Esame di Fondamenti di Programmazione
1-9-09 – canale AD – Sterbini

Esercizio 1 (matrici – 9 punti)

Si supponga di avere una scacchiera (8x8) e di voler visitare tutte le sue caselle facendo solo mosse del cavallo (che si sposta solo di 1 riga e 2 colonne oppure di 2 righe e 1 colonna, si veda la figura). Data una sequenza di mosse codificate ciascuna dalla differenza di righe e di colonne tra la posizione corrente e quella successiva, **dobbiamo verificare se tutte le mosse sono valide e se visitano tutte le caselle della scacchiera**.

- una mossa è valida se rimane nella scacchiera.
- la sequenza è sempre lunga 63 elementi (che sono le caselle da visitare dopo la prima).
- la sequenza può ripassare su una casella già visitata, ma in questo caso non riuscirà a completare la visita di tutte le 64 caselle della scacchiera.

Si scriva la funzione C **controlla_cavallo** che riceve come argomenti: le **coordinate** della casella iniziale, un vettore di struct di 63 elementi che contiene le **mosse**, e tutti gli altri argomenti che ritenete necessari, e che torna come risultato 1 se si sono visitate tutte le caselle e 0 altrimenti. Per ciascuna mossa valida si stampino le due coordinate della casella. La funzione deve uscire non appena arriva ad una mossa non valida o ad una casella già visitata.

	-1,+2		**+1,+2**				
-2,+1				**+2,+1**			
		C					
-2,-1				**+2,-1**			
	-1,-2		**+1,-2**				

Mosse disponibili a partire dalla posizione C, con i corrispondenti spostamenti

Soluzione

```
typedef struct mossa {                    // definisco la struct che contiene la mossa
     int dx;
     int dy;
} Mossa;
int controlla_cavallo(int x, int y, Mossa mosse[63]) {
     int scacchiera[8][8] = { 0 };                  // creo una scacchiera vuota
     int posx = x;                                  // mi metto nella posizione iniziale
     int posy = y;
     int i;
     scacchiera[y][x] = 1;                          // e marco la casella come visitata
     for (i=0 ; i<63 ; i++) {                       // per 63 volte
          printf("%d %d\n", posx, posy);            // stampo la posizione
          posx += mosse[i].dx;                      // mi muovo di dx
          posy += mosse[i].dy;                      // e dy rispetto alla pos corrente
          // se esco dalla scacchiera o torno su una casella visitata
          if (posx < 0 || posx > 7 || posy < 0 || posy > 7 || scacchiera[posy][posx])
               return 0;                            // esco tornando 0
          else                                      // altrimenti
               scacchiera[posy][posx] = 1;          // marco la casella visitata
     }
     return 1;}                                     // se arrivo qui le ho visitate tutte
```

Esercizio 2 (stringhe – 9 punti)

Si supponga di dover recuperare per forza bruta una password dimenticata e che si disponga solo della forma crittata della password (una stringa) ed un **dizionario**, che è un vettore di stringhe che contiene un elenco di **N** parole ciascuna lunga al massimo 8 caratteri.

Si scriva la funzione C **recupera_password** che riceve come argomenti: la **password** crittata (una stringa), il **dizionario** (un vettore di stringhe), il valore intero **N** e tutti gli altri argomenti che ritenete necessari e che cerca la password in chiaro provando a crittare una per una le parole prese dal dizionario, e che torna la prima password trovata. Una password è stata trovata la se sua versione crittata con crypt è uguale alla password da decrittare (per uno dei possibili valori della coppia di caratteri **salt, vedi sotto**).

Per crittare ciascuna parola si usi la funzione di sistema **crypt** di Unix, il cui prototipo è

char *crypt(const char *_password_, const char *_salt_);

che, data una **password** in chiaro, ed un valore di inizializzazione chiamato **salt,** che è una coppia di caratteri presi dall'insieme [**a-zA-Z0-9./**] (e quindi può assumere 4096 valori diversi), fornisce come risultato la stringa che contiene la sua forma crittata.

Soluzione

Il cuore del problema era saper costruire una stringa di due caratteri e controllare se due stringhe (le parole crittate) sono uguali

```
char *crypt(const char *password, const char *salt);     // prototipo di crypt
char * recupera_password( char * crittata, char * dizionario[] , int N) {
     char * alfabeto = "abcde...xyzABC...XYZ01...89./";  // caratteri del salt
     int i;
     char * c1;                                  // puntatori che scorrono l'alfabeto
     char * c2;
     char * temp;                                // parola corrente crittata
     char salt[3] = { 0 };            // stringa di due caratteri (più '\0' terminatore)
     for (i=0 ; i<N ; i++) {                     // per tutte le parole del dizionario
          for (c1=alfabeto ; *c1 ; c1++) {       // per tutti i caratteri dell'alfabeto
               salt[0] = *c1;                    // copio il primo carattere in salt
               for (c2=alfabeto ; *c2 ; c2++) {  // per tutti i caratteri dell'alfabeto
                    salt[1] = *c2;               // copio il secondo carattere in salt
                    temp = crypt(dizionario[i],salt);     // provo a crittare la parola
                    if (strcmp(temp, crittata) == 0) {    // se viene lo stesso risultato
                         printf(dizionario[i]);           // stampo la parola in chiaro
                         return &(dizionario[i]);         // e la torno come risultato
                    }
               }
          }
     }
     printf("password non trovata");             // se arrivo qui non l'ho trovata
}
```

Esercizio 3 (alberi – 9 punti)

Si ricorda che in un albero binario di ricerca ogni nodo ha un valore maggiore di **tutti** i valori memorizzati nei nodi del suo sottoalbero sinistro e minore di **tutti** i valori dei nodi del suo sottoalbero destro.

Sia dato un tipo struct definito come segue:

```
typedef struct nodo {
        int valore;
        long memo;
        struct nodo * figlio_sx;
        struct nodo * figlio_dx;
} Nodo;
```

Scrivere una funzione **aggiorna_inserisci** che riceve come argomenti un puntatore a Nodo **albero**, un intero **valore** e un long **memo** (e tutti gli altri argomenti che ritenete necessari); la funzione deve ricercare all'interno dell'albero binario di ricerca (puntato dall'argomento albero) un nodo con campo valore uguale all'intero valore ricevuto come argomento. Se tale nodo è presente all'interno dell'albero, la funzione deve aggiornare il campo memo del nodo individuato con il valore dell'argomento memo. Altrimenti, la funzione deve creare un nuovo nodo, assegnare ai campi valore e memo del nodo i valori degli argomenti valore e memo, e inserire il nuovo nodo nell'abero binario di ricerca (nella corretta posizione).

Soluzione

Per aggiornare un albero bisogna passarlo per riferimento, l'esercizio è uguale ad un normale inserimento in un albero binario di ricerca (vedete il libro), con la sola modifica che se si trova un nodo che contiene il valore bisogna cambiare il campo memo invece di ignorarlo.

```
void aggiorna_inserisci(Nodo ** albero, int valore, long memo) {
     if (*albero == NULL) {                          // se l'albero è vuoto
          *albero = malloc(sizeof(Nodo));            // va aggiunto un nuovo nodo
          (*albero)->val = valore;
          (*albero)->memo = memo;
          (*albero)->figlio_sx = NULL;
          (*albero)->figlio_dx = NULL;
     } else                                          // altrimenti c'è almeno la radice
          if ((*albero)->val == valore)              // se la radice contiene il valore
               (*albero)->memo = memo;               // aggiorno il campo memo
          else {    // altrimenti devo decidere se scendere a sinistra o a destra
               if ((*albero)->val > valore)          // se la radice è maggiore
                    // devo aggiornare il sottoalbero sinistro
                    aggiorna_inserisci(&((*albero)->figlio_sx), valore, memo);
               else                                  // altrimenti è minore
                    // devo aggiornare il sottoalbero destro
                    aggiorna_inserisci(&((*albero)->figlio_dx), valore, memo);
          }
}
```

Esercizio 4 (liste – 9 punti)

Sia dato un tipo struct definito come segue:

```
typedef struct nodo {
        int valore;
        struct nodo * next;
} Nodo;
```

Scrivere una funzione **prima_pari** che riceve come argomenti una **coda** con elementi di tipo Nodo. La funzione deve estrarre il nodo in testa alla coda; se il suo valore è pari deve stampare ed eliminare il nodo (deallocando correttamente la memoria), mentre se il valore è dispari il nodo va reinserito in fondo alla coda, incrementandone il valore di 1. La funzione deve ripetere questa operazione finchè la coda non è vuota.

Soluzione

Per aggiornare una coda bisogna passare sia la testa che la coda per riferimento.

Svolgo l'esercizio definendo le classiche funzioni enqueue e dequeue che sono anche sul libro

```
int deque(Nodo ** testa, Nodo ** coda) {
    Nodo * primo = *testa;         // prendo il primo nodo
    int val = 0;
    if (primo) {                   // se c'e' almeno un nodo nella coda
        *testa = primo->next;      // aggiorno la testa spostandola sul prossimo
        val = primo->val;          // leggo il valore
        free(primo);               // libero il nodo
    };
    if (*testa == NULL)            // caso speciale, ho svuotato tutta la coda
        *coda = NULL;              // aggiorno il puntatore alla coda
    return val;                    // torno il valore
}
void enqueue(Nodo ** testa, Nodo ** coda, int valore) {     // accodo un valore
    Nodo * nuovo = malloc(sizeof(Nodo));       // alloco il nodo
    nuovo->val = valore;           // ci metto il valore
    nuovo->next = NULL;            // e NULL perchè sarà l'ultimo
    if (*coda)                     // se c'era almeno un nodo nella coda
        (*coda)->next = nuovo;     // gli aggiungo quello nuovo
    *coda = nuovo;                 // e aggiorno il puntatore all'ultimo
    if (*testa == NULL)            // se non c'era nessun nodo
        *testa = nuovo;            // la testa punta a quello nuovo
}
void prima_pari(Nodo ** testa, Nodo ** coda) {
    int valore;
    while (*testa) {                           // ripeto finchè ci sono nodi
        valore = deque(testa, coda);           // estraggo e dealloco il primo
        if (valore % 2)                        // se è dispari
            enqueue(testa, coda, valore+1);    // lo riaccodo incrementato
        else                                   // altrimenti è pari
            printf("%d\n", valore);            // e lo stampo
    }
}
```