NOTATKI I UZUPEŁNIENIA DO WYKŁADU 4

26 listopada 2016

Sortowanie szybkie

Dziel: tablica $A[p..r]$ jest dzielona na dwie podtablice $A[p..q-1]$ i $A[q+1..r]$, takie że
  każdy element $A[p..q-1]$ jest $\leq A[q]$
  każdy element $A[q+1..r]$ jest $> A[q]$

Zwyciężaj: podtablice $A[p..q-1]$ i $A[q+1..r]$
  są sortowane za pomocą rekurencyjnych
  wywołań algorytmu sortowania szybkiego

Uwaga: Elementem rozdzielającym $A[q]$ może być **dowolny** element tablicy; przyjmujemy, że jest nim ostatni element, tj. $A[r]$

QUICK-SORT (A, p, r)
  if $p < r$
    then $q =$ PODZIAL (A, p, r)
      QUICK-SORT $(A, p, q-1)$
      QUICK-SORT $(A, q+1, r)$

Aby posortować całą tablicę $A[1..n]$, wywołujemy QUICK-SORT $(A, 1, n)$

PODZIAL (A, p, r)
  $x = A[r]$
  $i = p - 1$
  for $j = p$ **to** $r - 1$
    do if $A[j] \leq x$
      then $i = i + 1$
        $A[i] \leftrightarrow A[j]$
  $A[i+1] \leftrightarrow A[r]$

Typeset by $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX

return $i + 1$

Złożoność pesymistyczna

W każdym wywołaniu rekurencyjnym procedury QUICK-SORT, procedura dzieląca tworzy jeden obszar złożony z 1 elementu a drugi złożony z pozostałych. Koszt podziału wynosi $\Theta(n)$. Zatem

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

tj.

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

Stosując metodę podstawiania otrzymujemy

$$W(n) = \Theta(n^2)$$

Złożoność optymistyczna

Przy najrówniejszym możliwym podziale otrzymujemy dwa podproblemy, z których każdy ma rozmiar nie większy niż $n/2$, tj.

$$T(n) \leq 2T(n/2) + \Theta(n)$$

$$B(n) = O(n \lg n)$$

Złożoność średnia

Zakładamy, że wszystkie permutacji liczb na wejściu są jednakowo prawdopodobne

$$A(n) = O(n \lg n)$$

duża stała ukryta w notacji duże O.

Randomizowana wersja quicksort

Wybieramy losowo indeks z zakresu $p, \ldots r$, zapewniając, że element rozdzielający $x = A[r]$ może być z takim samym prawdopodobieństwem dowolnym spośród $r - p + 1$ elementów podtablicy $A[p..r]$

procedure R-PODZIAL (A, p, r)
    $i =$ RANDOM (p, r)
    $A[r] \leftrightarrow A[i]$
    return PODZIAL (A, p, r)

R-QUICK-SORT (A, p, r)
    if $p < r$
        then $q =$ R-PODZIAL (A, p, r)
                R-QUICK-SORT $(A, p, q - 1)$
                R-QUICK-SORT $(A, q + 1, r)$

Złożoność średnia dla dowolnego układu danych wejściowych

$$A(n) = \Theta(n \lg n)$$

Twierdzenie. Dowolny algorytm sortujący n elementów **za pomocą porównań** w przypadku pesymistycznym wymaga

$$\Omega(n \lg n)$$

porównań

Sortowanie przez zliczanie

Założenia:

* każdy z n elementów ciągu wejściowego jest liczbą całkowitą z przedziału od 0 do k dla pewnego ustalonego całkowitego k

* dane wejściowe zawarte są w tablicy $A[1..n]$

* dane posortowane zostają umieszczone w tablicy $B[1..n]$

* tablica $C[0..k]$ początkowo wyzerowana przechowuje tymczasowe dane pomocnicze

Idea:

* dla każdego elementu i z ciągu wejściowego wyznaczamy liczbę elementów mniejszych od i; w ten sposób otrzymujemy dokładną pozycję danego elementu w ciągu posortowanym

* jeżeli dozwolonych jest więcej elementów o tej samej wartości, to postępowanie należy zmodyfikować, aby wszystkie takie elementy nie trafiały na tę samą pozycję

Algorytm:

– dla każdego $i = 0, 1, \ldots, k$ wyznacz liczbę elementów tablicy A równych i ($C[i]$: liczba elementów równych i)

for $j = 1$ **to** n
  do $C[A[j]] = C[A[j]] + 1$

– dla każdego $i = 0, 1, \ldots, k$ wyznacz liczbę elementów tablicy A mniejszych bądź równych i
($C[i]$: liczba elementów $\leq i$)

for $i = 1$ **to** k
  do $C[i] = C[i] + C[i - 1]$

– umieść wszystkie elementy $A[j]$ na właściwych pozycjach w tablicy B

for $j = n$ **downto** 1
  do $B[C[A[j]]] = A[j]$
    $C[A[j]] = C[A[j]] - 1$

SORT-COUNT (A, B, n, k)
  for $j = 1$ **to** k
    do $C[j] = 0$
  for $j = 1$ **to** n
    do $C[A[j]] = C[A[j]] + 1$
  for $i = 1$ **to** k
    do $C[i] = C[i] + C[i-1]$
  for $j = n$ **downto** 1
    do $B[C[A[j]]] = A[j]$
      $C[A[j]] = C[A[j]] - 1$

Całkowity czas działania, to

$$\Theta(n+k)$$

Jeżeli $k = O(n)$, to czas sortowania wynosi $\Theta(n)$

ZBIORY DYNAMICZNE

Zbiór dynamiczny – zbiór mogący się powiększać, zmniejszać lub zmieniać w czasie działania algorytmu

Zbiór dynamiczny, na którym można wykonać operacje:

wstawiania elementu do zbioru

usuwania elementu ze zbioru

sprawdzania, czy dany element należy do zbioru

nazywa się **słownikiem**

Elementy zbioru dynamicznego

Każdy element zbioru dynamicznego jest reprezentowany przez **obiekt**, którego atrybuty można odczytywać oraz modyfikować, jeśli dysponujemy **wskaźnikiem** do tego obiektu

W wielu przypadkach jeden z atrybutów każdego obiektu jest wyróżniony jako jego

klucz

Obiekt może także zawierać **dodatkowe dane** zawarte w innych atrybutach obiektu (nie wpływają one w istotny sposób na realizację zbioru)

Podstawowe operacje na zbiorach dynamicznych

– **zapytania**: pozwalają uzyskać pewne informacje na temat zbioru

– **operacje modyfikujące**: pozwalają zmienić zbiór

Lista typowych operacji

SEARCH (S, k)

zapytanie, które dla danego zbioru S oraz wartości klucza k daje w wyniku wskaźnik x do takiego elementu w zbiorze S, że $x.key = k$ lub NIL, jeśli żaden taki element nie należy do S

INSERT (S, x)

operacja modyfikująca, dodająca do S element wskazany przez x

DELETE (S, x)

operacja modyfikująca, która dla danego wskaźnika x do elementu w zbiorze S usuwa ten element ze zbioru S (argumentem jest tutaj wskaźnik do elementu, a nie wartość klucza!)

STOSY

Stos (*stack*) jest implementacją zbioru dynamicznego, w którym element **usuwany** jest wyznaczony **jednoznacznie** – ze zbioru jest usuwany element, który został do niego dodany najpóźniej

last-in, first-out

LIFO

Podstawowe operacje stosowe

PUSH – odłożenie elementu na stos; rozmiar stosu zwiększa się o jeden

POP – zdjęcie i udostępnienie elementu ze szczytu stosu; rozmiar stosu zmniejsza się o jeden; próba zdjęcia elementu z pustego stosu powoduje wystąpienie błędu

SIZE – zwracanie liczby elementów aktualnie znajdujących się na stosie

STACK-EMPTY – sprawdzenie czy stos jest pusty

CLEAR – usunięcie wszystkich elementów ze stosu; stos staje się stosem pustym

Tablicowa implementacja stosu

Stos zawierający nie więcej niż n elementów można zaimplementować w tablicy $S[1..n]$, z którą jest związany **atrybut** $S.top$, którego wartość jest indeksem ostatnio wstawionego elementu

Stos składa się z elementów

$$S[1..S.top]$$

gdzie $S[1]$ jest elementem na dnie stosu, a $S[S.top]$ jest elementem na wierzchołku stosu

Jeżeli $S.top = 0$, to stos jest *pusty*

STACK-EMPTY (S)
  if $S.top == 0$
    then return true
    else return false

Operacja odkładania elementu do stosu

PUSH (S, x)
  $S.top = S.top + 1$
  if $S.top > n$
    then error „przepełnienie"
    else $S[S.top] = x$

Operacja usuwania elementu ze stosu

POP (S)
  if STOCK-EMPTY (S)
    then error „niedomiar"
    else $S.top = S.top - 1$
      return $S[S.top + 1]$

Każda z tych operacji działa w czasie $O(1)$

Operacja usuwania k elementów z wierzchołka stosu S (lub opróżniania go jeśli na stosie było mniej niż k elementów)

MULTIPOP (S, k)
  while not STACK-EMPTY(S) **and** $k \neq 0$
    do POP(S)
      $k = k - 1$

Całkowity koszt wynosi $\min(s, k)$, gdzie s=SIZE(S)

KOLEJKI

Kolejka (*queue*) – ze zbioru jest usuwany element ,,najstarszy” w zbiorze tj. dodany do niego najwcześniej

first-in, first-out **FIFO**

Podstawowe operacje kolejkowe

ENQUEUE – umieszcza element na końcu kolejki; rozmiar kolejki zwiększa się o jeden

DEQUEUE – pobiera element z początku kolejki; rozmiar kolejki zmniejsza się o jeden; próba pobrania elementu z pustej kolejki powoduje wystąpienie błędu

SIZE – zwraca liczbę elementów aktualnie znajdujących się w kolejce

QUEUE-EMPTY – sprawdza czy kolejka jest pusta

CLEAR – ”usunwa” wszystkie elementy z kolejki; kolejka staje się pusta

Tablicowa implementacja kolejki

Kolejkę o co najwyżej $n - 1$ elementach można zaimplementować za pomocą tablicy $Q[1..n]$

W pseudokodzie przyjmujemy, że $n = Q.length$

Atrybuty:

$Q.head$ – indeks elementu tablicy, w którym znajduje się pierwszy element kolejki

$Q.tail$ – podaje wolną pozycję, na którą można wstawić nowy element

Elementy kolejki są na pozycjach

$$Q.head, Q.head + 1, \dots , Q.tail - 1$$

Tablica Q jest „cykliczna”, tj. pozycja o numerze 1 jest bezpośrednim następnikiem pozycji o numerze n

Jeśli $Q.head = Q.tail$, to kolejka jest pusta

Jeśli $Q.head = Q.tail + 1$, to kolejka jest pełna

Początkowo $Q.head = Q.tail = 1$

Operacja wstawiania elementu do kolejki

ENQUEUE (Q, x)
    $Q[Q.tail] = x$
    if $Q.tail == n$
        then $Q.tail = 1$
        else $Q.tail = Q.tail + 1$

Próba wstawienia nowego elementu do kolejki pełnej sygnalizowana jest jako błąd *przepełnienia*

Operacja usuwania elementu z kolejki

DEQUEUE (Q)
    $x = Q[Q.head]$
    if $Q.head == n$
        then $Q.head = 1$
        else $Q.head = Q.head + 1$
    return x

Próba usunięcia elementu z kolejki pustej jest sygnalizowana jako błąd *niedomiaru*

LISTY

Lista – uporządkowana kolekcja elementów

Podstawową własnością różniącą listy od tablic jest ich rozmiar:

– rozmiar tablicy jest ustalony (z wyjątkiem tablic dynamicznych)

– rozmiar listy może się zmieniać

Lista tablicowa

– najefektywniejsza, gdy dostęp do elementów listy odbywa się głównie na podstawie indeksów lub w sposób sekwencyjny

– nieefektywność operacji wstawiania i usuwania elementów

Lista z dowiązaniami

– struktura danych, w której elementy są ułożone w liniowym porządku

– porządek na liście określają *wskaźniki* związane z każdym elementem listy

Listy dwukierunkowe

Każdy element listy jest obiektem składającym się z trzech atrybutów:

key – zawiera klucz elementu

next – dla danego elementu x na liście $x.next$ wskazuje na jego następnik na liście

prev – dla danego elementu x na liście $x.prev$ wskazuje na jego poprzednik

Jeżeli $x.prev = NIL$, to x nie ma poprzednika – pierwszy element listy (x jest **głową** listy)

Jeżeli $x.next = NIL$, to x nie ma następnika – ostatni element listy (x jest **ogonem** listy)

Atrybut $L.head$ wskazuje na pierwszy element listy L

Jeżeli $L.head = NIL$, to lista jest pusta

Wyszukiwanie na listach z dowiązaniami

Procedura LIST-SEARCH wyznacza pierwszy element o kluczu k na liście L. W wyniku wywołania otrzymujemy wskaźnik do tego elementu, a jeśli nie ma elementu o kluczu k, to zwracana jest wartość NIL

LIST-SEARCH (L, k)
    $x = L.head$
    while $(x.key \neq k)$ **and** $(x \neq \mathrm{NIL})$
            do $x = x.next$
    return x

Pesymistyczny czas działania procedury LIST-SEARCH na liście o n elementach wynosi $\Theta(n)$

Wstawianie do listy z dowiązaniami

Procedura LIST-INSERT przyłącza element x (którego atrybut key został wcześniej zainicjowany) na początek listy

LIST-INSERT (L, x)
    $x.next = L.head$
    if $L.head \neq \mathrm{NIL}$
        then $L.head.prev = x$
    $L.head = x$
    $x.prev = \mathrm{NIL}$

Usuwanie z listy z dowiązaniami

Procedura LIST-DELETE powoduje usunięcie elementu x z listy. Do LIST-DELETE zostaje przekazany wskaźnik do elementu x, po czym element ten zostaje ,,usunięty" z listy przez modyfikację wartości odpowiednich wskaźników

LIST-DELETE (L, x)
  if $x.prev \neq$ NIL
    then $x.prev.next = x.next$
    else $L.head = x.next$
  if $x.next \neq$ NIL
    then $x.next.prev = x.prev$

Procedura LIST-DELETE działa w czasie $O(1)$, lecz pesymistyczny koszt usunięcia elementu o zadanym kluczu wynosi $\Theta(n)$, ponieważ musimy najpierw wywołać procedurę LIST-SEARCH

Inne rodzaje list

– **listy jednokierunkowe**: w elementach listy pomijamy wskaźnik *prev*

– **listy posortowane**: kolejność elementów na liście jest zgodna z porządkiem na ich kluczach; element o najmniejszym kluczu znajduje się w głowie listy, a o największym kluczu w jej ogonie

– **listy cykliczne**: atrybut *prev* elementu w głowie wskazuje na ogon, a atrybut *next* w ogonie wskazuje na głowę

Wartownik

Z listą L związany jest element $L.nil$, który odgrywa rolę stałej NIL, ale jest **obiektem** o takich samych atrybutach jak wszystkie zwykłe elementy listy

Wartownika $L.nil$ wstawiamy pomiędzy głowę a ogon; atrybut

$L.nil.next$

wskazuje na głowę listy, a

$L.nil.prev$

wskazuje na ogon listy

Lista pusta składa się tylko z wartownika i oba atrybuty $L.nil.next$ oraz $L.nil.prev$ wskazują na $L.nil$

Wersja usuwania z wartownikiem

LIST-DELETE* (L, x)

$x.prev.next = x.next$

$x.next.prev = x.prev$