

Сбалансированные и специальные деревья

Лекция 5

План лекции

- ❶ Интерфейс абстракции *отображение*.
- ❷ Деревья поиска.
- ❸ Декартовы деревья.
- ❹ Сбалансированные деревья поиска.
 - ▶ Красно-чёрные деревья.
 - ▶ AVL-деревья.
- ❺ Списки с пропусками.
- ❻ Внешний поиск. В-деревья.

Интерфейс абстракции *отображение.*

Абстракция отображение

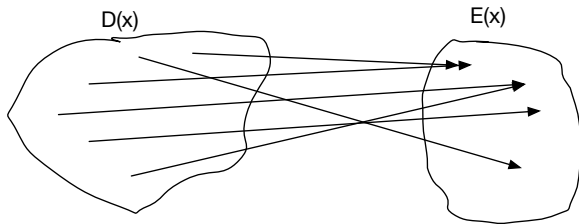
- Абстракция *отображение* устанавливает соответствие между двумя множествами — множеством ключей и множеством данных.



Абстракция отображение

- Абстракция *отображение* есть аналог дискретной функции.
- Одно из определений математической функции: **Функция есть отображение множества D на множество E .**

$F(x)$



Отображение как полезная структура данных

- Разновидность отображения — *таблица символов, словарь*
- Цель словаря — удобная реализация операций вставки и поиска.
- В обычном словаре ключи — словарные входы, данные — словарные статьи.
- Банк: ключ — номер счёта, данные — информация о счёте.

Абстракция отображение

- Самый удобный способ создать отображение — воспользоваться синтаксисом индексации.

```
map<string,int> m;  
m["Shanghai"] = 24150000;  
m["Karachi"] = 23500000;  
m["Beijing"] = 21150000;  
m["Delhi"] = 17830000;  
...  
int BeijingPopulation = m["Beijing"];  
...  
for (auto x: m) {  
    printf("Population of '%s' is %d\n",  
        x.first, x.second);  
}
```

Абстракция отображение

Интерфейс абстракции отображение

- ***insert(key, value)*** — добавить элемент с ключом *key* и значением *value*
- ***Item find(key)*** — найти элемент с ключом *key* и вернуть его.
- ***erase(key)*** — удалить элемент с ключом *key*
- ***walk*** — получить все ключи (или все пары ключ/значение) в каком-либо порядке.

Абстракция отображение: C++

Интерфейс абстракции *отображение*

- *insert(key, value)* — `m[key] = value;`
- *Item find(key)* — `auto val = m[key];`
или `auto r = m.find(key); if (r != m.end()) { found }`
- *erase(key)* — `m.erase(key);`
- *walk* — `for (auto q: m) { use q.first, q.second; }`

Абстракция отображение

Цели:

- Реализовать операции, исполняющиеся минимальное время:
 - ▶ Вставки
 - ▶ Замены
 - ▶ Удаления
 - ▶ Поиска
 - ▶ Перечисления

В дальнейшем под термином **ключ** мы понимаем пару **ключ+значение**, в которой определена операция сравнения по ключу.

Связь множества и отображения

- Возможная реализация отображения — множество с прикреплёнными данными.
- Каждое представление множества, кроме битовой карты, расширяется на отображение.
- С другой стороны — множество есть отображение множества ключей на логическую истину.
- Наиболее универсальное представление и множеств, и отображений — бинарное дерево поиска.

Деревья поиска

Деревья: поиск

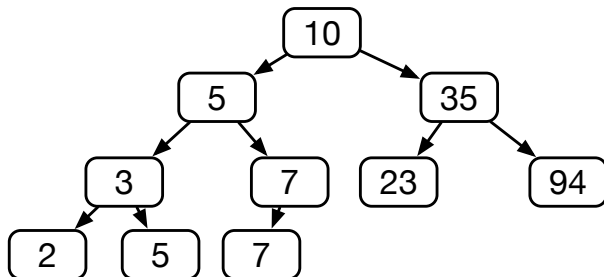
Использование деревьев для поиска.

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

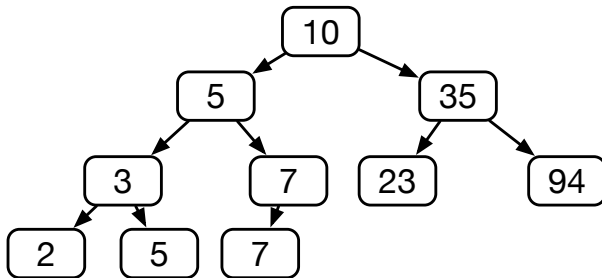
Поиск по дереву после получения элемента с ключом X :

- 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 построением бинарных деревьев поиска.

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

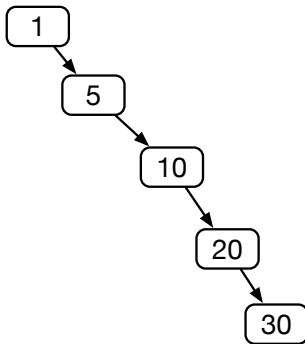


Неплохое дерево

Бинарные деревья поиска

Отвратительное дерево

{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.

Бинарные деревья поиска

Определение средней глубины случайного дерева.

- Пусть $\bar{d}(N+1)$ — средняя глубина всех узлов случайного дерева с $N+1$ узлами.
- Пусть k — узел, добавленный первым. Вероятность добавления узла k есть $p_k = \frac{1}{N+1}$
- Остальные узлы разобьются на группы, каждая из которых начнётся с высоты 1. В левую группу войдут элементы $\{0, \dots, k-1\}$, в правую — $\{k+1, \dots, N\}$.

$$\bar{d}(N+1) = \sum_{k=0}^N \frac{1}{N+1} \left(1 + \frac{k}{N} \cdot \bar{d}(k) + \frac{N-k}{N} \cdot \bar{d}(N-k) \right)$$

Бинарные деревья поиска

$$\bar{d}(N+1) = \frac{2}{N(N+1)} \sum_{k=0}^N k \cdot \bar{d}(k)$$

Используя предел

$$\lim_{n \rightarrow \infty} \left(\sum_{k=1}^n \frac{1}{k} - \ln n \right) = \gamma = 0.57721\dots$$

получаем

$$\lim_{N \rightarrow \infty} (\bar{d}(N) - 2 \ln N) \rightarrow C$$

Бинарные деревья поиска

- **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

```
tree * 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

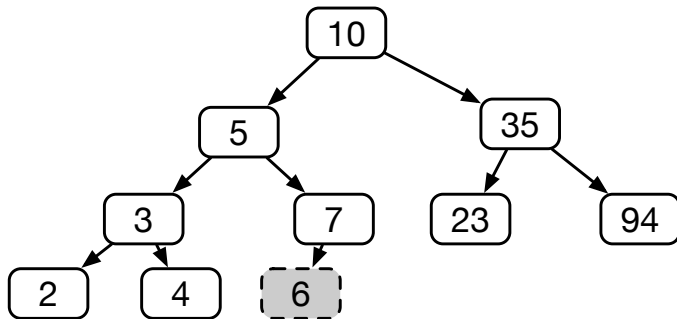
- Процедура удаления сложнее, три случая:
 - 1 Нет потомков — удаляем узел у родителя.
 - 2 Один потомок — переставляем узел у родителя на потомка

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

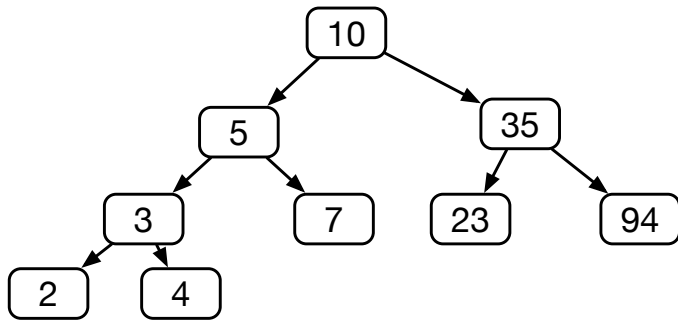
Бинарные деревья поиска: свойства

Первый случай: до удаления



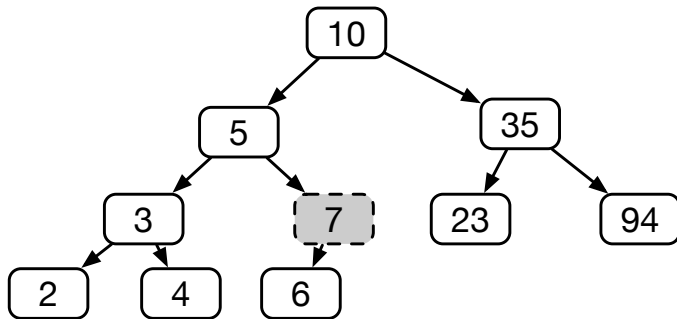
Бинарные деревья поиска: свойства

Первый случай: после удаления



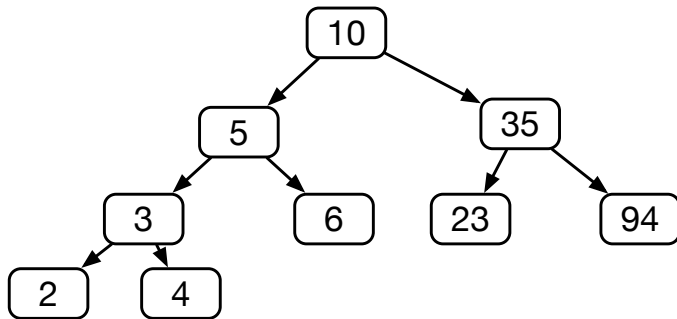
Бинарные деревья поиска: свойства

Второй случай: до удаления



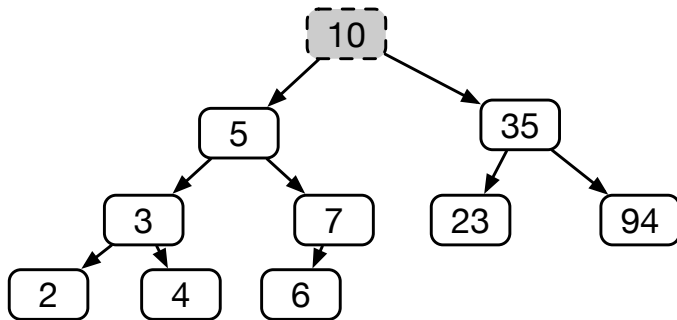
Бинарные деревья поиска: свойства

Второй случай: после удаления



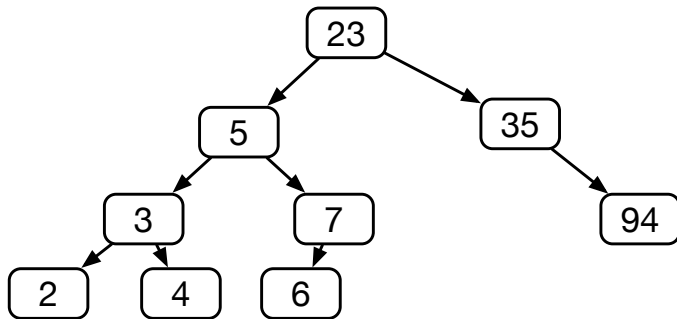
Бинарные деревья поиска: свойства

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

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

Борьба с дисбалансом

Попытка:

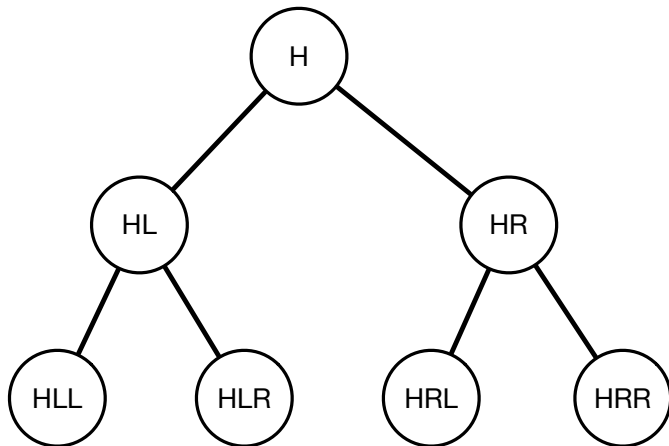
- Предлагается: вставлять новые элементы всегда в корень.
- Последствия: если вставляемый элемент больше корня, то старый корень сделаем левым поддеревом, а его правое поддерево — нашим правым поддеревом.
- Аналогично рассуждаем для случая, когда вставляемый элемент меньше корня.
- Упорядоченность может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

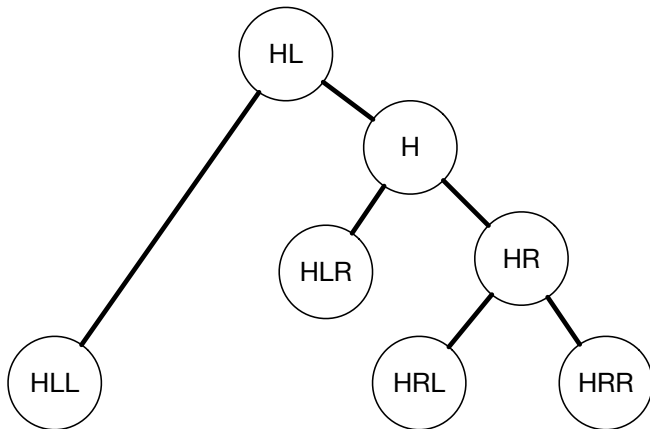
Повороты дерева

Перед поворотом



Повороты дерева

После поворота направо

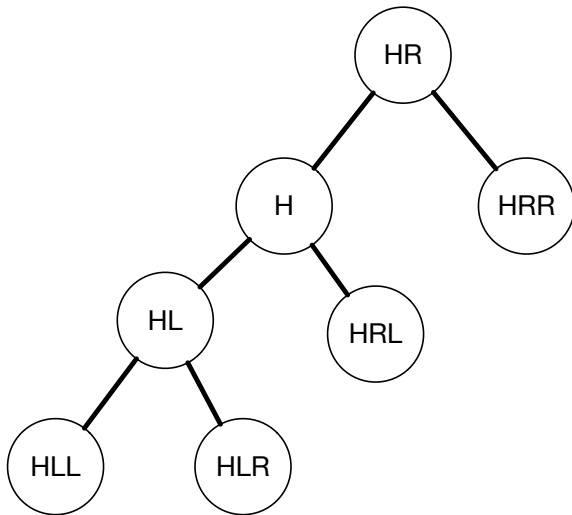


Повороты дерева

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

Повороты дерева

После поворота налево



Вставка в корневой узел

Рекурсивный алгоритм.

```
void insert(node* &head, item x) {  
    if (head == nullptr) {  
        head = new node(x);  
        return;  
    }  
    if (x.key < head->item->key) {  
        insert(head->left, x);  
        rotateRight(head);  
    } else {  
        insert(head->right, x);  
        rotateLeft(head);  
    }  
}
```

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.

Декартовы деревья

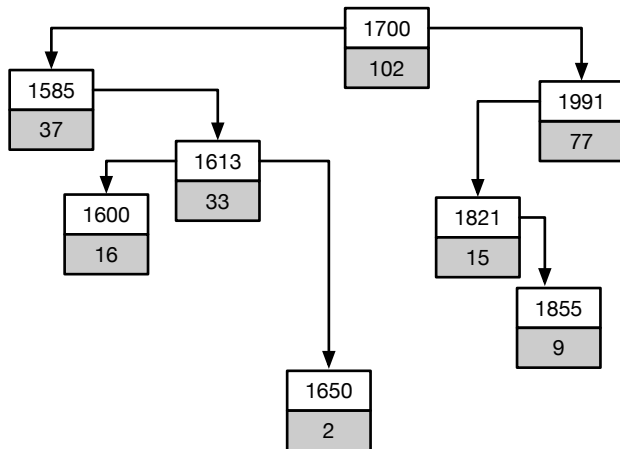
Декартовы деревья

- Случайные бинарные деревья поиска близки к идеальным по сложности ($H = O(\log N)$).
- Можно внести ещё более серьёзный элемент случайности, добавив второй ключ, генерируемый случайно.
- Декартово дерево есть комбинация бинарного дерева поиска (BST) и бинарной кучи (BH) .
- При поиске информации декартово дерево — BST .
- Узлы упорядочиваются по отношениям BH.

Декартовы деревья: свойства

- При вставке в BST можно получить комбинаторное количество различных деревьев, содержащих те же самые элементы.
- При вставке в BST с вторичным упорядочиванием по отношениям ВН получается единственное дерево со свойствами случайного BST.

Декартовы деревья: пример



Декартовы деревья: операции

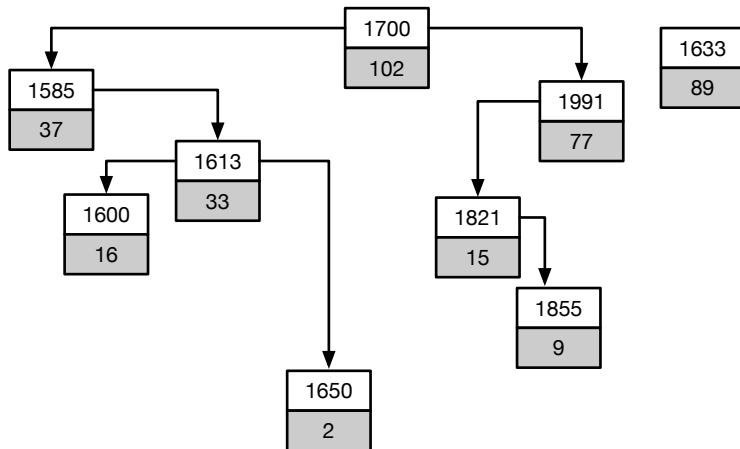
find — Декартово дерево есть BST. ($\log N$)

Декартовы деревья: операции

insert — Декартово дерево есть BST + ВН.

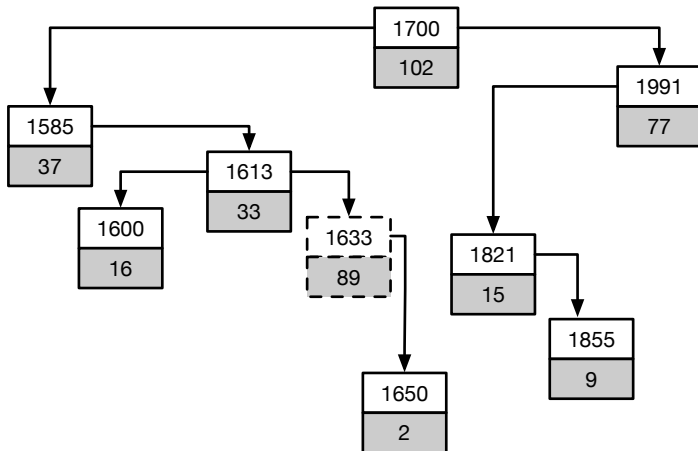
- Первичная вставка проводится в BST. При этом может быть нарушено свойство ВН.
- Если вставленный элемент не нарушает свойства ВН, то вставка завершена.
- Если свойство ВН нарушается, проводится вращение, поднимающее вставленный элемент.
- Подъём происходит до тех пор, пока нарушено свойство ВН.

Декартовы деревья: вставка элемента



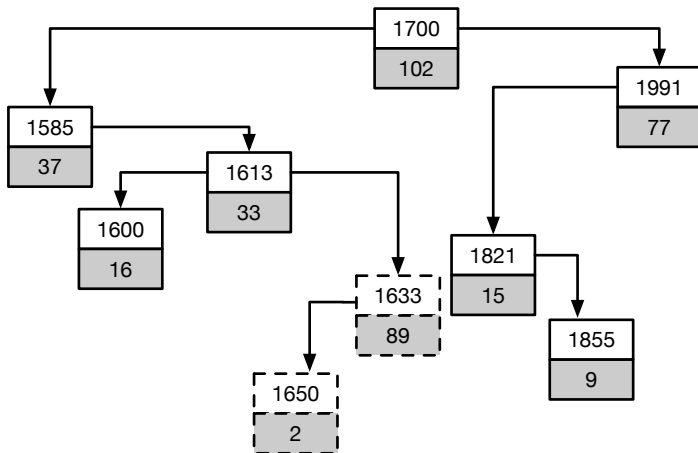
Декартовы деревья: вставка элемента

Элемент вставлен по правилам BST, но он не упорядочен по правилам ВН.



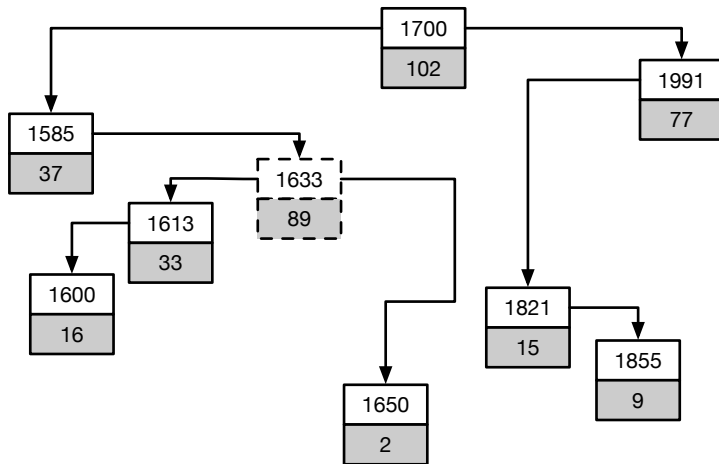
Декартовы деревья: вставка элемента

Попытка обмена с родителем нарушает свойства BST.



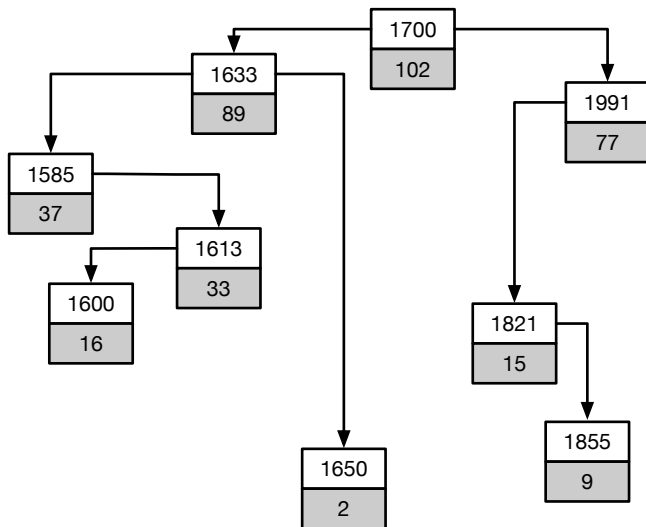
Декартовы деревья: вставка элемента

Вращение в сторону родителя не нарушает свойства BST, но свойство ВН ещё нарушено.



Декартовы деревья: вставка элемента

Ещё одно вращение в сторону родителя и все свойства восстановлены.



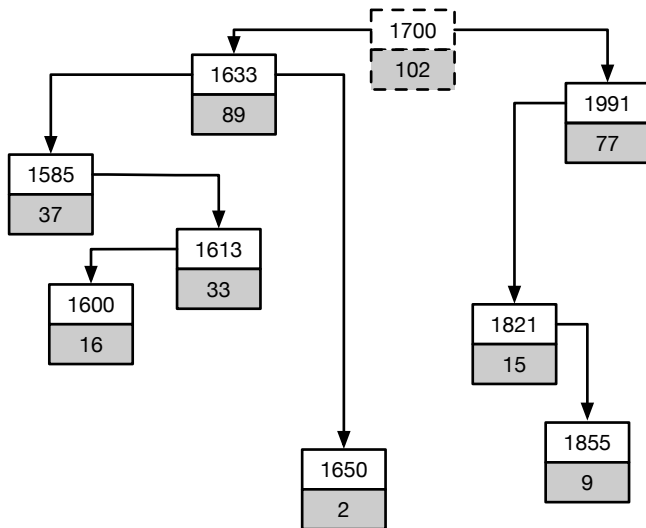
Декартовы деревья: операции

remove — Декартово дерево есть BST + ВН.

- Так как удаление узлов, отличных от вершин, нетривиально, а удаление вершин — тривиально, задача — сделать удаляемый узел терминальным.
- Для этого на каждом шаге вращаем удаляемый узел с его ребёнком, имеющим наибольшее значение u до тех пор, пока он не станет терминальной вершиной.
- На этапе спуска мы не обращаем внимания на сохранение свойства ВН, нас интересуют только значения u .

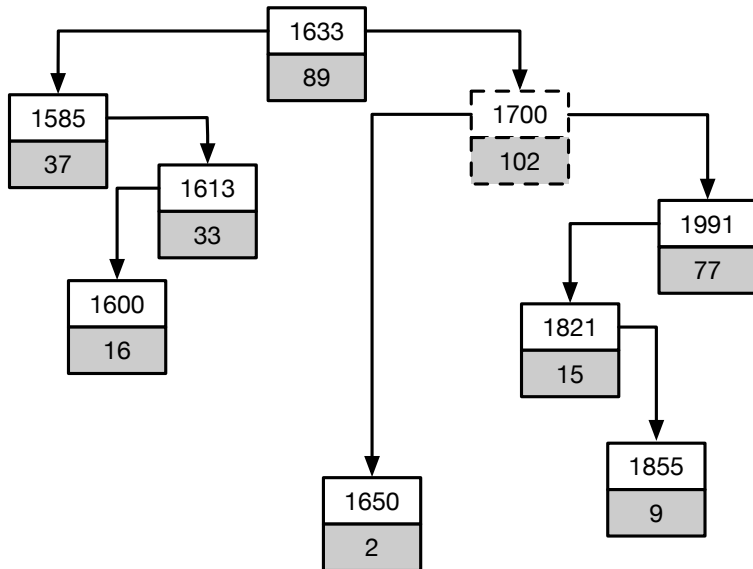
Декартовы деревья: удаление элемента

Попытаемся удалить корневой элемент. (1633,89) имеет наибольшее значение u из детей, вращаем его по направлению к родителю.



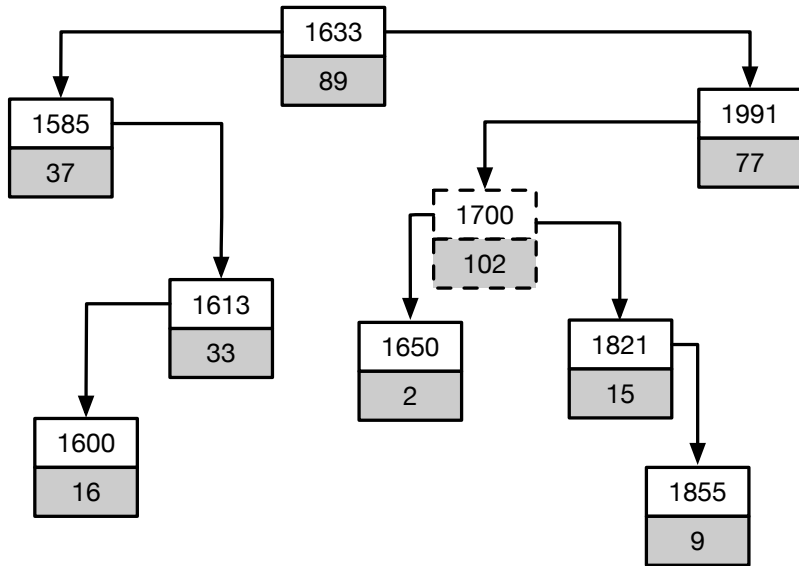
Декартовы деревья: удаление элемента

Теперь новый объект для вращения — узел (1991, 77).



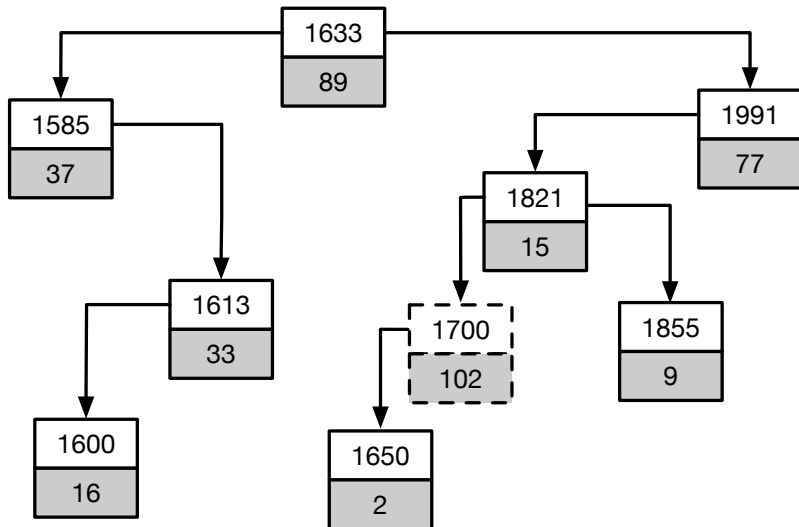
Декартовы деревья: удаление элемента

Следующее направление — узел (1821,15).



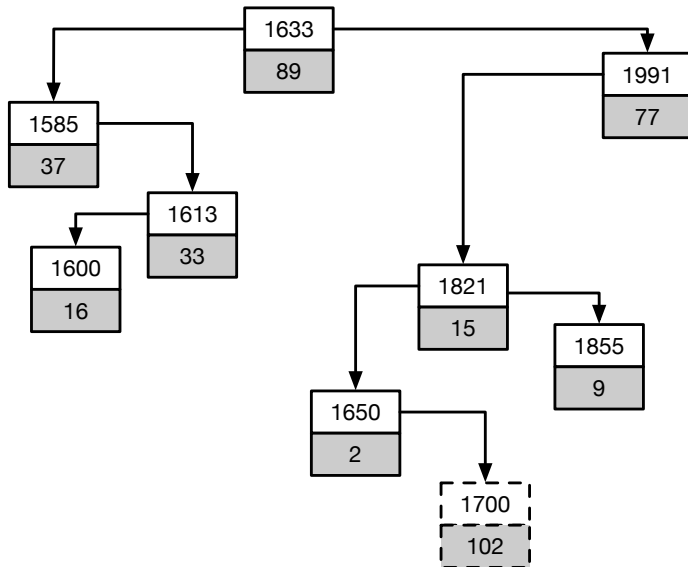
Декартовы деревья: удаление элемента

Последнее направление — узел (1650, 2).



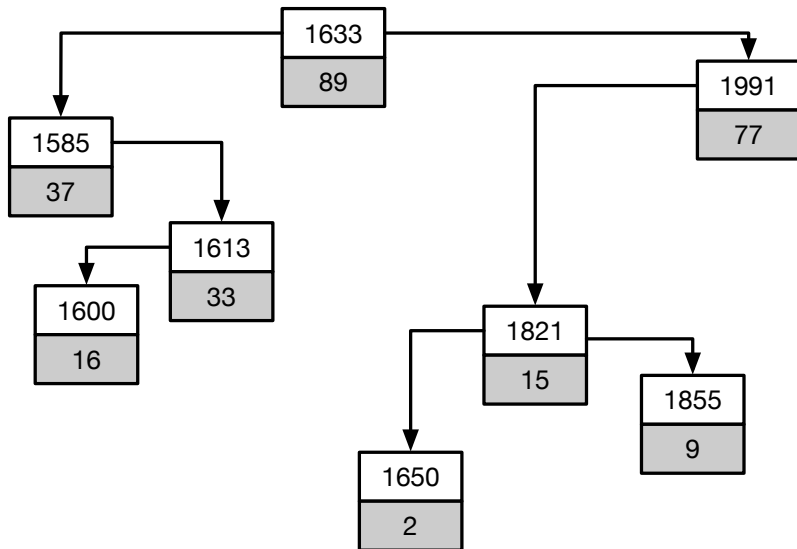
Декартовы деревья: удаление элемента

Удаляемый узел добрался до вершин и может быть удалён.



Декартовы деревья: удаление элемента

Заключительное состояние.



Сбалансированные деревья поиска

Сбалансированные деревья поиска

- Задача: реализовать операции с деревьями, имеющие время в худшем $\Theta(\log N)$.

$$H < A \cdot \log N + B,$$

где A и B — некоторые фиксированные константы.

- Решение:
 - ▶ использовать сбалансированные деревья;
 - ▶ использовать алгоритмы, не нарушающие сбалансированность.

Сбалансированные деревья поиска: критерии сбалансированности

Высота дерева H_t не превосходит $A \log N + B$, если в бинарном дереве с N узлами выполнено хотя бы одно из условий:

- 1 для любого узла количество узлов в левом и правом поддереве N_l, N_r отличаются не более, чем на 1

$$N_r \leq N_l + 1, \quad N_l \leq N_r + 1$$

- 2 для любого узла количество подузлов в левом и правом поддеревьях удовлетворяют условиям

$$N_r \leq 2N_l + 1, \quad N_l \leq 2N_r + 1$$

- 3 для любого узла высоты левого и правого поддеревьев H_l, H_r удовлетворяют условиям

$$H_r \leq H_l + 1, \quad H_l \leq H_r + 1$$

Сбалансированные деревья поиска

Случай 1 — идеально сбалансированное дерево.

Пусть $H_{ideal}(N)$ — максимальная высота идеально сбалансированного дерева.

- N — нечётно и равно $2M + 1$. Тогда левое и правое поддеревья должны содержать ровно по M вершин.

$$H_{ideal}(2M + 1) = 1 + H_{ideal}(M)$$

- N — чётно и равно $2M$. Тогда

$$H_{ideal}(2M) = 1 + \max(H_{ideal}(M - 1), H_{ideal}(M))$$

Так как $H_{ideal}(M)$ — неубывающая функция, то

$$H_{ideal}(2M) = 1 + H_{ideal}(M)$$

$$H_{ideal}(N) \leq \log_2 N$$

Сбалансированные деревья поиска

Случай 2. Примерная сбалансированность количества узлов.

Пусть $H(M)$ — максимальная высота сбалансированного дерева со свойством 2.

- Тогда $H(1) = 0$, $H(2) = H(3) = 1$.
- При добавлении узла один из узлов будет корнем, остальные $N - 1$ распределятся в отношении $N_l : N_r$, где $N_l + N_r = N - 1$.
- Не умаляя общности, предположим, что $N_r \geq N_l$, тогда $N_r \leq 2N_l + 1$.

$$H(N) = \max_{N_l, N_r} (1 + \max(H(N_l), H(N_r)))$$

Сбалансированные деревья поиска

Функция $H(N)$ — неубывающая, поэтому

$$H(N) = 1 + H(\max(N_r, N_l))$$

При ограничениях $N_r \leq 2N_l + 1$ и $N_l + N_r = N + 1$ получаем

$$H(N) = 1 + H\left(\left\lfloor \frac{2N-1}{3} \right\rfloor\right)$$

$$H(N) > 1 + H\left(\left\lfloor \frac{2N}{3} \right\rfloor\right)$$

$$H(N) > \log_{3/2} N + 1 \approx 1.71 \log_2 N + 1$$

Сбалансированные деревья поиска

Случай 3. Примерная сбалансированность высот. AVL-деревья.

Пусть $N(H)$ — минимальное число узлов в AVL-дереве с высотой H (минимальное AVL-дереве).

- Пусть левое дерево имеет высоту $H - 1$.
- Правое дерево будет иметь высоту $H - 1$ или $H - 2$.
- $N(H)$ — неубывающая, для минимального AVL-деревя высота правого равна $H - 2$.
- Число узлов в минимальном AVL-дереве:

$$N(H) = 1 + N(H - 1) + N(H - 2)$$

$$\lim_{h \rightarrow \infty} \frac{N(h + 1)}{N(h)} = \varphi = \frac{\sqrt{5} + 1}{2}$$

$$H(N) \approx \log_{\varphi}(N - 1) + 1 \approx 1.44 \log_2 N + 1$$

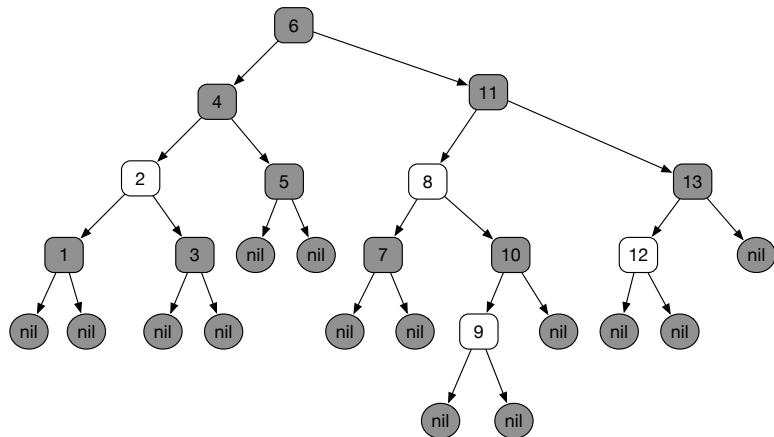
Красно-чёрные деревья.

Красно-чёрные деревья

Красно-чёрное дерево это сбалансированное бинарное дерево поиска.

- Вершины разделены на **красные** и **чёрные**.
- Каждая вершина хранит поля **ключ** и **значение**.
- Каждая вершина имеет указатель *left*, *right*, *parent*
- Отсутствующие указатели помечаются указателями на фиктивный узел *nil*
- Каждый лист *nil* — чёрный
- Если вершина — красная, то её потомки — чёрные
- Все пути от корня *root* к листьям содержат одинаковое число чёрных вершин. Это число называется **чёрной высотой дерева**, **black height**, $bh(root)$

Красно-чёрные деревья



Красно-чёрные деревья

Теорема: красно-чёрное дерево с N внутренними листьями имеет высоту не более $\log_2(N + 1)$

- Для листьев чёрная высота равна нулю.
- Докажем, что $|T_x| \geq 2^{bh(x)}$.
- База индукции: Пусть вершина x является листом. Тогда $bh(x) = 0$ и $|T(x)| = 1 = 2^0$.
- Пусть вершина x не является листом и $bh(x) = k$. Тогда для обоих потомков $bh(l) \geq k - 1$, $bh(r) \geq k - 1$, т. к. красный будет иметь высоту k , чёрный — $k - 1$.
- По предположению индукции
 $|T_l|, |T_r| \geq 2^{k-1} \rightarrow |T_k| = |T_l| + |T_r| \geq 2^k - 1$



Красно-чёрные деревья

- По свойству (3) не менее половины узлов составляют чёрные вершины.
- $bh(t) \geq H/2$
- $N \geq 2^{H/2} - 1$

$$H \leq 2 \cdot \log_2 N + 1$$

Красно-чёрные деревья: операция вставки

- При обычной вставке свойства красно-чёрности могут нарушаться.
- Для изменения структуры применяются операции поворота деревьев.
- Для изменения красно-чёрности применяется корректировка.
- Для удобства полагаем, что для дерева имеется узел *nil*

Красно-чёрные деревья: структуры данных

```
struct tree {
    struct tnode *root, *nil;
    tree();
    ~tree() { delete nil; }
};

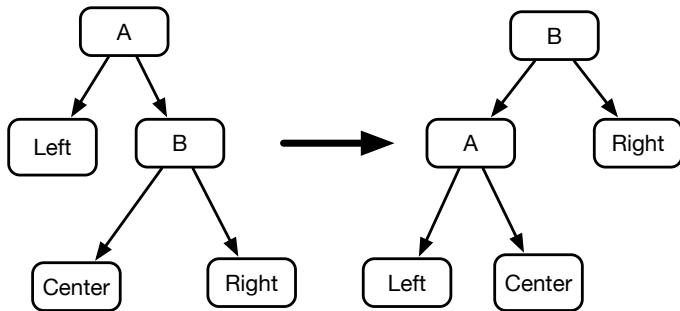
struct tnode {
    tnode *left, *right, *parent;
    bool black;
    mydata data;
    tnode(tree *t) {
        left = right = parent = t->nil;
    }
};

tree::tree() {
    nil = new tnode(); nil->black = true;
};
```

Красно-чёрные деревья: повороты

Для поддержания сбалансированности применяется операция *вращение* или *поворот*.

Для этого отцепляется поддерево и переносится на другую сторону.



Левый поворот дерева.

Красно-чёрные деревья: вставка

- Вставляем почти как в обычное бинарное дерево поиска.
- Красим узел в красный цвет
- Корректируем дерево для сохранения красно-чёрности.

Красно-чёрные деревья: вставка

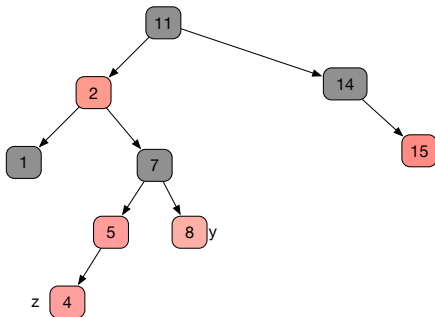
```
void tree_insert(tree *t, tnode *z) {
    tree *y = t->nil; tree *x = t->root;
    while (x != t->nil) {
        y = x;
        if (z->key < x->key) x = x->left;
        else                x = x->right;
    }
    z->parent = y;
    if (y == t->nil) t->root = z;
    else {
        if (z->key < y->key) y->left = z;
        else                y->right = z;
    }
    z->left = z->right = t->nil;
    z->black = false;
    insert_fixup(t, z);
}
```

Красно-чёрные деревья: коррекция (фрагмент)

```
void insert_fixup(tree *t, tnode *z) {
    while(!z->parent->black) {
        if (z->parent == z->parent->parent->left) {
            tnode *y = z->parent->parent->right;
            if (!y->black) {
                z->parent->black = true;
                y->black = true;
                z->parent->parent->black = false;
                z = z->parent->parent;
            } else {
                if (z == z->parent->right) {
                    z = z->parent;
                    rotate_left(t, z);
                    z->parent->black = true;
                    z->parent->parent->black = false;
                    rotate_right(t, z->parent->parent);
                }
            }
        } else ... left <-> right
    }
    t->root->black = true;
}
```

Красно-чёрные деревья

Вставка 4, корректирование

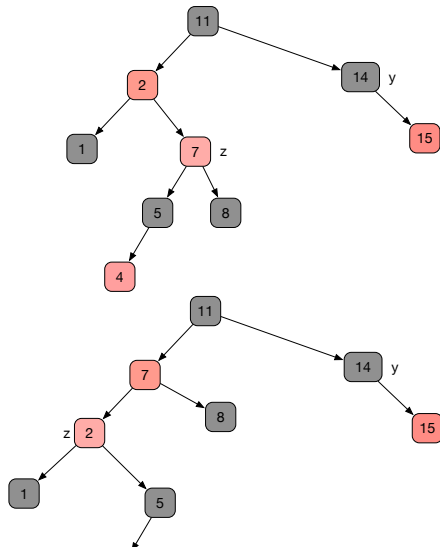


Смена цветов

Красно-чёрные деревья

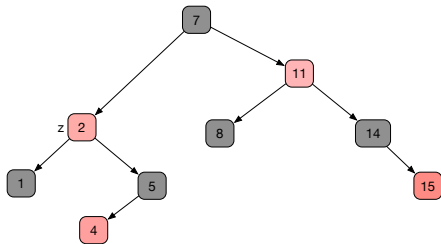
Поворот

Вставка 4, корректирование



Красно-чёрные деревья

Заключительная коррекция



Красно-чёрные деревья vs AVL-деревья

	RB-tree	AVL-tree
Средняя высота	до $1.38N$	N
Поиск/вставка	до $1.38t$	t
Поворотов при вставке	до 2	до 1
Поворотов при удалении	до 3	до $\log N$
Дополнительная память	1 бит	1 счётчик

Параллельное использование алгоритмов поиска. Списки с пропусками.

Параллельное использование

- При параллельном программировании к одному элементу данных может обратиться несколько потоков.
- Результат при этом может быть недетерминирован.

```
int a = 0, b = 0;
```

```
//thread 1
```

```
b = 2;
```

```
a = b + 1;
```

```
//thread 2
```

```
a = 4;
```

```
b = a - 3;
```

Параллельное использование

- Критерий Бернстайна: Поместим объекты, которые читаются в потоке i в множество R_i , а те, которые пишутся, в множество W_i .
- Для нашего кода $R_1 = \{b\}$, $W_1 = \{a, b\}$, $R_2 = \{a\}$, $W_2 = \{a, b\}$.
- Критерий гласит, что если все пересечения множеств $R_1 \cap W_2$, $R_2 \cap W_1$, $W_1 \cap W_2$ пусты, то конфликтов (*race conditions*) не возникнет.
- В нашем случае: $R_1 \cap W_2 = \{a\}$, $R_2 \cap W_1 = \{a\}$, $W_1 \cap W_2 = \{a, b\}$, то есть *race conditions* возможны.

Параллельное использование

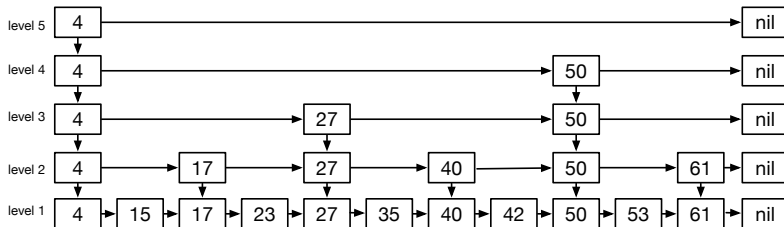
- Одно из средств избежать *race conditions* — использование атомарных операций.
- Существуют машинные команды типа *Compare-And-Swap*, выполняющиеся атомарно.
- Они позволяют атомарно обменять две ячейки памяти, которые, возможно, содержат указатели.
- При вставке в односвязный список достаточно атомарных операций для замены цепочки указателей.
- Односвязный список — идеальная структура данных для параллельного программирования.

Параллельное использование

- Операция поиска в односвязном списке $T(N) = O(N)$
- Операции вставки и удаления в односвязном списке $T(N) = O(N)$
- Требуется по возможности сохранить свойства операций вставки и удаления в лучшем случае и ускорить операцию поиска.

Списки с пропусками

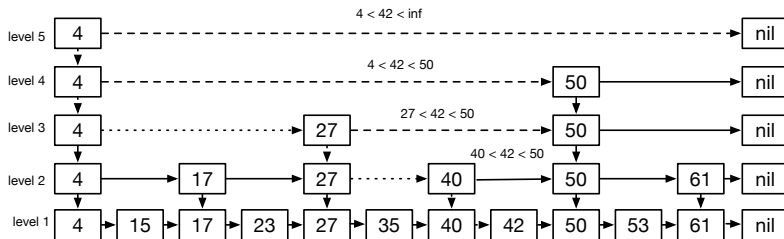
Рассмотрим следующую структуру данных:



- Она представляет из себя несколько списков, организованных в виде списков.
- Каждый следующий список примерно в два раза короче предыдущего и он пропускает примерно половину элементов предыдущего.

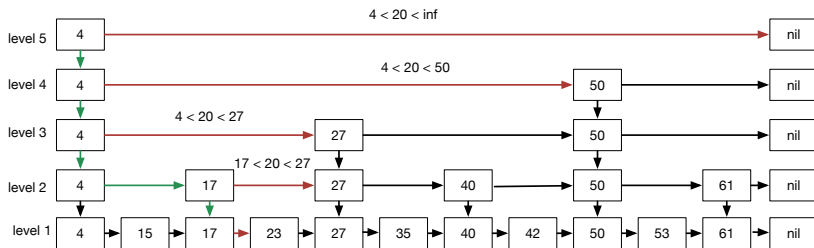
Списки с пропусками

- Поиск существующего элемента.



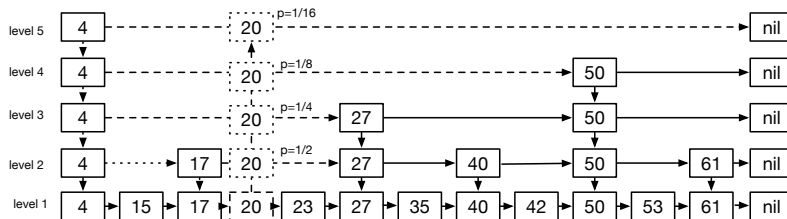
Списки с пропусками

- Поиск несуществующего элемента.



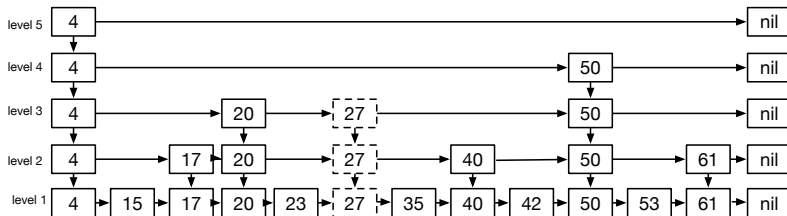
Списки с пропусками

• Вставка элемента.



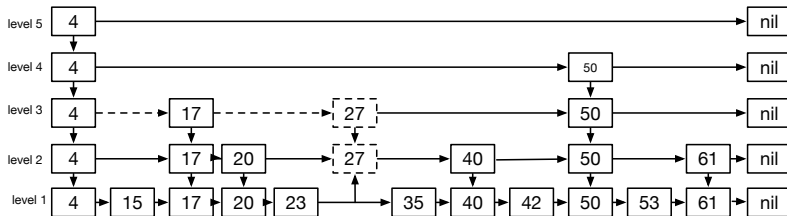
Списки с пропусками

- Удаление элемента. Поиск и пометка столбца.



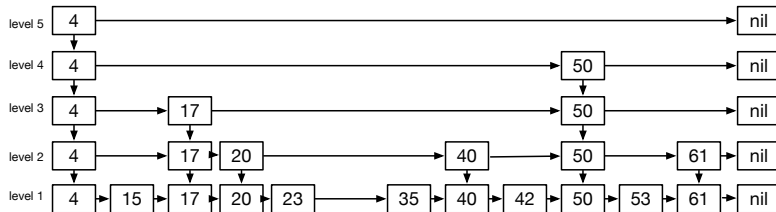
Списки с пропусками

- Удаление элемента. Удаление из строк.



Списки с пропусками

- Удаление элемента. Заключительное удаление.



Списки с пропусками

- Вставка 10^6 элементов в структуру данных.

Укладывание	Array	RBTree	SkipList
Случайно	127033 ms	1020 ms	1737 ms
По возрастанию	108 ms	457 ms	536 ms
По убыванию	256337 ms	358 ms	407 ms

Списки с пропусками

Амортизационная сложность списков с пропусками:

- Вставка — $T(N) = O(\log N)$
- Поиск — $T(N) = O(\log N)$
- Удаление — $T(N) = O(\log N)$

Внешний поиск. В-деревья.

Внешний поиск с использованием В-деревьев

- Основной носитель информации — жёсткий диск.
- Информация на жёстком диске располагается в *секторах*, которые логически расположены на *дорожках*.
- Размер сектора типично 512/2048/4096 байт.
- Информация считывается и записывается *головками чтения/записи*.
- Для чтения/записи информации требуется *подвести* головку чтения записи к нужной дорожке и дождаться подхода нужного сектора.
- Типичные скорости вращения жёстких дисков — 5400/7200/10033/15000 оборотов в минуту.
- Один оборот совершается за время от 1/90 до 1/250 секунды.
- Операция перехода на соседнюю дорожку примерно 1/1000 секунды.

Работа с внешними носителями

- Внешние сортировки используют многократный последовательный проход по данным, расположенным на носителях информации.
- Последовательное считывание информации с жёсткого диска 100-150 мибайт в секунду.
- Смена позиции в файле часто требует:
 - ▶ ожидания подвода головки на нужную дорожку;
 - ▶ ожидания подхода нужного сектора к головкам чтения/записи;
- Операция последовательного чтения 4096 байт занимает $\frac{4096}{100 \times 10^6} \approx 40 \times 10^{-6}$ секунд
- Операция случайного чтения 4096 байт занимает не менее $5 - 10 \times 10^{-3}$ секунд.

Работа с внешними носителями

- Второй популярный носитель — SSD диск.
- Информация хранится в энергонезависимой памяти на микросхемах.
- Операции производятся блоками размером 64-1024 кибибайт.
- Время доступа к блоку $\approx 10^{-6}$ секунд.
- HDD и SSD используют *буферизацию* для ускорения работы.
- Алгоритмы поиска во внешней памяти должны минимизировать число обращений к внешней памяти.

Работа с SSD носителями

- На логическом уровне обращения происходят блоками любого размера, кратного 512 байт.
- На физическом уровне всё сложнее.
- Размер физического блока от 64 до 1024 кибибайт.
- Операция частичной записи 512 байт:
 - ❶ Считывается полный блок (всегда).
 - ❷ Заменяется 512 байт в требуемом месте.
 - ❸ Записывается полный блок (всегда).
- Выровненная запись целого блока — минимум двукратное ускорение.

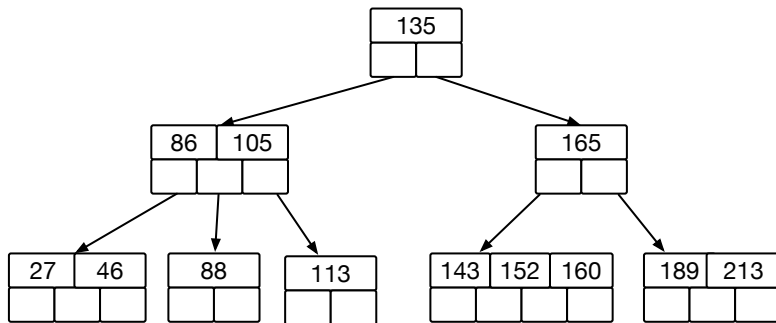
Оценка применимости внешнего поиска

- Пусть имеется бинарное дерево поиска, состоящее из:
 - ① Данных размером 64 байта.
 - ② Ключа размером 8 байт.
 - ③ Указателей left и right размером 8 байт.
- Общий размер узла — 88 байт.
- В оперативную память размером 16 гигабайт поместится $\frac{16 \times 2^{30}}{88} \approx 195 \times 10^6$ узлов.
- Как хранить словарь из 10^9 элементов?

B-деревья

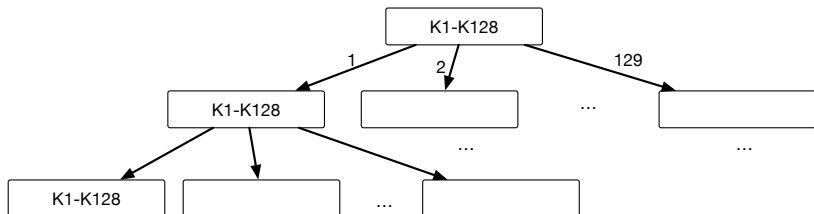
- *B-дерево* — сбалансированное дерево поиска, узлы которого хранятся во внешней памяти.
- В оперативной памяти хранится часть узлов.

B-деревья: свойства



- Высота дерева не более $O(\log N)$, где N — количество узлов.
- Каждый узел может содержать 1 ключ и больше.
- Количество детей узла равно $K + 1$, где K — количество ключей в узле.

B-деревья: свойства



- Пусть в узле помещается 128 ключей.
- Высота дерева — 3
- Тогда общее количество узлов

$$1 + 129 + 129^2 = 16771$$

- Общее количество ключей

$$16771 \times 128 = 2146688$$

В-деревья: определение

- **В-дерево** — корневое дерево, обладающее свойствами:

- ① Каждый узел содержит:

- ★ количество ключей n , хранящихся в узле.
- ★ индикатор листа `final`.
- ★ n ключей в порядке возрастания.
- ★ $n+1$ указатель на детей, если узел не корневой.

- ② Ключи есть границы диапазонов ключей в поддеревьях.

- ③ Все листья расположены на одинаковой глубине h .

- ④ Имеется показатель t — минимальная степень дерева.

- ⑤ В корневом узле от 1 до $2^t - 1$ ключей.

- ⑥ Во внутренних узлах минимум $t - 1$ ключей.

- ⑦ Во внешних узлах максимум $2^t - 1$ ключей.

- ⑧ Заполненный узел имеет $2^t - 1$ ключ.

В-деревья: высота

- **Теорема:** Высота В-дерева с $n \geq 1$ ключами и минимальной степенью $t \geq 2$ в худшем случае не превышает $\log_t \frac{n+1}{2}$
- **Доказательство.** В максимально высоком дереве высоты h в каждом узле, кроме корневого, содержится $t - 1$ ключ. Тогда общее количество ключей в дереве есть

$$\begin{aligned} & 1 + 2 + 2t + 2t^2 + \dots + 2t^{h-1} = \\ & = 1 + 2(t-1)(1 + t + t^2 + \dots + t^{h-1}) = \\ & = 1 + 2(t-1)\frac{t^h - 1}{t - 1} \end{aligned}$$

Отсюда

$$h = \log_t \frac{n+1}{2}$$

B-деревья: операции

- Используем операции Load и Store.
- Корень сохраняем в оперативной памяти.
- Минимизируем количество операций.

B-деревья: операция `find` поиска ключа k

- 1 Операцией бинарного поиска ищем самый левый ключ $key_i \geq k$
- 2 Если $key_i = k$, то узел найден.
- 3 Исполняем `Load` для дочернего узла и рекурсивно повторяем операцию.
- 4 Если $final = true$, то ключ не найден.

Количество операций $T_{load} = O(h) = O(\log_t n)$

Добавление ключа

- 1 Операцией `find` находим узел для вставки.
- 2 Если лист не заполнен, сохраняя упорядоченность вставляем ключ.
- 3 Если лист заполнен ($2t-1$ ключей), разбиваем его на два листа по $t-1$ ключу поиском медианы.
- 4 Медиана рекурсивно вставляется в родительский узел.

Сложность в худшем случае: каждый раз разбивается узел на каждом уровне, $O(t \log_t n)$

Количество операций: $T_{ext} = O(h) = O(\log_t n)$

Разновидности В-деревьев

- B^+ -дерево содержит информацию только в листьях, ключи — только во внутренних узлах.
- Используется в файловых системах XFS, JFS, NTFS, Btrfs, HFS, APFS, ...
- Используется для хранения индексов в базах данных Oracle, Microsoft SQL, IBM DB2, Informix, ...

Спасибо за внимание.

Следующая лекция —
Обобщённый быстрый поиск.