INFORMATICA A

A.A. 2017-18

Laboratorio n°6

Ing. Gian Enrico Conti

Dott. Michele Zanella

HEAP LAB

Info Logistiche (cont'd)

- Calendario laboratori

Data	Orario	Squadra	Aula	Resp.	Programma
23/10/2017	9:00 - 12:00	A	CS 0.2	Zanella	Lab 1
23/10/2017	9:00 - 12:00	B	CS 1.4	Conti	
26/10/2017	14:15-17:15	C	CS 0.11	Conti	
30/10/2017	9:00 - 12:00	A	CS 0.2	Zanella	Lab 2
30/10/2017	9:00 - 12:00	B	CS 1.4	Conti	
02/11/2017	14:15-17:15	C	CS 0.11	Conti	
06/11/2017	9:00 - 12:00	A	CS 0.2	Zanella	Lab 3
06/11/2017	9:00 - 12:00	B	CS 1.4	Conti	
09/11/2017	14:15-17:15	C	CS 0.11	Conti	
20/11/2017	9:00 - 12:00	A	CS 0.2	Zanella	Lab 4
20/11/2017	9:00 - 12:00	B	CS 1.4	Conti	
23/11/2017	14:15-17:15	C	CS 0.11	Conti	
27/11/2017	9:00 - 12:00	A	CS 0.2	Zanella	Lab 5
27/11/2017	9:00 - 12:00	B	CS 1.4	Conti	
30/11/2017	14:15-17:15	C	CS 0.11	Conti	
04/12/2017	9:00 - 12:00	A	CS 0.2	Zanella	Lab 6
04/12/2017	9:00 - 12:00	B	CS 1.4	Conti	
14/12/2017	14:15-17:15	C	CS 0.11	Conti	

Recap dei laboratori

- LAB1: Compilazione e I/O di base (`printf, scanf, ...`)
- LAB2: Cicli (`for, while,...`) e Condizioni (`if, switch,...`)
- LAB3: Array, Matrici e Struct
- LAB4: Header files, Funzioni, Puntatori, File I/O (.txt, .csv,...)
- LAB5: Puntatori, File I/O, Ricorsione
- LAB6: Allocazione Dinamica e Liste Dinamiche

- Da LAB3 a LAB6: applicare le nozioni viste per migliorare il progetto sugli aeroporti

Allocazione dinamica

- La memoria è divisa sostanzialmente in due parti:
  - STACK: Parte *statica*, contiene le variabili locali delle funzioni, gli argomenti passati e viene allocata quando viene eseguita una chiamata a funzione
  - HEAP: Parte *dinamica*, contiene le variabili allocate durante l'esecuzione del programma ("runtime"), la cui dimensione non è nota a priori.

Allocazione dinamica

- Funzioni utili, bisogna includere *stdlib.h*:
  - `malloc()`: permette di allocare sullo heap blocchi di Byte in numero e di dimensione a priori (i.e., in fase di compilazione) non noto.
  - `free()`: libera la memoria allocata
  - `realloc()`: modifica uno spazio di memoria precedentemente allocato

Casting!

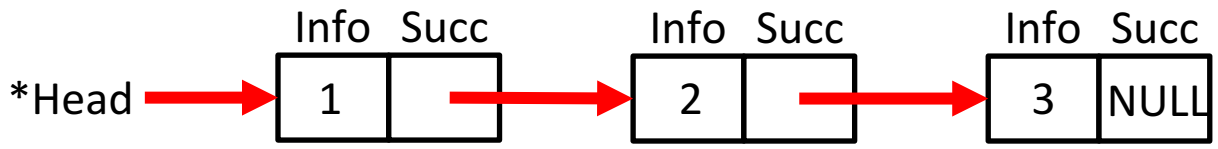
```
#include <stdlib.h>

int main() {
      int* num;

      num = (int*) malloc(sizeof(int));
}
```

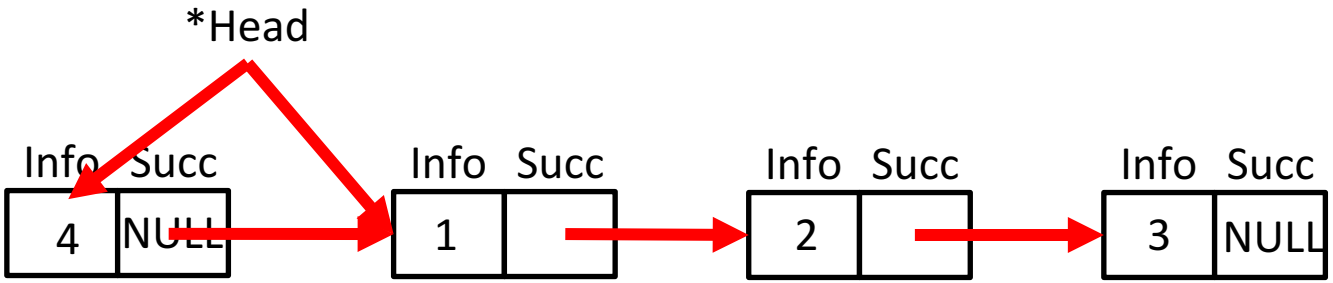
Liste

- Una **lista** è una collezione di elementi omogenei MA che occupano in memoria una posizione qualsiasi.
- La sua dimensione può cambiare dinamicamente durante l'esecuzione.
- Gli elementi della lista possono contenere vari campi per informazioni (come le *struct* normali), ma devono contenere un *puntatore all'elemento successivo* (in alcuni casi anche a quello precedente).



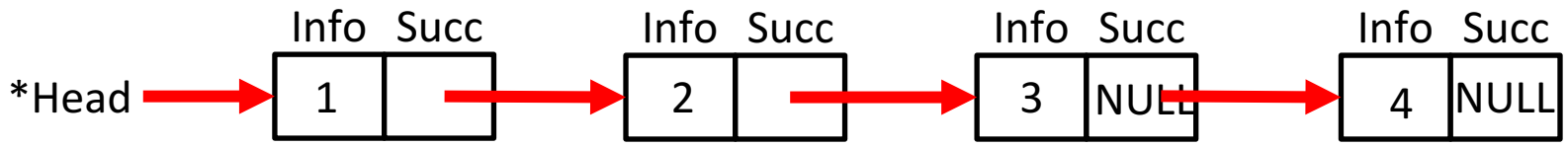
Liste: Aggiunta elementi

- Aggiungere un elemento in testa alla lista:
  1. Allocare la memoria per quell'elemento
  2. Sostituire NULL con l'indirizzo dell'attuale *head*
  3. Aggiornare la *head* con l'indirizzo dell'elemento creato al punto 1



Liste : Aggiunta elementi

- Aggiungere un elemento in fondo alla lista:
  1. Allocare la memoria per quell'elemento
  2. Percorrere la lista dalla corrente *head* fino a quando non si raggiunge il campo *succ* dell'ultimo elemento (che sarà NULL)
  3. Sostituire NULL con l'indirizzo dell'elemento creato al punto 1
  4. Assegnare NULL al campo *succ* di questo ultimo elemento



Esercizio 1: Array dinamico

Si scriva un programma che permetta di inserire dei numeri interi in un array di dimensione decisa dall'utente

Hint: Allocare dinamicamente la memoria dell'array una volta saputa la dimensione dall'utente

Esercizio 2: Runtime Array

Si scriva un programma che permetta di inserire dei numeri interi in un array di dimensione calcolata runtime.

In questo caso l'utente può continuare ad aggiungere numeri ad ogni iterazione.

Hint: Reallocare la memoria dinamicamente ad ogni iterazione.

Esercizio 3: Lista contatti

Si scriva un programma che permetta di inserire i contatti di una persona

Ogni contatto e' una struttura con:

- nick
- mail

Poiché il numero di contatti non e' noto, va usata una struttura dinamica (lista monodirezionale)

Il programma (dotato di menu minimale) deve permettere:

1) inserimento in testa
2) inserimento in coda
3) stampa di tutti i contatti dall'inizio della lista
4) cancellazione di tutti i contatti
5) ricerca di un contatto data una parte del nick
6) stampa di tutti i contatti dala fine della lista

Esercizio 3: Lista contatti (Cont'd)

Creiamo il tipo di dato elementare:

```
typedef struct Contact{

} Contact;
```

deve contenere nick e mail, ma anche il puntatore al prossimo elemento..

Attenzione…. serve:

```
#include <stdlib.h>      /* malloc, free, rand */
```

Esercizio 3: Lista contatti (Cont'd)

La nostra *struct* potrebbe essere:

```
#define MAX_LEN 256

typedef char String[MAX_LEN];

typedef struct Contact{
     String nick;
     String email;
     struct Contact * next;
} Contact;
```

Ora il menu, basta copiare dalla traccia…

Esercizio 3: Lista contatti (Cont'd)

Per essere piu ordinati facciamo una funzione `menu()` :

```
int menu(){
      printf("1) inserimento in testa\n");
      printf("2) inserimento in coda\n");
      printf("3) stampa di tutti i contatti\n");
      printf("4) cancellazione di tutti i contatti\n");
      printf("5) ricerca di un contatto data una parte del nick\n");
      printf("6) stampa di tutti i contatti in ordine di
inserimento\n");

      int selection;
      scanf("%d", &selection);
      return selection;
}
```

Ritorna un `int` cosi possiamo fare switch nel *main*.

Esercizio 3: Lista contatti (Cont'd)

Switch nel main:

```
while(1){
      int selection = menu();
      switch (selection){
            case 1:
                  InsertAtTheBeginning();
                  break;

            case 2:
                  InsertAtTheEnd();
                  break;

      }     // end of switch
}     // end of while
```

per gli altri: scrivete per prima cosa le f. vuote, **POI** le riempite.. cosi compila SEMPRE!

Esercizio 3: Lista contatti (Cont'd)

Ci serve creare la funzione `InsertAtTheBeginning()` e le altre funzioni

Abbiamo 2 strade:

A) tutta la funzione PRIMA del main
B) prototipo PRIMA del main e "corpo" dopo

sono equivalenti, scegliete quello che preferite in base alla maggior leggibilita' del programma

Esercizio 3: Lista contatti (Cont'd)

Se A) avremo:

```
void InsertAtTheBeginning()
{
     …
}

void InsertAtTheEnd()
{
     …
}

int main(int argc, const char *
argv[]) {

….
```

Esercizio 3: Lista contatti (Cont'd)

Le altre funzioni da implementare saranno (dati i prototipi):

- Inserimento alla fine

```
Contact * InsertAtTheEnd(Contact*head);
```

- Inserimento info contatto

```
void FillContact (Contact* c);
```

- Stampa contatti (e singolo contatto)

```
void Print (Contact* head);
```

```
void PrintContact(Contact* c);
```

- Cancella tutta la lista

```
void DeleteAll (Contact* head);
```

- Trova nickname

```
void FindNick (Contact* head);
```

- Stampa dalla fine (ricorsiva)

```
void PrintFromLast (Contact* head);
```

Esercizio 3: Lista contatti (Cont'd)

Ci serve una variabile che faccia riferimento ("punti") al primo elemento delle lista.

Per comodità puo' essere una variabile globale, ma meglio se locale al main, e le funzioni ritornano la lista aggiornata, Quindi:

```
int main(int argc, const char * argv[])
{

     Contact*head = NULL;
….
```

ed anche:

```
Contact *
InsertAtTheBeginning(Contact*head){
  Contact * newHead = …..
     …
    return newHead;
}
```

Esercizio 3: Lista contatti (Cont'd)

```
void InsertAtTheEnd(Contact*head)
```

Questa funzione non modifica la testa… quindi void ?

……

Esercizio 3: Lista contatti (Cont'd)

NO!

se lista NULL deve ritornare un valore!

```
Contact* InsertAtTheEnd(Contact*head)
```

Invece la stampa e' void.

e le altre?

……

Esercizio 3: Lista contatti (Cont'd)

```
Contact *
InsertAtTheBeginning(Contact*head)
{
      Contact* newHead =
malloc(sizeof(Contact));


   ….

   return newHead;
}
```

In ogni caso allochiamo… e ritorniamo sempre il nuovo blocco, sia che la lista esista o no.

Sara' diversa la logica.

PS. dovremmo controllare che la malloc non sia fallita..