Programozói dokumentáció

Működés

A `main()` funkció a program összes lefutásakor betölti az adatokat, valamint a végén elmenti és felszabadítja azokat. A futtatáskor a felhasználó által megadott argumentumok alapján dől el, hogy milyen funkciók kerülnek majd meghívásra.

Ha a program először fut le, ignorálja a megadott extra argumentumokat és megkéri a felhasználót, hogy állítsa be az alap paramétereket (legalább egy asztal és egy menü elem).

Bármely lefutáskor ha nincs megadva argumentum, akkor a program menü üzemmódban fut addig, amíg a felhasználó le nem állítja (ez a funkció is elérhető a programban).

A program építőelemei

Input

A felhasználóval való kommunikációt segítő funkciók.

`void printWelcome()`

Üdvözli a felhasználót a program első lefutásakor, vagy a menü üzemmódban.

`MenuItem requestMenuItem()`

Létrehoz egy új `MenuItem` objektumot, majd beolvassa szükséges adatokat hozzá a konzolról (az étel neve és ára). A létrehozott objektumot visszaadja, így az később hozzáfűzhető egy láncolt listához.

`Table requestTable()`

Létrehoz egy új `Table` objektumot, majd beolvassa a szükséges adatokat hozzá a konzolról (asztal helye és kapacitása). A létrehozott objektumot visszaadja, így az később hozzáfűzhető egy láncolt listához.

`char *readString(char end)`

Dinamikus `string`-be olvas a konzolról, a megadott karakterig.

`char *readStringLine(char end)`

A `\n` karakterig olvas be a konzolról, dinamikus `string`-be.

Generikus láncolt lista

A programban külön modul foglalkozik a láncolt listák kezelésével. A generikus láncolt lista implementáció lehetővé teszi az asztalok, a menün szereplő ételek és a rendelések moduláris kezelését a C `void*` típusának használatával.

`ListItem` adatstruktúra

A struktúra leírja, hogy hogyan néz ki egy adott elem a láncolt listában. A *data* `void` típusú pointer tárolja a memóriacímet az adott lista elem értékéhez, míg a *next* `ListElem` típusú pointer a következő elemre mutat a listában. Amennyiben a következő elem nem létezik, az értéke `NULL`.

```
typedef struct ListItem {
    void *data;
    struct ListItem *next;
} ListItem;
```

`ListItem *push(ListItem *list, void *data)`

Hozzáad a láncolt lista végéhez egy új elemet, majd visszaadja a lista első elemére (*head*) mutató pointert. Ez azért fontos, mert ha a lista eddig üres volt, akkor ez az első elem maga a hozzáadott elem lesz, így a *head* változik.

> **Fontos:** Az új ListItem típusú pointert fel kell szabadítani, ha már nincs rá szükség. Ez a programozó felelőssége.

`ListItem *pop(ListItem *list, int index)`

Kiszed a láncolt listából egy elemet, majd visszaadja a lista első elemére (*head*) mutató pointert. Hasonlóan a `push()` funkcióhoz, a *head* változhat, ha az első elemet távolítjuk el, ezért fontos a visszatérési érték.

> **Megjegyzés:** A funkció automatikusan felszabadítja az eltávolított elem memóriáját.

`void freeList(ListItem *list)`

Amennyiben már nincsen szükség a lista egyik elemére sem, ez a funkció végigmegy rajta és felszabadítja az összes elemet, valamint az értékeit is.

Példa a generikus lista használatára

A program fontos része a generikus láncolt lista kezelő modul, mert három féle adattípust is ezzel tárolunk, így indokolt használatának példásítása.

Vegyünk egy egyszerű láncolt listát, ami `int` típusú értékeket tárol. Ezt a következő módon hozhatjuk létre:

```
// egy üres lista inicializálása
// ez a változó tartalmazza a lista "head"-jét
// (vagyis a lista első elemére mutató pointert)
ListItem *listanev = NULL;

// hozzuk létre az első elem értékét
// ez BÁRMILYEN típusú lehet
int elso = 34;

// hozzunk létre egy új lista elemet
listanev = (ListItem*) malloc(sizeof (ListItem));

// végül adjuk meg az értékét a korábban
// megadott értékre mutató pointerrel
listanev->data = &elso;

// ne felejtsük el felszabadítani a listát később
freeList(listanev);
```

A listához egyszerűbben is hozzáfűzhetünk új elemeket (akár a létrehozásnál/első elemnél is):

```
// hozzuk létre a második elem értékét
int masodik = 87;

// a lista végéhez fűzzük
listanev = push(listanev, &masodik);
```

```
// ne felejtsük el felszabadítani a listát később
freeList(listanev);
```

A listából természetesen törölhetünk is elemet:

```
// a masodik elem törlése
listanev = pop(listanev, 1);

// a lista maradékát így is fontos felszabadítani
freeList(listanev);
```

Menü

Az étterem menüjét kezelő funkciók.

`MenuItem` adatstruktúra

Leírja, hogy hogyan épül fel egy menün szereplő elem. Tartalmazza az integer típusú árat (Forintban), valamint a nevét, ami egy dinamikus `string` első karakterére mutató `char` pointer. Használható láncolt lista építésére a generikus láncolt lista modullal, ha egy ilyen adatstruktúrára mutató pointert adunk meg a lista értékére.

```
typedef struct MenuItem {
    char *name;
    int price;
} MenuItem;
```

`ListItem *loadMenu()`

A funkció betölti a már elmentett menü elemeket a `menu.txt` fájlból. Végigmegy a fájl összes során, dinamikus `string`-be olvassa azokat, majd azt feldolgozva beolvassa az étel árát és a nevét (szintén dinamikusan).

A funkció a generikus láncolt lista modulra épül, így a menüt is egy ilyen listába olvassa.

> **Fontos:** A visszaadott elemeket és a dinamikusan beolvasott étel neveket a funkció meghívójának saját felelőssége felszabadítani.

> Az utóbbira külön kell figyelni, ha eltávolítanak egy elemet a menü listából, hogy ne legyen memóriaszivárgás.

`void saveMenu(ListItem *list)`

Elmenti az adott menü láncolt lista elemeit a `menu.txt` fájlba. A lista összes elemét egy-egy sorba rakja, ahol az étel nevét és árát egy tabulátor választja el.

`void freeMenu(ListItem *list)`

Felszabadítja a menü láncolt lista elemeit, valamint a dinamikusan beolvasott étel nevét is.

Asztal

Az asztalokat kezelő funkciók.

`Table` adatstruktúra

Az adatstruktúra leírja az asztalokat tároló változók felépítését. Tárolja az asztal helyét az étteremben (x és y koordinátával), hogy hány ember fér el körülötte, valamint hogy foglalt-e. Használható láncolt lista építésére a generikus láncolt lista modullal, ha egy ilyen adatstruktúrára mutató pointert adunk meg a lista értékére.

```
typedef struct Table {
    int x, y;
    int capacity;
    bool occupied;
} Table;
```

`ListItem *loadTables()`

Betölti az asztalokat a `tables.txt` fájlból. Végigmegy a fájl összes során és beolvassa a struktúra összes adatát.

> **Fontos:** A visszaadott elemeket a funkció hívójának felelőssége felszabadítani.

`void saveTables(ListItem *list)`

Elmenti az asztalokat a `tables.txt` fájlba. Minden elemet a saját sorába ment, az értékeket tabulátorral választja el.

Rendelés

A rendeléseket kezelő funkciók.

TODO

Parancsok

A parancs üzemmódban megadott parancsokat kezelő funkciók.

TODO

GUI

A menü üzemmódot kezelő funkciók.

TODO

Debugmalloc

Debugoláshoz használt külső modul, ami segít feltárni az esetleges memóriaszivárgásokat. A hozzá tartozó dokumentációt a *header* fájl tartalmazza.