Домашня робота «АНАЛІЗ ШВИДКОДІЇ АЛГОРИТМІВ ТА ОПЕРАЦІЙ НАД СТРУКТУРАМИ ДА-НИХ»

Під керівництвом Авраменко О.А.

16.06.2016

КП-51, Варіант 16

Бабенко В.П.

Зміст

[Опис середовища 2](#_Toc454134134)

[Завдання 1: Сортування вибором у односпрямованому списку 3](#_Toc454134135)

[Псевдокод алгоритму і його асимптотичний аналіз 3](#_Toc454134136)

[Код мовою Java 4](#_Toc454134137)

[Специфікація вхідного набору даних 6](#_Toc454134138)

[Таблиця з результатами експерименту 6](#_Toc454134139)

[Графік з результатами експерименту 6](#_Toc454134140)

[Висновок 7](#_Toc454134141)

[Завдання 2: Швидке базове і швидке з медіаною з 3-х елементів сортування 8](#_Toc454134142)

[Псевдокод алгоритму і його асимптотичний аналіз 8](#_Toc454134143)

[Код мовою Java 11](#_Toc454134144)

[Специфікація вхідного набору даних 14](#_Toc454134145)

[Таблиця з результатами експерименту 14](#_Toc454134146)

[Графік з результатами експерименту 14](#_Toc454134147)

[Висновок 15](#_Toc454134148)

[Завдання 3: Видалення елемента у бінарному дереві vs в неупорядкованому масиві 16](#_Toc454134149)

[Псевдокод алгоритму і його асимптотичний аналіз 16](#_Toc454134150)

[Код мовою Java 18](#_Toc454134151)

[Специфікація вхідного набору даних 22](#_Toc454134152)

[Таблиця з результатами експерименту 22](#_Toc454134153)

[Графік з результатами експерименту 23](#_Toc454134154)

[Висновок 24](#_Toc454134155)

Розробник: Бабенко Валерій Павлович гр. КП-51

Дата створення: 16.06.2016

Дата останніх змін: 19.06.2016

# Опис середовища

Ноутбук: HP ProBook 450

Операційна система: Windows 7 Maximum (SP 1) x64

Апаратна частина: Процесор: Intel Core i-7 5500U 2.40GHz, ОЗУ: 8 Gb;

Інтегроване середовище розробки: Eclipse Luna.

Мова програмування: Java.

# Завдання 1: Сортування вибором у односпрямованому списку

## Псевдокод алгоритму і його асимптотичний аналіз

|  |
| --- |
| Pseudocode |
| Input: head, unsorted linked list;  Output: sorted linked list;  Node currentI = head (c0 = 1)  **for** i=0 to size – 1 do (c1 = size)  currentMin = currentI (c2 = size - 1)  currentJ = currentI.next (c3 = size - 1)  **for** j=i+1 to size do (c4 = )  **if** currentJ.data < currentI.data (c4 = )  then currentMin = current (c4 = )  currentJ = currentJ.next (c4 = )  **if** currentI != currentMin (c8 = size - 1)  then tmp = currentI.data (c9 = size - 1)  currentI.data = currentMin.data (c10 = size - 1)  currentMin.data = tmp (c11 = size - 1)  currentI = currentI.next (c12 = size - 1) |

У найкращому випадку ці операції не відбуваються.

Асимптотичний аналіз найкращого випадку: An^2 + Bn + C – O(n^2)

Асимптотичний аналіз найгіршого випадку: An^2 + Bn + C – O(n^2)

## Код мовою Java

|  |
| --- |
| Node.java |
| **package** ex1;  **public** **class** Node {  **int** data;  Node next;    **public** Node(**int** data){  **this**.data = data;  next = **null**;  }  } |

|  |
| --- |
| MyList.java |
| **package** ex1;  **public** **class** MyList {  **private** **int** size;  **private** Node head;    **public** MyList(){  size=0;  }    **void** addLast(**int** data){  Node newNode = **new** Node(data);  **if** (size==0){  head = newNode;  }  **else**{  Node tail = head;  **while**(tail.next!=**null**){  tail = tail.next;  }  tail.next = newNode;  }  size++;  }    **void** view(){  **if** (size==0){  System.***out***.println("My linked list is empty");  **return**;  }  Node current = head;  System.***out***.print("My linked list is: ");  **while**(current!=**null**){  System.***out***.print(current.data+" ");  current = current.next;  }  System.***out***.println("");  }    **void** selectionSort(){  Node currentI = head;  Node currentJ;  Node currentMin;  **for** (**int** i=0; i<**this**.size-1; i++){  currentMin = currentI;  currentJ = currentI.next;  **for** (**int** j=i+1; j<**this**.size; j++){  **if** (currentJ.data < currentI.data){  currentMin = currentJ;  }  currentJ = currentJ.next;  }  **if** (currentI != currentMin){  **int** tmp = currentI.data;  currentI.data = currentMin.data;  currentMin.data = tmp;  }  currentI = currentI.next;  }  }  } |

## Специфікація вхідного набору даних

При сортуванні найкращий випадок – вже відсортований масив чисел, а найгірший – відсортований масив у зворотному порядку. Також я додав випадок «середнє» - масив з випадкових чисел. Кожний випадок «тип даних – кількість» – середнє значення з декількох спроб.

## Таблиця з результатами експерименту

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Тип даних \ Кількість | 20 000 | 40 000 | 60 000 | 120 000 |
| Sort | 1.79 | 7.21 | 16.1 | 67.8 |
| Sort in reverse | 1.83 | 7.22 | 16.5 | 69.0 |
| Random | 1.84 | 7.18 | 16.4 | 68.8 |

## Графік з результатами експерименту

## Висновок

Результати виявилися очікуваними, а значить і правильними. Час, витрачений на сортування вже відсортованого масиву лише трохи кращий за результати з випадкових чисел і найгіршого випадку. Це все через 3 операції присвоєння чи операцію swap, яка при сортуванні вже відсортованого масиву не використовується.

# Завдання 2: Швидке базове і швидке з медіаною з 3-х елементів сортування

## Псевдокод алгоритму і його асимптотичний аналіз

|  |
| --- |
| Pseudocode QuickSort |
| quickSort  Input: array arr[]; lo – lower element, hi – high element (0, array size -1 in start);  Output: sorted array;  **if** lo < hi then  q = partitionHoare(arr, lo, hi)  quickSort(arr, lo, q-1);  quickSort(arr, q+1, hi);    partitionHoare  Input: array arr[]; lo – lower element, hi – high element;  Output: element to partition;  pivot = arr[lo];  i = lo;  j = hi + 1;  **while** (**true**) do  **while** ++i <= hi and arr[i] < pivot  ;  **while** --j >= lo and arr[j] > pivot  ;  **if** i < j then  swap (arr, i, j);  **else**  **break**;    swap (arr, lo, j);  **return** j;  swap  Input: array arr[]; lo – lower element, hi – high element;  Output: swap elements;  temp = arr[lo];  arr[lo] = arr[hi];  arr[hi] = temp; |

Асимптотичний аналіз найкращого випадку: O(n\*log(n))

Асимптотичний аналіз найгіршого випадку: O(n^2)

Асимптотичний аналіз середнього випадку: O(n\*log(n))

|  |
| --- |
| Pseudocode MedianOf3 |
| quickSort  Input: array arr[]; left – left element, right – right element (0, array size -1 in start);  Output: sorted array;  size = right - left + 1;  **if** size <= 3  *manualSort*(arr, left, right);    **else**  **int** q = *partitionMedian*(arr, left, right);  *quickSort*(arr, left, q - 1);  *quickSort*(arr, q + 1, right);    medianOf3  Input: array arr[]; left – left element, right – right element;  Output: array with sort 1 element;  center = (left + right) / 2;  **if** arr[left] > arr[center]  *swap*(arr, left, center)  **if** arr[left] > arr[right]  *swap*(arr, left, right)    **if** arr[center] > arr[right]  *swap*(arr, center, right)    *swap*(arr, center, right - 1)  **return** arr[right - 1]  swap  Input: array arr[]; in1 – first element, in2 – second element;  Output: sorted array;  temp = arr[in1];  arr[in1] = arr[in2];  arr[in2] = temp;    Input: array arr[]; left – left element, right – right element;  Output: element to partition;  median = *medianOf3*(arr, left, right);  i = left;  j = right - 1;  **while** (**true**) do  **while** arr[++i] < median  ;  **While** arr[--j] > median  ;  **if** i >= j then  **break**  **else**  *swap*(arr, i, j)    *swap*(arr, i, right - 1)  **return** i  manual  Input: array arr[]; left – left element, right – right element;  size = right - left + 1;  **if** size <= 1 then  **return**;    **if** size == 2 then  **if** arr[left] > arr[right] then  *swap*(arr, left, right)  **return**;  **else**  **if** arr[left] > arr[right - 1] then  *swap*(arr, left, right - 1)    **if** arr[left] > arr[right] then  *swap*(arr, left, right)    **if** arr[right - 1] > arr[right] then  *swap*(arr, right - 1, right); |

Асимптотичний аналіз найкращого випадку: O(n\*log(n))

Асимптотичний аналіз найгіршого випадку: O(n^2)

Асимптотичний аналіз середнього випадку: O(n\*log(n))

## Код мовою Java

|  |
| --- |
| QuickSort.java |
| **package** ex2;  **public** **class** QuickSort {  **private** **void** quickSort(**int** [] arr, **int** lo, **int** hi){  **if** (lo < hi){  **int** q = partitionHoare(arr, lo, hi);  quickSort(arr, lo, q-1);  quickSort(arr, q+1, hi);  }  }    **public** **void** sort(**int** [] arr){  quickSort(arr, 0, arr.length-1);  }    **private** **int** partitionHoare(**int** [] arr, **int** lo, **int** hi){  **int** pivot = arr[lo];  **int** i = lo;  **int** j = hi + 1;  **while** (**true**){  **while** (++i <= hi && arr[i] < pivot)  ;  **while** (--j >= lo && arr[j] > pivot)  ;  **if** (i < j){  swap (arr, i, j);  }  **else**{  **break**;  }  }  swap (arr, lo, j);  **return** j;  }    **private** **void** swap(**int** [] arr, **int** lo, **int** hi){  **int** temp = arr[lo];  arr[lo] = arr[hi];  arr[hi] = temp;  }  } |

|  |
| --- |
| MedianOf3.java |
| **package** ex2;  **public** **class** MedianOf3 {  **public** **void** sort(**int**[] arr) {  *quickSort*(arr, 0, arr.length - 1);  }  **public** **static** **void** quickSort(**int**[] arr, **int** left, **int** right) {  **int** size = right - left + 1;  **if** (size <= 3){  *manualSort*(arr, left, right);  }  **else**{  **int** q = *partitionMedian*(arr, left, right);  *quickSort*(arr, left, q - 1);  *quickSort*(arr, q + 1, right);  }  }  **public** **static** **int** medianOf3(**int**[] arr, **int** left, **int** right) {  **int** center = (left + right) / 2;  **if** (arr[left] > arr[center]){  *swap*(arr, left, center);  }  **if** (arr[left] > arr[right]){  *swap*(arr, left, right);  }    **if** (arr[center] > arr[right]){  *swap*(arr, center, right);  }    *swap*(arr, center, right - 1);  **return** arr[right - 1];  }  **public** **static** **void** swap(**int**[] arr, **int** in1, **int** in2) {  **int** temp = arr[in1];  arr[in1] = arr[in2];  arr[in2] = temp;  }  **public** **static** **int** partitionMedian(**int**[] arr, **int** left, **int** right) {  **double** median = *medianOf3*(arr, left, right);  **int** i = left;  **int** j = right - 1;  **while** (**true**) {  **while** (arr[++i] < median)  ;  **while** (arr[--j] > median)  ;  **if** (i >= j){  **break**;  }  **else**{  *swap*(arr, i, j);  }  }  *swap*(arr, i, right - 1);  **return** i;  }  **public** **static** **void** manualSort(**int**[] arr, **int** left, **int** right) {  **int** size = right - left + 1;  **if** (size <= 1){  **return**;  }  **if** (size == 2){  **if** (arr[left] > arr[right]){  *swap*(arr, left, right);  }  **return**;  }  **else** {  **if** (arr[left] > arr[right - 1]){  *swap*(arr, left, right - 1);  }  **if** (arr[left] > arr[right]){  *swap*(arr, left, right);  }  **if** (arr[right - 1] > arr[right]){  *swap*(arr, right - 1, right);  }  }  }  } |

## Специфікація вхідного набору даних

При швидкому сортуванні найгірший випадок – це випадок, коли масив буде розбиватися на підмасиви 1 і n-1. Найкращий – коли масив буде розбиватися максимально рівномірно.

Швидке сортування із медіаною із 3-х елементів вибирає медіану із першого, середнього і останнього елементів. Це означає, що вірогідність найгіршого випадку буде зменшена.

## Таблиця з результатами експерименту

|  |  |  |  |
| --- | --- | --- | --- |
| Тип даних \ Кількість | 10 000 | 50 000 | 100 000 |
| Найкращий (quick) | 0,0004482 | 0,0026907 | 0,0042042 |
| Середнє | 0,0006014 | 0,0031303 | 0,0063861 |
| Найгірший | 0,0021408 | 0,0038967 | 0,0089192 |
| Найкращий (median) | 0,0002105 | 0,0022093 | 0,0036503 |
| Середнє | 0,0007195 | 0,0036682 | 0,0075075 |
| Найгірший | 0,0020946 | 0,0039547 | 0,0087192 |

## Графік з результатами експерименту

## Висновок

В загальному випадку сортування медіаною поступається в часі базовому швидкому, але в найкращому випадку медіана працює швидше. Найгірший випадок для медіани розв’язується за трохи менший час за базове сортування.

# Завдання 3: Видалення елемента у бінарному дереві vs в неупорядкованому масиві

## Псевдокод алгоритму і його асимптотичний аналіз

|  |
| --- |
| Pseudocode Array |
| Input: Array arr, size;  Ouput: size of Array;  **for** index=0 to arr.length do  **if** id = arr[index].id then  **for** j=index to arr.length-1 do  arr[j] = arr[j+1]  **return** size-1  **return** size |

Асимптотичний аналіз: O(n)

|  |
| --- |
| Pseudocode Tree |
| Input: MyTree; root; toDel – node to delete;  Output: element, that was removed  x=**null**, y=**null**;  **if** toDel.left = **null** or toDel.right = **null**  y = toDel;  **else**  y = toDel.treeSuccessor()    **if** y.left != **null then**  x = y.left  **else**  x = y.right      **if** x!=**null then**  x.parent = y.parent      **if** y.parent = **null** then  root = x;  **else** **if** y == y.parent.left then  y.parent.left = x    **else**  y.parent.right = x      **if** y!=toDel then  toDel.id = y.id;  toDel.data = y.data;    **return** y.data; |

Асимптотичний аналіз: O(log(n))

Асимптотичний аналіз найгіршого випадку: O(n)

## Код мовою Java

|  |
| --- |
| Element.java |
| **package** ex3;  **public** **class** Element {  **private** **int** id;  **private** **int** data;    **public** Element(**int** id, **int** data){  **this**.data = data;  **this**.id = id;  }    **public** **int** getData(){  **return** **this**.data;  }    **public** **int** getId(){  **return** **this**.id;  }  } |

|  |
| --- |
| TreeNode.java |
| **package** ex3;  **public** **class** TreeNode {  **public** **int** id;  **public** **int** data;  **public** TreeNode right;  **public** TreeNode left;  **public** TreeNode parent;  **public** TreeNode(**int** id, **int** data) {  **this**.data = data;  **this**.id = id;  }  **public** TreeNode(**int** id, **int** data, TreeNode parent) {  **this**.data = data;  **this**.id = id;  **this**.parent = parent;  }    **public** **void** printNode(){  System.***out***.print(**this**.id+") ");  System.***out***.print(**this**.data+"; ");  }    **public** **void** postOrderTreeWalk(){  **if** (**this**.left != **null**){  left.postOrderTreeWalk();  }  **if** (**this**.right != **null**){  right.postOrderTreeWalk();  }  **this**.printNode();  }    **public** TreeNode treeSearch(**int** key){  **if** (key < **this**.id && left != **null**){  **return** left.treeSearch(key);  }  **if**(key > **this**.id && right != **null**){  **return** right.treeSearch(key);  }  **if** (key == **this**.id){  **return** **this**;  }  **return** **null**;  }    **public** TreeNode treeMin(TreeNode node){  **if** (node!=**null**){  **while** (node.left != **null**){  node = node.left;  }  **return** node;  }  **else**{  **return** **null**;  }  }    **public** TreeNode treeSuccessor(){  **if** (**this**.right != **null**){  **return** treeMin(**this**.right);  }  **return** **null**;  }  } |

|  |
| --- |
| Main.java |
| **private** **static** **int** removeArray(Element [] arr, **int** id, **int** size){  **for** (**int** index=0; index<arr.length; index++){  **if** (id == arr[index].getId()){  **for**(**int** j=index; j<arr.length-1; j++){  arr[j] = arr[j+1];  }  **return** size-1;  }  }  System.***out***.print("Element with this id: '"+id+"' is absent here");  **return** size;  }    **private** **static** **void** printArray(Element [] arr, **int** size){  System.***out***.print("There are numbers in Array:\n");  **for** (**int** i=0; i<size; i++){  System.***out***.print(arr[i].getId()+") "+arr[i].getData()+"; ");  }  System.***out***.print("\n");  } |

|  |
| --- |
| MyTree.java |
| **package** ex3;  **public** **class** MyTree {  **private** TreeNode root;  **public** MyTree(){}  **public** **void** traverse(){  **if** (root!=**null**){  System.***out***.println("There are numbers in Tree:");  root.postOrderTreeWalk();  System.***out***.println("");  **return**;  }  System.***out***.println("This tree is empty\n");  **return**;  }  **public** **int** insert(**int** id, **int** value) {  **if** (root == **null**){  root = **new** TreeNode(id, value);  **return** 0;  }    TreeNode oldNode = root.treeSearch(id);  **if** (oldNode != **null**){  **int** old = oldNode.data;  oldNode.data = value;  **return** old;  }    TreeNode z = **new** TreeNode(id, value);  TreeNode y = **null**;  TreeNode x = root;  **while**(x != **null**){  y = x;  **if** (z.id < x.id){  x = x.left;  }  **else** {  x = x.right;  }  }    z.parent = y;  **if**(z.id < y.id){  y.left = z;  }  **else**{  y.right = z;  }    **return** 0;  }  **public** **int** remove(**int** id){  TreeNode current = root.treeSearch(id);  **return** remove(current);  }  **private** **int** remove(TreeNode toDel){  TreeNode x=**null**, y=**null**;  **if** (toDel.left == **null** || toDel.right == **null**){  y = toDel;  }  **else** {  y = toDel.treeSuccessor();  }    **if** (y.left != **null**){  x = y.left;  }  **else**{  x = y.right;  }    **if** (x!=**null**){  x.parent = y.parent;  }    **if** (y.parent == **null**){  **this**.root = x;  }  **else** **if**(y == y.parent.left){  y.parent.left = x;  }  **else**{  y.parent.right = x;  }    **if** (y!=toDel){  toDel.id = y.id;  toDel.data = y.data;  }  **return** y.data;  }  } |

## Специфікація вхідного набору даних

При видаленні із дерева найкращий випадок – це видалення найбільш лівого елемента. Це через те, що при post-order пошуку ми знайдемо цей елемент першим, а його видалення – видалення без дочірніх вузлів. Найгірший випадок – видалення кореня. Ми знайдемо його останнім, а tree-successor буде знаходитися максимально далеко. Видалення у цьому разі – видалення вузла з двома дочірніми.

При видаленні із масиву найкращий випадок – видалення першого елемента, а найгірший – останнього. В першому разі ми будемо робити максимальну кількість присвоєнь, а в останньому – викликати метод на отримання даних і порівнювати (див. код).

Середній варіант – видалення з довільного місця у масиві і дереві.

## Таблиця з результатами експерименту

|  |  |  |  |
| --- | --- | --- | --- |
| Тип даних \ Кількість | 1 000 | 5 000 | 9 000 |
| Найбільш лівий (найкращий) | 0,00000599 | 0,00000755 | 0,00000834 |
| Корінь (найгірший) | 0,00002865 | 0,00005517 | 0,00008382 |
| Середнє | 0,00001155 | 0,00003721 | 0,00004796 |
| Початок масиву  (найкращий) | 0,00002501 | 0,00013314 | 0,00018606 |
| Кінець масиву (найгірший) | 0,00009879 | 0,00025317 | 0,00062372 |
| Середнє | 0,00004790 | 0,00024889 | 0,00037419 |

## Графік з результатами експерименту

## Висновок

Операція видалення у дереві працює швидше за операцію видалення у неупорядкованому масиві. Це поясняється тим, що по дереву легше шукати елемент, який потрібно викинути, аніж по масиву, в дереві змінюються лише вказівники, а в масиві ми маємо справу з реальними елементами і їх багатократними порівняннями.