АВЛ-дерево

АВЛ-дерево — сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на 1.

АВЛ — аббревиатура, образованная первыми буквами создателей (советских учёных) Адельсон-Вельского Георгия Максимовича и Ландиса Евгения Михайловича.

Общие свойства

В АВЛ-дереве высоты h имеют не меньше F_h узлов, где F_h — число Фибоначчи. Поскольку

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}} = \frac{\phi^n - (-\phi)^{-n}}{\phi - (-\phi)^{-1}},$$

где $\phi = \frac{1+\sqrt{5}}{2}$ — золотое сечение,

то имеем оценку на высоту АВЛ-дерева $h = O(log(n))$, где n — число узлов. Следует помнить, что $O(log(n))$ — мажоранта [1], и её можно использовать только для оценки (Например, если в дереве только два узла, значит в дереве два уровня, хотя $log(2) = 1$). Для точной оценки глубины дерева следует использовать пользовательскую подпрограмму.

```
  function TreeDepth(Tree : TAVLTree) : byte;
    begin
       if Tree <> nil then
          result := 1 +
Max(TreeDepth(Tree^.left),TreeDepth(Tree^.right))
      else
          result := 0;
    end;
```

Тип дерева можно описать так

```
    TKey = LongInt;
    TInfo = LongInt;
    TBalance = -2..2; // диапазон в районе от -1 до 1 , но включим для
простоты нарушения -2 и 2
    PAVLNode = ^ TAVLNode;
      TAVLNode = record
      case integer of
       0:(left, right : PAVLNode;
       key : TKey;
       info : TInfo;
       { Поле определяющее сбалансированность вершины }
       balance : TBalance;);
       1:(childs:array[boolean] of PAVLNode); // представление веток
дерева в виде массива для упрощения переходов
     end;
    TAVLTree = PAVLNode;
```

AVL-условия можно проверить так

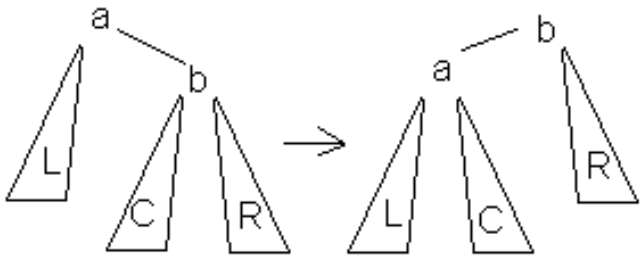
```
    function TestAVLTree(V:PAVLNode):integer; //возвращает высоту
дерева
    var a,b:integer;
    begin
      Result:=0;
      if V=nil then exit;
      a:=TestAVLTree(V.Left);
      b:=TestAVLTree(V.Right);


      if ((a-b)<>V.Balance)or(abs(a-b)>=2) then begin
        raise Exception.CreateFmt('%d - %d  balancefactor
%d',[a,b,V.Balance]);
      end;
      Result:=1+max(a,b);
    end;
```

Балансировка

Относительно АВЛ-дерева балансировкой вершины называется операция, которая в случае разницы высот левого и правого поддеревьев = 2, изменяет связи предок-потомок в поддереве данной вершины так, что разница становится <= 1, иначе ничего не меняет. Указанный результат получается вращениями поддерева данной вершины.

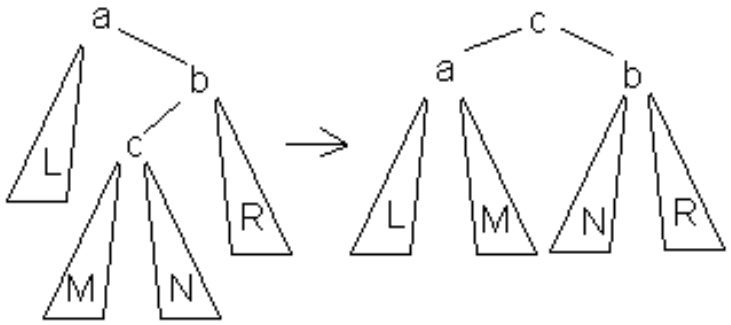
```
Используются 4 типа вращений:
```

```
1.Малое левое вращение
```



Данное вращение используется тогда, когда (высота b-поддерева - высота L) = 2 и высота C <= высота R.

```
2.Большое левое вращение
```

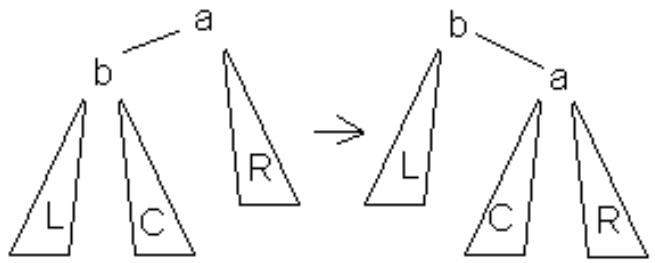


Данное вращение используется тогда, когда (высота b-поддерева - высота L) = 2 и высота c-поддерева > высота R.

```
//Функция для устранения правого нарушения с помощью вышеописанных
поворотов,
//возвращает True если высота дерева уменьшилась, False - если осталась
 той же

function AVL_FixWithRotateLeft(var N:PAVLNode):boolean;
var R,RL,RLR,RLL:PAVLNode;
begin
    R:=N.Right;
    RL:=R.Left;
    Result:=true;
    case R.Balance of
     -1 :begin
          N.Balance:= 0;     // h(RL)=H-3 h(L)=H-3 => h(N) =H-2
          R.Balance:= 0;     // h(RR)=H-2 => h(R)= H-1
          N.Right:=RL;

          R.Left:=N;
          N:=R;
        end;
     0 :begin
          N.Balance:= -1;     // h(RL)=H-2 h(L)=H-3 => h(N) =H-1
          R.Balance:= 1;     // h(RR)=H-2 => h(L)= H

          N.Right:=RL;
          R.Left:=N;
          N:=R;
          Result:=false;
        end;
     1:begin
          RLR:=RL.Right;
          RLL:=RL.Left;
```

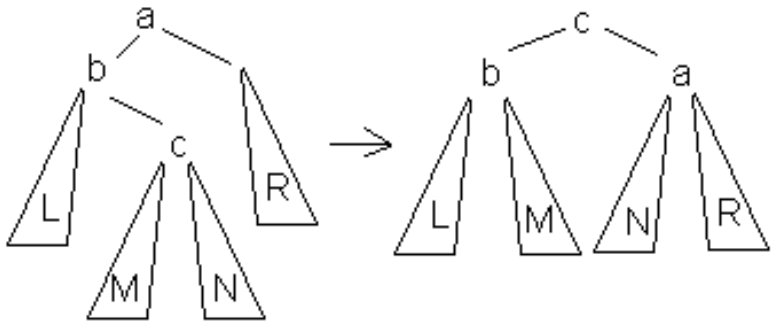
```
          R.Left:=RLR;
          R.Balance:=min(-RL.Balance,0); //1 =>-1, 0 =>0, -1 =>0

          N.Right:=RLL;
          N.Balance:=max(-RL.Balance,0); //1 => 0, 0 =>0, -1 => 1

          RL.Right:=R;
          RL.Left:=N;
          RL.Balance:=0;

          N:=RL;
        end;
    end;
end;
```

3.Малое правое вращение



Данное вращение используется тогда, когда (высота b-поддерева — высота R) = 2 и высота C <= высота L.

4.Большое правое вращение



Данное вращение используется тогда, когда (высота b-поддерева - высота R) = 2 и высота c-поддерева > высота L.

```
//Функция для устранения левого нарушения с помощью вышеописанных
поворотов,
//возвращает True если высота дерева уменьшилась, False - если осталась
 той же
function AVL_FixWithRotateRight(var N:PAVLNode):boolean;
var L,LR,LRL,LRR:PAVLNode;
begin
    L:=N.Left;
    LR:=L.Right;
    Result:=true;
    case L.Balance of
     1:begin
          N.Balance:= 0;     // h(LR)=H-3 h(R)=H-3 => h(N) =H-2
          L.Balance:= 0;     // h(LL)=H-2 => h(L)= H-1
          N.Left:=LR;
          L.Right:=N;
          N:=L;
       end;
     0 :begin
          N.Balance:=1;     // h(LR)=H-2 h(R)=H-3 => h(N) =H-1
          L.Balance:= -1;     // h(LL)=H-2 => h(L)= H
          N.Left:=LR;
          L.Right:=N;
          N:=L;
          Result:=false;
       end;
     -1 :begin
            LRL:=LR.Left;
            LRR:=LR.Right;

            L.Right:=LRL;
            L.Balance:=max(-LR.Balance,0); //1 =>0, 0 =>0, -1 =>1

            N.Left:=LRR;
            N.Balance:=min(-LR.Balance,0); //1 => -1, 0 =>0, -1 => 0

            LR.Left:=L;
            LR.Right:=N;
            LR.Balance:=0;
            N:=LR;
       end;
    end;
end;
```

В каждом случае достаточно просто доказать то, что операция приводит к нужному результату и что полная высота уменьшается не более чем на 1 и не может увеличиться.

Также можно заметить, что большое вращение это комбинация правого и левого малого вращения.

Из-за условия балансированности высота дерева O(log(N)), где N- количество вершин, поэтому добавление элемента требует O(log(N)) операций.

Алгоритм добавления вершины

Показатель сбалансированности в дальнейшем будем интерпретировать как разность между высотой левого и правого поддерева, а алгоритм будет основаться на типе TAVLTree, описанном выше. Непосредственно при вставке (листу) присваивается нулевой баланс. Процесс включения вершины состоит из трех частей:

1. Прохода по пути поиска, пока не убедимся, что ключа в дереве нет.
2. Включения новой вершины в дерево и определения результирующих показателей балансировки.
3. "Отступления" назад по пути поиска и проверки в каждой вершине показателя сбалансированности. Если необходимо - балансировка.

Будем возвращать в качестве результата функции, уменьшилась высота дерева или нет. Предположим, что процесс из левой ветви возвращается к родителю (рекурсия идет назад), тогда возможны три случая: { h_l - высота левого поддерева, h_r - высота правого поддерева } Включение вершины в левое поддерево приведет к

1. $h_l < h_r$: выравняется $h_l = h_r$. Ничего делать не нужно.
2. $h_l = h_r$: теперь левое поддерево будет больше на единицу, но балансировка пока не требуется.
3. $h_l > h_r$: теперь $h_l - h_r = 2$, - требуется балансировка.

В третьей ситуации требуется определить балансировку левого поддерева. Если левое поддерево этой вершины (Tree^.left^.left) выше правого (Tree^.left^.right), то требуется большое правое вращение, иначе хватит малого правого. Аналогичные (симметричные) рассуждения можно привести и для включение в правое поддерево.

вспомогательная функция сравнивающая два ключа

```
function KeyCompare(const V1,V2:TKey):integer;
begin
  if V2>V1 then begin
    Result:=-1;
  end else
  if V2=V1 then begin
    Result:=0;
  end else
    Result:=1;
end;
```

Рекурсивная процедура вставки:

```
 function AVL_InsertNode(Var Tree : TAVLTree; const aKey : TKey; const
ainfo : TInfo): Boolean;
 Var
   c:integer;
 begin
   if Tree = nil then begin
      New(Tree);
      Result := true;
      with Tree^ do
        begin
          key := akey;
          info := ainfo;
```

```
          left := nil;
          right := nil;
          balance := 0;
        end;
    end else begin
      c:= KeyCompare(aKey,Tree^.key);
      if c=0 then begin
       Tree^.info:=ainfo;
       Result := false;
      end else begin
       Result:=AVL_InsertNode(Tree^.childs[c>0],akey,ainfo);
       if Result then begin
         if c>0 then Tree^.balance:= Tree^.balance-1 else Tree^.balance:=
 Tree^.balance+1;
        case Tree^.balance of
          2: Result:=not AVL_FixWithRotateRight(Tree);
          -2: Result:=not AVL_FixWithRotateLeft(Tree);
          0: Result:=false;
        end
       end;
      end;
    end;
 end;
```

Алгоритм удаления вершины

Для простоты опишем рекурсивный алгоритм удаления. Если вершина - лист, то удалим её и вызовем балансировку всех её предков в порядке от родителя к корню. Иначе найдём самую близкую по значению вершину в поддереве наибольшей высоты (правом или левом) и переместим её на место удаляемой вершины, при этом вызвав процедуру её удаления.

Упрощённый вариант удаления можно описать таким образом

```
// Функция очень далека от оптимальной,
// сравнение происходит даже после нахождения удаляемого ключа
// передаются сразу все параметры, некоторые из которые можно не
использовать,
// разбив на 3 процедуры с более упрощённой функциональностью :
// 1.движение только влево
// function AVL_DropNodeLeft(Var Tree : TAVLTree; DropedNode:TAVLTree):
 Boolean;
// 2.движение только вправо
// function AVL_DropNodeRight(Var Tree : TAVLTree;
DropedNode:TAVLTree): Boolean;
// 3.поиск
// function AVL_DropNode(Var Tree : TAVLTree; const aKey :
Boolean;
function AVL_DropNode(Var Tree : TAVLTree; const aKey :
TKey;DropedNode:TAVLTree=nil): Boolean;
```

```
var c:integer;
begin
  if Tree = nil then begin
    Result := false;
    exit;
  end;
  c:= KeyCompare(aKey,Tree^.key);
  if c=0 then begin
    DropedNode:=Tree;
    c:=-DropedNode.balance;//пойдём в более высокую или левую ветвь
дерева если их высоты равны
  end;
  if (Tree^.childs[c>0]=nil)and(DropedNode<>nil) then begin
    DropedNode^.Key:=Tree^.Key;
    DropedNode^.info:=Tree^.info;
    DropedNode:=Tree;
    //поставим вместо текущего лист с противоположного направления
    Tree:=Tree^.childs[c<=0];
    Dispose(DropedNode);
    Result:=true;
    exit;
  end;
  Result:=AVL_DropNode(Tree^.childs[c>0],aKey,DropedNode);
  if Result then begin
    if c>0 then Tree^.balance:= Tree^.balance+1 else Tree^.balance:=
Tree^.balance-1;
    case Tree^.balance of
      -2: Result:=AVL_FixWithRotateLeft(Tree);
      -1,1: Result:=false;
      2: Result:=AVL_FixWithRotateRight(Tree);
    end;
  end;
end;
```

Докажем, что данный алгоритм сохраняет балансировку. Для этого докажем по индукции по высоте дерева, что после удаления некоторой вершины из дерева и последующей балансировки высота дерева уменьшается не более, чем на 1. База индукции: Для листа очевидно верно. Шаг индукции: Либо условие балансированности в корне (после удаления корень может изменится) не нарушилось, тогда высота данного дерева не изменилась, либо уменьшилось строго меньшее из поддеревьев => высота до балансировки не изменилась => после уменьшится не более чем на 1.

Очевидно, в результате указанных действий процедура удаления вызывается не более 3 раз, так как у вершины, удаляемой по 2-му вызову, нет одного из поддеревьев. Но поиск ближайшего каждый раз требует O(N) операций, отсюда видна очевидная оптимизация: поиск ближайшей вершины производится по краю поддерева. Отсюда количество действий O(log(N)).

Нерекурсивная вставка в АВЛ-дерево сверху-вниз

Нерекурсивный алгоритм сложнее чем рекурсивная реализация.

1. Находится место вставки и вершина высота которой не изменится при вставке (это вершина у которой высота левого поддерева не равна высоте правого, будем называть её PrimeNode)
2. Производится спуск от PrimeNode до места вставки с изменением балансов
3. Производится ребалансировка PrimeNode при наличии переполнения

```
type
  PAVLTree=^TAVLTree; //дополнительный тип для указания на место где
хранится указатель на листок

// функция возвращает True если было добавление нового литка, false -
произошла замена значения ключа
function AVL_InsertNode2(var Root:TAVLTree;const aKey:TKey;const
Value:TInfo):boolean;
var PrimeNode,p,q:PAVLTree;
  c:integer;
begin
  q:=@Root;
  PrimeNode:=q;
  //1-я часть алгоритма
  if q^<>nil then begin
    repeat
      c:=KeyCompare(aKey,q^.Key);
      if c=0 then begin
        q^.info:=Value;
        Result:=false;
        exit;
      end;
      if (q^.Balance<>0) then begin
        PrimeNode:=q;
      end;
      q:=@q^.Childs[c>0];
    until q^=nil;
  end;
  New(q^);
  with q^^ do begin
      key := akey;
      info := Value;
      left := nil;
      right := nil;
      balance := 0;
  end;
  if PrimeNode<>q then begin
    //2-я часть алгоритма
    p:=PrimeNode;
    repeat
```

```
      c:=KeyCompare(aKey,p^.Key);
      if c>0 then begin
        p^.Balance:=p^.Balance-1;
        p:=@p^.Right;
      end else begin
        p^.Balance:=p^.Balance+1;
        p:=@p^.Left;
      end;
    until p=q;
    //3-я часть алгоритма
    case PrimeNode^.Balance of
     2:  AVL_FixWithRotateRight(PrimeNode^);
     -2: AVL_FixWithRotateLeft(PrimeNode^);
    end;
  end;
  Result:=true;
end;
```

Нерекурсивное удаление из АВЛ-дерева сверху-вниз

Для реализации удаления будем исходить из того же принципа что и при вставке, будем искать вершину, удаление из которой не приведёт к изменению её высоты, существуют всего два таких варианта

1. самый простой, когда высота левого поддерева равна высоте правого поддерева (исключая случай когда у листка нет поддеревьев)
2. когда высота дерева по направлению движения меньше противоположной("брат" направления) и баланс "брата" равен 0 (разбор этого варианта довольно сложен - так что пока без доказательства)

```
function AVL_DropNode2(var Root:PAVLNode;const Key:TKey):boolean;
var PrimeNode,p,q,b:PAVLTree;
  c:integer;
  last:boolean;
  DropedNode:PAVLNode;
begin
  p:=nil;
  q:=@Root;
  PrimeNode:=q;
  last:=false;
  DropedNode:=nil;
  while q^<>nil do begin
    if (p<>nil) then begin
      if (q^^.Balance=0)and(q^^.Left<>nil) then begin
          PrimeNode:=q;
      end else
      if (last and(p^^.Balance=1))or((not last) and(p^^.Balance=-1))
then begin
        b:=@p^^.Childs[not last];
        if b^.Balance=0 then begin
          PrimeNode:=p;
```

```
        end;
      end;
    end;
    c:=KeyCompare(Key,q^^.Key);
    last:=c>0;
    p:=q;
    q:=@q^^.Childs[last];
    if c=0 then begin
      DropedNode:=p^;
    end;
  end;
  if DropedNode=nil then begin
    Result:=false;
    exit;
  end;
  Result:=true;
  while PrimeNode<>p do begin
    c:=KeyCompare(Key,PrimeNode^.Key);
    if c>0 then begin
      PrimeNode^.Balance:=PrimeNode^.Balance+1;
      if PrimeNode^.Balance=2 then begin
        AVL_FixWithRotateRight(PrimeNode^);
        PrimeNode:=@PrimeNode^.Right; // пропускаем из обработки, там
наша текушая вершина теперь
      end;
      PrimeNode:=@PrimeNode^.Right;
    end else begin
      PrimeNode^.Balance:=PrimeNode^.Balance-1;
      if PrimeNode^.Balance=-2 then begin
        AVL_FixWithRotateLeft(PrimeNode^);
        PrimeNode:=@PrimeNode^.Left; // пропускаем из обработки, там
наша текушая вершина теперь
      end;
      PrimeNode:=@PrimeNode^.Left;
    end;
  end;
  DropedNode^.Key:=p^^.Key;
  DropedNode^.info:=p^^.info;
  DropedNode:=p^;
  //поставим вместо текущего лист с противоположного направления
  p^:=p^^.childs[(p^^.Left=nil)];
  Dispose(DropedNode);
end;
```

Сам алгоритм без всех оптимизаций для упрощения его понимания. В отличие от рекурсивного алгоритма при нахождении удаляемой вершины она будет заменена значением из левой подветви, данный алгоритм можно оптимизировать так же как и для рекурсивной версии за счёт того что после нахождения удаляемой вершины направление движения нам известно

1. ищем удаляемый элемент и попутно находим нашу замечательную вершину
2. производим изменение балансов, в случае необходимости делаем ребалансировку
3. удаляем наш элемент (в действительности не удаляем, а заменяем его ключ и значение, учёт перестановок вершин будет немного сложнее)

Расстановка балансов при удалении

Как уже говорилось, если удаляемая вершина — лист, то она удаляется, и обратный обход дерева происходит от родителя удалённого листа. Если не лист — ей находится «замена», и обратный обход дерева происходит от родителя «замены». Непосредственно после удаления элемента — «замена» получает баланс удаляемого узла.

При обратном обходе: если при переходе к родителю пришли слева — баланс увеличивается на 1, если же пришли справа — уменьшается на 1.

Это делается до тех пор, пока при изменении баланса он не станет равным −1 или 1 (обратите внимание на различие с вставкой элемента!): в данном случае такое изменение баланса будет гласить о неизменной дельта-высоте поддеревьев. Повороты происходят по тем же правилам, что и при вставке.

Расстановка балансов при одинарном повороте

Обозначим:

«Current» — узел, баланс которого равен −2 или 2: то есть тот, который нужно повернуть (на схеме - элемент **a**)

«Pivot» — ось вращения. +2: левый сын Current'a, −2: правый сын Current'a (на схеме - элемент **b**)

Если поворот осуществляется при вставке элемента, то баланс Pivot'a равен либо 1, либо −1. В таком случае после поворота балансы обоих устанавливаются равными 0.

При удалении всё иначе: баланс Pivot'a может стать равным 0 (в этом легко убедиться).

Приведём сводную таблицу зависимости финальных балансов от направления поворота и исходного баланса узла Pivot:

Направление поворота	**Old Pivot.Balance**	**New Current.Balance**	**New Pivot.Balance**
Левый или Правый	-1 или +1	0	0
Правый	0	-1	+1
Левый	0	+1	-1

Расстановка балансов при двойном повороте

Pivot и Current — те же самые, но добавляется третий участник поворота. Обозначим его за «Bottom»: это (при двойном правом повороте) левый сын Pivot'a, а при двойном левом — правый сын Pivot'a.

При данном повороте — Bottom в результате всегда приобретает баланс 0, но от его исходного баланса зависит расстановка балансов для Pivot и Current.

Приведём сводную таблицу зависимости финальных балансов от направления поворота и исходного баланса узла Bottom:

Направление	Old Bottom.Balance	New Current.Balance	New Pivot.Balance
Левый или Правый	0	0	0
Правый	+1	0	-1
Правый	-1	+1	0
Левый	+1	-1	0
Левый	-1	0	+1

Оценка эффективности

Г.М.Адельсон-Вельский и Е.М.Ландис доказали теорему, согласно которой высота АВЛ-дерева с N внутренними вершинами заключена между $\log_2(N+1)$ и $1.4404*\log_2(N+2)-0.328$, то есть высота АВЛ-дерева никогда не превысит высоту идеально сбалансированного дерева более, чем на 45%. Для больших N имеет место оценка $1.04*\log_2(N)$. Таким образом, выполнение основных операций 1 – 3 требует порядка $\log_2(N)$ сравнений. Экспериментально выяснено, что одна балансировка приходится на каждые два включения и на каждые пять исключений.

Литература

- Вирт Н. *Алгоритмы и структуры данных* М.:Мир, 1989. Глава 4.5 (С. 272-286)
- Г. М. Адельсон-Вельский, Е. М. Ландис. Один алгоритм организации информации // Доклады АН СССР. 1962. Т. 146, № 2. С. 263–266.
- GNU libavl 2012 Ben Pfaff.

Примечания

[1] http://ru.wiktionary.org/wiki/мажоранта

Источники и основные авторы

АВЛ-дерево *Источник*: http://ru.wikipedia.org/w/index.php?oldid=54837248 *Редакторы*: Altes, Bazanovp, Bunker by, Dipsy, Evatutin, Greck, Guzanof, HAL9000, Ilyan, Megamarsik, Meur, Mnogo, OckhamTheFox, Overrider, Paul Pogonyshev, Pavel Sutyrin, Peni, PeterMinin, Russische Patriot, Saaska, Sealle, Swappp, Szviag, Vprisivko, X7q, Александр Дмитриев, Алекснадр Мехоношин, 105 анонимных правок

Источники, лицензии и редакторы изображений

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

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

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

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

Лицензия