Дерево Фибоначчи

Деревом Фибоначчи называется такое двоичное дерево поиска, в котором, как и в матричном дереве, каждому ключу предоставлено заранее определённое место. Данная структура схожа с матричным деревом ещё и в том, что в качестве ключей используются только элементы множества натуральных чисел.

Для объяснения чисел Фибоначчи использовались рекуррентные соотношения, или принцип рекурсии: каждое следующее число находится с помощью двух предыдущих (в данном случае, равно их сумме). Естественно, у рекурсии должна быть база: необходимо явно назвать два первых элемента последовательности Фибоначчи. В книгах для школьников чаще всего этим «базовым» элементам присваиваются номера 1 и 2. В нашем случае нумерацию уместно начать с нуля.

Итак, два младших числа Фибоначчи заданы явно, $F_0 = 1$, $F_1 = 1$, для прочих чисел $F_i = F_{i-1} + F_{i-2}$ $(i = 2, 3, 4, \ldots)$. Таким образом, $F_2 = 2$, $F_3 = 3$, $F_4 = 5$, $F_5 = 8$, $F_6 = 13$, $F_7 = 21$, $F_8 = 34$, … .

Дерево Фибоначчи порядка K определяется следующим образом:

- Если $K = 0$, то дерево Фибоначчи ***пусто***;
- если $K = 1$, то дерево Фибоначчи состоит из единственного узла, который содержит ключ $F_1 = 1$;
- если $K \geq 2$, то корень дерева Фибоначчи содержит ключ F_K, левое поддерево есть в точности дерево Фибоначчи порядка $K - 1$, правое поддерево есть дерево Фибоначчи порядка $K - 2$ с ключами в узлах, *увеличенными* на F_K.

Порядок дерева на единицу больше его высоты.

Ниже показаны примеры деревьев Фибоначчи небольших высот.

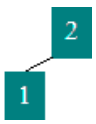

Рис. 1. Полное дерево высоты 1 $(K = 2)$.

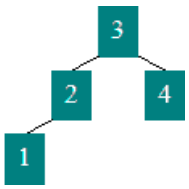

Рис. 2. Полное дерево высоты 2 $(K = 3)$.

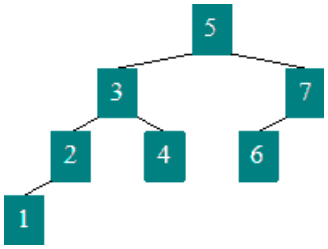

Рис. 3. Полное дерево высоты 3 $(K = 4)$.

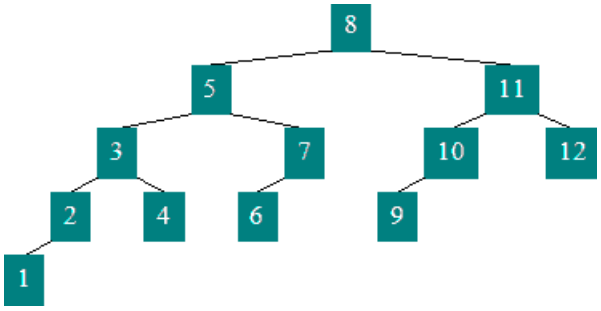

Рис. 4. Полное дерево высоты 4 ($K = 5$).

Дерево Фибоначчи является АВЛ–деревом. Обратное неверно.

У АВЛ–дерева высота левого и правого поддеревьев для некоторых узлов может быть одинаковой. У дерева Фибоначчи левое поддерево всегда выше, чем правое, на единицу. В этом смысле дерево Фибоначчи, пожалуй, несколько хуже АВЛ–дерева. Но программировать работу с ним более приятно. Дерево Фибоначчи математически красиво.

Порядок дерева Фибоначчи можно определить по значению его корневого ключа: если этот ключ оказался равным F_K, дерево имеет порядок K. Если корневой ключ не является числом Фибоначчи, то и дерево не является деревом Фибоначчи. Можно, правда, рассуждать о фрагменте дерева Фибоначчи, на равных правах с фрагментом матричного дерева, но только не в этом учебном материале.

Прочие ключи дерева можно «вычислять» по следующему правилу. Всякий узел возглавляет поддерево некоторого порядка k и содержит некий ключ Φ_k. Левый потомок узла (который существует только при $k \geq 2$) содержит ключ $\Phi_k - F_{k-2}$, правый потомок узла (который существует только при $k \geq 3$) содержит ключ $\Phi_k + F_{k-2}$.

Этой информации, а также рекурсивности структуры, достаточно, чтобы построить *полное* дерево Фибоначчи заданной высоты.

При внесении каждого **нового** ключа в дерево Фибоначчи первым делом следует выяснить, «вмещается» ли ключ в дерево при его-то высоте. Если «вмещается», дальше всё очень просто: сравниваем **новый** ключ с текущим (на первом шаге – с корневым) ключом, если новый ключ меньше текущего, углубляемся в левое поддерево, если больше – в правое. Если (о ужас!) «нужного» поддерева нет, мы его *начинаем* создавать. Ключ, который должен возглавить вновь создаваемое дерево, мы вычислять уже умеем.

Дерево Фибоначчи заданной высоты является *неполным*, если какой-то из узлов должен, по структуре дерева, иметь потомка, но не имеет его. То есть, *неполное* дерево «подрезано» снизу, отсутствуют его «концевые» поддеревья. Дерево, имеющее нарушение («подрезание») структуры в «середине», уже не вправе называться деревом Фибоначчи.

Если **новый** ключ *не вмещается* в дерево Фибоначчи имеющейся высоты, придётся эту высоту увеличить. Новый корень должен стать предком старого корня (звучит-то как!), или непосредственным, или дальним. Увеличение высоты означает, что следует выполнить два действия.

1. Поставить во главе дерева новый корневой узел, проложить от него «дорожку» к старому корю.

2. Проложить от нового корня «дорожку» к **новому** ключу.

Что значит «Проложить дорожку»? Ну, скажем, дорожку к **новому** ключу? Пользуясь отношением «больше–меньше», выясняем, в левое или в правое поддерево следует углубиться в поисках места для **нового** ключа. Если нужного левого или правого потомка нет, создаём его. Если оказалось, что только что созданный потомок содержит **новый** ключ, процесс прекращается. Процесс может НЕ прекратиться только в том случае, если ранее созданное дерево не является деревом Фибоначчи. Но такой фильм ужасов – не для нас.

«Прокладывание дорожки» означает, кроме всего прочего, построение такого нового фрагмента дерева, который представляет вырождение дерева в линейный однонаправленный список. Дерево после такого «прокладывания» уж точно будет неполным. Новые узлы, появляющиеся в дереве, можно условно разделить на «полезные» и «паразитные». Полезный узел внесён в дерево потому, что его ключ в явном виде «просили» внести. Паразитный ключ был вставлен где-то между новым корневым узлом и узлом с **новым** ключом либо между новым корневым узлом и старым корневым узлом, потому что так требовала структура дерева. Паразитный узел имел, сразу после создания, только одного потомка. И только новый корневой узел, если он появился, *мог* получить сразу двух потомков.

Паразитный узел может быть помечен, скажем, с помощью значения true логического поля bParasitic, совмещённого в памяти с полем bRed для красно-чёрного дерева. Теоретически, дерево может быть избавлено от цепочки подряд идущих (в отношении предок – непосредственный потомок) паразитных узлов. Тогда средняя длина пути в дереве уменьшится. Но, строго говоря, дерево перестанет быть деревом Фибоначчи.

Пока не прояснён вопрос, что значит, **новый** ключ *вмещается* в дерево Фибоначчи имеющейся высоты.

Пусть G_K есть максимальный из ключей в дереве Фибоначчи порядка K. Два младших члена последовательности задаются явно, $G_1 = 1$, $G_2 = 2$, для прочих чисел справедливо рекуррентное соотношение $G_i = G_{i-1} + G_{i-2} + 1$ ($i = 3, 4, 5, \ldots$). Таким образом, $G_3 = 4$, $G_4 = 7$, $G_5 = 12$, $G_6 = 20$, $G_7 = 33$, $G_8 = 54$, … .

Формула общего члена:

$$G_K = \frac{5+3\sqrt{5}}{10}\left(\frac{1+\sqrt{5}}{2}\right)^K + \frac{5-3\sqrt{5}}{10}\left(\frac{1-\sqrt{5}}{2}\right)^K - 1 .$$

Если в дерево требуется внести новый ключ N, следует найти минимальное K, при котором $N \le G_K$. Поскольку $G_K \approx \frac{5+3\sqrt{5}}{10}\left(\frac{1+\sqrt{5}}{2}\right)^K$ при больших K, для больших N можно примерно оценить порядок дерева по формуле

$$K \approx \left\lceil \frac{\ln\left(\frac{3\sqrt{5}-5}{2}(N+1)\right)}{\ln\left(\frac{\sqrt{5}+1}{2}\right)} \right\rceil . \qquad (1)$$

Здесь $\lceil x \rceil$ есть минимальное целое, большее либо равное x.

Простая формула (1) названа приближённой. В подавляющем большинстве случаев она даёт верный результат. Однако, она даёт завышенный на единицу результат при $N = G_i$, если i – нечётное число. Пытливому читателю предлагается вывести точную формулу.

Алгоритм включения в дерево Фибоначчи

Метод Include(int iKey) вносит *новый* ключ N в дерево Фибоначчи порядка k. Это дерево возглавляется узлом, содержащим ключ F_k. Если дерево пусто, $k = 0$.

Сначала находится наименьшее K, при котором $N \le G_K$.

Если $K \le k$, место новому ключу в дереве находится так же, как это делалось в матричном дереве.

Если $K > k$, создаётся новый паразитный корневой узел с ключом F_K. От него нужно проложить две дорожки к потомкам: к узлу N и к старому корневому узлу.

Алгоритм исключения из дерева Фибоначчи

Метод Exclude(int iKey) удаляет ключ из дерева Фибоначчи точно так же, как это делалось в матричном дереве. Единственное отличие состоит в отсутствии избавления дерева от начальной цепи паразитных узлов.

Текст модуля класса дерева Фибоначчи

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.InteropServices;
using System.Windows.Forms;
using System.Drawing;
using System.IO;

namespace WFABinaryTrees
{
   class ClassFTree : ClassBaseTree
   {
      private int[] mF, mG;
      private int nHeight;

      public ClassFTree(PictureBox pictureBoxParam)
      {
         ImageGraph = pictureBoxParam;

         CommonPart();

         nHeight = -1;
      }

      private void IncludeKey(int iKey)
      {
         int iKeyCurr, iKeyNext, iKeyRoot, idKey, iHeight, nHeightOld;
         MyNodeType pCurr, pRootNew;

         if (pRoot == null)
            iKeyRoot = -1;
         else
         {
            pCurr = pRoot;
            iKeyCurr = pCurr.iKey;

            while (true)
            {
               if (iKey < iKeyCurr)
               {
                  pCurr = pCurr.pLeft;
                  if (pCurr == null) break;
                  iKeyCurr = pCurr.iKey;
               }
               else
```

```
        if (iKey > iKeyCurr)
        {
            pCurr = pCurr.pRight;
            if (pCurr == null) break;
            iKeyCurr = pCurr.iKey;
        }
        else
        {
            if (pCurr.AuxField.bParasitic)
                pCurr.AuxField.bParasitic = false;
            else
                KeyIsAlready(iKey, "Фибоначчи");

            return;
        }
    }

    iKeyRoot = pRoot.iKey;
}

nHeightOld = nHeight;
nHeight = -1;

while (true)
{
    nHeight++;

    if (nHeight > nHeightOld)
    {
        Array.Resize(ref mF, nHeight + 2);
        Array.Resize(ref mG, nHeight + 1);

        if (nHeight == 0)
        {
            mF[nHeight] = 1;
            mF[nHeight + 1] = 1;
            mG[nHeight] = 1;
        }
        else
        if (nHeight == 1)
        {
            mF[nHeight + 1] = 2;
            mG[nHeight] = 2;
        }
        else
        for (iHeight = 2; iHeight <= nHeight; iHeight++)
        {
            mF[nHeight + 1] = mF[nHeight] + mF[nHeight - 1];
```

```
                mG[nHeight] = mG[nHeight - 1] + mG[nHeight - 2] + 1;
            }
        }

        if (iKey <= mG[nHeight])
            break;
    }

    if (nHeight < nHeightOld) nHeight = nHeightOld;

    if (nHeight > nHeightOld)
    {
        if (pRoot == null)
        {
            iKeyCurr = mF[nHeight + 1];
            NewNode(ref pRoot, iKeyCurr);

            if (iKey == iKeyCurr)
            {
                pRoot.AuxField.bParasitic = false;
                return;
            }
            else
                pRoot.AuxField.bParasitic = true;
        }
        else
        {
// Прокладываем дорожку к старому корню.

            iKeyRoot = pRoot.iKey;

            iHeight = nHeight;

            iKeyCurr = mF[nHeight + 1];
            pRootNew = null;
            NewNode(ref pRootNew, iKeyCurr);
            pRootNew.AuxField.bParasitic = true;
            pCurr = pRootNew;

            while (true)
            {
                idKey = mF[iHeight - 1];
                iKeyNext = -1;

                if (iKeyRoot < iKeyCurr)
                {
                    iKeyNext = iKeyCurr - idKey;
                    if (iKeyNext < 1) ItIsUnimpossible("F401");
```

```
            if (iKeyNext == iKeyRoot)
            {
               pCurr.pLeft = pRoot;
               break;
            }

            iHeight--;

            NewNode(ref pCurr.pLeft, iKeyNext);
            pCurr = pCurr.pLeft;
            pCurr.AuxField.bParasitic = true;
            iKeyCurr = iKeyNext;
         }
         else
         if (iKeyRoot > iKeyCurr)
         {
            iKeyNext = iKeyCurr + idKey;
            if (iKeyNext > mG[nHeight]) ItIsUnimpossible("F402");

            if (iKeyNext == iKeyRoot)
            {
               pCurr.pRight = pRoot;
               break;
            }

            iHeight -= 2;

            NewNode(ref pCurr.pRight, iKeyNext);
            pCurr = pCurr.pRight;
            pCurr.AuxField.bParasitic = true;
            iKeyCurr = iKeyNext;
         }

         if (iKeyNext == iKeyRoot)
            break;

         if (iHeight < 0) ItIsUnimpossible("F403");
      }

      pRoot = pRootNew;

   }    //  if (pRoot != null)
}     //  if (nHeight > nHeightOld)

iHeight = nHeight;

pCurr = pRoot;
```

```
iKeyCurr = pCurr.iKey;

if (iKeyCurr == iKey)
{
   pCurr.AuxField.bParasitic = false;
   return;
}

idKey = mF[iHeight - 1];

while (true)
{
   iKeyNext = -1;

   if (iKey < iKeyCurr)
   {
      iKeyNext = iKeyCurr - idKey;
      if (iKeyNext < 1) ItIsUnimpossible("F404");

      iHeight--;

      if (pCurr.pLeft == null)
      {
         NewNode(ref pCurr.pLeft, iKeyNext);
         pCurr.pLeft.AuxField.bParasitic = true;
      }
      else
      if (pCurr.pLeft.iKey != iKeyNext) ItIsUnimpossible("F405");

      pCurr = pCurr.pLeft;
      iKeyCurr = iKeyNext;
   }
   else
   if (iKey > iKeyCurr)
   {
      iKeyNext = iKeyCurr + idKey;
      if (iKeyNext > mG[nHeight]) ItIsUnimpossible("F406");

      iHeight -= 2;

      if (pCurr.pRight == null)
      {
         NewNode(ref pCurr.pRight, iKeyNext);
         pCurr.pRight.AuxField.bParasitic = true;
      }
      else
      if (pCurr.pRight.iKey != iKeyNext) ItIsUnimpossible("F407");
```

```
            pCurr = pCurr.pRight;
            iKeyCurr = iKeyNext;
        }

        if (iKeyNext == iKey)
        {
            pCurr.AuxField.bParasitic = false;
            break;
        }

        if (iHeight < 0) ItIsUnimpossible("F408");

        idKey = mF[iHeight - 1];
    }

}

private void SettleWithColors()
{
    foreach (MyNodeType pNode in NodeList)
    {
        pNode.colorText = Color.White;

        if (pNode.AuxField.bParasitic)
        {
            pNode.colorBack = Color.Red;
            pNode.colorBorder = Color.Red;
        }
        else
        {
            pNode.colorBack = Color.DarkCyan;
            pNode.colorBorder = Color.DarkCyan;
        }
    }
}

public override void Include(int iKey)
{
    IncludeKey(iKey);

    SettleWithColors();
}


private void ExcludeKey(ref MyNodeType pStart, int iKey)
{
// Рекурсивный метод удаления ключа из матричного дерева.
// Если есть паразитные узлы, есть специфика работы с ними.
```

```
        if (pStart == null)
        {
            KeyIsNotYet(iKey, "Фибоначчиевом");
            return;
        }

        if (iKey < pStart.iKey)
        {
            ExcludeKey(ref pStart.pLeft, iKey);

            if (pStart.AuxField.bParasitic)
                if (pStart.pLeft == null && pStart.pRight == null)
// Потомков нет => паразитный узел можно удалять.
                {
                    NodeList.Remove(pStart);
                    pStart = null;
                }
        }
        else
            if (iKey > pStart.iKey)
            {
                ExcludeKey(ref pStart.pRight, iKey);

                if (pStart.AuxField.bParasitic)
                    if (pStart.pRight == null && pStart.pLeft == null)
// Потомков нет => паразитный узел можно удалять.
                    {
                        NodeList.Remove(pStart);
                        pStart = null;
                    }
            }
            else
            {
// iKey == pStart.iKey.  Ключ, размещённый в узле “pStart” требуется удалить.

                if (pStart.pLeft == null && pStart.pRight == null)
// Потомков нет => узел можно удалять.
                {
                    NodeList.Remove(pStart);
                    pStart = null;
                }
                else
// Потомки есть => узел надо пометить, как паразитный. Удалять нельзя.
                    pStart.AuxField.bParasitic = true;
            }
    }
```

```
    public override void Exclude(int iKey)
    {
        ExcludeKey(ref pRoot, iKey);

        SettleWithNodePositions(null);
        SettleWithColors();
    }

  }
}
```