ALGORITMI E STRUTTURE DATI

ESERCITAZIONI
ANDREA ORLANDINI
http://www.dia.uniroma3.it/~orlandin/asd/
e-mail: orlandin@dia.uniroma3.it

ORARIO DI RICEVIMENTO: Martedì 14.00 - 16.00

PUNTATORI E LISTE

STUDENTIDIA FORUM
http://forum.studentidia.org/

DOVE TROVARE I LUCIDI PRESENTATI A LEZIONE
http://limongelli.dia.uniroma3.it/asd/

Puntatori in C

- Una variabile puntatore è una variabile *speciale* che contiene un'indirizzo di memoria che rappresenta la posizione, all'interno della memoria, di un'altra variabile. Detta anche **Riferimento** o semplicemente **Puntatore**
- Perchè una variabile possa essere utilizzata come puntatore deve essere dichiarata come tale

```
tipo *nome_variabile;
```

- *tipo* indica a quale tipo di variabile il puntatore indirizza
- *nome_variabile* è il nome della variabile puntatore

Operatori su puntatori

- l'operatore & restituisce l'indirizzo di memoria della variabile a cui è applicato

```
int conta = 10;
int *indirizzo_conta;

indirizzo_conta = &conta;
```

- la precedente istruzione pone in *indirizzo_conta* l'indirizzo di memoria della variabile *conta*
- l'operatore * è il complemento di &: restituisce il valore della variabile indirizzata dalla variabile puntatore a cui è applicato

```
int val = *indirizzo_conta;
```

- la precedente istruzione pone in *val* il valore di *conta* definita sopra

- NULL puntatore nullo ('\0')

- gli operatori * e & sono identici agli operatori di moltiplicazione matematica e di AND su bit, ma hanno precedenza sugli operatori aritmetici

- E' importante specificare il tipo per definire le variabili e i puntatori

```
void main()
{
   float x,y;
   int *p;

   p=&x;
   y=*p;
}
```

- questo programma non assegna il valore di x a y: essendo p dichiarato come puntatore a un intero, solo due byte vengono trasferiti a y

Espressioni con puntatori

- i puntatori possono essere utilizzati come variabili normali

```
void main()
{
   int x;
   int *p1, *p2;

   p1=&x;
   p2=p1;
   printf(``%p'',p2); /* stampa in formato esadecimale
                         l'indirizzo di x */
}
```

Aritmetica dei puntatori

- il C consente l'uso di due operatori aritmetici sui puntatori: + e -
- sia p1 un puntatore di tipo intero il cui valore corrente è 2000

```
p1++; /* il nuovo valore di p1 è 2002 */

p1--; /* il nuovo valore di p1 è 1998 */

p1 = p1 + 9; /* il nuovo valore di p1 è 2018 */
```

Funzioni di allocazione dinamica

- per poter attingere alle risorse di memoria (heap) a tempo di esecuzione

- per controllare l'utilizzo della memoria libera del sistema

- esistono molte funzioni per l'allocazione ma due sono le più importanti:

- **malloc**: alloca porzioni di memoria ritornando l'indirizzo di memoria assegnato

- **free**: restituisce al sistema operativo porzioni di memoria

- i programmi che utilizzano le funzioni di allocazione dinamica devono includere il file STDLIB.H

Malloc

```
void *malloc(size_t numero_di_byte);
```

- Restituisce un puntatore di tipo *void* (da intendere come tipo indefinito)
- Indirizzo al primo byte della regione di memoria allocata

```
char *p;
p = malloc(1000); /* alloca 1000 byte */
```

- Se non c'è memoria disponibile il ritorno è un valore nullo

```
p=malloc(100);
if (!p){
   printf(``Memoria Finita\n'');
   exit(1);
}
```

- Utilizzo della **sizeof** per allocare memoria di un determinato tipo

```
p = malloc(50*sizeof(int));
```

Free

```
void free(void *p);
```

- Restituisce al sistema memoria precedentemente allocata liberando la zona di memoria puntata dal parametro

  ```
  free(s);
  ```

- Operazione pericolosa: se l'argomento è sbagliato potremmo causare la perdita di informazioni di sistema

- Operazione utile: rilascio le zone di memoria non più utilizzate per evitare fallimenti di allocazione

Problemi connessi all'utilizzo dei puntatori

- Difficoltà nella correzione degli errori
- Operazioni di lettura: lettura di informazioni non significative
- Operazioni di scrittura: è possibile alterare altre porzioni di codice
- Problemi di inizializzazione

```
void main(void) /* Questo programma è errato */
{
   int x, *p;
   x=10;
   *p=x;
}
```

- Incomprensione dell'uso dei puntatori

```
void main(void) /* Questo programma è errato */
{
   int x, *p;
   x=10;
   p=x;
   printf(``%d'',*p);
}
```

Problemi connessi all'uso dei puntatori

- Errore potenzialmente pericoloso

```
void main(void) {
   char *p1; char s[80];

   p1 = s;
   do {
      scanf(``%s'',s); /* legge una stringa */

      while (!p1)
       {
       printf(`` %d ``,*p1); /* stampa int corrispondente al char*/
       p1++;
       }

      printf(``\n'');
   } while (strcmp(s,''fatto''));
}
```

Puntatore di puntatore

- Un puntatore è a sua volta una variabile e quindi è possibile definire il puntatore di un puntatore
- Utilizzato per il passaggio per riferimento dei puntatori nelle funzioni

```
int **p;
```

Utilizzo puntatori per passaggio parametri

```
void modificaValoreNonPermanente(int valore_intero)
{
  valore_intero = valore_intero + 20;
}

void modificaValorePermanente(int *pvalore_intero)
{
  *pvalore_intero = *pvalore_intero + 20;
}

void main()
{
  int valore=3;
  printf ("valore = %d\n",valore);

  modificaValoreNonPermanente(valore);

  printf ("valore= %d\n",valore);

  modificaValorePermanente(&valore);

  printf ("valore = %d\n",valore);
}
```

Ma anche i puntatori sono variabili

```
void modificaPuntatorePermanente(int **puntatore1,int *puntatore2)
{
  *puntatore1 = puntatore2;
}

void main()
{
  int var1 = 10;
  int var2 = 20;

  int *puntatore1,*puntatore2;
  puntatore1 = &var1;
  puntatore2 = &var2;

  printf ("var1 = %d\n",*puntatore1);
  printf ("var2 = %d\n",*puntatore2);

  modificaPuntatorePermanente(&puntatore1,puntatore2);

  printf ("var1 = %d\n",*puntatore1);
  printf ("var2 = %d\n",*puntatore2);
}
```

Liste concatenate

Insiemi di elementi di dimensione massima non prefissata

L'inserimento e la cancellazione avvengono sempre "in testa" alla lista

Operazioni (non "distruttive"):

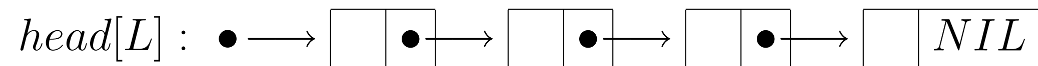
Empty()	riporta la lista vuota
Null(L)	test lista-vuota
Insert(L,x)	riporta la lista che si ottiene inserendo x in testa a L
Rest(L)	riporta la lista che si ottiene cancellando da L l'elemento x in testa alla lista
First(L)	riporta l'elemento in testa alla lista L, senza modificarla
Setlist(L,L')	modifica la lista L ponendola uguale a L'.

Un elemento x di una lista è un oggetto con almeno due campi (ed eventualmente altri campi per dati satellite):

$key[x]$	campo chiave
$next[x]$	puntatore al successore dell'oggetto nella lista

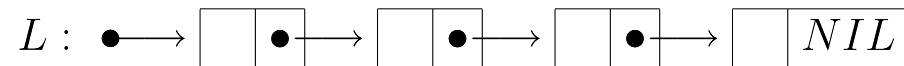
Una lista L è un oggetto con un campo:

$head[L]$ puntatore al primo elemento x della lista



Lista vuota: $head[L] = NIL$

Se il linguaggio fornisce puntatori (questo è il caso del linguaggio C), la lista è rappresentata semplicemente da un puntatore a un elemento:



Implementazione delle operazioni di base

```
EMPTY()
1. ;; eventuale allocazione della memoria per L
2. head[L] <- NIL
3. return L

NULL(L)
1. return (head[L] = NIL)

INSERT(L,x)
1. ;; eventuale allocazione della memoria per L'
2. next[x] <- head[L]
3. head[L'] <- x
4. return L'
```

```
REST(L)
1. if NULL(L)
2.     then error "empty list"
3.     else
4.          ;; eventuale allocazione della memoria
            ;; per L'
5.          head[L'] <- next[head[L]]
6.          return L'
```

```
FIRST(L)
1. if NULL(L)
2.     then error "empty list"
3.     else return head[L]
            ;; viene riportato un puntatore a un oggetto
```

Altre operazioni di base

Operazione di assegnazione per le liste:

```
SETLIST(L,L')
1. head[L] <- head[L']
```

Occorre, eventualmente, definire anche un'operazione per liberare la memoria da un oggetto lista.

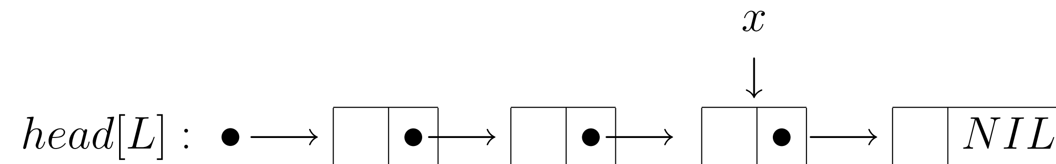
Astrazione sul tipo di dati elemento:

```
KEY(x)
1. return key[x]

NEXT(x)
1. return next[x]

SET-KEY(x,k)
1. key[x] <- k

SET-NEXT(x,y)
1. next[x] <- y

MK-ELEM()
1. ;; allocazione della memoria per x
2. return x
```

Occorre, eventualmente, definire anche un'operazione per liberare la memoria da un elemento.

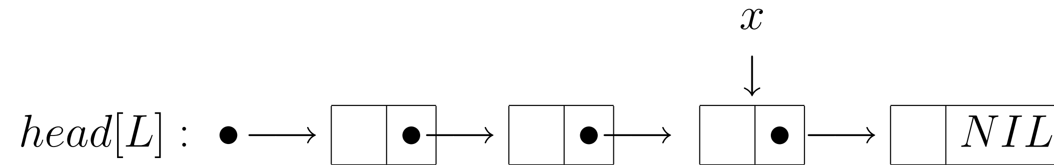
Ricerca di un elemento in una lista

```
SEARCH(L,k)
1. if NULL(L)
2.     then return NIL
3.     else x <- FIRST(L)
4.          while x != NIL and KEY(x) != k
5.             do x <- NEXT(x)
6.          return x
      ;; riporta un puntatore all'oggetto
      ;; NIL se la ricerca fallisce
```



Cancellazione di un elemento da una lista

x è un puntatore a un elemento:




```
DELETE(L,x)
1. if not NULL(L) ;; se la lista e' vuota non
                  ;; c'e' niente da cancellare
2.    then if FIRST(L) = x      ;; x e` il primo elemento
3.            then SETLIST(L,REST(L))
4.         else y <- FIRST(L)
                ;; ricerca del predecessore di x nella lista
5.              while NEXT(y) != NULL and NEXT(y) != x
6.                 do y <- NEXT(y)
                ;; ora se NEXT(y) != NULL, y e`
                ;; il predecessore di x
7.              if NEXT(y) != NULL
8.                 then SET-NEXT (y,NEXT(x))
```