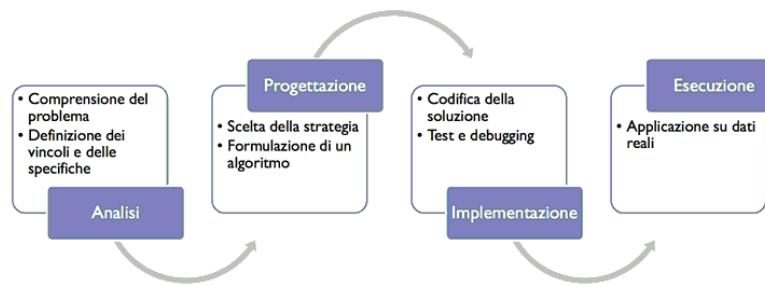


PER ALTRI APPUNTI CONSULTARE IL SITO:
https://luigi-v.github.io/Appunti_Universita/

L01.1. SVILUPPO DI PROGRAMMI

Un **algoritmo** è composto da una sequenza *finita* di istruzioni *elementari*, che ha come obiettivo quello di risolvere un problema, partendo dai dati di input ed ottenendo i dati di output, risolvendo il problema dato.

FASI DELLO SVILUPPO:



■ ANALISI:

Specificare **cosa** fa il programma ed individua i dati di **input** e di **output** coi relativi **vincoli**.

I vincoli definiti sono: **precondizione**, condizione definita sui dati di input che deve essere soddisfatta affinché la funzione sia applicabile, e **postcondizione**, condizione definita su dati di output e dati di input e che deve essere soddisfatta al termine dell'esecuzione del programma (definendo cosa sono i dati di output in funzione di quelli di input).

È buona norma utilizzare un **dizionario dei dati** da arricchire durante le varie fasi del ciclo di vita, ovvero una tabella il cui schema è "identificatore", "tipo" e "descrizione" (serve a specificare meglio l'identificatore e a descrivere il contesto in cui il dato viene usato).

Esempio di ordinamento di una sequenza di interi:

- Dati di ingresso: sequenza s di n interi
- Precondizione: $n > 0$
- Dati di uscita: sequenza s1 di n interi
- Postcondizione: s1 è una permutazione di s dove
$$\forall i \in [0, n-2], s_1[i] \leq s_1[i+1]$$

Identificatore	Tipo	Descrizione
Dizionario dei dati	s	sequenza
	s1	sequenza
	n	intero
	i	intero
		numero di elementi nella sequenza
		indice per individuare gli elementi nella sequenza

■ PROGETTAZIONE:

Definisce **come** il programma effettua la trasformazione specificata, progettazione dell'algoritmo per raffinamenti successivi (*stepwise refinement*), definendo i vari step e utilizzando la decomposizione funzionale.

Esempio di ordinamento di una sequenza di interi:

- Input sequenza s in un array a di dimensione n
- Ordina array a di dimensione n
- Output sequenza s1 contenuta in array a di dimensione n

Raffinamento del programma principale, ovvero definiamo delle funzioni corrispondenti agli step individuati:

- input_array(a, n)
- ordina_array(a, n)
- output_array(a, n)

Per ognuno di questi si effettua *specificazione*, *progettazione*, *codifica* e *verifica*.

■ IMPLEMENTAZIONE:

Codifica dell'algoritmo nel linguaggio scelto, e lo si implementa su un elaboratore.

■ ESECUZIONE:

Verifica (testing) del programma (individuazione dei malfunzionamenti), effettuando delle scelte di casi di prova, si esegue il programma e si verificano i risultati ottenuti rispetto ai risultati attesi.

L01.1.1 ORGANIZZAZIONE DEL CODICE

La **progettazione** è l'insieme delle attività relative al concepimento della soluzione informatica di un problema, si sviluppa a partire da una specifica (prodotta in fase di analisi), come se fosse una vera e propria architettura (software) fatta da entità autonome che interagiscono tra loro.

Ci sono 4 **principi inspiratori** di un progetto software e sono:

■ ASTRAZIONE:

Procedimento mentale che consente di evidenziare le caratteristiche pregnanti di un problema e offuscare o ignorare gli aspetti che si ritengono secondari rispetto ad un determinato obiettivo. Esistono due tipi:

- Astrazione funzionale e procedurale**, una funzionalità di un programma è delegata ad un sottoprogramma (funzione o procedura), è definita ed usabile indipendentemente dall'algoritmo che la implementa (es. algoritmi di ordinamento di un array);
- Astrazione sui dati**, un dato o un tipo di dato è totalmente definito insieme alle operazioni che sul dato possono essere fatte, pertanto, sia le operazioni che il dato (o il tipo di dato) sono usabili a prescindere dalle modalità di implementazione.

■ INFORMATION HIDING:

Nascondere il funzionamento interno, deciso in fase di progetto, di una parte di un programma. I **vantaggi** sono:

- Parti del programma che non devono essere modificate sono inaccessibili;
- Correzione degli errori facilitata: un errore presente in un modulo può essere corretto modificando soltanto quel modulo.

■ RIUSO DEL CODICE:

Pratica, estremamente comune nella programmazione, di richiamare o invocare parti di codice precedentemente già scritte ogni qualvolta risulta necessario, senza doverle riscrivere daccapo. Una soluzione è usare funzioni e librerie di funzioni.

■ MODULARITÀ:

È una tecnica di suddividere un progetto software (aiuta a gestire la complessità), implementando dei moduli (funzioni, classi, package).

Un **modulo** è una unità di programma che mette a disposizione risorse e servizi computazionali (dati, funzioni, ...) e devono avere due proprietà:

1. **Elevata coesione**: le varie funzionalità messe a disposizione da un singolo modulo sono strettamente correlate;
2. **Indipendenza**: sviluppabili separatamente dal resto del programma, con compilazione e testing separati.

Un modulo è costituito da due parti:

1. **Interfaccia**, definisce le risorse ed i servizi messi a disposizione dei "clienti" (programma o altri moduli), completamente visibile ai clienti;
2. **Sezione implementativa (body)**, implementa le risorse ed i servizi esportati ed è completamente occultato.

Altre particolarità dei moduli sono che quest'ultimo può usare altri moduli e può essere compilato indipendentemente dal modulo che lo usa.

In C non esiste un apposito costrutto per realizzare un modulo, per esportare le risorse definite in un file (modulo), il C fornisce un particolare tipo di file, chiamato **header file** (estensione .h) che rappresenta l'interfaccia di un modulo verso gli altri moduli.

Per accedere alle risorse messe a disposizione da un modulo bisogna includere il suo header file, esempio `#include "modulo.h"`.

Il modulo mette a disposizione attraverso la sua interfaccia funzioni e procedure, si presenta come una "libreria" di funzioni e per poter garantire l'information hiding non ci devono essere variabili globali e funzioni nascoste.

Esempio modulo Utils:

```
// Interfaccia del modulo: file utils.h
/* Specifica della funzione scambia */
void scambia(int * x, int * y);
// dichiarazione altre funzioni ...

// Implementazione del Modulo: file utils.c
/* commenti relativi alla progettazione e realizzazione della funzione scambia */
void scambia(int * x, int * y){
    int temp = *x;
    *x = *y;
    *y = temp;
}
// definizione altre funzioni ...
```

Vista del cliente, può usare tali risorse e servizi esportati dal modulo.

Reale implementazione del modulo.

USO DEI COMMENTI:

I commenti relativi alla specifica di una funzione possono essere inseriti nell'header file prima del prototipo della funzione, serve da documentazione per chi dovrà usare la funzione (modulo client). I commenti relativi alla progettazione e realizzazione di una funzione possono essere inseriti nel file .c prima della definizione della funzione (o anche all'interno del corpo della funzione), serve da documentazione per chi dovrà modificare la funzione.

Esempio modulo Vettore:

vettore.h	vettore.c	main.c
<pre>void input_array(int a[], int n); void output_array(int a[], int n); void ordina_array(int a[], int n); int ricerca_array(int a[], int n, int elem); int minimo_array(int a[], int n); ... </pre>	<pre>#include <stdio.h> #include "vettore.h" #include "utils.h" int minimo_i(int a[], int i, int n); int minimo_i(int a[], int i, int n); // contiene funzione scambia // dichiarazione locale void input_array(int a[], int n) { ... } void output_array(int a[], int n) { ... } void ordina_array(int a[], int n) { ... } int ricerca_array(int a[], int n, int elem) { ... } int minimo_array(int a[], int n) { ... } int minimo_i(int a[], int i, int n) { ... } // usata da ordina_array</pre>	<pre># include <stdio.h> # include "vettore.h" # define MAXELEM 100 int main(){ ... }</pre>

COMPILAZIONE DEI FILE:

I due moduli possono essere compilati indipendentemente:

- `gcc -c utils.c`
- `gcc -c vettore.c`
- `gcc -c main.c`
- `gcc -c utils.c vettore.c main.c`

In entrambi i casi si ottengono tre file con estensione .o e per **collegare** (link) i tre moduli e produrre l'eseguibile:

- `gcc utils.o vettore.o main.o -o vettore.exe`

Possibile compilazione e collegamento in un sol passo:

- `gcc utils.c vettore.c main.c -o vettore.exe`

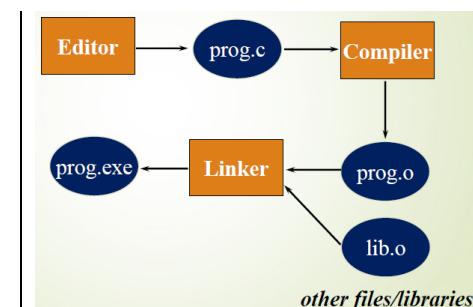
MAKEFILE E MAKE:

Il comando **make** esegue compilazione e collegamento dei vari moduli che compongono il progetto.

Il **makefile** invece è costituito da specifiche del tipo: `target_file: dipendenze_da_file`.

L'esecuzione avviene tramite comando: `make target_file`

L'ordine dei comandi non è importante ma è buona norma inserire come prima specifica quella per la costruzione del programma eseguibile.



```

prog : main.o list.o item.o
       gcc main.o list.o item.o -o prog.exe
       ./prog.exe

item.o :
       gcc -c item.c

list.o :
       gcc -c list.c

main.o :
       gcc -c main.c

clean:
       rm -f *.o *.exe

```

L01.2. OPERAZIONI SU ARRAY

Un Array è una struttura contenente un certo numero di valori, tutti dello stesso tipo. Questi valori vengono chiamati **elementi** e possono essere selezionati individualmente tramite la loro posizione nell'array.

Per dichiarare un vettore dobbiamo specificare il **tipo** ed il **numero di elementi**: `int a[10];`

Per accedere ad un dato elemento del vettore dobbiamo scrivere il **nome del vettore** seguito da un **valore** (operazione di **indicizzazione**), gli elementi del vettore vengono sempre contati a partire da 0 ad n-1 : `a[0], a[1], ..., a[9];`

INPUT DI ARRAY	OUTPUT DI ARRAY
<pre>void input_array(int *a, int n){ int i; for(i=0; i<n; i++) scanf("%d", &a[i]); }</pre>	<pre>void output_array(int *a, int n){ int i; for(i=0; i<n; i++) printf("%d ", a[i]); }</pre>

Per eliminare un elemento dell'array lo si elimina e si spostano, se esistono, tutti gli elementi successivi di una posizione a sinistra.

▪ **Analisi:**

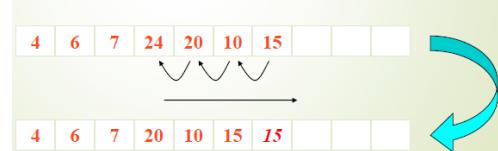
- Dati di ingresso: **Array a di n elementi, posizione pos**
 - Precondizione: $0 \leq pos < n$
- Dati di uscita: **Array a1 di n1 elementi**
 - Postcondizione: $\forall i \in [0, pos-1] a1[i] = a[i] \text{ AND } \forall j \in [pos, n-1] a1[j] = a[j+1] \text{ AND } n1 = n-1$

Identificatore	Tipo	Descrizione
<u>Dizionario dei dati</u>		
a	array	array di interi in input
n	intero	numero di elementi nell'array a
pos	intero	indice dell'elemento da eliminare
a1	array	array di interi in output
n1	intero	numero di elementi nell'array a1

▪ **Progettazione:**

Si spostano indietro tutti gli elementi dell'array a compresi tra le posizioni pos+1 e n-1 e si decrementa n.

Esempio: eliminare l'elemento in posizione 3



Errore tipico:

eseguire il ciclo in senso decrescente, il risultato sarà il ricopimento a sinistra dell'elemento finale.

▪ **Implementazione:**

```
/*precondizione deve essere controllata dalla funzione chiamante*/
void delete(int *a, int *n, int pos){           //array a, intero n di elementi dell'array, posizione pos da eliminare
    int i;
    for(i=pos; i<(*n)-1; i++)
        a[i] = a[i+1];
    (*n)--;                                     //Decrementiamo la taglia dell'array n
}
```

Per visitare (o analizzare) gli elementi dell'array si scorrono tutti gli elementi, esistono due tipi di visite:

- Visita totale:** vengono analizzati tutti gli elementi, in questo caso bisogna usare un ciclo e visitare gli elementi dell'array in un verso.
`for(i=0; i<n; i++) oppure for(i=n-1; i>=0; i--)`
- Visita finalizzata:** la visita termina quando un elemento dell'array verifica una certa condizione, in questo caso bisogna usare due condizioni di uscita, una sull'indice di scansione (visitati tutti gli elementi si esce) e l'altra che dipende dal problema specifico.

Per ricercare un dato elemento dell'array lo si scorre e si verifica che c'è o meno.

▪ **Analisi:**

- Dati di ingresso: **Array a di n interi, elemento el**
 - Precondizione: $n > 0$
- Dati di uscita: **Intero pos**
 - Postcondizione: se el è contenuto in a allora pos è la posizione della prima occorrenza di el in a altrimenti pos = -1

Identificatore	Tipo	Descrizione
<u>Dizionario dei dati</u>		
a	array	array di interi in input
n	intero	numero di elementi nell'array
el	intero	elemento da ricercare
pos	intero	indice dell'elemento trovato o -1

▪ **Progettazione:**

Si scorre l'array di input finché non si trova l'elemento o non si raggiunge la fine dell'array (**visita finalizzata**). Se l'elemento è stato trovato allora si restituisce la posizione corrente, in questo caso all'uscita del ciclo l'indice dell'array corrisponde a quello dell'elemento cercato. Altrimenti si restituisce "-1".

Si utilizza un indice per scorrere e una variabile booleana che dice se è stato trovato o meno l'elemento.

La condizione del ciclo sarà: `(i < n && !trovato)`

▪ **Implementazione:**

```
int ricerca(int a[], int n, int elem) {
    int i=0;                                /* indice dell'array */
    int trovato=0;                          /* nello schema di visita indica che è stato trovato */
    while(i<n && !trovato)                 /* visita finalizzata */
        if (a[i] == elem)
            trovato=1;                      /* permette di uscire dal ciclo */
        else i++;                           /* se non trovato incrementa l'indice */
    /* se trovato restituisce la posizione i dell'elemento altrimenti restituisce -1 */
    return (trovato ? i : -1);
}
```

SHORT CIRCUIT EVALUATION:

Invece dello schema di visita finalizzata precedente, lo si può ridurre nel seguente modo:

`while(i < n && a[i] != elem) i++;` (codice completo sotto)

Se la condizione a sinistra fosse falsa la condizione a destra non viene valutata.

RICERCA BINARIA:

Per quanto riguarda la ricerca di un elemento dell'array, se quest'ultimo fosse ordinato, non sarebbe necessario arrivare alla fine dell'array per stabilire che l'elemento è stato trovato o meno, ci si può fermare appena si trova un elemento maggiore (o uguale) di quello dato.

Ovviamente, se l'elemento è maggiore di tutti quelli presenti nell'array, allora si visiterà l'intero array (caso peggiore).

Esempio:

4	6	7	10	12	15	18	20	24	30
---	---	---	----	----	----	----	----	----	----

La ricerca di 13 e di 15 terminano quando l'elemento corrente è 15, mentre quella di 40 termina quando si sono visitati tutti gli elementi.

La **ricerca lineare in un array ordinato** può essere quindi effettuata diversamente.

Prendiamo in considerazione l'implementazione precedente ma in forma **short-cut evaluation**:

```
int ricercaord(int a[], int n, int elem) {
    int i = 0;
    while(i < n && a[i] < elem)           // visita finalizzata
        i++;
    return (a[i]==elem ? i : -1);
}
```

Una nuova implementazione consiste nel dividere l'array in due metà e confrontare l'elemento da cercare con l'elemento centrale dell'array:

- se questi due elementi sono uguali allora è stato già trovato (e ci fermiamo);
- se l'elemento da cercare è minore del centrale allora si cerca nella prima metà dell'array;
- se l'elemento da cercare è maggiore del centrale allora si cerca nella seconda metà dell'array.

Se l'elemento non è presente, l'array si ridurrà ad un solo elemento, non divisibile in due (terminazione), nel caso peggiore si visitano $\log_2 n$ elementi dell'array.

L'implementazione della **ricerca binaria** in un array ordinato sarà la seguente:

```
int ricercabin(int num, int arr[], int n){
    int begin = 0, end = n-1, center;
    while(end >= begin){
        center = (begin+end)/2;
        if(num==arr[center])
            return center;
        else if (num < arr[center])
            end = center -1;
        else if (num > arr[center])
            begin = center + 1;
    }
    return -1;
}
```



L01.3. ORDINAMENTO

Il Problema dell'Ordinamento è quello di elencare gli elementi di un insieme secondo una sequenza stabilita da una relazione d'ordine.

Gli algoritmi di ordinamento hanno determinate proprietà:

- **Stabile**, due elementi con la medesima chiave mantengono lo stesso ordine con cui si presentavano prima dell'ordinamento;
- **In loco**, in ogni istante al più è allocato un numero costante di variabili, oltre all'array da ordinare;
- **Adattivo**, il numero di operazioni effettuate dipende dall'input;
- **Interno** (dati contenuti nella memoria RAM) vs **esterno** (dati contenuti su disco o nastro o file).

Algoritmi di ordinamento semplici

Numero di operazioni quadratico rispetto alla taglia dell'input $O(n^2)$:

- Selection sort
- Insertion sort
- Bubble sort

Algoritmi di ordinamento avanzati

Numero di operazioni rispetto alla taglia dell'input: $O(n \log n)$:

- Merge sort
- Quick sort (caso medio $O(n \log n)$, caso peggiore $O(n^2)$)

SELECTION SORT:

L'algoritmo consiste nella ricerca del minimo valore, presente nel vettore, e posizionarlo nella prima posizione, cercare il secondo minimo e metterlo nella seconda posizione e così via, il procedimento va eseguito n volte se n è la dimensione del vettore. Come struttura dati di ingresso necessita di un vettore. Tale algoritmo ha un approccio incrementale, pertanto il vettore concettualmente sarà suddiviso in due sotto-vettori:

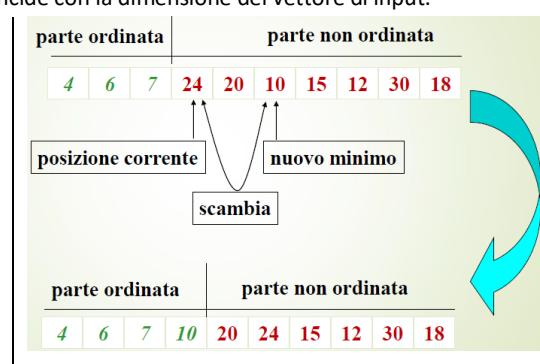
- Il **sotto-vettore sinistro**, che contiene gli elementi ordinati, inizialmente è vuoto;
- Il **sotto-vettore destro**, contiene gli elementi da ordinare, inizialmente coincide con il vettore in input.

Si ha la terminazione dell'algoritmo nel caso in cui la dimensione del sotto-vettore sinistro coincide con la dimensione del vettore di input.

```
void selection_sort(int *a, int n) {
    int i, m;
    for (i=0; i<n-1; i++) {
        m = min(&a[i], n-i)+i;
        swap(&a[i], &a[m]);
    }
}
```

```
int min(int *a, int n){
    int min=0, i;
    for(i=1; i<n; i++)
        if(a[i]< a[min])
            min=i;
    return min;
}

void swap(int *a, int *b){
    int temp=*a;
    *a=*b;
    *b=temp;
}
```

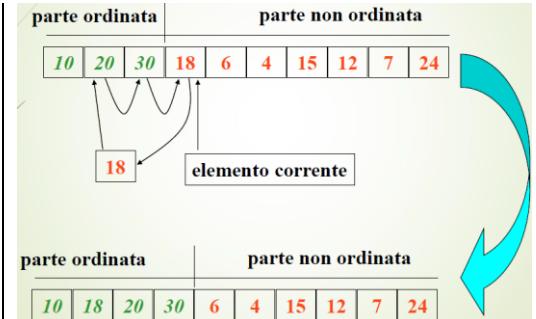


INSERTION SORT:

Effettuiamo una visita totale dell'array e ad ogni passo gli elementi che precedono l'elemento corrente sono ordinati, si inserisce l'elemento corrente nella posizione che garantisce il mantenimento dell'ordinamento e gli elementi precedenti se sono maggiori vengono spostati in avanti, il primo elemento è già "ordinato" e quindi si parte dal secondo. Si termina l'algoritmo quando il sotto-vettore ordinato ha dimensione uguale al vettore.

```
void insertion_sort(int *a, int n){
    int i;
    for(i=1; i<n;)
        insertion_sorted_array(a, a[i], &i);
}
```

```
void insertion_sorted_array(int *a, int val, int *n){
    int i;
    for(i=*n; i>0; i--)
        if(val<a[i-1])
            a[i]=a[i-1];
        else
            break;
    a[i]=val;
    (*n)++;
}
```



BUBBLE SORT:

Tale algoritmo ha un approccio incrementale, pertanto il vettore concettualmente sarà suddiviso in due sotto-vettori:

- **sotto-vettore sinistro**, che contiene gli elementi ancora da ordinare, inizialmente coincide con il vettore in input;
- **sotto-vettore destro**, contiene gli elementi già ordinati, inizialmente è vuoto.

Si ha la terminazione dell'algoritmo nel caso in cui la dimensione del sotto-vettore destro coincida con la dimensione del vettore di input.

Di questo algoritmo esistono due versioni:

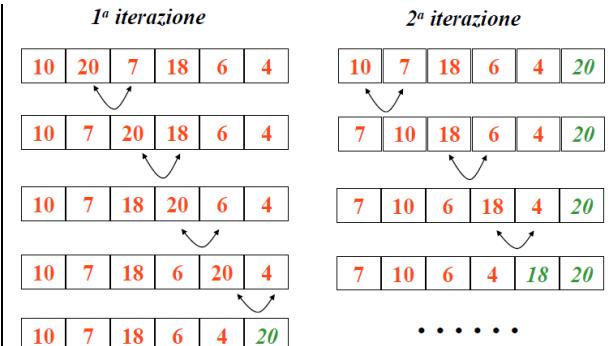
1. **Iterativo**, si effettuano $n-1$ visite dell'array, alla visita i -esima si confrontano elementi adiacenti dal primo al $(n-i)$ -esimo elemento e gli elementi adiacenti che non risultano ordinati vengono scambiati.

```
void bubble_sort(int *a, int n){
    int i, j;
    for(i=1; i<n; i++)
        for(j=0; j<n-i; j++)
            if(a[j]>a[j+1])
                swap(&a[j], &a[j+1]);
}
```

NB: ad ogni passo l'elemento più grande viene portato nella sua posizione finale, dopo il passo i -esimo, gli elementi tra le posizioni $n-i$ ed $n-1$ risultano ordinati e nelle loro posizioni finali.

2. **Adattivo**, se in una visita dell'array non è stato effettuato alcuno scambio, allora l'array è ordinato, in tal caso si può interrompere.

```
int adaptive_bubble_sort(int *a, int n){
    int i, j, sorted=0;
    for (i=1; i<n && !sorted; i++){
        sorted=1;
        for(j=0; j<n-i; j++)
            if(a[j]>a[j+1]){
                swap(&a[j], &a[j+1]);
                sorted=0;
            }
    }
    return i-1; //Ritorna il numero di scambi fatti
}
```



L02.1. TESTING DEI PROGRAMMI

Il testing e il debugging sono una parte dell'implementazione del software. **Definizione di Testing:** esercitare il programma con dati di test per verificare che il suo comportamento sia conforme a quello atteso (definito nella *specifica*).

- **Oracolo:** è l'output atteso, quello che ci si aspetta il programma produca;
- **Malfunzionamento:** comportamento del programma diverso da quello atteso.

L'obiettivo principale del testing è quello di individuare malfunzionamenti, che è causato da un difetto (errore o bug) nel codice. L'errore può essere introdotto in fase di analisi, specifica, progettazione o codifica.

Il **Debugging** serve per individuare e correggere quel difetto che ha causato il malfunzionamento, più alta è la fase in cui si introduce il difetto, maggiore è la difficoltà nel rimuoverlo. La ricerca di un difetto può essere fatta inserendo nel codice sorgente punti di ispezione dello stato delle variabili, una volta corretto il difetto, bisogna rieseguire tutti i casi di test.

Testare il programma con tutti i possibili dati di test è impraticabile, l'obiettivo è individuare **classi di dati di test**, selezionare un caso di test da ogni classe ed evitare casi di test ridondanti. Può essere utile creare una **test suite**, ovvero un insieme di casi di test per un programma.

Esempio:

In un ordinamento di un array bisogna tener conto di alcuni **aspetti** nella scelta dei casi di test, come ad esempio:

- Il numero n di elementi dell'array:
 - caso generale un array con $n > 1$;
 - caso particolare $n = 1$.
- La disposizione degli elementi:
 - caso generale un array non ordinato;
 - un array già ordinato in modo crescente/decrescente.

Considerati questi aspetti possiamo produrre dei **test case**:

- **Test case 1 – TC1** (un solo elemento)
 - *Array di input:* 5
 - *Oracolo:* 5
- **Test case 2 – TC2** (input ordinato in maniera crescente)
 - *Array di input:* 1 2 3 4 5 6 7 8 9
 - *Oracolo:* 1 2 3 4 5 6 7 8 9
- **Test case 3 – TC3** (input ordinato in maniera decrescente)
 - *Array di input:* 10 9 8 7 6 5 4 3 2 1
 - *Oracolo:* 1 2 3 4 5 6 7 8 9 10
- **Test case 4 – TC4** (non ordinato)
 - *Array di input:* 5 8 2 9 10 1 4 7 3 6 12 11
 - *Oracolo:* 1 2 3 4 5 6 7 8 9 10 11 12

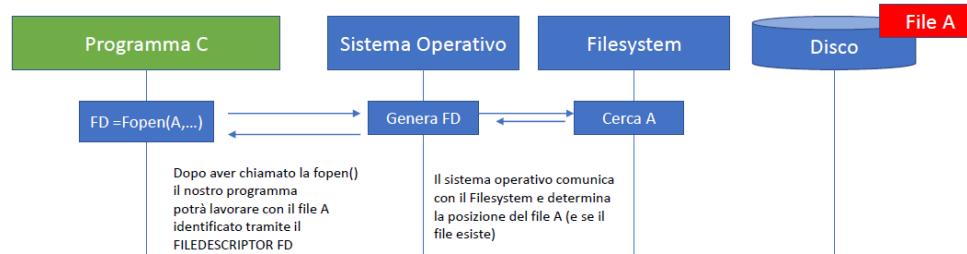
Per automatizzare il test si possono usare i **file** per leggere dati di input e scrivere i dati di output, semplificando così operazioni di test da ripetere.

L02.2. USO DEI FILE

In C, il termine **stream** (o flusso) indica una sorgente di input o una destinazione per l'output. Molti programmi (piccoli) ottengono il loro input da uno stream (ad es. la tastiera) e lo inviano ad un altro stream (ad esempio il video), programmi più grandi possono avere necessità di usare più stream.

Gli stream rappresentano file memorizzati da qualche parte (hard disk o altri tipi di memoria a lungo termine) e sono associati a periferiche (schede di rete, stampanti, ecc...).

Per accedere ad un file su disco è necessario chiedere al sistema operativo di restituirci un **FILE DESCRIPTOR**, ovvero un identificatore che, nel nostro programma, sarà collegato al file stesso.



Per richiedere un **FILE DESCRIPTOR**, si utilizza la funzione **fopen** che restituisce un puntatore ad una struttura FILE (il file descriptor).

Per utilizzare i file dobbiamo **sempre** effettuare due operazioni:

1. **FILE *fopen(const char *filename, const char *mode)**, *filename* è il nome (drive o percorso) del file da aprire, *mode* è una "stringa di modalità" che specifica quali operazioni abbiamo intenzione di compiere sul file, il valore di ritorno è un puntatore a FILE.
2. **Int fclose(FILE *stream)**, ritorna 0 in caso di successo ed EOF in caso di fallimento.

Per quanto riguarda la stringa **mode**, può assumere diversi valori:

“r”	Apre il file in lettura (il file deve esistere)
“w”	Apre il file in scrittura (non è necessario che il file esista)
“a”	Apre il file in accodamento (non è necessario che il file esista)
“r+”	Apre il file in lettura e scrittura (il file deve esistere)
“w+”	Apre il file in lettura e scrittura -tronca il file se esiste
“a+”	Apre il file in lettura e scrittura -accoda se il file esiste

Per effettuare **input e output da stream** esistono diverse funzioni:

1. `char *fgets(char *s, int size, FILE *stream)`, legge da stream fino al carattere newline (o finché size-1 caratteri sono stati letti) e li memorizza in s.
Ritorna s in caso di successo, NULL in caso di errore o se si raggiunge la fine del file senza aver letto alcun carattere.
2. `Int fscanf(FILE *stream, const char *format, ...)`, legge da stream fino ad un carattere di spazio e non lo memorizza.
Restituisce il numero di dati letti e scritti con successo.
3. `Int fprintf(FILE *stream, const char *format, ...)`, scrive su stream.
Restituisce il numero di caratteri scritti, -1 in caso di errore.

Per effettuare **input e output su stringhe** utilizziamo due funzioni:

1. `Int sprintf(char *str, const char *restrict format, ...)`, differisce dalla funzione printf() in quanto i caratteri vengono scritti nell'area puntata da str.
Naturalmente str deve essere ampia a sufficienza per contenere i caratteri in output più il terminatore '\0'.
Restituisce il numero di caratteri memorizzati.
2. `Int sscanf(const char *str, const char *format, ...)`, legge i caratteri da una stringa, ha lo stesso comportamento della funzione scanf(), ma l'input avviene da str. Restituisce il numero di dati letti e scritti con successo.

Esempio uso file:

```
fgets(str, sizeof(str), stdin);           /* legge una riga dell'input */  
sscanf(str, "%d%d", &i, &j);           /* estrae due interi */
```

La prima funzione permette di leggere un'intera riga da input (fino a newline), mentre la seconda permette di estrarre due interi dalla stringa.

Esempio testing ordinamento:

Usiamo 2 file per i dati di input:

1. Input.txt, contenente gli elementi dell'array di input;
2. Oracle.txt, contenente gli elementi dell'array ordinato.

Mentre per l'output:

1. Output.txt, risultante dall'esecuzione del programma (output effettivo), oltre ad una indicazione dell'esito del test (PASS / FAIL).

input.txt	oracle.txt
5	5
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
10 9 8 7 6 5 4 3 2 1	1 2 3 4 5 6 7 8 9 10
5 8 2 9 10 1 4 7 3 6 12 11	1 2 3 4 5 6 7 8 9 10 11 12

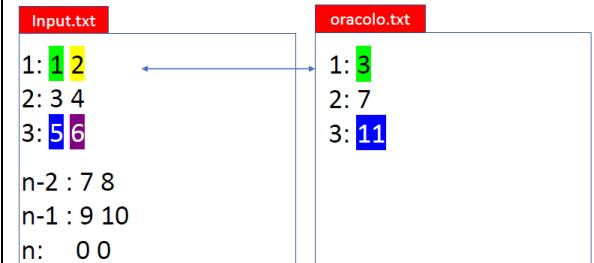
Esempio file di testing somma:

Per ottimizzare il testing, possiamo pensare di scrivere un programma **Driver** che prenda in input una o più coppie di interi da un file (una coppia per riga) e vi applichi, ad esempio, somma per registrarne l'output. Possiamo pensare di mantenere i valori attesi della somma in un file esterno (che chiameremo **Oracolo**) immettendo su ogni riga il valore corretto della somma corrispondente agli interi i1 e i2 presenti nel file di input alla riga i-esima.

Per quanto riguarda il driver (nel caso di somma) automatizza il processo di testing leggendo un file di input (input.txt), applicando sull'input la funzione che vogliamo (in questo caso la somma) ed infine confrontando il valore stampato dalla printf con il valore atteso.

Se in qualsiasi momento il risultato della funzione somma non è quella aspettata (quella definita in oracolo) il driver si arresta e restituisce fallito. Se tutto va bene, il driver restituisce «superato».

```
#include <stdio.h>  
Int somma( int a, int b )  
    return a+b  
}  
void main (){  
    FILE *input;  
    FILE *oracolo;  
    input = fopen (" input.txt ","r");  
    oracolo = fopen("oracolo.txt", "r");  
    int i1, i2, esito=1;  
    while(fscanf(input,"%d %d \n",&i1,&i2)!=EOF) {  
        int r = somma(i1,i2), or;  
        fscanf( oracolo,"%d \n",&or);  
        if(or!=r){  
            esito=0;  
            break;  
        }  
    }  
    fclose(input);  
    fclose(oracolo);  
    if(esito==0) printf("fallito");  
    else printf("superato");  
}
```



L02.3. PUNTATORI E ALLOCAZIONE DINAMICA DELLA MEMORIA

Gli array sono caratterizzati da una **cardinalità**, ovvero dalla dimensione massima, e da **riempimento**, ovvero la dimensione utilizzata in un certo momento che può variare in base alle operazioni di inserimento e rimozione.

Quando usiamo un array a dimensione **statica** dobbiamo prevedere però una cardinalità molto grande e sufficiente per tutte le esecuzioni del programma, questo però ne consegue uno spreco di memoria in quanto è possibile che non utilizziamo tutta la memoria allocata.

Con l'**allocazione dinamica** occupiamo solo la memoria necessaria, permettendo di creare strutture dati la cui dimensione varia durante l'esecuzione.

Per usare l'allocazione dinamica esistono funzioni specifiche fornite dalla libreria *stdlib*, in particolare 3 funzioni:

1. `void *malloc(size_t size);`

Allocà un blocco di memoria di size bytes senza inizializzarlo, `size_t` è un tipo intero senza segno definito nella libreria del C.
Restituisce il puntatore al blocco.

2. `void *calloc(size_t nelements, size_t elementSize);`

Allocà un blocco di memoria di nelements* elementSize bytes e lo inizializza a 0 (clear) e restituisce il puntatore al blocco.

3. `void *realloc(void *pointer, size_t size);`

Cambia la dimensione del blocco di memoria precedentemente allocato puntato da pointer.

Restituisce il puntatore ad una zona di memoria di dimensione size, che contiene gli stessi dati della vecchia regione indirizzata da pointer, troncata alla fine nel caso la nuova dimensione sia minore di quella precedente.

Un **puntatore** è una variabile che contiene l'indirizzo di un'altra variabile e questi puntatori sono "type bound" cioè ad ogni puntatore è associato il tipo a cui il puntatore si riferisce.

Nella dichiarazione di un puntatore bisogna specificare un asterisco (*) prima del nome della variabile pointer: T *p. Esempio:

- `int *pointer; // puntatore a intero`
- `char *pun_car; // puntatore a carattere`
- `float *flt_pnt; // puntatore a float`

L'accesso all'oggetto puntatore avviene attraverso l'**operatore di deferenziazione** “*”. Esempio:

- `*pointer = 5 //assegna all'oggetto puntato da pointer il valore 5`
- `x = *flt_pnt //assegna il valore dell'oggetto puntato da flt_pnt alla variabile x`

Prima di poter usare un pointer questo deve essere inizializzato, ovvero deve contenere l'indirizzo di un oggetto.

Per ottenere l'indirizzo di un oggetto si usa l'operatore unario **&**. Esempio:

```
int i = 10, *p1;  
p1 = &i;  
printf("%d \n", *p1);
```

I **tipi di variabili** in C possono essere di vario tipo:

- **Globali**, dichiarate esternamente alle funzioni, sono visibili a tutte le funzioni la cui definizione segue la dichiarazione della variabile nel file sorgente e sono dette statiche, perché la loro allocazione in memoria avviene all'atto del caricamento del programma (e la loro deallocazione al termine del programma);
- **Locali**, dichiarate e visibili solo all'interno di una funzione;
- **Automatiche**, dichiarate in blocchi interni alle funzioni e vengono allocate in memoria a tempo di esecuzione (dell'istruzione dichiarativa) e deallocate al termine del blocco.

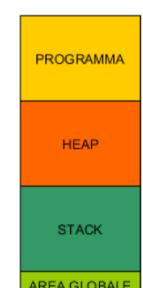
Quando si considera una variabile bisogna valutare tre aspetti importanti:

1. **Scope**, parte del programma in cui è attiva una dichiarazione (dice quando può essere usato un identificatore);
2. **Visibilità**, parte del programma in cui è accessibile una variabile (non sempre coincide con lo scope ...);
3. **Durata**, periodo durante il quale una variabile è allocata in memoria.

<code>int n;</code> ... <code>int main()</code> { long n; ... { double n; ... } ... }	<ul style="list-style-type: none">▪ <code>int n (globale)</code><ul style="list-style-type: none">▪ <code>scope: intero file</code>▪ <code>visibilità: non è visibile nel main</code>▪ <code>long n (locale)</code><ul style="list-style-type: none">▪ <code>scope: main</code>▪ <code>visibilità: non è visibile nel blocco interno</code>▪ <code>double n (automatica)</code><ul style="list-style-type: none">▪ <code>scope: blocco interno</code>▪ <code>visibilità: coincide con lo scope</code>
--	--

Il Sistema Operativo riserva ad un processo (un programma in esecuzione), un segmento di memoria RAM. Questo, in generale è suddiviso in quattro distinte aree di memoria:

1. **L'area del programma**, che contiene le istruzioni macchina del programma;
2. **L'area globale**, che contiene le costanti e le variabili globali;
3. **Lo stack**, che contiene la pila dei record di attivazione creati durante ciascuna chiamata delle funzioni;
4. **L'heap**, che contiene le variabili allocate dinamicamente.



Variabili globali definite in un file F1 possono essere usate in un file F2 dichiarandole **extern** in F2:

```
extern int n;
```

NB: con la dichiarazione extern non si definisce la variabile e non si alloca memoria

Dichiarazioni **static** di variabili globali le rendono private al file in cui sono dichiarate:

```
static int n;
```

Dichiarazioni **static** di funzioni le rendono private al file in cui sono dichiarate, ossia ne modificano lo scope. Nel modulo vettore, la funzione minimo è usata solo localmente al modulo dalla funzione ordina_array.

Per renderla privata al modulo basta aggiungere la dichiarazione static:

```
static int minimo(int a[], int n);
```

vettore.h

```
int ricerca_array(int a[], int n, int elem);  
int minimo_array(int a[], int n);  
...
```

vettore.c

```
#include <stdio.h>  
#include "utile.h" // contiene funzione scambia  
static int minimo(int a[], int n); // dichiarazione locale  
int ricerca_array(int a[], int n, int elem) { ... }  
int minimo_array(int a[], int n) { ... }
```

L03.1. ABSTRACT DATA TYPES

Abbiamo detto che l'**astrazione** è un procedimento mentale che ci consente di evidenziare le caratteristiche principali e trascurare aspetti "secondari", e che esistono due tipi: **funzionale e procedurale**, che ha la finalità di ampliare l'insieme dei modi di operare sui tipi di dati già disponibili attraverso la definizione di nuovi operatori (delegano operazioni a nuovi sotto-programmi), e quella sui **dati**, che ha la finalità di ampliare i tipi di dati disponibili attraverso l'introduzione sia di nuovi tipi di dati che di nuovi operatori.

Un **tipo di dati** è definito da un **dominio di valori** e un insieme di **operazioni** previste su quei valori, ad esempio il tipo interi può essere definito da un range di valori come da 1 a 9, mentre le operazioni possono essere "+, -, *, /, ...". Nel linguaggio C esistono vari tipi di dati:

- **Dati primitivi**, forniti direttamente dal linguaggio: int, char, float, double;
- **Dati aggregati**, array, strutture, enumerazioni, unions;
- **Puntatori**.

Unendo questi due concetti, ovvero astrazione e tipi di dati, otteniamo i "**tipi di dati astratti**" (ADT), quest'ultimo è un tipo di dati che estende dei dati esistenti, definito distinguendo **specifiche** e **implementazione**:

- **Specifiche**, viene fatta definendo il tipo dei dati e l'insieme degli operatori di cui si disporrà su questi dati. Esistono due tipi di specifiche:
 1. **Sintattica**, vengono definite delle regole, ovvero **nomi e tipi**;
 2. **Semantica**, vengono definiti i significati, ovvero **valori e vincoli**.
- **Implementazione**, indipendente dalla specifica, è la codifica di quanto viene definito nella specifica, questa fase è spesso nascosta al programmatore, seguendo il principio dell'**incapsulamento** (information hiding).

Definiremo i tipi di dati astratti usando una tabella che ci semplifica il lavoro della specifica, ed è strutturata in questo modo:

	Sintattica	Semantica
Tipi di dati	<ul style="list-style-type: none">• Nome dell'ADT• Tipi da dati già usati	<ul style="list-style-type: none">• Insieme dei valori
Operatori: Per ogni operatore	<ul style="list-style-type: none">• Nome dell'operatore• Tipi di dati di input e di output	<ul style="list-style-type: none">Funzione associata all'operatore<ul style="list-style-type: none">• Precondizioni: definiscono quando l'operatore è applicabile• Postcondizioni: definiscono relazioni tra dati di input e output

Esempio tipo dati astratto chiamato "punto":

Sintattica	Semantica
<p>Nome del tipo: Punto Tipi usati: Reale</p> <p>creaPunto (reale, reale) → punto</p> <p>ascissa (punto) → reale</p> <p>ordinata (punto) → reale</p> <p>distanza (punto, punto) → reale</p>	<p>Dominio: Insieme delle coppie (ascissa, ordinata) dove ascissa e ordinata sono numeri reali</p> <p>creaPunto(x, y) = p<ul style="list-style-type: none">• pre: true• post: p = (x, y)</p> <p>ascissa(p) = x<ul style="list-style-type: none">• pre: true• post: p = (x, y)</p> <p>ordinata(p) = y<ul style="list-style-type: none">• pre: true• post: p = (x, y)</p> <p>distanza(p1, p2) = d<ul style="list-style-type: none">• pre: true• post: d = sqrt((ascissa(p1)-ascissa(p2))^2 + (ordinata(p1)-ordinata(p2))^2)</p>

ADT STRUCT:

Per implementare gli ADT in C, un primo costrutto è la **Struttura (struct)**, ovvero un tipo dati composito che include un elenco di variabili fisicamente raggruppate in un unico blocco di memoria, ha il vantaggio che migliora la leggibilità dei programmi:

```
int main(){  
    struct point {          //Definizione della struttura  
        float x;            //Campi della struttura  
        float y;  
    };  
  
    struct point p;          //Variabile di tipo struttura  
    p.x= 2.0;                //Per accedere ai campi si usa la "dot sintax"  
    p.y= 3.0;  
    printf("coordinate del punto: (%.1f, %.1f)", p.x, p.y);  
    struct point p1 = {2.0, 3.0}; //Inizializzazione della struttura  
}
```

Si può usare il **typedef** sulla struttura precedentemente definita:

`typedef struct point Point;`

Oppure usare in combinazione il **typedef** e la definizione della struttura:

```
typedef struct{  
    float x;  
    float y;  
} Point;  
  
int main(){  
    Point p = {2.0, 3.0};  
    printf("coordinate del punto: (%.1f, %.1f)", p.x, p.y);  
}
```

Possiamo anche allocare memoria dinamica per l'ADT struttura utilizzando le consuete funzioni:

`Point *p = malloc(sizeof(Point));`

È possibile accedere ai campi della struttura da un puntatore usando l'operatore **freccia ->**:

```
int main(){  
    Point *p = malloc(sizeof(Point));  
    p->x = 2.0;  
    p->y = 3.0;  
    printf("coordinate del punto: (%.1f, %.1f)", p->x, p->y);  
}
```

Esempio completo di implementazione dell'ADT punto:

punto.h

```
typedef struct{
    float x;
    float y;
}Punto;
```

```
Punto creaPunto(float x, float y);
float ascissa(Punto p);
float ordinata(Punto p);
float distanza(Punto p1, Punto p2);
```

```
typedef struct punto *Punto;
```

```
Punto creaPunto(float x, float y);
float ascissa(Punto p);
float ordinata(Punto p);
float distanza(Punto p1, Punto p2);
```

Questa implementazione lascia un **problema**, ovvero l'implementazione della struttura del tipo punto è nell'header file, visibile quindi al modulo client, che potrebbe quindi accedere direttamente ai campi della struct senza usare gli operatori dell'ADT.

Per soddisfare l'**information hiding** spostiamo l'implementazione della struttura Punto dall'header file al file di implementazione delle funzioni (**punto.c**).

Una volta spostata l'implementazione della struttura, nell'header file definiamo il tipo Punto come **puntatore** alla struttura.

NOTA: All'atto della compilazione del modulo client, essendo il tipo punto un puntatore, il compilatore sa quanta memoria deve allocare per una variabile di quel tipo, indipendentemente dalla dimensione dell'elemento puntato.

L03.2. PSEUDO-GENERICs IN C

I **Generics** sono uno strumento che permette la definizione di un tipo parametrizzato, che viene esplicitato successivamente in fase di compilazione (o linkaggio) secondo le necessità, permettono di eseguire algoritmi su tipi di dati diversi e applicare ADT su tipi di dati diversi.

Per operare su tipi diversi di dati bisognerebbe modificare l'algoritmo vero e proprio, ad esempio per il **bubble sort** se vogliamo ordinare delle stringhe:

Bubble sort su interi:

```
void swap_int(int*a, int*b){
    int temp= *a;
    *a = *b;
    *b = temp;
}
void bsort_int(int a[], int n){
    int i, j;
    for(i=1; i<n; i++)
        for(j=0; j<n-i; j++)
            if(a[j] > a[j+1])
                swap_int(&a[j], &a[j+1]);
}
```

Main bubble sort su interi:

```
int main(){
    int i, n = 5;
    int arr[n];
    printf("Introduci il vettore: ");
    for(i=0; i<n; i++)
        scanf("%d",&arr[i]);
    bsort_int(arr, n);
    printf("Vettore ordinato: ");
    for(i=0; i<n; i++)
        printf("%d ",arr[i]);
}
```

Bubble sort su stringhe:

```
void swap_string(char**a, char**b){
    char *temp= *a;
    *a = *b;
    *b = temp;
}
void bsort_string(char*a[], int n){
    int i, j;
    for(i=1; i<n; i++)
        for(j=0; j<n-i; j++)
            if(strcmp(a[j], a[j+1])>0)
                swap_string(&a[j], &a[j+1]);
}
```

Main bubble sort su stringhe:

```
int main(){
    Int i, n = 5;
    char*arr[5];
    printf("Introduci il vettore: ");
    for(i=0; i<n; i++){
        arr[i] = malloc(20*sizeof(char));
        scanf("%s",arr[i]);
    }
    bsort_string(arr, n);
    printf("Vettore ordinato: ");
    for(i=0; i<n; i++)
        printf("%s ",arr[i]);
}
```

Quindi il nostro obiettivo è quello di realizzare algoritmi e/o ADT che siano in grado di funzionare con tipi di dati diversi. Una **soluzione** generale è quella di generalizzare il nostro algoritmo, in modo che possa funzionare con tipi diversi (interi, stringhe o qualunque struttura). Si procede così:

1. Creare un tipo "Item" (interfaccia, file .h) che supporti input, output e confronto;
2. Modificare le librerie in modo che operino sul tipo Item;
3. Realizzare Item in file .c che supportano le varianti intero, stringa e struttura;
4. Linkare ed eseguire separatamente (con make) le varianti.

Per implementare quanto detto in C si procede on questo modo:

Questo "Item" lo definiamo come puntatore a void, quindi un generico puntatore per il quale non si specifica il tipo a cui punta: **void *p_void;**

E poi assegnarlo al tipo desiderato:

- **int* p_int= p_void;**
- **char* p_char= p_void;**
- **struct studente{**
 - char nome[20];**
 - int matricola;****};**
- typedef struct studente *Studente;**
- Studente p_studente= p_void;**

L04. ADT LISTA

Il tipo astratto **Lista** è una sequenza di elementi di un *determinato tipo* (*anche Generics*), in cui è possibile aggiungere o togliere elementi, è possibile specificare la posizione relativa nella quale l'elemento va aggiunto o tolto.

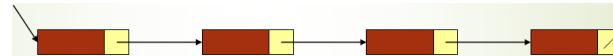
Liste concatenate

Dimensione variabile, accesso diretto solo al primo elemento della lista.

Per accedere ad un generico elemento, occorre **scandire sequenzialmente** gli elementi della lista:

- Per accedere all'i-esimo elemento occorre scorrere la lista dal primo all'i-esimo elemento (*tempo max proporzionale ad n*);
- Dato un elemento, è possibile eliminarlo o aggiungerne uno dopo direttamente (*tempo costante*).

Per quanto riguarda la progettazione di liste concatenate, ogni elemento di una lista concatenata è un **record** con un campo puntatore che serve da collegamento per il record successivo.



Si accede alla struttura attraverso il puntatore al primo record ed il campo puntatore dell'ultimo record contiene il valore **NULL**.

Sintattica	Semantica
Nome del tipo: List Tipi usati: Item, boolean	Dominio: insieme di sequenze $L = a_1, \dots, a_n$ di tipo Item L'elemento nil rappresenta la lista vuota
newList() \rightarrow List	newList() \rightarrow L • Post: L = nil
isEmpty(List) \rightarrow boolean	isEmpty(L) \rightarrow b • Post: se L=nil allora b = true altrimenti b = false
addHead(List, Item) \rightarrow List	addHead(L, e) \rightarrow L' • Post: L = <a1, a2, ..., an> AND L' = <e, a1, ..., an>
removeHead(List) \rightarrow List	removeHead(L) \rightarrow L' • Pre: L = <a1, a2, ..., an> n>0 • Post: L' = <a2, ..., an>
getHead(List) \rightarrow Item	getHead(L) \rightarrow e • Pre: L = <a1, a2, ..., an> n>0 • Post: e = a1

Vediamo come implementare il tipo **Lista**:

Per prima cosa occorre dichiarare il tipo lista (*rispettando l'inf. hiding*):

```
typedef struct list *List;      //NOTA: si troverà nel file .h
struct list{
    //NOTA: si troverà nel file .c
    int size;
    //Aggiornata di volta in volta
    struct node *head;          //Puntatore al nodo testa
};
```

Per istanziare la lista occorre riservare memoria e inizializzarla vuota:

```
List list = malloc(sizeof(struct list));
list->size = 0;
list->head = NULL;
```

Per usare una lista concatenata serve una struttura che rappresenti i **nodi**:

```
struct node {
    //NOTA: si troverà nel file .c
    Item value;                //dati nel nodo
    struct node *next;          //puntatore al prossimo nodo
};
```

La struttura conterrà i dati necessari ed un puntatore al prossimo elemento della lista.

Per iterare sui nodi si può usare un ciclo for:

```
for(p = list->head; p != NULL; p = p->next)
    outputItem(p->value);
```

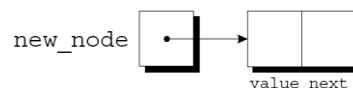
Man mano che costruiamo la lista, creiamo dei nuovi nodi da aggiungere alla lista. I passi per creare un nodo sono:

1. **Allocare** la memoria necessaria;
2. **Memorizzare** i dati nel nodo;
3. **Inserire** il nodo nella lista.

Per creare un nodo ci serve un puntatore temporaneo che punti al nodo:

Possiamo usare malloc per allocare la memoria necessaria e salvare l'indirizzo in new_node:

new_node adesso punta ad un blocco di memoria che contiene la struttura di tipo node:

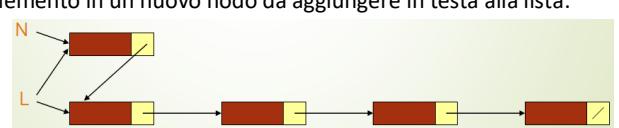


```
struct node *new_node;
```

```
new_node = malloc(sizeof(struct node));
```

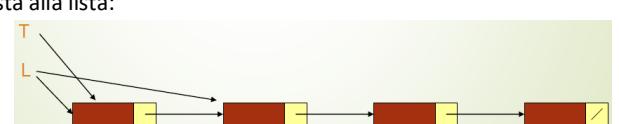
Per **inserire un nuovo elemento in una lista concatenata semplice** L è inserire l'elemento in un nuovo nodo da aggiungere in testa alla lista:

1. Si alloca il nuovo nodo N;
2. Si aggiunge il collegamento con il record iniziale della lista;
3. Si aggiorna L facendolo puntare a N.



Per **eliminare un elemento in una lista concatenata semplice** L è eliminarlo in testa alla lista:

1. Si crea un puntatore temporaneo T, copia di L
2. Si aggiorna L facendolo puntare al successivo di L
3. Si elimina il nodo puntato da T, liberando la memoria



Alcuni operatori richiedono una **visita parziale** o **totale della lista**, ad esempio:

- **Visita totale**: usata per calcolare la size, dove scorriamo i nodi incrementando un contatore (*ottimizzazione*: mantenere un contatore da aggiornare), oppure la stampa degli elementi;
- **Visita parziale**: inserimento o rimozione in una posizione i oppure la ricerca di un elemento.

Adesso implementiamo alcune funzioni per una **lista concatenata più complessa**.

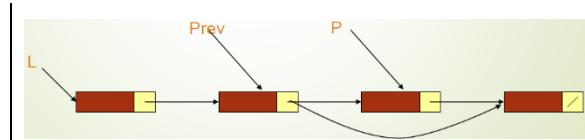
Sintattica	Semantica
Nome del tipo: List Tipi usati: Item, boolean	Dominio: insieme di sequenze $L = a_1, \dots, a_n$ di tipo Item L'elemento nil rappresenta la lista vuota
searchItem(List, Item) \rightarrow int	searchItem(L, i) \rightarrow pos • Post: se i in L allora pos = pos. di i in L else pos = -1
removeItem(List, Item) \rightarrow List	removeItem(L, e) \rightarrow L' • Pre: $L = <a_1, a_2, \dots, e, \dots, a_n>$ $n > 0$ • Post: $L' = L - <e>$
removeItem(List, int) \rightarrow List	removeItem(L, pos) \rightarrow L' • Pre: $L = <a_1, a_2, \dots, a_{pos}, \dots, a_n>$ $1 \leq pos \leq n$ • Post: $L' = L - <a_{pos}>$
insertItem(List, Item, int) \rightarrow List	insertItem(L, e, pos) \rightarrow L' • Pre: $L = <a_1, \dots, a_n>$ & $1 \leq pos \leq n+1$ • Post: $L' = <a_1, \dots, a_{pos}, \dots, a_{n+1}>$ & $a_{pos} = e$
insertTail(List, Item) \rightarrow List	insertTail(L, e) \rightarrow L' • Post: $L = <a_1, \dots, a_n>$ & $L' = <a_1, \dots, a_n, e>$
reverseList(List) \rightarrow List	reverseList(L) \rightarrow L' • Post: $L = <a_1, a_2, \dots, a_n>$ AND $L' = <a_n, \dots, a_2, a_1>$
cloneList(List) \rightarrow List	cloneList(L) \rightarrow L' • Post: $L = <a_1, a_2, \dots, a_n>$ AND $L' = <a_1, a_2, \dots, a_n>$

Per quanto riguarda la **ricerca di un elemento**, dati una lista L e un elemento val , restituisce la posizione della lista in cui appare la prima occorrenza dell'elemento, oppure -1 se l'elemento non è presente.

Richiede una **visita finalizzata** della lista, cioè usciamo da ciclo quando troviamo l'elemento cercato oppure quando raggiungiamo la fine della lista e possiamo ottenere sia il riferimento all'Item ricercato, sia la sua posizione, implementando la funzione: **Item searchItem(List list, Item item, int*pos)**

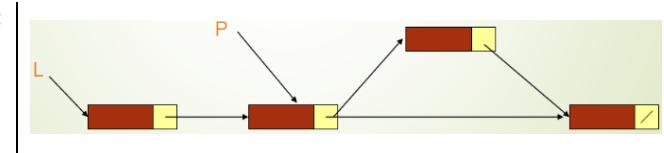
Per **eliminare un elemento** (esempio il terzo) in una lista concatenata L :

1. Si fanno avanzare due puntatori $Prev$ e P fino a che P punta al nodo da eliminare;
2. Si aggiorna il nodo puntato da $Prev$, facendolo puntare al successivo di P ;
3. Si elimina il nodo puntato da P , liberando la memoria.



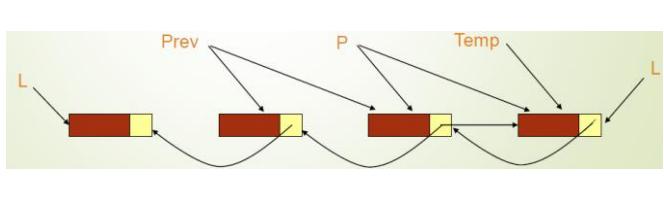
Per **inserire un elemento** (esempio in terza posizione) in una lista concatenata L :

1. Si fa avanzare un puntatore P , fino a che punta al nodo precedente alla posizione dell'inserimento;
2. Si crea il nuovo nodo e lo si fa puntare al successivo di P ;
3. Si fa puntare P al nuovo nodo.



Per **invertire una lista concatenata L** :

1. **Per ogni nodo** (con i nodi fino a $Prev$ già invertiti) si utilizzano due puntatori $Prev$ e P :
 - Si salva il successivo di P in un puntatore temporaneo $Temp$
 - Si aggiorna il nodo puntato da P , facendolo puntare a $Prev$
 - Si fanno avanzare P e $Prev$
2. Si aggiorna la testa facendola puntare all'ultimo nodo

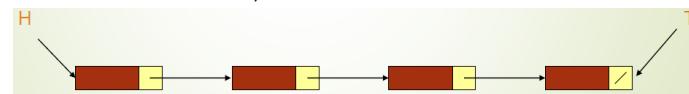


Per **clonare una lista concatenata** esistono diverse scelte progettuali:

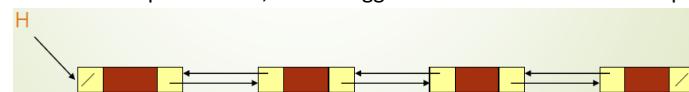
- **Clonare solo la struct list**, i nodi sono gli stessi ed una modifica su una lista viene riflessa nell'altra;
- **Clonare i nodi ma non gli item**, modifiche alla struttura di una lista non vengono riflesse nell'altra, ma se si modifica un item, risulta modificato in entrambe le liste;
- **Clonare i nodi e gli item**, le liste una volta clonate sono totalmente indipendenti.

Per concludere, esistono implementazioni alternative della struttura List:

- **Lista a collegamento singolo**: manteniamo la struttura lista vista precedentemente, aggiungendo un puntatore **tail** alla fine della lista, questo ci consente di ottimizzare l'operazione di inserimento in coda;



- **Lista circolare**: l'ultimo nodo ha un collegamento al primo nodo della lista, il vantaggio è di poter iterare lungo tutta la lista partendo da un nodo qualsiasi;
- **Lista a collegamento doppio**: ogni nodo ha due puntatori, uno al nodo precedente ed uno al successivo, il vantaggio è che possiamo effettuare operazioni di cancellazione e inserimento in tempo costante, lo svantaggio è che il codice diventa complesso.



Tutte queste possibili implementazioni possono essere combinate.

item.h

```
typedef void* Item;
```

```
Item inputItem();
void outputItem(Item);
int cmpItem(Item,Item);
Item cloneItem(Item);
```

lista.h

```
#include "item.h"
```

```
typedef struct list *List;
```

```
List newList();
int isEmpty(List);
void addHead(List, Item);
Item removeHead(List);
Item getHead(List);
int sizeList(List);
void printList(List);
void sortList(List);
Item searchList(List, Item, int *);
Item removeListItem(List, Item);
Item removeListItemPos(List, int );
int addListItem(List, Item, int);
int addListTail(List, Item);
void reverseList(List);
List cloneList(List);
```

lista.c

```
#include <stdio.h>
#include <stdlib.h>
#include "item.h"
#include "list.h"
#include "utils.h"
```

```
struct list {
    int size;
    struct node *head;
};
```

```
struct node {
    Item item;
    struct node *next;
};
```

```
struct node * minimo (struct node *p);
```

```
List newList(){
    List list = malloc(sizeof(struct list));
    list->size = 0;
    list->head = NULL;
    return list;
}
```

```
int isEmpty(List list){
    return list->head == NULL;
}
```

```
void addHead(List list, Item item){
    struct node *x = malloc(sizeof(struct node));
    x->next = list->head;
    x->item = item;
    list->head = x;           //Aggiorna head
    list->size++;            //Aggiorna size
}
```

```
Item removeHead(List list){
    Item app;
    if(isEmpty(list)==1){
        fprintf(stderr,"Lista vuota");
        return NULL;
    }
    struct node *temp = list->head;
    list->head = temp->next;
    app=temp->item;
    free(temp);
    list->size--;
    return app;
}
```

Item getHead(List list){

```
if(isEmpty(list)==1){
    fprintf(stderr,"Lista vuota");
    return NULL;
}
return list->head->item;
}
```

```
int sizeList(List list){
    return list->size;
}
```

```
void printList(List list){
    struct node *p;
    for(p = list->head; p != NULL; p = p->next){
        outputItem(p->item);
        printf("\n");
    }
}
```

```
void sortList(List list){
    struct node *p, *pos_minimo;
    for (p=list->head; p != NULL; p = p->next){
        pos_minimo = minimo(p);
        swap(&(pos_minimo->item), &(p->item));
    }
}
```

```
struct node * minimo (struct node *p){
    struct node *i, *min = p;
    for (i = p; i != NULL; i = i->next){
        if ((cmpItem(min->item, i->item)) > 0)
            min = i;
    }
    return min;
}
```

```
Item searchList(List list, Item item, int *pos){
    struct node *p;
    *pos=0;
    for (p=list->head; p != NULL; p = p->next){
        if(cmpItem(item,p->item) == 0)
            return p->item;
        ++*pos;
    }
    *pos=-1;
    return NULL;
}
```

```
Item removeListItem(List list, Item item){
    struct node *prev, *p;
    Item i;
```

```
if(isEmpty(list)==1){
    fprintf(stderr,"Lista vuota");
    return NULL;
}
```

```
for (p=list->head; p != NULL; prev = p, p = p->next){
    if(cmpItem(item,p->item) == 0) {
        if(p == list->head)
            return removeHead(list);
        else {
            prev->next = p->next;
            i = p->item;
            free(p);
            list->size--;
            return i;
        }
    }
}
```

```
return NULL;
}
```

Item removeListItemPos(List list, int pos){

```
struct node *prev, *p;
```

```
Item i;
```

```
int j;
```

```
if(isEmpty(list)==1){
    fprintf(stderr,"Lista vuota");
    return NULL;
}
```

```
j = 0;
```

```
for (p=list->head; p != NULL; prev = p, p = p->next, j++){
    if(j==pos) {
        if(pos == 0)
            return removeHead(list);
        else {
            prev->next = p->next;
            i = p->item;
            free(p);
            list->size--;
            return i;
        }
    }
}
return NULL;
}
```

int addListItem(List list, Item item, int pos){

```
if (pos == 0){
    addHead(list, item);
    return 1;
}
```

```
if (pos > sizeList(list))
    return 0;
struct node *p;
int i;
```

```
for (p = list -> head, i = 0; p; i++, p = p -> next){
    if (i == pos-1){
        struct node *new = malloc(sizeof(struct node));
        new -> next = p -> next;
        p -> next = new;
        new -> item = item;
        list -> size++;
        return 1;
    }
}
return 0;
}
```

int addListTail(List list, Item item){

```
return addListItem(list, item, sizeList(list));
}
```

void reverseList(List list){

```
struct node *prev=NULL, *p, *temp;
```

```
for(p=list->head; p; prev=p, p=temp){
    temp= p->next;
    p->next=prev;
}
list->head=prev;
}
```

List cloneList(List list){

```
List clone= newList();
```

```
struct node *p;
```

```
for (p = list -> head; p; p = p -> next){
    Item item=cloneItem(p->item);
    addListTail(clone,item);
}
```

```
return clone;
}
```

L05.1. ADT QUEUE

Una **queue** (coda) è una sequenza di elementi di un determinato tipo, in cui gli elementi si aggiungono da un lato (**tail**) e si tolgono dall'altro (**head**).

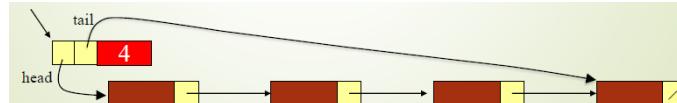
La sequenza viene gestita con la **modalità FIFO** (First-in-first-out), ovvero il primo elemento inserito nella sequenza sarà il primo ad essere eliminato.

La coda è una struttura dati lineare a dimensione variabile, si può accedere direttamente solo alla testa (**head**) della lista e non è possibile accedere ad un elemento diverso da head, se non dopo aver eliminato tutti gli elementi che lo precedono (cioè quelli inseriti prima).

Tra le possibili implementazioni, le più usate sono realizzate tramite **Lista concatenata** e **Array**, concentriamoci sulla prima.

È possibile utilizzare gli operatori di **rimozione dalla testa** e **aggiunta in coda** definite nella Lista concatenata.

Per motivi di efficienza, conviene avere accesso sia al primo elemento sia all'ultimo ed occorre modificare il tipo lista come un puntatore ad una struct che contiene un intero **numelem** che indica il numero di elementi della coda, un puntatore **head** ad uno **struct node** e un puntatore **tail** ad uno **struct node**.



Dobbiamo innanzitutto aggiungere il puntatore tail, poi bisogna modificare gli operatori principali:

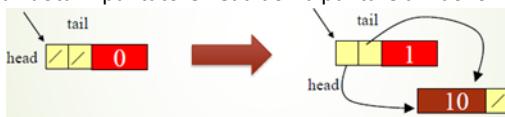
- **removeHead**, deve eventualmente aggiornare entrambi i puntatori head e tail.

Bisogna prima salvare il puntatore al nodo da eliminare (quello puntato da head), head dovrà quindi puntare al successivo. A questo punto si può deallocare la memoria del nodo da rimuovere, *se la coda aveva un solo elemento, ora è vuota, per cui bisogna porre anche il puntatore tail a NULL;*

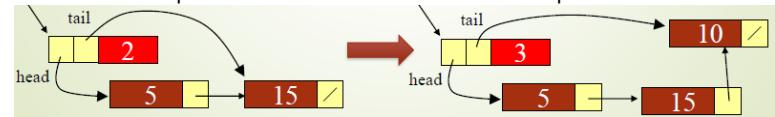
- **addListTail**, grazie alla presenza del puntatore tail, non deve più scorrere gli elementi della lista fino all'ultimo e deve eventualmente aggiornare entrambi i puntatori head e tail.

Dobbiamo innanzitutto creare un nuovo nodo a cui dovrà puntare il puntatore tail, poi bisogna distinguere il caso in cui la coda di input è vuota e il caso in cui non è vuota.

Coda vuota: il puntatore head dovrà puntare al nuovo nodo:



Coda non vuota: il puntatore next dell'ultimo nodo dovrà puntare a nuovo:



Osserviamo adesso l'implementazione della Queue tramite **Array**.

La coda è implementata come un puntatore ad una **struct queue** che contiene tre elementi, ovvero un array di **MAXQUEUE** elementi, un intero che indica la posizione **head** della coda e un intero che indica la posizione **tail** della coda. Quando la coda si riempie, non è possibile eseguire enqueue.



Ma con questa implementazione sorgono alcuni *problem*i.

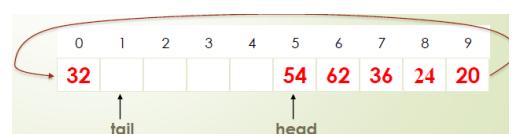
Se l'array viene gestito normalmente, cioè mantenendo **head <= tail**, ci sono dei problemi:



Se rimuoviamo uno alla volta i primi tre elementi in coda otteniamo:



Supponiamo di voler inserire 20 e 32 in coda:



Adesso **tail < head**, perché la posizione 0 segue la posizione N-1. In questo ordine circolare il successore di p è (p + 1) % N, ogni volta che si inserisce un elemento tail avanza: $\text{tail} = (\text{tail} + 1) \% \text{N}$ e ogni volta che si rimuove un elemento head avanza: $\text{head} = (\text{head} + 1) \% \text{N}$.

La coda è piena (non si può usare enqueue) se il successore di tail in questo ordine circolare è head, $(\text{tail} + 1) \% \text{N} == \text{head}$, **la condizione comporta una locazione vuota necessaria a distinguere la condizione di buffer vuoto da quella di pieno**. Quando la coda è vuota, head e tail coincidono.

Sintattica	Semantica
Nome del tipo: Queue Tipi usati: Item, boolean	Dominio: insieme di sequenze $S = a_1, \dots, a_n$ di tipo Item L'elemento nil rappresenta la coda vuota
newQueue() → Queue	newQueue() → q • Post: q = nil
isEmptyQueue(Queue) → boolean	isEmptyQueue(s) → b • Post: se q=nil allora b = true altrimenti b = false
enqueue(Queue, Item) → Queue	enqueue(q, e) → q' • Post: q = <a1, ..., an> AND q' = <a1, ..., an, e>
dequeue(Queue) → Queue	dequeue(q) → q' • Pre: q = <a1, a2, ..., an> n>0 • Post: q' = <a2, ..., an>

item.h

```
typedef void* Item;

Item inputItem();
void outputItem(Item);
int cmpItem(Item,Item);
Item cloneItem(Item);
void libera(Item const);
```

coda.c

```
#include <stdlib.h>
#include "coda.h"

struct coda {
    int size;
    struct node *head;
    struct node *tail;
};

struct node {
    Item item;
    struct node *next;
};

Coda newQueue() {
    Coda list = malloc(sizeof(struct coda));
    list->size = 0;
    list->head = NULL;
    list->tail = NULL;
    return list;
}

bool isEmpty(Coda list) {
    if(list != NULL) {
        return list->size == 0;
    } else
        return false;
}

bool enqueue(Coda list, Item const item) {
    if(list != NULL) {
        struct node *nodo = malloc(sizeof(struct node));
        nodo->next = NULL;
        nodo->item = cloneItem(item);
        if(list->tail != NULL)
            list->tail->next = nodo;
        list->tail = nodo;
        if(list->head == NULL)
            list->head = nodo;
        list->size++;
        return true;
    }
    return false;
}

Item dequeue(Coda list) {
    if((list != NULL) && !isEmpty(list)) {
        Item item = list->head->item;
        struct node *tmp = list->head;
        list->head = tmp->next;
        free(tmp);
        list->size--;
        return item;
    } else {
        return NULL;
    }
}

void freeQueue(Coda list) {
    if(list != NULL) {
        while(!isEmpty(list)) {
            free(dequeue(list));
        }
        free(list);
    }
}
```

coda.h

```
#include <stdbool.h>
#include "item.h"

typedef struct coda *Coda;

Coda newQueue();
bool isEmpty(Coda);
bool enqueue(Coda, Item const);
Item dequeue(Coda);
void freeQueue(Coda);
```

L05.2. ADT STACK

Una **pila** è una sequenza di elementi di un determinato tipo, in cui è possibile aggiungere o togliere elementi esclusivamente da un unico lato (**top dello stack**). La pila è una struttura dati lineare a dimensione variabile in cui si può accedere direttamente solo al primo elemento della lista, non è possibile accedere ad un elemento diverso dal primo se non dopo aver eliminato tutti gli elementi che lo precedono (inseriti dopo) ed è una Lista gestita con la **modalità LIFO (Last-in-first-out)** cioè l'ultimo elemento inserito nella sequenza sarà il primo ad essere eliminato.

Tra le possibili implementazioni, le più usate sono realizzate tramite **Array** e **Lista concatenata**, concentriamoci sulla prima.

Lo stack è implementato come un puntatore ad una **struct stack** che contiene due elementi, un array di **MAXSTACK** elementi e un intero che indica la posizione del top dello stack. Quando lo stack si riempie, non è possibile eseguire l'operazione push.

Osserviamo adesso l'implementazione dello Stack tramite **Lista concatenata**.

Il tipo stack è definito come un puntatore ad una struct che contiene un elemento **item** di tipo **list**, non serve più nemmeno l'intero **MAXSTACK** che indica la capienza massima dello stack, anche se abbiamo un solo elemento nella struct, continuiamo a definire il tipo stack come puntatore a **struct stack** per non cambiare la definizione nell'header file.

Sintattica	Semantica
Nome del tipo: Stack Tipi usati: Item, boolean	Dominio: insieme di sequenze $S = a_1, \dots, a_n$ di tipo Item L'elemento nil rappresenta la pila vuota
newStack() → Stack	newStack() → s • Post: s = nil
isEmptyStack(Stack) → boolean	isEmpty(s) → b • Post: se s=nil allora b = true altrimenti b = false
push(Stack, Item) → Stack	push(s, e) → s' • Post: s = <a1, a2, ..., an> AND s' = <e, a1, ..., an>
pop(Stack) → Stack	pop(s) → s' • Pre: s = <a1, a2, ..., an> n>0 • Post: s' = <a2, ..., an>
top(Stack) → Item	top(s) → e • Pre: s = <a1, a2, ..., an> n>0 • Post: e = a1

item.h

```
#include <stdbool.h>
typedef void* Item;

Item inputItem();
void outputItem(Item const);
bool cmpItem(Item const, Item const);
Item cloneItem( Item const);
void libera(Item const);
```

pila.h

```
#include "item.h"

typedef struct pila *Pila;

Pila newStack();
bool isEmpty(Pila);
bool push(Pila, Item const);
Item top(Pila);
void pop(Pila);
void freeStack(Pila);
```

pila.c

```
#include <stdlib.h>
#include "pila.h"
#define max_elem 5
#define add_elem 5

struct pila {
    int dim;
    int size;
    Item *items;
};

Pila newStack() {
    Pila pila = malloc(sizeof(struct pila));
    pila->dim = 0;
    pila->size = max_elem;
    pila->items = malloc(max_elem * sizeof(Item));
    return pila;
}

bool isEmpty(Pila pila) {
    if(pila != NULL) {
        return pila->dim == 0;
    }
    return true;
}

bool push(Pila pila, Item const item) {
    if((pila != NULL) && (item != NULL)) {
        if(pila->dim >= pila->size) {
            pila->size += add_elem;
            pila->items = realloc(pila->items, pila->size * sizeof(item));
        }
        pila->items[pila->dim] = cloneItem(item);
        pila->dim++;
        return true;
    }
    return false;
}

Item top(Pila pila) {
    if((pila != NULL) && (!isEmpty(pila))) {
        return cloneItem(pila->items[pila->dim-1]);
    }
    return NULL;
}

void pop(Pila pila) {
    if((pila != NULL) && (!isEmpty(pila))) {
        pila->dim--;
        free(pila->items[pila->dim]);
        if((pila->size - pila->dim) > add_elem) {
            pila->size -= add_elem;
            pila->items = realloc(pila->items, pila->size * sizeof(item));
        }
    }
}

void freeStack(Pila pila) {
    if(pila != NULL) {
        while(!isEmpty(pila)) {
            pop(pila);
        }
        free(pila);
    }
}
```

L06. RICORSIONE

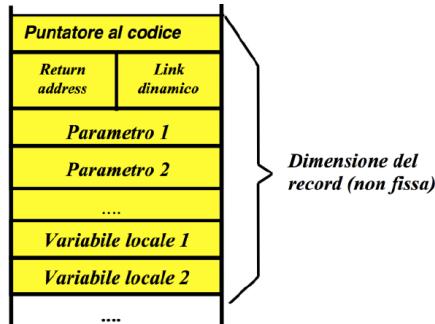
Tecnica usata in programmazione per la quale un sotto-programma chiama se stesso.

RECORD DI ATTIVAZIONE:

Il sistema operativo gestisce la ricorsione tramite una zona di memoria chiamata Stack, in cui vengono inseriti all'interno di questo Stack dei **record di attivazione**, quest'ultima è una struttura dati creata dinamicamente ogni volta che viene invocata una funzione, in particolare, ciò che avviene nel nostro sistema è che:

1. si crea una nuova **attivazione** (o istanza) della funzione chiamata;
2. viene **allocata la memoria** per i parametri e le variabili locali;
3. si effettua il **passaggio dei parametri**;
4. si **trasferisce il controllo** alla funzione chiamata;
5. si **esegue il codice** della funzione.

Struttura record di attivazione:



Questi sono i campi che troviamo all'interno del record di attivazione, in particolare, troviamo:

- i **parametri formali**, cioè la copia dei parametri che vengono passati alla nostra funzione;
- le **variabili locali**;
- degli indirizzi che servono a gestire il flusso di istruzioni del programma in particolare abbiamo un **indirizzo di ritorno** che indica il punto a cui tornare (nel codice del chiamante) al termine della funzione;
- un collegamento al record di attivazione del chiamante, detto **link dinamico**;
- l'**indirizzo del codice** della funzione, cioè un puntatore alla prima istruzione del corpo di questa funzione, cioè quella a cui è riferito questo record di attivazione.

Per quanto riguarda il **ciclo di vita** di un record di attivazione, cioè quando viene creato e quando viene distrutto.

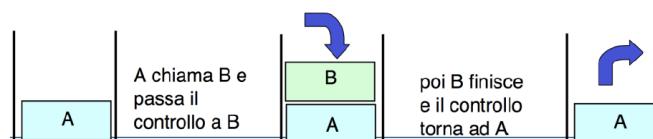
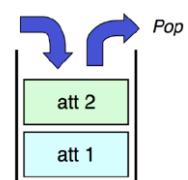
Un nuovo record di attivazione viene creato nel momento in cui si invoca una funzione f e permane per tutto il tempo in cui questa funzione è in esecuzione e viene invece distrutto al termine dell'esecuzione della funzione.

È chiaro che per ogni invocazione di una stessa funzione ci sarà un nuovo record di attivazione, quindi il record di attivazione non è associato alla funzione ma associato alla sua esecuzione, quindi se eseguiremo all'interno di un programma n volte una funzione f avremo n volte n record di attivazioni corrispondenti a quella funzione.

NOTA: funzioni diverse potrebbero avere una dimensione del record di attivazione differente, invece, la stessa funzione ogni volta che viene invocata, ha sempre la stessa dimensione del record e si può conoscere a priori.

Il sistema operativo, per l'esecuzione di un programma, mantiene un'area di memoria, in cui vengono allocati record di attivazione, che viene gestita come una lista **LIFO** e corrisponde esattamente ad uno **Stack**, ogni elemento di questo Stack è appunto un record di attivazione.

Come già visto, lo Stack supporta operazioni di **push** e **pop**, la prima ci consente di aggiungere un elemento in cima allo Stack e la seconda invece di prelevarlo ed eliminarlo dalla cima dello Stack, anche questa struttura del sistema operativo che mantiene i record di attivazione, funziona esattamente in questo modo.



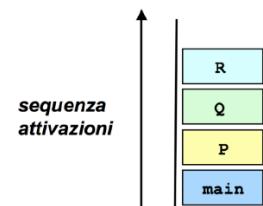
Supponiamo di avere una funzione A che chiama una funzione B, nello Stack abbiamo soltanto il record di attivazione della funzione A, poi questa chiama B e passa il controllo a B, quindi quello che viene fatto è una Push all'interno dello Stack in cui viene inserito il record di attivazione corrispondente all'esecuzione della funzione B, poi B finisce il controllo e quindi il controllo torna ad A e quello che abbiamo è una pop di B dallo Stack, quindi nel nostro Stack sarà scomparso perché viene distrutto il record di attivazione corrispondente a B.

Supponiamo di avere 4 funzioni che si chiamano tra loro, in particolare:

- `int R(int a) { return a+1; }`
- `int Q(int x) { return R(x); }`
- `int P(void) { int a=10; return Q(a); }`
- `main() { int x = P(); }`

Avremo una sequenza di chiamate che effettuerà il sistema operativo:

S.O. → main → P() → Q() → R()



RICORSIONE:

Fatta quindi questa premessa su come il sistema operativo gestisce le chiamate a funzione tramite questa struttura, vediamo adesso che cos'è la ricorsione e che cosa comporta una chiamata ad una funzione ricorsiva o un sottoprogramma ricorsivo.

Un **sottoprogramma ricorsivo** è un sottoprogramma che direttamente o indirettamente, cioè tramite un'altra funzione, chiama sé stesso. I linguaggi che possono gestire la ricorsione lo fanno mediante la gestione dei **record di attivazione**.

Prima di scrivere una funzione ricorsiva dobbiamo pensare ad un approccio che risolva ricorsivamente un problema, questo approccio prevede:

- L'**identificazione di un caso base**, in cui c'è una soluzione nota;
- Esprimere la soluzione al **caso generico n in termini dello stesso problema** in uno o più casi più semplici, per casi più semplici si intende la risoluzione dello stesso problema ma con una taglia inferiore dell'input, quindi se la funzione risolve il problema per un generico caso n le funzioni che verranno chiamate ricorsivamente risolvono il problema per $n-1$, $n-2$ o cose simili.

Molte **funzioni matematiche** sono definite ricorsivamente quando nella sua definizione compare un riferimento a sé stessa.

La dimostrazione che queste funzioni risolvono il problema è basata sul principio di **induzione matematica**:

- se una proprietà P vale per $n=n_0$ (**CASO BASE**);
- si può provare che, **assumendola valida per n** , allora vale per $n+1$ (allora P vale per ogni $n \geq n_0$).

La funzione fattoriale(n), denotato come $n!$, è definito per tutti gli interi $n \geq 0$ come:

- $n! = 1$ se $n=0$;
- $n! = n*(n-1)!$ Se $n>0$.

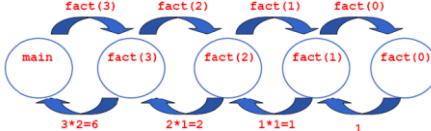
Data una funzione ricorsiva, questa fa sia da **Servitore** (funzione chiamata) che **Cliente** (funzione chiamante):

```
Int fact(int n){
    if(n==0) return 1;
    else return n*fact(n-1);
}

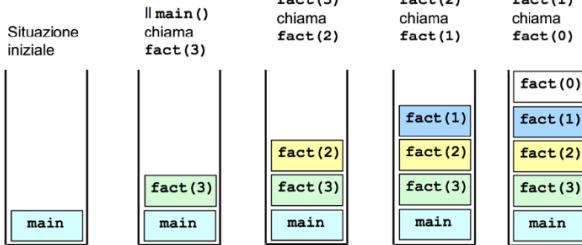
main() {
    int fz, z=5;
    fz=fact(z-2);
}
```

1. Il main chiama fact(3);
2. fact(3) chiama fact(2);
3. fact(2) chiama fact(1);
4. fact(1) chiama fact(0);
5. fact(0) restituisce 1;
6. fact(1) restituisce 1;
7. fact(2) restituisce 2;
8. fact(3) restituisce 6;
9. il main riceve 6 da fact(3) e lo assegna alla variabile fz.

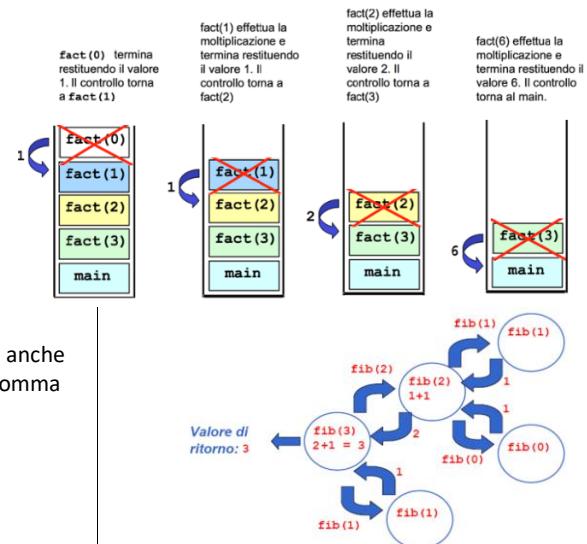
La si può rappresentare anche:



Nello Stack di sistema avviene la seguente situazione:



main	$\text{fact}(3) = 3 * \text{fact}(2) = 2 * \text{fact}(1) = 1 * \text{fact}(0)$
Cliente di fact(3)	Cliente di fact(2)
Servitore del main	Servitore di fact(3)
Servitore di fact(2)	Servitore di fact(1)



Un altro esempio famoso di funzione ricorsiva è la **successione di Fibonacci**, detta anche successione Aurea, è una successione di interi positivi in cui ciascun numero è la somma dei due precedenti, i primi due invece sono per definizione pari a 1.

Definizione ricorsiva di questa successione:

- La base è costituita da due relazioni: $F_0 = 1$ ed $F_1 = 1$;
- L'ennesimo numero di Fibonacci: $F_n = F_{n-1} + F_{n-2}$ per ogni $n > 1$.

Negli esempi visti finora, si inizia a sintetizzare il risultato «a ritroso», solo dopo che le chiamate si sono chiuse.

Il risultato viene sintetizzato a partire dalla fine, perché occorre prima arrivare al caso «banale»:

- Il caso banale fornisce il valore di partenza;
- Poi si sintetizzano a ritroso i successivi risultati parziali.

ITERAZIONE VS RICORSIONE:

Vediamo le caratteristiche principali che distinguono la ricorsione dall'iterazione. Prendiamo in considerazione un una funzione che calcola il fattoriale in modo iterativo:

```
int fact(int n) {
    int i=1;
    int F=1; /*inizializzazione del fattoriale*/
    while (i <= n)
        { F=F*i;
          i=i+1; }
    return F;
}
```

DIFERENZA CON LA VERSIONE RICORSIVA: ad ogni passo viene accumulato un risultato intermedio

La variabile F accumula risultati intermedi: se $n = 3$ inizialmente $F=1$, poi al primo ciclo $F=1$, poi al secondo ciclo F assume il valore 2. I ne all'ultimo ciclo $i=3$ e F assume il valore 6

- Al primo passo F accumula il fattoriale di 1
- Al secondo passo F accumula il fattoriale di 2
- Al passo i -esimo F accumula il fattoriale di i

La caratteristica principale dell'iterazione è quella che per calcolare un risultato, i risultati intermedi vengono portati in avanti, possiamo dire che il risultato viene sintetizzato in avanti e ad ogni passo è disponibile quindi un risultato parziale, dopo K passi si ha a disposizione un risultato parziale relativo al caso K.

Questo invece non avviene nei processi computazionali ricorsivi in cui non abbiamo alcun risultato finché non giungiamo al caso elementare, cioè alla base della ricorsione.

In realtà è possibile simulare un processo computazionale iterativo attraverso funzioni ricorsive, in questo caso per simulare l'iterazione utilizziamo una variabile che chiamiamo **accumulatore** e praticamente questa variabile è destinata ad esprimere in ogni istante la soluzione corrente e viene passato come parametro ad ogni chiamata della funzione. Questo tipo di ricorsione che simula un processo computazionale iterativo è detta **ricorsione Tail**.

RICORSIONE TAIL:
Questa ricorsione, che realizza un processo computazionale iterativo, è detta anche “**ricorsione apparente**”, perché è una ricorsione che in realtà simula un’iterazione.

La chiamata ricorsiva, in queste funzioni che utilizzano la ricorsione apparente, è sempre l'ultima istruzione, questa chiamata serve soltanto per proseguire la computazione, i calcoli invece vengono fatti prima, da questo il nome **ricorsione Tail**.

È possibile **trasformare** una funzione iterativa in una funzione ricorsiva che effettua la stessa computazione. Prendiamo il ciclo iterativo e lo trasformiamo in un if con la stessa condizione, il corpo del ciclo rimane invariato, quindi copiamo ciò che abbiamo, ricordandoci però che come ultima istruzione dell'if metteremo la chiamata ricorsiva per far continuare la computazione.

È chiaro che la funzione ricorsiva dovrà ospitare dei nuovi parametri per portare avanti le variabili di stato.

<code>while (condizione) { <corpo del ciclo> }</code>	<code>if (condizione) { <corpo del ciclo> <chiamata ricorsiva> }</code>
---	---

Vediamo la funzione fattoriale nella versione ricorsione Tail:

```

int fact(int n) {
    return factIt(n, 1, 1);
}

int factIt(int n, int F, int i){
    if (i <= n)
        {F = i*F;
        i = i+1;
        return factIt(n,F,i);
    }
    return F; Accumulatore del risultato parziale
}

```

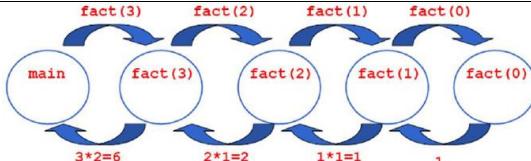
Inizializzazione dell'accumulatore: corrisponde al fattoriale di 1
Contatore del passo

Al posto del l'iterazione che avevamo prima con while mettiamo una condizione con un if, la condizione all'interno rimane la stessa e anche le prime due istruzioni che avevamo nel while rimangono le stesse, quindi abbiamo la moltiplicazione di $i*F$ riassegnata ad F e l'incremento della variabile i e poi infine aggiungiamo come ultima istruzione all'interno del blocco del if la chiamata ricorsiva.

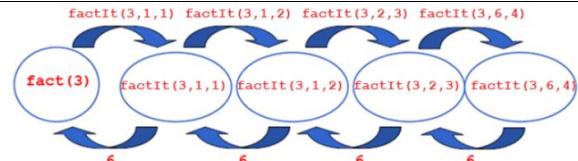
La funzione non ha più un solo parametro, cioè N (numero del quale intendiamo calcolare il fattoriale), ma abbiamo aggiunto due parametri che portiamo avanti per continuare la computazione, F che è l'accumulatore (variabile che conserva il risultato parziale) e la i che è il contatore del passo che ci dice in quale passo siamo arrivati.

Chiamiamo questa funzione $factIt$ all'interno di una funzione $fact$ che prende soltanto n come parametro, questa funzione serve per nascondere la chiamata alla ricorsione Tail e questa prima chiamata che facciamo prendere i parametri inizializzati correttamente.

RICORSIONE CLASSICA



RICORSIONE TAIL



Quindi questa soluzione ricorsiva per il fattoriale ha tutte le caratteristiche di una funzione ricorsiva poiché esegue una computazione ricorsiva di fatto, ma in realtà dà luogo ad un processo computazionale iterativo per questo motivo la chiamiamo ricorsione apparente o ricorsione Tail.

La differenza principale con la ricorsione classica è questa caratteristica che il risultato viene sintetizzato in avanti, quindi il calcolo avviene nel corpo del if e poi viene portato avanti. Arrivati alla fine non si fa altro che riportare indietro fino al chiamante il risultato già ottenuto.

STRUTTURE RICORSIVE:

Alcune strutture dati sono inherentemente ricorsive, questo vale per molte delle sequenze e delle strutture ad albero. Su questo tipo di struttura la formulazione ricorsiva di algoritmi risulta essere molto naturale.

Ad esempio, una lista può essere definita come una lista vuota oppure un elemento seguito da una lista, il primo caso somiglia alla base di una ricorsione, mentre il secondo invece somiglia ad un passo ricorsivo, quindi a tutti gli effetti una lista è una struttura ricorsiva si può fare lo stesso ragionamento anche per gli alberi. Possiamo implementare alcuni algoritmi per la **lista linkata** in modo ricorsivo:

- **Stampa lista:** stampo l'elemento corrente p e chiamo ricorsivamente la funzione di stampa sulla lista puntata da $p->next$;
- **Ricerca:** verifico se il dato cercato è presente nell'elemento corrente (in caso affermativo restituisco l'elemento) e chiamo ricorsivamente la funzione di ricerca sulla lista puntata da $p->next$;
- **Numero di occorrenze di un item:** verifico se il dato cercato è presente nell'elemento corrente (in caso affermativo incremento il contatore) e chiamo ricorsivamente la funzione di conteggio sulla lista puntata da $p->next$;
- **Deallocazione degli elementi:** chiamo ricorsivamente la funzione di deallocazione sulla lista puntata da $p->next$ e libero la memoria corrispondente all'elemento corrente p .

Le differenze tra la ricorsione e l'iterazione, entrambe effettuano una computazione che ripete le stesse istruzioni, nell'iterazione questa viene fatta tramite la dichiarazione di un ciclo, nella ricorsione invece la computazione viene ripetuta attraverso chiamate alla stessa funzione. Per terminare la computazione l'iterazione utilizza la condizione all'interno del ciclo iterativo che fallisce quando si termina, invece nella ricorsione si arriva al caso base e poi si va a ritroso. Entrambe sia la ricorsione che l'iterazione, possono dar luogo a cicli infiniti quindi andare in loop.

Una prima caratteristica a favore dell'iterazione è quella relativa alla performance, infatti la ricorsione richiede un notevole overhead o sovraccarico al tempo di esecuzione dovuto alla gestione dello Stack, ogni chiamata a funzione presuppone la creazione di un record di attivazione e dell'inserimento di questo all'interno dello Stack, di contro invece la ricorsione è una buona pratica di Ingegneria del software perché risulta spesso essere più chiara come lettura del codice.

Consigliamo quindi l'utilizzo di algoritmi ricorsivi quando dobbiamo implementare delle funzioni che sono effettivamente ricorsive, abbiamo ad esempio visto fattoriale e Fibonacci che per la loro stessa definizione conviene che vengano implementate ricorsivamente, abbiamo anche detto che molte strutture presentano una natura inherentemente ricorsiva quindi su queste strutture, come strutture ad albero o sequenze, spesso la formulazione ricorsiva di algoritmi risulta essere più naturale, la ricorsione invece andrebbe evitata quando la soluzione iterativa è abbastanza ovvia e quando le prestazioni sono un elemento critico del nostro programma.

```

void recursivePrintList(struct node *p){
    if(p != NULL) {
        outputItem(p->item);
        recursivePrintList(p->next);
    }
}

void printList(List list){
    recursivePrintList(list->head);
    printf("\n");
}

```

```

struct node * minimo (struct node *p){
    struct node *i, *min = p;
    for (i = p; i != NULL; i = i->next){
        if ((cmpItem(min->item, i->item)) > 0)
            min = i;
    }
    return min;
}

void recursiveSort(struct node *p){
    if(p != NULL) {
        struct node *pos_minimo = minimo(p);
        swap(&(pos_minimo->item), &(p->item));
        recursiveSort(p->next);
    }
}

void sortList(List list){
    recursiveSort(list->head);
}

```

```

Item recursiveSearchList(struct node *p, Item item, int *pos){

    if(p != NULL) {
        if(cmpItem(item,p->item) == 0)
            return cloneItem(p->item);
        else {
            ++*pos;
            return recursiveSearchList(p->next, item, pos);
        }
    } else {
        return NULL;
    }
}

Item searchList(List list, Item item, int *pos){
    *pos=0;
    struct node *result = recursiveSearchList(list->head, item, pos);
    if(result == NULL)
        *pos=-1;
    return result;
}

```

L07. CENNI SULLA COMPLESSITÀ COMPUTAZIONALE

L'analisi della complessità computazionale è la stima del costo degli algoritmi in termini di risorse di calcolo, quando parliamo di risorse di calcolo ci riferiamo al tempo di esecuzione, avvolte ci riferiremo anche allo spazio di memoria.

Esempio:

Dato un vettore di n interi ordinati in maniera non decrescente, verificare se l'intero K è presente o meno nel vettore. Per risolvere il problema utilizziamo questo l'algoritmo:

```
int ricerca(int v[], int size, int k){  
    int i;  
    for (i=0; i<size; i++)  
        if(v[i] == k) return i;  
    return -1;  
}
```

Algoritmo banale che effettua *un'intera visita* del vettore e quando trova l'elemento cercato restituisce la posizione nel vettore altrimenti restituisce -1. Questo algoritmo non è efficiente, dovuta al fatto che l'algoritmo non ha nessun vantaggio dal fatto che il vettore è ordinato, ma sappiamo invece che esiste un algoritmo migliore che è quello della ricerca binaria su un Array ordinato.

In particolare, vogliamo essere in grado di analizzare e valutare il tempo di esecuzione degli algoritmi, esistono diverse variabili che influenzano il tempo di esecuzione:

- **Macchina usata**, se seguiamo il nostro algoritmo su un supercalcolatore è differente che se lo eseguiamo su un dispositivo mobile;
- **Dimensione dei dati**, ad esempio per il problema dell'ordinamento, ordinare pochi numeri interi è diverso da ordinare i record degli studenti;
- **Configurazione dei dati**, nel problema precedente sapevamo che i dati erano ordinati e quindi potevamo regolarci di conseguenza.

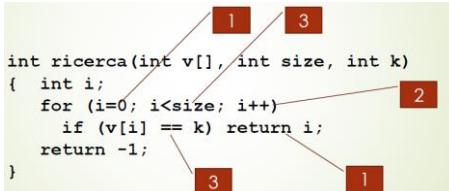
Quello che vogliamo ottenere è un modello astratto per la valutazione del tempo di esecuzione che:

- Sia indipendente dalla macchina usata, così che possa essere applicato in tante situazioni;
- Stimi il tempo in funzione della dimensione dell'input;
- Abbia un **comportamento asintotico**, quindi che prende in considerazione il tempo di esecuzione per taglie dell'input molto grandi perché è proprio per taglie di input grandi che si vede se un algoritmo efficiente oppure no;
- Ci fornisca una stima nel **caso peggiore** della configurazione dei dati, perché anche nel caso peggiore l'algoritmo deve dimostrare la sua efficienza.

Vediamo un esempio di modello di una macchina astratta che conta le istruzioni e le condizioni atomiche che sono presenti all'interno dell'algoritmo:

- Istruzioni e condizioni atomiche hanno costo unitario;
- Le strutture di controllo hanno un costo pari alla somma dei costi dell'esecuzione delle istruzioni interne, più la somma dei costi delle condizioni;
- Le chiamate a funzione hanno un costo pari al costo di tutte le istruzioni e condizione in esse contenute, non applichiamo alcun costo per il passaggio dei parametri;
- Istruzioni e condizioni con chiamate a funzioni hanno costo pari alla somma del costo delle funzioni invocate più uno.

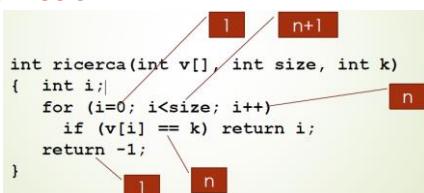
Calcolare il costo per l'esempio precedente nel caso $v[n] = \{1, 3, 9, 17, 34, 95, 96, 101\}$ e $k=9$.



Contiamo quante istruzioni o condizioni.

La prima inizializzazione della variabile verrà fatta una volta sola, il controllo della condizione verrà fatto tre volte, l'incremento verrà fatto due volte, il controllo della condizione del if verrà fatto tre volte, una volta verrà fatto il Return, in totale abbiamo 10 tra istruzioni e condizioni.

CASO PEGGIORI:



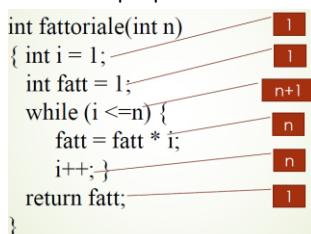
Il caso peggiore è quando l'elemento che cerchiamo non è presente. Quindi saremo costretti a scorrere l'intero vettore, facciamo di nuovo il conteggio dei passi necessari: l'inizializzazione viene fatta sempre una volta sola, il controllo verrà fatto $n + 1$ volte, incremento della variabile verrà fatto n volte, il controllo del if verrà fatto n volte, il return alla fine verrà fatto una volta sola, quindi in totale avremo **$3n + 3$ operazioni**.

CASO MEDIO:

Spesso siamo interessati anche a calcolare il caso medio. Tornando al nostro esempio di ricerca all'interno di un Array ordinato, il caso medio è quello in cui il numero cercato è presente e tutte le posizioni abbiano la stessa probabilità di contenere questo numero, quindi per N numeri la probabilità che il numero da cercare K sia in una di questa posizione, cioè nella posizione i è $\frac{1}{N}$.

DIMENSIONE DELL'INPUT:

Abbiamo detto che uno dei requisiti del modello è quello di calcolare il costo computazionale come funzione della dimensione dell'input. Bisogna intendersi su cosa sia la dimensione dell'input, in un vettore ci siamo riferiti alla dimensione dell'input come il numero di elementi contenuti nel vettore. Un ragionamento analogo si può fare per strutture sequenziali come le liste oppure come gli Stack e le code, nel caso di un albero potremmo andare a contare il numero di nodi, trovandoci invece di fronte ad un grafo la dimensione dell'input potrebbe essere data dal numero dei nodi più il numero degli archi. Nel caso invece del fattorialeabbiamo che l'input è un numero intero non limitato, chiaramente quanto più grande è questo numero tanti più passi dovremmo andare a fare, vediamo quanti passi effettuiamo per il calcolo del fattoriale:



Le prime due inizializzazioni vengono fatte una volta sola, il controllo del while viene fatto $n + 1$ volte, la moltiplicazione all'interno viene fatta n volte, l'incremento pure viene fatto n volte, l'operazione di ritorno viene fatto una volta sola, quindi abbiamo un totale di **$3n + 4$ operazioni**.

Quindi considerando il parametro n intero avremo che la nostra funzione fattoriale richiede un numero di operazioni che è lineare rispetto ad n , infatti $3n + 4$ è una funzione lineare. Se invece utilizzassimo un modo diverso per rappresentare n (l'input) potremmo considerarlo come il numero d di bit necessari per rappresentare il numero, in questo caso d sarebbe circa uguale al $\log_2 n$, quindi il costo diventerebbe $3x2^d+4$ che è una funzione esponenziale, quindi chiaramente il modo in cui sceglieremo di rappresentare l'input ha un effetto anche sul costo dell'algoritmo.

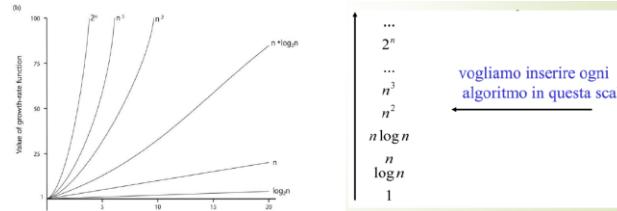
COMPORTAMENTO ASINTOTICO:

Un'altra caratteristica importante dei modelli è quello di dover avere un'analisi del comportamento asintotico, cioè il risultato che più ci interessa è quello che si ottiene per n (input) molto grandi, questo è dovuto al fatto che per valori piccoli di n il tempo richiesto è comunque basso, quindi qualunque algoritmo va bene, invece giudicare un algoritmo comporta dare un giudizio sull'efficienza dell'algoritmo per valori grandi di n , questo tipo di analisi viene detta **analisi del comportamento asintotico**.

Se ci concentriamo sul comportamento di un algoritmo al crescere della dimensione n dei dati all'infinito, allora quello a cui arriveremo è una situazione in cui classificheremo i nostri algoritmi in classi di complessità, quindi arriveremo a trascurare tutte le costanti moltiplicative ed additive, e invece faremo riferimento soltanto alla funzione matematica che mette in relazione il tempo di esecuzione con la taglia n dell'input.

Le classi di complessità più frequenti sono:

▪ a	costante
▪ $a n+b$	lineare
▪ $a n^2 + bn + c$	quadratica
▪ $a \log n + h$	logaritmica
▪ a^n	esponenziale
▪ n^n	esponenziale



Facciamo delle considerazioni, una prima cosa è che per piccole dimensioni dell'input tutti gli algoritmi hanno tempi di risposta non significativamente differenti, questo è il motivo per il quale le analisi si fanno sempre per n molto grandi quindi in tal modo possiamo analizzare il comportamento asintotico.

NOTAZIONE O E Ω:

Utilizziamo un modello che è molto famoso nell'analisi degli algoritmi che si chiama **notazione asintotica** che utilizza delle funzioni **Omicron** e **Omega**.

Consideriamo due funzioni f e g definite sui numeri naturali e con valori nel campo reale diremo che:

- $f(n)$ è **O** di $g(n)$, oppure $f(n) \in O(g(n))$, se esistono due costanti positive c ed n_0 tali che se $n \geq n_0$, $f(n) \leq c g(n)$.

Applicata alla funzione di complessità $f(n)$, la notazione **O** ne limita superiormente la crescita e fornisce quindi una indicazione della bontà dell'algoritmo.

- $f(n)$ è **Omega** di $g(n)$, $f(n) \in \Omega(g(n))$, se esistono due costanti positive c ed n_0 tali che se $n \geq n_0$, $c g(n) \leq f(n)$.

La notazione **Ω** limita inferiormente la complessità, indicando così che il comportamento dell'algoritmo non è migliore di un comportamento assegnato.

REGOLE PER LA VALUTAZIONE DELLA COMPLESSITÀ:

Una **prima regola** è la **scomposizione**, supponiamo che:

- alg è la sequenza di alg1 ed alg2;
- alg1 è $O(g1(n))$;
- alg2 è $O(g2(n))$;
- alg è $O(\max(g1(n), g2(n)))$.

A prevalere sarà quello che ha la complessità più grande.

Una **seconda regola** è quella dei **blocchi annidati**, supponiamo che:

- alg è composto da due blocchi annidati;
- blocco esterno è $O(g1(n))$;
- blocco interno è $O(g2(n))$;
- alg è $O(g1(n)*g2(n))$.

A prevalere sarà la moltiplicazione delle due complessità.

Esempio calcolo complessità:

Consideriamo l'algoritmo per il prodotto di due matrici, ricordiamo che per poter applicare il prodotto di due matrici è necessario che il numero di colonne della prima matrice sia pari al numero di righe della seconda, in questo esempio abbiamo quindi una matrice A con N righe ed M colonne di cui vogliamo fare il prodotto con una matrice B con M righe e P colonne, il risultato sarà una matrice C con N righe e P colonne. Per poter effettuare il prodotto dobbiamo calcolare il risultato che metteremo in ciascuna cella della matrice, in particolare ciascuna cella conterrà una somma di prodotti.

Abbiamo bisogno di tre cicli iterativi un ciclo più esterno che scorre le righe della prima matrice, un ciclo intermedio che scorre le colonne della seconda matrice e un terzo ciclo più interno che invece calcolerà la somma di prodotti.

Prima della terza iterazione sarà necessario azzerare il contenuto della cella e quindi poi all'interno della terza iterazione andremo a sommare di volta in volta il prodotto ottenuto in questa iterata.

Analizziamo la complessità, abbiamo tre blocchi annidati quindi possiamo applicare la regola in cui per ottenere la complessità di un programma in cui ci sono dei blocchi annidati basta moltiplicare la complessità dei blocchi ed è quello che facciamo anche qui. Vediamo che il blocco più esterno viene fatto un numero di volte che è $O(N)$, il secondo blocco viene fatto un numero di volte che è $O(P)$, il terzo blocco viene eseguito un numero di volte pari a $O(M)$. Quindi in totale dobbiamo moltiplicare questi valori e abbiamo che la complessità asintotica del programma è $O(N \cdot P \cdot M)$.

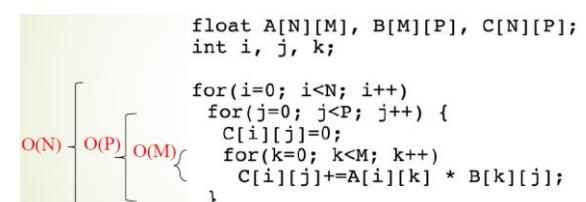
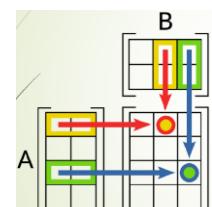
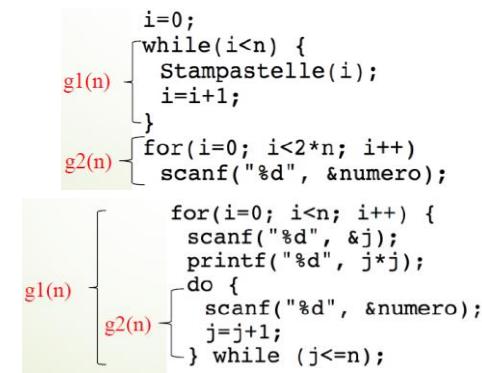
Una **terza regola** è quella di avere dei **sottoprogrammi ripetuti**:

- alg applica ripetutamente un certo insieme di istruzioni la cui complessità all'i-esima esecuzione vale $f_i(n)$;
- il numero di ripetizioni è $g(n)$;
- alg è $O(\sum_{i=1}^{g(n)} f_i(n))$;
- per $f_i(n)$ tutte uguali ... $O(g(n) f(n))$.

Esempio: Insertion Sort.

Una **quarta regola** è l'**operazione dominante**:

- Sia $f(n)$ il costo di esecuzione di un algoritmo alg;
- Un'istruzione i è dominante se viene eseguita $g(n)$ volte, con $f(n) \leq a g(n)$;
- Se un algoritmo ha una operazione dominante allora è $O(g(n))$. *Esempio: ordinamento, il confronto tra due elementi dell'array è dominante.*



COMPLESSITÀ DEI PROBLEMI:

Abbiamo parlato finora di complessità degli algoritmi, un discorso invece un po' più complesso riguarda la complessità dei problemi.

Studiare la complessità di un problema è molto diverso da studiare la complessità di un algoritmo, la relazione tra problema e algoritmo è di uno a molti, cioè un problema può essere risolto da diversi algoritmi, ciascuno dei quali può avere una sua complessità.

Trovare un limite superiore alla complessità del problema è abbastanza semplice, per porre un limite superiore a tempo di esecuzione di un problema basta trovare un algoritmo che lo risolva con quella complessità, se troviamo ad esempio un algoritmo che ha complessità $O(g(n))$ e che risolve il dato problema, allora possiamo dire che quel problema a complessità $O(g(n))$.

Porre invece un limite inferiore al tempo di esecuzione di un problema è molto più difficile, perché una volta affermato che quel problema ha un dato tempo di esecuzione, che può essere $\Omega(g(n))$, vuol dire che non esiste un algoritmo che lo risolve meglio di quella complessità.

Questo potrebbe riuscire abbastanza difficile da dimostrare poiché quell'algoritmo potrebbe esistere ma non è stato ancora trovato, quindi per trovare un limite inferiore dobbiamo attuare delle strategie e fare delle dimostrazioni matematiche che ci consentono di stabilire che quello è il limite inferiore non si può fare di meglio.

Esistono delle **strategie** che ci consentono di individuare i limiti inferiori, una di queste è analizzare la **taglia dei dati**.

Supponiamo che la taglia sia di dimensione n , se sappiamo che l'algoritmo analizza tutto l'input allora chiaramente l'algoritmo sarà $\Omega(n)$, come esempio possiamo citare la ricerca di un elemento o del massimo all'interno di un Array, chiaramente questi algoritmi hanno complessità lineare poiché sono costretti ad analizzare l'intero input, è possibile comunque trovare dei limiti inferiori più stretti, quindi analizzando la taglia dell'input troviamo un limite inferiore ma non è detto che sia un limite inferiore stretto.

Un'altra tecnica per individuare un limite inferiore è quella degli **eventi contabili**.

Se ad esempio sappiamo che per risolvere un problema dobbiamo ripetere un evento un certo numero di volte allora sappiamo che il numero di volte che questo evento viene ripetuto sarà sicuramente un limite inferiore alla complessità del problema, come esempio consideriamo il problema di generare tutte le permutazioni di n oggetti, siamo costretti a fare l'operazione di generazione un numero di volte che è pari a tutte le permutazioni degli oggetti che sappiamo essere $n!$.

Tornando al problema dell'ordinamento sappiamo che l'evento che avviene più spesso è quello del confronto e si può dimostrare matematicamente che tutti gli algoritmi che operano per confronti devono effettuare questa operazione almeno $n \log n$ volte, quindi $\Omega(n \log n)$, è un limite inferiore al problema dell'ordinamento.

Vediamo un esempio in cui applichiamo l'**istruzione dominante**, consideriamo il problema della ricerca binaria che abbiamo già risolto iterativamente:

The diagram shows a code snippet for a binary search algorithm. A red box labeled "Istruzione dominante" points to the line of code: "if (k==v[med])". The code is as follows:

```
int ricerca(int v[], int size, int k)
{ int inf = 0, sup = size-1;
  while (sup >=inf)
  { int med = (sup + inf) / 2;
    if (k==v[med])
      return med;
    else if (k>v[med])
      inf = med+1;
    else sup = med-1
  }
  return -1;
}
```

Nella ricerca binaria abbiamo un'istruzione dominante che è quella della verifica se l'elemento che stiamo cercando è uguale all'elemento corrente, quindi l'operazione dominante sarà un confronto.

Dobbiamo capire quante volte viene eseguito questo confronto, osserviamo che la dimensione del problema si dimezza ad ogni ciclo, inizialmente è n poi abbiamo $n/2$, poi $n/4$ e così via, ci fermeremo quando la dimensione del problema diventa 1, a quel punto avremo fatto circa $\log n$ iterazioni. Avendo fatto quindi un numero costante di confronti per ogni iterazione, il numero massimo di confronti sarà $O(\log n)$ quindi il tempo di esecuzione della nostra funzione sarà $O(\log n)$.

VALUTAZIONE COMPLESSITÀ DELLE FUNZIONI RICORSIVE:

Negli algoritmi ricorsivi la soluzione di un problema si ottiene applicando lo stesso algoritmo ad uno o più sotto problemi che saranno di taglia inferiore a quella iniziale. Possiamo esprimere la complessità di questi algoritmi nella forma di una relazione di ricorrenza, abbiamo un albero decisionale in cui possiamo valutare alcune caratteristiche dell'algoritmo per decidere la sua complessità, la prima cosa che osserviamo è il lavoro di combinazione, cioè quel lavoro che fa l'algoritmo per preparare le chiamate ricorsive e poi il lavoro che farà dopo aver fatto le chiamate ricorsive per rielaborare i risultati ottenuti. Nell'albero decisionale che vedremo considereremo soltanto il caso in cui il lavoro di combinazione possa essere costante oppure lineare, quindi abbiamo due scelte, poi, avanzati di un livello, andremo a vedere la forma dell'equazione di ricorrenza che può essere con o senza partizione dei dati, ed infine l'ultima cosa che andremo a valutare sarà il numero di termini ricorsivi, cioè il numero di chiamate ricorsive nella funzione. A questo punto aggiungeremo un valore che ci esprime la complessità del nostro algoritmo ricorsivo. L'albero decisionale sarà costruito in questo modo:

1. Lavoro di combinazione costante

a) $T(n) = a_1 T(n-1) + a_2 T(n-2) + \dots + a_h T(n-h) + b$ per $n > h$

- Esponenziale con n: se sono presenti almeno 2 termini (l'algoritmo contiene almeno 2 chiamate ricorsive)
- Lineare con n: se è presente un solo termine (singola chiamata ricorsiva)

b) $T(n) = a T(n/p) + b$ per $n > 1$

- $\log n$ se $a = 1$ (singola chiamata ricorsiva)
- $n^{\log_p a}$ se $a > 1$ (più chiamate ricorsive)

2. Lavoro di combinazione lineare

a) $T(n) = T(n-h) + b n + d$ per $n > h$

Quadratico con n

b) $T(n) = a T(n/p) + b n + d$

- Lineare con n se $a < p$
- $n \log n$ se $a = p$
- $n^{\log_p a}$ se $a > p$

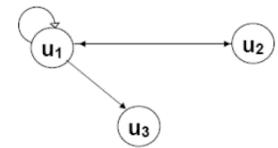
L08.1. ADT ALBERO

Un **grafo orientato G** è una coppia $\langle N, A \rangle$ dove:

- N è un insieme finito non vuoto di **nodi**;
- $A \subseteq N \times N$ è un insieme finito di coppie ordinate di nodi, detti **archi**.

Se la coppia ordinata di nodi $\langle u_i, u_j \rangle \in A$, vuol dire che nel grafo vi è un **arco diretto** da nodo u_i al nodo u_j .

In figura abbiamo che l'insieme $N = \{u_1, u_2, u_3\}$, l'insieme $A = \{(u_1, u_1), (u_1, u_2), (u_2, u_1), (u_1, u_3)\}$.



Un **albero** è un tipo particolare di grafo, ma anche una lista può essere considerata un tipo particolare di grafo.

L'albero è una struttura informativa che serve a rappresentare diverse cose, ad esempio organizzazioni gerarchiche di dati o partizioni successive di un insieme in sottoinsiemi disgiunti, un esempio di queste due prime categorie può essere il file system di un sistema operativo, ma può anche rappresentare procedimenti decisionali enumerativi, come l'albero decisionale per rappresentare la complessità degli algoritmi ricorsivi.

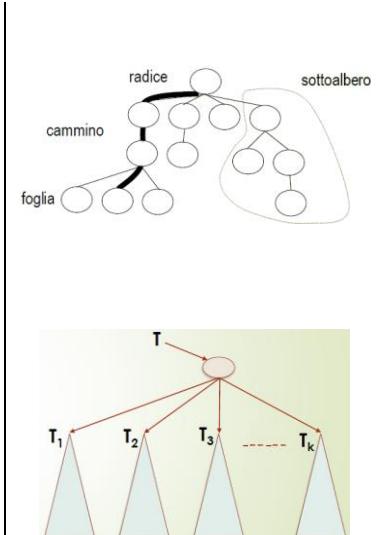
Questa struttura albero ha diverse proprietà, come:

- Ogni nodo ha un unico arco entrante, tranne la **radice**, che non ha archi entranti;
- Ogni nodo può avere zero o più archi uscenti, ed i nodi senza archi uscenti sono detti **foglie**;
- Un arco nell'albero induce una **relazione padre-figlio**;
- A ciascun nodo è solitamente associato un valore, detto **etichetta** del nodo;
- Il **grado** di un nodo è il numero di figli del nodo;
- L'**ordine** dell'albero è il grado max tra tutti i nodi;
- Un **cammino** è una sequenza di nodi $\langle n_0, n_1, \dots, n_k \rangle$ dove il nodo n_i è padre del nodo n_{i+1} , per $0 \leq i < k$;
- Il **livello** di un nodo è la lunghezza del cammino dalla radice al nodo stesso;
Definizione ricorsiva: il livello della radice è 0, il livello di un nodo non radice è $1 +$ il livello del padre.
- L'**altezza** dell'albero è la lunghezza del più lungo cammino, parte dalla radice e termina in una foglia.

Un albero è un grafo diretto aciclico, in cui per ogni nodo esiste un solo arco entrante (tranne la radice che non ne ha nessuno). Non tutti i nodi sono collegati tra loro da un cammino, ma se esiste un cammino che va da un nodo u ad un altro nodo v , tale cammino è sicuramente unico, la radice invece è collegata a qualunque altro nodo, quindi esiste sempre un cammino che va dalla radice ad un qualunque altro nodo.

Dato un nodo u , i suoi discendenti costituiscono un albero, detto **sottoalbero** di radice u .

Anche gli alberi hanno una natura ricorsiva, se consideriamo tutte le parti dell'albero collegate alla radice vediamo che ciascuna di esse costituisce un sottoalbero della radice e questo è a sua volta un albero.



ALBERI BINARI:

Sono dei particolari alberi e n-ari in cui ogni nodo può avere 0 o al più 2 figli, distinguendo due sottoalberi, ovvero il sottoalbero SX e il sottoalbero DX. Anche gli alberi binari possono essere definiti ricorsivamente:

- **Base**: albero binario vuoto;
- **Passo**: è una terna (s, r, d) , dove r è un nodo (la radice), s e d sono alberi binari.

Per la progettazione e l'implementazione degli alberi binari, avremmo come operatori un costruttore bottom-up, che ci consente di costruire un albero a partire da singoli nodi (partendo dalle foglie), degli operatori di selezione, in cui potremo ottenere delle parti dell'albero, e degli operatori di visita, mentre le etichette associate agli alberi potranno essere numeriche, stringhe o qualunque altro tipo di ADT a nostra scelta.

Per l'implementazione degli alberi binari possiamo utilizzare, come abbiamo fatto per le liste, una struttura **auto-referenziale**, questa struttura avrà due puntatori, uno alla radice del sottoalbero SX ed uno al DX, in più avremo l'item che è l'etichetta del nodo. Per rappresentare l'intero albero binario possiamo utilizzare un puntatore che punterà ad una di queste istanze che è la root dell'albero, se l'albero binario è vuoto questo puntatore sarà NULL.

La dichiarazione del tipo nodo sarà la seguente:

```
struct node{
    Item value;           /*etichetta del nodo*/
    struct node *left;   /*puntatore al sottoalbero sinistro*/
    struct node *right;  /*puntatore al sottoalbero destro*/
};
```

All'interno dell'interfaccia, cioè del file .h in cui andremo a realizzare la nostra libreria che realizza l'ADT albero binario, metteremo un puntatore ad una struttura struct node che chiameremo tramite un'istruzione di **typedef struct node *Btree**, questa variabile punterà al nodo radice dell'albero.

Quando andremo ad inizializzare questa struttura all'interno dell'operatore che ci costruisce un nuovo Btree, porremo **Btree T = NULL**, indicando che l'albero è inizialmente vuoto.

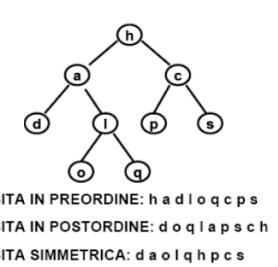
Per aggiungere nodi all'albero binario procederemo in modo bottom-up, cioè aggiungere un nodo alla volta dalle foglie, il motivo è che per ciascun nodo che andremo a creare dovremo indicare i suoi due sottoalberi e l'etichetta, quindi partendo dalle foglie potremo indicare che i due sottoalberi sono vuoti e dovremo soltanto aggiungere l'item all'interno del nodo. Salendo verso la radice invece avremo già creato i nodi foglia e quindi potremmo agganciarli ai nodi del livello immediatamente superiore e così via fino ad arrivare alla radice dell'albero.

Per **creare un nodo**, dobbiamo rispettare questi passi:

1. **Allocare** la memoria necessaria;
2. **Memorizzare** i dati nel nodo;
3. **Collegare** il sottoalbero SX e DX, già costruiti in precedenza.

Per la stampa delle informazioni contenute in una struttura albero, l'ordine con cui possiamo stampare questi nodi può cambiare, infatti sono noti tre algoritmi di visita che propongono tre modi differenti di visitare i nodi di un albero:

- **Visita in pre-ordine**: si applica ad un albero non vuoto e richiede dapprima l'analisi della radice dell'albero e, poi, la visita, effettuata con lo stesso metodo, dei due sottoalberi, prima il sinistro, poi il destro;
- **Visita in post-ordine**: si applica ad un albero non vuoto e richiede dapprima la visita, effettuata con lo stesso metodo, dei sottoalberi, prima il sinistro e poi il destro, e, in seguito, l'analisi della radice dell'albero;
- **Visita simmetrica**: richiede prima la visita del sottoalbero sinistro (effettuata sempre con lo stesso metodo), poi l'analisi della radice, e poi la visita del sottoalbero destro.



Sintattica	Semantica
Nome del tipo: BTree Tipi usati: Item, boolean	Dominio: $T = \text{nil} \mid T = \langle N, T_1, T_2 \rangle$ $N \in \text{NODO}, T_1 \in T_2$ sono BTree
newBTree() \rightarrow BTree	newBTree() \rightarrow T • Post: T = nil
isEmpty(BTree) \rightarrow boolean	isEmpty(T) \rightarrow b • Post: se T=nil allora b = true altrimenti b = false
buildBTree(Btree, Btree, Item) \rightarrow BTree	buildBTree(T1, T2, e) \rightarrow T • Pre: e != nil • Post: T = $\langle N, T_1, T_2 \rangle$; N ha etichetta e
getBTreeRoot(BTree) \rightarrow Item	getBTreeRoot(T) \rightarrow e • Pre: T = $\langle N, T_{\text{left}}, T_{\text{right}} \rangle$ non è vuoto • Post: N ha etichetta e
getLeft(BTree) \rightarrow Btree getRight(BTree) \rightarrow BTree	getLeft(T) \rightarrow T' • Pre: T = $\langle N, T_{\text{left}}, T_{\text{right}} \rangle$ non è vuoto • Post: T' = T_{left}

btree.c

```
#include "btree.h"
#include "item.h"

struct node{
    Item value;
    struct node *left;
    struct node *right;
};

BTree newTree(){
    return NULL;
}

int isEmptyTree(BTree t){
    if(t==NULL)
        return 1;
    return 0;
}

BTree buildTree(BTree l, BTree r, Item value){
    BTree t=malloc(sizeof(struct node));
    t->left=l;
    t->right=r;
    t->value=value;
    return t;
}

Item getBTreeRoot(BTree t){
    if(!isEmptyTree(t))
        return t->value;
    return NULL;
}

BTree getLeft(BTree t){
    if(!isEmptyTree(t))
        return t->left;
    return NULL;
}

BTree getRight(BTree t){
    if(!isEmptyTree(t))
        return t->right;
    return NULL;
}
```

```
void preOrder(BTree t){
    if(!isEmptyTree(t)){
        outputItem(t->value);
        preOrder(t->left);
        preOrder(t->right);
    }
}

void postOrder(BTree t){
    if(!isEmptyTree(t)){
        postOrder(t->left);
        postOrder(t->right);
        outputItem(t->value);
    }
}

void inOrder(BTree t){
    if(!isEmptyTree(t)){
        inOrder(t->left);
        outputItem(t->value);
        inOrder(t->right);
    }
}
```

btree.h

```
#include "item.h"

typedef struct node *BTree;

BTree newTree();
int isEmptyTree(BTree);
BTree buildTree(BTree, BTree, Item);
Item getBTreeRoot(BTree);
BTree getLeft(BTree);
BTree getRight(BTree);
void preOrder(BTree);
void postOrder(BTree);
void inOrder(BTree);
```

main.c

```
#include <stdio.h>
#include "item.h"
#include "btree.h"

int main(){
    Item h="h",a="a",d="d",l="l",o="o",q="q",c="c",p="p",s="s";
    BTree th,ta,td tl,to,tq,tc,tp,ts;
    td=buildTree(NULL,NULL,d); //FOGLIE
    to=buildTree(NULL,NULL,o);
    tq=buildTree(NULL,NULL,q);
    tp=buildTree(NULL,NULL,p);
    ts=buildTree(NULL,NULL,s);
    tl=buildTree(to,tq,l); //NODI INTERNI
    ta=buildTree(td,tl,a);
    tc=buildTree(tp,ts,c);
    th=buildTree(ta,tc,h);
    printf("preorder: ");
    preOrder(th);
    printf("\npostorder: ");
    postOrder(th);
    printf("\ninOrder: ");
    inOrder(th);
    printf("\n");
}
```

L08.2. ALBERO BINARIO DI RICERCA (BINARY SEARCH TREE)

Questo tipo di albero può essere utilizzato per la realizzazione di insiemi ordinati e supporta operazioni particolarmente efficienti di ricerca, inserimento e cancellazione.

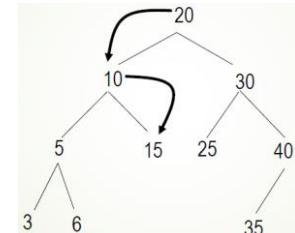
Definiamo questo nuovo ADT, se l'albero non è vuoto ogni elemento del sottoalbero di sinistra precede la radice:

- se prendiamo uno qualunque degli elementi del sottoalbero sinistro vediamo che la sua chiave è minore rispetto a quella della radice;
- se prendiamo un elemento del sottoalbero destro vedremo che questo avrà chiave maggiore di quella della radice;
- Inoltre, sia il sottoalbero sinistro che il sottoalbero destro sono a loro volta degli alberi binari di ricerca quindi godono della stessa proprietà della radice.

L'operatore **Search** ci consente di effettuare una ricerca di un elemento all'interno dell'albero, deve essere efficiente in quanto il BSTree è pensato proprio per rendere efficiente la ricerca all'interno di un insieme. Possiamo progettare questo algoritmo di ricerca ricorsivamente:

- **Base:** Se l'albero è vuoto allora restituisce null.
- **Passo:**
 1. Se l'elemento cercato coincide con la radice dell'albero restituisce l'item della radice;
 2. Se l'elemento cercato è minore della radice restituisce il risultato della ricerca dell'elemento nel sottoalbero sinistro;
 3. Se l'elemento cercato è maggiore della radice restituisce il risultato della ricerca dell'elemento nel sottoalbero destro.

Ricerca di 15:



L'operatore **min** ci consente di ottenere l'elemento minimo all'interno di un BSTree, lo stesso ragionamento che facciamo adesso per il minimo può essere fatto per il **max**. Possiamo implementare sia una versione ricorsiva che una iterativa di questo algoritmo. Per la prima versione ricorsiva:

- **Base:**
 1. Se l'albero è vuoto allora restituisci null;
 2. Se non esiste un sottoalbero sinistro (destro), ritorna l'item associato alla radice.
- **Passo:** Se esiste un sottoalbero sinistro (destro) effettua la ricerca del minimo (massimo) nel sottoalbero sinistro (destro).

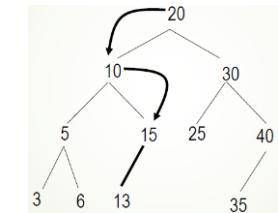
Pseudocodice versione iterativa:

```
Tree_minimum(x)
  while(x.left != NULL)
    x = x.left;
  return x;
```

L'operatore **insert** ci consente di inserire un nodo all'interno della struttura, anche in questo caso lo definiamo ricorsivamente:

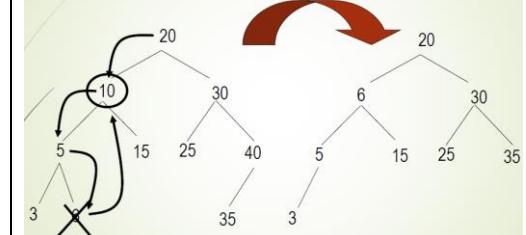
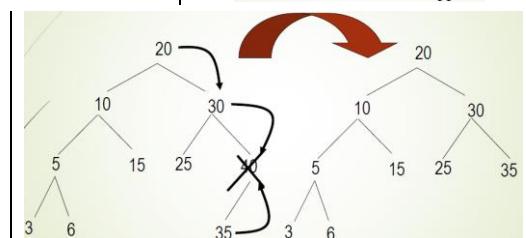
- **Base:** Se l'albero è vuoto allora crea un nuovo albero con un solo elemento.
- **Passo:**
 1. Se l'elemento coincide con la radice non si fa niente (elemento già presente);
 2. Se l'elemento è minore della radice allora lo inserisce nel sottoalbero sinistro;
 3. Se l'elemento è maggiore della radice allora lo inserisce nel sottoalbero destro.

Inserimento di 13:



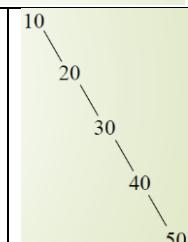
L'operatore **delete** ci consente di eliminare un nodo all'interno della struttura, la prima cosa che facciamo è di ricercare ricorsivamente il nodo da rimuovere, trovato il nodo da rimuovere si presentano due casi:

1. Se il nodo ha al più un solo sottoalbero di radice r, si bypassa il nodo da rimuovere e agganciando direttamente il suo unico sottoalbero al padre per poi rimuovere il nodo.
2. Se il nodo ha entrambi i sottoalberi, si sostituisce l'elemento da eliminare con il max nel sottoalbero sinistro (da notare che tale elemento non ha sottoalbero destro, la cui radice altrimenti sarebbe maggiore), alternativamente si cerca e si sostituisce con l'elemento minimo nel sottoalbero destro, infine si chiama ricorsivamente la delete sul sottoalbero sinistro del nodo contenente l'elemento max.



Per quanto riguarda la **complessità** delle operazioni di ricerca, inserimento e cancellazione, hanno tutte e tre la stessa complessità che è funzione dell'altezza dell'albero, quindi questa complessità sarà $O(h)$, poiché nel caso peggiore dovremo attraversare un percorso che parte dalla radice e arriva alla foglia più lontana, quindi attraverseremo l'albero per tutta la sua altezza. Se l'albero binario di ricerca fosse bilanciato, l'altezza dell'albero sarà $\log_2 n$, sarebbe un caso fortunato però non è detto che si verifica.

Ad esempio, creare l'albero e inserire i nodi le cui etichette sono ordinate in modo crescente (10, 20, 30, 40, 50), avremo quindi un albero che somiglia molto ad una lista, quindi l'altezza sarà lineare rispetto al numero di nodi dell'albero ed una qualunque operazione, come l'eliminazione o la ricerca del 50 ci costerà sempre un numero lineare di operazioni, quindi $O(n)$.



Sintattica	Semantica
<p>Nome del tipo: BST Tipi usati: Item, boolean</p> <p>newBST() → BST</p> <p>isEmpty(BST) → boolean getLeft(BST) → BST getRight(BST) → BST</p> <p>search(BST, Item) → Item min(BST) → Item max(BST) → Item</p> <p>insert(BST, Item) → BST</p> <p>delete(BST, Item) → BST</p>	<p>Dominio: T = nil T = <N, T1, T2> N ∈ NODO, T1 e T2 sono BST</p> <p>newBST() → T • Post: T = nil</p> <p>isEmpty(T) → b • Post: se T=nil allora b = true altrimenti b = false</p> <p>search(T, e) → e' • Pre: e ≠ nil • Post: e' = e se e ∈ T; e' = nil altrimenti</p> <p>insert(T, e) → T' • Post: T' contiene i nodi di T con l'aggiunta di e</p> <p>delete(T, e) → T' • Pre: T non è vuoto • Post: T' = T - {e}</p>

bst.c

```
#include "item.h"
#include "bst.h"
#include "queue.h"

struct node {
    Item value;
    struct node *left;
    struct node *right;
};

BST newBST(){ return NULL; }

int isEmptyBST(BST t){
    if(t==NULL) return 1;
    else return 0;
}

BST getLeft(BST t){
    if(isEmptyBST(t)) return NULL;
    else return t->left;
}

BST getRight(BST t){
    if(isEmptyBST(t)) return NULL;
    else return t->right;
}

Item getItem(BST t){
    if(isEmptyBST(t)) return NULL;
    else return t->value;
}

Item search(BST t, Item elem){
    if(isEmptyBST(t)) return NULL;
    else{
        int c;
        c=cmpItem(elem,t->value);
        if(c<0) return search(t->left,elem);
        else if(c>0) return search(t->right,elem);
        else return t->value;
    }
}

Item min(BST t){
    if(isEmptyBST(t)) return NULL;
    else if(t->left==NULL) return t->value;
    else return min(t->left);
}

Item max(BST t){
    if(isEmptyBST(t)) return NULL;
    else if(t->right==NULL) return t->value;
    else return max(t->right);
}
```

bst.h

```
void insertBST(BST *t, Item elem){
    if(isEmptyBST(*t)){
        *t=malloc(sizeof(struct node));
        (*t)->value=elem;
        (*t)->right=NULL;
        (*t)->left=NULL;
    }
    else if(cmpItem(elem,(*t)->value)<0)
        insertBST(&((*t)->left),elem);
    else if(cmpItem(elem,(*t)->value)>0)
        insertBST(&((*t)->right),elem);
}

Item deleteBST(BST *tree, Item elem){
    if(isEmptyBST(*tree)) return NULL;
    else{
        int c;
        c=cmpItem(elem,(*tree)->value);
        if(c<0) return deleteBST(&((*tree)->left), elem);
        else if(c>0) return deleteBST(&((*tree)->right), elem);
        else{
            BST temp = NULL;
            if(isEmptyBST((*tree)->right)){
                temp = *tree;
                *tree = temp->left;
                Item it = temp->value;
                free(temp);
                return it;
            }
            else if(isEmptyBST((*tree)->left)){
                temp = *tree;
                *tree = temp->right;
                Item it = temp->value;
                free(temp);
                return it;
            }
            else{
                Item it = max((*tree)->left);
                Item it2 = (*tree)->value;
                (*tree)->value = it;
                deleteBST(&((*tree)->left), it);
                return it2;
            }
        }
    }
}

void visitLayers(BST t) {
    if(isEmptyBST(t)) return;
    Queue q = newQueue();
    enqueue(q,t);
    while(!isEmptyQueue(q)){
        BST node = dequeue(q);
        outputItem(node->value);
        if(node->left != NULL) enqueue(q,node->left);
        if(node->right != NULL) enqueue(q,node->right);
    }
}
```

bst.h

```
#include "item.h"

typedef struct node *BST;

BST newBST();
int isEmptyBST(BST);
BST getLeft(BST);
BST getRight(BST);
Item getItem(BST);
Item search(BST, Item);
Item min(BST);
Item max(BST);
void insertBST(BST *, Item);
Item deleteBST(BST *, Item);
void visitLayers(BST);
```

L08.3. AVL

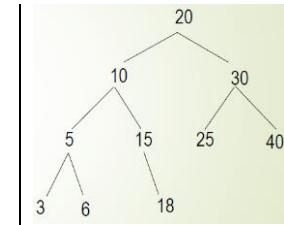
Per rendere efficienti gli alberi binari di ricerca possiamo utilizzare una particolare implementazione, che si chiama alberi AVL.

ALBERO BILANCIATO E ALBERO Δ -BILANCIATO:

Un albero binario di ricerca si dice Delta bilanciato se per ogni nodo la differenza in valore assoluto tra le altezze dei suoi due sottoalberi è minore uguale a Delta, per $\Delta=1$ si parla di alberi bilanciati.

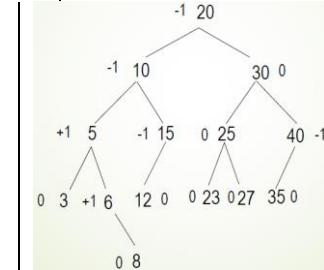
L'albero a destra è bilanciato poiché le altezze dei suoi due sottoalberi differiscono di un solo livello.

Un albero Delta bilanciato è particolarmente favorevole dal punto di vista dell'efficienza poiché si può dimostrare che la sua altezza è $\Delta + \log_2 n$, essendo Delta una costante le operazioni su questo albero binario di ricerca hanno complessità logaritmica.



Passiamo a parlare degli alberi **AVL**, che sono un esempio di alberi bilanciati, per evitare che l'albero non sia bilanciato dobbiamo aggiungere un ulteriore dato ad ogni nodo, in particolare questo dato sarà un marcatore che può assumere i seguenti valori:

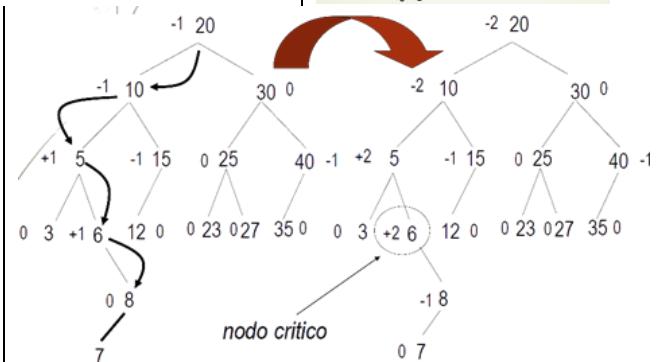
- -1, se l'altezza del sottoalbero sinistro è maggiore (di 1) dell'altezza del sottoalbero destro;
- 0, se l'altezza del sottoalbero sinistro è uguale all'altezza del sottoalbero destro;
- +1, se l'altezza del sottoalbero sinistro è minore (di 1) dell'altezza del sottoalbero destro.



Una operazione di inserimento o di cancellazione può provocare uno sbilanciamento dell'albero, quindi aggiornando sempre questi marcatori troveremo che un marcatore si troverà ad avere il valore +2 o -2, indicando che l'albero non è più bilanciato.

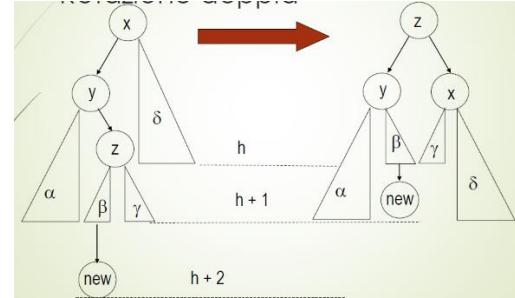
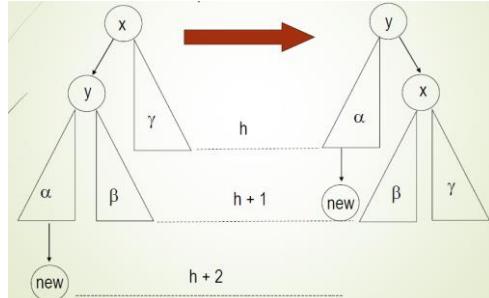
Se questo avviene bisogna ribilanciare l'albero, esistono delle **operazioni di rotazione** che possono essere **semplice** oppure **doppia**.

Si applicano queste operazioni sul nodo x a profondità massima che presenta un non bilanciamento, tale nodo viene detto **nodo critico** e si trova sul percorso che va dalla radice al nodo inserito oppure al nodo cancellato.



Dopo lo sbilanciamento dovremmo quindi eseguire un'operazione di rotazione sul nodo critico, abbiamo due tipi di rotazione:

- **Rotazione semplice**, quando avviene un inserimento nel sottoalbero sinistro del figlio sinistro del nodo critico, oppure inserimento nel sottoalbero destro del figlio destro del nodo critico;
- **Rotazione doppia**, quando avviene un inserimento nel sottoalbero destro del figlio sinistro del nodo critico, oppure inserimento nel sottoalbero sinistro del figlio destro del nodo critico.



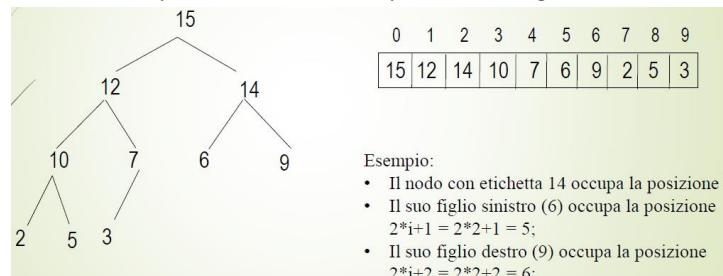
L08.4. CENNI SUGLI HEAP

Un **heap** è un albero binario bilanciato con le seguenti proprietà:

- Le foglie (nodi a livello h) sono tutte addossate a sinistra;
- Ogni nodo v ha la caratteristica che l'informazione ad esso associata è la più grande tra tutte le informazioni presenti nel sottoalbero che ha v come radice. Quindi la radice è più grande di tutti gli altri nodi contenuti nei suoi sottoalberi e questo vale per tutti i suoi sottoalberi.

Uno heap ha come principale applicazione la realizzazione delle **code a priorità** che sono degli ADT in cui possiamo inserire degli elementi oppure rimuovere soltanto l'elemento massimo, cioè quello che si trova nella radice.

L'heap si presta particolarmente bene ad essere rappresentato realizzato tramite un Array, i nodi sono disposti all'interno dell'array in ordine di livello e la radice occupa la posizione 0 dell'array, poi se un nodo occupa la posizione i il suo figlio sinistro occupa la posizione $2*i+1$ e il suo figlio destro occupa la posizione $2*i+2$. Possiamo passare molto rapidamente, con un semplice calcolo sugli indici, da un nodo ai suoi due sottoalberi.



L'operazione di **inserimento** in un Heap, viene fatto in ordine di livello, i passi sono:

- Si inserisce il nuovo nodo come ultima foglia;
- Il nodo inserito risale lungo il percorso che porta alla radice per individuare la posizione giusta.

Ad esempio, inseriamo il numero 18 →

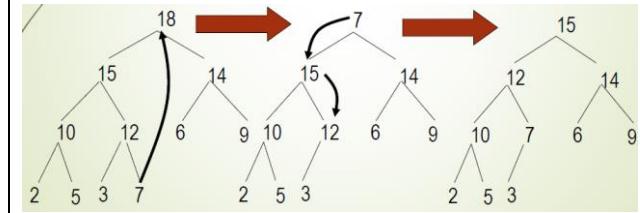
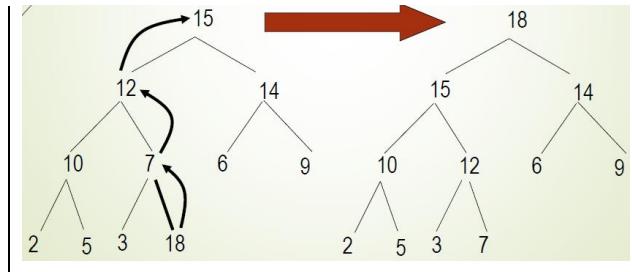
Il 18 è maggiore del suo nodo genitore, cioè il 7, e abbiamo violato la proprietà, facciamo risalire il 18 finché non sarà di nuovo bilanciato.

L'operazione di **rimozione** in un Heap si elimina sempre la radice, poiché nella realizzazione che viene utilizzata per realizzare le code a priorità, la cancellazione è sempre relativa nodo con valore massimo.

Quello che si fa è porre l'ultima foglia al posto della radice, quindi si scambia l'informazione contenuta nella radice con la maggiore dei suoi due sottoalberi e si ripete il procedimento fino ad arrivare ad una foglia.

Ad esempio, cancelliamo il 18, prendiamo l'ultima foglia e sostituiamola al 18, quello che otterremo è uno heap con un nodo in meno, violando il bilanciamento.

Per ribilanciare scambieremo il nodo 7 con il maggiore dei suoi due figli, ovvero il 15, e metteremo il 7 al posto del 15, ancora una volta scambieremo il 7 con il maggiore dei suoi due figli che in questo caso è il 12. A questo punto poiché il 7 è maggiore del 3 avremmo che l'heap è sistemato.



L09. ORDINAMENTI RICORSIVI

Esistono algoritmi di ordinamento, in forma ricorsiva, come il **Merge Sort** e **Quicksort**, che sono più efficienti rispetto ad altri visti in precedenza. Abbiamo detto che il problema dell'ordinamento consiste nell'elencare gli elementi di un insieme secondo una sequenza stabilità da una relazione d'ordine. Questo problema si può presentare in forme differenti, dalle più semplici alle più complesse, ad esempio ci si può chiedere di ordinare una sequenza di numeri oppure di mettere un elenco di nomi in ordine alfabetico, quindi possiamo avere tipi di dati differenti o una forma più complessa potrebbe essere ordinare i record degli studenti secondo la data di nascita, questo problema un po' più complesso sia perché abbiamo a che fare con dati strutturati sia perché abbiamo una taglia maggiore dell'input. In quest'ultimo caso dobbiamo ordinare dei record secondo una chiave che può essere un singolo campo, ma anche come in altri casi la combinazione di diversi campi.

Ricordiamo che questi algoritmi di ordinamento hanno diverse proprietà, ovvero:

- **Stabile**, due elementi con la medesima chiave mantengono lo stesso ordine con cui si presentavano prima dell'ordinamento.

Per tipi di dati semplici come gli interi o anche le stringhe non ci rendiamo conto se un algoritmo è stabile, quando invece abbiamo a che fare con dei record, un algoritmo stabile ci garantisce che dei record, che si presentavano ordinati prima dell'esecuzione dell'algoritmo, mantengono questo stesso ordinamento anche dopo l'esecuzione dell'algoritmo;

- **In loco**, in ogni istante al più è allocato un numero costante di variabili, oltre all'array da ordinare;

- **Adattivo**, il numero di operazioni effettuate dipende dall'input.

Ad esempio, tra gli algoritmi come Insertion Sort e Bubble Sort possono essere implementati in modo adattivo;

- **Interno** (dati contenuti nella memoria RAM) vs **esterno** (dati contenuti su disco o nastro o file).

Tutti gli algoritmi visti, compresi questi nuovi, ordinano per confronti e non tutti gli algoritmi di ordinamento fanno questo, infatti esistono algoritmi che non ordinano per confronti.

Nome	Migliore	Medio	Peggior	Memoria	Stabile	In Loco
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No	Sì
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Sì	Sì
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Sì	Sì
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(n)^*$	No	Sì
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Sì	No

*spazio aggiuntivo dovuto alla gestione della ricorsione

DIVIDE ET IMPERA:

Per quanto riguarda il Merge Sort ed il Quicksort, entrambi questi algoritmi utilizzano, per la risoluzione del problema dell'ordinamento, un approccio chiamato **Divide et impera**. Questo approccio può essere usato per risolvere vari problemi computazionali ed ha diverse caratteristiche:

- **Divide**: si procede alla suddivisione dei problemi in problemi di dimensione minore;
- **Impera**: i problemi vengono risolti in modo ricorsivo. Quando i sotto-problemi arrivano ad avere una dimensione sufficientemente piccola (si arriva al caso base), essi vengono risolti in modo immediato;
- **Combina**: si ricombina l'output, dopo la chiamata ricorsiva, ottenuto dalle precedenti chiamate ricorsive al fine di ottenere il risultato finale.

MERGE SORT:

È un algoritmo che utilizza il paradigma del **Divide et impera**, possiamo ottenere una versione stabile però richiede uno spazio ausiliario $O(n)$, quindi non possiamo dire che ordina in loco, ma è molto efficiente per tutte le dimensioni dell'input, in genere viene fornito come algoritmo standard in molte librerie di alcuni linguaggi di programmazione, come ad esempio Java.

Per quanto riguarda la progettazione del Merge Sort, si divide a metà il vettore, quindi avremo un sotto-vettore sinistro e un sotto-vettore destro rispetto al centro:

▪ Divide:

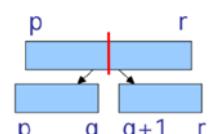
Supponiamo che abbiamo un vettore che vada dall'indice p all'indice r e che il centro si trova all'indice q . Quindi avrei un sotto-vettore sinistro che va da p a q , e non sotto-vettore destro che vada $q+1$ ad r .

▪ Impera:

Identifichiamo il caso base che è quello in cui abbiamo un sotto-vettore di taglia unitaria oppure di taglia zero, quindi quando $p=r$ oppure $p>r$, questo sotto-vettore si può considerare già ordinato. Nel caso in cui invece non abbiamo un vettore semplice di taglia unitarie dovremmo fare due chiamate ricorsive, quindi chiameremo ricorsivamente il merge sort sul sotto-vettore di sinistra e poi sul sotto-vettore di destra.

▪ Combina:

Chiamiamo una procedura, detta **merge**, che fonde i due sotto-vettori ordinati in un unico vettore ordinato, sceglie ripetutamente il minimo dei due sotto-vettori e lo mette in una sequenza di output.



ANALISI FUNZIONE MERGE

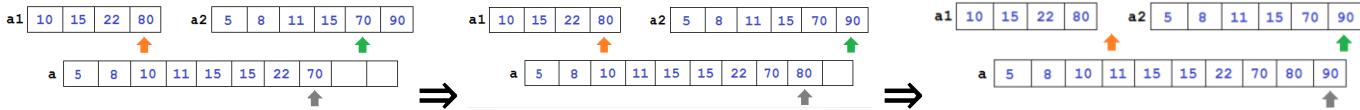
- Dati di ingresso: Array a_1 di n_1 elementi, a_2 di n_2 elementi
- Precondizione: $\forall 1 \leq i \leq n_1 \ a_1[i-1] \leq a_1[i]; \forall 1 \leq j \leq n_2 \ a_2[j-1] \leq a_2[j]$
- Dati di uscita: Array a di n_1+n_2 elementi
- Postcondizione: $\forall 1 \leq i \leq n_1+n_2: a[i-1] \leq a[i]$

Identificatore	Tipo	Descrizione
Dizionario dei dati	a_1, a_2 n_1, n_2 a i, j	array intero array intero array di interi in input # di elementi negli array a_1, a_2 array di interi in output usati per indicizzare i vettori

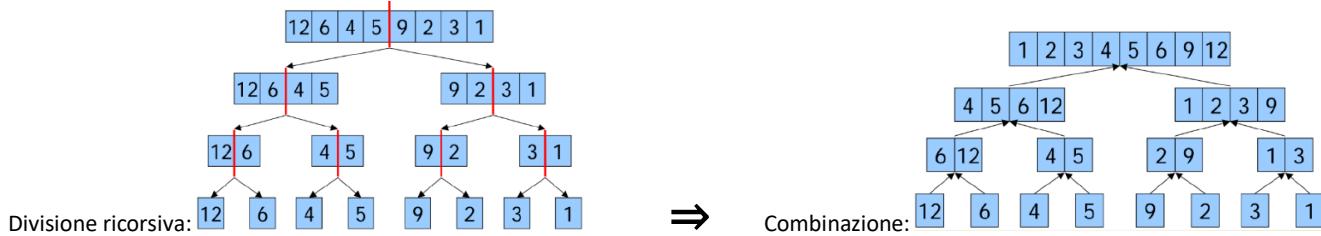
PROGETTAZIONE FUNZIONE MERGE

- Scorriamo i due vettori a_1 e a_2 utilizzando due indici i e j , rispettivamente;
- Confrontiamo $a_1[i]$ e $a_2[j]$ finché $i < n_1$ e $j < n_2$:
 - Se $a_1[i] \leq a_2[j]$: Inseriamo $a_1[i]$ in a e incrementiamo i ;
 - Altrimenti: inseriamo $a_2[j]$ in a e incrementiamo j ;
- Riversiamo tutti gli elementi restanti in a_1 o in a_2 in a .

Esempio Merge:



Esempio Merge Sort:



Per quanto riguarda il costo computazionale:

$$\text{Livelli di ricorsione: } \log_2 n \quad \text{Operazioni per livello: } n$$

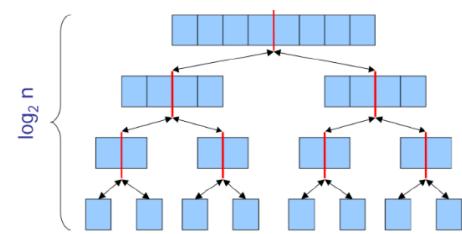
Operazioni totali: $n \log_2 n$

Equazione alle ricorrenze:

- $T(n) = 2T(n/2) + \Theta(n)$ per $n \geq 2$
- $T(1) = 1$

Soluzione:

- $T(n) = \Theta(n \log n)$



QUICKSORT:

Come nel caso del merge sort, è un algoritmo ricorsivo che fa uso del paradigma **Divide et impera**, a differenza del merge sort è in grado di ordinare un Array in loco, cioè non richiede una Array addizionale in cui riversare i valori durante l'esecuzione, però a differenza del merge sort non è stabile, quindi se prendiamo due elementi con la stessa chiave che prima dell'ordinamento si presentano in un certo ordine, non è detto che dopo l'esecuzione del quicksort questi si ripresentino nello stesso ordine.

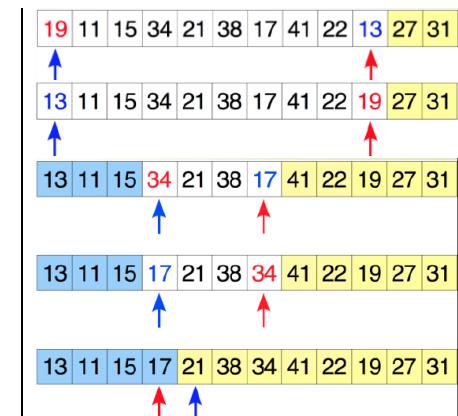
Il quicksort si basa su un algoritmo che viene chiamato **Partition**, questo algoritmo, dato un elemento **pivot**, mette tutti gli elementi minori uguali del pivot a sinistra e tutti gli elementi maggiori uguali del pivot sul lato destro dell'Array, restituirà la posizione in cui l'Array è stato suddiviso in questi numeri che sono minori e maggiori rispetto al pivot.

Supponiamo di aver scelto come pivot il valore 17, a questo punto, partiremo con due indici il blu sul lato sinistro dell'Array e il rosso sul lato destro.

Il puntatore rosso arretra finché non trova un elemento che sia minore o uguale del pivot, mentre il puntatore blu avanza finché non trova un elemento maggiore o uguale del pivot.

Il puntatore rosso scenderà di 3 posizioni fino ad arrivare a puntare al valore 13 e si fermerà poiché il 13 è minore o uguale del 17, il puntatore blu avanza e si ferma alla prima casella dell'array in quanto il contenuto è 19, che è maggiore uguale rispetto al pivot. A questo punto scambiamo i due elementi, cioè quello puntato dal puntatore blu e quello puntato dal puntatore rosso.

La procedura termina quando i due puntate si incrociano, infatti il puntatore posso scendere fino ad arrivare alla locazione contenente il 17, ma a questo punto i due puntatori si saranno incrociati e quindi termineremo la nostra procedura.



DESCRIZIONE PROCEDURA PARTITION:

Il primo passo è quello di scegliere il pivot, quindi mettiamo in x il valore del pivot che indicheremo con $A[p]$, poi eseguiremo un ciclo infinito che terminerà quando i due puntato si incroceranno, quindi individueremo $A[i]$ ed $A[j]$ i due elementi che si trovano fuori posto, in particolare troveremo $A[j]$ decrementando il puntatore j fino a trovare un elemento minore uguale del pivot x e incrementeremo i fino a trovare un elemento maggiore o uguale del pivot x . Se i due puntatori si sono incrociati restituiremo il valore di j che sarà esattamente il punto in cui l'Array risulta diviso da questa procedura, quindi si troverà il puntatore j sul valore dell'ultimo elemento del sotto-array di sinistra. Infine, come ultima istruzione del while scambieremo $A[i]$ ed $A[j]$.

Procedura Partition:

- Pivot $x=A[p]$;
- While(1)
- Individua $A[i]$ e $A[j]$ elementi "fuori posto"
 - Decrementa j fino a trovare un elemento minore o uguale del pivot x
 - Incrementa i fino a trovare un elemento maggiore o uguale del pivot x
- If($i \geq j$) return j
- Scambia $A[i]$ e $A[j]$

Vediamo quindi come utilizzare questa procedura Partition all'interno dell'algoritmo di ordinamento. Quicksort è un algoritmo ricorsivo che utilizza il paradigma **Divide et impera**, in particolare avremo a che fare con un array A i cui indici per comodità indicheremo da p ad r , $A[p...r]$.

▪ Divide:

Prevede di partizionare tale vettore in due sotto-vettori SX e DX rispetto ad un pivot x che si trova in una posizione q , questa posizione per come abbiamo progettato il nostro algoritmo Partition verrà restituito come valore di ritorno di tale procedura.

▪ Impera:

Eseguiamo il quicksort prima sul sotto-vettore SX $A[p...q]$ e poi chiameremo ricorsivamente quicksort su DX $A[q+1...r]$. La base della ricorsione avviene quando il vettore ha un unico elemento e quindi lo possiamo considerare un sotto-vettore ordinato.

Analizziamo il comportamento asintotico del quicksort, questo algoritmo ha un metodo di ordinamento basato su confronti, quindi quello che faremo è contare il numero di confronti che questo algoritmo effettua, inclusi i confronti che fa la Partition.

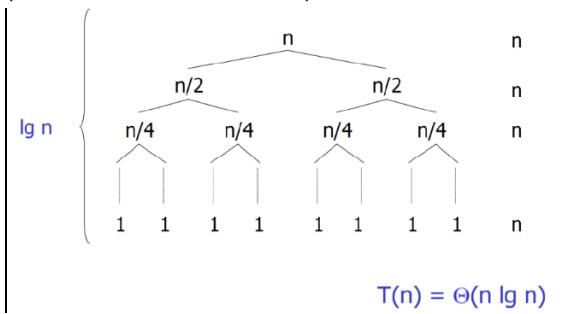
In realtà tutti i confronti vengono eseguiti nella procedura Partition poiché la procedura quicksort è molto semplice e contiene soltanto la chiamata da Partition e le due chiamate ricorsive. La procedura Partition fa avanzare quei due puntatori finché non si incrociano, ad ogni avanzamento o arretramento di uno dei due puntatori avremo che si eseguirà un confronto, quindi è corretto dire che se abbiamo un array di lunghezza m , eseguiremo m confronti.

L'array viene quindi partizionato, cioè diviso in due parti, e faremo delle chiamate ricorsive prima sulla parte sinistra e poi sulla parte destra, l'equazione di ricorrenza sarà:

$$T(n) = \begin{cases} 0 & \text{se } n = 1 \\ T(r) + T(n-r) + n & \text{se } n > 1 \end{cases}$$

La forma di questa ricorsione non rientra nei casi che abbiamo studiato finora, a meno che non specializziamo i nostri casi (peggiore, migliore e medio). L'efficienza del Quicksort dipende da come va il bilanciamento delle partizioni, cioè come la Partition avrà diviso l'array, se è diviso in parti quasi uguali allora avremo un comportamento quasi ottimale, se invece l'algoritmo di Partition divide in modo molto sbilanciato gli Array avremo che l'algoritmo avrà un tempo di esecuzione peggiore, in particolare il **caso peggiore** avviene quando la divisione della Partition ci restituisce un Array da un singolo elemento e l'altro da $n-1$ elementi, ma si può anche presentare quando l'array è già ordinato (sia decrescente che crescente).

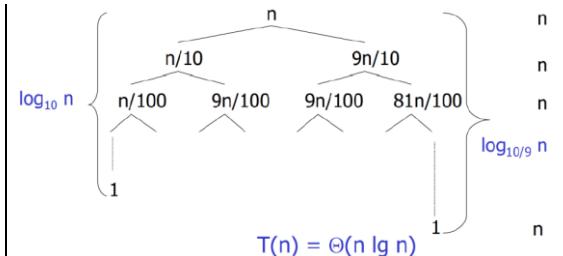
Il **caso migliore** invece si verifica quando l'array viene diviso esattamente a metà, quindi i due array saranno di $n/2$ elementi, possiamo vederlo con questo grafico:



È chiaro che il bilanciamento può essere più o meno buono a seconda di come viene scelto il pivot nella Partition.

Ed infine abbiamo un **caso medio**, possiamo vedere vari casi e capire se dobbiamo aspettarci più frequentemente un tempo quadratico oppure un tempo $n \log n$.

Supponiamo di essere sfortunati ed abbiamo che le due regioni sia una 9 volte più grande dell'altra:



Riassumendo, se abbiamo delle partizioni bilanciate, l'algoritmo ordina in tempo $\Theta(n \log n)$, se invece siamo sfortunati e abbiamo delle partizioni sempre sbilanciate, avremo un costo di n^2 . Tuttavia, considerando vari casi intermedi il tempo di esecuzione può essere sempre ricondotto ad un tempo asintotico $\Theta(n \log n)$, quindi possiamo dire che nel caso medio, il quicksort ordina in tempo $n \log n$.

RANDOM PIVOTING:

Abbiamo detto che la scelta del pivot determina il bilanciamento della partizione e che la scelta del primo elemento come pivot, nel caso in cui la sequenza di ingresso si presenti già ordinata, risulta in un partizionamento sfavorevole, quello che possiamo fare per evitare che delle sequenze particolari di ingresso ci creino questo problema di sbilanciamento è cambiare schema di scelta del pivot, non più il primo elemento ma una scelta casuale del pivot.

Infatti, il tempo di esecuzione del quicksort con **pivot casuale** è $\Theta(n \log n)$ con una probabilità molto alta. Tuttavia, la scelta casuale ci costringe a chiamare una procedura di scelta di numeri pseudocasuali ad ogni esecuzione della Partition, questa cosa può diventare svantaggiosa dal momento che gli algoritmi di scelta numeri casuali hanno un loro costo, potrebbero rallentare l'esecuzione del quicksort, per questo abbiamo degli altri schemi di scelta del pivot che sono molto più rapidi, in particolare possono essere fatti in tempo costante.

Possiamo scegliere la strategia "**medio di 3**" che consiste nel considerare elementi che sono nella prima e ultima posizione dell'array e l'elemento che si trova in posizione mediana e scegliere di questi il valore mediano.

Supponiamo di avere un Array di interi, sceglieremo come pivot il valore intermedio tra 39 che è il primo elemento, 31 è l'ultimo e 18 che è quello intermedio.



Sceglieremo il valore di mezzo che in questo caso è il 31, che sarà il nostro pivot, è possibile scegliere questo valore mediano facendo due confronti quindi può essere fatta questa scelta in tempo costante.

MERGESORT

```
void merge(Item *a,int na, Item *b, int nb, Item *c){
    int i=0,j=0,k=0;
    Item v[na+nb];
    for(;i<na && j<nb; ++k)
        if(cmpItem(a[i], b[j]) <= 0)
            v[k] = a[i++];
        else
            v[k]=b[j++];
    while(i<na)
        v[k++]=a[i++];
    while(j<nb)
        v[k++]=b[j++];
    for(k=0;k<na+nb;++k)
        c[k]=v[k];
}

void mergeSort(Item *a,int n){
    if(n>1){
        mergeSort(a,n/2);
        mergeSort(a+n/2,n-n/2);
        merge(a,n/2,a+n/2,n-n/2,a);
    }
}
```

QUICK SORT

```
void quickSort(Item*a,int n){    qSort(a,0,n-1);    }

void qSort(Item*a, int low, int high){
    if(high>low){
        int x = partition(a,low,high);
        qSort(a,low,x);
        qSort(a,x+1,high);
    }
}
int partition(Item*a, int low, int high){
    Item pivot = a[low];
    int i = low-1, j = high+1;
    while(1){
        do{
            j--;
        }while(cmpItem(a[j],pivot)>0);
        do{
            i++;
        }while(cmpItem(a[i],pivot)<0);
        if(i>=j) return j;
        swap(&a[i],&a[j]);
    }
}
```

L10. TABELLE HASH

Una **tabella hash** è una struttura dati usata per mettere in corrispondenza una data chiave con un dato valore, sia la chiave che il valore possono appartenere a diversi tipi primitivi o strutturati. Ad esempio, associare l'età, che è un intero, ad una persona utilizzando il suo nome, che è una stringa. Il linguaggio C non fornisce un supporto diretto per le tabelle hash, per surrogarle possiamo utilizzare un array, a patto però che si associno prima gli indici interi alle persone e poi l'età agli indici.

Per creare tabelle hash dobbiamo memorizzare delle coppie chiave-valore, chiamate **entry**, implementeremo tre operatori di base:

- **INSERT(key, value)**: Inserisce un elemento nuovo, con un certo valore (unico) di un campo chiave, cioè una nuova coppia chiave-valore;
- **SEARCH(key)**: Determina se un elemento con un certo valore della chiave esiste, se esiste, lo restituisce;
- **DELETE(key)**: Elimina l'elemento identificato dal campo chiave, se esiste, cioè forniamo la chiave associata a quella entry.

Oltre a questa funzione di base è possibile supportare una serie di altre funzioni come l'ordinamento dell'insieme dei dati oppure la ricerca del massimo o del successore e così via. Alcune definizioni:

- **U** - indichiamo l'**universo di tutte le possibili chiavi**;
- **K** - l'insieme delle **chiavi effettivamente memorizzate**.

CHIAVI INTERE – INDIRIZZAMENTO DIRETTO:

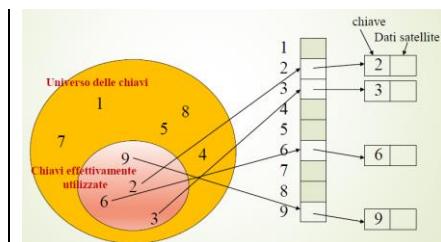
Se l'universo delle chiavi è piccolo e le chiavi sono intere, allora possiamo utilizzare una **tabella ad indirizzamento diretto**, che è un array.

Nell'array avremo quindi che ad una data chiave intera corrisponde una posizione, detta anche **slot**, all'interno della tabella. Come sappiamo possiamo accedere in tempo costante a ciascuna posizione all'interno della tabella data la chiave, perché il meccanismo di **indirizzamento** è esattamente quello dell'array.

Quando l'universo delle chiavi è piccolo, come gli interi che vanno da 1 a 9, soltanto una parte di queste chiavi sono effettivamente utilizzate e quello che possiamo fare è usare la chiave come indice.

Per esempio, preso l'elemento con chiave 3 lo andiamo a memorizzare all'interno dell'elemento 3 dell'array e questo elemento 3 avrà un puntatore al dato che è che una coppia chiave-valore, cioè una entry all'interno della tabella.

Conviene utilizzare questo schema quando l'universo delle chiavi è piccolo e solo una parte significativa di questo universo delle chiavi viene effettivamente utilizzato.



Ma se le chiavi non sono intere oppure se l'universo delle possibili chiavi è molto grande, allora diventa non più conveniente o impossibile utilizzare questo metodo delle tabelle ad indirizzamento diretto. Se l'universo delle chiavi è troppo grande, può non essere possibile avere una tabella con tutte queste righe oppure se comunque il numero di chiavi utilizzate è molto minore rispetto alla taglia dell'universo delle chiavi, avremo delle tabelle con pochi dati, quasi vuote, ed avremo quindi uno spreco di spazio inaccettabile.

Ricapitolando abbiamo due casi:

- Se $|K| \sim |U|$, avremo che la taglia delle chiavi effettivamente utilizzate è simile alla taglia dell'universo delle chiavi:
In questo caso possiamo utilizzare una tabella ad **indirizzamento diretto** perché non spreciamo molto spazio e abbiamo il vantaggio che le operazioni possono essere fatte in *tempo costante*.
- Se $|K| \ll |U|$, avremo che la taglia delle chiavi effettivamente utilizzate è molto minore rispetto alla taglia dell'universo delle chiavi:
La soluzione della tabella indirizzamento diretto non è più praticabile, ad esempio, se volessimo memorizzare in una tabella ad indirizzamento diretto il record degli studenti utilizzando la matricola come chiave, quello che succede è che se la matricola a 6 cifre l'array deve avere spazio per contenere 10^6 elementi. Supponiamo che gli studenti siano 30, lo spazio realmente occupato dalle chiavi memorizzate è $30/10^6 = 0,003\%$ dello spazio effettivamente allocato.

Per conservare l'efficienza nelle operazioni ma avere anche un compromesso riguardante i requisiti di memoria, quindi utilizzare una memoria poco più grande rispetto a quella effettivamente utilizzata, ricorriamo alle tabelle hash.

La differenza sostanziale tra il **metodo di indirizzamento diretto** e le **tabelle hash**, consiste nel fatto che mentre il primo memorizza nella tabella in posizione k un elemento con chiave k , nel metodo hash invece un elemento con chiave k viene memorizzato nella tabella in posizione $h(k)$ dove h è una funzione detta **funzione hash**, il cui scopo è di definire una corrispondenza tra l'universo U delle chiavi e le posizioni di una tabella hash $T[0..m-1]$, con m minore rispetto alla taglia dell'universo delle chiavi. Questa funzione h è definita come $h: U \rightarrow \{0, 1, \dots, m-1\}$.

GESTIONE DELLE COLLISIONI:

La **funzione hash** ci consente di ricavare l'indice in cui andare a mettere il valore corrispondente ad una data chiave. L'inconveniente è che la funzione hash non può essere **iniettiva**, cioè due chiavi distinte possono produrre lo stesso valore hash, quando questo avviene abbiamo due chiavi $k_i \neq k_j$ e avviene che $h(k_i) = h(k_j)$, diremo che si è verificata una **collisione**.

Per avere una struttura dati efficiente occorre minimizzare il numero di collisioni e quindi ottimizzando la **funzione hash**, però comunque gestire le collisioni che comunque possono avvenire, cioè quando avviene una collisione deve essere possibile inserire più elementi all'interno della stessa locazione oppure trovare un modo per inserire nella tabella entry che hanno una stessa chiave.

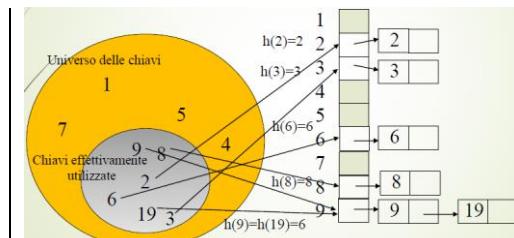
Esistono due metodi principali per **risolvere le collisioni**:

- **Metodo di concatenazione**;
- **Metodo di indirizzamento aperto**.

METODO DI CONCATENAZIONE:

Sfrutta l'idea di mettere tutti gli elementi che collidono in una lista concatenata, quindi la tabella avrà nella posizione j -esima una lista che contiene tutti gli elementi associati a quel dato valore hash e un puntatore nullo se non ci sono elementi. L'implementazione di questo metodo realizzerà la tabella hash come un vettore di liste.

Supponiamo di avere un universo delle chiavi che corrisponde ai numeri interi e poche chiavi effettivamente utilizzate, abbiamo una tabella che è un array indicizzato da 1 a 9, a ciascuna posizione è associata una lista, in particolare nella posizione 9 abbiamo due elementi con chiavi diverse, 9 e 19, però la cui funzione hash associa lo stesso valore, ovvero 6, quindi due elementi sono inseriti nella stessa posizione utilizzando una lista.



FUNZIONI HASH:

Come facciamo, dato un qualunque valore, ad ottenere l'indice con cui andare ad indicizzare la entry nella tabella. Una caratteristica importante è che deve avere una funzione hash è il cosiddetto **criterio di uniformità semplice** che dice che il valore hash di una chiave k è uno dei valori 0...m-1 in modo equiprobabile, questo ci consente di minimizzare le collisioni.

Un altro requisito è che una buona funzione hash dovrebbe usare tutte le cifre della chiave per produrre un valore hash.

Una **funzione hash** molto semplice, ma anche molto utilizzata, quando abbiamo a che fare con chiavi a **valori interi** è il **metodo di divisione**, la funzione hash è del tipo: $h(k) = k \text{ mod } m$, cioè il valore hash è il resto della divisione di k per m. Questo metodo è molto veloce e può essere calcolata con una semplice operazione.

Invece una **funzione hash** per il **tipo stringa**, per convertire la stringa in un numero naturale che poi è l'indice della tabella, consideriamo la stringa come un numero in base 128, infatti questo numero è il numero di simboli diversi per ogni cifra di una stringa.

Esiste una codifica che associa un simbolo ad un numero naturale ad esempio la codifica ASCII, la conversione viene fatta nel modo tradizionale cioè prendendo ogni carattere che compone la stringa e moltiplicando per 128 elevato alla posizione, per esempio se dobbiamo convertire in numero la stringa "pt" faremo " $p''*128^1 + "t''*128^0$ che verrà 14452.

Questo tipo di conversione può funzionare bene per stringhe molto corte, ma già se abbiamo una stringa un po' più lunga la cosa diventa non fattibile perché il numero che otteniamo comincia a diventare troppo grande per poter essere rappresentato. Conviene avere una funzione hash modulare che trasforma un pezzo di chiave alla volta, sfruttiamo il fatto che l'equazione, con i vari prodotti per 128 elevato alla posizione, può essere riscritta in modo più conveniente. Questo modo è dato dalla **regola di Horner** che ci consente di ridurre notevolmente il numero di moltiplicazioni che dobbiamo fare per valutare la nostra espressione.

$(((((((97*128 + 118)*128 + 101)*128 + 114)*128+121)*128 + 108)*128 + 111)*128 + 110)*128 + 103)*128 + 107)*128 + 101)*128 + 12).$

Come possiamo vedere nella nuova espressione non abbiamo più l'elevamento a potenza ma abbiamo delle semplici moltiplicazioni alternate a delle addizioni.

```
int hash(char*v, int m){  
    int h = 0, a = 128;  
    for (; *v != '\0'; v++)  
        h = (h*a + *v) % m;  
    return h;  
}
```

La funzione restituisce un valore intero che è l'indice, prende come parametro una stringa v e un intero m che è la taglia della tabella. Inizializza un valore $h=0$ e $a=128$, il valore h sarà il valore che ritorneremo come indice. Scorriamo l'intera stringa e andiamo ad inserire in h il valore di questa espressione, $h*a +$ il valore ASCII della stringa, il tutto modulo m.

In ogni iterata facciamo un'operazione di modulo che ci consente di mantenere basso il numero che dobbiamo restituire. Questo calcolo ci consente di evitare l'overflow degli interi.

METODO DI INDIRIZZAMENTO APERTO:

In questo metodo non utilizziamo più una lista per ogni cella ma utilizziamo tutta la tabella, quindi memorizziamo gli elementi sempre all'interno della tabella ma in una locazione differente se quella iniziale risulta essere occupata.

Il metodo con cui troviamo una nuova locazione in cui andare a salvare un elemento è quello di generare un nuovo valore hash fino a trovare una nuova posizione vuota in cui inserire l'elemento, più nel dettaglio estendiamo la nostra funzione hash perché generi non solo un valore hash ma una **sequenza di scansione**.

Quindi la nostra funzione hash diventa una funzione di due variabili, una prima variabile presa dall'universo delle chiavi, una seconda variabile intera che vada 0 a m-1 dove m è la dimensione della tabella e il risultato è un valore compreso tra 0 ed m-1, $h : U \times \{0,1,\dots,m-1\} \rightarrow \{0,1,\dots,m-1\}$, cioè prenda in ingresso una chiave e un indice di posizione e generi una nuova posizione.

La nostra funzione hash non genera più un singolo indice ma genera una **sequenza di scansione**, data una chiave k.

Per prima cosa si genera un indice con $h(k, 0)$, se questo indice generato corrisponde ad una posizione già occupata, avremmo avuto quindi una collisione, genereremo un secondo indice utilizzando $h(k, 1)$, in caso di ulteriori collisioni genereremo $h(k, 2)$ e così via con $h(k, i)$.

I valori generati dalla funzione hash sono una sequenza $\langle h(k, 0), \dots, h(k, m-1) \rangle$.

INSERIMENTO	RICERCA	ELIMINAZIONE
<pre>Hash-Insert(T,k) 1 i ← 0 2 repeat j ← h(k,i) 3 if T[j]=NIL 4 then T[j] ← k 5 return j 6 else i ← i+1 7 until i=m 8 error "overflow"</pre>	<pre>Hash-Search(T,k) 1 i ← 0 2 repeat j ← h(k,i) 3 if T[j]=k 4 then return j 5 i ← i+1 6 until i=m o T[j]=NIL 7 return NIL</pre>	<p>Questa operazione risulta essere abbastanza difficile. Il motivo è che non possiamo marcare una posizione vuota con un valore Nullo, perché comporterebbe il fallimento del codice della funzione di ricerca, questa viene interrotta quando si trova un valore Nullo perché vuol dire che la chiave non è stata trovata, ma è possibile che sia stata inserita in una posizione di scansione successiva, quindi il valore Nullo non può essere utilizzato come marcatore per indicare un elemento cancellato. Se invece si vuole supportare la cancellazione bisogna utilizzare uno speciale marcatore. Una posizione cancellata si può poi sovrascrivere quando viene effettuato un inserimento.</p>

CARATTERISTICHE DI h:

Una proprietà desiderabile delle funzioni hash è quella di **uniformità semplice** riferita in particolare alla proprietà di una buona funzione hash di poter generare con la stessa probabilità tutti i valori interi dell'intervallo compreso nel range della tabella. Estendiamo tale concetto anche per le funzioni hash utilizzate nello schema ad **indirizzamento aperto**, in particolare la proprietà di uniformità della funzione hash prevede che per ogni chiave k, la sequenza di scansione generata da h, deve essere una qualunque delle m fattoriale permutazioni degli interi compresi tra 0 ed m-1, quindi per essere ideale la nostra funzione h deve poter generare tutti i valori del range della tabella una volta sola.

Purtroppo, rispettare questo criterio di uniformità per una funzione hash è molto difficile, quindi si ricorre a delle approssimazioni. Ci sono diversi schemi per poter realizzare funzioni che hanno una buona approssimazione di questa proprietà di uniformità. In particolare, abbiamo:

- **Scansione lineare;**
- **Scansione quadratica;**
- **Hashing doppio.**

Queste funzioni riescono a generare una permutazione ma non riescono a generare tutte le m fattoriale permutazioni.

Data una funzione $h' : U \rightarrow \{0, 1, \dots, m-1\}$, il metodo di scansione lineare costruisce una $h(k,i) = (h'(k)+i) \text{ mod } m$, questo equivale a generare tutti gli indici di posizione consecutivamente, per prima cosa si genera la posizione $h'(k)$, quindi la posizione $h'(k)+1$, e così via fino alla posizione $m-1$, poi si scandisce in modo circolare la posizione 0,1,2 fino a tornare a $h'(k)-1$. Abbiamo quindi generato tutte le posizioni a partire da una data posizione e ciclando circolarmente su tutta la tabella.

ADT Key

Sintattica	Semantica
Nome del tipo: Key Tipi usati: Int, boolean	Dominio: $k \in U$
equals(Key, Key) → boolean	equals(k_1, k_2) → b • Post: b = true se $k_1=k_2$; b = false altrimenti
hashValue(Key, int) → int	hashValue(k , size) → index • Pre: $k \neq \text{nil}$, size > 0 • Post: $0 \leq \text{index} < \text{size}$
inputKey() → Key outputKey(Key)	

ADT Entry

Sintattica	Semantica
Nome del tipo: Entry Tipi usati: Item, Key	Dominio: Coppia (chiave, valore) chiave è di tipo Key, valore è di tipo Item
newEntry(Key, Item) → Entry	newEntry(key, value) → e • Post: e = (key, value)
getKey(Entry) → Key getValue(Entry) → Item	getKey(e) → key • Post: e = (key, value) getValue(e) → value • Post: e = (key, value)

ADT HashTable

Sintattica	Semantica
Nome del tipo: Hashtable Tipi usati: Entry, boolean, Key	Dominio: insieme di elem. $T=\{a_1, \dots, a_n\}$ di tipo Entry
newHashtable() → Hashtable	newHashtable() → t • Post: $t = \{\}$
insertHash(Hashtable, Entry) → Hashtable	insertHash(t, e) → t' • Post: $t = \{a_1, a_2, \dots, a_n\}$, $t' = \{a_1, a_2, \dots, e, \dots, a_n\}$
searchHash(Hashtable, Key) → Entry	searchHash(t, k) → e • Pre: $t = \{a_1, a_2, \dots, a_n\}$ $n > 0$ • Post: $e = a_i$ con $1 \leq i \leq n$ se $a_i(k) = k$
deleteHash(Hashtable, Key) → Hashtable	deleteHash(t, k) → t' • Pre: $t = \{a_1, a_2, \dots, a_n\}$ $n > 0$, $a_i(k) = k$, $1 \leq i \leq n$ • Post: $t' = \{a_1, a_2, \dots, a_{i-1}, a_{i+1}, \dots, a_n\}$

key.h

```
typedef void *Key;
int equals(Key, Key);
int hashValue(Key, int);
Key inputKey();
void outputKey(Key);
```

key-string.c

```
#include "key.h"
#define SIZE 30

int equals(Key k1, Key k2){
    char *chiave1, *chiave2;
    chiave1=k1;
    chiave2=k2;
    return (strcmp(chiave1, chiave2)==0);
}

int hashValue(Key k, int size){
    int h=0, a=128;
    char *v=k;
    for(;*v]!='\0'; v++)
        h=(h*a + *v)%size;
    return h;
}
```

```
Key inputKey(){
    char *k=malloc(SIZE*sizeof(char));
    scanf("%s", k);
    return k;
}

void outputKey(Key k){
    char *v=k;
    printf("%s ", v);
}
```

entry.h

```
#include "key.h"
#include "item.h"
typedef struct entry *Entry;

Entry newEntry(Key, Item);
Key getKey(Entry);
Item getValue(Entry);
```

entry.c

```
#include "entry.h"
#include "key.h"
#include "item.h"

struct entry{
    Key key;
    Item value;
};

Entry newEntry(Key k, Item value){
    Entry e = malloc(sizeof(struct entry));
    e->key=k;
    e->value=value;
    return e;
}
```

```
Key getKey(Entry e){
    if(e==NULL)    return NULL;
    return (e->key);
}
```

```
Item getValue(Entry e){
    if(e==NULL)    return NULL;
    return (e->value);
}
```

hashtable.h

```
#include "entry.h"
#include "key.h"
typedef struct hashtable *HashTable;

HashTable newHashtable(int);
int insertHash(HashTable, Entry);
Entry searchHash(HashTable, Key);
Entry deleteHash(HashTable, Key);
```

hashtable.c

```
#include "hashtable.h"
#include "list.h"
#include "entry.h"
#include "key.h"
#define N 20
struct hashtable{
    int size;
    List *entries;
};

HashTable newHashtable(int size){
    HashTable h = malloc(sizeof(struct hashtable));
    h->size=size;
    h->entries= malloc(size*sizeof(List));
    for(int i=0; i<size; i++)
        h->entries[i]= newList();
    return h;
}
```

```
int insertHash(HashTable h, Entry e){
    if(h==NULL)    return 0;
    Key k = getKey(e);
    int index= hashValue (k, h->size);
    addHead(h->entries[index], e);
    return 1;
}
```

```
Entry searchHash(HashTable h, Key k){
    if(h==NULL)    return NULL;
    int index= hashValue (k, h->size);
    Entry e = newEntry(k, NULL);
    int pos;
    return searchList(h->entries[index], e, &pos);
}
```

```
Entry deleteHash(HashTable h, Key k){
    if(h==NULL)    return NULL;
    int index= hashValue (k, h->size);
    Entry e = newEntry(k, NULL);
    return removeListItem (h->entries[index], e);
}
```

item-entry.c

```
#include "item.h"
#include "key.h"
#include "entry.h"

int cmpItem(Item item1, Item item2){
    Entry val1, val2;
    val1 = item1;
    val2 = item2;
    return !equals(getKey(val1), getKey(val2));
}
```