Древесные структуры данных

Баталов Семен

2 мая 2020 г.

Оглавление

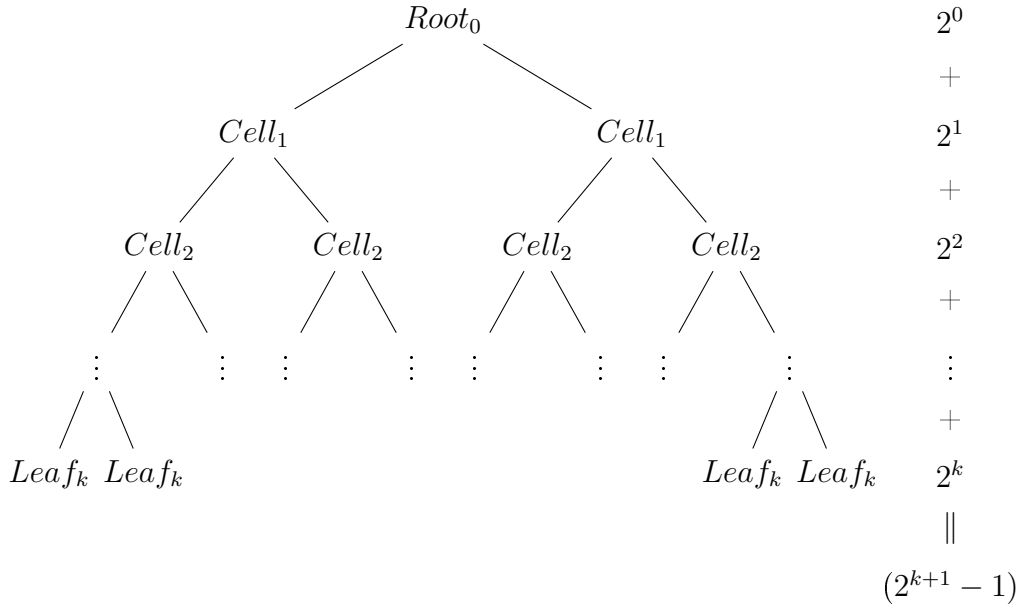
0.1. Предисловие

В данном разделе будут рассматриваться древесные структуры данных на примере бинарных деревьев. Будут освещены некоторые алгоритмы работы с деревьями, свойства и разновидности этих структур данных.

0.2. Основная терминология

Ниже представлено бинарное дерево (дерево, в котором каждый узел имеет не более двух потомков). Справа от него расположено количество элементов на каждом уровне и итоговое количество элементов дерева.

Стоит отметить, что среднее время поиска одного элемента $O(n)$, где $n = 2^{k+1}-1$. При этом среднее время вставки логарифмическое $O(\log n)$.



0.2.1. Определения

- **Потомки** – это те элементы, в которые можем попасть из текущего.
- **Предки** – это те элементы, из которых можем попасть в текущий.
- **Корень** – это элемент дерева без предков.

- **Лист** – это элемент дерева без потомков.
- **Прямой потомок** – это тот потомок, в который попадаем за один шаг.
- **Прямой предок** – это тот предок, из которого попадаем за один шаг.
- **Глубина элемента** – это расстояние от корня до текущего элемента.
- **Высота дерева** – это максимальная глубина.
- **Уровень** – это множество элементов с одинаковой глубиной.
- **Идеально сбалансированное дерево** – это дерево, в котором у всех элементов кроме элементов последних двух уровней есть ровно два потомка.
- **Сбалансированное дерево** – это дерево, высота которого логарифмична количеству элементов.

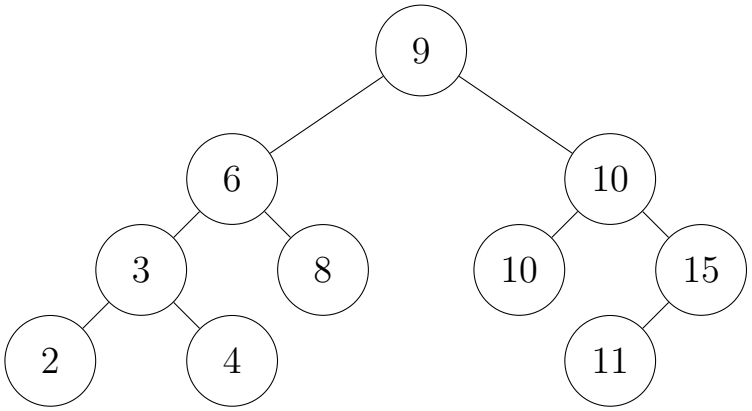
0.3. Дерево поиска

Рассмотрим бинарное дерево, где каждому элементу приписали ключ (некоторое целое число), уникальный для каждого элемента.

0.3.1. Определение

- **Дерево поиска** – это дерево, в котором для каждого элемента выполнено свойство: Значение ключа всех левых потомков меньше значения ключа текущего, а значение ключей всех правых потомков больше текущего.

Ниже расположен пример дерева поиска. За счет упорядоченной структуры это дерево позволяет эффективно решать задачу поиска. Далее мы рассмотрим всевозможные способы обхода дерева поиска.



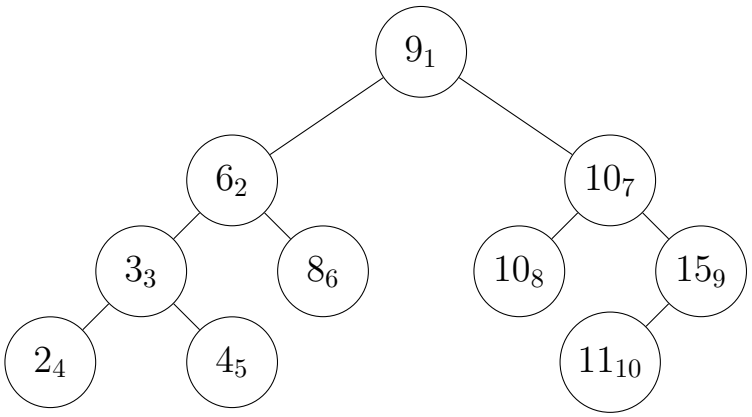
0.3.2. Обход в глубину (prefix)

1) Обрабатываем элемент $\{O(1)\}$

2) Рекурсивно переходим к левому потомку $\{O(1)\}$

3) Рекурсивно переходим к правому потомку $\{O(1)\}$

Далее показан пример такого обхода в глубину. Нижний индекс указывает, каким по счету будет обрабатываться элемент. Общая оценка сложности обхода в глубину $O(n)$.



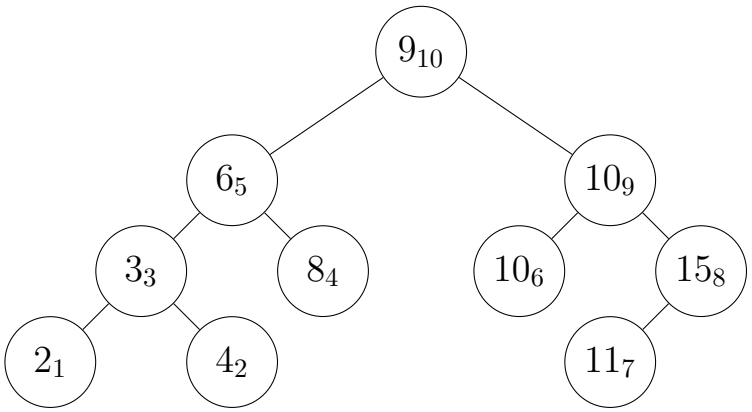
0.3.3. Обход в глубину (postfix)

1) Рекурсивно переходим к левому потомку $\{O(1)\}$

2) Рекурсивно переходим к правому потомку $\{O(1)\}$

3) Обрабатываем элемент $\{O(1)\}$

Далее показан пример такого обхода в глубину. Нижний индекс указывает, каким по счету будет обрабатываться элемент. Общая оценка сложности обхода в глубину $O(n)$.



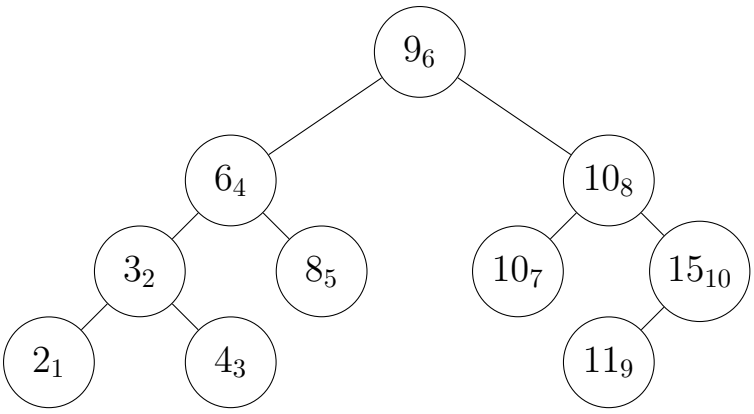
0.3.4. Обход в глубину (infix)

1) Рекурсивно переходим к левому потомку $\{O(1)\}$

2) Обрабатываем элемент $\{O(1)\}$

3) Рекурсивно переходим к правому потомку $\{O(1)\}$

Далее показан пример такого обхода в глубину. Нижний индекс указывает, каким по счету будет обрабатываться элемент.



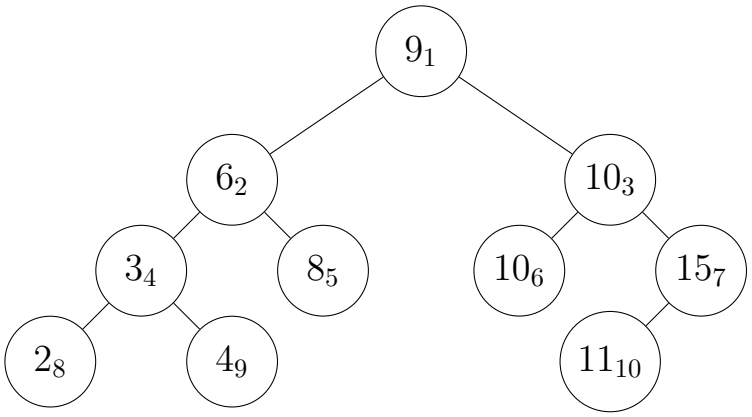
Данный способ обхода позволяет отсортировать элементы дерева. Причем сложность такой сортировки $O(n \cdot \log n)$, так как в дереве всего n элементов, вставка каждого и которых занимает $O(\log n)$, а общая оценка сложности обхода $O(n)$.

Объем памяти, требуемый для обхода в глубину: $O(h)$, где h – высота дерева.

0.3.5. Обход в ширину (bfs)

Данный обход производится последовательно по каждому уровню. Достаточно составить очередь, в которую будем записывать указатели на потомки текущего элемента (слева направо). Будем обрабатывать элементы из очереди, пока они не кончатся.

Далее показан пример такого обхода в ширину. Нижний индекс указывает, каким по счету будет обрабатываться элемент.



Объем памяти, требуемый для обхода в ширину: $O(w)$, где w – максимальное количество элементов на уровне.

0.4. Самобалансирующиеся деревья

Здесь мы рассмотрим условия сбалансированности бинарных деревьев поиска и методы их балансировки.

0.4.1. Определение

- **АВЛ дерево** – это бинарное дерево поиска, в котором для каждого элемента выполнено свойство: Разница высот левого и правого поддеревьев по модулю не превосходит единицы.

Рассмотрим АВЛ дерево высоты k. Оценим количество элементов в нем сверху и снизу.

$$F(k) \leq n \leq 2^{k+1} - 1$$

В этом неравенстве $F(k) = 1 + F(k-1) + F(k-2)$ – нижняя оценка. При этом $F(k-1)$ и $F(k-2)$ – количество элементов в левом и правом поддеревьях. Очевидно $F(k) > Fib(k)$ – k-ое число Фибоначчи.

Из формулы Бине вытекает следующее:

$$n > \left(\frac{1+\sqrt{5}}{2}\right)^k$$

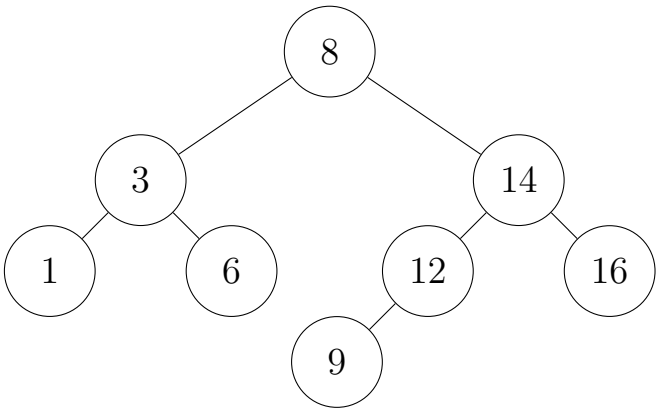
В итоге получаем оценку сверху и снизу для количества элементов АВЛ дерева:

$$\varphi^k \leq n \leq \psi^k$$

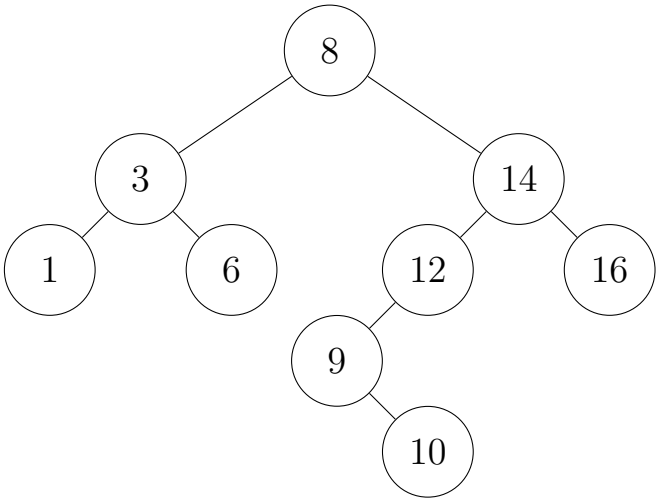
То есть $n = \theta^k$ – это показательная функция. Следовательно, высота дерева $k = \log n$. По определению такое дерево является сбалансированным. То есть АВЛ дерево по определению является сбалансированным.

0.5. Вставка и балансировка

Здесь будут рассматриваться способы балансировки АВЛ дерева при вставке в него элементов. Возьмем некоторое АВЛ дерево:



Попробуем осуществить вставку элемента 10 в это дерево так, как показано на следующем рисунке. Дерево перестает быть сбалансированным.



0.5.1. Определение

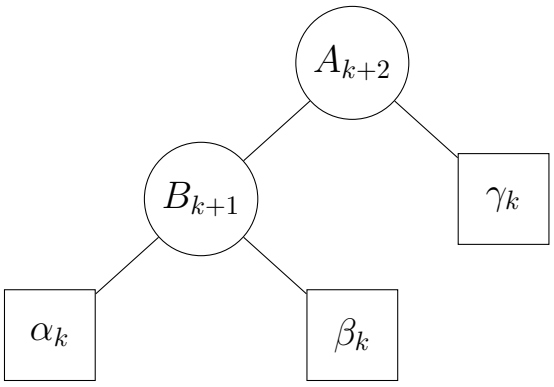
- **Поворот дерева** – это операция, которая позволяет изменить структуру дерева не меняя порядка элементов.

Для решения проблемы балансировки дерева используются следующие повороты:

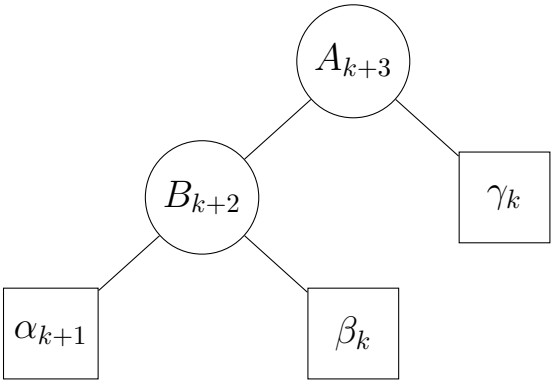
1) Малый левый поворот (SLR)
2) Малый правый поворот (SRR)
3) Большой левый поворот (BLR)
4) Большой правый поворот (BRR)

0.5.2. Малый правый и левый повороты

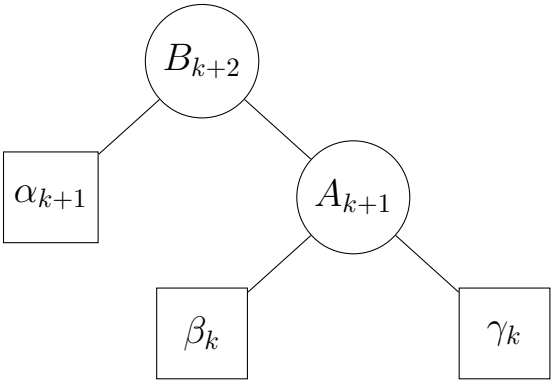
Рассмотрим следующее АВЛ дерево:



Высоты его поддеревьев таковы: $H(\alpha) = H(\beta) = H(\gamma) = k, H(A) = k + 2, H(B) = k + 1$. Условие баланса выполнено. Увеличим высоту поддерева α, добавив элемент, и получим: $H(\alpha) = k + 1, H(\beta) = k, H(\gamma) = k, H(A) = k + 3, H(B) = k + 2$. Дерево разбалансировано.



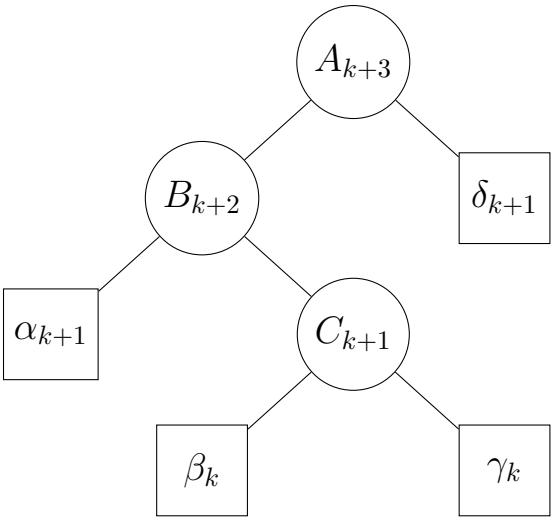
Преобразуем это дерево малым правым поворотом:



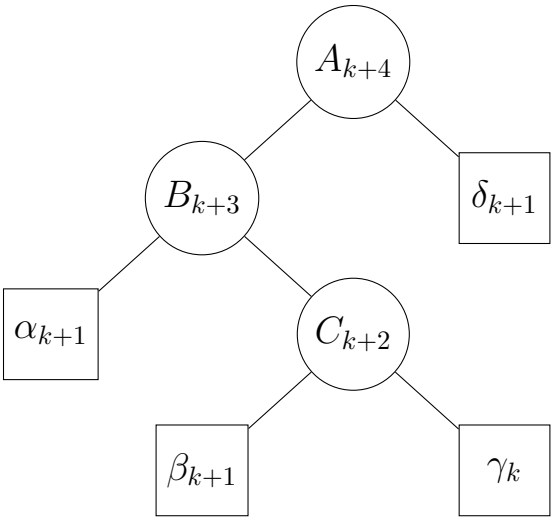
Переписав три указателя, мы сбалансировали это дерево за $O(1)$. Причем сохранилось свойство поиска. Малый левый поворот выполняется аналогично, но только в инвертированной ситуации.

0.5.3. Большой правый и левый повороты

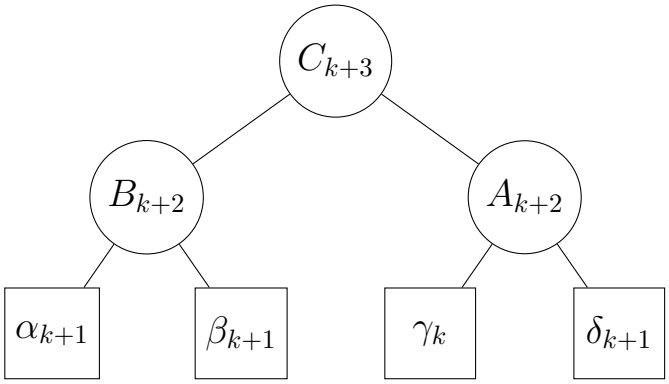
Рассмотрим следующее АВЛ дерево:



Высоты его поддеревьев таковы: $H(\beta) = H(\gamma) = k, H(\alpha) = H(\delta) = k+1, H(A) = k+3, H(B) = k+2, H(C) = k+1$. Условие баланса выполнено. Увеличим высоту поддерева β, добавив элемент, и получим: $H(\alpha) = H(\delta) = H(\beta) = k+1, H(\gamma) = k, H(A) = k+4, H(B) = k+3, H(C) = k+2$. Дерево разбалансировано.



Преобразуем это дерево большим правым поворотом:

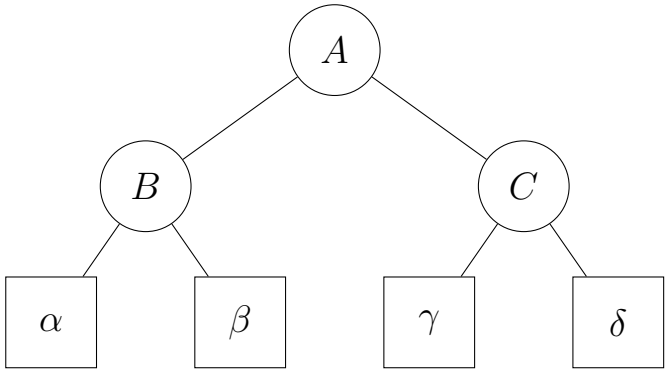


Переписав пять указателей, мы сбалансировали это дерево за $O(1)$. Причем сохранилось свойство поиска. Большой левый поворот выполняется аналогично, но только в инвертированной ситуации.

Так же стоит отметить, что неважно, в какое поддерево вершины C будет добавлен элемент. В любом случае большой поворот сбалансирует дерево.

0.5.4. Добавление элемента

Мы хотим вставлять элементы в АВЛ дерево за $O(\log n)$, поэтому нам нужно постоянно поддерживать его сбалансированным. Введем для каждой вершины величину баланса $balance(X) \in \{-1, 0, 1\}$, равную разности высот правого и левого поддеревьев этой вершины. Рассмотрим следующее АВЛ дерево:



Будем добавлять элементы в левое и правое поддерево и смотреть, как меняются величины баланса для вершин A, B, C. Составим таблицу возможных изменений:

Rotate	*balance*(B)	*balance*(A)		
		-1	0	1
none	$1 \longrightarrow -1$	-1	0	1
none	$-1 \longrightarrow 0$	-1	0	1
SRR	$0 \longrightarrow -1$	**0***	-1	0
none	$0 \longrightarrow 0$	-1	0	1
BRR	$0 \longrightarrow 1$	**0***	-1	0
none	$1 \longrightarrow 1$	-1	0	1
none	$1 \longrightarrow 0$	-1	0	1

Rotate	*balance*(C)	*balance*(A)		
		-1	0	1
none	$1 \longrightarrow -1$	-1	0	1
none	$-1 \longrightarrow 0$	-1	0	1
BLR	$0 \longrightarrow -1$	0	1	**0***
none	$0 \longrightarrow 0$	-1	0	1
SLR	$0 \longrightarrow 1$	0	1	**0***
none	$1 \longrightarrow 1$	-1	0	1
none	$1 \longrightarrow 0$	-1	0	1

Теперь, опираясь на данные таблицы, можно составить алгоритм вставки нового элемента в АВЛ дерево. Далее представлена процедура вставки, написанная на C++, но в виде алгоритмического языка.

0.5.5. Процедура вставки

```
void insert(Note* N, int x)
{
  int bal = 0;      // next vertex balance
  if (N != NULL)
    {
      if (N.value <= x)
      {
        bal = N.right*.bal;
        insert(N.right*, x);
        if (bal == 0 && N.right*.bal == 1 && N.bal == 1)
        {
          use SLR;
        }
        else
        if (bal == 0 && N.right*.bal == -1 && N.bal == 1)
        {
          use BLR;
        }
          update N.bal;
      }
      else
      {
        bal = N.left*.bal;
        insert(N.left*, x);
        if (bal == 0 && N.left*.bal == 1 && N.bal == -1)
```

```
            {
              use BRR;
            }
            else
            if (bal == 0 && N.left*.bal == -1 && N.bal == -1)
            {
              use SRR;
            }
            update N.bal;
          }
        }
        else
        {
          N = new Note;
          N.value = x;
        }
}
```