Wrocław University of Science and Technology

HR EXCELLENCE IN RESEARCH

Struktury danych

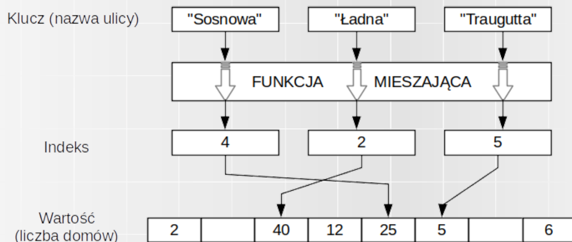
Wykład 8

Tablice mieszające

dr inż. Jarosław Rudy

Tablica mieszająca (1)

- Tablica mieszająca (hash table) – struktura danych służąca do implementacji słownika.
- Za pomocą funkcji mieszającej klucz zamieniany jest na indeks.
- Indeks wykorzystywany jest do dostępu do tablicy „statycznej”.



Funkcja mieszająca (1)

- Funkcja mieszająca $h(x)$ przekształca klucz x na indeks elementu (kubełka) tablicy.
- Zbiór indeksów jest skończony i ma rozmiar m (nie mylić z n!).
- Zbiór kluczy jest z reguły znacznie większy niż zbiór indeksów, może być też nieskończony.
  - Z tego powodu funkcję mieszająca nazywa się też funkcją skrótu.
- Konieczność obliczenia funkcji mieszającej powoduje narzut czasowy.

Funkcja mieszająca (2)

- Dla liczb całkowitych powszechną formą funkcji $h(x)$ jest operacja modulo:

$$h(x) = x \bmod m \tag{1}$$

  - Dzielenie jest stosunkowo powolne.
  - Problem klasteryzacji.
  - Wystarczająco dobre w praktyce w wielu przypadkach.

- Prostym wariantem dla łańcuchów tekstowych jest zsumowanie wszystkkich znaków łańcucha.

Funkcja mieszająca (3)

- Funkcja mieszająca z użyciem mnożenia:

$$h_\alpha(x) = \lfloor (\alpha x \bmod W)/(W/m) \rfloor \tag{2}$$

  - Konieczność właściwego doboru α i W.
- Haszowanie algebraiczne.
- Haszowanie Fibonacciego.
- Haszowanie z unikalną permutacją.
- Funkcja identyczności $h(x) = x$, sensowne jeśli zbiór kluczy jest „mały”.

Funkcja mieszająca (4)

- Ponieważ m jest mniejsze niż liczba kluczy, to będą istnieć takie klucze $x_1 \neq x_2$, że $h(x_1) = h(x_2)$ (wynika to z zasady szufladkowej Dirichleta).
- Taką sytuację nazywamy kolizją.
- W przypadku kolizji dwa różne klucze mapowane są do tego samego miejsca (kubełka) w tablicy mieszającej.
  - Różne implementacje tablicy radzą sobie z tym w różny sposób.
- Im mniej kolizji powoduje funkcja mieszająca tym lepiej.[1]

[1] W kryptografii dobiera się „większe” funkcje skrótu, przy których praktyczna szansa wystąpienia kolizji jest minimalna.

Funkcja mieszająca (5)

Paradoks dnia urodzin

Załóżmy, że w pokoju znajduje się k osób. Jakie jest prawdopodobieństwo, że co najmniej 2 z nich mają urodziny tego samego dnia?

- Dla $k = 1$ szansa jest 0%.
- Dla $k = 367$ szansa jest 100% (licząc rok przestępny).
- Dla jakiego k szansa jest $\geqslant 50\%$? Naiwna interpolacja wskazała by wartość $k \geqslant 183$, ale w rzeczywistości jest to $k \geqslant 23$!
- Wniosek: pierwsza kolizja może wystąpić szybciej niż mogło by się wydawać!

Funkcja mieszająca (6)

Załóżmy tablicę mieszającą z $m = 10^6$ i jednorodną funkcją mieszającą. Szansa na wystąpienie kolizji dla k operacji insert() wynosi:

- 10% przy $k = 460$,
- 25% przy $k = 759$,
- 50% przy $k = 1178$,
- 90% przy $k = 2146$,
- 99% przy $k = 3034$ (0,3034% m!).

Funkcja mieszająca (7)

y

1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1

Prawdopodobieństwo

x

2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50

Ludzie

Funkcja mieszająca (8)

Idealna funkcja mieszająca

- Przyporządkowuje każdemu kluczowi osobny indeks (kubełek).
- Matematycznie jest funkcją różnowartościową.
- Brak kolizji.
- Możliwa do skonstruowania, gdy klucze znane są zawczasu.
- Idealna funkcja mieszająca jest minimalna, gdy indeksy są kolejnymi liczbami całkowitymi.

Funkcja mieszająca (9)

Cechy dobrej funkcji mieszającej:

- Deterministyczna.
- Jednorodna – każdy indeks powinien mieć zbliżoną liczbę kluczy, które się na niego mapują.
- Małe zmiany klucza powinny powodować duże zmiany indeksu (redukuje problem klasteryzacji).
- Kompromis czasu obliczenia względem częstości kolizji.
- Pozwala na różny rozmiar klucza i różne wartości m.

Rozwiązywanie kolizji (1)

Istnieją dwa podstawowe sposoby rozwiązywania kolizji:

- Metoda łańcuchowa (separate chaining).
  - Pojedynczy kubułek przechowuje wszystkie klucze, które się do niego mapują.
  - Konieczność organizacji wielu wartości w ramach kubełka.
- Adresowanie otwarte (open addressing).
  - Każdy kubełek przechowuje najwyżej 1 element.
  - Gdy drugi element zmapuje się do zajętego kubełka, to trzeba wyznaczyć mu inny kubełek.

Współczynnik zajętości (1)

- Najważniejszym parametrem opisującym tablicę mieszającą jest współczynnik zajętości (load factor), definiowany jako:

$$\alpha = \frac{n}{m}, \tag{3}$$

gdzie n to liczba elementów przechowywanych w strukurze, a m to liczba kubełków.

- Wysokość load factor znacząco wpływa na wydajność operacji na tablicy mieszającej.

Współczynnik zajętości (2)

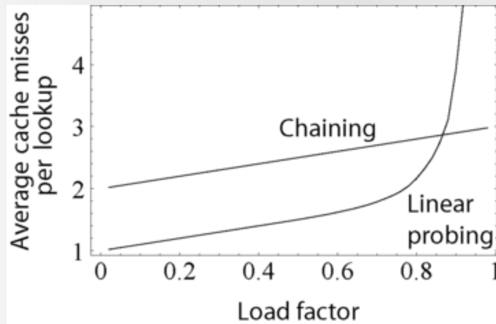
- Dla open addressing dopuszczalne wartości α są w przedziale $[0, 1]$ tj. $n \leqslant m$.
  - Nie można przechować więcej niż m kluczy, bo każda wymaga osobnego kubełka.
- Gdy α zbliża się do 1, coraz trudniej znaleźć wolny kubełek i wydajność drastycznie spada.
- Gdy α przekracza wartość graniczną (w praktyce od 0.6 do 0.8), to należy zwiększyć rozmiar tablicy.
- Często tablicę zmniejsza się, gdy α spadnie do $\frac{1}{4}$ wartości granicznej.
  - Pozwala ograniczyć zużycie pamięci (typowo połowa lub więcej tablicy jest pusta!).

Współczynnik zajętości (3)

- Dla separate chaining w teorii α może być dowolnie duże dla stałego *m*.
  - Oznacza to, że średnio w jednym kubełku jest α kluczy.
- Wydajność jednak wciąż spada wraz ze wzrostem α, więc w praktyce stosuje się „miękką” wartość graniczną (zwykle pomiędzy 1 a 3), po przekroczeniu której zwiększa się rozmiar tablicy.
- Analogicznie tablicę można zmniejszać, gdy α odpowiednio spadnie.

Współczynnik zajętości (4)

Metody open addressing (np. linear probing) mogą średnio wymagać mniej czasu niż separate chaining, o ile α nie jest bliskie 1.



W praktyce dobrze zarządzana tablica mieszająca typu open addressing rzadko potrzebuje przejrzeć więcej niż 3 kubełki.

Zmiana rozmiaru (1)

- Zmiana rozmiaru tablicy oznacza zmianę liczby kubełków m, a to z kolei zmienia dopuszczalny zakres kluczy.
  - Potrzebna jest więc nowa funkcja mieszająca, zaś operacja nie jest oczywista.
- Typowe metody zwiększenia rozmiaru:
  - Zmiana jednorazowa.
  - Zmiana stopniowa.
  - Linear hashing.

Zmiana rozmiaru (2)

Zmiana jednorazowa:

- Dana jest stara tablica mieszajaca z funkcja mieszająca $h_{\mathrm{old}}(x)$.
- Tworzymy nową (większą) tablicę mieszającą.
- Tworzymy nową funkcje mieszającą $h_{\mathrm{new}}(x)$.
- Dla każdego zajętego kubełka w starej tablicy dodajemy jego zawartość do nowej tablicy (z użyciem $h_{\mathrm{new}}(x)$).
- Zajmuje dużo czasu.
  - Jeśli nowa tablica jest o czynnik (np. 2x) większa od starej, to zamortyzowany koszt zwiększenia jest stały (analogicznie jak dla tablicy dynamicznej).

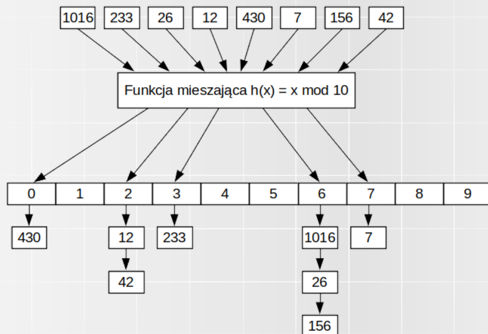
Zmiana rozmiaru (3)

Zmiana stopniowa:

- Tworzymy nową tablicę i funkcję mieszającą $h_{\mathrm{new}}(x)$.
- Na czas przenosin stosowane są obie tablice:
  - find() przeszukuje obie tablice (używając $h_{\mathrm{old}}(x)$ oraz $h_{\mathrm{new}}(x)$).
  - insert() dodaje do nowej tablicy (używając $h_{\mathrm{new}}(x)$).
    - Każdy insert() przenosi też k wpisów ze starej tablicy do nowej.
- Gdy przeniesione zostaną wszystkie elementy, starą tablicę można usunąć.

Separate chaining (1)

- Domyślnie separate chaining w każdym kubełku przechowuje listę wiązaną elementów.
- Operacje find()/insert()/delete() muszą operować też na liście.



Separate chaining (2)

- Lista wiązana ma pesymistyczny czas wyszukiwania $O(n)$, rzutujący na złożoność całej tablicy.
  - Możemy poprawić stosując zbalansowane drzewa BST.
- Lista wiązana słabiej współpracuje z pamięcią podręczną procesora.
  - Możemy poprawić stosując tablicę dynamiczną.
  - Problem pamięci podręcznej dotyczy też samej głównej tablicy – sięgamy do „losowych” indeksów, część indeksów jest pusta.

Metoda 2-choice hashing:

- Modyfikacja separate chaining (choć może też działać dla innych metod rozwiązywania kolizji).
- Jedna tablica, dwie funkcje mieszające $h_1(x)$ oraz $h_2(x)$.
- Podczas insert() stosowane są obie funkcje, zaś element trafia do kubełka mającego mniej elementów.
  - Jeśli kubełki mają równy rozmiar, to wykorzystujemy ten, który wskazała funkcja $h_1(x)$.
- Podobnie find() poszukuje elementu w obu możliwych kubełkach.
- Dzięki zasadzie power of 2 choices, znacząco zmniejszamy możliwość wystąpienia dużych kubełków.

Open addressing (1)

Metoda open addressing:

- Przy dodawaniu elementu najpierw obliczamy standardowo indeks funkcją mieszającą.
- Jeśli wyznaczony kubełek jest zajęty, wyznaczamy kolejny kubełek do sprawdzenia.
- Jeśli nowy kubełek jest zajęty, to sytuacja się powtarza aż do znalezienia pustego kubełka.
  - Powstaje ciąg poszukiwań (probing sequence), zaś jego długość określa wydajność tablicy.
- Analogiczny proces należy przeprowadzić dla find() i delete().

Open addressing (2)

Istnieją różne sposoby tworzenia ciągu poszukiwań:

- Linear probing – kolejny sprawdzany kubełek jest w odstępie C kubełków (często $C = 1$) od poprzedniego.
  - Indeks k-tego kubełka do sprawdzenia dany jest więc wzorem:

$$h(x) + kC. \tag{4}$$

  - Isotne jest by funkcja mieszająca unikała klasteryzacji/grupowania, gdzie indeksy kluczy gromadzą się blisko siebie.
    - Grupowanie, nawet jeśli nie powoduje kolizji, jest szkodliwe dla adresowania otwartego (ale nie dla metody łańcuchowej).
  - Lepiej współpracuje z pamięcią podręczną procesora (dla małych C).

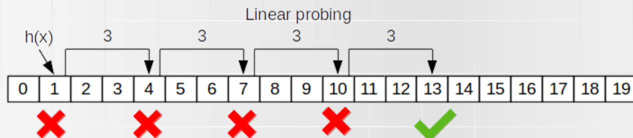
Open addressing (3)

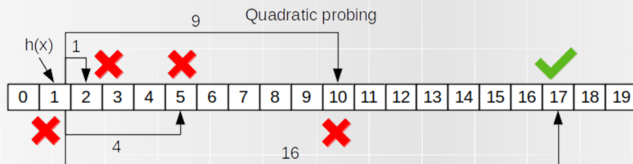
- Quadratic probing – kolejne odstępy od początkowego indeksu $h(x)$ dane są kolejnymi wartościami pewnej funkcji kwadratowej.
  - Bardziej odporne na grupowanie, podatne na grupowanie wtórne.
- Double hashing – odstępy jest liniowy, ale zależny od klucza x i dany drugą funkcją mieszającą $h_2(x)$. Indeks w k-tej próbie wynosi więc:

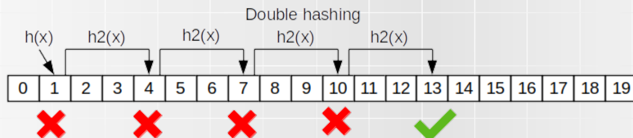
$$h(x) + kh_2(x). \tag{5}$$

  - Odporne na problem grupowania.

Open addressing (4)







Cuckoo hashing (1)

- Haszowanie kukułcze – odmiana open addressing.
- Dwie tablice T_1 oraz T_2 o równym rozmiarze.
- Tablice mają osobne funkcje haszujące, odpowiednio $h_1(x)$ oraz $h_2(x)$.
- Każdy klucz ma więc dwie możliwe lokalizacje: podstawową oraz alternatywną.
- Przy dodawaniu nowy element x_1 wstawiany jest do T_1. Jeśli był tam już jakiś inny element x_2, to usuwamy x_2 z T_1 i wstawiamy go na jego alternatywne miejsce w T_2.
  - Procedura się powtarza dla x_2: jeśli miejsce x_2 w T_2 zajmował x_3, to x_3 jest usuwany i wstawiany w swoje alternatywne miejsce w T_1.

Cuckoo hashing (2)

- Może wystąpić cykl.
  - Wykrywany przez przekroczenie licznika kroków wstawiania.
  - Naprawiany poprzez stworzenie nowych funkcji $h_1(x)$ i $h_2(x)$ oraz ponowane haszowanie zawartości tablic.
- Każdy klucz jest w $h_1(x)$ w T_1 lub w $h_2(x)$ w T_2.
  - find() jest więc w pesymistycznym czasie $O(1)$, podobnie remove().
- insert() może zająć długi czas, ale koszt zamortyzowany jest $O(1)$, nawet uwzględniając konieczność przehaszowania tablic.

Cuckoo hashing (3)

E	A			C			
		D	F		B		

Cuckoo hashing (4)

- Haszowanie kukułcze ma więc bardzo dobrą złożoność teoretyczną.
- Złożoność zakłada jednak, że load factor jest poniżej 0.5.
  - Niefektywne użycie pamięci.
  - Można użyć 3 funkcji i trzech tablic, co średnio pozwala wykorzystać ponad 90% miejsca kosztem spadku wydajności (jednak wciąż $O(1)$).
- W praktyce cuckoo hashing jest średnio wolniejsze od linear probing (więcej chybień podczas find()), ale może być przydatne w sytuacjach gdzie ważna jest redukcja przypadku pesymistycznego.

Zastosowania

Zastosowania tablic mieszających:

- Słowniki.
- Zbiory (brak kolejności).
- Pamięci podręczne.
- Indeksy baz danych.
- Tablice transpozycji gier.

Słowniki – podsumowanie (1)

Struktura	find()/remove()		insert()		Uporządk.
	Avg.	Worst	Avg.	Worst	
Hash table[2]	$O(1)$	$O(n)$	$O(1)$	$O(n)$	nie
Hash table[3]	$O(1)$	$O(\log n)$	$O(1)$	$O(\log n)$	nie
Hash table[4]	$O(1)$	$O(1)$	$O(1)$	$O(1)$[5]	nie
AVL/black-red tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	tak
BST	$O(\log n)$	$O(n)$	$O(\log n)$	$O(n)$	tak
Lista[6]	$O(n)$	$O(n)$	$O(1)/O(n)$	$O(1)/O(n)$	„tak”

[2] Adresowanie otwarte lub kubełki z listą
[3] Adresowanie zamknięte plus kubełki ze zbalansowanym BST
[4] Cuckoo hashing
[5] Koszt zamortyzowany
[6] Dwa warianty: dodawanie na koniec lub dodawanie w pozycji posortowanej