Zadanie 3.07

Aplikacja obsługująca zajezdnie autobusowe - przechowuje informacje na temat autobus w, zajezdni oraz relacji pomiędzy nimi. Umożliwia także manipulację tymi danymi.

Warstwa algorytmiczna

Wykorzystane struktury

Lista

Cały program bazuje na uniwersalnej, dwukierunkowej liście. Każdy jej węzeł przechowuje dowolny wskaźnik na miejsce w pamięci. Elementy w liście są unikalne i posortowane (względem podawanych przy dodawaniu element w funkcji por wnujących).

```
typedef struct List {
    unsigned int length;
    ListNode *head;
    ListNode *tail;
} List;

typedef struct ListNode {
    void *object;
    struct ListNode *next;
    struct ListNode *prev;
} ListNode;
```

Funkcje, kt re bezpośrednio na niej operują (w celu zachowania całkowitej uniwersalności), przyjmują jako argumenty wywołania wskaźniki na inne funkcje o określonych zadaniach (np. funkcja wyłuskująca z `ListNode` wskaźnik na zajezdnię czy funkca por wnująca numery boczne autobus w)

Autobus

Dane o konkretnym autobusie znajdują się w pojedyńczej, statycznej strukturze. Pole numeru bocznego jest także wyr żnikiem każdego z autobus w (w jednej liście nie mogą pojawić się 2 takie same numery boczne). W polu `memberships` znajduje się lista zawierająca wskaźniki na zajezdnie, do kt rych należy dany autobus.

```
typedef struct Bus {
    int side_no;
    int line_no;
    char driver_name[64];
    char driver_pesel[12];
    List memberships;
} Bus;
```

Autor: Tomasz Gałecki

Zajeznia

Kolejna statyczna struktura, kt ra w pełni opisuje pojedyńczą zajeznię. Nazwa zajezni jest jej wyr żnikiem - musi być unikalna. Lista `members` zawiera wskaźniki na autobusy, kt re przynależą do danej zajezdni.

```
typedef struct Depot {
    char name[64];
    List members;
} Depot;
```

Spos b realizacji wymagań dot. operacji na danych

Dodawanie, modyfikacja i usuwanie

Wszystkie dane są przechowywane w najmniejszej możliwej ilości egzemplarzy. Dodawanie nowych rekord w opiera się na pojedyńczym zaalokowaniu miejsca dla dodawanej struktury, a następnie zaalokowaniu miejsca na sam węzeł listy. Dany rekord zostaje w pamięci, a wskaźnik do niego jest cały czas dostępny. Przy modyfikacji p l, kt re są kluczami (numer boczny, nazwa zajezdni) następuje automatyczne przesortowanie listy tak, aby zachować w niej porządek rosnący. Podczas usuwania rekordu zostają także usunięte wszelkie do niego dowiązania, a sama pamięć od razu zwalniana.

Zarządzanie przypisaniami

Każdy autobus i zajeznia zawiera w sobie listę ze wskaźnikami przypisań (np. każdy autobus posiada w liście wskaźniki na zajezdnie, do kt rych zależy). Dodanie nowego dowiązania ogranicza się do dodania wskaźnika na konkretną strukturę do odpowiedniej listy zależności. Wszelkie mechanizmy list są dokładnie takie same, jak przy każdej innej strukturze. R żnicą są jedynie małe funkcje, kt re determinują inne zachowania tych niskopoziomowych działań (np. funkcja `del_node_only()` nie robi absolutnie nic. Ale dzięki temu przy usuwaniu referencji nie zostanie usunięty konkretny reprezentant struktury).

Filtrowanie

Filtrowanie bazuje na niskopoziomomwej funcji `find_occurrences(List*, void*, void *(ListNode*), int (const void*, const void*))`, kt ra zwraca zmienną `List` (czyli de facto wskaźnik na koniec i początek listy oraz jej długość). Przeszukuje ona listę podaną jako pierwszy argument według kryterium z drugiego parametru. Oczekuje także funkcji wyłuskującej (np. `get_side_no(ListNode*)`) oraz funkcji por wnującej (np. `side_no_cmp(void*, void*)`). Następnie, jeśli obiekt z danego węzła listy z argumentu spełnia podane kryteria, wskaźnik na niego zostaje dołączony do statycznej listy wewnątrz funkcji. Po iteracji na całej liście wejściowej, lista z elementami spełniającymi kryterium zostaje zwr cona.

Autor: Tomasz Gałecki

Zapis danych do pliku

Dane mogą zostać zapisane i odczytane z/do pliku `Dane.txt`. Funkcja przeprowadzająca zrzut bazy danych (list z autobusami i zajezdniami) dołącza także kilka komentarzy do teogoż pliku (kt re zostaną p źniej pominięte przy parsowaniu). Wszystkie dane zostają zapisane bez jakiejkolwiek redundancji. Po pierwsze - zostają zapisane same własności autobus w (numery boczne, imię kierowcy etc.). Następnie zapisane zostają zajezdnie wraz z przypisanymi do nich autobusami. Przy parsowaniu i konstrukcji bazy danych z pliku dowiązania te zostają automatycznie wykonane w obydwie strony (czyli każdy autobus będzie także zawierał informację o tym, że jest przypisany do konkretnej zajezdni).

Testy

Kod projektu zawiera pliki `tests`, w kt rych zawarte zostały bardzo prymitywne testy wszystkich funkcjonalności używanych w programie. Ich wykonanie nie wymaga ingerencji użytkownika, więc w naturalny spos b były bardzo pomocne przy testowaniu programu narzędziem do kontroli wyciek w pamięci (VALGRIND). Program nie posiada jakichkolwiek miejsc, w kt rych mogłoby dojść do wycieku pamięci albo nieautoryzowanego odwołania się do kom rek spoza sterty.

Dokumentacja funkcji

list.h

`int append_to(List* a_list, void* an_object, int (*cmp)(const void*, const void*))`: dodaje obiekt (np. autobus) do listy w odpowiednim miejscu (por wnywanie następuje przy pomocy funkcji podanej jako 3 argument). Domyślny porządek jest rosnący, choć może zostać odwr cony przez manipulację na funkcji por wnującej.

`int remove_from(List* a_list, void* an_element, void (*rm)(void*))`: usuwa obiekt (np. autobus) z listy. Wymaga także dekonstruktora danej struktury (np. funkcji usuwającej dowiązania do zajezdni oraz samą listę z zależnościami).

`ListNode *find_node_in(List* a_list, void* an_object)`: wyszukuje obiekt (np. zajeznię jako strukturę) w danej liście, a nastepnie zwraca wskaźnik węzła, w kt rym się on znajduje. Jeśli elementu nie znaleziono - zwraca NULL.

`void *find_object_with_item_in(List* the_list, void* item, void *(*get)(ListNode*), int (*cmp)(void*, void*))`: funkcja zwracająca wskaźnik na strukturę, kt ra zawiera podany element (*item*) we wskazanej liście. Wymaga funkcji wyłuskującej (np. wyłuskanie *side_no* z *ListNode*) oraz funkcji por wnującej (np. `side_no_cmp(const void*, const void*)`).

Autor: Tomasz Gałecki

`void delete_list(List* the_list, void (*rm_content)(void*))`: usuwa wszystkie węzły listy wraz z elementami wewnątrz każdego węzła (poprzez funkcję przekazywaną przez wskaźnik).

`void del_node_only(void* the_node_ptr)`: funkcja 'pusta'; podana jako argument np. `remove_from()` nie spowoduje usunięcia obietk w wewnątrz węzła (używana np. przy usuwaniu przypisań autobus w do zajezdni).

`List find_occurrences(List* the_list, void* item, void* (*get)(ListNode*), int (*cmp)(const void*, const void*))`: funkcja przeszukująca podaną listę względem zawierania (określanego przy pomocy funkcji `*cmp`) podanego kryterium (`void *item`). Jeśli jakiś element listy spełnia to kryterium, to zostaje dodany do statycznej listy, kt ra potem zostanie zwr cona. Wymaga dodatkowo funkcji wyłuskującej z `ListNode` analogicznego elementu (np. przy filtrowaniu po numerze linii - wymagana jest funkcja `get_line_no(ListNode*)`.

`void for_each_in(List* the_list, void (*do_sth)(void*))`: prymitywna implementacja operatora `for_each`, kt ra wykonuje jakąś operację na elemencie we wszystkich węzłach (używana np. do funkcji wyświetlających zawartość listy albo do usuwania nieistniejących już dowiązań).

comparing.h

`int buses_side_no_cmp(const void *first, const void *second)`: por wnuje numery boczne dw ch autobus w.

`int buses_line_no_cmp(const void *first, const void *second)`: por wnuje numery linii dw ch autobus w.

`int buses_drivers_names_cmp(const void* first, const void* second)`: por wnuje nazwiska kierowc w dw ch autobus w.

`int buses_drivers_pesels_cmp(const void* first, const void* second)`: por wnuje pesele kierowc w dw ch autobus w.

`int depots_names_cmp(const void* first, const void* second)`: por wnuje nazwy dw ch zajezdni.

extracting.h

`void *get_side_no(ListNode* the_node)`: wyłuskuje numer boczny z węzła listy.

`void *get_line_no(ListNode* the_node)`: wyłuskuje numer linii z węzła listy.

`void *get_driver_name(ListNode* the_node)`: wyłuskuje nazwisko kierowcy z węzła listy.

`void *get_driver_pesel(ListNode* the_node)`: wyłuskuje pesel kierowcy z węzła listy.

`void *get_depot_name(ListNode* the_node)`: wyłuskuje nazwę zajezdni z węzła listy.

bus.h

`Bus *new_bus(char* side_no, char* line_no, char* name, char* driver_pesel)`: konstruktor nowej instancji struktury. Przy ustawianiu konkretnych wartości następuje od razu ich sprawdzenie, zaś niespełnienie jakiegokolwiek kryterium skutkuje cofnięciem alokacji pamięci.

`int set_side_no(Bus* a_bus, char* side_no)`: ustawia numer boczny po uprzedniej walidacji.

`int set_line_no(Bus* a_bus, char* line_no)`: ustawia numer linii po uprzedniej walidacji.

`int set_driver_name(Bus* a_bus, char* name)`: ustawia nazwisko i imię kierowcy po uprzedniej walidacji.

`int set_driver_pesel(Bus* a_bus, char* driver_pesel)`: ustawia pesel kierowcy po uprzedniej walidacji.

`void print_bus_info(void* the_bus_ptr)`: wyświetla informacje o pojedyńczym autobusie w formie skr conej (bez przypisań).

`void print_bus_info_with_refs(void* the_bus_ptr)`: wyświetla informacje o pojedyńczym autobusie razem z przypisaniami do zajezdni.

`void print_bus_labels()`: wyświetla linię nagł wkową dla listy autobus w.

`void del_bus(void* the_bus_pointer)`: dekonstruktor pojedyńczego autobusu - usuwa listę dowiązań oraz sam autobus (wraz ze zwolnieniem pamięci).

depot.h

`Depot *new_depot(char* a_string)`: alokuje pamięć dla nowej zajezdni, ustawia jej nazwę i zwraca do niej wskaźnik.

`void del_depot(void* the_depot_pointer)`: dekonstruuje zajezdnię - usuwa listę dowiązań i samą zajezdnię.

`int set_depot_name(Depot* a_depot, char* name)`: ustawia nazwę zajezdni uprzednio ją walidując.

`void print_depot_info(void* the_depot_pointer)`: wyświetla informacje o pojedyńczej

zajezdni w skr conej formie (bez dowiązań).

`void print_depot_info_with_refs(void* the_depot_pointer)`: wyświetla informacje o pojedyńczej zajezdni razem z przypisanymi autobusami.

buses_management.h

`int add_bus(char* side_no, char* line_no, char* name, char* pesel)`: dodaje autobus o podanych parametrach do listy.

`int remove_bus(Bus* the_bus)`: usuwa wszystkie dowiązania autobusu w zajezdniach, a następnie sam autobus.

`void reappend_bus_memberships(Bus* the_bus)`: usuwa i na nowo tworzy przypisania autobusu we wszystkich zajezdniach. Ma to na celu naprawienie kolejności rekord w po edycji wartości klucza (po edycji numeru bocznego).

depots_management.h

`int add_depot(char* depot_name)`: dodaje zajezdnię o podanej nazwie do listy.

`int remove_depot(Depot* the_depot)`: usuwa zajezdnię wraz z miejscami, gdzie występowała.

`void reappend_depot_assignments(Depot* the_depot)`: usuwa i tworzy na nowo dowiązania zajezdni w przypisanych autobusach. Naprawia ich kolejność (po edycji nazwy zajezdni).

membership_management.h

`void assign_to(char* depot_name, int side_no)`: przypisuje zajezdnię o podanej nazwie do autobusu o podanym numerze bocznym.

`void assign_structs_to(Depot* the_depot, Bus* the_bus)`: analogicznie jak przy powyższej, tylko że wymaga już wskaźnik w na struktury.

`void remove_assignment_from(char* depot_name, int side_no)`: usuwa przypisanie autobusu o podanym numerze bocznym z zajezdni o podanej nazwie.

`void remove_assignment_structs_from(Depot* the_depot, Bus* the_bus)`: analogicznie jak powyżej, ale wymaga wskaźnik w na struktury.

`void move_to(char *from_depot_name, int side_no, char *to_depot_name)`: przenosi przypisanie autobusu o podanej nazwie z jednej zajezdni do drugiej.

filtering.h

`void print_filtered_by(enum FilterType filter_type, void* value)`: filtruje dane po konkretnym polu (podane przez enum) wykorzystując podane kryterium *value* i wypisuje je na wyjściu.

dumper.h

`int dump_database_to(const char* filename)`: zapisuje zrzut bazy danych do pliku o nazwie podanej w parametrze oraz tworzy backup poprzednich danych (z poprzedniego zapisu).

data_loader.h

`void load_database_from(const char* filename)`: wczytuje z pliku o podanej nazwie dane i tworzy z nich bazę danych, do kt rej dostęp będzie miał program.

cleanup.h

`void clean_up_mem()`: czyści całą bazę danych usuwając wszystkie zaalokowane wcześniej struktury. Nie pozostawia jakichkolwiek danych w pamięci.

tests.h

`void do_tests()`: wykonuje prymitywne testy wszystkich funkcjonalności programu, wyświetlając stan bazy danych po każdej jej edycji. Aby uruchomić program w trybie debugowania wystarczy zmienić flagę `DEBUG_MODE` w pliku `main.c` na 1.

messages.h

`void msg(enum Message message)`: switch operujący na enumie; zawiera wszystkie komunikaty błęd w, kt re mogą wystąpić w programie.

ui_printer.h

`void prt(enum PrintType prt_type)`: jeden wielki switch operujący na enumie. Zawiera wszystkie komunikaty używane w interfejsie użytkownika.

ui.h

`void start_program()`: procedura odpowiedzialna za obsługę całego programu. Opiera się na jednej pętli zdarzeń, kt re mogą wywoływać kolejne (pozostałe podmenu, zapis danych etc.). Dba także, aby po jakiejkolwiek edycji danych zapytać użytkownika przed wyjściem, czy chciałby zapisać dane do pliku.

Autor: Tomasz Gałecki

Autor: Tomasz Gałecki