ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к курсовому проекту по дисциплине

«Структуры и алгоритмы обработки данных»

на тему

AA-tree

|  |  |
| --- | --- |
| Выполнил студент | Ниевин Евгений Вячеславович |
|  | Ф.И.О. |

|  |  |
| --- | --- |
| Группы | ИВ-021 |
|  |  |

|  |  |  |
| --- | --- | --- |
| Работу принял |  | ст. преп. Кафедры ВС Д. М. Берлизов |
|  | подпись |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Защищена |  | Оценка |  |
|  |  |  |  |

Новосибирск – 2021

**Оглавление**

[**Введение** 2](#_Toc87331522)

[**Описание структуры** 3](#_Toc87331523)

[**Основные операции над структурой** 4](#_Toc87331524)

[**Сравнение структуры АА-tree и RB-tree** 11](#_Toc87331525)

[**Эффективность** 13](#_Toc87331526)

[**Заключение** 14](#_Toc87331527)

[**Список используемых источников** 16](#_Toc87331528)

[**Листинг программы** 17](#_Toc87331529)

# **Введение**

AA-tree является формой сбалансированного BS-tree и названо по первым буквам имени и фамилии изобретателя, Арне Андерсона, который впервые предложил данную модификацию RB-tree в 1993 году. В своей работе Андерcон приводит простое правило: “К одной вершине можно присоединить другую вершину того же уровня, но только одну и только справа”.

AA-tree можно рассматривать как RB-tree с двумя модификациями: только правый дочерний элемент может быть красным; и вместо использования красного/черного цветов каждый узел хранит целое число, обозначающее его уровень, и правый дочерний элемент концептуально считается красным, если он находится на том же уровне.

# **Описание структуры**

**Свойства АА-tree:**

1. Уровень каждого листа равен 1.
2. Уровень каждого левого ребенка на один меньше, чем у родителя.
3. Уровень каждого правого ребенка равен или на один меньше, чем у родителя.
4. Уровень каждого внука строго меньше, чем у его прародителя.
5. Каждая вершина с уровнем больше 1 имеет двоих детей.

Ссылка, где уровень дочернего элемента равен уровню его родителя, называется горизонтальной ссылкой и аналогична красной ссылке в RB-tree.

Разрешены отдельные правые горизонтальные ссылки, но запрещены последовательные; все левые горизонтальные ссылки запрещены.

Это более строгие ограничения, чем аналогичные ограничения для RB-tree, в результате чего повторная балансировка АА-tree процедурно намного проще, чем повторная балансировка RB-tree.

# **Основные операции над структурой**

Вставки и удаления могут временно привести к тому, что AA-tree станет несбалансированным. Для восстановления баланса нужны всего две операции: “skew” и “split”.

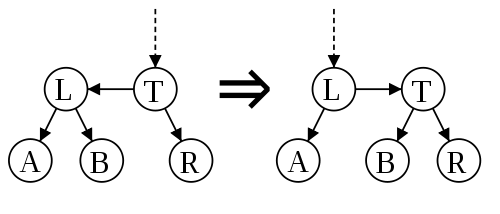
Skew - это поворот вправо для замены поддерева, содержащего левую горизонтальную ссылку на поддерево, содержащее правую горизонтальную ссылку.

Split - это левый поворот и повышение уровня для замены поддерева, содержащего две или более последовательных правых горизонтальных ссылки одной, содержащей на две последовательных правых горизонтальных ссылки меньше.

Реализация вставки и удаления с сохранением баланса упрощается за счет использования операций “skew” и “split” для изменения дерева только в случае необходимости, вместо того, чтобы заставлять их вызывающие стороны решать, перекосить или разделить.

## **Операция “skew”**

Устранение левой связи на одном уровне.



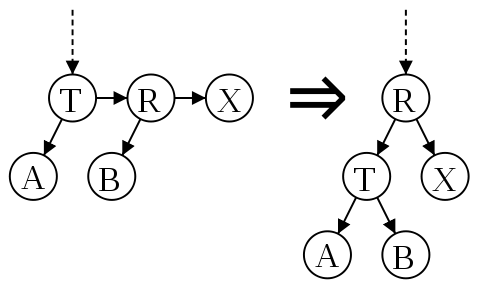
Данная операция устраняет горизонтальную левую связь при помощи вращения узла вправо каждый раз, когда левая горизонтальная связь найдена.

На рисунке горизонтальная стрелка обозначает одноуровневую связь, наклонная – связь между разными уровнями.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | **struct** aatree\* **aatree\_skew**(**struct** aatree\* tree)  {  **if** ((tree == NULL) || (tree->left == NULL))  {  **return** tree;  }  **if** (tree->level != tree->left->level)  {  **return** tree;  }  **struct** aatree\* left = tree->left;  tree->left = left->right;  left->right = tree;  tree = left;  **return** tree;  } |

## **Операция “split”**

Устранение двух правых связей на одном уровне.



Данная операция устраняет две последовательные правые горизонтальные связи при помощи вращения узлов влево и увеличение среднего узла на единицу.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | **struct** aatree\* **aatree\_split**(**struct** aatree\* tree)  {  **if** ((tree == NULL) || (tree->right->right == NULL))  {  **return** tree;  }  **if** (tree->right->right->level != tree->level)  {  **return** tree;  }  **struct** aatree\* right = tree->right;  tree->right = right->left;  right->left = tree;  tree = right;  tree->level++;  **return** tree;  } |

## **Операция “insert”**

Вставка нового элемента происходит как в обычном дереве поиска, только на пути вверх необходимо делать ребалансировку, используя “skew” и “split”.

Вставка может нарушить:

1. Правило “левого потомка” – исправляется операцией “skew”
2. Правило “двух правых потомков” – исправляется с помощью операции “split”.
3. Вставка может повлечь серию операций “skew” и “split”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | **struct** aatree\* **aatree\_insert**(**struct** aatree\* tree, **int** key)  {  **if** (tree == NULL)  {  tree = aatree\_init(key);  **return** tree;  }  **if** (key < tree->key)  {  tree->left = aatree\_insert(tree->left, key);  }  **else** **if** (key > tree->key)  {  tree->right = aatree\_insert(tree->right, key);  }  **else**  {  **return** tree;  }  tree = aatree\_skew(tree);  tree = aatree\_split(tree);  **return** tree;  } |

## **Операция “delete”**

Как и в большинстве сбалансированных бинарных деревьев, удаление внутренней вершины можно заменить на удаление листа, если заменить внутреннюю вершину на ее ближайшего предшественника или преемника.

Для восстановления баланса дерева можно использовать следующие шаги:

1. При необходимости уменьшить уровень (если дочерние элементы находятся на два уровня ниже).
2. Использовать операцию skew.
3. Использовать операцию split.

После удаления узла, необходимо подняться вверх, начиная с родителя фактически удаленного узла и выполнить балансировку.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | **struct** aatree\* **aatree\_delete**(**struct** aatree\* tree, **int** key)  {  **if** (tree == NULL)  {  **return** tree;  }  **struct** aatree\* deleted = NULL;  **struct** aatree\* last = tree;  **if** (key < tree->key)  {  tree->left = aatree\_delete(tree->left, key);  }  **else**  {  deleted = tree;  tree->right = aatree\_delete(tree->right, key);  }  **if** ((tree == last) && (deleted != NULL) && (key == deleted->key))  {  deleted->key = tree->key;  deleted = NULL;  tree = tree->right;  free(last);  }  **else** **if** ((tree->left == NULL) || (tree->right == NULL))  {  **return** tree;  }  **else** **if** ((tree->left->level < tree->level - **1**) || (tree->right->level < tree->level - **1**))  {  tree->level--;  **if** (tree->right->level > tree->level)  {  tree->right->level = tree->level;  }  tree = aatree\_skew(tree);  tree->right = aatree\_skew(tree->right);  tree->right->right = aatree\_skew(tree->right->right);  tree = aatree\_split(tree);  tree->right = aatree\_split(tree->right);  }  **return** tree;  } |

## **Операция “lookup”**

Операция поиска в AA-tree выполняется аналогично поиску в BS-tree.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | **struct** aatree\* **aatree\_lookup**(**struct** aatree\* tree, **int** key)  {  **while** (tree != NULL)  {  **if** (key == tree->key)  {  **return** tree;  }  **else** **if** (key < tree->key)  {  tree = tree->left;  }  **else**  {  tree = tree->right;  }  }  **return** tree;  } |

## **Операция “free”**

Данная операция рекурсивно освобождает память

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **void** **aatree\_free**(**struct** aatree\* tree)  {  **if** (tree == NULL)  {  **return**;  }  aatree\_free(tree->left);  aatree\_free(tree->right);  free(tree);  } |

## **Операции “min” и “max”**

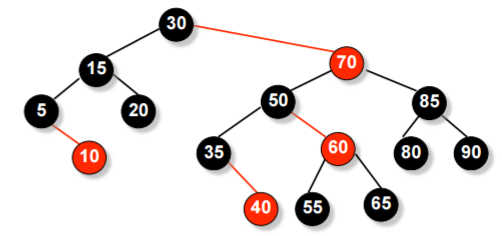
Данные операции аналогичны соответствующим операциям в BS-tree.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | **struct** aatree\* **aatree\_min**(**struct** aatree\* tree)  {  **if** (tree == NULL)  {  **return** tree;  }  **while** (tree->left != NULL)  {  tree = tree->left;  }  **return** tree;  } |

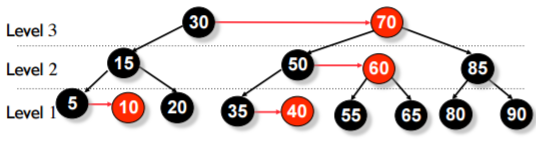
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | **struct** aatree\* **aatree\_max**(**struct** aatree\* tree)  {  **if** (tree == NULL)  {  **return** tree;  }  **while** (tree->right != NULL)  {  tree = tree->right;  }  **return** tree;  } |

# **Сравнение структуры АА-tree и RB-tree**

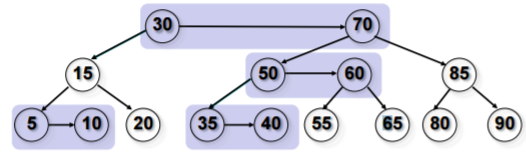
В отличие от RB-tree, к одной вершине можно присоединить вершину только того же уровня, только одну и только справа (другими словами, красные вершины могут быть добавлены только в качестве правого ребенка). На картинке ниже представлен пример RB-tree.



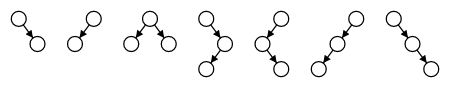
Теперь рассмотрим то же дерево, но с информацией об уровне каждой вершине. Горизонтальные ребра обозначают связи между ребрами одного уровня.



На практике в AA-tree вместо значения цвета для балансировки дерева в вершине хранится информация только об ее уровне.



Для поддержки баланса RB-tree необходимо обрабатывать 7 различных вариантов расположения вершин:



В АА-tree из-за строгих ограничений необходимо обрабатывать только два вида возможных расположений вершин, чтобы проверить соблюдается ли главное правило «одна правая горизонтальная связь». То есть мы должны проверить нет ли левой горизонтальной связи, как на левом рисунке ниже и нет ли двух последовательных правых горизонтальных связей, как на правом рисунке ниже:



**Чем лучше АА-tree?**

Реализация и количество случаев ротации в красно-черных деревьях сложны. АА-tree упрощают алгоритм.

АА-tree более “плоское”, что ускоряет время работы операций поиска элемента по ключу.

# **Эффективность**

Скорость работы AA-tree эквивалентна скорости работы RB-tree, но так как в реализации вместо цвета обычно хранят «уровень» вершины, дополнительные расходы по памяти достигают байта.

В то время как АА-tree делает больше вращений, чем красно-черное дерево, более простые алгоритмы, как правило, работают быстрее, и все это уравновешивается, чтобы привести к аналогичной производительности.

RB-tree более стабильно по своим характеристикам, чем АА-tree, но АА-tree имеет тенденцию быть более плоским, что приводит к немного более быстрому времени поиска.

Так как AA-tree сохраняет структуру RB-tree, то все операции происходят за O(log n).

|  |  |  |
| --- | --- | --- |
| Операция | Средний случай | Худший случай |
| ***lookup*** | *O(log n)* | *O(log n)* |
| ***insert*** | *O(log n)* | *O(log n)* |
| ***delete*** | *O(log n)* | *O(log n)* |
| ***min*** | *O(log n)* | *O(log n)* |
| ***max*** | *O(log n)* | *O(log n)* |

# **Экспериментальное сравнение сложности операций RB-tree и AA-tree**

# **(на упорядоченных данных)**

***10 тыс. элементов:***

|  |  |  |
| --- | --- | --- |
| Операция | AA-tree | RB-tree |
| Insert, c\*106 | 3277 | 1970 |
| Lookup, c\*106 | 735 | 1112 |
| Delete, c\*106 | 1201 | 803 |

***50 тыс. элементов:***

|  |  |  |
| --- | --- | --- |
| Операция | AA-tree | RB-tree |
| Insert, c\*106 | 20050 | 12243 |
| Lookup, c\*106 | 5263 | 7030 |
| Delete, c\*106 | 8663 | 5475 |

***100 тыс. элементов:***

|  |  |  |
| --- | --- | --- |
| Операция | AA-tree | RB-tree |
| Insert, c\*106 | 44392 | 26812 |
| Lookup, c\*106 | 11986 | 16589 |
| Delete, c\*106 | 20101 | 12194 |

Из приведённых выше данных можно сказать, что AA-tree эффективнее использовать, если количество операций “lookup” при работе со структурой данных превышает количество операций “delete” и “insert”.

# **Заключение**

В результате работы была разработана и исследована структура АА-tree, проведено сравнение с RB-tree.

Как и было сказано, все операции АА-tree эквиваленты операциям RB-tree и выполняются за О(log n).

Главным преимуществом АА-tree перед RB-tree является его простые алгоритмы вставки, удаления и балансировки дерева.

АА-tree делает больше вращений, нежели RB-tree, но более простые алгоритмы работают быстрее, и все это уравновешивается, что приводит к аналогичной производительности.

# **Список используемых источников**

1. *A. Andersson.* Balanced search trees made simple. // In Proc. Workshop on Algorithms and Data Structures. – 1993. – С. 60-71.
2. *A. Andersson*, *S. Nilsson*. Efficient implementation of suffix trees. // Software. Practice and Experience. – 1995. – С. 129-141.
3. AA-Tree или простое бинарное дерево – <https://habr.com/ru/post/110212/>
4. АА-дерево – <http://www.proteus2001.narod.ru/gen/txt/8/aa.html>
5. Introduction to AA trees - <https://iq.opengenus.org/aa-trees/>
6. <https://neerc.ifmo.ru/wiki/index.php?title=AA-%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE>
7. <https://en.wikipedia.org/wiki/AA_tree>

# **Листинг программы**

“AAtree.c”

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173 | #include "AAtree.h"  #include <stdio.h>  #include <stdlib.h>  **struct** aatree\* **aatree\_init**(**int** key)  {  **struct** aatree\* tree = (**struct** aatree\*)malloc(**sizeof**(**struct** aatree));  tree->key = key;  tree->level = **1**;  tree->left = NULL;  tree->right = NULL;  **return** tree;  }  **struct** aatree\* **aatree\_lookup**(**struct** aatree\* tree, **int** key)  {  **while** (tree != NULL)  {  **if** (key == tree->key)  {  **return** tree;  }  **else** **if** (key < tree->key)  {  tree = tree->left;  }  **else**  {  tree = tree->right;  }  }  **return** tree;  }  **struct** aatree\* **aatree\_min**(**struct** aatree\* tree)  {  **if** (tree == NULL)  {  **return** tree;  }  **while** (tree->left != NULL)  {  tree = tree->left;  }  **return** tree;  }  **struct** aatree\* **aatree\_max**(**struct** aatree\* tree)  {  **if** (tree == NULL)  {  **return** tree;  }  **while** (tree->right != NULL)  {  tree = tree->right;  }  **return** tree;  }  **void** **aatree\_free**(**struct** aatree\* tree)  {  **if** (tree == NULL)  {  **return**;  }  aatree\_free(tree->left);  aatree\_free(tree->right);  free(tree);  }  **struct** aatree\* **aatree\_skew**(**struct** aatree\* tree)  {  **if** ((tree == NULL) || (tree->left == NULL))  {  **return** tree;  }  **if** (tree->level != tree->left->level)  {  **return** tree;  }  **struct** aatree\* left = tree->left;  tree->left = left->right;  left->right = tree;  tree = left;  **return** tree;  }  **struct** aatree\* **aatree\_split**(**struct** aatree\* tree)  {  **if** ((tree == NULL) || (tree->right->right == NULL))  {  **return** tree;  }  **if** (tree->right->right->level != tree->level)  {  **return** tree;  }  **struct** aatree\* right = tree->right;  tree->right = right->left;  right->left = tree;  tree = right;  tree->level++;  **return** tree;  }  **struct** aatree\* **aatree\_insert**(**struct** aatree\* tree, **int** key)  {  **if** (tree == NULL)  {  tree = aatree\_init(key);  **return** tree;  }  **if** (key < tree->key)  {  tree->left = aatree\_insert(tree->left, key);  }  **else** **if** (key > tree->key)  {  tree->right = aatree\_insert(tree->right, key);  }  **else**  {  **return** tree;  }  tree = aatree\_skew(tree);  tree = aatree\_split(tree);  **return** tree;  }  **struct** aatree\* **aatree\_delete**(**struct** aatree\* tree, **int** key)  {  **if** (tree == NULL)  {  **return** tree;  }  **struct** aatree\* deleted = NULL;  **struct** aatree\* last = tree;  **if** (key < tree->key)  {  tree->left = aatree\_delete(tree->left, key);  }  **else**  {  deleted = tree;  tree->right = aatree\_delete(tree->right, key);  }  **if** ((tree == last) && (deleted != NULL) && (key == deleted->key))  {  deleted->key = tree->key;  deleted = NULL;  tree = tree->right;  free(last);  }  **else** **if** ((tree->left == NULL) || (tree->right == NULL))  {  **return** tree;  }  **else** **if** ((tree->left->level < tree->level - **1**) || (tree->right->level < tree->level - **1**))  {  tree->level--;  **if** (tree->right->level > tree->level)  {  tree->right->level = tree->level;  }  tree = aatree\_skew(tree);  tree->right = aatree\_skew(tree->right);  tree->right->right = aatree\_skew(tree->right->right);  tree = aatree\_split(tree);  tree->right = aatree\_split(tree->right);  }  **return** tree;  } |

“AAtree.h”

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | #pragma once  **struct** aatree  {  **int** key;  **int** level;  **struct** aatree\* left;  **struct** aatree\* right;  };  **struct** aatree\* **aatree\_init**(**int** key);  **struct** aatree\* **aatree\_lookup**(**struct** aatree\* tree, **int** key);  **struct** aatree\* **aatree\_min**(**struct** aatree\* tree);  **struct** aatree\* **aatree\_max**(**struct** aatree\* tree);  **void** **aatree\_free**(**struct** aatree\* tree);  **struct** aatree\* **aatree\_skew**(**struct** aatree\* tree);  **struct** aatree\* **aatree\_split**(**struct** aatree\* tree);  **struct** aatree\* **aatree\_insert**(**struct** aatree\* tree, **int** key);  **struct** aatree\* **aatree\_delete**(**struct** aatree\* tree, **int** key); |

“main.c”

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76 | #include "AAtree.h"  #include <stdio.h>  **void** **printMenu**()  {  printf("1. Добавить элемент**\n**");  printf("2. Удалить элемент**\n**");  printf("3. Элемент по ключу**\n**");  printf("4. Минимальный элемент**\n**");  printf("5. Максимальный элемент**\n**");  printf("6. Выход**\n**");  }  **int** **main**()  {  **struct** aatree\* root = NULL;  **int** item = -**1**;  **int** key;  **while** (item != **7**)  {  **struct** aatree\* node = NULL;  printMenu();  scanf("%d", &item);  **switch** (item)  {  **case** **1**:  printf("Введите ключ:**\n**");  scanf("%d", &key);  root = aatree\_insert(root, key);  **break**;  **case** **2**:  printf("Введите ключ:**\n**");  scanf("%d", &key);  root = aatree\_delete(root, key);  **break**;  **case** **3**:  printf("Введите ключ:**\n**");  scanf("%d", &key);  node = aatree\_lookup(root, key);  **if** (node == NULL)  {  printf("Элемент не найден**\n**");  }  **else**  {  printf("Элемент найден**\n**");  }  **break**;  **case** **4**:  node = aatree\_min(root);  **if** (node == NULL)  {  printf("Дерево пустое**\n**");  }  **else**  {  printf("Минимальный элемент: %d**\n**", node->key);  }  **break**;  **case** **5**:  node = aatree\_max(root);  **if** (node == NULL)  {  printf("Дерево пустое**\n**");  }  **else**  {  printf("Максимальный элемент: %d**\n**", node->key);  }  **break**;  **case** **6**:  aatree\_free(root);  **return** **0**;  }  }  } |

“makefile”

|  |  |
| --- | --- |
| 1  2  3  4 | **all:** AAtree.c main.c  gcc -Wall -o AAtree $^  **clean:**  rm AAtree |