Zbiory dynamiczne

Operacje na zbiorze S:

- `Search(`S, k`)` – zwraca wskaźnik do $x \in S$ takiego, że *key*[x]$= k$
- `Insert(`S, x`)` – dodaje element wskazywany przez x do S
- `Delete(`S, x`)` – usuwa z S element wskazywany przez x.
- `Minimum(`S`)` – zwraca el. o najmniejszym kluczu.
- `Maximum(`S`)` – zwraca el. o największym kluczu.
- `Successor(`S, x`)` – następnik x w S.
- `Predecessor(`S, x`)` – poprzednik x w S.

stos

$S[1 \ldots n]$ – tablica, *top*$[S]$ – szczyt stosu

```
Stack-Empty(S)
1 if top[S]=0
2   then return True
3   else return False
```

stos

$S[1 \dots n]$ – tablica, *top*$[S]$ – szczyt stosu

```
Stack-Empty(S)
if top[S]=0
  then return True
  else return False
```

```
Push(S,x)    ▷ brak obsługi przepełnienia
top[S]←top[S]+1
S[top[S]] ← x
```

stos

$S[1 \ldots n]$ – tablica, *top*$[S]$ – szczyt stosu

```
Stack-Empty(S)
1 if top[S]=0
2   then return True
3   else return False
```

```
Push(S,x)    ▷ brak obsługi przepełnienia
1 top[S]←top[S]+1
2 S[top[S]] ← x
```

```
Pop(S)
1 if Stack-Empty(S) then
2   error “niedomiar”
  else
3   top[S]←top[S]−1
4   return S[top[S]+1]
```

kolejka (cykliczna)

$Q[1 \ldots n]$ – tablica, *head*$[Q]$, *tail*$[Q]$

head$[Q]$=*tail*$[Q]$ – kolejka pusta

head$[Q]$$\equiv$*tail*$[Q]$$+1 \pmod{n}$ – kolejka pełna

```
Enqueue(Q,x)
1 Q[tail[Q]] ← x
2 if tail[Q]=length[Q]
3    then tail[Q] ← 1
4    else tail[Q] ← tail[Q] + 1
```

kolejka (cykliczna)

$Q[1\ldots n]$ – tablica, *head*$[Q]$, *tail*$[Q]$
head$[Q]$=*tail*$[Q]$ – kolejka pusta
head$[Q] \equiv$ *tail*$[Q]+1 \pmod{n}$ – kolejka pełna

Enqueue(Q,x)
$Q[$*tail*$[Q]] \leftarrow x$
if *tail*$[Q]$=*length*$[Q]$
    then *tail*$[Q] \leftarrow 1$
    else *tail*$[Q] \leftarrow$ *tail*$[Q]+1$

Dequeue(Q)
$x \leftarrow Q[$*head*$[Q]]$
if *head*$[Q] =$ *length*$[Q]$
    then *head*$[Q] \leftarrow 1$
    else *head*$[Q] \leftarrow$ *head*$[Q]+1$
return x

Uwaga: pominięto przypadki niedomiaru i przepełnienia

lista

lista L, *head*$[L]$ (*head*$[L]$=NIL – pusta)

element x o atrybutach: *next*$[x]$, *prev*$[x]$, *key*$[x]$

```
List-Search(L, k)
  x ← head[L]
  while x ≠ NIL and key[x] ≠ k do
      x ← next[x]
  return x      ▷ czas:  Θ(n)
```

lista

lista L, *head*$[L]$ (*head*$[L]$=NIL – pusta)
element x o atrybutach: *next*$[x]$, *prev*$[x]$, *key*$[x]$

```
List-Search(L, k)
1 x ← head[L]
2 while x ≠ NIL and key[x] ≠ k do
3     x ← next[x]
4 return x       ▷ czas: Θ(n)
```

```
List-Insert(L, x)
1 next[x] ← head[L]
2 if head[L] ≠ NIL then
3     prev[head[L]] ← x
4 head[L] ← x
5 prev[x] ← NIL     ▷ czas: Θ(1)
```

lista

```
List-Delete(L, x)
1 if prev[x] ≠ NIL
2    then next[prev[x]] ← next[x]
3    else head[L] ← next[x]
4 if next[x] ≠ NIL
5    then prev[next[x]] ← prev[x]
```

Czas: $O(1)$. (Ale znalezienie elementu o danym kluczu wymaga: $\Theta(n)$).

lista z wartownikiem

wartownik – el. wsk. przez *nil*$[L]$. (zastępuje NIL).

head$[L]$ – zastąpiony przez *next*[*nil*$[L]$]

`List-Delete'(`L`,`x`)`

1 *next*[*prev*$[x]$]$\leftarrow$*next*$[x]$

2 *prev*[*next*$[x]$]$\leftarrow$*prev*$[x]$

lista z wartownikiem

wartownik – el. wsk. przez *nil*$[L]$. (zastępuje NIL).

head$[L]$ – zastąpiony przez *next*[*nil*$[L]$]

```
List-Delete'(L,x)
1  next[prev[x]]←next[x]
2  prev[next[x]]←prev[x]
```

```
List-Insert'(L,x)
1  next[x] ←next[nil[L]]
2  prev[next[nil[L]]]← x
3  next[nil[L]]← x
4  prev[x] ←nil[L]
```

lista z wartownikiem

wartownik – el. wsk. przez *nil*$[L]$. (zastępuje NIL).

head$[L]$ – zastąpiony przez *next*[*nil*$[L]$]

List-Delete'(L,x)

next[*prev*$[x]$]$\leftarrow$*next*$[x]$
prev[*next*$[x]$]$\leftarrow$*prev*$[x]$

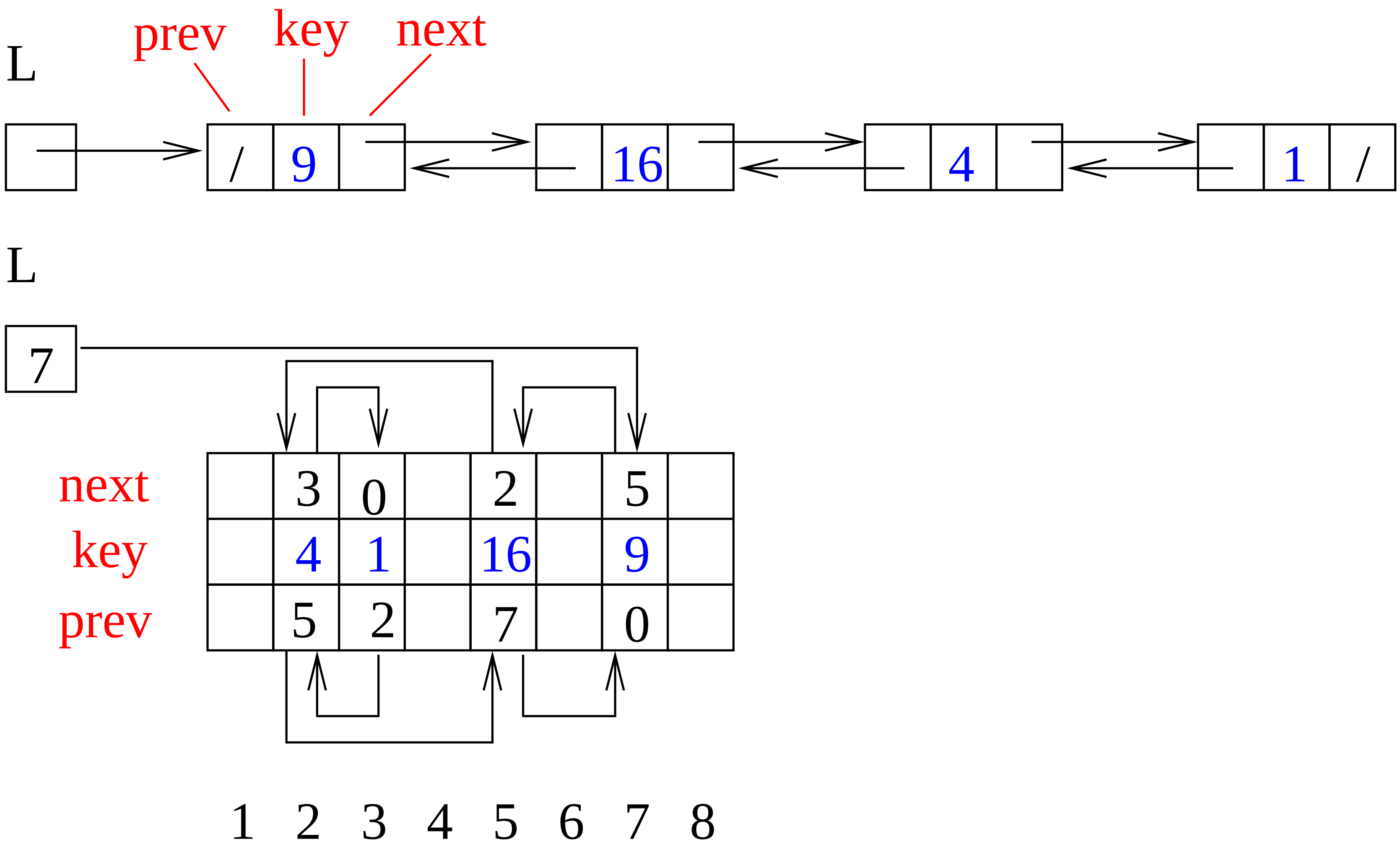
List-Insert'(L,x)

next$[x]$ $\leftarrow$*next*[*nil*$[L]$]
prev[*next*[*nil*$[L]$]]$\leftarrow x$
next[*nil*$[L]$]$\leftarrow x$
prev$[x]$ $\leftarrow$*nil*$[L]$

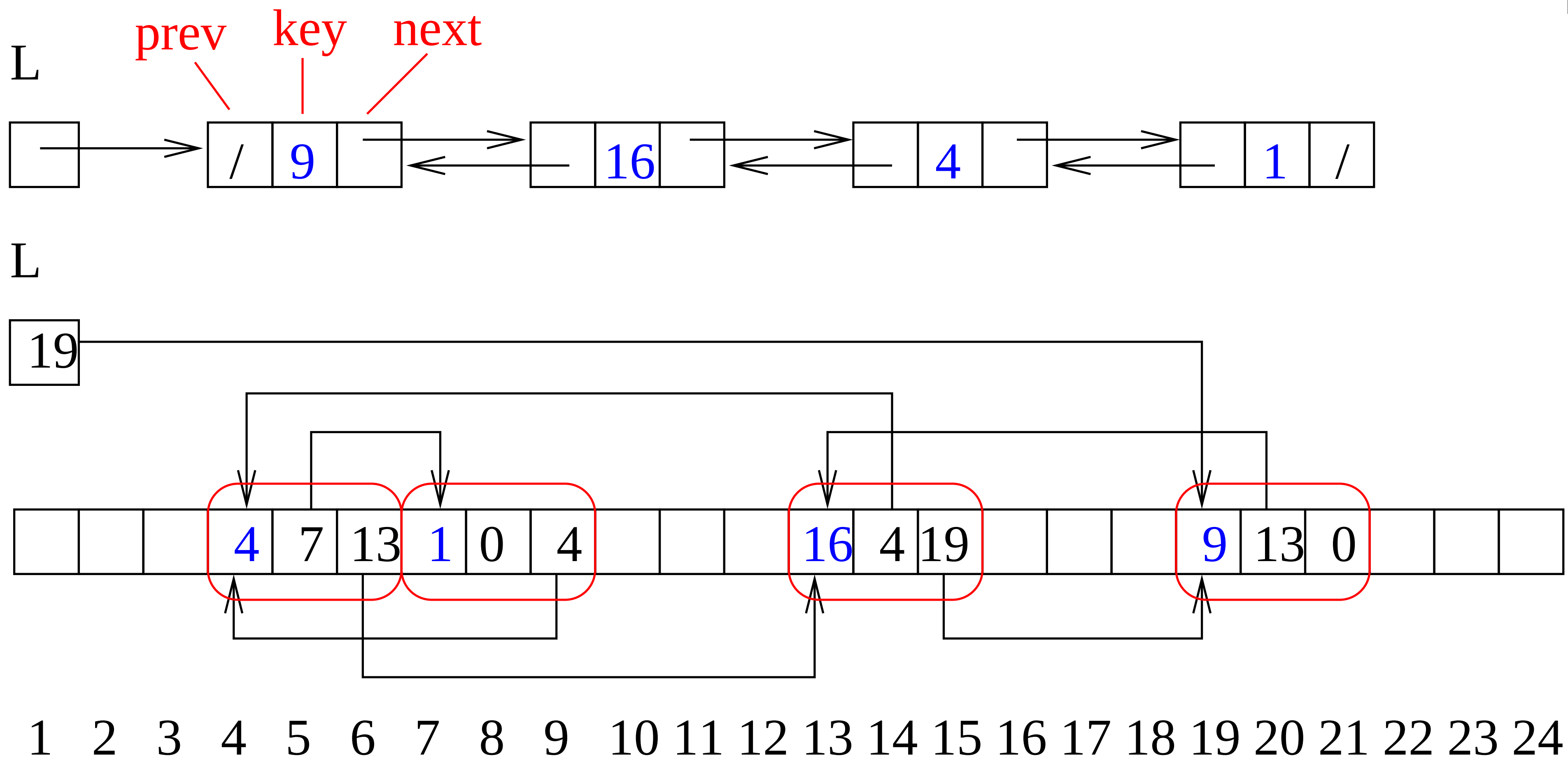
List-Search'(L,k)

$x \leftarrow$*next*[*nil*$[L]$]
while $x \neq$*nil*$[L]$ **and** *key*$[x] \neq k$ **do**
    $x \leftarrow$*next*$[x]$
return x

Reprezentacja wielotablicowa

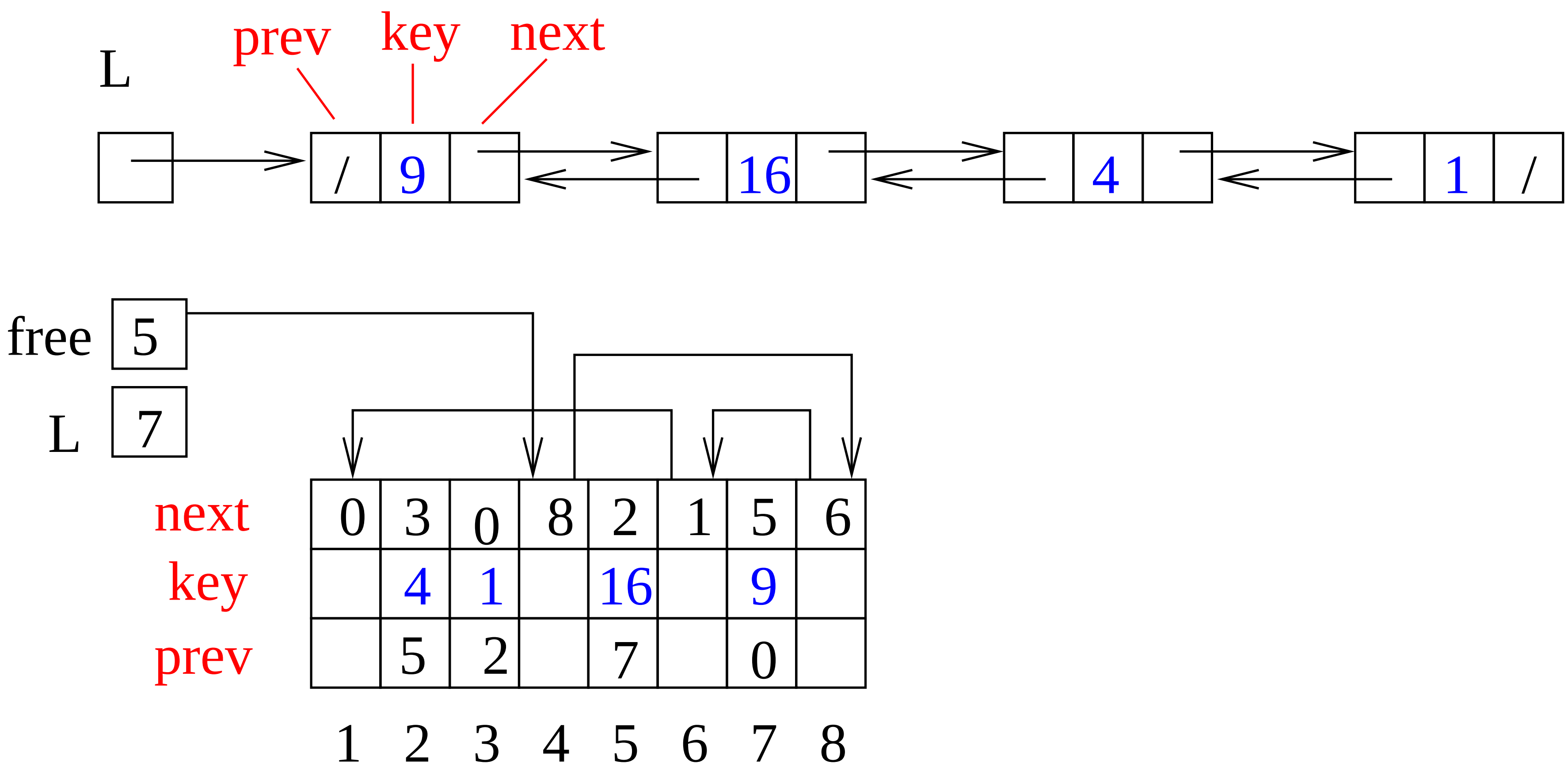


Reprezentacja jednotablicowa



Przydzielanie i zwalnianie pamięci

Wolne pozycje połączone w listę *free* przy pomocy pola *next*.



Przydzielanie i zwalnianie pamięci

`Allocate-Object()`

`if` *free* $= NIL$ `then`

    `error “brak pamięci”`

`else`

    $x \leftarrow$ *free*

    free $\leftarrow$ *next*$[x]$

    `return` x

Czas: $O(1)$.

`Free-Object(`x`)`

next$[x] \leftarrow$ *free*

free $\leftarrow x$

Czas: $O(1)$.

Uwaga: Jedna lista wolnych pozycji może obsługiwać wiele list (o elementach tego samego typu).

Tablice z adresowaniem bezpośrednim

- $U = \{0, 1, \ldots, m-1\}$ – uniwersum (zbiór możliwych wartości kluczy), m – małe
- $T[0 \ldots m-1]$ – tablica
- $T[k] = NIL$ – brak rekordu o kluczu k w zbiorze

Direct-Address-Search(T,k)
  return $T[k]$

Direct-Address-Insert(T,x)
  $T[\textit{key}[x]] \leftarrow x$

Direct-Address-Delete(T,x)
  $T[\textit{key}[x]] \leftarrow NIL$

Tablice z haszowaniem

- U – duże uniwersum
- K – zbiór kluczy przechowywanych w słowniku
  $|K| < |U|$
- $T[0 \ldots m-1]$ – tablica, $m = \Theta(K)$,
- $h : U \rightarrow \{0 \ldots m-1\}$ – *funkcja haszująca*, $h(k)$ – pozycja elementu o kluczu k w tablicy T
- problem: kolizje (h nie jest różnowartościowa)

Metoda łańcuchowa

Element x wstawiany na listę $T[h(\textit{key}[x])]$.

```
Chained-Hash-Insert(T,x)
  wstaw x na początek listy T[h(key[x])]

Chained-Hash-Search(T,k)
  wyszkaj element o kluczu k
  na liście T[h(k)]

Chained-Hash-Delete(T,x)
  usuń x z listy T[h(key[x])]
```

Analiza

Założenia:

- n – liczba przechowywanych elementów,
- $\alpha = n/m$ – *współczynnink zapełnienia*
- $h(x)$ przyjmuje każdą z wartości $0 \ldots m-1$ jednakowym ppb (dla losowo wybranego x)
- $h(k)$ można obliczyć w czasie $O(1)$

Tw. Średni czas wyszukiwania zakończonego porażką wynosi $\Theta(1+\alpha)$.

D-d. Przy szukaniu klucza k trzeba przejść do końca listy $T[h(k)]$. Średnia długość listy: $\alpha = n/m$. Całkowity czas (obliczenie $h(k)$ plus przejście listy) średnio: $\Theta(1+\alpha)$. $\square$

Analiza

Tw. Średni czas wyszukiwania zakończonego sukcesem wynosi $\Theta(1+\alpha)$.

D-d. Załóżmy, że nowy element wstawiany jest na koniec listy. (Nie ma to wpływu na *średni* czas wyszukiwania (ćw.).) Czas wyszukania elementu (bez liczenia $h(k)$): 1 plus dł. listy tuż przed wstawieniem tego elementu. Średnia dł. listy przed wstawianieniem i-tego elementu: $(i-1)/m$. Stąd średni czas wyszukiwania:

$$\begin{aligned} \frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{i-1}{m}\right) &= 1+\frac{1}{nm}\sum_{i=1}^{n}(i-1) \\ &= 1+\left(\frac{1}{nm}\right)\left(\frac{(n-1)n}{2}\right) \\ &= 1+\frac{\alpha}{2}-\frac{1}{2m} \end{aligned}$$

Dodając obliczenie $h(k)$: $\Theta(2+\alpha/2-1/(2m)) = \Theta(1+\alpha)$. $\square$

Funkcje haszujące

Pożądane właściwości: $P(k)$ – ppb wyboru klucza k z U.
Warunek równomiernego rozkładu: $\sum_{k:h(k)=j} P(k) = 1/m$ dla $j = 0 \ldots m-1$.
Np. klucze losowe liczby $k \in U = [0,1)$ wybierane wg rozkłau jednostajnego. Wtedy wystarcza $h(k) = \lfloor km \rfloor$.
W praktyce: heurystyczne metody wyboru funkcji haszujących.
Utożsamienie kluczy z liczbami naturalnumi: $N = \{0, 1, \ldots\}$.
Haszowanie modularne: $h(k) = k \bmod m$. (w praktyce: dobre wartości m – liczby pierwsze niezbyt bliskie potęgom dwójki)
Haszowanie przez mnożenie: $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$, gdzie $0 < A < 1$, (np. $A = (\sqrt{5}-1)/2 \approx 0.6180339887\ldots$).

Haszowanie uniwersalne

Losowe dobieranie funkcji haszującej niezależne od wstawianych do tablicy kluczy. (np. “złośliwych danych”.)
Rodzina f-cji haszujących $\mathcal{H}$ jest *uniwersalna*, jeśli dla każdej pary kluczy $x, y \in U$, $x \neq y$, liczba f-cji $h \in \mathcal{H}$, dla których $h(x) = h(y)$ wynosi $|\mathcal{H}|/m$.

Tw. Niech K dowolny zbiór n kluczy, $n \leq m$. Niech x dowolny klucz z K. Niech h losowo wybrane z uniwersalnej rodziny f-cji haszujących $\mathcal{H}$. Wtedy: oczekiwana liczba kolizji $h(x)$ z wartościami h dla innych kluczy z K jest < 1.

D-d. Dla $y, z \in K$, niech c_{yz} – zm. losowa: $c_{yz} = 1$ gdy $h(y) = h(z)$ i $c_{yz} = 0$ gdy $h(y) \neq h(z)$. Z def $\mathcal{H}$: $E[c_{yz}] = 1/m$.
Niech C_x – liczba kolizji dla x.
$E[C_x] = \sum_{y \in K, y \neq x} E[c_{xy}] = (n-1)/m$. Stąd $E[C_x] < 1$, bo $n \leq m$. $\square$

Uniwersalna rodzina f-cji haszujących

Przyjmujemy, że m – liczba pierwsza. Reprezentujemy klucz x jako ciąg "cyfr" $\langle x_0, \ldots x_r \rangle$, gdzie każde $x_i < m$. Niech $a = \langle a_0, \ldots a_r \rangle$ – ciąg $r+1$ losowych elementów z $\{0, 1, \ldots, m-1\}$. Niech $h_a(x) = \sum_{i=0}^{r} a_i x_i \bmod m$. Def.: $\mathcal{H} = \bigcup_a \{h_a\}$

Tw. $\mathcal{H}$ jest uniwersalną rodziną f-cji haszujących.

D-d. Weźmy parę różnych kluczy x, y. Niech $x_0 \neq y_0$ (podobnie dla dowolnego i, t. że $x_i \neq y_i$). Dla każdego ciągu $a_1, \ldots, a_r$ istnieje dokładnie jedno a_0, takie że: $a_0(x_0 - y_0) \equiv -\sum_{i=1}^{r} a_i(x_i - y_i) \pmod{m}$, bo m – pierwsza oraz $(x_0 - y_0) \neq 0$. Stąd: para x,y koliduje dla m^r wartości a. Możliwych ciągów a jest m^{r+1}. Czyli ppb. kolizji x i y jest $m^r/m^{r+1} = 1/m$. $\square$

adresowanie otwarte

Elementy – wprost w tablicy.

$h : U \times \{0 \ldots m-1\} \to \{0 \ldots m-1\}$, taka że

$\langle h(k,0) \ldots h(k,m-1) \rangle$ – permutacja ciągu $\langle 0 \ldots m-1 \rangle$.

$h(k,i)$ – i-ta próba dla klucza k.

```
Hash-Insert(T,k)
i ← 0
repeat
    j ← h(k,i)
    if T[j] = NIL then
        T[j] ← k
        return j
    else i ← i + 1
until i = m
error “przepełnienie”
```

adresowanie otwarte

```
Hash-Search(T,k)
i ← 0
repeat
    j ← h(k,i)
    if T[j] = k then
        return j
    i ← i + 1
until T[j] = NIL or i = m
return NIL
```

Uwaga: usuwanie elementu z pozycji i przez wstawienie $T[i] = NIL$ mogłoby “odciąć” klucze, przy których wstawianiu $T[i]$ była odwiedzona i zajęta. Można wstawiać stałą $DELETED$, traktowaną jak NIL w `Hash-Insert` ale nie w `Hash-Search`. (Zanika zależność czasu od α.)

adresowanie otwarte

Warunek równomiernego haszowania: Dla losowego klucza wszystkie $m!$ permutacji – jednakowo prawdopodobne. (W praktyce – trudno spełnić.)

Adresowanie liniowe: Niech $h' : U \to \{0 \dots m-1\}$ – zwykła f-cja haszująca. Stosujemy: $h(k,i) = (h'(k) + i) \bmod m$, dla $i = 0 \dots m$. Jest tylko m różnych ciągów kontrolnych.

Tendencja do grupowania: Jeśli wolną pozycję poprzedza i zajętych, to ppb jej zapełnienia wynosi: $(i+1)/m$ (znacznie więcej niż $1/m$).

Adresowanie kwadratowe: Stosujemy: $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$. (Należy odpowiednio dobrać c_1, c_2.) Również jest tylko m ciągów kontrolnych. Jeśli $h(k_1, 0) = h(k_2, 0)$, to całe ciągi takie same.

Haszowanie dwukrotne

Niech h_1, h_2 – f-cje haszujące. Stosujemy:
$h(k,i) = (h_1(k) + ih_2(k)) \bmod m$.
Pierwsza pozycja: $T[h_1(k)]$. Kolejne pozycje – oddalone o $h_2(k)$ modulo m. Ciąg kontrolny zależy od k a nie od początkowej pozycji. Aby w ciągu kontrolnym były wszystkie pozycje: $h_2(k)$ i m muszą być wzgl. pierwsze. Może być: m – liczba pierwsza i $0 < h_2(k) < m$, np. $h_1(k) = k \bmod m$, $h_2(k) = 1 + (k \bmod m')$, gdzie m' nieco mniejsze niż m.
Adresowanie dwukrotne dopuszcza $\Theta(m^2)$ różnych ciągów kontrolnych.

Analiza adresowania otwartego

Zakładamy “równomierne haszowanie”. Niech m – rozmiar tablicy, n – liczba elementów w tablicy. Niech $\alpha = n/m$.

Tw. Jeśli $\alpha < 1$, to oczekiwana liczba porównań przy poszukiwaniu zakończonym porażką jest $\leq 1/(1-\alpha)$.

D-d. Niech $p_i =$ $Pr\{$dokładnie i początk. pozycji w ciągu kontrolnym – zajętych$\}$

Oczekiwana l. porównań $= 1 + \sum_{i=0}^{\infty} ip_i$. Niech $q_i = Pr\{$co najmniej i pocz. pozycji jest zajętych$\}$. Mamy $\sum_{i=0}^{\infty} ip_i = \sum_{i=0}^{\infty} i(q_i - q_{i+1}) = \sum_{i=1}^{\infty} q_i$.

$q_1 = \frac{n}{m}$,

$q_2 = \frac{n}{m}\left(\frac{n-1}{m-1}\right)$ (pierwsza zajęta i jeden z $n-1$ pozostałych kluczy trafił na drugą pozycję)

Ogólnie: $q_i = \frac{n}{m}\left(\frac{n-1}{m-1}\right)\ldots\left(\frac{n-i+1}{m-i+1}\right) \leq \left(\frac{n}{m}\right)^i = \alpha^i$,bo $n \leq m$.

Dla $i > n$, $q_i = 0$. ...

Analiza adresowania otwartego

...

$$\begin{aligned} 1+\sum_{i=0}^{n} i p_i &= 1+\sum_{i=1}^{\infty} q_i \\ &\le 1+\alpha+\alpha^2+\alpha^3+\ldots \\ &= \frac{1}{1-\alpha} \end{aligned}$$

$\square$

Wniosek. Oczekiwana liczba porównań w `Hash-Insert`: $1/(1-\alpha)$

Tw. Jeśli $\alpha < 1$, to oczekiwana liczba porównań w wyszukiwaniu zakończonym sukcesem wynosi: $\le \frac{1}{\alpha}\ln\frac{1}{1-\alpha}+\frac{1}{\alpha}$

Analiza adresowania otwartego

D-d. Wyszukanie k – taka sama l. porównań jak przy wstawianiu k. Z **Wniosku**: jeśli k był $(i+1)$-szym wstawionym elementem, to oczekiwana l. porównań do jego wyszukania jest $\leq 1/(1-i/m) = m/(m-i)$. Uśredniając po n kluczach w tablicy:

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} = \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} = \frac{1}{\alpha}(H_m - H_{m-n})$$

gdzie $H_i = \sum_{j=1}^{i} 1/j$ – i-ta harmoniczna. Zachodzi: $\ln i \leq H_i \leq \ln i + 1$ (oszacowanie przez całkę).

$$\frac{1}{\alpha}(H_m - H_{m-n}) \leq \frac{1}{\alpha}(\ln m + 1 - \ln(m-n)) =$$
$$= \frac{1}{\alpha}\ln\frac{m}{m-n} + \frac{1}{\alpha} = \frac{1}{\alpha}\ln\frac{1}{1-\alpha} + \frac{1}{\alpha}$$

$\square$