Деревья

М. А. Ложников

28 ноября 2019 г.
Ревизия: 0

Это предварительная невыверенная версия. Читайте на свой страх и риск.

1 Простейшее несбалансированное бинарное дерево поиска

Вместо того, чтобы давать строгое определение этой структуры данных, разберём пример. А именно, решим задачу

Задача 1. *Построить бинарное дерево поиска из массива целых чисел.*

Введём структуру, описывающую узел бинарного дерева.

```
struct TreeNode {
  // Значение, которое хранится в узле дерева.
  int value;
  // Указатель на родителя этого узла (nullptr, если отсутствует).
  TreeNode* parent;
  // Указатель на левого потомка данного узла (nullptr, если отсутствует).
  TreeNode* left;
  // Указатель на левого потомка данного узла (nullptr, если отсутствует).
  TreeNode* right;

  TreeNode(int value,
           TreeNode* parent = nullptr,
           TreeNode* left = nullptr,
           TreeNode* right = nullptr) :
      value(value),
      parent(parent),
      left(left),
      right(right)
  { }

  ~TreeNode() {
    if (left)
      delete left;
    if (right)
      delete right;
  }
};
```

Узел дерева, не имеющий родителя называется корнем дерева. У каждого (непустого) дерева ровно один корень. Узлы дерева, не имеющие потомков называются листьями дерева. Для каждого узла дерева должен существовать путь, связывающий данный узел с корнем дерева. Кроме того, положим, что значение, лежащее в каждом левом потомке должно быть меньше или равно значению, лежащему в узле, а значение, лежащее в каждом правом потомке должно быть строго больше значения, лежащего в узле. Напишем решение поставленной задачи

```
TreeNode* AddElem(TreeNode* root, int elem) {
  if (root == nullptr) // Дерево было пустое, создаём новое.
    return new TreeNode(elem);

  TreeNode* node = root;

  /* Этот цикл завершится тогда, когда в дерево будет добавлен новый элемент. */
  while (true) {
    /* Определяем, в какую ветвь нужно добавить новый элемент. */
    if (elem < node->value) {
      if (node->left == nullptr) {
        /* Вставляем элемент в дерево. */
        node->left = new TreeNode(elem, node);
        return root;
      }
      /* Спускаемся дальше по ветви. */
      node = node->left;
    }
    else {
      if (node->right == nullptr) {
        /* Вставляем элемент в дерево. */
        node->right = new TreeNode(elem, node);
        return root;
      }
      /* Спускаемся дальше по ветви. */
      node = node->right;
    }
  }
  return root;
}

TreeNode* BuildTree(int* elems, int count) {
  /* Объявляем пустое дерево. */
  TreeNode* root = nullptr;

  for (int i = 0; i < count ; i++) {
    /* Добавляем в дерево по одному элементу. */
    root = AddElem(root, elems[i]);
  }

  return root;
}
```

2 Класс, работающий с бинарным деревом

Напишем класс, реализующий методы добавления элемента в дерево, поиска элемента, а также итераторы для этого класса.

2.1 Базовые методы класса

```
class BinaryTree {
 private:
  // Значение, лежащее в текущем узле.
  int value;
  // Родительский узел для данного узла (у корня он равен nullptr).
  BinaryTree* parent;
  // Левый дочерний узел (nullptr, если нет).
  BinaryTree* left;
  // Правый дочерний узел (nullptr, если нет).
  BinaryTree* right;

 public:
  BinaryTree(int value, BinaryTree* parent = nullptr) :
    value(value),
    parent(parent),
    left(nullptr),
    right(nullptr)
  { }

  ~BinaryTree() {
    if (left)
      delete left;
    if (right)
      delete right;
  }

  /* Функция добавляет элемент в дерево, используя рекурсию. */
  void Insert(int elem) {
    /* Определяем, в какую ветвь нужно добавить элемент. */
    if (elem < value) {
      if (left == nullptr) {
        /* Текущий узел является листом. Добавляем к нему левый дочерний узел. */
        left = new BinaryTree(elem, this);
        return;
      }

      /* Добавляем элемент рекурсивно. */
      left->Insert(elem);
    }
    else {
      if (right == nullptr) {
        /* Текущий узел является листом. Добавляем к нему правый дочерний узел. */
        right = new BinaryTree(elem, this);
```

```
44        return;
45      }
46
47      /* Добавляем элемент рекурсивно. */
48      right->Insert(elem);
49    }
50  }
51
52  /* Функция проверяет, содержится ли элемент в дереве. */
53  bool Contains(int elem) const {
54    if (elem == value)
55      return true;
56    if (left && elem <= value)
57      return left->Contains(elem);
58    if (right && elem > value)
59      return right->Contains(elem);
60
61    return false;
62  }
63
64  // Методы для чтения приватных полей
65  int Value() const { return value; }
66  const BinaryTree* Parent() const { return parent; }
67  const BinaryTree* Left() const { return left; }
68  const BinaryTree* Right() const { return right; }
69
70 private:
71  /* Следующие функции пригодятся для написания итераторов. Функция возвращает
72     указатель на самый левый лист дерева. */
73  BinaryTree* GetLeftLeaf() {
74    BinaryTree* retval = this;
75
76    while (retval->left != nullptr || retval->right != nullptr)
77    {
78      /* По возможности идём в левую ветвь. */
79      if (retval->left)
80        retval = retval->left;
81      else
82        retval = retval->right;
83    }
84
85    /* В retval записан указатель на самый левый лист. */
86    return retval;
87  }
88
89  /* Функция возвращает указатель на самый правый лист дерева. */
90  BinaryTree* GetRightLeaf() {
91    BinaryTree* retval = this;
92
93    while (retval->left != nullptr || retval->right != nullptr)
94    {
```

```
95        /* По возможности идём в правую ветвь. */
96        if (retval->right)
97          retval = retval->right;
98        else
99          retval = retval->left;
100     }
101
102     /* В retval записан указатель на самый правый лист. */
103     return retval;
104   }
105
106   /*
107     Итераторы
108   */
109 };
```

2.2 Итераторы

Ниже приведён код итератора, который реализует обход дерева в глубину (depth-first traversal). То есть, дерево обходится в следующем порядке:

```
1 void Traverse(const BinaryTree* root) {
2   if (root == nullptr)
3     return;
4
5   Traverse(root->Left());
6   Traverse(root->Right());
7
8   std::cout << root->Value() << std::endl;
9  }
```

Иными словами, для каждого узла итератор сначала идёт в левое поддерево, затем в правое поддерево, а узел он посещает в самом конце.

```
1 class BinaryTree {
2   /*
3     Базовые методы класса
4   */
5  public:
6   class Iterator {
7    public:
8     Iterator(BinaryTree* node) :
9         node(node)
10    { }
11
12    /* Унарная операция * возвращает значение, записанное в узле дерева, на который
13       указывает итератор. */
14    int& operator*() const {
15      return node->value;
```

```
16      }
17
18      /* Переход к следующему узлу. Сначала идём в левое поддерево, затем в правое,
19         в самом конце посещаем узел. */
20      Iterator& operator++() {
21        /* Корень является последним узлом в списке обхода. После него идёт фиктивный
22           итератор, указывающий на "элемент после последнего". */
23        if (!node->parent) {
24          node = nullptr;
25          return *this;
26        }
27
28        /* Правый сосед текущего узла (может совпадать с ним самим). */
29        BinaryTree* rightSibling = node->parent->right;
30
31        /* Если правый узел отсутствует или текущий узел и есть правый, то после него
32           итератор должен перейти в родительский узел. */
33        if(!rightSibling || node == rightSibling) {
34          node = node->parent;
35          return *this;
36        }
37
38        /* В противном случае переходим в правое поддерево. В нём нужно взять самый
39           левый лист. */
40        node = rightSibling->GetLeftLeaf();
41        return *this;
42      }
43
44      bool operator==(const Iterator& other) const {
45        return node == other.node;
46      }
47
48      bool operator!=(const Iterator& other) const {
49        return node != other.node;
50      }
51     private:
52      // Узел дерева, на который указывает итератор.
53      BinaryTree* node;
54    };
55
56    // Итераторы на начало и невалидный итератор на конец.
57    Iterator Begin() { return Iterator(GetLeftLeaf()); }
58    Iterator End() { return NULL; }
59
60    /* Для того, чтобы у итератора был доступ к приватным членам класса BinaryTree,
61       класс Iterator должен быть дружественным по отношению к классу BinaryTree. */
62    friend class Iterator;
63  };
```

3 Сбалансированные деревья

Материал этого раздела написан по книге [1]. Примеры исходного кода также взяты с незначительными модификациями из [1]. Рассмотрим простейшее сбалансированное дерево — АВЛ-дерево (Г. М. Адельсон-Вельский, Е. М. Ландис). АВЛ дерево является сбалансированным в том смысле, что для любого узла такого дерева длина каждого его поддерева отличается не более чем на 1. Под длиной дерева подразумевается длина его максимальной ветви.

Назовём балансом узла разность длин его правого и левого поддеревьев.

```
struct TreeNode {
  // Значение, которое хранится в узле дерева.
  int value;
  // Указатель на левого потомка данного узла (nullptr, если отсутствует).
  TreeNode* left;
  // Указатель на левого потомка данного узла (nullptr, если отсутствует).
  TreeNode* right;
  // Баланс поддерева.
  int balance;

  TreeNode(int value, TreeNode* left = nullptr, TreeNode* right = nullptr) :
      value(value),
      left(left),
      right(right),
      balance(0)
  { }
};

/* Функция для рекурсивного освобождения памяти. */
void RecoursiveDelete(TreeNode* root) {
  if (!root)
    return;

  RecoursiveDelete(root->left);
  RecoursiveDelete(root->right);
}
```

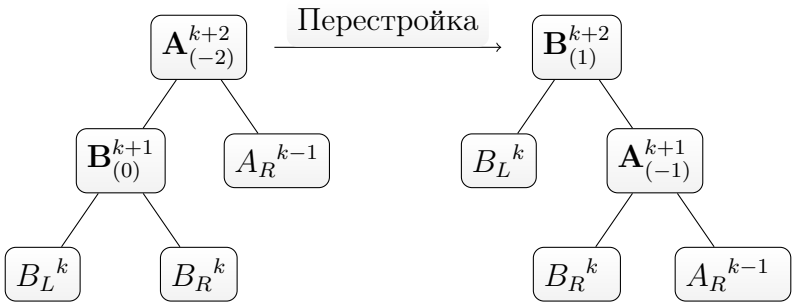
3.1 Балансировка дерева

При добавлении/удалении элемента из дерева могут появляться несбалансированные узлы (баланс которых равен 2 или -2) в той ветви, которая была затронута.

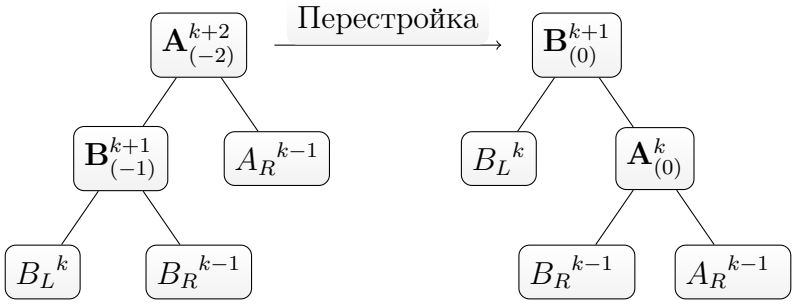
Введём следующие обозначения: высоту дерева будем указывать в верхнем индексе, а баланс в круглых скобках в нижнем индексе соответствующего узла. Например, запись $\mathbf{A}^{k+2}_{(-2)}$ означает, что поддерево, выходящее из узла **A** имеет высоту $k+2$ и узел **A** имеет баланс -2.

Пусть при добавлении/удалении узла появился несбалансированный узел **A**. В таком случае необходимо выполнить балансировку. Разберём несколько вариантов. Предположим, что все потомки узла **A** сбалансированы, то есть узел **A** является самым близким к листу несбалансированным узлом. Тогда все возможные ситуации описываются следующими шестью вариантами:

- Если баланс узла **A** равен -2, а баланс левого потомка узла **A** (узел **B**) равен 0.



- Если баланс узла **A** равен -2, а баланс левого потомка узла **A** (узел **B**) равен -1.



Оба случая описываются следующим кодом:

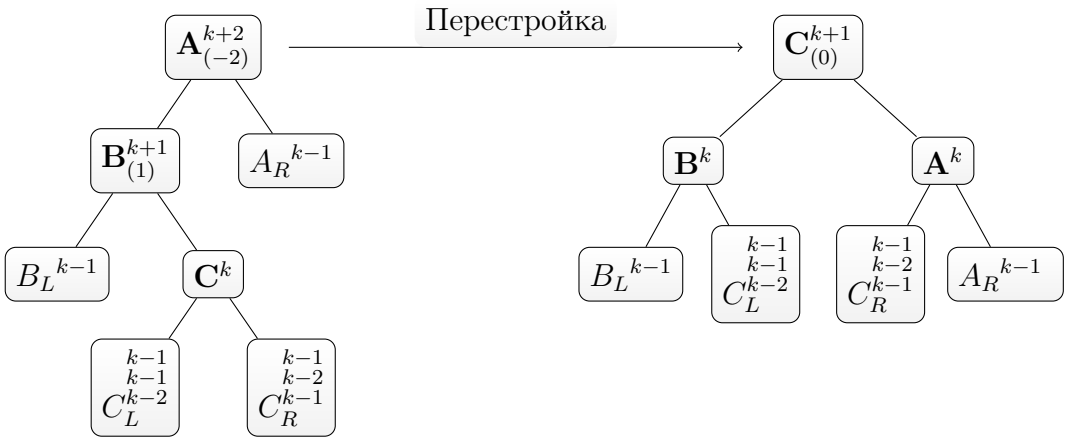
```
/* Узел node соответствует узлу A. */
TreeNode* RebuildL1(TreeNode* node) {
  /* Узел leftChild соответствует узлу B. */
  TreeNode* leftChild = node->left;

  /* Выполняем перестановку узлов. */
  node->left  = leftChild->right;
  leftChild->right = node;

  /* Балансы узлов A и B поменяются после перестановки. */
  if (leftChild->balance == 0) {
    node->balance =-1;
    leftChild->balance = 1;
  } else if (leftChild->balance == -1) {
    node->balance = 0;
    leftChild->balance = 0;
  } else {
    throw std::runtime_error("Something wrong has happened!");
  }

  /* Узел B становится корнем соответствующего поддерева. */
  return leftChild;
}
```

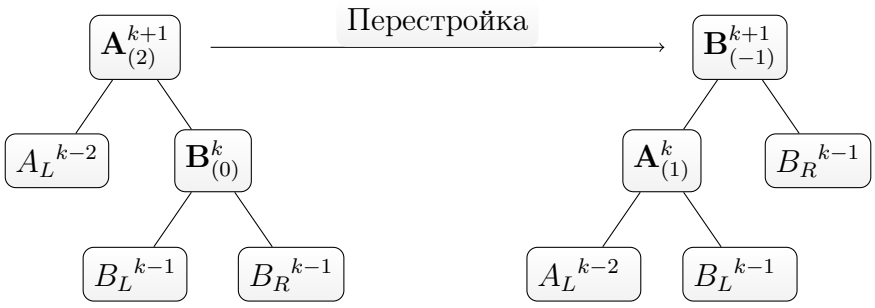
- Если баланс узла **A** равен -2, а баланс левого потомка узла **A** (узел **B**) равен 1.



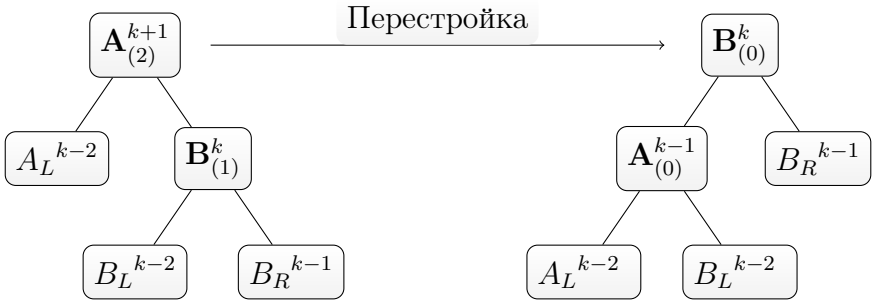
Данный случай описывается следующим кодом:

```
1  TreeNode* RebuildL2(TreeNode* A) {
2    TreeNode* B = A->left;
3    TreeNode* C = B->right;
4
5    /* Выполняем перестановку узлов. */
6    A->left  = C->right;
7    B->right = C->left;
8    C->left = B;
9    C->right = A;
10
11   /* Корректируем балансы. */
12   if (C->balance == 0) {
13     A->balance = 0;
14     B->balance = 0;
15   } else if (C->balance == -1) {
16     A->balance = 1;
17     B->balance = 0;
18   } else if (C->balance == 1) {
19     A->balance = 0;
20     B->balance =-1;
21   } else {
22     throw std::runtime_error("Something wrong has happened!");
23   }
24
25   C->balance = 0;
26
27   /* Узел C является новым корнем соответствующего поддерева. */
28   return C;
29 }
```

- Если баланс узла **A** равен 2, а баланс правого потомка узла **A** (узел **B**) равен 0.



- Если баланс узла **A** равен 2, а баланс правого потомка узла **A** (узел **B**) равен 1.



Два последних случая описываются следующим кодом:

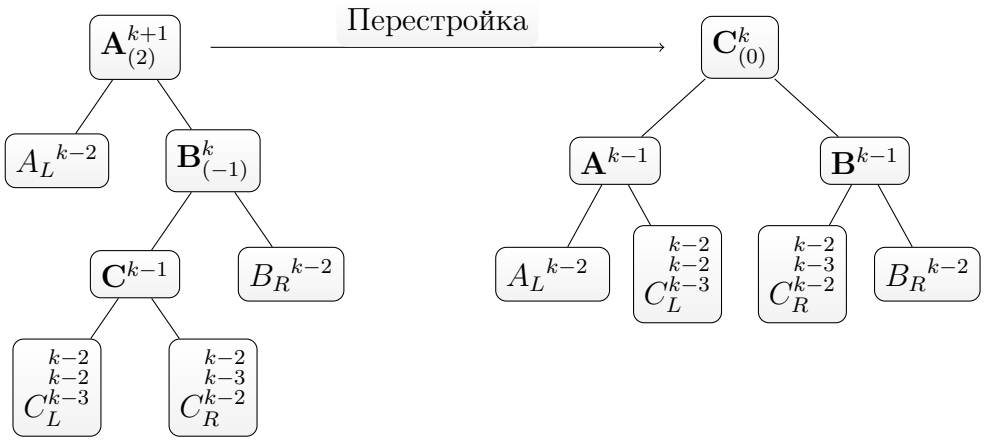
```
/* Узел node соответствует узлу A. */
TreeNode* RebuildR1(TreeNode* node) {
  /* Узел rightChild соответствует узлу B. */
  TreeNode* rightChild = node->right;

  /* Выполняем перестановку узлов. */
  node->right = rightChild->left;
  rightChild->left = node;

  /* Балансы узлов A и B поменяются после перестановки. */
  if (rightChild->balance == 0) {
    node->balance = 1;
    rightChild->balance = -1;
  } else if (rightChild->balance == 1) {
    node->balance = 0;
    rightChild->balance = 0;
  } else {
    throw std::runtime_error("Something wrong has happened!");
  }

  /* Узел B становится корнем соответствующего поддерева. */
  return rightChild;
}
```

- Если баланс узла **A** равен 2, а баланс правого потомка узла **A** (узел **B**) равен -1.



Данный случай описывается следующим кодом:

```
1  TreeNode* RebuildR2(TreeNode* A) {
2    TreeNode* B = A->right;
3    TreeNode* C = B->left;
4
5    /* Выполняем перестановку узлов. */
6    B->left = C->right;
7    A->right = C->left;
8    C->left = A;
9    C->right = B;
10
11   /* Корректируем балансы. */
12   if (C->balance == 0) {
13     A->balance = 0;
14     B->balance = 0;
15   } else if (C->balance == 1) {
16     A->balance = -1;
17     B->balance = 0;
18   } else if (C->balance == -1) {
19     A->balance = 0;
20     B->balance = 1;
21   } else {
22     throw std::runtime_error("Something wrong has happened!");
23   }
24
25   C->balance = 0;
26   /* Узел C является новым корнем соответствующего поддерева. */
27   return C;
28 }
```

Замечание 3.1. *Отметим, что указанные преобразования сохраняют упорядоченность узлов, то есть дерево остаётся деревом поиска.*

3.2 Добавление/удаление элемента

Напишем класс сбалансированного дерева, в котором будут доступны методы добавления и удаления элементов.

```
class BalancedTree {
 private:
  TreeNode* root;
 public:
  BalancedTree() :
    root(nullptr)
  { }

  ~BalancedTree() {
    RecursiveDelete(root);
  }

  /* Функция добавляет элемент в дерево. */
  void Add(int value) {
    AddBalance(value, root);
  }

  /* Функция удаляет элемент из дерева. */
  void Remove(int value) {
    RemoveBalance(value, root);
  }

  void Print() {
    Print(root);
  }

 private:

  static void Print(TreeNode* node) {
    if (node == nullptr)
      return;

    Print(node->left);
    Print(node->right);

    std::cout << node->value << std::endl;
  }

  /* Функция добавляет элемент в дерево. Узел, переданный в функцию, может измениться.
     Функция возвращает true, если длина дерева изменилась, то есть если может
     понадобиться балансировка. */
  static bool AddBalance(int value, TreeNode*& root) {
    /* Дерево является пустым. Добавляем новый элемент. */
    if (!root) {
      root = new TreeNode(value);
      /* Длина дерева увеличилась, возвращаем true. */
      return true;
    }

    /* Если дерево не пустое. */
```

```
51      if (value <= root->value) {
52        /* Добавляем элемент рекурсивно в левое поддерево. */
53        if (AddBalance(value, root->left)) {
54          /* Длина дерева поменялась. Баланс узла root уменьшился на единицу. В некоторых
55             случаях необходимо сделать балансировку. */
56          if (root->balance == 0) {
57            root->balance = -1;
58            /* Длина дерева увеличилась, возвращаем true. */
59            return true;
60          } else if (root->balance == 1) {
61            root->balance =  0;
62          } else if (root->balance == -1) {
63            /* Вот здесь необходима балансировка, поскольку баланс узла root стал
64               равен -2. Вызываем соответствующую функцию в зависимости от баланса левого
65               дочернего узла. */
66            if (root->left->balance == -1)
67              root = RebuildL1(root);
68            else if (root->left->balance == 1)
69              root = RebuildL2(root);
70          }
71        }
72      } else {  //  правая балансировка
73        /* Добавляем элемент рекурсивно в правое поддерево. */
74        if (AddBalance(value, root->right)) {
75          /* Длина дерева поменялась. Баланс узла root увеличился на единицу. В некоторых
76             случаях необходимо сделать балансировку. */
77          if (root->balance == 0) {
78            root->balance =  1;
79            /* Длина дерева увеличилась, возвращаем true. */
80            return true;
81          } else if (root->balance == -1) {
82            root->balance =  0;
83          } else if (root->balance == 1) {
84            /* Вот здесь необходима балансировка, поскольку баланс узла root стал
85               равен 2. Вызываем соответствующую функцию в зависимости от баланса правого
86               дочернего узла. */
87            if (root->right->balance == 1)
88              root = RebuildR1(root);
89            else if (root->right->balance == -1)
90              root = RebuildR2(root);
91          }
92        }
93      }
94
95      /* Длина дерева не увеличилась, балансировка не требуется, возвращаем false. */
96      return false;
97    }
98
99    /* Функция удаляет элемент из дерева. Узел, переданный в функцию, может измениться.
100      Функция возвращает true, если длина дерева изменилась, то есть если может
101      понадобиться балансировка. */
```

```
102 static bool RemoveBalance (int value, TreeNode*& root) {
103   // Дерево является пустым. Нечего удалять.
104   if (!root)
105     return false;
106 
107   if (root->value == value) {
108     /* Если нашёлся элемент, который нужно удалить. */
109     if (root->left == nullptr) {
110       /* Если нет левого дочернего узла. */
111       TreeNode* tmp = root;
112 
113       /* Вырезаем узел root (ныне tmp) из дерева. */
114       root = root->right;
115 
116       /* Освобождаем память вырезанного узла. */
117       delete tmp;
118 
119       /* Длина дерева изменилась, может понадобиться балансировка. */
120       return true;
121     }
122     if (root->right == nullptr) {
123       /* Если нет правого дочернего узла. */
124       TreeNode* tmp = root;
125 
126       /* Вырезаем узел root (ныне tmp) из дерева. */
127       root = root->left;
128 
129       /* Освобождаем память вырезанного узла. */
130       delete tmp;
131 
132       /* Длина дерева изменилась, может понадобиться балансировка. */
133       return true;
134     }
135 
136     /* В том случае, если у узла имеются оба дочерних, то мы не можем просто так
137        вырезать этот узел. Однако, мы можем переставить его местами с самым правым
138        листом левого поддерева, а затем удалить из левого поддерева соответствующее
139        значение. Это будет корректная операция, то есть упорядоченность узлов
140        не нарушится, и дерево останется деревом поиска. */
141     TreeNode* tmp = root->left;
142 
143     while (tmp->right)
144       tmp = tmp->right;
145 
146     /* Ставим значение правого листа левого поддерева в текущий корень. */
147     root->value = tmp->value;
148 
149     /* Удаляем соответствующее значение из левого поддерева. */
150     if (RemoveBalance(tmp->value, root->left)) {
151       /* Длина дерева поменялась. Баланс узла root увеличился на единицу. В некоторых
152          случаях необходимо сделать балансировку. */
```

```
153         if (root->balance == -1) {
154           root->balance = 0;
155           /* Длина дерева уменьшилась, возвращаем true. */
156           return true;
157         } else if (root->balance == 0)
158           root->balance = 1;
159         else if (root->balance == 1) {
160           if (root->right->balance == 1) {
161             /* Длина дерева уменьшилась, возвращаем true. */
162             return true;
163           } else if (root->right->balance == 0)
164             root = RebuildR1(root);
165           else if (root->right->balance == -1) {
166             root = RebuildR2(root);
167             /* Длина дерева уменьшилась, возвращаем true. */
168             return true;
169           }
170         }
171       }
172     } else {
173       /* В этой ветви условного перехода разбираем случай, если root->value != value. */
174       if (value < root->value) {
175         /* Удаляем элемент из левого поддерева. */
176         if (RemoveBalance (value, root->left)) {
177           /* Длина дерева поменялась. Баланс узла root увеличился на единицу.
178              В некоторых случаях необходимо сделать балансировку. */
179           if (root->balance == -1) {
180             root->balance = 0;
181             /* Длина дерева уменьшилась, возвращаем true. */
182             return true;
183           } else if (root->balance == 0) {
184             root->balance = 1;
185           } else if (root->balance == 1) {
186             if (root->right->balance == 1) {
187               /* Длина дерева уменьшилась, возвращаем true. */
188               return true;
189             } else if (root->right->balance == 0)
190               root = RebuildR1(root);
191             else if (root->right->balance == -1) {
192               root = RebuildR2(root);
193               /* Длина дерева уменьшилась, возвращаем true. */
194               return true;
195             }
196           }
197         }
198       } else {
199         /* Удаляем элемент из правого поддерева. */
200         if (RemoveBalance (value, root->right)) {
201           /* Длина дерева поменялась. Баланс узла root уменьшился на единицу.
202              В некоторых случаях необходимо сделать балансировку. */
203           if (root->balance == 1) {
```

```
204             root->balance = 0;
205             /* Длина дерева уменьшилась, возвращаем true. */
206             return true;
207           } else if (root->balance == 0)
208             root->balance = -1;
209           else if (root->balance -1) {
210             if (root->left->balance == -1) {
211             /* Длина дерева уменьшилась, возвращаем true. */
212               return true;
213             } else if (root->left->balance == 0)
214               root = RebuildL1(root);
215             else if (root->left->balance == 1) {
216               root = RebuildL2(root);
217               /* Длина дерева уменьшилась, возвращаем true. */
218               return true;
219             }
220           }
221         }
222       }
223     }
224
225      /* Длина дерева не изменилась, балансировка не требуется. */
226      return  false;
227   }
228 };
```

Список литературы

[1] *В. Д. Валединский, А. А. Корнев* Методы программирования в примерах и задачах на языках C и C++.

[2] *А. Шень* Программирование: теоремы и задачи. МЦНМО, 2017.

[3] *Роберт Седжвик* Фундаментальные алгоритмы на C++.