ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»
Институт металлургии, машиностроения и транспорта

Курсовой проект

по дисциплине «Программирование на языках высокого уровня»

«AVL-дерево»

Пояснительная записка

Выполнил
студент гр. 33335/2 _____ Марков М.Е.
(подпись)

Работу принял
_____ Ананьевский М. С.
(подпись)

Санкт-Петербург
2018 г.

Формулировка задачи, которую решает алгоритм.

АВЛ-дерево — сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на 1.При работе алгоритма реализуются добавление, удаление и поиск минимального ключа, а также балансировка дерева.

Особенности алгоритма.

Ключ любого узла дерева не меньше любого ключа в левом поддереве данного узла и не больше любого ключа в правом поддереве этого узла.

Для любого узла дерева высота его правого поддерева отличается от высоты левого поддерева не более чем на единицу.

В данной работе все значения ключей целочисленные.

Реализация алгоритма.

Алгоритм был реализован при помощи языка программирования C++. Узел дерева представлен структурой node, полями которой являются значение ключа в узле, высота дерева, указатель на структуру node для левой и правой ветви.

```
1.  struct node // структура для представления узлов дерева
2.  {
3.      int key;  // текущий ключ
4.      unsigned short height;//высота поддерева с корнем в текущем узле
5.      node* left;//указатель на левое дерево
6.      node* right;//указатель на правое дерево
7.  };
```

Создан класс tree, содержащий конструктор без параметров, который создаёт корень дерева root. Класс также содержит методы основных операций над деревьями и методы вспомогательных операций.

```
1.  class tree
2.  {
3.  public:
4.      node *root;
5.      node* turnright(node* p);
6.      node* turnleft(node* q);
7.      node* findmin(node* p);
8.      node* removemin(node* p);
9.      node* remove(node* p, int k);
10.     node* search(node* p, int k);
11.     node* insert(node* p, int k);
12.     node* balance(node* p);
13.     tree()
14.     {
15.         root=new node;
16.         root->left=0;
17.         root->right=0;
18.         root->key=0;
19.         root->height=1;
20.     };
21. private:
22.     unsigned short height(node* p);
23.     int8_t bfactor(node* p);
24.     void realheight(node* p); };
```

Отсутствие узлов слева или справа будем обнаруживать при помощи нулевого указателя в поле left и right соответственно. Для правильной работы программы необходимо реализовать функцию height:

```
1.  unsigned short tree::height(node* p)
2.  {
3.      return static_cast<unsigned short>(p ? p->height : 0);
4.  }
```

Если на вход подан отсутствующий узел, то она возвращает 0, иначе в поле height узла записывает высоту дерева с корнем в узле.

Функция bfactor возвращает разницу между высотой правой и левой ветви. По свойству AVL дерева он может принимать значения -1, 0, 1.

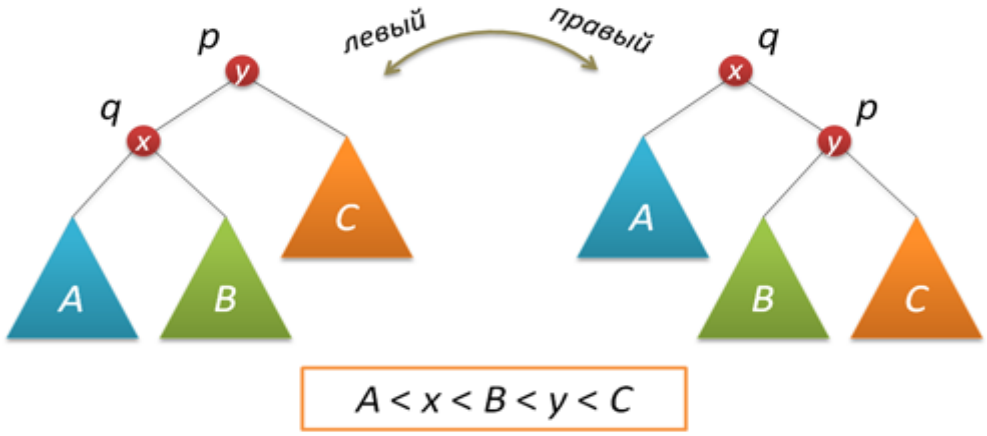
```
1.  int8_t tree::bfactor(node* p)
2.  {
3.      return (int8_t)(height(p->right) - height(p->left));
4.  }
```

При добавлении и удалении узлов может возникать ситуация, когда это условие нарушится. Для этого в программе предусмотрена функция балансировки дерева.

В качестве вспомогательной функции также выступает функция realheight, которая возвращает наибольшее значение высоты правой и левой ветви узла:

```
1.  void tree::realheight(node* p)
2.  {
3.      unsigned short hleft = height(p->left);
4.      unsigned short hright = height(p->right);
5.      p->height = (hleft > hright ? hleft : hright);
6.      p->height++;
7.  }
```

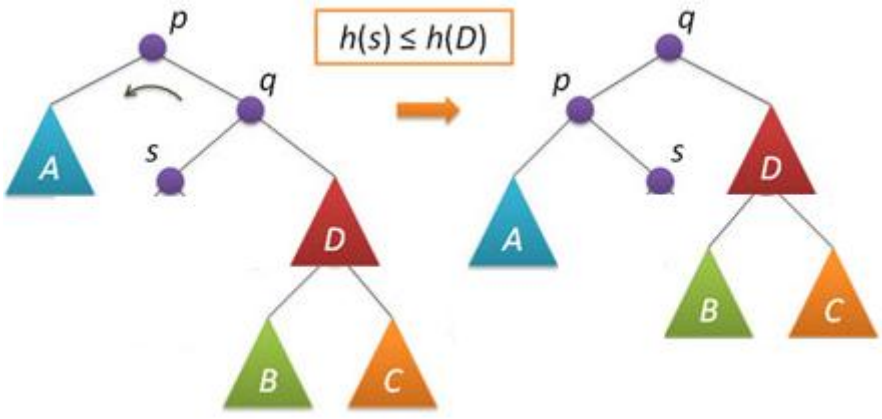
Балансировка узлов может быть осуществлена с помощью поворота вокруг узлов дерева. Правый поворот в программе реализован следующим образом: в функцию передаётся указатель на узел p, узлу q выбирается за левый от узла p. Затем левому узлу от p присваивается значение узла, правого от q. Затем объявляется, что узел p – это узел справа от q. Вычисляются новые высоты узлов p и q и возвращается узел q в качестве указателя на текущий узел. Левый поворот осуществлён аналогичным образом, что показано на рисунке ниже.



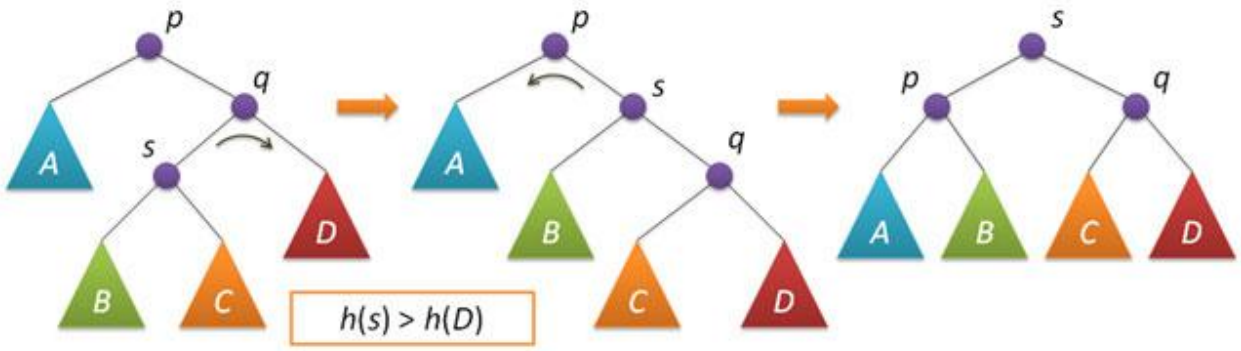
Код функций правого и левого поворота:

```
1.  node* tree::turnright(node* p) // правый поворот вокруг p
2.  {
3.      node* q = p->left;
4.      p->left = q->right;
5.      q->right = p;
6.      realheight(p);
7.      realheight(q);
8.      return q;
9.  }
10. node* tree::turnleft(node* q) // левый поворот вокруг q
11. {
12.     node* p = q->right;
13.     q->right = p->left;
14.     p->left = q;realheight(q);
15.     realheight(p);
16.     return p;
17. }
```

Непосредственно для исправления разбалансировки в узле p достаточно выполнить либо простой поворот влево вокруг p, либо большой поворот влево вокруг того же p. Простой поворот выполняется при условии, что высота левого поддерева узла q меньше или равна высоте его правого поддерева.



Большой поворот применяется при условии h(s)>h(D) и сводится в данном случае к двум простым — сначала правый поворот вокруг q и затем левый вокруг p.



Функция балансировки balance основана на проверке всех этих условий и возвращает тот же узел, который был подан на вход, но со сбалансированными ветвями.

```
1.  node* tree::balance(node* p) // балансировка узла p
2.  {
3.      realheight(p);
4.      if( bfactor(p)==2 )
5.      {
6.          if( bfactor(p->right) < 0 )
7.              p->right = turnright(p->right);
8.          return turnleft(p);
9.      }
10.     if( bfactor(p)==-2 )
11.     {
12.         if( bfactor(p->left) > 0 )
13.             p->left = turnleft(p->left);
14.         return turnright(p);
15.     }
16.     return p; // балансировка не нужна
17. }
18. }
```

Функция вставки ключа insert реализована с помощью прохода по дереву, сравнивая значение ключей в каждом из встретившихся узлов с тем, которое вставляется. В случае, если оно меньше, вызывается та же функция для левой ветви, иначе для правой. Если поданного узла не существует, то создаётся новая структура, обозначающая узел, она связана с последним существующим узлом (указатель на неё хранится в поле right или left последнего узла перед тем, как она была создана. При возврате из рекурсии производится балансировка.

```
1.  node* tree::insert(node* p, int k) // вставка ключа k в дерево с корнем p
2.  {
3.      if( !p ) {
4.          node* A=new node;
5.  A->height=1;
6.  A->key=k;
7.  A->left= nullptr;
8.  A->right= nullptr;
9.      return A;}
10.     if( k<p->key )
11.         p->left = insert(p->left,k);
12.     else
13.         p->right = insert(p->right,k);
14.     return balance(p);  }
```

Поиск ключей функцией search осуществляется проходом по дереву и сравнением искомого ключа и текущего, если ключ не найден, она сообщает об этом и возвращает нулевой указатель:

```
node* tree::search(node* p, int k){ // поиск ключа k  дерева p
    while((k != p->key)||((p->left != nullptr) && (p->right != nullptr))) {
        if (k < p->key)p = p->left;
        if (k > p->key) p = p->right;
        if (k == p->key) return p;
        if ((p->left == nullptr) && (p->right == nullptr)) {
            std::cout << "can't find the key\n"<<"return zero pointer"<<std::endl;
            return p;
        }
    }
}
```

Удаление ключей происходит функцией remove по следующей схеме: находится узел с заданным значением ключа k, затем если узел лист или не имеет правой ветви (указатель поля right нулевой), то в качестве текущего узла возвращается левый узел. Если правая ветвь у узла с заданным значением ключа имеется, то ищется минимальное значение в правой ветви (необходимо спускаться по дереву, держась только левых узлов до конца), узел с текущим значением q удаляется, на его место вставляется минимальный элемент правой ветви и дерево балансируется.

Код, осуществляющий все перечисленные действия, представлен ниже.

```
node* tree::findmin(node* p) // поиск узла с минимальным ключом в дереве p
{
    return p->left?findmin(p->left):p;
}

node* tree::removemin(node* p) // удаление узла с минимальным ключом из дерева p
{
    if( p->left==nullptr )
        return p->right;
    p->left = removemin(p->left);
    return balance(p);
}

node* tree::remove(node* p, int k) // удаление ключа k из дерева p
{
    if( !p ) return nullptr;
    if( k < p->key )
        p->left = remove(p->left,k);
    else if( k > p->key )
        p->right = remove(p->right,k);
    else // k == p->key
    {
        node* q = p->left;
        node* r = p->right;
        delete p;
        if( !r ) return q;
        node* min = findmin(r);
        min->right = removemin(r);
        min->left = q;
        return balance(min);
    }
    return balance(p);
}
```

Анализ алгоритма

1. Время работы алгоритма

Высота h АВЛ-дерева с n ключами лежит в диапазоне от $\log_2(n + 1)$ до $1.44 \log_2(n + 2) - 0.328$. Основные операции над двоичными деревьями поиска (поиск, вставка и удаление узлов) линейно зависят от его высоты, что гарантирует логарифмическую зависимость времени работы этих алгоритмов от числа ключей, хранимых в дереве.

Расход памяти O(n).

Проверка времени работы алгоритма:

Проверка функции вставки ключа(insert)

При 5000 операциях время выполнения: `Delta t2-t1: 5000300 nanoseconds`

При 10000 операциях время выполнения: `Delta t2-t1: 10000500 nanoseconds`

При 50000 операциях время выполнения: `Delta t2-t1: 61003500 nanoseconds`

При 100000 операциях время выполнения: `Delta t2-t1: 133007600 nanoseconds`

Проверка функции удаления ключа(remove):

При 5000 операциях время выполнения: `Delta t2-t1: 5000300 nanoseconds`

При 10000 операциях время выполнения: `Delta t2-t1: 10000600 nanoseconds`

При 50000 операциях время выполнения: `Delta t2-t1: 53003000 nanoseconds`

При 100000 операциях время выполнения: `Delta t2-t1: 94005400 nanoseconds`

Применение алгоритма

АВЛ-деревья могут быть применены для упорядоченного хранения элементов, вставки, поиска и удаления за время от $\log_2(n + 1)$ до $1.44 \log_2(n + 2) - 0.328$, что требуется, например, для баз данных.

Список литературы

1. Вирт Н. Алгоритмы и структуры данных. — М.: Мир, 1989. — С. 272—286.
2. Адельсон-Вельский Г. М., Ландис Е. М. Один алгоритм организации информации // Доклады АН СССР. — 1962. — Т. 146, № 2. — С. 263—266.
3. Д. Кнут, Искусство программирования