Algoritmi e Strutture Dati I: Sperimentazioni

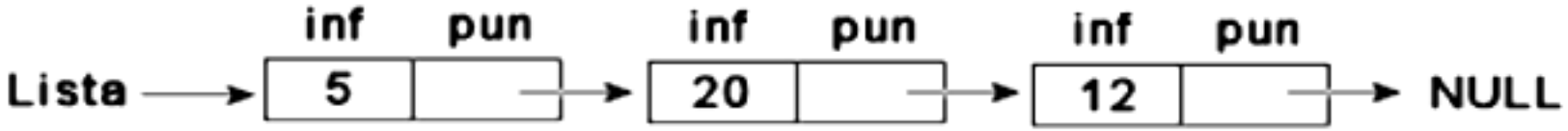
Stefania Montani
stefania.montani@mfn.unipmn.it

Giorgio Leonardi
giorgio.leonardi@mfn.unipmn.it

Le liste: implementazione con memoria dinamica

Liste puntate

- **Una lista è una collezione di elementi omogenei.**
- **A differenza dell'array, la dimensione di una lista non è nota a priori e può variare nel tempo. Solitamente, la lista parte con dimensione pari a 0 e aumenta man mano che viene riempita.**
- **Ogni elemento (nodo) nella lista ha uno o più campi contenenti informazioni, e, necessariamente, deve contenere un puntatore per mezzo del quale è legato all'elemento successivo**
- **Una lista puntata (semplice) ha una gestione sequenziale, in cui è sempre possibile individuare la testa e la coda della lista.**



Implementazione in C

- **Per definire la struttura di un elemento di una lista bisogna utilizzare un puntatore alla stessa struttura,che punterà all'elemento successivo:**

```
typedef int Elemento;

// oppure

/* typedef struct elemento {
   int campo1;
   char campo2[20];
   ...
} Elemento; */

typedef struct Nodo_lista {
   Elemento inf;
   struct Nodo_lista *next;
} Lista;
```

Operazioni sulle liste

- **Le operazioni che agiscono su una lista rappresentano gli operatori elementari che agiscono sulle variabili di tipo lista**
- **Corrispondono a dei sottoprogrammi (funzioni o procedure)**
- **Operazioni tipiche:**
  - **Inizializzazione: definizione della struttura dati che rappresenta il nodo**
  - **Inserimento in testa di un nodo**
  - **Inserimento in coda di un nodo**
  - **Inserimento di un nodo all'interno della lista**
  - **Verifica lista vuota**
  - **Ricerca di un elemento**
  - **Stampa a video del contenuto della lista**

Inizializzazione

```
#include <stdlib.h>
#include "tipi.h"


Lista *makeLista() {
   return NULL;
}

int emptyL(Lista *l) {
   return (l == NULL);
}

Elemento primo(Lista *l){
   if (l != NULL)
     return (l->inf);
}
```

Creazione di un nodo

```
Lista *creaNodo (Elemento el) {
      Lista *l;

      l = (Lista *) malloc(sizeof(Lista));
      if (l != NULL) {
            l->inf = el;
            l->next = NULL;
      }

      return l;
}
```

Funzione visualizza_lista()

```
void visualizza_lista(Lista *l) {
   Lista *p;
   p= l;

   printf("\n lista : \n");
   while(p != NULL) {
       printf("\t %d \n", p->inf); /* visualizza
   l'informazione */
       p = p->next; /* scorre la lista di un elemento
   */
   }
   printf("NULL\n\n");
}
```

Inserimento in testa

```
Lista *inserisci(Elemento el,Lista  *l){
  Lista *l1;
  l1 = creaNodo (el);

  if (l1 != NULL) {
    l1->next = l;
    l= l1;
  }

  return l;
}
```

Eliminazione in testa

```
Lista *rimuovi(Lista *lptr){
  if (lptr != NULL) {
     Lista *l = lptr;
     lptr= lptr->next;
     free(l);
  }

  return lptr;
}
```

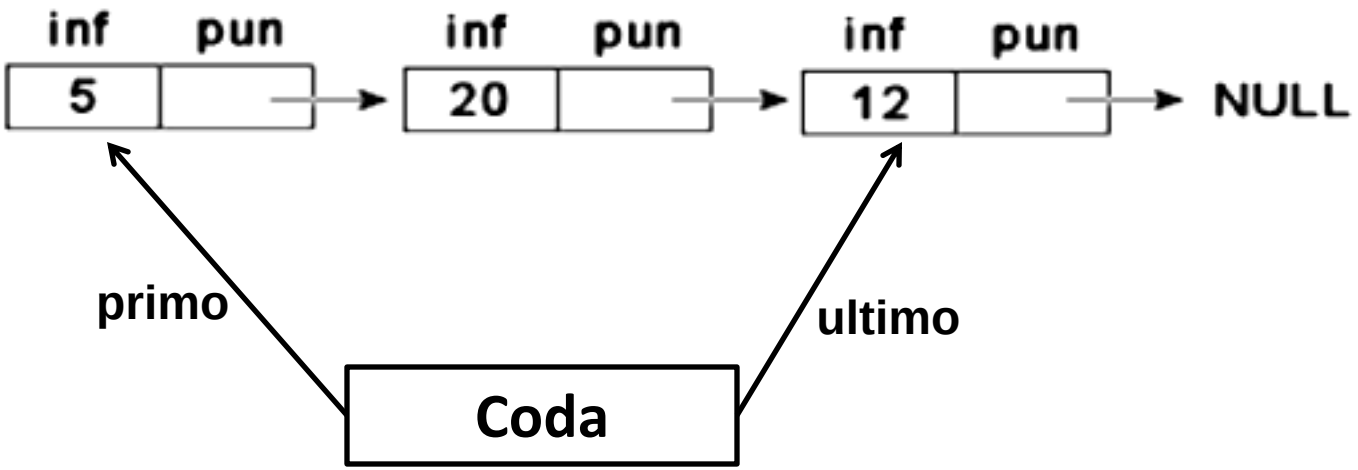
Altre funzioni

- **Come posso implementare le funzioni mancanti?**
  - **Inserimento di un nodo in coda**
  - **Inserimento di un nodo in mezzo alla lista**
  - **Eliminazione di un nodo dalla lista**

Code implementate con liste

Struttura dati coda

```
typedef struct coda {
    Lista *primo;
    Lista *ultimo;
} Coda;
```



Inizializzazione della coda

```
Coda makeCoda(){
   Coda c;
   c.primo= c.ultimo= NULL;
   return c;
}

int emptyC(Coda c){
   return emptyL(c.primo);
}

Elemento first(Coda c){
   return primo(c.primo);
}
```

Inizializzazione della coda (2)

```
Coda *makeCoda(){
   Coda *c = (Coda *)malloc (sizeof(Coda));
   c->primo= c->ultimo= NULL;
   return c;
}

int emptyC(Coda *c){
   return emptyL(c->primo);
}

Elemento first(Coda *c){
   return primo(c->primo);
}
```

Aggiunta di un elemento

```
void enqueue(Elemento el, Coda *c){
   Lista *l;
   l= creaNodo(el);

   if (l != NULL) {
       if(emptyC(*c)) {
              c->primo = l;
              c->ultimo = l;
       } else {
              c->ultimo->next = l;
              c->ultimo = l;
       }
   }
}
```

Rimozione di un elemento

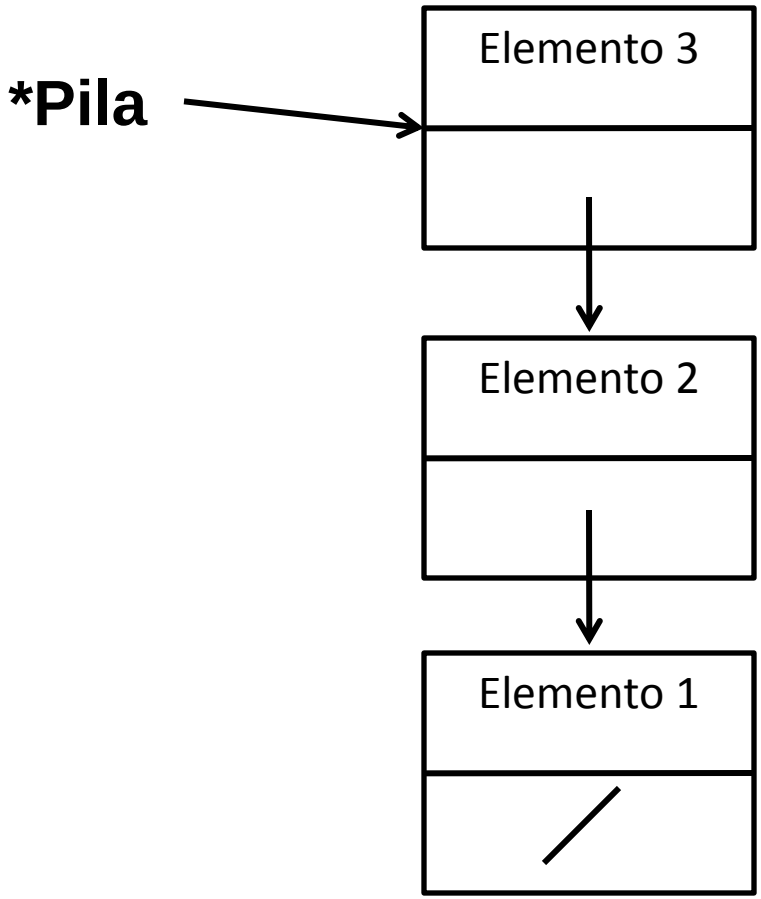
```
Elemento dequeue(Coda *c){
   Elemento el;

   if (!emptyC(c)) {
       el= first(c);
       c->primo= rimuovi(c->primo);
       return el;
   }
}
```

Stack implementati con liste

Struttura dati pila

```
typedef Lista Pila;
```



Inizializzazione della pila

```
Pila *makePila(){
  return makeLista();
}

int emptyP(Pila *p){
  return emptyL(p);
}
```

Gestione della pila

```
Pila *push(Elemento el,Pila *p){
   return inserisci(el, p);
}

Elemento top(Pila *p){
   if (!emptyP(p))
        return primo(p);
}

Elemento pop(Pila **p){
   Elemento el;
   if (!emptyP(*p)) {
        el= primo(*p);
        *p= rimuovi(*p);
        return el;
   }
}
```

Esercitazione di oggi

Un gioco di carte

- **Si vuole implementare un meccanismo per far giocare al computer il classico gioco della "scopa"**

- **Una carta è implementata come una struttura:**

```
typedef struct elemento {
   int valore;
   char seme; // (C = Cuori; Q = Quadri; F =
     Fiori; P = Picche)
} Elemento;
```

Situazione al tavolo

Giocatore 1

Giocatore 4

Tavolo

Giocatore 2

Giocatore 3

Mazzo di carte

Situazione iniziale

Giocatore 1:
0 carte

Giocatore 4:
0 carte

Tavolo: 0 carte

Giocatore 2:
0 carte

Giocatore 3:
0 carte

Mazzo di carte: 40 carte (mischiate)

Situazione alla prima mano

Giocatore 1:
3 carte

Giocatore 4:
3 carte

Tavolo: 4 carte

Giocatore 2:
3 carte

Giocatore 3:
3 carte

Mazzo di carte: 24 carte

Implementazione

Giocatore 1:
Stack

Giocatore 4:
Stack

Tavolo: Lista

Giocatore 2:
Stack

Giocatore 3:
Stack

Mazzo di carte: Coda

Da fare

- **Implementare:**
  - **Ogni giocatore come uno stack**
  - **Le carte sul tavolo come una lista**
  - **Il mazzo di carte come una coda**
  - **Per la gestione di liste, stack e code è possibile utilizzare la libreria fornita sul sito, oppure il codice di queste slides**

- **Situazione iniziale:**
  - **Giocatori: 4 stack inizializzati e vuoti**
  - **Carte sul tavolo: lista inizializzata e vuota**
  - **Mazzo di carte: coda di 40 carte "ordinate":**
    - **1 di picche, 2 di picche, 3 di picche, …, 10 di picche**
    - **1 di quadri, 2 di quadri, 3 di quadri, …, 10 di quadri**
    - **…**

Da fare

- **Prima mano:**
  - **“mescolare” le carte nel mazzo: per 40 volte:**
    - **Scelgo 2 carte (due nodi della coda) e le scambio**
      - **Scambiando il contenuto dei nodi, oppure**
      - **scambiando le posizioni dei puntatori**
  - **Dare le carte ai giocatori. Per 3 volte:**
    - **Dequeue di una carta dal mazzo e push della carta al giocatore 1**
    - **Dequeue di una carta dal mazzo e push della carta al giocatore 2**
    - **…**
  - **Mettere 4 carte sul tavolo:**
    - **Dequeue di una carta dal mazzo, e inserimento della carta nella lista del tavolo (per 4 volte)**

Da fare

- **Fase di gioco:**
  - **Per ogni mano, ogni giocatore gioca a turno una carta:**
    - **Giocatore uno fa una "pop" e aggiunge la carta al tavolo**
    - **Giocatore due fa una "pop" e aggiunge la carta al tavolo**
    - **…**
    - **Fino a che le carte in mano sono esaurite**
  - **A questo punto, si danno altre 3 carte a testa ai giocatori. Per 3 volte:**
    - **Dequeue di una carta dal mazzo e push della carta al giocatore 1**
    - **Dequeue di una carta dal mazzo e push della carta al giocatore 2**
    - **…**
  - **Tutto ciò finchè le carte del mazzo sono terminate, e i giocatori hanno esaurito le carte dell'ultima mano**

Da fare

- **Implementare questo “scheletro” di gioco, senza la logica del gioco (come si “prende” una carta dal tavolo, o come si contano i punti)**

- **Dare la possibilità di segure il gioco “passo a passo”:**
  - **Per ogni carta giocata da un giocatore, visualizzare la carta giocata e la situazione del gioco:**
    - **che carte sono in mano ad ogni giocatore**
    - **che carte sono sul tavolo**
    - **che carte sono ancora nel mazzo**

Relazione

- **La relazione conterrà (almeno):**
  - **Una veloce descrizione del “gioco”**
  - **La descrizione delle vostre scelte progettuali**
  - **La descrizione delle strutture dati utilizzate**
  - **L’algoritmo di “mescolamento” delle carte e relativo calcolo della complessità**

Implementazione della libreria

- **Per gestire nodi il cui campo inf è composto da campi di una struttura, è necessario apportare alcune modifiche:**

```
typedef struct elemento {
   int valore;
   char seme; // (C = Cuori; Q = Quadri; F =
   Fiori; P = Picche)
} *Elemento;


typedef struct Nodo_lista {
   Elemento *inf;
   struct Nodo_lista *next;
} Lista;
```

Le funzioni viste precedentemente funzionano SOLO quando il campo inf è un tipo di dato di base (int, char)

Implementazione della libreria

- **Per gestire nodi il cui campo inf è composto da campi di una struttura, è necessario apportare alcune modifiche:**

```
typedef struct elemento {
   int valore;
   char seme; // (C = Cuori; Q = Quadri; F =
   Fiori; P = Picche)
} *Elemento;


typedef struct Nodo_lista {
   Elemento *inf;
   struct Nodo_lista *next;
} Lista;
```

E' bene trattare il campo inf come un puntatore alla struttura per allocarlo, disallocarlo e restituirlo in un unico «blocco» alle funzioni che lo richiedono

Generatori di numeri random

- **Il linguaggio C offre le seguenti funzioni random:**
  - `srand (time(NULL));` → da richiamare una sola volta ad inizio programma
  - `int rand() % <K>;` → genera un numero da 0 a (K-1)
  - `int K + rand() % (N-K);` → genera un numero da K a N(>K)

- `srand(int seed)` e `rand()` **sono fornite dalla libreria** `<stdlib.h>`
- `time (time_t *T)` **è fornita da** `<time.h>`