МИНОБРНАУКИ РОССИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
высшего образования
«Московский технологический университет»

МИРЭА

Институт информационных технологий (ИТ)
Кафедра инструментального и прикладного программного обеспечения (ИППО)

Отчет по практической работе № 2
по дисциплине
«Структуры и алгоритмы обработки данных»

Выполнил студент группы ИСБОп-01-14 Карих Д.С.

Принял Шмелёва Д.В.

Москва 2017

Двоичное дерево с принудительной балансировкой

В ходе данной практической работы было реализовано двоичное дерево поиска с принудительной балансировкой и хранением указателя на родителя в каждом элементе. Хранение информации о потомках реализовано в виде массива для обеспечения простоты доступа к левому и правому потомку программными методами.

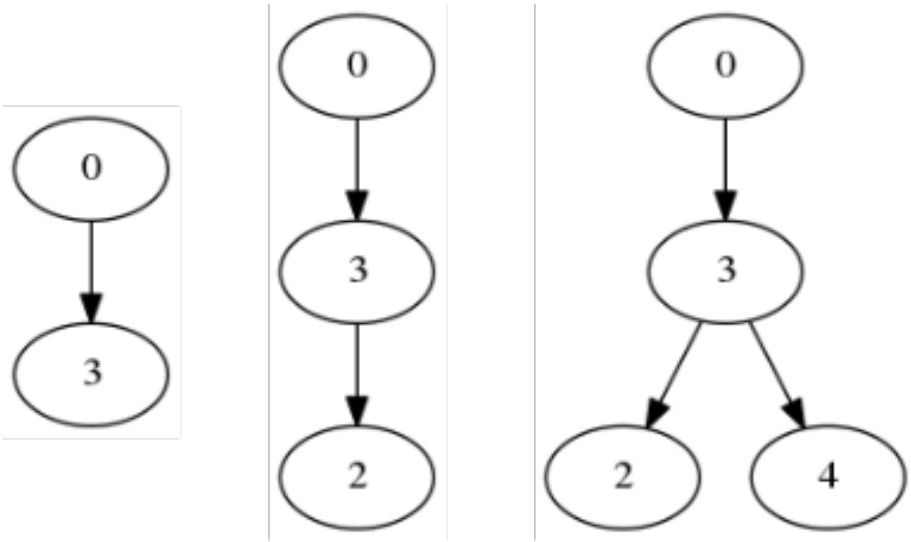
```
class Tree {
  private:
    Tree* parent = NULL;
    Tree* children[2] = {NULL,NULL};
    int key;
};
```

Функция поиска (find)

Для упрощения доступа к элементам по их ключу была реализована функция поиска. Если ключ, переданный в аргументе, находится в дереве, функция возвращает указатель на соответствующий элемент. В противном случае возвращается указатель на тот элемент, который подходит для вставки элемента с этим ключом.

Добавление элемента (add)

Данная функция находит место для вставки элемента при помощи функции find(), после чего добавляет новый элемент либо в виде левого потомка (если новый ключ меньше текущего), либо в виде правого потомка.



Удаление элемента (remove)

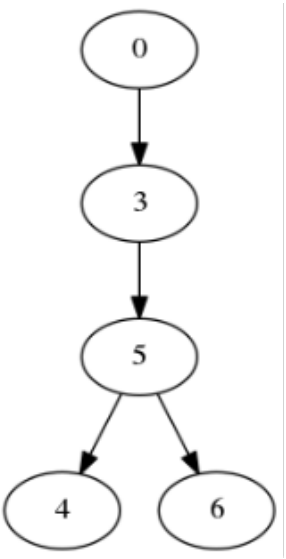
При удалении элемента необходимо предусмотреть три основных ситуации:

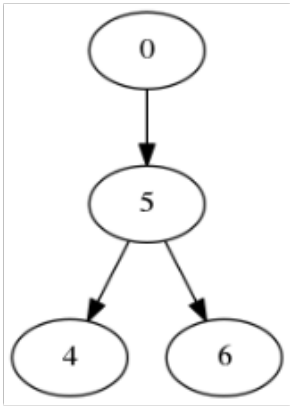
1. У удаляемого элемента нет потомков

В данном случае просто обнуляем указатель на этот элемент у родителя и очищаем память.

2. У удаляемого элемента есть один потомок

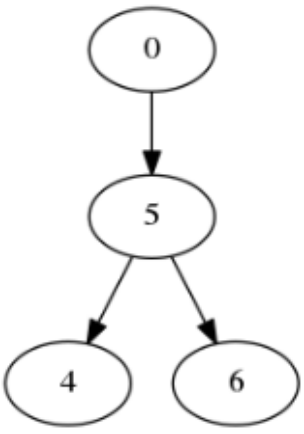
В данном случае заменяем удаляемый элемент его потомком. При этом дочерние элементы потомка никак не затрагиваются.

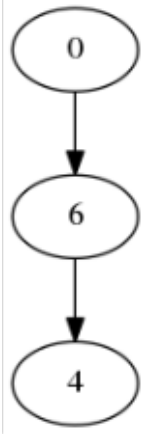




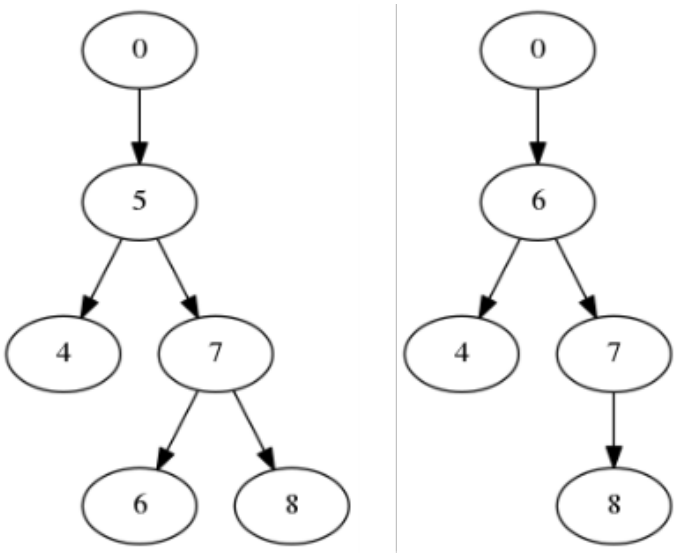
3. У удаляемого элемента есть оба потомка

    1. Если у правого потомка нет левой ветви, то применяем метод, аналогичный методу (2). Единственное отличие – прикрепляем левую ветвь удаляемого элемента к правому потомку.



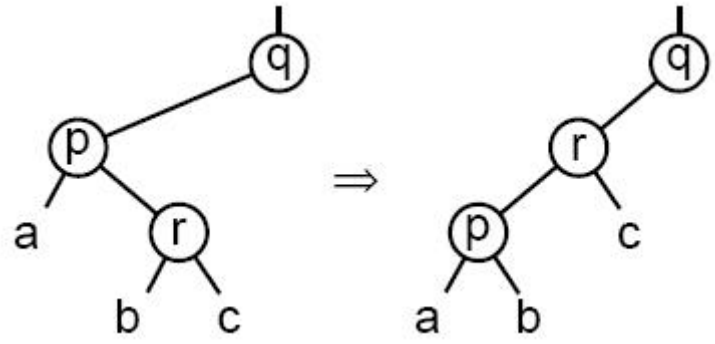


2. Если у правого элемента есть левая ветвь, то предыдущий способ нам не подходит, так как эта ветвь будет заменена и утеряна. В данном случае мы можем заменить удаляемый элемент минимальным из правого поддерева. При этом нужно учесть ситуацию, когда у этого минимального элемента есть правый потомок, просто удалив его при помощи метода (2).



Левый и правый поворот (rotate)

Для проведения балансировки нам понадобятся функции правого и левого поворота. Наиболее наглядно этот алгоритм показан на примере левого поворота на следующей иллюстрации.



Правый поворот является зеркальным отражением левого, поэтому аналогичен в реализации. Проще всего реализовать универсальную функцию поворота, если хранить указатели на потомков в массиве.

Преобразование в лозу (vine)

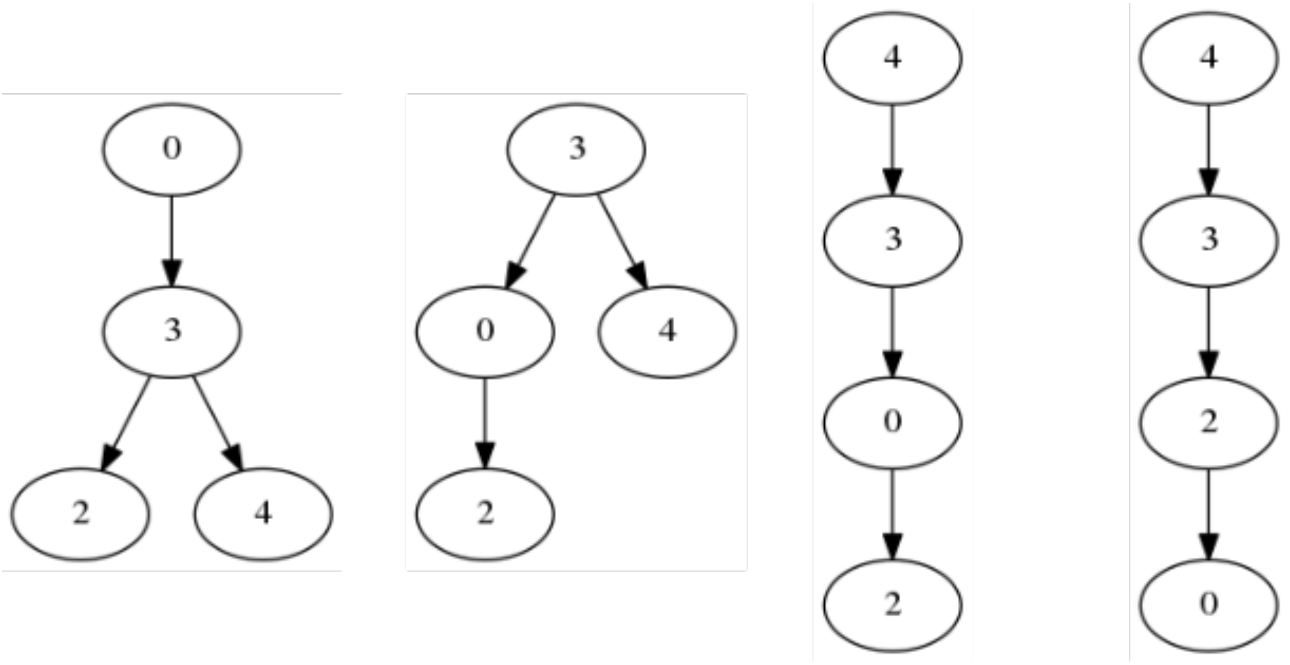
Данная функция приводит дерево к отсортированному списку, выполняя правый поворот для всех элементов, у которых есть правый потомок. Таким образом дерево вытягивается в список (лозу). При этом основное свойство дерева – упорядоченность элементов – не нарушается.

```
void vine() {
    Tree* tmp = this;

    while (tmp != NULL) {
        while (tmp->children[1] != NULL) {
            rotate(tmp->key, false);
        }
        tmp = tmp->children[0];
    }
}
```

На иллюстрации ниже можно увидеть пример выполнения функции vine() на дереве из 4 элементов.

```
rotate right 0
rotate right 3
rotate right 0
```



Балансировка дерева (balance)

В нашем случае балансировка является обратной операцией относительно функции vine(). Основное требование для успешной балансировки – количество элементов $=2^n-1$.

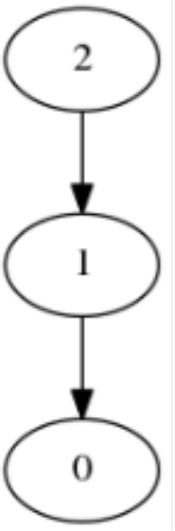
Необходимо обойти лозу $\log_2(\text{количество элементов}+1)$ раз, совершая левый поворот каждого элемента.

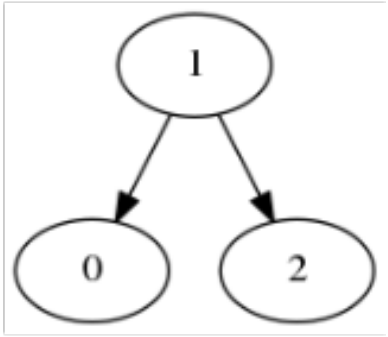
```
void balance() {
  float size_log = log2(size() + 1);

  Tree* tmp = NULL;
  for (int i = 1; i < size_log; i++) {
    tmp = this;
    for (int j = 0; j < pow(2, size_log-i) - 1; j++) {
      rotate(tmp->key, true); // правый поворот
      tmp = tmp->children[0];
    }
  }
}
```

Например, для дерева из $2^2-1=3$ элементов для полной балансировки достаточно совершить всего один правый поворот.

```
rotate right 2
```





Заключение

Двоичное дерево поиска позволяет ускорить нахождение элемента в памяти по сравнению с обходом простого списка. Однако его реализация гораздо сложнее, а также требуется периодическая балансировка (в нашем случае).

Список источников

1. Википедия: Двоичное дерево поиска (дата обращения: 11.04.17) // https://ru.wikipedia.org/wiki/Двоичное_дерево_поиска
2. Балансировка дерева (дата обращения: 11.04.17) // http://rain.ifmo.ru/cat/view.php/vis/trees/balanced-bst-2008/algorithm