9. I tipi di dato *astratti*

La nozione di algoritmo è indipendente dalla sua codifica nel linguaggio di programmazione; analogamente con tipo di dato astratto si intende la specifica di un tipo di dato indipendente dal linguaggio di programmazione.

9.1. Nozione di tipo astratto

Un *tipo di dato astratto* è costituito da tre componenti:

- il *dominio di interesse*, che rappresenta l'insieme di valori che costituiscono il tipo astratto, ed eventuali *altri domini*, che rappresentano altri valori coinvolti
- un insieme di *costanti*, che denotano valori del dominio di interesse
- un insieme di *operazioni* che si effettuano sugli elementi del dominio di interesse, utilizzando eventualmente elementi degli altri domini.

9.1.1. Il tipo astratto *Booleano*

- dominio di interesse: $\{vero, falso\}$
- costanti: *true* e *false*, che denotano rispettivamente *vero* e *falso*
- operazioni: `and, or, not`

Nota: altre operazioni si possono definire a partire dalle operazioni elementari.

9.2. Implementazione dei tipi di dato astratti

Il tipo di dato astratto deve essere realizzato usando i costrutti del LDP:

1. rappresentazione dei *domini* usando i *tipi concreti* del LDP

2. codifica delle *costanti* attraverso i *costrutti* del LDP

3. realizzazione delle *operazioni* attraverso opportune *funzioni* del LDP

9.2.1. Implementazione in C del tipo *Booleano*

Dominio:
il valore *falso* si rappresenta con il valore 0.
il valore *vero* si rappresenta tutti il valore 1.

Costanti:

```
typedef int bool;
#define TRUE  1
#define FALSE 0
```

Operazioni: si possono utilizzare direttamente le operazioni del C:

```
&&,||,!
```

9.2.2. Osservazioni

- Per uno stesso tipo astratto si possono avere *più implementazioni* diverse.

- Una implementazione potrebbe avere delle *limitazioni* rispetto al tipo di dato astratto.

- Un tipo di dato può essere *parametrico*, cioè definito a partire da uno o più tipi di dato (struttura di dati).

NB nel seguito ometteremo il termine astratto, a cui abitualmente viene associata una caratterizzazione matematica e ci riferiremo semplicemente ai tipi di dato fornendo per essi una specifica informale e diverse possibili implementazioni.

9.3. Il tipo di dato *lista*

Una lista è una sequenza di dati omogenei.

Se `TipoInfo` è il tipo degli elementi il dominio del tipo di dato lista è costituito da tutte le sequenze di elementi di tipo `TipoInfo`.

Per denotare una lista si usa abitualmente la notazione parentetica:

- `()` denota la lista vuota
- `(e1 e2 ...)` denota una lista il cui primo elemento è `e1`, il secondo `e2` etc.

9.3.1. Il tipo di dato *lista*: operazioni

Le operazioni sul tipo di dato lista sono:

- `primo`: restituisce il primo elemento della lista (CAR)
- `resto`: restituisce una lista senza il primo elemento (CDR)
- `add`: aggiunge un elemento alla lista (CONS)

Esempio: `add(e2,add(e1,()))`

9.3.2. Operazioni del tipo di dato *lista* in C

```
typedef ... TipoInfo;
typedef ... TipoLista;

// restituisce true se scl e' la lista vuota
bool emptyLista(TipoLista lis);
// restituisce il primo elemento di una lista non vuota
void primoLista(TipoLista lis, TipoInfo * e);
// modifica la lista togliendo il primo elemento se non vuota
void restoLista(TipoLista * resto);
// assegna al parametro la lista vuota
void initLista(TipoLista * lis);
// modifica la lista aggiungendo l'elemento e in prima posizione
void addLista(TipoLista * lis, TipoInfo e);
```

9.3.3. Implementazione tramite scl

Una prima implementazione del tipo di dato lista si può ottenere direttamente tramite le strutture collegate lineari.

La lista vuota è rappresentata da `NULL` e la definizione di tipo è analoga a quella delle scl.

```
struct nodoLista {
  TipoInfo info;
  struct nodoLista *next;
};
typedef struct nodoLista NodoLista;
typedef NodoLista *TipoLista;
```

Le operazioni hanno una corrispondenza diretta con quelle viste per le scl.

9.3.4. Altre Implementazioni tramite scl

È importante notare che questa non è l'unica rappresentazione per le liste, che si può realizzare con le strutture collegate.

- con la memorizzazione del numero di elementi oltre che del puntatore iniziale;
- con puntatore sia all'elemento precedente che al successivo (*lista doppia*);

9.3.5. Tipo *lista*: Caratterizzazione funzionale

Le operazioni del tipo di dato lista si possono caratterizzare anche in modalità funzionale analogamente a quanto visto a proposito delle strutture collegate lineari.

```
// restituisce true se lis e' la lista vuota
bool emptyLista(TipoLista lis);
// restituisce il primo di una lista non vuota
TipoInfo primoLista(TipoLista lis);
// restituisce il resto di una lista non vuota
TipoLista restoLista(TipoLista lis);
// restituisce la lista vuota
TipoLista initLista();
// restituisce una lista costruita aggiungendo a lis
// l'elemento e in prima posizione
TipoLista addLista(TipoLista lis, TipoInfo e);
```

Anche in quesito caso la corrispondenza con le funzioni realizzate per le scl è immediata.

Alcuni linguaggi di programmazione forniscono un'implementazione del tipo di dato lista..

9.3.5.1. Implementazione tramite array

Una implementazione del tipo di dato lista si può ottenere anche tramite array.

```
struct StructLista {
  TipoInfo * info;
  int Nelem;
};
typedef struct StructLista TipoLista;
```

La lista vuota è rappresentata da una struttura con `0` elementi.

L'implementazione delle operazioni del tipo lista può essere realizzata utilizzando il numero di elementi come indice dell'array (per una discussione dettagliata si rimanda alla implementazione del tipo di dato pila).

9.4. Il tipo di dato coda

Una *coda* è una sequenza di elementi omogenei in cui gli elementi si utilizzano secondo la regola:

> L'elemento che viene eliminato tra quelli presenti nella coda deve essere quello che è stato inserito per primo.

Si parla di gestione *FIFO* (per "*F*irst *I*n, *F*irst *O*ut").

La coda serve per elabore i dati nello stesso ordine in cui sono stati generati.

9.4.1. Realizzazione in C del tipo di dato coda

Indipendentemente da come verrà realizzato il tipo astratto, si possono specificare le definizioni di tipo:

```
typedef int bool;
typedef ... TipoElemCoda;
typedef ... TipoCoda;
```

la creazione di una coda vuota:

```
InitCoda(TipoCoda *pc);
```

ed i prototipi delle funzioni che realizzano le diverse operazioni:

```
bool TestCodaVuota(TipoCoda c);
void InizioCoda(TipoCoda c, TipoElemCoda *pv);
void InCoda(TipoCoda *pc, TipoElemCoda v);
void OutCoda(TipoCoda *pc, TipoElemCoda *pv);
```

9.4.2. Rappresentazione sequenziale delle code

Una coda è rappresentata mediante *un array* e *due indici*:

- `primo`
- `ultimo`.

Per la coda vuota: `primo` è pari a -1.

Usando due indici si evita di spostare tutti gli elementi verso l'alto ad ogni operazione di eliminazione di un elemento dalla coda.

9.4.2.1. Definizione di tipo

```
#define MaxCoda ...

typedef ..... TipoElemCoda;

typedef int TipoPosCoda;

struct StructCoda {
  TipoElemCoda coda[MaxCoda];
  TipoPosCoda primo, ultimo;
};

typedef struct StructCoda TipoCoda;
```

9.4.2.2. Operazioni: gestione coda vuota

```
void InitCoda(TipoCoda *c) {
/* inizializza la coda c ponendo a -1 i puntatori al primo e
   all'ultimo elemento della coda */
  c->primo = -1;
  c->ultimo = -1;
}  /* InitCoda */

bool TestCodaVuota(TipoCoda c){
```

```
/* restituisce TRUE se la coda c e' vuota, FALSE altrimenti */
  return (c.primo == -1);
}  /* TestCodaVuota */
```

9.4.2.3. Analisi del primo elemento della coda

```
void InizioCoda(TipoCoda c, TipoElemCoda *v)
/* restituisce in v il primo elemento della coda c senza modificare c */
{
  if (TestCodaVuota(c))
    printf("ERRORE: CODA VUOTA\n");
  else
    *v = c.coda[c.primo];
}  /* InizioCoda */
```

9.4.2.4. Uso circolare dell'array

Verifica di coda piena (due condizioni):

```
bool TestCodaPiena(TipoCoda c) {
/* restituisce TRUE se la coda c e' piena, FALSE altrimenti */

  if ((c.primo - c.ultimo == 1)||((c.ultimo - c.primo) == (MaxCoda-1)))
    return TRUE;
  else
    return FALSE;
}  /* TestCodaPiena */
```

9.4.2.5. Inserimento in coda

```
 {
void InCoda(TipoCoda *c, TipoElemCoda v)
/* inserisce l'elemento v all'ultimo posto della coda c */
  if (TestCodaPiena(*c))
    printf("ERRORE: CODA PIENA\n");
  else { /* posizionamento ultimo prossima posizione libera */
    if (c->primo == -1) { /* c vuota: */
      c->ultimo = 0;
      c->primo = 0;
    }
    else /* c non vuota: cambia solo l'ultimo elemento */
        c->ultimo = (c->ultimo + 1) \% MaxCoda;
```

```
    c->coda[c->ultimo] = v;
  }
}  /* InCoda */
```

9.4.2.6. Estrazione di un elemento dalla coda

```
void OutCoda(TipoCoda *c, TipoElemCoda *v){
/* elimina il primo record della coda c, restituendone il valore in v */
  if (TestCodaVuota(*c))
    printf("ERRORE: CODA VUOTA\n");
  else {
    *v = c->coda[c->primo];
    /* l'elemento eliminato era l'unico elemento presente nella coda */
    if (c->primo == c->ultimo) {
      c->ultimo = -1;
      c->primo = -1;
    }
    else /* c non vuota: cambia solo il primo elemento */
        c->primo = (c->primo + 1) \% MaxCoda;
  }
}  /* OutCoda */
```

Anche per le code si può usare una rappresentazione sequenziale tramite vettore dinamico.

9.4.3. Rappresentazione collegata delle code

Come per le pile, si può rappresentare una coda tramite una scl nella quale il primo elemento è il primo elemento della coda (ovvero quello che deve uscire per primo).

Per rappresentare la coda vuota: `NULL`

Per facilitare la realizzazione delle operazioni, si usa una rappresentazione mediante *scl e due puntatori*:

- un puntatore al primo elemento della coda
  (permette di realizzare *InizioCoda* e *OutCoda* a costo costante)
- un puntatore all'ultimo elemento della coda
  (permette di realizzare *InCoda* a costo costante)

9.4.3.1. Rappresentazione in C

```
/* definizioni preliminari */
```

```
typedef ..... TipoElemCoda;
typedef TipoElemCoda TipoInfoSCL;

/* definizione del tipo scl */
...


/* definizione del tipo coda */
struct StructCoda {
  TipoSCL primo, ultimo;
};
typedef struct StructCoda TipoCoda;
```

9.4.3.2. Gestione della coda vuota

```
void InitCoda(TipoCoda *c) {
/* inizializza la coda c ponendo a NULL i puntatori al primo e
   all'ultimo elemento della coda */
  InitSCL(&c->primo);
  InitSCL(&c->ultimo);
}  /* InitCoda */

bool TestCodaVuota(TipoCoda c) {
/* restituisce TRUE se la coda c e' vuota, FALSE altrimenti */
  return(emptySCL(c.primo));
}  /* TestCodaVuota */
```

9.4.3.3. Analisi del primo elemento della coda

```
void InizioCoda(TipoCoda c, TipoElemCoda *v) {
/* restituisce in v il primo elemento della coda c senza modificare c */
  primoSCL(c.primo, v);
}  /* InizioCoda */
```

9.4.3.4. Inserimento di un elemento nella coda

```
void InCoda(TipoCoda *c, TipoElemCoda v) {
/* inserisce l'elemento v all'ultimo posto della coda c */
  if (TestCodaVuota(*c)) {
    /* c e' vuota: l'elemento da inserire sara' sia
       il primo che l'ultimo elemento della coda */
```

```
    c->primo = malloc(sizeof(TipoNodoSCL));
    c->ultimo = c->primo;
  }
  else {
    c->ultimo->next = malloc(sizeof(TipoNodoSCL));
    c->ultimo = c->ultimo->next;
  }
  c->ultimo->info = v;
  c->ultimo->next = NULL;
}  /* InCoda */
```

9.4.3.5. Estrazione di un elemento dalla coda

```
void OutCoda(TipoCoda *c, TipoElemCoda *v) {
/* elimina il primo nodo della coda c, restituendone il valore in v */
  InizioCoda(*c, v); /* copia il valore del primo elemento
                        di c in v (se esiste) */
  delSCL(&c->primo); /* elimina il primo elemento
                                    di c (se esiste) */
  /* se l'elemento eliminato era l'unico elemento presente nella coda,
     allora si pone a NULL anche il puntatore all'ultimo elemento */
  if (c->primo == NULL)
    c->ultimo = NULL;
}  /* OutCoda */
```

9.5. Esercizio

Si consideri un record Persona contenente i campi nome e cognome di tipo stringa (lunghezza massima 50 caratteri). Si vuole scrivere un programma per gestire una fila di persone, per servirle in ordine di arrivo. Il primo elemento della fila è detto *capofila* e rappresenta la prossima persona che sarà servita. Quando il capofila è servito, esso viene rimosso dalla fila e la persona successiva diventa capofila. Quando una persona si aggiunge alla fila, viene inserita in ultima posizione. Tuttavia, una persona può essere raccomandata e, in tal caso, essere inserita in un punto arbitrario della fila.

9.5.1. Record `Persona`

```
const int STRLEN = 51;
```

```
struct persona {
        char nome[STRLEN];
        char cognome[STRLEN];
};

typedef struct persona Persona;
```

9.5.2. Implementazione tramite coda sequenziale

```
typedef Persona TipoElemCoda;

/* definizioni del tipo coda */
...

typedef struct StructCoda TipoCoda;
```

9.5.3. Realizzazione delle funzioni

```
bool TestCodaVuota(TipoCoda c);
void InizioCoda(TipoCoda c, TipoElemCoda *pv);
void InCoda(TipoCoda *pc, TipoElemCoda v);
void OutCoda(TipoCoda *pc, TipoElemCoda *pv);
```

9.5.4. Funzioni richieste per il tipo persona

```
void leggiPersona(file * f, Persona * p);
void scriviPersona(file * f, Persona * p);
```

9.5.5. Funzioni ausiliarie

```
void raccomandaPersona(TipoCoda * c, Persona * p, int pos);
int lunghezzaCoda(TipoCoda c);
int cercaPersona(TipoCoda c, Persona * p);
```

Per il tipo persona si richiede anche:

```
bool stessaPersona(Persona *p1, Persona* p2);
```

9.5.6. Implementazione della lunghezza

```
int lunghezzaCoda(TipoCoda c);
        if (primo > ultimo) return MaxCoda - (primo - ultimo);
```

```
        else if (primo < ultimo) return primo - ultimo;
        else return 1;
```

9.5.7. Implementazione della raccomandazione sequenziale

```
void raccomandaPersona(TipoCoda * c, Persona * p, int pos){
if (TestCodaPiena(*c)) printf("coda piena");
else {
        int Nelem = lunghezzaCoda(*c);
        if (Nelem < pos) InCoda(c,*p);
        else insertCoda(c,p,pos); //inserisci in posizione intermedia
        }
}
```

9.5.8. Inserisci in posizione intermedia

```
void inserisciCoda(TipoCoda * c, Persona * p, int pos){
        // Calcola la posizione di inserimento
        int j = (c->primo + pos - 1) % MaxCoda;
        int i = c->ultimo;
        while(i != j){
                // Sposta avanti di uno, da ultimo fino a j
                c->coda[(i+1)%MaxCoda] = c -> coda[i];
                i--;
                if(i < 0){
                        i = MaxCoda -1;
                }
        }
        c->coda[(j+1)%MaxCoda] = c -> coda[j];
        c->coda[j] = *p; // Inserisce p in j
        c->ultimo = (c -> ultimo + 1) % MaxCoda; // Aggiorna ultimo
}
```

9.5.9. Implementazione della raccomandazione collegata

```
void addPosSCL(TipoSCL * scl, TipoInfoSCL e, int n){
        if (n == 1) {
                TipoSCL temp = *scl;
                *scl = (TipoNodoSCL*) malloc(sizeof(TipoNodoSCL));
                (*scl)->info = e;
                (*scl)->next = temp;
```

```
        }
        else addPosSCL(&(*scl)->next, e, n-1);
}
```

9.6. Il tipo di dato pila

Una *pila* è una sequenza di elementi (tutti dello stesso tipo) in cui l'inserimento e l'eliminazione di elementi avvengono secondo la regola:

> L'elemento che viene eliminato tra quelli presenti nella pila deve essere quello che è stato inserito per ultimo.

Si parla di gestione *LIFO* ("*L*ast *I*n, *F*irst *O*ut").

9.6.1. Utilizzo delle pile: inverti input

Con una pila si risolve facilmente il problema di stampare in ordine inverso gli elementi letti.

- metto in pila gli elementi
- quando i dati sono finiti li estraggo dalla pila e li stampo

9.6.2. Utilizzo delle pile

Quando esistono diverse alternative nella soluzione di problemi:

- metto in pila ciascuna alternativa
- se l'alternativa corrente fallisce si prende la successiva alternativa memorizzata nella pila (backtracking)

Esempio: Ricerca di un cammino nella palude

9.6.3. Realizzazione in C del tipo di dato pila

La pila, come la lista, è un tipo di dato parametrico.
La costante `PilaVuota` indica una pila vuota.
Le operazioni sulla pila sono:

- `top`: restituisce l'elemento in cima alla pila
- `pop`: restituisce una pila senza il primo elemento
- `push`: aggiunge un elemento alla pila

Esempio: `push(e1,push(e2,PilaVuota))`

9.6.4. Caratterizzazione delle operazioni in C

```
typedef ... TipoElemPila;
typedef ... TipoPila;

// restituisce una pila vuota
void InitPila(TipoPila *pp);
// restituisce true se la p e' una pila vuota
bool TestPilaVuota(TipoPila p);
// restituisce l'elemento in cima alla pila
void TopPila(TipoPila p, TipoElemPila *pv);
// restituisce la pila senza l'elemento "top"
void Pop(TipoPila *pp, TipoElemPila *pv);
// aggiunge un elemento in cima alla pila
void Push(TipoPila *pp, TipoElemPila v);
```

9.6.5. Rappresentazione sequenziale delle pile

Una pila è rappresentata mediante:

- un *vettore di elementi* del tipo degli elementi della pila, riempito fino ad un certo indice con gli elementi della pila
- un *indice* che rappresenta la posizione nel vettore dell'elemento affiorante della pila

È necessario fissare la dimensione del vettore a tempo di compilazione. Si ha quindi un limite superiore al numero di elementi nella pila.

Si rende ulte un'operazione di verifica di pila piena:

– `bool TestPilaPiena(TipoPila p);`

9.6.5.1. Implementazione

Dichiarazioni di tipo:

```
#define MaxPila 100
typedef ... TipoElemPila;
struct tipoPila {
  TipoElemPila pila[MaxPila];
  int pos;
};
typedef struct tipoPila TipoPila;
```

Per la pila vuota: `p.pos` viene scelta pari a -1.

9.6.5.2. Operazioni: `InitPila,TestPilaVuota`

Inizializzazione di una pila:

```
void InitPila(TipoPila *p)
/* inizializza la pila p ponendo a -1 il puntatore all'elemento
   affiorante della pila */
{
  p->pos = -1;
}  /* InitPila */
```

Test pila vuota:

```
bool TestPilaVuota(TipoPila p)
/* restituisce TRUE se la pila p e' vuota, FALSE altrimenti */
{
  return (p.pos == -1);
}  /* TestPilaVuota */
```

9.6.5.3. Operazioni: `top`

Elemento affiorante della pila;

```
void TopPila(TipoPila p, TipoElemPila *v)
/* restituisce in v l'elemento affiorante della pila p,
   senza modificare la pila */
{
  if (TestPilaVuota(p))
    printf("ERRORE: PILA VUOTA\n");
  else
    *v = p.pila[p.pos];
}  /* TopPila */
```

9.6.5.4. Operazioni: `pop`

Estrazione dalla pila:

```
void Pop(TipoPila *p, TipoElemPila *v)
/* elimina l'elemento affiorante della pila p, restituendone
   il valore in v */
{
  if (TestPilaVuota(*p))
    printf("ERRORE: PILA VUOTA\n");
  else {
```

```
    *v = p->pila[p->pos];
    p->pos--;
  }
}  /* Pop */
```

9.6.5.5. Operazioni: `TestPilaPiena`

Verifica di pila piena (ausiliaria):

```
bool TestPilaPiena(TipoPila p){
/* restituisce TRUE se la pila p e' piena, FALSE altrimenti */
  return (p.pos == (MaxPila - 1));
}  /* TestPilaPiena */
```

9.6.5.6. Operazioni: `push`

Inserimento nella pila:

```
void Push(TipoPila *p, TipoElemPila v) {
/* inserisce l'elemento v in cima alla pila p */
  if (TestPilaPiena(*p))
    printf("ERRORE: PILA PIENA\n");
  else {
    p->pos++;
    p->pila[p->pos] = v;
  }
}  /* Push */
```

9.6.6. Rappresentazione sequenziale tramite vettori dinamici

Per utilizzare un vettore senza dover fissare la dimensione massima della pila a tempo di compilazione si può utilizzare un *vettore allocato dinamicamente*:

- si fissa una dimensione iniziale del vettore (che determina anche quella minima)
- la dimensione può crescere e decrescere a seconda delle necessità

```
#define DimInizialePila 10
typedef ... TipoElemPila;
struct tipoPila { TipoElemPila *pila;
                  int pos;
                  int dimCorrente; };
typedef struct tipoPila TipoPila;
```

9.6.6.1. Implementazione delle operazioni

- inizializzazione:
  effettua l'allocazione dinamica della memoria per il vettore

- verifica di pila vuota e analisi dell'elemento affiorante:
  Come per la rappresentazione sequenziale statica.

- inserimento di un elemento:
  Se la pila è piena se ne raddoppia la dimensione (`realloc`).

- eliminazione dell'elemento affiorante:
  Se il numero di elementi nella pila è meno di un terzo della dimensione massima (e più della dimensione iniziale), allora si alloca una nuova zona di memoria di metà dimensione (sempre con `realloc`).

9.6.6.2. Implementazione: Inizializzazione della pila

```
void InitPila(TipoPila *p)
/* inizializza la pila p, allocando la memoria per il vettore p->pila e
    ponendo a -1 il puntatore all'elemento affiorante della pila */
{
  p->pila = malloc(DimInizialePila * sizeof(TipoElemPila));
  p->pos = -1;
  p->dimCorrente = DimInizialePila;
}  /* InitPila */
```

9.6.6.3. Implementazione: Inserimento nella pila

```
void Push(TipoPila *p, TipoElemPila v){
/* inserisce l'elemento v in cima alla pila p e, se la pila e' piena,
   aumenta la dimensione della pila */
  if (p->pos == (p->dimCorrente - 1))
    { /* pila piena: raddoppia la dimensione del vettore p->pila */
      p->dimCorrente = p->dimCorrente * 2;
      p->pila = realloc(p->pila, p->dimCorrente * sizeof(TipoElemPila));
    }
  /* inserisce l'elemento v in cima alla pila p */
  p->pos++;
  p->pila[p->pos] = v;
}  /* Push */
```

9.6.6.4. Implementazione: Estrazione dalla pila

```
void Pop(TipoPila *p, TipoElemPila *v){
/* elimina l'elemento affiorante della pila p, restituendone
   il valore in v, e se necessario riduce la dimensione della pila */
  if (TestPilaVuota(*p))
    printf("ERRORE: PILA VUOTA\n");
  else {
   *v = p->pila[p->pos];
   p->pos--;
    /* se la pila contiene un numero di elementi maggiore di
       DimInizialePila e inferiore ad un terzo della sua dimensione
       corrente, viene dimezzata la sua dimensione */
   if (p->dimCorrente > DimInizialePila && p->pos < p->dimCorrente/3) {
      p->dimCorrente=p->dimCorrente/2;
      p->pila = realloc(p->pila, p->dimCorrente * sizeof(TipoElemPila));
     }
  }
}  /* Pop */
```

9.6.7. Rappresentazione collegata delle pile

Una pila è rappresentata mediante una dcl nella quale il primo elemento è l'elemento affiorante della pila.

Per rappresentare la pila vuota: `NULL`

9.6.7.1. Rappresentazione collegata delle pile: dichiarazioni

```
/* definizioni preliminari */
typedef ..... TipoElemPila;
typedef TipoElemPila TipoInfoSCL;


/* definizione del tipo pila */
typedef TipoSCL TipoPila;
```

9.6.7.2. Le operazioni

- `TestPilaVuota` diventa `emptySCL`
- `InitPila` diventa `InitSCL`
- `top` diventa `primoSCL`

- `pop` diventa (quasi) `restoSCL`
- `push` diventa `addSCL`

9.6.7.3. Rappresentazione sequenziale statica

+ implementazione semplice
+ può essere usata anche con linguaggi privi di allocazione dinamica della memoria (ad ed. FORTRAN)
- è necessario fissare a priori il numero massimo di elementi nella pila
- l'occupazione di memoria è sempre pari alla dimensione massima

9.6.7.4. Rappresentazione sequenziale dinamica

+ pila può crescere indefinitamente
- le operazioni di *Push* e *Pop* possono richiedere la copia di tutti gli elementi presenti nella pila non sono più a tempo di esecuzione costante

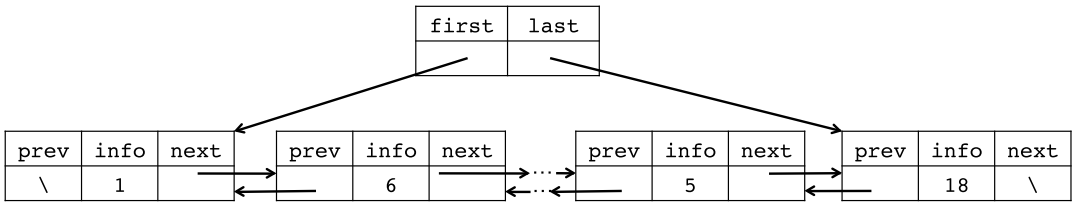
9.6.7.5. Rappresentazione collegata

+ pila può crescere indefinitamente
+ occupazione di memoria è proporzionale al numero di elementi nella pila
- maggiore occupazione di memoria dovuta ai puntatori
- implementazione leggermente più complicata

9.7. Strutture non Lineari

- Generalizzano le strutture lineari (es. mappe, matrici)
- I nodi hanno più riferimenti ad altri nodi
- Eccetto rari casi (es. matrici) rappresentazione collegata necessaria

9.7.1. Esempio: Lista Collegata Doppia



In una *lista collegata doppia* ciascun nodo ha un riferimento sia al successore che al predecessore.

+ Inserimento e cancellazione non richiedono la memorizzazione del riferimento al predecessore/successore
+ La lista può essere visitata in entrambe le direzioni
- Inserimento e cancellazione richiedono l'aggiornamento di un numero maggiore di collegamenti

9.7.1.1. Struttura Dati

```
typedef char TipoInfo;

struct ElemListaDoppia{
        TipoInfo info;
        struct ElemListaDoppia* next;
        struct ElemListaDoppia* prev;
};


typedef struct ElemListaDoppia TipoNodoListaDoppia;

struct RecordListaDoppia{
        TipoNodoListaDoppia* first;
        TipoNodoListaDoppia* last;
};

typedef struct RecordListaDoppia TipoListaDoppia;
```

9.7.1.2. Operazioni Principali

```
/* modifica la lista aggiungendo l'elemento e in prima posizione*/
void addTestaLista(TipoListaDoppia * lis, TipoInfo e);
```

```
/* modifica la lista aggiungendo l'elemento e in ultima posizione */
void addCodaLista(TipoListaDoppia * lis, TipoInfo e);

/* modifica la lista eliminando l'elemento in prima posizione*/
void delTestaLista(TipoListaDoppia *lis);

/* modifica la lista eliminando l'elemento in ultima posizione*/
void delCodaLista(TipoListaDoppia *lis);

/* modifica la lista aggiungendo l'elemento e in posizione p*/
void addPosLista(TipoListaDoppia * lis, TipoInfo e, int p);

/* modifica la lista eliminando l'elemento in posizione p*/
void delPosLista(TipoListaDoppia * lis, int p);
```

9.7.1.3. Implementazione (ricorsiva) delle operazioni principali

```
void addTestaLista(TipoListaDoppia *lis, TipoInfo e){
        if (lis == NULL){
                printf ("ERRORE: lis non e' una lista valida");
                exit(1);
        }

        TipoNodoListaDoppia* aux = (TipoNodoListaDoppia*)
                                malloc(sizeof(TipoNodoListaDoppia));
        aux -> info = e;
        if (emptyLista(*lis)){
                aux -> prev = NULL;
                aux -> next = NULL;
                lis -> first = aux;
                lis -> last = aux;
                return;
        }
        aux -> prev = NULL;
        aux -> next = lis -> first;
        lis -> first -> prev = aux;
        lis -> first = aux;
}
```

NOTA: addCodaLista sostanzialmente analoga

```
void delCodaLista(TipoListaDoppia *lis){
        if (lis == NULL){
                printf ("ERRORE: lis non e' una lista valida");
                exit(1);
        }
        if (emptyLista(*lis)){//Nessuna modifica
                return;
        }
        if (lis -> first == lis -> last){//un solo elemento
                free(lis -> first);
                lis -> first = NULL;
                lis -> last = NULL;
        }
        lis -> last = lis -> last -> prev;
        free(lis -> last -> next);
        lis -> last -> next = NULL;
}
```

NOTA: delTestaLista sostanzialmente analoga

```
/* modifica la lista aggiungendo l'elemento e in posizione p*/
void addPosLista(TipoListaDoppia* lis, TipoInfo e, int p){
        if (lis == NULL){
                printf ("ERRORE: lis non e' una lista valida");
                exit(1);
        }
        if (p == 0){
                addTestaLista(lis,e);
                return;
        }
        if (emptyLista(*lis)){
                //p!=0, quindi posizione non valida: nessuna modifica
                return;
        }
        int i = 0;
        TipoNodoListaDoppia* aux = lis->first;
        while (i < p && aux != NULL){
                i++;
```

```
                aux = aux -> next;
        }
        if (p != i){// Posizione non valida: nessuna modifica
                return;
        }
        if (aux == NULL) {
                addCodaLista(lis,e);
                return;
        }
        aux -> prev -> next = (TipoNodoListaDoppia*)
                              malloc(sizeof(TipoNodoListaDoppia));
        aux -> prev -> next -> info = e;
        aux -> prev -> next -> prev = aux -> prev;
        aux -> prev = aux -> prev -> next;
        aux -> prev -> next = aux;
}

/* modifica la lista eliminando l'elemento in posizione p*/
void delPosLista(TipoListaDoppia * lis, int p){
        if (lis == NULL){
                printf ("ERRORE: lis non e' una lista valida");
                exit(1);
        }
        if (emptyLista(*lis)){//lista vuota: nessuna modifica
                return;
        }
        if (p == 0){
                delTestaLista(lis);
                return;
        }
        int i = 0;
        TipoNodoListaDoppia* aux = lis->first;
        while (i < p && aux  != lis -> last){
                i++;
                aux = aux -> next;
        }
        if (p != i){// Posizione non valida: nessuna modifica
                return;
        }
```

```
        if (aux == lis -> last) {
                delCodaLista(lis);
                return;
        }
        aux -> prev -> next = aux -> next;
        aux -> next -> prev = aux -> prev;
        free(aux);
}
```

9.7.2. Esempio: Matrice con Rappresentazione Collegata

9.7.2.1. Struttura Dati

```
typedef int TipoInfo;

struct ElemMatriceSCL{
        TipoInfo info;
        struct ElemMatriceSCL* next_riga;
        struct ElemMatriceSCL* next_colonna;
};

typedef struct ElemMatriceSCL TipoNodoMatriceSCL;

typedef TipoNodoMatriceSCL* MatriceSCL;
```

9.7.2.2. Operazioni

```
/* Assegna una matrice di r righe e c colonne a *m */
void crea(MatriceSCL* m, unsigned int r, unsigned int c);

/* Assegna a *r il numero di righe di m */
void nRighe(MatriceSCL m, unsigned int* r);

/* Assegna a *c il numero di colonne di m */
void nColonne(MatriceSCL m, unsigned int* c);

/* Assegna v all'elemento di *m in riga r e colonna c */
void setVal(MatriceSCL m, unsigned int r, unsigned int c, TipoInfo v);

/* Assegna a *v il valore dell'elemento in riga r e colonna c di m */
void getVal(MatriceSCL m, unsigned int r, unsigned int c, TipoInfo* v);
```

```
/* Inserisce una nuova riga in posizione r in *m
(assume matrice non vuota) */
void addRiga(MatriceSCL* m, unsigned int r);

/* Rimuove la riga r da *m */
void delRiga(MatriceSCL* m, unsigned int r);

/* Inserisce una nuova colonna in posizione c in *m
(assume matrice non vuota)*/
void addColonna(MatriceSCL* m, unsigned int c);

/* Rimuove la colonna c da *m */
void delColonna(MatriceSCL* m, unsigned int c);

/* Dealloca la matrice m */
void destroy(MatriceSCL m);
```

9.7.2.3. Implementazione (ricorsiva) delle operazioni

```
/* Assegna una matrice di r righe e c colonne a *m */
void crea(MatriceSCL* m, unsigned int r, unsigned int c){
        if (r == 0 || c == 0){// La matrice e' vuota
                *m = NULL;
                return;
        }
        *m = (TipoNodoMatriceSCL*) malloc(sizeof(TipoNodoMatriceSCL));
        crea(&((*m)->next_colonna),1,c-1);
        crea(&((*m)->next_riga),r-1,c);
}
```

```
/* Assegna a *r il numero di righe di m */
void nRighe(MatriceSCL m, unsigned int* r){
        if (m == NULL){
                *r = 0;
                return;
        }
        unsigned int r1;
        nRighe(m -> next_riga, &r1);
        *r = 1 + r1;
}
```

NOTA: `nColonne` sostanzialmente analoga

```
/* Assegna v all'elemento di *m in riga r e colonna c */
```

```
void setVal(MatriceSCL m, unsigned int r, unsigned int c, TipoInfo v){
        if (m == NULL){// Posizione non valida
                exit(1);
        }
        if (r==0 && c==0){
                m -> info = v;
                return;
        }
        if (r > 0) {
                setVal(m->next_riga,r-1,c,v);
                return;
        }
        setVal(m->next_colonna,r,c-1,v);
}
```

NOTA: getVal sostanzialmente analoga

```
/* Inserisce una nuova riga in posizione r in *m
(assume matrice non vuota)*/
void addRiga(MatriceSCL* m, unsigned int r){
        if (m==NULL || (*m == NULL && r != 0)){//Matrice o pos non valic
                exit(1);
        }
        MatriceSCL new;
        unsigned int c;
        nColonne(*m,&c);
        crea(&new,1,c);
        if (r == 0){
                new -> next_riga = *m;
                *m = new;
        }
        else{
                MatriceSCL aux = *m;
                while (r > 1) {
                        r--;
                        aux = aux -> next_riga;
                }
                new -> next_riga = aux->next_riga;
                aux -> next_riga = new;
        }
```

```
}

/* Rimuove la riga r da *m */
void delRiga(MatriceSCL* m, unsigned int r){
        if (m==NULL || (*m == NULL)){//Matrice o pos non valida
                exit(1);
        }
        if (r == 0){
                MatriceSCL aux = *m;
                *m = (*m)->next_riga;
                destroy(aux->next_colonna);
                free(aux);
                return;
        }
        delRiga(&((*m)->next_riga),r-1);
}

/* Inserisce una nuova colonna in posizione c in *m
(assume matrice non vuota)*/
void addColonna(MatriceSCL* m, unsigned int c){
        if (m==NULL || (*m == NULL && c != 0)){//Matrice o pos non valida
                exit(1);
        }
        if (c == 0){
                TipoNodoMatriceSCL* new = (TipoNodoMatriceSCL*)
                                malloc(sizeof(TipoNodoMatriceSCL));
                if (*m!= NULL) {
                        new -> next_colonna = (*m);
                        new -> next_riga = (*m) -> next_riga;
                        (*m) -> next_riga = NULL;
                }
                else{
                        new -> next_colonna = NULL;
                        new -> next_riga = NULL;
                }
                if (new -> next_riga != NULL){
                        addColonna(&(new->next_riga),c);
                }
                *m = new;
                return;
```

```
        }
        addColonna(&((*m) -> next_colonna),c-1);
        if ((*m) -> next_riga != NULL) {
                addColonna(&((*m) -> next_riga),c);
        }
}

/* Rimuove la colonna c da *m */
void delColonna(MatriceSCL* m, unsigned int c){
        if (m==NULL || (*m == NULL)){//Matrice o pos non valida
                exit(1);
        }
        if (c == 0){
                TipoNodoMatriceSCL* aux = *m;
                *m = aux -> next_colonna;
                if (aux -> next_riga != NULL){
                        (*m) -> next_riga = aux -> next_riga;
                        delColonna(&((*m)->next_riga),c);
                }
                free(aux);
                return;
        }
        delColonna(&((*m)->next_colonna),c-1);
        if ((*m) -> next_riga != NULL) {
                delColonna(&((*m) -> next_riga),c);
        }
}

/* Dealloca la matrice m */
void destroy(MatriceSCL m){
        if (m == NULL) {
                return;
        }
        destroy(m -> next_colonna);
        destroy(m -> next_riga);
        free(m);
}
```

9.8. Liste ordinate

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

```
typedef char TipoInfo;

struct ElemLista{
        TipoInfo info;
        struct ElemLista* next;
};

typedef struct ElemLista TipoNodoLista;

typedef TipoNodoLista* TipoLista;
```

9.8.1. Funzioni

```
/* restituisce true se lis e' la lista vuota
 (analoga a lista semplice) */
int emptyLista(TipoLista lis);

/* restituisce il primo elemento di una lista non vuota
 (analoga a lista semplice) */
void primoLista(TipoLista lis, TipoInfo * e);

/* restituisce la lista privata del primo elemento
 (analoga a lista semplice) */
void restoLista(TipoLista * lis);

/* assegna al parametro la lista vuota
(analoga a lista semplice) */
void initLista(TipoLista * lis);

/* modifica la lista aggiungendo l'elemento e in posizione ordinata*/
void addLista(TipoLista * lis, TipoInfo e);

/* modifica la lista eliminando l'elemento e*/
void addLista(TipoLista * lis, TipoInfo e);

/* verifica se l'elemento e e' presente nella lista l*/
int presente(TipoLista l, TipoInfo e);
```

```
/* Inserisce (in maniera ordinata) gli elementi di l2 in l1 */
void merge(TipoLista* l1, TipoLista l2);

void stampa(TipoLista lis);
```

9.8.2. Aggiungi in posizione ordinata

```
void addLista(TipoLista * lis, TipoInfo e){
  if (*lis == NULL){
     TipoNodoLista* new = (TipoNodoLista*)
                          malloc(sizeof(TipoNodoLista));
     new -> info = e;
     new -> next = NULL;
     *lis = new;
     return;
  }
  else if (e < (*lis) -> info) {
     TipoNodoLista* new = (TipoNodoLista*)
                          malloc(sizeof(TipoNodoLista));
     new -> info = e;
     new -> next = *lis;
     *lis = new;
     return;
  }
  else addLista(&((*lis) -> next),e);
}
```

9.8.3. Merge

```
void merge(TipoLista* l1, TipoLista l2){
        if (emptyLista(l2)){
                return;
        }
        addLista(l1,l2-> info);
        merge(l1,l2->next);
}
```

9.8.4. Merge funzionale

```
TipoLista merge1(TipoLista l1, TipoLista l2){
        if (emptyLista(l1)&&emptyLista(l2)){
```

```
                return ris;
        }
        else if (emptyLista(l1))
                 return cons(first(l2),merge1(l1,resto(l2)));
        else if (emptyLista(l2))
                 return cons(first(l1),merge1(resto(l1),l2));
        else if (primo(l1) < primo(l2))
                 return cons(first(l1),merge1(resto(l1),l2));
        else return cons(first(l2),merge1(l1,resto(l2)));
}
```

9.8.5. Merge con modifica entrambi argomenti

```
// assume l1 non vuota
void merge2(TipoLista* l1, TipoLista * l2){
        if (emptyLista(*l2)) return;
        else if (primo(*l1) < primo(*l2))
                merge2(&(*l1->next),l2);
        else {
                TipoLista paux = *l2;
                *l2 = *l2->next;
                paux->next = *l1;
                *l1 = paux->next;
            merge2(l1,l2);
        }
}
```