Алгоримы и структуры данных

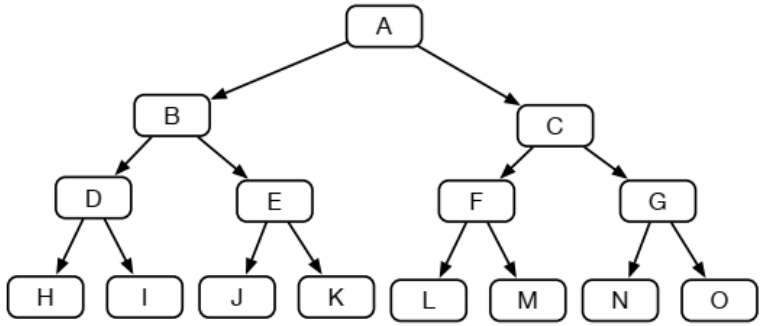
Наумов Д.А., доц. каф. КТ

Алгоритмы и структуры данных, 2021

Содержание лекции

Полное бинарное дерево

T_H высоты H есть бинарное дерево, у которого путь от корня до любой вершины содержит ровно H рёбер, при этом у всех узлов дерева, не являющихся листьями, есть и правый, и левый потомок.



Полное бинарное дерево

T_H высоты H есть бинарное дерево, у которого к корню прикреплены левое и правое полные бинарные поддеревья T_{H-1} высоты $H-1$.

Число узлов в дереве T_H есть $N = 2^{H+1} - 1$, $H = log_2(N+1)$.

Приоритетная очередь (priority queue)

очередь, элементы которой имеют приоритет, влияющий на порядок извлечения: первым извлекается наиболее приоритетный элемент.

Значение (`value`)	Приоритет (`priority`)
Москва	12000000
Казань	1500000
Урюпинск	10000
Малиновка	200

Интерфейс адстракции приоритетной очереди:

- insert – добавляет элемент в очередь;
- fetchPriorityElement – получает самый приоритетный элемент, но не извлекает его из очереди;
- extractPriorityElement – извлекает самый приоритетный элемент из очереди.

Использование упорядоченного массива:

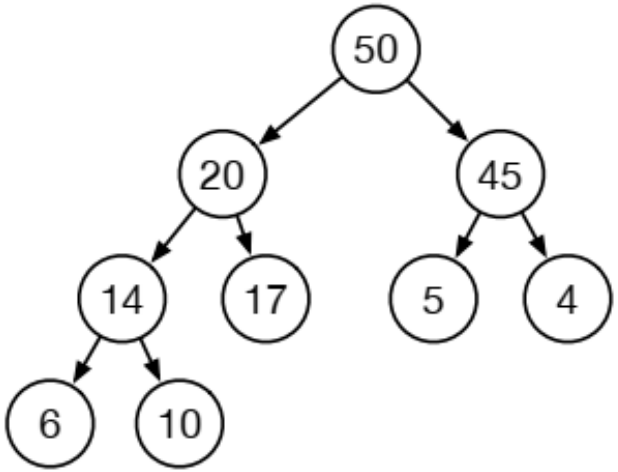
- получение самого приоритетного элемента: отсортировать данные, взять последний элемент;
- извлечение элемента: $O(1)$;
- вставка элемента: поиск места вставки $O(log(N))$, сдвиг элементов вправо $O(N)$.

Бинарная куча (пирамида, heap)

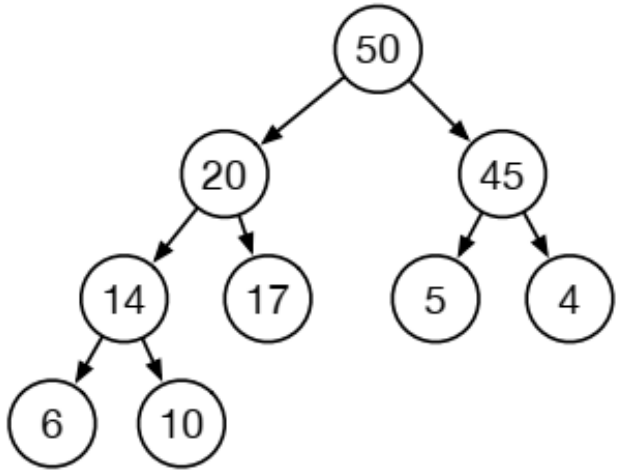
бинарное дерево, удовлетворяющее следующим условиям:

1. Приоритет любой вершины не меньше приоритета потомков.
2. Дерево является правильным подмножеством полного бинарного, допускающим плотное хранение узлов в массиве.

В невозрастающей пирамиде приоритет каждого родителя не меньше приоритета потомков.



Бинарная куча:



Хранение бинарной кучи в виде массива с индексами 1..N:

50	20	45	14	17	5	4	6	10

Удобство такого хранения трудно переоценить:

- Индекс корня всегда равен 1 – самый приоритетный элемент.
- Индекс родителя узла i всегда равен $\lfloor \frac{i}{2} \rfloor$.
- Индекс левого потомка узла i всегда равен $2i$:
- Индекс правого потомка узле i всегда равен $2i + 1$:

Структура узла бинарной кучи:

```
struct bhnode { // Узел
        string data;
        int priority;
};
```

Бинарная куча:

```
struct binary_heap {
        bhnode *body;
        int bodysize;
        //Будем фиксировать количество помещённых в кучу элеме
        int numnodes;
        binary_heap(int maxsize);
        ...
};
```

```
//Операция создания бинарной кучи определённого размера заключ
//Нулевой элемент массива мы использовать не будем.
binary_heap::binary_heap(int maxsize) {
        body = new bhnode[maxsize+1];
        bodysize = maxsize;
        numnodes = 0;
}
~binary_heap::binary_heap() {
        delete body;
}
//Ещё нам понадобится операция обмена элементов кучи по их инде
void binary_heap::swap(int a, int b) {
        std::swap(body[a],body[b]);
}
```

Сложность операции создания бинарной кучи – $T_{create} = O(N)$.

Операция поиска самого приоритетного элемента тривиальна. Её сложность – $T_{fetchMin} = O(1)$:

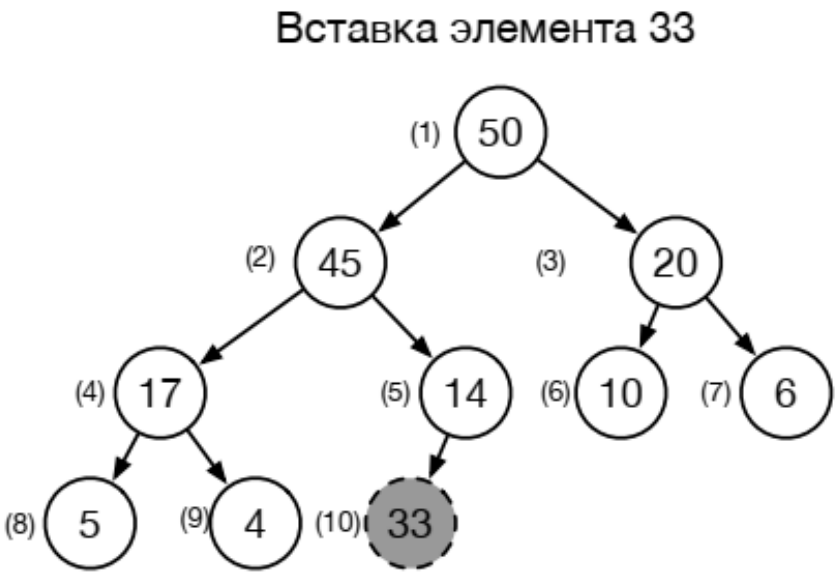
```
bhnode *binary_heap::fetchPriorityElement() {
        return numnodes == 0? nullptr : body + 1;
}
```

Правильная бинарная куча должна поддерживать два инварианта:

- структурную целостность: представимости в виде бинарного дерева
- упорядоченную целостности: то есть свойства «потомки узла не могут иметь приоритет, больший, чем у родителя».

Добавление элемента в бинарную кучу

Этап 1: вставка в конец кучи



Отлично! Структура кучи не испортилась!

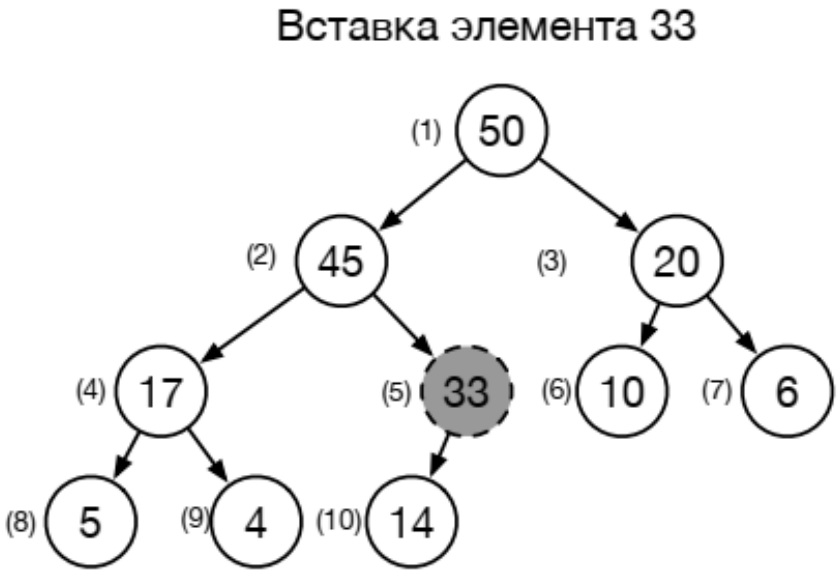
50	45	20	17	14	10	6	5	4	33

Однако пока не выдержана упорядоченность.

Добавление элемента в бинарную кучу

Этап 2: Корректировка значений

Только что вставленный элемент может оказаться более приоритетным, чем его родитель. Тогда поменяем их местами.

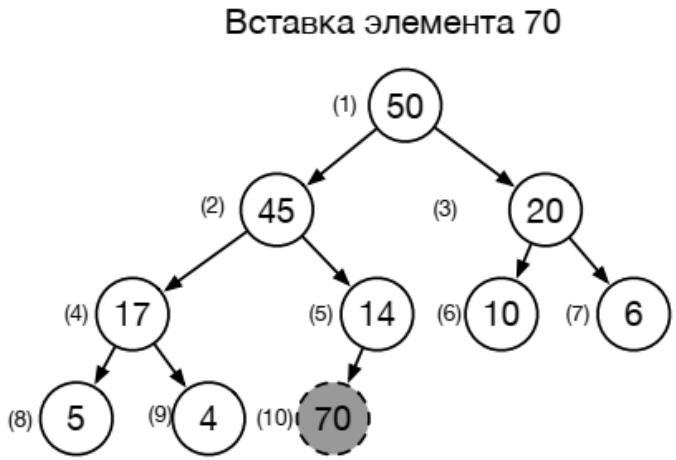


Куча удовлетворяет всем условиям.

50	45	20	17	33	10	6	5	4	14

Добавление элемента в бинарную кучу

Попытаемся вставить элемент, который имеет приоритет больше, чем все элементы в куче.

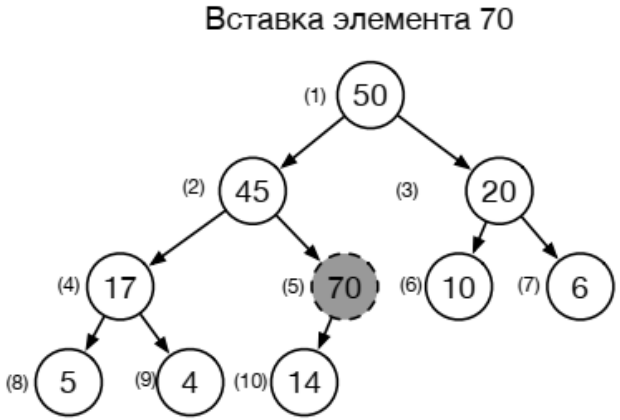


Он находится не на своём месте...

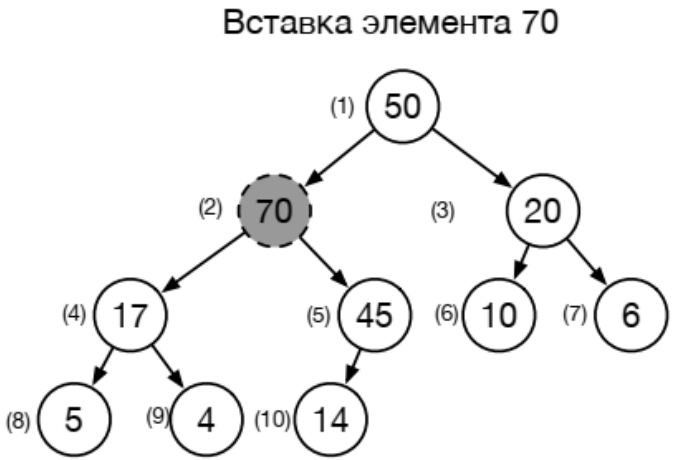
50	45	20	17	14	10	6	5	4	70

Добавление элемента в бинарную кучу

... и меняется местами с родителем (ползёт вверх по дереву).

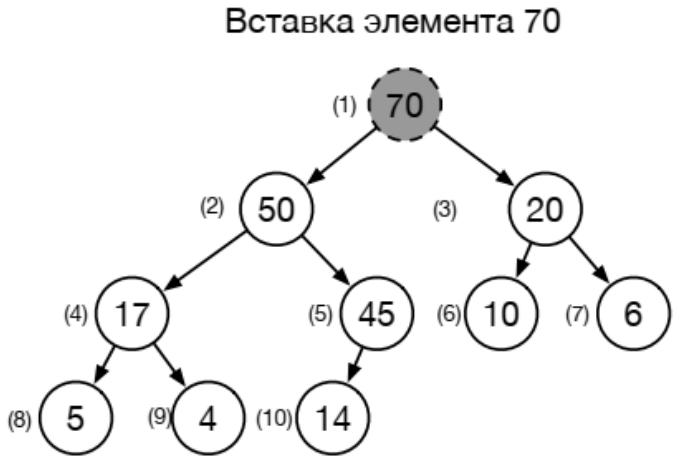


Опять он не на своём месте — и опять меняется местами с родителем.



Добавление элемента в бинарную кучу

Максимальный элемент переместился в корень.



Алгоритм завершён

70	50	20	17	45	10	6	5	4	14

Корректность алгоритма базируется на двух фактах:

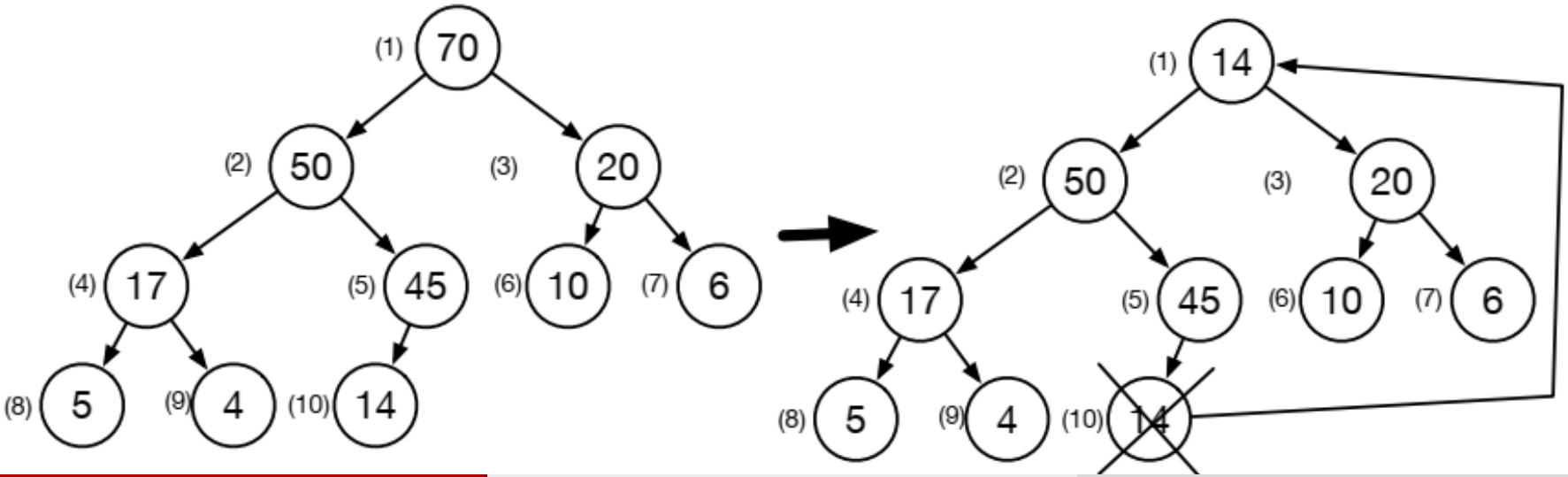
- Инвариант структурной целостности не нарушен ни в один момент времени.
- После каждого шага перемещения вставленного элемента поддерево с корнем в текущем элементе поддерживает инвариант упорядоченной целостности.

Сложность алгоритма определяется высотой дерева и составляет $T_{Insert} = O(log(N))$.

```
int binary_heap::insert(bhnode node) {
        if (numnodes > bodysize) {
                return -1; // или расширяем.
        }
        body[++numnodes] = node;
        for (int i = numnodes; i > 1 &&
                body[i].priority > body[i/2].priority; i /= 2)
                swap(i, i/2);
        }
}
```

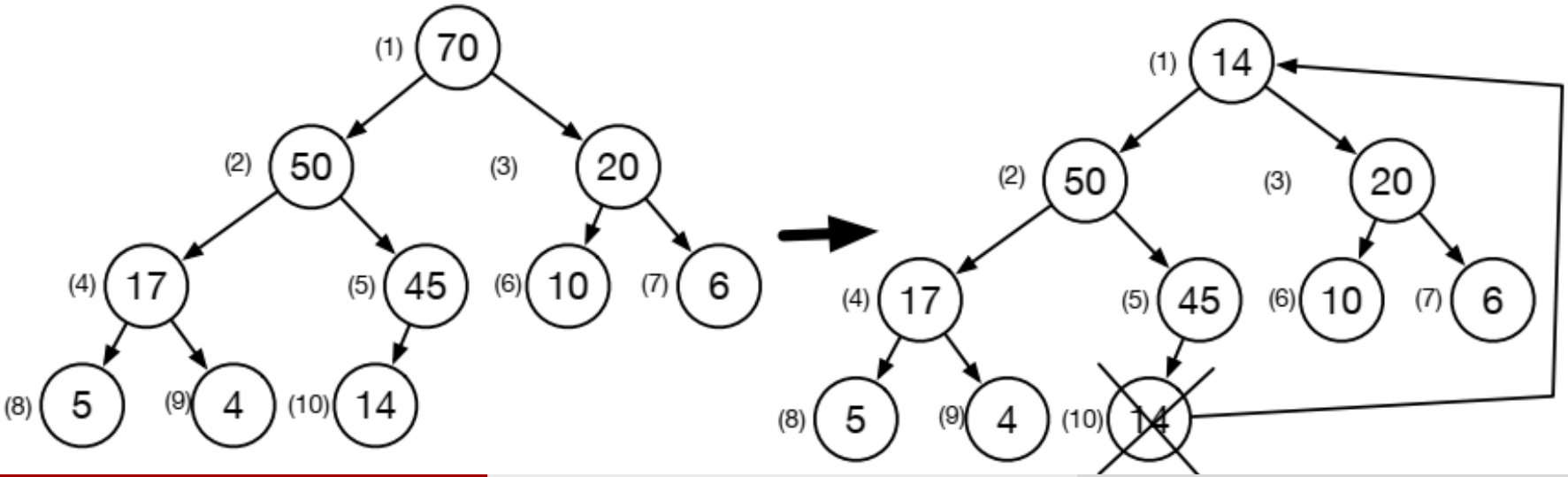
Операция удаления самого приоритетного элемента

- Операция удаления самого приоритетного элемента кажется более сложной – ведь после удаления корневого элемента нарушается структурная целостность.
- Первый шаг при удалении корневого элемента должен сохранять структурную целостность.
- Так как после удаления корня количество элементов уменьшится на один, то отправляем самый последний элемент кучи в корень, уменьшая при этом numnodes.



Операция удаления самого приоритетного элемента

- Операция удаления самого приоритетного элемента кажется более сложной – ведь после удаления корневого элемента нарушается структурная целостность.
- Первый шаг при удалении корневого элемента должен сохранять структурную целостность.
- Так как после удаления корня количество элементов уменьшится на один, то отправляем самый последний элемент кучи в корень, уменьшая при этом numnodes.

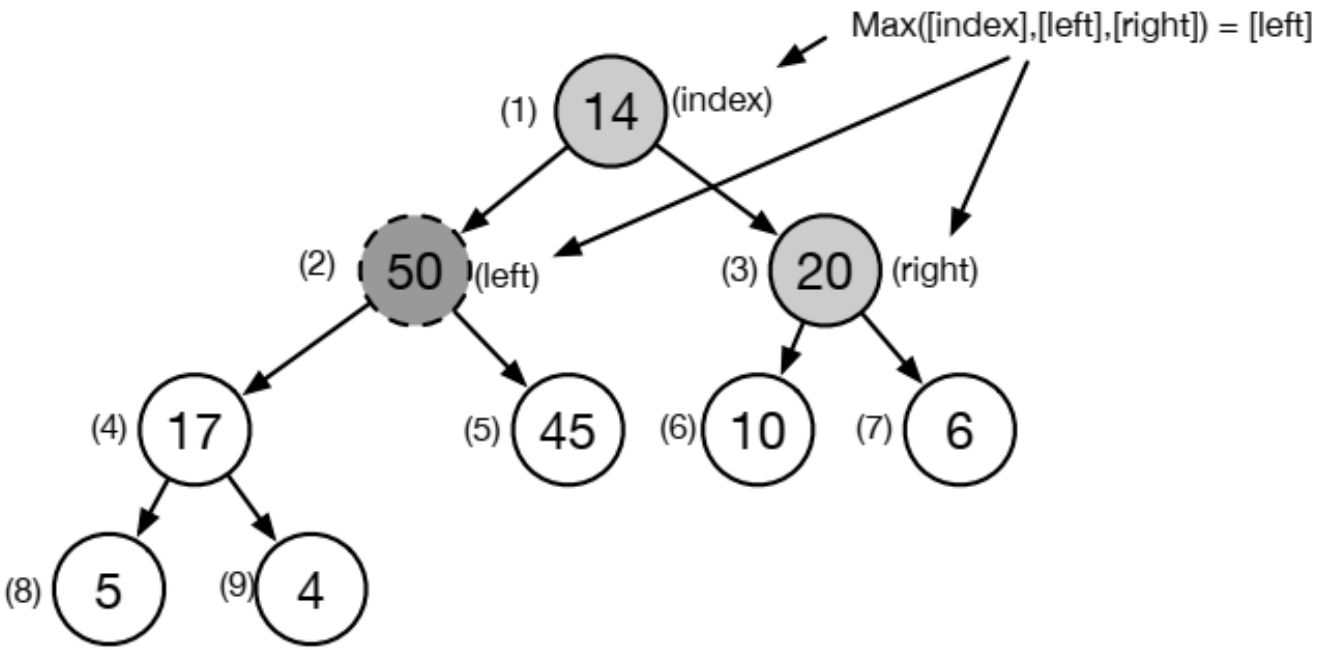


Структурная целостность не изменилась, но могла измениться упорядоченная целостность. Требуется восстановление, функция heapify.

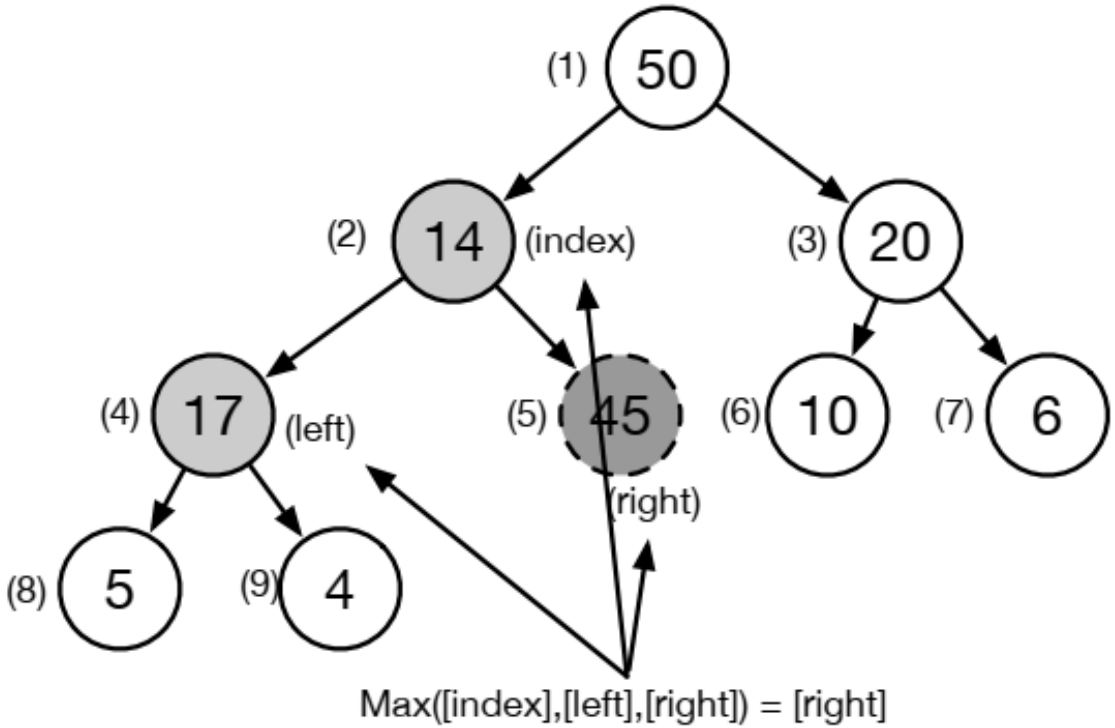
- Идея функции проста: начиная с корневого элемента, мы проводим соревнование между тремя кандидатами – теми, кто может занять это место.
- Если кандидаты (левый и правый потомки) менее приоритетны, чем текущий претендент, то алгоритм завершён.
- Иначе претендент меняется местами с самым приоритетным из потомков – и операция повторяется уже на уровне ниже. Так как каждая операция обмена передвигает претендента на один уровень вниз, процесс обязательно завершается не более, чем за $O(logN)$ шагов, что и составляет сложность данного алгоритма.

Проиллюстрируем алгоритм на примере.

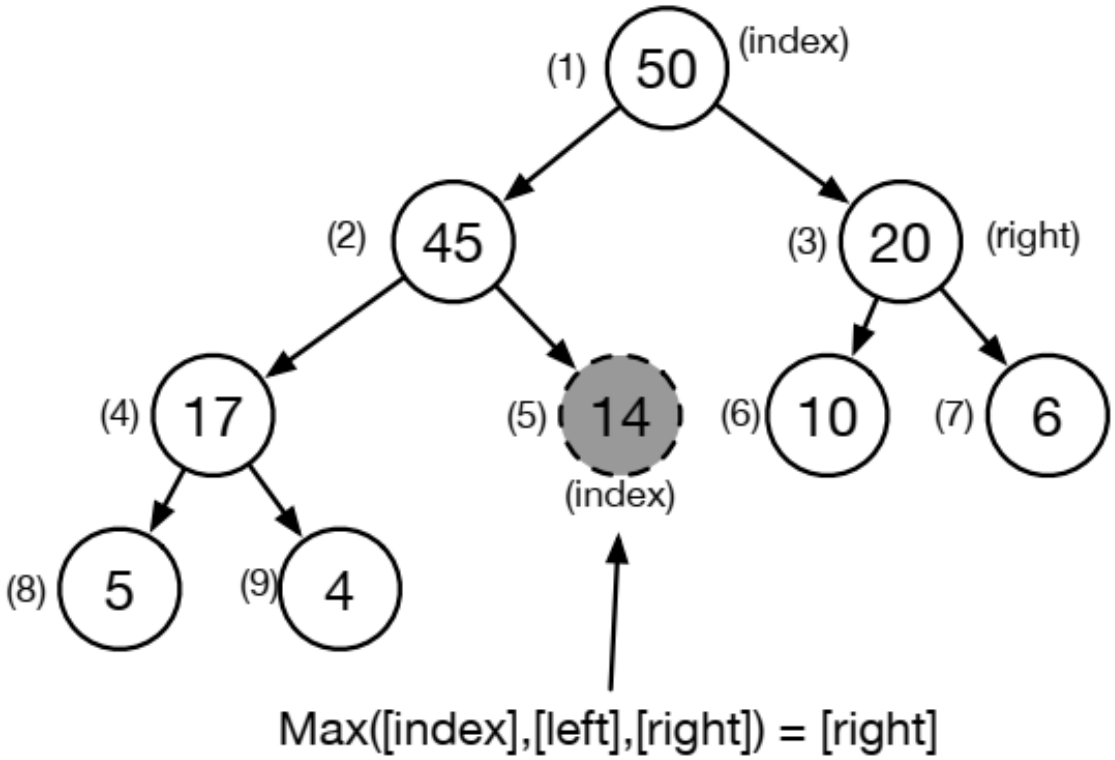
- После перемещения последнего узла (14) в корень он становится претендентом, а кандидатами оказываются узлы (50) и (20).
- Индекс восстановления (номер претендента) пока равен 1.



- Претендента обменяли на кандидата (50) с индексом 2.
- Теперь элемент под этим индексом – новый претендент.



- Теперь претендентом становится элемент с индексом 5.
- После завершения этой операции восстановление завершено.



```
void binary_heap::heapify(int index) {
  for (;;) {
    int left = index + index;
    int right = left + 1;
    // Кто больше, [index], [left], [right]?
    int largest = index;
    if (left <= numnodes &&
        body[left].priority > body[index].priority)
      largest = left;
    if (right <= numnodes &&
        body[right].priority > body[largest].priority)
      largest = right;
    if (largest == index) break;
    swap(index, largest);
    index = largest;
  }
}
```

Алгоритм HeapSort

Возможность получать из бинарной кучи самый приоритетный элемент за $O(logN)$ и добавлять элементы в бинарную кучу за $O(logN)$ вызывает желание реализовать ещё один алгоритм сортировки. Что интересно, этот алгоритм будет иметь сложность $O(NlogN)$ в худшем случае.

1. Создать бинарную кучу размером N. Это потребует сложности $O(N)$.
2. Поочерёдно вставить в неё все N элементов массива. Сложность этого этапа есть $O(log1) + O(log2) + ... + O(logN)) < O(logN) + ... + O(logN)) = Nlog(N)$.
3. Поочерёдно извлекать с удалением самый приоритетный элемент из бинарной кучи - -с помещением в последовательные N позиций исходного массива.

Такая прямолинейная организация не особенно хороша: потребуется добавочная память на бинарную кучу размером N элементов.

Модифицируем функцию heapify для того, чтобы она могла работать с произвольным массивом, адресуемым с нуля:

```
void heapify(int *a, int i, int n){
  int curr = a[i];
  int index = i;
  for (;;) {
    int left = index + index + 1;
    int right = left + 1;
    if ( left < n && a[left] > curr)
      index = left;
    if ( right < n && a[right] > a[index])
      index = right;
    if (index == i ) break;
    a[i] = a[index];
    a[index] = curr;
    i = index;
  }
}
```

Алгоритм HeapSort

- Теперь сортировка заключается в том, что мы создаём бинарную кучу размером n на месте исходного массива, переставляя его элементы.
- Затем на шаге i мы обмениваем самый приоритетный элемент кучи (он всегда располагается на позиции 0) с элементом под номером $n - i - 1$.
- Размер кучи при этом уменьшается на единицу, а самый приоритетный элемент занимает теперь положенное ему по рангу место.

```
void sort_heap(int *a, int n) {
  for(int i = n/2-1; i >= 0; i--) {
    heapify(a, i, n);
  }
  while( n > 1 ) {
    n--;
    swap(a[0],a[n]);
    heapify(a, 0, n);
  }
}
```

Вопрос: если эта сортировка гарантирует нам сложность $O(Nlog(N))$ даже в самом худшем случае, а быстрая сортировка, QuickSort, не гарантирует, то почему не использовать только эту сортировку?

Алгоритм HeapSort

1. Первая причина в том, что в быстрой сортировке используется меньшее количество операций обмена с памятью, а излишней работы с памятью на современных компьютерах стоит избегать.
2. Вторая причина тоже связана с памятью, точнее – с тем фактом, что N обращений к последовательным ячейкам памяти исполняется до 10-15 раз быстрее, чем столько же обращений к случайным ячейкам памяти. Это связано с наличием ограниченного количества аппаратной кэш-памяти на современных процессорах.

При наличии различных алгоритмов, исполняющих одну и ту же задачу, некоторые из них могут быть дружелюбны к кэшу (cache-friendly), а некоторые – нет. Поэтому наилучшие по времени исполнения алгоритмы могут быть разными в различное время и на различных вычислительных системах.