

Alberi binari

Definizione della struttura dati:

```
struct tree
{
    int dato;
    struct tree *sx, *dx;
};

typedef struct tree tree;
```

Esercizi su alberi binari

1. Scrivere una funzione che cerchi un intero k all'interno di un albero binario.
2. Scrivere una funzione che restituisca il valore massimo contenuto nei nodi di un albero binario.
3. Scrivere una funzione che effettui la visita in ordine di un albero binario utilizzando l'iterazione.
4. Scrivere una funzione che verifichi se due alberi binari sono uguali
5. Scrivere una funzione che, dato un albero, inserisca in una lista solo i nodi di livello pari
6. Scrivere una funzione che, dato un albero che rappresenta una espressione (es. $(1+2)*5$), ne calcoli il valore e lo restituisca in output

Scrivere sempre pre e post condizione di ogni funzione

Esercizio 1

Scrivere una funzione che cerchi un intero k all'interno di un albero binario.

Pre condizioni:

La funzione prende in ingresso un albero binario e un intero k da cercare nei nodi dell'albero

Post condizioni:

La funzione restituisce 1 se k è contenuto in almeno un nodo, 0 altrimenti. Se l'albero è vuoto, la funzione restituisce 0.

Implementazione ricorsiva

```
int ricerca(tree *t, int k)
{
    if (t == NULL) return 0;

    return t->dato == k
        || ricerca(t->sx, k)
        || ricerca(t->dx, k);
}
```

Esercizio 2

Scrivere una funzione che restituisca il valore massimo contenuto nei nodi di un albero.

Pre condizioni:

la funzione prende in ingresso un albero binario

Post condizioni:

La funzione restituisce il valore massimo tra quelli presenti nei nodi dell'albero, -1 se l'albero è vuoto.

Svolgimento

```
#define max3(a,b,c) (a > b ? (c > a ? c : a)
: (c > b ? c : b))

int get_max(tree *t)
{
    int max_sx, max_dx;
    /* caso base: albero vuoto */
    if (t == NULL) return -1;

    max_sx = get_max(t->sx);
    max_dx = get_max(t->dx);

    return max3(max_sx, max_dx, t->dato);
}
```

Esercizio analogo: restituire la profondità massima di un albero.

Esercizio 3

Scrivere una funzione che effettui la visita in ordine di un albero binario **utilizzando l'iterazione**.

Pre condizioni:

la funzione prende in ingresso un albero binario

Post condizioni:

la funzione stampa i valori associati ai nodi dell'albero "in ordine"

Svolgimento

Ricordiamo l'implementazione ricorsiva della visita in ordine:

```
void in_ordine(tree *t)
{
    if (t != NULL)
    {
        in_ordine(t->sx);
        printf("%d", t->dato);
        in_ordine(t->dx);
    }
}
```

Idea

L'implementazione iterativa della visita in ordine è meno intuitiva:

- a) è necessario utilizzare una pila,
- b) si visita l'albero scendendo iterativamente verso sinistra e inserendo nella pila i nodi visitati,
- c) quando non ci sono più nodi a sinistra, se la pila è vuota, esci,
- d) si estrae il nodo in cima alla pila, lo si stampa e si scende a destra,
- e) si torna al passo (b).

Svolgimento

```
void in_ordine(tree *t) {
    pila *p = crea_pila();
    while(1) {
        while(t != NULL)
        {
            push(p, t);
            t = t->sx;
        }
        if (pila_vuota(p)) break;
        t = pop(p);
        printf("%d", t->dato);
        t = t->dx;
    }
}
```

Esercizio 4

Scrivere una funzione che verifichi se due alberi binari sono uguali.

Pre condizioni:

la funzione prende in ingresso due alberi binari eventualmente vuoti

Post condizioni:

la funzione restituisce 1 se i due alberi hanno la stessa struttura e i medesimi contenuti

Svolgimento

```
int uguali(tree *t1, tree *t2) {
    /* caso base: entrambi gli alberi vuoti */
    if ((t1 == NULL)&&(t2 == NULL)) return 1;
    /* caso base: un albero vuoto, un albero no */
    if ((t1 == NULL)|| (t2 == NULL)) return 0;
    /* i due nodi in esame devono contenere lo
    stesso valore e i due sottoalberi sx e dx
    devono essere uguali (ricorsivamente) */
    return t1->dato == t2->dato
        && uguali(t1->sx, t2->sx)
        && uguali(t1->dx, t2->dx);
}
```

Esercizio 5

Scrivere una funzione che, dato un albero in input, inserisca in una lista solo i nodi di livello pari.

Pre condizioni:

La funzione prende in ingresso un albero binario e un puntatore a puntatore a una lista

Post condizioni:

La funzione imposta i nodi di livello pari nella lista.

Implementazione

```
void attraversa_pari(tree *t, list **l);
void attraversa_dispari(tree *t, list **l)
{
    if (t == NULL) return;
    attraversa_pari(t->sx, l);
    attraversa_pari(t->dx, l);
}
void attraversa_pari(tree *t, list **l)
{
    list *nuovo;
    if (t == NULL) return;
    nuovo = (list *)malloc(sizeof(list));
    nuovo->next = *l;
    nuovo->dato = (void *)t->dato;
    *l = nuovo;

    attraversa_dispari(t->sx, l);
    attraversa_dispari(t->dx, l);
}
```

Esercizio 6

Scrivere una funzione che, dato un albero che rappresenta una espressione (es. $(1+2)*5$), ne calcoli il valore e lo restituisca in output.

- Nota: I nodi intermedi rappresentano gli operatori, le foglie dell'albero rappresentano gli operandi

Pre condizioni:

La funzione prende in ingresso un albero binario che rappresenta un'espressione (si suppone che l'albero codifichi correttamente l'espressione)

Post condizioni:

La funzione restituisce il valore dell'espressione.

Implementazione

```
int get_value(tree *t) {
    int val_sx, val_dx;
    /* foglia: intero */
    if (t->sx == NULL && t->dx == NULL) return t->val;
    /* nodo intermedio: operatore
    val_sx = get_value(t->sx);
    val_dx = get_value(t->dx);
    switch(t->val)
    {
        case '+': return val_sx+val_dx;
        case '-': return val_sx-val_dx;
        case '*': return val_sx*val_dx;
        case '/': return val_sx/val_dx;
    }
    printf("espressione mal formata!");
    return -1;
}
```