Лекция 6.
Динамические структуры данных

СОДЕРЖАНИЕ

1 СПИСКИ

1.1 Стек (LIFO)

Стеком называют структуру данных организованную по принципу LIFO (англ. last in—first out, последним вошёл — первым вышел). Графически данную структуру можно представить следующим образом:

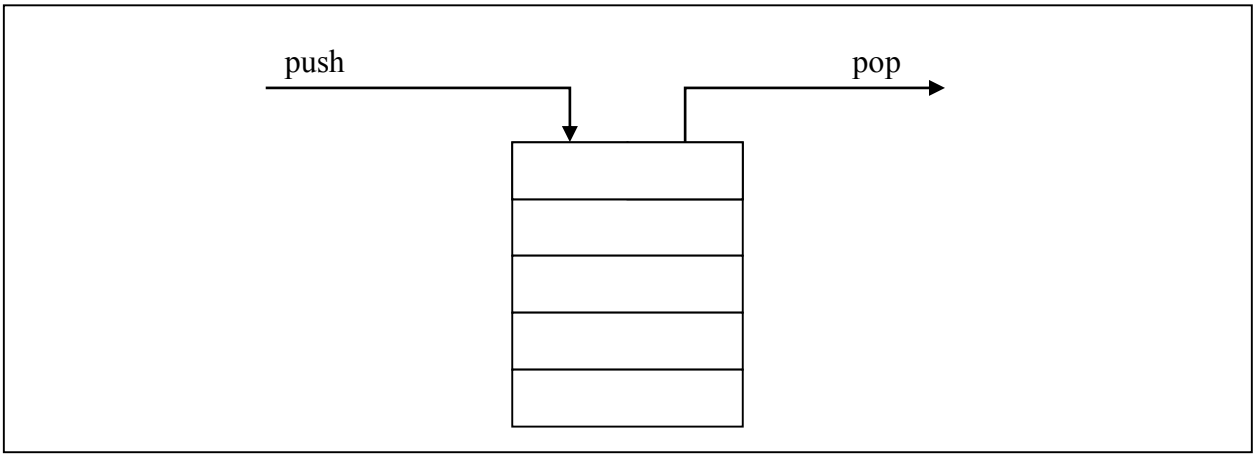

Рисунок 1 — Иллюстрация стека

Простым примером реализацией стека может служить статический массив, дополненный указателем на последний записанный элемент.

```
1       class Stack
2       {
3       public:
4            Stack() : last(0) { }
5       public:
6            bool is_full() {
7                 return (last == sizeof(array)/sizeof(array[0]));
8            }
9            bool is_empty() {
10                return !last;
11           }
12           int pop() {
13                if (!is_empty())
14                     return array[--last];
15                return 0;
16           }
17           void push(int element) {
18                if (!is_full())
19                array[last++] = element;
20           }
```

```
21        private:
22              int array[10];
23              size_t last;
24        };
25        …
26        Stack stack;
27        stack.push(10);
28        stack.push(11);
29        stack.push(12);
30        cout << "STACK:";
31        while (!stack.is_empty())
32              cout << " " << stack.pop();
33        cout << endl;   // STACK: 12 11 10
```

Листинг 1 — Пример реализации стека

1.2 Очередь (FIFO)

Очередью называют структуру данных, организованную по принципу FIFO (first in—first out, первым вошёл — первым вышел). Графически данную структуру можно представить следующим образом:

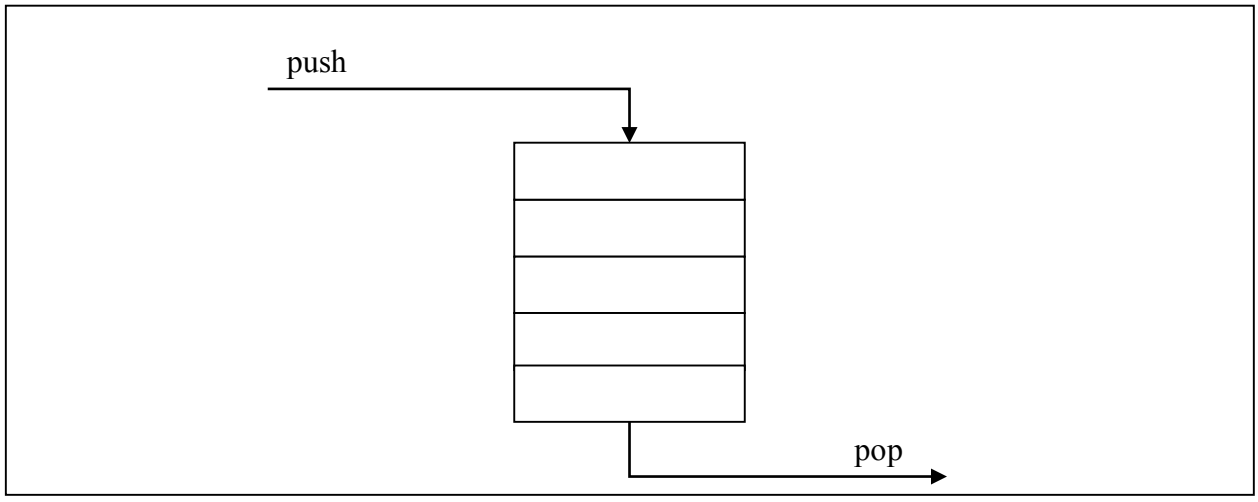

Рисунок 2 — Иллюстрация очереди

Простым примером реализацией стека может служить статический массив, дополненный указателем на первый и на последний записанный элемент. Эти указатели инкрементируются циклически: при достижении адреса последнего элемента массива следующей ячейкой становится первая.

```
1         class Queue
2         {
3         public:
4               Queue() : first(0), last(0) { }
```

```
5       public:
6            bool is_full() {
7                 return (last - first) ==
8                      sizeof(array)/sizeof(array[0]);
9            }
10           bool is_empty() {
11                return last == first;
12           }
13           int pop() {
14                if (is_empty())
15                     return 0;
16                if (first == sizeof(array)/sizeof(array[0]))
17                     first = 0;
18                return array[first++];
19           }
20           void push(int element) {
21                if (!is_full()) {
22                     if (last == sizeof(array)/sizeof(array[0]))
23                          last = 0;
24                     array[last++] = element;
25                }
26           }
27      private:
28           int array[10];
29           size_t first, last;
30      };
31      …
32      Queue queue;
33      queue.push(10);
34      queue.push(11);
35      queue.push(12);
36      cout << "QUEUE:";
37      while (!queue.is_empty())
38           cout << " " << queue.pop();
39      cout << endl;   // QUEUE: 10 11 12
```

Листинг 2 — Пример реализации очереди

1.3 Односвязный и двусвязный списки

Связанный список — это динамическая структура данных, которая состоит из узлов, каждый из которых содержит непосредственно данные и одну или две связи, указывающие на следующий и/или на предыдущий элемент. Применение того или иного списка зависит от требований решаемой задачи. Графическая интерпретация таких структур приведена ниже:

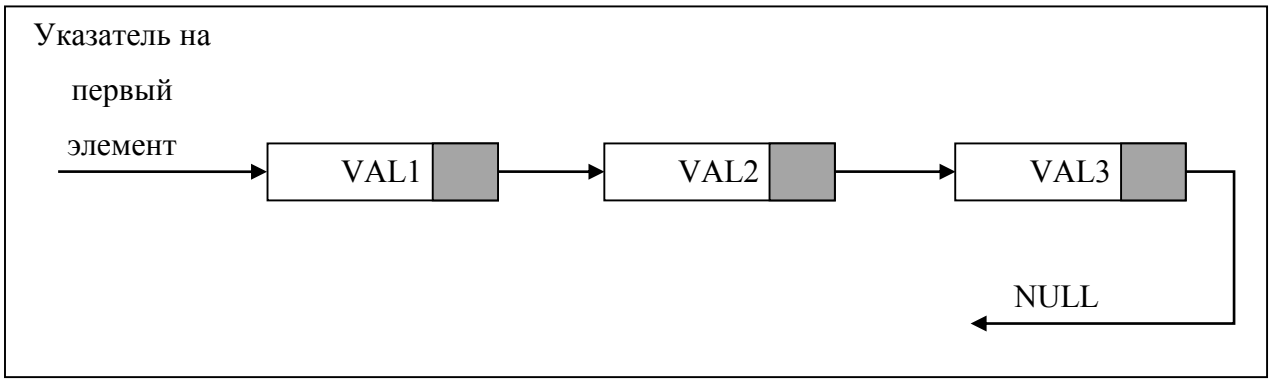

Рисунок 3 — Односвязный список

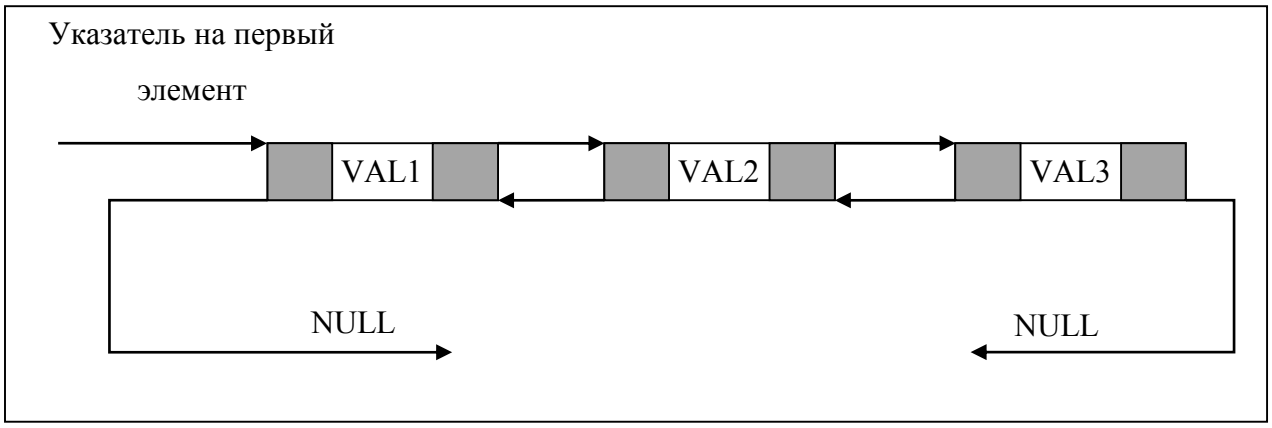

Рисунок 4 — Двусвязный список

Для примера покажем простую реализацию двусвязного списка. Класс позволяет выполнить такие функции как добавление нового элемента в конец (`push_back`) или в начало (`push_front`) списка, а также изъятие последнего (`pop_back`) или начального (`pop_front`) элемента из списка. Данную реализацию можно легко дополнить расширенным функционалом поиска, добавления или удаления элемента.

```
1       struct list_element
2       {
3            list_element *prev;
4            list_element *next;
5            int value;
6       };
7       class list
8       {
9       public:
10           list() : first(NULL), last(NULL) {}
11           ~list() {
12                while(!is_empty())
13                     pop_back();
14           }
15      public:
16           bool is_empty() {
```

```
17                return !(last);
18           }
19           void push_back(int element) {
20                list_element *new_element = new list_element;
21                new_element->next = nullptr;
22                new_element->prev = last;
23                new_element->value = element;
24                if (is_empty())
25                     first = new_element;
26                else
27                     last->next = new_element;
28                last = new_element;
29           }
30           int pop_back() {
31                if (is_empty())
32                     return 0;
33                int value = last->value;
34                if (last->prev) {
35                     last = last->prev;
36                     delete last->next;
37                     last->next = nullptr;
38                }
39                else {
40                     delete last;
41                     last = nullptr;
42                     first = nullptr;
43                }
44                return value;
45           }
46           void push_front(int element) {
47                list_element *new_element = new list_element;
48                new_element->next = first;
49                new_element->prev = nullptr;
50                new_element->value = element;
51                if (is_empty())
52                     last = new_element;
53                else
54                     first->prev = new_element;
55                first = new_element;
56           }
57           int pop_front() {
58                if (is_empty())
59                     return 0;
60                int value = first->value;
```

```
61                  if (first->next) {
62                        first = first->next;
63                        delete first->prev;
64                        first->prev = nullptr;
65                  }
66                  else {
67                        delete first;
68                        last = nullptr;
69                        first = nullptr;
70                  }
71                  return value;
72            }
73      private:
74            list_element *last, *first;
75      };
```

Листинг 3 – Пример реализации двусвязного списка

В строках 1—6 определяется тип-элемент списка `list_element`, который содержит сами данные и два указателя: на следующий узел и на предыдущий. Далее идёт реализация непосредственно динамической структуры.

2 АССОЦИАТИВНЫЙ МАССИВ. ХЭШ-ТАБЛИЦА

Хэш-таблица — это структура данных, реализующая интерфейс ассоциативного массива, а именно, она позволяет хранить пары (ключ, данные) и выполнять три операции: добавление новой пары, поиск и удаление пары по ключу.

Число хранимых элементов, делённое на размер массива (число возможных значений хэш-функции), называется коэффициентом заполнения хэш-таблицы.

Выполнение любой операции (поиск, добавление и удаление) в хэш-таблице начинается с вычисления хэш-функции от ключа. Получающееся значение является индексом в массиве. Ситуация, когда для различных ключей получается одно и то же значение хэш-функции, называется коллизией. Ниже рассмотрено несколько вариантов реализаций, которые устраняют данную проблему.

2.1 Прямая адресация

В массиве хранятся указатели на связанные списки пар. Также имеются реализации, где в ячейке таблицы хранится сама пара и указатель на следующий элемент. Коллизии просто приводят к тому, что появляются списки длиной более одного элемента. Пример такой таблицы приведён ниже. Данная реализация может быть усовершенствованная путём использования вместо связанных списков других динамических структур данных.

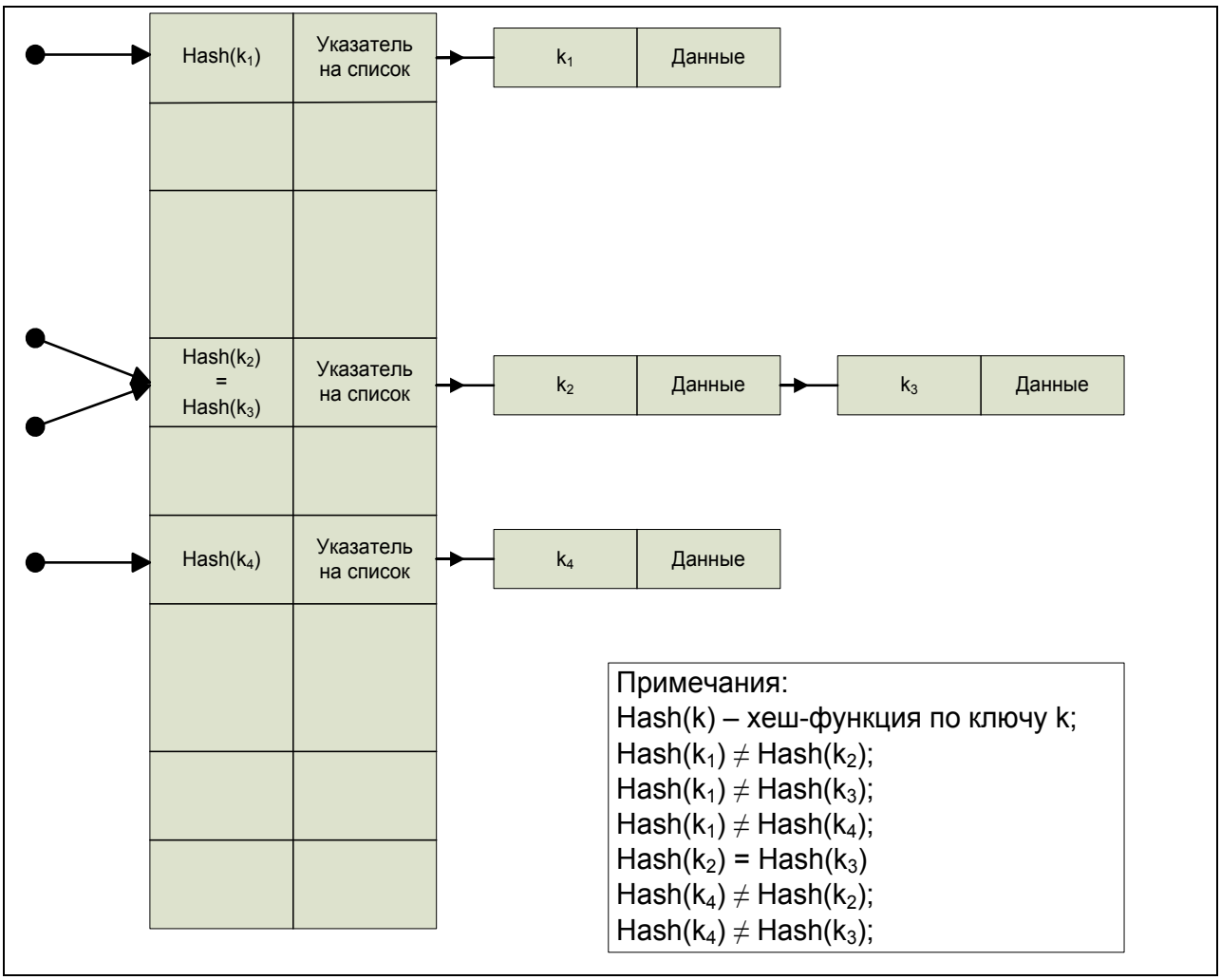

Рисунок 5 — Реализация хэш-таблицы с прямой адресацией

2.2 Открытая адресация

В массиве хранятся сами пары (ключ, данные). В случае возникновения коллизии, алгоритм поиска (удаления, добавления) объекта просто перемещается на ячейку вправо до момента разрешения коллизии. Разрешение коллизии происходит при достижении пустой ячейки или ячейки, в которой хранится пара с заданным ключом. Размер шага смещения вправо может зависеть от значения ключа и вычисляться с помощью второй хэш-функции. Данная техника называется двойным хэшированием с открытой адресацией. Пример такой таблицы приведён ниже.

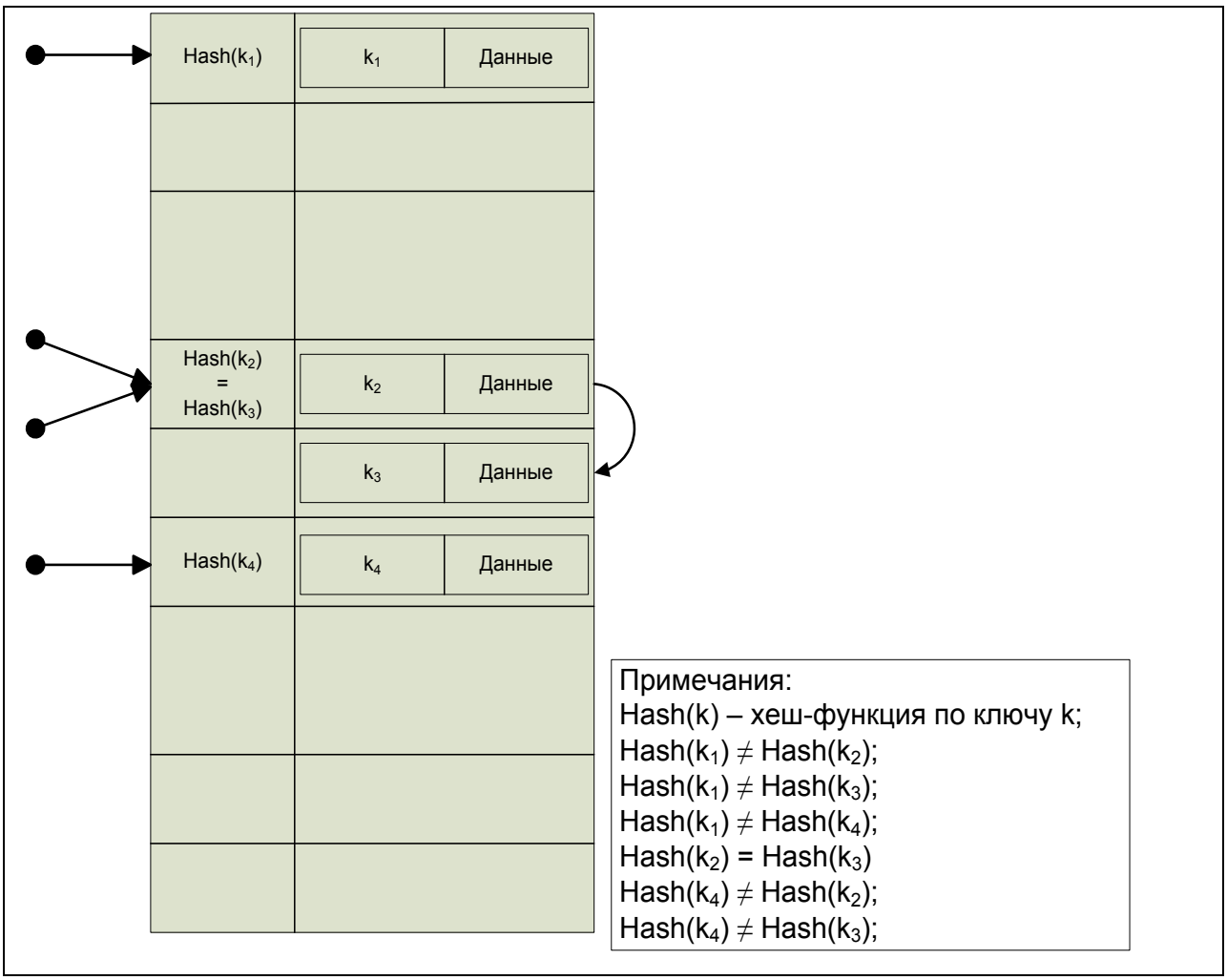

Рисунок 6 — Реализация хэш-таблицы с открытой адресацией

Важное свойство хэш-таблицы состоит в том, что все три операции в среднем выполняются за время O(1). Но при этом не гарантируется, что время выполнения отдельной операции мало. Это связано с тем, что при достижении некоторого значения коэффициента заполнения необходимо осуществлять перестройку индекса хэш-таблицы: увеличить значение размера массива и заново добавить в пустую хэш-таблицу все пары. Для данной структуры требуется память, равная размеру таблицы, следовательно, пространственная сложность алгоритма составляет O(1).

Хэш-таблицы являются очень удобными для реализации ассоциативных массивов, но у них имеется несколько недостатков:

1) Хэш-таблица занимает много памяти даже при нахождении в ней малого количества элементов.

2) Время любого поиска одинаково (всегда нужно рассчитывать хэш-функцию), то есть при поиске ключа, которого нет в таблице, будет расходоваться столько же времени, сколько и на поиск элемента, который там есть, в отличие от поиска в бинарном дереве когда ещё на ранних стадиях можно определить, что искомого элемента нет.

3 ДЕРЕВЬЯ

3.1 Бинарное (двоичное) дерево

Бинарное дерево поиска является типичным примером динамической структуры данных. Дадим его описание.

Основное свойство *дерева* заключаются в том, что узел указывается только одним другим узлом, называемым *родительским*. Это относится ко всем элементам дерева, исключая *корень*.

Определяющее свойство *бинарного дерева* — наличие у каждого узла двух связей (левой и правой). Связи указывают на другие двоичные деревья. Для выполнения поиска каждый узел также имеет поле со значением ключа. Исходя из всего вышесказанного, дадим несколько определений, касающихся бинарного дерева поиска.

Дерево бинарного поиска — это бинарное дерево, с каждым из узлов которого связан ключ, причём ключ в любом узле больше ключа во всех узлах левого поддерева или меньше ключа во всех узлах правого поддерева этого узла.

Под высотой дерева (поддерева) подразумевается количество узлов от корня к самому дальнему узлу.

Полностью сбалансированное бинарное дерево – это дерево, для которого соблюдается правило, что для каждой вершины высота её правого и левого поддерева отличается не более чем на 1.

Полностью несбалансированное бинарное дерево — это дерево, высота которого равна количеству всех узлов, содержащихся в нём.

Узел с двумя нулевыми связями называется листом.

Двоичные деревья поиска, в которых элементами являются числа, представлены на рисунках ниже. Они образованны из одного набора целочисленных ключей: 1, 2, 4, 5, 6, 9 и 10. Эти рисунки показывают два крайних случая: полностью сбалансированное бинарное дерево и полностью несбалансированное дерево, когда оно вырождается в связанный список.

Если данные деревья были сконструированы из одних и тех же ключей, от чего же такое различие? Дело просто в том, что структура бинарного дерева поиска в большей степени зависит от последовательности вставки ключей, а не от их значений. В первом случае ключи вставляются в следующей последовательности: 5, 2, 9, 1, 4, 6, 10. Тогда как во втором — 1, 2, 4, 5, 6, 9, 10. Отсюда вытекает основной недостаток бинарных деревьев

поиска — зависимость структуры дерева, а следовательно, и всех его показателей, от последовательности добавления новых элементов.

Рассмотренное бинарное дерево поиска хранит целочисленные ключи. Вместо них могут быть использованы любые типы данных, для которых выполняются операции сравнения (больше, меньше и равно). Это вещественные числа и массивы (строки), которые сравниваются поэлементно.

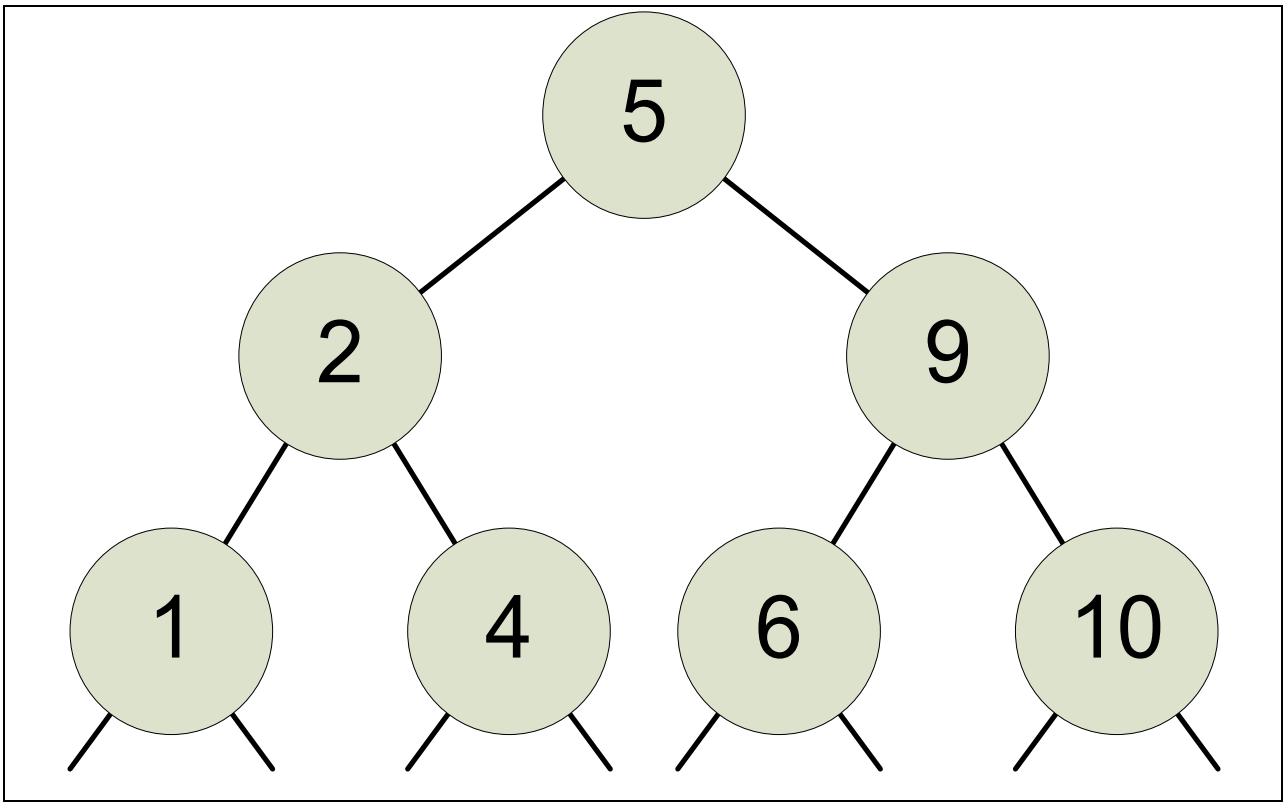

Рисунок 7 — Полностью сбалансированное бинарное дерево

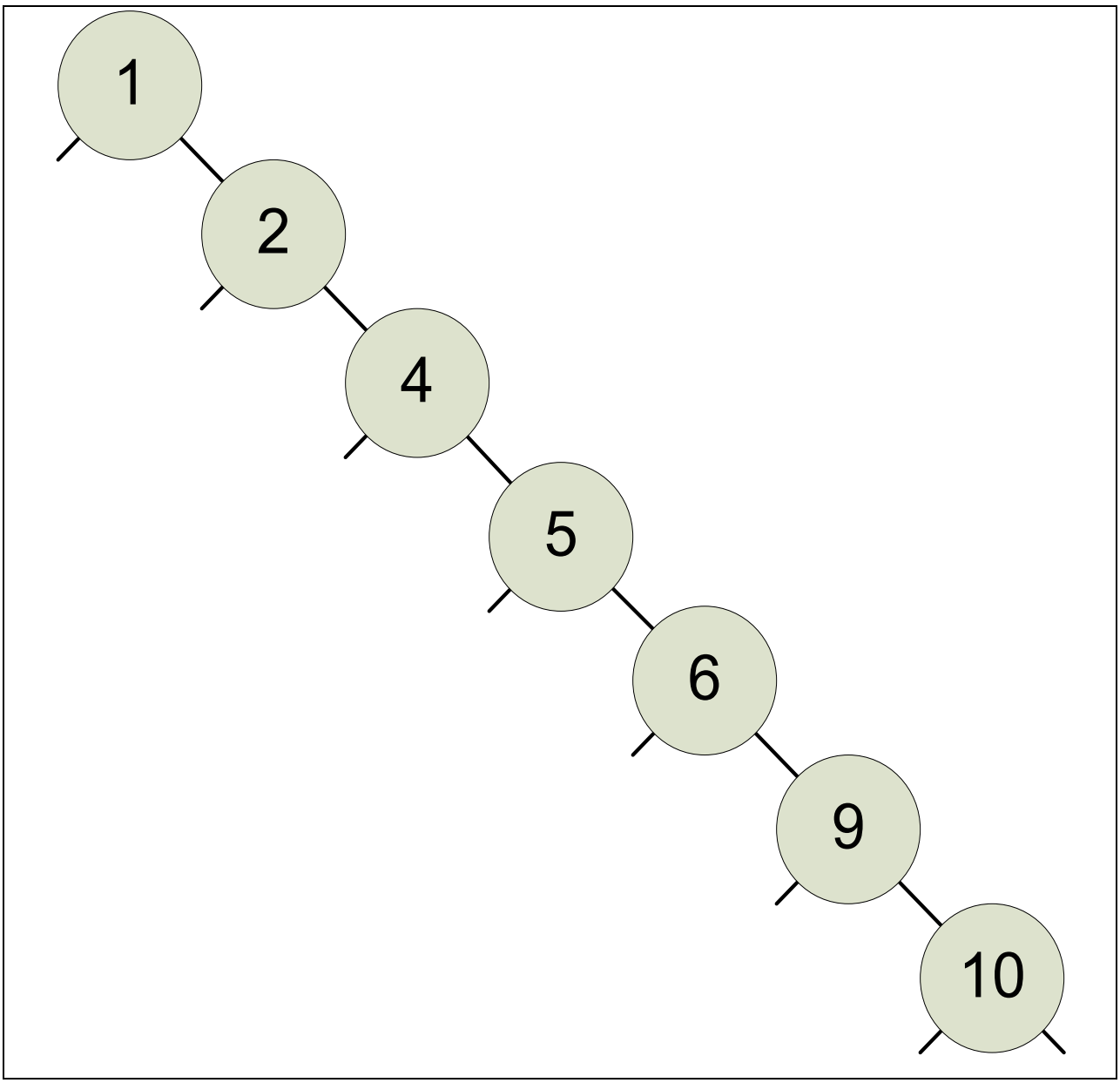

Рисунок 8 — Полностью несбалансированное бинарное дерево (вырождается в связанный список)

Основными функциями любой динамической структуры данных являются поиск, вставка и удаление. Собственно, из названия этих операций и вытекает их назначение. Рассмотрим алгоритм работы каждой из них более подробно на примере бинарного дерева поиска.

3.1.1 Поиск

Входные данные: дерево T, ключ K.

Выходные данные: ссылка на узел с ключом K либо нулевая ссылка.

Задача: проверить, есть ли узел с ключом K в дереве T; если есть, вернуть ссылку на этот узел, иначе — нулевую ссылку.

Алгоритм 1 — Поиск ключа в бинарном дереве

Если дерево пустое, вернуть нулевую ссылку и остановиться.

Иначе сравнить K со значением ключа (X) корневого узла.

Если K = X, выдать ссылку на это узел и остановиться.

Если K > X, рекурсивно искать ключ K в правом поддереве T.

Если K < X, рекурсивно искать ключ K в левом поддереве T.

3.1.2 Вставка

Входные данные: дерево T, ключ K.

Выходные данные: ссылка на узел с ключом K.

Задача: добавить узел с ключом K в дерево T.

Алгоритм 2 — Добавление пары ключ-значение в бинарное дерево

Если дерево пустое, создать дерево с одним узлом, ключ которого равен K, выдать на него ссылку и остановится. Иначе сравнить K со значением ключа (X) корневого узла.

Если K = X, выдать ссылку на этот узел и остановится.

Если K > X, рекурсивно добавить ключ K в правом поддереве T.

Если K < X, рекурсивно добавить ключ K в левом поддереве T.

3.1.3 Удаление

Входные данные: дерево T, ключ K.

Выходные данные: отсутствуют.

Задача: удалить из дерева T узел с ключом K.

Алгоритм 3 — Удаление узла из бинарного дерева по ключу

Если дерево пустое, остановиться.

Иначе сравнить K со значением ключа (X) корневого узла.

Если K > X, рекурсивно удалить ключ K в правом поддереве T.

Если K < X, рекурсивно удалить ключ K в левом поддереве T.

Если K = X, необходимо рассмотреть три случая:

Если обе связи равны нулю, то удаляем текущий узел и обнуляем на него родительскую связь.

Если одна из связей равна нулю, то заменяем родительскую связь на ненулевую. Удаляем текущий узел.

Если обе связи не равны нулю, то ищем самый левый узел правого поддерева (M). Копируем значения ключа узла M в узел для удаления. Заменяем родительскую связь узла M на правую связь узла M (может равняться нулю). Удаляем узел M.

3.1.4 Характеристики производительности

Определение временной сложности алгоритма обработки бинарного дерева поиска может быть сведено к анализу его структуры. В лучшем случае можно рассчитывать на логарифмическую вычислительную сложность O(ln N), тогда как в худшем — только на линейную O(N).

Для хранения бинарного дерева требуется память, кратная количеству элементов, следовательно, пространственная сложность алгоритма линейна — O(N).

Как было сказано выше, характеристики бинарного дерева поиска сильно зависят от порядка добавления новых элементов. Этот отрицательный момент успешно решается в самобалансирующихся древовидных структурах, таких как АВЛ-дерево и красно-чёрное дерево. Они имеют встроенные алгоритмы, которые поддерживают их в «хорошей» форме.

3.2 АВЛ-дерево

АВЛ-дерево — самобалансирующееся по высоте двоичное дерево поиска.

АВЛ-дерево является полностью сбалансированным. Пример такого дерева был представлен выше. Алгоритмы поиска, вставки и удаления аналогичны бинарным деревьям. Но после любой операции, которая изменяет структуру дерева (вставка или удаление) проверяется, нарушился ли баланс (высоты двух поддеревьев для какого-либо узла различаются больше чем на 1). Если баланс нарушен, в действие вступают алгоритмы самобалансировки, которые основаны на определённых операциях, называемых «вращением». Всего используют 4 вида вращений.

3.2.1 Малое правое вращение

Данное вращение используется тогда, когда ((высота b – высота L) = 2) и (высота C ≤ высота R). Иллюстрация данного случая показана ниже.

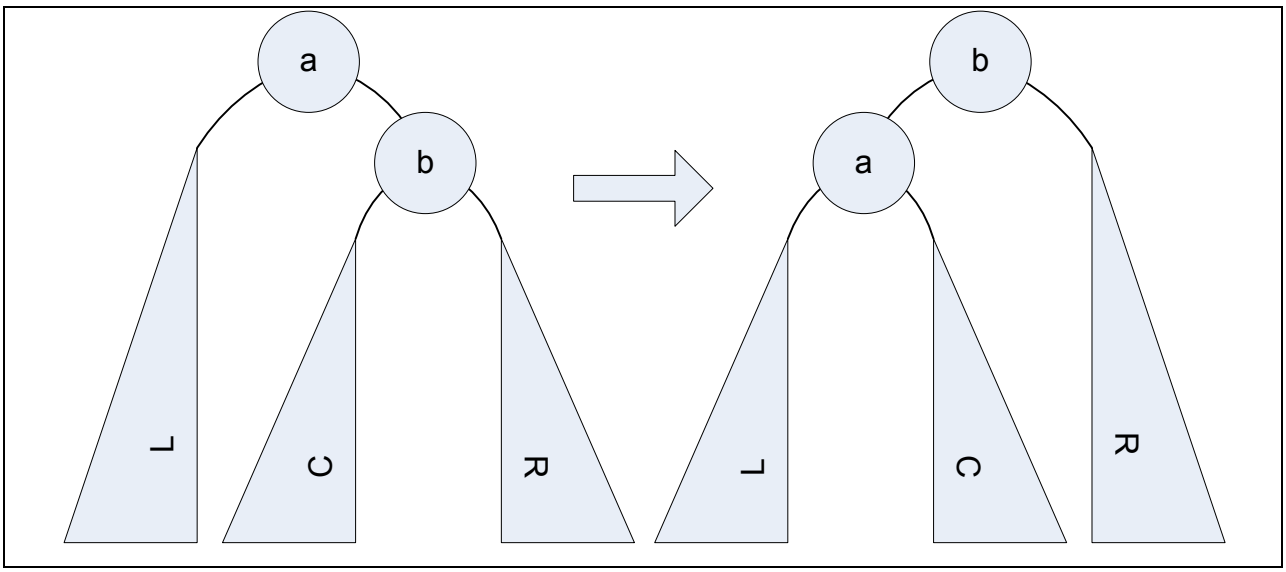

Рисунок 9 — Малое правое вращение в АВЛ-деревьях

3.2.2 Большое правое вращение

Данное вращение используется тогда, когда ((высота b – высота L) = 2) и (высота C > высота R). Иллюстрация данного случая показана ниже.

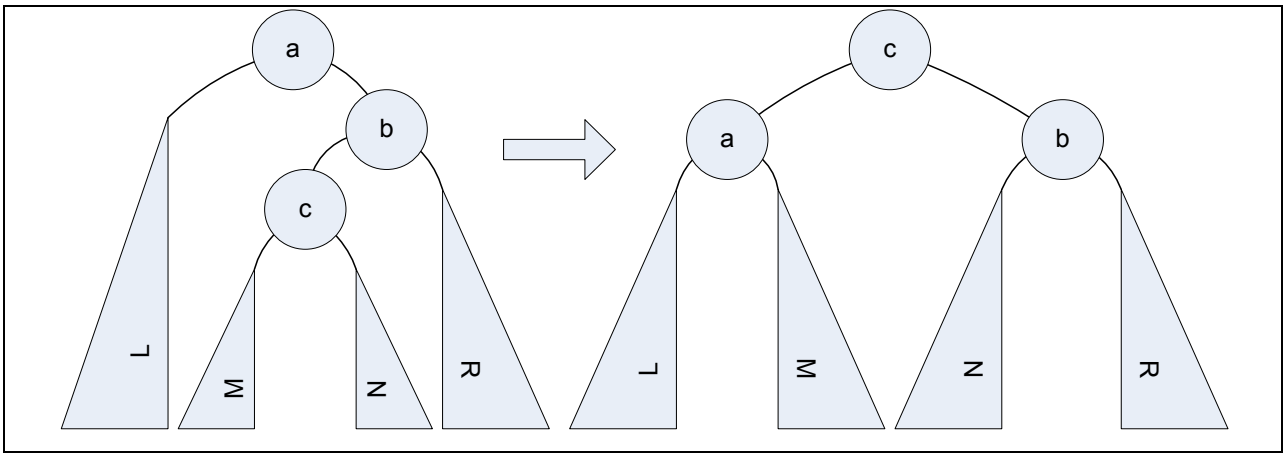

Рисунок 10 — Большое правое вращение в АВЛ-деревьях

3.2.3 Малое левое вращение.

Данное вращение используется тогда, когда ((высота b – высота R) = 2) и (высота C ≤ высота L). Иллюстрация данного случая показана ниже.

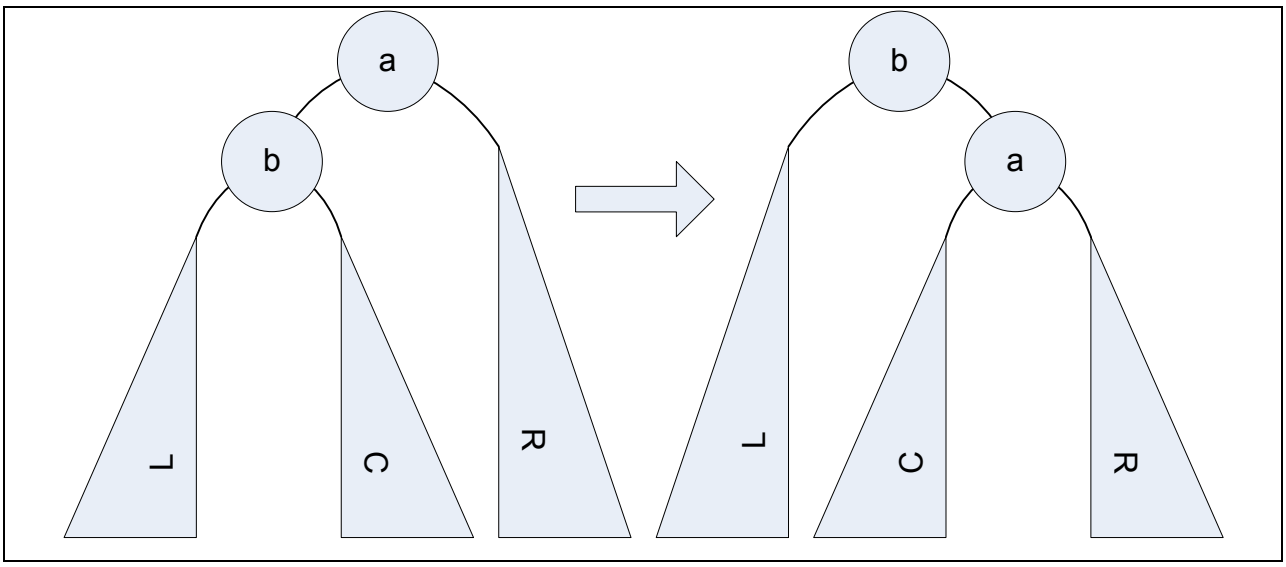

Рисунок 11 — Малое левое вращение в АВЛ-деревьях

3.2.4 Большое левое вращение

Данное вращение используется тогда, когда ((высота b – высота R) = 2) и (высота C > высота L). Иллюстрация данного случая показана ниже.

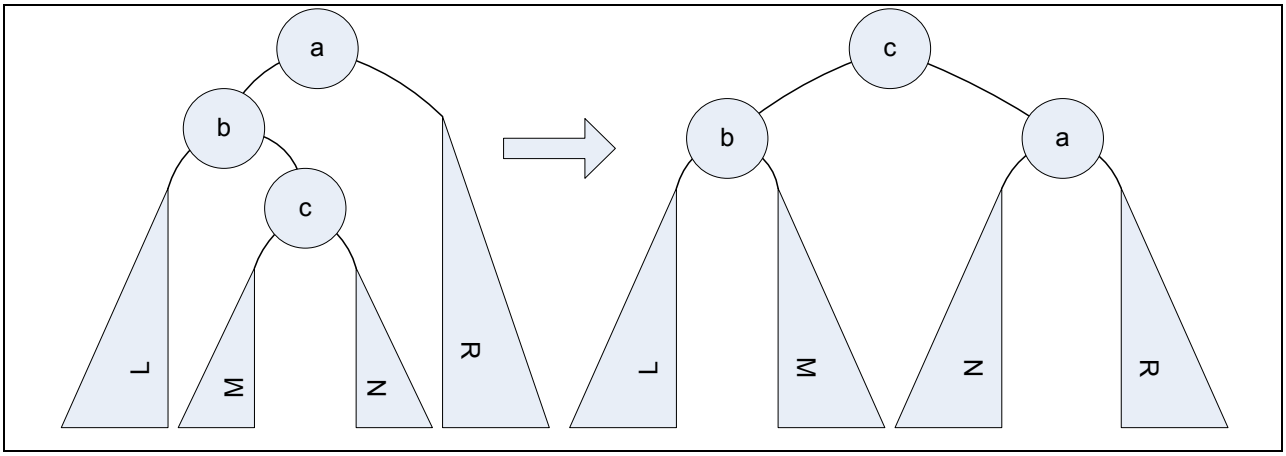

Рисунок 12 — Большое левое вращение в АВЛ-деревьях

Достоинство таких деревьев очевидно — они всегда готовы обеспечить наилучшую вычислительную сложность поиска информации для бинарных деревьев, а именно O(ln N). Но, как видно из выше сказанного, имеется и очень неприятная особенность — необходимость достаточно часто вызывать процедуры балансировки, что пагубно сказывается на производительности алгоритмов вставки и удаления.

Для АВЛ-дерева требуется память, кратная количеству элементов, следовательно, пространственная сложность алгоритма линейна O(N), как и в случае бинарного дерева поиска.

АВЛ-деревья редко используются на практике из-за больших накладных расходов при вставке и удалении элементов. В качестве альтернативы данной структуре чаще всего

используются красно-чёрные деревья. Они обладают хорошей скоростью поиска, как и АВЛ-деревья, но при этом реже выполняют процедуру балансировки.

3.3 Красно-чёрное дерево

Красно-чёрное дерево — это самобалансирующееся двоичное дерево поиска, гарантирующее логарифмический рост высоты дерева от числа узлов и быстро выполняющее основные операции дерева поиска: добавление, удаление и поиск узла.

Сбалансированность данного типа дерева достигается введением дополнительных правил формирования дерева, а также добавлением специального атрибута узла — цвета. Этот атрибут может принимать одно из двух возможных значений: «чёрный» или «красный».

Правила формирования красно-чёрного дерева:

1. Каждый узел покрашен либо в «красный», либо в «чёрный» цвет;
2. Любой «красный» узел имеет только «чёрных» потомков;
3. Все пути от корня к листьям имеют одинаковое число чёрных узлов.

При этом для удобства, листьями красно-чёрного дерева считаются фиктивные «нулевые» узлы, не содержащие данных. Пример данного дерева представлен ниже.

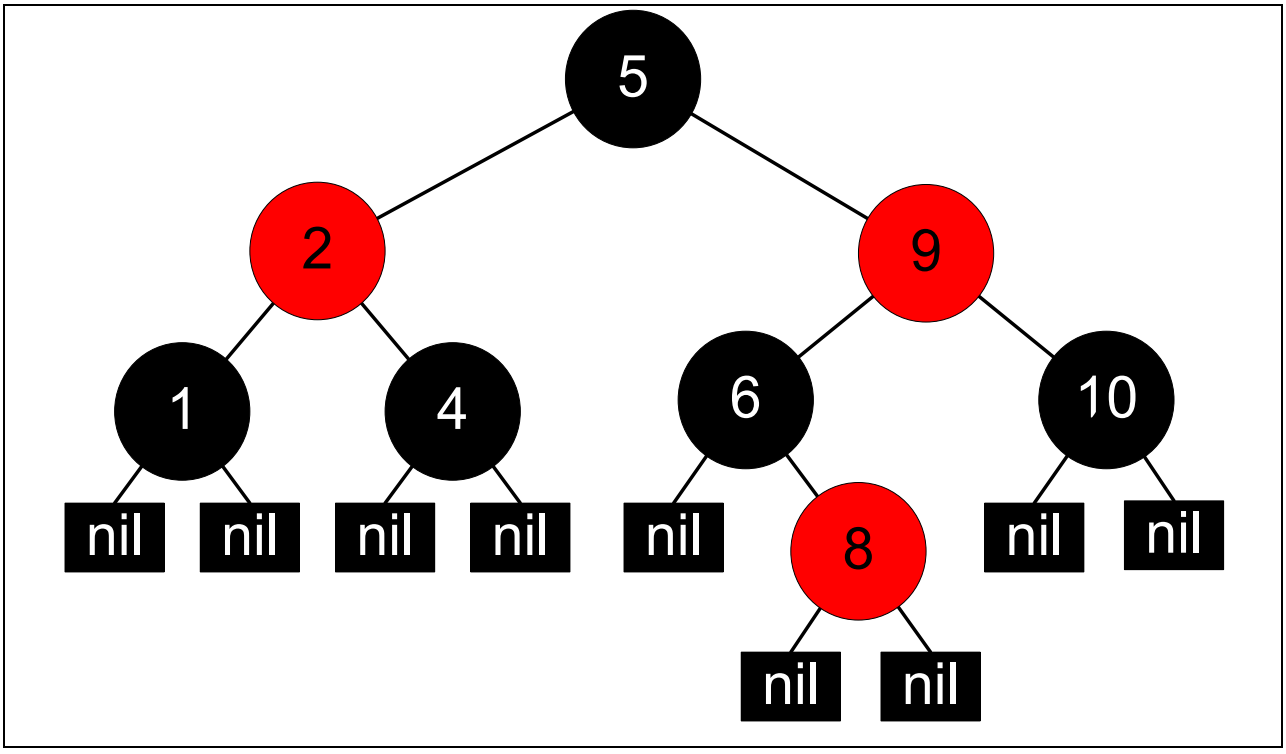

Рисунок 13 — Пример красно-чёрного дерева

При добавлении в дерево новый узел окрашивается в «красный» цвет. Далее проверяются правила, приведённые выше; если они нарушаются, дерево балансируется. Балансировка используется также и при удалении элемента.

Проиллюстрируем вставку ключа «7» в дерево, представленное выше. Добавление узла производится по алгоритму 2. После вставки узел окрашивается, как было сказано ранее, в «красный» цвет. Результат этой операции изображен ниже.

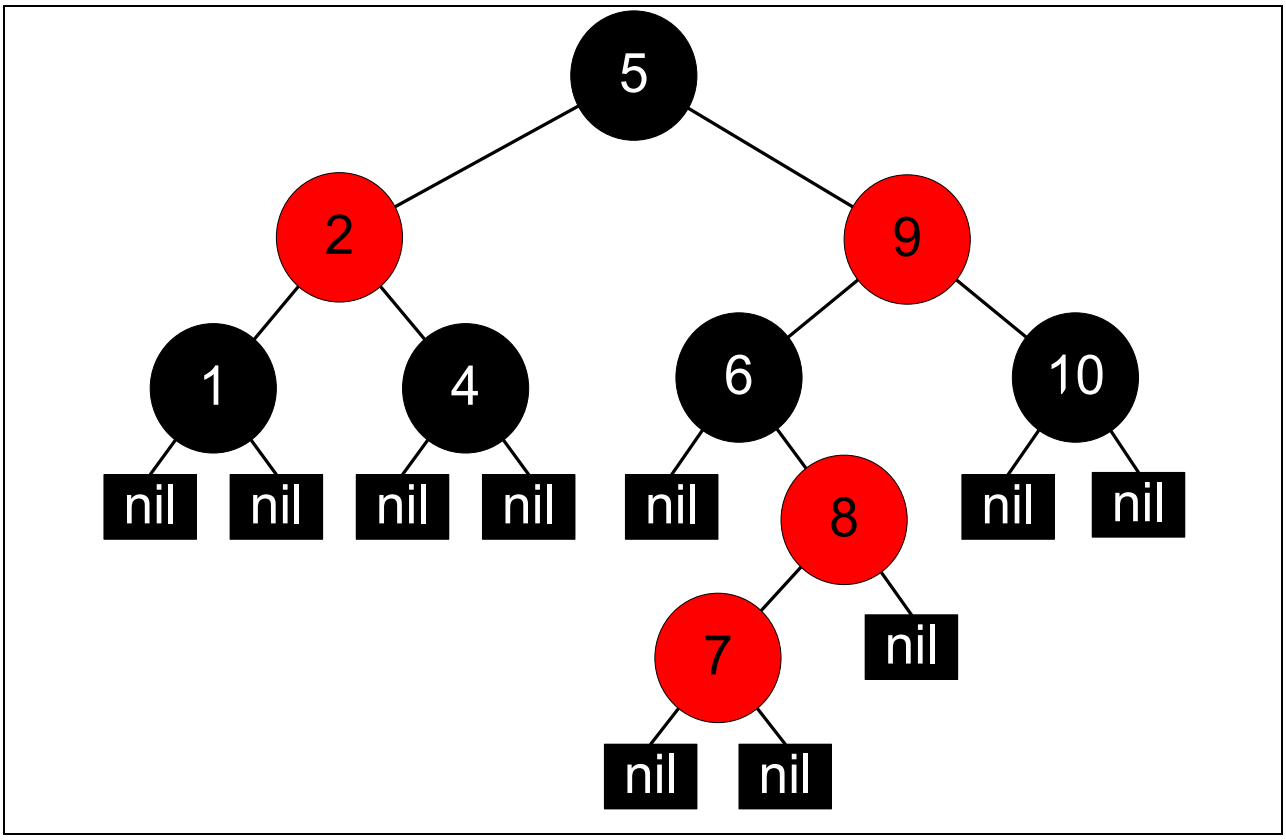

Рисунок 14 — Несбалансированное красно-чёрное дерево

Полученное дерево противоречит правилу 2, поэтому нам следует начать процедуру балансировки. Для данного случая нужно использовать вращения, которые рассматривались ранее в АВЛ-деревьях, а именно большое правое вращение. Полученный результат приведён на рисунке. В некоторых случаях обходятся без вращения узлов, довольствуясь перекраской родительского узла.

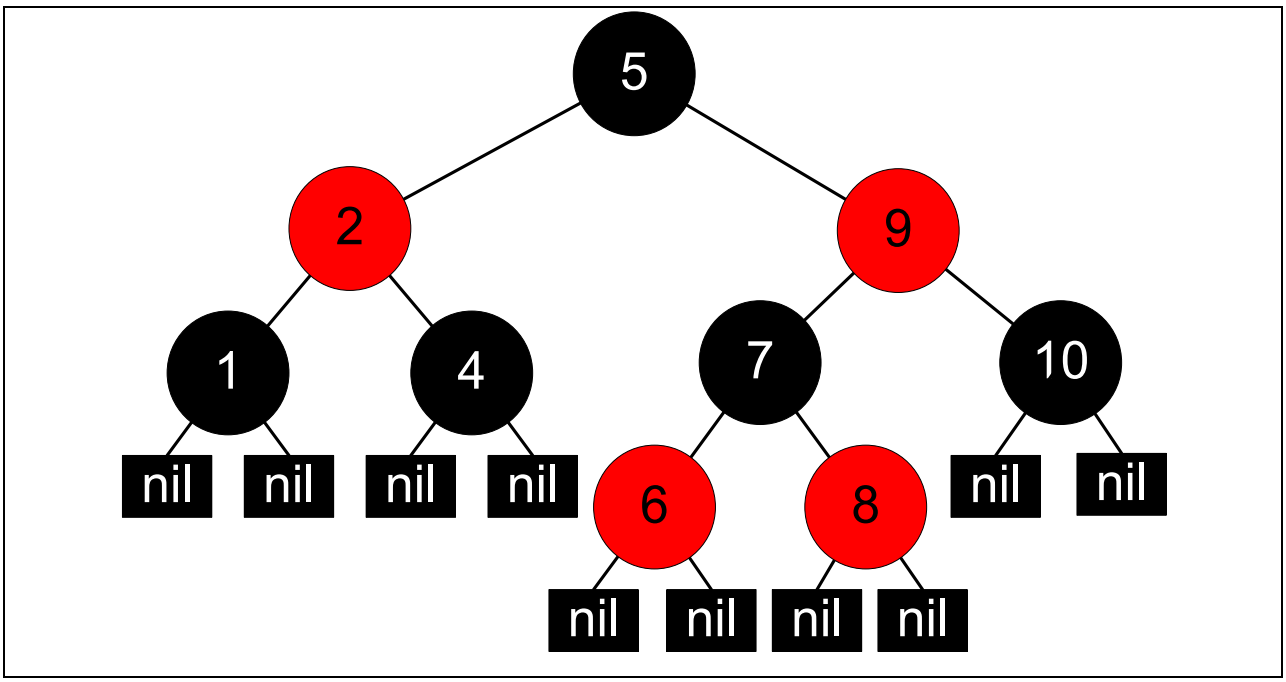

Рисунок 15 — Сбалансированное красно-чёрное дерево

Красно-чёрные деревья имеют вычислительную сложность поиска, одинаковую с АВЛ-деревьями, а именно O(ln N). Причём поддержание структуры в «оптимальной форме» требует на порядок меньших усилий. Для данной структуры требуется память, кратная количеству элементов, следовательно, пространственная сложность алгоритма линейна — O(N), как и в случае бинарного дерева поиска.

4 БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Роберт Седжвик, Алгоритмы на C++. Фундаментальные алгоритмы и структуры данных / пер. с англ. (Algorithms in C++) — М.: Вильямс. — 2011 г. — 1056 с.