# 8. Strutture collegate lineari

Inizialmente saranno discusse le limitazioni degli array nella gestione dinamica della memoria e successivamente saranno presentate le *strutture collegate lineari* che offrono un meccanismo estremamente flessibile per la gestione dinamica della memoria.

## 8.1. Limitazioni degli array

L'uso di array per memorizzare collezioni di oggetti presenta alcune limitazioni nella gestione della memoria:

- la dimensione dell'array è stabilita al momento della sua creazione e può essere inefficiente modificarla successivamente;
- l'array utilizza uno spazio di memoria proporzionale alla sua dimensione, indipendentemente dal numero di elementi validi effettivamente memorizzati;
- per mantenere un ordinamento degli oggetti della collezione e inserire (o rimuovere) un valore in una posizione specifica dobbiamo fare uno spostamento per una buona parte degli elementi dell'array.

Le strutture collegate che introduciamo in questa unità sono definite quindi appositamente per consentire di allocare e deallocare memoria in maniera dinamica, a seconda delle esigenze del programma nella memorizzazione dei dati di interesse per l'applicazione.

## 8.2. Strutture collegate

Un meccanismo molto flessibile per la gestione dinamica della memoria è dato dalle *strutture collegate*, che vengono realizzate in maniera

tale da consentire facilmente la modifica della propria struttura, gli inserimenti e le cancellazioni in posizioni specifiche, ecc.

In questa unità vedremo le *strutture collegate lineari*, che consentono quindi di memorizzare collezioni di oggetti organizzate sotto forma di sequenze (cioè in cui ogni elemento ha un solo successore).

Successivamente vedremo invece gli alberi, che sono un esempio di strutture collegate non lineari.

### 8.2.1. La dichiarazione di Struttura Collegata Lineare SCL

```
typedef ... TipoInfoSCL;

struct ElemSCL {
        TipoInfoSCL info;
        struct ElemSCL * next;
};

typedef struct ElemSCL TipoNodoSCL;
typedef  TipoNodoSCL * TipoSCL;
```

### 8.2.2. Creazione e collegamento di nodi

Il seguente frammento di programma crea una struttura lineare di 3 nodi contenenti stringhe.

#### SCL vuota

```
TipoSCL scl = NULL;
```

#### Creazione di un nodo

```
        TipoSCL scl = (TipoNodoSCL*) malloc(sizeof(TipoNodoSCL));
        scl->info = e;
        scl->next = NULL;
```

#### Collegamento

```
        TipoSCL temp = (TipoNodoSCL*) malloc(sizeof(TipoNodoSCL));
        scl->next = temp;
```

### 8.2.3. Operazioni sulle strutture collegate lineari

Le operazioni più comuni sulle strutture collegate lineari sono:

- *inserimento* ed *eliminazione*
- *creazione*
- *verifica* che la struttura sia vuota,
- *accesso* ad un nodo,
- *ricerca* di un'informazione nella struttura,
- calcolo della *dimensione* della struttura,
- *modifica* di elementi
- *distruzione*
- *conversioni* da e verso array, stringhe, ecc.

### 8.2.4. Inserimento di un nodo in prima posizione

```
void addSCL(TipoSCL * scl, TipoInfoSCL e){
        TipoSCL temp = *scl;
        *scl = (TipoNodoSCL*) malloc(sizeof(TipoNodoSCL));
        (*scl)->info = e;
        (*scl)->next = temp;
}
```

### 8.2.5. Costruzione SCL con `addSCL`

```
TipoSCL scl1 = NULL;
addSCL(&scl1,3);
addSCL(&scl1,5);
addSCL(&scl1,4);
```

Si noti che le istruzioni descritte funzionano correttamente anche se la struttura collegata è inizialmente vuota nel qual caso viene generata una sclcon un solo elemento.

### 8.2.6. Eliminazione di un nodo in prima posizione

```
void delSCL(TipoSCL * scl) {
        TipoSCL temp = *scl;
        *scl = (*scl)->next;
```

```
        free(temp);
}
```

## 8.3. Ricorsione per le operazioni su SCL

L'implementazione ricorsiva delle operazioni sulle strutture collegate lineari risulta generalmente più semplice di quella iterativa. Ciò deriva dal modo in cui sono definite le strutture collegate. Una struttura collegata è:

- vuota;
- formata un elemento seguito da una struttura collegata.

### 8.3.1. Schema di ricorsione per SCL

```
if (scl vuota?)
  passo base

else
  elaborazione primo elemento della scl
  chiamata ricorsiva sul resto della scl
```

## 8.4. Lettura e scrittura di SCL

### 8.4.1. Lettura e scrittura dell'informazione

```
// la lettura e scrittura del tipoInfo
// in questo caso int
int readTipoInfo(TipoInfoSCL*d) {
        return scanf("%d",d);
}
void writeTipoInfo(TipoInfoSCL d) {
        printf("%d ",d);
}
int readTipoInfoF(FILE *i,TipoInfoSCL * d) {
        return fscanf(i,"%d",d);
}
void writeTipoInfoF(FILE *i,TipoInfoSCL d) {
        fprintf(i,"%d ",d);
}
```

### 8.4.2. Lettura di una struttura collegata

```
void readSCL(TipoSCL * scl) {
        TipoInfoSCL dat;
        if (readTipoInfo(&dat)==EOF) *scl = NULL;
        else {
                *scl = (TipoNodoSCL*) malloc(sizeof(TipoNodoSCL));
                (*scl)->info = dat;
                readSCL(&(*scl)->next);
                }
}
```

### 8.4.3. Variante lettura con addSCL

```
void readAddSCL(TipoSCL * scl) {
        TipoInfoSCL dat;
        if (readTipoInfo(&dat)==EOF) *scl = NULL;
        else {
                readAddSCL(scl);
                addSCL(scl,dat);
                }
}
```

### 8.4.4. Scrittura di una struttura collegata

```
void writeSCL(TipoSCL scl){
        if (emptySCL(scl)) printf("\n");
        else {
                printf("%d ",scl->info);
                writeSCL(scl->next);
        }
}
```

## 8.5. Altre operazioni su SCL

### 8.5.1. Copia

```
void copySCL(TipoSCL scl, TipoSCL * ris){
        if (emptySCL(scl)) *ris=NULL;
        else {
                *ris = (TipoNodoSCL*) malloc(sizeof(TipoNodoSCL));
                (*ris)->info = scl->info;
```

```
                copySCL(scl->next,&((*ris)->next));
        }
}
```

### 8.5.2. Verifica

```
// restituisce true se scl e' NULL
int emptySCL(TipoSCL scl) {
        return scl == NULL;
}

// restituisce true se trova e in scl
int isinSCL(TipoSCL scl, TipoInfoSCL e) {
        if (emptySCL(scl)) return 0;
        else if (scl->info==e) return 1;
        else return isinSCL(scl->next,e);
}
```

### 8.5.3. Lunghezza

```
// restituisce la lunghezza della scl
int lengthSCL(TipoSCL scl) {
        if (emptySCL(scl)) return 0;
        else return 1 + lengthSCL(scl->next);
}
```

### 8.5.4. Ricerca

```
// restituisce il puntatore al nodo che contiene e,
// se trova e in scl, altrimenti NULL
void findSCL(TipoSCL scl, TipoInfoSCL e, TipoSCL * ris) {
        if (emptySCL(scl)) *ris = NULL;
        else if (scl->info==e) *ris = scl;
        else {
                findSCL(scl->next, e, ris);
        }
        return;
}
```

## 8.6. Ciclo di scansione di una SCL

Per effettuare operazioni su tutti gli elementi di una struttura collegata lineare, oppure su uno specifico elemento caratterizzato da una proprietà, è necessario effettuare cicli di scansione per accedere a tutti gli elementi.

Lo schema di ciclo per accedere a tutti i nodi della struttura il cui primo nodo è puntato dalla variabile a è il seguente.

```
TipoSCL a = ...
TipoSCL p = a;
while (p!=null)
   elaborazione del nodo puntato da p
   p = p->next;
```

*Nota*: in generale, il confronto tra le informazioni contenute nei nodi viene effettuato con l'operatore == per i tipi di dati primitivi, ma occorre definire funzioni specifiche per i tipi strutturati dall'utente.

### 8.6.1. Verifica (iterativa)

```
// restituisce true se trova e in scl
int isinSCL(TipoSCL scl, TipoInfoSCL e) {
  int trovato = 0;
  while (!emptySCL(scl) && !trovato) {
    if (scl->info==e) trovato = 1; /* forza l'uscita dal ciclo */
    else scl=scl->next; /* aggiorna scl al resto della lista */
    }
  return trovato;
}
```

### 8.6.2. Creazione iterativa di una SCL

Si usa usa la tecnica del *nodo generatore*.

Il nodo generatore è un nodo ausiliario che viene anteposto alla lista che vogliamo costruire e da cui partono le operazioni di creazione dei nodi successivi. Al termine della creazione della lista tale nodo viene rimosso e viene restituita la lista a partire dal nodo successivo. Tale soluzione è usata per trattare uniformemente tutti gli elementi della

lista, compreso il primo. Si osservi infatti che il metodo è corretto anche nel caso in cui la lista su cui viene invocato sia la lista vuota.

```
void copySCL(TipoSCL scl, TipoSCL *copia){
  TipoSCL prec;   /* puntatore all'elemento precedente */
   /* creazione del nodo generatore */
   prec = (TipoNodoSCL*) malloc(sizeof(TipoNodoSCL));
   /* scansione e copia della lista */
  *copia = prec;
  while (scl != NULL) { /* copia dell'elemento corrente */
    prec->next = (TipoNodoSCL*) malloc(sizeof(TipoNodoSCL));
    prec = prec->next;
    prec->info = scl->info;
    scl = scl->next;
  }
  prec->next = NULL; /* chiusura della scl */
  /* eliminazione del nodo generatore */
  prec = *copia;
  *copia = (*copia)->next;
  free(prec);
}
```

## 8.7. Operazioni basate sulla posizione

Supponiamo di numerare gli elementi du una SCL a partire da 1.

- ricerca dell'elemento in posizione n
  `void findSCL(TipoSCL scl, int n, TipoSCL * ris)`
- inserimento di un elemento in posizione n
  `void addPosSCL(TipoSCL * scl, TipoInfoSCL e, int n);`
- eliminazione di un elemento in posizione n `void delPosSCL(TipoSCL * scl, int n);`

### 8.7.1. Ricerca di un elemento tramite posizione

```
void findPosSCL(TipoSCL scl, int n, TipoSCL * ris){
        if (n==1) *ris = scl;
        else findSCL(scl->next, n-1, ris);
}
```

### 8.7.2. Inserimento in posizione data

```
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);
}
```

### 8.7.3. Eliminazione di un elemento in posizione data

```
void delPosSCL(TipoSCL * scl, int n){
        if (n == 1) {
                TipoSCL temp = *scl;
                (*scl) = (*scl)->next;
                free(temp);
        }
        else delPosSCL(&(*scl)->next, n-1);
}
```

### 8.7.4. Controllo del numero di elementi

```
void delPosSCL(TipoSCL * scl, int n){
        if(lengthSCL(scl) >=n delPosSCL1(scl, n);
        else printf("posizione inesistente");
}
```

Nota: cambia nome la funzione ricorsiva ausiliaria.

### 8.7.5. Memorizzare la lunghezza della SCL

La lunghezza della struttura si può memorizzare con il record:

```
struct LSCL {
        int lunghezza;
        struct TipoNodoSCL *scl;
};
typedef  LSCL * TipoLSCL;
```

## 8.8. Lettura da file

```
void readFileSCL(char *nomefile, int * n, TipoSCL * scl){
        FILE *datafile;
        datafile = fopen(nomefile, "r");
        (*n) =0;
        if (datafile == NULL) {  /* errore in apertura in lettura */
        printf("Errore apertura file ‘%s’ in lettura\n", nomefile);
                scl = NULL;
        }
        else
                readFileRSCL(datafile, n, scl);
    fclose(datafile);
}
```

### 8.8.1. Lettura da file ausiliaria

```
void readFileRSCL(FILE * i, int * n, TipoSCL * scl){
        TipoInfoSCL dat;
        if (readTipoInfoF(i,&dat)==EOF) *scl = NULL;
        else {
                (*n)++;
                *scl = (TipoNodoSCL*) malloc(sizeof(TipoNodoSCL));
                (*scl)->info = dat;
                readFileRSCL(i, n, &(*scl)->next);
                }
}
```

## 8.9. SCL con elementi ordinati

- ricerca dell'elemento
  `void findOSCL(TipoSCL scl, TipoInfoSCL e, TipoSCL * ris)`

- inserimento di un elemento in posizione ordinata
  `void addOpSCL(TipoSCL * scl, TipoInfoSCL e);`

- eliminazione di un elemento seguente eT
  `void delOdSCL(TipoSCL * scl, TipoInfoSCL e, TipoInfoSCL eT);`

Varianti inserimento prima di un elemento, dopo un elemento, con l'elemento indicato tramite valore, o tramite puntatore

### 8.9.1. Inserimento in ultima posizione

- inserimento di un elemento in ultima posizione
  `void addLastSCL(TipoSCL * scl, TipoInfoSCL e)`
- eliminazione di un elemento in ultima posizione
  `void delLastSCL(TipoSCL * scl)`

### 8.9.2. Modifica dell'informazione in un nodo

Sia `p` (il riferimento a) un nodo qualsiasi della struttura collegata, vogliamo modificare la sua informazione con il valore della stringa `x`.

```
TipoNodoSCL * p = ... // p e' il riferimento al nodo
TipoInfoSCL x = ... // x e' il nuovo valore da memorizzare in p
p->info = x;
```

In questo caso non è necessario modificare la struttura collegata, ma solamente il contenuto dell'informazione del nodo in questione.

### 8.9.3. Esempio: sostituzione

```
// sostituisce l'elemento eT della SCL in input con l'elemento e

void substElemSCL(TipoSCL scl, TipoInfoSCL e, TipoInfoSCL eT){
        if (!emptySCL(scl)) {
                if(scl->info==e) scl->info = eT;
                else substElemSCL(scl->next,e,eT);
        }
}
```

### 8.9.4. Esempio: sostituzione

```
// multiple occorrenze

void substNElemSCL(TipoSCL scl, TipoInfoSCL e, TipoInfoSCL eT){
        if (!emptySCL(scl)) {
                if(scl->info==e) {
                        scl->info = eT;
                        substNElemSCL(scl->next,e,eT);
```

```
                }
                else substNElemSCL(scl->next,e,eT);
        }
}
```

## 8.10. SCL: Implementazione funzionale

```
// restituisce il puntatore al nodo che contiene e, se trova e in scl
// altrimenti NULL

TipoSCL findSCL(TipoSCL scl, TipoInfoSCL e);
```

invece di: `void findSCL(TipoSCL scl, TipoInfoSCL e, TipoSCL * scl);`

### 8.10.1. Funzioni primitive

```
// restituisce true se scl e' NULL
int emptySCL(TipoSCL scl);

// restituisce il primo di una scl non vuota
TipoInfoSCL primoSCL(TipoSCL scl);

// restituisce il resto di una scl non vuota
TipoSCL restoSCL(TipoSCL scl);

// aggiunge l'elemento e in prima posizione alla SCL in input
TipoSCL addSCL(TipoSCL scl, TipoInfoSCL e);
```

### 8.10.2. Primo

```
TipoInfoSCL primoSCL(TipoSCL scl) {
        if (!emptySCL(scl)) return scl->info;
        else {
                printf("primo di una lista vuota");
                return ErrInfoSCL;
        }
}
```

### 8.10.3. Resto

```
TipoSCL restoSCL(TipoSCL scl) {
```

```
        if (!emptySCL(scl)) return scl->next;
        else {
                printf("resto di una lista vuota");
                return NULL;
        }
}
```

### 8.10.4. Costruzione di una SCL

```
TipoSCL addSCL(TipoSCL scl, TipoInfoSCL e){
        TipoSCL temp = (TipoNodoSCL*) malloc(sizeof(TipoNodoSCL));
        temp->info = e;
        temp->next = scl;
        return temp;
}
```

### 8.10.5. Altri esempi

```
TipoSCL copySCL(TipoSCL scl);

TipoSCL readSCL();

TipoSCL addPosSCL(TipoSCL scl, TipoInfoSCL e, int n);

TipoSCL delPosSCL(TipoSCL scl, int n);
```

### 8.10.6. Copia di una SCL

```
TipoSCL copySCL(TipoSCL scl){
        if (emptySCL(scl)) return NULL;
        else return addSCL(copySCL(restoSCL(scl)),
                           primoSCL(scl));
}
```

### 8.10.7. Inserimento in posizione n

```
// aggiunge l'elemento e in posizione n > 0 alla SCL in input
// assume che la SCL contenga almeno n-1 elementi

TipoSCL addPosSCL(TipoSCL scl, TipoInfoSCL e, int n) {
        if (n==1) return addSCL(scl,e);
```

```
        else return addSCL(addPosSCL(restoSCL(scl),e,n-1),
                           primoSCL(scl));
}
```

### 8.10.8. Inserimento in posizione n: con copia

```
// aggiunge l'elemento e in posizione n > 0 alla SCL in input
// assume che la SCL contenga almeno n-1 elementi

TipoSCL addPosSCL(TipoSCL scl, TipoInfoSCL e, int n) {
        if (n==1) return addSCL(copySCL(scl),e);
        else return addSCL(addPosSCL(restoSCL(scl),e,n-1),
                           primoSCL(scl));
}
```

### 8.10.9. Eliminazione in posizione n

```
// restituisce una SCL senza l'elemento in posizione n
// assume che la SCL contenga almeno n elementi

TipoSCL delPosSCL(TipoSCL scl, int n) {
        if (n==1) return restoSCL(scl);
        else return addSCL(delPosSCL(restoSCL(scl),n-1),
                           primoSCL(scl));
}
```

### 8.10.10. Inversa funzionale

```
TipoSCL InvertiSCL(TipoSCL scl){
  if (emptySCL(scl)) return NULL;
  else return addLastSCL(invertiSCL(restoSCL(scl)),primoSCL(scl));
}
```

### 8.10.11. Inversa funzionale: rivista

```
TipoSCL InvertiSCL(TipoSCL scl){
        Inverti1SCL(scl, NULL);
}

TipoSCL InvertiSCL1(TipoSCL scl, TipoSCL ris){
  if (emptySCL(scl)) return ris;
  else return Inverti1SCL(restoSCL(scl),
```

```
                        addSCL(ris,primoSCL(scl)));
}
```

### 8.10.12. Inversa (side-effect)

```
void InvertiSCL(TipoSCL *scl){
  TipoSCL prec = NULL; /* elemento precedente */
  TipoSCL suc;      /* elemento successivo */

  while (*scl != NULL) {
    suc = *scl;
    *scl = (*scl)->next;
    suc->next = prec;
    prec = suc;
  }
  *scl = prec;
}
```

### 8.10.13. Distruzione (side-effect)

```
void EliminaSCL(TipoSCL *scl) {
  TipoSCL paux;
  while (*scl != NULL) {
    paux = *scl;
    *scl = (*scl)->next;
    free(paux);
  }
}
```