Язык C. Введение в алгоритмы и структуры данных.

Лекция 12
Деревья

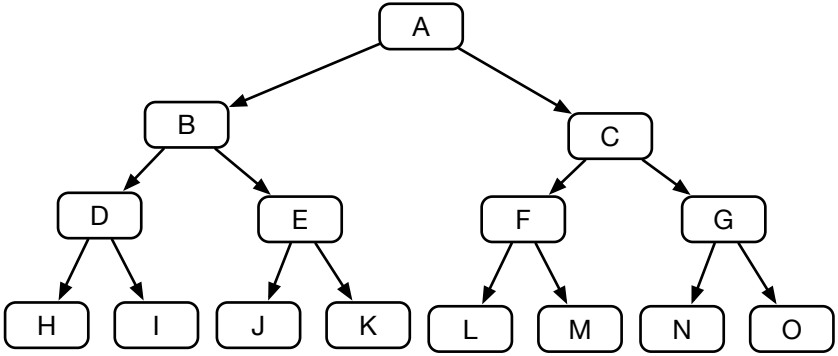
Сергей Леонидович Бабичев

Полные бинарные деревья

Определение:

- **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 являющимися листьями, есть и правый, и левый потомок.

Полное бинарное дерево высоты 3.



Абстракция *приоритетная очередь*

Приоритетная очередь

- Приоритетная очередь (`priority queue`) — очередь, элементы которой сравнимы и имеют приоритет.
- После вставки элемента очередь остаётся в упорядоченном по приоритету состоянии.
- Первым извлекается наиболее приоритетный элемент (максимум или минимум).

Интерфейс абстракции:

- `insert` — добавление элемента в очередь;
- `fetch` — получает самый приоритетный элемент, не извлекая его.
- `erase` — удаляет самый приоритетный элемент;

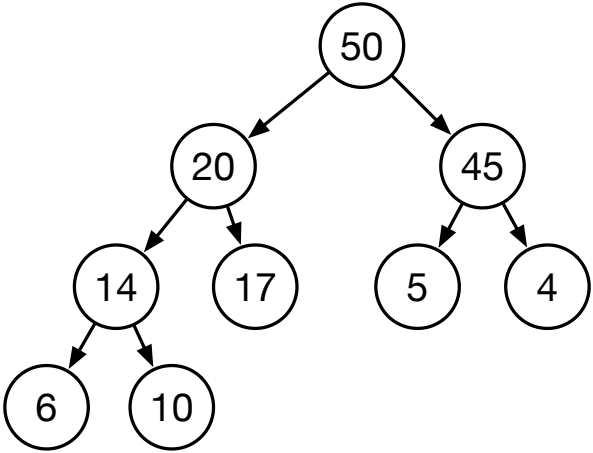
Приоритетная очередь:представление

- *Бинарная куча* — бинарное дерево, удовлетворяющее условиям:
  - Любая вершина не менее приоритетна, чем потомки (упорядоченность по данным).
  - Дерево является правильным подмножеством полного бинарного (структурная упорядоченность).

Другое название — пирамида (`heap`).

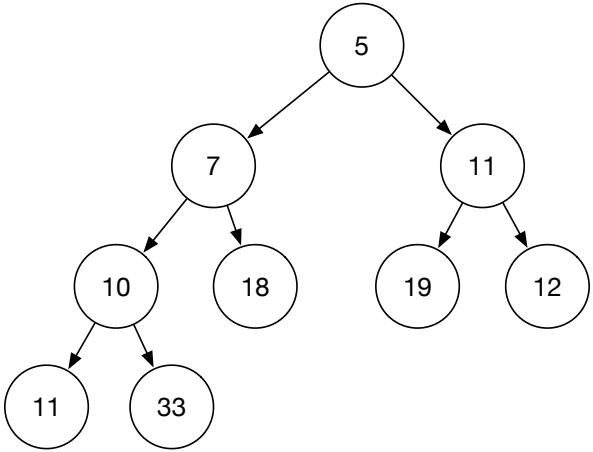
Приоритетная очередь

- Невозрастающая пирамида

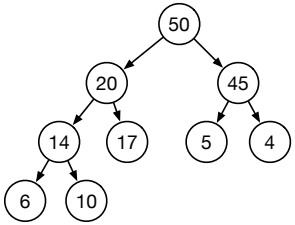


Приоритетная очередь

- Неубывающая пирамида



Приоритетная очередь:реализация



Хранение в виде массива с индексами от 1 до N:

50	20	45	14	17	5	4	6	10

- Индекс корня дерева всегда равен 1 — максимальный (минимальный) элемент
- Индекс родителя узла i равен $\lfloor \frac{i}{2} \rfloor$
- Индекс левого потомка узла i равен $2i$
- Индекс правого потомка узле i равен $2i + 1$

Приоритетная очередь: реализация на базе массива.

```
typedef struct bhnode_s { // node
    int priority;
} bhnode;

typedef struct binary_heap_s {
    bhnode *body;
    int     bodysize;
    int     numnodes;
} binary_heap;
```

Создание бинарной кучи

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef int bhnode;

typedef struct binary_heap_s {
    bhnode *body;
    int     bodysize;
    int     numnodes;
} binary_heap;

binary_heap *binary_heap_new(int maxsize) {
    binary_heap *bh = malloc(sizeof (binary_heap));
    bh->body = malloc(sizeof(bhnode) * (maxsize+1));
    bh->bodysize = maxsize;
    bh->numnodes = 0;
    return bh;
}
```

Бинарная куча: поиск минимума(максимума)

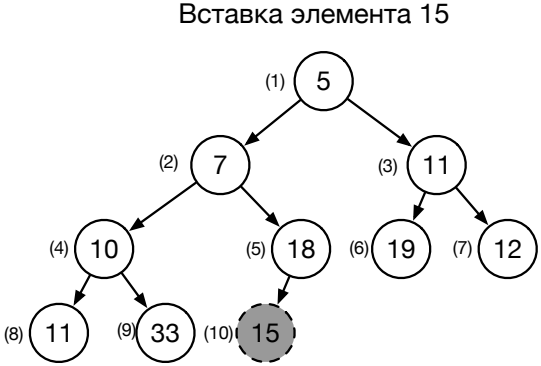
```
void binary_heap_destroy(binary_heap *bh) {
    free(bh->body);
    free(bh);
}

void binary_heap_swap(binary_heap *bh, int a, int b) {
    bhnode tmp = bh->body[a];
    bh->body[a] = bh->body[b];
    bh->body[b] = tmp;
}

bhnode binary_heap_fetch(binary_heap *bh) {
    assert(bh->numnodes > 0);
    return bh->body[1];
}
```

Бинарная куча: вставка элемента

- Этап 1. Вставка в конец кучи.

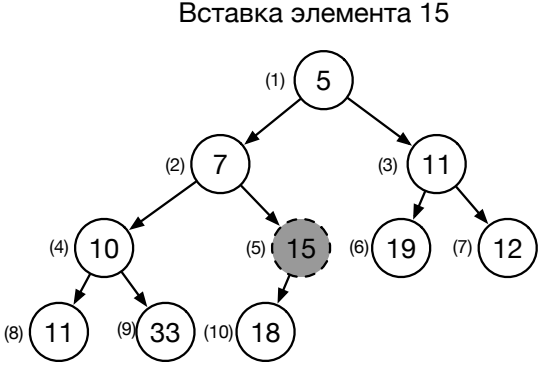


Отлично! *Структура* кучи не испортилась!

50	45	20	17	14	10	6	5	4	33

Бинарная куча: вставка элемента

- Этап 2. Корректировка значений.

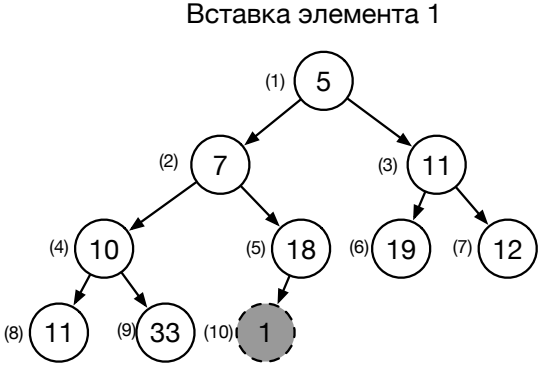


Куча удовлетворяет всем условиям.

50	45	20	17	33	10	6	5	4	14

Бинарная куча: вставка элемента

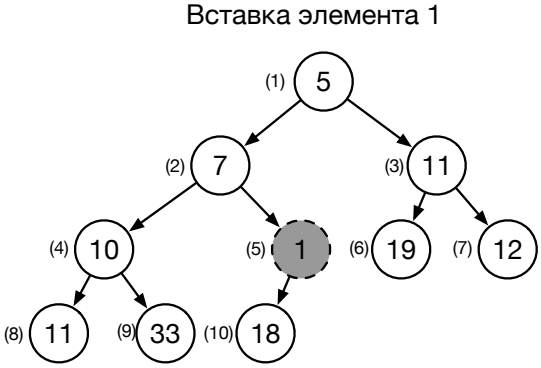
- Попытаемся вставить максимальный элемент.



50	45	20	17	14	10	6	5	4	70

Бинарная куча: вставка элемента

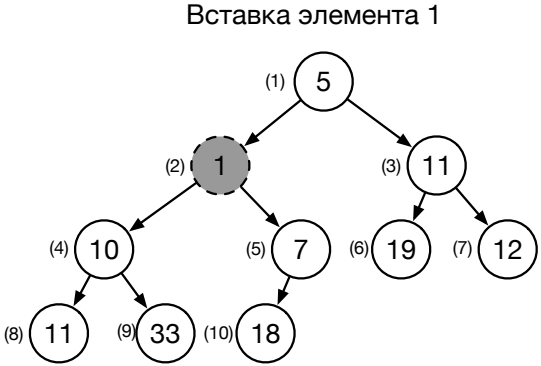
- Максимальный элемент ползёт вверх по дереву.



50	45	20	17	70	10	6	5	4	14

Бинарная куча: вставка элемента

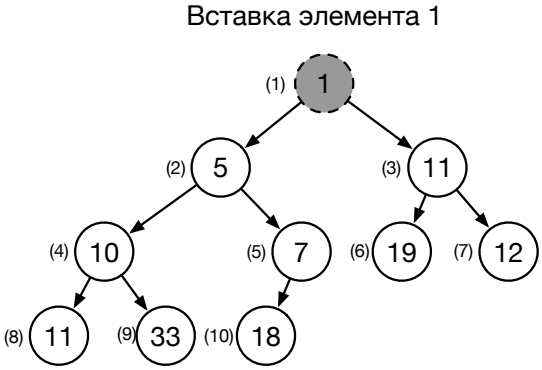
- Максимальный элемент ползёт вверх по дереву.



50	70	20	17	45	10	6	5	4	14

Бинарная куча: вставка элемента

- Максимальный элемент ползёт вверх по дереву.



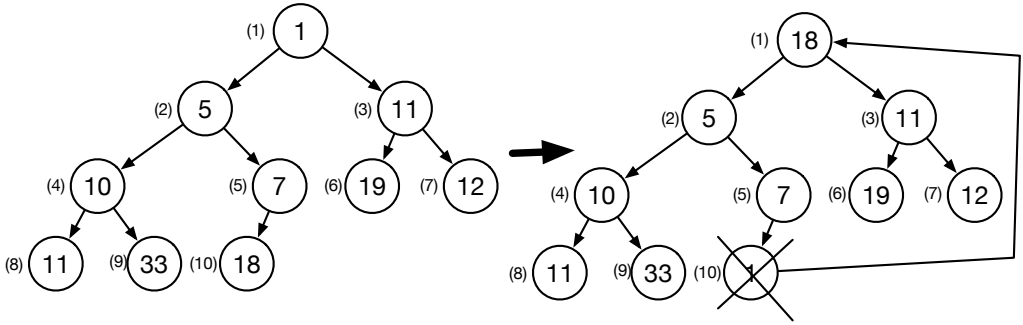
70	50	20	17	45	10	6	5	4	14

Бинарная куча: вставка элемента

```
int binary_heap_insert(binary_heap *bh, bhnode node) {
    if (bh->numnodes > bh->bodysize) {
        return -1; // or expand
    }
    bh->body[++bh->numnodes] = node;
    for (size_t i = bh->numnodes;
       i > 1 && bh->body[i] > bh->body[i/2];
       i /= 2) {
        binary_heap_swap(bh, i, i/2);
    }
    return 0;
}
```

$$T_{Insert} = O(\log N)$$

Бинарная куча: удаление максимального (минимального)



Свойства кучи нарушены. Требуется восстановление.

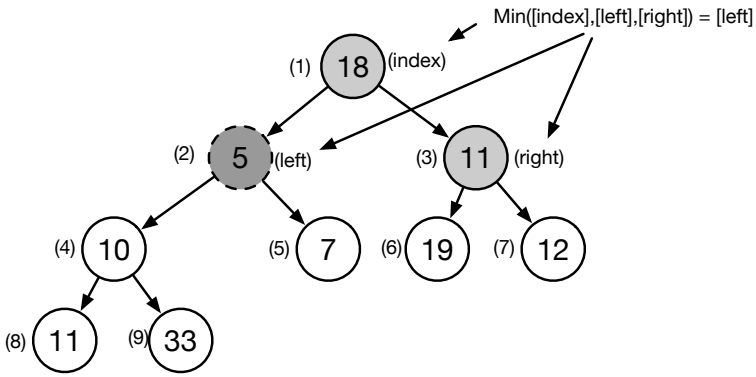
Бинарная куча: удаление вершины

```
void binary_heap_erase(binary_heap *bh) {
    assert(bh->numnodes > 0);
    bh->body[1] = bh->body[--bh->numnodes];
    size_t index = 1;
    for (;;) {
        size_t left = index + index, right = left + 1;
        // Who is greater, [index], [left], [right]?
        size_t largest = index;
        if (left <= bh->numnodes && bh->body[left] > bh->body[index])
        largest = left;
        if (right <= bh->numnodes && bh->body[right] > bh->body[largest])
        largest = right;
        if (largest == index) break;
        binary_heap_swap(bh, index, largest);
        index = largest;
    }
}
```

$$T_{erase} = O(\log N)$$

Бинарная куча: восстановление свойств

- Восстановление индекса (1)

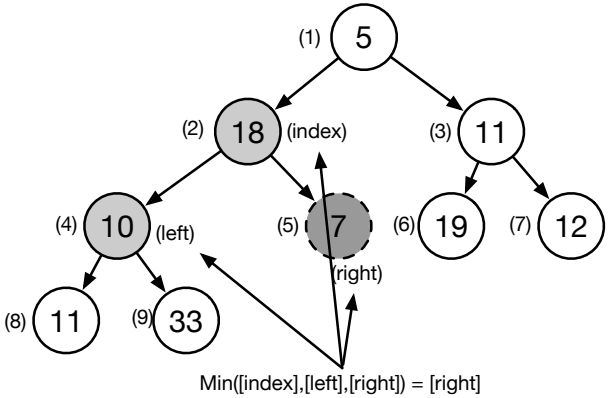


14	50	20	17	45	10	6	5	4

Новый индекс для восстановления (2)

Бинарная куча: восстановление свойств

- Восстановление индекса (2)

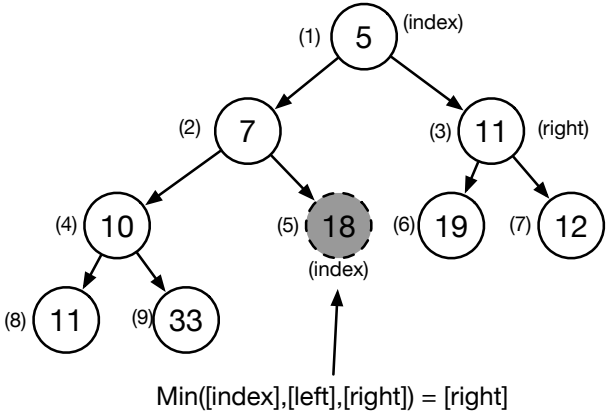


50	14	20	17	45	10	6	5	4

Новый индекс для восстановления (5)

Бинарная куча: восстановление свойств

- Восстановление индекса (5)



50	45	20	17	14	10	6	5	4

Восстановление закончено.

Бинарная куча: сложность операций

- **insert** — $O(\log N)$
- **erase** — $O(\log N)$
- **fetch** — $O(1)$

Деревья поиска

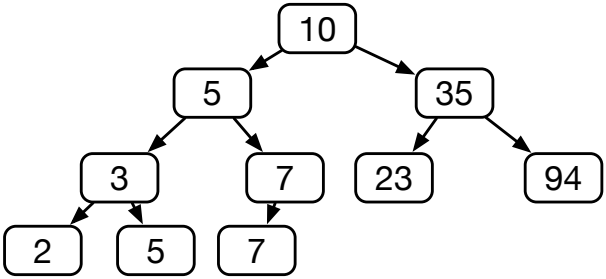
Деревья: поиск

Использование деревьев для поиска.
Задача:

- Вход: последовательность чисел.
- Выход: 2-дерево, в котором все узлы справа от родителя больше родителя, а слева — не больше.

Деревья: поиск

{10, 5, 35, 7, 3, 23, 94, 2, 5, 7}



Деревья: поиск

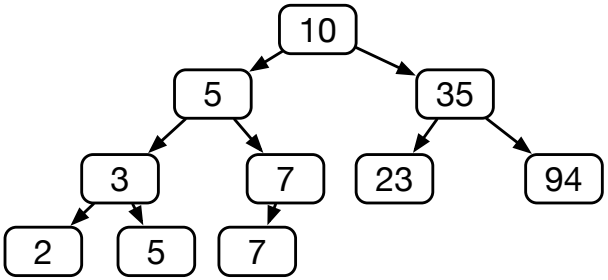
Поиск по дереву после получения элемента с ключом X:

1. Делаем текущий узел корневым
2. Переходим в текущий узел C.
3. Если $X = C.Key$ то алгоритм завершён.
4. Если $X > C.Key$ и C имеет потомка справа, то делаем текущим узлом потомка справа. Переходим к п. 2.
5. Если $X < C.Key$ и C имеет потомка слева, то делаем текущим узлом потомка слева. Переходим к п. 2.
6. Ключ не найден. Конец алгоритма.

Бинарные деревья поиска

Наивное построение бинарных деревьев поиска.
Неплохое дерево

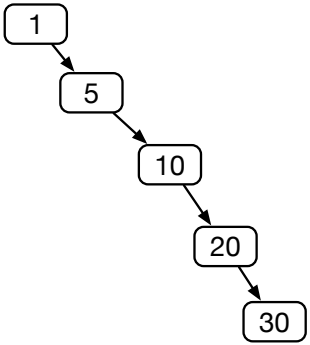
{10, 5, 35, 7, 3, 23, 94, 2, 5, 7}



Бинарные деревья поиска

Отвратительное дерево (бамбук)

{1, 5, 10, 20, 30}



Бинарные деревья поиска

Определение:

- **Случайное бинарное дерево** T размера n — дерево, получающееся из пустого бинарного дерева поиска после добавления в него n узлов с различными ключами в случайном порядке и все $n!$ возможных последовательностей добавления равновероятны.
- **Средняя** глубина узлов случайного бинарного дерева есть $O(\log_2 N)$.
- **Средние** времена выполнения операций вставки, удаления и поиска в случайном бинарном дереве есть $O(\log_2 N)$.

Бинарные деревья поиска: свойства

Полезные свойства бинарного дерева поиска:

- Наименьший элемент всегда находится в самом низу левого поддерева.
- Наибольший элемент всегда находится в самом низу правого поддерева.

```
tree * minNode(tree *t) {
    if (t == NULL) return NULL;
    while (t->left != NULL) {
        t = t->left;
    }
    return t;
}
```

Бинарные деревья поиска: свойства

- Простая процедура поиска

```
tree * searchNode(tree *t, keytype key) {
    tree *p = t;
    while (t != NULL) {
        p = t;
        if (t->key == key) return t;
        t = key > t->key? t->right : t->left;
    }
    return p;
}
```

Бинарные деревья поиска: свойства

- Простая процедура вставки нового.

```
void insertNode(tree *t, keytype key, valtype value) {
    tree *parent = t;
    while (t != NULL) {
        parent = t;
        if (t->key == key) return; // Already here
        t = key > t->key? t->right : t->left;
    }
    tree *node = new tree(key, value);
    if (key < parent->key) parent->left  = node;
    else                   parent->right = node;
}
```

Бинарные деревья поиска: свойства

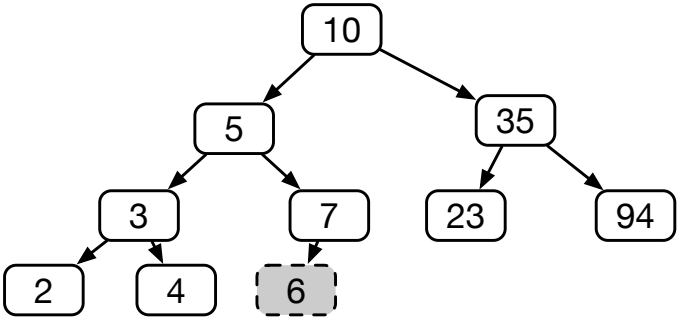
- Процедура удаления сложнее, три случая:
  1. Нет потомков — удаляем узел у родителя.
  2. Один потомок — переставляем узел у родителя на потомка

Бинарные деревья поиска: свойства

- Процедура удаления сложнее, три случая:
  1. Нет потомков — удаляем узел у родителя.
  2. Один потомок — переставляем узел у родителя на потомка
  3. Два потомка — находим самый левый лист в правом поддереве и им замещаем удаляемый

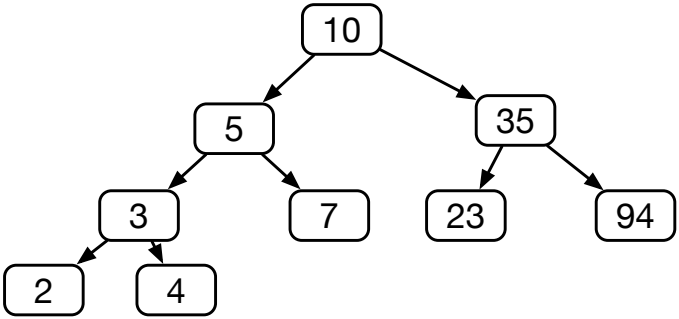
Бинарные деревья поиска: свойства

Первый случай: до удаления



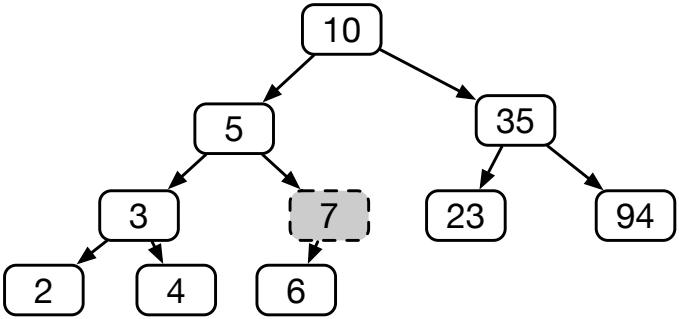
Бинарные деревья поиска: свойства

Первый случай: после удаления



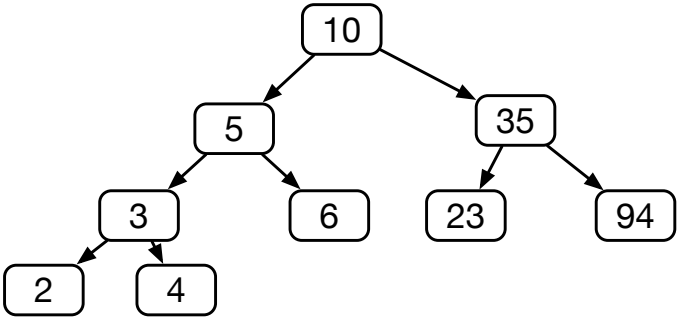
Бинарные деревья поиска: свойства

Второй случай: до удаления



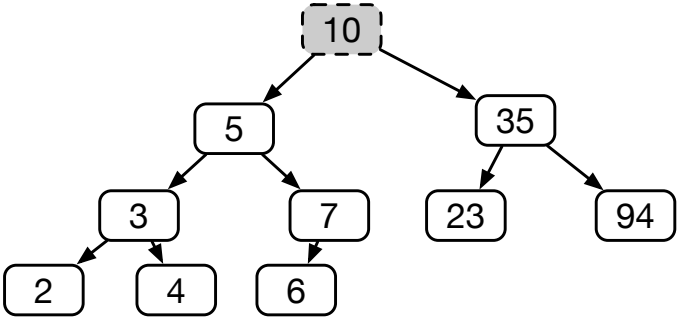
Бинарные деревья поиска: свойства

Второй случай: после удаления



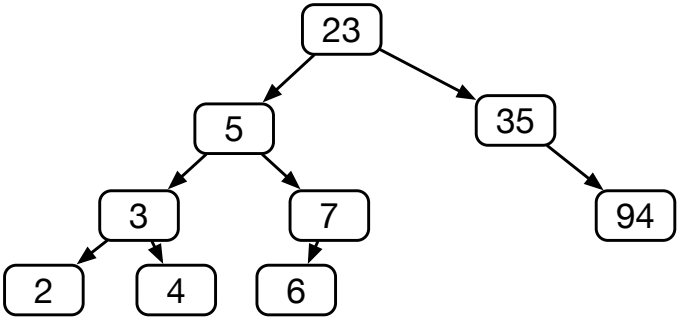
Бинарные деревья поиска: свойства

Третий случай: до удаления



Бинарные деревья поиска: свойства

Третий случай: после удаления



Бинарные деревья поиска

Структура хранилища	вставка	удаление	поиск
Бинарное дерево поиска (наихудшее)	$O(N)$	$O(N)$	$O(N)$
Бинарное дерево поиска (среднее)	$O(\log N)$	$O(\log N)$	$O(\log N)$

Борьба с дисбалансом

1. Сложность всех алгоритмов в бинарных деревьях поиска (BST) определяется средневзвешенной глубиной
2. Операции вставки/удаления могут привести к дисбалансу и ухудшению средних показателей
3. Для борьбы с дисбалансом применяют рандомизацию и балансировку.

Борьба с дисбалансом

Попытка:

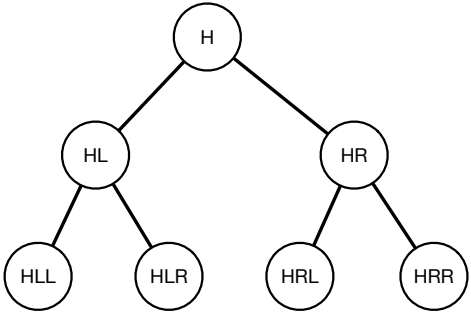
- Предлагается: вставлять новые элементы всегда в корень.
- Последствия: если вставляемый элемент больше корня, то старый корень сделаем левым поддеревом, а его правое поддерево — нашим правым поддеревом.
- Аналогично рассуждаем для случая, когда вставляемый элемент меньше корня.
- Упорядоченность может нарушиться в обоих случаях.

Борьба с дисбалансом

- Чтобы нарушений не происходило, требуется сохранять инвариант упорядоченности.
- Для этого введём понятие поворота, не изменяющего свойства дерева, но меняющего высоту поддеревьев.

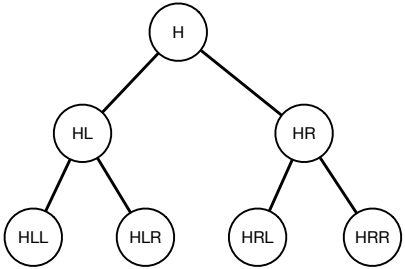
Повороты дерева

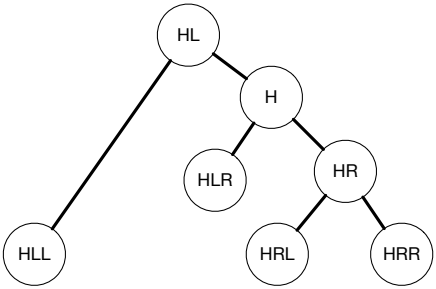
Перед поворотом



Повороты дерева

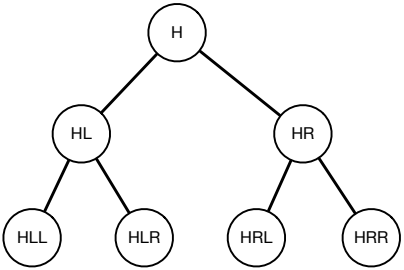
После поворота направо

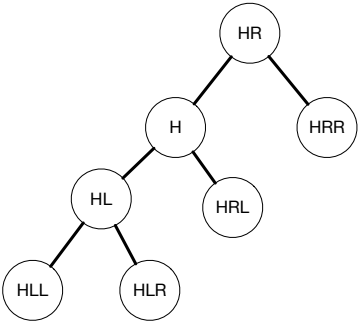




Повороты дерева

После поворота налево





Эксперименты с деревом

```
#include <stdio.h>
#include <stdlib.h>

typedef double item;

typedef struct node_s {
    struct node_s *left, *right;
    item val;
} node;

node *new_node(item it) {
    node *ret = (node *)malloc(sizeof (node));
    ret->left = ret->right = NULL;
    ret->val = it;
    return ret;
}
```

Повороты дерева

```
node *rotateRight(node* head) {
    node *temp = head->left;
    head->left = temp->right;
    temp->right = head;
    return temp;
}

node *rotateLeft(node* head) {
    node *temp = head->right;
    head->right = temp->left;
    temp->left = head;
    return temp;
}
```

Вставка в корневой узел

Рекурсивный алгоритм.

```
node * insert(node* head, item x) {
    if (head == NULL) {
        head = new_node(x);
        return head;
    }
    if (x < head->val) {
        head->left = insert(head->left, x);
        return rotateRight(head);
    } else {
        head->right = insert(head->right, x);
        return rotateLeft(head);
    }
}
```

```
void print(node const *head, int lev) {
    if (head->right != NULL)  print(head->right, lev+1);
    for (int i = 0; i < lev; i++) {
        printf("  ");
    }
    printf("%.2f\n", head->val);
    if (head->left != NULL)  print(head->left, lev+1);
}

int main() {
    node *head = NULL;
    head = insert(head, 2);
    head = insert(head, 1);
    head = insert(head, 3);
    print(head, 0);
}
```

Рандомизированное дерево

- Проблема вырождения дерева при вставке в корень не решена.
- Однако появилась инфраструктура для достижения меньшей сложности.
- С вероятностью $\frac{1}{N+1}$ вставляем новый узел в корень дерева размером N.
- Свойства любого дерева будут соответствовать свойствам случайного дерева.