Информационные деревья

**Информационные деревья**

Деревья в теории графов и теории алгоритмов

Деревья как информационные структуры (списки):

*  некоторая начальная вершина – корень дерева,
* каждой вершине соответствует, по крайней мере, одно значение ключа,
* каждая вершина имеет от 0 до  ссылок на другие вершины (максимальное число ссылок  – степень дерева).

**Случайное бинарное дерево** (, 1 значение  вершины)

Строится по заданному набору ключевых значений без их предварительной обработки (случайно выбранные значения).

1-е значение помещается в корень, для каждого последующего выделяется новая вершина. Поиск места для новой вершины начинается от корня и проводится по определенному правилу.

**Пример**: дерево, построенное по значениям 7, 3, 1, 11, 6, 8, 4.

11

7

3

1

6

8

4

Корень, сыновья вершины, листья (вершины без потомков).

Левое и правое поддерево вершины.

Уровень вершины – длина пути от корня до вершины.

Глубина (высота) дерева – длина максимального пути.

**Правило формирования бинарного дерева**:

 вершины  все вершины его левого поддерева содержат значения , все вершины правого .

Добавление элементов и поиск в дереве начинаются с корня и производятся в соответствии с правилом формирования дерева. Добавляемая вершина всегда будет листом дерева.

Поиск вершины является составной частью всех операций над деревом. Трудоемкость поиска вершины определяется длиной пути от корня до данной вершины.

7

7

3

7

3

1

11

7

3

1

11

7

3

1

6

11

7

3

1

6

8

11

7

3

1

6

8

4

**Наихудший вариант** бинарного дерева с  вершинами – вырождение в линейный список: длина пути составляет  и .

**Наилучший вариант** – идеальное (оптимальное, идеально сбалансированное) дерево, для которого выполняется:

* число вершин в поддеревьях одной вершины отличаются 

или

* в дереве заполнены все уровни, кроме, может быть, последнего.

При заданном числе вершин  идеальное дерево имеет минимальную высоту среди всех бинарных деревьев.

Наихудший по трудоемкости поиска случай для идеального дерева: заполнены все уровни (т.е. построено максимальное число вершин максимального уровня)

*уровень* *число вершин*





















Общее число вершин .

**Максимальная длина пути в полном идеальном дереве**:

.

**Средняя длина пути в полном идеальном дереве**:



В общем случае идеальное дерево с  уровнями содержит

 вершин, и оценки длин путей сохраняются:

, .

Построение идеального дерева (сортировка набора ключей).

Недостатки идеального дерева

**Средняя длина пути в случайном дереве**

Предположим, что  ключевых значений были отсортированы, но при построении дерева выбирались случайным образом. Если корень содержит элемент с номером , то в его левое поддерево попадают элементы , а в правое – .

=>

Средняя длина пути в дереве с  вершинами и корнем :

, где  – это средняя длина пути для левого поддерева, а  – вероятность попадания в левое поддерево (аналогично  и  для правого поддерева).

Усредняем по всем возможным значениям  и получаем среднюю длину пути в случайном бинарном дереве:



Очевидно, что ,  и .

Для  на основе матиндукции можно показать, что

.

**Средняя трудоемкость построения** случайного дерева:

.

**Удаление элементов из случайного бинарного дерева**

Пусть требуется удалить вершину со значением .

Если эта вершина – лист, то нужно просто удалить ссылку на нее в родительской вершине .

Если удаляется не лист, то в дереве необходимо найти вершину-замену и перестроить дерево, не нарушая правила его формирования.

Пусть оба поддерева вершины  не пустые,  – мн-во значений вершин левого поддерева,  – максимальный элемент левого поддерева,  – множество значений правого поддерева и  – минимальный элемент правого поддерева.

Тогда , и заменой  могут быть  или . Но если в дереве допустимы одинаковые значения (в левом поддереве), то  должно заменяться на .

Варианты удаления вершины со значением  (штриховые стрелки показывают правило замены значения , красным отмечены вершины, которые исключаются из дерева):

лист 1 поддерево 2 поддерева

Структура (класс), представляющая вершину дерева, должна содержать, по крайней мере, 3 поля:

























Вариант 1

Вариант 2

* значение ключа (**val**),
* 2 ссылки на сыновей (**left**, **right**).

Кроме того, в нее можно включить конструктор для создания отдельной вершины с заданным значением ключа.

Для дерева с целочисленными ключами:

**struct Node**

**{**

**int val;**

**Node \*left, \*right;**

**Node(int v)**

**{ val = v; left = right = NULL; }**

**}**

В класс для представления случайного бинарного дерева можно включить дополнительные рабочие поля.

**Вариант 1** (с указателем на корневую вершину):

**class Tree**

**{**

**Node \*root;**  // указатель на корень дерева

**Node \*current;** // указатель на текущую вершину

**Node \*parent;** // указатель на родительскую вершину

**public:**

**Tree() //** пустое дерево

**{ root = current = parent = NULL; }**

**Tree(int v)** // дерево с одной вершиной - корнем

**{**

**root = new Node(v);**

**current = parent = NULL;**

**}**

**}**

**Вариант 2** (с фиксированной корневой вершиной):

**class Tree**

**{**

**Node \*root;**  // указатель на условный корень дерева

**Node \*current;** // указатель на текущую вершину

**Node \*parent;** // указатель на родительскую вершину

**public:**

**Tree() //** пустое дерево с условным корнем

**{**

**root = new Node(MAX\_KEY);**

**current = parent = NULL;**

**}**

**}**

Данный вариант позволяет избежать дополнительных проверок при добавлении и удалении элементов и будет использоваться далее.

Добавление новой вершины с ключом **v**:

**void Tree::add(int v)**

**{**

**Node \*newnode = new Node(v);**

**parent = root; current = root->left;**

**while (current != NULL)**

**{**

**parent = current;**

**current = (v > current->val)?**

**current->right : current->left;**

**}**

**if (v > parent->val) parent->right = newnode;**

**else parent->left = newnode;**

**}**

Поиск вершины с ключом **v (**указатель **current)** и ее родительской вершины **(**указатель **parent)**:

**Node\* Tree::search(int v)**

**{**

**parent = root; current = root->left;**

**while (current != NULL && current->val != v)**

**{**

**parent = current;**

**current = (v > current->val)?**

**current->right : current->left;**

**}**

**return current;** // **NULL**, если вершина не найдена

**}**

Удаление вершины с ключом **v**:

**void Tree::remove(int v)**

**{**

**if (!search(v)) return;**

// удаляется лист

**if (!current->left && !current->right)**

**{**

**if (current == parent->left)**

**parent->left = NULL;**

**else parent->right = NULL;**

**}**

// только правое поддерево

**else if (!current->left)**

**{**

**if (current == parent->left)**

**parent->left = current->right;**

**else parent->right = current->right;**

**}**

// только левое поддерево

**else if (!current->right)**

**{**

**if (current == parent->left)**

**parent->left = current->left;**

**else parent->right = current->left;**

**}**

// 2 поддерева

**else**

**{**

**Node \*found = current;**

**parent = current; current = current->left;**

**while (current->right != NULL)**

**{**

**parent = current; current = current->right;**

**}**

**if (current == parent->left)** // вариант 1

**parent->left = current->left;**

**else parent->right = current->left;**

**found->val = current->val;**

**}**

**delete(current);**

**}**

**Сбалансированные деревья (АВЛ-деревья)**

Условие АВЛ-дерева: для любой вершины высоты ее левого и правого поддерева отличаются не более, чем на 1.

Наихудший случай – деревья Фибоначчи (для всех вершин высоты их поддеревьев отличаются на 1):

 – пусто,  – одна вершина (корень),

 – корневая вершина с поддеревьями  и .

Деревья Фибоначчи имеют минимальное число вершин при заданной высоте среди всех сбалансированных деревьев:

, ,  – числа Леонарда.

 растут быстрее, чем числа Фибоначчи: , но при росте  этим различием можно пренебречь.

=>

При переходе к дереву Фибоначчи следующего уровня число его вершин увеличивается не менее, чем в  раз.

Следовательно, любое сбалансированное дерево с  вершинами имеет высоту (трудоемкость поиска в наихудшем) .

Доказано, что средняя трудоемкость поиска в АВЛ-дереве с  вершинами  (немного хуже, чем в идеальных).

**Важная особенность**: для сбалансированных деревьев существуют эффективные алгоритмы восстановления сбалансированности при добавлении и удалении элементов.

**Добавление вершин к АВЛ-дереву**

Состоит из 2 шагов:

* добавление вершины, как к случайному дереву,
* проверка и восстановление сбалансированности.

Добавление новой вершины всегда увеличивает высоту какого-то поддерева (хотя бы у родительской вершины), но не всегда нарушает сбалансированность.

Проверку сбалансированности нужно проводить только для вершин, лежащих на пути от новой вершины до корня.

Структура, представляющая вершины АВЛ-дерева, должна содержать 2 дополнительных поля:

**parent** – указатель (ссылка) на родительскую вершину,

**balance** – показатель сбалансированности (разность между высотой правого и левого поддеревьев).

**struct Node**

**{**

**int val, balance;**

**Node \*left, \*right, \*parent;**

**Node(int v)**

**{**

**val = v; balance = 0;**

**left = right = parent = NULL;**

**}**

**}**

Допустимые значения **balance** -1/0/1, при нарушениях -2/2.

Примеры изменения **balance** при добавлении вершин 10,3,5:

исходное **+ 10**

8(-1) 8(0)

/ \ / \

4(0) 9(0) 4(0) 9(1)

/ \ / \ \

2(0) 6(0) 2(0) 6(0) 10(0)

**+ 3** **+ 5**

8(-2) 8(-2)

/ \ / \

4(-1) 9(0) 4(1) 9(0)

/ \ / \

2(1) 6(0) 2(0) 6(-1)

\ /

3(0) 5(0)

Очевидно, что для новой вершины **balance=0**, а для остальных вершин на пути к корню **balance** изменяется по следующим правилам:

* уменьшается на 1 при переходе из левого поддерева,
* увеличивается на 1 при переходе из правого поддерева.

Если для некоторой вершины **v** новое значение **balance=0**, то это означает, что общая высота поддерева с корнем **v** не изменилась.

=>

Для всех вершин, лежащих на пути от **v** к корню значение **balance** не изменится, поэтому и проверять их не надо.

Ниже приведены алгоритмы добавления новой вершины (листа) и последующей проверки сбалансированности.

Добавление новой вершины (листа) со значением **V**:

**void Tree::add(int v)**

**{**

**parent = root; current = root->left;**

**while (current != NULL) {**

**parent = current;**

**current = (v > current->val)?**

**current->right : current->left;**

**}**

**current = new Node(v);**

**current->parent = parent;**

**if (v > parent->val) parent->right = current;**

**else parent->left = current;**

**}**

Пересчет и проверка сбалансированности от текущей вершины (**current**)

Возвращает указатель на вершину с нарушением или NULL

**Node\* Tree::check\_balance()**

**{**

**parent = current->parent;**

**while (parent != root)**

**{**

**parent->balance +=**

**(current == parent->left)? -1 : 1;**

**if (parent->balance == 0) return NULL;**

**if (abs(parent->balance) == 2) return parent;**

**current = parent; parent = parent->parent;**

**}**

**return NULL;**

**}**

Для восстановления сбалансированности выполняются повороты АВЛ-дерева относительно вершины, в которой баланс нарушен.

Ниже показано, как выполняются повороты, если добавление вершины  увеличивает высоту левого поддерева вершины  и приводит к нарушению баланса в  (- и -повороты).

Изображенные на рисунках поддеревья 1-6 имеют одинаковую высоту . Для обрабатываемого поддерева вершины  указана его высота до и после включения , а также после поворота.  может добавляться только в одно из поддеревьев 1-4, но на рисунках приводятся все возможные случаи.

Выделены вершины, в которых изменяются ссылки **left**, **right** и\или **parent**. Очевидно, что для всех вершин от  до  пересчитывается значение **balance**.

**LL-поворот:**

























































 ()

 (=исх.)

 (исх.)

По условию формирования бинарного дерева: .

**LR-поворот:**

























































 (исх.)

 ()

 (=исх.)

При -повороте у корневых вершин деревьев 3 и 4 также изменяются ссылки на родительскую вершину (**parent**).

После перебалансировки (поворота) высота изменяемого поддерева вершины  становится такой же, как до добавления . Поэтому значения **balance** для вершин от  до корня дерева уже не изменятся, и дальнейших проверок не требуется.

Для симметричных относительно  вариантов (правое поддерево на 1 выше левого, новая вершина  попадает в правое поддерево и увеличивает его высоту) используются симметричные к указанным выше - и -повороты.

Добавление вершины  к сбалансированному дереву требует в наихудшем  шагов на поиск положения ,  на проверку сбалансированности и  на поворот дерева.

Экспериментально установлено, что на 2 добавления вершин приходится в среднем 1 поворот.**Удаление вершин из АВЛ-дерева**

Состоит из 2 шагов:

* удаление вершины, как в случайном дереве,
* проверка и восстановление сбалансированности от низа дерева (вершины-замены) до корня включительно.

Удаление вершины всегда уменьшает высоту какого-то поддерева (хотя бы у родительской вершины), но не всегда нарушает сбалансированность.

Для всех вершин на пути к корню значение **balance**:

* увеличивается на 1 при переходе из левого поддерева,
* уменьшается на 1 при переходе из правого поддерева.

Пусть для некоторой вершины  новое значение **balance** равно 1 или -1. Это означает, что старым значением для  было **balance=0**.

=>

Высоты поддеревьев  до удаления были равны, а после удаления высота одного поддерева уменьшилась на 1, а другого – не изменилась.

=>

Общая высота дерева с корнем  не изменилась.

=>

Для всех вершин, лежащих на пути от  к корню значение **balance** не изменится, поэтому и проверять их не надо.

Перебалансировка производится с помощью -, -, - и -поворотов (в среднем 1 раз на 5 удаленных вершин).

**Пример удаления вершины ** (для проверяемых вершин указаны значения **balance**):

*исходное дерево* *удалена вершина* 

*после -поворота в*  *после -поворота в* 











































































































Для всех вершин, в которых при поворотах изменяются указатели **left** и\или **right**, должно пересчитываться значение **balance**.

После -поворота в вершине  **balance=0**, поэтому проверка продолжается от  до корня.

Красно-черные деревья (другой тип сбалансированных, лучше АВЛ-деревьев в перестроениях, немного похуже в поиске).

Прошитые деревья (2 типа дополнительных ссылок).

Преимущества бинарных деревьев перед упорядоченными массивами и хеш-таблицами:

* динамические структуры,
* позволяют проводить поиск в диапазоне, выделять min и max,
* позволяют найти предыдущее\следующее значение ключа и соответствующие объекты.

Недостатком бинарных деревьев является высокая трудоемкость поиска вершин при хранении деревьев на ВУ: в среднем  чтений файла **с произвольных позиций**.

**Разделение идеального дерева на блоки**

Идеальное дерево с  вершинами делится на блоки (страницы), содержащие по  вершин. Чтение из файла производится целыми страницами – в наихудшем нужная вершина находится за  обращений к файлу.

Непригодность данного метода для неидеальных деревьев:

* большие затраты памяти на пустые страницы,
* неприспособленность к динамическим изменениям.

**B-деревья** (сильно ветвящиеся деревья)

B-дерево порядка  строится по следующим правилам:

* все значения в дереве различные,
* каждая вершина занимает 1 блок (страницу),
* каждая вершина может содержать от  до  упорядоченных по возрастанию значений (от 1 до  значений для корня),
* каждая вершина с  значениями содержит  ссылку на вершины-потомки (у листьев эти ссылки пустые),
* все листья находятся на одном уровне и не имеют потомков.

Структура вершины В-дерева ()































**Поиск вершины** со значением  в В-дереве начинается с корня В-дерева. Если текущая вершина (страница) не содержит , то следующая вершина выбирается по ссылке, соответствующей диапазону значений, в который попадает :

 – переход по ,

,  – переход по ,

 – переход по .

**Добавление значения к В-дереву**

Новое значение всегда должно включаться в соответствующий лист.

Если данный лист содержит  значений, то новое значение просто добавляется к листу с соблюдением условия упорядоченности.

Если же лист уже содержит  значений, то производится его сбалансированное расщепление:

*  значений ( старых + 1 новое) сортируются по возрастанию,
*  начальных значений формируют один новый лист (левый),  конечных – другой (правый), а медианное значение выталкивается наверх, в родительскую вершину.

*c*

*a*

*b*

*d*

*f*

*a*

*c*

*d*

*f*

+ *b* => { , , , ,  } =>

Если родительская вершина изменилась (в нее было добавлено новое значение), то она должна обрабатываться так же, как начальный лист. Процесс расщепления вершин может продолжаться вплоть до корня В-дерева.

Если происходит расщепление корневой вершины, то образуется новый корень (содержащий 1 значение), то есть В‑дерево растет корнем вверх.

**Пример построения В-дерева порядка 2** (красным указаны добавляемые значения, в фигурных скобках – множества значений, используемых при расщеплении вершин):

1

3

6

12

3

+ 3 + 1, 6, 12

6

1

3

12

17

+ 17 => { 1, 3, 6, 12, 17 } =>

1

2

3

5

6

8

10

12

17

+ 2, 5, 8,10

1

2

3

5

6

8

10

15

17

12

+ 15 => { 8, 10, 12, 15, 17 } =>

1

2

4

5

6

8

10

15

17

12

3

+ 4 => { 1, 2, 3, 4, 5 } =>

+ 7, 11, 14, 16

1

2

4

5

6

8

10

14

17

12

3

16

15

11

7

1

2

4

5

6

8

10

14

17

12

3

16

15

11

7

9

+ 9 => { 7, 8, 9, 10, 11 } =>

+ 13 => { 13, 14, 15, 16, 17 }

=> { 3, 6, 9, 12, 15 } =>

1

2

4

5

6

8

10

13

17

12

3

16

15

11

7

9

14

**Удаление значения  из В-дерева**

Если  находится в листе, то  удаляется непосредственно оттуда.

Если  находится в вершине более высокого уровня , то нужно найти в дереве замену – максимальное значение, меньшее . Такое значение всегда располагается в некотором листе  (пусть это значение , как на рисунке).  должно заменить  в , а затем  нужно удалить из .

*…*

*v*

*…*

*f*

*g*

*h*

*k*

*l*

*m*

*r*

*s*

*t*

Вершина W

*f < g < h < v*

*h < k < l < m < v*

лист *L*  *m < r < s < t < v*

При удаления значения из некоторого листа  возможны 2 случая:

1.  содержит  значений. Необходимо просто удалить нужное значение, сохраняя упорядоченность оставшихся.

2.  содержит ровно  значений. В этом случае нужно объединить остающиеся значения из  со значениями правого или левого соседа  (листа ), и тем значением  из родительской (для  и ) вершины, для которого выполняется условие  или  – в зависимости от взаимного расположения  и .

*f*

*d*

***p***

*z*

*m*

***n***

*u*

*v*

*w*

*x*

*родитель*

*L R*

{ *m, p, u, v, w, x* }

Пусть множество  содержит  значений.

=>

1. Если , то медианный элемент перемещается в родительскую вершину, а из значений меньших и больших медианного формируются новые вершины  и .

{ *m, p, u, v, w, x* }

*f*

*d*

***u***

*z*

*m*

***p***

*v*

*w*

*x*

*родитель*

*L R*

2. Если , то из данных значений нужно сформировать одну новую вершину , удалить старую вершину , и провести с родительской вершиной такие же проверки, как с  (родительская вершина изменилась, т.к. из нее было удалено значение ).

{ *m, p, u, v* }

*f*

*d*

*u*

*z*

*m*

***p***

*v*

*…*

*…*

*родитель*

*L*

Процесс удаления вершин из В-дерева демонстрируют рисунки из пункта «Построение В-дерева» – в обратном порядке.

Наихудшим для В-дерева порядка  с  значениями является случай, когда все вершины содержат по  значений и  ссылке (а также 1 значение и 2 ссылки в корне). При этом память используется на 50%, а высота дерева (трудоемкость поиска в наихудшем) равна .

Например, при  (относительно небольшие страницы) высота В-дерева, содержащего  значений, составляет всего 4 в наихудшем.

При большом числе операций поиска выгодно хранить верх В‑дерева в оперативной памяти.

Простейший вариант – **В-деревья порядка 1** или **2-3-деревья** – используются при работе с динамическими объектами в ОП и являются альтернативой сбалансированным деревьям (2-3-деревья немного лучше в перестроениях, но похуже в поиске).

Вершины 2-3-дерева можно хранить как вершины бинарного дерева, в которых правая ссылка указывает либо на вершину следующего уровня, либо на соседнюю вершину по горизонтали (должен храниться и тип ссылки):

*a*

*b*

*a*

*b*

     

Другие типы деревьев:

- деревья оптимального поиска,

- декартовы, тетра- и R-деревья.