

Двоичные деревья поиска. Очередь с приоритетами

Федор Царев
Спецкурс «Олимпиадное
программирование»
Лекция 3

19.01.2009, 26.01.2009
Санкт-Петербург, Гимназия 261





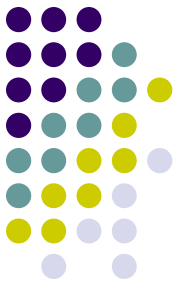
Цель лекции

- Изучить теоретические основы двоичных деревьев поиска и очереди с приоритетами и алгоритмы их обработки
- Изучить методы реализации этих структур данных и алгоритмов на языке программирования Pascal (Delphi)



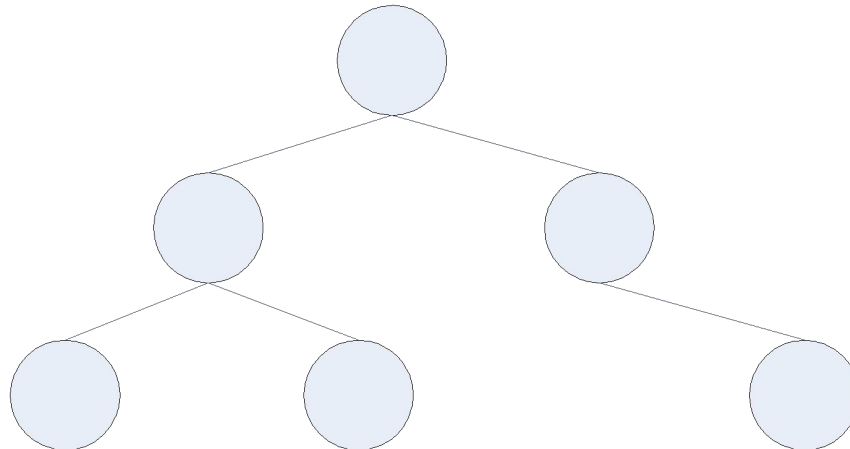
Постановка задачи

- Реализовать структуру данных, хранящую набор чисел и поддерживающую операции:
 - добавить число
 - найти число
 - удалить число
 - найти минимум
 - найти максимум
 - найти k -ое по порядку число (k -ую порядковую статистику)



Двоичное дерево

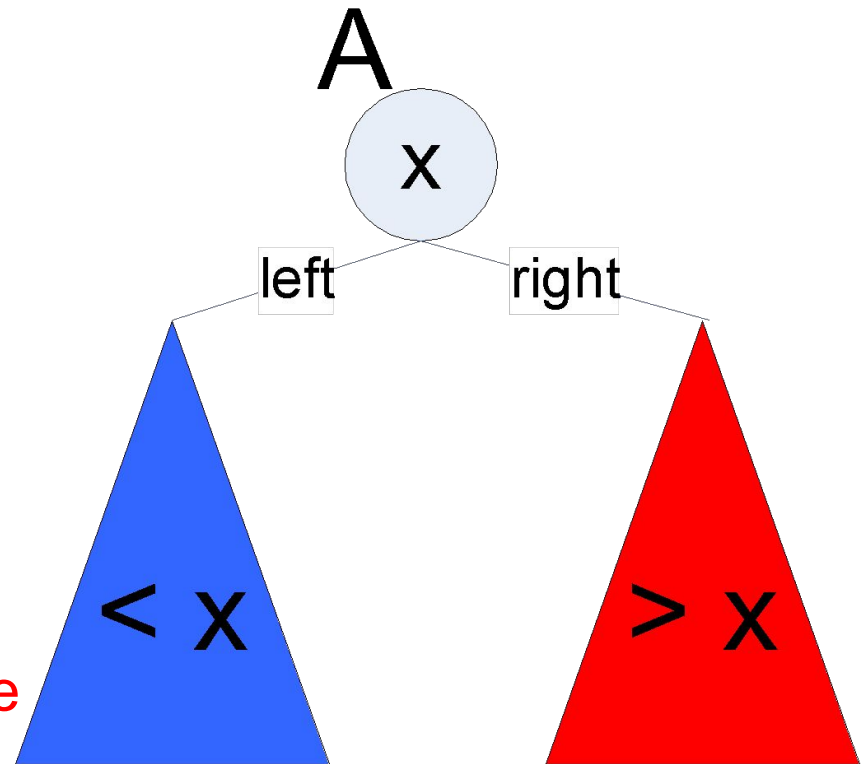
- Дерево, каждая вершина которого имеет не **более двух детей**
- На детях каждой вершины задан **порядок** – есть «левый» ребенок и «правый» ребенок
- Самая верхняя вершина – **корень** – не имеет родителя
- **Глубина вершины** – расстояние от нее до корня
- **Высота дерева** – наибольшая глубина его вершины



Двоичное дерево поиска



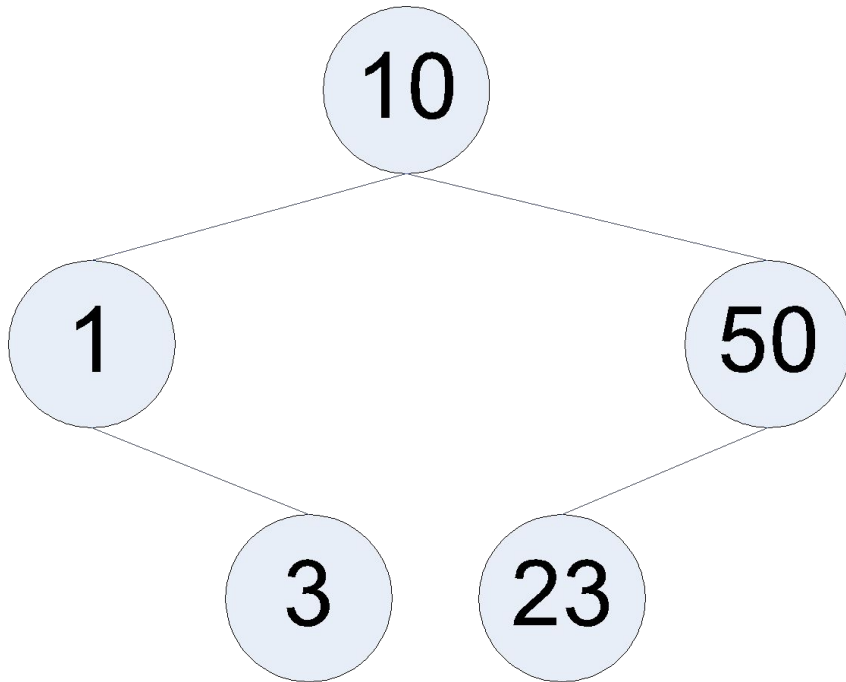
- Это двоичное дерево, в каждой вершине которого хранится число
- При этом если в некоторой вершине A хранится число x , то:
 - все числа в левом поддереве A меньше чем x
 - все числа в правом поддереве A больше чем x



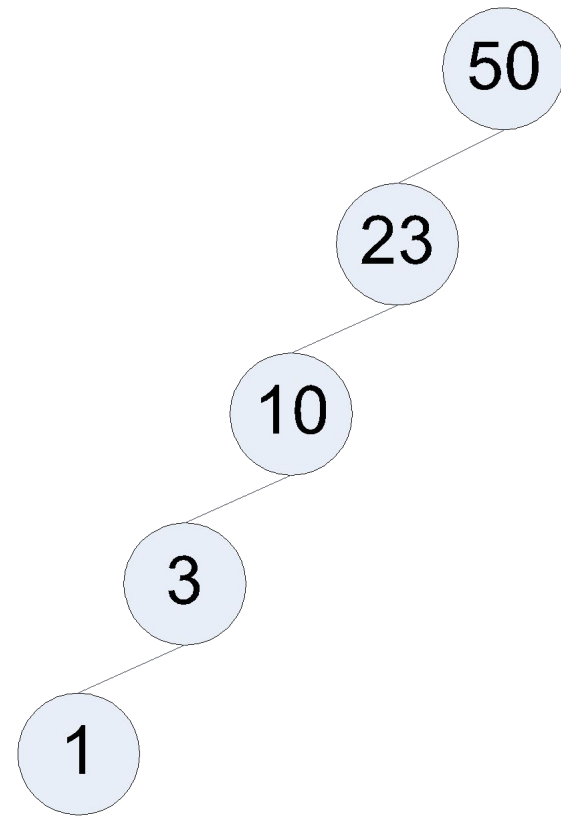
Дерево поиска неоднозначно



- Набор чисел: 1, 3, 10, 23, 50



Высота есть $O(\log n)$

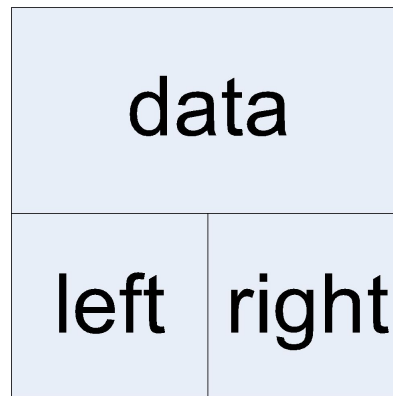


Высота есть $O(n)$

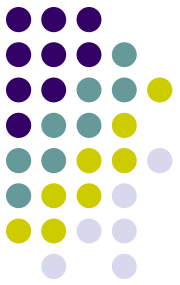


Вершина дерева

```
node = record
  data : integer;
  left, right : integer; // при реализации
                        // на собственном
                        // менеджере памяти
end;
```



-1 соответствует тому, что соответствующего ребенка у вершины нет

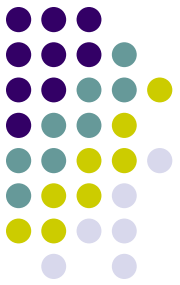


Поиск элемента в дереве

- Если значение y в текущей вершине равно искомому x , то поиск завершен
- Если $x > y$, то перейти к правому ребенку
- Если $x < y$, то перейти к левому ребенку

Программа

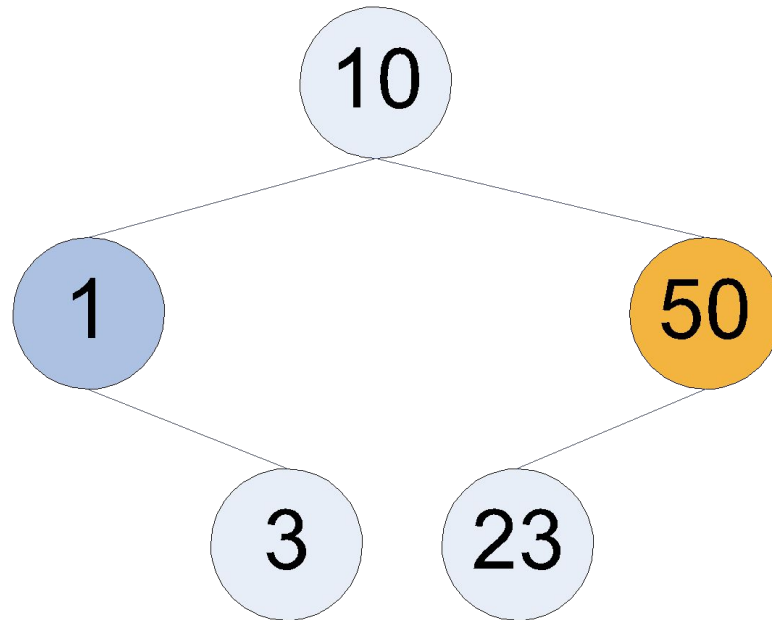
```
function find(x : integer) : boolean;
begin
  cur := root; // root - корень дерева
  result := false;
  while (cur <> -1) do begin
    if (memory[cur].data = x) then begin
      result := true;
      exit;
    end;
    if (x > memory[cur].data) then begin
      cur := memory[cur].right;
    end else begin
      cur := memory[cur].left;
    end;
  end;
end;
end;
```



Поиск минимума и максимума



- Минимум – самый левый элемент в дереве
- Максимум – самый правый элемент в дереве



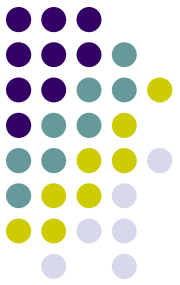
Добавить число



- Необходимо:
 1. Найти место, в котором должно находиться число
 2. Создать новый узел дерева и поместить в него это число
- Считаем, что все числа в дереве различны

Программа (1)

```
function add(root, x : integer) : integer;
begin
  if (root = -1) then begin
    cur := free;
    inc(free);
    memory[cur] := data;
    result := cur;
    exit;
  end;
  if (memory[root].data = x) then begin
    // Элемент уже есть в дереве
    result := root;
    exit;
  end;
```



Программа (2)



```
if (x < memory[root].data) then begin
  cur := add(memory[root].left, x);
  memory[root].left := cur;
  result := root;
  exit;
end;
if (x > memory[root].data) then begin
  cur := add(memory[root].right, x);
  memory[root].right := cur;
  result := root;
  exit;
end;
end;
```



Вызов процедуры

```
root := add(root, x);
```

Здесь `root` – это корень дерева.

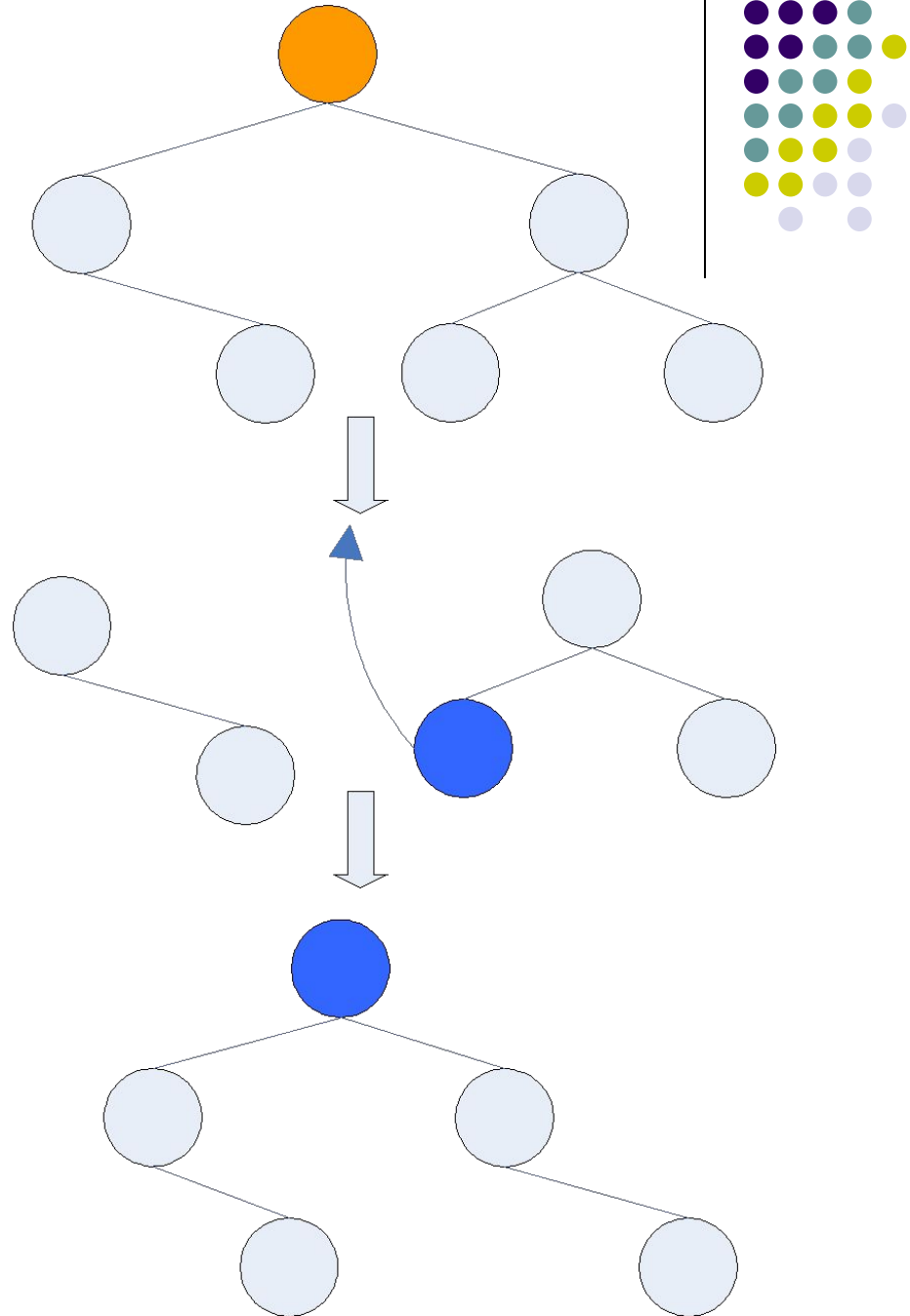


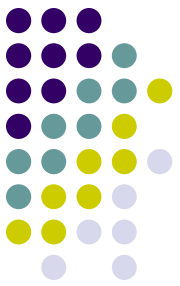
Удаление элемента

- Необходимо найти элемент
- Три случая:
 1. вершина является листом – просто удаляем
 2. у вершины один ребенок – заменяем на ребенка
 3. у вершины два ребенка

Третий случай

- Необходимо найти минимальную вершину в правом поддереве и переместить ее на место удаляемой
- Проблем с перемещением не возникнет, так как у **перемещаемой** вершины не более одного ребенка





Время работы операций

- Время работы всех рассмотренных операций пропорционально высоте дерева – **в худшем случае есть $O(n)$**
- Существуют методы выполнения этих операций так, чтобы высота все время была $O(\log n)$:
 - AVL-деревья
 - красно-черные деревья
 - ...
 - Они будут рассмотрены в следующих лекциях



Упражнение

- Предложить алгоритм нахождения следующего и предыдущего элемента в двоичном дереве поиска со временем работы пропорциональным высоте дерева

Поиск k-ой порядковой статистики



- В каждом узле дерева необходимо хранить дополнительную информацию – число узлов в его поддереве

```
node = record
```

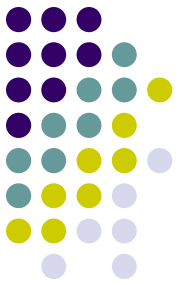
```
  data : integer;
```

```
  left, right : integer;
```

```
  size : integer;
```

```
end;
```

- $\text{memory}[a].\text{size} = \text{memory}[\text{memory}[a].\text{left}].\text{size} + \text{memory}[\text{memory}[a].\text{right}].\text{size} + 1$



Поиск k -ого элемента

- Дано:
 - Корень дерева
 - Число k
- Если в левом поддереве не меньше, чем k элементов, то переходим к нему
- Если в левом поддереве ровно $k-1$ элемент, то k -ый элемент находится в корне
- Иначе переходим к правому поддереву

Программа



```
function getKth(root, k : integer) : integer;
begin
  sizeLeft := 0;
  if (memory[root].left <> -1) then begin
    sizeLeft := memory[memory[root].left].size;
  end;
  if (k <= sizeLeft) then begin
    result := getKth(memory[root].left, k);
    exit;
  end;
  if (sizeLeft = k - 1) then begin
    result := memory[root].data;
    exit;
  end;
  result := getKth(memory[root].right, k-sizeLeft-1);
end;
```



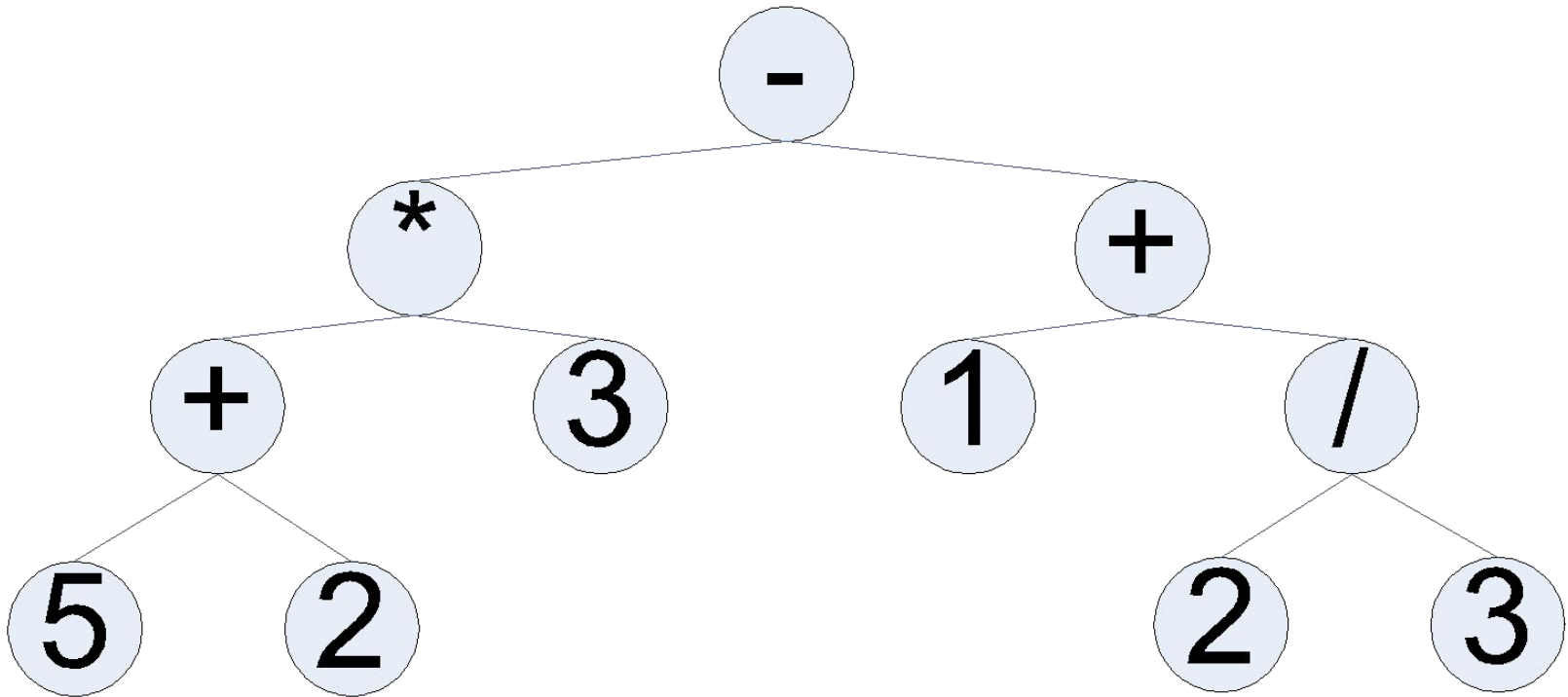
Упражнение

- Модифицировать операции добавления и удаления вершины, так чтобы они обновляли значения поля «размер поддеревя». Время работы операций не должно измениться.



Деревья выражений

- $(5+2)*3 - (1 + 2/3)$

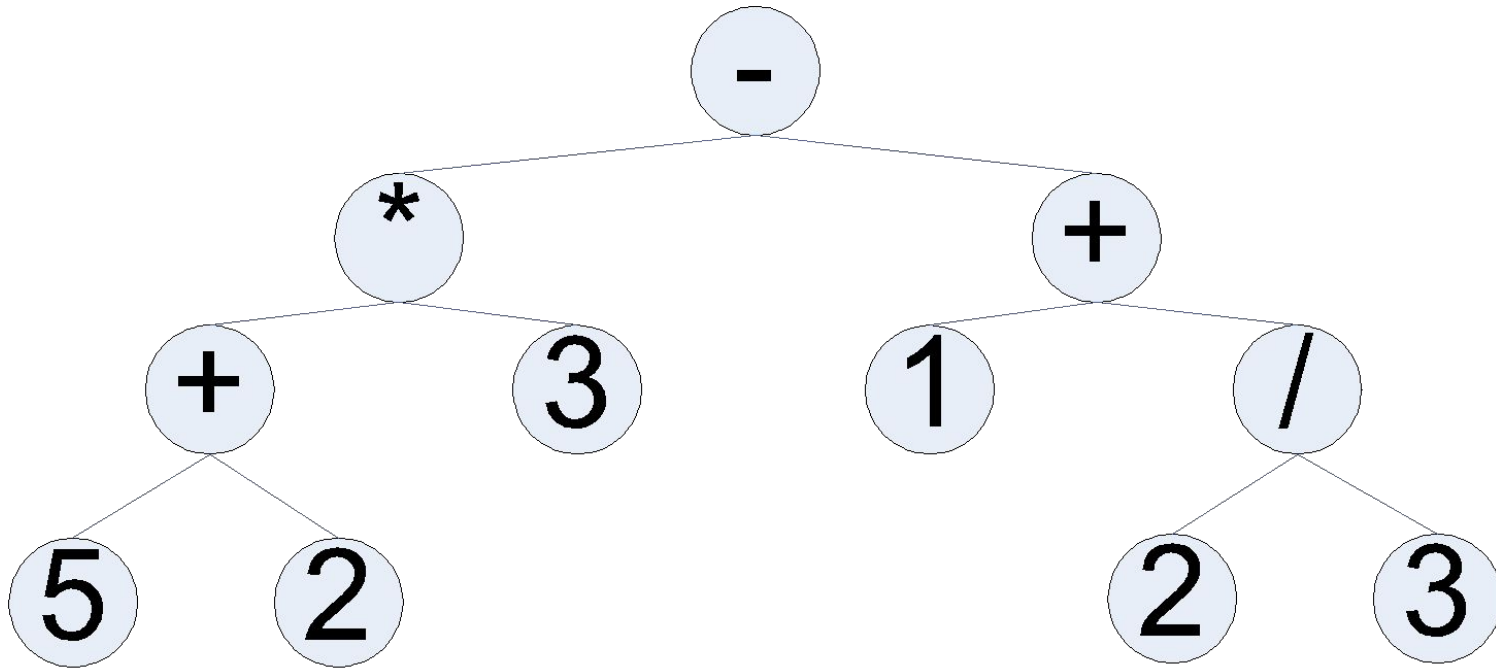


Обходы двоичных деревьев



- Основные варианты обхода двоичного дерева:
 - корень – левое поддерево – правое поддерево
 - левое поддерево – корень – правое поддерево
 - левое поддерево – правое поддерево – корень
- Им соответствуют:
 - префиксная запись
 - инфиксная запись
 - постфиксная запись

Примеры



- - * + 5 2 3 + 1 / 2 3
- 5 + 2 * 3 - 1 + 2 / 3
- 5 2 + 3 * 1 2 3 / + -



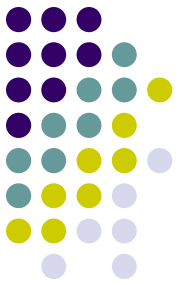
Упражнения

- Предложить алгоритм вычисления выражений, заданных в постфиксной записи (указание: используйте стек)
- Применение какого из обходов позволяет вывести элементы дерева поиска в отсортированном порядке?



Очередь с приоритетами

- Структура данных с двумя операциями:
 - добавить элемент и назначить ему приоритет
 - извлечь элемент с наименьшим приоритетом
- Как с помощью такой структуры данных реализовать стек? Обычную очередь?



Простая реализация

- Храним массив записей

```
element = record
```

```
    data, priority : integer;
```

```
end;
```

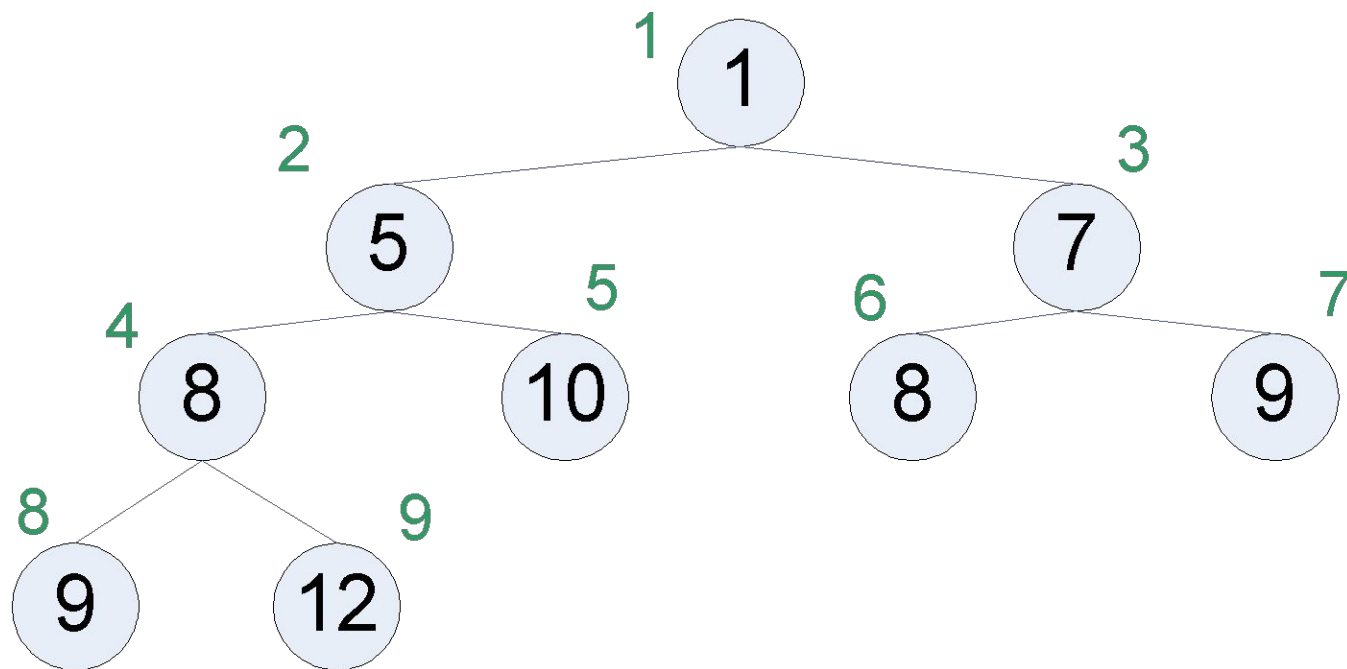
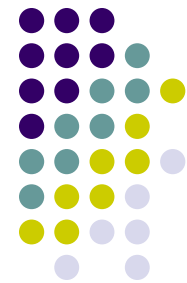
- Добавление – в конец массива ($O(1)$)
- Извлечение – просматриваем весь массив и ищем минимум ($O(n)$)



Структура данных «куча»

- Полное двоичное дерево
- Хранится в массиве:
 - Левый ребенок – $2 * i$
 - Правый ребенок – $2 * i + 1$
 - Родитель – $i \text{ div } 2$
- **Основное свойство: $heap[i] > heap[i \text{ div } 2]$**
- **Высота есть $O(\log n)$**
- Можно рассматривать кучу для максимума, а для минимума
- Иногда эту структуру данных называют «пирамида»

Пример



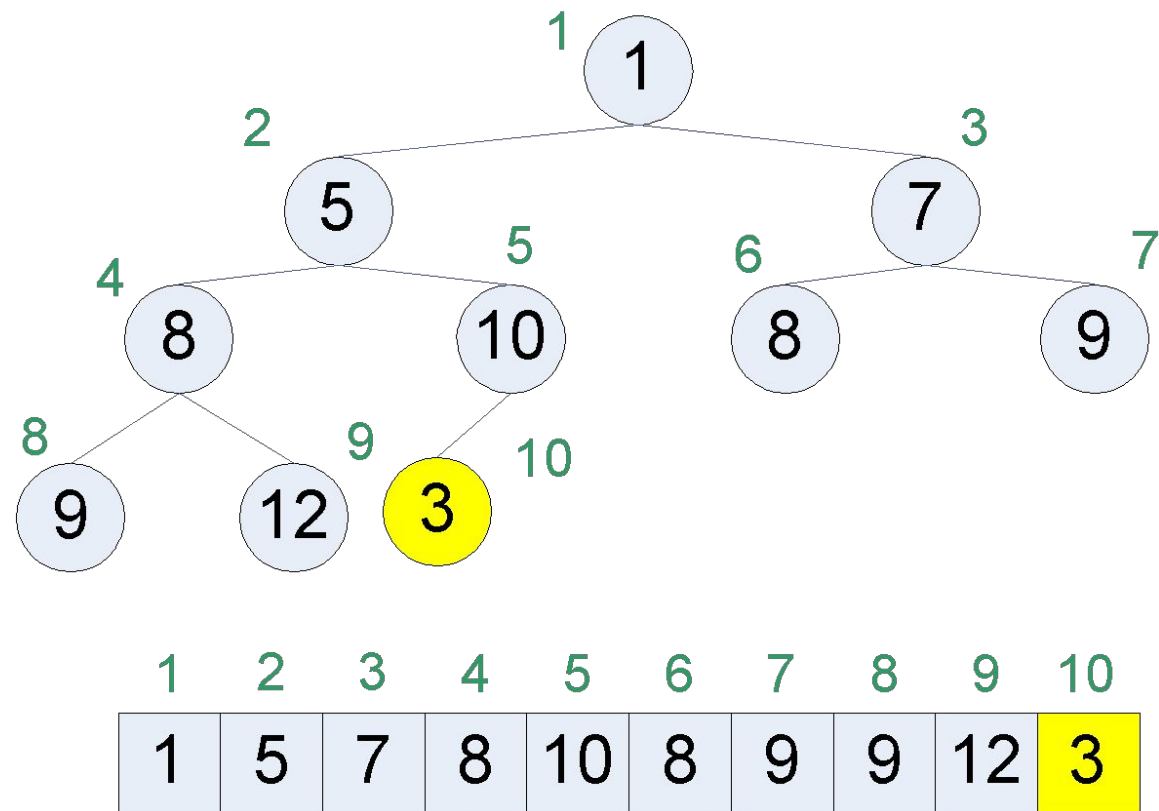
1 2 3 4 5 6 7 8 9

1	5	7	8	10	8	9	9	12
---	---	---	---	----	---	---	---	----



Добавление элемента (1)

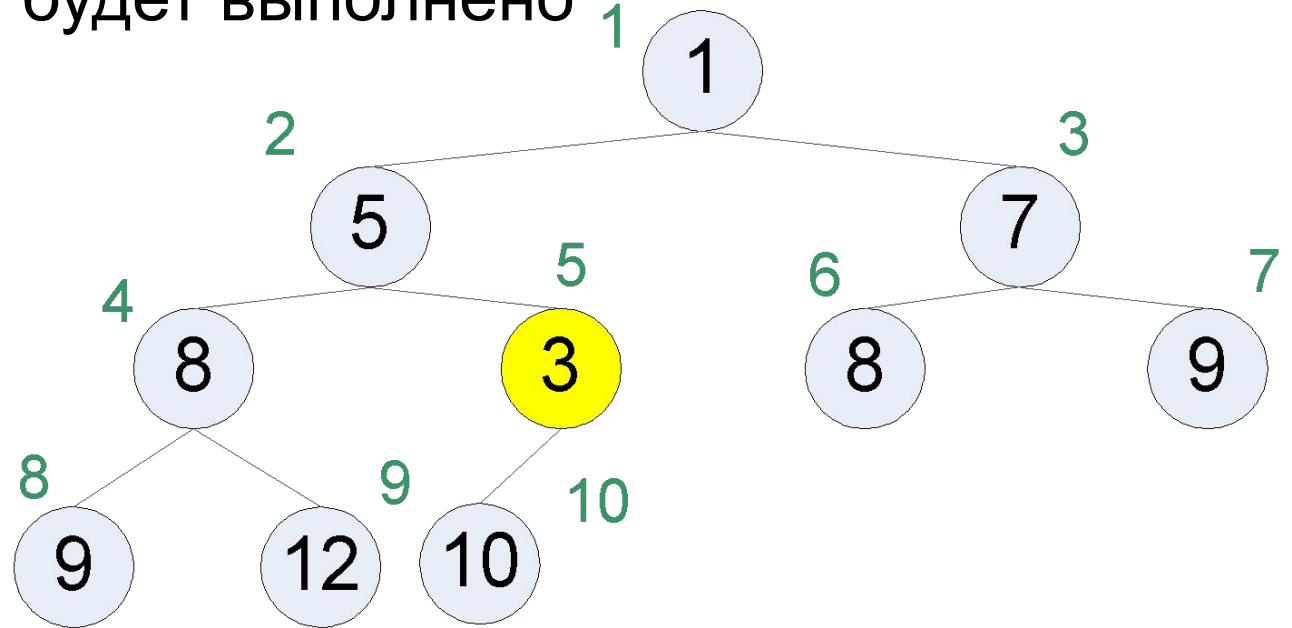
- Добавим в конец массива
- После этого основное свойство кучи может быть нарушено

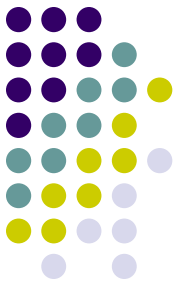




Добавление элемента (2)

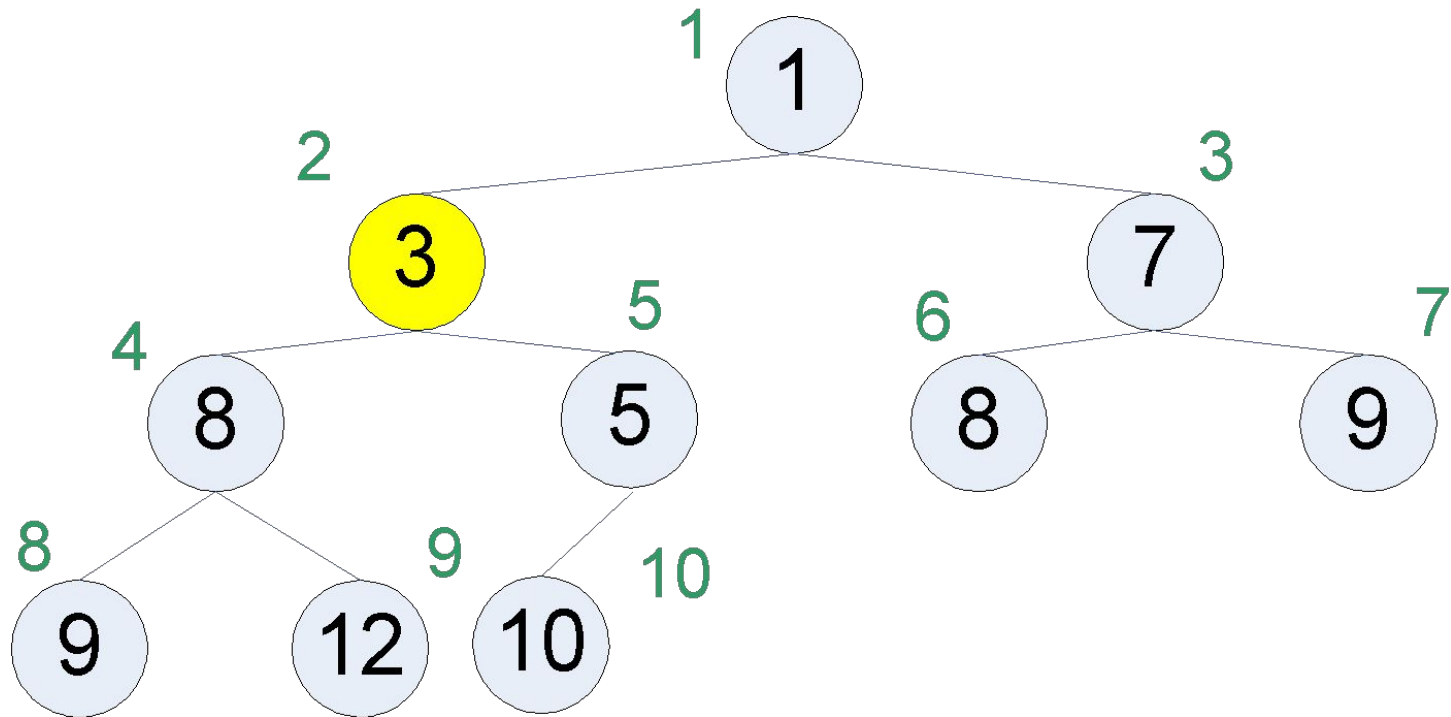
- Необходимо восстановить нарушенное основное свойство
- Для этого выполним просеивание вверх – будем поднимать элемент вверх до тех пор, пока свойство не будет выполнено





Добавление элемента (3)

- Число обменов не превышает высоту дерева – то есть $O(\log n)$



Добавление элемента (4)



```
procedure add(x : integer);
begin
  inc(size);
  heap[size] := x;
  i := size;
  while (i > 1) and do begin
    if (heap[i div 2] > heap[i]) then begin
      swap(heap[i], heap[i div 2]);
    end else begin
      break;
    end;
  end;
end;
end;
```



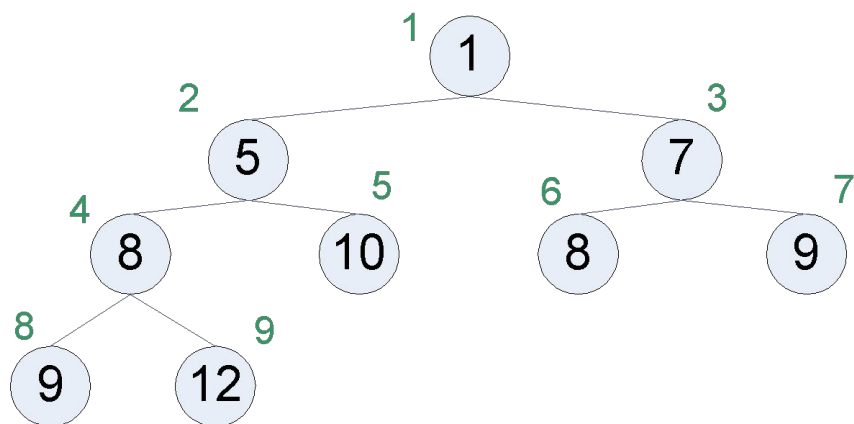
Извлечение элемента (1)

- Минимальный элемент находится на вершине кучи
- Поменяем его местами с последним и удалим
- После этого может нарушиться основное свойство

Извлечение элемента (2)

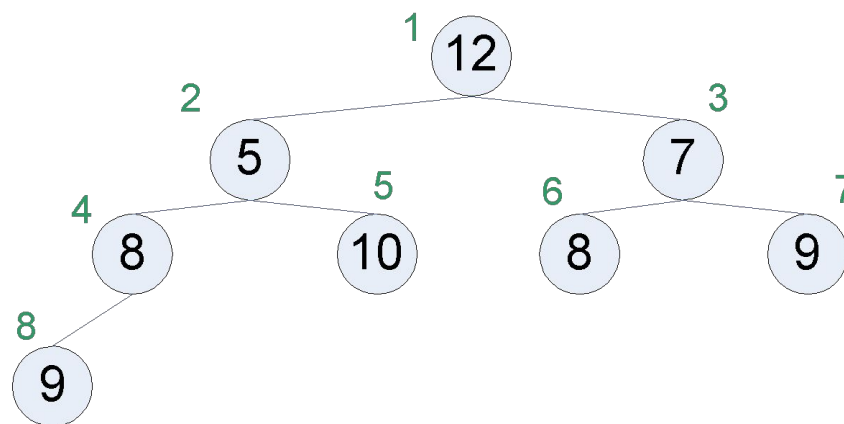


Было



1	2	3	4	5	6	7	8	9
1	5	7	8	10	8	9	9	12

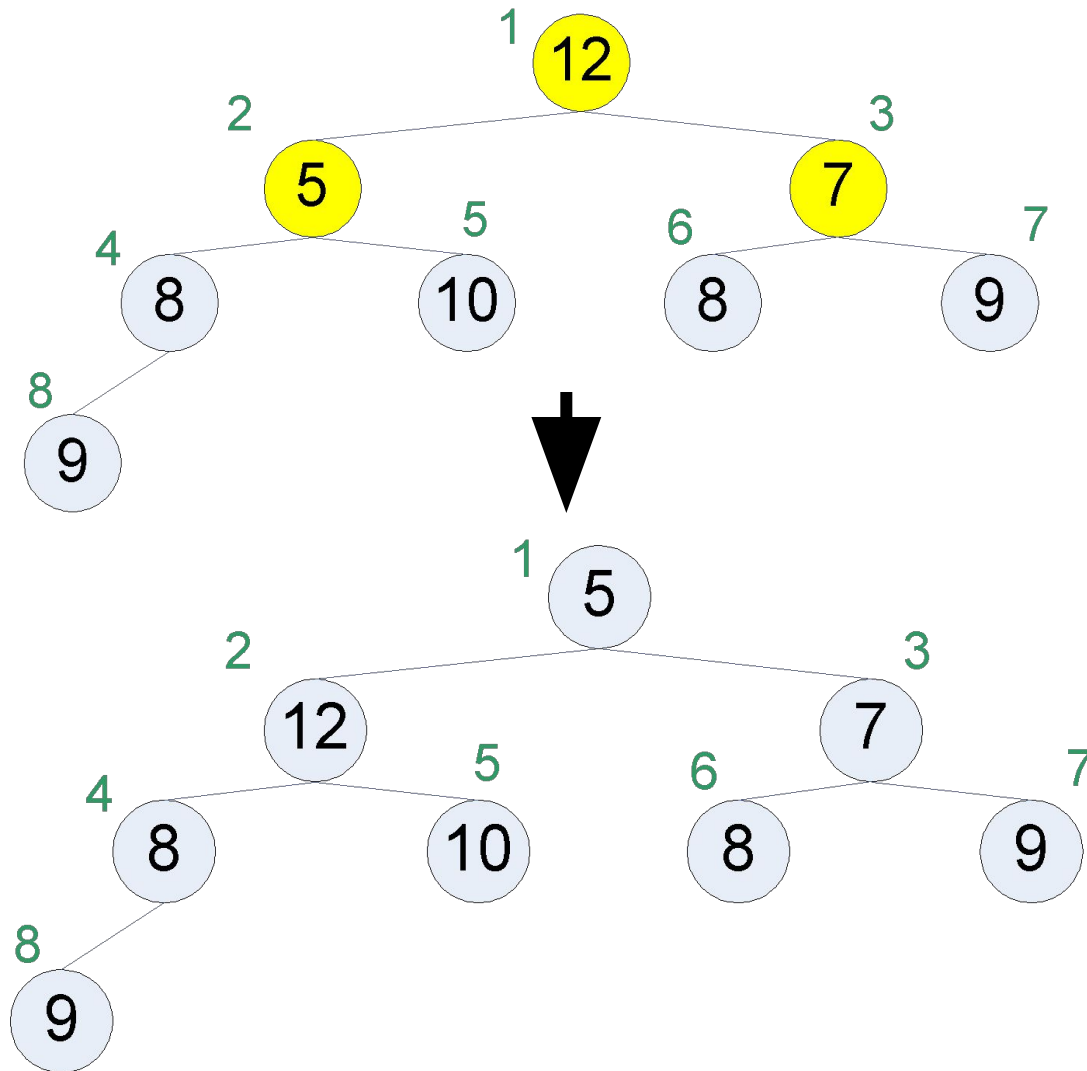
Стало



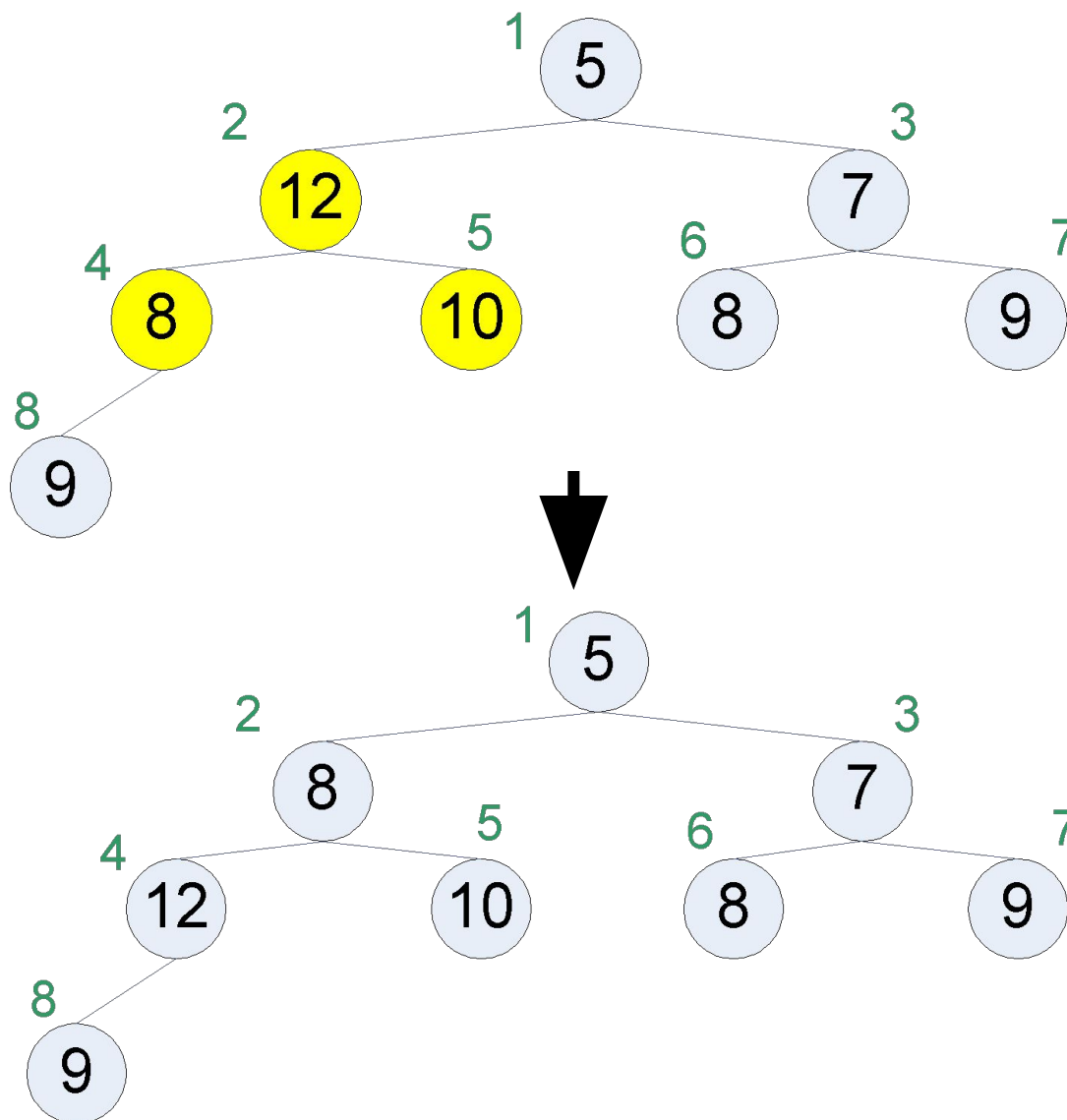
1	2	3	4	5	6	7	8
12	5	7	8	10	8	9	9

Необходимо поместить верхний элемент на его место!

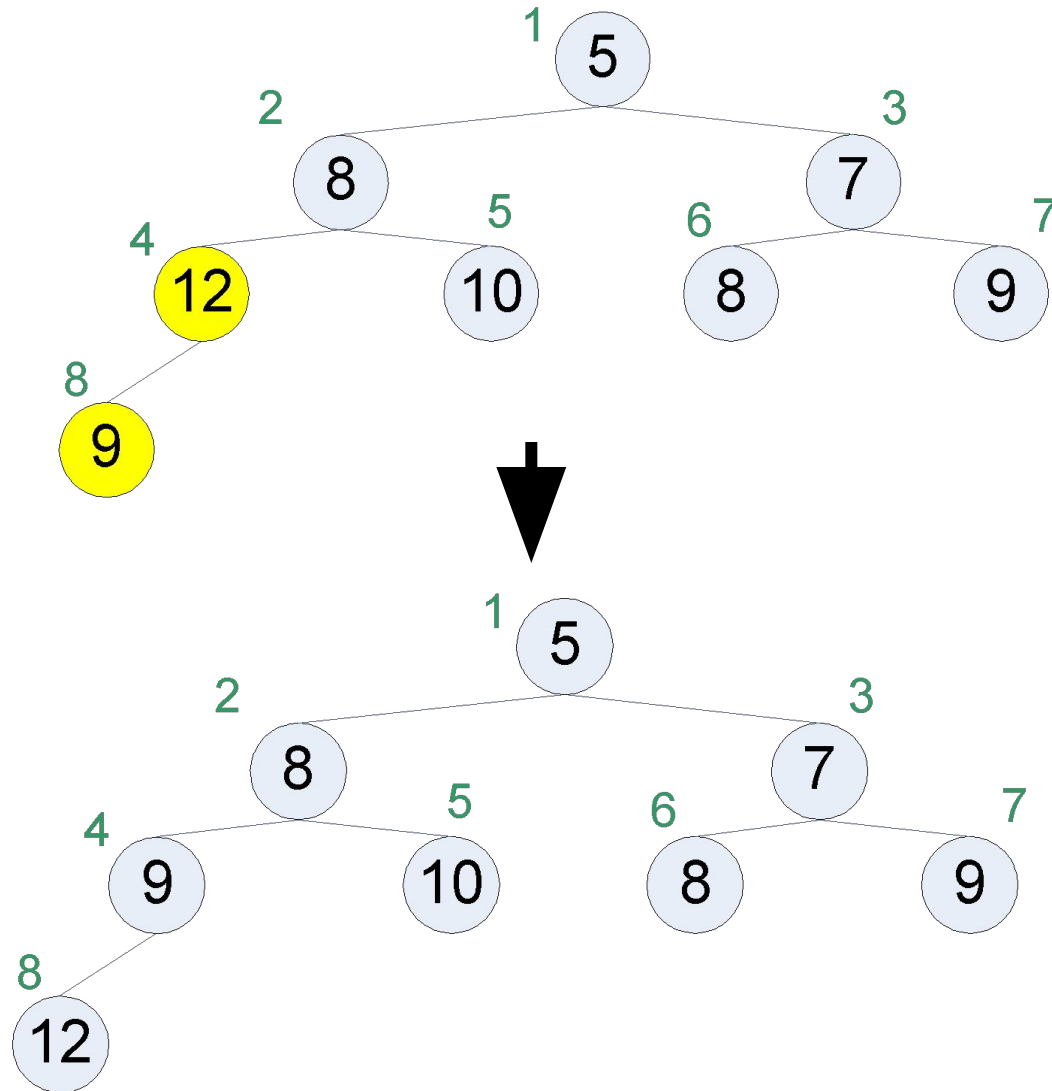
Извлечение элемента (3)



Извлечение элемента (4)



Извлечение элемента (5)



Извлечение элемента (5)



```
function remove() : integer;
begin
    result := heap[1];
    heap[1] := heap[size];
    dec(size);
    i := 1;
    while (2 * i <= size) and do begin
        min := i;
        if (heap[2 * i] < heap[min]) then begin
            min := 2 * i;
        end;
        if (2 * i + 1 <= size) and
            (heap[2 * i + 1] <= heap[min]) then begin
            min := 2 * i + 1;
        end;
        if (min = i) then begin
            break;
        end;
        swap(heap[i], heap[min]);
        i := min;
    end;
end;
```


Сортировка с помощью кучи



- Можно добавить все элементы массива в кучу, а потом по очереди извлечь
- Требуется дополнительный массив – на хранение кучи
- Чтобы сделать без дополнительной памяти надо:
 - Научиться преобразовывать массив в кучу
 - Научиться преобразовывать кучу в упорядоченный массив

Построение кучи из массива



```
for i := n div 2 downto 1 do begin
    siftDown(i);
end;
procedure siftDown(i : integer);
begin
    while (2 * i <= size) and do begin
        min := i;
        if (heap[2 * i] < heap[min]) then begin
            min := 2 * i;
        end;
        if (2 * i + 1 <= size) and
            (heap[2 * i + 1] <= heap[min]) then begin
            min := 2 * i + 1;
        end;
        if (min = i) then begin
            break;
        end;
        swap(heap[i], heap[min]);
        i := min;
    end;
end;
```



Время работы

- Грубая оценка – $O(n \log n)$
- Более точная оценка – $O(n)$

$$\sum_{i=0}^h \frac{n}{2^i} \leq \sum_{i=0}^{\infty} \frac{n}{2^i} = 2n$$

Построение упорядоченного массива из кучи

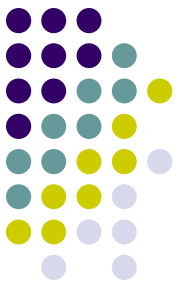


```
for j := 1 to n do begin
  swap(heap[1], heap[size]);
  dec(size);
  i := 1;
  while (2 * i <= size) and do begin
    min := i;
    if (heap[2 * i] < heap[min]) then begin
      min := 2 * i;
    end;
    if (2 * i + 1 <= size) and
      (heap[2 * i + 1] <= heap[min]) then begin
      min := 2 * i + 1;
    end;
    if (min = i) then begin
      break;
    end;
    swap(heap[i], heap[min]);
    i := min;
  end;
end;
```

Время работы heapSort



- Общее время работы heapSort есть $O(n + n \log n) = O(n \log n)$



Выводы

- Двоичные деревья поиска позволяют быстро выполнять словарные операции при условии, что у них небольшая высота
- С помощью двоичных деревьев можно представлять математические выражения
- Очередь с приоритетами реализуется на основе структуры данных «куча»

Спасибо за внимание!

Вопросы? Комментарии?

fedor.tsarev@gmail.com

