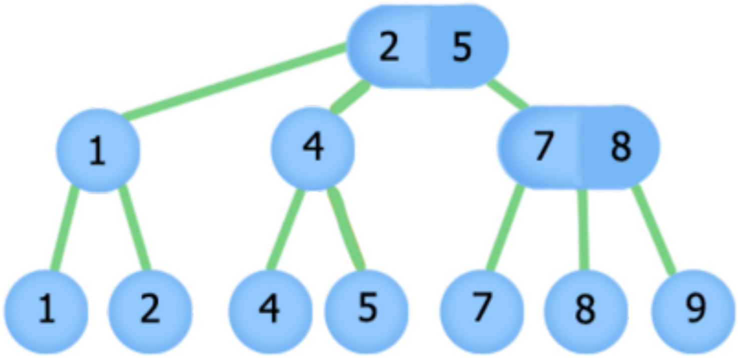
2-3 дерево

2-3 дерево (англ. *2-3 tree*) — структура данных, представляющая собой сбалансированное дерево поиска, такое что из каждого узла может выходить две или три ветви и глубина всех листьев одинакова. Является частным случаем B+ дерева.



Пример 2-3 дерева

Содержание

Свойства

2-3 дерево — сбалансированное дерево поиска, обладающее следующими свойствами:

- нелистовые вершины имеют либо 2, либо 3 сына,
- нелистовая вершина, имеющая двух сыновей, хранит максимум левого поддерева. Нелистовая вершина, имеющая трех сыновей, хранит два значения. Первое значение хранит максимум левого поддерева, второе максимум центрального поддерева,
- сыновья упорядочены по значению максимума поддерева сына,
- все листья лежат на одной глубине,
- высота 2-3 дерева $O(\log n)$, где n — количество элементов в дереве.

Операции

Введем следующие обозначения:

- $\mathtt{root}$ — корень 2-3 дерева.

Каждый узел дерева обладает полями:

- $\mathtt{parent}$ — родитель узла,
- $\mathtt{sons}$ — сыновья узла,
- $\mathtt{keys}$ — ключи узла,
- $\mathtt{length}$ — количество сыновей.

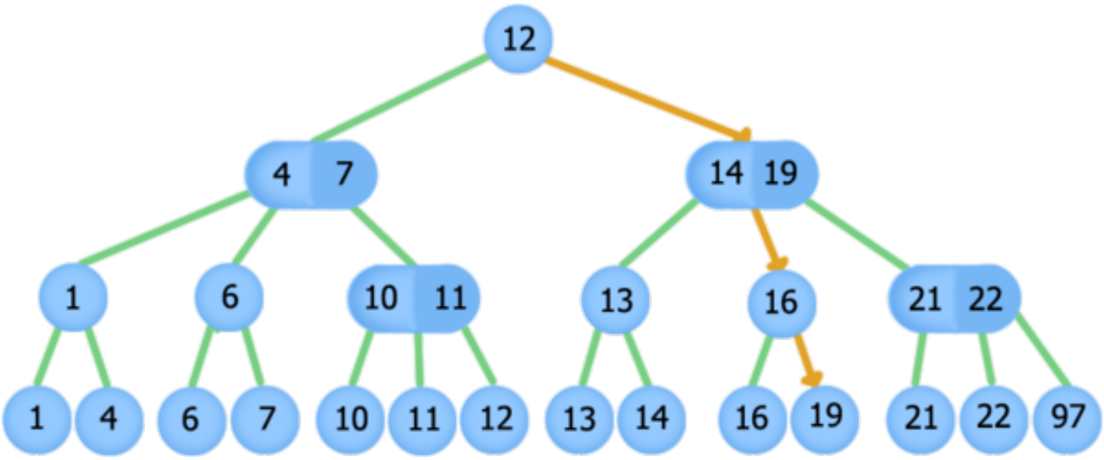
Поиск

- x — искомое значение,
- t — текущая вершина в дереве.

Изначально $t = \mathtt{root}$. Будем просматривать ключи в узлах, пока узел не является листом. Рассмотрим два случая:

- у текущей вершины два сына. Если её значение меньше x, то $t = \mathtt{t.sons[1]}$, иначе $t = \mathtt{t.sons[0]}$.
- у текущей вершины три сына. Если второе значение меньше x, то $t = \mathtt{t.sons[2]}$. Если первое значение меньше x, то $t = \mathtt{t.sons[1]}$, иначе $t = \mathtt{t.sons[0]}$.

```
T search(T x):
  Node t = root
  while (t не является листом)
    if (t.length == 2)
      if (t.keys[0] < x)
        t = t.sons[1]
      else
        t = t.sons[0]
    else if (t.keys[1] < x)
      t = t.sons[2]
    else if (t.keys[0] < x)
      t = t.sons[1]
    else
      t = t.sons[0]
  return t.keys[0]
```



Поиск элемента 19, оранжевые стрелки обозначают путь по дереву при поиске

Вставка элемента

- x — добавляемое значение,
- t — текущая вершина в дереве. Изначально $t = \mathtt{root}$.

Если корня не существует — дерево пустое, то новый элемент и будет корнем (одновременно и листом). Иначе поступим следующим образом:

Найдем сперва, где бы находился элемент, применив $\mathtt{search(x)}$. Далее проверим есть ли у этого узла родитель, если его нет, то в дереве всего один элемент — лист. Возьмем этот лист и новый узел, и создадим для них родителя, лист и новый узел расположим в порядке возрастания.

Если родитель существует, то подвесим к нему ещё одного сына. Если сыновей стало 4, то разделим родителя на два узла, и повторим разделение теперь для его родителя, ведь у него тоже могло быть уже 3 сына, а мы разделили и у него стало на 1 сына больше. (перед разделением обновим ключи).

```
function splitParent(Node t):
 if (t.length > 3)
   Node a = Node(sons = {t.sons[2], t.sons[3]}, keys = {t.keys[2]}, parent = t.parent, length = 2)
   t.sons[2].parent = a
   t.sons[3].parent = a
   t.length = 2
   t.sons[2] = null
   t.sons[3] = null
   if (t.parent != null)
     t.parent[t.length] = a
```

```
    t.length++
    сортируем сыновей у t.parent
    splitParent(t.parent)
  else                   // мы расщепили корень, надо подвесить его к общему родителю, который будет новым корнем
   Node t = root
   root.sons[0] = t
   root.sons[1] = a
   t.parent = root
   a.parent = root
   root.length = 2
   сортируем сыновей у root
```

Если сыновей стало 3, то ничего не делаем. Далее необходимо восстановить ключи на пути от новой вершины до корня:

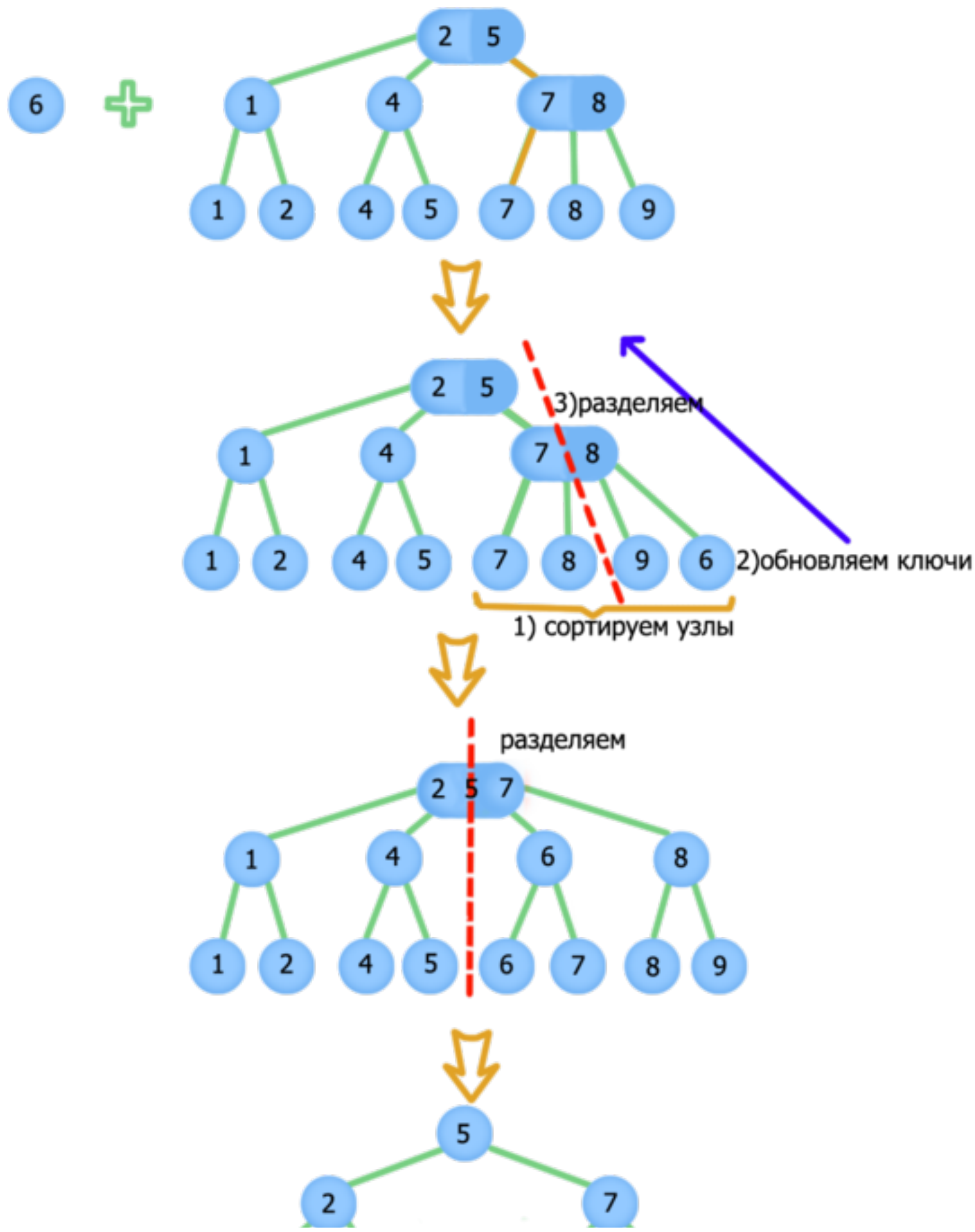
```
function updateKeys(Node t):
  Node a = t.parent
  while (a != null)
   for i = 0 .. a.length - 1
     a.keys[i] = max(a.sons[i]) // max — возвращает максимальное значение в поддереве.
   a = a.parent                 // Примечание: max легко находить, если хранить максимум
                                // правого поддерева в каждом узле — это значение и будет max(a.sons[i])
```

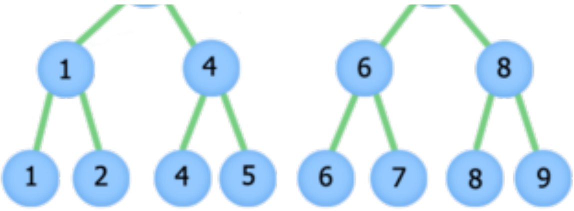
$\mathtt{updateKeys}$ необходимо запускать от нового узла. Добавление элемента:

```
function insert(T x):
  Node n = Node(x)
  if (root == null)
   root = n
   return
  Node a = searchNode(x)
  if (a.parent == null)
    Node t = root
    root.sons[0] = t
    root.sons[1] = n
    t.parent = root
    n.parent = root
    root.length = 2
    сортируем сыновей у root
  else
    Node p = a.parent
    p.sons[p.length] = n
    p.length++
    n.parent = p
    сортируем сыновей у p
    updateKeys(n)
    split(n)
  updateKeys(n)
```

Так как мы спускаемся один раз, и поднимаемся вверх при расщеплении родителей не более одного раза, то $\mathtt{insert}$ работает за $O(\log n)$.

Примеры добавления:


3)разделяем
2)обновляем ключи
1) сортируем узлы
разделяем



Добавление элемента с ключом 6

Удаление элемента

- x — значение удаляемого узла,
- t — текущий узел,
- b — брат t,
- p — отец t,
- np — соседний брат p,
- gp — отец p.

Пусть изначально $t = \mathtt{searchNode(x)}$ — узел, где находится x.

Если у t не существует родителя, то это корень (одновременно и единственный элемент в дереве). Удалим его.

Если p существует, и у него строго больше 2 сыновей, то просто удалим t, а у p уменьшим количество детей.

Если у родителя t два сына, рассмотрим возможные случаи (сперва везде удаляем t):

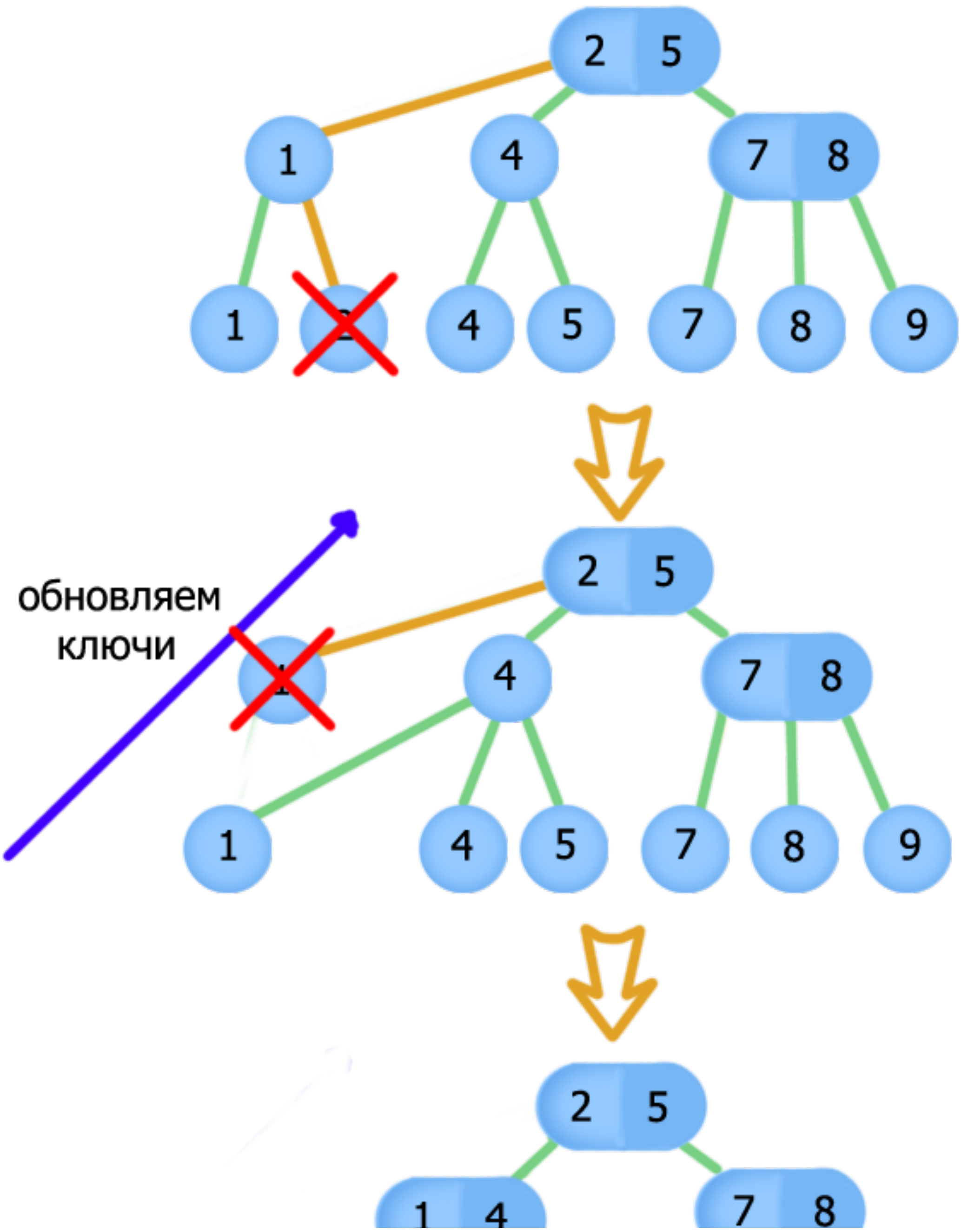
- np не существует, тогда мы удаляем одного из сыновей корня, следовательно, другой сын становится новым корнем,
- у gp оказалось 2 сына, у np оказалось 2 сына. Подвесим b к np и удалим p. Так как у gp — родителя p, оказалось тоже два сына,повторяем для p такие же рассуждения,
- у gp оказалось 2 или 3 сына, у np оказалось 3 сына. Просто заберем ближайшего к нам сына у np и прицепим его к p. Восстановим порядок в сыновьях p. Теперь у p оказалось снова два сына и все узлы 2-3 дерева корректны,
- у gp оказалось 3 сына, у np оказалось 2 сына. Подвесим b к np и удалим p, а у gp уменьшим количество детей. Так как у np оказалось три сына, а у gp все ещё больше одного сына, то все узлы 2-3 дерева корректны.

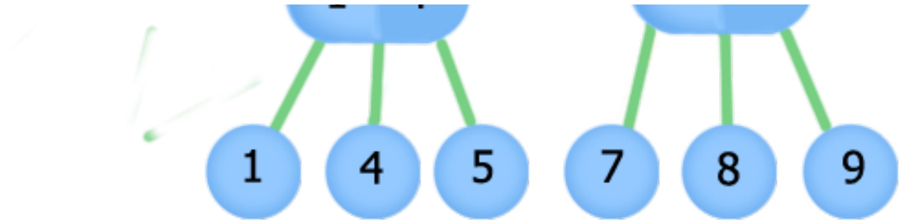
Обобщим алгоритм при удалении когда у родителя t два сына:

- Если np не существует, то оказывается, что мы сейчас удаляем какого-то из сыновей корня (для определенности далее левого, с правым аналогично). Тогда теперь правый сын становится корнем. На этом удаление заканчивается.

- Если np существует, то удалим t, а его брата (b) перецепим к np. Теперь у np могло оказаться 4 сына, поэтому повторим аналогичные действия из $\mathtt{insert}$: вызовем $\mathtt{updateKeys}(b)$ и $\mathtt{splitParent}(np)$. Теперь рекурсивно удалим p.

В результате мы получаем корректное по структуре 2-3 дерево, но у нас есть нарушение в ключах в узлах, исправим их с помощью $\mathtt{updateKeys()}$, запустившись от b.


2 5
1
4
7 8
1
4
5
7
8
9
обновляем
ключи
2 5
4
7 8
1
4
5
7
8
9
2 5
1 4
7 8



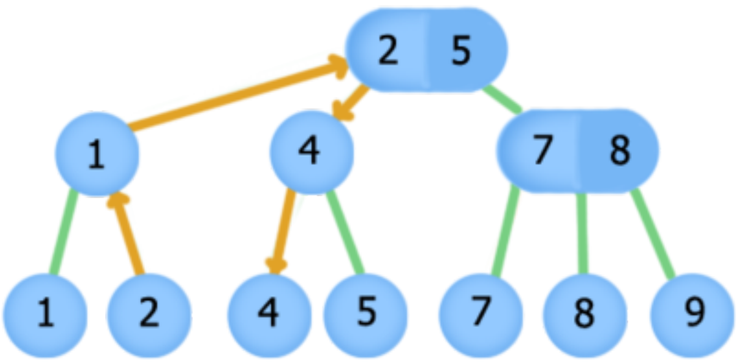
Удаление элемента с ключом 2

Следующий и предыдущий

- x — поисковый параметр,
- t — текущий узел.

В силу того, что наши узлы отсортированы по максимуму в поддереве, то следующий объект — это соседний лист справа. Попасть туда можно следующим образом: будем подниматься вверх, пока у нас не появится первой возможности свернуть направо вниз. Как только мы свернули направо вниз, будем идти всегда влево. Таким образом, мы окажемся в соседнем листе. Если мы не смогли ни разу свернуть направо вниз, и пришли в корень, то следующего объекта не существует. Случай с предыдущим симметричен.

```
T next(T x):
  Node t = searchNode(x)
  if (t.keys[0] > x)  //x не было в дереве, и мы нашли следующий сразу
    return t.keys[0]
  while (t != null)
    t = t.parent
    if (можно свернуть направо вниз)
     в t помещаем вершину, в которую свернули
     while (пока t — не лист)
      t = t.sons[0]
    return t
  return t.keys[0]
```



Путь при поиске следующего элемента после 2

Нахождение m следующих элементов

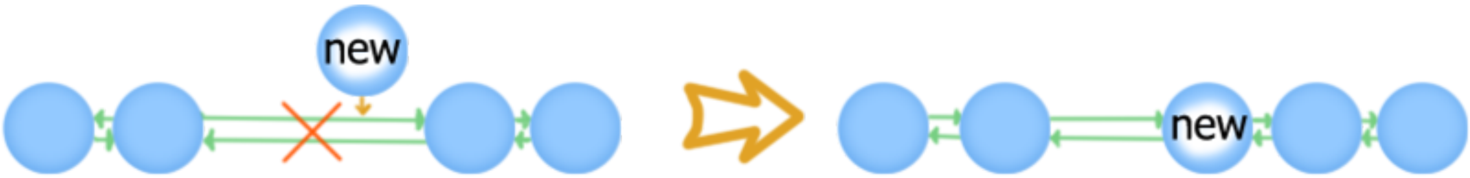
B+ деревья, поддерживают операцию `find`, которая позволяет находить m следующих элементов. Наивная реализация выглядит следующим образом: будем вызывать m раз поиск следующего элемента, такое решение работает за $O(m \log n)$. Но 2-3 деревья, позволяют находить m следующих элементов за $O(m + \log n)$, что значительно ускоряет поиск при больших m. По построению, все листья у нас отсортированы в порядке возрастания, воспользуемся этим для нахождения m элементов. Нам необходимо связать листья, для этого модифицируем `insert` и `delete`. Добавим к узлам следующие поля:

- `right` — указывает на правый лист,
- `left` — указывает на левый лист.

Пусть t — добавленный узел. Изменим `insert` следующим образом: в самом конце, после того как мы уже обновили все ключи, найдем `next(t)` и запишем ссылку на него в `t.right`. Аналогично с левым.

Пусть t — удаляемый узел. Изменим `delete` следующим образом: в самом начале, до удаления t, найдем следующий `next` и запишем в `next.left` правый лист относительно t. С левым поступим аналогично.

В итоге, мы имеем двусвязный список в листьях, и чтобы нам вывести m элементов, нам достаточно один раз найти нужный элемент и пробежаться вправо на m элементов.



thumb

См. также

- B-дерево
- Splay-дерево
- АВЛ-дерево
- Декартово дерево
- Красно-черное дерево

Источники информации

- is.ifmo.ru — Визуализатор 2-3 дерева (http://is.ifmo.ru/vis/tree23/tree23_ru.html)
- rain.ifmo.ru — Визуализатор 2-3 дерева (http://rain.ifmo.ru/cat/view.php/vis/trees/2-3-2002)
- Википедия — 2-3 дерево (http://ru.wikipedia.org/wiki/2-3-дерево)
- Д. Кнут «Искусство программирования. Сортировка и поиск» — стр. 508-509

Источник — «http://neerc.ifmo.ru/wiki/index.php?title=2-3_дерево&oldid=60391»

- Эта страница последний раз была отредактирована 31 января 2017 в 11:58.