PHast

Perfect Hashing made fast

Piotr Beling i Peter Sanders

2026

Na podstawie artykułu
Piotr Beling, Peter Sanders *PHast - Perfect Hashing made fast* 2026 Proceedings of the SIAM Symposium on Algorithm Engineering and Experiments (ALENEX),
którego pełna, najnowsza wersja dostępna jest na https://arxiv.org/abs/2504.17918,
oraz slajdów pokazanych na konferencji ALENEX 2026.

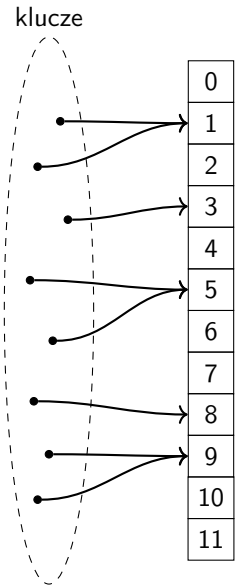
Perfect hashing

Dla zadanego zbioru kluczy $\{k_1, \ldots, k_n\}$, funkcja haszująca (*Hash Function*) $f : \{k_1, \ldots, k_n\} \to \{0, \ldots, m-1\}$ jest

- *Doskonała* (*Perfect; PHF*)
  jeśli jest różnowartościowa,
- *Minimalna Doskonała* (*Minimal Perfect; MPHF*)
  jesli, dodatkowo, $m = n$.

Pożądane cechy MPHF:

- **mały rozmiar**
  (teoretyczne minimum to $\log_2 e \approx 1.44$ bitów na klucz, niezależnie od typów kluczy),
- **maksymalna szybkość**
  konstruowania i obliczania wartości.



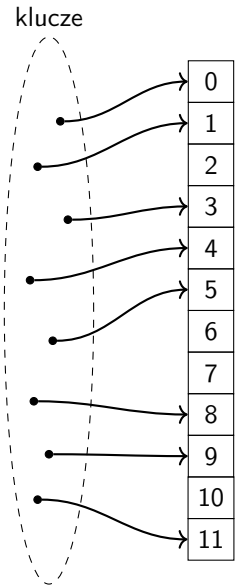
Perfect hashing

Dla zadanego zbioru kluczy $\{k_1, \ldots, k_n\}$, funkcja haszująca (*Hash Function*) $f : \{k_1, \ldots, k_n\} \to \{0, \ldots, m-1\}$ jest

- *Doskonała* (*Perfect; PHF*)
  jeśli jest różnowartościowa,
- *Minimalna Doskonała* (*Minimal Perfect; MPHF*)
  jesli, dodatkowo, $m = n$.

Pożądane cechy MPHF:

- **mały rozmiar**
  (teoretyczne minimum to $\log_2 e \approx 1.44$ bitów na klucz, niezależnie od typów kluczy),
- **maksymalna szybkość**
  konstruowania i obliczania wartości.



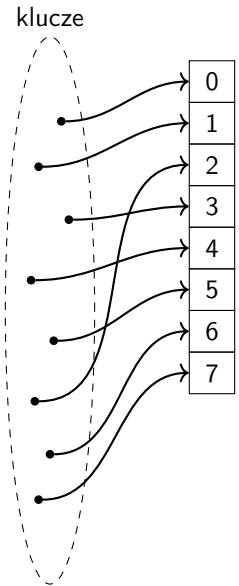
Perfect hashing

Dla zadanego zbioru kluczy $\{k_1, \ldots, k_n\}$, funkcja haszująca (*Hash Function*) $f : \{k_1, \ldots, k_n\} \to \{0, \ldots, m-1\}$ jest

- *Doskonała* (*Perfect; PHF*)
  jeśli jest różnowartościowa,
- *Minimalna Doskonała* (*Minimal Perfect; MPHF*)
  jesli, dodatkowo, $m = n$.

Pożądane cechy MPHF:

- **mały rozmiar**
  (teoretyczne minimum to $\log_2 e \approx 1.44$ bitów na klucz, niezależnie od typów kluczy),
- **maksymalna szybkość**
  konstruowania i obliczania wartości.



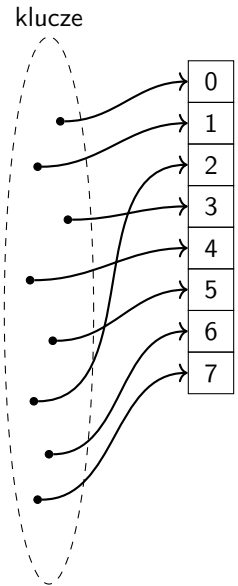
Perfect hashing

Dla zadanego zbioru kluczy $\{k_1, \ldots, k_n\}$, funkcja haszująca (*Hash Function*) $f : \{k_1, \ldots, k_n\} \to \{0, \ldots, m-1\}$ jest

- *Doskonała* (*Perfect; PHF*)
  jeśli jest różnowartościowa,
- *Minimalna Doskonała* (*Minimal Perfect; MPHF*)
  jesli, dodatkowo, $m = n$.

Pożądane cechy MPHF:

- **mały rozmiar**
  (teoretyczne minimum to $\log_2 e \approx 1.44$ bitów na klucz, niezależnie od typów kluczy),
- **maksymalna szybkość**
  konstruowania i obliczania wartości.



Zastosowania doskonałych funkcji haszujących

(Minimalne) Doskonałe Funkcje Haszujące są używane w:

- bazach danych,
- indeksowaniu tekstu,
- statycznych tablicach haszujących,
- klasyfikacji pakietów w routerach i firewallach,
- bioinformatyce.

- **Szybka ewaluacja jest kluczowa** w wielu zastosowaniach, i dlatego była głównym celem projektowym PHast (Perfect Hashing made fast).
- Dlatego PHast bazuje na technice zwanej *umieszczaniem wiader* (*bucket placement*), wykorzystywanej przez najszybsze, znane doskonałe funkcje haszujące.

Zastosowania doskonałych funkcji haszujących

(Minimalne) Doskonałe Funkcje Haszujące są używane w:

- bazach danych,
- indeksowaniu tekstu,
- statycznych tablicach haszujących,
- klasyfikacji pakietów w routerach i firewallach,
- bioinformatyce.

- **Szybka ewaluacja jest kluczowa** w wielu zastosowaniach, i dlatego była głównym celem projektowym PHast (Perfect Hashing made fast).
- Dlatego PHast bazuje na technice zwanej *umieszczaniem wiader* (*bucket placement*), wykorzystywanej przez najszybsze, znane doskonałe funkcje haszujące.

Zastosowania doskonałych funkcji haszujących

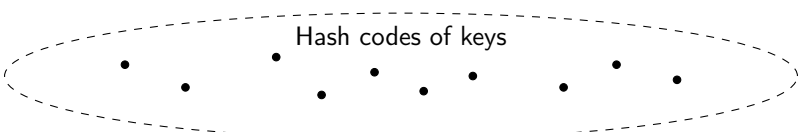
(Minimalne) Doskonałe Funkcje Haszujące są używane w:

- bazach danych,
- indeksowaniu tekstu,
- statycznych tablicach haszujących,
- klasyfikacji pakietów w routerach i firewallach,
- bioinformatyce.

- **Szybka ewaluacja jest kluczowa** w wielu zastosowaniach, i dlatego była głównym celem projektowym PHast (Perfect Hashing made fast).
- Dlatego PHast bazuje na technice zwanej *umieszczaniem wiader* (*bucket placement*), wykorzystywanej przez najszybsze, znane doskonałe funkcje haszujące.

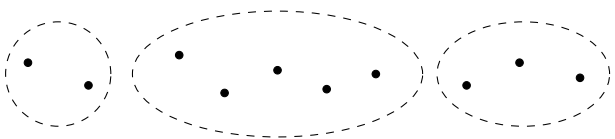
Konstruowanie typowego PHF bezującego na umieszczaniu kubełków

1. przypisanie wartości haszującej $c = h(k)$ do każdego klucza k
2. pseudolosowy podział zbioru wartości haszujących na kubełki
3. dla każdego kubełka, w kolejności od największego do najmniejszego: znalezienie najmniejszego zalążka s dla którego funkcja $p(s, c)$ bezkolizyjnie przypisuje wartości do wszystkich c w kubełku
4. zapis znalezionych zalążków w (opcjonalnie skompresowanej) tablicy *seeds*



Konstruowanie typowego PHF bezującego na umieszczaniu kubełków

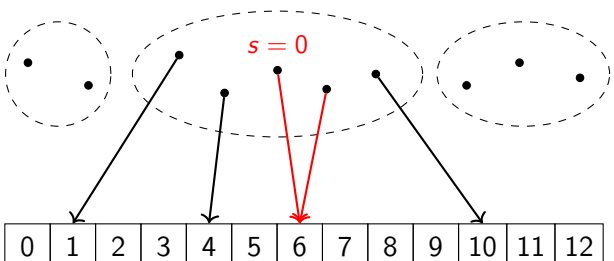
1. przypisanie wartości haszującej $c = h(k)$ do każdego klucza k
2. pseudolosowy podział zbioru wartości haszujących na kubełki
3. dla każdego kubełka, w kolejności od największego do najmniejszego: znalezienie najmniejszego zalążka s dla którego funkcja $p(s, c)$ bezkolizyjnie przypisuje wartości do wszystkich c w kubełku
4. zapis znalezionych zalążków w (opcjonalnie skompresowanej) tablicy *seeds*



0	1	2	3	4	5	6	7	8	9	10	11	12

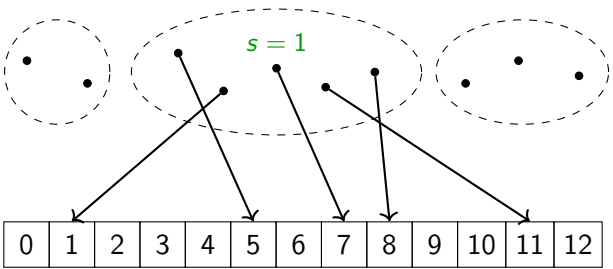
Konstruowanie typowego PHF bezującego na umieszczaniu kubełków

1. przypisanie wartości haszującej $c = h(k)$ do każdego klucza k
2. pseudolosowy podział zbioru wartości haszujących na kubełki
3. dla każdego kubełka, w kolejności od największego do najmniejszego: znalezienie najmniejszego zalążka s dla którego funkcja $p(s, c)$ bezkolizyjnie przypisuje wartości do wszystkich c w kubełku
4. zapis znalezionych zalążków w (opcjonalnie skompresowanej) tablicy *seeds*



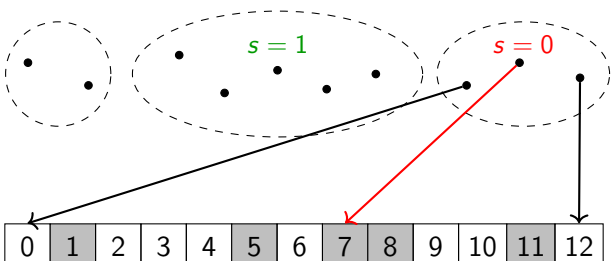
Konstruowanie typowego PHF bezującego na umieszczaniu kubełków

1. przypisanie wartości haszującej $c = h(k)$ do każdego klucza k
2. pseudolosowy podział zbioru wartości haszujących na kubełki
3. dla każdego kubełka, w kolejności od największego do najmniejszego: znalezienie najmniejszego zalążka s dla którego funkcja $p(s, c)$ bezkolizyjnie przypisuje wartości do wszystkich c w kubełku
4. zapis znalezionych zalążków w (opcjonalnie skompresowanej) tablicy *seeds*



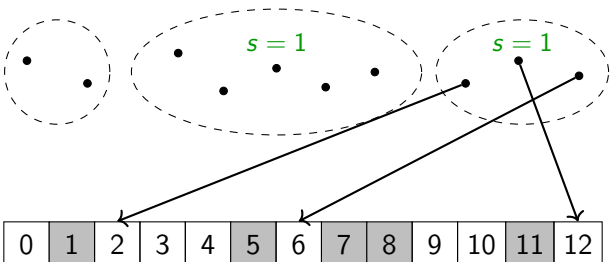
Konstruowanie typowego PHF bezującego na umieszczaniu kubełków

1. przypisanie wartości haszującej $c = h(k)$ do każdego klucza k
2. pseudolosowy podział zbioru wartości haszujących na kubełki
3. dla każdego kubełka, w kolejności od największego do najmniejszego: znalezienie najmniejszego zalążka s dla którego funkcja $p(s, c)$ bezkolizyjnie przypisuje wartości do wszystkich c w kubełku
4. zapis znalezionych zalążków w (opcjonalnie skompresowanej) tablicy *seeds*



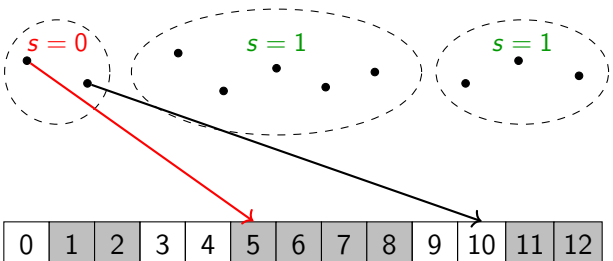
Konstruowanie typowego PHF bezującego na umieszczaniu kubełków

1. przypisanie wartości haszującej $c = h(k)$ do każdego klucza k
2. pseudolosowy podział zbioru wartości haszujących na kubełki
3. dla każdego kubełka, w kolejności od największego do najmniejszego: znalezienie najmniejszego zalążka s dla którego funkcja $p(s, c)$ bezkolizyjnie przypisuje wartości do wszystkich c w kubełku
4. zapis znalezionych zalążków w (opcjonalnie skompresowanej) tablicy *seeds*



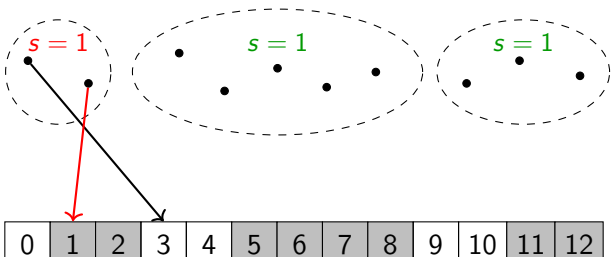
Konstruowanie typowego PHF bezującego na umieszczaniu kubełków

1. przypisanie wartości haszującej $c = h(k)$ do każdego klucza k
2. pseudolosowy podział zbioru wartości haszujących na kubełki
3. dla każdego kubełka, w kolejności od największego do najmniejszego: znalezienie najmniejszego zalążka s dla którego funkcja $p(s, c)$ bezkolizyjnie przypisuje wartości do wszystkich c w kubełku
4. zapis znalezionych zalążków w (opcjonalnie skompresowanej) tablicy *seeds*



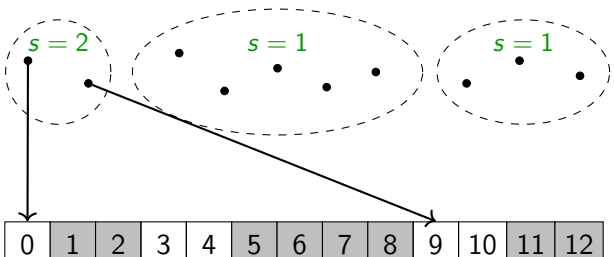
Konstruowanie typowego PHF bezującego na umieszczaniu kubełków

1. przypisanie wartości haszującej $c = h(k)$ do każdego klucza k
2. pseudolosowy podział zbioru wartości haszujących na kubełki
3. dla każdego kubełka, w kolejności od największego do najmniejszego: znalezienie najmniejszego zalążka s dla którego funkcja $p(s, c)$ bezkolizyjnie przypisuje wartości do wszystkich c w kubełku
4. zapis znalezionych zalążków w (opcjonalnie skompresowanej) tablicy *seeds*



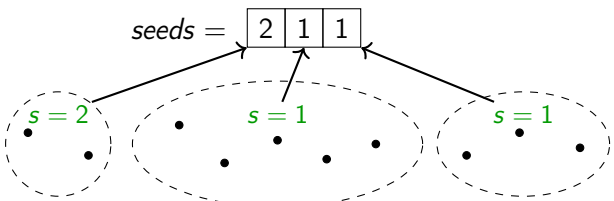
Konstruowanie typowego PHF bezującego na umieszczaniu kubełków

1. przypisanie wartości haszującej $c = h(k)$ do każdego klucza k
2. pseudolosowy podział zbioru wartości haszujących na kubełki
3. dla każdego kubełka, w kolejności od największego do najmniejszego: znalezienie najmniejszego zalążka s dla którego funkcja $p(s, c)$ bezkolizyjnie przypisuje wartości do wszystkich c w kubełku
4. zapis znalezionych zalążków w (opcjonalnie skompresowanej) tablicy *seeds*



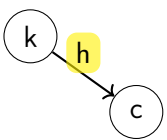
Konstruowanie typowego PHF bezującego na umieszczaniu kubełków

1. przypisanie wartości haszującej $c = h(k)$ do każdego klucza k
2. pseudolosowy podział zbioru wartości haszujących na kubełki
3. dla każdego kubełka, w kolejności od największego do najmniejszego: znalezienie najmniejszego zalążka s dla którego funkcja $p(s, c)$ bezkolizyjnie przypisuje wartości do wszystkich c w kubełku
4. zapis znalezionych zalążków w (opcjonalnie skompresowanej) tablicy *seeds*



0	1	2	3	4	5	6	7	8	9	10	11	12

Ewaluacja typowgo PHF bazującego na umieszczaniu kubełków



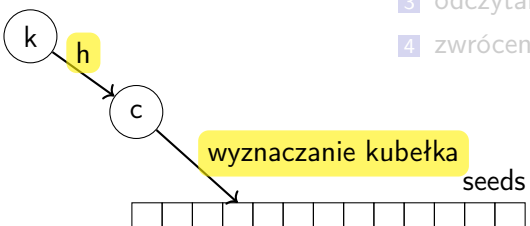
Obliczanie (ewaluacja) $f(k)$:

1. obliczenie wartości haszującej $c = h(k)$;
2. wyznaczenie indeksu i kubełka zawierającego c;
3. odczytanie zalążka $s = seeds[i]$ i-tego kubełka;
4. zwrócenie $p(s, c)$.

Ewaluacja typowgo PHF bazującego na umieszczaniu kubełków

Obliczanie (ewaluacja) $f(k)$:

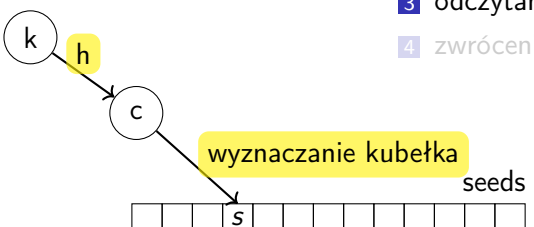
1. obliczenie wartości haszującej $c = h(k)$;
2. wyznaczenie indeksu i kubełka zawierającego c;
3. odczytanie zalążka $s = seeds[i]$ i-tego kubełka;
4. zwrócenie $p(s, c)$.



Ewaluacja typowgo PHF bazującego na umieszczaniu kubełków

Obliczanie (ewaluacja) $f(k)$:

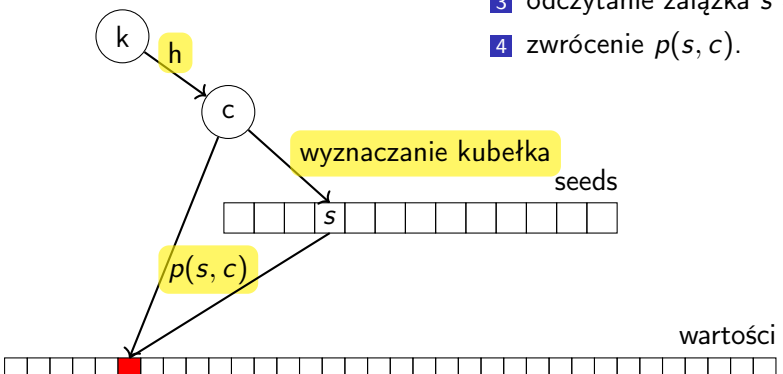
1. obliczenie wartości haszującej $c = h(k)$;
2. wyznaczenie indeksu i kubełka zawierającego c;
3. odczytanie zalążka $s = seeds[i]$ i-tego kubełka;
4. zwrócenie $p(s, c)$.



Ewaluacja typowgo PHF bazującego na umieszczaniu kubełków

Obliczanie (ewaluacja) $f(k)$:

1. obliczenie wartości haszującej $c = h(k)$;
2. wyznaczenie indeksu i kubełka zawierającego c;
3. odczytanie zalążka $s = seeds[i]$ i-tego kubełka;
4. zwrócenie $p(s, c)$.



Aspekty umieszczania kubełków poprawione w PHast

Zalążki dla ostatnich kubełków są typowo znacznie wyższe bo trudniej je znaleźć.

- **Kodowanie o zmiennej długości** jest niezbędne do uzyskania małego rozmiaru.
- By wyrównać trudność znalezienia zalążków, PHFy takie jak PHOBIC i PTHash mocniej różnicują wielkości kubełków używając **nieliniowego przypisywania kubełków** do wartości haszujących.

W razie kolizji wartości haszujących ($h(k_1) = h(k_2)$ dla $k_1 \neq k_2$), konieczne jest **ponowne haszowanie** kluczy za pomocą innego h.

- Ze względu na paradoks dnia urodzin, h **musi mieć duży zbiór wartości**.

Konstruowanie wielowątkowe wymaga **podziału zbioru wejściowego na partycje** (po jednej na wątek), co wiąże się z koniecznością odczytu i dodania przesunięcia partycji podczas ewaluacji.

Wszystko to spowalnia ewaluację.

Aspekty umieszczania kubełków poprawione w PHast

Zalążki dla ostatnich kubełków są typowo znacznie wyższe bo trudniej je znaleźć.

- **Kodowanie o zmiennej długości** jest niezbędne do uzyskania małego rozmiaru.
- By wyrównać trudność znalezienia zalążków, PHFy takie jak PHOBIC i PTHash mocniej różnicują wielkości kubełków używając **nieliniowego przypisywania kubełków** do wartości haszujących.

W razie kolizji wartości haszujących ($h(k_1) = h(k_2)$ dla $k_1 \neq k_2$), konieczne jest **ponowne haszowanie** kluczy za pomocą innego h.

- Ze względu na paradoks dnia urodzin, h **musi mieć duży zbiór wartości**.

Konstruowanie wielowątkowe wymaga **podziału zbioru wejściowego na partycje** (po jednej na wątek), co wiąże się z koniecznością odczytu i dodania przesunięcia partycji podczas ewaluacji.

Wszystko to spowalnia ewaluację.

Aspekty umieszczania kubełków poprawione w PHast

Zalążki dla ostatnich kubełków są typowo znacznie wyższe bo trudniej je znaleźć.

- **Kodowanie o zmiennej długości** jest niezbędne do uzyskania małego rozmiaru.
- By wyrównać trudność znalezienia zalążków, PHFy takie jak PHOBIC i PTHash mocniej różnicują wielkości kubełków używając **nieliniowego przypisywania kubełków** do wartości haszujących.

W razie kolizji wartości haszujących ($h(k_1) = h(k_2)$ dla $k_1 \neq k_2$), konieczne jest **ponowne haszowanie** kluczy za pomocą innego h.

- Ze względu na paradoks dnia urodzin, h **musi mieć duży zbiór wartości**.

Konstruowanie wielowątkowe wymaga **podziału zbioru wejściowego na partycje** (po jednej na wątek), co wiąże się z koniecznością odczytu i dodania przesunięcia partycji podczas ewaluacji.

Wszystko to spowalnia ewaluację.

Aspekty umieszczania kubełków poprawione w PHast

Zalążki dla ostatnich kubełków są typowo znacznie wyższe bo trudniej je znaleźć.

- **Kodowanie o zmiennej długości** jest niezbędne do uzyskania małego rozmiaru.
- By wyrównać trudność znalezienia zalążków, PHFy takie jak PHOBIC i PTHash mocniej różnicują wielkości kubełków używając **nieliniowego przypisywania kubełków** do wartości haszujących.

W razie kolizji wartości haszujących ($h(k_1) = h(k_2)$ dla $k_1 \neq k_2$), konieczne jest **ponowne haszowanie** kluczy za pomocą innego h.

- Ze względu na paradoks dnia urodzin, h **musi mieć duży zbiór wartości**.

Konstruowanie wielowątkowe wymaga **podziału zbioru wejściowego na partycje** (po jednej na wątek), co wiąże się z koniecznością odczytu i dodania przesunięcia partycji podczas ewaluacji.

Wszystko to spowalnia ewaluację.

Aspekty umieszczania kubełków poprawione w PHast

Zalążki dla ostatnich kubełków są typowo znacznie wyższe bo trudniej je znaleźć.

- **Kodowanie o zmiennej długości** jest niezbędne do uzyskania małego rozmiaru.
- By wyrównać trudność znalezienia zalążków, PHFy takie jak PHOBIC i PTHash mocniej różnicują wielkości kubełków używając **nieliniowego przypisywania kubełków** do wartości haszujących.

W razie kolizji wartości haszujących ($h(k_1) = h(k_2)$ dla $k_1 \neq k_2$), konieczne jest **ponowne haszowanie** kluczy za pomocą innego h.

- Ze względu na paradoks dnia urodzin, h **musi mieć duży zbiór wartości**.

Konstruowanie wielowątkowe wymaga **podziału zbioru wejściowego na partycje** (po jednej na wątek), co wiąże się z koniecznością odczytu i dodania przesunięcia partycji podczas ewaluacji.

Wszystko to spowalnia ewaluację.

Aspekty umieszczania kubełków poprawione w PHast

Zalążki dla ostatnich kubełków są typowo znacznie wyższe bo trudniej je znaleźć.

- **Kodowanie o zmiennej długości** jest niezbędne do uzyskania małego rozmiaru.
- By wyrównać trudność znalezienia zalążków, PHFy takie jak PHOBIC i PTHash mocniej różnicują wielkości kubełków używając **nieliniowego przypisywania kubełków** do wartości haszujących.

W razie kolizji wartości haszujących ($h(k_1) = h(k_2)$ dla $k_1 \neq k_2$), konieczne jest **ponowne haszowanie** kluczy za pomocą innego h.

- Ze względu na paradoks dnia urodzin, h **musi mieć duży zbiór wartości**.

Konstruowanie wielowątkowe wymaga **podziału zbioru wejściowego na partycje** (po jednej na wątek), co wiąże się z koniecznością odczytu i dodania przesunięcia partycji podczas ewaluacji.

Wszystko to spowalnia ewaluację.

Aspekty umieszczania kubełków poprawione w PHast

Zalążki dla ostatnich kubełków są typowo znacznie wyższe bo trudniej je znaleźć.

- **Kodowanie o zmiennej długości** jest niezbędne do uzyskania małego rozmiaru.
- By wyrównać trudność znalezienia zalążków, PHFy takie jak PHOBIC i PTHash mocniej różnicują wielkości kubełków używając **nieliniowego przypisywania kubełków** do wartości haszujących.

W razie kolizji wartości haszujących ($h(k_1) = h(k_2)$ dla $k_1 \neq k_2$), konieczne jest **ponowne haszowanie** kluczy za pomocą innego h.

- Ze względu na paradoks dnia urodzin, h **musi mieć duży zbiór wartości**.

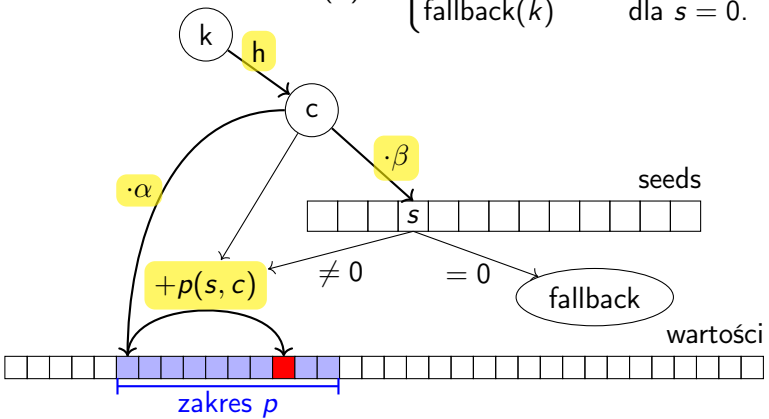
Konstruowanie wielowątkowe wymaga **podziału zbioru wejściowego na partycje** (po jednej na wątek), co wiąże się z koniecznością odczytu i dodania przesunięcia partycji podczas ewaluacji.

Wszystko to spowalnia ewaluację.

Obliczanie wartości przez PHast

Wartość PHast dla klucza k o haszu $c = h(k)$ i zalążka $s = seeds[i]$ (gdzie $i = \lfloor \beta c \rfloor$ jest indeksem kubełka) to

$$f(k) = \begin{cases} \lfloor \alpha c \rfloor + p(s, c) & \text{dla } s \neq 0, \\ \text{fallback}(k) & \text{dla } s = 0. \end{cases}$$

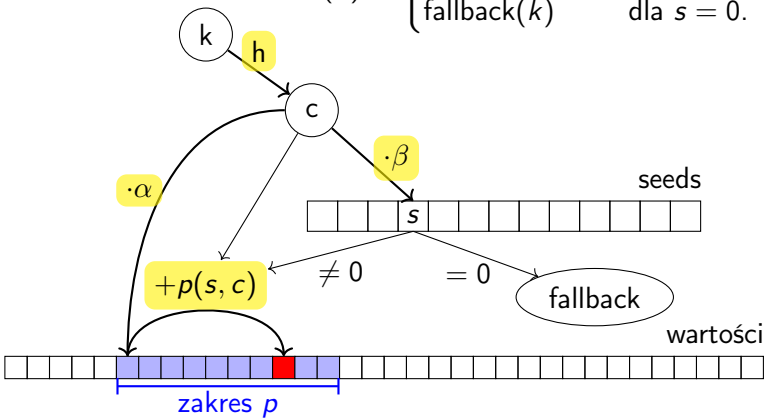


- $s = 0$ oznacza **wybicie** kubełka (dla którego nie znaleziono zalążka)
- dzięki wybijaniu mamy:
  - stały rozmiar zalążków
  - h o małym zakresie
  - proste (i szybkie) p
- dzięki małemu zakresowi p, i-ty kubełek pokrywa ograniczony zakres, przesuwany ku większym wartością gdy wzrasta i
- to czyni konstruowanie przyjaznym dla CPU-cache i wielowątkowości

Obliczanie wartości przez PHast

Wartość PHast dla klucza k o haszu $c = h(k)$ i zalążka $s = seeds[i]$ (gdzie $i = \lfloor \beta c \rfloor$ jest indeksem kubełka) to

$$f(k) = \begin{cases} \lfloor \alpha c \rfloor + p(s, c) & \text{dla } s \neq 0, \\ \text{fallback}(k) & \text{dla } s = 0. \end{cases}$$

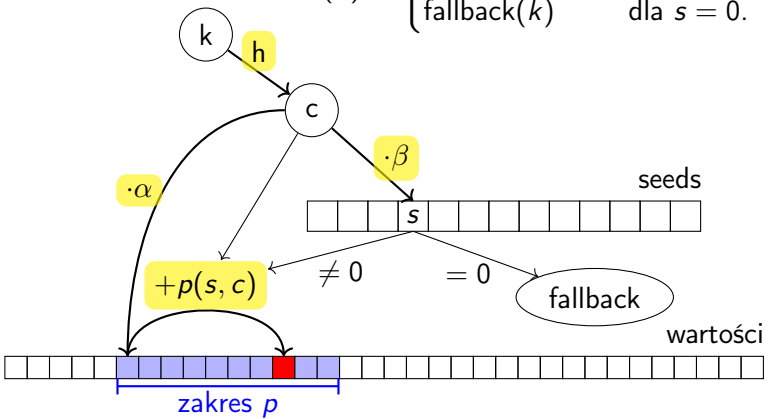


- $s = 0$ oznacza **wybicie** kubełka (dla którego nie znaleziono zalążka)
- dzięki wybijaniu mamy:
  - stały rozmiar zalążków
  - h o małym zakresie
  - proste (i szybkie) p
- dzięki małemu zakresowi p, i-ty kubełek pokrywa ograniczony zakres, przesuwany ku większym wartością gdy wzrasta i
- to czyni konstruowanie przyjaznym dla CPU-cache i wielowątkowości

Obliczanie wartości przez PHast

Wartość PHast dla klucza k o haszu $c = h(k)$ i zalążka $s = seeds[i]$ (gdzie $i = \lfloor \beta c \rfloor$ jest indeksem kubełka) to

$$f(k) = \begin{cases} \lfloor \alpha c \rfloor + p(s, c) & \text{dla } s \neq 0, \\ \text{fallback}(k) & \text{dla } s = 0. \end{cases}$$

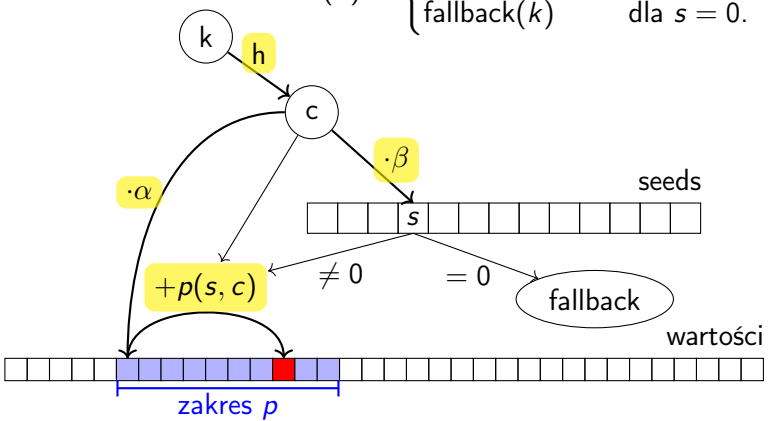


- $s = 0$ oznacza **wybicie** kubełka (dla którego nie znaleziono zalążka)
- dzięki wybijaniu mamy:
  - stały rozmiar zalążków
  - h o małym zakresie
  - proste (i szybkie) p
- dzięki małemu zakresowi p, i-ty kubełek pokrywa ograniczony zakres, przesuwany ku większym wartością gdy wzrasta i
- to czyni konstruowanie przyjaznym dla CPU-cache i wielowątkowości

Obliczanie wartości przez PHast

Wartość PHast dla klucza k o haszu $c = h(k)$ i zalążka $s = seeds[i]$ (gdzie $i = \lfloor \beta c \rfloor$ jest indeksem kubełka) to

$$f(k) = \begin{cases} \lfloor \alpha c \rfloor + p(s, c) & \text{dla } s \neq 0, \\ \text{fallback}(k) & \text{dla } s = 0. \end{cases}$$

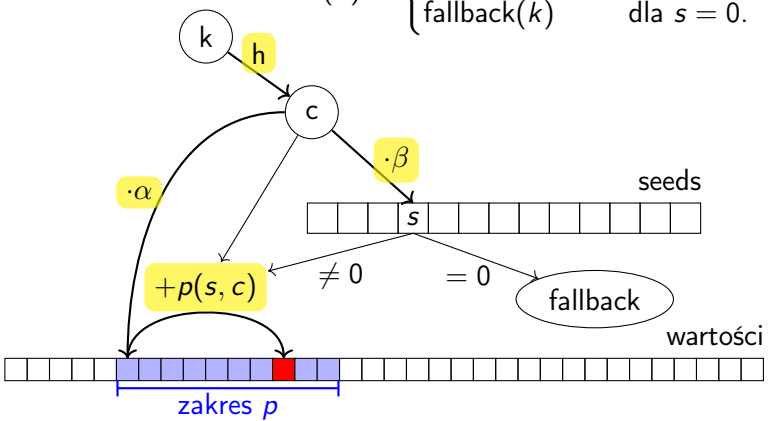


- $s = 0$ oznacza **wybicie** kubełka (dla którego nie znaleziono zalążka)
- dzięki wybijaniu mamy:
  - stały rozmiar zalążków
  - h o małym zakresie
  - proste (i szybkie) p
- dzięki małemu zakresowi p, i-ty kubełek pokrywa ograniczony zakres, przesuwany ku większym wartością gdy wzrasta i
- to czyni konstruowanie przyjaznym dla CPU-cache i wielowątkowości

Obliczanie wartości przez PHast

Wartość PHast dla klucza k o haszu $c = h(k)$ i zalążka $s = seeds[i]$ (gdzie $i = \lfloor \beta c \rfloor$ jest indeksem kubełka) to

$$f(k) = \begin{cases} \lfloor \alpha c \rfloor + p(s, c) & \text{dla } s \neq 0, \\ \text{fallback}(k) & \text{dla } s = 0. \end{cases}$$



- $s = 0$ oznacza **wybicie** kubełka (dla którego nie znaleziono zalążka)
- dzięki wybijaniu mamy:
  - stały rozmiar zalążków
  - h o małym zakresie
  - proste (i szybkie) p
- dzięki małemu zakresowi p, i-ty kubełek pokrywa ograniczony zakres, przesuwany ku większym wartością gdy wzrasta i
- to czyni konstruowanie przyjaznym dla CPU-cache i wielowątkowości

Konstruowanie PHasta

zalążki dla kubełków:

4	0	3	6	2	7	5	1		5		2						

- Kubełki są przetwarzane (przypisywane są do nich zalążki) mniej więcej po kolei, zgodnie z kolejnością ich indeksów, od lewej do prawej.
- W konsekwencji, wartości funkcji też są zajmowane mniej więcej od lewej do prawej.

Konstruowanie PHasta

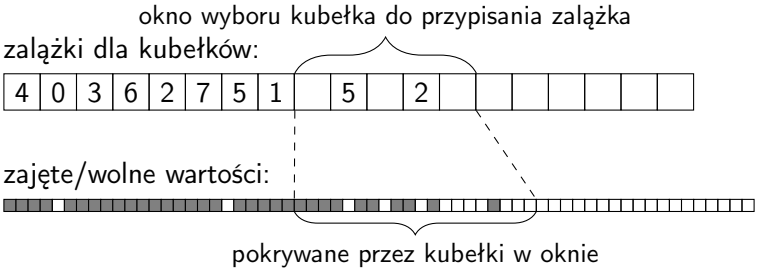
zalążki dla kubełków:

4	0	3	6	2	7	5	1		5		2						

zajęte/wolne wartości:

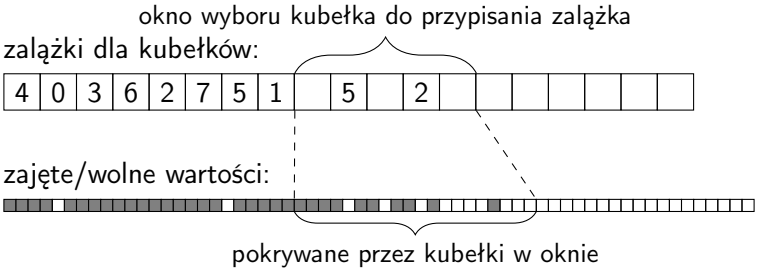
- Kubełki są przetwarzane (przypisywane są do nich zalążki) mniej więcej po kolei, zgodnie z kolejnością ich indeksów, od lewej do prawej.
- W konsekwencji, wartości funkcji też są zajmowane mniej więcej od lewej do prawej.

Konstruowanie PHasta



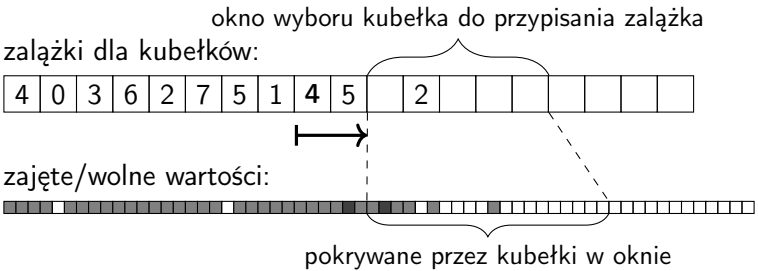
- Kolejne kubełki są wybierane z okna o wielkości W, które pokrywa ograniczony zakres wartości.
  $W = 5$ na rysunku i $W = 256$ w rzeczywistej implementacji.

Konstruowanie PHasta



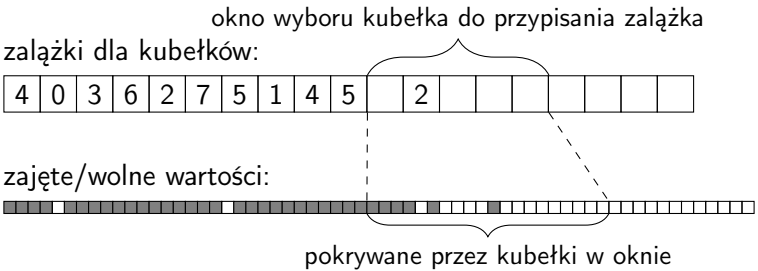
- Okno składa się z pierwszego od lewej, niepustego kubełka któremu nie przypisano jeszcze zalążka i $W - 1$ następujących po nim kubełków.
- Po przypisaniu zalążka do tego pierwszego kubełka, okno jest przesuwane w prawo.

Konstruowanie PHasta



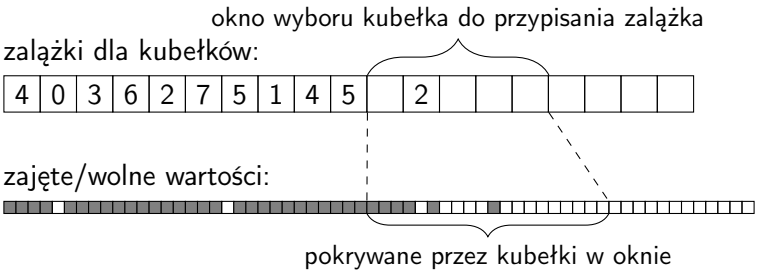
- Okno składa się z pierwszego od lewej, niepustego kubełka któremu nie przypisano jeszcze zalążka i $W - 1$ następujących po nim kubełków.
- Po przypisaniu zalążka do tego pierwszego kubełka, okno jest przesuwane w prawo.

Konstruowanie PHasta



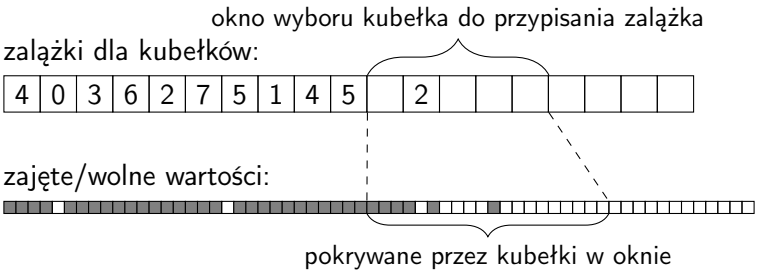
- Niepuste kubełki bez zalążków są trzymane w, i wybierane z kolejki priorytetowej typu maksimum.
- Priorytet wzrasta wraz z wielkością kubełka i maleje wraz z jego indeksem.
- To zapewnia przetwarzanie w przybliżeniu od lewej do prawej, przy jednoczesnym zachowaniu szansy na przypisanie zalążków dużym kubełkom.

Konstruowanie PHasta



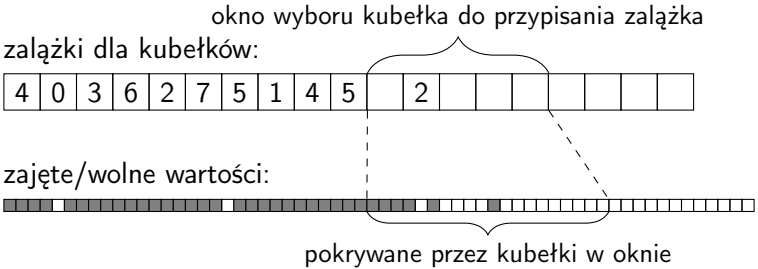
- Niepuste kubełki bez zalążków są trzymane w, i wybierane z kolejki priorytetowej typu maksimum.
- Priorytet wzrasta wraz z wielkością kubełka i maleje wraz z jego indeksem.
- To zapewnia przetwarzanie w przybliżeniu od lewej do prawej, przy jednoczesnym zachowaniu szansy na przypisanie zalążków dużym kubełkom.

Konstruowanie PHasta



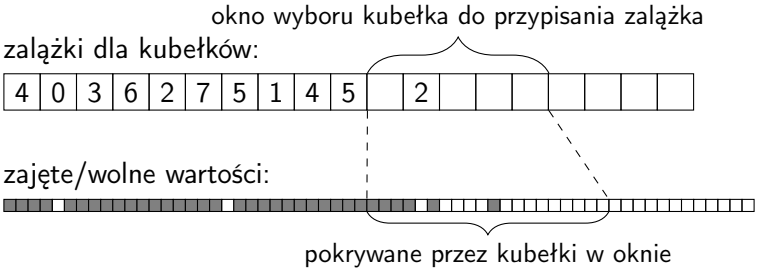
- Niepuste kubełki bez zalążków są trzymane w, i wybierane z kolejki priorytetowej typu maksimum.
- Priorytet wzrasta wraz z wielkością kubełka i maleje wraz z jego indeksem.
- To zapewnia przetwarzanie w przybliżeniu od lewej do prawej, przy jednoczesnym zachowaniu szansy na przypisanie zalążków dużym kubełkom.

Konstruowanie PHasta



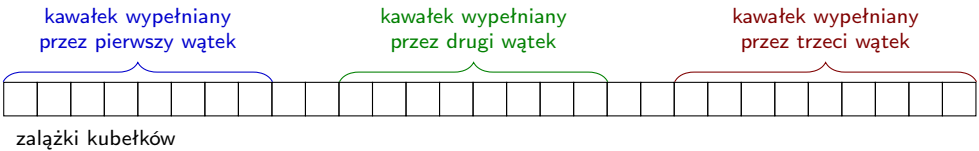
- Do wybranego kubełka przypisywany jest ten spośród zalążków niepowodujących kolizji, który minimalizuje sumę wartości przypisanych kluczom.
- Zużywanie niskich wartości funkcji jest pożądane, bo nie są one dostępne dla większości jeszcze nieprzetworzonych kubełków (tych bardziej na prawo).

Konstruowanie PHasta



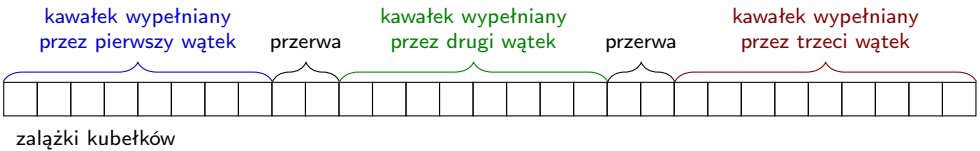
- Do wybranego kubełka przypisywany jest ten spośród zalążków niepowodujących kolizji, który minimalizuje sumę wartości przypisanych kluczom.
- Zużywanie niskich wartości funkcji jest pożądane, bo nie są one dostępne dla większości jeszcze nieprzetworzonych kubełków (tych bardziej na prawo).

Konstruowanie wielowątkowe



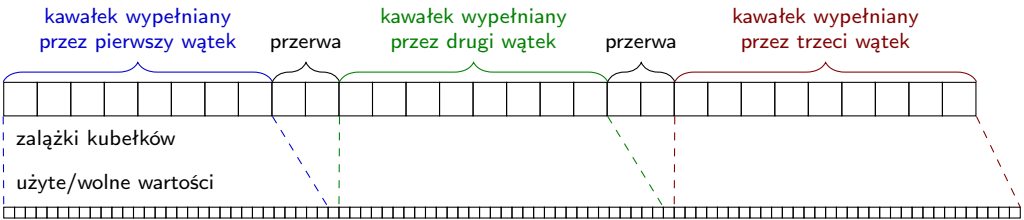
- tablica zalążków jest podzielona na kawałki wypełniane przez osobne wątki,
- i (niewielkie) przerwy pomiędzy tymi kawałkami, wypełniane później;
- dzięki tym przerwom:
- wartości funkcji pokrywane przez kawałki nie zazębiają się,
- dzięki czemu nie jest potrzebna żadna komunikacja pomiędzy wątkami

Konstruowanie wielowątkowe



- tablica zalążków jest podzielona na kawałki wypełniane przez osobne wątki,
- i (niewielkie) przerwy pomiędzy tymi kawałkami, wypełniane później;
- dzięki tym przerwom:
- wartości funkcji pokrywane przez kawałki nie zazębiają się,
- dzięki czemu nie jest potrzebna żadna komunikacja pomiędzy wątkami

Konstruowanie wielowątkowe



- tablica zalążków jest podzielona na kawałki wypełniane przez osobne wątki,
- i (niewielkie) przerwy pomiędzy tymi kawałkami, wypełniane później;
- dzięki tym przerwom:
- wartości funkcji pokrywane przez kawałki nie zazębiają się,
- dzięki czemu nie jest potrzebna żadna komunikacja pomiędzy wątkami

Szczegóły implementacyjne

- Nasza implementacja PHasta jest rozwijana w Ruście i może używać dowolnej funkcji 64-bitowej funkcji haszującej h.
- Domyślnie używany jest GxHash, który odznacza się dużą szybkością i jakością.
- Regularny PHast używa następującej funkcji rozmieszczającej

$$p(s, c) = (\lfloor s \cdot 5871781006564002453/2^{64} \rfloor \cdot c) \bmod L,$$

gdzie L ogranicza zakres p (by móc szybko wykonywać dzielenie modulo za pomocą *bitowego and*, używamy jedynie L będących potęgami dwójki).

- PHast$^+$ – wariant z szybkim (bitowo-zrównoleglonym) konstruowaniem – używa:

$$p(s, c) = c \bmod L + s - 1, \tag{1}$$

$$p(s, c) = (c + \delta s) \bmod L, \tag{2}$$

gdzie δ jest parametrem metody (używamy $\delta \in \{1, 2, 3\}$).

Szczegóły implementacyjne

- Nasza implementacja PHasta jest rozwijana w Ruście i może używać dowolnej funkcji 64-bitowej funkcji haszującej h.
- Domyślnie uzywany jest GxHash, który odznacza się dużą szybkością i jakością.
- Regularny PHast używa następującej funkcji rozmieszczającej

$$p(s, c) = (\lfloor s \cdot 5871781006564002453/2^{64} \rfloor \cdot c) \bmod L,$$

gdzie L ogranicza zakres p (by móc szybko wykonywać dzielenie modulo za pomocą *bitowego and*, używamy jedynie L będących potęgami dwójki).

- PHast$^+$ – wariant z szybkim (bitowo-zrównoleglonym) konstruowaniem – używa:

$$p(s, c) = c \bmod L + s - 1, \tag{1}$$

$$p(s, c) = (c + \delta s) \bmod L, \tag{2}$$

gdzie δ jest parametrem metody (używamy $\delta \in \{1, 2, 3\}$).

Szczegóły implementacyjne

- Nasza implementacja PHasta jest rozwijana w Ruście i może używać dowolnej funkcji 64-bitowej funkcji haszującej h.
- Domyślnie uzywany jest GxHash, który odznacza się dużą szybkością i jakością.
- Regularny PHast używa następującej funkcji rozmieszczającej

$$p(s, c) = (\lfloor s \cdot 5871781006564002453/2^{64} \rfloor \cdot c) \bmod L,$$

gdzie L ogranicza zakres p (by móc szybko wykonywać dzielenie modulo za pomocą *bitowego and*, używamy jedynie L będących potęgami dwójki).

- PHast$^+$ – wariant z szybkim (bitowo-zrównoleglonym) konstruowaniem – używa:

$$p(s, c) = c \bmod L + s - 1, \tag{1}$$

$$p(s, c) = (c + \delta s) \bmod L, \tag{2}$$

gdzie δ jest parametrem metody (używamy $\delta \in \{1, 2, 3\}$).

Szczegóły implementacyjne

- Nasza implementacja PHasta jest rozwijana w Ruście i może używać dowolnej funkcji 64-bitowej funkcji haszującej h.
- Domyślnie uzywany jest GxHash, który odznacza się dużą szybkością i jakością.
- Regularny PHast używa następującej funkcji rozmieszczającej

$$p(s, c) = (\lfloor s \cdot 5871781006564002453/2^{64} \rfloor \cdot c) \bmod L,$$

gdzie L ogranicza zakres p (by móc szybko wykonywać dzielenie modulo za pomocą *bitowego and*, używamy jedynie L będących potęgami dwójki).

- PHast$^+$ – wariant z szybkim (bitowo-zrównoleglonym) konstruowaniem – używa:

$$p(s, c) = c \bmod L + s - 1, \tag{1}$$

$$p(s, c) = (c + \delta s) \bmod L, \tag{2}$$

gdzie δ jest parametrem metody (używamy $\delta \in \{1, 2, 3\}$).

Szybkie konstruowanie PHast$^+$

Niech C będzie multi-zbiorem wartości haszujących kluczy w kubełku.
Dla $p(s, c) = c \bmod L + s - 1$, zalążek s minimalizujący $\sum_{c \in C} p(s, c)$ można szybko znaleźć, gdyż:

- ponieważ wartości p rosną z s ($p(s + 1, c) = p(s, c) + 1$), wystarczy znaleźć najmniejszy bezkolizyjny zalążek;
- test kolizji pomiędzy kluczami w kubełku wystarczy wykonać raz, bo jego wynik nie zależy od s;
- można zrównoleglić sprawdzanie bezkolizyjności zalążków za pomocą operacji bitowych.

Dla $p(s, c) = (c + \delta s) \bmod L$:

- powyższe fakty zachodzą i pomagają znaleźć optymalne zalążki osobno w przedziałach, takich że:
  zalążki a, b są w tym samym przedziale $\Leftrightarrow \lfloor \frac{c+\delta a}{L} \rfloor = \lfloor \frac{c+\delta b}{L} \rfloor$ dla wszystkich $c \in C$;
- większe δ to większa liczba przedziałów i, co za tym idzie, czas konstruowania, ale też mniejszy rozmiar skonstruowanej funkcji.

Szybkie konstruowanie PHast$^+$

Niech C będzie multi-zbiorem wartości haszujących kluczy w kubełku.
Dla $p(s, c) = c \bmod L + s - 1$, zalążek s minimalizujący $\sum_{c \in C} p(s, c)$ można szybko znaleźć, gdyż:

- ponieważ wartości p rosną z s ($p(s + 1, c) = p(s, c) + 1$), wystarczy znaleźć najmniejszy bezkolizyjny zalążek;
- test kolizji pomiędzy kluczami w kubełku wystarczy wykonać raz, bo jego wynik nie zależy od s;
- można zrównoleglić sprawdzanie bezkolizyjności zalążków za pomocą operacji bitowych.

Dla $p(s, c) = (c + \delta s) \bmod L$:

- powyższe fakty zachodzą i pomagają znaleźć optymalne zalążki osobno w przedziałach, takich że:
  zalążki a, b są w tym samym przedziale $\Leftrightarrow \lfloor \frac{c+\delta a}{L} \rfloor = \lfloor \frac{c+\delta b}{L} \rfloor$ dla wszystkich $c \in C$;
- większe δ to większa liczba przedziałów i, co za tym idzie, czas konstruowania, ale też mniejszy rozmiar skonstruowanej funkcji.

Szybkie konstruowanie PHast$^+$

Niech C będzie multi-zbiorem wartości haszujących kluczy w kubełku.
Dla $p(s, c) = c \bmod L + s - 1$, zalążek s minimalizujący $\sum_{c \in C} p(s, c)$ można szybko znaleźć, gdyż:

- ponieważ wartości p rosną z s ($p(s + 1, c) = p(s, c) + 1$), wystarczy znaleźć najmniejszy bezkolizyjny zalążek;
- test kolizji pomiędzy kluczami w kubełku wystarczy wykonać raz, bo jego wynik nie zależy od s;
- można zrównoleglić sprawdzanie bezkolizyjności zalążków za pomocą operacji bitowych.

Dla $p(s, c) = (c + \delta s) \bmod L$:

- powyższe fakty zachodzą i pomagają znaleźć optymalne zalążki osobno w przedziałach, takich że:
  zalążki a, b są w tym samym przedziale $\Leftrightarrow \lfloor \frac{c+\delta a}{L} \rfloor = \lfloor \frac{c+\delta b}{L} \rfloor$ dla wszystkich $c \in C$;
- większe δ to większa liczba przedziałów i, co za tym idzie, czas konstruowania, ale też mniejszy rozmiar skonstruowanej funkcji.

Szybkie konstruowanie PHast$^+$

Niech C będzie multi-zbiorem wartości haszujących kluczy w kubełku.
Dla $p(s, c) = c \bmod L + s - 1$, zalążek s minimalizujący $\sum_{c \in C} p(s, c)$ można szybko znaleźć, gdyż:

- ponieważ wartości p rosną z s ($p(s + 1, c) = p(s, c) + 1$), wystarczy znaleźć najmniejszy bezkolizyjny zalążek;
- test kolizji pomiędzy kluczami w kubełku wystarczy wykonać raz, bo jego wynik nie zależy od s;
- można zrównoleglić sprawdzanie bezkolizyjności zalążków za pomocą operacji bitowych.

Dla $p(s, c) = (c + \delta s) \bmod L$:

- powyższe fakty zachodzą i pomagają znaleźć optymalne zalążki osobno w przedziałach, takich że:
  zalążki a, b są w tym samym przedziale $\Leftrightarrow \lfloor \frac{c+\delta a}{L} \rfloor = \lfloor \frac{c+\delta b}{L} \rfloor$ dla wszystkich $c \in C$;
- większe δ to większa liczba przedziałów i, co za tym idzie, czas konstruowania, ale też mniejszy rozmiar skonstruowanej funkcji.

Szybkie konstruowanie PHast$^+$

Niech C będzie multi-zbiorem wartości haszujących kluczy w kubełku.
Dla $p(s, c) = c \bmod L + s - 1$, zalążek s minimalizujący $\sum_{c \in C} p(s, c)$ można szybko znaleźć, gdyż:

- ponieważ wartości p rosną z s ($p(s + 1, c) = p(s, c) + 1$), wystarczy znaleźć najmniejszy bezkolizyjny zalążek;
- test kolizji pomiędzy kluczami w kubełku wystarczy wykonać raz, bo jego wynik nie zależy od s;
- można zrównoleglić sprawdzanie bezkolizyjności zalążków za pomocą operacji bitowych.

Dla $p(s, c) = (c + \delta s) \bmod L$:

- powyższe fakty zachodzą i pomagają znaleźć optymalne zalążki osobno w przedziałach, takich że:
  zalążki a, b są w tym samym przedziale $\Leftrightarrow \lfloor \frac{c+\delta a}{L} \rfloor = \lfloor \frac{c+\delta b}{L} \rfloor$ dla wszystkich $c \in C$;
- większe δ to większa liczba przedziałów i, co za tym idzie, czas konstruowania, ale też mniejszy rozmiar skonstruowanej funkcji.

Szybkie konstruowanie PHast$^+$

Niech C będzie multi-zbiorem wartości haszujących kluczy w kubełku.
Dla $p(s, c) = c \bmod L + s - 1$, zalążek s minimalizujący $\sum_{c \in C} p(s, c)$ można szybko znaleźć, gdyż:

- ponieważ wartości p rosną z s ($p(s + 1, c) = p(s, c) + 1$), wystarczy znaleźć najmniejszy bezkolizyjny zalążek;
- test kolizji pomiędzy kluczami w kubełku wystarczy wykonać raz, bo jego wynik nie zależy od s;
- można zrównoleglić sprawdzanie bezkolizyjności zalążków za pomocą operacji bitowych.

Dla $p(s, c) = (c + \delta s) \bmod L$:

- powyższe fakty zachodzą i pomagają znaleźć optymalne zalążki osobno w przedziałach, takich że:
  zalążki a, b są w tym samym przedziale $\Leftrightarrow \lfloor \frac{c+\delta a}{L} \rfloor = \lfloor \frac{c+\delta b}{L} \rfloor$ dla wszystkich $c \in C$;
- większe δ to większa liczba przedziałów i, co za tym idzie, czas konstruowania, ale też mniejszy rozmiar skonstruowanej funkcji.

Bitowo-zrównleglone sprawdzanie zalążków w PHast$^+$

Równoczesne badanie kolizyjności k ($k = 64$ dla współczesnych CPU; $k = 8$ na rysunku) kolejnych zalążków $(s, s+1, \ldots, s+k-1)$:

- Dla każdego c w kubełku, z bitmapy użytych wartości, czytamy k kolejnych bitów poczynając wartości dla pary (s, c).
- Niech r będzie *bitowym LUB* odczytanych słów.
- i-ty bit r to 0 $\Leftrightarrow$ zalążek $s + i$ nie powoduje kolizji.
- Najmniej znaczące 0 odpowiada szukanemu, pierwszemu zalążkowi, który nie powoduje kolizji.
- Jeśli nie ma 0 w r, sprawdzamy kolejne k zalążków...
- ...aż do napotkania bezkolizyjnego albo ostatniego.
- Dla $p(s, c) = (c + \delta s) \bmod L$ i $\delta > 1$, bity odpowiadające nielegalnym przesunięciom wypełniamy jedynkami.

wartość użyta (bit 1)

wartość wolna (bit 0)

Bitowo-zrównleglone sprawdzanie zalążków w PHast$^+$

Równoczesne badanie kolizyjności k ($k = 64$ dla współczesnych CPU; $k = 8$ na rysunku) kolejnych zalążków $(s, s+1, \ldots, s+k-1)$:

- Dla każdego c w kubełku, z bitmapy użytych wartości, czytamy k kolejnych bitów poczynając wartości dla pary (s, c).
- Niech r będzie *bitowym LUB* odczytanych słów.
- i-ty bit r to 0 $\Leftrightarrow$ zalążek $s + i$ nie powoduje kolizji.
- Najmniej znaczące 0 odpowiada szukanemu, pierwszemu zalążkowi, który nie powoduje kolizji.
- Jeśli nie ma 0 w r, sprawdzamy kolejne k zalążków...
- ...aż do napotkania bezkolizyjnego albo ostatniego.
- Dla $p(s, c) = (c + \delta s) \bmod L$ i $\delta > 1$, bity odpowiadające nielegalnym przesunięciom wypełniamy jedynkami.

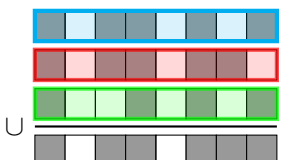
wartość użyta (bit 1)

wartość wolna (bit 0)

Bitowo-zrównleglone sprawdzanie zalążków w PHast$^+$

Równoczesne badanie kolizyjności k ($k = 64$ dla współczesnych CPU; $k = 8$ na rysunku) kolejnych zalążków $(s, s+1, \ldots, s+k-1)$:

- Dla każdego c w kubełku, z bitmapy użytych wartości, czytamy k kolejnych bitów poczynając wartości dla pary (s, c).
- Niech r będzie *bitowym LUB* odczytanych słów.
- i-ty bit r to 0 $\Leftrightarrow$ zalążek $s + i$ nie powoduje kolizji.
- Najmniej znaczące 0 odpowiada szukanemu, pierwszemu zalążkowi, który nie powoduje kolizji.
- Jeśli nie ma 0 w r, sprawdzamy kolejne k zalążków...
- ...aż do napotkania bezkolizyjnego albo ostatniego.
- Dla $p(s, c) = (c + \delta s) \bmod L$ i $\delta > 1$, bity odpowiadające nielegalnym przesunięciom wypełniamy jedynkami.



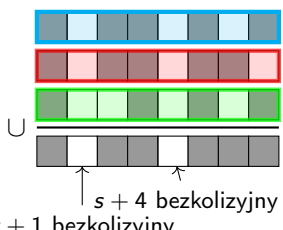
wartość użyta (bit 1)

wartość wolna (bit 0)

Bitowo-zrównleglone sprawdzanie zalążków w PHast$^+$

Równoczesne badanie kolizyjności k ($k = 64$ dla współczesnych CPU; $k = 8$ na rysunku) kolejnych zalążków $(s, s+1, \ldots, s+k-1)$:

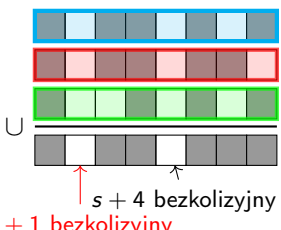
- Dla każdego c w kubełku, z bitmapy użytych wartości, czytamy k kolejnych bitów poczynając wartości dla pary (s, c).
- Niech r będzie *bitowym LUB* odczytanych słów.
- i-ty bit r to 0 $\Leftrightarrow$ zalążek $s + i$ nie powoduje kolizji.
- Najmniej znaczące 0 odpowiada szukanemu, pierwszemu zalążkowi, który nie powoduje kolizji.
- Jeśli nie ma 0 w r, sprawdzamy kolejne k zalążków...
- ...aż do napotkania bezkolizyjnego albo ostatniego.
- Dla $p(s, c) = (c + \delta s) \bmod L$ i $\delta > 1$, bity odpowiadające nielegalnym przesunięciom wypełniamy jedynkami.



Bitowo-zrównleglone sprawdzanie zalążków w PHast$^+$

Równoczesne badanie kolizyjności k ($k = 64$ dla współczesnych CPU; $k = 8$ na rysunku) kolejnych zalążków $(s, s+1, \ldots, s+k-1)$:

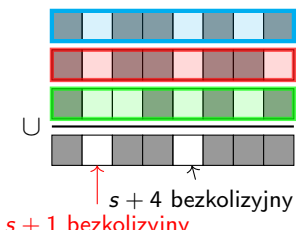
- Dla każdego c w kubełku, z bitmapy użytych wartości, czytamy k kolejnych bitów poczynając wartości dla pary (s, c).
- Niech r będzie *bitowym LUB* odczytanych słów.
- i-ty bit r to 0 $\Leftrightarrow$ zalążek $s + i$ nie powoduje kolizji.
- Najmniej znaczące 0 odpowiada szukanemu, pierwszemu zalążkowi, który nie powoduje kolizji.
- Jeśli nie ma 0 w r, sprawdzamy kolejne k zalążków...
- ...aż do napotkania bezkolizyjnego albo ostatniego.
- Dla $p(s, c) = (c + \delta s) \bmod L$ i $\delta > 1$, bity odpowiadające nielegalnym przesunięciom wypełniamy jedynkami.



Bitowo-zrównleglone sprawdzanie zalążków w PHast$^+$

Równoczesne badanie kolizyjności k ($k = 64$ dla współczesnych CPU; $k = 8$ na rysunku) kolejnych zalążków $(s, s+1, \ldots, s+k-1)$:

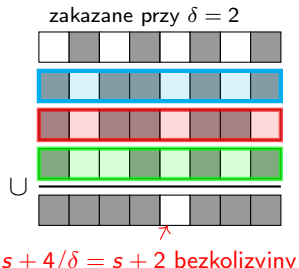
- Dla każdego c w kubełku, z bitmapy użytych wartości, czytamy k kolejnych bitów poczynając wartości dla pary (s, c).
- Niech r będzie *bitowym LUB* odczytanych słów.
- i-ty bit r to 0 $\Leftrightarrow$ zalążek $s + i$ nie powoduje kolizji.
- Najmniej znaczące 0 odpowiada szukanemu, pierwszemu zalążkowi, który nie powoduje kolizji.
- Jeśli nie ma 0 w r, sprawdzamy kolejne k zalążków...
- ...aż do napotkania bezkolizyjnego albo ostatniego.
- Dla $p(s, c) = (c + \delta s) \bmod L$ i $\delta > 1$, bity odpowiadające nielegalnym przesunięciom wypełniamy jedynkami.



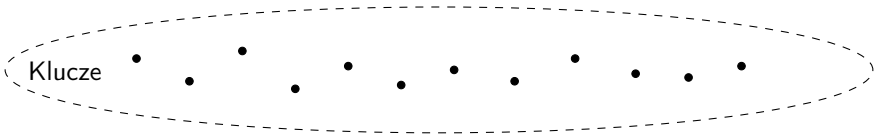
Bitowo-zrównleglone sprawdzanie zalążków w PHast$^+$

Równoczesne badanie kolizyjności k ($k = 64$ dla współczesnych CPU; $k = 8$ na rysunku) kolejnych zalążków $(s, s+1, \ldots, s+k-1)$:

- Dla każdego c w kubełku, z bitmapy użytych wartości, czytamy k kolejnych bitów poczynając wartości dla pary (s, c).
- Niech r będzie *bitowym LUB* odczytanych słów.
- i-ty bit r to 0 $\Leftrightarrow$ zalążek $s + i$ nie powoduje kolizji.
- Najmniej znaczące 0 odpowiada szukanemu, pierwszemu zalążkowi, który nie powoduje kolizji.
- Jeśli nie ma 0 w r, sprawdzamy kolejne k zalążków...
- ...aż do napotkania bezkolizyjnego albo ostatniego.
- Dla $p(s, c) = (c + \delta s) \bmod L$ i $\delta > 1$, bity odpowiadające nielegalnym przesunięciom wypełniamy jedynkami.



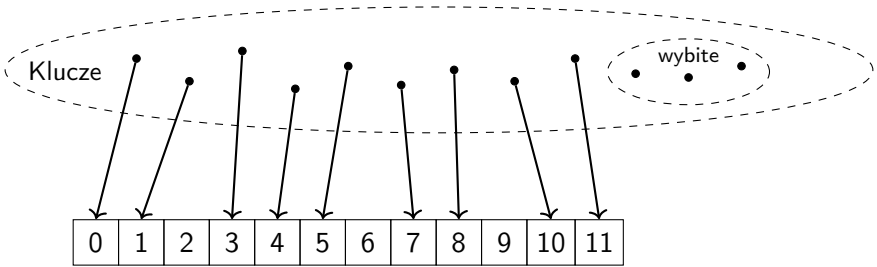
(Minimalne) Doskonałe Funkcje Haszujące



Budowa **doskonałej funkcji haszującej** dla n_1 kluczy polega na skonstruowaniu szeregu funkcji (które mogą wybijać niektóre klucze):

- f_1 mapuje n_1 kluczy wejściowych na $\{0, \ldots, n_1 - 1\}$,
- f_2 mapuje wszystkie n_2 klucze wybite przez f_1 na $\{0, \ldots, n_2 - 1\}$,
- f_3 mapuje wszystkie n_3 klucze wybite przez f_2 na $\{0, \ldots, n_3 - 1\}$,
- ... i tak dalej, aż żadne klucze nie są wybite.

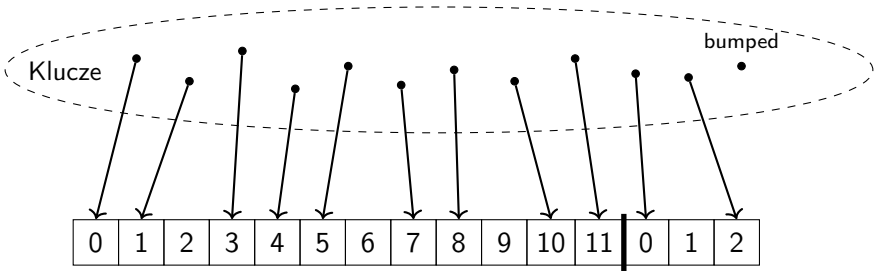
(Minimalne) Doskonałe Funkcje Haszujące



Budowa **doskonałej funkcji haszującej** dla n_1 kluczy polega na skonstruowaniu szeregu funkcji (które mogą wybijać niektóre klucze):

- f_1 mapuje n_1 kluczy wejściowych na $\{0, \ldots, n_1 - 1\}$,
- f_2 mapuje wszystkie n_2 klucze wybite przez f_1 na $\{0, \ldots, n_2 - 1\}$,
- f_3 mapuje wszystkie n_3 klucze wybite przez f_2 na $\{0, \ldots, n_3 - 1\}$,
- ... i tak dalej, aż żadne klucze nie są wybite.

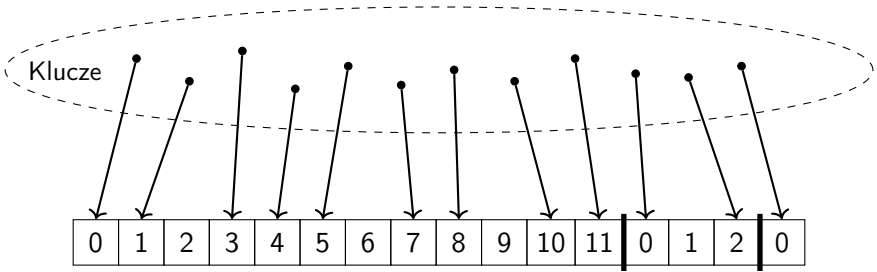
(Minimalne) Doskonałe Funkcje Haszujące



Budowa **doskonałej funkcji haszującej** dla n_1 kluczy polega na skonstruowaniu szeregu funkcji (które mogą wybijać niektóre klucze):

- f_1 mapuje n_1 kluczy wejściowych na $\{0, \ldots, n_1 - 1\}$,
- f_2 mapuje wszystkie n_2 klucze wybite przez f_1 na $\{0, \ldots, n_2 - 1\}$,
- f_3 mapuje wszystkie n_3 klucze wybite przez f_2 na $\{0, \ldots, n_3 - 1\}$,
- ... i tak dalej, aż żadne klucze nie są wybite.

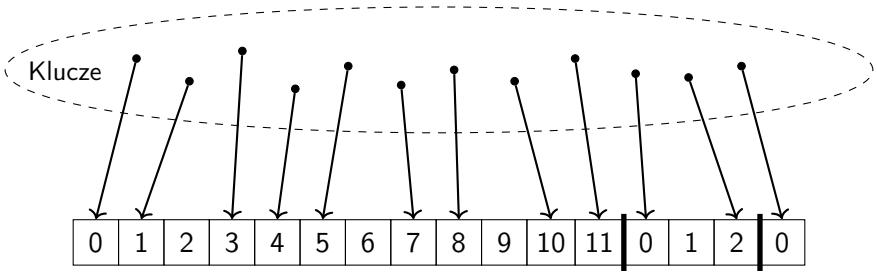
(Minimalne) Doskonałe Funkcje Haszujące



Budowa **doskonałej funkcji haszującej** dla n_1 kluczy polega na skonstruowaniu szeregu funkcji (które mogą wybijać niektóre klucze):

- f_1 mapuje n_1 kluczy wejściowych na $\{0, \ldots, n_1 - 1\}$,
- f_2 mapuje wszystkie n_2 klucze wybite przez f_1 na $\{0, \ldots, n_2 - 1\}$,
- f_3 mapuje wszystkie n_3 klucze wybite przez f_2 na $\{0, \ldots, n_3 - 1\}$,
- ... i tak dalej, aż żadne klucze nie są wybite.

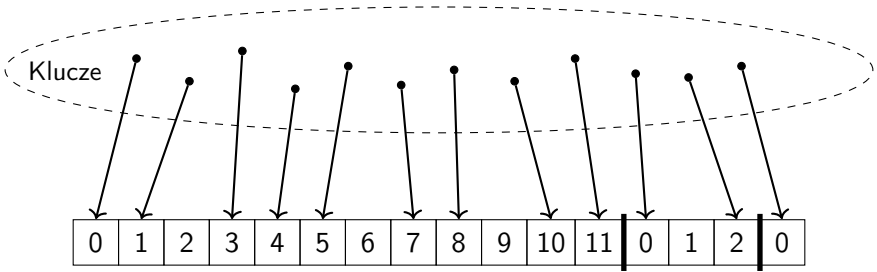
(Minimalne) Doskonałe Funkcje Haszujące



Budowa **doskonałej funkcji haszującej** dla n_1 kluczy polega na skonstruowaniu szeregu funkcji (które mogą wybijać niektóre klucze):

- f_1 mapuje n_1 kluczy wejściowych na $\{0, \ldots, n_1 - 1\}$,
- f_2 mapuje wszystkie n_2 klucze wybite przez f_1 na $\{0, \ldots, n_2 - 1\}$,
- f_3 mapuje wszystkie n_3 klucze wybite przez f_2 na $\{0, \ldots, n_3 - 1\}$,
- ... i tak dalej, aż żadne klucze nie są wybite.

(Minimalne) Doskonałe Funkcje Haszujące

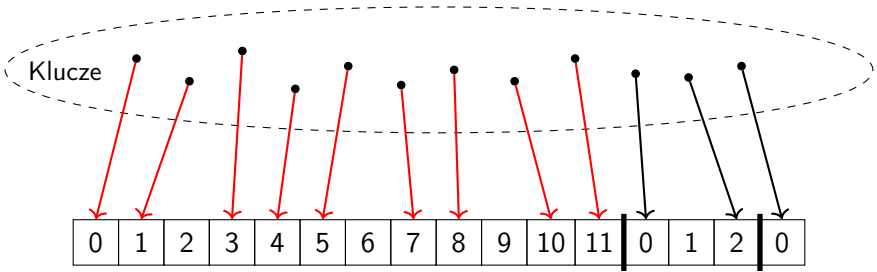


Obliczenie wartości dla klucza k polega na policzeniu $f_1(k), f_2(k), \ldots, f_i(k)$, gdzie f_i jest pierwszą funkcją niewybijającą k.

Jeśli już f_1 nie wybija k (co zdarza się dla około $\sim 99\%$ kluczy w typowych konfiguracjach) $f_1(k)$ jest ostatecznie zwróconą wartością.

W przeciwnym razie, jest nią $n_1 + \cdots + n_{i-1} + f_i(k)$.

(Minimalne) Doskonałe Funkcje Haszujące

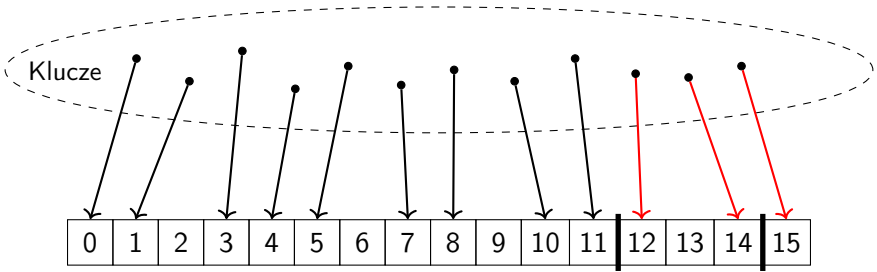


Obliczenie wartości dla klucza k polega na policzeniu $f_1(k), f_2(k), \ldots, f_i(k)$, gdzie f_i jest pierwszą funkcją niewybijającą k.

Jeśli już f_1 nie wybija k (co zdarza się dla około $\sim 99\%$ kluczy w typowych konfiguracjach) $f_1(k)$ jest ostatecznie zwróconą wartością.

W przeciwnym razie, jest nią $n_1 + \cdots + n_{i-1} + f_i(k)$.

(Minimalne) Doskonałe Funkcje Haszujące

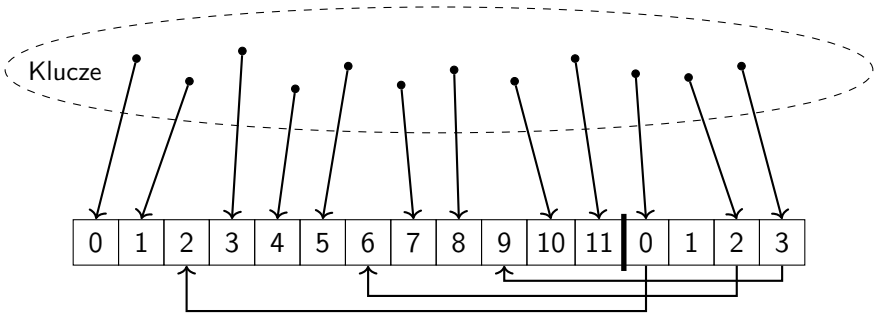


Obliczenie wartości dla klucza k polega na policzeniu $f_1(k), f_2(k), \ldots, f_i(k)$, gdzie f_i jest pierwszą funkcją niewybijającą k.

Jeśli już f_1 nie wybija k (co zdarza się dla około $\sim 99\%$ kluczy w typowych konfiguracjach) $f_1(k)$ jest ostatecznie zwróconą wartością.

W przeciwnym razie, jest nią $n_1 + \cdots + n_{i-1} + f_i(k)$.

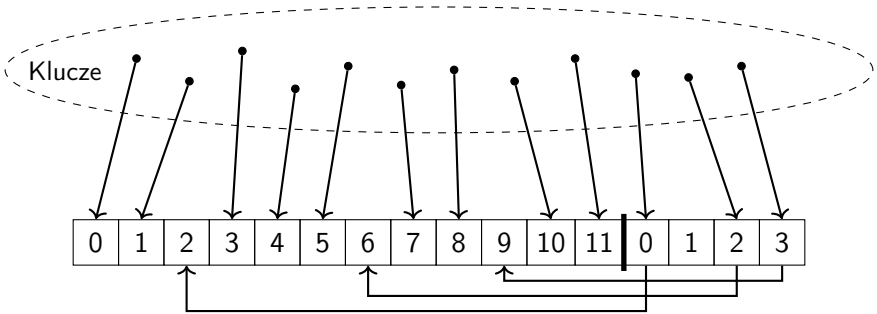
(Minimalne) Doskonałe Funkcje Haszujące



By uzyskać funkcję **minimalną**, używana jest tablica M mapująca kolejne wartości powyżej $n_1 - 1$, na kolejne nieużyte (ze względu na wybijanie przez f_1) wartości poniżej n_1:
v jest mapowane na $M[v - n_1]$.

M można zwięźle zapisać używając kodowania Eliasa-Fano.

(Minimalne) Doskonałe Funkcje Haszujące



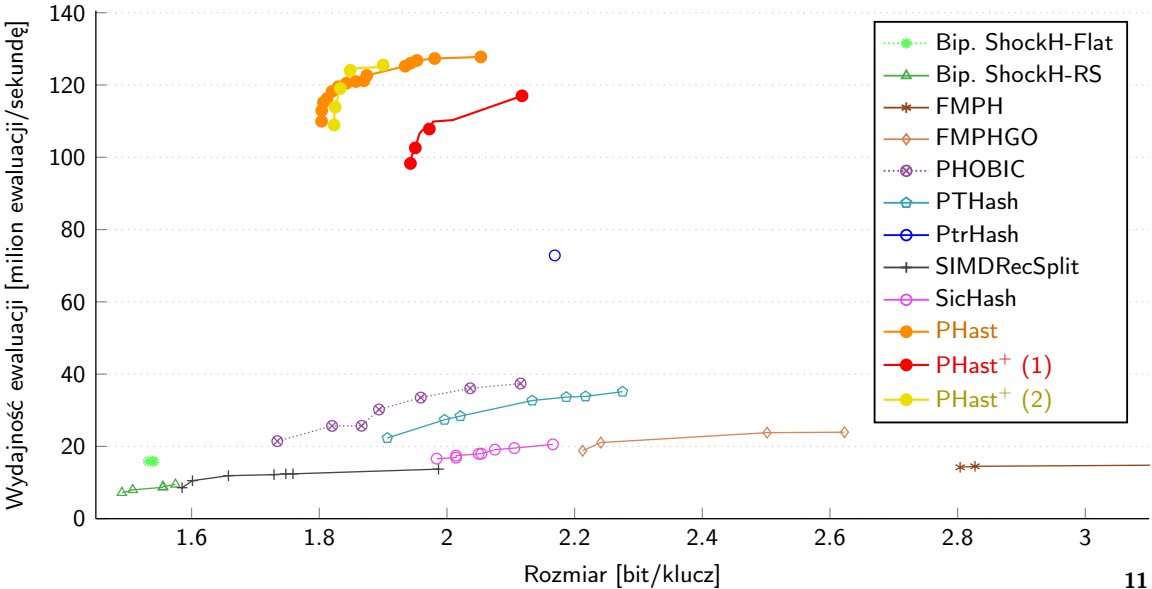
By uzyskać funkcję **minimalną**, używana jest tablica M mapująca kolejne wartości powyżej $n_1 - 1$, na kolejne nieużyte (ze względu na wybijanie przez f_1) wartości poniżej n_1:
v jest mapowane na $M[v - n_1]$.

M można zwięźle zapisać używając kodowania Eliasa-Fano.

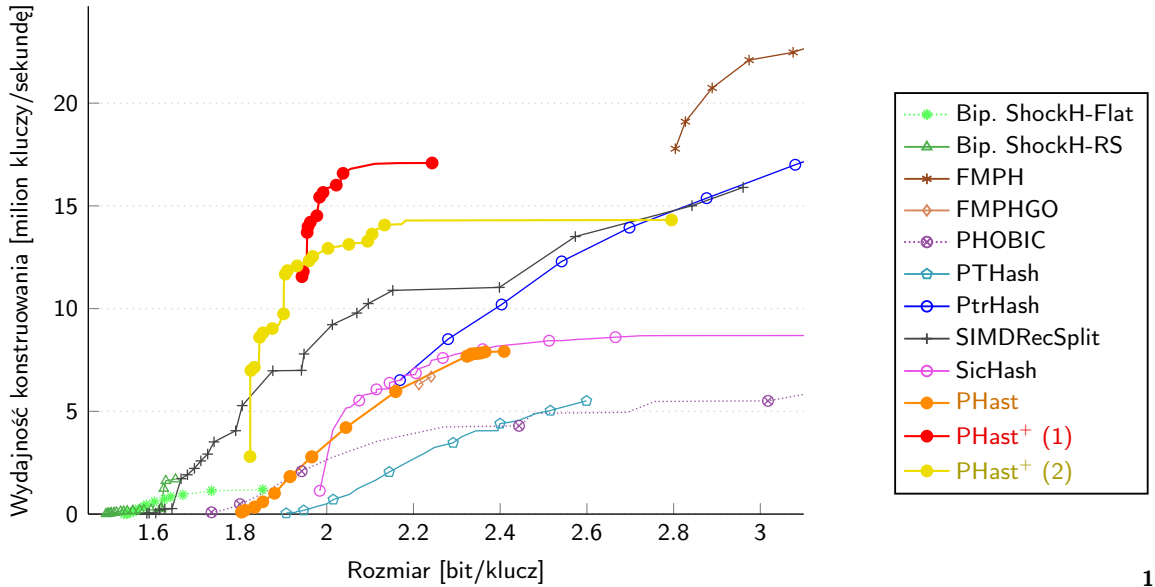
Badanie Wydajności

- badanie za pomocą oprogramowania *MPHF-Experiments* rozwijanego przez Hansa-Petera Lehmanna;
- dla 50 milionów klucz,
- o średniej długości $\sim$ 30 bajtów / klucz;
- konstruowania jedno-wątkowego;
- na AMD Ryzen 5600G @3.9GHz CPU;
- metody zaimplementowane w Ruście (w tym PHast) i C++.

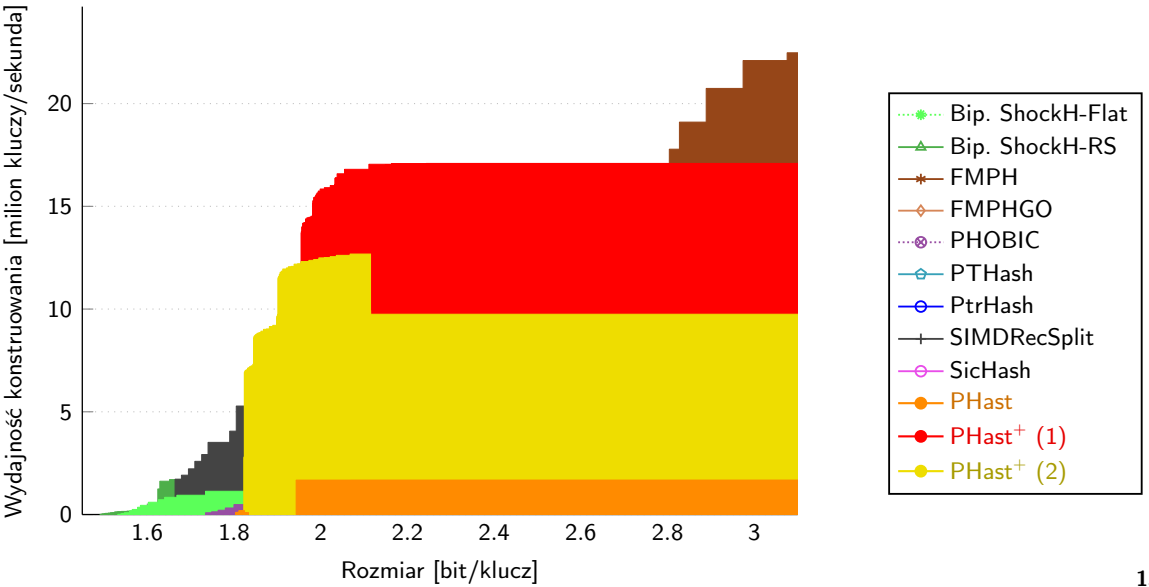
Osiągane kompromisy pomiędzy wielkością i szybkością ewaluacji



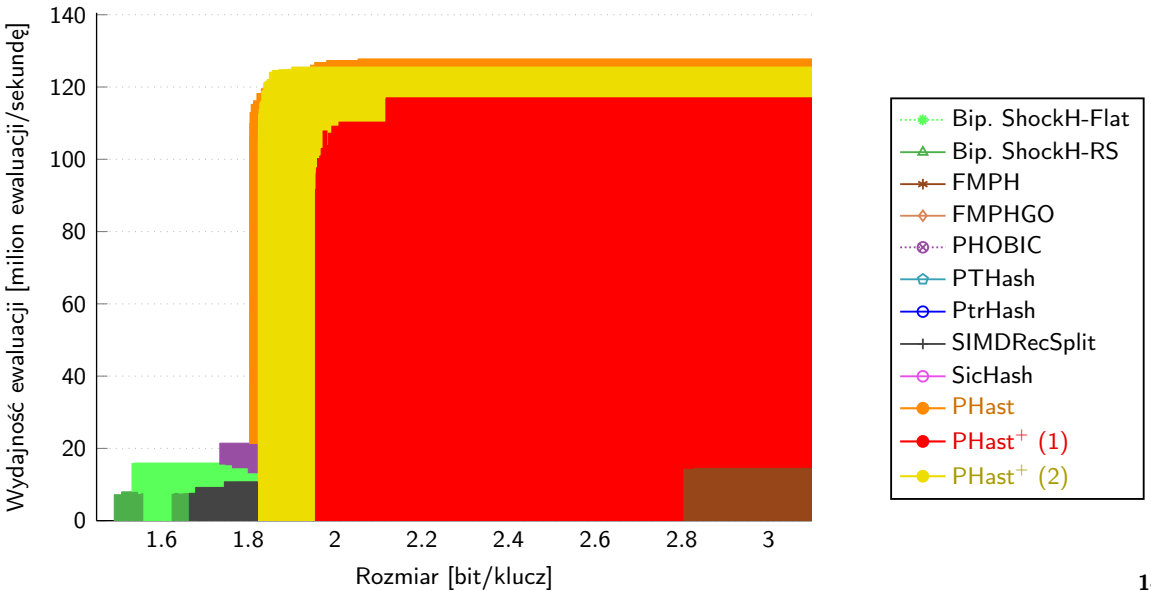
Osiągane kompromisy pomiędzy wielkością i szybkością konstruowania



Najszybsze w ewulacji dla zadanych rozmiaru/szybkości konstruowania



Najszybsze do skonstruowania dla zadanych rozmiaru/szybkości ewaluacji



Bibliografia

- Piotr Beling, Peter Sanders *PHast - Perfect Hashing made fast* 2026 Proceedings of the SIAM Symposium on Algorithm Engineering and Experiments (ALENEX) pełna, najnowsza wersja dostępna jest na https://arxiv.org/abs/2504.17918
- Hans-Peter Lehmann, Thomas Mueller, Rasmus Pagh, Giulio Ermanno Pibiri, Peter Sanders, Sebastiano Vigna, Stefan Walzer, *Modern Minimal Perfect Hashing: A Surve*, ACM Computing Surveys, tom 58, numer 10, dostępny na https://arxiv.org/abs/2506.06536
- Stefan Hermann, Hans-Peter Lehmann, Giulio Ermanno Pibiri, Peter Sanders, Stefan Walzer, *PHOBIC: Perfect Hashing With Optimized Bucket Sizes and Interleaved Coding*, ESA 2024
- Giulio Ermanno Pibiri, Roberto Trani, *PTHash: Revisiting FCH Minimal Perfect Hashing*, SIGIR 2021
- Oprogramowanie: https://crates.io/crates/ph https://github.com/beling/bsuccinct-rs https://github.com/beling/MPHF-Experiments