Università degli Studi di Cagliari
Corso di Laurea in Ingegneria Biomedica

ELEMENTI DI INFORMATICA

http://people.unica.it/gianlucamarcialis/

A.A. 2020/2021

Docente: **Gian Luca Marcialis**

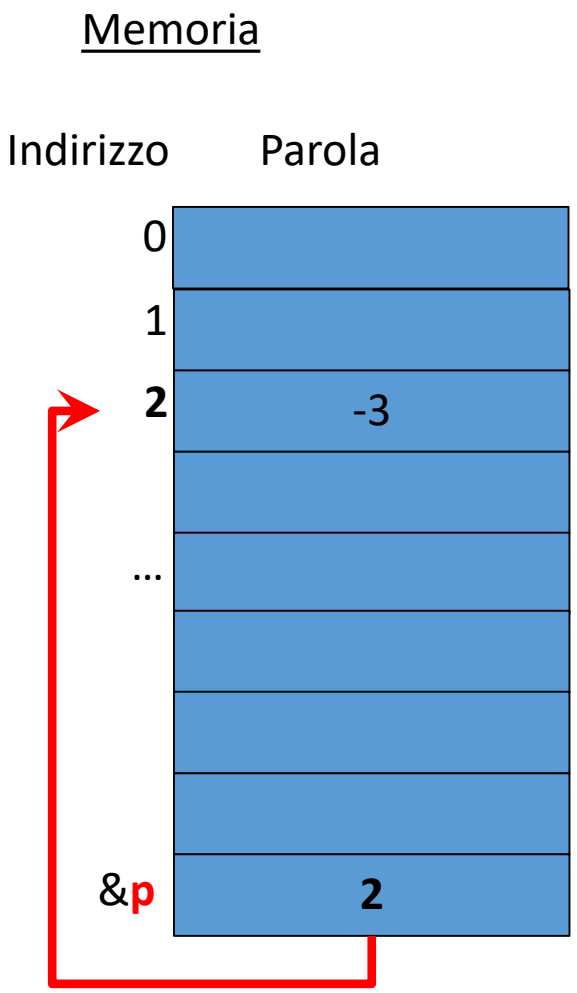
LINGUAGGIO C
Puntatori
Strutture dati dinamiche

Sommario

- Definizione
- Allocazione di variabili e vettori
- Le liste
  - Liste concatenate

Il tipo puntatore

- Il tipo puntatore è definito sull'intero supporto degli indirizzi di memoria allocabile
- Può dunque assumere uno tra i possibili valori e tale valore corrisponde ad un indirizzo di memoria
- L'indirizzo di memoria "puntato" può essere riferito ad un tipo semplice o strutturato
- In figura è rappresentata una variabile puntatore **p**
  - Essa ha il suo indirizzo di locazione in &p
  - Il suo valore p equivale però ad un indirizzo di memoria (2)
  - Attraverso esso si risale all'indirizzo 2 ovvero al valore puntato (l'intero -3)



Puntatori e tipi semplici

- Un puntatore che contiene l'indirizzo di una variabile intera si dichiara così:

```
int* p;
```

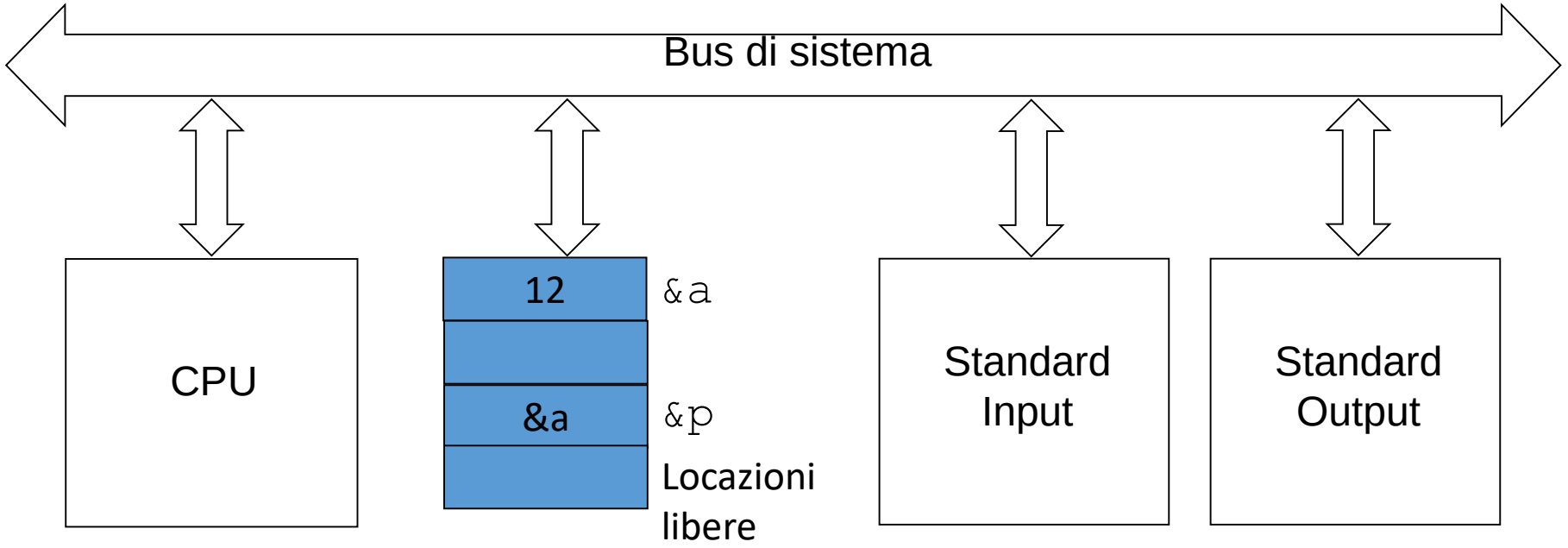
- Per esempio, se nel codice è stata dichiarata una variabile intera `a`, l'indirizzo di `a` può essere assegnato a `p`:

```
p = &a;
```

- L'accesso al contenuto effettivo di `a` può avvenire ora tramite la scrittura `*p`

Visione nella macchina virtuale C

- Lo stato della “macchina virtuale C” dopo l’assegnamento di `&a` a `p`



Allocazione di un puntatore

- Quando si dichiara una variabile di tipo puntatore (associata ad un tipo semplice o strutturato), come per una variabile normale, il suo contenuto è qualunque, e può essere anche quello di un'area di memoria "non consentito"
- E' dunque necessaria un'operazione addizionale di "allocazione", che corrisponde ad assegnare alla variabile puntatore un valore consentito
- In ogni caso, prima di utilizzare un puntatore, è sempre meglio far seguire la dichiarazione con un assegnamento ad un valore costante `NULL`, che indica "locazione non assegnata":

```
int *p;
p=NULL;
```

Allocazione : la funzione malloc

- Per allocare un puntatore si richiama la funzione di libreria `malloc()` presente nelle libreria `stdlib.h` e `alloc.h`
- Nel seguito tale operazione viene svolta subito dopo la dichiarazione:

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

- La funzione **malloc** restituisce l'indirizzo ad un'area di memoria di estensione pari al parametro in ingresso, in questo caso corrispondente all'estensione di una varaibile intera (**`sizeof(int)`**)
- Il tipo a cui è assegnato l'indirizzo dev'essere lo stesso della variabile p, per questo motivo il cast di uscita della funzione viene forzato a int*

Assegnamento di un valore intero puntato

- Per assegnare un valore alla locazione puntata da `p`:

```
*p = 12; /*l'asterisco indica il contenuto della
     locazione di indirizzo p*/
```

- Facendo precedere il nome delle variabili-puntatore dall'asterisco, possiamo eseguire qualunque altra operazione accedendo ai contenuti:

```
*p3 = *p1 + *p2;
/* somma i contenuti delle locazioni p1 e p2 e li memorizza
  nella locazione p3 */
```

- Naturalmente, nell'esempio di cui sopra, `p1`, `p2` e `p3` sono già state allocate separatamente con `malloc`

Esercizio

- Indicare quale sarà il valore finale stampato dal seguente programma C:

```
/*Programma*/
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int *p;
  int a;
  a=-1;
  p=&a;
  printf("\n%d\n",*p);
  return 0;
}
```

Ritorno alle funzioni: il passaggio per «variabile»

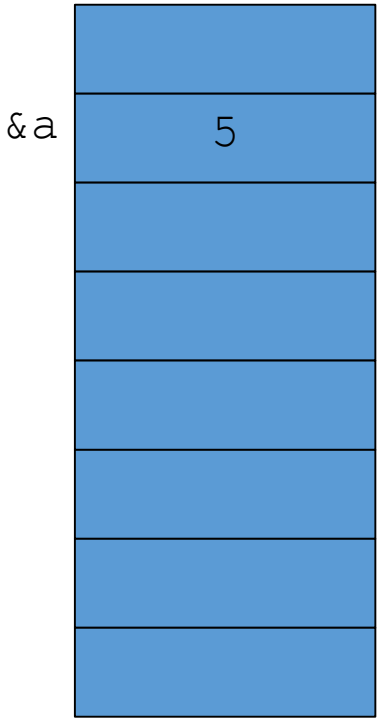
```
void alteraPerValore(int v)
{
      v=12;
}

void alteraPerVariabile(int* v)
{
      *v=12;
}
```

```
void main()
{
   int a;

   a=5;

   alteraPerValore(a);
   printf("%d",a);

   alteraPerVariabile(&a);
   printf("%d",a);

}
```

Come si spiega alla luce dei puntatori?

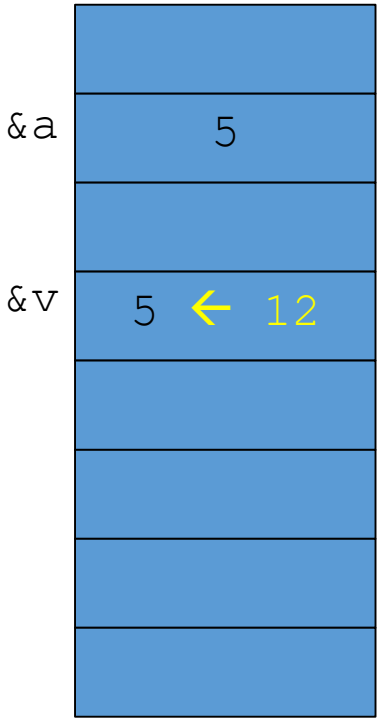
Stato iniziale

Memoria



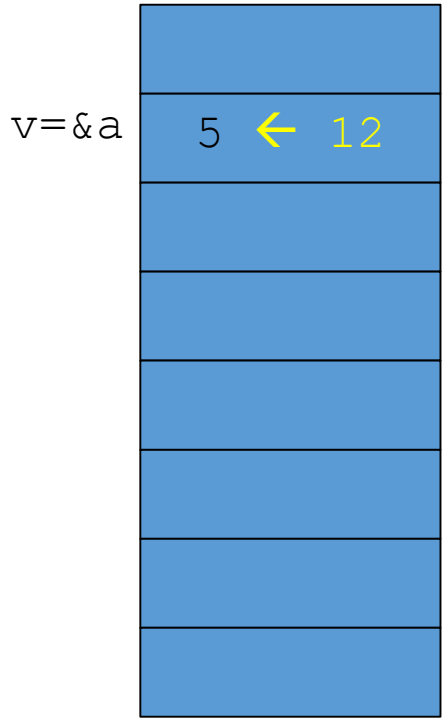
Chiamata a `alteraPerValore`

Memoria



Chiamata a `alteraPerVariabile`

Memoria



“Deallocazione” di un puntatore

- Se una variabile puntatore `p` **non ci serve più**, ed era stato allocato con `malloc`, **prima di uscire dal codice**, sia esso una funzione od il programma principale, l’area di memoria assegnata al puntatore dev’essere liberata con la funzione `free()`

  ```
  free(p); /*il parametro è la variabile*/
  ```
- In questo modo, quell’area potrà essere utilizzata da altre variabili, che, come noto, necessitano di occupare una zona di memoria

Ricapitolando

```
{
    /*parte dichiarativa di blocco generico*/

    /* dichiarazione di variabile puntatore a
    variabile di tipo TIPO*/
    TIPO *p;

    /*allocazione della variabile*/
    p=(TIPO*)malloc(sizeof(TIPO));

    /*deallocazione della variabile*/
    free(p);
}
```

Allocazione di un vettore tramite puntatori

- Per allocare un vettore di N elementi interi:

```
int* vettore;
vettore=(int*)malloc(sizeof(int)*N);
```

- Da questo momento, il vettore è trattabile come quelli che abbiamo imparato ad usare nel corso:
  - Per l'accesso alla `i`-esima componente si scrive: `vettore[i]`
  - In alternativa si scrive: `*(vettore+i)`
    - somma l'indirizzo iniziale (`vettore`), alla posizione della variabile da estrarre (`i`) e poi, con l'operatore `*`, la estrae
  - Scrivere `vettore[i]` o `*(vettore+i)` non fa alcuna differenza

- Se il vettore non ci serve più, possiamo "deallocarlo":

```
free(vettore);
```

Vettori statici e dinamici

- Dichiarazione di vettore statico di `N` interi: **`int`** `v[N];`
  - Una volta dichiarato, viene implicitamente allocato in memoria lo spazio per `N` interi consecutivi a partire dall'indirizzo base `v` o `&v[0]`
  - Deallocato implicitamente all'uscita del blocco di allocazione

- Dichiarazione di puntatore a intero: **`int`**`* v;`
  - Significa soltanto che c'è una variabile indirizzo di memoria che contiene un indirizzo a un intero
  - Il vettore non è allocato finché non assegnato esplicitamente con `malloc` con spazio per `N` interi a partire dall'indirizzo base `v` o `&v[0]`
  - Deallocato esplicitamente con `free`
  - Per le ragioni di cui sopra chiamato vettore *dinamico*

Ricapitolando

```
{
    /*dichiarazione di vettore statico*/
    TIPO vstatico[N];

    /* dichiarazione di variabile puntatore a variabile
    di tipo TIPO*/
    TIPO *vdinamico;

    /*allocazione del vettore dinamico*/
    vdinamico=(TIPO*)malloc(sizeof(TIPO)*N);

    /*deallocazione del vettore dinamico*/
    free(vdinamico);
}
```

Esercizio

- Scrivere una funzione `somme_cumulative` C che, ricevendo in ingresso una variabile intera N, **restituisca un vettore di N elementi** che contenga nella posizione i-esima le somme cumulative da 0 a i
  - In altri termini, chiamando `v` il vettore da restituire, deve risultare:

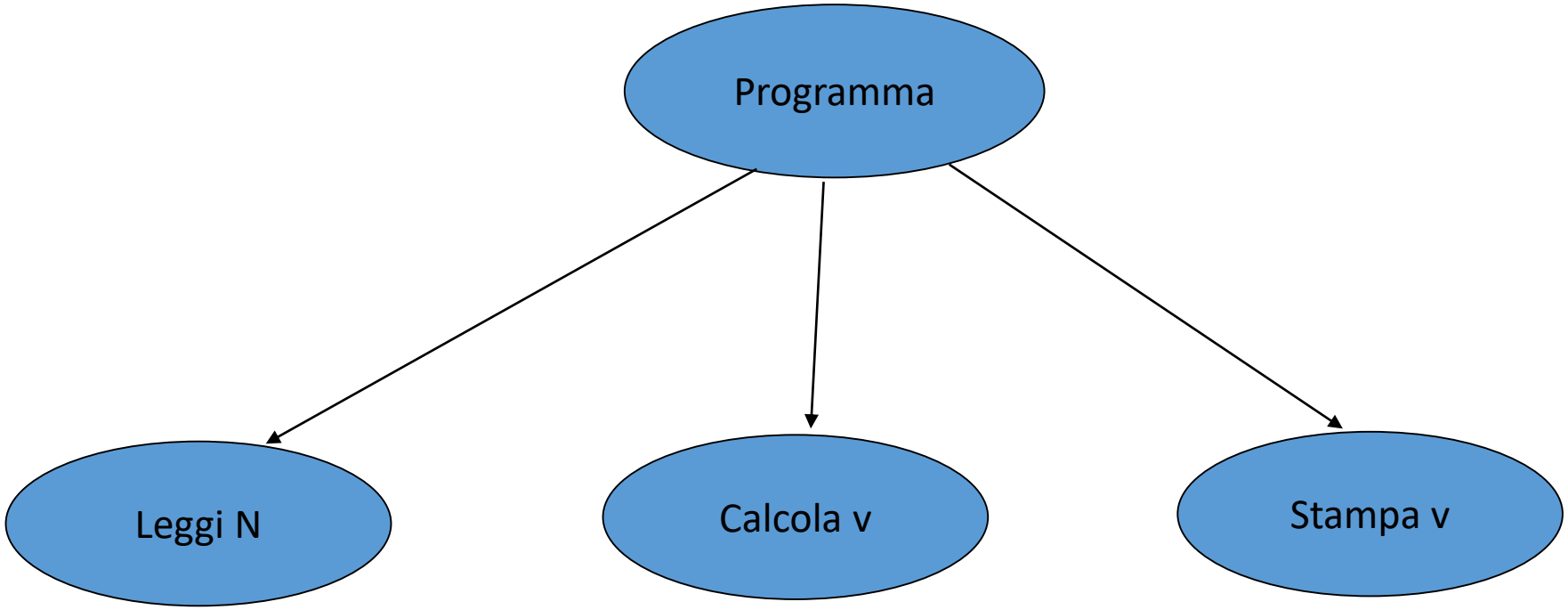
$$v[i] = \sum_{j=0}^{i} j$$

- Scrivere un programma C che, ricevendo da tastiera un intero N, stampi a video le somme cumulative contenute in un vettore definito come sopra.
- Per risolvere il problema si scrivano inoltre le seguenti funzioni:
  - Una funzione `leggi` che legga da tastiera un intero N e lo restituisca
  - Una funzione `stampa` che ricevendo in ingresso un vettore di interi lo stampi su file «output.txt»

Funzione somme

```
int* somme(int N)
{
 int *v, i;
 v=(int*)malloc(sizeof(int)*(N+1));
 v[0]=0;
 for(i=1; i<=N; i++)
     v[i]= i + v[i-1];
 return v;
}
```

Vista top-down



Funzioni leggi e stampa

```
int leggi(void)
{
 int N;
 scanf("%d",&N);
 return N;
}
```

```
void stampa(int *v, int N)
{
  int i;
  for(i=0; i<N; i++)
   printf("%d",v[i]);
  return;
}
```

Programma principale

```
/*Programma per la stamma dei valori sum_j(j) dato in ingresso un intero N*/
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int *v, N;

  N=leggi();
  v=somme(N);
  stampa(v,N);

  free(v); /*prima di uscire, bisogna deallocare il vettore!!!! */

  return 0;
}
```

Passaggio per variabile e puntatori

- Scrivere una funzione che, ricevendo due valori interi, assegnati alle variabili a e b, scambi i contenuti delle due variabili
- Sappiamo già risolvere l'esercizio passando a e b **per variabile**. Ora sappiamo cosa significa l'uso del carattere '*' per indicare tale passaggio:

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

Puntatori e dati strutturati

- Una variabile puntatore può essere associata ad un tipo strutturato. Ad esempio:

```
typedef struct
{
  int giorno, mese, anno;
} tipo_data;
```

- A questo punto un puntatore ad una variabile di tipo tipo_data si dichiara così:

```
tipo_data *data1, data2;
```

- La differenza **formale** nella dichiarazione di `data1` con `data2` è l'uso dell'asterisco.
- La differenza **sostanziale** è che alla variabile `data1` non è associata alcun'area di memoria, ma solo l'area che conterrà l'indirizzo iniziale di quella variabile strutturata. A `data2` è invece stata associata un'area di memoria grande quanto una variabile di tipo `tipo_data`.

Allocazione di variabili strutturate

- Esattamente come con tipi semplici:

```
data1=(tipo_data*)malloc(sizeof(tipo_data));
```

- Per assegnare un valore alle componenti di `data1`, ci sono due modi (nell'esempio utilizziamo la componente `ora`):

```
1) (*data1).ora = 23;
2) data1->ora = 23;
```

- Deallocazione:

```
free(data1);
```

Esercizio

- Definire un tipo strutturato composto da due interi, un float e una stringa di 20 caratteri e chiamarlo `strut`
- Nella main:
  - Dichiarare una variabile puntatore ad una variabile di tipo `strut`
  - Allocare la variabile dichiarata
  - Deallocare la variabile dichiarata

Soluzione

```
/*Definizione*/
typedef struct
{
        int v1, v2;
        float v3;
        char s[20];
} strut;
```

```
void main()
{
    /*dichiarazione*/
    strut *t;

    /*allocazione*/
    t=(strut*)malloc(sizeof(strut));

    /*deallocazione*/
    free(t);
}
```

Esercizio

- Scrivere un programma C che, leggendo da file ‘dati.txt’ una sequenza di interi, li memorizzi via via in un vettore v
- Se il valore letto è già stato inserito, il valore non deve essere aggiunto al vettore
  - La dimensione massima del vettore NON è nota e non deve essere fissata a priori.
- Il vettore ottenuto sia salvato nel file ‘output.txt’

Esempio:
Dati nel file dati.txt: 12 25 1 4 25 2 2 1 12 30
Dati nel file output.txt: 12 25 1 4 2 30

Soluzione

```
/*Programma di lettura dinamica di un vettore*/
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int *v, N; /* N è la dimensione del vettore */
  int a; /* in a metto il valore letto dal file*/
  FILE *fp;
  fp=fopen("dati.txt","r");

  N=0; v=NULL; /*a v non è assegnato alcun indirizzo*/
  while(feof(fp)!=0) /*Finché il file non è finito*/
  {
        a=leggi(fp); /*Leggi un valore intero da fp*/
        /*Se a non è già stato inserito in v*/
                /*aggiungi a a v*/
  }
  fclose(fp);
  /*Stampa v nel file output.txt*/
  /*Dealloca v*/
  return 0;
}
```

Per ripassare…

- Scriviamo il codice della funzione di lettura e di stampa del vettore:

```
int leggi(FILE *fp)
{
  int a;
  fscanf(fp,”%d”,&a);
  return a;
}
```

```
void stampa(int *v, int N)
{
   int i;
   FILE *fp;
   if(N==0) return;
   fp=fopen(“output.txt”,”w”);
   for(i=0; i<N; i++)
      fprintf(fp,”%d\n”,v[i]);
   fclose(fp);
}
```

/*Se a non è già stato inserito in v*/

- Che problema è?

- **RICERCA SEQUENZIALE**

- Scriviamo la funzione già nota (v. Cap. 2.7)

```
int ricerca_sequenziale(int* v, int x, int N)
{
  int i=0;

  while((i<N)&&(v[i]!=x)) i++;
  return i;
}
```

- La modifichiamo leggermente in modo che risponda con un valore di verità **1**/**0** se l'elemento **è**/**non è** stato trovato

Funzione ricerca_sequenziale **modificata**

```
int ricerca_sequenziale(int* v, int x, int N)
{
 int i=0;

 while(i<N)
    if(v[i]==x)
        return 1;
    else
        i++;

 return 0;
}
```

/*Aggiungi a a v*/

- Poiché non è possibile allungare il vettore con una `malloc` aggiungendogli locazioni, si può fare così:
  1. Si alloca un nuovo vettore w più grande di v di un elemento
  2. Si copia v in w
  3. Si dealloca v
  4. Si assegna a v il puntatore a w
  5. Si aggiorna N
- L'operazione 2 si può realizzare tramite una funzione apposita
- Le 1, 3-5 le lasciamo al livello di programma principale

Passo 2: Copia v in w

```
void copia(int*v, int *w, int N)
{
 int i;
 for(i=0; i<N; i++)
    w[i]=v[i];
}
```

Passi 1-5

1. Si alloca un nuovo vettore w più grande di v di un elemento
   - `w=(int*)malloc(sizeof(int)*(N+1));`
2. Si copia v in w e si aggiunge il nuovo valore in coda
   - `copia(v,w,N);`
   - `w[N]=a;`
3. Si dealloca v
   - `free(v);`
4. Si assegna a v il puntatore a w
   - `v=w;`
5. Si aggiorna N
   - `N++;`

Ritorniamo al programma principale

```
/*Programma di lettura dinamica di un vettore*/
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int *v, N, *w; /* N è la dimensione del vettore */
   int a; /* in a metto il valore letto dal file*/
   FILE *fp;
   fp=fopen("dati.txt","r");

   N=0; v=NULL; /*a v non è assegnato alcun indirizzo*/
   while(feof(fp)!=0) /*Finché il file non è finito*/
   {
          a=leggi(fp); /*Leggi un valore intero da fp*/
          if(!ricerca_sequenziale(v,a,N))
          {
                    w=(int*)malloc(sizeof(int)*(N+1));
                    copia(v,w,N);
                    w[N]=a;
                    free(v);
                    v=w;
                    N++;
          }
   }
   fclose(fp);
   stampa(v,N);
   free(v); /*Bisogna sempre deallocare v prima di uscire!*/
   return 0;
}
```

Esercizi

- Nell'esercizio precedente l'elemento `a`, se presente, veniva inserito in coda al vettore `v` (indice `N-1`). Si modifichi l'esercizio precedente in modo che l'elemento `a`, se presente, venga inserito in testa al vettore (indice `0`).

- Scrivere una funzione C `concatena` che, ricevendo in ingresso una stringa `s`, una stringa `t`, ed un intero `i`:
  - Restituisca una nuova stringa `s_new` in cui `t` è inserita a partire dalla posizione `i`, se `0<=i<=strlen(s)+1`
  - Restituisca la stringa `s` altrimenti
  - Esempio:
    - `s`←"**ciamo**"; `t`←"**Saluti**"; `i=2;`
    - L'esito di `nuova=concatena(s,t,i);` è tale che `nuova`←"**ci**Saluti**amo**"
  - Note:
    - ricordarsi il carattere `'\0'` a fine stringa
    - Non è richiesta la deallocazione di `s` e `t`

Introduzione alle liste

- Il vettore è un'ottima soluzione per tante possibili applicazioni ma presenta limiti dovuti al fatto che dobbiamo sempre dichiarare in anticipo la sua dimensione
- Le operazioni per «accorciarlo» ed «allungarlo» non sono semplicissime e impiegano tempo
- Se invece disponessimo di un tipo di dato più flessibile, in grado di «accorciarsi» ed «allungarsi» più facilmente?
- **Soluzione**: tipi di dati «astratti» implementati mediante puntatori

Le liste

- Sono un tipo di dato astratto (L) definito come sequenza di variabili di tipo semplice o strutturato qualsiasi (T).
- Su L sono definite le seguenti operazioni:
  - `CONS: L x T → L`
    - Prende una lista e un'istanza di T, e restituisce una lista in cui l'istanza di T figura in testa alla lista
  - `HEAD: L → T`
    - Restituisce l'istanza T in testa alla lista
  - `TAIL: L → L`
    - Restitusce la lista privata dell'istanza HEAD(L)
  - `ISEMPTY: L → {0,1}`
    - Restituisce 1 se la lista non contiene istanze di T (è vuota), 0 altrimenti

Importanza delle liste

- Le liste hanno un ruolo fondamentale nelle soluzioni di problemi complessi, ad esempio:
  - la gestione dei processi nei sistemi operativi (schedulazione), in cui i vari processi pronti, in attesa, sospesi, vengono gestiti attraverso *liste*, chiamate ***code*** per via della filosofia di accesso a ciascun dato (FIFO – First In First Out)
  - la gestione della memoria nelle chiamate (anche ricorsive) delle funzioni, attraverso la memorizzazione delle chiamate in un'opportuna lista chiamata ***stack*** (o pila), in cui la chiusura delle chiamate è in un ordine temporale opposto a quello di chiamata: l'ultima chiamata è anche la prima ad essere chiusa (LIFO – Last In First Out)
  - implementazione del sistema di paginazione/segmentazione nei sistemi operativi

Altri problemi reali in cui si usano liste e varianti (pile e code)

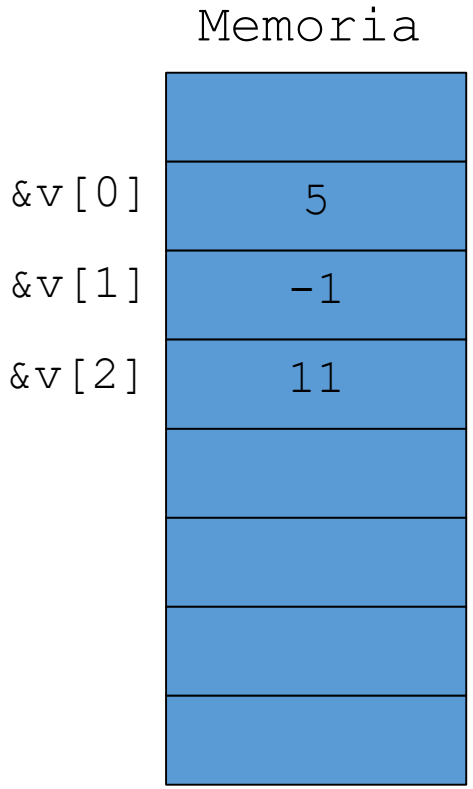
- Tutti i problemi in cui è necessaria una forma di «arbitraggio» fra esigenze «concorrenti»:
  - Passeggeri di un certo volo
  - Aerei in arrivo/partenza
  - Studenti in attesa di borsa di studio
  - Graduatorie concorsuali
  - Beni/servizi
  - Reti di calcolatori
- Approfondimenti
  - T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, *Introduzione agli algoritmi e strutture dati*, McGraw-Hill, 2005

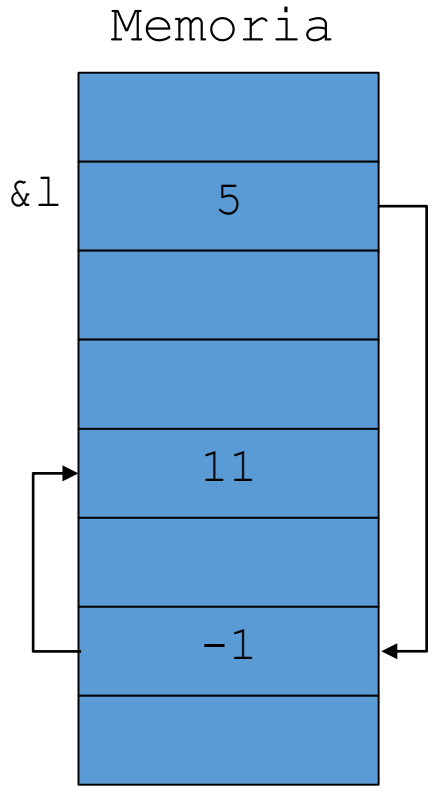
Implementazione di liste

- Per come sono state definite, le liste e relative funzioni possono essere facilmente realizzate con vettori
- Tuttavia, le liste non presumono che ciascuna istanza di T figuri in locazioni di memoria *contigue*: ciò ci slega dall'implementazione di liste con vettori sfruttando al massimo la *paginazione* della memoria
- Un'implementazione del tipo astratto «lista» che utilizza questo principio, prende il nome di **lista concatenata**

Differenza fra vettore e lista concatenata

- Per semplicità pensiamo a contenuti interi





Definizione in C del tipo lista concatenata

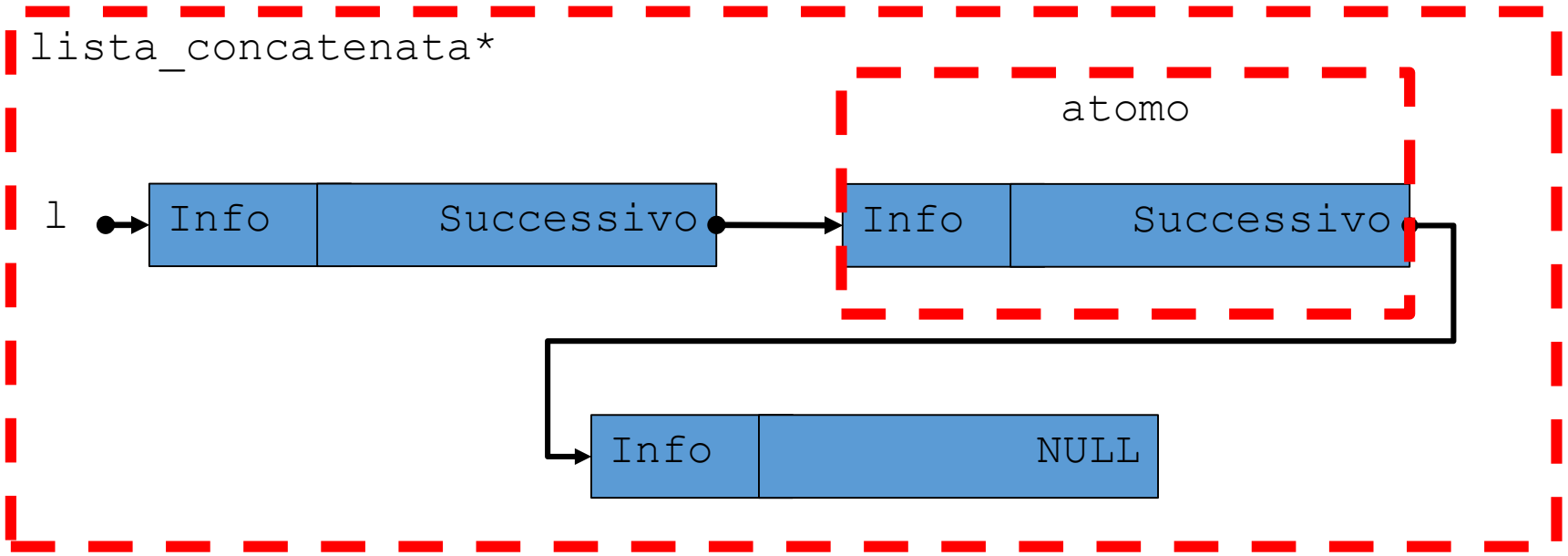
Definizione di tipo:

```
typedef struct atomo
{
 tipo_elemento Info;
 struct atomo *Successivo;
} lista_concatenata;
```

Dichiarazione di variabile:

```
lista_concatenata *l;
```

Rappresentazione grafica di una lista concatenata



- Convenzionalmente l'ultimo elemento di lista ha un puntatore a NULL

Funzioni ISEMPTY, TAIL e HEAD

```
int ISEMPTY(lista_concatenata *l)
{
  /* usiamo NULL come indicatore di lista vuota*/
  return l==NULL;
}

lista_concatenata* TAIL(lista_concatenata *l)
{
  /* implementazione «base» */
  return l->Successivo;
}

tipo_elemento HEAD(lista_concatenata *l)
{
  return l->Info;
}
```

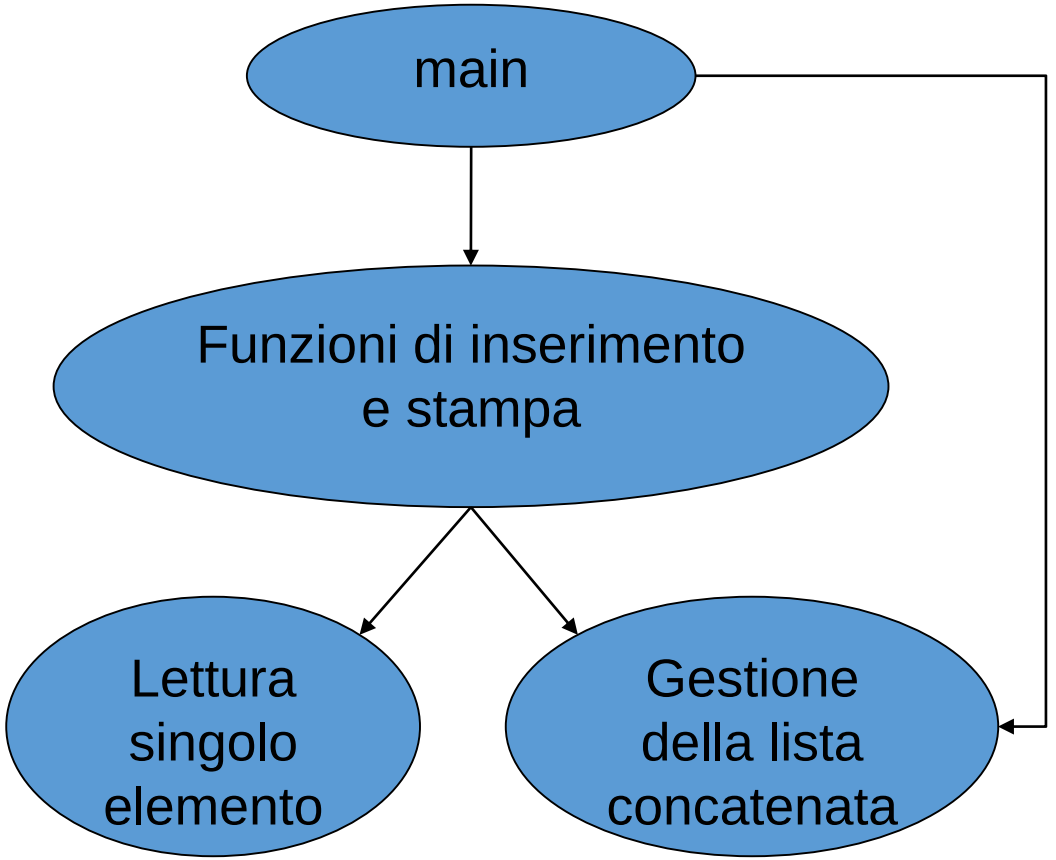
Funzione CONS

```
lista_concatenata * CONS(lista_concatenata *l, tipo_elemento t)
{
  lista_concatenata *testa;

  testa=(lista_concatenata *)malloc(sizeof(lista_concatenata));
  testa->Info=t;
  testa->Successivo=l;

  return testa;
}

void main()
{
  tipo_elemento x;
  lista_concatenata *l=NULL; /*lista inizialmente vuota*/

  x=assegna_elemento();
  l=CONS(l,x); /*inserimento di x in l*/
  … /*continua*/
```

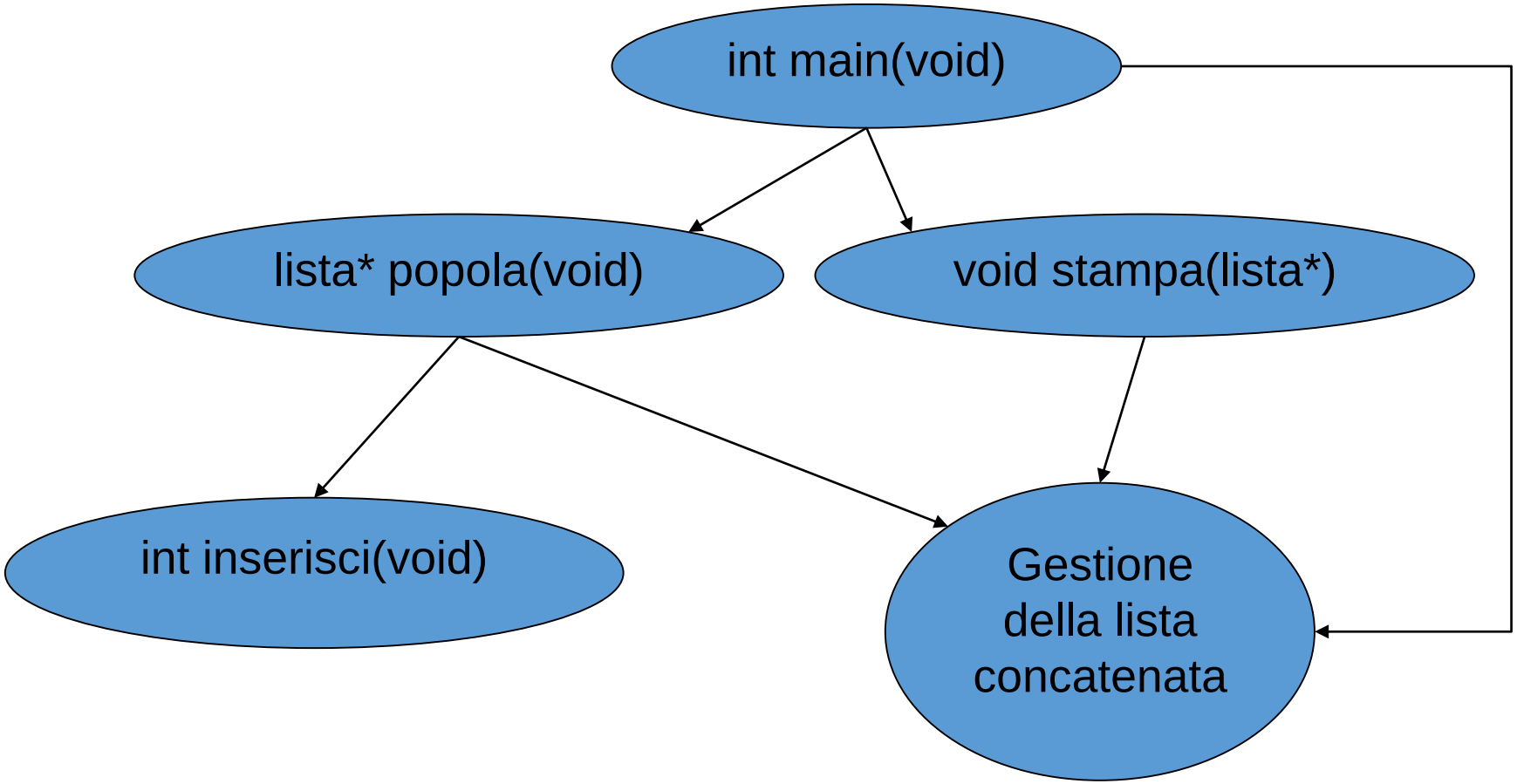
Esercizio 1

- Scrivere un programma C che inserisca in una lista concatenata un certo numero di interi fino alla lettura di uno zero
- Stampare a video tutti i valori letti
- Modularizzare il codice

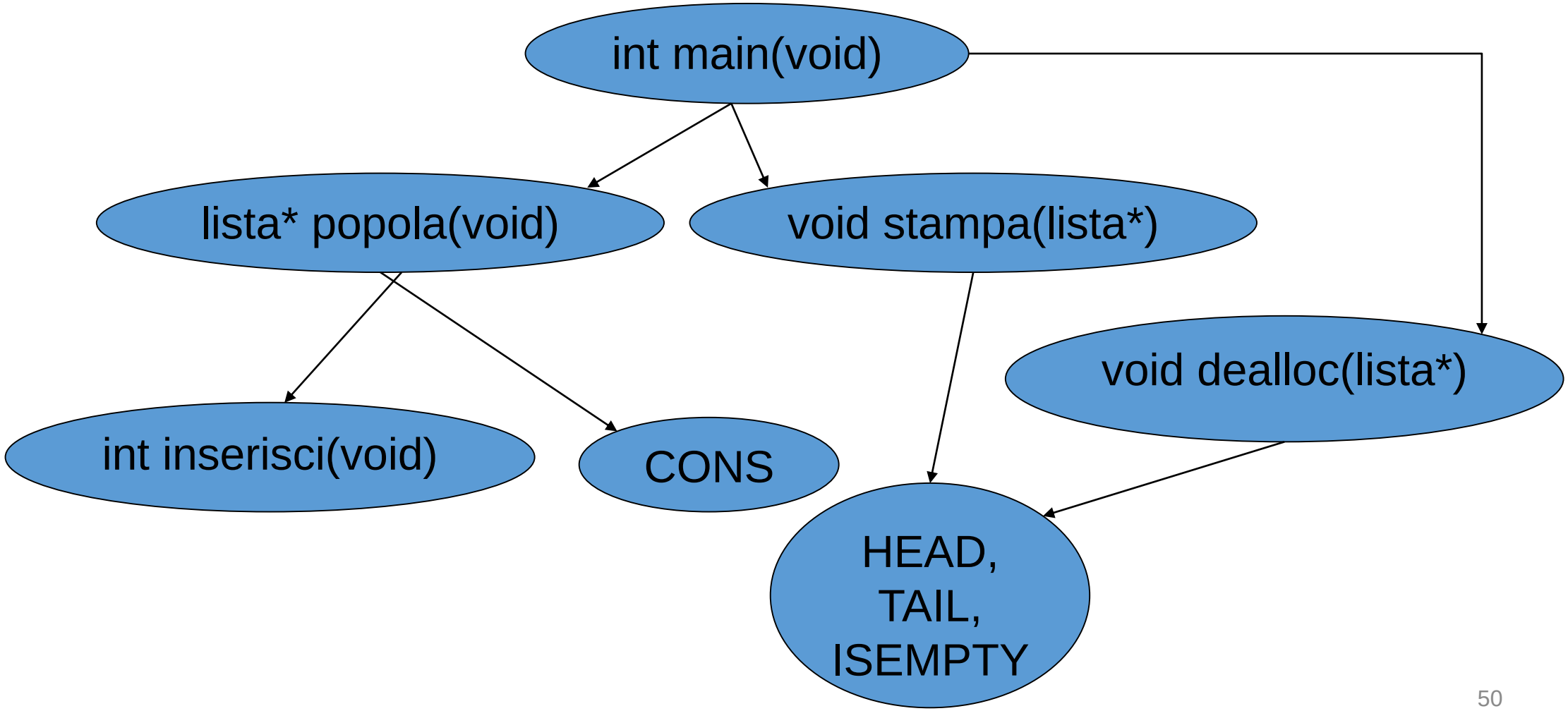
Vista top-down



Espansione della vista



Espansione dell'ultimo nodo



Esercizi

- Utilizzando la lista concatenata dell'esercizio precedente:
  2. Scrivere una funzione che conti il numero di atomi della lista. Sia con il tipo di dato originale che quello modificato nell'esercizio precedente.
  3. Scrivere una funzione che estragga il puntatore all'atomo i della lista a partire dalla testa (atomo in testa: i=0). Se i eccede il numero di elementi presenti, restituisca NULL.
  4. Scrivere una funzione che scambi gli elementi presenti nei puntatori agli atomi i e j di una lista
  5. Scrivere una funzione che estragga il valore del massimo elemento della lista
  6. Scrivere una funzione che estragga il puntatore all'atomo contenente il massimo elemento della lista
  7. Scrivere una funzione che ordini una lista secondo l'algoritmo selection-sort

Esercizio 8

- Scrivere un programma C che, leggendo da file con nome letto da tastiera, una tripla di valori numerici per riga, due **int** e un **float**, li memorizzi in una opportuna variabile strutturata e inserisca progressivamente la variabile letta in una lista.
- Stampi infine a video il numero degli elementi della lista per i quali la somma dei due interi è maggiore del valore float.
- Per far questo:
  - Si definisca il tipo strutturato di cui sopra e lo si chiami `Valori` e il relativo tipo associato alla lista, da chiamare `Lista`.
  - Si adattino le funzioni precedenti al tipo testé definito.
  - Si scriva una funzione che legga una riga di file, **già aperto altrove**, secondo il formato indicato e la restituisca sotto forma di una variabile di tipo `Valori`. Si chiami `leggi` detta funzione.
  - Si scriva una funzione di stampa a video della lista secondo i requisiti richiesti.
  - Si scriva la relativa funzione `main`.

Definizione dei tipi necessari

```
typedef struct {
     int v1, v2;
     float v3;
} Valori;

typedef struct Elemento_di_Lista
{
     Valori elemento;
     struct Elemento_di_Lista* successivo;
} Lista;
```

Adattamento funzione ISEMPTY

```
/*Funzione generica*/
int ISEMPTY(lista_concatenata *l)
{
 return l==NULL;
}

/*Funzione adattata*/
int ISEMPTY(Lista *l)
{
 return l==NULL;
}
```

Adattamento funzione TAIL

```
/* Funzione generica */
lista_concatenata* TAIL(lista_concatenata *l)
{
 return l->Successivo;
}


/* Funzione adattata */
Lista* TAIL(Lista *l)
{
 return l->Successivo;
}
```

Adattamento funzione HEAD

```
/* Funzione generica */
tipo_elemento HEAD(lista_concatenata *l)
{
 return l->Info;
}


/* Funzione adattata */
Valori HEAD(Lista *l)
{
 return l->elemento;
}
```

Adattamento funzione CONS

```
/* Funzione generica */
lista_concatenata* CONS(lista_concatenata *l, tipo_elemento t)
{
  lista_concatenata *testa;

  testa=(lista_concatenata *)malloc(sizeof(lista_concatenata));
  testa->Info=t;
  testa->Successivo=l;

  return testa;
}

Lista* CONS(Lista *l, Valori t) /*Funzione adattata*/
{
  Lista *testa;

  testa=(Lista *)malloc(sizeof(Lista));
  testa->elemento=t;
  testa->Successivo=l;

  return testa;
}
```

Funzione leggi

```
Valori leggi(FILE* f)
{
     Valori v;

     fscanf(f, "%d %d %f", &v.v1,&v.v2,&v.v3);

     return v;
}
```

Funzione stampa

```
void stampa(Lista *l)
{
     float s;
     Valori v;

     while (!ISEMPTY(l)) //Finché non è vuota…
     {
          v=HEAD(l);//…estraggo elemento in testa…
          s=v.v1+v.v2; //…elaboro..
          if (s>v.v3)
               printf("%s\n",v.nome);
          l=TAIL(l);//…passo all'elemento successivo
     }
}
```

Funzione main

```
int main()
{
        Lista *l;
        char nomefile[20];
        FILE *fp;
        Valori v;

        printf("Immetti il nome del file:\n");
        scanf("%s",&nomefile[0]);

        fp=fopen(nomefile, "r");
        l=NULL; /*Una lista vuota va inizializzata a NULL*/
        while (!feof(fp))
        {
                v=leggi(fp);
                l=CONS(l,v);
        }
        fclose(fp);

        stampa(l);

        return 0;
}
```

COSA MANCA A QUESTO CODICE?

Funzione di deallocazione

- Scrivere una funzione che deallochi tutta la lista prima di uscire dalla main

```
void dealloca(Lista *l) /*la lista va deallocata tutta*/
{
      Lista *p=l;

      while(!ISEMPTY(p))
      {
            l=TAIL(l);
            free(p);
            p=l;
      }
}
```

Funzione main corretta

```
int main()
{
        Lista *l;
        char nomefile[20];
        FILE *fp;
        Valori v;

        printf("Immetti il nome del file:\n");
        scanf("%s",&nomefile[0]);

        fp=fopen(nomefile, "r");
        l=NULL; /*Una lista vuota va inizializzata a NULL*/
        while (!feof(fp))
        {
                v=leggi(fp);
                l=CONS(l,v);
        }
        fclose(fp);

        stampa(l);
        dealloca(l); /* Dealloco tutti gli elementi della lista */
        free(l); /* Termino la deallocazione */
        return 0;
}
```

Per saperne di più

- Ceri, Mandriola, Sbattella, *Informatica – arte e mestiere*, Capp. 5, 10, McGraw-Hill
- Kernighan, Ritchie, *Il linguaggio C*, Cap. 5, Pearson-Prentice Hall
- T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, *Introduzione agli algoritmi e strutture dati*, McGraw-Hill, 2005