Разработка и анализ алгоритмов

Лекция 10
Деревья поиска

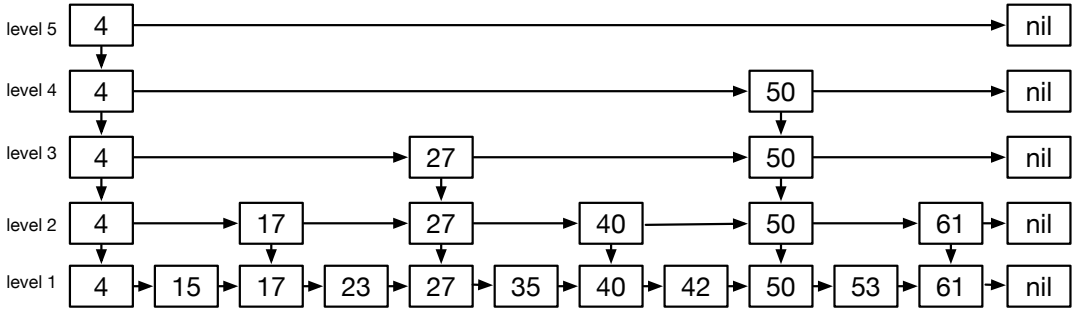
Сергей Леонидович Бабичев

Ускорение списков для CRUD

- Пока: операция поиска в односвязном списке $T(N) = O(N)$
- Пока: 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 все операции.

Списки с пропусками

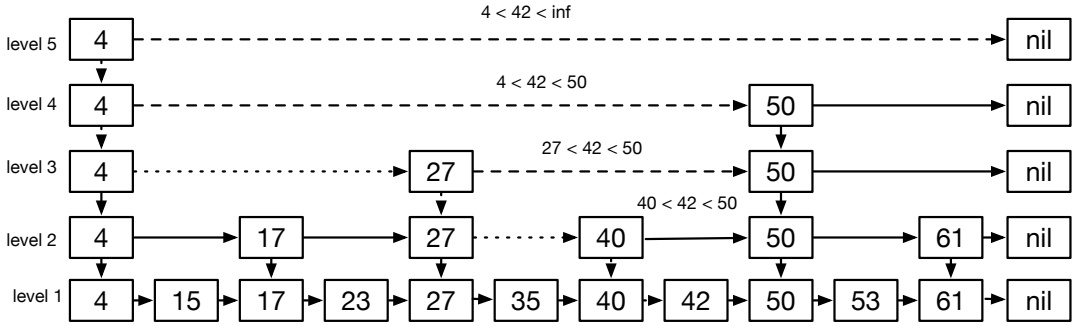
Рассмотрим структуру данных `SkipList`:



- Это — несколько списков, организованных в виде списков.
- Каждый следующий список примерно в два раза короче предыдущего и он пропускает примерно половину элементов предыдущего.

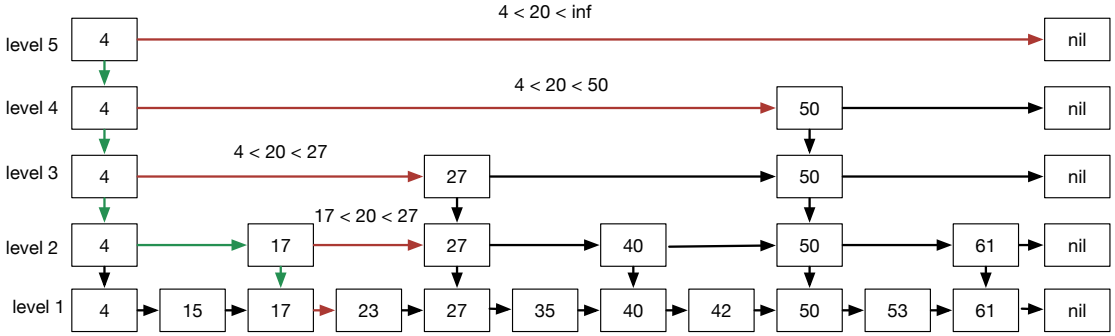
Списки с пропусками

- Поиск существующего элемента.



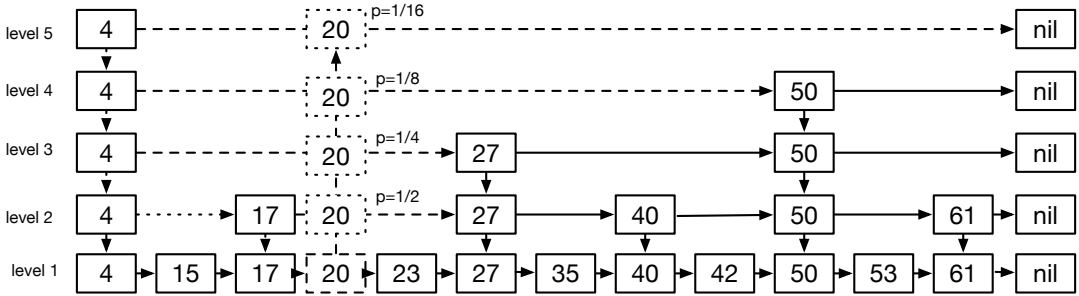
Списки с пропусками

- Поиск несуществующего элемента.



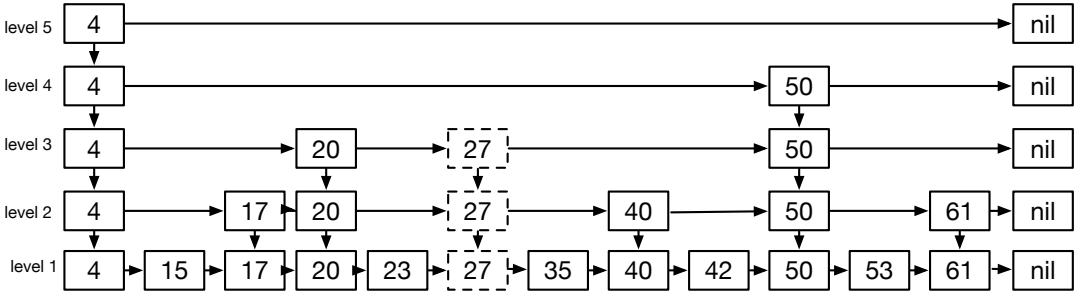
Списки с пропусками

- Вставка элемента.



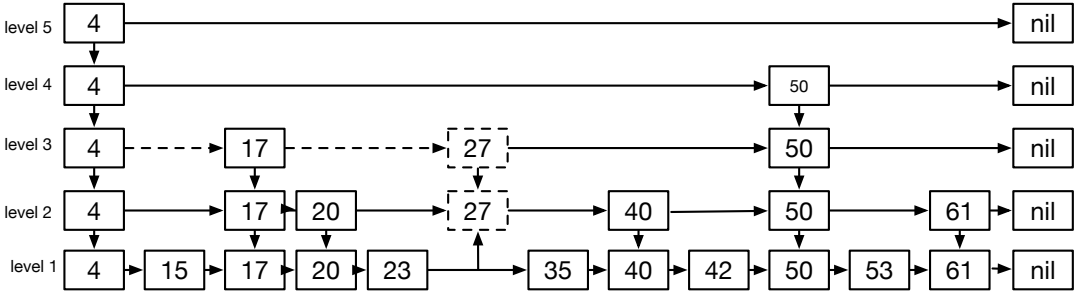
Списки с пропусками

- Удаление элемента. Поиск и пометка столбца.



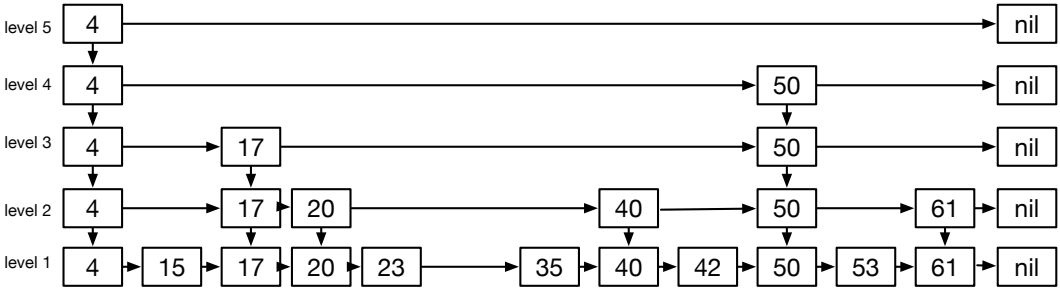
Списки с пропусками

- Удаление элемента. Удаление из строк.



Списки с пропусками

- Удаление элемента. Заключительное удаление.



Списки с пропусками как CRUD-структура

- Сложность операций:
  - ***Create*** — $O(\log N)$
  - ***Read*** — $O(\log N)$
  - ***Update*** — $O(\log N)$
  - ***Delete*** — $O(\log N)$

Структура данных «дерево».

Деревья: особенности

Основная особенность деревьев — наличие узлов и нескольких наследников у них.

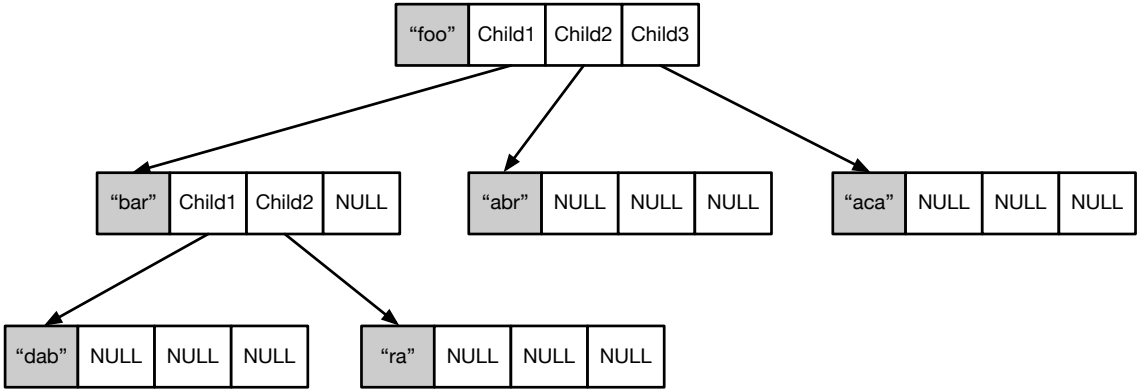
```
struct tree {
  struct tree *children[3];
  myType data;
  ...
};
```

Деревья: соглашения

- Любое `N`-ричное дерево может представлять деревья меньшего порядка.
- Соглашение: если наследника нет, соответствующий указатель равен `NULL`.
- Деревья `1`-ричного порядка существуют (списки).

Деревья: троичное дерево

Пример дерева троичного дерева или дерева 3-порядка.



Деревья: классификация

- Условно все элементы дерева делят на две группы:
  - ***Вершины***, не содержащие связей с потомками.
  - ***Узлы***, содержащие связи с потомками.
- Второй вариант — все элементы дерева называют ***узлами***, а ***вершина*** — частный случай ***узла***, ***терминальный узел***.
- Ещё термины:
  - ***Родитель (parent)***
  - ***Дети (children)***
  - ***Братья (sibs)***
  - ***Глубина (depth)***

$$D_{node} = D_{parent} + 1$$

Деревья: создание узла

- Добавим метод создания элемента (узла) дерева:

```
typedef struct tree_s {
  char *data data;
  struct tree_s *child[3];
} tree;

tree* new_tree(const char *init) { //  constructor
    data = strdup(init);
    child[0] = child[1] = child[2] = NULL;
}
```

Деревья: пример построения

- Дерево из примера строится, например, так:

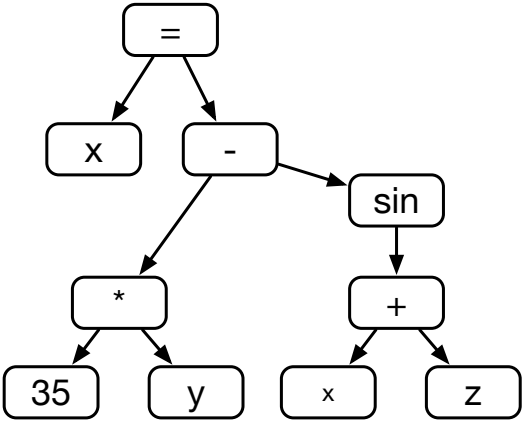
```
tree *root = new_tree("foo");
root->child[0] = new_tree("bar");
root->child[1] = new_tree("abr");
root->child[2] = new_tree("aca");
root->child[0]->child[0] = new_tree("dab");
root->child[0]->child[1] = new_tree("ra");
```

Деревья: пример использования

Использование деревьев:

- Для представления выражений в языках программирования.

x = 35y - sin(x+z);



Деревья: обход

- Алгоритмы работы с деревьями часто рекурсивны.
- Всего существует $6=3!$ способов обхода бинарного дерева.
- На практике применяют четыре основных варианта рекурсивного обхода:
  - Прямой (pre-order)
  - Симметричный (in-order)
  - Обратный (post-order)
  - Обратно симметричный

Деревья: обход

Бинарное дерево.

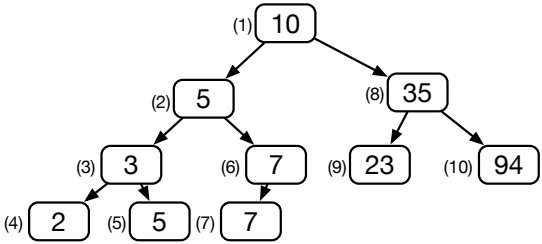
```
typedef struct tree_s {
  char *data;
  struct tree_s *left, *right;
} tree;

tree* new_tree(const char *init) { //  constructor
    tree *t = calloc(1, sizeof(tree));
    t->data = strdup(init);
    return t;
}
```

Деревья: обход pre-order

Прямой способ обхода.

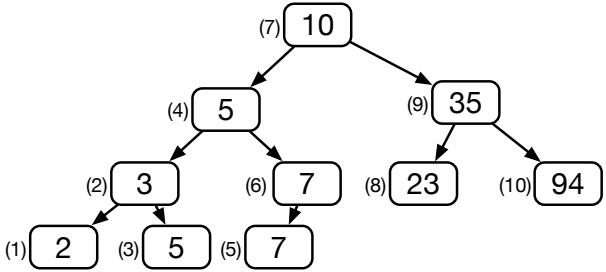
```
void walk(tree *t) {
  work(t);
  if (t->left != NULL) walk(t->left);
  if (t->right != NULL) walk(t->right);
}
```



Деревья: обход in-order

Симметричный способ обхода.

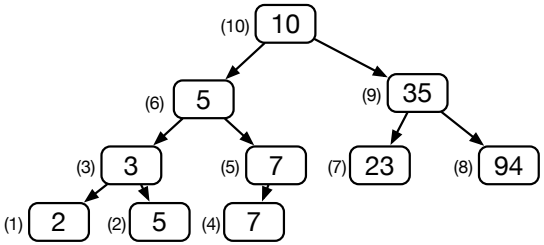
```
void walk(tree *t) {
  if (t->left != NULL) walk(t->left);
  work(t);
  if (t->right != NULL) walk(t->right);
}
```



Деревья: обход out-order

Обратный способ обхода.

```
void walk(tree *t) {
  if (t->left != NULL) walk(t->left);
  if (t->right != NULL) walk(t->right);
  work(t);
}
```

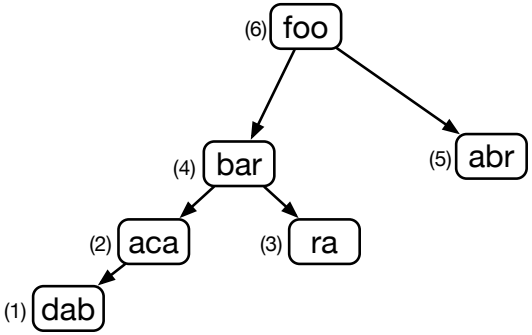


Деревья: обход

Функция обработки может быть параметром.

```
typedef void (* walkFunction)(tree *);
void walk(tree *t, walkFunction wf) {
  if (t->left != NULL) walk(t->left, wf);
  if (t->right != NULL) walk(t->right, wf);
  wf(t);
}
void printData(tree *t) {
  printf("t[%p]='%s'\n", t, t->data);
}
int main() {
  tree *root = new_tree("foo");
  root->left = new_tree("bar");
  root->right = new_tree("abr");
  root->left->left = new_tree("aca");
  root->left->left->left = new_tree("dab");
  root->left->right = new_tree("ra");
  walk(root, printData);
  // memory leak
}
```

Деревья: обход




```
t[0x7ff0e1c03290]='dab'
t[0x7ff0e1c03260]='aca'
t[0x7ff0e1c032d0]='ra'
t[0x7ff0e1c03200]='bar'
t[0x7ff0e1c03230]='abr'
t[0x7ff0e1c031d0]='foo'
```

Деревья: обход

Вывод генеалогического дерева (обратно симметричный обход):

```
typedef void (* walkFunction)(tree *, int);
void walk(tree *t, walkFunction wf, int lev) {
  if (t->right != NULL) walk(t->right, wf, lev+1);
  wf(t, lev);
  if (t->left != NULL) walk(t->left, wf, lev+1);
}
void printData(tree *t, int lev) {
  for (int i = 0; i < lev; i++) {
    printf("  ");
  }
  printf("%s\n", t->data);
}
int main() {
  tree *root = new_tree("foo");
  ...
  walk(root, printData, 0);
  // memory leak
}
```

Деревья: обход

Вывод программы:

```
  abr
foo
    ra
  bar
    aca
      dab
```

Деревья: обход

- При использовании рекурсивно-динамических структур данных для некоторых операций важно выбрать верный обход дерева.
- Освободим память, начиная от корня дерева.
- Заказ памяти под поддеревья происходил динамически.
- Имеется узел, от которого шло построение дерева.
- Так как в данном дереве не хранится информация о том, кто является предком узла, корневой узел — центр всего построения.
- При операции освобождения памяти узла исказятся указатели на подузлы.

Деревья: освобождение памяти

Рекурсивный вариант:

```
tree *tree_destroy(tree *t) {
    if (t->left != NULL) t->left = tree_destroy(t->left);
    if (t->right != NULL) t->right = tree_destroy(t->right);
    free(t->data);
    return NULL;
}
```

Деревья поиска

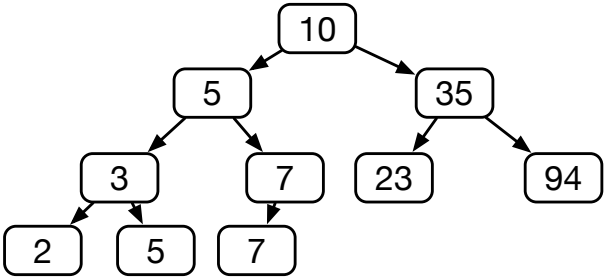
Деревья: поиск

Использование деревьев для поиска.
Задача:

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

Деревья: поиск

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



Деревья: поиск

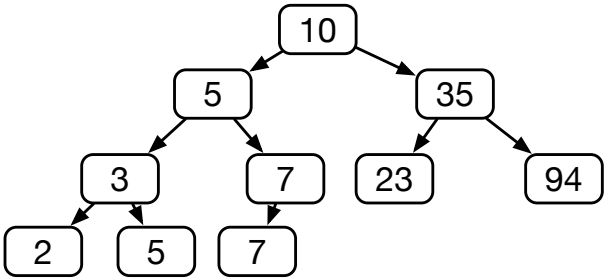
Поиск по дереву после получения элемента с ключом X:

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

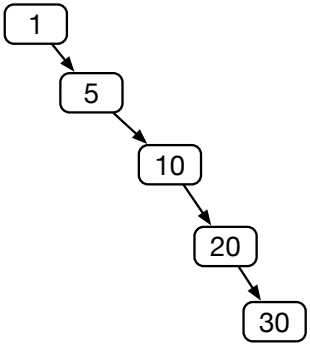
{10, 5, 35, 7, 3, 23, 94, 2, 5, 7}



Бинарные деревья поиска

Отвратительное дерево (бамбук)

{1, 5, 10, 20, 30}



Бинарные деревья поиска

Definition

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 $p_k = \dfrac{1}{N+1}$
- Остальные узлы разобьются на группы, каждая из которых начнётся с высоты 1. В левую группу войдут элементы $\{0, \ldots, k-1\}$, в правую — $\{k+1, \ldots, N\}$.

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

Бинарные деревья поиска

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

Используя предел

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

получаем

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

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;
}

tree * minNodeRec(tree *t) {
   return t == NULL? NULL: minNodeRec(t->left);
}
```

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

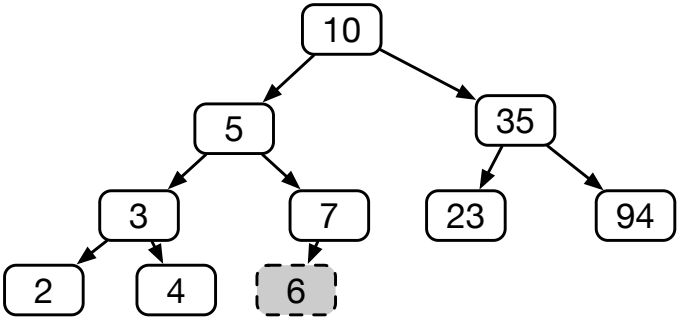
- Процедура удаления сложнее, три случая:
  1. Нет потомков — удаляем узел у родителя.
  2. Один потомок — переставляем узел у родителя на потомка.

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.

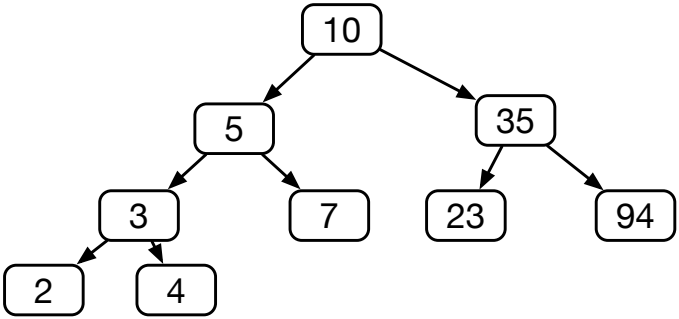
Бинарные деревья поиска: свойства

Первый случай: до удаления



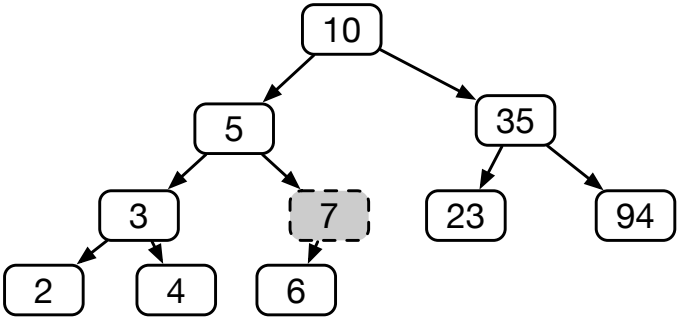
Бинарные деревья поиска: свойства

Первый случай: после удаления



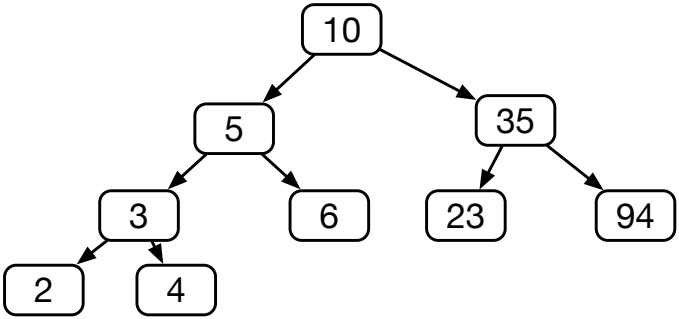
Бинарные деревья поиска: свойства

Второй случай: до удаления



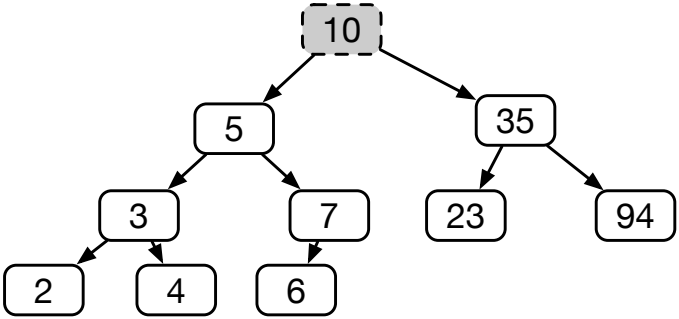
Бинарные деревья поиска: свойства

Второй случай: после удаления



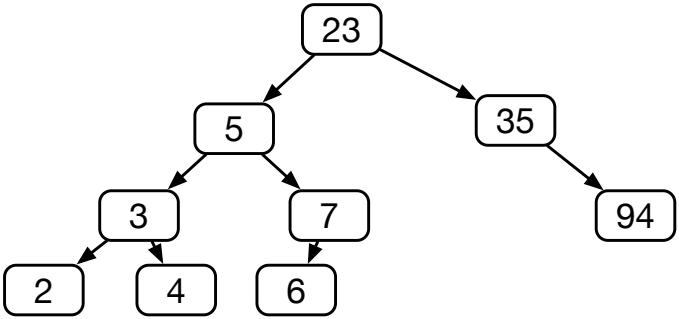
Бинарные деревья поиска: свойства

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

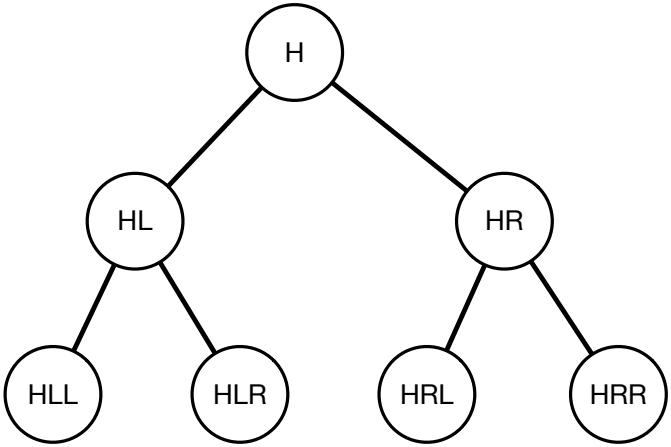
Попытка:

- 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.
- Структура дерева 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.

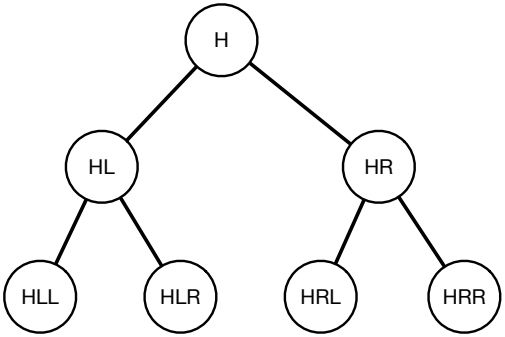
Повороты дерева

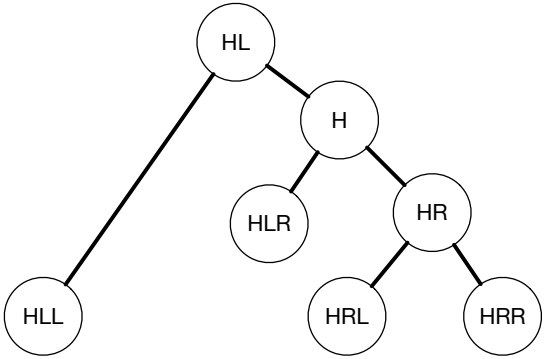


Перед поворотом

Повороты дерева

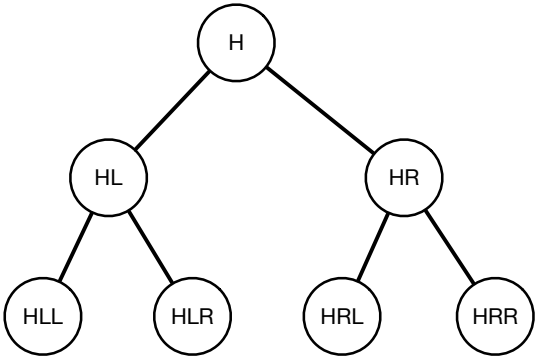
После поворота направо

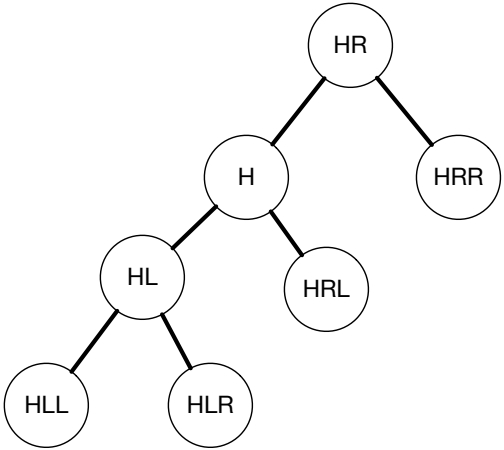




Повороты дерева

После поворота налево





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) {
      return new_node(x);
   }
   if (x.key < head->item->key) {
      head->left = insert(head->left, x);
      return rotateRight(head);
   } else {
      head->right = insert(head->right, x);
      return 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

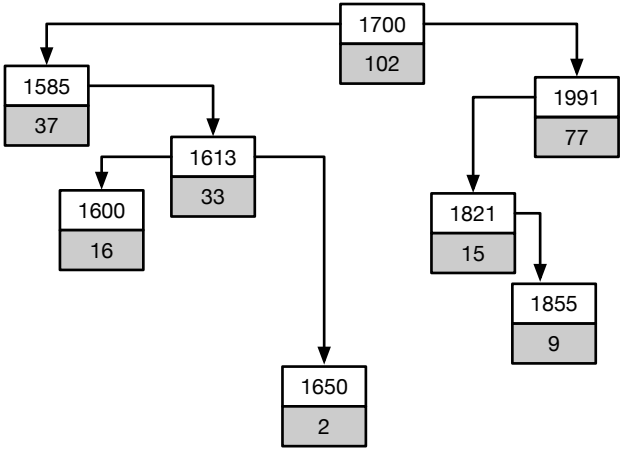
Декартовы деревья

- 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 BH получается единственное дерево со свойствами случайного BST.

Декартовы деревья: пример



Декартовы деревья: операции

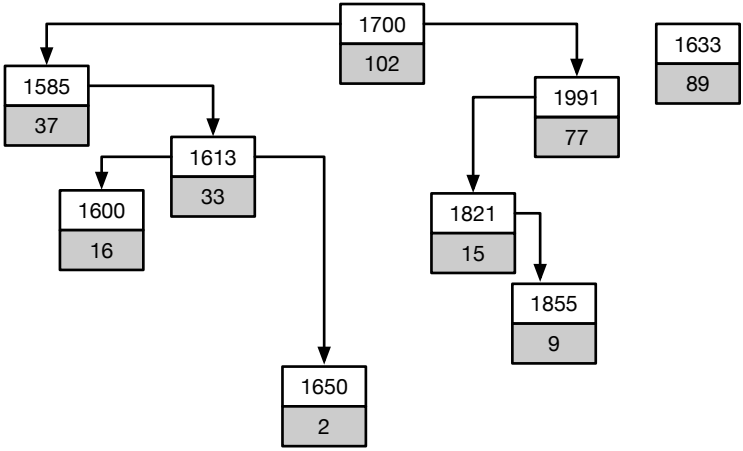
find — Декартово дерево есть BST. $(\log N)$

Декартовы деревья: операции

insert — Декартово дерево есть BST + BH.

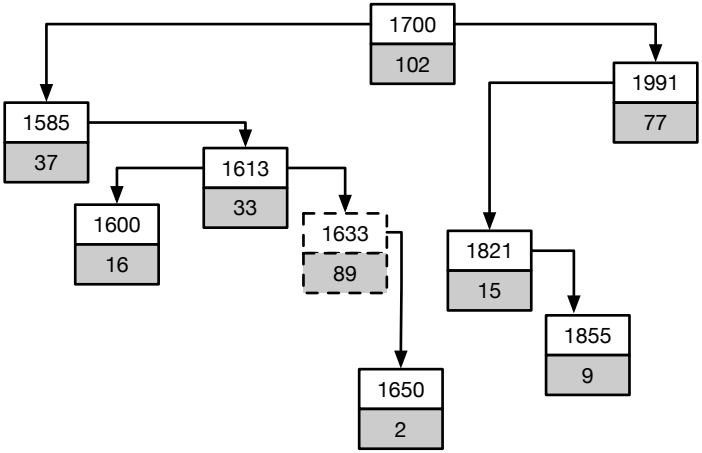
- Первичная вставка проводится в BST. При этом может быть нарушено свойство BH.
- Если вставленный элемент не нарушает свойства BH, то вставка завершена.
- Если свойство BH нарушается, проводится вращение, поднимающее вставленный элемент.
- Подъём происходит до тех пор, пока нарушено свойство BH.

Декартовы деревья: вставка элемента



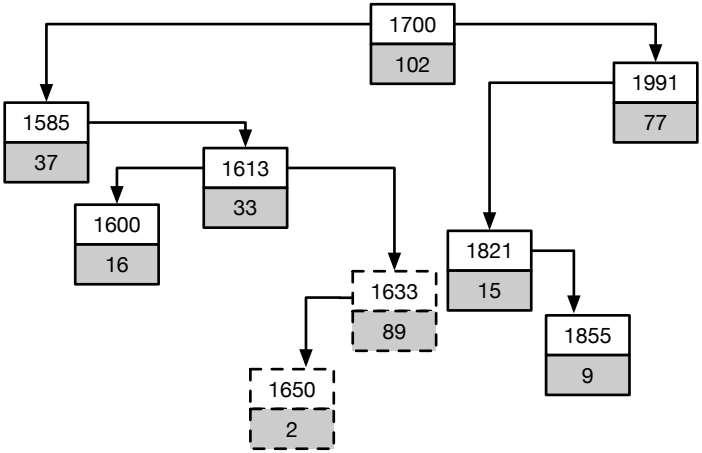
Декартовы деревья: вставка элемента

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



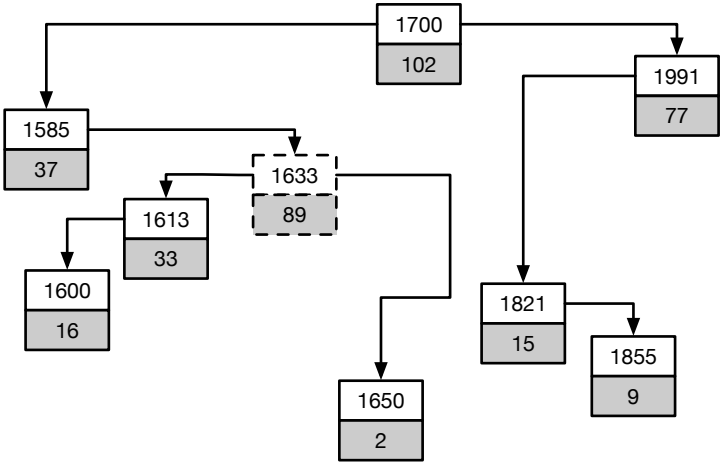
Декартовы деревья: вставка элемента

Попытка обмена с родителем нарушает свойства BST.



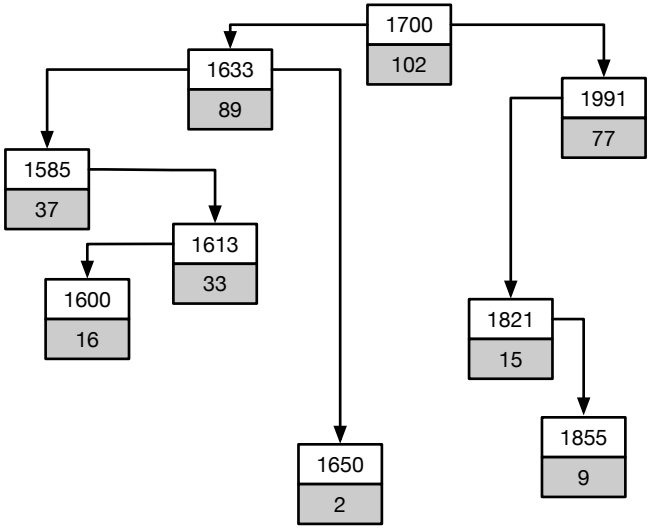
Декартовы деревья: вставка элемента

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



Декартовы деревья: вставка элемента

Ещё одно вращение в сторону родителя и все свойства восстановлены.



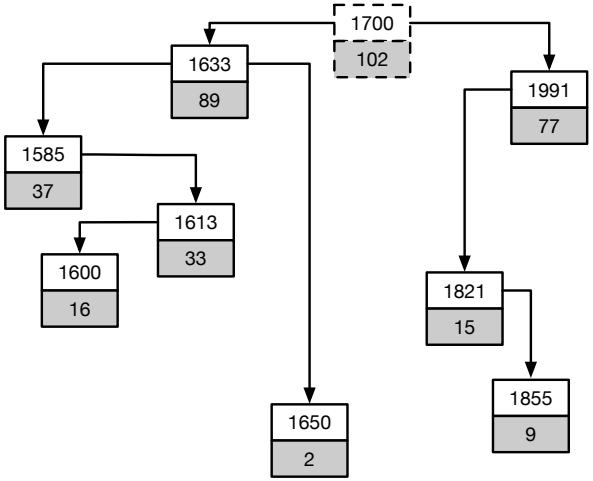
Декартовы деревья: операции

remove — Декартово дерево есть BST + BH.

- 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 y до тех пор, пока он не станет терминальной вершиной.
- На этапе спуска мы не обращаем внимания на сохранение свойства BH, нас интересуют только значения y.

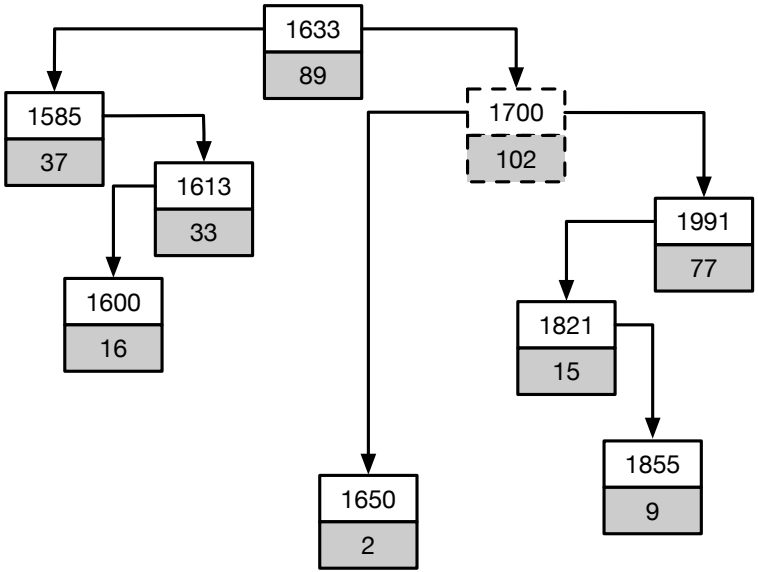
Декартовы деревья: удаление элемента

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



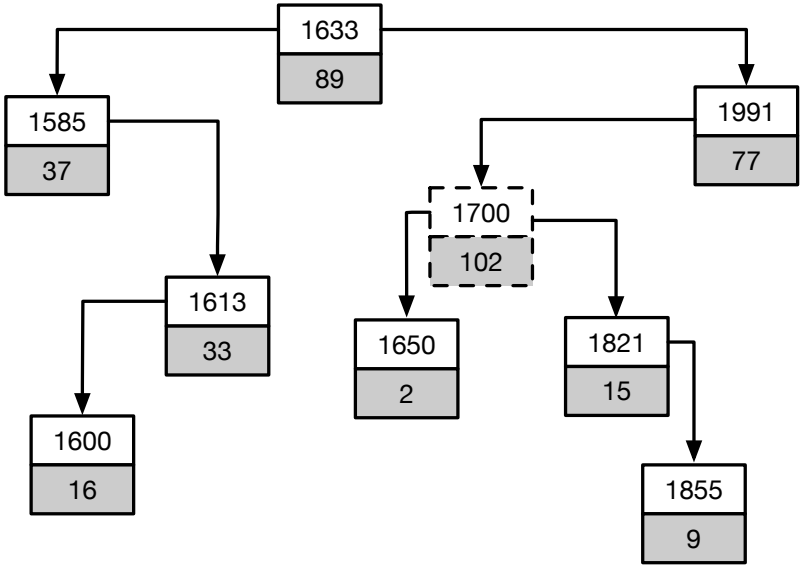
Декартовы деревья: удаление элемента

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



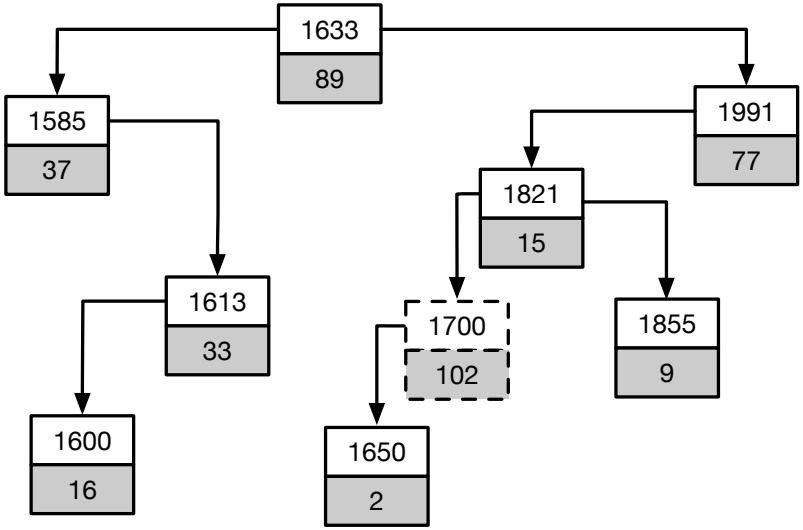
Декартовы деревья: удаление элемента

Следующее направление — узел (1821,15).



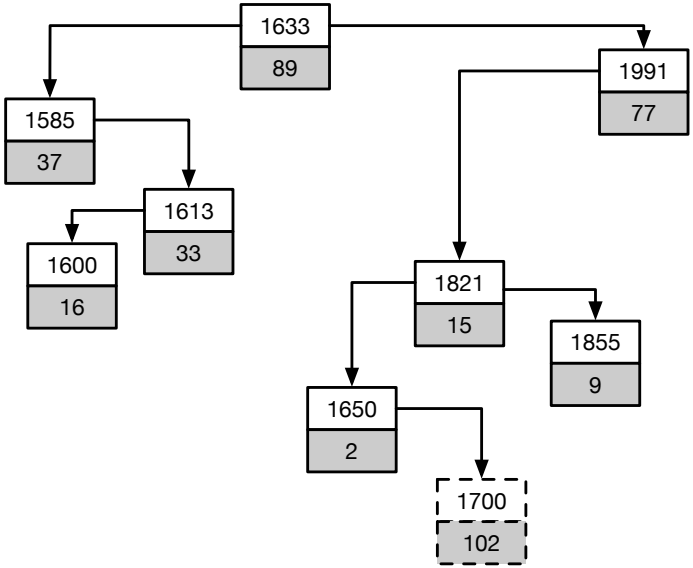
Декартовы деревья: удаление элемента

Последнее направление — узел (1650,2).



Декартовы деревья: удаление элемента

Удаляемый узел добрался до вершин и может быть удалён.



Декартовы деревья: удаление элемента

Заключительное состояние.