Strutture Dati

Lezione 9
Le liste

Oggi parleremo di …

- Le liste
  - lista singolarmente concatenata
    - rappresentazione
    - operazioni
  - lista doppiamente concatenata
    - rappresentazione
- Stack e code dinamicamente concatenate

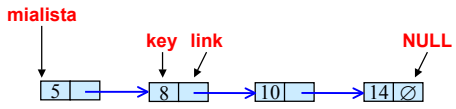
Liste concatenate

- Una **lista concatenata o puntata** è un insieme dinamico in cui ogni elemento ha una chiave ed un riferimento all'elemento successivo dell'insieme.
- È una struttura dati ad accesso strettamente sequenziale!
- Le rappresentazioni concatenate si ottengono mediante strutture che si *auto-referenziano*:

```
struct nodo {
        miotipo dato;
        altromiotipo altrodato;
        ...
        struct nodo *link;
        };
struct nodo *mialista;
```

Liste concatenate

- Ogni elemento della lista è composto da
  - un valore dell'elemento della lista, key (possono essere anche più valori);
  - un puntatore che identifica la locazione di memoria in cui è contenuto l'elemento successivo, link.
- L'elemento iniziale è comunque un puntatore.



Liste concatenate: le operazioni

- Le operazioni possibili per le liste sono le stesse degli array
  - creazione della lista
  - lettura di un elemento
  - inserimento di un elemento
  - eliminazione di un elemento

Creazione e lettura di una lista

- Si parla di creazione quando viene allocato il primo elemento.

```
if((mialista = malloc(sizeof(struct nodo)))== NULL)
  {
   fprintf(stderr, "malloc");
   exit(1);
  }
```

- Il recupero e lettura di un elemento di indice *k* non sono immediate a differenza di quanto visto per gli array ma implicano lo scorrimento della lista.

```
struct nodo *ii;
int counter = 0;

for(ii = mialista; counter != k;  ii = ii->link)   ++counter;
printf("Elemento %d: %s\n", counter, ii->miodato);
```

Inserimento e Cancellazione

- Nella fase di inserimento e cancellazione occorre considerare tre possibili casi:
  - inserimento o cancellazione dell'ultimo elemento;
  - inserimento o cancellazione del primo elemento;
  - inserimento o cancellazione in mezzo ad altri elementi.

Inserimento e Cancellazione

- **Inserimento come ultimo elemento** Occorre trovare la fine della lista.

```
struct nodo *ii;

for(ii = mialista;  ii->link;  ii = ii->link);
ii->link = indirizzoelementodaaggiungere;
ii->link->link = NULL;
```

`indirizzoelementodaaggiungere` potrebbe essere una `malloc` così come un puntatore a qualche elemento già allocato.

- **Cancellazione dell'ultimo elemento** Occorre trovare la fine della lista.

```
struct nodo *ii, *penultimo;

for(ii = mialista; ii->link;  ii = ii->link) penultimo = ii;
free(ii);
penultimo->link = NULL;
```

Inserimento e Cancellazione

- **Inserimento del primo elemento** E' estremamente scomodo quando utilizzo delle funzioni e la mia lista è una variabile locale. In questo caso occorre utilizzare puntatori a puntatori!

```
struct nodo *tmp;

tmp = mialista;
mialista = nuovoelemento;
nuovoelemento->link = tmp;
```

- **Cancellazione del primo elemento** In questo caso occorre utilizzare puntatori a puntatori!

```
struct nodo *tmp;

tmp = mialista;
mialista = mialista->link;
free(tmp);
```

Inserimento e Cancellazione

- **Inserimento in mezzo alla lista** Supponiamo di voler inserire un elemento puntato da `nuovodato` successivamente ad un elemento puntato da `precedente` nella mia lista. Il caso è del tutto analogo a quello dell'inserimento ad inizio lista.

```
nuovodato->link = precedente->link;
precedente->link = nuovodato;
```

- **Cancellazione in mezzo alla lista** Supponiamo di voler cancellare un elemento puntato da `delete`. Dovrò scorrere la lista fino al precedente.

```
struct nodo *ii;

for(ii = mialista;  ii->link! = delete;  ii = ii->link);
ii->link = delete->link;
free(delete);
```

Implementazione una lista

Creazione di una lista

```
typedef struct list_node *list_pointer;
typedef struct list_node {
          int key;
          list_pointer link;
        }lista;

list_pointer testa = NULL;
```

```
#define IS_FULL(ptr) (!(ptr))
#define IS_EMPTY(ptr) (!(ptr))
```

Visualizzazione di una lista

```
void visualizza(list_pointer ptr)
{
 if (IS_EMPTY(ptr)) { printf("\nLa lista e' vuota"); return; }

 printf("\nLa lista contiene: ");
 for(; ptr; ptr = ptr->link)
       printf("%4d", ptr->key);
 printf("\n");
}
```

Implementazione una lista

Inserimento di un elemento in una lista

```
void inserisci(list_pointer *testa, int item)
{
 list_pointer nodo, ptr;

 ptr = *testa;
 nodo = ptr;

 if(ptr && ptr->key > item) insert_testa(testa, item);
 else {
    for(; ptr && ((ptr->key)<item); ptr=ptr->link )  nodo = ptr;
    insert_dopo(testa, nodo, item);
    }
}
```

Implementazione una lista

Inserimento di un elemento in una lista

```
void insert_testa(list_pointer *ptr, int item)
{
 list_pointer temp;

   temp = (list_pointer)malloc(sizeof(list_node));
   if(IS_FULL(temp)) {
        fprintf(stderr, "La memoria e' piena");
        exit(1);
     }
  temp->key = item;
  temp->link = *ptr;
  *ptr = temp;
}
```

Implementazione una lista

Inserimento di un elemento in una lista

```
void insert_dopo(list_pointer *ptr, list_pointer nodo, int item)
{
   list_pointer temp;

   temp = (list_pointer)malloc(sizeof(list_node));
   if(IS_FULL(temp)) {
        fprintf(stderr, "La memoria e' piena");
        exit(1);
        }
   temp->key = item;
   if (*ptr) {
        temp->link = nodo->link;
        nodo->link = temp;
        }
   else {
        temp->link = NULL;
        *ptr = temp;
        }
}
```

Implementazione una lista

Cancellazione di un elemento in una lista

```
void cancella(list_pointer *testa, int item)
{
  list_pointer nodo, ptr;

  if (IS_EMPTY(*testa)) { printf("\nLa lista e' vuota"); return; }

  ptr = *testa;
  for(; ptr && ((ptr->key)!=item); ptr=ptr->link )
        nodo = ptr;
  if (ptr == NULL) printf("\nL'elemento non e' presente nella lista");
  else { if (ptr == *testa) nodo = NULL;
       cancella_dopo(testa, nodo, ptr);
      }
}
```

```
void cancella_dopo(list_pointer *testa, list_pointer nodo, list_pointer ptr)
{
 if(nodo)         nodo->link = ptr->link;
 else             *testa = (*testa)->link;
 free(ptr);
}
```

Implementazione una lista

- Le chiamate per la visualizzazione, l'inserimento e la cancellazione sono

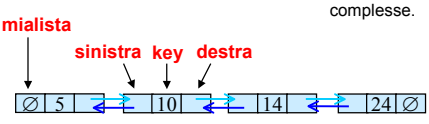
```
visualizza(testa);
inserisci(&testa, item);
cancella(&testa, item);
```

Liste doppiamente concatenate

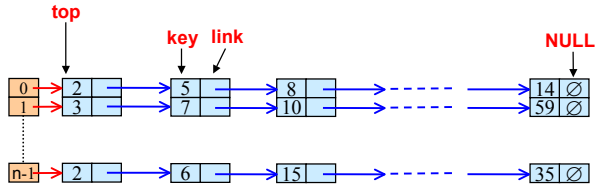
- In molti casi diventa comodo poter scorrere la lista in ambo i sensi. A tal fine si possono usare le **liste doppiamente concatenate** o **simmetriche**:

```
struct nodo {
        miotipo dato;
        altromiotipo altrodato;
        ...
        struct nodo *destra;
        struct nodo *sinistra;
        };
struct nodo *mialista;
```

Le operazioni di inserimento e cancellazione di elementi da una lista doppiamente concatenata sono più complesse.



Stack dinamicamente concatenati




```
#define MAX_STACKS 10  //numero max di stacks

typedef struct {
            int key;
            // altri campi
        } elemento;

typedef struct stack *stack_pointer;
typedef struct stack{
        elemento item;
        stack_pointer link;
        } pila;
stack_pointer top[MAX_STACKS];
```

Inserimento e Cancellazione

```
inserisci(&top[no_stack], item);

item = cancella(&top[no_stack]);
```

```
for (no_stack=0;
   no_stack<MAX_STACKS;no_stack++)
   top[no_stack] = NULL;
```

Stack dinamicamente concatenati

Inserimento

```
void inserisci(stack_pointer *top, elemento item)
{
 // aggiunge un elemento in cima allo stack

 stack_pointer temp = (stack_pointer)malloc(sizeof(stack));
 if(IS_FULL(temp)) {
                  fprintf(stderr, "La memoria e' piena");
                  exit(1); }
 temp->item = item;
 temp->link = *top;
 *top = temp;
}
```

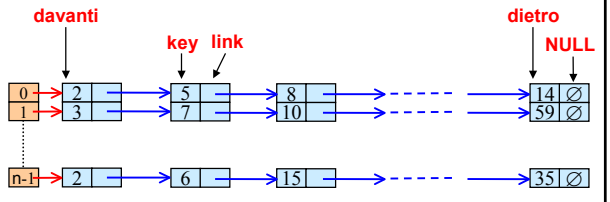
Cancellazione

```
elemento cancella(stack_pointer *top)
{
 //cancella un elemento in cima allo stack

 stack_pointer temp = *top;
 elemento item;

 if (IS_EMPTY(temp)) { printf("Lo stack e' vuoto");
                  exit(1); }
 item = temp->item;
 *top = temp->link;
 free(temp);
 return(item);
}
```

Code dinamicamente concatenate




```
#define MAX_CODE 10        //numero max di code

typedef struct {
        int key;
        // altri campi
        } elemento;

typedef struct coda *coda_pointer;
typedef struct coda{
          elemento item;
          coda_pointer link;
        } queue;
coda_pointer davanti[MAX_CODE],dietro[MAX_CODE];
```

Inserimento e Cancellazione

```
inserisci(&davanti[no_coda],
    &dietro[no_coda], item);

item=cancella(&davanti[no_coda]);
```

```
for (no_coda=0; no_coda<MAX_CODE;
    no_coda++)
    davanti[no_coda] = NULL;
```

Code dinamicamente concatenate

Inserimento

```
void inserisci(coda_pointer *davanti, coda_pointer *dietro,
                elemento item)
{
 coda_pointer temp = (coda_pointer)malloc(sizeof(coda));
 if(IS_FULL(temp)) {
        fprintf(stderr, "La memoria e' piena");
        exit(1); }
 temp->item = item;
 temp->link = NULL;
 if (*davanti)    (*dietro)->link = temp;
 else             *davanti = temp;
 *dietro = temp;
}
```

Cancellazione

```
elemento cancella(coda_pointer *davanti)
{
   coda_pointer temp = *davanti;
   elemento item;

   if(IS_EMPTY(temp))     { printf("\nLa coda e' vuota");
                          exit(1); }
   item = temp->item;
   *davanti = temp->link;
   free(temp);
   return(item);
}
```