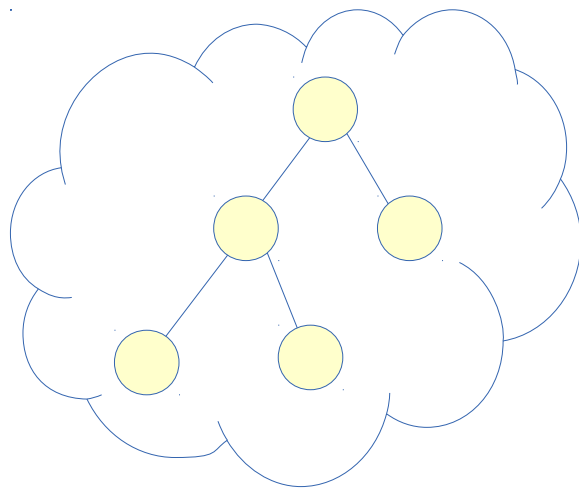


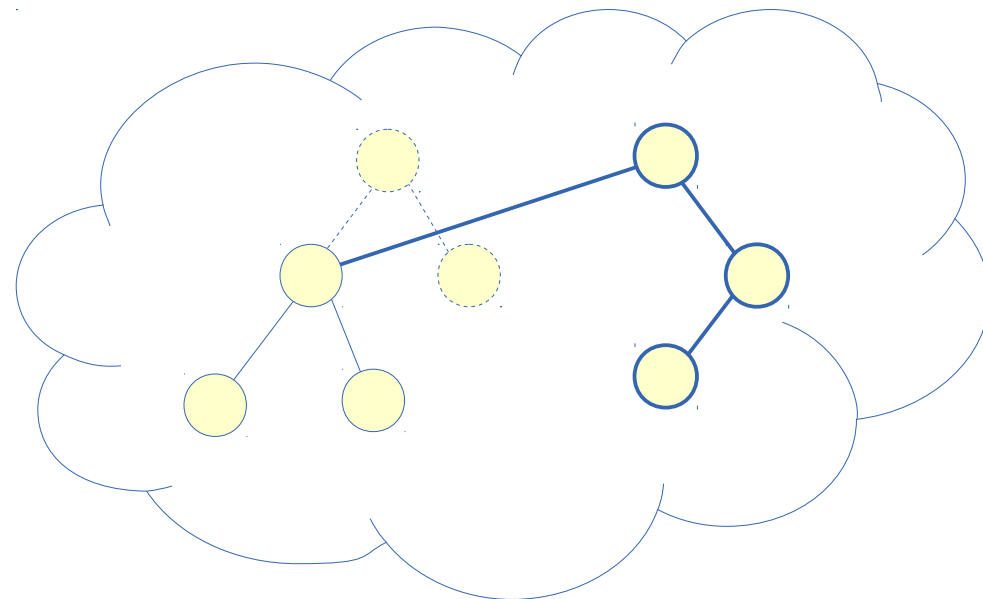
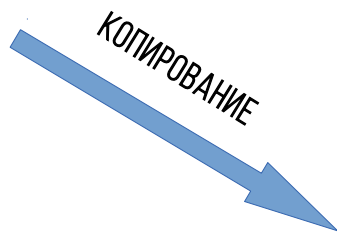
СТРУКТУРЫ ДАННЫХ

функциональный подход

Функциональные структуры данных — неизменяемые (persistent)



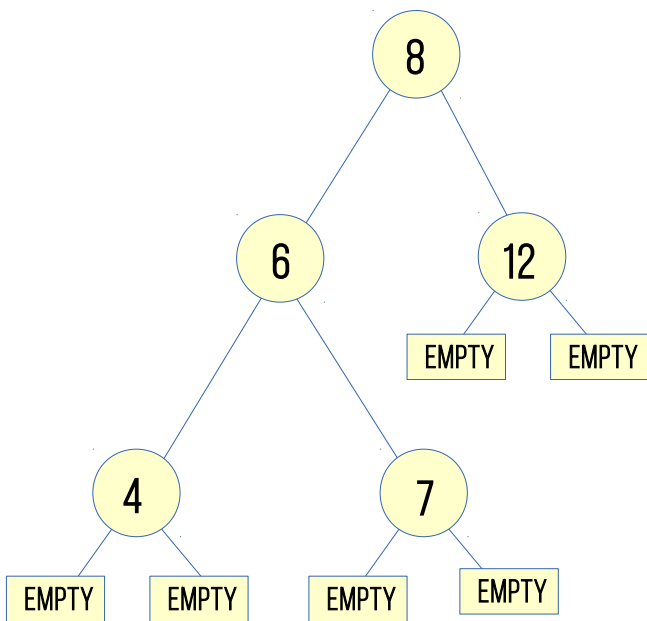
СОСТОЯНИЕ ДО ОБНОВЛЕНИЯ



СОСТОЯНИЕ ПОСЛЕ ОБНОВЛЕНИЯ

ДВОИЧНЫЕ ДЕРЕВЬЯ ПОИСКА

BINARY SEARCH TREES



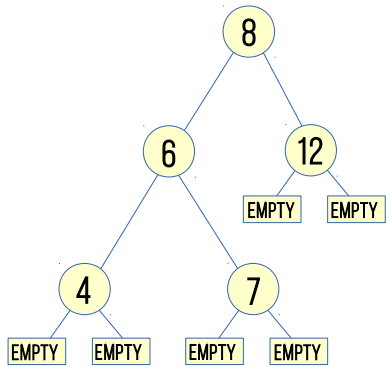
Свойства:

- Узлы могут содержать объекты или быть пустыми
- Каждый узел имеет два потомка
- Элемент в каждом узле больше любого элемента в левом поддереве и меньше любого элемента в правом

OCAML

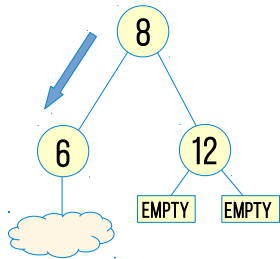
```
type 'a tree = E | T of 'a tree * 'a * 'a tree  
let my_tree = T (T (T (E, 4, E), 6, T (E, 7, E)), 8, T (E, 12, E))
```

МНОЖЕСТВА



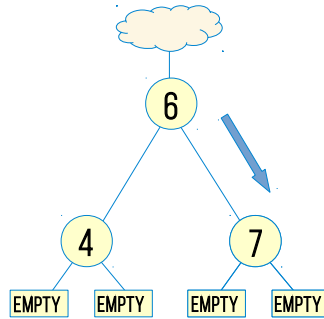
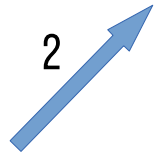
?

7



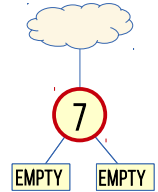
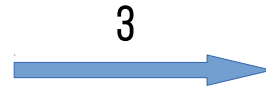
?

7



?

7

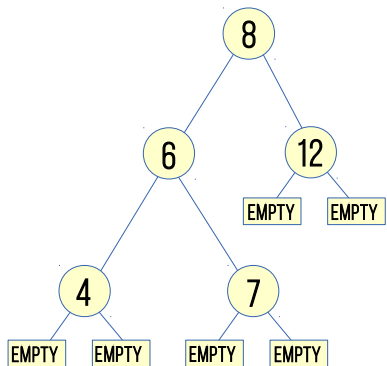


OCAML

```
let rec member n bst =  
  match bst with  
  | E -> false  
  | T (left, x, right) ->  
    if n < x then  
      member n left  
    else if n > x then  
      member n right  
    else true
```

МНОЖЕСТВА

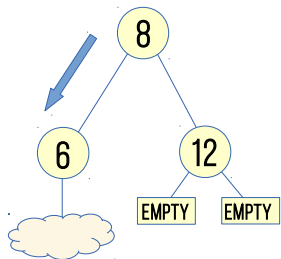
ПРОДОЛЖЕНИЕ



+

5

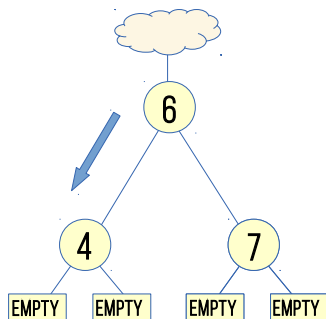
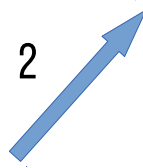
1



+

5

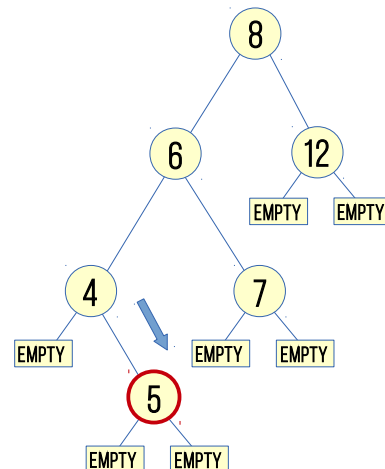
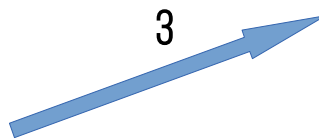
2



+

5

3



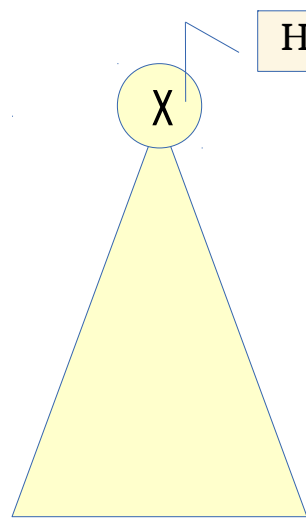
OCAML

```

let rec insert n bst =
  match bst with
  | E -> T (E, n, E)
  | T (left, x, right) ->
    if n < x then
      T (insert n left, x, right)
    else if n > x then
      T (left, x, insert n right)
    else T(left, x, right)
    
```

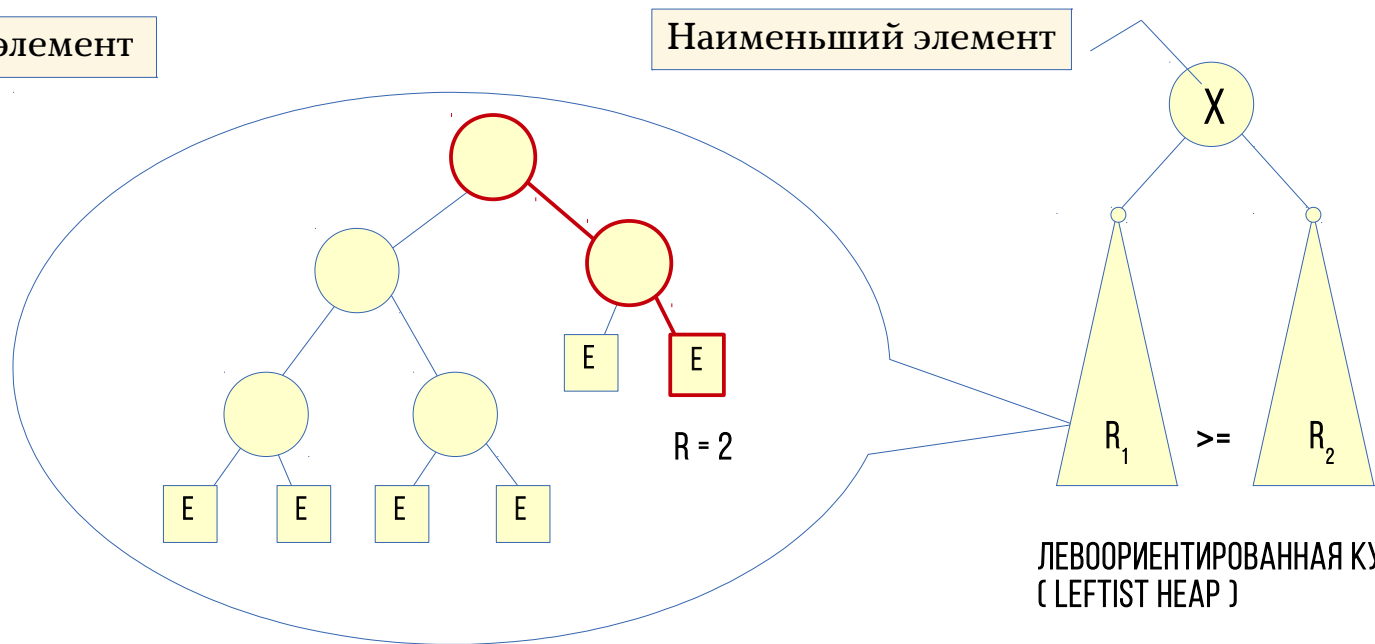
ЛЕВООРИЕНТИРОВАННЫЕ КУЧИ

LEFTIST HEAPS



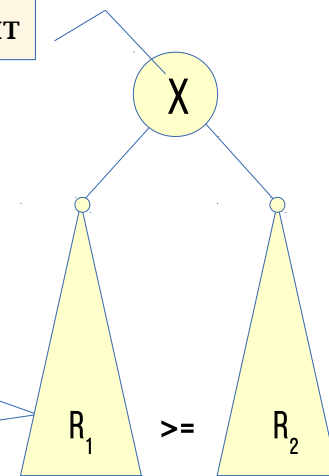
Наименьший элемент

КУЧА (HEAP)



Наименьший элемент

$R = 2$



ЛЕВООРИЕНТИРОВАННАЯ КУЧА
(LEFTIST HEAP)

ЛЕВООРИЕНТИРОВАННЫЕ КУЧИ

ПРОДОЛЖЕНИЕ

Свойства:

- Правая периферия корня — кратчайший путь до внешнего узла
- Количество внутренних узлов равно $2^{\text{rank}(\text{root})} - 1$, где $\text{rank}(\text{root})$ — ранг корневого узла
- Из предыдущих свойств следует, что длина правой периферии в худшем случае логарифм размера кучи (по основанию 2)

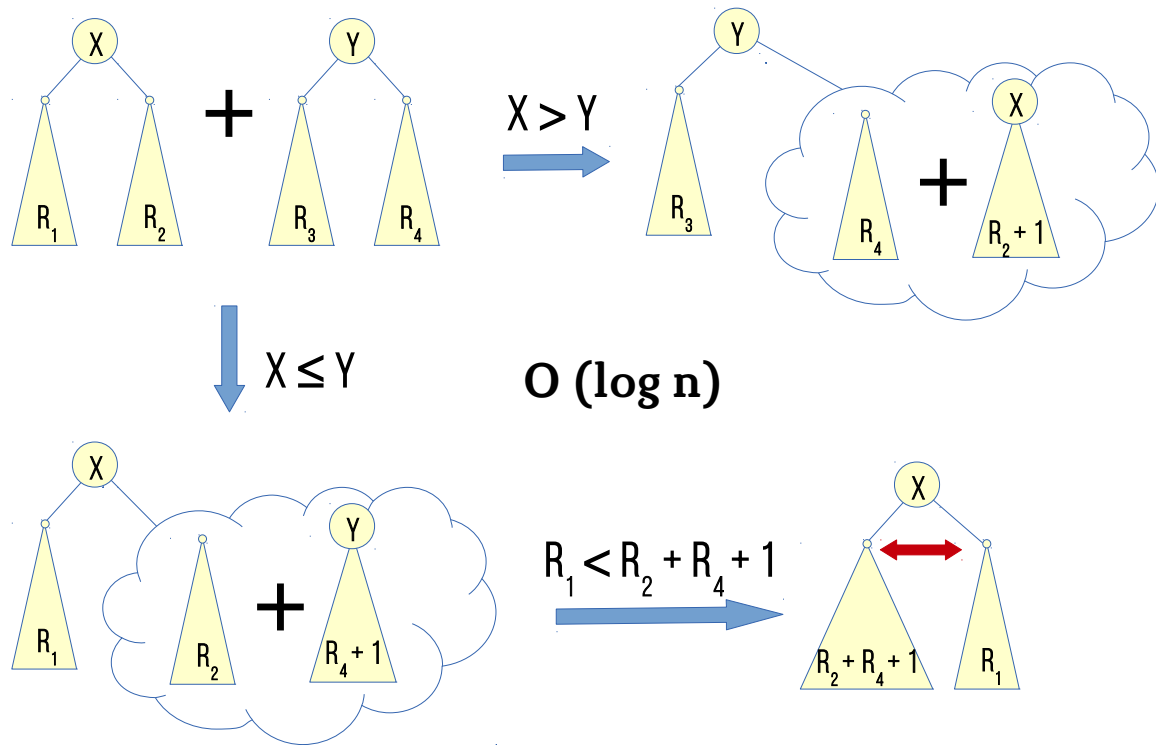
$$2^{\text{rank}(\text{root})} - 1 \leq n \Rightarrow 2^{\text{rank}(\text{root})} \leq n + 1 \Rightarrow \text{rank}(\text{root}) \leq \log_2(n + 1)$$

OCAML

```
type 'a heap = E | T of int * 'a * 'a heap * 'a heap
let my_heap = T(1, 1, T (1, 3, E, E), E)
```

ЛЕВООРИЕНТИРОВАННЫЕ КУЧИ

ПРОДОЛЖЕНИЕ



OCAML

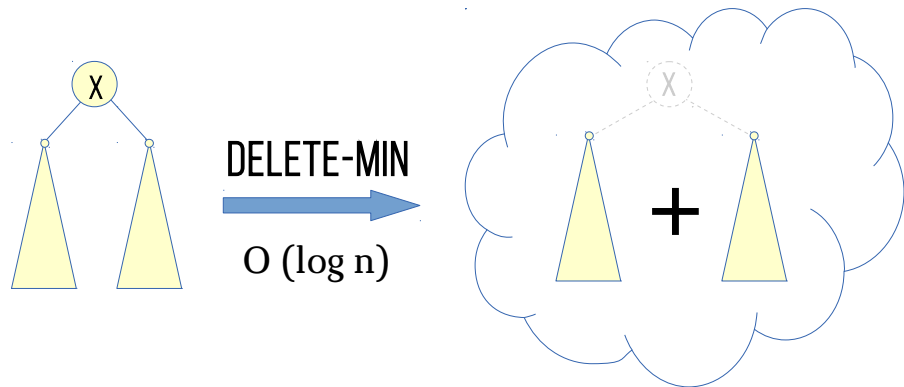
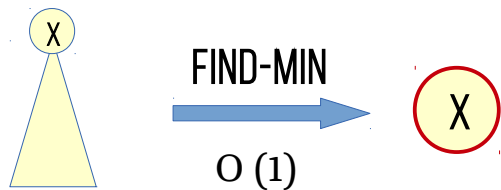
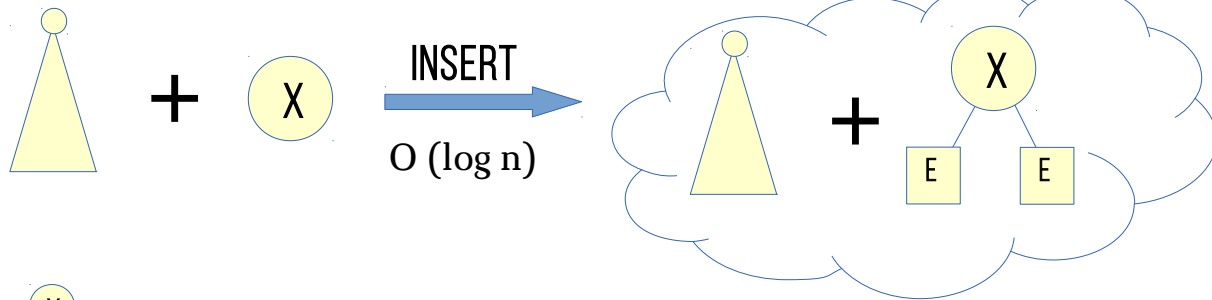
```
let rank = function
| E -> 0
| T (r, _, _, _) -> r
```

```
let makeT x a b =
  if rank a >= rank b then
    T (rank b + 1, x, a, b)
  else
    T (rank a + 1, x, b, a)
```

```
let rec merge h1 h2 =
  match (h1, h2) with
  | E, h -> h
  | h, E -> h
  | T (_, x, a1, b1), T (_, y, a2, b2) ->
    if x < y then
      makeT x a1 (merge b1 h2)
    else
      makeT y a2 (merge h1 b2)
```


ЛЕВООРИЕНТИРОВАННЫЕ КУЧИ

ПРОДОЛЖЕНИЕ



OCAML

```
let insert x h = merge (T (1, x, E, E)) h

let findMin = function
  | E -> failwith "empty"
  | T (_, x, _, _) -> x

let deleteMin = function
  | E -> failwith "empty"
  | T (_, _, a, b) -> merge a b
```

БИНОМИАЛЬНЫЕ КУЧИ

BINOMIAL HEAPS

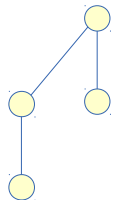
РАНГ = 0



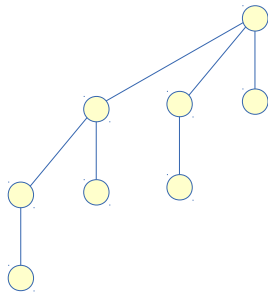
РАНГ = 1



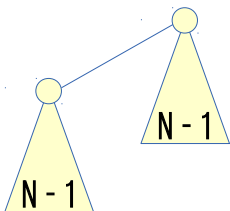
РАНГ = 2



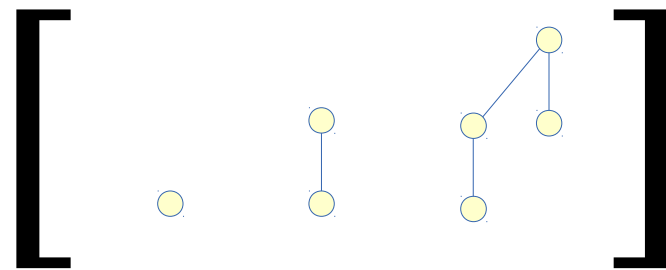
РАНГ = 3



РАНГ = N



- Дерево ранга N имеет 2^N узлов
- Список потомков хранится в убывающем порядке ранга
- Элементы хранятся в порядке кучи
- Чтобы сохранять порядок, дерево с большим корнем привязывается к дереву с меньшим корнем
- Связываться могут только деревья с одинаковым рангом
- Биномиальная куча — это список биномиальных деревьев в порядке возрастания ранга, каждое из которых не может иметь одинаковый ранг



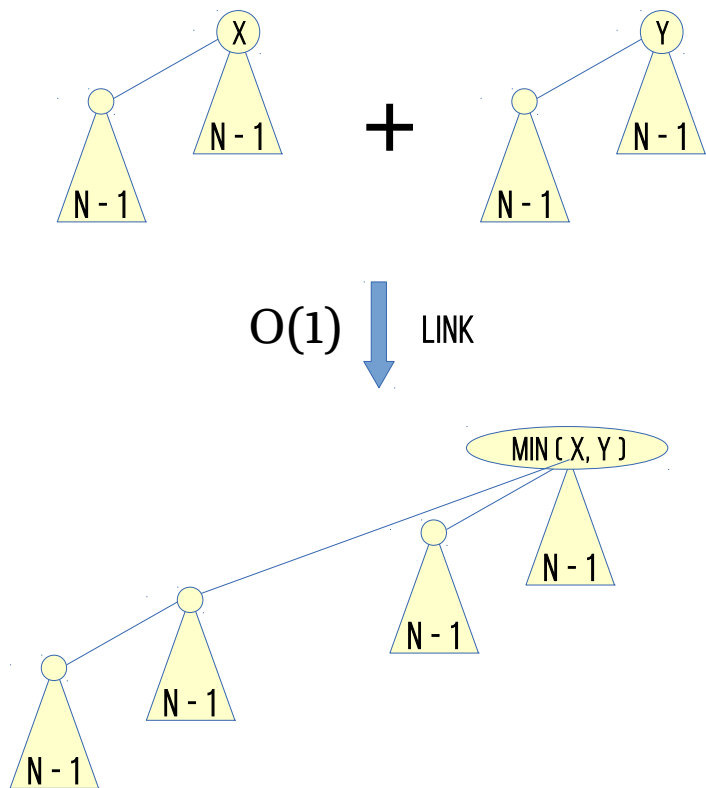
$$7_2 = 111$$

OCAML

```
type 'a bintree =  
  Node of int * 'a * 'a bintree list  
  
let my_heap =  
  [Node (0, 1, []); Node (1, 2, Node (0, 3, []))]
```

БИНОМИАЛЬНЫЕ КУЧИ

ПРОДОЛЖЕНИЕ



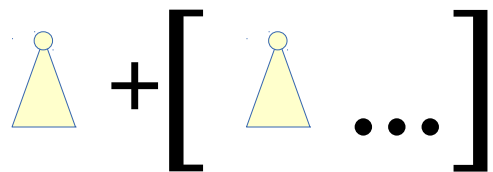
OCAML

```
let link (Node (r, x1, c1) as t1)
        (Node (_, x2, c2) as t2) =
  if x1 <= x2 then
    Node (r + 1, x1, t2 :: c1)
  else
    Node (r + 1, x2, t1 :: c2)
```

БИНОМИАЛЬНЫЕ КУЧИ

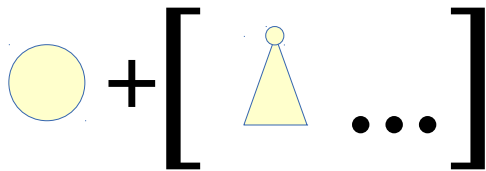
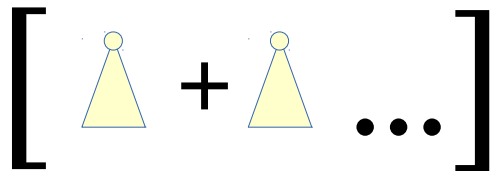
ПРОДОЛЖЕНИЕ

OCAML



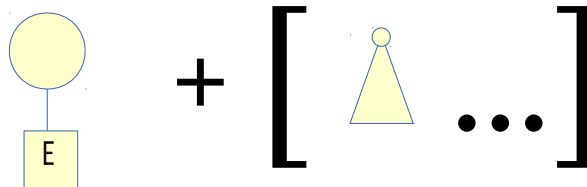
$O(\log n)$

INS-TREE



$O(\log n)$

INSERT

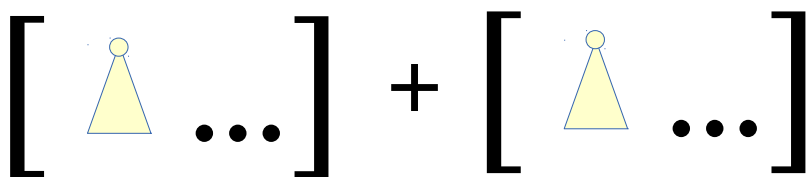


```
let rec insTree t ts =  
  match ts with  
  | [] -> [t]  
  | t' :: ts' ->  
    if rank t < rank t' then  
      t :: ts  
    else  
      insTree (link t t') ts'
```

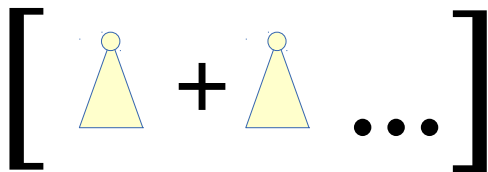
```
let insert x ts =  
  insTree (Node (0, x, [])) ts
```

БИНОМИАЛЬНЫЕ КУЧИ

ПРОДОЛЖЕНИЕ



$O(\log n)$ MERGE



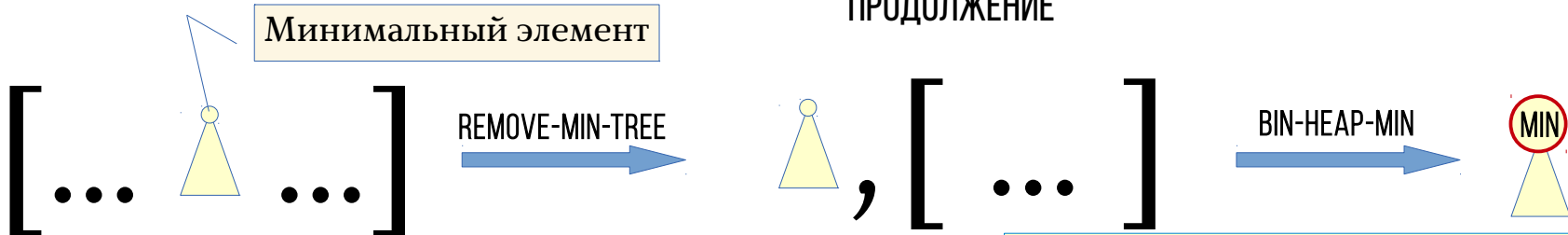
OCAML

```
let rec merge ts1 ts2 =  
  match (ts1, ts2) with  
  | _, t -> t  
  | t, _ -> t  
  | t1 :: ts1', t2 :: ts2' ->  
    if rank t1 < rank t2 then  
      t1 :: merge ts1' ts2  
    else if rank t2 < rank t1 then  
      t2 :: merge ts1 ts2'  
    else  
      insTree (link t1 t2) (merge ts1' ts2')
```

БИНОМИАЛЬНЫЕ КУЧИ

ПРОДОЛЖЕНИЕ

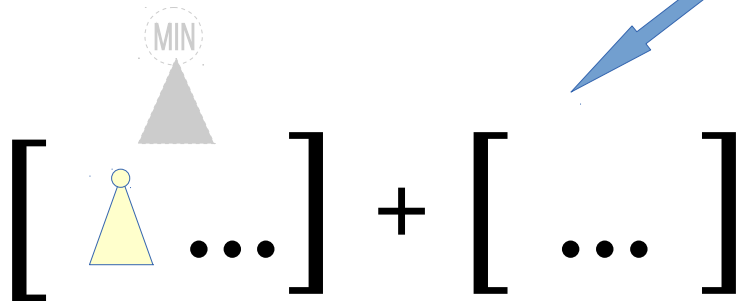
Минимальный элемент



OCAML

$O(\log n)$

BIN-HEAP-DELMIN



```
let root (Node (r, x, c)) = x;;
```

```
let rec removeMinTree = function
```

```
| [] -> failwith "empty"
```

```
| [t] -> t, []
```

```
| t :: ts ->
```

```
  let t', ts' = removeMinTree ts in
```

```
    if root t <= root t' then
```

```
      t, ts
```

```
    else
```

```
      t', t :: ts'
```

```
let findMin ts =
```

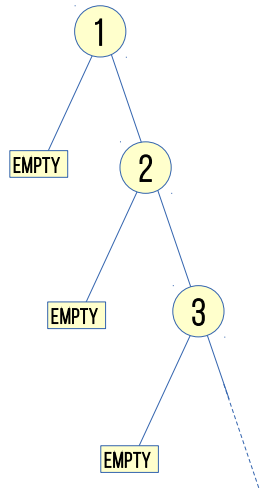
```
  let t, _ = removeMinTree ts in root t
```

```
let deleteMin ts =
```

```
  let Node (_, x, ts1), ts2 = removeMinTree ts in  
  merge (List.rev ts1) ts2
```

КРАСНО-ЧЁРНЫЕ ДЕРЕВЬЯ

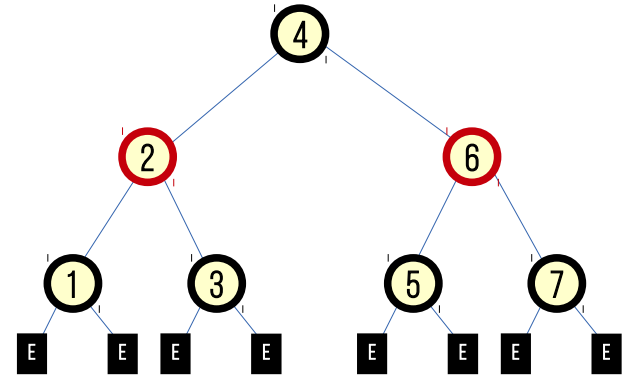
RED-BLACK TREES



$O(n)$

OCAML

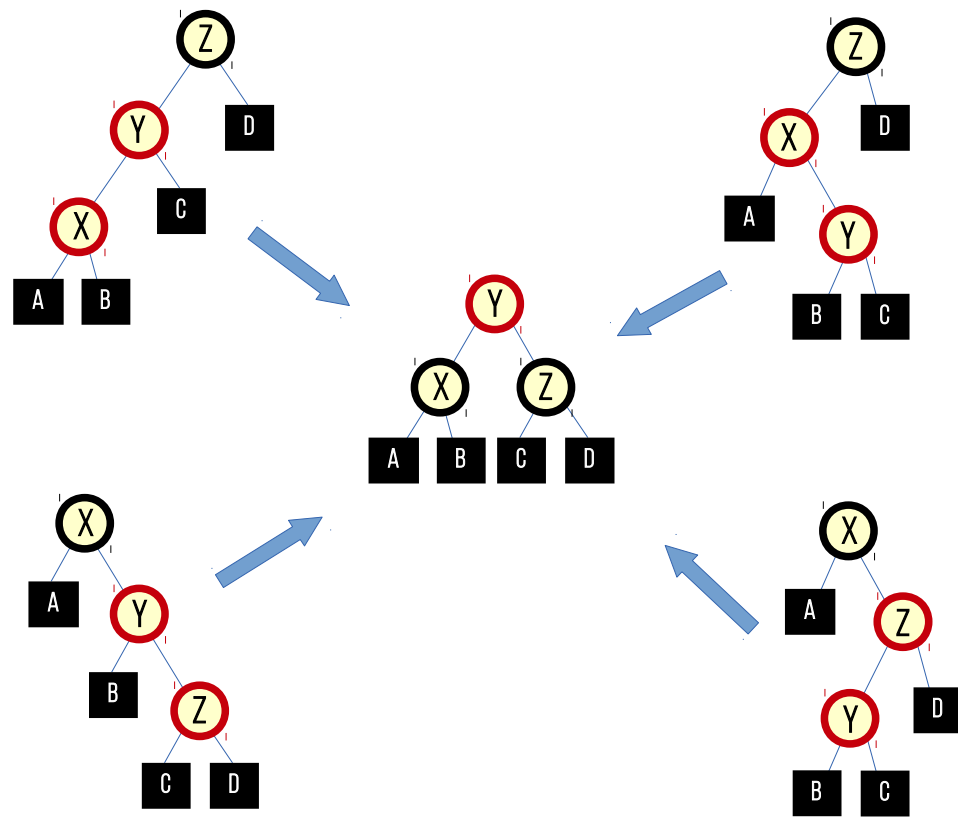
```
let insert x s =  
  let rec ins = function  
    | E -> T (R, E, x, E)  
    | T (color, a, y, b) ->  
      if x < y then  
        balance color (ins a) y b  
      else if y < x then  
        balance color a y (ins b)  
      else s in  
  let T (_, a, y, b) = ins s in  
  T (B, a, y, b)
```



$O(\log n)$

КРАСНО-ЧЁРНЫЕ ДЕРЕВЬЯ

ПРОДОЛЖЕНИЕ



OCAML

```
let balance col lt el rt =  
  match (col, lt, el, rt) with  
  | B, T (R, T (R, a, x, b), y, c), z, d  
  | B, T (R, a, x, T (R, b, y, c)), z, d  
  | B, a, x, T (R, T (R, b, y, c), z, d)  
  | B, a, x, T (R, b, y, T (R, c, z, d)) ->  
    T (R, T (B, a, x, b), y, T (B, c, z, d))  
  | _ -> T (col, lt, el, rt)
```


АМОРТИЗАЦИОННЫЙ АНАЛИЗ

AMORTIZED ANALYSIS

- Время выполнения последовательности операций усредняется
- Гарантируется средняя производительность операций в наихудшем случае
- Рассмотрим два распространённых метода анализа: метод банкира или бухгалтерского учёта (accounting method or banker's method) и метод физика или потенциалов (potential method or physicist's method)

МЕТОД БАНКИРА

BANKER'S METHOD

OCAML

- Каждый вид операций характеризуется своей амортизированной стоимостью.
- Операции, выполняющиеся быстрее амортизированной стоимости, сохраняют кредиты, которые идут на оплату других операций

Пример (очередь FIFO):

- Операция `append` выполняется за время $O(1)$, операция `tail` за $O(n)$ в худшем случае. Однако последовательность из `append` или `tail` выполняется за амортизированное время $O(1)$. Назначим `append` стоимость 2, один кредит расходуется на саму операцию, второй накапливается. Операция `tail`, которая не переворачивает хвостовой список не потребляет и не добавляет кредитов. `Tail`, которая переворачивает хвостовой список, выполняет $m + 1$ шагов, где m — длина хвостового списка, и потребляет m кредитов. Значит амортизированная стоимость этой операции $m + 1 - m = 1$.

```
type 'a queue =  
  'a list * 'a list;;  
  
let q = [3; 6; 4], [7; 1; 2]  
  
let checkf = function  
  | [], r -> List.rev r, []  
  | _ as q -> q  
  
let head = function  
  | [], _ -> failwith "empty"  
  | x :: f, _ -> x  
  
let tail = function  
  | [], _ -> failwith "empty"  
  | x :: f, r -> checkf (f, r)  
  
let append ((f, r), x) =  
  checkf (f, x :: r)
```

МЕТОД ФИЗИКА

PHYSICIST'S METHOD

OCAML

- Дана функция Φ , которая принимает объект и возвращает его потенциал
- Амортизированная стоимость операции равна сумме фактической стоимости и разности потенциалов между текущим состоянием объекта и предыдущим $a_i = t_i + \Phi(d_i) - \Phi(d_{i-1})$
- Реальная стоимость последовательности операций будет равна сумме амортизированных стоимостей операций плюс разность потенциалов между первым и последним состоянием бъекта.

Пример (очередь FIFO):

- Φ — длина хвостового списка. Каждая операция `append` увеличивает потенциал на 1. Каждый вызов `tail`, который переворачивает хвостовой список занимает $m + 1$ шагов и уменьшает потенциал на m . Амортизированная стоимость равна $m + 1 - m = 1$.

```
type 'a queue =  
  'a list * 'a list;;  
  
let q = [3; 6; 4], [7; 1; 2]  
  
let checkf = function  
  | [], r -> List.rev r, []  
  | _ as q -> q  
  
let head = function  
  | [], _ -> failwith "empty"  
  | x :: f, _ -> x  
  
let tail = function  
  | [], _ -> failwith "empty"  
  | x :: f, r -> checkf (f, r)  
  
let append ((f, r), x) =  
  checkf (f, x :: r)
```

РАСШИРЯЮЩИЕСЯ ДЕРЕВЬЯ

SPLAY HEAPS

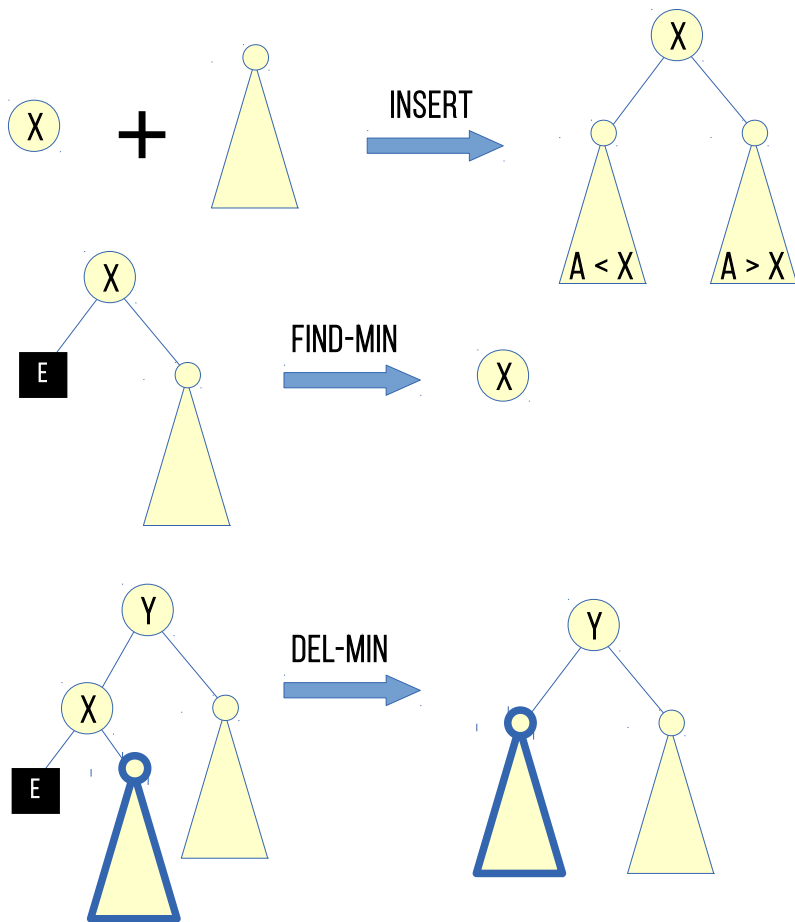
- Расширяющиеся деревья близки к сбалансированным двоичным деревьям поиска
- Не хранят никакую информацию о балансе явно
- Каждая операция, включая запросы, а не только обновления, перестраивает дерево, увеличивая сбалансированность

OCAML

```
type 'a heap = E | T of 'a heap * 'a * 'a heap
```

РАСШИРЯЮЩИЕСЯ ДЕРЕВЬЯ

ПРОДОЛЖЕНИЕ

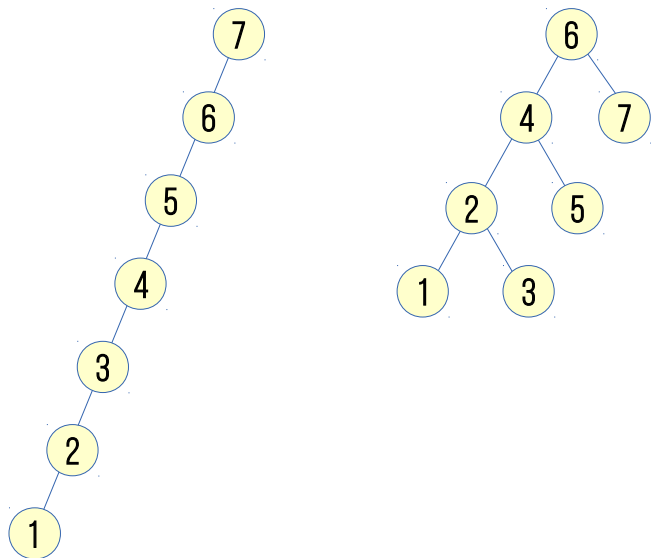


OCAML

```
let insert x t =  
  let a, b = partition x t in  
  T (a, x, b)  
  
let rec findMin = function  
| E -> failwith "empty"  
| T (E, x, b) -> x  
| T (a, x, b) -> findMin a  
  
let rec deleteMin = function  
| E -> failwith "empty"  
| T (E, x, b) -> b  
| T (T (E, x, b), y, c) -> T (b, y, c)  
| T (T (a, x, b), y, c) ->  
  T (deleteMin a, x, T (b, y, c))
```

РАСШИРЯЮЩИЕСЯ ДЕРЕВЬЯ

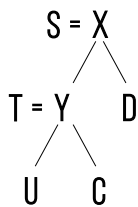
ПРОДОЛЖЕНИЕ



partition 0 t

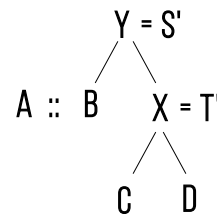
OCAML

```
let rec partition el t =
  match t with
  | E -> E, E
  | T (a, x, b) ->
    if x <= el then
      match b with
      | E -> t, E
      | T (b1, y, b2) ->
        if y <= el then
          let small, big = partition el b2 in
            T (T (a, x, b), y, small), big
        else
          let small, big = partition el b1 in
            T(a, x, small), T (big, y, b2)
    else
      match a with
      | E -> E, t
      | T (a1, y, a2) ->
        if y <= el then
          let small, big = partition el a2 in
            T (a1, y, small), T (big, x, b)
        else
          let small, big = partition el a1 in
            small, T (big, y, T (a2, x, b))
```



РАСШИРЯЮЩИЕСЯ ДЕРЕВЬЯ

ПРОДОЛЖЕНИЕ



Амортизированная стоимость операции insert равна $O(\log n)$

- $T(t)$ — реальная стоимость вызова partition
- $A(t) = T(t) + \Phi(a) + \Phi(b) - \Phi(t)$ — амортизированная стоимость, a и b — поддеревья, возвращенные этими функциями
- $\#t$ — размер дерева t плюс 1
- $\phi(t) = \log(\#t)$ — потенциал узла t
- $\Phi(t)$ — потенциал всего дерева, равный сумме потенциалов всех узлов

Лемма 1:

Для всех положительных x , y и z , таких, что $y + z \leq x$
 $1 + \log y + \log z < 2 \log x$

Предположим, что $y \leq z$, тогда $y \leq x/2$ и $z < x$. Тогда

$1 + \log y \leq \log x$ и $\log z < \log x$

$$\begin{aligned}
 & A(s) \\
 = & \{\text{определение } A\} \\
 & T(s) + \Phi(a) + \Phi(s') - \Phi(s) \\
 = & \{T(s) = 1 + T(u)\} \\
 & 1 + T(u) + \Phi(a) + \Phi(s') - \Phi(s) \\
 = & \{T(u) = A(u) - \Phi(a) - \Phi(b) + \Phi(u)\} \\
 & 1 + A(u) - \Phi(a) - \Phi(b) + \Phi(u) + \Phi(a) + \Phi(s') - \Phi(s) \\
 = & \{\text{упростим и распишем } \Phi(s') \text{ и } \Phi(s)\} \\
 & 1 + A(u) + \phi(s') + \phi(t') - \phi(s) - \phi(t) \\
 \leq & \{\text{индукция: } A(u) \leq 1 + 2\phi(u)\} \\
 & 2 + 2\phi(u) + \phi(s') + \phi(t') - \phi(s) - \phi(t) \\
 < & \{\phi(u) < \phi(t) \text{ и } \phi(s') \leq \phi(s)\} \\
 & 2 + \phi(u) + \phi(t') \\
 < & \{\#u + \#t' < \#s \text{ и лемма 1}\} \\
 & 1 + 2\phi(s)
 \end{aligned}$$

ЛЕНИВЫЕ ВЫЧИСЛЕНИЯ

LAZY EVALUATION

- **call-by-value** — обычные строгие вычисления
- **call-by-name** — ленивые вычисления без мемоизации
- **call-by-need** — ленивые вычисления с мемоизацией

OCAML

```
let rec fib n =  
  if n < 2 then n  
  else fib (n - 1) + fib (n - 2)  
  
let fibns n =  
  List.map fib (range n)  
  
let lazy_fibns n =  
  Stream.from  
    (fun i -> if i = n then None  
              else Some (fib n))
```

OCAML

```
let rec memo_fib =  
  let cache = Hashtbl.create 10 in  
  fun n ->  
    try Hashtbl.find cache n  
    with Not_found -> begin  
      if n < 2 then n  
      else  
        let f = memo_fib (n-1) + memo_fib (n-2) in  
        Hashtbl.add cache n f; f  
    end  
  
let memo_fibns n =  
  Stream.from  
    (fun i -> if i = n then None  
              else Some (memo_fib n))
```


- `stream_cons` — добавляет элемент в начало потока
- `stream_first` — вывозвращает первый элемент потока
- `stream_rest` — возвращает остаток потока без первого элемента
- `stream_append` — конкатенация потоков
- `stream_rev` — обращает поток, т. е. возвращает элементы в обратном порядке
- `++` - псевдоним для `stream_append`

```
type 'a stream = Nil | Cons of 'a * 'a stream Lazy.t;;

let empty_stream = Nil;;

let stream_cons x s = Cons (x, lazy s);;

let stream_first = function
  | Nil -> failwith "empty stream"
  | Cons (a, _) -> a

let stream_rest = function
  | Nil -> failwith "empty stream"
  | Cons (_, lazy b) -> b

let rec stream_append s1 s2 =
  match s1, s2 with
  | Nil, s -> s
  | Cons (a, lazy b), _ ->
    Cons (a, lazy (stream_append b s2))

let rec stream_rev = function
  | Nil -> Nil
  | Cons (a, lazy b) ->
    stream_append (stream_rev b) (Cons (a, lazy empty_stream))

let (++) = stream_append
```

АМОРТИЗАЦИЯ 2: RELOADED

- Нераздельная стоимость операции — это реальное время, требуемое для выполнения операции, предполагая, что все задержанные вычисления в начале операции были вынуждены и мемоизированы*.
- Разделяемая стоимость — время которое необходимо для выполнения всех задержек, созданных, но не выполненных этой операцией.
- Полная стоимость — сумма нераздельной и разделяемой стоимостей.
- Реализованная — стоимость задержек, которые вынуждаются в процессе полного вычисления
- Нереализованная — стоимость задержек, которые так и остаются невыполненными

ПРИМЕР: ОЧЕРЕДИ

- `stream_rev` — мемоизированная процедура обращения потока
- $q_0 = (m, m)$, $q_i = \text{tail } q_{i-1}$, $i < 0 \leq m + 1$
- Рассмотрим q_k при $k = m$ (после поворота списка) и $k = 0$ (до поворота списка). Повторим $q_k \dots q_{m+1}$ d раз.
- ($k = m, m + 1 + d$ операций)

Так как `rev` мемоизирована, то при каждом вызове вычисления не выполняются заново

- ($k = 0, (m + 1) * (d + 1)$ операций)
rev каждый раз вынуждается заново, но теперь у нас $(m + 1) * (d + 1)$ операций, следовательно...
- Результат в обоих случаях — амортизированное время $O(1)$

OCAML

```
type 'a queue = int * 'a stream * int * 'a stream

let check (lenf, f, lenr, r as q) =
  if lenf <= lenr then q
  else lenf + lenr, f ++ stream_rev r, 0, empty_stream

let append (lenf, f, lenr, r) x =
  check (lenf, f, lenr + 1, stream_cons x r)

let head = function
  | _, Nil, _, _ -> failwith "empty"
  | lenf, Cons (x, _), _, _ -> x

let tail = function
  | _, Nil, _, _ -> failwith "empty"
  | lenf, Cons (x, lazy f'), lenr, r ->
    check (lenf - 1, f', lenr, r)
```

РАСПИСАНИЯ

SCHEDULING

- Иногда нужно добиться быстрой работы функций в худшем случае.
- Например в следующих областях:
- Системы реального времени. Если из-за дорогой операции система пропустит жёсткий предельный срок, неважно будет, сколько дешёвых операций завершилось раньше назначенного времени
- Параллельные системы. Если один процессор в синхронной системе выполняет дорогую операцию в то время, как остальные выполняют дешёвые, то остальным процессорам придётся ждать, пока закончит работу самый медленный.
- Диалоговые системы. Например, пользователи могут предпочесть 100 ответов с задержкой 1 секунда варианту с 99 ответами при задержке 0.25 секунд и одним ответом с задержкой 25 секунд, даже при том, что второй из этих сценариев вдвое быстрее.

ПРИМЕР: ОЧЕРЕДИ РЕАЛЬНОГО ВРЕМЕНИ

OCAML

```
type 'a queue = 'a stream * 'a list * 'a stream;;

let rec rotate = function
  | Nil, y :: _, a -> Cons (y, lazy a)
  | Cons (x, lazy xs), y :: ys, a ->
    Cons (x, lazy (rotate (xs, ys, Cons (y, lazy a))))

let exec = function
  | f, r, Cons (_, lazy s) -> f, r, s
  | f, r, Nil ->
    let f' = rotate f, r, empty_stream in
      f', [], f'

let append (f, r, s) x = exec (f, x :: r, s);;

let head = function
  | Nil, _, _ -> failwith "empty"
  | Cons (x, lazy f), _, _ -> x

let tail = function
  | Nil, _, _ -> failwith "empty"
  | Cons (x, lazy f), r, s -> exec (f, r, s)
```

...

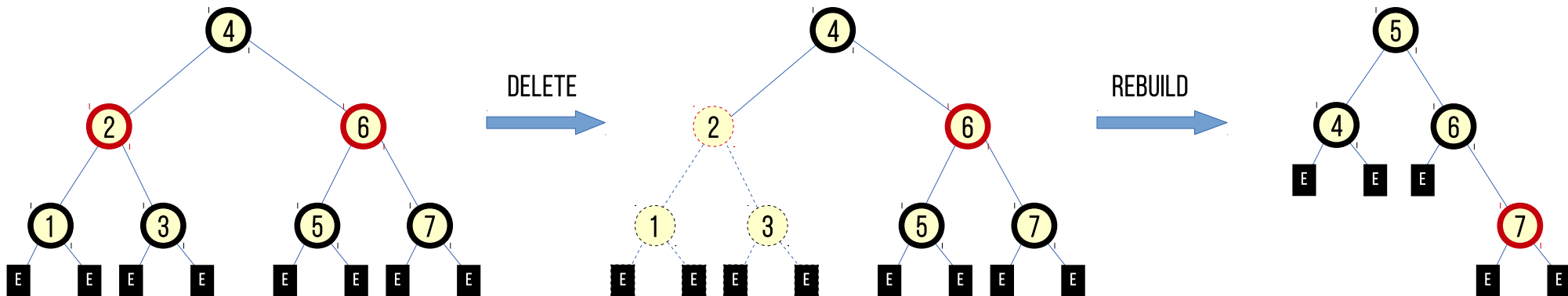
МЕТОДИКИ ПРОЕКТИРОВАНИЯ

- Порционная перестройка
- Глобальная перестройка
- Ленивая перестройка (вариация глобальной)
- Числовые представления
- Развёртка структур данных

ПОРЦИОННАЯ ПЕРЕСТРОЙКА

BATCHED REBUILDING

- Если мы хотим достичь границы $O(f(n))$ на каждую операцию, а для перестроения требуется $O(g(n))$ времени, то перебалансировка не может быть запущена чаще чем через каждые $c * g(n) / f(n)$ операции (c — константный множитель).
- Перестройка красно-чёрного дерева занимает время $O(n)$, чтобы получить амортизированную стоимость $O(\log n)$ нужно запускать перестройку не чаще чем каждые $c * n / \log n$ операции обновления.



ГЛОБАЛЬНАЯ ПЕРЕСТРОЙКА

GLOBAL REBUILDING

- Основная идея - производить трансформацию постепенно, по нескольку шагов при каждой операции.
- Поддерживаются две копии объекта. Первичная (рабочая) копия — это исходная структура, вторичная — та, которая перестраивается

OCAML

```
...

let exec = function
  | Reversing (ok, x :: f, f', y :: r, r') ->
    Reversing (ok + 1, f, x :: f', r, y :: r')
  | Reversing (ok, [], f', [y], r') ->
    Appending (ok, f', y :: r')
  | Appending (0, f', r') -> Done (r')
  | Appending (ok, x :: f', r') ->
    Appending (ok - 1, f', x :: r')
  | _ as state -> state

...
```

OCAML

```
type 'a rotation_state =
  Idle
  | Reversing of int * 'a list * 'a list * 'a list * 'a list
  | Appending of int * 'a list * 'a list
  | Done of 'a list

type 'a queue = int * 'a list * 'a rotation_state * int * 'a list

...

Empty => (0, [], Idle, 0, [])
1 => (1, [1], Idle, 0, [])
2 => (1, [1], Idle, 1, [2])
3 => (3, [1], Appending (1, [1], [2; 3]) 0 [])
4 => (3, [1; 2; 3], Idle, 1, [4])
5 => (3, [1; 2; 3], Idle, 2, [5; 4])
6 => (3, [1; 2; 3], Idle, 3, [6; 5; 4])
7 => (7, [1; 2; 3], Reversing (2, [3], [2; 1], [5; 4], [6; 7]) 0 [])
8 => (7, [1; 2; 3], Appending (3, [3; 2; 1], [4; 5; 6; 7]), 1, [8])
9 => (7, [1; 2; 3], Appending (1, [1], [2; 3; 4; 5; 6; 7]), 2 [9; 8])
10 => (7, [1; 2; 3; 4; 5; 6; 7], Idle, 3, [10; 9; 8])
```


ЛЕНИВАЯ ПЕРЕСТРОЙКА

LAZY REBUILDING

OCAML

```
type 'a queue = 'a stream * 'a list * 'a stream;;

let rec rotate = function
| Nil, y :: _, a -> Cons (y, lazy a)
| Cons (x, lazy xs), y :: ys, a ->
  Cons (x, lazy (rotate (xs, ys, Cons (y, lazy a))))

let exec = function
| f, r, Cons (_, lazy s) -> f, r, s
| f, r, Nil ->
  let f' = rotate f, r, empty_stream in
  f', [], f'

let append (f, r, s) x = exec (f, x :: r, s);;

let head = function
| Nil, _, _ -> failwith "empty"
| Cons (x, lazy f), _, _ -> x

let tail = function
| Nil, _, _ -> failwith "empty"
| Cons (x, lazy f), r, s -> exec (f, r, s)
```

ЧИСЛОВЫЕ ПРЕДСТАВЛЕНИЯ

NUMERICAL REPRESENTATIONS

OCAML

Двоичная система счисления

$A = \{0, 1\}$, $w_i = 2^i$ — вес разряда

$$1011_2 = 2^0 * 1 + 2^1 * 0 + 2^2 * 1 + 2^3 * 1$$

Избыточная двоичная система счисления

$A = \{0, 1, 2\}$, $w_i = 2^i$ — вес разряда

$$1201_2 = 2^0 * 1 + 2^1 * 2 + 2^2 * 0 + 2^3 * 1$$

```
(* natural numbers *)
type nat = Zero | Succ of nat

let pred = function
  | Succ n -> n
  | _ -> failwith "zero"

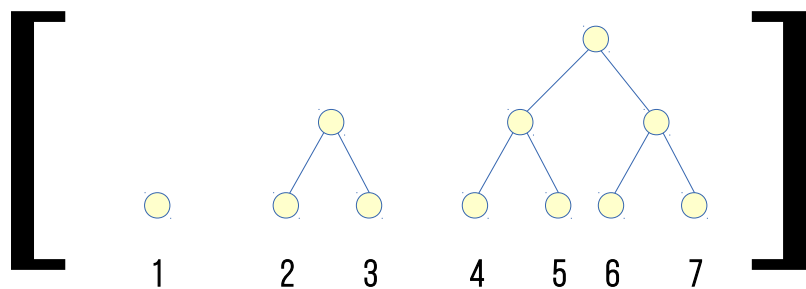
let rec plus m n =
  match m with
  | Zero -> n
  | Succ x -> Succ (plus x n)

(* lists *)
let tail = function
  | x :: xs -> xs
  | _ -> failwith "empty"

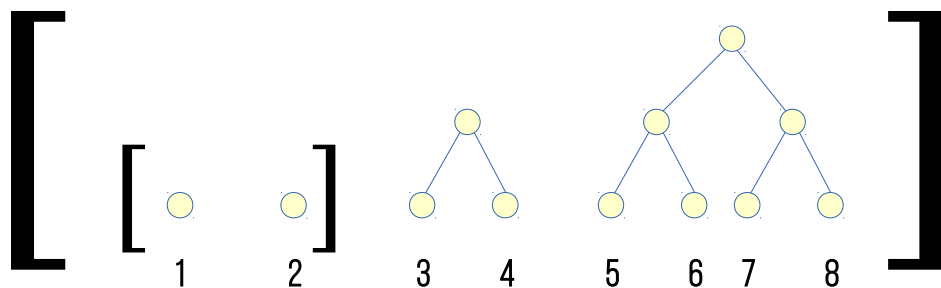
let rec append l1 l2 =
  match l1 with
  | [] -> l2
  | x :: xs -> x :: (append xs l2)
```

СПИСКИ С ПРОИЗВОЛЬНЫМ ДОСТУПОМ

RANDOM-ACCESS LISTS



- Cons, head, tail — $O(\log n)$
- Lookup, update — $O(\log n)$



- Head, tail - $O(1)$
- Cons — $O(\log n)$
- Lookup, update — $O(\log n)$

$$211_2 = 0001_2$$

СПИСКИ С ПРОИЗВОЛЬНЫМ ДОСТУПОМ

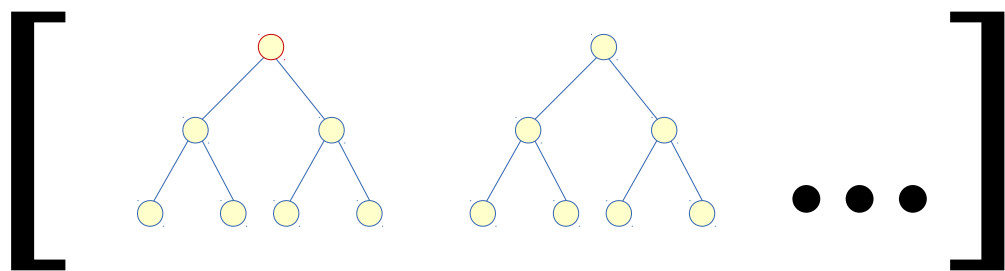
ПРОДОЛЖЕНИЕ

Скошенные двоичные числа

$A = \{0, 1, 2\}$, вес разряда $w_i = 2^{i+1} - 1$

$$002101_2 = (2^{0+1} - 1) * 0 + (2^{1+1} - 1) * 0 + (2^{2+1} - 1) * 2 + (2^{3+1} - 1) * 1 + (2^{4+1} - 1) * 0 + (2^{5+1} - 1) * 1 = 92$$

Канонический вид: только первая ненулевая цифра = 2



- Head, tail - $O(1)$
- Cons — $O(1)$
- Lookup, update — $O(\log n)$

002101₂

РАЗВЁРТКА СТРУКТУР ДАННЫХ

STRUCTURAL BOOTSTRAPING

OCAML

```
(* гомогенно рекурсивный тип *)
type 'a list = Nil | Cons of 'a * 'a list

(* гетерогенно рекурсивный тип *)
type 'a seq = Nil' | Cons' of 'a * ('a * 'a) seq

(* время работы -  $O(n)$  *)
let rec sizeL = function
  | Nil -> 0
  | Cons (_, xs) -> 1 + sizeL xs

(* время работы  $O(\log n)$  *)
let rec sizeS : 'a. 'a seq -> int = function
  | Nil' -> 0
  | Cons' (_, xs) -> 1 + 2 * sizeS xs
```

- Структурная декомпозиция — развёртка полных структур данных из неполных

СТРУКТУРНАЯ ДЕКОМПОЗИЦИЯ

OCAML

```
(* Списки с произвольным доступом *)
type 'a seq = Nil | Zero of ('a * 'a) seq | One of 'a * ('a * 'a) seq

let rec cons : 'a . 'a -> 'a seq -> 'a seq =
  fun x s -> match s with
  | Nil -> One(x, Nil)
  | Zero ps -> One(x, ps)
  | One (y, ys) -> Zero(cons (x, y) ys);;

let rec uncons : 'a . 'a seq -> 'a * 'a seq = function
  | One(x, Nil) -> x, Nil
  | One(x, xs) -> x, Zero xs
  | Zero xs -> let (x, y), xs' = uncons xs in x, One(y, xs')
  | Nil -> failwith "empty"

let head s = let x, _ = uncons s in x

let tail s = let _, xs = uncons s in xs

(* lookup 4 (One(1, One((2, 3), One(((4, 5), (6, 7))), Nil)))) *)
let rec lookup : 'a. int -> 'a seq -> 'a =
  fun idx s -> match idx, s with
  | _, Nil -> failwith "subscript"
  | 0, One (x, xs) -> x
  | i, One (x, xs) -> lookup (i - 1) (Zero xs)
  | i, Zero xs -> let x, y = lookup (i / 2) xs in
    if i mod 2 == 0 then x else y
```

СТРУКТУРНАЯ АБСТРАКЦИЯ

- Имеем функцию $\text{insert} : 'a \rightarrow 'a\ C \rightarrow 'a\ C$, нужно реализовать $\text{insertB} : 'a \rightarrow 'a\ B \rightarrow 'a\ B$, $\text{joinB} : 'a\ B \rightarrow 'a\ B \rightarrow 'a\ B$, где $'a\ B$ - развёрнутый тип. Также должна быть определена функция unit для конструирования коллекции с одним элементом.
- insert и join могут быть представлены абстрактно:

$$\begin{aligned}\text{insertB}\ (x, b) &= \text{join}\ (\text{unitB}\ x, b) \\ \text{joinB}\ (b1, b2) &= \text{insert}\ (b1, b2)\end{aligned}$$

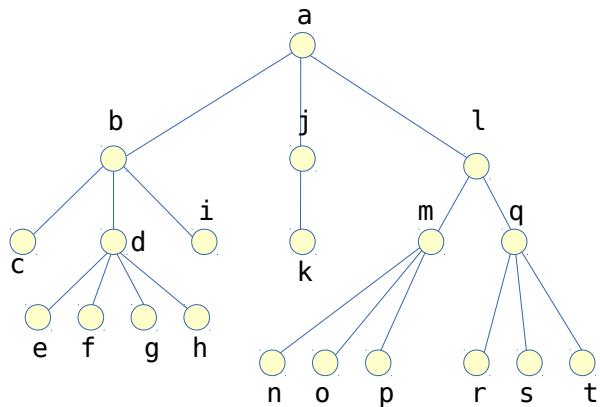
- Если тип коллекции $\text{type}\ 'a\ B = E \mid B\ \text{of}\ 'a * ('a\ B)\ C$, то шаблоны insert и join следующие

$$\begin{aligned}\text{insertB}\ (x, E) &= B\ (x, \text{empty}) \\ \mid \text{insertB}\ (x, B\ (y, e)) &= B\ (x, \text{insert}\ (\text{unitB}\ y, e))\end{aligned}$$
$$\begin{aligned}\text{join}\ B\ (b, E) &= b \\ \mid \text{join}\ B\ (E, b) &= b \\ \mid \text{join}\ B\ (B\ (x, e), b) &= B\ (x, \text{insert}\ (b, e))\end{aligned}$$

- Реализовать deleteB для извлечение особого элемента (первого или наименьшего)

СТРУКТУРНАЯ АБСТРАКЦИЯ

OCAML



head — $O(1)$ в худшем случае

++ и tail — $O(1)$ амортизированное.

```
(* Списки с эффективной конкатенацией *)
(* Q — какой-либо модуль с реализацией FIFO очередей *)
type 'a cat = Empty | Cat of 'a * 'a cat Lazy.t Q.queue

let link t s =
  match t with
  | Cat (x, q) -> Cat (x, Q.append q s)
  | Empty -> Empty

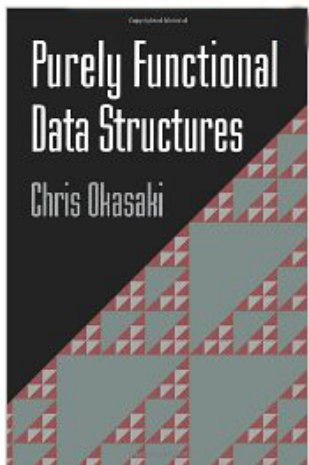
let rec linkAll q =
  let (lazy t) = Q.head q in
  let q' = Q.tail q in
  if Q.isEmpty q then t
  else link t (lazy (linkAll q'))

let (++) l1 l2 =
  match l1, l2 with
  | l, Empty -> l
  | Empty, l -> l
  | a, b -> link a (lazy b)

let cons x xs = Cat(x, Q.empty) ++ xs
let append xs x = xs ++ Cat(x, Q.empty)

let head = function
  | Empty -> failwith "empty"
  | Cat(x, _) -> x

let tail = function
  | Empty -> failwith "empty"
  | Cat(x, q) -> if Q.isEmpty q then Empty else linkAll q
```

Название: Purely Functional Data Structures

Автор: Chris Okasaki

Издательство: Cambridge University Press (June 13, 1999)

Перевод: <https://github.com/gogabr/pfds/>

<https://github.com/whiterabbit1983/ds>

Название: Introduction to Algorithms

Автор: Thomas H. Cormen et al.

Издательство: The MIT Press; third edition (July 31, 2009)

