Appunti di Programmazione 1
Università degli Studi di Trento

Bozzo Francesco Conti Samuele

Febbraio 2019

Indice

Capitolo 1

Introduzione all'informatica e al computer

1.1 Definizione di informatica

Informàtica s.f.[dal fr. informatique, comp. di informat(ion) e (automat)ique «informazione automatica»] è definita dalla Treccani come l'insieme dei vari aspetti scientifici e tecnici che sono specificamente applicati alla raccolta e al trattamento dell'informazione e in partica all'elaborazione automatica dei dati.

1.2 Elementi di base

Prima di addentrarsi nel mondo della programmazione è giusto comprendere (o per alcuni semplicemente ricordare) cos'è l'informatica e che cosa studia, nonchè quali sono i suoi obbiettivi ed i suoi concetti basilari: spesso capita di addentrarsi così tanto tra le righe di codice da perdere la limpidezza della visione generale su quello che si sta facendo.
A questo scopo abbiamo introdotto la definizione precedente e questo breve elenco dei concetti fondamentali di questa branca della scienza:

- **Algoritmo**: un algoritmo è una sequenza precisa (=deterministica) e finita di operazioni, comprensibili da un esecutore, che portano alla realizzazione di un compito. L'esecutore non deve essere per forza di cose un computer, anche un libretto di istruzioni per montare un mobile è di per sè un algoritmo e l'umano che lo utilizza ne è l'esecutore. Le proprietà fondamentali di un algoritmo sono *correttezza* ed *efficienza*. Quando si opera con i calcolatori, gli algoritmi sono descritti utilizzando linguaggi che essi comprendono, i linguaggi di programmazione.

- **Linguaggio di programmazione**: linguaggio specifico del campo informatico dedicato a descrivere le istruzioni ad un calcolatore, perciò deve essere deterministico e rigoroso; inoltre esso è caratterizzato da:
  - una *sintassi*, cioè le regole che descrivono le stringhe di parole riconosciute dal linguaggio.
  - una *semantica*, ovvero le regole per interpretazione delle stringhe e che descrivono i processi computazionali dell'esecutore.

  I linguaggi di programmazione si possono definire di alto o basso livello a seconda della loro maggiore vicinanza al linguaggio naturale (alto livello, facilmente interpretabile dagli umani) o a quello della macchina (basso livello, facilmente interpretabile dai calcolatori).

- **Programma**: è un algoritmo codificato tramite uno specifico linguaggio. Data la complessità di alcuni programmi, sono stati sviluppati linguaggi intermedi (di alto livello) più vicini al linguaggio naturale che facilitano la scrittura dei programmi e che poi possono essere tradotti in linguaggi di basso livello, ovvero più vicini alla macchina (ed interpretabili da essa); esempi sono lo pseudocodice ed i diagrammi di flusso.

1.3 Il computer

Ora che abbiamo chiarito velocemente i concetti fondamentali possiamo iniziare a parlare degli strumenti di cui la scienza dell'informatica fa uso.
Al giorno d'oggi esistono molti tipi di computer con caratteristiche e scopi diversi tra loro. Nonostante ciò, si possono distinguere alcune componenti comuni:

- **CPU**: è l'unità di elaborazione del calcolatore (*Central Processing Unit*), essa carica istruzioni da eseguire dalla memoria centrale, interpreta le istruzioni ed infine le esegue. Il lavoro della CPU è scandito da impulsi generati da un orologio interno (*clock*): più è elevata la frequenza degli impulsi del *clock* più sono le istruzioni eseguite nell'unità di tempo. Seppur da molti anni la velocità del *clock* non si scosti di molto da 3GHz per problemi relativi alla dissipazione del calore, tuttavia la velocità dei computer è andata comunque aumentando per via dell'introduzione di nuove architetture nella progettazione dei calcolatori (es.: computer *multi-core*).

- **Memoria centrale (RAM)**: utilizzata per memorizzare dati e istruzioni, la *Random Access Memory* è una memoria il cui tempo di accesso

è indipendente dall'indirizzo del dato al quale si vuole accedere (al contrario delle memorie di massa). Si tratta di una memoria volatile, cioè il contenuto viene perso quando cessa l'alimentazione del sistema.

- **Bus di sistema**: interconnette tutti gli altri componenti, consentendo lo scambio di dati; esso è suddiviso in tre insiemi di *linee*:
  - Bus **dati**, per la trasmissione dei dati;
  - Bus **indirizzi**, un bus unidirezionale attraverso il quale la CPU decide in quale indirizzo scrivere o leggere i dati;
  - Bus **di controllo**: trasporta informazioni relative alla modalità di trasferimento e alla temporizzazione.

  In ogni istante è dedicato a collegare due unità, di cui una trasmette ed una riceve.

- **Periferiche di I/O**: ne esistono vari tipi: memorie di massa, monitor, tastiere, schede di rete, sensori etc.; non sono componenti fondamentali del calcolatore ma permettono che esso venga utilizzato per un'enorme quantità di funzioni diverse.

- **Memorie ROM**: sono un tipo particolare di memorie su cui non è consentita la scrittura (*Read Only Memory*). A esempio vengono utilizzate per memorizzare i firmware (software di basso livello, che comunicano direttamente con l'hardware, come il **BIOS**).

1.4 La macchina di Von Neumann

L'hardware sulla maggior parte dei computer moderni è progettato secondo lo schema della macchina di Von Neumann, facendo uso delle componenti indicate nell'elenco precedente. Il funzionamento di questa macchina si può riassumere schematicamente:

- le fasi di elaborazione si susseguono in modo sincrono rispetto all'orologio di sistema (*clock*).
- durante ogni intervallo di tempo, l'unità di controllo (parte del processore) stabilisce la funzione da svolgere e l'intera macchina opera in maniera sequenziale (anche se le architetture più evolute prevedono esecuzione contempoarnea di più istruzioni).
- il Bus di sistema collega tutte le componenti del calcolatore tra loro ed alla CPU, la quale gestisce tutti i flussi in ingresso ed uscita. Usando una metafora musicale, essa è il direttore dell'orchestra.

1.5 Astrazione e stratificazione

I concetti di **astrazione** e **stratificazione** si sono rivelati fondamentali nell'evoluzione dell'informatica e nello sviluppo della tecnologia. Con il passare del tempo si sono costruiti calcolatori sempre più potenti, con la conseguente richiesta di software sempre più complessi, in grado di gestire appunto gli hardware più evoluti.
I programmatori hanno provveduto quindi ad una progressiva astrazione dei software stessi al fine di trovarsi ad operare su rappresentazioni semplificate della macchina. In aggiunta, anche i programmi stessi hanno iniziato a diventare sempre più onerosi da gestire.
La soluzione più comunemente adottata consiste nello stratificare in vari livelli di astrazione l'intero software. Un esempio lampante è l'architettuta dei moderni sistemi operativi, che si sviluppa partendo dalla macchina fisica fino al livello delle applicazioni utilizzate dall'utente finale.
Segue uno schema della tipica architettura a strati di un sistema operativo:

Programmi utente
Interprete dei comandi
File system
Gestione delle periferiche
Gestione della memoria
Gestione dei processi
Macchina fisica

Tabella 1.1: Architettura a strati di un sistema operativo

1.6 Rappresentazione dell'informazione

L'ultimo argomento che tatteremo prima di passare al linguaggio C è la rappresentazione dell'informazione nella scienza informatica.
Essendo strutturalmente basato su dispositivi bistabili (con due stati stabili che possono essere utilizzati come base della rappresentazione), l'elaboratore elettronico può operare solo su sequenze di simboli binari. I due simboli convenzionalmente usati sono 0 e 1. Con il termine **BIT** (da *Bynary digIT*) si intende l'unità elementare di informazione; istruzioni e dati sono rappresentati nel computer tramite sequenze di BIT. Un insieme di 8 bit si chiama **byte**. Sono spesso utilizzati i multipli del byte (secondo la notazione del sistema internazionale): $KiloByte$(= 1000 byte), $MegaByte$(= 10^6 byte), $GigaByte$(= 10^9 byte).

Capitolo 2

Introduzione alla programmazione C

2.1 La nascita del C

La nascita del linguaggio C avviene come conseguenza della necessità di introdurre un nuovo linguaggio di programmazione che renda possibile una maggiore astrazione rispetto al linguaggio macchina; i punti su cui il linguaggio C si basa e che lo hanno reso così importante sono appunto:

- astrazione della memoria: utilizzare alias per identificare determinate celle di memoria.
- astrazione delle istruzioni: esprimere istruzioni complesse in un linguaggio comprensibile al programmatore.
- astrazione del linguaggio di definizione dell'algoritmo.

Il C venne creato nel 1972 da *Brian Kernighan* e *Dennis Ritchie*, ai *Bell Telephone Laboratories* con l'obiettivo di fornire un linguaggio flessibile e vicino alla macchina, tale da diventare il linguaggio di sistema per Unix.
Molti linguaggi di programmazione, tra cui il C, necessitano di una macchina astratta per funzionare; questa altro non è che una copia "logica" della macchina di Von Neumann con un ingrediente in più: l'interprete del suo rispettivo linguaggio. Il vantaggio di usare una macchina astratta sta nel fatto che un programma sviluppato in un certo linguaggio può funzionare su qualsiasi hardware su cui è installata la rispettiva macchina astratta.

2.2 Astrazione della memoria

Si definisce come **variabile** un contenitore di un determinato tipo di dato situato in una specifica porzione di memoria. Il dato contenuto all'interno di una variabile è suscettibile a modifica nel corso dell'esecuzione del programma.

Dichiarazione di variabili

```
1 int numero; // dichiarazione di una variabile di tipo
      intero
2 char carattere; // dichiarazione di una variabile di tipo
      carattere
3 float virgola_1, virgola_2;// dichiarazione di due
      variabili per contenere valori con la virgola
4
5 numero = 1; // inizializzazione della variabile numero
```

Una variabile è dunque caratterizzata da un *identificatore* (*left-value*) e da un *valore* (*right-value*). Quando in una espressione utilizziamo l'identificatore di una variabile, durante il calcolo viene considerato il suo valore.
Bisogna però prestare attenzione:

- Non è possibile inizializzare o utilizzare una variabile se non è stata precedentemente dichiarata.
- Gli identificatori delle variabili possono contenere unicamente caratteri alfanumerici della tabella *ASCII* e "_". Il primo carattere non può essere un numero.
- Gli identificatori delle variabili non possono essere *keyword* del linguaggio C.
- Il C è case sensitive: `nomeVariabile` è dunque diverso da `nomevariabile`.

2.2.1 L'attributo *const*

Attraverso l'attributo *const* è possibile dichiarare una costante: essa è del tutto analoga ad una variabile, tranne per il fatto che il suo valore non è modificabile durante il corso del programma.

Dichiarazione di una costante

```
1 const float pi = 3.14;
```

2.3 Le espressioni

Le **espressioni** in C vengono valutate seguendo l'albero sintattico indotto dalla *priorità degli operatori* e dalle parentesi.
Il valore di un'espressione può essere calcolato anche senza valutare l'intero albero sintattico, ove possibile. Questo meccanismo prende il nome di *lazy-evaluation*.

```
1 int a = 5;
2 bool test = (a<4) && (a>0); // in questo caso viene
      valutata solo (a<4), in quanto il suo valore e' false.
```

2.4 IO con il terminale

2.4.1 La funzione *printf*

La funzione `printf()` permette di scrivere del testo sulla standard output. Il primo argomento della funzione è una stringa formattata attraverso dei *descrittori di formato* (chiamati anche segnaposto). Seguono tanti argomenti quanti sono i segnaposto della stringa formattata. In base al tipo di dato che si vuole stampare si utilizzano diversi descrittori di formato:

- `%d` per numeri interi.
- `%f` per numeri con virgola.
- `%c` per caratteri.
- `%s` per stringhe.

È possibile specificare quanti caratteri riservare ad un segnaposto: l'output legato a quel descrittore di formato viene allineato a destra. Se necessario, il padding di default viene effettuato con `" "`: è comunque possibile specificare un carattere diverso.

Esempi di output con printf

```
1 printf("%d + %d = %d", 1, 2, 1+2); 1 + 2 = 3
2 printf("%2d/%2d/%4d %2d:%2d", giorno, mese, anno, ore,
      minuti); // padding con " "
3 printf("%02d/%02d/%4d %02d:%02d", giorno, mese, anno, ore,
      minuti); // padding con "0"
```

All'interno della stringa formattata è possibile utilizzare i caratteri di *escape*:

- `"\n"` per andare a capo.

- `"\r"` carriage return.
- `"\t"` per una tabulazione.
- `"\\"` per scrivere "\".
- `"\""` per scrivere """.

2.4.2 La funzione *scanf*

La funzione `scanf()` permette di leggere dei dati dallo standard input. La sintassi è del tutto analoga alla funzione di printf, tranne per il fatto di porre una "&" prima di ogni variabile utilizzata come argomento della funzione (nel caso di array non bisogna utilizzare "&"). Ulteriori spiegazioni verranno date alle sezioni 2.7 e 2.8.
Come di seguito, si consiglia di leggere un solo valore per ciascuna singola chiamata di scanf.

Esempi di input con scanf

```
1 int a, b;
2 char c;
3 scanf("%d", &a);
4 scanf("%d", &b);
5
6 fflush(stdin); // da utilizzare su Windows prima di leggere
      un carattere
7 scanf("%c, &c);
```

Leggere una sequenza di caratteri uno alla volta

```
1 int c; // getchar restituisce un intero (deve contenere il
     valore EOF)
2 while ((c=getchar()) != EOF) { ... }
```

2.5 Strutture di controllo

2.5.1 Il costrutto *if-else*

Il costrutto `if` permette di compiere delle azioni sulla base della valutazione di una espressione booleana. Il blocco di codice fra parentesi graffe viene infatti eseguito se l'espressione fra parentesi tonde è valutata vera.

Struttura del costrutto if

```
1 if (<espressione>){
2     ...
```

```
3 }
```

Quando si utilizza il costrutto if, è possibile (quindi è opzionale) farlo seguire dal blocco `else`. È possibile inoltre creare una catena a cascata di `else-if`:

Utilizzo di if, else-if e ramo else

```
1 if (numero > 0){
2     printf("numero positivo");
3 } else if (numero < 0) {
4     printf("numero negativo");
5 } else {
6     printf("numero nullo");
7 }
```

2.5.2 Il costrutto *switch*

In quei casi in cui la catena a cascata di else-if risulta troppo prolissa, è possibile utilizzare il costrutto `switch` in maniera del tutto simile:

Struttura del costrutto switch

```
1  switch (<integral value>){ // carattere o un numero
2      case value1:
3          ...
4          break;
5      case value2:
6          ...
7          break;
8      default:
9          ...
10         break;
11 }
```

È importante sottolineare che le istruzioni all'interno del costrutto switch vengono eseguite in sequenza a partire dal `case` il cui valore corrisponde a quello della variabile fra parentesi tonde. L'istruzione di `break` permette di uscire dal blocco dello switch.

Significato dell'istruzione break

```
1 int carattere = 'a';
2 switch (carattere){
3     case 'a':
4     case 'e':
5     case 'i':
6     case 'o':
7     case 'u':
8         printf("vocale");
```

```
9       default:
10          printf("non e' una vocale");
11          break;
12  }
```

Anche se è consigliata, l'istruzione break non è necessaria nel caso di default. Il costrutto switch è molto utilizzato con le enumerazioni.

2.6 Cicli

2.6.1 Il costrutto *while*

Il blocco di codice fra parentesi graffe viene ripetuto fino a quando l'espressione fra parentesi tonde è vera. L'espressione viene valutata prima di eseguire il blocco di istruzioni.

Struttura del costrutto while

```
1  while (<espressione>){
2      ...
3  }
```

2.6.2 Il costrutto *do-while*

Il blocco di codice fra parentesi graffe viene ripetuto fino a quando l'espressione fra parentesi tonde è vera. L'espressione viene valutata dopo aver eseguito il blocco di istruzioni (dunque il blocco di istruzioni viene eseguito almeno una volta). Spesso è utilizzato nei controlli dell'input.

Struttura del costrutto do-while

```
1  do {
2      ...
3  } while (<espressione>);
```

Lettura carattere da tastiera

```
1  char carattere;
2  do {
3      scanf("%c", &carattere);
4  } while (carattere<'a' || carattere>'z');
```

2.6.3 Il costrutto *for*

Il costrutto `for` è una versione specializzata del costrutto while. Si consiglia l'utilizzo del for in quei casi dove si necessita di una variabile contatore.

Struttura del costrutto for

```
1 for (<inizializzazione>; <espressione>; <incremento/
      decremento contatore){
2     ...
3 }
```

È consigliato dichiarare la variabile con funzione di contatore al di fuori del costrutto `for`.

Stampa i numeri da 1 a 10

```
1 int i;
2 for (i=1; i<=10; i++){
3     printf("%d\n", i);
4 }
```

2.6.4 Teorema di Bohem-Jacopini

Teorema (di Bohem-Jacopini)**.** *Qualunque algoritmo può essere implementato in fase di programmazione utilizzando tre sole strutture dette strutture di controllo: la sequenza, la selezione ed il ciclo (iterazione), da applicare ricorsivamente alla composizione di istruzioni elementari.*

Come abbiamo visto nelle precedenti sezioni, il C presenta tutte e tre le ipotesi ed è dunque riconosciuto come *linguaggio completo*.

2.7 Puntatori

Un puntatore è una variabile che contiene l'indirizzo di un'altra variabile (di un tipo specifico). Esso viene dichiarato utilizzando l'operatore di *dereferenziazione* `*`.

Dichiarazione generica di una variabile puntatore

```
1 tipoDato *nomePuntatore; // puntatore a una variabile di
      tipo tipoDato
```

Un altro operatore utile con i puntatori è `&`, che, se posto a sinistra di una variabile, restituisce l'indirizzo della variabile stessa.

Utilizzo dell'operatore &

```
1 int numero = 12;
2 int *ptr = &numero; // ptr contiene l'indirizzo di numero
```

È possibile anche dichiarare puntatori a puntatori:

Puntatori di puntatori

```
1 int numero = 12;
2 int *singlePtr = &numero;
3 int **doublePtr = &singlePtr; // doublePtr contiene l'
    indirizzo di singlePtr
```

Tramite i puntatori si può accedere al right-value di una variabile (puntata) e operare su di esso:

Operazioni su variabili tramite puntatori

```
1 int x = 42;
2 int *x_pointer;
3 // si crea il puntatore a intero "x_pointer"
4 x_pointer = &x;
5 // si associa l'indirizzo di x al puntatore
6 *x_pointer = 39;
7 // tramite la dereferenziazione (*) si opera su x
    assegnandogli il valore di 39
```

2.7.1 Aritmetica dei puntatori

È possibile eseguire operazioni di incremento, decremento, somma e differenza fra puntatori:

Esempi sull'aritmetica dei puntatori

```
1 int vettore[] = {1, 2, 3};
2 int *ptr = vettore; // ptr punta alla cella contenente 1
3 ptr++; // ora ptr punta alla cella di memoria successiva,
    contenente 2
4 ptr--; // ora ptr punta alla cella cella contenente 1
5 ptr = ptr+2; // ora ptr punta alla cella cella contenente 3
6 int result = ptr - vettore // e' pari a 2, ossia al numero
    di elementi int fra i due puntatori
```

2.8 Funzioni

2.8.1 Sottoprogrammi: funzioni e procedure

Le **funzioni** sono veri e propri sottoprogrammi che una volta inseriti nel codice possono essere riutilizzati più e più volte, senza creare ripetizioni nel codice. Ciò migliora il mantenimento e l'usabilità del codice.
In C posso definire due tipi di sottoprogrammi:

- funzioni: restituiscono un valore al chiamante, analogamente alle funzioni matematiche (sia tipi di default che user-defined).
- procedure: svolgono un compito per il chiamante ma non restituiscono un valore specifico (restituiscono `void`).

Nota: una funzione non può restituire un array (se necessario restituire un puntatore al primo elemento del vettore).
Gli argomenti della funzione, elencati nella testata, vengono chiamati i **parametri formali** della funzione. I **parametri effettivi** sono invece quelli con i quali la funzione viene invocata.
Per utilizzare una funzione in un programma si deve prima *dichiararla*, ovvero scriverne un prototipo, il prototipo di una funzione viene posto nella parte dichiarativa globale per essere invocata a seguire nel programma. Oltre a questo la funzione deve essere *definita* (ne deve essere descritto il procedimento).

Dichiarazione di una funzione

```
1 // dichiarazione
2 <return type> functionName (<type arg1> arg1, <type arg2>
      arg2, ...);
3
4 // definizione
5 <return type> functionName (<type arg1> arg1, <type arg2>
      arg2, ...){
6     ...
7 }
```

2.8.2 L'ambiente di una funzione in memoria

Quando in C un programma richiama una funzione, esso si interrompe momentaneamente per permettere alla macchina di svolgere tale funzione (che è essenzialmente anch'essa un programma): spesso viene creata una specifica macchina dedicata alla funzione con una propria memoria d'ambiente, nettamente separata da quella del programma chiamante.

Per gestire gli ambienti delle funzioni chiamate si sfrutta il meccanismo di *gestione della memoria a pila* (*stack*) con una struttura dati *LIFO* (*Last In First Out*).
L'invocazione di una funzione comporta l'allocazione di un nuovo ambiente in memoria sulla cima dello stack. Quando termina l'esecuzione del sottoprogramma si recupera il risultato e si rimuove l'ambiente dalla pila.

2.8.3 Passaggio dei parametri a funzioni

In C tutti i parametri passati alle funzioni vengono passati **per valore**, ossia ne viene effettuata una copia. Dunque, se viene modificato un argomento nel corpo di una funzione, la funzione chiamante non risentirà di questi cambiamenti.

Il passaggio per valore

```c
void cambiaValore(int a){
    a = 6;
}
int main(){
    int a = 5;
    cambiaValore(a);
    printf("%d", a); // stampa 5
}
```

Al contrario, utilizzando un puntatore è possibile eseguire un passaggio **per riferimento**, che consiste nel passare (sempre per valore) il punatore alla funzione e poi usare il suo valore per accedere alla variabile a cui punta: se viene modificato in questo modo un argomento nel corpo di una funzione, l'ambiente della funzione chiamante risentirà di questi cambiamenti.

Il passaggio per riferimento

```c
void cambiaValore(int* a){
    *a = 6;
}
int main(){
    int a = 5;
    cambiaValore(&a);
    printf("%d", a); // stampa 6
}
```

È inoltre possibile, attraverso l'attributo `const`, impedire la modifica della variabile puntata da puntatore.

Utilizzo dell'attributo const

```c
void cambiaValore(const int* a){
```

```
2       // non e' possibile modificare a
3   }
4   int main(){
5       int a = 5;
6       cambiaValore(&a);
7       printf("%d", a); // stampa 5
8   }
```

Tuttavia gli array sono trattati in modo particolare quando vengono utilizzati come parametro effettivo: viene passato per valore l'indirizzo di base dell'array, cioè l'indirizzo del primo elemento. Inoltre, quando si dichiara una funzione che accetta un vettore come parametro formale, non è necessario indicarne la misura fra parentesi quadre. Al contrario, per matrici di dimensione superiore a 1, è necessario specificare la grandezza di ciascuna dimensione (fatta eccezione per l'ultima) fra parentesi graffe.

Dichiarazione di funzioni con vettori e matrici come parametri formali

```
1   void funzione1(int vettore[]);
2   void funzione2(int matrice[3][]);
3   void funzione3(int matrice[3][4][2][]);
```

Nota: In C vige il meccanismo della deroga (*mascheramento*) fra i blocchi di codice: la dichiarazione di un elemento in un blocco maschera eventuali entità omonime esterne al blocco in considerazione. (Un blocco è una sezione di codice delimitata dalle parentesi graffe).

2.8.4 Funzioni ricorsive

Si definisce **ricorsiva** una funzione che richiama se stessa. Questo meccanismo viene sfruttato per trasformare problemi complessi in problemi elementari di immediata risoluzione.
Una funzione ricorsiva è costituita principalmente da due parti:

- *caso base*: soluzione del problema elementare.
- *caso induttivo*: divisone del problema complesso in problemi più semplici.

Calcolo del fattoriale di un numero implementato ricorsivamente

```
1   int fattoriale(int n) {
2       if (n < 0){
3           return -1;
4       }
5       if (n == 0){
6           return 1;
```

```
7       } else {
8           return n * fattoriale(n-1);
9       }
10  }
```

Seppur sia un meccanismo molto potente, programmare per ricorsione non è sempre la scelta più efficiente: ad ogni chiamata di funzione infatti viene creato un nuovo ambiente nello stack. Con input di dimensioni ragionevoli ciò non si nota, ma scalando si possono incontrare facilemente problemi legati all'eccessiva occupazione di memoria.

2.8.5 Funzione per la generazione di numeri pseudocasuali

Con l'ausilio della funzione `rand()` inclusa in `stdlib.h`, è comodo costruire una funzione che genera numero pseudocasuali compresi fra un minimo e un massimo:

Funzione per la generazione di numeri pseudocasuali

```
1  #include <stdlib.h>
2
3  int generaNumeroCasuale(int min, int max){
4      return (rand() % (max-min+1) + min);
5  }
```

2.9 Enumerazioni

Il costrutto `enum` permette di definire un nuovo tipo le cui variabili possono assumere solo un numero finito e predeterminato di valori.

Utilizzo di un tipo enum

```
1  enum seme {cuori, picche, fiori, quadri}; // definizione
2  enum seme semeCarta; // dichiarazione
3  semeCarta = cuori; // inizializzazione
```

In realtà, i membri di un'enumerazione sono rappresentati da costanti intere. Nell'esempio precedente "cuori" è rappresentato da uno 0, "picche" da un 1, e così via. È inoltre possibile modificare questa numerazione:

Modifica numerazione di un enum

```
1  enum seme {cuori=3, picche, fiori, quadri};
2  // il compilatore automaticamente setta:
3  //  - picche=4
```

```
4 //   - fiori=5
5 //   - quadri=6
```

Spesso una definizione di un tipo enum è accompagnata da un'operazione di `typedef`, che permette di definire un nuovo tipo:

Utilizzo di un tipo enum con typedef

```
1 typedef enum seme {cuori, picche, fiori, quadri} seme; //
      definizione
2 seme semeCarta; // dichiarazione
3 semeCarta = cuori; // inizializzazione
```

In realtà, attraverso una semplice istruzione del tipo `typedef a b;` tutte le occorrenze nel codice di `a` vengono sostituite con `b`: typedef è infatti semplicemente un operatore sintattico.

2.10 Strutture

2.10.1 Definizione e dot notation

In C è possibile definire una **struct**, ossia una struttura dati che raggruppa variabili eterogenee (di tipo diverso). Ogni elemento che la compone viene definito *campo* della struttura: data una variabile di tipo struttura è possibile accedere ai campi della stessa utilizzando la *dot notation*.

Esempio di definizione di struttura

```
1  // definizione
2  struct Impiegato{
3      char nome[20];
4      char cognome[20];
5      float stipendio;
6      char codiceFiscale[16];
7  } Impiegato;
8
9  struct Impiegato impiegato1; // dichiarazione
10 impiegato1.stipendio = 1200.50; // dot notation
```

Come per le enumerazioni, si può utilizzare anche con le strutture l'operatore sintattico `typedef`:

Esempio di definizione di struttura con typedef

```
1 typedef struct Punto{
2    float x, y;
3 } Punto;
4
```

```
5 Punto punto1;
```

È possibile utilizzare anche per le strutture l'operatore unario `sizeof`, il quale restituisce il numero di byte occupati in memoria.
Nota

- non è definito l'operatore `==` di confronto fra strutture.
- per effettuare la copia di una struttura basta usare l'operatore di assegnazione `=`.

2.10.2 Puntatori a strutture

Quando vi è la necessità di passare una determinata struttura ad una funzione come parametro, spesso si utilizza il passaggio per riferimento: è infatti molto meno onerosa la copia di un singolo puntatore piuttosto che la copia di una intera struttura. Spesso perciò torna utile il qualificatore `const`, che vieta alla funzione di modificare la struttura che le è stata passata per riferimento.

Quando si utilizza un puntatore a struttura, per accedere ad un campo della stessa è possibile utilizzare la notazione `->` piuttosto che la dot notation.

Accesso ai campi di un puntatore a struttura

```
1 Punto p1;
2 Punto *ptr = &p1;
3
4 (*p1).x = 1; // dot notation
5 p1->y = 2; // notazione ->
```

Capitolo 3

Vettori e matrici

3.1 Il concetto di array

Il vettore, chiamato più comunemente **array**, è il più semplice esempio di dato strutturato, esso consiste in una sequenza di celle consecutive ed omogenee, ovvero una sorta di contenitore di tante variabili dello stesso tipo a cui è possibile accedere tramite il nome del vettore e l'indice della variabile racchiuso tra parentesi quadre.
Di per sé l'array nel linguaggio C non è effettivamente un tipo di dato, esso è un costruttore di tipo che permette di creare il tipo di dato appena descritto.
Un array si crea ponendo l'identificatore del tipo di elemnti che conterrà, il nome dell'array e la dimensione dello stesso tra parentesi quadre nel modo seguente:

Implementazione di un array

```
1 //esempio con array con 3 interi
2 int vettore[3];
3
4 //esempio con array di 6 caratteri
5 char vettore_di_caratteri[6];
6
7 //esempio con array di 3 array di 2 interi
8 int vett[3][2];          //matrice 3*2
```

Si può usare come indice qualsiasi dato di tipo int e char. Si possono creare array contenenti qualsiasi tipo di dato, sia esso *built-in* o *user-defined*, semplice o strutturato. Come si è appena visto nell'esempio, si possono creare anche array di array (detti comunemente array bidimensionali o matrici) e non c'è limite al numero di dimensioni che si possono sfruttare.

3.2 Caratteristiche tecniche

I singoli elementi di un array vengono trattati dal C come vere e proprie variabili (del tipo definito nella dichiarazione dell'array): essi dunque possono essere coinvolti in tutte le operazioni riguardanti il loro tipo. L'array invece non può essere coinvolto globalmente in operazioni nè di assegnamento nè di confronto. Il metodo più comune per agire sull'array intero spesso prevede l'utilizzo di cicli: in particolare il ciclo *for* in cui il contatore viene utilizzato per scorrere l'array, come nell'esempio riportato qui sotto.

Implementazione di un array

```
1 //si vuole inizializzare il seguente array di interi con
      una sequenza di 30
2 int voti[10];
3 int i;
4 for(i=0; i<10; i++){
5     voti[i]=30;
6 }
```

3.3 Stringhe

Gli array vengono utilizzati anche per memorizzare stringe, i metodi di implementazione di qusto tipo di dato ed i suoi utilizzi saranno specificati in seguito.

3.4 Punatori e array

Esiste un legame stretto tra puntatori ed array: infatti, dichiarando un array `a[n]`, è possibile accedere al suo i-esimo elemento sia attraverso la scrittura `a[i]` che `*(a+i)`. Questo significa che l'identificatore `a` in realtà è un puntatore *read-only* alla prima cella dell'array. Si osservi il seguente esempio:

Utilizzo di puntatori come array

```
1 int vett[5];
2 int *p;
3 // ottenimento indirizzo del vettore
4 p=&vett[0]; // analogo a p=vett;
5
6 // vett = &p; ERRORE: l'identificatore vett e' read-only
7
8 // accesso ad un elemento del vettore
9 *(p+2)=7;     // analogo a vett[2]=7
```

```
10
11 // chiamata a funzione
12 int result = funz_random(vett); // analogo a funz_random(&
       vett[0])
13
14 // definizione di funzione
15 funzione_2(int v[]); // analogo a funzione_2(int *v)
```

Nelle ultime due righe abbiamo visto per passare gli array alle funzioni si utilizza l'indirizzo del loro primo elemento.
Tuttavia va ricordato che puntatori e vettori hanno caratteristiche molto diverse: i primi sono un tipo di dato semplice, mentre i secondi sono un tipo di dato strutturato. Come si può osservare in seguito dunque non possono essere sottoposti alle stesse operazioni (in particolare assegnamento ed incremento):

Differenze puntatori/array

```
1  int vett[5];
2  int *p;
3
4  // operazioni valide
5  p=vett;
6  p++;
7
8  // operazioni NON valide
9  // vett=p;
10 // vett++;
```

3.5 Nota teorica

Osservando da vicino la struttura dell'array, si nota che la principale limitazione consiste nel fatto che la dimensione di un array non può variare durante l'esecuzione del programma; questo implica che nel caso non si conoscano a tempo di compilazione le dimensioni di un input da memorizzare, si dovrà necessariamente sovrastimare le dimensioni dell'array per evitare il rischio di **overflow**, con conseguente spreco memoria.
Tale complicazione è dovuta alla complessità di realizzazione del compilatore di un linguaggio: se la macchina astratta (di un determinato linguaggio) conosce la quantità di memoria da allocare per un programma prima della sua esecuzione, potrà operare in maniera molto più efficiente, riservando a priori la memoria necessaria. In questo modo si evita la ricerca di eventuali porzioni di memoria libera durante l'esecuzione del programma.
Questi concetti verranno ripresi e approfonditi nel capitolo 6.

Capitolo 4

Stringhe

4.1 Il concetto di stringa

Come già visto nel capitolo 3, le stringhe sono strettamente legate al concetto di array, precisamente queste altro non sono che array di elementi di tipo char nalla cui ultima cella è contenuto il carattere "\0", identificato *terminatore di stringa*.
Essendo usate comunemente, le stringhe hanno una serie di comandi che ne semplificano l'input e l'output (utilizzando il descrittore di formato `%s`).

4.2 Operazioni con le stringhe

Oltre a questo esiste una libreria standard interamente dedicata, `string.h`, che comprende alcune utili funzioni quali `strcpy()`, `strlen()` e `strcmp()`:

Alcune operazione con le stringhe

```
1  char str1[5], str2[5];
2
3  fflush(stdin); // per Windows, da utilizzare prima di
       qualsiasi lettura di stringhe
4  scanf("%s", str1);
5  printf("%s", str1);
6
7  strcpy(str1, "cane"); //copia in str1 la stringa "cane"
8  strcpy(str2, str1); // copia il contenuto di str1 in str2
9
10 int lunghezza = strlen(str1); // restituisce la lunghezza
       di str1
11
12 bool uguali = strcmp(str1, "cane") == 0; // restituisce
       zero se uguali, >0 se str1>cane, <0 altrimenti
```

4.3 Nota tecnica

Considerando che l'ultimo carattere di una stringa deve essere il terminatore "\0", e per scrivere una parola di n lettere, bisogna utilizzare un array di lunghezza *n+1*.

Capitolo 5

Gestione file

5.1 Il concetto di file

Il C gestisce le interazioni con le varie periferiche fisiche rappresentandole come **file** su cui agisce attraverso un meccanismo chiamato stream. Esso funziona come un'interfaccia consistente, cioè permette di interagire con tutti i tipi di periferiche allo stesso modo. Un file viene associato ad uno **stream** (flusso) tramite un'operazione di *open*, (avvio delle comunicazioni tra software e periferica).
Il lingaggio C ha almeno 3 file costantemente aperti: `stdin` (file di input), `stdout` (file di output) e `stderr` (file di registrazione degli errori).

5.2 Come apriree chiudere un file

Nella libreria standard del C esistono diversi strumenti per operare su file di testo e file binari. In entrambi i casi la funzione di apertura del flusso è però la medesima:

Interfaccia funzione open()

```
1 FILE *file = fopen(<nome del file>, <tipo di apertura>);
```

La funzione `fopen()` inoltre restituisce l'indirizzo della struttura file associata al flusso creato; se al contrario incontra qualche errore, restituisce *NULL*. Di seguito sono elencate le diverse modalità di apertura disponibili:

- `"w"` : scrittura su file di testo.
- `"r"` : lettura da file di testo.

- "a" : aggiunta a file di testo.[1]
- "wb" : scrittura su file binari.
- "rb" : lettura da file binari.
- "ab" : aggiunta a file binari.

Esistono anche modalità avanzate ("w+" , "r+" , "a+") che permettono di modificare il cursore del file, decidendo in che posizione del file agire.
Ad ogni apertura di un file deve corrispondere una chiusura, attraverso la funzione fclose() :

Interfaccia funzione close()

```
1 fclose(<file>);
```

5.3 Operazioni di lettura e scrittura su file

Per la scrittura su un file di testo:

Scrittura su file di testo

```
1 fputc(int c, file_output); // ritorna EOF come errore
2 fputs(stringa, file_output); // ritorna EOF come errore
3 fprintf(file_output, [formato_output], [var_list variabili
    ]); // ritorna il numero di elementi scritti
```

Per la lettura da un file di testo:

Lettura da file di testo

```
1 carattere = fgetc(file_input); // ritorna EOF come errore
2 fgets(stringa, lunghezza, file_input); // fino a newline o
    EOF; ritorna EOF come errore
3 fscanf(file_input, [formato_input], [var_list puntatori]);
    // ritorna il numero di elementi letti
```

Per verificare se ci si trova alla fine del file:

Controllo fine file

```
1 int result = feof(file_input); // se la fine e' stata
    raggiunta, result assume un valore diverso da 0
```

Seppur la libreria standard del C offre questa funzione, quando bisogna leggere in insieme di elementi di cui non si conosce la dimensione è consigliato

[1] La modalità aggiunta è necessaria poichè ogni volta che un file viene aperto con comando "w" o "wb" il suo contenuto precedente viene eliminato

utilizzare `fscanf()`. In particolare il controllo da eseguire è sul valore di ritorno di fscanf: se non sono stati letti tanti elementi quanti ci si aspetterebbe, allora il file è concluso.

Per lettura e scrittura su file binario:

Controllo fine file

```
1 int n_read_elem = fread (void *ptr, len_elemento,
     n_elementi, file_input);
2 int n_written_elem = fwrite(void *ptr, len_elemento,
     n_elementi, file_input);
```

Una caratteristica interessante delle funzioni relative ai file binari è quella di offrire la possibilità di lettura/scrittura di tipi di dato, sia *built-in* che *user-defined*, con una singola istruzione.

5.4 Alcuni esempi

Scrittura su file di testo

```
1  // creazione del puntatore alla struttura file
2  FILE *f_txt;
3  int x = 7;
4
5  // apertura file in modalita' scrittura con controllo sull'
       apertura
6  if ((f_txt=fopen("nomefile.txt", "w")) == NULL){
7          printf("errore apertura file fase w");
8  }
9
10 // scrittura sul file
11 fprintf(f_txt, "numero_piccioni: %d", x);
12 fprintf(f_txt, "si puo' scrivere anche solo testo", NULL);
13
14 fclose(f_txt);
```

Lettura da file di testo

```
1 FILE *f_txt;
2 if((f_txt=fopen("nomefile.txt", "r"))==NULL){
3     printf("errore apertura file fase r");
4 }
5
6 int x;
7 char testo[20];
8 fscanf(f_txt, "%s: %d", &testo, &x); //lettura
```

```
9
10 fclose(f_txt);
```

Append di un file di testo

```
1  FILE *f_txt;
2  if ((f_txt=fopen("nomefile.txt", "a")) == NULL){
3      printf("errore apertura file fase a");
4  }
5
6  fprintf(f_txt, "testo aggiunto\n", NULL); // append
7
8  fclose(f_txt);
```

File binari (.bin)

```
1  //creazione del puntatore alla struttura file
2  FILE *f_bin;
3  int x=12;
4
5  if ((f_txt=fopen("nomefile.bin", "wb")) == NULL){
6      printf("errore apertura file fase wb");
7  }
8
9  // se si sta programmando su Windows, per poter controllare
       il file si deve comunque dichiararlo come .txt
10
11 fwrite(&x, sizeof(int), 1, f_bin); // scrittura sul file
       bin
12 fclose(f_bin);
```

Scrittura di una struttura su file binario

```
1  typedef struct Tpunto {
2      float x, y;
3  } Tpunto;
4
5  Tpunto p1;
6  p1.x = p2.x = 0.3;
7  FILE *f_bin;
8  if ((f_txt=fopen("nomefile.bin", "wb")) == NULL){
9      printf("errore apertura file fase wb");
10 }
11
12 fwrite(&p1, sizeof(Tpunto), 1, f_bin); // scrittura binaria
       di un elemento Tpunto
13 fclose(f_bin);
```

Lettura di una struttura da file binario

```
1  typedef struct Tpunto {
2      float x, y;
3  } Tpunto;
4
5  Tpunto p1;
6  FILE *f_bin;
7  if((f_txt=fopen("nomefile.bin", "rb")) == NULL){
8      printf("errore apertura file fase rb");
9  }
10
11 fread(&p1, sizeof(Tpunto), 1, f_bin); // lettura binaria di
       un elemento Tpunto
12
13 fclose(f_bin);
```

Scrittura di una stuttura su file binario in append

```
1  typedef struct Tpunto {
2      float x, y;
3  } Tpunto;
4
5  Tpunto p1;
6  p1.x = p2.x = 0.3;
7  FILE *f_bin;
8  if ((f_txt=fopen("nomefile.txt", "ab")) == NULL){
9      printf("errore apertura file fase ab");
10 }
11
12 fwrite(&d2, sizeof(Tdato), 1, f_bin); // scrittura binaria
       di un elemento Tpunto
13
14 fclose(f_bin);
```

Capitolo 6

Allocazione dinamica della memoria

6.1 Heap vs stack

Il linguaggio C permette al programmatore di utilizzare dei meccanismi di allocazione e deallocazione dinamica della memoria per gestire situazioni in cui un'allocazione statica sarebbe svantaggiosa. Un semplice esempio riguarda quei casi in cui non si è a conoscenza, al tempo della compilazione, delle dimensioni con cui si svilupperanno certe strutture dati durante l'esecuzione.
A differenza delle **variabili automatiche** che vengono memorizzate nello *stack*, le variabili allocate dinamicamente si trovano nell'**heap**.
Naturalmente da grandi poteri derivano grandi responsabilità: ogni istruzione di allocazione deve essere accompagnata da una successiva istruzione di deallocazione. Nel caso contrario si possono riscontrare errori molto subdoli e nascosti come la produzione di *garbage* (ovvero celle di memoria piene ma non recuperabili) e di *dangling references* (ovvero celle di memoria a cui si può accedere da più puntatori, con il rischio di modificarle non intenzionalmente).

6.2 Allocazione e deallocazione...

Nel caso di fallimento di un'operazione di allocazione viene restituito il valore *NULL*.
È possibile utilizzare le istruzioni di deallocazione ad un puntatore se e solo se esso contiene un indirizzo sull'heap assegnato dinamicamente.

6.2.1 ... in C

Nel linguaggio C è necessario includere la libreria *stdlib.h* per utilizzare le istruzioni per allocazione e deallocazione:

Allocazione e deallocazione dinamica di una variabile in C

```
1 tipo* ptr_var = (tipo*) malloc(sizeof(tipo));
2 free(ptr_var);
```

Allocazione e deallocazione dinamica di un vettore in C

```
1 tipo* ptr_arr = (tipo*) malloc(sizeof(tipo)*len_array);
2 free(ptr_arr);
```

Nota: `sizeof` è un operatore unario che restituisce il numero di byte che occupa in memoria il tipo di dato che gli viene fornito.

6.2.2 ... in C++

Nel linguaggio C++ le istruzioni per allocazione e deallocazione sono:

Allocazione e deallocazione dinamica di una variabile in C++

```
1 tipo* ptr_var = new tipo;
2 delete ptr_var;
```

Allocazione e deallocazione dinamica di un vettore in C++

```
1 tipo* ptr_arr = new tipo[];
2 delete [] ptr_arr;
```

6.3 Tre sezioni di memoria

Per ricapitolare, esistono principalmente tre sezioni di memoria in un programma C/C++:

- *Memoria automatica*: contiene le variabili che vengono allocate/deallocate automaticamente nello stack quando viene invocata/termina la funzione in cui sono definite.
- *Memoria dinamica*: viene allocata a run-time nello heap dagli operatori presentati precedentemente.
- *Memoria statica*: contiene le variabili statiche e globali.

Capitolo 7

Teoria sulla complessità

7.1 La funzione tempo di esecuzione

Dato un qualsiasi algoritmo, è possibile ottenere la funzione $T(n)$ ad esso associata, ossia la **funzione tempo di esecuzione** rispetto alla dimensione dell'input n. Questa funzione prende in considerazione solamente il numero di istruzioni e non aspetti dinamici d'esecuzione.
Studiando l'andamento asintotico di $T(n)$, si può notare che è rilevante ciò che cresce più velocemente: *es* $T(n) = a + bn + cn^2 \Rightarrow T(n) = \Theta(n^2)$.

Il caso peggiore, migliore e medio

Si possono costruire *tre* funzioni $T(n)$ per stimare l'efficienza dei un programma:

- $T_{\text{peggiore}}(n)$ nel caso peggiore
- $T_{\text{migliore}}(n)$ nel caso migliore
- $T_{\text{medio}}(n)$ nel caso medio

Di ciascuna di esse si può determinare l'andamento asintotico attraverso le notazioni successivamente fornite.

7.2 Notazione per l'andamento asintotico

Notazione limite superiore O-grande

$$f(n) = O(g(n))$$
$$\text{se } \exists n_0 > 0, \exists c_2 > 0 \text{ tc. } f(n) \leq c_2 g(n) \forall n > n_0$$

Notazione limite inferiore Ω-omega

$$f(n) = \Omega(g(n))$$
$$\text{se } \exists n_0 > 0, \exists c_1 > 0 \text{ tc. } f(n) \geq c_1 g(n) \forall n > n_0$$

Notazione limite superiore e inferiore Θ-theta

$$f(n) = \Theta(g(n))$$
$$\text{se } \exists n_0 > 0, \exists c_1 > 0, c_2 > 0 \text{ tc. } c_1 g(n) \leq f(n) \leq c_2 g(n) \forall n > n_0$$

7.3 Definizione di complessità

Complessità di un algoritmo La complessità di un algoritmo equivale alla misura del numero di istruzioni da eseguire per risolvere il problema.

Complessità di un problema La complessità di un problema equivale alla complessità del migliore algoritmo che lo risolve.

7.4 Le equazioni di ricorrenza

Quando si deve analizzare un algoritmo ricorsivo del tipo *divide et impera*, non è possibile scrivere l'espressione analitica di $T(n)$. Bisogna utilizzare un'equazione di ricorrenza:

$$T(n) = \begin{cases} \Theta(1), n \leq c \\ aT\left(\frac{n}{b}\right) + D(n) + C(n), n > c \end{cases}$$

Dove:

- c costante
- a numero di problemi generati da *divide*
- b dimensione dei sottoproblemi rispetto a quello originale
- $D(n)$ tempo impiegato per dividere il sottoproblema (operazione di *divide*)
- $C(n)$ tempo impiegato per ricombinare le soluzioni dei sottoproblemi (operazione di *combina*)

7.5 Risoluzione delle equazioni di ricorrenza

7.5.1 Metodo iterativo

Questo metodo consiste nel calcolare $T(n)$ in funzione delle sue iterazioni successive.

Esempio

$$\begin{cases} T(1) = 1 \\ T(n) = 1 + T(\frac{n}{2}) \end{cases}$$

$$T\Big(\frac{n}{2}\Big) = T\Big(\frac{n}{4}\Big) + 1 \quad T\Big(\frac{n}{4}\Big) = T\Big(\frac{n}{8}\Big) + 1$$

$$T(n) = T\Big(\frac{n}{4}\Big) + 1 + 1$$

$$T(n) = T\Big(\frac{n}{8}\Big) + 1 + 1 + 1$$

Dunque, alla i-esima iterazione: $T(n) = T\big(\frac{n}{2^i}\big) + i$
Per ottenere il limite asintotico superiore, ci si ferma quando: $\frac{n}{2^i} = 1 \Rightarrow n = 2^i \Rightarrow i = \log_2 n$

$$T(n) = T(1) + \log_2 n = 1 + \log_2 n \Rightarrow T(n) = O(\log_2 n)$$

7.5.2 Metodo dell'esperto

Si definisce il seguente teorema:

$$a, b \in \mathbb{N}, a \geq 1, b \geq 2, c \in \mathbb{R}, \beta \in \mathbb{R}, c \geq 0, \beta \geq 0$$

$$T(n) = \begin{cases} aT\big(\frac{n}{b}\big) + cn^\beta, n > 1 \\ d, n \leq 1 \end{cases}$$

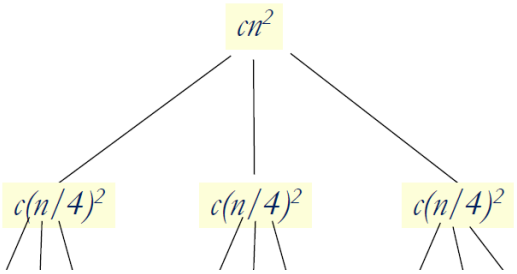
Posto $\alpha = \log_b a = \frac{\log a}{\log b}$, si ha che:

$$T(n) = \begin{cases} \Theta(n^\alpha), \alpha > \beta \\ \Theta(n^\alpha \log_2 n), \alpha = \beta \\ \Theta(n^\beta), \alpha < \beta \end{cases}$$

7.5.3 Metodo dell'albero di ricorsione

Esempio

$$T(n) = \begin{cases} \Theta(1), n \leq c \\ 3T(\frac{n}{4}) + \Theta(n^2), n > c \end{cases}$$



1. si calcola il numero dei nodi per livello: 3^i in questo caso.
2. si calcola il peso di ciascun livello moltiplicando il numero di nodi per livello per il peso di ciascun nodo di quel livello
3. si calcola il numero di livelli: $log_4 n + 1$ in questo caso. Si noti che il 4 è il valore di b dell'equazione di ricorrenza. Il numero dei livelli deve essere dimostrato per induzione.
4. si calcola il numero di foglie: in questo caso 3^{altezza}, dove l'altezza dell'albero è il numero di livelli meno 1.
5. T(n) è pari dunque alla somma dei pesi di $i-1$ livelli più la somma dei pesi delle foglie.

7.5.4 Metodo della sostituzione

Per applicare il metodo di sostituzione:

1. si indovini la soluzione $\Theta(f(n))$ (si consiglia il metodo dell'esperto).
2. dimostrare per induzione la validità del limite superiore, inferiore o entrambi in base alla richiesta.

Esempio

$$\begin{cases} T(1) = 1 \\ T(n) = n + 2T(\frac{n}{2}) \end{cases}$$

Si ha: $T(n) = \Theta(n \log_2 n) \Rightarrow c_1 n \log_2 n \leq T(n) \leq c_2 n \log_2 n$ per il teorema dell'esperto.
Si studia il limite superiore trovando per quali $c > 0$ vale $T(n) \leq c_2 n \log_2 n$:

1. Si prova il caso base tramite il principio dell'induzione.
2. Si utilizza l'ipotesi induttiva: $k < n, T(k) \leq ck \log_2 k$.

Quindi: $T(n) = 2T(\frac{n}{2}) + n \leq 2c\frac{n}{2} \log_2 \frac{n}{2} + n$, e così via.
Per il limite inferiore il procedimento è del tutto analogo:

1. Si prova il caso base tramite il principio dell'induzione.
2. Si utilizza l'ipotesi induttiva: $k < n, T(k) \geq dk \log_2 k$.

Capitolo 8

Algoritmi di ricerca e ordinamento

8.1 Algoritmi di ricerca

Un **algoritmo di ricerca** permette di trovare un elemento avente determinate caratteristiche all'interno di un insieme di elementi ognuno dei quali identificato da una chiave.

8.1.1 Ricerca sequenziale

Consiste nel scorrere tutti gli elementi fino a quando non si trova quello desiderato:

Ricerca sequenziale

```
1 int sequential_search(char v[], int len, char key){
2     int i;
3     for (i=0; i<len; i++){
4         if (v[i] == key){
5             return i;
6         }
7     }
8     return -1;
9 }
```

In questo caso, la funzione $T(n)$ è pari:

- nel caso peggiore: n.
- nel caso migliore 1.
- nel caso medio $\frac{n}{2}$.

8.1.2 Ricerca sequenziale con sentinella

Come sopra, ma evita un confronto: posiziono al termine del vettore l'elemento da cercare:

Ricerca sequenziale con sentinella

```
1  // la funzione chiamante deve assicurare la presenza di un
       elemento vuoto alla fine del vettore
2  int sequential_search_sentinella(char v[], int len, char
       key){
3      int i;
4      v[len] = key;
5      for (i=0; v[i]!=key; i++){  }
6
7      if (i == len) {
8          return -1;
9      }
10     return i;
11 }
```

8.1.3 Ricerca sequenziale su vettore ordinato

Questa ricerca sfrutta il fatto di ricevere un vettore ordinato in input:

Ricerca sequenziale su vettore ordinato

```
1  int sequential_search_order(char v[], int len, char key){
2      int i;
3      for (i=0; key>=v[i] && i<len; i++){
4          if (v[i] == key){
5              return i;
6          }
7      }
8      return -1;
9  }
```

8.1.4 Ricerca binaria

Si presuppone che l'input sia un insieme ordinato di elementi con la possibilità di accesso casuale (*es.* array):

Ricerca binaria

```
1  int binary_search(char v[], int len, char key){
2      int bottom = 0;
3      int top = len - 1;
4      int mid;
```

```
5
6       while (bottom <= top){
7           mid = (top + bottom) / 2;
8           if (v[mid] == key) {
9               return mid;
10          }
11          if (key < v[mid]){
12              top = mid - 1;
13          } else {
14              bottom = mid + 1;
15          }
16      }
17      return -1;
18  }
```

In questo caso, la funzione $T(n)$ è pari:

- nel caso peggiore: $\log_2 n$.
- nel caso migliore 1.
- nel caso medio $\log_2 n$.

8.2 Algoritmi di ordinamento

8.2.1 Insertion sort

L'algoritmo di **insertion sort** permette di riordinare in maniera efficiente un piccolo insieme di elementi. La complessità asintotica dell'algoritmo:

- caso medio: $\Theta(n^2)$
- caso peggiore: $\Theta(n^2)$

Dato un vettore v, l'algoritmo si basa sul costruire (a partire dall'indice zero di v) un sottoarray ordinato.

Insertion sort

```
1   void insertion_sort(char v[], int len){
2       int i, j;
3       char temp;
4       for (i=1; i<len; i++){
5           j = i-1;
6           temp = v[i];
7           while (j>=0 && v[j]>temp){
8               v[j+1] = v[j];
9               j--;
10          }
11          v[j+1] = temp;
12      }
13  }
```

8.2.2 Merge Sort

Viene qui di seguito presentata la versione ricorsiva dell'algoritmo di ordinamento **merge sort**. Esso è definito come un algoritmo *divide et impera*, in quanto:

divide : separa la sequenza di n elementi in 2 da $\frac{n}{2}$.
impera : ordina le due sottosequenze generate da divide.
combina : unisce le due sottosequenze rispettivamente ordinate in un tempo $\Theta(n)$.

Utilizzando l'implementazione ricorsiva, sono indispensabili i casi base:

- dato un vettore da un elemento, esso è gia ordinato.
- dato un vettore di due elementi, o è già ordinato o è necessario invertirne gli elementi.

La complessità asintotica dell'algoritmo:

- caso medio: $\Theta(n \log n)$
- caso peggiore: $\Theta(n \log n)$

Merge Sort

```
1  // passo di combina, con allocazione dinamica della memoria
2  void merge(char v[], int p, int mid, int r){
3      int i, j, k;
4      int len1 = mid-p+1;
5      int len2 = r-mid;
6      char *L = new char[len1+1];
7      char *R = new char[len2+1];
8
9      for (i=0; i<len1; i++){
10         L[i] = v[p+i];
11     }
12     for (i=0; i<len2; i++){
13         R[i] = v[mid+1+i];
14     }
15     L[len1] = R[len2] = (char)'z'; // sentinella
16
17     i = j = 0;
18     for (k=p; k<=r; k++){
19         if (L[i] < R[j]){
20             v[k] = L[i++];
21         } else {
22             v[k] = R[j++];
23         }
24     }
25     delete [] L;
26     delete [] R;
27 }
```

```
28
29 void merge_sort(char v[], int p, int r){
30     // caso base: 1 elemento
31     if (r-p <= 0){
32         return;
33     // caso base: 2 elementi
34     } else if (r-p == 1) {
35         if (v[p] > v[r]){
36             char temp = v[p];
37             v[p] = v[r];
38             v[r] = temp;
39         }
40         return;
41     // caso induttivo
42     } else {
43         int mid = (p+r)/2;
44         merge_sort(v, p, mid);
45         merge_sort(v, mid+1, r);
46         merge(v, p, mid, r);
47     }
48 }
```

8.2.3 Teorema della complessità nel caso pessimo di un algoritmo di ordinamento

La complessità nel caso pessimo di un algoritmo di ordinamento sul posto che confronta e scambia elementi consecutivi è $\Omega(n^2)$. Algoritmi più efficienti richiedono scambi di elementi lontani fra loro.

Capitolo 9

Liste

9.1 Implementazione della struttura

Una **lista** è una struttura di dati astratta che permette di contenere un insieme di elementi. Attraverso la gestione dinamica della memoria non è inoltre necessario specificare la dimensione della struttura al momento della compilazione (attraverso costanti o *define*). A differenza di un array, in cui i propri elementi occupano celle contigue di memoria, gli elementi di una lista possono essere localizzati posizioni diverse dell'heap.
Una lista può essere implementata in due diverse modalità:

- **semplicemente concatenata (sc)**: ogni elemento ha un puntatore all'elemento successivo
- **doppiamente concatenata (dc)**: ogni elemento ha due puntatori: uno all'elemento precedente e l'altro all'elemento successivo

Seppur la presenza di un ulteriore puntatore nell'implementazione *dc* possa sembrare un'inutile ridondanza, in realtà permette di semplificare e ottimizzare alcuni algoritmi.
Entrambe le strutture presentano un metodo di accesso *sequenziale* e non casuale (tipico degli array).

Implementazione lista singolarmente concatenata

```
1  typedef struct Tnodo{
2      Tdato dato;
3      Tnodo *next;
4
5      // costruttori
6      Tnodo(Tdato d);
7      Tnodo(Tdato d, Tnodo *n);
```

```
8 } Tnodo;
9 typedef Tnodo *ListaSemplicementeConcatenata;
```

Implementazione lista doppiamente concatenata

```
1  struct Tnodo {
2      Tdato dato;
3      Tnodo *next;
4      Tnodo *prev;
5
6      // costruttori
7      Tnodo(Tdato);
8      Tnodo(Tdato d, Tnodo *p, Tnodo *n);
9  };
10 typedef Tnodo *ListaDoppiamenteConcatenata;
```

9.2 Ricerca di un elemento

Come anticipato precedentemente, si può accedere ad un elemento di una lista in maniera sequenziale e non casuale. Dunque, nel caso peggiore, la complessità dell'algoritmo è $O(n)$.

Ricerca di un elemento di una lista semplicemente/doppiamente concatenata

```
1  Lista ricerca(const Lista ls, Tdato d){
2      Lista temp = ls;
3      while (!is_empty_lista(temp)){
4          if (temp->dato.isEqual(d)) {
5              return temp;
6          }
7          temp = temp->next;
8      }
9      return NULL;
10 }
```

9.3 Inserimento di un elemento

Nel caso di una lista *sc*, quando la si scorre per inserire un elemento in coda o in ordine, è necessario tenere memoria anche dell'elemento precedente a quello in cui si sta visitando.
È molto oneroso effuare operazioni in coda: è necessario infatti visitare tutta la lista.

Inserimento in testa, in coda, ordinato in una lista *sc*

```
1  Lista inserisci_in_testa(Lista ls, Tdato d){
2      return new Tnodo(d, ls);
3  }
4
5  Lista inserisci_in_coda(Lista ls, Tdato d){
6      Lista temp = ls;
7      if (is_empty_lista(temp)){
8          return inserisci_testa(ls, d);
9      }
10     while (temp->next != NULL){
11         temp = temp->next;
12     }
13     temp->next = new Tnodo(d, NULL);
14     return ls;
15 }
16
17 Lista inserisci_ordinato(Lista ls, Tdato d){
18     Lista temp = ls;
19     Lista pred = NULL;
20     if (is_empty_lista(temp)){
21         return inserisci_testa(ls, d);
22     }
23     while ((!is_empty_lista(temp)) && temp->dato.
           isLowerThan(d)){
24         //isLowerThan=metodo per verificare se dato e'
               minore di d
25         pred = temp;
26         temp = temp->next;
27     }
28     if (is_empty_lista(temp)){
29         pred->next = new Tnodo(d, NULL);
30     } else if (temp==ls) {
31         return inserisci_testa(ls, d);
32     } else {
33         pred->next = new Tnodo(d, temp);
34     }
35     return ls;
36 }
```

Invece, nel caso di una lista doppiamente concatenata, è possibile risalire all'elemento precedente attraverso al campo *prev*:

Inserimento in testa, in coda, ordinato in una lista *dc*

```
1  ListaDoppia inserisci_testa_doppia(ListaDoppia ls,Tdato d){
2      ListaDoppia temp = new Tnodo(d, NULL, ls);
3      if (ls != NULL){
4          ls->prev = temp;
```

```
5       }
6       return temp;
7   }
8
9   ListaDoppia inserisci_coda_doppia(ListaDoppia ls, Tdato d){
10      if (ls == NULL){
11          return inserisci_testa_doppia(ls, d);
12      }
13      ListaDoppia temp = ls;
14      while (temp->next != NULL){
15          temp = temp->next;
16      }
17      temp->next = new Tnodo(d, temp, NULL);
18      return ls;
19  }
20
21  ListaDoppia inserisci_ord_doppia(ListaDoppia ls, Tdato d){
22      if (ls == NULL){
23          return inserisci_testa_doppia(ls, d);
24      }
25      ListaDoppia temp = ls;
26      ListaDoppia old = NULL;
27      while ((temp!=NULL) && (temp->dato.isLowerThan(d))){
28          old = temp;
29          temp = temp->next;
30      }
31      Tnodo *t = NULL;
32      if (temp == NULL){
33          t = new Tnodo(d, old, NULL);
34          old->next = t;
35      } else {
36          if (temp->prev != NULL){
37              t = new Tnodo(d, temp->prev, temp);
38              temp->prev->next = t;
39              temp->prev = t;
40          } else {
41              t = new Tnodo(d, NULL, temp);
42              temp->prev = t;
43              return t;
44          }
45      }
46      return ls;
47  }
```

9.4 Rimozione di un elemento

Come per l'inserimento, è molto oneroso effuare operazioni in coda: è necessario infatti visitare tutta la lista.

Eliminazione di un elemento, di un elemento in testa e in coda con lista *sc*

```
1  Lista delete_el(Lista ls, Tdato d){
2      Lista temp = ls;
3      Lista pred = NULL;
4      while (!is_empty_lista(temp)){
5      if (temp->dato.isEqual(d)){
6          if (pred != NULL){
7              pred->next = temp->next;
8              delete temp;
9              return ls;
10         } else {
11             Lista res = temp->next;
12             delete temp;
13             return res;
14         }
15     }
16     pred = temp;
17     temp = temp->next;
18     }
19 }
20
21 Lista delete_testa(Lista ls){
22     if (ls != NULL) {
23         Lista temp = ls;
24         ls = ls->next;
25         delete temp;
26     }
27     return ls;
28 }
29
30 Lista delete_coda(Lista ls){
31     if (ls != NULL) {
32         Lista temp = ls;
33         Lista pred = NULL;
34         while (!is_empty_lista(temp->next)){
35             pred = temp;
36             temp = temp->next;
37         }
38         if (pred == NULL){
39             return delete_testa(ls);
40         }
41         delete temp;
42         pred->next = NULL;
```

```
43      }
44      return ls;
45  }
```

Eliminazione di un elemento, di un elemento in testa e in coda con lista *dc*

```
1   ListaDoppia delete_el_doppia(ListaDoppia ls, Tdato d){
2       if (ls != NULL){
3           if (ls->dato.isEqual(d)){
4               return delete_testa_doppia(ls);
5           }
6           ListaDoppia temp = ls;
7           while ((temp!=NULL) && (!temp->dato.isEqual(d))){
8               temp = temp->next;
9           }
10          if (temp != NULL){
11              if (temp->next != NULL){
12                  temp->next->prev = temp->prev;
13              }
14              temp->prev->next = temp->next;
15              delete temp;
16          }
17      }
18      return ls;
19  }
20
21  ListaDoppia delete_testa_doppia(ListaDoppia ls){
22      ListaDoppia temp = ls->next;
23      delete ls;
24      temp->prev = NULL;
25      return temp;
26  }
27
28  ListaDoppia delete_coda_doppia(ListaDoppia ls){
29      if (ls != NULL){
30          ListaDoppia temp = ls;
31          while (temp->next != NULL){
32              temp = temp->next;
33          }
34          if(temp->prev != NULL){
35              temp->prev->next = NULL;
36              delete temp;
37          } else {
38              delete temp;
39              return init_doppia();
40          }
41      }
42      return ls;
43  }
```

9.5 Calcolare la lunghezza di una lista

Sia per le liste *sc* e *dc* l'implementazione è la medesima:

```
1 unsigned int length_lista(const Lista ls){
2     unsigned int len = 0;
3     Lista temp = ls;
4     while (!is_empty_lista(temp)){
5         len++;
6         temp = temp->next;
7     }
8     return len;
9 }
```

Capitolo 10

Liste circolari

Il concetto di **lista circolare** è del tutto analogo a quello di una lista semplicemente o doppiamente concatenata: l'unica differenza è data dal fatto che l'ultimo elemento della lista è legato al primo. Dunque, bisogna porre attenzione a come scorrere la lista per evitare di incorrere in loop infiniti.

10.1 Implementazione della struttura

Principalmente si possono distinguere due implementazioni per la lista circolare:

- attraverso liste semplicemente o doppiamente concatenate.
- attraverso un vettore.

Implementazione di una lista circolare in C++

```
1  typedef struct ListaCircolare{
2      // campi
3      Tdato v[MAX];
4      unsigned int n;      // numero di elementi nella lista
5      unsigned int testa; // indice dell'elemento in testa
6      unsigned int coda;  // indice dell'elemento successivo
           alla coda
7
8      // costruttore
9      ListaCircolare(){
10         n = testa = coda = 0;
11     }
12
13     // metodi
14     ...
15 } ListaCircolare;
```

10.2 Ricerca di un elemento

Metodo per la ricerca di un elemento in una lista circolare

```
1 bool contains(Tdato d){
2     int j;
3     for (j=0; j<n; j++){
4         if (v[(testa+j) % MAX] == d){
5             return true;
6         }
7     }
8     return false;
9 }
```

10.3 Inserimento di un elemento

Metodo per l'inserimento in testa in una lista circolare

```
1  void insertFirst(Tdato d){
2      if (!isFull()){
3          testa = (testa-1) % MAX;
4          if (testa < 0){
5              testa += MAX;
6          }
7          v[testa] = d;
8          n++;
9      }
10 }
```

Metodo per l'inserimento in coda in una lista circolare

```
1 void insertLast(Tdato d){
2     if (!isFull()){
3         v[coda] = d;
4         coda = (coda+1) % MAX;
5         n++;
6     }
7 }
```

10.4 Rimozione di un elemento

Metodo per la rimozione della testa da una lista circolare

```
1 void removeFirst(){
```

```
2       if (!isEmpty()){
3           testa = (testa+1) % MAX;
4           n--;
5       }
6   }
```

Metodo per la rimozione della coda da una lista circolare

```
1   void removeLast(){
2       if (!isEmpty()){
3           coda = (coda-1) % MAX;
4           if (coda < 0){
5               coda += MAX;
6           }
7           n--;
8       }
9   }
```

10.5 Calcolare la lunghezza della lista

Si può notare com'è immediato restituire la lunghezza di una lista circolare:

Metodo per restituire la lunghezza di una lista circolare

```
1   int length(){
2       return n;
3   }
```

10.6 Controllo lista piena/vuota

In modo analogo, anche qui il campo n (contenente il numero di elementi) torna utile:

Metodo per controllare se la lista circolare è vuota

```
1   bool isEmpty(){
2       return n == 0;
3   }
```

Metodo per controllare se la lista circolare è piena

```
1   bool isFull(){
2       return n == MAX;
3   }
```

Capitolo 11

Stack e code

11.1 Stack

11.1.1 Definzione e implementazione

Lo **stack** (in italiano **pila**) è una struttura dati astratta di tipo **LIFO** (*Last In First Out*).
Sia per definizione, che per problemi di ottimizzazione, lo stack prevede un solo punto di accesso per l'inserimento e la rimozione di un elemento: la testa.
La pila può essere implementata con:

- *array*: gestione statica della memoria; limite massimo di dimensione.
- *liste*: gestione dinamica della memoria; occupazione variabile e non limitata.

11.1.2 Funzioni

Le principali funzioni che uno stack deve fornire sono:

- *init()*: inizializza la struttura.
- *push()*: inserisce in elemento in testa con complessità $O(1)$.
- *pop()*: elimina un elemento dalla testa con complessità $O(1)$.
- *top()*: restituisce l'elemento in testa.
- *isEmpty()*: verifica se lo stack è vuoto.
- *isFull()*: verifica se lo stack è pieno.
- *remove()*: rimozione di un elemento (utilizzando un secondo stack).

11.1.3 Applicazioni

Alcuni utilizzi dello stack riguardano:

- trasformazione da espressioni in notazione infissa a postfissa.
- valutazione di espressioni in notazione postfissa.
- implementazione iterativa di algoritmi riguardo gli alberi binari di ricerca.
- struttura degli ambienti di chiamata delle funzioni in runtime.

11.2 Code

11.2.1 Definizione

La **coda** (in inglese **queue**) è una struttura dati astratta di tipo **FIFO** (*First In First Out*).
Sia per definizione, che per ottimizzare sia l'estrazione che l'inserimento, la queue prevede 2 punti di accesso:

- *front* (in italiano *testa*) per l'estrazione.
- *rear* (in italiano *coda*) per l'inserimento.

11.2.2 Implementazione

La coda può essere implementata attraverso:

- liste concatenate.
- liste circolari.

Implementazione di una coda attraverso liste concatenate

```
1 typedef struct Queue {
2     Tnodo *head;
3     Tnodo *tail;
4 } Queue;
```

11.2.3 Funzioni

Le principali funzioni che una queue deve fornire sono:

- *enqueue()*: inserisce un elemento nel rear con complessità $O(1)$.
- *dequeue()*: estrae un elemento dal front con complessità $O(1)$.
- *queueIsEmpty()*: verifica se la coda è vuota.
- *queueIsFull()*: verifica se la coda è piena (se implementata con liste circolari).
- *front()*: restituisce l'elemento in front.

Capitolo 12

Alberi binari di ricerca

12.1 Definizioni di albero

- Un *albero* è un grafo non orientato, connesso e privo di cicli.
- Attraverso la *ricorsione*:
  - l'insieme vuoto ϕ è un albero binario.
  - T_s e T_d sono alberi binari ed r è un nodo: la terna (r, T_s, T_d) è un albero binario.
- Un *albero binario di ricerca* è un albero in cui ogni nodo ha al più due figli tale che ogni nodo è maggiore (o uguale) dei valori dei nodi del suo sottoalbero sinistro e minore (o uguale) dei valori dei nodi del suo sottoalbero destro.

Alcune sigle per identificare un albero binario di ricerca sono *ABR* e *BST* (dall'inglese *binary search tree*).

Implementazione di un albero binario di ricerca

```
1 typedef struct Tnodo {
2 Tdato dato;
3 Tnodo *sx;
4 Tnodo *dx;
5 Tnodo *parent; //non utilizzato in lab
6 } Tnodo;
7 typedef ABR *Tnodo;
```

12.2 Proprietà

1. Un ABR non ha una rappresentazione univoca. Alcuni rappresentazioni infatti risultano più bilanciate di altre.

2. Un ABR si dice *bilanciato* se i suoi nodi sono equamente distribuiti, senza eccedere in altezza.
3. *Altezza*: lunghezza massima tra radice e una foglia, ossia $h = n_{\text{livelli}} - 1$.
4. *Foglia*: nodo senza figli.
5. *Radice*: nodo senza padre.
6. *Cammino* di un ABR: somma per ciascun nodo la sua distanza dalla radice.
7. Il percorso dalla radice ad ogni foglia è unico.

12.3 Algoritmi

12.3.1 Visita in preordine

Visita in preordine di un ABR

```
1 void visita_preordine(BST tree){
2     if (tree != NULL){
3         tree->dato.print();
4         visita_preordine(tree->sx);
5         visita_preordine(tree->dx);
6     }
7 }
```

12.3.2 Visita in ordine

Attraverso una stampa della visita in ordine si ottiene un elenco ordinato.

Visita in ordine di un ABR

```
1 void visita_ordine(BST tree){
2     if (tree != NULL){
3         visita_ordine(tree->sx);
4         tree->dato.print();
5         visita_ordine(tree->dx);
6     }
7 }
```

12.3.3 Visita in postordine

Visita in postordine di un ABR

```
1 void visita_postordine(BST tree){
2     if (tree != NULL){
3         visita_postordine(tree->sx);
```

```
4           visita_postordine(tree->dx);
5           tree->dato.print();
6       }
7   }
```

12.3.4 Visita in level-ordine

Nella visita in level-ordine si percorrono i nodi per ciascun livello.

12.3.5 Ricerca di un elemento

In maniera ricorsiva, cerca fra i sottoalberi di un nodo. Nel caso peggiore ha complessità $O(h)$.

Ricerca ricorsiva di un elemento in un ABR

```
1   Tnodo *search_element(BST tree, Tdato d){
2       if (tree->dato.isEqual(d)){
3           return tree;
4       } else if (d.isLowerThan(tree->dato)){
5           return search_element(tree->sx, d);
6       } else {
7           return search_element(tree->dx, d);
8       }
9   }
```

12.3.6 Ricerca del minimo

Si continua iterativamente a percorrere il sottoalbero sinistro. Nel caso peggiore ha complessità $O(h)$.

Ricerca del minimo in un ABR

```
1   Tdato minimo(BST tree){
2       BST temp = tree;
3       while (temp->sx != NULL){
4           temp = temp->sx;
5       }
6       return temp->dato;
7   }
```

12.3.7 Ricerca del massimo

Si continua iterativamente a percorrere il sottoalbero destro. Nel caso peggiore ha complessità $O(h)$.

Ricerca del massimo in un ABR

```
1 Tdato massimo(BST tree){
2     BST temp = tree;
3     while (temp->dx != NULL){
4         temp = temp->dx;
5     }
6     return temp->dato;
7 }
```

12.3.8 Ricerca del successore e predecessore

Per implementare i seguenti algoritmi, nell'implementazione dell'ABR si necessita del riferimento ai genitori. Nel caso peggiore ha complessità $O(h)$.

12.3.9 Calcolo dell'altezza

L'algoritmo consiste nel percorrere tutti i percorsi, cercandone il più lungo.

Calcolo dell'altezza in un ABR

```
1 int altezza(BST tree){
2     if (tree == NULL){
3        return -1; // dovuto alla definizione di altezza
4     } else {
5        return max(1+altezza(tree->sx),1+altezza(tree->dx));
6     }
7 }
```

12.3.10 Inserimento di un elemento

Nel caso peggiore ha una complessità $O(h)$.

Inserimento di un elemento in un ABR

```
1  BST insert_node(BST tree, Tdato d){
2      if (tree == NULL){
3          return new Tnodo(d, NULL, NULL);
4      } else {
5          insert_ric(tree, d);
6          return tree;
7      }
8  }
9
10 // la funzione deve restituire un BST, in quanto non basta
       creare un nuovo nodo: bisogna anche collegarlo al
       precedente
```

```
11 BST insert_ric(BST tree, Tdato d){
12     if (tree == NULL){
13         return new Tnodo(d, NULL, NULL);
14     } else {
15         if (d.isGreaterThan(tree->dato)){
16             tree->dx = insert_ric(tree->dx, d);
17         } else {
18             tree->sx = insert_ric(tree->sx, d);
19         }
20     }
21     return tree;
22 }
```

12.3.11 Conta numero di nodi

Conta numero di nodi di un ABR

```
1 int conta_nodi(BST tree){
2     if (tree == NULL){
3         return 0;
4     } else {
5         return 1 + conta_nodi(tree->sx) + conta_nodi(tree->
              dx);
6     }
7 }
```

12.3.12 Cancellazione di un elemento

Di seguito si descrive l'algoritmo per la *cancellazione* di un elemento z:

- caso 1: se z non ha figlio *sx*:
  1. se dx != NULL, sostituisci *dx* a z.
  2. se ha entrambi i figli *NULL*, nessun problema
- caso 2: se z ha solo figlio *sx*, sostituisci *sx* a z.
- caso 3: se z ha due figli. Sia y il successore di z (y ha necessariamente sottoalbero sinistro nullo e si trova nel sottalbero destro di z):
  1. se y è il diretto figlio destro di z: sostituisci y (mantenendo il suo sottoalbero destro) a z.
  2. se y non è il diretto figlio destro di z: scambia y con il diretto figlio destro di z; sostituisci y (mantenendo il suo sottoalbero destro) a z.

Capitolo 13

Hashing

La **tabella di hash** è una struttura che permette di effettuare ricerca e inserimento $O(1)$ sotto *ipotesi ragionevoli*: non si basa su confronti e mantenimento dell'ordine dei dati. Se mal costruita, la ricerca nel caso peggiore è $\Theta(n)$.
La ricerca consiste nell'associare ad ogni *chiave* i suoi rispettivi *dati satellite*. Un semplice esempio è un database finalizzato a memorizzare dei dispositivi: ciascun record infatti ha associato una chiave (es. MAC Address) e dei dati satellite (es. marca, caratteristiche, ecc.). L'insieme di chiavi U deve essere piccolo.

13.1 Tavole a indirizzamento diretto

Il metodo attraverso una **tavola a indirizzamento diretto** utilizza un vettore V di puntatori, di dimensione $|U|$.
Quando si inserisce un elemento nella tavola, viene generato dinamicamente il corrispondente putatore ai dati satellite, che viene inserito nella cella di V corrispondente alla propria chiave. Dunque vi è una corrispondenza biunivoca tra celle di V e le chiavi possibili.

Vantaggi

- le operazioni di inserimento, ricerca e cancellazione risultano di costo unitario.

Svantaggi

- il costo oneroso iniziale nel settare a $NULL$ tutti gli elementi di V.

- se la cardinalità di U è troppo elevata, non si può utilizzare la tavola ad indirizzamento diretto in quanto occuperebbe troppa memoria.

13.2 Tabella di Hash

Con la **tabella di hash** si introduce la **funzione di hashing** F: riceve in input una chiave c e restituisce l'indice del vettore V che punta ai dati satellite di c. A differenza della tavola a indirizzamento diretto, la chiave non viene più utilizzata come indice di V.

Collisioni Seppur il numero di chiavi utilizzate sia piccolo rispetto a $|U|$, F può presentare delle collisioni. Si incontra una *collisione* quando a chiavi diverse vengono assegnati gli stessi dati satellite. Minore è la probabilità di trovare collisioni, maggiore è qualità di F.

Chaining La tecnica di *chaining* permette di risolvere il problema delle collisioni. Ad ogni elemento dell'immagine di F viene associata una lista concatenata L di chiavi (con i corrispondenti dati satellite): nel caso della ricerca, prima si applica F e poi, attraverso un algoritmo $\Theta(n)$, si cerca l'elemento con la chiave voluta all'interno della lista associata.
Il *fattore di carico alpha* $= \frac{n}{m}$, dove n rappresenta il numero di chiavi e m rappresenta il numero di celle della tavola, è pari alla lunghezza media di una lista L. Nel caso peggiore, la ricerca nella lista $\Theta(1 + \alpha)$. Dunque, più ridotte solo le liste, più veloce è la ricerca. La soluzione ottimale è trovare una funzione F che distribuisce le chiavi in maniera più uniforme possibile.
Un esempio riguarda la seguente funzione di hash: $h(k) = k\%m + 1$. Se m si trova vicino ad una potenza di 2, allora $h(k)$ non ha una distribuzione uniforme delle proprie chiavi.

Funzione di hash universale Ogni funzione di hash ha delle distribuizioni di probabilità non uniformi per alcuni input. Si utilizza perciò la *funzione di hash universale* se per ogni coppia di chiavi distinte vi sono al più $\frac{|H|}{m}$ funzioni di hash che causano collisione fra le due chiavi. H è l'insieme delle funzioni di hash utilizzate, da cui se ne sceglie una casualmente.

Appendice A

Tabella di precedenza degli operatori

Operatori	Associatività
`()`, `[]`, `.`, `->`	da sinistra a destra
`++`, `--`, `+`, `-`, `!`, `~`, `(type)`, `*`, `&`, `sizeof`	da destra a sinistra
`*`, `/`, `%`,	da sinistra a destra
`+`, `-`	da sinistra a destra
`<<`, `>>`	da sinistra a destra
`<`, `<=`, `>`, `>=`	da sinistra a destra
`==`, `!=`	da sinistra a destra
`&`	da sinistra a destra
`^`	da sinistra a destra
`|`	da sinistra a destra
`&&`	da sinistra a destra
`||`	da sinistra a destra
`?:`	da destra a sinistra
`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `&=`, `^=`, `|=`	da destra a sinistra
`,`	da sinistra a destra