Wstęp do Programowania
potok funkcyjny

Marcin Kubica

2010/2011

Outline

1. Zasada „dziel i rządź" i analiza złożoności
   - Problem sortowania
   - Przykłady algorytmów sortowania
   - Oczekiwana złożoność czasowa algorytmu quick-sort
   - Koszt zamortyzowany i heap-sort

Problem sortowania

Definition

Problem sortowania:

- Zbiór wartości:
  - nieograniczonej mocy,
  - porządek liniowy $\leq$,
  - $\leq$ to jedyna dostępna operacja na elementach,
  - stały koszt porównania elementów.
- Dane: ciąg n elementów.
- Wynik: uporządkowany ciąg elementów, będący permutacją danego ciągu.
- Oszacujmy pesymistyczny koszt sortowania.

Drzewa decyzyjne

Definition

Ustalmy n.

Drzewo decyzyjne:

- drzewo binarne,
- węzły wewnętrzne — operacje porównania elementów na konkretnych pozycjach,
- liście — permutacje określające wyniki sortowania.

Obliczeniu odpowiada przejście ścieżki od korzenia drzewa do liścia.

Eliminacja algorytmów randomizowanych

Jeśli nasz algorytm jest randomizowany, to ustalamy z góry ciąg losowanych liczb i analizujemy go, jak deterministyczny.

Drzewa decyzyjne

Definition

Ustalmy n.

Drzewo decyzyjne:

- drzewo binarne,
- węzły wewnętrzne — operacje porównania elementów na konkretnych pozycjach,
- liście — permutacje określające wyniki sortowania.

Obliczeniu odpowiada przejście ścieżki od korzenia drzewa do liścia.

Eliminacja algorytmów randomizowanych

Jeśli nasz algorytm jest randomizowany, to ustalamy z góry ciąg losowanych liczb i analizujemy go, jak deterministyczny.

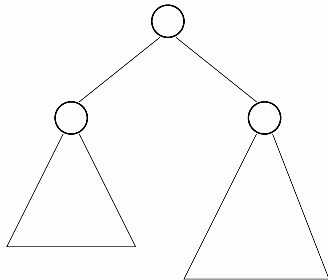
Drzewa decyzyjne

Fact

Jeśli drzewo binarne ma k liści, to jego wysokość jest niemniejsza niż $\log_2 k$.

Dowód.

1. $k = 1$
   Drzewo = jeden liść.
2. $k > 0$
   Jedno z poddrzew ma przynajmniej $\lceil \frac{k}{2} \rceil$ liści.
   Z założenia indukcyjnego, jego wysokość $\geq \log_2 \frac{k}{2}$.
   Wysokość drzewa $\geq \log_2 k$.



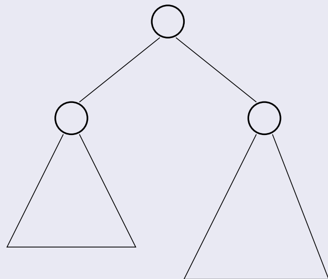
Drzewa decyzyjne

Fact

Jeśli drzewo binarne ma k liści, to jego wysokość jest niemniejsza niż $\log_2 k$.

Dowód.

1. $k = 1$
   Drzewo = jeden liść.
2. $k > 0$
   Jedno z poddrzew ma przynajmniej $\lceil \frac{k}{2} \rceil$ liści.
   Z założenia indukcyjnego, jego wysokość $\geq \log_2 \frac{k}{2}$.
   Wysokość drzewa $\geq \log_2 k$.



Analiza problemu sortowania

Lemma

$3^n \cdot n! \geq n^n$

Dowód.

Dowód przebiega indukcyjnie.

1. $n = 1$: $3 \geq 1$.
2. $n > 1$:

$$3^{n+1} n!(n+1) \geq 3(n+1)n^n \geq$$
$$\geq e(n+1)n^n \geq \left(1+\frac{1}{n}\right)^n (n+1)n^n =$$
$$= \frac{(n+1)^n}{n^n}(n+1)n^n = (n+1)^{n+1}$$

□

Analiza problemu sortowania

Lemma

$3^n \cdot n! \geq n^n$

Dowód.

Dowód przebiega indukcyjnie.

1. $n = 1$: $3 \geq 1$.
2. $n > 1$:

$$
\begin{aligned}
3^{n+1} n!(n+1) &\geq 3(n+1)n^n \geq \\
\geq e(n+1)n^n &\geq \left(1 + \frac{1}{n}\right)^n (n+1)n^n = \\
= \frac{(n+1)^n}{n^n}(n+1)n^n &= (n+1)^{n+1}
\end{aligned}
$$

Analiza problemu sortowania

Theorem

Pesymistyczny koszt sortowania jest rzędu $\Omega(n \log n)$.

Dowód.

- Rozpatrujemy wszystkie możliwe wykonania algorytmu dla ciągu n różnych elementów.
- Przebieg obliczenia zależy tylko od wyników porównań.
- Obliczenia przedstawiamy w postaci drzewa decyzyjnego.
- Drzewo ma co najmniej $n!$ liści.
- Z lematu i faktu wynika, że wysokość drzewa h:

$$h \;\geq\; \log_2 n! \;\geq\; \log_2 \left(\frac{n^n}{3^n}\right) \;=\; n \log_2 \left(\frac{n}{3}\right) = \Omega(n \log n)$$

□

Analiza problemu sortowania

Theorem

Pesymistyczny koszt sortowania jest rzędu $\Omega(n \log n)$.

Dowód.

- Rozpatrujemy wszystkie możliwe wykonania algorytmu dla ciągu n różnych elementów.
- Przebieg obliczenia zależy tylko od wyników porównań.
- Obliczenia przedstawiamy w postaci drzewa decyzyjnego.
- Drzewo ma co najmniej $n!$ liści.
- Z lematu i faktu wynika, że wysokość drzewa h:

$$h \geq \log_2 n! \geq \log_2 \left(\frac{n^n}{3^n}\right) = n \log_2 \left(\frac{n}{3}\right) = \Omega(n \log n)$$

Selection sort

Example

- Podział: wybór maksimum + posortowanie krótszego ciągu + sklejenie.
- Złożoność czasowa:

$$\begin{array}{rcl} T(0) & = & \Theta(1) \\ T(n) & = & \Theta(n) + T(n-1) \quad = \quad \Theta(n^2) \end{array}$$

- Złożoność pamięciowa:
  Rekurencja ogonowa + żonglowanie n wartościami + wynik

$$M(n) = \Theta(n)$$

Selection sort

Example

```
let select_max (h::t) =
  fold_left (fun (m, r) x ->
                          if x > m then (x, m::r) else (m, x::r))
               (h, []) t;;

let selection_sort l =
  let przenies (s, l) =
    let (m, r) = select_max l
    in (m::s, r)
  in
    iteruj ([], l)
           przenies
           (fun (_, l) -> l = [])
           (fun (s, _) -> s);;
```

Insertion sort

Example

- Podział: posortowanie krótszego ciągu + wstawienie elementu.

```
let wstaw l x =
  (filter (fun y -> y <= x) l) @
  (x :: (filter (fun y -> y > x) l));;

let insertion_sort l = fold_left wstaw [] l;;
```

- Złożoność czasowa:

$$T(0) = \Theta(1)$$
$$T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$$

- Złożoność pamięciowa:
  W każdej chwili pamiętamy kilka list o łącznej długości $\Theta(n)$.
  $M(n) = \Theta(n)$

Insertion sort

Example

- Podział: posortowanie krótszego ciągu + wstawienie elementu.

```
let wstaw l x =
  (filter (fun y -> y <= x) l) @
  (x :: (filter (fun y -> y > x) l));;

let insertion_sort l = fold_left wstaw [] l;;
```

- Złożoność czasowa:

$$\begin{aligned} T(0) &= \Theta(1) \\ T(n) &= T(n-1) + \Theta(n) = \Theta(n^2) \end{aligned}$$

- Złożoność pamięciowa:
  W każdej chwili pamiętamy kilka list o łącznej długości $\Theta(n)$.
  $M(n) = \Theta(n)$

Insertion sort

Example

- Podział: posortowanie krótszego ciągu + wstawienie elementu.

```
let wstaw l x =
  (filter (fun y -> y <= x) l) @
  (x :: (filter (fun y -> y > x) l));;

let insertion_sort l = fold_left wstaw [] l;;
```

- Złożoność czasowa:

$$\begin{aligned} T(0) &= \Theta(1) \\ T(n) &= T(n-1) + \Theta(n) = \Theta(n^2) \end{aligned}$$

- Złożoność pamięciowa:
  W każdej chwili pamiętamy kilka list o łącznej długości $\Theta(n)$.
  $M(n) = \Theta(n)$

Insertion sort

Definition

- *Inwersją* w ciągu $[x_1; x_2; \ldots; x_n]$ nazywamy każdą taką parę indeksów $1 \leq a < b \leq n$, dla której $x_a > x_b$.
  Oznaczenie: i.
- Wszystkich możliwych par indeksów do rozważenia jest $\frac{n(n-1)}{2}$.
  Jeżeli dany ciąg jest malejący, to $i = \frac{n(n-1)}{2}$.
  Z drugiej strony, dla ciągu rosnącego mamy $i = 0$.
  $i = \Theta(n^2)$.
- Oczekiwana liczba inwersji w losowej permutacji jest też rzędu $\Theta(n^2)$.

Insertion sort

Example

```
let wstaw x l =
  let rec wst a x l =
    match l with
      []    -> rev(x::a) |
      h::t ->
        if x > h then wst (h::a) x t
        else rev (x::a) @ l
  in wst [] x l;;

let insertion_sort l = fold_right wstaw l [];;
```

- W każdym kroku iteracji w wst pozbywamy się jednej inwersji.
- Złożoność czasowa rzędu $\Theta(n+i)$.

Insertion sort

Example

```
let wstaw x l =
  let rec wst a x l =
    match l with
      []    -> rev(x::a) |
      h::t ->
        if x > h then wst (h::a) x t
        else rev (x::a) @ l
  in wst [] x l;;

let insertion_sort l = fold_right wstaw l [];;
```

- W każdym kroku iteracji w `wst` pozbywamy się jednej inwersji.
- Złożoność czasowa rzędu $\Theta(n+i)$.

Sortowanie przez scalanie

Example

Podział: podziel listę + posortuj powstałe listy + scal.

```
let rec merge_sort l =
  match l with
    [] -> []|
    [x] -> [x]|
    _ ->
      let (l1, l2) = split l
      in
        merge (merge_sort l1) (merge_sort l2);;
```

Sortowanie przez scalanie

Example

```
let split l =
  fold_left (fun (l1, l2) x -> (l2, x::l1)) ([], []) l;;

let merge l1 l2 =
  let rec mrg a l1 l2 =
    match l1 with
      [] -> (rev a) @ l2 |
      (h1::t1) ->
        match l2 with
          [] -> (rev a) @ l1 |
          (h2::t2) ->
            if h1 > h2 then mrg (h2::a) l1 t2
            else mrg (h1::a) t1 l2
  in mrg [] l1 l2;;
```

Sortowanie przez scalanie

Example

Złożoność czasowa:

Procedury `split` i `merge` działają w czasie $\Theta(n)$.

Dwa wywołania rekurencyjne dla ciągów o połowę krótszych.

$$\begin{aligned} T(1) &= \Theta(1) \\ T(n) &= \Theta(2n) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \\ T(2^k) &= \Theta(2^{k+1}) + 2T\left(2^{k-1}\right) = \sum_{i=0}^{k} 2^{k+1} = \\ &= (k+1)2^{k+1} = \Theta(n \log n) \end{aligned}$$

$T(n)$ jest monotoniczna.

$T(n) = \Theta(n \log n)$ dla dowolnych n. (optymalna)

Sortowanie przez scalanie

Example

Złożoność pamięciowa:
Największe zagłębienie rekurencji + drugie wywołania rekurencyjne:

- lista do posortowania (n),
- dwie połówki, dla drugiej wywołanie rekurencyjne $\left(\frac{n}{2}\right)$,
- pierwsza połówka jest już posortowana $\left(\frac{n}{2}\right)$, a druga jeszcze nie.

$$M(n) = \frac{3}{2}n + M(n/2) \leq \frac{3n}{2}\sum_{k\geq 0}\frac{1}{2^k} = \Theta(n)$$

Optymalna złożoność pamięciowa.

Quick-sort

Example

Podział:

- wybieramy losowy element s z ciągu,
- dzielimy elementy ciągu na $< s$, $= s$ i $> s$,
- sortujemy ciąg $< s$ i $> s$,
- sklejamy uzyskane ciągi.

Quick-sort

Example

```
let rec quick_sort l =
  let split l s = (filter (fun y -> y < s) l,
                   filter (fun y -> y = s) l,
                   filter (fun y -> y > s) l)
  in
    if length l < 2 then l else
      let  s = nth l (Random.int (length l))
      in
        let (ll, le, lg) = split l s
        in (quick_sort ll) @ le @ (quick_sort lg);;
```

Quick-sort

Example

- Nienajlepsza złożoność pesymistyczna.
- Algorytm nie jest odporny na wybór elementów w porządku rosnącym lub malejącym.

$$M(0) = \Theta(1)$$

$$M(n) = \Theta(n) + M(n-1) = \Theta(n^2)$$

$$T(0) = \Theta(1)$$

$$T(n) = \Theta(n) + T(n-1) = \Theta(n^2)$$

Quick-sort

Theorem

Algorytm Quick-sort, dla permutacji zbioru $\{1, \ldots, n\}$ *ma oczekiwaną złożoność czasową* $\Theta(n \log n)$.

Quick-sort

Dowód twierdzenia

- Oznaczmy przez $T(n)$ oczekiwany czas działania.
- Każda z wartości s jest tak samo prawdopodobna:

$$
\begin{aligned}
T(n) &= \frac{1}{n}\left(\sum_{s=1}^{n}(T(s-1)+T(n-s)+\Theta(n))\right) = \\
&= \frac{2}{n}\left(\sum_{s=0}^{n-1}T(s)\right)+\Theta(n) \\
T(0) &= T(1) = \Theta(1)
\end{aligned}
$$

- Komentarz: Operacje na składnikach $\Theta(\ldots)$.

Quick-sort

Dowód twierdzenia

$$
\begin{aligned}
nT(n) &= 2\sum_{s=0}^{n-1} T(s) + \Theta(n^2) \quad = \\
&= 2\sum_{s=0}^{n-1} T(s) + \Theta(n^2) + (n-1)T(n-1) - \\
&- 2\sum_{s=0}^{n-2} T(s) - \Theta((n-1)^2) \quad = \\
&= 2T(n-1) + (n-1)T(n-1) + \Theta(n) \quad = \\
&= (n+1)T(n-1) + \Theta(n)
\end{aligned}
$$

Quick-sort

Dowód twierdzenia.

$$\begin{aligned}\frac{T(n)}{n+1} &= \frac{T(n-1)}{n} + \Theta\left(\frac{1}{n+1}\right) &= \\ &= \Theta\left(\frac{1}{n+1} + \frac{1}{n} + \cdots + \frac{1}{3}\right) + \Theta(1) &= \\ &= \Theta(\log n)\end{aligned}$$

Stąd $T(n) = \Theta(n \log n)$. □

Kolejka priorytetowa

Example

- Zasada pobożnych życzeń:
  Załóżmy, że mamy dostępną *kolejkę priorytetową*.
- Kolejka priorytetowa = kolekcja elementów + porządek liniowy.
  Porządek liniowy: $\leq$

Sygnatura kolejki priorytetowej

Example

```
module type PRI_QUEUE = sig
  type 'a pri_queue                                    typ kolejek
  val empty_queue : 'a pri_queue                    pusta kolejka
  val is_empty : 'a pri_queue -> bool                 czy pusta?
  val put : 'a pri_queue -> 'a -> 'a pri_queue
                                               włożenie elementu
  val getmax : 'a pri_queue -> 'a                        maximum
  val removemax : 'a pri_queue -> 'a pri_queue
                                                   usuń maximum
  exception Empty_Queue                        gdy kolejka pusta
end;;
```

Heap-sort

Example

- Zasada pobożnych życzeń: mamy dostępną kolejkę priorytetową.
- Najpierw wkładamy wszystkie elementy do kolejki.
- Następnie wyjmujemy je w porządku nierosnącym.

```
let heap_sort l =
  let wloz = fold_left put empty_queue l
  and wyjmij (l, q) = ((getmax q)::l, removemax q)
  in iteruj
       ([], wloz)
       wyjmij
       (fun (_, q) -> is_empty q)
       (fun (l, _) -> l)
```

Heap-sort

Example

- Zasada pobożnych życzeń: mamy dostępną kolejkę priorytetową.
- Najpierw wkładamy wszystkie elementy do kolejki.
- Następnie wyjmujemy je w porządku nierosnącym.

```
let heap_sort l =
  let wloz = fold_left put empty_queue l
  and wyjmij (l, q) = ((getmax q)::l, removemax q)
  in iteruj
      ([], wloz)
      wyjmij
      (fun (_, q) -> is_empty q)
      (fun (l, _) -> l)
```

Heap-sort

Example

- Złożoność czasowa:
  - zależy implementacji kolejki priorytetowej,
  - n operacji włożenia i n operacji wyjęcia maksimum.
  - pozostałe operacje — $\Theta(n)$.
- Kosztu pamięciowy:
  - założenie: kolejka rozmiaru n zajmuje $\Theta(n)$ pamięci,
  - $M(n) = \Theta(n)$

Implementacja listowa kolejek

Example

- Kolejka = lista nieuporządkowana.
- Funkcja abstrakcji: lista $\rightarrow$ multizbiór jej elementów.
- Prosta, ale nieefektywna implementacja. $T(n) = \Theta(n^2)$
- Otrzymujemy sortowanie przez wybieranie.

Implementacja listowa kolejek

Example

```
module Unordered_Pri_Queue : PRI_QUEUE = struct
  exception Empty_Queue
  type 'a pri_queue = 'a list
  let empty_queue = []
  let is_empty q = q = []
  let put q x = x::q
  let getmax q =
    if q = [] then raise Empty_Queue
    else fst (select_max q)
  let removemax q =
    if q = [] then raise Empty_Queue
    else snd (select_max q)
end;;
```

select_max pochodzi z sortowania przez wybieranie.

Implementacja listowa kolejek

Example

- Kolejka = lista uporządkowana nierosnąco.
- Funkcja abstrakcji: lista $\rightarrow$ multizbiór jej elementów $(1-1)$.
- Otrzymujemy sortowanie przez wstawianie. $T(n) = \Theta(n^2)$

Implementacja listowa kolejek

Example

```
module Ordered_Pri_Queue : PRI_QUEUE = struct
  exception Empty_Queue
  type 'a pri_queue = 'a list
  let empty_queue = []
  let is_empty q = q = []
  let put q x = (filter (fun y -> y > x) q) @
                (x :: (filter (fun y -> y <= x) q))
  let getmax q =
    if q = [] then raise Empty_Queue
    else hd q
  let removemax q =
    if q = [] then raise Empty_Queue
    else tl q
end;;
```

Stóg

Example

Stóg to:

- drzewo binarne,
- w węzłach znajdują się elementy,
- dla każdego poddrzewa:
  wartość w korzeniu poddrzewa = maksimum z wartości w poddrzewie,
- dla każdego poddrzewa, pamiętamy jego wielkość.

Stóg

Example

```
type 'a pri_queue =
  Node of 'a * 'a pri_queue * 'a pri_queue * int |
  Null

let empty_queue = Null

let is_empty q = q = Null

let size q =
  match q with
    Null -> 0 |
    Node (_, _, _, n) -> n
```

Stóg

Example

```
let getmax h =
  match h with
    Null -> raise Empty_Queue |
    Node (r, _, _, _) -> r

let set_root h r =
  match h with
    Null -> Node (r, Null, Null, 1) |
    Node (_, l, p, n) -> Node (r, l, p, n)
```

Stóg

Example

```
let rec put h x =
  match h with
    Null -> Node (x, Null, Null, 1) |
    Node (r, l, p, n) ->
      if size l <= size p then
        Node((max x r), (put l (min x r)), p, (n+1))
      else
        Node((max x r), l, (put p (min x r)), (n+1))
```

Stóg

Example

```
let rec removemax h =
    match h with
      Null -> raise Empty_Queue |
      Node (_, Null, Null, _) -> Null |
      Node (_, l, Null, _) -> l |
      Node (_, Null, p, _) -> p |
      Node (_, (Node (rl, _, _, _) as l),
               (Node (rp, _, _, _) as p), n) ->
        if rl >= rp then
          Node (rl, removemax l, p, n - 1)
        else
          Node (rp, l, removemax p, n - 1)
```

Stóg

Fact

Wykonujemy n operacji `put` *i n operacji* `getmax`.
Wówczas wysokość drzewa stogu nie przekracza $\lfloor \log_2 n \rfloor$.

Dowód.

- Jeśli stóg nie jest drzewem pełnym, to wstawienie nie zwiększa jego wysokości.
- Wystarczy rozważyć wyłącznie operacje `put`.
- Dla każdego węzła zachodzi warunek:
  liczba wierzchołków w lewym i prawym poddrzewie różnią się o co najwyżej 1.
- Stóg jest drzewem zrównoważonym.
- Drzewo binarne o n wierzchołkach ma wysokość $\lfloor \log_2 n \rfloor$. □

Stóg

Fact

Wykonujemy n operacji `put` *i n operacji* `getmax`.
Wówczas wysokość drzewa stogu nie przekracza $\lfloor \log_2 n \rfloor$.

Dowód.

- Jeśli stóg nie jest drzewem pełnym, to wstawienie nie zwiększa jego wysokości.
- Wystarczy rozważyć wyłącznie operacje `put`.
- Dla każdego węzła zachodzi warunek:
  liczba wierzchołków w lewym i prawym poddrzewie różnią się o co najwyżej 1.
- Stóg jest drzewem zrównoważonym.
- Drzewo binarne o n wierzchołkach ma wysokość $\lfloor \log_2 n \rfloor$. □

Heap-sort

Example

Koszt czasowy:

- operacji wstawiania jest rzędu $\Theta(\log k) = O(\log n)$, gdzie k jest aktualną liczbą elementów w stogu,
- usuwania jest co najwyżej rzędu wysokości stogu, czyli $O(\log n)$,
- sortowania za pomocą stogu wynosi $\Theta(n \log n)$.

Koszt zamortyzowany operacji na stogu

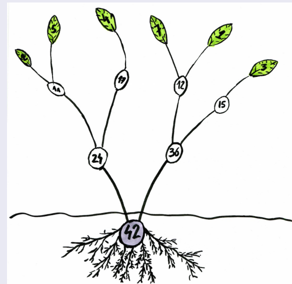
Example

- **Problem:**
  Ciąg n operacji wstawień i usunięć elementów ze stogu.
  Zwykle rozmiar stogu jest dużo mniejszy niż n.
  Czy potrafimy lepiej oszacować koszt wstawiania i usuwania?
- Koszt wstawiania jest rzędu $\Theta(\log k)$, gdzie k to rozmiar stogu. (Rozmiar drzewa, do którego wstawiamy, w każdym kroku maleje przynajmniej $2\times$.)
- Koszt usuwania jest rzędu $\Theta(h)$, gdzie h to wysokość stogu.
- h może być istotnie większe niż $\Theta(\log k)$.
- „Sumarycznie” możemy przyjąć, że:
  - koszt operacji wstawiania jest $\Theta(\log k)$.
  - koszt operacji usuwania jest stały.

Intuicje

Intuicje

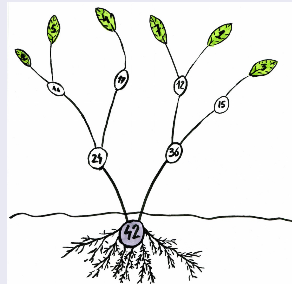
- Obliczenia wymagają energii.
- Zamiast czasu liczymy zużytą energię.
- Struktura danych może magazynować energię.
- Energia potencjalna stogu — wkładając element, wydatkujemy energię proporcjonalną do liczby kroków.
  Zużywamy energię rzędu $\Theta(\log_2 k)$.



Intuicje

Intuicje

- Energię potrzebną do usunięcia maksimum pokrywamy spadkiem energii potencjalnej stogu. Musimy dostarczyć energię rzędu $\Theta(1)$.
- Początkowa energia stogu wynosi 0, a końcowa jest dodatnia.
  Łączny czas obliczeń jest *co najwyżej* taki, jak zużyta energia.



Koszt zamortyzowany

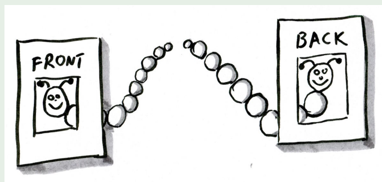
Definition

- Określamy *funkcję potencjału* struktury danych.
- Koszt zamortyzowany operacji =
  koszt czasowy operacji + zmiana funkcji potencjału.
- Kosz czasowy ciągu operacji =
  koszt zamortyzowany ciągu operacji +
  początkowa wartość funkcji potencjału −
  końcowa wartość funkcji potencjału.

Kolejka FIFO

Example

- Kolejka FIFO – ang. *first in, first out*.
- Cel: koszt zamortyzowany operacji $\Theta(1)$.
- Elementy kolejki przechowujemy na dwóch listach.
  Do jednej dokładamy elementy, a z drugiej wyjmujemy.
  Gdy kolejka elementów do wyjmowania jest pusta, przenosimy elementy z jednej kolejki do drugiej, odwracając ich kolejność.
- Dodatkowo pamiętamy rozmiar kolejki i jej pierwszy element.



Kolejka FIFO

Example

```
module type QUEUE =
  sig
    exception EmptyQueue
    type 'a queue
    val empty : 'a queue
    val is_empty : 'a queue -> bool
    val insert : 'a queue -> 'a -> 'a queue
    val front : 'a queue -> 'a
    val remove : 'a queue -> 'a queue
  end;;
```

Kolejka FIFO

Example

```
module Fifo : QUEUE = struct
    exception EmptyQueue

    type 'a queue = {front: 'a list; back: 'a list; size: int}

    let empty = {front=[]; back=[]; size=0}

    let size q = q.size

    let is_empty q = size q = 0

    let balance q =
      match q with
        {front=[]; back=[]}         -> q |
        {front=[]; back=b; size=s} -> {front=rev b; back=[]; size=s} |
        _                           -> q
```

Kolejka FIFO

Example

```
    let insert {front=f; back=b; size=n} x =
      balance {front=f; back=x::b; size=n+1}

    let front q =
      match q with
        {front=[]}   -> raise EmptyQueue |
        {front=x::_} -> x

    let remove q =
      match q with
        {front=_::f} -> balance {front=f; back=q.back; size=q.size-1} |
        _            -> raise EmptyQueue
  end;;
```

Kolejka FIFO

Example

- Funkcja potencjału = `length q.back`.
- Procedury `is_empty_queue`, `size` i `first` mają stały koszt czasowy i nie zmieniają funkcji potencjału.
- Funkcja potencjału pustej kolejki = 0.
- `put` — koszt czasowy $\Theta(1)$ + zwiększenie funkcji potencjału o 1 = koszt zamortyzowany $\Theta(1)$.
- `remove` — $\Theta(1)$ + koszt procedury `balance`.
  Rzeczywisty koszt czasowy `balance` = $\Theta(n)$.
  Spadek funkcji potencjału do 0.
  Koszt zamortyzowany `balance` = $\Theta(1)$.
  Koszt zamortyzowany `remove` = $\Theta(1)$.

Deser

```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

http://xkcd.com/221/