#include <stdio.h>  
#include <stdlib.h>  
#include "Btree.h"  
#include "queue.h"  
#include "list.h"  
/\*  
TRACCIA:  
restituisce il conteggio dei nodi che hanno un fratello nell'albero T  
dato in input. Nel conteggio il fratello sinistro del fratello destro e  
viceversa contano 2.  
  
SPECIFICA SINTATTICA:  
fratelli(albero binario) -> intero  
SPECIFICA SEMANTICA:  
fratelli(Btree T) -> int  
PRECONDIZIONI:  
 -  
POSTCONDIZIONI:  
ritorna il numero dei fratelli nel Btree  
  
PROGETTAZIONE:  
  
PASSI BASE:   
 se un nodo è vuoto allora ritorno 0  
 altrimenti se un nodo ha sia il figlio sx che il dx allora aggiungo +2 al contatore  
PASSO RICORSIVO: sommo al contatore i fratelli del sottoalbero sx e del sottoalbero dx  
\*/  
int fratelli(Btree T) {  
 int contatore = 0;  
 if (emptyBtree(T)) {  
 return 0;  
 }  
 if (figlioSX(T) != NULL && figlioDX(T) != NULL) {  
 contatore = 2;  
 }  
 return contatore + fratelli(figlioSX(T)) + fratelli(figlioDX(T));  
}  
  
/\*  
TRACCIA: restituisce VERO se le code q1 e q2 contengono gli stessi elementi in ordine  
speculare l'una rispetto all'altra. Le due code al termine della funzione devono risultare inalterate  
  
SPECIFICA SINTATTICA:  
speculare(CODA, CODA) -> INTERO  
SPECIFICA SEMANTICA:  
speculare(queue q1, queue q2) -> int  
PRECONDIZIONI:  
q1 e q2 non nulle  
POSTCONDIZIONI:  
q1 = <a1, a2, ... , an> e q2 = <an, ... , a2, a1> allora VERO  
FALSO altrimenti  
  
PROGETTAZIONE:  
1.0 se la dimensione delle due code è diversa posso già dire che non sono speculari  
2.0 creo un array che conterrà gli elementi della coda e li reincodo per preservarla  
3.0 scorrendo l'array in senso inverso:  
 confronto se l'elemento in coda è uguale al i-esimo elemento nell'array  
 se sono uguali continuo a confrontare risistemando la coda  
 altrimenti non effettuo più confronti e risistemo solamente la coda  
4.0 ritorno se è speculare o no   
\*/  
int speculare(queue q1, queue q2) {  
 int size\_q1 = sizeQueue(q1);  
 int size\_q2 = sizeQueue(q2);  
 if (size\_q1 != size\_q2) return 0;  
  
 item copia\_q1[size\_q1];  
 for (int i = 0; i < size\_q1; i++) {  
 item el = dequeue(q1);  
 copia\_q1[i] = el;  
 enqueue(el, q1);  
 }  
  
 int is\_speculare = 1;  
 for (int i = size\_q1-1; i >= 0; i--) {  
 item el = dequeue(q2);  
 if (!eq(copia\_q1[i], el && is\_speculare)) {  
 is\_speculare = 0;   
 }  
 enqueue(el, q2);  
 }  
 return is\_speculare;  
}  
  
/\*  
TRACCIA: item max\_duplicazioni(list L); che restituisce l'item maggiormente presente nella lista,  
in altri termini quelo che ha il maggior numero di duplicati nella lista.  
  
SPECIFICA SINTATTICA:  
max\_duplicazoni(LISTA) -> ITEM  
SPECIFICA SEMANTICA:  
max\_duplicazoni(list L) -> item  
PRECONDIZIONI:  
L non deve essere nullo  
POSTCONDIZIONI:  
ritorna l'elemento che appare più volte in L  
  
PROGETTAZIONE:  
1.0 creo una lista di item elementi e una lista di interi occorrenze  
2.0 per ogni elemento di L  
 se è stato già trovato precedentemente  
 nella stessa posizione nella lista delle occorrenze aumenta il contatore  
 altrimenti aggiungi alla lista l'elemento e l'occorenze ad uno  
3.0 cerco l'indice del numero più alto di occorrenze  
4.0 restituisco il rispettivo elemento alla stessa posizione  
  
ESEMPIO:  
L = <a, b, c, a, c, a>  
elementi | occorrenze  
 a | 3  
 b | 1   
 c | 2   
\*/  
item max\_duplicazioni(list L) {  
 list elementi = newList();  
 list occorrenze = newList(); // lista di interi  
 for (int i = 0; i < sizeList(L); i++) {  
 item actual = getItem(L, i);  
 int posizione\_trovata;  
 if ((posizione\_trovata = posItem(elementi, actual)) != -1) { // actual si trova in elementi  
 int apparizioni = getItem(occorrenze, posizione\_trovata);  
 apparizioni++;  
 // per aggiornarlo devo eliminarlo e poi inserirlo nella stessa posizione  
 removeList(occorrenze, posizione\_trovata);  
 insertList(occorrenze, posizione\_trovata, apparizioni);  
 }  
 else {  
 insertList(elementi, sizeList(elementi), actual);  
 insertList(occorrenze, sizeList(occorrenze), 1);  
 }  
 }  
 int max\_occorrenze = -999;  
 int max\_occorrenze\_i = -1;  
 for (int i = 0; i < sizeList(occorrenze); i++) {  
 int actual = getItem(occorrenze, i);  
 if (max\_occorrenze < actual) {  
 max\_occorrenze = actual;  
 max\_occorrenze\_i = i;  
 }  
 }  
 return getItem(elementi, max\_occorrenze\_i);

}