🆕

Introduzione al corso

Durante il nostro corso di studi di PSD, possiamo dire che ci concentreremo su quattro argomenti principali:

- **analisi**: comprensione del problema, definizione dei vincoli e delle specifiche.
- **progettazione**: scelta della strategia, formulazione di un algoritmo.
- **implementazione**: codifica della soluzione, test e debugging.
- **esecuzione**: applicazione su dati reali.

Prima di andare a scrivere un programma, sicuramente dovremo definire il problema, con tutti i suoi dati di input, di output ed eventuali dati intermedi. Il risultato sarà un algoritmo, ossia un insieme finito di istruzioni che è in grado di risolvere il problema di partenza. I dati di un problema sono la rappresentazione degli elementi che lo caratterizzano. Risolvere algoritmicamente un problema significa elaborare i dati che lo caratterizzano e arrivare a rappresentare una soluzione. Un tipo di dati è definito da un dominio di lavori e un insieme di operazioni previste. In C, il linguaggio in cui ci concentreremo, esistono dei tipi di **dati primitivi** (`int`, `char`, `float`), dei tipi di **dati aggregati** (array, strutture, enumerazioni, file) e i **puntatori**. Come costruiamo nuovi tipi di dati? Attraverso specifica, quindi definizione del dominio dei valori e degli operatori, e implementazione, cioè la codifica di quanto definito nella specifica. Le strutture dati che andremo a studiare sono collezioni di dati in maniera strutturata. Possiamo già fare una prima distinzione tra strutture dati.

- **SD lineari**: i dati vengono organizzati in maniera lineare, quindi hanno una dimensione fissa o che varia dinamicamente. È possibile accedere, aggiungere, togliere elementi in determinate posizioni. Tra queste SD individuiamo pile, code e liste.
- **SD non lineari**: i dati vengono organizzati in modo più complesso, creando relazioni gerarchiche o reticolari. Tra queste distinguiamo alberi, grafi, tabelle

hash.

Uno degli obiettivi del corso è certamente la **programmazione modulare**. Impareremo a suddividere i programmi in moduli, cioè in porzioni di programma che lavorano in maniera quanto più possibile indipendente da altri. Il secondo obiettivo è l'**astrazione** e l'***information hiding***. Vorremo creare un'interfaccia separata dalla codifica. Una particolare importanza la assume anche la ricorsione, cioè quando i dati sono organizzati in strutture definite in termini di sé stesse. In base a queste conoscenze, vorremo andare a creare **ADT**, ***Abstract Data Type***. La prima cosa che siamo intenzionati a fare è sicuramente quella di creare una specifica funzionante. Andremo a creare un ADT `Libro`. La prima cosa da fare è la definizione del dominio dei valori, nel nostro caso è una quadrupla, quindi autore, titolo, editore e anno, tre stringhe e un intero. Dopodiché, ci concentriamo sull'insieme degli operatori, che avrà due tipi di specifiche. La prima, è la **specifica sintattica**. Essa stabilisce la sintassi delle operazioni che è possibile eseguire sui valori del tipo di dato. Ne vediamo brevemente un esempio.

- `creaLibro (String, String, String, Intero)` $\longrightarrow$ `libro`
- `titolo (libro)` $\longrightarrow$ `String`
- `autore (libro)` $\longrightarrow$ `String`
- `editore (libro)` $\longrightarrow$ `String`
- `anno (libro)` $\longrightarrow$ `Intero`

La seconda è la specifica semantica, che spiega come funzionano le operazioni.

- `creaLibro (Aut, Tit, Edit, Anno) = Lib`
  - *Pre*:
  - *Post*: `Lib = (Aut, Tit, Edit, Anno)`
- `titolo (lib) = Titolo`
  - *Pre*: `lib = (Aut, Tit, Edit, Anno)`
  - *Post*: `Titolo = Tit`
- `autore (lib) = Autore`
  - *Pre*: `lib = (Aut, Tit, Edit, Anno)`

    - *Post*: `Autore = Aut`
- `editore (lib) = Editore`
    - *Pre*: `lib = (Aut, Tit, Edit, Anno)`
    - *Post*: `Editore = Edit`
- `anno (lib) = A`
    - *Pre*: `lib = (Aut, Tit, Edit, Anno)`
    - *Post*: `A = Anno`

    Una volta definite queste specifiche, potrebbe essere abbastanza intuitivo scrivere un'efficace implementazione di questo ADT.

```
struct Libro{
   char autore[26];
   char titolo[53];
   char editore[26];
   int anno;
};

typedef struct Libro *libro;
libro creaLibro(char *A, char *T, char *E, int anno)
{
   libro L;
   L = malloc(sizeof(struct Libro));
   if(!L) return NULL;
   strcpy(L→autore, A);
   strcpy(L→titolo, T);
   strcpy(L→editore, E);
   L→anno=anno;
   return L;
}
char *autore(libro L)
{
   char *aut;
   aut = calloc(26, sizeof(char));
```

```
    if(aut)
        strcpy(aut, L→autore);
    return aut;
}
char *titolo(libro L)
{
    char *tit;
    tit = calloc(53, sizeof(char));
    if(tit)
        strcpy(tit, L→titolo);
    return tit;
}
char *editore(libro L)
{
    char *ed;
    ed = calloc(26, sizeof(char));
    if(ed)
        strcpy(ed, L→editore);
    return ed;

}
int anno(libro L)
{
    return L→anno;
}
```

Analisi e Progettazione

Una parte fondamentale dello sviluppo di un programma è sicuramente l'analisi. Essa definisce cosa fa il programma, individuando i dati di ingresso e di uscita, basandosi sulla **precondizione** e sulla **postcondizione**. La prima è una condizione sui dati in ingresso, che deve essere soddisfatta affinché sia possibile l'esecuzione. La seconda è una condizione definita sui dati in uscita e deve essere soddisfatta al termine dell'esecuzione.

- **ES** = vogliamo analizzare un algoritmo che ordina una sequenza di interi.
  - *Dati in ingresso*: una sequenza s di n interi
  - Precondizione: $n > 0$
  - *Dati di uscita*: sequenza $s1$ di n interi
  - *Postcondizione*: $s1$ è un permutazione di s dove $\forall i \in [0, n-2], s1_i \leq s1_{i+1}$.

Un'altra fase sicuramente importante è la progettazione, ossia la definizione di come il programma effettua la trasformazione specificata, attraverso decomposizione funzionale e raffinamenti successivi. Dopodiché, l'algoritmo viene codificato nel linguaggio scelto, testando gli output. Prendendo l'esempio di prima, un buon algoritmo di ordinamento prende in input una sequenza s di un array A di dimensione n, ordina A e manda in output una sequenza $s1$ contenuta in A. Raffinando i passai, possiamo ottenere tre funzioni:

- `input_array(A, n)`
- `ordina_array(A, n)`
- `output_array(A, n)`

Analizziamo la funzione di ordinamento, scegliendo la strategia `SelectionSort`. Questo algoritmo noto di ordinamento, effettua una visita totale delle posizioni dell'array. Per ogni posizione visitata, individua l'elemento che dovrebbe occupare quella

posizione nell'array ordinato e scambia l'elemento trovato con quello che occupa attualmente la posizione. Quindi, come effettuiamo questo ordinamento? Prima individuiamo la posizione p dell'elemento minimo compreso tra i e $n-1$ dell'array a e poi scambiamo gli elementi di a di posizioni i e p.

- Ecco la specifica:
  - *Dati in ingresso*: array a di dimensione n, posizione i
  - Precondizione: $n > 0 \land i \geq 0 \land i < n$
  - *Dati di uscita*: posizione `pmin`
  - *Postcondizione*: $\forall j \in [i, n-1], a[pmin] \leq a[j]$

Prima di andare avanti, sappiamo che un **puntatore** è un dato che rappresenta l'indirizzo di un altro dato. I 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 (`int *pointer`). Ci tocca anche richiamare il concetto di **dereferenziazione**, ossia quando l'accesso all'oggetto puntato avviene attraverso l'operatore di dereferenziazione. Prima di poter usare un puntatore, questo deve essere inizializzato, ovvero deve contenere l'indirizzo di un oggetto. Per ottenere l'indirizzo di un oggetto si usa l'operatore unario `&`.
Un'altra strategia nota per l'ordinamento di un array è sicuramente la ricerca binaria. Si divide l'array in due metà e si confronta l'elemento da cercare con l'elemento centrale dell'array. Possiamo descrivere altri algoritmi di ordinamento, come
`InsertionSort`, che effettua una visita totale, ossia ad ogni passo, gli elementi che precedono l'elemento corrente sono ordinati. Ma anche `BubbleSort`, che implementa un algoritmo iterativo. Finché l'array non risulta ordinato, si effettua una visita durante la quale si scambiano gli elementi adiacenti che non risultano ordinati. Se in un'iterazione non è stato effettuato nessuno scambio, allora l'array è ordinato.

Astrazione e Modularizzazione

Abbiamo già accennato come la progettazione sia l'insieme delle attività relative al concepimento della soluzione informatica di un problema. La **modularizzazione** è una divisione effettuata con l'obiettivo di gestire le complessità. I moduli sono delle unità di programma che mettono a disposizione risorse e servizi computazionali e che sono fondamentali, poi, per l'astrazione e l'*information hiding*. In informatica, l'astrazione è una rappresentazione che nasconde le specifiche tecniche ad un consumatore di servizi. Ne abbiamo di diversi tipi:

- **astrazione funzionale**: ci concentriamo su cosa fa un sottoprogramma, astraendo da come esso realizza il suo compito.
- **astrazione sui dati**: un dato o un tipo di dato è totalmente definito insieme alle operazioni che sul dato possono essere fatte.
- **astrazione sul controllo**: un meccanismo di controllo è totalmente definito ed usabile indipendentemente dalle modalità e dalle tecniche con cui è realizzato.

Un modulo è costituito da un'**interfaccia**, che definisce le risorse e i servizi messi a disposizione dai clienti e una sezione implementativa, anche conosciuta come ***body***, che è completamente occultato dai clienti. Un modulo può usare altri moduli e può essere compilato indipendentemente. In C un modulo è generalmente un file. Per esportare le risorse definite in un modulo, il C fornisce un tipo di file chiamato ***header***, con l'estensione `.h`. L'header rappresenta l'interfaccia del modulo verso altri moduli. Per accedere alle risorse, possiamo includerle all'interno del file che ci interessa, un po' come si fa con le librerie predefinite. Una particolare importanza assume anche l'uso dei **commenti**, che servono da documentazione a chi dovrà usare la funzione. Come abbiamo già accennato, i moduli possono essere compilati indipendentemente, ma anche insieme. In entrambi casi, otterremo un file oggetto `.o`. Il linker combina il file oggetto e le librerie necessarie per avere l'eseguibile finale. In aiuto di questo tipo di compilazione viene il ***makefile***, cioè un file di testo che definisce le regole per la compilazione e il collegamento di un progetto. Per interpretare le istruzioni del

makefile, esiste il comando UNIX `make`, che determina quali file devono essere compilati.

Puntatori e Allocazione Dinamica della Memoria

Sappiamo che lo spazio di allocazione per le variabili ha, normalmente, una dimensione fissata. Ad esempio, quando andiamo a dichiarare un array, indichiamo la dimensione massima, la cosiddetta cardinalità, mentre la dimensione utilizzata, il riempimento, può essere inferiore. Bisogna, quindi, prevedere una cardinalità abbastanza grande per tutte le esecuzioni, evitando, però, di sprecare memoria inutilmente. Il C supporta l'**allocazione dinamica della memoria**, cioè viene allocata la memoria durante l'esecuzione del programma e si assegna l'indirizzo del blocco di memoria allocato ad un puntatore. Ci permette di creare strutture dati la cui dimensione varia durante l'esecuzione. La libreria `<stdlib.h>` dichiara generalmente tre funzioni per l'allocazione dinamica:

- `void *malloc(size_t size)`: alloca un blocco di memoria senza inizializzarlo. Come parametri prende il numero di bytes da allocare. Restituisce un puntatore al blocco di memoria allocato.
- `void *calloc(size_t nelements, size_t elementSize)`: alloca e inizializza un blocco di memoria contigua, impostando ogni byte a 0. Come parametri prende il numero di elementi da allocare e la dimensione in bytes. Restituisce un puntatore al blocco di memoria allocato e azzerato.
- `void *realloc(void *pointer, size_t size)`: ridimensiona un blocco di memoria precedentemente allocato con `malloc` o `calloc` a una nuova dimensione. Come parametri prende il puntatore al blocco di memoria da ridimensionare e una nuova dimensione desiderata in bytes. Restituisce un puntatore al nuovo blocco di memoria.

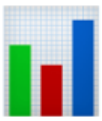
Nel nostro ambito, una particolare importanza la assumono gli **argomenti sulla linea di comando** del `main()`, che ci permettono di specificare input o opzioni direttamente al momento dell'esecuzione del programma.

- `int main(int argc, char *argv[])`

L'***argument count*** indica il numero di argomenti passati al programma, includendo il nome del programma stesso, quindi vale almeno 1. L'***argument vector*** è un array di puntatori e stringhe, che rappresentano ciascun argomento passato.
Fino ad ora, abbiamo assunto che le variabili siano dichiarate all'interno di funzioni. Questo tipo di variabile si dice
locale ed è visibile all'interno della funzione in cui è dichiarata. È possibile dichiarare dati anche in blocchi più interni. Tali variabili sono dette **automatiche**, poiché vengono allocate in memoria a tempo di esecuzione e deallocate al termine del blocco in cui sono dichiarate. Esistono anche variabili **globali**. Generalmente, una variabile globale è visibile a tutte le funzioni la cui definizione segue la dichiarazione della variabile nel file sorgente. Queste variabili sono anche dette **statiche**, perché la loro allocazione avviene all'atto del caricamento del programma. Le variabili globali possono essere usate per scambiare informazioni tra sottoprogrammi. Esistono tre aspetti importanti di una dichiarazione:

- **scope**: parte del programma in cui è attiva una dichiarazione.
- **visibilità**: dice quali variabili sono accessibili in una determinata parte del programma.
- **durata**: periodo durante il quale una variabile è allocata in memoria.

In generale, una funzione dichiarata in un file `.c` è utilizzabile in un altro file, anche in assenza della dichiarazione esplicita del prototipo nell'header file. Dichiarare una funzione statica la rende privata nel file in cui è dichiarata, quindi ne modifichiamo lo scope. Un discorso analogo può essere fatto per le variabile globali, dichiarandole come `extern`. Invece, dichiarazioni `static` di variabili globali, le rendono private al file in cui sono dichiarate, cambiandone lo scope. È possibile la dichiarazione static anche in variabili locali, che ne cambia la classe di allocazione. Come effetto, il valore di una variabile sopravvive all'esecuzione della funzione in cui è stato definito.



Testing dei Programmi e Uso dei File

Il testing esercita 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 che il programma produca, mentre un **malfunzionamento** è un comportamento diverso da quello atteso. Sappiamo che testare il programma con tutti i possibili dati di test è impraticabile, quindi bisogna individuare classi di test, selezionare un caso di test da esse, cercando di evitare ridondanze. Una test suite è un insieme di casi di test. Ad esempio, tenendo sempre conto del programma per l'ordinamento di un array che stiamo testando, dobbiamo tener conto della dimensione dell'array, ma anche considerare il caso in cui l'array è già ordinato, ad esempio. Creiamo una test suite per il nostro esempio:

- **Test Case 1 (array con un solo elemento)**
  - Input: `[5]`
  - Oracolo: `[5]`
- **Test Case 2 (array già ordinato)**
  - Input: `[1 2 3 4 5 6 7 8 9]`
  - Oracolo: `[1 2 3 4 5 6 7 8 9]`
- **Test Case 3 (array ordinato in maniera decrescente)**
  - Input: `[9 8 7 6 5 4 3 2 1]`
  - Oracolo: `[1 2 3 4 5 6 7 8 9]`
- **Test Case 4 (array non ordinato)**
  - 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]`

Un malfunzionamento, in un programma, è causato da un difetto, un errore o un bug, nel codice e può verificarsi in ogni fase dello sviluppo. Facciamo **debugging** quando individuiamo e correggiamo il difetto che ha causato il malfunzionamento. Da notare che, una volta individuato il bug, bisogna rieseguire tutti i casi di test. Per automatizzare il test, si possono usare i file per leggere dati di input e scrivere i dati di output. In C, il termine stream indica una sorgente di input o una destinazione. Ad esempio, per il TC4, potremo avere un file `TC4_input.txt` che legge gli elementi dell'array, un file `TC4_oracle.txt` e un altro file `TC4_output.txt` che serve per l'output. Cosa fare quando il programma ha più funzioni? Esistono due tipologie principali di strategie.

- **strategia big-bang**: integra il programma con tutti i sottoprogrammi e lo verifica nel suo insieme. Non è sicuramente una buona strategia, soprattutto per i programmi più grandi.
- **strategia incrementale**: si testa e si integra un sottoprogramma alla volta, considerando la struttura delle chiamate tra i sottoprogrammi, quindi *bottom-up*, *top-down*, *sandwich*, ecc ...Proprio bottom-up è una delle più utilizzate, che parte dai sottoprogrammi terminali e poi via via verso quelli di livello superiore, costruendo un programma main, detto ***driver***, che acquisisce i dati di ingresso del sottoprogramma, lo invoca e visualizza i dati di uscita.

Liste

Una **lista** è una sequenza di elementi di un determinato tipo, in cui è possibile aggiungere o togliere questi ultimi. Una sequenza vuota la chiameremo **lista vuota**. A differenza di un array, che è una struttura a dimensione fissa, dove è possibile accedere direttamente ad ogni elemento specificandone l'indice, la lista è a dimensione variabile e si può accedere direttamente solo al primo elemento della lista. Per accedere, quindi, ad un elemento generico, occorre scandire sequenzialmente gli elementi della lista.

1. Tipo di riferimento: `list`
2. Tipi usati: `item` (la lista è indipendente dal tipo di elementi contenuti in essa), `boolean`
3. Operatori: `newList()` → `list`, post: `l = nil`
    a. `emptyList(list)` → `boolean`, post: se `l=nil`, allora `b=true`, altrimenti `b=false`
    b. `consList(item, list)` → `list'`, post `<a_1, a_2, ..., a_n>` && `list' = <e, a_1, a_2, ..., a_n>`
    c. `tailList(list)` → `list'`, pre `l=<a_1, a_2, ..., a_n>` e `n>0`, post `l'=<a_2, ..., a_n>`
    d. `getFirst(list)` → `item`, pre `l=<a_1, a_2, ..., a_n>` e `n>0`, post `e = a_1`

`list` è l'insieme delle sequenze `L = a_1, a_2, ..., a_n` di tipo `item`. L'insieme `list` contiene, inoltre, un elemento `nil` che rappresenta una lista vuota, priva di elementi. Una possibile implementazione efficace, potrebbe riguardare le **liste concatenate**. Ogni elemento di una lista concatenata è un nodo con riferimento che serve da collegamento per il nodo successivo. Si accede alla struttura con il riferimento al primo nodo, mentre l'ultimo nodo contiene valore nullo. Una lista concatenata è più flessibile di un array, sfruttando solo la memoria strettamente necessaria. Gli elementi, però, non sono accessibili direttamente, quindi per accedere all'i-esimo elemento della lista, bisogna scorrere essa dal primo fino all'elemento i, appunto. Il modo più semplice per inserire un nuovo elemento in una lista concatenata `L` è inserire l'elemento in un nuovo nodo da aggiungere in testa alla lista. Quindi

creiamo un nuovo nodo, aggiungiamo il collegamento col nodo iniziale e aggiorniamo `L`, facendolo puntare al nodo appena raggiunto.

Pile

Una **pila**, che possiamo chiamare tranquillamente anche **stack**, è una sequenza di elementi di un determinato tipo, in cui è possibile aggiungere o togliere elementi, uno alla volta, esclusivamente da un unico lato, ossia il **top** dello stack. Questo vuol dire che la sequenza viene gestita con la modalità **LIFO**, ossia *Last-In-First-Out*. Lo stack è una struttura dati lineare, omogenea e 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. Uno stack è un insieme di sequenze `S = a_1, a_2, ..., a_n` di tipo item e contiene anche un elemento `nil` che rappresenta la pila vuota. Gli operatori sono:

- `newStack()` → `s`, post `s=nil`
- `emptyStack(s)` → `b`, post se `s=nil` allora `b=true` altrimenti `b=false`
- `push(e, s)` → `s'`, post `s = <a_1, a_2, ..., a_n>` && `s' = <e, a_1, a_2, ..., a_n>`
- `pop(s)` → `s'`, pre `s=<a_1, a_2, ..., a_n>` e `n>0` e post `s' = <a_1, a_2, ..., a_n>`
- `top(s)` → `e`, pre `s=<a_1, a_2, ..., a_n>` e `n>0` e post `e=a_1`

Tra le possibili implementazioni di uno stack, possiamo usare o un array o una lista concatenata. Lo stack verrà implementato come un puntatore ad una struct `c_stack` che contiene due elementi, un array di `MAXSTACK` elementi e un intero top che indica la posizione successiva a quella dell'elemento in cima allo stack.

Code

Una **coda**, spesso chiamata anche **queue**, 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**. Questo vuol dire che la sequenza viene gestita con la modalità **FIFO**, *First-In-First-Out*, cioè il primo elemento inserito nella sequenza sarà il primo ad essere eliminato. La coda è una struttura dati lineare a dimensione variabile, in cui si può accedere direttamente solo alla testa della lista.

Specifica Sintattica

- **Tipo di riferimento**: `queue`
- **Tipi usati**: `item` , `boolean`
- **Operatori**:
  - `newQueue()` → `boolean`
  - `emptyQueue(queue)` → `boolean`
  - `enqueue(item, queue)` → `queue`
  - `dequeue(queue)` → `item`

Specifica semantica

- **Tipo di riferimento queue**: queue è l'insieme delle sequenze `Q = a_1, a_2, ..., a_n` di tipo `item`. L'insieme queue contiene inoltre un elemento `nil`, che rappresenta la coda vuota.
- **Operatori**:
  - `newQueue()` → `q`
    - Post: `q = nil`
  - `emptyQueue(q)` → `b`

    - Post: se `q = nil`, allora `b=true`, altrimenti `b=false`
- `enqueue(e, q)` → `q'`
    - Post: se `q = nil`, allora `q' = <e>`, altrimenti se `q = <a_1, a_2, ..., a_n>`, allora `q' = <a_1, a_2, ..., a_n, e>`
- `dequeueu(q)` → `a`
    - Pre: `q = <a_1, a_2, ..., a_n>`
    - Post: `a = a_1` e l'elemento `a_1` viene rimosso da `q`

Ci sono, generalmente, due possibili implementazioni per le code: o con liste concatenate o con array.

Ricorsione

Ogni volta che viene invocata una nuova funzione, si crea una nuova istanza della funzione chiamante, viene allocata memoria per i parametri e per le variabili locali, si effettua il passaggio dei parametri e si trasferisce il controllo alla funzione chiamante, eseguendo poi, il codice. Al momento dell'invocazione, viene creata dinamicamente una struttura dati che contiene il *binding* dei parametri e degli identificatori definiti localmente. Questa struttura viene detta **record di attivazione** e contiene tutto ciò che serve per la chiamata alla quale è associato. Il record di attivazione viene creato al momento dell'invocazione e permane per tutto il tempo in cui la funzione è in esecuzione. Viene deallocato solamente al termine dell'esecuzione. Il record ha una dimensione variabile da una funzione all'altra. L'area di memoria in cui risiedono i record di attivazione viene gestita come uno **stack**, quindi con la politica LIFO e con le solite operazioni di `push()` e `pop()` che abbiamo già visto.
Una funzione matematica è definita **ricorsivamente** quando, nella sua stessa definizione, compare un riferimento a sé stessa. La ricorsione si basa molto sul principio di induzione matematica. In informatica, un sottoprogramma ricorsivo richiama direttamente o indirettamente sé stesso. I programmi che realizzano la ricorsione, lo fanno tramite i record di attivazione, in cui ad ogni chiamata ricorsiva ne è associato uno.

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

Studiando la ricorsione, possiamo giungere alla conclusione che le chiamate ricorsive decompongono via via il problema, ma non calcolano nulla e che, quindi, il risultato viene sintetizzato a partire dalla fine, andando a ritroso per i vari risultati parziali. È questa la vera essenza della ricorsione. E se il codice di prima lo volessimo **iterativo**?

```
int fact(int n)
{
   int i=1
   int F=1
   while(i<=n){
      F = F*i
      i = i+1
   return F
}
```

Notiamo come, ad ogni passo, viene accumulato un risultato intermedio, quindi il risultato viene sintetizzato in avanti. Il vantaggio principale è che, ad ogni passo, è disponibile un risultato parziale, contrariamente ai processi ricorsivi, in cui nulla è disponibile fino a che non giungiamo al caso elementare. In realtà, un processo computazionale iterativo può essere realizzato tramite funzioni ricorsive e si basa sulla disponibilità di una variabile, detta **accumulatore**, destinata ad esprimere, in ogni istante, la soluzione corrente. Una ricorsione che realizza un processo computazionale iterativo è detta **ricorsione apparente** o **ricorsione tail**. La chiamata ricorsiva è sempre l'ultima istruzione, e i calcoli sono effettuati precedentemente, con essa che serve solo e soltanto a proseguire la computazione.

```
int factIT(int n, int F, int i)
{
   if(i<=n)
      F = i*F;
      i=i+1;
      return factIT(n, F, i)
```

```
    return F
}
```

La soluzione è sintatticamente ricorsiva, ma dà luogo ad un processo computazionale iterativo

Complessità Computazionale

Per complessità computazionale, intendiamo il costo degli algoritmi in termini di risorse e calcolo, quindi tempo e spazio di memoria. Il tempo viene misurato in base a molti fattori, come la macchina usata, la configurazione dei dati, la dimensione dei dati e il comportamento asintotico. Generalmente, per i calcoli, si utilizza una **macchina astratta**, in cui:

- le istruzioni e le 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 sue istruzioni e condizioni; il passaggio dei parametri ha costo nullo.
- istruzioni e condizioni con chiamate a funzioni hanno costo pari alla somma del costo delle funzioni invocate più uno

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 il comportamento per grandi valori di n, quindi il **comportamento asintotico**. Gli algoritmi sono generalmente divisi in classi di complessità, come lineare, quadratica, logaritmica, esponenziale e c'è una scala che li divide. Siano f e g due funzioni positive. Diremo che

- $f(n)$ è $O(g(n))$ se esistono due costanti positive c ed n_0 tali che, se $n \geq n_0$, allora $f(n) \leq c \cdot g(n)$. La notazione O limita, quindi, superiormente la crescita dell'algoritmo e fornisce un'indicazione della bontà dell'algoritmo.
- $f(n)$ è $\Omega(g(n))$ se esistono due costanti positive c ed n_0 tali che, se $n \geq n_0$, allora $f(n) \geq c \cdot g(n)$. La notazione Ω limita inferiormente la complessità, indicando quindi che il comportamento.

Alberi Binari

Partiamo col dare la definizione di **grafo**. Un grafo orientato $G = \{V, E\}$ è una coppia dove V è l'insieme dei nodi ed E è l'insieme degli archi. Un grafo è una struttura dati alla quale si possono ricondurre strutture più semplici, come liste ed **alberi**. Gli alberi vengono utilizzati per rappresentare partizioni successive di un insieme in sottoinsieme disgiunti, organizzazioni gerarchiche di dati e procedimento decisionali enumerativi. In un albero ogni nodo ha un unico arco entrante, tranne uno, ossia la **radice**, che non ha archi entranti. Ogni nodo può avere zero o più archi uscenti. Esiste un cammino che congiunge la radice con ciascuno dei nodi e quest'ultimo è unico. Non esistono cicli e i nodi senza archi uscenti sono detti **foglie**. Un arco dell'albero induce una relazione padre-figlio. A ciascun nodo è associato un valore, chiamato **etichetta**. Il numero di figli di un nodo è detto **grado**. Un **cammino** è una sequenza di nodi $n_1, n_2, \ldots, n_k$ dove n_i è padre del nodo n_{i+1} e la lunghezza del cammino sarà k. Il **livello** di un nodo è la lunghezza del cammino dalla radice al nodo. La lunghezza del cammino più lungo dell'albero è detta **altezza**. Dato un albero e un suo nodo u, i suoi discendenti costituiscono un **sottoalbero** di radice u.
Un albero può essere definito ricorsivamente e può essere vuoto. Un albero binario è un albero particolare, in cui ogni nodo ha al più due figli e si fa sempre la distinzione tra il figlio sinistro, che viene prima, e il figlio destro

Specifica Sintattica

- **Tipo di riferimento**: `alberobin`
- **Tipi usati**: `boolean`, `nodo`, `item`
- **Operatori**:
  - `newBtree()` → `alberobin`
  - `emptyTree(alberobin)` → `boolean`
  - `getRoot(alberobin)` → `nodo`

- `figlioSX(alberobin)` → `alberobin`
- `figlioDX(alberobin)` → `alberobin`
- `consBtree(item, alberobin, alberobin)` → `alberobin`

Specifica semantica

- **Tipo di riferimento alberobin**: è l'insieme degli alberi binari dove **Δ è l'albero vuoto**
- **Operatori**:
  - `newBtree()` → `T`
    - Post: `T = Δ`
  - `emptyBtree(T)` → `b`
    - Post: se `T = Δ`, allora `b=true`, altrimenti `b=false`
  - `getRoot(T)` → `n'`
    - Pre: `T = <n, tsx, tdx>` non è vuoto
    - Post: `n = n'`
  - `figlioSX(T)` → `T'`
    - Pre: `T = <n, tsx, tdx>` non è vuoto
    - Post: `T' = tsx`
  - `figlioDX(T)` → `T'`
    - Pre: `T = <n, tsx, tdx>` non è vuoto
    - Post: `T' = tdx`
  - `consBtree(e, T1, T2)` → `T'`
    - Pre: `e ≠ NULLITEM`
    - Post: `T' = <n, T1, T2>` con `n` che è un nodo con etichetta `e`

Il modo più semplice per implementare un albero binario è una struttura a puntatori con nodi doppiamente concatenati. Ogni nodo ha tre componenti: un puntatore alla radice del sottoalbero sinistro, del sottoalbero destro e un'etichetta.

È frequente arricchire l'ADT Albero Binario con l'aggiunta di operatori per inserire o cancellari nodi in determinate posizioni dell'albero. La visita di un albero non vuoto consiste nel seguire una rotta di viaggio che consenta di esaminare ogni nodo dell'albero esattamente una volta. Esistono tre tipi di visite:

- **visita in pre-ordine**: richiede dapprima la visita della radice, poi del sottoalbero sinistro e poi del destro.
- **visita in post-ordine**: richiede dapprima la visita dell'albero sinistro, poi il destro e infine la radice.
- **visita simmetrica**: richiede prima la visita del sottoalbero sinistro, poi l'analisi della radice e poi la visita del sottoalbero destro.

Alberi da Ricerca Binari

Un insieme, **set**, è una struttura dati che può contenere una sola volta elementi di un determinato tipo. A differenza di una lista, in un set gli elementi non hanno una determinata posizione. Sugli elementi di un set può essere definita una relazione di ordinamento totale, indicata con $<$. Su un set, generalmente, sono definite operazioni di inserimento, eliminazione o ricerca di un elemento, ma possiamo anche implementare unione e intersezione.

Specifica Sintattica

- **Tipo di riferimento**: `set`
- **Tipi usati**: `item` , `integer` , `boolean`
- **Operatori**:
  - `newSet()` → `set`
  - `emptySet(set)` → `boolean`
  - `contains(set, item)` → `boolean`
  - `insertSet(set, item)` → `set`
  - `removeSet(set, item)` → `set`
  - `cardinality(set)` → `integer`
  - `unionSet(set, set)` → `set`
  - `intersectSet(set, set)` → `set`

Specifica semantica

- **Tipo di riferimento set**: è l'insieme
- **Operatori**:
  - `newSet()` → `s`

        - Post: `s = ø`
    - `emptySet(s)` → `b`
        - Post: se `s = ø`, allora `b=true`, altrimenti `b=false`
    - `contains(s, e)` → `b`
        - Post: se `e ∈ s`, allora `b=true`, altrimenti `b=false`
    - `insertSet(s, e)` → `s'`
        - Pre: `e ∉ s`
        - Post: `s' = s U {e}`
    - `removeSet(s, e)` → `s'`
        - Pre: `e ∈ s`
        - Post: `s = s' U {e} && e ∉ s'`
    - `cardinality(s)` → `n`
        - Post: `n` è il numero di elementi contenuti in `s`
    - `unionSet(s1, s2)` → `s'`
        - Post: `s' = s1 U s2`, con `s'` che contiene tutti gli elementi di `s1` e `s2`
    - `intersectSet(s1, s2)` → `s'`
        - Post: `s' = s1 ∩ s2`, con `s'` che contiene tutti gli elementi di `s1` e `s2`

Un set può essere implementato in diversi modi, come un array o una lista concatenata. L'implementazione cambia nel caso di un **orderder set**, ossia se tutti gli elementi sono ordinati. Di solito, però, si usano alberi da ricerca binari, che presentano operazioni efficienti di ricerca, inserimento e cancellazione. Se l'albero non è vuoto, ogni elemento del sottoalbero di sinistra precede la radice e ogni elemento del sottoalbero di destra segue la radice.
Le operazioni sul BST hanno complessità logaritmica se l'albero è **perfettamente bilanciato**. Questo significa che tutti i nodi interni hanno entrambi i sottoalberi e le foglie sono al livello massimo. Invece, un albero da ricerca si dirà **Δ-bilanciato** se, per ogni nodo, accade che la differenza tra le altezze dei suoi due sottoalberi è minore o uguale a Δ. Quando $\Delta = 1$, allora parliamo di **alberi AVL**.

Code a Priorità

Un albero quasi perfettamente bilanciato di altezza h è un albero perfettamente bilanciato fino al livello $h-1$. Un **heap** è un albero binario che la seguenti proprietà:

- proprietà strutturale: quasi perfettamente bilanciato e le foglie a livello h occupano tutte le posizioni disponibili a partire da sinistra verso destra.
- proprietà di ordinamento: 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.

L'heap, di solito, viene utilizzato per realizzare le **code a priorità**. Una coda a priorità è una struttura dati i cui elementi sono coppie `(key, value)` dette ***entry***, dove `key` e `value` appartengono a due insiemi qualsiasi `K` e `V`. Le *entry* vengono inserite in ordine qualsiasi, ma estratte in ordine di priorità, secondo il valore della `key`. Sull'insieme di chiavi è definita una relazione d'ordine $\leq$. Per convenzione, una entry `E1 = (k1, v1)` ha proprità su `E2 = (k2, v2)` se e solo se `k2` $\leq$ `k1`.

Specifica Sintattica

- **Tipo di riferimento**: `priorityqueue`
- **Tipi usati**: `boolean`, `item`
- **Operatori**:
  - `newPQ()` → `priorityqueue`
  - `emptyPQ(priorityqueue)` → `boolean`
  - `getMax(priorityqueue)` → `item`
  - `deleteMax(priorityqueue)` → `priorityqueue`
  - `insertPQ(priorityqueue, item)` → `priorityqueueù`

Specifica semantica

- **Tipo di riferimento priorityqueue**: è l'insieme delle code a priorità dove `∧` è la coda a priorità vuota.
- **Operatori**:
  - `newPQt()` → PQ
    - Post: PQ = ∧
  - `emptyPQ(PQ)` → `b`
    - Post: se PQ = ∧, allora `b=true`, altrimenti `b=false`
  - `getMax(PQ)` → `e`
    - Pre: `PQ` non è vuota
    - Post: `e` è la entry con la massima priorità tra quelle contenute in PQ
  - `deleteMax(PQ)` → `PQ'`
    - Pre: `PQ` non è vuota
    - Post: `PQ'` contiene tutte le entry di `PQ` tranne quella con massima priorità
  - `insertPQ(PQ, e)` → `PQ'`
    - Post: `PQ'` contiene `e` e tutte le entry di `PQ`

Per semplicità, tendiamo a supporre che le chiavi siano valori interi. Per l'inserimento, inseriamo il nuovo nodo nell'ultima foglia, con quest'ultimo che risale lungo il percorso che porta alla radice per individuare la posizione giusta. Per la rimozione, si rimuove sempre la radice e si pone l'ultima foglia al posto della radice. Per caratteristiche, un heap può essere realizzato con gli array, usando dei livelli.

Tabelle Hash

In molte applicazioni, potrebbe essere necessario che un insieme dinamico fornisca solamente le seguenti operazioni:

- `insert(key, value)`: inserisce un elemento nuovo, con un certo valore unico di campo chiave.
- `search(key)`: determina se un elemento con un certo valore della chiave esiste e lo restituisce.
- `delete(key)`: elimina l'elemento identificato dal campo chiave, se esiste.

Generalmente, si tende ad identificare con U l'insieme universo di tutte le chiavi e con K l'insieme delle chiavi effettivamente utilizzate. Se l'universo delle chiavi è piccolo, allora basterà utilizzare una **tabella a indirizzamento diretto**, simile ad un array, dove ad ogni chiave corrisponde una posizione. Altrimenti, se l'universo delle chiavi è molto grande, potremo avere delle risorse limitate di memoria. Per questo, allora, andremo ad utilizzare le **tabelle hash**, cioè strutture dati che trattano il problema della ricerca e permettono di mediare i requisiti di memoria ed efficienza nelle operazioni. Con il metodo hash, un elemento con chiave `k` viene memorizzato nella tabella in posizione `h(k)`, con `h()` che è detta **funzione hash**. Lo scopo della funzione hash è di definire una corrispondenza tra l'universo di chiavi e le posizioni di un'ipotetica tabella hash `T`. Un elemento con chiave `k` ha posizione pari al valore hash di `k`, denotato con `h(k)`. Necessariamente, la funzione hash non può essere iniettiva, cioè due chiavi distinte non possono produrre lo stesso valore hash, altrimenti si verificherebbe una **collisione**. Bisogna assolutamente limitare le collisioni.

Specifica Sintattica

- **Tipo di riferimento**: `hashtable`
- **Tipi usati**: `item`, `key`

- **Operatori**:
  - `newHashTable()` → `hashtable`
  - `insertHash(item, hashtable)` → `hashtable`
  - `searchHash(key, hashtable)` → `item`
  - `deleteHash(key, hashtable)` → `hashtable`

Specifica semantica

- **Tipo di riferimento hashtable**: è l'insieme di elementi `T = {a_1, a_2, ..., a_n}` di tipo `item`. Un item contiene un campo chiave di tipo `key` e dei dati associati.
- **Operatori**:
  - `newHashtable()` → `t`
    - Post: `t = {}`
  - `insertHash(e, t)` → t'
    - Pre: `e` deve avere key diversa da quelle presenti
    - Post: `t = {a_1, a_2, ..., a_n}` allora `t' = {a_1, a_2, ..., e, ...a_n}`
  - `deleteHash(k, t)` → `t'`
    - Pre: `t = {a_1, a_2, ..., a_n}`, `a_i(key) = k`
    - Post: `t' ={a_1, a_2, a_i-1, a_i+1, ..., a_n}`
  - `searchHash(k, t)` → `e`
    - Pre: `t = {a_1, a_2, ..., a_n}`
    - Post: `e = a_i` con $1 \leq i \leq n$ se `a_i(key) = k`, altrimenti `e=NULLITEM`

Una funzione hash deve avere un **criterio di uniformità semplice**, cioè il valore hash di una chiave `k` è uno dei valori $0 \ldots m-1$ in modo equiprobabile. Un altro requisito molto importante, è che una funzione hash dovrebbe poter utilizzare tutte le componenti della chiave per produrre un valore hash. In genere, le funzioni hash assumono che l'universo delle chiavi sia un sottoinsieme dei numeri naturali. Quando questo non si verifica, allora si convertono le chiavi in numero naturale. Spesso capita di avere chiavi di enormi dimensioni, tali da non poter essere

rappresentate da un'architettura comune. Un metodo alternativo, in questi casi, è quello di utilizzare una **funzione hash modulare**, trasformando un pezzo di chiave alla volta.