

Programmazione e Strutture Dati

Sviluppo di Programmi

1) Analisi

L'analisi specifica cosa fa il programma. Si comprende dapprima il problema e si individuano **i dati in ingresso e i loro vincoli** e poi **i dati in uscita e i loro vincoli**.

I *vincoli* sono:

- **Precondizione**: ovvero una condizione definita sui dati in ingresso che deve essere soddisfatta affinché la funzione sia applicabile.
- **Postcondizione**: ovvero una condizione definita sui dati di uscita e dati in ingresso e che deve essere soddisfatta al termine dell'esecuzione del programma. In sintesi, la *postcondizione* definisce cosa sono i dati di output in funzione quelli in input.

Dizionario dei Dati

Buona norma utilizzare un *dizionario dei dati* da arricchire durante le varie fasi del ciclo di vita. Non è altro che una tabella il cui schema comprende: un **Identificatore**, un **Tipo**, una **Descrizione**.

La descrizione serve a specificare meglio l'identificatore e a descrivere il contesto in cui il dato viene usato.

ESEMPIO:

- | | |
|----------------------------|--|
| ● Dati in 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 $\forall i \in [0, n - 2], s1_i \leq s1_{i+1}$ |

Identificatore	Tipo	Descrizione
s	Sequenza	Sequenza di interi in input
s1	Sequenza	Sequenza di interi di output
n	Intero	Numero di elementi nella sequenza
i	intero	Indice per individuare gli elementi nella sequenza

2) Progettazione

Si definisce come il programma effettua la trasformazione specificata, si progetta l'algoritmo per raffinamenti successivi (*stepwise refinement*): si definiscono gli step e si scomponete il problema in funzioni più semplici.

3) Implementazione

Codifichiamo la soluzione ed effettuiamo un test e un debugging.

4) Esecuzione

Applicazione su dati reali.

Preprocessore

Le direttive sono gestite dal **preprocessore**, un software che manipola i programmi C immediatamente prima della compilazione.

Il comportamento del preprocessore è controllato dalle **direttive di preprocessamento**: dei comandi che iniziano con il carattere #.

La direttiva `#define` definisce una macro. Il preprocessore risponde alla direttiva `#define` memorizzando il nome della macro assieme alla sua definizione.

La direttiva `#include` dice al preprocessore di aprire un particolare file e di “includere” il suo contenuto come parte del file che deve essere compilato.

Algoritmi di Ordinamento

Il problema dell’ordinamento consiste nell’elencare gli elementi di un insieme secondo una sequenza stabilita da una relazione d’ordine, ad esempio:

- Ordinare una breve sequenza di numeri;
- Mettere un elenco di nomi in ordine alfabetico;
- Ordinare i record degli studenti Unisa secondo la data di nascita. In questo caso abbiamo una **chiave**, ovvero la data di nascita. La **chiave** può essere un singolo campo o la combinazione di più campi.

Proprietà Algoritmi di ordinamento

Possiamo classificare gli algoritmi di ordinamento in base ad alcune proprietà. Un algoritmo si dice:

- **Stabile**: due elementi con la stessa chiave mantengono lo stesso ordine con cui si presentavano prima dell’ordinamento
- **In loco**: in ogni dato istante questo algoritmo alloca un numero costante di variabili, oltre all’array da ordinare.
- **Adattivo**: il numero di operazioni effettuate dipende dall’input e non dalla sua dimensione.
- **Interno**: quando i dati sono contenuti nella memoria RAM.
- **Esterno**: quando i dati sono presenti su disco o su nastro.

La differenza principale tra i due tipi di ordinamento (**Interno vs Esterno**) sta nel fatto che mentre nel primo è possibile accedere direttamente a un record, nel secondo i record devono essere indirizzati in modo sequenziale o al più per grandi blocchi.

Classificazione degli Algoritmi

Andremo ad analizzare vari algoritmi di ordinamento. Questi ordinano per **confronti** cioè prendono due variabili o due record ed effettuano un confronto, ovvero se due numeri o due stringhe sono l’una più grande dell’altra.

Gli algoritmi si classificano in **semplici** e **avanzati**. La differenza risiede nel numero di operazioni che questi algoritmi richiedono per portare a termine tali operazioni.

Gli **algoritmi semplici** effettuano un numero di operazioni quadratico rispetto alla taglia dell’input: $O(n^2)$. Questi algoritmi sono: Selection sort, Insertion sort, Bubble sort.

Gli **algoritmi avanzati** richiedono un tempo inferiore e sono più efficienti come ad esempio il Merge sort che richiede un numero di operazioni: $O(n \log n)$.

Mentre il Quick sort, nel caso medio, richiede un numero di operazioni sempre di $O(n \log n)$ e nel caso peggiore un numero di operazione quadratico.

Selection Sort

Il **selection sort** è un algoritmo di ordinamento, è di tipo non adattivo, ossia il suo tempo di esecuzione non dipende dall'input ma dalla dimensione dell'array.

L'algoritmo *seleziona* di volta in volta il numero minore nella sequenza di partenza e lo sposta nella sequenza ordinata; la sequenza viene suddivisa in due parti: la sotto sequenza ordinata, che occupa le prime posizioni dell'array, e la sotto sequenza da ordinare, che costituisce la parte restante dell'array.

Dovendo ordinare un array A di lunghezza n , si fa scorrere l'indice i da 1 a $n-1$ ripetendo i seguenti passi:

1. si cerca il più piccolo elemento della sotto sequenza $A[i..n]$;
2. si scambia questo elemento con l'elemento i -esimo.

Per quanto riguarda le prestazioni dell'algoritmo, il **Selection Sort** effettua $N(N - 1)/2$ confronti, dunque la complessità di tale algoritmo è dell'ordine di $\Theta(n^2)$.

L'algoritmo risulta poco efficiente, per la sua complessità esponenziale, ma è utile per array di piccole dimensioni.

Insertion Sort

L'**insertion Sort** è un algoritmo di ordinamento che utilizza lo stesso metodo che un essere umano usa per ordinare le sue carte in mano. Fa un tipo di ordinamento **in loco**, ovvero non crea un array di appoggio, risparmiando in questo modo memoria.

L'**insertion Sort** è un algoritmo molto semplice da utilizzare ma non ha una grande efficienza, se non in caso di array iniziali parzialmente ordinati.

L'algoritmo utilizza due indici: il primo punta al secondo elemento dell'array, il secondo inizia dal primo. Se il primo elemento è maggiore del secondo, i due valori vengono scambiati. Poi il primo indice avanza di una posizione e il secondo indice riparte dall'elemento precedente quello puntato dal primo. Se l'elemento puntato dal secondo indice non è maggiore di quello a cui punta il primo indice, il secondo indice indietreggia; e così via, finché si trova nel punto in cui il valore del primo indice deve essere *inserito* (da qui *insertion*). L'algoritmo così tende a spostare man mano gli elementi maggiori verso destra.

L'algoritmo utilizza una quantità minima di memoria indispensabile ma è lento con array di grosse dimensioni, invece con array di piccole dimensioni è l'algoritmo di ordinamento più veloce.

Effettua in media $\frac{N^2}{4}$ confronti ed altrettanti spostamenti (mezzi scambi), che possono diventare il doppio nel caso peggiore.

Bubble Sort

Il **Bubble sort** è un **algoritmo iterativo**, ossia basato sulla ripetizione di un procedimento fondamentale.

Lavora comparando ogni coppia di elementi adiacenti e invertendone le posizioni se sono nell'ordine sbagliato. L'algoritmo continua nuovamente a ri-eseguire questi passaggi per tutta la lista finché non vengono più eseguiti scambi, situazione che indica che la lista è ordinata.

L'algoritmo deve il suo nome al modo in cui gli elementi vengono ordinati in una lista: quelli più piccoli "*risalgono*" verso un'estremità della lista, mentre quelli più grandi "*affondano*" verso l'estremità opposta della lista, come le *bolle* in un bicchiere di spumante.

Esso ha comunque una complessità computazionale dell'ordine di $O(n^2)$ confronti, con n elementi da ordinare.

Il **Bubble sort** non è adattivo ma lo può diventare riscrivendo il codice. Infatti, se nella visita di un array non è stato effettuato alcuno scambio allora sappiamo che l'array è ordinato.

Merge Sort

Il **merge sort** è un algoritmo di ordinamento inventato da Von Neumann nel 1945.

È un algoritmo di ordinamento più complesso ma molto più efficiente degli altri visti in precedenza ([selection sort](#) e [insertion sort](#)), soprattutto con vettori di grandi dimensioni.

Sfrutta la tecnica *divide et impera*, ovvero la suddivisione del problema in sotto-problemi della stessa natura di dimensione a mano a mano sempre più piccola, li risolve ricorsivamente e infine combina le soluzioni parziali per ottenere la soluzione al problema di partenza.

Quindi si divide **ricorsivamente** il vettore in due parti da ordinare separatamente. Successivamente si fondono le due parti per ottenere un array ordinato globalmente. Il merge sort utilizza inoltre un array di appoggio.

L'algoritmo sia nel caso medio che in quello pessimo ha una complessità di $\Theta(n \log n)$. La complessità della funzione **merge** è lineare $\Theta(n)$.

Impera: Merge sort su vettore SX

Merge sort su vettore DX

Condizione di terminazione con 1 ($p=r$) o 0 ($p>r$)

Combina: usa merge per fondere i due sottovettori ordinati in un vettore ordinato. Si estrae ripetutamente il minimo die due sottovettori e lo si pone nella sequenza in uscita.

Quick Sort

Il **quick sort** è un algoritmo che viene utilizzato per ordinare i dati in un vettore (array). Esso utilizza, così come il [merge sort](#), il metodo **divide et impera**. Cioè è basato su questa tecnica:

- **Divide:** Si suddivide il problema in sotto-problemi più piccoli;
- **Impera:** si risolvono i problemi in maniera ricorsiva (ovvero un algoritmo che richiama sé stesso);
- **Combina:** al fine di ottenere il risultato finale si combina l'output ottenuto dalle precedenti chiamate ricorsive.

Il *Quick Sort* è infatti un algoritmo **ricorsivo** che ha, generalmente, prestazioni migliori tra quelli basati su confronto.

In questo algoritmo la ricorsione viene fatta non dividendo il vettore in base agli indici ma in base al suo contenuto.

Prima di risolvere l'algoritmo del *quick sort* in C, studiamo l'idea di base:

- Si sceglie l'elemento centrale del vettore (detto **pivot**) e lo si memorizza in una variabile;
- a sinistra si mettono gli elementi \geq pivot;
- a destra si mettono gli elementi \leq pivot;
- si ordina ricorsivamente la parte destra e la parte sinistra.

La procedura che divide in due parti il vettore si chiama *partition*. Chiaramente la scelta del pivot non è legata necessariamente al primo elemento, potrebbe anche essere l'ultimo o il mediano oppure essere un elemento random.

Astrazione e Modularizzazione

File sorgente – File header

Alcuni programmi C sono molto grandi e non sono sufficienti per essere posti in un unico file sorgente. I **file sorgente** contengono le definizioni delle funzioni e delle variabili esterne mentre i **file header** contengono le informazioni che devono essere condivise tra i file.

Un programma può essere suddiviso su più file sorgente che hanno estensione **.c**. Un file sorgente deve contenere una funzione chiamata *main* che fa da punto di partenza per il programma.

Attraverso la direttiva `#include` possiamo condividere le informazioni dei diversi file sorgente. Infatti, tale direttiva dice al processore di aprire uno specifico file e di inserire il suo contenuto all'interno del file corrente. I file header hanno estensione **.h**.

- **Compilazione:** ogni file sorgente presente nel programma deve essere compilato separatamente. Per ogni file sorgente, il compilatore genera un file contenente del codice oggetto. Questi file, conosciuti come **file oggetto**, hanno estensione **.o** in *UNIX* **.obj** in *Windows*.
- **Linking:** il linker combina i file oggetto creati nel passo precedente al fine di produrre un file eseguibile. Il linker è anche responsabile della risoluzione dei riferimenti esterni lasciati dal compilatore (un riferimento esterno si verifica quando una funzione presente in un file invoca una funzione definita in un altro file oppure accede a una variabile definita in un altro file).

L'opzione **-o** specifica che vogliamo un file eseguibile chiamato *nome_file*.

L'opzione **-c** dice al compilatore di compilare *nome_file* in un file oggetto, senza effettuare il linking.

Makefile

Per facilitare il building di grandi programmi, dall'ambiente UNIX ha avuto origine il **makefile**. Il comando **make** compila e collega i vari moduli che compongono il progetto. Il **makefile** è costituito da specifiche del tipo:

```
target_file: dipendenze_da_file  
    comandi
```

Per eseguirlo faremo **make target_file** dal *Prompt dei comandi*. L'ordine delle specifiche non è importante ma è buona norma inserire come prima specifica quella per la costruzione del programma eseguibile, in questo caso per lanciare il processo basta digitare il comando **make**. Ogni comando presente nel *makefile* deve essere preceduto da un carattere *tab*.

Modularità

Grazie a CPU più veloci e memorie più capaci hanno reso possibile la scrittura di grandi programmi che sarebbero stati impossibili fino a pochi anni fa. La popolarità delle interfacce grafiche ha incrementato parecchio la lunghezza media dei programmi. La scrittura di programmi di grandi dimensioni è abbastanza diversa da quella per i piccoli programmi.

Un programma di grandi dimensioni richiede più attenzione allo stile, dato che vi lavoreranno molte persone, un'attenta documentazione e la pianificazione della manutenzione, dal momento che probabilmente dovrà essere modificato più volte.

Spesso, quando si progetta un programma in C è utile vederlo come costituito da un certo numero di **moduli indipendenti**.

La **modularità** è una tecnica che ci permette di suddividere un progetto software in modo tale da gestire la complessità del codice.

Il modulo è costituito da una collezione di servizi, alcuni dei quali devono essere resi disponibili alle altre parti del programma, i cosiddetti **client**. Il modulo ha importanti proprietà:

- **Elevata coesione:** le varie funzionalità messe a disposizione da un singolo modulo sono strettamente correlate
- **Indipendenza:** i moduli si sviluppano separatamente dal resto del programma, con compilazione e testing separati.
- I moduli hanno modalità di interazioni ben definite, possono essere implementati (funzioni, procedure, classi, package).

Il modulo è costituito da:

- 1) Un'**interfaccia** (cioè il prototipo della funzione) che definisce le risorse ed i servizi (astrazione) messi a disposizione dei "clienti" (programma o altri moduli).
- 2) Un **corpo** che comprende l'**implementazione** delle risorse, incluso il codice sorgente.

Un modulo può usare anche altri moduli, e può essere compilato indipendentemente dal programma che lo usa.

In C non esiste un apposito costrutto per realizzare un modulo, di solito un modulo coincide con un file. Per esportare le risorse definite in un modulo, il C permette un particolare tipo di file chiamato **header file** con estensione **.h** (**header file** rappresenta l'interfaccia del modulo verso altri moduli e nel **source file** abbiamo il codice del modulo cioè l'implementazione). Per accedere alle risorse messe a disposizione da un modulo bisogna includere il suo **header file**.

Per le librerie predefinite del C usiamo `#include <stdio.h>`, per i moduli definiti dall'utente usiamo `#include "modulo.h"`.

I **commenti**, relativi alla specifica di una funzione, possono essere inseriti nell'**header file** prima del prototipo della funzione e sono utili per documentare il codice. 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. Tutto ciò serve da documentazione per chi dovrà modificare la funzione.

Suddividere un programma in moduli presenta diversi vantaggi a partire dall'**astrazione**, la **riusabilità** e la **manutenibilità**.

Astrazione

L'astrazione è un procedimento mentale che consente di evidenziare le caratteristiche principali di un problema ignorando i dettagli.

Abbiamo due tipi di astrazione: **astrazione funzionale e procedurale, astrazione sui dati**.

- **L'astrazione funzionale e procedurale** abbiamo che una funzionalità di un programma è scelta ad un sottoprogramma (funzione o procedura). E' definita ed usabile indipendentemente dall'algoritmo che la implementa.
- **L'astrazione sui dati** abbiamo un dato o un tipo di dato definito insieme alle operazioni che possono essere fatte su di esso, infatti, sia le operazioni che il dato sono usabili a prescindere dall' implementazione.

Riuso del codice

Il **riuso del codice** è una pratica estremamente comune nella programmazione, infatti ci permette di richiamare o invocare parti di codice precedentemente già scritte ogni volta che lo riteniamo necessario in modo tale da ottimizzare i tempi e rendere un codice più pulito e leggibile.

Information hiding

Spesso un modulo ben progettato mantiene alcune informazioni segrete nei confronti dei suoi client.

Nascondere alcune informazioni ai client di un modulo è conosciuto come: **information hiding**.

Una pratica che consente di nascondere il funzionamento interno (deciso in fase di progetto) di una parte del programma. Il vantaggio è quello di non poter modificare l'interno del codice da parte di un altro utente e ci permette di effettuare una correzione degli errori facilitata perché se un errore è presente in un modulo, questo può essere corretto modificando soltanto quel modulo.

Per garantire l'*information hiding* non bisogna utilizzare variabili globali ne funzioni di servizio nascoste.

Testing e debugging

Il **testing** e il **debugging** sono una fase dell'**implementazione** del software. Prima di questa ci sarà stata l'analisi e la progettazione.

Il **testing** è un'attività che consiste nell'esercitare il programma con dati di test per verificare che il suo comportamento sia conforme a quello atteso definito nella specifica.

L'**Oracolo** è l'output atteso cioè quello che ci si aspetta il programma produca, mentre un **malfunzionamento** è un comportamento diverso da quello atteso. L'obiettivo del testing è individuare i malfunzionamenti.

Malfunzionamenti

Un **malfunzionamento** di un programma è causato da un difetto nel codice chiamato *errore* o *bug*.

L'errore può essere introdotto in fase di analisi e specifica, di progettazione o di codifica.

Il **debugging** consiste nell'individuazione e correzione del difetto. 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.

Testing

Testare un programma con tutti i possibili dati di test è impraticabile quindi l'obiettivo è di individuare alcune classi di dati di test, selezionare un caso di test da ogni classe ed evitare casi di test ridondanti. L'insieme dei casi di test formerà una **Test suite**.

File

Per attuare la fase di testing è comodo utilizzare i **file**.

Nel linguaggio C il termine **stream** indica una qualsiasi sorgente di input o una qualsiasi destinazione di output. In un programma C l'accesso ad uno **stream** avviene per mezzo di un puntatore a file, che è di tipo FILE * (il tipo FILE viene dichiarato all'interno di `<stdio.h>`).

File *`fopen(const char * restrict filename, const char * restrict mode);`

Aprire un file per usarlo come uno stream richiede una chiamata alla funzione **fopen**. Il primo argomento della funzione è una stringa contenente il nome del file che deve essere aperto (il "nome del file" può contenere informazioni riguardante la sua posizione, come il drive o il percorso). Il secondo argomento è una "stringa di modalità" che specifica quali operazioni abbiamo intenzione di compiere sul file.

I programmati Windows devono fare attenzione quando il nome del file in una chiamata `fopen` include il carattere \ perché il C tratta quest'ultimo come una sequenza di escape.

Ci sono due modi per evitare il problema. Il primo consiste nell'utilizzare \\ al posto di \, il secondo consiste nell'utilizzare / al posto del carattere \.

La funzione *fopen* restituisce un puntatore a file che il programma può salvare all'interno di una variabile per utilizzarlo per effettuare operazioni sui file. Quando non può aprire un file, *fopen* restituisce un **puntatore nullo**.

Modalità di apertura per i file testuali:

Stringa	Significato
“r”	Aprire un file in lettura
“w”	Aprire il file in scrittura (non è necessario che il file esista)
“a”	Aprire il file in accodamento (non è necessario che il file esista)
“r+”	Aprire il file in lettura e scrittura, comincia dall'inizio del file
“w+”	Aprire il file in lettura e scrittura (tronca il file se esiste)
“a+”	Aprire il file in lettura e scrittura (accoda se il file esiste)

Modalità di apertura per i file binari:

Stringa	Significato
“rb”	Aprire un file in lettura
“wb”	Aprire il file in scrittura (non è necessario che il file esista)
“ab”	Aprire il file in accodamento (non è necessario che il file esista)
“r+b” oppure “rb+”	Aprire il file in lettura e scrittura, comincia dall'inizio del file
“w+b” oppure “wb+”	Aprire il file in lettura e scrittura (tronca il file se esiste)
“a+b” oppure “ab+”	Aprire il file in lettura e scrittura (accoda se il file esiste)

La funzione *fclose* permette ad un programma di chiudere un file che non viene più utilizzato.

L'argomento di *fclose* deve essere un puntatore a file ottenuto da una chiamata alla *fopen* o alla *fropen*.

La funzione *fclose* restituisce 0 se il file è stato chiuso con successo, altrimenti restituisce il codice di errore *EOF*.

Puntatori e allocazione dinamica

I **puntatori** sono una delle caratteristiche più potenti del C, ma difficile da padroneggiare. Si usano per simulare la chiamata per riferimento e sono in stretta correlazione con vettori e stringhe.

Noi sappiamo che ogni *byte* possiede un indirizzo che lo distingue dagli altri presenti in memoria. Se nella memoria ci sono n byte, allora possiamo pensare che gli indirizzi vadano da 0 a $n - 1$ byte.

Il C ci permette di memorizzare gli indirizzi delle variabili all'interno di **variabili puntatore**.

Queste variabili puntatore contengono gli indirizzi, di una semplice variabile, come valore. Quando memorizziamo l'indirizzo di una variabile i in una variabile puntatore p , diciamo che p “*punta*” a i .

L'operatore indirizzo

Per trovare l'indirizzo di una variabile useremo l'operatore **&** (*Indirizzo*). Se x è una variabile, allora $\&x$ è il suo indirizzo di memoria. Inoltre, l'operatore **&** deve essere applicato ad una **variabile**, non può essere applicato a costanti, espressioni o a variabili dichiarate con la specifica classe di memoria *register*.

Operatore asterisco

Una volta che una variabile puntatore punta ad un oggetto, possiamo usare l'operatore ***** per accedere a quello che è il contenuto dell'oggetto stesso. Effettuiamo così il **deriferimento**, ossia il riferimento a un valore per mezzo di un puntatore.

Possiamo dire che ***** e **&** sono l'uno l'inverso dell'altro. Se applicati insieme si annullano.

Allocazione dinamica della memoria

Utilizzando l'allocazione di memoria dinamica della memoria, un programma può ottenere dei blocchi di memoria runtime. Le strutture allocate dinamicamente sono importanti nella programmazione C dal momento che possono essere collegate per formare liste, alberi, e altre strutture dati.

Per allocare dinamicamente la memoria abbiamo bisogno di invocare una delle tre funzioni che si trovano nell'*header <stdlib.h>*

- *Malloc*: alloca un blocco di memoria ma non lo inizializza;
- *Calloc*: alloca un blocco di memoria e lo azzera;
- *Realloc*: ridimensiona un blocco di memoria allocato precedentemente.

Delle tre, la funzione più utilizzata è la *malloc*. Quando chiamiamo una funzione di allocazione della memoria per richiederne un blocco, questa non può conoscere il tipo di dati che stiamo pensando di inserire in un tale blocco e quindi non può restituire un puntatore a un tipo ordinario come *int* o *char*. Al suo posto la funzione restituisce un valore di tipo *void*.

Puntatori nulli

Quando una funzione per la memoria viene invocata, c'è sempre la possibilità che questa non sia in grado di allocare un blocco di memoria sufficientemente grande a soddisfare la nostra richiesta. Se questo dovesse succedere la funzione restituirebbe un puntatore **NULL**.

Nel C i puntatori vengono considerati *true* o *false* secondo lo stesso criterio utilizzato per i numeri.

All'interno delle condizioni, tutti i puntatori non **NULL** vengono considerati come veri. Solo i puntatori **NULL** vengono considerati falsi.

Funzione malloc

La funzione *malloc* possiede il seguente prototipo: `void *malloc(size_t size);`

La funzione alloca un blocco di *size* byte e restituisce un puntatore a quest'ultimo. *Size* è *unsigned*, ovvero un intero senza segno.

Funzione *calloc*

La funzione *calloc* che possiede il seguente prototipo: `void *calloc (size_t nmemb, size_t size);`

La funzione alloca dello spazio per un vettore *nmemb* elementi, ognuno dei quali di *size_byte*. La funzione restituisce un puntatore nullo se lo spazio richiesto non è disponibile. Dopo avere alloggiato la memoria, la funzione *calloc* la inizializza impostando tutti i bit a 0.

Funzione *realloc*

La funzione *realloc* può ridimensionare il vettore per adeguarsi meglio ai nostri bisogni. Il prototipo della *realloc* è il seguente: `void *realloc(void *ptr, size_t size);`

Quando la *realloc* viene chiamata, *ptr* deve puntare al blocco di memoria ottenuto dalle chiamate precedenti alle funzioni *malloc*, *calloc*, *realloc*. Il parametro *size* rappresenta la nuova dimensione del blocco, la quale può essere più grande o più piccola della dimensione originale. Bisogna assicurarsi che è un puntatore passato alla funzione *realloc* provenga da una chiamata precedente alle funzioni *malloc*, *calloc*, *realloc*, altrimenti se non fosse così la chiamata alla *realloc* provocherebbe un comportamento indefinito.

Deallocare la memoria

La *malloc* e le altre funzioni di allocazione della memoria ottengono dei blocchi da un'area di memoria conosciuta come **heap**. Chiamare queste funzioni troppo spesso può esaurire lo **heap**, determinando la restituzione di un puntatore nullo da parte delle funzioni. A peggiorare le cose un programma può allocare dei blocchi di memoria e poi perdere traccia di essi, sprecando così dello spazio.

Un blocco di memoria che non sia più accessibile da un programma viene detto **garbage (spazzatura)**. Un programma che ti lasci indietro della spazzatura si dice che è effetto da **memory leak**. Alcuni linguaggi forniscono un **garbage collector** che trova automaticamente i blocchi spazzatura e li ricicla. Il C, tuttavia, non possiede questa funzionalità, ha bisogno quindi di chiamare la funzione **free** per rilasciare la memoria non necessaria.

Funzione *free*

La funzione *free* ha il seguente prototipo: `void free(void *ptr);`

Per utilizzare la funzione *free* dobbiamo passarle un puntatore a un blocco di memoria che non è più necessario. Chiamando la funzione *free*, questa rilascia il blocco di memoria che è puntata da *p*. Questo blocco adesso è disponibile per essere riutilizzato nelle successive chiamate alla *malloc* e altre funzioni di allocazione di memoria.

Tipi di variabili in C

- **Globali:** questo tipo di variabili sono dichiarate esternamente alle funzioni e sono visibili a tutte le funzioni la cui definizione segue la dichiarazione della variabile nel file sorgente.
Sono dette statiche, perché la loro allocazione in memoria avviene all'atto del caricamento del programma (e la loro deallocazione al termine del programma).
Le variabili globali definite in un file F1 possono essere usate in un file F2 dichiarandole **extern** in F2: `extern int n;` (con la dichiarazione **extern** non si definisce la variabile, non si alloca memoria).
Dichiarando le variabili globali **static**, queste saranno private al file in cui sono dichiarate: `static int n;`
- **Locali:** sono dichiarate e visibili solo all'interno di una funzione. Le variabili locali possono essere anche **static** però la loro classe di allocazione cambia, infatti da variabile locale si trasforma in **variabile statica** (ossia globale).
Cambia quindi la durata della variabile, non lo **scope** che rimane confinato alla funzione in cui la variabile è dichiarata. Come effetto, il valore di una variabile "sopravvive" all'esecuzione della funzione in cui è stato definito.
- **Automatiche:** sono dichiarate in blocchi interni alle funzioni. Queste variabili vengono allocate in memoria a tempo di esecuzione e deallocate al termine del blocco.

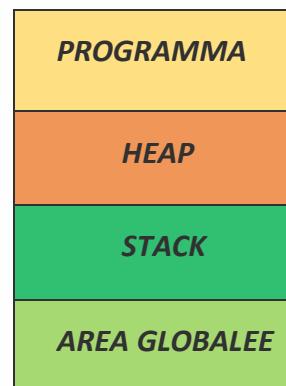
Ci sono tre aspetti importanti di una dichiarazione:

- 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:** indica il periodo durante il quale una variabile è allocata in memoria.

Durata: tre aree di allocazione

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

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



Dichiarare **static** le funzioni serve a realizzare *l'information hiding*, rendendo le funzioni private al file in cui sono dichiarate, ossia ne modificano lo scope.

Abstract Data Types (ADT)

Per descrivere la natura dei dati rappresentabili in un programma e le operazioni per manipolarli, si usa il concetto di **tipo di dato**. Ad ogni tipo di dato è assegnato un operatore. Nel momento in cui il programmatore crea un nuovo tipo di dato, a questo devono essere associati degli operatori. Un tipo di dato astratto è caratterizzato matematicamente da:

- Un **Dominio**: insieme dei possibili valori come numeri interi, numeri razionali, ecc.
- **Operazioni**: funzioni che operano sugli elementi del dominio
- **Letterali**: costanti matematiche

L'associazione **tipo-dato** limita gli errori perché protegge il programmatore da associazioni illogiche tra dati e operatori (non presente a livello macchina).

I tipi di dato si possono dividere in 3 classi:

- 1) **TD Primitivi (Elementari)**: basati direttamente sulla rappresentazione hardware (*int, char, real, boolean*)
- 2) **TD aggregati**: array, strutture, enumerazioni, unioni
- 3) **Puntatori**

Gli ADT sono tipi di dati che estendono i dati esistenti, distinguendo **specificità** e **implementazione**.

Con la **specificità**, definiamo il tipo di dato e gli diamo una definizione dell'insieme degli operatori.

Con l'**implementazione** abbiamo la codifica di quanto definito nella specifica, usando primitive e costrutti di un linguaggio di programmazione. Questa è spesso nascosta al programmatore, seguendo il principio dell'**incapsulamento** (*information hiding*).

Strutture

Le strutture sono collezioni di variabili aggregati da un unico nome. Possono contenere variabili di tipi di dato diversi. Sono utilizzate per definire record da memorizzare nei file e sono anche combinate con i puntatori, possono servire a creare tipi di dati strutturati come liste a puntatori, pile code ed alberi.

I membri di una struttura hanno dei nomi e per selezionare un particolare membro dobbiamo specificare il suo nome e non la sua posizione.

- **Struct** introduce la definizione della struttura NOMETIPO;
- **NomeTipo** è il nome della struttura (etichetta) ed è usata per dichiarare variabili del tipo della struttura;
- **NomeTipo** può contenere diversi tipi al suo interno.

I nomi dichiarati all'interno della struttura non andranno in conflitto con altri nomi (uguali) dichiarati nel programma. Nella terminologia del C, si dice che ogni struttura ha uno **spazio dei nomi** per i suoi membri.

I membri di una struttura vengono memorizzati nell'ordine in cui appaiono nella dichiarazione. Per accedere ad un membro di una struttura scriviamo il nome della struttura prima, poi un punto e poi il nome del membro.

Il punto usato per specificare un membro della struttura è un operatore ed ha precedenza su tutti gli altri operatori.

Le funzioni possono avere strutture come argomenti e come valori di ritorno. Una funzione con una struttura come argomento è la seguente:

Passare una struttura ad una funzione e restituire una struttura come valore di ritorno richiede una copia di tutti i membri della struttura. Per evitare troppe operazioni di copia è preferibile passare o restituire un puntatore alla struttura.

Pseudo – generics in C

Generics è uno strumento che permette la definizione di un tipo parametrizzato, che viene esplicitato successivamente in fase di compilazione (o *linkaggio*) secondo le necessità. L'obiettivo è di implementare algoritmi e ADT che siano in grado di funzionare, di volta in volta, con il tipo di dati desiderato.

ADT Liste

Una lista non è altro che una collezione di elementi omogenei, ma, a differenza dell'array, occupa in memoria una posizione qualsiasi, che tra l'altro può cambiare dinamicamente durante l'utilizzo della lista stessa; inoltre la sua dimensione non è nota a priori e può variare nel tempo (l'opposto dell'array, in cui la dimensione è ben nota e non è modificabile). Una lista è chiamata anche **FIFO (First-in-first-out)** cioè il primo elemento ad inserire sarà il primo ad essere eliminato. Una lista può contenere uno o più campi contenenti informazioni, e, necessariamente, deve contenere un puntatore per mezzo del quale è legato all'elemento successivo.

La lista base ha un solo campo informazione ed un puntatore, come mostrato di seguito:

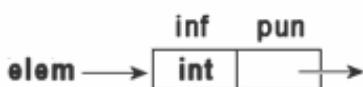
```
struct elemento {  
    int inf;  
    struct elemento *pun;  
}
```

Una particolarità delle liste (a differenza, ad esempio, degli array) è che sono costituite da due funzioni del C, le strutture ed i puntatori, quindi non sono un tipo di dato nuovo.

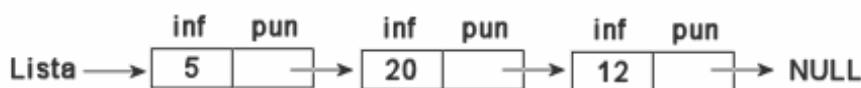
Una lista può contenere uno o più campi di informazione, che possono essere di tipo *int*, *char*, *float*, ecc...; mentre deve esserci sempre un campo che punta ad un altro elemento della lista, che è *NULL* se non ci sono altri elementi, mentre risulta essere una **struttura elemento** nel caso vi sia un successore. Per dichiarare una lista, basta scrivere:

- *struct elemento *lista;*

che altro non è che un puntatore ad una struttura elemento, e come tale potrebbe essere inizializzata anche ad un valore *NULL*, che identificherebbe una **lista vuota**. In questo modo definiamo, quindi, una **lista lineare**, ovvero una lista che deve essere visitata (o scandita) in ordine, cioè dal primo elemento fino all'ultimo, che viene identificato perché punta a *NULL*. Da ricordare che una lista risulta essere sequenziale, ogni volta che devo aggiungere un elemento alla lista, devo allocare la memoria relativa, connetterlo all'ultimo elemento ed inserirvi l'informazione.

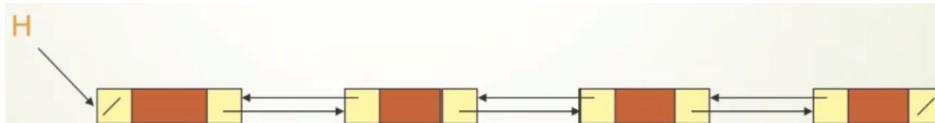


una lista risulta essere come segue:



Liste circolari: permettono l'attraversamento della lista partendo da un nodo qualsiasi; il vantaggio è di iterare tutta la lista partendo da un nodo qualsiasi.

Liste a collegamento doppio: ogni nodo ma ha due puntatori uno a quello precedente e uno al nodo successivo. Il campo precedente al primo nodo e il campo successivo dell'ultimo nodo sono dei puntatori nulli. Il vantaggio è che possiamo effettuare delle cancellazioni e inserimenti in tempo costante quando si ha il puntatore a nodo. Lo svantaggio è quello di rendere il codice più complicato a causa della necessità di aggiornare due puntatori.



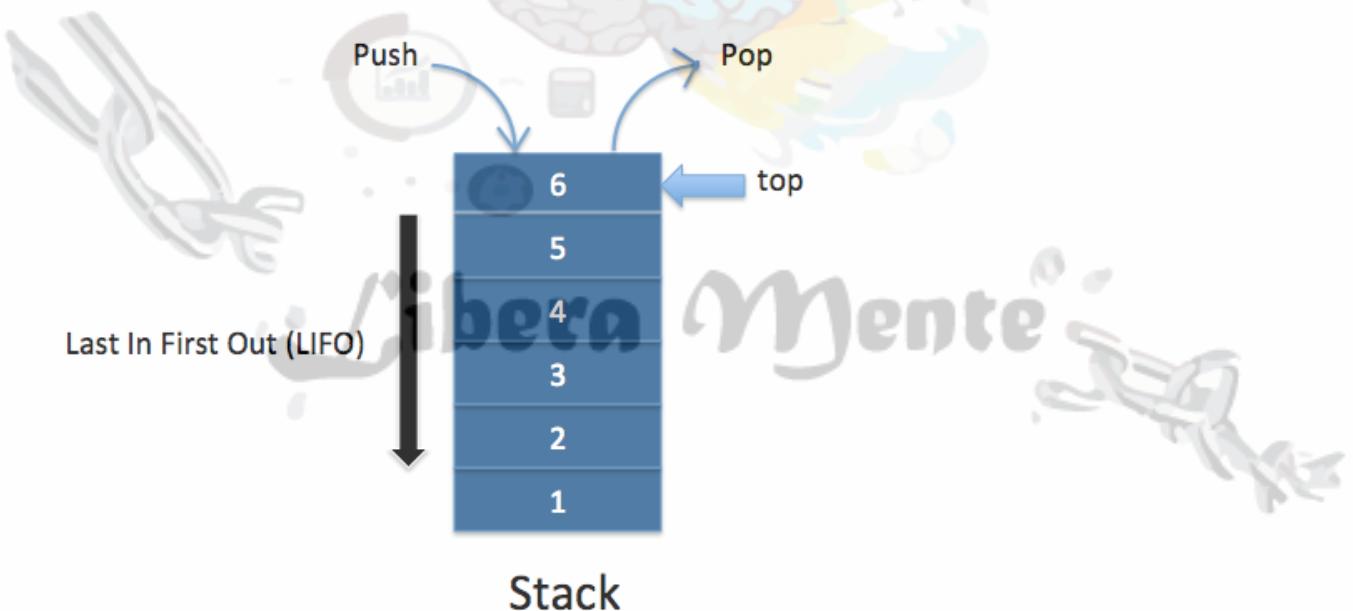
ADT Stack (Pila)

Una **pila** è una sequenza di elementi di un determinato tipo, in cui è possibile aggiungere o togliere elementi esclusivamente da un unico lato chiamato **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.

La pila è una lista chiamata anche **LIFO (Last-in-first-out)** cioè l'ultimo elemento inserito nella sequenza sarà il primo ad essere eliminato.

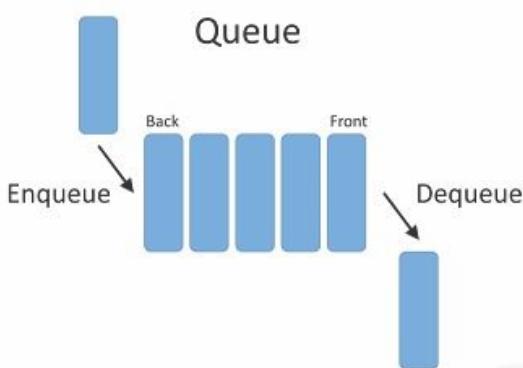


Tra le possibili implementazioni, abbiamo quella con:

- 1) **array:** lo *stack* è implementato come un puntatore ad una *struct stack* che contiene due elementi:
 - un array di *MAXSTACK* elementi;
 - un intero che indica la posizione del *top* dello *stack*.
 Inoltre, quando lo *stack* si riempie non è possibile eseguire operazioni di *push*.
- 2) **Lista concatenata:** lo *stack* è definito come un puntatore ad una *struct* che contiene un elemento *items* di tipo *list*. Non serve 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*.

ADT Queue (Coda)

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 lato (**head**).



La sequenza viene gestita con modalità **FIFO (First-in-First-out)**:

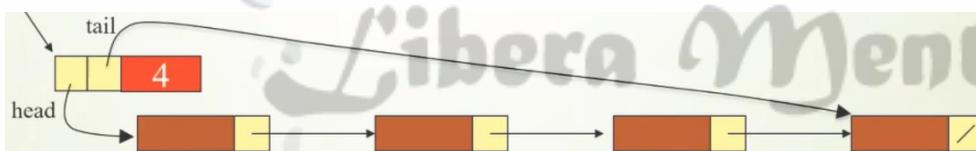
il primo elemento inserito nella sequenza sarà il primo ad essere eliminato. La coda è una struttura dati lineare a dimensione variabile. Infatti, si può accedere direttamente solo alla testa (**head**) della lista. Non è possibile accedere ad un elemento diverso da **head**, se non dopo aver eliminato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

Tra le possibili implementazioni, le più usate sono realizzate tramite: **Lista concatenata** e **array**.

Lista concatenata

E' possibile utilizzare gli operatori di rimozione della testa e aggiunta in coda. Per motivi di efficienza, conviene avere accesso sia al primo elemento sia all'ultimo. 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 nodo**;
- un puntatore **tail** ad uno **struct nodo**.



Per modificare l'implementazione **ADT Lista**, dobbiamo aggiungere il puntatore **tail**, poi bisogna modificare gli operatori principali quali **RemoveHead** (deve eventualmente aggiornare entrambi i puntatori **head** e **tail**), **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**).

Modifica removeHead

Bisogna prima salvare il puntatore al nodo da eliminare (quello puntato da **head**). **Head** dovrà quindi puntare al successivo e 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**.

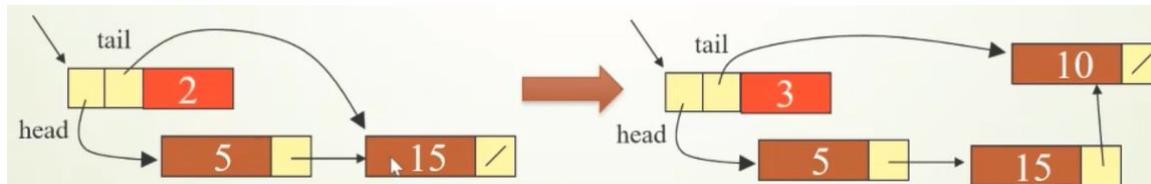
Modifica addListTail

Dobbiamo 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.



Array

La coda è implementata come un puntatore ad una ***struct queue*** che contiene tre elementi:

- 1) Un array di **MAXQUEUE** elementi;
- 2) Un intero che indica la posizione **head** della coda;
- 3) Un intero che indica la posizione **tail** della coda.

Quando la coda si riempie, non è possibile eseguire l'operazione **enqueue**. Se l'array viene gestito normalmente, cioè mantenendo $head \leq tail$, ci saranno dei problemi, avremo le posizioni a destra e di *tail* e a sinistra di *head* libere.

Una prima soluzione è di compattare l'array nelle posizioni iniziali, con uno shift degli elementi, ma ciò è molto costoso perché gli shift richiedono un numero lineari di operazioni.

Una seconda soluzione è di gestire l'array in modo circolare. Infatti, dopo aver inserito in posizione $N - 1$ (ultima posizione dell'array), se c'è ancora spazio in coda, si inseriscono ulteriori elementi a partire dalla posizione 0. In questo modo si riesce a garantire che ad ogni istante la coda abbia capacità massima di $N - 1$ elementi.

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: $tail = (tail + 1) \% N$.
- Ogni volta che si rimuove un elemento **head** avanza: $head = (head + 1) \% N$.

La coda è piena se il successore di *tail* in questo ordine circolare è *head*, ovvero $(tail + 1) \% N == head$.

Quando la coda è vuota, i valori di *head* e *tail* coincidono.

L'unico svantaggio di questo caso è di perdere una posizione, infatti possiamo inserire $N - 1$ elementi.

Sintesi:

	Pro	Contro
Lista	È un'implementazione espandibile (unico limite è la capacità di memoria)	La struttura è più complessa
Array circolare	Gli elementi sono memorizzati in modo contiguo e la struttura è più semplice	Dimensione fissata, bisogna conoscere a priori il numero massimo di elementi che la coda deve contenere, parte dello spazio è inutilizzabile.

Ricorsione

La **ricorsione** è una tecnica utilizzata in programmazione per la quale un sottoprogramma richiama sé stesso. La ricorsione è gestita dal *Sistema Operativo*; c'è un'area di memoria chiamata **stack** in cui sono inseriti dei **record di attivazione**. Ciascun record corrisponde all'esecuzione di una funzione.

Ogni volta che viene invocata una funzione viene creata dinamicamente una struttura dati detta *Record di attivazione*:

- Si crea una nuova **attivazione (istanza)** della funzione chiamata;
- Viene **allocata la memoria** per i parametri e per le variabili locali;
- Si effettua il **passaggio dei parametri**;
- Si **trasferisce il controllo** alla funzione chiamata;
- Si **esegue il codice** della funzione.

All'interno del record di attivazione troviamo:

- I **parametri formali**;
- Le **variabili locali**;
- L'**indirizzo di ritorno** (Return address RA) che indica il punto in cui tornare (nel codice chiamante) al termine della funzione;
- Un collegamento al record di attivazione del chiamante (**Link Dinamico DL**);
- L'**indirizzo del codice** della funzione (il puntatore alla prima istruzione del corpo).



Ciclo di vita di un Record di attivazione

Il record di attivazione associato a una chiamata di una funzione f è creato al momento dell'invocazione della funzione, permane per tutto il tempo in cui f è in esecuzione ed è distrutto (*deallocated*) al termine dell'esecuzione.

La dimensione del record di attivazione varia da una funzione all'altra e, per una data funzione, è fissa e calcolabile a priori.

Il Sistema Operativo mantiene un'area di memoria in cui vengono allocati i record di attivazione, gestiti come una **lista LIFO**, nella quale ogni elemento è un record di attivazione. La gestione dello stack avviene mediante due operazioni:

- 1) **Push**: aggiunta di un elemento (in cima alla pila);
- 2) **Pop**: prelievo di un elemento (dalla cima della pila).

L'ordine di allocazione dei record di attivazione nello stack indica la cronologia delle chiamate.

Programmazione ricorsiva

Un sottoprogramma ricorsivo è un sottoprogramma che richiama direttamente o indirettamente sé stesso. I linguaggi che gestiscono la ricorsione, lo fanno mediante **record di attivazione**.

Operativamente, risolvere un problema con un approccio ricorsivo comporta: identificare un “caso base”, con soluzione nota; esprimere la soluzione al caso generico n in termini dello stesso problema in uno o più casi più semplici ($n-1$, $n-2$, etc...).

Una funzione matematica è definita **ricorsivamente** quando nella sua definizione compare un riferimento a sé stessa. È basata sul **principio di induzione** matematica:

- **Base di induzione:** una proprietà P vale per $n = n_0$;
- **Passo Induttivo:** $P(n)$ vera $\rightarrow P(n + 1)$ vera $\forall n \geq n_0$

Iterazione e ricorsione

L'esecuzione di un algoritmo di calcolo che compiti “in avanti”, per accumulo, è un **processo computazionale iterativo**. Le caratteristiche fondamentali di un processo computazionale iterativo è che ad ogni passo è disponibile un **risultato parziale**. Infatti, dopo k passi si avrà a disposizione il risultato parziale al caso k .

Questo non è vero nei **processi computazionali ricorsivi**, in cui nulla è disponibile finché non si è giunti al caso elementare.

Ricorsione Tail

Un processo computazionale iterativo si può realizzare anche tramite funzioni ricorsive. Si basa sulla disponibilità di una variabile, detta accumulatore, destinata ad esprimere in ogni istante la soluzione corrente. L'accumulatore viene passato come parametro ad ogni chiamata della funzione. Questo tipo di ricorsione è detto **ricorsione tail**.

Una ricorsione che realizza un processo computazionale iterativo è una **ricorsione apparente**. La chiamata ricorsiva è sempre l'ultima istruzione: i calcoli sono fatti prima e la chiamata serve solo per proseguire la computazione. Questo tipo di ricorsione si chiama **Ricorsione Tail** (ricorsione in coda). Con la *ricorsione tail* il ciclo diventa un *if* con, in fondo, la chiamata *tail-ricorsiva*.

Complessità computazionale

In informatica, la **teoria della complessità computazionale** è una branca della teoria della computabilità che studia le risorse minime necessarie (principalmente **tempo di calcolo** e **memoria**) per la risoluzione di un problema. Con *complessità di un algoritmo* o *efficienza di un algoritmo* ci si riferisce dunque alle risorse di calcolo richieste. Per misurare l'efficienza di un algoritmo in maniera univoca, bisogna utilizzare una tecnologia indipendente, altrimenti uno stesso algoritmo potrebbe avere efficienza diversa a seconda della tecnologia sulla quale è eseguito. Per questo motivo si usa fare riferimento ad un modello di calcolo generico: la macchina di Turing. Inoltre, si fa una stima in funzione dell'input, si utilizza un comportamento asintotico e si stima il caso peggiore di configurazione dei dati.

Esempio di macchina astratta

Le istruzioni e condizioni atomiche hanno un 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 funzioni hanno un costo pari al costo di tutte le istruzioni e condizioni e il passaggio dei parametri ha costo nullo. Le istruzioni e le condizioni con chiamate a funzioni hanno un costo pari alla somma del costo delle funzioni invocate più uno.

Costo come funzione della dimensione dell'input

Cosa è la dimensione?

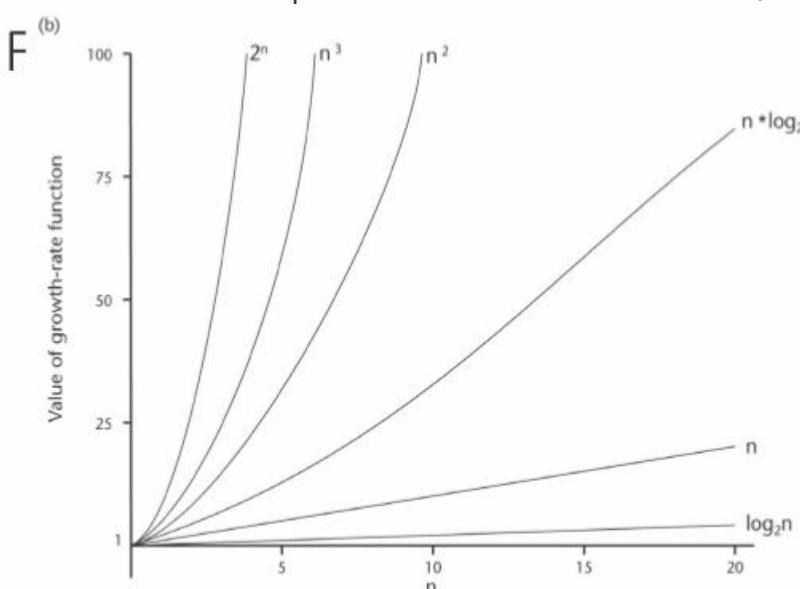
- **Vettore**: numero di elementi;
- **Albero**: numero dei nodi;
- **Grafo**: numero archi più numero nodi.

Comportamento asintotico

Nell'analizzare la complessità di tempo di un algoritmo siamo interessati a come aumenta il tempo al crescere della taglia n dell'input. Siccome per valori "piccoli" di n il tempo richiesto è comunque poco, ci interessa soprattutto il comportamento per valori "grandi" di n (il comportamento asintotico).

Trascuriamo tutte le costanti moltiplicative ed additive e tutti i termini di ordine inferiore. La suddivisione di algoritmi in classi di complessità è la seguente:

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



Complessità dei problemi

Studiare la complessità di un problema (ossia quello che un algoritmo risolve) è molto diverso dallo studiare la complessità di un algoritmo. Per poter dire che un problema ha complessità $O(g(n))$, ipotizzando di parlare del caso peggiore, basta trovare un qualsiasi algoritmo che lo risolva con $O(g(n))$. Per poter affermare che un problema è $\Omega(g(n))$ occorre invece dimostrare matematicamente che tutti i possibili algoritmi (inventati o non) lo risolvano alla meglio come $\Omega(g(n))$.

Per limitare superiormente un problema basta trovare almeno un algoritmo con complessità $O(g(n))$.

Per limitare il problema inferiormente bisogna studiare ogni possibile soluzione (il problema, in linea teorica, potrebbe essere risolto in tempo costante, ma si può sempre dimostrare il contrario).

Quando la complessità di un algoritmo è pari al limite inferiore di complessità determinato per un problema, l'algoritmo si dice ottimo (in ordine di grandezza).

Individuazione di limite inferiore

Dimensione n dei dati: se nel caso peggiore occorre analizzare tutti i dati allora $\Omega(n)$ è un limite inferiore alla complessità del problema.

Esempio: ricerca di un elemento o del massimo in un array.

Eventi contabili: la ripetizione di un evento un dato numero di volte è essenziale per la risoluzione di un problema.

Esempio: generare tutte le permutazioni di n oggetti. L'evento è la generazione di una nuova permutazione che si ripete per tutte le permutazioni, ossia $n!$ volte.

Ricorsione e valutazione della complessità

Negli algoritmi ricorsivi la soluzione di un problema si ottiene applicando lo stesso algoritmo ad uno o più sotto-problemi. La complessità viene espressa nella forma di una relazione di ricorrenza. La funzione di complessità $f(n)$ è definita in termini di se stessa su una dimensione inferiore dei dati.

Per la valutazione della complessità valutiamo:

- il lavoro di combinazione (preparazione delle chiamate ricorsive e combinazione dei risultati ottenuti), che può essere: costante, lineare, ...
- la forma dell'equazione di ricorrenza, che può essere con o senza partizione dei dati;
- il numero di termini ricorsivi (chiamate ricorsive nella funzione).

Gli Alberi

Prima di parlare di alberi dobbiamo prima conoscere cos'è un **grado**. Un **grafo** orientato G è una coppia $\langle N, A \rangle$ dove:

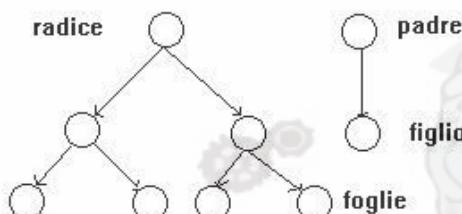
- N è un insieme finito non vuoto (insieme di nodi)
- $A \subseteq N \times N$ è un insieme finito di coppie ordinate di nodi, detti **archi** (o *spigoli* o *linee*).

Se $\langle u_i, u_j \rangle \in A$ nel grafo vi è un arco da u_i ad u_j . Il **Grafo** è una struttura dati alla quale si possono ricondurre strutture più semplici: Liste ed Alberi.

L'**Albero** è una struttura informativa utilizzata per rappresentare organizzazioni gerarchiche di dati, partizioni successive di un insieme in sottoinsiemi disgiunti, procedimenti decisionali enumerativi.

Proprietà

Ogni nodo ha un unico arco entrante, tranne la radice, che non ha archi entranti. Ogni nodo può avere zero o più archi uscenti. 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 massimo tra tutti i nodi.

Il **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 $0 \leq i < k$. La lunghezza del cammino è k .

Il **livello di un nodo** è la lunghezza del cammino dalla radice al nodo. Si può avere anche una definizione ricorsiva: il livello della radice è 0, il livello del nodo non radice è $1 + \text{il livello del padre}$.

L'**altezza dell'albero** è la lunghezza del più lungo cammino nell'albero; parte della radice e termina in una foglia.

Alberi vs Grafi

Un **albero** è un grafo diretto *aciclico*, in cui per ogni nodo esiste un solo arco entrante (tranne che per la radice che non ne ha nessuno). Se esiste un **cammino** che va da un nodo u ad un altro nodo v , tale cammino è unico. In un albero esiste un solo cammino che va dalla radice a qualunque altro nodo. Dato un nodo u , i suoi discendenti costituiscono un albero detto *sottoalbero* di radice u .

Struttura ricorsiva degli Alberi

Un albero è un insieme di nodi ai quali sono associate delle informazioni. Tra i nodi esiste un nodo particolare che è la radice (livello 0). Gli altri nodi sono partizionati in sottoinsiemi che sono a loro volta alberi (livelli successivi): vuoto o costituito da un solo nodo (detto radice) oppure è una radice cui sono connessi altri alberi.

Alberi binari

Sono particolari alberi n-ari: ogni nodo può avere al più due figli: *sottoalbero sinistro* e *sottoalbero destro*. La definizione ricorsiva di un albero è la seguente: un albero binario è vuoto oppure è una terna (s, r, d) , dove r è un nodo (la radice), s e d sono alberi binari. Abbiamo anche alberi binari semplificati: costruttore bottom-up, operatori di selezione, operatori di visita.

Realizzazione di Alberi binari

La realizzazione più diffusa di un albero binario è di utilizzare una struttura a puntatori con nodi doppiamente concatenati. Ogni nodo è una struttura con 3 componenti:

- 1) puntatore alla radice del sottoalbero sinistro;
- 2) puntatore alla radice del sottoalbero destro;
- 3) etichetta (useremo il tipo generico item per questo campo).

Un albero binario è definito come puntatore ad un nodo: se l'albero binario è vuoto, è un puntatore nullo; se l'albero binario non è vuoto, è un puntatore al nodo radice.

Creare un nodo dell'albero

Un albero binario viene costruito in maniera bottom-up. Man mano che costruiamo l'albero creiamo dei nuovi nodi da aggiungere come nodo radice. I passi per creare un nodo sono:

- 1) Allocare la memoria necessaria;
- 2) Memorizzare i dati nel nodo;
- 3) Collegare il sottoalbero sinistro e il sottoalbero destro, già costruiti in precedenza.

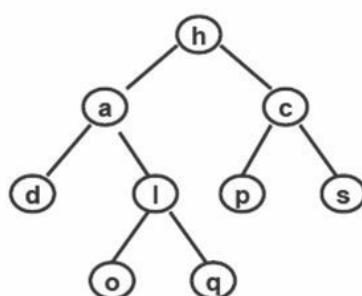
Algoritmi di visita dell'albero

La visita di un albero consiste nel seguire una rotta di viaggio che consente di esaminare ogni nodo dell'albero esattamente una volta.

- 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 dal primo 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.

ESEMPIO:

SIA UN ALBERO BINARIO CHE HA DEI CARATTERI NEI NODI



LA VISITA IN PREORDINE: h a d l o q c p s

LA VISITA IN POSTORDINE: d o q l a p s c h

LA VISITA SIMMETRICA: d a o l q h p c s

Operazioni: search

Se l'albero è vuoto allora restituisce *null*. Se l'elemento cercato coincide con la radice dell'albero restituisce l'*item* della **radice**. Se elemento cercato è minore della radice restituisce il risultato della ricerca dell'elemento nel sottoalbero sinistro. Se l'elemento cercato è maggiore della radice restituisce il risultato della ricerca dell'elemento nel sottoalbero destro.

Operazioni: min – Max

Algoritmo ricorsivo: Se l'albero è vuoto allora restituisci *null*. Se non esiste un sottoalbero sinistro (destro), ritorna l'item associato alla radice. Se esiste un sottoalbero sinistro (destro) effettua la ricerca del minimo (Massimo) nel sottoalbero sinistro (destro).

Algoritmo iterativo: *Tree_min(x)*

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

Operazioni: insert

Se l'albero è vuoto allora crea un nuovo albero con un solo elemento.

Se l'albero non è vuoto:

- se l'elemento coincide con la radice non si fa niente (elemento già presente);
- se l'elemento è minore della radice allora lo inserisce nel sottoalbero sinistro;
- se l'elemento è maggiore della radice allora lo inserisce nel sottoalbero destro;

Operazioni: delete

Si cerca ricorsivamente il nodo da rimuovere. Trovato il nodo:

Caso 1: se il nodo ha al più un **solo** sottoalbero di radice *r*, si bypassa il nodo da rimuovere agganciando direttamente il suo unico sottoalbero al padre e poi si rimuove il nodo.

Caso 2: il nodo ha entrambi i sottoalberi. Si sostituisce l'etichetta dell'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 sostituisce con l'elemento minimo nel sottoalbero destro. Si chiama ricorsivamente la **delete** sul sottoalbero sinistro nel nodo contenente l'elemento *Max*. L'albero risultante è un **ABR**.

Alberi bilanciati e alberi Δ -bilanciati

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

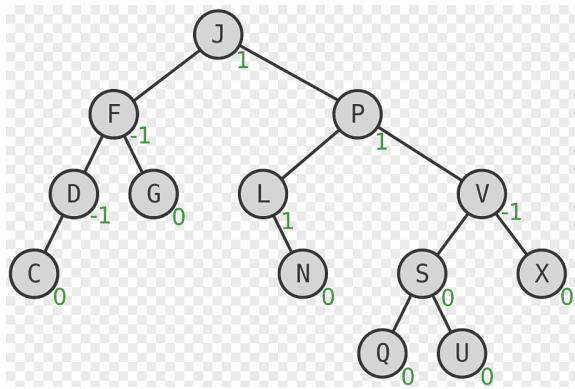
Le operazioni sull'albero binario di ricerca hanno complessità logaritmica se l'albero è Δ -bilanciato. Si può dimostrare che l'altezza dell'albero è $\Delta + \log_2 n$.

AVL

Un esempio di alberi bilanciati sono gli **alberi AVL** (dal nome dei suoi ideatori: Adel'son, Vel'skii Landis).

Per prevenire il non bilanciamento bisogna aggiungere un marcitore ad ogni nodo, che può assumere i seguenti valori:

- 1) **-1**, se l'altezza del sottoalbero sinistro è maggiore (di 1) dell'altezza del sottoalbero destro.
- 2) **0**, se l'altezza del sottoalbero sinistro è uguale all'altezza del sottoalbero destro.
- 3) **+1**, se l'altezza del sottoalbero sinistro è minore (di 1) dell'altezza del sottoalbero destro.



Ribilanciamento di alberi AVL

Un inserimento di una foglia può provocare uno sbilanciamento dell'albero: per almeno uno dei nodi l'indicatore non rispetta più uno dei tre stati precedenti.

In tal caso bisogna ribilanciare l'albero con operazioni di rotazione (semplice o doppia) agendo 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 foglia inserito. Considerazioni simili si possono fare anche per la rimozione di un nodo.

Ribilanciamento con Rotazioni

Rotazione semplice: Inserimento nel sottoalbero sinistro del figlio sinistro del nodo critico;

Inserimento nel sottoalbero destro del figlio destro del nodo critico;

Rotazione doppia: Inserimento nel sottoalbero destro del figlio sinistro del nodo critico;

Inserimento nel sottoalbero sinistro del figlio destro del nodo critico.

Heap

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

- 1) Le foglie (nodi a libello h) sono tutte addossate a sinistra;
- 2) Ogni nodo v ha la caratteristica che l'informazione ad esso è la più grande tra tutte le informazioni presenti nel sottoalbero che ha v come radice;

L'heap è utilizzato per realizzare code a priorità: le operazioni sono inserimento di un elemento e rimozione del max.

Un heap può essere realizzato con un array. I nodi sono disposti nell'array per livelli, la radice occupa la posizione 0 e se un nodo occupa la posizione i, il suo figlio sinistro occupa la posizione $2i + 1$ e il suo figlio destro occupa la posizione $2i + 2$.

Tabelle Hash

Una **tabella hash**, è una struttura dati usata per mettere in corrispondenza una data **chiave** con un dato **valore**. Chiavi e valori possono appartenere a diversi tipi primitivi o strutturati. Ad esempio, è possibile associare l'età (intero) ad una persona utilizzando il nome (stringa).

È possibile utilizzare un array, a patto che si associno prima gli indici interi alle persone e poi l'età agli indici.

Operatori

In molte applicazioni è necessario che un insieme dinamico fornisca solamente le seguenti operazioni:

- **INSERT (key, value)**: inserisce un elemento nuovo, con un certo valore (unico) di un campo **chiave**.
- **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.

Non è, ad esempio, necessario dover ordinare l'insieme dei dati o restituire l'elemento massimo, o il successore. Indicheremo con:

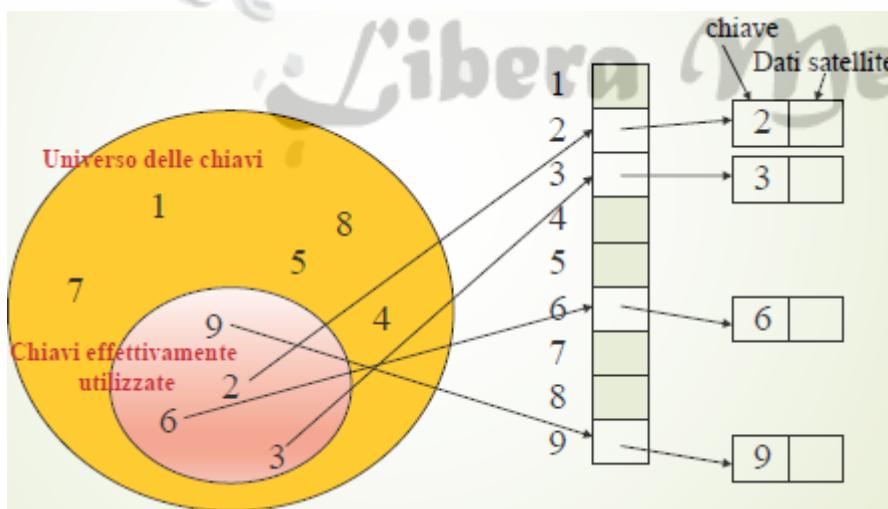
U -> l'universo di tutte le possibili chiavi

K -> l'insieme delle chiavi effettivamente memorizzate

Chiavi interne (indirizzamento diretto)

Se l'universo delle chiavi è piccolo e le chiavi sono intere allora è sufficiente utilizzare una **tabella ad indirizzamento diretto**. Questo tipo di tabella corrisponde al concetto di array:

- ad ogni chiave possibile corrisponde una posizione, o slot, nella tabella;
- una tabella restituisce il dato memorizzato nello slot di posizione indicato tramite la chiave in tempo $O(1)$.



Universo grande delle chiavi

Se le chiavi sono intere e/o l'universo delle possibili chiavi è molto grande non è possibile o conveniente utilizzare il metodo delle tabelle ad indirizzamento diretto. Può non essere possibile a causa della limitatezza delle risorse di memoria. Può non essere conveniente perché se il numero di chiavi effettivamente utilizzato è piccolo si hanno tabelle quasi vuote. Viene allocato spazio inutilizzato.

- Se $|K| \sim |U|$ non sprechiamo troppo spazio, avremo operazioni $O(1)$ nel caso peggiore.
- se $|K| \ll |U|$ allora avremo che se $U = \{\text{"numero di matricola"} \text{ degli studenti di PSD}\}$ e se il numero di matricola ha 6 cifre, l'array dovrà avere spazio per contenere 10^6 elementi.

Se gli studenti del corso sono ad esempio 30, lo spazio realmente occupato dalle chiavi memorizzate è di $\frac{30}{10^6} = 0.00003 = 0.003\%$

L'**hashing** permette di impiegare una quantità ragionevole di memoria che di tempo operando un compromesso tra i casi precedenti.

Con il metodo di indirizzamento diretto un elemento con chiave k viene memorizzato nella tabella in posizione k. Con il **metodo hash**, un elemento con chiave k viene memorizzato nella tabella in posizione $h(k)$. La funzione $h()$ è detta **funzione hash**.

Lo scopo della **funzione hash** è di definire una corrispondenza tra l'universo U delle chiavi e le posizioni di una tabella hash: $T[0...m-1]$ con $m << |U|$ $h: U \rightarrow \{0, 1, \dots, m-1\}$

Necessariamente la **funzione hash** non può essere **iniettiva**, ovvero due chiavi distinte possono produrre lo stesso valore hash.

Ogniqualvolta $h(k_i) = h(k_j)$ quando $k_i \neq k_j$ si verifica una **collisione**. Occorre quindi minimizzare il numero di collisioni (ottimizzando la funzione hash) e gestire le collisioni residue, quando avvengono (permettendo a più elementi di risiedere nella stessa locazione).

Risoluzioni delle collisioni

Per risolvere il problema delle collisioni si impiegano principalmente due strategie:

- 1) **metodo di concatenazione**;
- 2) **metodo di indirizzamento aperto**.

Metodo di concatenazione

L'idea è di mettere tutti gli elementi che collidono in una lista concatenata. La tabella contiene in posizione j un puntatore alla testa della *j-esima* lista oppure un puntatore nullo se non ci sono elementi.

Metodo di indirizzamento aperto

L'idea è di memorizzare tutti gli elementi nella tabella stessa; in caso di collisione si memorizza l'elemento nella posizione successiva: si genera un nuovo valore hash fino a trovare una posizione vuota dove inserire l'elemento.

Inoltre, si estende la funzione hash perché generi non solo un valore hash ma una sequenza di scansione, cioè prende in ingresso una chiave e un indice di posizione e genera una nuova posizione.

La caratteristica che deve avere h deve soddisfare la **proprietà di uniformità della funzione hash**: per ogni chiave k la sequenza di scansione generata da h deve essere una qualunque delle $m!$ permutazioni di $\{0, 1, \dots, m-1\}$.

È molto difficile scrivere funzioni hash che rispettino tale proprietà; si usano generalmente tre approssimazioni: **scansione lineare**, **scansione quadratica**, **hashing doppio**. Tutte queste classi di funzioni garantiscono di generare una permutazione ma nessuno riesce a generare tutte le $m!$ permutazioni.

Funzioni Hash

Le caratteristiche di una funzione hash sono: **criterio di uniformità semplice** → il valore di una chiave k è uno dei valori $0..m-1$ in modo equiprobabile.

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

Tipo Int

Metodo di divisione: la funzione hash è del tipo $h(k) = k \bmod m$ cioè il valore hash è il resto della divisione di k per m. Il pregio è che un metodo veloce.

Tipo Stringa

Per convertire una stringa in un numero naturale si considera la stringa come un numero in base 128. Esistono cioè 128 simboli diversi per ogni cifra di una stringa. È possibile stabilire una conversione fra ogni simbolo e i numeri naturali (codifica ASCII ad esempio). La conversione viene poi fatta nel modo tradizionale.

Es: convertire la stringa "pt" si ha $'p' * 128^1 + 't' * 128^0 = 112 * 128 + 116 * 1 = 14452$

Chiavi molto grandi

Spesso capita che le chiavi abbiano dimensioni tale da non poter essere rappresentate come numeri interi per una data architettura. Una soluzione è di utilizzare una funzione hash modulare, trasformando un pezzo di chiave alla volta. Per fare questo basta sfruttare le proprietà aritmetiche dell'operazione modulo e usare la regola di Horner per scrivere la conversione.

Operazione di cancellazione

L'operazione di cancellazione è difficile perché non si può marcare la posizione come vuota con *NIL* perché questo impedirebbe la ricerca degli elementi successivi nella sequenza.

Si potrebbe usare uno speciale marcatore *DELETED* invece di *NIL*. La procedura di inserzione dovrebbe poi scrivere in questi elementi.

In genere se si prevedono molte cancellazioni si utilizzano i metodi di concatenazione.