Doskonałe funkcje haszujące

Piotr Beling

Uniwersytet Łódzki

2020

http://pbeling.w8.pl

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).

Słowniki z haszowaniem – 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.
- Realizacja słowników z pomocą haszowania przypomina wyżej opisane postępowanie, z tym, że: miejsca oznaczają indeksy w tablicy, zaś za przydzielanie ich elementom odpowiada funkcja haszująca.

Słowniki z haszowaniem – 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.
- Realizacja słowników z pomocą haszowania przypomina wyżej opisane postępowanie, z tym, że: miejsca oznaczają indeksy w tablicy, zaś za przydzielanie ich elementom odpowiada funkcja haszująca.

Słowniki z haszowaniem – 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.
- Realizacja słowników z pomocą haszowania przypomina wyżej opisane postępowanie, z tym, że: miejsca oznaczają indeksy w tablicy, zaś za przydzielanie ich elementom odpowiada funkcja haszująca.

Kolizje

- Jeśli zbiór wszystkich kluczy jest większy niż zbiór indeksów tablicy haszującej, 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 $h(k_1) = h(k_2)$.
- Jednak, by operować jedynie na niewielkim zbiorze kluczy K (ewentualnie będącym podzbiorem większego uniwersum), wystarczy użyć tablicy o rozmiarze $|K|$, pod warunkiem, że dysponujemy tzw. *minimalną, doskonałą funkcją haszującą*.

Kolizje

- Jeśli zbiór wszystkich kluczy jest większy niż zbiór indeksów tablicy haszującej, 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 $h(k_1) = h(k_2)$.
- Jednak, by operować jedynie na niewielkim zbiorze kluczy K (ewentualnie będącym podzbiorem większego uniwersum), wystarczy użyć tablicy o rozmiarze $|K|$, pod warunkiem, że dysponujemy tzw. *minimalną, doskonałą funkcją haszującą*.

Kolizje

- Jeśli zbiór wszystkich kluczy jest większy niż zbiór indeksów tablicy haszującej, 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 $h(k_1) = h(k_2)$.
- Jednak, by operować jedynie na niewielkim zbiorze kluczy K (ewentualnie będącym podzbiorem większego uniwersum), wystarczy użyć tablicy o rozmiarze $|K|$, pod warunkiem, że dysponujemy tzw. *minimalną, doskonałą funkcją haszującą*.

Doskonała funkcja haszująca

- Niech: K – zbiór kluczy.
- Funkcję $h : K \to \{0, 1, \ldots, n-1\}$, gdzie $n \geq |K|$, nazywamy **doskonałą funkcją haszującą** (ang. *perfect hash function*) jeśli h jest różnowartościowa, tj. $k_1 \neq k_2 \implies h(k_1) \neq h(k_2)$ dla dowolnych $k_1, k_2 \in K$.
- Jeśli ponadto $n = |K|$ (h jest wtedy bijekcją) to h jest **minimalną** doskonałą funkcją haszującą (ang. *minimal perfect hash function*).

Doskonała funkcja haszująca

- Niech: K – zbiór kluczy.
- Funkcję $h : K \to \{0, 1, \ldots, n-1\}$, gdzie $n \geq |K|$, nazywamy **doskonałą funkcją haszującą** (ang. *perfect hash function*) jeśli h jest różnowartościowa, tj. $k_1 \neq k_2 \implies h(k_1) \neq h(k_2)$ dla dowolnych $k_1, k_2 \in K$.
- Jeśli ponadto $n = |K|$ (h jest wtedy bijekcją) to h jest **minimalną** doskonałą funkcją haszującą (ang. *minimal perfect hash function*).

Doskonała funkcja haszująca

- Niech: K – zbiór kluczy.
- Funkcję $h : K \to \{0, 1, \ldots, n-1\}$, gdzie $n \geq |K|$, nazywamy **doskonałą funkcją haszującą** (ang. *perfect hash function*) jeśli h jest różnowartościowa, tj. $k_1 \neq k_2 \implies h(k_1) \neq h(k_2)$ dla dowolnych $k_1, k_2 \in K$.
- Jeśli ponadto $n = |K|$ (h jest wtedy bijekcją) to h jest **minimalną** doskonałą funkcją haszującą (ang. *minimal perfect hash function*).

Słownik używający doskonałej funkcji haszującej

- Doskonała funkcja haszująca $h : K \to \{0, 1, \ldots, n-1\}$ gwarantuje brak (problemu) kolizji.
- Dlatego słownik o zbiorze kluczy K można zrealizować za pomocą zwykłej tablicy V o długości n.
- $V[h(k)]$ opisuje klucz $k \in K$.
- Zero-jedynkowa (bitowa) tablica V reprezentuje zbiór kluczy (podzbiór K): $V[h(k)] = 1$ gdy zbiór zawiera k.
- Mapę klucz→wartość reprezentujemy tablicą V w której $V[h(k)]$ zawiera wartość przypisaną do klucza $k \in K$ lub zawiera specjalną wartość oznaczającą brak przypisanej wartości.
  (zamiast specjalnej wartości, zajęte komórki można oznaczyć za pomocą dodatkowej tablicy, bitowej, zero-jedynkowej).

Słownik używający doskonałej funkcji haszującej

- Doskonała funkcja haszująca $h : K \to \{0, 1, \ldots, n-1\}$ gwarantuje brak (problemu) kolizji.
- Dlatego słownik o zbiorze kluczy K można zrealizować za pomocą zwykłej tablicy V o długości n.
- $V[h(k)]$ opisuje klucz $k \in K$.
- Zero-jedynkowa (bitowa) tablica V reprezentuje zbiór kluczy (podzbiór K): $V[h(k)] = 1$ gdy zbiór zawiera k.
- Mapę klucz→wartość reprezentujemy tablicą V w której $V[h(k)]$ zawiera wartość przypisaną do klucza $k \in K$ lub zawiera specjalną wartość oznaczającą brak przypisanej wartości.
  (zamiast specjalnej wartości, zajęte komórki można oznaczyć za pomocą dodatkowej tablicy, bitowej, zero-jedynkowej).

Słownik używający doskonałej funkcji haszującej

- Doskonała funkcja haszująca $h : K \to \{0, 1, \ldots, n-1\}$ gwarantuje brak (problemu) kolizji.
- Dlatego słownik o zbiorze kluczy K można zrealizować za pomocą zwykłej tablicy V o długości n.
- $V[h(k)]$ opisuje klucz $k \in K$.
- Zero-jedynkowa (bitowa) tablica V reprezentuje zbiór kluczy (podzbiór K): $V[h(k)] = 1$ gdy zbiór zawiera k.
- Mapę klucz→wartość reprezentujemy tablicą V w której $V[h(k)]$ zawiera wartość przypisaną do klucza $k \in K$ lub zawiera specjalną wartość oznaczającą brak przypisanej wartości.
  (zamiast specjalnej wartości, zajęte komórki można oznaczyć za pomocą dodatkowej tablicy, bitowej, zero-jedynkowej).

Słownik używający doskonałej funkcji haszującej

- Doskonała funkcja haszująca $h : K \to \{0, 1, \ldots, n-1\}$ gwarantuje brak (problemu) kolizji.
- Dlatego słownik o zbiorze kluczy K można zrealizować za pomocą zwykłej tablicy V o długości n.
- $V[h(k)]$ opisuje klucz $k \in K$.
- Zero-jedynkowa (bitowa) tablica V reprezentuje zbiór kluczy (podzbiór K): $V[h(k)] = 1$ gdy zbiór zawiera k.
- Mapę klucz→wartość reprezentujemy tablicą V w której $V[h(k)]$ zawiera wartość przypisaną do klucza $k \in K$ lub zawiera specjalną wartość oznaczającą brak przypisanej wartości.
  (zamiast specjalnej wartości, zajęte komórki można oznaczyć za pomocą dodatkowej tablicy, bitowej, zero-jedynkowej).

Słownik używający doskonałej funkcji haszującej

- Doskonała funkcja haszująca $h : K \to \{0, 1, \ldots, n-1\}$ gwarantuje brak (problemu) kolizji.
- Dlatego słownik o zbiorze kluczy K można zrealizować za pomocą zwykłej tablicy V o długości n.
- $V[h(k)]$ opisuje klucz $k \in K$.
- Zero-jedynkowa (bitowa) tablica V reprezentuje zbiór kluczy (podzbiór K): $V[h(k)] = 1$ gdy zbiór zawiera k.
- Mapę klucz→wartość reprezentujemy tablicą V w której $V[h(k)]$ zawiera wartość przypisaną do klucza $k \in K$ lub zawiera specjalną wartość oznaczającą brak przypisanej wartości.

  (zamiast specjalnej wartości, zajęte komórki można oznaczyć za pomocą dodatkowej tablicy, bitowej, zero-jedynkowej).

Słownik używający doskonałej funkcji haszującej

- Doskonała funkcja haszująca $h : K \to \{0, 1, \ldots, n-1\}$ gwarantuje brak (problemu) kolizji.
- Dlatego słownik o zbiorze kluczy K można zrealizować za pomocą zwykłej tablicy V o długości n.
- $V[h(k)]$ opisuje klucz $k \in K$.
- Zero-jedynkowa (bitowa) tablica V reprezentuje zbiór kluczy (podzbiór K): $V[h(k)] = 1$ gdy zbiór zawiera k.
- Mapę klucz$\to$wartość reprezentujemy tablicą V w której $V[h(k)]$ zawiera wartość przypisaną do klucza $k \in K$ lub zawiera specjalną wartość oznaczającą brak przypisanej wartości.
  (zamiast specjalnej wartości, zajęte komórki można oznaczyć za pomocą dodatkowej tablicy, bitowej, zero-jedynkowej).

Wyznaczanie doskonałych funkcji haszujących

- Mając dany zbiór kluczy K, można dla niego wyznaczyć minimalną, doskonałą funkcję haszującą h.
- Teoretycznie dowiedziono, że do zapisania funkcji h potrzeba co najmniej, około 1,44 bita/klucz [1, 2, 3],
- niezależnie od reprezentacji elementów K (typu danych).

---

[1] **M. L. Fredman, J. Komlós**, *On the size of separating systems and families of perfect hash functions*, SIAM Journal on Algebraic Discrete Methods 5 (1), 1984

[2] **J. Radhakrishnan**, *Improved bounds for covering complete uniform hypergraphs*, Information Processing Letters 41 (4), 1992.

[3] **D. Belazzougui, F. C. Botelho, M. Dietzfelbinger**, *Hash, displace, and compress*, Algorithms - ESA 2009

Wyznaczanie doskonałych funkcji haszujących

- Mając dany zbiór kluczy K, można dla niego wyznaczyć minimalną, doskonałą funkcję haszującą h.
- Teoretycznie dowiedziono, że do zapisania funkcji h potrzeba co najmniej, około 1,44 bita/klucz [1, 2, 3],
- niezależnie od reprezentacji elementów K (typu danych).

---

[1] **M. L. Fredman, J. Komlós**, *On the size of separating systems and families of perfect hash functions*, SIAM Journal on Algebraic Discrete Methods 5 (1), 1984

[2] **J. Radhakrishnan**, *Improved bounds for covering complete uniform hypergraphs*, Information Processing Letters 41 (4), 1992.

[3] **D. Belazzougui, F. C. Botelho, M. Dietzfelbinger**, *Hash, displace, and compress*, Algorithms - ESA 2009

Wyznaczanie doskonałych funkcji haszujących

- Mając dany zbiór kluczy K, można dla niego wyznaczyć minimalną, doskonałą funkcję haszującą h.
- Teoretycznie dowiedziono, że do zapisania funkcji h potrzeba co najmniej, około 1,44 bita/klucz [1, 2, 3],
- niezależnie od reprezentacji elementów K (typu danych).

---

[1] **M. L. Fredman, J. Komlós**, *On the size of separating systems and families of perfect hash functions*, SIAM Journal on Algebraic Discrete Methods 5 (1), 1984

[2] **J. Radhakrishnan**, *Improved bounds for covering complete uniform hypergraphs*, Information Processing Letters 41 (4), 1992.

[3] **D. Belazzougui, F. C. Botelho, M. Dietzfelbinger**, *Hash, displace, and compress*, Algorithms - ESA 2009

Wyznaczanie doskonałych funkcji haszujących

Istnieje szereg algorytmów, które potrafią wyznaczyć minimalną, doskonałą funkcję haszującą h dla zadanego zbioru kluczy K, np.:

- RecSplit – zapisuje h używając około 1,8 bita/klucz;
  E. Esposito, T. M. Graf, S. Vigna, *RecSplit: Minimal Perfect Hashing via Recursive Splitting*, 2019
- CHD – zapisuje h używając około 2,1 bita/klucz;
  D. Belazzougui, F. C. Botelho, M. Dietzfelbinger, *Hash, displace, and compress*, Algorithms - ESA 2009
- BBHash (*fingerprinting* lub *bloom-filter based*)
  - zapisuje h używając około 3 bitów/klucz,
  - jest prosty (dalej opiszemy jak działa);

  J. A. Chapman, I. Ho, S. Sunkara, S. Luo, G. P. Schroth, D. S. Rokhsar, *Meraculous: De novo genome assembly with short paired-end reads*, PLOS ONE 6 (8) 1–13, 08.2011

Oczekiwana złożoność czasowa wymienionych algorytmów to: $O(|K|)$ – skonstruowanie h, $O(1)$ – wyznaczenia wartości h.

Wyznaczanie doskonałych funkcji haszujących

Istnieje szereg algorytmów, które potrafią wyznaczyć minimalną, doskonałą funkcję haszującą h dla zadanego zbioru kluczy K, np.:

- RecSplit – zapisuje h używając około 1,8 bita/klucz;
  E. Esposito, T. M. Graf, S. Vigna, *RecSplit: Minimal Perfect Hashing via Recursive Splitting*, 2019
- CHD – zapisuje h używając około 2,1 bita/klucz;
  D. Belazzougui, F. C. Botelho, M. Dietzfelbinger, *Hash, displace, and compress*, Algorithms - ESA 2009
- BBHash (*fingerprinting* lub *bloom-filter based*)
  - zapisuje h używając około 3 bitów/klucz,
  - jest prosty (dalej opiszemy jak działa);

  J. A. Chapman, I. Ho, S. Sunkara, S. Luo, G. P. Schroth, D. S. Rokhsar, *Meraculous: De novo genome assembly with short paired-end reads*, PLOS ONE 6 (8) 1–13, 08.2011

Oczekiwana złożoność czasowa wymienionych algorytmów to: $O(|K|)$ – skonstruowanie h, $O(1)$ – wyznaczenia wartości h.

Wyznaczanie doskonałych funkcji haszujących

Istnieje szereg algorytmów, które potrafią wyznaczyć minimalną, doskonałą funkcję haszującą h dla zadanego zbioru kluczy K, np.:

- RecSplit – zapisuje h używając około 1,8 bita/klucz;
  E. Esposito, T. M. Graf, S. Vigna, *RecSplit: Minimal Perfect Hashing via Recursive Splitting*, 2019
- CHD – zapisuje h używając około 2,1 bita/klucz;
  D. Belazzougui, F. C. Botelho, M. Dietzfelbinger, *Hash, displace, and compress*, Algorithms - ESA 2009
- BBHash (*fingerprinting* lub *bloom-filter based*)
  - zapisuje h używając około 3 bitów/klucz,
  - jest prosty (dalej opiszemy jak działa);

  J. A. Chapman, I. Ho, S. Sunkara, S. Luo, G. P. Schroth, D. S. Rokhsar, *Meraculous: De novo genome assembly with short paired-end reads*, PLOS ONE 6 (8) 1–13, 08.2011

Oczekiwana złożoność czasowa wymienionych algorytmów to:
$O(|K|)$ – skonstruowanie h, $O(1)$ – wyznaczenia wartości h.

Wyznaczanie doskonałych funkcji haszujących

Istnieje szereg algorytmów, które potrafią wyznaczyć minimalną, doskonałą funkcję haszującą h dla zadanego zbioru kluczy K, np.:

- RecSplit – zapisuje h używając około 1,8 bita/klucz;
  E. Esposito, T. M. Graf, S. Vigna, *RecSplit: Minimal Perfect Hashing via Recursive Splitting*, 2019
- CHD – zapisuje h używając około 2,1 bita/klucz;
  D. Belazzougui, F. C. Botelho, M. Dietzfelbinger, *Hash, displace, and compress*, Algorithms - ESA 2009
- BBHash (*fingerprinting* lub *bloom-filter based*)
  - zapisuje h używając około 3 bitów/klucz,
  - jest prosty (dalej opiszemy jak działa);

  J. A. Chapman, I. Ho, S. Sunkara, S. Luo, G. P. Schroth, D. S. Rokhsar, *Meraculous: De novo genome assembly with short paired-end reads*, PLOS ONE 6 (8) 1–13, 08.2011

Oczekiwana złożoność czasowa wymienionych algorytmów to: $O(|K|)$ – skonstruowanie h, $O(1)$ – wyznaczenia wartości h.

BBHash – oznaczenia

Niech $h_0, h_1, \ldots$

- to klasyczne, niezależne od siebie funkcje haszujące,
- które przyporządkowują liczby naturalne (w zakresie od 0 do pożądanej wartości) do elementów K
- i (możliwie dobrze) spełniają założenie o prostym, równomiernym haszowaniu,
  tj. losowo wybranemu $k \in K$ z jednakowym prawdopodobieństwem przypisywana jest każda z możliwych wartości.

Przykłady dobrych funkcji do użycia w praktyce:

- xxHash (Yann Collet) www.xxhash.com
- SipHash (Jean-Philippe Aumasson i Daniel J. Bernstein)

BBHash – oznaczenia

Niech $h_0, h_1, \ldots$

- to klasyczne, niezależne od siebie funkcje haszujące,
- które przyporządkowują liczby naturalne (w zakresie od 0 do pożądanej wartości) do elementów K
- i (możliwie dobrze) spełniają założenie o prostym, równomiernym haszowaniu,
  tj. losowo wybranemu $k \in K$ z jednakowym prawdopodobieństwem przypisywana jest każda z możliwych wartości.

Przykłady dobrych funkcji do użycia w praktyce:

- xxHash (Yann Collet) www.xxhash.com
- SipHash (Jean-Philippe Aumasson i Daniel J. Bernstein)

BBHash – oznaczenia

Niech $h_0, h_1, \ldots$

- to klasyczne, niezależne od siebie funkcje haszujące,
- które przyporządkowują liczby naturalne (w zakresie od 0 do pożądanej wartości) do elementów K
- i (możliwie dobrze) spełniają założenie o prostym, równomiernym haszowaniu,
  tj. losowo wybranemu $k \in K$ z jednakowym prawdopodobieństwem przypisywana jest każda z możliwych wartości.

Przykłady dobrych funkcji do użycia w praktyce:

- xxHash (Yann Collet) www.xxhash.com
- SipHash (Jean-Philippe Aumasson i Daniel J. Bernstein)

BBHash – oznaczenia

Niech $h_0, h_1, \ldots$

- to klasyczne, niezależne od siebie funkcje haszujące,
- które przyporządkowują liczby naturalne (w zakresie od 0 do pożądanej wartości) do elementów K
- i (możliwie dobrze) spełniają założenie o prostym, równomiernym haszowaniu,
  tj. losowo wybranemu $k \in K$ z jednakowym prawdopodobieństwem przypisywana jest każda z możliwych wartości.

Przykłady dobrych funkcji do użycia w praktyce:

- xxHash (Yann Collet) www.xxhash.com
- SipHash (Jean-Philippe Aumasson i Daniel J. Bernstein)

BBHash – oznaczenia

Niech $h_0, h_1, \ldots$

- to klasyczne, niezależne od siebie funkcje haszujące,
- które przyporządkowują liczby naturalne (w zakresie od 0 do pożądanej wartości) do elementów K
- i (możliwie dobrze) spełniają założenie o prostym, równomiernym haszowaniu,
  tj. losowo wybranemu $k \in K$ z jednakowym prawdopodobieństwem przypisywana jest każda z możliwych wartości.

Przykłady dobrych funkcji do użycia w praktyce:

- xxHash (Yann Collet) www.xxhash.com
- SipHash (Jean-Philippe Aumasson i Daniel J. Bernstein)

BBHash – konstrukcja funkcji h

Niech $K_0 = K$. Dla kolejnych poziomów $l = 0, 1, \ldots$:

1. Stwórz tablicę bitową A_l o długości co najmniej $|K_l|$. (dobrze jest przyjąć np. $|A_l| = |K_l|$)
2. Zaznacz w A_l komórki do których funkcja $h_l : K_l \to [0, |A_l| - 1]$ bezkolizyjnie przyporządkowuje elementy K_l, tj. ustaw $A[i] = 1$ wtedy i tylko wtedy gdy $h_l(k) = i$ dla dokładnie jednego $k \in K_l$.
3. Oblicz zbiór kluczy, których wciąż nie udało się bezkolizyjnie przyporządkować:

$$K_{l+1} = \{k \in K_l : A_l[h_l(k)] = 0\}.$$

4. Zakończ gdy $K_{l+1} = \emptyset$. Ciąg $A_0, A_1, \ldots, A_l$ koduje funkcję h.

BBHash – konstrukcja funkcji h

Niech $K_0 = K$. Dla kolejnych poziomów $l = 0, 1, \ldots$:

1. Stwórz tablicę bitową A_l o długości co najmniej $|K_l|$. (dobrze jest przyjąć np. $|A_l| = |K_l|$)
2. Zaznacz w A_l komórki do których funkcja $h_l : K_l \to [0, |A_l| - 1]$ bezkolizyjnie przyporządkowuje elementy K_l, tj. ustaw $A[i] = 1$ wtedy i tylko wtedy gdy $h_l(k) = i$ dla dokładnie jednego $k \in K_l$.
3. Oblicz zbiór kluczy, których wciąż nie udało się bezkolizyjnie przyporządkować:

$$K_{l+1} = \{k \in K_l : A_l[h_l(k)] = 0\}.$$

4. Zakończ gdy $K_{l+1} = \emptyset$. Ciąg $A_0, A_1, \ldots, A_l$ koduje funkcję h.

BBHash – konstrukcja funkcji h

Niech $K_0 = K$. Dla kolejnych poziomów $l = 0, 1, \ldots$:

1. Stwórz tablicę bitową A_l o długości co najmniej $|K_l|$. (dobrze jest przyjąć np. $|A_l| = |K_l|$)
2. Zaznacz w A_l komórki do których funkcja $h_l : K_l \to [0, |A_l| - 1]$ bezkolizyjnie przyporządkowuje elementy K_l, tj. ustaw $A[i] = 1$ wtedy i tylko wtedy gdy $h_l(k) = i$ dla dokładnie jednego $k \in K_l$.
3. Oblicz zbiór kluczy, których wciąż nie udało się bezkolizyjnie przyporządkować:

$$K_{l+1} = \{k \in K_l : A_l[h_l(k)] = 0\}.$$

4. Zakończ gdy $K_{l+1} = \emptyset$. Ciąg $A_0, A_1, \ldots, A_l$ koduje funkcję h.

BBHash – konstrukcja funkcji h

Niech $K_0 = K$. Dla kolejnych poziomów $l = 0, 1, \ldots$:

1. Stwórz tablicę bitową A_l o długości co najmniej $|K_l|$. (dobrze jest przyjąć np. $|A_l| = |K_l|$)
2. Zaznacz w A_l komórki do których funkcja $h_l : K_l \to [0, |A_l| - 1]$ bezkolizyjnie przyporządkowuje elementy K_l, tj. ustaw $A[i] = 1$ wtedy i tylko wtedy gdy $h_l(k) = i$ dla dokładnie jednego $k \in K_l$.
3. Oblicz zbiór kluczy, których wciąż nie udało się bezkolizyjnie przyporządkować:

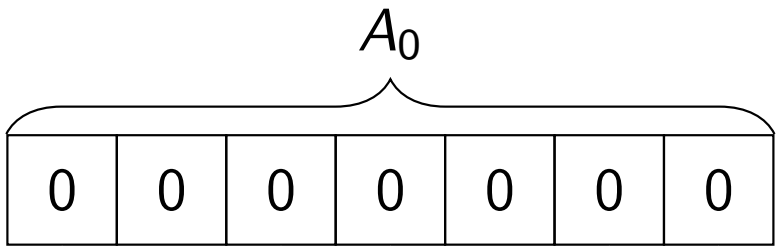
$$K_{l+1} = \{k \in K_l : A_l[h_l(k)] = 0\}.$$

4. Zakończ gdy $K_{l+1} = \emptyset$. Ciąg $A_0, A_1, \ldots, A_l$ koduje funkcję h.

BBHash – przykład konstrukcji funkcji

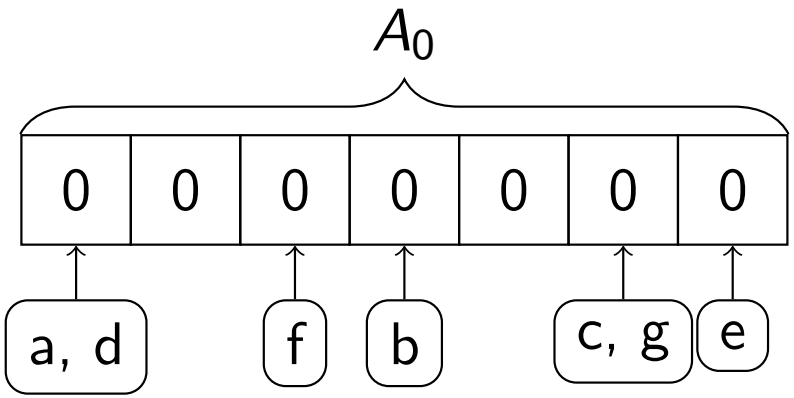
- Prześledźmy proces konstrukcji funkcji h dla zbioru kluczy: $K = K_0 = \{a, b, c, d, e, f, g\}$.
- Wpierw tworzymy tablicę bitową A_0.
- Za pomocą funkcji $h_0 : K_0 \to [0, |A_0| - 1]$ przypisujemy kluczom indeksy tablicy A_0.
- Zaznaczamy w A_0 komórki do których funkcja h_0 bezkolizyjnie przyporządkowała elementy, tj. $A[i] = 1$ wtedy i tylko wtedy gdy $h_0(k) = i$ dla dokładnie jednego $k \in K_0$.
- Zaznaczono jedynie bity przypisane literom f, b, oraz e.

BBHash – przykład konstrukcji funkcji



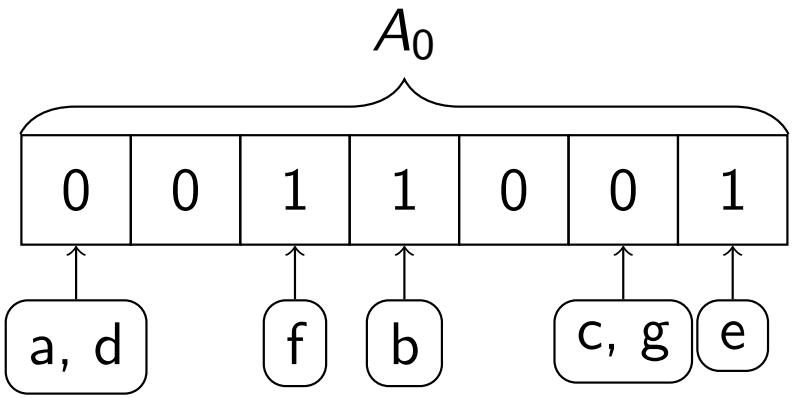
- Prześledźmy proces konstrukcji funkcji h dla zbioru kluczy: $K = K_0 = \{a, b, c, d, e, f, g\}$.
- Wpierw tworzymy tablicę bitową A_0.
- Za pomocą funkcji $h_0 : K_0 \to [0, |A_0| - 1]$ przypisujemy kluczom indeksy tablicy A_0.
- Zaznaczamy w A_0 komórki do których funkcja h_0 bezkolizyjnie przyporządkowała elementy, tj. $A[i] = 1$ wtedy i tylko wtedy gdy $h_0(k) = i$ dla dokładnie jednego $k \in K_0$.
- Zaznaczono jedynie bity przypisane literom f, b, oraz e.

BBHash – przykład konstrukcji funkcji



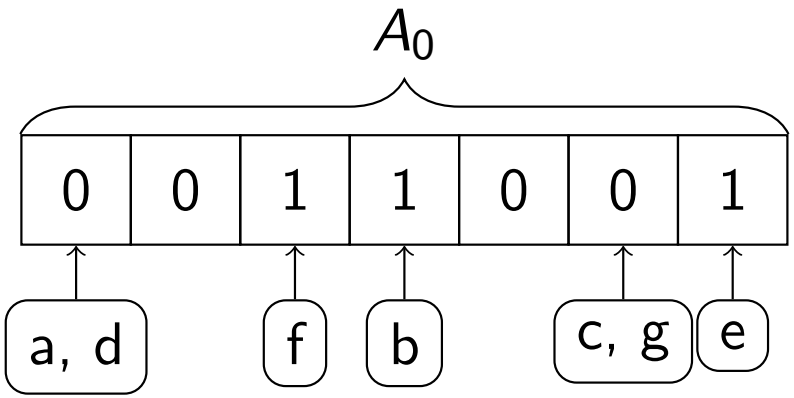
- Prześledźmy proces konstrukcji funkcji h dla zbioru kluczy: $K = K_0 = \{a, b, c, d, e, f, g\}$.
- Wpierw tworzymy tablicę bitową A_0.
- Za pomocą funkcji $h_0 : K_0 \to [0, |A_0| - 1]$ przypisujemy kluczom indeksy tablicy A_0.
- Zaznaczamy w A_0 komórki do których funkcja h_0 bezkolizyjnie przyporządkowała elementy, tj. $A[i] = 1$ wtedy i tylko wtedy gdy $h_0(k) = i$ dla dokładnie jednego $k \in K_0$.
- Zaznaczono jedynie bity przypisane literom f, b, oraz e.

BBHash – przykład konstrukcji funkcji



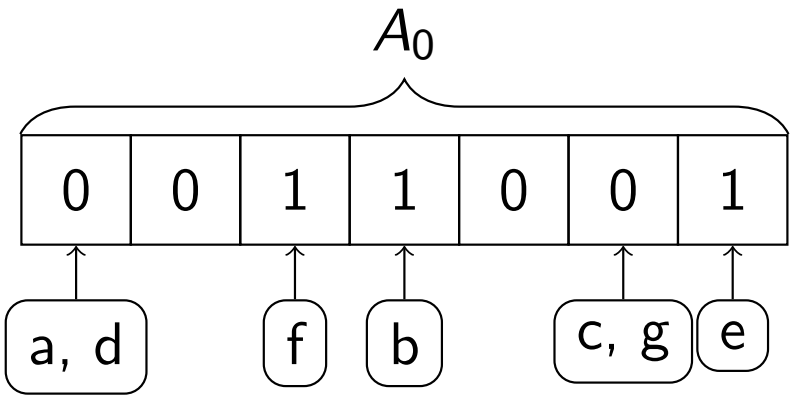
- Prześledźmy proces konstrukcji funkcji h dla zbioru kluczy: $K = K_0 = \{a, b, c, d, e, f, g\}$.
- Wpierw tworzymy tablicę bitową A_0.
- Za pomocą funkcji $h_0 : K_0 \to [0, |A_0| - 1]$ przypisujemy kluczom indeksy tablicy A_0.
- Zaznaczamy w A_0 komórki do których funkcja h_0 bezkolizyjnie przyporządkowała elementy, tj. $A[i] = 1$ wtedy i tylko wtedy gdy $h_0(k) = i$ dla dokładnie jednego $k \in K_0$.
- Zaznaczono jedynie bity przypisane literom f, b, oraz e.

BBHash – przykład konstrukcji funkcji



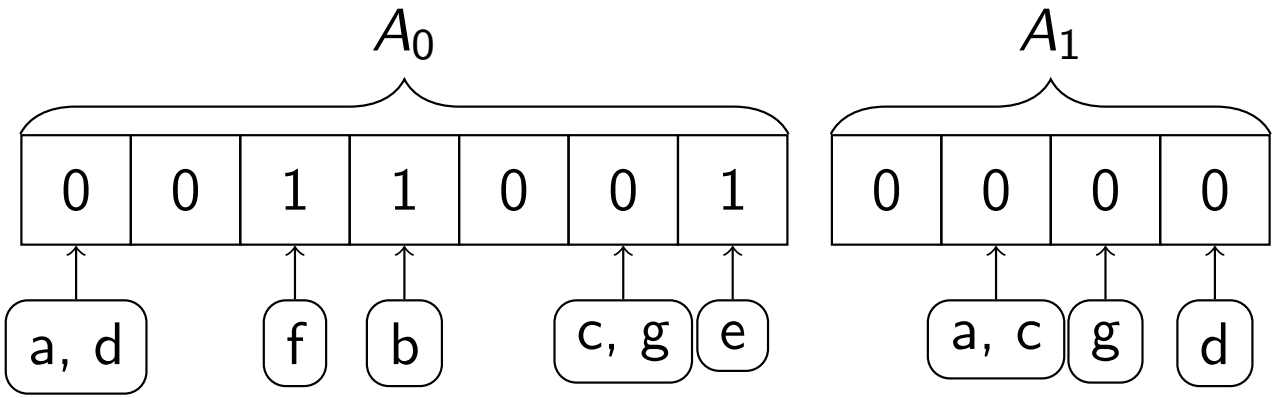
- Prześledźmy proces konstrukcji funkcji h dla zbioru kluczy: $K = K_0 = \{a, b, c, d, e, f, g\}$.
- Wpierw tworzymy tablicę bitową A_0.
- Za pomocą funkcji $h_0 : K_0 \to [0, |A_0| - 1]$ przypisujemy kluczom indeksy tablicy A_0.
- Zaznaczamy w A_0 komórki do których funkcja h_0 bezkolizyjnie przyporządkowała elementy, tj. $A[i] = 1$ wtedy i tylko wtedy gdy $h_0(k) = i$ dla dokładnie jednego $k \in K_0$.
- Zaznaczono jedynie bity przypisane literom f, b, oraz e.

BBHash – przykład konstrukcji funkcji



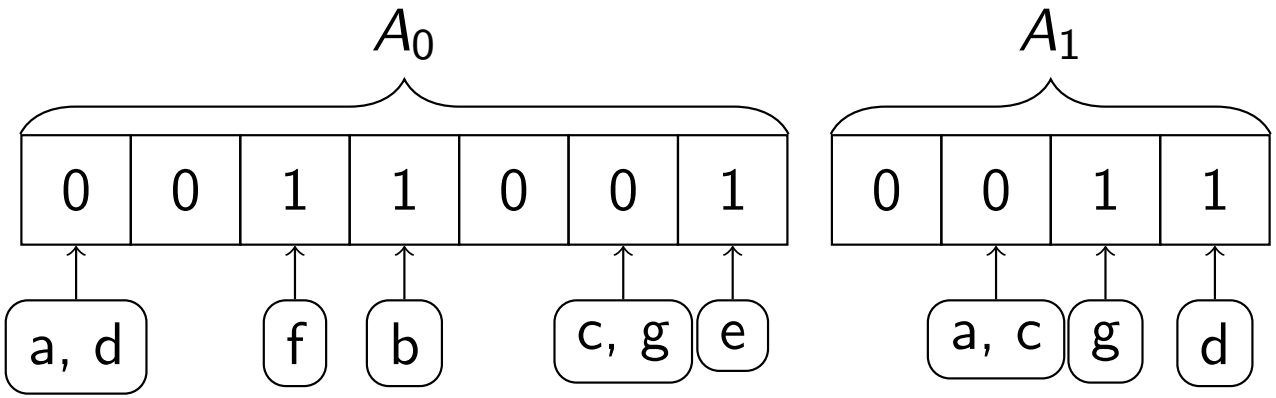
- Pozostałe litery tworzą zbiór $K_1 = \{k \in K_0 : A_0[h_0(k)] = 0\} = \{a, c, d, g\}$, który będzie zakodowany na kolejnych poziomach.
- Tworzymy tablicę A_1 i za pomocą funkcji h_1 przypisujemy indeksy tej tablicy do elementów K_1.
- Tym razem bezkolizyjnie udało się przypisać litery d oraz g. Przypisane im komórki A_1 ustawiamy na 1.
- Pozostałe litery trafiają do $K_2 = \{a, c\}$.

BBHash – przykład konstrukcji funkcji



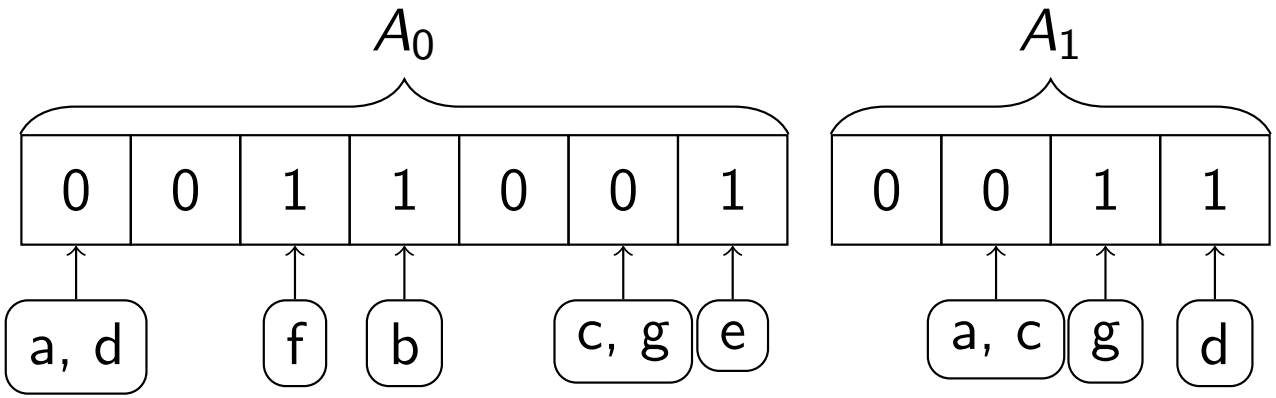
- Pozostałe litery tworzą zbiór $K_1 = \{k \in K_0 : A_0[h_0(k)] = 0\} = \{a, c, d, g\}$, który będzie zakodowany na kolejnych poziomach.
- Tworzymy tablicę A_1 i za pomocą funkcji h_1 przypisujemy indeksy tej tablicy do elementów K_1.
- Tym razem bezkolizyjnie udało się przypisać litery d oraz g. Przypisane im komórki A_1 ustawiamy na 1.
- Pozostałe litery trafiają do $K_2 = \{a, c\}$.

BBHash – przykład konstrukcji funkcji



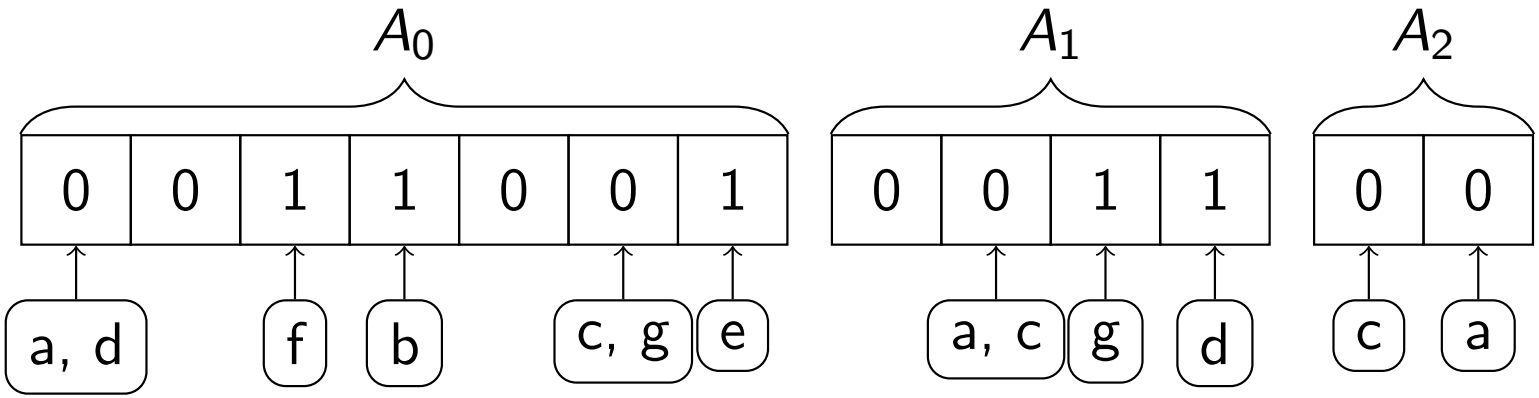
- Pozostałe litery tworzą zbiór $K_1 = \{k \in K_0 : A_0[h_0(k)] = 0\} = \{a, c, d, g\}$, który będzie zakodowany na kolejnych poziomach.
- Tworzymy tablicę A_1 i za pomocą funkcji h_1 przypisujemy indeksy tej tablicy do elementów K_1.
- Tym razem bezkolizyjnie udało się przypisać litery d oraz g. Przypisane im komórki A_1 ustawiamy na 1.
- Pozostałe litery trafiają do $K_2 = \{a, c\}$.

BBHash – przykład konstrukcji funkcji



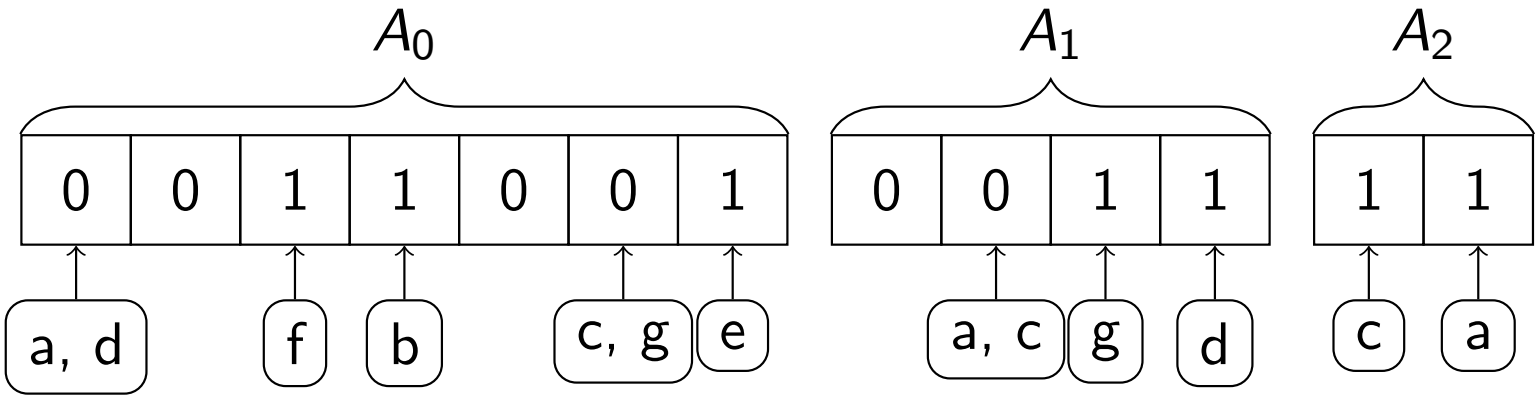
- Pozostałe litery tworzą zbiór $K_1 = \{k \in K_0 : A_0[h_0(k)] = 0\} = \{a, c, d, g\}$, który będzie zakodowany na kolejnych poziomach.
- Tworzymy tablicę A_1 i za pomocą funkcji h_1 przypisujemy indeksy tej tablicy do elementów K_1.
- Tym razem bezkolizyjnie udało się przypisać litery d oraz g. Przypisane im komórki A_1 ustawiamy na 1.
- Pozostałe litery trafiają do $K_2 = \{a, c\}$.

BBHash – przykład konstrukcji funkcji



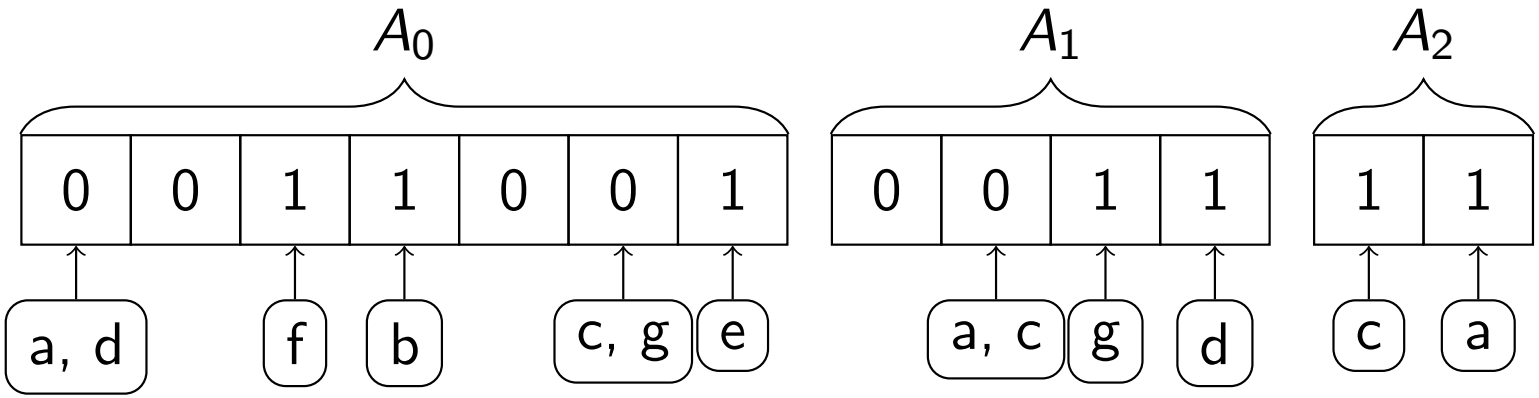
- Tworzymy tablicę A_2 i za pomocą funkcji h_2 przypisujemy indeksy tej tablicy do elementów K_2.
- Tym razem udało się bezkolizyjnie przypisać wszystkie litery (a oraz c). Przypisane im komórki A_1 ustawiamy na 1.
- Algorytm kończy, gdyż wszystkie klucze zostały już przypisane, tj. $K_3 = \{k \in K_2 : A_2[h_2(k)] = 0\} = \emptyset$.
- Tablice A_0, A_1, A_2 reprezentują utworzoną funkcję h. Zazwyczaj pamiętana jest ich konkatenacja A.

BBHash – przykład konstrukcji funkcji



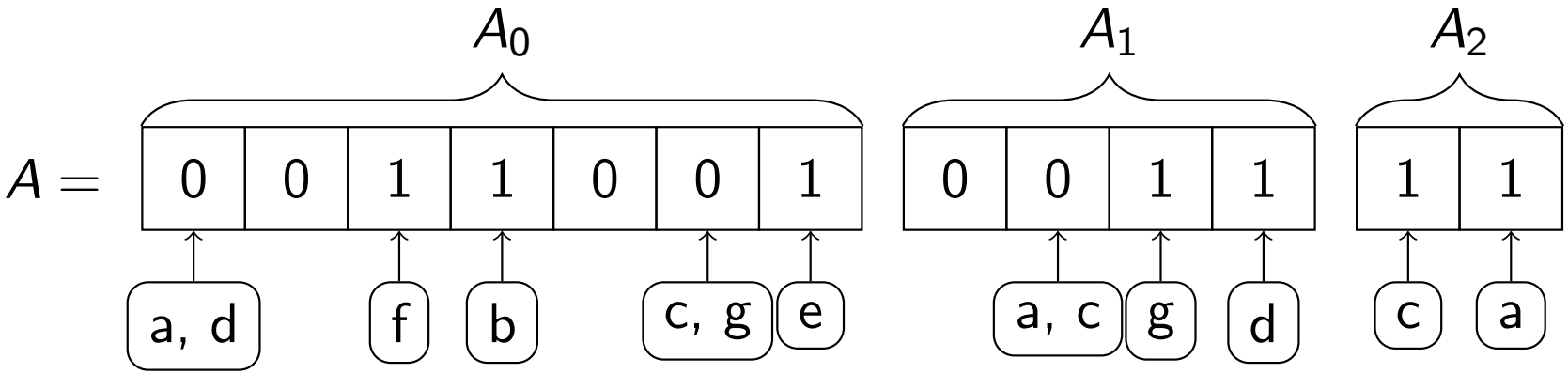
- Tworzymy tablicę A_2 i za pomocą funkcji h_2 przypisujemy indeksy tej tablicy do elementów K_2.
- Tym razem udało się bezkolizyjnie przypisać wszystkie litery (a oraz c). Przypisane im komórki A_1 ustawiamy na 1.
- Algorytm kończy, gdyż wszystkie klucze zostały już przypisane, tj. $K_3 = \{k \in K_2 : A_2[h_2(k)] = 0\} = \emptyset$.
- Tablice A_0, A_1, A_2 reprezentują utworzoną funkcję h. Zazwyczaj pamiętana jest ich konkatenacja A.

BBHash – przykład konstrukcji funkcji



- Tworzymy tablicę A_2 i za pomocą funkcji h_2 przypisujemy indeksy tej tablicy do elementów K_2.
- Tym razem udało się bezkolizyjnie przypisać wszystkie litery (a oraz c). Przypisane im komórki A_1 ustawiamy na 1.
- Algorytm kończy, gdyż wszystkie klucze zostały już przypisane, tj. $K_3 = \{k \in K_2 : A_2[h_2(k)] = 0\} = \emptyset$.
- Tablice A_0, A_1, A_2 reprezentują utworzoną funkcję h. Zazwyczaj pamiętana jest ich konkatenacja A.

BBHash – przykład konstrukcji funkcji



- Tworzymy tablicę A_2 i za pomocą funkcji h_2 przypisujemy indeksy tej tablicy do elementów K_2.
- Tym razem udało się bezkolizyjnie przypisać wszystkie litery (a oraz c). Przypisane im komórki A_1 ustawiamy na 1.
- Algorytm kończy, gdyż wszystkie klucze zostały już przypisane, tj. $K_3 = \{k \in K_2 : A_2[h_2(k)] = 0\} = \emptyset$.
- Tablice A_0, A_1, A_2 reprezentują utworzoną funkcję h. Zazwyczaj pamiętana jest ich konkatenacja A.

BBHash – reprezentacja funkcji h

Funkcję h zazwyczaj reprezentuje się za pomocą:

- tablicy bitowej A będącą konkatenacją tablic $A_0, A_1, \ldots, A_l$;
- wielkości $|A_0|, |A_1|, \ldots, |A_l|$;
- i struktury pomocniczej do szybkiego wyznaczenia wartości funkcji rank, gdzie $\text{rank}(A, i) = \sum_{b=0}^{i-1} A[b]$ to liczba jedynek w A, na pozycjach od 0 włącznie do i wyłącznie.

Przykładowo, jako struktury pomocniczej można użyć $R = [\text{rank}(A, 0 \cdot 512), \text{rank}(A, 1 \cdot 512), \text{rank}(A, 2 \cdot 512), \ldots]$ i $\text{rank}(A, i)$ wyznaczać jako sumę $R[\lfloor i/512 \rfloor]$ i $\sum_{b=512 \cdot \lfloor i/512 \rfloor}^{i-1} A[b]$ (współczesne procesory mają instrukcję *popcount* zliczającą liczbę ustawionych bitów w 64-bitowym słowie).

Lepszą strukturę zaproponowano w publikacji:
Dong Zhou, David G. Andersen, Michael Kaminsky
Space-Efficient, High-Performance Rank and Select Structures on Uncompressed Bit Sequences, Experimental Algorithms, 2013.

BBHash – reprezentacja funkcji h

Funkcję h zazwyczaj reprezentuje się za pomocą:

- tablicy bitowej A będącą konkatenacją tablic $A_0, A_1, \ldots, A_l$;
- wielkości $|A_0|, |A_1|, \ldots, |A_l|$;
- i struktury pomocniczej do szybkiego wyznaczenia wartości funkcji rank, gdzie $\text{rank}(A, i) = \sum_{b=0}^{i-1} A[b]$ to liczba jedynek w A, na pozycjach od 0 włącznie do i wyłącznie.

Przykładowo, jako struktury pomocniczej można użyć $R = [\text{rank}(A, 0 \cdot 512), \text{rank}(A, 1 \cdot 512), \text{rank}(A, 2 \cdot 512), \ldots]$ i $\text{rank}(A, i)$ wyznaczać jako sumę $R[\lfloor i/512 \rfloor]$ i $\sum_{b=512 \cdot \lfloor i/512 \rfloor}^{i-1} A[b]$ (współczesne procesory mają instrukcję *popcount* zliczającą liczbę ustawionych bitów w 64-bitowym słowie).

Lepszą strukturę zaproponowano w publikacji:
Dong Zhou, David G. Andersen, Michael Kaminsky
Space-Efficient, High-Performance Rank and Select Structures on Uncompressed Bit Sequences, Experimental Algorithms, 2013.

BBHash – reprezentacja funkcji h

Funkcję h zazwyczaj reprezentuje się za pomocą:

- tablicy bitowej A będącą konkatenacją tablic $A_0, A_1, \ldots, A_l$;
- wielkości $|A_0|, |A_1|, \ldots, |A_l|$;
- i struktury pomocniczej do szybkiego wyznaczenia wartości funkcji rank, gdzie $\text{rank}(A, i) = \sum_{b=0}^{i-1} A[b]$ to liczba jedynek w A, na pozycjach od 0 włącznie do i wyłącznie.

Przykładowo, jako struktury pomocniczej można użyć $R = [\text{rank}(A, 0 \cdot 512), \text{rank}(A, 1 \cdot 512), \text{rank}(A, 2 \cdot 512), \ldots]$ i $\text{rank}(A, i)$ wyznaczać jako sumę $R[\lfloor i/512 \rfloor]$ i $\sum_{b=512 \cdot \lfloor i/512 \rfloor}^{i-1} A[b]$ (współczesne procesory mają instrukcję *popcount* zliczającą liczbę ustawionych bitów w 64-bitowym słowie).

Lepszą strukturę zaproponowano w publikacji:
Dong Zhou, David G. Andersen, Michael Kaminsky
Space-Efficient, High-Performance Rank and Select Structures on Uncompressed Bit Sequences, Experimental Algorithms, 2013.

BBHash – reprezentacja funkcji h

Funkcję h zazwyczaj reprezentuje się za pomocą:

- tablicy bitowej A będącą konkatenacją tablic $A_0, A_1, \ldots, A_l$;
- wielkości $|A_0|, |A_1|, \ldots, |A_l|$;
- i struktury pomocniczej do szybkiego wyznaczenia wartości funkcji rank, gdzie $\text{rank}(A, i) = \sum_{b=0}^{i-1} A[b]$ to liczba jedynek w A, na pozycjach od 0 włącznie do i wyłącznie.

Przykładowo, jako struktury pomocniczej można użyć
$R = [\text{rank}(A, 0 \cdot 512), \text{rank}(A, 1 \cdot 512), \text{rank}(A, 2 \cdot 512), \ldots]$
i $\text{rank}(A, i)$ wyznaczać jako sumę $R[\lfloor i/512 \rfloor]$ i $\sum_{b=512 \cdot \lfloor i/512 \rfloor}^{i-1} A[b]$
(współczesne procesory mają instrukcję *popcount* zliczającą liczbę ustawionych bitów w 64-bitowym słowie).

Lepszą strukturę zaproponowano w publikacji:
Dong Zhou, David G. Andersen, Michael Kaminsky
Space-Efficient, High-Performance Rank and Select Structures on Uncompressed Bit Sequences, Experimental Algorithms, 2013.

BBHash – reprezentacja funkcji h

Funkcję h zazwyczaj reprezentuje się za pomocą:

- tablicy bitowej A będącą konkatenacją tablic $A_0, A_1, \ldots, A_l$;
- wielkości $|A_0|, |A_1|, \ldots, |A_l|$;
- i struktury pomocniczej do szybkiego wyznaczenia wartości funkcji rank, gdzie $\text{rank}(A, i) = \sum_{b=0}^{i-1} A[b]$ to liczba jedynek w A, na pozycjach od 0 włącznie do i wyłącznie.

Przykładowo, jako struktury pomocniczej można użyć
$R = [\text{rank}(A, 0 \cdot 512), \text{rank}(A, 1 \cdot 512), \text{rank}(A, 2 \cdot 512), \ldots]$
i $\text{rank}(A, i)$ wyznaczać jako sumę $R[\lfloor i/512 \rfloor]$ i $\sum_{b=512 \cdot \lfloor i/512 \rfloor}^{i-1} A[b]$
(współczesne procesory mają instrukcję *popcount* zliczającą liczbę ustawionych bitów w 64-bitowym słowie).

Lepszą strukturę zaproponowano w publikacji:
Dong Zhou, David G. Andersen, Michael Kaminsky
Space-Efficient, High-Performance Rank and Select Structures on Uncompressed Bit Sequences, Experimental Algorithms, 2013.

BBHash – reprezentacja funkcji h

Funkcję h zazwyczaj reprezentuje się za pomocą:

- tablicy bitowej A będącą konkatenacją tablic $A_0, A_1, \ldots, A_l$;
- wielkości $|A_0|, |A_1|, \ldots, |A_l|$;
- i struktury pomocniczej do szybkiego wyznaczenia wartości funkcji rank, gdzie $\text{rank}(A, i) = \sum_{b=0}^{i-1} A[b]$ to liczba jedynek w A, na pozycjach od 0 włącznie do i wyłącznie.

Przykładowo, jako struktury pomocniczej można użyć $R = [\text{rank}(A, 0 \cdot 512), \text{rank}(A, 1 \cdot 512), \text{rank}(A, 2 \cdot 512), \ldots]$ i $\text{rank}(A, i)$ wyznaczać jako sumę $R[\lfloor i/512 \rfloor]$ i $\sum_{b=512 \cdot \lfloor i/512 \rfloor}^{i-1} A[b]$ (współczesne procesory mają instrukcję *popcount* zliczającą liczbę ustawionych bitów w 64-bitowym słowie).

Lepszą strukturę zaproponowano w publikacji:
Dong Zhou, David G. Andersen, Michael Kaminsky
Space-Efficient, High-Performance Rank and Select Structures on Uncompressed Bit Sequences, Experimental Algorithms, 2013.

BBHash – reprezentacja funkcji h

Funkcję h zazwyczaj reprezentuje się za pomocą:

- tablicy bitowej A będącą konkatenacją tablic $A_0, A_1, \ldots, A_l$;
- wielkości $|A_0|, |A_1|, \ldots, |A_l|$;
- i struktury pomocniczej do szybkiego wyznaczenia wartości funkcji rank, gdzie $\text{rank}(A, i) = \sum_{b=0}^{i-1} A[b]$ to liczba jedynek w A, na pozycjach od 0 włącznie do i wyłącznie.

Przykładowo, jako struktury pomocniczej można użyć $R = [\text{rank}(A, 0 \cdot 512), \text{rank}(A, 1 \cdot 512), \text{rank}(A, 2 \cdot 512), \ldots]$ i $\text{rank}(A, i)$ wyznaczać jako sumę $R[\lfloor i/512 \rfloor]$ i $\sum_{b=512 \cdot \lfloor i/512 \rfloor}^{i-1} A[b]$ (współczesne procesory mają instrukcję *popcount* zliczającą liczbę ustawionych bitów w 64-bitowym słowie).

Lepszą strukturę zaproponowano w publikacji:
Dong Zhou, David G. Andersen, Michael Kaminsky
Space-Efficient, High-Performance Rank and Select Structures on Uncompressed Bit Sequences, Experimental Algorithms, 2013.

BBHash – wyznaczanie wartości $h(k)$

- W A jest dokładnie $|K|$ jedynek,
- każda odpowiada dokładnie jednemu kluczowi z K.
- Dla $k \in K$, wyznaczanie wartości $h(k)$ zaczynamy od znalezienia indeksu i zawierającego jedynkę ($A[i] = 1$) odpowiadającą kluczowi k.
  Ściślej $i = \sum_{p=0}^{l-1} |A_p| + h_l(k)$, gdzie l to najniższy numeru poziomu, taki że $A_l[h_l(k)] = 1$.
- Zwracamy $\text{rank}(A, i)$, gdzie $\text{rank}(A, i) = \sum_{b=0}^{i-1} A[b]$ to liczba jedynek w A, na pozycjach od 0 włącznie do i wyłącznie.

Pseudo-kod: Przypisz $A_l\text{shift} = 0$ ($A_l\text{shift}$ wskazuje pierwszy indeks A_l w A). Dla kolejnych poziomów $l = 0, 1, \ldots$:

1. Przypisz $i = A_l\text{shift} + h_l(k)$.
2. Jeśli $A[i] = 1$ (równoważnie: $A_l[h_l(k)] = 1$):
   - zakończ i zwróć $\text{rank}(A, i)$.
3. Ustaw $A_l\text{shift} = A_l\text{shift} + |A_l|$.

BBHash – wyznaczanie wartości $h(k)$

- W A jest dokładnie $|K|$ jedynek,
- każda odpowiada dokładnie jednemu kluczowi z K.
- Dla $k \in K$, wyznaczanie wartości $h(k)$ zaczynamy od znalezienia indeksu i zawierającego jedynkę ($A[i] = 1$) odpowiadającą kluczowi k.
  Ściślej $i = \sum_{p=0}^{l-1} |A_p| + h_l(k)$, gdzie l to najniższy numeru poziomu, taki że $A_l[h_l(k)] = 1$.
- Zwracamy $\text{rank}(A, i)$, gdzie $\text{rank}(A, i) = \sum_{b=0}^{i-1} A[b]$ to liczba jedynek w A, na pozycjach od 0 włącznie do i wyłącznie.

Pseudo-kod: Przypisz $A_l\text{shift} = 0$ ($A_l\text{shift}$ wskazuje pierwszy indeks A_l w A). Dla kolejnych poziomów $l = 0, 1, \ldots$:

1. Przypisz $i = A_l\text{shift} + h_l(k)$.
2. Jeśli $A[i] = 1$ (równoważnie: $A_l[h_l(k)] = 1$):
   - zakończ i zwróć $\text{rank}(A, i)$.
3. Ustaw $A_l\text{shift} = A_l\text{shift} + |A_l|$.

BBHash – wyznaczanie wartości $h(k)$

- W A jest dokładnie $|K|$ jedynek,
- każda odpowiada dokładnie jednemu kluczowi z K.
- Dla $k \in K$, wyznaczanie wartości $h(k)$ zaczynamy od znalezienia indeksu i zawierającego jedynkę ($A[i] = 1$) odpowiadającą kluczowi k.
  Ściślej $i = \sum_{p=0}^{l-1} |A_p| + h_l(k)$, gdzie l to najniższy numeru poziomu, taki że $A_l[h_l(k)] = 1$.
- Zwracamy $\text{rank}(A, i)$, gdzie $\text{rank}(A, i) = \sum_{b=0}^{i-1} A[b]$ to liczba jedynek w A, na pozycjach od 0 włącznie do i wyłącznie.

Pseudo-kod: Przypisz $A_l\text{shift} = 0$ ($A_l\text{shift}$ wskazuje pierwszy indeks A_l w A). Dla kolejnych poziomów $l = 0, 1, \ldots$:

1. Przypisz $i = A_l\text{shift} + h_l(k)$.
2. Jeśli $A[i] = 1$ (równoważnie: $A_l[h_l(k)] = 1$):
   - zakończ i zwróć $\text{rank}(A, i)$.
3. Ustaw $A_l\text{shift} = A_l\text{shift} + |A_l|$.

BBHash – wyznaczanie wartości $h(k)$

- W A jest dokładnie $|K|$ jedynek,
- każda odpowiada dokładnie jednemu kluczowi z K.
- Dla $k \in K$, wyznaczanie wartości $h(k)$ zaczynamy od znalezienia indeksu i zawierającego jedynkę ($A[i] = 1$) odpowiadającą kluczowi k.
  Ściślej $i = \sum_{p=0}^{l-1} |A_p| + h_l(k)$, gdzie l to najniższy numeru poziomu, taki że $A_l[h_l(k)] = 1$.
- Zwracamy $\text{rank}(A, i)$, gdzie $\text{rank}(A, i) = \sum_{b=0}^{i-1} A[b]$ to liczba jedynek w A, na pozycjach od 0 włącznie do i wyłącznie.

Pseudo-kod: Przypisz $A_l\text{shift} = 0$ ($A_l\text{shift}$ wskazuje pierwszy indeks A_l w A). Dla kolejnych poziomów $l = 0, 1, \ldots$:

1. Przypisz $i = A_l\text{shift} + h_l(k)$.
2. Jeśli $A[i] = 1$ (równoważnie: $A_l[h_l(k)] = 1$):
   - zakończ i zwróć $\text{rank}(A, i)$.
3. Ustaw $A_l\text{shift} = A_l\text{shift} + |A_l|$.

BBHash – wyznaczanie wartości $h(k)$

- W A jest dokładnie $|K|$ jedynek,
- każda odpowiada dokładnie jednemu kluczowi z K.
- Dla $k \in K$, wyznaczanie wartości $h(k)$ zaczynamy od znalezienia indeksu i zawierającego jedynkę ($A[i] = 1$) odpowiadającą kluczowi k.
  Ściślej $i = \sum_{p=0}^{l-1} |A_p| + h_l(k)$, gdzie l to najniższy numeru poziomu, taki że $A_l[h_l(k)] = 1$.
- Zwracamy $\text{rank}(A, i)$, gdzie $\text{rank}(A, i) = \sum_{b=0}^{i-1} A[b]$ to liczba jedynek w A, na pozycjach od 0 włącznie do i wyłącznie.

Pseudo-kod: Przypisz $A_l\text{shift} = 0$ ($A_l\text{shift}$ wskazuje pierwszy indeks A_l w A). Dla kolejnych poziomów $l = 0, 1, \ldots$:

1. Przypisz $i = A_l\text{shift} + h_l(k)$.
2. Jeśli $A[i] = 1$ (równoważnie: $A_l[h_l(k)] = 1$):
   - zakończ i zwróć $\text{rank}(A, i)$.
3. Ustaw $A_l\text{shift} = A_l\text{shift} + |A_l|$.

BBHash – wyznaczanie wartości $h(k)$

- W A jest dokładnie $|K|$ jedynek,
- każda odpowiada dokładnie jednemu kluczowi z K.
- Dla $k \in K$, wyznaczanie wartości $h(k)$ zaczynamy od znalezienia indeksu i zawierającego jedynkę ($A[i] = 1$) odpowiadającą kluczowi k.
  Ściślej $i = \sum_{p=0}^{l-1} |A_p| + h_l(k)$, gdzie l to najniższy numeru poziomu, taki że $A_l[h_l(k)] = 1$.
- Zwracamy $\text{rank}(A, i)$, gdzie $\text{rank}(A, i) = \sum_{b=0}^{i-1} A[b]$ to liczba jedynek w A, na pozycjach od 0 włącznie do i wyłącznie.

Pseudo-kod: Przypisz $A_l\text{shift} = 0$ ($A_l\text{shift}$ wskazuje pierwszy indeks A_l w A). Dla kolejnych poziomów $l = 0, 1, \ldots$:

1. Przypisz $i = A_l\text{shift} + h_l(k)$.
2. Jeśli $A[i] = 1$ (równoważnie: $A_l[h_l(k)] = 1$):
   - zakończ i zwróć $\text{rank}(A, i)$.
3. Ustaw $A_l\text{shift} = A_l\text{shift} + |A_l|$.

BBHash – wyznaczanie wartości $h(k)$

- W A jest dokładnie $|K|$ jedynek,
- każda odpowiada dokładnie jednemu kluczowi z K.
- Dla $k \in K$, wyznaczanie wartości $h(k)$ zaczynamy od znalezienia indeksu i zawierającego jedynkę ($A[i] = 1$) odpowiadającą kluczowi k.
  Ściślej $i = \sum_{p=0}^{l-1} |A_p| + h_l(k)$, gdzie l to najniższy numeru poziomu, taki że $A_l[h_l(k)] = 1$.
- Zwracamy $\text{rank}(A, i)$, gdzie $\text{rank}(A, i) = \sum_{b=0}^{i-1} A[b]$ to liczba jedynek w A, na pozycjach od 0 włącznie do i wyłącznie.

Pseudo-kod: Przypisz $A_l\text{shift} = 0$ ($A_l\text{shift}$ wskazuje pierwszy indeks A_l w A). Dla kolejnych poziomów $l = 0, 1, \ldots$:

1. Przypisz $i = A_l\text{shift} + h_l(k)$.
2. Jeśli $A[i] = 1$ (równoważnie: $A_l[h_l(k)] = 1$):
   - zakończ i zwróć $\text{rank}(A, i)$.
3. Ustaw $A_l\text{shift} = A_l\text{shift} + |A_l|$.

BBHash – wyznaczanie wartości $h(k)$

- W A jest dokładnie $|K|$ jedynek,
- każda odpowiada dokładnie jednemu kluczowi z K.
- Dla $k \in K$, wyznaczanie wartości $h(k)$ zaczynamy od znalezienia indeksu i zawierającego jedynkę ($A[i] = 1$) odpowiadającą kluczowi k.
  Ściślej $i = \sum_{p=0}^{l-1} |A_p| + h_l(k)$, gdzie l to najniższy numeru poziomu, taki że $A_l[h_l(k)] = 1$.
- Zwracamy $\text{rank}(A, i)$, gdzie $\text{rank}(A, i) = \sum_{b=0}^{i-1} A[b]$ to liczba jedynek w A, na pozycjach od 0 włącznie do i wyłącznie.

Pseudo-kod: Przypisz $A_l\text{shift} = 0$ ($A_l\text{shift}$ wskazuje pierwszy indeks A_l w A). Dla kolejnych poziomów $l = 0, 1, \ldots$:

1. Przypisz $i = A_l\text{shift} + h_l(k)$.
2. Jeśli $A[i] = 1$ (równoważnie: $A_l[h_l(k)] = 1$):
   - zakończ i zwróć $\text{rank}(A, i)$.
3. Ustaw $A_l\text{shift} = A_l\text{shift} + |A_l|$.

BBHash – wyznaczanie wartości $h(k)$

- W A jest dokładnie $|K|$ jedynek,
- każda odpowiada dokładnie jednemu kluczowi z K.
- Dla $k \in K$, wyznaczanie wartości $h(k)$ zaczynamy od znalezienia indeksu i zawierającego jedynkę ($A[i] = 1$) odpowiadającą kluczowi k.
  Ściślej $i = \sum_{p=0}^{l-1} |A_p| + h_l(k)$, gdzie l to najniższy numeru poziomu, taki że $A_l[h_l(k)] = 1$.
- Zwracamy $\text{rank}(A, i)$, gdzie $\text{rank}(A, i) = \sum_{b=0}^{i-1} A[b]$ to liczba jedynek w A, na pozycjach od 0 włącznie do i wyłącznie.

Pseudo-kod: Przypisz $A_l\text{shift} = 0$ ($A_l\text{shift}$ wskazuje pierwszy indeks A_l w A). Dla kolejnych poziomów $l = 0, 1, \ldots$:

1. Przypisz $i = A_l\text{shift} + h_l(k)$.
2. Jeśli $A[i] = 1$ (równoważnie: $A_l[h_l(k)] = 1$):
   - zakończ i zwróć $\text{rank}(A, i)$.
3. Ustaw $A_l\text{shift} = A_l\text{shift} + |A_l|$.

BBHash – wyznaczanie wartości $h(k)$

- W A jest dokładnie $|K|$ jedynek,
- każda odpowiada dokładnie jednemu kluczowi z K.
- Dla $k \in K$, wyznaczanie wartości $h(k)$ zaczynamy od znalezienia indeksu i zawierającego jedynkę ($A[i] = 1$) odpowiadającą kluczowi k.
  Ściślej $i = \sum_{p=0}^{l-1} |A_p| + h_l(k)$, gdzie l to najniższy numeru poziomu, taki że $A_l[h_l(k)] = 1$.
- Zwracamy $\text{rank}(A, i)$, gdzie $\text{rank}(A, i) = \sum_{b=0}^{i-1} A[b]$ to liczba jedynek w A, na pozycjach od 0 włącznie do i wyłącznie.

Pseudo-kod: Przypisz $A_l\text{shift} = 0$ ($A_l\text{shift}$ wskazuje pierwszy indeks A_l w A). Dla kolejnych poziomów $l = 0, 1, \ldots$:

1. Przypisz $i = A_l\text{shift} + h_l(k)$.
2. Jeśli $A[i] = 1$ (równoważnie: $A_l[h_l(k)] = 1$):
   - zakończ i zwróć $\text{rank}(A, i)$.
3. Ustaw $A_l\text{shift} = A_l\text{shift} + |A_l|$.

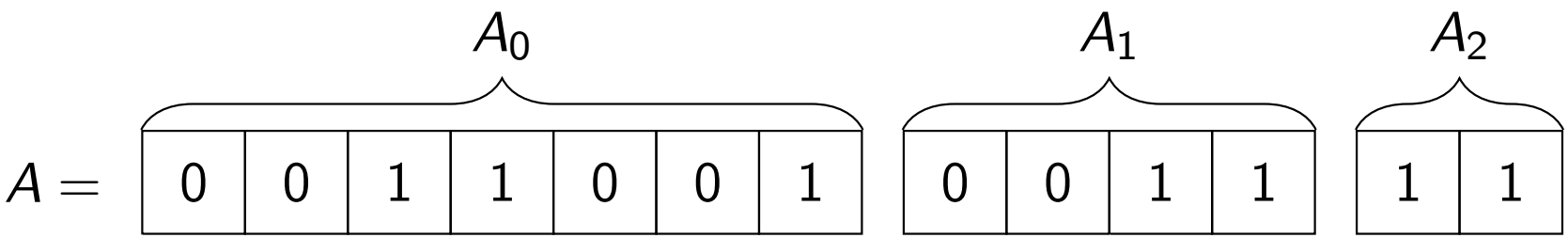
BBHash – wyznaczanie wartości $h(k)$

- W A jest dokładnie $|K|$ jedynek,
- każda odpowiada dokładnie jednemu kluczowi z K.
- Dla $k \in K$, wyznaczanie wartości $h(k)$ zaczynamy od znalezienia indeksu i zawierającego jedynkę ($A[i] = 1$) odpowiadającą kluczowi k.
  Ściślej $i = \sum_{p=0}^{l-1} |A_p| + h_l(k)$, gdzie l to najniższy numeru poziomu, taki że $A_l[h_l(k)] = 1$.
- Zwracamy $\text{rank}(A, i)$, gdzie $\text{rank}(A, i) = \sum_{b=0}^{i-1} A[b]$ to liczba jedynek w A, na pozycjach od 0 włącznie do i wyłącznie.

Pseudo-kod: Przypisz $A_l\text{shift} = 0$ ($A_l\text{shift}$ wskazuje pierwszy indeks A_l w A). Dla kolejnych poziomów $l = 0, 1, \ldots$:

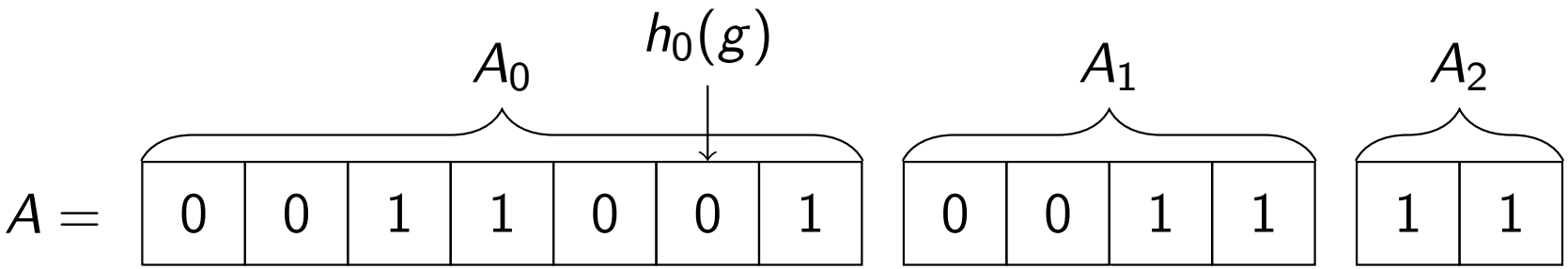
1. Przypisz $i = A_l\text{shift} + h_l(k)$.
2. Jeśli $A[i] = 1$ (równoważnie: $A_l[h_l(k)] = 1$):
   - zakończ i zwróć $\text{rank}(A, i)$.
3. Ustaw $A_l\text{shift} = A_l\text{shift} + |A_l|$.

BBHash – przykład wyznaczania wartości funkcji



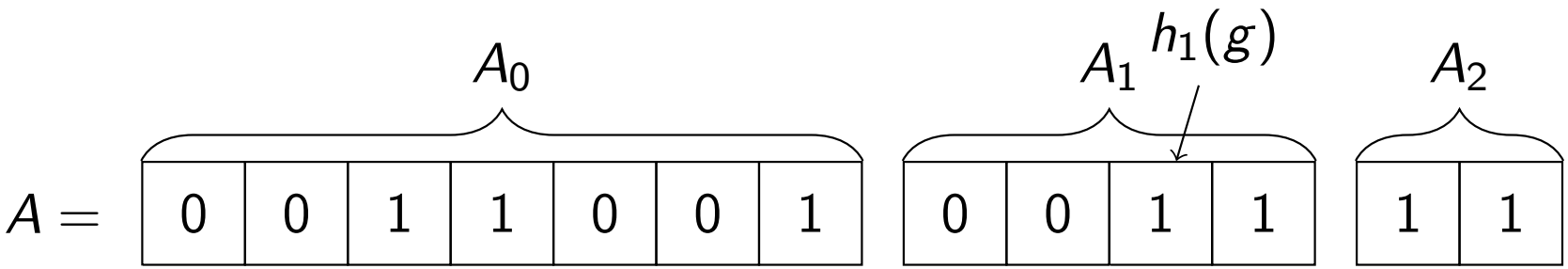
- Prześledźmy proces wyznaczania wartości $h(g)$.
- Sprawdzamy czy $A_0[h_0(g)] = 1$?
- Ponieważ tak nie jest, to sprawdzamy czy $A_1[h_1(g)] = 1$?
- To jest prawdą, więc algorytm kończy działanie i zwraca liczbę jedynek stojących w tablicy A (będącej konkatenacją $A_0, A_1, \dots$) na pozycjach poprzedzających wskazaną jedynkę.

BBHash – przykład wyznaczania wartości funkcji



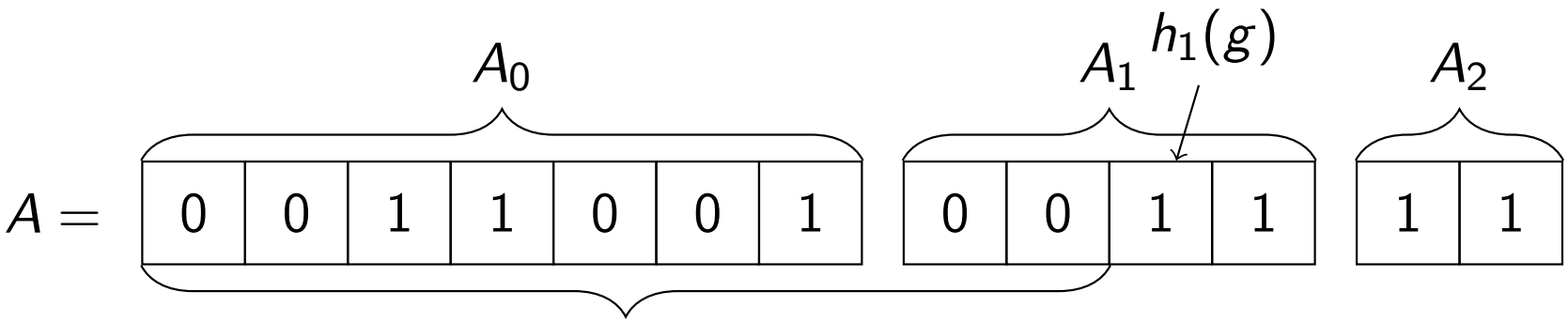
- Prześledźmy proces wyznaczania wartości $h(g)$.
- Sprawdzamy czy $A_0[h_0(g)] = 1$?
- Ponieważ tak nie jest, to sprawdzamy czy $A_1[h_1(g)] = 1$?
- To jest prawdą, więc algorytm kończy działanie i zwraca liczbę jedynek stojących w tablicy A (będącej konkatenacją $A_0, A_1, \dots$) na pozycjach poprzedzających wskazaną jedynkę.

BBHash – przykład wyznaczania wartości funkcji



- Prześledźmy proces wyznaczania wartości $h(g)$.
- Sprawdzamy czy $A_0[h_0(g)] = 1$?
- Ponieważ tak nie jest, to sprawdzamy czy $A_1[h_1(g)] = 1$?
- To jest prawdą, więc algorytm kończy działanie i zwraca liczbę jedynek stojących w tablicy A (będącej konkatenacją $A_0, A_1, \dots$) na pozycjach poprzedzających wskazaną jedynkę.

BBHash – przykład wyznaczania wartości funkcji



$h(g)$ = liczba jedynek w zaznaczonym zakresie = 3

- Prześledźmy proces wyznaczania wartości $h(g)$.
- Sprawdzamy czy $A_0[h_0(g)] = 1$?
- Ponieważ tak nie jest, to sprawdzamy czy $A_1[h_1(g)] = 1$?
- To jest prawdą, więc algorytm kończy działanie i zwraca liczbę jedynek stojących w tablicy A (będącej konkatenacją $A_0, A_1, \dots$) na pozycjach poprzedzających wskazaną jedynkę.

BBHash – optymalne rozmiary poziomów

- Oczekiwany rozmiar A jest najmniejszy gdy $|A_l| = |K_l|$ dla wszystkich poziomów l [1, 2].
- Zasadne jest też użycie większych tablic A_l gdyż wtedy:
  - rośnie liczba kluczy zakodowanych na kolejnych poziomach,
  - co prowadzi do zmniejszenia liczby poziomów
  - i ostatecznie do skrócenia czasu obliczania wartości funkcji h;
- Oczekiwana wielkość A to około $\gamma e^{1/\gamma}|K|$ bitów, jeśli $\gamma \geq 1$ i $|A_l| = \gamma|K_l|$ dla wszystkich l [2].
- Eksperymenty zawarte w [2] pokazują, że $\gamma = 2$ zapewnia dobry kompromis pomiędzy rozmiarem (całkowity rozmiar funkcji: 3,7 bita/klucz) i czasem dostępu.

---

[1] **J. A. Chapman, I. Ho, S. Sunkara, S. Luo, G. P. Schroth, D. S. Rokhsar**, *Meraculous: De novo genome assembly with short paired-end reads*, PLOS ONE 6 (8) 1–13, 08.2011

[2] **A. Limasset, G. Rizk, R. Chikhi, P. Peterlongo**, *Fast and Scalable Minimal Perfect Hashing for Massive Key Sets*, SEA 2017

BBHash – optymalne rozmiary poziomów

- Oczekiwany rozmiar A jest najmniejszy gdy $|A_l| = |K_l|$ dla wszystkich poziomów l [1, 2].
- Zasadne jest też użycie większych tablic A_l gdyż wtedy:
  - rośnie liczba kluczy zakodowanych na kolejnych poziomach,
  - co prowadzi do zmniejszenia liczby poziomów
  - i ostatecznie do skrócenia czasu obliczania wartości funkcji h;
- Oczekiwana wielkość A to około $\gamma e^{1/\gamma}|K|$ bitów, jeśli $\gamma \geq 1$ i $|A_l| = \gamma|K_l|$ dla wszystkich l [2].
- Eksperymenty zawarte w [2] pokazują, że $\gamma = 2$ zapewnia dobry kompromis pomiędzy rozmiarem (całkowity rozmiar funkcji: 3,7 bita/klucz) i czasem dostępu.

---

[1] **J. A. Chapman, I. Ho, S. Sunkara, S. Luo, G. P. Schroth, D. S. Rokhsar**, *Meraculous: De novo genome assembly with short paired-end reads*, PLOS ONE 6 (8) 1–13, 08.2011

[2] **A. Limasset, G. Rizk, R. Chikhi, P. Peterlongo**, *Fast and Scalable Minimal Perfect Hashing for Massive Key Sets*, SEA 2017

BBHash – optymalne rozmiary poziomów

- Oczekiwany rozmiar A jest najmniejszy gdy $|A_l| = |K_l|$ dla wszystkich poziomów l [1, 2].
- Zasadne jest też użycie większych tablic A_l gdyż wtedy:
  - rośnie liczba kluczy zakodowanych na kolejnych poziomach,
  - co prowadzi do zmniejszenia liczby poziomów
  - i ostatecznie do skrócenia czasu obliczania wartości funkcji h;
- Oczekiwana wielkość A to około $\gamma e^{1/\gamma}|K|$ bitów, jeśli $\gamma \geq 1$ i $|A_l| = \gamma|K_l|$ dla wszystkich l [2].
- Eksperymenty zawarte w [2] pokazują, że $\gamma = 2$ zapewnia dobry kompromis pomiędzy rozmiarem (całkowity rozmiar funkcji: 3,7 bita/klucz) i czasem dostępu.

---

[1] **J. A. Chapman, I. Ho, S. Sunkara, S. Luo, G. P. Schroth, D. S. Rokhsar**, *Meraculous: De novo genome assembly with short paired-end reads*, PLOS ONE 6 (8) 1–13, 08.2011

[2] **A. Limasset, G. Rizk, R. Chikhi, P. Peterlongo**, *Fast and Scalable Minimal Perfect Hashing for Massive Key Sets*, SEA 2017

BBHash – optymalne rozmiary poziomów

- Oczekiwany rozmiar A jest najmniejszy gdy $|A_l| = |K_l|$ dla wszystkich poziomów l [1, 2].
- Zasadne jest też użycie większych tablic A_l gdyż wtedy:
  - rośnie liczba kluczy zakodowanych na kolejnych poziomach,
  - co prowadzi do zmniejszenia liczby poziomów
  - i ostatecznie do skrócenia czasu obliczania wartości funkcji h;
- Oczekiwana wielkość A to około $\gamma e^{1/\gamma}|K|$ bitów, jeśli $\gamma \geq 1$ i $|A_l| = \gamma|K_l|$ dla wszystkich l [2].
- Eksperymenty zawarte w [2] pokazują, że $\gamma = 2$ zapewnia dobry kompromis pomiędzy rozmiarem (całkowity rozmiar funkcji: 3,7 bita/klucz) i czasem dostępu.

---

[1] **J. A. Chapman, I. Ho, S. Sunkara, S. Luo, G. P. Schroth, D. S. Rokhsar**, *Meraculous: De novo genome assembly with short paired-end reads*, PLOS ONE 6 (8) 1–13, 08.2011

[2] **A. Limasset, G. Rizk, R. Chikhi, P. Peterlongo**, *Fast and Scalable Minimal Perfect Hashing for Massive Key Sets*, SEA 2017

BBHash – optymalne rozmiary poziomów

- Oczekiwany rozmiar A jest najmniejszy gdy $|A_l| = |K_l|$ dla wszystkich poziomów l [1, 2].
- Zasadne jest też użycie większych tablic A_l gdyż wtedy:
  - rośnie liczba kluczy zakodowanych na kolejnych poziomach,
  - co prowadzi do zmniejszenia liczby poziomów
  - i ostatecznie do skrócenia czasu obliczania wartości funkcji h;
- Oczekiwana wielkość A to około $\gamma e^{1/\gamma}|K|$ bitów, jeśli $\gamma \geq 1$ i $|A_l| = \gamma|K_l|$ dla wszystkich l [2].
- Eksperymenty zawarte w [2] pokazują, że $\gamma = 2$ zapewnia dobry kompromis pomiędzy rozmiarem (całkowity rozmiar funkcji: 3,7 bita/klucz) i czasem dostępu.

---

[1] **J. A. Chapman, I. Ho, S. Sunkara, S. Luo, G. P. Schroth, D. S. Rokhsar**, *Meraculous: De novo genome assembly with short paired-end reads*, PLOS ONE 6 (8) 1–13, 08.2011

[2] **A. Limasset, G. Rizk, R. Chikhi, P. Peterlongo**, *Fast and Scalable Minimal Perfect Hashing for Massive Key Sets*, SEA 2017

BBHash – optymalne rozmiary poziomów

- Oczekiwany rozmiar A jest najmniejszy gdy $|A_l| = |K_l|$ dla wszystkich poziomów l [1, 2].
- Zasadne jest też użycie większych tablic A_l gdyż wtedy:
  - rośnie liczba kluczy zakodowanych na kolejnych poziomach,
  - co prowadzi do zmniejszenia liczby poziomów
  - i ostatecznie do skrócenia czasu obliczania wartości funkcji h;
- Oczekiwana wielkość A to około $\gamma e^{1/\gamma}|K|$ bitów, jeśli $\gamma \geq 1$ i $|A_l| = \gamma|K_l|$ dla wszystkich l [2].
- Eksperymenty zawarte w [2] pokazują, że $\gamma = 2$ zapewnia dobry kompromis pomiędzy rozmiarem (całkowity rozmiar funkcji: 3,7 bita/klucz) i czasem dostępu.

---

[1] **J. A. Chapman, I. Ho, S. Sunkara, S. Luo, G. P. Schroth, D. S. Rokhsar**, *Meraculous: De novo genome assembly with short paired-end reads*, PLOS ONE 6 (8) 1–13, 08.2011

[2] **A. Limasset, G. Rizk, R. Chikhi, P. Peterlongo**, *Fast and Scalable Minimal Perfect Hashing for Massive Key Sets*, SEA 2017

BBHash – optymalne rozmiary poziomów

- Oczekiwany rozmiar A jest najmniejszy gdy $|A_l| = |K_l|$ dla wszystkich poziomów l [1, 2].
- Zasadne jest też użycie większych tablic A_l gdyż wtedy:
  - rośnie liczba kluczy zakodowanych na kolejnych poziomach,
  - co prowadzi do zmniejszenia liczby poziomów
  - i ostatecznie do skrócenia czasu obliczania wartości funkcji h;
- Oczekiwana wielkość A to około $\gamma e^{1/\gamma}|K|$ bitów, jeśli $\gamma \geq 1$ i $|A_l| = \gamma|K_l|$ dla wszystkich l [2].
- Eksperymenty zawarte w [2] pokazują, że $\gamma = 2$ zapewnia dobry kompromis pomiędzy rozmiarem (całkowity rozmiar funkcji: 3,7 bita/klucz) i czasem dostępu.

---

[1] **J. A. Chapman, I. Ho, S. Sunkara, S. Luo, G. P. Schroth, D. S. Rokhsar**, *Meraculous: De novo genome assembly with short paired-end reads*, PLOS ONE 6 (8) 1–13, 08.2011

[2] **A. Limasset, G. Rizk, R. Chikhi, P. Peterlongo**, *Fast and Scalable Minimal Perfect Hashing for Massive Key Sets*, SEA 2017

Bibliografia – BBHash

- **A. Limasset, G. Rizk, R. Chikhi, P. Peterlongo**, *Fast and Scalable Minimal Perfect Hashing for Massive Key Sets*, SEA 2017
  - ta publikacja zawiera pseudo-kody algorytmu BBHash
  - https://github.com/rizkg/BBHash (implementacja w C++)
- **J. A. Chapman, I. Ho, S. Sunkara, S. Luo, G. P. Schroth, D. S. Rokhsar**, *Meraculous: De novo genome assembly with short paired-end reads*, PLOS ONE 6 (8) 1–13, 08.2011
- **I. Müller, P. Sanders, R. Schulze, W. Zhou**, *Retrieval and perfect hashing using fingerprinting*, Experimental Algorithms, 2014

Bibliografia – pozostałe

- **E. Esposito, T. M. Graf, S. Vigna**, *RecSplit: Minimal Perfect Hashing via Recursive Splitting*, 2019
- **D. Belazzougui, F. C. Botelho, M. Dietzfelbinger**, *Hash, displace, and compress*, Algorithms - ESA 2009
- **M. L. Fredman, J. Komlós**, *On the size of separating systems and families of perfect hash functions*, SIAM Journal on Algebraic Discrete Methods 5 (1), 1984
- **J. Radhakrishnan**, *Improved bounds for covering complete uniform hypergraphs*, Information Processing Letters 41 (4), 1992
- **Dong Zhou, David G. Andersen, Michael Kaminsky** *Space-Efficient, High-Performance Rank and Select Structures on Uncompressed Bit Sequences*, Experimental Algorithms, 2013