Fondamenti di Informatica T-1, 2013/2014 – Modulo 2

Prova d'Esame 6A di Giovedì 11 Settembre 2014 – tempo a disposizione 2h

Avvertenze per la consegna: apporre all'inizio di ogni file sorgente un commento contenente i propri dati (**cognome, nome, numero di matricola**) e il **numero** della prova d'esame. Al termine, **consegnare tutti i file sorgenti** e i file contenuti nello StartKit.
Nota: **il main non è opzionale**; i *test* richiesti vanno implementati.
Consiglio: per verificare l'assenza di *warning*, eseguire di tanto in tanto *"Rebuild All"*.

Una azienda specializzata nella costruzione di gru vuole realizzare un nuovo programma per la gestione dei pezzi di ricambio presenti nel proprio magazzino. Il programma dovrà offrire tre funzionalità principali: l'inserimento di nuovi pezzi nell'elenco dei pezzi disponibili in magazzino, il prelievo di pezzi dall'elenco dei pezzi in magazzino, e la ricerca di eventuali errori nell'elenco dei pezzi.
In particolare ogni pezzo è rappresentato da un insieme di informazioni: un codice **identificativo** unico del pezzo (un intero), una **descrizione** del pezzo (una stringa di al più 127 caratteri utili), la **quantità** disponibile a magazzino (un intero), e il **costo** unitario del pezzo (un float).

Esercizio 1 – Strutture dati Fattura, e funzioni di lett./scritt. (mod. element.h/c e pezzi.h/.c)

Si definisca un'opportuna struttura dati **`Pezzo`**, al fine di rappresentare i dati relativi ad un pezzo, come da descrizione precedente.

Si definisca la funzione:

```
Pezzo leggiUnPezzo();
```

che legga dallo standard inputi i dati relativi ad un nuovo pezzo e li restituisca in una apposita struttura dati. Si assuma che l'utente inserisca i dati in ordine, e termini ogni singolo campo con un carattere "invio" (quindi per un nuovo pezzo l'utente inserirà quattro volte il carattere invio: uno '**\n**' dopo l'id, uno '**\n**' dopo la descrizione, uno '**\n**' dopo la quantità, e infine uno '**\n**' dopo il costo unitario di un pezzo).

Il candidato implementi una funzione:

```
list inserimento(list esistente);
```

che riceva in ingresso una lista di strutture dati di tipo **`Pezzo`**, rappresentante i pezzi presenti a magazzino. La funzione ha lo scopo di permettere all'utente di inserire da standard input un insieme di nuovi pezzi. In particolare, la funzione deve come prima cosa chiedere all'utente se vuole inserire un pezzo. L'utente fornisce la propria risposta inserendo da standard input una stringa (di al massimo 5 caratteri utili): se la risposta è "si", allora deve leggere un pezzo usando obbligatoriamente la funzione precedente. Effettuata la lettura, la funzione dovrà richiedere all'utente se vuole inserire un altro pezzo, e così via. Quando l'utente segnalerà l'intenzione di non inserire altri pezzi, la funzione dovrà terminare e restituire una lista contenente sia i nuovi pezzi inseriti, sia quelli già presenti nella lista passata in ingresso come parametro.

Si definisca la procedura:

```
void salvaElenco(char * fileName, list elenco);
```

che, ricevuta in ingresso il nome di un file, e una lista di strutture dati di tipo **`Pezzo`**, scriva tale elenco sul file col nome specificato in ingresso. In particolare, le informazioni sul file dovranno essere scritte in formato binario.
Si definisca la funzione:

```
list leggiElenco(char * fileName);
```

che, ricevuto in ingresso il nome di un file binario contenente strutture dati di tipo **`Pezzo`**, legga dal file tali strutture dati e le restituisca tramite una lista. In caso di errore nell'apertura del file, deve essere restituita una lista vuota.
Il candidato abbia cura di realizzare nel main opportuni test al fine di verificare il corretto funzionamento delle funzioni di cui sopra.

Fondamenti di Informatica T-1, 2013/2014 – Modulo 2

Prova d'Esame 6A di Giovedì 11 Settembre 2014 – tempo a disposizione 2h

Esercizio 2 – Eliminazione dei pezzi ripetuti e ordinamento (modulo element.h/.c e pezzi.h/.c)

La procedura di inserimento specificata al punto 1 non effettua alcun controllo sul fatto che nuovi pezzi inseriti siano già presenti o meno a magazzino. Quindi, dopo un breve utilizzo del programma, a magazzino compaiono più ripetizioni dello stesso pezzo, ma con quantità e prezzi diversi.
Si definisca una funzione:

```
list accorpa(list magazzino);
```

che, ricevuta in ingresso una lista di strutture dati di tipo **`Pezzo`**, restituisca una nuova lista contenente l'elenco dei pezzi senza ripetizioni. Due strutture dati di tipo Pezzo si considerano uguali se hanno lo stesso codice identificativo e la stessa descrizione. Nella lista restituita in uscita la quantità di unità di un certo pezzo dovrà essere aggiornata opportunamente come la somma delle quantità a disposizione memorizzate nella lista in ingresso. Il prezzo unitario di un certo pezzo nella lista di uscita dovrà essere aggiornato al prezzo maggiore trovato nella lista di ingresso, relativamente al pezzo in questione. Ad esempio, se nella lista in ingresso ci sono due strutture riferibili a "bulloni", la prima con quantità 100 e costo 0.32€, e la seconda con quantità 50 e costo 0.45€, in uscita dovrà esserci una sola struttura riferibile a "bulloni", con quantità 150, e prezzo 0.45€.
A tal scopo prima si definiscano e si usino le seguenti funzioni di appoggio:

```
int member(Pezzo p, list l);
```

che restituisca un valore interpretabile come vero se la struttura dati p è presente nella lista l;

```
int sommaQuant(Pezzo p, list l);
```

che restituisca la somma delle quantità di un Pezzo p nella lista l;

```
float maxCosto(Pezzo p, list l);
```

che restituisca il massimo costo di un Pezzo p nella lista l.

Esercizio 3 –Ordinamento (modulo pezzi.h/.c)

Si realizzi una funzione:

```
Pezzo * ordina(list magazzino, int * dim);
```

che restituisca un vettore (della dimensione minima necessaria) di strutture dati di tipo **`Pezzo`**, contenente tutti i pezzi passati in ingresso tramite la lista, ma ordinati secondo il seguente criterio: in ordine lessicografico in base alla descrizione, e a parità di descrizione, in ordine crescente in base al codice identificativo. Tramite il parametro **`dim`** la funzione restituisca la dimensione del vettore restituito come risultato. Per effettuare l'ordinamento il candidato utilizzi uno qualunque degli algoritmi di ordinamento trattati durante il corso.
Il candidato abbia cura di realizzare nel main opportuni test al fine di verificare il corretto funzionamento delle funzioni di cui sopra.

Esercizio 4 – Lettura e salvataggio pezzi, e controllo ripetizioni (main.c)

Il candidato realizzi nella funzione **`main(…)`** un programma che chieda all'utente di specificare il nome del file binario contenente il magazzino attuale: il programma dovrà leggere da tale file l'elenco dei pezzi disponibili a magazzino; se il file non dovesse esistere, il programma dovrà comunque continuare, ma usando come lista dei pezzi attuali una lista vuota. Quindi il programma dovrà chiedere all'utente di inserire dei nuovi pezzi, usando le funzioni di cui al punto 1. Terminato l'inserimento, il programma dovrà provvedere ad accorpare i nuovi pezzi (punto 2), a salvare il nuovo elenco nel file binario del magazzino, e quindi stampare a video in maniera ordinata il novo elenco dei pezzi in magazzino.
Al termine del programma, il candidato abbia cura di de-allocare tutta la memoria allocata dinamicamente, ivi compresa la memoria allocata per le liste.

Fondamenti di Informatica T-1, 2013/2014 – Modulo 2

Prova d'Esame 6A di Giovedì 11 Settembre 2014 – tempo a disposizione 2h

"element.h":

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

#ifndef _ELEMENT_H
#define _ELEMENT_H

#define DIM_DESC 128
typedef struct {
	int id;
	char desc[DIM_DESC];
	int quant;
	float cost;
} Pezzo;
typedef Pezzo element;
int comparePezzo(Pezzo p1, Pezzo p2);

#endif
```

"element.c":

```
#include "element.h"

int comparePezzo(Pezzo p1, Pezzo p2) {
	int result = strcmp(p1.desc, p2.desc);
	if (result == 0)
		result = p1.id - p2.id;
	return result;
}
```

Fondamenti di Informatica T-1, 2013/2014 – Modulo 2

Prova d'Esame 6A di Giovedì 11 Settembre 2014 – tempo a disposizione 2h

"list.h"

```
#ifndef LIST_H
#define LIST_H

#include "element.h"

typedef struct      list_element
{
    element value;
    struct list_element  *next;
} item;

typedef  item* list;

typedef  int boolean;

/* PRIMITIVE  */
list emptylist(void);
boolean empty(list);
list cons(element, list);
element head(list);
list tail(list);

//void showlist(list l);
void freelist(list l);
//int member(element el, list l);
//list insord_p(element el, list l);

#endif
```

"list.c":

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

#include "list.h"

/* OPERAZIONI PRIMITIVE */
list  emptylist(void)            /* costruttore lista vuota */
{
    return NULL;
}

boolean  empty(list l)     /* verifica se lista vuota */
{
    return (l==NULL);
}

list  cons(element e, list l)
{
  list t;       /* costruttore che aggiunge in testa alla lista */
  t=(list)malloc(sizeof(item));
  t->value=e;
  t->next=l;
```

```
  return(t);
}

element head(list l) /* selettore testa lista */
{
  if (empty(l)) exit(-2);
  else return (l->value);
}

list  tail(list l)         /* selettore coda lista */
{
  if (empty(l)) exit(-1);
  else return (l->next);
}

void freelist(list l) {
       if (empty(l))
              return;
       else {
              freelist(tail(l));
              free(l);
       }
       return;
}
/*
list insord_p(element el, list l) {
       list pprec, patt = l, paux;
       int trovato = 0;

       while (patt!=NULL && !trovato) {
              if (compareTrasloco(el, patt->value)<0)
                     trovato = 1;
              else {
                     pprec = patt;
                     patt = patt->next;
              }
       }
       paux = (list) malloc(sizeof(item));
       paux->value = el;
       paux->next = patt;
       if (patt==l)
              return paux;
       else {
              pprec->next = paux;
              return l;
       }
}
*/
```

Fondamenti di Informatica T-1, 2013/2014 – Modulo 2

Prova d'Esame 6A di Giovedì 11 Settembre 2014 – tempo a disposizione 2h

"pezzi.h":

```
#ifndef _PEZZI_H
#define _PEZZI_H

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "element.h"
#include "list.h"

// es. 1
Pezzo leggiUnPezzo();
list inserimento(list esistente);
void salvaElenco(char * fileName, list elenco);
list leggiElenco(char * fileName);

// es. 2
int member(Pezzo p, list l);
int sommaQuant(Pezzo p, list l);
float maxCosto(Pezzo p, list l);
list accorpa(list magazzino);

// es. 3
Pezzo * ordina(list magazzino, int * dim);

#endif
```

"pezzi.c":

```
#include "pezzi.h"

Pezzo leggiUnPezzo() {
	Pezzo result;
	printf("Codice identificativo? ");
	scanf("%d", &(result.id) );
	printf("Descrizione (senza spazi)? ");
	scanf("%s", result.desc);
	printf("Quantita'? ");
	scanf("%d", &(result.quant));
	printf("Costo unitario? ");
	scanf("%f", &(result.cost));
	return result;
}

list inserimento(list esistente) {
	char answer[6];
	Pezzo temp;
	list result;

	result = esistente;
	do {
		printf("Vuoi inserire un pezzo?");
		scanf("%s", answer);
		if (strcmp("si",answer) == 0) {
			temp = leggiUnPezzo();
```

```
                result = cons(temp, result);
            }
        } while (strcmp("si",answer) == 0);
        return result;
}

void salvaElenco(char * fileName, list elenco) {
        FILE * fp;
        Pezzo temp;

        fp=fopen(fileName, "wb");
        if (fp == NULL) {
            printf("Errore nell'apertura del file %s in modalita' scrittura...",
fileName);
            exit(1);
        }
        else {
            while (!empty(elenco)) {
                temp = head(elenco);
                fwrite(&temp, sizeof(Pezzo), 1, fp);
                elenco = tail(elenco);
            }
            fclose(fp);
        }
        return;
}

list leggiElenco(char * fileName) {
        FILE * fp;
        list result;
        Pezzo temp;
        int res;

        fp=fopen(fileName, "rb");
        result = emptylist();
        if (fp == NULL) {
            printf("Errore nell'apertura del file %s ...", fileName);
        }
        else {
            res = fread(&temp, sizeof(Pezzo), 1, fp);
            while (res==1) {
                result = cons(temp, result);
                res = fread(&temp, sizeof(Pezzo), 1, fp);
            }
            fclose(fp);
        }
        return result;
}

int member(Pezzo p, list l) {
        int trovato = 0;
        while (!empty(l) && !trovato) {
            if (comparePezzo(p, head(l)) == 0)
                trovato = 1;
            l = tail(l);
        }
        return trovato;
```

```
}

int sommaQuant(Pezzo p, list l) {
	int result = 0;
	Pezzo temp;
	while (!empty(l)) {
		temp = head(l);
		if (comparePezzo(p, temp) == 0)
			result = result + temp.quant;
		l = tail(l);
	}
	return result;
}

float maxCosto(Pezzo p, list l) {
	float result = 0.0;
	Pezzo temp;

	while (!empty(l)) {
		temp = head(l);
		if (comparePezzo(p, temp) == 0 && temp.cost>result)
			result = temp.cost;
		l = tail(l);
	}
	return result;
}

list accorpa(list magazzino) {
	list result;
	list temp;
	Pezzo p;
	Pezzo newP;

	result = emptylist();
	temp = magazzino;
	while (!empty(temp)) {
		p = head(temp);
		if (!member(p, result)) {
			newP = p;
			newP.quant = sommaQuant(p, magazzino);
			newP.cost = maxCosto(p, magazzino);
			result = cons(newP, result);
		}
		temp = tail(temp);
	}
	return result;
}

// Es. 3
void scambia(Pezzo * p1, Pezzo * p2) {
	Pezzo temp;
	temp = *p1;
	*p1 = *p2;
	*p2 = temp;
}

void quickSortR(Pezzo a[], int iniz, int fine) {
```

```
	int i, j, iPivot;
	Pezzo pivot;
	if (iniz < fine) {
		i = iniz;
		j = fine;
		iPivot = fine;
		pivot = a[iPivot];
		do { /* trova la posizione del pivot */
			while (i < j && comparePezzo(a[i], pivot) <= 0) i++;
			while (j > i && comparePezzo(a[j], pivot) >= 0) j--;
			if (i < j) scambia(&a[i], &a[j]);
		} while (i < j);
		/* determinati i due sottoinsiemi */
		/* posiziona il pivot */
		if (i != iPivot && comparePezzo(a[i], a[iPivot]) != 0) {
			scambia(&a[i], &a[iPivot]);
			iPivot = i;
		}
		/* ricorsione sulle sottoparti, se necessario */
		if (iniz < iPivot - 1)
			quickSortR(a, iniz, iPivot - 1);
		if (iPivot + 1 < fine)
			quickSortR(a, iPivot + 1, fine);
	} /* (iniz < fine) */
} /* quickSortR */

Pezzo * ordina(list magazzino, int * dim) {
	Pezzo * result;
	list temp;
	int i;

	temp = magazzino;
	*dim = 0;
	while (!empty(temp)) {
		*dim = *dim + 1;
		temp = tail(temp);
	}
	result = (Pezzo*) malloc(sizeof(Pezzo) * *dim);
	temp = magazzino;
	for (i=0; i<*dim; i++) {
		result[i] = head(temp);
		temp = tail(temp);
	}
	quickSortR(result, 0, *dim-1);
	return result;
}
```

Fondamenti di Informatica T-1, 2013/2014 – Modulo 2

Prova d'Esame 6A di Giovedì 11 Settembre 2014 – tempo a disposizione 2h

"main.c":

```
#include <stdio.h>

#include "pezzi.h"

int main(int argc, char **argv)
{
	// es. 1
	/*
	{
		list elenco;
		elenco = emptylist();
		elenco = inserimento(elenco);
		salvaElenco("magaz.bin", elenco);
		showlist(elenco);
		freelist(elenco);
		elenco = leggiElenco("magaz.bin");
		showlist(elenco);
		freelist(elenco);
	}
	 * */

	// es. 2
	/*
	{
		list elenco;
		list elencoSemplice;
		elenco = leggiElenco("magaz.bin");
		elencoSemplice = accorpa(elenco);
		showlist(elenco);
		showlist(elencoSemplice);
		freelist(elenco);
		freelist(elencoSemplice);
	}
	 */

	// es. 3
	/*
	{
		list elenco;
		list elencoSemplice;
		Pezzo * vett;
		int dim;
		int i;
		elenco = leggiElenco("magaz.bin");
		elencoSemplice = accorpa(elenco);
		vett = ordina(elencoSemplice, &dim);
		for (i=0; i<dim; i++)
			printf("%d %s %d %f\n", vett[i].id, vett[i].desc, vett[i].quant,
vett[i].cost);
		free(vett);
		freelist(elenco);
		freelist(elencoSemplice);
	}
	 */

	// es. 4
```

Fondamenti di Informatica T-1, 2013/2014 – Modulo 2

Prova d'Esame 6A di Giovedì 11 Settembre 2014 – tempo a disposizione 2h

```
	{
		list esistente;
		list accorpato;
		char nomeFile[16];
		Pezzo * vett;
		int dim, i;

		printf("Nome del file contenente il magazzino? ");
		scanf("%s", nomeFile);
		esistente = leggiElenco(nomeFile);
		printf("Esistente:\n");
		showlist(esistente);

		esistente = inserimento(esistente);

		accorpato = accorpa(esistente);
		salvaElenco(nomeFile, accorpato);
		vett = ordina(accorpato, &dim);
		for (i=0; i<dim; i++)
			printf("%d %s %d %f\n", vett[i].id, vett[i].desc, vett[i].quant,
vett[i].cost);
		free(vett);
		freelist(esistente);
		freelist(accorpato);

	}

	getchar();
	return 0;
}
```