Općenita lista (List)

Lista je konačan niz podataka istog tipa, $(a_1, a_2, ..., a_n)$.

a.t.p. List

`elementtype`	. . .	bilo koji tip
`List`	. . .	konačan niz podataka tipa `elementtype`
`position`	. . .	podatak ovog tipa služi za identificiranje elemenata u listi
`LiEnd(L)`	. . .	vraća poziciju na kraju liste `L` (pozicija iza zadnjeg elementa)
`LiMakeNull(&L)`	. . .	pretvara listu `L` u praznu listu i vraća `LiEnd(L)`
`LiInsert(x,p,&L)`	. . .	ubacuje element `x` na poziciju `p` u listu `L`
`LiDelete(p,&L)`	. . .	izbacuje element na poziciji `p` iz liste `L`
`LiFirst(L)`	. . .	vraća prvu poziciju u listi `L` (ako je lista prazna, vraća `LiEnd(L)`)
`LiNext(p,L)`	. . .	vraća poziciju iza pozicije `p` u listi `L`
`LiPrevious(p,L)`	. . .	vraća poziciju ispred pozicije `p` u listi `L`
`LiRetrieve(p,L)`	. . .	vraća element na poziciji `p` u listi `L`

Zadatak 1.

Napišite potprogram `void Purge(List* L)` koji u listi `*L` eliminira elemente duplikate.

(a) Pretpostavite da nakon brisanja elementa na poziciji q procedura `LiDelete` na tu poziciju stavlja element koji je prije brisanja bio na poziciji iza te. U svemu ostalome potprogram treba biti neovisan o implementaciji a.t.p. List.

(b) Potprogram treba biti neovisan o implementaciji a.t.p. List.

Rješenje.

(a) Pretpostavili smo da se na poziciji q nakon brisanja nalazi sljedeći element u listi pa potprogram može izgledati ovako:

```
void Purge(List* L) {
    position p, q;
    p = LiFirst(*L);
    while (p != LiEnd(*L)) {
        q = LiNext(p, *L);
        while (q != LiEnd(*L))
            if (LiRetrieve(p, *L) == LiRetrieve(q, *L))
                LiDelete(q, L);
            else
                q = LiNext(q, *L);
        p = LiNext(p, *L);
    }
}
```

Ovaj potprogram bi funkcionirao za obje implementacije koje smo upoznali (pomoću polja i pomoću pointera), ali općenito nije potpuno neovisan o implementaciji jer bismo listu mogli implementirati i tako da `LiDelete` ne čini što smo pretpostavili.

Ovdje je `L` pointer na listu jer je deklariran kao `List* L`. (To možemo pisati i ovako: `List *L`, isto je značenje.) Lista na koju pokazuje pointer `L` je `*L`.

(b) Potprogram neovisan o implementaciji (uzimamo da ne znamo što `LiDelete` čini s pozicijama elemenata u listi) mogao bi biti ovaj ako bismo dopustili da se koristi pomoćna lista:

```
void Purge(List* L) {
    position p, q;
    int isduplicate;    // 1 true, 0 false
    List* H;        // pomocna lista *H
    // u listu *H ubacit cemo sad elemente liste *L bez ponavljanja elemenata
    p = LiFirst(*L);
    q = LiMakeNull(H);
    while (p != LiEnd(*L)) {
        isduplicate = 0;
        q = LiFirst(*H);
        while (q != LiEnd(*H) && isduplicate == 0)
            if (LiRetrieve(p, *L) == LiRetrieve(q, *H))
                isduplicate = 1;
            else
                q = LiNext(q, *H);
        if (isduplicate == 0)
            LiInsert(LiRetrieve(p, *L), LiEnd(*H), H);
        p = LiNext(p, *L);
    }
    p = LiMakeNull(L);      // prepisat cemo sada elemente liste *H u listu *L
    q = LiFirst(*H);
    while (q != LiEnd(*H)) {
        LiInsert(LiRetrieve(q, *H), LiEnd(*L), L);
        q = LiNext(q, *H);
    }
}
```

Razmislite kako biste napisali potprogram koji briše duplikate u listi neovisan o implementaciji a.t.p. List koristeći se samo jednom listom.

Pretpostavimo da je na skupu `elementtype` definiran neki uređaj $\leq$. Kažemo da je lista $(a_1, a_2, ..., a_n)$ *sortirana* ako vrijedi $a_1 \leq a_2 \leq ... \leq a_n$. Primijetimo da funkcija `LiInsert` općenito ne čuva sortiranost liste.

Zadatak 2.

Napišite program oblika `void SInsert(elementtype x, List* L)` kojim se u sortiranu listu `*L` ubacuje element `x` tako da lista ostane sortirana. Potprogram treba biti neovisan o implementaciji a.t.p. List.

Rješenje.

```
void SInsert(elementtype x, List* L) {
    position p;
    int found;    /* boolean varijabla, 1 je true, 0 je false */
    p = LiFirst(*L);
    found = 0;
```

```
    while (p != LiEnd(*L) && !found)
        if (x <= LiRetrieve(p, *L))
            found = 1;
        else
            p = LiNext(p, *L);
    LiInsert(x, p, L);
}
```

Prethodni program može služiti kao osnovni korak u algoritmu za sortiranje liste. Neka je L_1 polazna nesortirana lista. Stvaramo novu sortiranu listu L_2 tako da redom elemente iz L_1 ubacujemo u L_2 pomoću `SInsert`. Naprimjer, za $L_1 = (5, 3, 7, 1, 4, 3)$ postupak ide ovako: ...

Zadatak za DZ

Napišite ovaj algoritam za sortiranje kao potprogram neovisan o implementaciji a.t.p. List. Algoritam je poznat u literaturi pod nazivom Insertion Sort.

Zadatak 3.

Napišite potprogram oblika `void Merge(List L1, List L2, List* L3)` kojim se dvije sortirane liste `L1` i `L2` spajaju u novu sortiranu listu `*L3`. Potprogram mora biti neovisan o implementaciji a.t.p. List.

Rješenje.

```
void Merge(List L1, List L2, List* L3) {
    position p1, p2, p3;   /* tekuca pozicija u listi L1, L2, *L3 */
    p1 = LiFirst(L1);  p2 = LiFirst(L2);
    p3 = LiMakeNull(L3);
    while (p1 != LiEnd(L1) && p2 != LiEnd(L2)) {
        /* od elemenata na tekucim pozicijama biramo */
        /* onaj manji te ga prebacujemo u *L3        */
        if (LiRetrieve(p1, L1) <= LiRetrieve(p2, L2)) {
            LiInsert(LiRetrieve(p1, L1), p3, L3);
            p1 = LiNext(p1, L1);
        }
        else {
            LiInsert(LiRetrieve(p2, L2), p3, L3);
            p2 = LiNext(p2, L2);
        }
        p3 = LiNext(p3, *L3);
    }
    if (p1 == LiEnd(L1)) {
        /* L1 je potrosena, prebacimo ostatak L2 u *L3 */
        while (p2 != LiEnd(L2)) {
            LiInsert(LiRetrieve(p2, L2), p3, L3);
            p2 = LiNext(p2, L2);
            p3 = LiNext(p3, *L3);
        }
    }
    else {
        /* L2 je potrosena, prebacimo ostatak L1 u *L3 */
```

```
        while (p1 != LiEnd(L1)) {
            LiInsert(LiRetrieve(p1, L1), p3, L3);
            p1 = LiNext(p1, L1);
            p3 = LiNext(p3, *L3);
        }
    }
}
```

Prethodni potprogram može služiti kao osnovni korak u algoritmu za sortiranje. Polaznu nesortiranu listu L_1 razbijemo na (što veće) podliste koje su već sortirane. Zatim redom spajamo dvije po dvije podliste sve dok ne ostane samo jedna lista. Naprimjer, za $L_1 = (5, 3, 7, 1, 4, 3)$ postupak ide ovako:

Ovaj algoritam je poznat u literaturi pod nazivom MERGE SORT. Bilo koji redoslijed spajanja daje ispravni rezultat, ali vrijeme izvršavanja ovisi o redoslijedu spajanja podlista.

Na predavanju smo promatrali implementaciju liste pomoću pointera:

Lista je prikazana kao vezana lista ćelija. Svaka ćelija sadrži jedan element i pointer na sljedeću ćeliju. Također je dodana početna ćelija (header) koja ne sadrži element. Lista se poistovjećuje s pointerom na header. ... Pozicija elementa a_i je pointer na ćeliju koja sadrži pointer na a_i. To je malo neprirodno, ali služi da bi se efikasnije obavljale operacije Insert i Delete. Ova implementacija je direktno primjenjiva ako programski jezik u kojem radimo podržava pointere (npr. Pascal, C, C++, ...). No, ako programski jezik nema pointere (npr. FORTRAN, Algol) implementaciju treba modificirati tako da pointere smatramo kursorima.

Zadatak 4.

Razradite implementaciju liste pomoću kursora tako da u prethodnoj implementaciji pomoću pointera pointere zamijenite kursorima.

Rješenje. Slobodni prostor u memoriji računala zauzmemo jednim velikim poljem:

```
#define MAXLENGTH ...
struct celltype {
    elementtype element;
    int next;
} Space[MAXLENGTH];
```

Ovo polje predstavlja zalihu ćelija od kojih ćemo graditi liste. Svaka lista je prikazana vezanom listom ćelija, s time da su veze među ćelijama uspostavljene kursorima umjesto pointerima. Vezana lista je građena isto kao prije, dakle, postoji header i lista se poistovjećuje s kursorom na header. Sve liste s kojima radimo troše ćelije iz istog jedinstvenog polja Space. Naprimjer, za `MAXLENGTH 12` i za liste $L = (a, b, c)$, $M = (d, c)$ prikaz bi mogao izgledati ovako:

Sve slobodne ćelije koje nisu dio nijedne liste povezali smo u dodatnu vezanu listu (bez headera) koju zovemo available. Kad god nam treba nova ćelija, uzmemo ju s početka te liste. Kad obrišemo ćeliju u nekoj listi, vratimo ju na početak liste available.

Tipovi `List` i `position` iz a.t.p. List definirani su ovako:

```
typedef int List;
typedef int position;
```

Da bismo dovršili implementaciju, trebamo još napisati potprograme koji realiziraju operacije iz a.t.p. List. U tu svrhu je dovoljno prepisati potprograme iz implementacije pomoću pointera tako da pointere zamijenimo kursorima. To ćemo napraviti za `LiInsert`, za ostale je postupak sličan.

```
void LiInsert(elementtype x, position p) {
    /* nije potreban podatak o kojoj se listi radi jer je pozicija globalna */
    position tmp;
    if (available == -1)
        printf("Nema slobodnog prostora.");
    else {
        tmp = Space[p].next;
        Space[p].next = available;
        available = Space[available].next;
        Space[Space[p].next].element = x;
        Space[Space[p].next].next = tmp;
    }
}
```

Ovaj zadatak ilustrira općenitu ideju kako se bilo koja implementacija bazirana na pointerima može preraditi u implementaciju baziranu na kursorima. U ovom kolegiju izložit ćemo mnogo struktura podataka s pointerima. Iako to ne naglašavamo, podrazumijevamo da se iste strukture mogu realizirati i kursorima.