УДК 330

СПОСОБЫ ОРГАНИЗАЦИИ ДВОИЧНОГО ДЕРЕВА ПОИСКА И ДЛИННОЙ АРИФМЕТИКИ НА ЯЗЫКЕ ПРОГРАММИРОВАНИЯ C#

Попова Ольга Борисовна
к.т.н., доцент
Богацкий Никита Валерьевич
ФГБОУ ВО «Кубанский государственный технологический университет»

Аннотация: В данной статье предложены, описаны и реализованы существующие и новые способы представления двоичных деревьев поиска и длинной арифметике. Рассматриваемое решение реализовано с использованием Microsoft Visual Studio 2017. Результаты представляют собой теоретические выводы, основанные на практических исследованиях. Разработанный метод является альтернативой встроенному классу. Несмотря на более быструю обработку операций, предлагаемый метод проигрывает по скорости инициализации и размещении в памяти. Зато этот метод можно значительно ускорить, переработав инициализацию и сами операции с помощью сдвигов, при сохранении идеи деления числа на блоки. Предлагаемые способы можно использовать для хранения любых значений.

Ключевые слова: visualstudia 2017, методы реализации, бинарное дерева посиска, длинная арифметика, время обработки.

METHODS FOR ORGANIZING BINARY SEARCH TREE AND LONG ARITHMETICS IN C # PROGRAMMING LANGUAGE

Popova Olga Borisovna
Bogatsky Nikita Valerievich

Abstracts: this article proposes, describes and implements existing and new ways to represent binary search trees and long arithmetic. The solution considered is implemented using Microsoft Visual Studio 2017. Results conclude theoretical findings, based on practical research. As a result, this method shows itself as a possible alternative to the built-in class. Despite the faster processing of operations, the proposed method loses in terms of speed of initialization and memory. This

method can be significantly accelerated by reworking the initialization and the operations themselves using shifts, while maintaining the idea of dividing a number into blocks, the proposed method can also be redone to store any values.

Key words: visual studia 2017, implementation methods, search binary tree, long arithmetics, time of processing.

Для исследования были выбраны две темы: длинная арифметика [1, 2] и дерево бинарного поиска [3]. В ходе изучения данных тем было обнаружено, что можно ускорить время их обработки на языке C#, а также применить полученные сведения и в других языках программирования. Для начала рассмотрим тему – длинная арифметика. Следующий метод был реализован на языке программирования C#, но его основана идея может быть реализована на других языках. В основу данного метода легла идея разбиения числа, но не математическое, а физическое. Данный метод представлять длинное число, тесть число, которое длиннее машинного слова, в виде блоков из входящих цифр. Рассмотрим поля реализованного класса и конструктор [4].

```
public readonlystructmyBigInt
{internalreadonlyint A, B, C, flag;                    // ABC - блоки числа, flag - знак числа
private constintoverSize = 100000000;        //число, часто применяемое для вычисления
public myBigInt(int A, int B, int C, intflag) //конструктор принимающий на вход
                                                    блоки и флаг
{if (A == 0 && B == 0 && C == 0)                    //что бы ноль всегда был
                                                    положительным
flag = 0;
this.flag = flag;
this.A = A;
this.B = B;
this.C = C; } }
```

Данный метод имеет предел длинны числа, хоть её можно расширить путём воды массива, для простоты будет рассмотрена данная реализация. У нас есть четыре переменных, A,B и C отвечающие за блоки числа и flag, который принимает значения в зависимости от знака длинного числа. Блок B и блок C имеют условную длину в 8 цифр, которая контролируется при операциях и выводи числа. В блоке A при данной реализации в основном

храниться число длинной в две цифры. Это был вспомогательный конструктор, который на вход получает уже готовые блоки и флаг и «собирает» число. Теперь стоит рассмотреть основные конструкторы.

Конструктор, на вход которого принимается значения типа long.

```
PublicmyBigInt(long value)                    //конструктор
{stringbuf = Convert.ToString(value);
string A = “”;
string B = “”;
string C = “”;                                //инициализация блоков
for (int(C) = buf.Length – 1; (C)>= 0; i--)
{if (buf[i] == '-')
  continue;                                   //если минус, то пропускаем его
  if (buf.Length – i<= 8)
      C = buf[i] + C;
   elseif (buf.Length – C<= 16)
            B = buf[i] + B;
       else
            A = buf[i] + A;}
  if (value < 0) this.flag = -1;              //проверказнака
    else
     this.flag = 1;
     this.A = convert(A);
     this.B = convert(B);
     this.C = convert(C);                     //запись блоков }
```

В начале число конвертируется в текст, для удобного извлечения каждой цифры. Далее, игнорируя минус если он есть, заполняем блок С и В восемью цифрами каждый, оставшиеся записываем в старший блок. После чего флаг принимает значения входящего числа, хоть и можно сделать, что он принимал значения от того был ли встречен минус на входном значении, но для данной реализации − это не принципиально. Из данных участков кода видна основная идея метода, суть которого заключается в представлении длинного числа в виде нескольких маленьких и при операциях с данным числом задействовать только нужные блоки, а не всё число, что должно положительно сказываться на скорости обработки.

Теперь можно рассмотреть реализацию самих операций, но для начала стоит упомянуть два вспомогательных метода.

```
static myBigInt revers(myBigInt x, int flag)  //измененияфлага
{ return new myBigInt(x.A, x.B, x.C, x.flag * flag); }
```

Данный метод нужен для удобного изменения знака числа. Реализован через умножения переменной flag на число. И второй метод нужен для удобной конвертации пустой строки в 0, так как для записи числа мы используем строки, данная операция будет производится не один раз.

```
private static int convert(string s)          //вспомогательный метод, для перевода
                                              пустой строки в 0
{if (s == "") return 0;
  else return Convert.ToInt32(s); }
```

Перейдём к двум основным методам, которые реализуют сложения и вычитание положительных числе. Важно отметить, что на вход могут поступать только положительные числа, а на выходе могу получаться и отрицательные. Первый метод под названием add.

```
static myBigInt add(myBigInt left, int right) //сложения положительных myBigInt
                                                и int
{ int C = left.C + right;                     //складываем с младшим блоком
  intflag = C / overSize;                     //если блок вышел за 8 цифр
  if (flag != 0)  C -= flag * overSize;       //вычитаем лишнее
  int B = left.B + flag;                              //прибавляем лишнее из
прошлого блока
  flag = B / overSize;                        //проверка на выход за 8 цифр
  if (flag != 0)  B -= flag * overSize;       //вычитаем лишнее
  int A = left.A + flag;                              //прибавляем возможный
перенос
  return new myBigInt(A, B, C, left.flag); } //возвращаем готовое число
static myBigInt add(myBigIntleft, myBigIntright)      //сложение двух
                                                      положительных myBigInt
{int C = left.C + right.C;                    //складываем младшие блоки
int flag = C / overSize;                      //проверка на размер блока
if (flag != 0)   C -= flag * overSize;        //вычитание лишнего
int B = left.B + right.B + flag;                    //сложения средних блоков и
переноса
flag = B / overSize;                          //проверка
if (flag != 0)  B -= flag * overSize;
int A = left.A + right.A + flag;              //сложения последних блоков
```

return new myBig Int(A, B, C, left.flag); //собирания числа }

Данный метод реализован для сложения длинного числа с переменного типаint и для сложения с таким же длинным числом. При сложении с переменного типа int, сначала младший блок складывается с числом. Может получится так, что размер блока выйди за условные ограничения в 8 цифр. Делением на константу мы проверяем есть ли переполнения нашего блока, если да, то мы избавляем блок от него, а сам остаток добавляем к следящему блоку. В реализации, где складываются два длинных числа сначала складываются все блоки, а потом смотреться размер каждого и выполняется перенос. Таким образом мы не задействуем лишние блоки, только те, что нужны для выполнения операции. Второй основной метод – метод вычитания положительных числе, на выходе может получится и отрицательное число.

```
static myBigInt subtract(myBigIntleft, intright) //вычитает int из myBigInt с
                                                 положительным знаком
    {int x1 = right / overSize;                  //делим int на два блока
     int x2 = right % overSize;
     if (x1 != 0 && x2 == 0)                     //если int = overSize
      {x1 = 0;
       x2 = overSize;  }
     int B = left.B - x1;                        //вычитание второго
блока
     int C = left.C - x2;                        //вычитание младшего
блока
     int A = left.A;                             // определения старшего
блока
     if (C < 0&& (A != 0 || B != 0))       //занимаем разряд, если надо и если
есть где
       {  C += overSize;
          B -= 1;    }
          if (B < 0&& A != 0)              //занимаем по тем же правилам для
среднего блока
            {  B += overSize;
               A -= 1;   }
          else if (B < 0)                  //если число получилось отрицательным
          {B *= -1;
          return new myBigInt(A, B, C, -1);   }
```

```
            return new myBigInt(A, B, C, left.flag);   }     //возвращает полученное
число
static myBigInt subtract(myBigInt left, myBigInt right) //вычитание двух
                                                          положительных myBigInt
        {int C = left.C - right.C;
         int B = left.B - right.B;
         int A = left.A - right.A;          //вычитаем блоки
         if (C > 0 && B < 0)                //если должно выйти отрицательное
        { C = right.C - left.C;
          B = right.B - left.B;}
         if (A <= 0 && B <= 0 && C <= 0)   //если негде занимать
        {return new myBigInt(A * -1, B * -1, C * -1, -1); }
        if (C < 0&& (A != 0 || B != 0))          //если есть где занимать
          { C += overSize;
            B -= 1;  }
        if (B < 0&& A != 0)                      //занимаем для среднего блока
        { B += overSize;
           A -= 1; }
          elseif (B < 0)
            { B *= -1;
              return new myBigInt(A, B, C, -1); }
        return new myBigInt(A, B, C, left.flag); }
```

Вычитание имеет более сложную реализацию, относительно алгоритма сложения. Связанно это с возможностью получения отрицательного результата. Первое принципиальное различие в том, что мы делим входное число на блоки, так же обрабатываем случай, если переменная типа int будет равна переменной overSize. После чего идёт обычная инициализация и начальная обработка блоков. После чего сразу же идёт проверка на получения отрицательного результата. Если же результат будет положительным нужно помнить, что если при сложении мы делали перенос, то при вычитании мы должны позаботиться о займе лишнего разряда, но и во время занимания для младшего разряда, может получиться отрицательно число. При вычитании двух длинных чисел реализация примерно такая же, как и при вычитании переменной типа int от длинного числа. Только у нас уже есть готовые блоки, а все остальные проверки на отрицательный результат и заём лишнего разряда остаются теме же. Таким образом в вычитании блоки в отдельности, хоть и

проверки на отрицательный результат плохо влияют на время обработки операции, данный метод остаётся быстрее встроенного. Как и говорилось ранее, данные методы не могут обрабатывать отрицательные числа на входе. Для таких случаев, эти методы вызываются через перегруженные операторы с нужными параметрами. Подробно можно рассмотреть один оператор, остальные сделаны по образу и подобию.

```
public static myBigIntoperator +(myBigInt left, int right) //сложения myBigInt и int с любым знаком
 {boolrightFlag = false;
  if (right < 0)
    { rightFlag = true;
      right *= -1;   }
  boolleftFlag = false;
  if (left.flag< 0) leftFlag = true;
  if (leftFlag&&rightFlag)
   {return add(left, right);  }
  if (leftFlag)
   {return revers(subtract(revers(left, -1), right), -1);   }
  if (rightFlag)
   {return subtract(left, right);}
  return add(left, right);    }
```

В начале мы проверяем знаки входных значений. Далее если, что-то отрицательно в метод это будет поступать с положительным знаком. Корректный результат получается благодаря избавления минусов у входных значений и добавлений его к результату. В конце стоит так же упомянуть то, как это число представляется при выводе на экран.

```
public overridestringToString()
{string A = Convert.ToString(this.A);
 string B = Convert.ToString(this.B);
 string C = Convert.ToString(this.C);    //переводим блок в текст
if (A == "0")   A = "";                   //если ноль, то не выводим
if (B == "0"&& A == "")        B = "";      //если и A и B пустой, то не выводим их
  else if (A != "")                       //если A не пустой, а B пустой
   { int bLeng = 8 - B.Length;           //заполняем блок до 8 цифр
     for (inti = 0; i<bLeng; i++) B = '0' + B;       }
```

```
if (B != "")                                    //если B не пустой заполняем C до 8 цифр
{int cLeng = 8 - C.Length;
 for (inti = 0; i<cLeng; i++)    C = '0' + C;    }
 string res = "";
 if (this.flag == -1) res += '-';              //добавляем знак
 res += A;   res += B;   res += C;
 return res;    }
```

Это важный метод. Если методы реализующие сложения и вычитаний контролируют размер блока во время обработки, то данный метод при выводе на экран. Основная идея в том, что, если у нас перед левым блоком, есть другой блок, но мы выводим левый блок, заполняя его нулями до размера в восемь цифр. В конце учитываем знак. Представлены результаты тестирования реализации длинной арифметики и краткий вывод по полученным результатам в таблице 1.

Таблица 1

Сравнения скорости работы предложенного и встроенного метода

	1000000	10000000	100000000
Инициализация BigIngere	0:00:02	0:00:14	0:01:40
Инициализация myBigInt	0:00:59	0:09:18	1:27:50
BigInteger + i	0:00:08	0:01:18	0:07:39
myBigInt + i	0:00:05	0:00:40	0:06:30
BigInteger - i	0:00:08	0:01:18	0:07:26
myBigInt - i	0:00:05	0:00:44	0:06:07
BigInteger + BigInteger	0:00:08	0:01:16	0:08:10
myBigInt + myBigInt	0:00:06	0:00:37	0:07:23
BigInteger - BigInteger	0:00:09	0:01:38	0:09:01
myBigInt - myBigInti	0:00:06	0:00:57	0:06:47

Сразу же стоит заметить значительно преимущество встроенного класса по времени инициализации. Это связанно с тем, что в BigIntegerв конструкторах используются 32 разрядная система, которая имеет более быстрое время обработки. В предложенном же методе, цифра разбивается на блоки, но также возможно реализовать разбиение с помощью сдвигов, что значительно ускорит общую работу метода. Из остальной таблицы видно, что

разрыв между классами уменьшается, что и подтверждает идею предложенного метода, которая заключается в обработки части числа.

Перейдём к раскрытию следующей темы. Разработанный класс предлагает не только обычную реализацию дерева бинарного поиска, но и его улучшенную версию [5]. Для начала рассмотрим элемент класса, которые реализуют базовые принципы.

```
classMyTree                    //класс дерева
  {classMyNode                 //класс ячейки
    {public MyNode Left;       //левая ячейка
     public MyNode Right;      //правая ячейка
     public int data;          //значение ячейки
     public MyNode()           //конструктор ячейки
       { Left = null;
         Right = null; } } }
```

Данный класс использует вариант реализации через основной класс дерева и дополнительный класс ячейки. Класс ячейки содержит ссылки на правого и левого ребёнка, а также поля значения ячейки.

```
publicMyTree()                 //конструктор по умолчанию
    {root = null; }
```

Конструктор по умолчанию только устанавливает значения корня пустым.

```
public void Add(int i)                    //добавления элемента
    {MyNode childe = new MyNode(); //добавляемый элемент
     childe.data = i;                     //значение добавляемого элемента
     if (root == null)                    //если дерева нет
       {root = childe; }                  //начинаем создавать с корня
      else                                //если корень уже есть
       {MyNodeiteration = root;           //ячейка, в которой находимся
        while (true)                      //пока не дойдём до нужного места
    {if (i >iteration.data)               //если добавляемое значение больше
                                          текущего идём в право
          {if (iteration.Right == null)  //если нет правого ребёнка
            {iteration.Right = childe; //ставим на своё место
             break; }
            else
```

```
{iteration = iteration.Right; //если место нам не подошло,
                                продолжаем поиск
 continue;  } }
  else
{ if (iteration.Left == null)      //тоже самое, только если значение
                                меньше идём в левую сторону
      {iteration.Left = childe;
       break;  }
 else
  { iteration = iteration.Left;
    continue;  }  }  }  }  }
```

Метод добавления элемента полностью соответствует вышеописанному принципу добавления элемента в бинарное дерево поиска. Мы берём текущий элемент дерева и смотрим в какую сторону нам надо пойти. Дойдя до нужного места, мы добавляем туда элемент.

```
publicboolFind(int x)          //поиск элемента в дереве
{MyNode iteration = root;     //начинаем с корня
 while (iteration != null)     //если существует
  {int data = iteration.data;  //значения текущей ячейки
   if (data == x)   return true;  //если совпало, то найдено
    else if(data< x)             //если значение больше идём в право
      {iteration = iteration.Right;
       continue;   }
        else                     //если значение меньше идём в право
         {iteration = iteration.Left;
          continue;  }    }
       return false;   }          //если не нашли
```

Алгоритм поиска повторяет алгоритм добавления элемента, только вместо вставки элемента, мы возвращаем истину или лож. Это является основой дерева бинарного поиска (рис. 1). Как и говорилось ранее, дерево может уйти в одну сторону. Допустим если на вход будут поданы следующие значения: 1,4,5,6.

Рис. 1. Пример глубокого бинарного дерева поиска

Это происходит в том случае, если значения корне мало или велико, относительно поступающих элементов на добавления в дерево. В идеале нужно, чтобы дерево выглядело следующим образом (рис. 2).

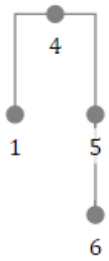

Рис. 2. Пример лучшего построение бинарного дерева

Реализованный класс содержит конструктор, который организует дерево подобным образом. Суть заключается в том, что на вход у нас подаётся сразу массив элементов, которые будут добавлены в граф. Массив сортируется. После чего берётся элемент, стоящий в середине отсортированного массива. Далее добавления делится на два этапа. Сначала бы добавим элементы больше среднего, потом меньше среднего. Добавлять мы так же будем особенным способом, а не все подряд, так как это вызовет сильный уход в глубь дерева. Мы будем добавлять парами, причём сначала большее значения в паре, потом меньшее. Таким образом мы добьёмся наиболее равномерного распределения ячеек по дереву.

```
public MyTree(int[] putArr)          //конструктор через массив
{Array.Sort(putArr);
int middle = putArr.Length / 2;
Add(putArr[middle]);
try
{for (int i = middle + 2; i<put Arr.Length; i++)
```

```
    {Add(putArr[i]);
     Add(putArr[i - 1]);
     i++;     }      }
catch
{Add(putArr[putArr.Length - 1]);  }
try
{for (int i = middle - 2; i>= 0; i--)
  {Add(putArr[i]);
    Add(putArr[i + 1]);
    i--;   }   }
catch
{Add(putArr[0]);  }  }
```

Сортировку же мы используем встроенную так, как она очень эффективная. Из минусов – мы тратим ресурсы на обработку массива, но это не является серьёзной проблемой. Так же если нам коллекция известна заранее, это станет плюсом. Далее рассмотрим результаты тестирования (табл.2).

Таблица 2

Время инициализации и поиска на 100000 элементов

100000	initialization	find
myTreeUp	07.23sec	07.29sec
myTree	22.56sec	27.34sec
ordinaryTree	62.59sec	63.07sec

В первой строчке – вариант бинарного дерева поиска через отсортированный массив, вторая строчка – классический вариант бинарного дерева поиска и третья строчка – обычное бинарное дерево. Из таблицы результатов видно, что второй метод почти в два раза быстрее обычно реализации дерева, а улучшенная версия значительно превосходит своих конкурентов (табл. 3).

Таблица 3

Занимаема память реализаций на 100000 элементов

memory	100000
myTreeUp	3999984
myTree	4000040
ordinaryTree	4800048

Из-за более рационального хранения элементов, результат такой же, как и в первой таблице. Так же значительно опережает свои конкурентов улучшенный вариант хранения бинарного дерева поиска. В заключение можно сказать, что предложенные реализации могут значительно ускорить время работы с данными темами, они могу использоваться как конкурентная альтернатива уже существующим методам и использоваться в системах выбора наилучшей альтернативы из множества существующих альтернатив [6]. Результаты исследования, изложенные в статье будут использованы в дальнейшем развитии грантовой темы, в рамках которой выполнялось задание.

Исследование проведено при финансовой поддержке РФФИ. Название проекта: «Развитие теории качественной оценки информации с учётом её структурной составляющей», № 19-47-230004, от 19.04.2019г. Все работы по составлению статьи и получению расчётных и экспериментальных данных были равномерно распределены по авторам статьи.

Список литературы

1. Brent RP, Zimmermann P. Modern Computer Arithmetic. Cambridge: Cambridge University Press; 2011.
2. Microsoft MSDN open resource.
3. Siehedazu Traversierung (mit Code be is pielen) und Ben Pfaff. An Introduction to Binary Search Trees and Balanced Trees. Free Software Foundation. Boston; 2004.
4. Popova O.B., Bogatsky N.V. Reduced Processing Time when Working with Long Arithmetic // The scientific heritage. − 2020. − №56 − C.77-99.
5. Popova O.B., Bogatsky N.V. A way to implement a binary search tree in programming language C# // The scientific heritage. − 2020. − №56 − C.100-110.
6. Popova O, Popov B, Karandey V, Gerashchenko A (2019) Entropy and Algorithm of Obtaining Decision Trees in a Way Approximated to the Natural Intelligence. International Journal of Cognitive Intelligent and Natural Intelligent. 13(3), P. 50-66.