Algorytmy 2
Laboratorium: lista

Przemysław Klęsk

1 października 2021

1 Cel

Celem zadania jest wykonanie implementacji struktury danych nazywanej *listą* lub *listą z dowiązaniami* (ang. *linked list*), będącej jedną z najbardziej podstawowych liniowych struktur danych. Obrany zostanie wariant listy dwukierunkowej. Oznacza to, że każdy element (węzeł) listy będzie zawierał trzy informacje:

- dane właściwe (rekord, obiekt) lub wskaźnik na nie,
- wskaźnik na element poprzedni,
- wskaźnik na element następny.

Sama zaś lista będzie przechowywała wskaźniki na pierwszy i ostatni element listy (tzw. głowę i ogon) oraz swój aktualny rozmiar (liczbę elementów).

Dodatkowym celem zadania jest wykonanie odpowiednich pomiarów czasowych pozwalających na sprawdzenie złożoności obliczeniowej takich operacji na liście jak: dodawanie, wyszukiwanie oraz usuwanie (przy coraz większej liczbie przechowywanych elementów). W ogólności, listy (w przeciwieństwie do tablic) nie zajmują ciągłego obszaru w pamięci i oferują sekwencyjny dostęp do danych. Tzn. operacje wymagające wędrówki po wskaźnikach mają liniową złożoność obliczeniową — czas dostępu do elementu o losowym indeksie i czas wyszukiwania elementu o pewnym kluczu są rzędu $\Theta(n)$, gdzie n oznacza rozmiar listy. Operacje dodawania i usuwania na krańcach listy mają zaś złożoność stałą — $O(1)$. Dodatkowo, w implementacji zbadany ma zostać również wariant dodawania elementów do listy z wymuszeniem porządku, co spowoduje liniową złożoność obliczeniową dodawania.

2 Instrukcje, wskazówki, podpowiedzi

1. Według własnych preferencji programistycznych implementację można wykonać strukturalnie lub obiektowo, przy czym niezbędne jest, aby wykorzystać mechanizm **szablonów** (`template`) dostępny w C++. Dzięki temu uzyska się ogólność i każda powołana w przyszłości do życia lista będzie mogła przechowywać dane dowolnego typu (ustalonego w chwili instancjonowania listy). Uwaga: możliwe, że lista jako kontener przyda się w późniejszych zadaniach laboratoryjnych (np. jako lista sąsiedztwa

do reprezentacji krawędzi w grafach, do sortowania kubełkowego, w tablicach mieszających do przechowywania wpisów z kolizją kluczy, itp.).

Na potrzeby niniejszego zadania chcemy, aby w uruchamianym programie na liście jako dane przechowywane były struktury zawierające przynajmniej dwa pola typów prostych. Poniżej pokazano poglądowy przykład, w którym `linked_list` oznacza typ (strukturę lub klasę) zaimplementowany przez studenta, reprezentujący listę.

```
...
struct some_object
{
    int field_1;
    char field_2;
};
...

int main()
{
...
linked_list<some_object>* ll = new linked_list<some_object>();
some_object so;
so.field_1 = 123;
so.field_2 = 'a';
ll->add(so);
...
delete ll;
...
}
```

W ogólności możliwe ma być powoływanie do życia list z dowolnym typem `T` dla danych właściwych wg mechanizmu szablonu tj.: `linked_list<T>* ll = new linked_list<T>();`. Warto tu podkreślić, że dane mogą być przechowywane bezpośrednio w węzłach listy lub poprzez wskaźnik, tj. implementacja powinna umożliwiać w szczególności zapisanie obu poniższych linii:

```
linked_list<some_object>* ll1 = new linked_list<some_object>();
linked_list<some_object*>* ll2 = new linked_list<some_object*>();
```

Ten drugi przypadek będzie spotykany często w bardziej zaawansowanych zadaniach, np. gdy raz stworzone dynamicznie obiekty będą przechowywane poprzez wskaźniki równocześnie w kilku kontenerach (np. w drzewie i na liście).

2. **Interfejs listy** powinien udostępniać następujące funkcje / metody:

   (a) dodanie nowego elementu na końcu listy (argument: dane),

   (b) dodanie nowego elementu na początku listy (argument: dane),

   (c) usunięcie ostatniego elementu,

   (d) usunięcie pierwszego elementu,

   (e) zwrócenie danych i-tego elementu listy (argument: indeks i żądanego elementu (numerując od zera); wynik: dane i-tego elementu lub niepowodzenie w razie indeksu poza zakresem),

(f) ustawienie (podmiana) danych i-tego elementu listy (argumenty: indeks i żądanego elementu (numerując od zera) oraz nowe dane; wynik: pusty lub niepowodzenie w razie indeksu poza zakresem),

(g) wyszukanie elementu (argumenty: dane wzorcowe oraz informacja lub komparator definiujące klucz wyszukiwania — szczegółowe wskazówki dalej; wynik: wskaźnik na odnaleziony element listy lub `NULL` w przypadku niepowodzenia),

(h) wyszukanie i usunięcie elementu (argumenty: jak wyżej; wynik: flaga logiczna sygnalizująca powodzenie lub niepowodzenie),

(i) dodanie nowego elementu z wymuszeniem porządku (argumenty: dane i informacja lub komparator definiujące klucz porządkowania),

(j) czyszczenie listy tj. usunięcie wszystkich elementów,

(k) zwrócenie napisowej reprezentacji listy — np. funkcja / metoda `to_string(...)` (format wyniku wg uznania programisty, może zawierać np. rozmiar listy, wypis pewnej liczby elementów początkowych / końcowych, opcjonalnie adres listy w pamięci; argumenty: również wg uznania programisty — np. liczba elementów do wypisania, wskaźnik na funkcję wypisującą pojedynczy rekord / obiekt danych).

Uwaga Poprzez „dane" w powyższych zapisach „argument: dane" można w szczególności rozumieć „wskaźnik na dane".

3. W programie należy odpowiednio zarządzać dynamicznie pamięcią (`new`, `delete`). Za uwalnianie pamięci związanej z samymi elementami (węzłami) listy powinna być na pewno odpowiedzialna sama lista. Natomiast trzeba rozważyć i wybrać, które miejsce w programie będzie odpowiedzialne za uwalnianie pamięci związanej z danymi zasilającymi listę, gdy te przechowywane są poprzez wskaźniki. W takim przypadku dane są zwykle powoływane do życia gdzieś „na zewnątrz" (np. w pętli w funkcji `main()`) i często przyjmuje się, że także te zewnętrzne miejsca są odpowiedzialne za uwalnianie pamięci. Jeżeli jednak planujemy, że lista będzie jedyną strukturą przechowującą nasze dane, to dobrze jest przewidzieć możliwość zwalaniania pamięci danych także na poziomie listy. Możliwym rozwiązaniem jest przeciążenie funkcji / metod związanych z usuwaniem (lub nadpisywaniem) elementów listy (c, d, f, h, j) poprzez wprowadzenie dodatkowego argumentu logicznego, który będzie wskazywał, czy to lista ma uwolnić pamięć danych związanych z aktualnie usuwanymi elementami. Zarówno w przypadku implementacji obiektowej jak i strukturalnej wygodne może tu być także zdefiniowanie odpowiednich konstruktorów i destruktorów.

4. Implementację samej listy wraz z jej interfejsem można (wg uznania) wydzielić do własnego pliku nagłówkowego `.h`. Plik `.cpp` zawierałby wówczas zasadniczy program — funkcja `main()` — i ewentualne potrzebne dodatkowe funkcje i typy.

5. Do pracy z napisami (np. przy sklejaniu napisu z zawartością listy) wygodne może być użycie typu `std::string` po wykonaniu `#include <string>`.

6. Do pomiarów czasowych wygodne może być dołączenie: `#include <time.h>`, udostępniające funkcję `clock()` oraz typ `clock_t`. Pomiar czasu w sekundach pewnego fragmentu programu przeprowadza się wówczas w następujący sposób:

```
clock_t t1 = clock();
// tu pewne czasochlonne zadanie ...
clock_t t2 = clock();
double time = (t2 - t1) / (double)CLOCKS_PER_SEC;
```

   W realizowanym programie może zachodzić potrzeba przeliczeń na mili lub mikrosekundy.

7. W interfejsie w ramach punktów (e) i (f) można rozważyć przeciążenie operatora indeksowania — `operator []` — dodatkowo lub zamiast funkcji `get(...)`, `set(...)`.

8. W interfejsie w punktach (g), (h), (i) jest mowa o pewnej informacji definiującej klucz do wyszukiwania na liście lub porządkowania. Chodzi tu o to, że w ogólności programista nie musi wiedzieć z góry, jaki typ danych będzie przechowywany na liście i co będzie stanowiło takowy klucz. Gdyby przechowywane były np. rekordy z danymi osobowymi, kluczem mogłoby być np. pole PESEL lub pole nazwisko, itp. Co więcej, w przypadku „remisu kluczy", należy zwykle przewidzieć pewną kombinację pól rozstrzygającą o „równości" (lub porządku) dwóch obiektów.

   Od strony programistycznej możliwe są tu przynajmniej trzy rozwiązania: (1) przewidzenie pewnej funkcji o konkretnej ustalonej nazwie do porównywania dwóch elementów, (2) dostarczenie wskaźnika na taką funkcję, (3) przeciążenie operatora `<=` (lub operatorów `<` i `==`). W przypadku (2) funkcja porównująca może zwracać wynik typu `int`, będący: zerem, gdy dwa obiekty są sobie „równe" w pożądanym sensie; liczbą ujemną, gdy obiekt pierwszy jest „mniejszy" od drugiego; liczbą dodatnią w przeciwnym razie. Tego typu mechanizm jest stosowany w wielu językach programowania. Nawiązując do wcześniejszego przykładu, funkcja (komparator) porównująca dwie zmienne typu strukturalnego (z dwoma polami prostymi) mogłaby wyglądać następująco:

```
int some_objects_cmp(some_object* so1, some_object* so2)
{
    int diff = so1->field_1 - so2->field_1;
    if (diff != 0)
        return diff;
    return so1->field_2 - so2->field_2;
}
```

   Szablonowy wskaźnik na tego rodzaju funkcje mógłby być zapisany jako `int (*data_cmp)(T, T)` i używany w odpowiednich miejscach interfejsu listy.

9. Funkcja / metoda zwracająca napisową reprezentację listy (np. `to_string(...)`) może wywoływać podrzędne funkcje `to_string(...)` właściwe przechowywanym obiektom. Tutaj również możliwym rozwiązaniem (ale nie jedynym) jest wskaźnik na funkcję.

3 Zawartość funkcji `main()`

Główny eksperyment zawarty w funkcji `main()` ma polegać na wielokrotnym dodawaniu coraz większej liczby elementów (danych) do listy, a następnie wyszukiwaniu (i usuwaniu) z niej pewnych losowych danych. Towarzyszyć mają temu pomiary czasowe.

Na potrzeby eksperymentu wygodne będzie stworzenie pomocniczej funkcji generującej pojedynczy obiekt z losowymi danymi — wartość pola typu `int` może być losowana np. ze zbioru $\{0, \ldots, 10\,000\}$, a pola `char` np. ze zbioru $\{$`'a'`$, \ldots,$`'z'`$\}$. Do losowania można wykorzystać funkcję `rand()` (zapoznaj się także z funkcją `srand(...)`). Dla uproszczenia dopuścimy możliwość powtórzeń na liście obiektów z taką samą zawartością.

Poniższy listing pokazuje schemat eksperymentu (proszę go traktować poglądowo):

```
int main()
{
    const int MAX_ORDER = 6; // maksymalny rzad wielkosci rozmiaru dodawanych danych
    linked_list<some_object*>* ll = new linked_list<some_object*>(); // stworzenie listy
    for (int o = 1; o <= MAX_ORDER; o++) // petla po kolejnych rzedach wielkosci
    {
        const int n = pow(10, o); // rozmiar danych

        // dodawanie do listy
        clock_t t1 = clock();
        for (int i = 0; i < n; i++) {
            some_object* so = ... // losowe dane
            ll->add(so);
        }
        clock_t t2 = clock();
        ... // wypis na ekran aktualnej postaci listy (skrotowej) i pomiarow czasowych

        // wyszukiwanie i usuwanie z listy
        const int m = pow(10, 4); // liczba prob wyszukania
        t1 = clock();
        for (int i = 0; i < m; i++) {
            some_object* so = ... // losowe dane jako wzorzec do wyszukiwania (obiekt chwilowy)
            ll->find_and_remove(so, some_objects_cmp);
            delete so;
        }
        t2 = clock();
        ... // wypis na ekran aktualnej postaci listy (skrotowej) i pomiarow czasowych

        ll->clear(true); // czyszczenie listy wraz z uwalnianiem pamieci danych
    }
    delete ll;
    return 0;
}
```

Uwaga 1 Należy raportować czas całkowity i średni w przeliczeniu na pojedynczą operację.

Uwaga 2 Podczas oddawania programu student musi być w stanie swobonie zmienić zawartość funkcji `main()` zgodnie z życzeniami prowadzącego (w szczególności np. zmienić nastawy / warunki eksperymentu lub powołać do życia nową małą listę i wykonać na niej pewne żądane operacje, itp.). W związku z

powyższym należy dobrze przetestować działanie wszystkich zaimplementowanych funkcji interfejsu lista (mimo, że niektóre z nich nie występują w powyższym schemacie eksperymentu).

4 Sprawdzenie antyplagiatowe — przygotowanie wiadomości e-mail do wysłania

1. Kod źródłowy programu po sprawdzeniu przez prowadzącego zajęcia laboratoryjne musi zostać przesłany na adres *algo2@zut.edu.pl*.

2. Plik z kodem źródłowym musi mieć nazwę wg schematu: `nr_albumu.algo2.nr_lab.main.c` (plik może mieć rozszerzenie `.c` lub `.cpp`). Przykład: `123456.algo2.lab06.main.c` (szóste zadanie laboratoryjne studenta o numerze albumu 123456). Jeżeli kod źródłowy programu składa się z wielu plików, to należy stworzyć jeden plik, umieszczając w nim kody wszystkich plików składowych.

3. Plik musi zostać wysłany z poczty ZUT (zut.edu.pl).

4. Temat maila musi mieć postać: `ALGO2 IS1 XXXY LAB06`, gdzie `XXXY` to numer grupy (np. `ALGO2 IS1 210C LAB06`).

5. W pierwszych trzech liniach pliku z kodem źródłowym w komentarzach muszą znaleźć się:

   - informacja identyczna z zamieszczoną w temacie maila (linia 1),
   - imię i nazwisko autora (linia 2),
   - adres e-mail (linia 3).

6. Mail nie może zawierać żadnej treści (tylko załącznik).

7. W razie wykrycia plagiatu, wszytkie uwikłane osoby otrzymają za dane zadanie ocenę 0 punktów (co jest gorsze niż ocena 2 w skali $\{2, 3, 3.5, 4, 4.5, 5\}$).