ZAAWANSOWANE PROGRAMOWANIE KOMPUTEROWE WNE (2022)

KRZYSZTOF ZIEMIAŃSKI

8. Przechowywanie danych

Większość programów potrzebuje przechowywać pewne dane. Zazwyczaj są to zestawy informacji tego samego typu (rekordy), np. gdy przechowujemy informacje o osobach, zapewne pamiętamy ich imię i nazwisko (i być może inne informacje). Wygodnie jest, kiedy każdy rekord ma klucz: pewną wartość unikalną dla całego zestawu danych; w przypadku osób tą rolę może pełnić numer PESEL. Ważne jest, aby wartości klucza można było porównywać (tj. stwierdzić, która wartość jest większa od drugiej). Na tych zajęciach porównamy efektywność kilku sposobów przechowywania danych; dla uproszczenia będziemy przechowywać tylko klucze będące liczbami całkowitymi. Będziemy rozważać następujące struktury danych:

- tablice uporządkowane (żądamy, aby wartości były przechowywane w kolejności rosnącej)
- tablice nieuporządkowane (dowolna kolejność)
- listy jednokierunkowe (uporządkowane)
- listy dwukierunkowe (uporządkowane).

Dla każdej z nich dopuszczamy następujące operacje

- dodanie elementu (D),
- usunięcie elementu (U),
- znalezienie elementu (Z),

Każdą z nich roważamy w trzech wariantach dodanie/usunięcie/znalezienie dowolnego elementu (D/U/Z), dodanie/usunięcie/znalezienie maksymalnego elementu (Dx/Ux/Mx) i minimalnego (Dn/Un/Zn). Przeanalizujemy złożoność tych operacji, tj. liczbę kroków potrzebnych do ich wykonania. Np.

- $O(n)$ (złożoność liniowa) oznacza, że liczba kroków jest proporcjonalna do liczby rekordów w tablicy/na liście,
- $O(1)$ (złożoność stała) oznacza, że liczba kroków jest proporcjonalna do 1, tzn. stała i niezależna od liczby rekordów,
- $O(n^2)$ (złożoność kwadratowa) oznacza, że liczba kroków jest proporcjonalna do kwadratu liczby rekordów (co się szczęśliwie nie zdarzy na tych zajęciach), itd.

8.1. **Tablice nieuporządkowane.** Tablice pamiętamy jako wektory

```
vector<int> t;
```

Dodanie nowego elementu `k` realizujemy przez `t.push_back(k);` przy czy nie ma tu znaczenia, czy dodajemy element dowolny, minimalny czy maksymalny. Złożoność tej operacji, tj. czas potrzebny do jej wykonania, jest zazwyczaj stały $O(1)$; jeśli jednak nie ma wolnych miejsc w tablicy musimy zarezerwować nową tablicę i przepisać elementy, więc pesymistycznie $O(n)$, gdzie n jest liczbą elementów w tablicy.

Znalezienie elementu jest proste, ale czasochłonne: musimy przeszukać całą tablicę. Ponownie nie ma znaczenia, czy szukamy dowolnego elementu, czy skrajnego. Usuwanie elementów jest szybkie, wystarczy usuwany element nadpisać ostatnim elementem tablicy. Trzeba jednak usuwany element najpierw odnaleźć, a to powoduje, że czas potrzebny do usunięcia elementu zależy liniowo od liczby elementów w tablicy. Wszystkie operacje znajdowania i usuwania mają złożoność $O(n)$.

8.2. **Tablice uporządkowane.** Również używamy wektorów. Dodanie maksymalnego elementu jest szybkie (podobnie jak dla tablic nieuporządkowanych: zazwyczaj $O(1)$, pesymistycznie $O(n)$). Jeśli jednak dodajemy dowolny bądź minimalny element, nie możemy go umieścić na końcu, gdyż zaburzyłoby to porządek. Nowy element wstawiamy w środku (dla dowolnego) lub na początku (dla minimalnego) i musimy przepisać średnio połowę lub wszystkie elementy tablicy. Złożoność wynosi więc $O(n)$. Podobnie jest w przypadku usuwania elementów: największy możemy usunąć w stałym czasie $O(1)$, w pozostałych przypadkach musimy przepisywać duży fragment tablicy, co powoduje złożoność liniową.

Korzyść w przypadku stosowania uporządkowanych tablic odnosimy w przypadku znajdowania elementów. Najmniejszy element to pierwszy element tablicy, największy to ostatni, a więc możemy je znaleźć w stałym czasie. Nawet jeśli szukamy dowolnego elementu, możemy to zrobić szybciej niż w czasie liniowym przy użyciu algorytmu zwanego *wyszukiwaniem binarnym*. Żeby znaleźć element `w` w posortowanej (rosnąco) tablicy `t` porównujemy go ze środkowym elementem tablicy `t[m]`:

- Jeśli `t[m]==w`, to element został znaleziony.
- Jeśli `t[m]>w`, to szukamy w pierwszej połowie tablicy.
- Jeśli `t[m]<w`, to szukamy w drugiej połowie tablicy.

Poszukiwania kończymy gdy znajdziemy element lub wykluczymy wszystkie elementy (tj. zawięzimy obszar poszukiwań do podciągu tablicy o długości 0). Poniżej funkcja (iteracyjna) wyszukująca binarnie element w tablicy posortowanej rosnąco.

```
bool znajdz(vector<int>& t, int w) {
    int a=0;
    int b=t.size(); // szukamy w przedziale [a,b)
    while(a<b) {
        int m=(a+b)/2; // środkowy element
        if(t[m]==w)
            return true;
        if(t[m]<w) // zawężamy przedział
            a=m+1;
```

```
            else
                b=m;
        }
        return false;
    }
```

Oszacujemy teraz złożoność wyszukiwania binarnego. Liczba operacji potrzebnych do wyszukania elementu w tablicy o długości n dana jest wzorem

$$F(n) = \begin{cases} 1 & \text{dla } n = 1 \\ 1 + F(n/2) & \text{dla } n > 1, \end{cases}$$

a więc złożoność jest rzędu $O(\log n)$ — istotnie mniej niż w przypadku tablic nieposortowanych.

8.3. **Listy.** Na listach jednokierunkowych operacje dodawania, usuwania i znajdowania można wykonać w czasie stałym o ile dotyczą minimalnego (pierwszego) elementu. Operacje na dowolnych i maksymalnych elementach wykonywane są w czasie liniowym: musimy przejrzeć całą lub średnio połowę listy. Dla list dwukierunkowych mamy dostęp do obu końców listy: operacje na elementach minimalnych i maksymalnych można wykonać szybko (w czasie stałym) a pozostałe — w czasie liniowym.

8.4. **Podsumowanie.** Poniższa tabela przedstawia złożoność operacji dla poszczególnych struktur do przechowywania danych:

	D	Dn	Dx	U	Un	Ux	Z	Zn	Zx
Tablica nieuporządkowana	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Tablica uporządkowana	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(\log(n))$	$O(1)$	$O(1)$
Lista jednokierunkowa	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$
Lista dwukierunkowa	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$

Wnioski:

(1) Tablice nieuporządkowane są efektywne kiedy gromadzimy dane, ale nie zamierzamy z nich natychmiast korzystać.

(2) Tablice uporządkowane najlepiej stosować, gdy zasób danych rzadko się zmienia, tzn. najczęściej wykonywaną operacją jest wyszukiwanie.

(3) List najlepiej używać, gdy działamy na skrajnych elementach: minimalnych lub maksymalnych.

Żadna z wymienionych struktur danych nie zapewnia szybkiego (tj. szybszego niż liniowe) działania wszystkich operacji. Czy takie uniwersalne struktury danych istnieją? O tym na kolejnych zajęciach.

8.5. **Zadania.**

(1) Napisać funkcję

```
bool znajdz(vector<int>& t, int w);
```

która wyszukuje binarnie element w tablicy używając rekurencji.

(2) Napisać klasę `Stos` przechowująca zbiór napisów (typu `string`). Dodawane napisy są dodawane zawsze na końcu; podobnie możemy usuwać napisy tylko z końca.

```
class Stos {
public:
    Stos(); // tworzy pusty stos
    void dodaj(string s); // dodaje napis na końcu
    string usun(); // usuwa ostatni napis i go zwraca
    int ile(); // zwraca liczbe elementów
    void drukuj(); // drukuje elementy (od końca)
};
```

Oczywiście można korzystać z klas, które pojawiły się na poprzednich zajęciach.

(3) Napisać klasę `Kolejka` przechowująca zbiór napisów (typu `string`). Tym razem dodajemy napisy na końcu, a usuwamy te początkowe.

```
class Kolejka {
public:
    Kolejka(); // tworzy pustą kolejkę
    void dodaj(string s); // dodaje napis na końcu
    string usun(); // usuwa pierwszy napis i go zwraca
    int ile(); // zwraca liczbe elementów
    void drukuj(); // drukuje elementy (od początku)
};
```