«Найти-добавить-удалить»

Постановка задачи

Задано множество S мощностью M. Требуется предложить структуру данных для хранения элементов этого множества, пригодную для выполнения следующих операций:

- поиск элемента по значению
- добавление нового (возможно, уникального) элемента (после поиска)
- удаление элемента (после поиска)

Ни одна из этих операций не должна выполняться с трудоёмкостью O(N), где N – количество хранимых элементов!

Допустимые структуры данных

Основные:

- массив
- бинарное поисковое дерево
- хэш-таблица

Дополнительные (сбалансированные деревья):

- АВЛ – деревья
- сильно ветвящиеся деревья
- красно-чёрные деревья

Использование массивов

Если величина M невелика, можно завести булевский массив из M элементов, и обозначать в этом массиве наличие или отсутствие соответствующего элемента.

Вариант такого подхода – битовый массив.

Если уникальности элементов не требуется, массив должен быть числовым, и в нём обозначается количество хранимых элементов.

Корневое дерево

Корневое дерево определяется как конечное множество T одного или более узлов со следующими свойствами:

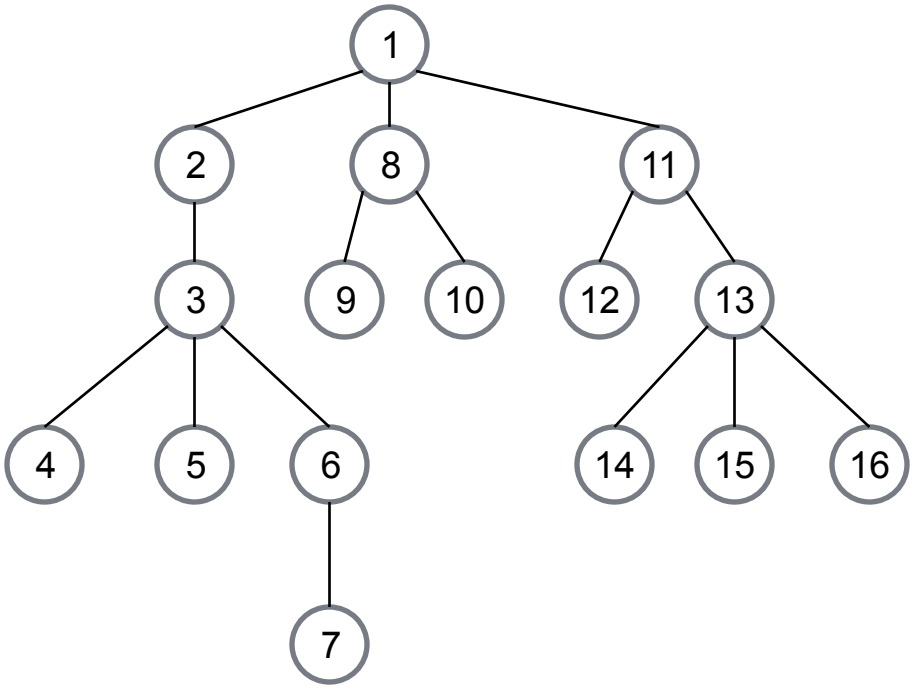
- существует один корень дерева T ;
- остальные узлы (за исключением корня) распределены среди непересекающихся множеств $T_1,...,T_m$, и каждое из множеств является *корневым деревом*; деревья $T_1,...,T_m$ называются **поддеревьями** корня T

Обход дерева

- Обход дерева – операция, связанная с *посещением* его вершин (под посещением понимается любая операция над вершиной дерева, не затрагивающая структуру дерева)
- При обходе каждая вершина должна быть посещена ровно один раз

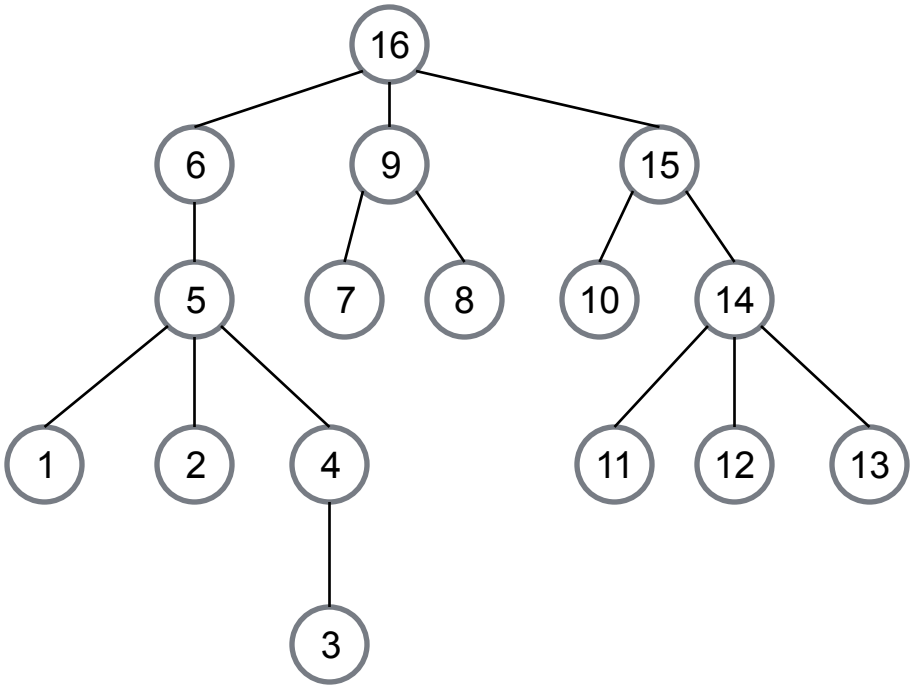
Прямой обход дерева

- посетить корень
- обойти все поддеревья в направлении слева направо



Обратный обход дерева

- обойти все поддеревья в направлении слева направо
- посетить корень



Бинарное дерево

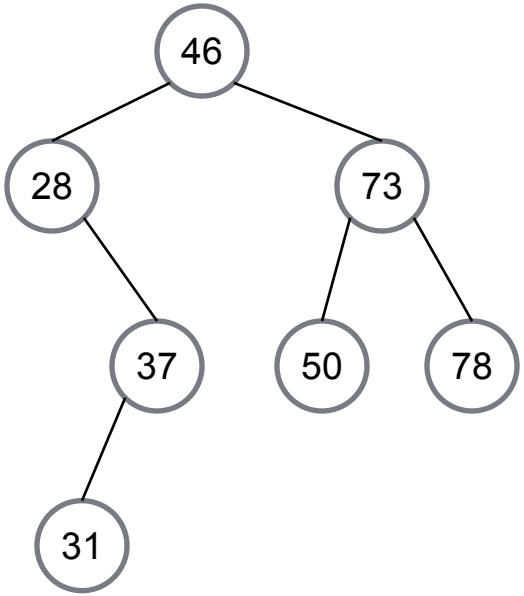
Бинарное (двоичное) дерево — ориентированное дерево, в котором число сыновей каждой вершины не превосходят 2.

На бинарном дереве основаны следующие структуры данных:

- бинарное поисковое дерево
- двоичная куча
- красно-чёрное дерево
- АВЛ-дерево
- Фибоначчиева куча

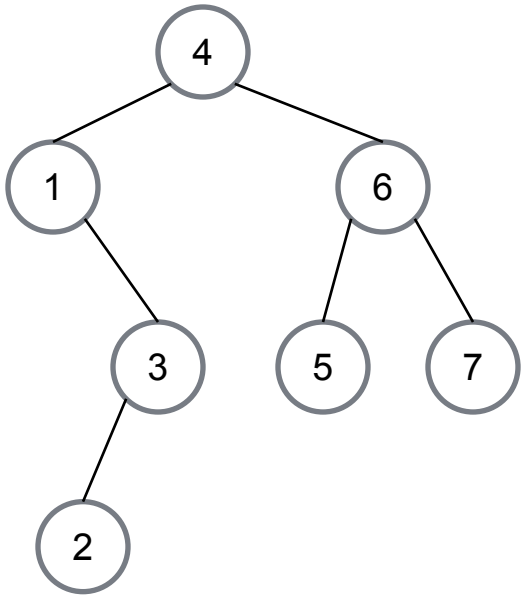
Структура бинарного поискового дерева

- Сыновья каждой вершины различаются: выделяются левый и правый сын
- Значения хранятся в каждой вершине дерева
- Для любого узла значения в левом поддереве меньше, а значения в правом поддереве – больше значения, хранящегося в корне



Концевой обход бинарного дерева

- Обойти левое поддерево
- Посетить корень
- Обойти правое поддерево



Если под посещением понимать вывод хранящегося в вершине значения, то концевым обходом БПД можно вывести отсортированное содержимое дерева.

Поиск значения в БПД

Пусть K – искомое значение, T – значение в корне дерева

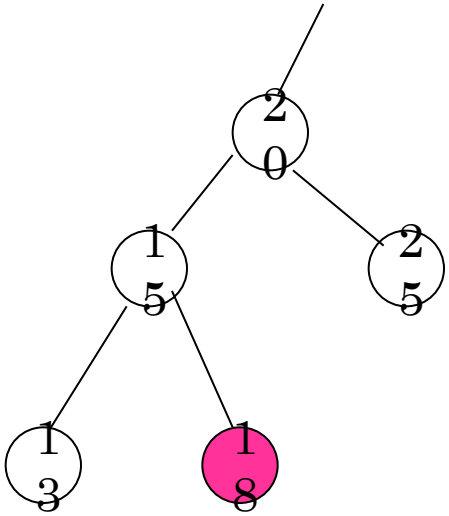
```
if (K==T) поиск завершен успешно
else if (K<T)
  if (есть левый сын)
    выполнить поиск в левом поддереве
  else
    поиск завершен неудачно
else
  if (есть правый сын)
    выполнить поиск в правом поддереве
  else
    поиск завершен неудачно
```

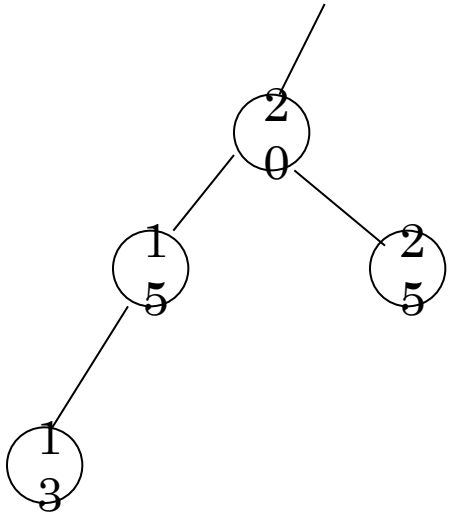
Добавление значения в БПД

- если БПД пусто, создаем единственную вершину и делаем ее корнем;
- иначе выполняем поиск вершины с добавляемым значением;
- если поиск успешен, вставка невозможна;
- в противном случае добавляем новую вершину и делаем ее сыном (левым или правым) той вершины, на которой мы остановились в процессе поиска

Удаление вершины из БПД

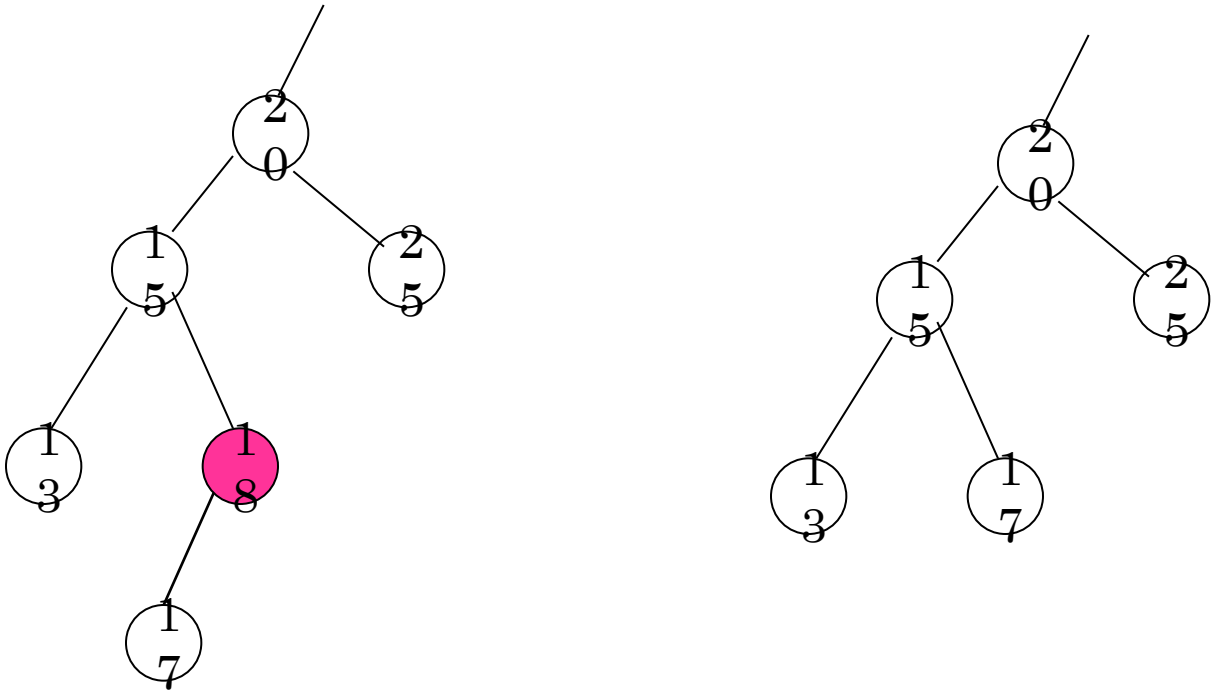
- Выполняем поиск удаляемой вершины. Если он завершился неудачно, завершаем работу.
- Если удаляемая вершина – лист, попросту удаляем её и ссылку на неё





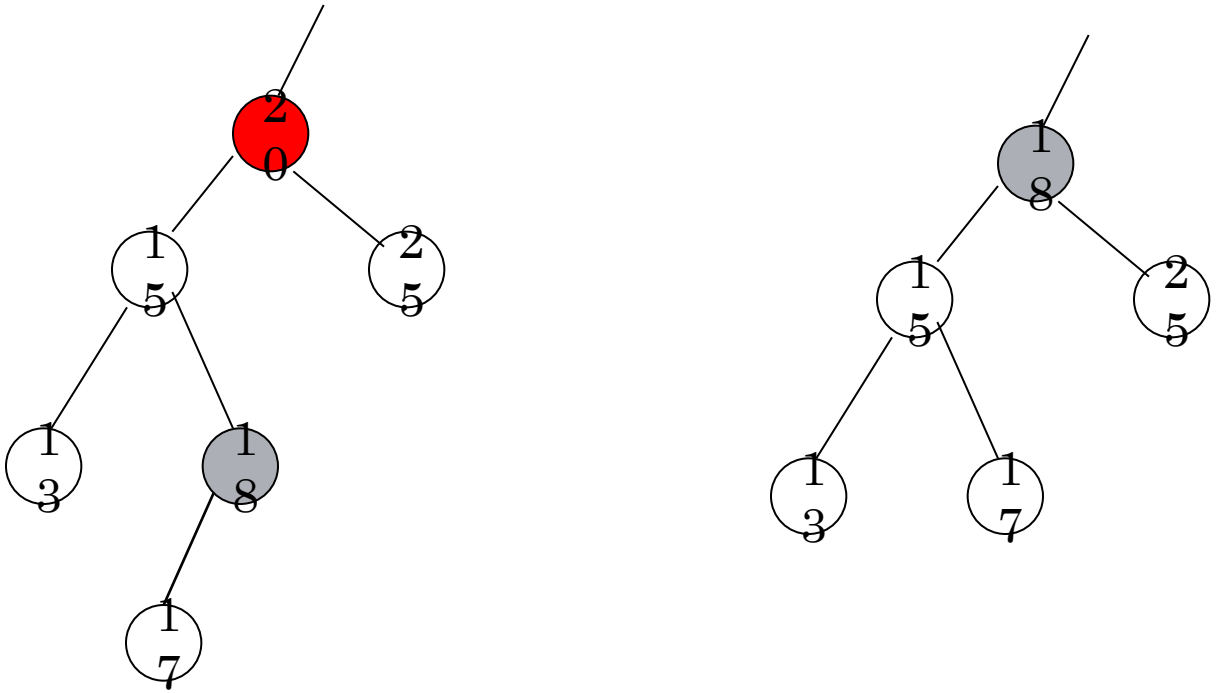
Удаление вершины из БПД (продолжение)

- Если удаляемая вершина A имеет одного сына, делаем этого сына сыном отца вершины A (или корнем, если удаляемая вершина – корень). При этом тип сына (левый или правый) может измениться



Удаление вершины из БПД (окончание)

- Если удаляемая вершина A имеет двух сыновей, находим самую правую вершину в левом поддереве, корнем которого является A (или самую левую – в правом поддереве) – вершину B. Информацию из вершины B переносим в A, а саму вершину B удаляем, как было сказано ранее.



Реализация БПД

Реализация бинарного поискового дерева на C# приведена в примере BST.zip

Хэширование

Пусть количество хранимых элементов N много меньше M (мощности множества S). Введём *хэш-функцию*

$h : S \rightarrow [1..N]$

и создадим массив из N элементов (*хэш-таблицу*).

Суть хэширования состоит в следующем: ***помещаем элемент s из множества S в хэш-таблицу по индексу h(s).***

Ситуация, когда необходимо поместить элементы s_1 и s_2, но $h(s_1) = h(s_2)$, называется коллизией.

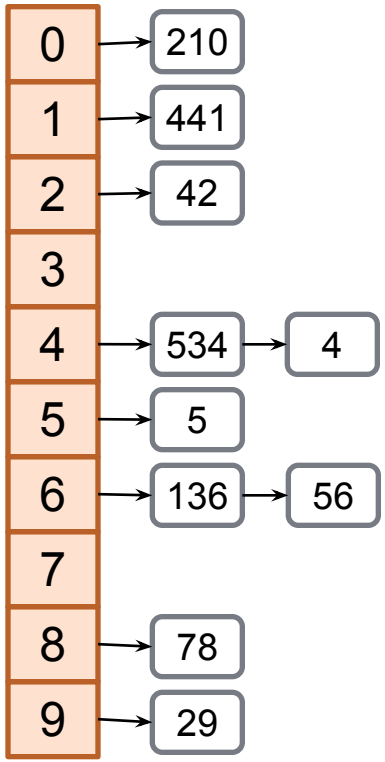
Для разрешения коллизий применяют:

- открытое хэширование
- закрытое хэширование

Открытое хэширование

Элементы, находящиеся в коллизии друг с другом, образуют список. Хэш-таблица хранит лишь информацию о начале списка.

Пусть хэш-функция возвращает остаток от деления на 10:



Примеры хэш-функций

- остаток от деления на N – объём хэш-таблицы (N лучше сделать простым, нежелательно делать $N = 10^k$ при хранении числовых данных)
- остаток от деления старших разрядов на N
- сумма цифр числа
- сумма квадратов цифр числа

Все эти хэш-функции некачественны, поскольку при определённых условиях вероятность коллизий будет велика!

Примеры хэш-функций (продолжение)

- Полиномиальная хэш-функция

Пусть дана строка S = $s_1 s_2 \ldots s_T$, состоящая из цифр от 0 до 9. Тогда значение полиномиальной хеш-функции H(S, x, N) вычисляется следующим образом:

$$H(S, x, N) = \left(\sum_{i=1}^{T} s_i \bullet x^{i-1}\right) \bmod N$$

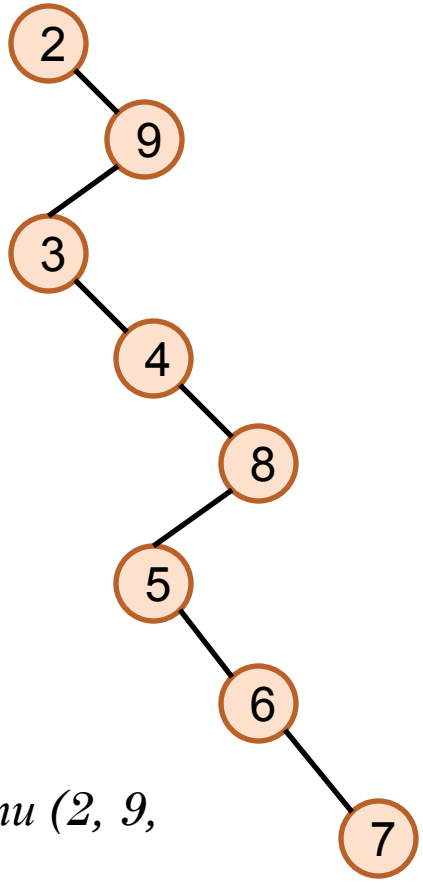
- Расчёт контрольной суммы (CRC)

Неудобные операции для хэш-таблиц

- найти минимальный (максимальный) элемент;
- вывести все элементы, соблюдая упорядоченность;
- найти элемент, ближайший к искомому (или отстоящий от искомого не более чем на K);
- выполнить поиск по началу строки.

Недостаток бинарных поисковых деревьев

- При создании дерева последовательностью вставок длины различных ветвей могут сильно различаться:

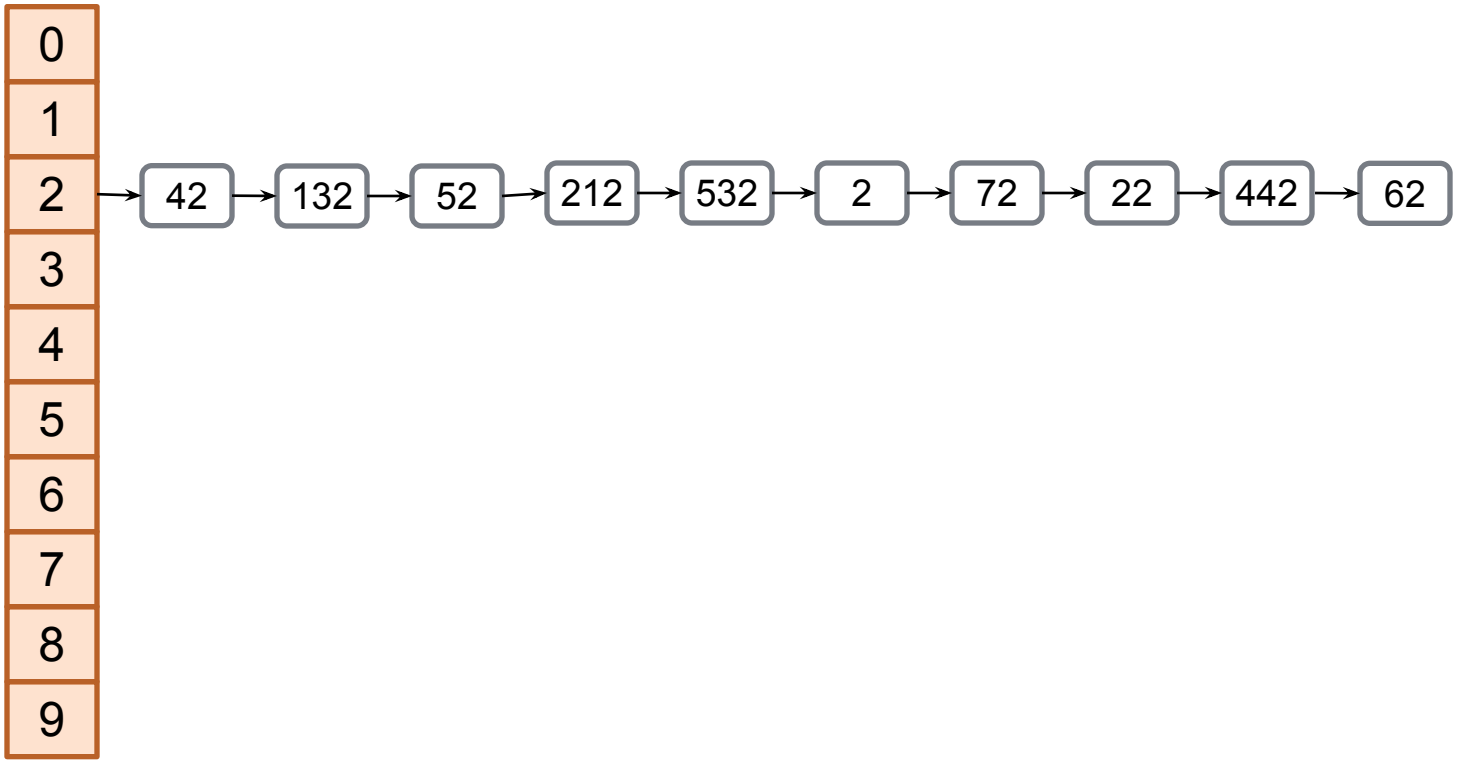


БПД, созданное для последовательности (2, 9, 3, 4, 8, 5, 6, 7)

Недостаток хэш-таблиц

- При неудачном выборе хэш-функции все вставленные элементы вступят в коллизию друг с другом

  Пусть хэш-функция возвращает остаток от деления на 10, а добавляются элементы с одинаковой последней цифрой:



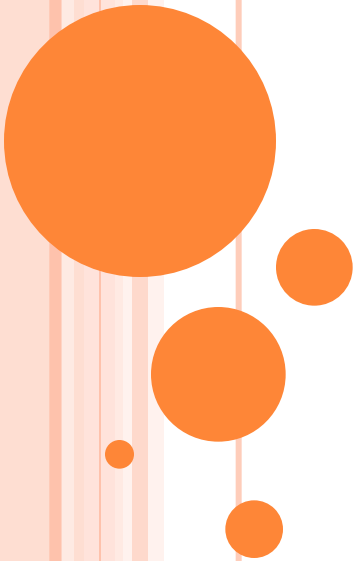
Подходы к решению проблемы

Создать древовидные структуры, у которых длины ветвей будут не сильно отличаться друг от друга (сбалансированные деревья)

При этом поддержание сбалансированности должно выполняться с трудоёмкостью не большей, чем O(log N).

Варианты решения:

- АВЛ-деревья;
- сильно ветвящиеся деревья;
- красно-чёрные деревья



ДВОИЧНАЯ КУЧА

Операции над двоичной кучей

Двоичная куча – структура данных, хранящая элементы, над которыми определено отношение «меньше». Она предназначена для выполнения следующего набора операций:

- добавить новый элемент;
- определить и, возможно, удалить минимальный элемент.

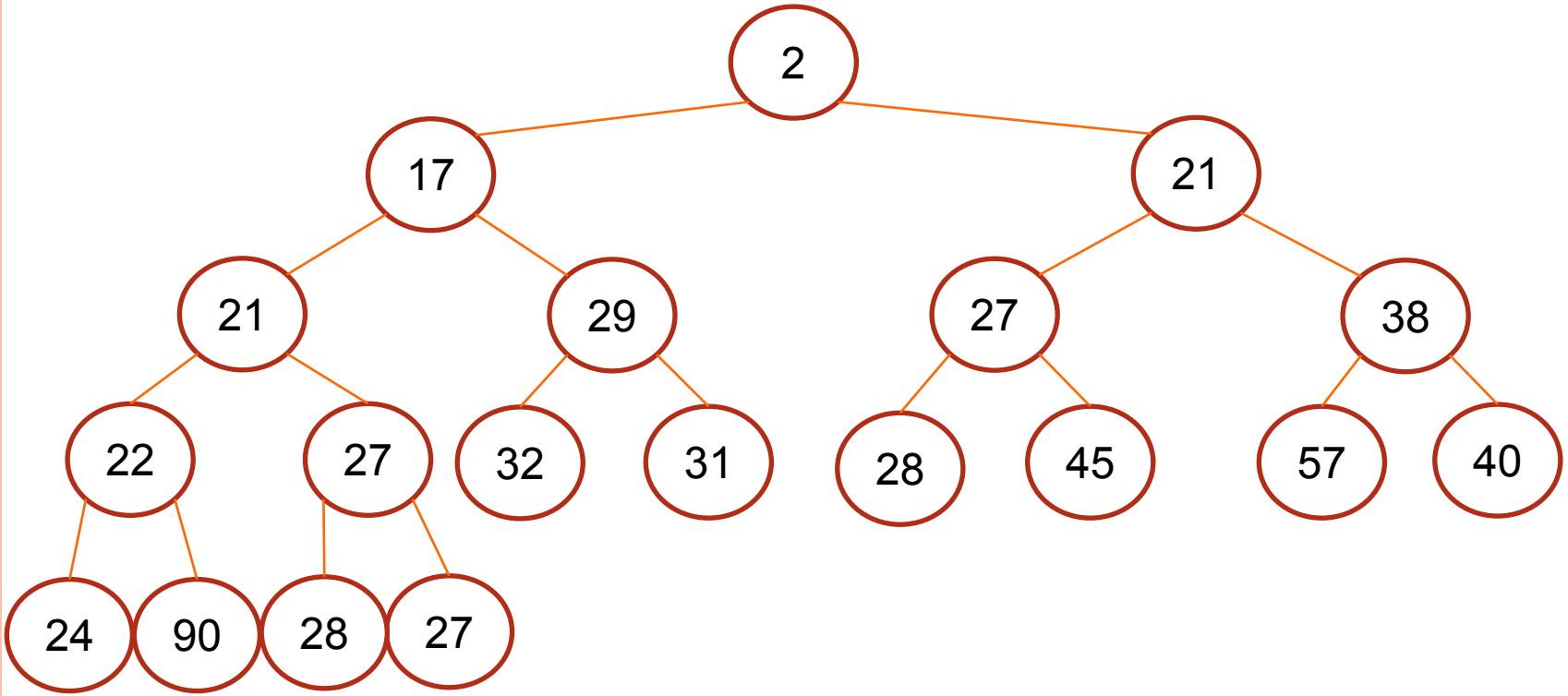
Двоичная куча как дерево

Двоичная куча – это двоичное дерево, полностью сбалансированное на всех уровнях, кроме последнего. Все вершины последнего уровня максимально прижаты влево.

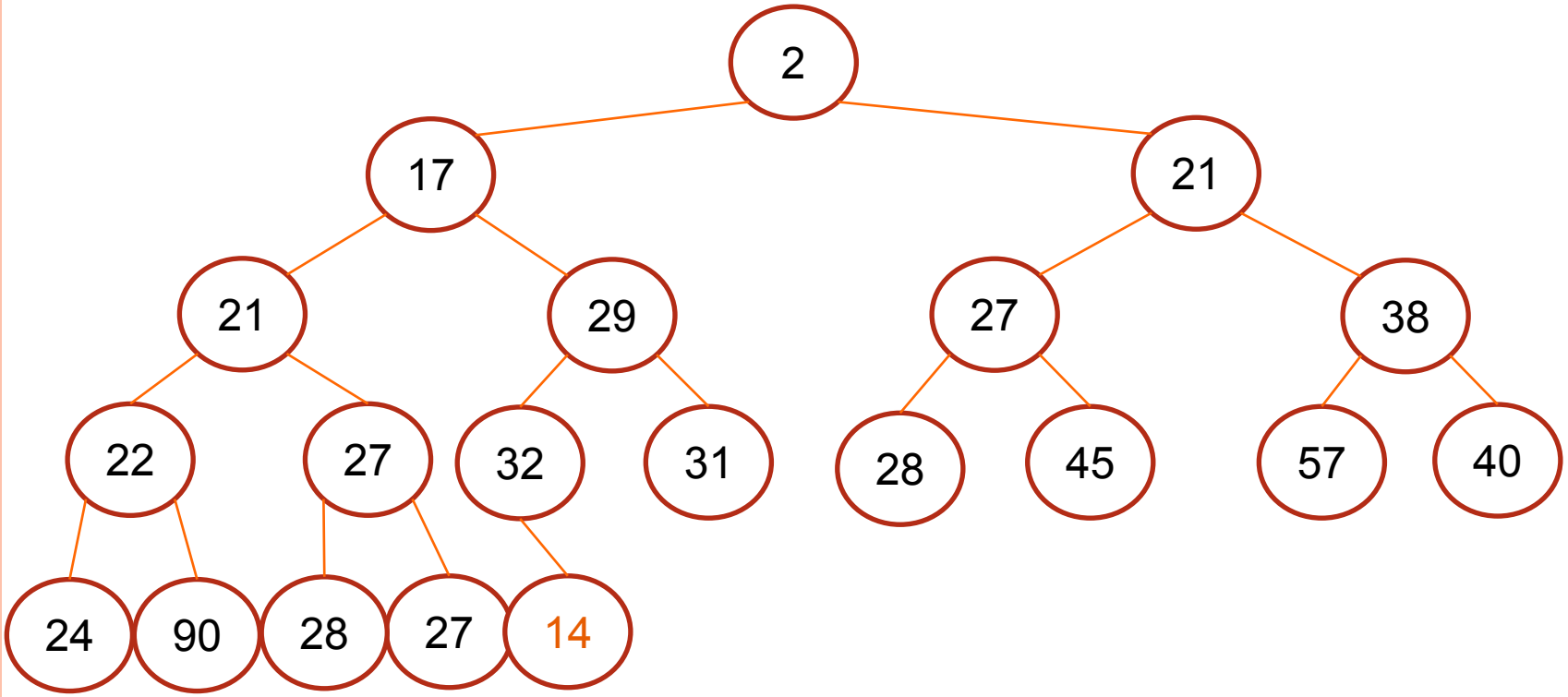
Кроме того, выполняется следующее условие:

для каждого поддерева значение, хранящееся в корне, не превосходит значений, хранящихся в его потомках

Пример двоичной кучи

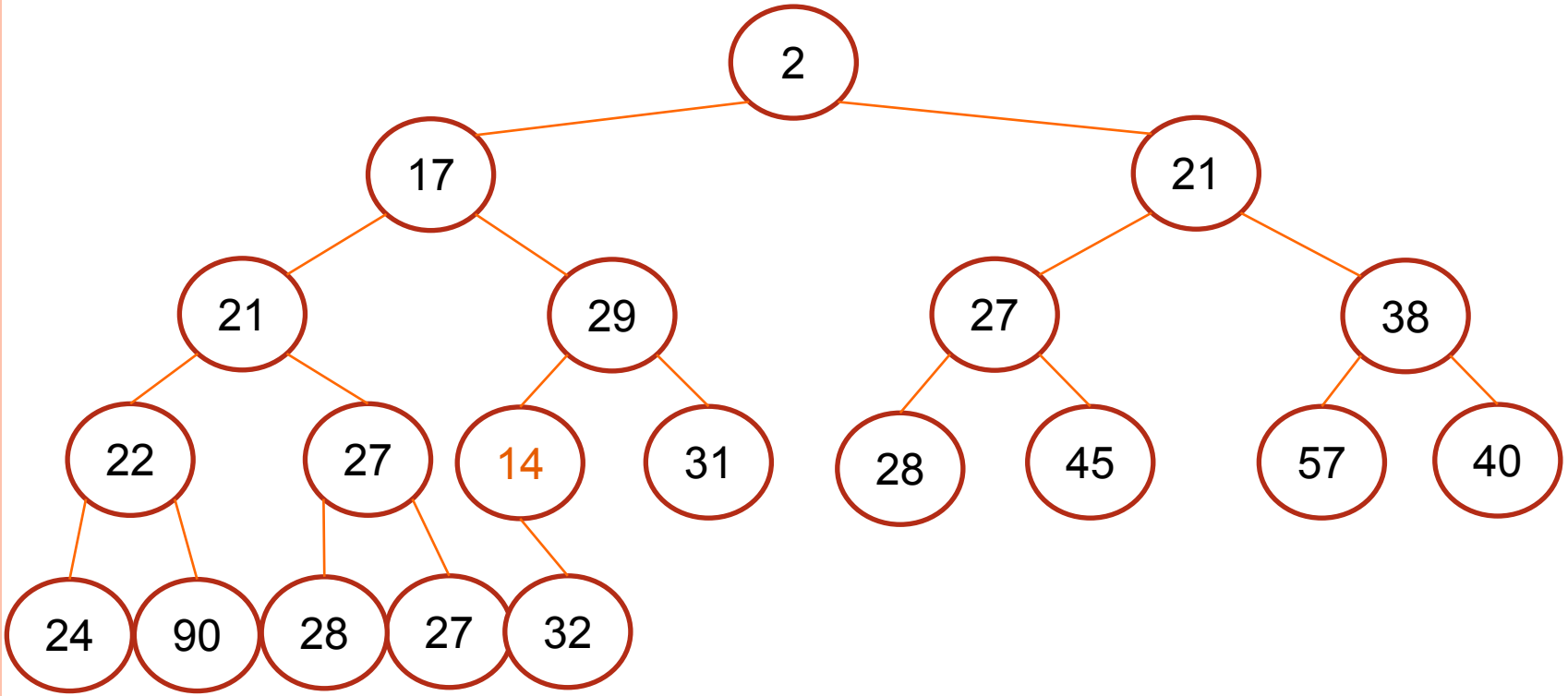


Добавление нового элемента в двоичную кучу

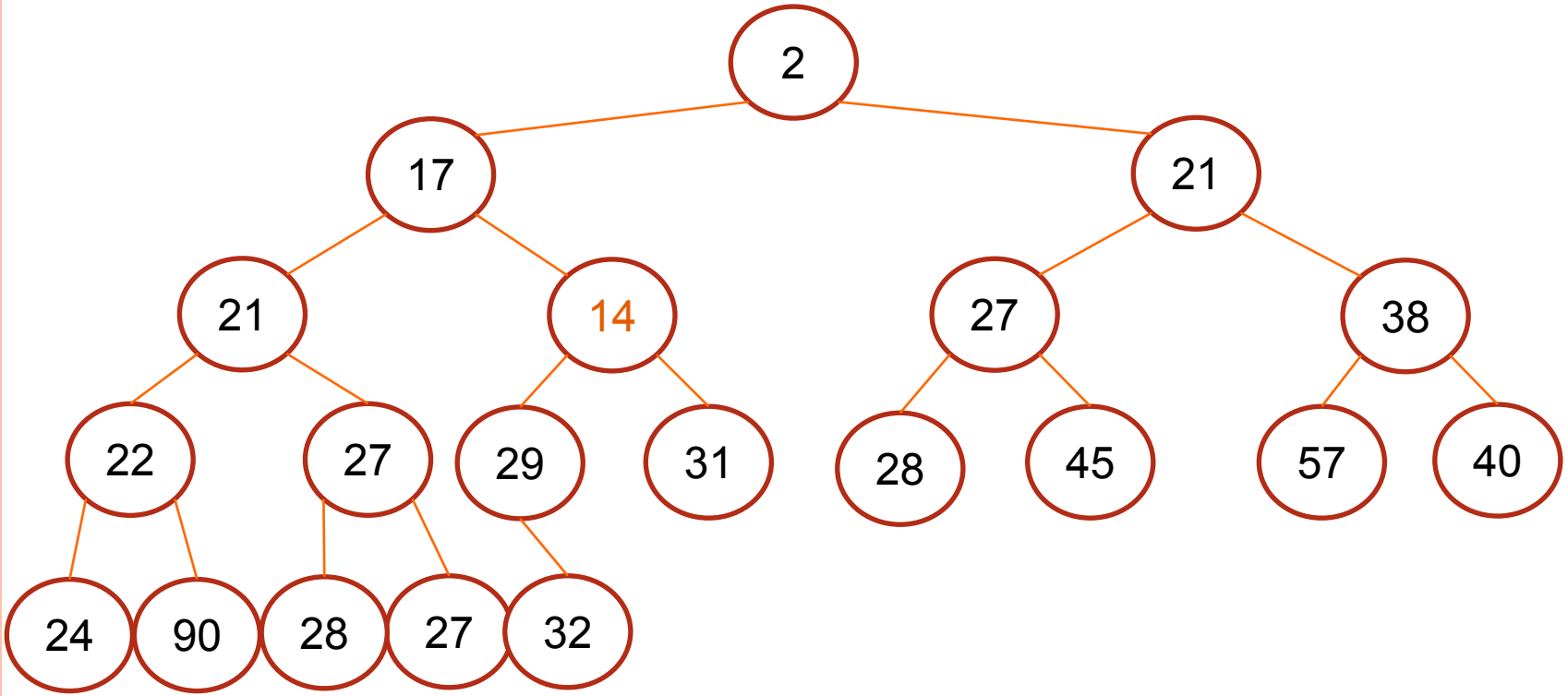


Добавляем новый элемент в конец дерева, а затем, поднимаясь вверх к корню, меняем при необходимости значения элементов.

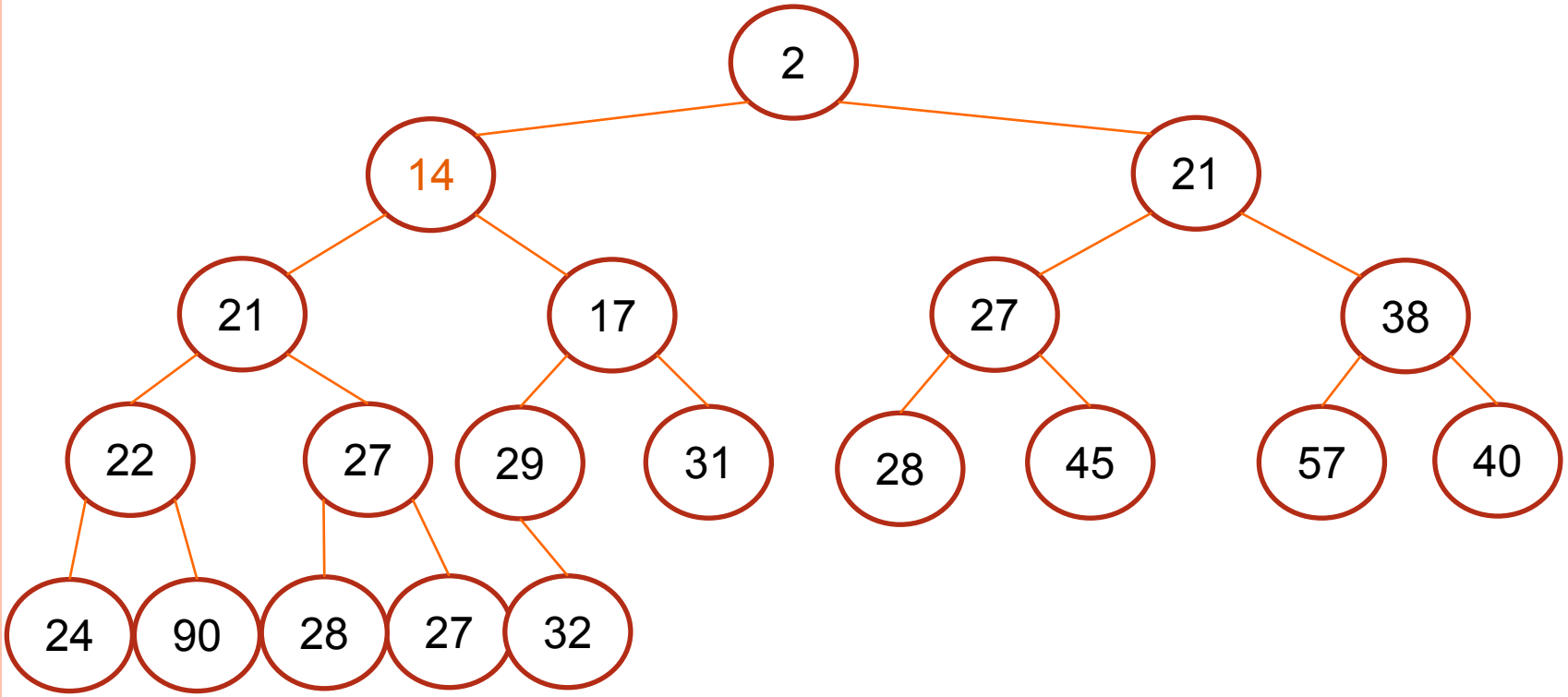
Добавление нового элемента в двоичную кучу (продолжение)



Добавление нового элемента в двоичную кучу (продолжение)

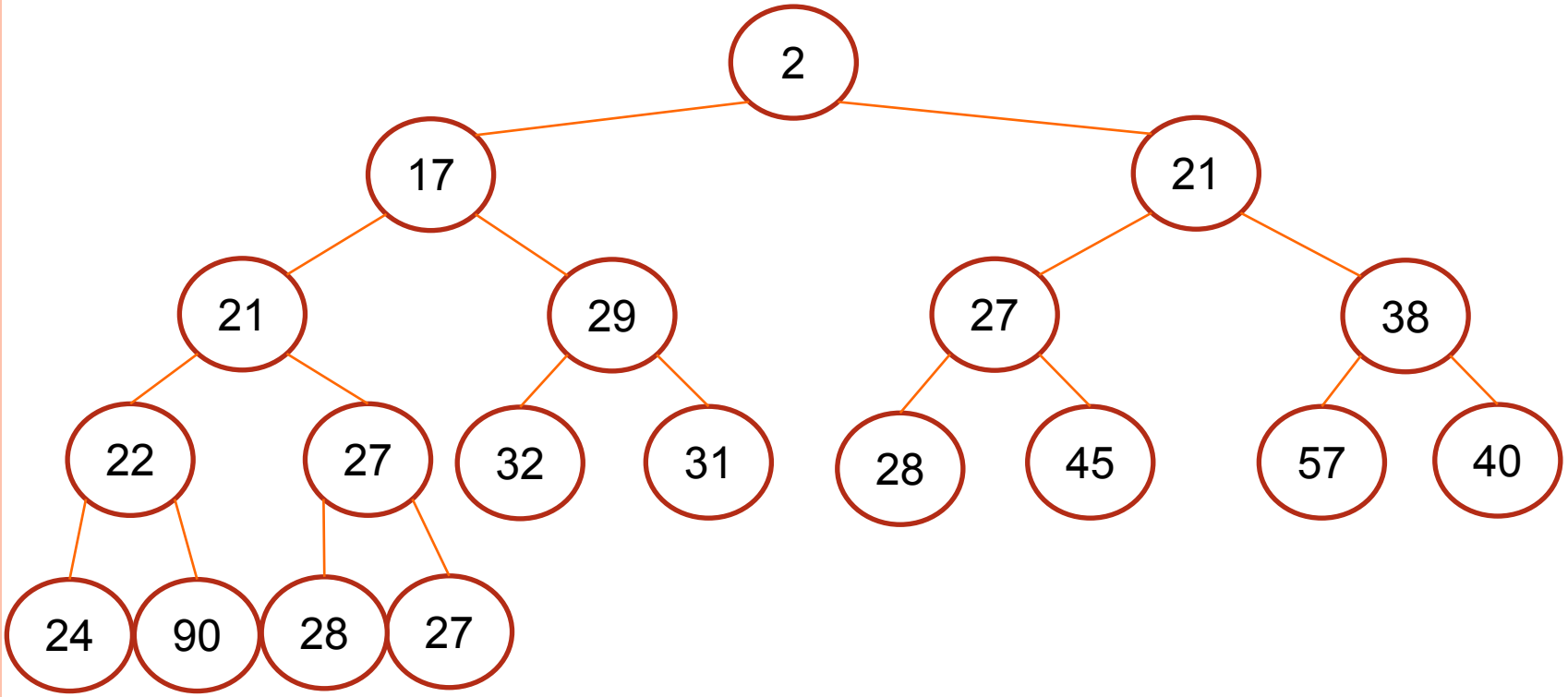


Добавление нового элемента в двоичную кучу (окончание)



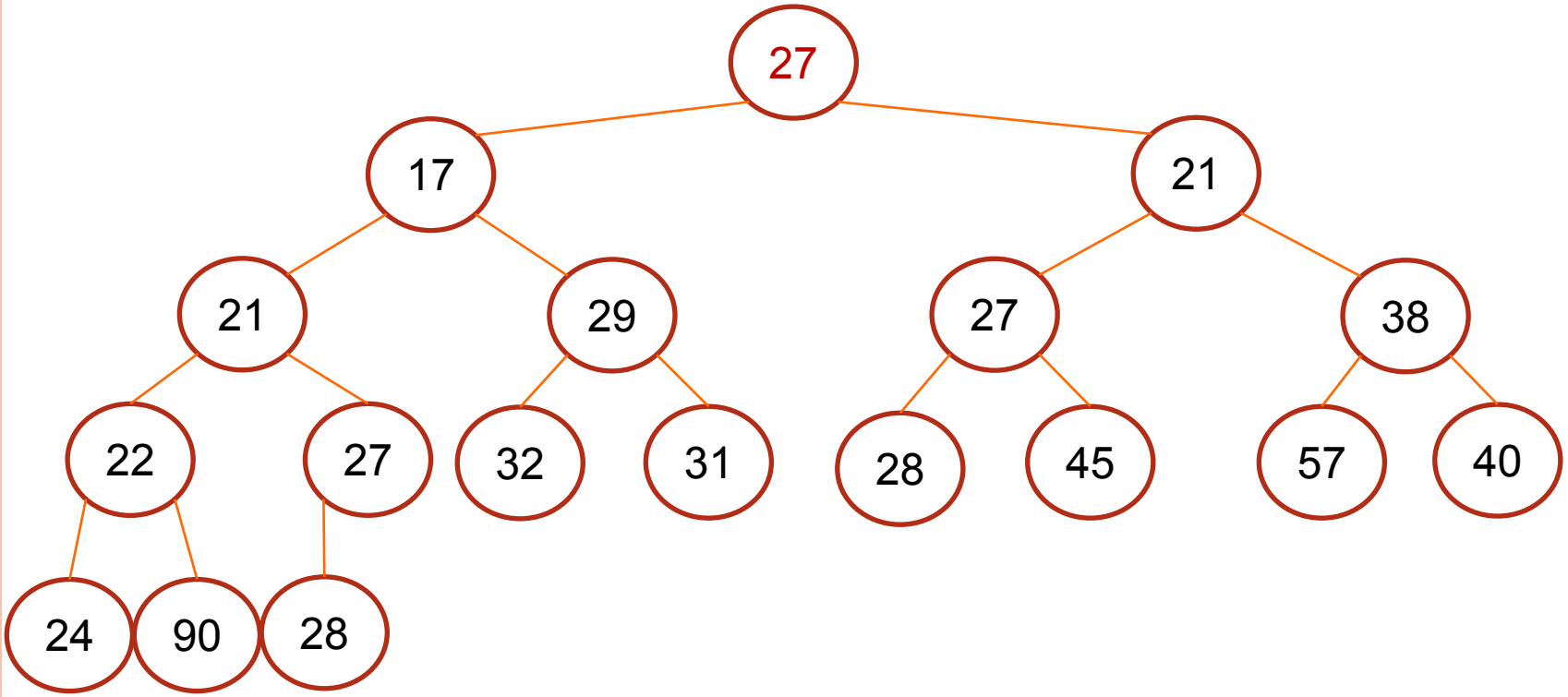
Свойства кучи восстановлены. Трудоёмкость операции добавления – O(log N), где N – объём кучи.

Удаление минимального элемента из двоичной кучи



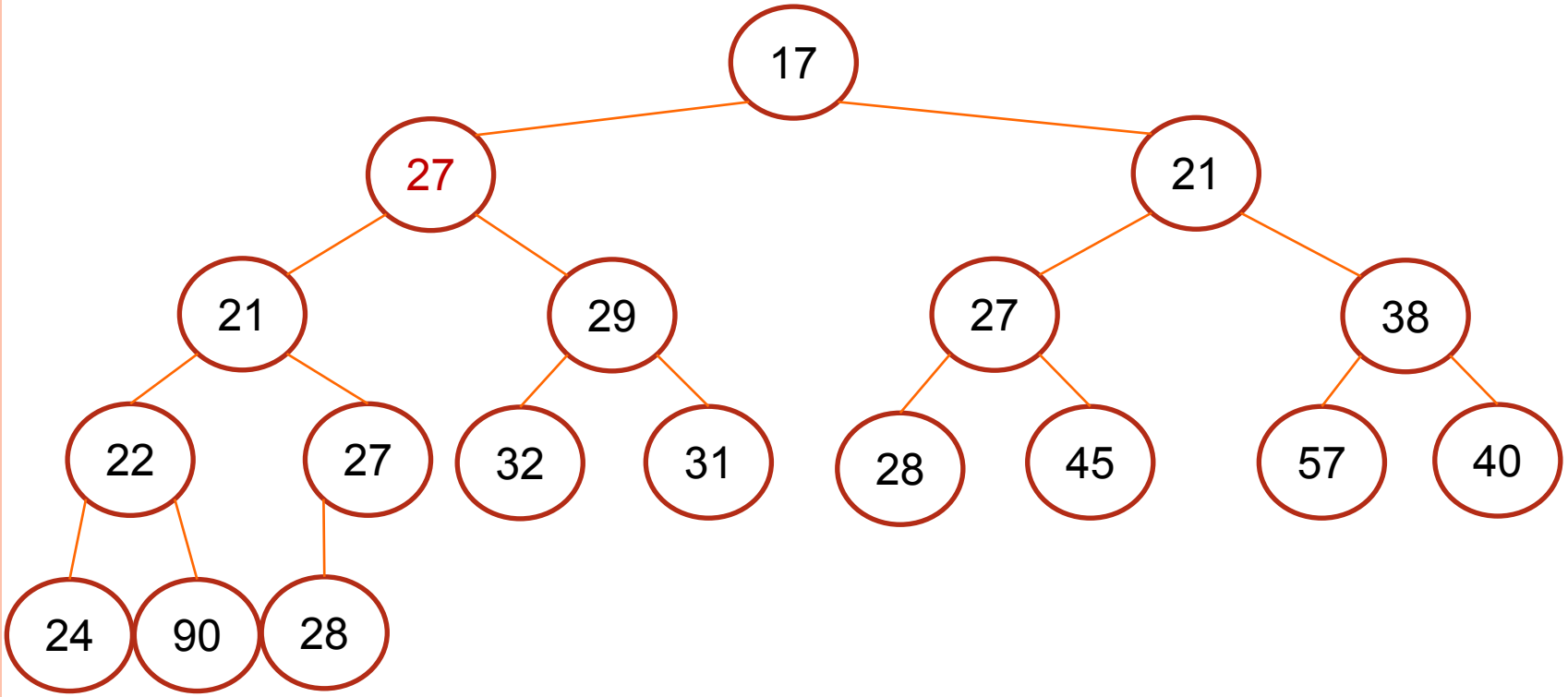
Минимальный элемент всегда находится в корне. Помещаем в корень значение из последнего элемента кучи, а сам этот элемент удаляем.

Удаление минимального элемента из двоичной кучи (продолжение)



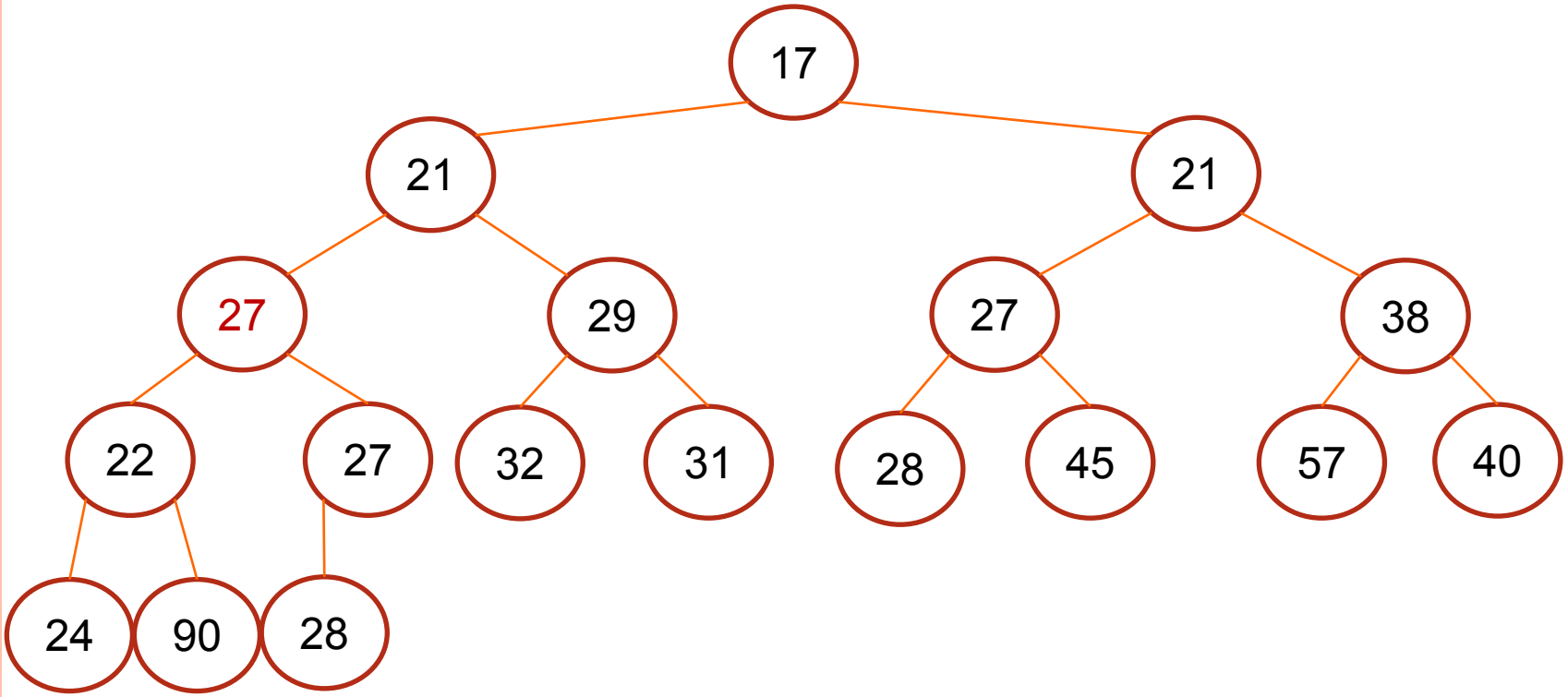
Минимальный элемент всегда находится в корне. Помещаем в корень значение из последнего элемента кучи, а сам этот элемент удаляем.

Удаление минимального элемента из двоичной кучи (продолжение)

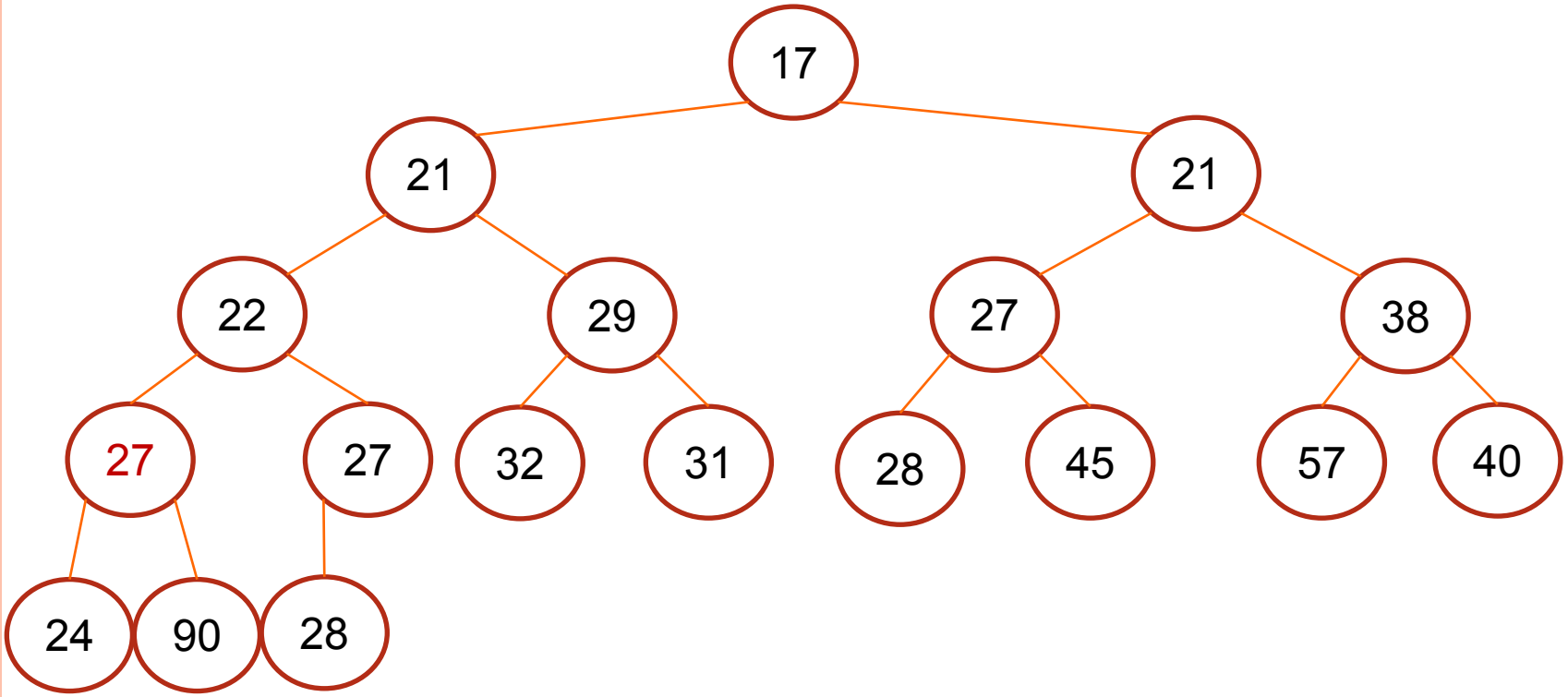


После этого, если свойства кучи нарушены, меняем значение в корне поддерева и минимальное из сыновей. Продолжаем спуск.

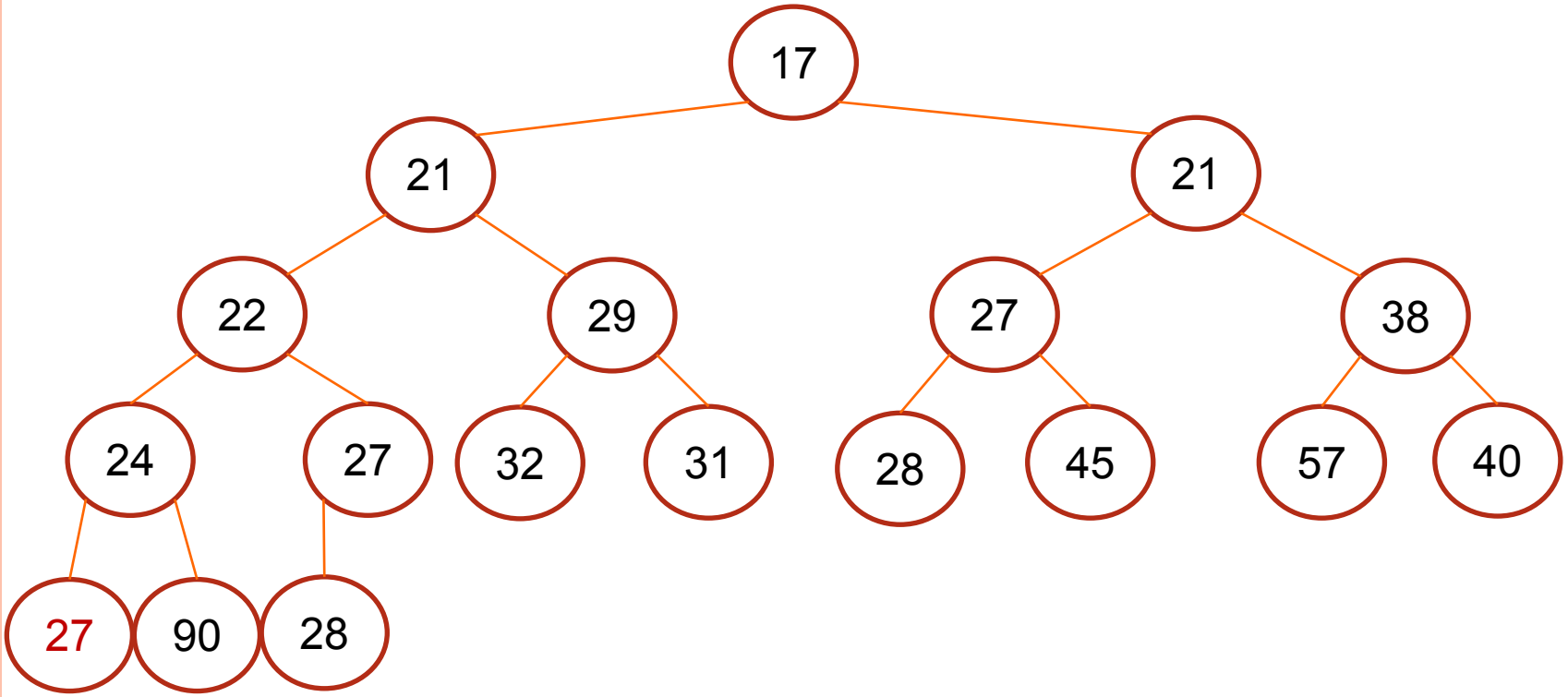
Удаление минимального элемента из двоичной кучи (продолжение)



Удаление минимального элемента из двоичной кучи (продолжение)



Удаление минимального элемента из двоичной кучи (окончание)

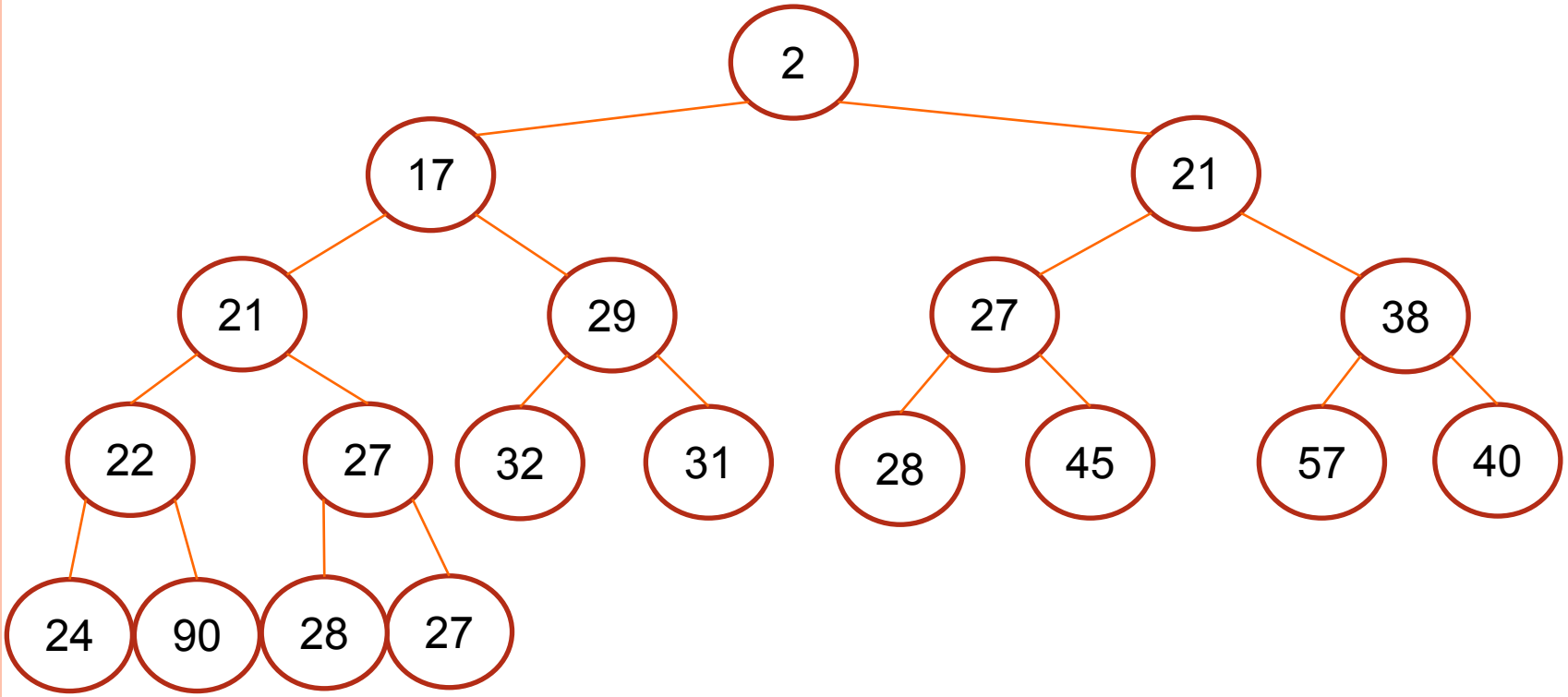


Свойства кучи восстановлены. Трудоёмкость операции удаления минимального элемента – также O(log N), где N – объём кучи.

Реализация двоичной кучи в виде массива

Создаём одномерный массив из N элементов. Значение корня помещается в массив по индексу 0, а значения сыновей вершины, хранящейся по индексу k, помещаются по индексам 2k+1 и 2k+2. Это соответствует обходу дерева по уровням.

Пример реализации двоичной кучи в виде массива



2	17	21	21	29	27	38	22	27	32	31	28

45	57	40	24	90	28	27

Использование двоичной кучи

- Сортировка с гарантированной трудоёмкостью O(N log N);
- Алгоритм Дийкстры поиска кратчайшего пути во взвешенном графе