Zadaci za vježbu za 1. kolokvij -- neka rješenja

Napomena: postoje i kraća rješenja nekih zadataka, no vjerojatno se iz ovako napisanih može lakše učiti.

Zadatak 5.

```
typedef struct {
    int alocirano, vrhStoga;
    elementtype *element;
} Stack;

void StMakeNull (Stack *S)
{
    S->alocirano = S->vrhStoga = 0;
    S->element = NULL;
}

void StPush (elementtype x, Stack *S)
{
    if (S->vrhStoga == S->alocirano) // nema mjesta za novi element?
    {
        S->element = (elementtype *)realloc (S->element, sizeof(elementtype)*(S->alocirano + 10));
        if (S->element == NULL)
            error("StPush: nema dovoljno memorije za realokaciju!");

        S->alocirano += 10;
    }

    S->element[S->vrhStoga] = x;
    S->vrhStoga++;
}

void StPop (Stack *S)
{
    if (StEmpty(*S)) // ili if (S->vrhStoga == 0)
        error("StPop: stog je prazan!");

    S->vrhStoga--;

    if (S->vrhStoga + 10 < S->alocirano) // ima vise od 10 praznih mjesta?
    {
        S->element = (elementtype *)realloc (S->element, sizeof(elementtype)*(S->alocirano - 10));
        if (S->element == NULL)
            error("StPop: nema dovoljno memorije za realokaciju!");

        S->alocirano -= 10;
    }
}
```

Ako vam je nepoznata naredba `void *realloc(void *stari_blok, int nova_velicina)`, njezin efekt je ovakav: neka je `stara_velicina` broj byteova kojeg zauzima `stari_blok`. Tada `realloc` alocira novi blok memorije veličine `nova_velicina` i u njega prekopira prvih min{`stara_velicina`, `nova_velicina`} byteova koji su se nalazili na mjestu na koje je pokazivao `stari_blok`. Znači, ako je `stara_velicina==0`, tj. `stari_blok==NULL`, onda se `realloc` ponaša posve isto kao `malloc`. Funkcija vraća pokazivač na novi blok memorije ili `NULL` ako alokacija nije uspjela.

Zadatak 10.

Promotrimo jedan primjer: `Q=(2,5,3,2,6,5,7,9)`.
Zašto je potreban dodatni atp? Zato da bismo mogli proći kroz sve elemente reda bez da ih "izgubimo": na primjer, da bismo došli do npr. osmog elementa u `Q`, trebamo maknuti iz reda prvih sedam, a budući da će nam oni trebati kasnije, moramo ih negdje spremiti. Mogli bismo tih 7 elemenata samo premještati sa početka na kraj reda, no tada nikako ne bi mogli zaključiti koliko red ima elemenata (zavrtili bi se u krug jer bi uvjet u `while(!QuEmpty(*Q))` bio uvijek istinit).
Zato koristimo dodatni atp; uzmimo još jedan Queue i označimo ga sa `pom`. Algoritam proveden na gornjem primjeru je sljedeći:

1. uzmimo prvi element od `Q`: `x=2` i izbacimo ga iz reda --> `x=2, Q=(5,3,2,6,5,7,9), pom=()`
2. sve dok se `Q` ne isprazni, uzimamo prve elemente od `Q`, izbacujemo iz `Q` i ubacujemo ih u `pom` samo ako su različiti od `x` --> `x=2, Q=(), pom=(5,3,6,5,7,9)`
3. vratimo `x` u `Q`, te nakon toga i sve elemente od `pom` --> `x=2, Q=(2,5,3,6,5,7,9), pom=()`

Sada smo sigurni da u `Q` postoji samo jedna kopija prvog elementa.
Kako osigurati da postoji jedna kopija drugog elementa? Potrebno je samo malo modificirati gornji algoritam: prvo u `pom` prebacimo jedan element iz `Q`, pa napravimo korake 1. i 2. Nakon njih je `x=5, Q=(), pom=(2,3,6,7,9)`. Zatim prebacimo jedan element iz `pom` u `Q`, pa prebacimo `x`, pa onda ostale elemente iz `pom` (to je korak 3). Dakle, imamo `Q=(2,5,3,6,7,9)`, tj. u `Q` više nema duplikata niti prvog niti drugog elementa. Slično je za uklanjanje duplikata k-tog elementa: prije prvog koraka prebacimo k elemenata iz `Q` u `pom`, i vratimo ih prije trećeg koraka. Kod izgleda ovako:

```
int SQ (Queue *Q)
{
    Queue pom; // elementtype je isti kao kod Q
    elementtype x, y;
    int i, brojRijesenih=0, gotovo=0, maxDuplica=0;

    QuMakeNull(&pom);
    while (!gotovo)
    {
        // prebaci brojRijesenih elemenata iz Q u pom
        for (i=0; i < brojRijesenih; i++)
        {
            y = QuFront(*Q); QuDequeue(Q);
            QuEnqueue (y, &pom);
        }

        if (!QuEmpty(*Q))
        {
            // uzmi prvi element iz Q (korak 1)
            int brojDuplica = 1;
            x = QuFront(*Q); QuDequeue(Q);

            // kopiraj sve elemente od Q koji nisu jednaki x u pom (korak 2)
            while (!QuEmpty(*Q))
            {
                y = QuFront(*Q); QuDequeue(Q);
                if (x != y)
                    QuEnqueue(y, &pom);
                else
                    brojDuplica++;
            }

            // vrati brojRijesenih elemenata iz pom u Q
            for (i=0; i < brojRijesenih; i++)
            {
                y = QuFront(pom); QuDequeue(&pom);
                QuEnqueue(y, Q);
            }

            // korak 3: vrati x u Q
            QuEnqueue(x, Q);
            if (brojDuplica > maxDuplica)
                maxDuplica = brojDuplica;
            brojRijesenih++;
        }
        else
            gotovo = 1; // ako je ovdje red prazan, onda smo gotovi

        // vrati i ostale iz pom u Q
        while (!QuEmpty(pom))
        {
            y = QuFront(pom); QuDequeue(&pom);
            QuEnqueue(y, Q);
        }
    }

    return maxDuplica;
}
```

Zadatak 15.
Očito ćemo trebati obići cijelo stablo i prebrojati koliko koji čvor ima k-potomaka. Znači možemo odmah pisati:

```
int max_kPotomaka;   // globalne
labeltype max_label; // varijable

labeltype potomak (BinaryTree B, int k)
{
    max_kPotomaka = 0;
    obidjiStablo (BiRoot(B), B, k);
    return max_label;
}

void obidjiStablo (node n, BinaryTree B, int k)
{
    int koliko;

    if (n == LAMBDA) return;

    koliko = broj_kPotomaka_od_n(n, B, k);
    if (koliko >= max_kPotomaka)
        max_kPotomaka = koliko, max_label = BiLabel(n, B);

    obidjiStablo (BiLeftChild(n, B), B, k);
    obidjiStablo (BiRightChild(n, B), B, k);
```

```
}
```

Treba nam još funkcija `broj_kPotomaka_od_n`. Da nju napišemo, treba samo uočiti da je čvor `m` k-potomak čvora `n` ako i samo ako je `m` (k-1)-potomak lijevog ili desnog djeteta od `n`, te da je svaki čvor svoj 0-potomak.

```
int broj_kPotomaka_od_n (node n, BinaryTree B, int k)
{
    if (n == LAMBDA) return 0;
    if (k == 0) return 1;

    return broj_kPotomaka (BiLeftChild(n, B), B, k-1) + broj_kPotomaka (BiRightChild(n, B), B, k-1);
}
```

Zadatak 17.
Zadatak je vrlo sličan prethodnom:

```
int min_nivo_lista;                 // globalne
labeltype label_lista_na_min_nivou; // varijable

labeltype najvisi_list (BinaryTree B)
{
    min_nivo_lista = -1;
    obidjiStablo (BiRoot(B), B);
    return label_lista_na_min_nivou;
}

void obidjiStablo (node n, BinaryTree B)
{
    if (n == LAMBDA) return;

    if (BiLeftChild (n, B) == LAMBDA && BiRightChild (n, B) == LAMBDA) // znaci, n je list
    {
        int nivo_od_n = nivo_cvora (n, B);

        if (nivo_od_n < min_nivo_lista)
            min_nivo_lista = nivo_od_n, label_lista_na_min_nivou = BiLabel(n, B);
        else if (nivo_od_n == min_nivo_lista && BiLabel(n, B) < label_lista_na_min_nivou)
            label_lista_na_min_nivou = BiLabel(n, B);
    }

    obidjiStablo (BiLeftChild(n, B), B);
    obidjiStablo (BiRightChild(n, B), B);
}
```

Za funkciju `nivo_cvora` iskoristite rješenje zadatka 16(f).

Zadatak 18.

```
Tree ogledalo (Tree T)
{
    Tree kopija;

    if (TrRoot(T) == LAMBDA) // ako je T prazno stablo
        return kopija; // onda je i slika u ogledalu prazno stablo
    else
    {
        TrMakeRoot(TrLabel(TrRoot(T), T), &kopija); // inace, korijen slike u ogledalu je isti kao korijen od T
        kopiraj_djecu (TrRoot(T), T, TrRoot(kopija), &kopija); // i treba mu ubaciti djecu od korijena od T
    }
}
```

Funkcija `kopiraj_djecu` dobiva dva čvora (`korijenOriginal` i `korijenKopija`) koji predstavljaju korijene podstabala u originalu i kopiji koji nas trenutno zanimaju. Ona treba prekopirati svu djecu od `korijenOriginal` tako da ona budu djeca od `korijenKopija` ali u obrnutom redoslijedu. Također treba u ogledalu prekopirati i podstabla od te djece (tj. ona stabla kojima su ta djeca korijeni). Kako ubaciti djecu da budu u naopakom poretku? Ako redoslijedom prvo_dijete -> iduci_brat -> iduci_brat -> ... obilazimo djecu od `korijenOriginal` i svako dijete ubacimo tako da bude prvo od `korijenKopija`, onda će ta djeca imati upravo suprotan redoslijed nego u originalnom stablu. Ovo ipak možemo izvesti i bez dodatnog stoga (isprike ako je nekog navelo na krivi put):

```
void kopiraj_djecu (node korijenOriginal, Tree original, node korijenKopija, Tree *kopija)
{
    node dijete, dijete_kopija;

    if (korijenOriginal == LAMBDA) return;

    for (dijete = TrFirstChild(korijenOriginal, original); dijete != LAMBDA; dijete = TrNextSibling(korijenOriginal, original))
    {
        dijete_kopija = TrInsertChild (TrLabel(dijete, original), korijenKopija, kopija);
        kopiraj_djecu (dijete, original, dijete_kopija, kopija);
    }
}
```

Evo i to nešto kompliciranije rješenje sa dodatnim stogom (ovdje nam stog služi za preokretanje redoslijeda djece, pa ih umjesto kao "prvo dijete" možemo ubacivati kao "zadnjeg brata"):

```
void kopiraj_djecu (node korijenOriginal, Tree original, node korijenKopija, Tree *kopija)
{
    node dijete, dijete_kopija;
    Stack S; // elementtype je node

    if (korijenOriginal == LAMBDA) return;

    StMakeNull (&S);
    for (dijete = TrFirstChild(korijenOriginal, original); dijete != LAMBDA; dijete = TrNextSibling(korijenOriginal, original))
        StPush (dijete, &S);

    if (!StEmpty(S))
    {
        dijete = StTop(S); StPop(&S);
        dijete_kopija = TrInsertChild (TrLabel(dijete, original), korijenKopija, kopija);
        kopiraj_djecu (dijete, original, dijete_kopija, kopija);
    }

    while (!StEmpty(S))
    {
        dijete = StTop(S); StPop(&S);
        dijete_kopija = TrInsertSibling (TrLabel(dijete, original), dijete_kopija, kopija);
        kopiraj_djecu (dijete, original, dijete_kopija, kopija);
    }
}
```