Древовидные структуры данных

Введение. Бинарные деревья поиска

Хотя массивы обеспечивают очень быстрый доступ к своим элементам (O(1)), задача поиска с их использованием решается намного медленнее. Так, для несортированного массива поиск потребует O(N) операций при O(1) на добавление в конец , а для сортированного – $O(\log_2 N)$ (в случае бинарнго поиска) и $O(n^2)$+O(1) (в случае QSort) на добавление.

По этой причине для задач поиска данных (например, в базах данных) используются древовидные структуры данных, которые обладают лучшими оценками сложности.

Простейшая из таких структур – бинарное дерево. Бинарное дерево – это структура данных, которая представляет собой либо пустой элемент, либо элемент с двумя потомками (возможно, пустыми), каждый из которых является бинарным деревом. Корень дерева – это узел, у которого нет родительских элементов. Листья – это элементы, потомки которых – пустые элементы. Данные располагаются внутри элементов дерева. Здесь и далее будем полагать, что в элементах содержатся целые числа.

Бинарное дерево поиска – это бинарное дерево, каждый элемент которого, как правило, дополнен дополнительными полями для осуществления задачи поиска. В алгоритмах поиска дочерние элементы у каждого узла дерева обозначают как правое и левое, полагая, что в правом поддереве содержатся элементы больше хранящегося в данном узле, а в левом – меньше. Из этого определения выводятся простейшие алгоритмы вставки и поиска элемента в дерево.

В стиле C-подобных нотаций будем обозначать как «=» оператор присваивания и как «==» оператор сравнения на равенство. NIL – Это пустой элемент, Tree(X) – функция, создающая дерево из переданного значения, value(x) – значение, хранящееся в узле x, left(x), right(x) – левое и правое поддеревья x.

```
Insert(T,x)
1       If T == NIL
2       then T=Tree(x)
3       else if x≤value(T)
4               then Insert(left(T), x)
5               else Insert(right(T),x)
```

Допускается вариант, когда повторная вставка существующего в дереве элемента не позволяется

```
Search(T,x)
1       If T == NIL then False
2       else If value(T)==x then true
3               else if value(T)<x then Search(Left(T),x)
```

 else Search(Right(T),x)

Удаление элемента – несколько более сложная операция. Если элемент не найден, то ничего не происходит. Если же он найден, то анализируется число его непустых потомков. Если оно равно 0, элемент заменяется на пустой. Если оно равно 1, непустой потомок найденного элемента присваивается как соответствующлее поддерево его родителя. Если оно равно 2, то вместо текущего элемента устанавливается самый правый потомок левого поддерева или самый левый – правого.

Простое бинарное дерево поиска не обладает в общем случае хорошими скоростными характеристиками. Предположим, в дерево производится последовательная вставка возрастающих чисел. В этом случае дерево будет расти только вправо, а значит, операция вставки и поиска будет обладать оценкой O(N). Эта ситуация исправляется с помощью самобалансирующихся деревьев.

Самобалансирующиеся деревья – это такие бинарные деревья поиска, для которых характерно выравнивание баланса после вставки элемента. Под балансом понимается равенство (или приблизительное равенство) высот поддеревьев или производных величин.

АВЛ-дерево

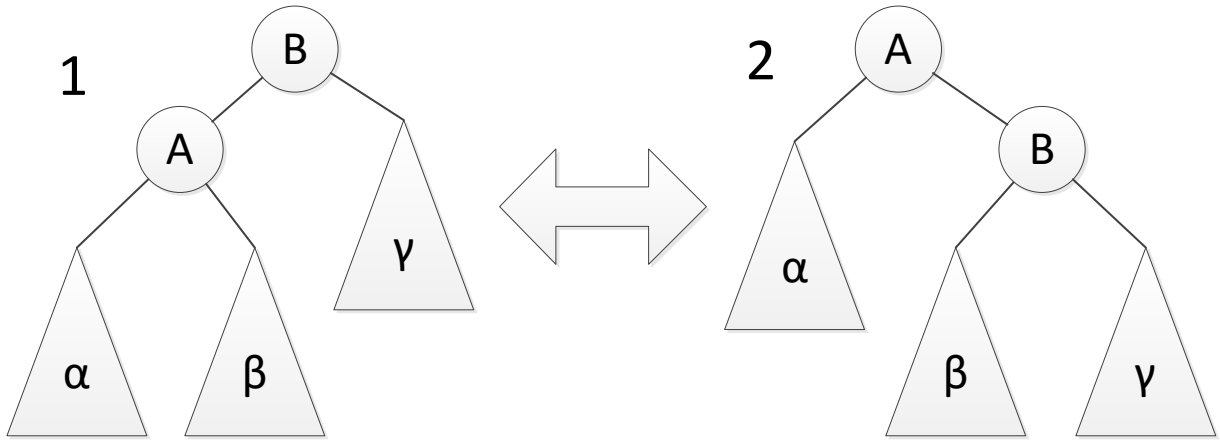
АВЛ-дерево – вариант двоичного дерева поиска. Помимо данных в узле этого дерева хранится информация о разнице высот его левого и правого поддеревьев. Если высота левого поддерева больше, то разница высот считается отрицательной, если высота правого – положительной. АВЛ-дерево считается сбалансированным, если модуль разницы высот для каждого узла не превышает единицу, и идеально сбалансированным, если все разницы высот равны нулю.

Поиск по АВЛ-дереву осуществляется так же, как и по обычному бинарному дереву поиска. В случае вставки возможно нескольких ситуаций, одна из которых приводит к необходимости балансировки дерева.

Пусть у некоторого узла X есть потомки L и R. Без потери общности предположим, что к L добавляется новый узел, и это приводит к увеличению высоты L на 1.

1) Если высота L была меньше высоты R, то высота дерева с корнем X не меняется, а дерево становится более сбалансированным.
2) Если высоты были равны, то дерево становится немного несбалансированным, но это не приводит к необходимости балансировки.
3) Если высота L уже была больше высоты R, то дерево становится ещё более несбалансированным, что приводит к необходимости балансировки.

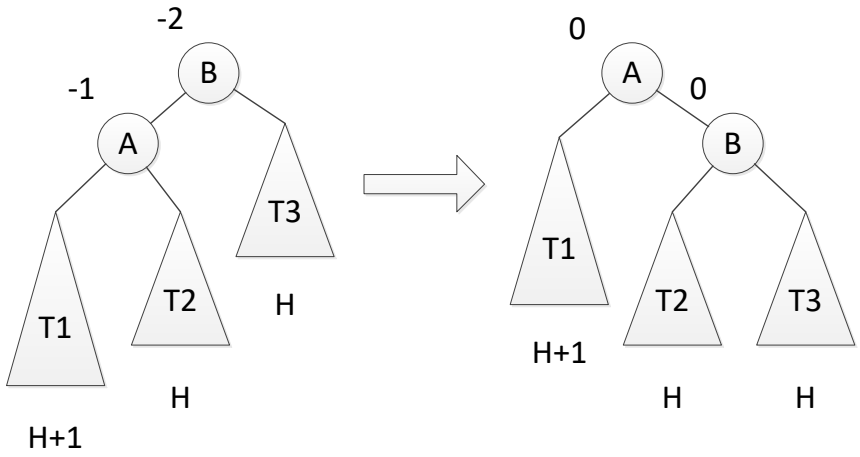
Для восстановления баланса применяются правые и левые вращения узлов, не приводящие к нарушению свойств дерева поиска:



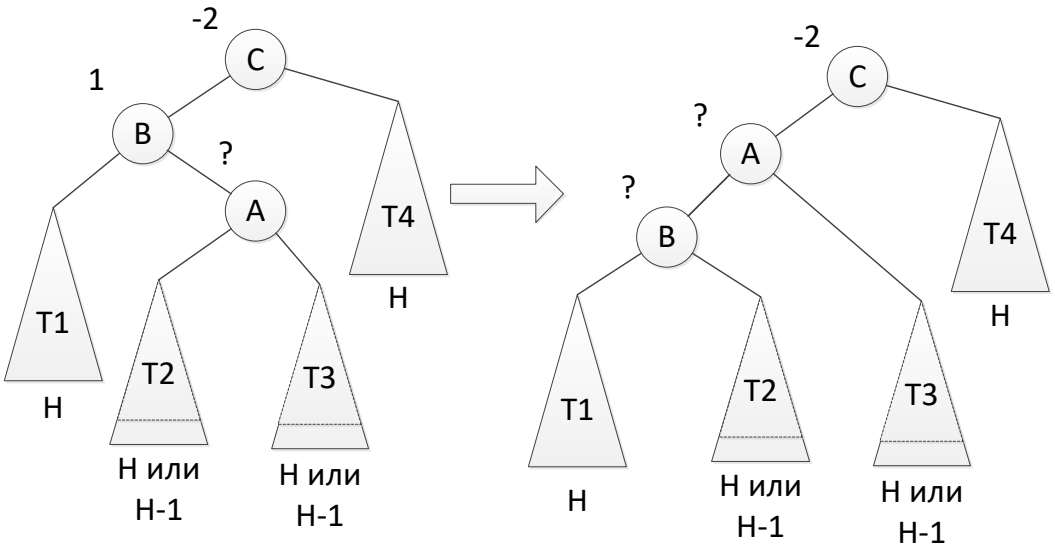
В обоих случаях a∈α, b∈β, c∈γ, a<A<b<B<c. Переход от 1 к 2 называется правым поворотом узла B (RightRotate(B)), переход от 2 к 1 – левым поворотом узла A (LeftRotate(A)).

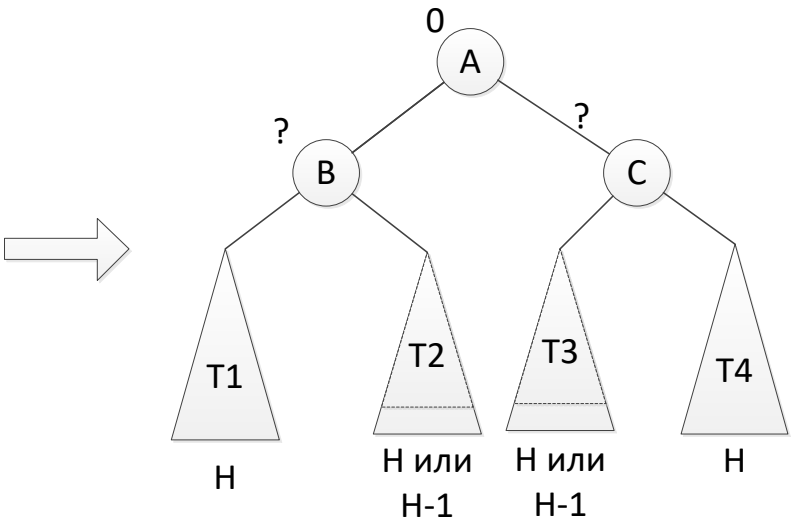
Доказано, что для восстановления баланса требуется либо один, либо два поворота. Выбор того или иного числа поворотов определяется знаками разниц балансов самого нижнего узла с модулем разницы 2 и корня его более высокого поддерева.

Если они равны, то потребуется один поворот:



Если они не равны, то потребуется два поворота. Первый поворот изменяет знаки разниц высот таким образом, чтобы свести этот случай к первому. Второй поворот приводит к балансировке дерева.





Одно из деревьев T2 или T3 обладает высотой H, а другое – H-1. Необходимо отметить, что ситуация, когда и T2, и T3 обладают высотой H-1, не требует балансировки, а ситуация, когда их высота равна H, требует балансировки, но невозможна на практике, так как на одном из предыдущих шагов одно из деревьев обладало высотой H-1, а другое - H.

Разницы высот на обоих шагах приведены в таблице:

H(T2)	H(T3)	PB(A1)	Промежуточный шаг		Окончательные значения	
			PB(A2)	PB(B2)	PB(B3)	PB(C3)
H-1	H	1	-1	-1	-1	0
H	H-1	-1	-2	0	0	1

Осуществима рекурсивная реализация алгоритма АВЛ, которая не требует предварительного вычисления весов по всему дереву, а использует вместе с разницами высот до вставки изменение высот поддеревьев после вставки. Благодаря этому вставка занимает время порядка $O(\log_2 N)$.

Красно-чёрное дерево

Красно-чёрное дерево – ещё один вариант двоичного дерева поиска. В узле дерева помимо данных и ссылок на потомков содержится дополнительный бит информации, отвечающий за цвет узла – красный или чёрный. Кроме того из-за особенностей алгоритма может оказаться полезным хранить в узле ссылку на его родителя. Важно также, что пустые элементы считаются полноценными элементами дерева, и поэтому они являются листьями красно-чёрного дерева.

Для этой структуры данных также определяется понятие чёрной высоты узла (black height, BH) – количество чёрных узлов на простом пути от данного узла до листа. Сам узел в подсчёте высоты не участвует.

Для красно-чёрного дерева справедливы следующие правила:

4) Все узлы красные или чёрные
5) Корень дерева и все листья - чёрные
6) Родителем каждого красного узла является чёрный узел
7) Для каждого узла BH для всех листьев этого поддерева одинакова

Вставка и удаление данных могут привести к тому, что какие-то из этих правил нарушатся. Восстановление справедливости этих правил приводит к балансировке дерева по чёрной высоте. Мы далее рассмотрим только случай вставки.

Можно доказать, что высота дерева $h \leq \log_2(N+1)$. Поскольку красные узлы составляют менее половины всех узлов на путях от корня к листьям, процедура поиска занимает время порядка $O(\log_2 N)$.

Процедура вставки нового элемента в красно-чёрное дерево состоит из двух частей – из обычной вставки в бинарное дерево поиска и перебалансировки дерева. Основная идея перебалансировки состоит в том, чтобы последовательно восстанавливать справедливость 4 правил для поддеревьев на пути от добавленного элемента к корню. Восстановление справедливости правил на уровне корня дерева приводит к восстановлению баланса для всего дерева.

Для восстановления баланса применяются перекраски узлов и правые и левые вращения, рассмотренные ранее.

Далее для произвольного узла X будем обозначать родительский узел - P(X), цвет узла – Color(X). Корень дерева T – Root(T).

Восстановление баланса в процедуре сводится к рассмотрению 6 случаев, которые благодаря симметричности случаев для левого и правого поддеревьев можно свести к 3. Поэтому мы будем рассматривать вставку в левое поддерево, а вставка в правое получается по аналогии, если заменить Left(X) на Right(X) и наоборот.

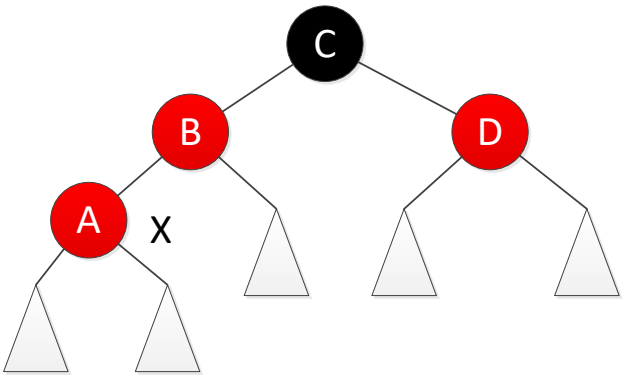
Листинг вставки на псевдокоде выглядит следующим образом:

```
1RB-INSERT(T, x)
2 INSERT(T, x)
3 Color(x) = RED
4 while x ≠ Root(T) and Color(p(x)) = RED
```

```
5       do if p(x) == Left(p(p(x)))
6             then y = Right(p(p(x)))
 7                  if Color(y) == RED then (Случай 1)
8                   else if x == right(p(x))
9                         then (Случай 2)
10                        (Случай 3)
12            else (случай правой вставки)
13      Color(Root(T)) = BLACK
```

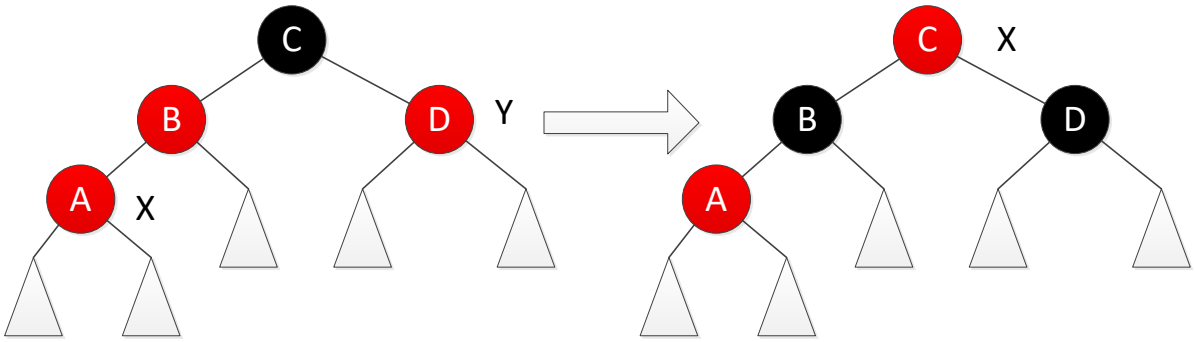
Перебалансировка начинается с покраски вставленного узла в красный цвет (строка 3), тем самым может нарушиться правило 3. Обратим внимание, что у x имеется два чёрных пустых потомка. После чего (строка 4) начинается цикл с предусловием, который анализирует состояние текущего узла. Если узел является корневым, то его поддерево было сбалансировано на предыдущих шагах, а значит, правила 3 и 4 уже выполнены, но его, возможно, надо перекрасить в чёрный цвет, что и делается в строке 13. Анализ состояния родительского узла нужен для того, чтобы определить ситуацию, когда у красного узла красный родитель, что противоречит правилу 3. Как будет показано далее, в случае перекраски в конце в качестве текущего узла x будет присвоен красный узел.

В строке 5 проверяется, является ли родитель x левым потомком своего родителя. Если это так, то мы имеем дерево следующего вида:



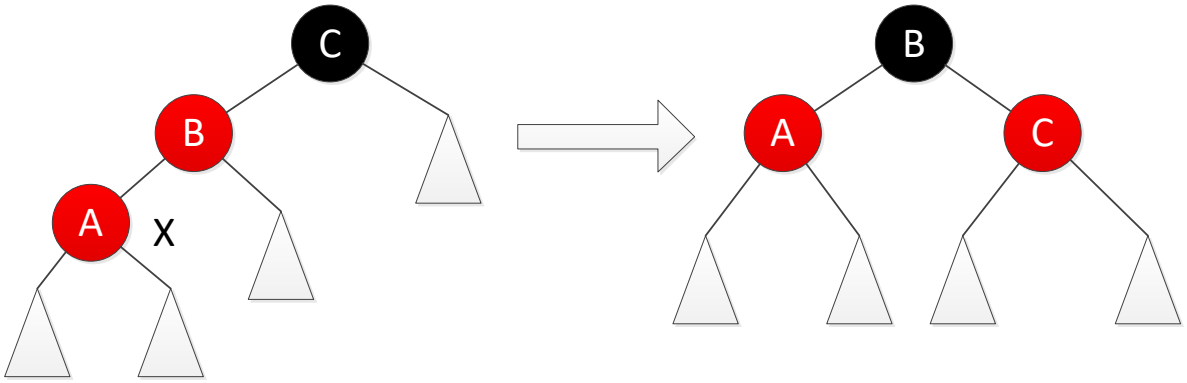
Треугольниками обозначены поддеревья соответствующих узлов. Эти поддеревья обладают одинаковой чёрной высотой. Здесь детей узла B можно поменять местами, оба эти случая учитываются в анализе. Если дерево действительно имеет такой вид, то мы анализируем цвет «дяди» x – в нашем случае это узел D.

Если D – красный (строка 7), то мы можем перенести проблему уровнем выше, перекрасив узлы следующим образом:



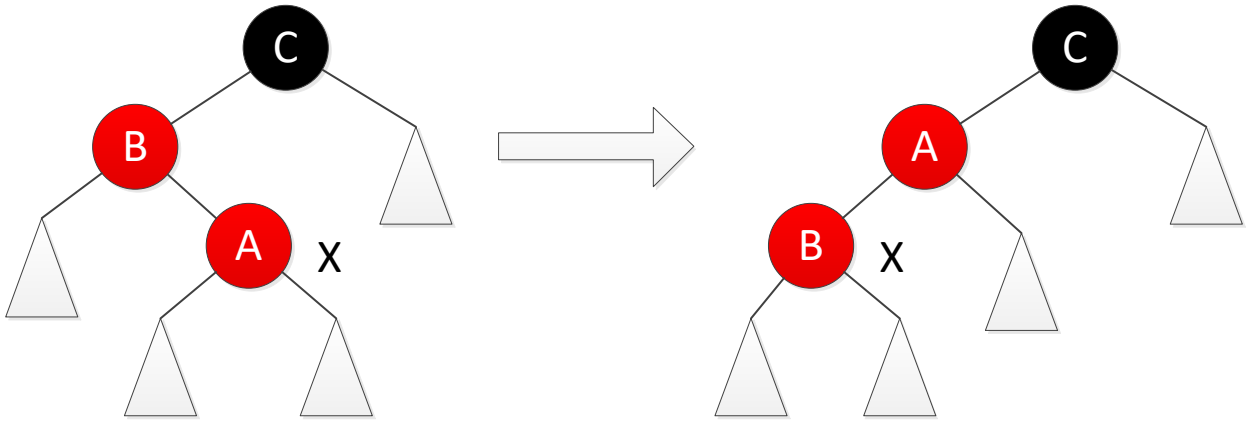
C становится новым узлом X, цикл переходит на следующую итерацию. При этом для поддерева, начинающегося в C, будут выполнены правила 1-4, но, возможно, сам C теперь нарушает какое-либо из правил.

Если же узел D является чёрным, возможно два случая. Первый – ситуация, когда x является левым потомком. В этом случае узел C вращается вправо:



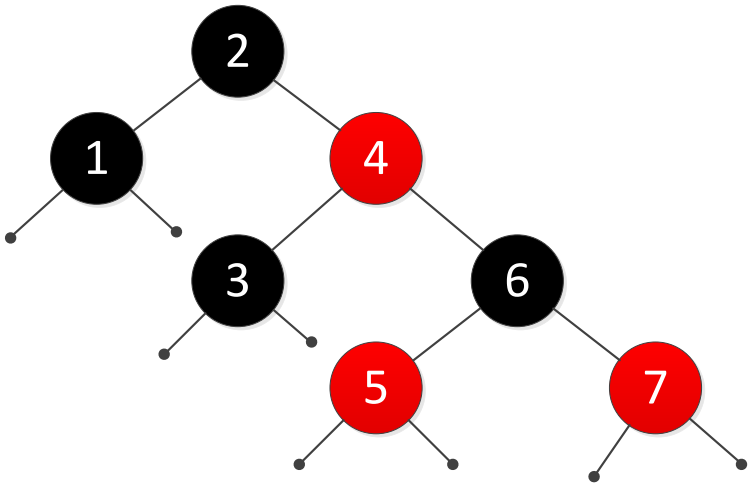
Это приводит к тому, что в дереве больше нет нарушений правил 2-4, поэтому алгоритм завершается.

Если же X был правым потомком (строка 8), узел B вращается влево и становится новым X, и производятся те же действия, что и в предыдущем случае (обратите внимание на то, что на строке 10 нет слова else!):

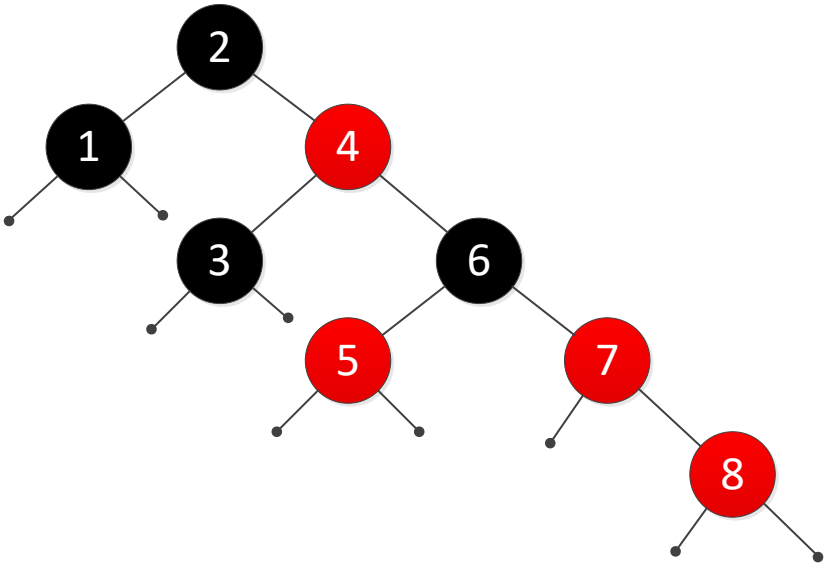


В худшем случае алгоритм требует $O(\log_2 N)$ операция для непосредственно вставки, не более $O(\log_2 N)$ операции перекраски и максимум 2 поворота сложности $O(1)$. Поэтому время вставки можно оценить как $O(\log_2 N)+O(1)$.

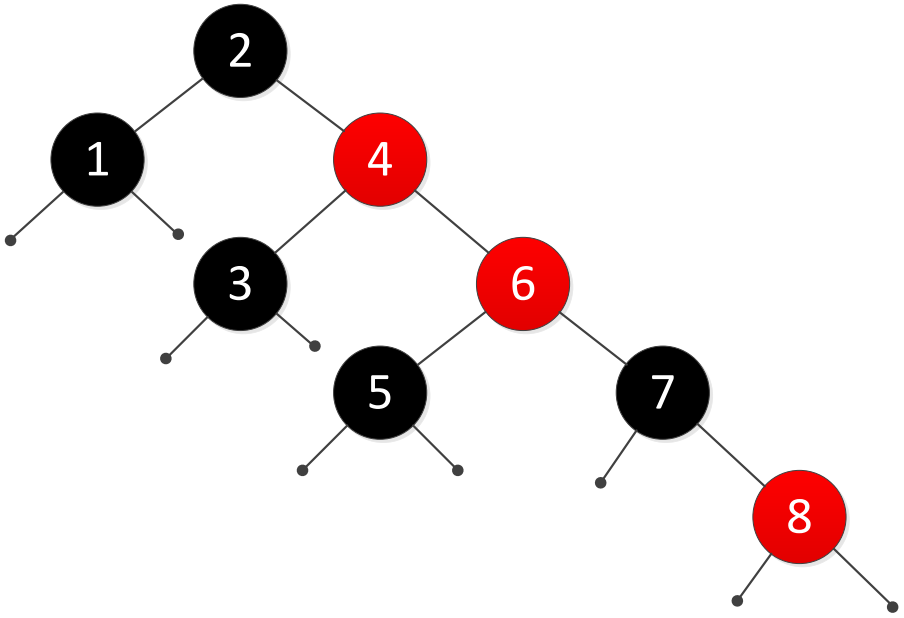
В качестве примера рассмотрим красно-чёрное дерево, полученное при последовательном добавлении чисел от 1 до 7:



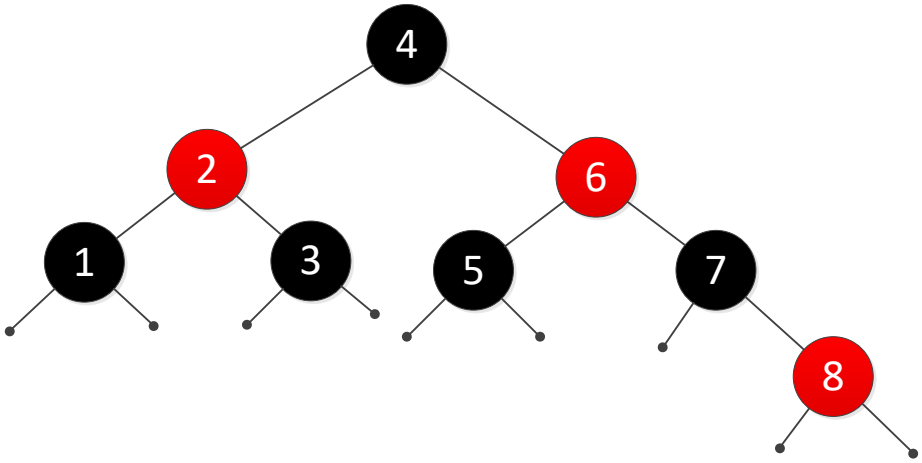
Это дерево, как видно, не является сбалансированным по высоте, однако его чёрная высота одинакова для всех листьев. Добавим 8 и покрасим его в красный цвет.



Правило 3 нарушено – у узла 8 красный родитель. Узлом y в данном случае выступает красный узел 5, поэтому производим перекраску:



Новый узел x – 6. Обратите внимание, что правило 4 не было нарушено – чёрная высота по-прежнему одинакова для всех листьев. Но узел нарушает правило 3. Узел y – чёрный узел 1, следовательно, вращаем узел 2 влево:



Новый узел x – 4. Он является и чёрным, и корнем, значит, вставка завершена. Операция смены цвета корня в данном случае избыточна.

Декартово дерево

Одним из недостатков показанных алгоритмов реализации бинарных деревьев поиска являются сложные с точки зрения программирования процедуры добавления и удаления элементов. Существует структура данных, которая лишена этого недостатка, однако обеспечивающая сбалансированность и логарифмическое время поиска только среднестатистически.

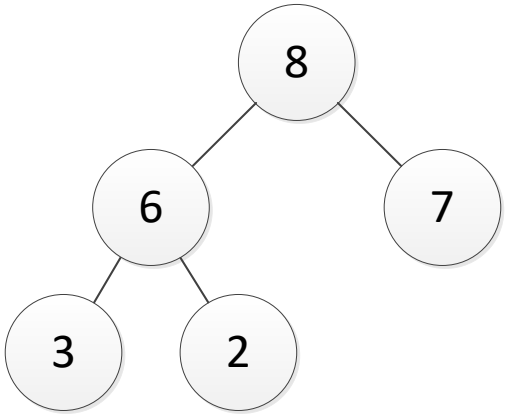
Для начала введём понятие такой структуры данных, как куча. Куча – это бинарное дерево, обладающее следующими свойствами:

1) Значение узла дерева всегда больше значения любого его потомка

2) Назовём слоем совокупность узлов, находящихся на определённой глубине. В куче все слои кроме, возможно, последнего заполнены полностью.

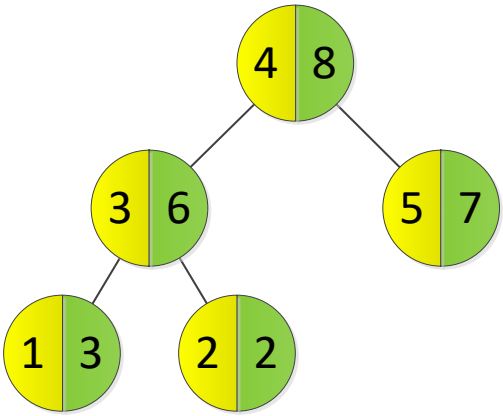
3) Последний слой заполняется слева направо.

Пример кучи показан на рисунке:



Видно, что элементы правого поддерева необязательно больше элементов левого.

Декартовым деревом (в узком смысле синонимично с англ. treap и рус. дерамида) называется такое бинарное дерево поиска, которое помимо данных X, называемых ключом, содержит в своих элементах содержит числовой приоритет Y и которое сформировано таким образом, что является бинарным деревом поиска по ключу и кучей по приоритету. Пример такого дерева показан на рисунке:



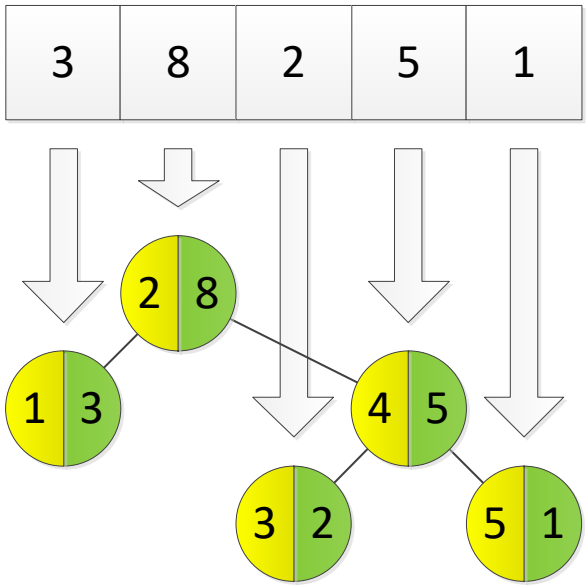
Здесь в жёлтых областях показаны ключи, в зелёных – приоритеты.

Для начала рассмотрим процедуру построения декартова дерева из заданной последовательности ключей. Очевидно, что возможно множество вариантов построения бинарного дерева поиска – от сбалансированных до вырожденных в список. Назначим каждому ключу в последовательности случайный, но уникальный приоритет. Тогда простейший алгоритм построения поддерева выглядит так:

1) Найти элемент с максимальным значением приоритета. Ключ и приоритет этого элемента образуют корень дерева. Если последовательность пустая, вернуть пустой элемент.

2) Выполнить шаг 1) для формирования левого поддерева из левой подпоследовательности и правого – из правой.

На рисунке ключом является номер квадрата, а число в нём – случайно назначенным приоритетом.



Условие уникальности приоритета позволяет однозначно построить декартово дерево из данной последовательности. Однако в результате назначения приоритетов мы можем получить как вырожденные в список деревья, так и идеально сбалансированные для данной последовательности. Доказано, что с высокой долей вероятности высота дерева не будет превышать $4\log_2 N$. Благодаря этому с такой же вероятностью обеспечивается логарифмическое время поиска по дереву.

Алгоритм поиска – такой же, как и в обычном бинарном дереве поиска, приоритеты игнорируются.

Перед рассмотрением добавления к существующему декартову дереву нового элемента, введём две дополнительные операции над деревом. Первая – это слияние деревьев (Merge), то есть объединение двух корректных декартовых дерева в одно. Она возможна только в том случае, когда любой ключ левого дерева меньше любого ключа правого дерева.

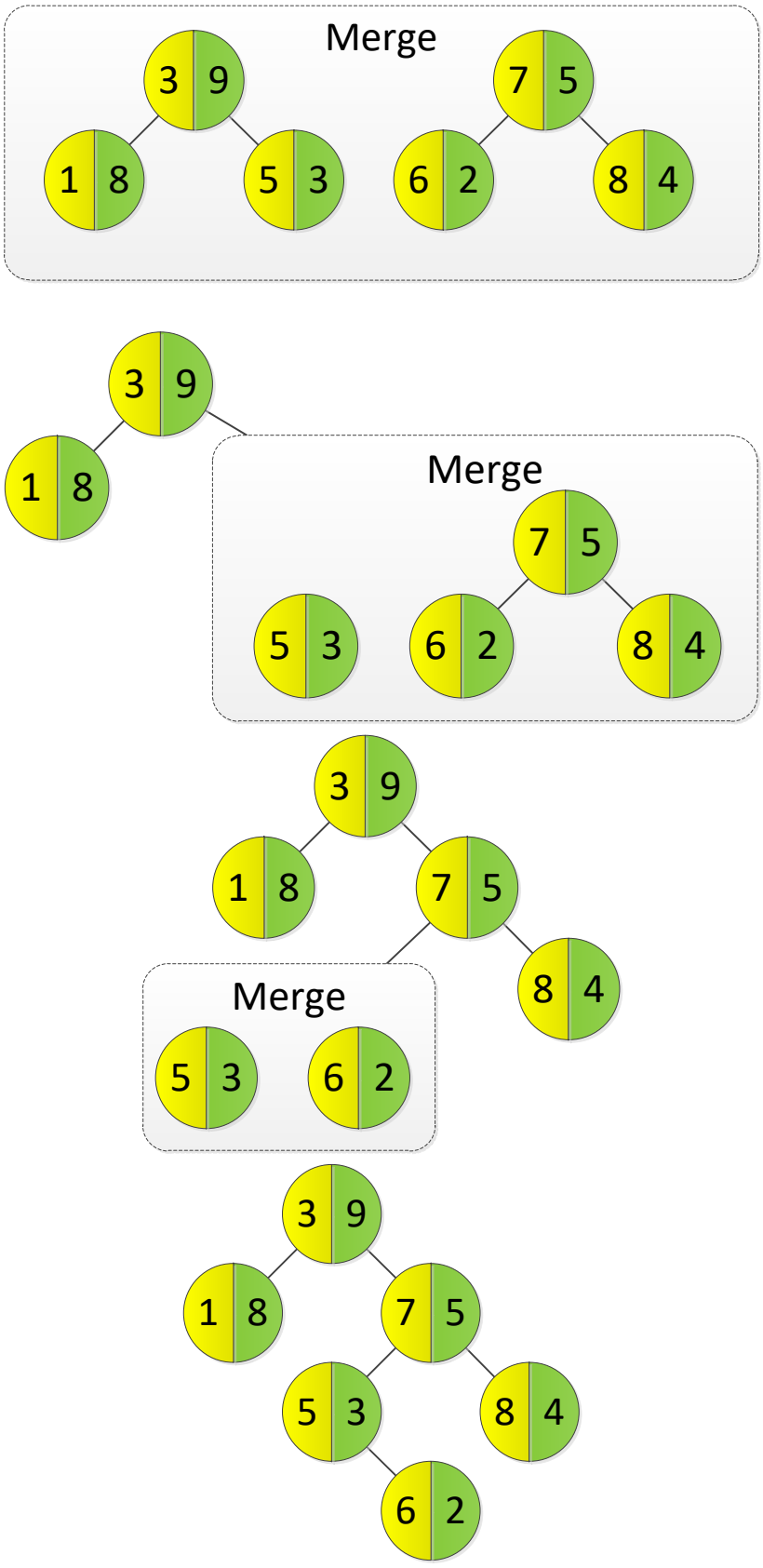
Алгоритм процедуры простой:

1) Если одно из деревьев пустое, возвращается непустое дерево. Если пусты оба, возвращается пустое дерево.

2) Если приоритет левого корня выше приоритета правого, левый корень становится корнем нового дерева, его левым потомком становится его левое поддерево, а правым – результат слияния его правого поддерева и правого дерева.

3) В противном случае по аналогии правый корень становится корнем нового дерева, его правым потомком становится его правое поддерево, а левым – результат слияния левого дерева и его левого поддерева.

На рисунках ниже приведён процесс слияния двух деревьев:



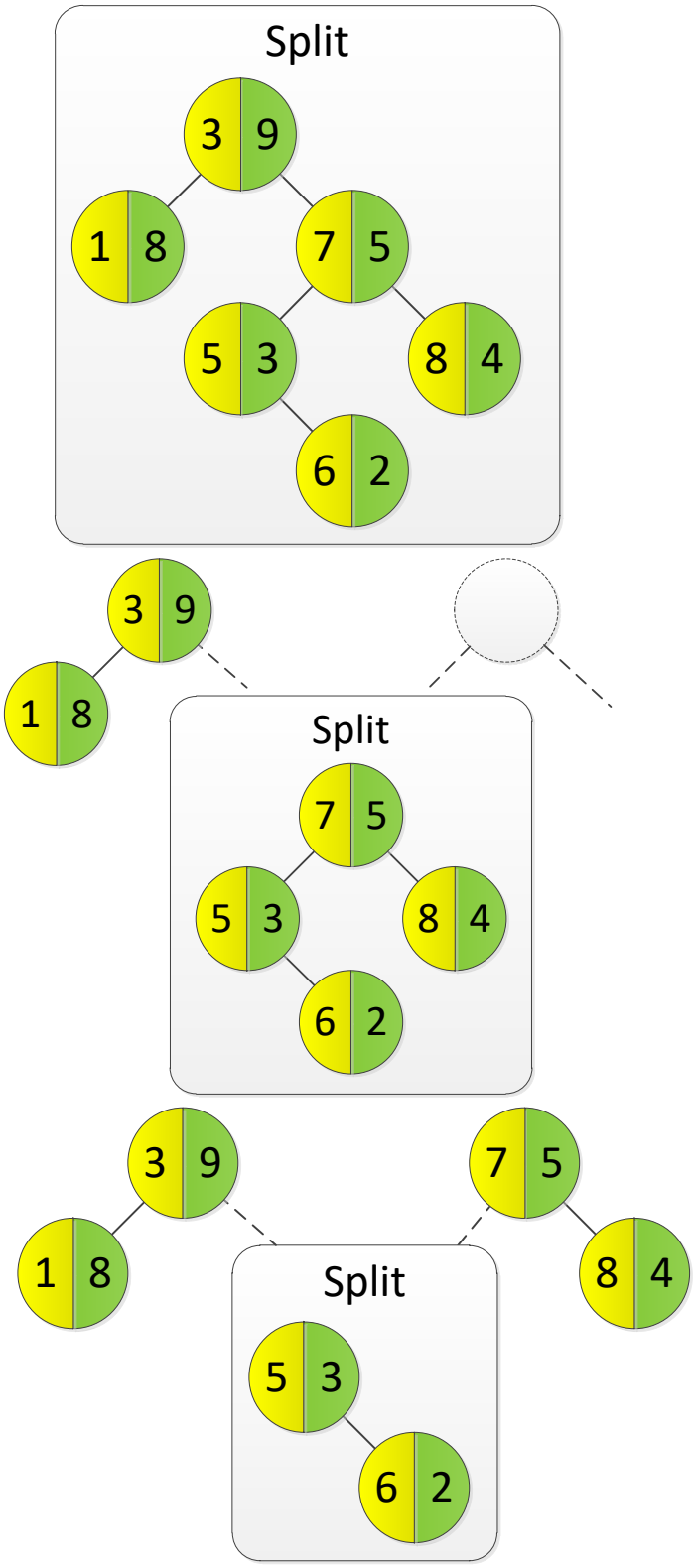
Вторая операция – операция разбиения корректного декартова дерева на два новых, в левом из которых находятся ключи, меньшие x_0, а в правом – большие и равные. Будем полагать, что в процедуру передаётся разбиваемое дерево, а возвращаются два новых, удовлетворяющих поставленному условию. Алгоритм следующий:

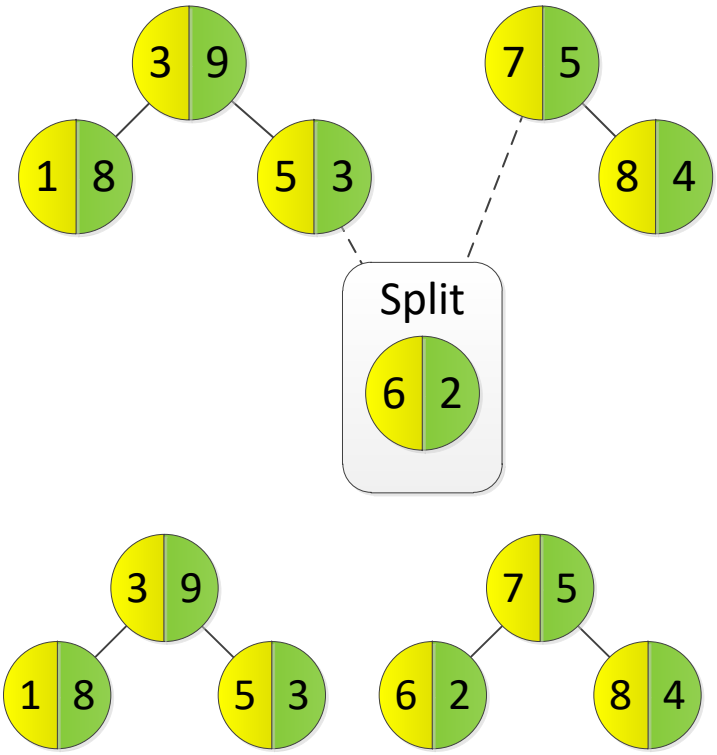
1) Если дерево пустое, вернуть два пустых дерева.
2) Если значение текущего узла меньше x_0, перейти к шагу 3, иначе к шагу 4.

3) Выполнить процедуру разбиения для правого поддерева текущего узла. Вернуть в качестве правого дерева правый результат, в качестве левого – текущий узел, у которого левым поддеревом является его текущий левый потомок, а правым – левый результат.

4) Выполнить процедуру разбиения для левого поддерева текущего узла. Вернуть в качестве левого дерева левый результат, в качестве правого – текущий узел, у которого правым поддеревом является его текущий правый потомок, а левым – правый результат.

В качестве иллюстрации, разобьём дерево по ключу 6:





Обе эти операции выполняются в среднем за логарифмическое время, так как напрямую зависят от высоты дерева. В терминах этих двух операций добавление нового узла формулируется просто:

1) Сформировать из добавляемого ключа x дерево M из одного узла с произвольным приоритетом;
2) Разделить дерево по x на левое L и правое R
3) Слить деревья L и M. Результат слияния слить с R.

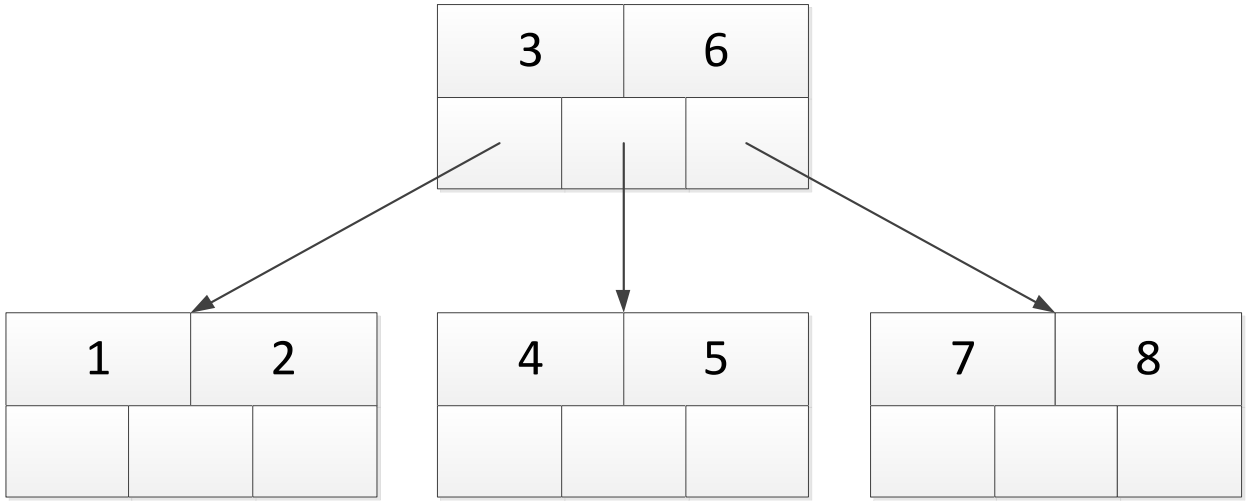
Так как операции слияния и расщепления выполняются в среднем за время порядка O(log 2 N), временная оценка всей процедуры такая же.

B-деревья

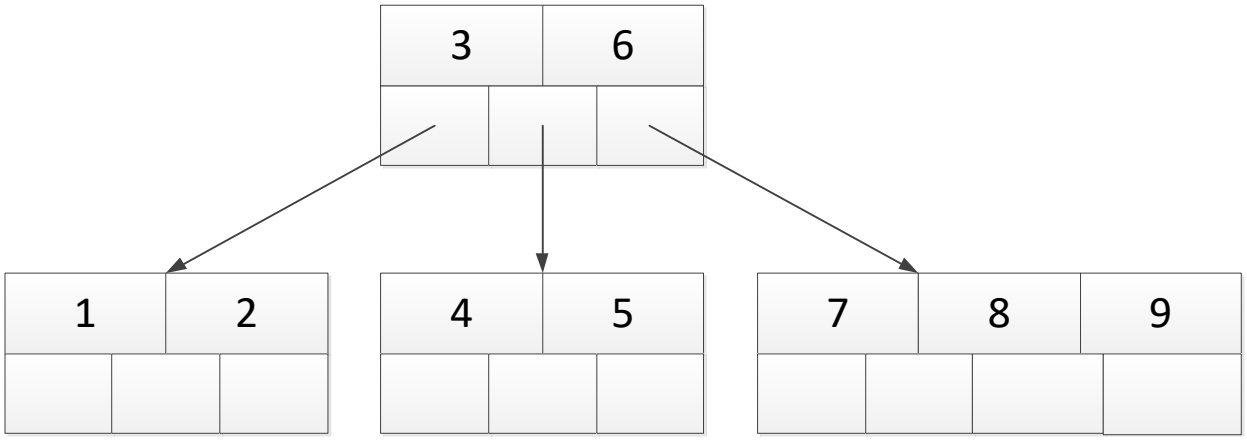
Обобщением двоичных деревьев поиска являются B-деревья. В узлах B-дерева содержится от T/2 до T чисел, где T ≥ 1 – заранее заданный параметр, однако в корне в силу специфики процедуры добавления содержится от 1 до T чисел. Для чисел N в узле выполняется неравенство $N_i < N_i+1$, i=[1,T]. Если узел является листом, то он содержит только числа, в противном случае он также хранит ссылки на 2..T+1 поддерева, значения в которых лежат в интервалах $N_i < N < N_{i+1}$ (для $N<N_1$ соотвтетвующий интервал $(-\infty;N_1)$, аналогично с N_T).

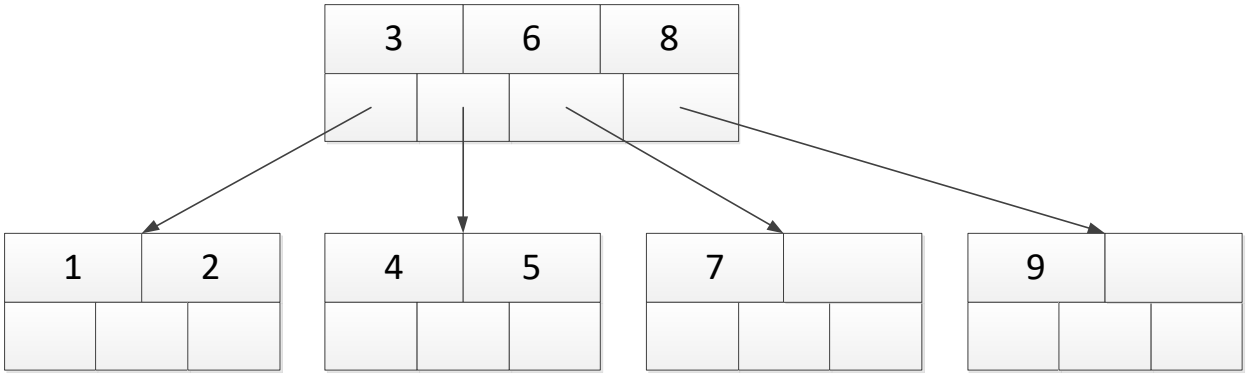
Как и в предыдущих случаях, алгоритм вставки элемента в B-дерево состоит из непосредственно добавления элемента и балансировки дерева. При добавлении элемента рекурсивно находится листовой узел, в который добавляется новый элемент. Если при этом количество элементов в узле становится равным T+1, производится процедура расщепления: создаются два новых узла, хранящие первые и последние T/2 элементов, а средний элемент добавляется в родительский узел, и процедура повторяется.

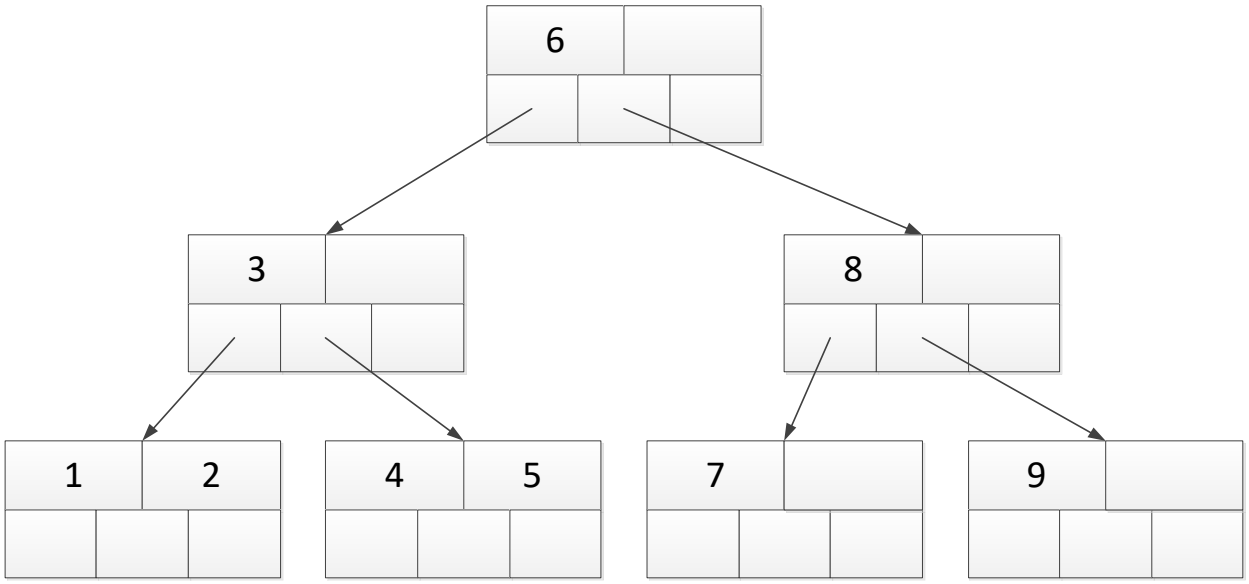
В качестве примера рассмотрим следующее дерево с T=2:



При добавлении в это дерево 10 нам придётся расщепить как правый узел, так и корень:







Таким образом, достигается сбалансированность дерева по высоте для всех листьев. Время доступа и добавления оценивается как $O(\log_{2t-1} N)$. Благодаря тому, что T может достигать порядка тысяч, время поиска мало, вследствие чего основанные на B-деревьях структуры данных широко используются в индексах баз данных.

R-деревья

B-деревья предназначены для хранения одномерных индексов. Однако их можно обобщить на случай N-мерных объектов, такая структура называется R-деревом. Мы будем рассматривать N=2, но получаемые результаты несложно экстраполировать и на большее значение.

В R-дереве могут содержаться различные геометрические объекты: как многоугольники или фигуры произвольной формы, так и отрезки и (с небольшими модификациями алгоритма) точки. Организация этой структуры данных во многом напоминает B-дерево:

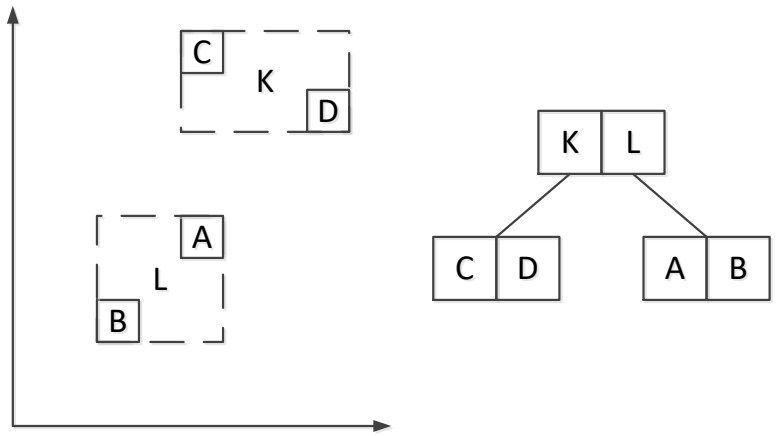
1) Для дерева определяется максимальное количество элементов в узле $M \geq 2$ и минимальное количество $m \leq M/2$.

2) Каждый лист (кроме случая, когда корень является листом) может хранить от m до M записей. Запись представляет собой минимальный прямоугольник, в который может быть вписан хранимый объект, и ссылку на него.

3) Каждый не являющийся корнем и листом узел содержит от m до M записей, каждая из которых является минимальным прямоугольником, который вмещает в себя все объекты в соответствующем поддереве.

4) Минимальное количество элементов в корне дерева – 2, если он не является листом (тогда возможно меньшее число)

Из определения видно, что R-дерево является структурой, сбалансированной по высоте. На рисунке показан пример R-дерева:



С другой стороны, недостатком R-дерева является тот факт, что минимальные прямоугольники могут пересекаться между собой, что может приводить к необходимости переходить к более чем одному нижележащему узлу в процессе поиска.

R-деревья ценны тем, что позволяют легко обрабатывать запросы вида «Какие объекты содержатся в заданной области?». Алгоритм обработки следующий:

1) Если текущий узел дерева не является листом, найти, какие хрянящиеся в нём прямоугольники пересекаются с заданной областью. Если их множество не является пустым, рекурсивно применить алгоритм для каждого из соответствующих этим прямоугольникам узлов.

2) Если узел является листом, вернуть все объекты, хранящиеся в данном листе и содержащиеся в заданной области.

Алгоритм вставки в R-дерево похож на алгоритм вставки в B-дерево. Однако поцедура разрешения переполнения узла в данном случае является нетривиальной. Рассмотрим алгоритм линейного разбиения переполненного узла на два новых.

Пусть γ - множество всех потомков данного переполненного узла.

1) Найти узлы A, B $\in$ γ, расстояние между прямоугольниками которых максимально. Обозначим множество узлов, которые наиболее близки к A (в том числе и само A) как α, а множество узлов, наиболее близких к B (в том числе и само B) как β.

2) Для всех остальных узлов C $\in$ γ пусть ΔS_α – увеличение площади минимального прямоугольника для узлов в α, если в α добавить C, ΔS_β – увеличение для β. Если ΔS_α больше ΔS_β, добавить C в β. Если ΔS_β больше ΔS_α, добавить C в α. Если ΔS_α равно ΔS_β, добавить C в множество с меньшей площадью минимального прямоугольника. Если площади прямоугольников равны, добавить элемент в множество с меньшим числом элементов. Обновить минимальный прямоугольник у множества, в которое произведено добавление.

3) Если в множестве γ осталось нерассмотренными n объектов, а в одном из множеств α или β содержится m-n объектов, поместить все оставшиеся объекты в это множество без анализа на шаге 2. Обновить минимальный прямоугольник этого множества.

4) Сформировать узлы по множествам α и β.

Линейное разбиение – не единственный возможный вариант разбиения, так как оно может приводить к неоптимальным по площади покрытия минимальным квадратам, однако оно является линейным по времени.

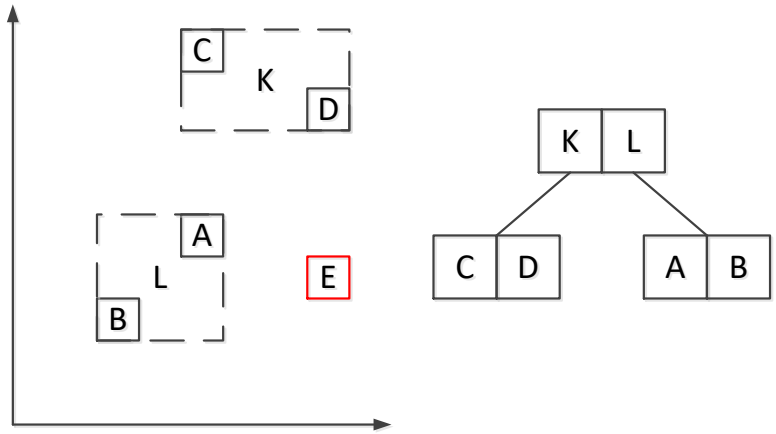
Алгоритм вставки выглядит следующим образом:

1) Среди потомков текущего узла выбрать тот узел, добавление нового элемента к которому приведёт к наименьшему увеличению площади минимального прямоугольника. Если кандидатов несколько, выбрать узел с минимальной площадью прямоугольника. Рекурсивно спускаться по дереву до листа.

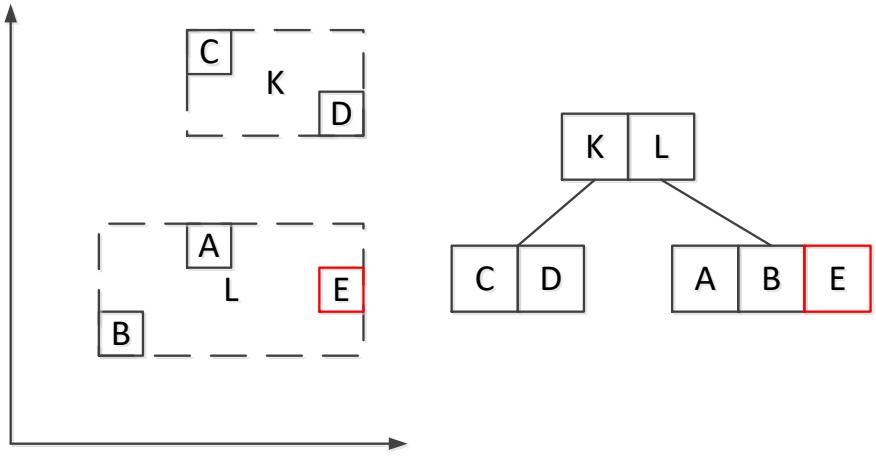
2) Если в найденном листе меньше M записей, добавить новый элемент, обновить с учётом добавленного элемента минимальные прямоугольники во всех узлах на пути от данного листа к корню.

3) Если число записей равно M, провести линейное разбиение узла на узлы L_1 и L_2. Заменить в предке текущего узла P текущий узел на L_1, обновить минимальные прямоугольники от P до корня, перейти к шагу 2 с P в качестве текущего узла и L_2 в качестве добавляемого элемента.

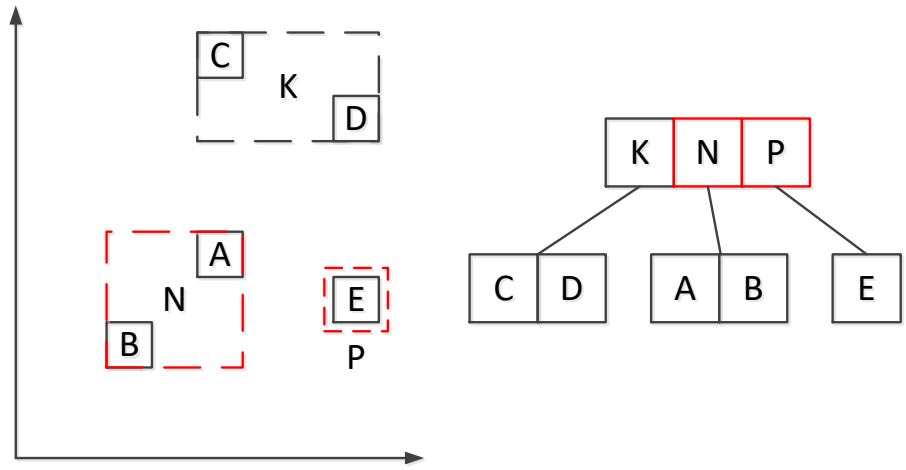
Рассмотрим алгоритм вставки на примере. Вставим в приведённое на рисунке R-дерево с M=2 новый узел E.



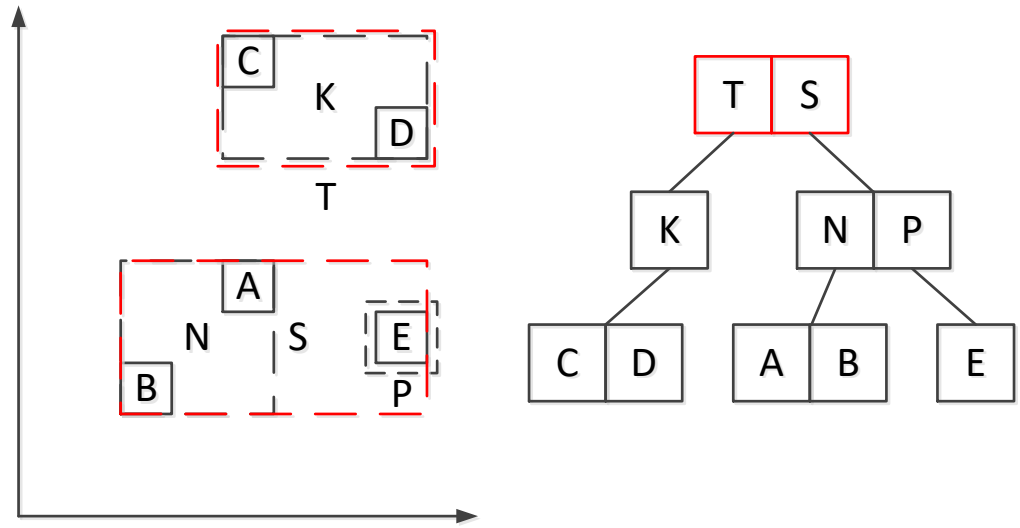
При вставке E площадь L увеличится меньше, чем площадь E, поэтому добавляем его к A и B.



Узел оказывается переполнен. Наиболее далеко друг от друга находятся элементы B и E. Узел A ближе к B, чем к E, поэтому образуются прямоугольники N, включающий A и B, и P, включающий E. Они оба добавляются к корню:



Здесь прямоугольник P показан чуть больше, чем E, в иллюстративных целях, так как его границы совпадают с E. Корень оказывается переполнен. В качестве двух первых узлов выберем K и N, так как входящие в их состав B и D максимально удалены друг от друга (хотя возможны и другие метрики, например, растояние между центрами). P добавляем к N. В итоге получается следующая структура:



Необходимо отметить, что если вставлять одни и те же данные в R-дерево в разном порядке, получатся разные деревья.