Algorytmy i struktury danych

Tablice haszowane

Krzysztof M. Ocetkiewicz
Krzysztof.Ocetkiewicz@eti.pg.edu.pl

Katedra Algorytmów i Modelowania Systemów, WETI, PG

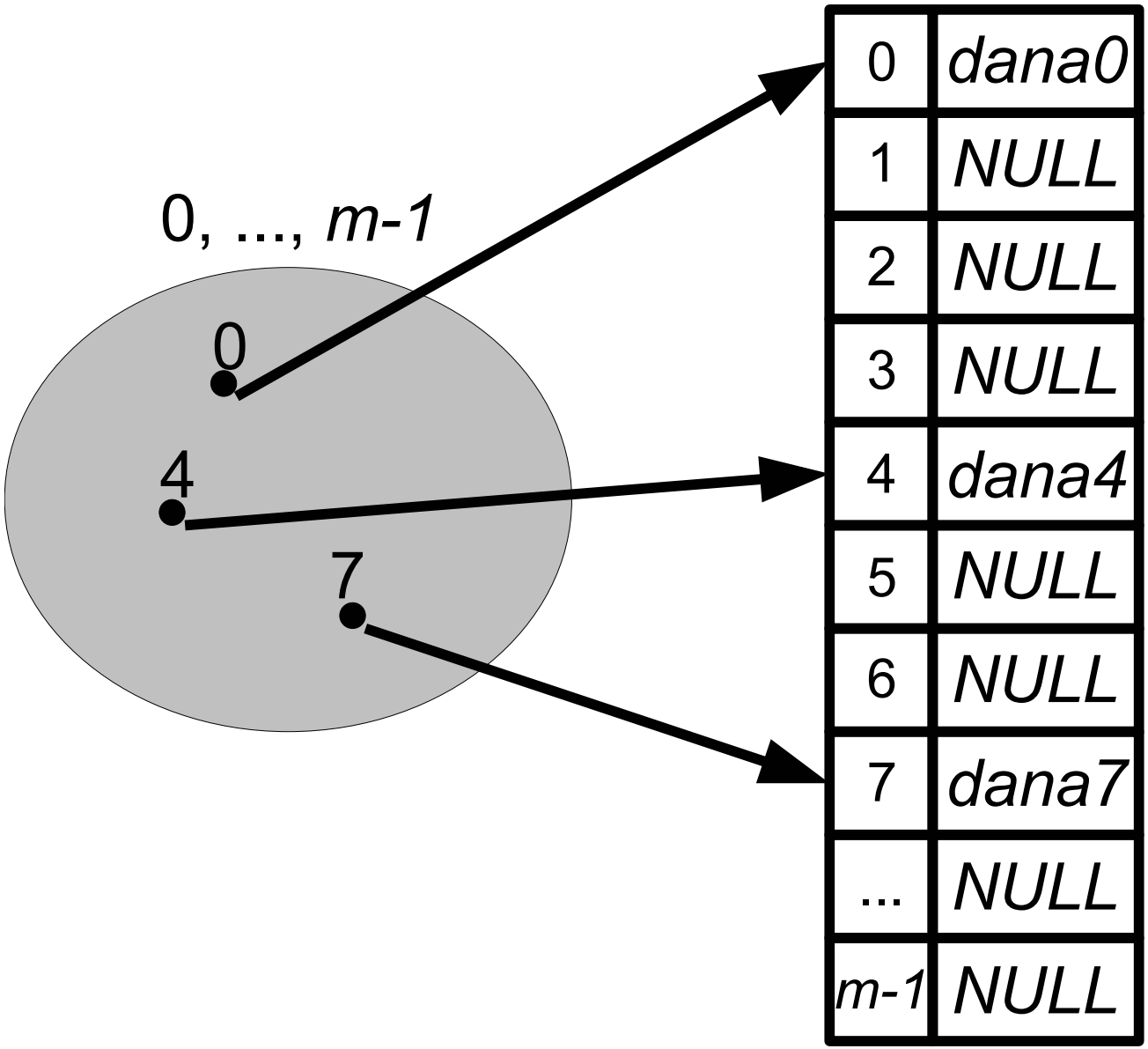
Tablice z haszowaniem

- pozwalają na szybkie wykonywanie operacji wstawiania, usuwania i wyszukiwania elementu o danym kluczu
- pesymistyczna złożoność każdej z tych operacji wynosi $O(n)$
- w praktyce, przy rozsądnych założeniach, oczekiwany czas tych operacji wynosi $O(1)$

Adresowanie bezpośrednie

- małe uniwersum kluczy z zakresu $0, \ldots, m-1$
- tablica m-elementowa T
- jeżeli element o kluczu k nie należy do tablicy, to $T[k] = \mathit{nullptr}$
- wstawianie:
  `WSTAW(klucz, wart) { T[klucz] = wart; }`
- usuwanie:
  `USUN(klucz) { T[klucz] =` *nullptr*`; }`
- wyszukiwanie:
  `ZNAJDZ(klucz) { return T[klucz]; }`

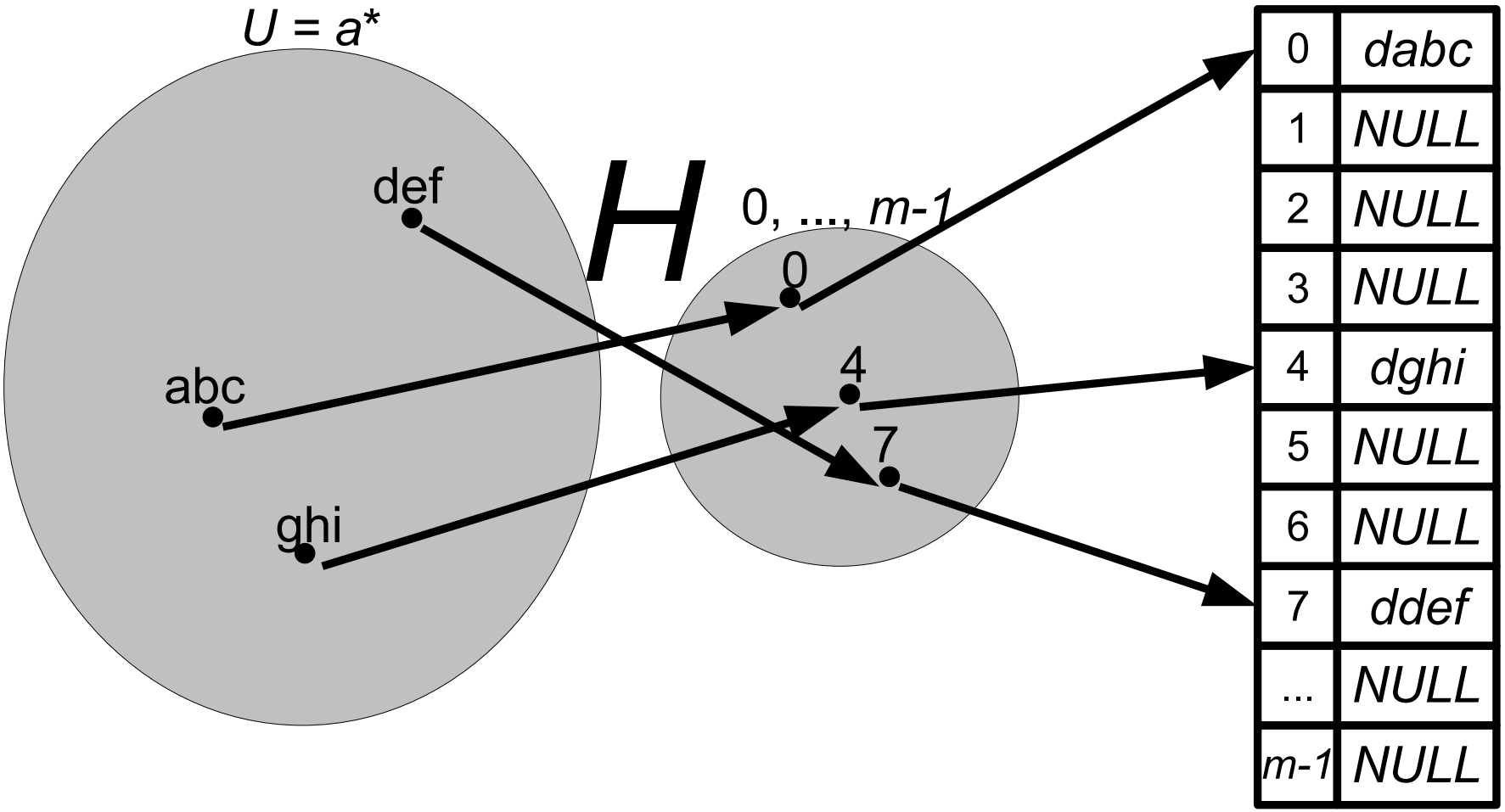
Adresowanie bezpośrednie



Funkcja haszująca

- co gdy uniwersum kluczy jest większe od dostępnej pamięci lub klucze nie są liczbami naturalnymi?
- w komputerze pamięć możemy adresować jedynie liczbami
- rozwiązanie: funkcja haszująca
- jest to funkcja która "przerabia" klucz na liczbę
- $H : klucz \rightarrow 0, \ldots m-1$
- np. suma lub iloczyn kodów znaków modulo m (marne, ale działa)
- np. suma wszystkich wartości liczbowych obiektu modulo m (j.w.)

Funkcja haszująca



Tablica z haszowaniem

- tablica m-elementowa T
- jeżeli element o kluczu k nie należy do tablicy, to $T[H(k)] = nullptr$
- wstawianie:
  `WSTAW(klucz, wart) { T[`H`(klucz)] = wart; }`
- usuwanie:
  `USUN(klucz) { T[`H`(klucz)] =` *nullptr*`; }`
- wstawianie:
  `ZNAJDZ(klucz) { return T[`H`(klucz)]; }`

Tablica z haszowaniem

0	*NULL*
1	*NULL*
2	*NULL*
3	*NULL*
4	*NULL*
5	*NULL*
6	*NULL*
7	*NULL*
...	*NULL*
m-1	*NULL*

Tablica z haszowaniem

T[abc] = dabc
H(abc) = 0
→ T[0]= dabc

0	*dabc*
1	*NULL*
2	*NULL*
3	*NULL*
4	*NULL*
5	*NULL*
6	*NULL*
7	*NULL*
...	*NULL*
m-1	*NULL*

Tablica z haszowaniem

T[abc] = dabc
H(abc) = 0
→ T[0]= dabc
T[def] = ddef
H(def) = 4
→ T[4]= ddef

0	*dabc*
1	*NULL*
2	*NULL*
3	*NULL*
4	*ddef*
5	*NULL*
6	*NULL*
7	*NULL*
...	*NULL*
m-1	*NULL*

Tablica z haszowaniem

T[abc] = dabc
H(abc) = 0
→ T[0]= dabc
T[def] = ddef
H(def) = 4
→ T[4]= ddef
T[def] = ?
H(def) = 4
T[4] → ddef

0	*dabc*
1	*NULL*
2	*NULL*
3	*NULL*
4	*ddef*
5	*NULL*
6	*NULL*
7	*NULL*
...	*NULL*
m-1	*NULL*

Tablica z haszowaniem

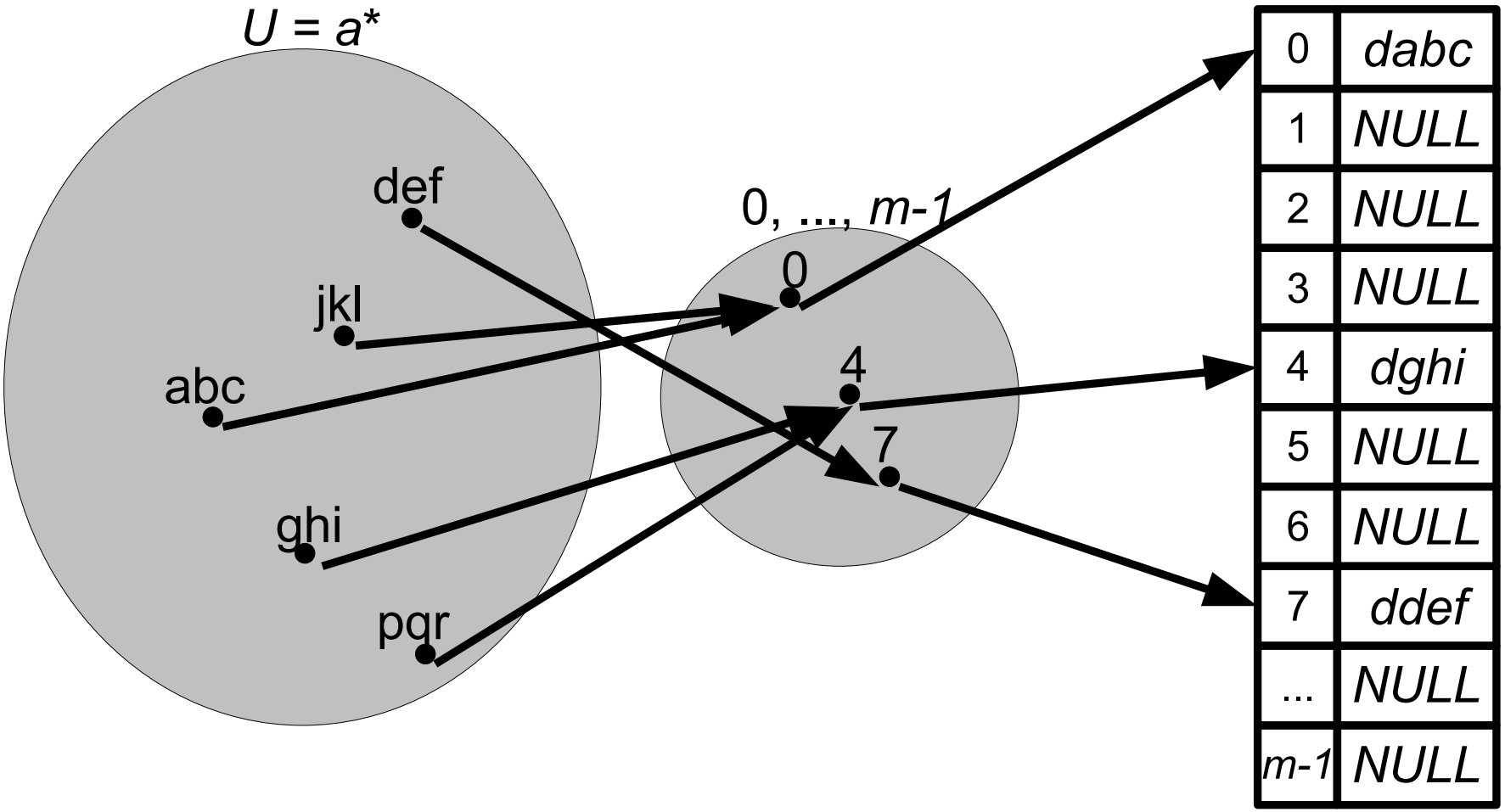
T[abc] = dabc
H(abc) = 0
→ T[0]= dabc
T[def] = ddef
H(def) = 4
→ T[4]= ddef
T[def] = ?
H(def) = 4
T[4] → ddef
T[ghi] = ?
H(ghi) = 7
T[7] → *NULL*

0	*dabc*
1	*NULL*
2	*NULL*
3	*NULL*
4	*ddef*
5	*NULL*
6	*NULL*
7	*NULL*
...	*NULL*
m-1	*NULL*

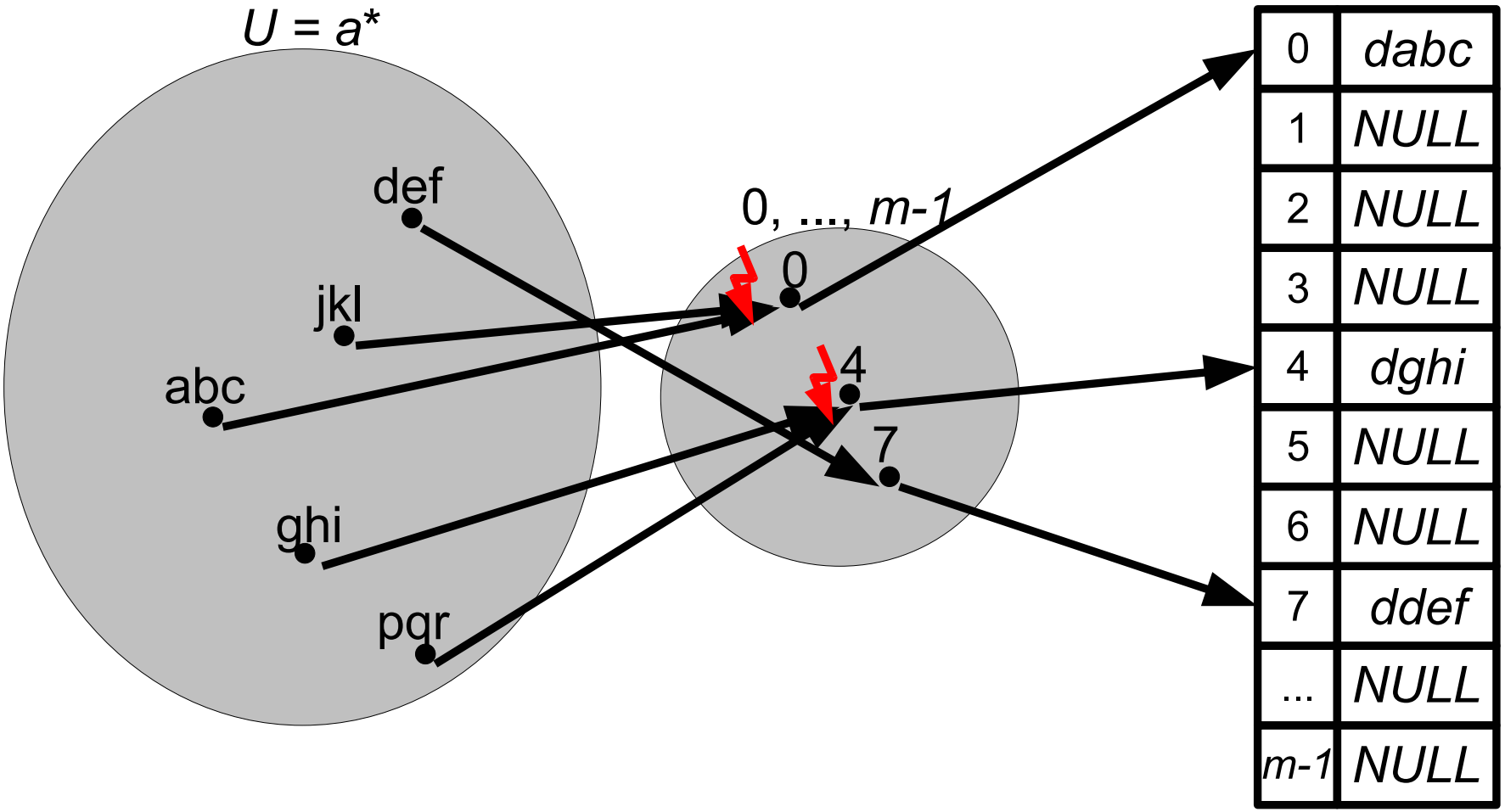
Właściwości funkcji haszującej

- powinna być prosta (szybka) do obliczenia
- dla tego samego klucza musi zwracać tą samą wartość (inaczej po wstawieniu mielibyśmy problem z ponownym znalezieniem elementu)
- rzeczywiste dane powinna jak najbardziej równomiernie rozkładać (rozpraszać) w przestrzeni adresów
- dla różnych kluczy powinna zwracać różne wartości (ale nie zawsze jest to możliwe — "dużą" przestrzeń kluczy mapujemy w "małą" przestrzeń adresów)
- kolizja — sytuacja, gdy dwa różne klucze mają taką samą wartość funkcji haszującej

Kolizje


U = a*
def
jkl
abc
ghi
pqr
0, ..., m-1
0
4
7
0 dabc
1 NULL
2 NULL
3 NULL
4 dghi
5 NULL
6 NULL
7 ddef
... NULL
m-1 NULL

Kolizje


U = a*
def
jkl
abc
ghi
pqr
0, ..., m-1
0
4
7
0
dabc
1
NULL
2
NULL
3
NULL
4
dghi
5
NULL
6
NULL
7
ddef
...
NULL
m-1
NULL

Rozwiązywanie kolizji

- w przypadku kolizji, w jednej komórce należy "upchnąć" więcej niż jedną daną
- razem z daną należy teraz przechowywać także klucz
  - w przypadku kolizji w jednej komórce mamy więcej niż jedną daną
  - szukamy tylko jednej z nich (według klucza), powinniśmy więc wiedzieć, pod jakim kluczem każda z nich jest przechowywana

Metody rozwiązywania kolizji

- łańcuchowanie oddzielne
- łańcuchowanie bezpośrednie
- łańcuchowanie z obszarem nadmiarowym
- adresowanie otwarte

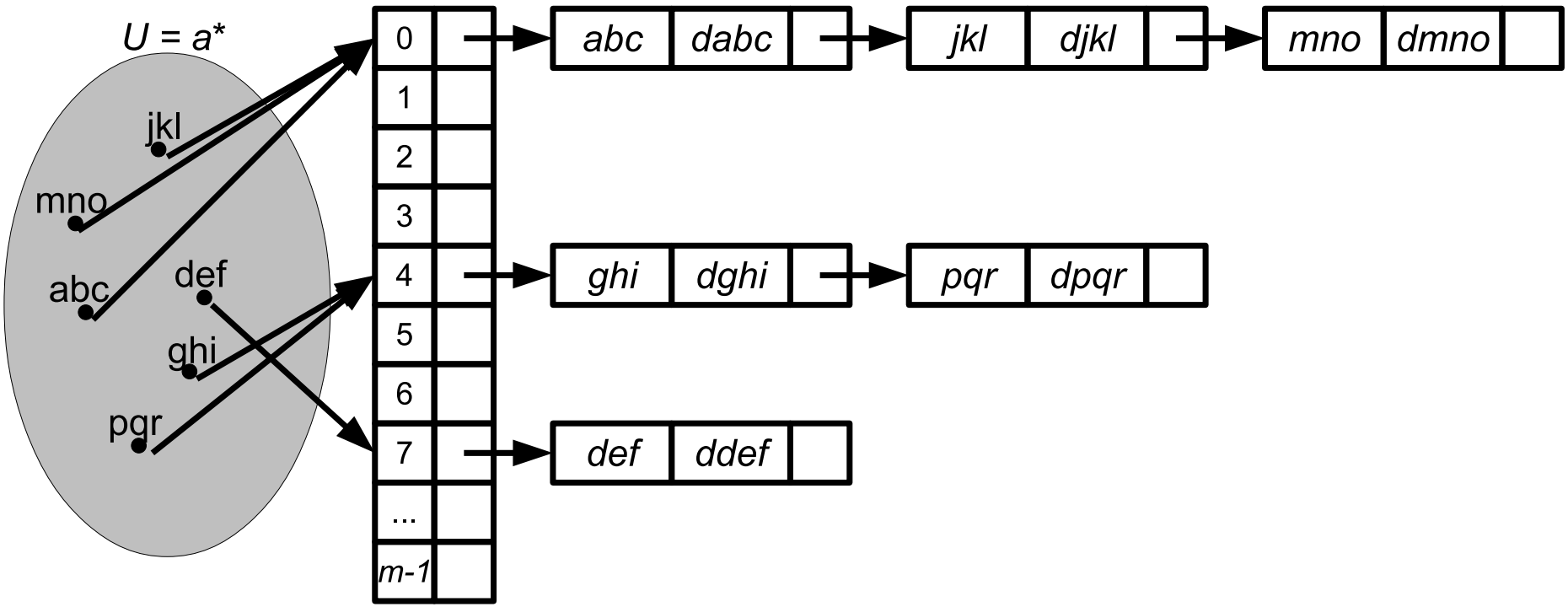
Łańcuchowanie oddzielne

- elementy kolidujące ze sobą przechowujemy w oddzielnych listach
- każda komórka tablicy T jest w rzeczywistości listą par $(klucz, wartosc)$
- dodając parę $(klucz, wartosc)$ dodajemy do listy $T(klucz)$ element $(klucz, wartosc)$
- usuwając element o kluczu *klucz* usuwamy z listy $T(klucz)$ element zawierający klucz *klucz*
- wyszukując element o kluczu *klucz* poszukujemy go pośród elementów na liście $T(klucz)$

Łańcuchowanie oddzielne

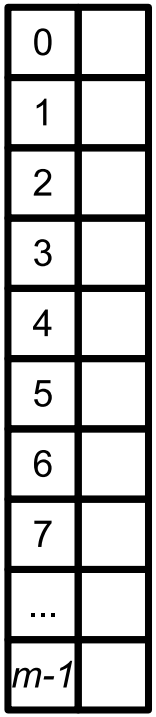
- oczekiwany czas wyszukiwania zakończonego porażką lub sukcesem: $O(1+\alpha)$
- α — współczynnik wypełnienia tablicy, $\alpha = n/m$, gdzie m to rozmiar tablicy, n to ilość elementów w tablicy
- średnia długość listy wynosi n/m
- w najgorszym przypadku będzie to jednak n

Łańcuchowanie oddzielne

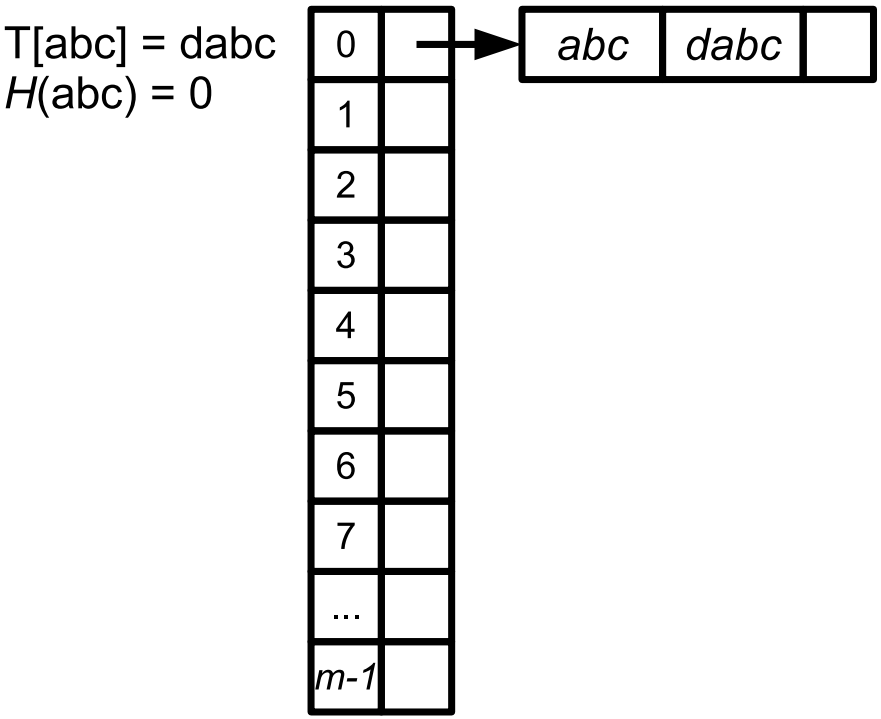


Łańcuchowanie oddzielne

T[abc] = dabc
H(abc) = 0

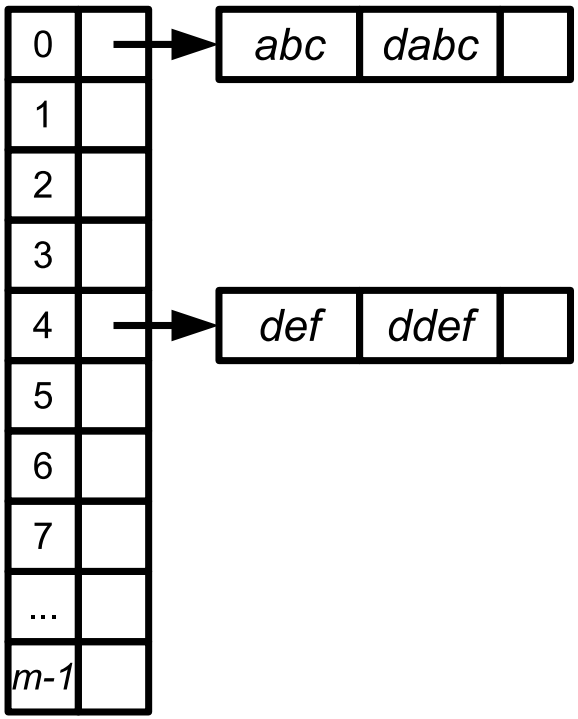


Łańcuchowanie oddzielne



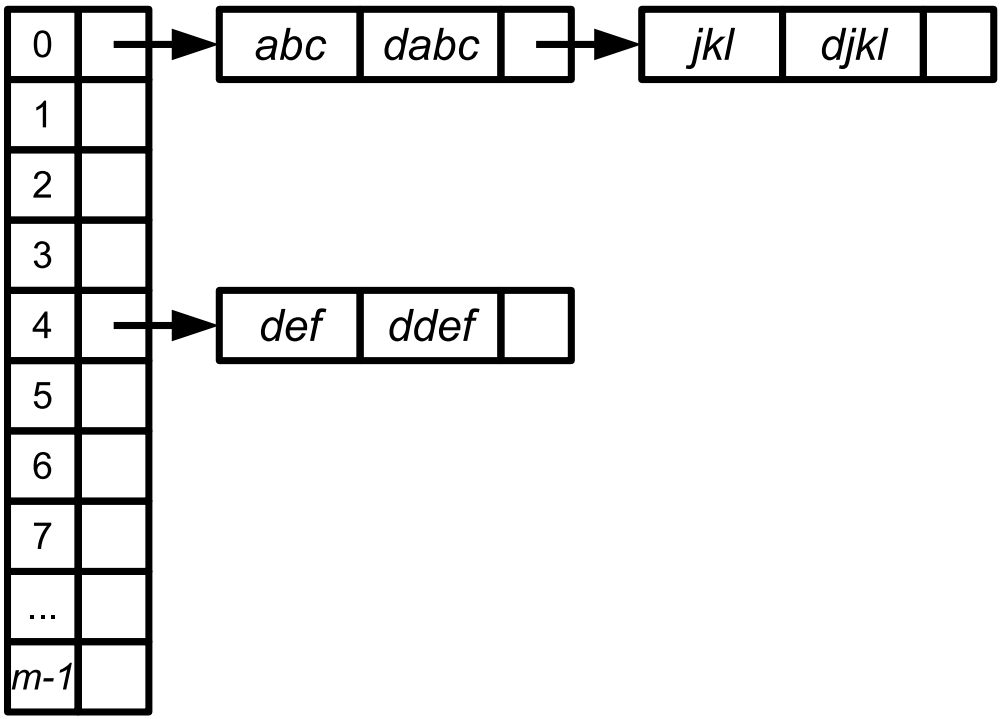
Łańcuchowanie oddzielne

T[abc] = dabc
H(abc) = 0
T[def] = ddef
H(def) = 4



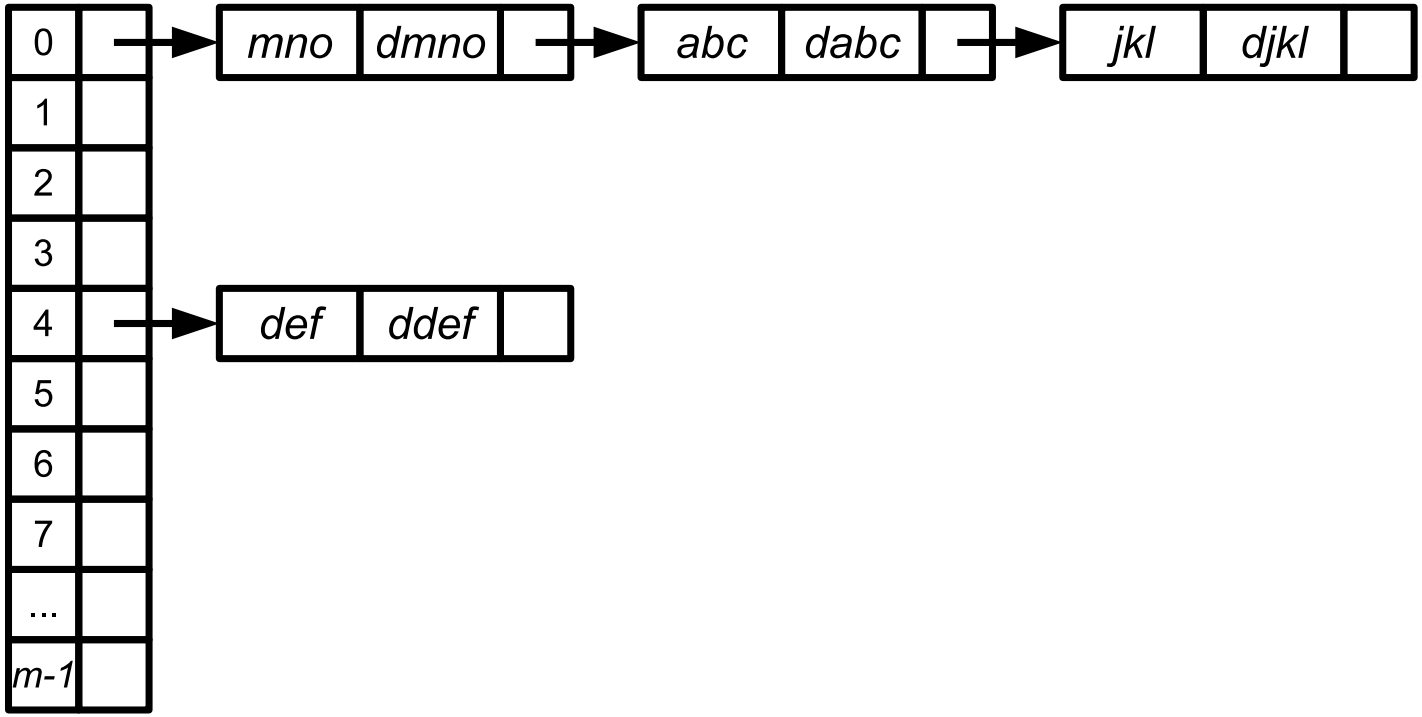
Łańcuchowanie oddzielne

T[abc] = dabc
H(abc) = 0
T[def] = ddef
H(def) = 4
T[jkl] = djkl
H(jkl) = 0

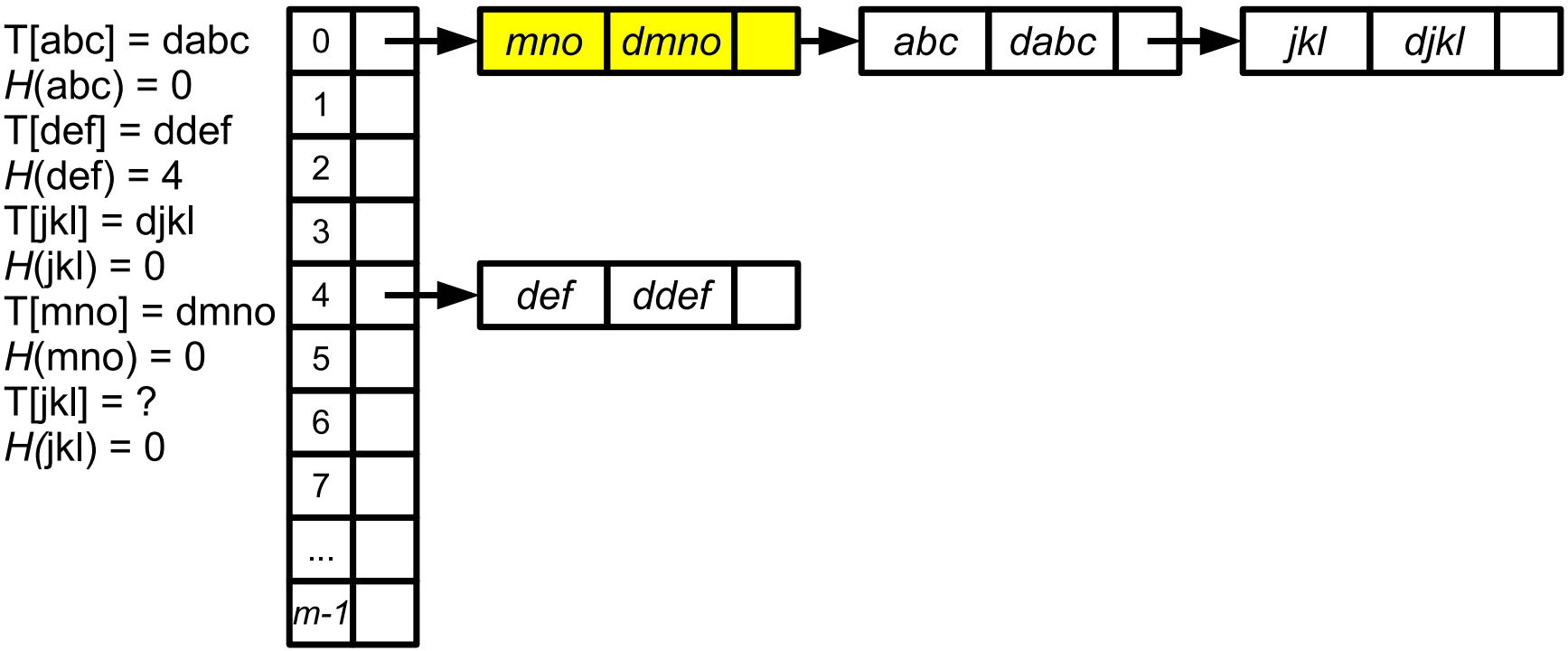


Łańcuchowanie oddzielne

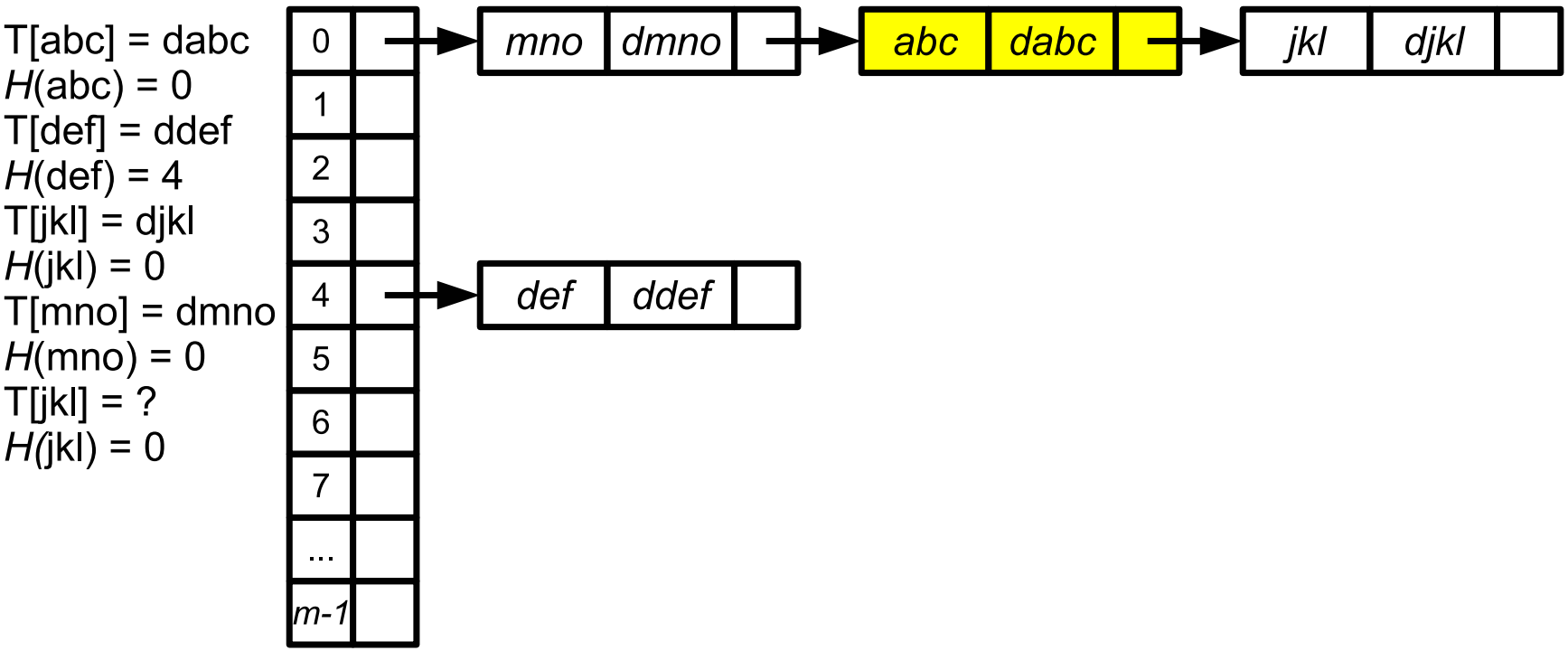
T[abc] = dabc
H(abc) = 0
T[def] = ddef
H(def) = 4
T[jkl] = djkl
H(jkl) = 0
T[mno] = dmno
H(mno) = 0



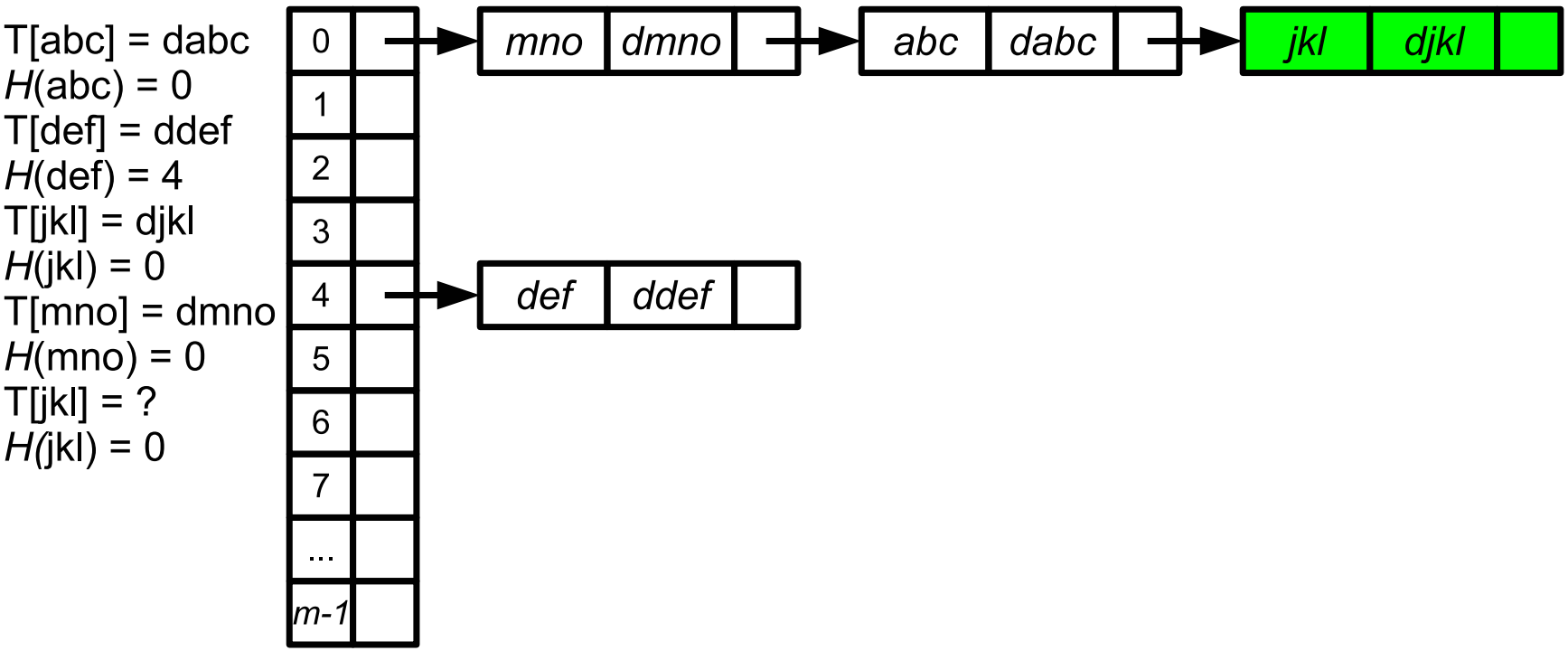
Łańcuchowanie oddzielne



Łańcuchowanie oddzielne



Łańcuchowanie oddzielne



Łańcuchowanie z obsz. nadmiarowym

- elementy pamiętamy bezpośrednio w tablicy
- każda komórka tablicy T poza kluczem i wartością, posiada także wskaźnik na następny element o takiej samej wartości funkcji haszującej
- miejsce na kolidujące elementy przydzielamy nie dynamicznie, ale z puli nazywanej obszarem nadmiarowym
- gdy miejsce w obszarze nadmiarowym wyczerpie się, możemy powiększyć całą tablicę, lub tylko "doalokować" nowy obszar nadmiarowy

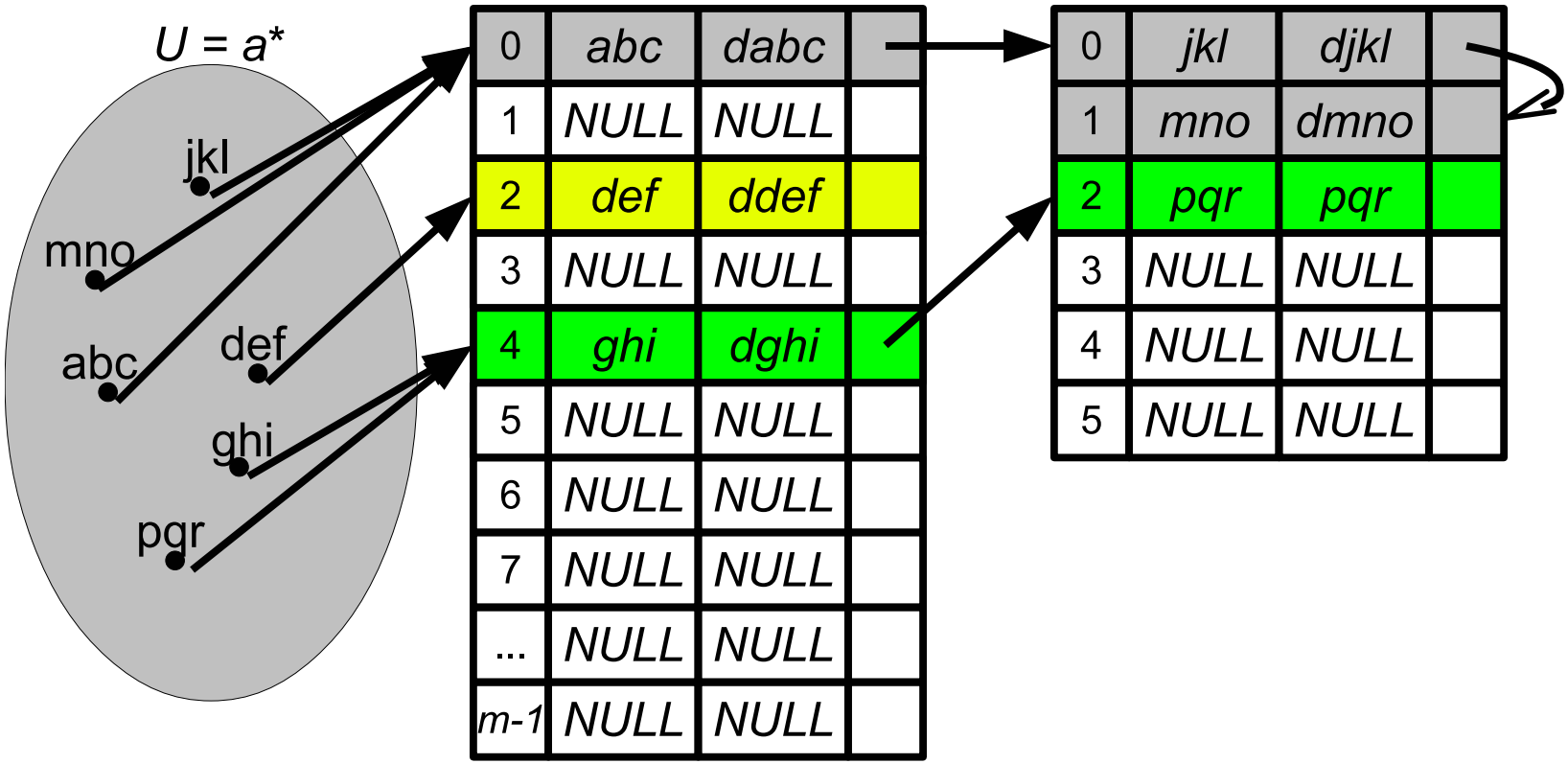
Łańcuchowanie z obsz. nadmiarowym

- wyszukując element o kluczu *klucz* :
  - rozpoczynamy od pola $T(klucz)$
  - przeglądamy kolejne pola (przechodząc po wskaźnikach) w poszukiwaniu tego, które zawiera klucz *klucz*
  - przerywamy gdy: znajdziemy poszukiwany klucz lub dojdziemy do końca listy

Łańcuchowanie z obsz. nadmiarowym

- dodając parę $(klucz, wartosc)$:
  - jeżeli komórka $T(klucz)$ jest pusta, wstawiamy tam parę $(klucz, wartosc)$
  - jeżeli komórka jest zajęta, bierzemy nową komórkę z obszaru nadmiarowego i wstawiamy tam parę $(klucz, wartosc)$
  - dołączamy tę nową komórkę do listy, której głową jest $T(klucz)$ (najłatwiej — za głową, ale można też na końcu)

Łańcuchowanie z obsz. nadmiarowym



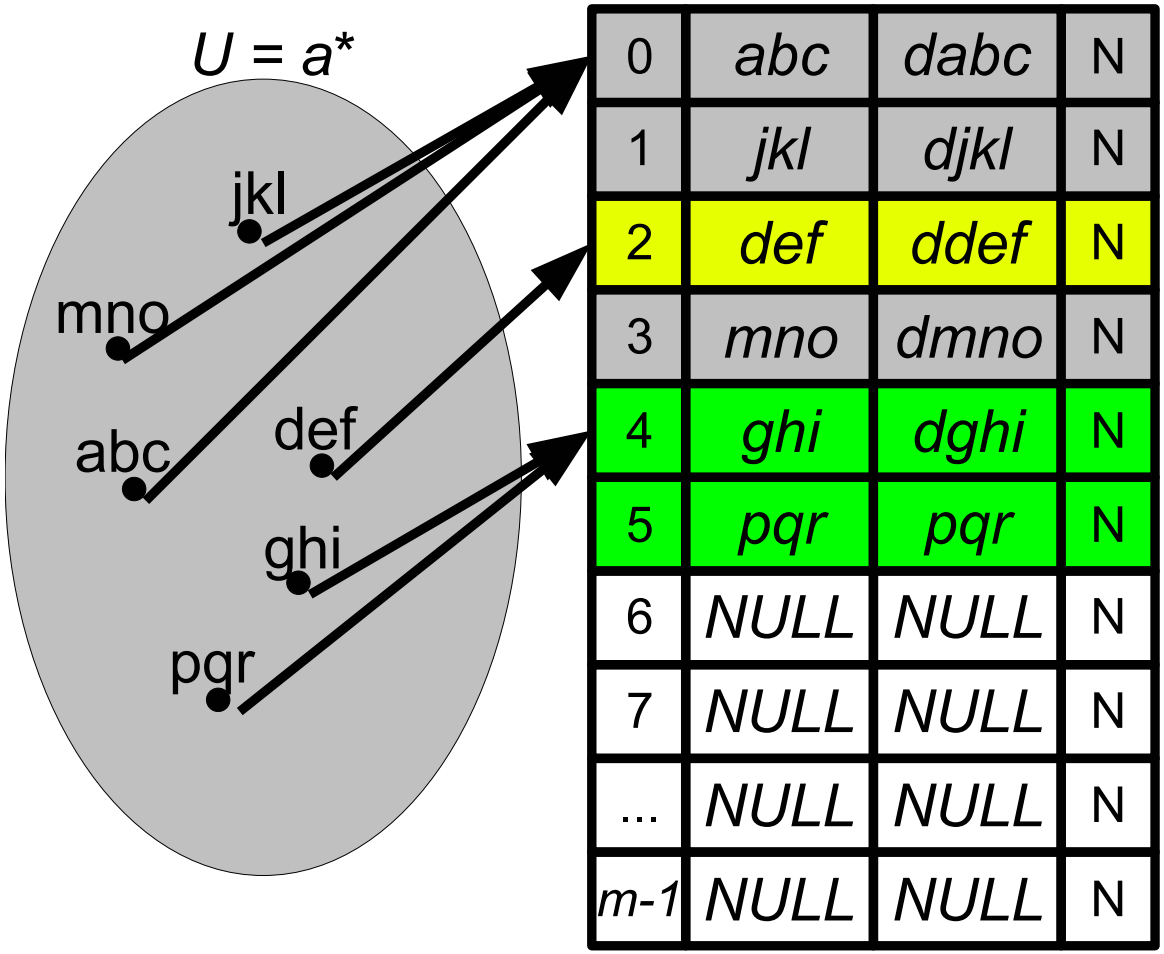
Adresowanie otwarte

- wszystkie elementy przechowywane są wprost w tablicy
- funkcja haszująca ma dwa parametry: klucz oraz numer próby
- funkcja haszująca wyznacza nam porządek przeszukiwania komórek tablicy
- ciąg $H(klucz, 0), H(klucz, 1), \ldots, H(klucz, m-1)$ powinien być permutacją ciągu $0, \ldots, m-1$ — inaczej nie wykorzystamy wszystkich komórek pamięci
- w tablicy przechowujemy parę $(klucz, wartosc)$ oraz flagę sygnalizującą, czy zawartość została usunięta

Adresowanie otwarte

- jeżeli podczas wstawiania komórka $T(klucz, 0)$ jest zajęta, próbujemy wstawić w komórkę $T(klucz, 1)$, $T(klucz, 2)$, itd.
- elementu szukamy w komórkach $T(klucz, 0)$, $T(klucz, 1)$, $T(klucz, 2)$, itd. aż dojdziemy do komórki pustej, której zawartość nie została usunięta
- usuwając element, wyszukujemy go, usuwamy i oznaczamy komórkę w której się znajdował jako usuniętą

Adresowanie otwarte



0	*abc*	*dabc*	N
1	*jkl*	*djkl*	N
2	*def*	*ddef*	N
3	*mno*	*dmno*	N
4	*ghi*	*dghi*	N
5	*pqr*	*pqr*	N
6	*NULL*	*NULL*	N
7	*NULL*	*NULL*	N
...	*NULL*	*NULL*	N
m-1	*NULL*	*NULL*	N

Adresowanie otwarte — wstawianie

```
WSTAW(T, klucz, wartosc)
 1: i = 0
 2: repeat
 3:         j = H(klucz, i)
 4:         if T[j] = nullptr then
 5:                 T[j].klucz = klucz
 6:                 T[j].wart = wartosc
 7:                 T[j].usuniete = Nie
 8:                 return j
 9:         else
10:                 i = i + 1
11:         end if
12: until i = m
```

Adresowanie otwarte — wyszukiwanie

```
ZNAJDZ(T, klucz)
 1: i = 0
 2: repeat
 3:         j = H(klucz, i)
 4:         if T[j].klucz = klucz then
 5:                 return T[j].wart
 6:         end if
 7:         i = i + 1
 8: until i = m lub
           (T[j].klucz = nullptr i T[j].usuniete = Nie )
```

Adresowanie otwarte — usuwanie

```
USUN(T, klucz)
 1: i = 0
 2: repeat
 3:         j = H(klucz, i)
 4:         if T[j].klucz = klucz then
 5:                 T[j].klucz = nullptr
 6:                 T[j].wart = nullptr
 7:                 T[j].usuniete = Tak
 8:                 return
 9:         end if
10:         i = i + 1
11: until i = m lub
          (T[j].klucz = nullptr i T[j].usuniete = Nie )
```

Adresowanie liniowe

- adresowanie liniowe:
  $H(klucz, i) = (H_1(klucz) + i) \mod m$
  - proste w implementacji
  - wadą jest grupowanie elementów: długie spójne ciągi szybko się powiększają
  - jeżeli w tablicy $\alpha = 0.5$ i zajęte są tylko nieparzyste komórki potrzeba średnio 1.5 porównań aby stwierdzić, że elementu nie ma w tablicy
  - jeżeli w tablicy $\alpha = 0.5$ i zajętych jest pierwszych $m/2$ komórek średnia liczba porównań jest równa $m/4$
  - w adresowaniu liniowym rozkład elementów częściej przypomina ten drugi

Adresowanie kwadratowe

- adresowanie kwadratowe:
  $H(klucz, i) = (H_1(klucz) + a_1 i + a_2 i^2) \mod m$
- adresowanie sześcienne:
  $H(klucz, i) = (H_1(klucz) + a_1 i + a_2 i^2 + a_3 i^3) \mod m$
  - wymaga odpowiedniego doboru wartości $a_1, a_2[, a_3]$
  - wadą jest (mniej groźne) grupowanie wtórne — dla dwóch kluczy o takiej samej wartości H_1 będziemy przeszukiwać komórki w takiej samej kolejności

Adresowanie kwadratowe

- w adresowaniu kwadratowym, gdy $m = 2^k$, dobrym kandydatem na a_1 i a_2 jest liczba $\frac{1}{2}$
- odwiedzamy wtedy komórki $H_1(klucz)$, $H_1(klucz) + 1$, $H_1(klucz) + 1 + 2$, $H_1(klucz) + 1 + 2 + 3$ itd. — po każdej próbie, do indeksu komórki dodajemy numer próby
- można pokazać, że tym sposobem odwiedzimy wszystkie komórki tablicy

Adresowanie dwukrotne

- adresowanie dwukrotne:
  $H(klucz, i) = (H_1(klucz) + iH_2(klucz)) \mod m$
  - wymaga obliczenia wartości dwóch funkcji haszujących (oczywiście H_1 i H_2 powinny być różnymi funkcjami)
  - H_2 powinna być odpowiednio dobrana aby mieć pewność, że zostanie przeszukana cała tablica
    - np. zapewniając, że $H_2(klucz)$ jest względnie pierwsze z rozmiarem tablicy (inaczej przeszukamy tylko $m/NWD(m, H_2(klucz))$ komórek)
    - np. m — pierwsze, $H_2(klucz) < m$
    - np. m — parzyste, $H_2(klucz)$ — zawsze nieparzyste

Adresowanie dwukrotne

- adresowanie dwukrotne:
  - grupowanie wtórne jest znikome
  - dla dwóch różnych kluczy o takiej samej wartości H_1 ciągi przeszukiwanych elementów zazwyczaj będą różne (aby były równe kolizja musiałaby wystąpić przy obu funkcjach)

Adresowanie otwarte

- przy równomiernym ("idealnym") haszowaniu oczekiwana liczba porównań kluczy:
  - w czasie wyszukiwania elementu nie występującego w tablicy (oraz w czasie wstawiania) jest nie większa niż $1/(1-\alpha)$
  - w czasie wyszukiwania elementu występującego w tablicy (oraz w czasie usuwania) jest nie większa niż $\frac{1}{\alpha} \ln \frac{1}{1-\alpha} + \frac{1}{\alpha}$
  - dla $\alpha = 0.5$ jest to 2 i ≈ 3.387
  - dla $\alpha = 0.9$ jest to 10 i ≈ 3.670

Adresowanie otwarte

- w przypadku adresowania liniowego:
  - w czasie wyszukiwania elementu nie występującego w tablicy: $1 + \frac{1}{(1-\alpha)^2}$
  - w czasie wyszukiwania elementu występującego w tablicy: $1 + \frac{1}{1-\alpha}$
  - dla $\alpha = 0.5$ jest to 4 i 3
  - dla $\alpha = 0.9$ jest to 100 i 11

Adresowanie otwarte

- aby czas dostępu był krótki należy utrzymywać wartość współczynnika wypełnienia poniżej określonego progu (zazwyczaj 0.5–0.8)
- jeżeli próg zostanie przekroczony, należy powiększyć tablicę
- wiąże się z ponownym wstawieniem wszystkich przechowywanych elementów (zmienia się m więc zmieniają się także wartości funkcji haszującej dla kluczy) — kosztuje to $O(n)$ (lub $O(m)$ w zależności od implementacji)

Adresowanie otwarte

- aby zachować średni czas wstawiania $O(1)$ należy tablicę powiększać wykładniczo (np. każde powiększenie podwaja rozmiar tablicy)
- spadek współczynnika wypełnienia poniżej pewnego progu (np. 0.1 lub 0.25) może być sygnałem do pomniejszenia rozmiaru tablicy (np. o połowę – aby usuwanie działało średnio w stałym czasie)

Wybór funkcji mieszającej

- przestrzeń wartości powinna być nie mniejsza niż rozmiar tablicy
  - inaczej część tablicy pozostanie niewykorzystana
- każdy legalny klucz powinien być odwzorowany w poprawny adres w tablicy T (tj. w liczbę z zakresu $0, \ldots, m-1$)

Wybór funkcji mieszającej

- dobra funkcja powinna spełniać (w przybliżeniu) założenie prostego równomiernego haszowania: losowo wybrany klucz jest z jednakowym prawdopodobieństwem odwzorowywany na każdą z pozycji
- klucze nieliczbowe należy najpierw zamienić w pewien sposób na liczbę
- po skonstruowaniu funkcji dobrze jest przetestować ją na danych rzeczywistych (zbadać, jak rozrzuca elementy)

Haszowanie modularne

- $H(klucz) = klucz \mod m$
- działa szybko (jedno dzielenie)
- m nie powinno być potęgą dwójki (wtedy bierzemy najmłodsze m bitów) – jeżeli nie mamy gwarancji, że ich rozkład jest równomierny, lepiej wybrać funkcję która zależy od wszystkich bitów klucza
- m nie powinno być postaci $2^p - 1$ — wartość funkcji haszujących różniących się kolejnością dwóch sąsiednich cyfr binarnych będzie taka sama
- podobnie dla innych podstaw (np. 10, czyli $m \neq 10^p$, $m \neq 10^p - 1$)
- dobrymi wartościami m są liczby pierwsze niezbyt bliskie potęgom 2

Haszowanie przez mnożenie

- mnożymy wartość liczbową klucza przez stałą A z przedziału $0 < A < 1$ i wyznaczamy część ułamkową
- wynik mnożymy przez m i z otrzymanej wartości bierzemy podłogę
- $H(klucz) = podloga(m(kA \mod 1))$
- m może w tym przypadku być dowolne
- jednak często wybiera się m o postaci 2^p — ułatwia to implementację (mnożymy k przez $podloga(A \cdot 2^w)$ i z wyniku wybieramy bity na pozycjach $w - p, \ldots, w - 1$, gdzie w to liczba bitów słowa maszynowego)

Haszowanie przez mnożenie

- metoda działa dla dowolnej wartości A, ale pewne wartości dają lepsze rozproszenie niż inne
- szczególnie dobra jest wartość $A = \frac{\sqrt{(5)}-1}{2} = 0.6180339887\ldots$
- np. dla $m = 10000$:

$$\begin{aligned} H(123456) &= \lfloor 10000 \cdot (123456 \cdot 0.61803\ldots \mod 1) \rfloor \\ &= \lfloor 10000 \cdot (76300.0041151\ldots \mod 1) \rfloor \\ &= \lfloor 10000 \cdot 0.0041151 \rfloor \\ &= \lfloor 41.151 \rfloor \\ &= 41 \end{aligned}$$

Haszowanie przez randomizację

- obliczamy $H(klucz) = Rand(klucz)$, gdzie $Rand$ to pewna funkcja pseudolosowa
- np. wybieramy z klucza pewną liczbę bitów, traktujemy je jako liczbę binarną i podnosimy do kwadratu
- np. ciąg bitów klucza dzielimy na kilka fragmentów, traktujemy je jako liczby binarne i sumujemy je
- np. ciąg bitów klucza dzielimy na kilka fragmentów, traktujemy je jako liczby binarne i sumujemy je modulo 2
- zazwyczaj należy połączyć tę metodę z haszowaniem modularnym (możemy otrzymać duże liczby)

Haszowanie uniwersalne

- jeżeli dane do tablicy wstawiane są przez "złośliwego przeciwnika", nie możemy wykluczyć sytuacji, gdy wszystkie klucze trafią w tę samą komórkę tablicy
- średni czas wykonywania operacji wynosi wówczas $O(n)$
- każda ustalona funkcja haszująca jest podatna na takie zagrożenie
- rozwiązaniem jest losowy wybór funkcji haszującej
- oczywiście wszystkie możliwe do wylosowania funkcje powinny odwzorowywać klucze w ten sam zakres adresów $(0, \ldots, m-1)$

Wybór funkcji mieszającej

- sumowanie kodów znaków jest złe:
  - kody liter: $97, \ldots, 122$ (małe), $65, \ldots, 90$ (duże)
  - znaki o kodach poniżej 32 prawie nigdy nie występują w napisach
  - małe wartości: 30 znaków da maksymalnie $30 \cdot 256 = 7680$, zdecydowanie częściej jednak tylko $30 \cdot 122 = 3660$
- iloczyn kodów znaków (modulo m) jest zły:
  - jeżeli m jest parzyste — katastrofa
  - częstsze kolizje na indeksach będących dzielnikami m

Wybór funkcji mieszającej

- konstruując funkcję należy pamiętać o właściwościach operacji matematycznych, np.:
  - operacje takie jak suma, iloczyn, suma modulo 2 są przemienne — zamiana kolejności znaków nie zmienia wartości funkcji haszującej
  - stosując te funkcje dobrze jest włączyć do wartości znaku jego pozycję
  - np. suma znaków pomnożonych przez wagę kolejnej pozycji, gdzie wagi pozycji to np. $137, 11, 113, 37, 19, \ldots$
  - iloczyn dwóch liczb parzystych, iloczyn liczby parzystej i nieparzystej daje liczbę parzystą; liczbę nieparzystą daje tylko iloczyn dwóch liczb nieparzystych
  - ...

Przykład

- tablica o rozmiarze 10^6 elementów
- kluczem są nazwiska
- wielkość liter nie ma znaczenia — będziemy używać tylko małych liter
- jeżeli spodziewamy się, że polskie znaki będą występować rzadko — możemy je pominąć lub zamienić na łacińskie odpowiedniki
- małe litery: $97, \ldots, 122$
- $H(klucz) = \sum_{i=0}^{n-1}(klucz[i] - 97) * 26^{(i \mod 6)}$

Przykład

- w przypadku długich napisów, warto rozważyć haszowanie tylko ich początkowych fragmentów
- prawdopodobieństwo, że dwa napisy będą identyczne na pierwszych 20–60 znakach jest znikome (o ile dane nie mają specyficznego formatu)
- w takich sytuacjach warto raczej zgodzić się na kolizję, niż próbować "na siłę" rozróżnić wartości funkcji haszującej — kosztem jednego czy dwóch porównań, raz na jakiś czas, zyskujemy przyspieszenie obliczania funkcji haszującej przy każdym dostępie

Przykład — djb2

1: **uint32_t** $hash = 5381$
2: **for** $i = 1$ do n **do**
3: $\quad hash = ((hash << 5) + hash) + bajt[i]$
4: **end for**
5: **return** $hash$

Przykład — sdbm

1: **uint32_t** $hash = 0$
2: **for** $i = 1$ do n **do**
3: $\quad hash = bajt[i] + ((hash << 6) + (hash << 16)) - hash$
4: **end for**
5: **return** $hash$

Przykład — Jenkin's one-at-a-time

1: **uint32_t** $hash = 0$
2: **for** $i = 1$ do n **do**
3: $\quad hash = hash + bajt[i]$
4: $\quad hash = hash + (hash << 10)$
5: $\quad hash = hash$ **xor** $(hash >> 6)$
6: **end for**
7: $hash = hash + (hash << 3)$
8: $hash = hash$ **xor** $(hash >> 11)$
9: $hash = hash + (hash << 15)$
10: **return** $hash$

Przykład — FNV-1a

1: **uint32_t** $hash = 2166136261$
2: **for** $i = 1$ do n **do**
3: $\quad hash = hash$ **xor** $bajt[i]$
4: $\quad hash = hash * 16777619$
5: **end for**
6: **return** $hash$

Przykład — ELF hash

```
1:  uint32_t hash = 0, high
2:  for i = 1 do n do
3:          hash = (hash << 4) + bajt[i]
4:          high = hash and 0xF0000000
5:          if high ≠ 0 then
6:                  hash = hash xor (high >> 24)
7:          end if
8:          hash = hash and not high
9:  end for
10: return hash
```

Przeglądanie całej tablicy

- tablice haszowane nie umożliwiają efektywnego przeglądania wszystkich elementów z tablicy
- "oczywiste" rozwiązanie:

```
for(i = 0; i < m; i++) {
    if(T[i] ≠ nullptr) wypisz(T[i]);
    };
```

- co jeżeli elementów jest mało a tablica jest duża? — dużo niepotrzebnych kroków
- rozwiązanie: powiązanie elementów listą (dwukierunkową)
  - dodatkowy narzut na dodawanie i usuwanie elementu to $O(1)$

Przeglądanie całej tablicy