Słowniki i ich realizacja za pomocą haszowania

Piotr Beling

Uniwersytet Łódzki

2019 (ostatnia aktualizacja: 2022)

http://pbeling.w8.pl

Abstrakcyjna struktura danych

Abstrakcyjna struktura danych zdefiniowana jest poprzez

- podanie zestawu operacji, które można na niej wykonać,
- wraz z opisem ich semantyki (zachowania),
- ale bez opisu ich implementacji.

Przykład: stos to abstrakcyjna struktura danych umożliwiająca:

- dodanie nowego element na szczyt stosu,
- zdjęcie elementu ze szczytu stosu i zwrócenie jego wartości.

Jedna abstrakcyjna struktura danych może mieć wiele różnych realizacji (implementacji).
Przykładowo stos można zaimplementować zarówno za pomocą tablicy jak i listy dowiązaniowej.

Abstrakcyjna struktura danych

Abstrakcyjna struktura danych zdefiniowana jest poprzez

- podanie zestawu operacji, które można na niej wykonać,
- wraz z opisem ich semantyki (zachowania),
- ale bez opisu ich implementacji.

Przykład: stos to abstrakcyjna struktura danych umożliwiająca:

- dodanie nowego element na szczyt stosu,
- zdjęcie elementu ze szczytu stosu i zwrócenie jego wartości.

Jedna abstrakcyjna struktura danych może mieć wiele różnych realizacji (implementacji).
Przykładowo stos można zaimplementować zarówno za pomocą tablicy jak i listy dowiązaniowej.

Abstrakcyjna struktura danych

Abstrakcyjna struktura danych zdefiniowana jest poprzez

- podanie zestawu operacji, które można na niej wykonać,
- wraz z opisem ich semantyki (zachowania),
- ale bez opisu ich implementacji.

Przykład: stos to abstrakcyjna struktura danych umożliwiająca:

- dodanie nowego element na szczyt stosu,
- zdjęcie elementu ze szczytu stosu i zwrócenie jego wartości.

Jedna abstrakcyjna struktura danych może mieć wiele różnych realizacji (implementacji).
Przykładowo stos można zaimplementować zarówno za pomocą tablicy jak i listy dowiązaniowej.

Abstrakcyjna struktura danych

Abstrakcyjna struktura danych zdefiniowana jest poprzez

- podanie zestawu operacji, które można na niej wykonać,
- wraz z opisem ich semantyki (zachowania),
- ale bez opisu ich implementacji.

Przykład: stos to abstrakcyjna struktura danych umożliwiająca:

- dodanie nowego element na szczyt stosu,
- zdjęcie elementu ze szczytu stosu i zwrócenie jego wartości.

Jedna abstrakcyjna struktura danych może mieć wiele różnych realizacji (implementacji).
Przykładowo stos można zaimplementować zarówno za pomocą tablicy jak i listy dowiązaniowej.

Abstrakcyjna struktura danych

Abstrakcyjna struktura danych zdefiniowana jest poprzez

- podanie zestawu operacji, które można na niej wykonać,
- wraz z opisem ich semantyki (zachowania),
- ale bez opisu ich implementacji.

Przykład: stos to abstrakcyjna struktura danych umożliwiająca:

- dodanie nowego element na szczyt stosu,
- zdjęcie elementu ze szczytu stosu i zwrócenie jego wartości.

Jedna abstrakcyjna struktura danych może mieć wiele różnych realizacji (implementacji).
Przykładowo stos można zaimplementować zarówno za pomocą tablicy jak i listy dowiązaniowej.

Abstrakcyjna struktura danych

Abstrakcyjna struktura danych zdefiniowana jest poprzez

- podanie zestawu operacji, które można na niej wykonać,
- wraz z opisem ich semantyki (zachowania),
- ale bez opisu ich implementacji.

Przykład: stos to abstrakcyjna struktura danych umożliwiająca:

- dodanie nowego element na szczyt stosu,
- zdjęcie elementu ze szczytu stosu i zwrócenie jego wartości.

Jedna abstrakcyjna struktura danych może mieć wiele różnych realizacji (implementacji).
Przykładowo stos można zaimplementować zarówno za pomocą tablicy jak i listy dowiązaniowej.

Abstrakcyjna struktura danych

Abstrakcyjna struktura danych zdefiniowana jest poprzez

- podanie zestawu operacji, które można na niej wykonać,
- wraz z opisem ich semantyki (zachowania),
- ale bez opisu ich implementacji.

Przykład: stos to abstrakcyjna struktura danych umożliwiająca:

- dodanie nowego element na szczyt stosu,
- zdjęcie elementu ze szczytu stosu i zwrócenie jego wartości.

Jedna abstrakcyjna struktura danych może mieć wiele różnych realizacji (implementacji).
Przykładowo stos można zaimplementować zarówno za pomocą tablicy jak i listy dowiązaniowej.

Abstrakcyjna struktura danych

Abstrakcyjna struktura danych zdefiniowana jest poprzez

- podanie zestawu operacji, które można na niej wykonać,
- wraz z opisem ich semantyki (zachowania),
- ale bez opisu ich implementacji.

Przykład: stos to abstrakcyjna struktura danych umożliwiająca:

- dodanie nowego element na szczyt stosu,
- zdjęcie elementu ze szczytu stosu i zwrócenie jego wartości.

Jedna abstrakcyjna struktura danych może mieć wiele różnych realizacji (implementacji).
Przykładowo stos można zaimplementować zarówno za pomocą tablicy jak i listy dowiązaniowej.

Abstrakcyjna struktura danych

Abstrakcyjna struktura danych zdefiniowana jest poprzez

- podanie zestawu operacji, które można na niej wykonać,
- wraz z opisem ich semantyki (zachowania),
- ale bez opisu ich implementacji.

Przykład: stos to abstrakcyjna struktura danych umożliwiająca:

- dodanie nowego element na szczyt stosu,
- zdjęcie elementu ze szczytu stosu i zwrócenie jego wartości.

Jedna abstrakcyjna struktura danych może mieć wiele różnych realizacji (implementacji).
Przykładowo stos można zaimplementować zarówno za pomocą tablicy jak i listy dowiązaniowej.

Abstrakcyjne struktury danych a interfejsy

- Pojęcie abstrakcyjnej struktury danych jest bliskie znaczeniowo do używanego w zorientowanych obiektowo językach programowania pojęcia interfejsu (który może być implementowany przez wiele różnych klas).
- W odróżnieniu od interfejsu, zdefiniowanie abstrakcyjnej struktury danych nie wymaga podania zależnych od języka szczegółów, jak np. konkretnych typów parametrów i wartości zwracanych przez poszczególne operacje/metody.

Abstrakcyjne struktury danych a interfejsy

- Pojęcie abstrakcyjnej struktury danych jest bliskie znaczeniowo do używanego w zorientowanych obiektowo językach programowania pojęcia interfejsu (który może być implementowany przez wiele różnych klas).
- W odróżnieniu od interfejsu, zdefiniowanie abstrakcyjnej struktury danych nie wymaga podania zależnych od języka szczegółów, jak np. konkretnych typów parametrów i wartości zwracanych przez poszczególne operacje/metody.

Słownik

Słownik – abstrakcyjna struktura danych służąca do przechowywania elementów, posiadająca następujące operacje:

- `find` – wyszukanie elmentu o zadanym kluczu,
- `insert` – dodanie elementu do słownika,
- `delete` – usunięcie elementu o zadanym kluczu.

Klucz:

- cecha elementu (np. pole w jego klasie),
- może być wyznaczany na podstawie elementu za pomocą pewnej funkcji (lub metody elementu),
- może być tożsamy z elementem (tak zakłada np. `std::set` w C++, czy `set` w Pythonie),
- element może być też parą klucz-wartość (tak zakłada np. `std::map` w C++, czy `dict` w Pythonie).

Słownik

Słownik – abstrakcyjna struktura danych służąca do przechowywania elementów, posiadająca następujące operacje:

- `find` – wyszukanie elmentu o zadanym kluczu,
- `insert` – dodanie elementu do słownika,
- `delete` – usunięcie elementu o zadanym kluczu.

Klucz:

- cecha elementu (np. pole w jego klasie),
- może być wyznaczany na podstawie elementu za pomocą pewnej funkcji (lub metody elementu),
- może być tożsamy z elementem (tak zakłada np. `std::set` w C++, czy `set` w Pythonie),
- element może być też parą klucz-wartość (tak zakłada np. `std::map` w C++, czy `dict` w Pythonie).

Słownik

Słownik – abstrakcyjna struktura danych służąca do przechowywania elementów, posiadająca następujące operacje:

- `find` – wyszukanie elmentu o zadanym kluczu,
- `insert` – dodanie elementu do słownika,
- `delete` – usunięcie elementu o zadanym kluczu.

Klucz:

- cecha elementu (np. pole w jego klasie),
- może być wyznaczany na podstawie elementu za pomocą pewnej funkcji (lub metody elementu),
- może być tożsamy z elementem (tak zakłada np. `std::set` w C++, czy `set` w Pythonie),
- element może być też parą klucz-wartość (tak zakłada np. `std::map` w C++, czy `dict` w Pythonie).

Słownik

Słownik – abstrakcyjna struktura danych służąca do przechowywania elementów, posiadająca następujące operacje:

- `find` – wyszukanie elmentu o zadanym kluczu,
- `insert` – dodanie elementu do słownika,
- `delete` – usunięcie elementu o zadanym kluczu.

Klucz:

- cecha elementu (np. pole w jego klasie),
- może być wyznaczany na podstawie elementu za pomocą pewnej funkcji (lub metody elementu),
- może być tożsamy z elementem (tak zakłada np. `std::set` w C++, czy `set` w Pythonie),
- element może być też parą klucz-wartość (tak zakłada np. `std::map` w C++, czy `dict` w Pythonie).

Słownik

Słownik – abstrakcyjna struktura danych służąca do przechowywania elementów, posiadająca następujące operacje:

- `find` – wyszukanie elmentu o zadanym kluczu,
- `insert` – dodanie elementu do słownika,
- `delete` – usunięcie elementu o zadanym kluczu.

Klucz:

- cecha elementu (np. pole w jego klasie),
- może być wyznaczany na podstawie elementu za pomocą pewnej funkcji (lub metody elementu),
- może być tożsamy z elementem (tak zakłada np. `std::set` w C++, czy `set` w Pythonie),
- element może być też parą klucz-wartość (tak zakłada np. `std::map` w C++, czy `dict` w Pythonie).

Słownik

Słownik – abstrakcyjna struktura danych służąca do przechowywania elementów, posiadająca następujące operacje:

- `find` – wyszukanie elmentu o zadanym kluczu,
- `insert` – dodanie elementu do słownika,
- `delete` – usunięcie elementu o zadanym kluczu.

Klucz:

- cecha elementu (np. pole w jego klasie),
- może być wyznaczany na podstawie elementu za pomocą pewnej funkcji (lub metody elementu),
- może być tożsamy z elementem (tak zakłada np. `std::set` w C++, czy `set` w Pythonie),
- element może być też parą klucz-wartość (tak zakłada np. `std::map` w C++, czy `dict` w Pythonie).

Słownik

Słownik – abstrakcyjna struktura danych służąca do przechowywania elementów, posiadająca następujące operacje:

- `find` – wyszukanie elmentu o zadanym kluczu,
- `insert` – dodanie elementu do słownika,
- `delete` – usunięcie elementu o zadanym kluczu.

Klucz:

- cecha elementu (np. pole w jego klasie),
- może być wyznaczany na podstawie elementu za pomocą pewnej funkcji (lub metody elementu),
- może być tożsamy z elementem (tak zakłada np. `std::set` w C++, czy `set` w Pythonie),
- element może być też parą klucz-wartość (tak zakłada np. `std::map` w C++, czy `dict` w Pythonie).

Słownik

Słownik – abstrakcyjna struktura danych służąca do przechowywania elementów, posiadająca następujące operacje:

- `find` – wyszukanie elmentu o zadanym kluczu,
- `insert` – dodanie elementu do słownika,
- `delete` – usunięcie elementu o zadanym kluczu.

Klucz:

- cecha elementu (np. pole w jego klasie),
- może być wyznaczany na podstawie elementu za pomocą pewnej funkcji (lub metody elementu),
- może być tożsamy z elementem (tak zakłada np. `std::set` w C++, czy `set` w Pythonie),
- element może być też parą klucz-wartość (tak zakłada np. `std::map` w C++, czy `dict` w Pythonie).

Realizacja słownika za pomocą nieposortowanej tablicy

Słownik możemy zrealizować za pomocą nieposortowanej tablicy, np. dynamicznie rozszerzalnej tablicy jak `vector` z C++ czy **`list`** z Pythona (o zamortyzowanym, stałym czasie dodawania na koniec i linowej złożoności pamięciowej), realizując operacje następująco:

- `find` – przeglądamy kolejne elementy aż do napotkania zadanego klucza lub końca tablicy. Złożoność czasowa: $O(n)$.
- `insert` – dodajemy element na koniec tablicy. Złożoność czasowa: zamortyzowana $O(1)$ albo (gdy nie dopuszczamy duplikatów) $O(n)$.
- `delete` – nadpisujemy kasowany element ostatnim (albo je zamieniamy) i pomniejszamy tablicę o 1 kasując ostatni element. Złożoność czasowa:
  - amortyzowana $O(1)$ gdy znamy indeks elementu do skasowania, lub
  - $O(n)$ gdy musimy go wpierw znaleźć.

Oznaczenia: n - liczba elementów w słowniku.

Realizacja słownika za pomocą nieposortowanej tablicy

Słownik możemy zrealizować za pomocą nieposortowanej tablicy, np. dynamicznie rozszerzalnej tablicy jak `vector` z C++ czy **`list`** z Pythona (o zamortyzowanym, stałym czasie dodawania na koniec i linowej złożoności pamięciowej), realizując operacje następująco:

- `find` – przeglądamy kolejne elementy aż do napotkania zadanego klucza lub końca tablicy. Złożoność czasowa: $O(n)$.
- `insert` – dodajemy element na koniec tablicy. Złożoność czasowa: zamortyzowana $O(1)$ albo (gdy nie dopuszczamy duplikatów) $O(n)$.
- `delete` – nadpisujemy kasowany element ostatnim (albo je zamieniamy) i pomniejszamy tablicę o 1 kasując ostatni element. Złożoność czasowa:
  - amortyzowana $O(1)$ gdy znamy indeks elementu do skasowania, lub
  - $O(n)$ gdy musimy go wpierw znaleźć.

Oznaczenia: n - liczba elementów w słowniku.

Realizacja słownika za pomocą nieposortowanej tablicy

Słownik możemy zrealizować za pomocą nieposortowanej tablicy, np. dynamicznie rozszerzalnej tablicy jak `vector` z C++ czy **`list`** z Pythona (o zamortyzowanym, stałym czasie dodawania na koniec i linowej złożoności pamięciowej), realizując operacje następująco:

- `find` – przeglądamy kolejne elementy aż do napotkania zadanego klucza lub końca tablicy. Złożoność czasowa: $O(n)$.
- `insert` – dodajemy element na koniec tablicy. Złożoność czasowa: zamortyzowana $O(1)$ albo (gdy nie dopuszczamy duplikatów) $O(n)$.
- `delete` – nadpisujemy kasowany element ostatnim (albo je zamieniamy) i pomniejszamy tablicę o 1 kasując ostatni element. Złożoność czasowa:
  - amortyzowana $O(1)$ gdy znamy indeks elementu do skasowania, lub
  - $O(n)$ gdy musimy go wpierw znaleźć.

Oznaczenia: n - liczba elementów w słowniku.

Realizacja słownika za pomocą nieposortowanej tablicy

Słownik możemy zrealizować za pomocą nieposortowanej tablicy, np. dynamicznie rozszerzalnej tablicy jak `vector` z C++ czy **`list`** z Pythona (o zamortyzowanym, stałym czasie dodawania na koniec i linowej złożoności pamięciowej), realizując operacje następująco:

- `find` – przeglądamy kolejne elementy aż do napotkania zadanego klucza lub końca tablicy. Złożoność czasowa: $O(n)$.
- `insert` – dodajemy element na koniec tablicy. Złożoność czasowa: zamortyzowana $O(1)$ albo (gdy nie dopuszczamy duplikatów) $O(n)$.
- `delete` – nadpisujemy kasowany element ostatnim (albo je zamieniamy) i pomniejszamy tablicę o 1 kasując ostatni element. Złożoność czasowa:
  - amortyzowana $O(1)$ gdy znamy indeks elementu do skasowania, lub
  - $O(n)$ gdy musimy go wpierw znaleźć.

Oznaczenia: n - liczba elementów w słowniku.

Haszowanie – idea

- By móc szybko wyszukiwać, ludzie przydzielają przedmiotom miejsca, np. koszule odkładają do szafy z ubraniami, garnki do odpowiedniej szafki w kuchni, itd.
- Dzięki temu nie muszą przeszukiwać całego domu, by znaleźć przedmiot. Wystarczy zajrzeć w przydzielone mu miejsce.
- Haszowania bardzo przypomina wyżej opisane postępowanie, z tym, że: miejsca oznaczają indeksy w tablicy, zaś za przydzielanie ich elementom odpowiada funkcja haszująca.

Haszowanie – idea

- By móc szybko wyszukiwać, ludzie przydzielają przedmiotom miejsca, np. koszule odkładają do szafy z ubraniami, garnki do odpowiedniej szafki w kuchni, itd.
- Dzięki temu nie muszą przeszukiwać całego domu, by znaleźć przedmiot. Wystarczy zajrzeć w przydzielone mu miejsce.
- Haszowania bardzo przypomina wyżej opisane postępowanie, z tym, że: miejsca oznaczają indeksy w tablicy, zaś za przydzielanie ich elementom odpowiada funkcja haszująca.

Haszowanie – idea

- By móc szybko wyszukiwać, ludzie przydzielają przedmiotom miejsca, np. koszule odkładają do szafy z ubraniami, garnki do odpowiedniej szafki w kuchni, itd.
- Dzięki temu nie muszą przeszukiwać całego domu, by znaleźć przedmiot. Wystarczy zajrzeć w przydzielone mu miejsce.
- Haszowania bardzo przypomina wyżej opisane postępowanie, z tym, że: miejsca oznaczają indeksy w tablicy, zaś za przydzielanie ich elementom odpowiada funkcja haszująca.

Haszowanie – idea – przykład

- Za pomocą haszowania i tablicy (`data`) o stałej długości 10, zrealizujemy prosty słownik o ograniczonej pojemności przechowujący liczby naturalne.
- Użyjemy funkcji haszującej $h(x) = x\%10$, gdzie % oznacza resztę z dzielenia.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:										

Haszowanie – idea – przykład

- Za pomocą haszowania i tablicy (`data`) o stałej długości 10, zrealizujemy prosty słownik o ograniczonej pojemności przechowujący liczby naturalne.
- Użyjemy funkcji haszującej $h(x) = x\%10$, gdzie % oznacza resztę z dzielenia.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:										

Haszowanie – idea – przykład

- Za pomocą haszowania i tablicy (`data`) o stałej długości 10, zrealizujemy prosty słownik o ograniczonej pojemności przechowujący liczby naturalne.
- Użyjemy funkcji haszującej $h(x) = x\%10$, gdzie % oznacza resztę z dzielenia.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:										

- Początkowo wszystkie komórki w tablicy są puste (słownik jest pusty).
- By oznaczyć komórkę jako pustą, można użyć jakiejś specjalnej wartości, np. -1.

Haszowanie – idea – przykład

- Za pomocą haszowania i tablicy (`data`) o stałej długości 10, zrealizujemy prosty słownik o ograniczonej pojemności przechowujący liczby naturalne.
- Użyjemy funkcji haszującej $h(x) = x\%10$, gdzie % oznacza resztę z dzielenia.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:								17		

- Wstawmy do słownika (tablicy) liczbę 17.
- Indeks pod który wpiszemy liczbę wyznaczamy za pomocą funkcji haszującej: $h(17) = 17\%10 = 7$.
- Wstawienie sprowadza się do pojedynczego przypisania `data[h(17)]`$\leftarrow$`17`, więc jego złożoność jest $O(1)$.

Haszowanie – idea – przykład

- Za pomocą haszowania i tablicy (`data`) o stałej długości 10, zrealizujemy prosty słownik o ograniczonej pojemności przechowujący liczby naturalne.
- Użyjemy funkcji haszującej $h(x) = x\%10$, gdzie % oznacza resztę z dzielenia.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:				23				17		

- Następnie, wstawmy liczbę 23.
- $h(23) = 23\%10 = 3$.

Haszowanie – idea – przykład

- Za pomocą haszowania i tablicy (`data`) o stałej długości 10, zrealizujemy prosty słownik o ograniczonej pojemności przechowujący liczby naturalne.
- Użyjemy funkcji haszującej $h(x) = x\%10$, gdzie % oznacza resztę z dzielenia.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:				23		75		17		

- Dodajmy jeszcze liczbę 75.
- $h(75) = 75\%10 = 5$.

Haszowanie – idea – przykład

- Za pomocą haszowania i tablicy (`data`) o stałej długości 10, zrealizujemy prosty słownik o ograniczonej pojemności przechowujący liczby naturalne.
- Użyjemy funkcji haszującej $h(x) = x\%10$, gdzie % oznacza resztę z dzielenia.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:				23		75		17		

- Poszukajmy liczby 23.
- Ponieważ $h(23) = 23\%10 = 3$, to liczba ta może znajdować jedynie pod indeksem 3.
- By stwierdzić czy 23 jest w słowniku, wystarczy więc sprawdzić warunek `data[h(23)]=23` (w naszym wypadku jest on spełniony), co można zrobić w czasie $O(1)$.

Haszowanie – idea – przykład

- Za pomocą haszowania i tablicy (`data`) o stałej długości 10, zrealizujemy prosty słownik o ograniczonej pojemności przechowujący liczby naturalne.
- Użyjemy funkcji haszującej $h(x) = x\%10$, gdzie % oznacza resztę z dzielenia.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:				23		75		17		

- Poszukajmy liczby 12.
- $h(12) = 12\%10 = 2$, więc 12 może znajdować się jedynie pod indeksem 2.
- Sprawdzamy warunek `data[h(12)]=12`. Nie jest on spełniony. Wnioskujemy więc, że 12 nie ma w słowniku.

Haszowanie – idea – przykład

- Za pomocą haszowania i tablicy (`data`) o stałej długości 10, zrealizujemy prosty słownik o ograniczonej pojemności przechowujący liczby naturalne.
- Użyjemy funkcji haszującej $h(x) = x\%10$, gdzie % oznacza resztę z dzielenia.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:				23				17		

- Usuńmy ze słownika liczbę 75.
- Wpierw wyznaczamy jej indeks: $h(75) = 75\%10 = 5$.
- Po upewnieniu się, że pod indeksem 5 rzeczywiście jest liczba 75 (tj. że `data[5]=75`), wystarczy komórkę o indeksie 5 oznaczyć jako pustą, poprzez przypisanie do niej specjalnej wartość (np. -1). Wszystko to można zrobić w czasie $O(1)$.

Haszowanie – idea – przykład

- Za pomocą haszowania i tablicy (`data`) o stałej długości 10, zrealizujemy prosty słownik o ograniczonej pojemności przechowujący liczby naturalne.
- Użyjemy funkcji haszującej $h(x) = x\%10$, gdzie % oznacza resztę z dzielenia.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:				23				17		

- Na koniec, spróbujmy jeszcze dodać do słownika liczbę 13.
- Powinna się ona znaleźć pod indeksem $h(13) = 13\%10 = 3$.
- Tam jednakże znajduje się już liczba 23.
- Mamy więc do czynienia z kolizją.
- Stanowi ona problem: pomimo wielu wolnych miejsc, nie możemy wstawić 13 (bez usunięcia/nadpisania 23).

Kolizje

- Jeśli zbiór wszystkich kluczy jest większy niż zbiór indeksów tablicy haszującej (a jest tak prawie zawsze), to funkcja haszująca h z pewnością nie jest różnowartościowa.
- Oznacza to, że musi dochodzić do **kolizji**, tj. że istnieją pary k_1, k_2 różnych kluczy ($k_1 \neq k_2$), takie że $h(k_1) = h(k_2)$.
- Jeśli chcemy by elementy o kolidujących kluczach mogły współistnieć w tablicy, musimy rozwiązać problem kolizji.
- Dwa najpopularniejsze **rozwiązania problemu kolizji** to: **metoda łańcuchowa** oraz **adresowanie otwarte**.
- W **metodzie łańcuchowej** każda komórka tablicy haszującej jest nieuporządkowaną listą lub dynamicznie rozszerzalną tablicą przechowującą elementy o kolidujących kluczach.
- W **adresowaniu otwartym** każda komórka może co prawda pomieścić tylko jeden element, ale funkcja haszująca wyznacza cały ciąg indeksów pod którymi element o danym kluczu może się znaleźć.

Kolizje

- Jeśli zbiór wszystkich kluczy jest większy niż zbiór indeksów tablicy haszującej (a jest tak prawie zawsze), to funkcja haszująca h z pewnością nie jest różnowartościowa.
- Oznacza to, że musi dochodzić do **kolizji**, tj. że istnieją pary k_1, k_2 różnych kluczy ($k_1 \neq k_2$), takie że $h(k_1) = h(k_2)$.
- Jeśli chcemy by elementy o kolidujących kluczach mogły współistnieć w tablicy, musimy rozwiązać problem kolizji.
- Dwa najpopularniejsze **rozwiązania problemu kolizji** to: **metoda łańcuchowa** oraz **adresowanie otwarte**.
- W **metodzie łańcuchowej** każda komórka tablicy haszującej jest nieuporządkowaną listą lub dynamicznie rozszerzalną tablicą przechowującą elementy o kolidujących kluczach.
- W **adresowaniu otwartym** każda komórka może co prawda pomieścić tylko jeden element, ale funkcja haszująca wyznacza cały ciąg indeksów pod którymi element o danym kluczu może się znaleźć.

Kolizje

- Jeśli zbiór wszystkich kluczy jest większy niż zbiór indeksów tablicy haszującej (a jest tak prawie zawsze), to funkcja haszująca h z pewnością nie jest różnowartościowa.
- Oznacza to, że musi dochodzić do **kolizji**, tj. że istnieją pary k_1, k_2 różnych kluczy ($k_1 \neq k_2$), takie że $h(k_1) = h(k_2)$.
- Jeśli chcemy by elementy o kolidujących kluczach mogły współistnieć w tablicy, musimy rozwiązać problem kolizji.
- Dwa najpopularniejsze **rozwiązania problemu kolizji** to: **metoda łańcuchowa** oraz **adresowanie otwarte**.
- W **metodzie łańcuchowej** każda komórka tablicy haszującej jest nieuporządkowaną listą lub dynamicznie rozszerzalną tablicą przechowującą elementy o kolidujących kluczach.
- W **adresowaniu otwartym** każda komórka może co prawda pomieścić tylko jeden element, ale funkcja haszująca wyznacza cały ciąg indeksów pod którymi element o danym kluczu może się znaleźć.

Kolizje

- Jeśli zbiór wszystkich kluczy jest większy niż zbiór indeksów tablicy haszującej (a jest tak prawie zawsze), to funkcja haszująca h z pewnością nie jest różnowartościowa.
- Oznacza to, że musi dochodzić do **kolizji**, tj. że istnieją pary k_1, k_2 różnych kluczy ($k_1 \neq k_2$), takie że $h(k_1) = h(k_2)$.
- Jeśli chcemy by elementy o kolidujących kluczach mogły współistnieć w tablicy, musimy rozwiązać problem kolizji.
- Dwa najpopularniejsze **rozwiązania problemu kolizji** to: **metoda łańcuchowa** oraz **adresowanie otwarte**.
- W **metodzie łańcuchowej** każda komórka tablicy haszującej jest nieuporządkowaną listą lub dynamicznie rozszerzalną tablicą przechowującą elementy o kolidujących kluczach.
- W **adresowaniu otwartym** każda komórka może co prawda pomieścić tylko jeden element, ale funkcja haszująca wyznacza cały ciąg indeksów pod którymi element o danym kluczu może się znaleźć.

Kolizje

- Jeśli zbiór wszystkich kluczy jest większy niż zbiór indeksów tablicy haszującej (a jest tak prawie zawsze), to funkcja haszująca h z pewnością nie jest różnowartościowa.
- Oznacza to, że musi dochodzić do **kolizji**, tj. że istnieją pary k_1, k_2 różnych kluczy ($k_1 \neq k_2$), takie że $h(k_1) = h(k_2)$.
- Jeśli chcemy by elementy o kolidujących kluczach mogły współistnieć w tablicy, musimy rozwiązać problem kolizji.
- Dwa najpopularniejsze **rozwiązania problemu kolizji** to: **metoda łańcuchowa** oraz **adresowanie otwarte**.
- W **metodzie łańcuchowej** każda komórka tablicy haszującej jest nieuporządkowaną listą lub dynamicznie rozszerzalną tablicą przechowującą elementy o kolidujących kluczach.
- W **adresowaniu otwartym** każda komórka może co prawda pomieścić tylko jeden element, ale funkcja haszująca wyznacza cały ciąg indeksów pod którymi element o danym kluczu może się znaleźć.

Kolizje

- Jeśli zbiór wszystkich kluczy jest większy niż zbiór indeksów tablicy haszującej (a jest tak prawie zawsze), to funkcja haszująca h z pewnością nie jest różnowartościowa.
- Oznacza to, że musi dochodzić do **kolizji**, tj. że istnieją pary k_1, k_2 różnych kluczy ($k_1 \neq k_2$), takie że $h(k_1) = h(k_2)$.
- Jeśli chcemy by elementy o kolidujących kluczach mogły współistnieć w tablicy, musimy rozwiązać problem kolizji.
- Dwa najpopularniejsze **rozwiązania problemu kolizji** to: **metoda łańcuchowa** oraz **adresowanie otwarte**.
- W **metodzie łańcuchowej** każda komórka tablicy haszującej jest nieuporządkowaną listą lub dynamicznie rozszerzalną tablicą przechowującą elementy o kolidujących kluczach.
- W **adresowaniu otwartym** każda komórka może co prawda pomieścić tylko jeden element, ale funkcja haszująca wyznacza cały ciąg indeksów pod którymi element o danym kluczu może się znaleźć.

Metoda łańcuchowa – przykład

Zrealizujemy słownik liczb naturalnych za pomocą tablicy haszującej z łańcuchową metodą rozwiązywania kolizji i funkcją haszującą $h(x) = x\%10$.

indeks:	0	1	2	3	4	5	6	7	8	9
wartości:										

Metoda łańcuchowa – przykład

Zrealizujemy słownik liczb naturalnych za pomocą tablicy haszującej z łańcuchową metodą rozwiązywania kolizji i funkcją haszującą $h(x) = x\%10$.

indeks:	0	1	2	3	4	5	6	7	8	9
wartości:										

- Początkowo słownik jest pusty.
- Nasza tablica haszująca składa się z 10 pustych łańcuchów, tj. list albo dynamicznie rozszerzalnych tablic.
- (w metodzie łańcuchowej nie trzeba stosować żadnych specjalnych wartości do oznaczania pustych komórek)

Metoda łańcuchowa – przykład

Zrealizujemy słownik liczb naturalnych za pomocą tablicy haszującej z łańcuchową metodą rozwiązywania kolizji i funkcją haszującą $h(x) = x\%10$.

indeks:	0	1	2	3	4	5	6	7	8	9
wartości:										

- Wstawmy do słownika (tablicy) liczbę 17.
- Indeks pod który wstawimy liczbę wyznaczamy za pomocą funkcji haszującej: $h(17) = 17\%10 = 7$.
- Dodajemy więc 17 do łańcucha o indeksie 7.

Metoda łańcuchowa – przykład

Zrealizujemy słownik liczb naturalnych za pomocą tablicy haszującej z łańcuchową metodą rozwiązywania kolizji i funkcją haszującą $h(x) = x\%10$.

indeks:	0	1	2	3	4	5	6	7	8	9
wartości:										

- Wstawmy do słownika (tablicy) liczbę 17.
- Indeks pod który wstawimy liczbę wyznaczamy za pomocą funkcji haszującej: $h(17) = 17\%10 = 7$.
- Dodajemy więc 17 do łańcucha o indeksie 7.

Metoda łańcuchowa – przykład

Zrealizujemy słownik liczb naturalnych za pomocą tablicy haszującej z łańcuchową metodą rozwiązywania kolizji i funkcją haszującą $h(x) = x\%10$.

indeks:	0	1	2	3	4	5	6	7	8	9
wartości:								17		

- Wstawmy do słownika (tablicy) liczbę 17.
- Indeks pod który wstawimy liczbę wyznaczamy za pomocą funkcji haszującej: $h(17) = 17\%10 = 7$.
- Dodajemy więc 17 do łańcucha o indeksie 7.

Metoda łańcuchowa – przykład

Zrealizujemy słownik liczb naturalnych za pomocą tablicy haszującej z łańcuchową metodą rozwiązywania kolizji i funkcją haszującą $h(x) = x\%10$.

indeks:	0	1	2	3	4	5	6	7	8	9
wartości:								17		

- Teraz wstawmy liczbę 23.
- Ponieważ $h(23) = 23\%10 = 3$, to 23 dodajemy do łańcucha o indeksie 3.

Metoda łańcuchowa – przykład

Zrealizujemy słownik liczb naturalnych za pomocą tablicy haszującej z łańcuchową metodą rozwiązywania kolizji i funkcją haszującą $h(x) = x\%10$.

indeks:	0	1	2	3	4	5	6	7	8	9
wartości:				23				17		

- Teraz wstawmy liczbę 23.
- Ponieważ $h(23) = 23\%10 = 3$, to 23 dodajemy do łańcucha o indeksie 3.

Metoda łańcuchowa – przykład

Zrealizujemy słownik liczb naturalnych za pomocą tablicy haszującej z łańcuchową metodą rozwiązywania kolizji i funkcją haszującą $h(x) = x\%10$.

indeks:	0	1	2	3	4	5	6	7	8	9
wartości:		1		23				17		

- Następnie wstawmy liczbę 1.
- $h(1) = 1\%10 = 1$.

Metoda łańcuchowa – przykład

Zrealizujemy słownik liczb naturalnych za pomocą tablicy haszującej z łańcuchową metodą rozwiązywania kolizji i funkcją haszującą $h(x) = x\%10$.

indeks:	0	1	2	3	4	5	6	7	8	9
wartości:		1		23				17		

- Wstawmy jeszcze liczbę 13.
- $h(13) = 13\%10 = 3$, więc 13 dodajemy do łańcucha o indeksie 3 (będąca tam liczba 23 nam nie przeszkadza).
- Ponieważ nie zakładamy żadnego uporządkowania łańcucha, to 13 można dodać w dowolnym jego miejscu.
- Zazwyczaj (np. jeśli łańcuchy to tablice) najszybsze będzie dodawanie na koniec, ale np. w przypadku niektórych implementacji list szybsze może być wstawianie na początek.

Metoda łańcuchowa – przykład

Zrealizujemy słownik liczb naturalnych za pomocą tablicy haszującej z łańcuchową metodą rozwiązywania kolizji i funkcją haszującą $h(x) = x\%10$.

indeks:	0	1	2	3	4	5	6	7	8	9
wartości:		1		23 13				17		

- Wstawmy jeszcze liczbę 13.
- $h(13) = 13\%10 = 3$, więc 13 dodajemy do łańcucha o indeksie 3 (będąca tam liczba 23 nam nie przeszkadza).
- Ponieważ nie zakładamy żadnego uporządkowania łańcucha, to 13 można dodać w dowolnym jego miejscu.
- Zazwyczaj (np. jeśli łańcuchy to tablice) najszybsze będzie dodawanie na koniec, ale np. w przypadku niektórych implementacji list szybsze może być wstawianie na początek.

Metoda łańcuchowa – przykład

Zrealizujemy słownik liczb naturalnych za pomocą tablicy haszującej z łańcuchową metodą rozwiązywania kolizji i funkcją haszującą $h(x) = x\%10$.

indeks:	0	1	2	3	4	5	6	7	8	9
wartości:		1		23 13				17		

- Wstawmy jeszcze liczbę 13.
- $h(13) = 13\%10 = 3$, więc 13 dodajemy do łańcucha o indeksie 3 (będąca tam liczba 23 nam nie przeszkadza).
- Ponieważ nie zakładamy żadnego uporządkowania łańcucha, to 13 można dodać w dowolnym jego miejscu.
- Zazwyczaj (np. jeśli łańcuchy to tablice) najszybsze będzie dodawanie na koniec, ale np. w przypadku niektórych implementacji list szybsze może być wstawianie na początek.

Metoda łańcuchowa – przykład

Zrealizujemy słownik liczb naturalnych za pomocą tablicy haszującej z łańcuchową metodą rozwiązywania kolizji i funkcją haszującą $h(x) = x\%10$.

indeks:	0	1	2	3	4	5	6	7	8	9
wartości:		1		23 13				17		

- Wstawmy jeszcze liczbę 13.
- $h(13) = 13\%10 = 3$, więc 13 dodajemy do łańcucha o indeksie 3 (będąca tam liczba 23 nam nie przeszkadza).
- Ponieważ nie zakładamy żadnego uporządkowania łańcucha, to 13 można dodać w dowolnym jego miejscu.
- Zazwyczaj (np. jeśli łańcuchy to tablice) najszybsze będzie dodawanie na koniec, ale np. w przypadku niektórych implementacji list szybsze może być wstawianie na początek.

Metoda łańcuchowa – przykład

Zrealizujemy słownik liczb naturalnych za pomocą tablicy haszującej z łańcuchową metodą rozwiązywania kolizji i funkcją haszującą $h(x) = x\%10$.

indeks:	0	1	2	3	4	5	6	7	8	9
wartości:		1		23 13				17		

- Wstawmy jeszcze liczby, kolejno 35, 18, 43, 67:
- 35 do łańcucha o indeksie $h(35) = 5$,
- 18 do łańcucha o indeksie $h(18) = 8$,
- 43 do łańcucha o indeksie $h(43) = 3$,
- 67 do łańcucha o indeksie $h(67) = 7$.

Metoda łańcuchowa – przykład

Zrealizujemy słownik liczb naturalnych za pomocą tablicy haszującej z łańcuchową metodą rozwiązywania kolizji i funkcją haszującą $h(x) = x\%10$.

indeks:	0	1	2	3	4	5	6	7	8	9
wartości:		1		23 13		35		17		

- Wstawmy jeszcze liczby, kolejno 35, 18, 43, 67:
- 35 do łańcucha o indeksie $h(35) = 5$,
- 18 do łańcucha o indeksie $h(18) = 8$,
- 43 do łańcucha o indeksie $h(43) = 3$,
- 67 do łańcucha o indeksie $h(67) = 7$.

Metoda łańcuchowa – przykład

Zrealizujemy słownik liczb naturalnych za pomocą tablicy haszującej z łańcuchową metodą rozwiązywania kolizji i funkcją haszującą $h(x) = x\%10$.

indeks:	0	1	2	3	4	5	6	7	8	9
wartości:		1		23 13		35		17	18	

- Wstawmy jeszcze liczby, kolejno 35, 18, 43, 67:
- 35 do łańcucha o indeksie $h(35) = 5$,
- 18 do łańcucha o indeksie $h(18) = 8$,
- 43 do łańcucha o indeksie $h(43) = 3$,
- 67 do łańcucha o indeksie $h(67) = 7$.

Metoda łańcuchowa – przykład

Zrealizujemy słownik liczb naturalnych za pomocą tablicy haszującej z łańcuchową metodą rozwiązywania kolizji i funkcją haszującą $h(x) = x\%10$.

indeks:	0	1	2	3	4	5	6	7	8	9
wartości:		1		23 13 43		35		17	18	

- Wstawmy jeszcze liczby, kolejno 35, 18, 43, 67:
- 35 do łańcucha o indeksie $h(35) = 5$,
- 18 do łańcucha o indeksie $h(18) = 8$,
- 43 do łańcucha o indeksie $h(43) = 3$,
- 67 do łańcucha o indeksie $h(67) = 7$.

Metoda łańcuchowa – przykład

Zrealizujemy słownik liczb naturalnych za pomocą tablicy haszującej z łańcuchową metodą rozwiązywania kolizji i funkcją haszującą $h(x) = x\%10$.

indeks:	0	1	2	3	4	5	6	7	8	9
wartości:		1		23		35		17	18	
				13				67		
				43						

- Wstawmy jeszcze liczby, kolejno 35, 18, 43, 67:
- 35 do łańcucha o indeksie $h(35) = 5$,
- 18 do łańcucha o indeksie $h(18) = 8$,
- 43 do łańcucha o indeksie $h(43) = 3$,
- 67 do łańcucha o indeksie $h(67) = 7$.

Metoda łańcuchowa – przykład

Zrealizujemy słownik liczb naturalnych za pomocą tablicy haszującej z łańcuchową metodą rozwiązywania kolizji i funkcją haszującą $h(x) = x\%10$.

indeks:	0	1	2	3	4	5	6	7	8	9
wartości:		1		23		35		17	18	
				13				67		
				43						

- Poszukajmy liczby 27.
- 27 może znajdować się tylko w łańcuchu o indeksie $h(27) = 7$ i dlatego wystarczy przeszukać tylko ten łańcuch.
- Wystarczą więc 2 porównania (ogólnie liczba porównań nie przekroczy długości przeszukiwanego łańcucha) by stwierdzić, że 27 nie ma w słowniku.

Metoda łańcuchowa – przykład

Zrealizujemy słownik liczb naturalnych za pomocą tablicy haszującej z łańcuchową metodą rozwiązywania kolizji i funkcją haszującą $h(x) = x\%10$.

indeks:	0	1	2	3	4	5	6	7	8	9
wartości:		1		23		35		17	18	
				13				67		
				43						

- Poszukajmy liczby 27.
- 27 może znajdować się tylko w łańcuchu o indeksie $h(27) = 7$ i dlatego wystarczy przeszukać tylko ten łańcuch.
- Wystarczą więc 2 porównania (ogólnie liczba porównań nie przekroczy długości przeszukiwanego łańcucha) by stwierdzić, że 27 nie ma w słowniku.

Metoda łańcuchowa – przykład

Zrealizujemy słownik liczb naturalnych za pomocą tablicy haszującej z łańcuchową metodą rozwiązywania kolizji i funkcją haszującą $h(x) = x\%10$.

indeks:	0	1	2	3	4	5	6	7	8	9
wartości:		1		23		35		17	18	
				13				67		
				43						

- Poszukajmy liczby 27.
- 27 może znajdować się tylko w łańcuchu o indeksie $h(27) = 7$ i dlatego wystarczy przeszukać tylko ten łańcuch.
- Wystarczą więc 2 porównania (ogólnie liczba porównań nie przekroczy długości przeszukiwanego łańcucha) by stwierdzić, że 27 nie ma w słowniku.

Metoda łańcuchowa – przykład

Zrealizujemy słownik liczb naturalnych za pomocą tablicy haszującej z łańcuchową metodą rozwiązywania kolizji i funkcją haszującą $h(x) = x\%10$.

indeks:	0	1	2	3	4	5	6	7	8	9
wartości:		1		23		35		17	18	
				13				67		
				43						

- Poszukajmy liczby 12.
- W tym celu przeszukujemy jedynie łańcuch o indeksie $h(12) = 2$ i, ponieważ ten łańcuch jest pusty, szybko stwierdzamy, że 12 nie ma w słowniku.

Metoda łańcuchowa – przykład

Zrealizujemy słownik liczb naturalnych za pomocą tablicy haszującej z łańcuchową metodą rozwiązywania kolizji i funkcją haszującą $h(x) = x\%10$.

indeks:	0	1	2	3	4	5	6	7	8	9
wartości:		1		23		35		17	18	
				13				67		
				43						

- Poszukajmy liczby 12.
- W tym celu przeszukujemy jedynie łańcuch o indeksie $h(12) = 2$ i, ponieważ ten łańcuch jest pusty, szybko stwierdzamy, że 12 nie ma w słowniku.

Metoda łańcuchowa – przykład

Zrealizujemy słownik liczb naturalnych za pomocą tablicy haszującej z łańcuchową metodą rozwiązywania kolizji i funkcją haszującą $h(x) = x\%10$.

indeks:	0	1	2	3	4	5	6	7	8	9
wartości:		1		23		35		17	18	
				13				67		
				43						

- Poszukajmy jeszcze liczby 13.
- Przeszukujemy jedynie łańcuch o indeksie $h(13) = 3$.
- Tam, po wykonaniu 2 porównań (ogólnie ich liczba nie przekroczy długości łańcucha), odnajdujemy 13.

Metoda łańcuchowa – przykład

Zrealizujemy słownik liczb naturalnych za pomocą tablicy haszującej z łańcuchową metodą rozwiązywania kolizji i funkcją haszującą $h(x) = x\%10$.

indeks:	0	1	2	3	4	5	6	7	8	9
wartości:		1		23		35		17	18	
				13				67		
				43						

- Poszukajmy jeszcze liczby 13.
- Przeszukujemy jedynie łańcuch o indeksie $h(13) = 3$.
- Tam, po wykonaniu 2 porównań (ogólnie ich liczba nie przekroczy długości łańcucha), odnajdujemy 13.

Metoda łańcuchowa – przykład

Zrealizujemy słownik liczb naturalnych za pomocą tablicy haszującej z łańcuchową metodą rozwiązywania kolizji i funkcją haszującą $h(x) = x\%10$.

indeks:	0	1	2	3	4	5	6	7	8	9
wartości:		1		23		35		17	18	
				13				67		
				43						

- Poszukajmy jeszcze liczby 13.
- Przeszukujemy jedynie łańcuch o indeksie $h(13) = 3$.
- Tam, po wykonaniu 2 porównań (ogólnie ich liczba nie przekroczy długości łańcucha), odnajdujemy 13.

Metoda łańcuchowa – przykład

Zrealizujemy słownik liczb naturalnych za pomocą tablicy haszującej z łańcuchową metodą rozwiązywania kolizji i funkcją haszującą $h(x) = x\%10$.

indeks:	0	1	2	3	4	5	6	7	8	9
wartości:		1		23		35		17	18	
				13				67		
				43						

- Na koniec, usuńmy ze słownika liczbę 23.
- Wpierw odnajdujemy ją w odpowiednim łańcuchu (o indeksie $h(23) = 3$), wykonując przy tym co najwyżej tyle porównań, ile elementów zawiera ten łańcuch.
- Znając jej miejsce w łańcuchu, usuwamy ją stamtąd w (zamortyzowanym) czasie $O(1)$, co jest banalne gdy łańcuchy są listami. Jeśli są zaś tablicami, to można nadpisać usuwany element ostatnim (albo je zamienić) i usunąć ten ostatni.

Metoda łańcuchowa – przykład

Zrealizujemy słownik liczb naturalnych za pomocą tablicy haszującej z łańcuchową metodą rozwiązywania kolizji i funkcją haszującą $h(x) = x\%10$.

indeks:	0	1	2	3	4	5	6	7	8	9
wartości:		1		23		35		17	18	
				13				67		
				43						

- Na koniec, usuńmy ze słownika liczbę 23.
- Wpierw odnajdujemy ją w odpowiednim łańcuchu (o indeksie $h(23) = 3$), wykonując przy tym co najwyżej tyle porównań, ile elementów zawiera ten łańcuch.
- Znając jej miejsce w łańcuchu, usuwamy ją stamtąd w (zamortyzowanym) czasie $O(1)$, co jest banalne gdy łańcuchy są listami. Jeśli są zaś tablicami, to można nadpisać usuwany element ostatnim (albo je zamienić) i usunąć ten ostatni.

Metoda łańcuchowa – przykład

Zrealizujemy słownik liczb naturalnych za pomocą tablicy haszującej z łańcuchową metodą rozwiązywania kolizji i funkcją haszującą $h(x) = x\%10$.

indeks:	0	1	2	3	4	5	6	7	8	9
wartości:		1		23		35		17	18	
				13				67		
				43						

- Na koniec, usuńmy ze słownika liczbę 23.
- Wpierw odnajdujemy ją w odpowiednim łańcuchu (o indeksie $h(23) = 3$), wykonując przy tym co najwyżej tyle porównań, ile elementów zawiera ten łańcuch.
- Znając jej miejsce w łańcuchu, usuwamy ją stamtąd w (zamortyzowanym) czasie $O(1)$, co jest banalne gdy łańcuchy są listami. Jeśli są zaś tablicami, to można nadpisać usuwany element ostatnim (albo je zamienić) i usunąć ten ostatni.

Metoda łańcuchowa – przykład

Zrealizujemy słownik liczb naturalnych za pomocą tablicy haszującej z łańcuchową metodą rozwiązywania kolizji i funkcją haszującą $h(x) = x\%10$.

indeks:	0	1	2	3	4	5	6	7	8	9
wartości:		1		43		35		17	18	
				13				67		
				43						

- Na koniec, usuńmy ze słownika liczbę 23.
- Wpierw odnajdujemy ją w odpowiednim łańcuchu (o indeksie $h(23) = 3$), wykonując przy tym co najwyżej tyle porównań, ile elementów zawiera ten łańcuch.
- Znając jej miejsce w łańcuchu, usuwamy ją stamtąd w (zamortyzowanym) czasie $O(1)$, co jest banalne gdy łańcuchy są listami. Jeśli są zaś tablicami, to można nadpisać usuwany element ostatnim (albo je zamienić) i usunąć ten ostatni.

Metoda łańcuchowa – przykład

Zrealizujemy słownik liczb naturalnych za pomocą tablicy haszującej z łańcuchową metodą rozwiązywania kolizji i funkcją haszującą $h(x) = x\%10$.

indeks:	0	1	2	3	4	5	6	7	8	9
wartości:		1		43 13		35		17 67	18	

- Na koniec, usuńmy ze słownika liczbę 23.
- Wpierw odnajdujemy ją w odpowiednim łańcuchu (o indeksie $h(23) = 3$), wykonując przy tym co najwyżej tyle porównań, ile elementów zawiera ten łańcuch.
- Znając jej miejsce w łańcuchu, usuwamy ją stamtąd w (zamortyzowanym) czasie $O(1)$, co jest banalne gdy łańcuchy są listami. Jeśli są zaś tablicami, to można nadpisać usuwany element ostatnim (albo je zamienić) i usunąć ten ostatni.

Metoda łańcuchowa – złożoność czasowa

- `find` wyznacza łańcuch za pomocą funkcji h (zakładamy, że w czasie $O(1)$) i następnie przeszukuje go w (oczekiwanym i pesymistycznym) czasie liniowym względem jego długości.
- Oczekiwana długość łańcucha zależy od funkcji h.
- Najgorsza funkcja h stale wskazuje ten sam łańcuch. Zawiera on wszystkie elementy więc złożoność czasowa `find` to $\Theta(s)$.
- Najlepsza h rozmieszcza elementy równomiernie – losowo wybrany element trafia do każdego z łańcuchów z jednakowym prawdopodobieństwem. Wtedy, każdy z łańcuchów ma oczekiwaną długość $\frac{s}{n}$, zaś złożoność `find` wynosi $\Theta(\frac{s}{n} + 1)$.
- `insert` (gdy duplikaty nie są dozwolone) oraz `delete` (szukające gdzie element do skasowania jest w łańcuchu) mają złożoność podobną do `find`.

Oznaczenia:

- s – liczba elementów w słowniku,
- n – liczba łańcuchów,

Metoda łańcuchowa – złożoność czasowa

- `find` wyznacza łańcuch za pomocą funkcji h (zakładamy, że w czasie $O(1)$) i następnie przeszukuje go w (oczekiwanym i pesymistycznym) czasie liniowym względem jego długości.
- Oczekiwana długość łańcucha zależy od funkcji h.
- Najgorsza funkcja h stale wskazuje ten sam łańcuch. Zawiera on wszystkie elementy więc złożoność czasowa `find` to $\Theta(s)$.
- Najlepsza h rozmieszcza elementy równomiernie – losowo wybrany element trafia do każdego z łańcuchów z jednakowym prawdopodobieństwem. Wtedy, każdy z łańcuchów ma oczekiwaną długość $\frac{s}{n}$, zaś złożoność `find` wynosi $\Theta(\frac{s}{n} + 1)$.
- `insert` (gdy duplikaty nie są dozwolone) oraz `delete` (szukające gdzie element do skasowania jest w łańcuchu) mają złożoność podobną do `find`.

Oznaczenia:

- s – liczba elementów w słowniku,
- n – liczba łańcuchów,

Metoda łańcuchowa – złożoność czasowa

- `find` wyznacza łańcuch za pomocą funkcji h (zakładamy, że w czasie $O(1)$) i następnie przeszukuje go w (oczekiwanym i pesymistycznym) czasie liniowym względem jego długości.
- Oczekiwana długość łańcucha zależy od funkcji h.
- Najgorsza funkcja h stale wskazuje ten sam łańcuch. Zawiera on wszystkie elementy więc złożoność czasowa `find` to $\Theta(s)$.
- Najlepsza h rozmieszcza elementy równomiernie – losowo wybrany element trafia do każdego z łańcuchów z jednakowym prawdopodobieństwem. Wtedy, każdy z łańcuchów ma oczekiwaną długość $\frac{s}{n}$, zaś złożoność `find` wynosi $\Theta(\frac{s}{n}+1)$.
- `insert` (gdy duplikaty nie są dozwolone) oraz `delete` (szukające gdzie element do skasowania jest w łańcuchu) mają złożoność podobną do `find`.

Oznaczenia:

- s – liczba elementów w słowniku,
- n – liczba łańcuchów,

Metoda łańcuchowa – złożoność czasowa

- `find` wyznacza łańcuch za pomocą funkcji h (zakładamy, że w czasie $O(1)$) i następnie przeszukuje go w (oczekiwanym i pesymistycznym) czasie liniowym względem jego długości.
- Oczekiwana długość łańcucha zależy od funkcji h.
- Najgorsza funkcja h stale wskazuje ten sam łańcuch. Zawiera on wszystkie elementy więc złożoność czasowa `find` to $\Theta(s)$.
- Najlepsza h rozmieszcza elementy równomiernie – losowo wybrany element trafia do każdego z łańcuchów z jednakowym prawdopodobieństwem. Wtedy, każdy z łańcuchów ma oczekiwaną długość $\frac{s}{n}$, zaś złożoność `find` wynosi $\Theta(\frac{s}{n} + 1)$.
- `insert` (gdy duplikaty nie są dozwolone) oraz `delete` (szukające gdzie element do skasowania jest w łańcuchu) mają złożoność podobną do `find`.

Oznaczenia:

- s – liczba elementów w słowniku,
- n – liczba łańcuchów,

Metoda łańcuchowa – złożoność czasowa

- `find` wyznacza łańcuch za pomocą funkcji h (zakładamy, że w czasie $O(1)$) i następnie przeszukuje go w (oczekiwanym i pesymistycznym) czasie liniowym względem jego długości.
- Oczekiwana długość łańcucha zależy od funkcji h.
- Najgorsza funkcja h stale wskazuje ten sam łańcuch. Zawiera on wszystkie elementy więc złożoność czasowa `find` to $\Theta(s)$.
- Najlepsza h rozmieszcza elementy równomiernie – losowo wybrany element trafia do każdego z łańcuchów z jednakowym prawdopodobieństwem. Wtedy, każdy z łańcuchów ma oczekiwaną długość $\frac{s}{n}$, zaś złożoność `find` wynosi $\Theta(\frac{s}{n} + 1)$.
- `insert` (gdy duplikaty nie są dozwolone) oraz `delete` (szukające gdzie element do skasowania jest w łańcuchu) mają złożoność podobną do `find`.

Oznaczenia:

- s – liczba elementów w słowniku,
- n – liczba łańcuchów,

Metoda łańcuchowa a stała złożoność czasowa

- Przy założeniu równomiernego rozkładania elementów przez h, oczekiwana złożoność operacji słownikowych wynosi $\Theta(\frac{s}{n} + 1)$.
- Gdy n jest stałe to $\frac{s}{n} = \Theta(s)$. Tak jak dla realizacji słownika za pomocą nieuporządkowanej tablicy, operacje działają w czasie liniowym, z tym, że z mniejszym współczynnikiem $\frac{1}{n}$.
- Oczekiwaną złożoność $O(1)$ można uzyskać poprzez zagwarantowanie, że $\frac{s}{n} \leq M$, gdzie M jest stałą.
- By to uczynić, należy zwiększyć liczbę łańcuchów n, jeśli s stanie się większe od nM.
- Wymaga to jednak tzw. rehaszowania, czyli przepisania elementów do większej tablicy (i wyliczenia im nowych wartości h, uwzględniających nowe n).

Oznaczenia:

- s – liczba elementów w słowniku,
- n – liczba łańcuchów.

Metoda łańcuchowa a stała złożoność czasowa

- Przy założeniu równomiernego rozkładania elementów przez h, oczekiwana złożoność operacji słownikowych wynosi $\Theta(\frac{s}{n} + 1)$.
- Gdy n jest stałe to $\frac{s}{n} = \Theta(s)$. Tak jak dla realizacji słownika za pomocą nieuporządkowanej tablicy, operacje działają w czasie liniowym, z tym, że z mniejszym współczynnikiem $\frac{1}{n}$.
- Oczekiwaną złożoność $O(1)$ można uzyskać poprzez zagwarantowanie, że $\frac{s}{n} \leq M$, gdzie M jest stałą.
- By to uczynić, należy zwiększyć liczbę łańcuchów n, jeśli s stanie się większe od nM.
- Wymaga to jednak tzw. rehaszowania, czyli przepisania elementów do większej tablicy (i wyliczenia im nowych wartości h, uwzględniających nowe n).

Oznaczenia:

- s – liczba elementów w słowniku,
- n – liczba łańcuchów.

Metoda łańcuchowa a stała złożoność czasowa

- Przy założeniu równomiernego rozkładania elementów przez h, oczekiwana złożoność operacji słownikowych wynosi $\Theta(\frac{s}{n} + 1)$.
- Gdy n jest stałe to $\frac{s}{n} = \Theta(s)$. Tak jak dla realizacji słownika za pomocą nieuporządkowanej tablicy, operacje działają w czasie liniowym, z tym, że z mniejszym współczynnikiem $\frac{1}{n}$.
- Oczekiwaną złożoność $O(1)$ można uzyskać poprzez zagwarantowanie, że $\frac{s}{n} \leq M$, gdzie M jest stałą.
- By to uczynić, należy zwiększyć liczbę łańcuchów n, jeśli s stanie się większe od nM.
- Wymaga to jednak tzw. rehaszowania, czyli przepisania elementów do większej tablicy (i wyliczenia im nowych wartości h, uwzględniających nowe n).

Oznaczenia:

- s – liczba elementów w słowniku,
- n – liczba łańcuchów.

Metoda łańcuchowa a stała złożoność czasowa

- Przy założeniu równomiernego rozkładania elementów przez h, oczekiwana złożoność operacji słownikowych wynosi $\Theta(\frac{s}{n} + 1)$.
- Gdy n jest stałe to $\frac{s}{n} = \Theta(s)$. Tak jak dla realizacji słownika za pomocą nieuporządkowanej tablicy, operacje działają w czasie liniowym, z tym, że z mniejszym współczynnikiem $\frac{1}{n}$.
- Oczekiwaną złożoność $O(1)$ można uzyskać poprzez zagwarantowanie, że $\frac{s}{n} \leq M$, gdzie M jest stałą.
- By to uczynić, należy zwiększyć liczbę łańcuchów n, jeśli s stanie się większe od nM.
- Wymaga to jednak tzw. rehaszowania, czyli przepisania elementów do większej tablicy (i wyliczenia im nowych wartości h, uwzględniających nowe n).

Oznaczenia:

- s – liczba elementów w słowniku,
- n – liczba łańcuchów.

Metoda łańcuchowa a stała złożoność czasowa

- Przy założeniu równomiernego rozkładania elementów przez h, oczekiwana złożoność operacji słownikowych wynosi $\Theta(\frac{s}{n} + 1)$.
- Gdy n jest stałe to $\frac{s}{n} = \Theta(s)$. Tak jak dla realizacji słownika za pomocą nieuporządkowanej tablicy, operacje działają w czasie liniowym, z tym, że z mniejszym współczynnikiem $\frac{1}{n}$.
- Oczekiwaną złożoność $O(1)$ można uzyskać poprzez zagwarantowanie, że $\frac{s}{n} \leq M$, gdzie M jest stałą.
- By to uczynić, należy zwiększyć liczbę łańcuchów n, jeśli s stanie się większe od nM.
- Wymaga to jednak tzw. rehaszowania, czyli przepisania elementów do większej tablicy (i wyliczenia im nowych wartości h, uwzględniających nowe n).

Oznaczenia:

- s – liczba elementów w słowniku,
- n – liczba łańcuchów.

Rehaszowanie w metodzie łańcuchowej – przykład

Tablica używająca 5 łańcuchów i funkcji haszującej $h(x) = x\%5$, w której jest 15 elementów, czyli średnio $15/5 = 3$ na łańcuch:

indeks:	0	1	2	3	4
wartości:	25	51	17	93	54
	40	16		8	29
	5	86		38	19
	15			43	

Tablica z tą samą zawartością, po rehaszowaniu (przepisaniu) do 10 łańcuchów, z funkcją haszującą $h(x) = x\%10$ (teraz przypada $15/10 = 1{,}5$ elementu na łańcuch):

indeks:	0	1	2	3	4	5	6	7	8	9
wartości:	40	51		93	54	25	16	17	8	29
	15			43		5	86		38	19

Metoda łańcuchowa a czasowa złożoność rehaszowania

Ponieważ rehaszowanie jest kosztowne (złożoność czasowa: $\Theta(s)$), to n należy zwiększyć od razu na tyle istotnie, by nie rehaszować zbyt często, np. podwajanie n pozwala uzyskać zamortyzowaną (przypadającą na pojedynczy `insert`) złożoność $O(1)$. Dowód:

- Rozważmy słownik w który, za pomocą s operacji `insert`, wstawiono s elementów.
- Niech $x \leq s$ – liczba elementów przepisanych przez ostatnie rehaszowanie. Poprzednie przepisały $\frac{x}{2}$, $\frac{x}{4}$, itd. elementów.
- Sumaryczna liczba przepisań: $X = x + \frac{x}{2} + \frac{x}{4} + \ldots \leq 2x \leq 2s$.
- Czyli, na pojedynczy `insert` przypada ich $\frac{X}{s} \leq \frac{2s}{s} = 2$.

Operacja `delete` może czasami zmniejszyć liczbę łańcuchów n. Trzeba jednak uważać, by żadne ciągi operacji `insert` i `delete` nie dokonywały zbyt wielu rehaszowań. W naszym wypadku, by uzyskać zamortyzowaną złożoność $O(1)$, można przedzielić n przez 2 gdy s spadnie poniżej $nM/4$. Wymagane po tym przepisanie s elementów będzie poprzedzone co najmniej s operacjami `delete`, więc na 1 `delete` przypadnie przepisanie co najwyżej 1 elementu.

Metoda łańcuchowa a czasowa złożoność rehaszowania

Ponieważ rehaszowanie jest kosztowne (złożoność czasowa: $\Theta(s)$), to n należy zwiększyć od razu na tyle istotnie, by nie rehaszować zbyt często, np. podwajanie n pozwala uzyskać zamortyzowaną (przypadającą na pojedynczy `insert`) złożoność $O(1)$. Dowód:

- Rozważmy słownik w który, za pomocą s operacji `insert`, wstawiono s elementów.
- Niech $x \leq s$ – liczba elementów przepisanych przez ostatnie rehaszowanie. Poprzednie przepisały $\frac{x}{2}$, $\frac{x}{4}$, itd. elementów.
- Sumaryczna liczba przepisań: $X = x + \frac{x}{2} + \frac{x}{4} + \ldots \leq 2x \leq 2s$.
- Czyli, na pojedynczy `insert` przypada ich $\frac{X}{s} \leq \frac{2s}{s} = 2$.

Operacja `delete` może czasami zmniejszyć liczbę łańcuchów n. Trzeba jednak uważać, by żadne ciągi operacji `insert` i `delete` nie dokonywały zbyt wielu rehaszowań. W naszym wypadku, by uzyskać zamortyzowaną złożoność $O(1)$, można przedzielić n przez 2 gdy s spadnie poniżej $nM/4$. Wymagane po tym przepisanie s elementów będzie poprzedzone co najmniej s operacjami `delete`, więc na 1 `delete` przypadnie przepisanie co najwyżej 1 elementu.

Metoda łańcuchowa a czasowa złożoność rehaszowania

Ponieważ rehaszowanie jest kosztowne (złożoność czasowa: $\Theta(s)$), to n należy zwiększyć od razu na tyle istotnie, by nie rehaszować zbyt często, np. podwajanie n pozwala uzyskać zamortyzowaną (przypadającą na pojedynczy `insert`) złożoność $O(1)$. Dowód:

- Rozważmy słownik w który, za pomocą s operacji `insert`, wstawiono s elementów.
  - Niech $x \leq s$ – liczba elementów przepisanych przez ostatnie rehaszowanie. Poprzednie przepisały $\frac{x}{2}$, $\frac{x}{4}$, itd. elementów.
  - Sumaryczna liczba przepisań: $X = x + \frac{x}{2} + \frac{x}{4} + \ldots \leq 2x \leq 2s$.
  - Czyli, na pojedynczy `insert` przypada ich $\frac{X}{s} \leq \frac{2s}{s} = 2$.

Operacja `delete` może czasami zmniejszyć liczbę łańcuchów n. Trzeba jednak uważać, by żadne ciągi operacji `insert` i `delete` nie dokonywały zbyt wielu rehaszowań. W naszym wypadku, by uzyskać zamortyzowaną złożoność $O(1)$, można przedzielić n przez 2 gdy s spadnie poniżej $nM/4$. Wymagane po tym przepisanie s elementów będzie poprzedzone co najmniej s operacjami `delete`, więc na 1 `delete` przypadnie przepisanie co najwyżej 1 elementu.

Metoda łańcuchowa a czasowa złożoność rehaszowania

Ponieważ rehaszowanie jest kosztowne (złożoność czasowa: $\Theta(s)$), to n należy zwiększyć od razu na tyle istotnie, by nie rehaszować zbyt często, np. podwajanie n pozwala uzyskać zamortyzowaną (przypadającą na pojedynczy `insert`) złożoność $O(1)$. Dowód:

- Rozważmy słownik w który, za pomocą s operacji `insert`, wstawiono s elementów.
- Niech $x \leq s$ – liczba elementów przepisanych przez ostatnie rehaszowanie. Poprzednie przepisały $\frac{x}{2}$, $\frac{x}{4}$, itd. elementów.
- Sumaryczna liczba przepisań: $X = x + \frac{x}{2} + \frac{x}{4} + \ldots \leq 2x \leq 2s$.
- Czyli, na pojedynczy `insert` przypada ich $\frac{X}{s} \leq \frac{2s}{s} = 2$.

Operacja `delete` może czasami zmniejszyć liczbę łańcuchów n. Trzeba jednak uważać, by żadne ciągi operacji `insert` i `delete` nie dokonywały zbyt wielu rehaszowań. W naszym wypadku, by uzyskać zamortyzowaną złożoność $O(1)$, można przedzielić n przez 2 gdy s spadnie poniżej $nM/4$. Wymagane po tym przepisanie s elementów będzie poprzedzone co najmniej s operacjami `delete`, więc na 1 `delete` przypadnie przepisanie co najwyżej 1 elementu.

Metoda łańcuchowa a czasowa złożoność rehaszowania

Ponieważ rehaszowanie jest kosztowne (złożoność czasowa: $\Theta(s)$), to n należy zwiększyć od razu na tyle istotnie, by nie rehaszować zbyt często, np. podwajanie n pozwala uzyskać zamortyzowaną (przypadającą na pojedynczy `insert`) złożoność $O(1)$. Dowód:

- Rozważmy słownik w który, za pomocą s operacji `insert`, wstawiono s elementów.
- Niech $x \leq s$ – liczba elementów przepisanych przez ostatnie rehaszowanie. Poprzednie przepisały $\frac{x}{2}$, $\frac{x}{4}$, itd. elementów.
- Sumaryczna liczba przepisań: $X = x + \frac{x}{2} + \frac{x}{4} + \ldots \leq 2x \leq 2s$.
- Czyli, na pojedynczy `insert` przypada ich $\frac{X}{s} \leq \frac{2s}{s} = 2$.

Operacja `delete` może czasami zmniejszyć liczbę łańcuchów n. Trzeba jednak uważać, by żadne ciągi operacji `insert` i `delete` nie dokonywały zbyt wielu rehaszowań. W naszym wypadku, by uzyskać zamortyzowaną złożoność $O(1)$, można przedzielić n przez 2 gdy s spadnie poniżej $nM/4$. Wymagane po tym przepisanie s elementów będzie poprzedzone co najmniej s operacjami `delete`, więc na 1 `delete` przypadnie przepisanie co najwyżej 1 elementu.

Metoda łańcuchowa a czasowa złożoność rehaszowania

Ponieważ rehaszowanie jest kosztowne (złożoność czasowa: $\Theta(s)$), to n należy zwiększyć od razu na tyle istotnie, by nie rehaszować zbyt często, np. podwajanie n pozwala uzyskać zamortyzowaną (przypadającą na pojedynczy `insert`) złożoność $O(1)$. Dowód:

- Rozważmy słownik w który, za pomocą s operacji `insert`, wstawiono s elementów.
- Niech $x \leq s$ – liczba elementów przepisanych przez ostatnie rehaszowanie. Poprzednie przepisały $\frac{x}{2}$, $\frac{x}{4}$, itd. elementów.
- Sumaryczna liczba przepisań: $X = x + \frac{x}{2} + \frac{x}{4} + \ldots \leq 2x \leq 2s$.
- Czyli, na pojedynczy `insert` przypada ich $\frac{X}{s} \leq \frac{2s}{s} = 2$.

Operacja `delete` może czasami zmniejszyć liczbę łańcuchów n. Trzeba jednak uważać, by żadne ciągi operacji `insert` i `delete` nie dokonywały zbyt wielu rehaszowań. W naszym wypadku, by uzyskać zamortyzowaną złożoność $O(1)$, można przedzielić n przez 2 gdy s spadnie poniżej $nM/4$. Wymagane po tym przepisanie s elementów będzie poprzedzone co najmniej s operacjami `delete`, więc na 1 `delete` przypadnie przepisanie co najwyżej 1 elementu.

Metoda łańcuchowa a czasowa złożoność rehaszowania

Ponieważ rehaszowanie jest kosztowne (złożoność czasowa: $\Theta(s)$), to n należy zwiększyć od razu na tyle istotnie, by nie rehaszować zbyt często, np. podwajanie n pozwala uzyskać zamortyzowaną (przypadającą na pojedynczy `insert`) złożoność $O(1)$. Dowód:

- Rozważmy słownik w który, za pomocą s operacji `insert`, wstawiono s elementów.
- Niech $x \leq s$ – liczba elementów przepisanych przez ostatnie rehaszowanie. Poprzednie przepisały $\frac{x}{2}$, $\frac{x}{4}$, itd. elementów.
- Sumaryczna liczba przepisań: $X = x + \frac{x}{2} + \frac{x}{4} + \ldots \leq 2x \leq 2s$.
- Czyli, na pojedynczy `insert` przypada ich $\frac{X}{s} \leq \frac{2s}{s} = 2$.

Operacja `delete` może czasami zmniejszyć liczbę łańcuchów n. Trzeba jednak uważać, by żadne ciągi operacji `insert` i `delete` nie dokonywały zbyt wielu rehaszowań. W naszym wypadku, by uzyskać zamortyzowaną złożoność $O(1)$, można przedzielić n przez 2 gdy s spadnie poniżej $nM/4$. Wymagane po tym przepisanie s elementów będzie poprzedzone co najmniej s operacjami `delete`, więc na 1 `delete` przypadnie przepisanie co najwyżej 1 elementu.

Metoda łańcuchowa a czasowa złożoność rehaszowania

Ponieważ rehaszowanie jest kosztowne (złożoność czasowa: $\Theta(s)$), to n należy zwiększyć od razu na tyle istotnie, by nie rehaszować zbyt często, np. podwajanie n pozwala uzyskać zamortyzowaną (przypadającą na pojedynczy `insert`) złożoność $O(1)$. Dowód:

- Rozważmy słownik w który, za pomocą s operacji `insert`, wstawiono s elementów.
- Niech $x \leq s$ – liczba elementów przepisanych przez ostatnie rehaszowanie. Poprzednie przepisały $\frac{x}{2}$, $\frac{x}{4}$, itd. elementów.
- Sumaryczna liczba przepisań: $X = x + \frac{x}{2} + \frac{x}{4} + \ldots \leq 2x \leq 2s$.
- Czyli, na pojedynczy `insert` przypada ich $\frac{X}{s} \leq \frac{2s}{s} = 2$.

Operacja `delete` może czasami zmniejszyć liczbę łańcuchów n. Trzeba jednak uważać, by żadne ciągi operacji `insert` i `delete` nie dokonywały zbyt wielu rehaszowań. W naszym wypadku, by uzyskać zamortyzowaną złożoność $O(1)$, można przedzielić n przez 2 gdy s spadnie poniżej $nM/4$. Wymagane po tym przepisanie s elementów będzie poprzedzone co najmniej s operacjami `delete`, więc na 1 `delete` przypadnie przepisanie co najwyżej 1 elementu.

Metoda łańcuchowa a czasowa złożoność rehaszowania

Ponieważ rehaszowanie jest kosztowne (złożoność czasowa: $\Theta(s)$), to n należy zwiększyć od razu na tyle istotnie, by nie rehaszować zbyt często, np. podwajanie n pozwala uzyskać zamortyzowaną (przypadającą na pojedynczy `insert`) złożoność $O(1)$. Dowód:

- Rozważmy słownik w który, za pomocą s operacji `insert`, wstawiono s elementów.
- Niech $x \leq s$ – liczba elementów przepisanych przez ostatnie rehaszowanie. Poprzednie przepisały $\frac{x}{2}$, $\frac{x}{4}$, itd. elementów.
- Sumaryczna liczba przepisań: $X = x + \frac{x}{2} + \frac{x}{4} + \ldots \leq 2x \leq 2s$.
- Czyli, na pojedynczy `insert` przypada ich $\frac{X}{s} \leq \frac{2s}{s} = 2$.

Operacja `delete` może czasami zmniejszyć liczbę łańcuchów n. Trzeba jednak uważać, by żadne ciągi operacji `insert` i `delete` nie dokonywały zbyt wielu rehaszowań. W naszym wypadku, by uzyskać zamortyzowaną złożoność $O(1)$, można przedzielić n przez 2 gdy s spadnie poniżej $nM/4$. Wymagane po tym przepisanie s elementów będzie poprzedzone co najmniej s operacjami `delete`, więc na 1 `delete` przypadnie przepisanie co najwyżej 1 elementu.

Metoda łańcuchowa a czasowa złożoność rehaszowania

Ponieważ rehaszowanie jest kosztowne (złożoność czasowa: $\Theta(s)$), to n należy zwiększyć od razu na tyle istotnie, by nie rehaszować zbyt często, np. podwajanie n pozwala uzyskać zamortyzowaną (przypadającą na pojedynczy `insert`) złożoność $O(1)$. Dowód:

- Rozważmy słownik w który, za pomocą s operacji `insert`, wstawiono s elementów.
- Niech $x \leq s$ – liczba elementów przepisanych przez ostatnie rehaszowanie. Poprzednie przepisały $\frac{x}{2}$, $\frac{x}{4}$, itd. elementów.
- Sumaryczna liczba przepisań: $X = x + \frac{x}{2} + \frac{x}{4} + \ldots \leq 2x \leq 2s$.
- Czyli, na pojedynczy `insert` przypada ich $\frac{X}{s} \leq \frac{2s}{s} = 2$.

Operacja `delete` może czasami zmniejszyć liczbę łańcuchów n. Trzeba jednak uważać, by żadne ciągi operacji `insert` i `delete` nie dokonywały zbyt wielu rehaszowań. W naszym wypadku, by uzyskać zamortyzowaną złożoność $O(1)$, można przedzielić n przez 2 gdy s spadnie poniżej $nM/4$. Wymagane po tym przepisanie s elementów będzie poprzedzone co najmniej s operacjami `delete`, więc na 1 `delete` przypadnie przepisanie co najwyżej 1 elementu.

Metoda łańcuchowa a czasowa złożoność rehaszowania

Ponieważ rehaszowanie jest kosztowne (złożoność czasowa: $\Theta(s)$), to n należy zwiększyć od razu na tyle istotnie, by nie rehaszować zbyt często, np. podwajanie n pozwala uzyskać zamortyzowaną (przypadającą na pojedynczy `insert`) złożoność $O(1)$. Dowód:

- Rozważmy słownik w który, za pomocą s operacji `insert`, wstawiono s elementów.
- Niech $x \leq s$ – liczba elementów przepisanych przez ostatnie rehaszowanie. Poprzednie przepisały $\frac{x}{2}$, $\frac{x}{4}$, itd. elementów.
- Sumaryczna liczba przepisań: $X = x + \frac{x}{2} + \frac{x}{4} + \ldots \leq 2x \leq 2s$.
- Czyli, na pojedynczy `insert` przypada ich $\frac{X}{s} \leq \frac{2s}{s} = 2$.

Operacja `delete` może czasami zmniejszyć liczbę łańcuchów n. Trzeba jednak uważać, by żadne ciągi operacji `insert` i `delete` nie dokonywały zbyt wielu rehaszowań. W naszym wypadku, by uzyskać zamortyzowaną złożoność $O(1)$, można przedzielić n przez 2 gdy s spadnie poniżej $nM/4$. Wymagane po tym przepisanie s elementów będzie poprzedzone co najmniej s operacjami `delete`, więc na 1 `delete` przypadnie przepisanie co najwyżej 1 elementu.

Pseudokod – założenia – klucze

Dalej przyjmiemy, że klucz elementu wyznaczany jest za pomocą funkcji `key`.

To założenie zapewnia ogólność, gdyż:

- gdy klucz jest tożsamy z elementem, to definiujemy:
  ```
  fun key(element): return element
  ```
- gdy klucz jest polem elementu, to definiujemy:
  ```
  fun key(element): return element.key
  ```
- gdy element jest parą: klucz, wartość, definiujemy:
  ```
  fun key(element): return element[0]
  ```

Pseudokod – założenia – klucze

Dalej przyjmiemy, że klucz elementu wyznaczany jest za pomocą funkcji `key`.

To założenie zapewnia ogólność, gdyż:

- gdy klucz jest tożsamy z elementem, to definiujemy:
  ```
  fun key(element): return element
  ```
- gdy klucz jest polem elementu, to definiujemy:
  ```
  fun key(element): return element.key
  ```
- gdy element jest parą: klucz, wartość, definiujemy:
  ```
  fun key(element): return element[0]
  ```

Pseudokod – założenia – klucze

Dalej przyjmiemy, że klucz elementu wyznaczany jest za pomocą funkcji `key`.

To założenie zapewnia ogólność, gdyż:

- gdy klucz jest tożsamy z elementem, to definiujemy:
  ```
  fun key(element): return element
  ```
- gdy klucz jest polem elementu, to definiujemy:
  ```
  fun key(element): return element.key
  ```
- gdy element jest parą: klucz, wartość, definiujemy:
  ```
  fun key(element): return element[0]
  ```

Pseudokod – założenia – klucze

Dalej przyjmiemy, że klucz elementu wyznaczany jest za pomocą funkcji `key`.

To założenie zapewnia ogólność, gdyż:

- gdy klucz jest tożsamy z elementem, to definiujemy:
  ```
  fun key(element): return element
  ```
- gdy klucz jest polem elementu, to definiujemy:
  ```
  fun key(element): return element.key
  ```
- gdy element jest parą: klucz, wartość, definiujemy:
  ```
  fun key(element): return element[0]
  ```

Pseudokod – założenia – funkcja haszująca

Będziemy używali funkcji haszującej postaci

$$h(k) = \text{hash}(k)\%n,$$

gdzie:

- Funkcja hash:
  - przyporządkowuje do klucza liczbę całkowitą z dużego zakresu, np. $[0, 2^{64})$,
  - jest zdefiniowana zależnie od typu klucza i równocześnie nie zależy od budowy samego słownika.

  Języki programowania zazwyczaj dostarczają takiej funkcji, np. w C++ mamy `std::hash`, zaś w Pythonie `hash`.
- n to liczba naturalna zależna od konstrukcji słownika, liczba różnych indeksów w słowniku.
- % to operacja dzielenia modulo (reszta z dzielenia), taka że $x\%n \in \{0, 1, \ldots, n-1\}$, gdzie x – dowolna wartości hash.

Pseudokod – założenia – funkcja haszująca

Będziemy używali funkcji haszującej postaci

$$h(k) = \text{hash}(k)\%n,$$

gdzie:

- Funkcja hash:
  - przyporządkowuje do klucza liczbę całkowitą z dużego zakresu, np. $[0, 2^{64})$,
  - jest zdefiniowana zależnie od typu klucza i równocześnie nie zależy od budowy samego słownika.

  Języki programowania zazwyczaj dostarczają takiej funkcji, np. w C++ mamy `std::hash`, zaś w Pythonie `hash`.
- n to liczba naturalna zależna od konstrukcji słownika, liczba różnych indeksów w słowniku.
- % to operacja dzielenia modulo (reszta z dzielenia), taka że $x\%n \in \{0, 1, \ldots, n-1\}$, gdzie x – dowolna wartości hash.

Pseudokod – założenia – funkcja haszująca

Będziemy używali funkcji haszującej postaci

$$h(k) = \text{hash}(k)\%n,$$

gdzie:

- Funkcja hash:
  - przyporządkowuje do klucza liczbę całkowitą z dużego zakresu, np. $[0, 2^{64})$,
  - jest zdefiniowana zależnie od typu klucza i równocześnie nie zależy od budowy samego słownika.

  Języki programowania zazwyczaj dostarczają takiej funkcji, np. w C++ mamy `std::hash`, zaś w Pythonie `hash`.
- n to liczba naturalna zależna od konstrukcji słownika, liczba różnych indeksów w słowniku.
- % to operacja dzielenia modulo (reszta z dzielenia), taka że $x\%n \in \{0, 1, \ldots, n-1\}$, gdzie x – dowolna wartości hash.

Pseudokod – założenia – funkcja haszująca

Będziemy używali funkcji haszującej postaci

$$h(k) = \text{hash}(k)\%n,$$

gdzie:

- Funkcja hash:
  - przyporządkowuje do klucza liczbę całkowitą z dużego zakresu, np. $[0, 2^{64})$,
  - jest zdefiniowana zależnie od typu klucza i równocześnie nie zależy od budowy samego słownika.

  Języki programowania zazwyczaj dostarczają takiej funkcji, np. w C++ mamy `std::hash`, zaś w Pythonie `hash`.
- n to liczba naturalna zależna od konstrukcji słownika, liczba różnych indeksów w słowniku.
- % to operacja dzielenia modulo (reszta z dzielenia), taka że $x\%n \in \{0, 1, \ldots, n-1\}$, gdzie x – dowolna wartości hash.

Metoda łańcuchowa – organizacja danych

Przy używaniu łańcuchowej metody rozwiązywania kolizji, słownik składa się z:

- `data` – tablica łańcuchów (dynamicznie rozszerzalnych tablic).
- `size` (tylko w przypadku rehaszowania) – liczba elementów w słowniku, sumaryczna długość tablic zawartych w `data`.

Notacja:

- `|data|` – długość tablicy `data` (liczba łańcuchów),
- `|data[i]|` – długość *i*-tego łańcucha,
- wszystkie tablice indeksujemy od 0.

Metoda łańcuchowa – organizacja danych

Przy używaniu łańcuchowej metody rozwiązywania kolizji, słownik składa się z:

- `data` – tablica łańcuchów (dynamicznie rozszerzalnych tablic).
- `size` (tylko w przypadku rehaszowania) – liczba elementów w słowniku, sumaryczna długość tablic zawartych w `data`.

Notacja:

- `|data|` – długość tablicy `data` (liczba łańcuchów),
- `|data[i]|` – długość *i*-tego łańcucha,
- wszystkie tablice indeksujemy od 0.

Metoda łańcuchowa – organizacja danych

Przy używaniu łańcuchowej metody rozwiązywania kolizji, słownik składa się z:

- `data` – tablica łańcuchów (dynamicznie rozszerzalnych tablic).
- `size` (tylko w przypadku rehaszowania) – liczba elementów w słowniku, sumaryczna długość tablic zawartych w `data`.

Notacja:

- `|data|` – długość tablicy `data` (liczba łańcuchów),
- `|data[i]|` – długość *i*-tego łańcucha,
- wszystkie tablice indeksujemy od 0.

Operacja find w metodzie łańcuchowej

Metoda pomocnicza zwracająca parę indeksów wskazujących kolejno:

1. tablicę w której może znajdować się element o kluczu k,
2. miejsce w tej tablicy, pod którym ten element się znajduje (lub -1 gdy go tam nie ma; więc nie ma go w słowniku)

```
fun find_index(k):
  h ← hash(k) % |data|
  for i ← 0, 1, ..., |data[h]|-1:
    if key(data[h][i]) = k:
      return h, i
  return h, -1
```

Zwraca element o kluczu `k` lub `None` gdy nie ma takiego elementu:

```
fun find(k):
  h, i ← find_index(k)
  if i = -1: return None
  return data[h][i]
```

Operacja find w metodzie łańcuchowej

Metoda pomocnicza zwracająca parę indeksów wskazujących kolejno:

1. tablicę w której może znajdować się element o kluczu k,
2. miejsce w tej tablicy, pod którym ten element się znajduje (lub -1 gdy go tam nie ma; więc nie ma go w słowniku)

```
fun find_index(k):
  h ← hash(k) % |data|
  for i ← 0, 1, ..., |data[h]|-1:
    if key(data[h][i]) = k:
      return h, i
  return h, -1
```

Zwraca element o kluczu `k` lub `None` gdy nie ma takiego elementu:

```
fun find(k):
  h, i ← find_index(k)
  if i = -1: return None
  return data[h][i]
```

Operacje insert i delete w metodzie łańcuchowej

Wstawianie elementu `element`:

```
fun insert(element):
  h, i ← find_index(key(element))
  if i = -1:
    dodaj element na koniec data[h]
  else: // mamy element o takim samym kluczu
    data[h][i] ← element // nadpisujemy go
```

Usuwanie elementu o kluczu `k`:

```
fun delete(k):
  h, i ← find_index(k)
  if i ≠ -1:
    data[h][i] ← ostatni element data[h]
    usuń ostatni element z data[h]
```

Operacje insert i delete w metodzie łańcuchowej

Wstawianie elementu `element`:

```
fun insert(element):
  h, i ← find_index(key(element))
  if i = -1:
    dodaj element na koniec data[h]
  else: // mamy element o takim samym kluczu
    data[h][i] ← element // nadpisujemy go
```

Usuwanie elementu o kluczu `k`:

```
fun delete(k):
  h, i ← find_index(k)
  if i ≠ -1:
    data[h][i] ← ostatni element data[h]
    usuń ostatni element z data[h]
```

Operacje insert i delete z rehaszowaniem

Metoda pomocnicza tworząca i zwracająca nową tablicę `new_size` tablic, do której przepisuje zawartość całego słownika:

```
fun new_data(new_size):
  r ← tablica new_size pustych tablic
  for tab in data:
    for e in tab:
      dodaj e na koniec r[hash(key(e))%|r|]
  return r
```

Operacje insert i delete z rehaszowaniem

Wstawianie elementu `element`:

```
fun insert(element):
  h, i ← find_index(key(element))
  if i = -1:
    dodaj element na koniec data[h]
    size ← size + 1
    if size > |data| * MAX_PER_LIST:
      data ← new_data(2*|data|)
  else: // mamy element o takim samym kluczu
    data[h][i] ← element // nadpisujemy go
```

Operacje insert i delete z rehaszowaniem

Usuwanie elementu o kluczu `k`:

```
fun delete(k):
  h, i ← find_index(k)
  if i ≠ -1:
    data[h][i] ← ostatni element data[h]
    usuń ostatni element z data[h]
    size ← size - 1
    if |data| > 1 and 4 * size < |data| * MAX_PER
      data ← new_data(|data|/2)
```

Adresowanie otwarte

- dane trzymane w zwykłej tablicy elementów (nazwijmy ją `data`, i oznaczmy jej pojemność przez n, tj. $n = |\,\text{data}\,|$);
- za pomocą funkcji haszującej wyznaczany jest tzw. ciąg kontrolny $h_0(k), h_1(k), \ldots, h_{n-1}(k)$ będący permutacją wszystkich indeksów tablicy `data`, np.
- $h_i(k) = (h(k) + i)\%n$ – adresowanie liniowe, w którym:
  - h wyznacza pierwszy ($i = 0$) wyraz ciągu kontrolnego,
  - następne wyrazy to kolejne (modulo n) pozycje `data`, tj. $h_i(k) = (h_{i-1}(k) + 1)\%n$ dla $i = 1, \ldots, n-1$.
- $h_i(k) = (h(k) + i \cdot \delta(k))\%n$ – haszowanie podwójne, w którym odległość pomiędzy kolejnymi wyrazami ciągu obliczana jest przez niezależną od h funkcję δ, tj. $h_i(k) = (h_{i-1}(k) + \delta(k))\%n$ dla $i = 1, \ldots, n-1$.
  Uwaga: by ciąg kontrolny przebiegł wszystkie indeksy `data`, $\delta(k)$ musi być względnie pierwsze z n, co można uzyskać zapewniając by n było potęgą dwójki, zaś $\delta(k)$ nieparzyste.

Adresowanie otwarte

- dane trzymane w zwykłej tablicy elementów (nazwijmy ją `data`, i oznaczmy jej pojemność przez n, tj. $n = |\,\text{data}\,|$);
- za pomocą funkcji haszującej wyznaczany jest tzw. ciąg kontrolny $h_0(k), h_1(k), \ldots, h_{n-1}(k)$ będący permutacją wszystkich indeksów tablicy `data`, np.
- $h_i(k) = (h(k) + i)\%n$ – adresowanie liniowe, w którym:
  - h wyznacza pierwszy ($i = 0$) wyraz ciągu kontrolnego,
  - następne wyrazy to kolejne (modulo n) pozycje `data`, tj. $h_i(k) = (h_{i-1}(k) + 1)\%n$ dla $i = 1, \ldots, n-1$.
- $h_i(k) = (h(k) + i \cdot \delta(k))\%n$ – haszowanie podwójne, w którym odległość pomiędzy kolejnymi wyrazami ciągu obliczana jest przez niezależną od h funkcję δ, tj. $h_i(k) = (h_{i-1}(k) + \delta(k))\%n$ dla $i = 1, \ldots, n-1$.
  Uwaga: by ciąg kontrolny przebiegł wszystkie indeksy `data`, $\delta(k)$ musi być względnie pierwsze z n, co można uzyskać zapewniając by n było potęgą dwójki, zaś $\delta(k)$ nieparzyste.

Adresowanie otwarte

- dane trzymane w zwykłej tablicy elementów (nazwijmy ją `data`, i oznaczmy jej pojemność przez n, tj. $n = |\,\text{data}\,|$);
- za pomocą funkcji haszującej wyznaczany jest tzw. ciąg kontrolny $h_0(k), h_1(k), \ldots, h_{n-1}(k)$ będący permutacją wszystkich indeksów tablicy `data`, np.
- $h_i(k) = (h(k) + i)\%n$ – adresowanie liniowe, w którym:
  - h wyznacza pierwszy ($i = 0$) wyraz ciągu kontrolnego,
  - następne wyrazy to kolejne (modulo n) pozycje `data`, tj. $h_i(k) = (h_{i-1}(k) + 1)\%n$ dla $i = 1, \ldots, n-1$.
- $h_i(k) = (h(k) + i \cdot \delta(k))\%n$ – haszowanie podwójne, w którym odległość pomiędzy kolejnymi wyrazami ciągu obliczana jest przez niezależną od h funkcję δ, tj. $h_i(k) = (h_{i-1}(k) + \delta(k))\%n$ dla $i = 1, \ldots, n-1$.
  Uwaga: by ciąg kontrolny przebiegł wszystkie indeksy `data`, $\delta(k)$ musi być względnie pierwsze z n, co można uzyskać zapewniając by n było potęgą dwójki, zaś $\delta(k)$ nieparzyste.

Adresowanie otwarte

- dane trzymane w zwykłej tablicy elementów (nazwijmy ją `data`, i oznaczmy jej pojemność przez n, tj. $n = |\,\text{data}\,|$);
- za pomocą funkcji haszującej wyznaczany jest tzw. ciąg kontrolny $h_0(k), h_1(k), \ldots, h_{n-1}(k)$ będący permutacją wszystkich indeksów tablicy `data`, np.
- $h_i(k) = (h(k) + i)\%n$ – adresowanie liniowe, w którym:
  - h wyznacza pierwszy ($i = 0$) wyraz ciągu kontrolnego,
  - następne wyrazy to kolejne (modulo n) pozycje `data`, tj. $h_i(k) = (h_{i-1}(k) + 1)\%n$ dla $i = 1, \ldots, n-1$.
- $h_i(k) = (h(k) + i \cdot \delta(k))\%n$ – haszowanie podwójne, w którym odległość pomiędzy kolejnymi wyrazami ciągu obliczana jest przez niezależną od h funkcję δ, tj. $h_i(k) = (h_{i-1}(k) + \delta(k))\%n$ dla $i = 1, \ldots, n-1$.
  Uwaga: by ciąg kontrolny przebiegł wszystkie indeksy `data`, $\delta(k)$ musi być względnie pierwsze z n, co można uzyskać zapewniając by n było potęgą dwójki, zaś $\delta(k)$ nieparzyste.

Adresowanie otwarte

- dane trzymane w zwykłej tablicy elementów (nazwijmy ją `data`, i oznaczmy jej pojemność przez n, tj. $n = |\,\mathrm{data}\,|$);
- za pomocą funkcji haszującej wyznaczany jest tzw. ciąg kontrolny $h_0(k), h_1(k), \ldots, h_{n-1}(k)$ będący permutacją wszystkich indeksów tablicy `data`, np.
- $h_i(k) = (h(k) + i)\%n$ – adresowanie liniowe, w którym:
  - h wyznacza pierwszy ($i = 0$) wyraz ciągu kontrolnego,
  - następne wyrazy to kolejne (modulo n) pozycje `data`, tj. $h_i(k) = (h_{i-1}(k) + 1)\%n$ dla $i = 1, \ldots, n-1$.
- $h_i(k) = (h(k) + i \cdot \delta(k))\%n$ – haszowanie podwójne, w którym odległość pomiędzy kolejnymi wyrazami ciągu obliczana jest przez niezależną od h funkcję δ, tj. $h_i(k) = (h_{i-1}(k) + \delta(k))\%n$ dla $i = 1, \ldots, n-1$.
  Uwaga: by ciąg kontrolny przebiegł wszystkie indeksy `data`, $\delta(k)$ musi być względnie pierwsze z n, co można uzyskać zapewniając by n było potęgą dwójki, zaś $\delta(k)$ nieparzyste.

Adresowanie otwarte

- dane trzymane w zwykłej tablicy elementów (nazwijmy ją `data`, i oznaczmy jej pojemność przez n, tj. $n = |\,\text{data}\,|$);
- za pomocą funkcji haszującej wyznaczany jest tzw. ciąg kontrolny $h_0(k), h_1(k), \ldots, h_{n-1}(k)$ będący permutacją wszystkich indeksów tablicy `data`, np.
- $h_i(k) = (h(k) + i)\%n$ – adresowanie liniowe, w którym:
  - h wyznacza pierwszy ($i = 0$) wyraz ciągu kontrolnego,
  - następne wyrazy to kolejne (modulo n) pozycje `data`, tj. $h_i(k) = (h_{i-1}(k) + 1)\%n$ dla $i = 1, \ldots, n-1$.
- $h_i(k) = (h(k) + i \cdot \delta(k))\%n$ – haszowanie podwójne, w którym odległość pomiędzy kolejnymi wyrazami ciągu obliczana jest przez niezależną od h funkcję δ, tj. $h_i(k) = (h_{i-1}(k) + \delta(k))\%n$ dla $i = 1, \ldots, n-1$.
  Uwaga: by ciąg kontrolny przebiegł wszystkie indeksy `data`, $\delta(k)$ musi być względnie pierwsze z n, co można uzyskać zapewniając by n było potęgą dwójki, zaś $\delta(k)$ nieparzyste.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:										

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:										

- Początkowo wszystkie komórki w tablicy są puste (słownik jest pusty).
- By oznaczyć komórkę jako pustą, można użyć jakiejś specjalnej wartości, np. -1.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:										

- Wstawmy do słownika (tablicy) liczbę 18.
- W tym celu przeglądamy kolejne indeksy ciągu kontrolnego

$$h_0(18) = 8, h_1(18) = 9, h_2(18) = 0, \ldots, h_9(18) = 7,$$

  aż do napotkania miejsca gdzie można wstawić.
- Ponieważ już komórka o indeksie $h_0(18) = 8$ jest pusta, to tam umieszczamy 18.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:										

- Wstawmy do słownika (tablicy) liczbę 18.
- W tym celu przeglądamy kolejne indeksy ciągu kontrolnego

$$h_0(18) = 8, h_1(18) = 9, h_2(18) = 0, \ldots, h_9(18) = 7,$$

  aż do napotkania miejsca gdzie można wstawić.
- Ponieważ już komórka o indeksie $h_0(18) = 8$ jest pusta, to tam umieszczamy 18.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:									18	

- Wstawmy do słownika (tablicy) liczbę 18.
- W tym celu przeglądamy kolejne indeksy ciągu kontrolnego

$$h_0(18) = 8, h_1(18) = 9, h_2(18) = 0, \ldots, h_9(18) = 7,$$

  aż do napotkania miejsca gdzie można wstawić.
- Ponieważ już komórka o indeksie $h_0(18) = 8$ jest pusta, to tam umieszczamy 18.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:									18	

- Wstawmy kolejno liczby 43, 27 i 10.
- Ponownie, już komórka wskazana przez pierwszy wyraz ciągu kontrolnego $h_0(43) = 3$ jest pusta. W nią wstawiamy 43.
- Analogicznie, 27 wstawiamy pod indeksem $h_0(27) = 7$.
- Zaś 10 do pustej komórki o indeksie $h_0(10) = 0$.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:				43					18	

- Wstawmy kolejno liczby 43, 27 i 10.
- Ponownie, już komórka wskazana przez pierwszy wyraz ciągu kontrolnego $h_0(43) = 3$ jest pusta. W nią wstawiamy 43.
- Analogicznie, 27 wstawiamy pod indeksem $h_0(27) = 7$.
- Zaś 10 do pustej komórki o indeksie $h_0(10) = 0$.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:				43				27	18	

- Wstawmy kolejno liczby 43, 27 i 10.
- Ponownie, już komórka wskazana przez pierwszy wyraz ciągu kontrolnego $h_0(43) = 3$ jest pusta. W nią wstawiamy 43.
- Analogicznie, 27 wstawiamy pod indeksem $h_0(27) = 7$.
- Zaś 10 do pustej komórki o indeksie $h_0(10) = 0$.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10			43				27	18	

- Wstawmy kolejno liczby 43, 27 i 10.
- Ponownie, już komórka wskazana przez pierwszy wyraz ciągu kontrolnego $h_0(43) = 3$ jest pusta. W nią wstawiamy 43.
- Analogicznie, 27 wstawiamy pod indeksem $h_0(27) = 7$.
- Zaś 10 do pustej komórki o indeksie $h_0(10) = 0$.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10			43				27	18	

- Wstawmy liczbę 58.
- Przeglądamy kolejne indeksy ciągu kontrolnego

$$h_0(58) = 8, h_1(58) = 9, h_2(58) = 0, \ldots, h_9(58) = 7,$$

  aż do napotkania miejsca gdzie można wstawić 58:
- Pozycja $h_0(58) = 8$ jest już zajęta przez 18, więc ją pomijamy.
- Pozycja $h_1(58) = 9$ jest wolna. W nią wstawiamy 58.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10			43				27	18	

- Wstawmy liczbę 58.
- Przeglądamy kolejne indeksy ciągu kontrolnego

$$h_0(58) = 8, h_1(58) = 9, h_2(58) = 0, \ldots, h_9(58) = 7,$$

  aż do napotkania miejsca gdzie można wstawić 58:
- Pozycja $h_0(58) = 8$ jest już zajęta przez 18, więc ją pomijamy.
- Pozycja $h_1(58) = 9$ jest wolna. W nią wstawiamy 58.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10			43				27	18	

- Wstawmy liczbę 58.
- Przeglądamy kolejne indeksy ciągu kontrolnego

$$h_0(58) = 8, h_1(58) = 9, h_2(58) = 0, \ldots, h_9(58) = 7,$$

  aż do napotkania miejsca gdzie można wstawić 58:
- Pozycja $h_0(58) = 8$ jest już zajęta przez 18, więc ją pomijamy.
- Pozycja $h_1(58) = 9$ jest wolna. W nią wstawiamy 58.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10			43				27	18	58

- Wstawmy liczbę 58.
- Przeglądamy kolejne indeksy ciągu kontrolnego

$$h_0(58) = 8, h_1(58) = 9, h_2(58) = 0, \ldots, h_9(58) = 7,$$

  aż do napotkania miejsca gdzie można wstawić 58:
- Pozycja $h_0(58) = 8$ jest już zajęta przez 18, więc ją pomijamy.
- Pozycja $h_1(58) = 9$ jest wolna. W nią wstawiamy 58.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10			43				27	18	58

- Wstawmy liczbę 67.
- Pozycja $h_0(67) = 7$ jest zajęta przez 27. Pomijamy ją.
- Pomijamy także zajętą komórkę $h_1(67) = 8$,
- oraz $h_2(67) = 9$,
- oraz $h_3(67) = 0$.
- Pierwszą wolną komórką jest $h_4(67) = 1$.
  W nią wstawiamy 67.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10			43				27	18	58

- Wstawmy liczbę 67.
- Pozycja $h_0(67) = 7$ jest zajęta przez 27. Pomijamy ją.
- Pomijamy także zajętą komórkę $h_1(67) = 8$,
- oraz $h_2(67) = 9$,
- oraz $h_3(67) = 0$.
- Pierwszą wolną komórką jest $h_4(67) = 1$.
  W nią wstawiamy 67.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10			43				27	18	58

- Wstawmy liczbę 67.
- Pozycja $h_0(67) = 7$ jest zajęta przez 27. Pomijamy ją.
- Pomijamy także zajętą komórkę $h_1(67) = 8$,
- oraz $h_2(67) = 9$,
- oraz $h_3(67) = 0$.
- Pierwszą wolną komórką jest $h_4(67) = 1$.
  W nią wstawiamy 67.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10			43				27	18	58

- Wstawmy liczbę 67.
- Pozycja $h_0(67) = 7$ jest zajęta przez 27. Pomijamy ją.
- Pomijamy także zajętą komórkę $h_1(67) = 8$,
- oraz $h_2(67) = 9$,
- oraz $h_3(67) = 0$.
- Pierwszą wolną komórką jest $h_4(67) = 1$.
  W nią wstawiamy 67.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10			43				27	18	58

- Wstawmy liczbę 67.
- Pozycja $h_0(67) = 7$ jest zajęta przez 27. Pomijamy ją.
- Pomijamy także zajętą komórkę $h_1(67) = 8$,
- oraz $h_2(67) = 9$,
- oraz $h_3(67) = 0$.
- Pierwszą wolną komórką jest $h_4(67) = 1$. W nią wstawiamy 67.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10	67		43				27	18	58

- Wstawmy liczbę 67.
- Pozycja $h_0(67) = 7$ jest zajęta przez 27. Pomijamy ją.
- Pomijamy także zajętą komórkę $h_1(67) = 8$,
- oraz $h_2(67) = 9$,
- oraz $h_3(67) = 0$.
- Pierwszą wolną komórką jest $h_4(67) = 1$.
  W nią wstawiamy 67.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10	67		43				27	18	58

- Poszukajmy liczby 23.
- W tym celu przeglądamy kolejne indeksy ciągu kontrolnego

$$h_0(23) = 3, h_1(23) = 4, \ldots, h_9(23) = 2,$$

  aż do napotkania liczby 23 albo pustego miejsca.
- Komórka $h_0(23) = 3$ nie zawiera liczby 23. Nie jest też ona pusta, więc kontynuujemy przeglądanie tablicy.
- Komórka $h_1(23) = 4$ także nie zawiera liczby 23. A ponieważ jest ona pusta, to wnioskujemy, że 23 nie ma w słowniku.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10	67		43				27	18	58

- Poszukajmy liczby 23.
- W tym celu przeglądamy kolejne indeksy ciągu kontrolnego

$$h_0(23) = 3, h_1(23) = 4, \ldots, h_9(23) = 2,$$

  aż do napotkania liczby 23 albo pustego miejsca.
- Komórka $h_0(23) = 3$ nie zawiera liczby 23. Nie jest też ona pusta, więc kontynuujemy przeglądanie tablicy.
- Komórka $h_1(23) = 4$ także nie zawiera liczby 23. A ponieważ jest ona pusta, to wnioskujemy, że 23 nie ma w słowniku.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10	67		43				27	18	58

- Poszukajmy liczby 23.
- W tym celu przeglądamy kolejne indeksy ciągu kontrolnego

$$h_0(23) = 3, h_1(23) = 4, \ldots, h_9(23) = 2,$$

  aż do napotkania liczby 23 albo pustego miejsca.
- Komórka $h_0(23) = 3$ nie zawiera liczby 23. Nie jest też ona pusta, więc kontynuujemy przeglądanie tablicy.
- Komórka $h_1(23) = 4$ także nie zawiera liczby 23. A ponieważ jest ona pusta, to wnioskujemy, że 23 nie ma w słowniku.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10	67		43				27	18	58

- Poszukajmy liczby 23.
- W tym celu przeglądamy kolejne indeksy ciągu kontrolnego

$$h_0(23) = 3, h_1(23) = 4, \ldots, h_9(23) = 2,$$

  aż do napotkania liczby 23 albo pustego miejsca.
- Komórka $h_0(23) = 3$ nie zawiera liczby 23. Nie jest też ona pusta, więc kontynuujemy przeglądanie tablicy.
- Komórka $h_1(23) = 4$ także nie zawiera liczby 23. A ponieważ jest ona pusta, to wnioskujemy, że 23 nie ma w słowniku.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10	67		43				27	18	58

- Poszukajmy liczby 58.
- Komórka $h_0(58) = 8$ jej nie zawiera. Ta komórka nie jest też pusta, więc kontynuujemy.
- 58 odnajdujemy pod indeksem $h_1(58) = 9$.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10	67		43				27	18	58

- Poszukajmy liczby 58.
- Komórka $h_0(58) = 8$ jej nie zawiera. Ta komórka nie jest też pusta, więc kontynuujemy.
- 58 odnajdujemy pod indeksem $h_1(58) = 9$.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10	67		43				27	18	58

- Poszukajmy liczby 58.
- Komórka $h_0(58) = 8$ jej nie zawiera. Ta komórka nie jest też pusta, więc kontynuujemy.
- 58 odnajdujemy pod indeksem $h_1(58) = 9$.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10	67		43				27	18	58

- Poszukajmy liczby 19.
- Niepusta komórka $h_0(19) = 9$ jej nie zawiera, więc kontynuujemy przeglądanie tablicy.
- Podobna sytuacja jest pod indeksem $h_1(19) = 0$
- oraz $h_2(19) = 1$.
- Także komórka $h_3(19) = 2$ nie zawiera 19. Ale ponieważ jest ona pusta, to wyszukiwanie kończy się wnioskiem, że 19 nie ma w słowniku.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10	67		43				27	18	58

- Poszukajmy liczby 19.
- Niepusta komórka $h_0(19) = 9$ jej nie zawiera, więc kontynuujemy przeglądanie tablicy.
- Podobna sytuacja jest pod indeksem $h_1(19) = 0$
- oraz $h_2(19) = 1$.
- Także komórka $h_3(19) = 2$ nie zawiera 19. Ale ponieważ jest ona pusta, to wyszukiwanie kończy się wnioskiem, że 19 nie ma w słowniku.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10	67		43				27	18	58

- Poszukajmy liczby 19.
- Niepusta komórka $h_0(19) = 9$ jej nie zawiera, więc kontynuujemy przeglądanie tablicy.
- Podobna sytuacja jest pod indeksem $h_1(19) = 0$
- oraz $h_2(19) = 1$.
- Także komórka $h_3(19) = 2$ nie zawiera 19. Ale ponieważ jest ona pusta, to wyszukiwanie kończy się wnioskiem, że 19 nie ma w słowniku.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10	67		43				27	18	58

- Poszukajmy liczby 19.
- Niepusta komórka $h_0(19) = 9$ jej nie zawiera, więc kontynuujemy przeglądanie tablicy.
- Podobna sytuacja jest pod indeksem $h_1(19) = 0$
- oraz $h_2(19) = 1$.
- Także komórka $h_3(19) = 2$ nie zawiera 19. Ale ponieważ jest ona pusta, to wyszukiwanie kończy się wnioskiem, że 19 nie ma w słowniku.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10	67		43				27	18	58

- Poszukajmy liczby 19.
- Niepusta komórka $h_0(19) = 9$ jej nie zawiera, więc kontynuujemy przeglądanie tablicy.
- Podobna sytuacja jest pod indeksem $h_1(19) = 0$
- oraz $h_2(19) = 1$.
- Także komórka $h_3(19) = 2$ nie zawiera 19. Ale ponieważ jest ona pusta, to wyszukiwanie kończy się wnioskiem, że 19 nie ma w słowniku.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10	67		43				27	18	58

- Usuńmy ze słownika liczbę 18.
- Najpierw odnajdujemy jej indeks w tablicy przeglądając kolejne indeksy jej ciągu kontrolnego $h_0(18), \ldots, h_9(18)$.
- Już pod pierwszym z nich ($h_0(18) = 8$) odnajdujemy 18.
- Nie możemy jednak po prostu oznaczyć komórki 8 jako pustej, ponieważ to uniemożliwiło by odnalezienie liczb 58 oraz 67.
- Oznaczymy więc tę komórkę jako skasowaną.
  W tym celu można użyć jakiejś specjalnej wartości, różnej od tej oznaczającej komórkę pustą, np. -2.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10	67		43				27	18	58

- Usuńmy ze słownika liczbę 18.
- Najpierw odnajdujemy jej indeks w tablicy przeglądając kolejne indeksy jej ciągu kontrolnego $h_0(18), \ldots, h_9(18)$.
- Już pod pierwszym z nich ($h_0(18) = 8$) odnajdujemy 18.
- Nie możemy jednak po prostu oznaczyć komórki 8 jako pustej, ponieważ to uniemożliwiło by odnalezienie liczb 58 oraz 67.
- Oznaczymy więc tę komórkę jako skasowaną.
  W tym celu można użyć jakiejś specjalnej wartości, różnej od tej oznaczającej komórkę pustą, np. -2.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10	67		43				27	18	58

- Usuńmy ze słownika liczbę 18.
- Najpierw odnajdujemy jej indeks w tablicy przeglądając kolejne indeksy jej ciągu kontrolnego $h_0(18), \ldots, h_9(18)$.
- Już pod pierwszym z nich ($h_0(18) = 8$) odnajdujemy 18.
- Nie możemy jednak po prostu oznaczyć komórki 8 jako pustej, ponieważ to uniemożliwiło by odnalezienie liczb 58 oraz 67.
- Oznaczymy więc tę komórkę jako skasowaną.
  W tym celu można użyć jakiejś specjalnej wartości, różnej od tej oznaczającej komórkę pustą, np. -2.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10	67		43				27		58

- Usuńmy ze słownika liczbę 18.
- Najpierw odnajdujemy jej indeks w tablicy przeglądając kolejne indeksy jej ciągu kontrolnego $h_0(18), \ldots, h_9(18)$.
- Już pod pierwszym z nich ($h_0(18) = 8$) odnajdujemy 18.
- Nie możemy jednak po prostu oznaczyć komórki 8 jako pustej, ponieważ to uniemożliwiło by odnalezienie liczb 58 oraz 67.
- Oznaczymy więc tę komórkę jako skasowaną.
  W tym celu można użyć jakiejś specjalnej wartości, różnej od tej oznaczającej komórkę pustą, np. -2.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10	67		43				27	X	58

- Usuńmy ze słownika liczbę 18.
- Najpierw odnajdujemy jej indeks w tablicy przeglądając kolejne indeksy jej ciągu kontrolnego $h_0(18), \ldots, h_9(18)$.
- Już pod pierwszym z nich ($h_0(18) = 8$) odnajdujemy 18.
- Nie możemy jednak po prostu oznaczyć komórki 8 jako pustej, ponieważ to uniemożliwiło by odnalezienie liczb 58 oraz 67.
- Oznaczymy więc tę komórkę jako skasowaną.
  W tym celu można użyć jakiejś specjalnej wartości, różnej od tej oznaczającej komórkę pustą, np. -2.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10	67		43				27	X	58

- Poszukajmy liczby 58.
- Komórka $h_0(58) = 8$ nie zawiera liczby 58. Nie jest też ona pusta (jest skasowana, a to co innego), więc kontynuujemy
- i odnajdujemy 58 pod indeksem $h_1(58) = 9$.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10	67		43				27	X	58

- Poszukajmy liczby 58.
- Komórka $h_0(58) = 8$ nie zawiera liczby 58. Nie jest też ona pusta (jest skasowana, a to co innego), więc kontynuujemy
- i odnajdujemy 58 pod indeksem $h_1(58) = 9$.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10	67		43				27	X	58

- Poszukajmy liczby 58.
- Komórka $h_0(58) = 8$ nie zawiera liczby 58. Nie jest też ona pusta (jest skasowana, a to co innego), więc kontynuujemy
- i odnajdujemy 58 pod indeksem $h_1(58) = 9$.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10	67		43				27	X	58

- Wstawmy liczbę 17.
- Pozycja $h_0(17) = 7$ jest zajęta przez 27. Pomijamy ją.
- Pozycja $h_1(17) = 8$ nie jest zajęta przez liczbę (jest skasowana), więc możemy w nią wstawić 17.
- **Uwaga:** jeśli nie dopuszczamy duplikatów, to powinniśmy sprawdzić, czy 17 nie znajduje się w dalszej części tablicy. Przykładowo, gdybyśmy zamiast 17 wstawiali 67, to na skasowaną komórkę (o indeksie $8 = h_1(67)$) natrafilibyśmy wcześniej niż na komórkę ($1 = h_4(67)$) zawierającą 67.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10	67		43				27	X	58

- Wstawmy liczbę 17.
- Pozycja $h_0(17) = 7$ jest zajęta przez 27. Pomijamy ją.
- Pozycja $h_1(17) = 8$ nie jest zajęta przez liczbę (jest skasowana), więc możemy w nią wstawić 17.
- **Uwaga:** jeśli nie dopuszczamy duplikatów, to powinniśmy sprawdzić, czy 17 nie znajduje się w dalszej części tablicy. Przykładowo, gdybyśmy zamiast 17 wstawiali 67, to na skasowaną komórkę (o indeksie $8 = h_1(67)$) natrafilibyśmy wcześniej niż na komórkę ($1 = h_4(67)$) zawierającą 67.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10	67		43				27	X	58

- Wstawmy liczbę 17.
- Pozycja $h_0(17) = 7$ jest zajęta przez 27. Pomijamy ją.
- Pozycja $h_1(17) = 8$ nie jest zajęta przez liczbę (jest skasowana), więc możemy w nią wstawić 17.
- **Uwaga:** jeśli nie dopuszczamy duplikatów, to powinniśmy sprawdzić, czy 17 nie znajduje się w dalszej części tablicy. Przykładowo, gdybyśmy zamiast 17 wstawiali 67, to na skasowaną komórkę (o indeksie $8 = h_1(67)$) natrafilibyśmy wcześniej niż na komórkę ($1 = h_4(67)$) zawierającą 67.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10	67		43				27	17	58

- Wstawmy liczbę 17.
- Pozycja $h_0(17) = 7$ jest zajęta przez 27. Pomijamy ją.
- Pozycja $h_1(17) = 8$ nie jest zajęta przez liczbę (jest skasowana), więc możemy w nią wstawić 17.
- **Uwaga:** jeśli nie dopuszczamy duplikatów, to powinniśmy sprawdzić, czy 17 nie znajduje się w dalszej części tablicy. Przykładowo, gdybyśmy zamiast 17 wstawiali 67, to na skasowaną komórkę (o indeksie $8 = h_1(67)$) natrafilibyśmy wcześniej niż na komórkę ($1 = h_4(67)$) zawierającą 67.

Adresowanie otwarte – przykład

Za pomocą tablicy (`data` o pojemności 10) z haszowaniem i adresowaniem otwartym liniowym ($h_i(x) = (x + i)\%10$) zrealizujemy słownik liczb naturalnych.

indeks:	0	1	2	3	4	5	6	7	8	9
wartość:	10	67		43				27	17	58

- Wstawmy liczbę 17.
- Pozycja $h_0(17) = 7$ jest zajęta przez 27. Pomijamy ją.
- Pozycja $h_1(17) = 8$ nie jest zajęta przez liczbę (jest skasowana), więc możemy w nią wstawić 17.
- **Uwaga:** jeśli nie dopuszczamy duplikatów, to powinniśmy sprawdzić, czy 17 nie znajduje się w dalszej części tablicy. Przykładowo, gdybyśmy zamiast 17 wstawiali 67, to na skasowaną komórkę (o indeksie $8 = h_1(67)$) natrafilibyśmy wcześniej niż na komórkę ($1 = h_4(67)$) zawierającą 67.

Adresowanie otwarte – obliczeniowa złożoność czasowa

Niech:

- s to liczba niepustych komórek,
- $n = |\,\text{data}\,|$ to pojemność słownika,
- $\alpha = \frac{s}{n}$ to współczynnik zapełnienia tablicy; załóżmy że $\alpha < 1$.

Załóżmy ponadto, że ciąg kontrolny dla losowo wybranego klucza jest z równym prawdopodobieństwem równy dowolnej permutacji zbioru indeksów[1]. Wtedy oczekiwana liczba porównań kluczy w czasie wyszukiwania elementu jest nie większa niż[2]:

- $\frac{1}{1-\alpha}$ gdy elementu o zadanym kluczu nie ma w tablicy,
- $\frac{1}{\alpha} \ln \frac{1}{1-\alpha} + \frac{1}{\alpha}$ gdy jest w tablicy (i gdy każdy klucz w tablicy jest z równym prawdopodobieństwem tym wyszukiwanym).

[1] Haszowanie podwójne jest zazwyczaj bliższe spełnienia tego wyidealizowanego warunku niż adresowanie liniowe.

[2] Dowód można znaleźć w książce: Thomas H. Cormen, Charles E. Leiserson, Roland L. Rivest, Clifford Stein *Wprowadzenie do algorytmów* (rozdział *Tablice z haszowaniem* w części *Struktury danych*).

Adresowanie otwarte – obliczeniowa złożoność czasowa

Niech:

- s to liczba niepustych komórek,
- $n = |\,\text{data}\,|$ to pojemność słownika,
- $\alpha = \frac{s}{n}$ to współczynnik zapełnienia tablicy; załóżmy że $\alpha < 1$.

Załóżmy ponadto, że ciąg kontrolny dla losowo wybranego klucza jest z równym prawdopodobieństwem równy dowolnej permutacji zbioru indeksów[1]. Wtedy oczekiwana liczba porównań kluczy w czasie wyszukiwania elementu jest nie większa niż[2]:

- $\frac{1}{1-\alpha}$ gdy elementu o zadanym kluczu nie ma w tablicy,
- $\frac{1}{\alpha} \ln \frac{1}{1-\alpha} + \frac{1}{\alpha}$ gdy jest w tablicy (i gdy każdy klucz w tablicy jest z równym prawdopodobieństwem tym wyszukiwanym).

[1] Haszowanie podwójne jest zazwyczaj bliższe spełnienia tego wyidealizowanego warunku niż adresowanie liniowe.

[2] Dowód można znaleźć w książce: Thomas H. Cormen, Charles E. Leiserson, Roland L. Rivest, Clifford Stein *Wprowadzenie do algorytmów* (rozdział *Tablice z haszowaniem* w części *Struktury danych*).

Adresowanie otwarte – obliczeniowa złożoność czasowa

Niech:

- s to liczba niepustych komórek,
- $n = |\,\text{data}\,|$ to pojemność słownika,
- $\alpha = \frac{s}{n}$ to współczynnik zapełnienia tablicy; załóżmy że $\alpha < 1$.

Załóżmy ponadto, że ciąg kontrolny dla losowo wybranego klucza jest z równym prawdopodobieństwem równy dowolnej permutacji zbioru indeksów[1]. Wtedy oczekiwana liczba porównań kluczy w czasie wyszukiwania elementu jest nie większa niż[2]:

- $\frac{1}{1-\alpha}$ gdy elementu o zadanym kluczu nie ma w tablicy,
- $\frac{1}{\alpha} \ln \frac{1}{1-\alpha} + \frac{1}{\alpha}$ gdy jest w tablicy (i gdy każdy klucz w tablicy jest z równym prawdopodobieństwem tym wyszukiwanym).

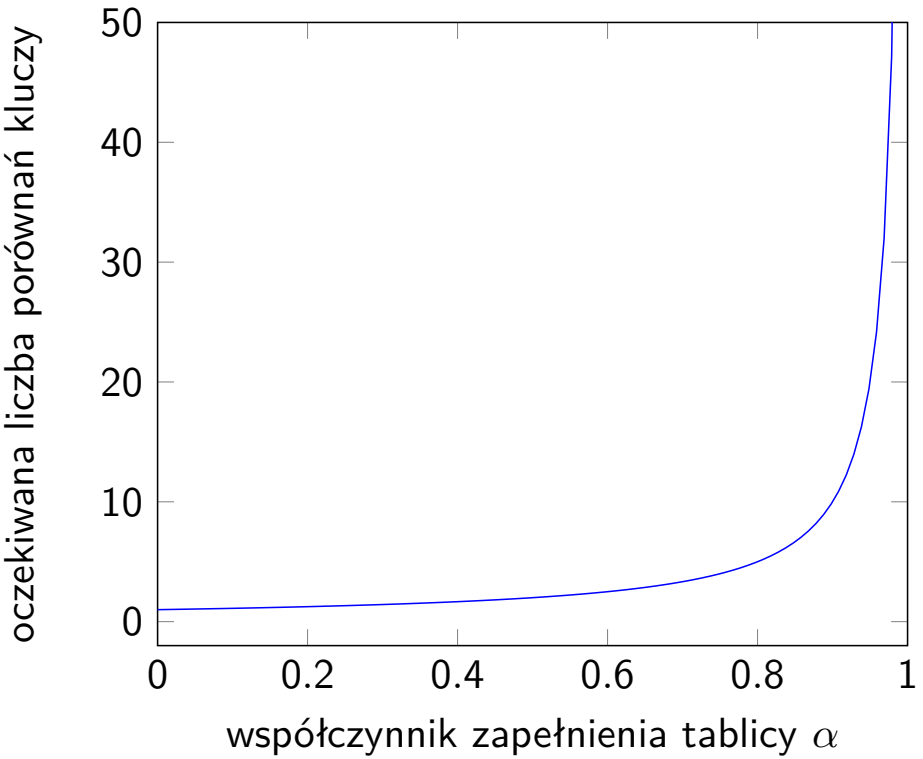
---

[1] Haszowanie podwójne jest zazwyczaj bliższe spełnienia tego wyidealizowanego warunku niż adresowanie liniowe.

[2] Dowód można znaleźć w książce: Thomas H. Cormen, Charles E. Leiserson, Roland L. Rivest, Clifford Stein *Wprowadzenie do algorytmów* (rozdział *Tablice z haszowaniem* w części *Struktury danych*).

Adresowanie otwarte – obliczeniowa złożoność czasowa

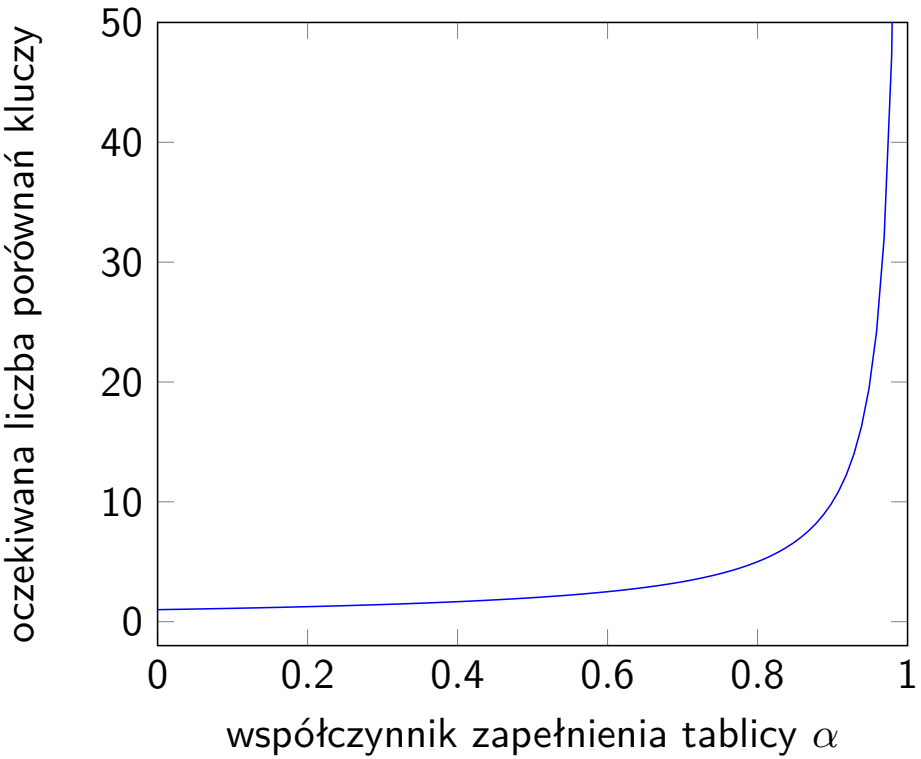
Czas wyszukiwaniu elementu którego nie ma w tablicy:



- By uzyskać oczekiwaną, zamortyzowaną złożoność $O(1)$, należy zapewnić, by współczynnik zapełnienia tablicy α nie przekroczył pewnej stałej $C < 1$, np. $C = 0{,}7$.
- Można podwajać pojemność tablicy gdy $\alpha > C$, oraz przedzielić ją przez 2 gdy $\alpha < C/4$, podobnie jak to czyniliśmy w przypadku adresowania łańcuchowego (i z podobnym kosztem rehaszowania).

Adresowanie otwarte – obliczeniowa złożoność czasowa

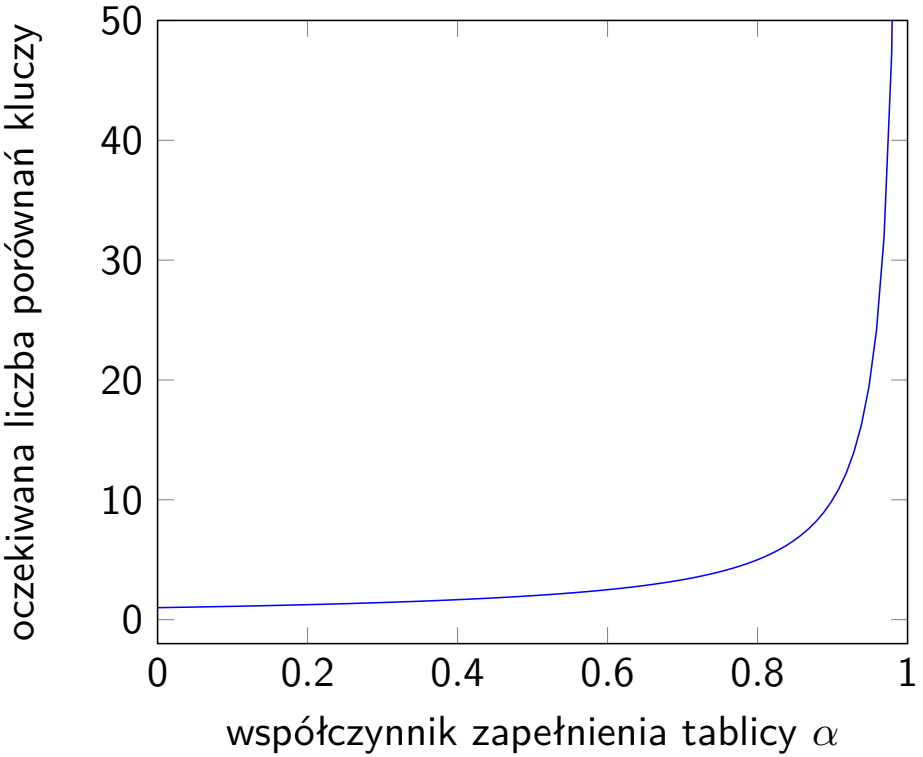
Czas wyszukiwaniu elementu którego nie ma w tablicy:



- By uzyskać oczekiwaną, zamortyzowaną złożoność $O(1)$, należy zapewnić, by współczynnik zapełnienia tablicy α nie przekroczył pewnej stałej $C < 1$, np. $C = 0{,}7$.
- Można podwajać pojemność tablicy gdy $\alpha > C$, oraz przedzielić ją przez 2 gdy $\alpha < C/4$, podobnie jak to czyniliśmy w przypadku adresowania łańcuchowego (i z podobnym kosztem rehaszowania).

Adresowanie otwarte – obliczeniowa złożoność czasowa

Czas wyszukiwaniu elementu którego nie ma w tablicy:



- By uzyskać oczekiwaną, zamortyzowaną złożoność $O(1)$, należy zapewnić, by współczynnik zapełnienia tablicy α nie przekroczył pewnej stałej $C < 1$, np. $C = 0{,}7$.
- Można podwajać pojemność tablicy gdy $\alpha > C$, oraz przedzielić ją przez 2 gdy $\alpha < C/4$, podobnie jak to czyniliśmy w przypadku adresowania łańcuchowego (i z podobnym kosztem rehaszowania).

Adresowanie otwarte – organizacja danych

Przy adresowanie otwartym, słownik składa się z:

- `data` – tablica elementów wzbogacona o możliwość zaznaczania pustych oraz skasowanych komórek (za pomocą wartości specjalnych albo osobnych pól);
- `size` (tylko w przypadku rehaszowania) – liczba elementów w `data`.

Notacja / założenia:

- tablicę `data` indeksujemy od 0, zaś `|data|` to jej długość,
- `empty(i)` – sprawdza czy `i`-ta komórka `data` jest pusta,
- `deleted(i)` – sprawdza czy `i`-ta komórka jest skasowania,
- implementujemy haszowanie podwójne, by uzyskać zaś liniowe wystarczy przyjąć krok $\delta(k) = 1$ dla wszystkich k,
- nie dopuszczamy duplikatów kluczy w słowniku.

Adresowanie otwarte – organizacja danych

Przy adresowanie otwartym, słownik składa się z:

- `data` – tablica elementów wzbogacona o możliwość zaznaczania pustych oraz skasowanych komórek (za pomocą wartości specjalnych albo osobnych pól);
- `size` (tylko w przypadku rehaszowania) – liczba elementów w `data`.

Notacja / założenia:

- tablicę `data` indeksujemy od 0, zaś `|data|` to jej długość,
- `empty(i)` – sprawdza czy `i`-ta komórka `data` jest pusta,
- `deleted(i)` – sprawdza czy `i`-ta komórka jest skasowania,
- implementujemy haszowanie podwójne, by uzyskać zaś liniowe wystarczy przyjąć krok $\delta(k) = 1$ dla wszystkich k,
- nie dopuszczamy duplikatów kluczy w słowniku.

Adresowanie otwarte – organizacja danych

Przy adresowanie otwartym, słownik składa się z:

- `data` – tablica elementów wzbogacona o możliwość zaznaczania pustych oraz skasowanych komórek (za pomocą wartości specjalnych albo osobnych pól);
- `size` (tylko w przypadku rehaszowania) – liczba elementów w `data`.

Notacja / założenia:

- tablicę `data` indeksujemy od 0, zaś `|data|` to jej długość,
- `empty(i)` – sprawdza czy `i`-ta komórka `data` jest pusta,
- `deleted(i)` – sprawdza czy `i`-ta komórka jest skasowania,
- implementujemy haszowanie podwójne, by uzyskać zaś liniowe wystarczy przyjąć krok $\delta(k) = 1$ dla wszystkich k,
- nie dopuszczamy duplikatów kluczy w słowniku.

Adresowanie otwarte – organizacja danych

Przy adresowanie otwartym, słownik składa się z:

- `data` – tablica elementów wzbogacona o możliwość zaznaczania pustych oraz skasowanych komórek (za pomocą wartości specjalnych albo osobnych pól);
- `size` (tylko w przypadku rehaszowania) – liczba elementów w `data`.

Notacja / założenia:

- tablicę `data` indeksujemy od 0, zaś `|data|` to jej długość,
- `empty(i)` – sprawdza czy `i`-ta komórka `data` jest pusta,
- `deleted(i)` – sprawdza czy `i`-ta komórka jest skasowania,
- implementujemy haszowanie podwójne, by uzyskać zaś liniowe wystarczy przyjąć krok $\delta(k) = 1$ dla wszystkich k,
- nie dopuszczamy duplikatów kluczy w słowniku.

Adresowanie otwarte – organizacja danych

Przy adresowanie otwartym, słownik składa się z:

- `data` – tablica elementów wzbogacona o możliwość zaznaczania pustych oraz skasowanych komórek (za pomocą wartości specjalnych albo osobnych pól);
- `size` (tylko w przypadku rehaszowania) – liczba elementów w `data`.

Notacja / założenia:

- tablicę `data` indeksujemy od 0, zaś `|data|` to jej długość,
- `empty(i)` – sprawdza czy `i`-ta komórka `data` jest pusta,
- `deleted(i)` – sprawdza czy `i`-ta komórka jest skasowania,
- implementujemy haszowanie podwójne, by uzyskać zaś liniowe wystarczy przyjąć krok $\delta(k) = 1$ dla wszystkich k,
- nie dopuszczamy duplikatów kluczy w słowniku.

Adresowanie otwarte – organizacja danych

Przy adresowanie otwartym, słownik składa się z:

- `data` – tablica elementów wzbogacona o możliwość zaznaczania pustych oraz skasowanych komórek (za pomocą wartości specjalnych albo osobnych pól);
- `size` (tylko w przypadku rehaszowania) – liczba elementów w `data`.

Notacja / założenia:

- tablicę `data` indeksujemy od 0, zaś `|data|` to jej długość,
- `empty(i)` – sprawdza czy `i`-ta komórka `data` jest pusta,
- `deleted(i)` – sprawdza czy `i`-ta komórka jest skasowania,
- implementujemy haszowanie podwójne, by uzyskać zaś liniowe wystarczy przyjąć krok $\delta(k) = 1$ dla wszystkich k,
- nie dopuszczamy duplikatów kluczy w słowniku.

Adresowanie otwarte – organizacja danych

Przy adresowanie otwartym, słownik składa się z:

- `data` – tablica elementów wzbogacona o możliwość zaznaczania pustych oraz skasowanych komórek (za pomocą wartości specjalnych albo osobnych pól);
- `size` (tylko w przypadku rehaszowania) – liczba elementów w `data`.

Notacja / założenia:

- tablicę `data` indeksujemy od 0, zaś `|data|` to jej długość,
- `empty(i)` – sprawdza czy `i`-ta komórka `data` jest pusta,
- `deleted(i)` – sprawdza czy `i`-ta komórka jest skasowania,
- implementujemy haszowanie podwójne, by uzyskać zaś liniowe wystarczy przyjąć krok $\delta(k) = 1$ dla wszystkich k,
- nie dopuszczamy duplikatów kluczy w słowniku.

Adresowanie otwarte – pseudokod metody pomocniczej

Metoda pomocnicza zwracająca indeks elementu o kluczu `k` albo, gdy nie ma takiego indeksu, indeks pierwszej skasowanej lub pustej komórki albo, gdy nie ma też takiej komórki, `-1`:

```
fun scan_for(k):
  f ← h(k) % |data| // pierwszy indeks
  s ← δ(k) // krok; 1 dla adresowania liniowego
  d ← -1 // indeks pierwszej skasowanej komórki
  i ← f
  while not empty(i):
    if deleted(i):
      if d = -1: d ← i
    else if key(data[i]) = k: return i
    i ← (i + s) % |data|
    if i = f: // przeszliśmy całą tablicę
      return d
  if d ≠ -1: return d
  return i
```

Adresowanie otwarte – pseudokod operacji słownikowych

Zwraca element o kluczu `k` lub `None` gdy nie ma takiego elementu:

```
fun find(k):
  i ← scan_for(k)
  if i = -1 or empty(i) or deleted(i):
    return None
  return data[i]
```

Wstawia element `e` lub nadpisuje element o kluczu `key(e)`:

```
fun insert(e):
  i ← scan_for(key(e))
  if i = -1: błąd, brak miejsca
  data[i] ← e // wstawienie lub nadpisanie
```

Usuwa element o kluczu `k`:

```
fun delete(k):
  i ← scan_for(k)
  if i ≠ -1 and not empty(i):
    oznacz i-tą komórkę data jako skasowaną
```

Adresowanie otwarte – pseudokod operacji słownikowych

Zwraca element o kluczu `k` lub `None` gdy nie ma takiego elementu:

```
fun find(k):
  i ← scan_for(k)
  if i = -1 or empty(i) or deleted(i):
    return None
  return data[i]
```

Wstawia element `e` lub nadpisuje element o kluczu `key(e)`:

```
fun insert(e):
  i ← scan_for(key(e))
  if i = -1: błąd, brak miejsca
  data[i] ← e // wstawienie lub nadpisanie
```

Usuwa element o kluczu `k`:

```
fun delete(k):
  i ← scan_for(k)
  if i ≠ -1 and not empty(i):
    oznacz i-tą komórkę data jako skasowaną
```

Adresowanie otwarte – pseudokod operacji słownikowych

Zwraca element o kluczu `k` lub `None` gdy nie ma takiego elementu:

```
fun find(k):
  i ← scan_for(k)
  if i = -1 or empty(i) or deleted(i):
    return None
  return data[i]
```

Wstawia element `e` lub nadpisuje element o kluczu `key(e)`:

```
fun insert(e):
  i ← scan_for(key(e))
  if i = -1: błąd, brak miejsca
  data[i] ← e // wstawienie lub nadpisanie
```

Usuwa element o kluczu `k`:

```
fun delete(k):
  i ← scan_for(k)
  if i ≠ -1 and not empty(i):
    oznacz i-tą komórkę data jako skasowaną
```

Adresowanie otwarte – rehaszowanie – metoda pomocnicza

Metoda pomocnicza tworząca i zwracająca nową tablicę o pojemności `new_size` elementów (musi być nie mniejsza niż liczba elementów w słowniku), do której przepisuje zawartość całego słownika :

```
fun new_data(new_size):
  r ← tablica new_size elementów,
       wszystkie komórki oznaczone jako puste
  for src_i ← 0, 1, ..., |data|-1:
    if not empty(src_i) and not deleted(src_i):
      k ← key(data[src_i])
      i ← h(k) % |r| // pierwszy indeks
      s ← δ(k) // 1 dla adresowania liniowego
      while komórka r[i] nie jest pusta:
        i ← (i + s) % |r|
      r[i] ← data[src_i]
  return r
```

Adresowanie otwarte – insert z rehaszowaniem

Wstawia element `e` lub nadpisuje element o kluczu `key(e)`:

```
fun insert(e):
  i ← scan_for(key(e))
  if empty(i) or deleted(i): // wstawianie
    data[i] ← e
    size ← size + 1
    if size * D > |data| * N:
      data ← new_data(2*|data|)
  else: // nadpisanie elementu:
    data[i] ← e
```

N i D to stałe, takie że $\frac{N}{D} < 1$, tj. $N < D$.
Gdy liczba elementów w tablicy (`size`) przekroczy $\frac{N}{D}$ jej pojemności (`|data|`), to pojemność tablicy jest podwajana (następuje rehaszowanie). Dzięki temu w tablicy jest zawsze miejsce w które można wstawić (pusta lub skasowana komórka), więc `scan_for` nie zwróci -1.

Adresowanie otwarte – delete z rehaszowaniem

Usuwa element o kluczu `k`:

```
fun delete(k):
  i ← scan_for(k)
  if not empty(i) and not deleted(i):
    oznacz i-tą komórkę data jako skasowaną
    size ← size - 1
    if |data| > 1 and size*4*D < |data|*N:
      data ← new_data(|data|/2)
```

Gdy liczba elementów w tablicy (`size`) spadnie poniżej $\frac{N}{4D}$ jej pojemności (`|data|`), to pojemność tablicy jest zmniejszana o połowę (następuje rehaszowanie).

Bibliografia

- Thomas H. Cormen, Charles E. Leiserson, Roland L. Rivest, Clifford Stein *Wprowadzenie do algorytmów* (głównie rozdział *Tablice z haszowaniem* w części *Struktury danych*).
- L. Banachowski, K. Diks, W. Rytter *Algorytmy i struktury danych*, Wydawnictwa Naukowo-Techniczne, 2006.
- K. Diks, A. Malinowski, W. Rytter, T. Waleń *Algorytmy i struktury danych/Wyszukiwanie*, WWW: http://wazniak.mimuw.edu.pl/index.php?title=Algorytmy_i_struktury_danych/Wyszukiwanie