Работа со структурами данных в языках Си и Python: Часть 8. Сбалансированные двоичные деревья (AVL-деревья)

Сергей Яковлев
Консультант
независимый специалист

02.06.2011

В статье рассматриваются характеристики AVL-деревьев (сбалансированных двоичных деревьев с минимальным временем поиска) и алгоритмы для работы с ними, реализованные на языке Си.

Введение

В этой главе будет рассматриваться другая разновидность бинарных поисковых деревьев - AVL-деревья или сбалансированные двоичные деревья с минимальным временем поиска по дереву. Это свойство AVL-деревьев обеспечивается их сбалансированностью, которая одновременно усложняет алгоритмы для вставки узлов в дерево и их последующего удаления. В рамках статьи будут рассмотрены характеристики AVL-деревьев и алгоритмы для работы с ними, реализованные на языке Си.

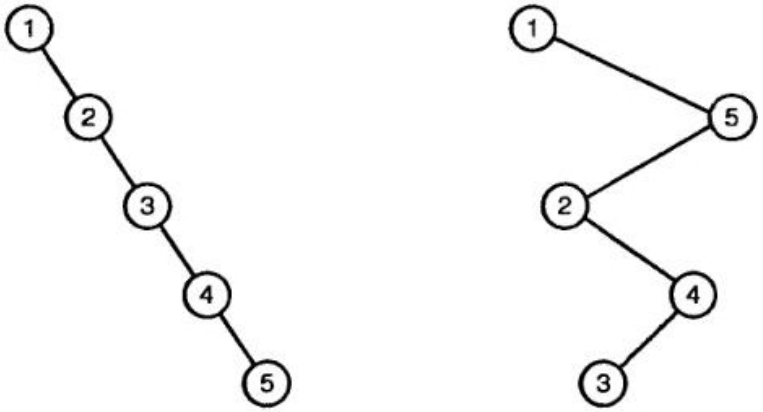
Сбалансированные деревья

Прежде чем переходить к обсуждению сбалансированных деревьев, необходимо вернуться к значению термина «высота дерева». Высота дерева – это число узлов от вершины до самого нижнего узла, т.е. максимальное количество узлов, по которому можно пройти, начав с корня дерева и перемещаясь в одном направлении.

Очевидно, что время поиска по двоичному дереву зависит от высоты дерева, и чем выше дерево - тем больше времени требуется на поиск определенного узла в дереве. Например, если высота дерева равна 5, то в наихудшем случае придется выполнить 5 сравнений.

Как было показано в прошлых статьях, из одного и того же набора узлов можно сконструировать различные варианты двоичных деревьев. В простейшем случае дерево можно построить так, что его высота будет равна числу элементов, и дерево выродится в односвязный список. В реальной жизни добавление и удаление элементов в дерево может происходить в произвольном порядке, что приводит к созданию различных деревьев, как показано на рисунке 1.

Рисунок 1. Примеры неупорядоченных деревьев



Чтобы создать сбалансированное двоичное дерево, необходимо изменить структуру узла, добавив в него дополнительный указатель на родительский узел, как показано в листинге 1.

Листинг 1. Структура узла для сбалансированного дерева

```
struct node
{
 int data;
 struct node * left;
 struct node * right;
 struct node * parent;
}
```

И определить функцию, которая будет искать «наследника» (ближайшее большее число) для указанного родительского узла.

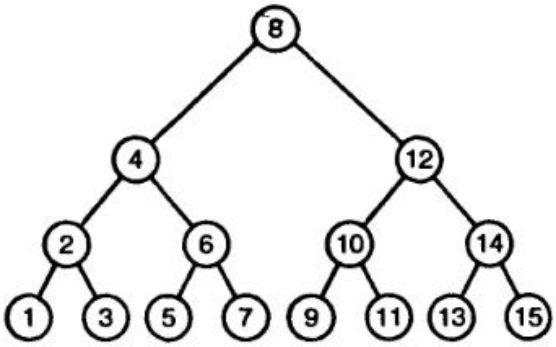
Листинг 2. Функция для поиска наследника

```
static struct node * successor(struct node * x)
{
 struct node * y;
 if(x->right != NULL)
 {
  y = x->right;
  while(y->left != NULL)
   y = y->left;
 }
 else
 {
  y = x->parent;
  while(y != NULL && x == y->right)
  {
   x = y ;
   y = y->parent;
  }
 }
 return y;
}
```

Если у узла имеется правый дочерний узел, то его наследником будет минимальное значение в правом поддереве (см. рисунок 2 – наследником узла **8** будет узел **9**, как минимальный в правом поддереве узла **8**). Если же правый потомок отсутствует, тогда

в поисках наследника необходимо перемещаться вверх и вправо (см. рисунок 2 – наследником узла **11** будет узел **12**).

Рисунок 2. Сбалансированное дерево



Указатель на родительский узел - это не единственный способ облегчить проход по дереву. Так, в дереве возможна ситуация, когда узел не имеет дочерних узлов. Это можно использовать для хранения указателей на другие части дерева, например, на предшественника и наследника данного узла. Для этого в узел потребуется добавить два битовых поля, как показано в листинге 3.

Листинг 3. Расширенная структура для описания узла дерева

```
struct node
{
 int data;
 struct node * left;
 struct node * right;
 unsigned l_thread:1;
 unsigned r_thread:1;
};
```

Эти поля используются следующим образом:

- если битовые поля содержат **0**, значит, указатели **left** и **right** указывают на левое и правое поддеревья данного узла;
- если битовые поля содержат **1**, значит, указатели **left** и **right** указывают на родителя и потомка данного узла.

Используя данные поля, можно переписать функцию для поиска наследника, как показано в листинге 4.

Листинг 4. Модифицированная функция для поиска наследника узла

```
static struct node * successor(struct  node * x)
{
 struct node * y;
 y = x->right;
 if(x->r_thread == 0)
 while(y->l_thread == 0)
  y = y->left;
 return y;
}
```

Для данной реализации дерева есть специальное название - дерево с полными ссылками. Также существуют деревья с правыми ссылками, где вместо двух битовых полей используется одно. Дерево с правыми ссылками более эффективно при поиске, так как на его прохождение требуется меньше времени.

Для получения сбалансированного двоичного дерева придется потратить определенные усилия. Так, если в дерево вставлять уже отсортированный массив, то оно может выродиться в список. Поэтому после каждой вставки необходимо проводить балансировку дерева.

А в общем случае после любой модификации дерева требуется выполнять балансировку дерева, чтобы минимизировать его высоту, так как меньше всего времени требуется на поиск по двоичному дереву с минимальной высотой. Балансировка влияет и на процессы вставки/удаления узлов, которые также будут проходить быстрее из-за более эффективного поиска. На данный момент не существует общепринятых стандартов для балансировки дерева, так что в этой области допускается определенная свобода при реализации новых или выборе из уже существующих алгоритмов балансировки.

AVL-деревья

Одним из известных типов сбалансированных двоичных деревьев является AVL-дерево, у которого коэффициент сбалансированности находится в пределах от -1 до +1. Полностью сбалансированное двоичное дерево с **n** узлами имеет высоту равную **log(n+1)** по основанию **2**, округленную до ближайшего целого числа. В листинге 5 приведен исходный код структуры, описывающей узел AVL-дерева.

AVL - аббревиатура, созданная из первых букв фамилий советских математиков: Г.М. Адельсона-Вельского и Е.М. Ландиса, придумавших данную структуру данных. Для определения AVL-дерева используется коэффициент сбалансированности - разница между высотой правого и левого поддеревьев.

Листинг 5. Структура, определяющая узел AVL-дерева

```
struct avl_node
{
 struct avl_node * link[2];
 int data;
 short bal;
};
```

В нем сразу видно отличие от классической реализации – указатели на поддеревья будут храниться в специальном массиве, а не в двух отдельных переменных. Переменная **bal** определяет значение коэффициента сбалансированности, и поэтому может принимать только значения из списка: **-1, 0, +1**.

Добавление узла в AVL-дерево

Процесс вставки узла в AVL-дерево отличается от добавления узла в обычное бинарное дерево. Так, для вставки нового узла в AVL-дерево, необходимо:

1. найти место, где должен будет находиться новый узел;

2. добавить узел в дерево на найденную позицию;
3. пересчитать коэффициенты сбалансированности для узлов, находящихся выше добавленного узла;
4. если дерево стало несбалансированным, то выполнить балансировку.

В листинге 6 приведен исходный код функции для вставки узла в AVL-дерево.

Листинг 6. Функция для вставки узла в AVL-дерево

```
int avl_insert(struct avl_tree * tree, int item)
{
 struct avl_node ** v, *w, *x, *y, *z;

/* если в дереве еще нет узлов */
 v = &tree->root;
 x = z = tree->root;
 if(x == NULL)
 {
  tree->root = new_node(tree,item);
  return tree->root != NULL;
 }

/* фрагмент №1 */
/* поиск подходящей позиции и последующая вставка элемента */
 for(;;)
 {
  int dir;
  /* такой элемент уже есть в дереве – функцию можно завершить */
  if(item == z->data) return 2;

  dir = (item > z->data) ;
  y = z->link[dir];
  if(y == NULL)
  {
   y = z->link[dir] = new_node(tree,item);
   if(y == NULL) return 0;
   break;
  }
  if(y->bal != 0)
  {
   v = &z->link[dir];
   x = y;
  }
  z = y;
 }

/* фрагмент №2 */
/* пересчет коэффициентов сбалансированности для узлов, затронутых вставкой */
 w = z = x->link[item > x->data];
 while(z != y)
  if(item < z->data)
  {
   z->bal = -1;
   z = z->link[0];
  }
  else
  {
   z->bal = +1;
   z = z->link[1];
  }

/* фрагмент № 3 */
/* балансировка при добавлении нового узла в левое поддерево */
 if(item < x->data)
 {
```

```
  if (x->bal != -1)
   x->bal --;
  else if(w->bal == -1)
  {
   *v=w;
   x->link[0] = w->link[1];
   w->link[1] = x;
   x->bal = w->bal = 0;
  }
  else
  {
   *v = z = w->link[1];
   w->link[1] = z->link[0];
   z->link[0] = w;
   x->link[0] = z->link[1];
   z->link[1] = x;
   if(z->bal == -1)
   {
    x->bal = 1;
    w->bal = 0;
   }
   else if(z->bal == 0)
    x->bal = w->bal = 0;
   else
   {
    x->bal = 0;
    w->bal = -1;
   }
   z->bal=0;
  }
 }
/* фрагмент № 4 */
/* балансировка при добавлении нового узла в правое поддерево */
 else
 {
  if( x->bal != +1)
   x->bal++;
  else if(w->bal == +1)
  {
   *v = w;
   x->link[1] = w->link[0];
   w->link[0] = x;
   x->bal = w->bal = 0;
  }
  else
  {
   *v = z = w->link[0];
   w->link[0] = z->link[1];
   z->link[1] = w;
   x->link[1] = z->link[0];
   z->link[0] = x;
   if(z->bal == +1)
   {
    x->bal = -1;
    w->bal = 0;
   }
   else if(z->bal == 0)
    x->bal = w->bal = 0;
   else
   {
    x->bal = 0;
    w->bal = 1;
   }
   z->bal = 0;
  }
 }
 return 1;
```

```
}
```

Так как функция **avl_insert** получилась довольно объемной, стоит рассмотреть ее в подробностях. В фрагменте №1 листинга 6 производится поиск позиции и вставка узла в AVL-дерево. В этом фрагменте используется временная переменная **z** для отслеживания текущего узла. Если значение добавляемого узла равно значению, содержащемуся в узле **z**, то функция завершается, так как бинарное дерево не может содержать узлы с дублирующимися значениями.

Затем в переменную **y** записывается указатель на следующий узел, находящийся в нужном поддереве (левом - если добавляемое значение меньше значения, содержащегося в текущем узле, или правом – если оно больше). Если **y** оказывается равным **NULL**, значит, у текущего узла отсутствует нужное поддерево и именно сюда и нужно вставлять добавляемое значение в виде нового узла. Для создания нового узла используется функция **new_node** приведенная в листинге 7.

Листинг 7. Функция для создания нового узла AVL-дерева

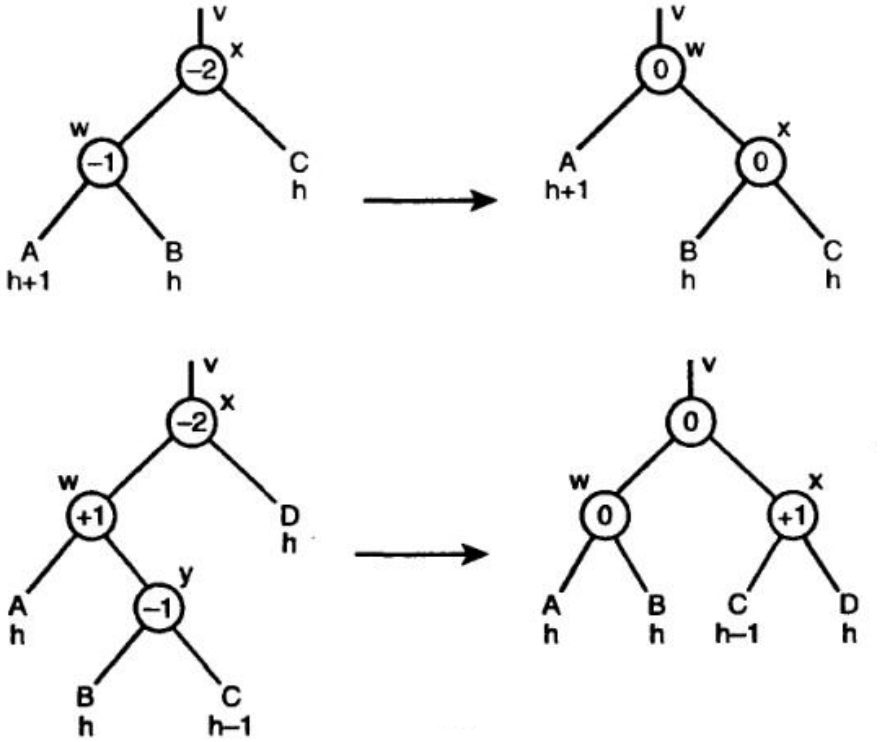
```
struct avl_node * new_node(struct avl_tree * tree, int item)
{
 struct avl_node * node = malloc(sizeof * node);
 node->data = item;
 node->link[0] = node->link[1] = NULL;
 node->bal = 0 ;
 tree->count ++ ;
 return node;
};
```

Переменные **x** и **v** во фрагменте №1 используются для отслеживания последнего пройденного узла с ненулевым коэффициентом сбалансированности. Так как если коэффициент сбалансированности равен **0**, то при вставке нового узла он изменится на **+1** или **-1**, и в этом случае балансировка не потребуется. Если же коэффициент изначально не равен **0**, то после вставки узла дерево придется балансировать.

Когда вставка нового узла **y** приводит к изменению коэффициентов для узлов, расположенных выше него, то требуется обновить коэффициенты для узлов, находящихся между узлами **x** и **y**. Эти действия выполняются во фрагменте №2 листинга 7.

Фрагмент № 3 листинга 6 соответствует ситуации, когда узел **y** находится в левом поддереве узла **x**. В этом случае, когда коэффициент сбалансированности для **x** изначально был равен **-1**, а после добавления узла стал равным **-2**, требуется выполнить балансировку. На рисунке 3 представлены два варианта выполнения балансировки.

Рисунок 3. Примеры балансировки дерева



В верхней половине рисунка 3 выполняется правая ротация узлов **w** и **x**, при этом узел **w** перемещается на место узла **x**. Узел **x** становится правым дочерним узлом для узла **w**, и в результате высота остается такой же, как и до вставки. В нижней половине рисунка 3 показана двойная ротация - сначала левая ротация узлов **w** и **y**, затем правая для узлов **y** и **x**.

Фрагмент №4 листинга описывает ситуацию с балансировкой, когда узел **y** находится в правом поддереве узла **x**.

Удаление узла из AVL-дерева

Удаление узла из AVL-дерева сложнее аналогичной операции для простого двоичного дерева и включает в себя следующие этапы:

1. поиск узла, который требуется удалить; в процессе поиска запоминаются пройденные узлы для выполнения последующей балансировки;
2. удаление искомого узла и обновление списка пройденных узлов;
3. выполнение балансировки и перерасчет коэффициентов сбалансированности.

Листинг 8. Функция для удаления узла из AVL-дерева

```
int avl_delete(struct avl_tree * tree, int item)
{
 struct avl_node * ap[32];
 int ad[32];

 int k = 1;
```

```
 struct avl_node ** y, * z;

 ad[0] = 0 ;
 ap[0] = (struct avl_node * ) &tree->root;

 z = tree->root;

/* поиск узла, выбранного для удаления */
 for(;;)
 {
  int dir;
  if(z == NULL)
   return 0 ;
  if (item == z->data)
   break;

  dir = item > z->data;
  ap[k] = z;
  ad[k++] = dir;
  z = z->link[dir];
 }

/* выполняется удаление узла из дерева */
 tree->count-- ;
 y = &ap[k - 1]->link[ad[k-1]];
 if(z->link[1] == NULL)
  *y = z->link[0];
 else
 {
  struct avl_node *x = z->link[1];
  if (x->link[0] == NULL)
  {
   x->link[0] = z->link[0];
   *y = x;
   x->bal = z->bal;
   ad[k] = 1;
   ap[k++] = x;
  }
  else
  {
   struct avl_node *w = x->link[0];
   int j = k++;
   ad[k] = 0 ;
   ap[k++] = x;
   while (w->link[0] != NULL)
   {
    x = w;
    w = x->link[0];
    ad[k] = 0 ;
    ap[k++] = x;
   }

   ad[j] = 1;
   ap[j] = w;
   w->link[0] = z->link[0];
   x->link[0] = w->link[1];
   w->link[1] = z->link[1];
   w->bal = z->bal;
   *y = w;
  }
 }
 free(z);

/* балансировка получившегося дерева */
 while(--k)
 {
  struct avl_node *w, *x;
```

```
 w = ap[k];
 if (ad[k] == 0 )
 {
  if (w->bal == -1)
  {
   w->bal = 0;
   continue;
  }
  else if (w->bal == 0 )
  {
   w->bal = 1;
   break;
  }

  x = w->link[1];
  if (x->bal > -1)
  {
   w->link[1]= x->link[0];
   x->link[0] = w;
   ap[k-1]->link[ad[k-1]] = x;
   if (x->bal == 0 )
   {
    x->bal = -1;
    break;
   }
   else
    w->bal = x->bal = 0 ;
  }
  else
  {
   z = x->link[0];
   x->link[0] = z->link[1];
   z->link[1] = x;
   w->link[1] = z->link[0];
   z->link[0] = w;
   if (z->bal == 1)
   {
    w->bal = -1;
    x->bal = 0;
   }
   else if (z->bal == 0 )
    w->bal = x->bal = 0;
   else
   {
    w->bal = 0;
    x->bal = 1;
   }
   z->bal = 0;
   ap[k-1]->link[ad[k-1]] = z;
  }
 }
 else
 {
  if (w->bal == 1)
  {
   w->bal = 0 ;
   continue;
  }
  else if (w->bal ==0)
  {
   w->bal = -1;
   break;
  }

  x = w->link[0];
  if (x->bal < 1)
  {
```

```
    w->link[0] = x->link[1]  ;
    x->link[1] = w;
    ap[k-1]->link[ad[k-1]] = x;
    if (x->bal == 0 )
    {
     x->bal = 1;
     break;
    }
    else
     w->bal = x->bal = 0;
   }
   else if (x->bal == 1)
   {
    z = x->link[1];
    x->link[1] = z->link[0];
    z->link[0] = x;
    w->link[0] = z->link[1];
    z->link[1] = w;
    if (z->bal == -1)
    {
     w->bal = 1;
     x->bal = 0;
    }
    else if (z->bal == 0 )
     w->bal = x->bal = 0 ;
    else
    {
     w->bal = 0 ;
     x->bal = -1;
    }
    z->bal = 0;
    ap[k-1]->link[ad[k-1]] = z;
   }
  }
 }
 return 1;
}
```

В начале этой функции идет поиск узла, который необходимо удалить. Каждый пройденный узел **z** сравнивается с целевым значением **item**. Пройденные узлы помещаются в массив **ap**, а направления, по которым осуществляется движение, сохраняются в массив **ad**.

После этого происходит удаление узла из дерева, при этом возможны те же три варианта развития событий, как и при удалении узла из обычного двоичного дерева. Однако для AVL-дерева необходимо дополнительно отслеживать список узлов, расположенных выше удаляемого узла. После удаления выполняется балансировка, как и в случае с добавлением узла в дерево, но не по левому/правому поддеревьям, а по расположенным сверху элементам.

Заключение

Несмотря на преимущества, предоставляемые AVL-деревьями при выполнении поиска по дереву, их использование затрудняется необходимостью балансировки после выполнения операций добавления / удаления узлов. Алгоритм балансировки представляет основную проблему, так как его неудачная реализация может негативно влиять на производительность программы, использующей данный тип дерева.

В статье были рассмотрены как теоретические аспекты работы с AVL-деревьями, так и основные алгоритмы для работы с ними, представленные в виде функций на языке Си. Изучив представленную реализацию и теоретические аспекты балансировки дерева, пользователь сможет сам разработать алгоритм балансировки AVL-дерева на Python.

Об авторе

Сергей Яковлев

Яковлев Сергей — независимый разработчик с многолетним опытом прикладного и системного программирования; вносит вклад в развитие open-source на своем персональном сайте www.iakovlev.org. Консультант.