

Appunti Algoritmi e Strutture Di Dati

by Picciau Daniele

Indice

1	Introduzione	3
1.1	Problemi computazionali e algoritmi	3
1.1.1	Esempi di problemi computazionali e algoritmi	3
1.1.2	Come valutare l'algoritmo scelto	4
1.1.3	Complessità di un algoritmo	5
1.2	Strutture dati	5
1.2.1	Tipologie di strutture dati	6
1.2.2	Le sequenze	6
1.2.3	Gli Insiemi	7
1.2.4	I dizionari	8
1.3	I puntatori	9
1.3.1	Puntatori nulli	9
2	Iterazione, induzione e ricorsione	11
2.1	Iterazione	11
2.2	Induzione	11
2.2.1	Induzione completa	12
2.3	Ricorsione	13
2.4	Invarianti dei cicli	13
2.4.1	Invariante del ciclo while VS invariante del ciclo for	14
2.4.2	Invarianti dei cicli - Esempio pratico con il selectionSort()	16
3	Le liste	19
3.1	Liste concatenate	20
3.1.1	Inserimento in testa	20
3.1.2	Inserimento generico	21
3.1.3	Cancellazione elemento corrente	23
3.1.4	Creazione lista	25
3.1.5	Cancellazione intera lista	25
3.1.6	Attraversamento con funzione	26
3.1.7	Ricerca elemento con funzione	27

4	Alberi	28
4.1	Terminologia	29
4.2	Alberi binari	29
4.2.1	Creazione di un albero binario	30
4.2.2	Costruzione di un albero binario	30
4.3	Le visite	31
4.3.1	Visita in preorder (DFS)	32
4.3.2	Visita in postorder (DFS)	33
4.3.3	Visita inorder (DFS)	35
4.3.4	Attraversamento di un albero binario in C	36
4.3.5	Cancellazione di un albero	37
4.4	Alberi binari di ricerca (BST)	39
4.4.1	Implementazione della ricerca	40
4.4.2	Creazione di un nuovo nodo	42
4.4.3	Inserimento di un nodo	43
4.4.4	Ricerca del massimo e del minimo	45
4.4.5	Ricerca del successore-predecessore	48
4.4.6	Cancellazione di un nodo	51
4.4.7	Alberi BST: alcune osservazioni	54
4.4.8	Alberi BST: Alberi Adelson-Velsky and Landis (AVL)	55
5	Analisi della complessità degli algoritmi	58
5.1	Complessità e dimensione dell'input	58
5.2	Definizione di tempo e modello di calcolo	59
5.2.1	Random Access Machine (RAM)	59
5.3	Valutazione del caso pessimo, medio e ottimo	60
5.3.1	Tempo di calcolo della funzione <code>min()</code> (iterativa)	60
5.3.2	Tempo di calcolo della funzione <code>binarySearch()</code> (ricorsiva)	61
5.4	Ordini di complessità	62
5.4.1	Principali classi di efficienza asintotiche	63
5.4.2	Notazioni asintotiche	63
5.5	Le ricorrenze	64
5.5.1	Metodo di sostituzione (o per tentativi)	65
5.5.2	Metodo dell'esperto	65
5.5.3	Metodo dell'albero di ricorsione (analisi per livelli)	66
5.6	Ordinamento	67
5.6.1	Selection Sort	68
5.6.2	Insertion sort	69
5.6.3	Merge sort	71

1 Introduzione

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

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

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

1.1 Problemi computazionali e algoritmi

Cos'è un problema computazionale?

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

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

Cos'è un algoritmo?

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

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

1.1.1 Esempi di problemi computazionali e algoritmi

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

Problema del minimo

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

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

Problema di ricerca

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

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

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

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

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

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

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

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

1.1.2 Come valutare l'algoritmo scelto

Valutazione di un algoritmo

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

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

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

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

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

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

1.1.3 Complessità di un algoritmo

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

Complessità di un algoritmo

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

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

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

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

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

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

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

1.2 Strutture dati

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

Cos'è una struttura dati?

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

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

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

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

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

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

1.2.1 Tipologie di strutture dati

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

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

1.2.2 Le sequenze

Cos'è una sequenza?

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

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

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

SEQUENCE

% Restituisce **true** se la sequenza è vuota

boolean isEmpty()

% Restituisce **true** se p è uguale a pos_0 oppure a pos_{n+1}

boolean finished(POS p)

% Restituisce la posizione del primo elemento

POS head()

% Restituisce la posizione dell'ultimo elemento

POS tail()

% Restituisce la posizione dell'elemento che segue p

POS next(POS p)

% Restituisce la posizione dell'elemento che precede p

POS prev(POS p)

SEQUENCE (continua)

% Inserisce l'elemento v di tipo ITEM nella posizione p .

% Restituisce la posizione del nuovo elemento, che diviene il predecessore di p

POS insert(POS p , ITEM v)

% Rimuove l'elemento contenuto nella posizione p .

% Restituisce la posizione del successore di p ,

% che diviene successore del predecessore di p

POS remove(POS p)

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

ITEM read(POS p)

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

write(POS p , ITEM v)

Esempio di creazione di una sequenza il linguaggio C++

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

Result: [1, 2, 3]

1.2.3 Gli Insiemi

Cos'è un insieme?

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

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

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

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

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

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

SET

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

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

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

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

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

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

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

Esempio di creazione di un insieme il linguaggio C++

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

1.2.4 I dizionari

Cos'è un dizionario?

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

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

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

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

DICTIONARY

% Restituisce il valore associato alla chiave k se presente, **nil** altrimenti
ITEM lookup(ITEM k)

% Associa il valore v alla chiave k
insert(ITEM k , ITEM v)

% Rimuove l'associazione della chiave k
remove(ITEM k)

1.3 I puntatori

Cos'è un puntatore?

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

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

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

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

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

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

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

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

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

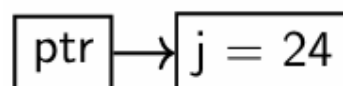
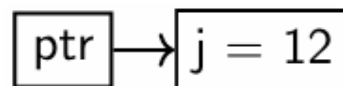
```
value = *balptr //se balptr contiene l'indirizzo di balance, il valore della variabile balance viene messo in value.
```

Esempio: esercizio con puntatori

```
#include <iostream>
int main() {
    int j = 12;
    int *ptr = &j;

    cout << *ptr << endl;

    j = 24;
    cout << *ptr << endl;
    cout << ptr << endl;
    return 0;
}
```



L'output del programma è il seguente:

12

24

0x7b03a928 (indirizzo di memoria)

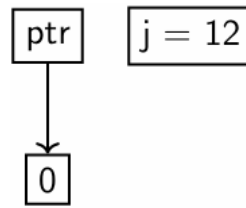
1.3.1 Puntatori nulli

Cos'è un puntatore?

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

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

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



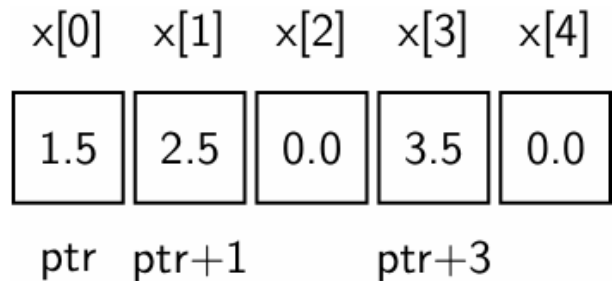
Segmentation violation (core dumped)!

Esempio: come i puntatori vengono utilizzati con gli array

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

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

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



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

2 Iterazione, induzione e ricorsione

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

2.1 Iterazione

Cosa si intende con iterazione?

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

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

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

2.2 Induzione

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

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

Cosa si intende con induzione?

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

Cos'è un asserto?

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

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

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

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

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

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



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

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

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

2.2.1 Induzione completa

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

Cosa si intende con induzione completa?

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

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

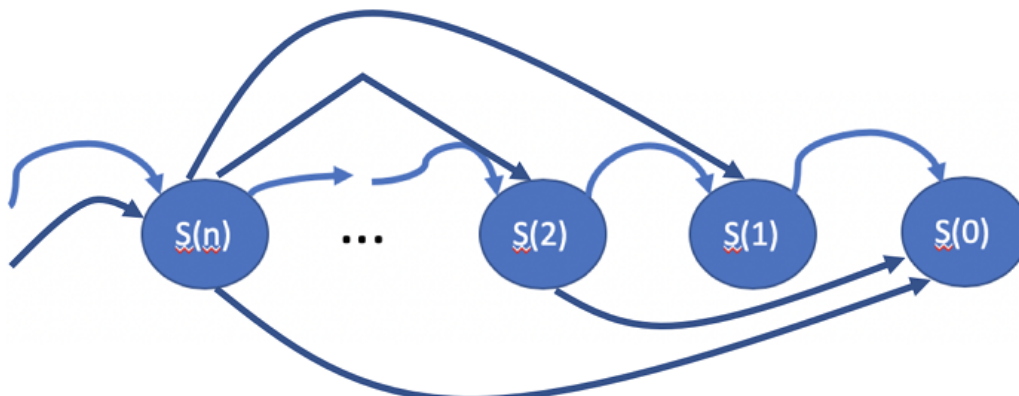
Anche in questo caso, per ottenere una dimostrazione induttiva dell'asserto $S(n)$ si utilizzano due passi:

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

Utilità dell'induzione completa

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

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



2.3 Ricorsione

Cosa si intende con ricorsione?

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

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

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

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

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

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

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

Induttività della ricorsione

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

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

Esempio: si immagini di voler calcolare il fattoriale di 4

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

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

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

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

2.4 Invarianti dei cicli

Cos'è un invariante?

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

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

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

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

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

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

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

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

2.4.1 Invariante del ciclo `while` VS invariante del ciclo `for`

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

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

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

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

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

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

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

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

Primo caso

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

Secondo caso

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

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

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

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

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

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

Esempio: dimostrazione della terminazione di un ciclo *while*

```

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

```

La seguente funzione "Fattoriale" calcola $n!$ passato da parametro, assumendo che $n \geq 1$. Lo scopo è quello di dimostrare che il ciclo *while* (righe 4-6) deve terminare.

Come detto precedentemente, per dimostrare la terminazione del ciclo bisogna determinare E .

Dunque, si cerca una grandezza che:

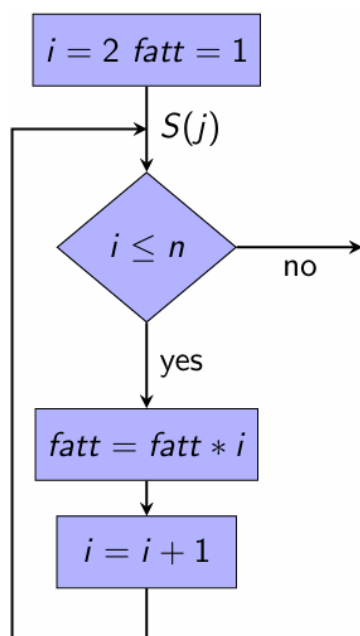
- parta positiva;
- diminuisca ad ogni iterazione;
- diventi negativa quando il ciclo finisce.

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

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

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

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



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

Cosa afferma l'asserto?

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

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

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

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

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

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

Obiettivo del `selectionSort()`

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

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

```

SelectionSort(ITEM[] A, integer n)
1 integer i, j, p
2 for i ← 1 to n - 1 do
3   p ← i
4   for j ← i + 1 to n do
5     if A[j] < A[p] then
6       p ← j
7   % A questo punto, p è l'indice del primo elemento più piccolo in
   A[i...n]
8   % Scambiamo allora A[p] e A[i]
9   A[p] ↔ A[i]

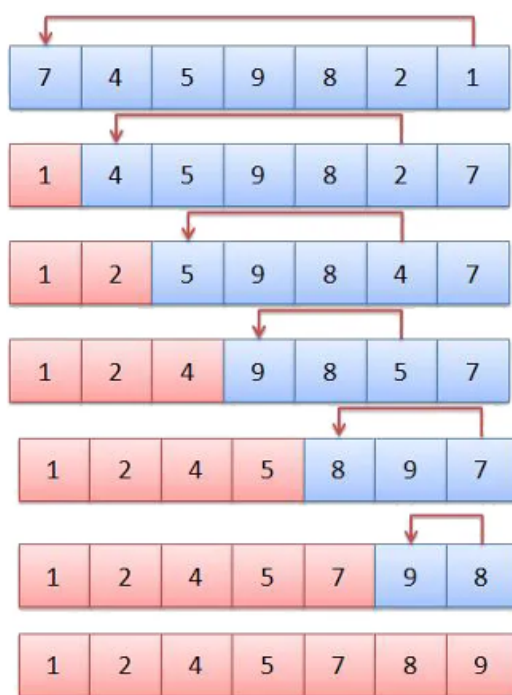
```

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

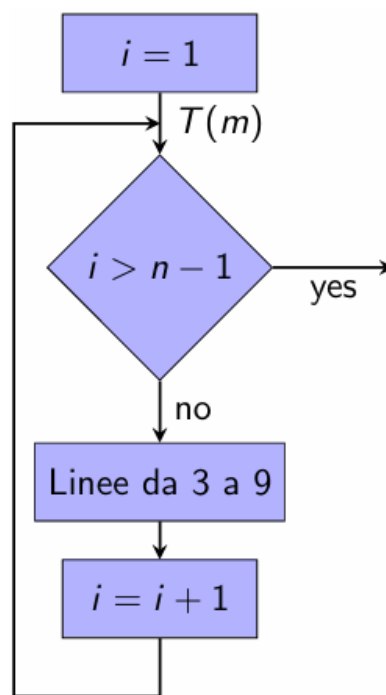
1. Nella prima iterazione troviamo (selezioniamo) uno degli elementi più piccoli tra tutti i valori in $A[1...n]$ e lo scambiamo con $A[1]$;
2. Nella seconda iterazione troviamo uno degli elementi più piccoli tra tutti i valori in $A[2...n]$ e lo scambiamo con $A[2]$;
3. Dopo l'iterazione i -esima, $A[1...i]$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

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

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



(a)



(b)

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

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

Esempio: dimostrazione tramite induzione

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

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

Cosa afferma l'asserto?

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

- Gli elementi $A[1...(m-1)]$ sono ordinati in senso non decrescente:
 $A[1] \leq A[2] \leq \dots \leq A[m-1]$;
- Tutti gli elementi di $A[m...n]$ sono maggiori o uguali a ogni elemento di $A[1...(m-1)]$.

1. **Caso base:** quando $i = m = 1$ siamo all'inizio dell'algoritmo, e non abbiamo ancora ordinato nulla. Per vedere se $T(1)$ è vera si controlla che l'asserto sia corretto:

- La parte (1) di $T(1)$ dice che $A[1...m-1] = A[1...0]$ è ordinato, ma $A[1...0]$ è vuoto, quindi è banalmente vero (una sequenza vuota è sempre ordinata).
- La parte (2) di $T(2)$ dice che tutti gli elementi in $A[m...n] = A[1...n]$ sono maggiori o uguali a quelli in $A[1...m-1] = A[1...0]$, ma anche in questo caso gli elementi in $A[1...0]$ non esistono, quindi la condizione è automaticamente vera.

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

2. **Passo induttivo:** in questo passaggio si assume che $T(m)$ sia vera per un generico $m \leq n-1$, che verifica le condizioni dell'asserto, quindi:

- La prima parte dell'array $A[1...(m-1)]$ è già ordinata;
- Tutti gli elementi compresi in $A[1...(m-1)]$ sono \leq di ogni elemento compreso in $A[m...n]$.

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

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

Anche dopo lo scambio rimane vero che:

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

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

3. **Conclusione:** nell'ultima iterazione, quando $i = m = n$

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

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

3 Le liste

Cos'è una lista (linked list)?

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

Contiguità delle liste

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

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

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

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

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

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

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

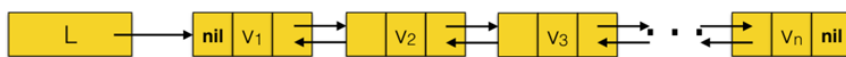
1. creare il nodo x ;
2. impostare $x.\text{next} = b$;
3. impostare $a.\text{next} = x$.

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

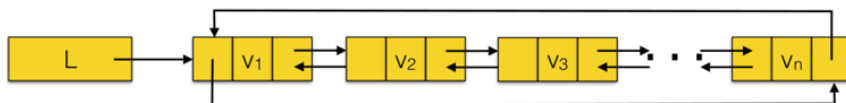
Al contrario, l'accesso a un elemento tramite il suo indice richiede di scorrere la lista dall'inizio fino al nodo desiderato, con un costo $O(n)$.



Monodirezionale



Bidirezionale



Bidirezionale circolare



Monodirezionale con sentinella

Le linked list presentano **diverse possibili implementazioni**:

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

3.1 Liste concatenate

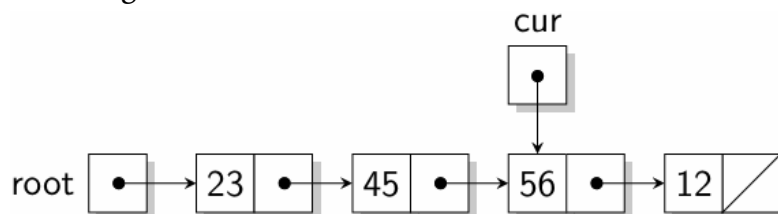
Come sono strutturate?

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

Le liste concatenate vengono definite nel seguente modo:

```
struct slist {  
    int dato;  
    struct slist *next;  
};
```

```
/* lista vuota */  
struct slist *root = NULL;
```



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

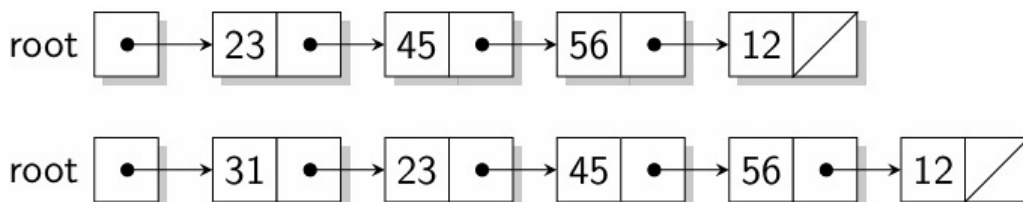
Per manipolare una lista semplicemente concatenata esistono **diverse funzioni**, tra cui:

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

3.1.1 Inserimento in testa

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

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



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

Esempio di utilizzo

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

3.1.2 Inserimento generico

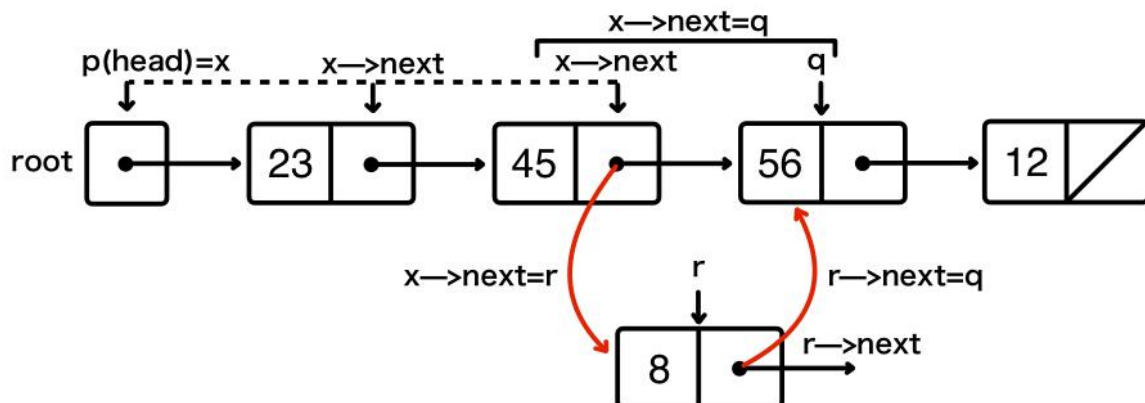
Questa funzione è un passo avanti rispetto alla `insert()`. `insert2()` permette un inserimento generico in qualunque posizione della lista (in testa, in mezzo o in coda).

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

```

21     return p;
22 }
23 }

```



- **Tipo di ritorno:** La funzione restituisce un puntatore a `slist`, ovvero il puntatore alla testa aggiornata della lista.
- **Parametri:** i parametri definiti nella funzione sono (`slist *p`, `slist *q`, `int elem`), rispettivamente:
 - il **puntatore corrente** alla testa della lista (che può essere `null` se la lista è vuota);
 - il **puntatore al nodo di riferimento** (davanti al quale inserire) oppure `null` se si vuole inserire in coda;
 - il **valore da inserire** nel nuovo nodo.
- **Funzionamento del codice:** per poter inserire un nuovo nodo in testa alla lista, supponiamo di possedere una struttura dati come quella definita al capitolo 3.1.
 - *riga 2:* `x` è un puntatore alla lista che serve come supporto per eseguire lo scorrimento della stessa;
 - *riga 3:* creo un puntatore `*r` al nuovo nodo di grandezza `slist`;
 - *riga 4-7:* eseguo un crontollo su `r` per assicurarmi che la `malloc` sia andata a buon fine;
 - *riga 8:* inizializzo il campo dato del nuovo nodo con il valore del campo `elem` passato tra i parametri;
 - *riga 9-12:* viene **gestito l'inserimento in testa**. Infatti, se il nodo `q` coincide con `p`, mi basta semplicemente collegare il nuovo nodo alla vecchia testa (`r -> next = p`), e poi restituire il nuovo nodo (`return r`);
 - *riga 13-21:* viene **gestito l'inserimento in mezzo o in coda**.
Tramite il ciclo `for` si fa scorrere il puntatore di appoggio (`x`) partendo dalla testa (`p`) della lista. Lo scorrimento continua fintanto che è verificata la condizione `x->next != q`, e **presenta due possibilità di terminazione**:
 1. La condizione diventa `x->next == q` poiché `q` è un nodo interno alla lista. In altre parole quando `x` precede il nodo `q` specificato nei parametri, si esce dal ciclo;
 2. La condizione non si avvera perché il `q` passato nei parametri è `null`, e il ciclo termina le sue iterazioni.

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

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

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

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

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

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

Esempio di utilizzo

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

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

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

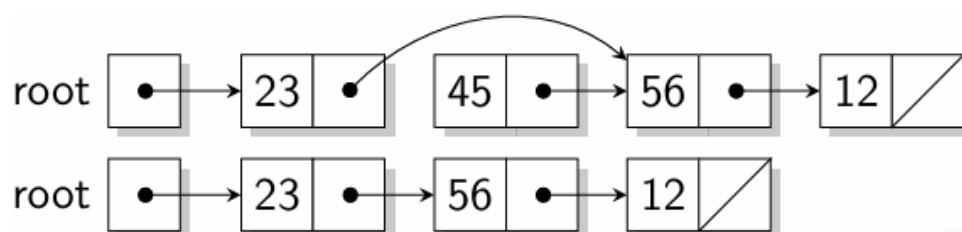
// Inserimento in coda (q == NULL)
head = insert2(head, NULL, 12);    // inserisco 12 alla fine
printList(head);    // lista: 23 -> 45 -> 8 -> 56 -> 12

return 0;
```

3.1.3 Cancellazione elemento corrente

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

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



- **Tipo di ritorno:** la funzione restituisce un puntatore a `slist`, ovvero il puntatore alla testa aggiornata della lista dopo l'eliminazione;
- **Parametri:** i parametri definiti nella funzione sono (`slist *p`, `slist *q`), rispettivamente il **puntatore corrente** alla testa della lista e il **puntatore al nodo di riferimento** (davanti al quale rimuovere);
- **Funzionamento del codice:** per poter eliminare un nodo all'interno della lista, supponiamo di possedere una struttura dati come quella definita al capitolo 3.1.
 - *riga 2:* dichiarazione di un puntatore ausiliario `r`, che servirà per scorrere la lista.
 - *riga 3:* se `p == q`, significa che il nodo da eliminare è la testa della lista. In questo caso basta avanzare la testa al nodo successivo: `p = p->next`;
 - *riga 4-8:* se il nodo da eliminare non è quello in testa alla lista (`p!=q`), utilizzando il puntatore ausiliario, utilizzo un ciclo per scorrere tutti i nodi, **ricordando che il nodo `q` specificato nei parametri non è null**.
 Proprio **per questo motivo**, a differenza di quanto detto nel capitolo 3.1.2, ho **un solo caso di terminazione**, e cioè quando `r->next=q`, ovvero il nodo precedente a quello da eliminare.
 Trovato il predecessore di `q`, quest'ultimo si salta collegando `r->next` direttamente al nodo successivo a `q`, ovvero `r->next=r->next->next`;
 - *riga 9:* si libera la memoria allocata dal nodo eliminato `q`;
 - *riga 10:* restituzione del puntatore alla testa aggiornato.

Esempio di utilizzo

```
slist *head = NULL;

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

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

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

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

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

return 0;
```


3.1.4 Creazione lista

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

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

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

Esempio di utilizzo

```
// Creazione di una lista vuota  
slist *head = createlist();  
  
// Inserimento di elementi  
head = insert2(head, head, 10);  
head = insert2(head, head, 20);  
head = insert2(head, head, 30);  
  
printList(head); // Output: 10 -> 20 -> 30 -> NULL  
  
return 0;
```

3.1.5 Cancellazione intera lista

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

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

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

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

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

Esempio di utilizzo

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

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

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

destroylist(head); // Distruzione della lista

return 0;
```

3.1.6 Attraversamento con funzione

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

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

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

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

Esempio di utilizzo

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

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

return 0;
```

3.1.7 Ricerca elemento con funzione

4 Alberi

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

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

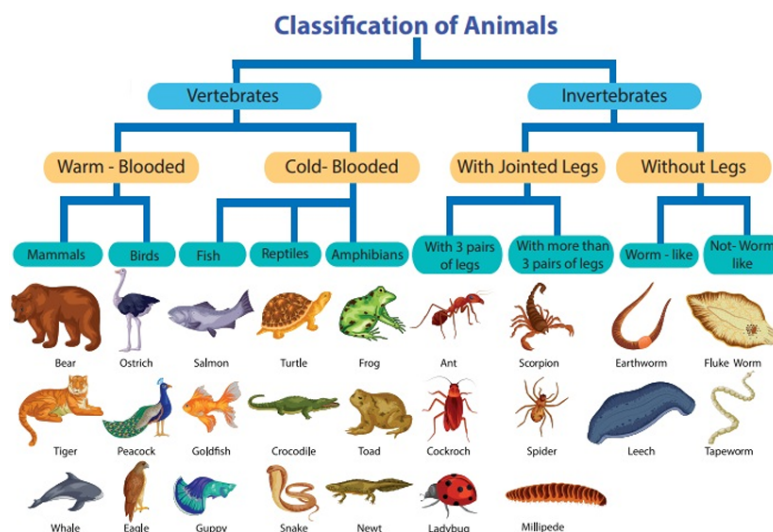
L'implementazione avviene, come per le liste, attraverso l'uso di **strutture autoreferenziali** e **puntatori**.

Struttura di un albero ordinato (o radicato)

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

Ogni albero presenta le seguenti proprietà:

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



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

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

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

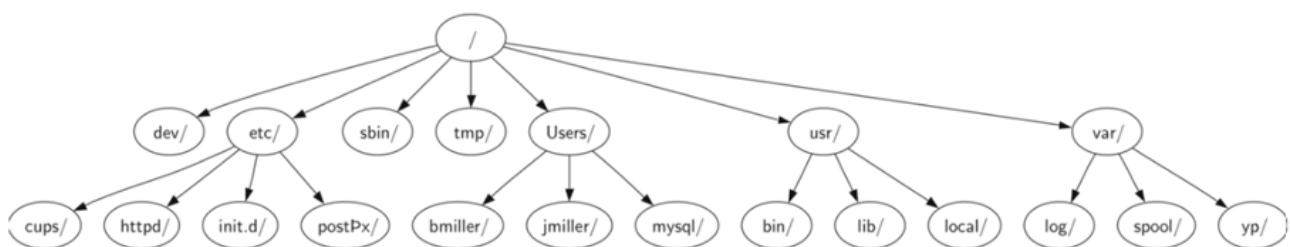


Figure 23: Albero del file system linux

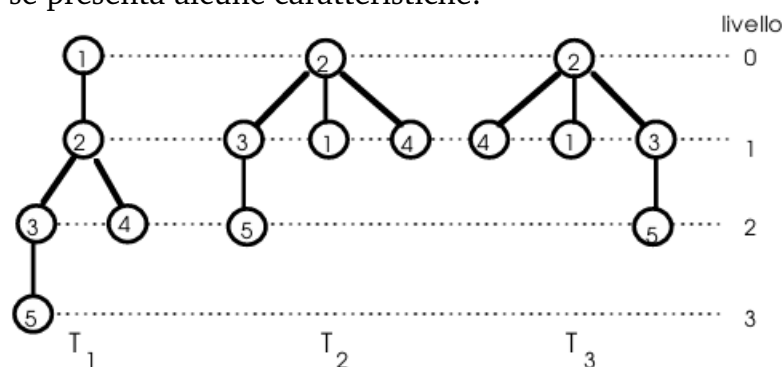
4.1 Terminologia

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

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

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

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



- **Profondità dei nodi (depth)**: profondità del cammino semplice dalla radice al nodo (misurata in numero di archi percorsi). Ad esempio, la radice avrà profondità 0, i figli avranno profondità 1, i figli dei figli avranno profondità 2 e così via...

la profondità massima delle sue foglie. Ad esempio, T_1 ha altezza 3, mentre T_2 e T_3 altezza 2;

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

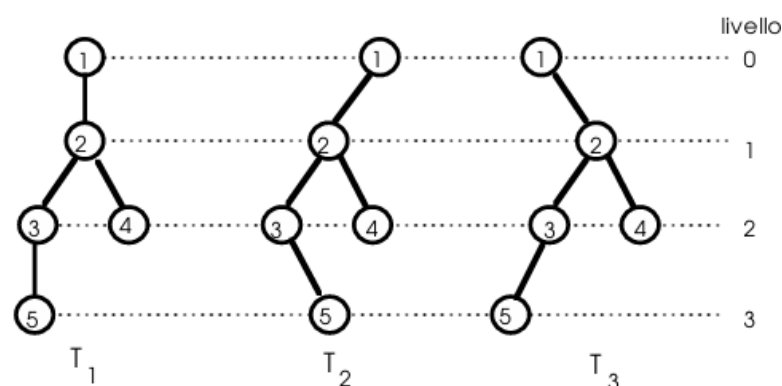
- **Altezza albero (height)**: indica

4.2 Alberi binari

Cos'è un'albero binario?

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

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



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

Nell'esempio mostrato, l'albero T_1 presenta al massimo due figli per ogni nodo, ma la rappresentazione grafica non aiuta a comprendere se si tratta di un figlio destro o sinistro. Al contrario T_2 e T_3 sono alberi binari e distinti, poiché non hanno gli stessi figli destri e sinistri.

4.2.1 Creazione di un albero binario

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

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

Inizializzazione dell'albero

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

Per quanto riguarda la struttura dell'albero:

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

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

Alberi binari e definizione ricorsiva

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

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

4.2.2 Costruzione di un albero binario

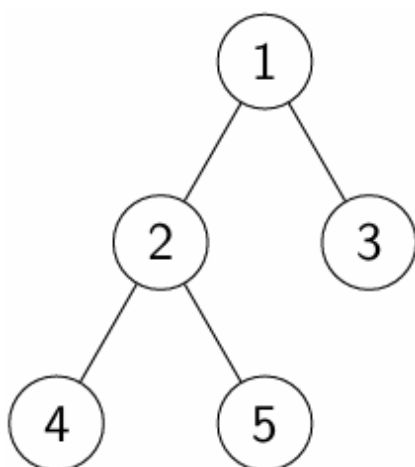
Dopo aver proceduto con la creazione della struttura e l'inizializzazione dell'albero, si procede con la sua creazione, allocando i vari nodi e definendo i puntatori per il figlio destro e sinistro.

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

```

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

```



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

4.3 Le visite

Cos'è una visita?

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

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

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

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

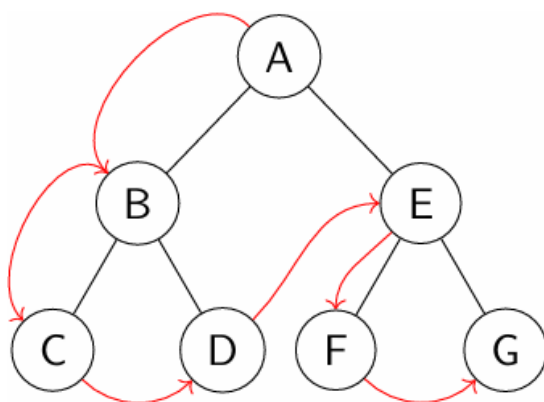
- **Visita in ampiezza** (Breadth First Search - BFS): partendo dalla radice, vengono visitati i nodi livello per livello, “orizzontalmente”, fino a raggiungere il livello massimo dell’albero. Proprio per questo, un’analisi di questo tipo viene anche detta **ordine per livelli**.

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

4.3.1 Visita in preorder (DFS)

Funzionamento della visita in preorder

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



(a) albero attraversato in preorder

```

dfs(Tree t)
1: if t ≠ nil then
2:   % pre-order visit of t
3:   print t
4:
5:   dfs(t.left())
6:
7:   dfs(t.right())
8: end if

```

(b) Pseudocodice visita in preorder

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

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

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

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

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

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

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

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

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

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

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

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

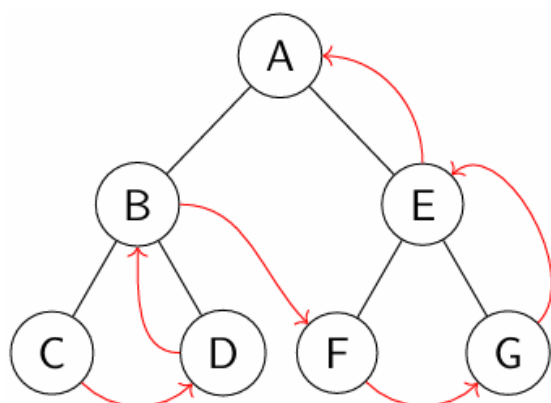
9. Ciò che succede è che in A la chiamata `dfs(t.left())` è stata completata (tutto il sottoalbero sinistro), quindi procedo con la chiamata ricorsiva per il nodo destro della radice E : in questo modo **il procedimento si ripete in modo analogo** per tutto il sottoalbero destro.

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

4.3.2 Visita in postorder (DFS)

Funzionamento della visita in postorder

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



(a) albero attraversato in postorder

`dfs(Tree t)`

```

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

(b) Pseudocodice visita in postorder

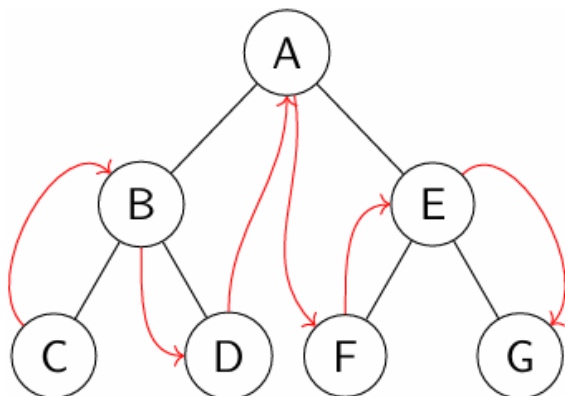
Per quanto riguarda la visita in postorder, anche in questo caso, dividiamo il procedimento in alcuni passi, monitorando l'evoluzione della sequenza di stampa e dello stack di chiamate:

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

4.3.3 Visita inorder (DFS)

Funzionamento della visita inorder

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



(a) albero attraversato inorder

dfs(Tree t)

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

(b) Pseudocodice visita inorder

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

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

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

4.3.4 Attraversamento di un albero binario in C

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

```

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

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

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

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

```

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

```
}
```

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

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

Esempio di utilizzo

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

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

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

    return 0;
}
```

4.3.5 Cancellazione di un albero

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

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

```

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

```

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

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

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

Esempio di utilizzo

```

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

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

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

    return 0;
}

```

4.4 Alberi binari di ricerca (BST)

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

I dizionari consentono **tre operazioni fondamentali**:

- $\text{lookup}(\text{Item } k) \rightarrow$ cercare un elemento con chiave k ;
- $\text{insert}(\text{Item } k, \text{Item } v) \rightarrow$ inserire un elemento con chiave k e valore v ;
- $\text{remove}(\text{Item } k) \rightarrow$ rimuovere l'elemento con chiave k .

Un dizionario può essere implementato in vari modi:

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

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

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

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

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

L'idea degli alberi binari di ricerca

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

Cos'è un albero binario di ricerca?

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

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

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

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

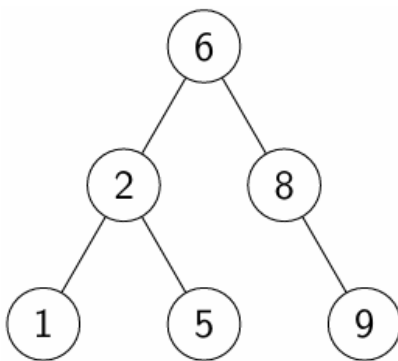
In questo modo tutte le operazioni di ricerca, inserimento e cancellazione sfruttano questa divisione **per ridurre progressivamente lo spazio di ricerca** (agevolare la ricerca).

Alcune precisazioni

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

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

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



Le operazioni previste dagli alberi binari di ricerca sono:

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

4.4.1 Implementazione della ricerca

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

Ricerca ricorsiva

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

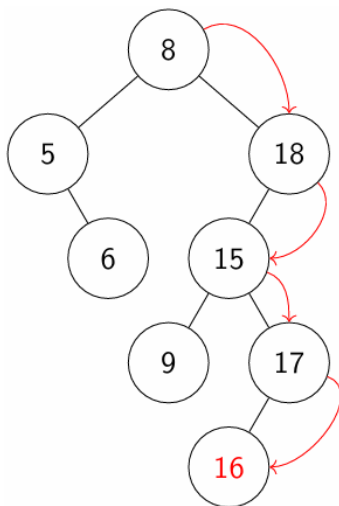
- **Tipo di ritorno:** la funzione restituisce un puntatore a un nodo della struttura bstree, la cui composizione è identica a quella di un albero binario vista al capitolo 4.2.1.

Il nodo restituito è:

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

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

Figure 35: Albero BST



- **riga 4:** la prima istruzione della funzione è un **controllo di terminazione** per fare in modo che, al ripetere della funzione ricorsiva, si capisca se la chiave del nodo è quella cercata o meno; Se `p==NULL`, significa che abbiamo raggiunto un ramo vuoto, la chiave non esiste e la funzione restituisce `NULL`, mentre se `p->data==x` abbiamo trovato il nodo desiderato e si ritorna il puntatore al nodo stesso.
- **riga 7:** questa seconda istruzione implementa la ricerca ricorsiva nel sotto albero destro. Se la chiave cercata `x` è **maggiore della chiave del nodo corrente** (`p->data`), allora, secondo la proprietà del BST (capitolo 4.4), il **valore** può trovarsi solo nel **sottoalbero destro**.
Dunque, la funzione richiama se stessa (in modo ricorsivo), passando come parametri il figlio destro del nodo sul quale si stava lavorando (`p->right`), e ancora una volta la chiave cercata (`x`). A questo punto la funzione ricomincia, e se il controllo di terminazione non si attiva, viene ricontrollato se `x` è maggiore o minore di `p->data`.
- **riga 10:** anche in questo caso viene implementata la ricerca ricorsiva ma nel sottoalbero sinistro. È **importante notare** che non viene specificato `x < p->left` proprio perché come descritto al capitolo capitolo 4.4 ("Alcune precisazioni"), **non possono essere presenti nodi con lo stesso valore di chiave**, dunque è scontato che sia l'unica altra opzione possibile.
Quindi, se la chiave cercata `x` è **minore della chiave del nodo corrente** (`p->data`), allora, secondo la proprietà del BST, il **valore** può trovarsi solo nel **sottoalbero sinistro**.
Anche in questo caso, viene richiamata ricorsivamente la funzione, con `p->left` come nuovo nodo di partenza, fino a che non si attiva il controllo di terminazione, o `x` è maggiore di `p->data`.

Ricerca iterativa

```

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

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

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

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

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

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

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

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

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

    return 0;
}
```

4.4.2 Creazione di un nuovo nodo

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

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

4.4.3 Inserimento di un nodo

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

Inserimento ricorsivo

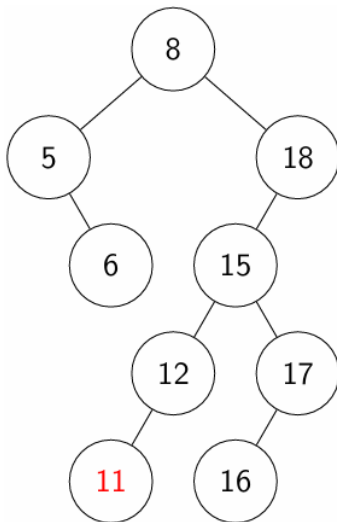
```

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

```

- **Tipo di ritorno:** la funzione restituisce un puntatore a `bstree`.
Il valore restituito rappresenta la radice del sottoalbero risultante dopo l'inserimento;
 - **Parametri:** la funzione accetta due parametri;
 - `bstree *p`: è il puntatore al nodo corrente dell'albero binario di ricerca.
In genere la prima chiamata della funzione riceve la radice dell'albero (`root`), ma nelle chiamate ricorsive il parametro `p` sarà aggiornato ai figli sinistro o destro.
 - `int x`: è la chiave da inserire nel campo `data` del nuovo nodo.
 - **Funzionamento del codice:** la struttura del codice è simile a quella che viene utilizzata per effettuare la ricerca di un nodo (capitolo 4.4.1). Come avviene per la ricerca di un nodo, anche l'**inserimento sfrutta la logica dicotomica** utilizzata per la creazione dell'albero binario in modo tale che **in base al valore della chiave del nodo che si vuole inserire**, venga "esclusa" una metà dell'albero ad ogni ricorsione o iterazione (in questo caso ad ogni ricorsione).
- Nell'esempio riportato in Figure 36 viene effettuato l'inserimento del valore 11.

Figure 36: Albero BST



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

Inserimento iterativo

```

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

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

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

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

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

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

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

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

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

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

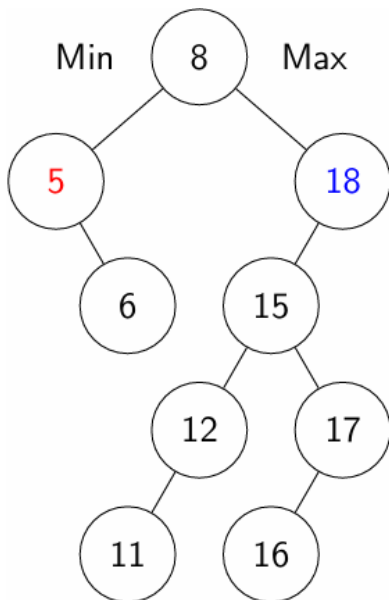
    // Inserimento ricorsivo dei valori
    root = insert(root, 8);
    root = insert(root, 5);
    // ...
    // ...
    root = insert(root, 11);

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

4.4.4 Ricerca del massimo e del minimo

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

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



Esempio pratico

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

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

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

Ricerca ricorsiva del massimo e del minimo

```

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

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

→ nel caso della ricerca del massimo si verifica se il **figlio destro** del nodo corrente non è nullo;

→ nel caso della ricerca del minimo si verifica se il **figlio sinistro** del nodo corrente non è nullo.

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

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

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

Ricerca iterativa del massimo e del minimo

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

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

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

Esempio di utilizzo (valido per entrambe le modalità)

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



```

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

```

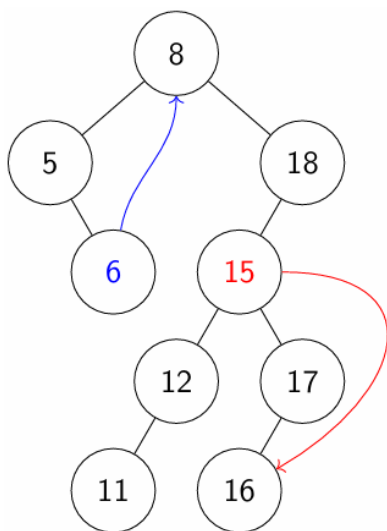
4.4.5 Ricerca del successore-predecessore

Definizione di successore

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

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

Figure 38: Albero BST



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

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

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

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

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

Ricerca iterativa del successore

```

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

```



```

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

```

```

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

```

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

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

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

→ *riga 4*: il valore della chiave del nodo `px` viene salvato nella variabile `x`;

→ *riga 5*: viene dichiarato un puntatore `y` e inizializzato a `NULL`: questo punterà di volta

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

Esempio di utilizzo

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

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

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

Passaggio da successore a predecessore

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

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

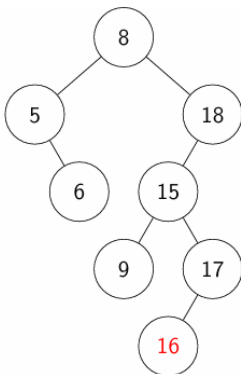
4.4.6 Cancellazione di un nodo

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

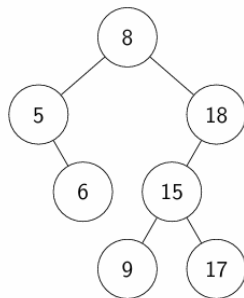
Durante la cancellazione di un determinato nodo possono sorgere **tre differenti casi**:

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

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



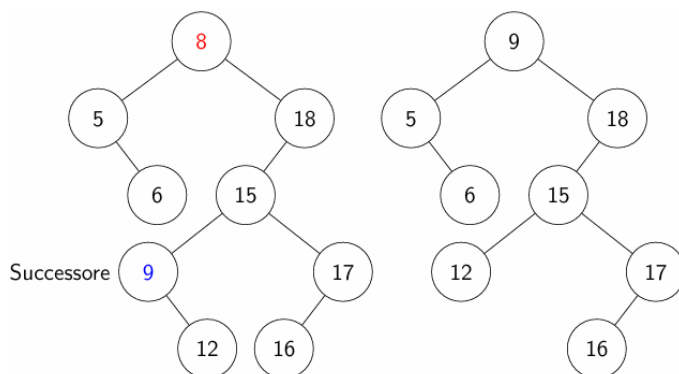
(a) Caso 1 - nessun figlio



(b) Caso 2 - un solo figlio

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

Figure 39: Cancellazione di un nodo



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

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

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

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

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

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

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

Cancellazione ricorsiva di un nodo

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

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

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

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

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

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

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

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

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

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

Esempio di utilizzo

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

```
int main(int argc, char *argv[]) {
    bstree *root = NULL;
    /* Creazione dell'albero BST in logica dicotomica illustrato in
    Figure 39 utilizzando la funzione di inserimento, come fatto al
    capitolo (4.4.3) */
    // ...
    // ...

    root = deleteNode(root, 8); // Cancellazione del nodo 8
    printf("La nuova radice e' %d\n", root->data);
}
```

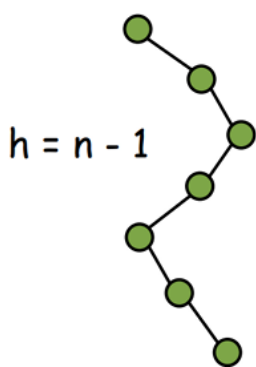
4.4.7 Alberi BST: alcune osservazioni

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

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

Tempo di ricerca

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



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

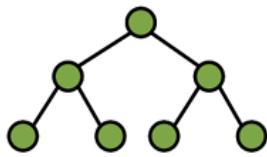
Gli alberi binari di ricerca, sono sì una buona idea per portare la ricerca binaria nel campo degli alberi, però **se i dati sono inseriti in certi modi sbagliati** (caso pessimo), si ottiene un'altezza dell'albero di $O(n)$ e quindi **non si ha nessun vantaggio** rispetto ad una banale **ricerca lineare**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

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

Cos'è un BST bilanciato?

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

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



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

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

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

Fattore di bilanciamento

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

Cos'è un albero AVL?

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

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

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

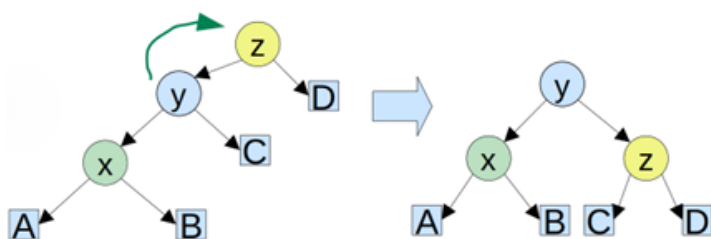
Cos'è una rotazione?

Una rotazione è un'operazione locale che viene **eseguita** sul **nodo sbilanciato** causando lo spostamento di tre nodi: il nodo sbilanciato stesso, suo figlio e il suo nipote. Lo sbilanciamento viene riequilibrato **senza violare le proprietà strutturali** dei BST. Dunque, le **rotazioni** permettono di **abbassare** il fattore di bilanciamento.

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

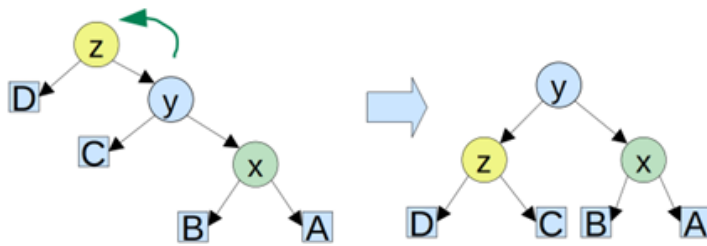
Invece, le **forme di sbilanciamento** di un albero BST sono quattro, e come detto prima, variano in base al punto di inserimento del nuovo nodo:

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



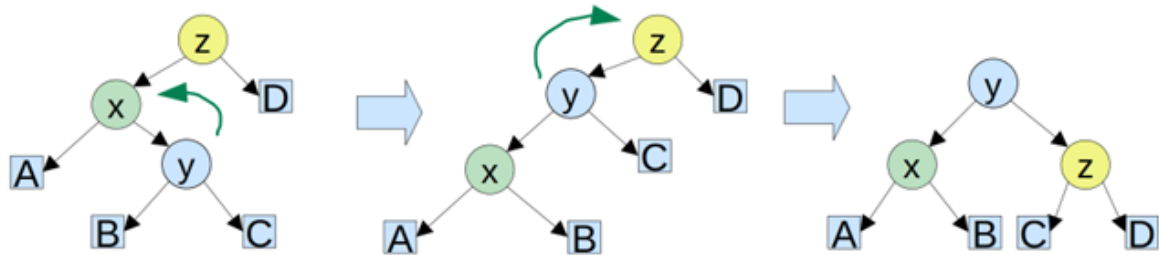
In questo caso il ramo sinistro "pesa" troppo e presenta un **fattore di bilanciamento** > 1 . Il caso **left-left** possiede una **forma lineare**, dunque, per ribilanciare l'albero basta effettuare una sola rotazione verso destra.

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



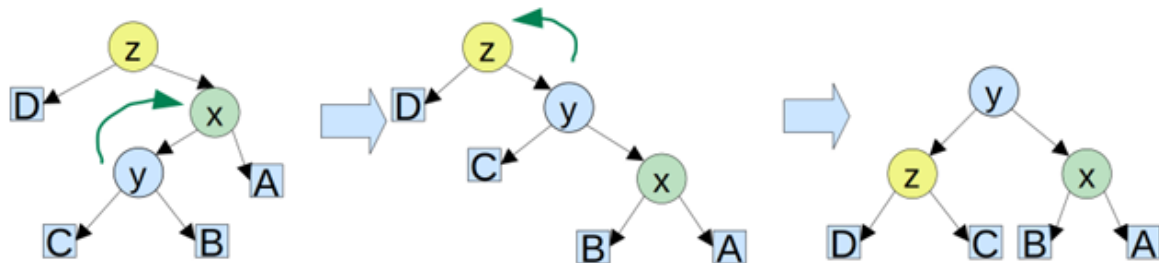
Anche in questo caso il ramo destro "pesa" troppo e presenta un **fattore di bilanciamento** < -1 . Il caso *right-right* possiede una **forma lineare**, e per ribilanciare l'albero, basta effettuare una sola rotazione verso sinistra.

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



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

1. **Rotazione a sinistra**: viene effettuata sul figlio destro del figlio sinistro del nodo sbilanciato, per fare in modo di riportarsi alla **forma left-left**;
 2. **Rotazione a destra**: viene effettuata sul nodo sbilanciato per completare il bilanciamento.
- *Caso right-left (RL)*: il caso *right-left* si verifica quando il nuovo nodo viene inserito nel sottoalbero sinistro del figlio destro del primo nodo, il quale diventa sbilanciato.



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

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

Rotazione a sinistra/destra per il bilanciamento

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

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

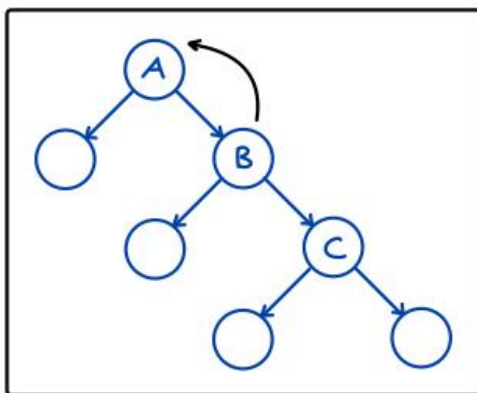

- **Tipo di ritorno:** entrambe le funzioni ritornano un puntatore alla struttura di un nodo. Infatti, dopo la rotazione la radice dell'albero cambia: ciò che viene restituito è proprio la nuova radice dell'albero;
- **Parametri:** entrambe le funzioni accettano un unico parametro, ovvero il puntatore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

Funzionamento di rotateleft (e rotateright)

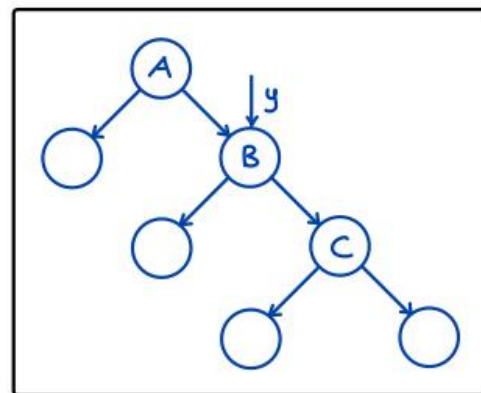
Per illustrare il funzionamento di rotateleft utilizziamo la seguente immagine.

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

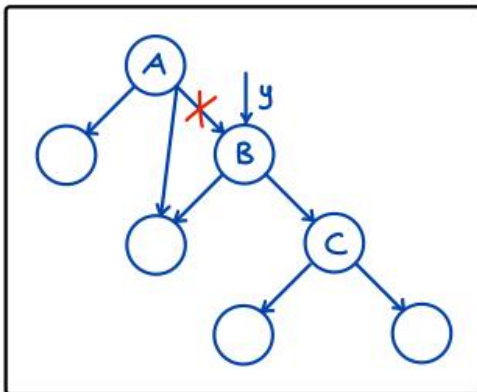
Caso Iniziale



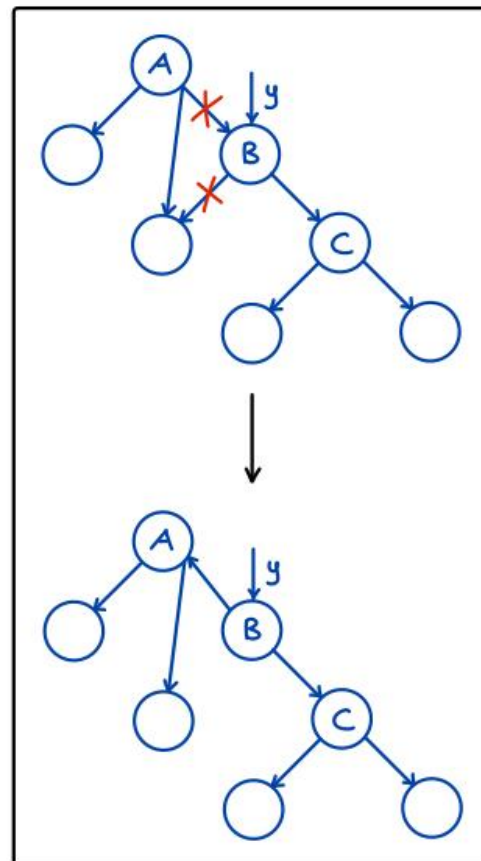
Riga 1 e 2



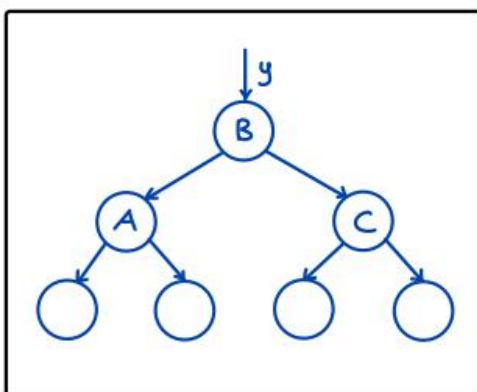
Riga 3



Riga 4



Stato Finale



5 Analisi della complessità degli algoritmi

Obiettivo

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

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

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

5.1 Complessità e dimensione dell'input

Cos'è la complessità?

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

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

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

Le due principali tipologie di risorse impiegate sono:

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

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

Dimensione dell'input

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

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

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

- **Criterio di costo logaritmico**: la taglia dell'input è il **numero di bit necessari per rappresentarlo**. In questo caso, il costo dell'elaborazione dipende direttamente dalla **lunghezza in bit** dei dati in ingresso, e non dal numero di elementi di cui è composto.

Esempio: avendo in ingresso un numero intero molto grande, si può considerare il numero di bit necessari per rappresentarli.

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

5.2 Definizione di tempo e modello di calcolo

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

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

Tempo \equiv numero istruzioni elementari

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

Cos'è il tempo di calcolo?

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

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

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

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

5.2.1 Random Access Machine (RAM)

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

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

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

5.3 Valutazione del caso pessimo, medio e ottimo

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

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

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

Perché valutiamo il caso pessimo se può verificarsi molto raramente?

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

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

5.3.1 Tempo di calcolo della funzione $\text{min}()$ (iterativa)

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

A questo punto possiamo:

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

item $\text{min}(\text{item}[] A, \text{int } n)$		
	Costo	# Volte
item $\text{min} = A[0]$	c_1	1
for $i = 1$ to $n - 1$ do	c_2	n
if $A[i] < \text{min}$ then	c_3	$n - 1$
$\text{min} = A[i]$	c_4	$n - 1$
return min	c_5	1

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

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

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

Osservazioni sull'identificazione del caso

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

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

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

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

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

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

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

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

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

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

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

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

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

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

La ricerca di un elemento non presente all'interno del vettore, causa l'esecuzione ricorsiva delle istruzioni con costo: $c_3, c_4, c_6, c_7 + T(n/2)$. Inoltre, ad ogni ricorsione viene esplorata la metà destra, scartatando metà degli elementi rimanenti, aggiornando gli indici di inizio (i), fine (j) e dell'elemento mediano del vettore (m), fino ad esaurire tutti gli elementi. Il costo della funzione è quindi: $T(n) = c_1 + c_2 + c_4 + c_6 + c_7 + T(\frac{n}{2}) = d + T(\frac{n}{2})$ dove d rappresenta la somma da $c_1 \dots c_7$.

Questo significa che l'equazione non è definita in modo semplice da un solo valore, ma da una **relazione di ricorrenza** (capitolo 5.5). Una tecnica che viene utilizzata per risolvere le relazioni di ricorrenza consiste nel **produrre una catena di uguaglianze** ottenute per sostituzioni successive.

Infatti possiamo scrivere che $T(n) = T(n/2) + d = T(n/4) + 2d = \dots = T(n/2^k) + kd$, da cui possiamo ricavare che $\frac{n}{2^k} \Rightarrow k = \log n$. Andando avanti con la ricorrenza si arriverà ad un punto in cui $T(n/2^k) + kd = T(1) + kd = [T(0) + d] + kd = T(0) + d(k+1) = c + kd + d = d \log n + (c + d)$.

5.4 Ordini di complessità

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

Concetto di crescita asintotica

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

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

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

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

Esempio intuitivo: Supponiamo di avere due algoritmi.

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

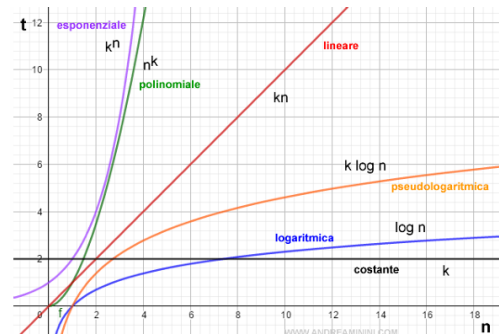
- Per $n = 10$, A fa 1000 operazioni, mentre B ne fa 100;
 → Per $n = 100$, A fa 100.000 operazioni, mentre B ne fa 1.000.000;
 Quindi asintoticamente A è il migliore, anche se su input piccolo B sembrava il migliore.

5.4.1 Principali classi di efficienza asintotiche

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

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

(a)



(b)

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

5.4.2 Notazioni asintotiche

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

Notazione Big-O (O grande)

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

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

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

Notazione Omega (Ω grande)

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

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

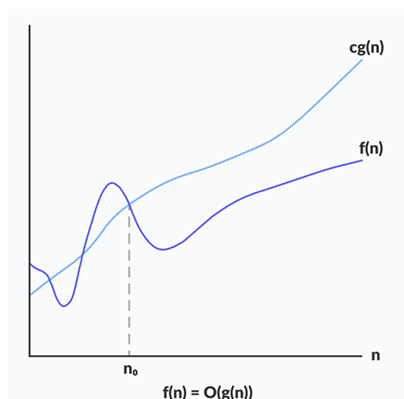
Esempio: se un algoritmo è $\Omega(n)$, significa che anche nel caso più favorevole deve comunque guardare almeno n elementi.

Notazione Theta (Θ grande)

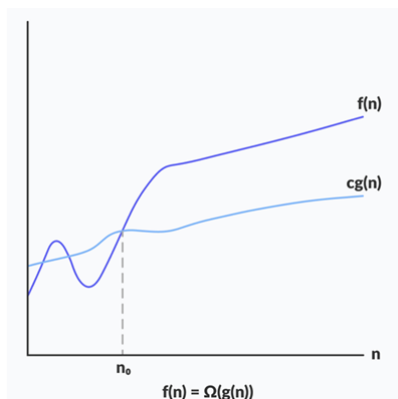
Descrive i due **limiti (inferiore e superiore)** della crescita di un algoritmo.

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

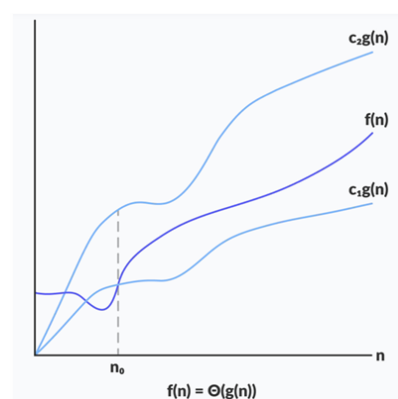
Esempio: se un algoritmo è $\Theta(n \log n)$, significa che cresce proprio in quel modo: non più veloce, non più lento.



(a) Notazione O grande



(b) Notazione Ω grande



(c) Notazione Θ grande

5.5 Le ricorrenze

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

Cos'è una ricorrenza?

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

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

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

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

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

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

5.5.1 Metodo di sostituzione (o per tentativi)

Idea di base

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

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

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

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

Osservazioni

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

5.5.2 Metodo dell'esperto

Idea di base

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

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

- Ogni a -esimo sottoproblema viene risolto nel tempo $T(n/b)$;
- Il costo per dividere il problema e combinare i risultati dei sottoproblemi è descritto da $f(n)$

Dunque, questa metodologia di soluzione per le ricorrenze **risulta perfetta solo per algoritmi classici** come mergesort, binary search, quicksort, ecc...

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

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

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

Esempio del Caso 1

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

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

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

Esempio del Caso 2

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

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

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

Esempio del Caso 3

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

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

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

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

Idea di base

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

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

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

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

Per comprendere meglio il concetto si consideri la ricorrenza:

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

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

- **Livello 0 (radice)**
 - Si ha un solo problema di dimensione n .
- **Livello 1**
 - Il problema si divide in 4 sottoproblemi (coefficiente 4);
 - Il costo di ciascuno è $n^2/4$;
 - Ognuno ha dimensione $n/2$;
 - Costo totale del livello $4 \cdot (n^2/4) = n^2$
- **Livello 2**
 - Ogni sottoproblema del livello 1 si divide di nuovo in 4 \rightarrow in totale $4^2 = 16$ sottoproblemi;
 - Il costo di ciascuno è $n^2/16$;
 - Ognuno ha dimensione $n/4$;
 - Costo totale del livello $16 \cdot (n^2/16) = n^2$

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

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

Osservazioni

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

5.6 Ordinamento

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

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

Obiettivo

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

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

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

Problema dell'ordinamento (sorting)

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

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

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

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

5.6.1 Selection Sort

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

Algoritmo selection sort

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

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

Algorithm selectionSort(Item[] A, int n)

```

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

```

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

```

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

```

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

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

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

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

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

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

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

(a)

(b)

Complessità del selectionSort()

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

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

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

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

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

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

5.6.2 Insertion sort

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

Algoritmo insertion sort

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

Algorithm insertionSort(Item[] A, int n)

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

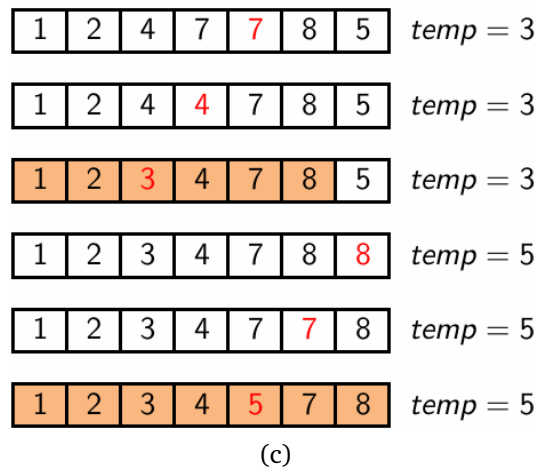
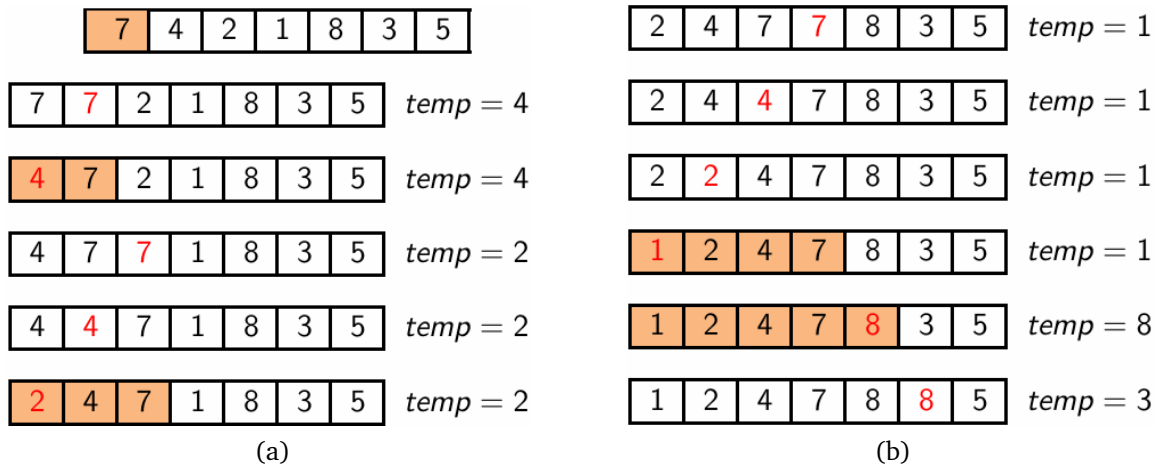
Immaginiamo di avere un input del tipo $A = \{2, 3, 1, 4, 7, 3\}$.

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

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

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

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



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

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

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

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

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

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

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

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

In questo caso, non avverrà mai che $A[j-1] [1]$ sia maggiore di $\text{temp} [2]$ e così via..., non permettendo al ciclo `while` di verificarsi. Rimane solo il ciclo esterno, che esegue operazioni di costo **costante**.

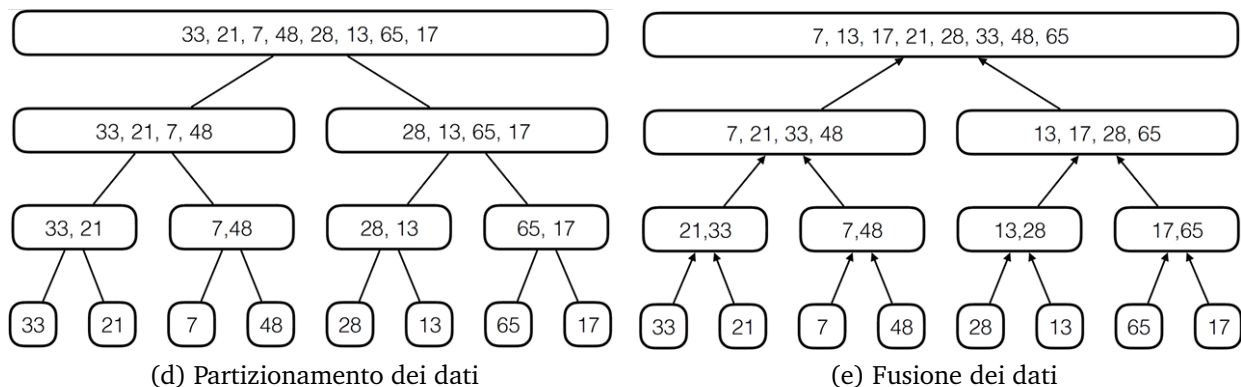
Complessità? In questo caso, la complessità per delle istruzioni costanti vale $O(n)$.

5.6.3 Merge sort

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

Algoritmo merge sort

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



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

```

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

```

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

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

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

Prima ricorsione (riga 2)

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

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

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

- `mergeSort(A, first, mid)` non prosegue poiché il sottovettore ha un solo elemento $[4]$;
 - anche, `mergeSort(A, mid+1, last)` non prosegue perché anche $[3]$ è un singolo elemento.
- Raggiunto il caso base per entrambi i sottovettori, è possibile richiamare `merge()` per combinarli e ottenere il vettore ordinato $[3, 4]$.

Seconda ricorsione (riga 3)

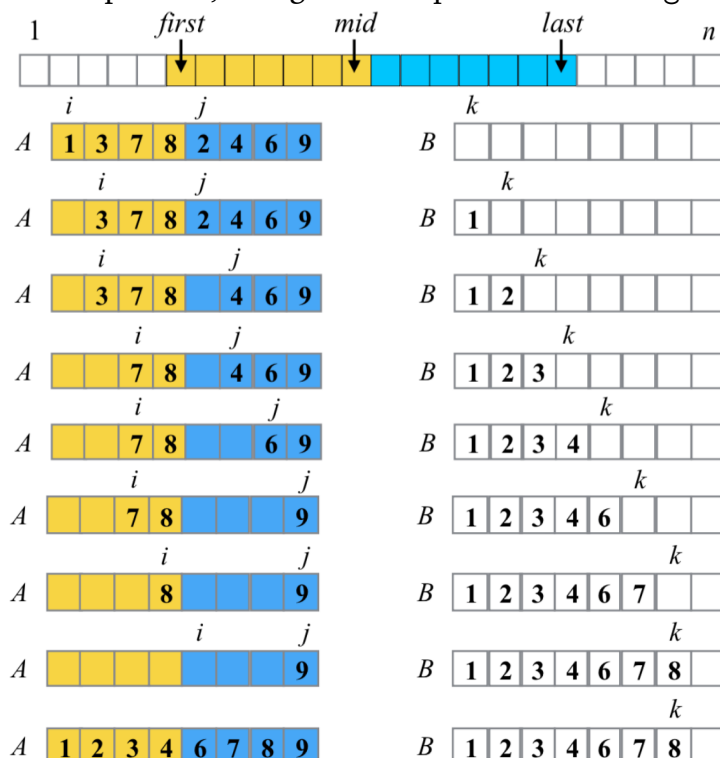
Terminata la ricorsione sulla parte sinistra del vettore iniziale, il controllo ritorna alla chiamata precedente, che ora può eseguire l'istruzione ricorsiva successiva (riga 4) sulla parte destra, utilizzando `mergeSort(A, mid+1, last)`.

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

Merge finale (riga 4)

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

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



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

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

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

Vengono quindi utilizzati tre indici *i*, *j* e *k* per scandire, rispettivamente, A[*first*..*mid*], A[*mid*+1..*last*] e B[*start*..*end*]. Ad ogni passo, sono

confrontati gli elementi A[*i*] e A[*j*], il minore viene copiato in B[*k*], e vengono incrementati di una posizione *k* e l'indice dell'elemento che risulta minore.

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

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

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

Figure 67: Pseudocodice della funzione merge()

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

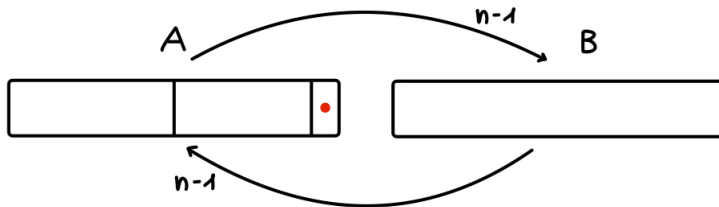
```

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

```

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

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



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

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

Osservazione sulla complessità di merge()

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

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

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

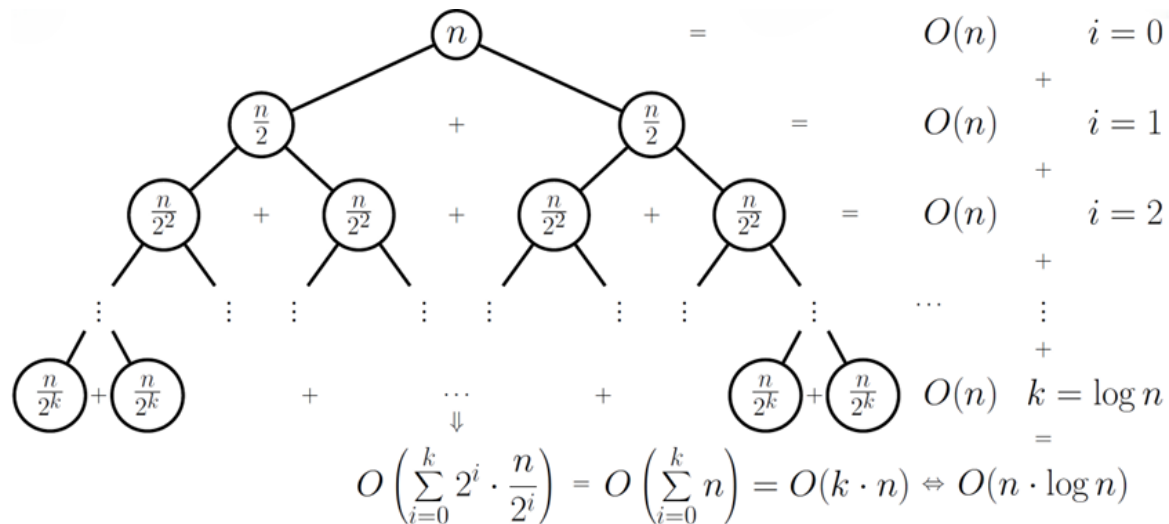
Quindi per ordinare l'array completo:

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

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

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

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



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

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

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