazione il nostro BST di partenza, ovvero:



   Se eliminassimo ad esempio la radice, che ha chiave 10, dobbiamo:

   - sostituire il 10 con il successore, ovvero 12
   - Eliminare il nodo che ha come figlio 14

   Otteremo dunque il seguente BST:



Implementazione

Per l'implementazione della cancellazione facciamo un overloading di una stessa funzione, in cui nella prima ritorniamo un puntatore a un nodo, e studiamo i primi 2 casi di cancellazione, ovvero rispettivamente quando il nodo è una foglia, e quando il nodo ha un figlio:

- Nel primo caso dobbiamo imporre, se il nodo da eliminare è un figlio sinistro, il figlio sinistro del genitore a nullptr. Specularmente nel caso in cui il nodo da eliminare è un figlio destro
- Nel secondo caso dobbiamo collegare il nodo figlio al padre del nodo da eliminare. Poi se il figlio del nodo da eliminare è sinistro, imporremo come figlio sinistro del padre del nodo da eliminare il figlio sinistro del nodo da eliminare. Stesso procedimento anche per il figlio destro.

In entrambi i casi ritorneremo comunque il nodo che stiamo andando ad eliminare che ci servirà nel secondo metodo. In tale metodo gestiamo anche il terzo caso, ovvero quando il nodo ha 2 figli. In tal caso ricerchiamo il nodo con la chiave passata come parametro. Se tale nodo non esiste ritorniamo *nullptr*, altrimenti richiamiamo il metodo precedente per vedere se la cancellazione di tale nodo rientri nei primi 2 casi. Se non dovesse rientrare nei primi 2 casi, allora ricerchiamo il nodo successore, facciamo uno swap tra le chiavi del nodo e del successore, ed eliminiamo il successore, che sicuramente rientrerà nei primi 2 casi di cancellazione. L'implementazione è la seguente:

```
0  BSTNode<T>* remove(BSTNode<T>* node) {
1      //CASO 1
2      //il nodo    una foglia e coincide con la radice
3      if(node == root && node->left == nullptr && node->right ==
       nullptr) {
4              root = nullptr;
5              return node;
6      }
7
8      //Il nodo    una foglia e non coincide con la radice
9      if(node->left == nullptr && node->right == nullptr) {
10       if(node == node->parent->left)
11         node->parent->left = nullptr;
12       else if (node == node->parent->right)
13         node->parent->right = nullptr;
14
15       return node;
16     }
17
18
19     //CASO 2
20     //il nodo da eliminare ha solo un figlio destro
21     if(node->left == nullptr && node->right != nullptr) {
22       node->right->parent = node->parent;
23
24       //il nodo da eliminare    figlio sx
25       if(node == node->parent->left) {
26         node->parent->left = node->right;
```

```
27        }
28        //il nodo da eliminare    figlio dx
29        else if(node == node->parent->right) {
30          node->parent->right = node->right;
31        }
32        return node;
33      }
34
35      //il nodo da eliminare ha solo un figlio sinistro
36      if(node->left != nullptr && node->right == nullptr) {
37        node->left->parent = node->parent;
38
39        //il nodo da eliminare    figlio sx
40        if(node == node->parent->left) {
41          node->parent->left = node->left;
42        }
43        //il nodo da eliminare    figlio dx
44        else if(node == node->parent->right) {
45          node->parent->right = node->left;
46        }
47        return node;
48      }
49
50      return nullptr;
51    }
52
53    BSTNode<T>* remove(T key) {
54      if(this->isEmpty()) {
55        return nullptr;
56      }
57
58      BSTNode<T>* node = this->search(key);
59      if(node == nullptr)
60        return nullptr;
61
62      BSTNode<T>* toDelete = this->remove(node);
63
64      if(toDelete != nullptr)
65        return toDelete;
66
67      //CASO 3
68      //il nodo da eliminare ha due figli
69      //sostituiamo la chiave nel nodo da eliminare con la chiave del
         suo successore
70      BSTNode<T>* next = this->successor(node);
71      //sostituzione della chiave
72      T swap = node->key;
73      node->key = next ->key;
74      next->key = swap;
75
76      //richiamo la procedura di cancellazione (casi 1 e 2) sul
        successore
77      toDelete = this->remove(next);
78
79      return toDelete;
80    }
```

Codice 8.4: Cancellazione di un nodo da un BST

Notiamo di aver fatto una piccola aggiunta per il caso 1, ovvero il caso in cui il nodo da eliminare sia una foglia e sia la radice stessa: in tal caso, poichè non esiste il parent per la radice, essa va semplicemente posta a *nullptr*.

Complessità dell'operazione di cancellazione su un BST

Come visto per ricerca e visita, anche per quanto riguarda la **cancellazione** la complessità è dell'ordine di $O(h)$, dove h rappresenta l'altezza dell'albero.

Capitolo 9

Grafi

L'ultimo tipo struttura dati che andiamo a vedere sono i **grafi**. Tale struttura dati è una delle più importanti in ambito informatico, poichè possiamo dire che è possibile rappresentare diversi tipi di algoritmi sulle reti, database, networking... E inoltre tale struttura ci permette essenzialmente di rappresentare tutte le altre strutture dati che abbiamo visto data la sua estrema flessibilità.

9.1 Definizioni e Terminologia

Prima di cimentarci nell'implementare i grafi, vediamo di capire tramite qualche definizione cosa sia effettivamente un grafo. Un grafo $G = (V, E)$ è un insieme di nodi collegati da archi che possono essere direzionati o meno. In tal caso:

- L'insieme V rappresenta l'insieme di **nodi**
- L'insieme E rappresenta l'insieme di **archi** che collegano i nodi

Altre definizioni concernenti i grafi sono:

- 2 nodi v_i, v_j sono **adiacenti** quando vi è un arco che li collega
- Un grafo si dice **sparso** quando $|E| << |V|^2$, ovvero la cardinalità dell'insieme di archi è molto minore della cardinalità dell'insieme di vertici al quadrato
- Un grafo si dice **denso** quando $|E| \approx |V|^2$, ovvero la cardinalità dell'insieme di archi si avvicina alla cardinalità dell'insieme di vertici al quadrato

Possiamo dire, inoltre, che il BST è uno speciale tipo di grafo orientato, e che in generale gli alberi sono particolari grafi.

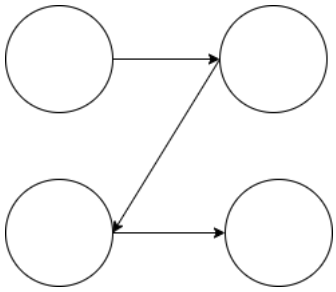


Figure 9.1: Esempio di grafo sparso

9.2 Implementazione

Per quanto rigurarda l'implementazione di un grafo ci serve avere le seguenti informazioni sui nodi:

- Informazioni sui **nodi adiacenti**
- Sapere se il **numero di nodi** è **limitato** o meno

Tramite queste informazioni sceglieremo il modo più adatto di implementare il grafo, che può avvenire tramite **matrici di adiacenza** o **liste di adiacenza**:

- La matrice di adiacenza può essere una matrice di *booleani*, ovvero di 0 e 1, dove se l'elemento $M[i][j]$ è impostato ad 1, significa che il nodo i è collegato da un arco al nodo j, ovvero i è *adiacente* a j. In particolare, se il grafo è *non orientato*, anche $M[j][i]$ è impostato ad 1, in quanto proprio per la proprietà di essere un grafo non orientato se i è collegato a j, allora anche j è collegato ad i. Si dice in tal caso, quindi, che la matrice è **simmetrica**, ovvero $M[i][j] = M[j][i]$. Possiamo dire che la matrice di adiacenza è sicuramente la scelta migliore per un grafo di cui sappiamo che vi è un numero limitato di nodi. Inoltre è la scelta indicata per un *grafo denso*: infatti per un grafo denso lo spazio di $|V| \cdot |V|$ per allocare la matrice viene quasi interamente occupato, in quanto $|E| \approx |V|^2$ e non andiamo a sprecare spazio. In un *grafo sparso*, proprio perchè $|E| << |V|^2$ andremmo a sprecare uno spazio ben maggiore.
- Le liste di adiacenza, invece, per ogni nodo, rappresentano tutti i *nodi adiacenti* al nodo considerato inseriti in una lista. In tal caso l'utilizzo delle liste di adiacenza si basa sul fatto che non sappiamo a priori il numero di nodi da inserire, e quindi abbiamo bisogno di una struttura dinamica, e inoltre è molto indicata per un *grafo sparso*, e quindi avremo delle liste più corte con un tempo di accesso agli elementi minore, ma a differenza delle matrici, non costante.

Possiamo inoltre dire che la complessità per tali strutture è la seguente:

- *matrici di adiacenza*: $\Theta(|V|^2)$, e inoltre avremo un tempo di accesso costante agli elementi della matrice
- *liste di adiacenza*: $\Theta(|V| + |E|)$

Implementazione tramite Matrice di Adiacenza

Nell'implementazione tramite *matrice di adiacenza* le proprietà che ci servono all'interno della classe sono le seguenti:

- **Nodo**: l'unico attributo del nodo in tal caso è solo la *chiave* visto che non abbiamo bisogno di avere informazioni sul nodo successivo o precedente, visto che stiamo parlando del grafo.
- **Grafo**: per l'implementazione del grafo ci serviamo di una *matrice di booleani* di dimensione $N \cdot N$, che ci da le informazioni su come i nodi sono collegati; e un array di puntatori a nodo di dimensione N che rappresenta la lista dei N nodi del grafo: tale array ci da informazioni sugli indici che dovremmo poi imporre a 1 nella matrice, e inoltre all'interno di un grafo non sempre tutti i nodi sono collegati, di conseguenza con tale array possiamo tenere traccia di tutti i nodi inseriti. Quindi, utilizzando un metodo per l'aggiunta dei nodi e uno per l'aggiunta degli archi, quando aggiungeremo un nuovo nodo, lo aggiungeremo all'array di puntatori a nodi; mentre quando aggiungeremo un arco tra due nodi (passando le 2 chiavi come parametro), assegneremo 1 alla locazione della matrice corrispondente esattamente agli indici dell'array di puntatori a nodi nel quale sono collocati tali nodi. I metodi che quindi andiamo ad implementare sono i seguenti:
  - *search*: ricerca l'indice nell'array di nodi associato alla chiave passata come parametro. Se non viene trovato ritorna -1. Tale metodo ci servirà poi nell'individuare all'interno della matrice i nodi da collegare.
  - *insertNode*: inseriamo all'interno dell'array di nodi un nuovo nodo con la chiave passata come parametro e incrementiamo l'indice *last*.
  - *insertArc*: passando come parametro 2 chiavi, ricerchiamo i loro indici associati e, se non vengono trovati non colleghiamo nessun nodo, altrimenti imponiamo la matrice, con i rispettivi indici che abbiamo ricercato, a true.

Vediamo quindi una prima implementazione:

```
0
1 //Classe Nodo del grafo
2 template<typename T>
3 class Node{
4     private:
5         T key;
6
7
8         template<typename U>
9         friend class Graph;
```

```
10      public:
11          Node(T key): key(key) {}
12  };
13
14
15  template<typename T>
16  class Graph{
17      private:
18          bool** matrix;
19          int dim;
20          int last; // rappresenta l'indice dell'ultimo elemento dell
    'array di puntatori
21          Node<T>** array;
22
23      public:
24          Graph(int _dim = 1000): dim(_dim), last(0) {
25              array = new Node<T>*[this->dim];
26              matrix = new bool*[this->dim];
27              for(int i=0; i<dim; i++) {
28                  matrix[i] = new bool[dim];
29                  for(int j=0; j<dim; j++) {
30                      matrix[i][j] = false;
31                  }
32              }
33          }
34
35          //Metodo di ricerca dell'indice del nodo all'interno dell'
    array di puntatori a nodo
36          int search(T key) {
37              if(this->last == 0)
38                  return -1;
39
40              for(int i=0; i<last; i++) {
41                  if(array[i]->key == key) {
42                      return i;
43                  }
44              }
45
46              return -1;
47          }
48
49          //Metodo di inserimento del nodo
50          void insertNode(T key) {
51              array[last++] = new Node<T>(key);
52          }
53
54
55          //Metodo di inserimento dell'arco fra 2 nodi
56          void addEdge(T key1, T key2) {
57              int index = search(key1), index1 = search(key2);
58
59
60              if(index == -1 || index1 == -1) {
61                  cout << "No existing keys with values passed as
    parameters" << endl;
62                  return;
63              }else
```

```
64                  matrix[index][index1] = true;
65          }
66
67
```

Codice 9.1: Grafo tramite Matrice di Adiacenza

Implementazione tramite Liste di Adiacenza

Organizziamo l'implementazione tramite liste di adiacenza in tal modo:

- Creiamo una classe *GraphVertex*, ovvero la classe rappresentante il nodo del grafo. Poichè, dunque, l'implementazione avviene tramite liste di adiacenza, associamo ad ogni nodo del grafo una lista dei nodi adiacenti al nodo considerato. Per questo facciamo in modo che tale classe erediti dalla classe lista, e ogni volta che inseriamo un nuovo nodo adiacente, lo aggiungiamo in coda alla lista, così da formare la **lista di adiacenza** del nodo. Essenzialmente, quindi, l'implementazione di tale classe è la seguente:

```
0 template<typename T>
1 class GraphVertex: public List<T>{
2
3     public:
4         GraphVertex(T key): List<T>() {
5             List<T>::insertTail(key);
6         }
7 };
8
```

- Dopo aver creato la classe *GraphVertex*, dobbiamo creare la classe rappresentante il grafo, che chiamiamo *GraphList*: il nome è indicativo, infatti andiamo a creare una lista di *GraphVertex*, che chiamiamo *vertices*, che è una lista di liste. I metodi presenti al suo interno sono:

  - *search()*: a differenza di quanto visto prima con le matrici, nel quale andavamo a ricercare l'indice associato al nodo ricercato, questa volta andiamo a ricercare il nodo, all'interno di vertices, associato alla chiave passata come parametro.
  - *addVertex()*: passando la chiave associata al nuovo nodo da inserire, creiamo una nuova istanza di *GraphVertex* avente tale chiave, e andiamo ad inserire in coda in *vertices* tale vertice.
  - *addEdge()*: passiamo le 2 chiavi dei nodi che vogliamo collegare. Ricerchiamo il vertice associato al primo nodo, e inseriamo il vertice associato al secondo in coda nella lista di adiacenza del primo nodo.

  L'implementazione sarà dunque la seguente:

```
template<typename T>
class GraphList{
    List< GraphVertex<T> > vertices;
    int nVertices; // numero di vertici del grafo
    bool isOriented;

    public:
        GraphList(bool isOriented = true): nVertices(0),
    isOriented(isOriented) {}

        void addVertex(T key) {
            GraphVertex<T> toInsert(key);
            vertices.insertTail(toInsert);
            nVertices++;
        }

        void addEdge(T key1, T key2) {
            Node<GraphVertex<T> >* node1 = this->search(key1);
            Node<GraphVertex<T> >* node2 = this->search(key2);

            if(node1 && node2) {
                node1->getVal().insertTail(key2);
                if(!(this->isOriented)) {
                    node2->getVal().insertTail(key1);
                }
            }
        }


        Node< GraphVertex<T> >* search(T key) {
            if(nVertices == 0)
                return NULL;

            Node< GraphVertex<T> >* ptr = vertices.getHead();
            while(ptr) {
                if(key == ptr->getVal().getHead()->getVal())
                    return ptr;
                ptr = ptr->getNext();
            }

            return NULL;
        }

```

9.3 Visita di un Grafo

Andiamo ad introdurre adesso degli algoritmi di **visita di un grafo**. Per i grafi esistono 2 tipi di algoritmi di visita:

- **Visita in ampiezza** (*Breadth First Search*)
- **Visita in profondità** (*Depth First Search*)

In tale sezione andremo a vedere come funzionano tali algoritmi, ma non andremo effettivamente a implementarli.

9.3.1 Visita in Ampiezza

Nella visita in ampiezza il nostro obiettivo è quello di *esplorare* tutti i possibili nodi adiacenti ad ogni nodo, che chiameremo *nodo sorgente*. Poichè esploriamo il grafo scoprendo ogni vertice adiacente a ogni nodo, la visita viene chiamata in ampiezza. Per realizzare dunque tale algoritmo, *coloriamo i nodi* con determinati colori che ci indicano se il nodo è stato scoperto, visitato o esplorato e utilizziamo 2 array:

- un array di *predecessori*: ovvero per ogni nodo definiamo un predecessore
- un array di *discovery* che calcola la distanza del nodo dal nodo sorgente.

Coloriamo i nodi nel seguente modo:

- Se il nodo *non è stato ancora scoperto*, ovvero non è stato visitato neanche una volta, allora avrà colore **bianco**
- Se il nodo è stato scoperto ma *non è stato esplorato*, ovvero è stata effettuata la visita su quel nodo ma non sono stati scoperti ancora i nodi adiacenti, allora tale nodo avrà colore **grigio**
- Se il nodo *è stato esplorato*, ovvero sono stati scoperti tutti i suoi nodi adiacenti, allora avrà colore **nero**.

Quindi l'idea dell'algoritmo è:

- Scegliere un **nodo sorgente** casuale e colorare tutti i nodi inizialmente di bianco.
- Utilizzare una coda nel quale inseriamo i nodi grigi, e dal quale spiliamo i nodi che vengono colorati neri. Ogni volta che spiliamo un nodo grigio dalla coda, sapremo che il nodo che abbiamo spilato, avrà tutti i suoi nodi adiacenti colorati di grigio. Inoltre aggiorniamo per ogni nodo adiacente l'array discovery di tali nodi sommando 1 al discovery del nodo considerato.
- Infine, una volta che tutti i nodi adiacenti sono stati colorati di grigio, e l'array discovery è stato aggiornato, allora si colora di nero il nodo considerato.

L'algoritmo in pseudocodice è il seguente:

Algorithm 4 BFS(breadth first search)

```
BFS(G, s)                          ▷ G rappresenta il grafo ed s il nodo sorgente
for ogni vertice u in G do
    color[u] = white
    d[u] = inf
    pred[u] = NULL
end for
color[s] = grey
d[s] = 0
pred[s] = null
Q.enqueue(s)
while Q ≠ ∅ do
    u = Q.dequeue()
    for ogni vertice v nella lista di adiacenza di u do
        if color[v] == white then
            color[v] = grey
            d[v] = d[u] + 1
            pred[v] = u
            Q.enqueue(v)
        end if
    end for
    color[u] = black
```

9.3.2 Visita in Profondità

A differenza di quanto visto nella *visita in ampiezza* (BFS), nella **visita in profondità**, o **DFS** (*depth first search*), il nostro obiettivo sarà quello di esplorare i nodi andando in profondità, quindi non consideriamo un nodo sorgente, ma esploriamo tutti gli archi uscenti dal nodo, fino a quando non ve ne sono più. A questo punto, quando terminano gli archi uscenti dal nodo v, si esplorano eventuali archi uscenti del nodo precedente a v. I nodi inoltre vengono colorati inoltre con lo stesso criterio visto nella BFS, ovvero:

- Un nodo è **bianco** quando non è stato ancora scoperto
- Un nodo è **grigio** quando è stato scoperto ma non ancora esplorato
- Un nodo è **nero** quando è stato esplorato

Inoltre definiamo:

- Un array di **predecessori**, dove se un nodo v è stato scoperto esplorando la lista di adiacenza di un nodo u, allora il predecessore di v è u.
- Una variabile *time* che contiene il tempo attuale e si incrementa di 1 ogni volta che scopriamo un nodo (nodo che da bianco passa ad essere grigio), e quando abbiamo esplorato tutta la lista di adiacenza del nodo (nodo che da

grigio passa ad essere nero). Per questo ci serviamo di 2 array che marcano temporalmente la visita di ogni nodo, ovvero:

- Un array **discovery**, che marca il tempo, utilizzando la variabile time, in cui il nodo è stato scoperto, ovvero quando passa da essere bianco a grigio
- Un array **final**, che marca il tempo in cui il nodo è stato esplorato, ovvero abbiamo visitato tutta la sua lista di adiacenza e passa da essere grigio a nero.

L'algoritmo quindi si sviluppa nel seguente modo:

- Impostiamo tutti i nodi del grafo *bianchi* e i predecessori nulli. Impostiamo inoltre il *time* a 0
- Infine eseguiamo la visita per ogni nodo bianco del grafo. Tale visita avviene ricorsivamente nel seguente modo, che è la procedura *DFS-visit*:
  - Quando chiamiamo la visita su un nodo, lo coloriamo di *grigio*, marchiamo il tempo di scoperta in *discovery*, incrementiamo il time.
  - Per ogni nodo v della lista di adiacenza del nodo considerato u, impostiamo u come predecessore di v e richiamiamo la visita su v, cosicchè la visita possa essere fatta in profondità.
  - Infine, dopo aver esplorato tutta la lista di adiacenza di u, impostiamo il suo colore a nero, marchiamo il tempo in *final* e incrementiamo il tempo.

Lo pseudo codice della DFS è il seguente:

Algorithm 5 DFS

```
DFS(G)
for ogni vertice u in G do
    color[u] = white
    pred[u] = NULL
end for
time = 0
for ogni vertice u in G do
    if color[u] == white then
        DFS − visit(u)
```