Laboratorio di algoritmi e strutture dati

G. Melideo

CdL in Informatica

A.A. 2009/2010

Indice

1. Presentazione del corso
   - Obiettivi del corso
   - Prerequisiti
   - Modalità d'esame
   - Altre informazioni
2. Linguaggio C: richiami
   - Indirizzi e puntatori
   - Regole di visibilità delle variabili
   - Il passaggio dei parametri
   - Array e puntatori
   - Le stringhe
   - Allocazione dinamica della memoria
3. Linguaggio C: nozioni avanzate
   - Array allocati dinamicamente
   - L'indirettezza multipla
   - Le strutture
   - typedef
   - Unioni
   - Generatore di numeri pseudo casuali
   - Puntatori a funzione
4. Le liste
   - Definizione di liste lineari semplici
   - Operazioni sulle liste lineari semplici
5. La ricorsione
6. Introduzione ai tipi di dato astratti
   - Tipo di dato astratto
   - Applicazione a "sequenze di elementi"
   - Sviluppo di un programma
   - Organizzazione di ADT in C
7. ADT Stack
   - Definizione
   - ADT `Stack[Item]`
   - Caso d'uso: valutazione espressione aritmetica postfissa
   - Caso d'uso: da infissa a postfissa
   - Postscript
8. ADT Coda
   - Definizione
   - ADT `Queue[Item]`
9. Alberi binari
   - Definizione
   - Rappresentazioni di alberi binari

Progettazione di algoritmi e strutture dati

Obiettivi congiunti con il corso di Algoritmi e Strutture Dati (ASD), insieme al quale LASD costituisce il corso integrato di **Algoritmi e Strutture Dati con Laboratorio** (12 CFU):

- Introdurre ai fondamenti della teoria degli algoritmi, delle strutture dati e all'analisi della complessità computazionale di programmi
- Familiarizzare lo studente con le principali problematiche e tecniche relative alla progettazione degli algoritmi e delle strutture dati
  - Gli algoritmi fondamentali studiati rappresentano la base di programmi più grandi in molte aree applicative

Studio delle prestazioni

- Stimolare la comprensione attraverso l'implementazione e la sperimentazione di programmi e la successiva applicazione a problemi reali
  - il linguaggio impiegato per tutte le implementazioni è il C (standard ANSI)
  - lo stile adottato assicura una semplice traduzione dei programmi in qualunque altro moderno linguaggio di programmazione
- Confrontare diversi algoritmi che risolvono lo stesso problema
- Confrontare prestazioni teoriche e prestazioni sperimentali

Abilità

Al termine del corso si ritiene che lo studente sia in grado di:

- Risolvere algoritmicamente problemi classici
- Scegliere motivatamente le strutture dati adatte ad ottenere soluzioni computazionalmente efficienti
- Realizzare in C gli algoritmi progettati
- Scrivere implementazioni portabili, eleganti, senza trascurare gli aspetti di efficienza durante le fasi di sviluppo del codice

Prerequisiti

Si assume che lo studente abbia acquisito le nozioni di base della programmazione e sia in grado di implementare semplici algoritmi in linguaggio C.
Pertanto, si consiglia fortemente di dedicarsi allo studio del corso di ASDL solo dopo aver sostenuto con esito positivo gli esami di dei corsi di:

- Fondamenti di Programmazione con Laboratorio
- Laboratorio di Programmazione II (propedeuticità)

Modalità d'esame

La prova d'esame prevede il superamento di:

- Una prova scritta integrata (prove intermedia e conclusiva oppure unica prova totale) che verte sugli argomenti di entrambi i corsi di ASD e LASD (la prova intermedia è riservata agli studenti in corso)
- Una prova orale obbligatoria che verte sugli argomenti del corso di ASD
- Una prova orale a discrezione della docente che verte sugli argomenti del corso di LASD
- Le iscrizioni alle prove d'esame si effettuano online attraverso il sistema di prenotazione di Ateneo

Sillabo e testi

- Sillabo del corso, orario di ricevimento ed altre informazioni sono disponibili sulla home page del corso
  http://www.di.univaq.it/melideo/lab-alg.html
- Libro di testo: *Algoritmi in C*, III ed., di Robert Sedgewick, edito da Addison-Wesley.
- Altri testi:
  - *C Corso Completo di Programmazione*, II ed., H.M.Deitel, P.J.Deitel, edito da Apogeo.
  - *Il Linguaggio C*, II ed., B.W.Kernighan, D.M.Ritchie, edito da Pearson Education Italia.
  - *C La Guida Completa*, III ed., Herbert Schildt, edito da MacGraw Hill.
  - *Linguaggio C*, II ed., A.Bellini, A.Guidi, edito da MacGraw Hill.

Le variabili puntatore

- Gli indirizzi di memoria (puntatori) formano un insieme di valori che possono essere manipolati
- Le variabili puntatore possono essere dichiarate nei programmi ed utilizzate per assumere come valori indirizzi di memoria
- Per indicare che una variabile è un puntatore, nella sua dichiarazione si antepone al nome della variabile un asterisco *. Esempio:

```
int *pa, *pb, *pc;
float *x, *y;
```

Operatori di indirizzamento

- L'operatore di indirizzo & applicato ad una variabile di qualunque tipo restituisce l'indirizzo di memoria in cui tale variabile è allocata

  &x è l'indirizzo di memoria di x

- L'operatore di indirizzamento indiretto * (o dereferenziazione) applicato ad un puntatore (un indirizzo di memoria) restituisce il valore memorizzato in tale locazione
- * è l'operatore inverso di &

Esempio

```
int a, b, *pa=NULL, *pb;
a=3; /* a contiene 3 */
pa=&a; pb=&b; /* pa punta ad a, pb punta a b */
*pb=*pa; /* b contiene 3 */
b=10;
pb=pa; /* pa e pb puntano ad a; b contiene 10 */
```

Operatore `sizeof`

- Operatore unario eseguito al momento della compilazione
- Restituisce la lunghezza in byte di una variabile oppure di uno specificatore di tipo
- Il linguaggio C definisce (con typedef) un tipo particolare chiamato `size_t` che tecnicamente costituisce il tipo del valore restituito dal `sizeof` (`size_t` corrisponde a grandi linee ad un intero `unsigned`)
- `sizeof` aiuta a generare codice trasportabile, legato alle dimensioni dei tipi standard del linguaggio.

Aritmetica dei puntatori

- In C è possibile usare l'addizione e la sottrazione con argomenti puntatore, consentendo di attraversare aree di memoria omogenee (contenenti valori dello stesso tipo)
- Quando si aggiunge k ad un puntatore (a un tipo T) si incrementa l'indirizzo di memoria del puntatore di $k * sizeof(T)$ caratteri
- La differenza tra due puntatori di tipo T restituisce il numero di elementi di tipo T contenuti nell'intervallo di memoria tra i due puntatori
- È possibile usare gli operatori di incremento e decremento per incrementare o decrementare di uno (nel senso di un elemento) il contenuto della variabile puntatore.

Dichiarazione di variabili

In C, come per ogni linguaggio che possiede il controllo dei tipi, ogni variabile deve essere dichiarata prima del suo utilizzo per permettere il controllo della consistenza

Le variabili possono essere dichiarate:

- all'interno di un blocco (variabili locali)
- nella definizione dei parametri formali delle funzioni
- all'esterno di tutte le funzioni (variabili globali)

Variabili locali

Le variabili locali dichiarate in un blocco

- hanno una visibilità locale a quel blocco.
- non possono essere utilizzate direttamente fuori dal blocco, vengono create quando si entra nel blocco e vengono distrutte al termine dell'esecuzione dello stesso

Quando una variabile dichiarata in un blocco più interno ha lo stesso nome di una variabile dichiarata in un blocco più esterno che lo contiene, la variabile del blocco interno maschera l'esistenza della variabile più esterna. La variabile esterna tuttavia continua ad esistere e sarà di nuovo disponibile fuori dal blocco.

Parametri formali

- Se una funzione deve utilizzare degli argomenti deve dichiarare le variabili (i parametri formali della funzione) che accoglieranno i valori passati come argomenti
- I parametri formali si comportano come una qualunque altra variabile locale alla funzione, quindi vengono anch'essi distrutti all'uscita della funzione

Variabili globali

Le variabili globali dichiarate all'esterno di ogni funzione:

- hanno una validità globale sull'intero programma e possono essere utilizzate in ogni punto del codice
- sono utili quando quando più funzioni devono utilizzare gli stessi dati
- vanno *evitate* se non necessarie in quanto:
  - conservano il proprio valore durante l'intera esecuzione del programma
  - aumentano la probabilità di errori a causa di effetti collaterali sconosciuti ed indesiderati
  - contribuiscono a diminuire la portabilità e la riutilizzabilità delle funzioni

Il passaggio dei parametri

Esistono tre modalità fondamentali:

1. mediante variabili globali (fortemente sconsigliato)
2. passando il valore di variabili locali
    - quando la funzione deve *fare uso* del valore di una variabile definita in un'altra funzione, senza la necessità di modificare tale valore nella variabile originale e senza dover accedere a locazioni di memoria adiacenti
3. passando l'indirizzo di variabili locali
    - quando la funzione deve *modificare* il valore di una variabile definita in un'altra funzione e quando deve accedere a locazioni di memoria adiacenti a quella della variabile ricevuta come parametro
    - Il C passa gli argomenti alle funzioni solo tramite *chiamate per valore*, quindi se si intende passare un indirizzo lo si deve fare esplicitamente indicandolo con l'operatore & o utilizzando una variabile puntatore di cui verrà passato il valore

Esempio

```
#include<stdio.h>

void prova_a_cambiarla(int);

int main(void){
    int x=100;
    printf("%d\n", x); /* viene stampato 100 */
    prova_a_cambiarla(x);
    printf("%d\n", x); /* viene stampato 100 */
    return 0;
}

void prova_a_cambiarla(int x){x=0;}
```

Esempio

```
#include<stdio.h>

void prova_a_cambiarla2(int *);

int main(void){
    int x=100;
    printf("%d\n", x); /* viene stampato 100 */
    prova_a_cambiarla2(&x);
    printf("%d\n", x); /* viene stampato 0 */
    return 0;
}

void prova_a_cambiarla2(int*x){*x=0;}
```

Array

- Sono collezioni di variabili tutte dello stesso tipo.
- La dichiarazione è effettuata indicando un identificatore e una *n*-pla di dimensioni (nel caso di array *n*-dimensionali).

  Forma generale della dichiarazione: $tipo\ nome[d_1] \ldots [d_n]$

  ```
  int V[4]; /* dichiarazione array di dimensione 4 */
  char V[2][3]; /* dichiarazione matrice */
  ```

- È possibile definire array di lunghezza variabile, ossia array le cui dimensioni sono specificate da un'espressione valida, incluse le espressioni il cui valore è noto solo run-time. Solo gli array locali possono essere di lunghezza variabile.

Array

- Gli array sono memorizzati in locazioni di memoria contigue
- Le matrici sono considerate array di array e sono memorizzate disponendo le righe una di seguito all'altra
- Ogni variabile è accessibile tramite l'identificatore, comune a tutte, specificandone la posizione con n indici, da 0 a $d_i - 1$.
  Ad esempio:

  `V[0],...,V[`$d_1 - 1$`]`

  `V[0][0],...,V[0][`$d_2 - 1$`],...,V[`$d_1 - 1$`][`$d_2 - 1$`]`

Array e puntatori

- Il nome di un array è un indirizzo di memoria (costante)
- Se V è un array monodimensionale si ha un'equivalenza d'uso e di notazione (la differenza è nella modalità di allocazione della memoria)
- Se M è una matrice di *n* righe ed *m* colonne, la formula per individuare la locazione di memoria dell'elemento M[i][j](di qualunque tipo sia la matrice) è la seguente:

  `&M[0][0]+i*m+j`

- Espressioni equivalenti per l'accesso agli elementi, supponendo p=&V[0]; (anche p=V;) :

  V[i], *(V+i), p[i], *(p+i)

Array come parametri di funzione

Nel passaggio di un array ad una funzione è possibile soltanto passare l'indirizzo di un elemento dell'array stesso, indipendentemente dal tipo di notazione utilizzata.

Prototipi equivalenti:

- `double somma(double a[], int n);`
- `double somma(double *a, int n);`

Se V è un array di double, possibili chiamate sono: `somma(&V[0],88);` `somma(V,88);` `somma(&V[i],5);` `somma(V+i,5);`

Le stringhe

- Una variabile stringa può essere dichiarata come array di caratteri o come puntatore a carattere:
  - `char s[N];`
    in tale caso vengono allocati N byte per contenere i caratteri.
  - `char *s;`
    in tale caso viene riservata l'area per l'indirizzo. Lo spazio effettivo per i caratteri andrà allocato esplicitamente in altro modo
- Le stringhe devono essere terminate con il carattere '\0', dunque `char s[N];` permette di allocare una stringa di N-1 caratteri
- Con una dichiarazione `char *p;` l'assegnamento a p di una stringa non ne provoca una copia ma solo l'attualizzazione dell'indirizzo

Le stringhe

- Se la stringa è dichiarata come array (`char s[];`), l'assegnamento ad `s` di una stringa risulterebbe come un tentativo per modificare l'indirizzo dell'array
- Analogamente il confronto tra due variabili stringa non è il confronto carattere per carattere delle stringhe, bensì il confronto fra i due indirizzi
- La modifica delle stringhe costanti è vietata dal linguaggio, per cui il seguente programma potrebbe non funzionare:

```
main(){
    char *p, *q;
    q=p="giovanna";
    p[1]='c';
    printf("%s %s%s\n",p,q,"giovanna");
}
```

Allocazione dinamica della memoria

- Consente un'efficace gestione della memoria a tempo di esecuzione
- Si usa generalmente per la gestione di dati di cui non è nota a priori la dimensione
- Il C fornisce le funzioni `calloc()` e `malloc()` nella libreria standard `stdlib.h` per creare dinamicamente spazio di memoria per variabili
  - `calloc()` sta per *allocazione contigua*
  - `malloc()` sta per *allocazione di memoria*

Allocazione dinamica della memoria

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

Restituisce il puntatore al primo byte dell'area di memoria allocata di dimensione `size`

```
void *calloc(size_t n, size_t elsize);
```

Riserva la memoria per `n` elementi contigui di dimensione `elsize` e restituisce il puntatore al primo byte dell'area di memoria allocata. Tutti i bit all'interno di tale spazio sono posti a zero.

malloc() e calloc()

- Restituiscono NULL se la memoria libera nello heap non è sufficiente per soddisfare la richiesta
- Si usano in genere insieme alla funzione C sizeof()
- Vengono sempre seguite da un operatore di *cast* per restituire un puntatore del tipo desiderato

```
int *p, *q, n;
p = (int *)malloc(sizeof(int));
q = (int *)malloc(sizeof(*q));
p = (int *)calloc(n, sizeof(int)); /* usa p come array */
```

free()

- Lo spazio allocato dinamicamente non viene restituito al sistema all'uscita delle funzioni.
- La memoria va restituita esplicitamente al sistema mediante la funzione `free()`

```
void free(void *ptr);
```

- `ptr` deve essere un puntatore precedentemente allocato con una delle funzioni di allocazione dinamica
- `free()` libera l'area puntata da `ptr`.

realloc()

```
void *realloc(void *ptr, size_t size);
```

- Fa diventare di dimensione `size` l'area puntata da `ptr` allocata in precedenza
- Viene restituito il puntatore al primo byte dell'area riallocata
- Quando il valore di `ptr` è `NULL`, è allocato un nuovo blocco di memoria di `size` bytes
- Se `size` vale 0, è chiamata la corrispondente funzione `free()` per liberare la memoria puntata da `ptr`

realloc()

In generale `realloc()` rialloca spazio come segue:

- Riduce o estende la dimensione del blocco di memoria allocato puntato da `ptr`
- Se non esiste un blocco abbastanza grande di memoria non allocata immediatamente seguente `ptr`:
  - alloca un nuovo blocco
  - copia il contenuto dell'area puntata da `ptr` nel nuovo blocco e libera il vecchio blocco. Per questa ragione non dovrebbero essere mantenuti puntatori al vecchio blocco di memoria, che in caso di allocazione di un nuovo blocco punterebbero a memoria liberata

realloc()

- Remark: `realloc()` restituisce `NULL` quando la memoria puntata da `ptr` non può essere riallocata
- In questo caso la memoria puntata da `ptr` non è liberata, per cui bisogna assicurarsi di mantenere un puntatore al vecchio blocco di memoria

```
buffer = (char *) realloc(buffer, 100);
```

- Se la funzione fallisce, `buffer` sarà posto a `NULL` e non punterà al vecchio blocco di memoria. Se `buffer` è l'unico puntatore al blocco di memoria, allora sarà perso l'accesso ad essa.

Array allocati dinamicamente

È possibile allocare memoria utilizzando le funzioni `calloc()` e `malloc()` ed usare tale area come se fosse un array, ovvero usando gli accessi indicizzati tipici degli array.

Ad esempio:

- Allocazione dinamica di spazio per una stringa lunga al più L caratteri:

```
char *s;

s=calloc(L+1,1); /* oppure s=malloc(L+1); */
```

Array allocati dinamicamente

Allocazione dinamica di una matrice M di interi di dimensione $n \times m$

- M è un puntatore ad un array che contiene m valori `int` in ciascuna riga

```
int (*M)[m];

M=(int(*)[m])malloc(n*m*sizeof(int));
/* oppure M=(int(*)[m])calloc(n,m*sizeof(int)); */
```

- M è un puntatore ad un puntatore a `int`

```
int **M, i;

M=calloc(n,sizeof(int *));
for(i=0; i<n;i++) M[i]=calloc(m,sizeof(int));
```

L'indirettezza multipla

Quando si crea un puntatore che punta ad un altro puntatore che a sua volta punta al valore di destinazione si parla di indirettezza multipla o di puntatori a puntatori

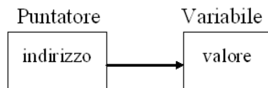


Figura: Indirettezza semplice

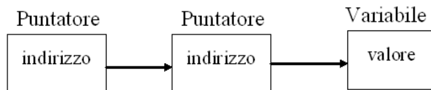


Figura: Indirettezza multipla

Le strutture

Una struttura è un insieme di variabili di tipi differenti raggruppate sotto uno stesso nome. Le componenti di una struttura sono dette campi o membri.

- Forma generale della dichiarazione:

```
struct nome_struttura {
tipo nome_membro;
tipo nome_membro;
...
tipo nome_membro;
};
```

- Forma generale dell'accesso ad un membro di una struttura:

```
nome_struttura.nome_membro
```

- Le informazioni contenute in una struttuta possono essere assegnate ad un'altra struttura dello stesso tipo usando un'unica istruzione di assegnamento

Le strutture

- I nomi dei campi di una stessa struttura devono essere unici, ma campi di strutture differenti possono avere lo stesso nome, in quanto per l'accesso ad essi è sempre necessario il nome della struttura di riferimento.
- Le strutture possono essere complesse, in quanto possono contenere campi che sono a loro volta array o strutture. Inoltre sono possibili array di strutture.
- L'identificatore della struttura è opzionale. In tale caso non si può fare riferimento alla struttura fuori dal contesto della dichiarazione delle variabili. Es:

```
struct {
tipo nome_membro;
...
tipo nome_membro;
} variabili_struttura;
```

Esempio

```
struct data {
    unsigned giorno;
    unsigned mese;
    unsigned anno;
};

struct studente {
    char nome[20];
    char cognome[20];
    struct data nascita;
};
```

Le strutture

Come nel caso degli array, è possibile inizializzare una variabile struttura in fase di dichiarazione. Se i valori sono in numero inferiore ai campi, generalmente viene assegnato 0 a quelli rimanenti

```
struct studente s1,s2 = {"Carla","Lo Re",{12,3,1987}};

struct studente elenco[2] = {{"Carla","Lo Re",{12,3,1987}},
                             {"Ugo", "Ricci", {14,5,1977}}};
```

- Il passaggio per valore di una struttura può risultare inefficiente se la struttura ha molti campi o campi che sono grandi array
- È consigliabile scrivere funzioni che ricevono come parametro l'indirizzo di strutture

Esempio

```
void leggi(char s[]){
    int i=0, c;
    while ((c=getchar())!='\n') s[i++]=c;
    s[i]='\0';
}
int main(){
struct studente st;
printf("Nome: "); leggi(st.nome);
printf("Cognome: "); leggi(st.cognome);
printf("Data di nascita gg/mm/aaaa: ");
scanf("%d/%d/%d",&st.nascita.giorno,
        &st.nascita.mese,&st.nascita.anno);
printf("\n%s %s, nata il %d/%d/%d\n",st.nome,st.cognome,
        st.nascita.giorno,st.nascita.mese,st.nascita.anno);
return 0; }
```

typedef

- `typedef` è un meccanismo che permette di associare un identificatore ad un tipo preesistente, ossia di dichiarare nuovi nomi di tipo
- Forma generale: `typedef tipo nuovo_nome;`

Vantaggi:

- rende più trasportabili i programmi dipendenti dalla macchina:

  definendo un proprio nome per ogni tipo di dati dipendente dalla macchina, basta cambiare le istruzioni `typedef` per rendere il programma compatibile con un nuovo ambiente
- può essere utile per l'autodocumentazione del codice, consentendo di usare nomi descrittivi per i tipi di dato

typedef: un esempio

```
typedef unsigned bool;
typedef struct {
    unsigned giorno, mese, anno;
} tipo_data;
typedef struct {
    char nome[20], cognome[20];
    tipo_data nascita;
} tipo_studente;

bool b; /* dichiarazioni di var */
tipo_studente stud;
```

typedef

Attenzione alle strutture !

```
typedef struct studente{
    char nome[20], cognome[20];
    tipo_data nascita;
} tipo_studente;
```

- tipo_studente è il nuovo nome di tipo associato al tipo struct studente

```
tipo_studente x;          /* dichiarazione corretta */
struct studente x;        /* dichiarazione corretta */
studente x;               /* dichiarazione errata */
struct tipo_studente x; /* dichiarazione errata */
```

operatore ->

- L'operatore -> permette l'accesso ai campi di una struttura per mezzo di un costrutto della forma

```
puntatore_a_struttura -> nome_campo
```

- Nel seguente esempio l'accesso ai campi avviene nei seguenti modi equivalenti:

```
struct studente *pst;

(*pst).nome          pst-> nome
```

strutture autoreferenzianti

- Le strutture autoreferenzianti (o strutture dati dinamiche) sono strutture contenenti puntatori alle strutture stesse
- Lo scopo di una definizione autoreferenziante è quello di disporre di un numero imprecisato di strutture innestate tra loro.
- Schema generale:

```
struct nome{
    <tipo> nome_campo;
    <tipo> nome_campo;
    ...
    struct nome *nome_campo;
    ...
    struct nome *nome_campo;
};
```

- Tali strutture richiedono routine per la gestione della memoria al fine di ottenere e liberare esplicitamente memoria.

Union

- Si definisce unione una variabile che può contenere (in momenti diversi) oggetti di tipo e di dimensione diversi.
- Le unioni costituisono un modo per intervenire su dati di diverso genere all'interno di un'unica area di memoria senza includere nel programma alcuna informazione vincolata alla macchina
- In pratica un'unione offre la possibilità di interpretare la stessa sequenza di bit in due o più modi diversi.

Union

- La sintassi si fonda sulle strutture:

```
union nome-unione {
    tipo nome-variabile;
    tipo nome-variabile;
    ...
    tipo nome-variabile;
} variabili-unione;
```

- Una variabile unione sarà abbastanza grande da contenere il più grande dei tipi; la dimensione precisa dipende dall'implementazione
- da un punto di vista sintattico l'accesso ai membri di un'unione è dato da `nome-unione.membro` oppure `puntatore-unione->membro`

rand() e srand()

- `int rand(void);`
  - La funzione `rand()` genera e restituisce un intero
  - chiamate ripetute generano una sequenza di numeri uniformemente distribuiti nell'intervallo [0,RAND_MAX], dove RAND_MAX è un valore definito in `stdlib.h` che vale almeno 32767
- void srand(unsigned seed);
  - Fornisce al generatore di numeri pseudocasuali un valore iniziale (seme). All'avvio del programma, il generatore agisce come se fosse stata chiamata `srand(1)`. L'istruzione `srand(time(NULL));` inizializza il generatore con un valore differente ad ogni esecuzione del programma

Esempio

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main () {
srand(time(NULL));  /* inizializza il seme */
/* genera alcuni numeri random */
printf("Numero tra 0 e RAND_MAX (%d): %d\n",RAND_MAX,rand());
printf("Numero tra 0 e 99: %d\n", rand()%100);
printf("Numero tra 20 e 29: %d\n", rand()%10+20);
return 0;
}
```

Puntatori a funzione

- Una funzione occupa una posizione fisica in memoria che può essere assegnata ad una variabile puntatore.
- Questo indirizzo è il punto di accesso alla funzione
- È possibile richiamare una funzione tramite il puntatore alla funzione
- I puntatori a funzione consentono anche di passare le funzioni come argomenti ad altre funzioni
- Per ottenere l'indirizzo di una funzione si usa il nome della funzione senza specificare le parentesi o gli argomenti

Puntatori a funzione

Confrontiamo le seguenti dichiarazioni:

- `double f (double, double);` (funzione)
- `double (*ptrf) (double, double);` (puntatore a funzione)

`ptrf` definisce un puntatore che punta a funzioni le quali prendono come argomenti due double, e restituiscono un double.

Puntatori a funzione

Il C tratta i nomi di funzioni come se fossero puntatori alle funzioni stesse.
Per assegnare ad un puntatore a funzione l'indirizzo di una certa funzione si deve effettuare un'operazione di assegnamento del nome della funzione al nome del puntatore a funzione:

```
double (*ptrf) (double, double);
double somma(double, double);
...
ptrf = somma;
```

Puntatori a funzione

Analogamente, l'esecuzione di una funzione mediante un puntatore che la punta, viene effettuata con una chiamata in cui si usa il nome del puntatore come se fosse il nome della funzione, seguito dai parametri necessari:

```
double (*ptrf) (double, double);
double somma(double, double);
double a = 5, b = 6, c;
...
ptrf = somma;
c = (*ptfr)(a, b); // oppure c = ptrf(a, b);
```

Definizione

- Astrazione di dato che cattura l'idea di sequenza di lunghezza indefinita (e dinamica) di elementi
- Una proprietà fondamentale delle liste è che la struttura logica non è collegata a quella fisica (come è invece per gli array). Questo rende essenziale in una lista la nozione di riferimento esplicito da un elemento all'elemento successivo.
- *Definizione ricorsiva.* Una lista è:
  - una lista vuota, oppure
  - un elemento (atomico o lista) seguito da una lista

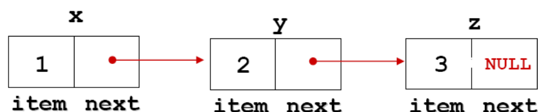
Liste perché?

Supponiamo di volere mantenere una sequenza ordinata di numeri, e di volere inserire in questa sequenza un nuovo numero.

- Rappresentare la sequenza tramite un array ordinato comporta:
  1. cercare la posizione corretta dove effettuare l'inserimento;
  2. spostare ordinatamente di una posizione tutti gli elementi maggiori di quello da inserire (eventualmente estendendo l'array);
  3. copiare nel buco così creato il nuovo elemento.
- Rappresentare la sequenza tramite una lista consente di:
  - evitare il passo 2 della procedura (spostamento degli elementi), dato che la contiguità logica non è associata alla contiguità fisica;
  - evitare di dovere necessariamente allocare a priori spazio di memoria per il numero massimo di elementi ipotizzabile.

Esempio

```
struct listNode{
    <tipo> item;
    struct listNode *next;
} x,y,z, *ptr;
x.item=1;  y.item=2;  z.item=3;
x.next=&y;  y.next=&z; ptr=&x;
```




```
x.next->item [equivalentemente *(x.next).item] vale 2
x.next->next->item vale 3
ptr++; /* ptr punta alla locazione ptr + sizeof(x) */
ptr=ptr->next /* ptr punta ad y*/
```

Tipo di dato lista v.1

```
typedef <tipo> itemType; /* def. dal programmatore */

struct listNode{
    itemType item;
    struct listNode*next;
};
typedef struct listNode listNode;
typedef listNode *List;
```

Tipo di dato lista v.2

```
typedef <tipo> itemType; /* def. dal programmatore */

typedef struct listNode{
    itemType item;
    struct listNode *next;
} listNode;
typedef listNode *List;
```

Tipo di dato lista v.3

```
typedef <tipo> itemType; /* def. dal programmatore */

typedef struct listNode *List;
typedef struct listNode{
    itemType item;
    List next;
} listNode;
```

Tipo di dato lista v.4

```
typedef <tipo> itemType; /* def. dal programmatore */

typedef struct listNode listNode;
typedef listNode *List;
struct listNode{
    itemType item;
    List next;
};
```

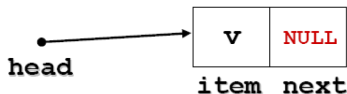
Allocazione di memoria

```
List head;
...
head = (List)malloc(sizeof(listNode));
/* oppure head = (List)malloc(sizeof(*head)); */
```

- Se head è una variabile di tipo `List`, con le chiamate di funzione si ottiene dal sistema un'area di memoria adeguata a contenere un `listNode` e si assegna l'indirizzo del primo byte dell'area a `head`

Creazione di un nuovo elemento

```
List newNode(itemType v){
    List head;
    head=(List)malloc(sizeof(*head));
    head->item=v;
    head->next=NULL;
    return head;
}
```



Inserimento di un valore in testa ad una lista

Versione 1:

```
List insertHead(itemType v, List head){
    List ptr=newNode(v);
    ptr->next=head;
    return ptr;
}
```

Esempio di chiamata alla funzione:

```
List p; itemType x;
...
p=insertHead(x,p);
```

Inserimento di un valore in testa ad una lista

Versione 2:

```
void insertHead2(itemType v, List *phead){
    List ptr=newNode(v);
    ptr->next=*phead;
    *phead=ptr;
}
```

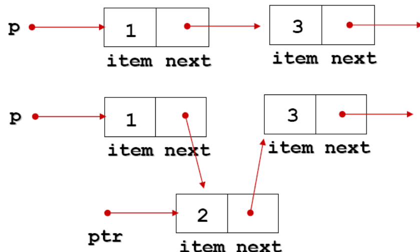
Esempio di chiamata alla funzione:

```
List p; itemType x;
...
insertHead2(x,&p);
```

Inserimento di un valore tra elementi adiacenti

Versione 1:

```
void insertNext(itemType v, List p){
    List ptr=(List)malloc(sizeof(listNode));
    ptr->item=v;
    ptr->next=p->next;
    p->next=ptr;
}
```



Inserimento di un valore tra elementi adiacenti

Versione 2:

```
void insertNext(itemType v, List p){
    List ptr=newNode(v);
    ptr->next=p->next;
    p->next=ptr;
}
```

Versione 3:

```
void insertNext(itemType v, List p){
    p->next=insertHead(v,p->next);
    /* insertHead(v,&(p->next)); */
}
```

Cancellazione dell'elemento in cima alla lista

Versione 1:

```
List deleteHead(List head){
    List ptr=head;
    head=head->next;
    free(ptr);
    return head;
}
```

Versione 2:

```
void deleteHead2(List *phead){
    List ptr=*phead;
    *phead=(*phead)->next;
    free(ptr);
}
```

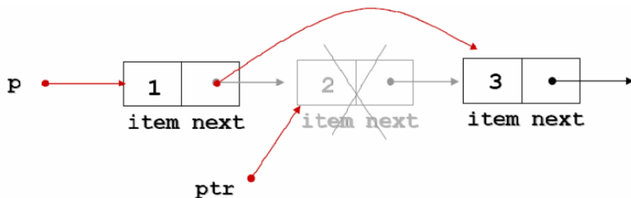
Cancellazione di un elemento intermedio

Versione 1:

```
void deleteNext(List p){
    List ptr=p->next;
    p->next=ptr->next;
    free(ptr);
}
```

Versione 2:

```
void deleteNext(List p){
    p->next=deleteHead(p->next);
}
```



Ricorsione

Una funzione è detta ricorsiva se richiama se stessa, direttamente o indirettamente (il corpo della funzione prevede chiamate a funzioni che a loro volta chiamano la funzione originaria).
Le funzioni ricorsive semplici hanno una struttura standard:

- uno o più casi base che vengono verificati all'inizio della ricorsione;
- caso generale in cui una delle variabili è passata come parametro in modo da raggiungere ad un certo punto il caso base.

Esempio 1: calcolo della somma dei primi *n* interi positivi:

```
int somma(int n){
    if (n<=1) return n;
    else return n + somma(n-1);
}
```

Note sull'esecuzione

Ad ogni invocazione il sistema crea un record di attivazione in cui memorizza i valori correnti delle *variabili locali*, dei *parametri* e la *locazione del valore di ritorno*.

La creazione del frame per ogni attivazione, permette di:

- tenere traccia dell'esecuzione;
- salvare lo stato corrente per ripristinarlo al ritorno dell'esecuzione;
- lavorare sulle variabili locali ad un'attivazione senza interferenze.

Requisiti

Le soluzioni ricorsive si applicano meglio a problemi con possibile definizione induttiva:

Caso base $\rightarrow$ Saper riconoscere quando il problema è abbastanza semplice da essere risolto direttamente (condizione di terminazione)

Passo induttivo $\rightarrow$ Usare la soluzione di un sottoproblema del problema per arrivare alla soluzione del problema (chiamata ricorsiva)

Iterazione vs Ricorsione

Ripetizione

- Iterazione : ciclo esplicito
- Ricorsione: chiamate ripetute di funzione

Terminazione

- Iterazione : fallisce la condizione del ciclo
- Ricorsione : passo base riconosciuto

Bilancio

- Scelta tra efficienza (iterazione) e semplicità ed eleganza (ricorsione)

Qual'è la questione?

Un tipo di dato è definito da un insieme di valori e da una collezione di operazioni su questi valori (ad es. `int` è definito dall'insieme dei numeri naturali e dalle operazioni $+, -, *, /, \%$)

Qual'è la questione? Come definire i nostri tipi di dato al fine di organizzare i programmi in modo efficace?

- Dato: informazione derivante da qualche descrizione formale (matematica) o informale di una realtà di interesse
- Tipo di dato astratto (ADT): specifica delle operazioni di interesse su un insieme di dati ben definiti. L'accesso ai dati è vincolato al solo uso delle operazioni definite nel ADT
- Struttura dati: organizzazione dei dati nei tipi di dato del linguaggio di programmazione in modo da supportare le operazioni di un ADT usando meno risorse di calcolo (tempo e/o spazio) possibile

Un formalismo per ADT

Un tipo di dato astratto (ADT) consiste di:

1. un insieme di domini (insieme di dati ben definiti)
2. un insieme di operazioni di base definite su questi domini
3. un insieme di regole (assiomi) che definiscono le operazioni, ossia che ne descrivono gli effetti

Definizione astratta di "Sequenza di elementi"

- Una *sequenza di elementi* è un insieme vuoto di elementi oppure consiste di un elemento e di una sequenza
- Le proprietà della sequenza sono indipendenti dalle caratteristiche degli elementi

Per gestire una sequenza di elementi sembra naturale definire operazioni per:

- creare una sequenza vuota (priva di elementi)
- inserire, estrarre e cancellare un elemento
- riconoscere se una sequenza è vuota

ADT Sequenza[Item]

1. Insieme di domini: { Sequenza, Item, Boolean }
2. Operazioni:
   - `init()`→`Sequenza`
     crea una sequenza vuota
   - `insert(Sequenza, Item)`→`Sequenza`
     inserisce un elemento come primo elemento della sequenza
   - `delete(Sequenza)`→`Sequenza`
     cancella il primo elemento della sequenza
   - `getItem(Sequenza)`→`Item`
     recupera il primo elemento della sequenza
   - `isEmpty(Sequenza)`→`Boolean`
     verifica se una sequenza è vuota o meno
3. Assiomi: $\forall i \in \mathit{Item}, \forall S \in \mathit{Sequenza}$:

```
isEmpty(init())= TRUE
isEmpty(insert (S, i))= FALSE
getItem(insert (S, i))= i
delete(insert (S, i))= S
```

ADT Sequenza[Item]

- Combinazioni delle precedenti operazioni di base sono sufficienti a definire un ampio insieme di funzioni per l'elaborazione delle sequenze da parte di un client
- Notare che la definizione del tipo `Item` può essere lasciata aperta e che la definizione del ADT *Sequenza* è completa senza esplicita definizione del tipo di dato astratto `Item`

Fasi dello sviluppo di un programma

L'idea di base è che lo sviluppo di un programma avviene in due fasi:

1. Definizione degli algoritmi e progetto degli ADT
2. Implementazione:
   - scelta delle strutture dati
   - traduzione delle operazioni astratte in funzioni e procedure

Approcci fondamentali per lo sviluppo di un programma: top-down (si parte dalle routine di livello più alto e si procede via via verso le routine che eseguono compiti più specifici), bottom-up, ad-hoc

Scelta delle strutture dati

La scelta delle strutture dati è strettamente correlata alle operazioni che devono essere eseguite su di esse

La scelta delle strutture dati e quella degli algoritmi sono strettamente correlate:

- per uno stesso insieme di dati vi sono strutture che richiedono più spazio di altre
- sebbene l'efficienza intrinseca degli algoritmi sia sempre determinata a livello della fase di progettazione astratta, per uno stesso insieme di operazioni sui dati, alcune strutture dati portano ad implementazioni più efficienti di altre

Interfaccia e implementazione

Chiamiamo:

- interfaccia la definizione delle strutture dati e la dichiarazione delle funzioni per la loro manipolazione
  - pensiamo all'interfaccia come a una definizione del tipo di dato
- implementazione un programma che specifica il tipo di dato, ossia che definisce le funzioni dichiarate nell'interfaccia
- client un programma che usa le funzioni dichiarate nell'interfaccia

Organizzazione di ADT in C

La struttura di un ADT comprende quindi almeno:

1. un file header contenente:
   - la definizione del tipo di dato (typedef)
   - la dichiarazione delle funzioni
2. un file di implementazione contenente:
   - una direttiva `#include` per importare le `typedef` e la dichiarazione delle funzioni
   - la definizione delle funzioni

Pro e contro

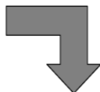
- ⇑ GLi ADT permettono di scrivere programmi per mezzo di astrazioni di livello più elevato
- ⇑ Le trasformazioni concettuali operate dai programmi sui dati sono distinte dalle specifiche implementazioni algoritmiche
- ⇑ Il codice associato al tipo di dato è incapsulato e quindi utilizzabile da altri programmi client
- ⇑ È possibile modificare la rappresentazione dei dati senza modificare il client
- ⇓ Dato che la rappresentazione nella pratica è comunque disponibile ai programmi client, si è esposti ad errori sottili se i client dipendono in qualche modo da tale rappresentazione
- ⇓ Quando le prestazioni sono veramente importanti si ha bisogno di conoscere i costi delle operazioni primitive

Esempio: ADT "punto nel piano"

point.h

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

float distance (point, point); …
```



user.c

```
#include "point.h"
…
```

point.c

```
#include "point.h"
#include <math.h>

float distance (point a, point b){
float dx=a.x-b.x, dy=a.y-b.y;
return sqrt(dx*dx+dy*dy);
}
```

Definizione

Uno stack è un insieme vuoto di elementi oppure consiste di un elemento (la cima dello stack) e di uno stack

- Uno stack è una collezione di elementi su cui sono imposte le seguenti limitazioni sull'accesso agli elementi:
  - disciplina di accesso LIFO (Last-In-First-Out): l'ultimo elemento inserito è il primo ad essere recuperato o cancellato
  - l'accesso diretto ad elementi diversi dall'ultimo inserito NON è consentito

Ad es., gli stack sono usati nei linguaggi di programmazione che supportano la ricorsione e nella valutazione di espressioni aritmetiche in forma post-fissa

ADT Stack[Item]

1. Insieme di domini: { Stack, Item, Boolean }
2. Operazioni:
   - `newStack()`$\rightarrow$`Stack`
     crea un nuovo stack vuoto
   - push(Item,Stack)$\rightarrow$Stack
     inserisce un elemento nello stack
   - pop(Stack)$\rightarrow$Stack
     cancella l'ultimo elemento inserito
   - top(Stack)$\rightarrow$Item
     recupera l'ultimo elemento inserito
   - isEmptyStack(Stack)$\rightarrow$Boolean
     verifica se uno stack è vuoto o meno
3. Assiomi: $\forall i \in Item, \forall S \in Stack$:

```
isEmptyStack(newStack())=TRUE
isEmptyStack(push(i,S))=FALSE
top(push(i,S))=i
pop(push(i,S))=S
```

Espressioni aritmetiche in notazione postfissa

- Nella maggior parte dei linguaggi di programmazione le espressioni matematiche sono scritte nella consueta notazione infissa, ovvero con l'operatore tra i due operandi (es: $31 + 123$).
- Una notazione alternativa è quella postfissa nella quale l'operatore segue gli operandi (es: 31 123 $+$).
- Vantaggi:
  - la notazione postfissa può rivelarsi utile perchè c'è un modo del tutto naturale per valutare espressioni postfisse con l'uso dello stack
  - non sono necessarie parentesi per controllare l'ordine delle operazioni. Per ottenere lo stesso risultato di $12 + 3 *$ con la notazione infissa dovremmo scrivere (1 + 2) * 3.

Metodo di valutazione espressione aritmetica postfissa

- A partire dall'inizio dell'espressione ricava un termine (operatore o operando) alla volta:
  - se il termine è un operando inseriscilo nello stack
  - se il termine è un operatore estrai dallo stack il numero di operandi previsto per l'operatore, elabora il risultato dell'operazione su di essi e inserisci il risultato nello stack
- Quando tutta l'espressione è stata elaborata nello stack, il risultato è rappresentato (ipotizzando che l'espressione postfissa sia corretta) dall'unico elemento presente nello stack

Conversione da espressione infissa in postfissa

- Per trasformare un'espressione infissa in una postfissa notiamo innanzitutto che gli operandi devono comparire nella notazione postfissa nello stesso ordine che in quella infissa. Quindi se scandiamo l'espressione infissa da sinistra verso destra ogni operando viene semplicemente accodato nell'espressione postfissa
- Per gli operatori invece bisognerebbe tener conto delle regole di precedenza: infatti $a + b * c$ corrisponde a $a\ b\ c\ *\ +$, in quanto il prodotto è calcolato prima della somma, ma $(a + b) * c$ deve essere tradotto in $a\ b\ +\ c\ *$, in modo che si calcoli prima la somma e poi il prodotto.

Tratteremo una versione semplificata che assume che l'espressione infissa sia scritta in forma completa attraverso l'uso di tutte le parentesi. Inoltre considereremo solo gli operatori $+$ e $*$.

Algoritmo di trasformazione da infissa in postfissa

- Scandisci l'espressione infissa da sinistra verso destra e:
  1. se il simbolo letto è (, ignoralo
  2. se il simbolo letto è un operando oppure ' ' (carattere spazio), accodalo alla postfissa
  3. se il simbolo letto è un operatore ($+$ o $*$), inseriscilo nello stack
  4. se il simbolo letto è), estrai dallo stack l'operatore e accodalo all'espressione postfissa
- Esempio:
  Infissa $= ((20 + (15 * (100 + 5))) + 21)$
  Postfissa $=$ 20 15 100 5 $+$ $*$ $+$ 21 $+$

Postscript

- PostScript (PS) è un linguaggio di programmazione sviluppato da Adobe Systems inizialmente per il controllo delle stampanti
- È un linguaggio di descrizione di pagina interpretato, orientato alla descrizione di pagine ed immagini
- Consente di descrivere pagine di testo e grafica in modo indipendente dalla risoluzione e dal dispositivo di visualizzazione
- I programmi sono scritti in forma postfissa e sono interpretati con l'aiuto di uno stack interno
- PS ha un numero di funzioni primitive che costituiscono le istruzioni per un dispositivo grafico astratto. L'utente può anche definire funzioni proprie.

Postscript

Un file PS è in realtà un file di testo puro consultabile con un qualsiasi editor di testo. Dunque è possibile generare un file PS scrivendo direttamente del codice.

```
%!PS
/Courier findfont
20 scalefont
setfont
72 500 moveto
(Hello world!) show
showpage
```

- Salvando questo codice come `Nomefile.ps`, esso può essere direttamente inviato ad una stampante PostScript o può essere letto da alcuni visualizzatori di documenti (ad esempio con Evince), oppure questo codice può essere messo direttamente in pasto ad un interprete PostScript come Ghostscript.

Definizione

La coda è un ADT fondamentale simile allo stack, ossia è una collezione "specializzata" di elementi.

- La differenza rispetto ad uno stack consiste nella diversa disciplina di accesso agli elementi:
  - disciplina di accesso FIFO (First-In-First-Out): l'elemento che è stato inserito meno recentemente è il primo ad essere recuperato o cancellato
  - l'accesso diretto ad elementi diversi dal primo inserito NON è consentito

Applicazioni

- Le code risultano estremamente utili in applicazioni relative alla programmazione di sistema, quali la schedulazione di risorse nei sistemi operativi e la scrittura di simulatori di eventi
- Es. di applicazioni:
  - un buffer tra un processo produttore ed uno consumatore è realizzato come una coda;
  - un sistema operativo multi-processing mantiene una coda di processi per essere eseguito

ADT Queue[Item]

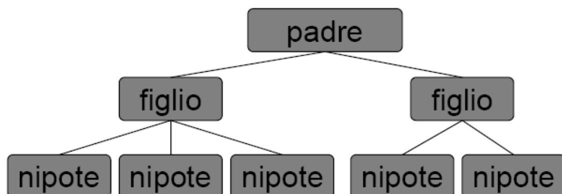
1. Insieme di domini: { Queue, Item, Boolean }
2. Operazioni:
   - newQueue()→Queue
     crea una nuova coda vuota
   - put(Item,Queue)→Queue
     inserisce un elemento nella coda
   - get(Queue)→
     cancella l'elemento inserito meno recentemente
   - getItemQueue(Queue)→Item
     recupera l'elemento inserito meno recentemente
   - isEmptyQueue(Queue)→Boolean
     verifica se la coda è vuota o meno
3. Assiomi: $\forall i \in Item, \forall Q \in Queue$:

```
isEmptyQueue(newQueue())=TRUE
isEmptyQueue(put(i,Q))=FALSE
get(put(i,Q))= if isEmptyQueue(Q) then Q else put(i,get(Q))
getItemQueue(put(i,Q))= if isEmptyQueue(Q) then i
                        else getItemQueue(Q)
```

Definizione

Un albero è una collezione di elementi (nodi), sui quali è definita una relazione di discendenza con due proprietà:

1. esiste un solo nodo radice senza predecessori
2. ogni altro nodo ha un unico predecessore



Definizione

- Un albero è una coppia $T =< V, E >$ dove V e un insieme di nodi (o vertici) , $E \subseteq V \times V$ è una relazione su V
- un nodo (o vertice) $v \in V$ è un oggetto che può essere dotato di un nome e di una informazione associata
- un arco e è una connessione tra nodi: $e =< v_{padre}, v_{figlio} >$, $e \in E$

Proprietà: esiste esattamente un cammino tra ogni coppia di nodi

- si dice cammino una sequenza di nodi distinti in cui i nodi successivi sono connessi da un arco

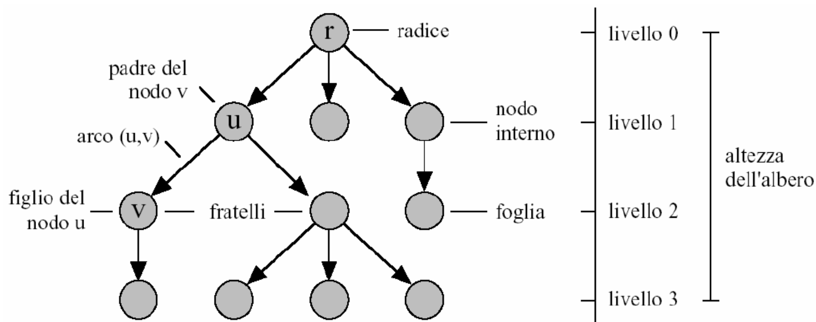
Motivazioni

Gli alberi giocano un ruolo centrale nella progettazione e nell'analisi degli algoritmi e nelle applicazioni informatiche:

- impieghiamo gli alberi per descrivere le proprietà dinamiche degli algoritmi
- gli alberi sono strettamente legati all'analisi sintattica dei linguaggi di programmazione
- Un file system ha una organizzazione ad albero .. usiamo frequentemente strutture che rappresentano implementazioni concrete di alberi

Terminologia

- Livello/Profondità di un nodo: distanza di un nodo dalla radice; la radice ha livello 0; ogni nodo interno (ovvero $\neq$ radice) ha livello pari a quello del padre + 1
- Altezza di un albero: massimo fra i livelli di tutti i suoi nodi



Terminologia

- Antenati di un nodo: nodi raggiungibili salendo di padre in padre lungo il cammino verso la radice
- Discendenti di un nodo: nodi raggiungibili scendendo di figlio in figlio nell'albero
- Albero ordinato: albero in cui è specificato un ordine totale tra i figli di ciascun nodo
- Grado (di uscita) di un nodo: numero dei figli del nodo
- Albero d-ario: albero in cui tutti i nodi hanno grado $\leq$d
- Albero d-ario completo di altezza h: albero in cui tutte le foglie hanno profondità h ed ogni altro vertice ha grado =d

Definizioni

- Un albero ordinato d-ario è vuoto oppure consiste di una radice connessa ad una sequenza ordinata di alberi ordinati d-ari
  - ogni nodo è la radice di un "sotto-albero" formato dal nodo stesso e da tutti i suoi discendenti
- Un albero binario è un albero ordinato 2-ario: ogni nodo ha esattamente un padre e (al più) due figli ordinati detti figlio sinistro e figlio destro

Proprietà

- Il livello i-mo di un albero binario contiene al più 2^i nodi
- Un albero binario completo di altezza h contiene $2^{h+1} - 1$ nodi (progressione geometrica: $2^0 + 2^1 + 2^2 + \ldots + 2^h$)
- Un albero completo di N nodi ha un'altezza $h = \log_2 (N + 1) - 1$
- Un albero binario completo di N nodi ha un'altezza $h = \lceil \log_2 (N + 1) \rceil - 1$
- L'altezza di un albero binario di N nodi è compresa tra $h = \lceil \log_2 (N + 1) \rceil - 1$ e $N - 1$

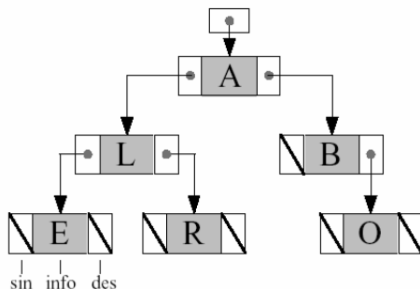
ADT Btree[Item]

1. Insieme di domini: { Btree, Item, Boolean }
2. Operazioni:
   - `newBtree()`→`Btree` (crea un nuovo albero binario vuoto)
   - `left(Btree)`→`Btree` (individua il sotto-albero sinistro)
   - `right(Btree)`→`Btree` (individua il sotto-albero destro)
   - `data(Btree)`→`Item` (recupera l'elemento contenuto nella radice)
   - `make(Btree,Item,Btree)`→`Btree`
     (costruisce un nuovo albero a partire dai tre parametri)
   - isEmptyBtree(Btree)→Boolean
     (verifica se un albero binario è vuoto o meno)
3. Assiomi: $\forall i \in Item, \forall L, R \in Btree$:

```
isEmptyBtree(newBtree())=TRUE
isEmptyBtree(make(L,i,R))=FALSE
left(make(L,i,R))=L
right(make(L,i,R))=R
data(make(L,i,R))=i
```

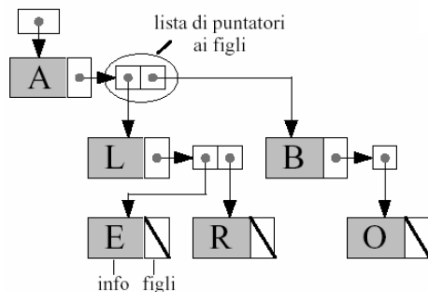
Rappresentazioni collegate

- Rappresentazione con puntatori ai figli sinistro e destro
- generalizzabile ad alberi ordinati d-ari



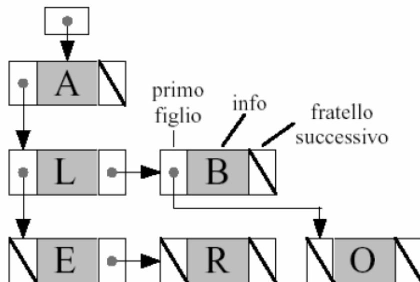
Rappresentazioni collegate

- Rappresentazione con liste di puntatori ai figli
- generalizzabile ad alberi ordinati d-ari e con numero illimitato di figli



Rappresentazioni collegate

- Rappresentazione di tipo primo figlio-fratello successivo
- generalizzabile ad alberi ordinati con numero arbitrario di figli



Rappresentazione mediante array

- Un albero binario di altezza h può essere rappresentato mediante un array di dimensione pari al numero di nodi di un albero completo, ossia $N = 2^{h+1} - 1$