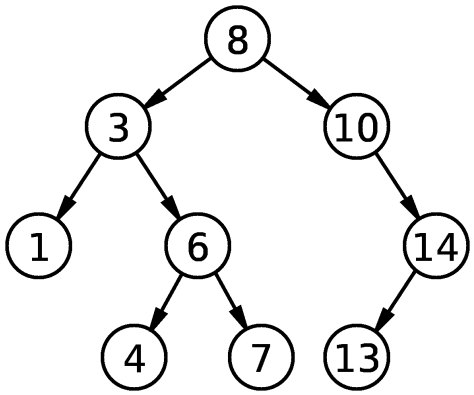
Двоичное дерево поиска

Двоичное дерево поиска		
Тип	Дерево	
Временная сложность в О-символике		
	В среднем	В худшем случае
Расход памяти	O(n)	O(n)
Поиск	O(h)	O(n)
Вставка	O(h)	O(n)
Удаление	O(h)	O(n) где h высота дерева

Двоичное дерево поиска (англ. *binary search tree*, BST) — это двоичное дерево, для которого выполняются следующие дополнительные условия (*свойства дерева поиска*):

- Оба поддерева — левое и правое, являются двоичными деревьями поиска.
- У всех узлов левого поддерева произвольного узла X значения ключей данных *меньше*, нежели значение ключа данных самого узла X.
- В то время, как у всех узлов правого поддерева того же узла X значения ключей данных *не меньше*, нежели значение ключа данных узла X.



Пример двоичного дерева поиска

Очевидно, данные в каждом узле должны обладать ключами, на которых определена операция сравнения *меньше*.

Как правило, информация, представляющая каждый узел, является записью, а не единственным полем данных. Однако, это касается реализации, а не природы двоичного дерева поиска.

Для целей реализации двоичное дерево поиска можно определить так:

- Двоичное дерево состоит из узлов (вершин) — записей вида (data, left, right), где data — некоторые данные, привязанные к узлу, left и right — ссылки на узлы, являющиеся детьми данного узла - левый и правый сыновья соответственно. Для оптимизации алгоритмов конкретные реализации предполагают также определения поля parent в каждом узле (кроме корневого) - ссылки на родительский элемент.
- Данные (data) обладают ключом (key), на котором определена операция сравнения "меньше". В конкретных реализациях это может быть пара (key, value) - (ключ и значение), или ссылка на такую пару, или простое определение операции сравнения на необходимой структуре данных или ссылке на неё.
- Для любого узла X выполняются свойства дерева поиска: key[left[X]] < key[X] ≤ key[right[X]], т. е. ключи данных родительского узла больше ключей данных левого сына и нестрого меньше ключей данных правого.

Двоичное дерево поиска не следует путать с двоичной кучей, построенной по другим правилам.

Основным преимуществом двоичного дерева поиска перед другими структурами данных является возможная высокая эффективность реализации основанных на нём алгоритмов поиска и сортировки.

Двоичное дерево поиска применяется для построения более абстрактных структур, таких как множества, мультимножества, ассоциативные массивы.

Основные операции в двоичном дереве поиска

Базовый интерфейс двоичного дерева поиска состоит из трех операций:

- FIND(K) — поиск узла, в котором хранится пара (key, value) с key = K.
- INSERT(K,V) — добавление в дерево пары (key, value) = (K, V).
- REMOVE(K) — удаление узла, в котором хранится пара (key, value) с key = K.

Этот абстрактный интерфейс является общим случаем, например, таких интерфейсов, взятых из прикладных задач:

- «Телефонная книжка» — хранилище записей (имя человека, его телефон) с операциями поиска и удаления записей по имени человека, и операцией добавления новой записи.
- Domain Name Server — хранилище пар (доменное имя, IP адрес) с операциями модификации и поиска.
- Namespace — хранилище имен переменных с их значениями, возникающее в трансляторах языков программирования.

По сути, двоичное дерево поиска — это структура данных, способная хранить таблицу пар (key, value) и поддерживающая три операции: FIND, INSERT, REMOVE.

Кроме того, интерфейс двоичного дерева включает ещё три дополнительных операции обхода узлов дерева: INFIX_TRAVERSE, PREFIX_TRAVERSE и POSTFIX_TRAVERSE. Первая из них позволяет обойти узлы дерева в порядке неубывания ключей.

Поиск элемента (FIND)

Дано: дерево T и ключ K.

Задача: проверить, есть ли узел с ключом K в дереве T, и если да, то вернуть ссылку на этот узел.

Алгоритм:

- Если дерево пусто, сообщить, что узел не найден, и остановиться.
- Иначе сравнить K со значением ключа корневого узла X.
  - Если K=X, выдать ссылку на этот узел и остановиться.
  - Если K>X, рекурсивно искать ключ K в правом поддереве T.
  - Если K<X, рекурсивно искать ключ K в левом поддереве T. === Добавление элемента (INSERT) === '''Дано''': дерево T и пара (K,V). '''Задача''': добавить пару (K, V) в дерево T. '''Алгоритм''': * Если дерево пусто, заменить его на дерево с одним корневым узлом ((K,V), null, null) и остановиться. * Иначе сравнить K с ключом корневого узла X. ** Если K>=X, рекурсивно добавить (K,V) в правое поддерево T.
  - Если K<X, рекурсивно добавить (K,V) в левое поддерево T.

Удаление узла (REMOVE)

Дано: дерево T с корнем `n` и ключом K.

Задача: удалить из дерева T узел с ключом K (если такой есть).

Алгоритм:

- Если дерево T пусто, остановиться;
- Иначе сравнить K с ключом X корневого узла `n`.
  - Если K>X, рекурсивно удалить K из правого поддерева T;
  - Если K<X, рекурсивно удалить K из левого поддерева T;
  - Если K=X, то необходимо рассмотреть три случая.
    - Если обоих детей нет, то удаляем текущий узел и обнуляем ссылку на него у родительского узла;

- Если одного из детей нет, то значения полей ребёнка `m` ставим вместо соответствующих значений корневого узла, затирая его старые значения, и освобождаем память, занимаемую узлом `m`;
- Если оба ребёнка присутствуют, то
  - найдём узел `m`, являющийся самым левым узлом правого поддерева с корневым узлом Right(n);
  - скопируем данные (кроме ссылок на дочерние элементы) из `m` в `n`;
  - рекурсивно удалим узел `m`.

Обход дерева (TRAVERSE)

Есть три операции обхода узлов дерева, отличающиеся порядком обхода узлов.

Первая операция — INFIX_TRAVERSE — позволяет обойти все узлы дерева в порядке возрастания ключей и применить к каждому узлу заданную пользователем функцию обратного вызова f. Эта функция обычно работает только с парой (K,V), хранящейся в узле. Операция INFIX_TRAVERSE реализуется рекурсивным образом: сначала она запускает себя для левого поддерева, потом запускает данную функцию для корня, потом запускает себя для правого поддерева.

- INFIX_TRAVERSE (f) — обойти всё дерево, следуя порядку (левое поддерево, вершина, правое поддерево).
- PREFIX_TRAVERSE (f) — обойти всё дерево, следуя порядку (вершина, левое поддерево, правое поддерево).
- POSTFIX_TRAVERSE (f) — обойти всё дерево, следуя порядку (левое поддерево, правое поддерево, вершина).

INFIX_TRAVERSE:

Дано: дерево T и функция f

Задача: применить f ко всем узлам дерева T в порядке возрастания ключей

Алгоритм:

- Если дерево пусто, остановиться.
- Иначе
  - Рекурсивно обойти левое поддерево T.
  - Применить функцию f к корневому узлу.
  - Рекурсивно обойти правое поддерево T.

В простейшем случае, функция f может выводить значение пары (K,V). При использовании операции INFIX_TRAVERSE будут выведены все пары в порядке возрастания ключей. Если же использовать PREFIX_TRAVERSE, то пары будут выведены в порядке, соответствующим описанию дерева, приведённого в начале статьи.

Разбиение дерева по ключу

Операция «разбиение дерева по ключу» позволяет разбить одно дерево поиска на два: с ключами $<K_0$ и $\geq K_0$.

Объединение двух деревьев в одно

Обратная операция: есть два дерева поиска, у одного ключи $<K_0$, у другого $\geq K_0$. Объединить их в одно дерево.

У нас есть два дерева: T_1 (меньшее) и T_2 (большее). Сначала нужно решить, откуда взять корень: из T_1 или T_2. Стандартного метода нет, возможные варианты:

- Взять наугад (см. декартово дерево).
- Если в каждом узле дерева поддерживается размер всей ветви (см. дерево с неявным ключом), легко можно оценить дисбаланс для того и другого варианта.

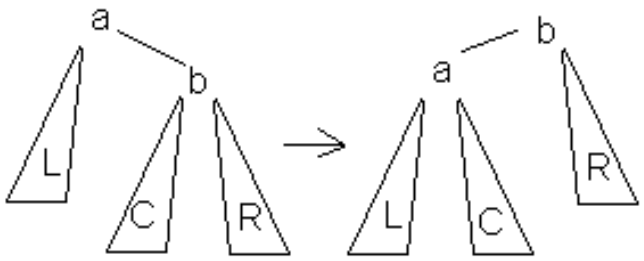
```
алг ОбъединениеДеревьев(Т1, Т2)
если Т1 пустое, вернуть Т2
если Т2 пустое, вернуть Т1
если решили сделать корнем Т1, то
  Т = ОбъединениеДеревьев(Т1.правое, Т2)
  Т1.правое = Т
  вернуть Т1
иначе
  Т = ОбъединениеДеревьев(Т1, Т2.левое)
  Т2.левое = Т
  вернуть Т2
```

Балансировка дерева

Всегда желательно, чтобы все пути в дереве от корня до листьев имели примерно одинаковую длину, т.е. чтобы глубина и левого, и правого поддеревьев была примерно одинакова в любом узле. В противном случае теряется производительность.

В вырожденном случае может оказаться, что все левое дерево пусто на каждом уровне, есть только правые деревья, и в таком случае дерево вырождается в список (идущий вправо). Поиск (а значит, и удаление и добавление) в таком дереве по скорости равен поиску в списке и намного медленнее поиска в сбалансированном дереве.

Для балансировки дерева применяется операция "поворот дерева". Поворот направо выглядит так:



- было Left(A) = L, Right(A) = B, Left(B) = C, Right(B) = R
- поворот меняет местами A и B, получая Left(A) = L, Right(A) = C, Left(B) = A, Right(B) = R
- также меняется в узле Parent(A) ссылка, ранее указывавшая на A, после поворота она указывает на B.

Поворот направо выглядит так же, достаточно заменить в вышеприведенном примере все Left на Right и обратно.

Достаточно очевидно, что поворот не нарушает упорядоченность дерева, и оказывает предсказуемое (+1 или -1) влияние на глубины всех затронутых поддеревьев.

Для принятия решения о том, какие именно повороты нужно совершать после добавления или удаления, используются такие алгоритмы, как "красно-чёрное дерево" и АВЛ.

Оба они требуют дополнительной информации в узлах - 1 бит у красно-черного или знаковое число у АВЛ.

Красно-черное дерево требует <= 2 поворотов после добавления и <= 3 после удаления, но при этом худший дисбаланс может оказаться до 2 раз (самый длинный путь в 2 раза длиннее самого короткого).

АВЛ-дерево требует <= 2 поворотов после добавления и до глубины дерева после удаления, но при этом идеально сбалансировано (дисбаланс не более, чем на 1).

Источники и основные авторы

Двоичное дерево поиска *Источник*: http://ru.wikipedia.org/w/index.php?oldid=54539186 *Редакторы*: Convallaria majalis, DenisKrivosheev, Evatutin, Greck, Grey horse, HAL9000, Ilana, Jego.ruS, Kays666, Krassotkin, Krishna, Kulhatzker, Mercury, Mikhalytch, Peni, Te Anton, Urod, Wform, X7q, Игорь Сердюков, Отец, 56 анонимных правок

Источники, лицензии и редакторы изображений

Файл:Binary search tree.svg *Источник*: http://ru.wikipedia.org/w/index.php?title=Файл:Binary_search_tree.svg *Лицензия*: Public Domain *Редакторы*: User:Booyabazooka, User:Dcoetzee

Файл:AVL LR.GIF *Источник*: http://ru.wikipedia.org/w/index.php?title=Файл:AVL_LR.GIF *Лицензия*: Public Domain *Редакторы*: Meur, Panther

Лицензия