

Министерство науки и высшего образования РФ ФГБОУ ВО “Удмуртский  
государственный университет” институт математики,  
информационных  
технологий и физики кафедра теоретических основ информатики

Отчет по теме

## **«Поиск в деревьях»**

Выполнил: студент  
группы ОБ-02.03.02.01-41  
Полянских Сергей Владимирович

Ижевск, 2023

## Задание

1. Построить дерево поиска с элементами — вещественными числами.
2. Определить количество элементов дерева на каждом уровне.
3. Удалить элементы с заданными значениями.
4. Вывести дерево на экран.

## Описание решения

Для построения дерева поиска нам нужно использовать рекурсивные функции для добавления элементов в дерево. Начнем с определения типа узла дерева:

*Листинг 1. Узел дерева поиска.*

```
type Node =  
  | Leaf  
  | Node of float * Node * Node
```

У нас есть два типа узлов: лист (конец дерева) и узел с вещественным числом и двумя потомками (левым и правым).

Теперь можем определить функцию добавления элемента в дерево. Данная функция рекурсивно обходит дерево и вставляет новый элемент в нужном месте, как в обычном дереве поиска.

*Листинг 2. Добавление элемента в дерево*

```
let rec insert x tree =  
  match tree with  
  | Leaf -> Node (x, Leaf, Leaf)  
  | Node (value, left, right) ->  
    if x < value then  
      Node (value, insert x left, right)  
    else  
      Node (value, left, insert x right)
```

Теперь нужно определить функцию для подсчета количества элементов на каждом уровне дерева. Мы можем использовать рекурсивную функцию, которая сначала определит глубину дерева, а затем пройдет по всем уровням дерева, подсчитывая количество элементов на каждом уровне:

*Листинг 3. Подсчет элементов на уровне*

```
let rec count_levels tree =  
  let rec count_nodes_at_level level = function  
    | Leaf -> 0  
    | Node (_, left, right) ->  
      match level with  
      | 0 -> 1  
      | _ -> count_nodes_at_level (level - 1) left +  
count_nodes_at_level (level - 1) right  
    let rec calculate_depth = function  
      | Leaf -> 0  
      | Node (_, left, right) -> 1 + max (calculate_depth left)  
(calculate_depth right)  
    let depth = calculate_depth tree  
    [0..depth-1] |> List.map (count_nodes_at_level >> string) |>  
String.concat "\n"
```

***count\_levels*** определяет две внутренние функции ***count\_nodes\_at\_level*** и ***calculate\_depth***. ***calculate\_depth*** рекурсивно определяет глубину дерева путем сравнения глубины левого и правого поддеревя. Функция ***count\_nodes\_at\_level*** используется для подсчета количества элементов на каждом уровне дерева. Она рекурсивно обходит дерево и суммирует количество узлов на каждом уровне.

Для удаления элементов из дерева, нужно определить функцию, которая будет удалять узел с заданным значением, заменяя его на максимальный элемент из левого поддеревя (или минимальный из

правого). Затем мы можем использовать рекурсивную функцию для удаления всех элементов с заданными значениями.

***delete*** определяет два случая - когда узел является листом (тогда мы его удаляем) и когда узел имеет двух потомков, в этом случае мы заменяем узел максимальным (или минимальным) элементом поддеревя. Мы также используем вспомогательную функцию ***remove\_min*** для нахождения минимального элемента в правом поддереве, который используется для замены узла с двумя потомками.

Чтобы удалить все элементы с заданными значениями, мы можем использовать рекурсивную функцию, которая будет вызывать ***delete*** для каждого элемента в списке.

#### *Листинг 4. Удаление узла*

```
let rec delete value = function
```

```
| Leaf -> Leaf
```

```
| Node (x, left, right) ->
```

```
  if value = x then
```

```
    match left, right with
```

```
    | _, Leaf -> left
```

```
    | Leaf, _ -> right
```

```
    | _, _ ->
```

```
      let min_node, new_left = remove_min right
```

```
      Node (min_node, left, new_left)
```

```
  else if value < x then
```

```
    Node (x, delete value left, right)
```

```
  else
```

```
    Node (x, left, delete value right)
```

```
and remove_min = function
```

```
| Leaf -> failwith "invalid argument: Tree has no elements"
```

```

| Node (x, Leaf, _) -> x, Leaf
| Node (x, left, right) ->
    let min, new_left = remove_min left
    min, (Node(x, new_left, right))

let rec delete_all xs tree =
    match xs with
    | [] -> tree
    | x::xs -> delete_all xs (delete x tree)

```

Для вывода на экран мы можем использовать обход в глубину и отступы для отображения каждого узла. Мы также можем использовать символы для отображения соединения между узлами.

Эта функция определяет рекурсивную функцию ***printTree***, которая выводит узел на экран, добавляя отступы для каждого уровня дерева. Функция работает рекурсивно для левого и правого поддеревьев. Когда мы доходим до листьев, они также выводятся на экран с символом "-".

### *Листинг 5. Вывод на экран*

```
let rec printTree indent = function
  | Leaf ->
    printfn "%s -" indent
  | Node (value, left, right) ->
    printfn "%s|-- %g" indent value
    printTree (indent + "| ") left
    printTree (indent + " ") right

let print tree =
  printTree "" tree
```

## **Проверка работы**

Для проверки работы выполним следующую программу, в которой будут последовательно выполняться следующие шаги:

1. Создание и заполнение дерева.
2. Печать дерева.
3. Удаление вершины дерева.
4. Печать дерева.
5. Вывод подсчета узлов на уровнях дерева.

## Листинг 6. Проверка работы

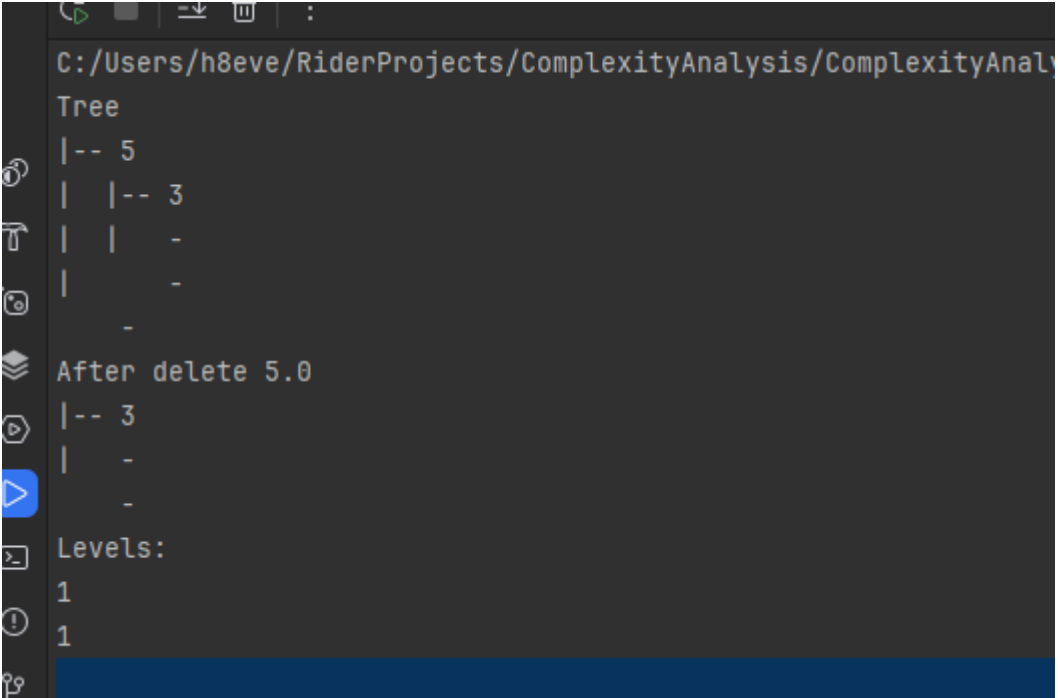
```
let nums = [5.0; 3.0; 1]
let tree = Array.fold (fun acc x -> insert x acc) Leaf nums
printfn "Tree"
tree |> print

printfn "After delete 5.0"
delete_all [5.0] tree |> print
let levels = count_levels tree

printfn "Levels:"
printf "${levels}"
```

После выполнения получаем следующий результат:

Рисунок 1. Вывод результатов.



```
C:/Users/h8eve/RiderProjects/ComplexityAnalysis/ComplexityAnaly
Tree
|-- 5
|  |-- 3
|  |  -
|  |  -
|  -
-
After delete 5.0
|-- 3
|  -
|  -
Levels:
1
1
```