Алгоритмы и структуры данных

Структуры данных

д.т.н., проф. Трифонов Петр Владимирович

Содержание лекции

1. Элементарные структуры
2. Хеш-таблицы
3. Двоичные деревья поиска
4. Красно-черные деревья

Стек

- Последним пришел — первым вышел
- Может быть реализован на базе массива или списка
- При реализации на базе массива необходимо предусмотреть обработку переполнения

Algorithm 1: StackEmpty

```
return top==0
```

Algorithm 2: Push(x)

```
S[top++]=x
```

Algorithm 3: Pop

```
if StackEmpty then
    error
else
    return S[- -top]
```

Очередь

- Первым пришел — первым вышел
- Может быть реализован на базе массива или списка
- При реализации на базе массива необходимо предусмотреть обработку переполнения

Algorithm 4: Enqueue(x)

```
Q[tail]=x;
if tail=n-1 then
    tail=0
else
    tail++
```

Algorithm 5: Dequeue

```
if head=tail then
    Error
x=Q[head];
if head=n-1 then
    head=0
else
    head++
```

Связанные списки

- Список — структура данных, в которой элементы расположены в линейном порядке, задаваемом указателями
- Односвязный (однонаправленный) список: каждый элемент содержит указатель на следующий (м.б. NULL)
- Двусвязный (двунаправленный) список: каждый элемент содержит указатель на следующий (м.б. NULL) и предыдущий (м.б. NULL) элементы
- Доступ к элементам по указателям (итераторам)
- Список может быть сколь угодно длинным; не надо заранее выделять память на максимальное число элементов
- Простые процедуры вставки и удаления элемента, слияния списков
- Значительные издержки на хранение указателей
- Выделение и освобождение памяти на отдельные элементы может быть трудоемким
- Применяются гибридные структуры (список массивов)

Деревья

- Элемент дерева - структура, содержащая указатели на структуры, соответствующие дочерним узлам и, иногда, родительский узел
- Лист дерева: указатели на дочерние структуры = NULL
- Корень дерева: указатель на структуру родителя= NULL
- Иногда применяется представление дерева в виде массива (см. кучу)

Хеш-таблицы

$$H[x] = y$$

Операции:

- Добавление элемента
- Поиск элемента
- Удаление элемента

Прямая адресация

- ключи $x \in U = \{0, \ldots, m-1\}$, m мало
- H — массив размерности m
- Сложность операций $O(1)$

Хеширование с цепочками

- Пусть множество ключей U велико, или число реально присутствующих записей мало
- Построим функцию $h : U \to \{0, 1, \ldots, m-1\}$
- В массиве $T[0..m-1]$ хранятся указатели на начало списков пар (x, y)
- Добавление элемента y с ключом x: вставить (x, y) в начало списка $T[h(x)]$: сложность $O(1)$
- Поиск элемента с ключом x: перебор элементов в списке $T[h(x)]$
- Удаление элемента с ключом x: найти и удалить элемент в списке $T[h(x)]$

Анализ хеширования с цепочками

- Пусть дана хеш-таблица с m позициями, в которой хранятся n элементов
- Коэффициент заполнения $\alpha = n/m$
- Если хеш-значения всех элементов совпадают, поиск сводится к поиску в списке со сложностью $O(n)$
- Гипотеза равномерного хеширования: $P\{h(x) = i\} = 1/m, 0 \leq i < m$, $h(x)$ – независимые СВ
- Если верна ГРХ, поиск элемента, отсутствующего в таблице, требует просмотра в среднем α элементов и имеет среднюю сложность $\Theta(1+\alpha)$

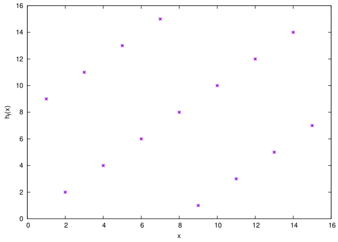
  ГРХ $\Rightarrow$ Среднее время поиска — среднее время просмотра одного списка; сложность вычисления $h(x)$ — $\Theta(1)$
- Если верна ГРХ, среднее время успешного поиска элемента в таблице — $\Theta(1+\alpha)$

  Среднее время поиска равно сумме позиций всех элементов в списках, поделенной на n

$$C = \Theta\left(\frac{1}{n}\sum_{i=0}^{n-1}(1+\frac{i}{m})\right) = \Theta\left(1+\frac{1}{mn}\frac{n(n-1)}{2}\right) = \Theta(1+\alpha/2-1/2m) = \Theta(1+\alpha)$$

Хеш-функции

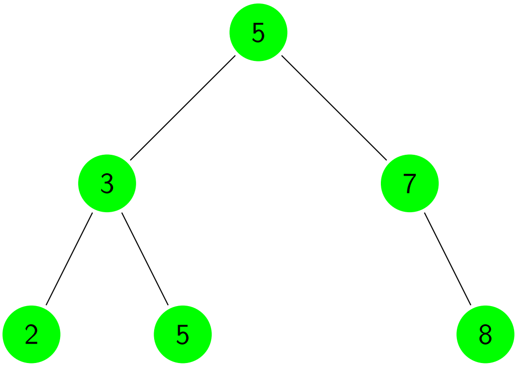
- Распределение x, как правило, неизвестно. В хеш-таблице могут храниться зависимые случайные величины
- Обычно выбирают $h(x)$ так, чтобы ее поведение не коррелировало с возможными закономерностями в данных
- Пусть ключи — натуральные числа.
  - $h(x) = x \bmod m$, m — простое число, далекое от 2^p
  - Хеш-функция Фибоначчи: $h_f(x) = xA \bmod m, m = 2^p$, $A = \lfloor m(\sqrt{5}-1)/2 \rfloor$



- Строки и иные многобайтные объекты могут рассматриваться как длинные целые числа
- Универсальное хеширование: хеш-функция выбирается случайным образом из

Двоичное дерево поиска

- Двоичное дерево, в котором каждый узел имеет поля key,value,left,right,parent
- left,right — указатели на дочерние узлы
- parent — указатель на родительский узел
- Если x — узел ДДП, а y находится в его левом поддереве, то $y.key \preceq x.key$. Если y находится в правом поддереве, то $x.key \preceq y.key$



Центрированный (симметричный) обход дерева

- Вывод всех элементов дерева в отсортированном порядке
- Для дерева с n узлами сложность $T(n) = T(k) + T(n-k-1) + \delta = \Theta(n)$ k — число элементов в левом поддереве, δ — сложность If, print

Algorithm 6: InorderTreeWork(x)

```
if x ≠ NULL then
    InorderTreeWork(x.left);
    print(x.key,x.value);
    InorderTreeWork(x.right);
```

Поиск элемента с заданным ключом

- Более эффективной является итеративная реализация
- Сложность $O(h)$, где h — высота дерева

Algorithm 7: TreeSearch(x,k)

if $x = NULL \lor x.key = k$ **then**
  return x
if $k<x.key$ **then**
  TreeSearch(x.left,k)
else
  TreeSearch(x.right,k)

Поиск минимума и максимума

- Сложность $O(h)$, где h — высота дерева

Algorithm 8: TreeMin(x)

```
while x.left ≠ NULL do
    x=x.left
return x
```

Algorithm 9: TreeMax(x)

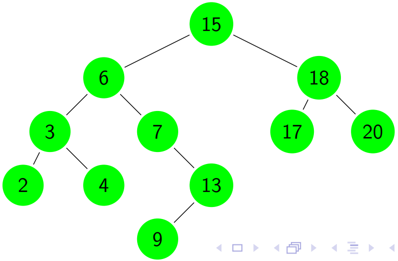
```
while x.right ≠ NULL do
    x=x.right
return x
```

Поиск следующего элемента

- Найти элемент, следующий за x в отсортированной последовательности
- Если правое поддерево x непусто, искомое значение — крайний левый узел в правом поддереве
- Если правое поддерево пусто и у x имеется следующий за ним элемент y, то y — наименьший предок x, чей левый наследник также является предком x
- Сложность $O(h)$, где h — высота дерева

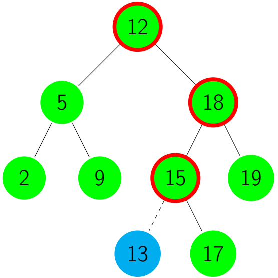
Algorithm 10: TreeSuccessor(x)

if $x.right \neq NULL$ **then**
    return *TreeMinimum(x.right)*
y=x.parent;
while $y \neq NULL \wedge x = y.right$ **do**
    x=y;y=y.parent
return y



Вставка элемента в дерево

- Вставка в дерево с корнем T узла с ключом k, значением v
- Будем считать, что ключи уникальны
- Возвращает корень модифицированного дерева
- Сложность $O(h)$, где h — высота дерева

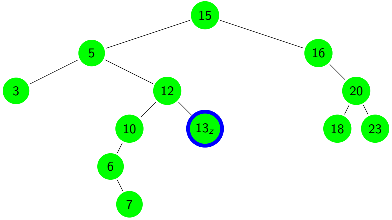


Algorithm 11: TreeInsert(T,k,v)

```
z.key=k;z.value=v;z.left=NULL;z.right=NULL;y=NULL
 x=T;
while x ≠ NULL do
    y=x;
    if z.key<x.key then
        x=x.left
    else
        x=x.right
z.parent=y;
if y=NULL then
    T=z
else
    if z.key<y.key then
        y.left=z
    else
        y.right=z
return z
```

Удаление элемента из дерева

- Удаление из дерева с корнем T узла z
- Возвращает корень модифицированного дерева
- Возможные случаи
  - у z нет дочерних узлов: изменить его родительский узел



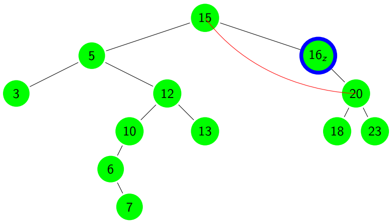
- Сложность $O(h)$, где h — высота дерева

Algorithm 12: TreeDelete(T,z)

```
y=(z.left = NULL ∨ z.right = NULL)?z: TreeSuccessor(z);
x=(y.left ≠ NULL)?y.left:y.right;
if x ≠ NULL then
    x.parent=y.parent
if y.parent=NULL then
    T=x
else
    if y=y.parent.left then
        y.parent.left=x
    else
        y.parent.right=x
if y ≠ x then
    z.key=y.key;z.value=y.value
delete y;
return T
```

Удаление элемента из дерева

- Удаление из дерева с корнем T узла z
- Возвращает корень модифицированного дерева
- Возможные случаи
  - один дочерний узел у z: переключить родительский узел на дочерний узел z

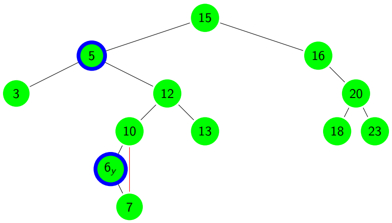


Algorithm 13: TreeDelete(T,z)

y=($z.left$ $=$ $NULL \vee z.right$ $=$ $NULL$)?z: TreeSuccessor(z);
x=($y.left \neq NULL$)?y.left:y.right;
if $x \neq NULL$ **then**
    x.parent=y.parent
if $y.parent{=}NULL$ **then**
    T=x
else
    if $y{=}y.parent.left$ **then**
        y.parent.left=x
    else
        y.parent.right=x
if $y \neq x$ **then**
    z.key=y.key;z.value=y.value
delete y;
return T

Удаление элемента из дерева

- Удаление из дерева с корнем T узла z
- Возвращает корень модифицированного дерева
- Возможные случаи
  - 2 дочерних узла z: найти следующий за ним узел без левого дочернего узла, заменить им узел z

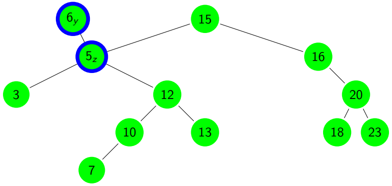


Algorithm 14: TreeDelete(T,z)

y=($z.left = NULL \lor z.right = NULL$)?z: TreeSuccessor(z);
x=($y.left \neq NULL$)?y.left:y.right;
if $x \neq NULL$ **then**
    x.parent=y.parent
if $y.parent{=}NULL$ **then**
    T=x
else
    if $y{=}y.parent.left$ **then**
        y.parent.left=x
    else
        y.parent.right=x
if $y \neq x$ **then**
    z.key=y.key;z.value=y.value
delete y;
return T

Удаление элемента из дерева

- Удаление из дерева с корнем T узла z
- Возвращает корень модифицированного дерева
- Возможные случаи
  - 2 дочерних узла z: найти следующий за ним узел без левого дочернего узла, заменить им узел z



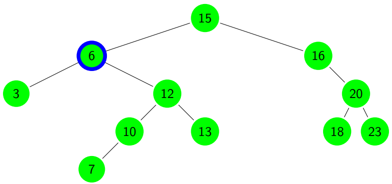
- Сложность $O(h)$, где h — высота дерева

Algorithm 15: TreeDelete(T,z)

```
y=(z.left = NULL ∨ z.right = NULL)?z: TreeSuccessor(z);
x=(y.left ≠ NULL)?y.left:y.right;
if x ≠ NULL then
    x.parent=y.parent
if y.parent=NULL then
    T=x
else
    if y=y.parent.left then
        y.parent.left=x
    else
        y.parent.right=x
if y ≠ x then
    z.key=y.key;z.value=y.value
delete y;
return T
```

Удаление элемента из дерева

- Удаление из дерева с корнем T узла z
- Возвращает корень модифицированного дерева
- Возможные случаи
  - 2 дочерних узла z: найти следующий за ним узел без левого дочернего узла, заменить им узел z



- Сложность $O(h)$, где h — высота дерева

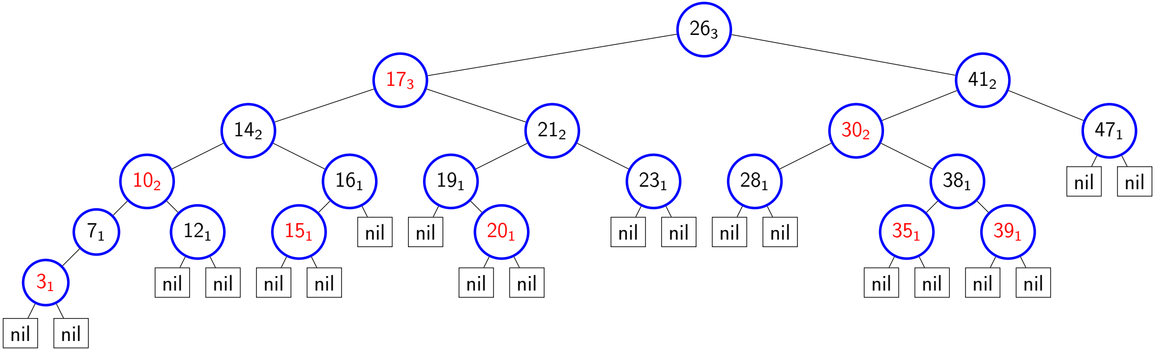
Algorithm 16: TreeDelete(T,z)

```
y=(z.left = NULL ∨ z.right = NULL)?z: TreeSuccessor(z);
x=(y.left ≠ NULL)?y.left:y.right;
if x ≠ NULL then
    x.parent=y.parent
if y.parent=NULL then
    T=x
else
    if y=y.parent.left then
        y.parent.left=x
    else
        y.parent.right=x
if y ≠ x then
    z.key=y.key;z.value=y.value
delete y;
return T
```

Красно-черные деревья

- Красно-черное дерево — двоичное дерево поиска, вершины которого окрашены в красный или черный цвет
  - Каждая вершина либо черная, либо красная
  - Каждый лист (nil) черный
  - Корень дерева черный
  - Если вершина красная, оба ее ребенка черные
  - Все пути, ведущие от корня к листьям, содержат одинаковое число черных вершин
- Узел — структура данных, имеющая поля:
  - $color \in \{red, black\}$
  - left,right — указатели на дочерние узлы
  - p — указатель на родительский узел
  - key — ключ
- Введем фиктивные узлы-листья nil. Все такие узлы будем считать тождественными
- Число черных узлов на пути от узла x вниз к листьям (не считая x)— черная высота узла bh(x)
- Ни один путь от корня к листу не отличается по длине от другого более чем в 2 раза

Пример красно-черного дерева



- Нижний индекс узла — его черная высота
- Узлы *nil* имеют черную высоту 0
- Черная высота дерева — черная высота его корня

Высота красно-черного дерева

Лемма

КЧД с n внутренними узлами (не nil) имеет высоту $h \leq 2\log_2(n+1)$

Доказательство.

Докажем, что поддерево с корнем в x содержит не менее $2^{\mathrm{bh}(x)} - 1$ внутренних вершин.
Для листьев (nil) $\mathrm{bh}(x) = 0$, поддерево содержит $1 > 2^0 - 1$ вершин.
Пусть утверждение верно для $x : 0 \leq \mathrm{bh}(x) < k$. Пусть x — не лист и $\mathrm{bh}(x) = k$. Его дочерние узлы $x_s, s \in \{0, 1\}$ имеют $\mathrm{bh}(x_s) \geq k - 1 \Rightarrow$ являются корнями поддеревьев с $\geq 2^{k-1} - 1$ узлами $\Rightarrow x$ — корень поддерева с $\geq 1 + 2(2^{k-1} - 1) = 2^k - 1$ узлами.

Высота красно-черного дерева

Лемма

КЧД с n внутренними узлами (не nil) имеет высоту $h \leq 2\log_2(n+1)$

Доказательство.

Докажем, что поддерево с корнем в x содержит не менее $2^{\mathrm{bh}(x)} - 1$ внутренних вершин.
Для листьев (nil) $\mathrm{bh}(x) = 0$, поддерево содержит $1 > 2^0 - 1$ вершин.
Пусть утверждение верно для $x : 0 \leq \mathrm{bh}(x) < k$. Пусть x — не лист и $\mathrm{bh}(x) = k$. Его дочерние узлы $x_s, s \in \{0, 1\}$ имеют $\mathrm{bh}(x_s) \geq k - 1 \Rightarrow$ являются корнями поддеревьев с $\geq 2^{k-1} - 1$ узлами $\Rightarrow x$ — корень поддерева с $\geq 1 + 2(2^{k-1} - 1) = 2^k - 1$ узлами. Пусть y — корень всего КЧД. *Если вершина красная, оба ее ребенка черные*$\Rightarrow \mathrm{bh}(y) \geq h/2 \Rightarrow n \geq 2^{h/2} - 1 \Rightarrow h \leq 2\log_2(n+1)$ □

Высота красно-черного дерева

Лемма

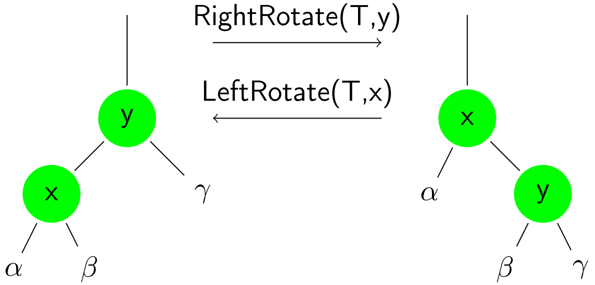
КЧД с n внутренними узлами (не nil) имеет высоту $h \leq 2\log_2(n+1)$

Доказательство.

Докажем, что поддерево с корнем в x содержит не менее $2^{\mathrm{bh}(x)} - 1$ внутренних вершин.
Для листьев (nil) $\mathrm{bh}(x) = 0$, поддерево содержит $1 > 2^0 - 1$ вершин.
Пусть утверждение верно для $x : 0 \leq \mathrm{bh}(x) < k$. Пусть x — не лист и $\mathrm{bh}(x) = k$. Его дочерние узлы $x_s, s \in \{0, 1\}$ имеют $\mathrm{bh}(x_s) \geq k - 1 \Rightarrow$ являются корнями поддеревьев с $\geq 2^{k-1} - 1$ узлами $\Rightarrow x$ — корень поддерева с $\geq 1 + 2(2^{k-1} - 1) = 2^k - 1$ узлами. Пусть y — корень всего КЧД. *Если вершина красная, оба ее ребенка черные*$\Rightarrow \mathrm{bh}(y) \geq h/2 \Rightarrow n \geq 2^{h/2} - 1 \Rightarrow h \leq 2\log_2(n+1)$ □

Для КЧД операции поиска, нахождения минимального, максимального и следующего элементов имеют сложность $O(\log n)$

Вращения



Если для всякого узла z:

$$\forall u \in z.left, v \in z.right : u.key \preceq z.key \preceq v.key,$$

то это свойство сохраняется и после вращения

Algorithm 17: LeftRotate(T,x)

```
y=x.right;
x.right=y.left;
if y.left ≠ nil then
    y.left.p=x
y.p=x.p;
if x.p=nil then
    T.root=y
else
    if x=x.p.left then
        x.p.left=y
    else
        x.p.right=y
y.left=x;x.p=y
```

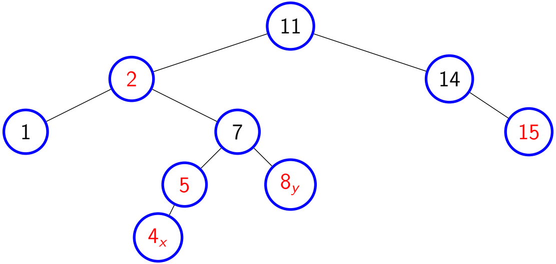
Сложность $O(1)$

Вставка узла в КЧД

Algorithm 18: RBInsert(T,k,v)

```
x=TreeInsert(T,k,v);x.color=red;
while x ≠ T.root ∧ x.p.color = red do
    if x.p=x.p.p.left then
        y=x.p.p.right;
        if y.color=red then
            x.p.color=black;y.color=black;
            x.p.p.color=red;x=x.p.p
        else
            if x=x.p.right then
                x=x.p;LeftRotate(T,x)
            x.p.color=black;x.p.p.color=red;
            RightRotate(T,x.p.p)
    else
        аналогично с заменой left ↔ right
T.root.color=black
```

На каждой итерации цикла ровно одна красная вершина x может иметь красного родителя

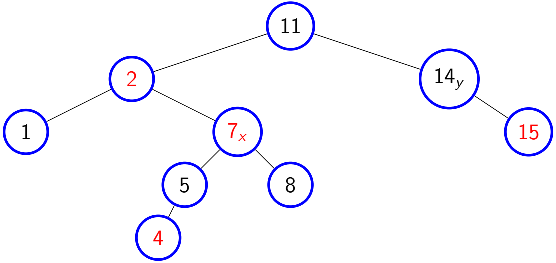


Вставка узла в КЧД

Algorithm 19: RBInsert(T,k,v)

```
x=TreeInsert(T,k,v);x.color=red;
while x ≠ T.root ∧ x.p.color = red do
    if x.p=x.p.p.left then
        y=x.p.p.right;
        if y.color=red then
            x.p.color=black;y.color=black;
            x.p.p.color=red;x=x.p.p
        else
            if x=x.p.right then
                x=x.p;LeftRotate(T,x)
            x.p.color=black;x.p.p.color=red;
            RightRotate(T,x.p.p)
    else
        аналогично с заменой left ↔ right
T.root.color=black
```

На каждой итерации цикла ровно одна красная вершина x может иметь красного родителя

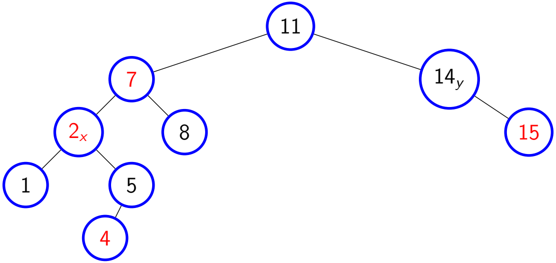


Вставка узла в КЧД

Algorithm 20: RBInsert(T,k,v)

```
x=TreeInsert(T,k,v);x.color=red;
while x ≠ T.root ∧ x.p.color = red do
    if x.p=x.p.p.left then
        y=x.p.p.right;
        if y.color=red then
            x.p.color=black;y.color=black;
            x.p.p.color=red;x=x.p.p
        else
            if x=x.p.right then
                x=x.p;LeftRotate(T,x)
            x.p.color=black;x.p.p.color=red;
            RightRotate(T,x.p.p)
    else
        аналогично с заменой left ↔ right
T.root.color=black
```

На каждой итерации цикла ровно одна красная вершина x может иметь красного родителя

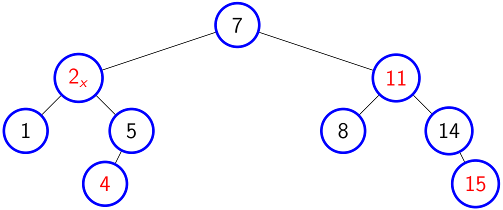


Вставка узла в КЧД

Algorithm 21: RBInsert(T,k,v)

```
x=TreeInsert(T,k,v);x.color=red;
while x ≠ T.root ∧ x.p.color = red do
    if x.p=x.p.p.left then
        y=x.p.p.right;
        if y.color=red then
            x.p.color=black;y.color=black;
            x.p.p.color=red;x=x.p.p
        else
            if x=x.p.right then
                x=x.p;LeftRotate(T,x)
            x.p.color=black;x.p.p.color=red;
            RightRotate(T,x.p.p)
    else
        аналогично с заменой left ↔ right
T.root.color=black
```

На каждой итерации цикла ровно одна красная вершина x может иметь красного родителя



- Высота КЧД, содержащего n элементов: $h = O(\log n)$
- Сложность *TreeInsert*: $O(h) = O(\log n)$
- Число итераций не более h
- Общая сложность $O(\log n)$
- Число вращений не более 2

Удаление узла из КЧД

- Возможные случаи
  - у z нет дочерних узлов: изменить его родительский узел
  - один дочерний узел у z: переключить родительский узел на дочерний узел z
  - 2 дочерних узла z: найти следующий за ним узел без левого дочернего узла, заменить им узел z
- nil — узлы-ограничители (есть поле .p). Как правило, достаточно единственного экземпляра nil, используемого многократно
- Если y — красный:
  - Никакая черная высота в дереве не меняется
  - Никакие красные узлы не становятся смежными
  - корень остается черным
- Аргумент x RBDeleteFixup: узел, бывший единственным потомком y перед его извлечением, или ограничитель nil

Algorithm 22: RBDelete(T,z)

```
if z.left = nil ∨ z.right = nil then
    y=z
else
    y=TreeSuccessor(z)
if y.left ≠ nil then
    x=y.left
else
    x=y.right
x.p=y.p;
if y.p=nil then
    T.root=x
else
    if y=y.p.left then
        y.p.left=x
    else
        y.p.right=x
    if y ≠ z then
        z.key=y.key;z.value=y.value
    if y.color=black then
        RBDeleteFixup(T,x)
return y
```

Восстановление свойств КЧД после удаления

- Если извлеченный из дерева y был корнем, а корнем стал его красный потомок, то это нарушение
- Если x и $y.p = x.p$ были красными, у красного узла нет 2 черных потомков
- Удаление y приводит к тому, что все проходящие через него пути имеют на один черный узел меньше. Это нарушение

Algorithm 23: RBDeleteFixup(T,x)

```
while x ≠ T.root ∧ x.color = black do
    if x=x.p.left then
        w=x.p.right;
        if w.color=red then
            w.color=black;x.p.color=red;
            LeftRotate(T,x.p);w=x.p.right
        if w.left.color = black ∧ w.right.color = black then
            w.color=red;x=x.p
        else
            if w.right.color=black then
                w.left.color=black;w.color=red;
                RightRotate(T,w);w=x.p.right
            w.color=x.p.color;x.p.color=black;
            w.right.color=black;
            LeftRotate(T,x.p);x=T.root
    else
        аналогичные действия с left ↔ right
x.color=black
```

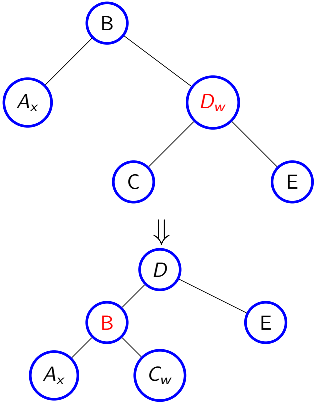
Восстановление свойств КЧД после удаления

- При извлечении из дерева x передадим его черноту его потомку x
  - дважды черный узел: ко всем путям, проходящим через него, прибавим 1 к числу черных узлов
  - красно-черный узел: такой цвет не предусмотрен КЧД
- x — дважды черный узел
- Переместим черноту по дереву вверх или повернем его поддеревья, пока не выполнится одно из условий:
  1. x указывает на красно-черный узел: перекрасим его в черный (стр. 18)
  2. x указывает на корень: уберем лишнюю черноту

Algorithm 24: RBDeleteFixup(T,x)

```
1  while x ≠ T.root ∧ x.color = black do
2      if x=x.p.left then
3          w=x.p.right;
4          if w.color=red then
5              w.color=black;x.p.color=red;
6              LeftRotate(T,x.p);w=x.p.right
7          if w.left.color = black ∧ w.right.color = black then
8              w.color=red;x=x.p
9          else
10             if w.right.color=black then
11                 w.left.color=black;w.color=red;
12                 RightRotate(T,w);w=x.p.right
13             w.color=x.p.color;x.p.color=black;
14             w.right.color=black;
15             LeftRotate(T,x.p);x=T.root
16     else
17         аналогичные действия с left ↔ right
18 x.color=black
```

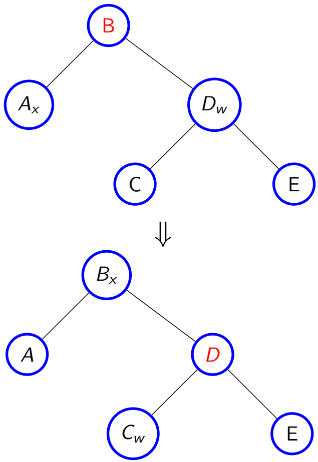
Восстановление свойств КЧД после удаления



Algorithm 25: RBDeleteFixup(T,x)

while $x \neq T.root \wedge x.color = black$ **do**
    if *x=x.p.left* **then**
        w=x.p.right;
        if *w.color=red* **then**
            w.color=black;x.p.color=red;
            LeftRotate(T,x.p);w=x.p.right
        if $w.left.color = black \wedge w.right.color = black$ **then**
            w.color=red;x=x.p
        else
            if *w.right.color=black* **then**
                w.left.color=black;w.color=red;
                RightRotate(T,w);w=x.p.right
            w.color=x.p.color;x.p.color=black;
            w.right.color=black;
            LeftRotate(T,x.p);x=T.root
    else
        аналогичные действия с *left* $\leftrightarrow$ *right*
x.color=black

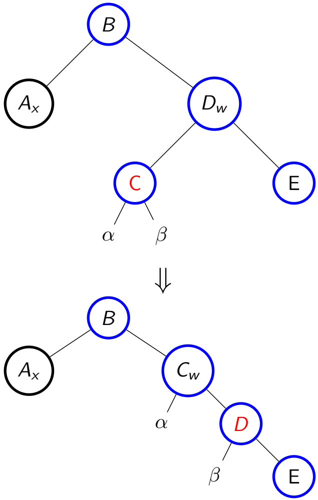
Восстановление свойств КЧД после удаления



Algorithm 26: RBDeleteFixup(T,x)

```
while x ≠ T.root ∧ x.color = black do
    if x=x.p.left then
        w=x.p.right;
        if w.color=red then
            w.color=black;x.p.color=red;
            LeftRotate(T,x.p);w=x.p.right
        if w.left.color = black ∧ w.right.color = black then
            w.color=red;x=x.p
        else
            if w.right.color=black then
                w.left.color=black;w.color=red;
                RightRotate(T,w);w=x.p.right
            w.color=x.p.color;x.p.color=black;
            w.right.color=black;
            LeftRotate(T,x.p);x=T.root
    else
        аналогичные действия с left ↔ right
x.color=black
```

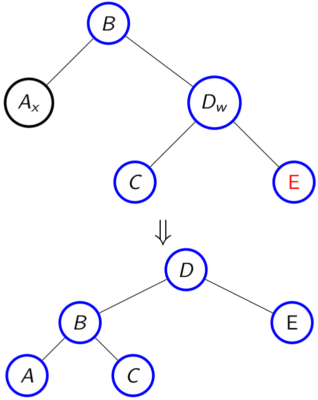
Восстановление свойств КЧД после удаления



Algorithm 27: RBDeleteFixup(T,x)

```
while x ≠ T.root ∧ x.color = black do
    if x=x.p.left then
        w=x.p.right;
        if w.color=red then
            w.color=black;x.p.color=red;
            LeftRotate(T,x.p);w=x.p.right
        if w.left.color = black ∧ w.right.color = black then
            w.color=red;x=x.p
        else
            if w.right.color=black then
                w.left.color=black;w.color=red;
                RightRotate(T,w);w=x.p.right
            w.color=x.p.color;x.p.color=black;
            w.right.color=black;
            LeftRotate(T,x.p);x=T.root
    else
        аналогичные действия с left ↔ right
x.color=black
```

Восстановление свойств КЧД после удаления



Algorithm 28: RBDeleteFixup(T,x)

```
while x ≠ T.root ∧ x.color = black do
    if x=x.p.left then
        w=x.p.right;
        if w.color=red then
            w.color=black;x.p.color=red;
            LeftRotate(T,x.p);w=x.p.right
        if w.left.color = black ∧ w.right.color = black then
            w.color=red;x=x.p
        else
            if w.right.color=black then
                w.left.color=black;w.color=red;
                RightRotate(T,w);w=x.p.right
            w.color=x.p.color;x.p.color=black;
            w.right.color=black;
            LeftRotate(T,x.p);x=T.root
    else
        аналогичные действия с left ↔ right
x.color=black
```

Восстановление свойств КЧД после удаления

- Количество черных вершин от корня дерева до его поддеревьев с корнями A, C, E не меняется
- На каждой итерации происходит $x = x.p$ или $x = T.root \Rightarrow$ число итераций не превосходит высоту дерева $h = O(\log n)$

Algorithm 29: RBDeleteFixup(T,x)

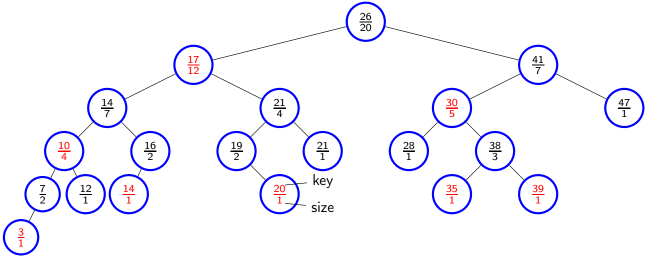
```
while x ≠ T.root ∧ x.color = black do
    if x=x.p.left then
        w=x.p.right;
        if w.color=red then
            w.color=black;x.p.color=red;
            LeftRotate(T,x.p);w=x.p.right
        if w.left.color = black ∧ w.right.color = black then
            w.color=red;x=x.p
        else
            if w.right.color=black then
                w.left.color=black;w.color=red;
                RightRotate(T,w);w=x.p.right
            w.color=x.p.color;x.p.color=black;
            w.right.color=black;
            LeftRotate(T,x.p);x=T.root
    else
        аналогичные действия с left ↔ right
x.color=black
```

Динамические порядковые статистики

- Быстрый поиск i-ой порядковой статистики
- Дерево порядковой статистики — КЧД, у которого каждый узел содержит дополнительное поле *size*, равное количеству внутренних узлов в его поддереве; $nil.size = 0$

$$x.size = x.left.size + x.right.size + 1$$

- OSSelect(T.root,i) возвращает i-ый в порядке возрастания ключа узел дерева, $i \geq 0$
- Сложность $O(h) = O(\log n)$



Algorithm 30: OSSelect(x,i)

```
r=x.left.size;
if i=r then
    return x
else if i<r then
    return OSSelect(x.left,i)
else
    return OSSelect(x.right,i-r)
```

Определение ранга элемента

- По указателю на x найти его порядковый номер при центрированном обходе дерева
- Инвариант: на каждой итерации r равно рангу $x.key$ в поддереве, корнем которого является y
  - Перед первой итерацией инвариант обеспечивается строкой 1
  - В строке 5 прибавляется число элементов из левого "братского"поддерева и 1 (родительский узел). Все эти элементы предшествуют x
  - Если x было в левом поддереве, его ранг в родительском дереве не меняется

Algorithm 31: OSRank(T,x)

```
1 r=x.left.size;
2 y=x;
3 while y ≠ T.root do
4     if y=y.p.right then
5         r=r+y.p.left.size+1;
6         y=p.y
7     return r
```

Сложность $O(\log n)$

Поддержание правильного значения size

- Вставка нового узла в дерево
  1. Новый узел подвешивается в качестве листа. x.size=1
  2. В последующей процедуре коррекции (цикл while в RBInsert) структурные изменения происходят только при вращении (не более 2)
  3. При одном вращении значения size становятся неправильными только в 2 узлах
  4. Дополнение к LeftRotate (аналогично в RightRotate): y.size=x.size;x.size=x.left.size+x.right.size+1
- Удаление узла
  1. Уменьшить .size у всех предков удаленного узла до корня включительно
  2. Использовать вышеописанное дополнение к LeftRotate, RightRotate
- Сложность по-прежнему $O(\log n)$

Выводы

- Двоичное дерево поиска позволяет осуществлять поиск элемента в дереве, нахождение минимума и максимума, нахождение следующего элемента, вставку и удаление со сложностью $O(h)$, где h — высота дерева
- Двоичное дерево поиска может стать несбалансированным с высотой $h = O(n)$
- Красно-черное дерево — надстройка над двоичным деревом поиска, обеспечивающая $h = O(\log n)$